7 IALU

The ADSP-TS201 TigerSHARC processor core contains two integer arithmetic logic units known as IALUs. Each IALU contains a register file and dedicated registers for circular buffer addressing. The IALUs can control the Data Alignment Buffers (DABs) for unaligned memory access operations. The IALUs and DABs are highlighted in Figure 7-1.

The internal bus structure on the ADSP-TS201 TigerSHARC processor differs from previous TigerSHARC processors. One important difference is that the IALUs are the internal bus masters that execute load, store, and move register instructions. For more information on the internal bus structure and operations, see "Memory Bus Arbitration" on page 9-33.

The ADSP-TS201 processor's two independent IALUs are referred to as the J-IALU and K-IALU. The IALUs support regular ALU operations and data addressing operations. The *arithmetic, logical, and function ALU operations* include:

- Add and subtract, with and without carry/borrow
- Arithmetic right shift, logical right shift, and rotation
- Logical operations: `AND`, `AND NOT`, `NOT`, `OR`, and `XOR`
- Functions: absolute value, min, max, compare

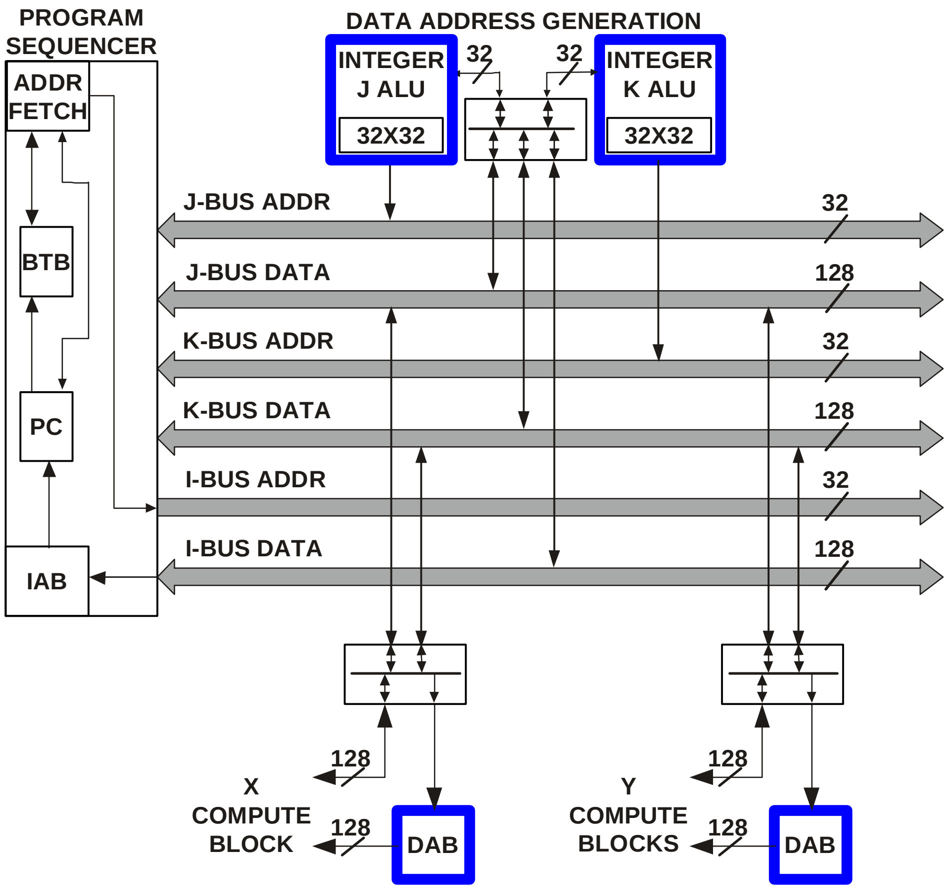


Figure 7-1. IALUs, DABs, and Data Buses

The IALUs provide memory addresses when data is transferred between memory and registers. Dual IALUs enable simultaneous addresses for multiple operand reads or writes. The IALU's *data addressing and data movement operations* include:

- Direct and indirect memory addressing
- Circular buffer addressing

- Bit reverse addressing
- Universal register (`Ureg`) moves and loads
- Memory pointer generation

Each move instruction specifies whether a normal, long, or quad word is accessed from each memory block. Because the processor has two IALUs, two memory blocks can be accessed on each cycle. Long word accesses can be used to supply two aligned normal words to one compute block or one aligned normal word to each compute block. Quad word accesses may be used to supply four aligned normal words to one compute block or two aligned normal words to each compute block. This is useful in applications that use real/imaginary data, or parallel data sets that can be aligned in memory—as are typically found in DSP applications. It is also used for fast save/restore of context during C calls or interrupts.

The IALU provides flexibility in moving data as single, dual, or quad words. Every instruction can execute with a throughput of one per cycle. IALU arithmetic instructions execute with a single cycle of latency while computation units have two cycles of latency. Normally, there are no dependency delays between IALU instructions, but if there are, three or four cycles of latency can occur. For more information, see "Dependency and Resource Effects on Pipeline" on page 8-59.

The IALUs each contain a 32-register data register file and eight dedicated registers for circular buffer addressing. All registers in the IALU are 32-bit wide, memory-mapped, universal registers. Some important points concerning the IALU register files are:

- The J-IALU register file registers are `J30`–`J0`, and the K–IALU register file registers are `K30`–`K0`. Except for `J31` and `K31`, these registers are general purpose and contain integer data only.
- The `J31` and `K31` registers are 32-bit status registers and can also be referred to as `JSTAT` and `KSTAT`.

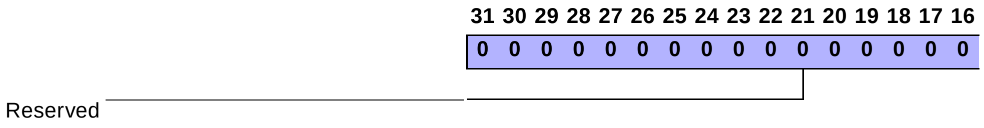


Figure 7-2. JSTAT/KSTAT (Upper Half) Register Bit Descriptions

The JSTAT (J31) and KSTAT (K31) registers appear in Figure 7-2 and Figure 7-3. These registers have special operations:

- When used as an operand in an IALU arithmetic, logical, or function operation, these registers are referred to as J31 and K31 registers, and the register's contents are treated as 0. If J31 is used as an output of an operation, it does not retain the result of the instruction, but the flags are set.

- When used for an IALU load, store, or move operation, these registers are referred to as JSTAT and KSTAT, and the operation does not clear the register contents.

For fast save and restore operations, load and store instructions can access J31:28 in quad-register format, saving or restoring J30:28 and JSTAT.

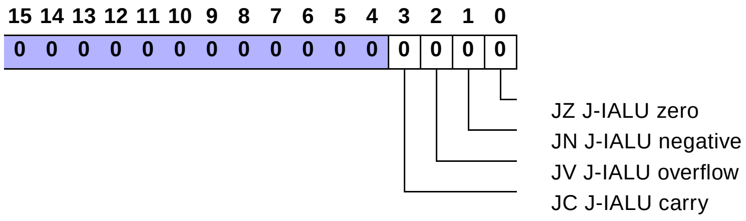


Figure 7-3. JSTAT/KSTAT (Lower Half) Register Bit Descriptions

The dedicated registers for circular buffer addressing in each IALU select the base address and buffer length for circular buffers. These dedicated registers work with the first four general-purpose registers in each IALU's register file to manage up to eight circular buffers. Some important points concerning the IALU dedicated registers for circular buffer addressing are:

- The circular buffer *index* (current address) is set by a general-purpose register. These are `J3`–`J0` in the J-IALU, and `K3`–`K0` in the K-IALU.
- The circular buffer *base* (starting address) is set by a dedicated register. These are `JB3`–`JB0` in the J-IALU, and `KB3`–`KB0` in the K-IALU.
- The circular buffer *length* (number of memory locations) is set by a dedicated register. These are `JL3`–`JL0` in the J-IALU, and `KL3`–`KL0` in the K-IALU.

- The circular buffer *modifier* (step size between memory locations) is set by either a general-purpose IALU register or an immediate value. The modifier may not be larger than the length of the circular buffer.
- The index, base, and length registers for controlling circular buffers work as a unit (`J0` with `JB0` and `JL0`, `J1` with `JB1` and `JL1`, and so on). Any IALU register file register (beside the one serving as index) in the IALU controlling the circular buffer may serve as the modifier.

The IALUs can use add and subtract instructions to generate memory pointers with or without circular buffer or bit reverse addressing. The modified address is stored in an IALU data register and (optionally) can be written to the program sequencer's computed jump (`CJMP`) register.

