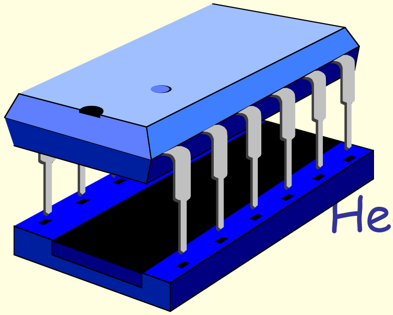
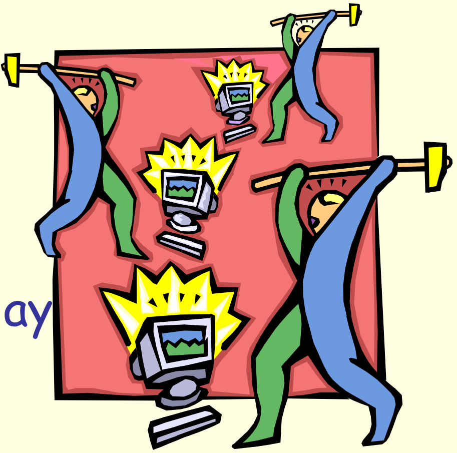
Pentium Architecture:

Registers & Addressing Modes

Professor Kin K. Leung
kin.leung@imperial.ac.uk
www.commsp.ee.ic.ac.uk/~kkleung/



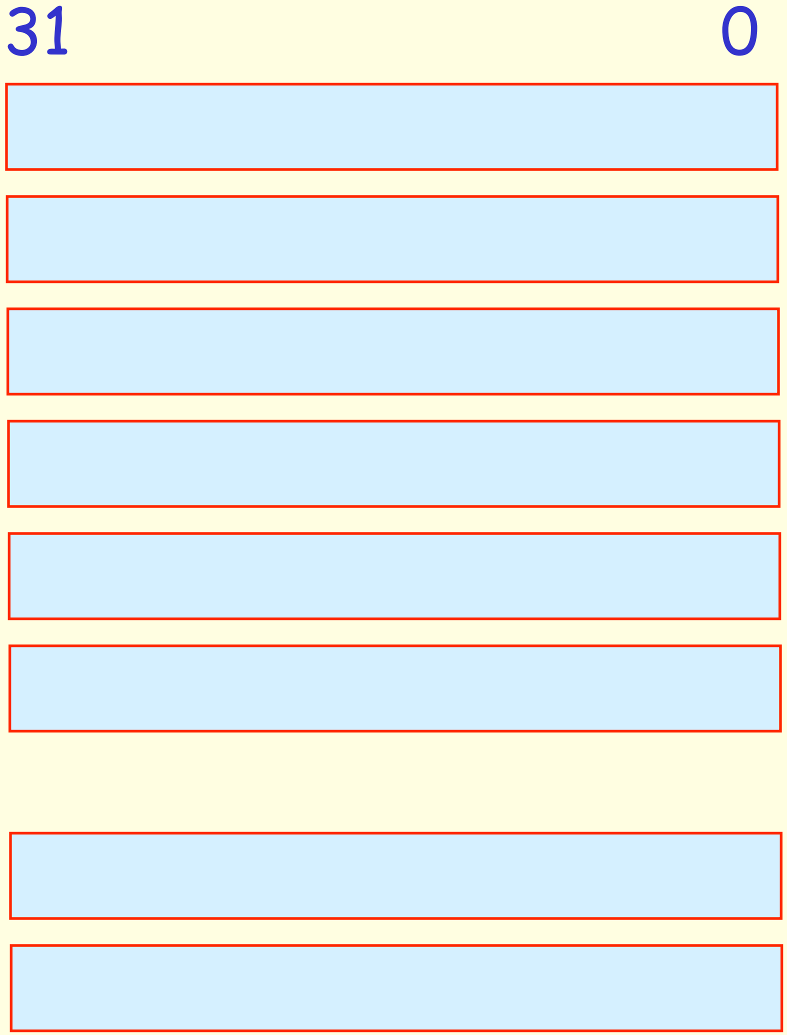
Heavily based on materials by Dr. Naranker Dulay



Intel Pentium Family

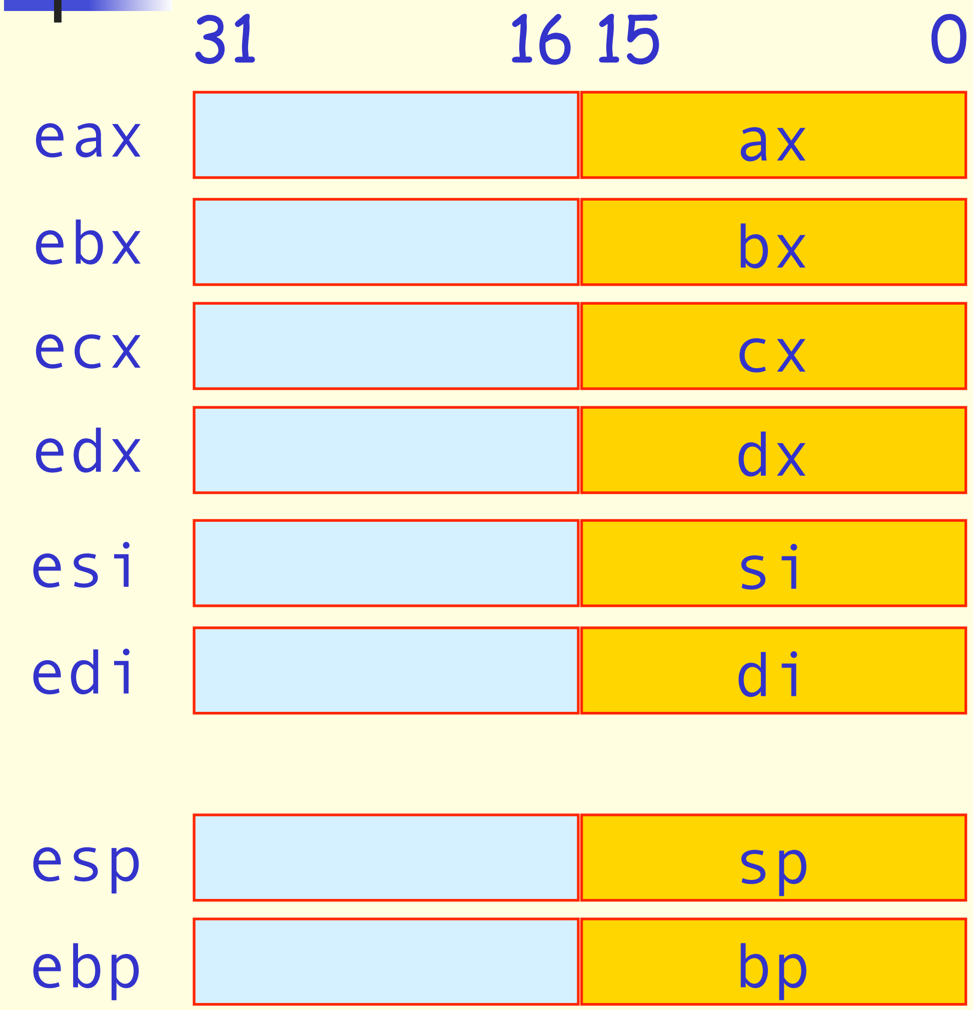
CPU	Year	Data Bus	Max. Mem.	Transistors	Clock MHz	Av. MIPS	Level-1 Caches
8086	1978	16	1MB	29K	5-10	0.8	
80286	1982	16	16MB	134K	8-12	2.7	
80386	1985	32	4GB	275K	16-33	6	
80486	1989	32	4GB	1.2M	25-100	20	8Kb
Pentium	1993	64	4GB	3.1M	60-233	100	8K Instr + 8K Data
Pentium Pro	1995	64	64GB	5.5M +15.5M	150-200	440	8K + 8K + Level2
Pentium II	1997	64	64GB	7M	266-450	466-	16K+16K + L2
Pentium III	1999	64	64GB	8.2M	500-1000	1000-	16K+16K + L2
Pentium 4	2001	64	64GB	42M	1300-2000		8K + L2

