AUBURN

UNIVERSITY

The Instruction Set (Part I)

Module Overview

Objectives

- Learn, understand, and use arithmetic instructions
- Learn, understand, and use Boolean instructions
- Learn, understand, and use shift/rotate instructions

Requirements

- Fluent in the binary and hexadecimal systems
- Know the x86 registers
- Know the MOV instruction
- Know and understand the x86 addressing modes

Key (Software) Characteristics of Any CPU

- **Registers (Previous Module)**
  - *Internal* cell storage. Access time much smaller than memory cells. Fastest storage in the computer system. **For speed, favor using registers over memory.**

- **Addressing modes (Previous Module)**
  - ways of specifying **operands** for the instructions.

- **Instruction set**
  - The instruction set will be divided in two parts:
    - **Part I** : arithmetic, Boolean, and shift/rotate instructions (**this module**)
    - **Part II**: branches and loops (**next module**)

Instruction Set of the 32-bit x86 Processor

- **Part I:**
  - **Data transfer** instructions
  - **Arithmetic** instructions (addition, subtraction, and multiplication)
  - **Boolean** instructions (NOT, AND, OR, XOR)
  - **Shift/Rotation** instructions (SHL, SHR, SAR, ROL, ROR, RCL, RCR)

- **Part II:**
  - **Branches** (conditional or unconditional jumps)
    - Comparison instructions
    - Conditional branches
    - Loop instruction
    - Unconditional jumps
    - Procedure calls (return)

Instruction Set (Part I: This module)

- **Data transfer** instructions: **MOV**
- **Arithmetic** instructions (**INC**, **DEC**, **ADD**, **ADC**, **SUB**, **MUL**)
- **Boolean** (logic) instructions (**NOT**, **AND**, **OR**, **XOR**)
- **Shift/Rotation** instructions (**SHL**, **SHR**, **SAR, ROL**, **ROR**, **RCL**, **RCR**)

- **NEXT MODULE (Part II):**
  - **Branches** (conditional or unconditional jumps)
    - Comparison instructions
    - Conditional branches
    - Loop instruction
    - Unconditional jumps
    - Procedure calls (return)

Data Transfer: MOV (Review)

- Our first instruction, a **data transfer** instruction: **MOV** *Destination*, *Source*
  - **Mnemonic**: MOV
  - **Operands**: Destination and Source
  - **Function**: Destination ← Source
    - moves *Source* (or **content** of *Source*) into *Destination*
    - Source remains **unchanged**
    - Destination and Source must have the same size (in bits)
  - **Examples:**

    MOV EAX, 1

    MOV AX, DX

    MOV CL, DH

    MOV var1, EBX

  - **Exception:** Source and Destination cannot be BOTH **memory** operands for the same MOV instruction. For example, these instructions **CANNOT** be executed:
    - MOV var1, var2 ; where var1 and var2 are memory operands
    - MOV var1,[ESI]
    - MOV [EDI],var2

Instructions Effects

- **Most** instructions have in general two effects:
  - They modify the **destination** operand
  - They may modify the **EFLAGS** register

- **EFLAGS Review**:

- The EFLAGS register contains multiple bits (called flags) that provide information about the latest result produced by the ALU. Here are some **key** flags:
  - **Carry** flag (**CY** bit $EFLAGS_0$): this bit is set to 1 if the latest operation (e.g., an addition) produced a carry
  - **Zero** flag (**ZR** bit $EFLAGS_6$); this bit is set to 1 if the latest instruction produced a **null** result (i.e., zero).
  - **Sign** flag (**PL** bit $EFLAGS_7$): this bit is set to 1 if the latest operation produced a negative result
  - There are other flags: we will ignore them for now

EFLAGS

Weight	…	7	6	4	5	3	2	1	0
Flag		PL	ZR						CY

Carry Flag (*CY*)

- The **Carry** flag is the least significant bit $EFLAGS_0$ in the register EFLAGS
- The register EFLAGS is called **EFL** in *Visual Studio* (Rightmost register in the registers' window)
- Example:

```
MOV  EAX,0FFFFFFFFh
ADD  EAX,1        ;   EAX ← EAX + 1
```

This means the result is **1**00000000h which needs 33 bits. EAX is a 32-bit register that cannot contain the value **1**00000000h(EAX will contain 0000 0000h and ***CY*** = **1**)

EFLAGS

Weight	…	7	6	4	5	3	2	1	0
Flag									**CY=1**

Zero Flag (**ZR**)

