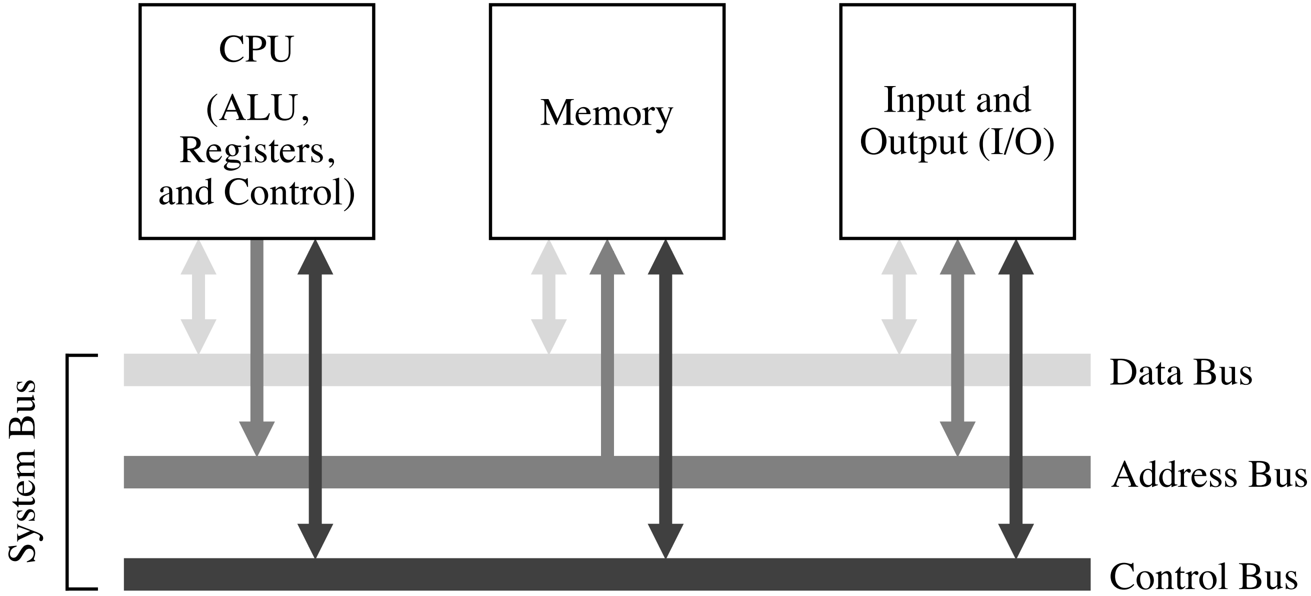
The System Bus Model

- **A refinement of the von Neumann model, the system bus model has a CPU (ALU and control), memory, and an input/output unit.**
- **Communication among components is handled by a shared pathway called the *system bus*, which is made up of the data bus, the address bus, and the control bus. There is also a power bus, and some architectures may also have a separate I/O bus.**



The Fetch-Execute Cycle

- **The steps that the control unit carries out in executing a program are:**

  (1) Fetch the next instruction to be executed from memory.

  (2) Decode the opcode.

  (3) Read operand(s) from main memory, if any.

  (4) Execute the instruction and store results.

  (5) Go to step 1.

This is known as the *fetch-execute cycle*.

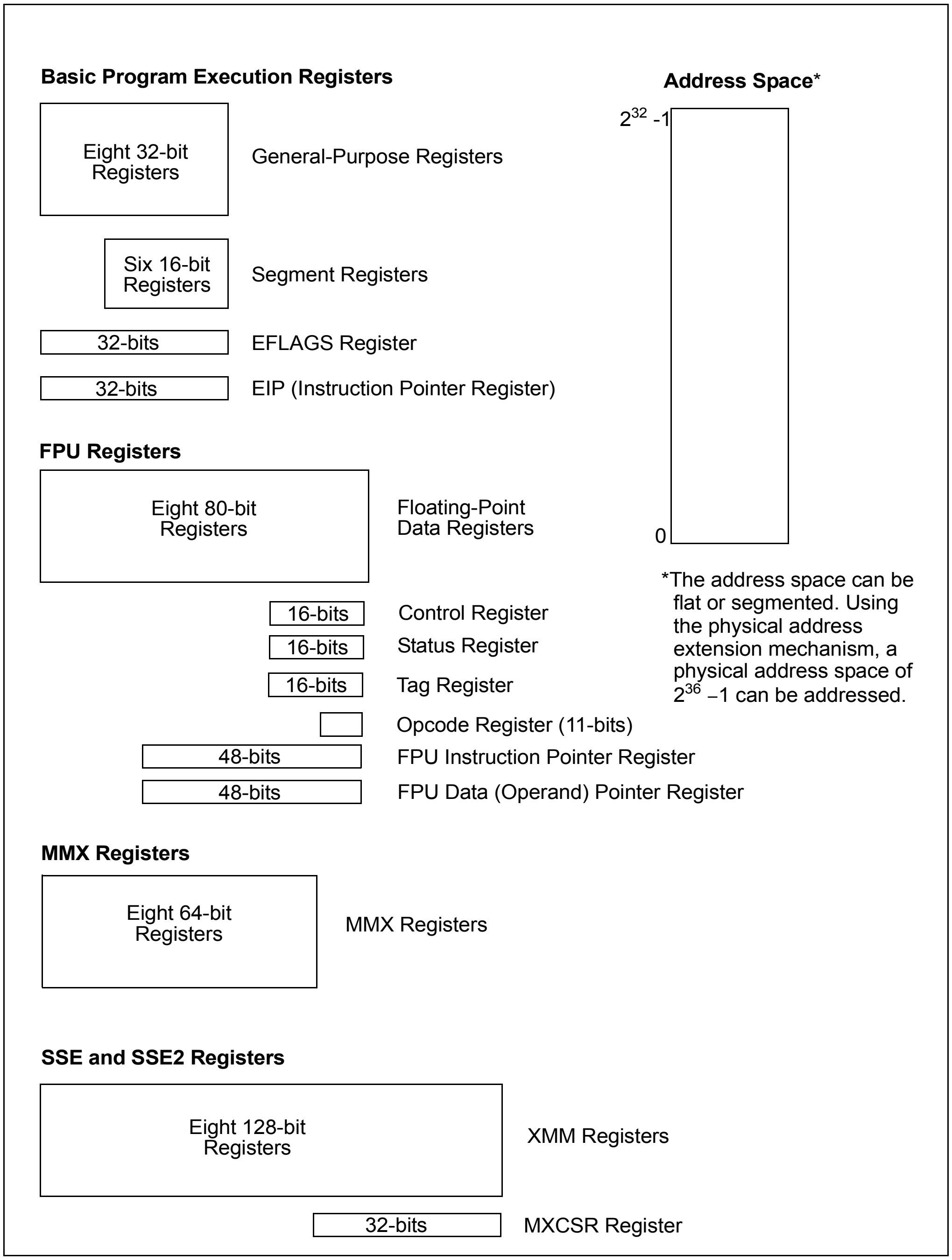


Figure 3-1. IA-32 Basic Execution Environment

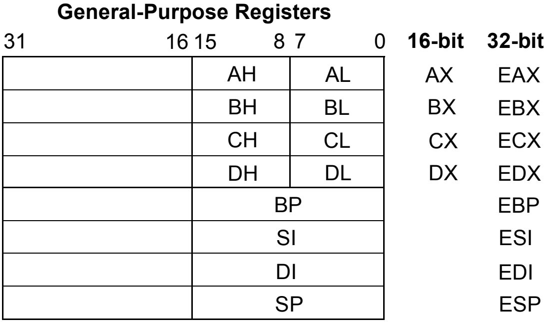


Figure 3-4. Alternate General-Purpose Register Names

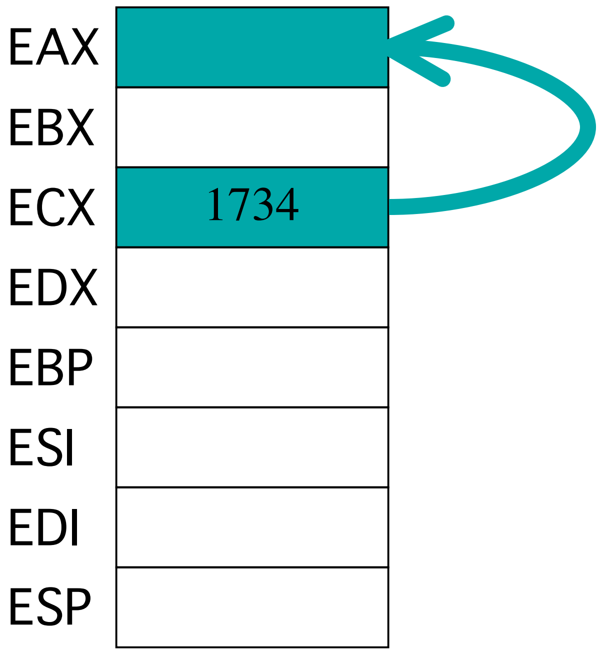
- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.
- ECX—Counter for string and loop operations.
- EDX—I/O pointer.
- ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.9
- EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- ESP—Stack pointer (in the SS segment).
- EBP—Pointer to data on the stack (in the SS segment).

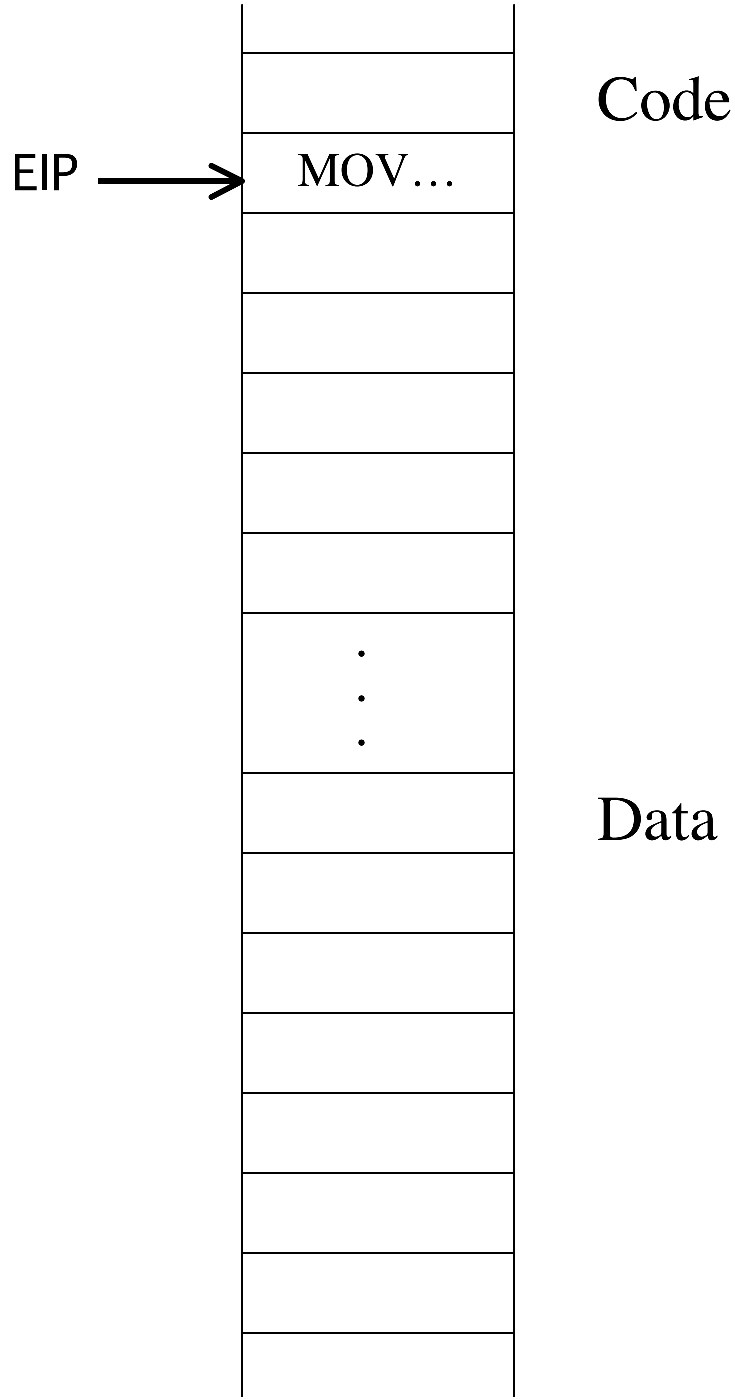
80x86 Addressing Modes

- We want to store the value 1734h.
- The value 1734h may be located in a register or in memory.
- The location in memory might be specified by the code, by a register, …
- Assembly language syntax for MOV

```
MOV        DEST,  SOURCE
```

