Parte I

Forme di codice

Tutto si comprende tramite confronto

Proverbio russo, autore sconosciuto

Capitolo 1

Italian text placeholder

Quando l'autore di questo libro ha cominciato ad imparare il C e, successivamente, , era solito scrivere piccoli pezzi di codice, compilarli e guardare l'output prodotto in linguaggio assembly. Questo procedimento ha facilitato la comprensione del comportamento del codice che aveva scritto.

[1]. Lo ha fatto talmente tante volte che la relazione tra il codice e ciò che viene prodotto dal compilatore si è impressa profondamente nella sua mente. E' facile immaginare a colpo d'occhio un contorno della forma e della funzione di un dato codice C. Magari questa tecnica può rivelarsi utile anche per gli altri.

In queso libro sono talvolta utilizzati vecchi compilatori al fine di ottenere il frammento di codice più piccolo (o semplice) possibile.

Quando l'autore di questo libro studiava il linguaggio assembly, era solito anche compilare piccola funzioni C e riscriverle gradualmente in assembly tentando di restringere il codice il più possibile. Questa pratica è oggi probabilmente inutile in uno scenario reale, in quanto è molto difficile competere, in termini di efficienza, con i moderni compilatori. Rappresenta comunque un ottimo modo di acquisire una migliore conoscenza dell'assembly. Sentitevi quindi liberi di prendere qualunque pezzo di codice assembly da questo libro e cercare di renderlo più piccolo. Tuttavia non dimenticate di testare il vostro risultato.

Livelli di ottimizzazione e informazioni di debug

Il codice sorgente può essere compilato da compilatori diversi e con vari livelli di ottimizzazione. Un compilatore tipico ne prevede solitamente tre, dei quali il livello zero corrisponde a nessuna ottimizzazione (ottimizzazione disabilitata)

L'ottimizzazione può essere orientata verso la dimensione del codice o la sua velocità di esecuzione. Un compilatore non ottimizzante è più veloce e produce codice più comprensibile (sebbene prolisso), mentre un compilatore ottimizzante è più lento e cerca di produrre codice più veloce in termini di performance (ma non necesariamente più compatto). Oltre a livelli e direzione di ottimizzazione, un compilatore può includere informazioni di debug nel file risultante, producendo quindi codice che può essere debuggato più facilmente. Una delle caratteristiche più importanti del codice di ´debug' è che può contenere collegamenti tra ogni riga del codice sorgente e l'indirizzo del corrispondente codice macchina. I compilatori ottimizzanti tendono invece a produrre output in cui intere righe di codice sorgente possono essere ottimizzate a tal punto da non essere neanche presenti nel codice macchina risultante. I reverse engineers possono incontrare entrambe le version, semplicemente perchè alcuni sviluppatori utilizzano le opzioni di ottimizzazione dei compilatori ed altri no. A causa di ciò negli esempi proveremo, quando possibile, a lavorare sia sulle versioni di debug che su quelle di release del codice illustrato in questo libro.

Italian text placeholderAlcune basi teoriche

[1] In effetti lo fa ancora oggi quando non riesce a capire cosa fa un particolare pezzo di codice.

Capitolo 2

Una breve introduzione alla CPU

La CPU[1] è il dispositivo che esegue il codice macchina di cui è fatto un programma.

Un breve glossario:

Istruzione : Un comando CPU primitivo. L'esempio piò semplice include: spostare dati da un registro all'altro, lavorare con la memoria, effettuare operazioni aritmetiche primitive. Di norma ogni CPU ha il suo insieme di istruzioni, detto instruction set architecture o (ISA).

Codice macchina : Codice che la CPU è in grado di processare direttamente. Ciascuna istruzione è solitamente codificata da diversi byte.

Linguaggio Assembly : Codici mnemonico ed alcune estensioni come le macro introdotti per facilitare la vita del programmatore.

Registro CPU : Ogni CPU ha un insieme finito di registri generici (GPR). ≈ 8 in x86, ≈ 16 in x86-64, ≈ 16 in ARM. Il modo più semplice per capire un registro è quello di pensare ad esso come una variabile temporanea senza tipo. Immagina se stessi lavorando con un linguaggio di programmazione di alto livello PL[2] e potessi usare solo otto variabili a 32 (o 64) bit.

Eppure solo con questi si possono fare un sacco!

Qualcuno potrebbe chiedersi perchè ci debba essere una differenza tra il codice macchina ed un acPL. La risposta risiede nel fatto che gli umani e i processori (CPU) non sono uguali—per un umano è molto più facille utilizzare un linguaggio di programmazione ad alto livello come , Java, Python, etc., mentre per una CPU è più semplice utilizzare un livello di astrazione più basso. Potrebbe essere forse possibile inventare una CPU in grado di eseguire codice di un linguaggio PL ad alto livello, ma sarebbe molto più complesso dei processori che conosciamo oggi. Allo stesso modo sarebbe del tutto sconveniente per gli umani scrivere in linguaggio assembly, a causa della sua natura di basso livello e la difficoltà nello scrivere senza commettere un enorme numero di fastidiosi errori. Il programma che converte il codice di un PL di alto livello in assembly è detto un *compilatore*. [3].

2.1 Qualche parola sulle diverse ISA

L'architettura x86 ISA è sempre stata una con opcodes di lunghezza variabile, e all'arrivo dell'era 64-bit l'estensione x64 non ha avuto impatti significativi sulla ISA. Infatti l'architettura x86 contiene molte istruzioni apparse per la prima volta nelle CPU a 16-bit 8086 che si trovano ancora nei processori odierni. ARM è una CPU RISC[4] progettata con opcodes di lunghezza costante, che in passato avevano alcuni vantaggi. Originariamente tutte le istruzioni ARM erano codificate in 4 byte[5]. Oggi è noto come «ARM mode». Successivamente si pensò che non fosse poi tanto economico come si immaginava inizialmente. Infatti, le istruzioni CPU più usate [6] , in applicazioni reali, possono essere codificate usando meno informazioni. Venne quindi aggiunta un'altra ISA, detta Thumb, in cui ogni istruzione era codificata in solo 2 byte. Oggi detto «Thumb mode». Tuttavia non *tutte* le istruzioni ARM possono essere codificate in 2 byte, e il set di istruzioni Thumb è quindi in qualche modo limitato. Vale la pena notare che il codice compilato in ARM mode e Thumb mode può tranquillamente coesistere all'interno dello stesso programma. I creatori di ARM pensarono di estendere Thumb, dando vita a Thumb-2, appartso per la prima volta in ARMv7. Thumb-2 utilizza ancora istruzioni da 2-byte, ma ha anche alcune nuove istruzioni da 4 byte.

[1] Central processing unit
[2] Italian text placeholder
[3] La vecchia letteratura russa in materia utilizza il termine «traduttore».
[4] Reduced instruction set computing
[5] A proposito, le istruzioni a lunghezza fissa sono utili perchè è possibile calcolare senza sforzo l'indirizzo della prossiama (o della precedente) istruzione. Questa caratteristica sarà discussa nella sezione dedicata all'operatore switch() operator (**??** on page ??) .
[6] Che sono MOV/PUSH/CALL/Jcc

Esiste la convinzione errata che Thumb-2 sia un mix di ARM e Thumb. Questo non è corretto. Thumb-2 è stato invece esteso per supportare completamente tutte le caratteristiche del processore così da competere con l'ARM mode— un goal che è stato chiaramente raggiunto, visto che la maggior parte delle applicazioni per iPod/iPhone/iPad sono compilate per il set di istruzioni Thumb-2 (in effetti, molto probabilmente dovuto anche al fatto che Xcode lo fa per default). Successivamente fu la volta di ARM a 64-bit. Questa ISA ha opcode di 4-byte, e non necessita di alcun Thumb mode aggiuntivo. Tuttavia i requisiti a 64-bit hanno avuto un impatto sulla ISA, motivo per cui abbiamo oggi tre set di istruzioni ARM: ARM mode, Thumb mode (incluso Thumb-2) e ARM64. Queste ISA si intersecano parzialmente, ma si può dire che sono ISA differenti piuttosto che varianti della stessa. Per questo motivo cercheremo di aggiungere frammenti di codice in tutte e tre le ISA ARM in questo libro. A proposito, esistono anche molte altre ISAs RISC con opcode a lunghezza fissa di 32-bit, come MIPS, PowerPC e Alpha AXP.

2.2 Sistemi di numerazione

Gli umani si sono abituati ad usare il sistema numerico decimale probabilmente perche' quasi tutti hanno 10 dita. Ciononostante, il numero 10 non ha alcun significato rilevante in scienze e in matematica. Il sistema di numerazione naturale nell'elettronica digitale e' il binario: 0 per l'assenza di corrente nel filo e 1 per la presenza. 10 in binario e' 2 in decimale, 100 in binary is 4 in decimale e cosi' via.

Come si fa a convertire un numero da un sistema ad un altro?

La notazione posizionale e' usata quasi dappertutto. Cio' vuol dire che la cifra (numero in un singolo carattere) ha un "peso" diverso in base alla posizone in cui si trova. Se 2 si trova nella parte piu' a destra del numero, e' 2. Se si trova di una posizione piu' a destra, e' 20.

Per cosa sta 1234?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4$ = 1234 or $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

Lo stesso vale per i numeri binari, ma la base e' 2 invece di 10. Per cosa sta 101011?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 1 \cdot 1 = 43$ or $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

La notazione posizionale e' opposta a quella non-posizionale come il sistema numerico Romano. Forse l'umanita' ha deciso di passare alla notazione posizionale perche' e' piu' facile effettuare semplici operazioni (addizione, moltiplicazione, etc) a mano su carta.

I numeri binari possono essere addizionati, sotratti e cosi' via, nello stesso modo in cui ci e' stato insegnato a scuola, con l'unica differenza chec si sono solo 2 cifre disponibili.

I numeri binary possono risultare ingombranti quando usati in codici sorgenti e dump, ed in questi casi puo' essere usato il sistema esadecimale.

Il sistema esadecimale usa i numeri 0..9 e 6 lettere latine: A..F. Ogni cifra esadecimale occupa 4bit o 4 binary digits, ed e; quindi molto facile convertire da binario a esadecimale e viceversa, anche a mente,

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Come identificare quale sistema viene usato?

I numeri decimali sono solitamente scritti cosi' come sono, es. 1234. Alcuni assembler consentono pero' di aggiungere enfasi al sistema decimale ed il numero puo essere scritto con il suffisso "d": 1234d.

I numeri binari sono a volte preceduti dal prefisso "0b": 0b100110111 (GCC[7] non ha un'estensione del linguaggio standard per questo: `https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html`). C'e' anche un altro modo: il suffisso "b" suffix, ad esempio: 100110111b. Cerchero' di usare il prefisso "0b" per identificare i numeri binari all'interno del libro.

I numeri esadecimali sono preceduti dal prefisso "0x" in e altri PLs: 0x1234ABCD. Oppure hanno il suffisso "h": 1234ABCDh - questo e' il modo comune di rapressentarli negli assembler e nei debugger. Se il numero inizia con una cifra A..F, 0 dovrebbe essere aggiunto all'inizio: 0ABCDEFh. Cerchero' di usare il prefisso "0x" per identificare i numeri esadecimali all'interno del libro.

E' il caso di imparare a convertire i numeri a mente? Una tabella di numeri decimali ad una cifra puo' essere memorizzata facilmente. Per numeri piu' grandi, probabilmente, non e' il caso di tormentarsi.

2.2.1 Sistema di numerazione ottale

Un altro sistema di numerazione molto usato in pasato in programmazione e' l'ottale: ci sono 8 cifre (0..7) e ciascuna e' associata a 3 bit, quindi e' facile convertire da una parte all'altra. E' stato comunque rimpiazzato dal sistema esadecimale quasi ovunque, ma sorprendentemente c'e' una utility *NIX usata spesso e da molte persone che ha per argomento un numero ottale: `chmod`.

Come molti utenti *NIX sanno, l'argomento `chmod` puo' essere un numero di 3 cifre. La prima rappresenta i diritti del proprietario del file, la seconda i diritti del gruppo (a cui il file appartiene), la terza i diritti per chiunque altro. E ogni cifra puo' essere rappresentata in forma binaria:

decimale	binario	significato
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

Quindi ogni bit corrisponde ad un flag: read/write/execute.

La ragione per cui sto parlando di `chmod` e' che l'intero numero dell'argomento puo' essere rappresentato in ottale. Prendiamo per esempio 644. Quando si esegue `chmod 644 file`, si impostano i permessi di lettura/scrittura per il proprietario, il permesso di lettura per il gruppo, e il permesso di lettura per tutti gli altri. Convertiamo il numero 644 in ottale a binario, sara' `110100100`, o (in gruppi di 3 bit) `110 100 100`.

Notiamo che ogni tripletta descrive i permessi per proprietario/gruppo/altri (owner/group/others): il primo e' `rw-`, il secondo e' `r--` ed il terzo e' `r--`.

Il sistema ottale era anche molto diffuso nei vecchi computer come PDP-8, perche' una word poteva essere 12, 24 o 36 bit, e questi numeri sono tutti divisibili per 3, quindi il sistema ottale era naturale in quell'ambiente. Oggi tutti i computer comuni utilizano word/indirizzi lunghi 16, 32 o 64 bit, e questi numeri sono divisibili per 4, di conseguenza l'esadecimale risulta piu' naturale.

Il sistema ottale e' supportato da tutti i compilatori standard. Talvolta cio' da' vita a confusione, perche' i numeri ottali sono codificati preponendo uno zero, per esempio 0277 e' 255. A volte si potrebbe scrivere per errore "09" invece di 9 , e il compilatore non lo consentirebbe. GCC potrebbe presentare un errore del genere: `error: invalid digit "9" in octal constant`.

2.2.2 Divisibilita'

Quando si vede un numero decimale come 120, si puo' velocemente dedurre che e' divisibile per 10, perche' l'ultima cifra e' zero. Allo stesso modo, 123400 e' divisibile per 100 perche' le ultime due cifre sono zeri.

In maniera simile, il numero esadecimale 0x1230 e' divisible per 0x10 (ovvero 16), 0x123000 e' divisibile per 0x1000 (ovvero 4096), etc.

[7] GNU Compiler Collection

Il numero binario 0b1000101000 e' dibisibile per 0b1000 (8), etc.

Questa proprieta' puo' essere spesso usata per capire velocemente se la dimensione di un dato blocco di memoria e' "allungata" (padded) per raggiungere un certo confine. Per esempio, le sezioni in un file PE[8] iniziano quasi sempre ad indirizzi che terminano con 3 zeri esadecimali: 0x41000, 0x10001000, etc. La ragione per cui questo accade risiede nel fatto che quasi tutte le sezioni di un PE sono allungate per raggiungere un confine di 0x1000 (4096) bytes.

[8] Portable Executable

Capitolo 3

La funzione più semplice

La funzione più semplice possibile è probabilmente quella che restituisce un valore costante:

Eccola:

Listing 3.1: Codice

```
int f()
{
        return 123;
};
```

Compiliamola!

