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

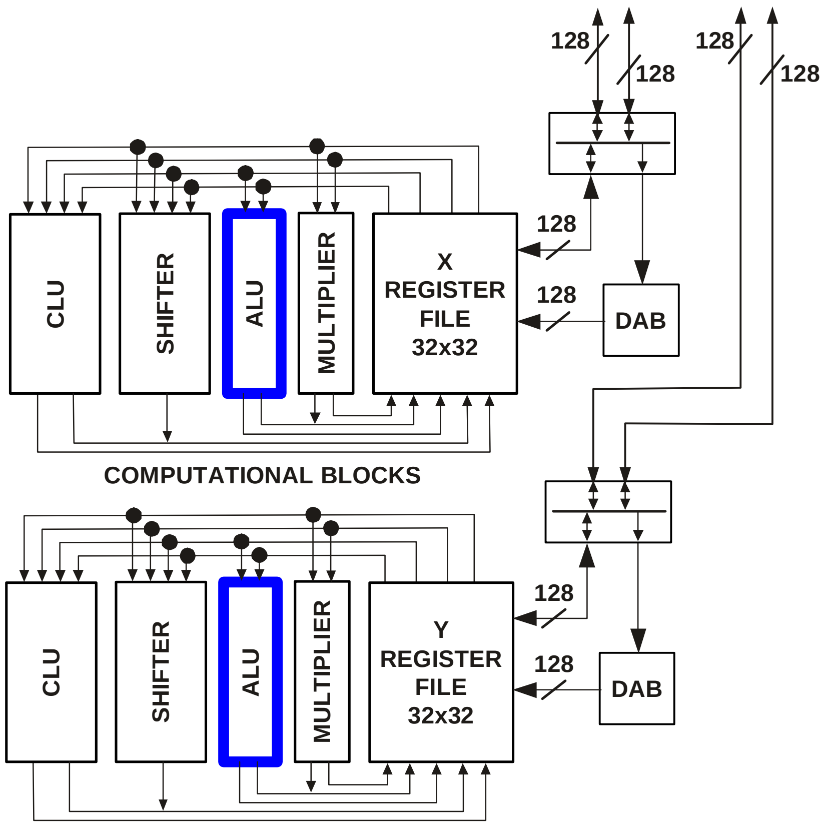

Figure 3-1. ALUs in Compute Block X and Y

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.