Addressing Modes

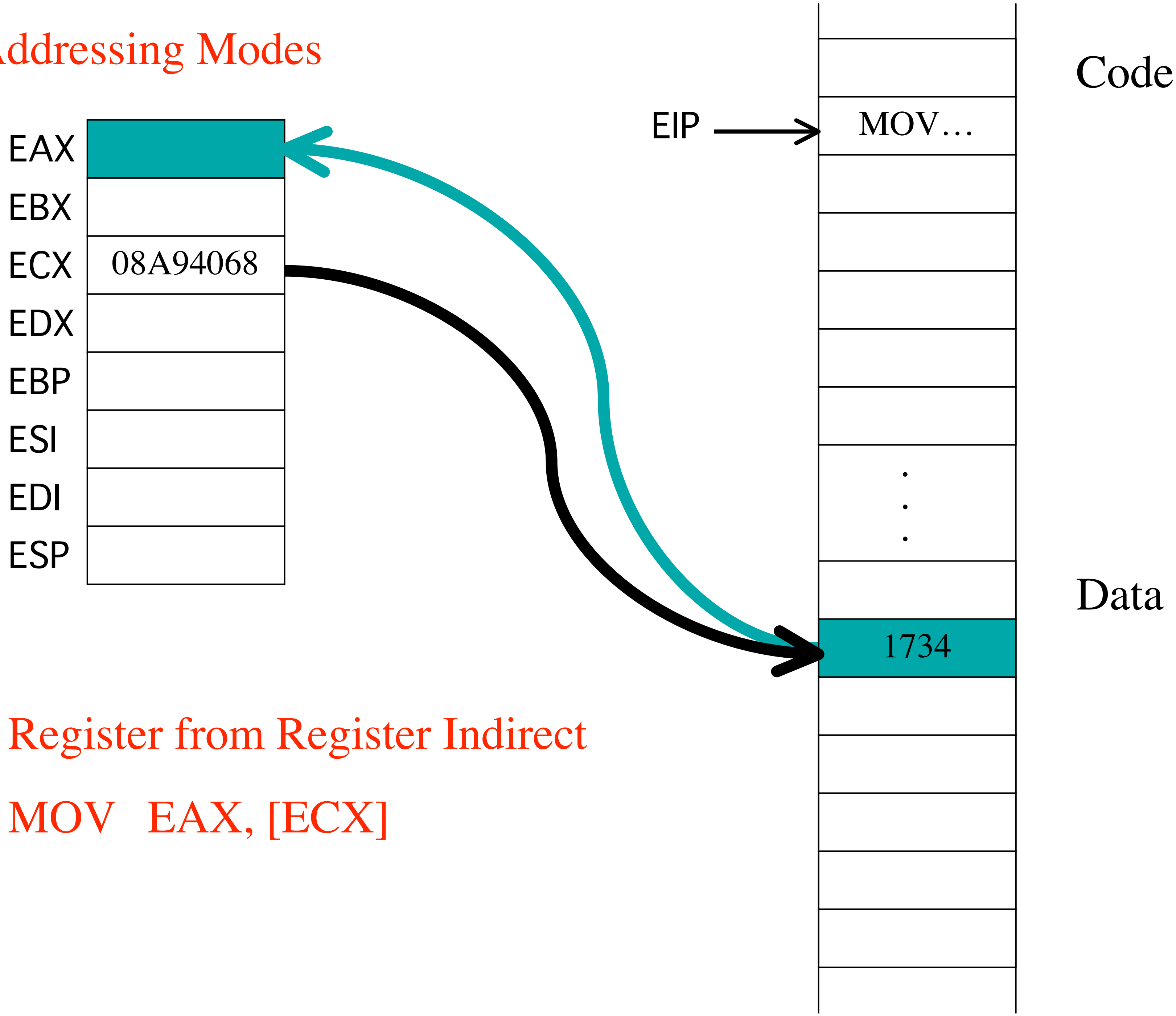




Register from Register

MOV   EAX, ECX

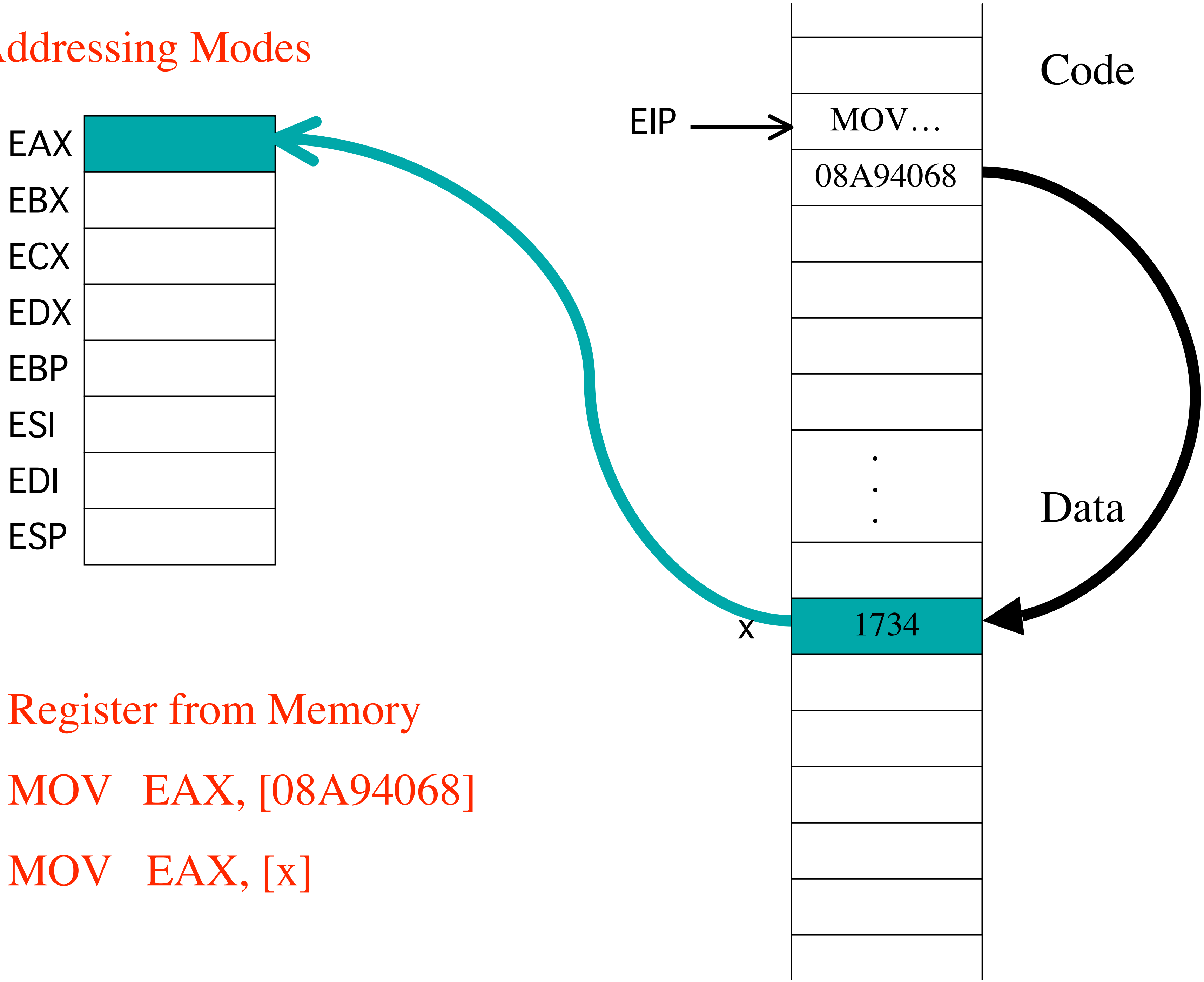
Addressing Modes



Register from Register Indirect

MOV EAX, [ECX]

Addressing Modes

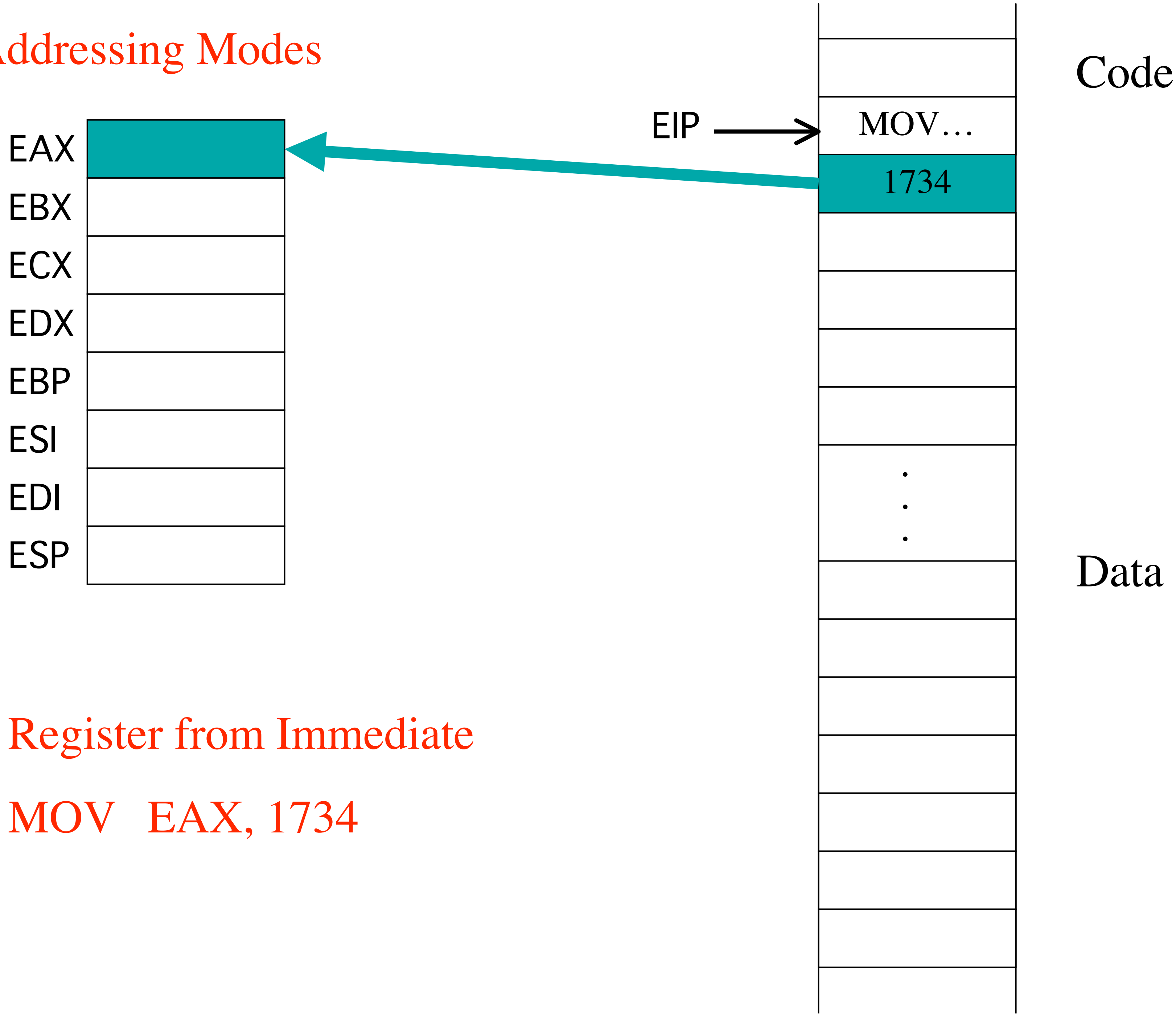


Register from Memory

MOV EAX, [08A94068]