3.1 x86

Questo è il risultato prodotto dai compilatori ottimizzanti GCC e MSVC sulla piattaforma x86:

Listing 3.2: GCC/MSVC ()

```
f:
        mov     eax, 123
        ret
```

Ci sono solo due istruzioni: la prima mette il valore 123 nel registro `EAX`, per convenzione usato per la memorizzazione del valore di ritorno, la seconda, `RET`, restituisce l'esecuzione alla funzione chiamante.

La funzione chiamante prenderà il risultato dal registro `EAX`.

3.2 ARM

Sulla piattaforma ARM ci sono alcune differenze:

Listing 3.3: Keil 6/2013 () ASM Output

```
f PROC
        MOV      r0,#0x7b ; 123
        BX       lr
        ENDP
```

ARM usa il registro `R0` per restituire i risultati delle funzioni, quindi 123 viene copiato in `R0`.

Il return address sulla piattaforma ARM ISA non viene salvato nello stack locale ma nel link register, così che l'istruzione `BX LR` faccia in modo che l'esecuzione "salti" a quell'indirizzo — restituendo effettivamente l'esecuzione alla funzione chiamante.

Vale la pena notare che il nome dell'istruzione `MOV` è fuorviante in entrambe le ISA x86 e ARM.

Il dato non viene infatti *spostato*, bensì *copiato*.

3.3 MIPS

Esistono due naming conventions (convenzioni di denominazione) usate nel mondo MIPS per fare riferimento ai registri: per numero (da $0 a $31) or per pseudonome ($V0, $A0, Italian text placeholder).

Il seguente output assembly di GCC elenca i registri per numero:

Listing 3.4: GCC 4.4.5 ()

```
        j       $31
        li      $2,123                  # 0x7b
```

Italian text placeholder

Listing 3.5: GCC 4.4.5 (IDA)

```
                jr      $ra
                li      $v0, 0x7B
```

Il registro $2 (o $V0) è usato per memorizzare il valore di ritorno di una funzione. `LI` sta per "Load Immediate" ed è l'equivalente MIPS di `MOV`.

L'altra istruzione è quella di jump (salto) (J or JR) che riporta il flusso di esecuzione alla funzione chiamante, *saltando* all'indirizzo specificato nel registro $31 (o $RA).

Questo registro è l'analogo di LR[1] in ARM.

Ci si potrebbe domandare perchè le posizioni delle istruzioni load (LI) e jump (J or JR) siano invertite. Cio' e' dovuto ad una feature di RISC detta "branch delay slot".

La ragione per cui ciò avviene è una peculiarità dell'architettura di alcune ISAs RISC e non è di importanza rilevante per il nostro scopo - sarà sufficiente semplicemente ricordare che in MIPS l'istruzione che segue un jump o branch viene eseguita *prima* del jump/branch stesso.

Di conseguenza, le istruzioni branch scambiano sempre il posto con l'istruzione che deve essere eseguita prima.

3.3.1 Una nota sui nomi delle istruzioni e dei registri MIPS

I nomi dei registri e delle istruzioni nel mondo MIPS sono solitamente scritti in minuscolo. Tuttavia, per consistenza, useremo nomi in maiuscolo secondo la convenzione seguita da tutte le altre ISAs trattate in questo libro.

[1]Link Register

Capitolo 4

Utilizziamo il famoso esempio dal libro [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, (1988)]:

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

4.1 x86

4.1.1 MSVC

Compiliamolo in MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(l'opzione `/Fa` indica al compilatore di generare un file con il listato assembly)

Listing 4.1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

Italian text placeholder La differenza tra le sintassi Intel e AT&T-syntax sarà discussa al 4.1.3 on page 12.

Il compilatore ha generato il file, `1.obj`, che deve essere linkato (collegato) in `1.exe`. Nel nostro caso, il file contiene due segmentu: `CONST` (per i dati constanti) e `_TEXT` (per il codice).

La stringa `hello, world` in ha tipo `const char[]` [Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], ma non ha un nome proprio. Il compilatore deve in qualche modo aver a che fare con la stringa, e la definisce quindi con il nome interno `$SG3830`.

Questo è il motivo per cui l'esempio potrebbe essere riscritto nel modo seguente:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Torniamo al listato assembly. Come possiamo vedere, la stringa è terminata con un byte zero, che è lo standard per la terminazione delle stringhe . Italian text placeholder: 43.1.1 on page 324.

Nel code segment, `_TEXT`, esiste fino ad ora solo una funzione: `main()`. La funzione `main()` inizia con il codice di prologo (prologue code) e termina con il codice di epilogo (epilogue code) (come quasi qualunque funzione) [1].

Dopo il prologo della funzione, notiamo la chiamata alla funzione `printf()` : `CALL _printf`. Prima della chiamata, l'indirizzo della stringa (o un puntatore ad essa) contenente il saluto viene messo sullo stack con l'aiuto dell'istruzione `PUSH`.

Quando la funzione `printf()` restituisce il controllo alla funzione `main()`, l'indirizzo della stringa (o il puntatore) si trova ancora sullo stack. Poichè non ne abbiamo più bisogno, lo stack pointer (il registro `ESP`) deve essere corretto.

`ADD ESP, 4` significa aggiungi 4 al valore del registro `ESP`.

Perchè 4? Essendo questo un programma a 32-bit, abbiamo esattamente bisogno di 4 bytes per passare un indirizzo attraverso lo stack. Se fosse stato codice x64 ne sarebbero serviti 8. `ADD ESP, 4` è a tutti gli effetti equivalente a `POP register` ma senza usare alcun registro[2].

Per lo stesso scopo, alcuni compilatori (come l'Intel C++ Compiler) potrebbero emettere l'istruzione `POP ECX` invece di `ADD` (ad esempio, queto tipo di pattern può essere nel codice di Oracle RDBMS che è compilato con l' Intel C++ compiler). Questa istruzione ha pressoché lo stesso effetto ma il contenuto del registro `ECX` sarà sovrascritto. Il compilatore Intel C++ usa probabilmente `POP ECX` poichè l'opcode di questa istruzione è più corto di `ADD ESP, x` (1 byte per `POP` contro 3 per `ADD`).

Ecco un esempio dell'uso di `POP` al posto di `ADD` da Oracle RDBMS:

Listing 4.2: Oracle RDBMS 10.2 Linux (Italian text placeholder)

```
.text:0800029A                 push    ebx
.text:0800029B                 call    qksfroChild
.text:080002A0                 pop     ecx
```

Dopo la chiamata a `printf()`, il codice originale contiene la direttiva `return 0` —restituisci 0 come risultato dalla funzione `main()`.

Nel codice generato, questa è implementata dall'istruzione `XOR EAX, EAX`.

`XOR` è infatti semplicemente «eXclusive OR, ovvero OR esclusivo»[3] ma i compilatori lo usano spesso al posto di `MOV EAX, 0` —ancora una volta poichè è un opcode leggermente più corto (2 byte per `XOR` contro 5 per `MOV`).

Alcuni compilatori emettono l'istruzione `SUB EAX, EAX`, che significa *sottrai (SUBtract) il valore nel registro* `EAX` *dal valore nel registro* `EAX`, che, in ogni caso, risulta uguale a zero.

L'ultima istruzione `RET` restituisce il controllo al chiamante (caller). Solitamente, questo è codice CRT[4], che, a sua volta, restituisce il controllo all' OS[5].

[1]Maggiori informazioni si trovano nella sezione su prologo ed epilogo delle funzioni (5 on page 23).
[2]i flag CPU vengono comunque modificati
[3]wikipedia
[4]C runtime library
[5]Italian text placeholder

4.1.2 GCC

Proviamo adesso a compilare lo stesso codice con il compilatore GCC 4.4.1 su Linux: `gcc 1.c -o 1`. Successivamente, con l'aiuto del disassembler IDA[6], vediamo come è stata creata la funzione `main()`. IDA, come MSVC, utilizza la sintassi Intel[7].

Listing 4.3: codice in IDA

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world\n"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Il risultato è pressoché lo stesso. L'indirizzo della stringa `hello, world` (memorizzato nel data segment) è caricato prima nel registro `EAX` e successivamente salvato sullo stack. Inoltre, il prologo della funzione contiene `AND ESP, 0FFFFFFF0h` — questa istruzione allinea il valore del registro `ESP` a 16-byte. Ciò fa sì che tutti i valori sullo stack siano allineati allo stesso modo (la CPU è più efficiente se i valori che tratta sono collocati in memoria ad indirizzi allineati a, ovvero multipli di, 4 o 16 byte)[8].

`SUB ESP, 10h` alloca 16 byte sullo stack. Tuttavia, come vedremo a breve, solo 4 sono necessari in questo caso.

Ciò è dovuto al fatto che la dimensione dello stack allocato è anch'essa allineata a 16 byte.

L'indirizzo della stringa (o un puntatore alla stringa) è quindi memorizzato direttamente sullo stack senza utilizzare l'istruzione `PUSH`. *var_10* — è una variabile locale ed è anche un argomento di `printf()`. Maggiori dettagli in seguito.

Infine viene chiamata la funzione `printf()`.

Diversamente da MSVC, quando GCC compila senza ottimizzazione emette `MOV EAX, 0` invece di un opcode più breve.

L'ultima istruzione, `LEAVE` —è l'equivalente della coppia di istruzioni `MOV ESP, EBP` e `POP EBP` —in altre parole, questa istruzione riporta indietro lo stack pointer (`ESP`) e ripristina il registro `EBP` al suo stato iniziale. Ciò è necessario poiché abbiamo modificato i valori di questi registri (`ESP` and `EBP`) all'inizio della funzione (eseguendo `MOV EBP, ESP` / `AND ESP, …`).

4.1.3 GCC:

Vediamo come tutto questo può essere rappresentato nella sintassi assembly AT&T. Questa sintassi è molto più popolare nel mondo UNIX.

Listing 4.4: compiliamo in GCC 4.7.3

```
gcc -S 1_1.c
```

Otteniamo questo:

Listing 4.5: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world\n"
```

[6] Italian text placeholder

[7] Possiamo anche fare in modo che GCC produca un listato assembly con la sintassi Intel tramite l'opzione `-S -masm=intel`.

[8] *Italian text placeholder*

```
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Il listato contiene molte macro (iniziano con il punto). Per il momento non ci interessano.

Per il momento, e solo per una questione di semplificazione, possiamo ignorarle (fatta eccezione per la macro *.string* che codifica una sequenza di caratteri che termina con il null-byte (zero) proprio come una stringa C). Consideriamo soltanto questo [9]:

Listing 4.6: GCC 4.7.3

```
.LC0:
        .string "hello, world\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Alcune delle differenze maggiori tra la sintassi Intel e quella AT&T sono:

- Italian text placeholder

  Sintassi Intel: <istruzione> <operando di destinazione> <operando di origine>.

  Sintassi AT&T: <istruzione> <operando di origine> <operando di destinazione>.

  Ecco un modo facile per memorizzare la differenza: quando si tratta di sintassi Intel immagina che ci sia un segno di uguaglianza (=) tra i due operandi, quando si tratta di sintassi AT&T immagina una freccia da sinistra a destra (→) [10].

- AT&T: Il simbolo di percentuale (%) deve essere scritto prima del nome di un registro, e il dollaro ($) prima dei numeri.
- AT&T: All'istruzione si aggiunge un suffisso che definisce le dimensioni dell'operando:
  - q — quad (64 bit)
  - l — long (32 bit)
  - w — word (16 bit)
  - b — byte (8 bit)

[9] Questa opzione di GCC può essere usata per eliminare le macro «superflue»: *-fno-asynchronous-unwind-tables*

[10] A proposito, in alcune funzioni standard C(es., memcpy(), strcpy()) gli argomenti sono elencati nello stesso modo della sintassi Intel: prima il puntatore al blocco di memoria di destinazione, e poi il puntatore al blocco di memoria di origine.

Torniamo al risultato compilato: è identico a quello che abbiamo visto in IDA. Con una piccola differenza: `0FFFFFFF0h` è presentato come `$-16`. E' la stessa cosa: `16` nel sistema decimale è `0x10` in esadecimale. `-0x10` è uguale a `0xFFFFFFF0` (per un tipo di dato a 32-bit).

Ancora una cosa: il valore di ritorno viene settato a 0 usando `MOV`, non `XOR`. `MOV` semplicemente carica un valore in un registro. Il suo nome è fuorviante (il dato non viene spostato, bensì copiato). In altre architectures questa istruzione è chiamata «LOAD» o «STORE» o qualcosa di simile.

4.2 x86-64

4.2.1 MSVC—x86-64

Proviamo anche con MSVC a 64-bit:

Listing 4.7: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

In x86-64, tutti i registri sono stati estesi a 64-bit ed il loro nome ha il prefisso `R-`. Per usare lo stack meno spesso (in altre parole, per accedere meno spesso alla memoria esterna/cache), esiste un metodo molto diffuso per passare gli argomenti delle funzioni tramite i registri (*fastcall*) 50.3 on page 346. Ovvero, una parte degli argomenti è passata attraverso i registri, il resto —attraverso lo stack. In Win64, 4 argpmenti di funzione sono passati nei registri `RCX`, `RDX`, `R8`, `R9`. Questo è ciò che vediamo qui: un puntatore alla stringa per `printf()` è adesso passato nel registro `RCX` anziché tramite lo stack. I puntatori adesso sono a 64-bit , quindi sono passati nei registri a 64-bit (aventi il prefisso `R-`). E' comunque possibile, per retrocompatibilità, accedere alle parti a 32-bit parts, usando il prefisso `E-`. I registri `RAX`/`EAX`/`AX`/`AL` in x86-64 appaiono così:

<table>
<tr><td colspan="3">Italian text placeholder</td></tr>
<tr><td colspan="3">Italian text placeholder</td></tr>
<tr><td colspan="3">RAX^{x64}</td></tr>
<tr><td></td><td colspan="2">EAX</td></tr>
<tr><td></td><td colspan="2">AX</td></tr>
<tr><td></td><td>AH</td><td>AL</td></tr>
</table>

La funzione `main()` restituisce un valore di tipo *int*, che in , per migliore retrocompatibilità e portabilità, resta ancora a 32-bit, motivo per cui il registro `EAX` viene svuotato invece di `RAX` alla fine della funzione (i.e., la parte a 32-bit del registro). Ci sono anche 40 byte allocati nello stack locale. Questo spazio è detto «shadow space», di cui parleremo più avanti: **??** on page ??.

4.2.2 GCC—x86-64

Italian text placeholder:

Listing 4.8: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ;
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Un metodo per passare argomenti di funzione nei registri usato anche in Linux, *BSD and Mac OS X è [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [11].

I primi 6 argomenti sono passati nei registri `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`, ed il resto—tramite lo stack.

Quindi il puntatore alla stringa viene passato in `EDI` (la parte a 32-bit del registro). Ma perchè no nusare la parte a 64-bit `RDI`?

E' importante ricordare che tutte le istruzioni `MOV` in modalità 64-bit che scrivono qualcosa nella parte bassa a 32-bit di un registro, azzera anche la parte alta a 32-bits. Ad esempio, `MOV EAX, 011223344h` scrive un valore in `RAX` correttamente, poichè i bit della parte alta saranno azzerati.

Se apriamo il file oggetto compilato (.o), possiamo anche vedere gli opcode di tutte le istruzioni [12]:

Listing 4.9: GCC 4.4.6 x64

