Assembly Programing with NASM *(and GAS)*

DR. ARKA PROKASH MAZUMDAR

NASM Data Directives

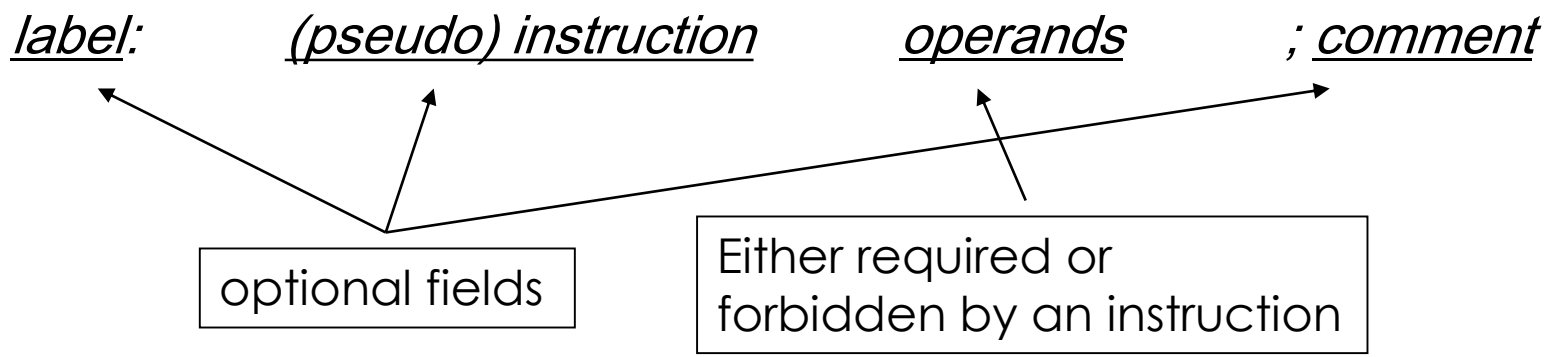
```
L1      db              0               ; byte
L2      dw              1000            ; word
L3      db              110101b         ; byte
L4      db              12h             ; byte
L5      db              17o             ; byte
L6      dd              1A92h           ; double word
L7      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 (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;

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)
```

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]

Instructions: Setup

- POP
  - reads the double word at [ESP]
  - And then adds 4 to ESP

- CALL
  - Call subprogram
  - Make an unconditional jump to a subprogram
  - And pushes the address of the next instruction on the stack

