Multifunction Instructions

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

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

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

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

```
| <ALU>*   |  , dreg   =   dreg;
| <MAC>*   |
| <SHIFT>* |
```

Table 15.2 Multifunction Instructions

<ALU>	Any ALU instruction *(except DIVS, DIVQ)*
<MAC>	Any multiply/accumulate instruction
<SHIFT>	Any shifter instruction *(except Shift Immediate)*

* May not be conditional instruction

† AR, MR result registers must be used—not AF, MF feedback registers. (See Section 15.4.1, “ALU/MAC with Data & Program Memory Read.”)

15.5 ALU, MAC & SHIFTER INSTRUCTIONS

This group of instructions performs computations. All of these instructions can be executed conditionally except the ALU division instructions and the Shifter SHIFT IMMEDIATE instructions.

15.5.1 ALU Group

Here is an example of one ALU instruction, Add/Add with Carry:

```
IF AC AR=AX0+AY0+C;
```

The (optional) conditional expression, IF AC, tests the ALU Carry bit (AC); if there is a carry from the previous instruction, this instruction executes, otherwise a NOP occurs and execution continues with the next instruction. The algebraic expression AR=AX0+AY0+C means that the ALU result register (AR) gets the value of the ALU X input and Y input registers plus the value of the carry-in bit.

Table 15.3 gives a summary list of all ALU instructions. In this list, *condition* stands for all the possible conditions that can be tested and *xop* and *yop* stand for the registers that can be specified as input for the ALU. The conditional clause is optional and is enclosed in square brackets to show this. A complete list of the permissible *xops* and *yops* is given in the reference page for each instruction. A complete list of conditions is given in Table 15.9.

ALU Instructions

```
[IF condition]   | AR |  =   xop    | + yop            |    ;
                 | AF |             | + C              |
                 |    |             | + yop + C        |
                                    | + constant       |
                                    | + constant + C   |

[IF condition]   | AR |  =   xop    | – yop                |  ;
                 | AF |             | – yop + C – 1        |
                 |    |             | + C – 1              |
                                    | – constant           |
                                    | – constant + C – 1   |

[IF condition]   | AR |  = | yop    | – xop            | |    ;
                 | AF |    |        | – xop + C – 1    | |
                           | – xop + C – 1               |
                           | – xop + constant            |
                           | – xop + constant + C – 1    |
```

[IF condition]	AR	=	xop	AND	yop	;
	AF			OR	constant	
				XOR		
[IF condition]	AR	=	TSTBIT n OF xop			;
	AF		SETBIT n OF xop			
			CLRBIT n OF xop			
			TGLBIT n OF xop			
[IF condition]	AR	=	PASS	xop		;
	AF			yop		
				constant		
[IF condition]	AR	=	–	xop		;
	AF			yop		
[IF condition]	AR	=	NOT	xop		;
	AF			yop		
[IF condition]	AR	=	ABS	xop		;
	AF					
[IF condition]	AR	=	yop	+ 1		;
	AF					
[IF condition]	AR	=	yop	– 1		;
	AF					

DIVS yop, xop ;
DIVQ xop ;

NONE = <ALU> ;

Table 15.3 ALU Instructions

15.5.2 MAC Group

Here is an example of one of the MAC instructions, Multiply/Accumulate:

```
IF NOT MV MR=MR+MX0*MY0(UU);
```

The conditional expression, IF NOT MV, tests the MAC overflow bit. If the condition is not true, a NOP is executed. The expression MR=MR+MX0*MY0 is the multiply/accumulate operation: the multiplier result register (MR) gets the value of itself plus the product of the X and Y input registers selected. The modifier in parentheses (UU) treats the operands as unsigned. There can be only one such modifier selected from the available set. (SS) means both are signed, while (US) and (SU) mean that either the first or second operand is signed; (RND) means to round the (implicitly signed) result.

