BCD (ASCII) Arithmetic

- The Intel Instruction set can handle both packed (two digits per byte) and unpacked BCD (one decimal digit per byte)
- We will first look at unpacked BCD
- Unpacked BCD can be either binary or ASCII. Consider the number 4567

  Bytes then look like 34h 35h 36h 37h

  OR: 04h 05h 06h 07h
- In packed BCD where there are two decimal digits per byte

  99h = 99d 4567d= 4567d

Unpacked BCD Arithmetic

- With unpacked BCD we wish to add strings such as '989' and '486' and come up with the string '1475'.
- With BCD you can use the standard input and output routines for strings to get numbers into and out of memory without converting to binary
- BCD arithmetic uses the standard binary arithmetic instructions and then converts the result to BCD using BCD adjustment instructions.

Where and Why is BCD used?

- BCD takes more space and more time than standard binary arithmetic
- It is used extensively in applications that deal with currency because floating point representations are inherently inexact
- Database management systems offer a variety of numeric storage options; "Decimal" means that numbers are stored internally either as BCD or as fixed-point integers
- BCD offers a relatively easy way to get around size limitations on integer arithmetic

From the SQL Server Manual

Exact Numerics	
bigint	numeric
bit	smallint
decimal	smallmoney
int	tinyint
money	
Approximate Numerics	
float	real

BCD Adjustment Instructions

- Four unpacked adjustment instructions are available:

  AAA (ASCII Adjust After Addition)

  AAS (ASCII Adjust After Subtraction)

  AAM (ASCII Adjust After Multiplication)

  AAD (ASCII ADjust BEFORE Division)
- Except for AAD the instructions are used to adjust results after performing a binary operation on ASCII or BCD data
- AAD (in spite of the mnemonic) is used after a DIV instruction

Packed BCD, ASCII, Unpacked BCD

- AAA and AAS can be used with both ASCII and unpacked BCD

  9701 in ASCII (hex) 39 39 30 31

  9701 in unpacked BCD 09 07 00 01
- Two Packed BCD operations are also available:

  DAA Decimal Adjust After Addition

  DAS Decimal Adjust After Subtraction

AAA

- This instruction has very complex semantics that are rarely documented correctly. Here are two examples:

```
IF AL > 9 OR AF = 1 THEN
    AL <- AL - 10
    AH <- AH + 1
    CF <- 1  AF <- 1
ELSE
    CF <- 0
    AF <- 0
END

IF AL > 9 OR AF = 1 THEN
    AL <- AL + 6
    AH <- AH + 1
    Bits 4-7 of AL set to 0
    AF and CF set
ELSE
    AF and CF clear
    Bits 4-7 of AL set to 0
END
```

Example

- Let's see what happens if we try to add ASCII strings byte by byte.

```
989 ——>  39 38 39
486 ——>  34 38 36
———      ————————
1475     6D 70 6F
```

- As you can see the result in binary does not look like what we want.
- When adding, AF is set whenever there is a carry from the lowest-order nibble to the next lowest nibble.
  - Recall that a NIBBLE is one hex digit or 4 bits.
  - When adding 9 and 6 in hex we get F and no binary carry.
- Note that AF is clear after the addition of 39h and 36h, because there was no carry from the low-order nibble to the next one
- If adding 9 and 7 we would get 10h, and AF would be set.

AAA Semantics

- AAA (ASCII Adjust for Addition) does the following things:

```
IF AL > 9 THEN
     clear top nibble of AL
     AL <- AL - 10
     AH <- AH + 1
     CF <- 1  AF <- 1
ELSE
     CF <- 0
     AF <- 0
     clear top nibble of AL
END
```

- Notes

1. addition result must be in AL in order for AAA to work
2. top nibble of AL always cleared so AAA will adjust for ASCII as well as unpacked BCD
3. Either AH or CF can be used for decimal carries.

