Syntax: [IF cond] | AR / AF | = xop | + yop / + C / + yop + C / + constant / + constant + C | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example:
```
IF EQ AR = AX0 + AY0 + C;
AR = AR + 512;
```

Description: Test the optional condition and, if true, perform the specified addition. If false then perform a no-operation. Omitting the condition performs the addition unconditionally. The addition operation adds the first source operand to the second source operand along with the ALU carry bit, AC, (if designated by the "+C" notation), using binary addition. The result is stored in the destination register. The operands are contained in the data registers or constant specified in the instruction.

The *xop + constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

(instruction continues on next page)

ADD / ADD with CARRY

Instruction Format:

Conditional ALU/MAC operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15 14 13	12 11 10	9 8 7 6	5 4 3 2	1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 10010 for xop + yop + C
AMF = 10011 for xop + yop

(Note that xop + C is a special case of xop + yop + C with yop=0.)

Z: Destination register
Yop: Y operand
Xop: X operand
COND: condition

(xop + constant) Conditional ALU/MAC operation, Instruction Type 9: *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

23	22	21	20	19	18	17 16 15 14 13	12 11 10	9 8 7 6	5 4	3 2	1 0
0	0	1	0	0	Z	AMF	YY	Xop	CC	BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 10010 for xop + constant + C
AMF = 10011 for xop + constant

Z: Destination register
COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

SUBTRACT X-Y / SUBTRACT X-Y with BORROW

Syntax: [IF cond] | AR / AF | = xop | – yop / – yop + C–1 / + C–1 / – constant / – constant + C–1 | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example: IF GE AR = AX0 – AY0;

Description: Test the optional condition and, if true, then perform the specified subtraction. If the condition is not true then perform a no-operation. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand, and optionally adds the ALU Carry bit (AC) minus 1 (H#0001), and stores the result in the destination register. The (C–1) quantity effectively implements a borrow capability for multiprecision subtractions. The operands are contained in the data registers or constant specified in the instruction.

The *xop – constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	-	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.

(instruction continues on next page)

AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC operation, Instruction type 9:

23	22	21	20	19	18	17 16 15	14 13 12	11 10 9	8 7 6 5	4 3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,

AMF = 10110 for xop – yop + C – 1 operation.
AMF = 10111 for xop – yop operation.

Note that xop + C – 1 is a special case of xop – yop + C – 1 with yop=0.

Z: Destination register Yop: Y operand
Xop: X operand COND: condition

(xop – constant) Conditional ALU/MAC operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

23	22	21	20	19	18	17 16 15	14 13 12	11 10 9	8 7 6 5	4 3 2 1 0
0	0	1	0	0	Z	AMF	YY	Xop	CC BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 10110 for xop – constant + C – 1
AMF = 10111 for xop – constant

Z: Destination register COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

SUBTRACT Y-X / SUBTRACT Y-X with BORROW

Syntax: [IF cond] | AR / AF | = | yop – | xop / xop + C – 1 | | ;

		–xop + C – 1
		–xop + constant
		–xop + constant + C – 1

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example: IF GT AR = AY0 – AX0 + C – 1;

Description: Test the optional condition and, if true, then perform the specified subtraction. If the condition is not true then perform a no-operation. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand, optionally adds the ALU Carry bit (AC) minus 1 (H#0001), and stores the result in the destination register. The (C–1) quantity effectively implements a borrow capability for multiprecision subtractions. The operands are contained in the data registers or constant specified in the instruction.

The *–xop + constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.

(instruction continues on next page)

SUBTRACT Y-X / SUBTRACT Y-X with BORROW

AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17 16 15	14 13 12	11 10 9	8 7 6 5	4 3 2 1 0
0	0	1	0	0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,

AMF = 11010 for yop – xop + C – 1
AMF = 11001 for yop – xop

(Note that –xop + C – 1 is a special case of yop – xop + C – 1 with yop=0.)

Z: Destination register
Yop: Y operand
Xop: X operand
COND: condition

(–xop + constant) Conditional ALU/MAC operation, Instruction Type 9: *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

23	22	21	20	19	18	17 16 15	14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
0	0	1	0	0	Z	AMF	YY	Xop	CC	BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 11010 for constant – xop + C – 1
AMF = 11001 for constant – xop

Z: Destination register
COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

Syntax: [IF cond] | AR / AF | = xop | AND / OR / XOR | | yop / constant | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example:
```
AR = AX0  XOR  AY0;
IF FLAG_IN  AR = MR0 AND 8192;
```

Description: Test the optional condition and if true, then perform the specified bitwise logical operation (logical AND, inclusive OR, or exclusive OR). If the condition is not true then perform a no-operation. Omitting the condition performs the logical operation unconditionally. The operands are contained in the data registers or constant specified in the instruction.

The *xop AND/OR/XOR constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

(instruction continues on next page)

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					Yop		Xop			0	0	0	0	COND			

AMF specifies the ALU or MAC operation. In this case,

AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

(xop AND/OR/XOR constant)
Conditional ALU/MAC operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					YY		Xop			CC		BO	COND				

AMF specifies the ALU or MAC operation, in this case:

AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Xop: X operand
COND: condition

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

TEST BIT, SET BIT, CLEAR BIT, TOGGLE BIT
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Syntax: [IF cond] | AR / AF | = | TSTBIT n OF xop / SETBIT n OF xop / CLRBIT n OF xop / TGLBIT n OF xop | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
AX0	MR2	EQ	LE	AC
AX1	MR1	NE	NEG	NOT AC
AR	MR0	GT	POS	MV
	SR1	GE	AV	NOT MV
	SR0	LT	NOT AV	NOT CE

Permissible n values (0=LSB)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Examples:
```
AF=TSTBIT 5 OF AR;
AR=TGLBIT 13 OF AX0;
```

Description: Test the optional condition and if true, then perform the specified bit operation. If the condition is not true then perform a no-operation. Omitting the condition performs the operation unconditionally. These operations cannot be used in multifunction instructions.

These operations are defined as follows:

TSTBIT is an AND operation with a 1 in the selected bit
SETBIT is an OR operation with a 1 in the selected bit
CLRBIT is an AND operation with a 0 in the selected bit
TGLBIT is an XOR operation with a 1 in the selected bit

The ASTAT status bits are affected by these instructions. The following instructions could be used, for example, to test a bit and branch accordingly:

```
AF=TSTBIT 5 OF AR;
IF NE JUMP set;     /*Jump to "set" if bit 5 of AR is
set*/
```

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

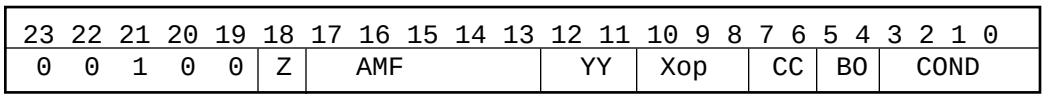
(instruction continues on next page)

TEST BIT, SET BIT, CLEAR BIT, TOGGLE BIT
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

Instruction Format:

(xop AND/OR/XOR constant)
Conditional ALU/MAC operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)



23	22	21	20	19	18	17 16 15 14	13 12	11 10	9 8 7	6 5 4	3 2 1 0
0	0	1	0	0	Z	AMF	YY	Xop	CC	BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Xop: X operand
COND: condition

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

Syntax: [IF cond] | AR / AF | = PASS | xop / yop / constant | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (all ADSP-21xx processors)
–1, 0, 1

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
2, 3, 4, 5, 7, 8, 9, 15, 16, 17, 31, 32, 33, 63, 64, 65, 127, 128, 129, 255, 256, 257, 511, 512, 513, 1023, 1024, 1025, 2047, 2048, 2049, 4095, 4096, 4097, 8191, 8192, 8193, 16383, 16384, 16385, 32766, 32767

–2, –3, –4, –5, –6, –8, –9, –10, –16, –17, –18, –32, –33, –34, –64, –65, –66, –128, –129, –130, –256, –257, –258, –512, –513, –514, –1024, –1025, –1026, –2048, –2049, –2050, –4096, –4097, –4098, –8192, –8193, –8194, –16384, –16385, –16386, –32767, –32768

Examples:
```
IF GE AR = PASS AY0;
AR = PASS 0;
AR = PASS 8191;      (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
```

Description: Test the optional condition and if true, pass the source operand unmodified through the ALU block and store in the destination register. If the condition is not true perform a no-operation. Omitting the condition performs the PASS unconditionally. The source operand is contained in the data register or constant specified in the instruction.

PASS 0 is one method of clearing AR. PASS 0 can also be combined with memory reads and writes in a multifunction instruction to clear AR.

The PASS instruction performs the transfer to the AR or AF register and affects the ASTAT status flags (for xop, yop, –1, 0, 1 only). This instruction is different from a register move operation which does not affect any status flags. The *PASS constant*

(instruction continues on next page)

operation (using any constant other than –1, 0, or 1) causes the ASTAT status flags to be undefined.

The *PASS constant* operation (using any constant other than –1, 0, or 1) is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV, AC Always cleared.

Note: The *PASS constant* operation (using any constant other than –1, 0, or 1) causes the ASTAT status flags to be undefined.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					Yop		Xop			0	0	0	0	COND			

AMF specifies the ALU or MAC operation. In this case,

AMF = 10000 for PASS yop
AMF = 10011 for PASS xop
AMF = 10001 for PASS 1
AMF = 11000 for PASS –1

Note that PASS xop is a special case of xop + yop, with yop=0.
Note that PASS 1 is a special case of yop + 1, with yop=0.
Note that PASS –1 is a special case of yop – 1, with yop=0.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

Conditional ALU/MAC operation, Instruction Type 9:
(PASS constant; constant ≠ 0,1,–1)
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					YY		Xop			CC		BO		COND			

AMF specifies the ALU or MAC operation. In this case,

AMF = 10000 for PASS yop (special case of yop, with yop=constant)
AMF = 10001 for PASS yop + 1 (special case of yop + 1, with yop=constant)
AMF = 11000 for PASS yop – 1 (special case of yop – 1, with yop=constant)

Z: Destination register COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

Syntax: [IF cond] | AR / AF | = – | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF LT AR = – AY0;

Description: Test the optional condition and if true, then NEGATE the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the NEGATE operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if operand = H#8000. Cleared otherwise.
AC Set if operand equals zero. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					Yop		Xop			0	0	0	0	COND			

AMF specifies the ALU or MAC operation. In this case,
AMF = 10101 for –yop operation.
AMF = 11001 for –xop operation

Note that –xop is a special case of yop – xop, with yop specified to be 0.