IALU Operations

The following sections describe the operation of each type of IALU instruction. These operation descriptions apply to both the J-IALU and K-IALU. The IALU operations are:

- "IALU Arithmetic and Logic Operations" on page 7-6
- "IALU Data Addressing and Transfer Operations" on page 7-14

IALU Arithmetic and Logic Operations

The IALU performs arithmetic and logical operations on fixed-point data. The DSP uses IALU register file registers for the input operands and output result from IALU operations. The IALU register file registers are `J30` through `J0` and `K30` through `K0`. The IALUs each have one special purpose register—the `JSTAT`/`KSTAT` register—for status. 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 are IALU instructions that demonstrate arithmetic operations.

```
J2 = J1 + J0 ;;
/* This is a fixed-point add of the 32-bit input operands J1 and
J0; the DSP places the result in J2. */

K0 = ABS K2 ;;
/* The DSP places the absolute value of fixed-point 32-bit input
operand K2 in the result register K0. */

J2 = ( J1 + J0 ) / 2 ;;
/* This is a fixed-point add and divide by 2 of the 32-bit input
operands J1+J0; the DSP places the result in J2. */
```

All IALU arithmetic, logical, and function instructions generate status flags to indicate the status of the result. If for example in the previous add/divide instruction the input was ((–2 +0) / 2), the operation would set the `JN` flag (J-IALU, IALU result negative) because operation resulted in a negative value. For more information on IALU status, see "IALU Execution Status" on page 7-11.

IALU Instruction Options

Most of the IALU 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 IALU instructions, see "IALU Instruction Summary" on page 7-42.

The IALU instruction options include:

- `()` signed operation, round-to-infinity, integer mode
- `(U)` unsigned operation
- `(CB)` circular buffer operation for result
- `(BR)` bit reverse operation for result
- `(CJMP)` load result into result and computed jump (`CJMP`) registers

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

```
J2 = J1 - J0 (CJMP);;
/* This is a fixed-point subtract of the 32-bit input operands;
the DSP loads the result into J2 and CJMP register. */

K1 = K3 + K4 (BR) ::
/* This is a fixed-point add of the 32-bit input operands with
bit reverse carry operation for the result. */

COMP(J1, J0) (U) ;;
/* This is a comparison of unsigned 32-bit input operands. */
```

IALU Data Types

In the IALU, all operations use 32-bit integer format data. For information on the supported numeric formats, see "Numeric Formats" on page 2-15.

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 32-bit data in the IALU may be two's-complement or (for the `COMP` instruction only) unsigned. For information on the supported numeric formats, see "Numeric Formats" on page 2-15.

Circular Buffer Option

The IALU add and subtract instructions support the circular buffer `(CB)` option. The instructions take the form:

```
Js = Jm +|- Jn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
Ks = Km +|- Kn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
```

For information on the conventions used in this instruction summary, see "IALU Instruction Summary" on page 7-42.

To use the `CB` option, the IALU add and subtract instructions require that the related J-IALU or K-IALU base and length registers are previously set up. The IALU add and subtract instructions with the `CB` option calculate the modified address from the index plus or minus the modifier and also performs circular buffer wrap (if needed) as part of the calculation. The IALU puts the modified value into *Js* or *Ks*. (*Jm* or *Km* is not modified.)

For information on circular buffer operations, see "Circular Buffer Addressing" on page 7-31.

Bit Reverse Option

The IALU add and subtract instructions support the bit reverse carry `(BR)` option. The instructions take the form:

```
Js = Jm +|- Jn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
Ks = Km +|- Kn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
```

For information on the conventions used in this instruction summary, see "IALU Instruction Summary" on page 7-42.

The IALU add and subtract instructions with the `BR` option use bit reverse carry to calculate the result of the index plus the modifier (there is no affect on the subtract operation because there is no carry) and put the modified value into *Js* or *Ks*. (*Jm* or *Km* is not modified.)

For information on bit reverse operations, see "Bit Reverse Addressing" on page 7-34.

Computed Jump Option

The IALU add and subtract instructions support the computed jump `(CJMP)` option. The instructions take the form:

```
Js = Jm +|- Jn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
Ks = Km +|- Kn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
```

For information on the conventions used in this instruction summary, see "IALU Instruction Summary" on page 7-42.

The computed jump (`CJMP`) option directs the IALU to place the result in the program sequencer's `CJMP` registers as well as the result (*Js* or *Ks*) register.

IALU Execution Status

IALU operations update status flags in the IALUs' status (JSTAT and KSTAT) registers. (See Figure 7-2 on page 7-4 and Figure 7-3 on page 7-5.) Programs can use status flags to control execution of conditional instructions and initiate software exception interrupts. For more information, see "IALU Execution Conditions" on page 7-13.

Table 7-1 shows the flags in JSTAT or KSTAT that indicate IALU status (a 1 indicates the condition) for the most recent IALU operation.

Table 7-1. IALU Status Flags

Flag	Definition	Updated By...
JZ	J-IALU zero	All J-IALU arithmetic, logical, and function ops
JN	J-IALU negative	All J-IALU arithmetic, logical, and function ops
JV	J-IALU overflow	All arithmetic ops and ABS op
JC	J-IALU carry	All arithmetic ops
KZ	K-IALU zero	All K-IALU arithmetic, logical, and function ops
KN	K-IALU negative	All K-IALU arithmetic, logical, and function ops
KV	K-IALU overflow	All arithmetic ops and ABS op
KC	K-IALU carry	All arithmetic ops

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

JN/KN–IALU Negative

The `JN` or `KN` flag is set whenever the result of a J-IALU or K-IALU operation is negative. The `JN` or `KN` flag is set to the most significant bit of the result. An exception is the instructions below, in which the `JN` or `KN` flag is set differently:

- `Js = ABS Jm ;` `JN` is `Jm` (input data) sign
- `Ks = ABS Km ;` `KN` is `Km` (input data) sign

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

JV/KV–IALU Overflow

The `JV` or `KV` flag is an overflow indication. In all J-IALU or K-IALU operations, the 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 `J5` and `J6` 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.

```
J10 = J5 + J6 ;;
```

JC/KC–IALU Carry

The `JC` or `KC` 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 (`JV` or `KV` is set when there is signed overflow). Bit reverse operations do not overflow and do not set the `JC` or `KC` flags.

IALU Execution Conditions

In a conditional IALU instruction, the execution of the entire instruction line can depend on the specified condition at the beginning of the instruction line. Conditional IALU 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 7-2 lists the IALU conditions. For more information on conditional instructions, see "Conditional Execution" on page 8-12.

Table 7-2. IALU Conditions

Condition	Description	Flags Set
JEQ	J-IALU equal to zero	JZ = 1
JLT	J-IALU less than zero	JN = 1
JLE	J-IALU less than or equal to zero	JN or JZ = 1
NJEQ	NOT(J-IALU equal to zero)	JZ = 0
NJLT	NOT(J-IALU less than zero	JN= 0
NJLE	NOT(J-IALU less than or equal to zero)	JN or JZ = 0
KEQ	K-IALU equal to zero	KZ = 1
KLT	K-IALU less than zero	KN = 1
KLE	K-IALU less than or equal to zero	KN or KZ = 1
NKEQ	NOT(K-IALU equal to zero)	KZ = 0
NKLT	NOT(K-IALU less than zero)	KZ = 0
NKLE	NOT(K-IALU less than or equal to zero)	KN or KZ = 0

IALU Static Flags

In the program sequencer, the static flag (SFREG) can store status flag values for later usage in conditional instructions. With SFREG, the IALU has two dedicated static flags GSCF0 (condition is SF0) and GSCF1 (condition is SF1). The following example shows how to load a compute block condition value into a static flag register.

```
GSCF0 = JEQ ;; /* Load J-IALU JEQ flag into GSCF0 bit in static
flags (SFREG) register */
IF SF0, J5 = J4 + J3 ;; /* the SF0 condition tests the GSCF0
static flag */
```

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

IALU Data Addressing and Transfer Operations

IALU data addressing instructions provide memory read and write access for loading and storing registers. For memory reads and writes, the IALU provides two types of addressing—direct addressing and indirect addressing. Both types of addressing use an index and a modifier.

The index is an address, and the modifier is a value added to the index either before (pre-modify) or after (post-modify) the addressing operation. IALU addressing instruction syntax uses square brackets ([ ]) to set the address calculation apart from the rest of the instruction.

