Intel Assembly III
- Data Transfer Instructions

Data Transfer Instructions

- General data movement
- Exchange
- Stack manipulation
- Type conversion

General Data Movement– mov Instruction

- mov : covered earlier
- Type of data movement

Type of Data Movement	Source → Destination
From memory to a register	Memory location → General-purpose register Memory location → Segment register
From a register to memory	General-purpose register → Memory location Segment register → Memory location
Between registers	General-purpose register → General-purpose register General-purpose register → Segment register Segment register → General-purpose register General-purpose register → Control register Control register → General-purpose register General-purpose register → Debug register Debug register → General-purpose register
Immediate data to a register	Immediate → General-purpose register
Immediate data to memory	Immediate → Memory location

General Data Movement– Conditional Move Instructions

- ***cmov*** (Pentium and up only):
  - moves data only if a condition is true.
  - Conditions are set by a previous instruction and include *Carry, Zero, Sign, Overflow* and *Parity*:

```
cmovz eax, ebx ;Move if Zero flag is set else do nothing.
```

  - There are many variations of this instruction

Variations of Conditional Move

Instruction Mnemonic	Status Flag States	Condition Description
Unsigned Conditional Moves		
CMOVA/CMOVNBE	(CF or ZF) = 0	Above/not below or equal
CMOVAE/CMOVNB	CF = 0	Above or equal/not below
CMOVNC	CF = 0	Not carry
CMOVB/CMOVNAE	CF = 1	Below/not above or equal
CMOVC	CF = 1	Carry
CMOVBE/CMOVNA	(CF or ZF) = 1	Below or equal/not above
CMOVE/CMOVZ	ZF = 1	Equal/zero
CMOVNE/CMOVNZ	ZF = 0	Not equal/not zero
CMOVP/CMOVPE	PF = 1	Parity/parity even
CMOVNP/CMOVPO	PF = 0	Not parity/parity odd
Signed Conditional Moves		
CMOVGE/CMOVNL	(SF xor OF) = 0	Greater or equal/not less
CMOVL/CMOVNGE	(SF xor OF) = 1	Less/not greater or equal
CMOVLE/CMOVNG	((SF xor OF) or ZF) = 1	Less or equal/not greater
CMOVO	OF = 1	Overflow
CMOVNO	OF = 0	Not overflow
CMOVS	SF = 1	Sign (negative)
CMOVNS	SF = 0	Not sign (non-negative)

Exchange Instructions

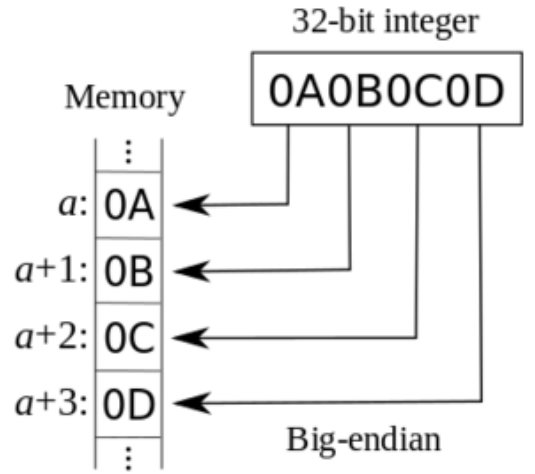
- ***xchg***:
  - Exchanges the contents of a register with the contents of any other register or memory location.
  - It can NOT exchange segment registers or memory-to-memory data.
  - Byte, word and doublewords can be exchanged using any addressing mode (except immediate, of course).

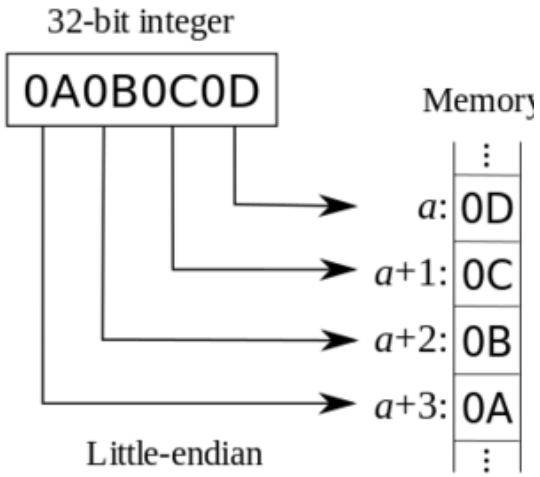
```
xchg  edx, esi      ;Exchange edx and esi
```

- ***bswap*** (80486 and up only):
  - Swaps the first byte with the forth, and the second byte with the third.
  - Used to convert between little endian and big endian:

Cf) Endianness

- **Endianness** refers to the order of the bytes, comprising a word, in computer memory. It also describes the order of byte transmission over a digital link.
- Big-endian : the most significant byte of a word is stored at a particular memory address and the subsequent bytes are stored in the following higher memory addresses, the least significant byte thus being stored at the highest memory address.
- Little-endian : reverses the order and stores the least significant byte at the lower memory address with the most significant byte being stored at the highest memory address.





