Software Security 05

Linguaggio macchina x86 e x86-64

Gianluca Amato
Università di Chieti-Pescara

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Il linguaggio macchina

Istruzioni x86 e x86-64

- Forniamo una breve introduzione al set di istruzioni assembly delle architetture x86 (32 bit) e x86-64 (64 bit).
- Descrizione breve (ma dettagliata) di tutte le istruzioni:
  - http://www.felixcloutier.com/x86/
- Un documento con le istruzioni più comuni:
  - https://web.stanford.edu/class/archive/cs/cs107/cs107.1166/onepage_x86-64.pdf
- Manuali esaustivi sulle due architetture:
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
  - …che Intel chiama *Intel 64* e *IA-32*

Registri (1)

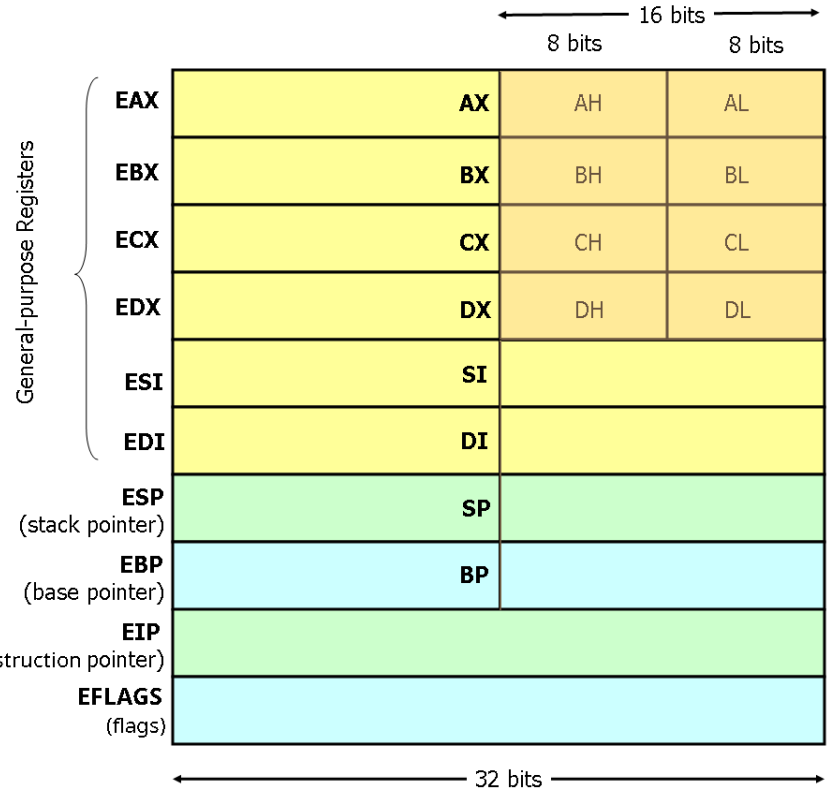
- Tutte le CPU hanno un certo numero di registri.
- Un registro è come una variabile in un linguaggio di programmazione.
- Le seguenti operazioni sono possibili sui registri:
  - Leggere un valore dalla memoria in un registro.
  - Scrivere il valore di un registro in memoria.
  - Eseguire operazioni aritmetiche o logiche sui valori dei registri.

Registri (2)

- Alcune istruzioni della CPU operano solo su registri, altre possono operare o sui registri o sulla memoria.
- In ogni caso, operare sui registri è molto più veloce, perché i registri sono interni alla CPU.