Direct and Indirect Addressing

Direct addressing uses an index that is set to zero and uses an immediate value for the modifier. The index is pre-modified, and the modified address is used for the access. The following instruction is a register load (memory read) that uses direct addressing:

```
YR1 = [J31 + 0x00015F00] ;;
/* This instruction reads a 32-bit word from memory location
0x00015F00 and loads the word into register YR1. Note that J31
always contains zero when used as an operand. */
```

Indirect addressing uses a non-zero index and uses either a register or an immediate value for the modifier. As shown in Figure 7-4, there are two types of indirect addressing.

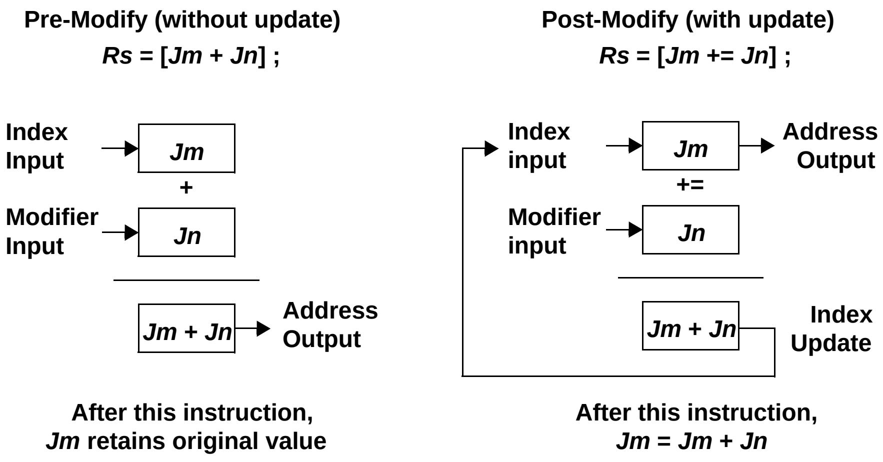


Figure 7-4. Pre- and Post-Modify Indirect Addressing

One type of indirect addressing is *pre-modify without update* (uses the `[ + ]` operator). These instructions provide memory access to the index + modifier address without changing the content of the index register. These instructions load the value into a destination register or store the value from a source register. For example,

```
XR0 = [J0 + J1] ;;
/* This instruction reads a 32-bit word from memory location
J0 + J1 (index + modifier) and loads the word into register XR0.
J0 (the index) is not updated with the address. */
```

The other type of indirect addressing is *post-modify with update* (uses the `[ += ]` operator). These instructions provide memory access to the indexed address. These instructions load a value into a destination register or store the value from a source register. After the access, the index register is updated by the modifier value. For example,

```
XR0 = [J0 += J1] ;;
/* This instruction reads a 32-bit word from memory location J0
(the index) and loads the word into register XR0. After the
access, J0 is updated with the value J0 + J1 (index + modifier).
*/
```

Post-modify indirect addressing is used with circular buffer and bit reversed addressing. For more information, see "Circular Buffer Addressing" on page 7-31 and "Bit Reverse Addressing" on page 7-34.

Normal, Merged, and Broadcast Memory Accesses

The IALU uses direct or indirect addressing to perform read and write memory accesses, loading or storing data in registers. There are three types of memory accesses—normal read/write accesses, merged read/write accesses, and broadcast read accesses. These access types differ as follows:

- Normal Read/Write Accesses – normal read/write memory accesses load or store data in universal registers. Normal accesses read or write one, two, or four 32-bit words needed to load or store the destination or source register indicated in the instruction. Normal accesses occur when the source or destination register size matches the IALU access operator. (See Figure 7-6, Figure 7-9, Figure 7-12, Figure 7-14, Figure 7-16, and Figure 7-18.) Examples of normal accesses (destination <=> source) are:
    - Single register (`Rs`) <=> no operator
    - Dual register (`Rsd`) <=> `L` (long) operator
    - Quad register (`Rsq`) <=> `Q` (quad) operator
- Broadcast Read Access – broadcast read memory accesses load data in compute block data registers. Broadcast accesses read one, two, or four 32-bit words needed to load the destination registers in both compute blocks with the same data as indicated in the instruction. Broadcast accesses occur when the source register size matches the IALU access operator and the register name uses XY (or no prefix) or YX to indicate both compute blocks; XY (or no prefix) and YX yield the *same* results. (See Figure 7-7, Figure 7-10, and Figure 7-13.) Examples of broadcast accesses are (destination <=> source) are:
    - Single register (`Rs`) <= no operator
    - Dual register (`Rsd`) <= `L` (long) operator
    - Quad register (`Rsq`) <= `Q` (quad) operator

- Merged Read/Write Accesses – merged read/write memory accesses load or store data in compute block data registers. Merged accesses read or write the number of 32-bit words needed to load or store the destination or source registers in both compute blocks with the different data as indicated in the instruction. Merged accesses occur when the source or destination register size is one-half the size indicated by the IALU access operator and the register name uses XY (or no prefix) or YX to indicate both compute blocks; XY (or no prefix) and YX syntax yield *different* results. (See Figure 7-8, Figure 7-11, Figure 7-15, and Figure 7-17.) Example merged accesses (destination <=> source) are:
    - Single register (`Rs`) <=> `L` (long) operator
    - Dual register (`Rsd`) <=> `Q` (quad) operator.

Figure 7-6 through Figure 7-17 show only a representative sample of memory access types. These figures only show memory accesses for data registers, not universal registers. Also, these figures only show memory accesses for post-modify, indirect addressing. For a complete list of IALU memory access instruction syntax, see "IALU Instruction Summary" on page 7-42.

Looking at Figure 7-5 (memory contents) and Figure 7-6 through Figure 7-17 (example accesses), it is important to note the relationship between *data size* and *data alignment*. For accesses that load or store a single register, data alignment in memory is not an issue—the index address can be to any address.

For accesses that load or store dual or quad registers, data alignment in memory is important. Dual register-loads or stores must use an index address that is divisible by two (*dual aligned*). Quad-register loads or stores must use an index address that is divisible by four (*quad aligned*). Aligning data in memory is possible with assembler directives and linker description file syntax.

DAB and SDAB accesses do not have these alignment restrictions. For more information on DAB and SDAB accesses, see "Data Alignment Buffer (DAB) Accesses" on page 7-24.

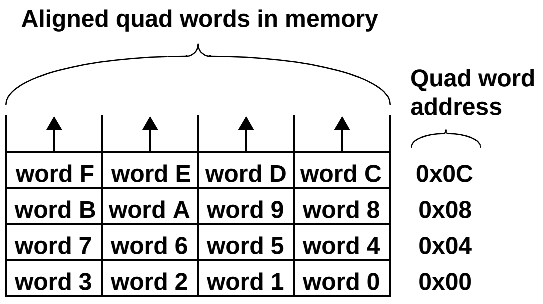
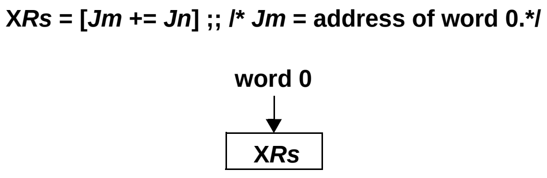
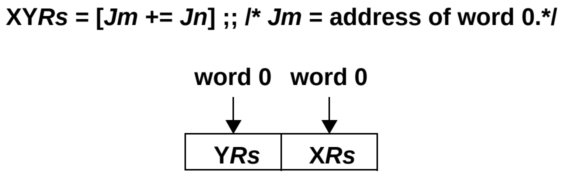


Figure 7-5. Memory Contents for Normal, Broadcast, and Merged Memory Access Examples



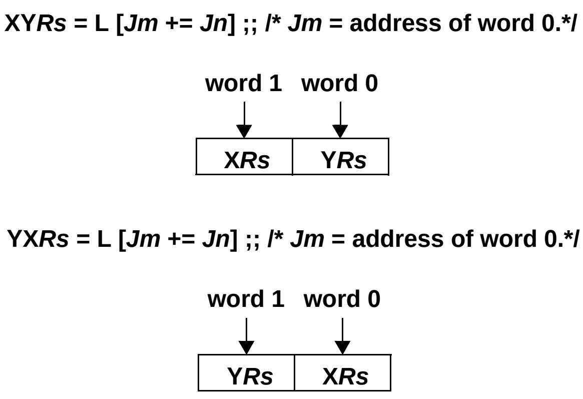
See Figure 7-5 for memory contents.

Figure 7-6. Single Register Normal Read Accesses



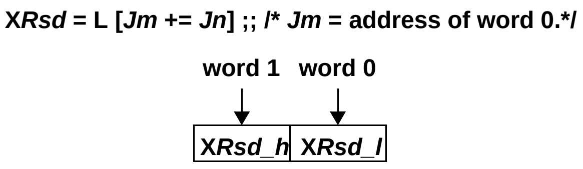
See Figure 7-5 for memory contents.

