$ - offset of the array.

Symbolic Constants:
→ cannot change at run-time
num = 5.
→ not stored in memory.
→ used by assembler when programe is scanned/assembled
→ used of directive to create a symbol which represents integer expression.
① Equal sign derivative
syntax: name = expression → integer expression [32 bit integer value]
symbol, equal sign derivative
eg count = 500 ∴ can be used in edata, & ; code.
→ mov eax, count
when the programme is assembled, assemble scans save file & produce the code
mov eax, 500
=> cant = 1000*2 ✓
count = 5+2+3 ✓
If count is used many time in programme we can easily redefine its value ∴ cant = 600.
esc-key = 27 ① mov al, esc-key
② Using dup operator; array dword cant dup (0).
↳ symbol reference

$ → current location pointer

EQU Directive → useful to define a value that is real number constant.
↳ associate a symbolic name with an integer expression or some text; name EQU expression.
symbol
<text>

06-march-2024

logical Instructions-

① AND Instruction:- → clear, carry, overflow flag

one operand as MASK

Perform and operation b/w each pair of matching bits in two operands

Syntax:-

and destination Source

X	Y	XnY
0	0	0
0	1	0
1	0	0
1	1	1

```
0011  1011
0000  1111
------------
0000  1011 → unchanged
cleared
```

Application:-

change a letter from a lower case to upper case

```
mov al, 'a'      , al = 97 = 01100001b
and al, 11011111 , al = 65 = 01000001
```

set 1 or move bit without effecting others → MASK. → Same operand combination as of and or mov

② OR Instruction:- → reset Carry overflow Auxiliary Flag.

Perform OR operation b/w each pair of matching bits in two operands.

Syntax:-

OR destination Source.

X	Y	X V Y
0	0	0
0	1	1
1	0	1
1	1	1

e.g

```
mov al, 75h   0111 0101
OR  al, 22h   0010 0010
              ---------
              0111 0111
```

Application:-

Used in micro controllers

eg: 1 2 3 4 5 6 7 8
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |

⇒ To on any bit we use OR to on bit sum → OR it with 0000001

⇒ To off any bit we use and e.g to off bit 6 → and with 1111101

⇒ decimal digit to its equivalent Ascii digit
mov al, 6 , al = 6 = 0000110
OR al, 00110000 al = 54 = 00110110

→ carry : clear

XOR instruction:-

If we want to on bit 2, we OR it with 00000100
OR al, 0000010 ; 11100111
mov al, 11100011

⇒ Perform XOR operation b/w each pair of matching bits in two operands.

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	1

syntax

XOR destination Source.

eg
00111011
00001111
00110100 → inverted

⇒ XOR reverse itself when applied to same

operand twice. so used for symmetric encryption.

=> use to check Parity of 16 bit Number, XOR upper & lower byte.

[:- use to check Parity of 16 bit Number,]

mov ax, 64c1h, 0110 0100 1100 0001

XOR ab, al, Parity Flagset.

NOT Instruction:-

It Toggles all the bits in operand

=> Result in 1's complement

NOT	
X	Nor
F	T
T	F

Syntax

→ Not reg

→ Not memory.

eg => mov al, 11110000

not al al = 00001111

=> NO flags effected.

Test Instruction:- zero flag effected.

Work similar to and Instruction but doesnot modify destination.

=> It can check several bits at once.

Syntax:-

Test Destination, Source.

=> Example:

check if bit 0 & 3 set in al

test al, 00000100 test 0 & 3.

=> check zero flag

If Zero flag set Both Bit 0 & 3 clear

zero flag not set bit 0 & 3 set

0101
1010 → 1's
+ 1
1011 → 11 + 3 = 14.

CMP Instruction

→ work same as sub instruction but not update the ~~destination~~ destination.

→ We use compare destination & Source.

Non destructive subtraction of source from destination.

Syntax:

cmp destination, Source.

① destination == Source. Zero flag set

mov al, 5
cmp al, 5

② destination < Source. carry flag set

mov al, 4
cmp al, 5

③ destination > Source

mov al, 6 ZF = 0, CF = 0
cmp al, 5

destination is always compared to the same.

