2 COMPUTE BLOCK REGISTERS

The ADSP-TS201 TigerSHARC processor core contains two compute blocks—compute block X and compute block Y. Each block contains a register file and four independent computation units. The units are the ALU, multiplier, shifter, and CLU. Because the execution of all computational instructions in the ADSP-TS201 processor depends on the input and output data formats and depends on whether the instruction is executed on one computational block or both, it is important to understand how to use the TigerSHARC processor's compute block registers. This chapter describes the registers in the compute blocks, shows how the register name syntax controls data format and execution location, and defines the available data formats.

This chapter describes:

- "Register File Registers" on page 2-5
- "Numeric Formats" on page 2-15

A general-purpose, multiport, 32-word data register file in each compute block serves for transferring data between the computation units and the data buses and stores intermediate results. Figure 2-1 shows how each of the register files provide the interface between the internal buses and the computational units within the compute blocks.

As shown in Figure 2-1, data input to the register file may pass through the data alignment buffer (DAB). The DAB is a two quad-word FIFO that provides aligned data for registers when dual- or quad-register loads receive misaligned data from memory. For more information on using the DAB, see "IALU" on page 7-1.

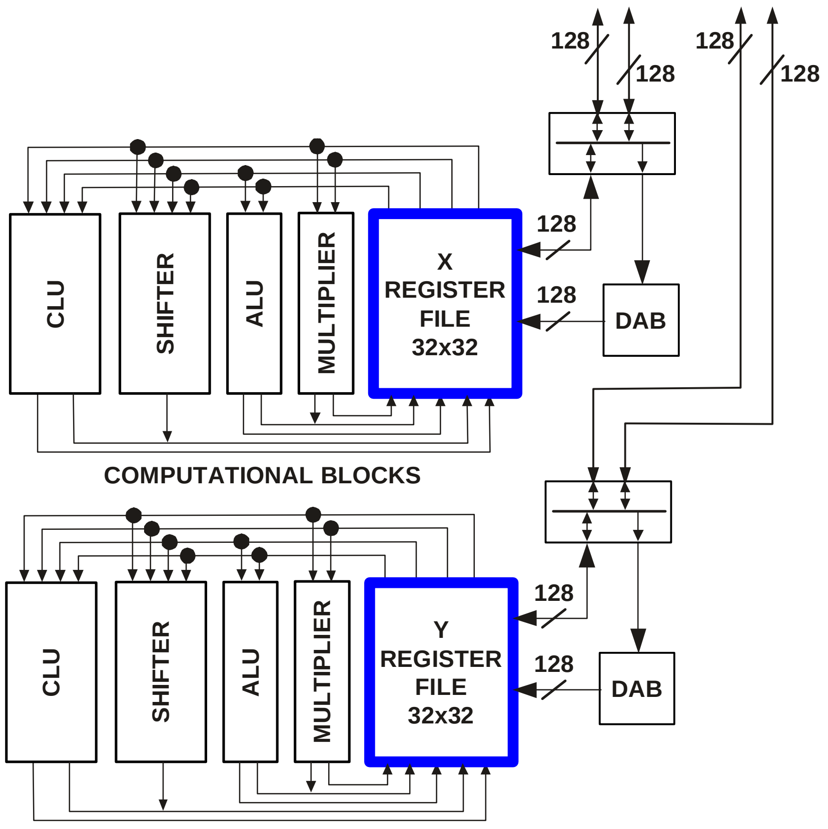


Figure 2-1. Data Register Files in Compute Block X and Y

Within the compute block, there are two types of registers—memory-mapped registers and non-memory-mapped registers. The memory mapped registers in each of the compute blocks are the general-purpose data register file registers `XR31-0` and `YR31-0`. Because these registers are memory mapped, they are accessible to external bus devices.

For operations within a single processor, the distinction between memory-mapped and non-memory-mapped compute block registers is important because the memory-mapped registers are *Universal registers* (*`Ureg`*). These registers are considered universal because *`Ureg`* registers may be used for many types of operations and may be used in operations outside the portion of the processor core where the `Ureg` register resides. The compute block `Ureg` registers can be used for additional operations unavailable to other `Ureg` registers. To distinguish the compute block register file registers from other `Ureg` registers, the `XR31-0` and `YR31-0` registers are also referred to as *Data registers* (`Dreg`).

For operations in a multiprocessing system, it is very useful that 90% of the registers in the TigerSHARC processor are memory-mapped registers. The memory-mapped registers have absolute addresses associated with them, meaning that they can be accessed by other processors through multiprocessor space or accessed by any other bus masters in the system.

ⓘ A processor can access its own registers by using the multiprocessor memory space, but the processor would have to tie up the external bus to access its own registers this way.

The compute blocks have a few registers that are non-memory mapped. These registers do not have absolute addresses associated with them. The non-memory-mapped registers are special registers that are dedicated for special instructions in each compute block. The unmapped registers in the compute blocks include:

- Compute block status (`XSTAT` and `YSTAT`) registers
- Parallel Result (`XPR1-0` and `YPR1-0`) registers—ALU
- Multiplier Result (`XMR3-0` and `YMR3-0`) and Multiplier Result Overflow (`XMR4` and `YMR4`) registers—Multiplier
- Bit FIFO Overflow Temporary (`XBFOTMP` and `YBFOTMP`) registers—Shifter
- Trellis Result (`XTR34-0` and `YTR34-0`), Trellis History Result (`xTHR3:0` and `yTHR3:0`), and Comminication Control (`CMCTL`) register—CLU

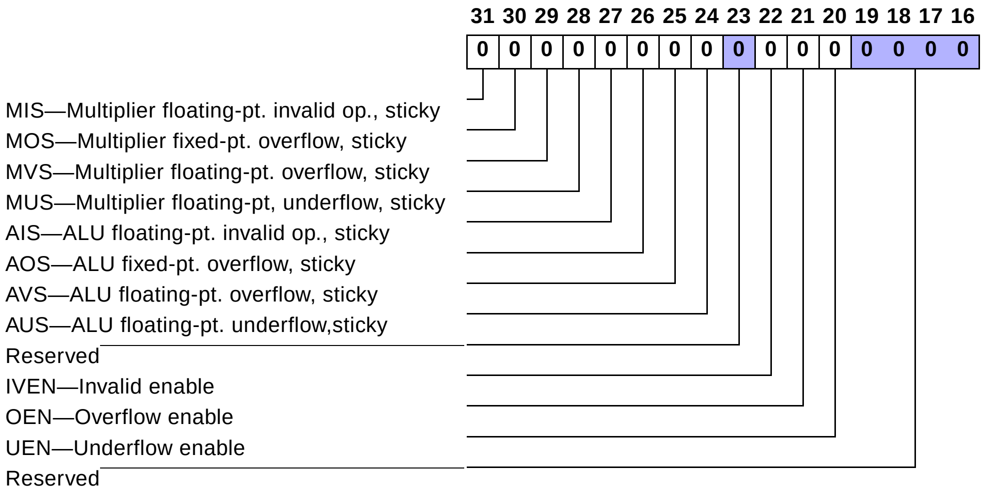

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
MIS—Multiplier floating-pt. invalid op., sticky
MOS—Multiplier fixed-pt. overflow, sticky
MVS—Multiplier floating-pt. overflow, sticky
MUS—Multiplier floating-pt, underflow, sticky
AIS—ALU floating-pt. invalid op., sticky
AOS—ALU fixed-pt. overflow, sticky
AVS—ALU floating-pt. overflow, sticky
AUS—ALU floating-pt. underflow,sticky
Reserved
IVEN—Invalid enable
OEN—Overflow enable
UEN—Underflow enable
Reserved

Figure 2-2. XSTAT/YSTAT (Upper) Register Bit Descriptions

The non-memory-mapped registers serve special purposes in each compute block. The X/YSTAT registers (shown in Figure 2-2 and Figure 2-3, also see "XSTAT/YSTAT Register" on page 10-189) hold the status flags for each compute block. These flags are set or reset to indicate the status of an instruction's execution a compute block's ALU, multiplier, shifter, and CLU. The X/YPR1-0 registers hold parallel results from the ALU's SUM, ABS, VMAX, and VMIN instructions. The X/YMR3-0 registers optionally hold results from fixed-point multiply operations, and the X/YMR4 register holds overflow from those operations. The X/YBFOTMP registers temporarily store or return overflow from GETBITS and PUTBITS instructions.

The X/YTR34-0 and X/YTHR3:0 registers on previous TigerSHARC processors held data related to Trellis instructions. In the ADSP-TS201 processor, the trellis registers are also used for DESPREAD and XCORRS instructions. The Comminication Control (CMCTL) register serves as an optional source for the CUT value in the XCORRS instruction.

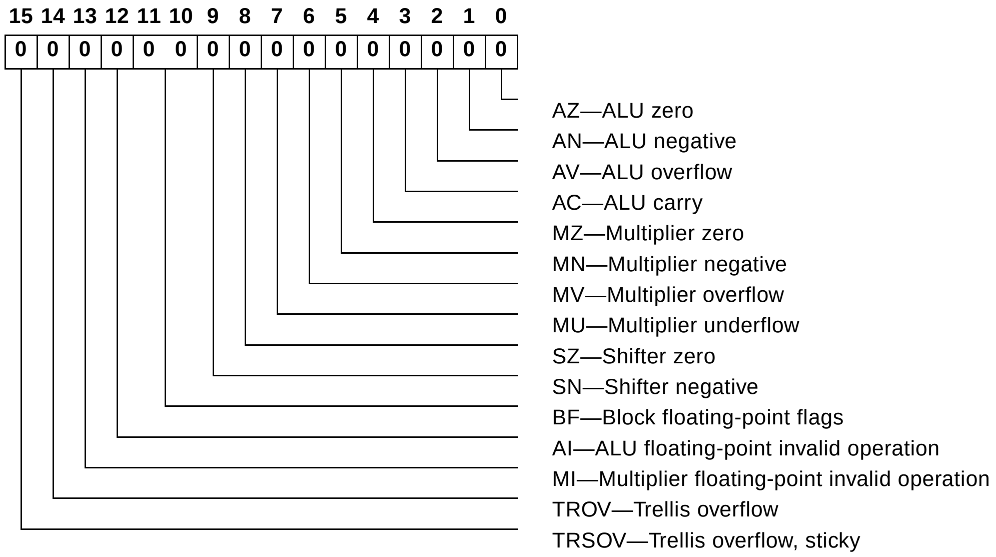


Figure 2-3. XSTAT/YSTAT (Lower) Register Bit Descriptions

Register File Registers

The compute block X and Y register files contain thirty-two 32-bit registers, which serve as a compute block's interface between processor's internal bus and the computational units. The register file registers—`XR31-0` and `YR31-0`—are both universal registers (*`Ureg`*) and data registers (*`Dreg`*).

Inputs for computations usually come from the register file. However, in some cases, inputs may come from compute block's internal registers. The results go to the register file, except when they go to the block's internal registers.

It is important to note that a register may be used once in an input *instruction slot*, but the assembly syntax permits using registers multiple times within an *instruction line* (which contains up to four instruction slots). The register file registers are hardware interlocked, meaning that there is dependency checking during each computation to make sure the correct values are being used. When a computation accesses a register, the processor performs a register check to make sure there are no other dependencies on that register. For more information on instruction lines and dependencies, see "Instruction Line Syntax and Structure" on page 1-22 and "Instruction Parallelism Rules" on page 1-26.

There are many ways to name registers in the TigerSHARC processor assembly syntax. The register name syntax provides selection of many features of computational instructions. Using the register name syntax in an instruction, you can specify:

- Compute block selection
- Register width selection
- Operand size selection
- Data format selection

Figure 2-4 shows the parts of the register name syntax and the features that the syntax selects.

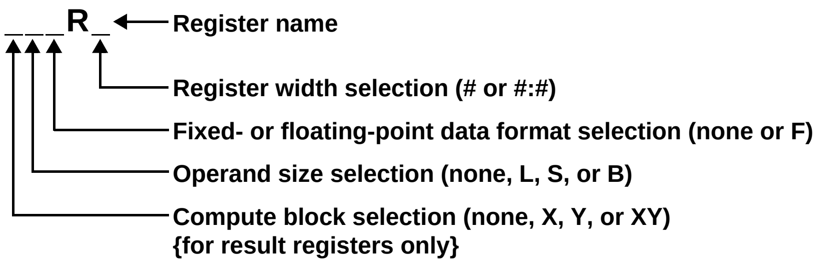

_ _ _R_
Register name
Register width selection (# or #:#)
Fixed- or floating-point data format selection (none or F)
Operand size selection (none, L, S, or B)
Compute block selection (none, X, Y, or XY)
{for result registers only}

Figure 2-4. Register File Register Name Syntax

The processor's assembly syntax also supports selection of integer or fractional and real or complex data types. These selections are provided as options to instructions and are not part of register file register name syntax.

Compute Block Selection

As shown in Figure 2-4, the assembly syntax for naming registers lets you select the compute block of the register with which you are working.

The X and Y register-name prefixes denote in which compute block the register resides: X = compute block X only, Y = compute block Y only, and XY (or no prefix) = both. The following ALU instructions provide some register name syntax examples.

```
XR0 = R1 + R2 ;; /* This instruction executes in block X */
```
This instruction uses registers XR0, XR1, and XR2.

```
YR1 = R5 + R6 ;; /* This instruction executes in block Y */
```
This instruction uses registers YR1, YR5, and YR6.

```
XYR0 = R0 + R2 ;; /* This instruction executes in block X & Y */
```
This instruction uses registers XR0, XR2, YR0, and YR2.

```
R0 = R22 + R3 ;; /* This instruction executes in block X & Y */
```
This instruction uses registers XR0, XR22, XR3, YR0, YR22, and YR3.

Because the compute block prefix lets you select between executing the instruction in one or both compute blocks, this prefix provides the selection between *Single-Instruction, Single-Data* (SISD) execution and *Single-Instruction, Multiple-Data* (SIMD) execution. Using SIMD execution is a powerful way to optimize execution if the same algorithm is being used to process multiple channels of data.

It is important to note that SISD and SIMD are not modes that are turned on or off with some latency in the change. SISD and SIMD execution are always available as execution options simply through register name selection.

To represent optional items, instruction syntax definitions use curley braces { } around the item. To represent choices between items, instruction syntax definitions place a vertical bar | between items. The following syntax definition example and comparable instruction indicates the difference for compute block selection:

```
{X|Y|XY}Rs = Rm + Rn ;;
/* the curly braces enclose options */
/* the vertical bars separate choices */

XYR0 = R1 + R0 ;;
/* code, no curly braces — no vertical bars */
```

Register Width Selection

As shown in Figure 2-4 on page 2-6, the assembly syntax for naming registers lets you select the width of the register with which you are working.

Each individual register file register (`XR31-0` and `YR31-0`) is 32 bits wide. To support data sizes larger than a 32-bit word, the processor's assembly syntax lets you combine registers to hold larger words. The register name syntax for register width works as follows:

- *`Rs`*, *`Rm`*, or *`Rn`* indicates a *Single register* containing a 32-bit word (or smaller).

  For example, these are register names such as `R1`, `XR2`, and so on.

- `Rsd`, `Rmd`, or `Rnd` indicates a *Double register* containing a 64-bit word (or smaller).

  For example, these are register names such as `R1:0`, `XR3:2`, and so on. The lower register must be evenly divisible by two.

- `Rsq`, `Rmq`, or `Rnq` indicates a *Quad register* containing a 128-bit word (or smaller).

  For example, these are register names such as `R3:0`, `XR7:4`, and so on. The lowest register must be evenly divisible by 4.

The combination of italic and code font in the register name syntax above indicates a user-substitutable value. Instruction syntax definitions use this convention to represent multiple register names. The following syntax definition example and comparable instruction indicates the difference for register width selection.

```
{X|Y|XY}Rsd = Rmd + Rnd ;;
/* replaceable register names, italics are variables */

XR1:0 = R3:2 + R1:0 ;;
/* code, no substitution */
```

Operand Size and Format Selection

As shown in Figure 2-4 on page 2-6, the assembly syntax for naming registers lets you select the operand size and fixed- or floating-point format of the data placed within the register with which you are working.

Single, double, and quad register file registers (`Rs`, `Rsd`, `Rsq`) hold *operands* (inputs and outputs) for instructions. Depending on the operand size and fixed- or floating-point format, there may be more that one operand in a register.

To select the operand size within a register file register, a register name prefix selects *a size that is equal or less than the size of the register*. These operand size prefixes for fixed-point data work as follows.

- B — Indicates Byte (8-bit) word data. The data in a single 32-bit register is treated as four 8-bit words. Example register names with byte word operands are `BR1`, `BR1:0`, and `BR3:0`.
- S — Indicates Short (16-bit) word data. The data in a single 32-bit register is treated as two 16-bit words. Example register names with short word operands are `SR1`, `SR1:0`, and `SR3:0`.
- None — Indicates Normal (32-bit) word data. Example register names with normal word operands are `R0 R1:0`, and `R3:0`.
- L — Indicates Long (64-bit) word data. An example register name with a long word operand is `LR1:0`.

The B, S, and L options apply for ALU and Shifter operations. Operand size selection differs slightly for the multiplier. For more information, see "Multiplier Operations" on page 5-5.

To distinguish between fixed- and floating-point data, the register name prefix F indicates that the register contains floating-point data. The processor supports the following floating-point data formats.

- None — Indicates fixed-point data
- *FRs*, *FRm*, or *FRn* (floating-point data in a single register) — Indicates normal (IEEE format, 32-bit) word data. An example register name with a normal word, floating-point operand is `FR3`.
- *FRsd*, *FRmd*, or *FRnd* (floating-point data in a double register) — Indicates extended (40-bit) word data. An example register name with an extended word, floating-point operand is `FR1:0`.

It is important to note that the operand size influences the execution of the instruction. For example, `SRsd = Rmd + Rnd;;` is an addition of four short data operands, stored in two register pairs. An example of this type of instruction follows and has the results shown in Figure 2-5.

```
SR1:0 = R31:30 + R25:24;;
```

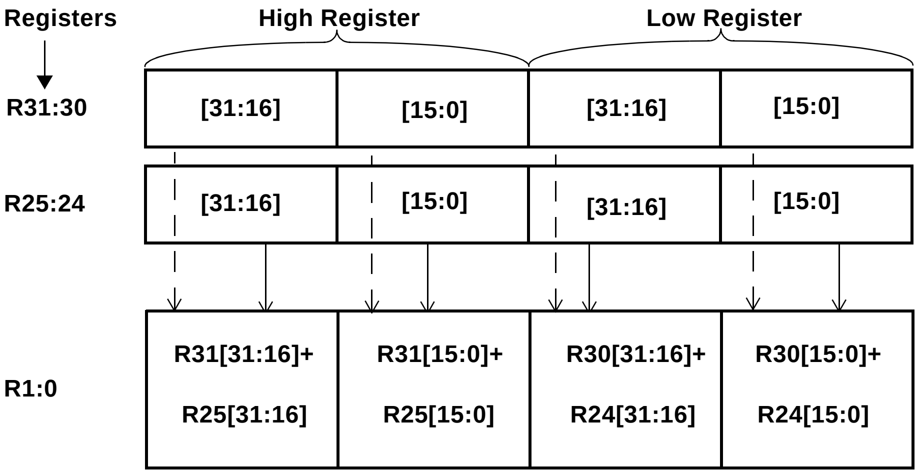


Figure 2-5. Addition of Four Short Word Operands in Double Registers

As shown in Figure 2-5, this instruction executes the operation on all 64 bits in this example. The operation is executed on every group of 16 bits separately.

Registers File Syntax Summary

Data register file registers are used in computational instructions and memory load/store instructions. The syntax for those instructions is described in:

- "ALU" on page 3-1
- "CLU" on page 4-1
- "Multiplier" on page 5-1
- "Shifter" on page 6-1

The following ALU instruction syntax description shows the conventions that all syntax descriptions use for data register file names.

```
{X|Y|XY}{F}Rsd = Rmd + Rnd ;;
```

where:

- `{X|Y|XY}` — The X, Y, or XY (none is same as XY) prefix on the register name selects the compute block or blocks to execute the instruction. The curly braces around these items indicate they are optional, and the vertical bars indicate that only one may be chosen.
- `{F}` — The F prefix on the register name selects floating-point format for the operation. Omitting the prefix selects fixed-point format.
- *Rsd* — The result is a double register as indicated by the *d*. The register name takes the form *R#:#*, where the lower number is evenly divisible by two (as in `R1:0`).
- *Rmd*, *Rnd* — The inputs are double registers. The *m* and *n* indicate that these must be different registers.

Here are some examples of register naming. In Figure 2-6, the register name XBR3 indicates the operation uses four fixed-point 8-bit words in the X compute block R3 data register. In Figure 2-7, the register name XSR3 indicates the operation uses two fixed-point 16-bit words in the X compute block R3 data register. In Figure 2-8, the register name XR3 indicates the operation uses one fixed-point 32-bit word in the X compute block R3 data register. In Figure 2-8, the register name XFR3 indicates floating-point data.

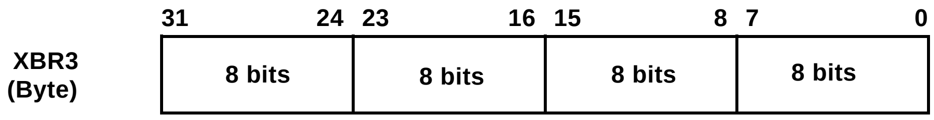


Figure 2-6. Register R3 in Compute Block X, Treated as Byte Data

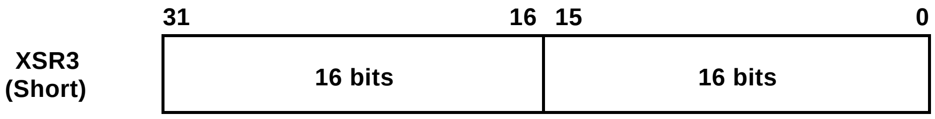


Figure 2-7. Register R3 in Compute Block X, Treated as Short Data

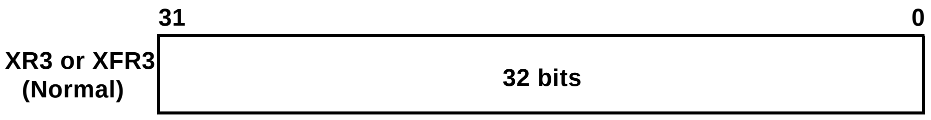


Figure 2-8. Register R3 in Compute Block X, Treated as Normal Data

Here are additional examples of register naming. Figure 2-9, Figure 2-10, and Figure 2-11 show examples of operand size in double registers, which are similar to the examples in Figure 2-6, Figure 2-7, and Figure 2-8.

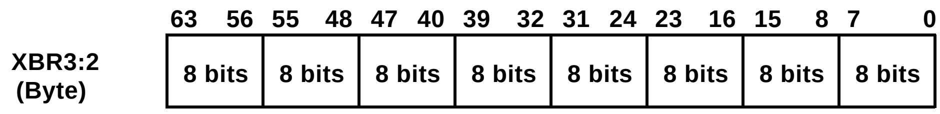


Figure 2-9. Register R3:2 in Compute Block X, Treated as Byte Data

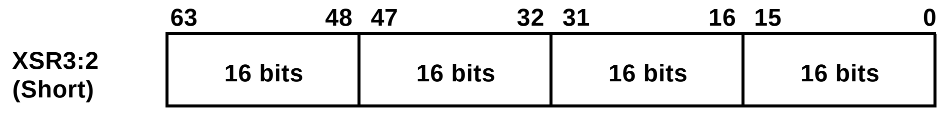


Figure 2-10. Register R3:2 in Compute Block X, Treated as Short Data

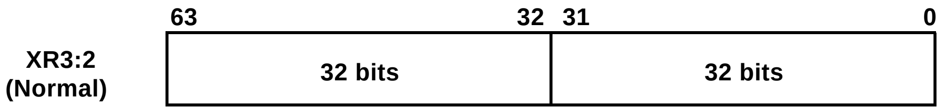


Figure 2-11. Register R3:2 in Compute Block X, Treated as Normal Data

The examples in Figure 2-12 and Figure 2-13 refer to two registers, but hold a single data word.

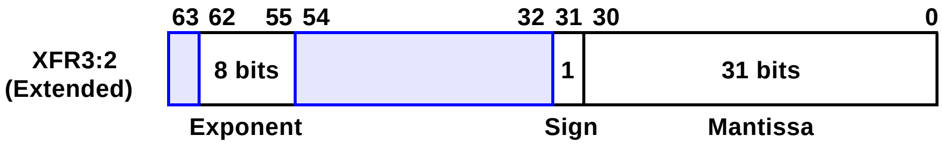


Figure 2-12. Register R3:2 in Compute Block X, Treated as Extended (Floating-Point) Data

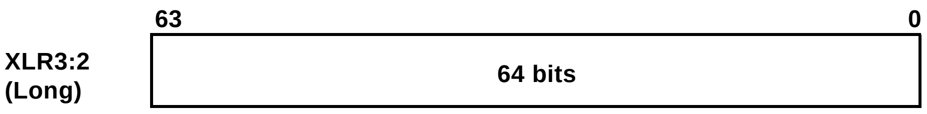


Figure 2-13. Register R3:2 in Compute Block X, Treated as Long Data

Numeric Formats

The processor supports the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the processor supports a 40-bit extended-precision version of the same format with eight additional bits in the mantissa. The processor also supports 8-, 16-, 32-, and 64-bit fixed-point formats—fractional and integer—which can be signed (two's-complement) or unsigned.

IEEE Single-Precision Floating-Point Data Format

IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in Figure 2-14. A number in this format consists of a sign bit *s*, a 24-bit significand, and an 8-bit unsigned-magnitude exponent *e*.

For normalized numbers, the significand consists of a 23-bit fraction *f* and a hidden bit of 1 that is implicitly presumed to precede f22 in the significand. The binary point is presumed to lie between this hidden bit and f22. The least significant bit (LSB) of the fraction is f0; the LSB of the exponent is e0.

The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. This bit also insures that the significand of any number in the IEEE normalized number format is always greater than or equal to 1 and less than 2.

The unsigned exponent *e* can range between 1 ≤ *e* ≤ 254 for normal numbers in the single-precision format. This exponent is biased by +127 (254/2). To calculate the true unbiased exponent, 127 must be subtracted from *e*.

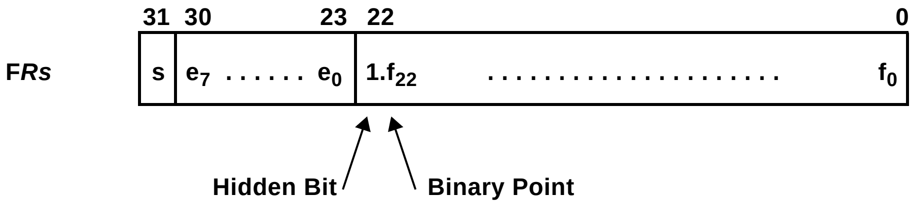


Figure 2-14. IEEE 32-Bit Single-Precision Floating-Point Format (Normal Word)

The IEEE standard also provides for several special data types in the single-precision floating-point format:

- An exponent value of 255 (all ones) with a nonzero fraction is a Not-A-Number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as 0 * ∞.
- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the number is signed, both positive and negative Infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with Infinity, both positive zero and negative zero can be represented.

The IEEE single-precision floating-point data types supported by the processor and their interpretations are summarized in Table 2-1.

Table 2-1. IEEE Single-Precision Floating-Point Data Types

Type	Exponent	Fraction	Value
NAN	255	Nonzero	Undefined
Infinity	255	0	$(-1)^s$ Infinity
Normal	$1 \le e \le 254$	Any	$(-1)^s (1.f_{22-0}) 2^{e-127}$
Zero	0	0	$(-1)^s$ Zero

The ADSP-TS201 processor is compatible with the IEEE single-precision floating-point data format in all respects, except for:

- The ADSP-TS201 processor does not provide inexact flags.
- NAN inputs generate an invalid exception and return a quiet NAN.
- Denormal operands are flushed to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation is flushed to zero and an underflow exception is generated.
- Round-to-nearest and round-towards-zero are supported. Round-to-±infinity are not supported.

Extended Precision Floating-Point Format

The extended precision floating-point format is 40 bits wide, with the same 8-bit exponent as in the standard format but with a 32-bit significand. This format is shown in Figure 2-15. In all other respects, the extended floating-point format is the same as the IEEE standard format.

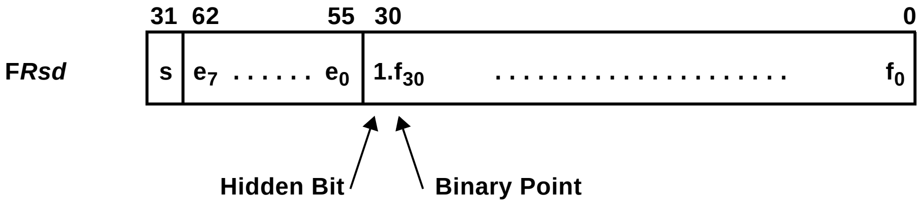


Figure 2-15. 40-Bit Extended-Precision Floating-Point Format (Extended Word)

Fixed-Point Formats

The processor supports fixed-point fractional and integer formats for 16-, 32-, and 64-bit data. In these formats, numbers can be signed (two's-complement) or unsigned. The possible combinations are shown in Figure 2-16 through Figure 2-31. In the fractional format, there is an implied binary point to the left of the most significant magnitude bit. In integer format, the binary point is understood to be to the right of the LSB. Note that the sign bit is negatively weighted in a two's-complement format.