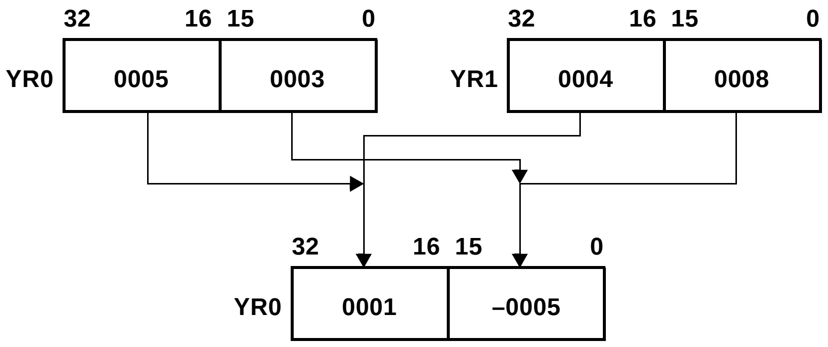

Figure 3-2. Input Operands for Parallel Subtract

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, `BRs = 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 ;;
```

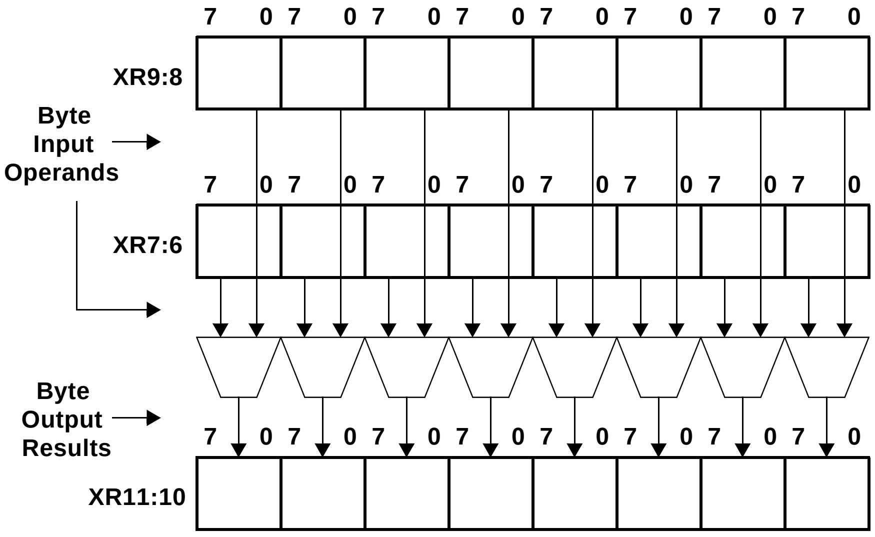

Figure 3-3. Input Operands for Parallel Add

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 */
```

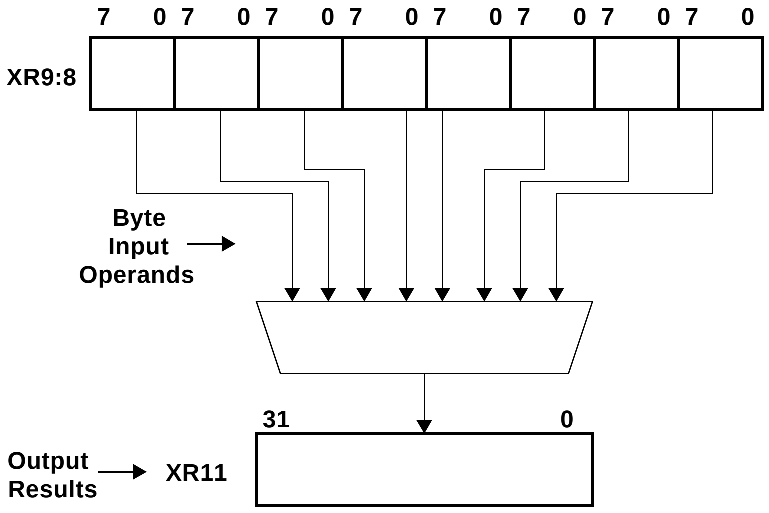

Figure 3-4. Input Operands for Sideways Add

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.

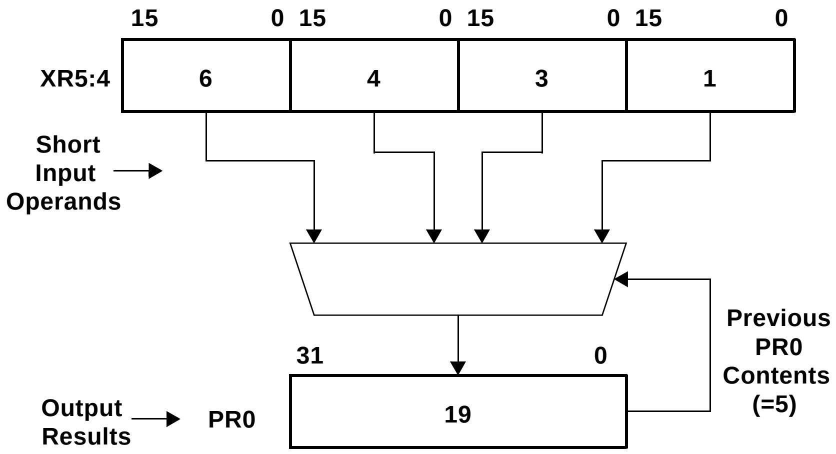

Figure 3-5. Input Operands for Parallel/Sideways Add

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