

INTEL IA-32 ARCHITECTURE

Registers and addressing modes

Richard Hayden (with thanks to **Naranker Dulay**)

rh@doc.ic.ac.uk

<http://www.doc.ic.ac.uk/~rh/teaching>

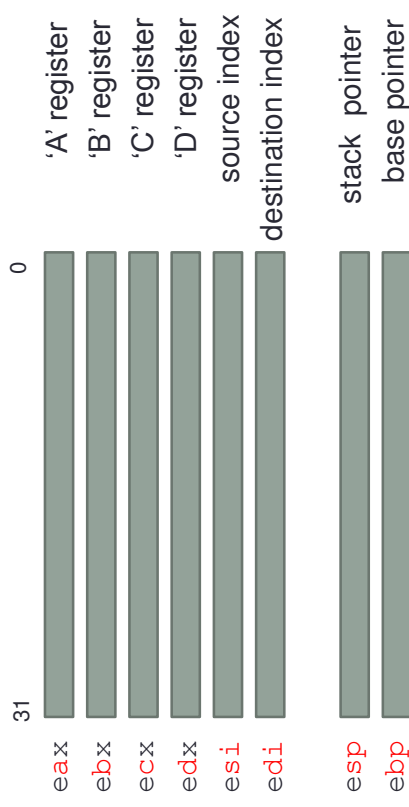
or

<https://www.doc.ic.ac.uk/~wl/teachlocal/arch1>

or

CATE

Registers (32-bit)



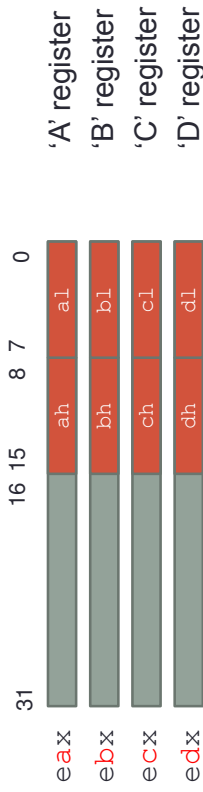
Next three weeks

- Lectures 1 – 5 (today, tomorrow and next Thursday): Intel IA-32 architecture
- Lectures 6 – 7: Representation of floating point numbers
- Lectures 8 – 9: Input / output

Intel architecture family

CPU	Cores	Year	Data bus	Max. mem.	Trans.	Clock speed	Average MIPS	Level-1 Caches
8086	1	1978	16	1Mb	29K	5 – 10MHz	0.8	
80286	1	1982	16	16Mb	134K	8 – 12MHz	2.7	
80386	1	1985	32	4Gb	275K	16 – 33MHz	6	
80486	1	1989	32	4Gb	1.2M	25 – 100MHz	20	8Kb
Pentium	1	1993	64	4Gb	3.1M	60 – 233MHz	100	8Kb+8Kb
Pentium Pro	1	1995	64	64Gb	5.5M	150 – 200MHz	440	8Kb+8Kb+L2
Pentium II	1	1997	64	64Gb	7M	266 – 450MHz	466+	16Kb+16Kb+L2
Pentium III	1	1999	64	64Gb	8.2M	0.5 – 1GHz	1000+	16Kb+16Kb+L2
Pentium 4	1	2001	64	64Gb	42M	1.3 – 3.8GHz	9000+	12Kb+8Kb+L2
Core 2	1 – 4	2006	64	256Tb	291M	1.06 – 3.33GHz	20000+	32Kb+32Kb+L2
Core i7	2 – 6	2008	64	256Tb	781M	1.6 – 3.4GHz	50000+	32Kb+32Kb+L2

Registers (8-bit)



No special names for the two least significant bytes of **esi**, **edi**, **esp** or **ebp**