- The **Zero** flag is Bit $EFLAGS_6$ in the register EFLAGS
- **Example**:

```
MOV AL, 12h
SUB AL, AL;  AL ← AL – AL (i.e., AL = 00h)
```

This means the result is 00h (the ALU produced a null result, therefore ***ZR*** ($EFLAGS_6$) is set to 1)

EFLAGS

Weight	…	7	6	4	5	3	2	1	0
Flag			**ZR= 1**						

Sign Flag (***PL***)

- The **Sign** flag is Bit $EFLAGS_7$ in the register EFLAGS
- **Example**:

```
MOV AL, 00h
SUB AL, 1;  AL ← AL – 1 (AL = -1 = FFh, i.e., *MSB is 1)
```

This means the result -1 is negative: the sign bit is 1 then ***PL*** ($EFLAGS_7$) is set to 1)

*Note that the Carry flag will also be set (check it with Visual Studio)

***MSB**: Most Significant Bit

EFLAGS

Weight	…	7	6	4	5	3	2	1	0
Flag		PL=1							CY=1

Arithmetic Instructions

- INC
- DEC
- ADD
- ADC
- SUB
- MUL

Arithmetic Instruction: INC

INC *operand*

- **Mnemonic**: INC
- **Operand**: : *operand* can be a register or memory operand
- **Function**: Operand ← Operand **+** 1
  - Increments the operand
  - May affect the zero (***ZR***) and sign (***PL***) flags, but **NOT** the carry (***CY***) flag

- **Example:**

```
MOV AL, 0FFh          ; AL ← FFh
INC AL                ; AL ← 00h PL= 0,  ZR = 1, and CY = 0
                      ; (CY is not set even though the result is 100h)
```

```
*Recall PL = Sign Flag and ZR = Zero Flag
```

Arithmetic Instruction: DEC

DEC *operand*

- **Mnemonic**: DEC
- **Operand**: *operand* can be a register or memory operand
- **Function**: Operand ← Operand **–** 1
  - Decrements the operand
  - May affect the zero (***ZR***) and sign (***PL***) flags, but **NOT** the carry (***CY***) flag

- **Example:**

```
MOV AL, 00h            ; AL ← 00h
DEC AL                 ; AL ← AL – 1 = FFh,  PL = 1, ZR = 0, and CY = 0
```

Arithmetic Instruction: ADD

ADD *Destination, Source*

- **Mnemonic**: ADD
- **Operands**:
  - *Destination* can be a register or a memory
  - *Source* can be a register, a memory, or an **immediate**.
- **Function**: Destination ← Destination **+** Source
  - Sum of Destination and Source is stored in Destination
  - Source unchanged
  - May affect the carry (**CY**), zero (**ZR**), and sign (**PL**) flags.

- **Example:**

```
MOV AL, 0FFh ; AL ← FFh
ADD AL, 1    ; AL ← AL + 1 = 100h, therefore AL = 00h, CY = 1, PL = 0, and ZR = 1
```

Arithmetic Instruction: ADC

- **ADC** *Destination, Source*
  - **Mnemonic**: ADC
  - **Operands**:
    - *Destination* can be a register or a memory
    - *Source* can be a register, a memory, or an immediate.
  - **Function**: Destination ← Destination **+** Source **+** Carry Flag
    - Sum of Destination, Source, and Carry Flag is stored in Destination
    - Source unchanged
    - May affect the carry (**CY**), zero (**ZR**), and sign (**PL**) flags.

  - **Example 1:**

```
MOV AX, 00FFh;  AX ← 00FFh
ADD AL, 1    ;  AL ← AL + 1 = 100h   : AL = 00h, CY = 1, PL = 0, and ZR = 1
ADC AH, 0    ;  AH ← AH + 0 + C = 1 : AH = 01h, CY = 0, PL = 0, and ZR = 0
```

Arithmetic Instruction: ADC

- **Example 2 (ADD EAX, EBX)**
  - The addition of two 32-bit numbers may produce a **33**-bit number
  - For example, let EAX = 8000 0000h and EBX = 8000 0000h
  - The sum of EAX and EBX will result in **1** 0000 0000h which is a 33-bit number
  - With 33 bits, the number **1** 0000 0000h cannot be stored in EAX
  - The sum of EAX and EBX could be stored in EDX and EAX (EDX:EAX), i.e., EAX will store the least significant bits and EDX will store the most significant bit.
  - Below is the code to compute EAX + EBX and store the sum in EDX:EAX

```
MOV EDX, 0h
ADD EAX, EBX    ; EAX ← EAX + EBX
ADC EDX, 0h     ; EDX ← EDX + 0 + Carry
                ; Carry would be 1 if ADD produced a carry)
```

