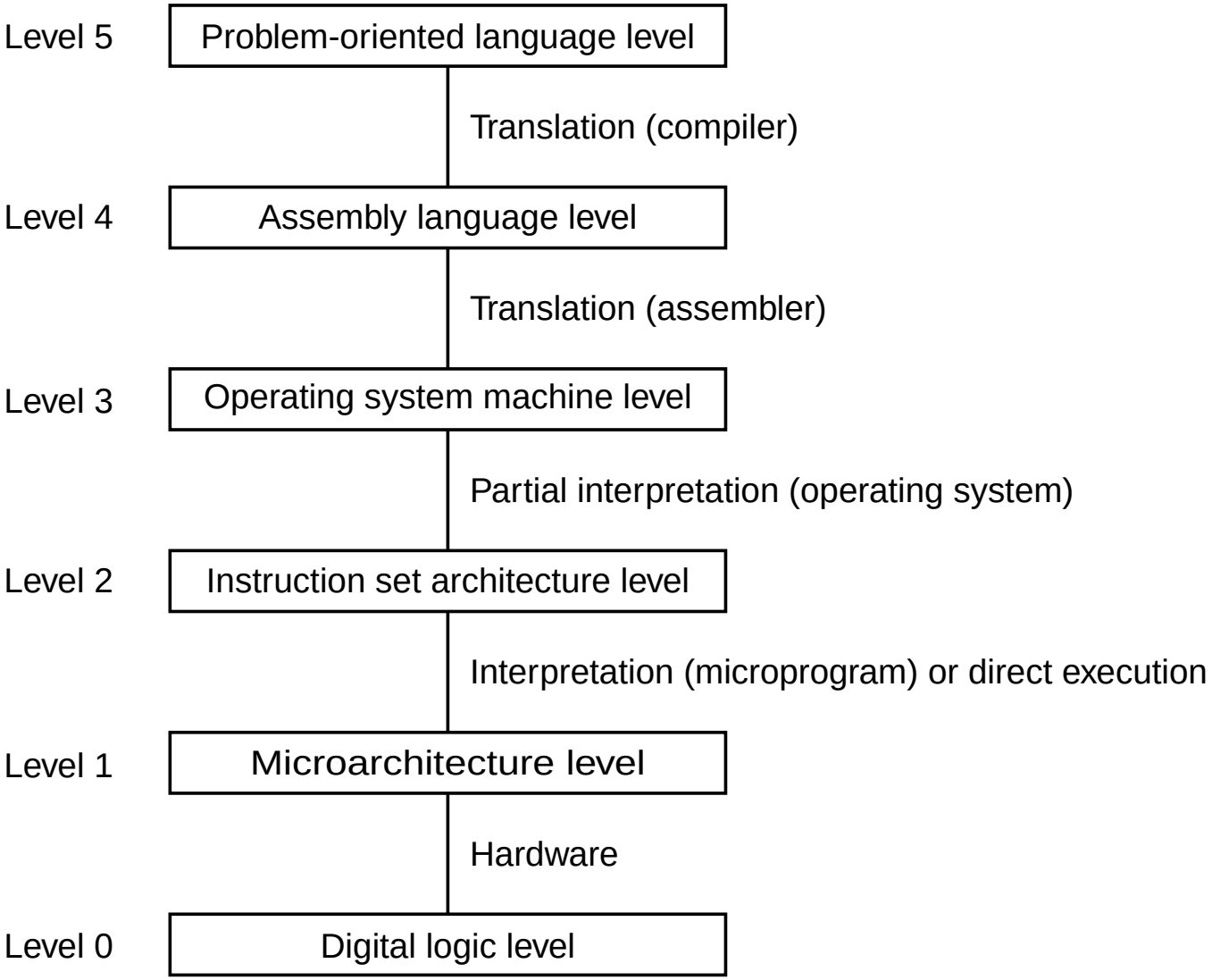
Livello 4: linguaggio assembly



Compilazione, interpretazione, traduzione

Scrivere direttamente le istruzioni macchina, cioé programmare in linguaggio macchina, è quasi impossibile. Per questo motivo sono stati realizzati compilatori, interpreti e traduttori.

