# 2 COMPUTATIONAL UNITS

The ADSP-21535 DSP's computational units perform numeric processing for DSP and general control algorithms. The six computational units are two arithmetic/logic units (ALUs), two multiplier/accumulator (multiplier) units, a shifter, and a set of video ALUs. These units get data from registers in the Data Register File. Computational instructions for these units provide fixed-point operations, and each computational instruction can execute every cycle.

The computational units handle different types of operations. The ALUs perform arithmetic and logic operations. The multipliers perform multiplication and execute multiply/add and multiply/subtract operations. The shifter executes logical shifts and arithmetic shifts and performs bit packing and extraction. The video ALUs perform single instruction, multiple data (SIMD) logical operations on specific 8-bit data operands.

Data moving in and out of the computational units goes through the Data Register File, which consists of eight registers, each 32 bits wide. In operations requiring 16-bit operands, the registers are paired, providing sixteen possible 16-bit registers.

The DSP's assembly language provides access to the Data Register File. The syntax lets programs move data to and from these registers and specify a computation's data format at the same time.

Figure 2-1 provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Tracing inputs to the computational units and considering details about

register files and data buses shows how to set up the data flow for computations. Next, details about the DSP's advanced parallelism reveal how to take advantage of multifunction instructions.

Figure 2-1 shows the relationship between the ADSP-21535 Data Register File and computational units: multipliers, ALUs, and shifter.

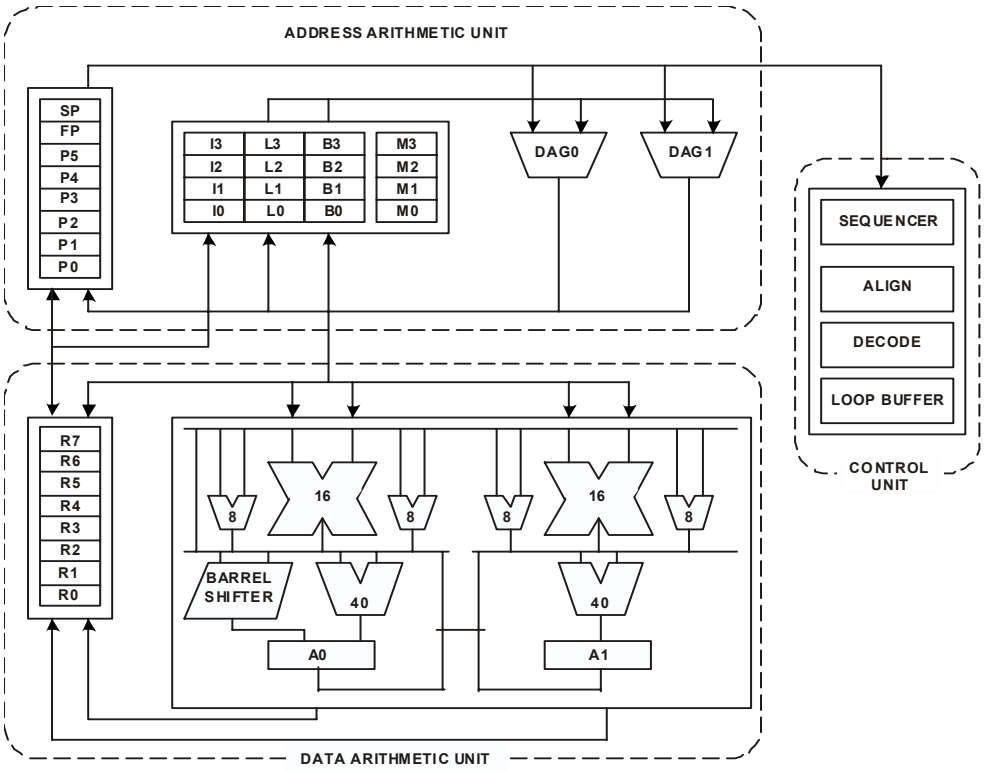


Figure 2-1. ADSP-21535 DSP Core Architecture

Single function multiplier, ALU, and shifter instructions have unrestricted access to the data registers in the Data Register File. Multifunction operations may have restrictions that are described in the section for that operation.

Two additional registers, A0 and A1, provide 40-bit accumulator results. These registers are dedicated to the ALUs and are used primarily for multiply-and-accumulate functions.

The traditional modes of arithmetic operations, such as fractional and integer, are specified directly in the instruction. Rounding modes are set from the `ASTAT` register, which also records status/conditions for the results of the computational operations.

# Using Data Formats

Blackfin DSPs are primarily 16-bit, fixed-point machines. Most operations assume a two's complement number representation, while others assume unsigned numbers or simple binary strings. Other instructions support 32-bit integer arithmetic, with further special features supporting 8-bit arithmetic and block floating point. For detailed information about each number format, see "Numeric Formats" on page D-1.

In the Blackfin DSP family arithmetic, signed numbers are always in two's complement format. These DSPs do not use signed-magnitude, one's complement, BCD, or excess-n formats.

## Binary String

The binary string format is the least complex binary notation; in it, sixteen bits are treated as a bit pattern. Examples of computations using this format are the logical operations: NOT, AND, OR, XOR. These ALU operations treat their operands as binary strings with no provision for sign bit or binary point placement.

## Unsigned

Unsigned binary numbers may be thought of as positive and having nearly twice the magnitude of a signed number of the same length. The DSP treats the least significant words of multiple precision numbers as unsigned numbers.

## Signed Numbers: Two's Complement

In Blackfin DSP arithmetic, the word *signed* refers to two's complement numbers. Most Blackfin DSP family operations presume or support two's complement arithmetic.

## Fractional Representation: 1.15

Blackfin DSP arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 ("one dot fifteen"). In the 1.15 format, one sign bit (the MSB) and fifteen fractional bits represent values from –1 up to one LSB less than +1.

Figure 2-2 shows the bit weighting for 1.15 numbers. and some examples of 1.15 numbers and their decimal equivalents.

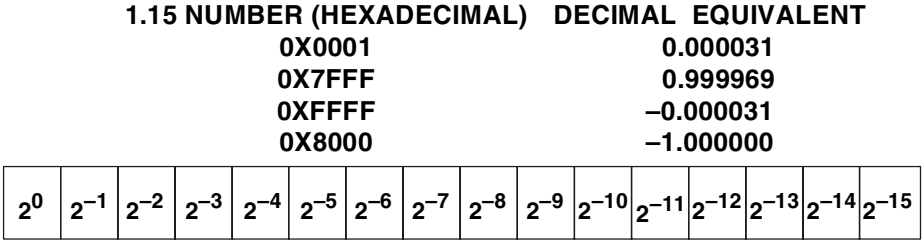


Figure 2-2. Bit Weighting for 1.15 Numbers

# Register Files

The ADSP-21535 DSP's computational units have three definitive register groups—a Data Register File, a Pointer Register File, and set of Data Address Generator (DAG) registers:

- The Data Register File receives operands from the data buses for the computational units and stores computational results.
- The Pointer Register File has pointers for addressing operations.
- The DAG registers are dedicated registers that manage zero-overhead circular buffers for DSP operations.

For more information, see "Data Address Generators" on page 5-1.

The ADSP-21535 DSP register files appear in Figure 2-3.

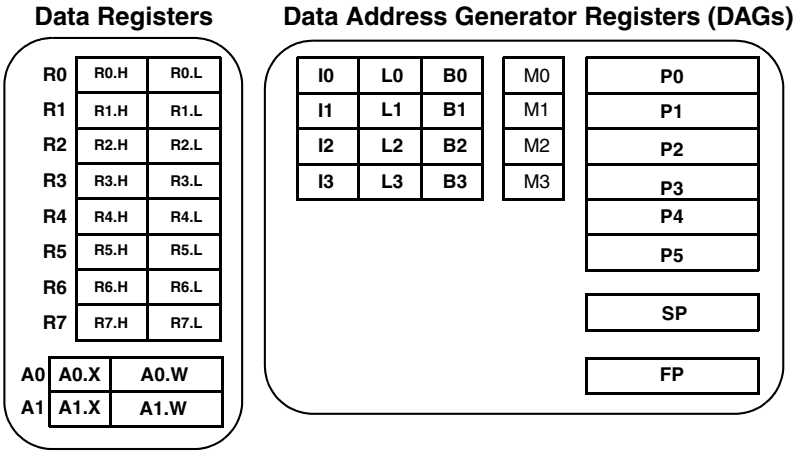


Figure 2-3. ADSP-21535 DSP Register Files

In the ADSP-21535 DSP, a *word* is 32 bits long; *H* denotes the high order 16 bits of a 32-bit register; *L* denotes the low order 16 bits of a 32-bit register. For example, `A0.W` contains the lower 32 bits of the 40-bit `A0` register; `A0.L` contains the lower 16 bits of `A0.W`, and `A0.H` contains the upper 16 bits of `A0.W`.

## Data Register File

The Data Register File consists of eight registers, each of which are 32 bits wide. Each register may be viewed as a pair of independent 16-bit registers. Each is denoted as the low half or high half. Thus the 32-bit register `R0` may be regarded as two independent register halves, `R0.L` and `R0.H`

Two separate buses connect the register file to the L1 data memory, each bus being 32 bits wide. Transfers between the Data Register File and the data memory can move up to four 16-bit words of valid data in each cycle.

## Accumulator Registers

In addition to the Data Register File, the ADSP-21535 DSP has two dedicated, 40-bit accumulator registers. Each may be referred to as its 16-bit low half (`An.L`) or high half (`An.H`) plus its 8-bit extension (`An.X`), or as a 32-bit register (`An.W`) consisting of the lower 32 bits, or as a complete 40-bit result register (`An`).

## Pointer Register File

The general-purpose address Pointer registers, also called the P-registers, are organized as:

- A 6-entry, P-register file `P[5:0]`
- A Frame Pointer (`FP`) used to point to the current procedure's activation record
- A Stack Pointer register (`SP`) used to point to the last used location on the runtime stack. See mode dependent registers in "Operating Modes and States" on page 3-1.

P-registers are 32 bits wide. Although P-registers are primarily used for address calculations, they may also be used for general integer arithmetic with a limited set of arithmetic operations, for instance, to maintain counters. However, unlike the Data registers, P-register arithmetic does not affect the Arithmetic Status (`ASTAT`) register status flags.

