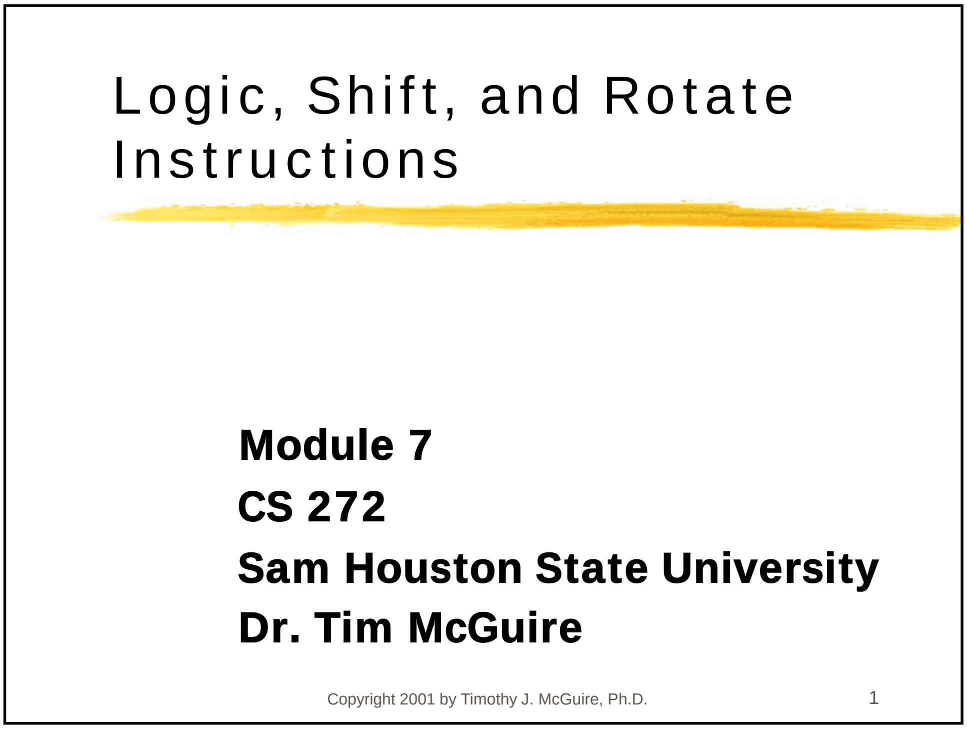


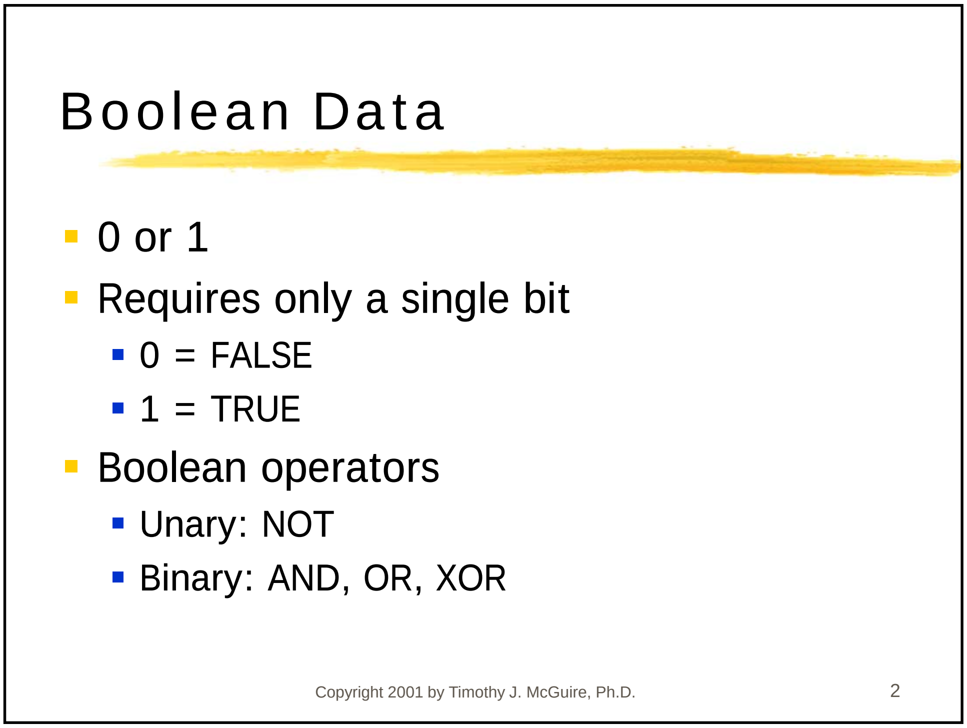
# Logic, Shift, and Rotate Instructions

**Module 7**

**CS 272**

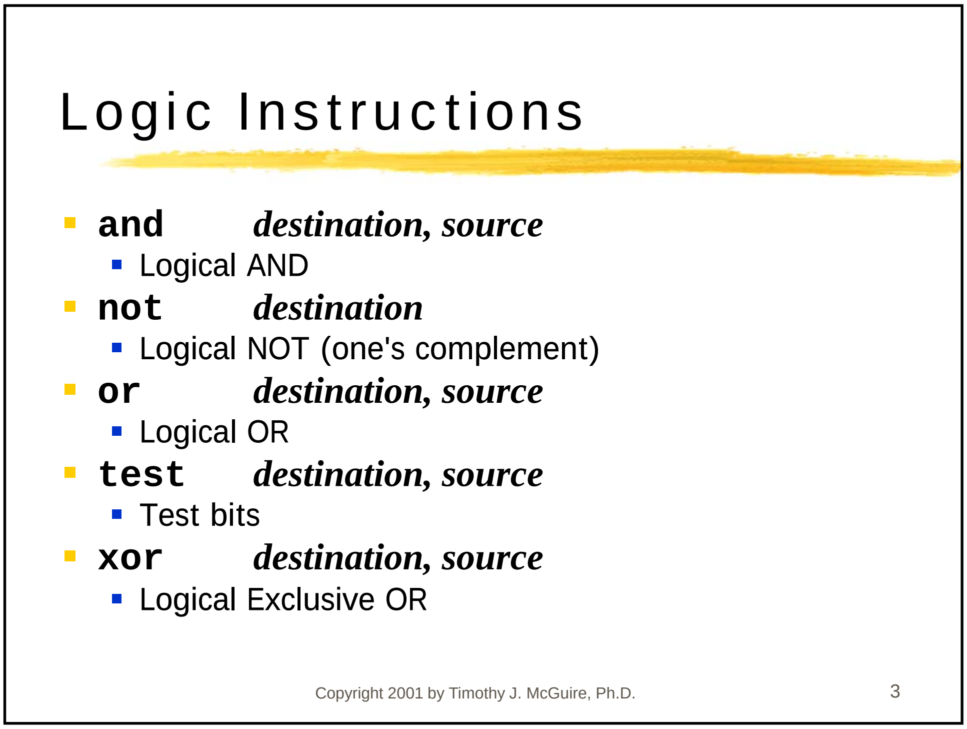
**Sam Houston State University**

**Dr. Tim McGuire**



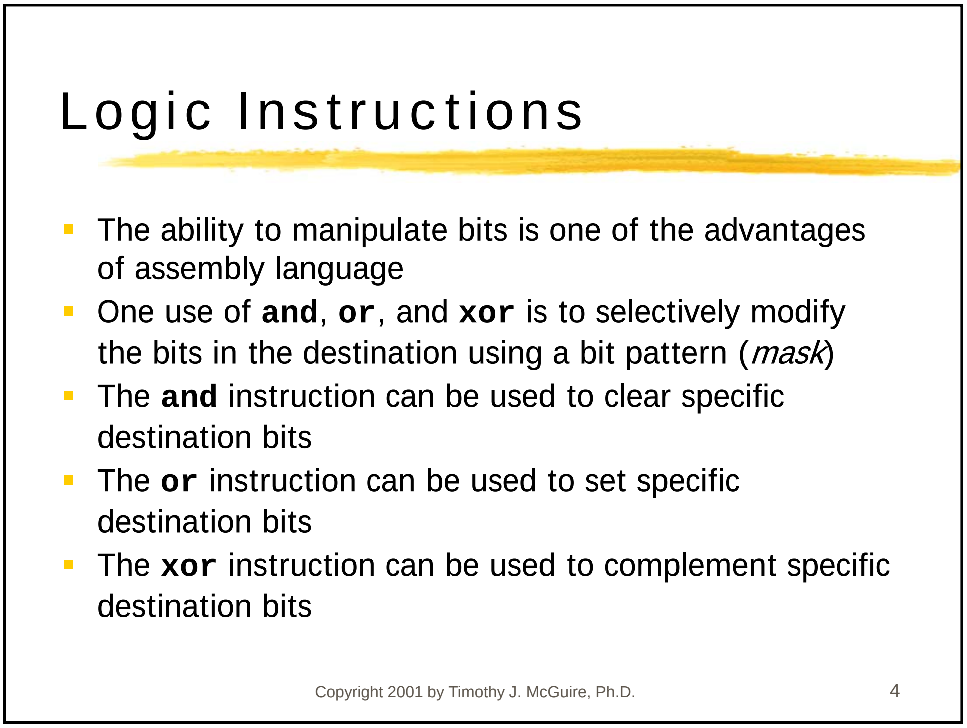
## Boolean Data

- 0 or 1
- Requires only a single bit
  - 0 = FALSE
  - 1 = TRUE