Il linguaggio di programmazione più prossimo al linguaggio macchina è assembly: le sue istruzioni sono quasi direttamente mappate in istruzioni macchina (traduzione).

Il programmatore assembly conosce esattamente il funzionamento della CPU. Per questo motivo sa adoperare efficacemente le sue risorse.

Assembly: vantaggi

L'esatto controllo delle risorse di calcolo e di memoria attraverso assembly può determinare enormi guadagni di efficienza allorquando un programma realizza delle funzioni di basso livello. Es. (driver): controllare una stampante. Es. (kernel): organizzare la memoria di massa.

Anche in un programma che realizza procedure di alto livello (es.: un algoritmo matematico), riscrivere in assembly poche righe strategiche (es.: 5% del codice) può portare a esecuzioni molto più efficienti (es.: velocità 5 volte maggiore).

Assembly: svantaggi

Un programma assembly non è portabile, essendo tradotto solo per la famiglia di processori in grado di eseguire le istruzioni macchina collegate.

È disagevole scrivere il codice, in quanto le istruzioni assembly sono elementari e conducono a programmi lunghi.

È facile scrivere codice semanticamente errato: le singole istruzioni infatti sono quasi sempre corrette, ma il programma risultante è poco strutturato.

I compilatori sono sempre più competitivi: difficile fare meglio programmando in assembly, soprattutto se si vuole sfruttare la superscalarità.

Dipendenza di assembly dalla CPU

Pur sintatticamente e strutturalmente simili, i linguaggy assembly ereditano le differenze tra processori.

L'assembly Intel è difficile da usare per problemi di legacy (compatibilità con linguaggi precedenti). Le istruzioni si sono stratificate nell'arco degli anni a partire dal progenitore assembly per il chip 8088, superato e diverso dai linguaggi attuali.

ARM (Acorn RISC Machine), possiede poche, semplici e lineari istruzioni macchina. L'assembly per ARM riflette questa pulizia.

La famiglia ARM

Architettura a registri a 32 bit fino ad ARM 7, registri a 64 bit da ARM 8.

Le istruzioni macchina nel tempo si sono arricchite di funzionalità sofisticate: divisione, operazioni vettoriali, esecuzione di bytecode da codice JAVA.

- Produzione: $\approx 10^{10}$ pezzi l'anno
- architettura a basso consumo
- a bordo di moltissimi smartphone e tablet
- scelta ideale per architetture embedded: smart TV, player, router, modem, eccetera.

Il modello hardware in ARM

Assembly per ARM accede a una semplice CPU astratta che normalmente include:

- una ALU
- un set uniforme di registri in cui memorizzare dati e indirizzi
- una rappresentazione uniforme della memoria principale
- alcuni registri con funzioni speciali
- un sistema di controllo dell'esecuzione.

Le funzionalità avanzate sono nascoste ad assembly: superscalarità, pipelining, registri ombra, suggerimenti per il predittore, caching, memoria estesa, periferiche.

Il modello di memoria in ARM

Assembly per ARM opera su un modello di memoria che include

- una memoria principale di 2^{32} locazioni ciascuna di un byte (totale: 4 GB). Ogni locazione quindi è individuata da un indirizzo di 32 bit.
- 16 registri ciascuno di 32 bit, genericamente etichettati `r0, ..., r15`. I registri `r13, r14, r15` sono specializzati e possono essere indicati rispettivamente come `sp` (stack pointer), `lr` (link register), `pc` (program counter), con funzioni di gestione della memoria dati (`sp` e `lr`) e accesso alla memoria di programma (`pc`) che approfondiremo.

Funzioni speciali in ARM

Assembly può accedere al registro speciale `cpsr` (Current Program Status Register), il valore dei cui bit segnala particolari informazioni correnti sul programma in esecuzione, definendone lo stato.