MOV EAX, [x]

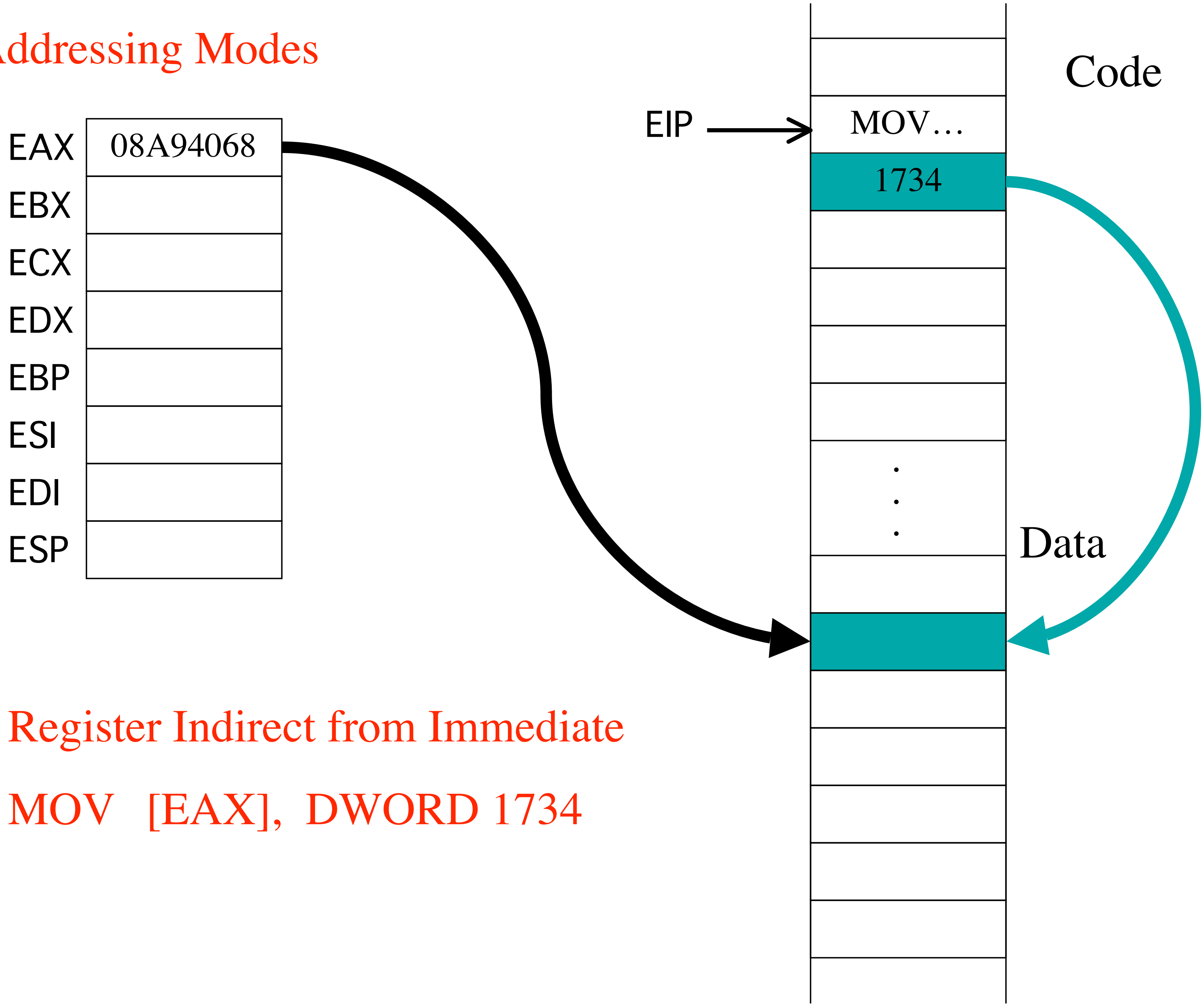
Addressing Modes



Register from Immediate

MOV EAX, 1734

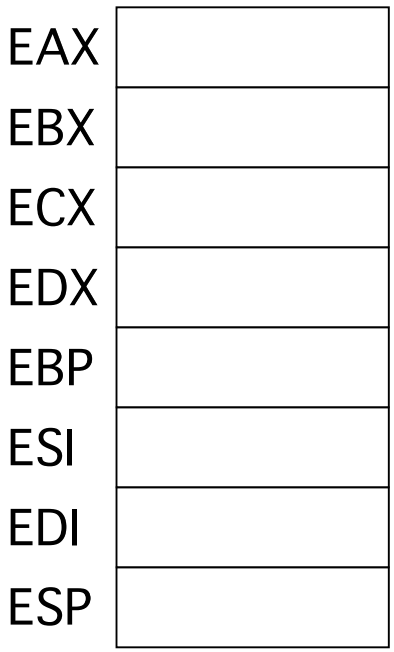
Addressing Modes

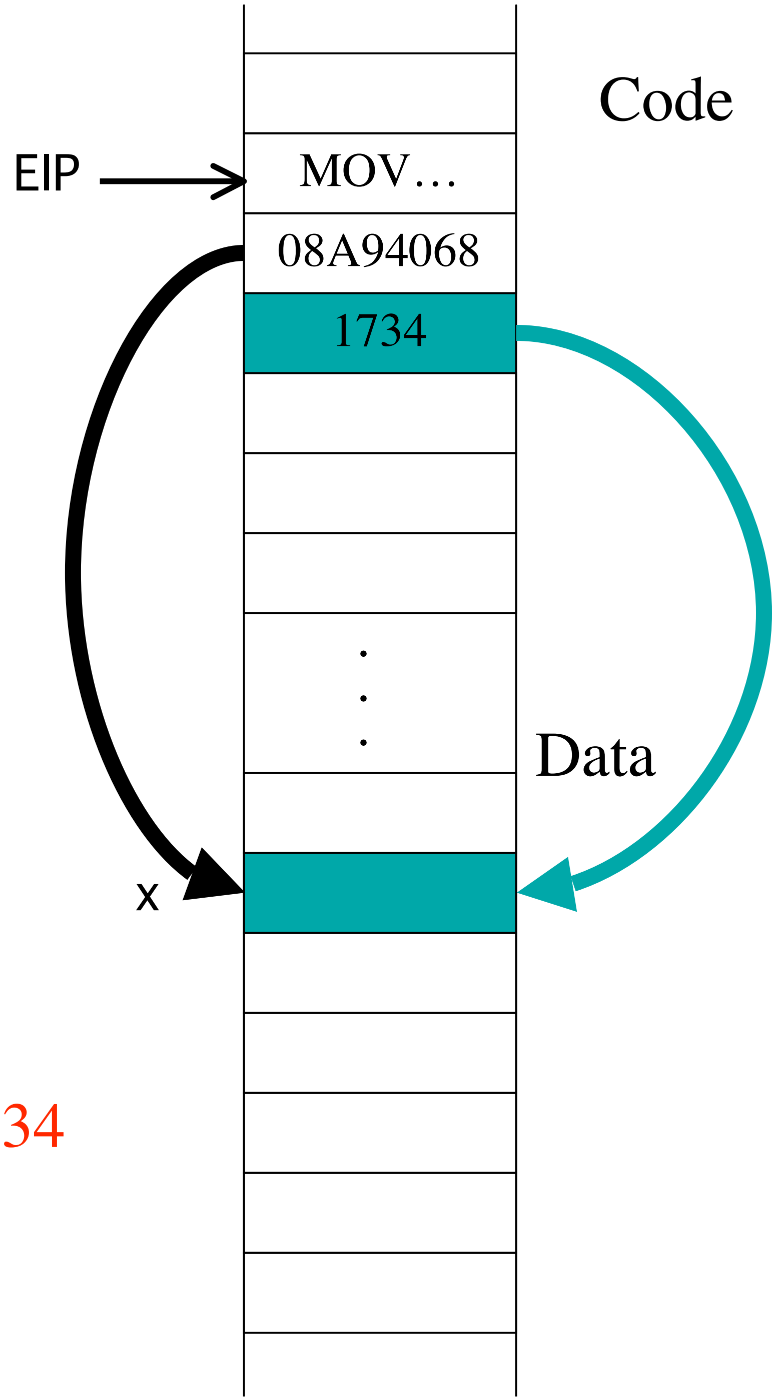


Register Indirect from Immediate

MOV [EAX], DWORD 1734

Addressing Modes





Memory from Immediate

MOV [08A94068], DWORD 1734

MOV [x], DWORD 1734

Notes on Addressing Modes

- More complicated addressing modes later:

  MOV EAX, [ESI+4*ECX+12]

- Figures not drawn to scale. Constants 1734h and 08A94068h take 4 bytes (little endian).
- Some addressing modes are not supported by some operations.
- Labels represent addresses not contents of memory.

Recap i386 Basic Architecture

- Registers are storage units inside the CPU.
- Registers are much faster than memory.
- 8 General purpose registers in i386:
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
  - subparts of EAX, EBX, ECX and EDX have special names
- The instruction pointer (EIP) points to machine code to be executed.
- Typically, data moves from memory to registers, processed, moves from registers back to memory.
- Different addressing modes used.

toupper.asm

- Prompt for user input.
- Use Linux system call to get user input.
- Scan each character of user input and convert all lower case characters to upper case.
- How to:
  - work with 8-bit data
  - specify ASCII constant
  - compare values
  - loop control

```
; File: toupper.asm last updated 09/26/2001
;
; Convert user input to upper case.
;
; Assemble using NASM:  nasm -f elf toupper.asm
; Link with ld:  ld toupper.o
;

%define STDIN 0
%define STDOUT 1
%define SYSCALL_EXIT  1
%define SYSCALL_READ  3
%define SYSCALL_WRITE 4
%define BUFLEN 256


        SECTION .data                   ; initialized data section

msg1:   db "Enter string: "             ; user prompt
len1:   equ $-msg1                      ; length of first message

msg2:   db "Original: "                 ; original string label
len2:   equ $-msg2                      ; length of second message

msg3:   db "Convert:  "                 ; converted string label
len3:   equ $-msg3

msg4:   db 10, "Read error", 10         ; error message
len4:   equ $-msg4


        SECTION .bss                    ; uninitialized data section
buf:    resb BUFLEN                     ; buffer for read
newstr: resb BUFLEN                     ; converted string
rlen:   resb 4                          ; length


        SECTION .text                   ; Code section.
        global  _start                  ; let loader see entry point

_start: nop                             ; Entry point.
start:                                  ; address for gdb

        ; prompt user for input
        ;
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, msg1               ; Arg2: addr of message
        mov     edx, len1               ; Arg3: length of message
        int     080h                    ; ask kernel to write

