Chapter 11: Decimal Arithmetic

Chapter 11 describes BCD representation schemes and the 80x86 instructions that are used with BCD numbers. It includes code to convert BCD representations for numbers to and from corresponding ASCII representations and some procedures for BCD arithmetic.

The two major classifications of BCD schemes are packed and unpacked, and many variations with respect to the number of bytes used and how the sign of a value is represented. This section and Section 11.2 discuss packed BCD numbers. Section 11.3 tells about unpacked BCD numbers.

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

Packed BCD representations store two decimal digits per byte, one in the high-order four bits and one in the low-order four bits.

For example, the bit pattern 01101001 represents the decimal number 69, using 0110 for 6 and 1001 for 9. One confusing thing about packed BCD is that this same bit pattern is written 69 in hexadecimal.

however, this just means that if 01101001 is thought of as a BCD number, it represents the decimal value 69, but if it is viewed as a signed or unsigned binary integer, the corresponding decimal value is 105.

This again makes the point that a given pattern of bits can have multiple numeric interpretations, as well as nonnumeric meanings.

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

If single bytes were used for packed BCD representations, then decimal numbers from 0 to 99 could be stored. This would not be very useful, so typically several bytes are used to store a single number. Many schemes are possible; some use a fixed number of bytes and some have variable length, incorporating a field for length as part of the representation.

The bit pattern for a number often includes one or more bits to indicate the sign of the number.

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

As mentioned in Chapter 10, the Microsoft Macro Assembler provides a DT directive that can be used to define a 10 byte packed decimal number. Although other representation systems are equally valid, this book concentrates on this scheme.

The directive

DT 123456789

reserves ten bytes of storage with initial values (in hex)

89 67 45 23 01 00 00 00 00 00

11.1 Packed BCD Representations

Notice that only the sign indicator changes for a negative number; other digits of the representation are the same as they would be for the corresponding positive number.

Since an entire byte is used for the sign indicator, only nine bytes remain to store decimal digits. Therefore the packed BCD scheme used by the DT directive stores a signed number up to decimal 18 digits long. With MASM 6.11, extra digits are truncated without warning.

Although DT directives can be used to initialize packed BCD numbers in an assembly language program and arithmetic can be done on these numbers with the aid of the instructions covered in the next section, packed BCD numbers are of little service unless they can be displayed for human use. Figure 11.1 gives the source code for a procedure *ptoaProc* that converts a packed BCD number to the corresponding ASCII string. This procedure does the same job for packed BCD numbers as *itoaProc* and *dtoaProc* do for 2's complement integers.

Chapter 11: Decimal Arithmetic

11.1 Packed BCD Representations

```
ptoaProc  PROC      NEAR      ;convert  10-byte  BCD  number  to  a  19-byte-long  ASCII
string                        ;parameter 1: address of BCD number
                              ;parameter 2: destination address
                              ;author: R. Detmer revised: 5/98 push ebp
                              ;establish stack frame mov ebp, esp push esi
                              ;save registers
          push      edi
          push      eax
          push      ecx
          mov       esi,[ebp+12]        ;source address
          mov       edi,[ebp+8]         ;destination address
          add       edi,18              ;point to last byte of destination
          mov       ecx,9               ;count of bytes to process
for1:     mov       al,[si]             ;byte with two bcd digits
          mov       ah,al               ;copy to high-order byte of AX
          and       al,00001111b        ;mask out higher-order digit
          or        al,30h              ;convert to ASCII character
          mov       [edi],al            ;save lower-order digit
          dec       edi                 ;point at next destination byte to left
```

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

```
          shr       ah,4                  ;shift out lower-order digit
          or        ah,30h                ;convert to ASCII
          mov       [edi],ah              ;save higher-order digit
          dec       edi                   ;point at next destination byte to left
          inc       esi                   ;point at next source byte
          loop      for1                  ;continue for 9 bytes
          mov       BYTE PTR [edi],' `    ;space for positive number
          and       BYTE PTR [esi],80h    ;check sign byte
          jz        nonNeg                ;skip if not negative
          mov       BYTE PTR [edi], '-'   ; minus sign
nonNeg:   pop       ecx                   ;restore registers
          pop       eax
          pop       esi
          pop       edi
          pop       ebp
          ret       8                     ;return, removing parameters
ptoaProc  ENDP
```

Figure 11.1: Packed BCD to ASCII conversion

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

The procedure ***ptoaProc*** has two parameters:

a 10-byte-long packed BCD source

and a 19-byte-long ASCII destination string,

each passed by location.