Infine, come per ogni programma utente, anche assembly ha la possibilità di far eseguire chiamate di sistema attraverso le quali il programma può

- interagire con le periferiche (schermo, tastiera, memoria di massa, . . .)
- allocare e rilasciare memoria dinamicamente
- conoscere il valore del tempo di CPU
- terminare l'esecuzione, restituendo il controllo della CPU al sistema operativo.

Il simulatore ArmSim

Il test di un programma assembly scritto direttamente nella CPU ha senso solo in fase avanzata di progettazione. La gran parte del progetto software viceversa è sviluppata all'interno di un simulatore.

Lasciando da parte simulatori professionali completi ma complessi da usare, scegliamo ArmSim che è un ottimo simulatore della CPU ARM a 32 bit per scopi didattici.

ArmSim implementa il manuale completo delle istruzioni ARM, che non si può portare all'esame. Invece, si potrà portare all'esame la tabella sintetica delle istruzioni (ARM Reference Chart) e la tabella delle chiamate di sistema (SWI I/O operations).

Simulazione di un programma in ArmSim

ArmSim

- controlla la correttezza sintattica del codice
- assembla il programma
- traduce le istruzioni
- simula il caricamento in memoria del codice macchina
- avvia virtualmente l'esecuzione, anche in modalità passo-passo
- simula eventuali chiamate al sistema operativo
- permette all'utente di accedere all'immagine della CPU e della memoria durante la simulazione dell'esecuzione.

Somma e sottrazione di interi

Richiedono tre argomenti:

- `add r0, r2, r3` scrive sul registro `r0` la somma dei contenuti di `r2` e `r3`
- `add r0, r2, #7` pone in `r0` la somma di `r2` e `7`
- `add r0, r2, #0xF` in `r0` la somma di `r2` e `15`
- `sub r0, r2, r3` subtract `r0 = r2 - r3`
- `rsb r0, r2, r3` reverse subtract `r0 = r3 - r2`.

ARM rappresenta gli interi in complemento a due.

Solo il terzo argomento può essere un numero (una costante).

N.B.: assembly tradizionalmente non distingue il maiuscolo dal minuscolo (es.: `R0` = `r0`).

Moltiplicazione e riporto di interi

- `mul r0, r2, r3` multiply `r0 = r2 * r3`
  `mul` non ammette argomento costante, e fino ad ARM 5 accettava solo tre argomenti tutti diversi.
- `adc r0, r2, r3` add with carry
  `r0 = r2 + r3 + c`, somma il bit di riporto `c` (carry) presente in `cpsr`, permettendo somme su 64 bit
- `sbc r0, r2, r3` subtract with carry
  `r0 = r2 - r3 + c - 1`, somma il bit di carry permettendo sottrazioni su 64 bit
- `rsc r0, r2, r3` reverse subtract with carry
  `r0 = r3 - r2 + c - 1`.

Esercizi:

Perchè la subtract with carry sottrae 1 al risultato?

Scrivere del codice ARM efficiente che carichi in `r1` il valore delle espressioni:

- `r1 = r1 + r2 + r3`

Esercizi:

Perchè la subtract with carry sottrae 1 al risultato?

Scrivere del codice ARM efficiente che carichi in `r1` il valore delle espressioni:

- `r1 = r1 + r2 + r3`
- `r1 = r1 - r2 - 3`

Esercizi:

Perchè la subtract with carry sottrae 1 al risultato?

Scrivere del codice ARM efficiente che carichi in `r1` il valore delle espressioni:

- `r1 = r1 + r2 + r3`
- `r1 = r1 - r2 - 3`
- `r1 = 4 × r2`

Esercizi:

Perchè la subtract with carry sottrae 1 al risultato?

