Assembly Programing with NASM *(and GAS)*

DR. ARKA PROKASH MAZUMDAR

NASM Data Directives

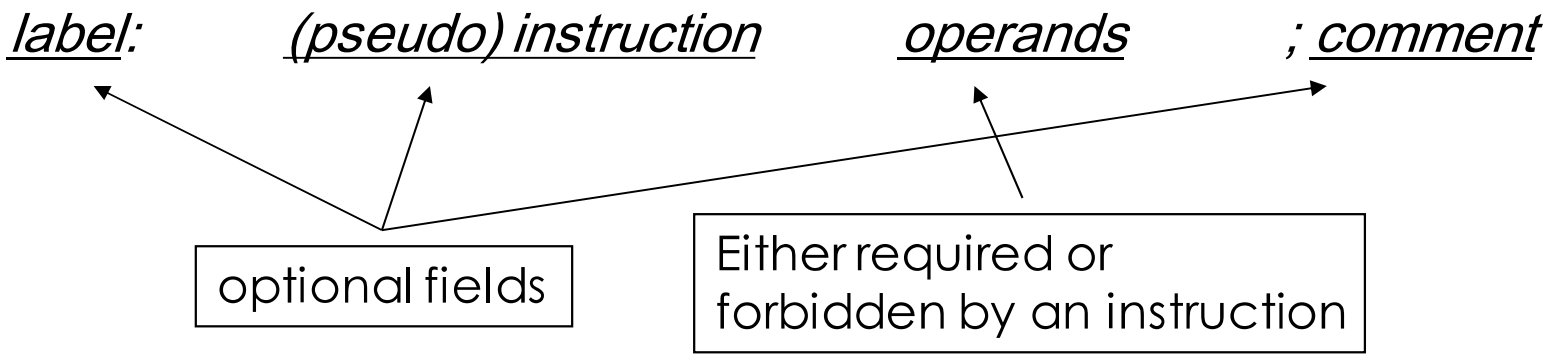
```
L1      db          0           ; byte
L2      dw          1000        ; word
L3      db          110101b     ; byte
L4      db          12h         ; byte
L5      db          17o         ; byte
L6      dd          1A92h       ; double word
L7      resb        1           ; uninitialized byte
L8      db          'A'         ; ascii code = 'A'
L9      db          0,1,2,3     ; 4 bytes
L10     db          'w','o','r','d',0   ; string
L11     db          'word', 0
L12     times 100 db 0          ; 100 bytes of zero
L13     resw        100         ; 100*2(word bytes)
```

Data Types in IA32 (AT&T format)

C declaration	Intel data type	Assembly code suffix	Size (by
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`long long int`	—	—	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/1

Instruction Basics

Each NASM standard source line contains a combination of the 4 fields



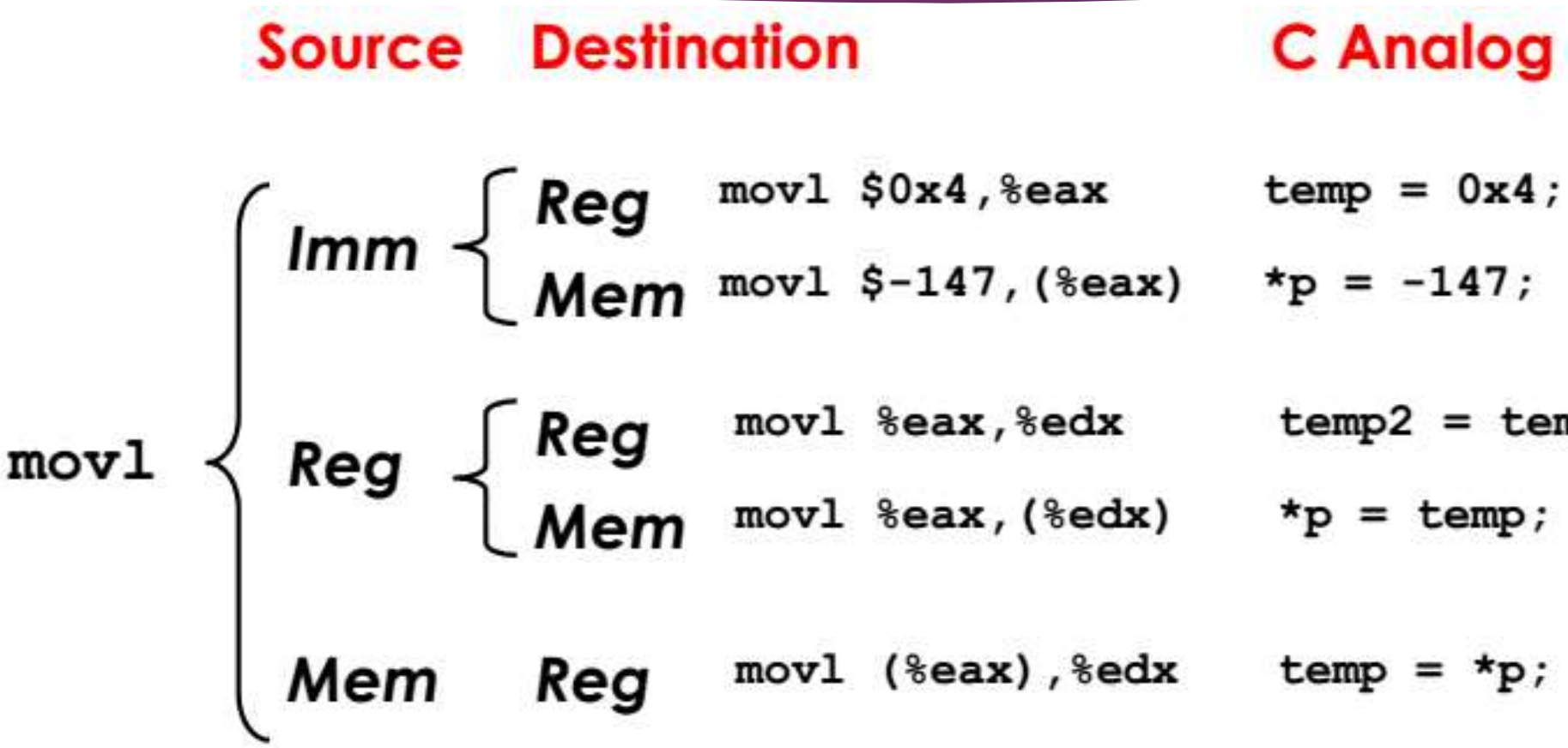
1. **Backslash (\)** uses as the line continuation character: if a line ends with backslash, the next line is considered to be a part of the backslash-ended line.
2. **No restrictions on white space** within a line.
3. A colon after a label is optional.

Example Program

```
1   int simple(int *xp, int y)
2   {
3     int t = *xp + y;
4     *xp = t;
5     return t;
6   }
```

```
1   simple:
2     pushl   %ebp              Save frame
3     movl    %esp, %ebp        Create new
4     movl    8(%ebp), %edx     Retrieve x
5     movl    12(%ebp), %eax    Retrieve y
6     addl    (%edx), %eax      Add *xp to
7     movl    %eax, (%edx)      Store t at
8     popl    %ebp              Restore fr
9     ret                       Return
```

Operand Modes



Examples

Data directives (different to MASM)

- **Mov al, [L1]** ; copy byte at L1
- **Mov eax, L1** ; eax = address of byte at L1
- **Mov [L1], ah** ; copy ah into byte at L1
- **Mov eax, [L6]** ; copy double word
- **Add eax, [L6]** ; eax = eax + double word at L6
- **Add [L6], eax** ; double word at L6 += eax
- **Mov al, [L6]** ; copy first byte of double word at L6 into al
- **Mov [L6], 1** ; operation size is not specified
- **Mov dword [L6], 1** ; store a 1 at L6