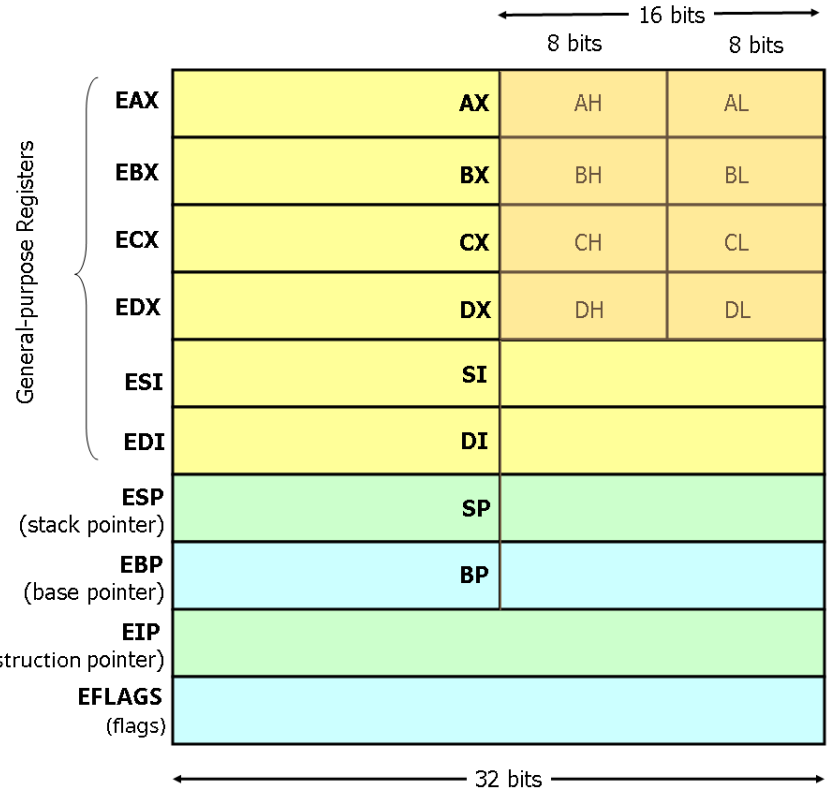
Registri x86 (1)

- Registri di uso generale
  - Possono essere usati in maniera interscambiabile.
- EIP (*Instruction Pointer*)
  - Indirizzo dell'istruzione in corso di esecuzione.
- EFLAGS
  - Una collezione di bit, che forniscono informazioni sullo stato della CPU.
    - Ad esempio, se l'ultima operazione aritmetica ha dato risultato zero
- EBP, ESP (*Base Pointer*, *Stack Pointer*)
  - Registri per la gestione dello stack.



Registri x86 (2)

- Alcuni “frammenti” dei registri sono accessibili con altri nomi.
  - AX: 16 bit meno significativi di EAX
  - AL: 8 bit meno significativi di AX
  - AH: 8 bit più significativi di AX
- La E iniziale sta per “Extended”



Registri x86 (3)

- Esistono in realtà molti altri registri speciali:
  - Per uso esclusivo del sistema operativo.
  - Per il calcolo in virgola mobile.
  - Per il calcolo parallelo.
- Noi però non ci avremo a che fare.

Registri x86-64 (1)

- I registri Exx diventano a 64 bit e si chiamano Rxx
  - il vecchio nome Exx continua ad essere valido, e si riferisce ai 32 bit meno significativi.
  - vengono aggiunti 8 nuovi registri da R8 ad R15

Registri x86-64 (2)