Figure 7-7. Single Register Broadcast Read Accesses


XYRs = L [Jm += Jn] ;; /* Jm = address of word 0.*/
word 1
word 0
XRs
YRs
YXRs = L [Jm += Jn] ;; /* Jm = address of word 0.*/
word 1
word 0
YRs
XRs

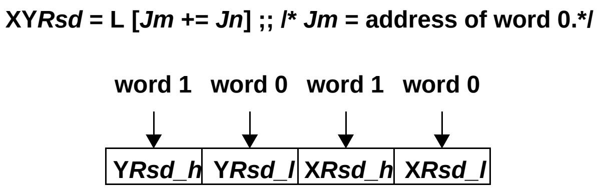
See Figure 7-5 for memory contents.
Note that XY and YX syntax produce different results.

Figure 7-8. Single Register Merged Read Accesses



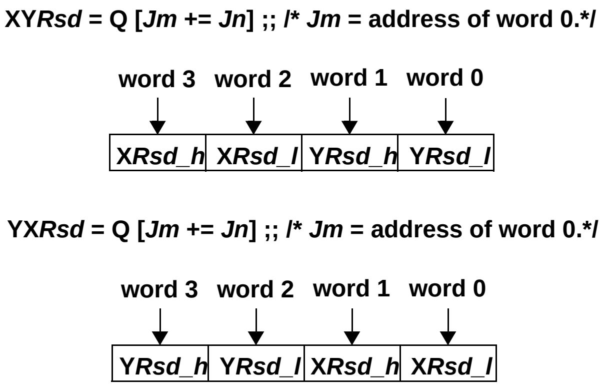
See Figure 7-5 for memory contents.

Figure 7-9. Dual Register Normal Read Accesses



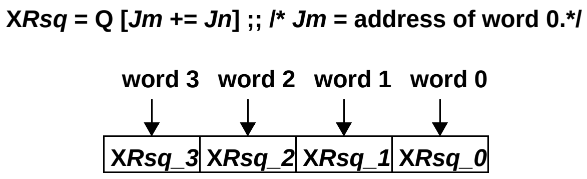
See Figure 7-5 for memory contents.

Figure 7-10. Dual Register Broadcast Read Accesses


XYRsd = Q [Jm += Jn] ;; /* Jm = address of word 0.*/
word 3
word 2
word 1
word 0
XRsd_h
XRsd_l
YRsd_h
YRsd_l
YXRsd = Q [Jm += Jn] ;; /* Jm = address of word 0.*/
word 3
word 2
word 1
word 0
YRsd_h
YRsd_l
XRsd_h
XRsd_l

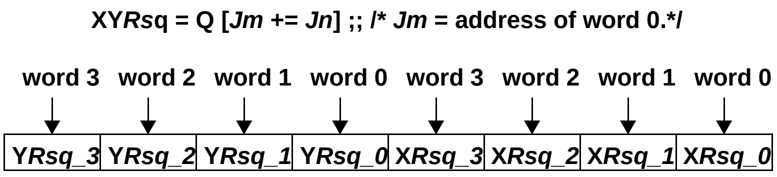
See Figure 7-5 for memory contents.
Note that XY and YX syntax produce different results.

Figure 7-11. Dual Register Merged Read Accesses



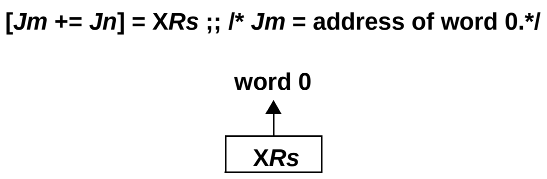
See Figure 7-5 for memory contents.

Figure 7-12. Quad Register Normal Read Accesses



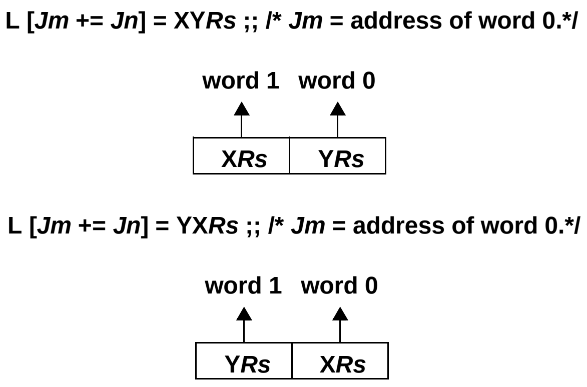
See Figure 7-5 for memory contents.

Figure 7-13. Quad Register Broadcast Read Accesses



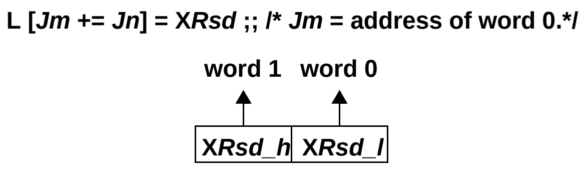
See Figure 7-5 for memory contents.

Figure 7-14. Single Register Normal Write Accesses


L [Jm += Jn] = XYRs ;; /* Jm = address of word 0.*/
word 1
word 0
XRs
YRs
L [Jm += Jn] = YXRs ;; /* Jm = address of word 0.*/
word 1
word 0
YRs
XRs

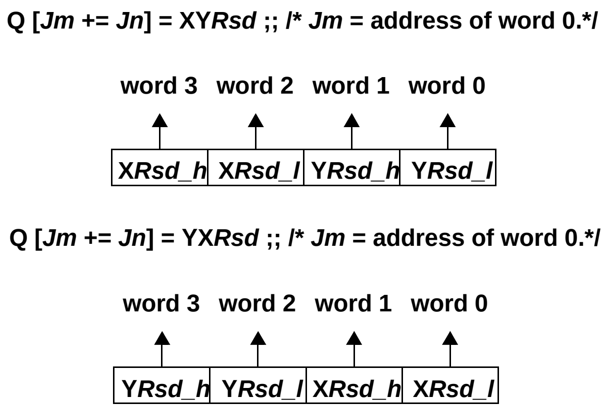
See Figure 7-5 for memory contents.
Note that XY and YX syntax produce different results.

Figure 7-15. Single Register Merged Write Accesses



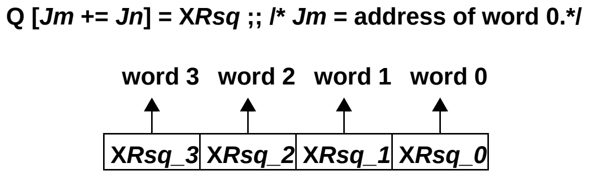
See Figure 7-5 for memory contents.

Figure 7-16. Dual Register Normal Write Accesses


Q [*Jm* += *Jn*] = XY*Rsd* ;; /* *Jm* = address of word 0.*/
word 3
word 2
word 1
word 0
XRsd_h
XRsd_l
YRsd_h
YRsd_l
Q [*Jm* += *Jn*] = YX*Rsd* ;; /* *Jm* = address of word 0.*/
word 3
word 2
word 1
word 0
YRsd_h
YRsd_l
XRsd_h
XRsd_l

See Figure 7-5 for memory contents.
Note that XY and YX syntax produce different results.

Figure 7-17. Dual Register Merged Write Accesses



See Figure 7-5 for memory contents.

Figure 7-18. Quad Register Normal Write Accesses

Data Alignment Buffer (DAB) Accesses

Each compute block has an associated data alignment buffer (X-DAB and Y-DAB) for accessing non-aligned data. Using the DABs, programs can perform a memory read access of non-aligned quad-word data—either four normal words or eight short words—to load data into quad data registers (`Rsq`).

Without using a `DAB` or `SDAB` operator, the data for dual- or quad-register load instructions must be aligned. For more information on data alignment, see "Normal, Merged, and Broadcast Memory Accesses" on page 7-17.

The DAB is a single quad-word FIFO. Aligned quad words from memory are input to the DAB, and non-aligned data for the register load is output from the DAB. The DAB uses its single quad-word buffer to hold data that crosses a quad-word boundary and uses data from the FIFO and current quad-word access to load the registers.

One way to understand DAB operation is to compare aligned versus non-aligned data access and compare DAB operations. Figure 7-19 shows how the DAB operates when an IALU instruction reads aligned data from memory. Compare this to the DAB access of non-aligned data in Figure 7-20.