## DAG Register Set

DSP instructions primarily use the Data Address Generator (DAG) register set for addressing. The DAG register set consists of these registers:

- `I[3:0]` contain index addresses
- `M[3:0]` contain modify values

- B[3:0] contain base addresses
- L[3:0] contain length values

All DAG registers are 32 bits wide.

The I (Index) registers and B (Base) registers always contain addresses of 8-bit bytes in memory. The Index registers contain an effective address. The M (Modify) registers contain an offset value that is added to one of the Index registers or subtracted from it.

The B (Base) and L (Length) registers define circular buffers. B contains the starting address of a buffer, and L contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, L0 and B0 are always associated with I0. However, any M register may be associated with any I register. For example, I0 may be modified by M3. For more information, see "Data Address Generators" on page 5-1.

## Register File Instruction Summary

Table 2-1 lists the register file instructions. For more information about assembly language syntax, see the *Blackfin DSP Instruction Set Reference*.

In Table 2-1, note the meaning of these symbols:

- Allreg denotes R[7:0], P[5:0], SP, FP, I[3:0], M[3:0], B[3:0], L[3:0], A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC[1:0], LT[1:0], LB[1:0], USP, SEQSTAT, SYSCFG, EMUDAT, CYCLES, and CYCLES2.
- A*n* denotes A0 or A1.
- Dreg denotes any Data Register File register.
- Sysreg denotes ASTAT, SEQSTAT, SYSCFG, RETI, RETX, RETN, RETE, or RETS, LC[1:0], LT[1:0], LB[1:0], EMUDAT, CYCLES, and CYCLES2.
- Preg denotes any Pointer register, FP or SP register.

- Dreg_even denotes R0, R2, R4, or R6.
- Dreg_odd denotes R1, R3, R5, or R7.
- DPreg denotes any Data Register File register or any Pointer register, FP, or SP register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- A*n*.L denotes the lower 16 bits of Accumulator A*n*.W.
- A*n*.H denotes the upper 16 bits of Accumulator A*n*.W.
- Dreg_byte denotes the low order 8 bits of each Data register.
- Option (X) denotes sign extended.
- Option (Z) denotes zero extended.
- * Indicates the flag may be set or cleared, depending on the result of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.

Table 2-1. Register File Instruction Summary

| Instruction | ASTAT Status Flags | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | CC | V<br>VS |
| allreg = allreg ; [1] | * | * | – | – | – | – | * |
| An = An ; | – | – | – | – | – | – | – |
| An = Dreg ; | – | – | – | – | – | – | – |

Table 2-1. Register File Instruction Summary (Cont'd)

| Instruction | ASTAT Status Flags | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | CC | V<br>VS |
| Sysreg = Preg ; | – | – | – | – | – | – | – |
| Dreg_even = A0 ; | * | * | – | – | – | – | * |
| Dreg_odd = A1 ; | * | * | – | – | – | – | * |
| Dreg_even = A0, Dreg_odd = A1 ; | * | * | – | – | – | – | * |
| Dreg_odd = A1, Dreg_even = A0 ; | * | * | – | – | – | – | * |
| Dreg_even = A0.X ; | * | * | – | – | – | – | * |
| Dreg_odd = A1.X ; | * | * | – | – | – | – | * |
| IF CC DPreg = DPreg ; | – | – | – | – | – | – | – |
| IF ! CC DPreg = DPreg ; | – | – | – | – | – | – | – |
| Dreg = Dreg_lo (Z) ; | * | ** | ** | – | – | – | **/– |
| Dreg = Dreg_lo (X) ; | * | * | ** | – | – | – | **/– |
| An.X = Dreg_lo ; | * | * | – | – | – | – | * |
| Dreg_lo = An.X ; | * | * | – | – | – | – | * |
| An.L = Dreg_lo ; | * | * | – | – | – | – | * |
| An.H = Dreg_hi ; | * | * | – | – | – | – | * |
| Dreg_lo = A0 ; | * | * | – | – | – | – | * |
| Dreg_hi = A1 ; | * | * | – | – | – | – | * |
| Dreg = Dreg_byte (Z) ; | * | ** | ** | – | – | – | **/– |
| Dreg = Dreg_byte (X) ; | * | * | ** | – | – | – | **/– |

1 Warning: not all register combinations are allowed. For details, see the Functional Description of the Move Register instruction in the *Blackfin DSP Instruction Set Reference*.

# Data Types

The ADSP-21535 processor supports 32-bit words, 16-bit half words, and bytes. The 32- and 16-bit words can be integer or fractional, but bytes are always integers. Integer data types can be signed or unsigned, but fractional data types are always signed.

Table 2-2 illustrates the data formats for data that resides in memory, in the register file, and in the accumulators. In the table, the letter *d* represents one bit, and the letter *s* represents one signed bit.

Some instructions manipulate data in the registers by sign extending or zero extending the data to 32 bits:

- Instructions zero-extend unsigned data
- Instructions sign-extend signed 16-bit half words and 8-bit bytes

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values. For details, refer to the instructions in the *Blackfin DSP Instruction Set Reference*.

## Data Formats

In Table 2-2, note the meaning of these symbols:

- s = sign bit(s)
- d = data bit(s)
- "." = decimal point by convention; however, a decimal point does not literally appear in the number.

- Italics denotes data from a source other than adjacent bits.

Table 2-2. Data Formats

| Format | Representation in Memory | Representation in 32-bit Register |
|---|---|---|
| 32.0 Unsigned Word | dddd dddd dddd dddd dddd dddd dddd dddd | dddd dddd dddd dddd dddd dddd dddd dddd |
| 32.0 Signed Word | sddd dddd dddd dddd dddd dddd dddd dddd | sddd dddd dddd dddd dddd dddd dddd dddd |
| 16.0 Unsigned Half Word | dddd dddd dddd dddd | 0000 0000 0000 0000 dddd dddd dddd dddd |
| 16.0 Signed Half Word | sddd dddd dddd dddd | ssss ssss ssss ssss sddd dddd dddd dddd |
| 8.0 Unsigned Byte | dddd dddd | 0000 0000 0000 0000 0000 0000 dddd dddd |
| 8.0 Signed Byte | sddd dddd | ssss ssss ssss ssss ssss ssss sddd dddd |
| 0.16 Unsigned Fraction | .dddd dddd dddd dddd | 0000 0000 0000 0000 .dddd dddd dddd dddd |
| 1.15 Signed Fraction | s.ddd dddd dddd dddd | ssss ssss ssss ssss s.ddd dddd dddd dddd |
| 0.32 Unsigned Fraction | .dddd dddd dddd dddd dddd dddd dddd dddd | .dddd dddd dddd dddd dddd dddd dddd dddd |
| 1.31 Signed Fraction | s.ddd dddd dddd dddd dddd dddd dddd dddd | s.ddd dddd dddd dddd dddd dddd dddd dddd |
| Packed 8.0 Unsigned Byte | dddd dddd *dddd dddd* dddd dddd *dddd dddd* | dddd dddd *dddd dddd* dddd dddd *dddd dddd* |
| Packed 0.16 Unsigned Fraction | .dddd dddd dddd dddd .dddd dddd dddd dddd | .dddd dddd dddd dddd *.dddd dddd dddd dddd* |
| Packed 1.15 Signed Fraction | s.ddd dddd dddd dddd *s.ddd dddd dddd dddd* | s.ddd dddd dddd dddd *s.ddd dddd dddd dddd* |

## Endianess

Both internal and external memory are accessed in little endian byte order. For more information, see "Memory Transaction Model" on page 6-80.

## ALU Data Types

Operations on each ALU treat operands and results as either 16- or 32-bit binary strings, except the signed division primitive (DIVS). ALU result status bits treat the results as signed, indicating status with the overflow flags (AV0, AV1) and the negative flag (AN). Each ALU has its own sticky overflow flag, AV0S and AV1S. Once set, these bits remain set until cleared by writing directly to the ASTAT register. An additional V flag is set or cleared depending on the transfer of the result from both accumulators to the register file. Furthermore, the sticky VS bit is set with the V bit and remains set until cleared.

The logic of the overflow bits (V, VS, AV0, AV0S, AV1, AV1S) is based on two's complement arithmetic. A bit or set of bits is set if the MSB changes in a manner not predicted by the signs of the operands and the nature of the operation. For example, adding two positive numbers must generate a positive result; a change in the sign bit signifies an overflow and sets AVn, the corresponding overflow flags. Adding a negative and a positive may result in either a negative or positive result, but cannot overflow.

The logic of the carry bits (AC0, AC1) is based on unsigned magnitude arithmetic. The bit is set if a carry is generated from bit 16 (the MSB). The carry bits (AC0, AC1) are most useful for the lower word portions of a multiword operation.

ALU results generate status information. For more information about using ALU status, see "ALU Instruction Summary" on page 2-30.

## Multiplier Data Types

Each multiplier produces results that are binary strings. The inputs are interpreted according to the information given in the instruction itself (whether it is signed multiplied by signed, unsigned multiplied by unsigned, a mixture, or a rounding operation). The 32-bit result from the multipliers is assumed to be signed; it is sign extended across the full 40-bit width of the `A0` or `A1` registers.

The ADSP-21535 DSPs support two modes of format adjustment: the fractional mode for fractional operands (1.15 format with 1 sign bit and 15 fractional bits) and the integer mode for integer operands (16.0 format).

When the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits and 30 fractional bits) number. In the fractional mode, the multiplier automatically shifts the multiplier product left one bit before transferring the result to the multiplier result register (`A0`, `A1`). This shift of the redundant sign bit causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. The resulting format appears in Figure 2-4 on page 2-17.

In the integer mode, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0 format. A left shift is not needed and would change the numerical representation. This result format appears in Figure 2-5 on page 2-18.

Multiplier results generate status information when they are transferred to a destination register in the register file. For more information, see “Multiplier Instruction Summary” on page 2-40.

## Shifter Data Types

Many operations in the shifter are explicitly geared to signed (two's complement) or unsigned values: logical shifts assume unsigned magnitude or binary string values, and arithmetic shifts assume two's complement values.

The exponent logic assumes two's complement numbers. The exponent logic supports block floating point, which is also based on two's complement fractions.

Shifter results generate status information. For more information about using shifter status, see "Shifter Instruction Summary" on page 2-56.

## Arithmetic Formats Summary

Table 2-3, Table 2-4, Table 2-5, and Table 2-9 summarize some of the arithmetic characteristics of computational operations.