Registers (32-bit)



eax	'A' register
ebx	'B' register
ecx	'C' register
edx	'D' register
esi	source index register
edi	destination index register
esp	**stack pointer** Register
ebp	**base pointer** Register

Registers (16-bit)

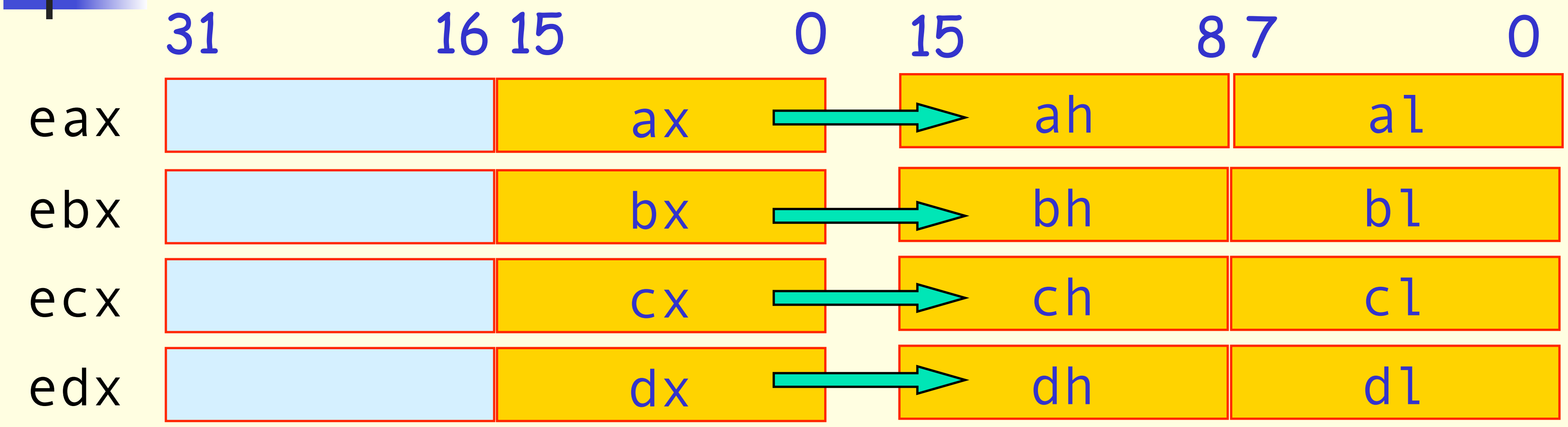


	31 … 16	15 … 0
eax		ax
ebx		bx
ecx		cx
edx		dx
esi		si
edi		di
esp		sp
ebp		bp

The least significant 16-bits of these registers have an additional **register name** that can be used for accessing **just** those 16-bits.

Note: There are no register names for the most significant 16-bits

Registers (8-bit)



The 2 least significant bytes of registers eax, ebx, ecx and edx also have register names, that can be used for accessing those bytes.

Note: There are **no** register names for accessing the 2 least significant bytes of esi, edi, esp, ebp.

Instruction Pointer Register

32-bit eip

- The **instruction pointer register** eip holds the address of the next instruction to be executed. The eip register corresponds to the program counter register in other architectures.
- eip is not normally manipulated explicitly by programs. However it is updated by special control-flow CPU instructions (e.g. **call**, **jmp**, **ret**) that are used to implement if's, while's, method calls etc.

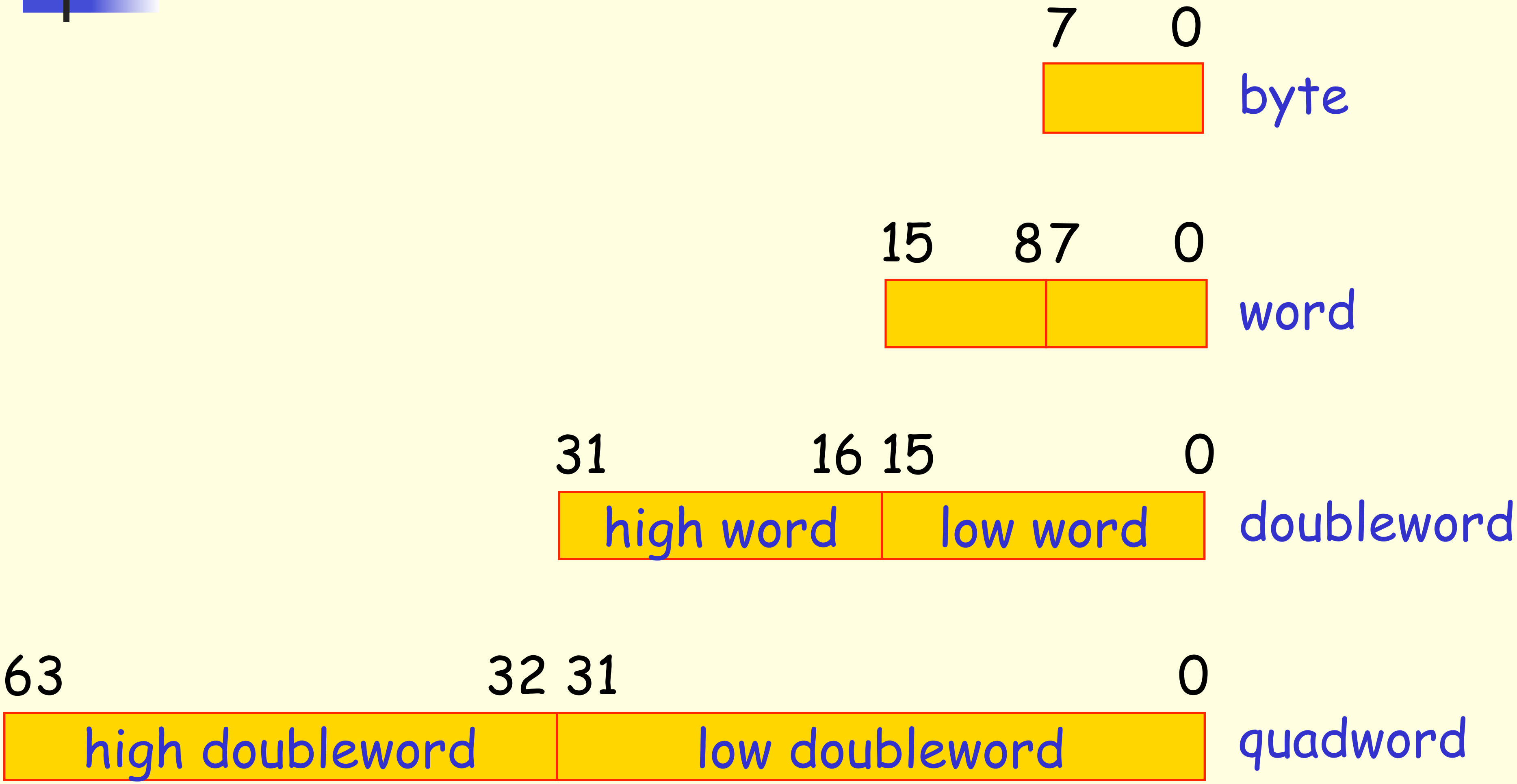
Flags Register

32-bit `eflags`

- The `eflags` register holds information about the current state of the CPU. Its 32-bits are mostly of interest to the Operating System; however, some of its bits are set/cleared after arithmetic instructions are executed, and these bits are used by conditional branch instructions:

Zero Flag (Bit 6)	Set (=1) if the result is zero, cleared (=0) otherwise
Sign Flag (Bit 7)	Set to MS-bit of result, which is the sign bit of a signed integer
Overflow Flag (Bit 11)	Set if result is too large a positive number or too small a negative number, cleared otherwise.
Carry Flag (Bit 0)	Set if carry or borrow out of MS-bit, cleared otherwise. Used in multi-precision arithmetic
Parity Flag (Bit 2)	Set if LS-byte of result contains an even number of bits cleared otherwise

Basic data types



Main Memory

Byte Addressable, Little Endian, Non-Aligned Accesses Allowed

Address	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 0BH ?
- Word at address 1H ?
- Word at address 2H ?
- Word at address 6H ?
- Doubleword at address 0AH ?
- Quadword at address 6H ?

