ASCII

Review conversion from one base to another in text as well as two's complement.

Table 1: ASCII (American Standard Code for Information Interchange)

Dec	Hex	Sym	Dec	Hex	Sym	Dec	Hex	Sym	Dec	Hex	Sym
0	0	NUL	32	20		64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o

ASCII

Table 2: ASCII (American Standard Code for Information Interchange)

Dec	Hex	Sym	Dec	Hex	Sym	Dec	Hex	Sym	Dec	Hex	Sym
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	

Assembly Directives

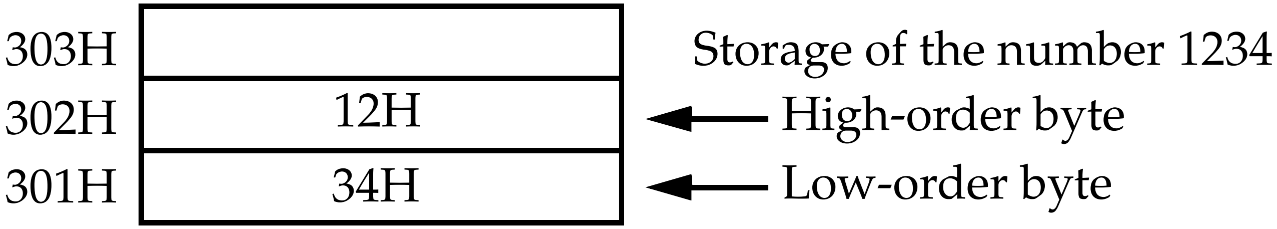
ASCII: Stored using an assembler directive *db*:

```
floatstr db 'Float number -> %f ', 10, 0
main1_str: db '     Rectangular Areas', 10, 0
temp_buf:              times 200 db 0
temp_buf_size:         equ $-temp_buf
```

Word-sized (*dw*) and doubleword-sized data (*dd*):

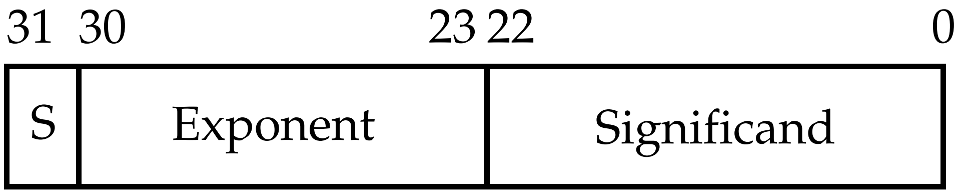
```
neg_exponent:          dd -100
```

Little endian: Least significant byte is always stored in the lowest memory location.



303H		Storage of the number 1234
302H	12H	High-order byte
301H	34H	Low-order byte

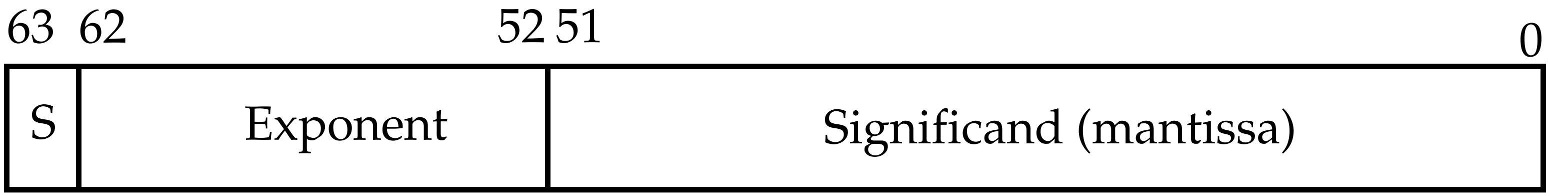
Floating Point Formats



31 30 23 22 0

S Exponent Significand

Single Percision



63 62 52 51 0

S Exponent Significand (mantissa)

Double Precision

For single percision, the sign bit + 8-bit exponent + 24-bit mantissa = 33 bits !

