
The MMX Instruction Set

Chapter Eleven

11.1 Chapter Overview

While working on the Pentium and Pentium Pro processors, Intel was also developing an instruction set architecture extension for multimedia applications. By studying several existing multimedia applications, developing lots of multimedia related algorithms, and through simulation, Intel developed 57 instructions that would greatly accelerate the execution of multimedia applications. The end result was their multimedia extensions to the Pentium processor that Intel calls the MMX Technology Instructions.

Prior to the invention of the MMX enhancements, good quality multimedia systems required separate digital signal processors and special electronics to handle much of the multimedia workload¹. The introduction of the MMX instruction set allowed later Pentium processors to handle these multimedia tasks without these expensive digital signal processors (DSPs), thus lowering the cost of multimedia systems. So later Pentiums, Pentium II, Pentium III, and Pentium IV processors all have the MMX instruction set. Earlier Pentiums (and CPUs prior to the Pentium) and the Pentium Pro do not have these instructions available. Since the instruction set has been available for quite some time, you can probably use the MMX instructions without worrying about your software failing on many machines.

In this chapter we will discuss the MMX Technology instructions and how to use them in your assembly language programs. The use of MMX instructions, while not completely limited to assembly language, is one area where assembly language truly shines since most high level languages do not make good use of MMX instructions except in library routines. Therefore, writing fast code that uses MMX instructions is mainly the domain of the assembly language programmer. Hence, it's a good idea to learn these instructions if you're going to write much assembly code.

11.2 Determining if a CPU Supports the MMX Instruction Set

While it's almost a given that any modern CPU your software will run on will support the MMX extended instruction set, there may be times when you want to write software that will run on a machine even in the absence of MMX instructions. There are two ways to handle this problem – either provide two versions of the program, one with MMX support and one without (and let the user choose which program they wish to run), or the program can dynamically determine whether a processor supports the MMX instruction set and skip the MMX instructions if they are not available.

The first situation, providing two different programs, is the easiest solution from a software development point of view. You don't actually create two source files, of course; what you do is use conditional compilation statements (i.e., `#IF..#ELSE..#ENDIF`) to selectively compile MMX or standard instructions depending on the presence of an identifier or value of a boolean constant in your program. See “Conditional Compilation (Compile-Time Decisions)” on page 962 for more details.

Another solution is to dynamically determine the CPU type at run-time and use program logic to skip over the MMX instructions and execute equivalent standard code if the CPU doesn't support the MMX instruction set. If you're expecting the software to run on an Intel Pentium or later CPU, you can use the `CPUID` instruction to determine whether the processor supports the MMX instruction set. If MMX instructions are available, the `CPUID` instruction will return bit 23 as a one in the feature flags return result.

The following code illustrates how to use the `CPUID` instruction. This example does not demonstrate the entire `CPUID` sequence, but shows the portion used for detection of MMX technology.

1. A good example was the Apple Quadra 660AV and 840AV computer systems; they were built around the Motorola 68040 processor rather than a Pentium, but the 68040 was no more capable of handling multimedia applications than the Pentium. However, an on-board DSP (digital signal processor) CPU allowed the Quadoras to easily handle audio applications that the 68040 could not.

```
// For a perfectly general routine, you should determine if this
// is a Pentium or later processor. We'll assume at least a Pentium
// for now, since most OSes expect a Pentium or better processor.

    mov( 1, eax );           // Request for CPUID feature flags.
    CPUID();                // Get the feature flags into EDX.
    test( $80_0000, edx );  // Is bit 23 set?
    jnz HasMMX;
```

This code assumes at least the presence of a Pentium Processor. If your code needs to run on a 486 or 386 processor, you will have to detect that the system is using one of these processors. There is tons of code on the net that detects different processors, but most of it will not run under 32-bit OSes since the code typically uses protected (non-user-mode) instructions. Some operating systems provide a system call or environment variable that will specify the CPU. We'll not go into the details here because 99% of the users out there that are running modern operating systems have a CPU that supports the MMX instruction set or, at least, the CPUID instruction.

11.3 The MMX Programming Environment

The MMX architecture extends the Pentium architecture by adding the following:

- Eight MMX registers (MM0..MM7).
- Four MMX data types (packed bytes, packed words, packed double words, and quad word).
- 57 MMX Instructions.

11.3.1 The MMX Registers

The MMX architecture adds eight 64-bit registers to the Pentium. The MMX instructions refer to these registers as MM0, MM1, MM2, MM3, MM4, MM5, MM6, and MM7. These are strictly data registers, you cannot use them to hold addresses nor are they suitable for calculations involving addresses.

Although MM0..MM7 appear as separate registers in the Intel Architecture, the Pentium processors alias these registers with the FPU's registers (ST0..ST7). Each of the eight MMX 64-bit registers is physically equivalent to the L.O. 64-bits of each of the FPU's registers (see Figure 11.1). The MMX registers overlay the FPU registers in much the same way that the 16-bit general purpose registers overlay the 32-bit general purpose registers.

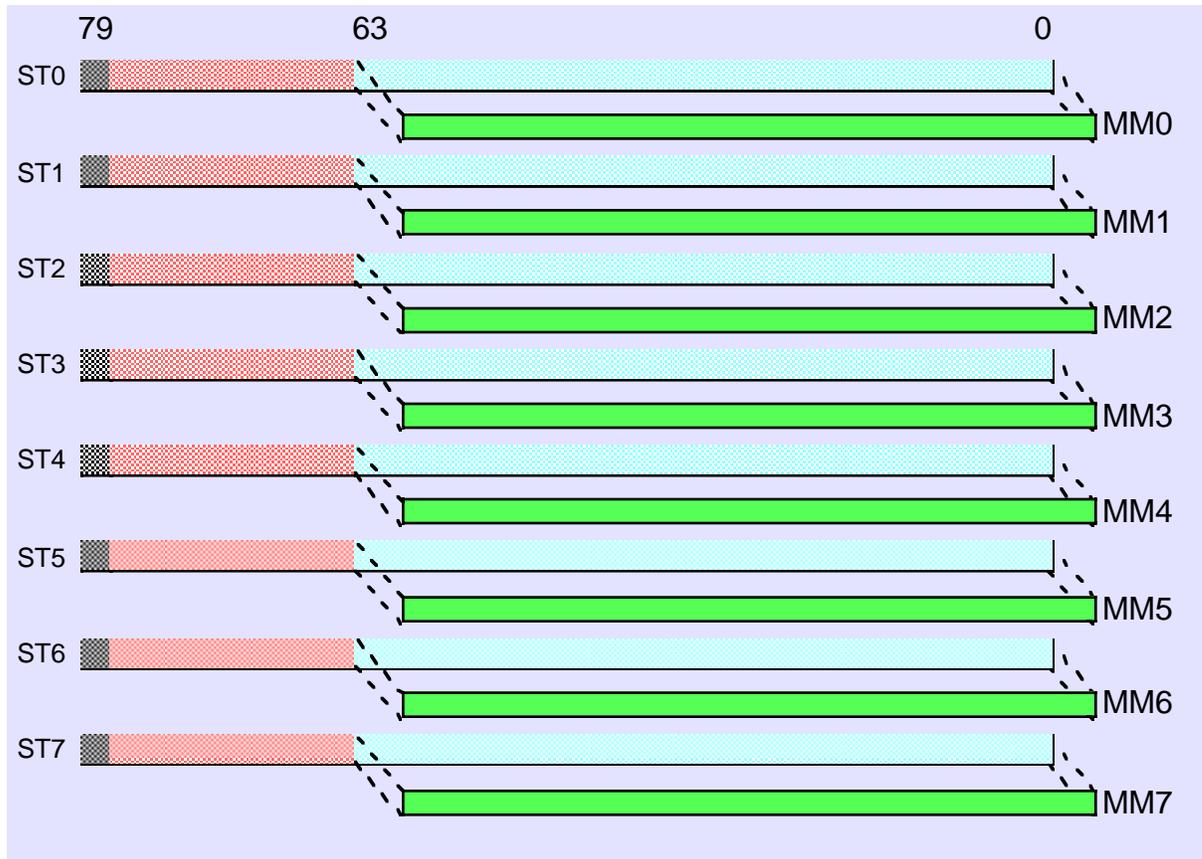


Figure 11.1 MMX and FPU Register Aliasing

Because the MMX registers overlay the FPU registers, you cannot mix FPU and MMX instructions in the same computation sequence. You can begin executing an MMX instruction sequence at any time; however, once you execute an MMX instruction you cannot execute another FPU instruction until you execute a special MMX instruction, EMMS (Exit MMX Machine State). This instruction resets the FPU so you may begin a new sequence of FPU calculations. The CPU does not save the FPU state across the execution of the MMX instructions; executing EMMS clears all the FPU registers. Because saving FPU state is very expensive, and the EMMS instruction is quite slow, it's not a good idea to frequently switch between MMX and FPU calculations. Instead, you should attempt to execute the MMX and FPU instructions at different times during your program's execution.

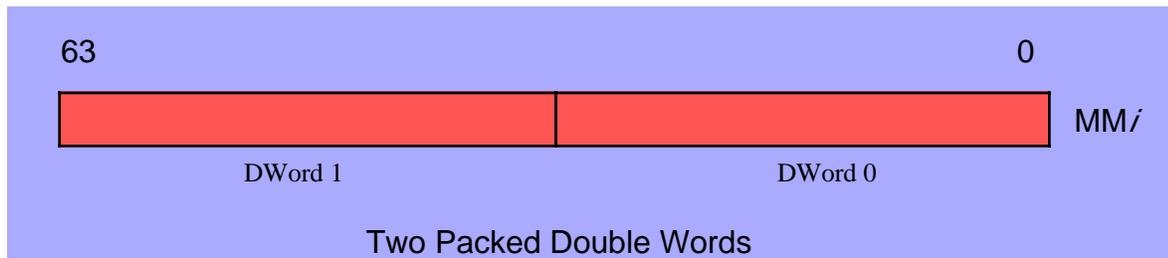
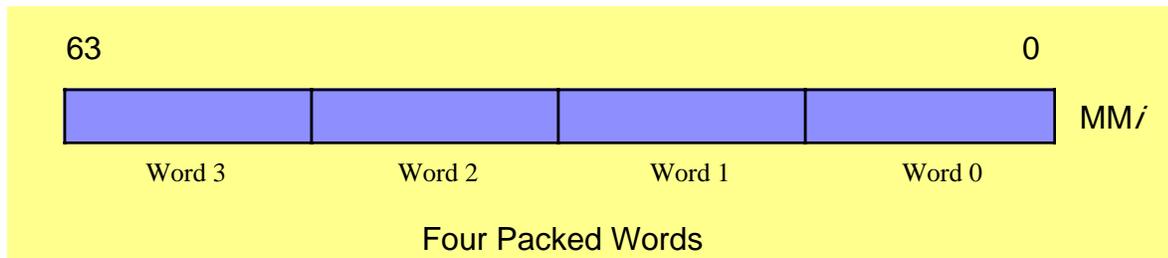
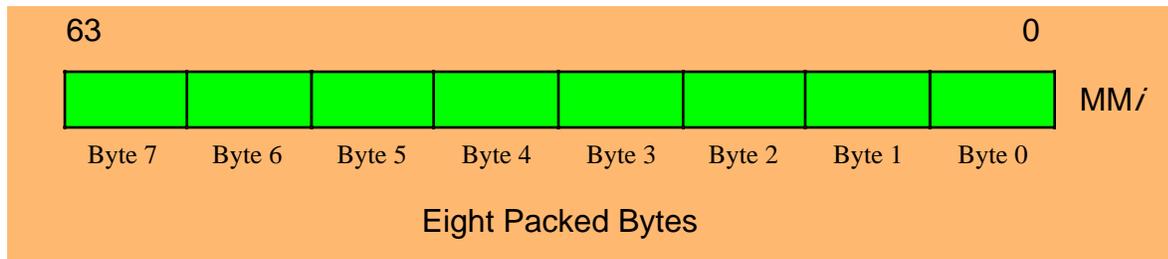
You're probably wondering why Intel chose to alias the MMX registers with the FPU registers. Intel, in their literature, brags constantly about what a great idea this was. You see, by aliasing the MMX registers with the FPU registers, Microsoft and other multitasking OS vendors did not have to write special code to save the MMX state when the CPU switched from one process to another. The fact that the OS automatically saved the FPU state means that the CPU would automatically save the MMX state as well. This meant that the new Pentium chips with MMX technology that Intel created were automatically compatible with Windows 95, Windows NT, and Linux without any changes to the operating system code.