- Boolean operators
  - Unary: NOT
  - Binary: AND, OR, XOR



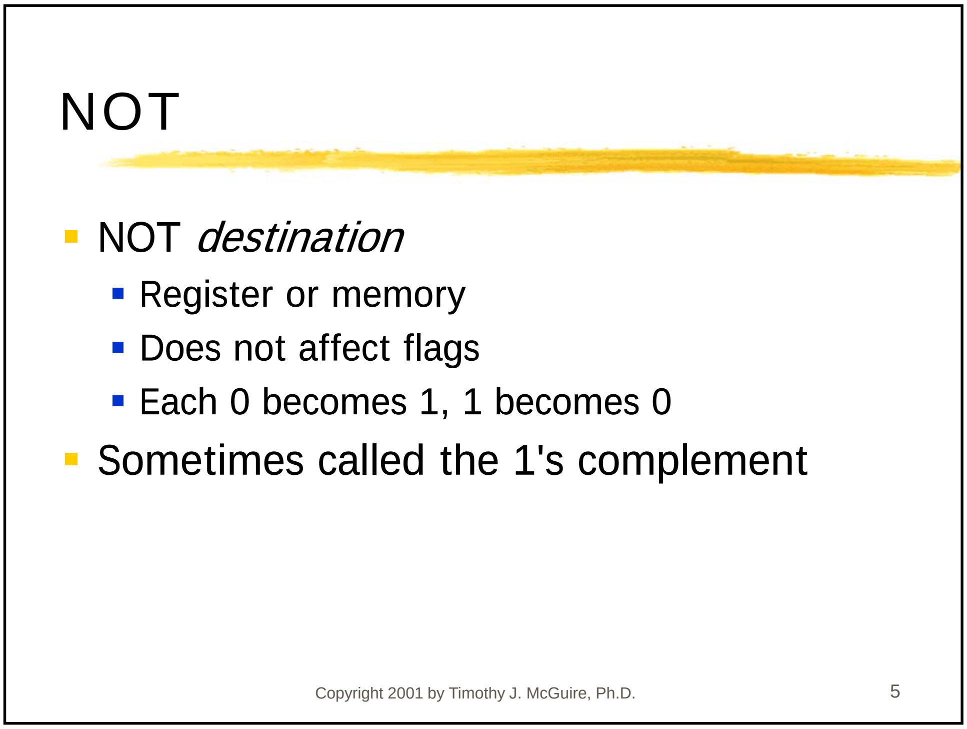
## Logic Instructions

- **and** *destination, source*
  - Logical AND
- **not** *destination*
  - Logical NOT (one's complement)
- **or** *destination, source*
  - Logical OR
- **test** *destination, source*
  - Test bits
- **xor** *destination, source*
  - Logical Exclusive OR



