



Assembly Programming with NASM

DR. ARKA PROKASH MAZUMDAR

NASM Data Directives

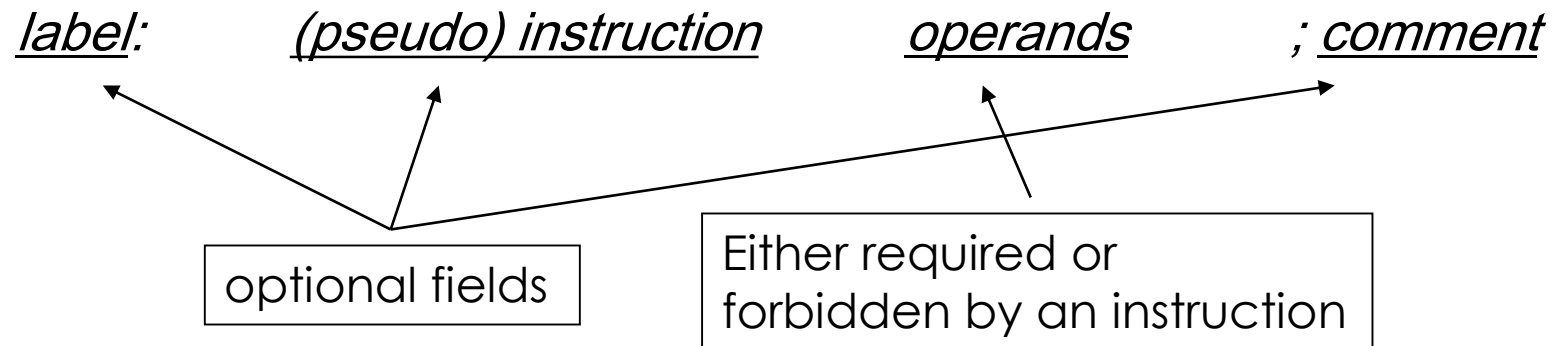
▶ L1	db	0	; byte
▶ L2	dw	1000	; word
▶ L3	db	110101b	; byte
▶ L4	db	12h	; byte
▶ L5	db	17o	; byte
▶ L6	dd	1A92h	; double word
▶ L7	resb	1	; uninitialized byte
▶ L8	db	'A'	; ascii code = 'A'
▶ L9	db	0,1,2,3	; 4 bytes
▶ L10	db	'w','o','r','d',0	; string
▶ L11	db	'word', 0	
▶ L12	times 100 db	0	; 100 bytes 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 (bytes)
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	10/12

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 pointer
3      movl     %esp, %ebp          Create new frame pointer
4      movl     8(%ebp), %edx        Retrieve xp
5      movl     12(%ebp), %eax       Retrieve y
6      addl     (%edx), %eax         Add *xp to get t
7      movl     %eax, (%edx)         Store t at xp
8      popl     %ebp                Restore frame pointer
9      ret                                Return
```

Operand Modes

Source Destination C Analog

movl	Imm	Reg	movl \$0x4,%eax	temp = 0x4;
		Mem	movl \$-147, (%eax)	*p = -147;
	Reg	Reg	movl %eax,%edx	temp2 = temp1;
		Mem	movl %eax, (%edx)	*p = temp;
	Mem	Reg	movl (%eax), %edx	temp = *p;

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 local variables.
- ▶ 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 the stack correctly so that **the right number is popped off by the RET.**

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
    } Set Up

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

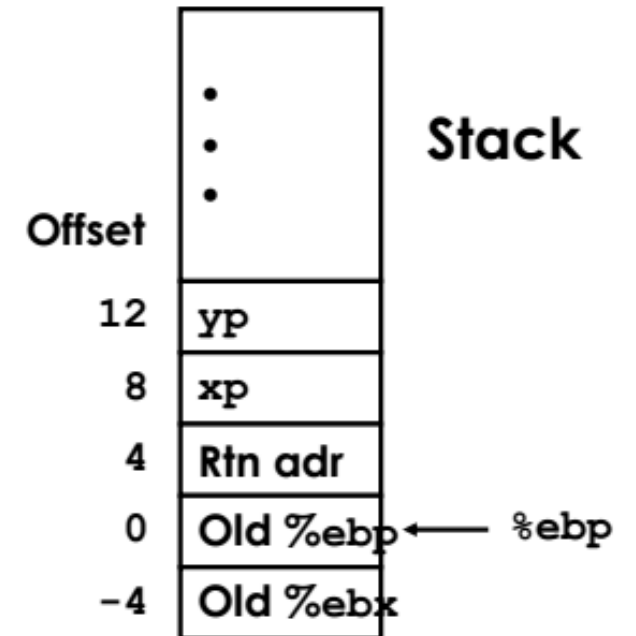
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
    } Finish
```