Cf) Endianness

- Big-endian is the most common format in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as **network byte order**.
- Little-endian storage is popular for microprocessors, in part due to significant influence on microprocessor designs by Intel (the Intel x86 processors use little-endian)
- Mixed forms also exist, for instance the ordering of bytes in a 16-bit word may differ from the ordering of 16-bit words within a 32-bit word. Such cases are sometimes referred to as **mixed-endian** or **middle-endian**. There are also some **bi-endian** processors that operate in either little-endian or big-endian mode.

Stack Manipulation - Push and Pop

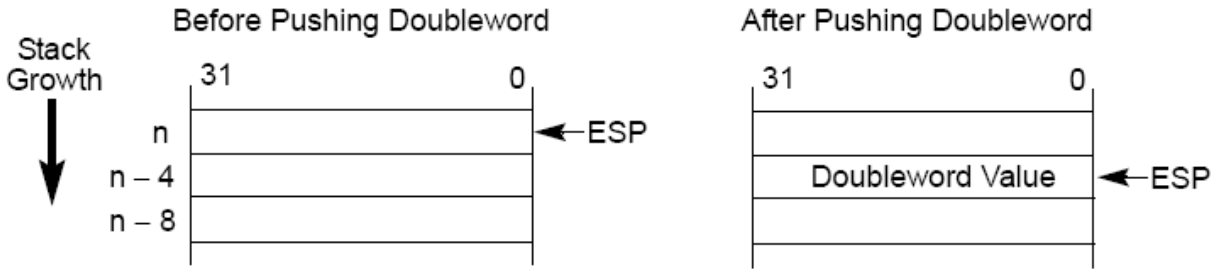
- The *push*, *pop*, *pusha*, and *popa* move data to and from the stack

- *push*
  - The source of the data may be any 16- or 32-bit register, immediate data, any segment register, any word or doubleword of memory data

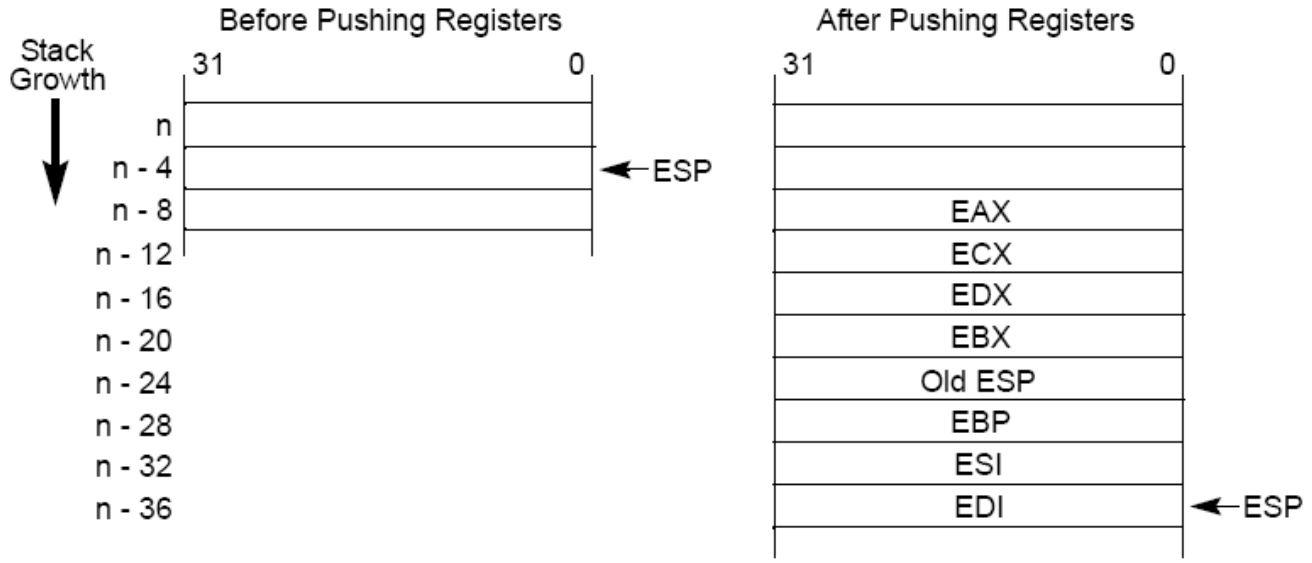
- *pop*
  - The source of the data may be any 16- or 32-bit register, any segment register (except for cs), any word or doubleword of memory data.