```
.text:00000000004004D0                          main  proc near
.text:00000000004004D0 48 83 EC 08              sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00           mov     edi, offset format ; "hello, world\n"
.text:00000000004004D9 31 C0                    xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF           call    _printf
.text:00000000004004E0 31 C0                    xor     eax, eax
.text:00000000004004E2 48 83 C4 08              add     rsp, 8
.text:00000000004004E6 C3                       retn
.text:00000000004004E6                          main  endp
```

Come possiamo notare, l'istruzione che scrive dentro `EDI` a `0x4004D4` occupa 5 byte. La stessa istruzione che scrive un valore a 64-bit dentro `RDI` occupa 7 bytes. Apparentemente, GCC sta cercando di risparmiare un po' di spazio. Inoltre, può essere sicuro che il segmento dati contenente la stringa non sarà allocato ad indirizzi maggiori di 4GiB.

Notiamo anche che il registro `EAX` è stato azzerato prima della chiamata alla funzione `printf()`. Ciò avviene perché il numbero dei registri vettore usati viene passato in `EAX` nei sistemi *NIX x86-64.

4.3 GCC—Ancora un'altra cosa

Il fatto che una stringa C *anonima* ha *const* tipo (4.1.1 on page 10), e che le stringhe C allocate nel constants segment siano garantite essere immutabili, hanno una conseguenza interessante: il compilatore potrebbe utilizzare una parte specifica della stringa.

Un esempio:

```
#include <stdio.h>

int f1()
{
        printf ("world\n");
}

int f2()
{
        printf ("hello world\n");
}

int main()
{
        f1();
        f2();
}
```

I comuni compilatori - (incluso MSVC) allocano due stringhe, ma vediamo cosa fa GCC 4.8.1:

Listing 4.10: GCC 4.8.1 + IDA

```
f1                proc near
```

[11] http://x86-64.org/documentation/abi.pdf
[12] Deve essere abilitato in **Options** → **Disassembly** → **Number of opcode bytes**

```
s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset s ; "world\n"
                call    _puts
                add     esp, 1Ch
                retn
f1              endp

f2              proc near

s               = dword ptr -1Ch

                sub     esp, 1Ch
                mov     [esp+1Ch+s], offset aHello ; "hello "
                call    _puts
                add     esp, 1Ch
                retn
f2              endp

aHello          db 'hello '
s               db 'world',0xa,0
```

Quando stampiamo la stringa «hello world» le due parole sono posizionate adiacenti in memoria e `puts()` chiamata dalla funzione `f2(`) non è al corrente che la stringa è in divisa. Infatti non lo è; è divisa soltanto «virtualmente», in questo listato.

Quando `puts()` è chiamata dalla funzione `f1()`, usa la stringa «world» più un byte zero. `puts()` non sa che c'è qualcos'altro prima di questa stringa!

Questo trucco intelligente è usato spesso, almeno da GCC, e può far risparmiare un po' di memoria.

4.4 ARM

Per gli esperimenti con i processori ARM, sono stati utilizzati diversi compilatori:

- Diffuso nel settore embedded: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE con il compilatore LLVM-GCC 4.2 [13].
- GCC 4.9 (Linaro) (Italian text placeholder ARM64), Italian text placeholder http://go.yurichev.com/17325.

In qesto libro, se non diversamente specificato, viene utilizzato codice ARM a 32-bit ARM (incluse le modalita' Thumb e Thumb-2). Italian text placeholder

4.4.1 Keil 6/2013 ()

Iniziamo a compilare il nostro esempio in Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Il compilatore *armcc* produce un listato assembly con sintassi Intel, e utilizza macro di alto livello legate al processore ARM [14], tuttavia e' piu' importante per noi vedere le istruzioni «cosi' come sono», quindi guardiamo il risultato compilato con IDA.

Listing 4.11: Keil 6/2013 () IDA

```
.text:00000000             main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      __2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF: main+4
```

[13] Apple Xcode 4.6.3 utilizza il compilatore open-source GCC come compilatore front-end ed il generatore di codice LLVM
[14] ad esempio, l' ARM mode 'e privo delle istruzioni `PUSH` / `POP`

Nell'esempio possiamo facilmente vedere che ogni istruzione ha lunghezza pari a 4 byte. Difatti abbiamo compilato il codice per la modalita' ARM e non Thumb.

La prima istruzione, `STMFD SP!, {R4,LR}` [15], funzione come l' istruzione `PUSH` in x86, scrivendo i valori di due registri (`R4` Italian text placeholder LR) nello stack. Infatti il listato di output prodotto dal compilatore *armcc*, per semplificazione, mostra l'istruzione `PUSH {r4,lr}`. Ma cio' non e' del tutto esatto. L'istruzione `PUSH` e' disponibile solo in modalita' Thumb. Utilizziamo quindi IDA per non fare confusione.

Questa istruzione dapprima decrementa il valore di SP[17] cosi' da farlo puntare alla porzione dello stack che' e' libera di ospitare nuovi dati, quindi salva il valore dei registri `R4` e LR all'indirizzo memorizzato nel registro SP appena modificato.

Questa istruzione (esattamente come `PUSH` in Thumb mode) e' in grado di salvare il valore di piu' registri contemporaneamente, cosa che e' puo' risultare molto utile. A proposito, non ha un equivalente in x86. Si puo' notare anche che l'istruzione `STMFD` e' una generalizzazione dell'istruzione `PUSH` (che estende le sue funzionalita'), poiche' puo' funzionare con qualunque registro, e non solo SP. In altre parole, `STMFD` puo' essere usata per memorizzare un insieme di registri all'indirizzo di memoria specificato.

L'istruzione `ADR R0, aHelloWorld` aggiunge o sottrae il valore nel registro PC[18] all'offset dove e' memorizzata la stringa `hello, world`. Ci si potrebbe chiedere, come e' utilizzato qui il registro `PC`? Cio' e' detto «» [19]. Questo tipo di codice puo' essere eseguito a indirizzi non fissi (variabili)in memoria. In altre parole, e' un indirizzamento relativo a PC (PC-relative addressing). L'istruzione `ADR` tiene conto della differenza tra l'indirizzo di questa istruzione e l'indirizzo dove si trova la stringa. Questa differenza (offset) sara' sempre la stessa, a prescindere dall'indirizzo in cui nostro codice sara' caricato dall'OS. Cio' spiega perche' bisogna soltanto aggiungere l'indirizzo dell'istruzione corrente (from PC) per ottenere l'indirizzo assoluto in memoria della nostra stringa C.

L'istruzione `BL __2printf` [20] chiama la funzione `printf()`. Questa istruzione funziona cosi':

- memorizza l'indirizzo successivo all'istruzione `BL` (`0xC`) nel registro LR;
- quindi passa il controllo a `printf()` scrivendo il suo indirizzo nel registro PC.

Quando la funzione `printf()` termina la sua esecuzione, deve sapere a chi restituire il controllo (dove ritornare). Per questo motivo ogni funzione passa il controllo all'indirizzo memorizzato nel registro LR.

Questa e' una differenza tra processori RISC «puri» come ARM e processori simili a CISC[21] come x86, nei quali il return address e' solitamente memorizzato nello stack [22].

A proposito, un indirizzo assoluto o un offset a 32-bit non puo' essere codificato nell'istruzione a 32-bit `BL` poiche' ha solo spazio per 24 bit. Come potremmo ricordare, tutte le istruzioni in ARM-mode hanno dimensione fissa di 4 byte (32 bit). Dunque possono essere collocate solo su indirizzi allineati a 4-byte. Cio' implica che gli ultimi 2 bits dell'indirizzo dell'istruzione (che sono sempre zero) possono essere omessi. Abbiamo in definitiva 26 bit per la codifica dell'offset (offset encoding). E cio e' sufficiente per codificare $current_PC \pm \approx 32M$.

L'istruzione successiva, `MOV R0, #0` [23] scrive soltanto 0 nel registro `R0`. Questo succede perche' la nostra funzione C restituisce 0, ed il valore di ritorno deve essere memorizzato nel registro `R0`.

L'ultima istruzione `LDMFD SP!, R4,PC` [24]. Carica i valori dallo stack (o qualunque altra zona di memoria) per salvarli nei registri `R4` e PC, e incrementa lo stack pointer SP. In questo caso funziona come `POP`.
N.B. La prima istruzione `STMFD` aveva salvato la coppia di registri `R4` e LR sullo stack, ma `R4` e PC vengono *ripristinati* durante l'esecuzione di `LDMFD`.

Come gia' sappiamo, l'indirizzo del posto a cui ogni funzione devere restituire il controllo e' solitamente salvato nel registro LR. La prima istruzione salva il suo valore nello stack perche' lo stesso registro sara' usato dalla nostra funzione `main()` per la chiamata a `printf()`. Al termine della funzione, questo valore puo' essere scritto direttamente nel registro PC, passando di fatti il controllo al punto in cui la nostra funzione era stata chiamata.

Dal momento che `main()` e' solitamente la funzione principale in , il controllo sara' restituito al loader dell' OS oppure ad un punto in una CRT, o qualcosa del genere.

[15] STMFD[16]
[17] stack pointer. SP/ESP/RSP x86/x64. SP ARM.
[18] Program Counter. IP/EIP/RIP x86/64. PC ARM.
[19] Maggiori informazioni sono fornite nella sezione (**??** on page ??)
[20] Branch with Link
[21] Complex instruction set computing
[22] Maggiori informazioni si trovano nella prossima sezione (6 on page 24)
[23] Italian text placeholder MOVe
[24] LDMFD[25] e' l'istruzione inversa rispetto a STMFD

Tutto cio' consente di omettere l'istruzione `BX LR` alla fine della funzione.

`DCB` e' una direttiva assembly che definisce un array di byte o una stringa ASCII, analoga alla direttiva DB in linguaggio assembly x86.

4.5 MIPS

4.5.1 Qualche parola sul «global pointer»

Un importante concetto MIPS e' il «global pointer». Come potremmo gia' sapere, ogni ustrizione MIPS ha lunghezza pari a 32 bit, quindi e' impossibile inserire un indirizzo a 32-bit in una sola istruzione: deve essere usata una coppia (come ha fatto GCC nell'esempio per il caricamento dell'indirizzo della stringa). E' comunque possibile caricare dati da un indirizzo nell'intervallo $register - 32768 ... register + 32767$ utilizzando una singola istruzione (perche' 16 bit di un signed offset possono essere codificati in una singola istruzione). Possiamo quindi allocare per questo scopo un registro e allocare anche un'area a 64KiB per i dati piu' usati. Questo registro dedicato e' detto «global pointer» e punta in mezzo dall'area di 64KiB. Questa area solitamente contiene variabili globali e indirizzi di funzioni importate come `printf()`, perche' gli sviluppatori di GCC hanno deciso che il recupero dell'indirizzo di una funzione deve essere veloce tanto quanto l'esecuzione di una singola istruzione invece di due.

In un file ELF questa area di 64KiB e' collocata parzialmente nelle sezioni .sbss («small BSS[26]») per dati non inizializzati e .sdata («small data») per dati inizializzati.

Cio' implica che il programmatore puo' scegliere a quale dati si possa accedere piu' velocemente e piazzarli nelle sezioni .sdata/.sbss. Alcuni programmatori old-school potrebbero ricordarsi del memory model MS-DOS 74 on page 505 o dei memory manger MS-DOS come XMS/EMS, in cui tutta la memoria era divisa in blocchi da 64KiB.

Questo concetto non e' unicamente di MIPS. Anche PowerPC usa la stessa tecnica.

4.5.2 GCC

Consideriamo il seguente esempio che illustra il concetto di «global pointer».

Listing 4.12: GCC 4.4.5 ()

```
1  $LC0:
2  ; \000 :
3          .ascii  "Hello, world!\012\000"
4  main:
5  ;
6  ;  GP:
7          lui     $28,%hi(__gnu_local_gp)
8          addiu   $sp,$sp,-32
9          addiu   $28,$28,%lo(__gnu_local_gp)
10 ; :
11         sw      $31,28($sp)
12 ;  $25:
13         lw      $25,%call16(puts)($28)
14 ;  $4 ($a0):
15         lui     $4,%hi($LC0)
16 ; :
17         jalr    $25
18         addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ;  RA:
20         lw      $31,28($sp)
21 ;  $zero  $v0:
22         move    $2,$0
23 ; :
24         j       $31
25 ; :
26         addiu   $sp,$sp,32 ; branch delay slot
```

Come possiamo vedere, il registro $GP e' settato nel prologo della funzione affinche' punti nel mezzo di questa area. Il registro RA[27] viene anche salvato sullo stack locale. `puts()` e' usata anche qui al posto di `printf()`. L'indirizzo della

[26] Block Started by Symbol
[27] Italian text placeholder

funzione `puts()` e' caticato in $25 usando `LW`, l'istruzione («Load Word»). Successivamente l'indirizzo della stringa viene caricato in $4 usando la coppia di istruzioni `LUI` («Load Upper Immediate») e `ADDIU` («Add Immediate Unsigned Word»). `LUI` setta i 16 bit alti del registro (da cui la parola «upper» nel nome dell'istruzione) e `ADDIU` aggiunge i 16 bit piu' bassi dell'indirizzo.

`ADDIU` segue `JALR` (ti ricordi dei *branch delay slots*?). Il registro $4 e' anche detto $A0, e' usato per passare il primo argomento di una funzione [28].

`JALR` («Jump and Link Register») salta all'indirizzo memorizzato nel registro $25 register (indirizzo di `puts()`) salvando l'indirizzo della prossima istruzione (LW) in RA. Questo e' molto simile ad ARM. Oh, e una cosa importate e' che l'indirizzo salvato in RA non e' l'indirizzo della prossima istruzione (perche' e' in un *delay slot* e viene eseguito prima prima dell'istruzione jump), ma l'indirizzo dell'istruzione dopo la prossima (dopo il *delay slot*).

Quindi, $PC + 8$ viene scritto in RA durante l'esecuzione di `JALR`, nel nostro caso, questo e' l'indirizzo dell'istruzione `LW` successiva a `ADDIU`.

`LW` («Load Word») a riga 20 ripristina RA dallo stack locale (questa istruzione e' in effetti parte dell'epilogo della funzione).

`MOVE` a riga 22 copia il valore dal registro $0 ($ZERO) al $2 ($V0).

MIPS ha un registro *costante*, il cui valore e' sempre zero. Apparentemente, gli sviluppatori MIPS hanno pensato che zero e' la costante piu' usata in programmazione, quindi usiamo il registro $0 ogni volta che serve il valore zero.

Un altro fatto interessante e' che in MIPS non c'e' un'istruzione che trasferisce dati tra registri. Infatti, `MOVE DST, SRC` e' `ADD DST, SRC, $ZERO` ($DST = SRC + 0$), che fa la stessa cosa. Apparentemente gli sviluppatori MIPS desideravano avere una tabella di opcode compatta. Questo non significa che un'addizione si verifichi per ogni istruzione `MOVE`. Molto probabilmente, la CPU ottimizza queste pseudoistruzioni e la ALU[29] non viene mai usata.

`J` a riga 24 salta all'indirizzo in RA, effettuando di fatti il ritorno dalla funzione. `ADDIU` dopo J e' in effetti eseguita prima di J (ricordi i *branch delay slots*?) e fa parte dell'epilogo della funzione. Ecco anche il listato generato da IDA. Ogni registro qui ha il suo pseudonimo:

Listing 4.13: GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
;
;  GP:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
; :
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
; :
; :
.text:00000010                 sw      $gp, 0x20+var_10($sp)
;  $t9:
.text:00000014                 lw      $t9, (puts & 0xFFFF)($gp)
;  $a0:
.text:00000018                 lui     $a0, ($LC0 >> 16)  # "Hello, world!"
; :
.text:0000001C                 jalr    $t9
.text:00000020                 la      $a0, ($LC0 & 0xFFFF)  # "Hello, world!"
;  RA:
.text:00000024                 lw      $ra, 0x20+var_4($sp)
;  $zero  $v0:
.text:00000028                 move    $v0, $zero
; :
.text:0000002C                 jr      $ra
; :
.text:00000030                 addiu   $sp, 0x20
```

[28] La tabella dei registri MIPS e' riportata in appendice C.1 on page 530
[29] Italian text placeholder

L'istruzione alla riga 15 salva il valore di GP sullo stack locale, e questa istruzione manca misteriosamente dal listato prodotto da GCC, forse per un errore di GCC [30]. Il valore di GP deve essere infatti salvato, perche' ogni funzione puo' usare la sua finestra dati da 64KiB. Il registro contenente l'indirizzo di `puts()` e' chiamato $T9, perche' i registri con il prefisso T- sono detti «temporaries» ed il loro contenuto puo' non essere preservato.