The processor supports a fixed-point, signed, integer format for 8-bit data. Data in the 8- and 16-bit formats is always *packed* in 32-bit registers as follows—a single register holds four 8-bit or two 16-bit words, a dual register holds eight 8-bit or four 16-bit words, and a quad register holds sixteen 8-bit or eight 16-bit words.

ALU outputs usually have the same width and data format as the inputs. The multiplier, however, produces a 32/64-bit product from two 32-bit operands or, for 32/16-bit products, from eight 16-bit operands. If operands are unsigned (integers or fractions), the result is unsigned. These formats are shown in Figure 2-26, Figure 2-27, Figure 2-30 and Figure 2-31.

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed. If inputs are signed fractions, the result is automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative fraction multiplied by itself.) Thus, the left shift normally removes a redundant sign bit, increasing the precision of the most significant product.

Also, if the data format is fractional, a single bit left shift renormalizes the MSB to a fractional format. The signed formats with and without left shifting are shown in Figure 2-28 and Figure 2-29.

The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. For more information on the multiplier and accumulator, see "Multiplier" on page 5-1.

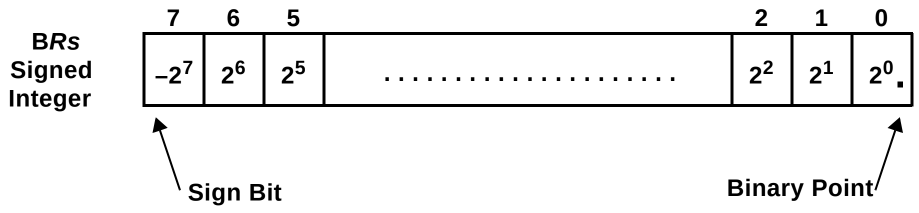


Figure 2-16. 8-Bit Fixed-Point Format, Signed Integer (Byte Word)

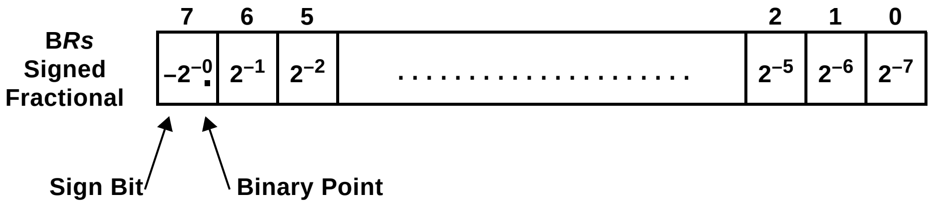


Figure 2-17. 8-Bit Fixed-Point Format, Signed Fractional (Byte Word)

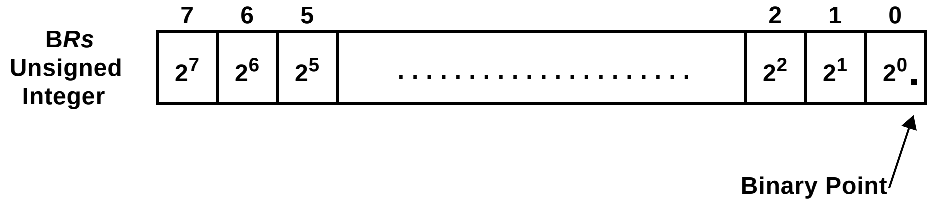


Figure 2-18. 8-Bit Fixed-Point Format, Unsigned Integer (Byte Word)

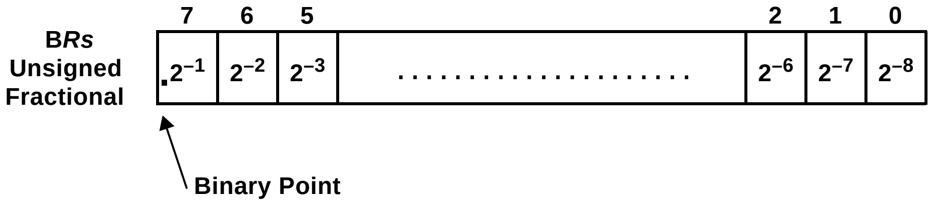


Figure 2-19. 8-Bit Fixed-Point Format, Unsigned Fractional (Byte Word)

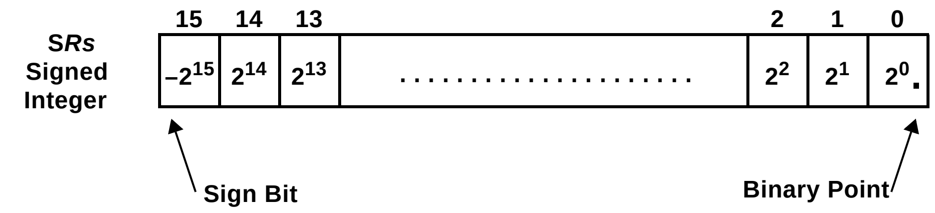


Figure 2-20. 16-Bit Fixed-Point Format, Signed Integer (Short Word)

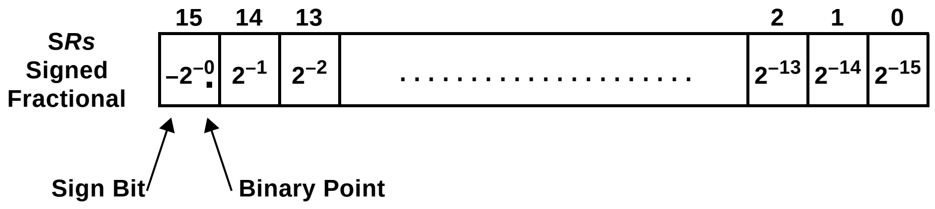


Figure 2-21. 16-Bit Fixed-Point Format, Signed Fractional (Short Word)

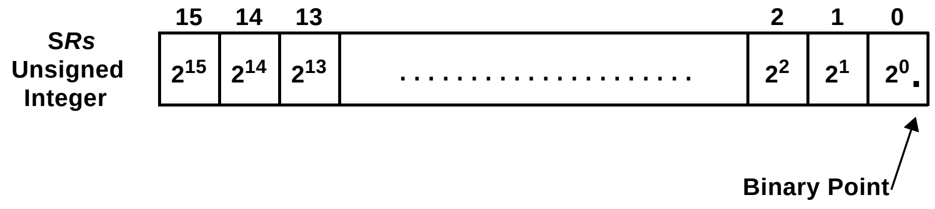


Figure 2-22. 16-Bit Fixed-Point Format, Unsigned Integer (Short Word)

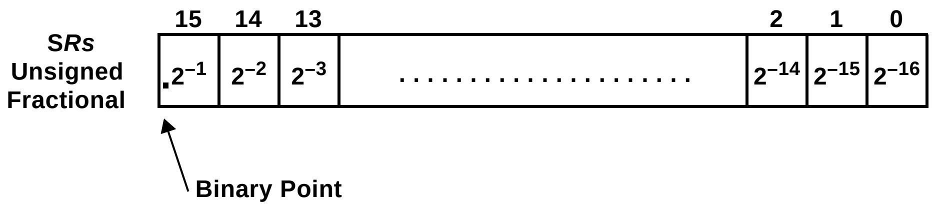


Figure 2-23. 16-Bit Fixed-Point Format, Unsigned Fractional (Short Word)

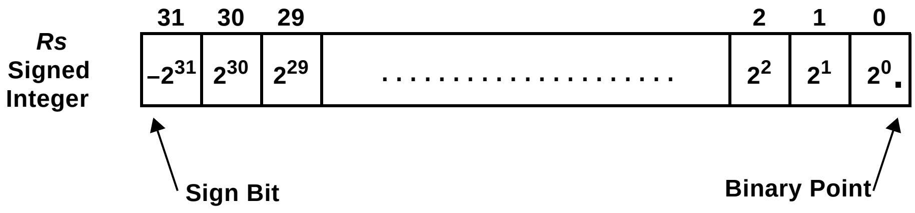


Figure 2-24. 32-Bit Fixed-Point Format, Signed Integer (Normal Word)

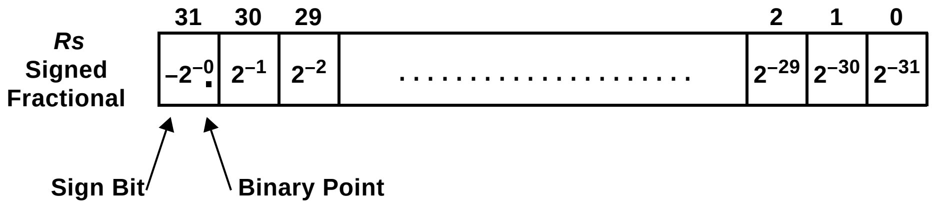


Figure 2-25. 32-Bit Fixed-Point Format, Signed Fractional (Normal Word)

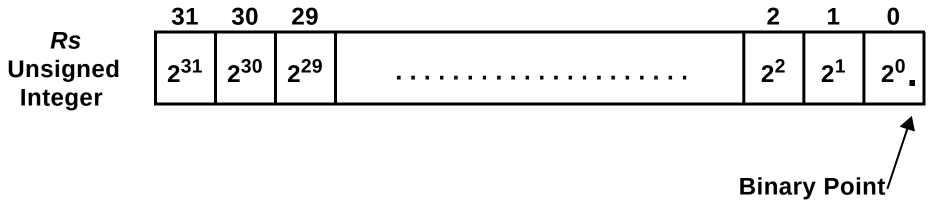


Figure 2-26. 32-Bit Fixed-Point Format, Unsigned Integer (Normal Word)

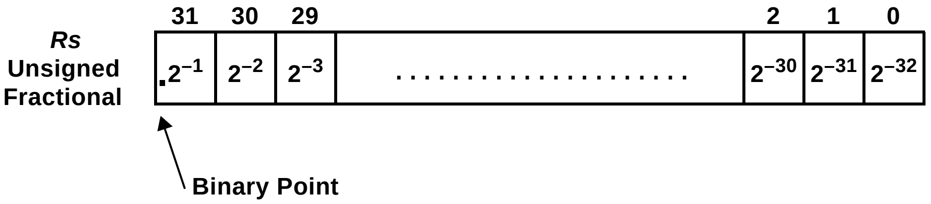


Figure 2-27. 32-Bit Fixed-Point Format, Unsigned Fractional (Normal Word)

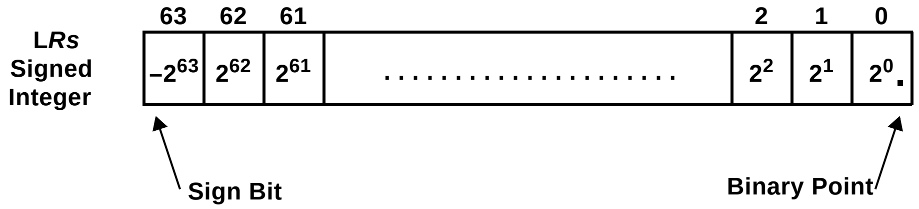


Figure 2-28. 64-Bit Fixed-Point Format, Signed Integer (Long Word)

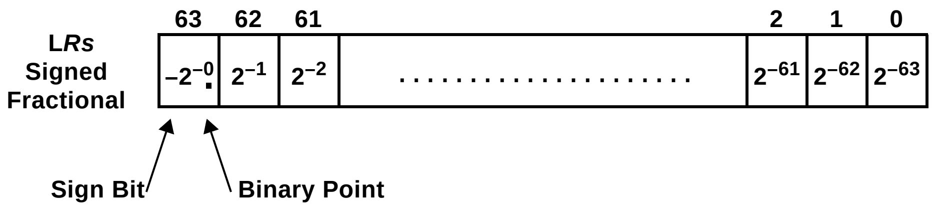


Figure 2-29. 64-Bit Fixed-Point Format, Signed Fractional (Long Word)

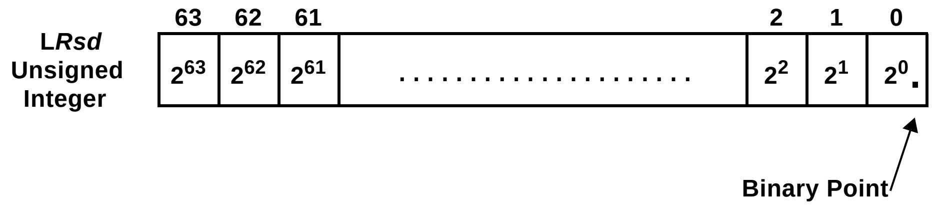


Figure 2-30. 64-Bit Fixed-Point Format, Unsigned Integer (Long Word)

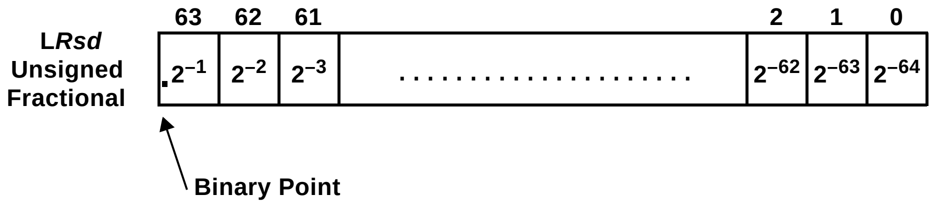


Figure 2-31. 64-Bit Fixed-Point Format, Unsigned Fractional (Long Word)

3 ALU

The ADSP-TS201 TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The Arithmetic Logic Unit (ALU) is highlighted in Figure 3-1. The ALU takes its inputs from the register file, and returns its outputs to the register file.

This unit performs all *arithmetic operations* (addition/subtraction) for the processor on data in fixed-point and floating-point formats and performs *logical operations* for the processor on data in fixed-point formats. The ALU also executes *data conversion operations* such as expand/compact on data in fixed-point formats.

Not all ALU operations can be applied to both fixed- and floating-point data. Relating ALU operations and supported data types shows that the 64-Bit ALU unit within each compute block supports:

- Fixed- and floating-point *arithmetic operations* — add (`+`), subtract (`-`), minimum (`MIN`), maximum (`MAX`), Viterbi maximum (`VMAX`), comparison (`COMP`), clipping (`CLIP`), and absolute value (`ABS`)
- Fixed-point only *arithmetic operations* — increment (`INC`), decrement (`DEC`), sideways add (`SUM`), parallel result of sideways add (`PRx=SUM`), one's complement (`ONES`), and bit FIFO pointer increment (`BFOINC`)
- Floating-point only *arithmetic operations* — floating-point conversion (`FLOAT`), fixed-point conversion (`FIX`), copy sign (`COPYSIGN`), scaling (`SCALB`), inverse or division seed (`RECIPS`), square root or

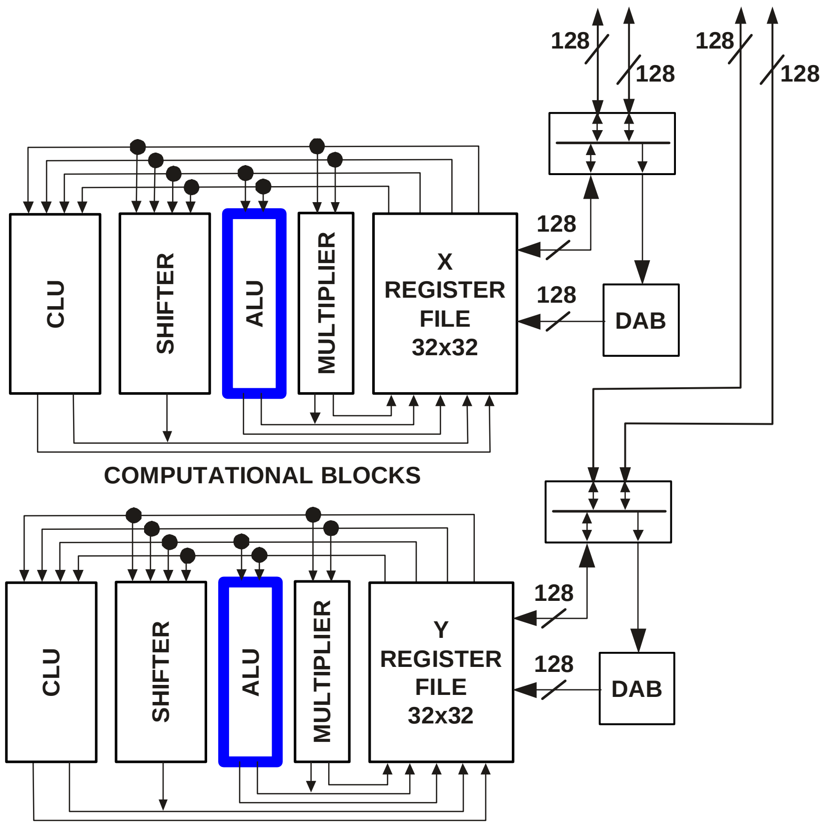


Figure 3-1. ALUs in Compute Block X and Y

inverse square root seed (`RSQRTS`), extract mantissa (`MANT`), extract exponent (`LOGB`), extend operand (`EXTD`), and translate extended to a single operand (`SNGL`)

- Fixed-point only *logical operations* — `AND`, `AND NOT`, `OR`, `XOR`, and `PASS`
- Fixed-point only *data conversion (promotion/demotion) operations* — expand (`EXPAND`), compact (`COMPACT`), and merge (`MERGE`)

Examining the supported operands for each operation shows that the ALU operations support these data types:

- Fixed-point *arithmetic operations* and *logical operations* support:
    - 8-bit (byte) input operands
    - 16-bit (short) input operands
    - 32-bit (normal) input operands
    - 64-bit (long) input operands
    - output 8-, 16-, 32- or 64-bit results
- Floating-point *arithmetic operations* support:
    - 32-bit (normal) input operands (IEEE standard)
    - 40-bit (extended) input operands
    - output 32- or 40-bit results
- Fixed-point *data conversion operations* support:
    - 8-bit (byte) input operands
    - 16-bit (short) input operands
    - 32-bit (normal) input operands
    - 64-bit (long) input operands
    - 128-bit (quad) input operands
    - output 8-, 16-, 32-, 64-, or 128-bit results; 128-bit input and output operands only apply for `EXPAND` and `COMPACT`

Within instructions, the register name syntax identifies the input operand and output result data size and type. For more information on data size and type selection for ALU instructions, see "Register File Registers" on page 2-5.

The remainder of this chapter presents descriptions of ALU instructions, options, and results using instruction syntax. For an explanation of the instruction syntax conventions used in ALU and other instructions, see "Instruction Line Syntax and Structure" on page 1-22. For a list of ALU instructions and their syntax, see "ALU Instruction Summary" on page 3-21.

ALU Operations

The ALU performs arithmetic operations on fixed-point and floating-point data and logical operations on fixed-point data. The processor uses compute block registers for the input operands and output result from ALU operations. The compute block register file registers are `XR31` through `XR0` and `YR31` through `YR0`. The ALU has one special-purpose double register—the `PR` register—for parallel results. The processor uses the `PR` register with the different types of `SUM`, `VMAX`, and `VMIN` instructions. For more information on the register files and register naming syntax for selecting data type and width, see "Register File Registers" on page 2-5. The following examples are ALU instructions that demonstrate arithmetic operations.

```
XR2 = R1 + R0 ;;
/* This is a fixed-point add of the 32-bit input operands XR1 and
XR0; the DSP places the result in XR2. */

YLR1:0 = ABS( R3:2 - R5:4 ) ;;
```

```
/* This is a fixed-point subtract of the 64-bit input operand
XR5:4 from XR3:2; the DSP places the absolute value of the result
in XR1:0; the "L" in the result register name directs the DSP to
treat the input and output as 64-bit long data. */

XYFR2 = ( R1 + R0 ) / 2 ;;
/* This is a floating-point add and divide by 2 of the 32-bit
input operands XR1+XR0 and YR1+YR0; the DSP places the results in
XR2 and YR2; this is a Single-Instruction, Multiple-Data (SIMD)
operation, executing in both compute blocks simultaneously. */
```

When multiple input operands are held in a single register, the DSP processes the data in parallel. For example, assume that the `YR0` register contains `0x00050003` and the `YR1` register contains `0x00040008` (as shown in Figure 3-2.

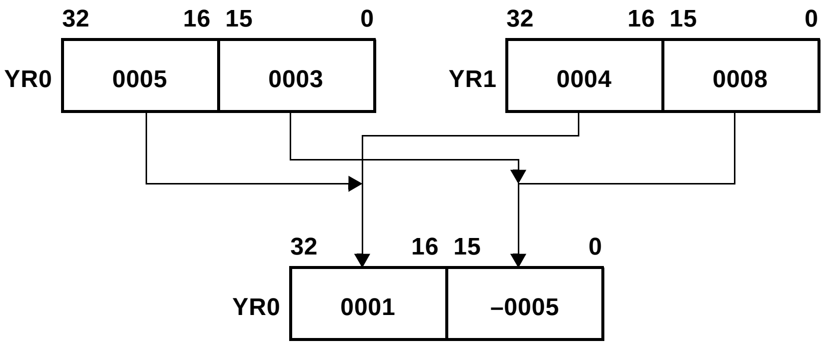


Figure 3-2. Input Operands for Parallel Subtract

After executing the instruction `YSR2 = R0 - R1;;`, the `YR2` register contains `0x0001FFFB` (`0x1` in upper half and `-0x5` in lower half).

All ALU instructions generate status flags to indicate the status of the result. Because multiple operations are occurring in a multiple operands of one instruction, the value of the flag is an ORing of the results of all of the operations. The instruction demonstrated in Figure 3-2 sets the `YAN` flag (Y compute block, ALU result negative) because one of the two subtractions resulted in a negative value. For more information on ALU status, see "ALU Execution Status" on page 3-10.

ALU Instruction Options

Most of the ALU instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions and options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at the end of the instruction's slot. For a list indicating which options apply for particular ALU instructions, see "ALU Instruction Summary" on page 3-21. The ALU instruction options include:

- `()` signed operation, no saturation[1], round-to-nearest even[2], fractional mode[3]
- `(S)` signed operation, saturation[1]
- `(U)` unsigned operation, no saturation[1], round-to-nearest even[2]
- `(SU)` unsigned operation, saturation[1]
- `(X)` extend operation for `ABS`
- `(T)` signed operation, truncate[4]

---

[1] Where saturation applies
[2] Where rounding applies
[3] Where applies for floating-point operations
[4] Where truncation applies

- `(TU)` unsigned operation, truncate[4]
- `(Z)` signed result returns zero operation for `MIN/MAX`
- `(UZ)` unsigned result returns zero operation for `MIN/MAX`
- `(I)` signed operation, integer mode[4]
- `(IU)` unsigned operation, integer mode[3]
- `(IS)` signed operation, saturation, integer mode[3]
- `(ISU)` unsigned operation, saturation, integer mode[3]

The following examples are ALU instructions that demonstrate arithmetic operations with options applied.

```
XR2 = R1 + R0 (S);;
/* This is a fixed-point add of the 32-bit input operands with
saturation. */

YLR1:0 = ABS( R3:2 - R5:4 ) (T) ::
/* This is a fixed-point subtract of the 64-bit input operands
with truncation. */

XYFR2 = ( R1 + R0 ) / 2 () ;;
/* This is a floating-point add and divide by 2 of the 32-bit
input operands without truncation; this is the same as omitting
the parenthesis. */
```

Signed/Unsigned Option

The processor always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. Fixed- and floating-point data in the ALU may be unsigned or two's-complement. For information on the supported numeric formats, see "Numeric Formats" on page 2-15.

Saturation Option

There are two types of saturation arithmetic that may be enabled for an instruction — signed or unsigned. For signed saturation, whenever overflow occurs (AV flag is set), the maximum positive value or the minimum negative value is replaced as the output of the operation. For unsigned saturation, overflow causes the maximum value or zero to be replaced as the output of the operation. Maximum and minimum values refer to the maximum and minimum values representable in the output format. For example, the maximum positive, minimum negative, and maximum unsigned values in 16-bit short word arithmetic are 0x7fff, 0x8000, and 0xffff respectively.

Under saturation arithmetic, the flags AV and AC reflect the state of the ALU operation *prior* to saturation. For example, with signed saturation when an operation overflows, the maximum or minimum value is returned and AV remains set. On the other hand, the flags AN and AZ are set according to the final saturated result, therefore they correctly reflect the sign and any equivalence to zero of the final result. This allows the correct evaluation of the conditions AEQ, ALT, and ALE even during overflow, when using saturation arithmetic.

Extension (ABS) Option

For the ABS instruction, the X option provides an extended output range. Without the X, the output range is 0 to the maximum positive signed value (0x0 through 0x7F…F). When ABS with the X option is used, the output range is extended from 0x0 to 0xFF…FF. The output numbers are unsigned in the extended range.

Truncation Option

For ALU instructions that support truncation as the T option, this option permits selection of the results rounding mode. The processor supports two modes of rounding — round-toward-zero and round-toward-nearest.

The rounding modes comply with the IEEE 754 standard and have these definitions:

- Round-Toward-Nearest (not using T option). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.
- Round-Toward-Zero (using T option). If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

Though these rounding modes comply with standards set for floating-point data, they also apply for fixed-point multiplier operations on fractional data. The same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Using its local result register for fixed-point operations, the multiplier rounds-to-zero by reading only the upper bits of the result and discarding the lower bits.

Return Zero (MAX/MIN) Option

For the MAX/MIN instructions, the Z option changes the operation, returning zero if the second input register contains the maximum (for MAX) or minimum (for MIN) value. For example, *without the Z option*, the pseudo code for the MAX instruction is:

```
Rsd = MAX (Rmd, Rnd) ;;
```

The ALU determines whether *Rmd* or *Rnd* contains the maximum and places the maximum in *Rsd*.

For example, *with the Z option*, the pseudo code for the MAX instruction is:

```
Rsd = MAX (Rmd, Rnd) (Z) ;;
```

The ALU determines whether *Rmd* or *Rnd* contains the maximum. If *Rmd* contains the maximum, the ALU places the maximum in *Rsd*. If *Rnd* contains the maximum, the ALU places zero in *Rsd*.

Fractional/Integer Option

The processor always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. In the ALU, fractional or integer format is available for the EXPAND and COMPACT instructions. The default is fractional format. Use the I option for integer format. For information on the supported numeric formats, see "Numeric Formats" on page 2-15.

ALU Execution Status

ALU operations update status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register (see Figure 2-2 on page 2-4 and Figure 2-3 on page 2-5). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see "ALU Execution Conditions" on page 3-13.

Table 3-1 shows the flags in XSTAT or YSTAT that indicate ALU status (a 1 indicates the condition) for the most recent ALU operation.

Table 3-1. ALU Status Flags

Flag	Definition	Updated By…
AZ	ALU fixed-point zero and floating-point underflow	All ALU ops
AN	ALU negative	All ALU ops
AV	ALU overflow	All arithmetic ops
AC	ALU carry	All fixed-point ops; cleared by floating-point ops
AI	ALU floating-point invalid operation	All floating-point ops; cleared by fixed-point ops

ALU operations also update sticky status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register. Table 3-2 shows the flags in XSTAT or YSTAT that indicate ALU sticky status (a 1 indicates the condition) for the most recent ALU operation. Once set, a sticky flag remains high until explicitly cleared.

Table 3-2. ALU Status Sticky Flags

Flag	Definition	Updated By…
AUS	ALU floating-point underflow, sticky	All floating-point ops
AVS	ALU floating-point overflow, sticky	All floating-point ops
AOS	ALU fixed-point overflow, sticky	All fixed-point ops
AIS	ALU floating-point invalid operation, sticky	All floating-point ops

Flag update occurs at the end of each operation and is available on the next instruction slot. A program cannot write the Arithmetic Status register explicitly in the same cycle that the multiplier is performing an operation.

Multi-operand instructions (for example, `B`*`Rs = Rm + Rn`*) produce multiple sets of results. In this case, the processor determines a flag by ORing the result flag values from individual results.

AN — ALU Negative

The `AN` flag is set whenever the result of an ALU operation is negative. The `AN` flag is set to the most significant bit of the result. An exception is the instructions below, in which the `AN` flag is set differently:

- `Rs = ABS Rm; AN` is `Rm` (input data) sign
- `FRs = ABS Rm; AN` is `Rm` (input data) sign
- `Rs = ABS (Rm {+|-} Rn); AN` is set to be the sign of the result prior to `ABS` operation
- `Rs = MANT FRm; AN` is `Rm` (input data) sign
- `FRs = ABS (Rm {+|-} Rn); AN` is set to be the sign of the result prior to `ABS` operation

The result sign of the above instructions is not indicated as it is always positive.

AV — ALU Overflow

The `AV` flag is an overflow indication. In all ALU operations, this bit is set when the correct result of the operation is too large to be represented by the result format. The overflow check is done always as signed operands, unless the instruction defines otherwise.

If in the following example `R5` and `R6` are `0x70…0` (large positive numbers), the result of the `Add` instruction (above) will produce a result that is larger than the maximum at the given format.

```
R10 = R5 + R6;;
```

As shown in the following example, an instruction can be composed of more than one operation.

```
R11:10 = expand (Rm + Rn)(I);
```

If `Rm` and `Rn` are `0x70…0`, the overflow is defined by the final result and is not defined by intermediate results. In the case above, there is no overflow.

AI — ALU Invalid

The `AI` flag indicates an invalid floating-point operation as defined by IEEE floating-point standard.