The destination is 19 bytes long to allow for a sign and 18 digits. The sign will be a space for a positive number and a minus sign for a negative number.

For the digits, leading zeros rather than spaces are produced. The procedure implements the following design:

11.1 Packed BCD Representations

The procedure implements the following design:

```
copy source address to ESI;
copy destination address to EDI;
add 18 to EDI to point at last byte of destination string;
for count := 9 down to 1 loop { process byte containing two digits }
           copy next source byte to AL;
           duplicate source byte in AH;
           mask out high-order digit in AL;
           convert low-order digit in AL to ASCII code;
           store ASCII code for low-order digit in destination string;
           decrement EDI to point at next destination byte to left;
           shift AH 4 bits to right to get only high-order digit;
           convert high-order digit in AH to ASCII code;
           store ASCII code for high-order digit in destination string;
           decrement EDI to point at next destination byte to left;
           increment ESI to point at next source digit to right;
end for;
move space to first byte of destination string;
if source number is negative
then
           move minus sign to first byte of destination string;
end if;
```

Chapter 11: Decimal Arithmetic

11.1 Packed BCD Representations

Sometimes it is necessary to convert an ASCII string to a corresponding packed BCD value. Figure 11.2 shows a procedure *atopProc* that accomplishes this task in a restricted setting.

The procedure has two parameters, the addresses of an ASCII source string and a 10 byte BCD destination string. The ASCII source string is very limited. It can consist only of ASCII codes for digits terminated by a null byte; no sign, no space, nor any other character code is permitted.

```
atopProc PROC NEAR32
;Convert ASCII string at to 10-byte BCD number
;parameter 1: ASCII string address parameter 2: BCD number address
;null-terminated source string consists only of ASCII codes for digits,
;author: R. Detmer revised: 5/98
         push      ebp                  ;establish stack frame
         mov       ebp,esp
         push      esi                  ;save registers
         push      edi
         push      eax
         push      ecx
```

Chapter 11: Decimal Arithmetic

11.1 Packed BCD Representations

```
          mov       esi,[ebp+12]        ;source address
          mov        edi,[ebp+8]        ;destination address
          mov       DWORD PTR [edi],0   ;zero BCD destination
          mov       DWORD PTR [edi+4],0
          mov       WORD PTR [edi+8], 0
; find length of source string and move ESI to trailing null
          mov       ecx,0               ;count := 0
While1:   cmp       BYTE PTR [esi],0    ;while not end of string (null)
          jz        endwhile1
          inc       ecx                 ;add 1 to count of characters
          inc       esi                 ;point at next character
          jmp       while1              ;check again
endwhile1:                              ;process source characters a pair at a time
While2:   cmp       ecx,0               ;while count > 0
          jz        endwhile2
          dec       esi                 ;point at next ASCII byte from right
          mov       al,BYTE PTR [esi]   ;get byte
          and       al,00001111b        ;convert to BCD digit
          mov       BYTE PTR [edi],al   ;save BCD digit
          dec ecx                       ;decrement count
```

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

```
          jz        endwhile2         ;exit loop if out of source digits
          dec       esi               ;point at next ASCII byte from right
          mov       al,BYTE PTR [esi] ;get byte
          shl       al,4              ;shift to left and convert to digit
          or        BYTE PTR [edi],al ;combine with other BCD digit
          dec       ecx               ;decrement count
          inc       edi               ;point at next destination byte
          jmp       while2            ;repeat for all source characters
Endwhile2:pop       ecx               ;restore registers
          pop       eax
          pop       esi
          pop       edi
          pop       ebp
          ret       8                 ;return, removing parameters
          atopProc  ENDP
```

Figure 11.2: ASCII to packed BCD conversion

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

The design of procedure *atopProc* is quite different from *atodProc* (Fig. 8.9) that produces a doubleword integer from an ASCII string.

The ASCII-to-doubleword routine scans source characters left to right one at a time, but the ASCII-to-packed BCD procedure scans the source string right to left, two characters at a time, in order to pack two decimal digits into one byte.

The procedure must begin by locating the right end of the string. If there is an odd number of source characters, then only one character will contribute to the last BCD byte. The design for *atopProc* appears below.

