Lecture 6: More About Integer Arithmetic

Table of contents

1 Objectives:

In this lecture, you will learn how to perform:

1. Shifting and rotation
2. Extended addition and subtraction
3. Binary-Coded Decimal operations.

2 Shifting and Rotation

- Bit shifting means to move bits right and left inside an operand.
- x86 architecture provides a rich set of instruction in this area.

Instruction	Description
SHL	Shift (logic) left
SHR	Shift (Logic) right
SAL	Shift arithmetic left

Instruction	Description
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate through carry left
RCR	Rotate through carry right
SHLD	Double-precision shift left
SHRD	Double-precision shift right

2.1 Logical and Arithmetic Shifts

Syntax

```
SHL     <destination>, <count>
SHR     <destination>, <count>
SAL     <destination>, <count>
SAR     <destination>, <count>
```

Rules

- The destination operand can be either register or memory location
- The count operand can be either:
  1. CL register
  2. immediate (of size 8-bit)
  3. 1

- Therefore, the valid forms are:

```
SHL   reg/mem, imm8
SHL   reg/mem, CL
SHL   reg/mem, 1
```

- The form `SHL reg/mem, 1` has its own,optimized op-codes in x86 architecture.
- The count is masked to 5 bits (or 6 bits with a 64-bit operand). The count range is limited to 0 to 31 (or 63 with a 64-bit operand).

2.1.1 Logical Shifts using `SHL` and `SHR` instructions

- If you treat the destination operand as unsigned integer, then use logical shifts.
- `SHL` and `SHR` instructions shift the bits of the destination operand to the left (toward most significant bit locations) and right (toward less significant bit locations), respectively.
- For each shift count to the right, the least significant bit of the destination operand is shifted into the `CF` flag, and the most significant bit is cleared. See Figure 1 (a).
- For each shift count to the left, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared. See Figure 1 (b).

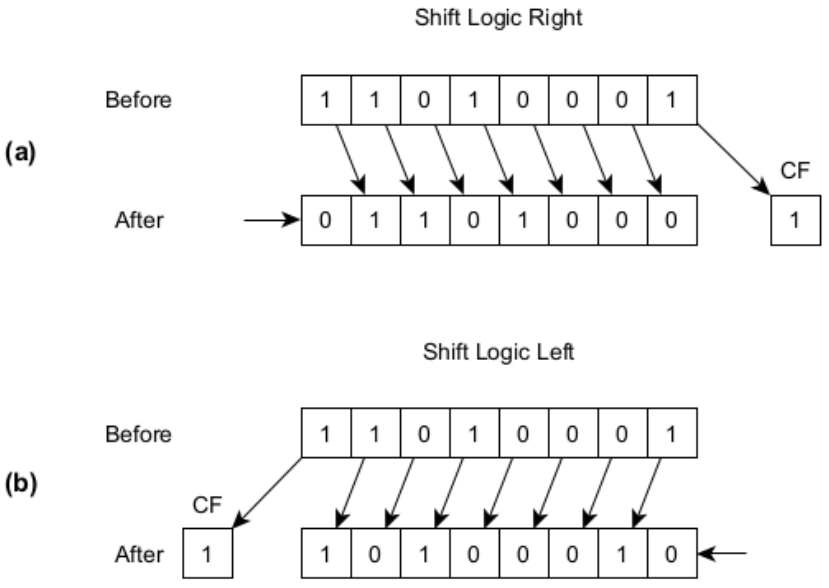

Figure 1

2.1.2 Arithmetic Shifts using `SAL` and `SAR` instructions

- If you treat the destination operand as signed integer, then use arithmetic shifts.
- For each shift count to the right, the least significant bit of the destination operand is shifted into the `CF` flag, and the most significant bit is set or cleared according to the sign of the original value. See Figure 2 (a).
- For each shift count to the left, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared. See Figure 2 (b).

