- 64-bit packed byte integers
- 64-bit packed word integers
- 64-bit packed doubleword integers

Each of these data types provides for multiple integer data elements to be contained (or packed) in a single 64-bit MMX register. Figure 7-6 demonstrates how each data type fits in the 64-bit register.

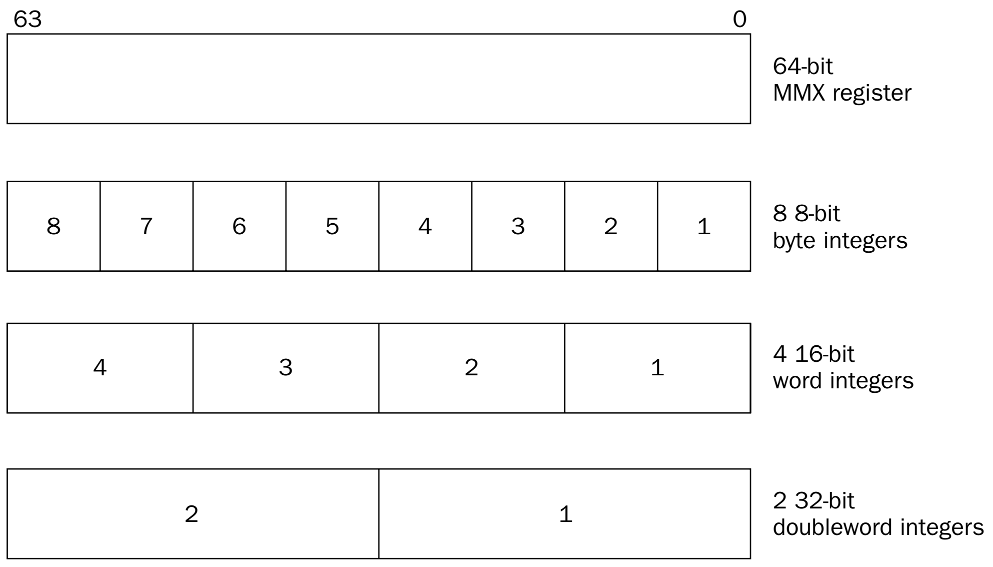

Figure 7-6

As shown in Figure 7-6, 8-byte integers, four-word integers, or two doubleword integers can be packed into a single 64-bit MMX register.

*As discussed in Chapter 2, the **MMX** registers are mapped to the **FPU** registers, so be careful when using **MMX** registers. Remember to save any data stored in the **FPU** registers in memory before using any **MMX** register instructions. This is covered in the "Moving floating-point values" section later in the chapter.*

The MMX platform provides additional instructions for performing parallel mathematical operations on each of the integer values packed into the MMX register.

Moving MMX integers

You can use the MOVQ instruction to move data into an MMX register, but you must decide which of the three packed integer formats your application will use. The format of the MOVQ instruction is

```
movq source, destination
```

where source and destination can be an MMX register, an SSE register, or a 64-bit memory location (although you cannot move MMX integers between memory locations).

The `mmxtest.s` program demonstrates loading doubleword and byte integers into `MMX` registers:

```
# mmxtest.s - An example of using the MMX data types
.section .data
values1:
   .int 1, -1
values2:
   .byte 0x10, 0x05, 0xff, 0x32, 0x47, 0xe4, 0x00, 0x01
.section .text
.globl _start
_start:
   nop
   movq values1, %mm0
   movq values2, %mm1
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The `mmxtest.s` program defines two data arrays. The first one (`values1`) defines two doubleword signed integers, while the second one (`values2`) defines 8-byte signed integer values. The `MOVQ` instruction is used to load the values into the first two `MMX` registers.

After assembling the source code, you can watch what happens in the debugger. After stepping through the `MOVQ` instructions, you can display the values in the `MM0` and `MM1` `MMX` registers:

```
(gdb) print $mm0
$1 = {uint64 = -4294967295, v2_int32 = {1, -1}, v4_int16 = {1, 0, -1, -1},
  v8_int8 = "\001\000\000\000ÿÿÿÿ"}
(gdb) print $mm1
$2 = {uint64 = 72308588487312656, v2_int32 = {855573776, 16835655},
  v4_int16 = {1296, 13055, -7097, 256}, v8_int8 = "\020\005ÿ2Gä\000\001"}
(gdb) print/x $mm1
$3 = {uint64 = 0x100e44732ff0510, v2_int32 = {0x32ff0510, 0x100e447},
  v4_int16 = {0x510, 0x32ff, 0xe447, 0x100}, v8_int8 = {0x10, 0x5, 0xff, 0x32,
    0x47, 0xe4, 0x0, 0x1}}
(gdb)
```

*On the Pentium processors, the **`MMX`** registers are mapped to the existing **`FPU`** registers, so depending on which version of **`gdb`** you are using, displaying the register information in the debugger might be a little tricky. In older versions of **`gdb`**, instead of being able to directly display the **`MMX`** registers, you must display their **`FPU`** register counterparts. The **`mm0`** register is mapped to the first **`FPU`** register, **`st0r`**, and the **`mm1`** register is mapped to the second **`FPU`** register, **`st1`** (this is described in detail in Chapter 9, "Advanced Math Functions"). Unfortunately, the debugger does not know how to interpret the data in the **`FPU`** registers, so you must display it as raw hexadecimal values and interpret it yourself.*

If you are using a newer version of the GNU debugger, you can directly display the `MMX` registers as shown in the preceding code. When displaying the registers, the debugger does not know what format the data is in, so it displays all of the possibilities. The first `print` command displays the contents of the `MM0` register as doubleword integer values. Because the example uses doubleword integer values, the only display format that makes sense is the int32, which displays the correct information. You can produce just this format from the debugger by using the `print/f` command.

Unfortunately, because the MM1 register contains byte integer values, it cannot be displayed in decimal mode. Instead, you can use the x parameter of the print command to display the raw bytes in the register. With this command, you can see that the individual bytes were properly placed in the MM1 register.

SSE integers

The Streaming SIMD Extensions (SSE) technology (also described in Chapter 2) provides eight 128-bit XMM registers (named XMM0 through XMM7) for handling packed data. The SSE2 technology (introduced in the Pentium 4 processor) provides four additional packed signed integer data types:

- 128-bit packed byte integers
- 128-bit packed word integers
- 128-bit packed doubleword integers
- 128-bit packed quadword integers

These values are packed into the 128-bit XMM registers as shown in Figure 7-7.

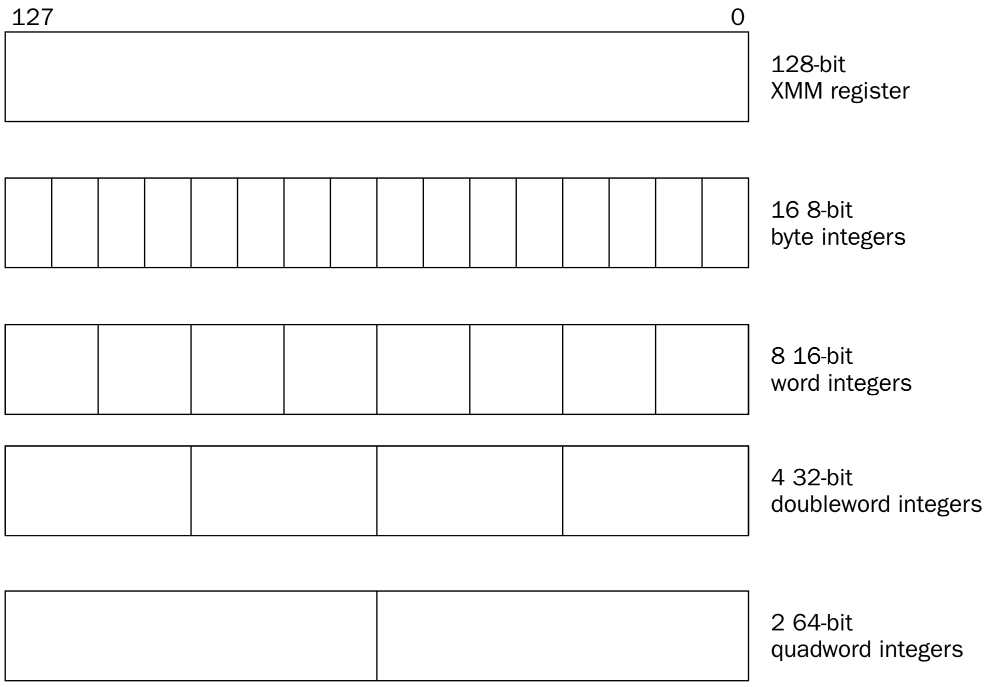

Figure 7-7

As shown in Figure 7-7, there can be 16-byte integers, eight-word integers, four doubleword integers, or two quadword integers packed into a single 128-bit SSE register. The SSE platform provides additional instructions for performing parallel mathematical operations on the packed data values in the SSE registers. This enables the processor to process significantly more information using the same clock cycles.

Moving SSE integers

The MOVDQA and MOVDQU instructions are used to move 128 bits of data into the XMM registers, or to move data between XMM registers. The A and U parts of the mnemonic stand for aligned and unaligned, referring to how the data is stored in memory. For data that is aligned on a 16-byte boundary, the A option is used; otherwise, the U option is used (Chapter 5, "Moving Data" describes aligned data).

The format of both the MOVDQA and MOVDQU instruction is