Instructions: Setup

- STACK
  - can be used as a convenient place to store data temporarily
  - Also used for making subprogram calls, passing parameters and loca
- Data can only be added in double word units
- PUSH
  - inserts a double word on the stack by **subtracting 4 from ESP**
  - And then stores the double word at [ESP]
- POP
  - reads the double word at [ESP]
  - And then adds 4 to ESP

Instructions: Setup

- Call subprogram
- CALL
  - Make an unconditional jump to a subprogram
  - And pushes the address of the next instruction on the stack
- RET
  - Pops off an address
  - And jumps to that address.
- When using this inst.: It is very important that one manage stack correctly so that **the right number is popped off by t**

Example

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

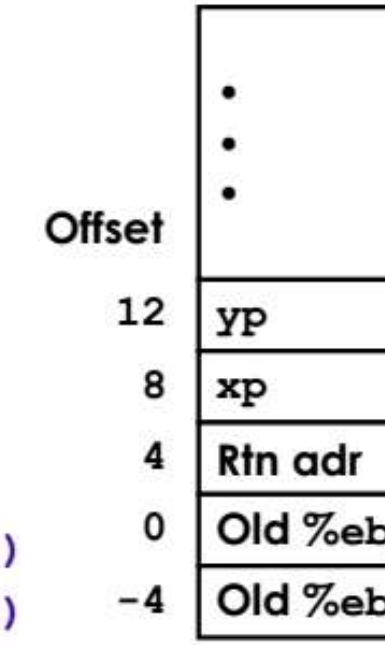
Finish

Example (contd.)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

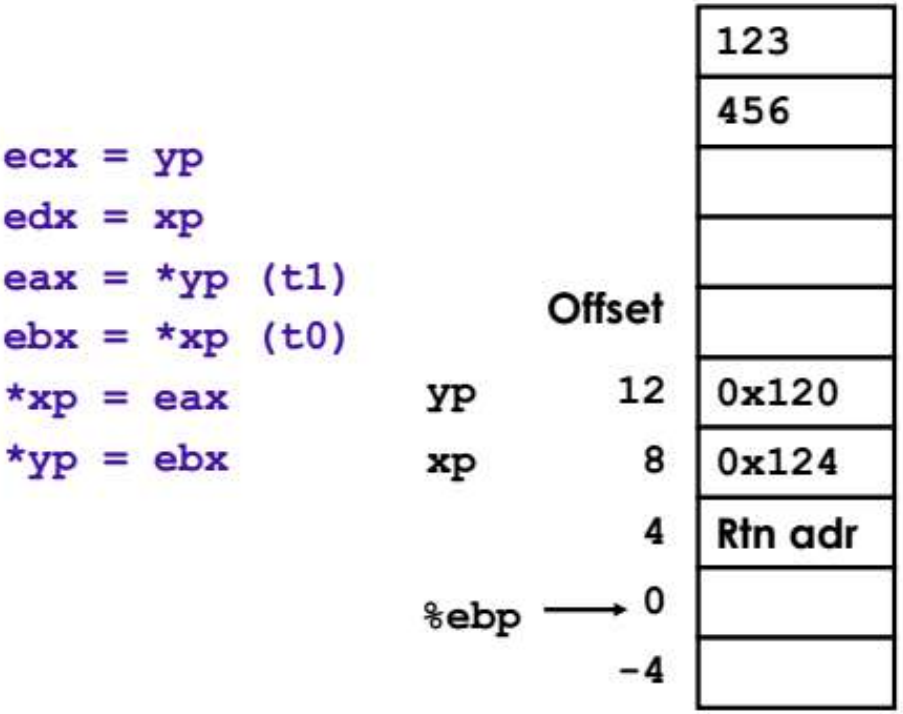
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

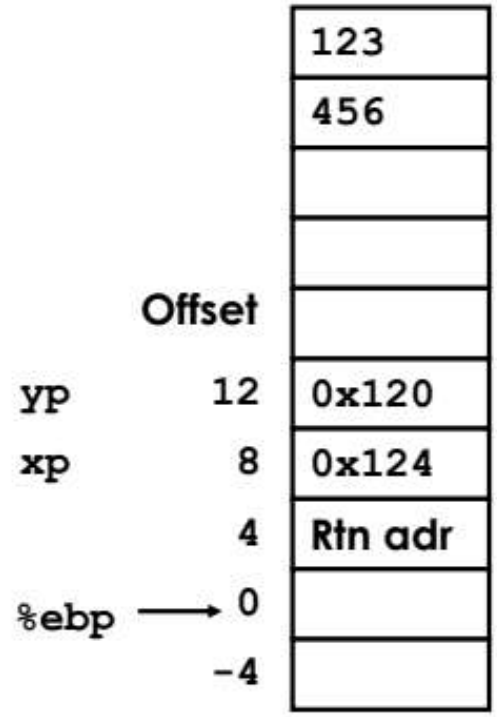
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

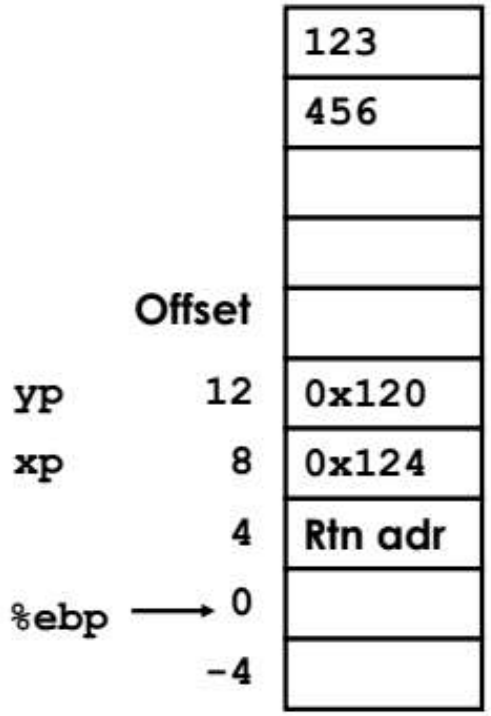
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

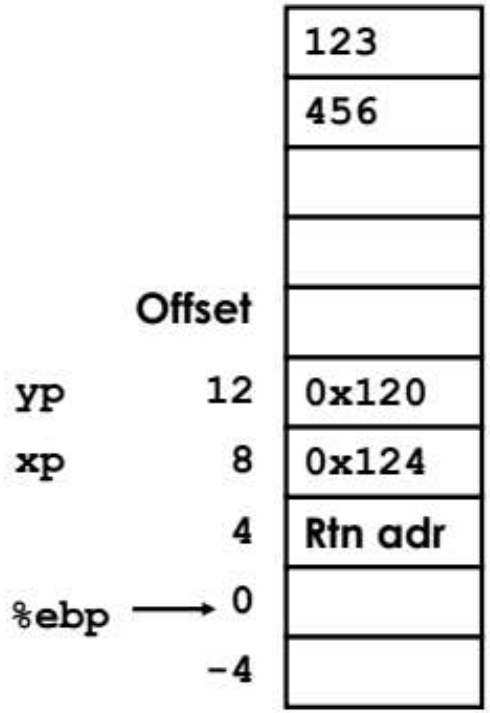
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

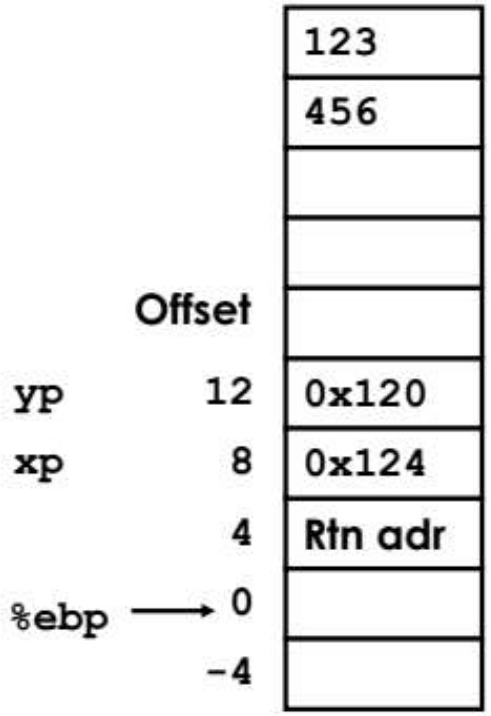
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