Table 2-3. ALU Arithmetic Formats

| Operation | Operand Formats | Result Formats |
|---|---|---|
| Addition | Signed or unsigned | Interpret flags |
| Subtraction | Signed or unsigned | Interpret flags |
| Logical | Binary String | Same as operands |
| Division | Explicitly signed or unsigned | Same as operands |

Table 2-4. Multiplier Fractional Modes Formats

| Operation | Operand Formats | Result Formats |
|---|---|---|
| Multiplication | 1.15 explicitly signed or unsigned | 2.30 shifted to 1.31 |
| Multiplication / addition | 1.15 explicitly signed or unsigned | 2.30 shifted to 1.31 |
| Multiplication / subtraction | 1.15 explicitly signed or unsigned | 2.30 shifted to 1.31 |

Table 2-5. Multiplier Arithmetic Integer Modes Formats

| Operation | Operand Formats | Result Formats |
|---|---|---|
| Multiplication | 16.0 explicitly signed or unsigned | 32.0 not shifted |
| Multiplication / addition | 16.0 explicitly signed or unsigned | 32.0 not shifted |
| Multiplication / subtraction | 16.0 explicitly signed or unsigned | 32.0 not shifted |

Table 2-6. Shifter Arithmetic Formats

| Operation | Operand Formats | Result Formats |
|---|---|---|
| Logical Shift | Unsigned binary string | Same as operands |
| Arithmetic Shift | Signed | Same as operands |
| Exponent Detect | Signed | Same as operands |

## Using Multiplier Integer and Fractional Formats

For multiply-and-accumulate functions, the ADSP-21535 DSP provides two choices: fractional arithmetic for fractional numbers (1.15) and integer arithmetic for integers (16.0).

For fractional arithmetic, the 32-bit product output is format adjusted—sign extended and shifted one bit to the left—before being added to accumulator `A0` or `A1`. For example, bit 31 of the product lines up with bit 32 of `A0` (which is bit 0 of `A0.X`), and bit 0 of the product lines up with bit 1 of `A0` (which is bit 1 of `A0.W`). The LSB is zero filled. The fractional multiplier result format appears in Figure 2-4.

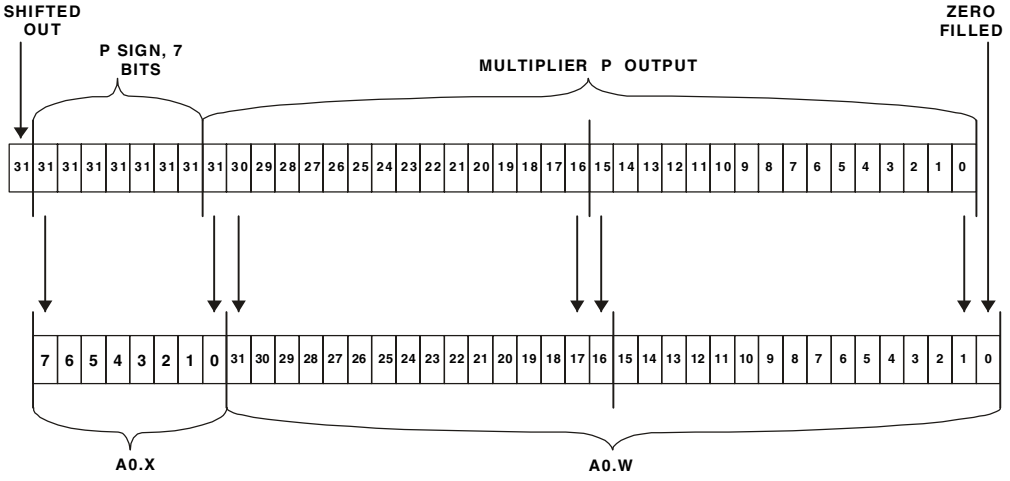


Figure 2-4. Fractional Multiplier Results Format

For integer arithmetic, the 32-bit product register is not shifted before being added to A0 or A1. Figure 2-5 shows the integer mode result placement.

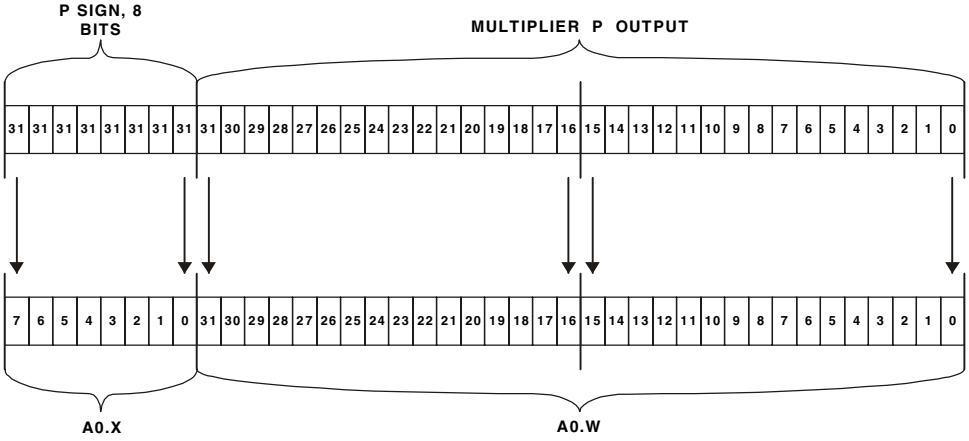


Figure 2-5. Integer Multiplier Results Format

With either fractional or integer operations, the multiplier output product is fed into a 40-bit adder/subtracter which adds or subtracts the new product with the current contents of the A0 or A1 register to produce the final 40-bit result.

## Rounding Multiplier Results

On many multiplier operations, the DSP supports multiplier results rounding (RND option). Rounding is a means of reducing the precision of a number by removing a lower order range of bits from that number's representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, the original num-

ber will have N bits of precision, whereas the new number will have only M bits of precision (where N>M). The process of rounding, then, removes N-M bits of precision from the number.

The RND_MOD bit in the ASTAT register determines whether the RND option provides biased or unbiased rounding. For *unbiased* rounding, set RND_MOD bit = 0. For *biased* rounding, set RND_MOD bit = 1.

For most algorithms, unbiased rounding is preferred.

## Unbiased Rounding

The *convergent rounding* method returns the number closest to the original. In cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. For example, when rounding the 3-bit, two's complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two's complement fraction, the result would be 0.0, because that is the even-numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called *unbiased rounding*.

Unbiased rounding uses the multiplier's capability of rounding the 40-bit result at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction code. When rounding is selected, A0.H/A1.H contain the rounded 16-bit result; the rounding effect in A0.H/A1.H affects A0.X/A1.X as well. The A0.X/A0.H and A1.X/A1.H registers represent the rounded 24-bit result, including sign extension and overflow.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding adds a 1 into bit position 15 of the adder chain. This method causes a net positive bias because the midway value (when A0.L/A1.L = 0x8000) is always rounded upward.

The accumulator eliminates this bias by forcing bit 16 in the result output to zero when it detects this midway point. Forcing bit 16 to zero has the effect of rounding odd `A0.L/A1.L` values upward and even values downward, yielding a zero large sample bias, assuming uniformly distributed values.

The following examples use *x* to represent any bit pattern (not all zeros). The example in Figure 2-6 shows a typical rounding operation for `A0`; the example also applies for `A1`.

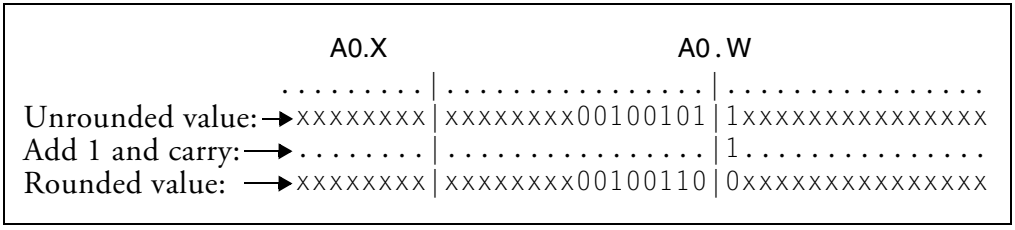


Figure 2-6. Typical Unbiased Multiplier Rounding

The compensation to avoid net bias becomes visible when all lower 15 bits are zero and bit 15 is one (the midpoint value) as shown in Figure 2-7.

In Figure 2-7, A0 bit 16 is forced to zero. This algorithm is employed on every rounding operation, but is evident only when the bit patterns shown in the lower 16 bits of the next example are present.

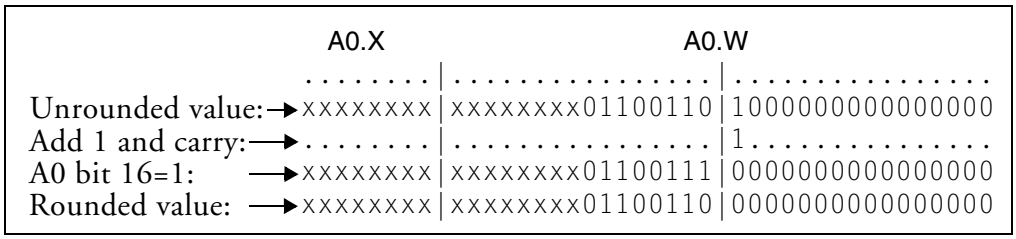


Figure 2-7. Avoiding Net Bias in Unbiased Multiplier Rounding

## Biased Rounding

The *round-to-nearest* method also returns the number closest to the original. However, by convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the 3-bit, two's complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two's complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called *biased rounding*.

The RND_MOD bit in the ASTAT register enables biased rounding. When the RND_MOD bit is cleared, the RND option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in "Unbiased Rounding" on page 2-19.

When the RND_MOD bit is set (=1), the DSP uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations with A0.L/A1.L set to 0x8000 round up, rather than only rounding odd values up. For an example of biased rounding, see Figure 2-8.