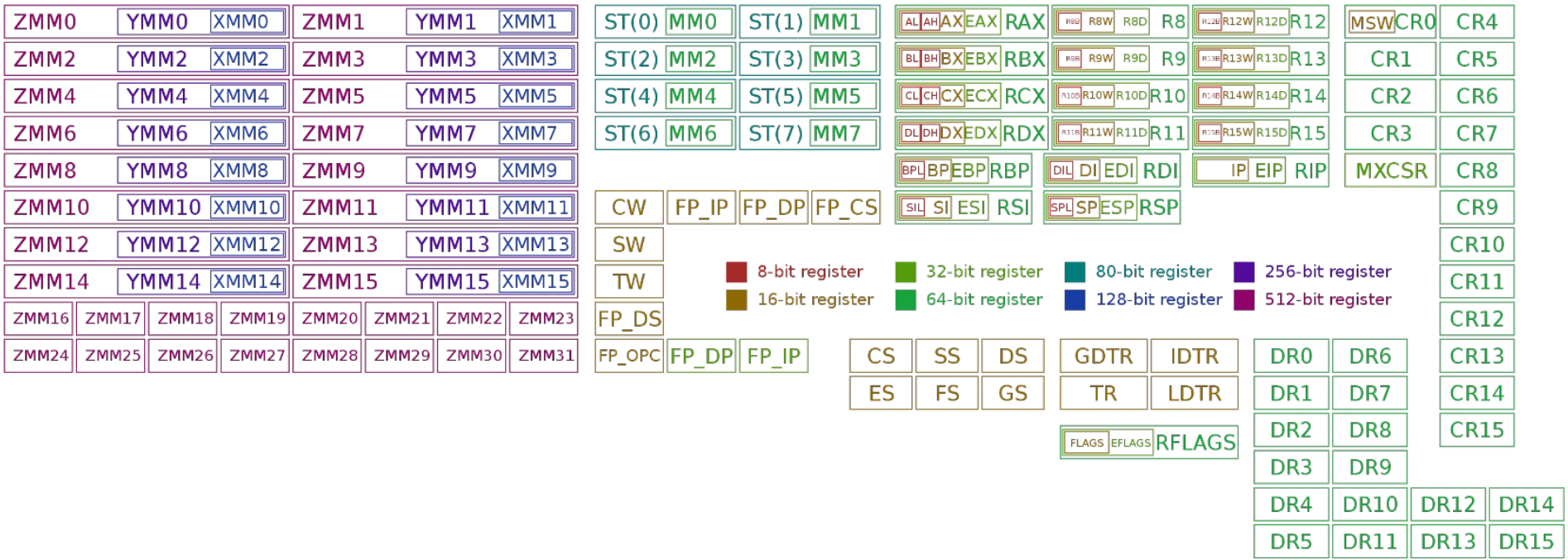
Not modified for 8-bit operands

Not modified for 16-bit operands

Register encoding	Zero-extended for 32-bit operands			Low 8-bit	16-bit	32-bit	64-bit
0			AH†	AL	AX	EAX	RAX
3			BH†	BL	BX	EBX	RBX
1			CH†	CL	CX	ECX	RCX
2			DH†	DL	DX	EDX	RDX
6				SIL‡	SI	ESI	RSI
7				DIL‡	DI	EDI	RDI
5				BPL‡	BP	EBP	RBP
4				SPL‡	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15
	63 32	31 16	15 8	7 0			

Registri x86-64 (3)

- In realtà, le cose sono più complesse…



Istruzioni (1)

- Operazione di spostamento dati
  - `mov op1, op2`
    - copia in `op1` il valore di `op2`
    - ma cosa sono gli operandi `op1` e `op2` ?
- La CPU supporta vari metodi di indirizzamento
  - Ovvero dei meccanismi per specificare un operando in una istruzione.
  - Non tutti i metodi di indirizzamento sono validi in tutte le istruzioni.

Metodi di indirizzamento (1)

- Registro
  - `mov rax, rbx`
    - Copia in `rax` il valore del registro `rbx`
    - `rax=rbx`
- Immediato
  - `mov rax, 183`
    - Copia in `rax` il valore 183
    - `rax=183`

Possibile interpretazione in C

Metodi di indirizzamento (2)

- Diretto
  - `mov rax, [292384]`
    - Copia in `rax` il contenuto delle celle di memoria a partire dall'indirizzo 292384.
    - Il numero di byte da copiare dipende dalla dimensione della destinazione
    - Poiché `rax` è un registro a 64bit, verranno copiati 8 byte.
  - `mov [292384],rax`
    - Come sopra, ma con sorgente e destinazione invertite
    - Copia in memoria all'indirizzo 292384 il valore del registro `rax`.
  - Normalmente in un programma in assembly invece di un numero c'è una etichetta che identifica quella locazione.
    - `mov rax,[etichetta]` / `mov [etichetta],rax`
    - `rax=etichetta` / `etichetta=rax`

Metodi di indirizzamento (3)

- Indiretto
  - `mov rax,[rbx]`
    - Copia in `rax` il contenuto delle celle di memoria a partire dall'indirizzo in `rbx`.
    - Il registro `rbx` svolge la funzione di puntatore.
    - `rax=*rbx`
- Indiretto con offset e indice scalato
  - `mov rax,[rbx+rsi*8+12]`
  - Copia in `rax` il contenuto delle celle di memoria a partire dall'indirizzo in `rbx+rsi*8+` 12.
  - Alcune componenti possono mancare.
  - I metodi diretto e indiretto si possono vedere come casi particolare di questo.
  - Il fattore di scala (8 in questo caso) deve essere una potenza di 2.

Istruzioni (2)

- Operazioni aritmetiche e logiche
  - `add op1,op2`: memorizza in `op1` il risultato di `op1+op2`
    - `op1 += op2`
  - `sub op1,op2`: memorizza in `op1` il risultato di `op1-op2`
    - `op1 -= op2`
  - `and op1,op2`
    - `op1 &= op2`
  - `or op1,op2`
    - `op1 |= op2`
  - `xor op1,op2`
    - `op1 ^= op2`
  - ....

Operazioni bit a bit.