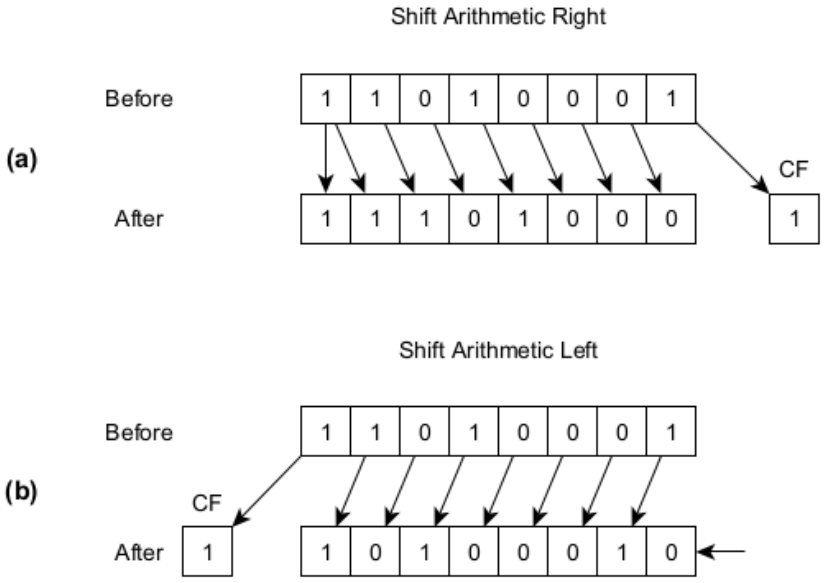

Figure 2

- Notice that there is no difference between `SHL` and `SAL`. Eventually, they have the same op-code.

2.1.3 Flags Affected

- The `CF` flag contains the value of the last bit shifted out of the destination operand.
- The `OF` flag is affected only on 1-bit shifts.

  - For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1.
  - For the SAR instruction, the OF flag is cleared for all 1-bit shifts.
  - For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

- The SF, ZF, and PF flags are set according to the result.
- The AF flag is undefined.
- If the count is 0, the flags are not affected.

2.2 Rotations

Syntax

```
ROL     <destination>, <count>
ROR     <destination>, <count>
RCL     <destination>, <count>
RCR     <destination>, <count>
```

- The rotation instructions have the same rule as in shift instructions.
- The rotate left (ROL) and rotate through carry left (RCL) instructions shift ALL bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location.
- The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

2.2.1 Rotations using ROL and ROR instructions

Figure 3 illustrates one-round of bits shifting. It shows that the bit shifted into CF is copied to the least or most significant bit (according to the shift direction).

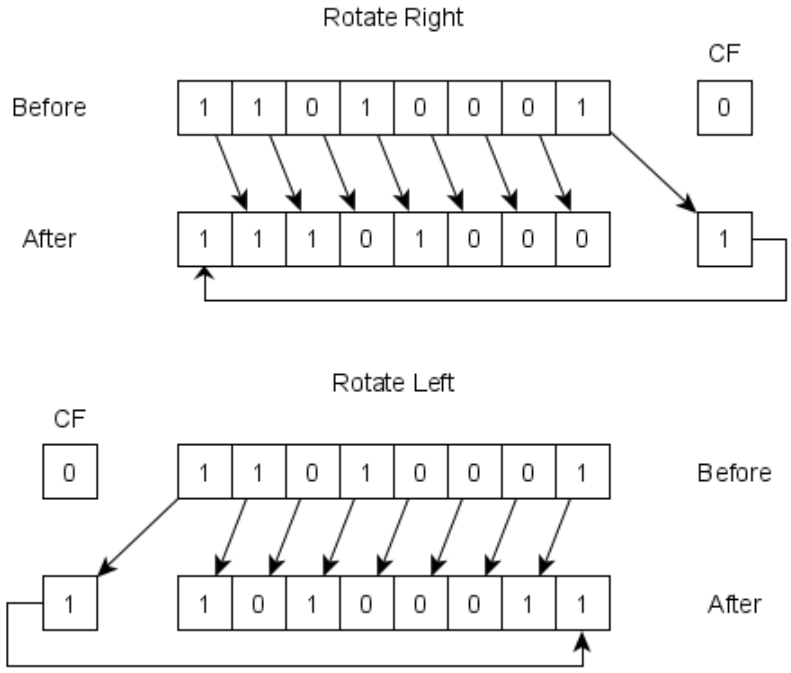

Figure 3

2.2.2 Rotations through carry using RCL and RCR

Figure 4 illustrates the rotation through `CF` flag. If the direction of shifting is to the right, the `CF` is acting as least significant bit. If the direction of shifting is to the left, the `CF` is acting as most significant bit.

2.2.3 Flags Affected

- The `CF` flag contains the value of the last bit shifted out of the destination operand.
- The OF flag is defined only for 1-bit rotates:
  - For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result.
  - For right rotates, the OF is set to the exclusive OR of the two most-significant bits of the result.