Instruction Format

Most Pentium instructions have either 2, 1 or 0 operands and take one of the forms:

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

`label` is an optional user-defined identifier that will have a value that is the address of the instruction or data item that follows.

We'll use the netwide assembler (`nasm`) which follows Intel syntax. Beware: some Linux/BSD Pentium assemblers follow a different syntax.

Directives for "Global" Variables (1)

- Data declaration directives are *special assembler* commands that allow "**global**" data (variables) to be declared. Global data is mapped to fixed memory locations and can be accessed by using the name of the variable. The address of the global variable is encoded into Pentium instructions as required.

```
Initialised data directives db (byte), dw (word), dd (doubleword)

  users    db   3           ; byte with value 3
  age      dw   21          ; word with value 21
  total    dd   999         ; doubleworld 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
```

Directives for “Global” Variables (2)

- Uninitialised data can be reserved with `resb (byte), resw (word), resd (doubleword)`

```
tiny    resb  10      ; reserve 10 bytes

little  resw  100     ; reserve 100 words (200 bytes)

big     resd  1000    ; reserve 1000 doubleworlds (4000 bytes)
```

equ directive for constants

- We can define named constants with an `equ` directive, e.g.:

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

Operands (Addressing Modes)

- Register Operands

  e.g. `eax, dx, al, si, bp`

- Immediate Operands (i.e. Constants)

  e.g. `23, 67H, 101010B, 'R', 'ON'`

- Memory Operands [*BaseReg* + *Scale***IndexReg* + *Displacement*]

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

Note: **The source and destination operands of a 2 operand Pentium instruction CANNOT both be memory operands.**

Examples

```
Label    Instruction            Comment

         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]
```

More Examples

```
Label     Instruction       Comment

          neg    ax         ; ax = -ax

          cmp    eax, ecx   ; compare operands and set
                            ; eflags register

          je     end        ; if flags.zf = 1 then
                            ;       eip = address end

          call   print      ; call method print

end:      ret               ; return from method
```

Register Operand

Register

- Operand found in the specified register

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

- Depending on the instruction and sometimes the processor model, a register operand can be in any of the general purpose registers. Some instructions will also accept the `eflags` and `eip` register.
- Some instructions such as `idiv` implicitly use operands contained in a pair of registers, e.g. in `ax` and `dx`.
- For most 2-operand instructions **destination & source operands must be of the same size**

Immediate (Constant) Operand

Constant

- Operand is an immediate (i.e. constant) value

```
mov   eax,  22
mov   ecx,  16h
mov   ax,   10110B
mov   ebx,  12345678H
mov   bx,   age
mov   eax,  total
mov   al,   ‘a’
mov   ax,   ‘mp’
```

- Immediate values are encoded directly into the instruction.
- *Are not normally applicable for destination operands.*
- If a data variable is used as an immediate operand then the address of the variable is used. *Microsoft's assembler (MASM) requires # before the variable or the keyword OFFSET.*

Memory Operands

Memory operands specify an *address* using expressions of the form:

```
[ Baseregister + Scale*Indexregister + Displacement ]
```

Base register:	eax, ebx, ecx, edx, esi, edi, ebp, esp
Index register:	eax, ebx, ecx, edx, esi, edi, ebp
Scale:	either 2 or 4 or 8
Displacement:	constant value

Expressions can be re-ordered, e.g. displacement can be written first. Omission of different parts of the expression give different modes.

Note: The size of an operand is normally inferred from the Instruction or register operand. In case of ambiguity we must explicitly prefix the operand with byte or word or dword to specify the size of the operand, e.g. byte [ebx], word 16, dword [ebx+2*edi+list]

Displacement (Direct Addressing)

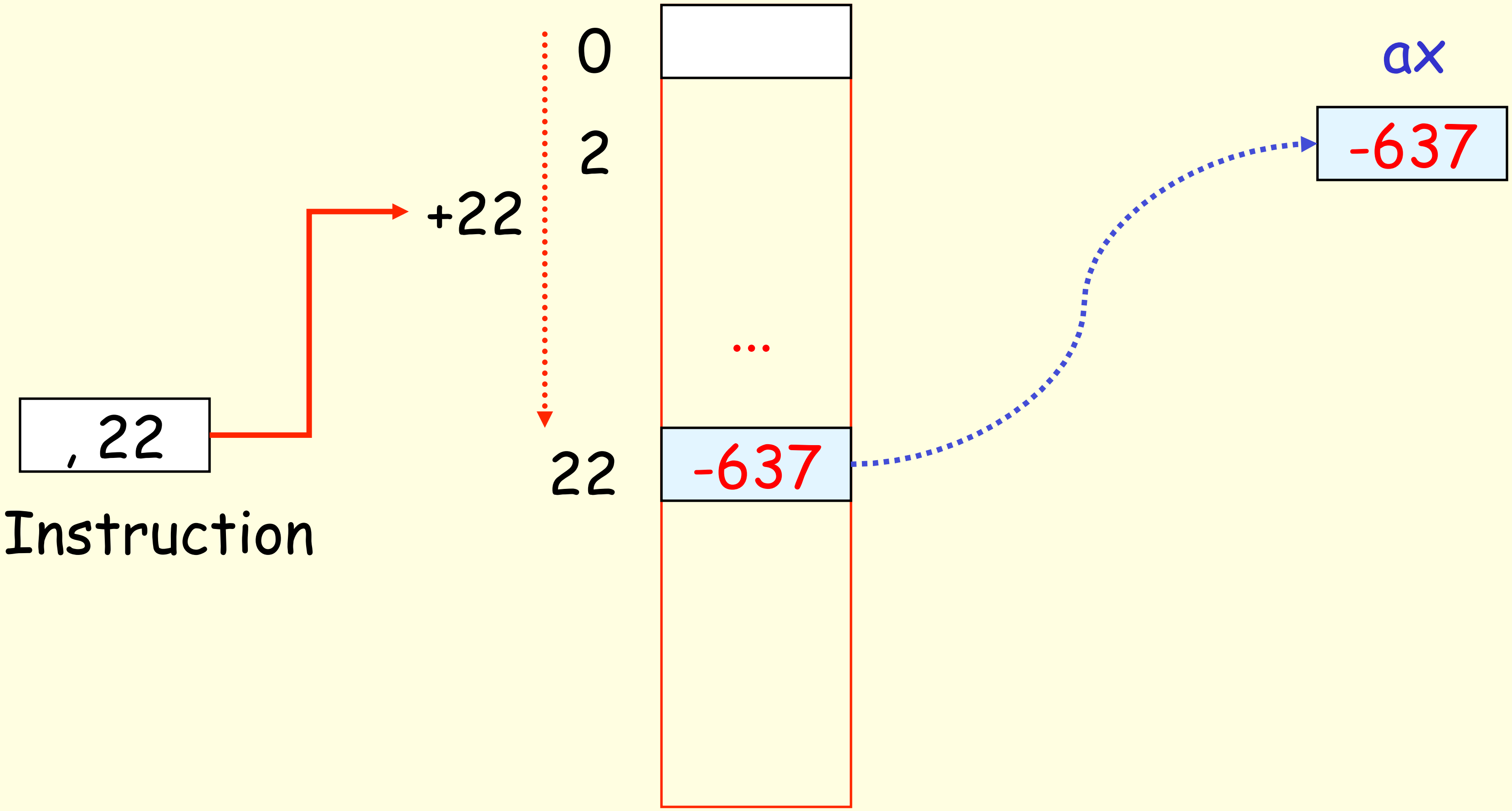
[Displacement]