13-03-23

Jump statement
↳ It is basically for block code
→ Jz ⟶ for setting up zero flag.
↳ jump if zero
Jnz ⟶ jump if not zero

1.1 Je jump if equal [same as Jz]
Jne jump if not equal [same as Jnz].

1.2 Jc → jump if carry [CF=1]
Jnc → jump if not carry [CF=0].
↳ if destination is above source jump is taken.

1.3 Ja → jump if above [ZF=0 & CF=0]
Jb → jump if below [CF=1]
↳ unsigned integer.

1.4 (for magnitude & for sign of number) JL ⟶ jump if less.
Jg ⟶ jump if greater

1.5 jae → jump if above or equal.
jbe → jump if below (unsigned) or equal
jge → jump if greater or equal.
jle → jump if lower or equal
signed

signed unsigned number.

2 0x0002
-2 FFFE, 2's complement
decimal 03534

dest -2 same 2 (65533, 2)

Unsigned numbers

```
mov ax, -2      ; ax = 65535.
mov bx, 2       ; bx = 2
cmp ax bx       ; ?
ja label 1      ; true 65534 > 2
```

Signed numbers

```
mov ax, -2
mov bx, 2
cmp ax bx
jg label 1      ; False -2 < 2
                ; signed numbers cmp.
```

```
mov ax, 5
start: inc ax
cmp ax, 10      ; keep unity whole.
jb start        ; (ax < 10)
mov val, ax.
```

```
start: mov var1, 05
① add var1, 10h
② mov ax, var1
③ inc var1
④ jump start
⑤ mov ax, var1.
```

Procedures:-

(Argument + Return type) Function ~ Procedure (None)
High level ~ low level.

=> An assembly no arguments in implicit procedural call code.

example:-

myproc . proc → start of procedure.

Procedure name

ret → (return statement is used before we end procedure)

myProc . endp → end of Procedure.

upto now

main Proc — main procedure
main endp — run first.

How to call procedure

Call Instruction

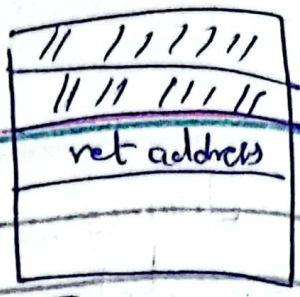
→ In main procedure.
→ use to call procedure.

e.g main
call myProc → now myProc will be executed

Return Instruction

Ret Instruction is used inside the called Procedure.

example: myProc → start Procedure
ret (It's Paiving up with call Instruction
myProc endP
→ end Procedure.



① Call (Push)

Ex

```
main Proc
    ...
0014  cal myProc
0016  mov eax, 12345678h
```

EIP 00 00 00 14 instruction address.

EIP 00 00 00 16 X

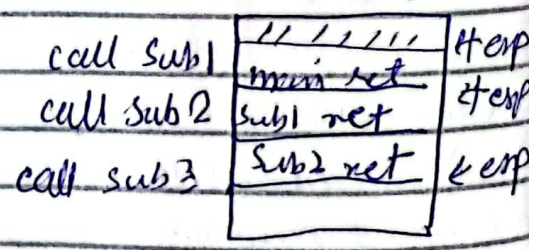
⇒ As we calling myProc so value of eip will be changed/altered. It will not be 0016 but first instruction of myProc.

```
myProc proc
00 00 10 10
    ...
    ret
myProc
```

Return: (PP)

① Restore eip
put value in eip which is of return address
⇒ EIP = ESP

② ESP move Back
ESP = ESP + 4



② Call

① make · Place on stack
- esp = esp - 4
② SSP = EIP
value which we have in EIP will be stored on stack.
③ EIP = Procedure stack addresses.

Esp → if using 32 bit then we call Esp
if 16 bit → then we call eIP.

Nested Procedure.