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

```

		123	0x124
		456	0x120
			0x11c
			0x118
	Offset		0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

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

```

		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
	Offset		
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		0x100

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

```

		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		0x100

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

```

		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		0x100

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

```

		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
	Offset		
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		0x100

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

```

		456	0x124
		456	0x120
			0x11c
			0x118
			0x114
			0x110
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		0x100

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

```

		456	0x124
		123	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

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`

$Dest = -Dest$

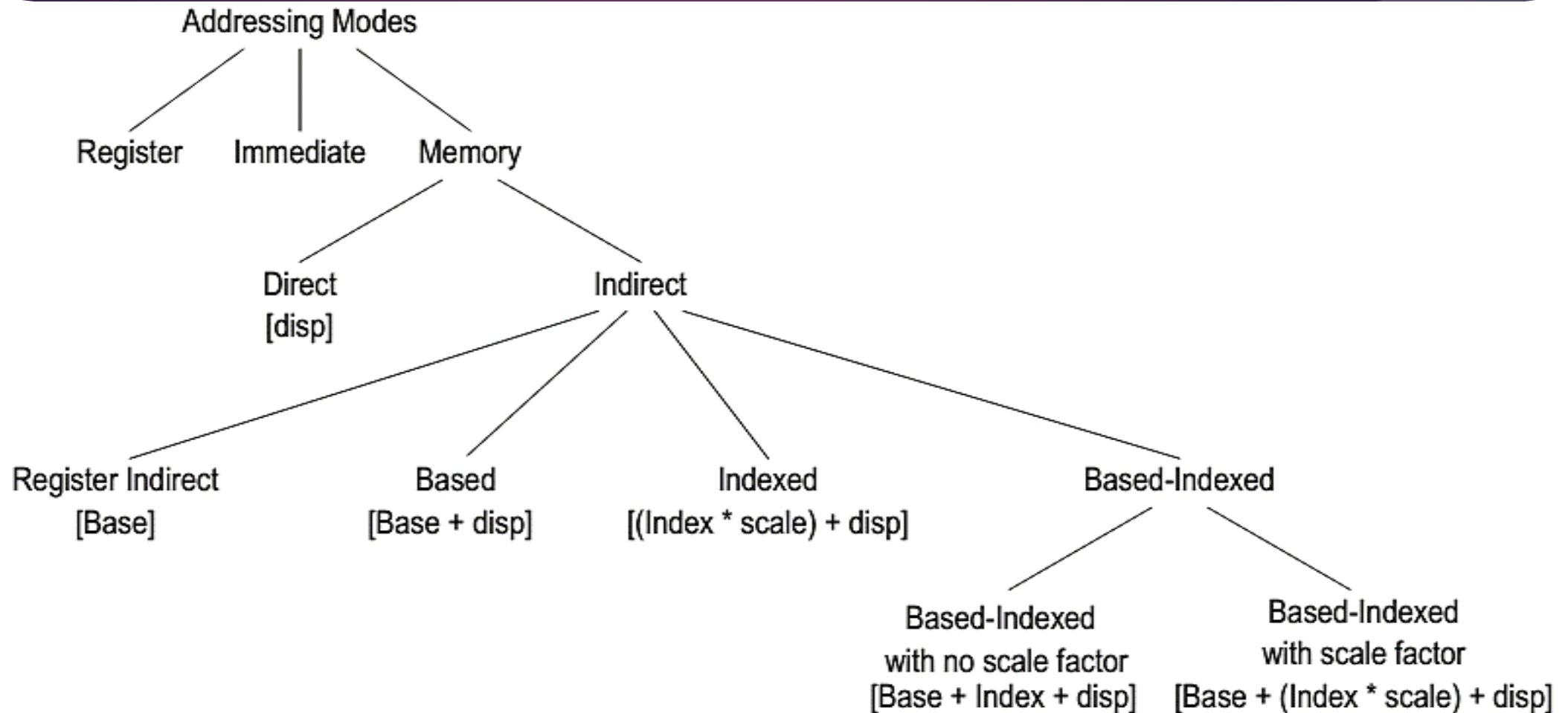
`notl Dest`

$Dest = \sim Dest$

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)



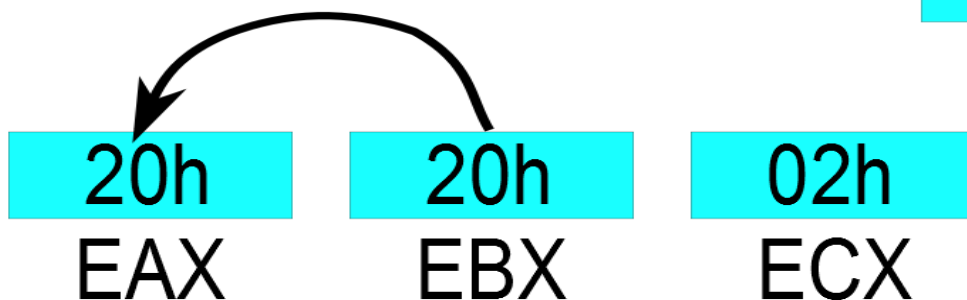
Addressing modes

- ▶ 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:

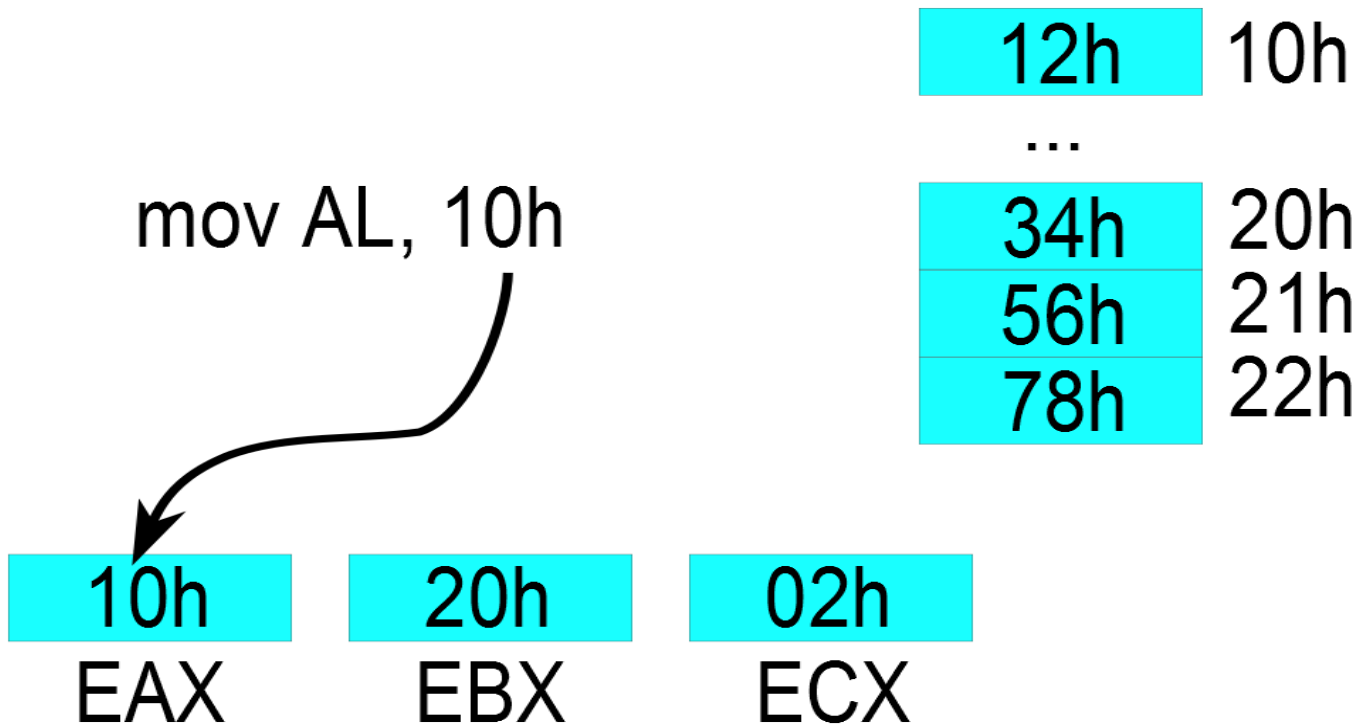
Register

mov AL, BL



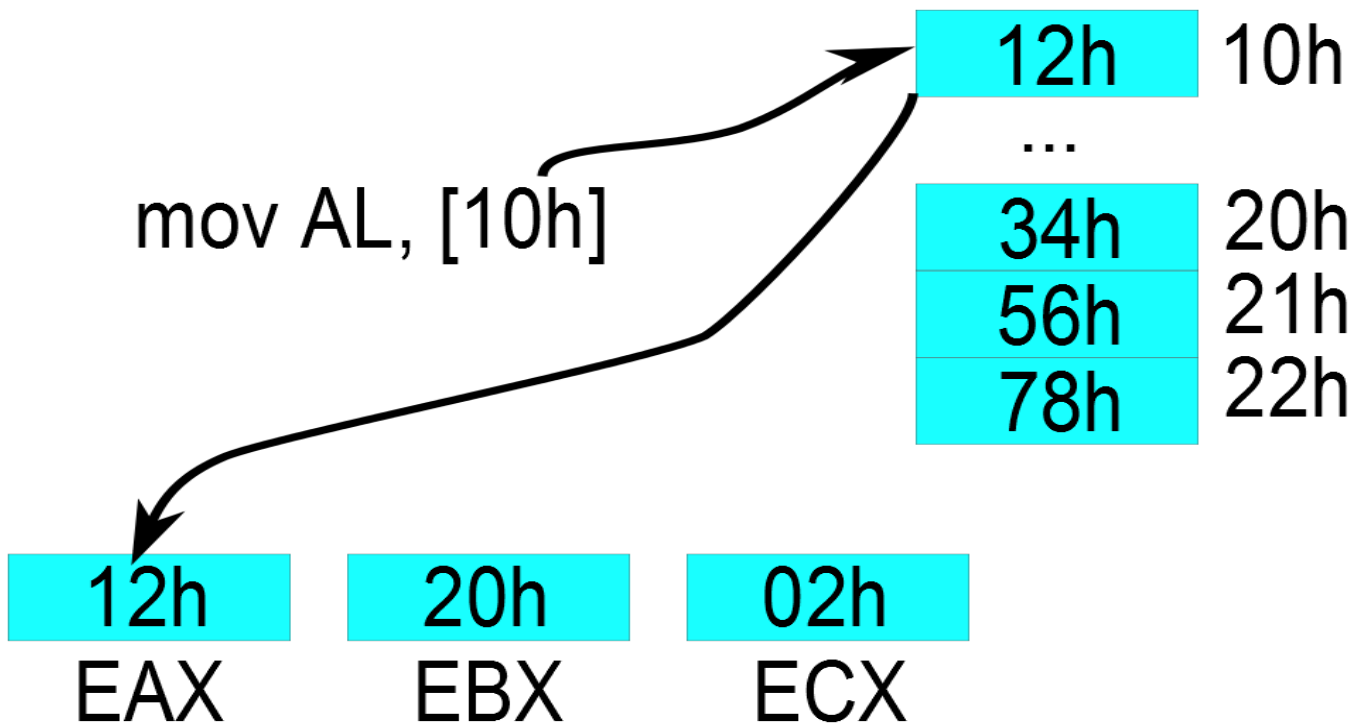
12h	10h
...	
34h	20h
56h	21h
78h	22h

Operands and addressing modes: Immediate

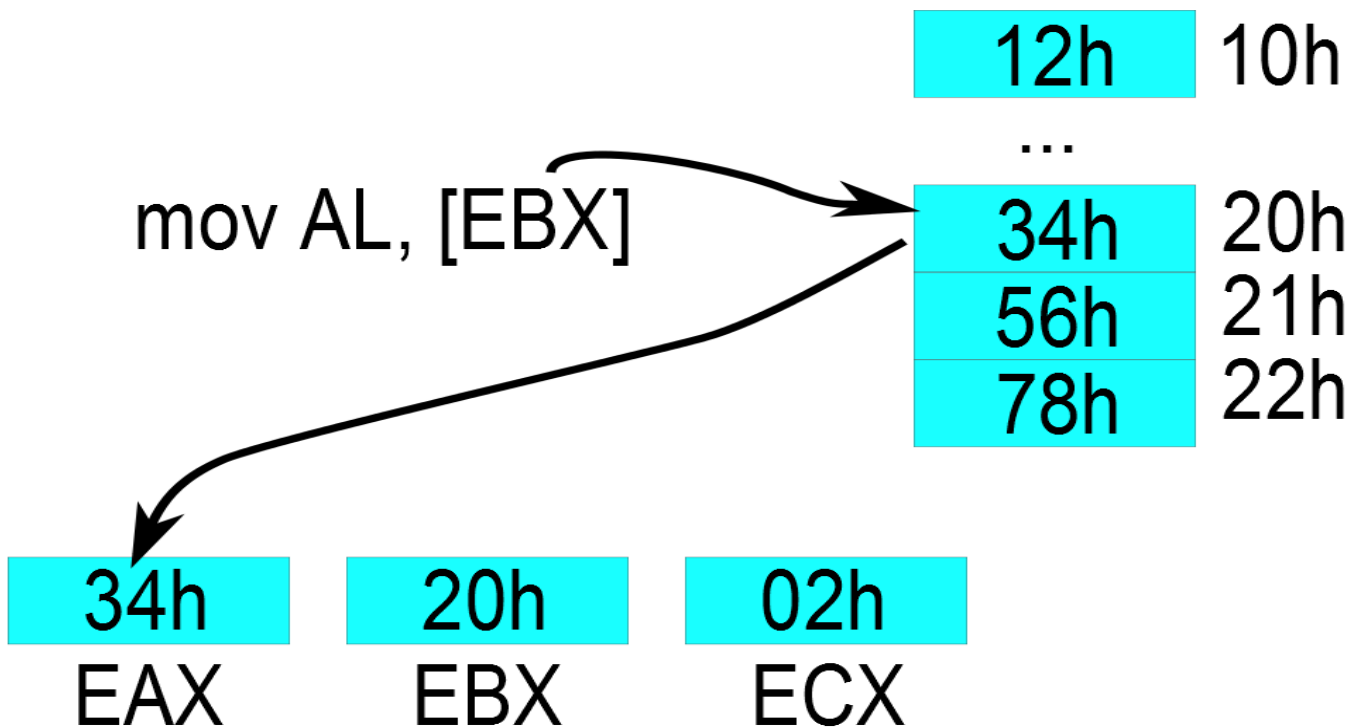


Operands and addressing modes:

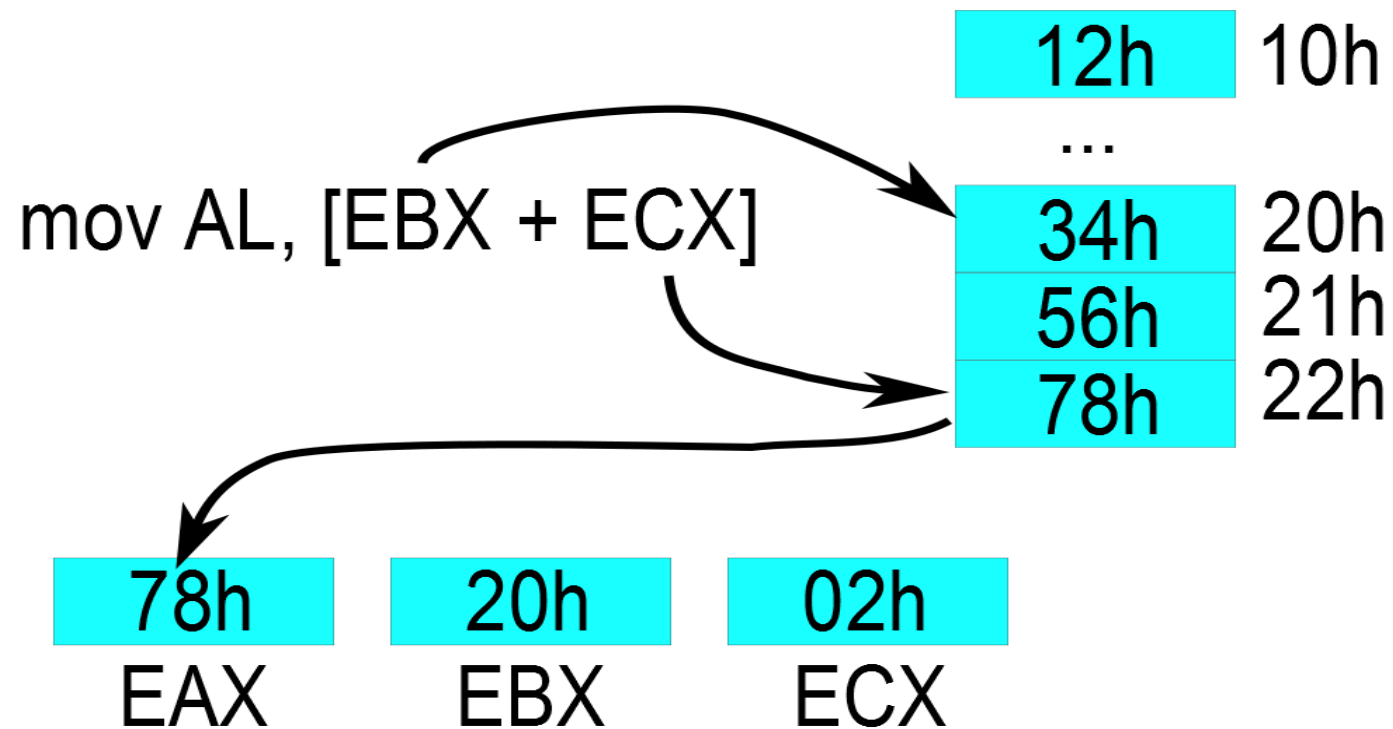
Direct



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  
inc eax
```

```
; esp points to start  
; increase eax for testing
```