Instructions: Setup

- 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

    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

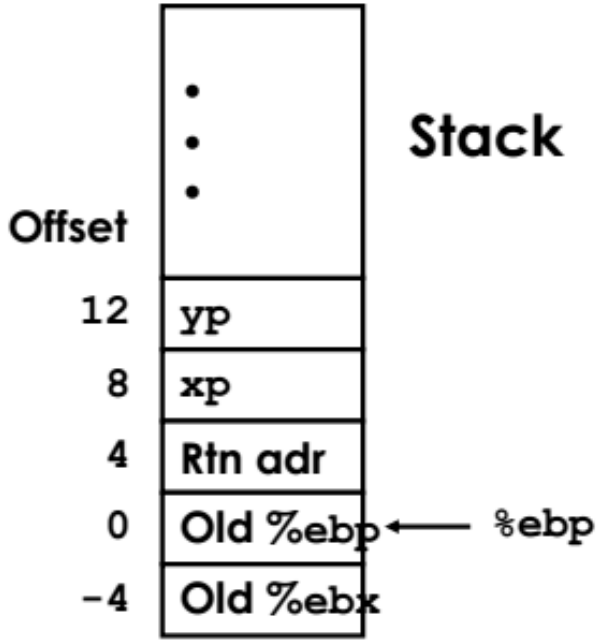
Finish

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;
}
```

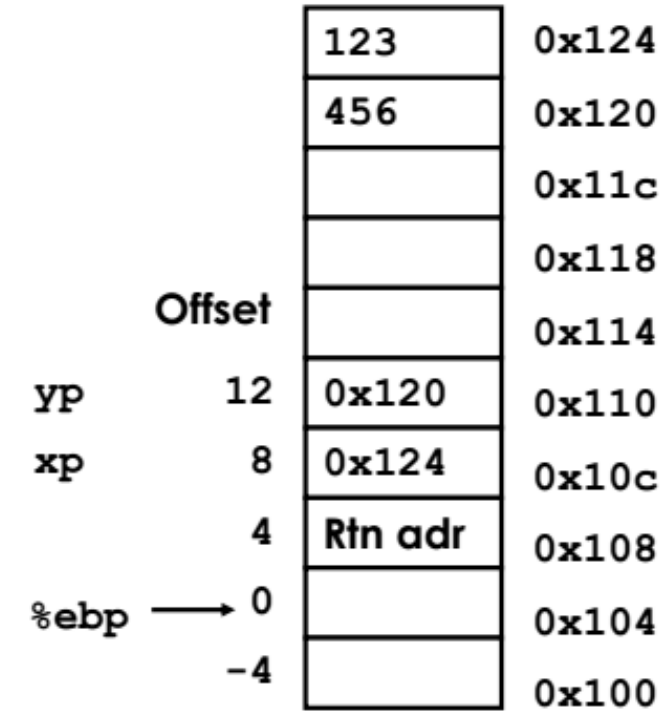
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

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