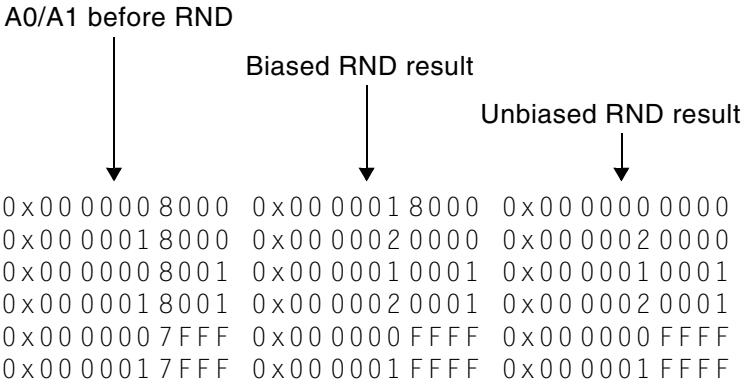


Figure 2-8. Biased Rounding in Multiplier Operation

Biased rounding affects the result only when the A0.L/A1.L register contains 0x8000; all other rounding operations work normally. This mode allows more efficient implementation of bit specified algorithms that use biased rounding, for example, the Global System for Mobile Communications (GSM) speech compression routines.

## Truncation

Another common way to reduce the significant bits representing a number is to simply mask off the N-M lower bits. This process is known as *truncation* and results in a relatively large bias. Instructions that do not support rounding revert to truncation. The RND_MOD bit in ASTAT has no effect on truncation.

## Special Rounding Instructions

The ALU provides the ability to round the arithmetic results directly into a data register with biased or unbiased rounding as described above. It also provides the ability to round on different bit boundaries. The options RND12, RND and RND20 extract 16-bit values from bit 12, bit 16 and bit 20, respectively, and perform biased rounding regardless of the state of the RND_MOD bit in ASTAT.

For example:

```
R3.L = R4 ( RND ) ;
```

performs biased rounding at bit 16, depositing the result in a half word.

```
R3.L = R4 + R5 ( RND12 ) ;
```

performs an addition of two 32-bit numbers, biased rounding at bit 12, depositing the result in a half word.

```
R3.L = R4 + R5 ( RND20 ) ;
```

performs an addition of two 32-bit numbers, biased rounding at bit 20, depositing the result in a half word.

# Using Computational Status

The multiplier, ALU, and shifter update the overflow and other status flags in the DSP's Arithmetic Status (ASTAT) register. To use status conditions from computations in program sequencing, use conditional instructions to test the CC flag (bit 5, ASTAT register) after the instruction executes. This method permits monitoring each instruction's outcome. The ASTAT register is a 32-bit register, with some bits reserved. To ensure compatibility with future implementations, writes to this register should write back the values read from these reserved bits.

# Arithmetic Status Register (ASTAT)

Figure 2-9 describes the ASTAT register. The DSP updates the status bits in ASTAT, indicating the status of the most recent ALU, multiplier, or shifter operation.

# Arithmetic Logic Unit (ALU)

The two ALUs perform arithmetic and logical operations on fixed-point data. ALU fixed-point instructions operate on 16-bit, 32-bit, and 40-bit fixed-point operands and output 16-bit, 32-bit, or 40-bit fixed-point results. ALU instructions include:

- Fixed-point addition and subtraction of registers
- Addition and subtraction of immediate values
- Accumulator and subtraction of multiplier results
- Logical AND, OR, NOT, XOR, bitwise XOR, Negate
- Functions: ABS, MAX, MIN, Round, division primitives

## ALU Operations

Primary ALU operations occur on ALU0, while parallel operations occur on ALU1, which performs a subset of ALU0 operations.

**Arithmetic Status Register (ASTAT)**

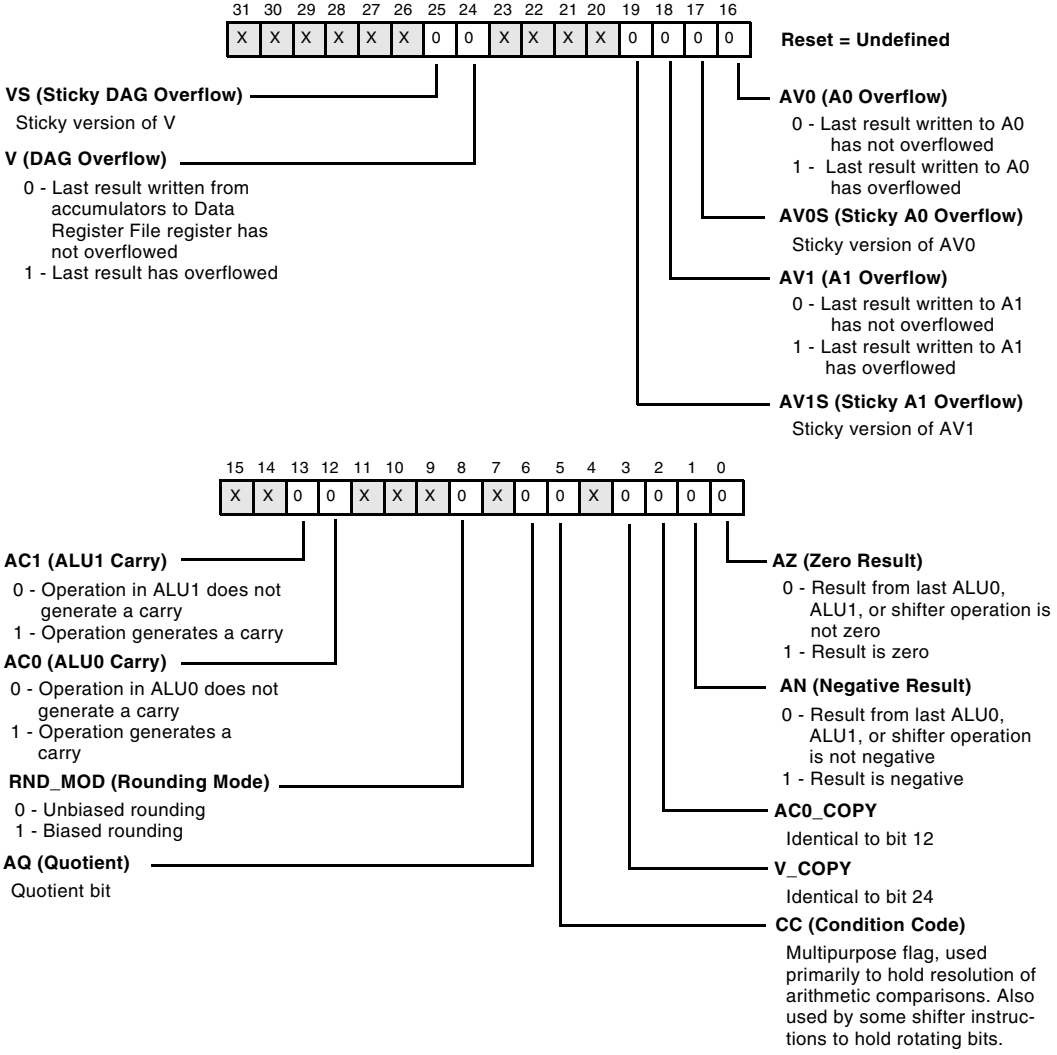


Figure 2-9. Arithmetic Status Register

Table 2-7 describes the possible inputs and outputs of each ALU.

Table 2-7. Inputs and Outputs of Each ALU

| Input | Output |
|---|---|
| Two or four 16-bit operands | One or two 16-bit results |
| Two 32-bit operands | One 32-bit result |
| 32-bit result from the multiplier | Combination of 32-bit result from the multiplier with a 40-bit accumulation result |

Combining operations in both ALUs can result in four 16-bit results, two 32-bit results, or two 40-bit results generated in a single instruction.

## Single 16-Bit Operations

In single 16-bit operations, any two 16-bit register halves may be used as the input to the ALU. An addition, subtraction, or logical operation produces a 16-bit result that is deposited into an arbitrary destination register half. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

```
R3.H = R1.H + R2.L (NS) ;
```

adds the 16-bit contents of `R1.H` (`R1` high half) to the contents of `R2.L` (`R2` low half) and deposits the result in `R3.H` (`R3` high half) with no saturation.

## Dual 16-Bit Operations

In dual 16-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as pairs of 16-bit operands. An addition, subtraction, or logical operation produces two 16-bit results that are deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

```
R3 = R1 +|- R2 (S) ;
```

adds the 16-bit contents of `R2.H` (`R2` high half) to the contents of `R1.H` (`R1` high half) and deposits the result in `R3.H` (`R3` high half) with saturation.

The instruction also subtracts the 16-bit contents of `R2.L` (`R2` low half) from the contents of `R1.L` (`R1` low half) and deposits the result in `R3.L` (`R3` low half) with saturation (see Figure 2-11 on page 2-35).

## Quad 16-Bit Operations

In quad 16-bit operations, any two 32-bit registers may be used as the inputs to ALU0 and ALU1, considered as pairs of 16-bit operands. A small number of addition or subtraction operations produces four 16-bit results that are deposited into two arbitrary, 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because there are only two 32-bit data paths from the Data Register File to the arithmetic units, the same two pairs of 16-bit inputs are presented to ALU1 as to ALU0. The instruction construct is identical to that of a dual 16-bit operation, and input operands must be the same for both ALUs.

For example:

```
R3 = R0 +|+ R1, R2 = R0 -|- R1 (S) ;
```

performs four operations:

- Adds the 16-bit contents of `R1.H` (`R1` high half) to the 16-bit contents of the `R0.H (R0` high half) and deposits the result in `R3.H`, with saturation.
- Adds `R1.L + R0.L` and deposits the result in `R3.L`, with saturation.
- Subtracts 16-bit contents of `R1.H` (`R1` high half) from the 16-bit contents of the `R0.H (R0` high half) and deposits the result in `R2.H`, with saturation.
- Subtracts `R1.L` from `R0.L` and deposits the result in `R2.L`, with saturation.

Explicitly, the four equivalent instructions are:

```
R3.H = R0.H + R1.H (S) ;
R3.L = R0.L + R1.L (S) ;
R2.H = R0.H - R1.H (S) ;
R2.L = R0.L - R1.L (S) ;
```

## Single 32-Bit Operations

In single 32-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as 32-bit operands. An addition, subtraction, or logical operation produces a 32-bit result that is deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

In addition to the 32-bit input operands coming from the Data Register File, operands may be sourced and deposited into the Pointer Register File, consisting of the eight registers `P[5:0], SP, FP`.

Instructions may not intermingle Pointer registers with Data registers.

For example:

```
R3 = R1 + R2 (NS) ;
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3` with no saturation.