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | AR / AF | = NOT | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1	0	GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF NE AF = NOT AX0;

Description: Test the optional condition and if true, then perform the logical complement (ones complement) of the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the complement operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ	Set if the result equals zero. Cleared otherwise.
AN	Set if the result is negative. Cleared otherwise.
AV	Always cleared.
AC	Always cleared.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14	13 12	11 10	9 8 7 6	5 4 3 2	1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND	

AMF specifies the ALU or MAC operation. In this case,
AMF = 10100 for NOT yop operation.

AMF = 11011 for NOT xop operation.

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | AR / AF | = ABS xop ;

Permissible xops		*Permissible conds (see Table 15.9)*		
AX0	MR2	EQ	LE	AC
AX1	MR1	NE	NEG	NOT AC
AR	MR0	GT	POS	MV
	SR1	GE	AV	NOT MV
	SR0	LT	NOT AV	NOT CE

Example: IF NEG AF = ABS AX0 ;

Description: Test the optional condition and, if true, then take the absolute value of the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the absolute value operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	0	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if *xop* is H#8000. Cleared otherwise.
AV Set if *xop* is H#8000. Cleared otherwise.
AC Always cleared.
AS Set if the source operand is negative. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	0 0	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 11111 for ABS xop operation.

INCREMENT

Z: Destination register
Xop: X operand COND: condition

Syntax: [IF cond] | AR / AF | = yop + 1 ;

Permissible yops	*Permissible conds (see Table 15.9)*		
AY0	EQ	LE	AC
AY1	NE	NEG	NOT AC
AF	GT	POS	MV
	GE	AV	NOT MV
	LT	NOT AV	NOT CE

Example: IF GT AF = AF + 1;

Description: Test the optional condition and if true, then increment the source operand by H#0001 and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the increment operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an overflow is generated. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14 13	12 11 10	9 8 7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,

AMF = 10001 for yop + 1 operation.

Note that the xop field is ignored for the increment operation.

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | AR / AF | = yop – 1 ;

Permissible yops	*Permissible conds (see Table 15.9)*		
AY0	EQ	LE	AC
AY1	NE	NEG	NOT AC
AF	GT	POS	MV
	GE	AV	NOT MV
	LT	NOT AV	NOT CE

Example: IF EQ AR = AY1 – 1 ;

Description: Test the optional condition and if true, then decrement the source operand by H#0001 and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the decrement operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ	Set if the result equals zero. Cleared otherwise.
AN	Set if the result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14 13	12 11 10 9	8 7 6 5	4 3 2 1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,

AMF = 11000 for yop – 1 operation.

Note that the xop field is ignored for the decrement operation.

DIVIDE

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax:
```
DIVS  yop , xop ;
DIVQ  xop ;
```

Permissible xops		*Permissible yops*
AX0	MR2	AY1
AX1	MR1	AF
AR	MR0	
	SR1	
	SR0	

Description: These instructions implement **yop ÷ xop**. There are two divide primitives, DIVS and DIVQ. A single precision divide, with a 32-bit numerator and a 16-bit denominator, yielding a 16-bit quotient, executes in 16 cycles. Higher precision divides are also possible.

The division can be either signed or unsigned, but both the numerator and denominator must be the same; both signed or unsigned. The programmer sets up the divide by sorting the upper half of the numerator in any permissible *yop* (AY1 or AF), the lower half of the numerator in AY0, and the denominator in any permissible *xop*. The divide operation is then executed with the divide primitives, DIVS and DIVQ. Repeated execution of DIVQ implements a non-restoring conditional add-subtract division algorithm. At the conclusion of the divide operation the quotient will be in AY0.

To implement a signed divide, first execute the DIVS instruction once, which computes the sign of the quotient. Then execute the DIVQ instruction for as many times as there are bits remaining in the quotient (e.g., for a signed, single-precision divide, execute DIVS once and DIVQ 15 times).

To implement an unsigned divide, first place the upper half of the numerator in AF, then set the AQ bit to zero by manually clearing it in the Arithmetic Status Register, ASTAT. This indicates that the sign of the quotient is positive. Then execute the DIVQ instruction for as many times as there are bits in the quotient (e.g., for an unsigned single-precision divide, execute DIVQ 16 times).

The quotient bit generated on each execution of DIVS and DIVQ is the AQ bit which is written to the ASTAT register at the end of each cycle. The final remainder produced by this algorithm (and left over in the AF register) is not valid and must be corrected if it is needed. For more information, consult the *Division Exceptions* appendix of this manual.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	*	–	–	–	–	–

AQ Loaded with the bit value equal to the AQ bit computed on each cycle from execution of the DIVS or DIVQ instruction.

Instruction Format:

DIVQ, Instruction Type 23:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	0	0	1	0	Xop			0	0	0	0	0	0	0	0

DIVS, Instruction Type 24:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	Yop		Xop			0	0	0	0	0	0	0	0

GENERATE ALU STATUS
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Xop: X operand Yop: Y operand

Syntax: NONE = <ALU> ;

*<ALU> may be any unconditional ALU operation except DIVS or DIVQ.**

Examples:
```
NONE = AX0 – AY0;
NONE = PASS SR0;
```

Description: Perform the designated ALU operation, generate the ASTAT status flags, then discard the result value. This instruction allows the testing of register values without disturbing the contents of the AR or AF registers.

* Note that the additional-constant ALU operations of the ADSP-217x, ADSP-218x, ADSP-21msp58/59 processors are also not allowed:

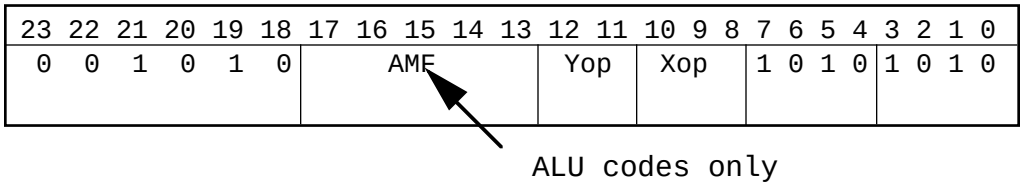
ADD (*xop + constant*)
SUBTRACT X–Y (*xop – constant*)
SUBTRACT Y–X (*–xop + constant*)
AND, OR, XOR (*xop ● constant*)
PASS (*PASS constant, using any constant other than –1, 0, or 1*)
TSTBIT, SETBIT, CLRBIT, TGLBIT.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:
ALU/MAC operation with Data Register Move, Instruction Type 8:



AMF specifies the ALU or MAC operation (only ALU operations are allowed).

Xop: X operand Yop: Y operand

Syntax: [IF cond] | MR / MF | = xop * | yop / xop | | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Examples: IF EQ MR = MX0 * MF (UU); *xop * yop*
MF = SR0 * SR0 (SS); *xop * xop*

Description: Test the optional condition and, if true, then multiply the two source operands and store in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the multiplication unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 31-16 of the product are stored in MF.

The *xop * xop* squaring operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors. Both *xops* must be the same register. This option allows single-cycle X^2 and $\sum X^2$ instructions.

The data format selection field following the two operands specifies whether each respective operand is in Signed (S) or Unsigned (U) format. The *xop* is specified first and *yop* is second. If the *xop * xop* operation is used, the data format selection field must be (UU), (SS), or (RND) only. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, rounds the result to the most significant 24 bits (or rounds bits 31-16 to 16 bits if there is no overflow from the multiply), and stores the result in the destination register. The two multiplication operands *xop* and *yop* (or *xop* and *xop*) are considered to be in twos complement format. All rounding is unbiased, except on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors, which offer a biased rounding mode. For a discussion of

(instruction continues on next page)

biased vs. unbiased rounding, see "Rounding Mode" in the "Multiplier/ Accumulator" section of Chapter 2, *Computation Units*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

*(xop * yop)* Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14 13	12 11 10	9 8 7 6	5 4 3 2	1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 0 1 0 0	xop * yop	(SS)	Signed	Signed
0 0 1 0 1	xop * yop	(SU)	Signed	Unsigned
0 0 1 1 0	xop * yop	(US)	Unsigned	Signed
0 0 1 1 1	xop * yop	(UU)	Unsigned	Unsigned
0 0 0 0 1	xop * yop	(RND)	Signed	Signed

Z: Destination register — Yop: Y operand register
Xop: X operand register — COND: condition

*(xop * xop)* Conditional ALU/MAC Operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

23 22 21 20 19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	0 0	Xop	0 0 0 1	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*
0 0 1 0 0	xop * xop	(SS)	Signed
0 0 1 1 1	xop * xop	(UU)	Unsigned
0 0 0 0 1	xop * xop	(RND)	Signed

Z: Destination register
Xop: X operand register

COND: condition

Syntax:

[IF cond] | MR / MF | = MR + xop * | yop / xop | | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Examples:

IF GE MR = MR + MX0 * MY1 (SS) ; *xop * yop*
MR = MR + MX0 * MX0 (SS); *xop * xop*

Description: Test the optional condition and, if true, then multiply the two source operands, add the product to the present contents of the MR register, and store the result in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the multiply/accumulate unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 31-16 of the 40-bit result are stored in MF.

The *xop * xop* squaring operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors. Both *xops* must be the same register. This option allows single-cycle X^2 and $\sum X^2$ instructions.