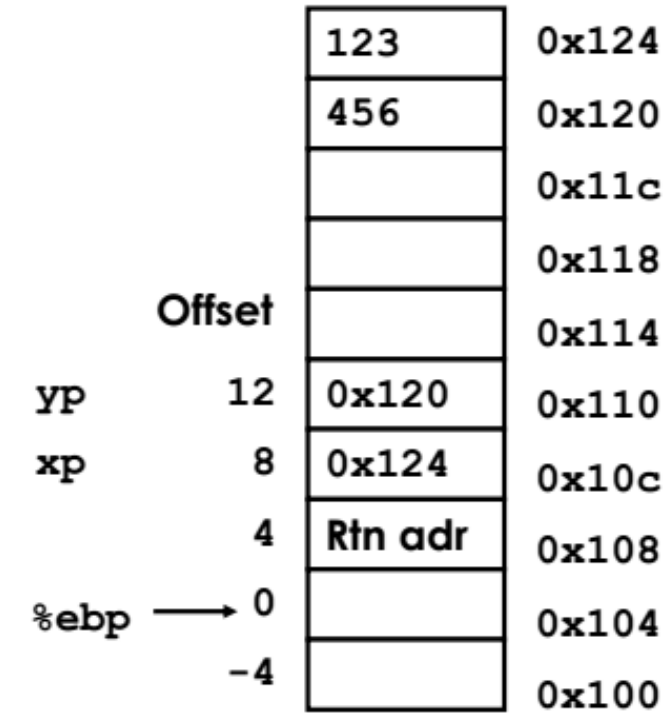
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

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

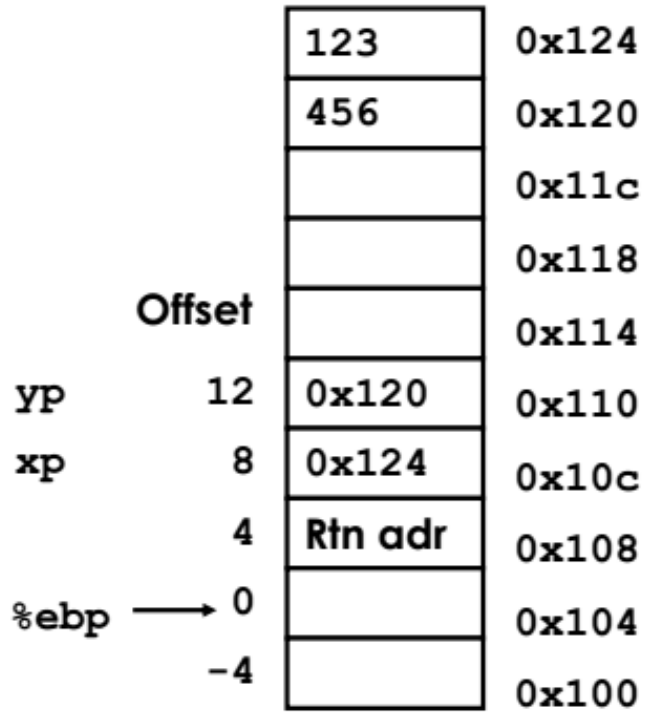
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

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

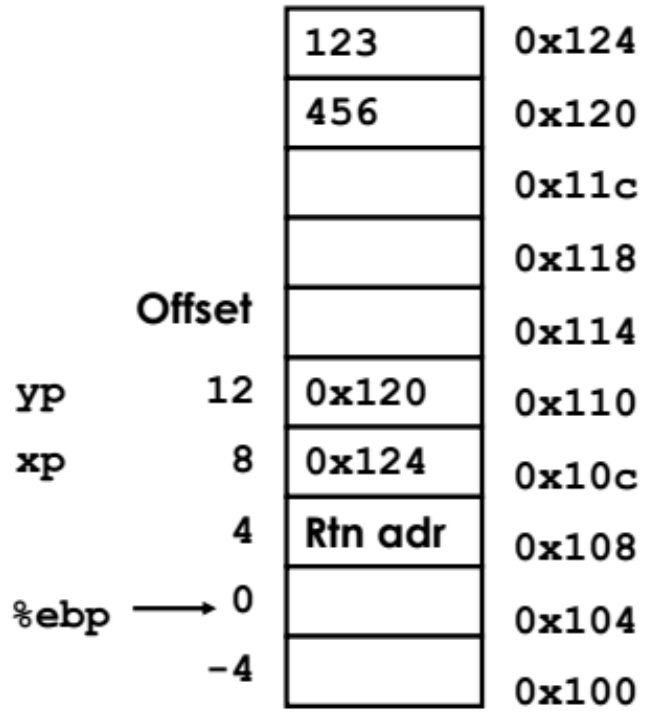
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