Of course, those operating systems have long since been upgraded and Microsoft (and Linux developers) could have easily provided a "service pack" to handle the new registers (had Intel chosen not to alias the FPU and MMX registers). So while aliasing MMX with the FPU provided a very short-lived and temporary benefit, in retrospect Intel made a big mistake with this decision. They've obviously realized their mistake, because as they've introduced new "streaming" instructions (the floating point equivalent of the MMX instruction set) they've added new registers (XMM0..XMM7) without using this trick. It's too bad they

don't fix the problem in their current CPUs (there is no technical reason why they can't create separate MMX and FPU registers at this point). Oh well, you'll just have to live with the fact that you can't execute interleaved FPU and MMX instructions.

11.3.2 The MMX Data Types

The MMX instruction set supports four different data types: an eight-byte array, a four-word array, a two element double word array, and a quadword object. Each MMX register processes one of these four data types (see Figure 11.2).



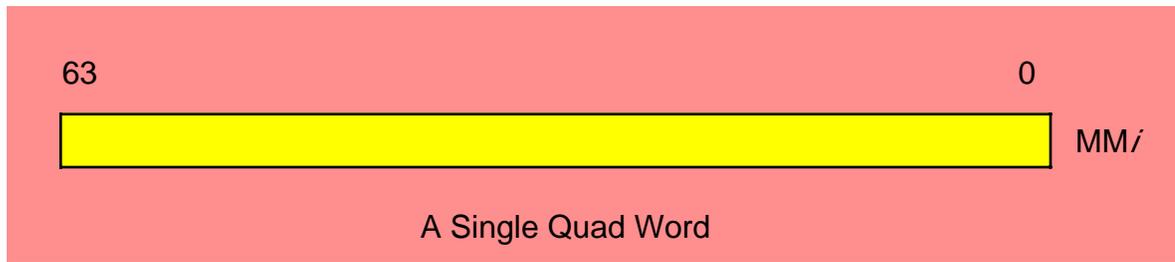


Figure 11.2 The MMX Data Types

Despite the presence of 64-bit registers, the MMX instruction set does not extend the 32-bit Pentium processor to 64-bits. Instead, after careful study Intel added only those 64-bit instructions that were useful for multimedia operations. For example, you cannot add or subtract two 64-bit integers with the MMX instruction set. In fact, only the logical and shift operations directly manipulate 64 bits.

The MMX instruction set was not designed to provide general 64-bit capabilities to the Pentium. Instead, the MMX instruction set provides the Pentium with the capability of performing multiple eight-, sixteen-, or thirty-two bit operations simultaneously. In other words, the MMX instructions are generally SIMD (Single Instruction Multiple Data) instructions (see “Parallel Processing” on page 268 for an explanation of SIMD). For example, a single MMX instruction can add eight separate pairs of byte values together. This is not the same as adding two 64-bit values since the overflow from the individual bytes does not carry over into the higher order bytes. This can accelerate a program that needs to add a long string of bytes together since a single MMX instruction can do the work of eight regular Pentium instructions. This is how the MMX instruction set speeds up multimedia applications – by processing multiple data objects in parallel with a single instruction. Given the data types the MMX instruction set supports, you can process up to eight byte objects in parallel, four word objects in parallel, or two double words in parallel.

11.4 The Purpose of the MMX Instruction Set

The Single Instruction Multiple Data model the MMX architecture supports may not look all that impressive when viewed with a SISD (Single Instruction, Single Data) bias. Once you’ve mastered the basic integer instructions on the 80x86, it’s difficult to see the application of the MMX’s SIMD instruction set. However, the MMX instructions directly address the needs of modern media, communications, and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words, and double words).

For example, most programs use a stream of bytes or words to represent audio and video data. The MMX instructions can operate on eight bytes or four words with a single instruction, thus accelerating the program by almost a factor of four or eight.

One drawback to the MMX instruction set is that it is not general purpose. Intel’s research that led to the development of these new instructions specifically targeted audio, video, graphics, and another multimedia applications. Although some of the instructions are applicable in many general programs, you’ll find that many of the instructions have very little application outside their limited domain. Although, with a lot of deep thought, you can probably dream up some novel uses of many of these instructions that have nothing whatsoever at all to do with multimedia, you shouldn’t get too frustrated if you cannot figure out why you would want to use a particular instruction; that instruction probably has a specific purpose and if you’re not trying to code a solution for that problem, you may not be able to use the instruction. If you’re questioning why Intel would put such limited instructions in their instruction set, just keep in mind that although you can use the instruction(s) for lots of different purposes, they are invaluable for the few purposes they are uniquely suited.

11.5 Saturation Arithmetic and Wraparound Mode

The MMX instruction set supports saturating arithmetic (see “Sign Extension, Zero Extension, Contraction, and Saturation” on page 73). When manipulating standard integer values and an overflow occurs, the standard integer instructions maintain the correct L.O. bits of the value in the integer while truncating any overflow². This form of arithmetic is known as *wraparound* mode since the L.O. bits wrap back around to zero. For example, if you add the two eight-bit values \$02 and \$FF you wind up with a carry and the result \$01. The actual sum is \$101, but the operation truncates the ninth bit and the L.O. byte wraps around to \$01.

In saturation mode, results of an operation that overflow or underflow are clipped (saturated) to some maximum or minimum value depending on the size of the object and whether it is signed or unsigned. The result of an operation that exceeds the range of a data-type saturates to the maximum value of the range. A result that is less than the range of a data type saturates to the minimum value of the range.

Table 1:

Data Type	Decimal		Hexadecimal	
	Lower Limit	Upper Limit	Lower Limit	Upper Limit
Signed Byte	-128	+127	\$80	\$7f
Unsigned Byte	0	255	0	\$ff
Signed Word	-32768	+32767	\$8000	\$7fff
Unsigned Word	0	65535	0	\$ffff

For example, when the result exceeds the data range limit for signed bytes, it is saturated to \$7f; if a value is less than the data range limit, it is saturated to \$80 for signed bytes. If a value exceeds the range for unsigned bytes, it is saturated to \$ff or \$00.

This saturation effect is very useful for audio and video data. For example, if you are amplifying an audio signal by multiplying the words in the CD-quality 44.1 kHz audio stream by 1.5, clipping the value at +32767, while introducing distortion, sounds far better than allowing the waveform to wrap around to -32768. Similarly, if you are mixing colors in a 24-bit graphic or video image, saturating to white produces much more meaningful results than wrap-around.

Since Intel created the MMX architecture to support audio, graphics, and video, it should come as no surprise that the MMX instruction set supports saturating arithmetic. For those applications that require saturating arithmetic, having the CPU automatically handle this process (rather than having to explicitly check after each calculation) is another way the MMX architecture speeds up multimedia applications.

11.6 MMX Instruction Operands

Most MMX instructions operate on two operands, a source and a destination operand. A few instructions have three operands with the third operand being a small immediate (constant) value. In this section we'll take a look at the common MMX instruction operands.

2. For some instructions the overflow may appear in another register or the carry flag, but in the destination register the high order bits are lost.

The destination operand is almost always an MMX register. In fact, the only exceptions are those instructions that store an MMX register into memory. The MMX instructions always leave the result of MMX calculations in an MMX register.

The source operand can be an MMX register or a memory location. The memory location is usually a quad word entity, but certain instructions operate on double word objects. Note that, in this context, “quad word” and “double word” mean eight or four consecutive bytes in memory; they do not necessarily imply that the MMX instruction is operating on a qword or dword object. For example, if you add eight bytes together using the PADDDB (packed add bytes) instruction, PADDDB references a qword object in memory, but actually adds together eight separate bytes.

For most MMX instructions, the generic HLA syntax is one of the following:

```
mmxInstr( source, dest );
```

The specific forms are

```
mmxInstr( mmi, mmi ); // i=0..7
mmxInstr( mem, mmi ); // i=0..7
```

MMX instructions access memory using the same addressing modes as the standard integer instructions. Therefore, any legal 80x86 addressing mode is usable in an MMX instruction. For those instructions that reference a 64-bit memory location, HLA requires that you specify an anonymous memory object (e.g., “[ebx]” or “[ebp+esi*8+6]”) or a qword variable.

A few instructions require a small immediate value (or constant). For example, the shift instructions let you specify a shift count as an immediate value in the range 0..63. Another instruction uses the immediate value to specify a set of four different count values in the range 0..3 (i.e., four two-bit count values). These instructions generally take the following form:

```
mmxInstr( imm8, source, dest );
```

Note that, in general, MMX instructions do not allow you to specify immediate constants as operands except for a few special cases (such as shift counts). In particular, the source operand to an MMX instruction has to be a register or a quad word variable, it cannot be a 64-bit constant. To achieve the same effect as specifying a constant as the source operand, you must initialize a quad word variable in the READONLY (or STATIC) section of your program and specify this variable as the source operand. Unfortunately, HLA does not support 64-bit constants, so initializing the value is going to be a bit of a problem. There are two solutions to this problem: break the constant into smaller pieces (bytes, words, or double words) and emit the constant in pieces that HLA can process; or you can write your own numeric conversion routine(s) using the HLA compile-time language to allow the emission of a 64-bit constant. We’ll explore both of those approaches here.

The first approach is the one you will most commonly use. Very few MMX instructions actually operate on 64-bit data operands; instead, they typically operate on a (small) array of bytes, words, or double words. Since HLA provides good support for byte, word, and double word constant expressions, specifying a 64-bit MMX memory operand as a short array of objects is probably the best way to create this data. Since the MMX instructions that fetch a source value from memory expect a 64-bit operand, you must declare such objects as qword variables, e.g.,

```
static
    mmxVar:qword;
```

The big problem with this declaration is that the qword type does not allow an initializer (since HLA cannot handle 64-bit constant expressions). Since this declaration occurs in the STATIC segment, HLA will initialize *mmxVar* with zero; probably not the value you’re interested in supplying here.

There are two ways to solve this problem. The first way is to attach the @NOSTORAGE option to the MMX variable declarations in the STATIC segment. The data declarations that immediately follow the variable definition provide the initial data for that variable. Here’s an example of such a declaration:

```
static
    mmxDVar: qword; @nostorage;
```

```
dword $1234_5678, $90ab_cdef;
```

Note that the `DWORD` directive above stores the double word constants in successive memory locations. Therefore, `$1234_5678` will appear in the L.O. double word of the 64-bit value and `$90ab_cdef` will appear in the H.O. double word of the 64-bit value. Always keep in mind that the L.O. objects come first in the list following the `DWORD` (or `BYTE`, or `WORD`, or ???) directive; this is opposite of the way you're used to reading 64-bit values.

The example above used a `DWORD` directive to provide the initialization constant. However, you can use any data declaration directive, or even a combination of directives, as long as you allocate at least eight bytes (64-bits) for each qword constant. The following data declaration, for example, initializes eight eight-bit constants for an MMX operand; this would be perfect for a `PADDB` instruction or some other instruction that operates on eight bytes in parallel:

```
static
  eightBytes: qword; @nostorage;
  byte 0, 1, 2, 3, 4, 5, 6, 7;
```

Although most MMX instructions operate on small arrays of bytes, words, or double words, a few actually do operate on 64-bit quantities. For such memory operands you would probably prefer to specify a 64-bit constant rather than break it up into its constituent double word values. This way, you don't have to remember to put the L.O. double word first and perform other mental adjustments.