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

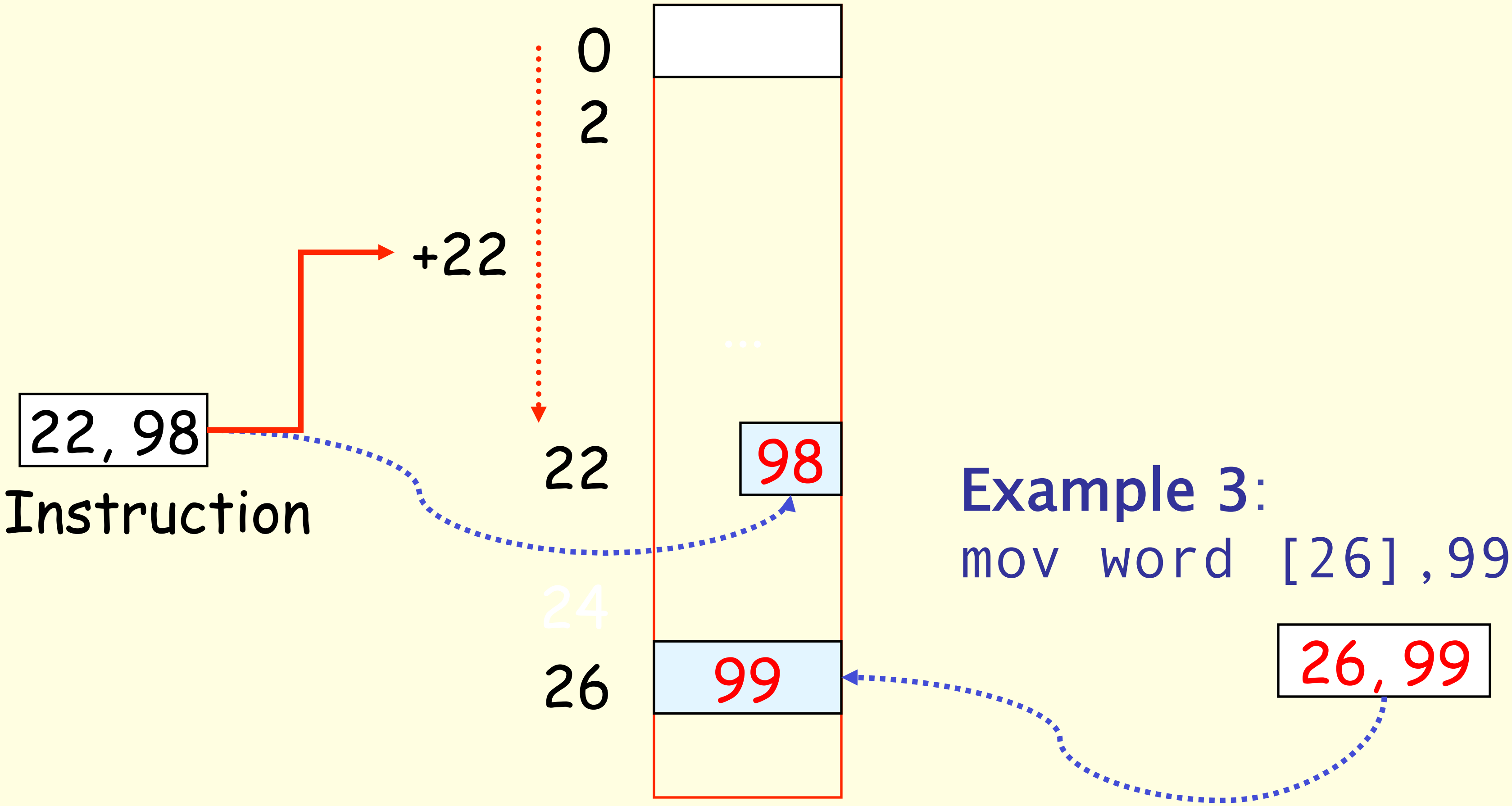
```
mov   eax, [22]
mov   [16H], esi
mov   byte [22], 98
mov   ebx, [12345678H]

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 allows displacement to be a constant expression, e.g. [list+22]

Example 1: mov ax,[22]



Example 2: mov byte [22],98



Example 3:

`mov word [26],99`

Base (Register Indirect)

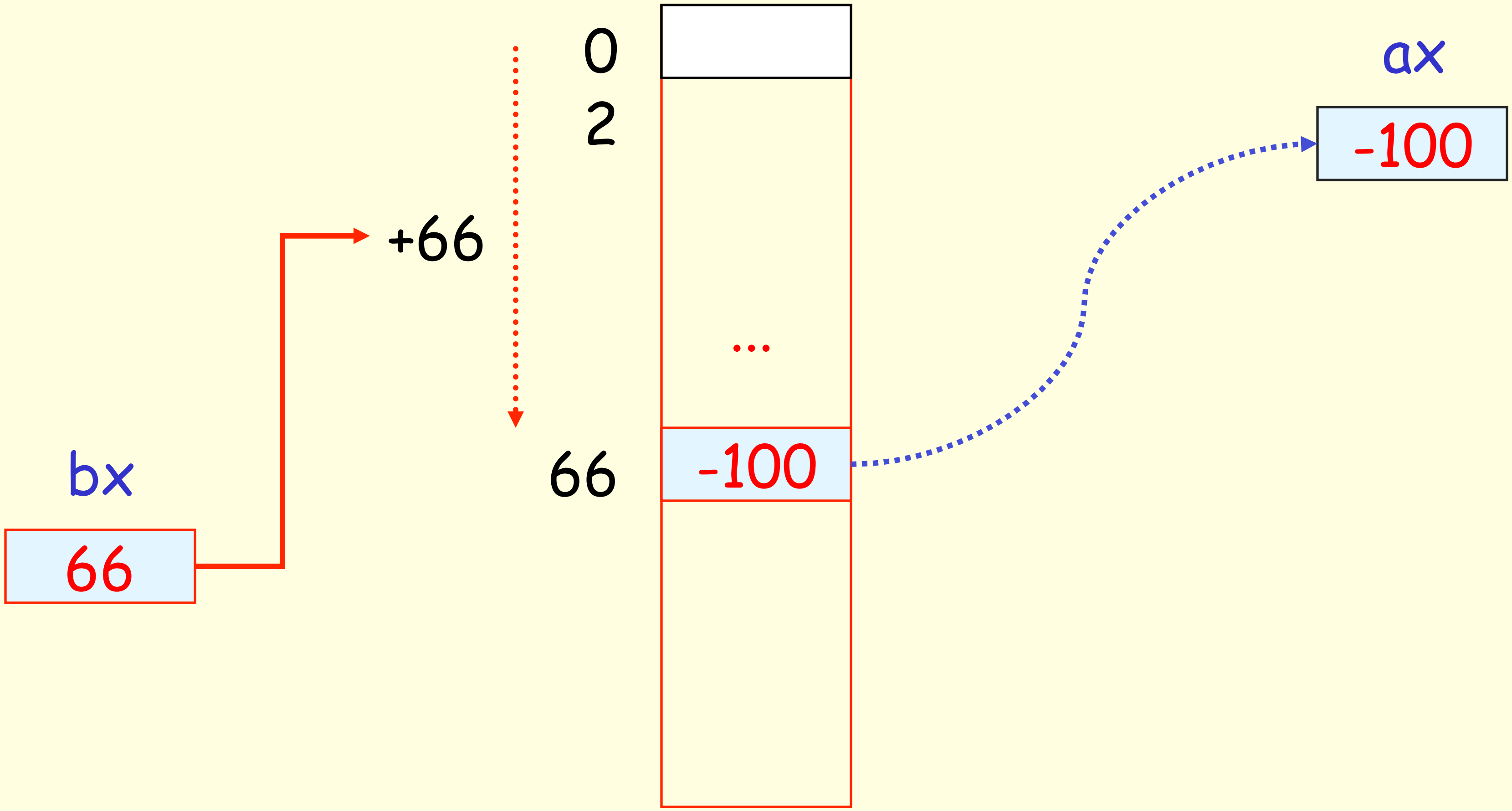
[Base]

- Contents of specified Base Register gives *address*

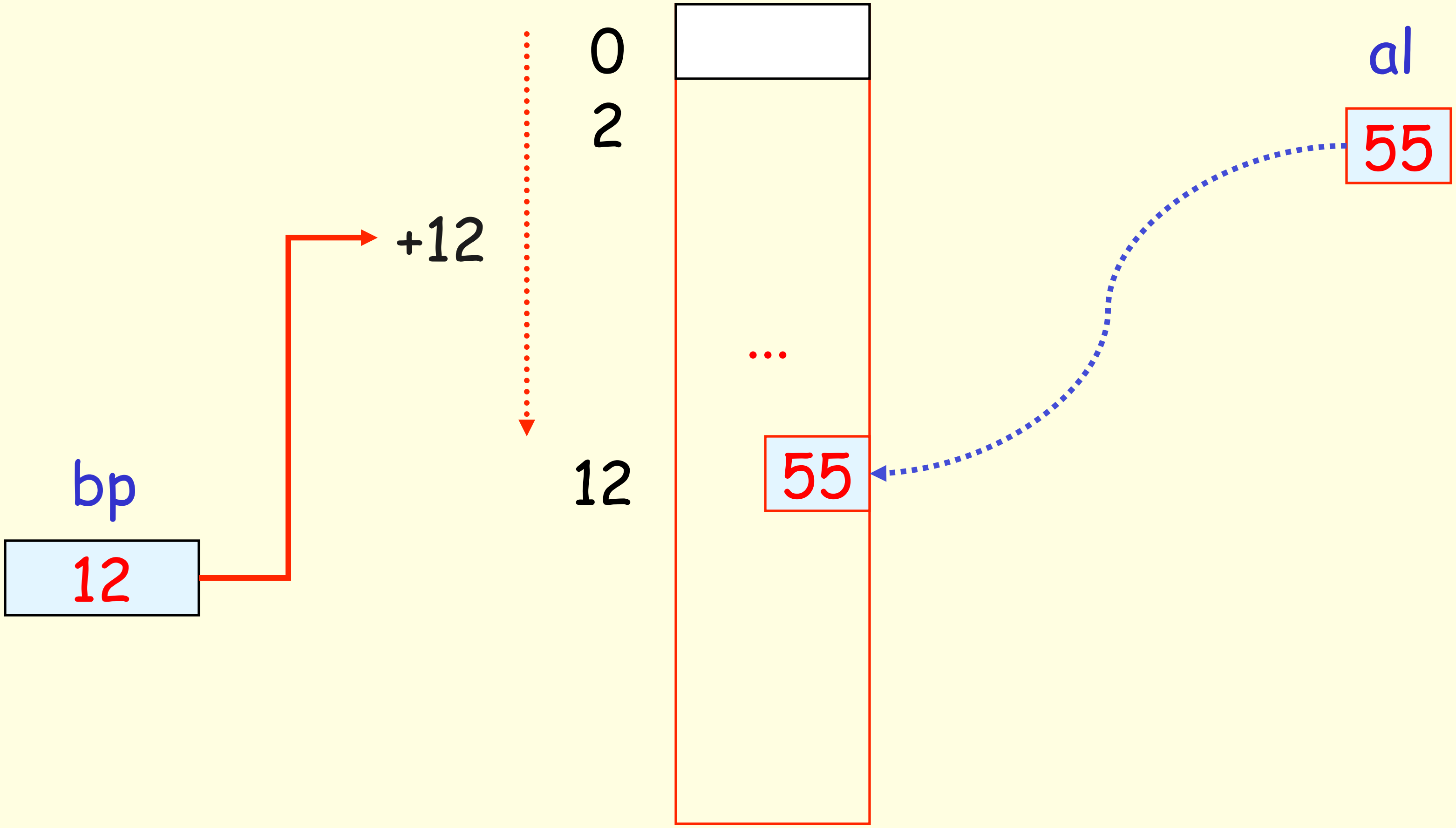
```
mov   ax,  [ebx]
mov   [ebp], al
mov   eax,  [edi]
mov   [esi], ah
mov   ebx,  [esi]
