**Easy Notes: Shift and Rotate Instructions (x86 Assembly)**

**1. What are Shift and Rotate Instructions?**

* **Shift Instructions**: Move bits left or right in a binary number.
* **Rotate Instructions**: Move bits in a circular way (bits that fall off one end are put back on the other).
* These instructions affect the **Carry Flag (CF)** and **Overflow Flag (OF)**.

**2. Types of Shift Instructions**

**(A) Logical Shifts**

* **SHL (Shift Left)**:
  + Bits move left.
  + Empty bit (LSB) fills with **0**.
  + Last shifted-out bit (MSB) goes into **CF**.
  + Example: SHL BL, 1
    - BL = 10001111 → After shift: 00011110 (CF = 1)
  + **Effect**: Multiplies by **2ⁿ** (if shifting left by n bits).
* **SHR (Shift Right)**:
  + Bits move right.
  + Empty bit (MSB) fills with **0**.
  + Last shifted-out bit (LSB) goes into **CF**.
  + Example: SHR AL, 1
    - AL = 11010000 → After shift: 01101000 (CF = 0)
  + **Effect**: Divides (unsigned) by **2ⁿ** (if shifting right by n bits).

**(B) Arithmetic Shifts**

* **SAR (Shift Arithmetic Right)**:
  + Bits move right, but **sign bit (MSB) is preserved**.
  + Used for **signed numbers**.
  + Example: SAR AL, 1
    - AL = 11110000 (-16) → After shift: 11111000 (-8) (CF = 0)
  + **Effect**: Divides (signed) by **2ⁿ**.
* **SAL (Shift Arithmetic Left)**:
  + Same as **SHL** (no difference in operation).

**3. Rotate Instructions**

* **ROL (Rotate Left)**:
  + Bits move left in a circle.
  + Last bit (MSB) goes to **CF** and **LSB**.
  + Example: ROL BL, 1
    - BL = 10001111 → After rotate: 00011111 (CF = 1)
* **ROR (Rotate Right)**:
  + Bits move right in a circle.
  + Last bit (LSB) goes to **CF** and **MSB**.
  + Example: ROR BL, 1
    - BL = 10001111 → After rotate: 11000111 (CF = 1)
* **RCL (Rotate Carry Left)**:
  + Like **ROL**, but **CF is included in rotation**.
  + Example: RCL BL, 1
    - If CF = 1 and BL = 10001111 → After rotate: 00011111 (CF = 1)
* **RCR (Rotate Carry Right)**:
  + Like **ROR**, but **CF is included in rotation**.
  + Example: RCR BL, 1
    - If CF = 1 and BL = 10001111 → After rotate: 11000111 (CF = 1)

**4. Double-Precision Shifts (SHLD / SHRD)**

* **SHLD (Shift Left Double)**:
  + Shifts a **destination** left, filling empty bits from a **source** operand.
* **SHRD (Shift Right Double)**:
  + Shifts a **destination** right, filling empty bits from a **source** operand.

**5. Key Uses of Shift/Rotate Instructions**

* **Multiplication/Division**:
  + SHL = Multiply by **2ⁿ**
  + SHR = Divide (unsigned) by **2ⁿ**
  + SAR = Divide (signed) by **2ⁿ**
* **Bit Manipulation**:
  + Extracting bits, setting flags, etc.
* **Sign Extension**:
  + Example: Extending a 16-bit signed number (AX) to 32 bits (EAX):

asm

Copy

mov ax, -128 ; AX = FF80h

shl eax, 16 ; EAX = FF800000h

sar eax, 16 ; EAX = FFFFFF80h (sign extended)

**Summary Table**

| **Instruction** | **Type** | **Operation** | **Effect on CF** |
| --- | --- | --- | --- |
| **SHL** | Logical Left | Bits shift left, LSB=0 | MSB → CF |
| **SHR** | Logical Right | Bits shift right, MSB=0 | LSB → CF |
| **SAR** | Arith. Right | Bits shift right, MSB=sign bit | LSB → CF |
| **ROL** | Rotate Left | Bits rotate left (CF not included) | MSB → CF & LSB |
| **ROR** | Rotate Right | Bits rotate right (CF not included) | LSB → CF & MSB |
| **RCL** | Rotate Left + CF | Like ROL but includes CF in rotation | MSB → CF, CF → LSB |
| **RCR** | Rotate Right + CF | Like ROR but includes CF in rotation | LSB → CF, CF → MSB |

**Final Notes**

* **Logical shifts** (SHL, SHR) work for **unsigned numbers**.
* **Arithmetic shifts** (SAR) work for **signed numbers**.
* **Rotations** (ROL, ROR, RCL, RCR) are useful for **bit-level operations**.
* **Double shifts** (SHLD, SHRD) help with **large-number operations**.

Hope this makes it easier to understand! 😊 Let me know if you need any clarifications.