4.5.3 GCC

GCC e' piu' verboso.

Listing 4.14: GCC 4.4.5 ()

```
1  $LC0:
2          .ascii  "Hello, world!\012\000"
3  main:
4  ;
5  ; :
6          addiu   $sp,$sp,-32
7          sw      $31,28($sp)
8          sw      $fp,24($sp)
9  ; :
10         move    $fp,$sp
11 ;  GP:
12         lui     $28,%hi(__gnu_local_gp)
13         addiu   $28,$28,%lo(__gnu_local_gp)
14 ; :
15         lui     $2,%hi($LC0)
16         addiu   $4,$2,%lo($LC0)
17 ; :
18         lw      $2,%call16(puts)($28)
19         nop
20 ;  puts():
21         move    $25,$2
22         jalr    $25
23         nop  ; branch delay slot
24
25 ; :
26         lw      $28,16($fp)
27 ;  $2 ($V0) :
28         move    $2,$0
29 ; .
30 ;  SP:
31         move    $sp,$fp
32 ;  RA:
33         lw      $31,28($sp)
34 ;  FP:
35         lw      $fp,24($sp)
36         addiu   $sp,$sp,32
37 ;  RA:
38         j       $31
39         nop  ; branch delay slot
```

Qui vediamo che il registro FP e' usato come un puntatore allo stack frame. Vediamo anche 3 NOP[31]s. Di cui il secondo e terzo seguono all'istruzione branch. Forse GCC aggiunge sempre NOPs (a causa dei *branch delay slots*) dopo le istruzioni branch e successivamente, se le ottimizzazioni sono attivate, forse li elimina. Quindi in questo caso sono rimasti.

Ecco anche il listato IDA:

Listing 4.15: GCC 4.4.5 (IDA)

```
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_8           = -8
5 .text:00000000 var_4           = -4
6 .text:00000000
7 ;
```

[30] Apparentemente, le funzioni che generano i listati non sono fondamentali per gli utenti GCC, quindi qualche errore non ancora corretto puo' esserci.
[31] No OPeration

```
8  ; :
9  .text:00000000              addiu   $sp, -0x20
10 .text:00000004              sw      $ra, 0x20+var_4($sp)
11 .text:00000008              sw      $fp, 0x20+var_8($sp)
12 ; :
13 .text:0000000C              move    $fp, $sp
14 ;  GP:
15 .text:00000010              la      $gp, __gnu_local_gp
16 .text:00000018              sw      $gp, 0x20+var_10($sp)
17 ; :
18 .text:0000001C              lui     $v0, (aHelloWorld >> 16)  # "Hello, world!"
19 .text:00000020              addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  # "Hello, world!"
20 ; :
21 .text:00000024              lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028              or      $at, $zero ; NOP
23 ;  puts():
24 .text:0000002C              move    $t9, $v0
25 .text:00000030              jalr    $t9
26 .text:00000034              or      $at, $zero ; NOP
27 ; :
28 .text:00000038              lw      $gp, 0x20+var_10($fp)
29 ;  $2 ($V0) :
30 .text:0000003C              move    $v0, $zero
31 ; .
32 ;  SP:
33 .text:00000040              move    $sp, $fp
34 ;  RA:
35 .text:00000044              lw      $ra, 0x20+var_4($sp)
36 ;  FP:
37 .text:00000048              lw      $fp, 0x20+var_8($sp)
38 .text:0000004C              addiu   $sp, 0x20
39 ;  RA:
40 .text:00000050              jr      $ra
41 .text:00000054              or      $at, $zero ; NOP
```

E' interessante notare che IDA ha riconosciuto la coppia di istruzioni `LUI` / `ADDIU` e le ha fusae in un'unica pseudoistruzione `LA` («Load Address») a riga 15. Possiamo anche vedere che questa pseudoistruzione e' lunga 8 bytes! Questa e' una pseudoistruzione (o *macro*) in quanto non e' una vera istruzione MIPS , ma soltanto un nome comodo per una coppia di istruzioni.

Un'altra cosa e' che IDA non ha riconosciuto le istruzioni NOP , che sono alle righe 22, 26 e 41. E' `OR $AT, $ZERO`. Essenzialmente, questa istruzione applica l'operazione OR al contenuto del registro $AT con zero, che e', ovviamente, un'istruzione nulla/inutile. MIPS, come molte altre ISA, non ha un'istruzione NOP propria.

4.5.4 Ruolo dello the stack frame in questo esempio

L'indirizzo della stringa e' passato nel registro. Perche' impostare allora ugualmente uno stack locale? La ragione sta nel fatto che i valori dei registri RA e GP devono essere salvati da qualche parte (poiche' viene chiamata `printf()`), e lo stack locale e' usato proprio per questo scopo. Se fosse stata una leaf function, sarebbe stato possibile fare a meno (disfarsi) del prologo e dell'epilogo, ad esempio: 3.3 on page 9.

4.5.5 GCC: carichiamolo in GDB

Listing 4.16: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mips-linux-gnu".
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

4.6

La differenza principale tra il codice x86/ARM e x64/ARM64 è che il puntatore alla stringa è adesso lungo 64 bit. Infatti, le moderne CPU sono ora a 64-bit grazie ai costi ridotti della memoria e alla sua grande richiesta da parte delle applicazioni moderne. Possiamo aggiungere ai nostri computer più memoria di quanto i puntatori a 32-bit siano in grado di indirizzare. Di conseguenza, tutti i puntatori sono oggi a 64-bit.

4.7

- http://challenges.re/48
- http://challenges.re/49

Capitolo 5

Prologo ed epilogo delle funzioni

Il prologo (o preambolo) di una funzione e' una sequenza di istruzioni all'inizio della funzione stessa. Spesso ha una forma simile al seguente frammento di codice:

```
push    ebp
mov     ebp, esp
sub     esp, X
```

Cosa fanno queste istruzioni: salvano il valore del registro EBP, impostano il valore del registro EBP con il valore di ESP e allocano spazio sullo stack per le variabili locali.

Il valore di EBP resta costante durante il periodo di esecuzione della funzione, ed e' usato per accede a variabili locali e argomenti. Per lo stesso scopo si puo' usare ESP, ma siccome cambia nel tempo non e' un approccio molto conveniente.

L'epilogo della funzione libera lo spazio allocato nello stack, ripristina il valore nel registro EBP al suo stato iniziale e restituisce il controllo al chiamante:

```
mov    esp, ebp
pop    ebp
ret    0
```

Prologo ed epilogo di funzioni sono solitamente identificati nei disassemblatori per delimitare le funzioni.

5.1

Epiloghi e prologhi possono avere un effetto negativo sulla performance in caso di ricorsione.

Maggiori informazioni sulla ricorsione in questo libro: **??** on page ??.

Capitolo 6

Lo stack e' una delle strutture dati piu' importanti in informatica [1]. AKA[2] **LIFO!**[3].

Tecnicamente, e' soltanto un blocco di memoria nella memoria di un processo insieme al registro ESP o RSP in x86 o x86, o il registro SP in ARM, come puntatore all'interno di quel blocco.

Le istruzioni di accesso allo stack piu' usate sono PUSH e POP (sia in x86 che in ARM Thumb-mode). PUSH sottrae da ESP / RSP /SP 4 in modalita' 32-bit (oppur 8 in modalita' 64-bit) e scrive successivamente il contenuto del suo unico operando nell'indirizzo di memoria puntato da ESP / RSP /SP.

POP e' l'operazione inversa: recupera il dato dalla memoria a cui punta SP, lo carica nell'operando dell'istruzione (di solito un registro) e successivamente aggiunge 4 (o 8) allo stack pointer.

A seguito dell'allocazione dello stack, lo stack pointer punta alla base (fondo) dello stack. PUSH decrementa lo stack pointer e POP lo incrementa. La base dello stack e' in realta' all'inizio della memoria allocata per il blocco (porzione) dello stack. Sembra strano, ma e' cosi'.

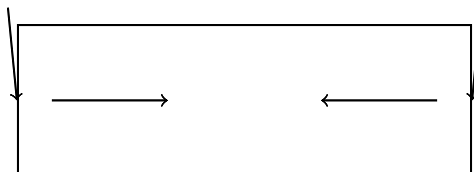
ARM supporta stack decrescenti e crescenti.

Ad esempio le istruzioni STMFD/LDMFD, STMED[4]/LDMED[5] sono fatte per operare con uno stack decrescente (che cresce verso il basso, inizia con un indirizzo alto e prosegue verso il basso). Le istruzioni STMFA[6]/LDMFA[7], STMEA[8]/LDMEA[9] sono fatte per operare con uno stack crescente (che cresce verso l'alto, da un indirizzo basso verso uno piu alto).

6.1 Perche' lo stack cresce al contrario?

Intuitivamente potremmo pensare che lo stack cresca verso l'alto, ovvero verso indirizzi piu' alti, come qualunque altra struttura dati.

La ragione per cui lo stack cresce verso il basso e' probabilmente di natura storica. Quando i computer erano talmente grandi da occupare un'intera stanza, era facile dividere la memoria in due parti, una per lo heap e l'altra per lo stack. Ovviamente non era possibile sapere a priori quanto sarebbero stati grandi lo stack e lo heap durante l'esecuzione di un programma, e questa soluzione era la piu' semplice.



In [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][10]possiamo leggere:

[1] wikipedia.org/wiki/Call_stack
[2] Also Known AsItalian text placeholder
[3] **LIFO!**
[4] Store Multiple Empty Descending ()
[5] Load Multiple Empty Descending ()
[6] Store Multiple Full Ascending ()
[7] Load Multiple Full Ascending ()
[8] Store Multiple Empty Ascending ()
[9] Load Multiple Empty Ascending ()
[10] `http://go.yurichev.com/17270`

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.
>
> Il nucleo utente di una immagine e' diviso in tre segmenti logici. Il segmento text del programma inizia in posizione 0 nel virtual address space. Durante l'esecuzione questo segmento viene protetto da scrittura, ed una sua singola copia viene condivisa tra i processi che eseguono lo stesso programma. Al primo limite di 8K byte dopra il segmento text del programma, nel virtual address space comincia un segmento dati scrivibile, non condiviso, le cui dimensioni possono essere estese da una chiamata di sistema. A partire dall'indirizzo piu' alto nel virtual address space c'e' lo stack segment, che automaticammente cresce verso il basso al variare dello stack pointer hardware.

Questo ricorda molto come alcuni studenti utilizzino lo stesso quaderno per prendere appunti di due diverse materie: gli appunti per la prima materia sono scritti normalmente, e quelli della seconda materia sono scritti a partire dalla fine del quaderno, capovolgendolo. Le note si potrebbero "incontrare" da qualche parte in mezzo al quaderno, nel caso in cui non ci sia abbastanza spazio libero.

6.2 Per cosa viene usato lo stack?

6.2.1 Salvare l'indirizzo di ritorno della funzione

x86

Quando si chiama una funzione con l'istruzione `CALL`, l'indirizzo del punto esattamente dopo la `CALL` viene salvato nello stack, e successivamente viene eseguito un jump non condizionale all'indirizzo dell'operando di CALL.

L'istruzione `CALL` e' equivalente alla coppia di istruzioni `PUSH indirizzo_dopo_call / JMP operando`.

`RET` preleva un valore dallo stack e effettua un jump ad esso — cio' equivale alla coppia di istruzioni `POP tmp / JMP tmp`.

Riempire lo stack fino allo straripamento e' semplicissimo. Basta ricorrere alla ricorsione eterna:

```
void f()
{
        f();
};
```

MSVC 2008 riporta il problema:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause ↙
    ↘ runtime stack overflow
```

...ma genera in ogni caso il codice correttamente:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ                               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                                          ; f
```

...Se attiviamo le ottimizzazioni del compilatore (`/Ox` option) il codice ottimizzato non causera' overflow dello stack e funzionera' invece *correttamente*[11]:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                                          ; f
```

GCC 4.4.1 genera codice simile in antrambi i casi, senza avvertire del problema.

ARM

Anche i programmi ARM usano lo stack per salvare gli indirizzi di ritorno, ma lo fanno in maniera diversa. Come detto in «» (4.4 on page 16), As mentioned in il RA viene salvato nel LR (link register). Se si presenta comunque la necessita' di chiamare un'altra funzione ed usare il registro LR ancora una volta, il suo valore deve essere salvato. Solitamente questo valore e' slvato nel preambolo della funzione.

Spesso vediamo istruzioni come `PUSH R4-R7,LR` insieme ad isrtuzioni nell'epilogo come `POP R4-R7,PC` —percio' i valori dei registri che saranno usati nella funzione vengono salvati nello stack, incluso LR.

Ciononostante, se una funzione non chiama al suo interno nessun'altra funzione, in terminologia RISC e' detta *leaf function*, o funzione foglia.[12]. Di conseguenza, le leaf functions non salvano il registro LR register (perche' difatti non lo modificano). Se una simile funzione e' molto breve e usa un piccolo numero di registri, potrebbe non usare del tutto lo stack. E' quindi possible chiamare le leaf functions senza usare lo stack, cosa che puo' essere piu' veloce che sulle macchine x86 perche' ;a RA< esterna non viene usata per lo stack [13]. Lo stesso principio puo' tornare utile quando la memoria per lo stack non e' stata ancora allocata o non e' disponibile.

Alcuni esempi di funzioni foglia: **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??, **??** on page ??.

6.2.2 x86: la funzione alloca()

Vale la pena esaminare la funzione `alloca()` [14]. Questa funzione opera come `malloc()`, ma alloca memoria direttamente nello stack. Il pezzo di memoria allocato non necessita di essere liberato tramite una chiamata alla funzione `free()` function call,
poiche' l'epilogo della funzione (5 on page 23) ripristina `ESP` al suo valore iniziale e la memoria allocata viene semplicemente *abbandonata*. Vale anche la pena notare come e' implementata la funzione `alloca()`. In termini semplici, questa funzione shifta `ESP` in basso, verso la base dello stack, per il numero di byte necessari e setta `ESP` per puntare al blocco *allocato*.

Proviamo:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
```

[11]sarcasmo, si fa per dire

[12]infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[13]Tempo fa, su PDP-11 and VAX, l'istruzione CALL instruction (chiamare altre funzioni) era costosa; poteva richiedere fino al 50% del tempo di esecuzione, ed era quindi consuetudine pensare che avere un grande numero di piccole funzioni fosee un anti-pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

[14]In MSVC, l'implementazione della funzione si trova in `alloca16.asm` e `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

La funzione `_snprintf()` opera come `printf()`, ma invece di inviare il risultato a stdout (es. al terminale o console), lo scrive nel buffer `buf`. La funzione `puts()` copia il contenuto di `buf` in stdout. Ovviamente questo due chiamate potrebbero essere rimpiazzate da una sola chiamata a `printf()`, ma in questo caso era necessario per illustrare l'uso di un piccolo buffer.