AC — ALU Carry

The `AC` flag is used as carry out of add or subtract instructions that can be chained. It can also be used as an indication for unsigned overflow in these operations. `AV` is set when there is signed overflow.

ALU Execution Conditions

In a conditional ALU instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional ALU instructions take the form:

```
IF cond; DO, instr.; DO, instr.; DO, instruct. ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `DO` before the instruction makes the instruction unconditional.

Table 3-3 lists the ALU conditions. For more information on conditional instructions, see "Conditional Execution" on page 8-12.

Table 3-3. ALU Conditions

Condition	Description	Flags Set
AEQ	ALU equal to zero	AZ = 1
ALT	ALU less than zero	AN and AZ = 1
ALE	ALU less than or equal to zero	AN or AZ = 1
NAEQ	NOT (ALU equal to zero)	AZ = 0
NALT	NOT (ALU less than zero)	AN and AZ = 0
NALE	NOT (ALU less than or equal to zero)	AN or AZ = 0

ALU Static Flags

In the program sequencer, the static flag (`SFREG`) can store status flag values for later usage in conditional instructions. With `SFREG`, each compute block has two dedicated static flags `X/YSCF0` (condition is `SF0`) and `X/YSCF1` (condition is `SF1`). The following example shows how to load a compute block condition value into a static flag register.

```
XSCF0 = XAEQ ;; /* Load X-compute block SEQ flag into XSCF0 bit
in static flags (SFREG) register */
IF SF0, XR5 = R4 + R3 ;; /* the SF0 condition tests the XSCF0
static flag */
```

For more information on static flags, see "Conditional Execution" on page 8-12.

ALU Examples

Listing 3-1 provides a number of example ALU arithmetic instructions. The comments with the instructions identify the key features of the instruction, such as fixed- or floating-point format, input operand size, and register usage.

Listing 3-1. ALU Instruction Examples

```
LR5:4 = R11:10 + R1:0 ;;
/* This is a fixed-point add of the 64-bit input operands
XR11:10 + XR1:0 and YR11:10 + YR1:0; the DSP places the result in
XR5:4 and YR5:4. */

YSR1:0 = R31:30 + R25:24 ;;
/* This is a fixed-point add of the four 16-bit input operands
YR31:30 and the four operands in YR25:24; the DSP places the four
results in YR1:0. */

XR3 = R5 AND R7 ;;
/* This is a logical AND of the 32-bit input operands XR5 and
XR7; the DSP places the result in XR3. */

YR4 = SUM SR3:2 ;;
/* This is a signed sideways sum of the four 16-bit input oper-
ands in YR3:2; the DSP places the result in YR4. */

R9 = R4 + R8, R2 = R4 - R8 ;;
/* This is a dual instruction (two instructions in one instruc-
tion slot); the first instruction is a fixed-point add of the
32-bit input operands XR4 + XR8 and YR4 + YR8; the DSP places the
results in XR9 and YR9; the second instruction is a fixed-point
subtract of the 32-bit input operands XR4 - XR8 and YR4 - YR8;
the DSP places the results in XR2 and YR2. */
```

```
FR9 = R4 + R8 ;;
/* This is a floating-point add of the 32-bit input operands in
XR4 +XR8 and YR4 + YR8; the DSP places the results in XR9 and
YR9. */

XFR9:8 = R3:2 + R5:4 ;;
/* This is a floating-point add of the 40-bit (Extended Word)
input operands in XR3:2 and XR5:4; the DSP places the result in
XR9:8. */
```

Example Parallel Addition of Byte Data

Figure 3-3 shows an ALU add using Byte input operands and dual registers. The syntax for the instruction is:

```
XBR11:10 =  R9:8 + R7:6 ;;
```

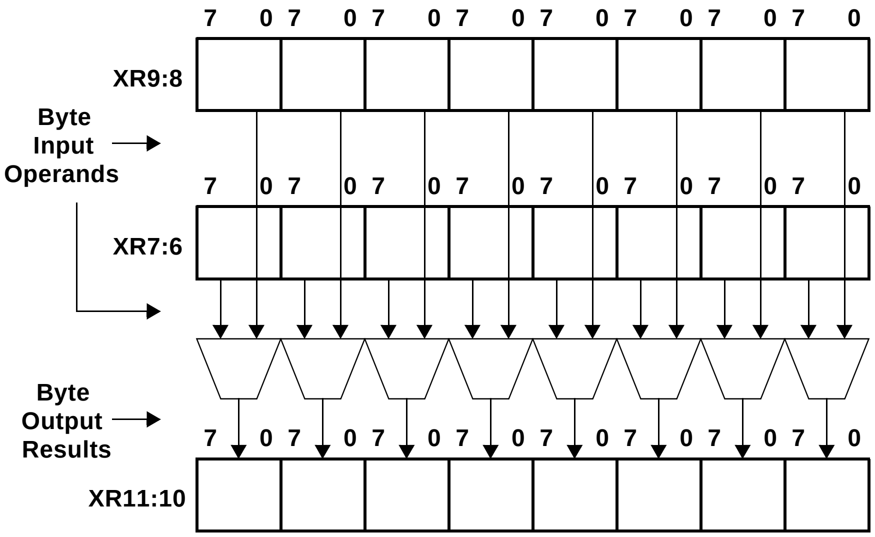


Figure 3-3. Input Operands for Parallel Add

It is important to note that the ALU processes the eight add operations independent of each other, but updates the arithmetic status based on an ORing of the status of all eight operations.

Example Sideways Addition of Byte Data

Figure 3-4 shows an ALU sideways sum using Byte input operands and a dual register. The syntax for the instruction is:

```
XR11 = SUM BR9:8 (U) ;; /* unsigned sideways sum */
```

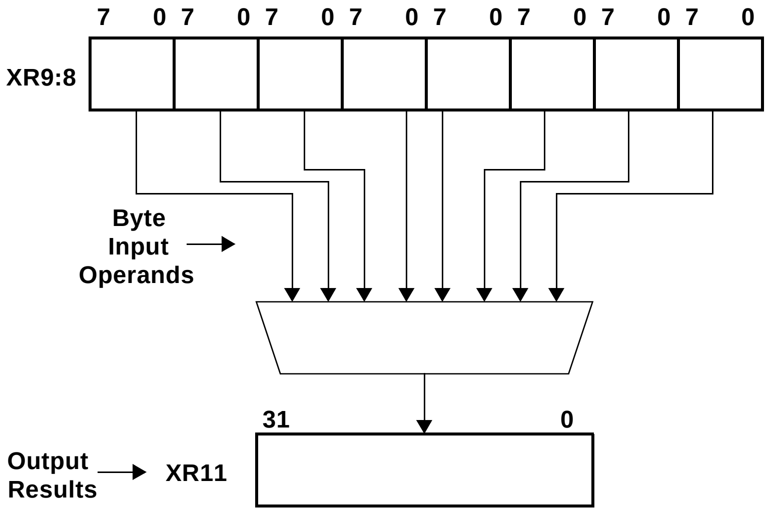


Figure 3-4. Input Operands for Sideways Add

Example Parallel Result (PR) Register Usage

The ALU supports a set of special instructions such as `SUM`, `VMAX`, `VMIN`, and `ABS` that can use the `PR1:0` register. The `PR1:0` register is an ALU register that is not memory mapped the way that data register file registers are mapped. To load or store, programs must load `PR1:0` from data registers, or store `PR1:0` to data registers—there is no memory load or store for the `PR1:0` register. To access the `PR1:0` registers, the application must use instructions with the pseudo code:

```
PR1:0 = Rmd ;;
Rsd = PR1:0 ;;
```

The instruction must operate on double registers even if only `PR0` or `PR1` is required.

The `SUM` instruction is one of the instructions that can use the `PR1:0` register to hold parallel results. When using the `PR1:0` register, the `SUM` instruction performs a short or byte wise parallel add of the input operands, adds this quantity to the contents of one of the `PR` registers, then stores the parallel result to the `PR` register.

This instruction performs four 16-bit additions and adds the result to the current contents of the `PR0` register.

```
PR0 += SUM SR5:4;;
```

Figure 3-5 shows a register use to get pParallel/Sideways Add.

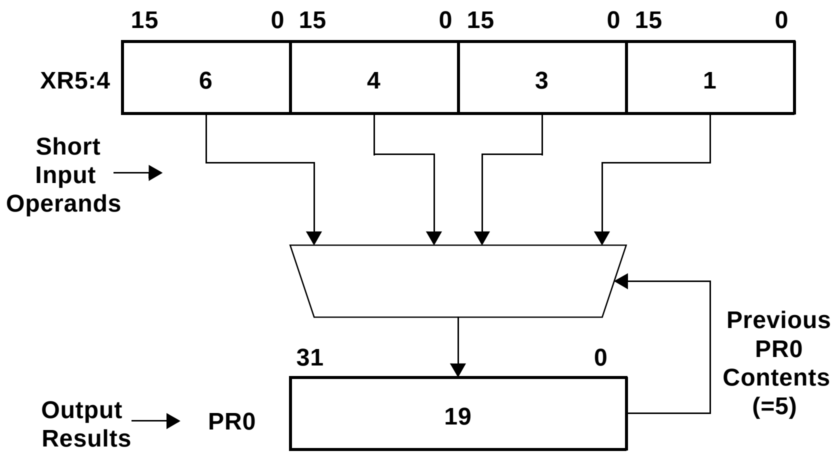


Figure 3-5. Input Operands for Parallel/Sideways Add

ALU Instruction Summary

The following listings show the ALU instructions' syntax:

- Listing 3-2 "ALU Fixed-Point Instructions"
- Listing 3-3 "ALU Logical Operation Instructions"
- Listing 3-4 "ALU Fixed-Point Miscellaneous"
- Listing 3-5 "Floating-Point ALU Instructions"

The conventions used in these listings for representing register names, optional items, and choices are covered in detail in "Register File Registers" on page 2-5. Briefly, these conventions are:

- `{ }` – the curly braces enclose options; these braces are not part of the instruction syntax.
- `|` – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *`Rmd`* – the register names in italic represent user-selectable single (*`Rs`*, *`Rm`*, or *`Rn`*), double (*`Rsd`*, *`Rmd`*, or *`Rnd`*) or quad (*`Rsq`*, *`Rmq`*, or *`Rnq`*) register names.

Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see "Instruction Line Syntax and Structure" on page 1-22 and "Instruction Parallelism Rules" on page 1-26.

Listing 3-2. ALU Fixed-Point Instructions

```
{X|Y|XY}{S|B}Rs = Rm +|- Rn {({S|SU})} ;[1]
{X|Y|XY}{L|S|B}Rsd = Rmd +|- Rnd {({S|SU})} ;[1]
{X|Y|XY}Rs = Rm + CI {-1} ;
{X|Y|XY}LRsd = Rmd + CI {-1} ;
{X|Y|XY}{S|B}Rs = Rm +|- Rn + CI {-1} {({S|SU})} ;[1]
{X|Y|XY}{L|S|B}Rsd = Rmd +|- Rnd + CI {-1} {({S|SU})} ;[1]
{X|Y|XY}{S|B}Rs = (Rm +|- Rn)/2 {({T}{U})} ;[2]
{X|Y|XY}{L|S|B}Rsd = (Rmd +|- Rnd)/2 {({T}{U})} ;[2]
{X|Y|XY}{S|B}Rs = ABS Rm ;
{X|Y|XY}{L|S|B}Rsd = ABS Rmd ;
{X|Y|XY}{S|B}Rs = ABS (Rm + Rn) {(X)} ;[3]
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd + Rnd) {(X)} ;[3]
{X|Y|XY}{S|B}Rs = ABS (Rm - Rn) {({X}{U})} ;[4]
{X|Y|XY}{L|S|B}Rsd = ABS (Rmd - Rnd) {({X}{U})} ;[4]
{X|Y|XY}{S|B}Rs = - Rm ;
{X|Y|XY}{L|S|B}Rsd = - Rmd ;
{X|Y|XY}{S|B}Rs = MAX|MIN (Rm, Rn) {({U}{Z})} ;[5]
{X|Y|XY}{L|S|B}Rsd = MAX|MIN (Rmd, Rnd) {({U}{Z})} ;[5]
{X|Y|XY}S|BRsd = VMAX|VMIN (Rmd, Rnd) ;
{X|Y|XY}{S|B}Rs = INC|DEC Rm {({S|SU})} ;[1]
{X|Y|XY}{L|S|B}Rsd = INC|DEC Rmd {({S|SU})} ;[1]
{X|Y|XY}{S|B}COMP(Rm, Rn) {(U)} ;[5]
{X|Y|XY}{L|S|B}COMP(Rnd,Rnd) {(U)} ;[5]
```

[1] Options include: ( ): no saturation, (S): saturation, signed, (SU): saturation, unsigned

[2] Options include: ( ): signed, round-to-nearest even, (T): signed, truncate, (U): unsigned, round-to-nearest even, (TU): unsigned, truncate

[3] Options include: (X): extend for ABS

[4] Options include: (X): extend for ABS, (U): unsigned, round-to-nearest even, (XU): unsigned, extend

[5] Options include: ( ): regular signed comparison, (U): comparison between unsigned numbers, (Z): returned result is zero if Rn is selected by MIN/MAX operation; otherwise returned result is Rm, (UZ): unsigned comparison with option (Z) as described above

```
{X|Y|XY}{S|B}Rs = CLIP Rm BY Rn ;
{X|Y|XY}{L|S|B}Rsd = CLIP Rmd BY Rnd ;
{X|Y|XY}Rs = SUM S|B Rm {(U)} ;[1]
{X|Y|XY}Rs = SUM S|B Rmd {(U)} ;[1]
{X|Y|XY}Rs = ONES Rm|Rmd ;
{X|Y|XY}PR1:0 = Rmd ;
{X|Y|XY}Rsd = PR1:0 ;
{X|Y|XY}Rs = BFOINC Rmd ;
{X|Y|XY}PR0|PR1 += ABS (SRmd - SRnd){(U)} ;[1]
{X|Y|XY}PR0|PR1 += ABS (BRmd - BRnd){(U)} ;[1]
{X|Y|XY}PR0|PR1 += SUM SRm {(U)} ;[1]
{X|Y|XY}PR0|PR1 += SUM SRmd {(U)} ;[1]
{X|Y|XY}PR0|PR1 += SUM BRm {(U)} ;[1]
{X|Y|XY}PR0|PR1 += SUM BRmd {(U)} ;[1]
{X|Y|XY}{S|B}Rs = Rm + Rn, Ra = Rm - Rn ; (dual operation)
{X|Y|XY}{L|S|B}Rsd = Rmd + Rnd, Rad = Rmd - Rnd ; (dual operation)
```

Listing 3-3. ALU Logical Operation Instructions

```
{X|Y|XY}Rs = PASS Rm ;
{X|Y|XY}LRsd = PASS Rmd ;
{X|Y|XY}Rs = Rm AND|AND NOT|OR|XOR Rn ;
{X|Y|XY}LRsd = Rmd AND|AND NOT|OR|XOR Rnd ;
{X|Y|XY}Rs = NOT Rm ;
{X|Y|XY}LRsd = NOT Rmd ;
```

Listing 3-4. ALU Fixed-Point Miscellaneous

```
{X|Y|XY}Rsd = EXPAND SRm {+|- SRn} {({I|IU})} ;[2]
{X|Y|XY}Rsq = EXPAND SRmd {+|- SRnd} {({I|IU})} ;[2]
{X|Y|XY}Rsd = EXPAND BRm {+|- BRn} {({I|IU})} ;[2]
```

[1] Options include: `( )`: signed, `(U)`: unsigned

[2] Options include: `( )`: fractional, `(I)`: integer signed, `(IU)`: integer unsigned

```
{X|Y|XY}Rsq = EXPAND BRmd {+|- BRnd} {({I|IU})} ;[2]
{X|Y|XY}SRs = COMPACT Rmd {+|- Rnd} {({T|I|IS|ISU})} ;[1]
{X|Y|XY}BRs = COMPACT SRmd {+|- SRnd} {({T|I|IS|ISU})} ;[1]
{X|Y|XY}Rs  = COMPACT LRmd {({U|IS|ISU})} ;
{X|Y|XY}Rsd = COMPACT QRmq {({U|IS|ISU})} ;
{X|Y|XY}BRsd = MERGE Rm, Rn ;
{X|Y|XY}BRsq = MERGE Rmd, Rnd ;
{X|Y|XY}SRsd = MERGE Rm, Rn ;
{X|Y|XY}SRsq = MERGE Rmd, Rnd ;
```

Listing 3-5. Floating-Point ALU Instructions

```
{X|Y|XY}FRs = Rm +|- Rn {(T)} ;[2]
{X|Y|XY}FRsd = Rmd +|- Rnd {(T)} ;[2]
{X|Y|XY}FRs = (Rm +|- Rn)/2 {(T)} ;[2]
{X|Y|XY}FRsd = (Rmd +|- Rnd)/2 {(T)} ;[2]
{X|Y|XY}FRs = MAX|MIN (Rm +|- Rn) {(T)} ;[3]
{X|Y|XY}FRsd = MAX|MIN (Rmd +|- Rnd) {(T)} ;[3]
{X|Y|XY}FRs = ABS (Rm) ;
{X|Y|XY}FRsd = ABS (Rmd) ;
{X|Y|XY}FRs = ABS (Rm +|- Rn) {(T)} ;[2]
{X|Y|XY}FRsd = ABS (Rmd +|- Rnd) {(T)} ;[2]
{X|Y|XY}FRs = - Rm ;
{X|Y|XY}FRsd = - Rmd ;
{X|Y|XY}FCOMP (Rm, Rn) ;
{X|Y|XY}FCOMP (Rmd, Rnd) ;
{X|Y|XY}Rs = FIX FRm|FRmd {BY Rn} {(T)} ;[2]
{X|Y|XY}FRs|FRsd = FLOAT Rm {BY Rn} {(T)} ;[2]
{X|Y|XY}FRsd = EXTD Rm ;
```

[1] Options include: `()`: fractional round, `(I)`: integer, no saturate, `(T)`: fractional, truncate, `(IS)`: integer, saturate, signed, `(ISU)`: integer, saturate, unsigned

[2] Options include: `()`: round, `(T)`: truncate

[3] Options include: `()`: round, `(T)`: truncate (`MIN` only)

```
{X|Y|XY}FRs = SNGL Rmd ;
{X|Y|XY}FRs = CLIP Rm BY Rn ;
{X|Y|XY}FRsd = CLIP Rmd BY Rnd ;
{X|Y|XY}FRs = Rm COPYSIGN Rn ;
{X|Y|XY}FRsd = Rmd COPYSIGN Rnd ;
{X|Y|XY}FRs = SCALB FRm BY Rn ;
{X|Y|XY}FRsd = SCALB FRmd BY Rn ;
{X|Y|XY}FRs = PASS Rm ;
{X|Y|XY}FRsd = PASS Rmd ;
{X|Y|XY}FRs = RECIPS Rm ;
{X|Y|XY}FRsd = RECIPS Rmd ;
{X|Y|XY}FRs = RSQRTS Rm ;
{X|Y|XY}FRsd = RSQRTS Rmd ;
{X|Y|XY}Rs = MANT FRm|FRmd ;
{X|Y|XY}Rs = LOGB FRm|FRmd {(S)} ;[1]
{X|Y|XY}FRs = Rm + Rn, FRa = Rm - Rn ; (dual instruction)
{X|Y|XY}FRsd = Rmd + Rnd, FRad = Rmd - Rnd ; (dual instruction)
```

[1] Options include: ( ): do not saturate, (S): saturate

4 CLU

The TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The Communications Logic Unit (CLU) is highlighted in Figure 4-1. The CLU takes its inputs from the register file, and returns its outputs to the register file. Most CLU instructions operate on the trellis registers (TR) and trellis history registers (THR).

This unit performs specialized communications functions, primarily to support decoding, CDMA despreading operations, and complex correlation. These functions may also be applied in non-communications algorithms.

This chapter contains:

The inclusion of the CLU instructions simplifies the programming of these algorithms, yet still retains the flexibility of a software approach. In this way, it is easy to tune the algorithm according to a user's specific requirements. Additionally, the instructions can be used for a variety of purposes; for example, the TMAX instruction, included to support the decoding of turbo codes, is also very useful in the decoding of low-density parity-check codes.

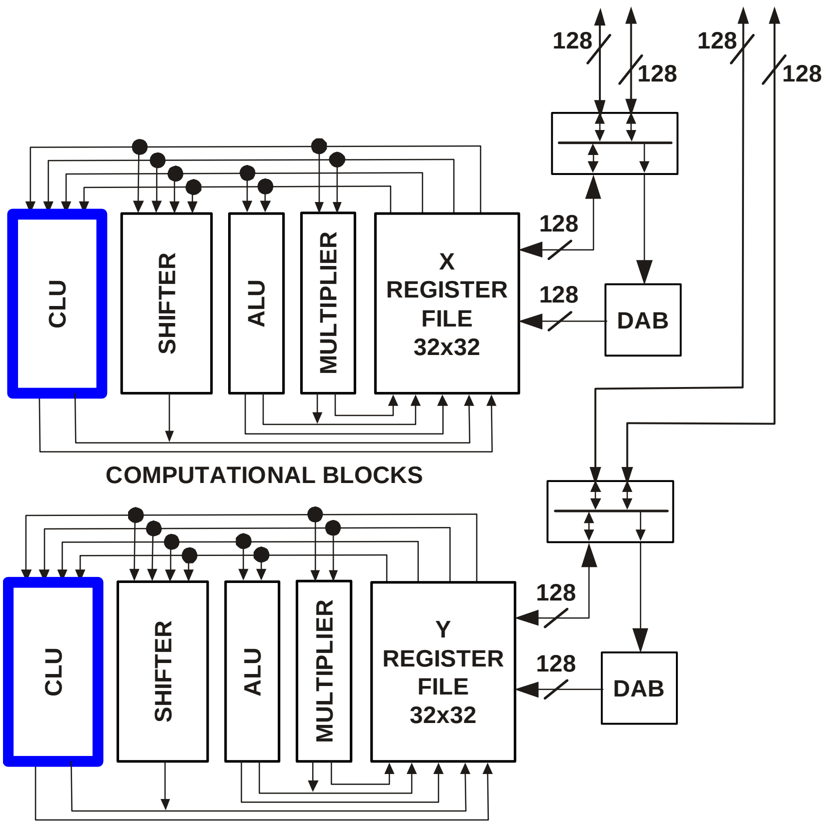


Figure 4-1. CLUs in Compute Block X and Y

The major strength of the TigerSHARC processor is the huge data transfer rate—two 128-bit memory accesses every cycle. For despreading, this enables 16 complex multiply-accumulate operations per cycle of 16-bit complex data (8-bit real, 8-bit imaginary). This enables calculation of a whole 16-bit 64-state trellis calculation every four cycles in both compute blocks together.

CLU operations are applied to fixed-point data. Relating CLU operations and supported real and complex data types shows that the 128-Bit CLU unit within each compute block supports:

- Maximum of values for Viterbi decode (`VMAX`)
- Jacobian logarithm for turbo decode (`TMAX`)
- CDMA despreader (`DESPREAD`)
- CDMA cross correllations (`XCORRS`)
- Polynomial reordering (`PERMUTE`)
- Trellis add/compare/select (`ACS`)

Examining the supported operands for each operation shows that the CLU operations support these data types:

- One or two 32-bit operands
- Two or four 16-bit operands
- Four or eight 8-bit operands
- Output 8-, 16-, or 32-bit results

Within instructions, the register name syntax identifies the input operand and output result data size and type. For more information on data size and type selection for CLU instructions, see "Register File Registers" on page 2-5.

The remainder of this chapter presents descriptions of CLU instructions, options, and results using instruction syntax. For an explanation of the instruction syntax conventions used in CLU and other instructions, see "Instruction Line Syntax and Structure" on page 1-22. For a list of CLU instructions and their syntax, see "CLU Instruction Summary" on page 4-40.

CLU Operations

The CLU performs communications logic operations on fixed-point data. The ADSP-TS201 processor uses compute block registers for the input operands and output result from CLU operations. The CLU instructions refer to three types of registers:

- `Rm,n,s`—register file (data) registers
- `TRm,n,s`—32 (trellis) registers that are dedicated to the CLU instructions
- `THR`—four (trellis history) registers used by the `ACS`, `DESPREAD`, and `XCORRS` instructions for shifted data
- `CMCTL`—communications control registers used by the `XCORRS` instruction as an optional source for `CUT` value

For more information on the register files and register naming syntax for selecting data type and width, see "Register File Registers" on page 2-5.

This section describes:

- "TMAX Function" on page 4-5
- "Trellis Function" on page 4-7
- "Despread Function" on page 4-18
- "Cross Correllations Function" on page 4-26
- "CLU Instruction Options" on page 4-37
- "CLU Execution Status" on page 4-37

For turbo and Viterbi decoding, the communications logic unit (CLU) input data sizes of 8- and 16-bit soft values are supported; output value data sizes of 16 and 32 bits are supported. Care should be taken when choosing the data size to prevent overflow in the calculation.

The DESPREAD function works with 16-bit complex numbers. Each 16-bit complex is composed of the real part (bits 7–0) and the imaginary part (bits 15–8). The result is always one or two complex words, each consisting of two shorts. Bits 15–0 represent the real part, and bits 31–16 represent the imaginary part (as complex numbers in the multiplier).

All CLU instructions generate status flags to indicate the status of the result. Because multiple operations are occurring in a parallel instruction, the value of the flag is an ORing of the results of all of the operations. For more information on CLU status, see "CLU Execution Status" on page 4-37.

TMAX Function

The TMAX function is commonly used in the decoding of turbo codes and other high-performance error-correcting codes. The function is:

$TMAX\,(a, b) = max\,(a, b) + ln\,(1 + e^{-|a-b|})$

The second term is implemented as a table:

$ln\,(1 + e^{-|a-b|})$

(for large $|a - b|$, *ln* of 1 is 0).

The TMAX table input is the result of the subtraction on clock high of the execute1 (EX1) pipe stage. If the result is negative, it is inverted (assuming that the difference between one's complement and two's complement is within the allowed error). The input to Table 4-1 is the seven LSB's of the compare subtract result. If the compare subtract result is larger than seven bits, the output is zero. Note that the implied decimal point is always placed before the five least significant bits.

Table 4-1 shows the TMAX values. The maximum output error is one LSB.

Table 4-1. TMAX Values

Negative Input (Binary)	Positive Input (Binary)	Output (Binary)
11..111.1111X	000.0000X	000.10110
11..111.1110X	000.0001X	000.10101
11..111.1101X	000.0010X	000.10100
11..111.1100X	000.0011X	000.10011
11..111.1011X	000.0100X	000.10010
11..111.1010X	000.0101X	000.10001
11..111.100XX	000.011XX	000.10000
11..111.011XX	000.100XX	000.01110
11..111.010XX	000.101XX	000.01101
11..111.001XX	000.110XX	000.01100
11..111.000XX	000.111XX	000.01011
11..110.111XX	001.000XX	000.01010
11..110.110XX	001.001XX	000.01000
11..110.10XXX	001.01XXX	000.00111
11..110.01XXX	001.10XXX	000.00110
11..110.00XXX	001.11XXX	000.00101
11..101.1XXXX	010.0XXXX	000.00011
11..101.0XXXX	010.1XXXX	000.00010
11..100.XXXXX	011.XXXXX	000.00001
<= 11..10XX.XXXXX	>= 1XX.XXXXX	000.00000

Trellis Function

The trellis diagram (Figure 4-2) is a widely-used tool in communications systems. For example, the Viterbi and turbo decoding algorithms both operate on trellises. The ADSP-TS201 processor provides specialized instructions for trellises with binary transitions and up to eight states. Trellis with larger numbers of states can often be broken up into subtrellises with eight states or fewer and then applied to these instructions.

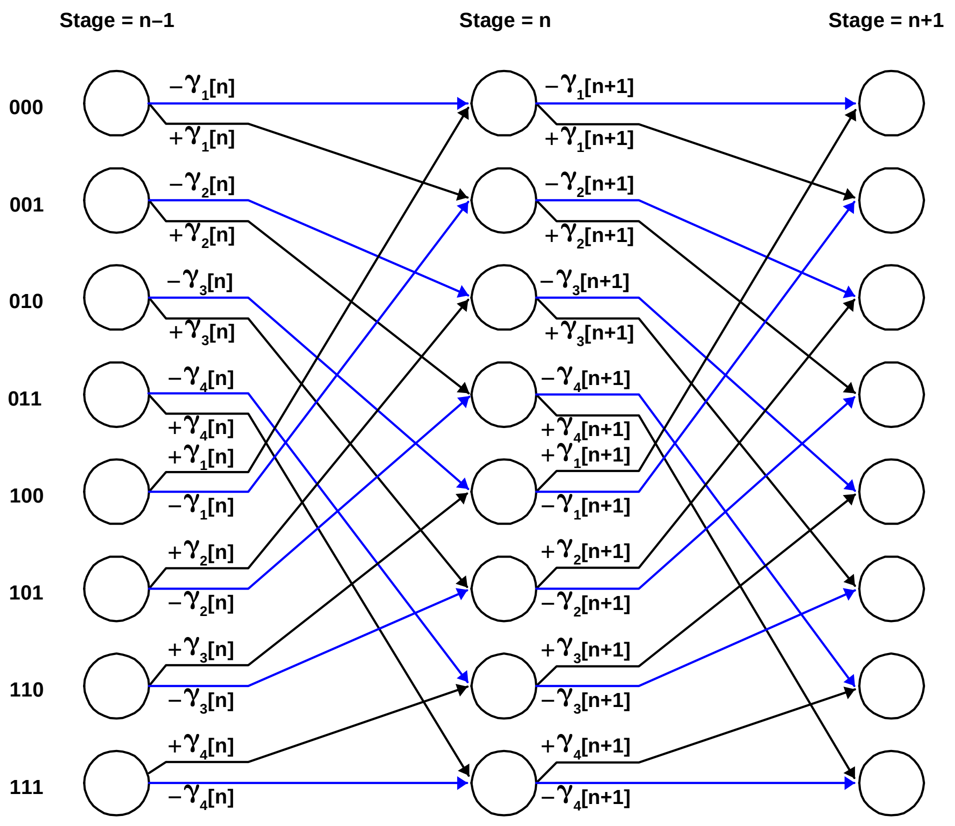


Figure 4-2. Trellis Diagram

A typical trellis diagram (shown in Figure 4-2) represents the set of possible state transitions from one stage to the next.

At each stage n, we need to compute the accumulated metrics for every state. For a particular state, this value will depend on the metrics of each of its two possible previous states (at stage n-1) as well as transition metrics γ[n]. (gamma[n]) corresponding to particular input/output combinations.

The particular symmetry among the transition metrics shown in Figure 4-2 is typical for practical error-correcting codes.

The ACS (Add-Compare-Select) instruction has options for two types of state metric computations. In the Viterbi algorithm, the metrics for each of the two possible previous state are updated, and the one with maximum value is selected. For example, the metric for state "100" (binary for 4) at stage n is computed as:

```
Metrics ("100", n) =
   max ((Metrics("010", n-1) - γ3(n)),
      (Metrics("110", n-1) + γ3(n)))
```

The ACS instruction computes the metrics for four or eight states in parallel, and additionally records information specifying the selected transitions for use in a trace back routine.

A second option, used in turbo decoding, replaces the MAX operation above with the TMAX operation defined in "TMAX Function" on page 4-5.

Trellis Function of the Form STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;

High part of *Rmq* is added to T*Rmd*, low part of *Rmq* is added to T*Rnd*, and TMAX function is executed between both add results, as illustrated in Figure 4-3 and Figure 4-4.

Saturation is supported in this instruction. For more details, see "Saturation Option" on page 3-8.

On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

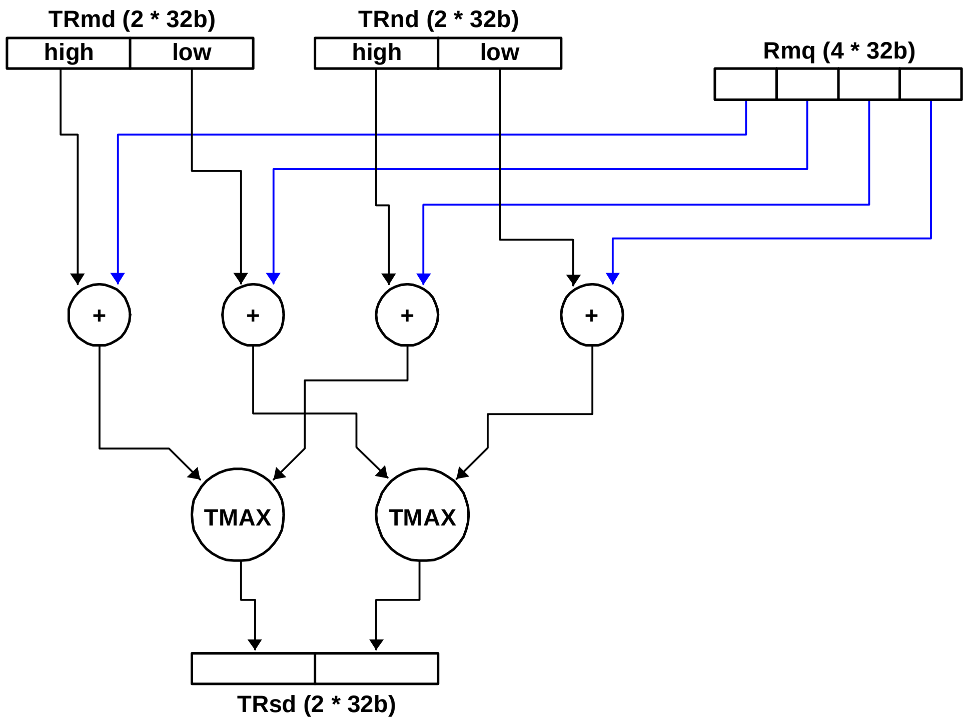


Figure 4-3. TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l)

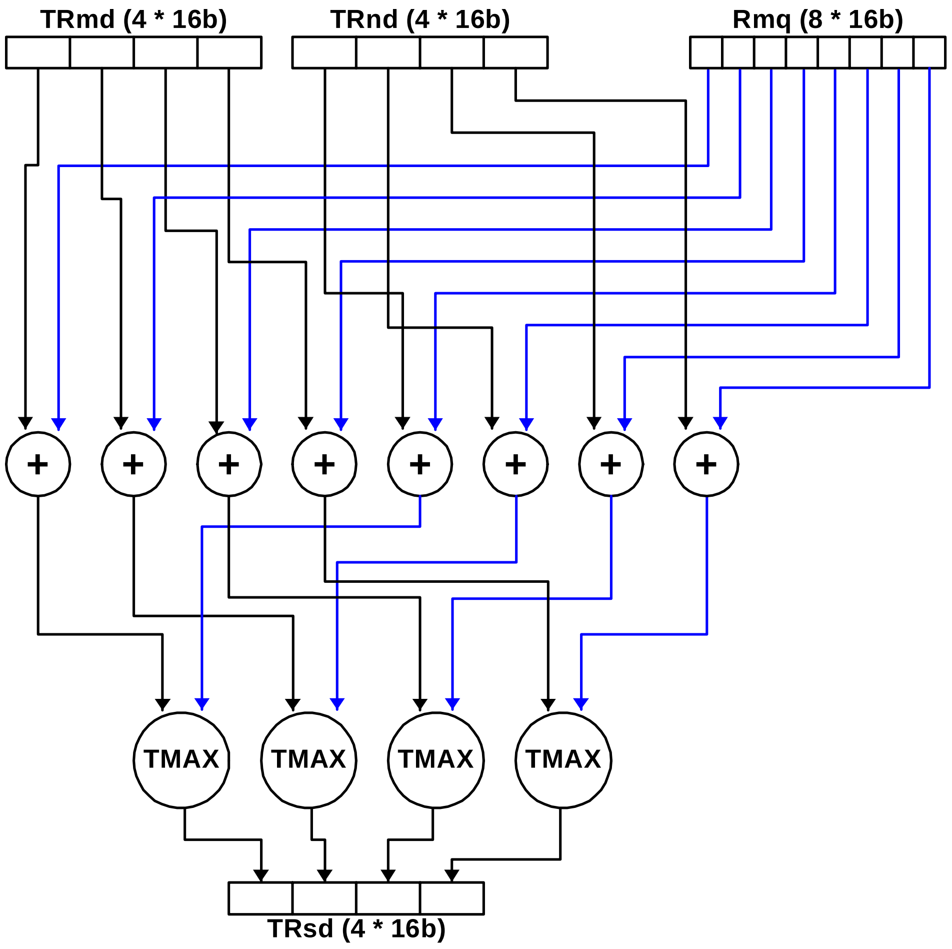


Figure 4-4. STRsd=TMAX(TRmd+Rmq_h, TRnd + RMq_l)

Trellis Function of the Form
`STRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;`

The high part of *Rmq* is subtracted from T*Rmd*, low part of *Rmq* is subtracted from T*Rnd*, and TMAX function is executed between both subtract results, as illustrated in Figure 4-6 and Figure 4-7. For subtraction, the order of operands appears in Figure 4-5.

Saturation is supported in this instruction. For more details, see "Saturation Option" on page 3-8.

On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

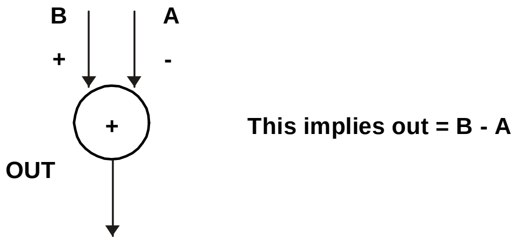


Figure 4-5. Order of Operands for Subtract Diagrams

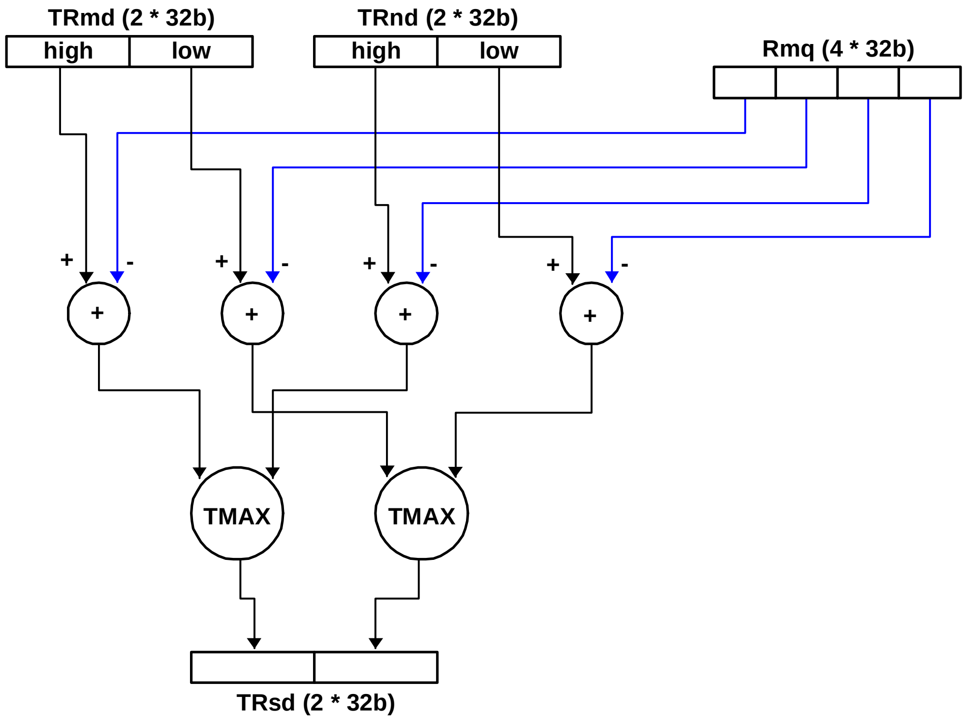


Figure 4-6. TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l)

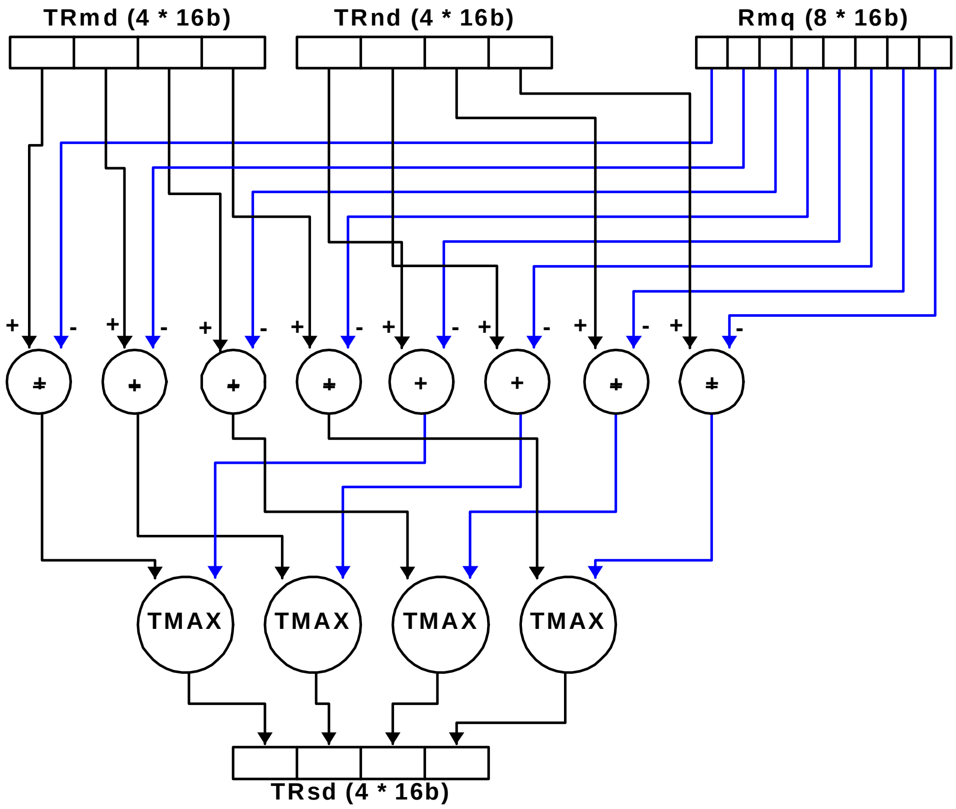


Figure 4-7. STRsd = TMAX(TRmd - Rmq_h, TRnd-Rmq_l)

Trellis Function of the Form `SRs = TMAX(TRm, TRn) ;`

TMAX function is executed between `TRm` and `TRn`, as illustrated in Figure 4-8 and Figure 4-9.

This instruction can be executed in parallel to ALU instructions, shifter instructions, multiplier instructions, and CLU register load. It cannot be executed in parallel to CLU instructions of the same compute block.

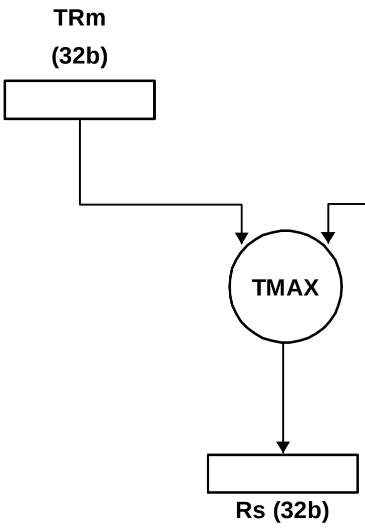


Figure 4-8. Rs = TMAX(TRm, TRn) instruction processing

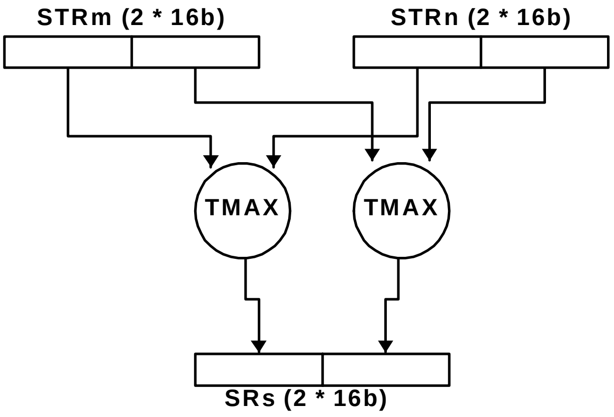


Figure 4-9. SRs = TMAX(TRm, TRn)

Trellis Function of the Form `STRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) ;`

High part of `Rmq` is added to `TRmd`, low part of `Rmq` is added to `TRnd`, and selects the MAX between both add results, as illustrated in Figure 4-10 and Figure 4-11.

Saturation is provided in this instruction. For more details, see "Saturation Option" on page 3-8.

On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

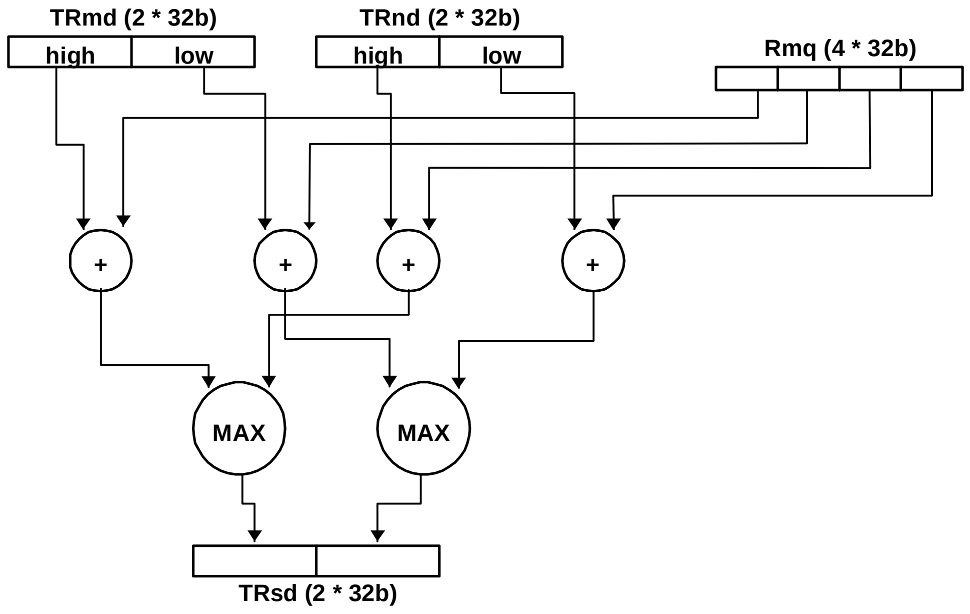


Figure 4-10. TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l)

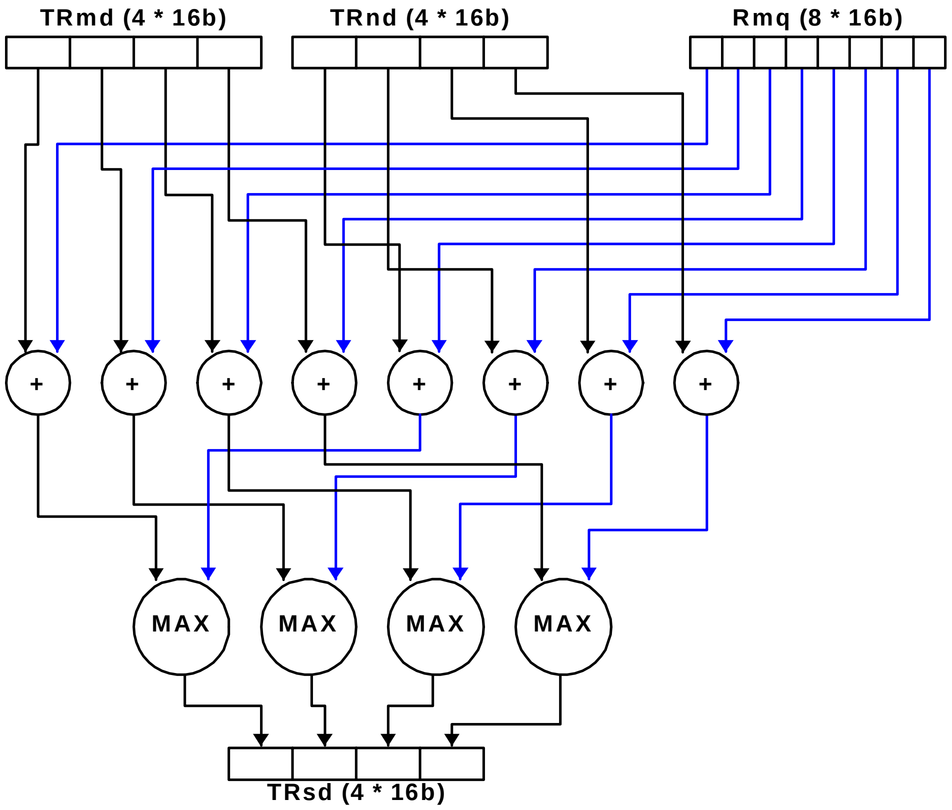


Figure 4-11. STRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l)

Trellis Function of the Form STRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) ;

High part of *Rmq* is subtracted from T*Rmd*, low part of Rmq is subtracted from T*Rnd*, and selects the MAX between both subtract results, as illustrated in Figure 4-12 and Figure 4-13.

Saturation is provided in this instruction. For more details, see "Saturation Option" on page 3-8.

On previous TigerSHARC processors, this instruction could not be executed in parallel to CLU and ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

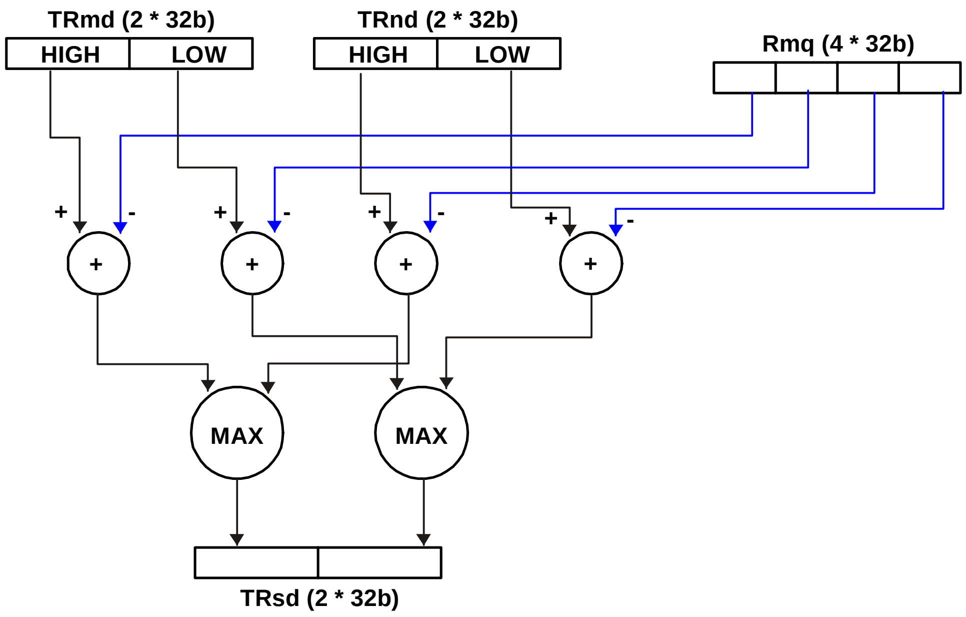


Figure 4-12. TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l)

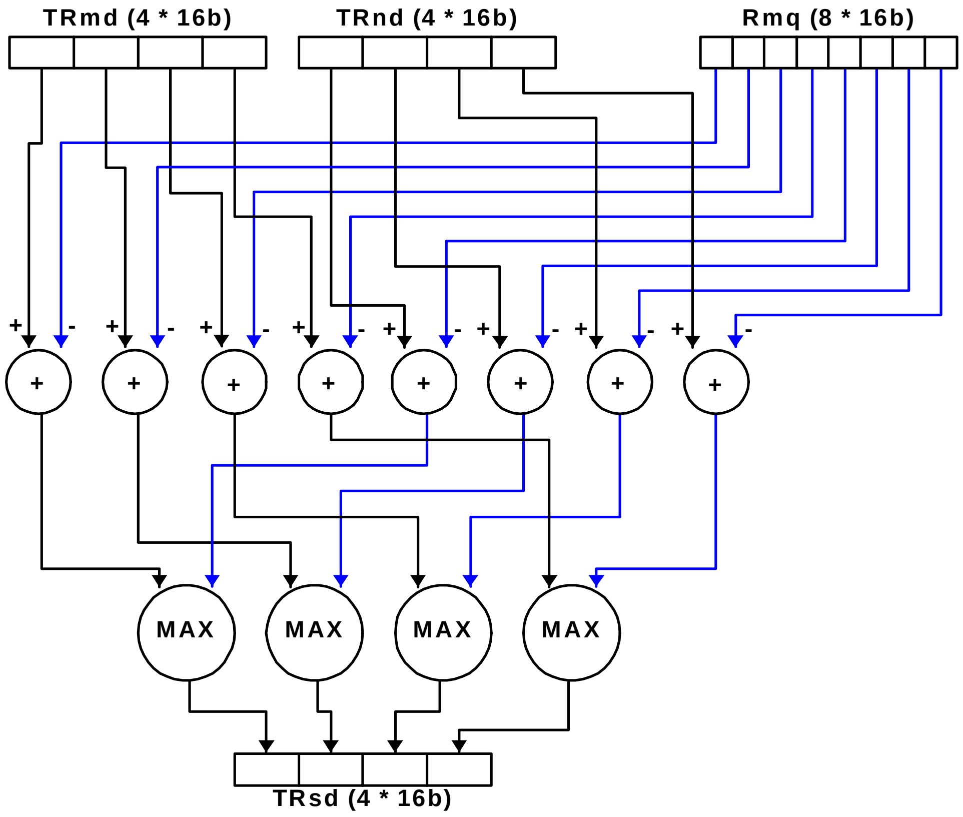


Figure 4-13. STRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l)

Despread Function

The `DESPREAD` instruction implements a highly parallel complex multiply-and-accumulate operation that is optimized for CDMA systems. Despreading involves computing samples of a correlation between complex input data and a precomputed complex spreading/scrambling code sequence.

The input data consists of samples with 8-bit real and imaginary parts. The code sequence samples, on the other hand, are always members of { 1+j, -1+j, -1-j, 1-j }, and are therefore specified by 1-bit real and imaginary parts. The `DESPREAD` instruction takes advantage of this property and is able to compute eight parallel complex multiply-and-accumulates in each block in a single cycle.

The `DESPREAD` instruction supports accumulations over lengths (spread factors) of four, eight, and multiples of eight samples.

The `DESPREAD` input register *Rmq* is composed of 8 complex shorts - D7 to D0, in which each complex number is composed of 2 bytes. The most significant byte is the imaginary, and the least significant is the real. As shown in Figure 4-14, Dn is composed of DnI and DnQ, where I denotes the real part and Q denotes the imaginary part.

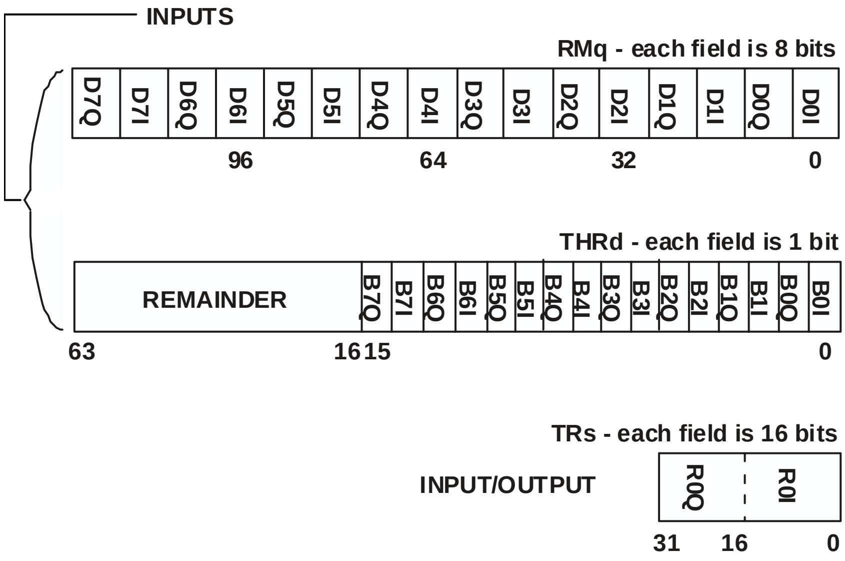


Figure 4-14. Bit field in registers for
TRs += DESPREAD (Rmq, THRd) ;

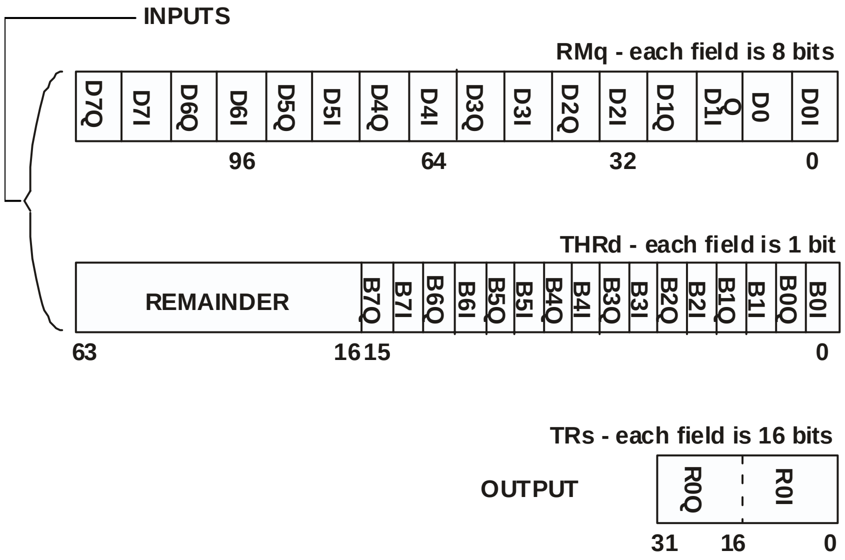


Figure 4-15. Bit field in registers for:
Rs = TRs, TRs = DESPREAD (Rmq, THRd) ;

The `THRd` register is composed of 8 complex numbers `{B7…B0}` and 48 Remainder bits. Each complex number is composed of one real bit (least significant) and one imaginary bit. Each bit represents the value of +1 (when clear) or –1 (when set). `THrd` is post-shifted right by 16 bits so that the lowest 16 bits of the remainder may be used for a despread on the next cycle.

Saturation is provided in this instruction. For more details, see "Saturation Option" on page 3-8.

On previous TigerSHARC processors, this instruction could not be executed in parallel to ALU instructions of the same compute block. This limitation does not apply to the ADSP-TS201 processor.

When you execute this instruction in parallel to `THR` register load the `THR` load instruction takes priority on the `THR` shift in this instruction.

Despread Function of the Form **T*Rs* += DESPREAD (*Rmq*, TH*Rd*) + T*Rs*;**

`TRs` is a complex word, composed of two shorts - real (least significant) and imaginary (most significant).

The function (illustrated in Figure 4-16) is:

```
RsI = (sum (n = 0 to 7) (BnI * DnI - BnQ * DnQ)) + TRsI
RsQ = (sum (n = 0 to 7) (BnI * DnQ + BnQ * DnI)) + TRsQ
```

The multiplication is by integer (±1), the result alignment is to least significant, and the sum is sign extended.

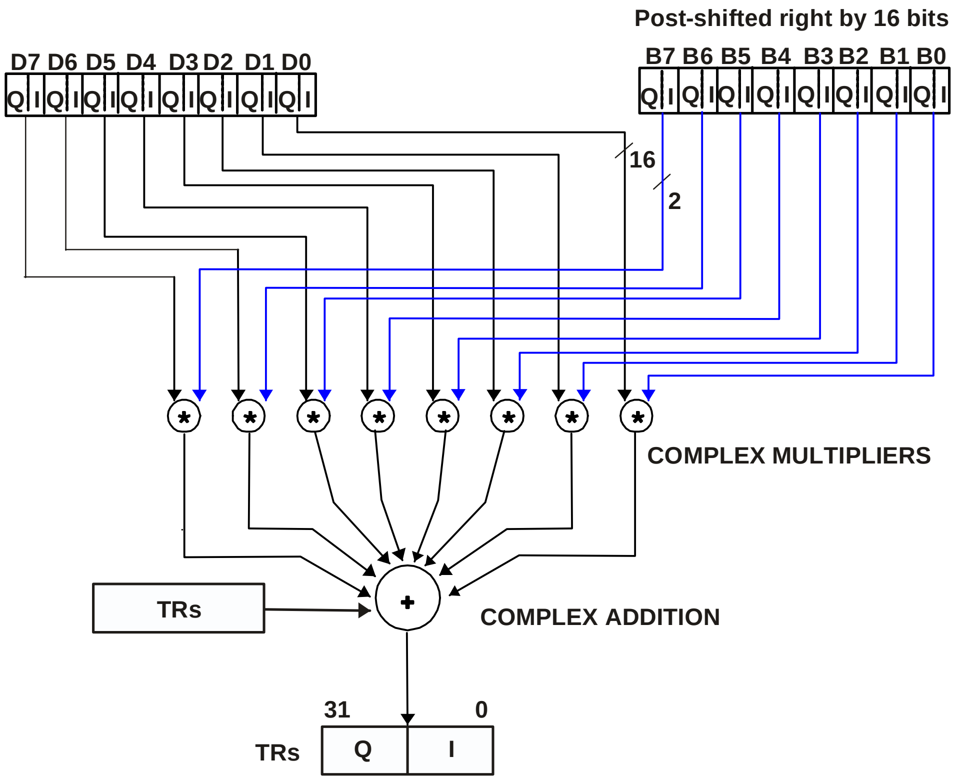


Figure 4-16. TRs += DESPREAD (Rmq, THRd) ;

Despread Function of the Form *Rs* = T*Rs*, T*Rs* = DESPREAD (*Rmq*, TH*Rd*) ;

T*Rs* is a complex word, composed of two shorts - real (least significant) and imaginary (most significant).

The function (illustrated in Figure 4-17) is:

```
RsI = (sum (n = 0 to 7) (BnI * DnI - BnQ * DnQ))
RsQ = (sum (n = 0 to 7) (BnI * DnQ + BnQ * DnI))
```

The value of `TRs` before the operation is stored in `Rs`.

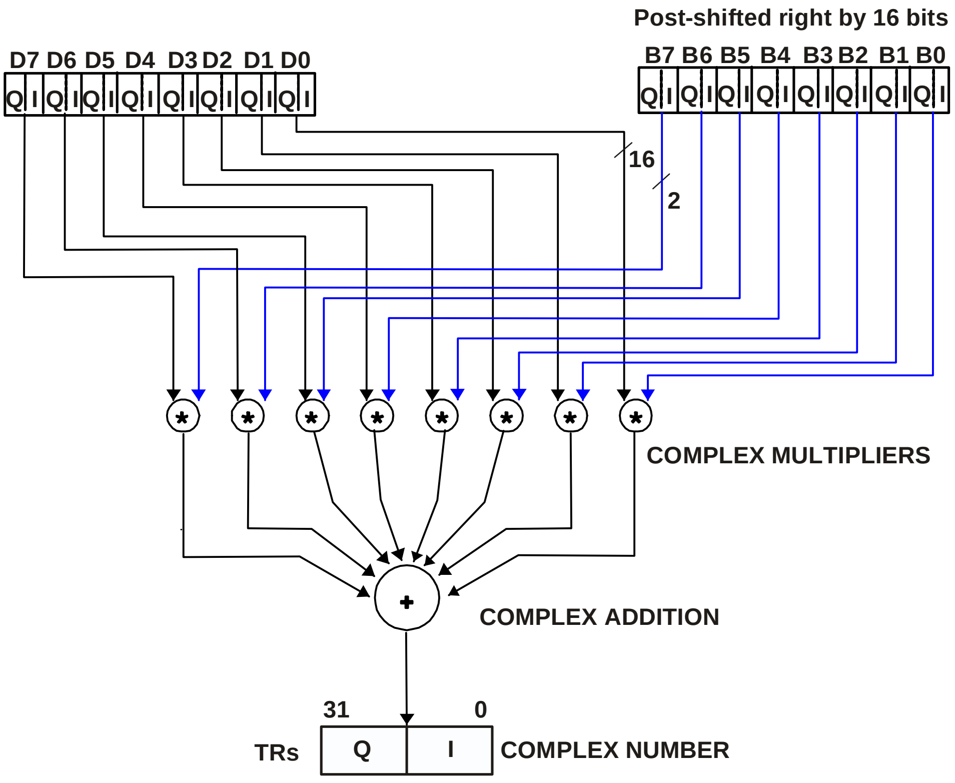


Figure 4-17. Rs = TRs, TRs = DESPREAD (Rmq, THRd) ;

Despread Function of the Form *Rsd* = T*Rsd*, T*Rsd* = DESPREAD (*Rmq*, TH*Rd*) ;

The function (illustrated in Figure 4-18 and Figure 4-19) is:

```
R0I = (sum (n = 0 to 3) (BnI * DnI - BnQ * DnQ))
R0Q = (sum (n = 0 to 3) (BnI * DnQ + BnQ * DnI))
R1I = (sum (n = 4 to 7) (BnI * DnI - BnQ * DnQ))
R1Q = (sum (n = 4 to 7) (BnI * DnQ + BnQ * DnI))
```

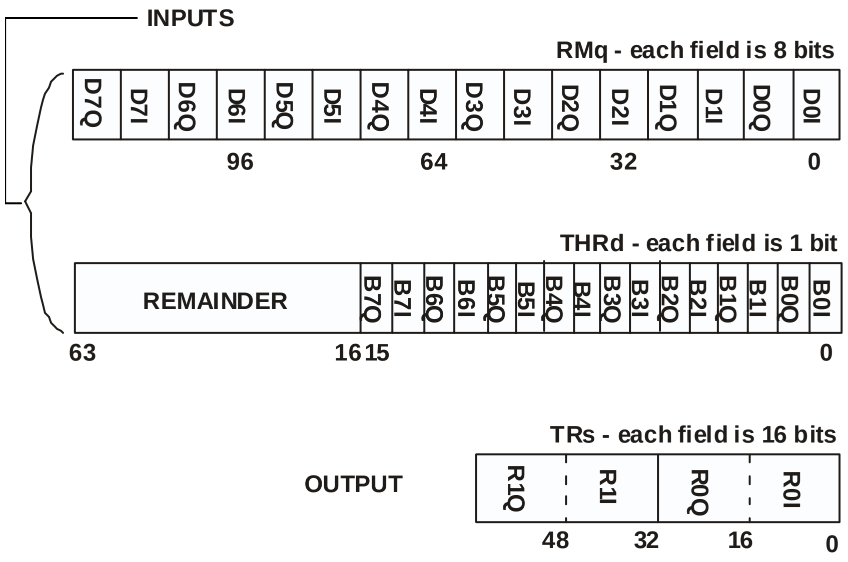


Figure 4-18. Bit fields in registers for:
Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ;

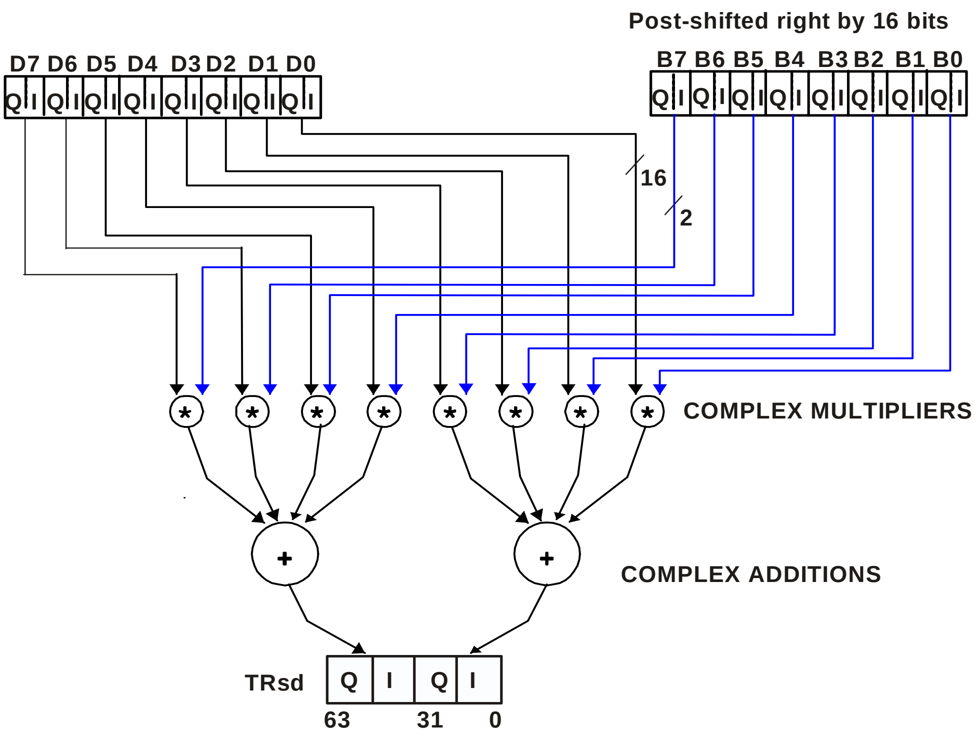


Figure 4-19. Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ;

Cross Correllations Function

The `XCORRS` instruction correlates long input sequence (such as 2048 complex input numbers of an 8 bit pilot) with a known reference sequence for multiple delays. It is convenient to view the `XCORRS` instruction as a single-cycle execution of 16 parallel `DESPREAD` instructions. Some important features of the `XCORRS` instruction include:

- Clear `(CLR)` option, providing a mechanism to clear the trellis registers before beginning a new cross correllation
- Cut inputs `(CUT)` option, providing a mechanism to discard some input data numbers (for initial and final cycles; lower and upper triangles)
- Extended-precision `(EXT)` option, supporting 16-bit input and 32-bit accumulation (instead of the default 8-bit input and 16-bit accumulation
- Outputs correlation strength for each delay
- Memory usage; execution considers memory bandwidth and access to multiple blocks

The `XCORRS` equations for every part X or Y performs the operations shown in Figure 4-20 where (depending on the `CUT` value) the inputs are as shown in Figure 4-21. The `CUT` function values are as shown in Figure 4-22.

```
for (k = 0; k ≤ 15; k++)
{
$TR[k] = TR[k] + \sum_{m=0}^{7} D[m]C[m-k+15]f_{CUT}[m-k+15];$
}
THR3:0 = THR3:0 >> 16;
```

Figure 4-20. XCORRS Equation

$$f_{CUT}[m-k+15] = +1 \text{ or } 0$$

$C[m-k+15] = \pm 1 \pm i \text{ (2 bits)}$

$D[m] = (D_{real}[m] + iD_{imaginary}[m])(16 \text{ bits})$

Figure 4-21. XCORRS Equation Inputs

$$f_{CUT}[w] = \begin{Bmatrix} 1 \text{ for } (w < CUT) \\ 0 \text{ for } (w \geq CUT) \end{Bmatrix}$$

Figure 4-22. XCORRS Equation, CUT Function Values

Figure 4-23 and Figure 4-24 show the bit placement for an `XCORRS` instruction using single-precision inputs (default operation. Note that Figure 4-23 and Figure 4-24 represent 16 multiply accumulate operations.

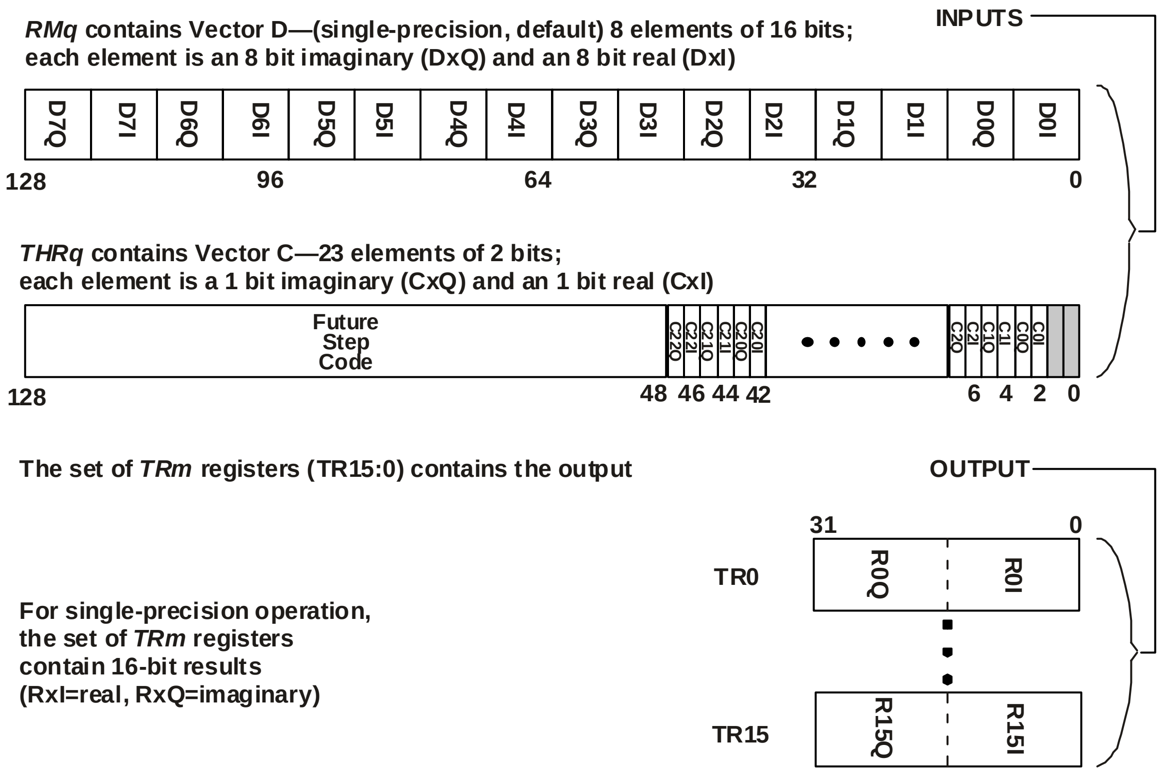

INPUTS
RMq contains Vector D—(single-precision, default) 8 elements of 16 bits; each element is an 8 bit imaginary (DxQ) and an 8 bit real (DxI)
D7Q D7I D6Q D6I D5Q D5I D4Q D4I D3Q D3I D2Q D2I D1Q D1I D0Q D0I
128 96 64 32 0
THRq contains Vector C—23 elements of 2 bits; each element is a 1 bit imaginary (CxQ) and an 1 bit real (CxI)
Future Step Code
C22Q C22I C21Q C21I C20Q C20I
C2Q C2I C1Q C1I C0Q C0I
128 48 46 44 42 6 4 2 0
The set of TRm registers (TR15:0) contains the output
OUTPUT
31 0
TR0 R0Q R0I
TR15 R15Q R15I
For single-precision operation, the set of TRm registers contain 16-bit results (RxI=real, RxQ=imaginary)

Figure 4-23. Bit fields in registers for:
TRm = XCORRS(Rmq, THRd);

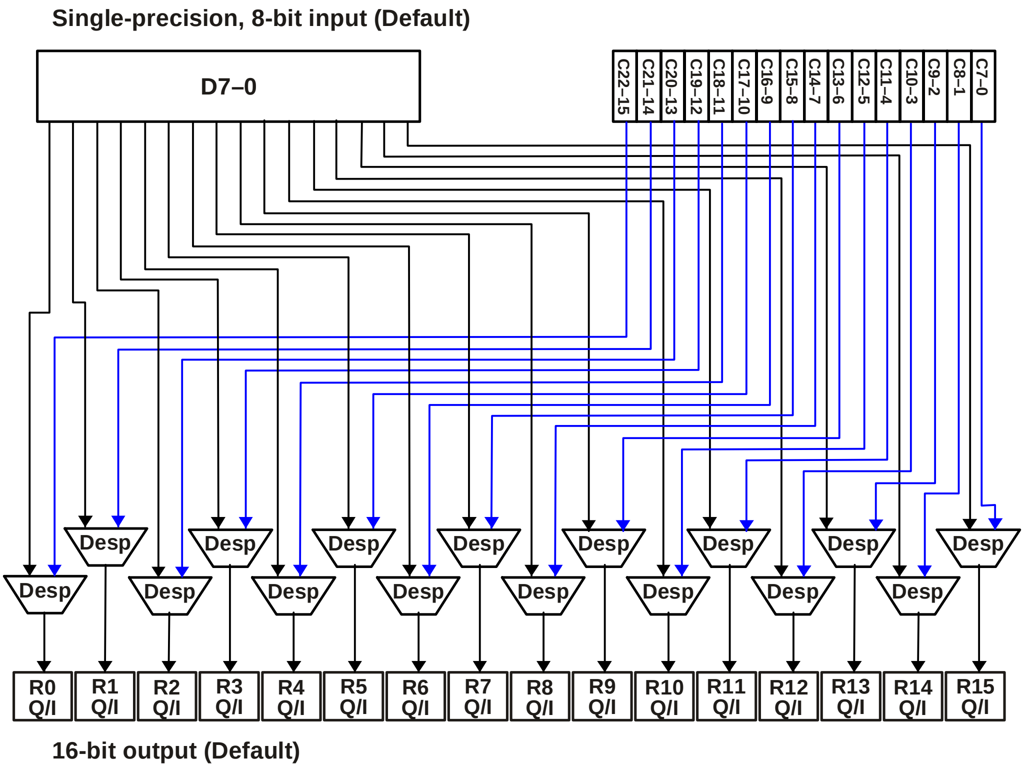


Figure 4-24. TRm = XCORRS(Rmq, THRd);

Figure 4-25 and Figure 4-26 show the bit placement for an XCORRS instruction using extended-precision inputs (EXT option operation. Note that Figure 4-25 and Figure 4-26 represent 8 multiply accumulate operations.

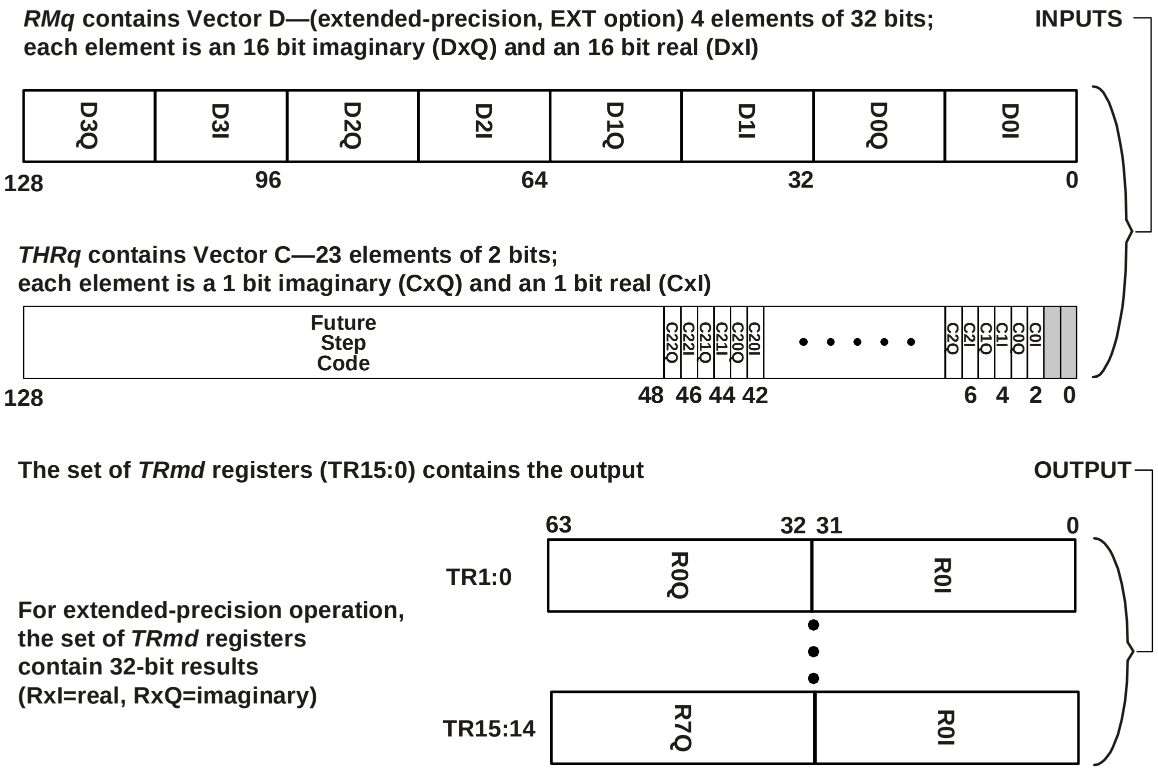

RMq contains Vector D—(extended-precision, EXT option) 4 elements of 32 bits;
each element is an 16 bit imaginary (DxQ) and an 16 bit real (DxI)
INPUTS
D3Q
D3I
D2Q
D2I
D1Q
D1I
D0Q
D0I
128
96
64
32
0
THRq contains Vector C—23 elements of 2 bits;
each element is a 1 bit imaginary (CxQ) and an 1 bit real (CxI)
Future Step Code
C22Q
C22I
C21Q
C21I
C20Q
C20I
C2Q
C2I
C1Q
C1I
C0Q
C0I
128
48
46
44
42
6
4
2
0
The set of *TRmd* registers (TR15:0) contains the output
OUTPUT
63
32
31
0
TR1:0
R0Q
R0I
For extended-precision operation, the set of *TRmd* registers contain 32-bit results (RxI=real, RxQ=imaginary)
TR15:14
R7Q
R0I

Figure 4-25. Bit fields in registers for:
TRm = XCORRS(Rmq, THRd) (EXT);

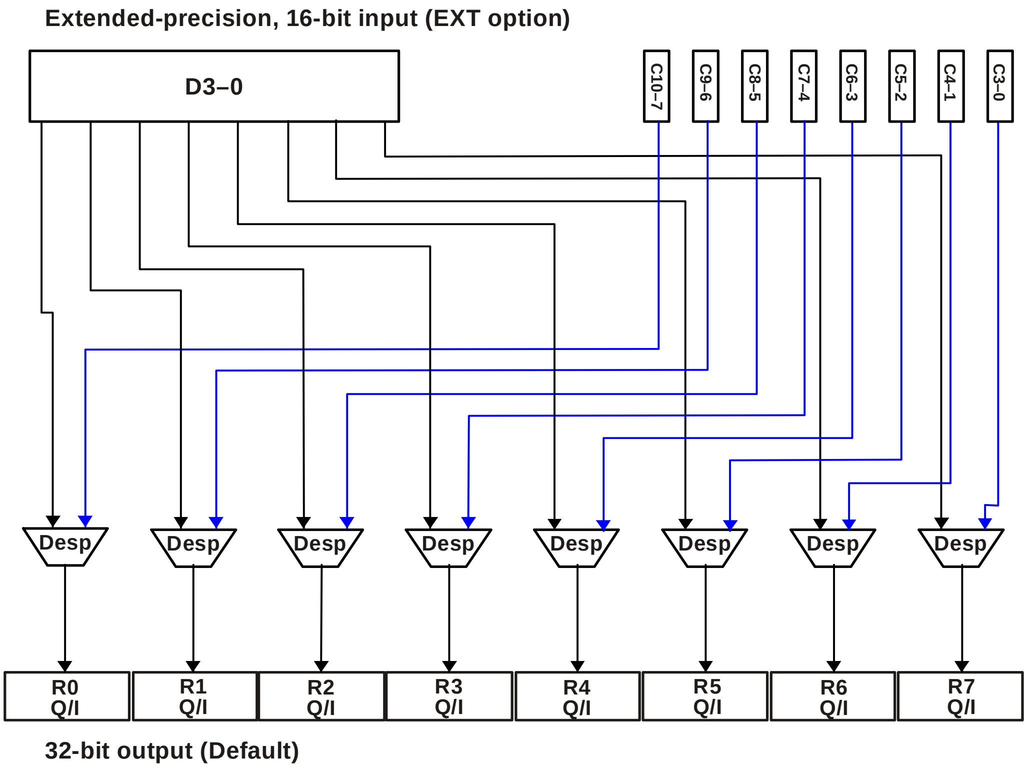


Figure 4-26. TRm = XCORRS(Rmq, THRd) (EXT);

When you execute this instruction in parallel to `THR` register load the `THR` load instruction takes priority on the `THR` shift in this instruction.

When the `CUT` option is omitted, the `CUT`=0, and the code index is as shown in Figure 4-27. Note that each code index in Figure 4-27 maps to a "C" input in Figure 4-24 on page 4-29.

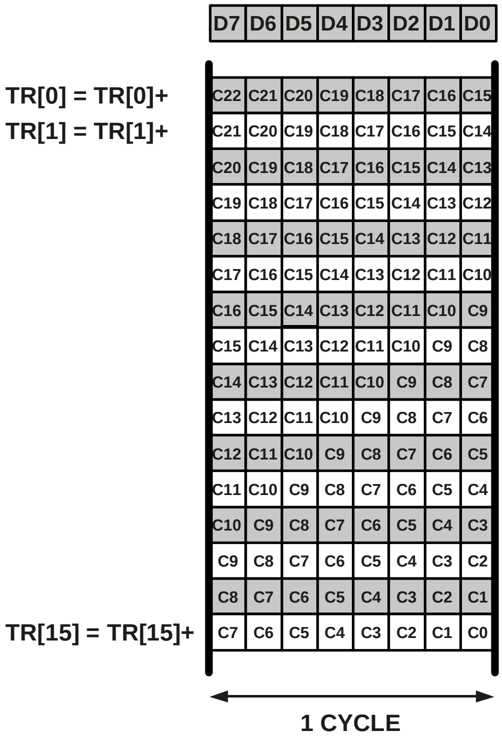

D7 D6 D5 D4 D3 D2 D1 D0
TR[0] = TR[0]+
TR[1] = TR[1]+
TR[15] = TR[15]+
C22 C21 C20 C19 C18 C17 C16 C15
C21 C20 C19 C18 C17 C16 C15 C14
C20 C19 C18 C17 C16 C15 C14 C13
C19 C18 C17 C16 C15 C14 C13 C12
C18 C17 C16 C15 C14 C13 C12 C11
C17 C16 C15 C14 C13 C12 C11 C10
C16 C15 C14 C13 C12 C11 C10 C9
C15 C14 C13 C12 C11 C10 C9 C8
C14 C13 C12 C11 C10 C9 C8 C7
C13 C12 C11 C10 C9 C8 C7 C6
C12 C11 C10 C9 C8 C7 C6 C5
C11 C10 C9 C8 C7 C6 C5 C4
C10 C9 C8 C7 C6 C5 C4 C3
C9 C8 C7 C6 C5 C4 C3 C2
C8 C7 C6 C5 C4 C3 C2 C1
C7 C6 C5 C4 C3 C2 C1 C0
1 CYCLE

Figure 4-27. CUT = 0 Definition – Code Index

When the `CUT` option is negative (for example, `CUT` = –k), all multiply-and-accumulate operations under `C[k]` are ignored.

Figure 4-28 shows an example with `CUT` = –7.

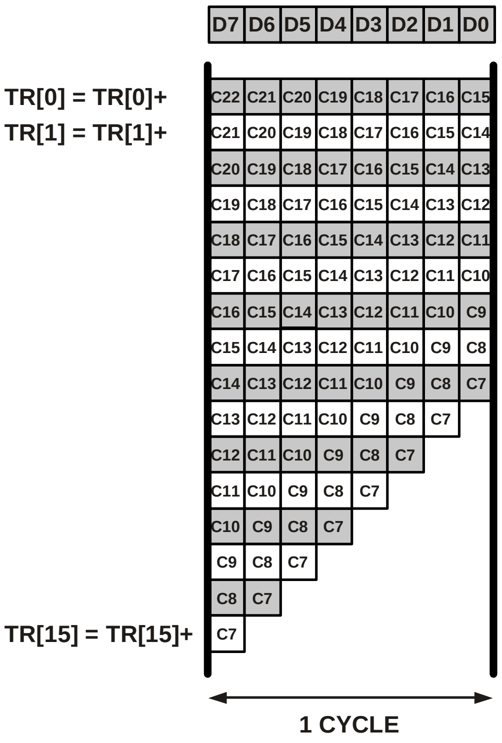

D7 D6 D5 D4 D3 D2 D1 D0
TR[0] = TR[0]+
TR[1] = TR[1]+
TR[15] = TR[15]+
C22 C21 C20 C19 C18 C17 C16 C15
C21 C20 C19 C18 C17 C16 C15 C14
C20 C19 C18 C17 C16 C15 C14 C13
C19 C18 C17 C16 C15 C14 C13 C12
C18 C17 C16 C15 C14 C13 C12 C11
C17 C16 C15 C14 C13 C12 C11 C10
C16 C15 C14 C13 C12 C11 C10 C9
C15 C14 C13 C12 C11 C10 C9 C8
C14 C13 C12 C11 C10 C9 C8 C7
C13 C12 C11 C10 C9 C8 C7
C12 C11 C10 C9 C8 C7
C11 C10 C9 C8 C7
C10 C9 C8 C7
C9 C8 C7
C8 C7
C7
1 CYCLE

Figure 4-28. CUT = Negative Example – Code Index

When the `CUT` option is positive (for example, `CUT = +k`), all multiply-and-accumulate operations for `C[k]` and above are ignored. Figure 4-29 shows an example with `CUT = +15`.

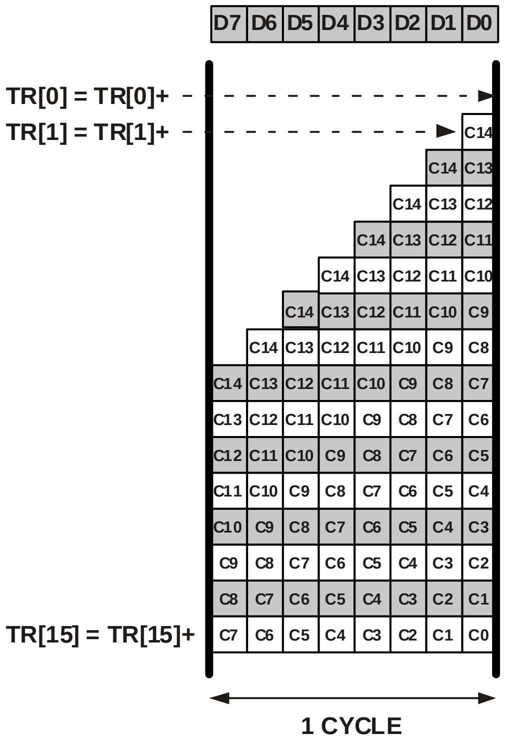


Figure 4-29. CUT = Positive Example – Code Index

When the TR registers are empty (for example, at the beginning of a new series of XCORRS instructions), the data fills the TR registers for the first two cycles as shown in Figure 4-30. This operation is similar to CUT = –15 in cycle one and CUT = –7 in cycle two.

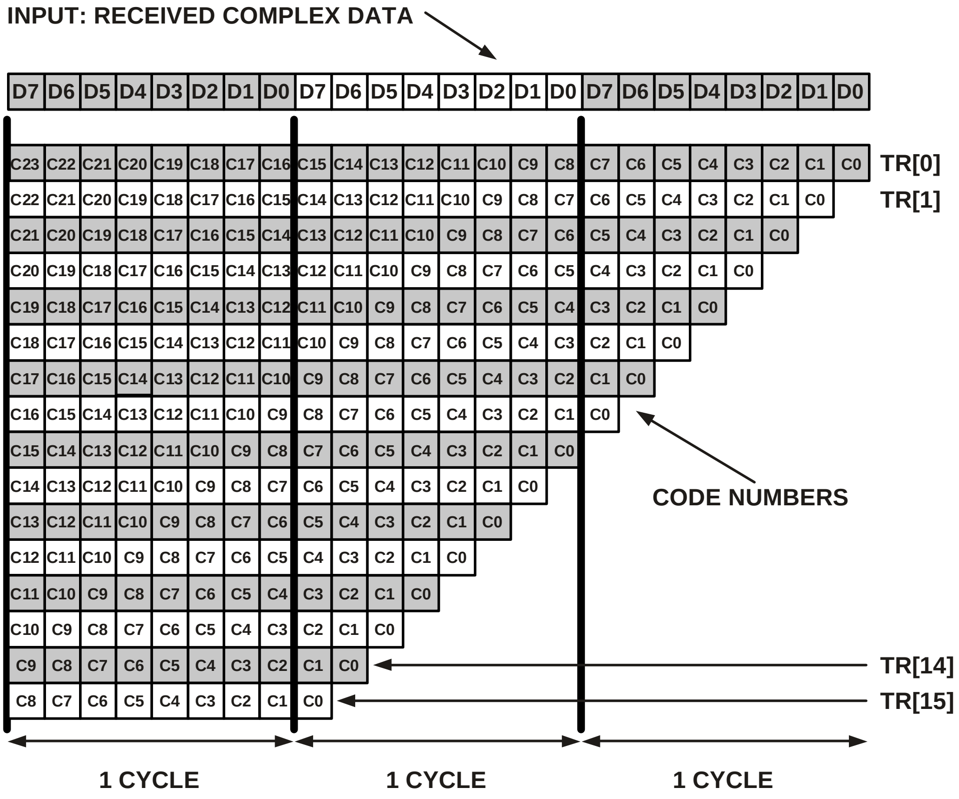

INPUT: RECEIVED COMPLEX DATA
D7 D6 D5 D4 D3 D2 D1 D0 D7 D6 D5 D4 D3 D2 D1 D0 D7 D6 D5 D4 D3 D2 D1 D0
TR[0]
TR[1]
TR[14]
TR[15]
CODE NUMBERS
1 CYCLE
1 CYCLE
1 CYCLE

Figure 4-30. Code Index – Example Data Start

The TR register are emptied at the end of the data set (for example, at the end of a series of XCORRS instructions), the data empties the TR registers during the last two cycles as shown in Figure 4-31. This operation is similar to CUT = 16 in the second-to-last cycle one and CUT = 8 in the last cycle.

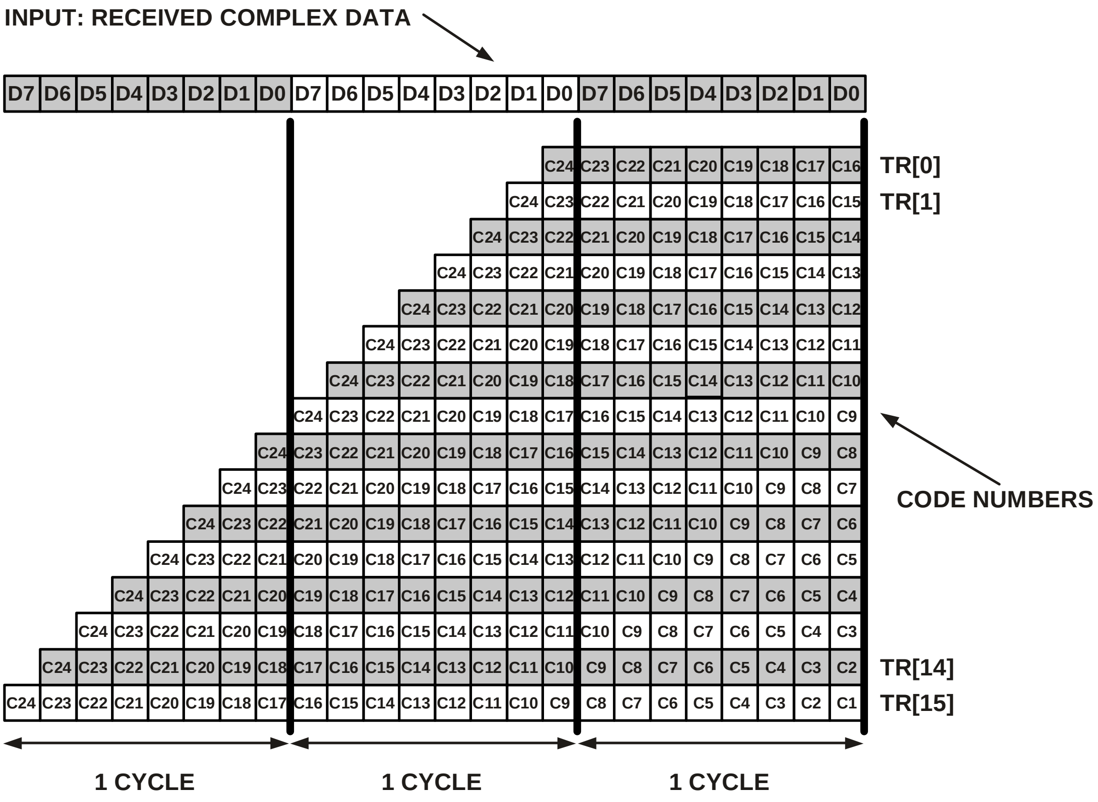

INPUT: RECEIVED COMPLEX DATA
D7 D6 D5 D4 D3 D2 D1 D0 D7 D6 D5 D4 D3 D2 D1 D0 D7 D6 D5 D4 D3 D2 D1 D0
TR[0]
TR[1]
TR[14]
TR[15]
CODE NUMBERS
C24 C23 C22 C21 C20 C19 C18 C17 C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1
1 CYCLE
1 CYCLE
1 CYCLE

Figure 4-31. Code Index – Example Data End

CLU Instruction Options

Some of the CLU instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions and options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at the end of the instruction's slot. For a list indicating which options apply for particular CLU instructions, see "CLU Instruction Summary" on page 4-40. The CLU instruction options include:

- `(TMAX)` selects `TMAX` operation instead of `MAX` operation[1]
- `(CUT` *`imm`*`)` selects data to cut from cross correlation set[2]
- `(CLR)` clears the selected trellis register prior to cross correlation summation[2]
- `(EXT)` selects extended precision for cross correlation input/results[2]

The following examples are CLU instructions that demonstrate arithmetic operations with options applied.