Table 15.4 gives a summary list of all MAC instructions. In this list, *condition* stands for all the possible conditions that can be tested and *xop* and *yop* stand for the registers that can be specified as input for the MAC. A complete list of the permissible *xops* and *yops* is given in the reference page for each instruction.
MAC Instructions

```
[IF condition] | MR | = xop      * | yop | ( | SS  | );
               | MF |              | xop |   | SU  |
                                             | US  |
                                             | UU  |
                                             | RND |

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

[IF condition] | MR | = MR – xop * | yop | ( | SS  | );
               | MF |              | xop |   | SU  |
                                             | US  |
                                             | UU  |
                                             | RND |

[IF condition] | MR | = 0;
               | MF |

[IF condition] | MR | = MR [( RND )];
               | MF |

IF MV SAT MR ;
```

Table 15.4 MAC Instructions

15.5.3 Shifter Group

Here is an example of one of the Shifter instructions, Normalize:

```
IF NOT CE SR= SR OR NORM SI (HI);
```

The conditional expression, IF NOT CE, tests the "not counter expired" condition. If the condition is false, a NOP is executed. The destination of all shifting operations is the Shifter Result register, SR. (The destination of exponent detection instructions is SE or SB, as shown below.) In this example, SI, the Shifter Input register, is the operand. The amount and direction of the shift is controlled by the signed value in the SE register in all shift operations except an immediate shift. Positive values cause left shifts; negative values cause right shifts.

The “SR OR” modifier (which is optional) logically ORs the result with the current contents of the SR register; this allows you to construct a 32-bit value in SR from two 16-bit pieces. “NORM” is the operator and “(HI)” is the modifier that determines whether the shift is relative to the HI or LO (16-bit) half of SR. If “SR OR” is omitted, the result is passed directly into SR.

Table 15.5 gives a summary list of all Shifter instructions. In this list, *condition* stands for all the possible conditions that can be tested.

Shifter Instructions

```
[IF condition]   SR   =   [SR OR] ASHIFT  xop        (  | HI  |  );
                                                        | LO  |

[IF condition]   SR   =   [SR OR] LSHIFT  xop        (  | HI  |  );
                                                        | LO  |

[IF condition]   SR   =   [SR OR] NORM  xop          (  | HI  |  );
                                                        | LO  |

[IF condition]   SE   =   EXP  xop                   (  | HI  |  );
                                                        | LO  |
                                                        | HIX |

[IF condition]   SB   =   EXPADJ  xop;

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

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

Table 15.5 Shifter Instructions

15.6 MOVE: READ & WRITE

MOVE instructions, shown in Table 15.6, move data to and from data registers and external memory. Registers are divided into two groups, referred to as *reg* which includes almost all registers and *dreg*, or data registers, which is a subset. Only the program counter (PC) and the ALU and MAC feedback registers (AF and MF) are not accessible.

Table 15.7 shows which registers belong to these groups. Many of the system control registers are memory-mapped (for the processors with on-chip memory); these registers are read and written as memory locations instead of with register names.

MOVE Instructions

```
reg          =      reg ;

reg          =      DM (<address>) ;

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

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

DM (<address>)  =       reg;

reg          =      <data>  ;

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

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

Table 15.6 MOVE Instructions

Registers: ***reg***	*Data Registers:* ***dreg***
SB PX I0 – I7, M0 – M7, L0 – L7 CNTR ASTAT, MSTAT, SSTAT IMASK, ICNTL, IFC TX0, TX1, RX0, RX1	AX0, AX1, AY0, AY1, AR MX0, MX1, MY0, MY1, MR0, MR1, MR2 SI, SE, SR0, SR1

Table 15.7 Processor Registers: reg & dreg

15.7 PROGRAM FLOW CONTROL

Program flow control on the ADSP-2100 family processors is simple but powerful. Here is an example of one instruction:

```
IF EQ JUMP my_label;
```

JUMP, of course, is a familiar construct from many other languages. *My_label* is any identifier you wish to use as a label for the destination jumped to. Instead of the label, an index register in DAG2 may be explicitly used. The default scope for any label is the source code module in which it is declared. The assembler directive .ENTRY makes a label visible as an entry point for routines outside the module. Conversely, the .EXTERNAL directive makes it possible to use a label declared in another module.