```

```
        ; read user input
        ;
        mov     eax, SYSCALL_READ       ; read function
        mov     ebx, STDIN              ; Arg 1: file descriptor
        mov     ecx, buf                ; Arg 2: address of buffer
        mov     edx, BUFLEN             ; Arg 3: buffer length
        int     080h

        ; error check
        ;
        mov     [rlen], eax             ; save length of string read
        cmp     eax, 0                  ; check if any chars read
        jg      read_OK                 ; >0 chars read = OK
        mov     eax, SYSCALL_WRITE      ; ow print error mesg
        mov     ebx, STDOUT
        mov     ecx, msg4
        mov     edx, len4
        int     080h
        jmp     exit                    ; skip over rest
read_OK:


        ; Loop for upper case conversion
        ; assuming rlen > 0
        ;
L1_init:
        mov     ecx, [rlen]             ; initialize count
        mov     esi, buf                ; point to start of buffer
        mov     edi, newstr             ; point to start of new string

L1_top:
        mov     al, [esi]               ; get a character
        inc     esi                     ; update source pointer
        cmp     al, 'a'                 ; less than 'a'?
        jb      L1_cont
        cmp     al, 'z'                 ; more than 'z'?
        ja      L1_cont
        and     al, 11011111b           ; convert to uppercase

L1_cont:
        mov     [edi], al               ; store char in new string
        inc     edi                     ; update dest pointer
        dec     ecx                     ; update char count
        jnz     L1_top                  ; loop to top if more chars
L1_end:
```

```
        ; print out user input for feedback
        ;
        mov     eax, SYSCALL_WRITE      ; write message
        mov     ebx, STDOUT
        mov     ecx, msg2
        mov     edx, len2
        int     080h

        mov     eax, SYSCALL_WRITE      ; write user input
        mov     ebx, STDOUT
        mov     ecx, buf
        mov     edx, [rlen]
        int     080h

        ; print out converted string
        ;
        mov     EAX, SYSCALL_WRITE      ; write message
        mov     EBX, STDOUT
        mov     ECX, msg3
        mov     EDX, len3
        int     080h

        mov     EAX, SYSCALL_WRITE      ; write out string
        mov     EBX, STDOUT
        mov     ECX, newstr
        mov     EDX, [rlen]
        int     080h


        ; final exit
        ;
exit:   mov     EAX, SYSCALL_EXIT       ; exit function
        mov     EBX, 0                  ; exit code, 0=normal
        int     080h                    ; ask kernel to take over
```

i386 Instruction Set Overview

- General Purpose Instructions
  - works with data in the general purpose registers
- Floating Point Instructions
  - floating point arithmetic
  - data stored in separate floating point registers
- Single Instruction Multiple Data (SIMD) Extensions
  - MMX, SSE, SSE2
- System Instructions
  - Sets up control registers at boot time

Common Instructions

- **Basic Instructions**
  - ADD, SUB, INC, DEC, MOV, NOP
- **Branching Instructions**
  - JMP, CMP, J*cc*
- **More Arithmetic Instructions**
  - NEG, MUL, IMUL, DIV, IDIV
- **Logical (bit manipulation) Instructions**
  - AND, OR, NOT, SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR
- **Subroutine Instructions**
  - PUSH, POP, CALL, RET

RISC vs CISC

- CISC = Complex Instruction Set Computer
  - ◇ Pro: instructions closer to constructs in higher-level languages
  - ◇ Con: complex instructions used infrequently
- RISC = Reduced Instruction Set Computer
  - ◇ Pro: simpler instructions allow design efficiencies (e.g., pipelining)
  - ◇ Con: more instructions needed to achieve same task

Read The Friendly Manual (RTFM)

- Best Source: Intel Instruction Set Reference
  - Available off the course web page in PDF.
  - Download it, you'll need it.
- Next Best Source: Appendix A NASM Doc.
- Questions to ask:
  - What is the instruction's basic function? (e.g., adds two numbers)
  - Which addressing modes are supported? (e.g., register to register)
  - What side effects does the instruction have? (e.g. OF modified)

ADD—Add

Opcode	Instruction	Description
04 *ib*	ADD AL,*imm8*	Add *imm8* to AL
05 *iw*	ADD AX,*imm16*	Add *imm16* to AX
05 *id*	ADD EAX,*imm32*	Add *imm32* to EAX
80 /0 *ib*	ADD *r/m8,imm8*	Add *imm8* to *r/m8*
81 /0 *iw*	ADD *r/m16,imm16*	Add *imm16* to *r/m16*
81 /0 *id*	ADD *r/m32,imm32*	Add *imm32* to *r/m32*
83 /0 *ib*	ADD *r/m16,imm8*	Add sign-extended *imm8* to *r/m16*
83 /0 *ib*	ADD *r/m32,imm8*	Add sign-extended *imm8* to *r/m32*
00 /*r*	ADD *r/m8,r8*	Add *r8* to *r/m8*
01 /*r*	ADD *r/m16,r16*	Add *r16* to *r/m16*
01 /*r*	ADD *r/m32,r32*	Add r32 to *r/m32*
02 /*r*	ADD *r8,r/m8*	Add *r/m8* to *r8*
03 /*r*	ADD *r16,r/m16*	Add *r/m16* to *r16*
03 /*r*	ADD *r32,r/m32*	Add *r/m32* to *r32*

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

DEST ← DEST + SRC;

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Intel Manual's Addressing Mode Notation

- **r8:** One of the 8-bit registers AL, CL, DL, BL, AH, CH, DH, or BH.
- **r16:** One of the 16-bit registers AX, CX, DX, BX, SP, BP, SI, or DI.
- **r32:** One of the 32-bit registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.
- **imm8:** An immediate 8-bit value.
- **imm16:** An immediate 16-bit value.
- **imm32:** An immediate 32-bit value.
- **r/m8:** An 8-bit operand that is either the contents of an 8-bit register (AL, BL, CL, DL, AH, BH, CH, and DH), or a byte from memory.
- **r/m16:** A 16-bit register (AX, BX, CX, DX, SP, BP, SI, and DI) or memory operand used for instructions whose operand-size attribute is 16 bits.
- **r/m32:** A 32-bit register (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI) or memory operand used for instructions whose operand-size attribute is 32 bits.

The EFLAGS Register

- A special 32-bit register that contains "results" of previous instructions
  - OF = overflow flag, indicates two's complement overflow.
  - SF = sign flag, indicates a negative result.
  - ZF = zero flag, indicates the result was zero.
  - CF = carry flag, indicates unsigned overflow, also used in shifting
- An operation may set, clear, modify or test a flag.
- Some operations leave a flag undefined.

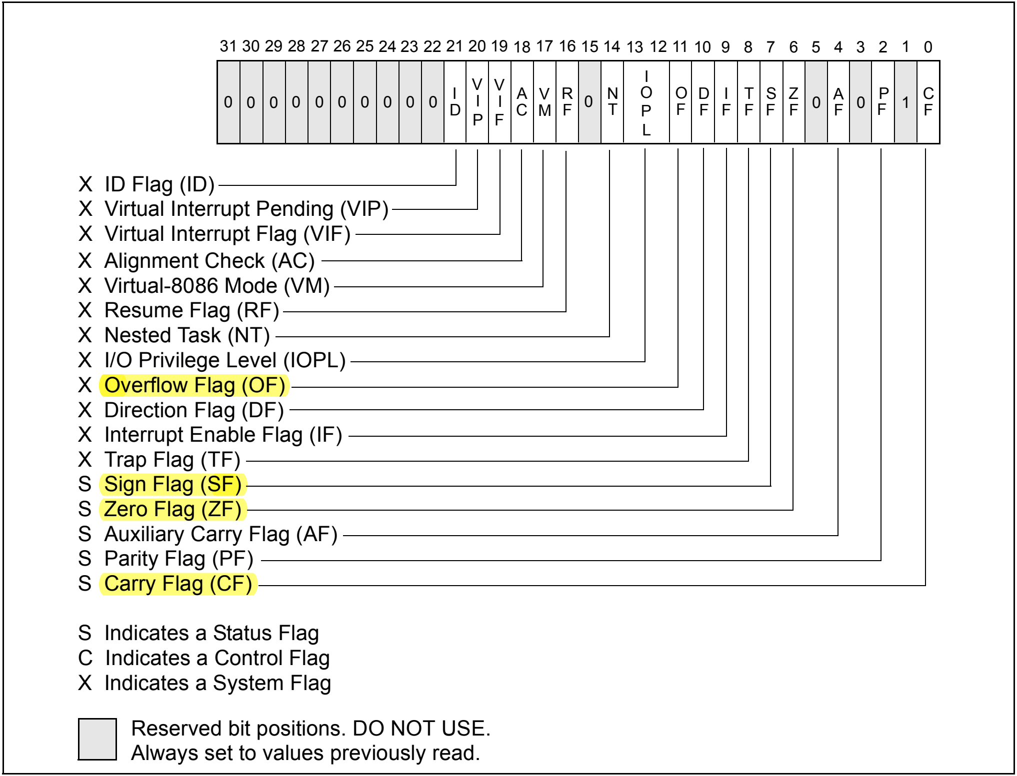


Figure 3-7. EFLAGS Register

Summary of ADD Instruction

- **Basic Function:**
  - Adds source operand to destination operand.
  - Both signed and unsigned addition performed.
- **Addressing Modes:**
  - Source operand can be immediate, a register or memory.
  - Destination operand can be a register or memory.
  - Source and destination cannot both be memory.
- **Flags Affected:**
  - OF = 1 if two's complement overflow occurred
  - SF = 1 if result in two's complement is negative (MSbit = 1)
  - ZF = 1 if result is zero
  - CF = 1 if unsigned overflow occurred

Branching Instructions

- **JMP** = unconditional jump
- Conditional jumps use the flags to decide whether to jump to the given label or to continue.
- The flags were modified by previous arithmetic instructions or by a compare (**CMP**) instruction.
- The instruction

```
CMP     op1, op2
```

computes the unsigned and two's complement subtraction **op1 - op2** and modifies the flags. The contents of **op1** are not affected.

Example of CMP instruction

- Suppose AL contains 254. After the instruction:

```
CMP     AL, 17
```

CF = 0, OF = 0, SF = 1 and ZF = 0.

- A JA (jump above) instruction would jump.
- A JG (jump greater than) instruction wouldn't jump.
- Both signed and unsigned comparisons use the same CMP instruction.
- Signed and unsigned jump instructions interpret the flags differently.

Table 7-4. Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF)=0	Above/not below or equal
JAE/JNB	CF=0	Above or equal/not below
JB/JNAE	CF=1	Below/not above or equal
JBE/JNA	(CF or ZF)=1	Below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	Equal/zero
JNC	CF=0	Not carry
JNE/JNZ	ZF=0	Not equal/not zero
JNP/JPO	PF=0	Not parity/parity odd
JP/JPE	PF=1	Parity/parity even
JCXZ	CX=0	Register CX is zero
JECXZ	ECX=0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) =0	Greater/not less or equal
JGE/JNL	(SF xor OF)=0	Greater or equal/not less
JL/JNGE	(SF xor OF)=1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	Less or equal/not greater
JNO	OF=0	Not overflow
JNS	SF=0	Not sign (non-negative)
JO	OF=1	Overflow
JS	SF=1	Sign (negative)

Closer look at JGE

- JGE jumps if and only if SF = OF
  - Examples using 8-bit registers. Which of these result in a jump?

1.
```
MOV   AL, 96
CMP   AL, 80
JGE   Somewhere
```

2.
```
MOV   AL, -64
CMP   AL, 80
JGE   Somewhere
```

3.
```
MOV   AL, 64
CMP   AL, -80
JGE   Somewhere
```

4.
```
MOV   AL, 64
CMP   AL, 80
JGE   Somewhere
```

- if OF=0, then use SF to check whether A-B >= 0.
- if OF=1, then do opposite of SF.
- JGE works after a CMP instruction, even when subtracting the operands result in an overflow!