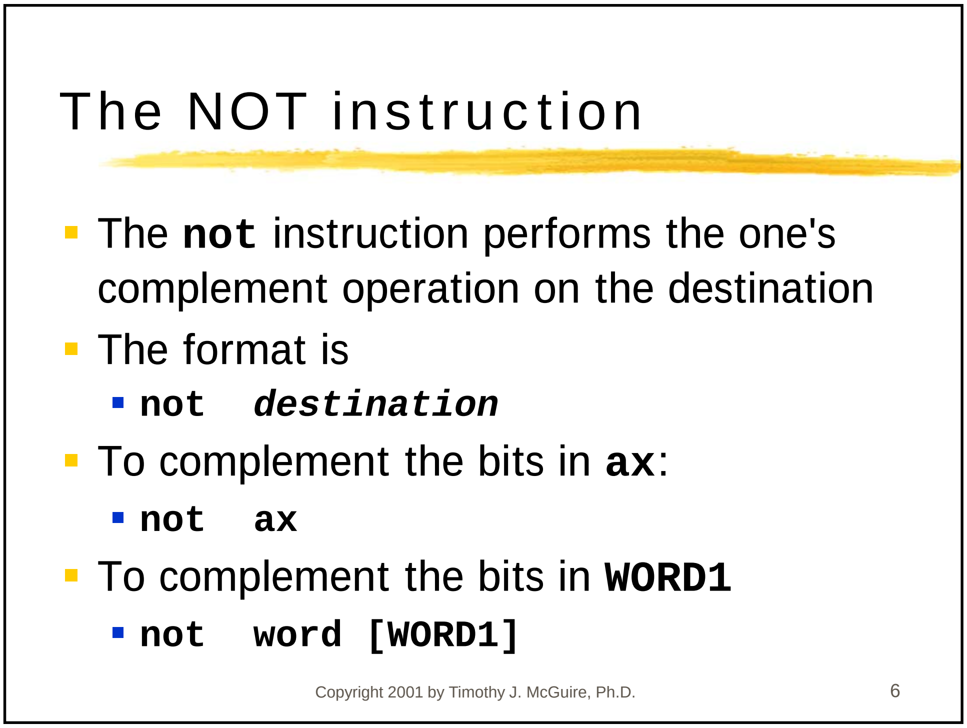
## Logic Instructions

- The ability to manipulate bits is one of the advantages of assembly language
- One use of **and**, **or**, and **xor** is to selectively modify the bits in the destination using a bit pattern (*mask*)
- The **and** instruction can be used to clear specific destination bits
- The **or** instruction can be used to set specific destination bits
- The **xor** instruction can be used to complement specific destination bits



## NOT

- NOT *destination*
  - Register or memory
  - Does not affect flags
  - Each 0 becomes 1, 1 becomes 0
- Sometimes called the 1's complement



## The NOT instruction

- The **`not`** instruction performs the one's complement operation on the destination
- The format is
  - **`not   destination`**
- To complement the bits in **`ax`**:
  - **`not   ax`**
- To complement the bits in **`WORD1`**
  - **`not   word [WORD1]`**

## AND, OR, XOR

- AND|OR|XOR *destination, source*
  - reg, reg|mem|immed
  - mem, reg|immed
- SF, ZF, PF are meaningfully set, CF=OF=0
- x AND y = 1 IFF x=y=1
- x OR y = 0 IFF x=y=0
- x XOR y = 0 IFF x=y

## Examples

- To clear the sign bit of **`al`** while leaving the other bits unchanged, use the **`and`** instruction with **`01111111b = 7Fh`** as the mask

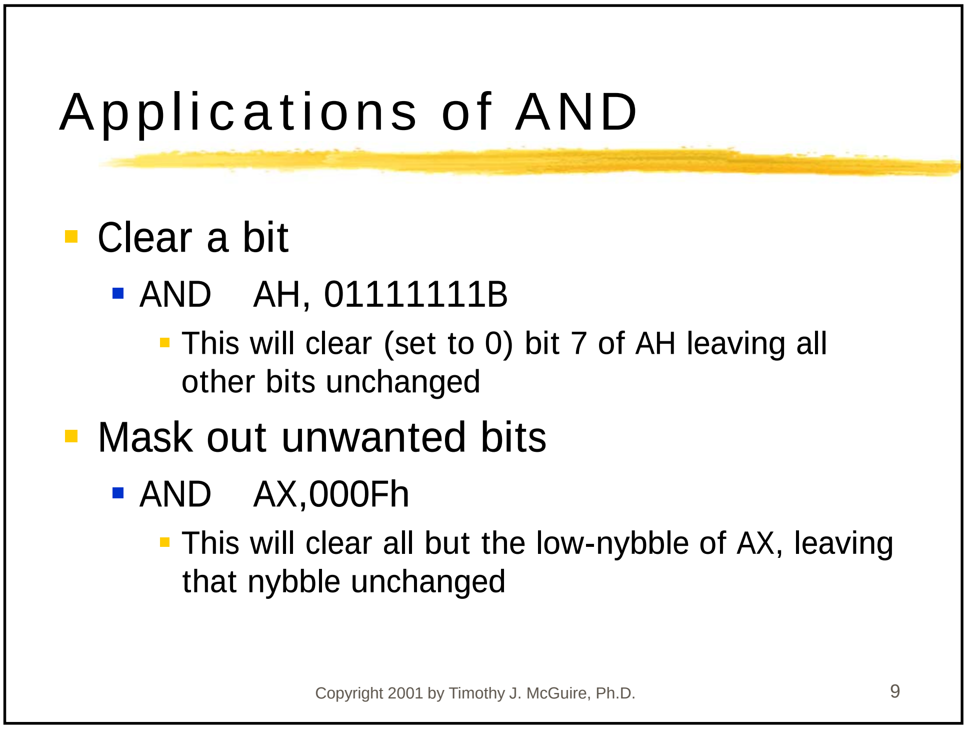
```
and     al,7Fh
```

- To set the most significant and least significant bits of **`al`** while preserving the other bits, use the **`or`** instruction with **`10000001b = 81h`** as the mask

```
or      al,81h
```

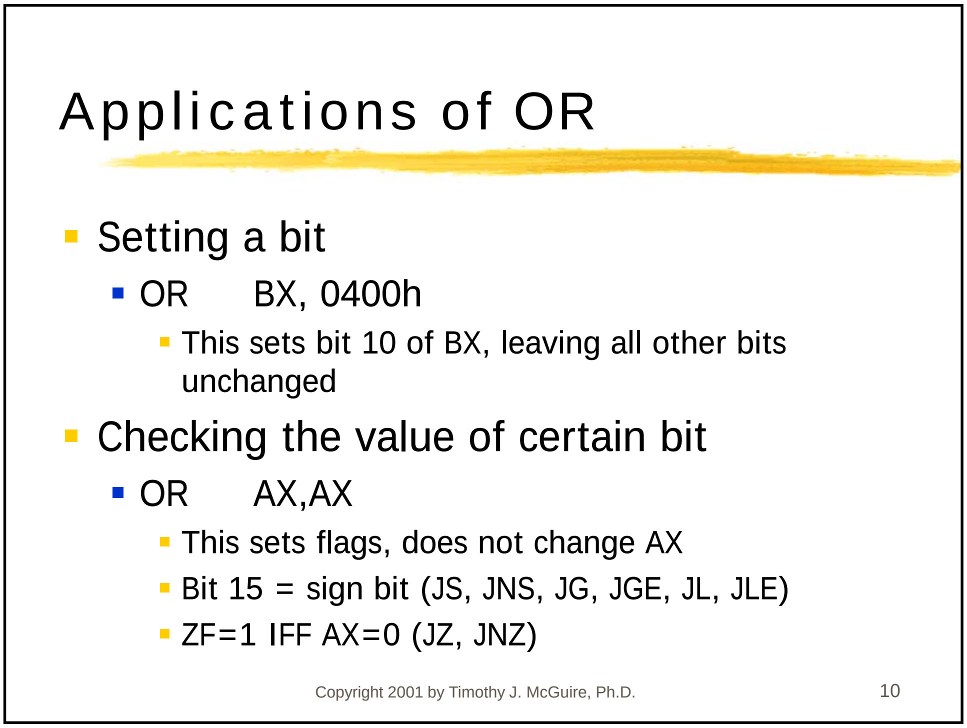
- To change the sign bit of **`dx`**, use the **`xor`** instruction with a mask of **`8000h`**

```
xor     dx,8000h
```