; print the result

```
push eax  
push msg2  
call printf
```

```
; push arg for print  
; push print message
```

; finish phase

```
add esp, 8  
mov esp, ebp  
pop ebp  
ret
```

```
; esp back to start
```

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  
inc eax
```

```
; esp points to start  
; increase eax for testing
```

; print the result

```
push eax  
push msg2  
call printf
```

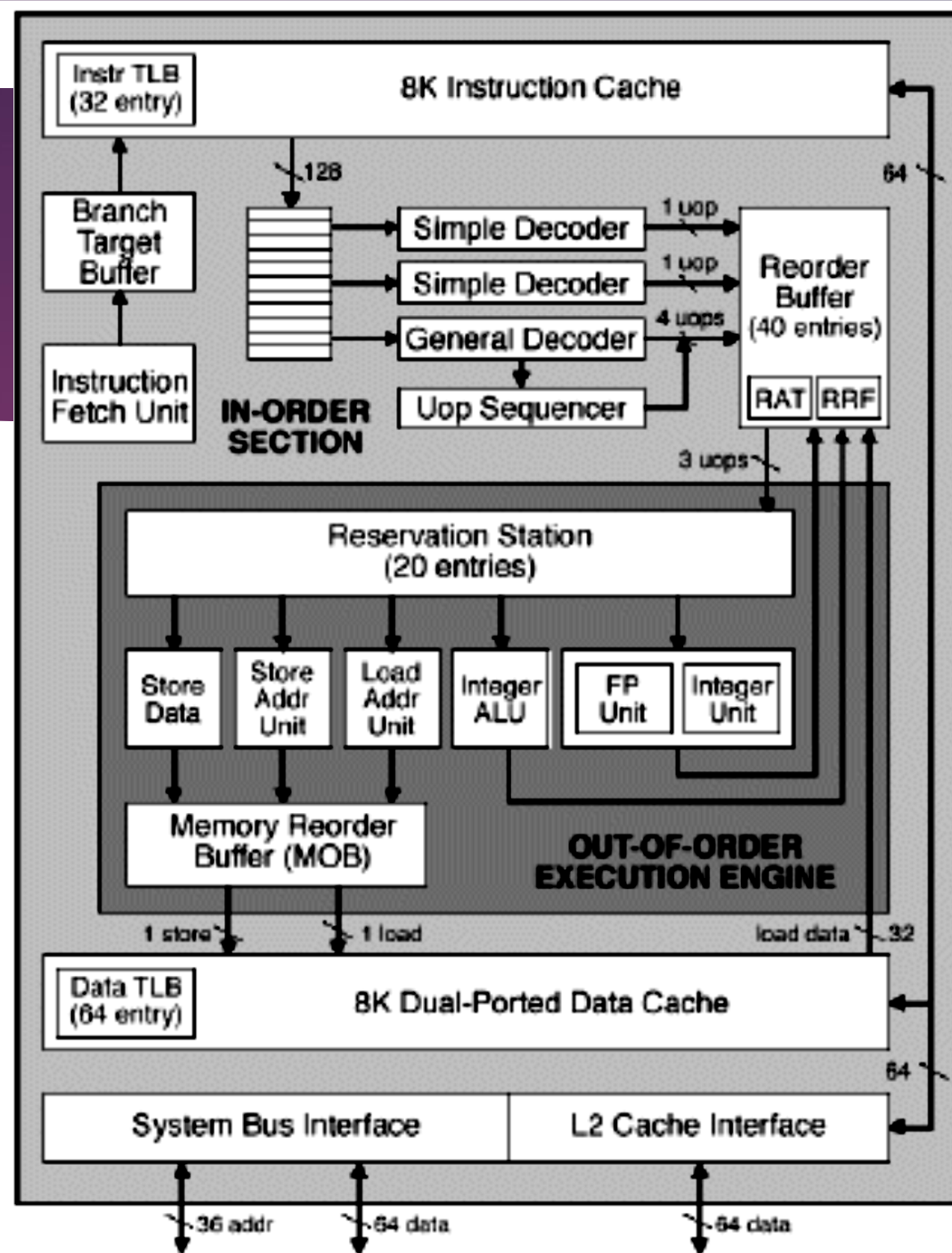
```
; push arg for print  
; push print message
```

; finish phase