**Easy Notes: Rotate and Double Shift Instructions (x86 Assembly)**

**1. Rotate Instructions (ROL, ROR, RCL, RCR)**

**(A) ROL (Rotate Left)**

* **Operation**: Bits move left in a circle.
  + The **highest bit (MSB)** goes into:
    - **Carry Flag (CF)**
    - **Lowest bit (LSB)**
* **Example**:

asm

Copy

mov al, 40h ; AL = 01000000b

rol al, 1 ; AL = 10000000b (CF = 0)

rol al, 1 ; AL = 00000001b (CF = 1)

* **Uses**:
  + Swapping upper & lower nibbles (e.g., 26h → 62h after ROL AL, 4).

**(B) ROR (Rotate Right)**

* **Operation**: Bits move right in a circle.
  + The **lowest bit (LSB)** goes into:
    - **Carry Flag (CF)**
    - **Highest bit (MSB)**
* **Example**:

asm

Copy

mov al, 01h ; AL = 00000001b

ror al, 1 ; AL = 10000000b (CF = 1)

ror al, 1 ; AL = 01000000b (CF = 0)

**(C) RCL (Rotate Carry Left)**

* **Operation**: Like **ROL**, but **includes CF in the rotation**.
  + **CF becomes the new LSB**.
* **Example**:

asm

Copy

clc ; CF = 0

mov bl, 88h ; BL = 10001000b

rcl bl, 1 ; BL = 00010000b (CF = 1)

rcl bl, 1 ; BL = 00100001b (CF = 0)

**(D) RCR (Rotate Carry Right)**

* **Operation**: Like **ROR**, but **includes CF in the rotation**.
  + **CF becomes the new MSB**.
* **Example**:

asm

Copy

stc ; CF = 1

mov ah, 10h ; AH = 00010000b

rcr ah, 1 ; AH = 10001000b (CF = 0)

**2. Double-Precision Shifts (SHLD, SHRD)**

**(A) SHLD (Shift Left Double)**

* **Operation**:
  + Shifts **destination** left by count bits.
  + Fills empty bits from the **source operand’s MSBs**.
* **Example**:

asm

Copy

.data

wval WORD 9BA6h

.code

mov ax, 0AC36h

shld wval, ax, 4 ; wval = BA6Ah

**(B) SHRD (Shift Right Double)**

* **Operation**:
  + Shifts **destination** right by count bits.
  + Fills empty bits from the **source operand’s LSBs**.
* **Example**:

asm

Copy

mov ax, 234Bh

mov dx, 7654h

shrd ax, dx, 4 ; AX = 4234h

**3. Key Points**

**(A) Overflow Flag (OF) in Rotations**

* **OF is set if the sign changes after a 1-bit shift/rotate**.
  + Example:

asm

Copy

mov al, +127 ; AL = 01111111b

rol al, 1 ; AL = 11111110b (OF = 1, now negative)

**(B) Uses of SHLD/SHRD**

* **Bit-mapped graphics**: Repositioning pixels.
* **Encryption**: Shuffling bits for security.
* **Large-number math**: Fast multiplication/division.

**(C) Example: Shifting an Array**

asm

Copy

.data

array DWORD 648B2165h, 8C943A29h, 6DFA4B86h, 91F76C04h, 8BAF9857h

.code

mov bl, 4 ; Shift count = 4

mov esi, OFFSET array

mov ecx, (LENGTHOF array) - 1

L1:

push ecx

mov eax, [esi + 4] ; Next DWORD

mov cl, bl

shrd [esi], eax, cl ; Shift right 4 bits

add esi, 4 ; Move to next DWORD

pop ecx

loop L1

shr DWORD PTR [esi], 4 ; Shift last DWORD

**Summary Table**

| **Instruction** | **Operation** | **Effect on CF** |
| --- | --- | --- |
| **ROL** | Rotate left (CF not included) | MSB → CF & LSB |
| **ROR** | Rotate right (CF not included) | LSB → CF & MSB |
| **RCL** | Rotate left + CF | MSB → CF, CF → LSB |
| **RCR** | Rotate right + CF | LSB → CF, CF → MSB |
| **SHLD** | Shift left, fill from source MSBs | Last shifted bit → CF |
| **SHRD** | Shift right, fill from source LSBs | Last shifted bit → CF |

**Final Notes**

* **ROL/ROR** = Simple bit rotation.
* **RCL/RCR** = Rotation with Carry Flag.
* **SHLD/SHRD** = Useful for multi-bit operations.
* **OF** = Warns if sign changes after 1-bit shift.

**1. Which instruction shifts each bit in an operand to the left and copies the highest bit into both the Carry flag and the lowest bit position?**

**Answer: ROL (Rotate Left)**

* It **moves all bits to the left**.
* The **leftmost (highest) bit** goes into the **Carry flag** and also into the **rightmost (lowest) bit** position.

