Sistemi Operativi
A. Y. 17/18

Lect. 20: IA-32

Modalità operative

- L'architettura IA-32 (a partire da Intel386) fornisce una serie di funzionalità per la realizzazione di software di sistema, queste funzionalità variano in funzione delle modalità con cui opera il processore :
  - Real mode
  - Protected mode
  - Virtual 8086 mode
  - System Management mode
  - IA-32e mode

Modalità operative

- Real-address mode: è la modalità operativa in cui il processore fornisce le stesse funzionalità presenti nell'Intel 8086 con l'aggiunta di alcune estensioni (la possibilità di cambiare modalità operativa in protected o system management mode)
- Protected mode: la modalità nativa del processore che fornisce un insieme di funzionalità predefinite per la realizzazione di sistemi multiprogrammati garantendo back compatibilità
- Virtual-8086 mode : dalla modalità protected è possibile passare alla modalità virtual-8086 mode, che consente al processore di eseguire software predisposto per l'architettura 8086 in un ambiente multiprogrammato e protetto

Modalità operative

- System management mode (SMM): SMM introdotto nel 1990 sul processore 386SL, è una modalità che consente al processore di eseguire codice memorizzato in una zona riservata della memoria nota come SMRAM (System Management RAM). Il codice memorizzato in questa porzione di memoria viene quindi eseguito ad livello di priorità estremamente elevato (-2)
  - Usato per la gestione di funzionalità hardware: accensione spegnimento dispositivi, diagnostica di componenti

IA-64

- Con l'avvento delle architetture a 64 bit è stata introdotta la nuova modalità operativa IA-32e che consente sia il supporto di applicazioni a 32 bit che quello di applicazioni a 64, più precisamente il software può essere eseguito in due sotto modalità:
  - 64-bit mode per il supporto di applicazioni e OS a 64-bit
  - Compatibility mode: che consente ad un sistema operativo a 64 bit di gestire applicazioni a 64/32-bit

Modalità processore

- All'accensione il processore viene automaticamente predisposto in real-address mode. Il valore del flag PE nel registro di controllo CR0 determinerà successivamente se il processore stia operando in real o protected mode
- Il processore entra in modalità SMM ogni volta che riceve un interrupt SMI, eseguito in una delle modalità real-address, protected, virtual-8086, o IA-32e modes.
- Il processore rientra nella sua modalità "normale" a seguito dell'esecuzione di un'istruzione RSM

Modalità processore

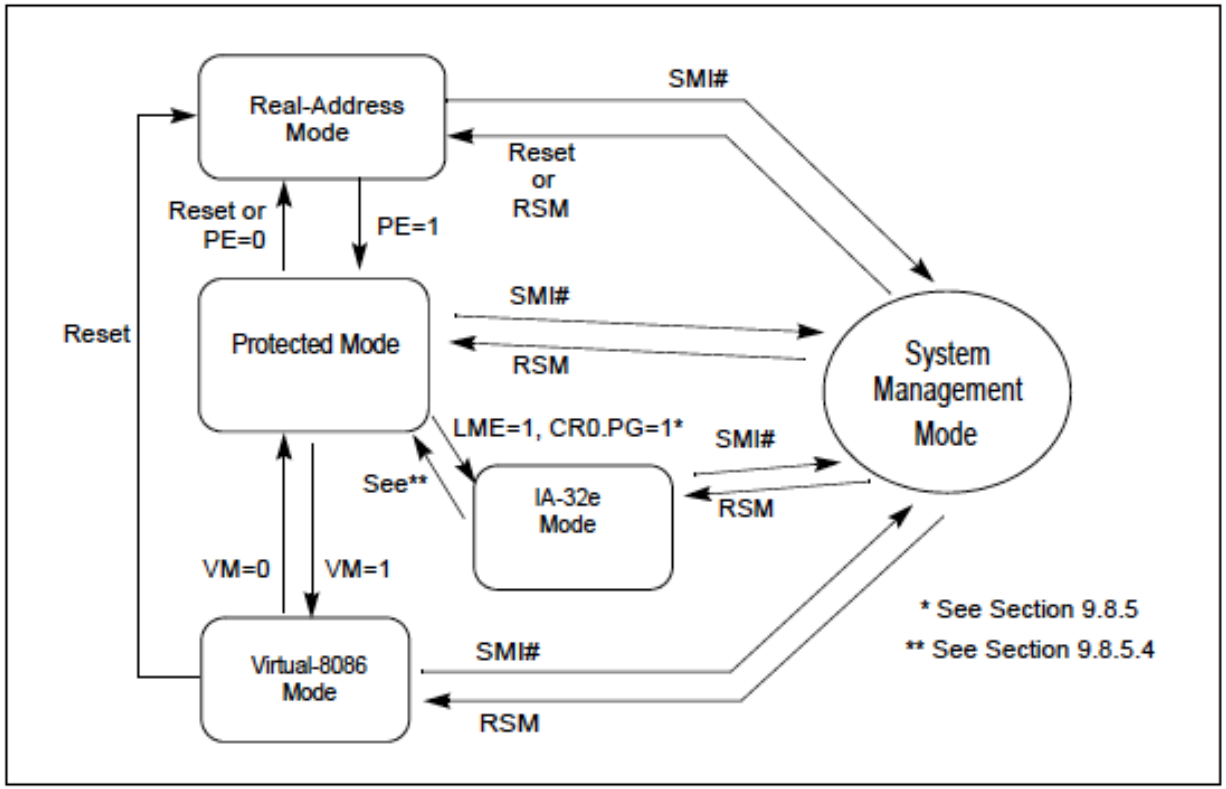


Figure 2-3. Transitions Among the Processor's Operating Modes

Protected mode

- Modalità introdotta per facilitare la soluzione di problemi di "sicurezza" derivanti dal ricorso al paradigma della multiprogrammazione. A tal fine l'architettura IA-32 include meccanismi di protezione e separazione che si prefiggono i seguenti obiettivi:
  - impedire a un processo di modificare l'area dati di un altro processo (restrizione sulla visibilità della memoria)
  - impedire a un processo di accedere direttamente alle risorse del sistema operativo
  - in caso di errore in un task, assicurare la sopravvivenza del sistema operativo e non compromettere la funzionalità degli altri task