		1000 0000 0000 0000	EAX
		+	
EDX	0000 0000 0000 0000	1000 0000 0000 0000	EBX
CY	1 +		
	0000 0000 0000 0001	0000 0000 0000 0000	
	EDX	EAX	

Arithmetic Instruction: SUB

SUB *Destination, Source*

- **Mnemonic**: SUB
- **Operands**:
  - *Destination* can be a register or a memory
  - *Source* can be a register, a memory, or an immediate.
- **Function**: Destination ← Destination - Source
  - Difference of Destination and Source is stored in Destination
  - Source unchanged
  - May affect the carry (**CY**) , zero (**ZR**), and sign (**PL**) flags.

- **Example:**

```
MOV AL, 00h    ; AL ← 00h
SUB AL, 1      ; AL ← AL - 1 = FFh    : AL = FFh, CY = 1, PL = 1, and ZR = 0
```

Arithmetic Instruction: MUL

MUL *Multiplier*

- **Mnemonic**: MUL
- **Operands**:
  - *Multiplier* can be a register or a memory of size 8 bits, 16 bits, or 32 bits
  - *Multiplicand is **IMPLICITLY** determined by the **size** of the Multiplier:*
    - *(8 bits → AL), (16 bits → AX), or (32 bits → EAX)*
  - ***Destination** (product) is **IMPLICITLY** determined by the **size** of the Multiplier:*
    - *(8 bits → AX), (16 bits → DX:AX), or (32 bits → EDX:EAX)*

Multiplier (Size)	Multiplicand	Product
Register (8) / Memory (8)	AL	AX
Register (16) / Memory (16)	AX	DX:AX
Register (32) / Memory (32)	EAX	EDX:EAX

- **Function**: Product ← Multiplicand * Multiplier (unsigned)
  - Product of *Multiplicand* and *Multiplier* is stored in AX, DX:AX, or EDX:EAX depending on the size of the multiplier
  - Multiplier may get changed (multiplier is AL, AX, or EAX)
  - May affect the carry (**CY**) : ***CY*** is **set** to 1 if upper half of product is not null.

Arithmetic Instruction: MUL (Example 1)

Example 1:

MOV EAX, 10203040h

MOV EBX, 40302010h

MUL BL ; Based on the table, since the multiplier BL is **8** bits, the multiplicand is **AL**
; and the product is stored in **AX**

; AX ← AL * BL = 40h * 10h = 0400h then AX = 0400h

; Upper half of product is 04h (AH). AH is not null, then carry CY is set to 1

; Observe that Multiplicand (AL) is modified.

Multiplier (Size)	Multiplicand	Product
Register (8) / Memory (8)	**AL**	**AX**
Register (16) / Memory (16)	AX	DX:AX
Register (32) / Memory (32)	EAX	EDX:EAX

Arithmetic Instruction: MUL (Example 11)

Example 11:

MOV EAX, 1020**3040**h

MOV EBX, 4030**2010**h

MUL BX ; Based on the table, since the multiplier **BX** is **16** bits, the multiplicand will be **AX**
; and the product will be stored in **DX:AX**

; DX:AX ← AX * BX = 3040h * 2010h = 60B 0400h then DX = 060Bh AX = 0400h

; Upper half of product is 060Bh (DX). DX is not null, then carry CY is set to 1

; Observe that Multiplicand (AX) is modified.

Multiplier (Size)	Multiplicand	Product
Register (8) / Memory (8)	AL	AX
Register (16) / Memory (16)	**AX**	**DX:AX**
Register (32) / Memory (32)	EAX	EDX:EAX

Arithmetic Instruction: MUL (Example III)

Example III:

MOV EAX, **10203040**h

MOV EBX, **40302010**h

MUL EBX ; Based on the table, since the multiplier **EBX** is **32**bits, the multiplicand is **EAX**
; and the product will be stored in **EDX:EAX**

; EDX:EAX ← EAX * EBX = 10203040h * 40302010h = 40B141E 140B0400h

; then EDX = 40B141Eh EAX = 140B0400h

; Upper half of product is 40B141Eh (EDX). EDX is not null, then carry CY is set to 1

Multiplier (Size)	Multiplicand	Product
Register (8) / Memory (8)	AL	AX
Register (16) / Memory (16)	AX	DX:AX
Register (32) / Memory (32)	**EAX**	**EDX:EAX**

Boolean Instructions

- NOT
- AND
- OR
- XOR

- **Key point**: whenever **possible**, use Boolean bitwise instructions instead of arithmetic instructions. In general, Boolean instructions are faster.