Esempio: 12 and 23 = 4

$12 = 1100_2$

$23 = 10111_2$

$00100_2 = 4$

Istruzioni di salto

- `jmp op`
  - Salto incondizionato
  - Salta alla locazione di memoria indicata da op
- `cmp op1, op2`
  - Confronta `op1` e `op2`
  - Non altera nessuno dei due, ma modifica il registro flag.
- `j<condition> op`
  - Salto condizionato
  - Se la condizione è vera salta all'indirizzo `op`, altrimenti non fai nulla
  - La condizione viene controllata sulla base del registro flag, e si riferisce di solito all'ultima istruzione `cmp` o logico/aritmetica eseguita.
  - Esempi di condizioni:
    - `je`: salta se i due valori confrontati erano uguali
    - `jz`: salta se l'ultima operazione aritmetica ha dato come risultato zero

Altre istruzioni

- `nop`
  - Non fa nulla
- `syscall`
  - Esegue una chiamata al sistema operativo (solo 64 bit)
    - Chiediamo al sistema operativo di svolgere un lavoro per noi
      - Visualizzare qualcosa su schermo, aprire un file, etc…
    - Bisogna preventivamente riempire i registri con valori opportuni a seconda di cosa stiamo chiedendo di fare
    - Lista delle chiamate di sistema: https://syscalls.mebeim.net/
    - Documentazione chiamate: comando `man` o sito https://man7.org/linux/man-pages
- `int 0x80`
  - Equivalente di syscall per le CPU a 32 bit.

Esempio di programma in LM

- Legge 10 caratteri e li visualizza in ordine inverso.

reverse.asm

```
section .bss
    input_buffer  resb 10     ; space for 10 bytes
    reversed      resb 10     ; space for reversed bytes

section .text
    global _start

_start:
    ; Read 10 bytes from stdin (fd = 0)
    mov rax, 0          ; syscall number: sys_read
    mov rdi, 0          ; file descriptor: stdin
    mov rsi, input_buffer
    mov rdx, 10         ; number of bytes to read
    syscall

    ; rax now contains the number of bytes actually read
    mov rcx, rax        ; rcx = number of bytes read
    dec rcx             ; rcx = index of last byte
    xor rbx, rbx        ; rbx = index in reversed buffer
                        ; this is equivalent to "mov rbx, 0"
```

```
reverse_loop:
    mov al, [input_buffer + rcx]
    mov [reversed + rbx], al
    inc rbx
    dec rcx
    cmp rcx, -1
    jg reverse_loop

    ; Write reversed buffer to stdout (fd = 1)
    mov rax, 1          ; syscall number: sys_write
    mov rdi, 1          ; file descriptor: stdout
    mov rsi, reversed
    mov rdx, rbx        ; number of bytes to write
    syscall

    ; Exit
    mov rax, 60         ; syscall number: sys_exit
    xor rdi, rdi        ; status = 0
    syscall
```

Come compilare il programma

- Il programma funziona solo su Linux su CPU x86-64.
- È necessario installare l'assemblatore NASM.
- Procedura:
  - Salvare il sorgente nel file `reverse.asm`
    - (il file lo trovate anche su Teams)
  - `nasm -f elf64 reverse.asm -o reverse.o`
  - `ld reverse.o -o reverse`

Sintassi Intel vs AT&T

- Esistono in realtà due sintassi per l'assembly delle CPU x86
  - Sintassi Intel (quella che abbiamo vista)
  - Sintassi AT&T
    - Usata dall'assemblatore GAS (GNU Assembler) di default su Linux
      - motivo per cui abbiamo usato NASM
    - È una sintassi "standard" che tenta di unificare le convenzioni di linguaggi assembly diversi.
- Nel mondo Windows è usata esclusivamente la sintassi Intel.
- Nel mondo Unix e derivati si usano invece entrambe e prevalentemente quella AT&T.
- Dovremo imparare a convivere con entrambe.

Differenze della sintassi AT&T

- L'ordine di sorgente e destinazione è invertito
- Le istruzioni hanno un suffisso che identifica la dimensione dei dati su cui operano. Il suffisso si può omettere se si capisce dal contesto.
  - `b` (byte, 8 bit)
  - `w` (word, 16 bit)
  - `l` (long, 32 bit)
  - `q` (quad word, 64 bit)
- I registri vengono scritti con un % prima del nome
  - `movq %rax, %rbx` (copia il contenuto del registro `rax` in `rbx`)
- I valori immediati vengono scritti con $ prima del numero
  - `movq $5, %rax` (mette il numero 5 nel registro `rax`)
- L'indirizzamento in memoria usa una notazione diversa:
  - `[rbx + rsi * 8 + 12]` diventa `12(%rbx, %rsi, 8)`