Chapter 11: Decimal Arithmetic
11.1 Packed BCD Representations

The design for *atopProc* appears below.

```
copy source address to ESI;
copy destination address to EDI;
initialize all 10 bytes of destination, each to 00;
counter := 0;
while ESI is not pointing at trailing null byte of ASCII source loop
          add 1 to counter;
          increment ESI to point at next byte of source string;
end while;
while counter > 0 loop
          decrement ESI to point at next source byte from right;
          copy source byte to AL;
          convert ASCII code to digit by zeroing leftmost 4 bits;
          save low-order digit in destination string;
          subtract 1 from counter;
          if counter = 0
          then
                    exit loop;
          end if;
          decrement ESI to point at next source byte from right;
          copy source byte to AL;
          shift AL 4 bits left to get digit in high order 4 bits;
          or AL with destination byte to combine with low-order digit;
          subtract 1 from counter;
          increment EDI to point at next destination byte;
end while;
```

Chapter 11: Decimal Arithmetic

11.1 Packed BCD Representations

The first *while* loop in the design simply scans the source string left to right, counting digits preceding the trailing null byte. Although this design allows only ASCII codes for digits, an extra loop could be included to skip leading blanks and a leading minus or plus (− or +) could be noted.

The second *while* loop processes the ASCII codes for digits that have been counted in the first loop. Two digits, if available, must be packed into a single destination byte. At least one source byte is there each time through the loop, so the first is loaded into AL, changed from an ASCII code to a digit, and stored in the destination string. (An alternative way to convert the ASCII code to a digit would be to subtract 30_{16}.) If source characters are exhausted, then the *while* loop is exited. Otherwise a second ASCII character is loaded into AL, a left shift instruction converts it to a digit in the left four bits of AL, and an or combines it with the right digit already stored in memory in the destination string.

The *atopProc* procedure could be used to convert a string obtained from the *input* macro. If some other method were used, one would have to ensure that the string has a trailing null byte.

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

Addition and subtraction operations for packed BCD numbers are similar to those for multicomponent 2's complement numbers (Section 4.5). Corresponding bytes of the two operands are added, and the carry-from-one addition is added to the next pair of bytes.

BCD operands have no special addition instruction;
the regular add and adc instructions are used.

However, these instructions are designed for binary values, not BCD values, so for many operands they give the wrong sums.

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

The 80x86 architecture includes a

daa (decimal adjust after addition) instruction

used after an addition instruction to correct the sum. This section explains the operation of the daa instruction and its counterpart das for subtraction.

Procedures for addition and subtraction of non-negative 10-byte packed BCD numbers are developed; then a general addition procedure is given.

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

A few examples illustrate the problem with using binary addition for BCD operands. The AF column gives the value of the auxiliary carry flag, the significance of which is discussed below.

Although each answer is correct as the sum of two unsigned binary integers, only the first result is correct as a BCD value. The second and third sums contain bit patterns that are not used in BCD representations, C_{16} in the second example and B_{16} in the third. The last two sums contain no invalid digit-they are simply wrong as decimal sums.

Before		After add al,bl		
AL	BL	AL	AF	CF
34	25	59	0	0
37	25	5C	0	0
93	25	B8	0	0
28	39	61	1	0
79	99	12	1	1

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

The daa instruction is used after an addition instruction to convert a binary sum into a packed BCD sum. The instruction has no operand; the sum to be converted must be in the AL register.

A daa instruction examines and sets both the carry flag CF and the auxiliary carry flag AF (bit 4 of the EFLAGS register). Recall that the carry flag is set to 1 during addition of two eight bit numbers if there is a carry out of the leftmost position. The AF flag similarly is set to 1 by add or adc instructions if there is a carry resulting from addition of the low-order four bits of the two operands. One way of thinking of this is that the sum of the two low-order hex digits is greater than F_{16}.

11.2 Packed BCD Instructions

A daa instruction first examines the right hex digit of the binary sum in AL. If this digit is over 9 (that is, A through F), then 6 is added to the entire sum and AF is set to 1. Notice that this would correct the result in the second example above since 5C + 6 = 62, the correct packed BCD sum of 37 and 25. The same correction is applied if AF=1 when the daa instruction is executed. Thus in the fourth example, 61 + 6 = 67.

After correcting the right digit, daa examines the left digit in AL. The action is similar: If the left digit is over 9 or CF=1, then 60_{16} is added to the entire sum. The carry flag CF is set to 1 if this correction is applied. In the third example, B8 + 60 = 18 with a carry of 1.

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

Both digits must be corrected in the last example, 12 + 6 = 18 and 18 + 60 = 78 (since CF=1).

The chart below completes the above examples, assuming that both of the following instructions are executed.