Boolean Instruction: NOT

NOT *Operand*

- **Mnemonic**: NOT
- **Operands**:
  - *Operand* can be a register or a memory
- **Function**: Operand ← ~Operand (bitwise NOT)
  - Bitwise one-complement is stored in Operand
  - No flags are affected.
- **Example:**

```
MOV AL, 65h    ;  AL ← 65h   = (0110 0101)₂
NOT AL         ;  AL ← ~AL   = (1001 1010)₂ = ~(0110 0101)₂ =9Ah
```

Boolean Instruction: AND

AND *Destination, Source*

- **Mnemonic**: AND
- **Operands**:
  - *Destination* can be a register or a memory
  - *Source* can be a register, a memory, or an immediate.
- **Function**: Destination ← Destination **&** Source
  - Bitwise **AND** of Destination and Source is stored in Destination
  - Source unchanged
  - AND clears the carry (**CY ← 0**)
  - AND may affect the zero (**ZR**) and sign (**PL**) flags.

- **Example:**

```
MOV AL, 32h   ;
AND AL, 0Fh   ;
```

; AL ← 32h = $(0011\ 0010)_2$

; AL ← AL & 0Fh = 32h & $(0000\ 1111)_2$ = **02h** *CY* = 0, *PL* = 0, and *ZR* = 0

Boolean Instruction: AND (Usage) 1/2

Use **AND** to convert the ASCII code of a decimal digit into the value of the digit

Digit	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII	30h	31h	32h	33h	34h	35h	36h	37h	38h	39h
Value	0	1	2	3	4	5	6	7	8	9

- Suppose that the register CL contains the ASCII code of a digit, let us convert the ASCII code into a value.
- Based on the table, the rightmost hexadecimal digit of the ASCII code is the value of the digit: for example, the ASCII code of '6' is 36h which corresponds to the value 6 (rightmost digit of 36h).
- The rightmost digit is made of the 4 least significant bits of the ASCII code. All we need to do is to isolate these four rightmost bits. We can do it by using the AND instruction.
- Specifically, we will do an AND with 0Fh.
- Example:
  - MOV AL, 35h ;AL contains the ASCII code of the digit '5'
  - AND AL, 0Fh ;AL ←AL & 0Fh = 35h & 0Fh = 05h. 05h is the value of the digit '5'
- **General Use:** "AND 1" can be used to **isolate** any bit in a register or memory.

Boolean Instruction: AND (Usage) 2/2

- Example:
  - MOV AL, 35h ;AL contains the ASCII code of the digit '5'
  - AND AL, 0Fh ;AL ←AL & 0Fh = 35h & 0Fh = 05h. 05h is the value of the digit '5'
- **General Use:** "AND 1" can be used to **isolate** any bit in a register or memory.

AL = 35h	0	0	1	1	0	1	0	1
0Fh	0	0	0	0	1	1	1	1
35h & 0Fh	0	0	0	0	0	1	0	1

"Eliminated" Copied

Boolean Instruction: OR

OR *Destination, Source*

- **Mnemonic**: OR
- **Operands**:
  - *Destination* can be a register or a memory
  - *Source* can be a register, a memory, or an immediate.
- **Function**: Destination ← Destination | Source
  - Bitwise **OR** of Destination and Source is stored in Destination
  - Source unchanged
  - OR clears the carry (**CY ← 0**)
  - OR may affect the zero (**ZR**) and sign (**PL**) flags.

- **Example:**

```
MOV AL, 41h    ; AL ← 41h
OR AL, 20h     ; AL ← AL | 20h = 41h | 20h = 61h      CY = 0, PL = 0, and ZR = 0
```

Boolean Instruction: OR (Usage)

OR can be used to set a particular bit of a variable to 1.