The data format selection field to the right of the two operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The *xop* is specified first and *yop* is second. If the *xop * xop* operation is used, the data format selection field must be (UU), (SS), or (RND) only. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, adds the product to the current contents of the MR register, rounds the result to the most significant 24 bits (or rounds bits 31-16 to the nearest 16 bits if there is no overflow from the multiply/accumulate), and stores the result in the destination register. The two multiplication operands *xop* and *yop* (or *xop* and *xop*) are considered to be in twos complement format. All

(instruction continues on next page)

rounding is unbiased, except on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors, which offer a biased rounding mode. For a discussion of biased vs. unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of Chapter 2, *Computation Units*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

*(xop * yop)* Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14	13 12 11	10 9 8	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 1 0 0 0	MR+xop * yop	(SS)	Signed	Signed
0 1 0 0 1	MR+xop * yop	(SU)	Signed	Unsigned
0 1 0 1 0	MR+xop * yop	(US)	Unsigned	Signed
0 1 0 1 1	MR+xop * yop	(UU)	Unsigned	Unsigned
0 0 0 1 0	MR+xop * yop	(RND)	Signed	Signed

Z: Destination register — Yop: Y operand register
Xop: X operand register — COND: condition

(*xop * xop*) Conditional ALU/MAC Operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

23 22 21 20 19	18	17 16 15 14	13 12	11 10 9	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	0 0	Xop	0 0 0 1	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*
0 1 0 0 0	MR+xop * xop	(SS)	Signed
0 1 0 1 1	MR+xop * xop	(UU)	Unsigned
0 0 0 1 0	MR+xop * xop	(RND)	Signed

Z: Destination register
Xop: X operand register
COND: condition

Syntax: [IF cond] | MR / MF | = MR – xop * | yop / xop | | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Examples:
IF LT MR = MR – MX1 * MY0 (SU) ; *xop * yop*
MR = MR – MX0 * MX0 (SS); *xop * xop*

Description: Test the optional condition and, if true, then multiply the two source operands, subtract the product from the present contents of the MR register, and store the result in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the multiply/subtract unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 16-31 of the 40-bit result are stored in MF.

The *xop * xop* squaring operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors. Both *xops* must be the same register.

The data format selection field to the right of the two operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The *xop* is specified first and *yop* is second. If the *xop * xop* operation is used, the data format selection field must be (UU), (SS), or (RND) only. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, subtracts the product from the current contents of the MR register, rounds the result to the most significant 24 bits (or rounds bits 31-16 to 16 bits if there is no overflow from the multiply/accumulate), and stores the result in the destination register. The two multiplication operands *xop* and *yop* (or *xop* and *xop*) are considered to be in twos complement format. All

(instruction continues on next page)

rounding is unbiased, except on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors, which offer a biased rounding mode. For a discussion of biased vs. unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of Chapter 2, *Computation Units*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of the upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

*(xop * yop)* Conditional ALU/MAC Operation, Instruction Type 9:

23 22 21 20 19	18	17 16 15 14 13	12 11 10 9 8	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	Yop	Xop	0 0 0 0	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 1 1 0 0	MR–xop * yop	(SS)	Signed	Signed
0 1 1 0 1	MR–xop * yop	(SU)	Signed	Unsigned
0 1 1 1 0	MR–xop * yop	(US)	Unsigned	Signed
0 1 1 1 1	MR–xop * yop	(UU)	Unsigned	Unsigned
0 0 0 1 1	MR–xop * yop	(RND)	Signed	Signed

Z: Destination register — Yop: Y operand register
Xop: X operand register — COND: condition

*(xop * xop)* Conditional ALU/MAC Operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

23 22 21 20 19	18	17 16 15 14 13	12 11	10 9 8	7 6 5 4	3 2 1 0
0 0 1 0 0	Z	AMF	0 0	Xop	0 0 0 1	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*
0 1 1 0 0	MR–xop * xop	(SS)	Signed
0 1 1 1 1	MR–xop * xop	(UU)	Unsigned
0 0 0 1 1	MR–xop * xop	(RND)	Signed

Z: Destination register COND: condition
Xop: X operand register

Syntax: [IF cond] | MR / MF | = 0 ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF GT MR = 0;

Description: Test the optional condition and, if true, then set the specified register to zero. If the condition is not true perform a no-operation. Omitting the condition performs the clear unconditionally. The entire 40-bit MR or 16-bit MF register is cleared to zero.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	0	–	–	–	–	–	–

MV Always cleared.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					1	1	0	0	0	0	0	0	0	COND			

AMF: Specifies the ALU or MAC Operation. In this case,
AMF = 00100 for clear operation.

15 MAC
TRANSFER MR

Note that this instruction is a special case of xop * yop, with yop set to zero.

Z: Destination register COND: condition

Syntax: [IF cond] | MR / MF | = MR [(RND)] ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF EQ MF = MR (RND);

Description: Test the optional condition and, if true, then perform the MR transfer according to the description below. If the condition is not true then perform a no-operation. Omitting the condition performs the transfer unconditionally.

This instruction actually performs a multiply/accumulate, specifying *yop = 0* as a multiplicand and adding the zero product to the contents of MR. The MR register may be optionally rounded at the boundary between bits 15 and 16 of the result by specifying the RND option. If MF is specified as the destination, bits 31-16 of the result are stored in MF. If MR is the destination, the entire 40-bit result is stored in MR.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					1	1	0	0	0	0	0	0	0	COND			

AMF: Specifies the ALU or MAC Operation. In this case,

AMF = 01000 for Transfer MR operation

Note that this instruction is a special case of MR + xop * yop, with yop set to zero.

Z: Destination register COND: condition
Syntax: IF MV SAT MR ;

Description: Test the MV (MAC Overflow) bit in the Arithmetic Status Register (ASTAT), and if set, then saturate the lower-order 32 bits of the 40-bit MR register; if the MV is not set then perform a no-operation.

Saturation of MR is executed with this instruction for one cycle only; MAC saturation is not a continuous mode that is enabled or disabled. The saturation instruction is intended to be used at the completion of a series of multiply/accumulate operations so that temporary overflows do not cause the accumulator to saturate.

The saturation result depends on the state of MV and on the sign of MR (the MSB of MR2). The possible results after execution of the saturation instruction are shown in the table below.

MV	*MSB of MR2*	*MR contents after saturation*
0	0	No change
0	1	No change
1	0	00000000 0111111111111111 1111111111111111
1	1	11111111 1000000000000000 0000000000000000

Status Generated: No status bits affected.

Instruction Format:
Saturate MR operation, Instruction Type 25:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ARITHMETIC SHIFT

Syntax: [IF cond] SR = [SR OR] ASHIFT xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF LT SR = SR OR ASHIFT SI (LO);

Description: Test the optional condition and, if true, then perform the designated arithmetic shift. If the condition is not true then perform a no-operation. Omitting the condition performs the shift unconditionally. The operation arithmetically shifts the bits of the operand by the amount and direction specified in the Shift Code from the SE register. Positive Shift Codes cause a left shift (upshift) and negative codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For ASHIFT with a positive Shift Code (i.e. positive value in SE), the operand is shifted left; with a negative Shift Code (i.e. negative value in SE), the operand is shifted right. The number of positions shifted is the count in the Shift Code. The 32-bit output field is sign-extended to the left (the MSB of the input is replicated to the left), and the output is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using an ASHIFT with the HI option; on the following cycle, the lower half of the number is shifted using an LSHIFT with the LO and OR options. This prevents sign bit extension of the lower word's MSB.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF *Shifter Function*
0 1 0 0 ASHIFT (HI)
0 1 0 1 ASHIFT (HI, OR)
0 1 1 0 ASHIFT (LO)
0 1 1 1 ASHIFT (LO, OR)

Xop: shifter operand COND: condition

LOGICAL SHIFT

Syntax: [IF cond] SR = [SR OR] LSHIFT xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GE SR = LSHIFT SI (HI) ;

Description: Test the optional condition and, if true, then perform the designated logical shift. If the condition is not true then perform a no-operation. Omitting the condition performs the shift unconditionally. The operation logically shifts the bits of the operand by the amount and direction specified in the Shift Code from the SE register. Positive Shift Codes cause a left shift (upshift) and negative Codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For LSHIFT with a positive Shift Code, the operand is shifted left; the numbers of positions shifted is the count in the Shift Code. The 32-bit output field is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped.

For LSHIFT with a negative Shift Code, the operand is shifted right; the number of positions shifted is the count in the Shift Code. The 32-bit output field is zero-filled from the left. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI option; on the following cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF	*Shifter Function*
0 0 0 0	LSHIFT (HI)
0 0 0 1	LSHIFT (HI, OR)
0 0 1 0	LSHIFT (LO)
0 0 1 1	LSHIFT (LO, OR)

Xop: shifter operand COND: condition

NORMALIZE

Syntax: [IF cond] SR = [SR OR] NORM xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: SR = NORM SI (HI) ;

Description: Test the optional condition and, if true, then perform the designated normalization. If the condition is not true then perform a no-operation. Omitting the condition performs the normalize unconditionally. The operation arithmetically shifts the input operand to eliminate all but one of the sign bits. The amount of the shift comes from the SE register. The SE register may be loaded with the proper Shift Code to eliminate the redundant sign bits by using the Derive Exponent instruction; the Shift Code loaded will be the negative of the quantity: (the number of sign bits minus one).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option. When the LO reference is selected, the 32-bit output field is zero-filled to the left. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped.

The 32-bit output field is zero-filled from the right. If the exponent of an overflowed ALU result was derived with the HIX modifier, the 32-bit output field is filled from left with the ALU Carry (AC) bit in the Arithmetic Status Register (ASTAT) during a NORM (HI) operation. In this case (SE=1 from the exponent detection on the overflowed ALU value) a downshift occurs.

To normalize a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI option; on the following cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF *Shifter Function*
1 0 0 0 NORM (HI)
1 0 0 1 NORM (HI, OR)
1 0 1 0 NORM (LO)
1 0 1 1 NORM (LO, OR)

Xop: shifter operand COND: condition

DERIVE EXPONENT

Syntax: [IF cond] SE = EXP xop | (HI) (LO) (HIX) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GT SE = EXP MR1 (HI) ;

Description: Test the optional condition and, if true, perform the designated exponent operation. If the condition is not true then perform a no-operation. Omitting the condition performs the exponent operation unconditionally.