Demo

Svolgere la challenge

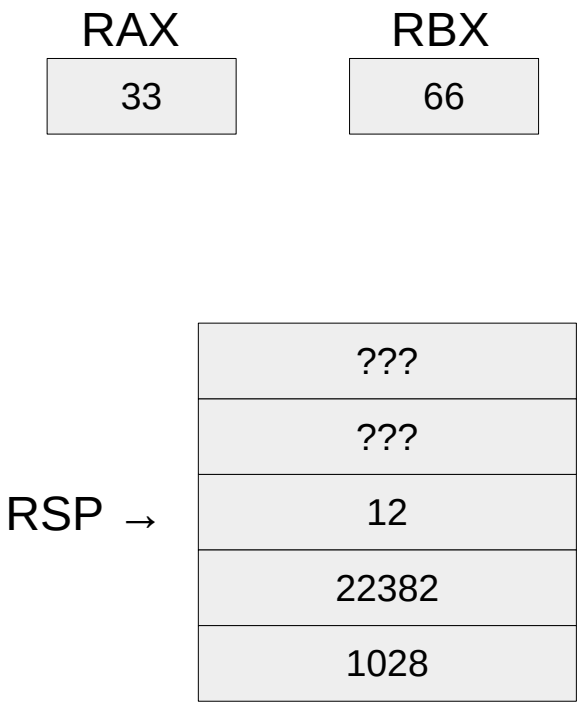
SS_1.02 – Slow Printer

Lo stack (1)

- Una parte della memoria della CPU è lo stack.
  - Lo stack cresce da indirizzi più grandi ad indirizzi più piccoli.
  - Il registro RSP (*stack pointer*) contiene l'indirizzo dell'ultimo dato inserito nello stack.
- Istruzioni che manipolano lo stack:
  - `push op`
    - Mette nello stack il valore op
  - `pop op`
    - Mette in op il valore in cima allo stack, che viene rimosso dallo stack

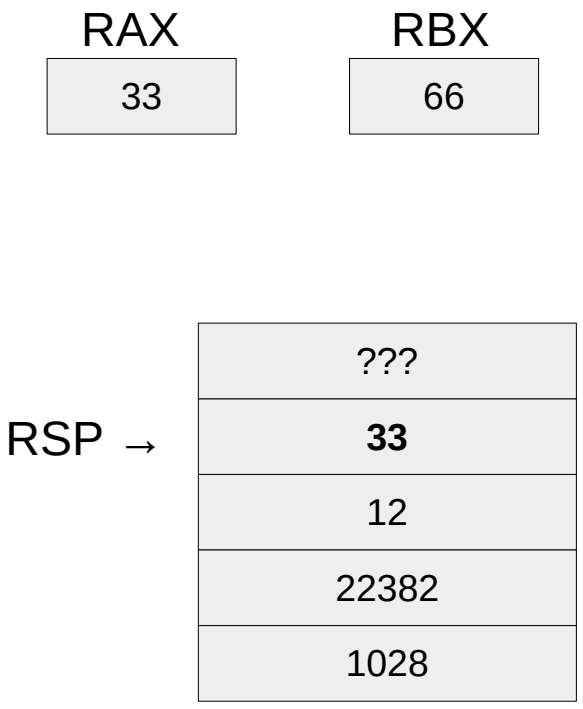
Lo stack (2)

- Esempio:
  - `push rax`
  - `push rbx`
  - `pop rax`
  - `pop rbx`



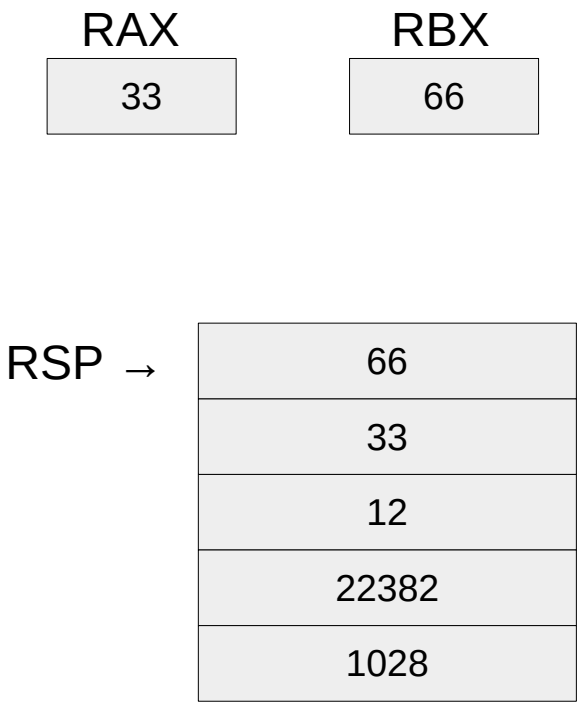
Lo stack (2)