The mantissa has a hidden 1 bit in the leftmost position that allows it to be stored as a 23-bit value.

The mantissa is first normalized to be >= 1 and < 2, e.g., 12 in binary is 1100, normalized is 1.1 X 23.

The exponent is also biased by adding 127 (single) or 1023 (double), e.g. the 3 in the previous example is stored as 127 + 3 = 130 (82H).

Floating Point Formats and Directives

Dec	Bin	Normal	Sign	Expon	Mantissa
+12	1100	1.1×2^3	0	10000010	1000000 00000000 00000000

There are two exceptions:

The number 0.0 is stored as all zeros.

The number infinity is stored as all ones in the exponent and all zeros in the mantissa. (The sign bit is used to indicate + or - infinity.)

Directive is ***dd*** for single, ***dq*** for double and ***dt*** for 10 bytes:

dd `1.2`
dq `1.e+10`
dt `3.141592653589793238462`

Intel Assembly

Format of an assembly instruction:

```
LABEL     OPCODE     OPERANDS     COMMENT
DATA1       db       00001000b    ;Define DATA1 as decimal 8
START:      mov      eax, ebx     ;Copy ebx to eax
```

LABEL:

Stores a symbolic name for the memory location that it represents.

OPCODE:

The instruction itself.

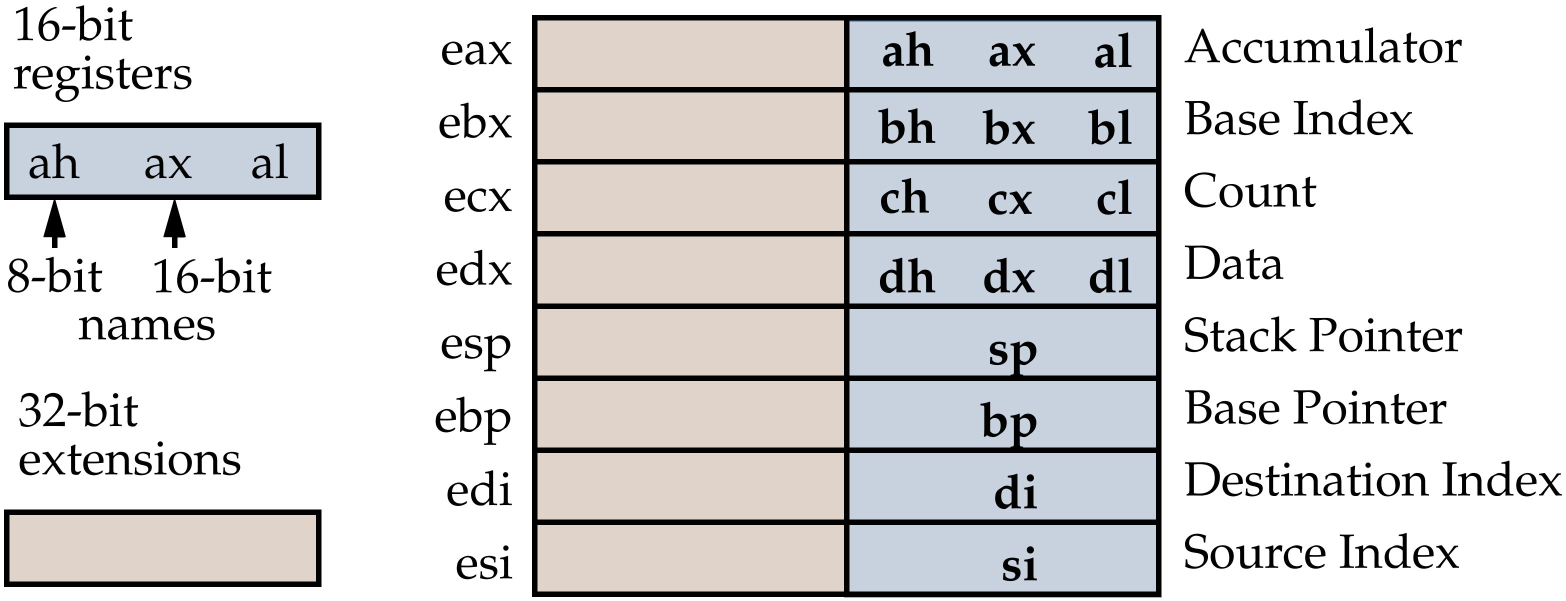
OPERANDS:

A register, an immediate or a memory address holding the values on which the operation is performed.