The EXP operation derives the effective exponent of the input operand to prepare for the normalization operation (NORM). EXP supplies the source operand to the exponent detector, which generates a Shift Code from the number of leading sign bits in the input operand. The Shift Code, stored in SE at the completion of the EXP instruction, is the effective exponent of the input value. The Shift Code depends on which exponent detector mode is used (HI, HIX, LO).

In the HI mode, the input is interpreted as a single precision signed number, or as the upper half of a double precision signed number. The exponent detector counts the number of leading sign bits in the source operand and stores the resulting Shift Code in SE. The Shift Code will equal the negative of the number of redundant sign bits in the input.

In the HIX mode, the input is interpreted as the result of an add or subtract which may have overflowed. HIX is intended to handle shifting and normalization of results from ALU operations. The HIX mode examines the ALU Overflow bit (AV) in the Arithmetic Status Register: if AV is set, then the effective exponent of the input is +1 (indicating that an ALU overflow occurred before the EXP operation), and +1 is stored in SE. If AV is not set, then HIX performs exactly the same operations as the HI mode.

In the LO mode, the input is interpreted as the lower half of a double precision number. In performing the EXP operation on a double precision number, the higher half of the number must first be processed with EXP in the HI or HIX mode, and then the lower half can be processed with EXP in the LO mode. If the upper half contained a non-sign bit, then the correct Shift Code was generated in the HI or HIX operation and that is the code that is stored in SE. If, however, the upper half was all sign bits, then EXP in the LO mode totals the number of leading sign bits in the double precision word and stores the resulting Shift Code in SE.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	–	–	–	–	–	–	–

SS — Set by the MSB of the input for an EXP operation in the HI or HIX mode with AV = 0. Set by the MSB inverted in the HIX mode with AV = 1. Not affected by operations in the LO mode.

Instruction Format:

Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF	*Shifter Function*
1 1 0 0	EXP (HI)
1 1 0 1	EXP (HIX)
1 1 1 0	EXP (LO)

Xop: shifter operand COND: condition

BLOCK EXPONENT ADJUST

Syntax: [IF cond] SB = EXPADJ xop ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GT SB = EXPADJ SI ;

Description: Test the optional condition and, if true, perform the designated exponent operation. If the condition is not true then perform a no-operation. Omitting the condition performs the exponent operation unconditionally. The Block Exponent Adjust operation, when performed on a series of numbers, derives the effective exponent of the number largest in magnitude. This exponent can then be associated with all of the numbers in a block floating point representation.

The Block Exponent Adjust circuitry applies the input operand to the exponent detector to derive its effective exponent. The input must be a signed twos complement number. The exponent detector operates in HI mode (see the EXP instruction, above).

At the start of a block, the SB register should be initialized to –16 to set SB to its minimum value. On each execution of the EXPADJ instruction, the effective exponent of each operand is compared to the current contents of the SB register. If the new exponent is greater than the current SB value, it is written to the SB register, updating it. Therefore, at the end of the block, the SB register will contain the largest exponent found. EXPADJ is only an inspection operation; no actual shifting takes place since the true exponent is not known until all the numbers in the block have been checked. However, the numbers can be shifted at a later time after the true exponent has been derived.

Extended (overflowed) numbers and the lower halves of double precision numbers can not be processed with the Block Exponent Adjust instruction.

Status Generated: Not affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

SF = 1111.

Xop: shifter operand COND: condition

15 SHIFTER
ARITHMETIC SHIFT IMMEDIATE

Syntax: SR = [SR OR] ASHIFT xop BY <exp> | (HI) / (LO) | ;

Permissible xops		*<exp>*
SI	MR0	Any constant between –128 and 127*
SR1	MR1	
SR0	MR2	
AR		

Example: SR = SR OR ASHIFT SR0 BY 3 (LO); {do not use "+3"}

Description: Arithmetically shift the bits of the operand by the amount and direction specified by the constant in the exponent field. Positive constants cause a left shift (upshift) and negative constants cause a right shift (downshift). A positive constant must be entered **without** a "+" sign.

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For ASHIFT with a positive shift constant the operand is shifted left; with a negative shift constant the operand is shifted right. The 32-bit output field is sign-extended to the left (the MSB of the input is replicated to the left), and the output is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same shift constant is used for both halves of the number. On the first cycle, the upper half of the number is shifted using an ASHIFT with the HI option; on the following cycle, the lower half is shifted using an LSHIFT with the LO and OR options. This prevents sign bit extension of the lower word's MSB.

* See Table 2.4 in Chapter 2.

Status Generated: None affected.

Instruction Format:
Shift Immediate Operation, Instruction Type 15:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			<exp>							

SF	*Shifter Function*
0 1 0 0	ASHIFT (HI)
0 1 0 1	ASHIFT (HI, OR)
0 1 1 0	ASHIFT (LO)
0 1 1 1	ASHIFT (LO, OR)

Xop: Shifter Operand <exp>: 8-bit signed shift value

15 SHIFTER
LOGICAL SHIFT IMMEDIATE

Syntax: SR = [SR OR] LSHIFT xop BY <exp> | (HI) / (LO) | ;

Permissible xops		*<exp>*
SI	MR0	Any constant between –128 and 127*
SR1	MR1	
SR0	MR2	
AR		

Example: SR = LSHIFT SR1 BY –6 (HI) ;

Description: Logically shifts the bits of the operand by the amount and direction specified by the constant in the exponent field. Positive constants cause a left shift (upshift); negative constants cause a right shift (downshift). A positive constant must be entered **without** a "+" sign.

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the contents of the SR register by selecting the SR OR option.

For LSHIFT with a positive shift constant, the operand is shifted left. The 32-bit output field is zero-filled to the left and from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. For LSHIFT with a negative shift constant, the operand is shifted right. The 32-bit output field is zero-filled from the left and to the right. Bits shifted out of the low order bit are dropped.

To shift a double precision number, the same shift constant is used for both parts of the number. On the first cycle, the upper half of the number is shifted using the HI option; on the following cycle, the lower half is shifted using the LO and OR options.

* See Table 2.4 in Chapter 2.

Status Generated: None affected.

Instruction Format:
Shift Immediate Operation, Instruction Type 15:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			<exp>							

SF	*Shifter Function*
0 0 0 0	LSHIFT (HI)
0 0 0 1	LSHIFT (HI, OR)
0 0 1 0	LSHIFT (LO)
0 0 1 1	LSHIFT (LO, OR)

Xop: Shifter Operand

<exp>: 8-bit signed shift value

Syntax: reg = reg ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR(*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2	I0-I7	SSTAT(*read only*)	TX0
	MR1	M0-M7	IMASK	TX1
	MR0	L0-L7	ICNTL	IFC(*write only*)

Example: I7 = AR;

Description: Move the contents of the source to the destination location. The contents of the source are always right-justified in the destination location after the move.

When transferring a smaller register to a larger register (e.g., an 8-bit register to a 16-bit register), the value stored in the destination is either sign-extended to the left if the source is a signed value, or zero-filled to the left if the source is an unsigned value. The unsigned registers which (when used as the source) cause the value stored in the destination to be zero-filled to the left are: I0 through I7, L0 through L7, CNTR, PX, ASTAT, MSTAT, SSTAT, IMASK, and ICNTL. All other registers cause sign-extension to the left.

When transferring a larger register to a smaller register (e.g., a 16-bit register to a 14-bit register), the value stored in the destination is right-justified (bit 0 maps to bit 0) and the higher-order bits are dropped.

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

(instruction continues on next page)

Instruction Format:

Internal Data Move, Instruction Type 17:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	DST RGP		SRC RGP		DEST REG				SOURCE REG			

SRC RGP (Source Register Group) and SOURCE REG (Source Register) select the source register according to the Register Selection Table (see Appendix A).

DST RGP (Destination Register Group) and DEST REG (Destination Register) select the destination register according to the Register Selection Table (see Appendix A).

Syntax: reg = <data> ;
dreg = <data> ;

data: <constant>
'%' <symbol>
'^' <symbol>

Permissible registers

dregs (*Instruction Type 6*)
(*16-bit load*)

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

regs (*Instruction Type 7*)
(*maximum 14-bit load*)

SB	CNTR
PX	OWRCNTR (*write only*)
ASTAT	RX0
MSTAT	RX1
IMASK	TX0
ICNTL	TX1
I0-I7	IFC(*write only*)
M0-M7	
L0-L7	

Example: I0 = ^*data_buffer*;
L0=%*data_buffer*;

Description: Move the data value specified to the destination location. The data may be a constant, or any symbol referenced with the "length of" (%) or "pointer to" (^) operators. The data value is contained in the instruction word, with 16 bits for data register loads and up to 14 bits for other register loads. The value is always right-justified in the destination location after the load (bit 0 maps to bit 0). When a value of length less than the length of the destination is moved, it is sign-extended to the left to fill the destination width.

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

For this instruction only, the RX and TX registers may be loaded with a maximum of 14 bits of data (although the registers themselves are 16 bits wide). To load these registers with 16-bit data, use the register-to-register move instruction or the data memory-to-register move instruction with direct addressing.

Status Generated: None affected.

(instruction continues on next page)

Instruction Format :

Load Data Register Immediate, Instruction Type 6:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	DATA																DREG			

DATA contains the immediate value to be loaded into the Data Register destination location. The data is right-justified in the field, so the value loaded into an N-bit destination register is contained in the lower-order N bits of the DATA field.

DREG selects the destination Data Register for the immediate data value. One of the 16 Data Registers is selected according to the DREG Selection Table (see Appendix A).

Load Non-Data Register Immediate Instruction Type 7:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	RGP		DATA														REG			

DATA contains the immediate value to be loaded into the Non-Data Register destination location. The data is right-justified in the field, so the value loaded into an N-bit destination register is contained in the lower-order N bits of the DATA field.