```
R3 = R1 + R2 (S) ;
```

adds the 32-bit contents of `R1` to the 32-bit contents of `R2` and deposits the result in `R3` with saturation.

## Dual 32-Bit Operations

In dual 32-bit operations, any two 32-bit registers may be used as the input to ALU0 and ALU1, considered as a pair of 32-bit operands. An addition or subtraction produces two 32-bit results that are deposited into two 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because only two 32-bit data paths go from the Register File to the arithmetic units, the same two 32-bit input registers are presented to ALU0 and ALU1.

For example:

```
R3 = R1 + R2, R4 = R1 - R2 (NS) ;
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3`.

The instruction also subtracts the 32-bit contents of `R2` from that of `R1` and deposits the result in `R4` with no saturation.

A specialized form of this instruction uses the ALU 40-bit result registers as input operands, creating the sum and differences of the A0 and A1 registers.

For example:

```
R3 = A0 +A1, R4 = A0-A1 (S) ;
```

transfers to the result registers two 32-bit, saturated, sum and difference values of the ALU registers.

## ALU Instruction Summary

Table 2-8 lists the ALU instructions. For more information about assembly language syntax and the effect of ALU instructions on the status flags, see the *Blackfin DSP Instruction Set Reference*.

In Table 2-8, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Preg denotes any Pointer register, FP, or SP register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- imm7 denotes a signed, 7 bits wide, immediate value.
- A*n* denotes either ALU result register A0 or A1.
- DIVS denotes a Divide Sign primitive.
- DIVQ denotes a Divide Quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.

- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result on bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result on bit 20.
- SIGNBITS denotes the number of sign bits in a number.
- EXPADJ denotes the absolute value of a 32-bit register.
- * Indicates the flag may be set or cleared, depending on results of instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.
- *d* indicates `AQ` contains the dividend MSB Exclusive-OR divisor MSB.

Table 2-8. ALU Instruction Summary

| Instruction | ASTAT Status Flags | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | V<br>VS | AQ |
| Preg = Preg + Preg ; | – | – | – | – | – | – | – |
| Preg += Preg ; | – | – | – | – | – | – | – |
| Preg -= Preg ; | – | – | – | – | – | – | – |
| Dreg = Dreg + Dreg ; | * | * | * | – | – | * | – |
| Dreg = Dreg – Dreg (S) ; | * | * | * | – | – | * | – |
| Dreg = Dreg + Dreg, Dreg = Dreg - Dreg ; | * | * | * | – | – | * | – |
| Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ; | * | * | * | – | – | * | – |
| Dreg_lo_hi = Dreg_lo_hi - Dreg_lo_hi (S) ; | * | * | * | – | – | * | – |
| Dreg = Dreg +\|+ Dreg ; | * | * | * | – | – | * | – |
| Dreg = Dreg +\|– Dreg ; | * | * | * | – | – | * | – |
| Dreg = Dreg –\|+ Dreg ; | * | * | * | – | – | * | – |
| Dreg = Dreg –\|– Dreg ; | * | * | * | – | – | * | – |
| Dreg = Dreg +\|+Dreg, Dreg = Dreg –\|– Dreg ; | * | * | * | – | – | * | – |
| Dreg = Dreg +\|– Dreg, Dreg = Dreg –\|+ Dreg ; | * | * | * | – | – | * | – |
| Dreg = An + An, Dreg = An – An ; | * | * | * | – | – | * | – |
| Dreg += imm7 ; | * | * | * | – | – | * | – |
| Preg += imm7 ; | – | – | – | – | – | – | – |
| Dreg= ( A0 += A1 ) ; | * | * | * | * | – | * | – |
| Dreg_lo_hi = ( A0 += A1) ; | * | * | * | * | – | * | – |
| A0 += A1 ; | * | * | * | * | – | * | – |
| A0 –= A1 ; | * | * | * | * | – | – | – |
| DIVS ( Dreg, Dreg ) ; | * | * | * | * | – | – | d |
| DIVQ ( Dreg, Dreg ) ; | * | * | * | * | – | – | d |
| Dreg = MAX ( Dreg, Dreg ) (V) ; | * | * | – | – | – | **/– | – |
| Dreg = MIN ( Dreg, Dreg ) (V) ; | * | * | – | – | – | **/– | – |
| Dreg = ABS Dreg (V) ; | * | ** | * | – | – | * | – |

Table 2-8. ALU Instruction Summary (Cont'd)

| Instruction | ASTAT Status Flags | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | V<br>VS | AQ |
| An = ABS An ; | * | ** | * | * | * | * | – |
| An = ABS An, An = ABS An ; | * | ** | * | * | * | * | – |
| An = –An ; | * | * | – | * | * | * | – |
| An = –An, An =– An ; | * | * | – | * | * | * | – |
| An = An (S) ; | * | * | – | * | * | – | – |
| An = An (S), An = An (S) ; | * | * | – | * | * | – | – |
| Dreg_lo_hi = Dreg (RND) ; | * | * | – | – | – | * | – |
| Dreg_lo_hi = Dreg + Dreg (RND12) ; | * | * | – | – | – | * | – |
| Dreg_lo_hi = Dreg - Dreg (RND12) ; | * | * | – | – | – | * | – |
| Dreg_lo_hi = Dreg + Dreg (RND20) ; | * | * | – | – | – | * | – |
| Dreg_lo_hi = Dreg - Dreg (RND20) ; | * | * | – | – | – | * | – |
| Dreg_lo = SIGNBITS Dreg ; | * | – | – | – | – | – | – |
| Dreg_lo = SIGNBITS Dreg_lo_hi ; | * | – | – | – | – | – | – |
| Dreg_lo = SIGNBITS An ; | * | – | – | – | – | – | – |
| Dreg_lo = EXPADJ ( Dreg, Dreg_lo ) (V) ; | – | – | – | – | – | – | – |
| Dreg_lo = EXPADJ ( Dreg_lo_hi, Dreg_lo ) ; | – | – | – | – | – | – | – |
| Dreg = Dreg & Dreg ; | * | * | ** | **/– | – | – | – |
| Dreg = ~ Dreg ; | * | * | ** | **/– | – | – | – |
| Dreg = Dreg \| Dreg ; | * | * | ** | **/– | – | – | – |
| Dreg = Dreg ^ Dreg ; | * | * | ** | **/– | – | – | – |
| Dreg =– Dreg ; | | | | | | | |

## ALU Data Flow Details

Figure 2-10 shows a more detailed diagram of the Arithmetic Units and Data Register File, which appears in Figure 2-1 on page 2-2.

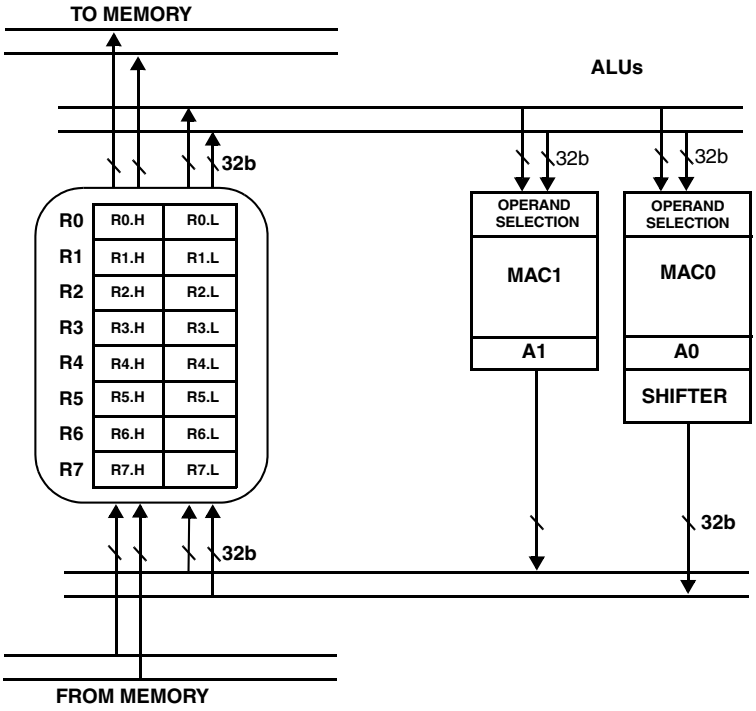


Figure 2-10. Register Files and ALUs

ALU0 is described here for convenience. ALU1 is very similar—a subset of ALU0.

Each ALU performs 40-bit addition for the accumulation of the multiplier results, as well as 32-bit and dual 16-bit operations. Each ALU has two 32-bit input ports that can be considered a pair of 16-bit operands or a single 32-bit operand. For single 16-bit operations, any of the four possible 16-bit operands may be used with any of the other 16-bit operands presented at the input to the ALU.

As shown in Figure 2-11, for dual 16-bit operations, the high halves and low halves are paired, providing four possible combinations of addition and subtraction:

```
(A) H+H, L+L   (B) H+H, L-L   (C) H-H, L+L   (D) H-H, L-L
```

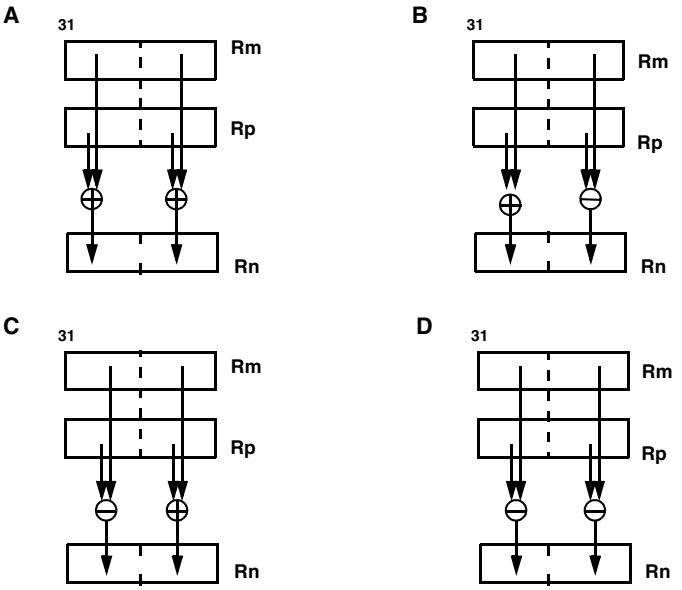


Figure 2-11. Dual 16-Bit ALU Operations

## Dual 16-Bit Cross Options

For dual 16-bit operations, the results may be *crossed*. Crossing the results changes where in the result register an operation places the result of a calculation. Usually, the result from the high side calculation is placed in the high half of the result register, and the result from the low side calculation is placed in the low half of the result register.