There can be from 0 to 3 operands.

Data Addressing Modes

Data registers:



16-bit registers

ah ax al

8-bit 16-bit names

32-bit extensions

eax		**ah**	**ax**	**al**	Accumulator
ebx		**bh**	**bx**	**bl**	Base Index
ecx		**ch**	**cx**	**cl**	Count
edx		**dh**	**dx**	**dl**	Data
esp			**sp**		Stack Pointer
ebp			**bp**		Base Pointer
edi			**di**		Destination Index
esi			**si**		Source Index

Let's cover the data addressing modes using the ***mov*** instruction.

Data movement instructions move data (bytes, words and doublewords) between registers and between registers and memory.

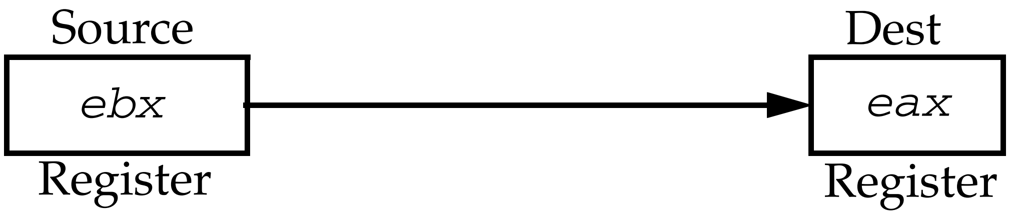
Only the ***movs*** (strings) instruction can have both operands in memory.

Most data transfer instructions do not change the **EFLAGS** register.

Data Addressing Modes

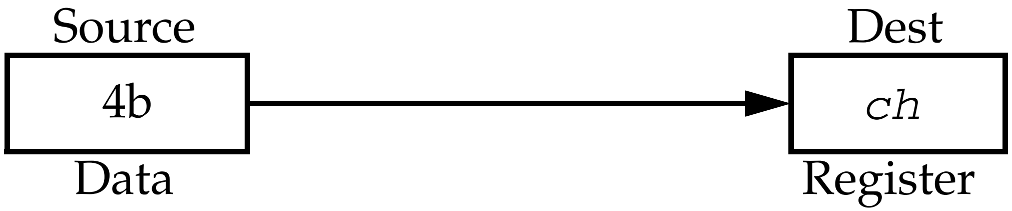
- ***Register***

```
mov eax, ebx
```



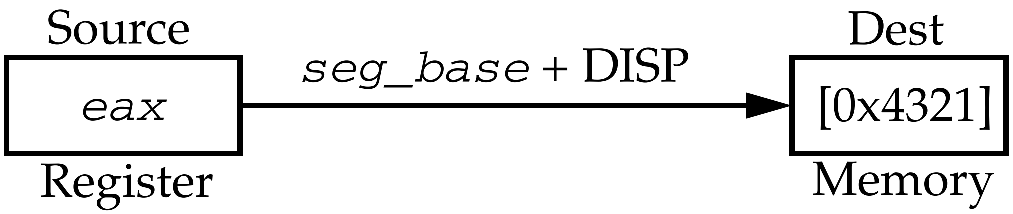
- ***Immediate***

```
mov ch, 0x4b
```



- ***Direct*** (eax), D***isplacement*** (other regs)