```
/* ADD EXAMPLES HERE*/
```

CLU Execution Status

CLU operations update status flags in the compute block's Arithmetic Status (`XSTAT` and `YSTAT`) register (see Figure 2-2 on page 2-4 and Figure 2-3 on page 2-5). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts.

[1] `ACS` instruction only
[2] `XCORRS` instruction only

Table 4-2 shows the flags in XSTAT or YSTAT that indicate CLU status (a 1 indicates the condition) for the most recent CLU operation.

Table 4-2. CLU Status Flags

Flag	Definition	Updated By…
TROV	CLU overflow	All CLU ops

CLU operations also update sticky status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register. Table 4-3 shows the flags in XSTAT or YSTAT that indicate CLU sticky status (a 1 indicates the condition) for the most recent CLU operation. Once set, a sticky flag remains high until explicitly cleared.

Table 4-3. CLU Status Sticky Flags

Flag	Definition	Updated By…
TRSOV	CLU overflow, sticky	All CLU ops

Flag update occurs at the end of each operation and is available on the next instruction slot. A program cannot write the arithmetic status register explicitly in the same cycle that the CLU is performing an operation.

Multi-operand instructions (for example, `STRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;`) produce multiple sets of results. In this case, the DSP determines a flag by ORing the result flag values from individual results.

CLU Examples

The communications logic unit (CLU) instructions are designed to support different algorithms used for communications applications. These instructions were designed primarily with the following algorithms in mind (although many other uses are possible):

- Viterbi Decoding
- Decoding of turbo codes
- Despreading for code-division multiple access (CDMA) systems

Listing 4-1 provides a number of example CLU arithmetic instructions. The comments with the instructions identify the key features of the instruction, such as input operand size and register usage.

Listing 4-1. CLU Instruction Examples

```
/*
   Examples are to be added in future revision of this document.
*/
```

CLU Instruction Summary

Listing 4-2 "CLU Instructions" shows the CLU instructions' syntax. The conventions used in these listings for representing register names, optional items, and choices are covered in detail in "Register File Registers" on page 2-5. Briefly, these conventions are:

- `{ }` – the curly braces enclose options; these braces are not part of the instruction syntax.
- `|` – the vertical bars separate choices; these bars are not part of the instruction syntax.
- *`Rmd`* – the register names in italic represent user-selectable single (*`Rs`*, *`Rm`*, or *`Rn`*), double (*`Rsd`*, *`Rmd`*, *`Rnd`*) or quad (*`Rsq`*) register names.

Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see "Instruction Line Syntax and Structure" on page 1-22 and "Instruction Parallelism Rules" on page 1-26.

Listing 4-2. CLU Instructions

```
{X|Y|XY}{S}TRsd = TMAX(TRmd + Rmq_h, TRnd + Rmq_l) ;
{X|Y|XY}{S}TRsd = TMAX(TRmd - Rmq_h, TRnd - Rmq_l) ;
{X|Y|XY}{S}Rs = TMAX(TRm, TRn) ;
{X|Y|XY}{S}TRsd = MAX(TRmd + Rmq_h, TRnd + Rmq_l) ;
{X|Y|XY}{S}TRsd = MAX(TRmd - Rmq_h, TRnd - Rmq_l) ;
{X|Y|XY}Rs = TRm ;
{X|Y|XY}Rsd = TRmd ;
{X|Y|XY}Rsq = TRmq ;
{X|Y|XY}TRs = Rm ;
{X|Y|XY}TRsd = Rmd ;
{X|Y|XY}TRsq = Rmq ;
```

```
{X|Y|XY}Rs = THRm ;
{X|Y|XY}Rsd = THRmd ;
{X|Y|XY}Rsq = THRmq ;[1]
{X|Y|XY}THRs = Rm ;
{X|Y|XY}THRsd = Rmd {(i)} ;
{X|Y|XY}THRsq = Rmq ;[1]
{X|Y|XY}TRs = DESPREAD (Rmq, THRd) + TRn ;
{X|Y|XY}Rs = TRs, TRs = DESPREAD (Rmq, THRd) ; (dual op)
{X|Y|XY}Rsd = TRsd, TRsd = DESPREAD (Rmq, THRd) ; (dual op)
{X|Y|XY}TRsa = XCORRS (Rmq, THRnq) {(CUT <Imm>|R)}{(CLR)}{(EXT)};
{X|Y|XY}Rsq = TRbq, TRsa = XCORRS (Rmq,THRnq)
  {(CUT <Imm>|R)}{(CLR)}{(EXT)} ; (dual operation)
/* where TRsa = TR15:0 or TR31:16 */
{X|Y|XY}{S}TRsq = ACS (TRmd, TRnd, Rm) {(TMAX)} ;
{X|Y|XY}Rsq = TRaq, {S}TRsq = ACS (TRmd, TRnd, Rm)
  {(TMAX)} ; (dual op)
{X|Y|XY}Rsd = PERMUTE (Rmd, Rn) ;
{X|Y|XY}Rsq = PERMUTE (Rmd, -Rmd, Rn) ;
```

[1] Not implemented, but syntax reserved

5 MULTIPLIER

The ADSP-TS201 TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The multiplier is highlighted in Figure 5-1. The multiplier takes its inputs from the register file, and returns its outputs to the register file.

The chapter describes:

- "Multiplier Operations" on page 5-5
- "Multiplier Examples" on page 5-22
- "Multiplier Instruction Summary" on page 5-24

The multiplier performs all *multiply operations* for the processor on fixed- and floating-point data and performs all *multiply-accumulate operations* for the processor on fixed-point data. This unit also performs all *complex multiply operations* for the processor on fixed-point data. The multiplier also executes *data compaction operations* on accumulated results when moving data to the register file in fixed-point formats.

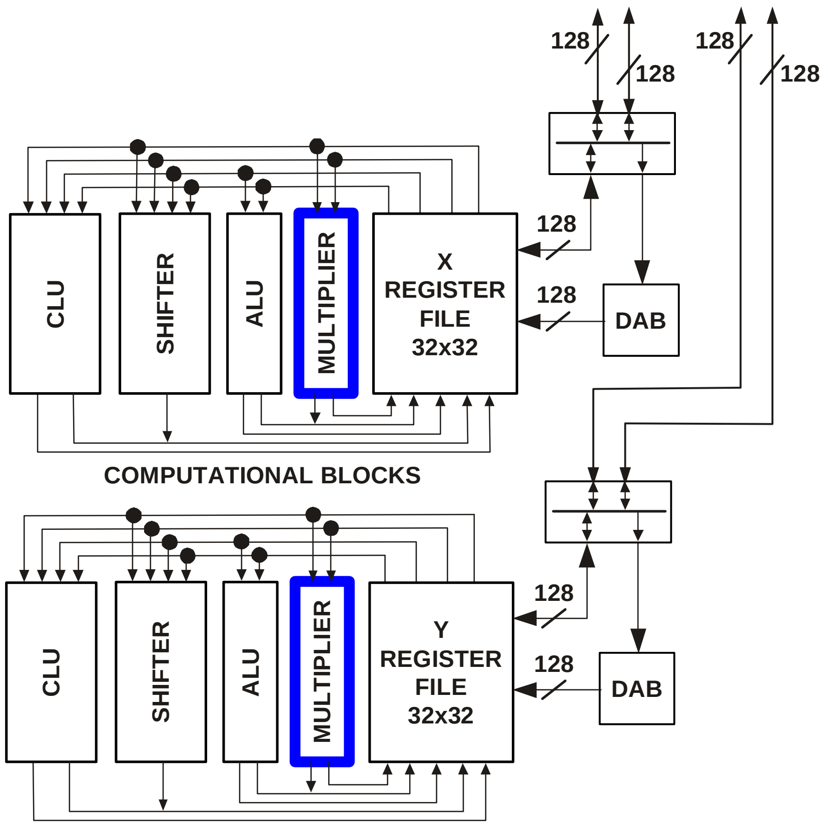


Figure 5-1. Multipliers in Compute Block X and Y

Examining the supported operands for each operation shows that the multiplier operations support these data types:

- Fixed-point fractional and integer *multiply operations* and *multiply-accumulate operations* support:
    - Eight 16-bit (short) input operands with four 16- or 32-bit results
    - Two 32-bit (normal) input operands with a 32- or 64-bit result
- Floating-point fractional *multiply operations* support:
    - Two 32-bit (single-precision) input operands (IEEE standard) with 32-bit result
    - Two 40-bit (extended precision) input operands with 40-bit result
- Fixed-point *data compaction operations* support:
    - 20-bit (short) input operands
    - 40-bit (normal) input operands
    - 80-bit (long) input operands
    - output 16- or 32-bit results

Fixed-point formats include these data size distinctions:

- The multiplier can operate on two 32-bit normal words producing either a 64-bit or a 32-bit result or operate on eight 16-bit short words producing either four 32-bit or four 16-bit results. There is no byte word support in the multiplier.
- The result of a multiplier operation (with the exception of the compress instruction) is always either the same size as the operands, or larger.
  - — Normal word multiplication results in either a normal word or a long word result.
  - — Quad short word multiplication always results in either a quad short-word or a quad-word results.

The ADSP-TS201 processor supports complex multiply-accumulates. Complex numbers are represented by a pair of short words in a 32-bit register. The least significant bits of the input operands (`Rm_L`, `Rn_L`) represent the real part, and the most significant bits of the input operands (`Rm_H`, `Rn_H`) represent the imaginary part. The result of a complex multiplication is always stored in a pair of `MR` registers. The complex multiply-accumulate (indicated with the `**` operator) is defined as:

$$\mathit{Real\ Result} = (\mathit{Real}_{Rm\_L} \times \mathit{Real}_{Rn\_L}) - (\mathit{Imaginary}_{Rm\_H} \times \mathit{Imaginary}_{Rn\_H})$$

$$\mathit{Imaginary\ Result} = (\mathit{Real}_{Rm\_L} \times \mathit{Imaginary}_{Rn\_H}) + (\mathit{Imaginary}_{Rm\_H} \times \mathit{Real}_{Rn\_L})$$

Complex multiply-accumulate operations have an option to multiply the first complex operand times the complex conjugate of the second. This complex conjugate operation is defined as:

$$\mathit{Real\ Result} = (\mathit{Real}_{Rm\_L} \times \mathit{Real}_{Rn\_L}) + (\mathit{Imaginary}_{Rm\_H} \times \mathit{Imaginary}_{Rn\_H})$$

$$\mathit{Imaginary\ Result} = (\mathit{Real}_{Rm\_L} \times \mathit{Imaginary}_{Rn\_H}) + (\mathit{Imaginary}_{Rm\_H} \times \mathit{Real}_{Rn\_L})$$

The complex conjugate option is denoted with a `(J)` following the instruction. (See "Complex Conjugate Option" on page 5-17.)

The ADSP-TS201 TigerSHARC processor is compatible with the IEEE 32-bit single-precision floating-point data format with minor exceptions. For more information, see "IEEE Single-Precision Floating-Point Data Format" on page 2-15.

Within instructions, the register name syntax identifies the input operand and output result data size and type. For information on data type selection for multiplier instructions, see "Register File Registers" on page 2-5. For information on data size selection for multiplier instructions, see the examples in "Multiplier Operations" on page 5-5.

Note that multiplier instruction conventions for selecting input operand and output result data size differ slightly from the conventions for the ALU and shifter.

The remainder of this chapter presents descriptions of multiplier instructions, options, and results using instruction syntax. For an explanation of the instruction syntax conventions used in multiplier and other instructions, see "Instruction Line Syntax and Structure" on page 1-22. For a list of multiplier instructions and their syntax, see "Multiplier Instruction Summary" on page 5-24.

Multiplier Operations

The multiplier performs fixed-point or floating point multiplication and fixed-point multiply-accumulate operations. The multiplier supports several data types in fixed- and floating-point. The floating-point formats are float and float-extended. The input operands and output result of most operations is the compute block register file.

The multiplier has one special purpose, five-word register—the `MR` register—for accumulated results. The ADSP-TS201 processor uses the `MR` register to store the results of fixed-point multiply-accumulate operations. Also, the multiplier can transfer the contents of the `MR` register to the register file before an accumulate operation. The upper 32 bits of the `MR` register (`MR4`) store overflow for multiply accumulate operations. For more information on the register files and register naming syntax for selecting data type and width, see "Register File Registers" on page 2-5.

Figure 5-2 on page 5-8 through Figure 5-4 on page 5-9 show the data flow for multiplier operations. The following are multiplier instructions that demonstrate multiply and multiply-accumulate operations.

```
XR2 = R1 * R0 ;;

/* This is a fixed-point multiply of two signed fractional 32-bit
input operands XR1 and XR0; the DSP places the rounded 32-bit
result in XR2. */

YR1:0 = R3 * R2 ;;

/* This is a fixed-point multiply of two signed fractional 32-bit
input operands YR3 and YR2; the DSP places the 64-bit result in
YR1:0. */
```

For fixed-point multiply operations, single register names (*`Rm`*, *`Rn`*) for input operands select 32-bit input operands. Single versus double register names output for select 32- versus 64-bit output results.

```
XR3:2 = R5:4 * R1:0 ;;

/* This is a fixed-point multiply of eight signed fractional
16-bit operands:
XR5_upper_half * XR1_upper_half,
XR5_lower_half * XR1_lower_half,
XR4_upper_half * XR0_upper_half,
XR4_lower_half * XR0_lower_half;
```

```
The DSP places the four rounded 16-bit results in XR3_upper_half,
XR3_lower_half, XR2_upper_half, and XR2_lower_half, respectively
*/

YR3:0 = R7:6 * R5:4 ;;

/* This is similar to the previous example of a quad 16-bit mul-
tiply, but the selection of a quad register for output produces
32-bit (instead of 16-bit) results; the DSP places the four
results in YR3, YR2, YR1, and YR0. */
```

For fixed-point multiply operations, double register names (`Rmd`, `Rnd`) for input operands select 16-bit input operands. Double versus quad register names for output select 16- versus 32-bit output results.

```
XFR2 = R1 * R0 ;;

/* This is a floating-point multiply of two single precision
input operands XR1 and XR0 (IEEE format); the DSP places the
single precision result in XR2. */

YFR1:0 = R5:4 * R3:2 ::

/* This is a floating-point multiply of two extended precision
40-bit input operands YR5:4 and YR3:2; the DSP places the 40-bit
result in YR1:0. */
```

For floating-point multiply operations, single register names (`Rm`, `Rn`) for input operands select 32-bit input operands and 32-bit output result. For floating-point multiply operations, double register names (`Rmd`, `Rnd`, `Rsd`) for input and output operands select 40-bit input operands and 40-bit output result.

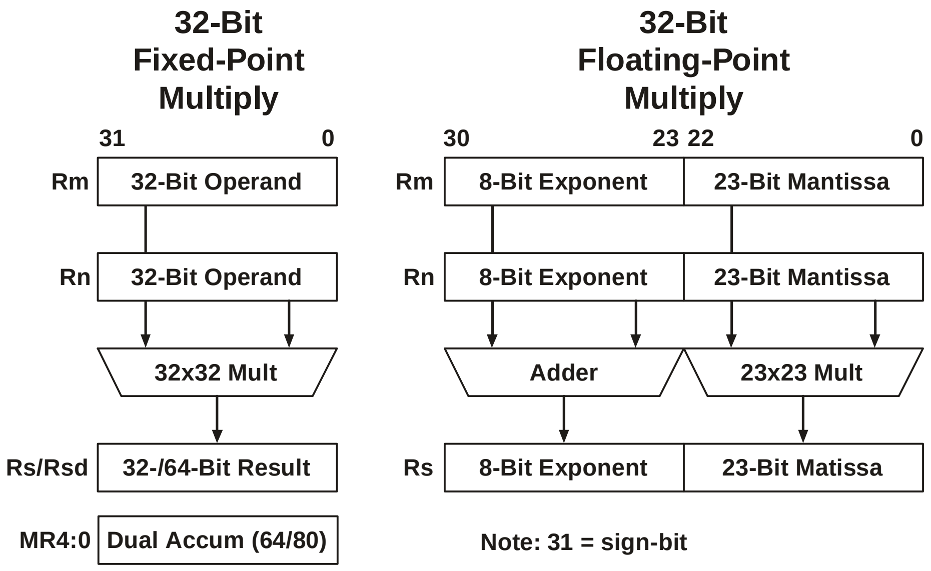


Figure 5-2. 32-Bit Multiplier Operations

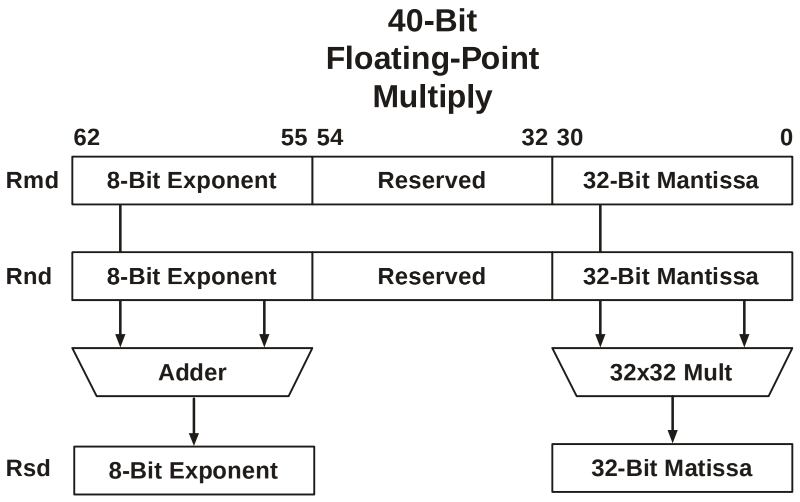


Figure 5-3. 40-Bit Multiplier Operations

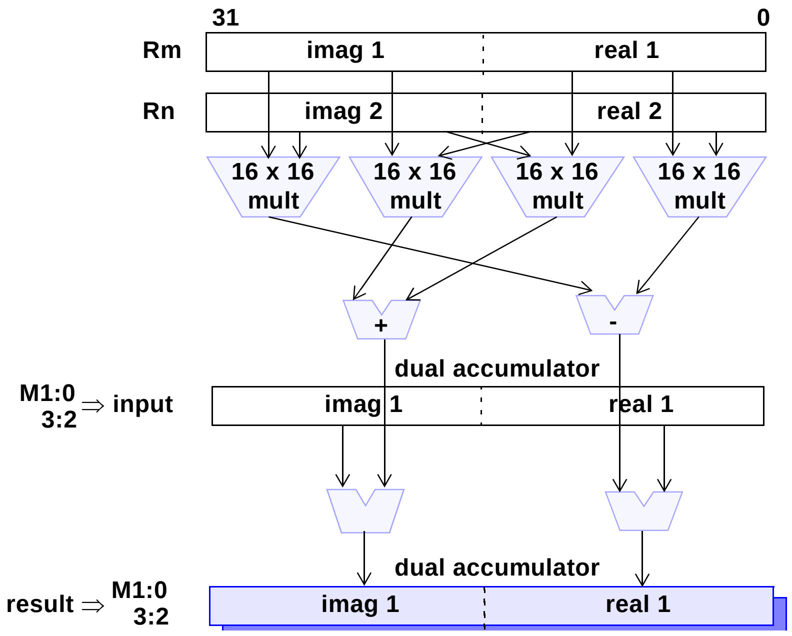


Figure 5-4. 16-Bit Complex Multiplier Operations

Multiplier Instruction Options

Most of the multiplier instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions and options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at the end of the instruction's slot.

For a list indicating which options apply for particular multiplier instructions, see "Multiplier Instruction Summary" on page 5-24.

The multiplier instruction options include:

- `()` signed operation, no saturation[1], round-to-nearest even[2], fractional mode[3]
- `(U)` unsigned operation, no saturation[1], round-to-nearest even[2]
- `(nU)` signed/unsigned input
- `(I)` signed operation, integer mode[3]
- `(S)` signed operation, saturation[1]
- `(T)` signed operation, truncation[4]
- `(C)` clear operation
- `(CR)` clear/round operation
- `(J)` complex conjugate operation

The following are multiplier instructions that demonstrate multiply and multiply-accumulate operations with options applied.