Before	After add	After daa
AL:34 BL:25	AL:59 AF:0, CF:0	AL:59 AF:0, CF:0
AL:37 BL:25	AL:5C AF:0, CF:0	AL:62 AF:1, CF:0
AL:93 BL:25	AL:B8 AF:0, CF:0	AL:18 AF:0, CF:1
AL:28 BL:39	AL:61 AF:1, CF:0	AL:67 AF:1, CF:0
AL:79 BL:99	AL:E4 AF:0, CF:1	AL:84 AF:0, CF:1

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

Each of the daa and das instructions encodes in a single byte. The daa instruction has opcode 27 and the das instruction has opcode 2F. Each requires three clock cycles to execute on a Pentium. In addition to modifying AF and CF, the SF, ZF and PF flags are set or reset by daa or das instructions to correspond to the final value in AL. The overflow flag OF is undefined and other flags are not affected.

The first BCD arithmetic procedure in this section adds two non-negative 10-byte numbers. This procedure will have two parameters, addresses of destination and source values, respectively. Each will serve as an operand, and the destination will be replaced by the sum, consistent with the way that ordinary addition instructions use the destination operand. We will not be concerned about setting flags; the exercises specify a more complete procedure that assigns appropriate values to SF, ZF, and CF. A design for the procedure is given below.

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

```
addBcd1   PROC      NEAR32
;add two non-negative 10 byte packed BCD numbers
;parameter1: address of operand1 (and destination)
;parameter2: address of operand2
;author: R. Detmer revised: 5/98 push ebp
;establish stack frame
          mov       ebp,esp
          push      esi                 ;save registers
          push      edi
          push      ecx
          push      eax
          mov       edi,[ebp+12]        ;destination address
          mov       esi,[ebp+8]         ;source address
          clc                           ;clear carry flag for first add
          mov       ecx,9               ;count of bytes to process

forAdd:   mov       al,[edi]            ;get one operand byte
          adc       al,[esi]            ;add other operand byte
          daa                           ;adjust to BCD
          mov       [edi],al            ;save sum
          inc       edi                 ;point at next operand bytes
          inc       esi
          loop      forAdd              ;repeat for all 9 bytes
          pop       eax                 ;restore registers
          pop       ecx
          pop       edi
          pop       esi
          pop       ebp
          ret       8                   ;return to caller
addBcd1   ENDP
```

Figure 11.3: Addition of non-negative packed BCD numbers

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

This design is implemented in the procedure *addBcd1*

```
point at first source and destination bytes;
for count := 1 to 9 loop copy destination byte to AL;
        add source byte to AL;
        use daa to convert sum to BCD;
        save AL in destination;
        point at next source and destination bytes;
end for;
```

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

```
point at first source and destination bytes;
for count := 1 to 9 loop
          copy destination byte to AL;
          subtract source byte from AL;
          use das to convert difference to BCD;
          save AL in destination string;
          point at next source and destination bytes;
end for;
if source > destination
then
          point at first destination byte;
          for count := 1 to 9 loop
                    put 0 in AL;
                    subtract destination byte from AL;
                    use das to convert difference to BCD;
                    save AL in destination string;
                    increment DI;
          end for;
          move sign byte 80 to destination string;
end if;
```

A subtraction procedure for 10-byte packed BCD numbers is more difficult. Even with the operands restricted to non-negative values, subtracting the source value (address in parameter 2) from the destination (address in parameter 1) will produce a negative result if the source is larger than the destination. A design for the procedure is below.

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

The first part of this design is almost the same as the design for addition. The condition *(source > destination)* is true if the carry flag is set after the first loop, and the difference is corrected by subtracting it from zero. If this were not done, then, for example, 3 −7 would produce 9999999999999999996 instead of −4. This design is implemented as procedure *subBcd1* in Fig. 11.4.

```
subBcd1   PROC      NEAR32
;subtract 2 non-negative 10 byte packed BCD numbers
;parameter1: address of operand1 (and destination)
;parameter2: address of operand2
;operand1 -- operand2 stored at destination
;author: R. Detmer revised: 5/98
          push      ebp                   ;establish stack frame
          mov       ebp,esp
          push      esi                   ;save registers
          push      edi
          push      ecx
          push      eax
          mov       edi,[ebp+12]          ;destination address (operand 1)
          mov       esi,[ebp+8]           ;source address (operand 2)
          clc                             ;clear carry flag
          mov       ecx,9                 ;count of bytes to process
```