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

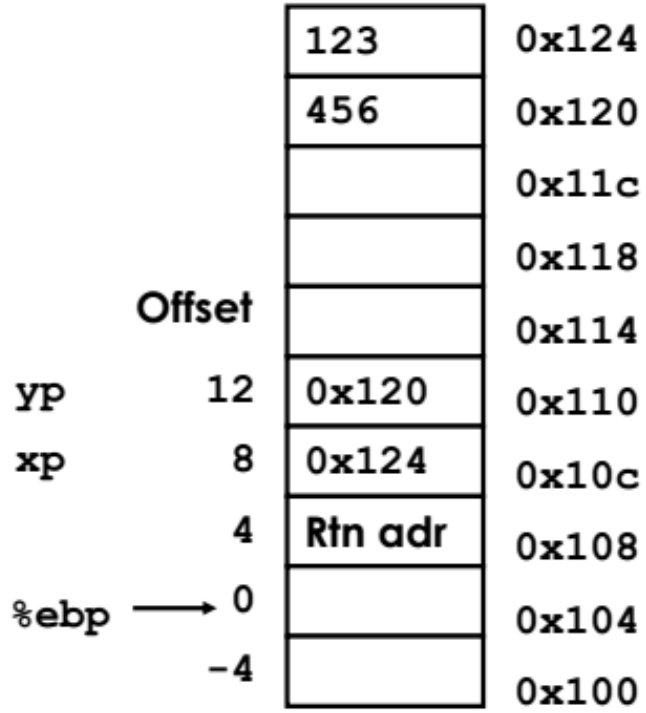
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

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

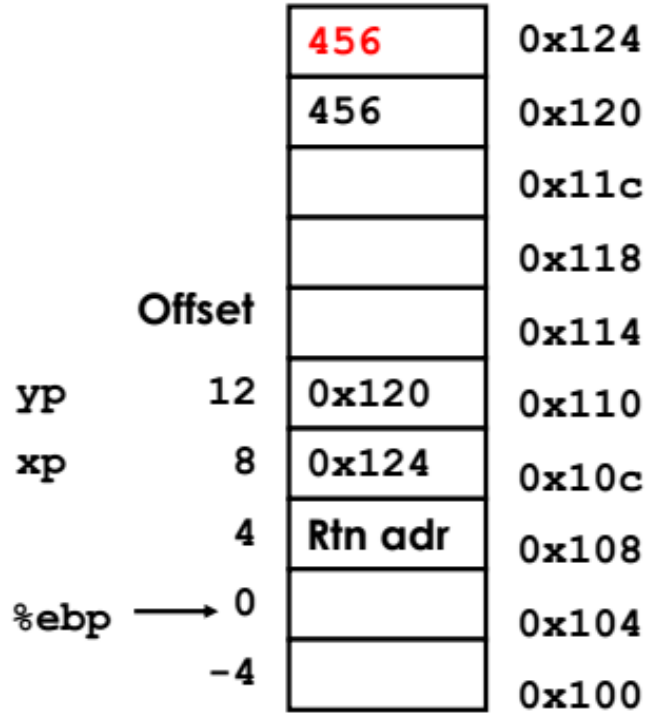
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



Example (contd.)

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

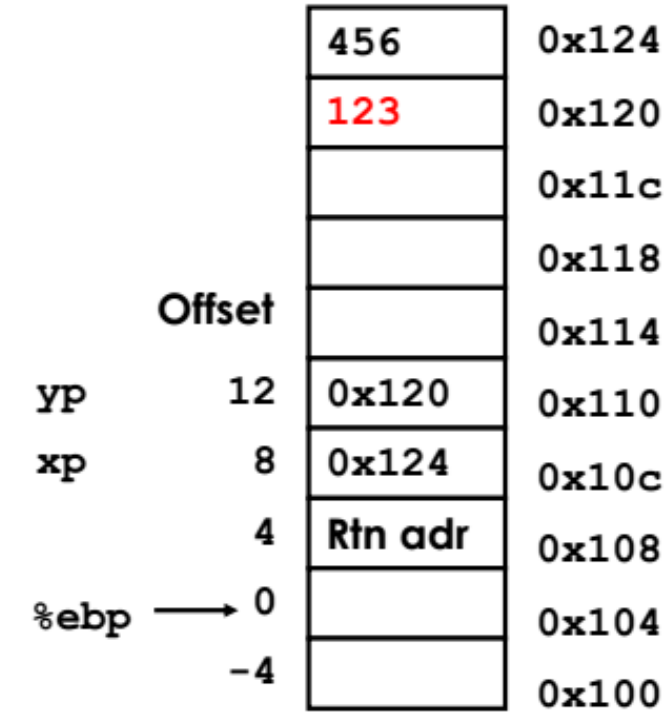