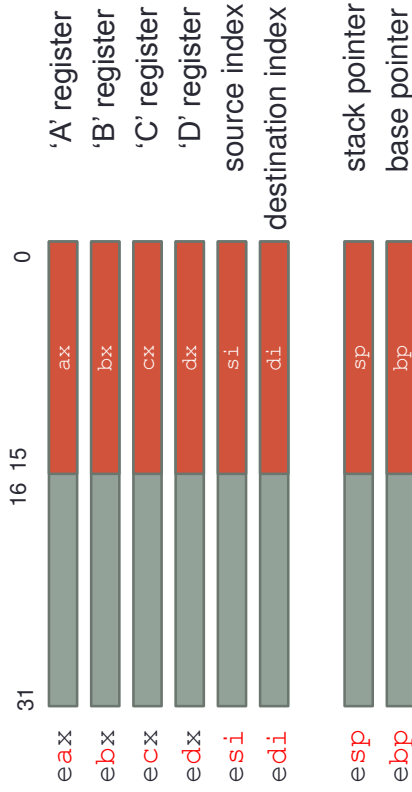
Flags register



Bits represent various CPU state information. Also set/cleared after arithmetic instructions.

- **Zero flag** (bit 6): 1 if result is zero, 0 otherwise
- **Sign flag** (bit 7): MS-bit of result, sign bit if a signed integer
- **Overflow flag** (bit 11): 1 if a signed result overflows, 0 otherwise
- **Carry flag** (bit 0): 1 if an unsigned result overflows, 0 otherwise
- **Parity flag** (bit 2): 1 if LS-byte of result contains an even number of bits, 0 otherwise

Registers (16-bit)



Instruction pointer register



- Holds the address of the next instruction to be executed, also known as “program counter” register
- Rarely manipulated directly by programs
- Updated implicitly by control-flow instructions such as **call**, **jmp** and **ret**
- Used to implement if and while statements; and method calls

Main memory

Byte addressable, little endian, non-aligned accesses allowed

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
12	31	CB	74	EF	F0	0B	23	A4	1F	36	06	FE	7A	FF	45

- Byte at address **9H**?
- Byte at address **BH**?
- Word at address **1H**?
- Word at address **2H**?
- Word at address **6H**?
- Doubleword at address **AH**?
- Quadword at address **6H**?

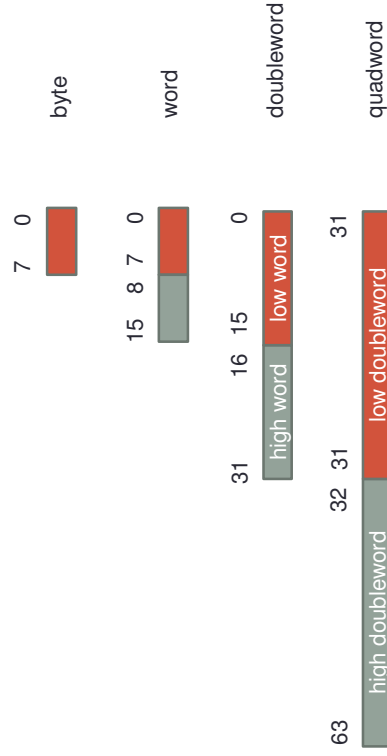
Directives for “global” variables (1)

- Data declaration directives are special assembler commands allowing “global” data (variables) to be declared
- Such data is mapped to fixed memory locations and can be accessed using the name of the variable
- The address of the global variable is then automatically encoded into instructions at assembly time

```

users      db      3           ; byte with value 3
age        dw      21          ; word with value 21
total     dd      999         ; doubleword with value 999
message   db      "hello"    ; 5-byte string hello
sequence  dw      1,2,3       ; 3 words with values 1, 2 and 3
array     times 100 dw 33     ; 100 words each with value 33
    
```

Basic data types



Instruction format

- Most Intel instructions have 2, 1 or 0 operands and have one of the forms:

```

Label: opcode Destination, Source ; comments
Label: opcode Operand             ; comments
Label: opcode                      ; comments
    
```

- `Label` is an optional user-defined identifier whose value is the address of the instruction or data item which follows
- We're using netwide assembler (nasm) which follows Intel syntax. Other syntaxes exist!