Chapter 11: Decimal Arithmetic
11.2 Packed BCD Instructions

```
forSub:   mov    al,[edi]             ;get one operand byte
          sbb    al,[esi]             ;subtract other operand byte
          das                         ;adjust to BCD
          mov    [edi],al             ;save difference
          inc    edi                  ;point at next operand bytes
          inc    esi
          loop   forSub               ;repeat for all 9 bytes
          jnc    endIfBigger          ;done if destination >= source
          sub    edi,9                ;point at beginning of destination
          mov    ecx,9                ;count of bytes to process
forSub1:  mov    al,0                 ;subtract destination from zero
          sbb    al,[edi]
          das    mov [edi],al
          inc    edi                  ;next byte
          loop   forSub1
          mov    BYTE PTR [edi],80h   ;negative result
endIfBigger:
          pop    eax                  ;restore registers
          pop    ecx
          pop    edi
          pop    esi
          pop    ebp
          ret    8                    ;return to caller subBcd1 ENDP
```

Figure 11.4: Subtraction of non-negative packed BCD numbers

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

Once you have the *addBcd1* and *subBcd1* procedures that combine non-negative operands, it is not too difficult to construct the general packed BCD addition and subtraction procedures. The design for addition

```
if operand1 ≥ 0
Then
          if operand2 ≥ 0
          then
                    addBcd1(operand1, operand2);
          else
                    subBcd1(operand1, operand2);
          end if;
else {operand1 < 0}
          if (operand2 < 0)
          then
                    addBcd1(operand1, operand2);
          else
                    change sign byte of operand1;
                    subBcd1(operand1, operand2);
                    change sign byte of operand1;
          end if;
end if;
```

11.2 Packed BCD Instructions

The design for negative *operand1* is a little tricky. When *operand2* is also negative, the result will be negative. Since *addBcd1* does not affect the sign byte of the destination (*operand1*), the result after adding *operand2* will be negative with no special adjustment required. Adding a non-negative *operand2* can result in either a positive or negative result. The reader should verify that this design and corresponding code produces the correct sign for the result. This design is implemented in procedure *addBcd*, shown in Fig. 11.5. A general procedure for subtraction is left as an exercise.

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

```
addBcd    PROC      NEAR32
;add two arbitrary 10 byte packed BCD numbers
;parameter1: address of operand1 (and destination)
;parameter2: address of operand2
;author: R. Detmer revised: 5/98
          push      ebp                 ;establish stack frame
          mov       ebp,esp
          push      esi                 ;save registers
          push      edi
          mov       edi,[ebp+12]        ;destination address
          mov       esi,[ebp+8]         ;source address
          push      edi                 ;parameter1 for next call
          push      esi                 ;parameter2 for next call
          cmp       BYTE PTR [edi+9],80h          ;operand1 >= 0?
          je        op1Neg
          cmp       BYTE PTR [esi+9],80h          ;operand2 >= 0?
          je        op2Neg
          call      addBcd1             ;add (>=0, >=0)
          jmp       endIfOp2Pos
```

Chapter 11: Decimal Arithmetic

11.2 Packed BCD Instructions

```
op2Neg:    call      subBcd1              ;sub (>=0, <0)
endIfOp2Pos:
           jmp       endIfOp1Pos          ;done
op1Neg:    cmp       BYTE PTR [esi+9],80h          ;operand2 < 0 ?
           jne       op2Pos
           call      addBcd1              ;add (<0, <0)
           jmp       endIfOp2Neg
op2Pos:    xor       BYTE PTR [edi+9],80h          ;change sign byte
           call      subBcd1              ;sub (<0, >=0)
           xor       BYTE PTR [edi+9],80h          ;change sign byte
endIfOp2Neg:
endIfOp1Pos:
           pop       edi                  ;restore registers
           pop       esi
           pop       ebp
           ret       8          ;return to caller
addBcd     ENDP
```

Figure 11.5: General BCD addition procedure

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

Unpacked BCD numbers differ from packed representations by storing one decimal digit per byte instead of two. The bit pattern in the left half of each byte is 0000. This section describes

- ✓ how to define unpacked BCD numbers,
- ✓ how to convert this representation to and from ASCII, and
- ✓ how to use 80x86 instructions to do some arithmetic operations with unpacked BCD numbers.

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

Unpacked BCD representations have no standard length. In this book each value will be stored in eight bytes, with high-order digits on the left and low-order digits on the right (opposite to the way a DT directive stores packed BCD numbers). No sign byte will be used, so only non-negative numbers will be represented. An ordinary BYTE directive can be used to initialize an unpacked BCD value. For example, the statement

```
BYTE 0,0,0,5,4,3,2,8
```

reserves eight bytes of storage containing 00 00 00 05 04 03 02 08, the unpacked BCD representation for 54328. The directive

```
BYTE 8 DUP (?)
```

establishes an eight-byte-long area that can be used to store an unpacked BCD value.