mov   [esp], ecx
```

- 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.

Example 1: mov ax,[bx]



Example 2: mov [bp],al



Base + Displacement (Register Relative)

[Base + Displacement] or [Displacement + Base]

- Sum of specified Base Register and Displacement gives *address offset*. Displacement can be negative.

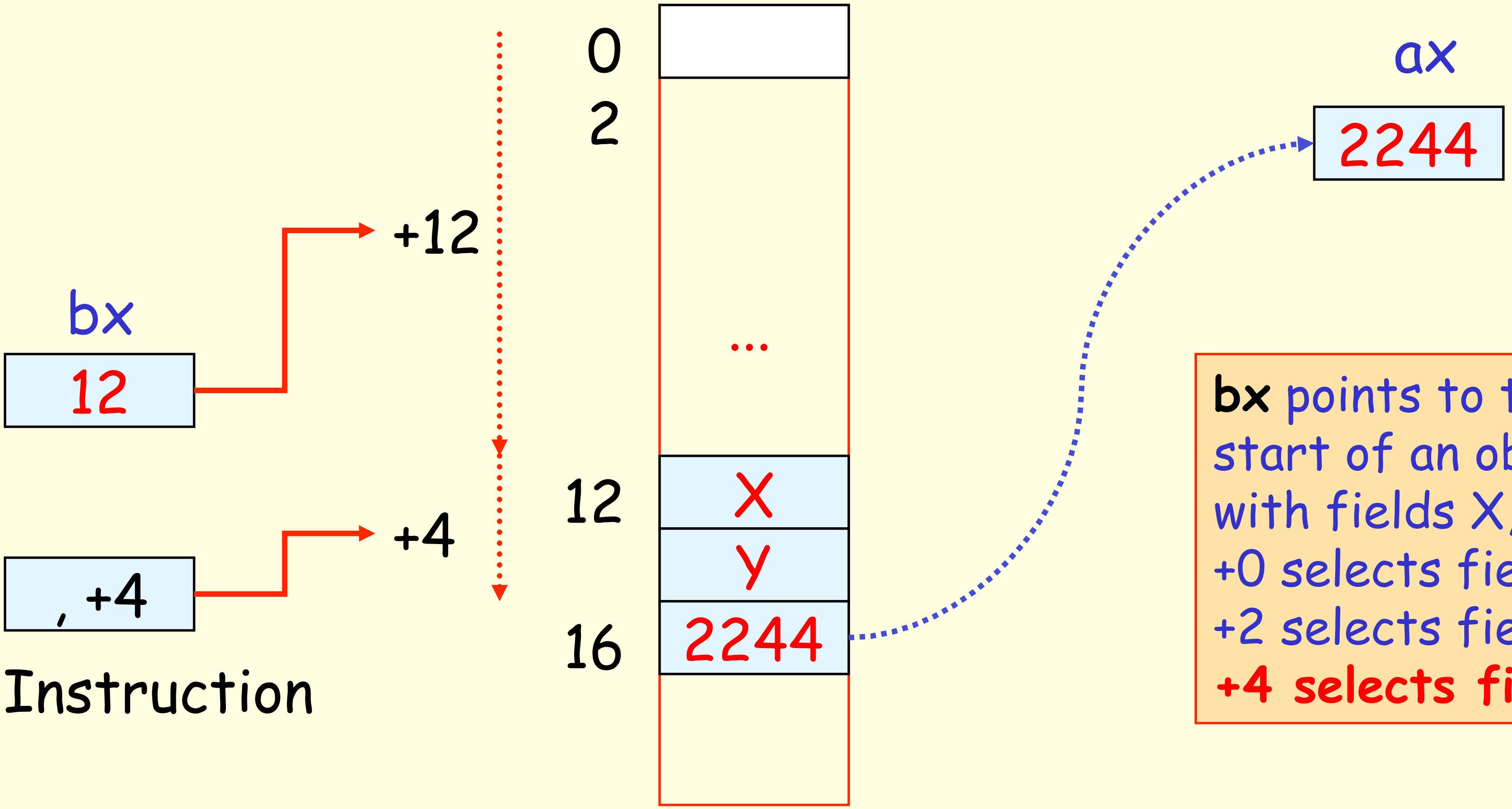
```
mov   ax, [ebx+4]
mov   [ebp+2], dh
mov   ax, [di-6]
mov   dl, [esi+age]
mov   [list+ebx], cx
mov   dx,[ebp+list-2]
```

Can be used to access **object fields**:
    Base Register = Start of Object,
    Displacement = Position of field
        within object.