AAA Example with no Aux Carry

```
AX=0000  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=0108   NV UP EI PL ZR NA PE NC
2BC5:0108 B039          MOV     AL,39
-t

AX=0039  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010A   NV UP EI PL ZR NA PE NC
2BC5:010A 0436          ADD     AL,36
-t

AX=006F  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010C   NV UP EI PL NZ NA PE NC
2BC5:010C 37            AAA
-t

AX=0105  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010D   NV UP EI PL NZ AC PO CY
```

AAA Example with Aux Carry

```
AX=0000  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=0108   NV UP EI PL ZR NA PE NC
2BC5:0108 B039          MOV     AL,39
-t

AX=0039  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010A   NV UP EI PL ZR NA PE NC
2BC5:010A 0438          ADD     AL,38
-t

AX=0071  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010C   NV UP EI PL NZ AC PE NC
2BC5:010C 37            AAA
-t

AX=0107  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010D   NV UP EI PL NZ AC PE CY
```

Example with Unpacked BCD

```
AX=0000  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=0108   NV UP EI PL ZR NA PE NC
2BC5:0108 B009          MOV     AL,09
-t

AX=0009  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010A   NV UP EI PL ZR NA PE NC
2BC5:010A 0409          ADD     AL,09
-t

AX=0012  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010C   NV UP EI PL NZ AC PE NC
2BC5:010C 37            AAA
-t

AX=0108  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010D   NV UP EI PL NZ AC PE CY
```

Example with No Carries

```
AX=0000  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=0108   NV UP EI PL ZR NA PE NC
2BC5:0108 B031          MOV     AL,31
-t

AX=0031  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010A   NV UP EI PL ZR NA PE NC
2BC5:010A 0431          ADD     AL,31
-t

AX=0062  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010C   NV UP EI PL NZ NA PO NC
2BC5:010C 37            AAA
-t

AX=0002  BX=0000  CX=0012  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=2BC5  ES=2BC5  SS=2BC5  CS=2BC5  IP=010D   NV UP EI PL NZ NA PO NC
```

BCD Addition Program

```
segment .data
str1 db '04989' ; leading 0 allows easy processing
str2 db '07486' ; in loop without concern for
segment .bss
sum  resb 5     ; extra digit for carry out

segment .text
...
 mov ssi, str1+4 ;point to LSD
 mov ebx, str2+4
 mov edi, sum +4
 mov ecx, 5             ;digits to process
 clc                    ;ensure cf clear
```

BCD Addition Program (2)

```
LP1:
 mov al,[esi]        ;get a digit from op1
 adc al,[ebx]        ;add prev cf + op2 digit
 aaa                 ;adjust
 mov [edi],al        ;save result
 dec ebx             ;advance all 3 pointers
 dec esi
 dec edi
 loop LP1

 mov ecx, 5          ;convert to ASCII
 inc edi             ;adjust di back to MSD
LP2:
 or byte [edi],30H   ;convert
 inc edi
 loop LP2
```

What's Wrong with This?

```
;this code tries to do it in a single loop
;but doesn't work correctly
LP1:
 mov al,[esi]      ;get a digit from op1
 adc al,[ebx]      ;add prev cf + op2 digit
 aaa               ;adjust
 or al, 30h        ;convert to ASCII
 mov [edi],al      ;save result
 dec ebx           ;advance all 3 pointers
 dec esi
 dec edi
 loop LP1

;How can we fix without using a 2nd loop?
```

AAS

- AAS (ASCII Adjust for Subtraction) works in a similar manner to AAA
- Note that negative results are expressed in 10's complement.

```
-a
1469:0100 mov al,31
1469:0102 sub al,39
1469:0104 aas
1469:0105
-t

AX=0031  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1469  ES=1469  SS=1469  CS=1469  IP=0102   NV UP EI PL NZ NA PO NC
1469:0102 2C39          SUB     AL,39
-t

AX=00F8  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1469  ES=1469  SS=1469  CS=1469  IP=0104   NV UP EI NG NZ AC PO CY
1469:0104 3F            AAS
-t

AX=FF02  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1469  ES=1469  SS=1469  CS=1469  IP=0105   NV UP EI NG NZ AC PO CY
```