11.3 Unpacked BCD Representations and Instructions

It is simple to convert an unpacked BCD value to or from ASCII. Suppose that the data segment of a program includes the directives

```
Ascii       DB 8 DUP (?)
Unpacked    DB 8 DUP (?)
```

If *unpacked* already contains an unpacked BCD value, the following code fragment will produce the corresponding ASCII representation at *ascii*.

```
        lea     edi,ascii           ;destination
        lea     esi,unpacked        ;source
        mov     ecx,8               ;bytes to process
For8:   mov     al,[esi]            ;get digit
        or      al,30h              ;convert to ASCII
        mov     [edi],al            ;store ASCII character
        inc     edi                 ;increment pointers
        inc     esi
        loop    for8                ;repeat for all bytes
```

11.3 Unpacked BCD Representations and Instructions

Converting from an ASCII string to an unpacked BCD representation is equally easy. The same loop structure can be used with the roles of EDI and ESI reversed, and with the or instruction replaced by

```
and al, 0fh ; convert ASCII to unpacked BCD
```

to mask the high-order four bits. Conversions between ASCII and unpacked BCD are even simpler if they are done "in place”.

11.3 Unpacked BCD Representations and Instructions

The 80x86 architecture includes four instructions to facilitate arithmetic with unpacked BCD representations. Each mnemonic begins with "aa" for "ASCII adjust"— Intel uses the word ASCII to describe unpacked BCD representations, even though the ASCII representation for a digit has 0011 in the left half byte and the unpacked representation has 0000.

Instruction	Mnemonic	Number of bytes	Opcode	Clocks (Pentium)
ASCII adjust after addition	**aaa**	1	37	3
ASCII adjust for subtraction	**aas**	1	3F	3
ASCII adjust after multiplication	**aam**	2	D4 0A	18
ASCII adjust before division	**aad**	2	D5 0A	10

Figure 11.6: Unpacked BCD instructions

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

An aaa instruction then corrects the value in AL if necessary. An aaa instruction sets flags and may also affect AH; recall that a daa affects only AL and flags.

The following algorithm describes how aaa works.

```
if (right digit in AL > 9) or (AF=1)
Then
        add 6 to AL;
        increment AH;
        AF := 1;
End if;
CF := AF;
left digit in AL := 0;
```

The action of an aas instruction is similar. The first two operations inside the *if* are replaced by

```
subtract 6 from AL;
decrement AH;
```

The OF, PF, SF, and ZF flags are left undefined by aaa and aas instructions

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

Here are some examples of showing how add and aaa work together.
In each example, assume that the following pair of instructions is executed.

Before	After add	After aaa
AX:00 04 CH:03	AX:00 07 AF:0	AX:00 07 AF:0, CF:0
AX:00 04 CH:07	AX:00 0B AF:0	AX:01 01 AF:1, CF:1
AX:00 08 CH:09	AX:00 11 AF:1	AX:01 07 AF:1, CF:1
AX:05 05 CH:07	AX:05 0C AF:0	AX:06 02 AF:1, CF:1

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

Another set of examples illustrates how sub and aas find differences of single byte unpacked BCD operands. This time assume that the following instructions are executed.

Before	After sub	After aaa
AX:00 08 DL:03	AX:00 05 AF:0	AX:00 07 AF:0, CF:0
AX:00 03 DL:07	AX:00 FC AF:1	AX:FF 06 AF:1, CF:1
AX:00 08 DL:03	AX:00 05 AF:0	AX:00 07 AF:0, CF:0
AX:05 02 DL:09	AX:05 F9 AF:1	AX:04 03 AF:1, CF:1

11.3 Unpacked BCD Representations and Instructions

Figure 11.7 displays a procedure *addUnp* that adds two eight-byte unpacked BCD numbers whose addresses are passed as parameters. This procedure is simpler than the similar *addBcd1* procedure in Fig. 11.3.

```
addUnp    PROC       NEAR32
;add two 8-byte unpacked BCD numbers
;parameter 1: operand1 and destination address
;parameter 2: operand2 address
;author: R. Detmer revised: 5/98
          push       ebp                  ;establish stack frame
          mov        ebp,esp
          push       esi                  ;save registers
          push       edi
          push       eax
          push       ecx
          mov        edi,[ebp+12]         ;destination address
          mov        esi,[ebp+8]          ;source address
          add        esi,8                ;point at byte after source
          add        edi,8                ;byte after destination
          clc                             ;clear carry flag
          mov        ecx,8                ;count of bytes to process
```