Constants

- Can also define named constants – these are **dereferenced at assembly time**

```
dozen      equ    12
century    equ    100
```

Example assembly instructions

```
mov  ah, cl      ; ah = cl
add  ax, [ebx]   ; ax = ax + memory16[ebx]

mov  eax, [ebp+4] ; eax = memory32[ebp+4]
sub  eax, 45     ; eax = eax - 45

mov  byte[ecx], 45 ; memory8[ecx] = 45
add  ch, [22]   ; ch = ch + memory8[22]
```

Directives for “global” variables (2)

- **Uninitialised data** can be reserved with the following directives

```
tiny      resb  10      ; reserve 10 bytes
little    resw  100     ; reserve 100 words (200 bytes)
big       resd  1000    ; reserve 1000 dwords (4K bytes)
```

- Why are these useful when you can just use the ones on the previous slide?

Operands (addressing modes)

- Register operands
e.g. `eax, dx, al, si, bp`
- Immediate operands (constants)
e.g. `23, 67H, 101010B, 'A'`
- Memory operands (cannot be **both** src and dest of an instruction) :

`[BaseReg + Scale * IndexReg + Disp]` where:

BaseReg can be any register

Scale $\in \{1,2,4,8\}$

IndexReg can be any register except `esp`

Disp is a 32-bit constant

e.g. `[24], [bp], [esi+2], [bp+8*di+16]`

Register operands

- Operand is the value in the specified register

```
mov  eax, edx
mov  ah,  bl
mov  esp, ebp
mov  edi, eax
```

- Depending on the instruction and sometimes processor, can be some subset of the **general purpose registers**
- Some instructions also accept `eflags` or `eip`
- Some instructions (e.g. `idiv`) **implicitly** use operands contained in other **specific** registers, e.g. in `ax` and `dx`
- Usually `dest` and `src` operands must be **same size**

Memory operands

- Operand is the value at the specified address
 $[\text{BaseReg} + \text{Scale} * \text{IndexReg} + \text{Disp}]$ where:

`BaseReg` can be any register

`Scale` $\in \{1,2,4,8\}$

`IndexReg` can be any register except `esp`

`Disp` is a 32-bit constant

- **Order is unimportant** to the assembler and components can be omitted to give **different memory addressing modes**
- Size of operands normally **inferred by assembler** – if ambiguous can use **byte, word or dword prefix**, e.g.
`byte [ecx], word 16, dword [ebx + 2*edi+11ist]`

Branching example

```
neg  ax                ; ax = -ax
cmp  eax, ecx          ; compute eax - ecx
                        ; and set EFLAGS
je   end               ; if flags.zf = 1
                        ; then eip = end
call print             ; call method print
                        ; (push return address on
                        ; stack and jump)

end:  ret              ; return from method
```

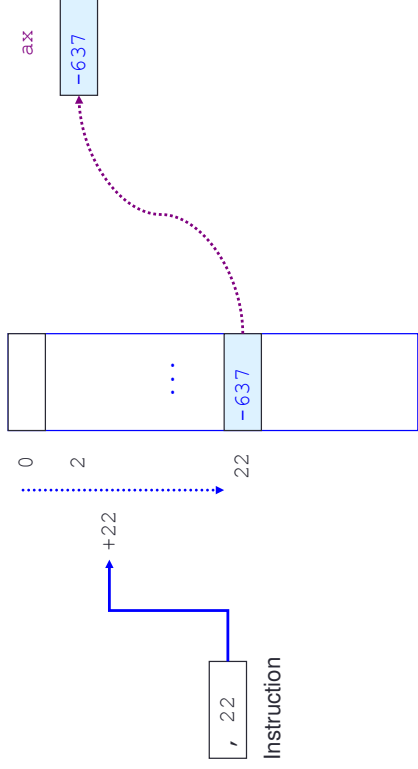
Immediate (constant) operands

- Operand is the constant value specified directly

```
mov  eax, 22
mov  ecx, 22h
mov  al, 'b'
mov  eax, total
mov  ebx, 12345678H
```