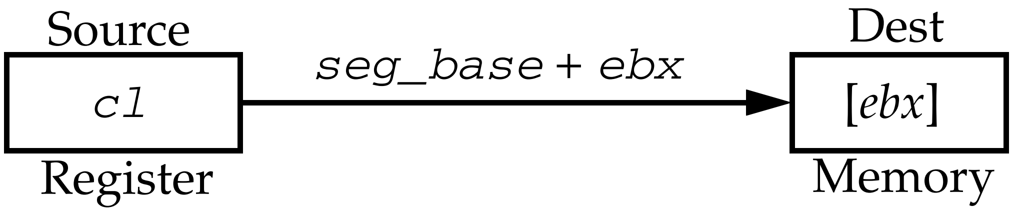
```
mov [0x4321], eax
```



Data Addressing Modes

- ### *Register Indirect*

mov [ebx], cl



Source

cl

Register

$seg\_base + ebx$

Dest

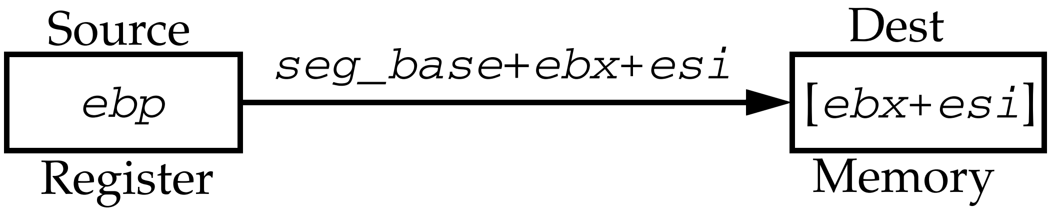
[*ebx*]

Memory

Any of *eax*, *ebx*, *ecx*, *edx*, *ebp*, *edi* or *esi* may be used.

- ### *Base-plus-index*

mov [ebx+esi], ebp



Source

ebp

Register

$seg\_base+ebx+esi$

Dest

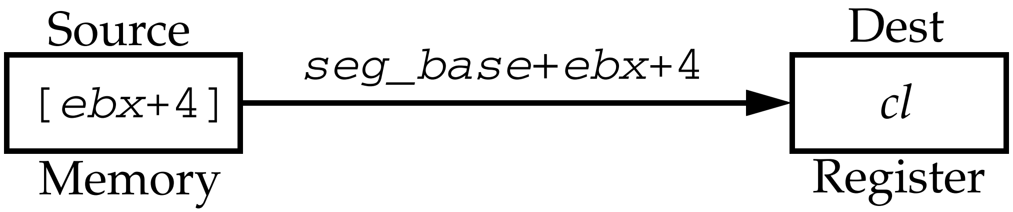
[ebx+esi]

Memory

Any combination of *eax*, *ebx*, *ecx*, *edx*, *ebp*, *edi* or *esi*.

- ### *Register relative*

mov cl, [ebx+4]



Source

[ebx+4]

Memory

$seg\_base+ebx+4$

Dest

cl

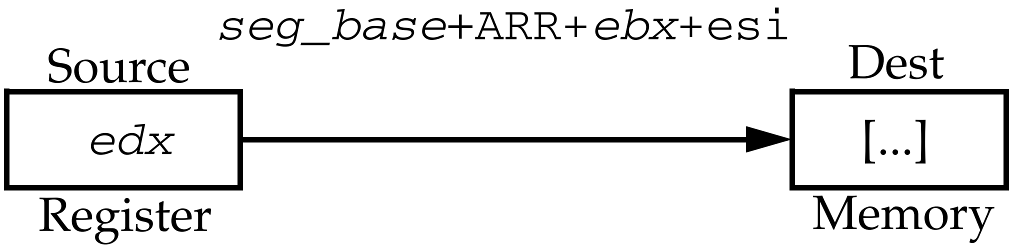
Register

A second variation includes: ***mov*** *eax*, [ARR+*ebx*]