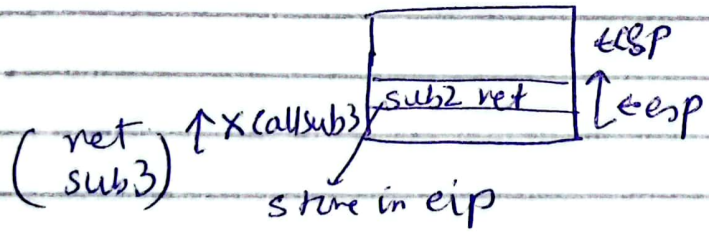
① main Proc
⋮
Cal sub 1
main endp

② Sub1 Proc
⋮
call sub2
⋮
ret
Sub1 end p

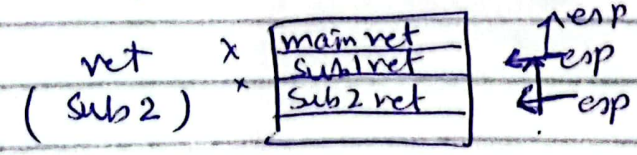
③ Sub 2 Proc
⋮
Cal sub 3
ret
sub 2 endp

④ sub 3 Proc
⋮
ret
⋮
sub 3 end p

When we call ret in sub3 then return address of sub2 will go in ~~end~~ P & eip deleted from stack.



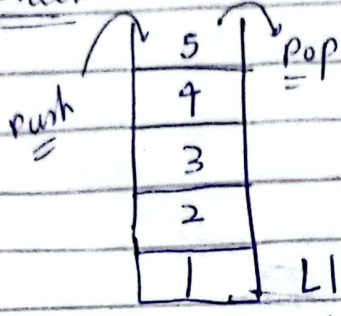
When we call ret in sub2 then return address of sub1 goes to eip & deled from stack



Run time stack:-

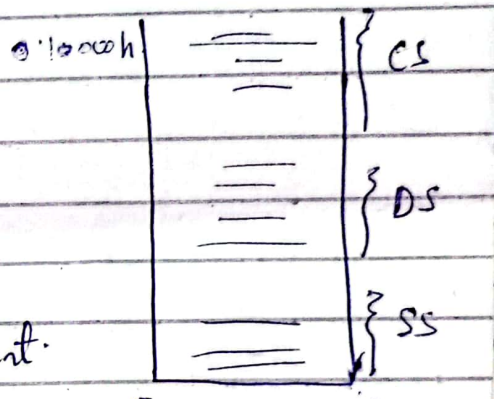
Programme → Instruction
→ Data
→ Stack

Stack



→ Programme execution:-

(1) PC assigned value
(2) DS Register initialize.
(3) CS Register initialize.
(4) SS Register initialize.

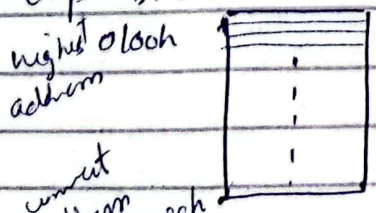


⇒ stack is not dedicated component.
⇒ Not att like RAM.
⇒ use LIFO concept

stack 0100h Reserve some space in memory.

esp ⇒ It has all the address of latest stored data item stored on stack
⇒ Program start, → IP, DS, CS, SS initialized

∴ esp start at the highest address.



es start data filling from highest address

Stack Operation

① Save on stack (Push)

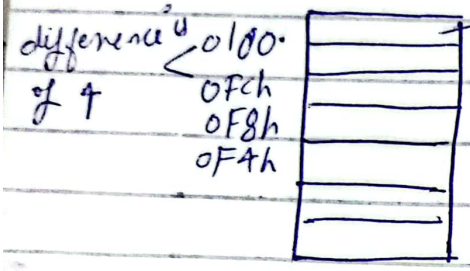
② Restore from stack (Pop)

Data element in stack.

⇒ Depending upon architecture.
(32bit, 16bit)

⇒ 32bit arch → each stack element 4bytes

⇒ 16bit arch → each stack element 2 bytes



Push

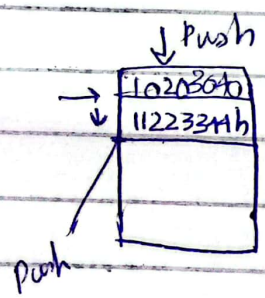
① esp = esp-4 (0100h - 4 = OFCh)

② Data stored to 0100 address in esp OFCh ← (stored data on address OFCh)

mov eax, 10203040

Push eax,

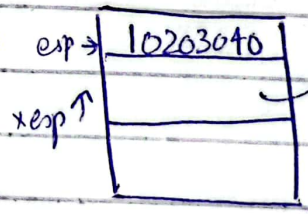
Push 11223344h



Pop edx, fetch from stack

edx = 11223344

esp = 0100h



Stack & little endian

⇒ In stack most significant bytes goes to highest address - we not explicitly Apply little endien order here.