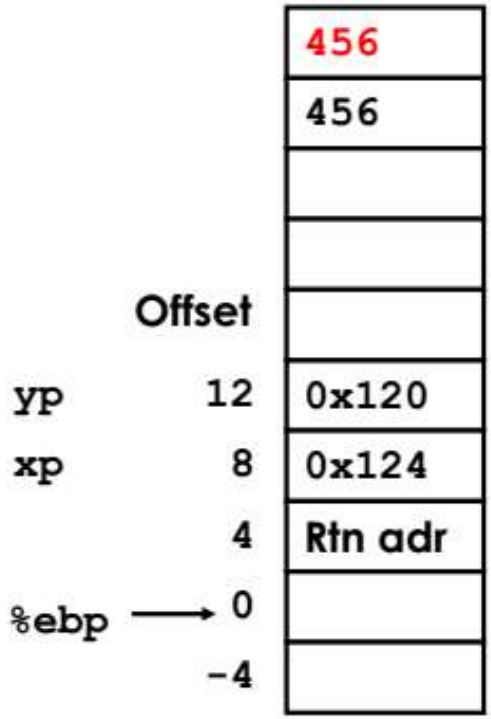
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

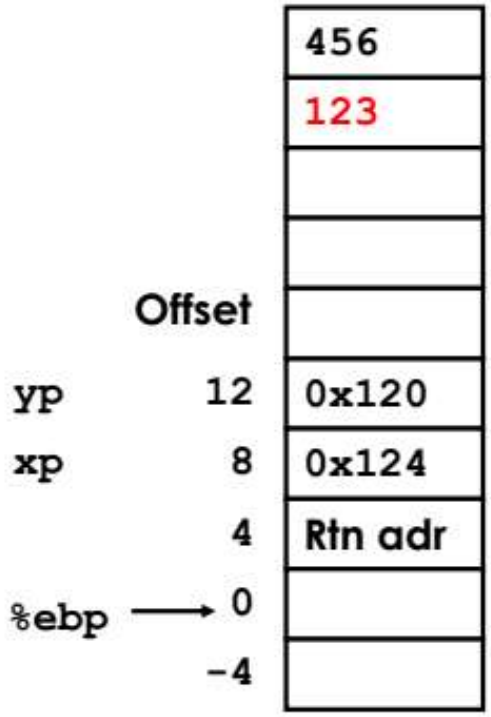
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Some Integer Instructions

Format	Computation	
Two-Operand Instructions		
`addl` *Src,Dest*	*Dest* = *Dest* + *Src*	
`subl` *Src,Dest*	*Dest* = *Dest* – *Src*	
`imull` *Src,Dest*	*Dest* = *Dest* * *Src*	
`sall` *k,Dest*	*Dest* = *Dest* << *k*	Also called `shll`
`sarl` *k,Dest*	*Dest* = *Dest* >> *k*	Arithmetic
`shrl` *k,Dest*	*Dest* = *Dest* >> *k*	Logical

k is an immediate value or contents of %cl

Some Integer Instructions

Format	Computation
Two-Operand Instructions	
`xorl` *Src,Dest*	*Dest* = *Dest* ^ *Src*
`andl` *Src,Dest*	*Dest* = *Dest* & *Src*
`orl` *Src,Dest*	*Dest* = *Dest* | *Src*

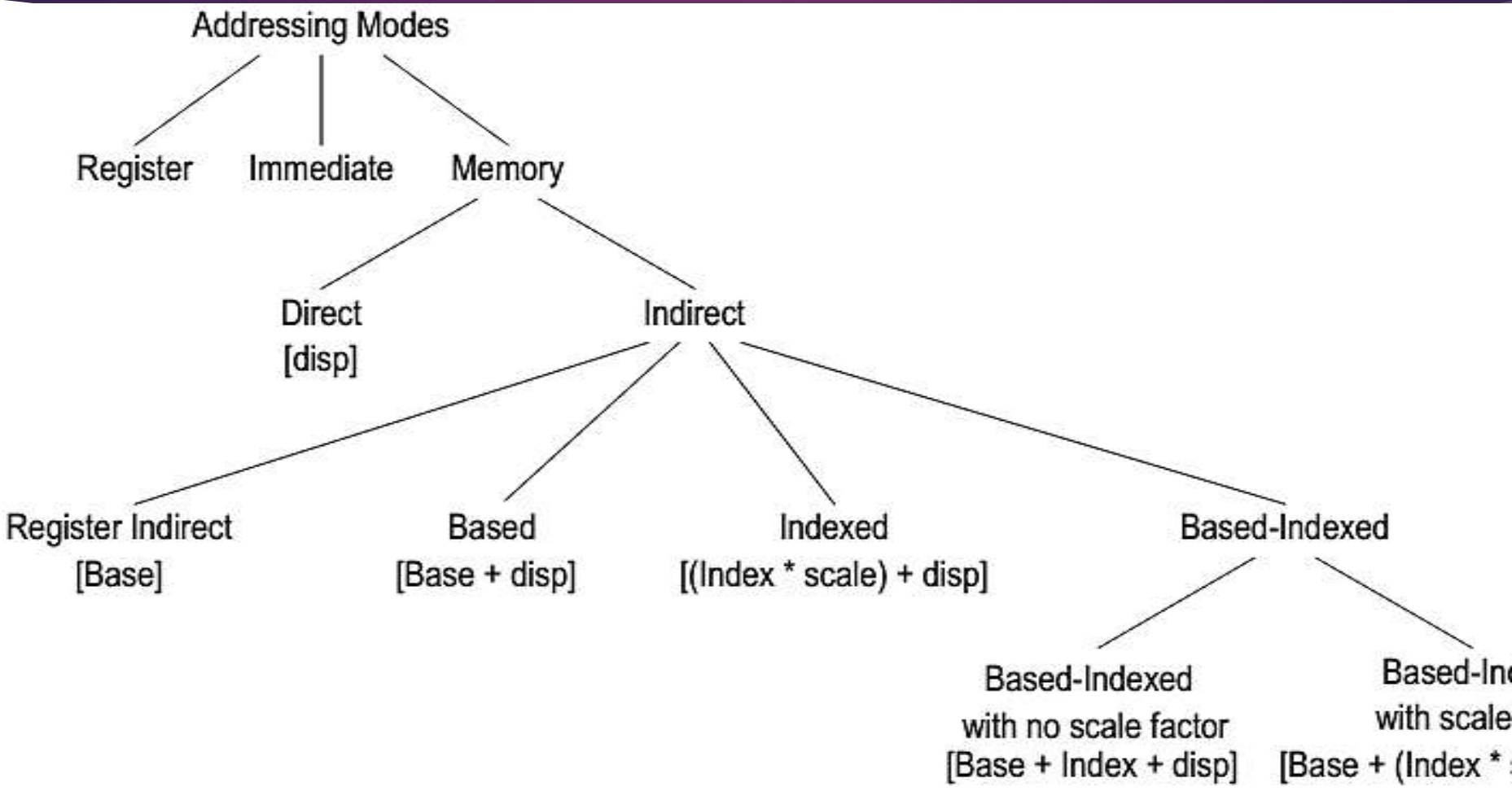
Some Integer Instructions

Format	Computation
One-Operand Instructions	
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = -*Dest*
`notl` *Dest*	*Dest* = ~*Dest*

Operands in X86

- Register: **MOV EAX, EBX**
  - *Copy* content from one register to another
- Immediate: **MOV EAX, 10h**
  - *Copy* constant to register
- Memory: different addressing modes
  - Typically at most one memory operand
  - Complex address computation supported