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

```
forAdd:  dec     edi            ;point at operand bytes to left
         dec     esi
         mov     al,[edi]       ;get one operand byte
         adc     al,[esi]       ;add other operand byte
         aaa                    ;adjust to unpacked BCD
         mov     [edi],al       ;save sum
         loop    forAdd         ;repeat for all 8 bytes
         pop     ecx            ;restore registers
         pop     eax
         pop     edi
         pop     esi
         pop     ebp
         ret     8              ;return, discarding paramters addUnp ENDP
```

Figure 11.7: Addition of two 8-byte unpacked BCD numbers

One interesting feature of the procedure *addUnp* is that it will give the correct unpacked BCD sum of eight byte ASCII (not unpacked BCD) numbers—Intel's use of "ASCII" in the unpacked BCD mnemonics is not as unreasonable as it first seems. The procedure is successful for ASCII strings since the action of the aaa instruction depends only on what add does with low-order digits, and aaa always sets the high-order digit in AL to zero. However, even if the operands are true ASCII character strings, the sum is not ASCII; it is unpacked BCD.

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

Two single byte unpacked BCD operands are multiplied using an ordinary mul instruction, resulting in a product in the AX register. Of course, this product will be correct as a binary number, not usually as a BCD value. The aam instruction converts the product in AX to two unpacked BCD digits in AH and AL. In effect, an aam instruction divides the number in AL by 10, putting the quotient in AH and the remainder in AL. The following examples assume that the instructions

```
mul     bh
aam
```

are executed

Before	After mul	After aam
AX:00 09 BH:06	AX:00 51	AX:08 01
AX:00 05 BH:06	AX:00 1E	AX:03 00
AX:00 06 BH:07	AX:00 2A	AX:04 02

Some flags are affected by an aam instruction. The PF, SF, and ZF flags are given values corresponding to the final value in AX; the AF, CF, and OF flags are undefined.

11.3 Unpacked BCD Representations and Instructions

Multiplication of single-digit numbers is not very useful. Figure 11.8 gives a procedure *mulUnp1* to multiply an eight-byte unpacked BCD number by a single-digit unpacked BCD number. The procedure has three parameters: (1) the destination address, (2) the address of the BCD source, and (3) a word containing the single-digit unpacked BCD number as its low-order byte.

```
mulUnp1   PROC      NEAR32
;multiply 8 byte and 1 byte unpacked BCD numbers
;parameter 1: destination address
;parameter 2: address of 8 byte unpacked BCD number
;parameter 3: word w/ low-order byte containing 1-digit BCD nbr
          push      ebp                 ;establish stack frame
          mov       ebp,esp
          push      esi                 ;save registers
          push      edi
          push      eax
          push      ebx
          push      ecx
          mov       edi,[ebp+14]        ;destination address
          mov       esi,[ebp+10]        ;source address
          mov       bx,[ebp+8]          ;multiplier
          add       esi,8               ;point at byte after source
```

11.3 Unpacked BCD Representations and Instructions

```
            add     edi,8               ;byte after destination
            mov     bh,0                ;lastCarry := 0
            mov     ecx,8               ;count of bytes to process
forMul:     dec     esi                 ;point at operand byte to left
            dec     edi                 ;and at destination byte
            mov     al,[esi]            ;digit from 8 byte number
            mul     bl                  ;multiply by single byte
            aam                         ;adjust to unpacked BCD
            add     al,bh               ;add lastCarry
            aaa                         ;adjust to unpacked BCD
            mov     [edi],al            ;store units digit
            mov     bh,ah               ;store lastCarry
            loop    forMul              ;repeat for all 8 bytes
            pop     ecx                 ;restore registers
            pop     ebx
            pop     eax
            pop     edi
            pop     esi
            pop     ebp
            ret     10                  ;return, discarding paramters
mulUnp1     ENDP
```

Figure 11.8: Multiplication of unpacked BCD numbers

11.3 Unpacked BCD Representations and Instructions

The algorithm implemented is essentially the same one as used by grade school children. The single digit is multiplied times the low-order digit of the multi-digit number, the units digit is stored, and the tens digit is recorded as a carry to add to the next product. All eight products can be treated the same by initializing a *last-Carry* variable to zero prior to beginning a loop. Here is the design that is actually implemented.

```
{ multiply X7X6X5X4X3X2X1X0 times Y giving Z7Z6Z5Z4Z3Z2Z1Z0}
lastCarry := 0;
for i := 0 to 7 loop
          multiply Xi times Y;
          add lastCarry;
          Zi := units digit;
          lastCarry := tens digit;
end for;
```

In the code for *mulUnp1*, the value for *lastCarry* is stored in the BH register. After a digit from the eight-byte BCD value is multiplied by the single digit in BL, the product is adjusted to unpacked BCD and *lastCarry* is added. It is then necessary to adjust the sum to unpacked BCD.