```
XR2 = R1 * R0 (U) ;;

/* This is a fixed-point multiply of two unsigned fractional
32-bit input operands XR1 and XR0; the DSP places the rounded
unsigned 32-bit result in XR2. */
```

[1] Where saturation applies
[2] Where rounding applies
[3] Where applies for fixed-point operations
[4] Where truncation applies

```
YR1:0 = R3 * R2 (I) ::

/* This is a fixed-point multiply of two integer 32-bit input
operands YR3 and YR2; the DSP places the 64-bit result in YR1:0.
*/

XFR2 = R1 * R0 (T) ;;

/* This is a floating-point multiply of two fractional 32-bit
input operands XR1 and XR0 (IEEE format); the DSP places the
truncated 32-bit result in XR2. */
```

Signed/Unsigned Option

The processor always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. Fixed-point data in the multiplier may be unsigned or two's-complement. Floating-point data in the multiplier is always signed-magnitude. For information on the supported numeric formats, see "Numeric Formats" on page 2-15.

All fixed-point multiplier instructions may use signed or unsigned data types. The options are:

`( )`	. Both input operands signed (default)
`(U)`	. Both input operands unsigned
`(nU)`	. `Rm` is signed, `Rn` is unsigned; this option is valid only for `Rs = Rm * Rn` or `Rsd = Rm * Rn`

Fractional/Integer Option

The processor always represents fixed-point numbers in 8, 16, 32, or 64 bits, using up to four 32-bit data registers. In the multiplier, fractional or integer format is available for the fixed-point multiply, multiply-accumulate and `COMPACT` instructions. For information on the supported numeric formats, see "Numeric Formats" on page 2-15.

The integer and fractional option are defined for the fixed-point operations:

`( )`	. Data is fractional (default)
`(I)`	. Data is integer

Saturation Option

Saturation is performed on fixed-point operations if option `(S)` is active[1] when the result *overflows*—crosses the maximum value in which the result format can be represented. In these cases, the returned result is the extreme value that can be represented in the format of the operation following the direction of the correct result. For example, if the format of the result is 16-bit signed and the full result is `-0x100000`, the saturated result would be `0x8000`. If the operation is unsigned, the result would be `0x0`; however, there could not a negative result for the unsigned operation. This can occur in three types of operations:

- Multiply operations

  When multiplying integer data and the actual result is outside the range of the destination format, the largest representable number in the destination format is returned. When multiplying fractional data, the special case of -1 times -1 (for example, `0x80…0` times `0x80…0`) always returns the largest representable fraction (for example, `0x7F…F`), if saturation is chosen. (See Note 1).

- Multiply-accumulate operations

  Saturation affects both integer and fractional data types. Accumulation values are kept at 80-, 40-, and 20-bit precision and are stored in the combination of `MR3:0` and `MR4` registers (See “Multiplier Examples” on page 5-22). When performing saturation in a

[1] Except for `Rsd = Rm * Rn` ,`Rsq = Rmd * Rnd`, where it is not optional and implied in the instruction.

multiply-accumulate operation, the resulting value out of the multiplier (at 64, 32, or 16 bits) is added to the current accumulation value. When the accumulation value overflows past 80, 40, or 20 bits, it is substituted by the maximum or minimum possible value. Note that multiply-accumulate operations always saturate.

- `MR` register transfers

  See "Multiplier Examples" on page 5-22.

The final saturated result at 32 --bits for all operations is:

- `0x7F…F` – if operation is signed and result is positive
- `0x80…0` – if operation is signed and result is negative
- `0xFF…F` – if operation is unsigned and result is positive
- `0x00…0` – if operation is unsigned and result is negative (only in signed `MR -= Rm * Rn`)

Saturation option exists for any fixed-point multiplications that may overflow. The following options are available:

`( )`	. No saturation (default)
`(S)`	. Saturation is active

Truncation Option

For multiplier instructions that support truncation as the (T) option, this option permits selection of the results rounding mode. The processor supports two modes of rounding—round-toward-zero and round-toward-nearest. The rounding modes comply with the IEEE 754 standard and have these definitions:

- Round-Toward-Nearest—not using (T) option. If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is the number that has an LSB equal to zero.

- Round-Toward-Zero—using (T) option. If the result before rounding is not exactly representable in the destination format, the rounded result is the number that is nearer to zero. This is equivalent to truncation.

Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

Though these rounding modes comply with standards set for floating-point data, they also apply for fixed-point multiplier operations on fractional data. The same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Using its local result register for fixed-point operations, the multiplier round-to-minus infinity by reading only the upper bits of the result and discarding the lower bits.

Round-to-nearest even is executed by adding the LSB to the truncated result if the first bit under the LSB is set and all bits under are cleared—for example, multiplying two short operands. The real result is a normal word. If the expected result is short, it has to be rounded.

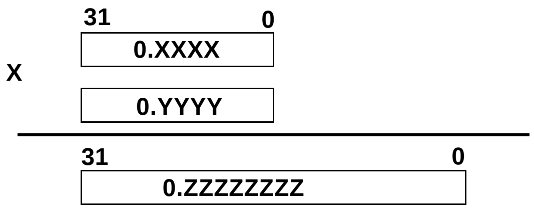


Figure 5-5. Rounding Multiplier Results

Bits 31–16 should be returned, and bits 15–0 should be rounded. The rounding is set according to bit 15 (*round bit*), bit 16 (LSB), and whether bits 14–0 (lower bits) are zero or non-zero:

- If bit 15 is zero, the result is not incremented
- If bit 15 is 1 and bits 14–0 are non-zero, the LSB is incremented by one
- If bit 15 is 1 and bits 14–0 are zero, add bit 16 to the result 31–16

There is no support for round-to-nearest even for multiply-accumulate instructions that also transfer the current `MR` contents to the register file. If round-to-nearest even is required, transfer the `MR` registers to the register file as a whole and use the ALU `COMPACT` instruction. As an alternative, if round-to-nearest (not even) is sufficient, this can be achieved by using the clear and round option in the first multiply-accumulate instruction of the series. For more information, see "Clear/Round Option" on page 5-15.

Rounding options are:

`( )`	. Round-to-nearest even (default)
`(T)`	. Truncate—round-to-zero for floating-point and round-to-minus infinity for fixed-point format

Clear/Round Option

Multiply operations and multiply-accumulate with `MR` register move operations support the clear `MR` (`C`) option. Using this option forces the multiplier to clear (=0) the relevant `MR` register part before the accumulate operation.

Multiply-accumulate operations (without an `MR` move[1]) also support the clear and round (`CR`) options as an alternative to the clear option.

[1] This is not a critetion. The distinction is the destination result width:
`Rs = MRa, MRa += Rm * Rn (R).`

Using the `CR` option forces the multiplier to clear `MR` and set the round bit before the accumulate operation. For more information about rounding and the round bit, see "Truncation Option" on page 5-13.

The clear and clear/round options are:

`( )`	. No change of `MR` register prior to multiply-accumulate operation (default)
`(C)`	. Set target `MR` to zero prior to multiply-accumulate operation
`(CR)`	. Set target `MR` to zero and set round bit prior to multiply-accumulate operation

ⓘ The `CR` options may be used only for fractional arithmetic, for example when the option `(I)` is not used.

When this option is set, the `MR` registers are set to an initial value of `0x00000000 80000000` for 32-bit fractional multiply-accumulate and `0x00008000` in each of the `MR` registers for quad 16-bit fractional multiply-accumulate. After this initialization, the result is rounded up by storing the upper part of the result in the end of the multiply-accumulate sequence.

For example with the `CR` option, assume a sequence of three quad short fractional multiply-accumulate operations (with quad short result) such that the multiplication results are:

```
Result 1 = 0x0024 0048, 0x0629 4501

Result 2 = 0x0128 0128, 0x2470 2885

Result 3 = 0x1011 fffe, 0x4A30 6d40

Sum      = 0x115d 016e, 0x74c9 dac6
```

In this example, the bottom 16 bits are not to be used if only a short result is expected. Extracting the top 16 bits will give a truncated result, which is `0x115d` for the first short and `0x74c9` for the second short. The rounded result is `0x115d` for the first short (no change) and `0x74ca` (increment) for the second short. If the `MR` registers are initialized to `0x00008000`, the sum result will be:

```
Sum       = 0x115d 816e, 0x74ca 5ac6
```

The two shorts are exactly the expected results. Note that this is round-to-nearest, and not round-to-nearest even.

The high short is exactly as expected. The rounding method is round-to-nearest, not round-to-nearest even. For information on rounding, see "Truncation Option" on page 5-13.

Complex Conjugate Option

For complex multiply-accumulate operations (** operator), the multiplier supports the complex conjugate `(J)` option. The `J` option directs the multiplier to multiply the `Rm` operand with the complex conjugate of *`Rn`* operand, negating the imaginary part of `Rn`. For more information, see the discussion on page 5-4. The options are:

`( )`	. No conjugate
`(J)`	. Conjugate for complex multiply

Multiplier Result Overflow (MR4) Register

The MR4 register holds the extra bits (overflow) from a multiply-accumulate operation. MR4 register fields are assigned according to the MR register used and the size of the result. See:

- Figure 5-6–Result is a long word (80-bit accumulation)
- Figure 5-7–Result is word (40-bit accumulation)
- Figure 5-8–Result is short (20-bit accumulation)

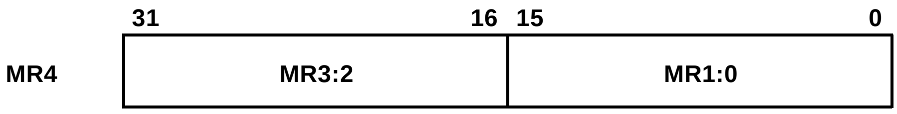


Figure 5-6. MR4 for Long Word Result (80-Bit Accumulation)

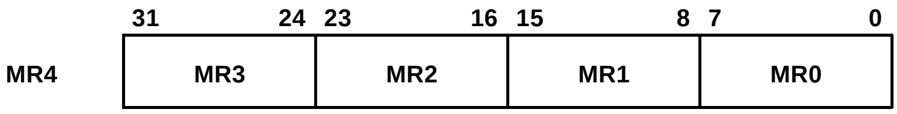


Figure 5-7. MR4 for Normal Word Result (40-Bit Accumulation)

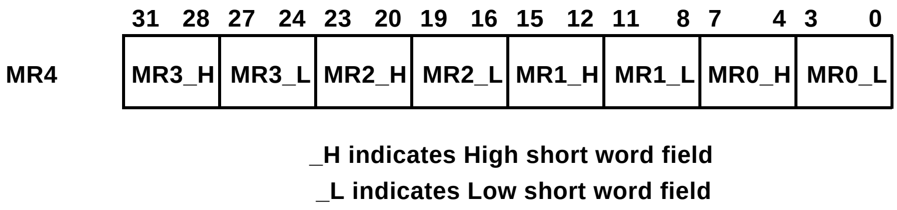


Figure 5-8. MR4 for Short Word Result (20-Bit Accumulation)

These bits are also used as the input to the accumulate step of the multiply-accumulate operation. The bits are cleared together with the clear of the corresponding MR register, and when stored, are used for saturation. The purpose of these bits is to enable the partial result of a multiply-accumulate sequence to go beyond the range assigned by the final result.

Multiplier Execution Status

Multiplier operations update status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register (see Figure 2-2 on page 2-4 and Figure 2-3 on page 2-5). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see "Multiplier Execution Conditions" on page 5-20.

Table 5-1 shows the flags in XSTAT or YSTAT that indicate multiplier status (a 1 indicates the condition) for the most recent multiplier operation.

Table 5-1. Multiplier Status Flags

Flag	Definitions	Updated By…
MZ	Multiplier fixed-point zero and floating-point underflow or zero	All fixed- and floating-point multiplier ops, except multply accumulate
MN	Multiplier result is negative	All fixed- and floating-point multiplier ops, except multply accumulate
MV	Multiplier overflow	All fixed- and floating-point multiplier ops, except multply accumulate
MU	Multiplier underflow	All floating-point multiplier ops; cleared by fixed-point ops, unchanged by multply accumulate
MI	Multiplier floating-point invalid operation	All floating-point multiplier ops; cleared by fixed-point ops, unchanged by multply accumulate

Multiplier operations also update sticky status flags in the compute block's Arithmetic Status (`XSTAT` and `YSTAT`) register. Table 5-2 shows the flags in `XSTAT` or `YSTAT` that indicate multiplier sticky status (a 1 indicates the condition) for the most recent multiplier operation. Once set, a sticky flag remains high until explicitly cleared.

Table 5-2. Multiplier Sticky Status Flags

Flag	Definition	Updated By…
MUS	Multiplier underflow, sticky	All floating-point multiply ops
MVS	Multiplier floating-point overflow, sticky	All floating-point multiply ops
MOS	Multiplier fixed-point overflow, sticky	All fixed-point multiply ops
MIS	Multiplier floating-point invalid operation, sticky	All floating-point multiply ops

Flag update occurs at the end of each operation and is available on the next instruction slot. A program cannot write the arithmetic status register explicitly in the same cycle that the multiplier is performing an operation.

Multi-operand instructions (for example, `Rsd = Rmd * Rnd`) produce multiple sets of results. In this case, the processor determines a flag by ORing the result flag values from individual results.

Multiplier Execution Conditions

In a conditional multiplier instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional multiplier instructions take the form:

```
IF cond; DO, instr.; DO, instr.; DO, instr. ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `DO` before the instruction makes the instruction unconditional.

Table 5-3 lists the multiplier conditions. For more information on conditional instructions, see "Conditional Execution" on page 8-12.

Table 5-3. Multiplier Conditions

Condition	Description	Flags Set
MEQ	Multiplier equal to zero	MZ = 1
MLT	Multiplier less than zero	MN and MZ = 1
MLE	Multiplier less than or equal to zero	MN or MZ = 1
NMEQ	NOT (Multiplier equal to zero)	MZ = 0
NMLT	NOT (Multiplier less than zero)	MN or MZ = 0
NMLE	NOT (Multiplier less than or equal to zero)	MN and MZ = 0

Multiplier Static Flags

In the program sequencer, the static flag (`SFREG`) can store status flag values for later usage in conditional instructions. With `SFREG`, each compute block has two dedicated static flags `X/YSCF0` (condition is `SF0`) and `X/YSCF1` (condition is `SF1`). The following example shows how to load a compute block condition value into a static flag register.

```
XSCF0 = XMEQ ;; /* Load X-compute block MEQ flag into XSCF0 bit
in static flags (SFREG) register */

IF SF0, XR5 = R4 * R3 ;; /* the SF0 condition tests the XSCF0
static flag */
```

For more information on static flags, see "Conditional Execution" on page 8-12.

Multiplier Examples

Listing 5-1 provides a number of example multiply and multiply-accumulate instructions. The comments with the instructions identify the key features of the instruction, such as fixed- or floating-point format, input operand size, and register usage.

Listing 5-1. Multiplier Instruction Examples

```
XYR4 = R6 * R8 ;;

/* This instruction is a 32-bit fractional multiply that produces
a 32-bit rounded result. */

XYR5:4 = R6 * R8 ;;

/* This instruction is a 32-bit fractional multiply that produces
a 64-bit result. */

XR11:10 =  R9:8 * R7:6 ;;

/* This instruction is a quad 16-bit multiply; the input operands
are XR9_H x XR7_H, XR9_L x XR7_L, XR8_H x XR6_H, and
XR8_L x XR6_L (where _H is high half and _L is low half); the
16-bit results go to XR11_H, XR11_L, XR10_H, and XR10_L (respec-
tively). */

XMR3:2 += R1 * R0 ;;

/* This is a multiplication of source operands XR1 and XR0, and
the multiplication result is added to the current contents of the
target XMR registers, overflowing into XMR4_H. */

YMR1:0 -= R3 * R2 ;;

/* This is a multiplication of source operands YR3 and YR2, and
the multiplication result is subtracted from the current contents
of the target YMR registers, overflowing into YMR4_L. */

XR7 = MR3:2, MR3:2 += R1 * R0 ;;
```

```
/* This instruction executes a multiply-accumulate and transfers
the previous MR registers into the register file; the previous
value in the MR registers is transferred to the register file. */

YMR3:0 += R5:4 * R7:6 ;;

/* This instruction is four multiplications of four 16-bit shorts
in register pair YR5:4 and four 16-bit shorts in pair YR7:6. The
four results are accumulated in MR3:0 as a word result. The over-
flow bits are written into MR4. */

XMR3:2 += R9:8 * R7:6 ;;

/* This instruction is a quad 16-bit multiply-accumulate with
16-bit results; the input operands are XR9_H x XR7_H,
XR9_L x XR7_L, XR8_H x XR6_H, and XR8_L x XR6_L (where _H is high
half and _L is low half); the 16-bit accumulated results go to
XMR3_H, XMR3_L, XMR2_H, and XMR2_L (respectively). */

MR3:0 += R9:8 * R7:6 ;;

/* This instruction is a quad 16-bit multiply-accumulate with
32-bit results; the input operands are XR9_H x XR7_H,
XR9_L x XR7_L, XR8_H x XR6_H, and XR8_L x XR6_L (where _H is high
half and _L is low half); the 32-bit accumulated results go to
XMR3, XMR2, XMR1, and XMR0 (respectively). */

XMR1:0 += R9 ** R7 ;;

/* This instruction is a multiplication of the complex value in
XR9 and the complex value in XR7. The result is accumulated in
XMR1:0. */

XFR20 = R22 * R23  (T) ;;

/* This is a 32-bit (single precision) floating-point multiply
instruction with 32-bit result; single registers select 32-bit
operation. */
```

```
YFR25:24 = R27:26 * R30:29 (T) ;;

/* This is a 40-bit (extended precision) floating-point multiply
instruction with 40-bit result; double registers select 40-bit
operation. */
```

Multiplier Instruction Summary

The following listings show the multiplier instructions' syntax:

- Listing 5-2 on page 5-25 "32-Bit Fixed-Point Multiplication Instructions"
- Listing 5-3 on page 5-26 "16-Bit Fixed-Point Quad Multiplication Instructions"
- Listing 5-4 on page 5-26 "16-Bit Fixed-Point Complex Multiplication Instructions"
- Listing 5-5 on page 5-26 "32- and 40-Bit Floating-Point Multiplication Instructions"
- Listing 5-6 on page 5-26 "Multiplier Register Load Instructions"

The conventions used in these listings for representing register names, optional items, and choices are covered in detail in "Register File Registers" on page 2-5. Briefly, these conventions are:

- `{ }` – The curly braces enclose options; these braces are not part of the instruction syntax.
- `|` – The vertical bars separate choices; these bars are not part of the instruction syntax.
- *`Rmd`* – The register names in italic represent user selectable single (*`Rs`*, *`Rm`*, *`Rn`*), double (*`Rsd`*, *`Rmq`*, *`Rnd`*) or quad (*`Rsq`*) register names.

Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see "Instruction Line Syntax and Structure" on page 1-22 and "Instruction Parallelism Rules" on page 1-26.

The `MR3:0` registers are four 32-bit accumulation registers. They overflow into `MR4`, which stores two 16-bit overflows for 32-bit multiples, four 8-bit overflows for quad 16-bit multiples, or eight 4-bit overflows for quad-short 16-bit multiples.

Listing 5-2. 32-Bit Fixed-Point Multiplication Instructions

```
{X|Y|XY}Rs = Rm * Rn {({U|nU}{I}{T}{S})} ;[1]
{X|Y|XY}Rsd = Rm * Rn {({U|nU}{I})} ;
{X|Y|XY}MRa += Rm * Rn {({U}{I}{C|CR})} ;[2]
{X|Y|XY}MRa -= Rm * Rn {({I}{C|CR})} ;
{X|Y|XY}Rs = MRa, MRa += Rm * Rn {({U}{I}{C|CR})} ; dual op
```

[1] Options include: `( )`: fractional, signed, and no saturation; `(S)`: saturation, signed, `(SU)`: saturation, unsigned

[2] Options include: `( )`: signed, round-to-nearest even, `(T)`: signed, truncate, `(U)`: unsigned, round-to-nearest even, `(TU)`: unsigned, truncate

```
{X|Y|XY}Rsd = MRa, MRa += Rm * Rn {({U}{I}{C})} ; dual op
/* where MRa is either MR1:0 or MR3:2 */
```

Listing 5-3. 16-Bit Fixed-Point Quad Multiplication Instructions

```
{X|Y|XY}Rsd = Rmd * Rnd {({U}{I}{T}{S})} ;
{X|Y|XY}Rsq = Rmd * Rnd {({U}{I})} ;
{X|Y|XY}MRb += Rmd * Rnd {({U}{I}{C|CR})} ;
{X|Y|XY}Rsd = MRb, MRb += Rmd * Rnd {{I}{C|CR})} ; dual op
/* where MRb is either MR1:0, MR3:2, or MR3:0 */
```

Listing 5-4. 16-Bit Fixed-Point Complex Multiplication Instructions

```
{X|Y|XY}MRa += Rm ** Rn {({I}{C|CR}{J})} ;
{X|Y|XY}MRa -= Rm ** Rn {({I}{C|CR}{J})} ;
{X|Y|XY}Rs = MRa, MRa += Rm ** Rn {({I}{C|CR}{J})} ; dual op
{X|Y|XY}Rsd = MRa, MRa += Rm ** Rn {({I}{C|CR}{J})} ; dual op
/* where MRa is either MR1:0 or MR3:2 */
```

Listing 5-5. 32- and 40-Bit Floating-Point Multiplication Instructions

```
{X|Y|XY}FRs = Rm * Rn {(T)} ;
{X|Y|XY}FRsd = Rmd * Rnd {(T)} ;
```

Listing 5-6. Multiplier Register Load and Store Instructions

```
{X|Y|XY}{S|L}MRa = Rmd {(SE|ZE)} ;
{X|Y|XY}MR4 = Rm ;
{X|Y|XY}{S}Rsd = MRa {({U}{S})} ;
{X|Y|XY}Rsq = MR3:0 {({U}{S})} ;
{X|Y|XY}Rs = MR4 ;
/* where MRa is either MR1:0 or MR3:2 */
```

```
{X|Y|XY}Rsd = SMRb {(U)} ; /* extract 2 short words */
{X|Y|XY}LRsd = MRb {(U)} ; /* extract 1 normal word */
/* where MRb is either MR0, MR1, MR2, or MR3 */
{X|Y|XY}Rsq = SMRa {(U)} ; /* extract 4 short words */
{X|Y|XY}LRsq = MRa {(U)} ; /* extract 2 normal words */
{X|Y|XY}QRsq = LMRa {(U)} ; /* extract 1 long word from MRa */
/* where MRa is either MR1:0 or MR3:2 */
{X|Y|XY}Rs = COMPACT MRa {({U}{I}{S})} ;
{X|Y|XY}SRsd = COMPACT MR3:0 {({U}{I}{S})} ;
/* where MRa is either MR1:0 or MR3:2 */
```

6 SHIFTER

The ADSP-TS201 TigerSHARC processor core contains two computation units known as compute blocks. Each compute block contains a register file and four independent computation units—an ALU, a CLU, a multiplier, and a shifter. The shifter is highlighted in Figure 6-1. The shifter takes its inputs from the register file, and returns its outputs to the register file.

This chapter provides:

- "Shifter Operations" on page 6-3
- "Shifter Examples" on page 6-18
- "Shifter Instruction Summary" on page 6-19

The shifter performs *bit wise operations* (arithmetic and logical shifts) and performs *bit field operations* (field extraction and deposition) for the processor. The shifter also executes *data conversion operations* such as fixed- and floating-point format conversions. Shifter operations include:

- Shift and rotate bit field, from off-scale left to off-scale right
- Bit manipulation; bit set, clear, toggle, and test
- Bit field manipulation; field extract and deposit
- Scaling factor identification, 16-bit block floating-point
- Extract exponent
- Count number of leading ones or zeros

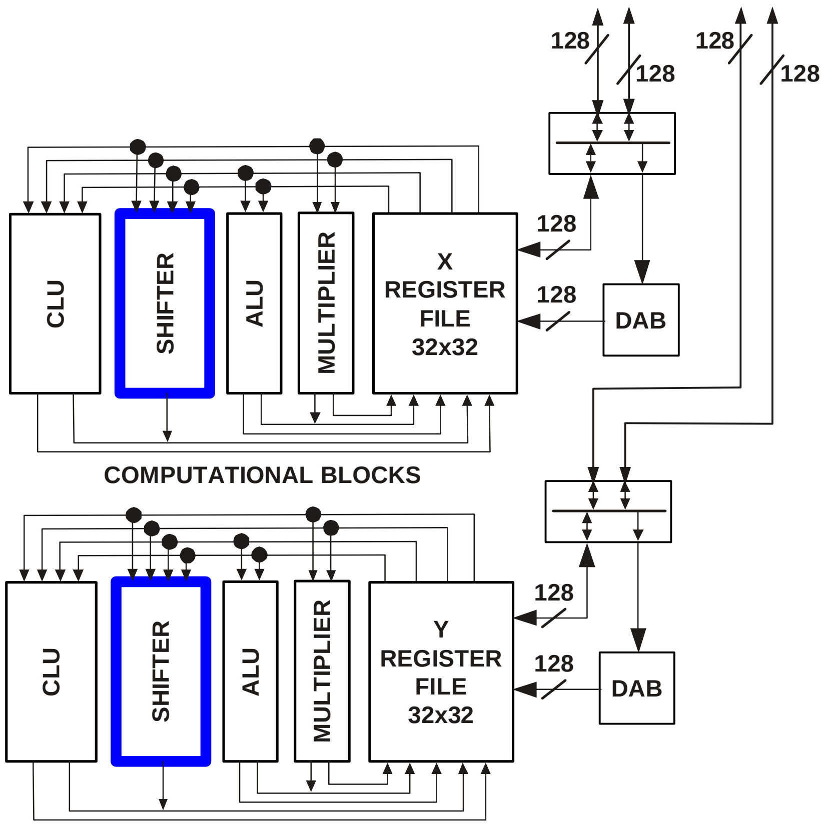


Figure 6-1. Shifters in Compute Block X and Y

The shifter operates on fixed-point data and can take the following as input:

- One long word (64-bit) operand
- One or two normal word (32-bit) operands
- Two or four short word (16-bit) operands
- Four or eight byte word (8-bit) operands

As shown in Figure 6-1, the shifter has four inputs and four outputs (unlike the ALU and multiplier, which have two inputs and outputs). The shifter's I/O paths within the compute block have some implications for instruction parallelism.

- Shifter instructions that use three inputs cannot be executed in parallel with any other compute block operations.
- For `FDEP`, `MASK`, `GETBITS` and `PUTBITS` instructions, there are three registers that are passed into the shifter. This operation uses three compute block ports. The output is being placed in the same port.

For more information on available ports and instruction parallelism, see "Instruction Parallelism Rules" on page 1-26.

Within instructions, the register name syntax identifies the input operand and output result data size and type. For more information on data size and type selection for shifter instructions, see "Register File Registers" on page 2-5.

The remainder of this chapter presents descriptions of shifter instructions and results using instruction syntax. For an explanation of the instruction syntax conventions used in shifter and other instructions, see "Instruction Line Syntax and Structure" on page 1-22. For a list of shifter instructions and their syntax, see "Shifter Instruction Summary" on page 6-19.

Shifter Operations

The shifter operates on one 64-bit, one or two 32-bit, two or four 16-bit, and four or eight 8-bit fixed-point operands. Shifter operations include:

- Shifts and rotates from off-scale left to off-scale right
- Bit manipulation operations, including bit set, clear, toggle and test

- Bit field manipulation operations, including field extract and deposit, using register `BFOTMP` (which is internal to the shifter)
- Bit FIFO operations to support bit streams with fields of varying length
- Support for ADSP-2100 family compatible fixed-point and floating-point conversion operations (such as exponent extract, number of leading 1s or 0s)

The shifter operates on the compute block register files and operates on the shifter register `BFOTMP`—an internal shifter register which is used for the `PUTBITS` instruction. Shifter operations can take their *`Rm`* input (data operated on) from the register file and take their *`Rn`* input (shift magnitudes) either from the register file or from immediate data provided in the instruction. In cases where the operation involves a third input operand, *`Rm`* and *`Rn`* inputs are taken from the register file, and the third input, *`Rs`*, is a read-modify-write (RMW).

Shift magnitudes for register file-based operations—where the shift magnitude comes from *`Rn`*—are held in the right-most bits of *`Rn`*. The shift magnitude size (number of bits) varies with the size of the output operand, where *`Rn`* is 8 bits for long word output, 7 bits for normal word output, 6 bits for short word output and 6 bits for byte word output. In this way, full-scale right and left shifts can be achieved. Bits of *`Rn`* outside of the shift magnitude field are masked.