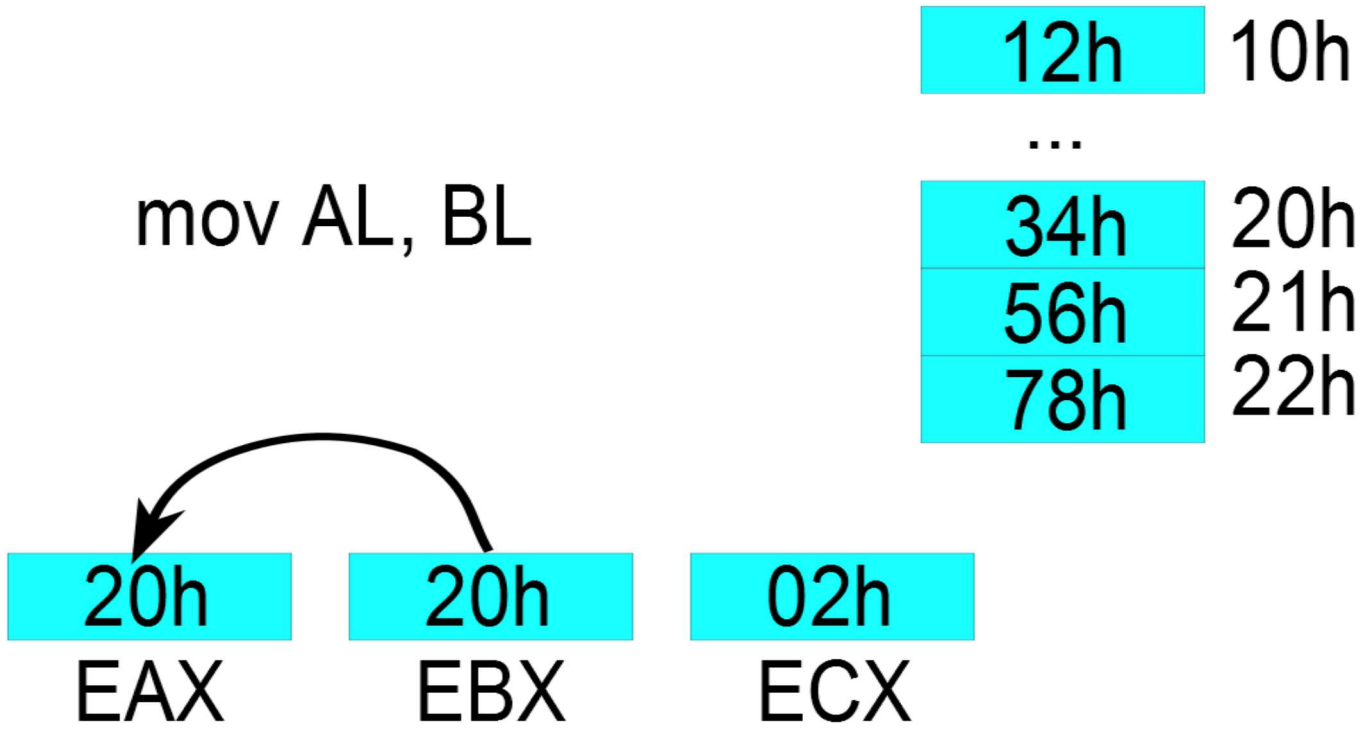
Addressing modes (32-bits)



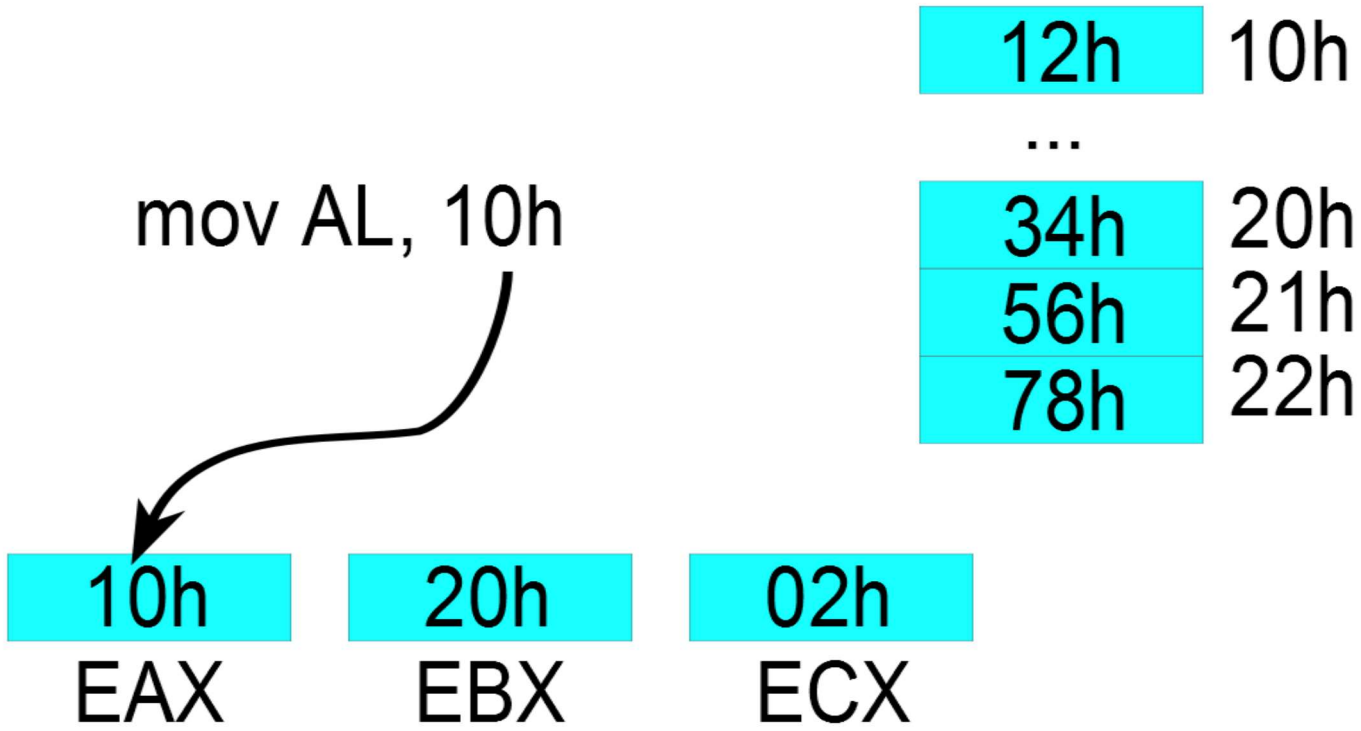
Addressing modes

- Direct: **MOV EAX, [10h]**
  - *Copy* value located at address 10h
- Indirect: **MOV EAX, [EBX]**
  - *Copy* value pointed to by register BX
- Indexed: **MOV AL, [EBX + ECX * 4 + 10h]**
  - *Copy* value from array (BX[4 * CX + 0x10])
- Pointers can be associated to type
  - **MOV AL, byte ptr [BX]**