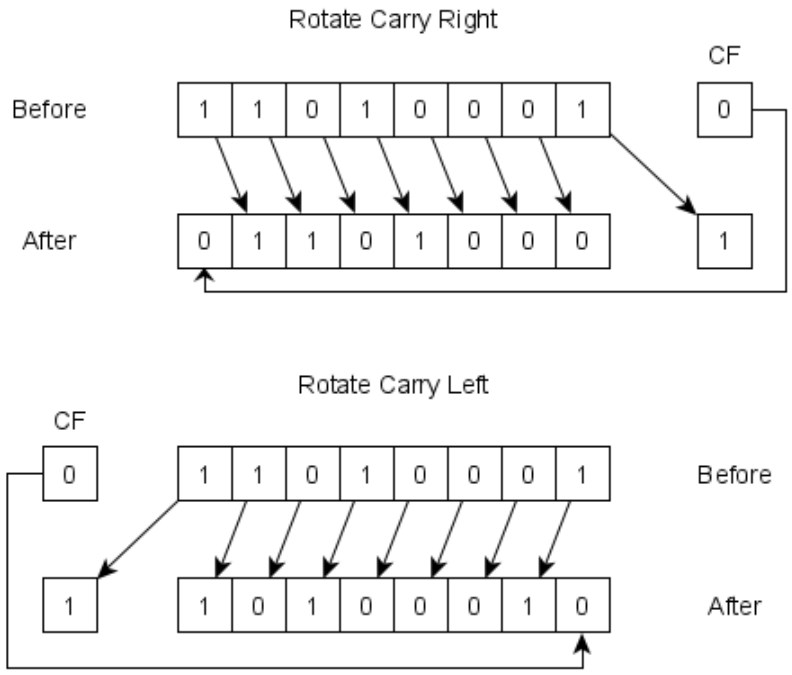

Figure 4

- If count is 0, all flags are not affected.
- The SF, ZF, AF, PF flags are always undefined.

2.3 Multi-precision Shifts

Syntax

```
SHLD    <destination>, <source>, <count>
SHRD    <destination>, <source>, <count>
```

Rules

- The destination operand can be a register or a memory location of size word or doubleword.

- The source operand must be a register
- The destination and source operands must be of the same size.
- The count operand can be unsigned immediate byte or CL register.
- In 32-bit, the count value should be between 0 and 31.

- According to the rules above, the valid forms are:

```
SHLD    reg/mem16, 1
SHLD    reg/mem16, CL
SHLD    reg/mem16, imm8
SHLD    reg/mem32, 1
SHLD    reg/mem32, CL
SHLD    reg/mem32, imm8
```

- The SHLD instruction shifts the first operand (destination operand) to the left the number of bits specified by the count operand. The empty bit positions opened up by the shift are filled by the most significant bits of the source operand.
- Similarly, the SHRD instruction shifts the destination operand to the right, and the empty bit positions are filled by the least significant bits of the source operand.
- Figure 5 illustrates the movement of shifted bits from the source operand to the destination operand to the CF flag.

Figure 5

- Figure 6 shows the exact bits that will shifted and moved from the source operand to destination operand when they are 32-bit.

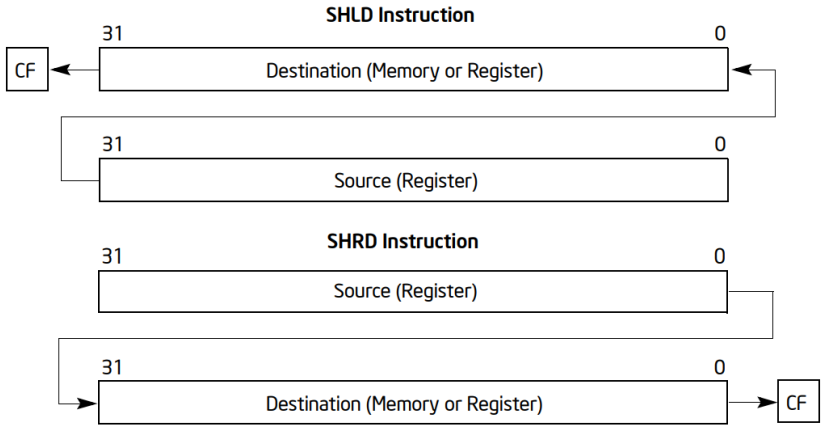

Figure 6

2.3.1 Flags Affected

- The CF flag contains the value of the last bit shifted out of the destination operand.
- The OF flag is affected only on 1-bit shifts, the OF flag is set if a sign change occurred; otherwise, it is cleared.
- The SF, ZF, and PF flags are set according to the result.
- The AF flag is undefined.
- If the count is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