MSVC

Compiliamo (MSVC 2010):

Listing 6.1: MSVC 2010

```
...

    mov     eax, 600          ; 00000258H
    call    __alloca_probe_16
    mov     esi, esp

    push    3
    push    2
    push    1
    push    OFFSET $SG2672
    push    600              ; 00000258H
    push    esi
    call    __snprintf

    push    esi
    call    _puts
    add     esp, 28          ; 0000001cH

...
```

L'unico argomento di `alloca()` e' passato tramite il registro `EAX` (anziche' metterlo nello stack) [15].

GCC +

GCC 4.4.1 fa lo stesso senza chiamare funzioni esterne:

Listing 6.2: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                         ;
        mov     DWORD PTR [esp], ebx             ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600           ; maxlen
```

[15] Questo perche' alloca() e' una "compiler intrinsic" (70 on page 495) piuttosto che una funzione normale. Una delle ragioni per cui abbiamo bisogno di una funzione separata, invece di un paio di istruzioni nel codice, e' che l'implementazione di alloca() di MSVC[16] ha anche del codice che legge dalla memoria appena llocata, per far si che l'OS effettui il mapping della memoria fisica in questa regione della VM[17]. Dopo la chiamata a `alloca()`, `ESP` punta al blocco di 600 byte, ed e' possibile utilizzarlo come memoria per l'array `buf`.

```
        call    _snprintf
        mov     DWORD PTR [esp], ebx                ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC +

Esaminiamo lo stesso codice, ma in sintassi AT&T:

Listing 6.3: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

The code e' uguale a quello del listato precedente.

A proposito, `movl $3, 20(%esp)` corrisponde a `mov DWORD PTR [esp+20], 3` in sintassi Intel. In sintassi AT&T, il formato registro+offset per indirizzare memoria appare come `offset(%register)`.

6.2.3 (Windows) SEH

I record SEH[18], se presenti, sono anch'essi memorizzati nello stack. Maggiori informazioni qui: (53.2 on page 360).

6.2.4 Protezione da buffer overflow

Maggiori informazioni qui (13.2 on page 60).

6.2.5 Deallocazione automatica dei dati nello stack

Probabilmente la ragione per cui si memorizzano nello stack le variabili locali e i record SEH deriva dal fatto che questi dati vengono "liberati" automaticamente all'uscita dalla funzione, usando soltanto un'istruzione per correggere lo stack pointer (spesso e' `ADD`). Si puo' dire che anche gli argomenti delle funzioni sono deallocati automaticamente alla fine della funzione. Invece, qualunque altra cosa memorizzata nello *heap* deve essere deallocata esplicitamente.

[18]Structured Exception Handling

6.3 Rumore nello stack

In questo libro si fa spesso riferimento a «rumore» o «spazzatura» (garbage) nello stack o in memoria.

Da dove arrivano? Sono cio' che resta dopo l'esecuzione di altre funzioni. Un piccolo esempio:

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};

void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
        f1();
        f2();
};
```

Compilando si ottiene:

Listing 6.4: MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
```

```
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Il compilatore si lamentera' un pochino…

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:st.exe
st.obj
```

Ma quando avvieremo il programma …

```
c:\Polygon\c>st
1, 2, 3
```

Oh, che cosa strana! Non abbiamo impostato il valore di alcuna variabile in `f2()`. Si tratta di valori «fantasma» , che si trovano ancora nello stack.

Carichiamo l'esempio in OllyDbg:

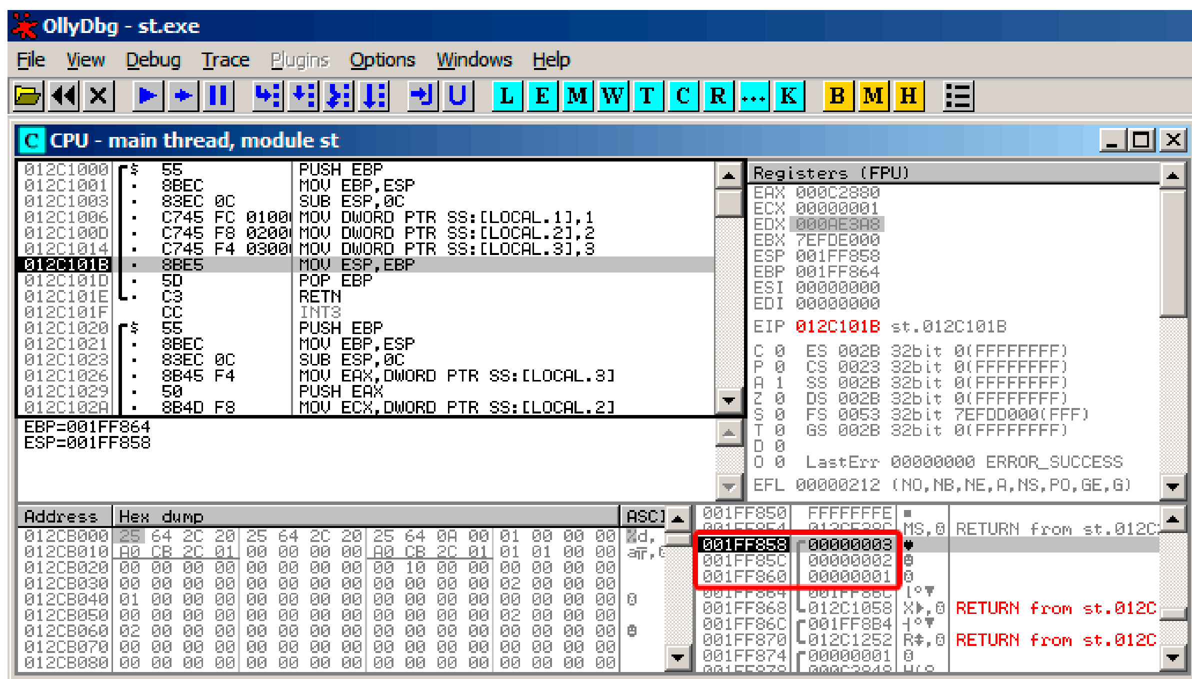


Figura 6.1: OllyDbg: `f1()`

Quando `f1()` assegna le variabili a, b e c, i loro valori sono memorizzati all'indirizzo `0x1FF860` e seguenti.

E quando viene eseguita `f2()`:

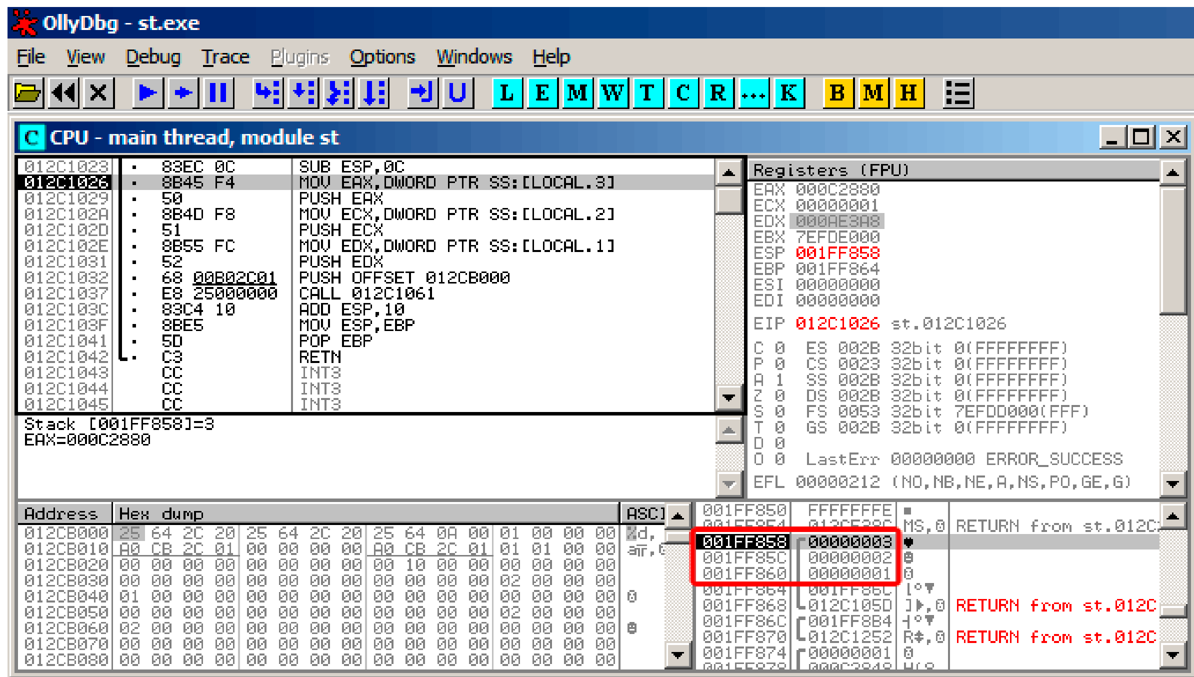


Figura 6.2: OllyDbg: `f2()`

... a, b e c di `f2()` si trovano agli stessi indirizzi! Nessuno ha ancora sovrascritto quei valori, e a quel punto restano intatti. Quindi, affinche' questa strana situazione si verifichi, piu' funzioni devono essere chiamate una dopo l'altra e SP deve essere uguale ad ogni ingresso nella funzione (ovvero le funzioni devono avere lo stesso numero di argomenti). A quel punto le variabili locali si troveranno nelle stesse posizioni nello stack. Per riassumere, tutti i valori nello stack (e nelle celle di memoria in generale) hanno valori lasciati li' dall'esecuzione di funzioni precedenti. Non sono letteralmente randomici, piuttosto hanno valori non predicibili. C'e' un'altra opzione? Sarebbe possibile ripulire porzioni dello stack prima di ogni esecuzione di una funzione, ma sarebbe un lavoro extra inutile.

6.3.1 MSVC 2013

L'esempio e' stato compilato con MSVC 2010. Un lettore di questo libro ha provato a compilare l'esempio con MSVC 2013, lo ha eseguito, ed ha ottenuto i 3 numeri in ordine inverso:

```
c:\Polygon\c>st
3, 2, 1
```

Perche'? Ho compilato anche io l'esempio in MSVC 2013 ed ho visto questo:

Listing 6.5: MSVC 2013

```
_a$ = -12                                                   ; size = 4
_b$ = -8                                                    ; size = 4
_c$ = -4                                                    ; size = 4
_f2     PROC

...

_f2     ENDP

_c$ = -12                                                   ; size = 4
_b$ = -8                                                    ; size = 4
_a$ = -4                                                    ; size = 4
_f1     PROC

...

_f1     ENDP
```

Contrariamente a MSVC 2010, MSVC 2013 ha allocato le variabili a/b/c nella funzione `f2()` in ordine inverso.E cio' e' del tutto corretto, perche' lo standard non ha una regola che definisce in quale ordine le variabili locali devono essere allocate nello stack. La ragione per cui si presenta questa differenza e' che MSVC 2010 ha un solo modo per farlo, mentre MSVC 2013 ha probabilmente subito modifiche all'interno del compilatore, e si comporta quindi in modo leggermente diverso.

6.4

- http://challenges.re/51
- http://challenges.re/52

Capitolo 7

printf() Italian text placeholder

Estendiamo l'esempio (4 on page 10) modificando la chiamata a `printf()` nella funzione `main()`:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

7.1 x86

7.1.1 x86: 3 argomenti

MSVC

Quando compiliamo l'esempio con MSVC 2010 Express otteniamo:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Notiamo che gli argomenti di `printf()` sono messi sullo stack in ordine inverso. Il primo argomento e' quello messo per ultimo.

A proposito, le variabili di tipo *int* in ambienti a 32-bit hanno dimensione pari a 32-bit, ovvero 4 byte.

Abbiamo 4 argomenti, quindi $4 * 4 = 16$ —essi occupano esattamente 16 bytes nello stack: un puntatore a 32-bit alla stringa, e 3 numeri di tipo *int*.

Quando lo stack pointer (registro `ESP`) viene ripristinato dall'istruzione `ADD ESP, X` dopo una chiamata a funzione, in molti casi il numero di argomenti della fuznione puo' essere dedotto semplicemente dividendo X per 4.

Questa caratteristca e' propria solo della calling convention *cdecl* e in ambienti a 32-bit..

Si veda anche la sezione sulle calling conventions (50 on page 345).