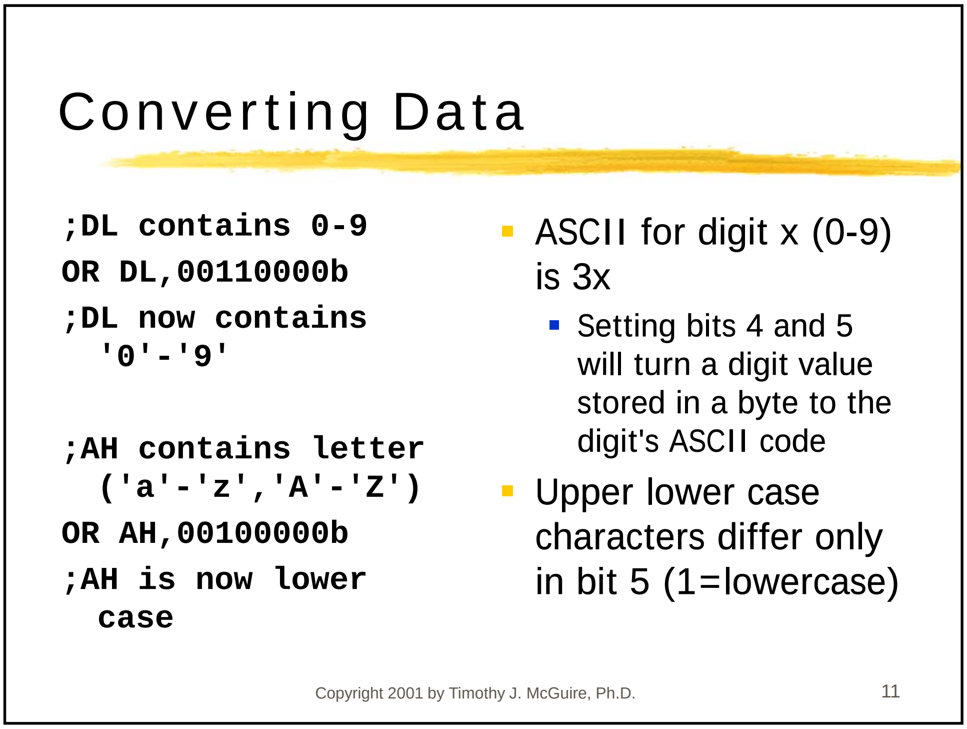
## Applications of AND

- Clear a bit
  - AND AH, 01111111B
    - This will clear (set to 0) bit 7 of AH leaving all other bits unchanged
- Mask out unwanted bits
  - AND AX,000Fh
    - This will clear all but the low-nybble of AX, leaving that nybble unchanged



## Applications of OR

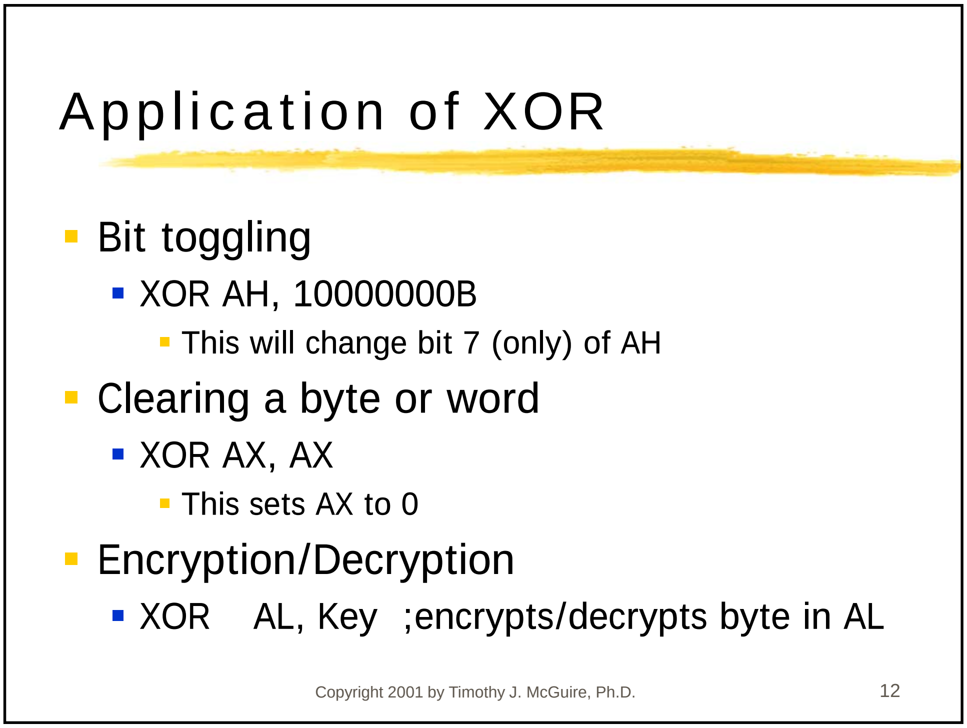
- Setting a bit
  - OR BX, 0400h
    - This sets bit 10 of BX, leaving all other bits unchanged
- Checking the value of certain bit
  - OR AX,AX
    - This sets flags, does not change AX
    - Bit 15 = sign bit (JS, JNS, JG, JGE, JL, JLE)
    - ZF=1 IFF AX=0 (JZ, JNZ)