2.4 Examples

Example 2.1.

```
    mov   bl, 8Fh      ; BL= 1000 1111b
    shl   bl, 1        ; BL= 0001 1110b, CF=1, OF=1

    mov   bl, 8Fh
    sal   bl, 2        ; BL= 0011 1100b, CF=0
```

Example 2.2.

```
    mov   al, 0D7h     ; AL= 1101 0111b
    shr   al, 2        ; AL= 0011 0101b, CF=1

    mov   al, 0D7h
    sar   al, 2        ; AL= 1111 0101b, CF=1
```

Example 2.3.

```
    mov   al, 0C0h     ; AL= 1100 0000b
    rol   al, 1        ; AL= 1000 0001b, CF=1
    rol   al, 1        ; AL= 0000 0011b, CF=1
    rol   al, 1        ; AL= 0000 0110b, CF=0

    mov   al, 26h
    rol   al, 4        ; AL= 62h
```

Example 2.4.

```
    clc                ; Clear CF (CF=0)
    mov   bl, 88h      ; BL= 1000 1000b
    rcl   bl, 1        ; BL= 0001 0000b, CF = 1
    rcl   bl, 1        ; BL= 0010 0001b, CF = 0
```

Example 2.5.

```
    stc                ; Set CF (CF=1)
    mov   ah, 10h      ; AL= 0001 0000b
    rcr   ah, 1        ; AL= 1000 1000b, CF=0
```

Example 2.6.

```
mov   ax, 0
mov   bx, 1234h
shld  ax, bx, 4    ; AX=0001h, BX=1234h, CF=1

mov   ax, 0
shrd  a, bx, 4     ; AX=4000h, BX=1234h, CF=0
```

2.5 Shift and Rotate Applications

1. The SHL and SAL instructions perform unsigned and signed multiplication when the multiplier is a power of 2. Shifting an integer n bits to the left multiplies it by 2^n .
2. The SHR and SAR instructions can be used to perform unsigned or signed division, respectively, of the destination operand by power of 2.
    - Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity.
3. The shift instructions can be used to extract data (field value) from a **bit string**.
    - For example, the first word of the Internet Protocol (IP) header has three fields: version, header length, and type of service, as shown below.

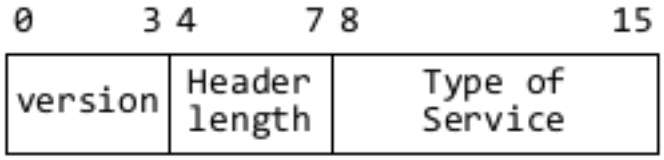

Figure 7

    - Assume this word has been read and stored in AX register.

- Then, we can extract `version` by masking off bits from position 4 up to position 15:

```
mov   dx, ax         ; make a copy of AX
and   al, 00001111b  ; clear bits 4-7
mov   [version], al  ; store the result
```

- Then, we extract `hdrlen` (header length) by shifting original `AX` to the right 4 bits and then mask off bits from position 4 up to 15:

```
mov   ax, dx         ; restore original value
shr   ax, 4
and   al, 0000111b   ; clear bits 4-7
mov   [hdrlen], al
```

- Finally, we extract `tos` (type of service) by shifting original AX to the right 8 bits and then mask off bits from position 8 up to 15:

```
mov   ax, dx         ; restore original value
shr   ax, 8
mov   [tos], al
```

- Of course, we can simply assign `DH` directly to `[tos]` without shifting `AX` register. However, the purpose of the example is to show you how to extract data from a **bit string**.

3 Extended Addition and Subtraction

3.1 `ADC` Instruction

Syntax

```
ADC   <destination>, <source>
```

- `ADC` (Add with Carry) instruction adds the destination operand, the source operand, and the CF flag and stores the result in the destination operand.

- It performs the following operation: DEST = DEST + SRC + CF
- The rules are similar to ADD rules.

3.2 SBB Instruction

Syntax

```
SBB    <destination>, <source>
```

- SBB (Subtract with Borrow) instruction adds the source operand and the CF flag, and subtracts the result from the destination operand. The result of the subtraction is stored in the destination operand.
- Operation: DEST = DEST - (SRC + CF)
- The rules are similar to ADD rules.

3.3 Flags Affected

All flags are set according to the result.