- **Encoded directly** into the instruction machine code
- Not normally applicable for **destination operands**
- If a data variable (or code label) is used as an immediate operand then its **address is used**

Example 1: `mov ax, [22]`



Base (register indirect)

- Contents of specified **base register** gives the address

```
mov ax, [ebx]
mov [ebp], al
mov eax, [edi]
mov [esi], ah
mov ebx, [esi]
mov [esp], ecx
```

- Since the value in a base register can be **updated**, this mode can be used to **dynamically address (point to)** variables in memory (e.g. **arrays** and **objects**) based on computed addresses

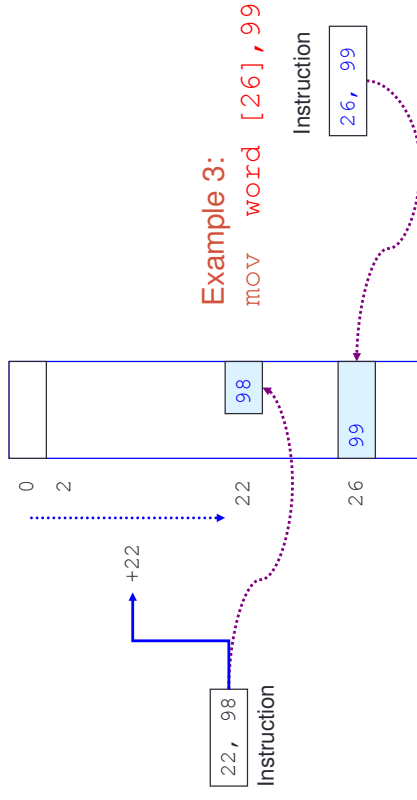
Displacement (direct addressing)

- Specified constant value (called the **displacement**) gives the address

```
mov eax, [22]
mov [22h], esi
mov byte [22], 98
mov eax, [12345678H]
mov cx, [users]
mov [mypointer], ah
```

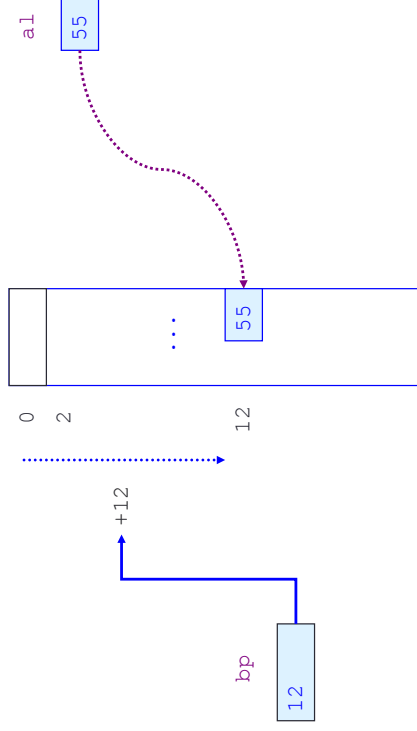
- Displacements are **encoded directly into the instruction**
- Direct addressing allows us to access variables with a fixed address (**global variables**)
- The nasm assembler also allows the displacement to be a **constant expression which it evaluates at assembly time**