```
add esp, 8  
mov esp, ebp  
pop ebp  
ret
```

```
; esp back to start
```

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

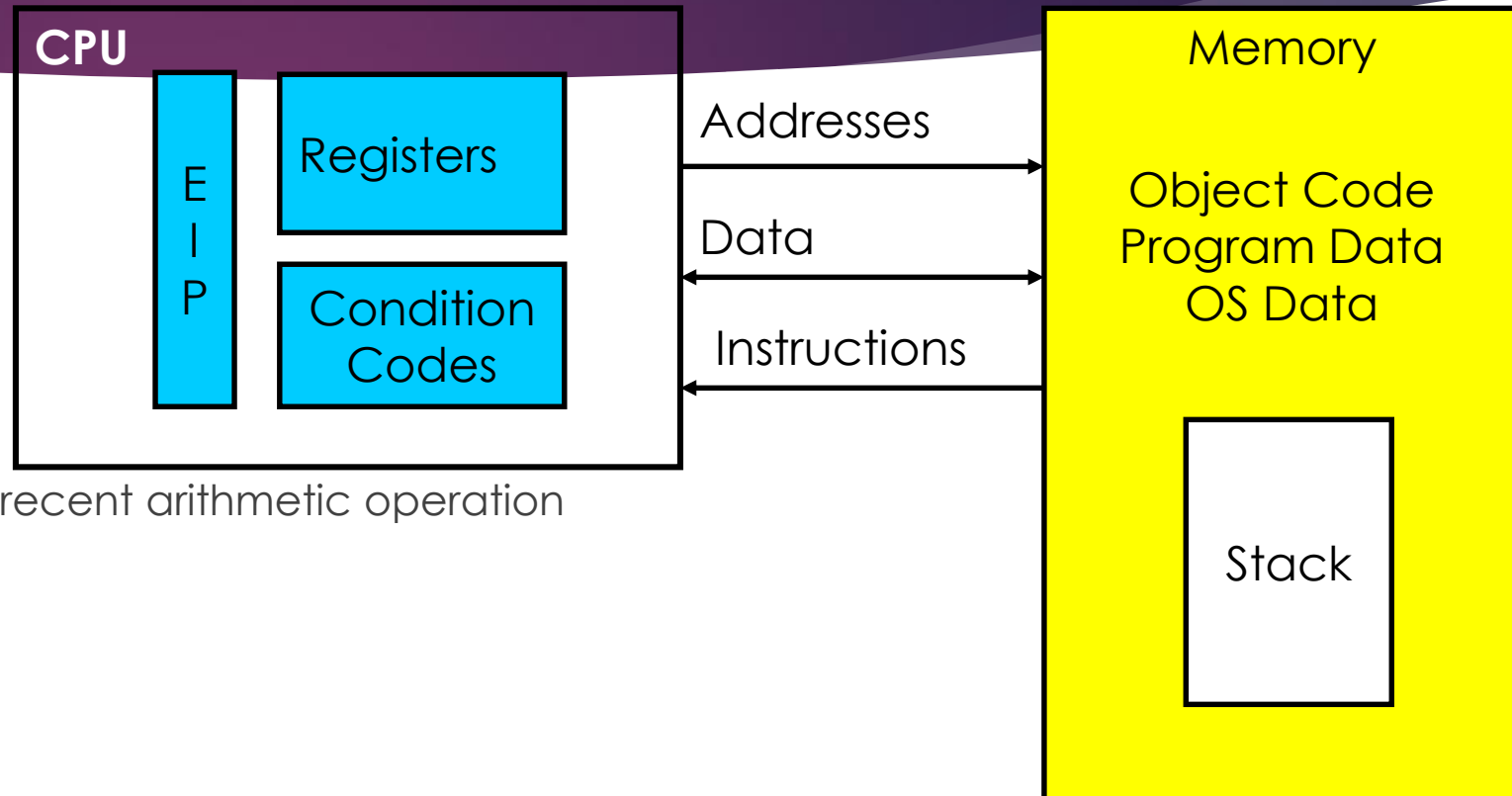
- Heavily used program data

► 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

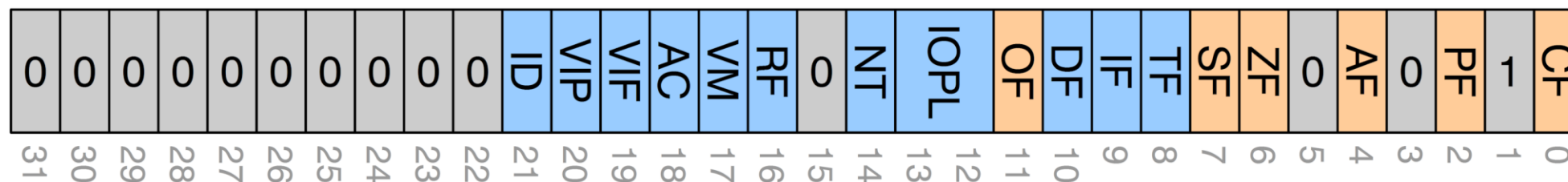


Flow Control Instructions

- ▶ JMP
- ▶ Jcc
- ▶ CALL
- ▶ RET

Revisit EFLAGS

eflags register



 Reserved flags

 System flags

 Arithmetic flags

Revisit EFLAGS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

- X ID Flag (ID) _____
- X Virtual Interrupt Pending (VIP) _____
- X Virtual Interrupt Flag (VIF) _____
- X Alignment Check / Access Control (AC) _____
- X Virtual-8086 Mode (VM) _____
- X Resume Flag (RF) _____
- X Nested Task (NT) _____
- X I/O Privilege Level (IOPL) _____
- S Overflow Flag (OF) _____
- C Direction Flag (DF) _____
- X Interrupt Enable Flag (IF) _____
- X Trap Flag (TF) _____
- S Sign Flag (SF) _____
- S Zero Flag (ZF) _____
- S Auxiliary Carry Flag (AF) _____
- S Parity Flag (PF) _____
- S Carry Flag (CF) _____

- S Indicates a Status Flag
- C Indicates a Control Flag
- X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

JMP: jump

► Syntax:

`JMP dest`

► Operation (absolute jump):

$EIP \leftarrow dest$

► Operation (relative jump):

$EIP \leftarrow EIP + dest$

-	-	-	-	-	-	-
---	---	---	---	---	---	---

of df sf zf af pf cf

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	zf	af	pf	cf

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

$(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t \geq 0)$

► **Not** Set by `leal` instruction

Condition Codes

Suffix	Meaning	Flags
O	<i>Overflow</i>	OF=1
NO	<i>No Overflow</i>	OF=0
C	<i>Carry</i>	CF=1
B	<i>Below</i>	
NAE	<i>Not Above nor Equal</i>	
NC	<i>No Carry</i>	CF=0
NB	<i>Not Below</i>	
AE	<i>Above or Equal</i>	

Condition Codes (continued)

Suffix	Meaning	Flags
Z E	<i>Zero</i> <i>Equal</i>	ZF=1
NZ NE	<i>Not Zero</i> <i>Not Equal</i>	ZF=0
BE NA	<i>Below or Equal</i> <i>Not Above</i>	CF=1 OR ZF=1
A NBE	<i>Above</i> <i>Not Below nor Equal</i>	CF=0 AND ZF=0

Condition Codes (continued)

Suffix	Meaning	Flags
S	<i>Sign</i>	SF=1
NS	<i>Not Sign</i>	SF=0
P	<i>Parity</i>	PF=1
PE	<i>Parity Even</i>	