Although HLA does not support 64-bit constants in the compile time language, HLA is flexible enough to allow you to extend the language to handle such declarations. Program 11.1 demonstrates how to write a macro to accept a 64-bit hexadecimal constant. This macro will automatically emit two `DWORD` declarations containing the L.O. and H.O. components of the 64-bit value you specify as the *qword16* (quadword constant, base 16) macro parameter. You would typically use the *qword16* macro as follows:

```
static
  HOOnes: qword; @nostorage;
  qword16( $FFFF_FFFF_0000_0000 );
```

The `qword16` macro would emit the following:

```
dword 0;
dword $FFFF_FFFF;
```

Without further ado, here's the macro (and a sample test program):

```
program qwordConstType;
#include( "stdlib.hhf" )

// The following macro accepts a 64-bit hexadecimal constant
// and emits two dword objects in place of the constant.

macro qword16( theHexVal ):hs, len, dwval, mplier, curch, didLO;

  // Remove whitespace around the macro parameter (shouldn't
  // be any, but just in case something weird is going on) and
  // convert all lower case characters to upper case.

  ?hs := @uppercase( @trim( @string:theHexVal, 0 ), 0 );

  // If there is a leading "$" symbol, strip it from the string.

  #if( @substr( hs, 0, 1 ) = "$" )

    ?hs := @substr( hs, 1, 256 );
```

```

#endif

// Process each character in the string from the L.O. digit
// through to the H.O. digit.  Add the digit, multiplied by
// some successive power of 16, to the current sum we're
// accumulating in dwval.  When we cross a dword boundary,
// emit the L.O. dword and start over.

?len := @length( hs );      // Number of characters to process.
?dwval:dword := 0;         // Accumulate value here.
?mplier:dword := 1;       // Power of 16 to multiply by.
?didLO:boolean := false;  // Checks for overflow.
#while( len > 0 )         // Repeat for each char in string.

    // For each character in the string, verify that it is
    // a legal hexadecimal character and merge it in with the
    // current accumulated value if it is.  Print an error message
    // if we come across an illegal character.

    ?len := len - 1;      // Next available char.
    ?curch := char( @substr( hs, len, 1 ) ); // Get the character.
    #if( curch in { '0'..'9' } ) // See if valid decimal digit.

        // Accumulate result if decimal digit.

        ?dwval := dwval +
            (uns8( curch ) - uns8( '0' )) * mplier;

    #elseif( curch in { 'A'..'F' } ) // See if valid hex digit.

        // Accumulate result if a hexadecimal digit.

        ?dwval := dwval +
            (uns8( curch ) - uns8( 'A' ) + 10) * mplier;

    // Ignore underscore characters and report an error for anything
    // else we find in the string.

    #elseif( curch <> '_' )

        #error( "Illegal character in 64-bit hexadecimal constant" )
        #print( "Character = '", curch, "' Rest of string: '", hs, "' )

#endif

// If it's not an underscore character, adjust the multiplier value.
// If we cross a dword boundary, emit the L.O. value as a dword
// and reset everything for the H.O. dword.

#if( curch <> '_' )

    // If the current value fits in 32 bits, process this
    // as though it were a dword object.

    #if( mplier < $1000_0000 )

        ?mplier := mplier * 16;

    #elseif( len > 0 )

```

```

// Down here we've just processed the last hex
// digit that will fit into 32 bits. So emit the
// L.O. dword and reset the multiplier and dwval constants.

?multiplier := 1;
dword dwval;
?dwval := 0;

// If we've been this way before, we've got an
// overflow.

#if( didLO )

    #error( "64-bit overflow in constant" );

#endif
?didLO := true;

#endif

#endif

#endif

// Emit the H.O. dword here.

dword dwval;

// If the constant only consumed 32 bits, we've got to emit a zero
// for the H.O. dword at this point.

#if( !didLO )

    dword 0;

#endif

endmacro;

static
    x:qword; @nostorage;
    qword16( $1234_5678_90ab_cdef );
    qword16( 100 );

begin qwordConstType;

    stdout.put( "64-bit value of x = $" );
    stdout.putq( x );
    stdout.newln();

end qwordConstType;

```

Program 11.1 qword16 Macro to Process 64-bit Hexadecimal Constants

Although it's a little bit more difficult, you could also write a *qword10* macro that lets you specify decimal constants as the macro operand rather than hexadecimal constants. The implementation of *qword10* is left as a programming exercise at the end of this volume.

11.7 MMX Technology Instructions

The following subsections describe each of the MMX instructions in detail. The organization is as follows:

- Data Transfer Instructions,
- Conversion Instructions,
- Packed Arithmetic Instructions,
- Comparisons,
- Logical Instructions,
- Shift and Rotate Instructions,
- the EMMS Instruction.

These sections describe *what* these instructions do, not *how* you would use them. Later sections will provide examples of how you can use several of these instructions.

11.7.1 MMX Data Transfer Instructions

```
movd( reg32, mmi );
movd( mem32, mmi );
movd( mmi, reg32 );
movd( mmi, mem32 );

movq( mem64, mmi );
movq( mmi, mem64 );
movq( mmi, mmi );
```

The MOVD (move double word) instruction copies data between a 32-bit integer register or double word memory location and an MMX register. If the destination is an MMX register, this instruction zero-extends the value while moving it. If the destination is a 32-bit register or memory location, this instruction copies the L.O. 32-bits of the MMX register to the destination.

The MOVQ (move quadword) instruction copies data between two MMX registers or between an MMX register and memory. If either the source or destination operand is a memory object, it must be a qword variable or HLA will complain.

11.7.2 MMX Conversion Instructions

```
packssdw( mem64, mmi );
packssdw( mmi, mmi );

packsswb( mem64, mmi );
packsswb( mmi, mmi );

packusdw( mem64, mmi );
packusdw( mmi, mmi );
```

```

packuswb( mem64, mmi );
packuswb( mmi, mmi );

punpckhbw( mem64, mmi );
punpckhbw( mmi, mmi );

punpckhdq( mem64, mmi );
punpckhdq( mmi, mmi );

punpckhwd( mem64, mmi );
punpckhwd( mmi, mmi );

punpcklbw( mem64, mmi );
punpcklbw( mmi, mmi );

punpckldq( mem64, mmi );
punpckldq( mmi, mmi );

punpcklwd( mem64, mmi );
punpcklwd( mmi, mmi );

```

The `PACKSSxx` instructions pack and saturate signed values. They convert a sequence of larger values to a sequence of smaller values via saturation. Those instructions with the *dw* suffix pack four double words into four words; those with the *wb* suffix saturate and pack eight signed words into eight signed bytes.

The `PACKSSDW` instruction takes the two double words in the source operand and the two double words in the destination operand and converts these to four signed words via saturation. The instruction packs these four words together and stores the result in the destination MMX register. See Figure 11.3 for details.

The `PACKSSWB` instruction takes the four words from the source operand and the four signed words from the destination operand and converts, via signed saturation, these values to eight signed bytes. This instruction leaves the eight bytes in the destination MMX register. See Figure 11.4 for details.

One application for these pack instructions is to convert UNICODE to ASCII (ANSI). You can convert UNICODE (16-bit) character to ANSI (8-bit) character if the H.O. eight bits of each UNICODE character is zero. The `PACKUSWB` instruction will take eight UNICODE characters and pack them into a string that is eight bytes long with a single instruction. If the H.O. byte of any UNICODE character contains a non-zero value, then the `PACKUSWB` instruction will store \$FF in the respective byte; therefore, you can use \$FF as a conversion error indication.

Another use for the `PACKSSWB` instruction is to translate a 16-bit audio stream to an eight-bit stream. Assuming you've scaled your sixteen-bit values to produce a sequence of values in the range -128..+127, you can use the `PACKSSWB` instruction to convert that sequence of 16-bit values into a packed sequence of eight bit values.

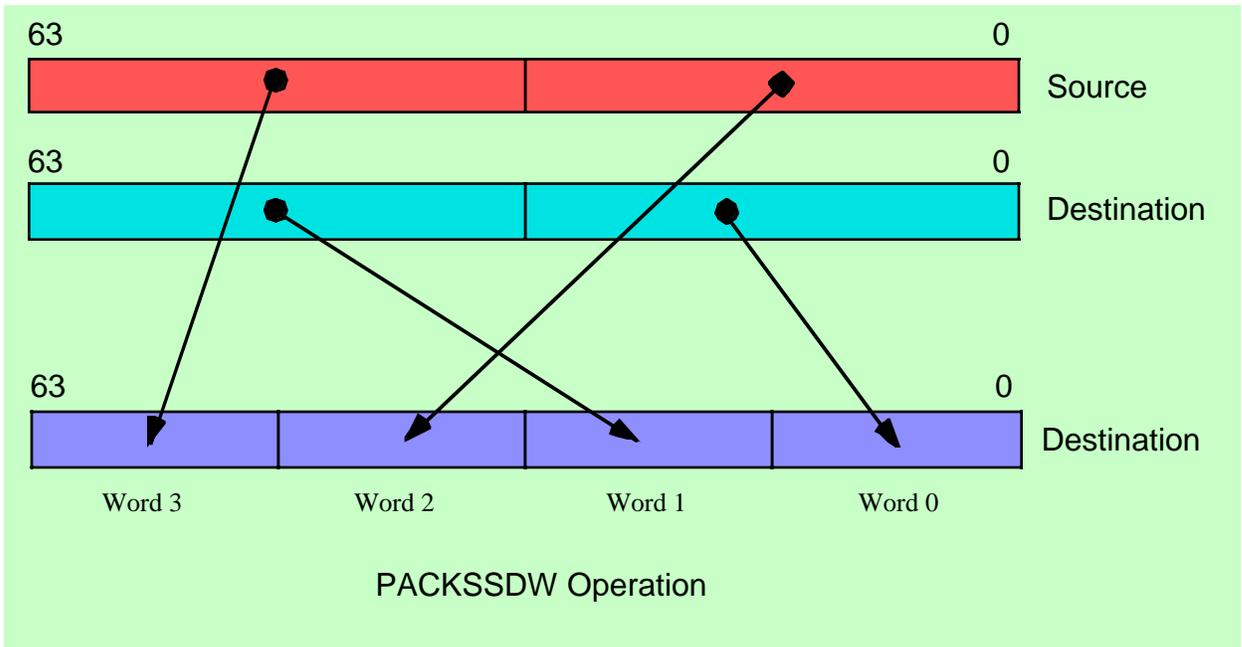


Figure 11.3 PACKSSDW Instruction

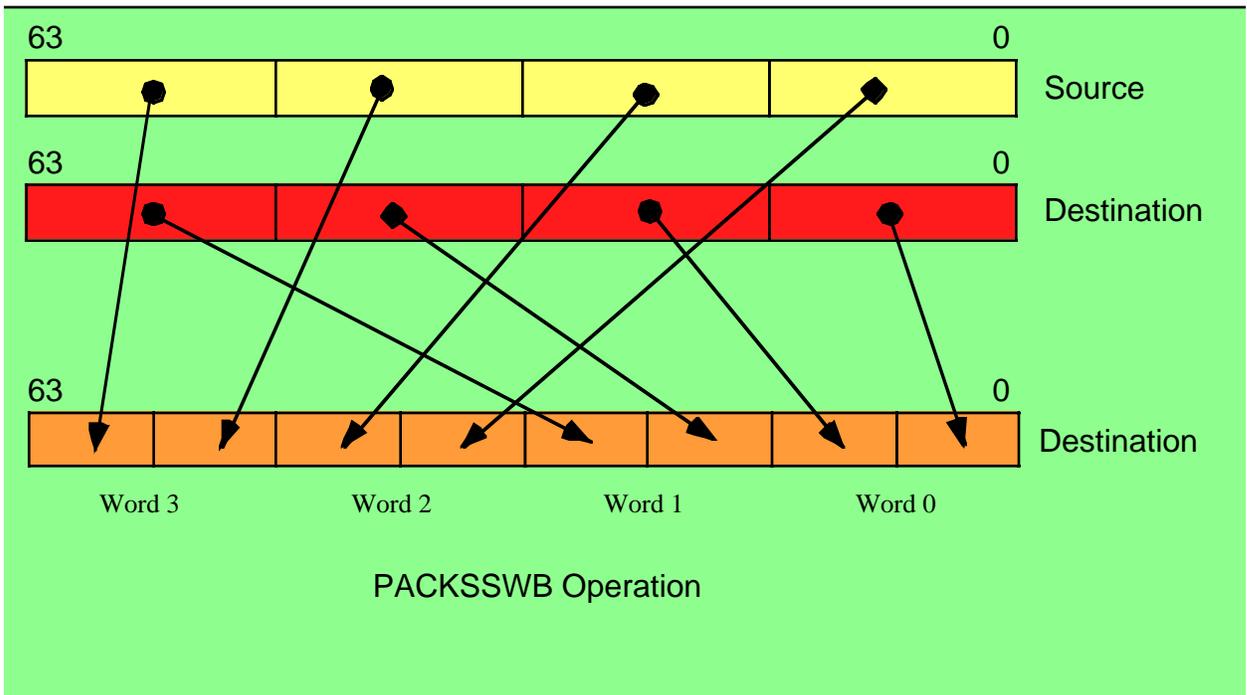


Figure 11.4 PACKSSWB Instruction

The unpack instructions (PUNPCKxxx) provide the converse operation to the pack instructions. The

unpack instructions take a sequence of smaller, packed, values and translate them into larger values. There is one problem with this conversion, however. Unlike the pack instructions, where it took two 64-bit operands to generate a single 64-bit result, the unpack operations will produce a 64-bit result from a single 32-bit result. Therefore, these instructions cannot operate directly on full 64-bit source operands. To overcome this limitation, there are two sets of unpack instructions: one set unpacks the data from the L.O. double word of a 64-bit object, the other set of instructions unpacks the H.O. double word of a 64-bit object. By executing one instruction from each set you can unpack a 64-bit object into a 128-bit object.

The PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ instructions merge (unpack) the L.O. double words of their source and destination operands and store the 64-bit result into their destination operand.