Figure 7-20 demonstrates some important points about DAB accesses. DAB accesses are intended for repeated series of memory accesses, using circular buffer addressing or linear addressing. It takes one read to prime the DAB—clear out previous data and load in the first correct data for the series—before the DAB is ready for repeated access. The DAB automatically determines the nearest quad-word boundary from the index address and reads the correct quad word from memory to load the DAB.

Because DAB accesses automatically perform circular buffer addressing, the circular buffer address registers—index, base, length, and modifier—must be set up before the DAB access begins. For this reason, DAB

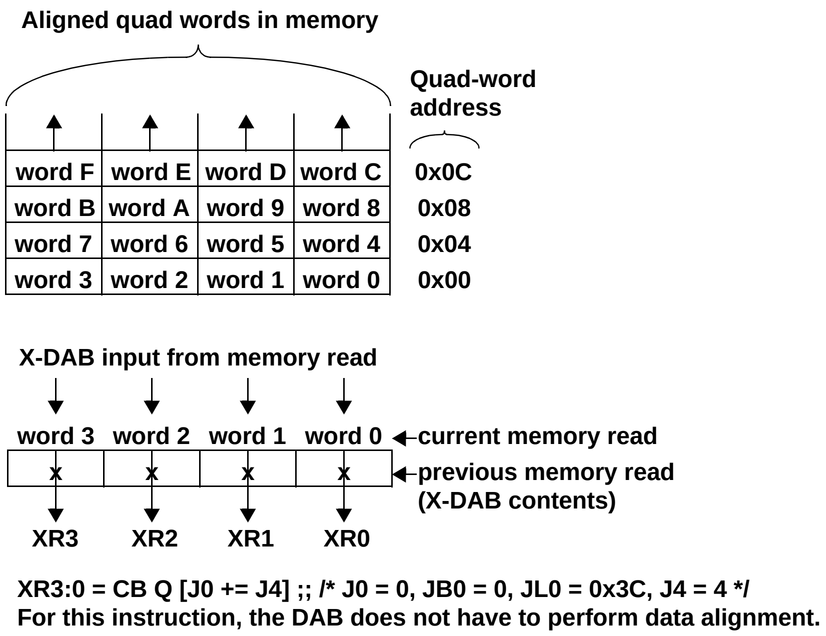
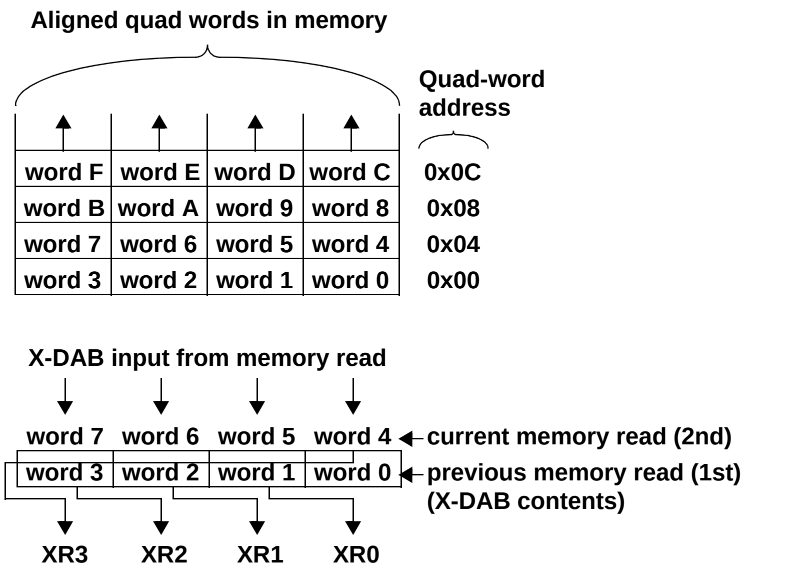

Aligned quad words in memory
Quad-word address
word F
word E
word D
word C
0x0C
word B
word A
word 9
word 8
0x08
word 7
word 6
word 5
word 4
0x04
word 3
word 2
word 1
word 0
0x00
X-DAB input from memory read
word 3
word 2
word 1
word 0
current memory read
x
x
x
x
previous memory read (X-DAB contents)
XR3
XR2
XR1
XR0
XR3:0 = CB Q [J0 += J4] ;; /* J0 = 0, JB0 = 0, JL0 = 0x3C, J4 = 4 */
For this instruction, the DAB does not have to perform data alignment.

Figure 7-19. DAB Operation for Aligned Data

instructions must only use the IALU registers that support circular buffer addressing (`J3–J0`, `K3–K0`). For more information on circular buffer addressing, see "Circular Buffer Addressing" on page 7-31. If circular buffer addressing is used, the modifier value (*`Jn`* or *`Kn`*) must be equal to four to support correct DAB operation.

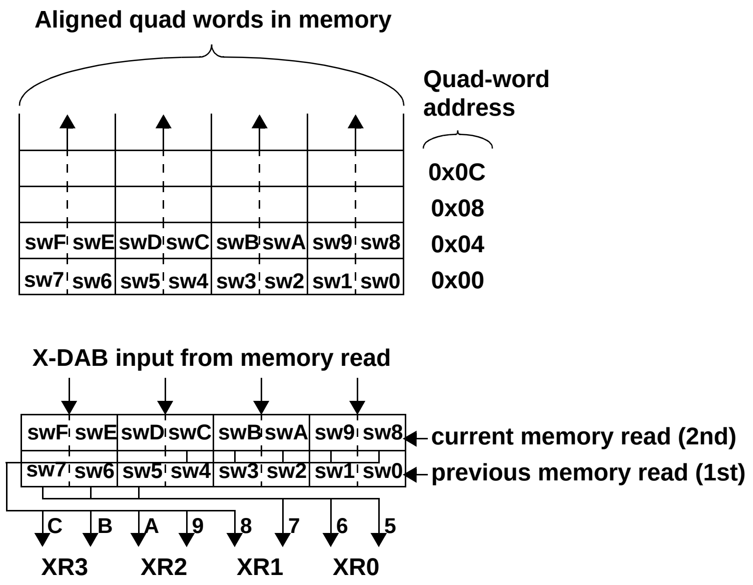
If DAB accesses need to use linear addressing, set the circular buffer length (corresponding *`JL`*/*`KL`* register) to 0.



XR3:0 = DAB Q [J0 += J4] ;; /* J0 = 1, JB0 = 1, JL0 = 0x3C, J4 = 4 */
For this instruction, the DAB performs data alignment. Two reads are needed to prime the DAB. After that, each repeated read places the needed data in the DAB.

Figure 7-20. DAB Operation for Non-Aligned Data

The DAB also provides access to non-aligned short word data in memory as shown in Figure 7-21. Short DAB (SDAB) access has the same requirements for setup and access as DAB access, with two exceptions. First, for correct circular buffer addressing operation the modifier value (`Jn` or `Kn`) must be equal to eight.


Aligned quad words in memory
Quad-word address
0x0C
0x08
0x04
0x00
swF swE swD swC swB swA sw9 sw8
sw7 sw6 sw5 sw4 sw3 sw2 sw1 sw0
X-DAB input from memory read
current memory read (2nd)
previous memory read (1st)
C B A 9 8 7 6 5
XR3 XR2 XR1 XR0

XR3:0 = SDAB Q [J0 += J4] ;; /* J0 = 5, JB0 = 5, JL0 = 0x78, J4 = 8 */
For this instruction, the DAB performs short data alignment. Two reads are needed to prime the DAB. After that, each repeated read places the needed data in the DAB.

Figure 7-21. SDAB Operation for Non-Aligned Data

Second, the index value for SDAB instructions is either 2x (for normal word aligned short words) or 2x+1 (for non-aligned short words). A comparison of these index values for DAB and SDAB instructions appears in Figure 7-22.

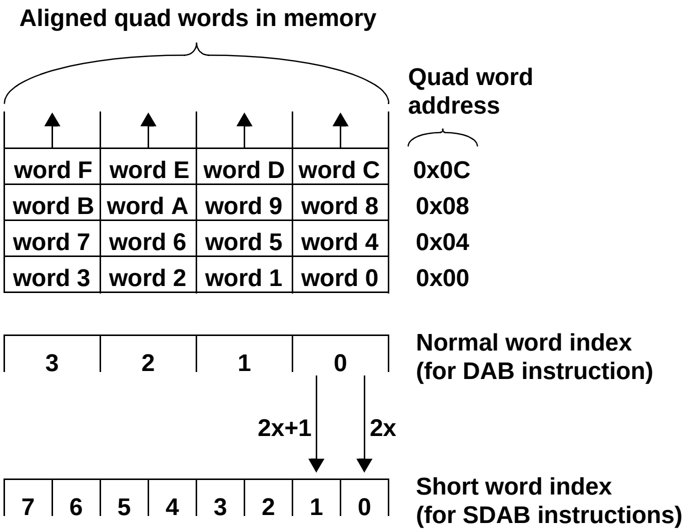


Figure 7-22. DAB Versus SDAB Index Values

Data Alignment Buffer (DAB) Accesses With Offset

The load of compute registers can be with a special DAB shift. Unlike the regular DAB, this operation has a fixed shift of 1 for one compute and 0 for the other compute, on top of the regular DAB shift. The shift can be by one short (16-bit) or one word (32-bit). The instruction syntax is shown in Listing 7-1.