RGP (Register Group) and REG (Register) select the destination register according to the Register Selection Table (see Appendix A).

DATA MEMORY READ (Direct Address)

Syntax: reg = DM (<addr>) ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR (*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2	I0-I7		TX0
	MR1	M0-M7	IMASK	TX1
	MR0	L0-L7	ICNTL	IFC(*write only*)

Example: SI = DM(*ad_port0*);

Description: The Read instruction moves the contents of the data memory location to the destination register. The addressing mode is direct addressing (designated by an immediate address value or by a label). The data memory address is stored directly in the instruction word as a full 14-bit field. The contents of the source are always right-justified in the destination register after the read (bit 0 maps to bit 0).

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

Instruction Format:
Data Memory Read (Direct Address), Instruction Type 3:

23	22	21	20	19 18	17 16 15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0
1	0	0	0	RGP	ADDR	REG

ADDR contains the direct address to the source location in Data Memory.

RGP (Register Group) and REG (Register) select the destination register according to the Register Selection Table (see Appendix A).

DATA MEMORY READ (Indirect Address)

Syntax: dreg = DM (I0 | I1 | I2 | I3 | I4 | I5 | I6 | I7 , M0 | M1 | M2 | M3 | M4 | M5 | M6 | M7) ;

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: AY0 = DM (I3, M1);

Description: The Data Memory Read Indirect instruction moves the contents of the data memory location to the destination register. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register after the read (bit 0 maps to bit 0).

Status Generated: None affected.

Instruction Format:
ALU / MAC Operation with Data Memory Read, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	0	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Read. In this case, AMF = 00000, indicating a no-operation for the ALU/MAC function.

DREG selects the destination Data Register . One of the 16 Data Registers is selected according to the DREG Selection Table (see Appendix A).

G specifies which Data Address Generator the I and M registers are selected from. These registers must be from the same DAG as separated by the gray bar above. I specifies the indirect address pointer (I register). M specifies the modify register (M register).

PROGRAM MEMORY READ (Indirect Address)

Syntax: dreg = PM (I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7) ;

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: MX1 = PM (I6, M5);

Description: The Program Memory Read Indirect instruction moves the contents of the program memory location to the destination register. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The 16 most significant bits of the Program Memory Data bus (PMD_{23-8}) are loaded into the destination register, with bit PMD_8 lining up with bit 0 of the destination register (right-justification). If the destination register is less than 16 bits wide, the most significant bits are dropped. Bits PMD_{7-0} are always loaded into the PX register. You may ignore these bits or read them out on a subsequent cycle.

Status Generated: None affected

Instruction Format:
ALU / MAC Operation with Program Memory Read, Instruction Type 5:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	AMF					0	0	0	0	0	DREG					I		M

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Read. In this case, AMF = 00000, indicating a no-operation for the ALU/MAC function.

DREG selects the destination Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

I specifies the indirect address pointer (I register). M specifies the modify register (M register).

DATA MEMORY WRITE (Direct Address)

Syntax: DM (<addr>) = reg ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	RX0
AY0	MY0	SR1	ASTAT	RX1
AY1	MY1	SR0	MSTAT	TX0
AR	MR2	I0-I7	SSTAT(*read only*)	TX1
	MR1	M0-M7	IMASK	
	MR0	L0-L7	ICNTL	

Example: DM (*cntl_port0*) = AR;

Description: Moves the contents of the source register to the data memory location specified in the instruction word. The addressing mode is direct addressing (designated by an immediate address value or by a label). The data memory address is stored directly in the instruction word as a full 14-bit field. Whenever a register less than 16 bits in length is written to memory, the value written is either sign-extended to the left if the source is a signed value, or zero-filled to the left if the source is an unsigned value. The unsigned registers which are zero-filled to the left are: I0 through I7, L0 through L7, CNTR, PX, ASTAT, MSTAT, SSTAT, IMASK, and ICNTL. All other registers are sign-extended to the left.

The contents of the source are always right-justified in the destination location after the write (bit 0 maps to bit 0).

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

Instruction Format:
Data Memory Read (Direct Address), Instruction Type 3:

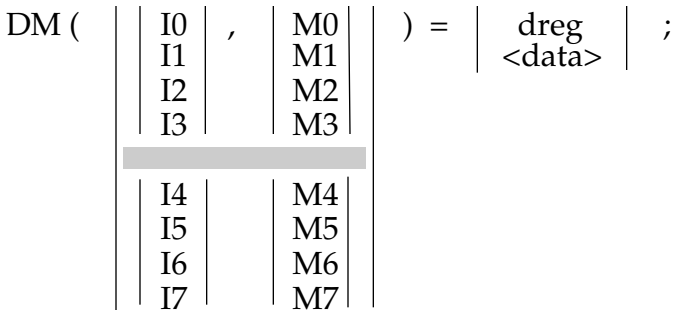
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	RGP				ADDR												REG			

ADDR contains the direct address of the destination location in Data Memory.

RGP (Register Group) and REG (Register) select the source register according to the Register Selection Table (see Appendix A).

DATA MEMORY WRITE (Indirect Address)

Syntax:



data: <constant>
'%' <symbol>
'^' <symbol>

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: DM (I2, M0) = MR1;

Description: The Data Memory Write Indirect instruction moves the contents of the source to the data memory location specified in the instruction word. The immediate data may be a constant or any symbol referenced with the "length of" (%) or "pointer to" (^) operators.

The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** When a register of less than 16 bits is written to memory, the value written is sign-extended to form a 16-bit value. The contents of the source are always right-justified in the destination location after the write (bit 0 maps to bit 0).

Status Generated: None affected.

(instruction continues on next page)

DATA MEMORY WRITE (Indirect Address)

Instruction Format:
ALU / MAC Operation with Data Memory Write, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	1	0	AMF					0	0	0	0	0	DREG				I		M	

Data Memory Write, Immediate Data, Instruction Type 2:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	G	Data																I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Write. In this case, AMF = 00000, indicating a no-operation for the ALU / MAC function.

Data represents the actual 16-bit value.

DREG selects the source Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

G specifies which Data Address Generator the I and M registers are selected from. These registers must be from the same DAG as separated by the gray bar in the Syntax description above. I specifies the indirect address pointer (I register). M specifies the modify register (M register).

PROGRAM MEMORY WRITE (Indirect Address)

Syntax: PM (I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7) = dreg ;

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Example: PM (I6, M5) = AR;

Description: The Program Memory Write Indirect instruction moves the contents of the source to the program memory location specified in the instruction word. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The 16 most significant bits of the Program Memory Data bus (PMD_{23-8}) are loaded from the source register, with bit PMD_8 aligned with bit 0 of the source register (right justification). The 8 least significant bits of the Program Memory Data bus (PMD_{7-0}) are loaded from the PX register. Whenever a source register of length less than 16 bits is written to memory, the value written is sign-extended to form a 16-bit value.

Status Generated: None affected.

Instruction Format:
ALU / MAC Operation with Program Memory Write, Instruction Type 5 (see Appendix A), as shown below:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Program Memory Write. In this case, AMF = 00000, indicating a no-operation for the ALU / MAC function.

DREG selects the source Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

I specifies the indirect address pointer (I register). M specifies the modify register (M register).

15 MOVE
I/O SPACE READ/WRITE
(ADSP-218x only)

Syntax:
IO (<addr>) = dreg ; *I/O write*
dreg = IO (<addr>) ; *I/O read*

<addr> is an 11-bit direct address value between 0 and 2047

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR1
AY1	MY1	SR0
AR	MR2	
	MR1	
	MR0	

Examples:
IO(23) = AX0;
MY1 = IO(2047);

Description: The I/O space read and write instructions are used to access the ADSP-218x's I/O memory space. These instructions move data between the processor data registers and the I/O memory space.

Status Generated: None affected.

Instruction Format:
I/O Memory Space Read/Write, Instruction Type 29:

23	22	21	20	19	18	17	16	15	14 13 12 11 10 9 8 7 6 5 4	3 2 1 0
0	0	0	0	0	0	0	1	D	ADDR	DREG

ADDR contains the 11-bit direct address of the source or destination location in I/O Memory Space.

DREG selects the Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

D specifies the direction of the transfer (0=read, 1=write).

Syntax: [IF cond] JUMP | (I4) / (I5) / (I6) / (I7) / <addr> | ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF NOT CE JUMP *top_loop*; {CNTR is decremented}

Description: Test the optional condition and, if true, perform the specified jump. If the condition is not true then perform a no-operation. Omitting the condition performs the jump unconditionally. The JUMP instruction causes program execution to continue at the effective address specified by the instruction. The addressing mode may be direct or register indirect.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field. For register indirect jumps, the selected I register provides the address; it is not post-modified in this case.

If JUMP is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. If NOT CE is used as the condition, execution of the JUMP instruction decrements the processor's counter (CNTR register).

Status Generated: None affected.

Instruction Field:
Conditional JUMP Direct Instruction Type 10:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	ADDR														COND			

Conditional JUMP Indirect Instruction Type 19:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	0	COND			

I specifies the I register (Indirect Address Pointer).

ADDR: immediate jump address COND: condition

CALL

Syntax: [IF cond] CALL | (I4) (I5) (I6) (I7) <addr> | ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF AV CALL *scale_down;*

Description: Test the optional condition and, if true, then perform the specified call. If the condition is not true then perform a no-operation. Omitting the condition performs the call unconditionally. The CALL instruction is intended for calling subroutines. CALL pushes the PC stack with the return address and causes program execution to continue at the effective address specified by the instruction. The addressing modes available for the CALL instruction are direct or register indirect.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field. For register indirect jumps, the selected I register provides the address; it is not post-modified in this case.

If CALL is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

Status Generated: None affected.

Instruction Field:
Conditional JUMP Direct Instruction Type 10:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	ADDR														COND			

Conditional JUMP Indirect Instruction Type 19:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	1	COND			

I specifies the I register (Indirect Address Pointer).