For example, the ASCII code of an uppercase letter differs by only one bit from the ASCII of the lowercase. Consider the letters 'E' and 'e':

ASCII of 'E'	0	1	0	0	0	1	0	1
Value of 20h	0	0	1	0	0	0	0	0
ASCII of 'e'	0	1	1	0	0	1	0	1

To transform the ASCII code of an uppercase letter into the ASCII code of the lowercase, we need to set 6^{th} bit (from the left) to 1. We can do this by using the OR operation with the number 20h. The number 20h has only one '1': it is the 6^{th} bit.

```
MOV BL, 45h   ; store in BL the ASCII code of 'E'
OR  BL, 20h   ; BL ← BL | 20h = 45h | 20h = 65h (ASCII code of 'e')
```

Boolean Instruction: XOR (1/1)

XOR *Destination, Source*

- **Mnemonic**: XOR
- **Operands**:
  - *Destination* can be a register or a memory
  - *Source* can be a register, a memory, or an immediate.
- **Function**: Destination ← Destination | Source
  - Bitwise **XOR** of Destination and Source is stored in Destination. XOR is the exclusive OR. XOR is the same as OR except that **1 XOR 1 = 0**. Bitwise XOR pinpoints bitwise differences.
  - Source unchanged
  - XOR clears the carry (**CY ← 0**)
  - XOR may affect the zero (**ZR)** and sign (**PL**) flags.

- **Example:**

```
MOV AL, 63h   ; AL ← 63h
XOR  AL, 21h  ; AL ← AL XOR 21h = 63h XOR 21h = 42h     CY = 0, PL = 0, and ZR = 0
```

Boolean Instruction: XOR (2/2)

- **Example:**

```
MOV  AL,  63h   ;  AL ← 63h
XOR   AL,  21h  ;  AL ← AL XOR 21h = 63h XOR 21h = 42h
```

CY = 0, ***PL*** = 0, and ***ZR*** = 0

AL = 63h	0	1	1	0	0	0	1	1
21h	0	0	1	0	0	0	0	1
AL XOR 21h	0	1	0	0	0	0	1	0

Different Bits

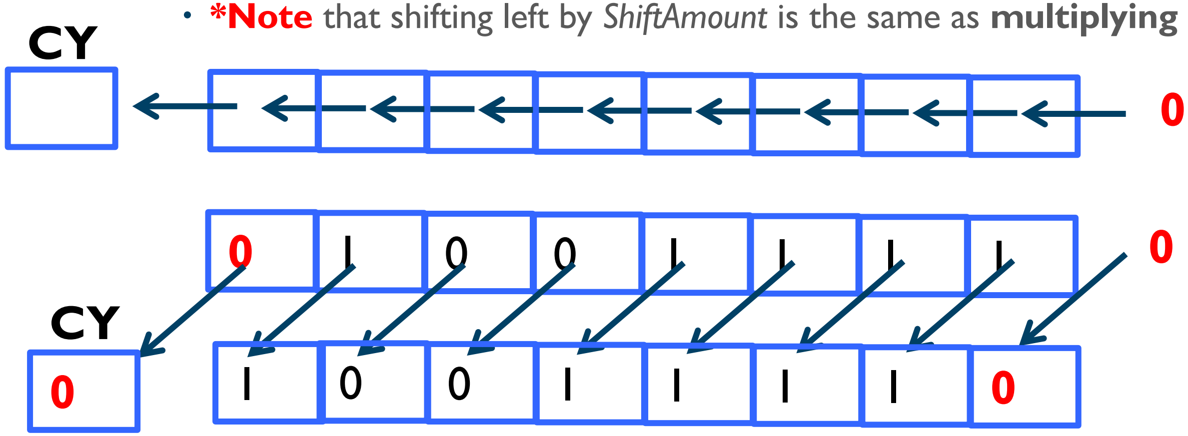
Shift/Rotation Instructions

- SHL
- SHR
- SAR
- ROL
- ROR
- RCL
- RCR

Shift Instruction: SHL (Shift Left)

SHL *Destination, ShiftAmount*

- **Mnemonic**: SHL
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Destination **<<** ShiftAmount
  - Destination is shifted left by *ShiftAmount* bits. Least significant bit is set to 0. Carry is set to the previous most significant bit.
  - ***Note** that shifting left by *ShiftAmount* is the same as **multiplying** by $2^{ShiftAmount}$.