Can be used to access **array elements**:
    Displacement = start of array,
    Base Register = position of array
        element

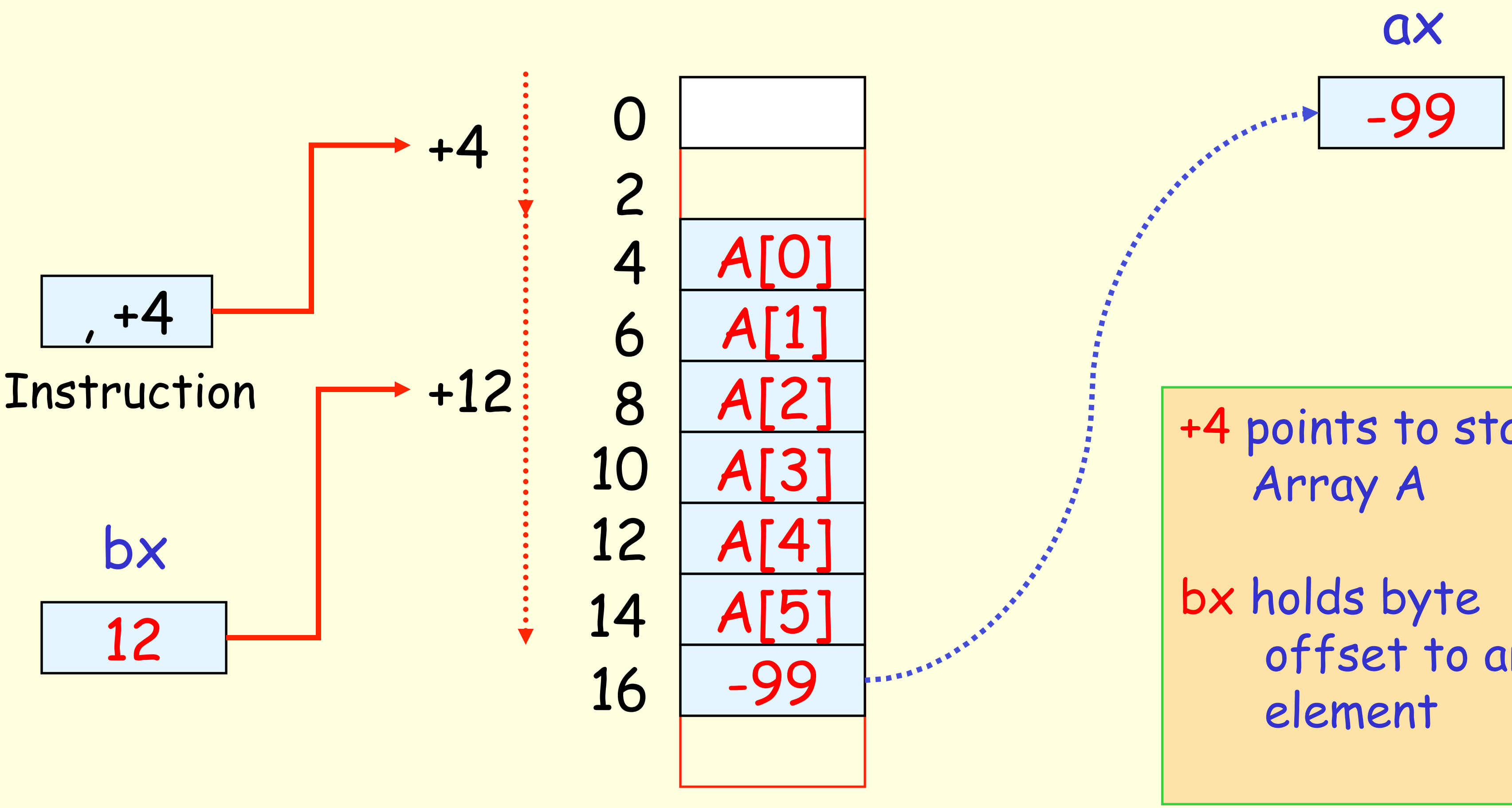
Can be used to access **parameters** &
    local variables (covered later)

Example 1: mov ax , [bx+4]



bx points to the start of an object with fields X, Y & Z
+0 selects field X
+2 selects field Y
+4 selects field Z

Example 2: mov ax, [bx+4]



+4 points to start of Array A

bx holds byte offset to array element

Base + Index (Based Indexed)

[Base + Index]

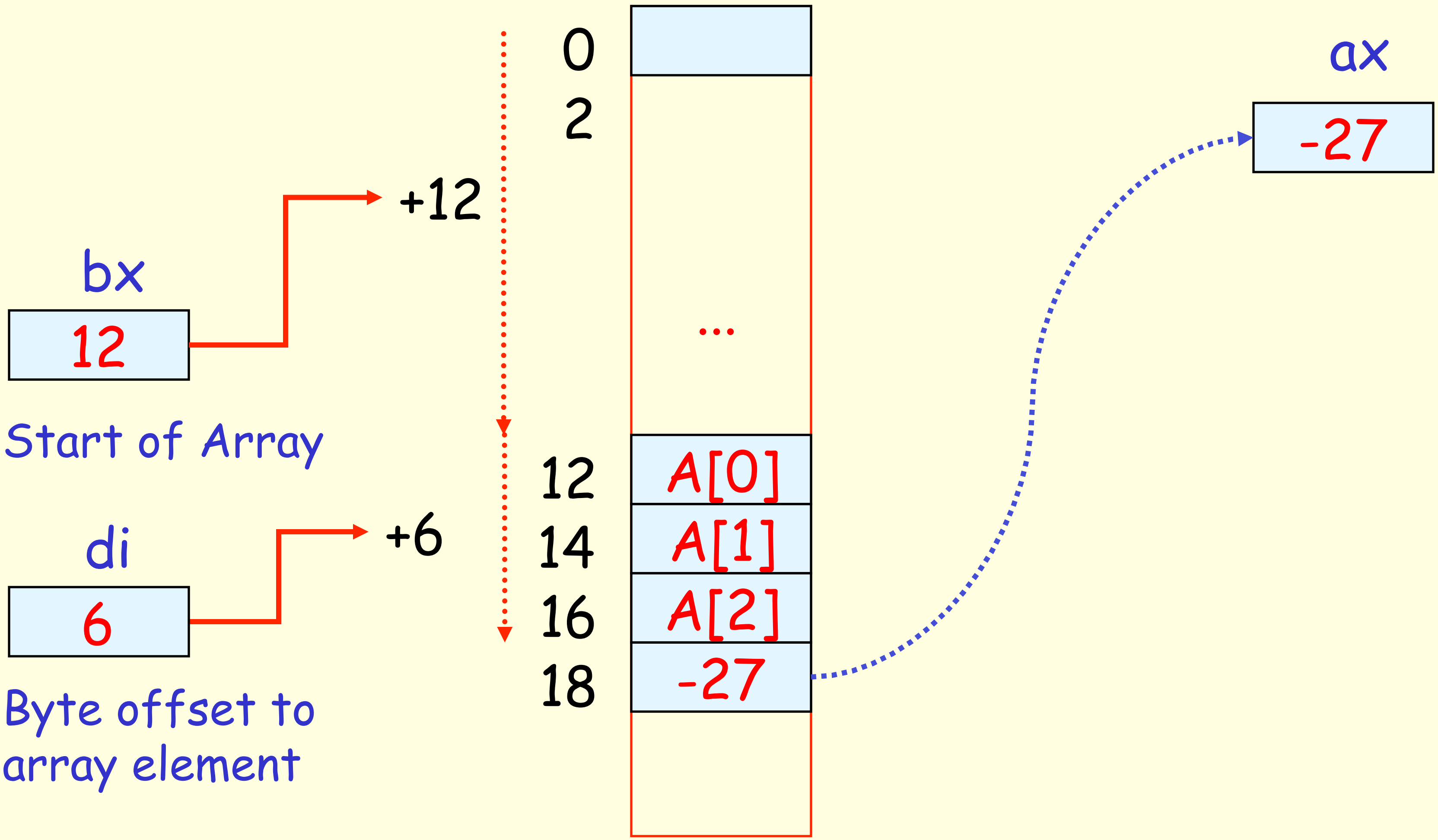
- Sum of specified Base Register and Index Register gives *address*