ADDR: immediate jump address COND: condition

JUMP or CALL ON FLAG IN PIN

Syntax: IF | FLAG_IN / NOT FLAG_IN | | JUMP / CALL | | <addr> | ;

Example: IF FLAG_IN JUMP *service_proc_three*;

Description: Test the condition of the FI pin of the processor and, if set to one, perform the specified jump or call. If FI is zero then perform a no-operation. Omitting the flag in condition reduces the instruction to a standard JUMP or CALL.

The JUMP instruction causes program execution to continue at the address specified by the instruction. The addressing mode for the JUMP on FI must be direct.

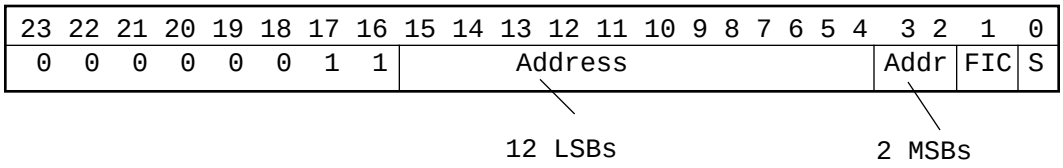
The CALL instruction is intended for calling subroutines. CALL pushes the PC stack with the return address and causes program execution to continue at the address specified by the instruction. The addressing mode for the CALL on FI must be direct.

If JUMP or CALL is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field.

Status Generated: None affected.

Instruction Field:
Conditional JUMP or CALL on Flag In Direct Instruction Type 27:



S: specifies JUMP (0) or CALL (1) FIC: latched state of FI pin

MODIFY FLAG OUT PIN

Syntax: [IF cond] | SET / RESET / TOGGLE | | FLAG_OUT / FL0 / FL1 / FL2 | [, …] ;

Example: IF MV SET FLAG_OUT, RESET FL1;

Description: Evaluate the optional condition and if true, set to one, reset to zero, or toggle the state of the specified flag output pin(s). Otherwise perform a no-operation and continue with the next instruction. Omitting the condition performs the operation unconditionally. Multiple flags may be modified by including multiple clauses, separated by commas, in a single instruction. This instruction does not directly alter the flow of your program—it is provided to signal external devices.

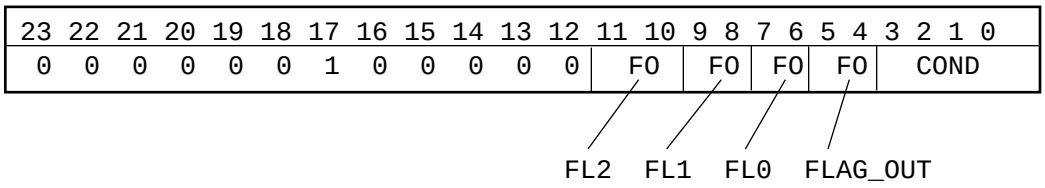
(Note that the FO pin is specified by "FLAG_OUT" in the instruction syntax.)

The following table shows which flag outputs are present on each ADSP-21xx processor:

processor	*flag pin(s)*
ADSP-2101	FO
ADSP-2105	FO
ADSP-2115	FO
ADSP-2111	FO, FL0, FL1, FL2
ADSP-217x	FO, FL0, FL1, FL2
ADSP-218x	FO, FL0, FL1, FL2
ADSP-21msp5x	FO, FL0, FL1, FL2

Status Generated: None affected.

Instruction Field:
Flag Out Mode Control Instruction Type 28:



FO: Operation to perform on flag output pin

COND: Condition code

Syntax: [IF cond] RTS ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF LE RTS ;

Description: Test the optional condition and, if true, then perform the specified return. If the condition is not true then perform a no-operation. Omitting the condition performs the return unconditionally. RTS executes a program return from a subroutine. The address on top of the PC stack is popped and is used as the return address. The PC stack is the only stack popped.

If RTS is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

Status Generated: None affected.

Instruction Field:
Conditional Return, Instruction Type 20:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	COND			

RTI

COND: condition

Syntax: [IF cond] RTI ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF MV RTI ;

Description: Test the optional condition and, if true, then perform the specified return. If the condition is not true then perform a no-operation. Omitting the condition performs the return unconditionally. RTI executes a program return from an interrupt service routine. The address on top of the PC stack is popped and is used as the return address. The value on top of the status stack is also popped, and is loaded into the arithmetic status (ASTAT), mode status (MSTAT) and the interrupt mask (IMASK) registers.

If RTI is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

Status Generated: None affected.

Instruction Field:
Conditional Return, Instruction Type 20:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	COND			

COND: condition

Syntax: DO <addr> [UNTIL term] ;

Permissible terms

EQ	NE	GT	GE	LT	FOREVER
LE	NEG	POS	AV	NOT AV	
AC	NOT AC	MV	NOT MV	CE	

Example: DO *loop_label* UNTIL CE ; {CNTR is decremented each pass through loop}

Description: DO UNTIL sets up looping circuitry for zero-overhead looping. The program loop begins at the program instruction immediately following the DO instruction, ends at the address designated in the instruction and repeats execution until the specified termination condition is met (if one is specified) or repeats in an infinite loop (if none is specified). The termination condition is tested during execution of the last instruction in the loop, the status having been generated upon completion of the previous instruction. The address (<addr>) of the last instruction in the loop is stored directly in the instruction word.

If CE is used for the termination condition, the processor's counter (CNTR register) is decremented once for each pass through the loop.

When the DO instruction is executed, the address of the last instruction is pushed onto the loop stack along with the termination condition and the current program counter value plus 1 is pushed onto the PC stack.

Any nesting of DO loops continues the process of pushing the loop and PC stacks, up to the limit of the loop stack size (4 levels of loop nesting) or of the PC stack size (16 levels for subroutines plus interrupts plus loops). With either or both the loop or PC stacks full, a further attempt to perform the DO instruction will set the appropriate stack overflow bit and will perform a no-operation.

Status Generated:

ASTAT: Not affected.

SSTAT:	7	6	5	4	3	2	1	0
	LSO	LSE	SSO	SSE	CSO	CSE	PSO	PSE
	*	0	–	–	–	–	*	0

LSO Loop Stack Overflow: set if the loop stack overflows; otherwise not affected.
LSE Loop Stack Empty: always cleared (indicating loop stack not empty)
PSO PC Stack Overflow: set if the PC stack overflows; otherwise not affected.
PSE PC Stack Empty: always cleared (indicating PC stack not empty)

(instruction continues on next page)

Instruction Format:

Do Until, Instruction Type 11:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	Addr														TERM			

ADDR specifies the address of the last instruction in the loop. In the Instruction Syntax, this field may be a program label or an immediate address value.

TERM specifies the termination condition, as shown below:

TERM	*Syntax*	*Condition Tested*
0 0 0 0	NE	Not Equal to Zero
0 0 0 1	EQ	Equal Zero
0 0 1 0	LE	Less Than or Equal to Zero
0 0 1 1	GT	Greater Than Zero
0 1 0 0	GE	Greater Than or Equal to Zero
0 1 0 1	LT	Less Than Zero
0 1 1 0	NOT AV	Not ALU Overflow
0 1 1 1	AV	ALU Overflow
1 0 0 0	NOT AC	Not ALU Carry
1 0 0 1	AC	ALU Carry
1 0 1 0	POS	X Input Sign Positive
1 0 1 1	NEG	X Input Sign Negative
1 1 0 0	NOT MV	Not MAC Overflow
1 1 0 1	MV	MAC Overflow
1 1 1 0	CE	Counter Expired
1 1 1 1	FOREVER	Always

Syntax: IDLE ;
IDLE (n); *Slow Idle*

Description: IDLE causes the processor to wait indefinitely in a low-power state, waiting for interrupts. When an interrupt occurs it is serviced and execution continues with the instruction following IDLE. Typically this next instruction will be a JUMP back to IDLE, implementing a low-power standby loop. (Note the restrictions on JUMP or IDLE as the last instruction in a DO UNTIL loop, detailed in Chapter 3.)

IDLE (n) is a special version of IDLE that slows the processor's internal clock signal to further reduce power consumption. The reduced clock frequency, a programmable fraction of the normal clock rate, is specified by a selectable divisor *n* given in the instruction: *n* = 16, 32, 64, or 128. The instruction leaves the processor fully functional, but operating at the slower rate during execution of the IDLE (n) instruction. While it is in this state, the processor's other internal clock signals (such as SCLK, CLKOUT, and the timer clock) are reduced by the same ratio.

When the IDLE (n) instruction is used, it slows the processor's internal clock and thus its response time to incoming interrupts—the 1-cycle response time of the standard IDLE state is increased by *n*, the clock divisor. When an enabled interrupt is received, the ADSP-21xx will remain in the IDLE state for up to a maximum of *n* CLKIN cycles (where *n* =16, 32, 64, or 128) before resuming normal operation.

When the IDLE (n) instruction is used in systems that have an externally generated serial clock, the serial clock rate may be faster than the processor's reduced internal clock rate. Under these conditions, interrupts must not be generated at a faster rate than can be serviced, due to the additional time the processor takes to come out of the IDLE state (a maximum of *n* CLKIN cycles).

Serial port autobuffering continues during IDLE without affecting the idle state.

Status Generated: None affected.

Instruction Field:
Idle, Instruction Type 31:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Slow Idle, Instruction Type 31:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	DV			

15 MISC
STACK CONTROL

Syntax: [PUSH | POP STS] [, POP CNTR] [, POP PC] [, POP LOOP] ;

Example: POP CNTR, POP PC, POP LOOP;

Description: Stack Control pushes or pops the designated stack(s). The entire instruction executes in one cycle regardless of how many stacks are specified.

The PUSH STS (Push Status Stack) instruction increments the status stack pointer by one to point to the next available status stack location; and pushes the arithmetic status (ASTAT), mode status (MSTAT), and interrupt mask register (IMASK) onto the processor's status stack. Note that the PUSH STS operation is executed automatically whenever an interrupt service routine is entered.

