NEUB CSE 322 Lab Manual 3: Logic, Shift, Rotate, Multiplication, and Division Instructions

Boolean Data

Boolean data are variable with only 2 values either 0 or 1. This data type is important because it is very simple to implement and can be represented by TRUE or FALSE for 1 and 0 respectively.

- Boolean operators
  - Unary (Operates on 1 variable): NOT
  - Binary (Operates on two variable): AND, OR, XOR

Logic Instructions

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

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

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)

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

To summarize the **and, or, xor**

b AND 1 = b	b OR 0 = b	b XOR 0 = b
b AND 0 = 0	b OR 1 = 1	b XOR 1 = ~b (complement of b)

Converting Data

```
; ACSII to number conversion  ;DL contains 0-9
     OR DL,00110000b
;DL now contains '0'-'9'

; Number to ascii conversion ;DL contains 0-9
     AND DL, 00001111b %0fh
;DL now contains '0'-'9'
```


PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Case conversion

Character	Code	Character	Code
a	01100001	A	01000001
b	01100010	B	01000010

```
;AH contains letter ('a'-'z','A'-'Z')
     OR AH,00100000b
;AH is now lower case
;AH contains letter ('a'-'z','A'-'Z')
     OR AH,11011111b ; 0dfh
;AH is now upper 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)

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 if it gets here
```

Shift and Rotate Instructions

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

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)

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

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!***

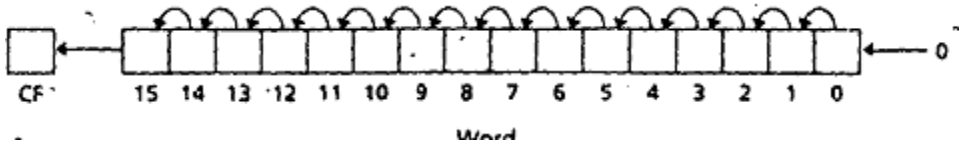


Figure 1 SHL and SAL instruction

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 = 001000010, cf = 1**

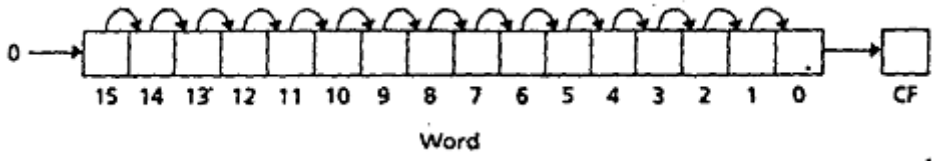


Figure 2 SHR Instruction

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

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

- The effect on the flags is the same as for **shr**
- If unsigned division is desired, **shr** should be used instead of **sar**

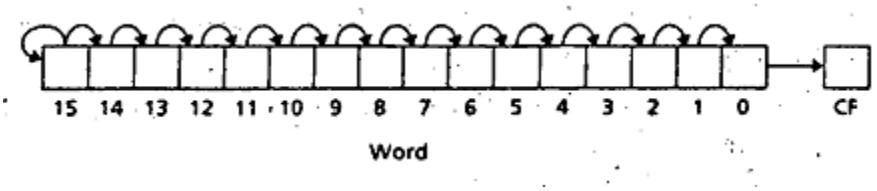
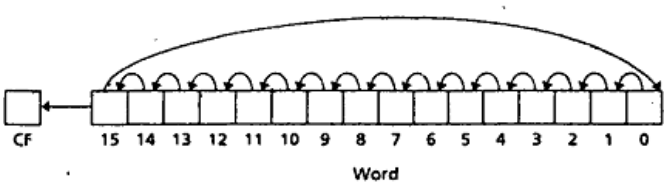


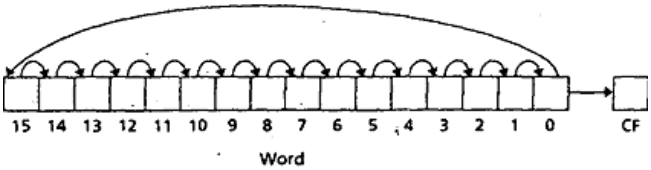
Figure 3 SAR inctruction

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

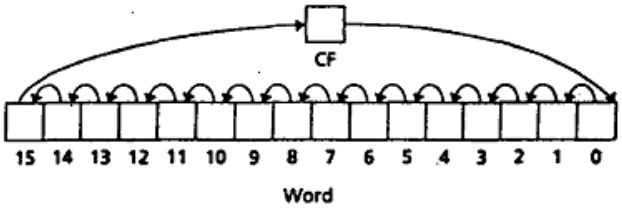


- **Rotate Right**
  - **ror** (*rotate right*) rotates bits to the right
  - the rightmost bit is shifted into the msb and also into the **cf**