⇒ stack goes from highest address to lowest address.

Example:

esp → extended stack pointer. Points at the highest address. (Last or current value).

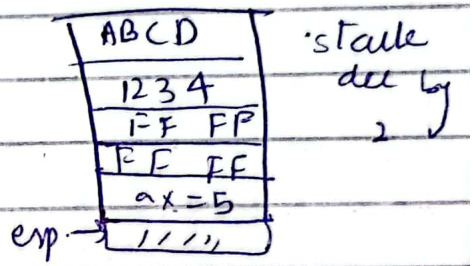
For exchanging:-

```
mov ax, 0ABCDh
mov bx, 1234h
Push ax
Push bx.
```

Pop ax → will Pop bx & store in ax

POP bx → will pop ax & store in bx

```
mov ax, 5
Push ax,
move ebx, -2
Push ebx
val dw 1234h
Push val
V2 dw ABCDH
Push V2
```



Pop val ; as val is dw so 2bytes Val = ABCD.

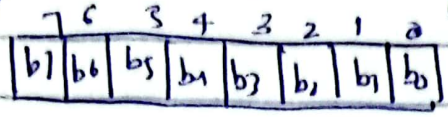
Pop v2 ; as V_2 is dw so 2bytes V2 = 1234

Pop ebx, ebx is 4 bytes so ebx = FFFFFFFF.

Computer Arithmetics

shift & rotate (Temporary Carry Flag)

shift



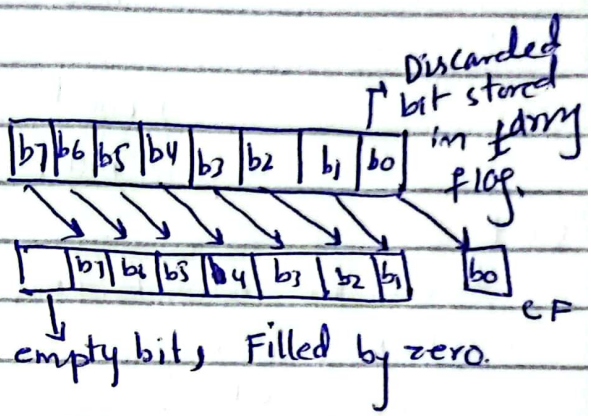
=> Reg / mem location.

with respect to direction

① left shift

② Right shift

Right shift

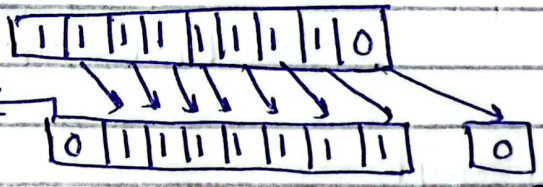


Example:

In al we have -2

-2 2's complement

shr al, 1



→ we have to retain sign so two method to fill empty bit

→ Fill it by zero => logical shift

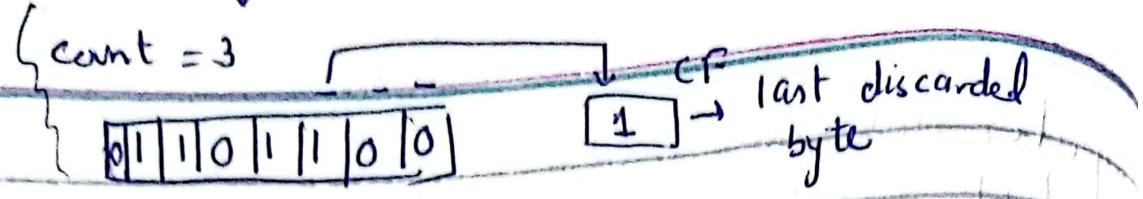
Fill it by MSB (sign bit) => Arithmetic shift

shL ~ shift left · logical

→ same syntax as shift Right only difference in direction.

→ discard bit from left shift.