**2. Which instruction shifts each bit to the right, copies the lowest bit into the Carry flag, and copies the Carry flag into the highest bit position?**

**Answer: RCR (Rotate through Carry Right)**

* Bits move to the **right**.
* The **rightmost bit** goes into the **Carry flag**.
* The **Carry flag's value** goes into the **leftmost bit**.

**3. Which instruction performs the following operation (CF = Carry flag)?**

Before: CF, AL = 1 11010101  
After: CF, AL = 1 10101011

**Answer: RCL AL, 1 (Rotate through Carry Left 1 time)**

* This moves **everything left**, including the **Carry flag**.
* The **Carry flag value becomes part of AL**, and the **leftmost bit of AL** becomes the new Carry flag.

**4. What happens to the Carry flag when the SHR AX, 1 instruction is executed?**

**Answer: It stores the last (rightmost) bit of AX into the Carry flag.**

* SHR means **Shift Right**.
* Every bit moves 1 position to the right.
* The **last bit** that falls off goes into the **Carry flag**.

**5. Challenge: Shift the lowest bit of AX into the highest bit of BX without SHRD, then with SHRD.**

**Without SHRD:**

assembly

CopyEdit

; Assume AX and BX already have values

SHL BX, 1 ; Make space in BX by shifting left

AND AX, 1 ; Isolate the lowest bit of AX

OR BX, AX ; Put that bit into the lowest of BX (which is now the highest due to shift)

ROL BX, 1 ; Rotate left to move the new bit to the highest position

**With SHRD:**

assembly

CopyEdit

; SHRD BX, AX, 1 shifts BX right 1 bit and brings in bits from AX

SHLD BX, AX, 15 ; First, shift left to make space

SHRD BX, AX, 1 ; Shift right with AX feeding the bit

**6. Challenge: Calculate parity (even/odd number of 1's) of EAX bits using loop and set Parity Flag**

assembly

CopyEdit

MOV ECX, 32 ; 32 bits to check

XOR EBX, EBX ; EBX = 0, will store number of 1’s

PARITY\_LOOP:

SHR EAX, 1 ; Shift EAX right, LSB into Carry Flag

ADC EBX, 0 ; Add Carry to EBX (only adds 1 if Carry was 1)

LOOP PARITY\_LOOP

; Now EBX has count of 1s

; Set Parity flag (PF = 1 if even number of 1s)

TEST BL, 1 ; Check if the number is even or odd

SETZ AL ; AL = 1 if even (parity), else 0

; AL now reflects the parity

**🌀 7.2 Shift and Rotate Applications - Easy Notes**

**✅ Why Use Shifts and Rotates?**

* We use them to **move bits** around inside a number.
* It helps to bring **specific bits to the front (position 0)** so they’re easy to check or use.
* Very useful in **low-level programming**, especially in assembly language.

**🔄 7.2.1 Shifting Multiple Doublewords**

**💡 What’s a Doubleword?**

* A **doubleword = 4 bytes = 32 bits**.

**📥 How Are Numbers Stored?**

* On **x86 processors**, numbers are stored in **little-endian** format:
  + The **lowest byte (least significant)** is stored **first**.
  + Then the next byte, then the next, up to the **highest byte**.

✅ Example (in bytes):  
If you have a number stored in [ESI], [ESI+1], [ESI+2]

* + [ESI] = lowest byte
  + [ESI+2] = highest byte

**🧮 How to Shift an Array of Bytes 1 Bit to the Right**

Let’s say we have 3 bytes (3 memory locations) stored in [ESI], [ESI+1], and [ESI+2].

**Step-by-Step:**

**Step 1:**

* Start with the **highest byte**: [ESI+2]
* Shift it 1 bit to the **right**.
* Its **last (lowest) bit** goes into the **Carry flag (CF)**.

**Step 2:**

* Now shift [ESI+1] to the **right**.
* Before shifting, take the **Carry flag (CF)** and insert it into the **leftmost bit** of this byte.

**Step 3:**

* Repeat for [ESI], using the new CF from previous step.

So basically:

* **Start from the top** (highest byte) → go down to the lowest.
* After each shift, **pass the lowest bit** into the **CF**.
* **Next byte** grabs that CF and uses it as its **highest bit**.

