L'architettura Intel/AMD a 64 bit

G. Lettieri

3 Marzo 2017

1 Registri e istruzioni

I processori Intel/AMD a 64 bit sono una evoluzione dei precedenti processori a 32 bit, evoluzione a loro volta dei precedenti a 16 bit.

Lo *stato* dell'elaboratore è dato dal contenuto dei registri del processore, dal contenuto della memoria e dal contenuto dei registri di I/O. Vediamoli di seguito.

All'interno del processore si trovano 16 registri di uso generale, i cui nomi sono mostrati in Figura 1 insieme al registro `RIP`, che è l'instruction pointer, e al registro `RFLAGS`, che è il registro dei flag. Tutti i registri sono grandi 64 bit. Ci sono anche altri registri di controllo, che vedremo successivamente.

È possibile anche riferire solo alcune parti di ogni registro, utilizzando un nome diverso, secondo la Tabella 1. In particolare, è possibile riferire i 32 bit meno significativi, i 16 bit meno significativi e gli 8 bit meno significativi. Per i soli registri `RAX`, `RBX`, `RCX` ed `RDX` è possibile anche riferire gli 8 bit successivi a `AL`, `BL`, `CL` e `DL`, utilizzando i nomi nell'ultima colonna della Tabella 1. Tuttavia ci sono delle complicate limitazioni sul loro utilizzo, e conviene ignorare l'esistenza di questi ulteriori sotto-registri.

Per quanto riguarda lo spazio di memoria, il processore può potenzialmente riferire 2^{64} byte distinti utilizzando indirizzi di 64 bit. In pratica, però, tutti i processori Intel/AMD ad oggi disponibili limitano i bit realmente utilizzabili a 48, ottenendo uno spazio di memoria grande $2^{48}\,\mathrm{B} = 256\,\mathrm{TiB}$[1], che è comunque molto grande. I 48 bit liberamente utilizzabili sono quelli meno significativi. Gli altri 16 devono essere tutti uguali al bit numero 47. Questo vuol dire che la memoria indirizzabile si presenta come in Figura 2: si possono indirizzare soltanto due porzioni contigue, ciascuna grande 2^{47} byte, una all'inizio e l'altra alla fine dello spazio degli indirizzi. Gli indirizzi che rispettano questa regola sono detti in *forma canonica*. È il processore stesso a generare un errore se si tenta di utilizzare un indirizzo che non è in forma canonica.

Lo spazio di I/O, infine, è costituito da 2^{16} locazioni di un byte (64 KiB).

Le istruzioni riconoscite dal processore prevedono normalmente due operandi di ingresso e sovrascrivono il risultato sul secondo operando. La sintassi

[1] B sta per byte mentre Ki = 1024, Mi = 1024^2, Gi = 1024^3 e Ti = 1024^4.