The PUNPCKLBW instruction unpacks and interleaves the low-order four bytes of the source (first) and destination (second) operands. It places the L.O. four bytes of the destination operand at the even byte positions in the destination and it places the L.O. four bytes of the source operand in the odd byte positions of the destination operand.(see Figure 11.5).

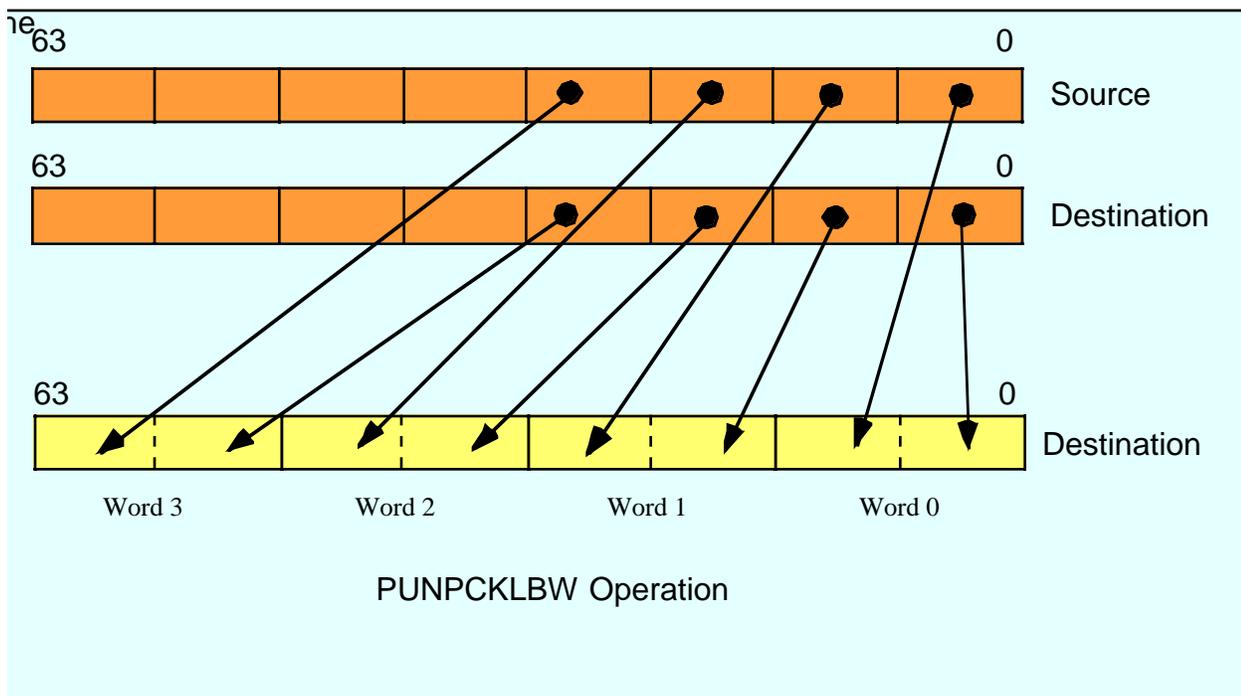


Figure 11.5 UNPCKLBW Instruction

The PUNPCKLWD instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand (see Figure 11.6).

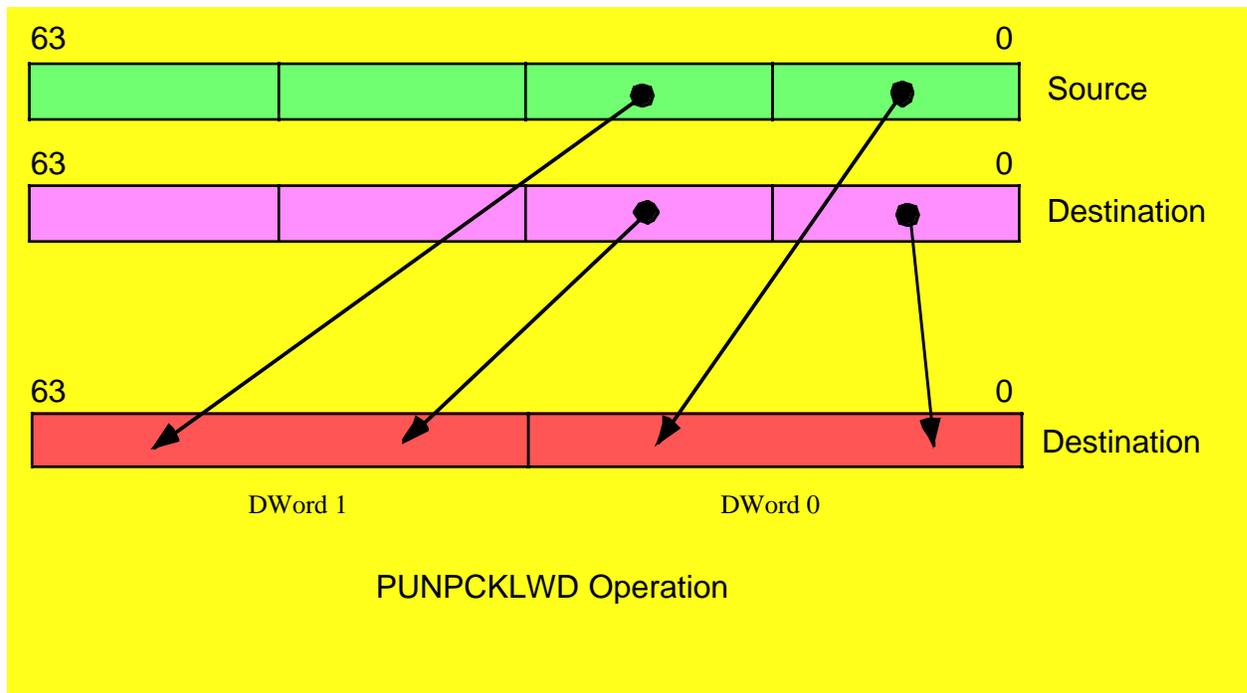


Figure 11.6 The PUNPCKLWD Instruction

The PUNPCKDQ instruction copies the L.O. dword of the source operand to the L.O. dword of the destination operand and it copies the (original) L.O. dword of the destination operand to the L.O. dword of the destination (i.e., it doesn't change the L.O. dword of the destination, see Figure 11.7).

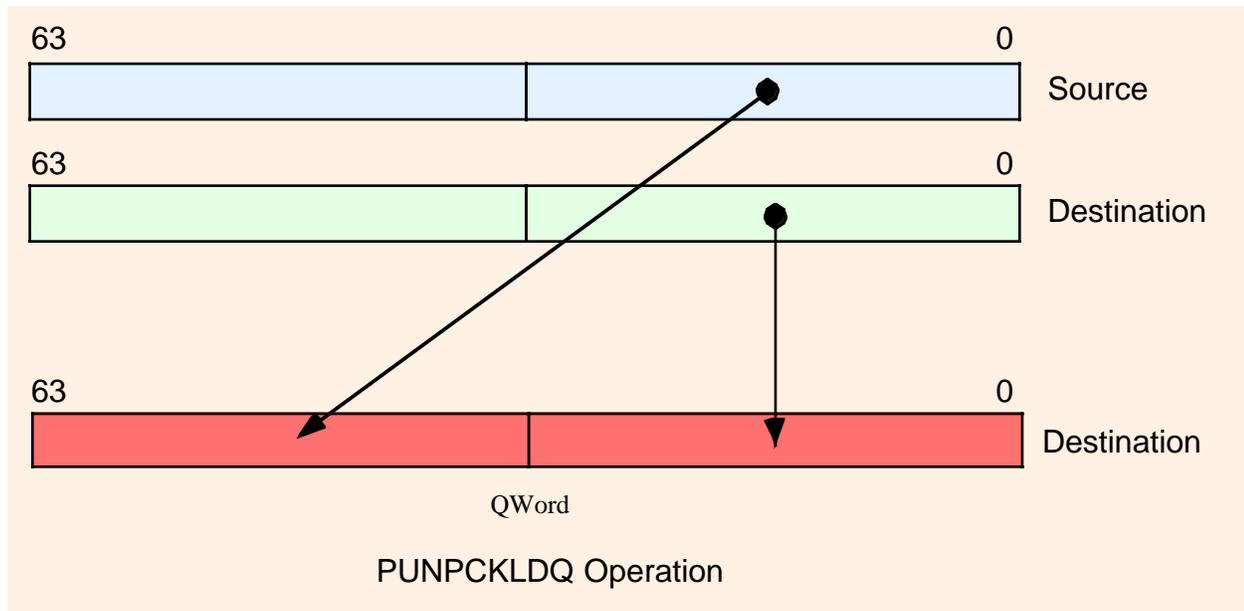


Figure 11.7 PUNPCKLDQ Instruction

The PUNPCKHBW instruction is quite similar to the PUNPCKLBW instruction. The difference is that it unpacks and interleaves the high-order four bytes of the source (first) and destination (second) operands. It places the H.O. four bytes of the destination operand at the even byte positions in the destination and it places the H.O. four bytes of the source operand in the odd byte positions of the destination operand (see Figure 11.8).

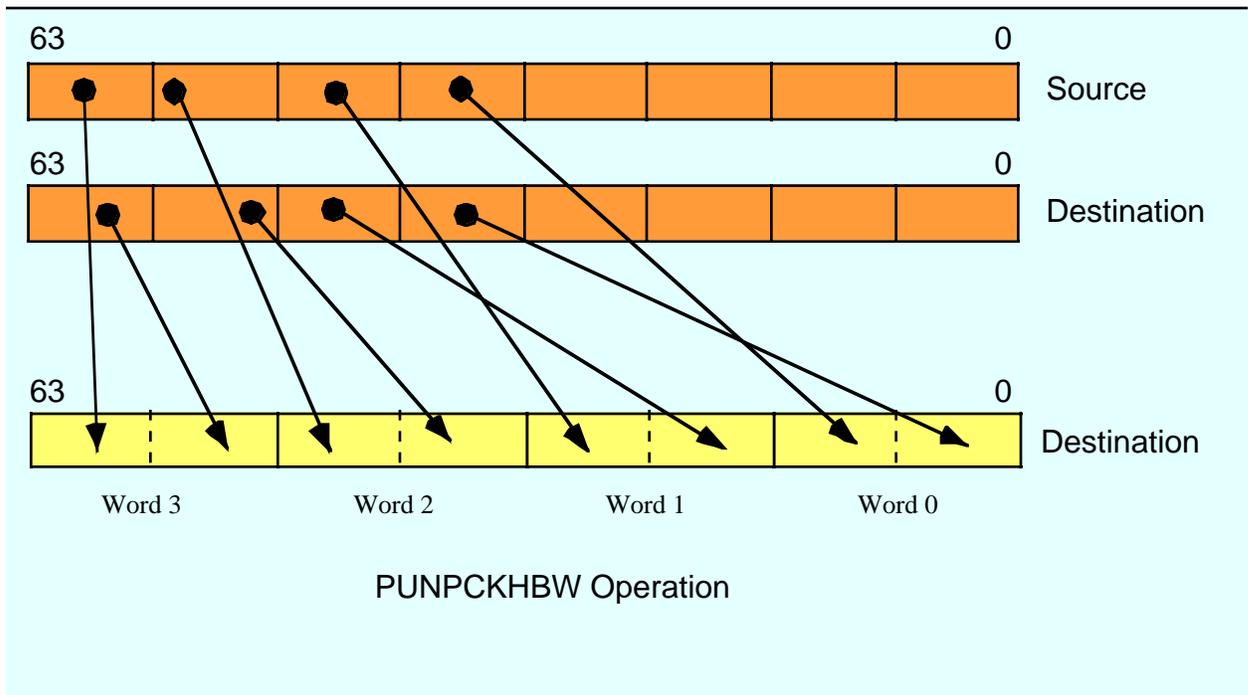


Figure 11.8 PUNPCKHBW Instruction

The PUNPCKHWD instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand (see Figure 11.9)