Example 2: `mov byte [22], 98`

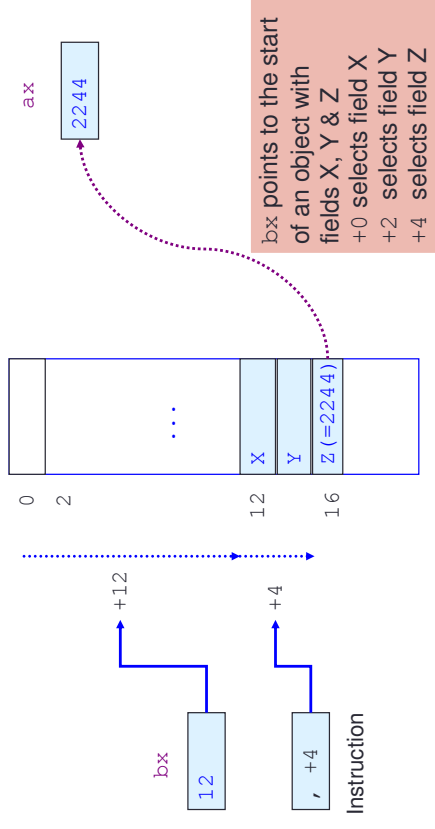


Example 3:
`mov word [26], 99`

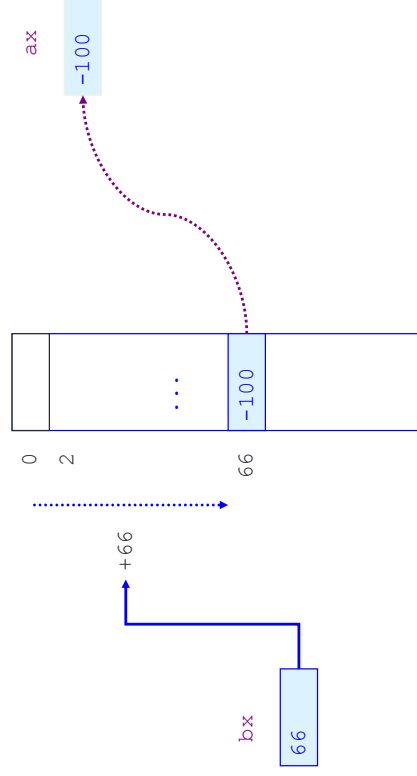
Example 2: `mov [bp], al`



Example 1: `mov ax, [bx+4]`



Example 1: `mov ax, [bx]`



Base + displacement (register relative)

- Sum of specified **base register** and **displacement** gives the address. Displacement can be **negative**.

```

mov ax, [ebx+4]
mov [ebp+2], dh
mov ax, [di-6]
mov dl, [esi+age]
mov [list+ebx], cx
mov dx, [ebp+list-2]

```

- Can be used to access **object fields**:
`base register = start of object`
`displacement = byte position of field within object`
- Can be used to access **array elements**:
`displacement = start of array`
`base register = byte index of array element`
- Can be used to access **parameters** and **local variables** (covered later)

Base + index (based indexed)

- Sum of specified **base register** and **index register** gives the address

```

mov  cx, [bx+di]
mov  [eax+ebx], ecx
mov  ch, [bp+si]
mov  [bx+si], sp
mov  cl, [edx+edi]
mov  [eax+ebx], ecx
mov  [bp+di], cx
    
```

Can be used to access **array elements** where **start of array is dynamically determined** at run-time:
base register = start of array
index register = byte index of array element

Base + index + disp (based relative index)