In certi casi in qui diverse funzioni ritornano una dopo l'altra, il compilatore potrebbe accorpare piu' istruzioni «ADD ESP, X» dopo l'ultima chiamata, emettendo una sola istruzione:

```
push a1
push a2
call ...
```

```
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Ecco un esempio reale:

Listing 7.1: x86

```
.text:100113E7                 push    3
.text:100113E9                 call    sub_100018B0 ; takes one argument (3)
.text:100113EE                 call    sub_100019D0 ; takes no arguments at all
.text:100113F3                 call    sub_10006A90 ; takes no arguments at all
.text:100113F8                 push    1
.text:100113FA                 call    sub_100018B0 ; takes one argument (1)
.text:100113FF                 add     esp, 8       ; drops two arguments from stack at once
```

MSVC and OllyDbg

Proviamo ora ad esaminare l'esempio con OllyDbg, uno dei piu' diffusi debugger win32 user-land. Possiamo compilarlo in MSVC 2012 con l'opzione `/MD`, che indica al compilatore di linkare con `MSVCR*.DLL`, in maniera tale da poter vedere in modo chiaro, nel debugger, le funzioni importate.

Carichiamo quindi l'eseguibile in OllyDbg. Il primo breakpoint avviene in `ntdll.dll`, premiamo F9 (run) per continuare l'esecuzione. Il secondo breakpoint e' nel codice CRT. Ora dobbiamo trovare la funzione `main()`.

Per farlo scorriamo il codice verso l'alto (MSVC alloca la funzione `main()` proprio all'inizio della sezione code):

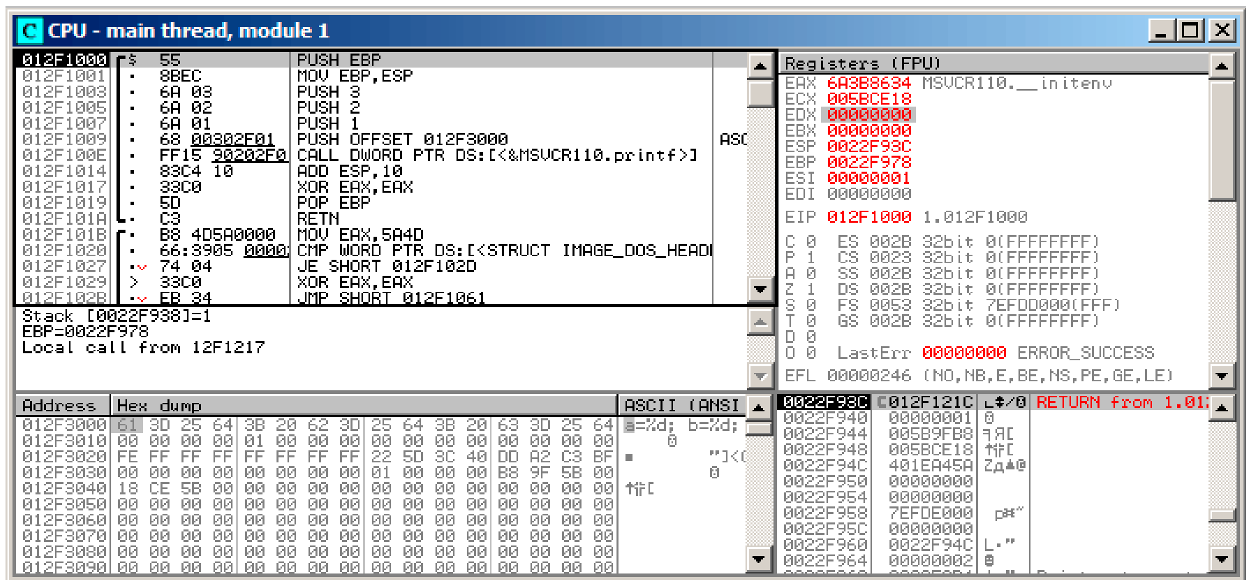


Figura 7.1: OllyDbg: the very start of the `main()` function

Clickiamo sull'istruzione `PUSH EBP`, premiamo F2 (set breakpoint) and quindi F9 (run). Queste azioni ci consentono di saltare tutta la parte legata al codice CRT, in quanto per il momento non ci interessa.

Premiamo F8 () 6 volte, ovverso saltiamo (avanziamo di) 6 istruzioni:

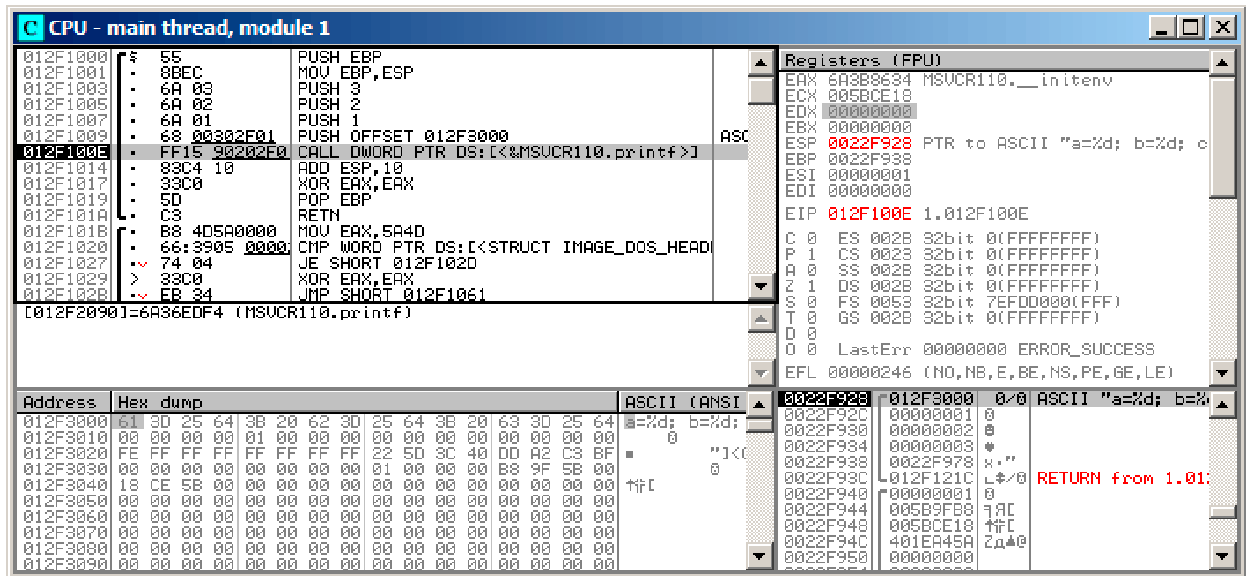


Figura 7.2: OllyDbg: before `printf()` execution

Adesso il PC punta all'istruzione `CALL printf`. OllyDbg, come altri debugger, evidenzia il valore dei registri che sono stati modificati. Quindi ogni volta che si preme F8, `EIP` cambia ed il suo valore e' mostrato in rosso. Anche `ESP` cambia, poiche' i valori degli argomenti vengono messi sullo stack.

Dove sono i valori messi nello stack? Diamo un'occhiata alla finestra del debugger in basso a destra:

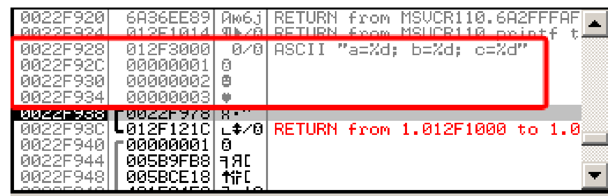


Figura 7.3: OllyDbg: stato dello stack dopo il push dei valori degli argomenti (il rettangolo rosso e' stato aggiunto dall'autore per evidenziare la finestra)

Notiamo 3 colonne: indirizzo nello stack, valore nello stack ed alcuni commenti aggiuntivi di OllyDbg. OllyDbg riconosce le stringhe `printf()`-like , e riporta quindi la stringa insieme ai 3 valori *associati*.

Facendo click destro sulla formato string, quindi click su «Follow in dump», e' possibile vedere la format string nella finestra del debugger in basso a sinistra, che mostra una zona di memoria. I valori mostrati possono anche essere modificati. E' ad esempio possibile cambiare la format string, che renderebbe diverso il risultato dell'esempio. Non e' molto utile in questo particolare caso, ma puo' comunque essere un esercizio utile per iniziare a prendere dimestichezza con lo strumento.

Premiamo F8 ().

Il seguente output viene riportato in console:

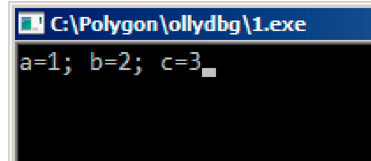


Figura 7.4: `printf()` function executed

Vediamo come sono cambiati i registri e lo stack:

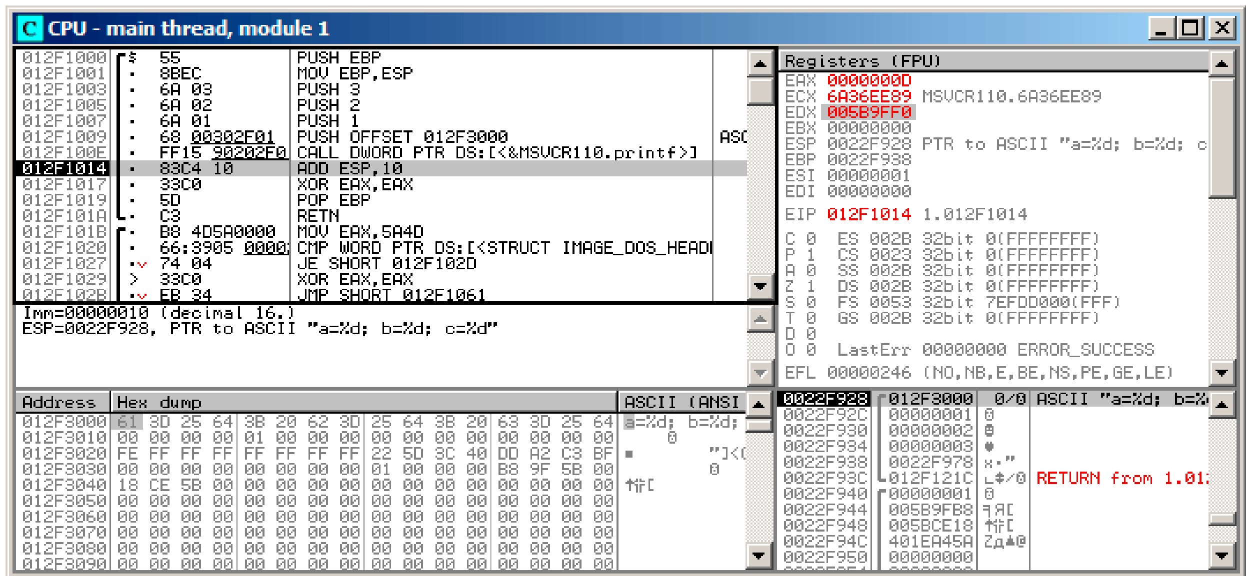


Figura 7.5: OllyDbg dopo dell'esecuzione di `printf()`

Il registro `EAX` adesso contiene `0xD` (13). Questo valore e' corretto, poiche' `printf()` restituisce il numero di caratteri stampati. Il valore di `EIP` e' cambiato: adesso contiene infatti l'indirizzo dell'istruzione che viene dopo `CALL printf`. Anche i valori `ECX` e `EDX` sono cambiati. Apparentemente, i meccanismi interni alla funzione `printf()` hanno usato quei registri durante l'esecuzione di printf per le sue necessita'.

Un fatto molto importante e' che ne' il valore di ESP ne' lo stack sono stati modificati! Vediamo chiaramente che la format string ed i suoi 3 valori si trovano ancora li'. Questo e; infatti il comportamento della calling convention *cdecl*: il callee (la funzione chiamata) non ripristina `ESP` al suo valore precedente. Farlo e' una responsabilita' del The caller (chiamante).

Premiamo F8 nuovamente per eseguire l'istruzione `ADD ESP, 10`:

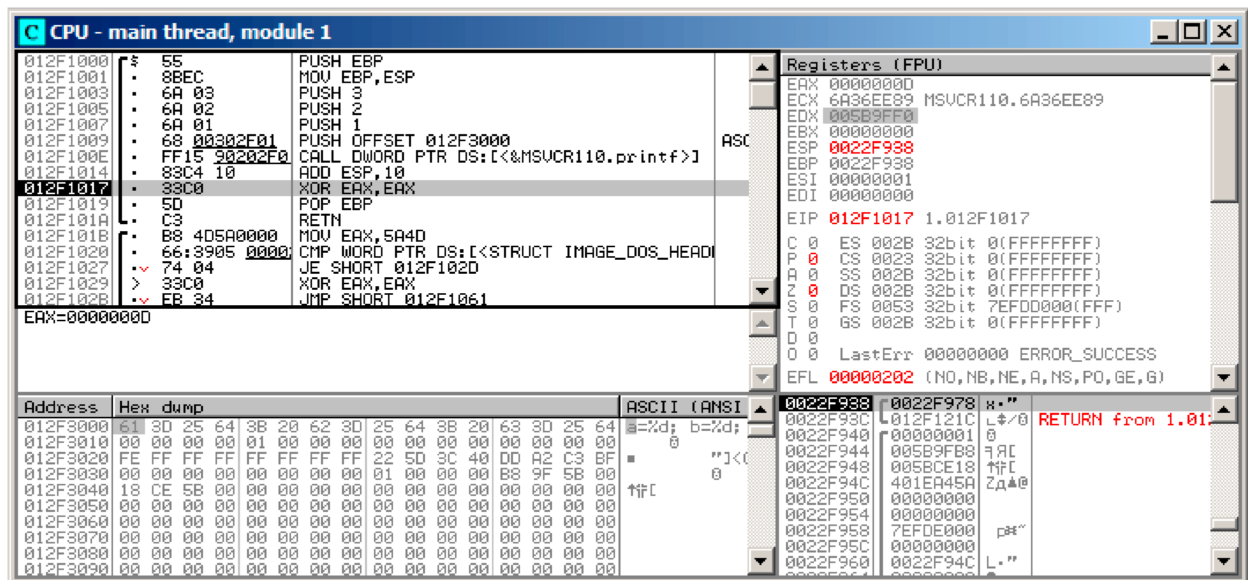


Figura 7.6: OllyDbg: after `ADD ESP, 10` instruction execution

`ESP` e' cambiato, ma i valori si trovano ancora sullo stack! Ovviamente si: non c'e' necessita' di azzerare i valori o effettuare altre simili operazioni di "pulizia". Qualunque cosa si trovi sopra lo stack pointer (SP) e' *noise* o e non ha alcun significato. Pulire i valori non utilizzati nello stack sarebbe una perdita di tempo inutile, e non vi e' alcuna necessita' di farlo.

GCC

Compiliamo adesso lo stesso programma su Linux usando GCC 4.4.1, e diamo un'occhiata al risultato con IDA:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Si nota che la differenza tra il codice prodotto da MSVC e GCC risiede soltanto nel modo in cui gli argomenti sono memorizzati sullo stack. In questo caso GCC lavora diversamente con lo stack senza l'uso di `PUSH`/`POP`.

GCC e GDB

Proviamo l'esempio anche con GDB[1] su Linux.

L'opzione `-g` indica al comiplatore di includere le informazioni di debug nel file eseguibile.

[1] GNU debugger

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 7.2: impostiamo un breakpoint su `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Avviamolo. Non abbiamo il sorgente della funzione `printf()` qui, quindi GDB non puo' mostrarlo, anche se potrebbe.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Stampiamo 10 elementi dello stack elements. La colonna piu' a sinistra contiene l'indirizzo nello stack.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

Il primo elemento e' il RA (`0x0804844a`). Possiamo verificarlo disassemblando la memoria a questo indirizzo:

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Le due istruzioni `XCHG` sono istruzioni «inutili» (idle), analoghe a NOP.

Il secondo elemento (`0x080484f0`) e' l'indirizzo della format string:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

I successivi 3 elementi (1, 2, 3) sono gli argomenti di `printf()`. Il resto degli elementi potrebbe essere «immondizia» nello stack, oppure valori provenienti da altre funzioni, come le loro variabili locali, Italian text placeholder. Per il momento li possiamo ignorare.

Eseguiamo il comando «finish». Questo comando dice a GDB di «eseguire tutte le istruzioni fino alla fine della funzione». In questo caso: esegui fino alla fine di `printf()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB mostr il risultato di `printf()` restituito in `EAX` (13). Questo e' il numero di caratteri stampati, proprio come nell'esempio in OllyDbg.

Vediamo anche «return 0;» e l'informazione che questa espressione si trova nel file `1.c` a riga 6. Infatti il file `1.c` si trova nella directory corrente, e GDB trova la stringa li'. Come fa GDB a sapere quale riga del codice C viene eseguita? Cio' e' dovuto al fatto che il compilatore, quando genera le informazioni di debug, salva anche una tabella di relazioni tra le righe del codice sorgente e gli indirizzi delle istruzioni. Dopotutto GDB e' un «source-level debugger».

Esaminiamo i registri. 13 in `EAX`:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Disassembliamo le istruzioni correnti. La freccia punta all'a prossima istruzione da eseguire.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB usa la sintassi AT&T di default. Ma e' anche possibile passare alla sintassi Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Eseguiamo la prossima istruzione. GDB mostra la parentesi graffa chiusa, che sta a significare la fine del blocco di codice.