**📌 Summary Tips:**

| **Term** | **Meaning** |
| --- | --- |
| **Shift Right** | Move all bits one place to the right. |
| **Carry Flag (CF)** | Temporarily holds the bit that was “pushed out”. |
| **Little-endian** | Lowest byte comes first in memory. |
| **[ESI], [ESI+1], [ESI+2]** | 3 bytes in memory – lowest to highest. |

**1. Write assembly instructions to calculate EAX \* 24 using binary multiplication.**

**Easy trick:**  
24 = 16 + 8  
Which means: EAX \* 24 = (EAX \* 16) + (EAX \* 8)

**Assembly code:**

asm

CopyEdit

mov ebx, eax ; Copy EAX to EBX

shl eax, 4 ; Multiply EAX by 16 (shift left 4 times)

shl ebx, 3 ; Multiply EBX by 8 (shift left 3 times)

add eax, ebx ; Add both to get EAX \* 24

**2. Write assembly instructions to calculate EAX \* 21 using binary multiplication.**

**Easy trick:**  
21 = 16 + 4 + 1  
So: EAX \* 21 = (EAX \* 16) + (EAX \* 4) + EAX

**Assembly code:**

asm

CopyEdit

mov ebx, eax ; Copy EAX to EBX

shl eax, 4 ; EAX = EAX \* 16

mov ecx, ebx

shl ecx, 2 ; ECX = EAX \* 4

add eax, ecx ; Add EAX + (EAX \* 4)

add eax, ebx ; Add the original EAX to get EAX \* 21

**3. What change to make in BinToAsc procedure to display binary in reverse order?**

**Easy answer:**  
Instead of printing the **highest bit first**, print the **lowest bit first**.

**How?**

* Change the loop so it starts from **bit 0** and goes to **bit 15** (instead of the other way).
* Reverse the bit-processing direction.

**4. How to extract only the minutes from a file timestamp and save it to a byte variable?**

**Explanation:**

* Minutes are stored in **bits 5 to 10** (6 bits total).
* So, we need to:
  1. Shift right by 5 bits (to bring bits 5–10 to the front)
  2. Use AND with 00111111 (binary) or 3Fh (hex) to remove extra bits

**Assembly code:**

asm

CopyEdit

mov ax, timestamp ; Load the timestamp

shr ax, 5 ; Shift right 5 bits to bring minutes to lowest bits

and ax, 3Fh ; Mask to keep only 6 bits (minutes)

mov bMinutes, al ; Store the minutes in bMinutes

**📚 7.3 Multiplication and Division Instructions – Easy Notes**

**🔁 What This Section Is About**

* In assembly (low-level) language, you can **multiply and divide** numbers directly using special instructions.
* These instructions can handle **8-bit, 16-bit, 32-bit**, and even **64-bit** numbers.

**✖️ Multiplication Instructions**

**👉 MUL = Unsigned Multiplication**

* Use this when you're multiplying **positive numbers** (unsigned).
* There are **three versions** in 32-bit mode:

| **Operand Size** | **Multiply With** | **Result Stored In** |
| --- | --- | --- |
| 8-bit | AL | AX |
| 16-bit | AX | DX:AX |
| 32-bit | EAX | EDX:EAX |

📝 DX:AX means the result is split: the **first part in AX**, the **second part in DX**  
📝 EDX:EAX = same idea, but for 32-bit

✅ Format:

asm

CopyEdit

MUL reg/mem8

MUL reg/mem16

MUL reg/mem32

* reg/mem means: You can multiply a **register or a memory location**, but **not a direct number** like 5 or 10 (not an "immediate value").

**🔄 How MUL Works**

* You only give **one operand** to MUL.
* It **automatically uses a specific register** as the other number:
  + AL (for 8-bit), AX (for 16-bit), EAX (for 32-bit)
* The result is **twice as big** as the original numbers.

**⚠️ Flags Set by MUL**

* **Carry Flag (CF)** and **Overflow Flag (OF)** are both set **if the top half of the result is not zero**.
* Why? Because the result couldn’t fit in just the lower half = **overflow happened**.

✅ Example:  
If you multiply two 16-bit numbers and the result is too big to fit into AX, the **extra part goes to DX**, and **CF and OF are set**.

**➗ Division Instructions (Intro)**

* DIV = Unsigned Division
* IDIV = Signed Division (for positive & negative numbers)  
  (More details probably come in the next parts of the section.)

**🕒 7.3.3 Measuring Program Execution Time (Simple Notes)**

**✅ Why Measure Execution Time?**

* Programmers often want to **compare which version of code runs faster**.
* Measuring how long a part of a program takes helps improve performance.

**🧰 Tool Used**

* **GetMseconds**: A **built-in function** from the **Irvine32 library**.
* It tells you how many **milliseconds** have passed since **midnight**.
* Works using the **Windows API** behind the scenes.

**🧪 How It Works (Step-by-Step)**

1. **Call GetMseconds** before running the code you want to test.
   * This records the **start time**.
2. **Run the code** (like a procedure or function).
3. **Call GetMseconds again** after the code runs.
   * This gives the **current time**.
4. **Subtract** the start time from the current time.
   * This gives you the **total time** the code took to run.
5. **Save** the result (execution time) into a variable.