If the counter condition (CE, NOT CE) is to be used, an assignment to CNTR must be executed to initialize the counter value. JUMP and CALL permit the additional conditionals "FLAG_IN" and "NOT FLAG_IN" to be used for branching on the state of the FI pin, but only with direct addressing, not with DAG2 as the address source.

RTS (return from subroutine) and RTI (return from interrupt) provide for conditional return from CALL or interrupt vectors respectively.

The IDLE instruction provides a way to wait for interrupts. IDLE causes the processor to wait in a low-power state until an interrupt occurs. When an interrupt is serviced, control returns to the instruction following the IDLE statement. IDLE uses less power than loops created with JUMP.

Table 15.8 gives a summary of all program flow control instructions. The *condition* codes are described in Table 15.9.

Program Flow Control Instructions

```
[IF condition]     JUMP    |    (I4)      |   ;
                           |    (I5)      |
                           |    (I6)      |
                           |    (I7)      |
                           | <address>    |

IF | FLAG_IN     |         JUMP              <address> ;
   | NOT FLAG_IN |

[IF condition]     CALL    |    (I4)      |   ;
                           |    (I5)      |
                           |    (I6)      |
                           |    (I7)      |
                           | <address>    |

IF | FLAG_IN     |         CALL              <address> ;
   | NOT FLAG_IN |

[IF condition]     RTS ;

[IF condition]     RTI ;

DO <address>  [UNTIL termination] ;

IDLE [(n)];
```

Table 15.8 Program Flow Control Instructions

Syntax	*Status Condition*	*True If:*
EQ	Equal Zero	AZ = 1
NE	Not Equal Zero	AZ = 0
LT	Less Than Zero	AN .XOR. AV = 1
GE	Greater Than or Equal Zero	AN .XOR. AV = 0
LE	Less Than or Equal Zero	(AN .XOR. AV) .OR. AZ = 1
GT	Greater Than Zero	(AN .XOR. AV) .OR. AZ = 0
AC	ALU Carry	AC = 1
NOT AC	Not ALU Carry	AC = 0
AV	ALU Overflow	AV = 1
NOT AV	Not ALU Overflow	AV = 0
MV	MAC Overflow	MV = 1
NOT MV	Not MAC Overflow	MV = 0
NEG	X Input Sign Negative	AS = 1
POS	X Input Sign Positive	AS = 0
NOT CE	Not Counter Expired	
FLAG_IN*	FI pin	Last sample of FI pin = 1
NOT FLAG_IN*	Not FI pin	Last sample of FI pin = 0

Table 15.9 IF Condition Codes

* Only available on JUMP and CALL instructions

15.8 MISCELLANEOUS INSTRUCTIONS

There are several miscellaneous instructions. NOP is a no operation instruction. The PUSH/POP instructions allows you to explicitly control the status, counter, PC and loop stacks; interrupt servicing automatically pushes and pops some of these stacks.

The Mode Control instruction enables and disables processor modes of operation: bit-reversal on DAG1, latching ALU overflow, saturating the ALU result register, choosing the primary or secondary register set, GO mode for continued operation during bus grant, multiplier shift mode for fractional or integer arithmetic, and timer enabling.

A single ENA or DIS can be followed by any number of mode identifiers, separated by commas; ENA and DIS can also be repeated. All seven modes can be enabled, disabled, or changed in a single instruction.

The MODIFY instruction modifies the address pointer in the I register selected with the value in the selected M register, without performing any actual memory access. As always, the I and M registers must be from the same DAG; any of I0-I3 may be used only with one from M0-M3 and the same for I4-I7 and M4-M7. If circular buffering is in use, modulus logic applies (See Chapter 4, "Data Transfer," for more information).