```
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 = position of element.

Example: mov ax,[bx+di]



Base+Index+Displacement (Based Relative Index)

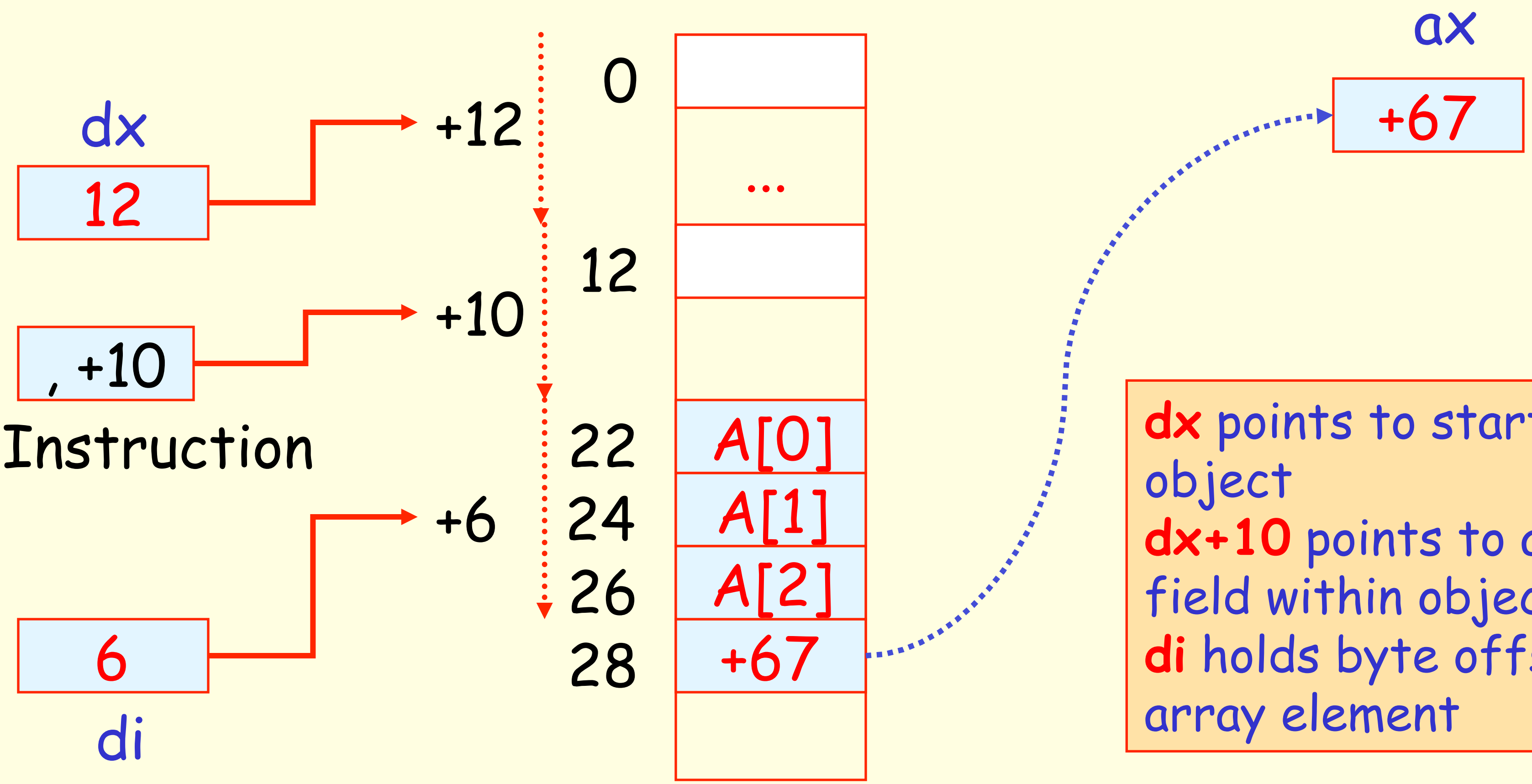
[Base + Index + Displacement] or [Displacement + Base + Index]

- Sum of specified Base Register and Index Register and Displacement gives 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: mov ax,[dx+di+10]



dx points to start of object
dx+10 points to array field within object
di holds byte offset of array element

(Scale*Index) + Displacement (Scaled Index)