- Esempio:
  - `push rax`
  - `push rbx`
  - `pop rax`
  - `pop rbx`



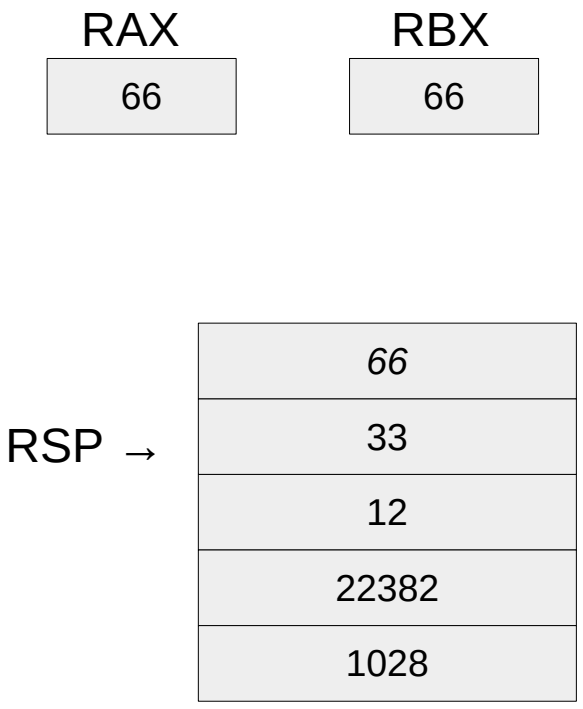
Lo stack (2)

- Esempio:
  - `push rax`
  - `push rbx`
  - `pop rax`
  - `pop rbx`



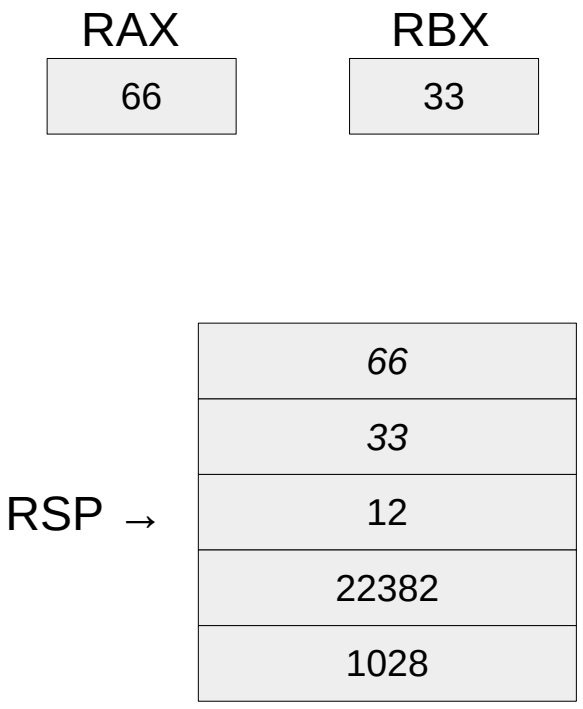
Lo stack (2)

- Esempio:
  - `push rax`
  - `push rbx`
  - `pop rax`
  - `pop rbx`



Lo stack (2)

- Esempio:
  - `push rax`
  - `push rbx`
  - `pop rax`
  - `pop rbx`



Chiamata di funzione

- L'implementazione del concetto di funzione è realizzata tramite la seguente coppia di istruzioni:
  - `CALL op`
    - È come `JMP` ma salva l'indirizzo dell'istruzione successiva nello stack.
  - `RET`
    - Estrae dallo stack l'indirizzo della prossima istruzione da eseguire.
- Notare che non c'è un meccanismo per il passaggio dei parametri
  - I parametri devono essere passati in altro modo.
  - Normalmente tramite registri o nello stack stesso (vedremo tra un attimo).

```
…
mov rax, 5
call func
call func
…           ; rax ora vale 7

func:
  inc rax   ; incrementa rax di 1
  ret
```