**🧾 Sample Code with Comments**

asm

CopyEdit

.data

startTime DWORD ? ; stores the starting time

procTime1 DWORD ? ; stores the time of first code

procTime2 DWORD ? ; stores the time of second code

.code

call GetMseconds

mov startTime, eax ; Save start time before running code

call FirstProcedureToTest ; Run the first procedure

call GetMseconds

sub eax, startTime ; Calculate how long it took

mov procTime1, eax ; Store result in procTime1

Repeat similar steps for the **second procedure** you want to compare.

**📌 Important Notes**

* **Calling GetMseconds takes a little bit of time**, but it's **so small** that it doesn’t affect results much.
* This method is useful for **comparing two pieces of code** to see which one is **faster**.

**🧠 Topic: Implementing Arithmetic Expressions (7.3.6)**

**📌 What is this section about?**

* It's about using **multiplication and division** in assembly code to write math expressions.
* Previously (Chapter 4), we only used **addition and subtraction**.
* Now we can use **all 4 operations** like in regular math formulas.

**🛠️ Why learn this in assembly?**

* It helps you **understand how compilers work**.
* You can see **how C++ code turns into assembly code**.
* You learn to **handle big numbers properly** (like checking if results are too big for 32 bits).

**⚠️ Warning about compilers:**

* High-level languages like C++ often **ignore extra bits** in multiplication (upper 32 bits).
* But in assembly, **you can detect overflow** using **Carry and Overflow flags**.
  + These tell you if the result didn’t fit properly.
  + Flags are explained in **Sections 7.4.1 and 7.4.2**.

**💡 Tip to See Assembly from C++:**

You can see the assembly version of C++ code using these two methods:

1. **In Visual Studio** – Right-click while debugging ➝ Click **Go to Disassembly**.
2. **Generate a listing file**:
   * Go to **Project ➝ Properties**.
   * Under **Microsoft Macro Assembler**, choose:
     + Generate Preprocessed Source Listing → Yes
     + List All Available Information → Yes

**✅ Example – C++ to Assembly**

C++ code:

cpp

CopyEdit

var4 = (var1 + var2) \* var3;

Assembly logic:

1. Add var1 and var2
2. Multiply the result by var3
3. Store the final result in var4

📝 This is **straightforward** because you follow the math **from left to right** (add first, then multiply).

**1. Why can't overflow happen with MUL and one-operand IMUL?**

* Because the **result (product)** is stored in **double the size** of the operand.
* Example: If you multiply 16-bit numbers, the result goes into a 32-bit space (DX:AX).
* So, there’s **enough room** to store the full result, and no overflow happens.

**2. How is one-operand IMUL different from MUL?**

* **MUL** does **unsigned multiplication** (only positive numbers).
* **IMUL** does **signed multiplication** (can handle negative numbers too).
* IMUL also checks for signs and may set flags based on result's sign.

**3. When does one-operand IMUL set the Carry and Overflow flags?**

* When the **final result is too big** to fit in the **lower half** of the result registers.
* For example, if a 16-bit result can’t fit into **AX** alone, the flags are set.

**4. In a DIV instruction with EBX as the operand, where is the quotient stored?**

* The **quotient (answer)** is stored in **EAX**.
* **EDX** holds the **remainder**.

**5. In a DIV instruction with BX as the operand, where is the quotient stored?**

* The **quotient** is stored in **AX**.
* **DX** will have the **remainder**.

**6. If BL is the operand in a MUL instruction, which registers hold the result?**

* The product (result) is stored in **AX**.
* AL × BL = result in AX (AL = low byte, AH = high byte).

**7. Example of sign extension before using IDIV with a 16-bit operand**

Before doing signed division with a 16-bit number using **IDIV**, you must extend the sign of **AX** into **DX**.

Here’s how:

asm

CopyEdit

cwd ; sign-extend AX into DX (needed before IDIV with 16-bit divisor)

idiv bx ; divide DX:AX by BX (signed division)

* **cwd** copies the **sign bit** of AX into DX so the processor knows if it's negative or positive.

**🧠 7.4 Extended Addition and Subtraction**

**📌 What is Extended Precision Addition/Subtraction?**

* It's a way to **add or subtract very large numbers** (more than 32 or 64 bits).
* In C++, you **cannot** directly add huge numbers like **1024-bit integers**.
* But in **Assembly Language**, you can do this using:
  + ADC = **Add with Carry**
  + SBB = **Subtract with Borrow**

**🔸 7.4.1 ADC Instruction (Add with Carry)**

**✅ What does ADC do?**

* ADC **adds 3 things** together:
  1. Destination operand (where the result goes)
  2. Source operand (value being added)
  3. **Carry flag** (used when there's a leftover from the previous addition)

**🔧 Formats of ADC (Operands must be same size):**

asm

CopyEdit

ADC reg, reg ; register + register

ADC mem, reg ; memory + register

ADC reg, mem ; register + memory

ADC mem, imm ; memory + immediate value

ADC reg, imm ; register + immediate value

**🔍 Example Explained:**

asm

CopyEdit

mov dl, 0 ; set DL to 0

mov al, 0FFh ; set AL to FFh (255 in decimal)

add al, 0FFh ; AL = AL + FFh → AL becomes FEh (254), Carry is set

adc dl, 0 ; DL = DL + 0 + Carry → DL becomes 1

Now you get a **16-bit sum** stored in DL:AL = 01FEh.

