Intel Assembly

Data Movement Instruction:

- mov (covered already)
- push, pop
- lea (mov and offset)
- lds, les, lfs, lgs, lss
- movs, lods, stos
- ins, outs
- xchg, xlat
- lahf, sahf (not covered)
- in, out
- movsx, movzx
- bswap
- cmov

Stack Instructions

There are six forms of the **push** and **pop** instructions.

Register, memory (memory-to-memory copy), immediate, segment register, flags, and all registers

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

pushad pushes **eax**, **ecx**, **edx**, **ebx**, **esp**, **ebp**, **edi** and **esi** where the value of **esp** saved on the stack is its value before the **pushad**.

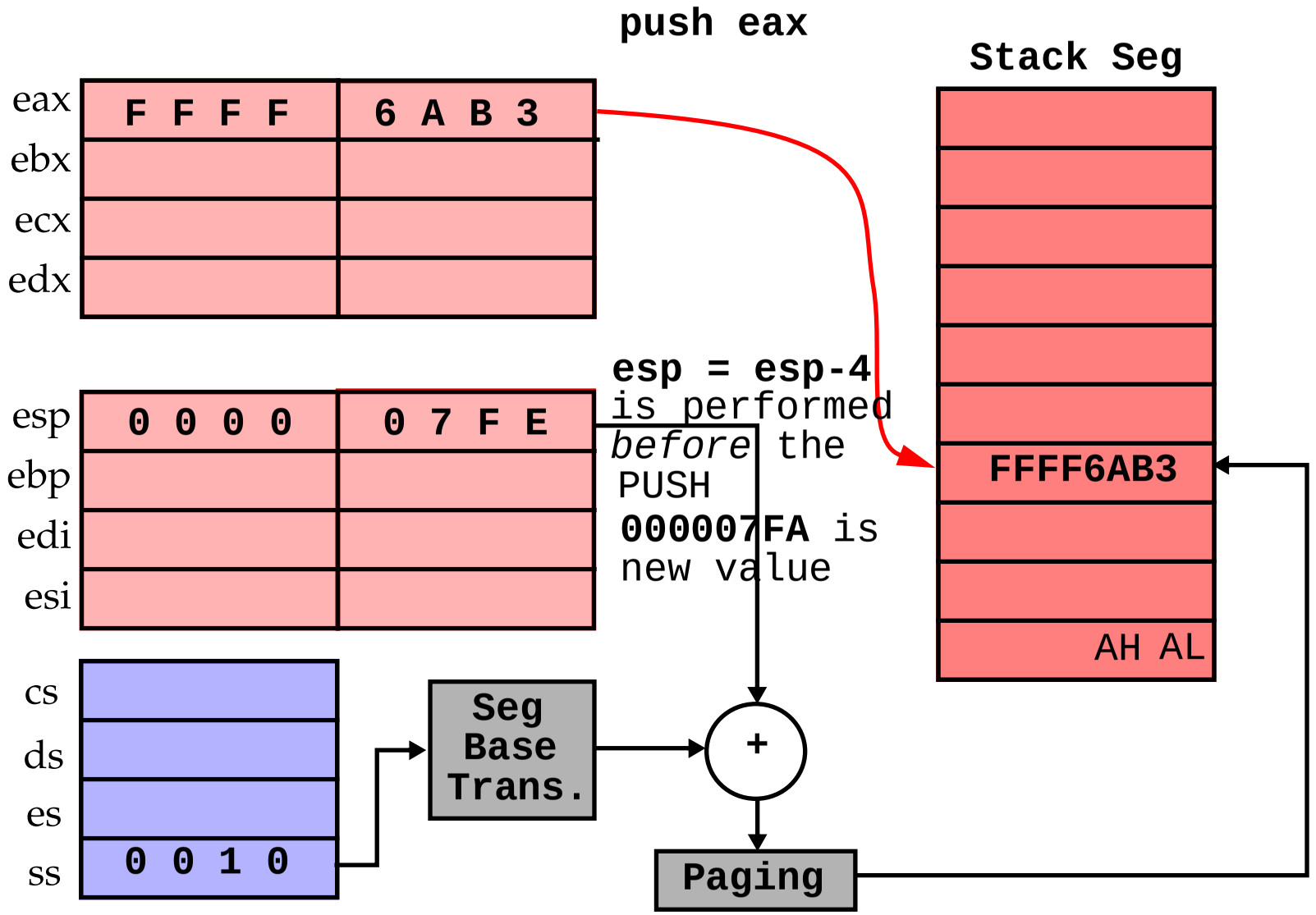
pop:

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.

Stack Instructions

push:



Address Loading Instructions

Load-Effective Address.

- **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

Load-Effective Address.