Scrivere del codice ARM efficiente che carichi in `r1` il valore delle espressioni:

- `r1 = r1 + r2 + r3`
- `r1 = r1 - r2 - 3`
- `r1 = 4 × r2`
- `r1 = - r2`

Esercizi:

Perchè la subtract with carry sottrae 1 al risultato?

Scrivere del codice ARM efficiente che carichi in `r1` il valore delle espressioni:

- `r1 = r1 + r2 + r3`
- `r1 = r1 - r2 - 3`
- `r1 = 4 × r2`
- `r1 = - r2`
- `r0 r1 = r2 r3 + r4 r5` (somma a 64 bit)
- `r0 r1 = r2 r3 - r4 r5` (sottrazione a 64 bit).

Moltiplicazione estesa

`umull r0, r1, r2, r4` unsigned multiplication long. Calcola il prodotto di `r2` e `r4` interi senza segno; deposita il risultato a 64 bit nei registri `r1` (cifre più significative) e `r0` (cifre meno significative).

`smull r0, r1, r2, r4` signed multiplication long: Calcola il prodotto di `r2` e `r4` interi in complemento a due; deposita il risultato a 64 bit nei registri `r1` e `r0`.

Non esistono istruzioni per la divisione di interi. L'aritmetica floating-point è disponibile attivando la floating point unit (FPU), che esegue anche calcolo vettoriale (VFP, estensione NEON).

Operazioni logiche e copia

L'operatore è applicato bit a bit all'intero registro.
L'ordine e uso degli argomenti è quello già noto dalle operazioni aritmetiche:

- `and r0, r2, r3` `r0` = `r2` AND `r3`
- `and r0, r2, #5` `r0` = `r2` AND 5
- `orr r0, r2, r3` `r0` = `r2` OR `r3`
- `eor r0, r2, #5` `r0` = `r2` XOR 5
- `bic r0, r2, r3` `r0` = `r2` AND NOT(`r3`).

Copiare registri (operazione move):

- `mov r0, r2` `r0` = `r2` (N.B.: `r2` è copiato in `r0`)
- `mvn r0, r2` `r0` = NOT(`r2`) (move negate).

Esercizio: calcolare il resto della divisione per 16.

Costanti rappresentabili

Assembly accetta costanti decimali (#5), binarie (#0b1101), ottali (#0h777), esadecimali (#0xFDE7E).

Le istruzioni ARM riservano 12 bit per le costanti intere (rappresentazione immediate_8r):

- 8 bit per la mantissa
- 4 bit per l'esponente (positivo) moltiplicato due.

Es.: `0xFF, 255, 256, 0xCC00, 0x1FC00` sono costanti ammesse come argomenti di un'istruzione.
Es.: `0x101, 257, 0x102, 258` non lo sono.

L'assemblatore se possibile sostituisce una costante inammissibile con un'istruzione in grado di rappresentarla. Altrimenti genera un errore di traduzione.

Esercizi

Scrivere serie di istruzioni che realizzano le seguenti espressioni:

- `r1 = 57`

Esercizi

Scrivere serie di istruzioni che realizzano le seguenti espressioni:

- `r1 = 57`
- `r1 = 1024`

Esercizi

Scrivere serie di istruzioni che realizzano le seguenti espressioni:

- `r1 = 57`
- `r1 = 1024`
- `r1 = 257`

Esercizi

Scrivere serie di istruzioni che realizzano le seguenti espressioni:

- `r1 = 57`
- `r1 = 1024`
- `r1 = 257`
- `r1 = 0xAABBCCDD`

Esercizi

Scrivere serie di istruzioni che realizzano le seguenti espressioni:

- `r1 = 57`
- `r1 = 1024`
- `r1 = 257`
- `r1 = 0xAABBCCDD`
- `r1 = -1.`

Traslazione e rotazione di argomenti

In ogni istruzione aritmetica e logica, se l'ultimo argomento è un registro il suo contenuto può essere contestualmente traslato (shift) o ruotato (rotate) di N = 0,. . . , 31 bit.