With the *cross* option, the high result is placed in the low half of the destination register, and the low result placed in the high half of the destination register (see Figure 2-12). This is particularly useful when dealing with complex math and portions of the Fast Fourier Transform (FFT).

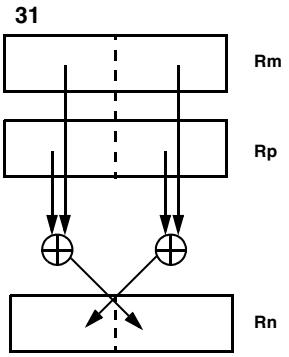


Figure 2-12. Cross Options for Dual 16-Bit ALU Operations

## ALU Status Signals

Each ALU generates six status signals: the zero (AZ) status, the negative (AN) status, the carry (ACn) status, the sticky overflow (AVnS) status, immediate overflow (AVn) status and the quotient (AQ) status. All arithmetic status signals are latched into the arithmetic status register (ASTAT) at the end of the cycle. For the effect of ALU instructions on the status flags, see Table 2-8 on page 2-32.

Depending on the instruction, the inputs can come from the Data Register File, the Pointer Register File, or the Arithmetic Result registers. Arithmetic on 32-bit operands directly support multiprecision operations in the ALU.

## ALU Division Support Features

The ALU supports division with two special divide primitives. These instructions (`DIVS`, `DIVQ`) let programs implement a non-restoring, conditional (error checking), add-subtract-division algorithm.

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in the *Blackfin DSP Instruction Set Reference*.

## Special SIMD Video ALU Operations

Four 8-bit Video ALUs enable the ADSP-21535 processor to process video information with high efficiency. Each Video ALU instruction may take from one to four pairs of 8-bit inputs and return one to four 8-bit results. The inputs are presented to the Video ALUs in two 32-bit words from the Data Register File. The possible operations include:

- Byte alignment
- Quad-byte-sum absolute differences
- Quad-byte averaging
- Quad-byte pack and unpack
- Addition with trimming

For more information about the operation of these instructions, see the *Blackfin DSP Instruction Set Reference*.

# Multiply Accumulators (Multipliers)

The two ADSP-21535 DSP multipliers (MAC0 and MAC1) perform fixed-point multiplication and multiply and accumulate operations. Multiply and accumulate operations are available with either cumulative addition or cumulative subtraction.

Multiplier fixed-point instructions operate on 16-bit fixed-point data and produce 32-bit results that may be added or subtracted from a 40-bit accumulator.

Inputs are treated as fractional or integer, unsigned or two's complement. Multiplier instructions include:

- Multiplication
- Multiply and accumulate with addition, rounding optional
- Multiply and accumulate with subtraction, rounding optional
- Dual versions of the above

## Multiplier Operation

Each multiplier has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply and accumulate instructions, these operands can be any data registers in the Data Register File. Each multiplier can accumulate results in its Accumulator register, `A1` or `A0`. The accumulator results can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

Each multiplier instruction determines whether the inputs are either both in integer format or both in fractional format. The format of the result matches the format of the inputs. In MAC0, both inputs are treated as signed or unsigned. In MAC1, there is a mixed-mode option.

If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Multiplier instruction options specify the data format of the inputs:

- `FU` for Fractional Unsigned
- `IS` for Integer Signed
- `M` for Mixed signed and unsigned operands

Also available is the `W32` option, which specifies 32-bit saturation of the accumulation result.

## Placing Multiplier Results in Multiplier Accumulator Registers

As shown in Figure 2-10 on page 2-34, each multiplier has a dedicated accumulator, `A0` or `A1`. Each accumulator register is divided into three sections: `A0.L/A1.L` (bits 15:0), `A0.H/A1.H` (bits 31:16), and `A0.X/A1.X` (bits 39:32).

When the multiplier writes to its result accumulator registers, the 32-bit result is deposited into the lower bits of the combined accumulator register, and the MSB is sign extended into the upper eight bits of the register (`A0.X/A1.X`).

Multiplier output can be deposited not only in the `A0` or `A1` registers, but also in a variety of 16- or 32-bit Data registers in the Data Register File.

## Rounding or Saturating Multiplier Results

On a multiply and accumulate operation, the accumulator data can be saturated and, optionally, rounded for extraction to a register or register half. When a multiply deposits a result only in a register or register half, the saturation and rounding works the same way.

The rounding and saturation operations work as follows:

- Rounding is applied only to fractional results except for the IH option, which applies rounding and high half extraction to an integer result.

  The rounded result is obtained by adding 0x8000 to the accumulator (for MAC) or multiply result (for mult) and then saturating to 32-bits. For more information, see "Rounding Multiplier Results" on page 2-18.

- If an overflow or underflow has occurred, the saturate operation sets the specified result register to the maximum positive or negative value. For more information, see the following section.

## Saturating Multiplier Results on Overflow

These bits in ASTAT indicate multiplier overflow status:

- Bit 16 (AV0) and bit 18 (AV1) record overflow condition (whether the result has overflowed 32 bits) for the A0 and A1 accumulators, respectively.

  If the bit is cleared (=0), no overflow or underflow has occurred. If the bit is set (=1), an overflow or underflow has occurred. The AV0S and AV1S bits are sticky bits.

- Bit 24 (V) and bit 25 (VS) are set if overflow occurs in extracting the accumulator result to a register.

## Multiplier Instruction Summary

Table 2-9 lists the multiplier instructions. For more information about assembly language syntax and the effect of multiplier instructions on the status flags, see the *Blackfin DSP Instruction Set Reference*.

In Table 2-9, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo_hi denotes any 16-bit register half in any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- An denotes either MAC Accumulator register `A0` or `A1`.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- – Indicates no effect.

For information on multiplier instruction options, see "Multiplier Instruction Options" on page 2-42.

Table 2-9. Multiplier Instruction Summary

| Instruction | ASTAT Status Flags | | |
|---|---|---|---|
| | AV0<br>AV0S | AV1<br>AV1S | V<br>VS |
| Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ; | * | * | – |
| Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ; | * | * | – |
| Dreg = Dreg_lo_hi * Dreg_lo_hi ; | * | * | – |
| An = Dreg_lo_hi * Dreg_lo_hi ; | * | * | – |
| An += Dreg_lo_hi * Dreg_lo_hi ; | * | * | – |
| An –= Dreg_lo_hi * Dreg_lo_hi ; | * | * | – |
| Dreg_lo = ( A0 = Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |

Table 2-9. Multiplier Instruction Summary (Cont'd)

| Instruction | ASTAT Status Flags | | |
|---|---|---|---|
| | AV0<br>AV0S | AV1<br>AV1S | V<br>VS |
| Dreg_lo = ( A0 += Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg_lo = ( A0 –= Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg_hi = ( A1 = Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg_hi = ( A1 += Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg_hi = ( A1 -= Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg = ( An = Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg = ( An += Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg = ( An –= Dreg_lo_hi * Dreg_lo_hi ) ; | * | * | * |
| Dreg *= Dreg ; | – | – | * |

## Multiplier Instruction Options

The following descriptions of multiplier instruction options provide an overview. Not all options are available for all instructions. For information about how to use these options with their respective instructions, see the *Blackfin DSP Instruction Set Reference.*

*default* — No option; input data is signed fraction.

`(IS)` — Input data operands are signed integer. No shift correction is made.

`(FU)` — Input data operands are unsigned fraction. No shift correction is made.

`(IU)` — Input data operands are unsigned integer. No shift correction is made.

| | |
|---|---|
| (T) | Input data operands are signed fraction. When copying to the destination half register, truncates the lower 16 bits of the Accumulator contents. |
| (TFU) | Input data operands are unsigned fraction. When copying to the destination half register, truncates the lower 16 bits of the Accumulator contents. |
| (S2RND) | If multiplying and accumulating to a register:<br><br>Input data operands are signed fraction. When copying to the destination register, scales Accumulator contents (multiply x2 by a one-place shift-left) and rounds. If scaling and rounding produce a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value. |
| (S2RND) | If multiplying and accumulating to a half register:<br><br>Input data operands are signed fraction. When copying to the destination register, scales Accumulator contents (multiply x2 by a one-place shift-left) and rounds the upper 16 bits before truncating the lower 16 bits of the Accumulator. If scaling and rounding produce a signed value larger than 16 bits, the number is saturated to its maximum positive or negative value. |
| (ISS2) | If multiplying and accumulating to a register:<br><br>Input data operands are signed integer. When copying to the destination register, scales Accumulator contents (multiply x2 by a one-place shift-left). If scaling produces a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value. |

| | |
|---|---|
| (ISS2) | If multiplying and accumulating to a half register:<br><br>When copying the lower 16 bits to the destination half-register, scales the Accumulator contents. If scaling produces a signed value greater than 16 bits, the number is saturated to its maximum positive or negative value. |
| (IH) | This option indicates integer multiplication with high half word extraction. The Accumulator is saturated at 32 bits, and bits [31:16] of the Accumulator are rounded, then copied into the destination half register. |
| (W32) | Input data operands are signed fraction with no extension bits in the Accumulators at 32 bits.<br><br>Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit Accumulators. |
| (M) | Operation uses mixed multiply mode. Valid only for MAC1 versions of the instruction. Multiplies a signed fraction by an unsigned fractional operand with no left-shift correction.<br><br>Operand one is signed; operand two is unsigned. MAC0 performs an unmixed multiply on signed fractions by default or another format, as specified. The (M) option can be used alone or in conjunction with one other format option. |

## Multiplier Data Flow Details

Figure 2-13 shows the multiplier/accumulators.