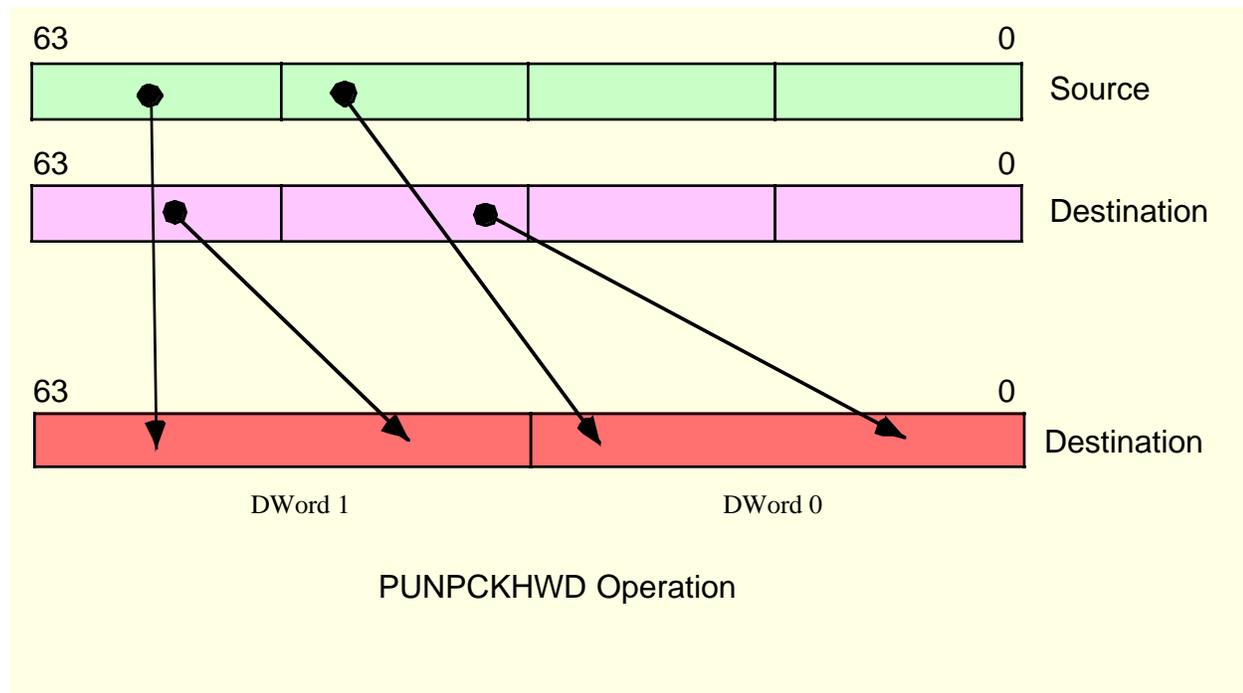


Figure 11.9 PUNPCKHWD Instruction

The PUNPCKHDQ instruction copies the H.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) H.O. dword of the destination operand to the L.O. dword of the destination (see Figure 11.10).

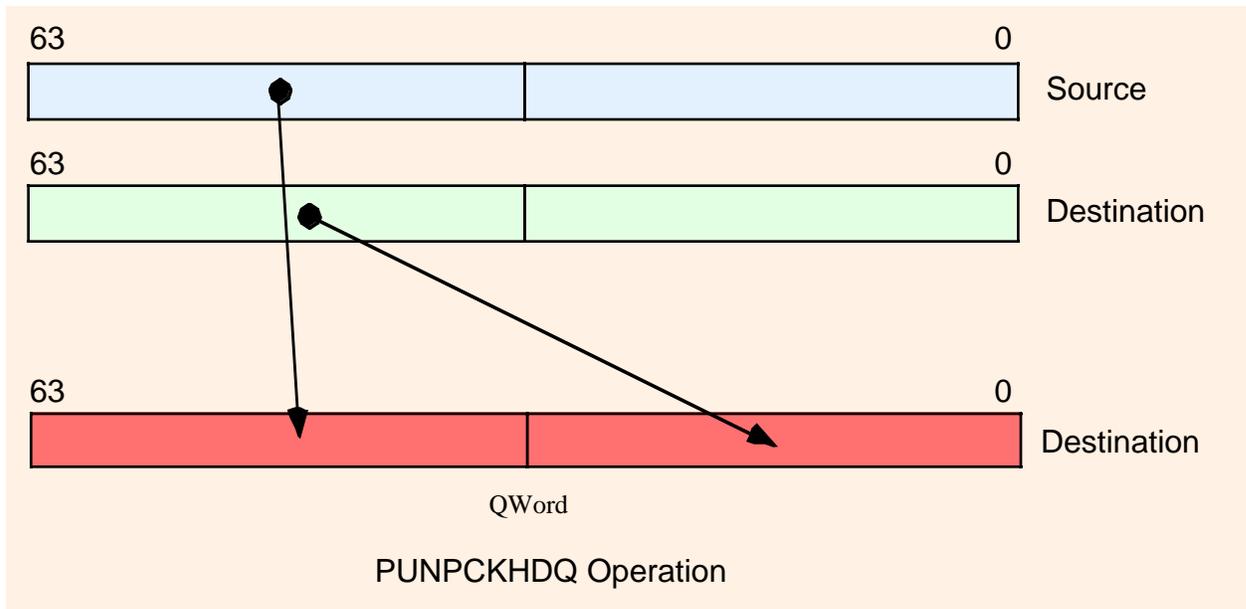


Figure 11.10 PUNPCKDQ Instruction

Since the unpack instructions provide the converse operation of the pack instructions, it should come as no surprise that you can use these instructions to perform the inverse algorithms of the examples given earlier for the pack instructions. For example, if you have a string of eight-bit ANSI characters, you can convert them to their UNICODE equivalents by setting one MMX register (the source) to all zeros. You can convert each four characters of the ANSI string to UNICODE by loading those four characters into the L.O. double word of an MMX register and executing the PUNPCKLBW instruction. This will interleave each of the characters with a zero byte, thus converting them from ANSI to UNICODE.

Of course, the unpack instructions are quite valuable any time you need to interleave data. For example, if you have three separate images containing the blue, red, and green components of a 24-bit image, it is possible to merge these three bytes together using the PUNPCKLBW instruction³.

11.7.3 MMX Packed Arithmetic Instructions

```
paddb( mem64, mmi );
paddb( mmi, mmi );
```

```
paddd( mem64, mmi );
paddd( mmi, mmi );
```

```
paddsb( mem64, mmi );
paddsb( mmi, mmi );
```

```
paddsw( mem64, mmi );
paddsw( mmi, mmi );
```

3. Typically you would merge in a fourth byte of zero and then store the resulting double word every three bytes in memory to overwrite the zeros.

```

paddusb( mem64, mmi );
paddusb( mmi, mmi );

paddusw( mem64, mmi );
paddusw( mmi, mmi );

psubb( mem64, mmi );
psubb( mmi, mmi );

psubw( mem64, mmi );
psubw( mmi, mmi );

psubd( mem64, mmi );
psubd( mmi, mmi );

psubsb( mem64, mmi );
psubsb( mmi, mmi );

psubsw( mem64, mmi );
psubsw( mmi, mmi );

psubusb( mem64, mmi );
psubusb( mmi, mmi );

psubusw( mem64, mmi );
psubusw( mmi, mmi );

pmulhuw( mem64, mmi );
pmulhuw( mmi, mmi );

pmulhw( mem64, mmi );
pmulhw( mmi, mmi );

pmullw( mem64, mmi );
pmullw( mmi, mmi );

pmaddwd( mem64, mmi );
pmaddwd( mmi, mmi );

```

The packed arithmetic instructions operate on a set of bytes, words, or double words within a 64-bit block. For example, the PADDW instruction computes four 16-bit sums of two operand simultaneously. None of these instructions affect the CPU's FLAGS register. Therefore, there is no indication of overflow, underflow, zero result, negative result, etc. If you need to test a result after a packed arithmetic computation, you will need to use one of the packed compare instructions (see "MMX Comparison Instructions" on page 1134).

The PADDB, PADDW, and PADDD instructions add the individual bytes, words, or double words in the two 64-bit operands using a wrap-around (i.e., non-saturating) addition. Any carry out of a sum is lost; it is your responsibility to ensure that overflow never occurs. As for the integer instructions, these packed add instructions add the values in the source operand to the destination operand, leaving the sum in the destination operand. These instructions produce correct results for signed or unsigned operands (assuming overflow/underflow does not occur).

The PADDSB and PADDSW instructions add the eight eight-bit or four 16-bit operands in the source and destination locations together using signed saturation arithmetic. The PADDUSB and PADDUSW instructions add their eight eight-bit or four 16-bit operands together using unsigned saturation arithmetic. Notice that you must use different instructions for signed and unsigned value since saturation arithmetic is different depending upon whether you are manipulating signed or unsigned operands. Also note that the instruction set does not support the saturated addition of double word values.

The PSUBB, PSUBW, and PSUBD instructions work just like their addition counterparts, except of course, they compute the wrap-around difference rather than the sum. These instructions compute `dest=dest-src`. Likewise, the PSUBSB, PSUBSW, PSUBUSB, and PSUBUSW instruction compute the difference of the destination and source operands using saturation arithmetic.

While addition and subtraction can produce a one-bit carry or borrow, multiplication of two n-bit operands can produce as large as a 2*n bit result. Since overflow is far more likely in multiplication than in addition or subtraction, the MMX packed multiply instructions work a little differently than their addition and subtraction counterparts. To successfully multiply two packed values requires two instructions - one to compute the L.O. component of the result and one to produce the H.O. component of the result. The PMULLW, PMULHW, and PMULHUW instructions handle this task.

The PMULLW instruction multiplies the four words of the source operand by the four words of the destination operand and stores the four L.O. words of the four double word results into the destination operand. This instruction ignores the H.O. words of the results. Used by itself, this instruction computes the wrap-around product of an unsigned or signed set of operands; this is also the L.O. words of the four products.

The PMULHW and PMULHUW instructions complete the calculation. After computing the L.O. words of the four products with the PMULLW instruction, you use either the PMULHW or PMULHUW instruction to compute the H.O. words of the products. These two instruction multiply the four words in the source by the four words in the destination and then store the H.O. words of the results in the destination MMX register. The difference between the two is that you use PMULHW for signed operands and PMULHUW for unsigned operands. If you compute the full product by using a PMULLW and a PMULHW (or PMULHUW) instruction pair, then there is no overflow possible, hence you don't have to worry about wrap-around or saturation arithmetic.

The PMADDWD instruction multiplies the four words in the source operand by the four words in the destination operand to produce four double word products. Then it adds the two L.O. double words together and stores the result in the L.O. double word of the destination MMX register; it also adds together the two H.O. double words and stores their sum in the H.O. word of the destination MMX register.

11.7.4 MMX Logic Instructions

```
pand( mem64, mmi );
pand( mmi, mmi );
```

```
pandn( mem64, mmi );
pandn( mmi, mmi );
```

```
por( mem64, mmi );
por( mmi, mmi );
```

```
pxor( mem64, mmi );
pxor( mmi, mmi );
```

The packed logic instructions are some examples of MMX instructions that actually operate on 64-bit values. There are no packed byte, packed word, or packed double word versions of these instructions. Of course, there is no need for special byte, word, or double word versions of these instructions since they would all be equivalent to the 64-bit logic instruction. Hence, if you want to logically AND eight bytes together in parallel, you use the PAND instruction; likewise, if you want to logically AND four words or two double words together, you just use the PAND instruction.

The PAND, POR, and PXOR instructions do the same thing as their 32-bit integer instruction counterparts (AND, OR, XOR) except, of course, they operate on two 64-bit MMX operands. Hence, no further discussion of these instructions is really necessary here. The PANDN (AND NOT) instruction is a new logic instruction, so it bears a little bit of a discussion. The PANDN instruction computes the following result:

```
dest := dest and (not source);
```

As you may recall from the chapter on Introduction to Digital Design, this is the inhibition function. If the destination operand is B and the source operand is A, this function computes $B = BA'$. (see “Boolean Functions and Truth Tables” on page 205 for details of the inhibition function). If you’re wondering why Intel chose to include such a weird function in the MMX instruction set, well, this instruction has one very useful property: it forces bits to zero in the destination operand everywhere there is a one bit in the source operand. This is an extremely useful function for merging to 64-bit quantities together. The following code sequence demonstrates this:

```
readonly
    AlternateNibbles: qword; nostorage;
    qword16( $F0F0_F0F0_F0F0_F0F0 ); // Note: needs qword16 macro!
    .
    .
    .
// Create a 64-bit value in MM0 containing the Odd nibbles from MM1 and
// the even nibbles from MM0:

    pandn( AlternateNibbles, mm0 ); // Clear the odd numbered nibbles.
    pand( AlternateNibbles, mm1 ); // Clear the even numbered nibbles.
    por( mm1, mm0 ); // Merge the two.
```

The PANDN operation is also useful for compute the set difference of two character sets. You could implement the *cs.difference* function using only six MMX instructions:

```
// Compute csdest := csdest - cssrc;

    movq( (type qword csdest), mm0 );
    pandn( (type qword cssrc), mm0 );
    movq( mm0, (type qword csdest) );
    movq( (type qword csdest[8]), mm0 );
    pandn( (type qword cssrc[8]), mm0 );
    movq( mm0, (type qword csdest[8]) );
```

Of course, if you want to improve the performance of the HLA Standard Library character set functions, you can use the MMX logic instructions throughout that module. Examples of such code appear later in this chapter.