Protected Mode

- I meccanismi predisposti dall'hardware per risolvere i suddetti problemi sono basati su due principi:
  - ISOLAMENTO: ad ogni processo utente è assegnato uno spazio di indirizzamento diverso, quindi processi diversi non hanno modo di interferire tra loro, se non attraverso sistemi controllati di IPC
  - La porzione di memoria dedicata al SO non può essere acceduta da processi utente
  - Protezione: ai diversi oggetti che popolano il sistema sono assegnati livelli di protezione (0-1-2-3), oggetti con livello di protezione k possono accedere solo ad oggetti con un livello di protezione uguale o maggiore a k

I Segmenti

- Il meccanismo di isolamento è implementato attraverso il processo di segmentazione, i segmenti sono porzioni di memoria DISGIUNTE in cui sono caricati processi, dati e opportune strutture dati
- Nella modalità di esecuzione protected i processi utente devono essere “spezzati” in due o più segmenti che possono essere di due tipi, in funzione del contenuto: codice o Dati/ stack
- I processi sono gestiti dal processore attraverso un apposito segmento noto come Task State Segment (TSS)
- Un ulteriore segmento (opzionale) è costituito da opportune tabelle, con l’elenco degli oggetti accessibili a un processo (LocalDescriptorTable LDT)

I segmenti

- Le tipologie di segmento imposte dall'architettura e che studieremo sono quindi:
  - Codice
  - Dati/stack
  - TSS
  - LDT

Descrittori

- Ciascuno dei segmenti sopra indicati deve essere provvisto di un opportuna struttura dati che ne descrive le principali proprietà, questa struttura dati è chiamata DESCRITTORE
- Un descrittore di segmento fornisce al processore le seguenti informazioni:
  - La dimensione e l'indirizzo di partenza di un segmento,
  - Informazioni per il controllo degli accessi e sullo stato del segmento
- I descrittori di segmento sono generati dal SO o da programmi di sistema

Contenuti di un Segment Descriptor

- Base Address
  - Un numero di 32-bit che definisce l'indirizzo di partenza del segmento
  - 32-bit Base Address + 32-bit Offset = 32-bit Linear Address
- Segment Limit
  - Un numero di 20-bit che specifica la dimensione del segmento
  - La dimensione può essere specificata in byte o pagine (4 KB)
- Diritti di accesso
  - Se il segmento contiene codice o dati
  - Se i dati sono read-only, in sola lettura o anche scrivibili
  - Il livello di privilegio del segmento

Descrittori speciali

- L'architettura prevede anche un insieme di descrittori speciali, non legati a segmenti, chiamati GATE (call gate, interrupt gate, trap gate e task gate)
- Si tratta di un meccanismo fornito dall'hw per controllare l'esecuzione di codice privilegiato da parte di applicazioni che operano in user space

Layout descrittore segmento (8 byte)