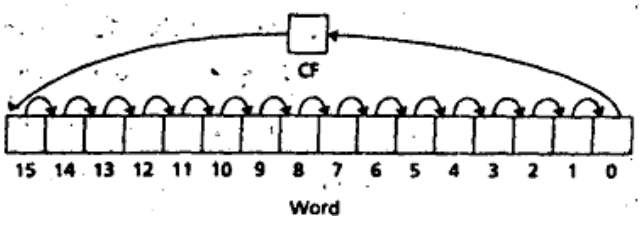


Rotate through Carry

- **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



PREPARED BY
SHAHADAT HUSSAIN PARVEZ

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

Bit Shifting (Summary)

Slide bits in byte or word to left or right

- What happens to bit that is shifted out?
  - It is copied into the **CF**
- What bit value is shifted in?
  - **SHR, SHL, SAL**: =0
  - **SAR**: =sign bit
  - **ROR, ROL**: =bit shifted out
  - **RCR, RCL**: =CF

Application: Binary and HEX I/O (Marut 7.4) [Self study]

Algorithm for Binary Input

```
Clear BX    /* BX will hold binary value */
Input a character /* '0' or '1' */
WHILE character <> CR DO
  Convert character to binary value
  Left shift BX
  Insert value into lsb of BX
  Input a character
END_WHILE
```

Algorithm for Binary Output

```
FOR 16 times DO
  Rotate left BX /* BX holds output value,
        put msb into CF */
  IF CF = 1
   THEN
    output '1'
   ELSE
    output '0'
  END_IF
END_FOR
```

Algorithm for Hex Input

```
Clear BX    /* BX will hold input value */
input hex character
WHILE character <> CR DO
  convert character to binary value
  left shift BX 4 times
  insert value into lower 4 bits of BX
  input a character
END_WHILE
```


PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Algorithm for Hex Output

```
FOR 4 times DO
   Move BH to DL    /* BX holds output value */
   shift DL 4 times to the right
   IF DL < 10
   THEN
    convert to character in '0'..'9'
    ELSE
     convert to character in 'A'..'F'
   END_IF
   output character
   Rotate BX left 4 times
END_FOR
```

Byte and Word Multiplication

- If two bytes are multiplied, the result is a 16-bit word
- If two words are multiplied, the result is a 32-bit ***doubleword***
- For the byte form, one number is contained in the source and the other is assumed to be in **al** -- the product will be in **ax**
- For the word form, one number is contained in the source and the other is assumed to be in **ax** -- the most significant 16 bits of the product will be in **dx** and the least significant 16 bits will be in **ax**

Multiply Instruction

Signed Multiply

IMUL *source*

Unsigned Multiply

MUL *source*

- ***source*** can be a register or memory location (not a constant)
- Byte form
  - AX=AL**source*
- Word form
  - DX:AX=AX**source*
- CF=OF
  - 1 if answer size > op size
  - 0 otherwise
- SF, ZF, AF, and PF
  - Undefined

Examples

- If **ax** contains **0002h** and **bx** contains **01FFh**

```
mul bx
dx = 0000h      ax = 03FEh
```

- If **ax** contains **0001h** and **bx** contains **FFFFh**

```
mul bx
dx = 0000h      ax = FFFFh
imul bx
dx = FFFFh      ax = FFFFh
```

Another IMUL Example

```
mov  AWORD,-136
mov  AX,6784
imul AWORD
```

- AWORD contains 0FF78h, AX is 1A80h
- After IMUL
  - DX:AX contains **FFF1EC00h** (-922624d)
  - CF=OF=1 (result requires a doubleword)

Another MUL Example

```
mov  AWORD,-136
mov  AX,6784
mul  AWORD
```

- AWORD is 0FF78h (65400d), AX is 1A80h
- After MUL
  - DX:AX contains 1A71EC00h (443673600d)
  - CF=OF=1 (result requires a doubleword)

One More IMUL Example

```
mov  AX,-1
mov  DX,2
imul DX
```

- AX is 0FFFFh, BX is 0002h
- After IMUL
  - DX:AX contains 0FFFFFFFEh (-2d)
  - CF=OF=0 (result still fits in a word)

Example 9.6 Translate the high-level language assignment statement A = 5 × A – 12 × B into assembly code. Let A and B be word variables, and suppose there is no overflow. Use IMUL for multiplication.