Data Addressing Modes

❍ *Base relative-plus-index*

mov `[ARR+`*`ebx`*`+`*`esi`*`], `*`edx`*



seg_base+ARR+*ebx*+esi

Source

edx

Register

Dest

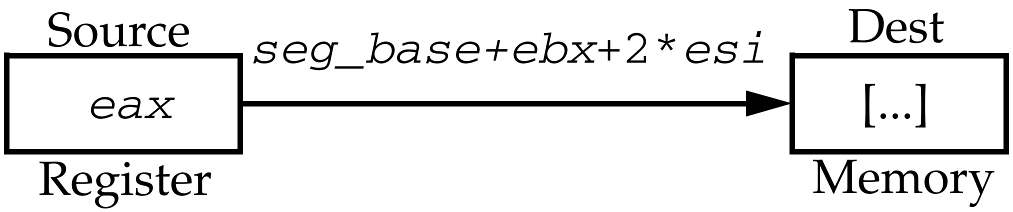
[...]

Memory

A second variation includes: ***mov*** *eax*, [*ebx*+*edi*+4]

❍ *Scaled-index*

mov `[`*`ebx`*`+2*`*`esi`*`], `*`eax`*



Source

*seg_base+ebx+2*esi*

eax

Register

Dest

[...]

Memory

A second variation includes: ***mov*** *eax*, *ebx**2+*ecx*+offset

Scaling factors can be 2X, 4X or 8X.

Data Addressing Modes

Register addressing

Note: ***mov*** really COPIES data from the source to destination register.

- Never mix an 16-bit register with a 32-bit, etc.

  For example

```
mov  eax, bx      ;ERROR: NOT permitted.
```

- None of the ***mov*** instruction effect the EFLAGS register.

Immediate addressing:

The value of the operand is given as a constant in the instruction stream.

```
mov  eax, 0x12345
```

- Use ***b*** for binary, ***q*** for octal and nothing for decimal.
- ASCII data requires a set of apostrophes:

```
mov  eax, 'A'     ;Moves ASCII value 0x41 into eax.
```

Data Addressing Modes

Register and immediate addressing example:

```
       global main
       section .text  ;start of the code segment.
main:
       mov  eax, 0     ;Immediate addressing.
       mov  ebx, 0x0000
       mov  ecx, 0
       mov  esi, eax   ;Register addressing.
       ...
```

Direct addressing:

Transfers between memory and *al*, *ax* and *eax*.

Usually encoded in 3 bytes, sometime 4:

```
mov al, DATA1      ;Copies a byte from DATA1.
mov al, [0x4321]   ;Some assemblers don’t allow this.
mov al, ds:[0x1234]
mov DATA2, ax      ;Copies a word to DATA2.
```

Data Addressing Modes

Displacement:

```
mov cl, DATA1        ;Copies a byte from DATA1.
mov edi, SUM         ;Copies a doubleword from SUM.
```

Displacement instructions are encoded with up to 7 bytes (32 bit register and a 32 bit displacement).

Direct and displacement addressing example:

```
                        global main
0000                    section .data
0000 10          DATA1  db      0x10
0001 00          DATA2  db      0

0000                    section .text
                 main:
0017 A0 0000 R          mov al, DATA1
001A 8B 1E 0001 R       mov bx, DATA2
```

Note: Direct addressing (using **al**) requires 3 bytes to encode while Displacement (using **bx**) requires 4.

Data Addressing Modes

Register Indirect addressing:

Offset stored in a register is added to the segment register.

```
mov ecx, [ebx]
mov [edi], [ebx]
```

The memory to memory ***mov*** is allowed with string instructions.

Any register EXCEPT **esp** for the 80386 and up.

For **eax**, **ebx**, **ecx**, **edx**, **edi** and **esi**: The data segment is the default.

For **ebp**: The stack segment is the default.

Some versions of register indirect require special assembler directives ***byte, word***, or ***dword***

```
mov al, [edi]        ;Clearly a byte-sized move.
mov [edi], 0x10      ;Ambiguous, assembler can't size.
```

Does [*edi*] address a byte, a word or a double-word?

Use:

```
mov byte [edi], 0x10     ;A byte transfer.
```

Data Addressing Modes

Base-Plus-Index addressing:

Effective address computed as:
seg_base + base + index.

Base registers: Holds starting location of an array.