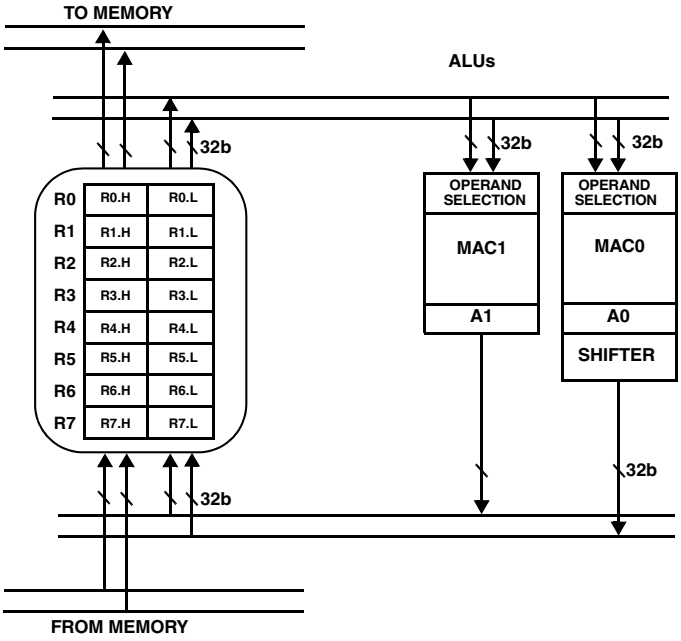


Figure 2-13. Register Files and ALUs

Each multiplier has two 16-bit inputs, performs a 16-bit multiplication, and stores the result in a 40-bit accumulator or extracts to a 16-bit or 32-bit register. Two 32-bit words are available at the MAC inputs, providing four 16-bit operands to chose from.

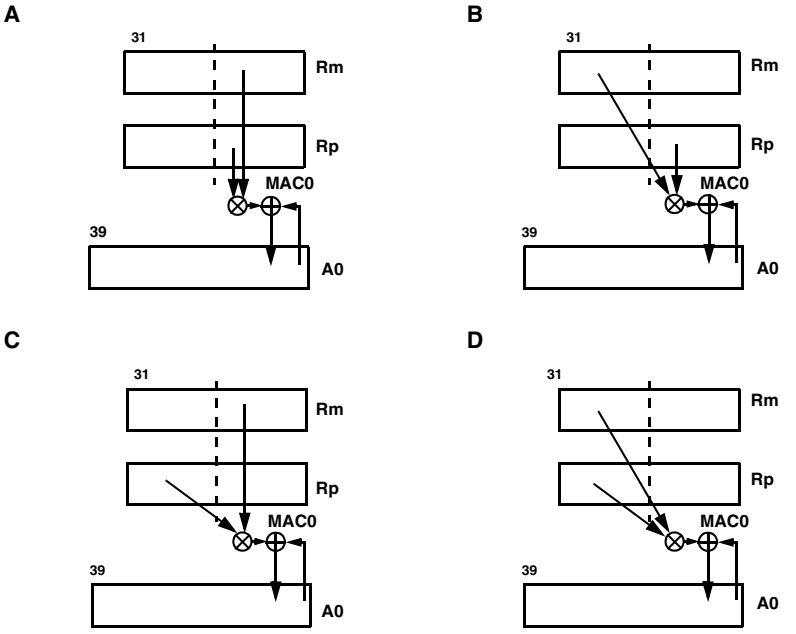


Figure 2-14. Four Possible Combinations of MAC Operations

One of the operands must be selected from the low half or the high half of one 32-bit word. The other operand must be selected from the low half or the high half of the other 32-bit word. Thus, each MAC is presented with four possible input operand combinations. The two 32-bit words can contain the same register information giving the options for squaring and multiplying the high half and low half of the same register. Figure 2-14 show these possible combinations.

The 32-bit product is passed to a 40-bit adder/subtracter, which may add or subtract the new product from the contents of the accumulator result register or pass the new product directly to the Data Register File results register. For results, the A0 and A1 registers are 40 bits wide. Each of these registers consists of smaller 32-bit and 8-bit registers: A0.W, A1.W, A0.X, and A1.X.

Some example instructions follow:

```
A0 = R3.L * R4.H (S) ;
```

The MAC0 multiplier/accumulator performs a multiply and puts the result in the accumulator register.

```
A1 += R3.H * R4.H (S) ;
```

The MAC1 multiplier/accumulator performs a multiply and accumulates the result with the previous results in the A1 accumulator.

## Multiply Without Accumulate

The multiplier may operate without the accumulation function. If accumulation is not used, the result can be directly stored in a register from the Data Register File or the Accumulator register. The destination register may be 16 bits or 32 bits. If a 16-bit destination register is a low half, then MAC0 is used; if it is a high half, then MAC1 is used. For a 32-bit destination register, either MAC0 or MAC1 is used.

If the destination register is 16-bits, then the word that is extracted from the multiplier depends on the data type of the input:

- If the multiplication uses fractional operands or the `IH` option, then the high half of the result is extracted and stored in the 16-bit destination registers (see Figure 2-15).
- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 16-bit destination registers. These extractions provide the most useful information in the resultant 16-bit word for the data type chosen (see Figure 2-16 on page 2-49).

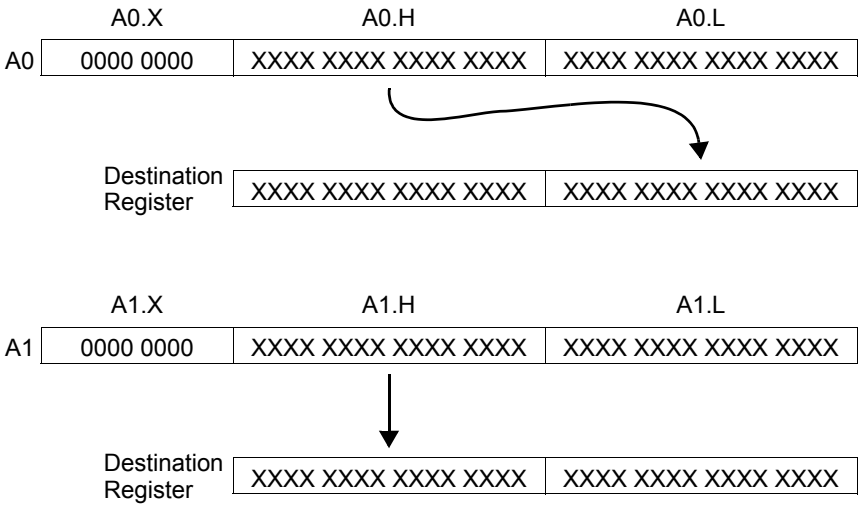


Figure 2-15. Multiplication of Fractional Operands

This example uses fractional, unsigned operands:

```
R0.L = R1.L * R2.L (FU) ;
```

The instruction deposits the upper 16 bits of the multiply answer with rounding and saturation into the lower half of `R0`, using MAC0.

This example uses unsigned integer operands:

```
R0.H = R2.H * R3.H (IU) ;
```

The instruction deposits the lower 16 bits of the multiply answer with any required saturation into the high half of `R0`, using MAC1.

```
R0 = R1.L * R2.L (S) ;
```

Regardless of operand type, the preceding operation deposits 32 bits of the multiplier answer with saturation into `R0`, using MAC0.

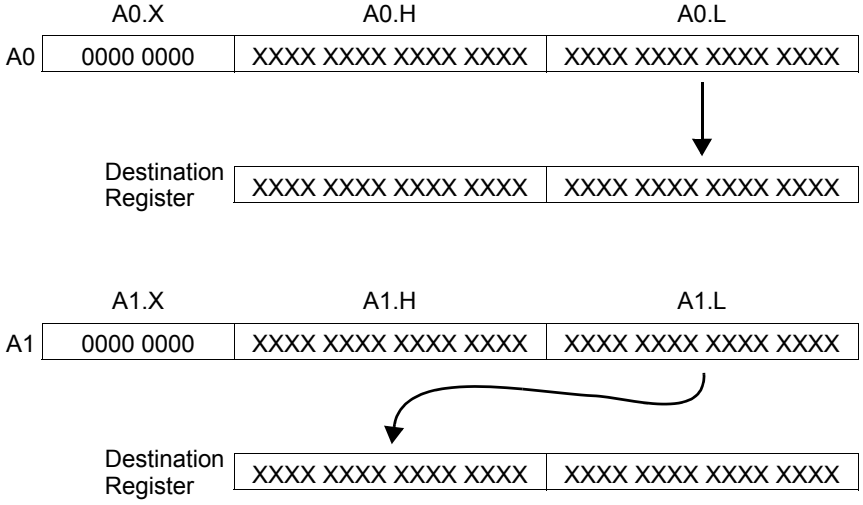


Figure 2-16. Multiplication of Integer Operands

## Special 32-Bit Integer MAC Instruction

The ADSP-21535 DSP supports a multicycle 32-bit MAC instruction, that is,

```
Dreg *= Dreg
```

The single instruction multiplies two 32-bit integer operands and provides a 32-bit integer result, destroying one of the input operands.

The instruction takes multiple cycles to execute. Refer to the product data sheet and *Blackfin DSP Instruction Set Reference* for more information about the exact operation of this instruction. This 'macro' function is interruptable and does not modify the data in either accumulator register `A0` or `A1`.

## Dual MAC Operations

The ADSP-21535 DSP has two 16-bit MACs. Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers are offered to each MAC unit, providing each with four possible combinations of 16-bit input operands. Dual MAC operations are frequently referred to as *Vector* operations, because a program could store vectors of samples in the four input operands and perform vector computations.

An example of a dual multiply and accumulate instruction is

```
A1 += R1.H * R2.L, A0 += R1.L * R2.H ;
```

This instruction represents two multiply and accumulate operations:

- In one operation, in MAC1, the high half of `R1` is multiplied by the low half of `R2` and added to the contents of the `A1` accumulator.
- In the second operation, in MAC0, the low half of `R1` is multiplied by the high half of `R2` and added to the contents of `A0`.

The results of the MAC operations may be written to registers in a number of ways: as a pair of 16-bit halves, as a pair of 32-bit registers, or as an independent 16-bit half register or 32-bit register.

For example:

```
R3.H = ( A1 += R1.H * R2.L ), R3.L = ( A0 += R1.L * R2.L ) ;
```

In this instruction, the 40-bit accumulator is packed into a 16-bit half register. The result from MAC1 must be transferred to a high half of a destination register and the result from MAC0 must be transferred to the low half of the same destination register.