Note :-



Instruction for shifts :-

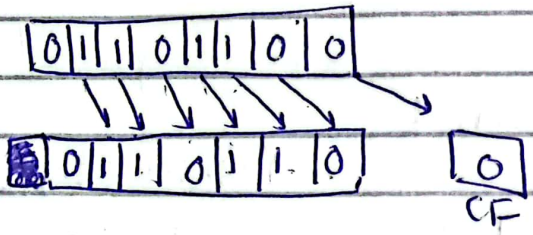
SHR ~ logical Right shift

syntax: shr |mem|reg , Count → how many position to move

; count == mm | cl
must be cl

Count
↳ can be number or we store its value in cl Register.

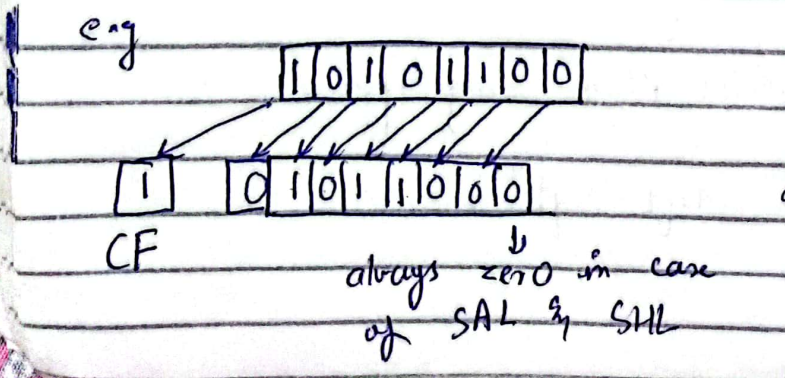
```
mov cl, 1
shr al, cl
```



Arithmetic shift

SAL ~ shift Arithmetic left.
→ Nothing to do with sign bit so always zero

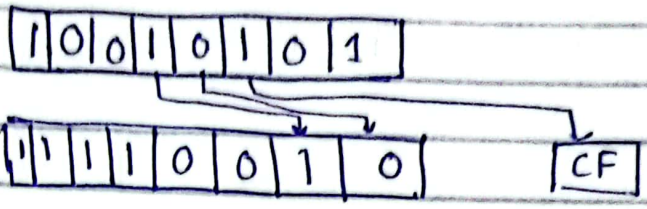
e.g



∴ discarded value will be in CF.

So, SHL = SAL

SAR ~ Shift Arithmetic Right Perform right shift



Application:

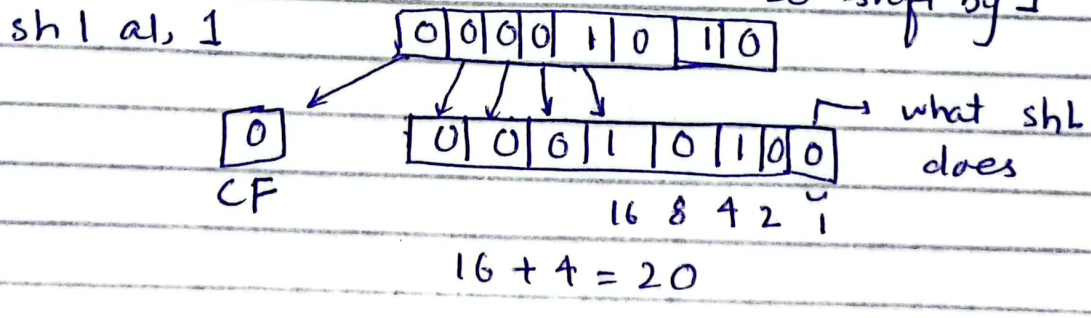
① FAst Multiplication.

→ mul a number by Power of 2

Use SHL (shift left by Power of 2)

e.g $10 \times 2^1 = 20$ ∴ 2^1 = Power = 1

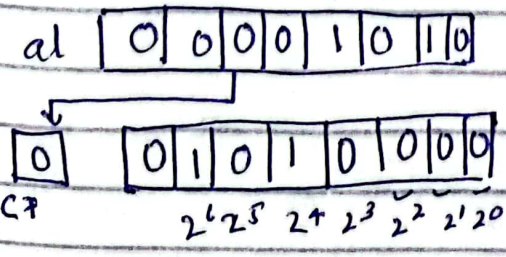
so shift by 1



16 + 4 = 20

```
mov al, 10
mov cl, 3
shL al, cl
```

; $10 \times 2^3 = 80$



// SHL used for Fast multiplication //

when we want to mul a num by Power of 2.