Short Jumps vs Near Jumps

- **Jumps use relative addressing**
  - Assembler computes an "offset" from address of current instruction
  - Code produced is "relocatable"
- **Short jumps use 8-bit offsets**
  - Target label must be -128 bytes to +127 bytes away
  - Conditional jumps use short jumps by default. To use a near jump:

```
JGE      NEAR Somewhere
```

- **Near jumps use 32-bit offsets**
  - Target label must be -2^{32} to $+2^{32}-1$ bytes away (4 gigabyte range)
  - Unconditional jumps use near jumps by default. To use a short jump:

```
JMP      SHORT Somewhere
```

```
; File: jmp.asm
;
; Demonstrating near and short jumps
;

        section .text
        global _start

_start: nop

        ; initialize

start:  mov     eax, 17         ; eax := 17
        cmp     eax, 42         ; 17 - 42 is ...

        jge     exit            ; exit if 17 >= 42
        jge     short exit
        jge     near exit

        jmp     exit
        jmp     short exit
        jmp     near exit

exit:   mov     ebx, 0          ; exit code, 0=normal
        mov     eax, 1          ; Exit.
        int     080H            ; Call kernel.
```

```
                          ; File: jmp.asm
                          ;
                          ; Demonstrating near and short jumps
                          ;

                                  section .text
                                  global _start

00000000 90               _start: nop

                                  ; initialize

00000001 B811000000       start:  mov     eax, 17           ; eax := 17
00000006 3D2A000000               cmp     eax, 42           ; 17 - 42 is ...

0000000B 7D14                     jge     exit              ; exit if 17 >= 42
0000000D 7D12                     jge     short exit
0000000F 0F8D0C000000             jge     near exit

00000015 E907000000               jmp     exit
0000001A EB05                     jmp     short exit
0000001C E900000000               jmp     near exit

00000021 BB00000000       exit:   mov     ebx, 0            ; exit code, 0=normal
00000026 B801000000               mov     eax, 1            ; Exit.
0000002B CD80                     int     080H              ; Call kernel.
```

Converting an if Statement

```
if (x < y) {
    statement block 1 ;
} else {
    statement block 2 ;
}
```

```
        MOV     EAX,[x]
        CMP     EAX,[y]
        JGE     ElsePart
         .                      ; if part
         .                      ; statement block 1
         .
        JMP     Done            ; skip over else part

ElsePart:
         .                      ; else part
         .                      ; statement block 2
         .

Done:
```

Converting a while Loop

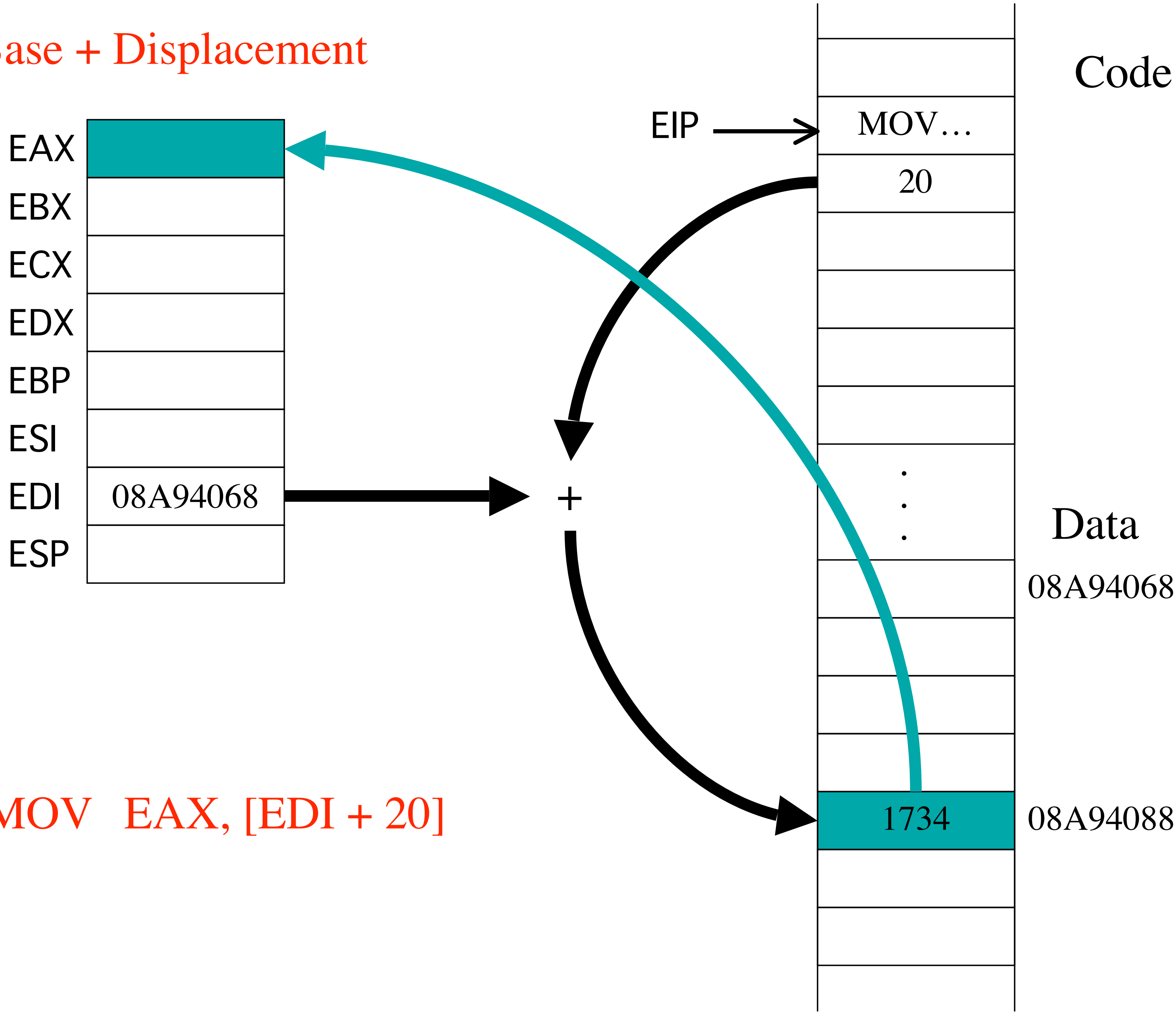
```
while(i > 0) {
    statement 1 ;
    statement 2 ;
    …
}
```

```
WhileTop:
        MOV     EAX,[i]
        CMP     EAX, 0
        JLE     Done
         .                  ; statement 1
         .
         .
         .                  ; statement 2
         .
         .
        JMP     WhileTop
Done:
```

Indexed Addressing

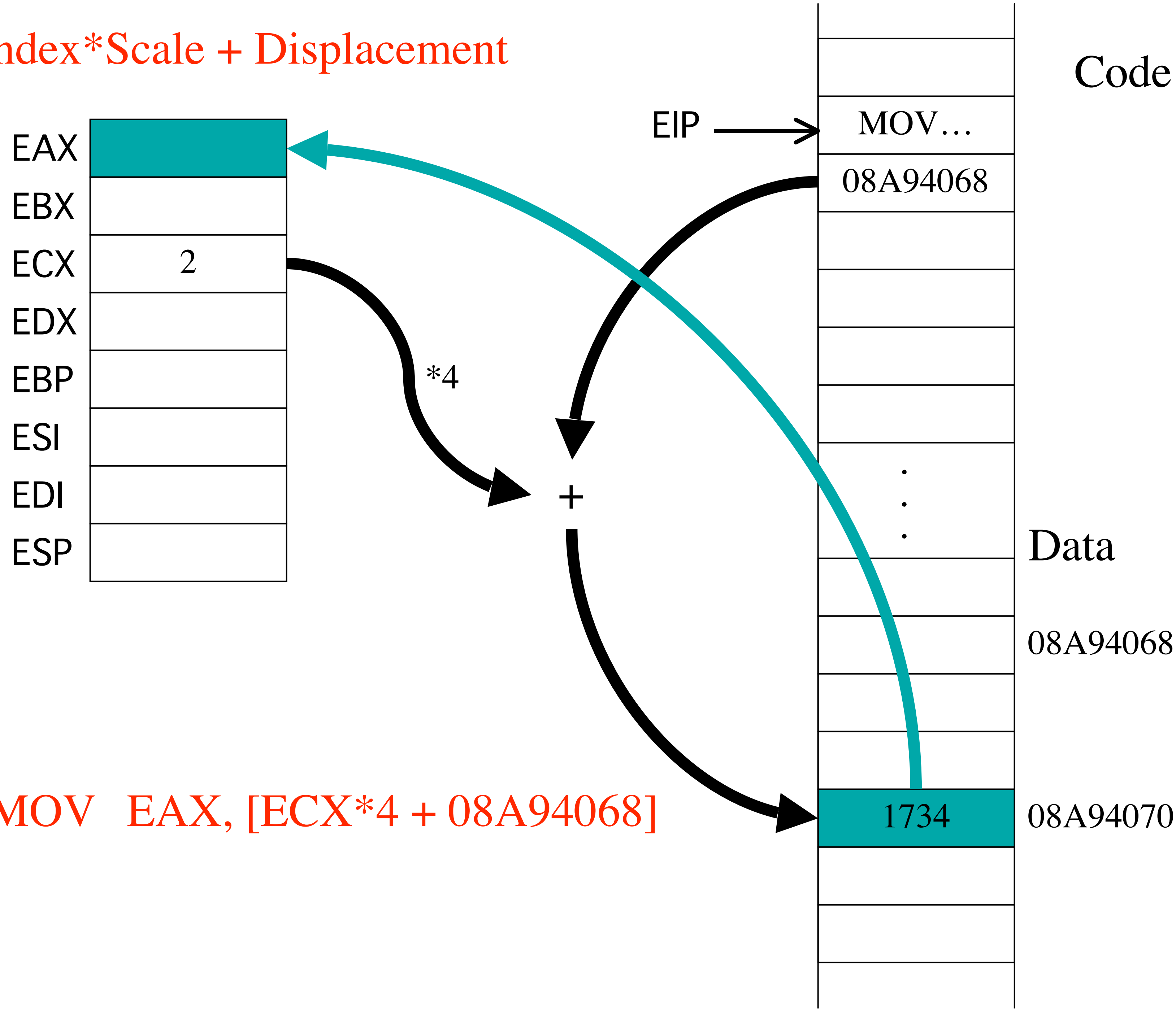
- Operands of the form: [ESI + ECX*4 + DISP]
- ESI = Base Register
- ECX = Index Register
- 4 = Scale factor
- DISP = Displacement
- The operand is in memory
- The address of the memory location is
  ESI + ECX*4 + DISP

Base + Displacement

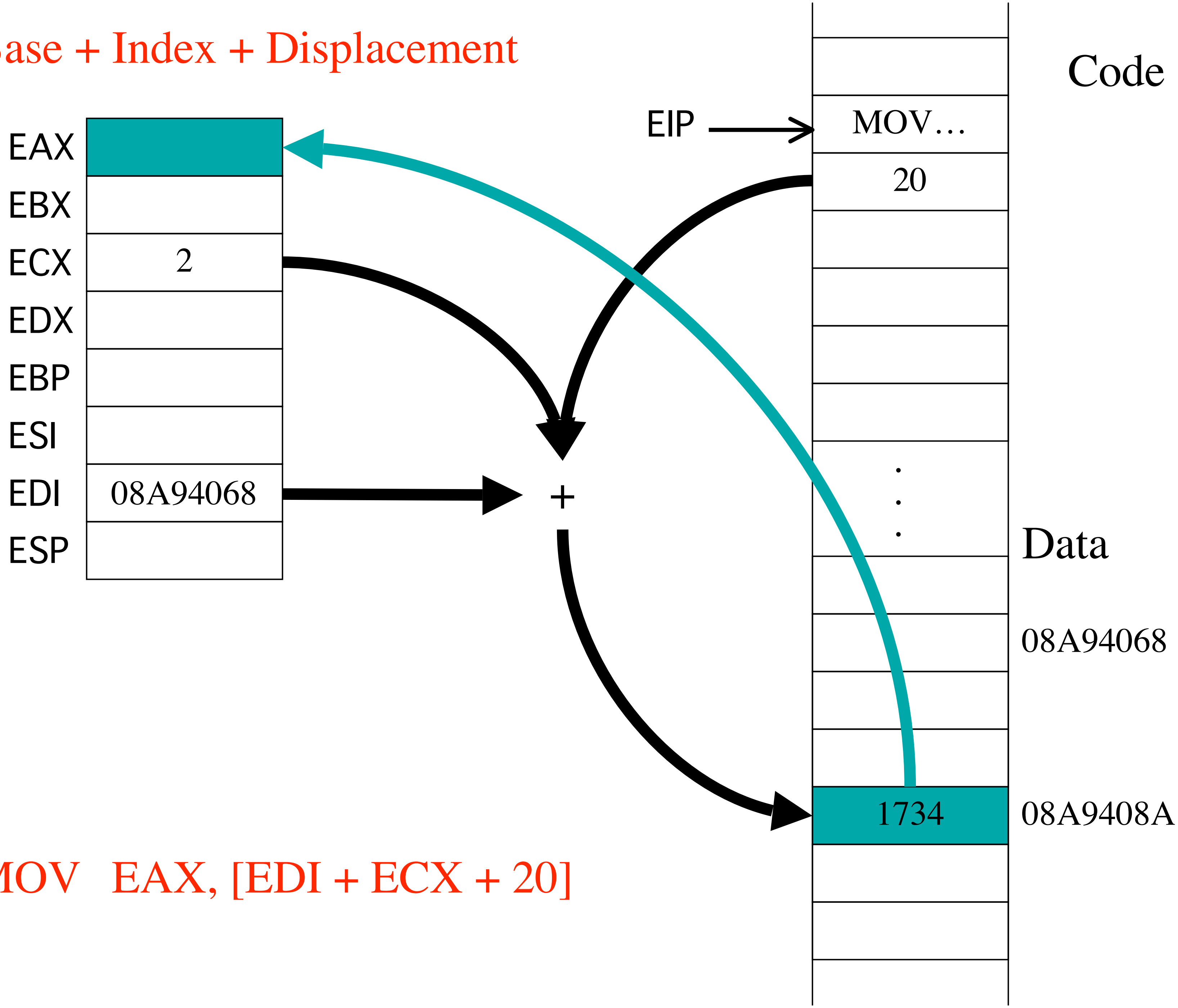


MOV   EAX, [EDI + 20]