Listing 7-1. Syntax for DAB Accesses With Offset

```
{XY|YX}Rsq = Q XDAB [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts X by one 32-bit word more than address misalignment,
and shift Y by the misalignment */

{XY|YX}Rsq = Q XSDAB [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts X by one 16-bit more than address misalignment, and
shift Y by the misalignment */

{XY|YX}Rsq = Q YDAB [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts Y by one 32-bit word more than address misalignment,
and shift X by the misalignment */

{XY|YX}Rsq = Q YSDAB [Jm += Jn|<Imm8>|<Imm32>] ;
/* shifts Y by one 16-bit word more than address misalignment,
and shift X by the misalignment */
{XY|YX}Rsq = Q XDAB [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts X by one 32-bit word more than address misalignment,
and shift Y by the misalignment */

{XY|YX}Rsq = Q XSDAB [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts X by one 16-bit more than address misalignment, and
shift Y by the misalignment */

{XY|YX}Rsq = Q YDAB [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts Y by one 32-bit word more than address misalignment,
and shift X by the misalignment */
```

```
{XY|YX}Rsq = Q YSDAB [Km += Kn|<Imm8>|<Imm32>] ;
/* shifts Y by one 16-bit word more than address misalignment,
and shift X by the misalignment */

/* m must be 0,1,2, or 3 for bit reverse or circular buffers */
```

Decode of bits 12:10 in load instruction:

- 100 XDAB
- 101 SXDAB
- 110 YDAB
- 111 SYDAB

The alignment for DAB accesses with offset is shown in Table 7-3.

Table 7-3. Data Alignment for DAB Accesses With Offset

Address[2:0] ([1:0] for word access)	**Short DAB**	**Short DAB + misalignment**	**DAB**	**DAB + misalignment**
000	Old[7:0]	New[0], Old[7:1]	Old[3:0]	New[0], Old[3:1]
001	New[0], Old[7:1]	New[1:0], Old[7:2]	New[0], Old[3:1]	New[1:0], Old[3:2]
010	New[1:0], Old[7:2]	New[2:0], Old[7:3]	New[1:0], Old[3:2]	New[2:0], Old[3]
011	New[2:0], Old[7:3]	New[3:0], Old[7:4]	New[2:0], Old[3]	New[3:0]
100	New[3:0], Old[7:4]	New[4:0], Old[7:5]		
101	New[4:0], Old[7:5]	New[5:0], Old[7:6]		

Table 7-3. Data Alignment for DAB Accesses With Offset (Cont'd)

Address[2:0] ([1:0] for word access)	Short DAB	Short DAB + misalignment	DAB	DAB + misalignment
110	New[5:0], Old[7:6]	New[6:0], Old[7]		
111	New[6:0], Old[7]	New[7:0]		

Circular Buffer Addressing

The IALUs support addressing *circular buffers*—a range of addresses containing data that IALU memory accesses step through repeatedly, wrapping around to repeat stepping through the range of addresses in a circular pattern. The memory read or write access instruction uses the operator `CB` to select circular buffer addressing.

To address a circular buffer, the IALU steps the index pointer through the buffer, post-modifying and updating the index on each access with a positive or negative modify value. If the index pointer falls outside the buffer, the IALU subtracts or adds the length of the buffer from or to the value, wrapping the index pointer back to the start of the buffer.

The IALUs use register file and dedicated circular buffering registers for addressing circular buffers. These registers operate as follows for circular buffering:

- The index register contains the value that the IALU outputs on the address bus. In the instruction syntax summary, the index register is represented with *`Jm`* or *`Km`*. This register can be `J3`–`J0` in the J-IALU or `K3`–`K0` in the K-IALU.
- The modify value provides the post-modify amount (positive or negative) that the IALU adds to the index register at the end of each memory access. The modify value can be any register file register in the same IALU as the index register. The modify value also

can be an immediate value instead of a register. The size of the modify value, whether from a register or immediate, must be less than the length of the circular buffer.

- The length register must correspond to the index register; for example, `J0` used with `JL0`, `K0` used with `KL0`, and so on. The length register sets the size of the circular buffer and the address range that the IALU circulates the index register through. If a length register's value is zero, its circular buffer operation is disabled.

- The base register must correspond to the index register; for example, `J0` used with `JB0`, `K0` used with `KB0`, and so on. The base register (the buffer's base address) or the base register plus the length register (the buffer's end address) is the value that the IALU compares the modified index value with after each access to determine buffer wraparound.

Circular buffer addressing may only use post-modify addressing. The IALU cannot support pre-modify addressing for circular buffering, because circular buffering requires the index be updated on each access.

If the *`JL`*/*`KL`* register is set to zero, then circular buffering will not be used.

Example code showing the IALU's support for circular buffer addressing appears in Listing 7-2, and a description of the word access pattern for this example code appears in Figure 7-23.

As shown in Listing 7-2, programs use the following steps to set up a circular buffer:

1. Load the starting address within the buffer into an index register in the selected J-IALU or K-IALU. In the J-IALU, `J3–J0` can be index registers. In the K-IALU, `K3–K0` can be index registers.
2. Load the buffer's base address into the base register that corresponds to the index register. For example, `JB0` corresponds to `J0`.
3. Load the buffer's length into the length register that corresponds to the index register. For example, `JL0` corresponds to `J0`.
4. Load the modify value (step size) into a register file register in the same IALU as the index register. The J-IALU register file is `J30–J0`, and the K-IALU register file is `K30–K0`. Alternatively, an immediate value can supply the modify value.

Listing 7-2. Circular Buffer Addressing Example

```
.section program ;
   JB0 = 0x100000 ;;  /* Set base address */
   JL0 = 11 ;;        /* Set length of buffer */
   J0 = 0x100000 ;;   /* Set location of first address */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100000 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100004 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100008 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100001 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100005 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100009 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100002 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100006 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x10000A */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100003 */
   XR0 = CB [J0 += 4] ;; /* Loads from address 0x100007 */
   XR0 = CB [J0 += 4] ;; /* wrap to load from 0x100000 again */
```

Figure 7-23 shows the sequence order in which the IALU code in Listing 7-2 accesses the 11 buffer locations in one pass. On the twelfth access, the circular buffer wrap around occurs.

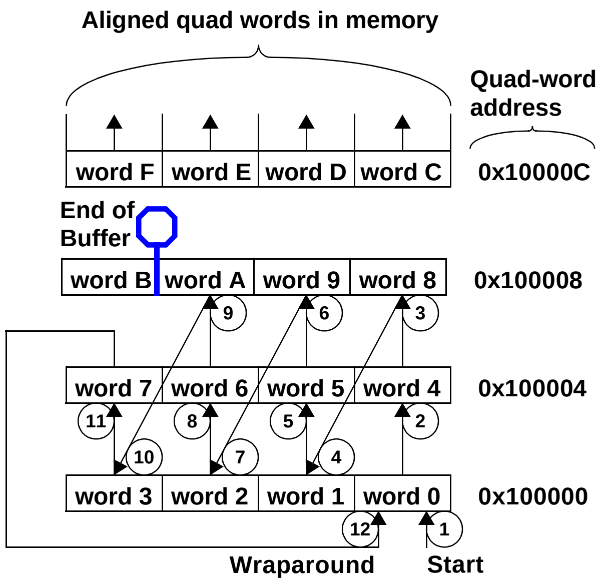


Figure 7-23. Circular Buffer Addressing – Word Access Order

Bit Reverse Addressing

The IALUs support bit reverse addressing through the bit reverse carry operator (BR). When this operator is used with an indirect post-modify read or write access, the bit wise carry moves to the right (instead of left) in the post-modify calculation.

Figure 7-24 provides an example of the bit reverse carry operation. For a regular add operation `0xA5A5 + 0x2121`, the result is `0xB6B6`. For the same add operation with bit reversed carry, the result is `0x9494`.