11.7.5 MMX Comparison Instructions

```
pcmpeqb( mem64, mmi );
pcmpeqb( mmi, mmi );

pcmpew( mem64, mmi );
pcmpew( mmi, mmi );

pcmpqd( mem64, mmi );
pcmpqd( mmi, mmi );

pcmpgtb( mem64, mmi );
pcmpgtb( mmi, mmi );

pcmpgtw( mem64, mmi );
pcmpgtw( mmi, mmi );

pcmpgtd( mem64, mmi );
pcmpgtd( mmi, mmi );
```

The packed comparison instructions compare the destination (second) operand to the source (first) operand and to test for equality or greater than. These instructions compare eight pairs of bytes (PCMPEQB, PCMPGTB), four pairs of words (PCMPEQW, PCMPGTW), or two pairs of double words (PCMPEQD, PCMPGTD).

The first big difference to notice about these packed comparison instructions is that they compare the second operand to the first operand. This is exactly opposite of the standard CMP instruction (that compares the first operand to the second operand). The reason for this will become clear in a moment; however, you do have to keep in mind when using these instructions that the operands are opposite what you would normally expect. If this ordering bothers you, you can create macros to reverse the operands; we will explore this possibility a little later in this section.

The second big difference between the packed comparisons and the standard integer comparison is that these instructions test for a specific condition (equality or greater than) rather than doing a generic comparison. This is because these instructions, like the other MMX instructions, do not affect any condition code bits in the FLAGS register. This may seem contradictory, after all the whole purpose of the CMP instruction is to set the condition code bits. However, keep in mind that these instructions simultaneously compare two, four, or eight operands; that implies that you would need two, four, or eight sets of condition code bits to hold the results of the comparisons. Since the FLAGS register maintains only one set of condition code bits, it is not possible to reflect the comparison status in the FLAGS. This is why the packed comparison instructions test a specific condition - so they can return true or false to indicate the result of their comparison.

Okay, so where do these instructions return their true or false values? In the destination operand, of course. This is the third big difference between the packed comparisons and the standard integer CMP instruction - the packed comparisons modify their destination operand. Specifically, the PCMPEQB and PCMPGTB instruction compare each pair of bytes in the two operands and write false (\$00) or true (\$FF) to the corresponding byte in the destination operand, depending on the result of the comparison. For example, the instruction “pcmpgtb(MM1, MM0);” compares the L.O. byte of MM0 (A) with the L.O. byte of MM1 (B) and writes \$00 to the L.O. byte of MM0 if A is not greater than B. It writes \$FF to the L.O. byte of MM0 if A is greater than B (see Figure 11.11).

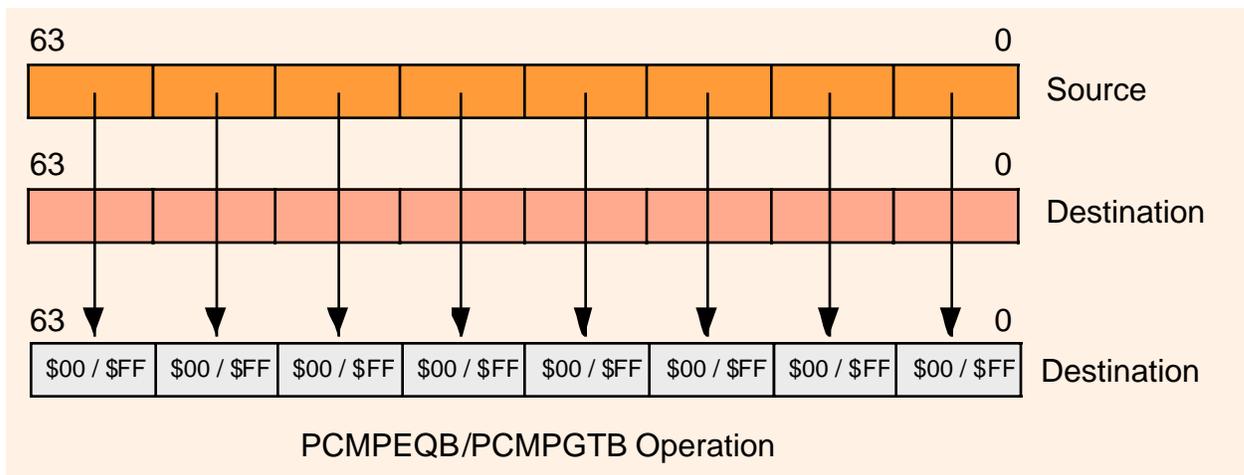


Figure 11.11 PCMPEQB and PCMPGTB Instructions

The PCMPEQW, PCMPGTW, PCMPEQD, and PCMPGTD instructions work in an analogous fashion except, of course, they compare words and double words rather than bytes (see Figure 11.12 and Figure 11.13).

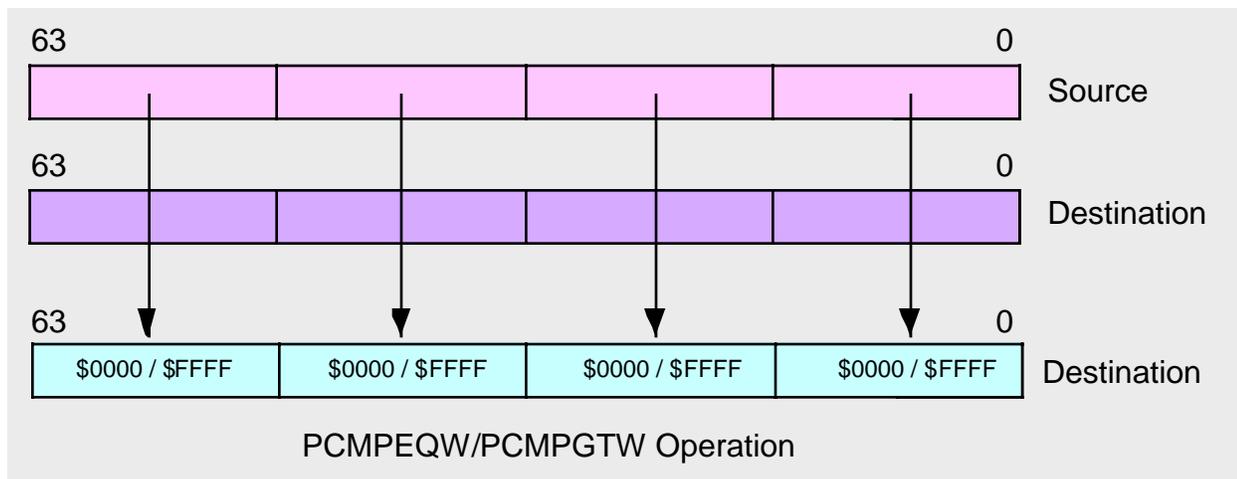


Figure 11.12 PCMPEQW and PCMPGTW Instructions

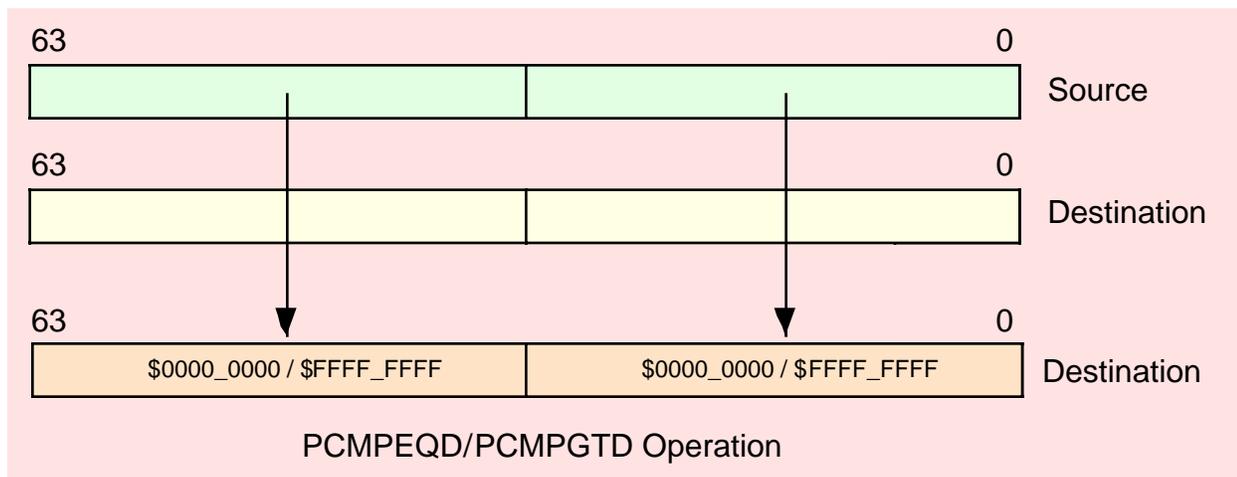


Figure 11.13 PCMPEQD and PCMPGTD Instructions

You’ve probably already noticed that there isn’t a set of PCMPPLTx instructions. Intel chose not to provide these instructions because you can simulate them with the PCMPGTx instructions by reversing the operands. That is, $A > B$ implies $B < A$. Therefore, if you want to do a concurrent comparison of multiple operands for less than, you can use the PCMPGTx instructions to do this by simply reversing the operands. The only time this isn’t directly possible is if your source operand is a memory operand; since the destination operand of the packed comparison instructions has to be an MMX register, you would have to move the memory operand into an MMX register before comparing them.

In addition to the lack of a packed less than comparison, you’re also missing the not equals, less than or equal, and greater than or equal comparisons. You can easily synthesize these comparisons by executing a PXOR or POR instruction after the packed comparison.

To simulate a PCMPNEx instruction, all you’ve got to do is invert all the bits in the destination operand after executing a PCMPEQx instruction, e.g.,

```

pcmpeqb( mm1, mm0 );
pxor( AllOnes, mm0 ); // Assumption: AllOnes is a qword variable
                        // containing $FFFF_FFFF_FFFF_FFFF.

```

Of course, you can save the PXOR instruction by testing for zeros in the destination operand rather than ones (that is, use your program's logic to invert the result rather than actually computing the inverse).

To simulate the PCMPGEx and PCMPLEx instructions, you must do two comparisons, one for equality and one for greater than or less than, and then logically OR the results. Here's an example that computes $MM0 \leq MM1$:

```

movq( mm1, mm2 ); // Need a copy of destination operand.
pcmpgtb( mm0, mm1 ); // Remember: A<B is equal to B>A, so we're
pcmpeqb( mm0, mm2 ); // MM0<MM1 and MM0=MM1 here.
por( mm2, mm1 ); // Leaves boolean results in MM1.

```

If it really bothers you to have to reverse the operands, you can create macros to create your own PCMPLTx instructions. The following example demonstrates how to create the PCMPLTB macro:

```

#macro pcmpltb( mmOp1, mmOp2 );

    pcmpltb( mmOp2, mmOp1 );

#endmacro

```

Of course, you must keep in mind that there are two very big differences between this PCMPLTB "instruction" and a true PCMPLTB instruction. First, this form leaves the result in the first operand, not the second operand, hence the semantics of this "instruction" are different than the other packed comparisons. Second, the first operand has to be an MMX register while the second operand can be an MMX register or a quad word variable; again, just the opposite of the other packed instructions. The fact that this instruction's operands behave differently than the PCMPGTB instruction may create some problems. So you will have to carefully consider whether you really want to use this scheme to create a PCMPLTB "instruction" for use in your programs. If you decide to do this, it would help tremendously if you always commented each invocation of the macro to point out that the first operand is the destination operand, e.g.,

```

pcmpltb( mm0, mm1 ); // Computes mm0 := mm1<mm0!

```

If the fact that the packed comparison instruction's operands are reversed bothers you, you can also use macros to swap those operands. The following example demonstrates how to write such macros for the PEQB (PCMPEQB), PGTB (PCMPGTB), and PLTB (packed less than, byte) instructions.

```

#macro peqb( leftOp, rightOp );

    pcmpeqb( rightOp, leftOp );

#endmacro

#macro pgtb( leftOp, rightOp );

    pcmpltb( rightOp, leftOp );

#endmacro

#macro pltb( leftOp, rightOp );

    pcmpltb( leftOp, rightOp );

#endmacro

```

Note that these macros don't solve the PLTB problem of having the wrong operand as the destination. However, these macros do compare the first operand to the second operand, just like the standard CMP instruction.