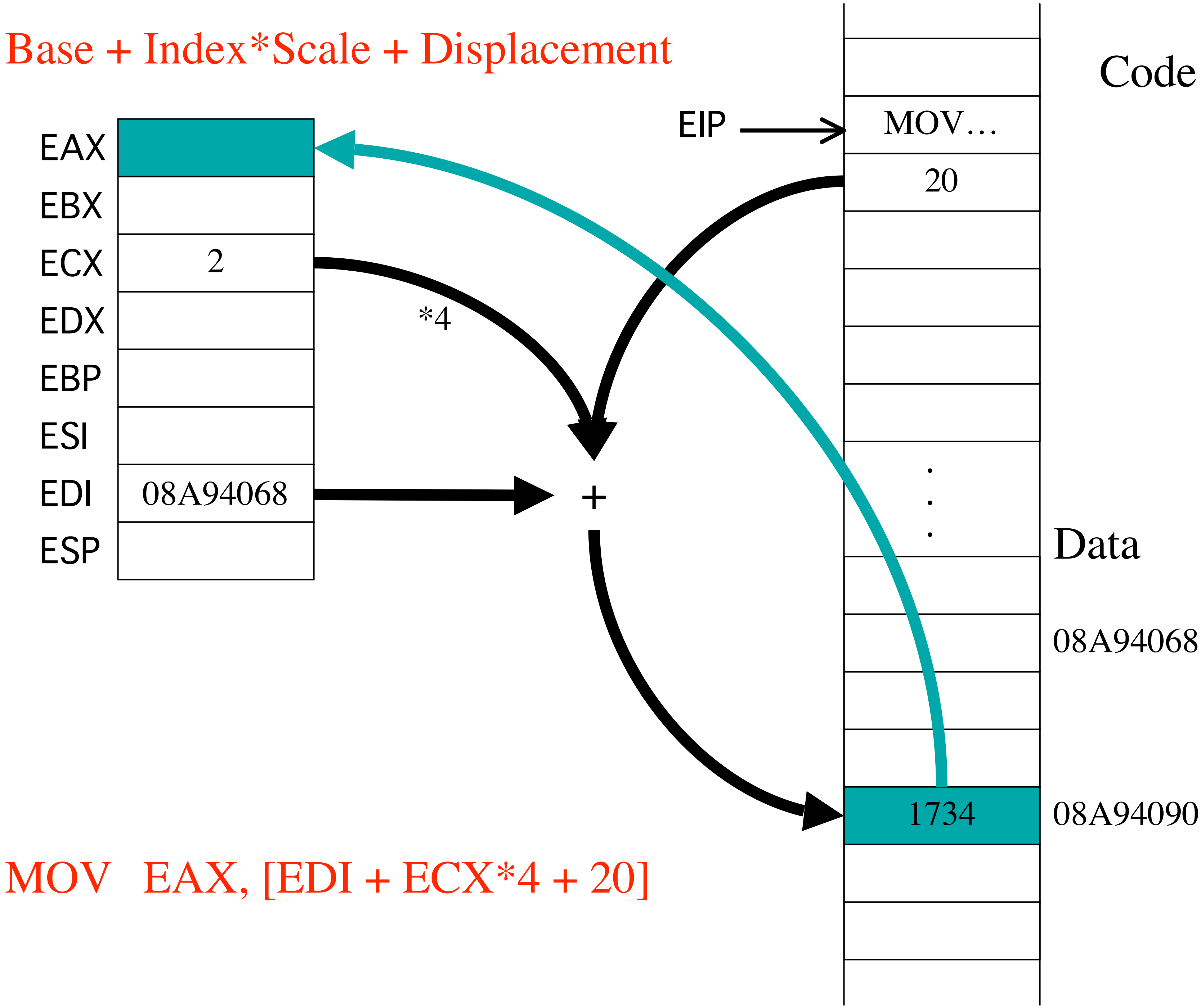
Index*Scale + Displacement



Base + Index + Displacement



MOV EAX, [EDI + ECX + 20]


Base + Index*Scale + Displacement
EAX
EBX
ECX
2
EDX
EBP
ESI
EDI
08A94068
ESP
*4
+
EIP
MOV…
20
Code
Data
08A94068
1734
08A94090
MOV EAX, [EDI + ECX*4 + 20]

Typical Uses for Indexed Addressing

- **Base + Displacement**
  - access character in a string or field of a record
  - access a local variable in function call stack
- **Index*Scale + Displacement**
  - access items in an array where size of item is 2, 4 or 8 bytes
- **Base + Index + Displacement**
  - access two dimensional array (displacement has address of array)
  - access an array of records (displacement has offset of field in a record)
- **Base + (Index*Scale) + Displacement**
  - access two dimensional array where size of item is 2, 4 or 8 bytes

```
; File: index1.asm
;
; This program demonstrates the use of an indexed addressing mode
; to access array elements.
;
; This program has no I/O. Use the debugger to examine its effects.
;
        SECTION .data                   ; Data section

arr:    dd 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ; ten 32-bit words
base:   equ arr - 4


        SECTION .text                   ; Code section.
        global _start
_start: nop                             ; Entry point.

        ; Add 5 to each element of the array stored in arr.
        ; Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;     arr[i] += 5 ;
        ;  }

init1:  mov     ecx, 0                  ; ecx simulates i
loop1:  cmp     ecx, 10                 ; i < 10 ?
        jge     done1
        add     [ecx*4+arr], dword 5    ; arr[i] += 5
        inc     ecx                     ; i++
        jmp     loop1
done1:

        ; more idiomatic for an assembly language program
init2:  mov     ecx, 9                  ; last array elt's index
loop2:  add     [ecx*4+arr], dword 5
        dec     ecx
        jge     loop2                   ; again if ecx >= 0


        ; another way
init3:  mov     edi, base               ; base computed by ld
        mov     ecx, 10                 ; for(i=10 ; i>0 ; i--)
loop3:  add     [edi+ecx*4], dword 5
        loop    loop3                   ; loop = dec ecx, jne

alldone:
        mov     ebx, 0                  ; exit code, 0=normal
        mov     eax, 1                  ; Exit.
        int     80H                     ; Call kernel.
```

```
Script started on Fri Sep 19 13:06:02 2003
linux3% nasm -f elf index1.asm
linux3% ld index1.o

linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...
(gdb) break *init1
Breakpoint 1 at 0x8048081
(gdb) break *init2
Breakpoint 2 at 0x8048099
(gdb) break *init3
Breakpoint 3 at 0x80480ac
(gdb) break * alldone
Breakpoint 4 at 0x80480bf
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in init1 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        0       1       2       3
0x80490dc <arr+16>:     4       5       6       7
0x80490ec <arr+32>:     8       9
(gdb) cont
Continuing.

Breakpoint 2, 0x08048099 in init2 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        5       6       7       8
0x80490dc <arr+16>:     9       10      11      12
0x80490ec <arr+32>:     13      14
(gdb) cont
Continuing.

Breakpoint 3, 0x080480ac in init3 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        10      11      12      13
0x80490dc <arr+16>:     14      15      16      17
0x80490ec <arr+32>:     18      19
(gdb) cont
Continuing.

Breakpoint 4, 0x080480bf in alldone ()
(gdb) x/10wd &arr
0x80490cc <arr>:        15      16      17      18
0x80490dc <arr+16>:     19      20      21      22
0x80490ec <arr+32>:     23      24
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux3% exit
exit

Script done on Fri Sep 19 13:07:41 2003
```

```
; File: index2.asm
;
; This program demonstrates the use of an indexed addressing mode
; to access 2 dimensional array elements.
;
; This program has no I/O. Use the debugger to examine its effects.
;
        SECTION .data                       ; Data section

        ; simulates a 2-dim array
twodim:
row1:   dd 00, 01, 02, 03, 04, 05, 06, 07, 08, 09
row2:   dd 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
        dd 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
        dd 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
        dd 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
        dd 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
        dd 60, 61, 62, 63, 64, 65, 66, 67, 68, 69
        dd 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
        dd 80, 81, 82, 83, 84, 85, 86, 87, 88, 89
        dd 90, 91, 92, 93, 94, 95, 96, 97, 98, 99

rowlen: equ row2 - row1

        SECTION .text                       ; Code section.
        global _start
_start: nop                                 ; Entry point.

        ; Add 5 to each element of row 7. Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;     towdim[7][i] += 5 ;
        ;  }

init1:  mov     ecx, 0                      ; ecx simulates i
        mov     eax, rowlen                 ; offset of twodim[7][0]
        mov     edx, 7
        mul     edx                         ; eax := eax * edx
        jc      alldone                     ; 64-bit product is bad

loop1:  cmp     ecx, 10                     ; i < 10 ?
        jge     done1
        add     [eax+4*ecx+twodim], dword 5
        inc     ecx                         ; i++
        jmp     loop1
done1:

alldone:
        mov     ebx, 0                      ; exit code, 0=normal
        mov     eax, 1                      ; Exit.
        int     80H                         ; Call kernel.
```

```
Script started on Fri Sep 19 13:19:22 2003
linux3% nasm -f elf index2.asm
linux3% ld index2.o
linux3%
linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...
(gdb) break *init1
Breakpoint 1 at 0x8048081
(gdb) break *alldone
Breakpoint 2 at 0x80480a7
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in init1 ()
(gdb) x/10wd &twodim
0x80490b4 <twodim>:     0       1       2       3
0x80490c4 <twodim+16>:  4       5       6       7
0x80490d4 <twodim+32>:  8       9
(gdb) x/10wd &twodim+60
0x80491a4 <row2+200>:   60      61      62      63
0x80491b4 <row2+216>:   64      65      66      67
0x80491c4 <row2+232>:   68      69
(gdb)
0x80491cc <row2+240>:   70      71      72      73
0x80491dc <row2+256>:   74      75      76      77
0x80491ec <row2+272>:   78      79
(gdb)
0x80491f4 <row2+280>:   80      81      82      83
0x8049204 <row2+296>:   84      85      86      87
0x8049214 <row2+312>:   88      89
(gdb) cont
Continuing.

Breakpoint 2, 0x080480a7 in done1 ()
(gdb) x/10wd &twodim+60
0x80491a4 <row2+200>:   60      61      62      63
0x80491b4 <row2+216>:   64      65      66      67
0x80491c4 <row2+232>:   68      69
(gdb)
0x80491cc <row2+240>:   75      76      77      78
0x80491dc <row2+256>:   79      80      81      82
0x80491ec <row2+272>:   83      84
(gdb)
0x80491f4 <row2+280>:   80      81      82      83
0x8049204 <row2+296>:   84      85      86      87
0x8049214 <row2+312>:   88      89
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux3% exit
exit

Script done on Fri Sep 19 13:20:35 2003
```

i386 String Instructions

- Special instructions for searching & copying strings
- Assumes that AL holds the data
- Assumes that ECX holds the “count”
- Assumes that ESI and/or EDI point to the string(s)
- Some examples (there are many others):
  - LODS: loads AL with [ESI], then increments or decrements ESI
  - STOS: stores AL in [EDI], then increments or decrements EDI
  - CLD/STD: clears/sets direction flag DF. Makes LODS & STOS auto-inc/dec.
  - LOOP: decrements ECX. Jumps to label if ECX != 0 after decrement.
  - SCAS: compares AL with [EDI], sets status flags, auto-inc/dec EDI.
  - REP: Repeats a string instruction