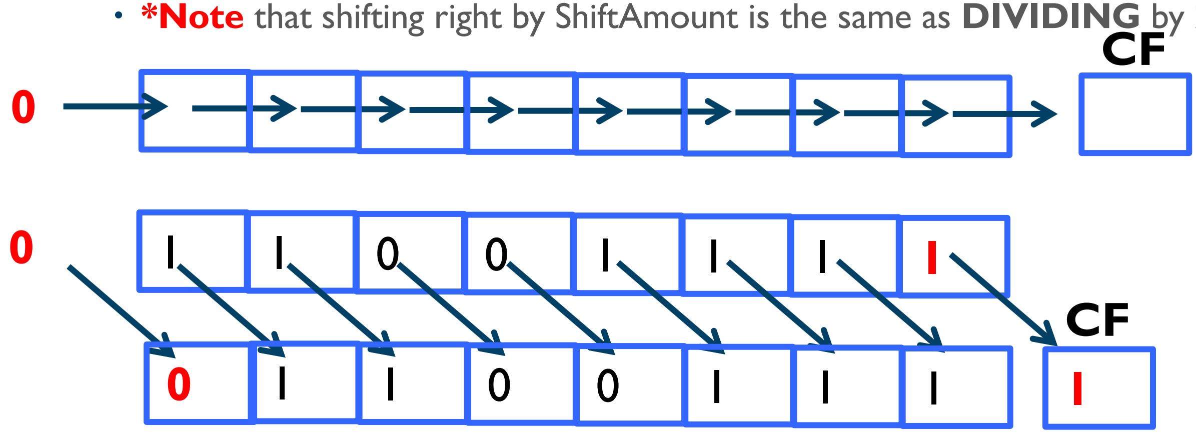
- **Example:**

```
MOV AL, 4Fh    ; AL ← 4Fh   (=79_10)
SHL AL, 1      ; AL ← AL << 1 = 4Fh << 1 = 9Eh = (158_10 = 79_10*2) and CY = 1
```

Shift Instruction: SHR (Shift Right)

SHR *Destination, ShiftAmount*

- **Mnemonic**: SHR
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Destination **>>** ShiftAmount
  - Destination is shifted right by *ShiftAmount* bits. Most significant bit is set to 0. Carry is set to the previous least significant bit.
  - ***Note** that shifting right by ShiftAmount is the same as **DIVIDING** by $2^{ShiftAmount}$.



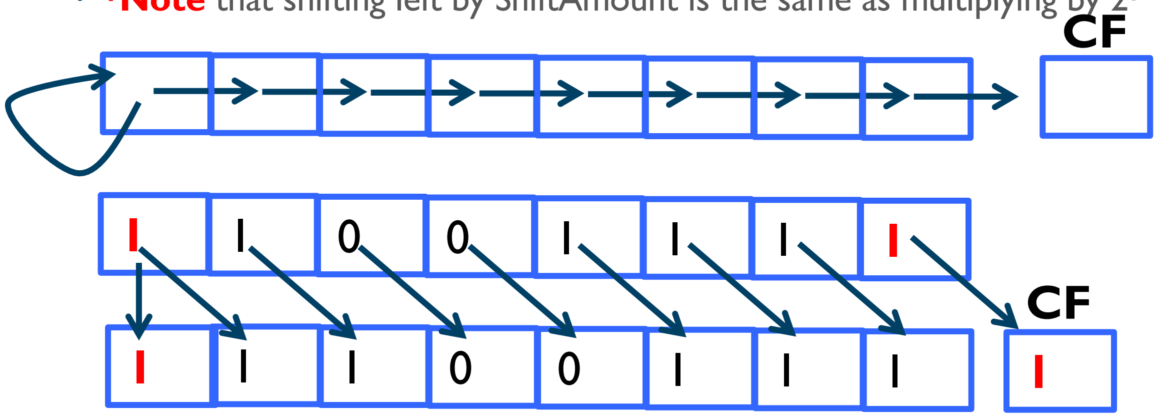
- **Example:**

```
MOV AL, 0CFh  ;  AL ← CFh (207₁₀ )
SHR AL, 1     ;  AL ← AL >> 1 = CFh >> 1 = 67h = (103₁₀ = 207₁₀/ 2)  and CY = 1
```

Shift Instruction: SAR (Shift Arithmetic Right)

SAR *Destination, ShiftAmount*

- **Mnemonic**: SAR
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Destination **>>** ShiftAmount (Safeguards Sign bit)
  - Destination is shifted right by *ShiftAmount* bits. Most significant bit (**MSB**) unchanged . Carry is set to the previous least significant bit (**LSB**).
  - ***Note** that shifting left by ShiftAmount is the same as multiplying by $2^{ShiftAmount}$.