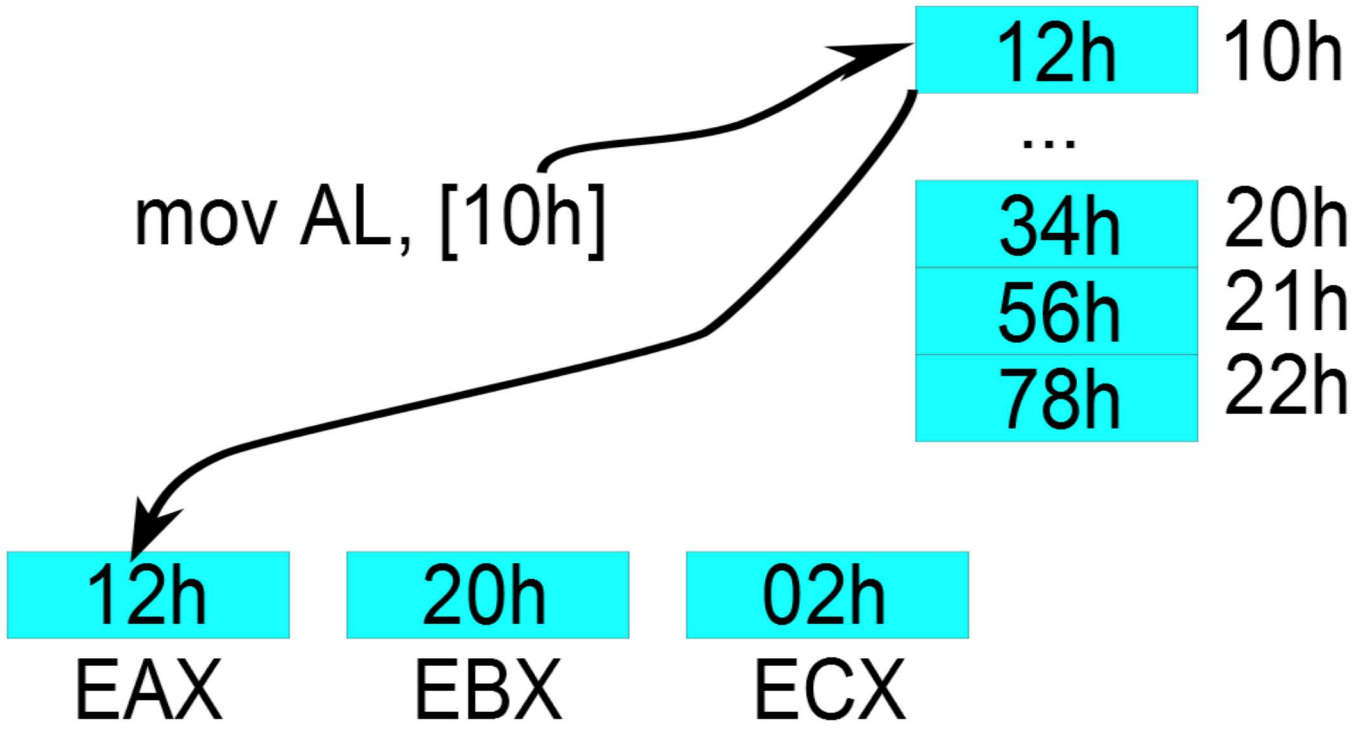
Operands and addressing modes: Register



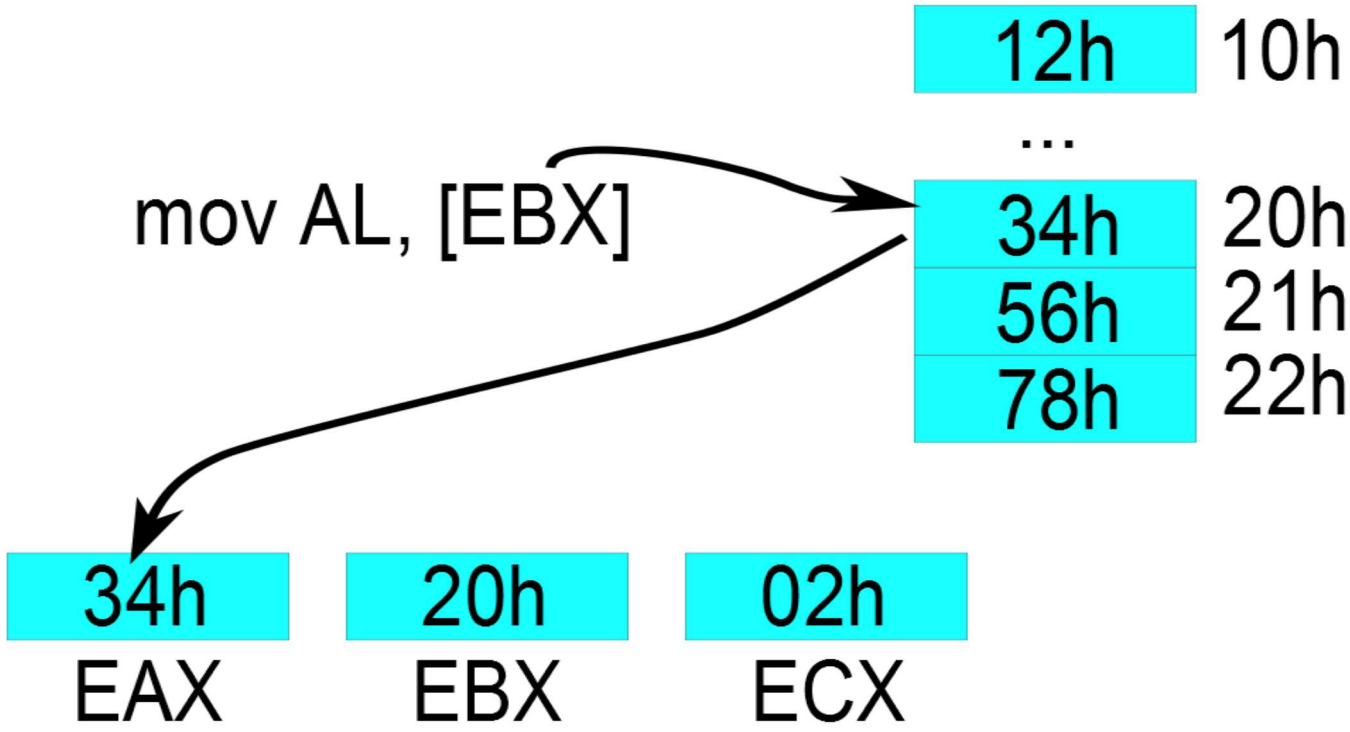
Operands and addressing modes: Immediate