```
(gdb) step
7       };
```

Esaminiamo i registri dopo l'esecuzione dell'istruzione `MOV EAX, 0`. `EAX` e' zero.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
```

```
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

7.2 ARM

7.2.1 ARM: 3 arguments

Lo schema tradizionale per il passaggio di argomenti (calling convention) di ARM si comporta in questo modo: i primi 4 argomenti vengono passati attraverso i registri `R0` - `R3` , i restanti attraverso lo stack. Cio' ricorda molto il metodo per il passaggio di argomenti in fastcall (50.3 on page 346) o win64 (50.5.1 on page 347).

32-bit ARM

Keil 6/2013 ()

Listing 7.3: Keil 6/2013 ()

```
.text:00000000 main
.text:00000000 10 40 2D E9   STMFD   SP!, {R4,LR}
.text:00000004 03 30 A0 E3   MOV     R3, #3
.text:00000008 02 20 A0 E3   MOV     R2, #2
.text:0000000C 01 10 A0 E3   MOV     R1, #1
.text:00000010 08 00 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB   BL      __2printf
.text:00000018 00 00 A0 E3   MOV     R0, #0          ; return 0
.text:0000001C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

I primi 4 argomenti sono quindi passati attraverso i registri `R0` - `R3` nel seguente ordine: un puntatore alla format string di `printf()` in `R0` , 1 in `R1` , 2 in `R2` e 3 in `R3` . L'istruzione a `0x18` scrive 0 in `R0` —questo equivale allo statement C *return 0*. Niente di insolito fino a qui.

Keil 6/2013 genera lo stesso codice.

Keil 6/2013 ()

Listing 7.4: Keil 6/2013 ()

```
.text:00000000 main
.text:00000000 10 B5         PUSH    {R4,LR}
.text:00000002 03 23         MOVS    R3, #3
.text:00000004 02 22         MOVS    R2, #2
.text:00000006 01 21         MOVS    R1, #1
.text:00000008 02 A0         ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d"
.text:0000000A 00 F0 0D F8   BL      __2printf
.text:0000000E 00 20         MOVS    R0, #0
.text:00000010 10 BD         POP     {R4,PC}
```

Non c'e' nessuna differenza significativa nel codice non ottimizzato per modo ARM.

Keil 6/2013 () + rimozione di return

Modifichiamo leggermente l'esempio rimuovendo *return 0*:

```
#include <stdio.h>

void main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

Il risultato e' alquanto insolito:

Listing 7.5: Keil 6/2013 ()

```
.text:00000014 main
.text:00000014 03 30 A0 E3   MOV     R3, #3
.text:00000018 02 20 A0 E3   MOV     R2, #2
.text:0000001C 01 10 A0 E3   MOV     R1, #1
.text:00000020 1E 0E 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA   B       __2printf
```

Questa e' la versione ottimizzata (`-O3`) per ARM mode e stavolta notiamo `B` come ultima istruzione, al posto della familiare `BL` . Un'altra differenza tra questa versione ottimizzata e la precedente (compilata senza ottimizzazione) e' la mancanza di prologo ed epilogo della funzione (le istruzioni che preservano i valori dei registri `R0` e LR). L'istruzione `B` salta semplicemente ad un altro indirizzo, senza alcuna manipolazione del registro LR, in modo simile a `JMP` in x86. Perche' funziona? Perche' questo codice e' infatti equivalente al precedente. Principalmente per due motivi: 1) ne' lo stack ne' SP (lo stack pointer) vengono modificati; 2) la chiamata a `printf()` e' l'ultima istruzionie, quindi non succede niente dopo di essa. Al completamento, `printf()` restituisce semplicemente il controllo all'indirizzo memorizzato in LR. Poiche' LR attualmente contiene l'indirizzo del punto da cui la nostra funzione era stata chiamata, il controllo verra' restituito da `printf()` a quello stesso punto. Pertanto non c'e' alcun bisogno di salvare LR in quanto non abbiamo necessita' di modificare LR. E non vogliamo affatto modificare LR poiche' non ci sono altre chiamate a funzione ad eccezzione di `printf()`. Inoltre, dopo questa chiamata non abbiamo nient'altro da fare! Queste le ragione per cui una simile ottimizzazione e' possibile.

Questa ottimizzazione e' spesso usata in funzioni in cui l'ultimo statement e' una chiamata ad un'altra funzione. Un esempio simile e' fornito di seguito: **??** on page ??.

ARM64

GCC (Linaro) 4.9

Listing 7.6: GCC (Linaro) 4.9

```
.LC1:
        .string "a=%d; b=%d; c=%d"
f2:
; save FP and LR in stack frame:
        stp     x29, x30, [sp, -16]!
; set stack frame (FP=SP):
        add     x29, sp, 0
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        bl      printf
        mov     w0, 0
; restore FP and LR
        ldp     x29, x30, [sp], 16
        ret
```

La prima istruzione `STP` (*Store Pair*) salva FP[2] (X29) e LR (X30) nello stack. La seconda istruzione `ADD X29, SP, 0` forma lo stack frame. Scrive semplicemente il valore di SP in X29.

Successivamente vediamo la familiare coppia di istruzioni `ADRP` / `ADD`, che forma un puntatore alla stringa. *lo12* indica 12 bit bassi (low 12 bits), ovvero, il linker scrivera' i 12 bit bassi dell'indirizzo di LC1 nell'opcode dell'istruzione `ADD`. `%d` nella format string di `printf()` e' un *int* a 32-bit, quindi u valori 1, 2 e 3 sono caricati nelle parti a 32-bit dei registri.

GCC (Linaro) 4.9 genera lo stesso codice.

7.2.2 ARM: 8 arguments

Usiamo nuovamente l'esempio con 9 argomenti della sezione precedente: **??** on page ??.

[2] Frame Pointer

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

Keil 6/2013:

```
.text:00000028             main
.text:00000028
.text:00000028             var_18 = -0x18
.text:00000028             var_14 = -0x14
.text:00000028             var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR     LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB     SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV     R3, #8
.text:00000034 07 20 A0 E3  MOV     R2, #7
.text:00000038 06 10 A0 E3  MOV     R1, #6
.text:0000003C 05 00 A0 E3  MOV     R0, #5
.text:00000040 04 C0 8D E2  ADD     R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA   R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV     R0, #4
.text:0000004C 00 00 8D E5  STR     R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV     R3, #3
.text:00000054 02 20 A0 E3  MOV     R2, #2
.text:00000058 01 10 A0 E3  MOV     R1, #1
.text:0000005C 6E 0F 8F E2  ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g↙
    ↘ =%"...
.text:00000060 BC 18 00 EB  BL      __2printf
.text:00000064 14 D0 8D E2  ADD     SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR     PC, [SP+4+var_4],#4
```

Il codice puo' essre diviso in piu' parti

- Preambolo della funzione:

  La prima istruzione `STR LR, [SP,#var_4]!` salva LR sullo stack, poiche' questo registro sara' usato per la chiamata a `printf()`. Il punto esclamativo all fine indica il *pre-index*.

  Questo implica che SP deve essere prima decrementato di 4, e successivamente LR sara' salvato all'indirizzo memorizzato in SP. Tutto cio' e' simile a `PUSH` in x86. Maggiori informazioni qui: 20.2 on page 150.

  La seconda istruzione `SUB SP, SP, #0x14` decrementa SP (lo stack pointer) per allocare `0x14` (20) byte sullo stack. Infatti dobbiamo passare 5 valori a 32-bit tramite lo stack per la funzione `printf()`, e ciascuno di essi occupa 4 byte, che e' esattamente $5 * 4 = 20$. Gli altri 4 valori a 32-bit saranno passati tramite registri.

- Passaggio di 5, 6, 7 e 8 tramite lo stack: sono memorizzati nei registri `R0`, `R1`, `R2` e `R3`, rispettivamente. Successivamente l'istruzione `ADD R12, SP, #0x18+var_14` scrive l'indirizzo dello stack, dove queste 4 variabili saranno memorizzate, nel registri `R12`. *var_14* e' una macro assembly, uguale a -0x14, creata da IDA per visualizzare in maniera conveniente il codice che accede allo stack. Le macro *var_?* generate da IDA riflettono le variabili locali nello stack.

  Quindi, `SP+4` sara' memorizzato nel registro `R12`. L'istruzione successiva `STMIA R12, R0-R3` scrive il contenuto dei registri `R0` - `R3` alla memoria puntata da `R12`. `STMIA` e' abbreviazione per *Store Multiple Increment After*. *«Increment After»* (incrementa dopo) implica che `R12` deve essere incrementato di 4 dopo ciascuna scrittura di un valore nei registri.

- Passaggio di 4 tramite lo stack: 4 e' memorizzato in `R0` e questo valore, con l'aiuto dell'istruzione `STR R0, [SP,#0x18+var_` viene salvato sullo stack. *var_18* e' -0x18, quindi l'offset deve essere 0, da cui il valore dal registro `R0` (4) sara' scritto all'indirizzo memorizzato in SP.

- Passaggio di 1, 2 e 3 tramite registri: I valori dei primi 3 numeri (a, b, c) (1, 2, 3 rispettivamente) sono passati attraverso i registri `R1`, `R2` e `R3` poco prima della chiamata a `printf()` call, e gli altri 5 valori sono passati tramite lo stack:

- chiamata a `printf()` .
- epilogo della funzione:

  L'istruzione `ADD SP, SP, #0x14` ripristina il puntatore SP al suo valore precedente, pulendo quindi lo stack. Ovviamente quello che era stato memorizzato nello stack rimarra li', e sara' probabilmente riscritto interamente durante l'e secuzione delle funzioni seguenti.

  L'istruzione `LDR PC, [SP+4+var_4],#4` carica il valore di LR salvato dallo stack nel nel registro PC, causando quindi l'uscita dalla funzione. Non c'e' il punto esclamativo —infatti PC e' caricato prima dall'indirizzo memorizzato in SP ($4 + var_4 = 4 + (-4) = 0$, questa istruzione e' quindi analoga a `LDR PC, [SP],#4`), e successivamente SP e' incrementato di 4. Questo e' detto *post-index*[3]. Perche' IDA mostra l'istruzione in quel modo? Perche' vuole illustrare il layout dello stack ed il fatto che `var_4` e' allocata per salvare il valore di LR nello stack locale. Questa istruzione e' piu' o meno simile a `POP PC` in x86[4].

Keil 6/2013:

```
.text:0000001C             printf_main2
.text:0000001C
.text:0000001C             var_18 = -0x18
.text:0000001C             var_14 = -0x14
.text:0000001C             var_8  = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; ↙
    ↳ g=%"...
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E             loc_3E    ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD     SP, SP, #0x14
.text:00000040 00 BD        POP     {PC}
```

L'output e' quasi identico al precedente esempio. Tuttavia questo e' codice Thumb e i valori sono disposti nello stack in modo differente: 8 per primo, quindi 5, 6, 7, e infine 4.

Xcode 4.6.3 (LLVM):

```
__text:0000290C             _printf_main2
__text:0000290C
__text:0000290C             var_1C = -0x1C
__text:0000290C             var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1   MOV    R7, SP
__text:00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3   MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3   MOV    R12, #7
__text:00002920 00 00 40 E3   MOVT   R0, #0
__text:00002924 04 20 A0 E3   MOV    R2, #4
__text:00002928 00 00 8F E0   ADD    R0, PC, R0
```

[3]Maggiori dettagli: 20.2 on page 150.

[4]E' impossibile settare il valore di `IP/EIP/RIP` usando `POP` in x86, ma in ogni caso hai capito l'analogia.

```
__text:0000292C 06 30 A0 E3   MOV     R3, #6
__text:00002930 05 10 A0 E3   MOV     R1, #5
__text:00002934 00 20 8D E5   STR     R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9   STMFA   SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3   MOV     R9, #8
__text:00002940 01 10 A0 E3   MOV     R1, #1
__text:00002944 02 20 A0 E3   MOV     R2, #2
__text:00002948 03 30 A0 E3   MOV     R3, #3
__text:0000294C 10 90 8D E5   STR     R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB   BL      _printf
__text:00002954 07 D0 A0 E1   MOV     SP, R7
__text:00002958 80 80 BD E8   LDMFD   SP!, {R7,PC}
```

Quasi lo stesso codice visto prima, ad eccezione dell'istruzione `STMFA` (Store Multiple Full Ascending), che e' sinonimo di `STMIB` (Store Multiple Increment Before). Questa istruzione incrementa il valore nel registro SP e solo successivamente scrive il prossimo valore del registro in memoria, invece che operare le due azioni in ordine inverso.

Un'altra cosa che salta all'occhio e' che le istruzioni sono disposte in maniera apparentemente casuale. Ad esempio, il valore nel registro `R0` e' manipolato in tre posti diversi agli indirizzi `0x2918`, `0x2920` e `0x2928`, quando invece sarebbe stato possibile farlo in un punto solo.

Ad ogni modo, il compilatore ottimizzante avra' avuto le sue ragioni per ordinare le istruzioni in questa maniera ed ottenere una maggiore efficacia durante l'esecuzione del codice.

Solitamente il processero prova ad eseguire simultaneamentele istruzioni vicine. Ad esempio, istruzioni come `MOVT R0, #0` e `ADD R0, PC, R0` non possono essere eseguite simultaneamente poiche' entrambe modificano il registro `R0`. D'altra parte, `MOVT R0, #0` e `MOV R2, #4` possono invece essere eseguite simultaneamente poiche' l'effetto della loro esecuzione non genera conflitti tra loro. Presumibilmente, il compilatore prova a generare codice in questo modo (quando possibile).

Xcode 4.6.3 (LLVM):