* This is how you can **combine 8-bit registers to store bigger values**.

**📊 What Happens Internally?**

* First: AL gets FFh + FFh = FEh and sets Carry flag.
* Second: DL gets 0 + Carry = 1
* Final Result: DL:AL = 01FEh

**💡 Summary:**

**1. Describe the ADC instruction.**

* **ADC** stands for **Add with Carry**.
* It adds three things:
  1. The value in the **destination operand**.
  2. The value in the **source operand**.
  3. The **Carry flag** (which holds a leftover value from a previous operation).
* It’s used for **adding large numbers** that might require more than one operation or register.

**2. Describe the SBB instruction.**

* **SBB** stands for **Subtract with Borrow**.
* It subtracts three things:
  1. The value in the **destination operand**.
  2. The value in the **source operand**.
  3. The **Carry flag** (which represents a borrow from a previous subtraction).
* It’s used for **subtracting large numbers**, similar to ADC, but when there is a borrow involved.

**3. What will be the values of EDX:EAX after the following instructions execute?**

asm

CopyEdit

mov edx,10h ; EDX = 10h (16 in decimal)

mov eax,0A0000000h ; EAX = A0000000h (2684354560 in decimal)

add eax,20000000h ; EAX = EAX + 20000000h = C0000000h (3221225472 in decimal)

adc edx,0 ; EDX = EDX + 0 + Carry flag = 10h (16 in decimal) since Carry = 0

* **Final Values**:
  + **EDX = 10h** (16 in decimal)
  + **EAX = C0000000h** (3221225472 in decimal)

**4. What will be the values of EDX:EAX after the following instructions execute?**

asm

CopyEdit

mov edx,100h ; EDX = 100h (256 in decimal)

mov eax,80000000h ; EAX = 80000000h (2147483648 in decimal)

sub eax,90000000h ; EAX = EAX - 90000000h = FFFFFFFFh (4294967295 in decimal)

sbb edx,0 ; EDX = EDX - 0 - Carry flag = 100h (256 in decimal) since Carry = 0

* **Final Values**:
  + **EDX = 100h** (256 in decimal)
  + **EAX = FFFFFFFFh** (4294967295 in decimal)

**5. What will be the contents of DX after the following instructions execute (STC sets the Carry flag)?**

asm

CopyEdit

mov dx,5 ; DX = 5

stc ; Set the Carry flag (Carry = 1)

mov ax,10h ; AX = 10h (16 in decimal)

adc dx,ax ; DX = DX + AX + Carry flag = 5 + 16 + 1 = 22h (34 in decimal)

* **Final Value of DX**:
  + **DX = 22h** (34 in decimal)

**Notes on ASCII and Unpacked Decimal Arithmetic (Easy to Understand)**

**Introduction**

* **32-bit Mode**: The instructions discussed in this section apply when programming in 32-bit mode.
* **Binary vs. ASCII**: The CPU calculates in binary, but it can also handle arithmetic with ASCII (text) decimal numbers. ASCII is useful because it's easy for users to input and display on the screen without converting to binary.

**Arithmetic on ASCII Decimal Strings**

* Example: You can input two numbers, like 3402 and 1256, and get the sum (4658) without needing to convert them to binary.

**Two Ways to Handle ASCII Arithmetic:**

1. **Convert to Binary First**: Convert both numbers to binary, add them, then convert the result back to ASCII.
2. **Directly Add ASCII Digits**: Add each digit directly, adjusting the result after each addition.

**Four Key Instructions for ASCII Arithmetic:**

1. **AAA**: Adjusts after an ASCII addition.
2. **AAS**: Adjusts after an ASCII subtraction.
3. **AAM**: Adjusts after an ASCII multiplication.
4. **AAD**: Adjusts before performing an ASCII division.

**ASCII and Unpacked Decimal**

* **Unpacked Decimal**: Each byte stores one decimal digit.
* **ASCII Decimal**: The high 4 bits are 0011b, and each byte stores one ASCII character representing a digit.

Example:

* ASCII Format: 33 34 30 32 (This is 3402 in ASCII)
* Unpacked Format: 03 04 00 02 (This is 3402 in unpacked format)

**Advantages of ASCII Arithmetic:**

1. **No Conversion Needed**: You don't need to convert strings to binary before arithmetic.
2. **Accuracy**: Since ASCII uses an assumed decimal point, it avoids floating-point rounding errors.