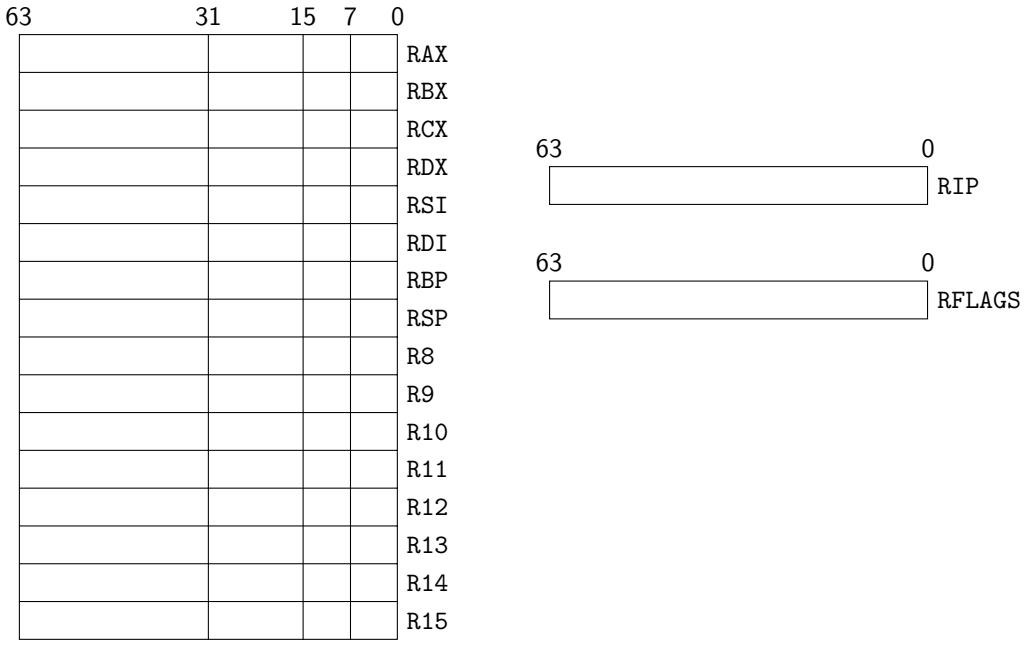

Figura 1: I registri del processore.

64b	32b	16b	8b	8b
RAX	EAX	AX	AL	AH
RBX	EBX	BX	BL	BH
RCX	ECX	CX	CL	CH
RDX	EDX	DX	DL	DH
RDI	EDI	DI	DIB	
RSI	ESI	SI	SIB	
RBP	EBP	BI	BPB	
RSP	ESP	SP	SPB	
R8	R8D	R8W	R8B	
R9	R9D	R9W	R9B	
R10	R10D	R10W	R10B	
R11	R11D	R11W	R11B	
R12	R12D	R12W	R12B	
R13	R13D	R13W	R13B	
R14	R14D	R14W	R14B	
R15	R15D	R15W	R15B	

Tabella 1: Nomi dei sotto-registri.

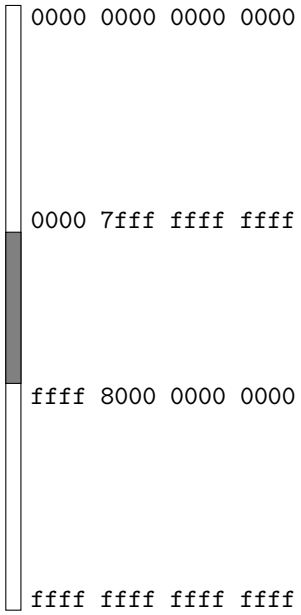

Figura 2: Memoria indirizzabile dal processore.

assembler è la seguente:

codice-operativo primo-operando`,`*secondo-operando*

Sono possibili tre modalità di indirizzamento degli operandi:

- **immediato**: l'operando è una costante contenuta nell'istruzione stessa (al massimo su 32 bit); in assembler l'operando deve essere preceduto dal carattere `$`;
- **registro**: l'operando è contenuto in uno dei registri del processore; in assembler si usa il carattere `%` seguito dal nome del registro;
- **memoria**: l'operando è contenuto in memoria; l'istruzione deve dunque specificarne l'indirizzo.

Il caso di indirizzamento di memoria è il più complicato, in quanto è possibile chiedere al processore di *calcolare* l'indirizzo richiesto (ricordarsi che gli indirizzi sono numeri). Ci sono due modalità principali di calcolo dell'indirizzo, con vari sotto casi:

- La modalità con spiazzamento, base, indice e scala, che usa la seguente sintassi:

  spiazzamento`(`*base*`,`*indice*`,`*scala*`)`

  dove:

  - *spiazzamento* è una costante (al massimo su 32 bit);
  - *base* è il nome di un registro a 64b;
  - *indice* è il nome di un registro a 64b;
  - *scala* può valere 1, 2, 4 oppure 8.

  Il processore calcolerà l'indirizzo sommando lo *spiazzamento* (esteso con segno a 64b), il contenuto del registro *base* e il contenuto del registro *indice* moltiplicato per la *scala*.