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```



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

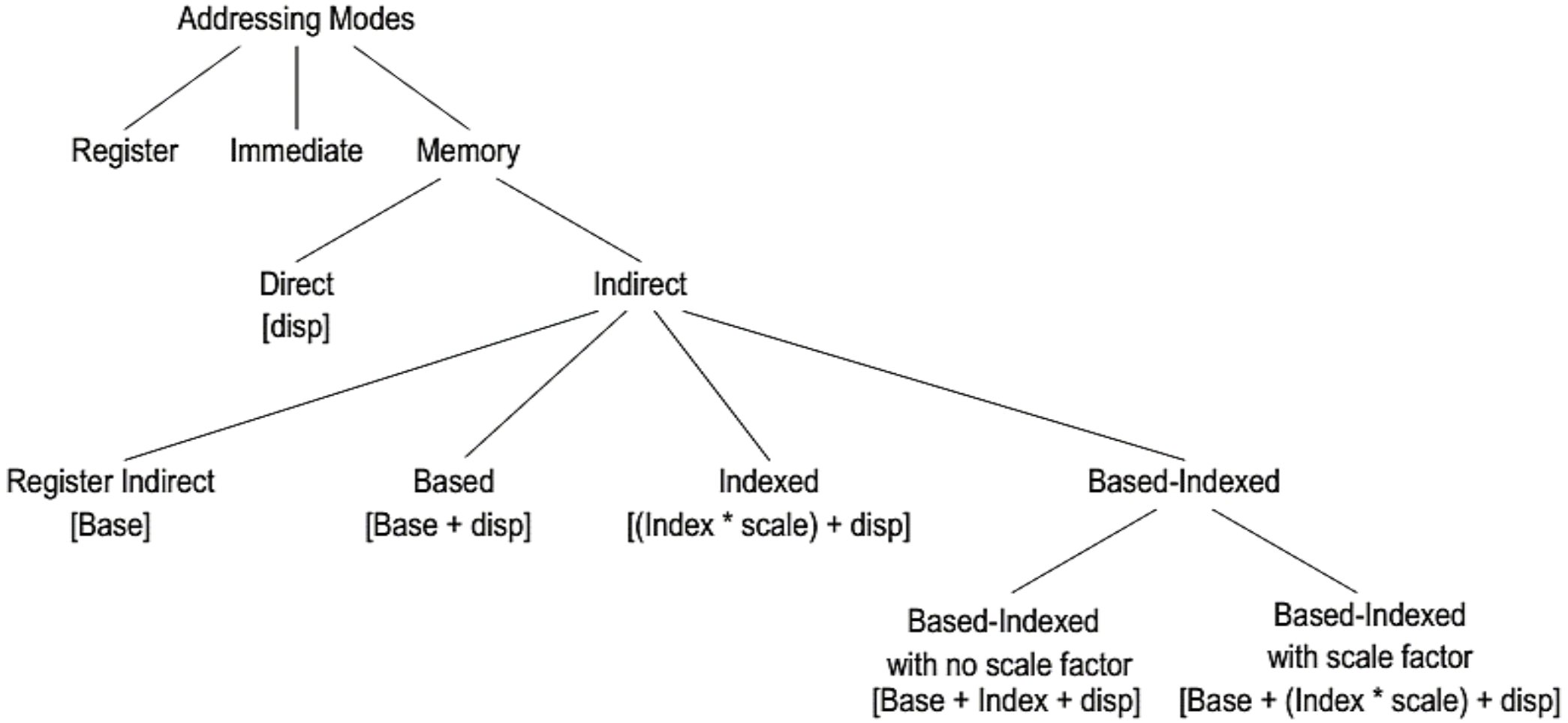
Some Integer Instructions

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*

Addressing modes (32-bits)



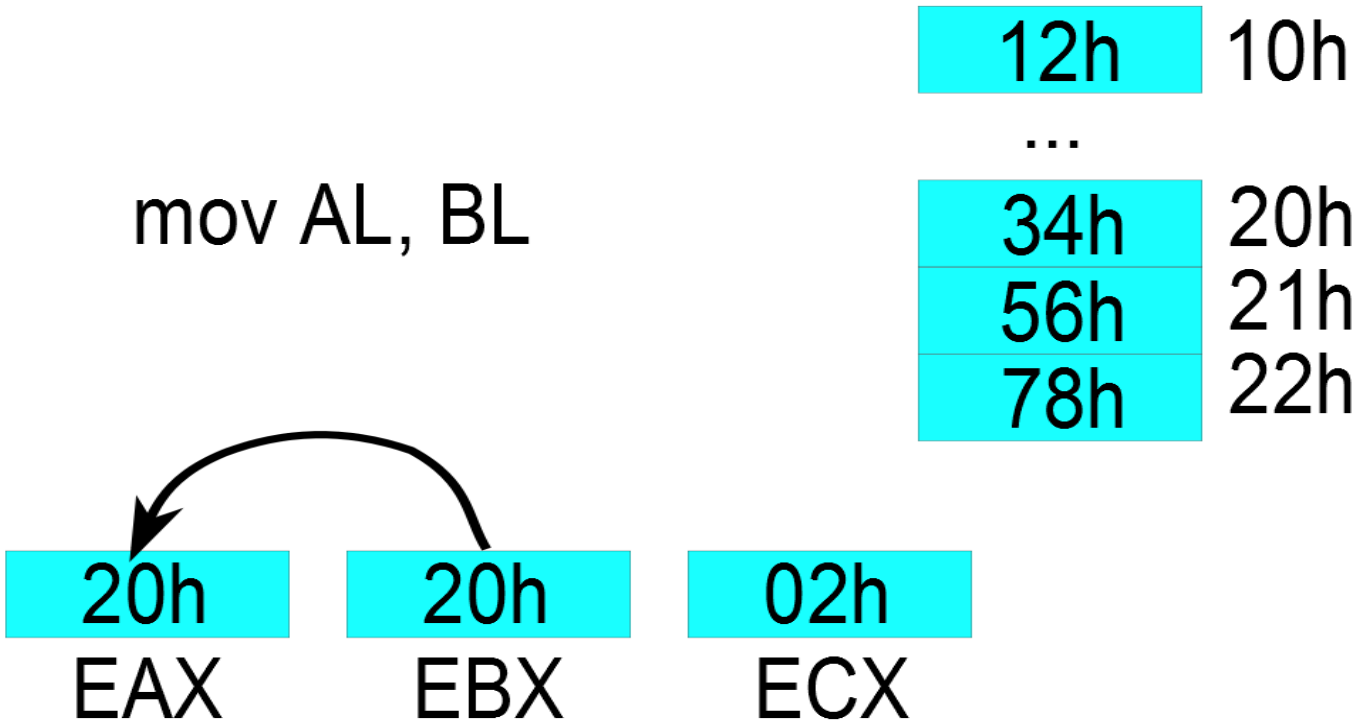
Operands in X86 (NASM)

- 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

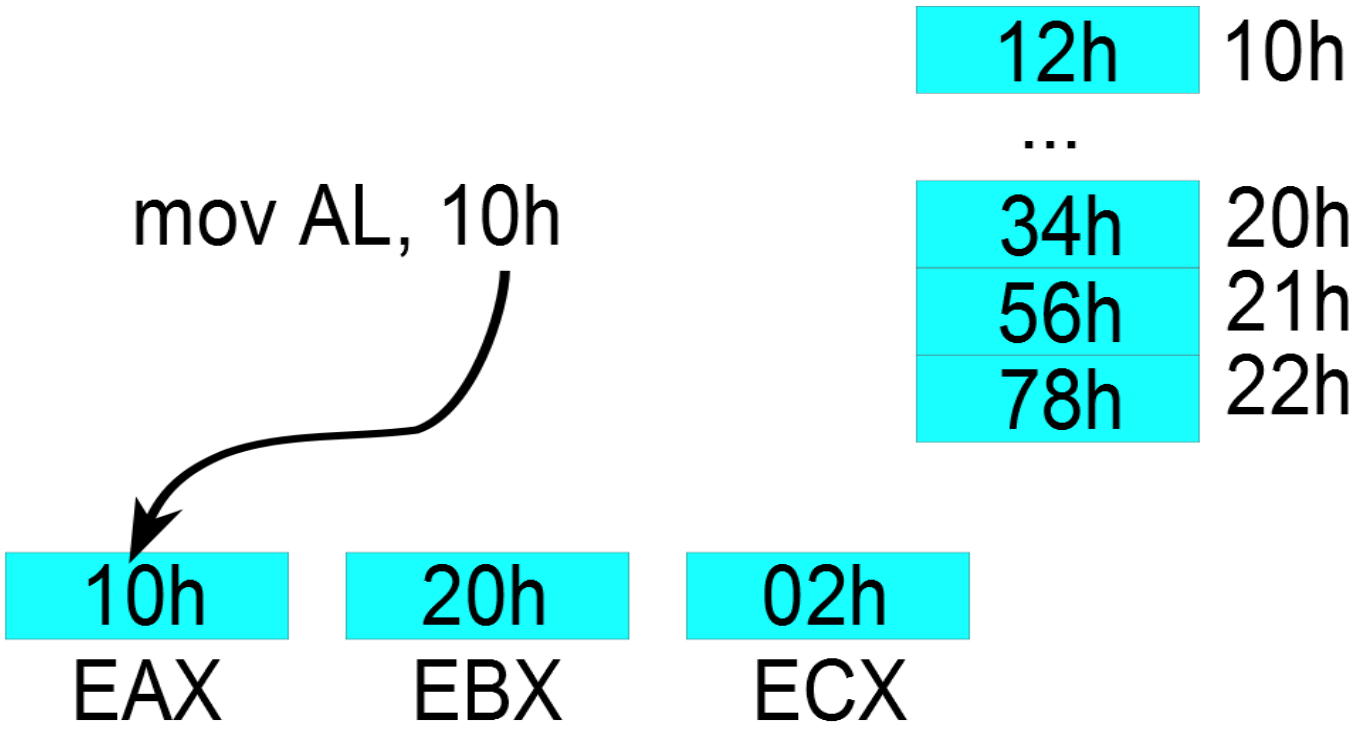