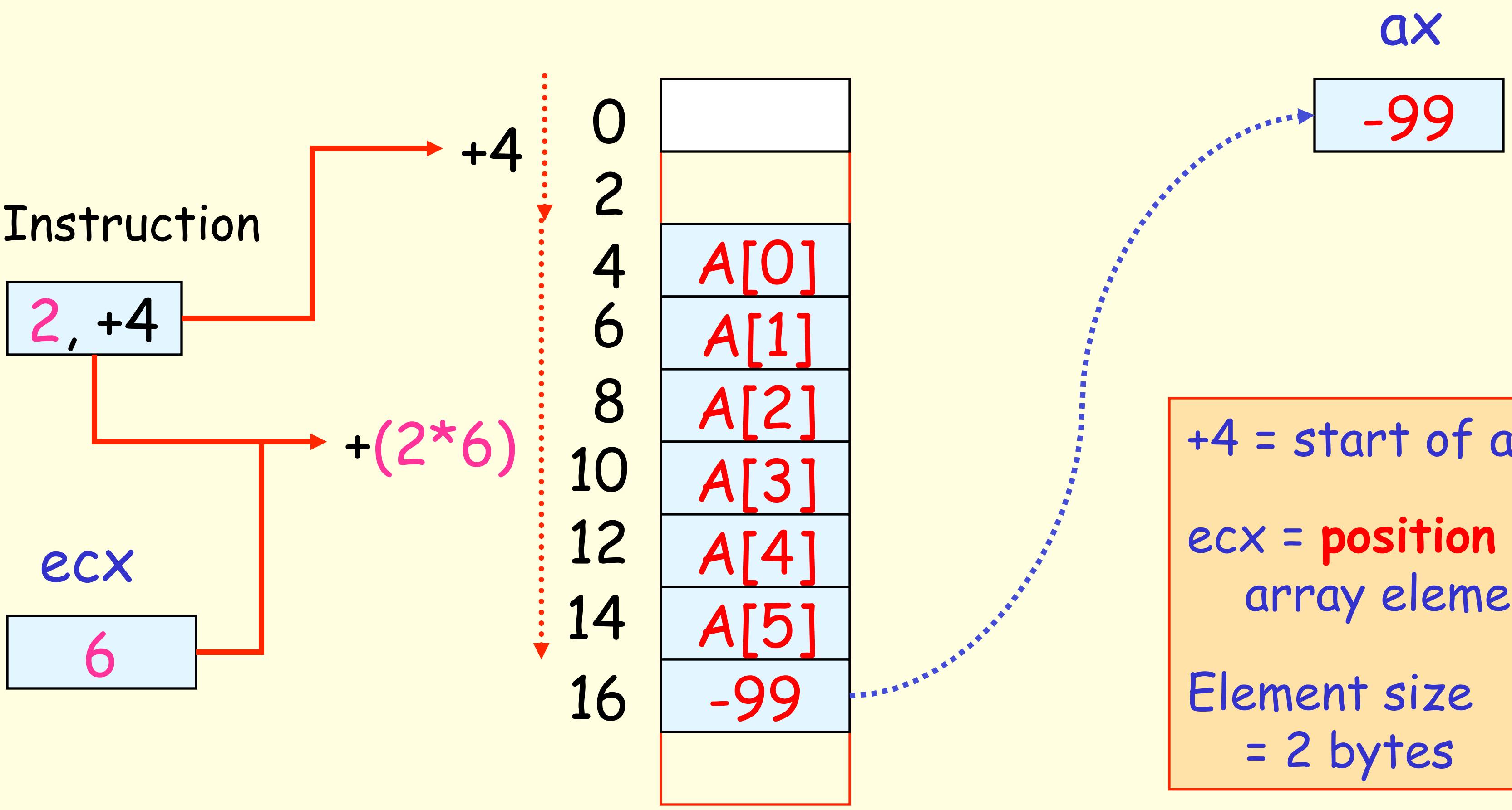
[Scale * Index + Displacement] or [Displacement + Scale * Index]

- Product of Index Register and a constant scaling factor (2, 4 or 8) added to specified Displacement to give *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 (but only 2, 4 or 8)

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



+4 = start of a Array

ecx = **position** of
array element

Element size
= 2 bytes

Base + (Scale * Index) + Displacement

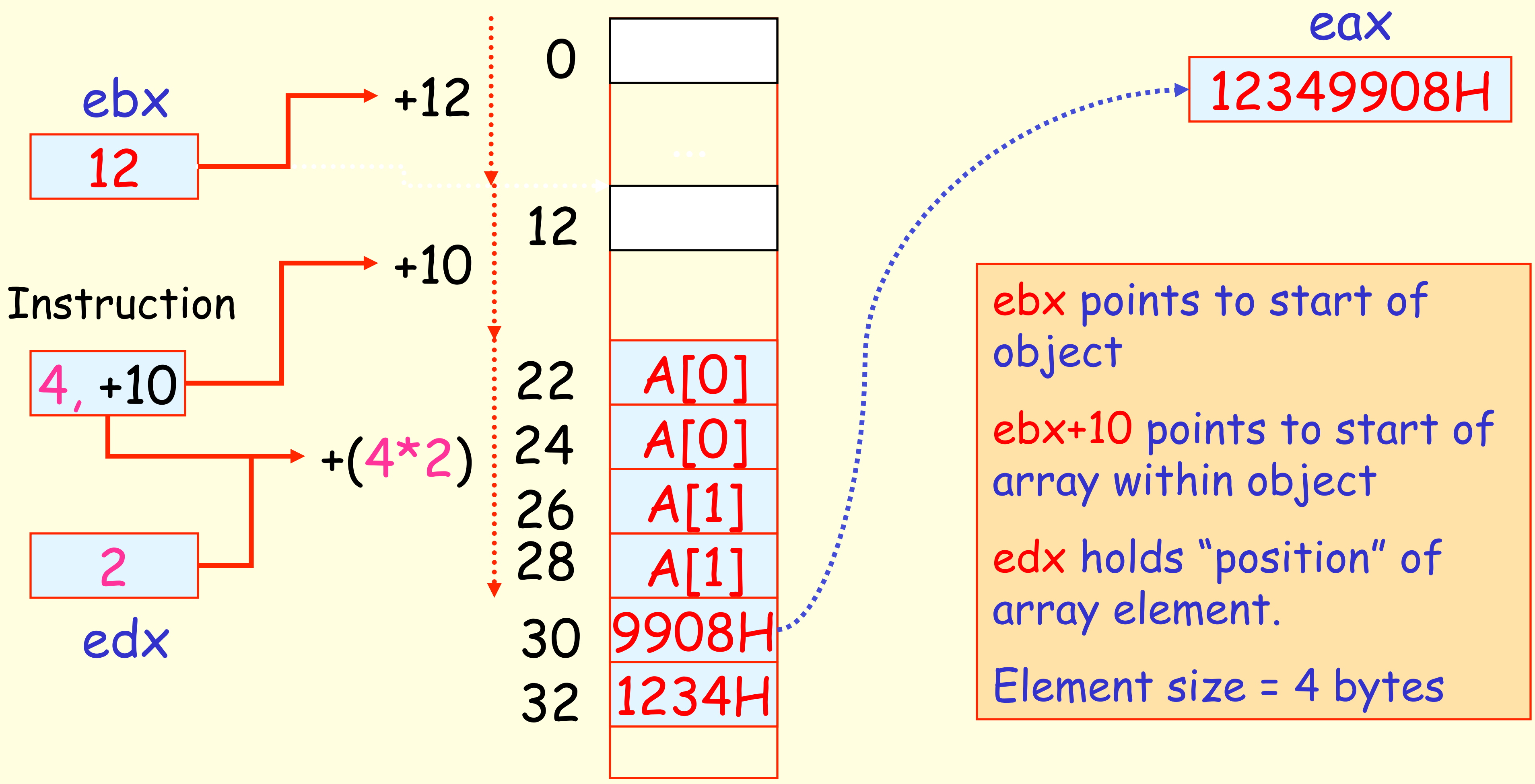
[Base + Scale * Index + Displacement] or
[Displacement + Base + Scale * Index]

- Product of Index Register and a constant scaling factor (2, 4 or 8) added to specified Base Register and Displacement to give *address* offset.

```
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 arrays within objects and on the stack when the element size is 2, 4 or 8 bytes.

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



ebx points to start of object

ebx+10 points to start of array within object

edx holds "position" of array element.

Element size = 4 bytes

Linux oriented books

Guide to Assembly Language Programming in Linux

- **Sivarama Dandamudi**, Springer, 2005. Good introduction to Linux assembly programming.

Computer Systems: A Programmer's Perspective

- **Randal E. Bryant & David O'Hallaron**, Prentice-Hall, 2003.
  Excellent book.
  Geared for Linux/BSD. Uses GNU assembler (gas) and C.

Intel Pentium 4 Manuals

- http://developer.intel.com/design/Pentium4/documentation.htm

Internet Resources

PC Assembly Language

- **Paul Carter**
  Download from www.drpaulcarter.com/pcasm

The Art of Assembly Language Programming

- **Randall Hyde**
  Over 1200 pages!! Download for personal use via:
  http://webster.cs.ucr.edu/AoA/index.html