Solution:

```
MOV   AX,5          ;AX = 5
IMUL  A             ;AX = 5 x A
MOV   A,AX          ;A = 5 x A
MOV   AX,12         ;AX = 12
IMUL  B             ;AX = 12 x B
SUB   A,AX          ;A = 5 x A - 12 x B
```

Division instructions

- **cbw**
  - convert byte to word
- **cwd**
  - convert word to doubleword
- **div *source***
  - unsigned divide
- **idiv *source***
  - integer (signed) divide

Byte and Word Division

- When division is performed, there are two results, the quotient and the remainder
- These instructions divide 8 (or 16) bits into 16 (or 32) bits
- Quotient and remainder are same size as the divisor
- For the byte form, the 8 bit divisor is contained in the source and the dividend is assumed to be in **ax** -- the quotient will be in **al** and the remainder in **ah**
- For the word form, the 16 bit divisor is contained in the source and the dividend is assumed to be in **dx:ax** -- the quotient will be in **ax** and the remainder in **dx**

Examples

- If **dx** = **0000h**, **ax** = **00005h**, and **bx** = **0002h**

```
div bx
ax = 0002h      dx = 0001h
```

- If **dx** = **0000h**, **ax** = **0005h**, and **bx** = **FFFEh**

```
div bx
ax = 0000h      dx = 0005h
idiv bx
ax = FFFEh      dx = 0001h
```

Divide Overflow

- It is possible that the quotient will be too big to fit in the specified destination (**al** or **ax**)
- This can happen if the divisor is much smaller than the dividend
- When this happens, the program terminates and the system displays the message "**Divide Overflow**"

Sign Extension of the Dividend

- Word division
  - The dividend is in **dx:ax** even if the actual dividend will fit in **ax**
  - For **div**, **dx** should be cleared
  - For **idiv**, **dx** should be made the sign extension of **ax** using **cwd**
- Byte division
  - The dividend is in **ax** even if the actual dividend will fit in **al**
  - For **div**, **ah** should be cleared
  - For **idiv**, **ah** should be made the sign extension of **al** using **cbw**

Divide Instruction

Signed Divide

IDIV *divisor*

Unsigned Divide

DIV *divisor*

- **divisor can be a register or memory location (not a constant)**
- **Byte form**
  - `AL=AX / divisor [The Quotient]`
  - `AH=AX % divisor [The remainder]`
- **Word form**
  - `AX=DX:AX / divisor [The Quotient]`
  - `DX=DX:AX % divisor [The remainder]`
- All Flags undefined
- For IDIV
  - sign of dividend = sign of remainder

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Divide Overflow

It is possible that the quotient will be too big to fit in the specified destination (AL or AX). This can happen if the divisor is much smaller than the dividend. When this happens, the program terminates (as shown later) and the system displays the message "*Divide Overflow*".

Example 9.8 Suppose DX contains 0000h, AX contains 0005h, and BX contains 0002h.

Instruction	*Decimal quotient*	*Decimal remainder*	*AX*	*DX*
DIV BX	2	1	0002	0001
IDIV BX	2	1	0002	0001

Dividing 5 by 2 yields a quotient of 2 and a remainder of 1. Because both dividend and divisor are positive, DIV and IDIV give the same results.

Example 9.10 Suppose DX contains FFFFh, AX contains FFFBh, and BX contains 0002.

Instruction	*Decimal quotient*	*Decimal remainder*	*AX*	*DX*
IDIV BX	–2	–1	FFFE	FFFF
DIV BX	DIVIDE OVERFLOW			

For IDIV, DX:AX = FFFFFFFBh = –5, BX = 2. –5 divided by 2 gives a quotient of –2 = FFFEh and a remainder of –1 = FFFFh.

For DIV, the dividend DX:AX = FFFFFFFBh = 4294967291 and the divisor = 2. The actual quotient is 2147483646 = 7FFFFFFEh. This is too big to fit in AX, so the computer prints DIVIDE OVERFLOW and the program terminates. This shows what can happen if the divisor is a lot smaller than the dividend.

Formatted Numeric Output

- Want to produce a sequence of characters representing a number in some representation scheme
  - decimal, hex, octal, binary are common representation schemes
  - decimal format usually implies a sign might be shown if the number is negative (assuming we are displaying a signed value)

Generating the Digits

- `least_significant_digit = value % base`
- we can reuse this computation for the next digit if we modify value:

  `value = value / base`
- The process repeats until a 0 is obtained or the desired number of digits determined
- Problem: we generate the digits in the wrong order for display

PREPARED BY
SHAHADAT HUSSAIN PARVEZ

Reversing the Digits

Solution A

- Push digits on stack as they are discovered
- Pop each digit off the stack and display it

Solution B

- Store digits in an array as they are discovered
- Display digits from the array in the opposite order

1. Marut Chapter 7 example 1-13
2. Marut Chapter 7 exercise question 1-7
3. Marut Chapter 9 example 1-12
4. Marut Chapter 9 exercise question 1-5

Programming assignment for LAB

1. Binary input code [Chapter 7]
2. Binary output code [Chapter 7]
3. Hex input code [Chapter 7]
4. Hex output code [Chapter 7]
5. Marut Chapter 7 exercise question 8-14
6. Marut Chapter 9 example 7
7. Decimal Input and Output Procedures [Chapter 9.5]