Debugging Assembly Language Programs

- Cannot just put print statements everywhere.
- Use gdb to:
  - examine contents of registers
  - exmaine contents of memory
  - set breakpoints
  - single-step through program
- READ THE GDB SUMMARY ONLINE!

gdb ommand Summary

Command	Example	Description
run		start program
quit		quit out of gdb
cont		continue execution after a break
break [addr]	break *_start+5	sets a breakpoint
delete [n]	delete 4	removes nth breakpoint
delete		removes all breakpoints
info break		lists all breakpoints
stepi		execute next instruction
stepi [n]	stepi 4	execute next n instructions
nexti		execute next instruction, stepping over function calls
nexti [n]	nexti 4	execute next n instructions, stepping over function calls
where		show where execution halted
disas [addr]	disas _start	disassemble instructions at given address
info registers		dump contents of all registers
print/d [expr]	print/d $ecx	print expression in decimal
print/x [expr]	print/x $ecx	print expression in hex
print/t [expr]	print/t $ecx	print expression in binary
x/NFU [addr]	x/12xw &msg	Examine contents of memory in given format
display [expr]	display $eax	automatically print the expression each time the program is halted
	display/i $eip	print machine instruction each time the program is halted
info display		show list of automatically displays
undisplay [n]	undisplay 1	remove an automatic display

Stack Instructions

- **PUSH *op***
  - the stack pointer ESP is decremented by the size of the operand
  - the operand is copied to [ESP]
- **POP *op***
  - the reverse of PUSH
  - [ESP] is copied to the destination operand
  - ESP is incremented by the size of the operand
- **Where is the stack?**
  - The stack has its own section
  - Linux processes wake up with ESP initialized properly
  - The stack grows “upward” – toward smaller addresses
  - Memory available to the stack set using ‘limit’

Subroutine Instructions

- **CALL *label***
  - Used to call a subroutine
  - PUSHes the instruction pointer (EIP) on the stack
  - jump to the label
  - does NOTHING else
- **RET**
  - reverse of CALL
  - POPs the instruction pointer (EIP) off the stack
  - execution proceeds from the instruction after the CALL instruction
- **Parameters?**

Linux/gcc/i386 Function Call Convention

- Parameters pushed right to left on the stack
  - first parameter on top of the stack
- Caller saves EAX, ECX, EDX if needed
  - these registers will probably be used by the callee
- Callee saves EBX, ESI, EDI
  - there is a good chance that the callee does not need these
- EBP used as index register for parameters, local variables, and temporary storage
- Callee must restore caller's ESP and EBP
- Return value placed in EAX

A typical stack frame for the function call:

```
int foo (int arg1, int arg2, int arg3) ;
```

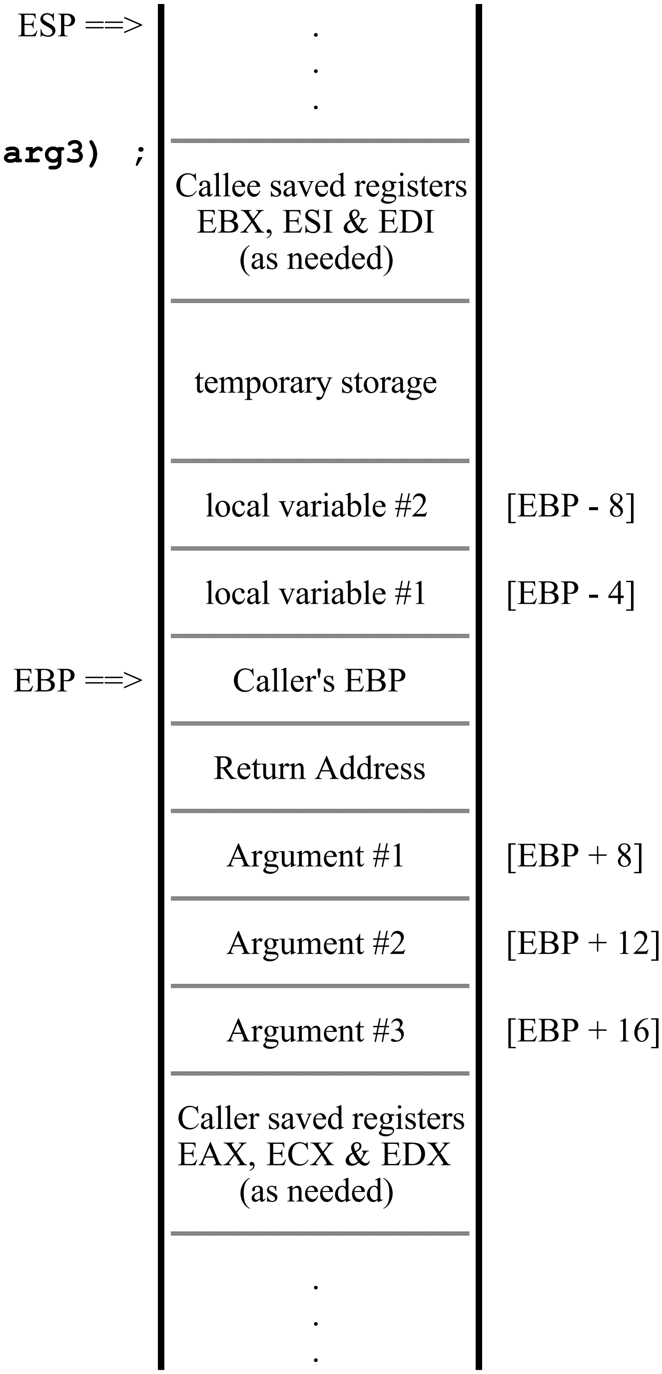


Fig. 1

The caller's actions before the function call

- Save EAX, ECX, EDX registers as needed
- Push arguments, last first
- CALL the function

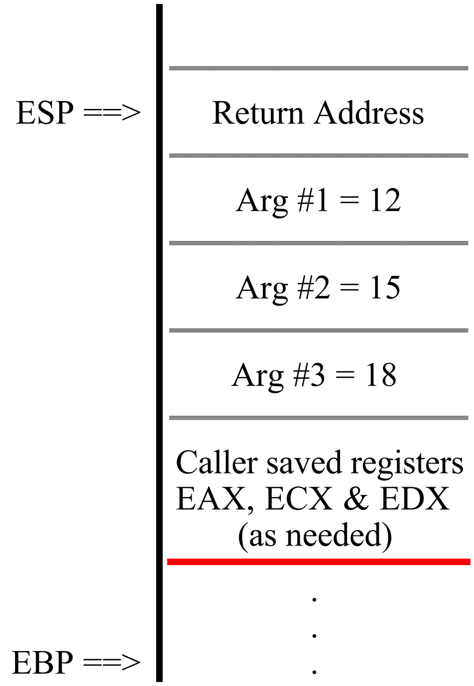


Fig. 2

The callee's actions after function call

- Save main's EBP, set up own stack frame

```
push    ebp
mov     ebp, esp
```

- Allocate space for local variables and temporary storage
- Save EBX, ESI and EDI registers as needed

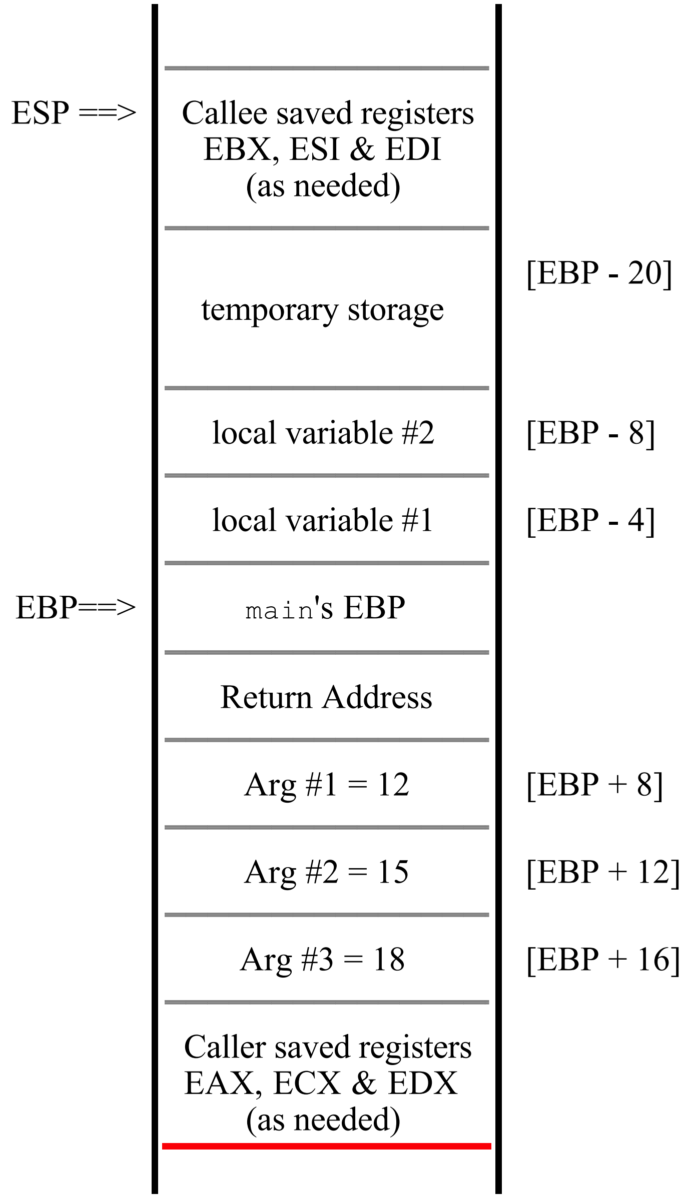


Fig. 4

The callee's actions before returning

- Store return value in EAX
- Restore EBX, ESI and EDI registers as needed
- Restore main's stack frame

```
mov      esp, ebp
pop      ebp
```

- RET to main

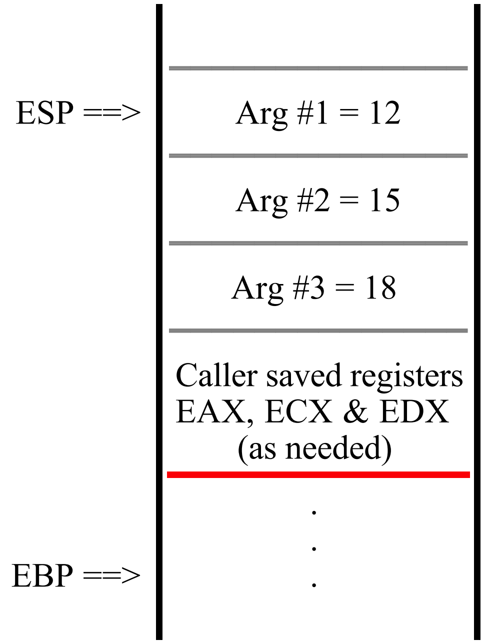


Fig. 5

The caller's actions after returning

- POP arguments off the stack
- Store return value (which is in EAX) somewhere
- Restore EAX, ECX and EDX registers as needed