Addressing modes

- 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]**

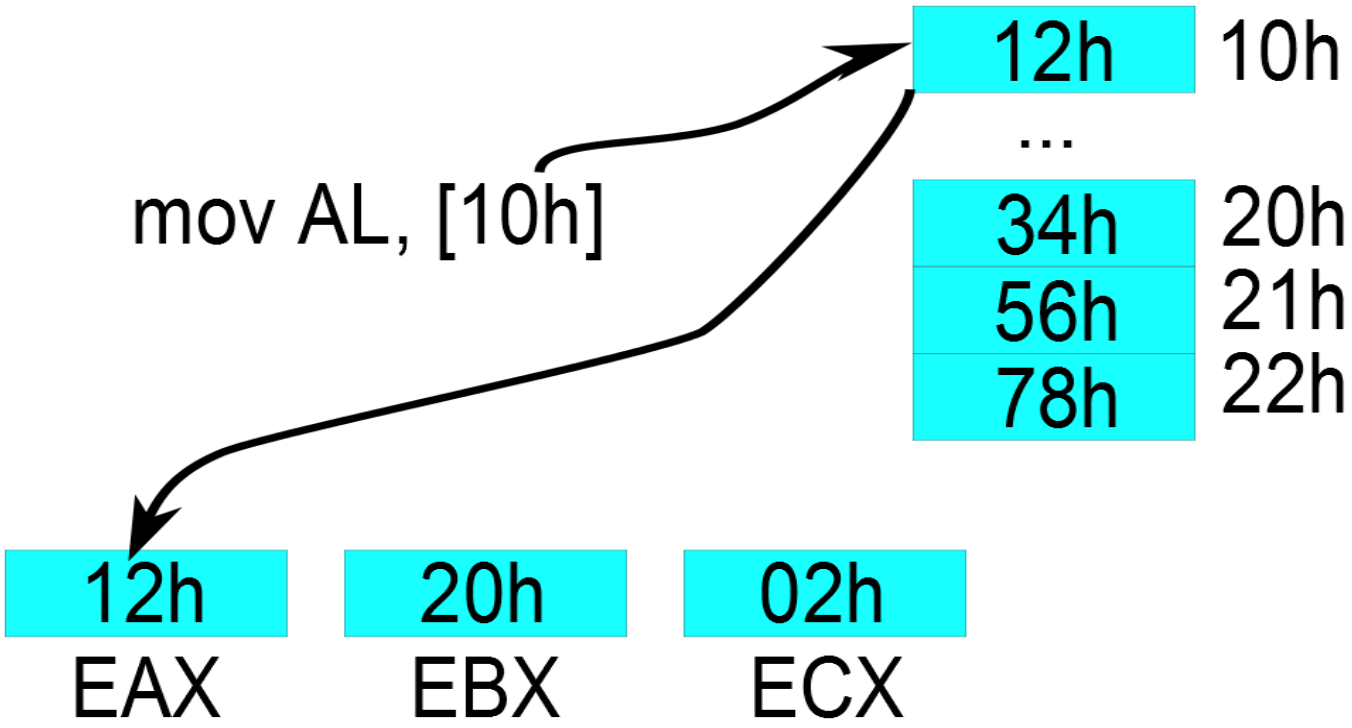
Operands and addressing modes: Register



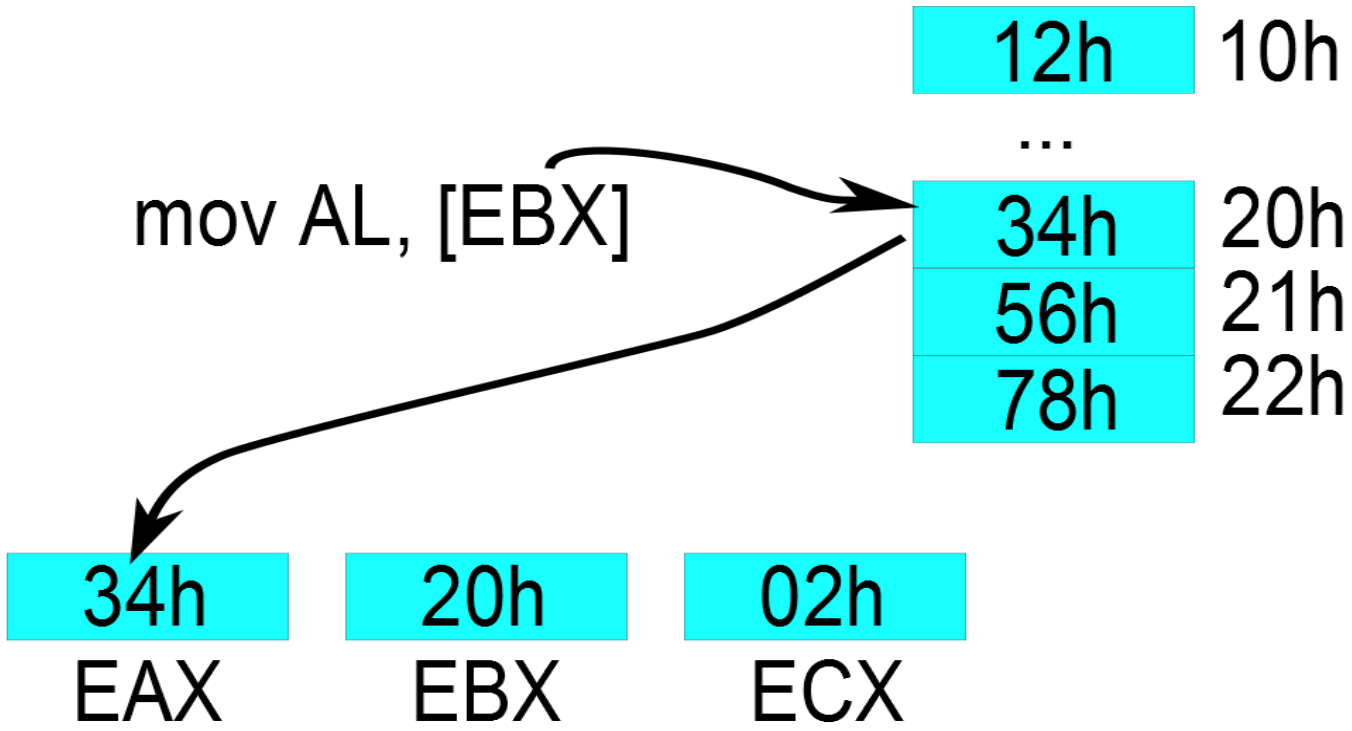
Operands and addressing modes: Immediate



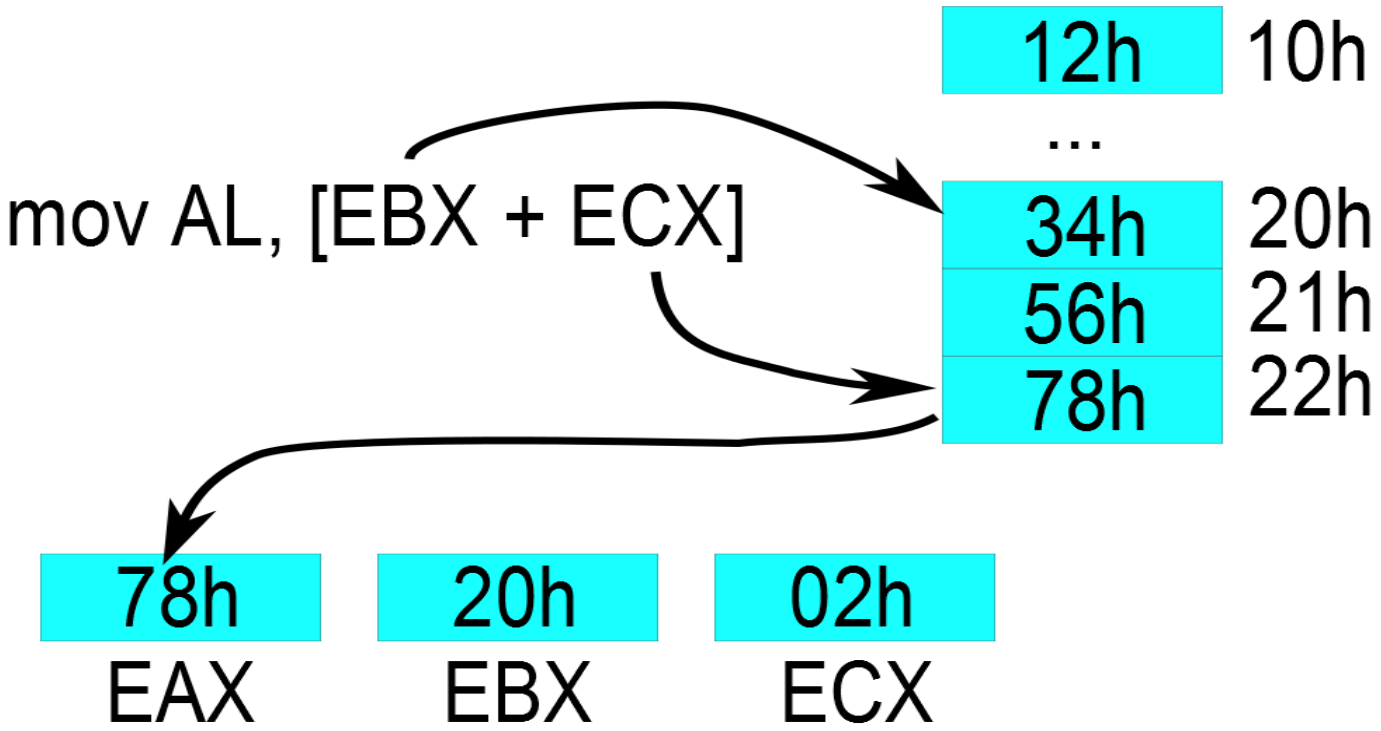
Operands and addressing modes: Direct



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?
```