3.4 Examples

Example 3.1. The following instructions add two 8-bit integers (FFh + FFh), producing a 16-bit sum in DL : AL, which is 01FEh.

```
mov     dl, 0
mov     al, 0FFh
add     al, 0FFh        ; AL=FFh + FFh = FEh, CF = 1
adc     dl, 0           ; DL=1, CF=0
```

Example 3.2. This example demonstrates a technique of adding two numbers having unlimited size. In C++, for example, no standard operator permits you to add two 1024-bit integers.

Let us assume, we have two numbers `a` and `b` that store their numbers as array of bytes of size `n`. The least significant byte is stored at index 0, and the most significant byte is stored at index `n-1`. The following diagram shows how to add two numbers (4-byte) and store the result in array `s (5-byte)`.

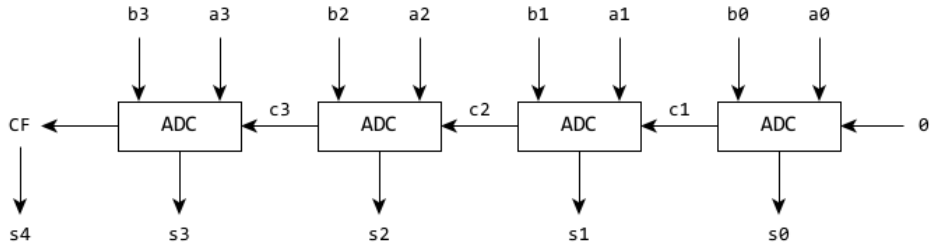

Figure 8

Notice that the size of array `s` is `n+1`. The following code implements the **extended** addition:

```
      mov   esi, a      ; ESI is pointing to array a
      mov   edi, b      ; EDI is pointing to array b
      mov   ebx, s      ; EBX is pointing to array s
      mov   ecx, 128    ; a & b are 1024-bit, s is 1032-bit
      mov   al, 0       ; set AL to zero
L1:
      clc               ; set CF to 0
      add   al, [esi]
      adc   al, [edi]   ; AL = a[i] + b[i]
      mov   [ebx], al   ; store result at s[i]
      mov   al, 0       ; reset AL
      adc   al, 0       ; store CF to AL for next round
      inc   esi
      inc   edi
      inc   ebx
      loop  L1
      mov   [ebx], al   ; store last CF
```

4 Unpacked and Packed Decimal Arithmetic

- So far, we have studied arithmetic operations where the values are in binary representation. In this section, we study the IA-32's support for Binary-Coded Decimal (BCD).
- Recall that, in BCD, a decimal digit can be represented either as upacked BCD or packed BCD. Please refer to Lecture 1.
- x86 architecture provides a set of instructions for BCD integers. Eventually, the CPU always calculates in binary. It supports BCD arithmetic by adjusting the binary result into a BCD value.
- For example, if we need to add two BCD values 5 and 8, then,

```
-   We first add them by using the binary addition instruction, such as `A

    ``` nasm
          mov   ax, 5      ; AX = 0005h
          add   ax, 8      ; AX = 000Dh
    ```

-   Then, the CPU has an instruction to convert the result `000Dh` to unpa
```

- Here are the list of instructions that support BCD conversions:

Mnemonic	Description
AAA	ASCII adjust after addition for unpacked BCD
AAS	ASCII adjust after subtraction for unpacked BCD
AAM	ASCII adjust after multiplication for unpacked BCD
AAD	ASCII adjust before division for unpacked BCD
DAA	Decimal adjust after addition for packed BCD
DAS	Decimal adjust after subtraction for packed BCD

- These instructions have no operand. The destination operand is implied either AL or AX register.
- In 64-bit mode, these instructions are not accessible.

4.1 AAA Instruction

- This instruction adjusts the sum of two unpacked BCD values to create an unpacked BCD result.
- The AL register is the implied source and destination operand.
- If the adjustment produces a decimal carry, the AH register is incremented by 1, and CF and AF flags are set.

Example 4.1. Consider the following code:

```
mov   ah, 0
mov   al, 4      ; AX = 0004h
add   al, 7      ; AX = 000Bh
aaa              ; AX = 0101h
```

Example 4.2. We can also add two ASCII digits:

```
mov   ah, 0
mov   al, '8'      ; AX = 0038h
add   al, '2'      ; AX = 006Ah
aaa                ; AX = 0100h, CF=1, AF=1
or    ax, 3030h    ; AX = 3130h = '10'
```

4.1.1 Flags Affected

- The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0.
- The OF, SF, ZF, and PF flags are undefined.