The operand type determines the correct bits to extract from the accumulator and deposit in the 16-bit destination register. See "Multiply Without Accumulate" on page 2-47.

```
R3 = ( A1 += R1.H * R2.L ), R2=( A0 += R1.L * R2.L ) ;
```

In this instruction, the 40-bit accumulators are packed into two 32-bit registers. The registers must be register pairs ( `R[1:0]` ; `R[3:2]` ; `R[5:4]` ; `R[7:6]` )

```
R3.H = ( A1 += R1.H * R2.L ), A0 += R1.L * R2.L ;
```

This instruction is an example of one accumulator—but not the other—being transferred to a register. Either a 16- or 32-bit register may be specified as the destination register.

# Barrel Shifter (Shifter)

The shifter provides bitwise shifting functions for 16- or 32-bit inputs, yielding a 16-, 32-, or 40-bit output. These functions include arithmetic shift, logical shift, rotate, and various bit-test, set, pack, unpack and expo-

nent-detection functions. These shift functions can be combined to implement numerical format control, including full floating-point representation.

## Shifter Operations

The shifter instructions (>>>, >>, ASHIFT, LSHIFT, ROT) can be used various ways, depending on the underlying arithmetic requirements. ASHIFT and >>> represents the arithmetic shift. LSHIFT and >> represent the logical shift.

The arithmetic shift and logical shift operations can be further broken into subsections. Instructions that are intended to operate on 16-bit single or paired numeric values, as would occur in many DSP algorithms, can use the instructions ASHIFT and LSHIFT. These are typically three operand instructions.

Instructions that are intended to operate on a 32-bit register value and use two operands, such as instructions frequently used by a compiler, can use the >>> and >> instructions.

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter-related instructions, see "Shifter Instruction Summary" on page 2-56.

### Two Operand Shifts

Two operand shift instructions shift an input register and deposit the result in the same register.

### Immediate Shifts

An immediate shift instruction shifts the input bit pattern to the right (down-shift) or left (up-shift) by a given number of bits. Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value down-shifted.

```
R0 = 0x0000 B6A3 ;
R0 >>= 0x04 ;
```

results in

```
R0 = 0x0000 0B6A ;
```

The following example shows the input value up-shifted.

```
R0 = 0x0000 B6A3 ;
R0 <<= 0x04 ;
```

results in

```
R0 = 0x000B 6A30 ;
```

### Register Shifts

Register based shifts use a register to hold the shift value. The entire 32-bit register is used to derive the shift value, and when the magnitude of the shift is greater than 32, then the result is either 0 or –1.

The following example shows the input value up-shifted.

```
R0 = 0x0000 B6A3 ;
R2 = 0x0000 0004 ;
R0 <<= R2 ;
```

results in

```
R0 = 0x000B 6A30 ;
```

## Three Operand Shifts

Three operand shifter instructions shift an input register and deposit the result in a destination register.

### Immediate Shifts

Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value down-shifted.

```
R0 = 0x0000 B6A3 ;
R1 = R0 >> 0x04 ;
```

results in

```
R1 = 0x0000 0B6A ;
```

The following example shows the input value up-shifted.

```
R0.L = 0xB6A3 ;
R1.H = R0.L << 0x04 ;
```

results in

```
R1.H = 0x6A30 ;
```

### Register Shifts

Register based shifts use a register to hold the shift value. When a register is used to hold the shift value, for `ASHIFT`, `LSHIFT` or `ROT`, then the shift value is always found in the low half of a register (`Rn.L`). The bottom 6 bits of `Rn.L` are masked off and used as the shift value.

The following example shows the input value up-shifted.

```
R0 = 0x0000 B6A3 ;
R2.L = 0x0004 ;
R1 = R0 ASHIFT by R2.L ;
```

results in

```
R1 = 0x000B 6A30 ;
```

The following example shows the input value rotated. Assume the Condition Code (CC) bit is set to 0. For more information about CC, see "Condition Code Flag" on page 4-13.

```
R0 = 0xABCD EF12 ;
R2.L = 0x0004 ;
R1 = R0 ROT by R2.L ;
```

results in

```
R1 = 0xBCDE F125 ;
```

Note that the CC bit is included in the result, at bit 3.

## Bit Test, Set, Clear, Toggle

The shifter provides the method to test, set, clear, and toggle specific bits of a data register. All instructions have two arguments—the source register and the bit-field value. The test instruction does not change the source register. The result of the test instruction resides in the CC bit.

The following examples show a variety of operations.

```
BITCLR ( R0, 6 ) ;
BITSET ( R2, 9 ) ;
BITTGL ( R3, 2 ) ;
CC = BITTST ( R3, 0 ) ;
```

### Field Extract and Field Deposit

If the shifter is used, a source field may be deposited anywhere in a 32-bit destination field. The source field may be from 1 bit to 16 bits in length. In addition, a 1- to 16-bit field may be extracted from anywhere within a 32-bit source field.

Two register arguments are used for these functions. One holds the 32-bit destination or 32-bit source. The other holds the extract/deposit value, its length, and its position within the source.

## Shifter Instruction Summary

Table 2-10 lists the shifter instructions. For more information about assembly language syntax and the effect of shifter instructions on the status flags, see the *Blackfin DSP Instruction Set Reference*.

In Table 2-10, note the meaning of these symbols:

- Dreg denotes any Data Register File register.
- Dreg_lo denotes the lower 16 bits of any Data Register File register.
- Dreg_hi denotes the upper 16 bits of any Data Register File register.
- * Indicates the flag may be set or cleared, depending on results of instruction.
- * 0 Indicates versions of the instruction that send results to Accumulator `A0` set or clear `AV0`.
- * 1 Indicates versions of the instruction that send results to Accumulator `A1` set or clear `AV1`.
- ** indicates the flag is cleared.

- *** Indicates CC contains the latest value shifted into it.
- – Indicates no effect.

Table 2-10. Shifter Instruction Summary

| Instruction | ASTAT Status Flag | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | CC | V<br>VS |
| BITCLR ( Dreg, uimm5 ) ; | * | * | ** | – | – | – | **/– |
| BITSET ( Dreg, uimm5 ) ; | ** | * | ** | – | – | – | **/– |
| BITTGL ( Dreg, uimm5 ) ; | * | * | ** | – | – | – | **/– |
| CC= BITTST ( Dreg, uimm5 ) ; | – | – | – | – | – | * | – |
| CC= !BITTST ( Dreg, uimm5 ) ; | – | – | – | – | – | * | – |
| Dreg = DEPOSIT ( Dreg, Dreg ) ; | * | * | ** | – | – | – | **/– |
| Dreg = EXTRACT ( Dreg, Dreg ) ; | * | * | ** | – | – | – | **/– |
| BITMUX ( Dreg, Dreg, A0 ) ; | – | – | – | – | – | – | – |
| Dreg_lo = ONES Dreg ; | – | – | – | – | – | – | – |
| Dreg = PACK ( Dreg_lo_hi, Dreg_lo_hi) ; | – | – | – | – | – | – | – |
| Dreg >>>=uimm5 ; | * | * | – | – | – | – | * |
| Dreg >>=uimm5 ; | * | * | – | – | – | – | **/– |
| Dreg <<=uimm5 ; | * | * | – | – | – | – | **/– |
| Dreg = Dreg >>> uimm5 ; | * | * | – | – | – | – | * |
| Dreg = Dreg >> uimm5 ; | * | * | – | – | – | – | **/– |

Table 2-10. Shifter Instruction Summary (Cont'd)

| Instruction | ASTAT Status Flag | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | CC | V<br>VS |
| Dreg = Dreg << uimm5 ; | * | * | – | – | – | – | **/– |
| Dreg = Dreg >>> uimm4 (V) ; | * | * | – | – | – | – | * |
| Dreg = Dreg >> uimm4 (V) ; | * | * | – | – | – | – | **/– |
| Dreg = Dreg << uimm4 (V) ; | * | * | – | – | – | – | **/– |
| An = An >>>uimm5 ; | * | * | – | * 0 | * 1 | – | – |
| An = An >>uimm5 ; | * | * | – | * 0 | * 1 | – | – |
| An = An <<uimm5 ; | * | * | – | * 0 | * 1 | – | – |
| Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ; | * | * | – | – | – | – | * |
| Dreg_lo_hi = Dreg_lo_hi >> uimm4 ; | * | * | – | – | – | – | **/– |
| Dreg_lo_hi = Dreg_lo_hi << uimm4 ; | * | * | – | – | – | – | **/– |
| Dreg >>>= Dreg ; | * | * | – | – | – | – | * |
| Dreg >>= Dreg ; | * | * | – | – | – | – | **/– |
| Dreg <<= Dreg ; | * | * | – | – | – | – | **/– |
| Dreg = ASHIFT Dreg BY Dreg_lo ; | * | * | – | – | – | – | * |
| Dreg = LSHIFT Dreg BY Dreg_lo ; | * | * | – | – | – | – | **/– |
| Dreg = ROT Dreg BY imm6 ; | – | – | – | – | – | *** | – |
| Dreg = ASHIFT Dreg BY Dreg_lo (V) ; | * | * | – | – | – | – | * |

Table 2-10. Shifter Instruction Summary (Cont'd)

| Instruction | ASTAT Status Flag | | | | | | |
|---|---|---|---|---|---|---|---|
| | AZ | AN | AC0<br>AC1 | AV0<br>AV0S | AV1<br>AV1S | CC | V<br>VS |
| Dreg = LSHIFT Dreg BY Dreg_lo (V) ; | * | * | – | – | – | – | **/– |
| Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ; | * | * | – | – | – | – | * |
| Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ; | * | * | – | – | – | – | **/– |
| An = An ASHIFT BY Dreg _lo ; | * | * | – | * 0 | * 1 | – | – |
| An = An ROT BY imm6 ; | – | – | – | – | – | *** | – |
| Preg = Preg >> 1 ; | – | – | – | – | – | – | – |
| Preg = Preg >> 2 ; | – | – | – | – | – | – | – |
| Preg = Preg << 1 ; | – | – | – | – | – | – | – |
| Preg = Preg << 2 ; | – | – | – | – | – | – | – |
| Dreg = ( Dreg + Dreg ) << 1 ; | * | * | * | – | – | – | * |
| Dreg = ( Dreg +Dreg ) << 2 ; | * | * | * | – | – | – | * |
| Preg = ( Preg +Preg ) << 1 ; | – | – | – | – | – | – | – |
| Preg = ( Preg +Preg ) << 2 ; | – | – | – | – | – | – | – |
| Preg = Preg + ( Preg << 1 ) ; | – | – | – | – | – | – | – |
| Preg = Preg + ( Preg << 2 ) ; | – | – | – | – | – | – | – |