Compilazione del C in LM

Esempio di programma C

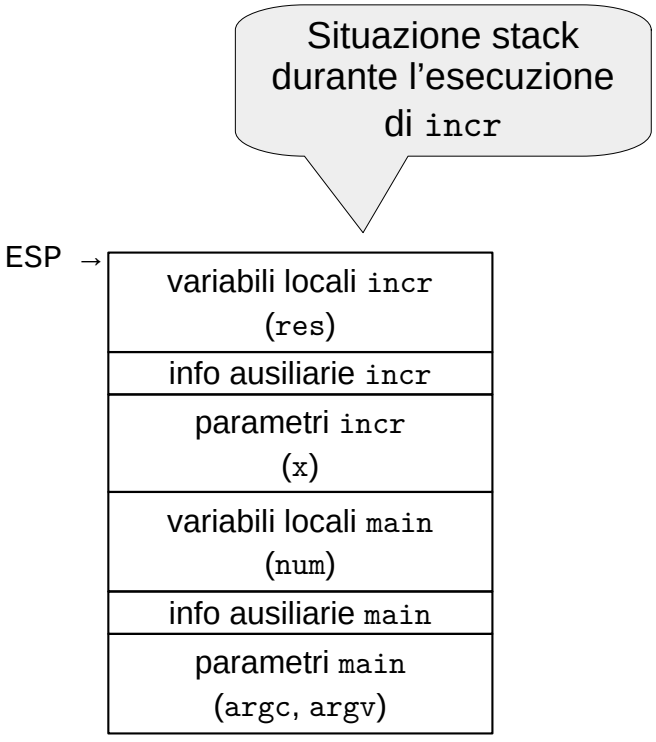
esempio.c

```c
#include <stdio.h>

/* Legge un numero intero e lo stampa
   incrementato di 1. */

int incr(int x) {
    int res = x+1;
    return res;
}

void main() {
    int num;
    scanf("%d", &num);
    num = incr(num);
    printf("%d\n", num);
}
```

- Vediamo come viene compilato il codice.
- Questa volta parleremo prima dei sistemi a 32 bit e solo dopo di quelli a 64 bit.

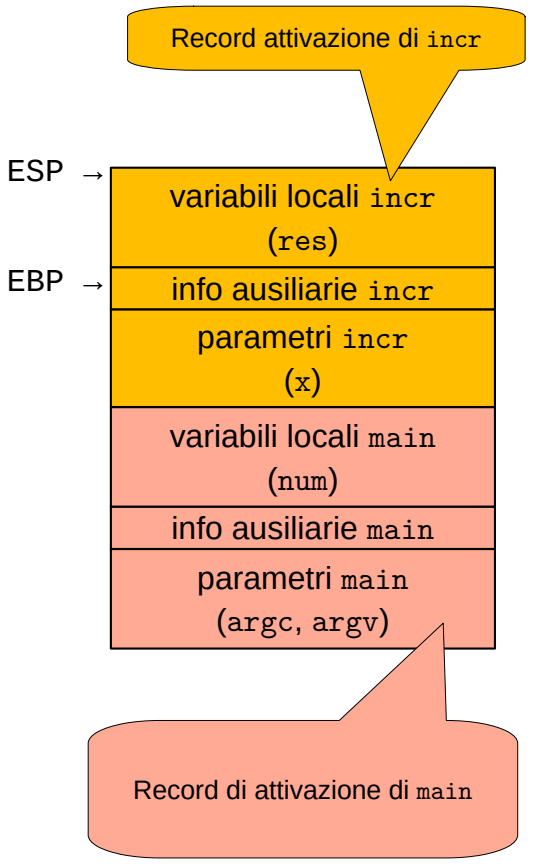
Record di attivazione (1)

- Vi ricordo che ogni volta che una funzione C viene invocata, si crea un record di attivazione.
- I record di attivazione vengono salvati nello stack.
- Ma come fa la CPU a “trovare le variabili locali e i parametri” ?
  - Dovrebbe conoscerne gli indirizzi, ma sono imprevedibili perché lo stack non si trova in memoria ad indirizzi fissi !



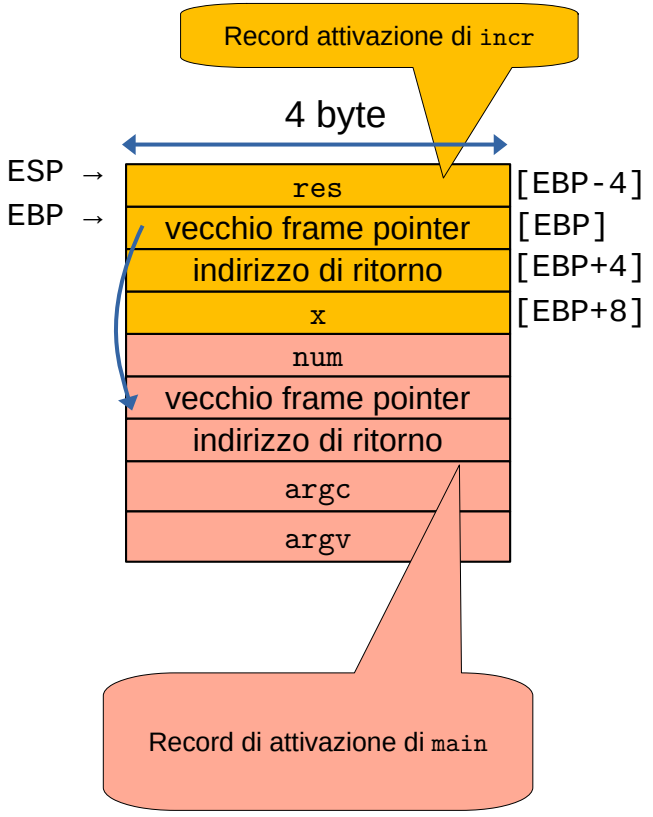
Record di attivazione (2)