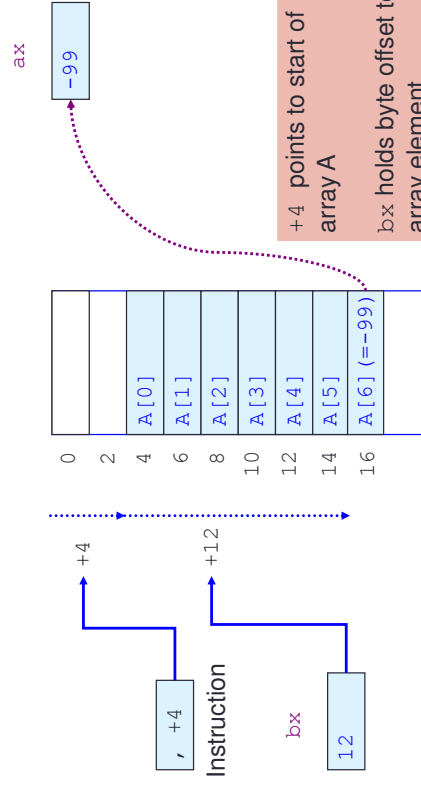
- Sum of specified **base register**, **index register** and **displacement** gives the address

```

mov  ax, [bp+di+10]
mov  dh, [bx+di-6]
mov  [list+bp+di], dx
mov  eax, [ebx+ecx+list+2]
    
```

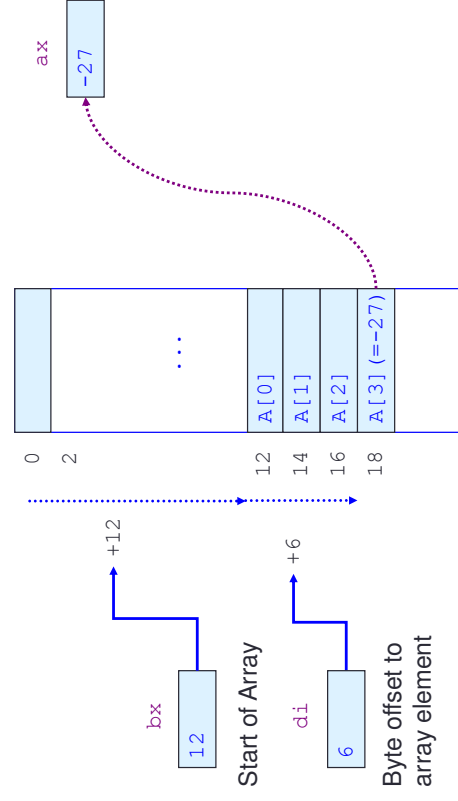
Also known as **relative based index**
 Can be used to access **arrays of objects, arrays within objects and arrays on the stack**

Example 2: mov ax, [bx+4]



+4 points to start of array A
 bx holds byte offset to array element

Example 1: mov ax, [bx+di]



Byte offset to array element

(Scale * index) + disp (scaled indexed)

- Product of **index register** and a **constant (2, 4 or 8)** added to specified **displacement** gives the **address**

```

mov  eax, [4*ecx+4]
mov  [2*ebx], cx
mov  [list+2*ebx], dx
mov  eax, [4*edi+list]

```

- Supports **efficient access to array elements** when the element size is 2, 4 or 8 bytes, e.g.:
displacement = start of array
index register = position of array element
scale = element size in bytes (only 2, 4 or 8)

Base + (scale * index) + disp

- Product of **index register** and a **constant (2, 4 or 8)** added to specified **base register** and **displacement** gives the **address**