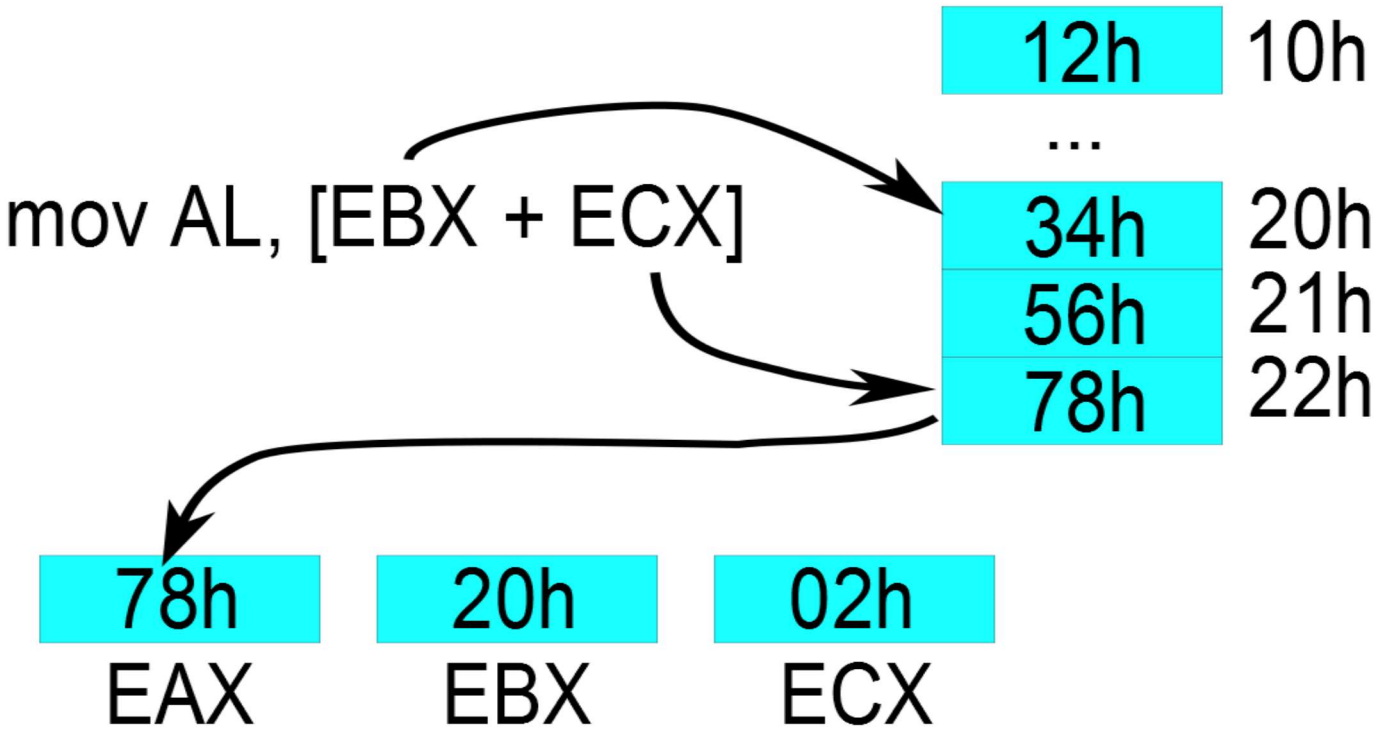
Operands and addressing modes: Direct



Operands and addressing modes: Indirect



Operands and addressing modes: Indexed



Generic 32bit Addressing Mode

- **Segment + Base + (Index * Scale) + displacement**

CS	EAX	EAX	1	no displacement
SS	EBX	EBX	2	8-bit displacement
DS	ECX	ECX	4	32-bit displacement
ES	EDX	EDX	8	
FS	ESI	ESI		
GS	EDI	EDI		
	EBP	EBP		
	ESP			

Addressing Mode GNU-AS

- **D(Rb, Ri, S)** **Mem [Reg[Rb] + S*Reg[Ri] + D]**
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, **except for `%esp`**
    - Unlikely you'd use `%ebp`, either
  - S: Scale: 1, 2, 4, or 8

GAS - Indexed Addressing Modes

- **Special Cases**
  - (Rb, Ri) ***Mem* [Reg[Rb] + Reg[Ri]]**
  - D(Rb, Ri) ***Mem* [Reg[Rb] + Reg[Ri] + D]**
  - (Rb, Ri, S) ***Mem* [Reg[Rb] + S*Reg[Ri]]**

Solve the Followings:

Expression
0x8(%edx)
(%edx,%ecx)
(%edx,%ecx,4)
0x80(,%edx,2)

- Given:

%edx	0xf000
%ecx	0x100

Another Programing Example

Objective:

- Take an integer through Command-line
- Print the value
- Add 1 to that
- Print the result

Another Program (1)

```
SECTION .text
        global main
main:
; set-up phase
        push ebp
        mov ebp, esp

; finish phase
        mov esp, ebp
        pop ebp
        ret
```

Another Program (2)

```
SECTION .text
        global main
main:
; set-up phase
        push ebp
        mov ebp, esp

; get the command-line data
        mov ebx, DWORD [esp + 12]   ; get argv starting address
        mov ebx, [ebx + 4]          ; get the second argument data

; finish phase
        mov esp, ebp
        pop ebp
        ret
```

Another Program (3)

```
SECTION .data
msg:    db      "You Entered - %s", 10, 0  ; print argv[1] data

SECTION .text
        extern printf
        global main

main:
; set-up phase
        push ebp
        mov ebp, esp
```

Another Program (3)

```
; get the command-line data
        mov ebx, DWORD [esp + 12]   ; get argv starting address
        mov ebx, [ebx + 4]          ; get the second argument data

; print the value
        push ebx                    ; put data on stack for call
        push msg                    ; print the value
        call printf

; finish phase
        add esp, 8                  ; esp back to start
        mov esp, ebp
        pop ebp
        ret
```

Another Program (4)

```
SECTION .data
msg:   db     "You Entered - %s", 10, 0  ; print argv[1] data
msg2:  db     "This is int - %d", 10, 0  ; print INT equivalent

SECTION .text
       extern printf
       extern atoi
       global main

main:
; set-up phase
       push ebp
       mov ebp, esp
```

Another Program (4)

```
; get the command-line data
        mov ebx, DWORD [esp + 12]   ; get argv starting address
        mov ebx, [ebx + 4]          ; get the second argument data

; print the value
        push ebx                    ; put data on stack for call
        push msg                    ; print the value
        call printf

; convert to integer
        add esp, 4                  ; stack points to entry edx
        call atoi                   ; call atoi - return in EAX?
```

Another Program (4)

```
; get return value and add 1
        add esp, 4                      ; esp points to start
        inc eax                         ; increase eax for testing

; print the result
        push eax                        ; push arg for print
        push msg2                       ; push print message
        call printf

; finish phase
        add esp, 8                      ; esp back to start
        mov esp, ebp
        pop ebp
        ret
```

Another Program (final)

```
SECTION .data
msg:   db     "You Entered - %s", 10, 0  ; print argv[1] data
msg2:  db     "This is int - %d", 10, 0  ; print INT equivalent

SECTION .text
       extern printf
       extern atoi
       global main

main:
; set-up phase
       push ebp
       mov ebp, esp
```

Another Program (final)

```
; get the command-line data
        mov ebx, DWORD [esp + 12]   ; get argv starting address
        mov ebx, [ebx + 4]          ; get the second argument data

; print the value
        push ebx                    ; put data on stack for call
        push msg                    ; print the value
        call printf

; convert to integer
        add esp, 4                  ; stack points to entry edx
        call atoi                   ; call atoi - return in EAX?
```

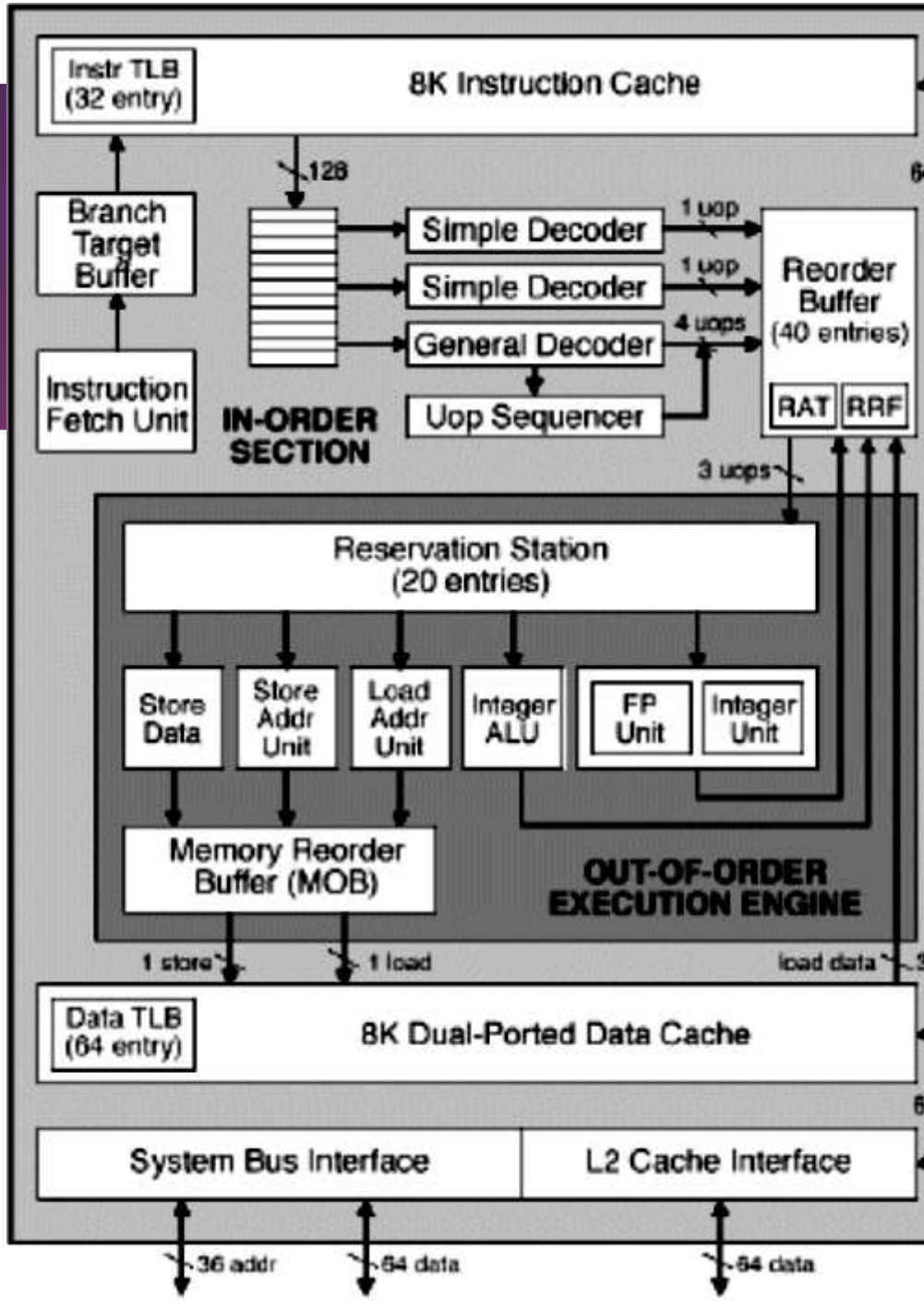
Another Program (final)

```
; get return value and add 1
        add esp, 4                      ; esp points to start
        inc eax                         ; increase eax for testing

; print the result
        push eax                        ; push arg for print
        push msg2                       ; push print message
        call printf

; finish phase
        add esp, 8                      ; esp back to start
        mov esp, ebp
        pop ebp
        ret
```