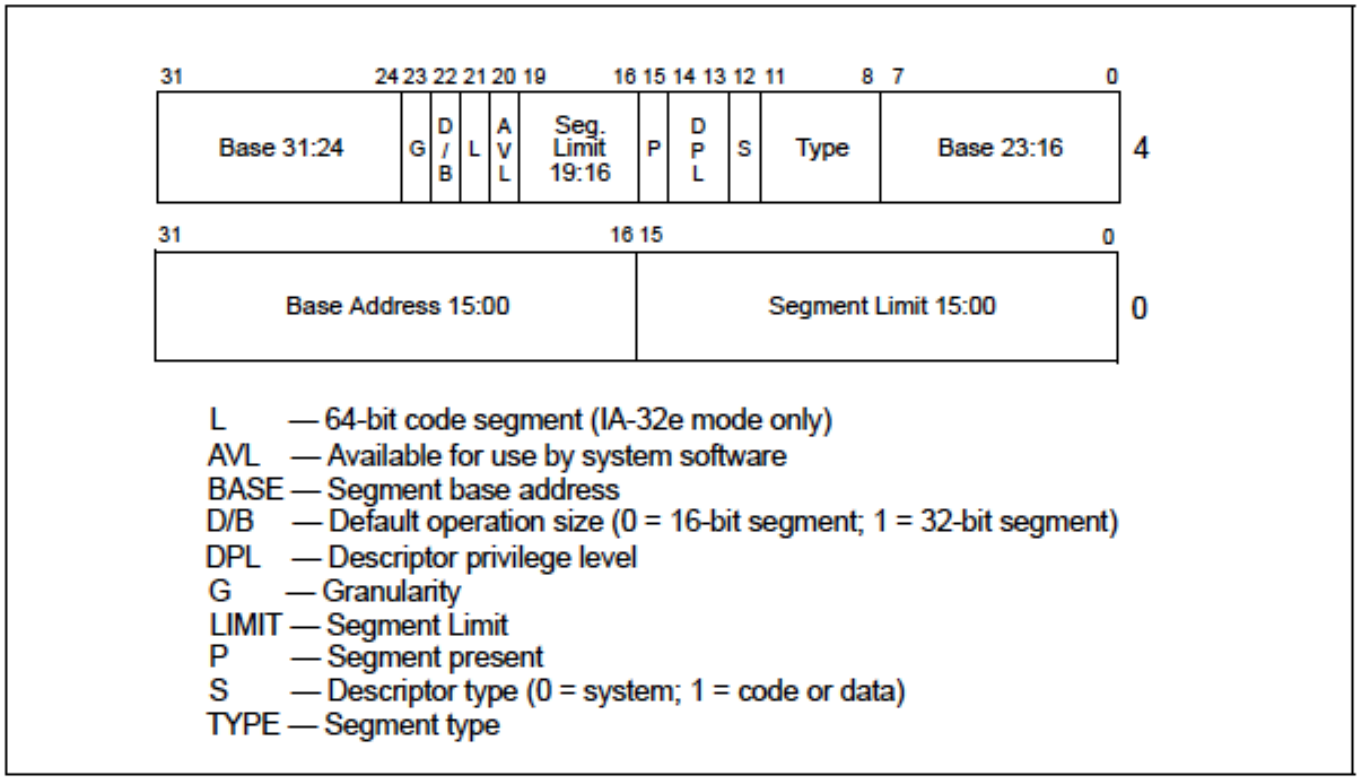


Figure 3-8. Segment Descriptor

Segment Descriptor

```
// Segment Descriptors
struct Segdesc {
    unsigned sd_lim_15_0 : 16;  // Low bits of segment limit
    unsigned sd_base_15_0 : 16; // Low bits of segment base address
    unsigned sd_base_23_16 : 8; // Middle bits of segment base address
    unsigned sd_type : 4;       // Segment type (see STS_ constants)
    unsigned sd_s : 1;          // 0 = system, 1 = application
    unsigned sd_dpl : 2;        // Descriptor Privilege Level
    unsigned sd_p : 1;          // Present
    unsigned sd_lim_19_16 : 4;  // High bits of segment limit
    unsigned sd_avl : 1;        // Unused (available for software use)
    unsigned sd_rsv1 : 1;       // Reserved
    unsigned sd_db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
    unsigned sd_g : 1;          // Granularity: limit scaled by 4K when set
    unsigned sd_base_31_24 : 8; // High bits of segment base address
};
```

Bit field

- ***Bit Fields*** allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples
  - Packing several objects into a machine word. ***e.g.*** 1 bit flags can be compacted -- Symbol tables in compilers.
  - Reading external file formats -- non-standard file formats could be read in. ***E.g.*** 9 bit integers.
- C lets us do this in a structure definition by putting :***bit length*** after the variable. ***i.e.***

```
struct packed_struct {
      unsigned int f1:1;
      unsigned int f2:1;
      unsigned int f3:1;
      unsigned int f4:1;
      unsigned int type:4;
      unsigned int funny_int:9;
               } pack;
```

Tabelle dei descrittori

- A loro volta i DESCRITTORI sono radunati in TABELLE di DESCRITTORI
- Le TABELLE di DESCRITTORI sono organizzate in forma di vettori e sono referenziate attraverso REGISTRI di sistema
- Si accede ai DESCRITTORI all'interno delle TABELLE attraverso un indice lineare detto "SELETTORE"

Selettori

- I selettori hanno la seguente struttura:

Indice di 13 bit (max 8K descrittori per tab)	Indicatore di tabella TI (un bit)	RPL (2 bit)

- TI vale 1 nei selettori della LDT; TI vale 0 nei selettori della GDT
- RPL (Requestor Privilege Level) è il livello di previlegio del segmento di codice che ha generato il selettore

Le principali tabelle

- GDT o Global Descriptor Table è una tabella contente descrittori di segmenti accessibili a tutti i processi (non è un segmento, il suo accesso non avviene attraverso selettori)
- LDT o Local Descriptor Table è una tabella di descrittori di segmenti visibili solo dal processo (task) a cui la LDT è associata
- IDT o Interrupt Descriptor Table è una tabella contenente i descrittori dei Gates di accesso alle procedure o ai task associati agli interrupt. La IDT corrisponde alla Interrupt Table dell'8086. La IDT può contenere soltanto descrittori di Gate (Interrupt, Trap e Task Gate).

Costruzione GDT

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL                # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG(STA_W, 0x0, 0xffffffff)          # data seg

gdtdesc:
  .word   0x17                                # sizeof(gdt) - 1
  .long   gdt                                 # address gdt
```

GDT

```
/*
 * Macros to build GDT entries in assembly.
 */
#define SEG_NULL                        \
    .word 0, 0;                         \
    .byte 0, 0, 0, 0
#define SEG(type,base,lim)                          \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)