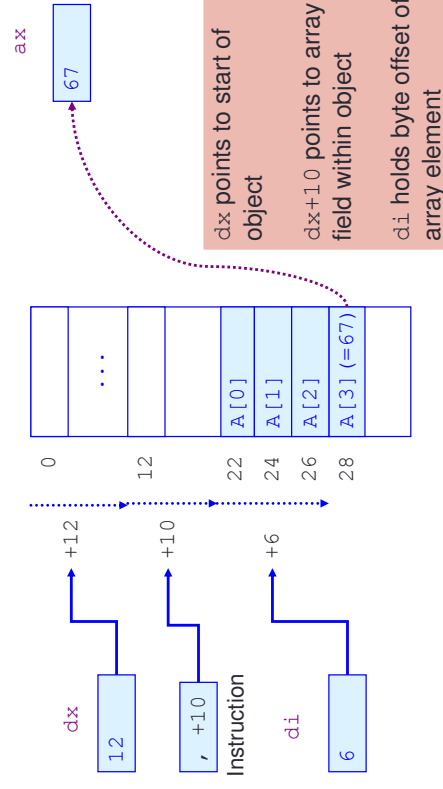
```

mov  eax, [ebx+4*ecx]
mov  [eax+2*ebx], ecx
mov  ax, [ebp+2*edi+age]
mov  [32+eax+2*ebx], dx

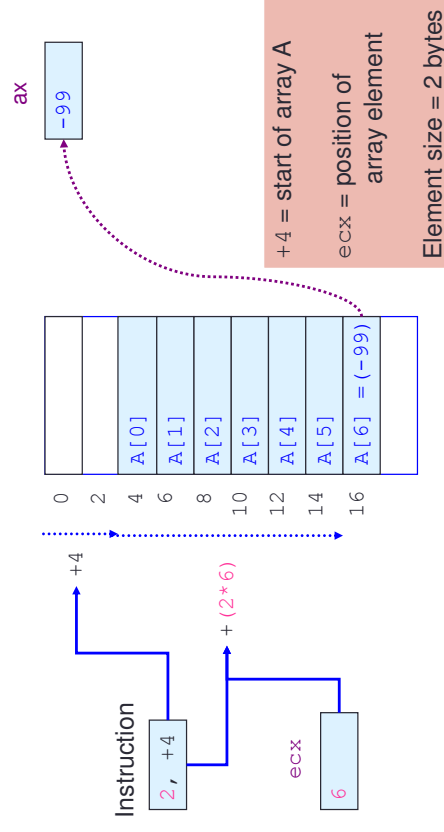
```

- Supports **efficient access to array elements within objects** and **on the stack** when the element size is 2, 4 or 8 bytes

Example 1: mov ax, [dx+di+10]



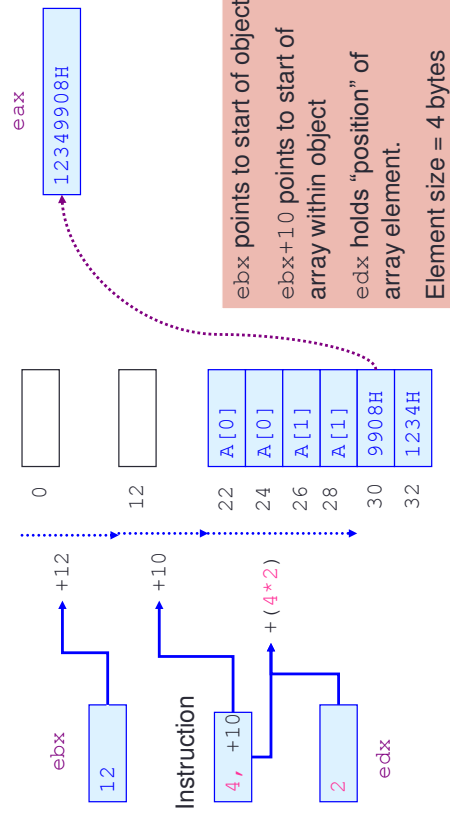
Example 1: mov ax, [2*ecx+4]



Books

- **Guide to assembly language programming in Linux**
Silvarama Dandamudi, Springer 2005
Good introduction to Linux assembly programming
- **Computer systems: a programmer's perspective**
Randal E. Bryant and David O'Hallaron, Prentice-Hall 2003
Aimed at Linux/BSD. Uses GNU assembler (gas) and C.
- **Intel IA-32 manuals**
<http://developer.intel.com/products/processor/manuals/index.htm>

Example 1: `mov eax, [ebx+4*edx+10]`



Internet resources

- **PC assembly language**
Paul Carter
<http://www.drpaulcarter.com/pcasm>
- **The art of assembly language programming**
Randall Hyde
<http://webster.cs.ucr.edu/AoA/index.html>