PentiumPro Block Diagram



Operations

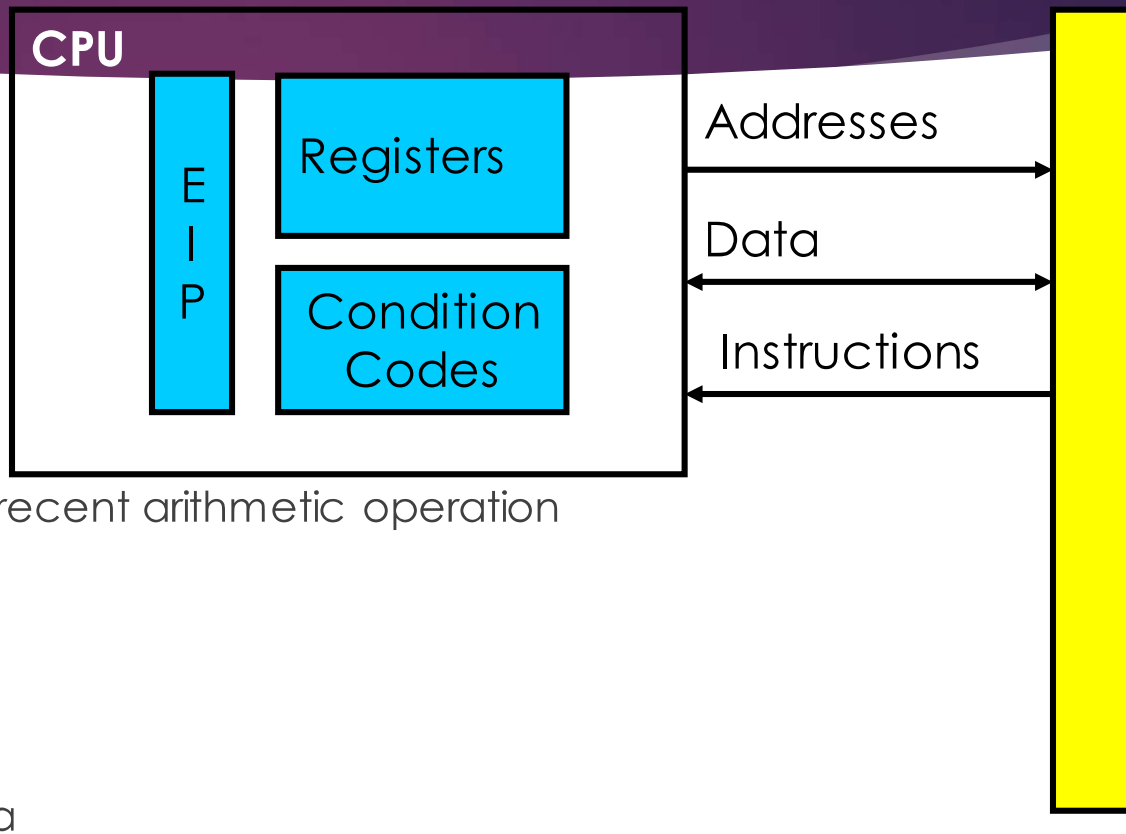
- Translates instructions dynamically into “Uops” / “µops”
  - 128 bits wide
- Holds
  - Operation 32
  - Two sources, and + 32 + 32
  - Destination + 32 = 128bits **WHY??**
- Executes Uops with **“Out of Order”** engine
  - Uop executed when
    1. Operands are available
    2. Functional unit available

Operations

- Execution controlled by "Reservation Stations"
  - Keeps track of data **dependencies** between uops
  - Allocates resources
- Consequences
  - Indirect relationship between
    - IA32 code &
    - What actually gets executed
  - Tricky to predict / optimize performance at assembly level

Machine View

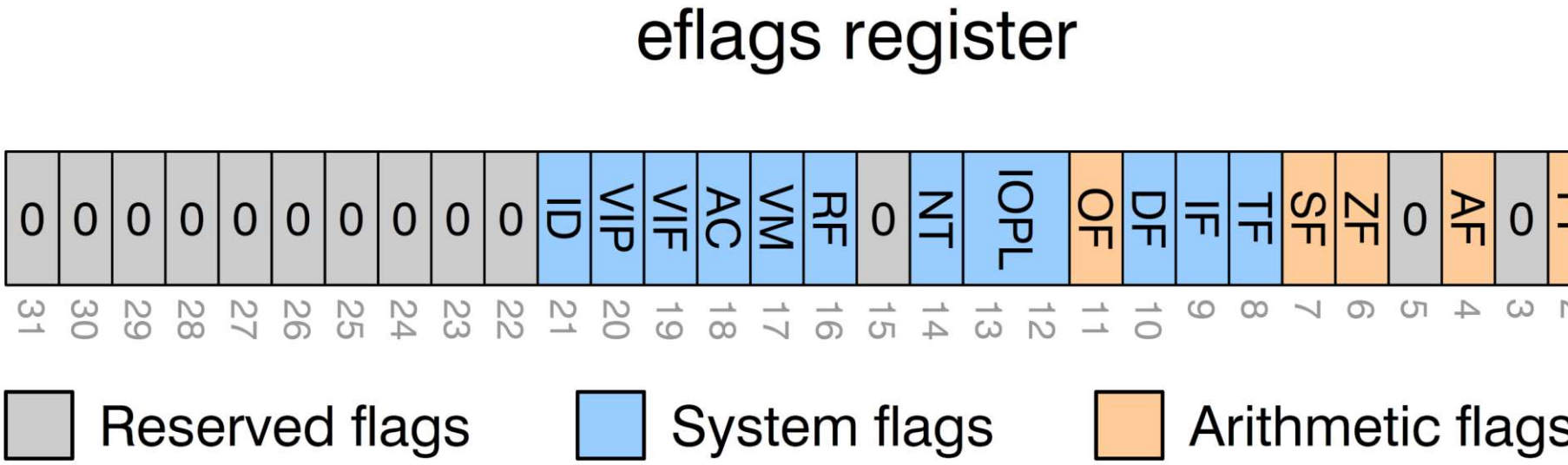
- EIP (Program Counter)
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures



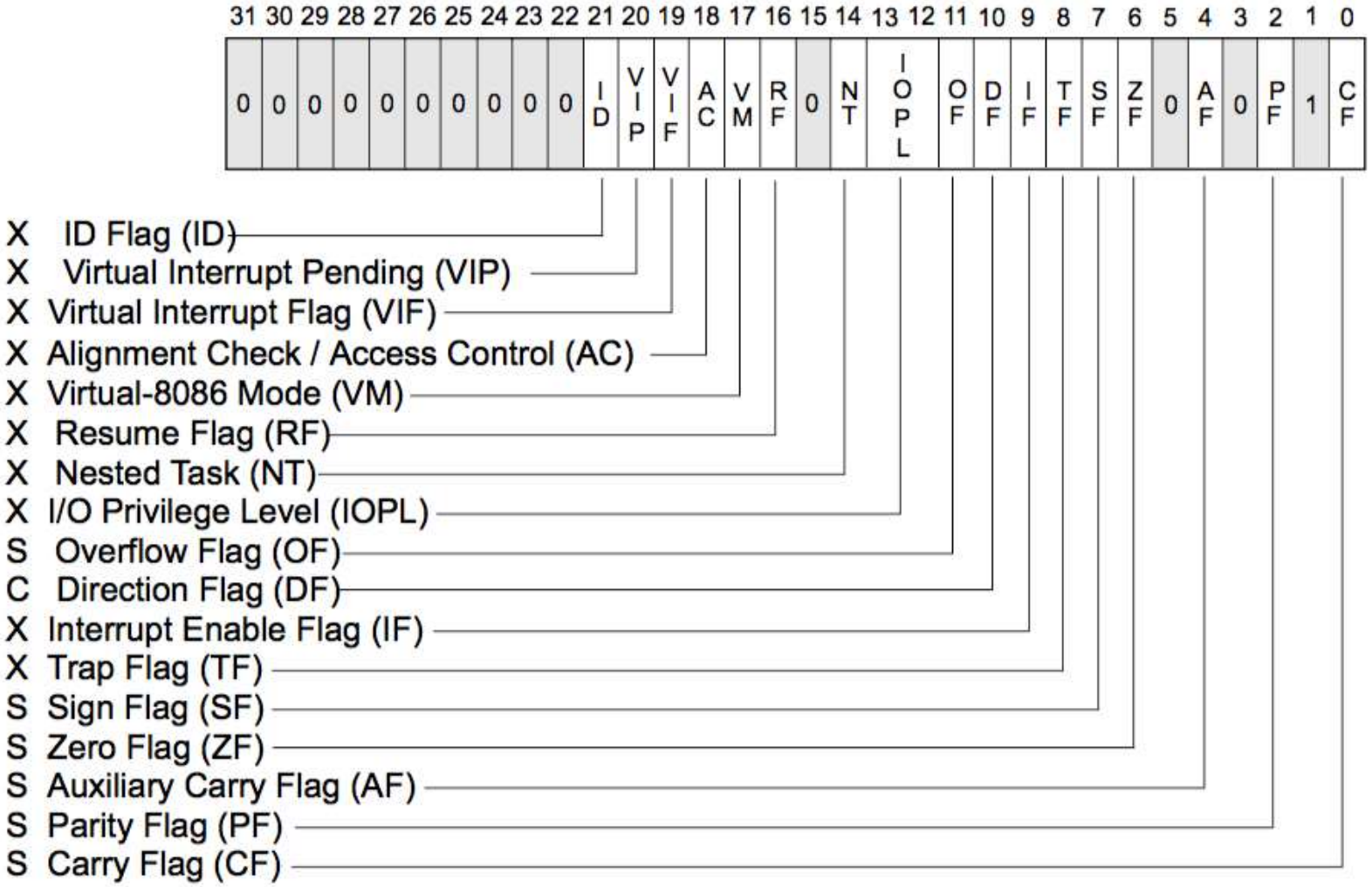
Flow Control Instructions

- JMP
- Jcc
- CALL
- RET

Revisit EFLAGS



Revisit EFLAGS



JMP: jump

- **Syntax**:

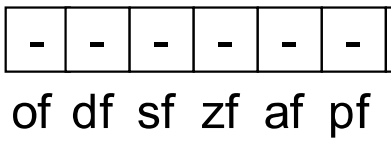
  `JMP` *dest*

- **Operation (absolute jump)**:

  `EIP` ← *dest*

- **Operation (relative jump)**:

  `EIP` ← `EIP` + *dest*



Unconditional Jumps

- Jmp statement label

- We have two types of jumps,
  - Intersegment
  - Intrasegment
- Address can be in a register, variable or label.

Unconditional Jumps

- Example:

```
Start:    MOV AX, 0
          INC AX,
          JMP Start
```

Jcc: short jump conditional

- **Syntax**:

```
Jcc dest
```

- **Operation**:

```
if(cc)
    EIP ← EIP + dest
endif
```

- **Notes:** *cc* is any of the condition codes. *dest* must be within a signed 8-bit range (-128 to 127).

-	-	-
of	df	sf

Condition Codes

- Implicitly Set By Arithmetic Operations

  `addl` ***Src*, *Dest***

  C analog: `t = a + b`

  - CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  - ZF set if `t == 0`
  - SF set if `t < 0`
  - OF set if two's complement overflow

    ```
    (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
    ```

- ***Not*** Set by `leal` instruction

Condition Codes

Sufix	Meaning	Flags
O	*Overflow*	`OF=1`
NO	*No Overflow*	`OF=0`
C	*Carry*	`CF=1`
B	*Below*	
NAE	*Not Above nor Equal*	
NC	*No Carry*	`CF=0`
NB	*Not Below*	
AE	*Above or Equal*	

Condition Codes (continued)

Sufix	Meaning	Flags
Z E	*Zero* *Equal*	`ZF=1`
NZ NE	*Not Zero* *Not Equal*	`ZF=0`
BE NA	*Below or Equal* *Not Above*	`CF=1 OR ZF=1`
A NBE	*Above* *Not Below nor Equal*	`CF=0 AND ZF=0`

Condition Codes (continued)

Sufix	Meaning	Flags
S	*Sign*	`SF=1`
NS	*Not Sign*	`SF=0`
P PE	*Parity* *Parity Even*	`PF=1`
NP PO	*Not Parity* *Parity Odd*	`PF=0`

Condition Codes (continued)

Sufix	Meaning	Flags
L NGE	*Less* *Not Greater nor Equal*	SF<>OF
GE NL	*Greater or Equal* *Not Less*	SF=OF
LE NG	*Less or Equal* *Not Greater*	ZF=1 OR SF<>OF
G NLE	*Greater* *Not Less nor Equal*	ZF=0 AND SF=OF

Condition Codes (continued)

- **Above** and **Below** are used for unsigned integer comparisons.
- **Greater** and **Less** are used for signed integer comparisons.

Conditional Jumps

- Dependent on condition codes.
- Example:

JZ → jump if zero flag is set.

Conditional Codes

- Code the following C routine using assembly languag instructions.

Add a value X to eax;
If x < 0
Then
… (body for negative condition)
Else if x = 0
… (body for zero condition)
Else
… (body for positive condition)
End if

Conditional Codes

- Solution

```
        Add eax, X          ;add a value to eax
        JNS elseIfZero      ;jump if eax is not negative
        …                   ; code for negative condition
        JMP endCheck
elseifZero:
        JNZ elsePos         ; jump if x is not zero
        …                   ; code for zero condition
        JMP endCheck
elsePos: …                  ; code for positive balance
endCheck:
```

Conditional Codes

```
        Add eax, X          ;add a value to eax
        JNS elseIfZero      ;jump if eax is not negative
        …                   ; code for negative condition
        JMP endCheck
elseifZero:
        JNZ elsePos         ; jump if x is not zero
        …                   ; code for zero condition
        JMP endCheck
elsePos:
        …                   ; code for positive balance
endCheck:
```

Add a value X to eax

If x < 0

Then

… (body for negative co

Else if x = 0

… (body for zero conditi

Else

… (body for positive co

End if