Any POP pops the value on the top of the designated stack and decrements the same stack pointer to point to the next lowest location in the stack. POP STS causes the arithmetic status (ASTAT), mode status (MSTAT), and interrupt mask (IMASK) to be popped into these same registers. This also happens automatically whenever a return from interrupt (RTI) is executed.

POP CNTR causes the counter stack to be popped into the down counter. When the loop stack or PC stack is popped (with POP LOOP or POP PC, respectively), the information is lost. Returning from an interrupt (RTI) or subroutine (RTS) also pops the PC stack automatically.

Status Generated:

SSTAT:	7	6	5	4	3	2	1	0
	LSO	LSE	SSO	SSE	CSO	CSE	PSO	PSE
	–	*	*	*	–	*	–	*

PSE PC Stack Empty: set if a pop results in an empty program counter stack; cleared otherwise.
CSE Counter Stack Empty: set if a pop results in an empty counter stack; cleared otherwise.
SSE Status Stack Empty: for PUSH STS, this bit is always cleared (indicating status stack not empty). For POP STS, SSE is set if the pop results in an empty status stack; cleared otherwise.
SSO Status Stack Overflow: for PUSH STS set if the status stack overflows; otherwise not affected.
LSE Loop Stack Empty: set if a pop results in an empty loop stack; cleared otherwise.

Note that once any Stack Overflow occurs, the corresponding stack overflow bit is set in SSTAT, and this bit stays set indicating there has been loss of information. Once set, the stack overflow bit can only be cleared by resetting the processor.

Instruction Format:

Stack Control, Instruction Type 26:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Pp		Lp	Cp	Spp

Pp: PC Stack Control
Lp: Loop Stack Control
Cp: Counter Stack Control
Spp: Status Stack Control

TOPPCSTACK

A special version of the Register-to-Register Move instruction, Type 17, is provided for reading (and popping) or writing (and pushing) the top value of the PC stack. The normal POP PC instruction does not save the value popped from the stack, so to save this value into a register you must use the following special instruction:

```
reg = TOPPCSTACK;       {pop PC stack into reg}
                        {"toppcstack" may also be lowercase}
```

The PC stack is also popped by this instruction, after a one-cycle delay. A NOP should usually be placed after the special instruction, to allow the pop to occur properly:

```
reg = TOPPCSTACK;
NOP;                    {allow pop to occur correctly}
```

There is no standard PUSH PC stack instruction. To push a specific value onto the PC stack, therefore, use the following special instruction:

```
TOPPCSTACK = reg;       {push reg contents onto PC stack}
```

The stack is pushed immediately, in the same cycle.

Note that "TOPPCSTACK" may not be used as a register in any other instruction type!

Examples:

```
AX0 = TOPPCSTACK;       {pop PC stack into AX0}
NOP;

TOPPCSTACK = I7;        {push contents of I7 onto PC stack}
```

Only the following registers may be used in the special TOPPCSTACK instructions:

ALU, MAC, & Shifter Registers			*DAG Registers*					
AX0	AR	SI	I0	I4	M0	M4	L0	L4
AX1	MR0	SE	I1	I5	M1	M5	L1	L5
MX0	MR1	SR0	I2	I6	M2	M6	L2	L6
MX1	MR	SR1	I3	I7	M3	M7	L3	L7
AY0								
AY1								
MY0								
MY1								

There are several restrictions on the use of the special TOPPCSTACK instructions; they are described in Chapter 3, Program Control.

(instruction continues on next page)

Instruction Format:

TOPPCSTACK=reg
Internal Data Move, Instruction Type 17:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	1	1	SRC RGP		1	1	1	1	SOURCE REG			

SRC RGP (Source Register Group) and SOURCE REG (Source Register) select the source register according to the Register Selection Table (see Appendix A).

reg=TOPPCSTACK
Internal Data Move, Instruction Type 17:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	DST RGP		1	1	DEST REG				1	1	1	1

DST RGP (Destination Register Group) and DEST REG (Destination Register) select the destination register according to the Register Selection Table (see Appendix A).

Syntax:

ENA DIS		BIT_REV AV_LATCH AR_SAT SEC_REG G_MODE M_MODE TIMER	[, …] ;

Example: DIS AR_SAT, ENA M_MODE;

Description: Enables (ENA) or disables (DIS) the designated processor mode. The corresponding mode status bit in the mode status register (MSTAT) is set for ENA mode and cleared for DIS mode. At reset, MSTAT is set to zero, meaning that all modes are disabled. Any number of modes can be changed in one cycle with this instruction. Multiple ENA or DIS clauses must be separated by commas.

MSTAT Bits:

0	SEC_REG	Alternate Register Data Bank
1	BIT_REV	Bit-Reverse Mode on Address Generator #1
2	AV_LATCH	ALU Overflow Status Latch Mode
3	AR_SAT	ALU AR Register Saturation Mode
4	M_MODE	MAC Result Placement Mode
5	TIMER	Timer Enable
6	G_MODE	Enables GO Mode

The data register bank select bit (SEC_REG) determines which set of data registers is currently active (0=primary, 1=secondary).

The bit-reverse mode bit (BIT_REV), when set to 1, causes addresses generated by Data Address Generator #1 to be output in bit reversed order.

The ALU overflow latch mode bit (AV_LATCH), when set to 1, causes the AV bit in the arithmetic status register to stay set once an ALU overflow occurs. In this mode, if an ALU overflow occurs, the AV bit will be set and will remain set even if subsequent ALU operations do not generate overflows. The AV bit can only be cleared by writing a zero into it directly over the DMD bus.

(instruction continues on next page)

The AR saturation mode bit, (AR_SAT), when set to 1, causes the AR register to saturate if an ALU operation causes an overflow, as described in Chapter 2, "Computation Units."

The MAC result placement mode (M_MODE) determines whether or not the left shift is made between the multiplier product and the MR register.

Setting the Timer Enable bit (TIMER) starts the timer decrementing logic. Clearing it halts the timer.

The GO mode (G_MODE) allows an ADSP-21xx processor to continue executing instructions from internal memory (if possible) during a bus grant. The GO mode allows the processor to run; only if an external memory access is required does the processor halt, waiting for the bus to be released.

Instruction Format:

Mode Control, Instruction Type 18:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	TI			MM			AS		OL		BR		SR		GM		0	0

TI: Timer Enable
AS: AR Saturation Mode Control
BR: Bit Reverse Mode Control
GM: GO Mode
MM: Multiplier Placement
OL: ALU Overflow Latch Mode Control
SR: Secondary Register Bank Mode

Syntax: MODIFY (I0 | I1 | I2 | I3 , M0 | M1 | M2 | M3) ;

I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7

Example: MODIFY (I1, M1);

Description: Add the selected M register (M_n) to the selected I register (I_m), then process the modified address through the modulus logic with buffer length as determined by the L register corresponding to the selected I register (L_m), and store the resulting address pointer calculation in the selected I register. The I register is modified as if an indexed memory address were taking place, but no actual memory data transfer occurs. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.**

The selection of the I and M registers is constrained to registers within the same Data Address Generator: selection of I0-I3 in Data Address Generator #1 constrains selection of the M registers to M0-M3. Similarly, selection of I4-I7 constrains the M registers to M4-M7.

Status Generated: None affected.

Instruction Format:
Modify Address Register, Instruction Type 21:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	G	I		M	

G specifies which Data Address Generator is selected. The I and M registers specified must be from the same DAG, separated by the gray bar above. I specifies the I register (depends on which DAG is selected by the G bit). M specifies the M register (depends on which DAG is selected by the G bit).

NOP

Syntax: NOP ;

Description: No operation occurs for one cycle. Execution continues with the instruction following the NOP instruction.

Status Generated: None affected.

Instruction Format:
No operation, Instruction Type 30 (see Appendix A), as shown below:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

INTERRUPT ENABLE & DISABLE
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Syntax:
```
ENA INTS ;
DIS INTS ;
```

Description: Interrupts are enabled by default at reset. Executing the DIS INTS instruction causes all interrupts (including the powerdown interrupt) to be masked, without changing the contents of the IMASK register.

Executing the ENA INTS instruction allows all unmasked interrupts to be serviced again.

Note: Disabling interrupts does not affect serial port autobuffering or ADSP-218x DMA transfers (IDMA or BDMA). These operations will continue normally whether or not interrupts are enabled.

Status Generated: None affected.

Instruction Format:
DIS INTS, Instruction Type 26:

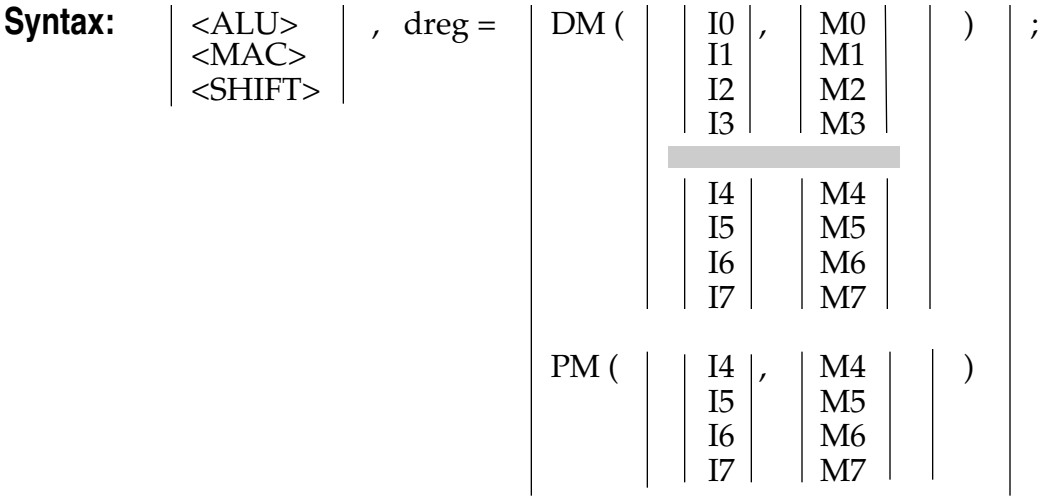
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

ENA INTS, Instruction Type 26:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

15 MULTIFUNCTION COMPUTATION with MEMORY READ

Syntax:



Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR0
AY1	MY1	SR1
AR	MR0	
	MR1	
	MR2	

Description: Perform the designated arithmetic operation and data transfer. The read operation moves the contents of the source to the destination register. The addressing mode when combining an arithmetic operation with a memory read is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, memory read second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (using the –s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as source in one clause and a destination in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

(1) AR = AX0 + AY0, AX0 = DM (I0, M0);

is a legal version of this multifunction instruction and is not flagged by the assembler. Reversing the order of clauses, as in

(2) AX0 = DM (I0, M0) , AR = AX0 + AY0;

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the data memory value is loaded into AX0 and then used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Using the same register as a destination in both clauses, however, produces an indeterminate result and should not be done. The assembler issues a warning unless semantics checking is turned off. Regardless of whether or not the warning is produced, however, this practice is not supported.