```
__text:00002BA0               _printf_main2
__text:00002BA0
__text:00002BA0               var_1C = -0x1C
__text:00002BA0               var_18 = -0x18
__text:00002BA0               var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5          PUSH     {R7,LR}
__text:00002BA2 6F 46          MOV      R7, SP
__text:00002BA4 85 B0          SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20    MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C    MOV.W    R12, #7
__text:00002BAE C0 F2 00 00    MOVT.W   R0, #0
__text:00002BB2 04 22          MOVS     R2, #4
__text:00002BB4 78 44          ADD      R0, PC  ; char *
__text:00002BB6 06 23          MOVS     R3, #6
__text:00002BB8 05 21          MOVS     R1, #5
__text:00002BBA 0D F1 04 0E    ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92          STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09    MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10    STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21          MOVS     R1, #1
__text:00002BCA 02 22          MOVS     R2, #2
__text:00002BCC 03 23          MOVS     R3, #3
__text:00002BCE CD F8 10 90    STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA    BLX      _printf
__text:00002BD6 05 B0          ADD      SP, SP, #0x14
__text:00002BD8 80 BD          POP      {R7,PC}
```

L'output e' quasi lo stesso dell'esempio precedente, ad eccezione dell'uso di istruzioni Thumb.

ARM64

GCC (Linaro) 4.9

Listing 7.7: GCC (Linaro) 4.9

```
.LC2:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; grab more space in stack:
        sub     sp, sp, #32
; save FP and LR in stack frame:
        stp     x29, x30, [sp,16]
; set stack frame (FP=SP):
        add     x29, sp, 16
        adrp    x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
        add     x0, x0, :lo12:.LC2
        mov     w1, 8           ; 9th argument
        str     w1, [sp]        ; store 9th argument in the stack
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; restore FP and LR
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret
```

I primi 8 argomenti sono passati nei registri X- o W-: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][5]. Un puntatore ad una string richiede un registro a 64-bit, quindi e' passato in `X0`. Tutti gli altri valori hanno timpo *int* a 32-bit, quindi sono memorizzati nella parte a 32-bit dei registri (W-). Il nono argomento (8) e' passato tramite lo stack. Infatti non e' possibile passare un grande numero di argomenti tramite registri, in quanto il loro numero e' limitato.

GCC (Linaro) 4.9 genera lo stesso codice.

7.3 MIPS

7.3.1 3 argomenti

GCC 4.4.5

La differenza principale con l'esempio «» e' che in questo caso `printf()` e' chiamata al posto di `puts()`, e 3 argomenti aggiuntivi sono passati attraverso i registri $5…$7 (o $A0…$A2). Questo e' il motivo per cui questi registri hanno il prefisso A-, che implica il loro uso per il passaggio di argomenti di funzioni.

Listing 7.8: GCC 4.4.5 ()

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; function prologue:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; load address of printf():
        lw      $25,%call16(printf)($28)
; load address of the text string and set 1st argument of printf():
        lui     $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; set 2nd argument of printf():
        li      $5,1                    # 0x1
; set 3rd argument of printf():
        li      $6,2                    # 0x2
```

[5] http://go.yurichev.com/17287

```
; call printf():
        jalr    $25
; set 4th argument of printf() (branch delay slot):
        li      $7,3                    # 0x3

; function epilogue:
        lw      $31,28($sp)
; set return value to 0:
        move    $2,$0
; return
        j       $31
        addiu   $sp,$sp,32 ; branch delay slot
```

Listing 7.9: GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
.text:00000010                 sw      $gp, 0x20+var_10($sp)
; load address of printf():
.text:00000014                 lw      $t9, (printf & 0xFFFF)($gp)
; load address of the text string and set 1st argument of printf():
.text:00000018                 la      $a0, $LC0        # "a=%d; b=%d; c=%d"
; set 2nd argument of printf():
.text:00000020                 li      $a1, 1
; set 3rd argument of printf():
.text:00000024                 li      $a2, 2
; call printf():
.text:00000028                 jalr    $t9
; set 4th argument of printf() (branch delay slot):
.text:0000002C                 li      $a3, 3
; function epilogue:
.text:00000030                 lw      $ra, 0x20+var_4($sp)
; set return value to 0:
.text:00000034                 move    $v0, $zero
; return
.text:00000038                 jr      $ra
.text:0000003C                 addiu   $sp, 0x20 ; branch delay slot
```

IDA ha fuso le coppie di istruzioni `LUI` e `ADDIU` in una unica pseudoistruzione `LA`. Questo e' il motivo per cui non c'e' nessuna istruzione all'indirizzo 0x1C: perche' `LA` *occupa* 8 byte.

GCC 4.4.5

GCC e' piu' verboso:

Listing 7.10: GCC 4.4.5 ()

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; function prologue:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
; load address of the text string:
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
```

```
; set 1st argument of printf():
        move    $4,$2
; set 2nd argument of printf():
        li      $5,1                    # 0x1
; set 3rd argument of printf():
        li      $6,2                    # 0x2
; set 4th argument of printf():
        li      $7,3                    # 0x3
; get address of printf():
        lw      $2,%call16(printf)($28)
        nop
; call printf():
        move    $25,$2
        jalr    $25
        nop

; function epilogue:
        lw      $28,16($fp)
; set return value to 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
; return
        j       $31
        nop
```

Listing 7.11: GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; load address of the text string:
.text:0000001C                 la      $v0, aADBDCD     # "a=%d; b=%d; c=%d"
; set 1st argument of printf():
.text:00000024                 move    $a0, $v0
; set 2nd argument of printf():
.text:00000028                 li      $a1, 1
; set 3rd argument of printf():
.text:0000002C                 li      $a2, 2
; set 4th argument of printf():
.text:00000030                 li      $a3, 3
; get address of printf():
.text:00000034                 lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038                 or      $at, $zero
; call printf():
.text:0000003C                 move    $t9, $v0
.text:00000040                 jalr    $t9
.text:00000044                 or      $at, $zero ; NOP
; function epilogue:
.text:00000048                 lw      $gp, 0x20+var_10($fp)
; set return value to 0:
.text:0000004C                 move    $v0, $zero
.text:00000050                 move    $sp, $fp
.text:00000054                 lw      $ra, 0x20+var_4($sp)
.text:00000058                 lw      $fp, 0x20+var_8($sp)
.text:0000005C                 addiu   $sp, 0x20
; return
.text:00000060                 jr      $ra
```

```
.text:00000064                  or      $at, $zero ; NOP
```

7.3.2 8 argomenti

Usiamo nuovamente l'esempio con 9 argomenti dalla sezione prcedente: **??** on page ??.

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

GCC 4.4.5

Solo i primi 4 argomenti sono passati nei registri $A0 …$A3, gli altri sono passati tramite lo stack.

Questa e' la calling convention O32 (che e' la piu' comune nel mondo MIPS). Altre calling conventions (come N32) possono usare i registri per scopi diversi.

`SW` e' l'abbreviazione di «Store Word» (da un registro alla memoria). MIPS manca di istruzioni per memorizzare un valore in memoria, e' quindi necessario usare una coppia di istruzioni (LI/SW).

Listing 7.12: GCC 4.4.5 ()

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-56
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,52($sp)
; pass 5th argument in stack:
        li      $2,4                    # 0x4
        sw      $2,16($sp)
; pass 6th argument in stack:
        li      $2,5                    # 0x5
        sw      $2,20($sp)
; pass 7th argument in stack:
        li      $2,6                    # 0x6
        sw      $2,24($sp)
; pass 8th argument in stack:
        li      $2,7                    # 0x7
        lw      $25,%call16(printf)($28)
        sw      $2,28($sp)
; pass 1st argument in $a0:
        lui     $4,%hi($LC0)
; pass 9th argument in stack:
        li      $2,8                    # 0x8
        sw      $2,32($sp)
        addiu   $4,$4,%lo($LC0)
; pass 2nd argument in $a1:
        li      $5,1                    # 0x1
; pass 3rd argument in $a2:
        li      $6,2                    # 0x2
; call printf():
        jalr    $25
; pass 4th argument in $a3 (branch delay slot):
        li      $7,3                    # 0x3

; function epilogue:
        lw      $31,52($sp)
; set return value to 0:
        move    $2,$0
; return
```

```
        j       $31
        addiu   $sp,$sp,56 ; branch delay slot
```

Listing 7.13: GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x38
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                 sw      $ra, 0x38+var_4($sp)
.text:00000010                 sw      $gp, 0x38+var_10($sp)
; pass 5th argument in stack:
.text:00000014                 li      $v0, 4
.text:00000018                 sw      $v0, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000001C                 li      $v0, 5
.text:00000020                 sw      $v0, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000024                 li      $v0, 6
.text:00000028                 sw      $v0, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000002C                 li      $v0, 7
.text:00000030                 lw      $t9, (printf & 0xFFFF)($gp)
.text:00000034                 sw      $v0, 0x38+var_1C($sp)
; prepare 1st argument in $a0:
.text:00000038                 lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d↙
    ↳ ; g=%"...
; pass 9th argument in stack:
.text:0000003C                 li      $v0, 8
.text:00000040                 sw      $v0, 0x38+var_18($sp)
; pass 1st argument in $a1:
.text:00000044                 la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d; c=%d; d=%d; e=%d; f↙
    ↳ =%d; g=%"...
; pass 2nd argument in $a1:
.text:00000048                 li      $a1, 1
; pass 3rd argument in $a2:
.text:0000004C                 li      $a2, 2
; call printf():
.text:00000050                 jalr    $t9
; pass 4th argument in $a3 (branch delay slot):
.text:00000054                 li      $a3, 3
; function epilogue:
.text:00000058                 lw      $ra, 0x38+var_4($sp)
; set return value to 0:
.text:0000005C                 move    $v0, $zero
; return
.text:00000060                 jr      $ra
.text:00000064                 addiu   $sp, 0x38 ; branch delay slot
```

GCC 4.4.5

GCC e’ piu’ verboso:

Listing 7.14: GCC 4.4.5 ()

```
$LC0:
        .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; function prologue:
```

```
        addiu   $sp,$sp,-56
        sw      $31,52($sp)
        sw      $fp,48($sp)
        move    $fp,$sp
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
        lui     $2,%hi($LC0)
        addiu   $2,$2,%lo($LC0)
; pass 5th argument in stack:
        li      $3,4                    # 0x4
        sw      $3,16($sp)
; pass 6th argument in stack:
        li      $3,5                    # 0x5
        sw      $3,20($sp)
; pass 7th argument in stack:
        li      $3,6                    # 0x6
        sw      $3,24($sp)
; pass 8th argument in stack:
        li      $3,7                    # 0x7
        sw      $3,28($sp)
; pass 9th argument in stack:
        li      $3,8                    # 0x8
        sw      $3,32($sp)
; pass 1st argument in $a0:
        move    $4,$2
; pass 2nd argument in $a1:
        li      $5,1                    # 0x1
; pass 3rd argument in $a2:
        li      $6,2                    # 0x2
; pass 4th argument in $a3:
        li      $7,3                    # 0x3
; call printf():
        lw      $2,%call16(printf)($28)
        nop
        move    $25,$2
        jalr    $25
        nop
; function epilogue:
        lw      $28,40($fp)
; set return value to 0:
        move    $2,$0
        move    $sp,$fp
        lw      $31,52($sp)
        lw      $fp,48($sp)
        addiu   $sp,$sp,56
; return
        j       $31
        nop
```

Listing 7.15: GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; function prologue:
.text:00000000                 addiu   $sp, -0x38
.text:00000004                 sw      $ra, 0x38+var_4($sp)
.text:00000008                 sw      $fp, 0x38+var_8($sp)
.text:0000000C                 move    $fp, $sp
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x38+var_10($sp)
```

```
.text:0000001C                  la      $v0, aADBDCDDDEDFDGD  # "a=%d; b=%d; c=%d; d=%d; e=%d; f↙
    ↘ =%d; g=%"...
; pass 5th argument in stack:
.text:00000024                  li      $v1, 4
.text:00000028                  sw      $v1, 0x38+var_28($sp)
; pass 6th argument in stack:
.text:0000002C                  li      $v1, 5
.text:00000030                  sw      $v1, 0x38+var_24($sp)
; pass 7th argument in stack:
.text:00000034                  li      $v1, 6
.text:00000038                  sw      $v1, 0x38+var_20($sp)
; pass 8th argument in stack:
.text:0000003C                  li      $v1, 7
.text:00000040                  sw      $v1, 0x38+var_1C($sp)
; pass 9th argument in stack:
.text:00000044                  li      $v1, 8
.text:00000048                  sw      $v1, 0x38+var_18($sp)
; pass 1st argument in $a0:
.text:0000004C                  move    $a0, $v0
; pass 2nd argument in $a1:
.text:00000050                  li      $a1, 1
; pass 3rd argument in $a2:
.text:00000054                  li      $a2, 2
; pass 4th argument in $a3:
.text:00000058                  li      $a3, 3
; call printf():
.text:0000005C                  lw      $v0, (printf & 0xFFFF)($gp)
.text:00000060                  or      $at, $zero
.text:00000064                  move    $t9, $v0
.text:00000068                  jalr    $t9
.text:0000006C                  or      $at, $zero ; NOP
; function epilogue:
.text:00000070                  lw      $gp, 0x38+var_10($fp)
; set return value to 0:
.text:00000074                  move    $v0, $zero
.text:00000078                  move    $sp, $fp
.text:0000007C                  lw      $ra, 0x38+var_4($sp)
.text:00000080                  lw      $fp, 0x38+var_8($sp)
.text:00000084                  addiu   $sp, 0x38
; return
.text:00000088                  jr      $ra
.text:0000008C                  or      $at, $zero ; NOP
```

7.4

Si riporta di seguito una lista di bozze di chiamate alla call:

Listing 7.16: x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; modify stack pointer (if needed)
```

Listing 7.17: x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)
```

Listing 7.18: x64 (GCC)

```
MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument
...
PUSH 7th, 8th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)
```

Listing 7.19: ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; pass 5th, 6th argument, etc, in stack (if needed)
BL function
; modify stack pointer (if needed)
```

Listing 7.20: ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; pass 9th, 10th argument, etc, in stack (if needed)
BL CALL function
; modify stack pointer (if needed)
```

Listing 7.21: MIPS (O32 calling convention)

```
LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; pass 5th, 6th argument, etc, in stack (if needed)
LW temp_reg, address of function
JALR temp_reg
```

7.5 A proposito...

Le differenze negli approcci utilizzati per il passaggio di argomenti in x86, x64, fastcall, ARM and MIPS e' un'ottima dimostrazione del fatto che la CPU e' inconsapevole di come gli argomenti vengono passati alle funzioni. Sarebbe anche possibile creare un compilatore ipotetico in grado di passare gli argomenti attraverso una struttura speciale, senza usare lo stack.

I registri MIPS $A0 ...$A3 sono indicati in questo modo soltanto per convenienza (cioe' nella O32 calling convention). I programmatori possono usare qualunque altro registro (tranne $ZERO) per passare i dati, o utilizzare qualunque altra calling convention.

La CPU non e' assolutamente consapevole delle calling conventions.

Possiamo anche ricordare come i programmatori principianti in assembly passano gli argomenti alle altre funzioni: di solito tramite i registri, senza un ordine esplicitop, o attraverso variabili globali. Questi approcci sono ovviamente validi e funzionanti.