NP	<i>Not Parity</i>	PF=0
PO	<i>Parity Odd</i>	

Condition Codes (continued)

Suffix	Meaning	Flags
L NGE	<i>Less</i> <i>Not Greater nor Equal</i>	SF<>OF
GE NL	<i>Greater or Equal</i> <i>Not Less</i>	SF=OF
LE NG	<i>Less or Equal</i> <i>Not Greater</i>	ZF=1 OR SF<>OF
G NLE	<i>Greater</i> <i>Not Less nor Equal</i>	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 language 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 condition)

Else if $x = 0$

... (body for zero condition)

Else

... (body for positive condition)

End if

Setting Codes

Explicit Setting by Compare Instruction

`cmp Src1, Src2`

- ▶ like computing **Src1-Src2** without setting destination
- ▶ **CF** set if carry out from most significant bit
 - ▶ Used for unsigned comparisons
- ▶ **ZF** set if `a == b`
- ▶ **SF** set if `(a-b) < 0`
- ▶ **OF** set if two's complement overflow
`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Compare Examples

- ▶ OP1= 3B
- ▶ OP2= 3B
- ▶ CF=OF=SF=0
- ▶ ZF=1
- ▶ OP1==OP2 signed and unsigned

Compare Examples

- ▶ $OP1 = 3B$
- ▶ $OP2 = 15$
- ▶ $OP1 - OP2 = 26$
- ▶ $CF = OF = SF = ZF = 0$
- ▶ $OP1 > OP2$ signed and unsigned

Compare Examples

- ▶ OP1=15
- ▶ OP2= F6
- ▶ OP1-OP2=1F
- ▶ CF=1 – borrow
- ▶ SF=OF=ZF=0
- ▶ Signed operation = op1>op2
- ▶ Unsigned operation =op1 < op2

$$15 - F6 \Rightarrow 15 + 0A$$

$$\begin{array}{r} 0001 \ 0101 \\ +0000 \ 1010 \\ \hline \mathbf{0001 \ 1111 = 1F} \end{array}$$

Compare Examples

Legal Examples

Cmp eax, 356

cmp value, 03dh

Cmp bh, '\$'

Illegal examples

Cmp 1000, total

Compare Programming Ex.

- ▶ Code the following routine in assembly language.

If val < 10

Then

add 1 to xcount;

Else

add 1 to ycount;

End if;

Compare Programming Ex

► Solution:

```
Cmp ebx, 10      ;value < 10
Jnl Elsey
Inc xcount       ;add 1 to xcount
Jmp endVal
Elsey:
Inc ycount       ;add 1 to ycount
endVal:
```


Compare Examples for JNL

- ▶ OP1=7F
- ▶ OP2= FF
- ▶ OP1-OP2=80
- ▶ **SF=1**
- ▶ **OF=1**
- ▶ ZF=0
- ▶ Signed operation: **op1>=op2**
- ▶ Unsigned operation =op1 < op2

7F – FF => 7F + 01 Assuming the Registers are 8bit in size

$$\begin{array}{r}
 0111 \ 1111 \\
 +0000 \ 0001 \\
 \hline
 1000 \ 0000 \quad = \ 80
 \end{array}$$