- **ebp** (stack)
- **ebx** (data)
- Any 32-bit register except **esp**.

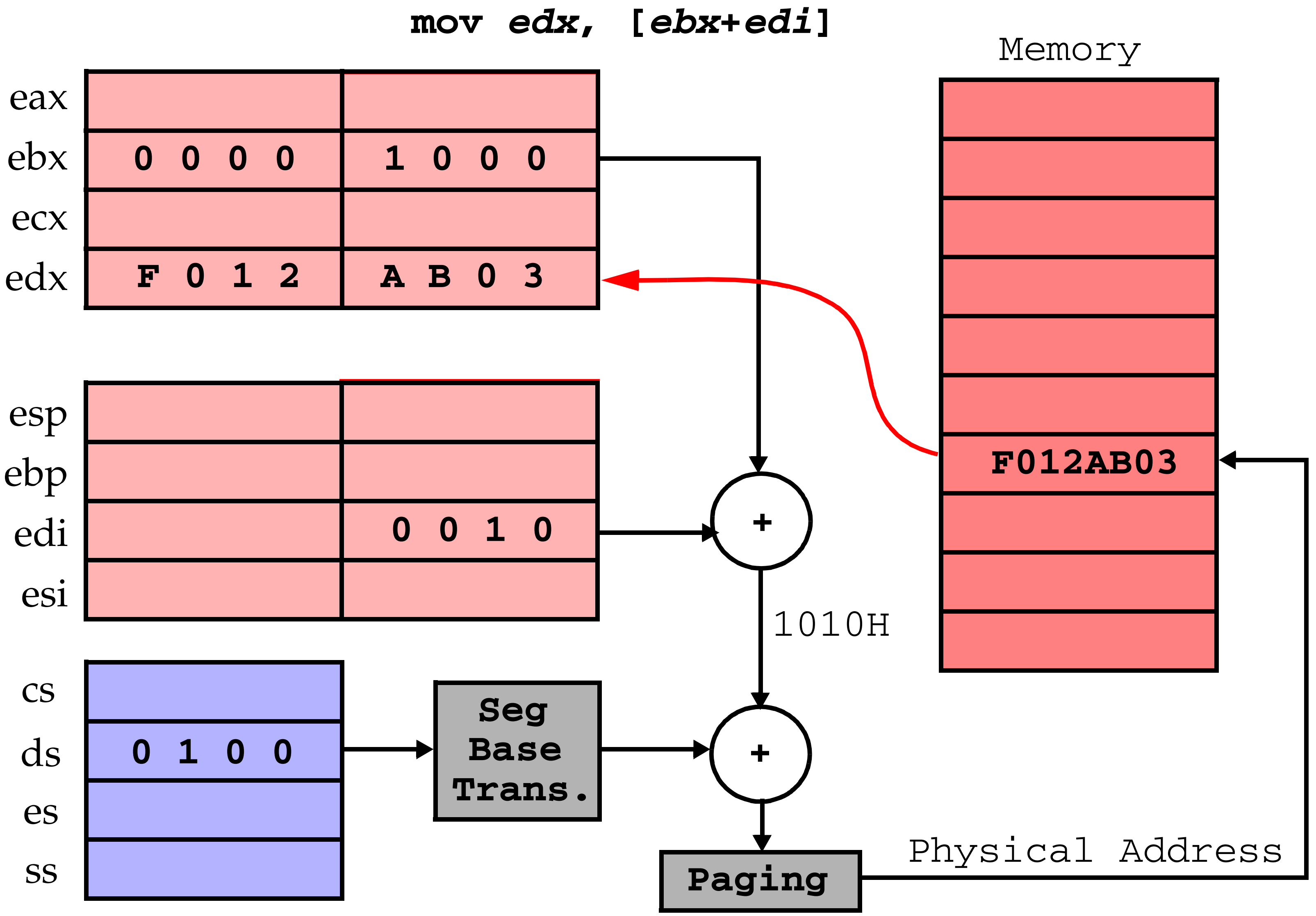
Index registers: Holds offset location.