Of course, once you obtain a boolean result in an MMX register, you'll probably want to test the results at one point or another. Unfortunately, the MMX instructions only provide a couple of ways to move comparison information in and out of the MMX processor – you can store an MMX register value into memory or you can copy 32-bits of an MMX register to a general-purpose integer register. Since the comparison instructions produce a 64-bit result, writing the destination of a comparison to memory is the easiest way to gain access to the comparison results in your program. Typically, you'd use an instruction sequence like the following:

```
pcmpeqb( mm1, mm0 );           // Compare 8 bytes in mm1 to mm0.
movq( mm0, qwordVar );        // Write comparison results to memory.
if((type boolean qwordVar )) then

    << do this if byte #0 contained true ($FF, which is non-zero). >>

endif;
if((type boolean qwordVar[1])) then

    << do this if byte #1 contained true. >>

endif;
etc.
```

11.7.6 MMX Shift Instructions

```
pshllw( mmi, mmi );
pshllw( imm8, mmi );

pshlld( mmi, mmi );
pshlld( imm8, mmi );

pshllq( mmi, mmi );
pshllq( imm8, mmi );

pshrlw( mmi, mmi );
pshrlw( imm8, mmi );

pshrld( mmi, mmi );
pshrld( imm8, mmi );

pshrq( mmi, mmi );
pshrq( imm8, mmi );

psraw( mmi, mmi );
psraw( imm8, mmi );

psrad( mmi, mmi );
psrad( imm8, mmi );
```

The MMX shift, like the arithmetic instructions, allow you to simultaneously shift several different values in parallel. The PSHLLx instructions perform a packed shift left logical operation, the PSHLRx instructions do a packed logical shift right operation, and the PSRAX instruction do a packed arithmetic shift right operation. These instructions operate on word, double word, and quad word operands. Note that Intel does not provide a version of these instructions that operate on bytes.

The first operand to these instructions specifies a shift count. This should be an unsigned integer value in the range 0..15 for word shifts, 0..31 for double word operands, and 0..63 for quadword operands. If the

shift count is outside these ranges, then these instructions set their destination operands to all zeros. If the count (first) operand is not an immediate constant, then it must be an MMX register.

The PSLW instruction simultaneously shifts the four words in the destination MMX register to the left the number of bit positions specified by the source operand. The instruction shifts zero into the L.O. bit of each word and the bit shifted out of the H.O. bit of each word is lost. There is no carry from one word to the other (since that would imply a larger shift operation). This instruction, like all the other MMX instructions, does not affect the FLAGS register (including the carry flag).

The PSLD instruction simultaneously shifts the two double words in the destination MMX register to the left one bit position. Like the PSLW instruction, this instruction shifts zeros into the L.O. bits and any bits shifted out of the H.O. positions are lost.

The PSLQ is one of the few MMX instructions that operates on 64-bit quantities. This instruction shifts the entire 64-bit destination register to the left the number of bits specified by the count (source) operand. In addition to allowing you to manipulate 64-bit integer quantities, this instruction is especially useful for moving data around in MMX registers so you can pack or unpack data as needed.

Although there is no PSLLB instruction to shift bits, you can simulate this instruction using a PSLW and a PANDN instruction. After shifting the word values to the left the specified number of bits, all you've got to do is clear the L.O. n bits of each byte, where n is the shift count. For example, to shift the bytes in MM0 to the left three positions you could use the following two instructions:

```
static
ThreeBitsZero: byte; @nostorage;
    byte $F8, $F8, $F8, $F8, $F8, $F8, $F8, $F8;
    .
    .
    .
    psllw( 3, mm0 );
    pandn( ThreeBitsZero, mm0 );
```

The PSLRW, PSLRD, and PSLRQ instructions work just like their left shift counterparts except that these instructions shift their operands to the right rather than to the left. They shift zeros into the vacated H.O. positions of the destination values and bits they shift out of the L.O. bits are lost. As with the shift left instructions, there is no PSLRB instruction but you can easily simulate this with a PSLRW and a PANDN instruction.

The PSRAW and PSRAD instructions do an arithmetic shift right operation on the words or double words in the destination MMX register. Note that there isn't a PSRAQ instruction. While shifting data to the right, these instructions replicate the H.O. bit of each word, double word, or quad word rather than shifting in zeros. As for the logical shift right instructions, bits that these instructions shift out of the L.O. bits are lost forever.

The PSLLQ and PSLRQ instructions provide a convenient way to shift a quad word to the left or right. However, the MMX shift instructions are not generally useful for extended precision shifts since all data shifted out of the operands is lost. If you need to do an extended precision shift other than 64 bits, you should stick with the SHLD and SHRD instructions. The MMX shift instructions are mainly useful for shifting several values in parallel or (PSLLQ and PSLRQ) repositioning data in an MMX register.

11.8 The EMMS Instruction

```
emms();
```

The EMMS (Empty MMX Machine State) instruction restores the FPU status on the CPU so that it can begin processing FPU instructions again after an MMX instruction sequence. You should always execute the EMMS instruction once you complete some MMX sequence. Failure to do so may cause any following floating point instructions to fail.

When an MMX instruction executes, the floating point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating point tag word as empty. This must occur before the execution of any following floating point instructions.

Of course, you don't have to execute the EMMS instruction immediately after an MMX sequence if you're going to execute some additional MMX instructions prior to executing any FPU instructions, but you must take care to execute this instruction if

- You call any library routines or OS APIs (that might possibly use the FPU).
- You switch tasks in a cooperative fashion (for example, see the chapter on Coroutines in the Volume on Advanced Procedures).
- You execute any FPU instructions.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating point exception event may be generated.
- A "soft exception" may occur. In this case floating-point code continues to execute, but generates incorrect results.

The EMMS instruction is rather slow, so you don't want to unnecessarily execute it, but it is critical that you execute it at the appropriate times. Of course, better safe than sorry; if you're not sure you're going to execute more MMX instructions before any FPU instructions, then go ahead and execute the EMMS instruction to clear the state.

11.9 The MMX Programming Paradigm

In general, you don't learn scalar (non-MMX) 80x86 assembly language programming and then use that same mindset when writing programs using the MMX instruction set. While it is possible to directly use various MMX instructions the same way you would the general purpose integer instructions, one phrase comes to mind when working with MMX: think parallel. This text has spent many hundreds of pages up to this point attempting to get you to think in assembly language; to think that this small section can teach you how to design optimal MMX sequence would be ludicrous. Nonetheless, a few simple examples are useful to help start you thinking about how to use the MMX instructions to your benefit in your programs. This section will begin by presenting some fairly obvious uses for the MMX instruction set, and then it will attempt to present some examples that exploit the inherent parallelism of the MMX instructions.

Since the MMX registers are 64-bits wide, you can double the speed of certain data movement operations by using MMX registers rather than the 32-bit general purpose registers. For example, consider the following code from the HLA Standard Library that copies one character set object to another:

```
procedure cs.cpy( src:cset; var dest:cset ); nodisplay;
begin cpy;

    push( eax );
    push( ebx );
    mov( dest, ebx );
    mov( (type dword src), eax );
    mov( eax, [ebx] );
    mov( (type dword src[4]), eax );
    mov( eax, [ebx+4] );
    mov( (type dword src[8]), eax );
    mov( eax, [ebx+8] );
```

```

mov( (type dword src[12]), eax );
mov( eax, [ebx+12] );
pop( ebx );
pop( eax );

end cpy;

```

Program 11.2 HLA Standard Library *cs.cpy* Routine

This is a relatively simple code sequence. Indeed, a fair amount of the execution time is spent copying the parameters (20 bytes) onto the stack, calling the routine, and returning from the routine. This entire sequence can be reduced to the following four MMX instructions:

```

movq( (type qword src), mm0 );
movq( (type qword src[8]), mm1 );
movq( mm0, (type qword dest));
movq( mm1, (type qword dest[8]));

```

Of course, this sequence assumes two things: (1) it's okay to wipe out the values in MM0 and MM1, and (2) you'll execute the EMMS instruction a little later on after the execution of some other MMX instructions. If either, or both, of these assumptions is incorrect, the performance of this sequence won't be quite as good (though probably still better than the *cs.cpy* routine). However, if these two assumptions do hold, then it's relatively easy to implement the *cs.cpy* routine as an in-line function (i.e., a macro) and have it run much faster. If you really need this operation to occur inside a procedure and you need to preserve the MMX registers, and you don't know if any MMX instructions will execute shortly thereafter (i.e., you'll need to execute EMMS), then it's doubtful that using the MMX instructions will help here. However, in those cases when you can put the code in-line, using the MMX instructions will be faster.

Warning: don't get too carried away with the MMX MOVQ instruction. Several programmers have gone to great extremes to use this instruction as part of a high performance MOVSD replacement. However, except in very special cases on very well designed systems, the limiting factor for a block move is the speed of memory. Since Intel has optimized the operation of the MOVSD instruction, you're best off using the MOVSD instructions when moving blocks of memory around.

Earlier, this chapter used the *cs.difference* function as an example when discussing the PANDN instruction. Here's the original HLA Standard Library implementation of this function:

```

procedure cs.difference( src:cset; var dest:cset ); nodisplay;
begin difference;

    push( eax );
    push( ebx );
    mov( dest, ebx );
    mov( (type dword src), eax );
    not( eax );
    and( eax, [ebx] );
    mov( (type dword src[4]), eax );
    not( eax );
    and( eax, [ebx+4] );
    mov( (type dword src[8]), eax );
    not( eax );
    and( eax, [ebx+8] );
    mov( (type dword src[12]), eax );
    not( eax );
    and( eax, [ebx+12] );
    pop( ebx );

```

```

    pop( eax );

end difference;

```

Program 11.3 HLA Standard Library *cs.difference* Routine

Once again, the high-level nature of HLA is hiding the fact that calling this function is somewhat expensive. A typical call to *cs.difference* emits five or more instructions just to push the parameters (it takes four 32-bit PUSH instructions to pass the *src* character set because it is a value parameter). If you're willing to wipe out the values in MM0 and MM1, and you don't need to execute an EMMS instruction right away, it's possible to compute the set difference with only six instructions – that's about the same number of instructions (and often fewer) than are needed to call this routine, much less do the actual work. Here are those six instructions:

```

    movq( dest, mm0 );
    movq( dest[8], mm1 );
    pandn( src, mm0 );
    pandn( src[8], mm1 );
    movq( mm0, dest );
    movq( mm1, dest[8] );

```

These six instructions replace 12 of the instructions in the body of the function. The sequence is sufficiently short that it's reasonable to code it in-line rather than in a function. However, were you to bury this code in the *cs.difference* routine, you needed to preserve MM0 and MM1⁴, and you needed to execute EMMS afterwards, this would cost more than it's worth. As an in-line macro, however, it is going to be significantly faster since it avoids passing parameters and the call/return sequence.

If you want to compute the intersection of two character sets, the instruction sequence is identical to the above except you substitute PAND for PANDN. Similarly, if you want to compute the union of two character sets, use the code sequence above substituting POR for PANDN. Again, both approaches pay off handsomely if you insert the code in-line rather than burying it in a procedure and you don't need to preserve MMX registers or execute EMMS afterwards.

We can continue with this exercise of working our way through the HLA Standard Library character set (and other) routines substituting MMX instructions in place of standard integer instructions. As long as we don't need to preserve the MMX machine state (i.e., registers) and we don't have to execute EMMS, most of the character set operations will be short enough to code in-line. Unfortunately, we're not buying that much over code the standard implementations of these functions in-line from a performance point of view (though the code would be quite a bit shorter). The problem here is that we're not "thinking in MMX." We're still thinking in scalar (non-parallel mode) and the fact that the MMX instruction set requires a lot of set-up (well, "tear-down" actually) negates many of the advantages of using MMX instructions in our programs.

The MMX instructions are most appropriate when you compute multiple results in parallel. The problem with the character set examples above is that we're not even processing a whole data object with a single instruction; we're actually only processing a half of a character set with a sequence of three MMX instructions (i.e., it requires six instructions to compute the intersection, union, or difference of two character sets). At best, we can only expect the code to run about twice as fast since we're processing 64 bits at a time instead of 32 bits. Executing EMMS (and, God help us, having to preserve MMX registers) negates much of what we might gain by using the MMX instructions. Again, we're only going to see a speed improvement if we process multiple objects with a single MMX instruction. We're not going to do that manipulating large objects like character sets.

One data type that will let us easily manipulate up to eight objects at one time is a character string. We can speed up many character string operations by operating on eight characters in the string at one time. Consider the HLA Standard Library *str.uppercase* procedure. This function steps through each character of

4. Actually, the code could be rewritten easily enough to use only one MMX register.

a string, tests to see if it's a lower case character, and if so, converts the lower case character to upper case. A good question to ask is "can we process eight characters at a time using the MMX instructions?" The answer turns out to be yes and the MMX implementation of this function provides an interesting perspective on writing MMX code.