10's Complement

- The reason that 2's complement is used in computers is that we can replace subtraction with addition
- We can apply the same principle to decimal arithmetic
- To obtain the 10's complement of a number take the 9's complement of each digit and then add 1
- Example:
  - 68 – 37 = ?
  - Compute 10's complement of 37:
    - 99 – 37 = 62
    - 62 + 1 = 63
  - Compute 68 + 63, ignoring any carry:
    - 63 + 68 = 131 = 31

10's Complement (2)

- Complement notation is used with "fixed size" integers.
- For a given size n digits, you can compute the 10's complement by subtracting from n 9's and then add 1
- Or you can subtract the number from 10^{n+1}
- Example: what is –77 in 5 digit 10's complement?
  - 99999 – 00077 = 99922 + 1 = 99923
  - Or 100000 – 77 = 99923
- Note that the leftmost digit is a "sign digit." If it is >= 5 the result is negative

Other Adjustment Instructions

- There are four other instructions used in BCD arithmetic:
- Unpacked BCD:
  - AAM — ASCII Adjust After Multiplication
  - AAD — ASCII Adjust before Division
- Packed BCD:
  - DAA — Decimal adjust after addition
  - DAS — Decimal adjust after subtraction
- Note that there are no multiplication or division instructions for packed BCD
- The BCD instructions can also be used for certain specialized conversions.
- We will take a brief look at AAM and AAD

AAM (ASCII Adjust after Multiplication)

- Used after multiplication of two unpacked BCD numbers (note: NOT ASCII!). Semantics:
  - AL <- AL mod 10, AH <- AL/10
  - SF <- high bit of AL
  - ZF <- set if AL = 0

```
AX=0000  BX=0000  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0105   NV UP EI PL ZR NA PE NC
22E5:0105 B007          MOV     AL,07
-t
AX=0007  BX=0000  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0107   NV UP EI PL ZR NA PE NC
22E5:0107 B306          MOV     BL,06
-t
AX=0007  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0109   NV UP EI PL ZR NA PE NC
22E5:0109 F6E3          MUL     BL
-t
AX=002A  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=010B   NV UP EI PL ZR NA PE NC
22E5:010B D40A          AAM
-t
AX=0402  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=010D   NV UP EI PL NZ NA PO NC
```

AAD (ASCII Adjust before Division)

- Unlike other BCD instructions, AAD is performed before a division rather than after
  - AL <- AH * 10 + AL
  - AH <- 0
  - ZF set if AL = 0, SF <- high bit of AL

```
AX=0402  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0110   NV UP EI PL NZ NA PO NC
22E5:0110 B306          MOV     BL,06
-t

AX=0402  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0112   NV UP EI PL NZ NA PO NC
22E5:0112 D50A          AAD
-t

AX=002A  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0114   NV UP EI PL NZ NA PO NC
22E5:0114 F6F3          DIV     BL
-t

AX=0007  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0116   NV UP EI PL ZR NA PE NC
```

Undocumented Operations with AAM and AAD

- In the original 8086, there was not enough room in the microcode for the constant 10d intended to be used for AAM and AAD.
  - Consequently the 10d is actually placed as an immediate value in the machine code, even though the instruction was documented only as a 0-operand instruction
- AAM and AAD are assembled as 2-byte, one operand instructions AAD *imm* and AAM *imm*
  - Values other than 10 can be used (if the assembler will do it or if you are willing to patch the assembled code manually)
- Both are particularly useful with the value 16
  - Because many programs came to rely on this behavior, Intel retained it but never documented it
  - Other manufacturers reproduced the behavior