Operation of *push*

- *push*

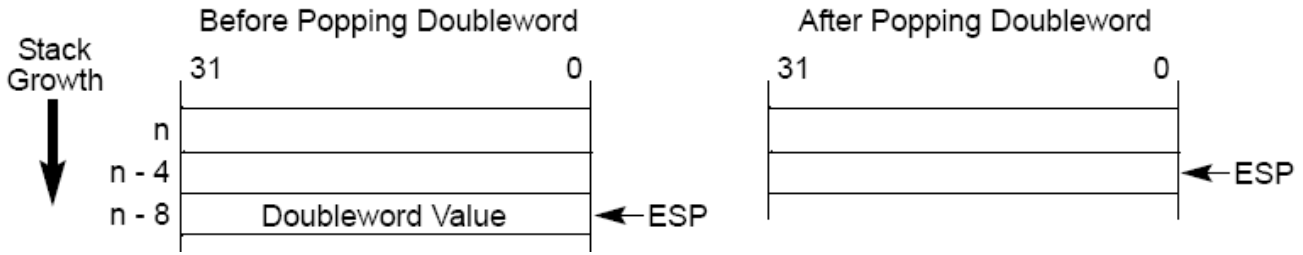


- *pusha*

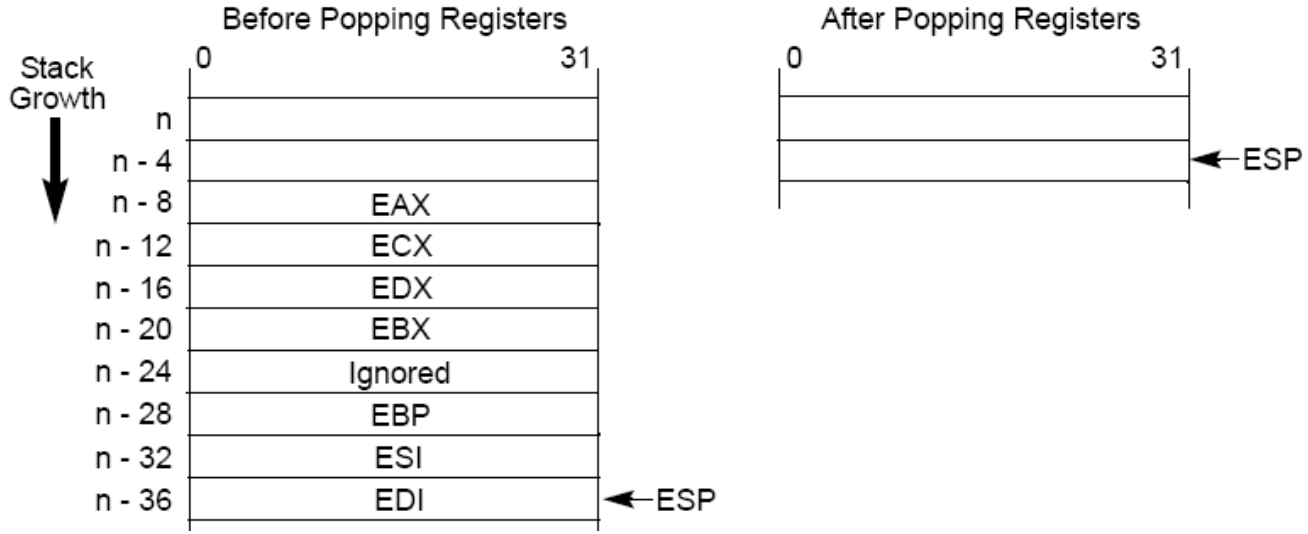


Operation of *pop*

- *pop*



- *popa*



Address Loading Instructions

- ***lea***
  - Loads any 32-bit register with the address of the data, as determined by the instruction addressing mode.
- ***lds*** and ***les***:
  - Load a 32-bit offset address and then **ds** or **es** from a 48-bit memory location.
- ***lfs***, ***lgs*** and ***lss*** (80386 and up):
  - Load any 32-bit offset address and then **fs**, **gs** or **ss** from a 48-bit memory location.

```
lea  eax,[ebx+ecx*4+100]  ;Loads eax with computed address.
lds    edi, LIST          ;Loads edi and ds.
lfs    esi, DATA1         ;Loads esi and fs.
```

- NOTE: ***lea*** calculates the **ADDRESS** given by the right arg and stores it in the left arg!