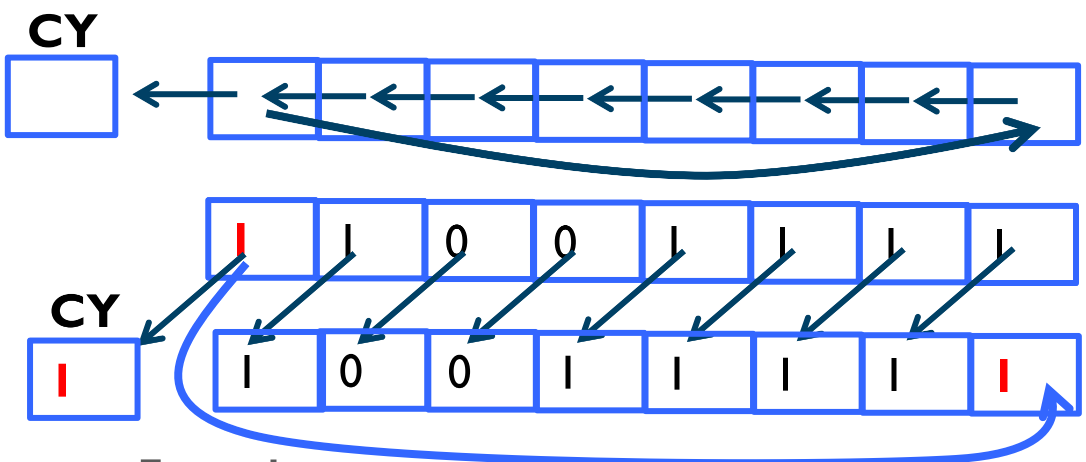
- **Example:**

```
MOV AL, 0CFh  ; AL ← CFh
SAR AL, 1     ; AL ← AL >> 1 = CFh << 1 =E7Fh and CY = 1
```

Rotate Instruction: ROL (Rotate Left)

ROL *Destination, ShiftAmount*

- **Mnemonic**: ROL
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Destination **<<** ShiftAmount (LSB copies MSB)
  - Destination is shifted left by *ShiftAmount* bits. LSB copies previous MSB . Carry is set to the previous most significant bit.



- **Example:**

```
MOV AL, 0CFh  ;
ROL AL, 1     ;
```

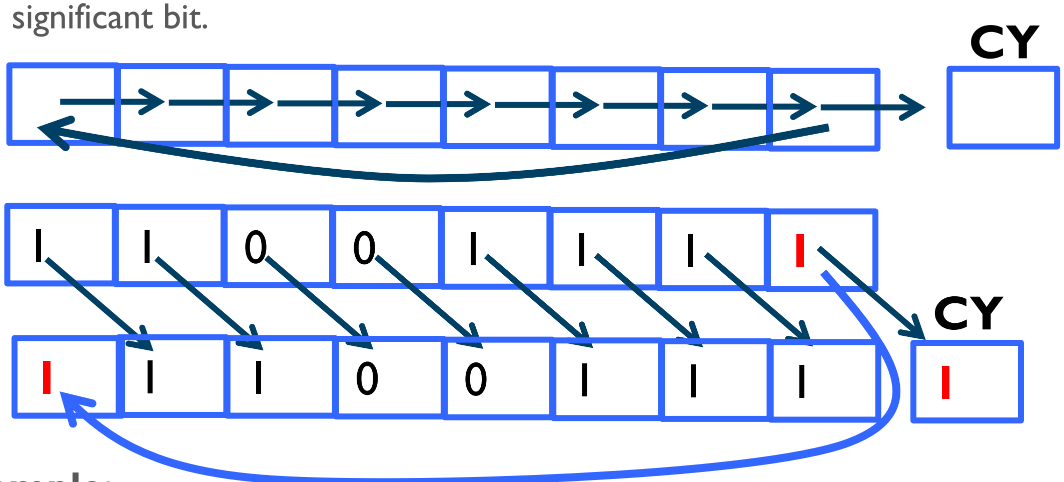
`MOV AL, 0CFh` ; AL ← CFh

`ROL AL, 1` ; AL ← AL rotated left, LSB copies MSB, and **CY** = MSB = 1

Rotate Instruction: ROR (Rotate Right)

ROR *Destination, ShiftAmount*

- **Mnemonic**: ROR
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Destination **>>** ShiftAmount (MSB copies LSB)
  - Destination is shifted right by ShiftAmount bits. MSB copies LSB . Carry is set to the previous least significant bit.