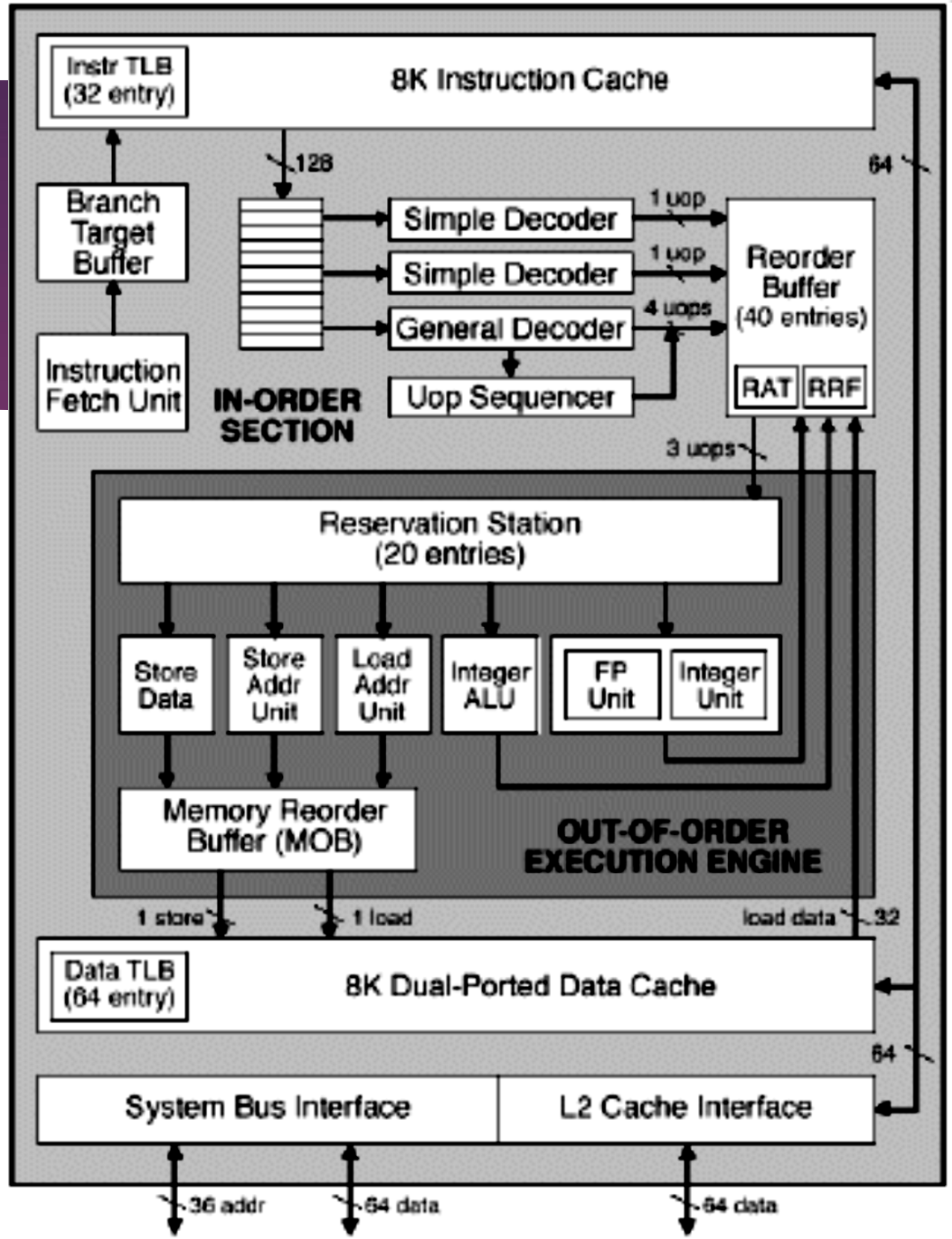
Another Program (final)

```
; 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

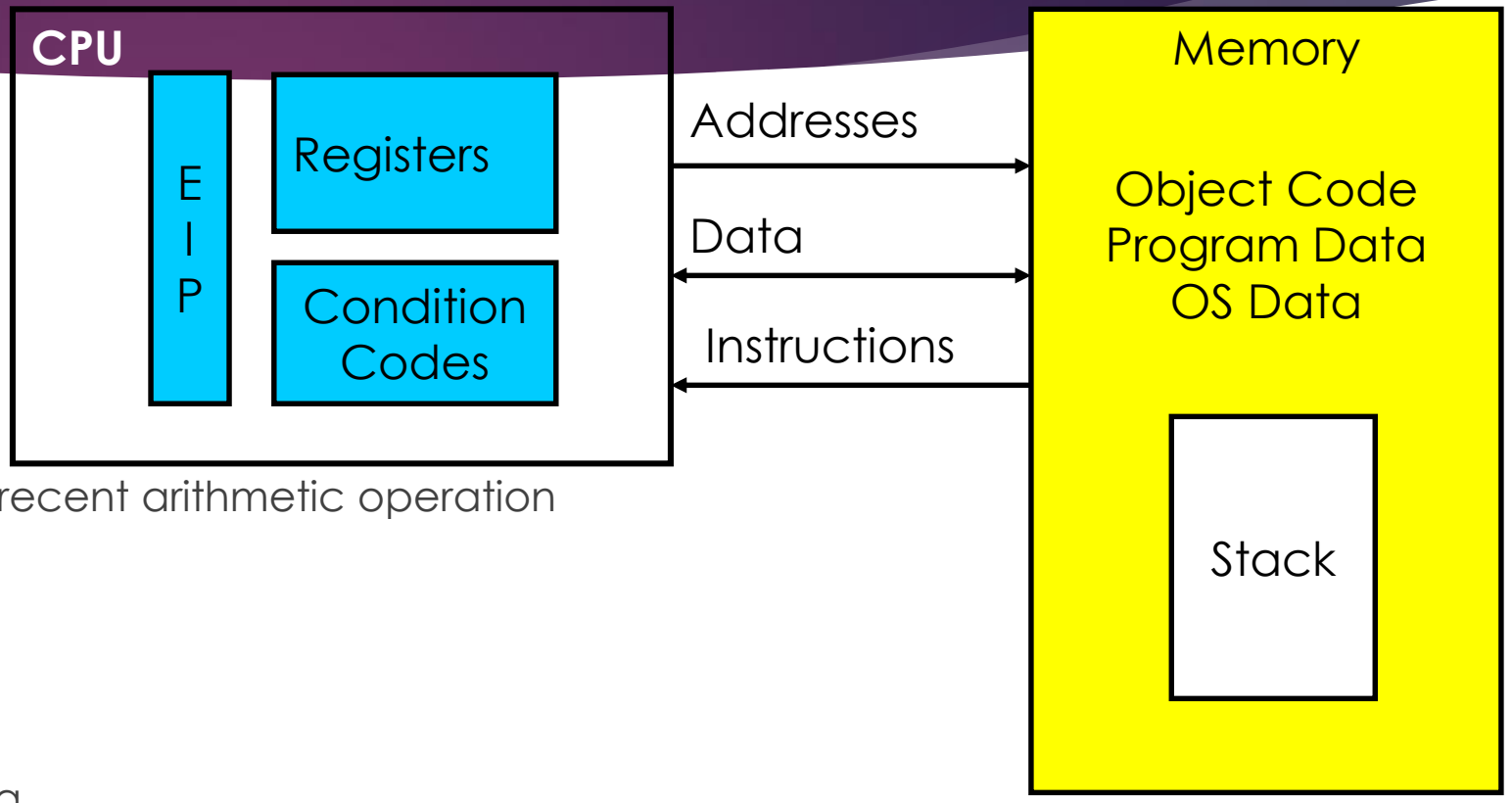
- Translates instructions dynamically into “Uops” / “µops”
  - 128 bits wide
- Holds
  - Operation 32
  - Two sources, and + 32 + 32
  - Destination + 32 = 128bits **WHY??**
- Executes Uops with **“Out of Order”** engine
  - Uop executed when
    1. Operands are available
    2. Functional unit available

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

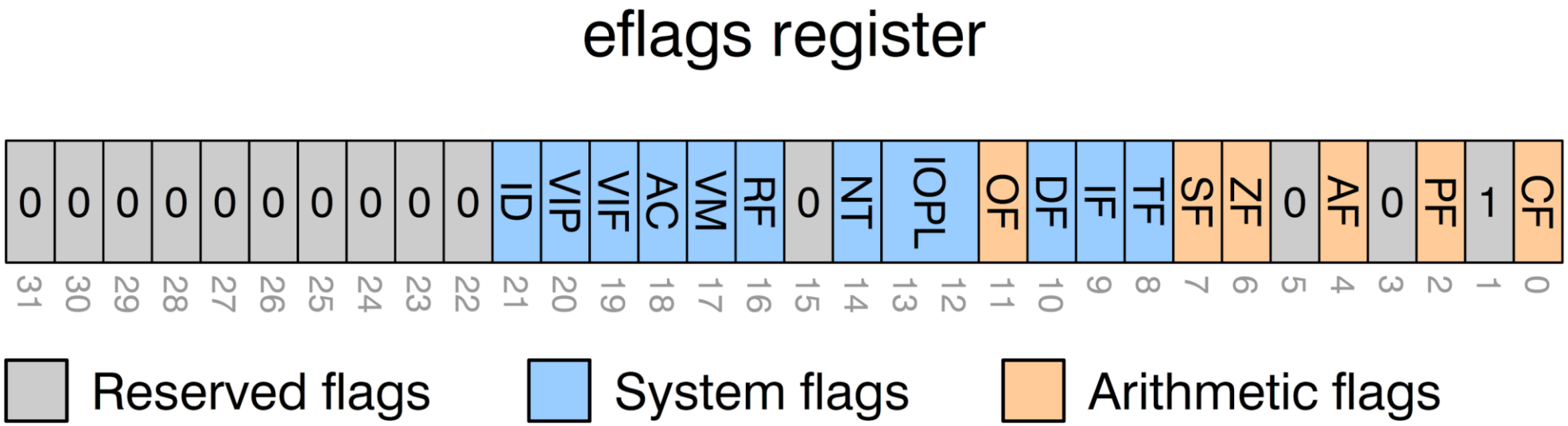
- EIP (Program Counter)
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes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



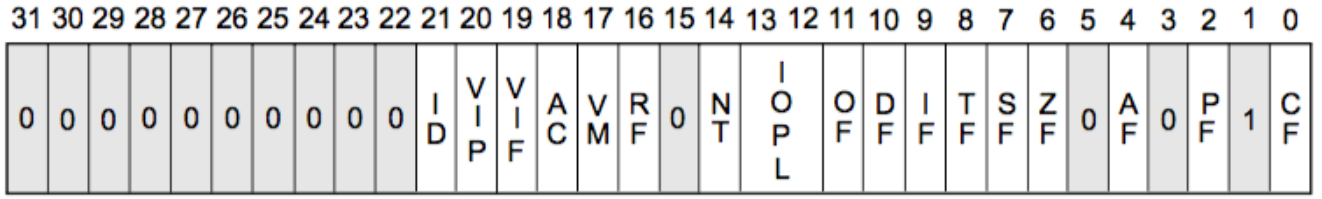
Flow Control Instructions

- JMP
- Jcc
- CALL
- RET

Revisit EFLAGS



Revisit EFLAGS



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` ← *`dest`*

- **Operation (relative jump)**:

  `EIP` ← `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>=0)`

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

Condition Codes

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

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 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
- **SF** set if `(a-b) < 0`
- **ZF** set if `a == b`
- **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

- SF= OF= ZF= 0s
- Signed operation = op1>op2
- CF= (1 – borrow)
- Unsigned operation =op1 < op2

15 – F6 => 15 + 0A