// Application segment type bits
#define STA_X         0x8     // Executable segment
#define STA_W         0x2     // Writeable (non-executable segments)
#define STA_R         0x2     // Readable (executable segments)
#define STA_A         0x1     // Accessed
```

Registri di sistema

- Al fine di operare in modo efficiente sulle suddette tabelle la CPU IA-32 dispone dei seguenti registri:
  - GDTR o Global Descriptor Table Register, un registro inizializzato in sede di boot, che punta alla GDT della piattaforma
  - LDTR o Local Descriptor Table Register: un registro aggiornato ad ogni operazione di Task Switching, che punta alla LDT del Task Running
  - IDTR o Interrupt Descriptor Table Register: un registro inizializzato in fase di boot, che punta alla IDT della piattaforma
  - TR o Task Register: un registro che punta al TSS del Task Running

Registri di sistema

- Altri registri sono usati dal processore per controllare e gestire l'intero sistema, sono generalmente registri accessibili (almeno in parte) dal sistema operativo con istruzioni privilegiate
- I principali di questi registri sono:
  - EIP
  - EFLAGS
  - Control register (CR0, CR2, CR3, and CR4)
  - Segment Register

EFLAGS

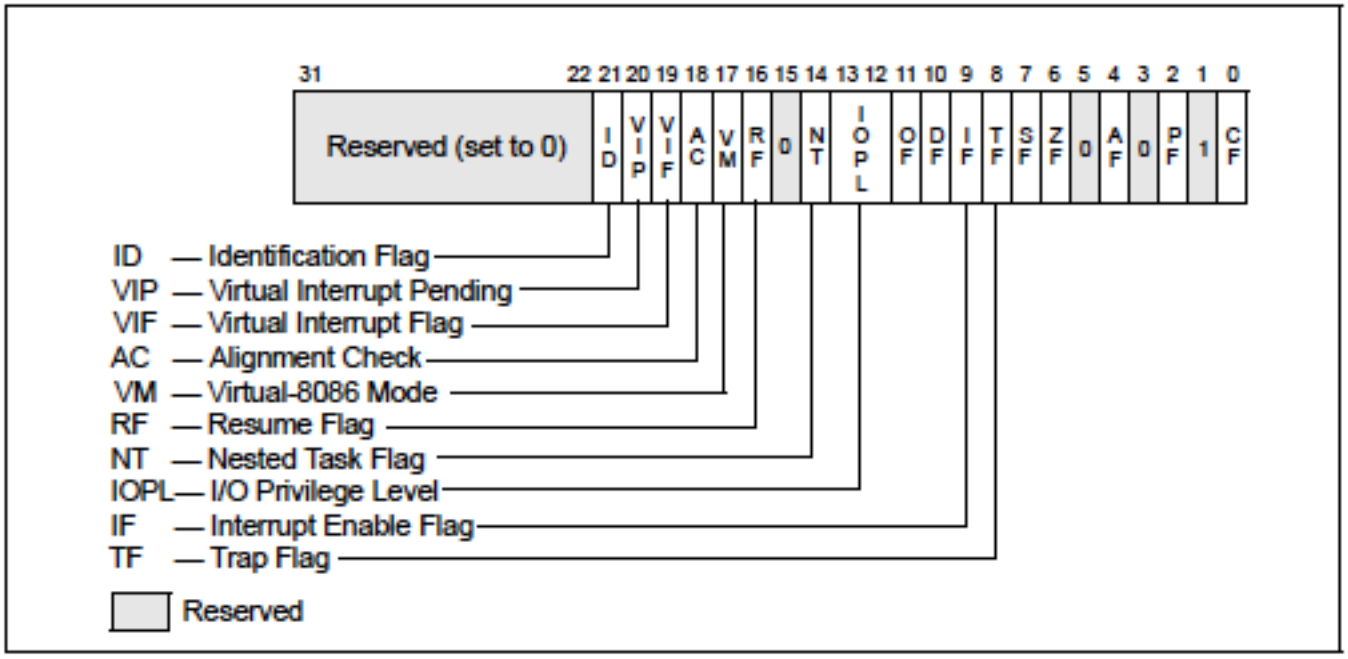


Figure 2-4. System Flags in the EFLAGS Register

Segment Register

- Sei 16-bit Segment Register
  - Servono per supportare la memoria segmentata, affinché un processo possa accedere al descrittore di un segmento è necessario che l'indirizzo di quel descrittore sia stato preventivamente caricato nel corrispondente registro
  - Sono usati come cache per i descrittori di segmento
  - I Segmenti contengono diverse tipologie di dati
    - Codice
    - Dati
    - Stack

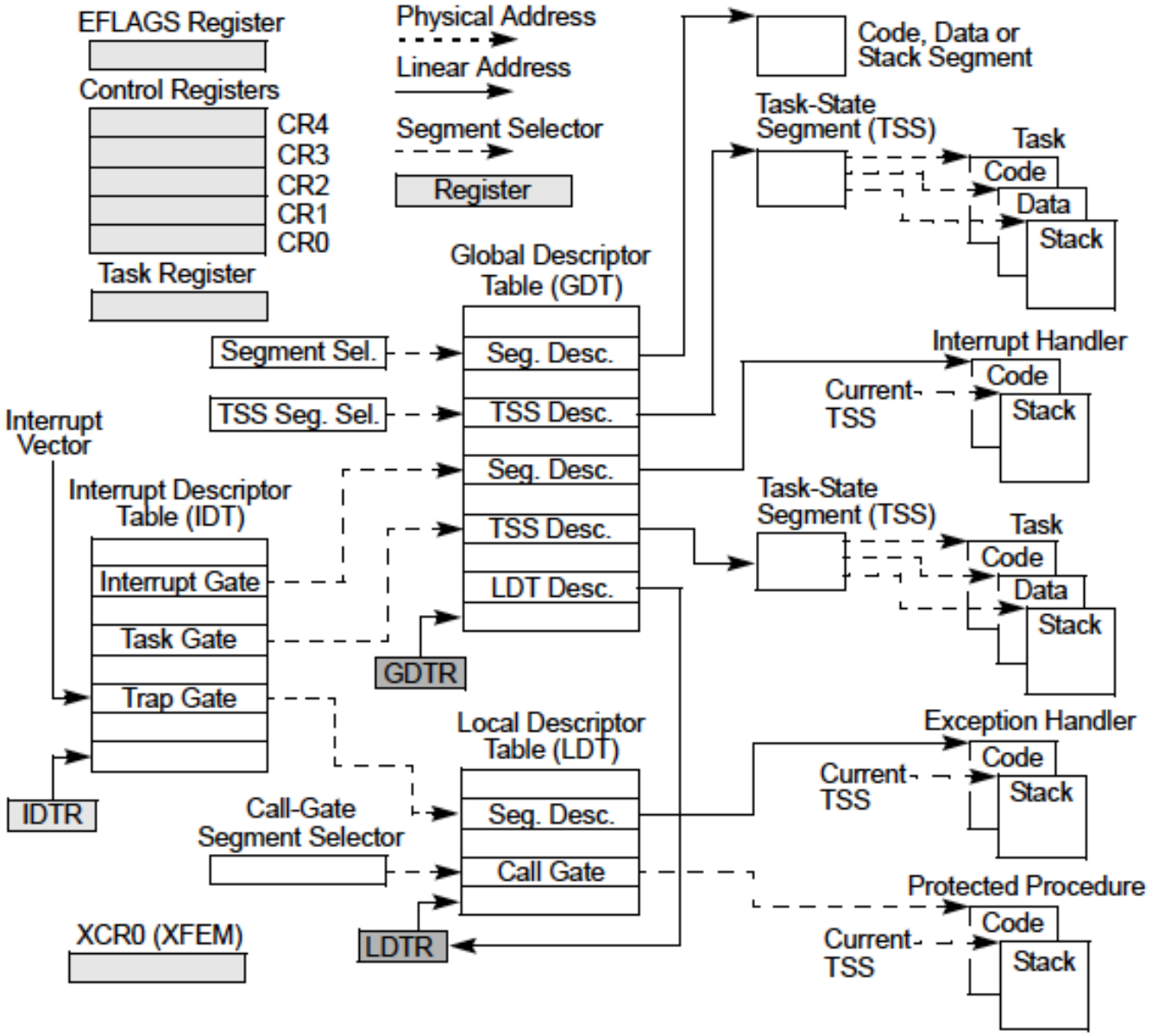
Segment Register

- Parte visibile = 16-bit Segment Register
  - CS, SS, DS, ES, FS, e GS sono visibili al programmatore
- Parte invisibile = Segment Descriptor (64 bits)
  - Caricato dall'hw

Visible part	Invisible part	
Segment selector	Segment base address, size, access rights, etc.	CS
Segment selector	Segment base address, size, access rights, etc.	SS
Segment selector	Segment base address, size, access rights, etc.	DS
Segment selector	Segment base address, size, access rights, etc.	ES
Segment selector	Segment base address, size, access rights, etc.	FS
Segment selector	Segment base address, size, access rights, etc.	GS

Segment Registers

- Segment registers hold the segment address of various items. They can only be set by a general register or special instructions. Some of them are critical for the good execution of the program
  - `CS : Holds the Code segment in which your program runs. Changing its value might make the computer hang.`