  Ci sono vari casi particolari, vediamo i più comuni:

  - se lo spiazzamento è zero, si può omettere;
  - se la scala è 1, si può omettere insieme alla seconda virgola;
  - si può omettere tutta la parte tra parentesi; in questo caso l'indirizzo coincide con lo spiazzamento e l'indirizzamento è detto *diretto*;
  - si possono omettere l'indice e la scala, con le relative virgole; in questo caso l'indirizzo è ottenuto sommando lo spiazzamento (estesto con segno a 64b) e il contenuto del registro base; se si omette anche lo spiazzamento, l'indirizzamento è detto *indiretto*.

- Relativa a `RIP`, con la seguente sintassi:

  spiazzamento`(%RIP)`

  dove *spiazzamento* è una costante (al massimo su 32 bit). Il processore calcolerà l'indirizzo sommando *spiazzamento* (esteso con segno a 64b) al contenuto di `RIP`.

Si noti che il secondo operando non può essere di tipo immediato e che al più un operando può essere di tipo memoria.

Solo l'istruzione `movabs` ammette operandi immediati di 64b e spiazzamenti di 64b, e solo nei seguenti formati:

- `movabs $`*costante*`,%`*registro*: copia la *costante* nel *registro*;
- `movabs` *spiazzamento*`,%RAX`: leggi dall'indirizzo *spiazzamento* in memoria e scrivi in `RAX` (o nei suoi sotto-registri).

Ogni istruzione può lavorare con operandi di 8, 16, 32 o 64 bit. Se uno dei due operandi è un registro, l'assemblatore può dedurre la dimensione degli operandi dal nome del registro, altrimenti è necessario specificarla aggiundendo uno dei seguenti suffissi al codice operativo dell'istruzione:

- `b` per operandi di tipo byte;
- `w` per operandi di tipo "parola" (*word*, due byte).
- `d` per operandi di tipo "doppia parola" (*double word*, quattro byte).
- `q` per operandi di tipo "parola quadrupla" (*quad word*, otto byte).

Si consiglia di aggiungere sempre il suffisso, anche quando non serve.

```
1   .data
2   num1:
3           .quad   0x1122334455667788
4   num2:
5           .quad   0x9900aabbccddeeff
6   risu:
7           .quad   -1
8   .text
9   .globl  _start, start
10  start:
11  _start:
12          movabsq $num1, %rax
13          movq (%rax), %rcx
14          movabsq  $num2, %rax
15          movq (%rax), %rbx
16          addq  %rbx, %rcx
17          movabsq $risu, %rax
18          movq %rcx, (%rax)
19
20          movq  $13, %rbx
21          mov  $1, %rax
22          int   $0x80
```

Figura 3: Un esempio di programma scritto in Assembler GNU per x86_64.

2 Esempio di programma

Prepariamo un semplice esempio di programma da fare eseguire al nostro elaboratore. Dato che ci interessa principalmente sapere cosa succede durante l'esecuzione all'interno dell'elaboratore, usiamo il linguaggio assembler, che è il più vicino al linguaggio macchina: ogni istruzione di assembler si traduce in una istruzione di linguaggio macchina. Sempre per lo stesso motivo, ci conviene pensare che tutta la procedura di preparazione del programma in linguaggio macchina (scrittura del file sorgente, assemblamento, collegamento) sia svolta all'esterno del calcolatore, su un altro calcolatore o in qualunque altro modo, anche se in pratica usiamo lo stesso calcolatore per fare tutto. L'esecuzione del nostro programma comincia dal momento in cui esso è stato caricato in memoria; a quel punto, esiste solo il linguaggio macchina e tutta la procedura precedente non conta più.