At first glance it might seem impractical to use the MMX instructions to test for lower case characters and convert them to upper case. Consider the typical scalar approach that tests and converts a single character at a time:

```
<< Get character to convert into the AL register >>

    cmp( al, 'a' );
    jb noConversion;
    cmp( al, 'z' );
    ja noConversion;
    sub( $20, al );    // Could also use AND($5f, al); here.
noConversion:
```

This code first checks the value in AL to see if it's actually a lower case character (that's the CMP and Jcc instructions in the code above). If the character is outside the range 'a'..'z' then this code skips over the conversion (the SUB instruction); however, if the code is in the specified range, then the sequence above drops through to the SUB instruction and converts the lower case character to upper case by subtracting \$20 from the lower case character's ASCII code (since lower case characters always have bit #5 set, subtracting \$20 always clears this bit).

Any attempt to convert this code directly to an MMX sequence is going to fail. Comparing and branching around the conversion instruction only works if you're converting one value at a time. When operating on eight characters simultaneously, any mixture of the eight characters may or may not require conversion from lower case to upper case. Hence, we need to be able to perform some calculation that is benign if the character is not lower case (i.e., doesn't affect the character's value) while converting the character to upper case if it was lower case to begin with. Worse, we have to do this with pure computation since flow of control isn't going to be particularly effective here (if we test each individual result in our MMX register we won't really save anything over the scalar approach). To save you some suspense, yes, such a calculation does exist.

Consider the following algorithm that converts lower case characters to upper case:

```
<< Get character to test into AL >>
    cmp( al, 'a' );
    setae( bl );    // bl := al >= 'a'
    cmp( al, 'z' );
    setbe( bh );    // bh := al <= 'z'
    and( bh, bl );  // bl := (al >= 'a') && (al <= 'z' );
    dec( bl );      // bl := $FF/$00 if false/true.
    not( bl );      // bl := $FF/$00 if true/false.
    and( $20, bl ); // bl := $20/$00 if true/false.
    sub( bl, al );  // subtract $20 if al was lowercase.
```

This code sequence is fairly straight-forward up until the DEC instruction above. It computes true/false in BL depending on whether AL is in the range 'a'..'z'. At the point of the DEC instruction, BL contains one if AL is a lower case character, it contains zero if AL's value is not lower case. After the DEC instruction, BL contains \$FF for false (AL is not lower case) and \$00 for true (AL is lowercase). The code is going to use this as a mask a little later, but it really needs true to be \$FF and false \$00, hence the NOT instruction that follows. The (second) AND instruction above converts true to \$20 and false to \$00 and the final SUB instruction subtracts \$20 if AL contained lower case, it subtracts \$00 from AL if AL did not contain a lower case character (subtracting \$20 from a lower case character will convert it to upper case).

Whew! This sequence probably isn't very efficient when compared to the simpler code given previously. Certainly there are more instructions in this version (nearly twice as many). Whether this code without any branches runs faster or slower than the earlier code with two branches is a good question. The important thing to note here, though, is that we converted the lower case characters to upper case (leaving

other characters unchanged) using only a calculation; no program flow logic is necessary. This means that the code sequence above is a good candidate for conversion to MMX. Even if the code sequence above is slower than the previous algorithm when converting one character at a time to upper case, it's positively going to scream when it converts eight characters at a shot (since you'll only need to execute the sequence one-eighth as many times).

The following is the code sequence that will convert the eight characters starting at location [EDI] in memory to upper case:

```
static
  A:qword; @nostorage;
    byte $60, $60, $60, $60, $60, $60, $60, $60; // Note: $60 = 'a'-1.
  Z:qword; @nostorage;
    byte $7B, $7B, $7B, $7B, $7B, $7B, $7B, $7B; // Note: $7B = 'z' + 1.
  ConvFactor:qword; @nostorage;
    byte $20, $20, $20, $20, $20, $20, $20, $20; // Magic value for lc->UC.
    .
    .
    .
  movq( ConvFactor, mm4 ); // Eight copies of conversion value.
  movq( A, mm2 ); // Put eight "a" characters in mm2.
  movq( Z, mm3 ); // Put eight "z" characters in mm3.
  movq( [edi], mm0 ); // Get next eight characters of our string.
  movq( mm0, mm1 ); // We need two copies.
  pcmptgb( mm2, mm1 ); // Generate 1's in MM1 everywhere chars >= 'a'
  pcmptgb( mm0, mm3 ); // Generate 1's in MM3 everywhere chars <= 'z'
  pand( mm3, mm1 ); // Generate 1's in MM1 when 'a'<=chars<='z'
  pand( mm4, mm1 ); // Generates $20 in each spot we have a l.c. char
  psubb( mm1, mm0 ); // Convert l.c. chars to U.C. by adding $20.
  movq( mm0, [edi]);
```

Note that this code compares the characters that [EDI] points at to 'a'-1 and 'z'+1 because we only have a greater than comparison rather than a greater or equal comparison (this saves a few extra instructions). Other than setting up the MMX registers and taking advantage of the fact that the PCMPGTB instructions automatically produce \$FF for true and \$00 for false, this is a faithful reproduction of the previous algorithm except it operates on eight bytes simultaneously. So if we put this code in a loop and execute it once for each eight characters in the string, there will be one-eighth the iterations of a similar loop using the scalar instructions.

Of course, there is one problem with this code. Not all strings have lengths that are an even multiple of eight bytes. Therefore, we've got to put some special case code into our algorithm to handle strings that are less than eight characters long and handle strings whose length is not an even multiple of eight characters. In the following program, the *mmxupper* function simply borrows the scalar code from the HLA Standard Library's *str.upper* procedure to handle the leftover characters. The following example program provides both an MMX and a scalar solution with a main program that compares the running time of both. If you're wondering, the MMX version is about three times faster (on a Pentium III) for strings around 35 characters long, containing mostly lower case (mostly lower case favors the scalar algorithm since fewer branches are taken with lower case characters; longer strings favor the MMX algorithm since it spends more time in the MMX code compared to the scalar code at the end).

```
program UpperCase;
#include( "stdlib.hhf" )

// The following code was stolen from the
// HLA Standard Library's str.upper function.
// It is not optimized, but then none of this
// code is optimized other than to use the MMX
// instruction set (later).
```

```

procedure strupper( dest: string ); @nodisplay;
begin strupper;

    push( edi );
    push( eax );

    mov( dest, edi );
    if( edi = 0 ) then

        raise( ex.AttemptToDerefNULL );

    endif;

    // Until we encounter a zero byte, convert any lower
    // case characters to upper case.

    forever

        mov( [edi], al );
        breakif( al = 0 );      // Quit when we find a zero byte.

        // If a lower case character, convert it to upper case
        // and store the result back into the destination string.

        if
        (#{
            cmp( al, 'a' );
            jb false;
            cmp( al, 'z' );
            ja false;
        }#) then

            and( $5f, al );      // Magic lc->UC translation.
            mov( al, [edi] );    // Save result.

        endif;

        // Move on to the next character.

        inc( edi );

    endfor;

    pop( edi );
    pop( eax );

end strupper;

```

```

procedure mmxupper( dest: string ); @nodisplay;
const
    zCh:char := char( uns8( 'z' ) + 1 );
    aCh:char := char( uns8( 'a' ) - 1 );

static

    // Create eight copies of the A-1 and Z+1 characters
    // so we can compare eight characters at once:

```

```

A:qword; @nostorage;
    byte aCh, aCh, aCh, aCh, aCh, aCh, aCh, aCh;

Z:qword; @nostorage;
    byte zCh, zCh, zCh, zCh, zCh, zCh, zCh, zCh;

// Conversion factor: UC := LC - $20.

ConvFactor: qword; @nostorage;
    byte $20, $20, $20, $20, $20, $20, $20, $20;

begin mxupper;

    push( edi );
    push( eax );

    mov( dest, edi );
    if( edi = 0 ) then

        raise( ex.AttemptToDerefNULL );

    endif;

    // Some invariant operations (things that don't
    // change on each iteration of the loop):

    movq( A, mm2 );
    movq( ConvFactor, mm4 );

    // Get the string length from the length field:

    mov( (type str.strRec [edi]).length, eax );

    // Process the string in blocks of eight characters:

    while( (type int32 eax) >= 8 ) do

        movq( [edi], mm0 ); // Get next eight characters of our string.
        movq( mm0, mm1 ); // We need two copies.
        movq( Z, mm3 ); // Need to refresh on each loop.
        pcmprgtb( mm2, mm1 ); // Generate 1's in MM1 everywhere chars >= 'a'
        pcmprgtb( mm0, mm3 ); // Generate 1's in MM3 everywhere chars <= 'z'
        pand( mm3, mm1 ); // Generate 1's in MM1 when 'a'<=chars<='z'
        pand( mm4, mm1 ); // Generates $20 in each spot we have a l.c. char
        psubb( mm1, mm0 ); // Convert l.c. chars to U.C. by adding $20.
        movq( mm0, (type qword [edi]));

        // Move on to the next eight characters in the string.

        sub( 8, eax );
        add( 8, edi );

    endwhile;

    // If we're processing less than eight characters, do it the old-fashioned
    // way (one character at a time). This also handles the last 1..7 chars
    // if the number of characters is not an even multiple of eight. This
    // code was swiped directly from the HLA str.upper function (above).

    if( eax != 0 ) then

```

```

    forever

        mov( [edi], al );
        breakif( al = 0 );      // Quit when we find a zero byte.

        // If a lower case character, convert it to upper case
        // and store the result back into the destination string.

        if
        (#{
            cmp( al, 'a' );
            jb false;
            cmp( al, 'z' );
            ja false;
        }#) then

            and( $5f, al );      // Magic lc->UC translation.
            mov( al, [edi] );    // Save result.

        endif;

        // Move on to the next character.

        inc( edi );

    endfor;

endif;
emms(); // Clean up MMX state.

pop( edi );
pop( eax );

end mmxupper;

static
    MyStr: string := "Hello There, MMX Uppercase Routine!";
    destStr:string;
    mmxCycles:qword;
    strCycles:qword;

begin UpperCase;

    // Charge up the cache (prefetch the code and data
    // to avoid cache misses later).

    mov( str.a_cpy( MyStr ), destStr );
    mmxupper( destStr );
    strupper( destStr );

    // Okay, time the execution of the MMX version:

    mov( str.a_cpy( MyStr ), destStr );

    rdtsc();
    mov( eax, (type dword mmxCycles));

```

```

mov( edx, (type dword mmxCycles[4]));
mmxupper( destStr );
rdtsc();
sub( (type dword mmxCycles), eax );
sbb( (type dword mmxCycles[4]), edx );
mov( eax, (type dword mmxCycles));
mov( edx, (type dword mmxCycles[4]));

stdout.put( "Dest String = `", destStr, "`", nl );

// Okay, time the execution of the HLA version:

mov( str.a_cpy( MyStr ), destStr );

rdtsc();
mov( eax, (type dword strCycles));
mov( edx, (type dword strCycles[4]));
strupper( destStr );
rdtsc();
sub( (type dword strCycles), eax );
sbb( (type dword strCycles[4]), edx );
mov( eax, (type dword strCycles));
mov( edx, (type dword strCycles[4]));

stdout.put( "Dest String(2) = `", destStr, "`", nl );

stdout.put( "MMX cycles:" );
stdout.puti64( mmxCycles );
stdout.put( nl "HLA cycles: " );
stdout.puti64( strCycles );
stdout.newln();

end UpperCase;

```

Program 11.4 MMX Implementation of the HLA Standard Library str.upper Procedure

Other string functions, like a case insensitive string comparison, can greatly benefit from the use of parallel computation via the MMX instruction set. Implementation of other string functions is left as an exercise to the reader; interested readers should consider converting string functions that involve calculations and tests on each individual characters in a string as candidates for optimization via MMX.

11.10 Putting It All Together

Intel's MMX enhancements to the basic Pentium instruction set allow the acceleration of certain algorithms. Unfortunately, the MMX instruction set isn't generally applicable to a wide range of problems. The MMX instructions, with their SIMD orientation, are generally useful for manipulating a large amount of data organized as byte, word, or double word arrays where the MMX instructions can calculate several values in parallel. Learning to effectively use the MMX instruction set requires a paradigm shift on the part of the programmer. You don't apply the same rules for scalar 80x86 instructions to the MMX instructions. However, if you take the time to master parallel programming techniques with the MMX instructions, then you will be able to accelerate many of your applications.