4.2 DAA Instruction

- This instruction adjusts the sum of two packed BCD values to create a packed BCD result.
- The AL register is the implied source and destination operand.

Example 4.3.

```
    add   al, bl     ; before AL=79h, BL=35h
                     ; after  AL=AEh, BL=35h CF=0
    daa              ; AL=14h, CF=1, AF=1
```

4.2.1 Flags Affected

- The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result.
- The SF, ZF, and PF flags are set according to the result.
- The OF flag is undefined.

4.3 AAS Instruction

- This instruction adjusts the result of the subtraction of two unpacked BCD values to create an unpacked BCD result.
- The AL is the implied source and destination operand.
- If the adjustment produces a decimal carry, the AH register is decremented by 1, and CF and AF flags are set.

4.3.1 Flags Affected

- Similar to AAA instruction

4.4 DAS Instruction

- This instruction adjusts the result of subtraction of two packed BCD values to create a packed BCD result.
- The AL register is the implied and destination operand.

4.4.1 Flags Affected

- Similar to DAA instruction.

4.5 AAM Instruction

- This instruction adjusts the result of multiplying two unpacked BCD values to create a pair of unpacked BCD values.
- The AX register is the implied source and destination operand.

Example 4.4.

```
mov   al, 8   ; AL=8h
mov   bl, 2   ; BL=2h
mul   bl      ; AX=10h
aam           ; AX=0106h
```

4.5.1 Flags Affected

- CF , OF, and AF flags are undefined.
- The SF, ZF, and PF flags are set according to the value in the AL.

4.6 AAD Instruction

- This instruction is different from the previous instructions. It adjusts two unpacked BCD digits stored in AX register to create a dividend value in binary format. Thus, when DIV is performed on the result, it will yield a correct unpacked BCD results.
- The AX register is the implied source and destination operand.

Example 4.5.

```
mov   ax, 0307h    ; dividend = decimal 37
aad                ; AX = 0025h
mov   bl, 5        ; divisor
div   bl           ; AX = 0207h
                   ; quotient in AL
                   ; remainder in AH
```

- AAD instruction eventually sets the value in the AL register to (AL + (10*AH)), and then clears the AH register to 00H.

4.6.1 Flags Affected

- Similar to AAM.

4.7 Example

This example represents a complete assembly program that shows how to add two ASCII decimal values. The implementation is similar to Example 3.2, in which the carry from each digit addition is propagated to the next higher position.

```
            global _main
            extern _printf

DECIMALSZ   EQU     10

            section .data
d1          db      "1234567890", 0
d2          db      "7856341289", 0
prt_fmt     db      `  %10s\n+ %10s\n------------\n %11s\n`, 0
            section .bss
d3          resb    DECIMALSZ+2

            section .code
_main:
            mov     esi, DECIMALSZ-1    ; Set Index to Last Decimal
            mov     ax, 0               ; clear AL and AH
            mov     ecx, DECIMALSZ      ; set loop counter to DECIMALSZ
top1:
            add     al, [d1 + esi]      ; AL = Carry + d1[i]
            add     al, [d2 + esi]      ; AL = AL + d2[i]
            aaa                         ; adjust to unpacked BCD
            or      al, 30h             ; covert AL to ASCII
            push    ax                  ; store result into stack
            mov     al, ah              ; store Carry into AL
            mov     ah, 0               ; clear AH
            dec     esi                 ; move ESI to next highest posit
            loop    top1

            ; the number of elements pushed onto stack is DECIMALSZ
            mov     ecx, DECIMALSZ

            ; Check if there is a carry out of the highest position
            cmp     al, 0               ; if AL == 0h
            je      done                ; jump if no carry
            or      al, 30h             ; convert to ASCII
            push    ax                  ; store onto stack
```

```
37              inc     ecx                     ; add one element
38  done:
39              ; now move result from stack into d3
40              mov     esi, 0
41  top2:
42              pop     ax
43              mov     [esi + d3], al
44              inc     esi
45              loop    top2
46
47              mov     BYTE [esi + d3], 0  ; insert null-character
48              ; prints out the result
49              push    d3
50              push    d2
51              push    d1
52              push    prt_fmt
53              call    _printf
54              add     esp, 16
55
56              xor     eax, eax
57              ret
```

Here is the output:

```
  1234567890
+ 7856341289
------------
  9090909179
```