The following, therefore, is illegal and not supported, even though assembler semantics checking produces only a warning:

(3) AR = AX0 + AY0, AR = DM (I0, M0); *Illegal!*

(instruction continues on next page)

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV	Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS	Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Memory Read, Instruction Type 4:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	0	Z	AMF					Yop			Xop			Dreg			I		M	

ALU/MAC operation with Program Memory Read, Instruction Type 5:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	Z	AMF					Yop			Xop			Dreg			I		M	

Shift operation with Data Memory Read, Instruction Type 12:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	G	0			SF			Xop			Dreg			I		M	

Shift operation with Program Memory Read, Instruction Type 13:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	0			SF			Xop			Dreg			I		M	

Z:	Result register	Dreg:	Destination register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand
G:	Data Address Generator	I:	Indirect address register
M:	Modify register		

15 MULTIFUNCTION COMPUTATION with REGISTER to REGISTER MOVE

Syntax:

<ALU> <MAC> <SHIFT>	, dreg = dreg ;	

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR0
AY1	MY1	SR1
AR	MR0	
	MR1	
	MR2	

Description: Perform the designated arithmetic operation and data transfer. The contents of the source are always right-justified in the destination register after the read.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, register transfer second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (–s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as source in one clause and a destination in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

(1) AR = AX0 + AY0, AX0 = MR1;

is a legal version of this multifunction instruction and is not flagged by the assembler. Reversing the order of clauses, as in

(2) AX0 = MR1, AR = AX0 + AY0;

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the MR1 register value is loaded into AX0 and then AX0 is used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Using the same register as a destination in both clauses, however, produces an indeterminate result and should not be done. The assembler issues a warning unless semantics checking is turned off. Regardless of whether or not the warning is produced, however, this practice is not supported.

The following, therefore, is illegal and not supported, even though assembler semantics checking produces only a warning:

(3) AR = AX0 + AY0, AR = MR1; *Illegal!*

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

(instruction continues on next page)

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV — Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS — Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Register Move, Instruction Type 8:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Z	AMF							Yop			Xop			Dreg dest			Dreg source	

Shift operation with Data Register Move, Instruction Type 14:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	SF					Xop			Dreg dest			Dreg source			

Z:	Result register	Dreg:	Data register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand

Syntax:

```
DM ( I0 | I1 | I2 | I3 , M0 | M1 | M2 | M3 ) = dreg , <ALU> | <MAC> | <SHIFT> ;
                       I4 | I5 | I6 | I7   M4 | M5 | M6 | M7

PM ( I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7 )
```

Permissible dregs

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SR0
AY1	MY1	SR1
AR	MR0	
	MR1	
	MR2	

Description: Perform the designated arithmetic operation and data transfer. The write operation moves the contents of the source to the specified memory location. The addressing mode when combining an arithmetic operation with a memory write is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (memory write first, computation second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (–s switch) the warning is not issued.

(instruction continues on next page)

Because of the read-first, write-second characteristic of the processor, using the same register as destination in one clause and a source in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

(1) DM (I0, M0) = AR, AR = AX0 + AY0;

is a legal version of this multifunction instruction and is not flagged by the assembler. Reversing the order of clauses, as in

(2) AR = AX0 + AY0, DM (I0, M0) = AR;

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the result of the computation in AR is then written to memory, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV — Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS — Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Memory Write, Instruction Type 4:

23	22	21	20	19	18	17 16 15 14 13	12 11 10	9 8 7	6 5 4	3	2 1 0
0	1	1	G	1	Z	AMF	Yop	Xop	Dreg	I	M

ALU/MAC operation with Program Memory Write, Instruction Type 5:

23	22	21	20	19	18	17 16 15 14 13	12 11 10	9 8 7	6 5 4	3	2 1 0
0	1	0	1	1	Z	AMF	Yop	Xop	Dreg	I	M

(instruction continues on next page)

Shift operation with Data Memory Write, Instruction Type 12:

23	22	21	20	19	18	17	16	15	14 13 12	11 10 9	8 7 6 5	4	3 2 1 0
0	0	0	1	0	0	1	G	1	SF	Xop	Dreg	I	M

Shift operation with Program Memory Write, Instruction Type 13:

23	22	21	20	19	18	17	16	15	14 13 12	11 10 9	8 7 6 5	4	3 2 1 0
0	0	0	1	0	0	0	1	1	SF	Xop	Dreg	I	M

Z: Result register
SF: Shifter operation
Yop: Y operand
I: Indirect address register
G: Data Address Generator; I & M registers must be from the same DAG, as separated by the gray bar in the Syntax description.

Dreg: Destination register
AMF: ALU/MAC operation
Xop: X operand
M: Modify register

Syntax:

AX0 AX1 MX0 MX1	= DM (	I0 I1 I2 I3	,	M0 M1 M2 M3	),	AY0 AY1 MY0 MY1	= PM (	I4 I5 I6 I7	,	M4 M5 M6 M7	);

Description: Perform the designated memory reads, one from data memory and one from program memory. Each read operation moves the contents of the memory location to the destination register. For this double data fetch, the destinations for data memory reads are the X registers in the ALU and the MAC, and the destinations for program memory reads are the Y registers. The addressing mode for this memory read is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register.

A multifunction instruction requires three items to be fetched from memory: the instruction itself and two data words. No extra cycle is needed to execute the instruction as long as only one of the fetches is from external memory.

If two off-chip accesses are required, however—the instruction fetch and one data fetch, for example, or data fetches from both program and data memory—then one overhead cycle occurs. In this case the program memory access occurs first, then the data memory access. If three off-chip accesses are required—the instruction fetch as well as data fetches from both program and data memory—then two overhead cycles occur.

Status Generated: No status bits are affected.

Instruction Format:

ALU/MAC with Data & Program Memory Read, Instruction Type 1:

23 22	21 20	19 18	17 16 15 14 13	12 11 10 9 8	7 6	5 4	3 2	1 0
1 1	PD	DD	AMF	0 0 0 0 0	PM I	PM M	DM I	DM M

AMF specifies the ALU or MAC function. In this case, AMF = 00000, designating a no-operation for the ALU or MAC function.

PD: Program Destination register
DD: Data Destination register
AMF: ALU/MAC operation
I: Indirect address register
M: Modify register

ALU / MAC with DATA & PROGRAM MEMORY READ

Syntax:

<ALU> <MAC>	,	AX0 AX1 MX0 MX1	= DM (	I0 I1 I2 I3	,	M0 M1 M2 M3	) ,	AY0 AY1 MY0 MY1	= PM (	I4 I5 I6 I7	,	M4 M5 M6 M7	) ;											

Description: This instruction combines an ALU or a MAC operation with a data memory read and a program memory read. The read operations move the contents of the memory location to the destination register. For this double data fetch, the destinations for data memory reads are the X registers in the ALU and the MAC, and the destinations for program memory reads are the Y registers. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register after the read.

A multifunction instruction requires three items to be fetched from memory: the instruction itself and two data words. No extra cycle is needed to execute the instruction as long as only one of the fetches is from external memory.

If two off-chip accesses are required, however—the instruction fetch and one data fetch, for example, or data fetches from both program and data memory—then one overhead cycle occurs. In this case the program memory access occurs first, then the data memory access. If three off-chip accesses are required—the instruction fetch as well as data fetches from both program and data memory—then two overhead cycles occur.

The computation must be unconditional. All ALU and MAC operations are permitted except the DIVS and DIVQ instructions. The results of the computation must be written into the R register of the computational unit; ALU results to AR, MAC results to MR.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, memory reads second) is intended to imply this. In fact, you may code this instruction with the order of clauses altered. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (–s switch) the warning is not issued.

The same data register may be used as a source for the arithmetic operation and as a destination for the memory read. The register supplies the value present at the beginning of the cycle and is written with the value from memory at the end of the cycle.

For example,

(1) MR=MR+MX0*MY0(UU), MX0=DM(I0, M0), MY0=PM(I4,M4);

is a legal version of this multifunction instruction and is not flagged by the assembler. Changing the order of clauses, as in

(2) MX0=DM(I0, M0), MY0=PM(I4,M4), MR=MR+MX0*MY0(UU);

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the data memory value is loaded into MX0 and MY0 and subsequently used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Status Generated: All status bits are affected in the same way as for the single operation version of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

(instruction continues on next page)

ALU / MAC with DATA & PROGRAM MEMORY READ

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV Set if the accumulated product overflows the lower-order 32-bits of the MR register. Cleared otherwise.

Instruction Format:

ALU/MAC with Data and Program Memory Read, Instruction Type 1:

23 22	21 20	19 18	17 16 15 14	13 12	11 10 9 8	7 6 5 4	3 2 1 0		
1 1	PD	DD	AMF	Yop	Xop	PMI	PMM	DMI	DMM

PD: Program Destination register
DD: Data Destination register
AMF: ALU/MAC operation
M: Modify register
Yop: Y operand
Xop: X operand
I: Indirect address register