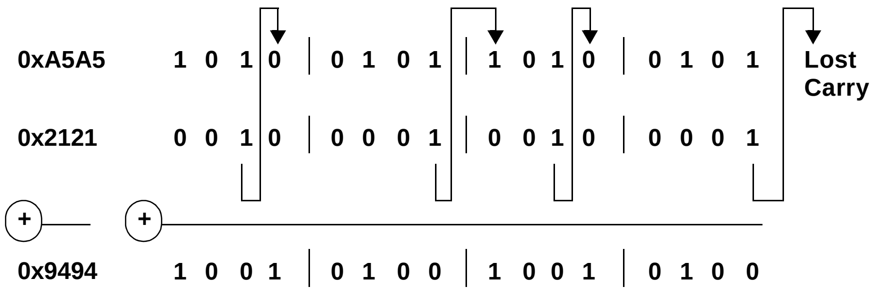


Figure 7-24. Bit Reverse Carry Operation (BR Option)

As with circular buffer operations, bit reverse addressing is only performed using registers `J3–J0` or `K3–K0`. Unlike circular buffers, the related base and length registers in the IALU do not need to be set up for bit reverse addressing.

In bit reverse operations there is no overflow.

Listing 7-3 demonstrates bit reverse addressing. The word access order resulting from the bit reverse carry on the address appears in Figure 7-25. Some important points to remember when reading Listing 7-3 include:

- The number of bit reverse locations (length of buffer) must be a power of 2 (for example, buffer length = 2^n), but the start address for the data buffer to be addressed must be aligned to an address that is a multiple of the number of locations in the buffer. In this case, the length of the buffer is 8, and the buffer start address is aligned to an address that is a multiple of 8.

- The assembler provides evaluation of expressions. For example, where N/2 appears in the code, the assembler evaluates the expression as 4. Also, the assembler `ADDRESS()` operator calculates the address of a symbol.
- For the repeated bit reversed accesses in the loop (`_my_loop`), the data written to the address pointed to by `J4` on each loop iteration is 0, 4, 2, 6, 1, 5, 3, then 7. Note the bit reverse carry operations on each repeated read access and note how they affect the address for each access as shown in Table 7-4.
- Listing 7-3 uses a conditional jump instruction to loop through repeated memory read and write accesses. For more information on conditional execution, see "Conditional Execution" on page 8-12.

Table 7-4. Post-Modify Operations With BR Addition

Loop Iteration	J0	XR0	BR [J0 += 4] note bit reverse carry (BRC) "…" indicates 1000 0000 0000
0	0x1000 0000	0x0	b#…0000 + b#0100 = b#…0100
1	0x1000 0004	0x4	b#…0100 + b#0100 = b#…0010 (BRC)
2	0x1000 0002	0x2	b#…0010 + b#0100 = b#…0110
3	0x1000 0006	0x6	b#…0110 + b#0100 = b#…0001 (BRCs)
4	0x1000 0001	0x1	b#…0001 + b#0100 = b#…0101
5	0x1000 0005	0x5	b#…0101 + b#0100 = b#…0011 (BRCs)
6	0x1000 0003	0x3	b#…0011 + b#0100 = b#…0111
7	0x1000 0007	0x7	b#…0111 + b#0100 = b#…0000 (BRCs)

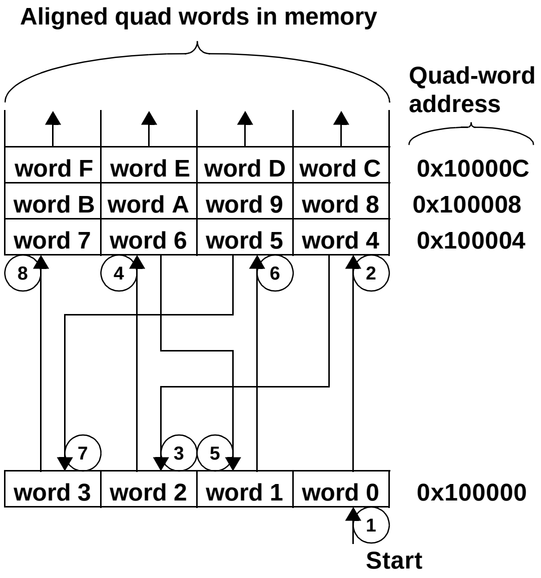


Figure 7-25. Bit Reverse Addressing – Word Access Order

Listing 7-3. Bit Reverse Addressing Example

```
#define N 8  /* N = 8; number of bit reverse locations; N must be
a power of 2 */
.section data1;
  .align N;
/* align the input buffer's start to an address that is a multi-
ple of N; assume for this example that the address is
0x1000 0000 0000 0000 */
  .var input[N]={0,1,2,3,4,5,6,7};
  .var output[N];
.section program;
_main:
  j0 = j31 + ADDRESS(input) ;;   /* Input pointer */
  j4 = j31 + ADDRESS(output) ;;  /* Output pointer */
  LC0 = N ;; /* Set up loop counter */
_my_loop:
  xr0 = BR [J0 += N/2] ;;   /* Data read with bit reverse; modi-
fier must be equal to N/2 */
  if NLC0E, jump _my_loop ; [j4+=1] = xr0 ;;  /* Write linear */
```

Universal Register Transfer Operations

The IALUs support data transfers between universal registers. Also, the IALUs support loading universal registers with 15- or 32-bit immediate data. The `Ureg` transfer and load instructions supported by the IALUs include:

```
Ureg_s = <Imm15>|<Imm32> ; /* load a single Ureg with 15- or
32-bit immediate data */

Ureg_s = Ureg_m ; /* transfer the contents of a single (32-bit)
Ureg to another Ureg */
```

```
Ureg_sd = Ureg_md ; /* transfer the contents of a dual (64-bit)
Ureg to another Ureg */
/* Numbered registers in compute block or IALU register files may
be treated as dual registers */

Ureg_sq = Ureg_mq ; /* transfer the contents of a quad (128-bit)
Ureg to another Ureg */
/* Numbered registers in compute block or IALU register files may
be treated as quad registers */
```

Immediate Extension Operations

Many IALU instructions permit immediate data as an operand. When the immediate data is larger that 8 bits for data addressing instructions or larger than 15 bits for universal register load instructions, the data is too large to fit within one 32-bit instruction. To hold this large immediate data, the DSP uses two instruction slots—one for the instruction and one for the extended data. Looking at the IALU instructions listed in the "IALU Instruction Summary" on page 7-42, note the number of instructions that can use 32-bit immediate data (`<Imm32>`). These instructions all require an immediate extension.

The DSP automatically supports immediate extensions, but the programmer must be aware that an instruction requires an immediate extension and leave an unused slot for the extension. For example, use three (not four) slots on a line in which the DSP must automatically use the second slot for the immediate extension, and place the instruction that needs the extension in the first slot of the instruction line.

Note that only one immediate extension may be in a single instruction line.

IALU Examples

Listing 7-4 and Listing 7-5 provide a number of example IALU arithmetic and data addressing instructions. The comments with the instructions identify the key features of the instruction, such as operands, destination or source addresses, addressing operation, and register usage.

Listing 7-4. IALU Instruction Examples

```
J1 = J0 + 0x81 ; /* Js = Jm + Imm8 data */

K2 = ROTR K0 ; /* Ks = 1 bit right rotate Km */

XSTAT = [J3 + J5] ; /* Load Ureg from addr. Jm + Jn */

J5:4 = L [J2 += 0x81] ; /* Load Ureg_sd from addr. Jm, and
post-modify Jm with Imm8 */

J31:28 = Q [K2 + K5] ; /* Load Ureg_sq from addr. Km + Jn */

XR2 = CB [J0 += 0x8181] ; /* Load Rs from addr. Jm; post-modify Jm
with Imm32 and circular buffer addressing; uses immediate exten-
sion */


YSTAT = 0x8181 ; /* Load Ureg_s from Imm32; */

KSTAT = JSTAT ; /* Load Ureg_s from Ureg_m */

YR3:0 = XR7:4 ; /* Load Ureg_sq from Ureg_mq */
```