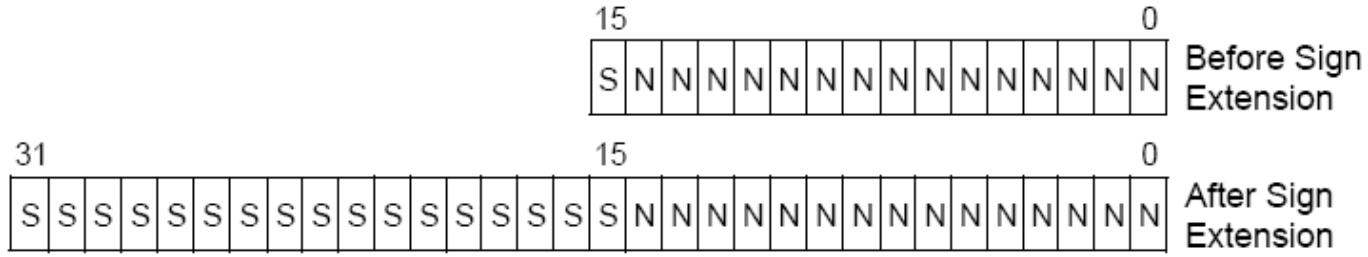
Address Loading Instructions

- *lea* versus *mov*

```
lea  ebx, [edi]     ;Load the contents of edi into ebx.  (1)
mov  ebx, [edi]     ;Load the value at edi into ebx.     (2)
```

Type Conversion Instructions

- Simple conversion
  - cbw (convert byte to word)
    - AX ← sign-extend of AL
  - cwde (convert word to doubleword extended)
    - EAX ← sign-extend of AX
  - cwd (convert doubleword)
    - DX:AX ← sign-extend of AX
    - used before 16-bit division
  - cdq (convert doubleword to quadword)
    - EDX:EAX ← sign-extend of EAX
    - used before 32-bit division



Type Conversion Instructions

- Move with sign or zero extension
  - ***movsx*** and ***movzx*** (80386 and up only)
  - Move-and-sign-extend and Move-and-zero-extend:

```
movsx  cx, bl          ;Sign-extends bl into cx
movzx  eax, DATA2      ;Zero extends word at DATA2 in eax.
```

String Operations

- ***movs, lods, stos, ins, outs***
  - Allow data transfers of a byte, a word or a double word, or if repeated, a block of each of these.
  - The *D* flag-bit (direction), ***esi*** and ***edi*** are implicitly used.
    - *D* = *0*: Auto increment ***edi*** and ***esi***
    - Use ***cld*** instruction to clear this flag
    - *D* = *1*: Auto decrement ***edi*** and ***esi***
    - Use ***std*** instruction to set it.
- *edi*:
  - Accesses data in the extra segment. Can NOT override.
- *esi*:
  - Accesses data in the data segment. Can be overridden with segment override prefix.

lods

- ***lods***:
  - Loads ***al, ax*** or ***eax*** with data stored at the data segment (or extra segment) + offset given by ***esi***
  - ***esi*** is incremented or decremented afterwards

```
lodsb                   ;al=ds:[esi]; esi=esi+/-1
lodsd                   ;eax=ds:[esi]; esi=esi+/-4
es lodsb DATA1          ;Override ds.
```

stosb

- ***stosb***:
  - Stores ***al, ax*** or ***eax*** to the extra segment (***es***) + offset given by ***edi*** (*es* cannot be overridden)
  - ***edi*** is incremented or decremented afterwards:

```
stosb                ;es:[edi]=al; edi=edi+/-1
stosd                ;es:[edi]=eax; edi=edi+/-4
```

movs

- ***movs***:
  - Moves a byte, word or doubleword from data segment and offset ***esi*** to extra segment and offset ***edi***
  - Increments/decrements both ***edi*** and ***esi***:

```
movsb       ;es:[edi]=ds:[esi]; edi+/-=1; esi+/-=1
movsd       ;es:[edi]=ds:[esi]; edi+/-=4; esi+/-=4
```

rep Prefix

- ***rep*** prefix:
  - Executes the instruction ***ecx*** times.
  - NOTE: ***rep*** does not make sense with the ***lodsb*** instruction.

```
mov  edi, 0              ;Offset 0.
mov  ecx, 25*80          ;Load count.
mov  eax, 0720H          ;Load value to write.
rep  stosw
```