```
 0001 0101
+0000 1010
-----------
 0001 1111 = 1F
```

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: *assume 'val' is in ebx*

```
        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

```
 0111 1111
+0000 0001
-----------
 1000 0000   = 80
```

Instruction: Test

Explicit Setting by Test instruction

```
testl Src2, Src1
test  Src1, Src2
```

- Sets condition codes based on value of: ***Src1 & Src2***
  - Useful to have one of the operands be a mask
- `testl b,a`
  - like computing `a&b` without setting destination

Instruction: Test

- ***Src1***
  - AL/AX/EAX (only if **Src2** is immediate)
  - Register
  - Memory
- ***Src2***
  - Register
  - Immediate

Instruction: Test

- Modified flags (Assume **Temp = a & b**)
  - **SF = MostSignificantBit(Temp)**
  - **If (Temp == 0) ZF = 1 else ZF = 0**
  - **PF = BitWiseXorNor(Temp[Max-1:0])**
  - **CF = 0**
  - **OF = 0**
  - **AF is undefined**

Set on Condition (setcc reg8)

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	-
SETNS	Set if no sign	Sign = 0	-
SETO	Set if overflow	Overflow=1	-
SETNO	Set if no overflow	Overflow=0	-
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

Condition Code Example

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

Condition Code Example

```
int max(int x, int y)
{
  if (x > y)
    goto Flag;

  return y;

Flag:
  return x;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

"Do-While" Example

C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

"Do-While" Example

Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Registers

  %edx x

  %eax result

Assembly

```
_fact_goto:
   pushl %ebp          # Setup
   movl %esp,%ebp      # Setup
   movl $1,%eax        # eax = 1
   movl 8(%ebp),%edx   # edx = x

L11:
   imull %edx,%eax     # result *= x
   decl %edx           # x--
   cmpl $1,%edx        # Compare x : 1
   jg L11              # if > goto loop

   movl %ebp,%esp      # Finish
   popl %ebp           # Finish
   ret                 # Finish
```

Multiplication

- **mul** arg Unsigned multiplication
  - This multiplies **"arg" by** the value of corresponding byte-length in the **EAX register**.

operand size	1 byte	2 bytes	4 bytes
other operand	AL	AX	EAX
higher part of result stored in:	AH	DX	EDX
lower part of result stored in:	AL	AX	EAX

- Signed Multi[plication: **imul** arg

Division

- Unsigned: **div** arg
  - This divides the value in the dividend register(s) by "arg", see table below

divisor size:	1 byte	2 bytes	4 bytes
dividend	AX	DX:AX	EDX:EAX
remainder stored in:	AH	DX	EDX
quotient stored in:	AL	AX	EAX

- Signed: **idiv** arg

Sign extended Register

- CDQ: Convert Double to Quad-Word (Intel)
- CLTD: AT&T

- Sign-extends EAX into EDX
- Forms the quad-word **EDX:EAX**.
- Since (I)DIV uses EDX:EAX as its input, CDQ must be called after setting EAX if EDX is not manually initialized (as in 64/32 division) before (I)DIV.

Load Effective Address

- AT&T: **leal** src, dest
- Intel: **lea** dest, src

- Calculates the **address** of the **src** operand and **loads** it into the **dest** operand
- ***Src:*** Immediate / Register / Memory
- ***Dest:*** Register

- No FLAGS are modified by this instruction

Load Effective Address

- Powerful operation
- Can also be used for calculating general-purpose unsigned integer arithmetic

- SRC can use **General Addressing Mode**
- **[eax + edx*4 -4] (Intel syntax)**
  - **or**
- **-4(%eax, %edx, 4) (GAS syntax)**

Load Effective Address Example

- Suppose you want to multiply the content of **EBX** with **18**
- LEA can be used as follows

```
lea ebx, [ebx*2]      ; Multiply ebx by 2
lea ebx, [ebx*8+ebx]  ; Multiply ebx by 9, which totals ebx*18
```

Jump Table

- To implement a "**switch-case**".
- To select a **function** to be evoked.

- We will construct a **array** of the **jump addresses** .
- For each number will jump to the **corresponding entry** in the jump table

Example

- Main.c

```
extern void jumper(int);
int main (int argc , char* argv)
{
    jumper (0);
    jumper (1);
    jumper (2);
    return 0;
}
```

jump.s

```
section .data
jt:
        dd      label_1
        dd      label_2

str0:   db      "Got the number 0",10,0
str1:   db      "Got the number 1",10,0
str2:   db      "Out of bound",10,0
str3:   db      "num = %d",10,0
section .text
        align   16
        global  jumper
        extern  printf
```

jump.s

```
jumper:
        push    ebp
        mov     ebp,esp
        pusha
        mov     ebx,dword [ebp+8]
        push    ebx
        push    str3
        call    printf                  ; Print num
        add     esp, 8
        cmp     ebx,0                   ; Check if num is in bounds
        jb      out_of
        cmp     ebx ,1
        ja      out_of
        shl     ebx,2                   ; num = num * 4
        jmp     dword [ebx+jt]          ; Jump according to address in table
```

jump.s

```
label_1:
        push    str0
        call    printf
        add     esp, 4
        jmp     end
label_2:
        push    str1
        call    printf
        add     esp, 4
        jmp     end
out_of:
        push    str2
        call    printf
        add     esp, 4
        jmp     end

end:
        popa
        pop     ebp
        ret
```

Output

num = 0

Got the number 0

num = 1

Got the number 1

num = 2

Out of bound

jump.s

```
section .data
jt:
        dd      label_1
        dd      label_2

str0:   db      "Got the number 0",10,0
str1:   db      "Got the number 1",10,0
str2:   db      "Out of bound",10,0
str3:   db      "num = %d",10,0
section .text
        align   16
        global  jumper
        extern  printf
```

jump.s

```
jumper:
        push    ebp
        mov     ebp,esp
        pusha
        mov     ebx,dword [ebp+8]
        push    ebx
        push    str3
        call    printf                  ; Print num
        add     esp, 8
        cmp     ebx,0                   ; Check if num is in bounds
        jb      out_of
        cmp     ebx ,1
        ja      out_of
        shl     ebx,2                   ; num = num * 4
        jmp     dword [ebx+jt]          ; Jump according to address in table
```

jump.s

```
label_1:
        push    str0
        call    printf
        add     esp, 4
        jmp     end
label_2:
        push    str1
        call    printf
        add     esp, 4
        jmp     end
out_of:
        push    str2
        call    printf
        add     esp, 4
        jmp     end

end:
        popa
        pop     ebp
        ret
```

Switch Statement Example (IA32)

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx          # x:6
   ja    .L61              # if > goto default
   jmp   *.L62(,%edx,4)    # goto JTab[x]
```

Jump Table

Table Contents

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
case 5:
case 6:       // .L60
    w -= z;
    break;
default:      // .L61
    w = 2;
}
```

Code Blocks (Partial)


```
switch(x) {
  . . .
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
  . . .
default:      // .L61
    w = 2;
}
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Rest)

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:       // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
   subl   %ecx, %ebx   # w -= z
   movl   %ebx, %eax   # Return w
   popl   %ebx
   leave
   ret
.L56: // Case 1:
   movl   12(%ebp), %ebx # w = y
   imull  %ecx, %ebx     # w*= z
   movl   %ebx, %eax   # Return w
   popl   %ebx
   leave
   ret
```