- Quando una funzione viene lanciata, il registro EBP (*base pointer*) viene impostato alla sezione “info ausiliarie” del record di attivazione.
  - Da lì, variabili locali e parametri sono accessibili come offset a partire da EBP.
  - In questo contesto, il registro EBP viene anche chiamato *frame pointer*
    - perché *stack frame* è un nome alternativo di record di attivazione.



Record di attivazione (3)

- Cosa sono le “info ausiliarie” ?
  - L’indirizzo a cui ritornare l’esecuzione alla fine della funzione.
  - Il vecchio valore del base pointer per la funzione precedente, in modo da poterlo ripristina prima di uscire dalla funzione.
- La parte iniziale del codice di una funzione costruisce inizializza il record di attivazione.
- La parte finale del codice di una funzione distrugge il record di attivazione e prepara il terreno all’istruzione `RET`.



Convenzioni di chiamata (1)

- Le regole che governano
  - come passare i parametri;
  - come passare il valore di ritorno di una funzione;
  - quali registri le funzioni possono modificare impunemente;
- prendono il nome di *convenzioni di chiamata* (*calling convention*)
  - Vedi https://wiki.osdev.org/Calling_Conventions per un breve riassunto delle convenzioni di chiamata più comuni.

Convenzioni di chiamata (2)

- Riassumendo, le convenzioni di chiamata per Linux su x86 prevedono:
  - Gli argomenti sono passati nello stack.
  - Il chiamante inserisce gli argomenti nello stack dall'ultimo al primo.
  - Viene eseguita la chiamata di funzione.
  - Al ritorno, il chiamate elimina gli argomenti dallo stack
  - Il valore di ritorno si trova nel registro `eax` (di solito...)

Esempio di programma compilato

esempio.s

```
main:
        push ebp
        mov  ebp, esp
        sub  esp, 4
        lea  eax, [ebp-4]
        push eax
        push OFFSET FLAT:.LC0
        call __isoc99_scanf
        add  esp, 8
        mov  eax, DWORD PTR [ebp-4]
        push eax
        call incrementa
        add  esp, 4
        mov  DWORD PTR [ebp-4], eax
        mov  eax, DWORD PTR [ebp-4]
        push eax
        push OFFSET FLAT:.LC1
        call printf
        add  esp, 8
        nop
        leave
        ret
```

```
incr:
        push ebp
        mov  ebp, esp
        sub  esp, 4
        mov  eax, DWORD PTR [ebp+8]
        add  eax, 1
        mov  DWORD PTR [ebp-4], eax
        mov  eax, DWORD PTR [ebp-4]
        leave
        ret
```

```
.LC0:
        .string   "%d"
.LC1:
        .string   "%d\n"
```

Il codice è leggermente semplificato rispetto a quello generato da gcc.

Le istruzioni sono le stesse, ma alcune direttive non rilevanti sono state rimosse

Demo

Esecuzione passo-passo del programma compilato.

Esempio di programma compilato con -O

Compilato senza `-O`

```
incr:
     push ebp
     mov  ebp, esp
     sub  esp, 4
     mov  eax, DWORD PTR [ebp+8]
     add  eax, 1
     mov  DWORD PTR [ebp-4], eax
     mov  eax, DWORD PTR [ebp-4]
     leave
     ret
```

Compilato con `-O1`

```
incr:
     mov  eax, DWORD PTR [esp+4]
     add  eax, 1
     ret
```

Caso a 64 bit

- Tutti i puntatori e i registri coinvolti diventano a 64 bit.
- Gli argomenti non si trovano tutti nello stack.
  - I primi 6 argomenti vengono passati nei registri `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - Gli altri eventualmente sono posti nello stack.

Demo

Svolgere la challenge

SS_2.02 - 1996

Gianluca Amato
Università di Chieti-Pescara

Software Security 05

Linguaggio macchina x86 e x86-64

FINE

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it