  - `DS: Holds the Data segment that your program accesses. Changing its value might give erronous data.`
  - `ES,FS,GS: These are extra segment registers available for far pointer addressing like video memory and such.`
  - `SS : Holds the Stack segment your program uses. Sometimes has the same value as DS. Changing its value can give unpredictable results, mostly data related.`


EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
Task Register
Physical Address
Linear Address
Segment Selector
Register
Code, Data or Stack Segment
Task-State Segment (TSS)
Task
Code
Data
Stack
Global Descriptor Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LDT Desc.
Interrupt Handler
Code
Stack
Current-TSS
Interrupt Vector
Interrupt Descriptor Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
GDTR
IDTR
Local Descriptor Table (LDT)
Call-Gate Segment Selector
Seg. Desc.
Call Gate
LDTR
Exception Handler
Protected Procedure
XCR0 (XFEM)

Registri General purpose

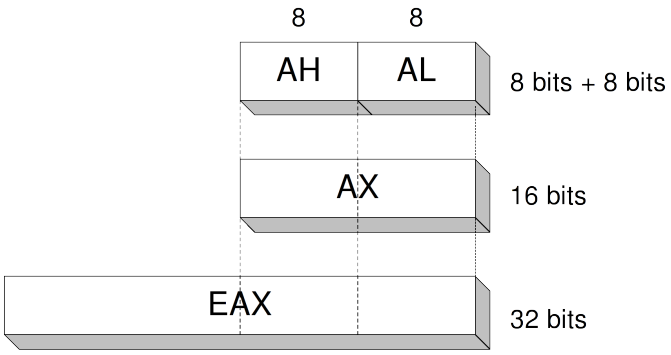
32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Accesso ai registri

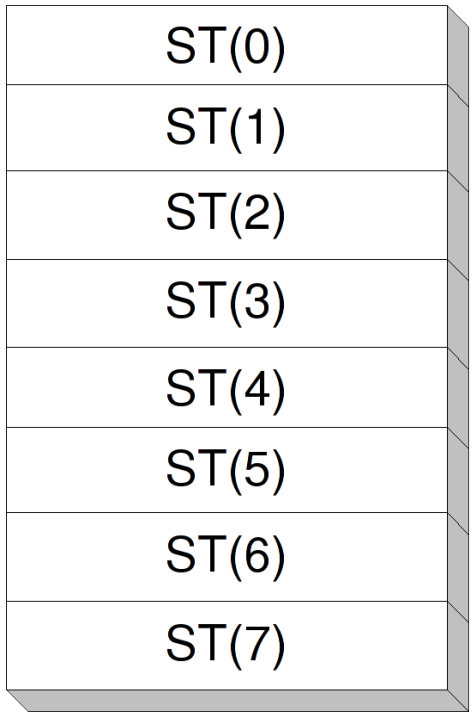
- EAX, EBX, ECX, e EDX sono registri a 32-bit
  - È possibile però accedere a soli 16-bit e 8-bit
  - I 16-bit meno significativi di EAX sono denotati con AX
  - AX è ulteriormente suddiviso
    - AL = 8 bit meno significativi
    - AH = 8 bit più significativi

- ESI, EDI, EBP, ESP: si può solo accedere ai 16 bit meno significativi



Registri Floating-Point, MMX, XMM

- Otto 80-bit floating-point data register
  - ST(0), ST(1), . . . , ST(7)
  - Usati da FPU
- Otto 64-bit MMX register
  - Usati dalle istruzioni MMX
- Otto 128-bit XMM registers
  - Usati dalle istruzioni SSE



Data Sizes

- Three main data sizes
  - Byte (b): 1 byte
  - Word (w): 2 bytes
  - Long (l): 4 bytes
- Separate assembly-language instructions
  - E.g., addb, addw, and addl

Declaring variables

- **.byte**
  - Bytes take up one storage location for each number. They are limited to
  - numbers between 0 and 255.
- **.word**
  - Ints (which differ from the **int** instruction) take up two storage locations for each number. These are limitted to numbers between 0 and 65535.9
- **.long**
  - Longs take up four storage locations. This is the same amount of space the registers use, which is why they are used in this program. They can hold numbers between 0 and 4294967295.
- **.ascii**
  - The **.ascii** directive is to enter in characters into memory. Characters each take up one storage location (they are converted into bytes internally). So, if you gave the directive **.ascii "Hello there\0"**, the assembler would reserve 12 storage locations (bytes).

Declaring Data