## Converting Data

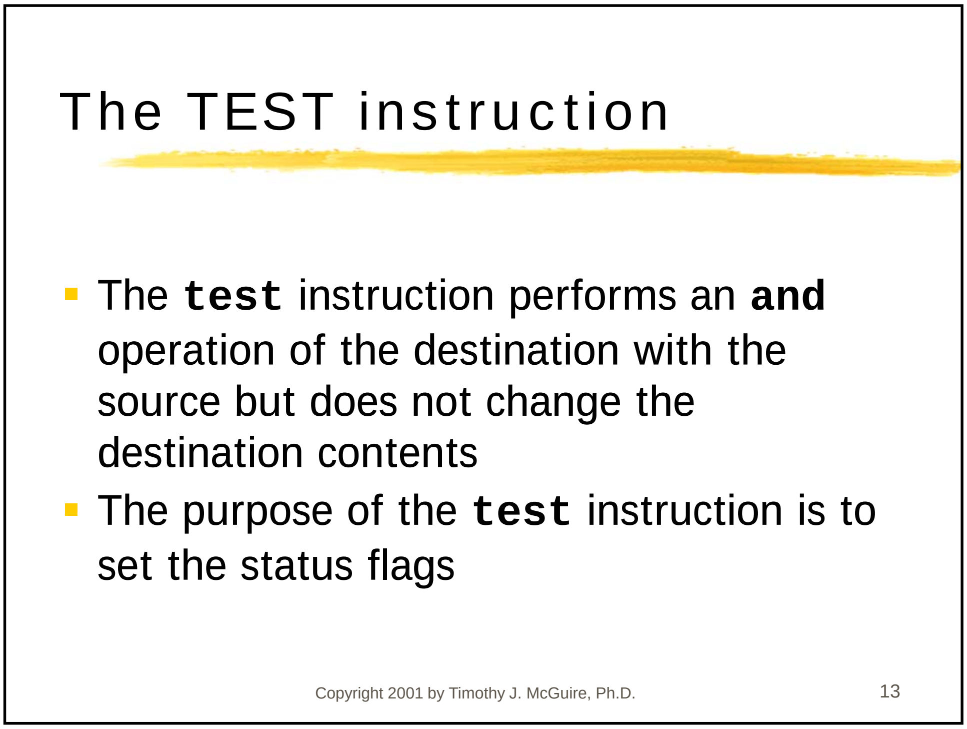
```
;DL contains 0-9
OR DL,00110000b
;DL now contains
  '0'-'9'

;AH contains letter
  ('a'-'z','A'-'Z')
OR AH,00100000b
;AH is now lower
  case
```

- ASCII for digit x (0-9) is 3x
  - Setting bits 4 and 5 will turn a digit value stored in a byte to the digit's ASCII code
- Upper lower case characters differ only in bit 5 (1=lowercase)



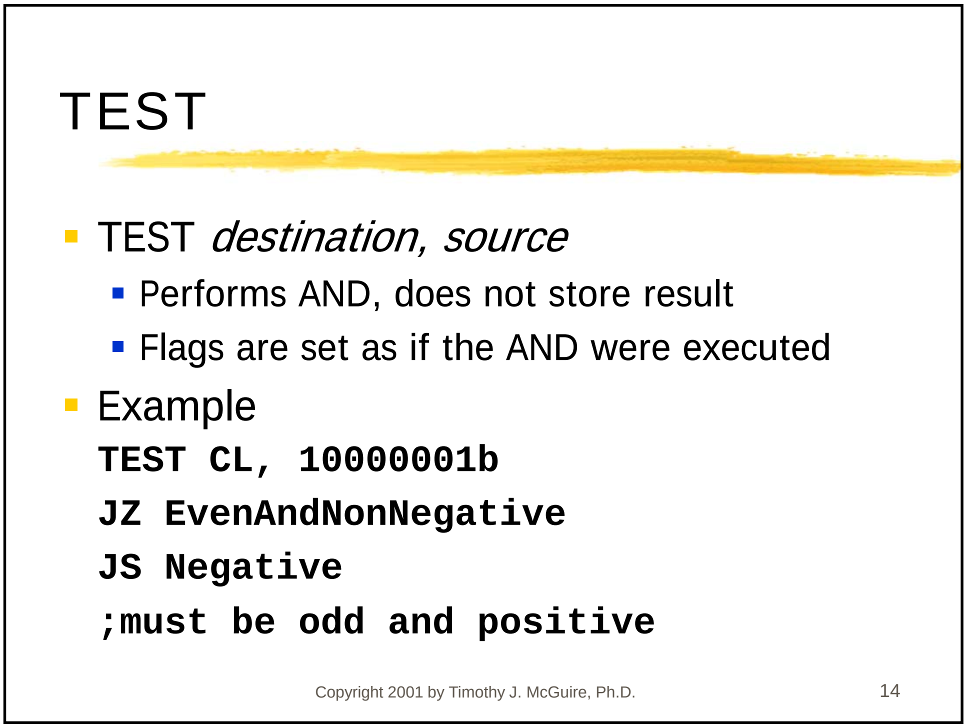
## Application of XOR

- Bit toggling
  - XOR AH, 10000000B
    - This will change bit 7 (only) of AH
- Clearing a byte or word
  - XOR AX, AX
    - This sets AX to 0
- Encryption/Decryption
  - XOR AL, Key ;encrypts/decrypts byte in AL



## The TEST instruction

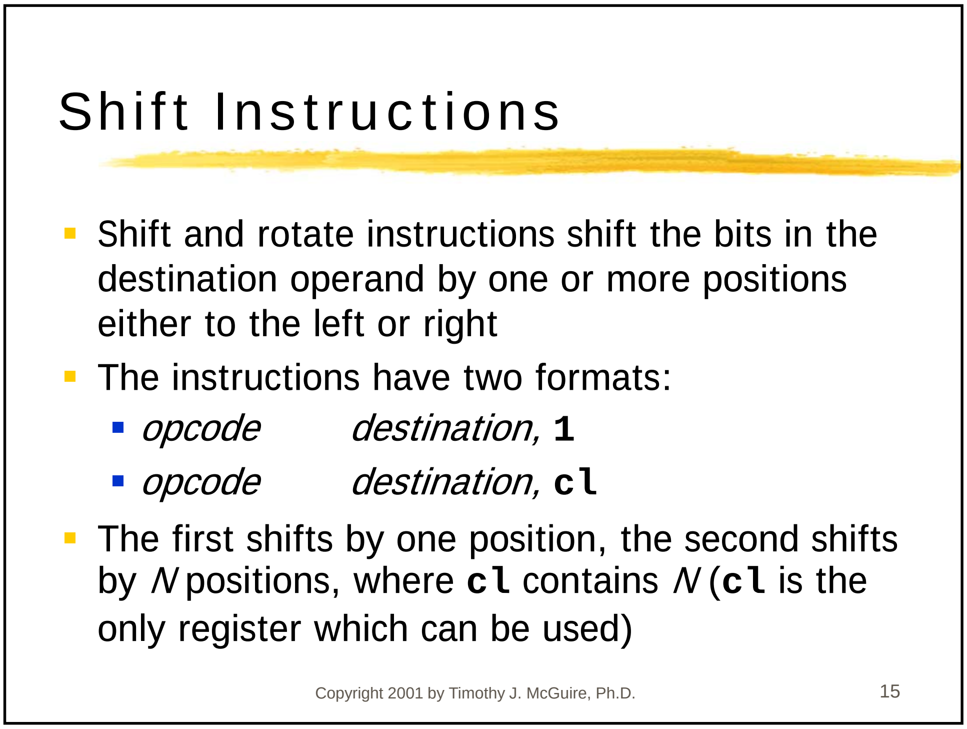
- The **`test`** instruction performs an **`and`** operation of the destination with the source but does not change the destination contents
- The purpose of the **`test`** instruction is to set the status flags