La procedura inizia preparando un file di testo che contiene il programma in linguaggio assembler. Si consideri il file di Figura 3. I numeri di riga servono per riferimento e non fanno parte del contenuto del file. L'assemblatore produce una o più sequenze di byte da caricare in memoria, in base alle direttive o alle istruzioni contenute nel file sorgente. Le parole chiave che cominciano per "." sono *direttive* e servono a chiedere all'assemblatore di svolgere vari compiti. Alla

riga 1 troviamo la direttiva `.data` che chiede all'assemblatore di aggiungere alla *sezione data* ciò che segue nel file fino alla prossima direttiva che specifica una nuova sezione (in questo caso, la direttiva `.text` alla riga 8). Le sezioni sono sequenze di byte che possono essere caricate indipendentemente. La sezione "data" verrà caricata in una zona di memoria accessibile sia in lettura che in scrittura, mente la sezione "text" in una zona di sola lettura. Normalmente la sezione data contiene variabili, mentre la sezione text contiene codice, ma niente di tutto ciò è imposto o controllato dall'assemblatore.

Alla riga 2 troviamo la definizione di una *etichetta* (`num1` in questo caso). Le etichette servono a dare un nome all'indirizzo del primo byte che le segue. Abbiamo bisogno delle etichette perché, avendo delegato all'assemblatore e al collegatore il compito di decidere dove caricare le sezioni, non sappiamo gli indirizzi delle entità che definiamo. Tramite le etichette possiamo riferirci a tali indirizzi simbolicamente, e lasciare che siano poi l'assemblatore e il collegatore a sostituirle con i veri indirizzi.

Alla riga 3 troviamo la direttiva `.quad` seguita da un numero. La direttiva chiede all'assemblatore di riservare 8 byte (a partire dal punto della sezione in cui è arrivato) e di inizializzarli con il numero specificato. Possiamo specificare il numero in varie basi (esadecimale, in questo caso): l'assemblatore provvederà a convertire in ogni caso il numero in binario. Più in generale è possibile riservare e inizializzare una sequenza di quad, semplicemente facendo seguire `.quad` da una lista di numeri seperati da virgole. Altre direttive di questo tipo sono `.byte`, per riservare e inizializzare una sequenza di byte, `.long` (per doppie parole) o `.word` (parole).

L'effetto delle righe 2 e 3 è di allocare e inizializzare una variabile che occupa una parola quadrupla e darle il nome `num1`. Le righe 4–5 e 5–6 fanno la stessa cosa con `num2` e `risu`.

Alla riga 8 diciamo all'assemblatore che quanto segue deve essere aggiunto alla sezione "text".

Alla riga 9 troviamo la direttiva `.global` seguita dalle etichette `_start` e `start`. Con questa direttiva stiamo chiedendo di rendere queste etichette visibili al collegatore. Il collegatore cercherà una di queste due per sapere qual è l'indirizzo della prima istruzione del programma (alcuni collegatori cercano `start` e altri `_start`, e per questo e le definiamo entrambe; se ci si limita a Linux è sufficiente `_start`). Il registro `RIP` verrà inizializzato con questo indirizzo quando il programma dovrà essere eseguito.

Alle righe 10 e 11 definiamo le etichette `start` e `_start`. Si noti che non c'è differenza sintattica tra la definizione di `num1`, `num2` e `risu` da una parte e `_start` (o `start`) dall'altra, anche se nelle nostre intenzioni le prime sono variabili mentre `_start` è un'etichetta del programma. Non è casuale, in quanto per l'assemblatore non c'è alcuna differenza tra queste etichette. In ogni caso l'etichetta serve a dare un nome all'indirizzo del byte che la segue. L'assemblatore non segnalerà alcun errore se proviamo a saltare a `num1` o se proviamo a leggere o scrivere all'indirizzo `_start`.

Alle righe 12–22 c'è il programma vero e proprio. Ogni riga contiene una istruzione per il processore. Quello che l'assemblatore farà sarà di tradurre ogni

riga nella corrispondente sequenza di byte di linguaggio macchina, andando così a formare la sezione text. Si noti che questo non è molto diverso da quanto l'assemblatore fa con la direttiva `.quad` (o `.byte`, etc.): si tratta in ogni caso di costruire a poco a poco la sequenza dei byte che compongono la sezione che poi verrà caricata in memoria. Di fatto, niente vieta di usare le direttive `.quad` etc. nella sezione text o di scrivere una istruzione nella sezione data. In ogni caso si ottengono dei byte, e questi non sono di per sé né dati, né istruzioni: è solo nel momento in cui li utilizziamo che vengono interpretati in un modo o in un altro.