- **edi**
- **esi**
- Any 32-bit register except **esp**.

```
mov  ecx,[ebx+edi]    ;Data segment copy.
mov  ch, [ebp+esi]    ;Stack segment copy.
mov  dl, [eax+ebx]    ;EAX as base, EBX as index.
```

Data Addressing Modes

Base-Plus-Index addressing:



Data Addressing Modes

Register Relative addressing:

Effective address computed as:

seg_base + base + constant.

```
mov  eax, [ebx+1000H]   ;Data segment copy.
mov  [ARRAY+esi], BL    ;Constant is ARRAY.
mov  edx, [LIST+esi+2]  ;Both LIST and 2 are constants.
mov  edx, [LIST+esi-2]  ;Subtraction.
```

Same default segment rules apply with respect to **ebp**, **ebx**, **edi** and **esi**.

Displacement constant is any *32-bit* signed value.

Base Relative-Plus-Index addressing:

Effective address computed as:

seg_base + base + index + constant.

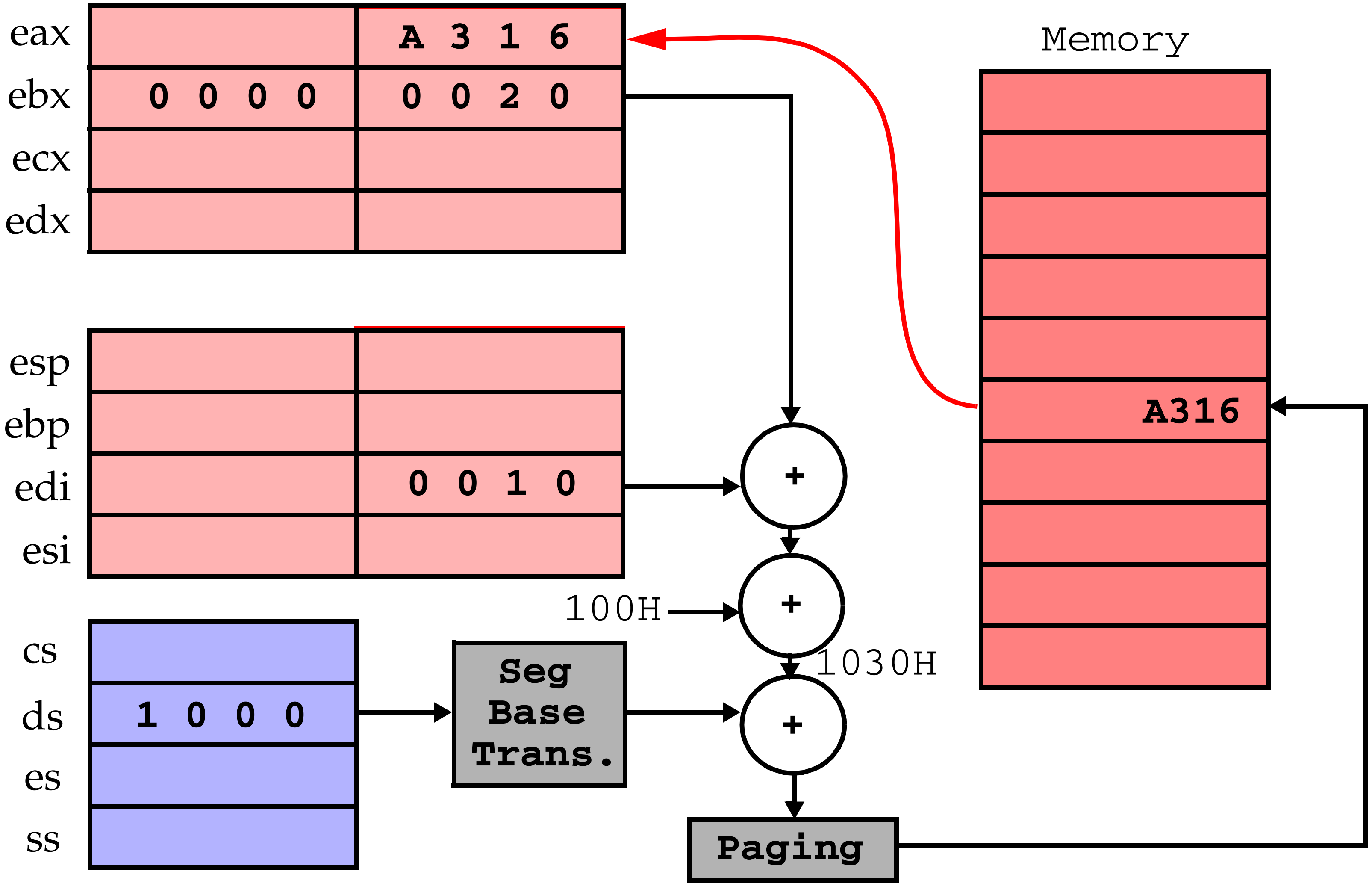
```
mov  dh, [ebx+edi+20H]          ;Data segment copy.
mov  ax, [FILE+ebx+edi]         ;Constant is FILE.
mov  [LIST+ebp+esi+4], dh       ;Stack segment copy.
mov  eax, [FILE+ebx+ecx+2]      ;32-bit transfer.
```

