

## Using the Gnu AS Assembler – FEATURE.ARTICLE

Pubblicato da *mammon\_* il 01/1999

Livello *base*

### Introduzione

/\* Questo articolo e' stato tratto dall'Assembly Programming Journal numero 2 e tradotto da Little-Jonh \*/

```

::/ \:.....
:/___\:.....
/|   \:.....
:|   _/\:.....
:|   _|\ \:.....
::\___\:.....FEATURE.ARTICLE

```

### Iniziamo

GAS e' parte del progetto GNU dell'assembler Unix AS; e' disponibile come partedel pacchetto binutils package incluso in ogni compilatore GNU (per esempio, GCC). GAS e' supportato nei vari compilatori GNU, e quindi GAS puo' essere utilizzato eseguendo il compilatore su un file .S (sorgente asm); comunque puo' anche essere eseguito su un qualsiasi file sorgente (per esempio, .asm) utilizzando il comando 'as'.

La documentazione GAS e' disponibile nelle installazioni Linux in formato infoThe GAS (.gz), ed e' visualizzabile usando il comando 'info as' o 'info -f as.info'.

Per i principianti, un crash course su info: i file sono strutturati ad albero, con ogni sezione considerata come un 'nodo'; h e' l'help, q chiude info, SPAZIO scrolla il testo in giu', DEL scrolla su il testo, b salta all'inizio del nodo, e salta alla fine del nodo, n salta al nodo successivo, p salta al nodo precedente, g salta al nodo specificato, m salto all'item di menu specificato, s cerca nel file info, e l va indietro di un nodo.

La sezione di piu' alto interesse nel manuale sara' quella sulle Direttive ('g Pseudo Ops'), Simboli ('g Symbols'), Costanti ('g Constants'), e Sezioni ('g Sections'). Per una reference piu' immediata, possono essere consultati gli argomenti specifici sull'Intel 386: 'g i386-Syntax', 'g i386-Opcodes', 'g i386-Regs', 'g i386-prefixes', 'g i386-Memory', 'g i386-jumps'.

### La sintassi AT&T

GAS usa la sintassi AT&T, nota per la confusione che crea a coloro che conoscono la sintassi degli assembler Intel. E' stato detto che la sintassi AT&T e' meno ambigua della Intel, e quindi ha il suo fascino.

### Registri

Una delle differenze lampanti nella sintassi e' la notazione utilizzata per i registri, che sono preceduti dal %. Quindi, 'eax ax al ah' saranno scritti '%eax %ax %al %ah' per il GAS.

### Formato degli Opcode e Ordine

Diversamente dalla sintassi Intel che usa il formato 'opcode dest, orig', la sintassi AT&T usa il formato 'opcode orig, dest'; quindi il comando 'mov eax, ebx' in Intel sara' 'mov %ebx, %eax' in AT&T. Inoltre gli opcode della sintassi AT&T portano tutti il suffisso che specifica la dimensione dell'operando (nota che questi suffissi possono essere normalmente ignorati, dato che GAS indovinera' la dimensione degli operandi dalla dimensione del registro che viene utilizzato)— quindi si aggiungera' 'w' ad un opcode che per specificare un operando word, 'b' per specificare un operando byte, e 'l' per specificare un operando long. L'opcode Intel 'mov' verra' 'trasformato' nella sintassi AT&T in 'movb', 'movw', o 'movl' a seconda delle circostanze. Nota che cio' riguarda anche le far calls; dato che la keyword 'FAR' non esiste in GAS, si DEVE prefissare (non suffissare) la call o il jump con il : quindi una 'far call' diventa 'lcall', 'far jmp' diventa 'ljmp', e 'ret far' diventa 'lret'.

### Valori Immediati e Assoluti

I valori immediati sono preceduti da un \$ nella sintassi AT&T, mentre nella sintassi Intel non sono marcati. Quindi una 'push 4' diventa 'push \$4' in AT&T. Anche un valore assoluto e' prefissato da un \*, mentre in Intel non e' marcato.

### Referenze alla memoria

Questa e' la parte che crea piu' confusione per chi ha utilizzato la sintassi Intel.

Intel usa la seguente sintassi per riferirsi alla memoria: SECTION:[BASE + INDEX\*SCALE + DISP] dove BASE e' il registro usato come base nella reference, INDEX e' il registro utilizzato per calcolare l'offset, SCALE e' il moltiplicatore usato per calcolare l'offset dal registro INDEX, e DISP e' il displacement da BASE o dal registro INDEX.

Alcuni esempi dal manuale di GAS:

```
[ebp - 4][BASE DISP] (Note: DISP is -4)
[foo + eax*4] [DISP + INDEX*SCALE]
[foo] [DISP] (Valore a cui si punta da 'foo')
gs:foo SECTION:DISP(Contenuto della variabile 'foo')
```

La AT&T usa la seguente sintassi per le referenze a memoria: SECTION:DISP(BASE, INDEX, SCALE). Come nella sintassi Intel, non sono tutti necessari (e sembra che BASE e INDEX sono raramente usati insieme). Il manuale di GAS fornisce i seguenti esempi equivalenti agli esempi Intel sopra:

```
-4(%ebp) DISP(BASE)
foo(,%eax,4) DISP(,INDEX,SCALE)
foo(,1) DISP(,SCALE) (Nota: una sola virgola e' intenzionale)
%gs:foo SECTION:DISP
```

Nota che devi inserire nelle parentesi una virgola ogni volta che salti un elemento (e.g., se non usi BASE).