Il programma vuole calcolare la somma dei due numeri memorizzati agli indrizzi `num1` e `num2` e scrivere il risultato all'indirizzo `risu`. Per farlo, carica il primo numero nel registro `RCX` (righe 12–13), il secondo numero nel registro `RBX` (righe 14–15), li somma scrivendo il risultato in `RCX` (riga 16), infine copia il risultato in `risu` (righe 17–18). Le righe 20–22 servono a dire al sistema operativo che il programma è terminato e per il momento conviene ignorarle.

Osserviamo la riga 12. L'obiettivo è di caricare l'indirizzo della prima variable in `RAX`, in modo da poter poi caricare il primo numero tramite indirizzamento indiretto (riga 13). Il primo operando è di tipo *immediato* (lo si riconosce dal carattere `$`). Non ci si lasci confondere dall'uso dell'etichetta: l'etichetta sparirà e verrà sostituita dal suo valore numerico (si ricordi che gli indirizzi sono numeri). L'istruzione sta caricando una costante in `RAX`. Questa costante (avendo usato l'etichetta `num1`) non è altro che l'indirizzo del primo numero da sommare. Stiamo usando `movabs` perché, non sapendo dove il programma verrà caricato, non possiamo sapere se tale indirizzo può essere contenuto in soli 32 bit, e `movabs` è l'unica istruzione che accetta operandi immediati a 64 bit.

Si noti che al posto delle istruzioni 12 e 13 avremmo potuto scrivere

```
movabsq num1, %rax
movq %rax, %rcx
```

In questo caso il primo operando della `movabs` è di tipo memoria (lo si riconosce dal fatto che non inizia né con `$`, né con `%`). L'istruzione sta ordinando al processore di eseguire una operazione di lettura in memoria di 8 byte (dal momento che la destinazione è `RAX`) a partire dall'indirizzo `num1` (di nuovo: nel linguaggio macchina finale `num1` sparirà e al suo posto ci sarà il vero indirizzo). Questa istruzione può caricare solo in `RAX`, quindi abbiamo bisogno della successiva per copiare il valore letto in `RCX`, dove lo volevamo.

Stesse considerazioni valgono per le righe 14–15 e, nella direzione opposta, per le righe 17–18.

Supponiamo che questo file si chiami `sum.s`. Per assemblarlo lanciamo il comando

```
as sum.s
```

Se non ci sono errori di sintassi, l'assemblatore non stampa niente e produce il file `a.out` (assembler output). È possibile cambiare il nome del file prodotto usando l'opzione `-o` seguita dal nome desiderato. Conviene farlo, visto che anche il collegatore usa il nome `a.out` come default:

```
sum-example/sum.o:     formato del file elf64-x86-64


Disassemblamento della sezione .text:

0000000000000000 <_start>:
   0: 48 b8 00 00 00 00 00  movabs $0x0,%rax
   7: 00 00 00
   a: 48 8b 08              mov    (%rax),%rcx
   d: 48 b8 00 00 00 00 00  movabs $0x0,%rax
  14: 00 00 00
  17: 48 8b 18              mov    (%rax),%rbx
  1a: 48 01 d9              add    %rbx,%rcx
  1d: 48 b8 00 00 00 00 00  movabs $0x0,%rax
  24: 00 00 00
  27: 48 89 08              mov    %rcx,(%rax)
  2a: 48 c7 c3 0d 00 00 00  mov    $0xd,%rbx
  31: 48 c7 c0 01 00 00 00  mov    $0x1,%rax
  38: cd 80                 int    $0x80
```

Figura 4: Output del comando `objdump -d sum`.

```
as sum.s -o sum.o -g
```

(Abbiamo aggiunto anche l'opzion `-g`, che include nel file le informazioni utilizzate dal debugger.)

Tale file deve essere poi passato al collegatore per produrre l'eseguibile:

```
ld sum.o -o sum -g
```

(Anche qui abbiamo usato l'opzione `-o` per specificare il nome dell'eseguibile e l'opzione `-g` per il debugger.)