Es.: l'argomento `r2, lsl #2` è la traslazione in `r2` di due posizioni verso sinistra. Quindi,

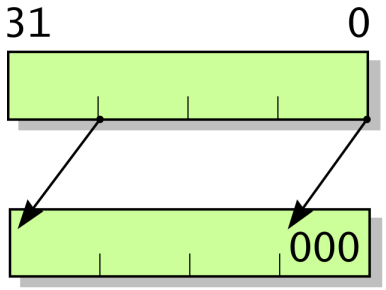
`add r0, r1, r2, lsl #2` esegue `r0` = `r1` + `r2` $\ll$ 2.

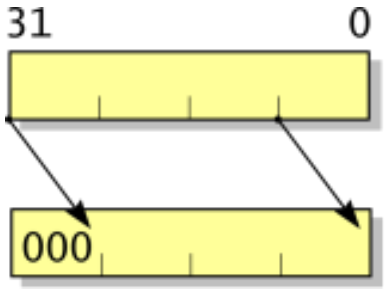
Es.: l'argomento `r2, lsl r3` è la traslazione in `r2` di un numero di posizioni verso sinistra specificate dagli 8 bit meno significativi di `r3`. Quindi,

`mov r0, r2, lsl r3` esegue `r0` = `r2` $\ll$ `r3`.

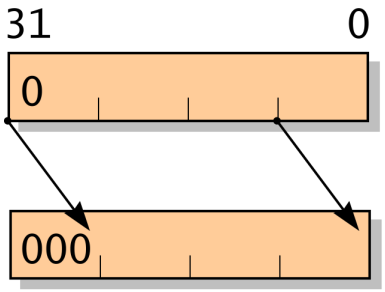
Traslazioni possibili

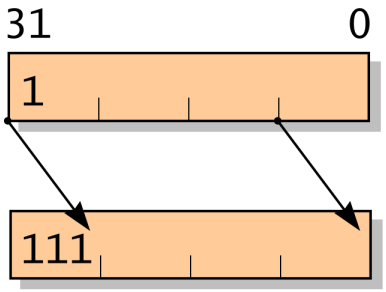
- `lsl` logical shift left





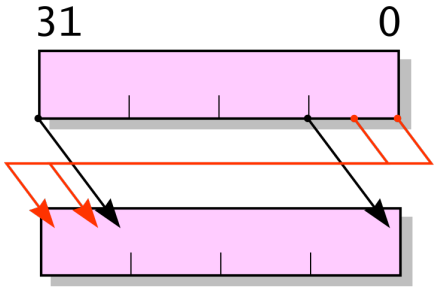
- `lsr` logical shift right
- `asr` arithmetic shift right, mantiene il segno del numero. Equivale a una divisione per 2^N.



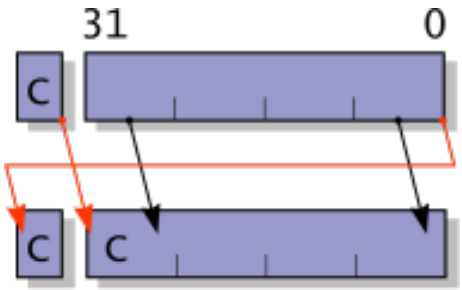


Rotazioni possibili

- `ror` rotate right: gli N bit uscenti a destra rientrano da sinistra



- `rrx` rotate right extended, rotazione a destra di un bit includendo il bit di carry; non ammette argomento.



Esempi di traslazioni

Es.: l'esecuzione di

```
mov r0,#1
mov r1,#2
add r2,r0,r0,lsl r1
```

assegna a `r2` il valore 5. Infatti il terzo argomento `r0,lsl r1` è una traslazione di `r0`, tuttavia il contenuto di `r0` non viene toccato.