The following sections describe shifter operation details:

- "Logical Shift Operation" on page 6-5
- "Arithmetic Shift Operation" on page 6-6
- "Bit Manipulation Operations" on page 6-7
- "Bit Field Manipulation Operations" on page 6-8

- "Bit Field Conversion Operations" on page 6-11
- "Bit Stream Manipulation Operations" on page 6-11

Logical Shift Operation

The following instruction is an example of a logical shift (`LSHIFT`). The operation shifts the contents of the `XR4` register by the shift value (number of bits) specified in the `XR3` register. The shifter places the result in the `XR5` register. Figure 6-2 shows how the bits in register `XR5` are placed for shift values of 4 and –4.

```
XR5 = LSHIFT R4 BY R3;;
```

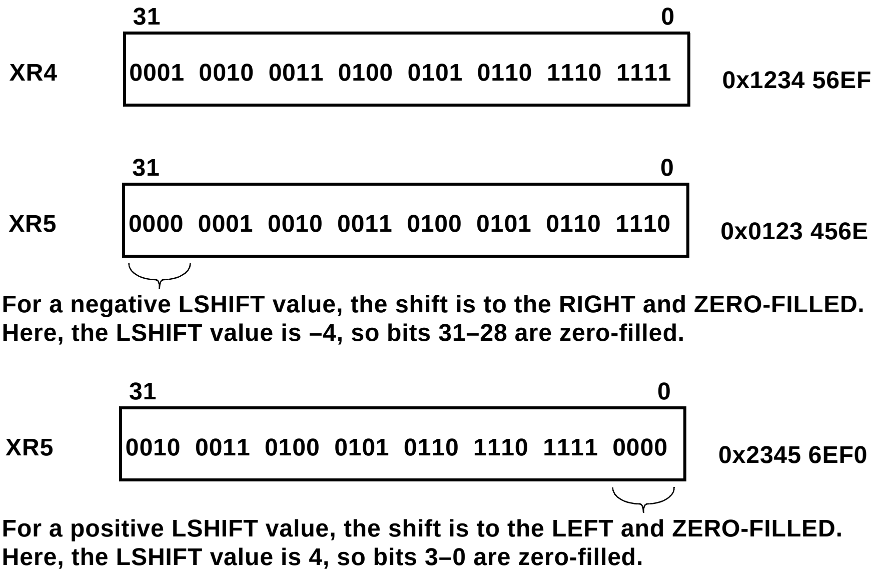

31
0
XR4
0001 0010 0011 0100 0101 0110 1110 1111
0x1234 56EF
31
0
XR5
0000 0001 0010 0011 0100 0101 0110 1110
0x0123 456E
For a negative LSHIFT value, the shift is to the RIGHT and ZERO-FILLED.
Here, the LSHIFT value is –4, so bits 31–28 are zero-filled.
31
0
XR5
0010 0011 0100 0101 0110 1110 1111 0000
0x2345 6EF0
For a positive LSHIFT value, the shift is to the LEFT and ZERO-FILLED.
Here, the LSHIFT value is 4, so bits 3–0 are zero-filled.

Figure 6-2. LSHIFT Instruction Example

Arithmetic Shift Operation

The following instruction is an example of an arithmetic shift (`ASHIFT`). The operation shifts the contents of the `XR4` register by the shift value (number of bits) specified in the `XR3` register. The shifter places the result in the `XR5` register. Figure 6-3 shows how the bits in register `XR5` are placed for shift values of 8 and –8.

```
XR5 = ASHIFT R4 BY R3 ;;
```

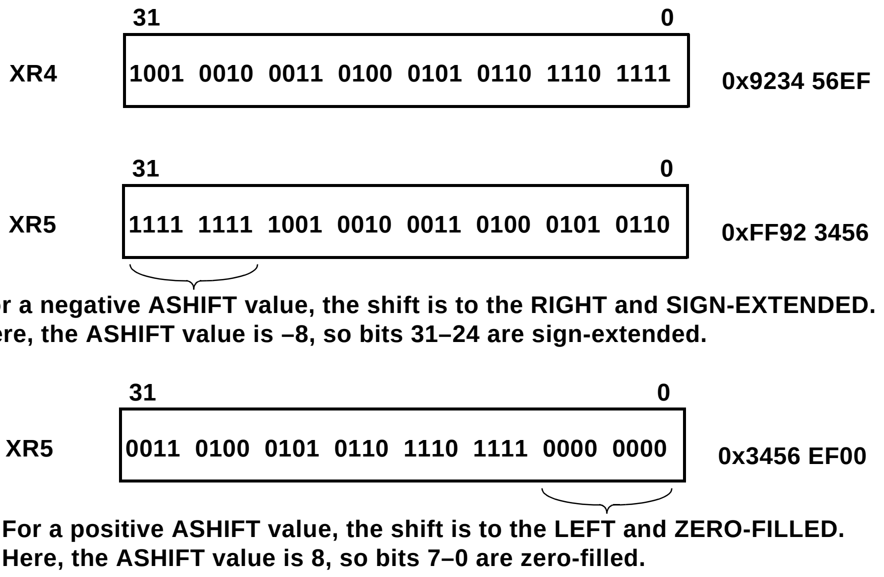

31
0
XR4
1001 0010 0011 0100 0101 0110 1110 1111
0x9234 56EF
31
0
XR5
1111 1111 1001 0010 0011 0100 0101 0110
0xFF92 3456
For a negative ASHIFT value, the shift is to the RIGHT and SIGN-EXTENDED.
Here, the ASHIFT value is –8, so bits 31–24 are sign-extended.
31
0
XR5
0011 0100 0101 0110 1110 1111 0000 0000
0x3456 EF00
For a positive ASHIFT value, the shift is to the LEFT and ZERO-FILLED.
Here, the ASHIFT value is 8, so bits 7–0 are zero-filled.

Figure 6-3. ASHIFT Instruction Example

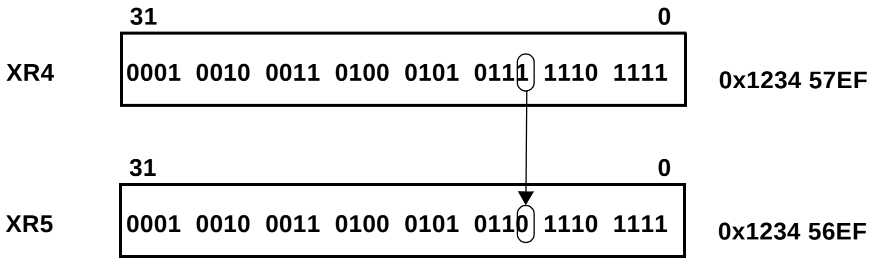
Bit Manipulation Operations

The shifter supports bit manipulation operations including bit clear (`BCLR`), bit set (`BSET`), bit toggle (`BTGL`), and bit test (`BITEST`). The operand size can be a normal word or a long word. For example:

```
R5 = BCLR  R3  By R2 ;; /* 32-bit operand */
R5:4 = BSET  R3:2  By R6 ;; /* 64-bit operand */
```

The following instruction is an example of bit manipulation (`BCLR`). The shifter clears the bit in the `XR4` register indicated by the bit number specified in the `XR3` register. The shifter places the result in the `XR5` register. Figure 6-4 shows how the bits in register `XR5` are affected for bit number 8.

```
XR5 = BCLR  R4  By R3 ;;
```



For a BCLR bit manipulation, the selected bit is CLEARED.
Because XR3=0x8 (the bit number), bit 8 is cleared.

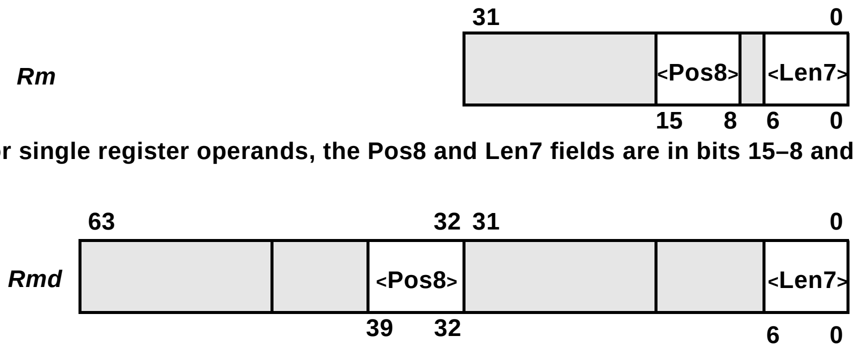
Figure 6-4. BCLR Instruction Example

Bit Field Manipulation Operations

The shifter supports bit field manipulation operations including:

- `FEXT`–Extracts a field from a register according to the length and position specified by another register
- `FDEP`–Deposits a right-justified field into a register according to the length and position specified by another register
- `MASK`–Copies a 32- or 64-bit field created by a mask
- `XSTAT`/`YSTAT`–Loads or stores all bits or 14 LSBs only of the `XSTAT` or `YSTAT` register

For field extract and deposit operations, the *Rn* operand contains the control information in two fields: <Len7> and <Pos8>. These fields select the number of bits to extract (Len7) and the starting position in *Rm* (Pos8). The location of these fields depends on whether *Rn* is a single- or dual-register as shown in Figure 6-5.



For single register operands, the Pos8 and Len7 fields are in bits 15–8 and 6–0.

For dual register operands, the Pos8 and Len7 fields are in bits 39–32 and 6–0.

Figure 6-5. FEXT and FDEP Instructions Pos8 and Len7 Fields

There are two versions of the FEXT and FDEP is instructions. One version takes the control information from a register pair. The other version takes control information from a single register. The FEXT instruction takes the data from the indicated position in the source register and places right-justified data in the destination register (*Rs*). The FDEP instruction takes the right-justified data from the source register and places data in the indicated position in the destination register (*Rs*).

The following instruction is an example of bit field extraction (FEXT). The shifter extracts the bit field in the XR4 register indicated by the field position (Pos8) and field length (Len7) values specified in the XR3 register. The shifter places the right-justified result in the XR5 register. The default operation zero-fills the unused bits in the destination register (XR5 in the example). If the FEXT instruction included the sign extend (SE) option, the most significant bit of the extracted field is extended. Figure 6-6 shows how the bits in register XR5 are affected for field position Pos8=5 and field length Len7=8.

```
XR5 = FEXT  R4  By R3 ;; /* Pos8=5, Len7=8, XR3=0x0000 0508 */
```

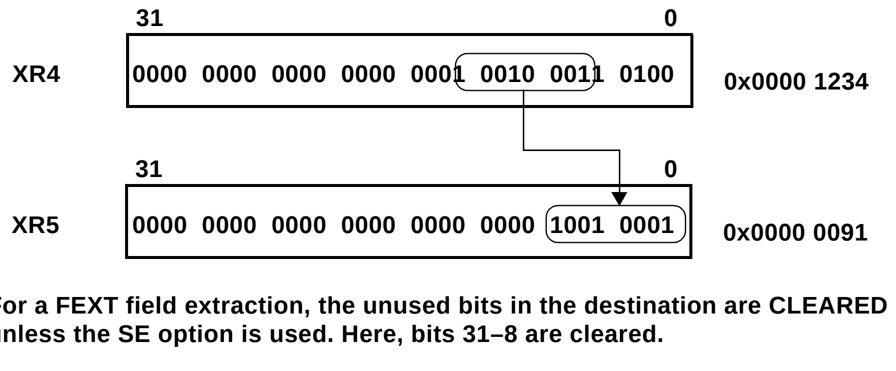

31
0
XR4
0000 0000 0000 0000 0001 0010 0011 0100
0x0000 1234
31
0
XR5
0000 0000 0000 0000 0000 0000 1001 0001
0x0000 0091
For a FEXT field extraction, the unused bits in the destination are CLEARED unless the SE option is used. Here, bits 31–8 are cleared.

Figure 6-6. FEXT Instruction Example

The following instruction is an example of bit field deposit (FDEP). The shifter extracts the right-justified bit field in the XR4 register field length (Len7) value specified in the XR3 register. The shifter places the result in the XR5 register in the location indicated by the field position (Pos8). The default operation does not alter the unused bits in the destination register (XR5 in the example). If the FDEP instruction included the sign extend (SE) option, the most significant bit of the extracted field is extended. If the FDEP instruction included the sign extend (ZF) option, the most significant unused bits of result register are zero filled. Figure 6-7 shows how the bits in register XR5 are affected for field position Pos8=5 and field length Len7=8.

```
XR5 = FDEP  R4  By R3 ;; /* Pos8=5, Len7=8, XR3=0x0000 0508, XR5
value before instruction was 0xA5A5 A5A5 */
```

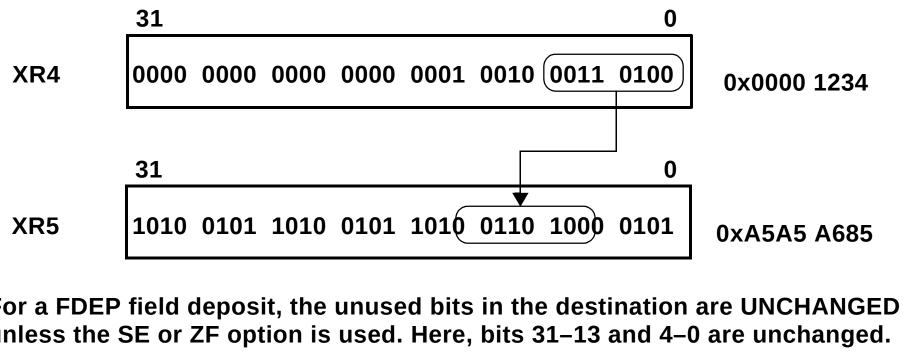

31
0
XR4
0000 0000 0000 0000 0001 0010 0011 0100
0x0000 1234
31
0
XR5
1010 0101 1010 0101 1010 0110 1000 0101
0xA5A5 A685
For a FDEP field deposit, the unused bits in the destination are UNCHANGED unless the SE or ZF option is used. Here, bits 31–13 and 4–0 are unchanged.

Figure 6-7. FDEP Instruction Example

The following instruction is an example of mask (MASK) operation. The shifter takes the bits from XR4 corresponding to the mask XR3 and ORs them into the XR5 register. The bits of XR5 outside the mask remain untouched.

```
XR3 = 0x00007B00;;
XR4 = 0x50325032;;
```

```
XR5 = 0x85FFFFFF;; /* before mask instruction */
XR5 += MASK R4 BY R3
/* After mask instruction, XR5 = 0x85FF50FF */
```

Bit Field Conversion Operations

The shifter supports fixed- to floating-point conversion operations including:

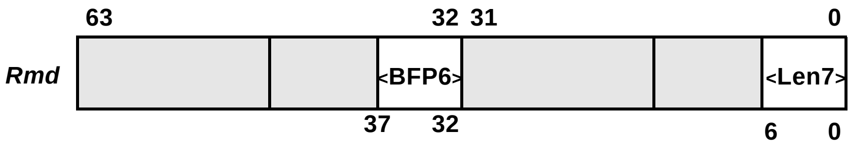
- `BKFPT`–Determines scaling factor used in 16-bit block floating-point
- `EXP`–Extracts the exponent
- `LDx`–Extracts leading zeros (0) or ones (1)

Bit Stream Manipulation Operations

The bit stream manipulation operations, in conjunction with the ALU `BFOINC` instruction, implement a bit FIFO used for modifying the bits in a contiguous bit stream. The shifter supports bit stream manipulation operations including:

- `GETBITS`–Extracts bits from a bit stream
- `PUTBITS`–Deposits bits in a bit stream
- `BFOTMP`–Temporarily stores or returns overflow from `GETBITS` and `PUTBITS` instructions

For bit stream extract (GETBITS) and deposit (PUTBITS) operations, the *Rnd* operand contains the control information in two fields: <BFP6> and <Len7>. These fields in the dual register, *Rnd*, appear in Figure 6-8.



The dual register operand provides the BFP6 and Len6 fields (bits 37–32 and 5–0). Note that the BFP must be incremented using the ALU's BFOINC instruction.

Figure 6-8. GETBITS and PUTBITS Instructions BFP6 and Len7 Fields

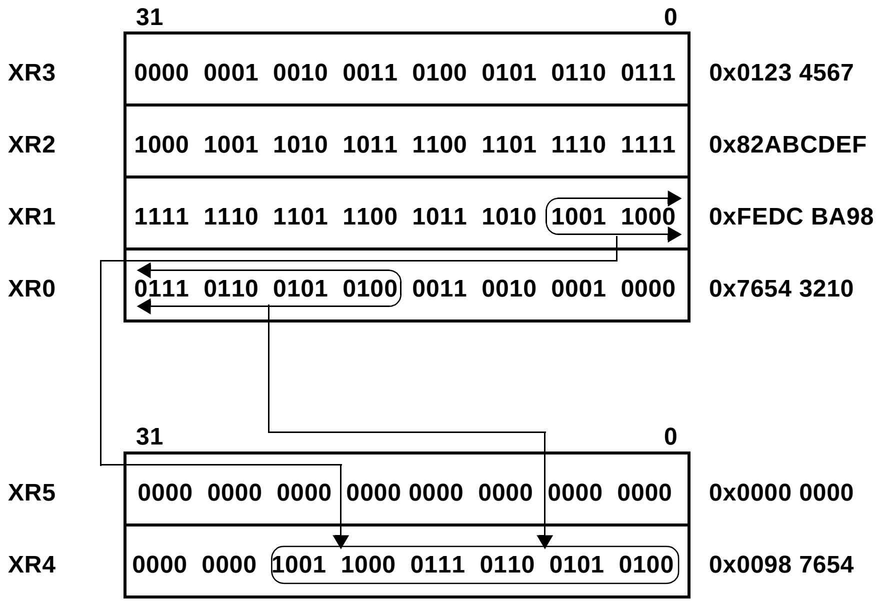
The GETBITS instruction extracts the number of bits indicated by Len7 starting at BFP6 and places the right-justified data in the output register. The unused bits are cleared unless the sign extend (SE) option is used. With the SE option, the most significant bit of the extract is extended to the most significant bit of the output register.

The following instruction is an example of bit stream extraction (GETBITS). The shifter extracts a portion of the bit stream in the XR3:0 quad register indicated by the bit FIFO position (BFP6) and field length (Len7) values specified in the XR7:6 dual register.

Use the ALU's BFOINC instruction to increment the bit FIFO pointer. Normally, an update of bit FIFO pointer is necessary after executing GETBITS. The ALU instruction BFOINC adds BFP6 and Len7 fields, divides them by 64 and returns the remainder to BFP6 field. If for example, BFP6 is 0x30 and Len7 is 0x18, the new value of BFP6 is 0x08 and the flag AN in XSTAT register is set. This flag may be used to identify this situation and proceed accordingly.

In the example, the shifter places the right-justified result in the XR5:4 dual register. The default operation zero-fills the unused bits in the destination register (XR5:4 in the example). If the GETBITS instruction included the sign extend (SE) option, the most significant bit of the extracted field is extended. Figure 6-9 shows how the bits in register XR5:4 are affected for field position BFP6=16 and field length Len7=24.

```
XR5:4 = GETBITS R3:0 BY R7:6 ;;
/* BFP6=16, Len7=24, XR7:6=0x0000 0010 0000 0018 */
```



For a GETBITS field extraction, the unused bits in the destination are CLEARED unless the SE option is used. Here, bits XR5 and bits 31–24 of XR4 are cleared.

Figure 6-9. GETBITS Instruction Example

The PUTBITS instruction deposits the 64 bits from *Rmd* registers into a contiguous bit stream held in the quad register composed of BFOTMP in the top and *Rsd* in the bottom. In PUTBITS, the BFP field specifies the starting bit where the insertion begins in *Rsd* register, but the Len7 field is ignored. Update of BFP may only be performed by the ALU with the instruction BFOINC.

The following instruction is an example of bit stream placement (PUT-BITS). The shifter puts the content of the registers XR3:2 into the bit FIFO composed by XR5:4 and BFOTMP beginning with bit 16 of XR4 (specified into BFP field of XR7).

```
XR3 = 0x01234567 ;;
XR2 = 0x89abcdef ;;
XR5 = 0x0 ;
XR4 = 0x0 ;
XR5:4 += PUTBITS R3:2 BY R7:6 ;; /* BFP6=16, XR7:6=0x0000 0010
0000 0018 */
/* After PUTBITS instruction, the registers hold:
   xBFOTMP = 0x0000 0000 0000 0123
   XR5       = 0x4567 89ab
   XR4       = 0xcdef 0000
```

Shifter Instruction Options

Some of the shifter instructions have options associated with them that permit flexibility in how the instructions execute. It is important to note that these options modify the detailed execution of instructions, and the options that are particular to a group of instructions—not all options are applicable to all instructions. Instruction options appear in parenthesis at the end of the instruction's slot.

For a list indicating which options apply for particular shifter instructions, see "Shifter Instruction Summary" on page 6-19. The shifter instruction options include:

- () zero filled, right justified
- (SE) sign extended; applies to FEXT, FDEP, and GETBITS instructions
- (ZF) zero filled; applies to FDEP instruction

The following are shifter instructions that demonstrate bit field manipulation operations with options applied.

```
XR5 = FEXT  R4  By R3 (SE) ;;
```
/ The SE option in this instruction sets bits 31–8 to 1 in Figure 6-6 on page 6-9* */

```
XR5 = FDEP  R4  By R3 (ZF) ;;
```
/ The ZF option in this instruction clears bits 31–13 and 4–0 in Figure 6-7 on page 6-10* */

Sign Extended Option

The sign extend (SE) option is available for the FEXT, FDEP, and GETBITS shifter instructions. If used, this option extends the value of the most significant bit of the placed bit field through the most significant bit of the output register.

Zero Filled Option

The zero filled (ZF) option is available only for the FDEP instruction. If used, this option clears all the unused bits above the most significant bit of the placed bit field in the output register.

Shifter Execution Status

Shifter operations update status flags in the compute block's Arithmetic Status (XSTAT and YSTAT) register (see Figure 2-2 on page 2-4 and Figure 2-3 on page 2-5). Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see "Shifter Execution Conditions" on page 6-16.

Table 6-1 shows the flags in `XSTAT` or `YSTAT` that indicate shifter status (a 1 indicates the condition) for the most recent shifter operation.

Table 6-1. Shifter Status Flags

Flag	Definition	Updated By…
SZ	Shifter fixed-point zero	All shifter ops
SN	Shifter negative	All shifter ops
BF1–0	Shifter block floating-point	BKFPT instruction only

Flag update occurs at the end of each operation and is available on the next instruction slot. A program cannot write the arithmetic status register explicitly in the same cycle that the ALU, CLU or multiplier are performing an operation.

Multi-operand instructions (for example, `B`*`Rs`* `= ASHIFT` *`Rn`* `BY` *`Rm`*;) produce multiple sets of results. In this case, the processor determines a flag by ORing the result flag values from individual results.

Shifter Execution Conditions

In a conditional shifter instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional shifter instructions take the form:

```
IF cond; DO, instr.; DO, instr.; DO, instruct. ;;
```

This syntax permits up to three instructions to be controlled by a condition. Omitting the `DO` before the instruction makes the instruction unconditional.

Table 6-2 lists the shifter conditions. For more information on conditional instructions, see "Conditional Execution" on page 8-12.

Table 6-2. Shifter Conditions

Condition	Description	Flags set
SEQ	Equal to zero	SZ = 1
SLT	Less than zero	SN = 1 and SZ = 0
NSEQ	Not equal to zero	SZ = 0
NSLT	Not less than zero	SN = 0 and SZ = 1

Shifter Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later usage in conditional instructions. With SFREG, each compute block has two dedicated static flags X/YSCF0 (condition is SF0) and X/YSCF1 (condition is SF1). The following example shows how to load a compute block condition value into a static flag register.

```
XSCF0 = XSEQ ;; /* Load X-compute block SEQ flag into XSCF0 bit
in static flags (SFREG) register */
IF SF0, XR5 = LSHIFT R4 BY R3 ;; /* the SF0 condition tests the
XSCF0 static flag */
```

For more information on static flags, see "Conditional Execution" on page 8-12.

Shifter Examples

Listing 6-1 provides a number of shifter instruction examples. The comments with the instructions identify the key features of the instruction, such as input operand size and register usage.

Listing 6-1. Shifter Instruction Examples

```
XR5 = LSHIFT R4 BY R3;;
/* This is a logical shift of register XR4 by the value contained
in XR3. */

YR1 = ASHIFT R2 BY R0;;
/* This is an arithmetic shift of register XR2 by the value con-
tained in XR0. */

R1:0 = ROT R3:2 BY 8;;
/* This instruction rotates the content of R3:2 in both the X-
and Y-ALUs by 8 and places the result in XR1:0 and YR1:0. */

XBITEST R1:0 BY R7;;
/*  This instruction tests the bit indicated in XR7 of XR1:0 and
sets accordingly the flags XSZ and XSN in XSTAT. */

R9:8 = BTGL R11:10 BY R13;;
/* This instruction toggles the bit indicated in R13 of XR11:10
and YR11:10 and puts the result in xR9:8 and yR9:8. */

XR15 = LDO R17;;
/* This instruction extracts the leading number of zeros of xR17
and places the result into xR15. */
```

Shifter Instruction Summary

Listing 6-2 shows the shifter instructions' syntax. The conventions used in these listings for representing register names, optional items, and choices are covered in detail in "Register File Registers" on page 2-5. Briefly, these conventions are:

- `{ }` – the curly braces enclose options; these braces are not part of the instruction syntax.
- `|` – the vertical bar separates choices; this bar is not part of the instruction syntax.
- *Rmd* – the register names in italic represent user selectable single (*Rs*, *Rm*, *Rn*), double (*Rsd*, *Rmd*, *Rnd*) or quad (*Rsq*, *Rmq*, *Rnq*) register names.

In shifter instructions, output register name relates to output operand size as follows:

- The L prefix on an output operand (register name) indicates long word (64-bit) output. For example, the following instruction syntax indicates single, long word output:

  ```
  LRsd = ASHIFT Rmd BY Rnd;
  ```

- The absence of a prefix on an output operand (register name) indicates normal word (32-bit) output. For example, the following instruction syntax indicates single, normal word output:

  ```
  Rs = ASHIFT Rm BY Rn;
  ```

- A dual register name on an output operand (register name) indicates two normal word (32-bit) outputs. For example, the following instruction syntax indicates two, normal word outputs:

  ```
  Rsd = ASHIFT Rmd BY Rnd;
  ```

- The S prefix on an output operand (register name) indicates two or four short word (16-bit) outputs. For example, the following instruction syntax indicates two or four, short word outputs:

```
SRs = ASHIFT Rm BY Rn /* two outputs */;
SRsd = ASHIFT Rmd BY Rnd; /* four outputs */
```

- The B prefix on an output operand (register name) indicates four or eight byte word (8-bit) outputs. For example, the following instruction syntax indicates four or eight, byte word outputs:

```
BRs = ASHIFT Rm BY Rn /* four outputs */;
BRsd = ASHIFT Rmd BY Rnd; /* eight outputs */
```

Each instruction presented here occupies one instruction slot in an instruction line. For more information about instruction lines and instruction combination constraints, see "Instruction Line Syntax and Structure" on page 1-22 and "Instruction Parallelism Rules" on page 1-26.

Listing 6-2. Shifter Instructions

```
{X|Y|XY}{B|S}Rs = LSHIFT|ASHIFT Rm BY Rn|<Imm> ;[1,2]
{X|Y|XY}{B|S|L}Rsd = LSHIFT|ASHIFT Rmd BY Rn|<Imm> ;[1,2]

{X|Y|XY}Rs = ROT Rm BY Rn|<Imm6> ;[1]
{X|Y|XY}{L}Rsd = ROT Rmd BY Rnd|<Imm> ;[1,2]

{X|Y|XY}Rs = FEXT Rm BY Rn|Rnd {(SE)} ;[3]
{X|Y|XY}LRsd = FEXT Rmd BY Rn|Rnd {(SE)} ;[3]

{X|Y|XY}Rs += FDEP Rm BY Rn|Rnd {(SE|ZF)} ;[3]
{X|Y|XY}LRsd += FDEP Rmd BY Rn|Rnd {(SE|ZF)} ;[3]

{X|Y|XY}Rs += MASK Rm BY Rn ;
{X|Y|XY}LRsd += MASK Rmd BY Rnd ;

{X|Y|XY}Rsd = GETBITS Rmq BY Rnd {(SE)} ;

{X|Y|XY}Rsd += PUTBITS Rmd BY Rnd ;

{X|Y|XY}BITEST Rm BY Rn|<Imm5> ;
{X|Y|XY}BITEST Rmd BY Rnd|<Imm6> ;

{X|Y|XY}Rs = BCLR|BSET|BTGL Rm BY Rn|<Imm5> ;
{X|Y|XY}Rsd = BCLR|BSET|BTGL Rmd BY Rn|<Imm6> ;

{X|Y|XY}Rs = LD0|LD1 Rm|Rmd ;

{X|Y|XY}Rs = EXP Rm|Rmd ;

{X|Y}STAT = Rm ;
```

[1] The `Rn` data size (bits) for the shift magnitude varies with the output operand: Byte: 5, Short: 6, Normal: 7, Long: 8.

[2] The size in bits of the `Imm` data varies with the output operand: Byte: 4, Short: 5, Normal: 6, Long: 7.

[3] The placement of the Pos8 and Len7 fields varies with the `Rn/Rnd` register, see Figure 6-5 on page 6-8.

```
{X|Y}STATL = Rm ;
{X|Y}Rs = {X|Y}STAT ;

{X|Y|XY}BKFPT Rmd, Rnd ;

{X|Y|XY}Rsd = BFOTMP ;
{X|Y|XY}BFOTMP = Rmd ;
```