Packing and Unpacking BCD

- AAM 16 will "unpack" packed BCD in AL into AH and AL

```
22E5:0116 B85600        MOV     AX,0056
-t
AX=0056  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0119   NV UP EI PL ZR NA PE NC
22E5:0119 D410          AAM     10
-t
AX=0506  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=011B   NV UP EI PL NZ NA PE NC
```

- AAD 16 will "pack" AH and AL into AL

```
AX=0000  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=011D   NV UP EI PL ZR NA PE NC
22E5:011D B008          MOV     AL,08
-t
AX=0008  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=011F   NV UP EI PL ZR NA PE NC
22E5:011F B405          MOV     AH,05
-t
AX=0508  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0121   NV UP EI PL ZR NA PE NC
22E5:0121 D510          AAD     10
-t
AX=0058  BX=0006  CX=0028  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=22E5  ES=22E5  SS=22E5  CS=22E5  IP=0123   NV UP EI PL NZ NA PO NC
```

2-Digit Decimal Conversions

- The zero-operand AAD and AAM with implied base 10 can be used for conversion of 2-digit decimal numbers
- Examples

```
; binary to ASCII
mov ah, 2ch      ; DOS get time function
int 21h
mov al, cl       ; load minutes into AL
aam              ; al <- al mod 10, ah <- al/10
or ax, 3030h     ; convert to ascii
```

ASCII to Binary

```
;ASCII to binary
mov al, lowDigit    ; get ones ASCII digit
mov al, highDigit   ; get tens ASCII digit
sub ax, 3030h       ; convert to binary
aad                 ; AL <- AH * 10 + AL
mov binNum, al
```

Generalized BCD Addition

```
; register parameter version
BCDAdd:
  ; esi points to first digit of operand 1
  ; edi points to first digit of operand 2
  ; ebx points to first digit of storage for result
  ; ecx contains digit count (same for both operands!)
  ; result is one byte longer for carry

  pusha          ; save regs
  ; adjust pointers to LSD of operands and result
  add ebx, ecx   ; start + digit count
  add esi, ecx   ; start + digit count
  dec esi        ; minus one
  add edi, ecx   ; start + digit count
  dec edi        ; minus one
```

Generalized BCD Addition:2

```
  clc            ; clear carry for adc in loop
LP1:
  mov al,[esi]   ; get digit from op1
  adc al,[edi]   ; add in corresponing digit from op2
  aaa            ; adjust
  pushf          ; save flags
  or al,30H      ; convert to ascii
  popf           ; restore CF
  mov [ebx],al   ; and store
  dec ebx        ; adjust pointers (CF unchanged!)
  dec esi
  dec edi
  loop LP1
; now we're done processing all digits,
; BUT we have to account for a possible carry
  mov byte [ebx], 0   ; clear MSD of result
  adc byte [ebx], 0   ; add in CF
  ret
```

cdecl BCD Addition

```
BCDAdd:
  ; this version gets a pointer to the START of the
  ; string instead of the END of the string
  ; call from C with BCDAdd(op1, op2, result, cnt)
  %define dcount   [ebp+20]
  %define result   [ebp+16]
  %define operand2 [ebp+12]
  %define operand1  [ebp+8]
  push ebp       ; save caller's frame pointer
  mov ebp, esp   ; set up frame pointer
  pusha          ; save regs for C
  mov esi, operand1   ; these params are pointers
  mov edi, operand2
  mov ebx, result
  dec esi             ; adjust to avoid 0 delimiter in C
  dec edi
  dec ebx
  mov ecx, dcount     ; this is a value
```

cdecl BCD Addition:2

```
  clc            ; ensure CF clear
LP1:
  ; use ecx with SIB addressing to
 count back

  mov al,[esi+ecx]     ; get digit
 from op1
                       ; add in
 corresponding
  adc al,[edi+ecx]     ; digit from
 op2
  aaa              ; BCD adjust
  mov [ebx+ecx],al     ; and store
```