Scrivere sequenze compatte di istruzioni assembly che calcolano le seguenti espressioni:

- `r1 = 8 * r2`

Esempi di traslazioni

Es.: l'esecuzione di

```
mov r0,#1
mov r1,#2
add r2,r0,r0,lsl r1
```

assegna a `r2` il valore 5. Infatti il terzo argomento `r0,lsl r1` è una traslazione di `r0`, tuttavia il contenuto di `r0` non viene toccato.

Scrivere sequenze compatte di istruzioni assembly che calcolano le seguenti espressioni:

- `r1 = 8 * r2`
- `r1 = r2 / 4`

Esempi di traslazioni

Es.: l'esecuzione di

```
mov r0,#1
mov r1,#2
add r2,r0,r0,lsl r1
```

assegna a `r2` il valore 5. Infatti il terzo argomento `r0,lsl r1` è una traslazione di `r0`, tuttavia il contenuto di `r0` non viene toccato.

Scrivere sequenze compatte di istruzioni assembly che calcolano le seguenti espressioni:

- `r1 = 8 * r2`
- `r1 = r2 / 4`
- `r1 = 5 * r2`

Esempi di traslazioni

Es.: l'esecuzione di

```
mov r0,#1
mov r1,#2
add r2,r0,r0,lsl r1
```

assegna a `r2` il valore 5. Infatti il terzo argomento `r0,lsl r1` è una traslazione di `r0`, tuttavia il contenuto di `r0` non viene toccato.

Scrivere sequenze compatte di istruzioni assembly che calcolano le seguenti espressioni:

- `r1 = 8 * r2`
- `r1 = r2 / 4`
- `r1 = 5 * r2`
- `r1 = 3/4 * r2`.

Accesso alla memoria principale

Trasferimenti tra registri e memoria principale: lettura (load, `ldr`) e scrittura (store, `str`) della memoria.

- `ldr r3, [r0]`
  copia nel registro `r3` un word (parola di 4 byte) a partire dall'indirizzo contenuto in `r0`
- `ldr r3, [r0, #8]` offset by constant
  copia nel registro `r3` un word a partire dall'indirizzo in `r0` + 8 byte
- `ldr r3, [r0, r1,lsl #3 ]` offset by shifted variable
  copia nel registro `r3` un word a partire dall'indirizzo in `r0` + offset specificato in `r1` traslato di 3 bit.

Aggiornamento degli indirizzi

Le modalità di accesso alla memoria appena viste possono essere declinate secondo tre possibilità di aggiornamento dei registri indirizzo:

- `ldr r3, [r0, #±8]` nessun aggiornamento
- `ldr r3, [r0, #±8]!` pre-aggiornamento
- `ldr r3, [r0], #±8` post-aggiornamento

Es.:

- `ldr r3, [r1, -r0]!`
  `r1 = r1 - r0; r3 = [r1]` (`r1` pre-aggiornato)
- `ldr r3, [r0], #8`
  `r3 = [r0]; r0 = r0 + 8` (`r0` post-aggiornato)
- `ldr r3, [r1, r0, lsl #2]`
  `r3 = [r1 + 4 * r0]` (`r1` non modificato).

Allineamento della memoria

Tutte le modalità di indirizzamento di `ldr` restano valide anche per `str`. Es.: `str r0, [r4,#-8]`.

L'accesso tramite `ldr` e `str` avviene solo su parole allineate. L'indirizzo del primo byte dev'essere un multiplo di 4, altrimenti viene generato un errore di traduzione.

I processori ARM normalmente seguono la convenzione little-endian. Possono tuttavia essere inizializzati per funzionare anche in modalità big-endian: detti per questo anche bi-endian.

Array (vettori)

L'array è una struttura dati formata da una sequenza indicizzata di elementi dello stesso tipo.

La gestione degli indirizzi in assembly favorisce l'organizzazione di un array come una sequenza di word allocati consecutivamente nella memoria principale. In tal modo, per accedere all'elemento di indice i dell'array bisogna:

- conoscere l'indirizzo base (quello del primo elemento) dell'array
- accedere (in lettura o scrittura) alla locazione di memoria di indirizzo
  indirizzo base + offset =
  indirizzo base + $i \cdot$ dimensione word.

