Appendix E

Integer Inquiry Functions

The CM-5 implementation of C* includes inquiry functions that provide bit-level information about integers. The functions are

Function	Use
leadz	Returns the number of leading 0 bits in the bit-level representation of a value.
leadz_nz	Like **leadz**, except its behavior is identical to the DPEAC **ffb** call.
dimulh	Returns the high 64 bits of the multiplication of two signed integer operands.
dumulh	Returns the high 64 bits of the multiplication of two unsigned integer operands.
popcnt	Returns the number of 1 bits (the population count) in the bit-level representation of an integer.
poppar	Returns 0 if the population parity of the argument value is even, 1 if odd.

All functions are overloaded for 32-bit and 64-bit integers, and for scalar and parallel types. (Overloadings for 64-bit integers are available only when compiling for execution on the vector units.)

If you use any of these functions, include the header file **<bitop.h>**.

Copyright © 1990–1995 Thinking Machines Corporation

E.1 The leadz and leadz_nz Functions

Use the **leadz** function to find out the number of leading 0s in the bit-level representation of a value. Its definitions are

```
#include <bitop.h>

unsigned int leadz(unsigned int x);
unsigned int:current leadz(unsigned int:current x);

unsigned long long leadz(unsigned long long x);
unsigned long long:current leadz (
                        unsigned long long:current x);
```

The function takes as its argument a scalar or parallel integer; the integer can be either 32- or 64-bit. It returns the number of consecutive bits set to 0, beginning from the high-order (leftmost) bit position, before a bit set to 1 is encountered.

If the argument is a parallel variable, **leadz** returns a parallel variable of the same shape, each element of which contains the number of consecutive high-order bits set to 0 for the corresponding element of the parallel-variable argument.

The definitions for **leadz_nz** are

```
#include <bitop.h>

unsigned int leadz_nz(unsigned int x);
unsigned int:current leadz_nz(unsigned int:current x);

unsigned long long leadz_nz(unsigned long long x);
unsigned long long:current leadz_nz (
                        unsigned long long:current x);
```

The **leadz_nz** function is identical to **leadz**, except for two important differences:

- The 32-bit overloading of **leadz_nz** counts the number of leading 0s as if the value contained 64 bits. Thus, if **a** is an **int** whose value is 1, **leadz_nz** returns 63, whereas **leadz** returns 31.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

- **leadz_nz** has a different behavior from that of **leadz** when the integer contains all 0s. In this situation
  - **leadz** returns 32 or 64 (depending on the size of the integer)
  - **leadz_nz** returns 0

The behavior of **leadz_nz** is identical to that of the hardware itself and the DPEAC instruction **ffb** (Find First Bit); for more information on **ffb**, see the *DPEAC Reference Manual*.

E.1.1 Examples

If the value of the scalar integer **a** is 2,

```
b = leadz(a);
```

assigns 30 to **b**, and

```
b = leadz_nz(a);
```

assigns 62 to **b**.

If **p1** is a parallel integer whose values are 2, 3, 4, 5,

```
p2 = leadz(p1);
```

assigns 30, 30, 29, 29 to **p2**, and

```
p2 = leadz_nz(p1);
```

assigns 62, 62, 61, 61 to **p2**.

E.2 The dimulh and dumulh Functions

The **dimulh** and **dumulh** functions return the high 64 bits of the 128-bit result of multiplying two 64-bit integers. (A normal multiply operation returns the low 64 bits.) The **dimulh** function works with signed integers;

Copyright © 1990–1995 Thinking Machines Corporation

dumulh works with unsigned **int**s. Both have parallel and scalar overloadings. Their definitions are

```
#include <bitop.h>

unsigned long long int dumulh(unsigned long long int x,
                              unsigned long long int y);

long long int dimulh(long long int x,
                     long long int y);

unsigned long long int:current dumulh(
         unsigned long long int:current x,
         unsigned long long int:current y);

long long int:current dimulh(
         long long int:current x,
         long long:current y);
```

dimulh and **dumulh** are available only for programs compiled to run on systems with vector units.

E.3 The popcnt Function

The **popcnt** function returns the *population count*; that is, the number of 1 bits in the bit-level representation of a value. Its definitions are

```
#include <bitop.h>

unsigned int popcnt(unsigned int x);
unsigned int:current popcnt(unsigned int:current x);

unsigned int popcnt(unsigned long long x);
unsigned int:current popcnt (
                        unsigned long long:current x);
```

popcnt takes as its argument a scalar or parallel integer; the integer can be either 32- or 64-bit.

If the argument is a parallel variable, **popcnt** returns a parallel variable of the same shape, each element of which is set to the population count for the value of the corresponding element of the argument.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation

E.3.1 Examples

If the value of the scalar integer **a** is 3,

```
b = popcnt(a);
```

assigns 2 to **b**.

If **p1** is a parallel integer whose values are 4, 5, 6, 7,

```
p2 = popcnt(p1);
```

assigns 1, 2, 2, 3 to **p2**.

E.4 The poppar Function

The **poppar** function returns the *population parity*. That is, if the number of 1s (or 0s) in the bit-level representation of a value is even, the function returns 0; if the number of 1s (or 0s) is odd, the function returns 1. Its definitions are

```
#include <bitop.h>

unsigned int poppar(unsigned int x);
unsigned int:current poppar(unsigned int:current x);

unsigned int poppar(unsigned long long x);
unsigned int:current poppar (
                        unsigned long long:current x);
```

poppar takes as its argument a scalar or parallel integer; the integer can be either 32- or 64-bit. If the argument is a parallel variable, **poppar** returns a parallel variable of the same shape, each element of which is set to the population parity for the value of the corresponding element of the argument.

Note that

```
poppar(x);
```

is equivalent to

```
popcnt(x) & 1;
```


Copyright © 1990–1995 Thinking Machines Corporation

E.4.1 Examples

If the value of the scalar integer **a** is 1,

```
b= poppar(a);
```

assigns 1 to **b**.

If the values of the parallel integer **p1** are 2, 3, 4, 5,

```
p2 = poppar(p1);
```

assigns 1, 0, 1, 0 to **p2**.

Version 7.2, May 1995
Copyright © 1990–1995 Thinking Machines Corporation