```
.section .data     # section declaration

msg:    .ascii "Introduci il numero:\n"   # our dear string
len = . - msg      # lunghezza messaggio
dieci:      .long  10
nrochar:    .word 0
zero:   .byte 0
num:    .long 0
num2:   .long 0
```

Little endian

- Intel is a little endian architecture
- Least significant byte of multi-byte entity is stored at lowest memory address: “Little end goes first”

- Es.: il dato 0x44332211 viene memorizzato
- If we display the memory dump of the same number 0x44332211 stored in memory at address 101 in Little-Endian order, we see something like this:
  - `ADDRESS: ---------- MEMORY BYTES ----------`
    `100: 00 11 22 33 44 00 00 00 00 00 ...`

Big endian

- Some other systems use big endian
- Most significant byte of multi-byte entity is stored at lowest memory address: “Big end goes first”

- Es.: data 0x44332211 is stored as
- If we display the memory dump of the same number 0x44332211 stored in memory at address 101 in Little-Endian order, we see something like this:
  - ```
    ADDRESS: ---------- MEMORY BYTES ----------
        100: 00 44 33 22 11 00 00 00 00 00 ...
    ```
    ```

GAS Instruction Format

- General format:
  - `[prefix] opcode operands`
- Prefix used only in String Functions
- Operands represent the direction of operands
  - Single operand instruction: **`opcode src`**
  - Two operand instruction : **`opcode src dest`**

Loading and Storing Data

- Data can be stored in:
  - Registers
  - Variables
- Variables are stored in memory
- Registers are “special” memory locations directly accessible by the processor
- The processor can only manipulate data inside registers
- The instruction to load from and store to memory, is `mov src,dest`

Accessing data

- Processors have many ways to access data known as “addressing modes”

- Register addressing: simply moves data in or out of a register
  - Example: `movl %edx, %ecx`
  - Copy value in register EDX into register ECX
  - Choice of register(s) embedded in the instruction

Immediate Addressing

- Immediate mode is used to load direct values into registers. For example, if you wanted to load the number 12 into **%eax**, you would simply do the following:

```
movl $12, %eax
```

- Notice that to indicate immediate mode, we used a dollar sign in front of the number. If we did not, it would be direct addressing mode, in which case the value located at memory location 12 would be loaded into **%eax** rather than the number 12 itself

Direct Addressing

- Load or store from a particular memory location
  - Memory address is embedded in the instruction
  - Instruction reads from or writes to that address
- `movl 2000, %ecx`
  - Four-byte variable located at address 2000
  - Read the four bytes value contained at location 2000
  - Load the value into the ECX register
- Can use a label for (human) readability
  - E.g. `movl  i,%eax`

Indirect Addressing

- Load or store from a previously-computed address
  - Register with the address is an operand in the instruction
  - Instruction reads from or writes to that address
- Example: `movl (%eax), %ecx`
  - EAX register stores a 32-bit address (e.g., 2000)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Dynamically allocated data referenced by a pointer
- The “(%eax)” essentially dereferences a pointer

Base pointer addressing

- Load or store with an offset from a base address
  - Register storing the base address
  - Fixed offset also embedded in the instruction
  - Instruction computes the address and does access
- Example: `movl 8(%eax), %ecx`
  - EAX register stores a 32-bit base address (e.g., 2000)
  - Offset of 8 is added to compute address (e.g., 2008)
  - Read long-word variable stored at that address
  - Load the value into the ECX register

Indexed Addressing Example

```
int a[20];
…
int i, sum=0;
for (i=0; i<20; i++)
    sum += a[i];
```

eax = ??
ebx = ??
ecx = ??

```
    movl $0, %eax
    movl $0, %ebx
sumloop:
    movl a(,%eax,4), %ecx
    addl %ecx, %ebx
    incl %eax
    cmpl $19, %eax
    jle sumloop
```

Summary

- Immediate addressing: data stored in the instruction itself
  - `movl $10, %ecx`
- Register addressing: data stored in a register
  - `movl %eax, %ecx`
- Direct addressing: address stored in instruction
  - `movl foo, %ecx`
- Indirect addressing: address stored in a register
  - `movl (%eax), %ecx`
- Base pointer addressing: includes an offset as well
  - `movl 4(%eax), %ecx`
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - `movl 2000(,%eax,1), %ecx`

Arithmetic Instructions

- Simple instructions
  - `add{b,w,l} source, dest` dest = source + dest
  - `sub{b,w,l} source, dest` dest = dest – source
  - `inc{b,w,l} dest` dest = dest + 1
  - `dec{b,w,l} dest` dest = dest – 1
  - `cmp{b,w,l} source1, source2` source2 – source1

Mul/Div

- Multiply
  - `mul (unsigned) or imul (signed)`
    - Performs signed multiplication and stores the result in the second operand. If the second operand is left out, it is assumed to be **%eax**, and the full result is stored in the double-word **%edx**:**%eax**
- Divide
  - `div (unsigned) or idiv (signed)`
    - Divides the contents of the double-word contained in the combined **%edx**:**%eax** registers by the value in the register or memory location specified. The **%eax** register contains the resulting quotient, and the **%edx** register contains the resulting remainder

Bitwise logic instructions

- Simple instructions
  - `and{b,w,l} source, dest` dest = source & dest
  - `or{b,w,l} source, dest` dest = source | dest
  - `xor{b,w,l} source, dest` dest = source ^ dest
  - `not{b,w,l} dest` dest = ~dest
  - `sal{b,w,l} source, dest` dest = dest << source
  - `sar{b,w,l} source, dest` dest = dest >> source

Control Flow

- We obtain contro flow using two instructions:

```
cmpl $0, %eax
je   end_loop
```

- The first one is the `cmpl` instruction which compares two values, and stores the result of the comparison in the status register EFLAGS. Notice that the comparison is to see if the second value is greater than the first
- The second one is the flow control instruction `JUMP` which says to jump to the end_loop depending on the values stored in the status register and on the condition expressed

Types of Jumps

- je: Jump if the values were equal
- jg: Jump if the second value was greater than the first value
- jge: Jump if the second value was greater than or equal to the first value
- jl: Jump if the second value was less than the first value
- jle: Jump if the second value was less than or equal to the first value
- jmp:Jump no matter what. This does not need to be preceeded by a comparison