## TEST

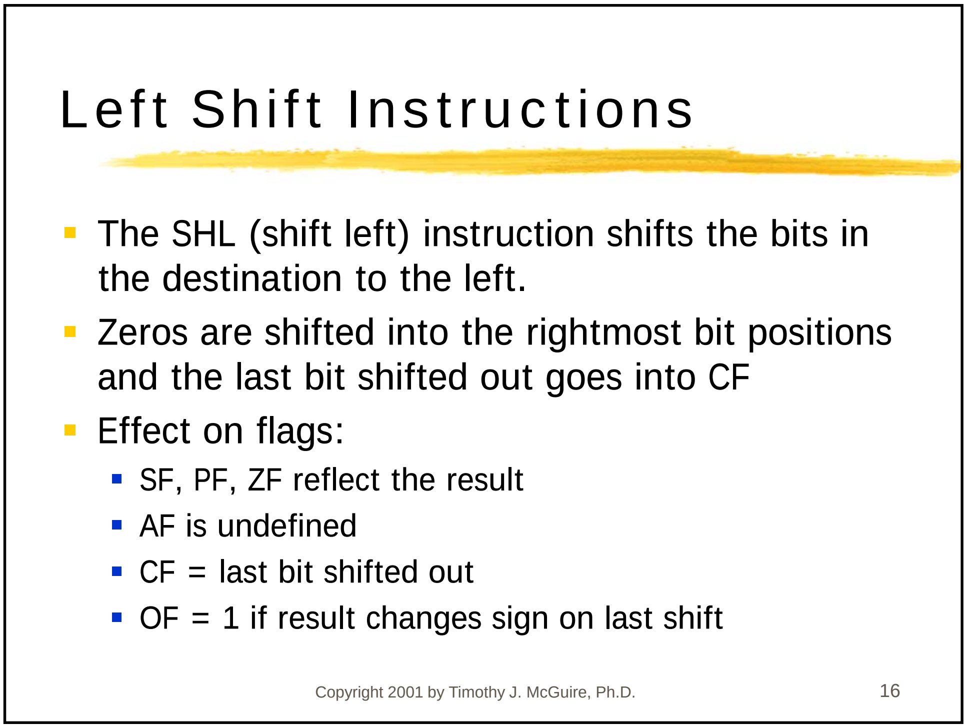
- TEST *destination, source*
  - Performs AND, does not store result
  - Flags are set as if the AND were executed
- Example

```
TEST CL, 10000001b
JZ EvenAndNonNegative
JS Negative
;must be odd and positive
```



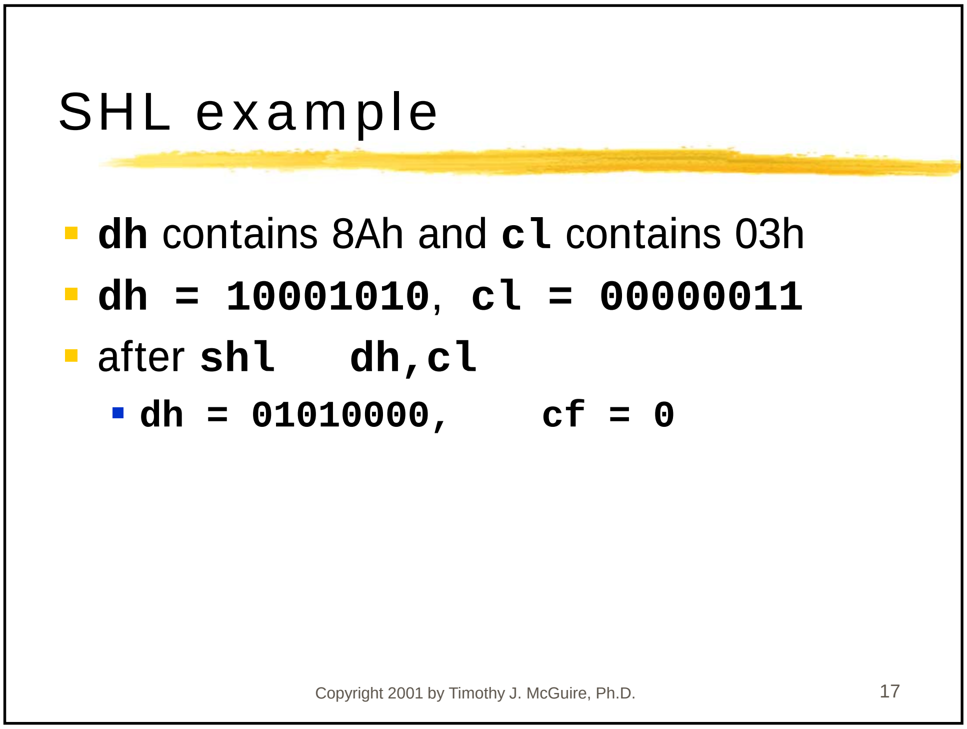
## Shift Instructions

- Shift and rotate instructions shift the bits in the destination operand by one or more positions either to the left or right
- The instructions have two formats:
  - *opcode* *destination,* **`1`**
  - *opcode* *destination,* **`cl`**
- The first shifts by one position, the second shifts by *N* positions, where **`cl`** contains *N* (**`cl`** is the only register which can be used)