Array: esempio

Scrivere in `r1` la somma dei primi tre elementi del vettore (di interi) con indirizzo base `r0`

Array: esempio

Scrivere in `r1` la somma dei primi tre elementi del vettore (di interi) con indirizzo base `r0`

```
ldr r1, [r0]
ldr r2, [r0, #4]
add r1, r1, r2
ldr r2, [r0, #8]
add r1, r1, r2
```

Array: esempio

Scrivere in `r1` il valore `a[i] + a[i+1] + a[i+2]` del vettore `a`, avente indirizzo base contenuto in `r0` e indice `i` in `r2`

Array: esempio

Scrivere in `r1` il valore `a[i] + a[i+1] + a[i+2]` del vettore `a`, avente indirizzo base contenuto in `r0` e indice `i` in `r2`

```
add r3, r0, r2, lsl #2
ldr r1, [r3]
ldr r4, [r3,#4]
add r1, r1, r4
ldr r4, [r3,#8]
add r1, r1, r4
```

Accesso a porzioni di parola

ARM permette anche di accedere a parole di una locazione (byte) o di due locazioni (half-word). Ciò permette di definire array con elementi di altro tipo:

- `ldrb` load register byte
  `ldrsb` load register signed byte (aritmetico)
  `strb` store register byte
- `ldrh` load register half word
  `ldrsh` load register signed half word
  `strh` store half word.

Anche gli half-word devono essere allineati: il loro indirizzo è accettato solo se è multiplo di 2.

Struttura del programma

Un programma assembly corrisponde a un file testo (`file.s`). Oltre a istruzioni e costanti contiene anche:

- etichette, che identificano l'indirizzo dell'istruzione che precedono se seguite dal simbolo : . Es.: `label_1:  mov r0,r1` . Il programmatore può quindi adoperare l'etichetta per riferirsi all'indirizzo dell'istruzione in questione, lasciando all'assemblatore l'onere di risolvere l'associazione etichetta/indirizzo
- direttive all'assemblatore, che forniscono indicazioni su come comportarsi da quel punto del programma. Le direttive sono precedute dal simbolo . . Es.: `.data` , `.globl` .

Direttive di uso comune

Alcune direttive compaiono in quasi tutti i programmi assembly:

- `.text` il testo che segue sono le istruzioni che costituiscono programma
- `.data` il testo che segue sono costanti da inserire in memoria nella regione dati
- `.globl` rende l'etichetta che segue visibile a moduli software esterni a quello che la contiene (adoperata in progetti software in cui più moduli sono assemblati assieme in un unico programma. Non affrontiamo il caso)
- `.end` specifica la fine del modulo.

Direttive sui tipi di dato

Attraverso le direttive è possibile specificare diversi tipi di dato, eventualmente anche etichettabili:

- `.word 34, -46, 0xAABBCCDD, 0b-1001100`
  ogni dato è allineato su 4 locazioni (di un byte)
- `.byte 45, 0x3a`
  ogni dato occupa una locazione
- `.ascii "del testo tra virgolette "`
  ogni carattere codificato ASCII in una locazione
- `.asciiz "altro esempio"`
  ogni carattere codificato ASCII in una locazione e si accoda una locazione zero (fine stringa)
- `.skip 64`
  allocazione di 64 locazioni inizializzate a zero.

Esempio di programma

```
.data
primes: .word 2, 3, 5, 7, 11
string: .asciiz  "alcuni numeri primi"
.text
main: ldr  r0, =primes    @ pseudo-istruzione
      ldr  r1, [r0]
      ldr  r2, [r0, #4]   @ leggo numero 3
      ldr  r0, =string    @ pseudo-istruzione
      ldrb r3, [r0]
      ldrb r4, [r0, #3]   @ leggo carattere u
      swi 0x11          @ comando di terminazione
.end
```

Pseudo-istruzioni e commenti inline

`ldr r0, =primes` è una pseudo-istruzione. Carica in `r0` l'indirizzo di memoria etichettato con `primes:`

Una pseudo-istruzione non fa parte del linguaggio macchina. É tradotta in una o più istruzioni macchina. L'uso di pseudo-istruzioni semplifica la programmazione senza astrarre dal livello assembly.