Exercise

- Write a program which compute in %ecx the sum of the first 1000 natural numbers

Compiling & Linking

- To assembly the program type in the command

  as name.s -o name.o

- as is the command which runs the assembler, name.s is the source file, and –o name.o tells the assemble to put it's output in the file name.o which is an object file. An object file is code that is in the machine's language, but has not been completely put together.
- In most large programs, you will have several source files, and you will convert each one into an object file
- The linker is the program that is responsible for putting the object files together and adding information to it so that the kernel knows how to load and run it.
- To link the file, enter the command

  ld name.o -o name

Executing

- You can run the executable prog by typing in the command

  ./prog

- The ./ is used to tell the computer that the program isn't in one of the normal program directories, but is the current directory instead

Stack

- Many CPU's have built-in support for a stack A stack is a Last-In First-Out (LIFO) list
- The stack is an area of memory that is organized in this fashion. The PUSH instruction adds data to the stack and the POP instruction removes data
- The data removed is always the last data added
- The ESP register contains the address of the data that would be removed from the stack. This data is said to be at the top of the stack
- The processor references the SS register automatically for all stack operations. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.
- Data can only be added in double word units. That is, one can not push a single byte on the stack

Stack

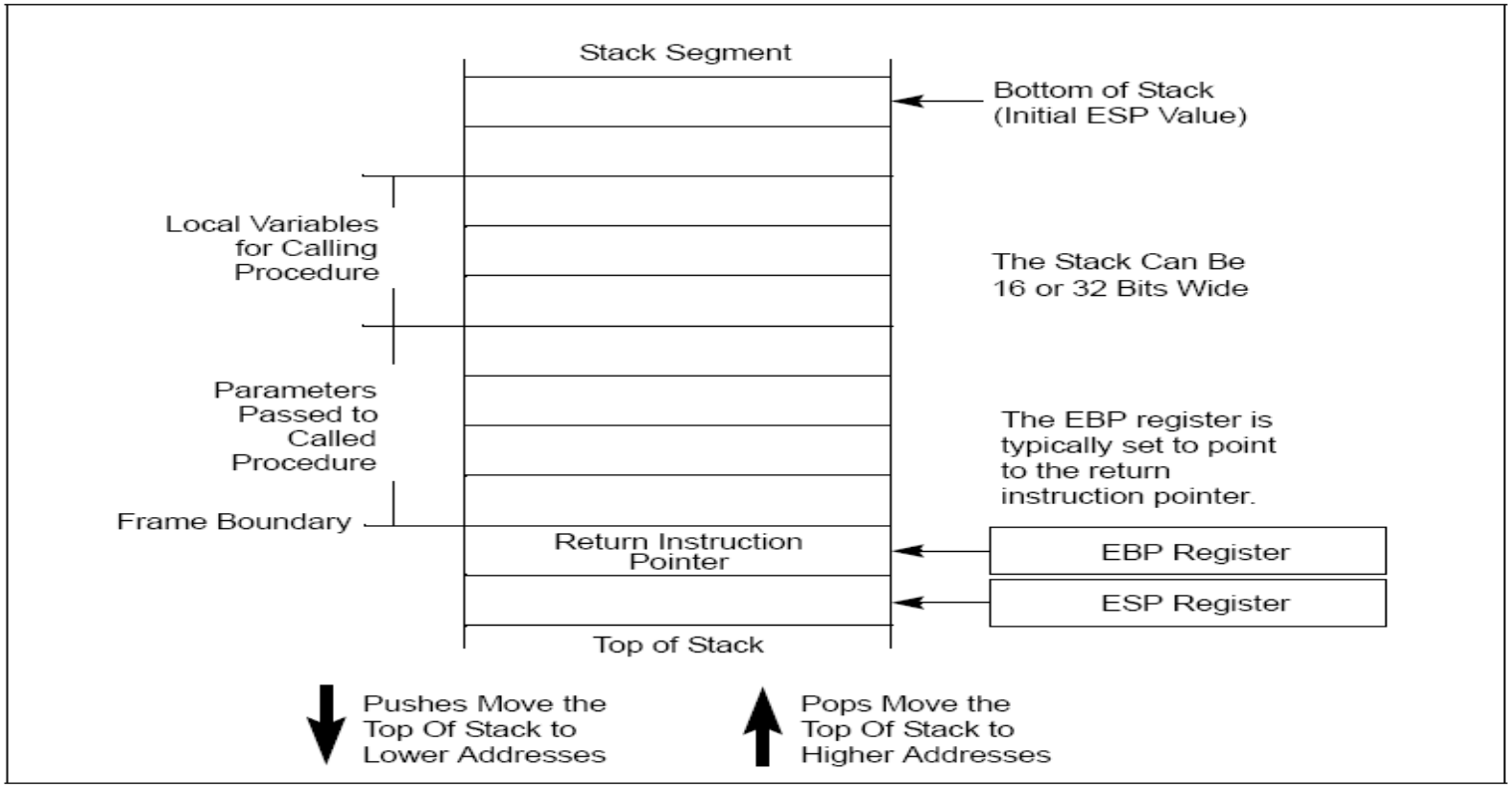


Figure 6-1. Stack Structure

PUSH

- The PUSH instruction inserts a double word on the stack by subtracting 4 from ESP and then stores the double word at [ESP]

```
pushl src   →      subl $4,%esp
                   movl src,(%esp)
```

- The 80x86 also provides a PUSHA instruction that pushes the values of EAX, EBX, ECX, EDX, ESI, EDI and EBP registers (not in this order)

POP

- The POP instruction reads the double word at [ESP] and then adds 4 to ESP

```
popl dest →   movl (%esp),dest
              addl $4,%esp
```

- The `popa` instruction, recovers the original values of the registers saved by the `pusha`