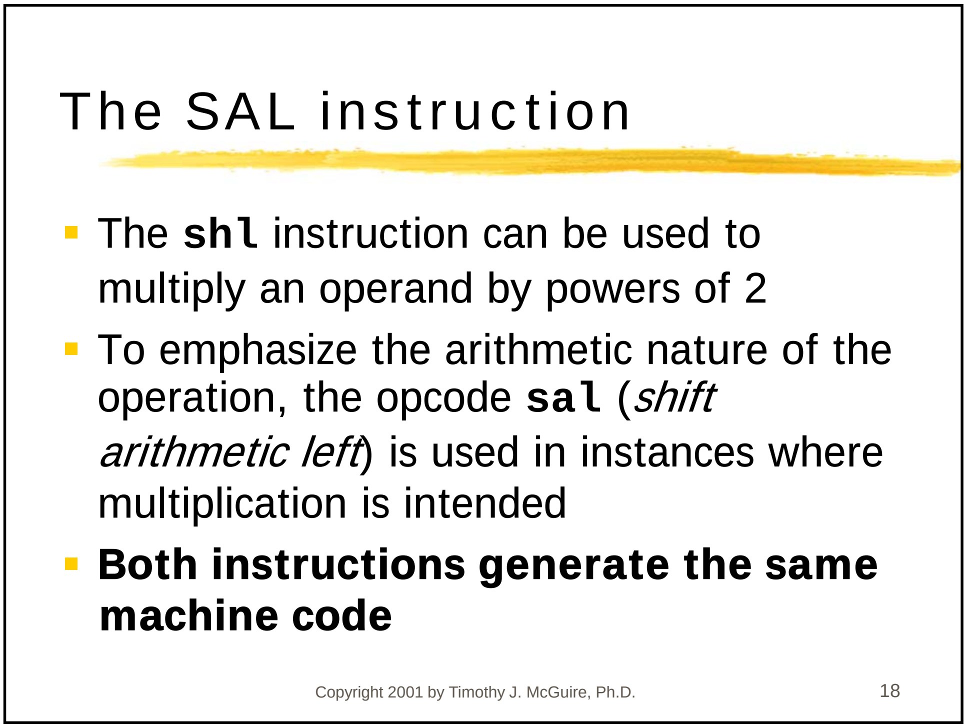
## Left Shift Instructions

- The SHL (shift left) instruction shifts the bits in the destination to the left.
- Zeros are shifted into the rightmost bit positions and the last bit shifted out goes into CF
- Effect on flags:
  - SF, PF, ZF reflect the result
  - AF is undefined
  - CF = last bit shifted out
  - OF = 1 if result changes sign on last shift



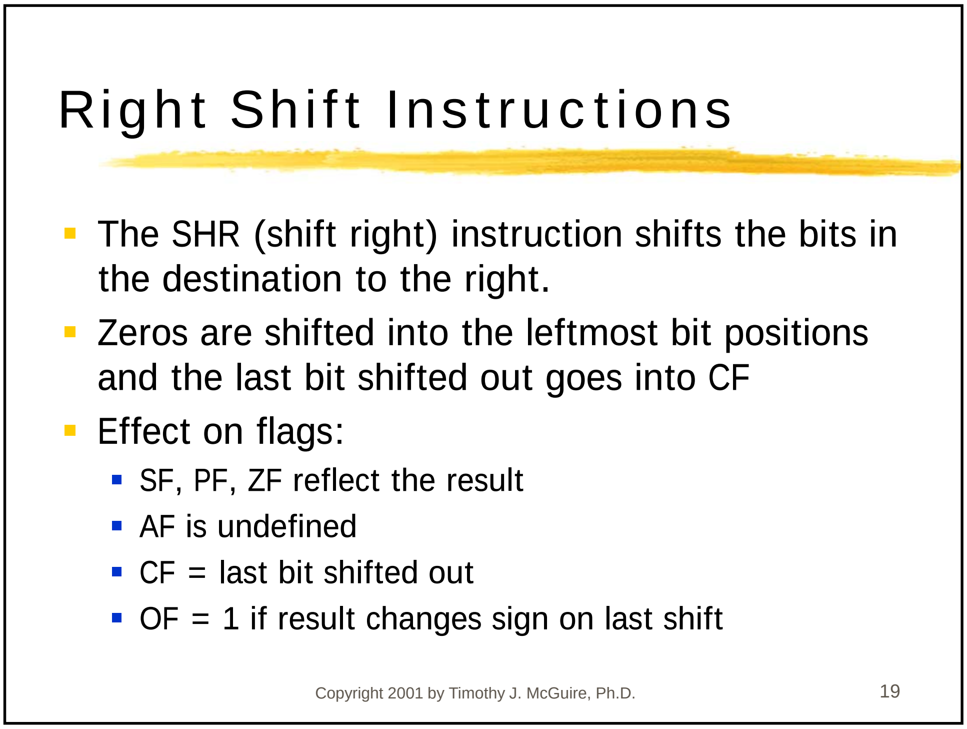
## SHL example

- **dh** contains 8Ah and **`cl`** contains 03h
- **`dh = 10001010`**, **`cl = 00000011`**
- after **`shl    dh,cl`**
  - **`dh = 01010000,    cf = 0`**



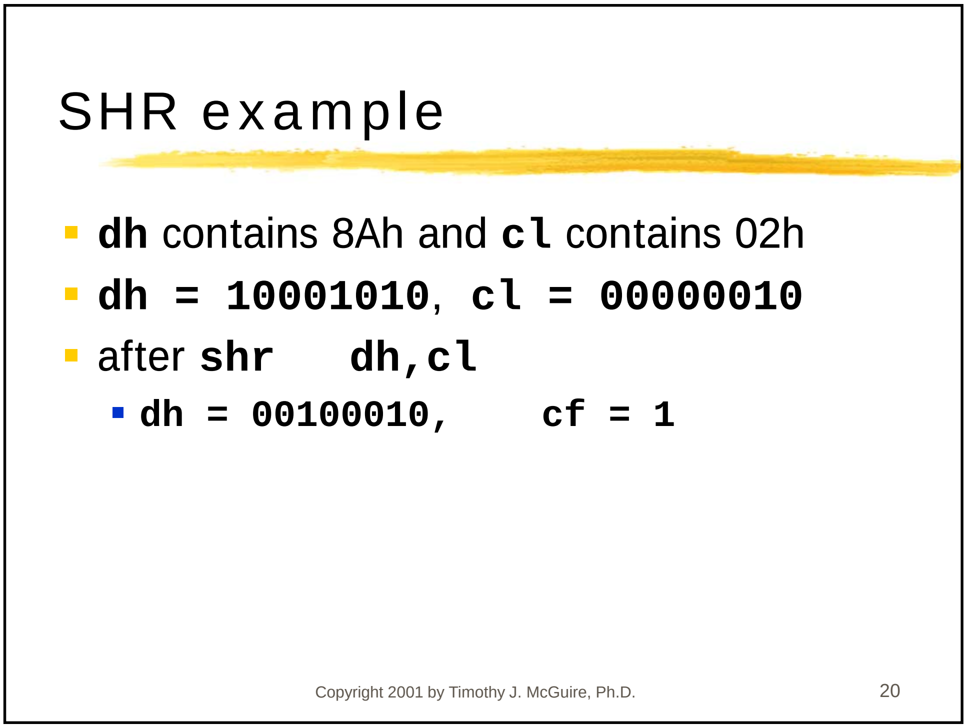
## The SAL instruction

- The **`shl`** instruction can be used to multiply an operand by powers of 2
- To emphasize the arithmetic nature of the operation, the opcode **`sal`** (*shift arithmetic left*) is used in instances where multiplication is intended
- **Both instructions generate the same machine code**