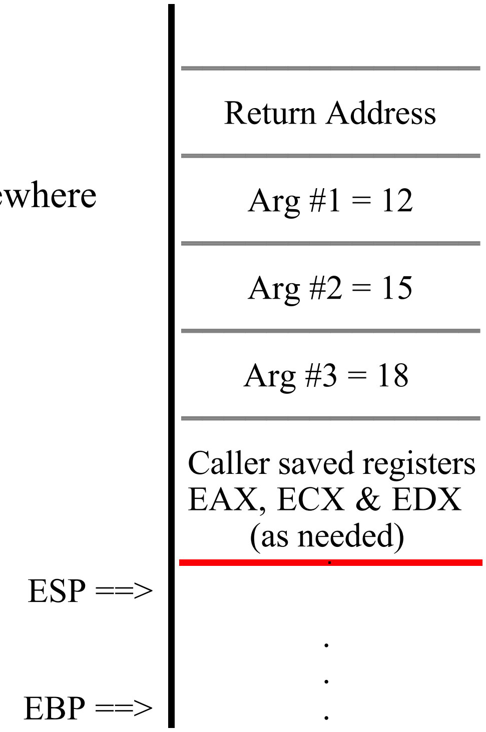


Fig. 6

```
// File: cfunc.c
//
// Example of C function calls disassembled
//

#include <stdio.h>

// a silly function
//
int foo(int x, int y) {

   int z ;

   z = x + y ;
   return z ;
}

int main () {
   int b ;

   b = foo(35, 64) ;
   b = b + b ;
   printf ("b = %d\n", b) ;
}
```

```
linux3% gcc cfunc.c
linux3% a.out
b = 198
linux3%


linux3% gcc -S cfunc.c
linux3% i2g -g cfunc.s >cfunc.asm
linux3%
```

```
        .file   "cfunc.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl foo
        .type    foo,@function
foo:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
        movl 8(%ebp),%eax
        movl 12(%ebp),%edx
        leal (%edx,%eax),%ecx
        movl %ecx,-4(%ebp)
        movl -4(%ebp),%edx
        movl %edx,%eax
        jmp .L1
        .p2align 4,,7
.L1:
        leave
        ret
```

```
.Lfe1:
        .size    foo,.Lfe1-foo
.section        .rodata
.LC0:
        .string "b = %d\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
        pushl $64
        pushl $35
        call foo
        addl $8,%esp
        movl %eax,%eax
        movl %eax,-4(%ebp)
        movl -4(%ebp),%eax
        addl %eax,-4(%ebp)
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $8,%esp
.L2:
        leave
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
```

```
        ;FILE "cfunc.c"
gcc2_compiled.:
SECTION .text
        ALIGN 4
GLOBAL foo
        GLOBAL foo:function
foo:
        push  ebp
        mov  ebp,esp
        sub  esp,4
        mov  eax, [ebp+8]
        mov  edx, [ebp+12]
        lea  ecx, [edx+eax]
        mov  [ebp-4],ecx
        mov  edx, [ebp-4]
        mov  eax,edx
        jmp L1
        ;ALIGN 1<<4 ; IF < 7
L1:
        leave
        ret
```

```
.Lfe1:
        GLOBAL   foo:function (.Lfe1-foo)
SECTION         .rodata
.LC0:
        db      'b = %d',10,''
SECTION .text
        ALIGN 4
GLOBAL main
        GLOBAL main:function
main:
        push  ebp
        mov  ebp,esp
        sub  esp,4
        push  dword 64
        push  dword 35
        call foo
        add  esp,8
        mov  eax,eax
        mov  [ebp-4],eax
        mov  eax, [ebp-4]
        add  [ebp-4],eax
        mov  eax, [ebp-4]
        push  eax
        push  dword .LC0
        call printf
        add  esp,8
L2:
        leave
        ret
.Lfe2:
        GLOBAL   main:function (.Lfe2-main)
        ;IDENT "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
```

```
.Lfe1:
        GLOBAL   foo:function (.Lfe1-foo)
SECTION         .rodata
.LC0:
        db      'b = %d',10,''
SECTION .text
        ALIGN 4
GLOBAL main
        GLOBAL main:function
main:
        push  ebp
        mov  ebp,esp
        sub  esp,4
        push  dword 64
        push  dword 35
        call foo
        add  esp,8
        mov  eax,eax
        mov  [ebp-4],eax
        mov  eax, [ebp-4]
        add  [ebp-4],eax
        mov  eax, [ebp-4]
        push  eax
        push  dword .LC0
        call printf
        add  esp,8
L2:
        leave
        ret
.Lfe2:
        GLOBAL   main:function (.Lfe2-main)
        ;IDENT "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
```

```
; File: printf1.asm
;
; Using C printf function to print
;
; Assemble using NASM:  nasm -f elf printf1.asm
;
; C-style main function.
; Link with gcc:  gcc printf1.o
;

; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section

msg:    db "Hello, world: %c", 10, 0    ; The string to print.


        SECTION .text                   ; Code section.

        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp

        push    dword 97                ; an 'a'
        push    dword msg               ; address of ctrl string
        call    printf                  ; Call C function
        add     esp, 8                  ; pop stack

        mov     esp, ebp                ; takedown stack frame
        pop     ebp                     ;   same as "leave" op

        ret
```

```
linux3% nasm -f elf printf1.asm
linux3% gcc printf1.o

linux3% a.out
Hello, world: a
linux3% exit
```

```
; File: printf2.asm
;
; Using C printf function to print
;
; Assemble using NASM:  nasm -f elf printf2.asm
;
; Assembler style main function.
; Link with gcc: gcc -nostartfiles printf2.asm
;

%define SYSCALL_EXIT  1

; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section

msg:    db "Hello, world: %c", 10, 0    ; The string to print.


        SECTION .text                   ; Code section.

        global _start
_start:
        push    dword 97                ; an 'a'
        push    dword msg               ; address of ctrl string
        call    printf                  ; Call C function
        add     esp, 8                  ; pop stack

        mov     eax, SYSCALL_EXIT       ; Exit.
        mov     ebx, 0                  ; exit code, 0=normal
        int     080H                    ; ask kernel to take over
```

```
linux3% nasm -f elf printf2.asm
linux3% gcc -nostartfiles printf2.o
linux3%

linux3% a.out
Hello, world: a
linux3%
```

```c
// File: arraytest.c
//
// C program to test arrayinc.asm
//

void arrayinc(int A[], int n) ;

main() {

int A[7] = {2, 7, 19, 45, 3, 42, 9} ;
int i ;

   printf ("sizeof(int) = %d\n", sizeof(int)) ;

   printf("\nOriginal array:\n") ;
   for (i = 0 ; i < 7 ; i++) {
      printf("A[%d] = %d  ", i, A[i]) ;
   }
   printf("\n") ;

   arrayinc(A,7) ;

   printf("\nModified array:\n") ;
   for (i = 0 ; i < 7 ; i++) {
      printf("A[%d] = %d  ", i, A[i]) ;
   }
   printf("\n") ;

}
```

```
linux3% gcc -c arraytest.c
linux3% nasm -f elf arrayinc.asm
linux3% gcc arraytest.o arrayinc.o
linux3%
linux3% a.out
sizeof(int) = 4

Original array:
A[0] = 2  A[1] = 7  A[2] = 19  A[3] = 45  A[4] = 3  A[5] = 42  A[6] = 9

Modified array:
A[0] = 3  A[1] = 8  A[2] = 20  A[3] = 46  A[4] = 4  A[5] = 43  A[6] = 10
linux3%
```

```
; File: arrayinc.asm
;
; A subroutine to be called from C programs.
; Parameters: int A[], int n
; Result: A[0], ... A[n-1] are each incremented by 1


        SECTION .text
        global arrayinc

arrayinc:
        push    ebp                         ; set up stack frame
        mov     ebp, esp

        ; registers ebx, esi and edi must be saved, if used
        push    ebx
        push    edi

        mov     edi, [ebp+8]                ; get address of A
        mov     ecx, [ebp+12]               ; get num of elts
        mov     ebx, 0                      ; initialize count

for_loop:
        mov     eax, [edi+4*ebx]            ; get array element
        inc     eax                         ; add 1
        mov     [edi+4*ebx], eax            ; put it back
        inc     ebx                         ; update counter
        loop    for_loop

        pop     edi                         ; restore registers
        pop     ebx

        mov     esp, ebp                    ; take down stack frame
        pop     ebp

        ret
```

```c
// File: cfunc3.c
//
// Example of C function calls disassembled
// Return values with  more than 4 bytes
//

#include <stdio.h>

typedef struct {
   int part1, part2 ;
} stype ;


// a silly function
//
stype foo(stype r) {

   r.part1 += 4;
   r.part2 += 3 ;
   return r ;
}


int main () {
   stype r1, r2, r3  ;
   int n ;

   n = 17 ;
   r1.part1 = 74 ;
   r1.part2 = 75 ;
   r2.part1 = 84 ;
   r2.part2 = 85 ;
   r3.part1 = 93 ;
   r3.part2 = 99 ;

   r2 = foo(r1) ;

   printf ("r2.part1 = %d, r2.part2 = %d\n",
    r1.part1, r2.part2 ) ;

   n = foo(r3).part2 ;
}
```

```
        ;FILE "cfunc3.c"
gcc2_compiled.:
SECTION .text
        ALIGN 4
GLOBAL foo
        GLOBAL foo:function
foo:                                        ; comments & spacing added
        push  ebp                           ; set up stack frame
        mov   ebp,esp

        mov   eax, [ebp+8]                  ; addr to store return value
        add   dword [ebp+12],4              ; r.part1 = [ebp+12]
        add   dword [ebp+16],3              ; r.part2 = [ebp+16]

        ; return value
        ;
        mov   edx, [ebp+12]                 ; get r.part1
        mov   ecx, [ebp+16]                 ; get r.part2
        mov   [eax],edx                     ; put r.part1 in return valu
        mov   [eax+4],ecx                   ; put r.part2 in return valu
        jmp L1
L1:
        mov   eax,eax                       ; does nothing
        leave                               ; bye-bye
        ret 4                               ; pop 4 bytes after return
.Lfe1:
```

```
        GLOBAL   foo:function (.Lfe1-foo)
SECTION        .rodata
.LC0:
        db      'r2.part1 = %d, r2.part2 = %d',10,''
SECTION .text
        ALIGN 4
GLOBAL main
        GLOBAL main:function
main:                                   ; comments & spacing added
        push  ebp                       ; set up stack frame
        mov  ebp,esp
        sub  esp,36                     ; space for local variables

        ; initialize variables
        ;
        mov  dword [ebp-28],17          ; n = [ebp-28]
        mov  dword [ebp-8],74           ; r1 = [ebp-8]
        mov  dword [ebp-4],75
        mov  dword [ebp-16],84          ; r2 = [ebp-16]
        mov  dword [ebp-12],85
        mov  dword [ebp-24],93          ; r3 = [ebp-24]
        mov  dword [ebp-20],99

        ; call foo
        ;
        lea  eax, [ebp-16]              ; get addr of r2
        mov  edx, [ebp-8]               ; get r1.part1
        mov  ecx, [ebp-4]               ; get r1.part2
        push  ecx                       ; push r1.part2
        push  edx                       ; push r1.part1
        push  eax                       ; push addr of r2
        call foo
        add  esp,8                      ; pop r1
                                        ; ret 4 popped r2's addr

        ; call printf
        ;
        mov  eax, [ebp-12]              ; get r2.part2
        push  eax                       ; push it
        mov  eax, [ebp-8]               ; get r2.part1
        push  eax                       ; push it
        push  dword .LC0                ; string constant's addr
        call printf
        add  esp,12                     ; pop off arguments
```

```
        ; call foo again
        ;
        lea  eax, [ebp-36]                  ; addr of temp variable
        mov  edx, [ebp-24]                  ; get r3.part1
        mov  ecx, [ebp-20]                  ; get r3.part2
        push  ecx                           ; push r3.part2
        push  edx                           ; push r3.part1
        push  eax                           ; push addr of temp var
        call foo
        add  esp,8                          ; pop off arguments

        ; assign to n
        ;
        mov  eax, [ebp-32]                  ; get part2 of temp var
        mov  [ebp-28],eax                   ; store in n

L2:
        leave                               ; bye-bye
        ret
.Lfe2:
        GLOBAL   main:function (.Lfe2-main)
        ;IDENT "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
```