**Types of Operations in ASCII Arithmetic**

* **Addition and Subtraction**: Can use ASCII or unpacked decimal.
* **Multiplication and Division**: Only unpacked decimal numbers can be used.

**Instructions for ASCII Arithmetic:**

**1. AAA (ASCII Adjust After Addition)**

* This instruction adjusts the result of an addition of ASCII digits.
* It works by moving the result into two unpacked decimal digits.

Example:

asm

CopyEdit

mov ah, 0

mov al, '8'

add al, '2'

aaa ; Adjust AL for ASCII

or ax, 3030h ; Convert result to ASCII

After AAA, the result is converted to ASCII and stored in AX.

**2. AAS (ASCII Adjust After Subtraction)**

* **AAS** is used after a subtraction (SUB or SBB) to adjust the result of the ASCII subtraction.
* It handles negative results, adjusting the carry and ensuring the result is in the correct ASCII form.

Example:

asm

CopyEdit

mov ah, 0

mov al, '8'

sub al, '9'

aas ; Adjust AL for ASCII

This converts the result (which might be negative) to the correct ASCII representation.

**3. AAM (ASCII Adjust After Multiplication)**

* The **AAM** instruction is used after multiplying two unpacked decimal numbers.
* It converts the binary result of multiplication into an unpacked decimal.

Example:

asm

CopyEdit

mov bl, 05h

mov al, 06h

mul bl ; Multiply

aam ; Adjust the result to unpacked decimal

After AAM, the result in AX is the unpacked decimal form of the multiplication.

**4. AAD (ASCII Adjust Before Division)**

* The **AAD** instruction prepares an unpacked decimal value in AX for division.
* It converts the unpacked decimal to binary before performing a division operation.

Example:

asm

CopyEdit

mov ax, 0307h

aad ; Adjust for division

mov bl, 5

div bl ; Perform division

After AAD, the division is done, and the result is in AL (quotient) and AH (remainder).

**Summary**

* **ASCII Arithmetic** allows handling decimal strings (like 3402 or 1256) directly, which is useful for user-friendly input/output.
* Instructions like **AAA, AAS, AAM, and AAD** help adjust the results of addition, subtraction, multiplication, and division when working with ASCII decimal numbers.
* This method avoids the need to convert strings to binary, making it simpler to work with decimal numbers directly in programs.

**Section Review (Easy to Understand)**

1. **Convert a Two-Digit Unpacked Decimal Integer in AX to ASCII Decimal:**

**Instruction:**

* + **AAM**: This converts the binary number in AX to an unpacked decimal format.

**Explanation**: The AAM instruction adjusts the result of the multiplication, turning the binary value in AX into unpacked decimal digits.

asm

CopyEdit

aam ; Converts the unpacked decimal in AX to ASCII decimal

1. **Convert a Two-Digit ASCII Decimal Integer in AX to Unpacked Decimal:**

**Instruction:**

* + **AAD**: This converts an ASCII decimal number in AX to a binary (unpacked decimal) format.

**Explanation**: The AAD instruction adjusts the result of an ASCII value in AX to be used in further binary operations, converting it into an unpacked decimal format.

asm

CopyEdit

aad ; Converts the ASCII decimal number in AX to unpacked decimal

1. **Convert a Two-Digit ASCII Decimal Number in AX to Binary:**

**Instructions:**

* + **AAD**: First, adjust the ASCII decimal to unpacked decimal.
  + **DIV**: Then divide the unpacked decimal by 1 or another divisor to get the binary result.

**Explanation**: First, use AAD to convert the ASCII decimal number into unpacked decimal. Then, perform a division to get the result in binary.

asm

CopyEdit

aad ; Convert ASCII decimal to unpacked decimal

div bl ; Divide by a number to get the binary result

1. **Convert an Unsigned Binary Integer in AX to Unpacked Decimal:**

**Instruction:**

* + **AAM**: This converts a binary value in AX to an unpacked decimal number.

**Explanation**: The AAM instruction adjusts the binary value into unpacked decimal format in AX.

asm

CopyEdit

aam ; Converts an unsigned binary integer in AX to unpacked decimal

**Packed Decimal Arithmetic Notes (Easy Explanation)**

Packed decimal is a way to store decimal numbers where each byte holds two decimal digits, and each digit is represented by 4 bits. If there's an odd number of digits, the highest nibble (half of a byte) is filled with zero. Here's how packed decimals are stored:

* **Examples** of packed decimal storage:
  + bcd1 QWORD 2345673928737285h = 2,345,673,928,737,285 (decimal)
  + bcd2 DWORD 12345678h = 12,345,678 (decimal)
  + bcd3 DWORD 08723654h = 8,723,654 (decimal)
  + bcd4 WORD 9345h = 9,345 (decimal)
  + bcd5 WORD 0237h = 237 (decimal)
  + bcd6 BYTE 34h = 34 (decimal)

**Benefits of Packed Decimal:**

1. **Accuracy**: Packed decimal numbers can store many digits, which makes calculations very accurate.
2. **Simple Conversion**: Converting packed decimal numbers to ASCII (and vice versa) is easy.