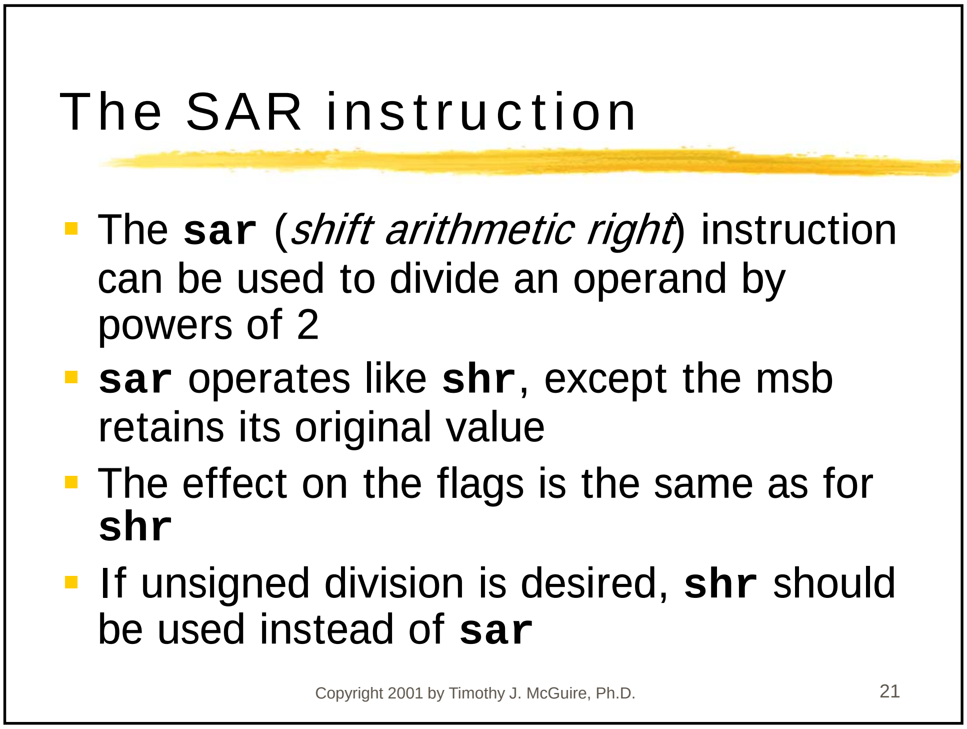
## Right Shift Instructions

- The SHR (shift right) instruction shifts the bits in the destination to the right.
- Zeros are shifted into the leftmost bit positions and the last bit shifted out goes into CF
- Effect on flags:
  - SF, PF, ZF reflect the result
  - AF is undefined
  - CF = last bit shifted out
  - OF = 1 if result changes sign on last shift



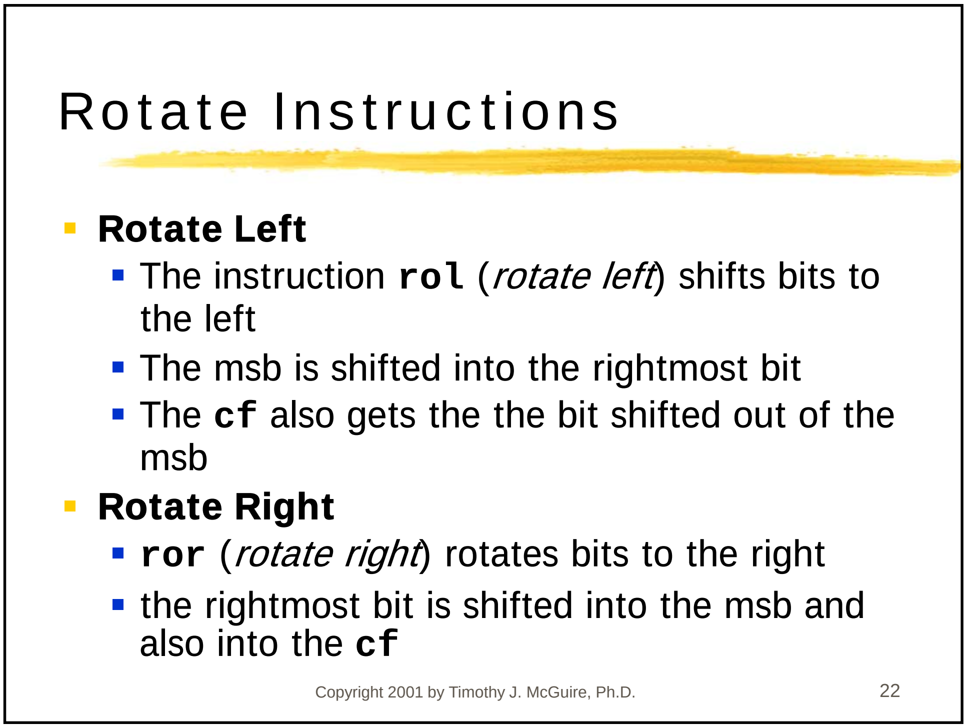
## SHR example

- **dh** contains 8Ah and **cl** contains 02h
- **dh = 10001010**, **cl = 00000010**
- after **shr dh,cl**
  - **dh = 00100010, cf = 1**



## The SAR instruction

- The **sar** (*shift arithmetic right*) instruction can be used to divide an operand by powers of 2
- **sar** operates like **shr**, except the msb retains its original value
- The effect on the flags is the same as for **shr**
- If unsigned division is desired, **shr** should be used instead of **sar**



## Rotate Instructions

- **Rotate Left**
  - The instruction **rol** (*rotate left*) shifts bits to the left
  - The msb is shifted into the rightmost bit
  - The **cf** also gets the the bit shifted out of the msb
- **Rotate Right**
  - **ror** (*rotate right*) rotates bits to the right
  - the rightmost bit is shifted into the msb and also into the **cf**

# Rotate through Carry

- **Rotate through Carry Left**
  - The instruction **`rcl`** shifts bits to the left
  - The msb is shifted into **`cf`**
  - **`cf`** is shifted into the rightmost bit
- **Rotate through Carry Right**
  - **`rcr`** rotates bits to the right
  - The rightmost bit is shifted into **`cf`**
  - **`cf`** is shifted into the msb
- See **`SHIFT.ASM`** for an example

# Multiplication by 5

- ;Assume AX contains a number N to be multiplied by 5

```
MOV     DX,AX     ;DX=N also
SHL     AX,1      ;AX=2N
SHL     AX,1      ;AX=4N
ADD     AX,DX     ;AX=4N+N=5N
```

- This is likely to be much faster than a multiply instruction

Overflow (signed or unsigned) would be checked after each operation by examining OF or CF

# Application: Binary Output

- Problem: Output AX in binary format
  - Each bit must be translated to '0' or '1' and output
  - We can build up the string in memory, or output each character as it is determined
    - Our sample solution will output the bits directly
  - Note: The ASCII codes for '0' and '1' differ only in bit position 0

# Binary Output - Details

See binbin.asm for an example