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)
mov ebx, edi     ;Move the contents of edi into ebx. (3)
```

1 and 3 are equivalent.

So what are the differences?

3 is faster than 1 and is preferred.

However, **mov** only works with single args and cannot be used with *LIST*[**edi**].

lea can take any address, e.g., **lea esi**, [**ebx** + **edi**].

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*: Autoincrement **edi** and **esi**.
  Use **cld** instruction to clear this flag.
- *D = 1*: Autodecrement **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.

String Operations

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:

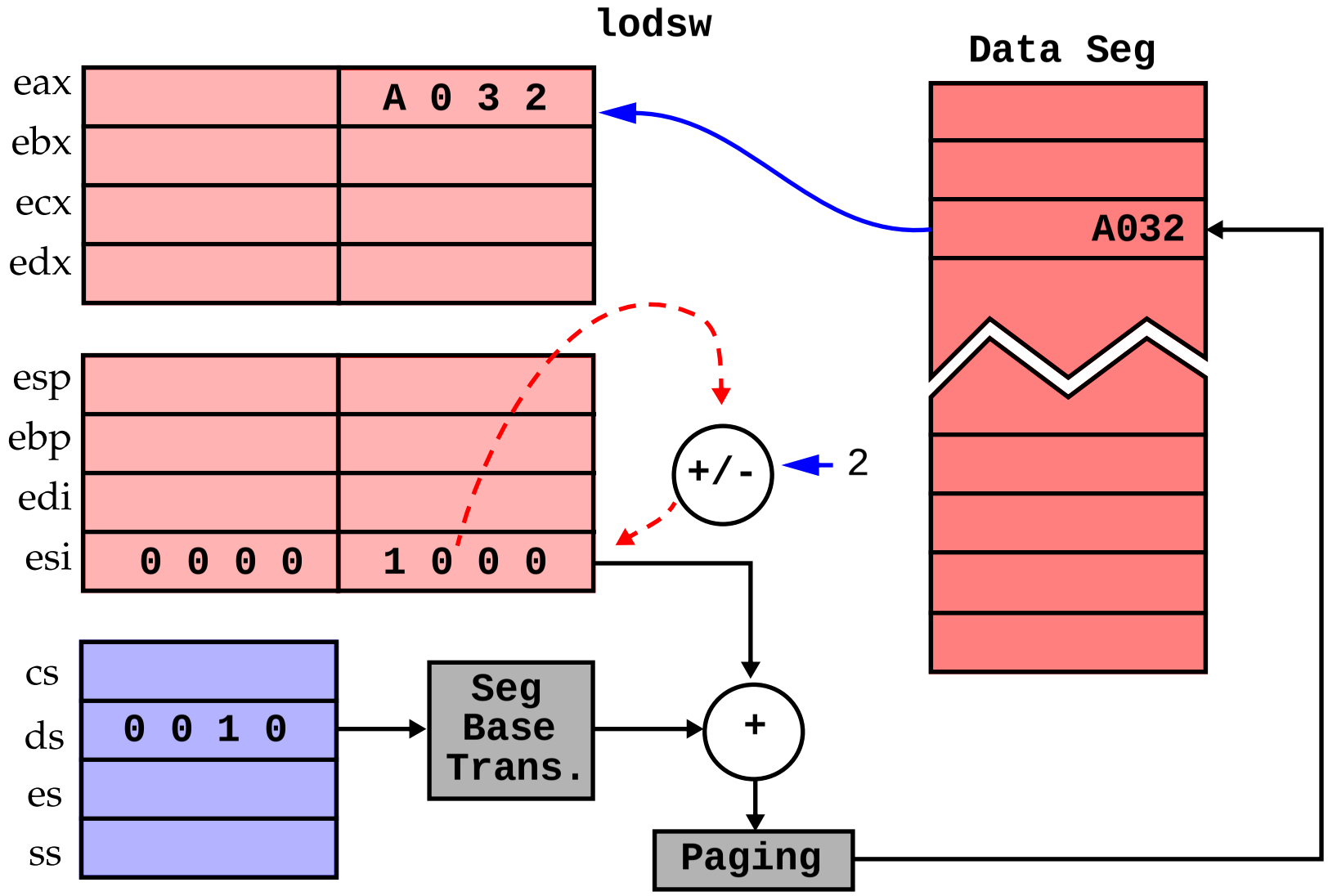
Stores **al**, **ax** or **eax** to the extra segment (es) + offset given by **edi**. es cannot be overridden.

edi is incremented or decremented afterwards:

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

String Operations

An example



String Operations

rep prefix:

Executes the instruction **ecx** times.

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

NOTE: **rep** does not make sense with the **lodsb** instruction.

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
```

String Operations

ins/outs:

Transfers a byte, word or doubleword of data from/to an I/O device into/out of the extra/data segment + offset **edi**/**esi**, respectively.

The I/O address is stored in the **edx** register.

```
insb                ;es:[edi]=[edx]; edi+/-=1
insd                ;es:[edi]=[edx]; edi+/-=4
insw W1, W2         ;es:[edi]=[edx]; edi+/-=2; esi+/-=2
outsb               ;[edx]=ds:[esi]; esi=esi+/-1
```

Miscellaneous Data Transfer Operations:

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
```

Miscellaneous Data Transfer Operations

in and **out**:

Transfers a byte, word or doubleword of data from/to an I/O device into/out of **al**, **ax** and **eax**, respectively.

Memory operations are not available (as they are in **ins** and **outs**):

```
in al, p8      ;8-bits are saved to al from I/O port p8.
out dx, eax    ;32-bits are written to port dx from eax.
```

- Two forms: Fixed-port addressing (*8-bit* port value encoded in instruction) and variable-port addressing (*16-bit* port value stored in **dx**).

in and **out** example:

```
        in al, 61H          ;Read port 61H (speaker).
        or al, 3            ;Set rightmost two bits.
        out 61H, al         ;Turn on the speaker.
        mov cx, 1000H       ;Delay count.
L1:     loop L1             ;Time delay.
        in al, 61H          ;Read port 61H.
        and al, 0FCH        ;Clear rightmost two bits.
        out 61H, al         ;Speaker off.
```

The old contents of the higher-order bits of the port are preserved.

Miscellaneous Data Transfer Operations

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.
```

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:

cmov (Pentium and up only):

These instructions move 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 (see Appendix B of text).

Assembler Directives

Segment Override Prefix:

Allows the programmer to override the default segment.

```
es outsb
es cmpsb
```

Code and Data Segment Declarations:

```
section .data           ;Define a data segment.
section .text           ;Define a code segment.
```

Procedure Calls

```
push args...            ;Push the arguments on stack.
call procedure_name     ;Call the procedure
add esp, #              ;Restore the stack pointer.
```

See nasm slides and documentation on other details.