**DAA (Decimal Adjust After Addition) Instruction:**

* **Purpose**: In 32-bit mode, DAA converts a binary sum (from ADD or ADC instructions) into a packed decimal format.
* **Example**:
  + Add packed decimal 35 and 48.
  + The sum in binary is 7Dh, but after applying DAA, it converts into the packed decimal result 83h.

**Code Example**:

asm

CopyEdit

mov al,35h ; Load 35 into AL

add al,48h ; Add 48 to AL

daa ; Adjust result to packed decimal

**Result**: AL will be 83h after adjustment.

* **Packed Decimal Addition Example**: A program that adds two 16-bit packed decimal numbers:

asm

CopyEdit

; Packed Decimal Example (AddPacked.asm)

; Demonstrate packed decimal addition.

INCLUDE Irvine32.inc

.data

packed\_1 WORD 4536h

packed\_2 WORD 7207h

sum DWORD ?

.code

main PROC

; Initialize sum and index.

mov sum,0

mov esi,0

; Add low bytes.

mov al,BYTE PTR packed\_1[esi]

add al,BYTE PTR packed\_2[esi]

daa

mov BYTE PTR sum[esi],al

; Add high bytes, include carry.

inc esi

mov al,BYTE PTR packed\_1[esi]

adc al,BYTE PTR packed\_2[esi]

daa

mov BYTE PTR sum[esi],al

; Add final carry, if any.

inc esi

mov al,0

adc al,0

mov BYTE PTR sum[esi],al

; Display the sum in hexadecimal.

mov eax,sum

call WriteHex

call Crlf

exit

main ENDP

END main

**Explanation**: This program adds packed decimal numbers in packed\_1 and packed\_2, adjusts the sum using daa, and stores the result in sum.

**DAS (Decimal Adjust After Subtraction) Instruction:**

* **Purpose**: In 32-bit mode, DAS converts the result of a subtraction (from SUB or SBB instructions) in AL to packed decimal format.
* **Example**:
  + Subtract packed decimal 48 from 85.
  + The result in binary is 3Dh, but after applying DAS, it converts into the packed decimal result 37h.

**Code Example**:

asm

CopyEdit

mov bl,48h ; Load 48 into BL

mov al,85h ; Load 85 into AL

sub al,bl ; Subtract 48 from 85

das ; Adjust result to packed decimal

**Result**: AL will be 37h after adjustment.

**Summary:**

* **Packed Decimal** stores two decimal digits per byte, making it efficient for handling decimal values.
* **DAA** is used after addition to convert binary results to packed decimal.
* **DAS** is used after subtraction to convert binary results to packed decimal.
* Packed decimal arithmetic is very useful for high-accuracy calculations in decimal format.

**7.6.3 Section Review (Easy Explanation)**

1. **When does the DAA instruction set the Carry flag?**
   * **Answer**: The DAA (Decimal Adjust After Addition) instruction sets the **Carry flag** if the addition of the two packed decimal numbers results in a carry (meaning the sum is greater than what can be stored in the current byte).
   * **Example**:  
     Let's add two packed decimals:

asm

CopyEdit

mov al, 99h ; Load 99 (decimal 99) into AL

add al, 99h ; Add 99 (decimal 99) to AL

daa ; Adjust result to packed decimal

* + - The sum of 99h + 99h = 132h in binary, which is greater than what can fit in one byte (0-99 range in packed decimal). Therefore, DAA sets the Carry flag.

1. **When does the DAS instruction set the Carry flag?**
   * **Answer**: The DAS (Decimal Adjust After Subtraction) instruction sets the **Carry flag** if the subtraction results in a borrow (meaning the subtraction goes below zero in packed decimal).
   * **Example**:  
     Let's subtract two packed decimals:

asm

CopyEdit

mov al, 50h ; Load 50 (decimal 50) into AL

mov bl, 60h ; Load 60 (decimal 60) into BL

sub al, bl ; Subtract 60 from 50

das ; Adjust result to packed decimal

* + - The result of 50h - 60h is a negative value, which triggers the Carry flag to indicate a borrow.

1. **When adding two packed decimal integers of length n bytes, how many storage bytes must be reserved for the sum?**
   * **Answer**: When adding two packed decimal integers of length **n bytes**, you need to reserve **n + 1** bytes for the sum.  
     This is because the sum could require one additional byte to store the carry if the result is too large to fit in the original space.
     + **Example**:  
       If you are adding two packed decimal integers, each 2 bytes long, the sum requires **3 bytes** to store the result.