Listing 7-5. DAB Usage Example

```
#define N 8
.SECTION data1 ;
.VAR abuff[5] = 0x0, 0x1, 0x2, 0x3, 0x4 ;
.SECTION external ;
.VAR destx[N] = 0x00000000, 0x11111111,
                0x22222222, 0x33333333,
                0x44444444, 0x55555555,
                0x66666666, 0x777777777;
.VAR desty[N] ;
.SECTION program ;
.GLOBAL _main;
_main:
  J0 = 0x3 ;;
  J1 = J0 + destx ;;
  /* The program needs to load in 0x3333333 through
      0x666666 with a quad load, which isn't aligned
      correctly. So, it must use DAB. */
  /* xR3:0 = Q[J1 += 4];; This command gives a runtime error
     (not a compiler error), access to misaligned memory */
  XR3:0 = DAB Q[J1 += 4] ;;
  /* For quad access, must have modify value of 4 for next
     DAB access */
  XR3:0 = DAB Q[J1 += 4] ;;
  /* DAB access takes two of the same commands. First loads in
     nearest quad boundary, and the second loads in correct
     value. Now, every access to the same buffer is
     aligned. */
  JL3 = 0x5 ;; /* Use circular buffers */
  JB3 = abuff ;;
  J3 = abuff ;; /* Must set pointer and JB3 */
  NOP ;;
  NOP;;
```

```
   /* Need 4-cycle latency between JB and JL setup and
      its use */
   /* set up loop to count through 12 steps */
   LC0 = 12 ;; /*initialize loop counter*/
start_loop:
   XR0 = CB[J3 += 0x1] ;;
   /* uses CB to institute circular buffer */
   /* once the use of J3 is a circular buffer, any use
      of J3 is a circular buffer*/
   /* To rest, reset JL3 and JB3 */
   IF NLC0E, JUMP start_loop ;;
   /* execute loop while loop counter doesn't equal zero */
___lib_prog_term:
    jump ___lib_prog_term (NP);;
   /* Done. */
```

IALU Instruction Summary

The following listings show the IALU instructions' syntax:

- Listing 7-6 "IALU Integer, Logical, and Function Instructions"
- Listing 7-7 "IALU Ureg Register Load (Data Addressing) Instructions"
- Listing 7-8 "IALU Dreg Register Load Data Addressing (and DAB Operation) Instructions"
- Listing 7-9 "IALU Ureg Register Store (Data Addressing) Instructions"
- Listing 7-10 "IALU Dreg Register Store (Data Addressing) Instructions"
- Listing 7-11 "IALU Universal Register Transfer 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.
- *`Jm`* or *`Km`* – the letter J or K in register names in italic indicate selection of a register in the J-IALU or K-IALU.
- *`Jm`* – the register names in italic represent user selectable single (*`Jm`*, *`Jn`*, *`Js`*, *`Rs`*, *`Ureg_s`*), double (*`Rsd`*, *`Rmd`*, *`Ureg_sd`*, *`Ureg_md`*) or quad (*`Rsq`*, *`Rmq`*, *`Ureg_sq`*, *`Ureg_mq`*) register names.

In IALU data addressing instructions, special operators identify the input and output operand size as follows:

- The `L` operator before an input or output operand (indirect or direct address) indicates a long word (64-bit) memory read or write. For example, the following instruction syntax indicates long word memory read: `Rsd = L [Km += Kn] ;`
- The `Q` operator before an input or output operand (indirect or direct address) indicates a quad-word (128-bit) memory read or write. For example, the following instruction syntax indicates quad-word memory read: `Rsq = Q [Km += Kn] ;`
- The absence of an `L` or `Q` operator before an input or output operand (indirect or direct address) indicates a normal-word (32-bit) memory read or write. For example, the following instruction syntax indicates normal-word memory read: `Rs = [Km += Kn] ;`

Each instruction presented here occupies one instruction slot in an instruction line, except for those using instructions which require immediate extensions. For more information about instruction lines and instruction combination constraints, see "Immediate Extension Operations" on page 7-39, "Instruction Line Syntax and Structure" on page 1-22, and "Instruction Parallelism Rules" on page 1-26.

Listing 7-6. IALU Arithmetic, Logical, and Function Instructions

```
Js = Jm +|- Jn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
JB0|JB1|JB2|JB3|JL0|JL1|JL2|JL3 = Jm +|-Jn|<Imm8>|<Imm32> ;
Js = Jm + Jn|<Imm8>|<Imm32> + JC ;
Js = Jm - Jn|<Imm8>|<Imm32> + JC - 1 ;
Js = (Jm +|- Jn|<Imm8>|<Imm32>)/2 ;
COMP(Jm, Jn|<Imm8>|<Imm32>) {(U)} ;
Js = MAX|MIN (Jm, Jn|<Imm8>|<Imm32>) ;
Js = ABS Jm ;
Js = Jm OR|AND|XOR|AND NOT Jn|<Imm8>|<Imm32> ;
Js = NOT Jm ;
Js = ASHIFTR|LSHIFTR Jm ;
Js = ROTR|ROTL Jm ;

Ks = Km +|- Kn|<Imm8>|<Imm32> {({CJMP|CB|BR})} ;
KB0|KB1|KB2|KB3|KL0|KL1|KL2|KL3 = Km +|-Kn|<Imm8>|<Imm32> ;
Ks = Km + Kn|<Imm8>|<Imm32> + KC ;
Ks = Km - Kn|<Imm8>|<Imm32> + KC - 1 ;
Ks = (Km +|- Kn|<Imm8>|<Imm32>)/2 ;
COMP(Km, Kn|<Imm8>|<Imm32>) {(U)} ;
Ks = MAX|MIN (Km, Kn|<Imm8>|<Imm32>) ;
Ks = ABS Km ;
Ks = Km OR|AND|XOR|AND NOT Kn|<Imm8>|<Imm32> ;
Ks = NOT Km ;
Ks = ASHIFTR|LSHIFTR Km ;
Ks = ROTR|ROTL Km ;
```

Listing 7-7. IALU Ureg Register Load (Data Addressing) Instructions

```
Ureg_s  =   [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sd = L [Jm +|+= Jn|<Imm8>|<Imm32>] ;
Ureg_sq = Q [Jm +|+= Jn|<Imm8>|<Imm32>] ;

Ureg_s  =   [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sd = L [Km +|+= Kn|<Imm8>|<Imm32>] ;
Ureg_sq = Q [Km +|+= Kn|<Imm8>|<Imm32>] ;

/* Ureg suffix indicates: _s=single, _sd=double, _sq=quad */
```

Listing 7-8. IALU Dreg Register Load Data Addressing (and DAB Operation) Instructions

```
{X|Y|XY}Rs  = {CB|BR}   [Jm += Jn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsd = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rs   = {CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Jm += Jn|<Imm8>|<Imm32>] ;

{XY|YX}Rsq = Q XDAB [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rsq = Q XSDAB [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rsq = Q YDAB [Jm += Jn|<Imm8>|<Imm32>] ;
{XY|YX}Rsq = Q YSDAB [Jm += Jn|<Imm8>|<Imm32>] ;

{X|Y|XY}Rs  = {CB|BR}   [Km += Kn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsd = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rs   = {CB|BR} L [Km += Kn|<Imm8>|<Imm32>] ;
{X|Y|XY}Rsq = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rsd  = {CB|BR|DAB|SDAB} Q [Km += Kn|<Imm8>|<Imm32>] ;

{XY|YX}Rsq = Q XDAB [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rsq = Q XSDAB [Km += Kn|<Imm8>|<Imm32>] ;
```

```
{XY|YX}Rsq = Q YDAB [Km += Kn|<Imm8>|<Imm32>] ;
{XY|YX}Rsq = Q YSDAB [Km += Kn|<Imm8>|<Imm32>] ;

/* R suffix indicates: _s=single, _sd=double, _sq=quad */
/* m must be 0,1,2, or 3 for bit reverse or circular buffers */
```

Listing 7-9. IALU Ureg Register Store (Data Addressing) Instructions

```
  [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_s ;
L [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_sd ;
Q [Jm +|+= Jn|<Imm8>|<Imm32>] = Ureg_sq ;

  [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_s ;
L [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_sd ;
Q [Km +|+= Kn|<Imm8>|<Imm32>] = Ureg_sq ;
```

Listing 7-10. IALU Dreg Register Store (Data Addressing) Instructions

```
{CB|BR}   [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rs ;
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsd ;
{CB|BR} L [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rs ;
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {X|Y}Rsq ;
{CB|BR} Q [Jm += Jn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;

{CB|BR}   [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rs ;
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsd ;
{CB|BR} L [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rs ;
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {X|Y}Rsq ;
{CB|BR} Q [Km += Kn|<Imm8>|<Imm32>] = {XY|YX}Rsd ;

/* R suffix indicates: _s=single, _sd=double, _sq=quad */
/* m = 0,1,2 or 3 for bit reverse or circular buffers */
```

Listing 7-11. IALU Universal Register Transfer Instructions

```
Ureg_s = <Imm15>|<Imm32> ;
Ureg_s = Ureg_m ;
Ureg_sd = Ureg_md ;
Ureg_sq = Ureg_mq ;
```