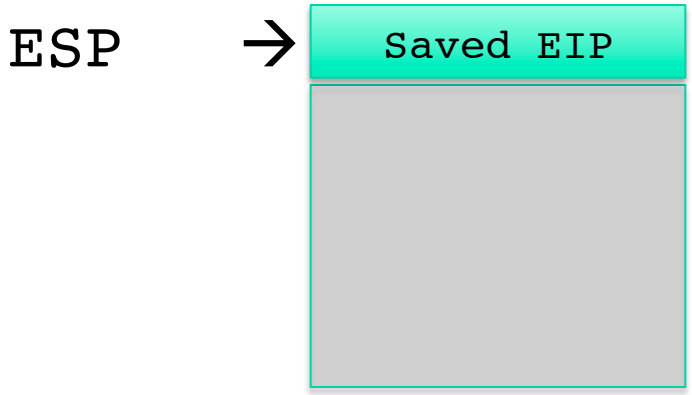
CALL/RET

- The 80x86 provides two instructions that use the stack to make calling subprograms quick and easy. The CALL instruction makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack.
- The RET instruction pops off an address and jumps to that address.
- When using these instructions, it is very important that one manage the stack correctly so that the right number is popped off by the RET instruction

Implementation of Call

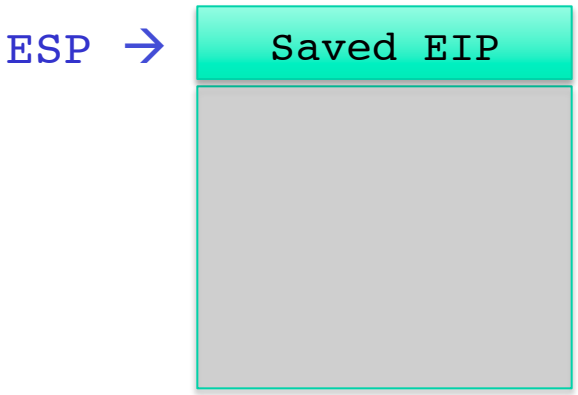
- `call subprogram1`
- becomes:

```
pushl   %eip
jmp     subprogram1
```



Implementation of ret

- `ret`
- `becomes:`
  - `pop  %eip`

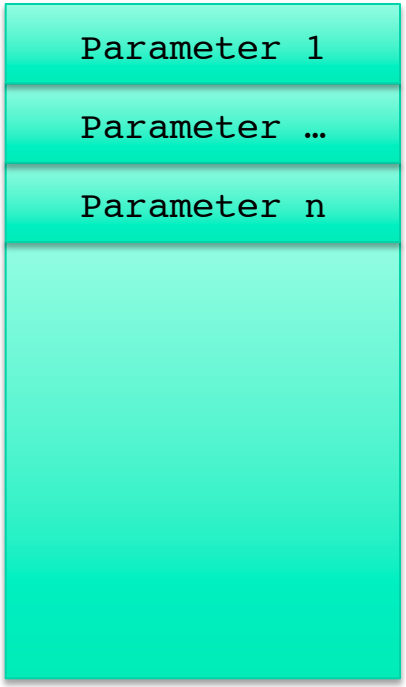


Passing Parameters

- How does caller function pass parameters to callee function?
- Attempted solution: Pass parameters in registers
  - Problem: Cannot handle nested function calls
  - Also: How to pass parameters that are longer than 4 bytes?
- Caller pushes parameters before executing the call instruction
- Parameters are pushed in the reverse order
  - Push the n-th parameter first
  - Push 1° parameter last

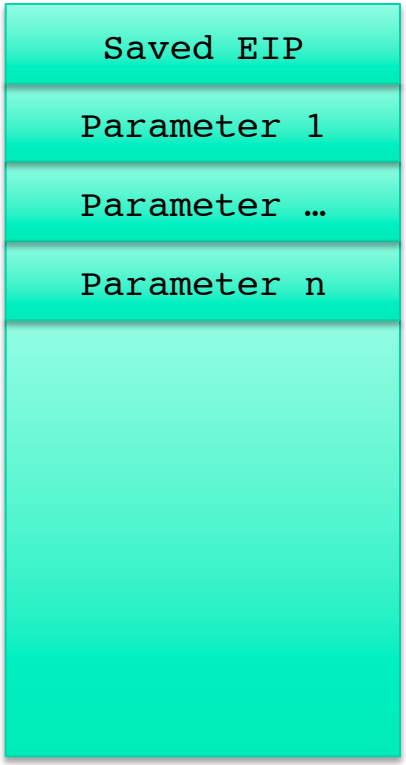
Parameters

ESP before → call



Parameter

ESP after → call



Callee addresses params relative to ESP:
Param 1 as 4(%esp)

Parameter

- After returning to the caller, the caller pops the parameters from the stack

```
sub:
# Push parameters       …
pushl $5              movl 4(%esp),var1
pushl $4              movl 8(%esp),var2
pushl $3              movl 12(%esp), var3
call sub              …
# Pop parameters        ret
addl $12, %esp
```

%ebp

- As callee executes, ESP may change
  - E.g., preparing to call another function
- It can be very error prone to use ESP when referencing parameters. To solve this problem, the 80386 supplies another register to use: EBP. This register's only purpose is to reference data on the stack
- Use EBP as fixed reference point to access params

Using EBP (prolog)

- A subprogram before overwriting ebp first save the old value of EBP on the stack and then set EBP to be equal to ESP. This allows ESP to change as data is pushed or popped off the stack without modifying EBP

```
pushl %ebp
movl  %esp, %ebp
(sub  Local_bytes, %esp)
```

- Regardless of ESP, the subprogram can reference param 1 as 8(%ebp), param 2 as 12(%ebp), etc.

Using ebp (epilog)

- Before returning, callee must restore ESP and EBP to their old values executing the epilog

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

Enter/Leave

- The ENTER instruction performs the prologue code and the LEAVE performs the epilogue
- The ENTER instruction takes two immediate operands.
- For the C calling convention, the second operand is always 0. The first operand is the number bytes needed by local variables. The LEAVE instruction has no operands

```
subprogram_label:
      enter  LOCAL_BYTES, 0      ; = # bytes needed by locals
; subprogram code
      leave
      ret
```