```
movdqa source, destination
```

where source and destination can be either an SSE 128-bit register, or a 128-bit memory location (but again, you cannot move data between two memory locations). The SSE instructions perform faster when using aligned data. Also, if a program uses the MOVDQA instruction on unaligned data, a hardware exception will result.

The ssetest.s program demonstrates moving 128-bit data values into SSE registers:

```
# ssetest.s - An example of using 128-bit SSE registers
.section .data
values1:
   .int 1, -1, 0, 135246
values2:
   .quad 1, -1
.section .text
.globl _start
_start:
   nop
   movdqu values1, %xmm0
   movdqu values2, %xmm1

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

The ssetest.s program defines two data arrays containing different types of integer data. The values1 array contains four doubleword signed integer values, while the values2 array contains two quadword signed integer values. The MOVDQU instruction is used to move both data arrays into SSE registers.

After assembling the program, you can watch the results in the debugger. The debugger is able to display the SSE registers (XMM0 through XMM7) using the print command:

```
(gdb) print $xmm0
$1 = {v4_float = {1.40129846e-45, -nan(0x7fffff), 0, 1.89520012e-40},
  v2_double = {-nan(0xfffff00000001), 2.8699144274488922e-309},
  v16_int8 = "\001\000\000\000ÿÿÿÿ\000\000\000\000N\020\002", v8_int16 = {1,
    0, -1, -1, 0, 0, 4174, 2}, v4_int32 = {1, -1, 0, 135246}, v2_int64 = {
    -4294967295, 580877146914816},
  uint128 = 0x0002104e00000000ffffffff00000001}
(gdb) print $xmm1
$2 = {v4_float = {1.40129846e-45, 0, -nan(0x7fffff), -nan(0x7fffff)},
    v2_double = {4.9406564584124654e-324, -nan(0xfffffffffffff)},
    v16_int8 = "\001\000\000\000\000\000\000\000ÿÿÿÿÿÿÿÿ", v8_int16 = {1, 0, 0,
      0, -1, -1, -1, -1}, v4_int32 = {1, 0, -1, -1}, v2_int64 = {1, -1},
    uint128 = 0xffffffffffffffff0000000000000001}
(gdb)
```

After the `MOVDQU` instructions, the `XMM0` and `XMM1` registers contain the data values defined in the data section. The `XMM0` register contains the four doubleword signed integer data values, and the `XMM1` register contains the two quadword signed integer data values.

Remember that the ***`ssetest.s`*** *program will only run on Pentium III or later processors. Chapter 17, "Using Advanced IA-32 Features," describes the* ***`MMX`*** *and* ***`SSE`*** *instruction sets and demonstrates how they are used.*

Binary Coded Decimal

The Binary Coded Decimal (BCD) data type has been available for quite a long time in computer systems. The BCD format is often used to simplify working with devices that use decimal numbers (such as devices that must display numbers to humans, such as clocks and timers). Instead of converting decimal numbers to binary for mathematical operations, and then back to decimal again, the processor can keep the numbers in BCD format and perform the mathematical operations. Understanding how BCD works and how the processor uses it can come in handy in your assembly language programming. The following sections describe the BCD format and how the BCD data type is handled by the IA-32 platform.

What is BCD?

BCD does pretty much what is says, it codes decimal numbers in a binary format. Each BCD value is an unsigned 8-bit integer, with a value range of 0 to 9. The 8-bit values higher than 9 are considered invalid in BCD. Bytes containing BCD values are combined to represent decimal digits. In a multibyte BCD value, the lowest byte holds the decimal ones value, the next higher one holds the tens value, and so on. This is demonstrated in Figure 7-8.

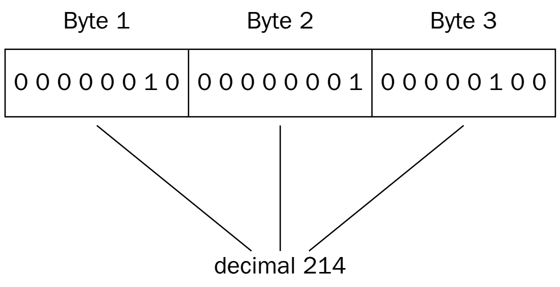

Figure 7-8