Possiamo esaminare il prodotto dell'assemblatore e del collegatore con il comando `objdump`. Per esempio, possiamo chiedere di vedere il codice macchina prodotto dall'assemblatore nella sezione text:

```
objdump -d sum.o
```

In questo caso dovremmo ottenere un output simile a quello di Figura 4. Le righe a partire da quella che inizia con "`0:`" mostrano il contenuto della sezione text. Ogni riga mostra un offset all'interno della sezione, seguita da una sequenza di byte che si trovano a partire da quell'offset (tutti i numeri sono in esadecimale). Sull'estrema destra tali byte sono interpretati come istruzioni di assembler. Si noti che questa interpretazione, che è l'operazione inversa rispetto a quanto fa l'assemblatore, è fatta da `objdump` senza guardare il file sorgente. Si noti come `num1`, `num2` e `risu` sono state sostituite con zero. Questo perché neanche

```
sum-example/sum:     formato del file elf64-x86-64


Disassemblamento della sezione .text:

00000000004000b0 <_start>:
  4000b0: 48 b8 ea 00 60 00 00  movabs $0x6000ea,%rax
  4000b7: 00 00 00
  4000ba: 48 8b 08              mov    (%rax),%rcx
  4000bd: 48 b8 f2 00 60 00 00  movabs $0x6000f2,%rax
  4000c4: 00 00 00
  4000c7: 48 8b 18              mov    (%rax),%rbx
  4000ca: 48 01 d9              add    %rbx,%rcx
  4000cd: 48 b8 fa 00 60 00 00  movabs $0x6000fa,%rax
  4000d4: 00 00 00
  4000d7: 48 89 08              mov    %rcx,(%rax)
  4000da: 48 c7 c3 0d 00 00 00  mov    $0xd,%rbx
  4000e1: 48 c7 c0 01 00 00 00  mov    $0x1,%rax
  4000e8: cd 80                 int    $0x80
```

Figura 5: Output del comando `objdump -d sum`.

l'assemblatore sa quanto valgono, in quanto è solo il collegatore che decide dove le varie sezioni dovranno essere caricate.

Per vedere il risultato prodotto dal collegatore (sempre nella sezione text), scriviamo

```
objdump -d sum
```

L'output è mostrato in Figura 5 e si interpreta in modo simile a quello di Figura 4. In questo caso, però, il numero all'inizio di ogni riga rappresenta l'*indirizzo* (scelto dal collegatore) a partire dal quale verranno caricati i byte mostrati in quella riga. Qui vediamo che `num1` etc. hanno assunto il loro valore finale. Si noti che il nostro programma consiste dei byte che si trovano nella parte centrale dell'output: questo è tutto quello che il processore vedrà quando il programma sarà caricato e messo in esecuzione.

Il programma può essere infine caricato ed eseguito scrivendo

```
./sum
```

In questo caso il programma non produce alcun output, perché ci siamo limitati a scrivere il risultato in memoria, senza inviarlo ad alcuna periferica di I/O. Possiamo però vederlo in funzione utilizzando il debugger. Scriviamo

```
gdb sum
```

`gdb` è un debugger a riga di comando, per il quale esistono varie interfacce grafiche (per esempio, `ddd`). Possiede una semplice interfaccia semi-grafica incorporata, che si può far partire premendo prima i tasti "Ctrl+x" e poi il tasto "a". Per mettere in funzione il nostro programma settiamo prima un breakpoint all'etichetta `_start` scrivendo `b _start` (invio), e poi facciamo partire l'esecuzione scrivendo `r` (e poi invio). Possiamo far avanzare il programma di una istruzione alla volta premendo `n` (invio). Per esaminare il contenuto dei registri possiamo cambiare il layout dell'interfaccia con il comando `layout reg` (invio). In qualunque momento possiamo terminare il debugger scrivendo `q` (invio). `gdb` ha molti comandi, si consiglia di consultare qualche guida in rete o l'help incorporato (comando `help`).