Per altri esempi, ecco altri esempi di memory references con alcuni opcodes (da <http://www.castle.net/~avly/djasm.html>):

__AT&T	__Intel
movl 4(%ebp), %eax	mov eax, [ebp+4]
addl (%eax,%eax,4), %ecx	add ecx, [eax + eax*4]
movb \$4, %fs:(%eax)	mov fs:eax, 4
movl _array(%eax,4), %eax	mov eax, [4*eax + array]
movw _array(%ebx,%eax,4), %cx	mov cx, [ebx + 4*eax + array]

## Etichette & Simboli

Le etichette in GAS sono uguali a quelle degli altri assembleri: il nome dell'etichetta e' seguito dai 2punti. Tutti i nomi dei simboli cominciano con una lettera, un period, o un underscore. I simboli locali sono definiti usando le cifre 0-9 seguite dai 2punti, e ci si riferisce a loro usando quel numero seguito da una b (per un riferimento all'indietro) o da una f (per un riferimento in avanti); nota che questo permette solo 10 simboli locali. Si assegna il valore ad un simbolo usando l'uguale (e.g. 'TRUE = 1') o usando le direttive .set o .equ.

## Direttive

GAS permette la maggior parte delle direttive assembler standard; quelle che seguono sono le piu' utilizzate.

.align

Padra la sezione ad un alignment specifico (e.g. 4 bytes); questa direttiva prende come argomento la dimensione dell'alignment, come anche un argomento opzionale che specifichi il byte da utilizzare le aree di padding (per default e' 00).

.ascii, .asciz, .string

Ognuna di queste direttive prende una o piu' stringhe separate da virgole; con la direttiva .ascii le stringhe non sono terminated, nella .asciz e .string le stringhe sono zero-terminated.

.byte, .double, .int, .word

Ognuna di queste direttive acquisisce come argomento una espressione (per esempio, value1 + value2) e definisce il numero specificato di bytes (byte, int, word, etc) nella locazione corrente al risultato dell'espressione.

.data, .section, .text

La direttiva .section permette alle sezioni o ai segmenti del programma finale di essere definiti per il linker. La direttiva .section acquisisce un nome di sezione, come pure le flags di sezione (b = bss, w = writable, d = data, r = read-only, x = eseguibile per i file COFF; a = allocatable, w = writable, x = executable, @progbits = data, @nobits = no data per file ELF). Le direttive .data e .text sono predefinite per le sezioni di dati e codice.

.equ, .set

Ognuna di queste setta il primo argomento (un simbolo) con il risultato del secondo argomento (una espressione), per esempio `.equ TRUE 1` setta il simbolo `TRUE` al valore `1`.

```
.extern
```

La direttiva tradizionale `EXTERN` e' disponibile ma ignorata; GAS tratta tutti i simboli non definiti come externs.

```
.global, .globl
```

Queste direttive definiscono i simboli global (esportati); ognuna prende per argomento il simbolo da rendere globale.

```
.if /.endif
```

GAS fornisce le usuali direttive `IF...ENDIF` per l'assemblaggio condizionale; la `.if` e' seguita da un'espressione, e tutto il codice tra `.if` e `.endif` e' assemblato solo se quella espressione restituisce non-zero.

```
.include
```

Questa direttiva include un file nella posizione corrente; prende come argomento il nome del file tra virgolette, per esempio `.include "stdio.inc"`.

### Assemblare un Programma

Un programma GAS puo' essere assemblato invocando il GCC con l'opzione `O2` (ottimizza: livello 2). Nota che tutti i programmi GAS devono avere una sezione `.text` e una etichetta globale "main". Ecco un esempio di programma stile 'hello world' in GAS:

```
; gashello.S =====
.text
message:
.ascii "Helloooo, nurse!\0"

.globl main

main:
    pushl $message
    call puts
    popl %eax
    ret
; EOF =====
```

Questo puo' essere compilato con il comando

```
gcc -O2 gashello.S -o ghello
```

o con

```
as gashello.S -o gashello.o
ld -o gashello gashello.o -lc -s -defsym _start=main
```

Nota che e' molto piu' facile usare GCC che AS, dato che devi esplicitare le librerie per linkarlo (da cui il parametro `-lc`) quando chiami LD.

L'Int80 "pid.asm" dall'articolo del mese scorso di Liux sarebbe riscritto cosi' per GAS:

```
;pid.S=====
.global main
.text

szText1:
.asciz "Getting Current Process ID..."

szDone:
.asciz "Done!"
```

```

szError:
.asciz "Error in int 80!"

szOutput:
.string "%d\n"

main:
    pushl $szText1
    call puts
    popl %ecx
    mov $20, %eax
    int $128
    cmp $0, %eax
    je Error
    pushl %eax
    pushl $szOutput
    call printf
    popl %ecx
    popl %ecx
    pushl $szDone
    call puts
    jmp Exit
Error:
    pushl $szError
    call puts
Exit:
    popl %ecx
    ret
; EOF =====

```

Puo' essere compilato alla stessa maniera dell'esempio precedente; nota, pero', la necessita' dei numeri decimali per chiamare gli interrupt (la sintassi 0x?? per specificare un intero esadecimale provoca il mancato riconoscimento dell'opcode da parte dell'assembler)