Designed to be used as a mechanism to address a two-dimensional array.

Data Addressing Modes

Base Relative-Plus-Index addressing:

```
MOV ax, [ebx+esi+100H]
```



Data/Code Addressing Modes

Scaled-Index addressing:

Effective address computed as:

seg_base + base + constant*index.

```
mov  eax, [ebx+4*ecx]          ;Data segment DWORD copy.
mov  [eax+2*edi-100H], cx      ;Whow !
mov  eax, [ARRAY+4*ecx]        ;Std array addressing.
```

Code Memory-Addressing Modes:

Used in *jmp* and *call* instructions.

Three forms:

- 🟩 Direct
- 🟩 PC-Relative
- 🟩 Indirect

Direct

Absolute jump address is stored in the instruction following the opcode.

Code Addressing Modes

An *inter*segment jump:

Opcode	Offset (low)	Offset (high)	Segment(low)	Segment(high)
E A	0000	0000	00	10

This ***far jmp*** instruction loads **cs** with 1000H and **eip** with 00000000H.

A ***far call*** instruction is similar.

PC-Relative

A displacement is added to the **EIP** register.

This constant is encoded into the instruction itself, as above.

*Intra*segment jumps:

- Short jumps use a 1-byte signed displacement.
- Near jumps use a 4-byte signed displacement.

The assembler usually computes the displacement and selects the appropriate form.

Code Addressing Modes

Indirect

Jump location is specified by a register.

There are three forms:

- Register:

  Any register can be used: **eax**, **ebx**, **ecx**, **edx**, **esp**, **ebp**, **edi** or **esi**.

```
jmp  eax                  ;Jump within the code seg.
```

- Register Indirect:

  *Intra*segment jumps can also be stored in the data segment.

```
jmp  [ebx]                ;Jump address in data seg.
```

- Register Relative:

```
jmp  [TABLE+ebx]          ;Jump table.
jmp  [edi+2]
```

Stack Addressing Modes

The stack is used to hold temporary variables and stores return addresses for procedures.

- ***push*** and ***pop*** instructions are used to manipulate it.
- ***call*** and ***ret*** also refer to the stack implicitly.

Two registers maintain the stack, **esp** and **ss**.

- A **LIFO** (Last-in, First-out) policy is used.
- The stack grows toward lower address.
- Data may be pushed from any of the registers or segment registers.
  - Data may be popped into any register except **cs.**

```
popfd                 ;Pop doubleword for stack to EFLAG.
pushfd                ;Pushes EFLAG register.
push 1234H            ;Pushes 1234H.
push dword [ebx]      ;Pushes double word in data seg.
pushad                ;eax,ecx,edx,ebx,esp,ebp,esi,edi
pop eax               ;Pops 4 bytes.
```