The FO (Flag Out), FL0, FL1 and FL2 pins can each be set, cleared, or toggled. This instruction provides a control structure for multiprocessor communication.

Miscellaneous Instructions

```
NOP;

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

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

MODIFY  (      | I0 |   ,   | M0 |  ) ;
               | I1 |   ,   | M1 |
               | I2 |   ,   | M2 |
               | I3 |   ,   | M3 |
                ______________
               | I4 |   ,   | M4 |
               | I5 |   ,   | M5 |
               | I6 |   ,   | M6 |
               | I7 |   ,   | M7 |

[IF condition]     |SET   |  |FLAG_OUT|    [ , ]    ;
                   |RESET |  |   FL0  |
                   |TOGGLE|  |   FL1  |
                             |   FL2  |

|ENA|  INTS ;
|DIS|
```

Table 15.10 Miscellaneous Instructions

15.9 EXTRA CYCLE CONDITIONS

All instructions execute in a single cycle except under certain conditions, as explained below.

15.9.1 Multiple Off-Chip Memory Accesses

The data and address busses of the ADSP-21xx processors are multiplexed off-chip. Because of this, the processors can perform only one off-chip access per instruction in a single cycle. If two off-chip accesses are required—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.

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. (Two fetches must be from on-chip memory, either PM or DM.)

15.9.2 Wait States

All family processors allow the programming of wait states for external memory chips. Up to seven extra wait state cycles may be added to the processor's access time for external memory. Extra cycles inserted due to wait states are in addition to any caused by multiple off-chip accesses (as described above). Wait state programming is described in the "Memory Interface" chapter.

Wait states and multiple off-chip memory accesses are the two cases when an extra cycle is generated during instruction execution. The following case, SPORT autobuffering and DMA, causes the insertion of extra cycles *between* instructions.

15.9.3 SPORT Autobuffering & DMA

If serial port autobuffering or ADSP-2181 DMA is being used to transfer data words to or from internal memory, then one memory access is "stolen" for each transfer. The stolen memory access occurs only between complete instructions. If extra cycles are required to execute any instruction (for one of the two reasons above), the processor waits until it is completed before "stealing" the access cycle.

15.10 INSTRUCTION SET SYNTAX

The following sections describe instruction set syntax and other notation conventions used in the reference page of each instruction.

15.10.1 Punctuation & Multifunction Instructions

All instructions terminate with a semicolon. A comma separates the clauses of a multifunction instruction but does not terminate it. For example, the statements below in Example A comprise one multifunction instruction (which can execute in a single cycle). Example B shows two separate instructions, requiring two instruction cycles.

Example A: One multifunction instruction

```
AX0 = DM(I0, M0),
AY0 = PM(I4, M4);
```
a comma is used in multifunction instructions

Example B: Two separate instructions

```
AX0 = DM(I0, M0);
AY0 = PM(I4, M4);
```
a semicolon terminates an instruction

15.10.2 Syntax Notation Example

Here is an example of one instruction, the ALU Add/Add with Carry instruction:

```
[ IF cond ]   | AR | =  xop  +   | yop      |      ;
              | AF |             | C        |
                                 | yop + C  |
```

The permissible *conds*, *xops* and *yops* are given in a list. The conditional IF clause is enclosed in square brackets, indicating that it is optional.

The destination register for the add operation must be either AR or AF. These are listed within parallel bars, indicating that one of the two must be chosen.

Similarly, the *yop* term may consist of a Y operand, the carry bit, or the sum of both. One of these three terms must be used.

15.10.3 Status Register Notation

The following notation is used in the discussion of the effect each instruction has on the processors' status registers:

*	An asterisk indicates a bit in the status word that is changed by the execution of the instruction.
–	A dash indicates that a bit is not affected by the instruction.
0 or 1	Indicates that a bit is unconditionally cleared or set.

For example, the status word ASTAT is shown below:

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

Here the MV bit is updated and the AV bit is cleared.