11.3 Unpacked BCD Representations and Instructions

The aad instruction essentially reverses the action of the aam instruction. It combines a two-digit unpacked BCD value in AH and AL into a single binary value in AX, multiplying the digit in AH by 10 and adding the digit in AL. The AH register is always cleared to 00. The PF, SF, and ZF flags are given values corresponding to the result; AF, CF, and OF are undefined.

The aad instruction is used *before* a div instruction. The examples below assume that the instructions

```
aad
div     dh
```

are executed

Before	After aad	After div
AX:07 05 DH:08	AX:00 4B DH:08	AX:03 09
AX:06 02 DH:04	AX:00 3E DH:04	AX:02 0F
AX:09 03 DH:02	AX:00 5D DH:02	AX:01 2E

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

Notice that the problems illustrated by the previous examples cannot occur when the original digit in AH is smaller than the divisor in DH. The elementary school algorithm for dividing a single digit into a multidigit number works left to right through the dividend, dividing a two digit number by the divisor. The first of the two digits is the remainder from the previous division, which must be smaller than the divisor. The following design formalizes the grade school algorithm.

```
{ divide X7X6X5X4X3X2X1X0 by Y giving Z7Z6Z5Z4Z3Z2Z1Z0 }

lastRemainder := 0;
for i := 7 downto 0 loop
        dividend := 10*lastRemainder + Xi;
        divide dividend by Y getting quotient & lastRemainder;
        Zi := quotient;
end for;
```

11.3 Unpacked BCD Representations and Instructions

Code that implements this design is given in Fig. 11.9. The AH register is ideally suited to store *lastRemainder* since that is where the remainder ends up following division of a 16-bit binary number by an 8-bit number.

```
divUnp1   PROC      NEAR32
;parameter 1: destination address
;parameter 2: address of 8 byte unpacked BCD number
;parameter 3: word w/ 1-digit BCD number as low-order byte
;author: R. Detmer revised: 5/98
          push      ebp                   ;establish stack frame
          mov       ebp,esp
          push      esi                   ;save registers
          push      edi
          push      eax
          push      ebx
          push      ecx
          mov       edi,[ebp+14]          ;destination address
          mov       esi,[ebp+10]          ;source address
          mov       bx,[ebp+8]            ;divisor
          mov       ah,0                  ;lastRemainder := 0
          mov       ecx,8                 ;count of bytes to process
```

Chapter 11: Decimal Arithmetic

11.3 Unpacked BCD Representations and Instructions

```
forDiv:  mov      al,[esi]          ;digit from 8 byte number
         aad                        ;adjust to binary
         div      bl                ;divide by single byte
         mov      [edi],al          ;store quotient
         inc      esi               ;point at next digit of dividend
         inc      edi               ;and at next destination byte
         loop     forDiv            ;repeat for all 8 bytes
         pop ecx                    ;restore registers
         pop      ebx
         pop      eax
         pop      edi
         pop      esi
         pop      ebp
         ret      10                ;return, discarding paramters
divUnp1  ENDP
```

Figure 11.9: Division of unpacked BCD numbers

Chapter 11: Decimal Arithmetic

11.4 Other Architectures: VAX Packed Decimal Instructions

Since the 80x86 architecture provides very limited support for packed decimal operations, a large procedure library is necessary to use packed decimal types. Some other architectures provide extensive hardware support for packed decimal. This section briefly examines packed decimal instructions defined in the VAX architecture, although not necessarily implemented in all VAX machines.

The VAX architecture defines a packed decimal string by its length and starting address. The length gives the number of decimal digits stored in the string, not the number of bytes. The last four bits (half byte) are always a sign indicator, normally C_{16} for positive and D_{16} for negative. Since decimal digits are packed two per byte, the length (in bytes) of a packed decimal string is approximately half the number of digits. More precisely, for *n* decimal digits it is $(n + 1)/2$ if *n* is odd and $(n + 2)/2$ if *n* is even.

The VAX architecture includes a complete set of instructions for performing packed decimal arithmetic: ADDP (add packed), DIVP (divide packed), MULP (multiply packed), and SUBP (subtract packed). Each of these has at least four operands to specify the length and address of each of the packed decimal strings involved. When just two strings are specified, one serves both as a source and the destination. All also have six-operand formats where the sources are specified separately from the destination. (MULP and DIVP have only the six-operand formats.) The MOVP (move packed) instruction copies a packed decimal string from one address to another. The CMPP (compare packed) instruction compares two packed decimal strings, setting condition codes (flags).