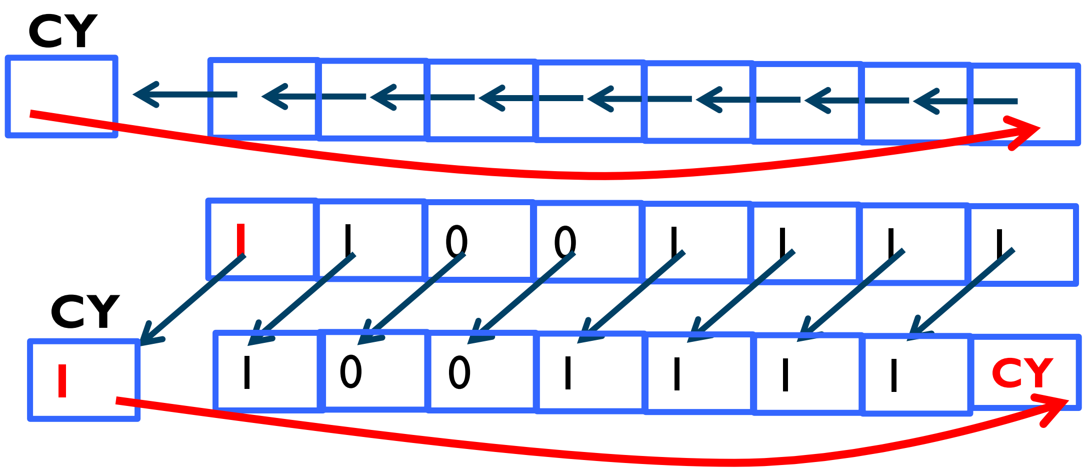
- **Example:**

```
MOV AL, 0CFh  ;  AL ← CFh
ROR AL, 1     ;  AL ← AL rotated right, MSB copies LSB, and CY = LSB = 1
```

Rotate Instruction: RCL (Rotate Left Through Carry)

RCL *Destination, ShiftAmount*

- **Mnemonic**: RCL
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Rotate left (Destination and Carry)
  - Rotate left Destination and carry.



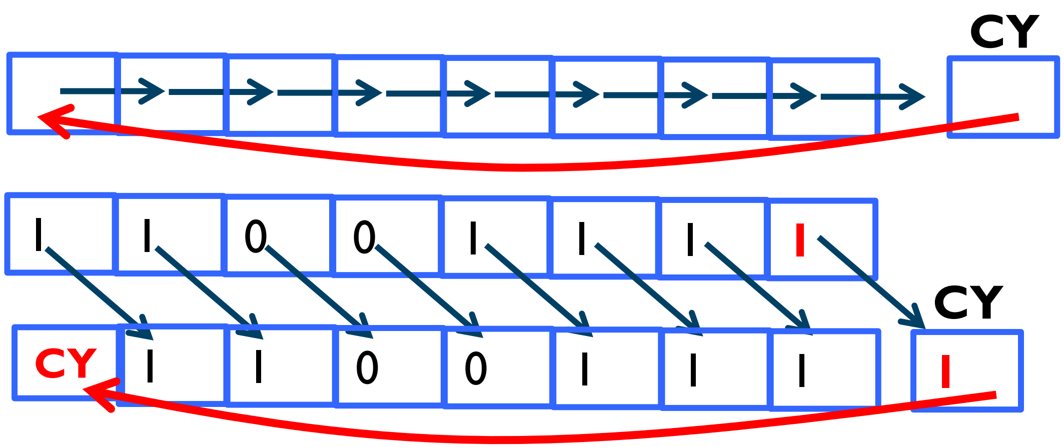
- **Example:**

```
MOV AL, 0CFh  ;  AL ← CFh (Assume CY = 0)
RCL AL, 1     ;  AL ← AL  and Carry rotated left = 9Eh and CY = 1
```

Boolean Instruction: RCR (Rotate Right Through Carry)

ROR *Destination, ShiftAmount*

- **Mnemonic**: RCR
- **Operands**:
  - *Destination* can be a register or a memory
  - *ShiftAmount* can be Register **CL** or an immediate from 0 to 255.
- **Function**: Destination ← Rotate Right (Destination and Carry)
  - Rotate right (Destination and carry).



- **Example:**

```
MOV AL, 0CFh ;
RCR AL, 1    ;
```

MOV AL, 0CFh ; AL ← CFh (suppose CY = 1)

RCR AL, 1 ; AL ← AL and Carry rotated right = E7h and ***CY*** = 1

Module Wrap Up

- **Data transfer** instructions: **MOV**
- **Arithmetic** instructions (**INC**, **DEC**, **ADD**, **ADC**, **SUB**, **MUL**)
- **Boolean** (logic) instructions (**NOT**, **AND**, **OR**, **XOR**)
- **Shift** instructions (**SHL**, **SAL**, **SHR**)
- **Rotate** instructions (**ROL**, **ROR**, **RCL**, **RCR**)