Un commento in riga (inline) è preceduto dal simbolo `;` o `@` .

Controllo di flusso in assembly

Nei linguaggi ad alto livello il flusso dell'esecuzione è controllato dai costrutti

- condizione (`if-then-else`)
- iterazione condizionata (`while, repeat`)
- iterazione incondizionata (`for`).

I costrutti di alto livello sono compilati nell'elementare costrutto base del salto, eventualmente eseguito sotto condizione.

Assembly per ARM fornisce

- l'istruzione di salto relativo di `#const` word (branch): `b #const`
- la possibilità di condizionare l'esecuzione delle istruzioni.

Istruzioni condizionate

A ogni istruzione può essere aggiunto un suffisso di due lettere, che specifica la condizione che dev'essere verificata affinchè l'istruzione sia eseguita.

La condizione dipende dal valore del registro di stato `cprs`.

Inoltre, alcune istruzioni modificano i bit nel registro di stato se a esse è aggiunto il suffisso `s`.

Es.: `add` `addne` `adds` `addnes` .

Esempio di esecuzione condizionata

Es.:

```
subs r0, r1, r2
addeq r2, r2, #1
beq  label
```

L'istruzione `subs` modifica il registro `cprs`: se il risultato della sottrazione è 0 allora il bit `Z` (zero) di `cprs` è posto a 1.

Le istruzioni `addeq r2, r2, #1` e `beq label` sono eseguite se il bit `Z` di `cprs` vale 1.

N.B.: quasi sempre l'argomento dell'istruzione `b` è un'etichetta!

Condizioni in ARM

Le condizioni dipendono da 4 bit (flag) del registro `cprs`:

- Z zero
- C carry
- N negative
- V overflow .

Suffix	Description	Flags
eq	Equal / equals zero	Z
ne	Not equal	!Z
cs / hs	Carry set / uns. higher or same	C
cc / lo	Carry clear / unsigned lower	!C

Elenco condizioni

Suffix	Description	Flags
mi	Minus / negative	N
pl	Plus / positive or zero	!N
vs	Overflow	V
vc	No overflow	!V
hi	Unsigned higher	C and !Z
ls	Unsigned lower or same	!C or Z
ge	Signed greater than or equal	N == V
lt	Signed less than	N != V
gt	Signed greater than	!Z and (N == V)
le	Signed less than or equal	Z or (N != V)
al	Always (default)	any

Istruzioni di confronto

Modificano solo i flag del registro di stato:

- compare `cmp r0, r1` , confronta `r0` con `r1`, aggiorna i flag come `subs r r0 r1`
- compare negated `cmn r0, r1` , confronta `r0` con `-r1`, aggiorna i flag come `adds r r0, r1`
- test `tst r0, r1` , aggiorna i flag come `ands r r0 r1`
- test equal `teq r0 r1` , aggiorna i flag come `eors r r0 r1`.

Es.:

```
cmp r2, #7
ble label
```

salta a label solo se `r2` è minore o uguale a `7`.

Es.: alternativa tra due istruzioni

Es.: `if (i == j) then i = i + 1 else i = j fi`

con `i` $\Rightarrow$ `r1` , `j` $\Rightarrow$ `r2` :

Es.: alternativa tra due istruzioni

Es.: `if (i == j) then i = i + 1 else i = j fi`

con `i` $\Rightarrow$ `r1` , `j` $\Rightarrow$ `r2` :

```
        cmp r1, r2
        beq then
        mov r1, r2
        b fine
then:   add r1, r1, #1
fine:
```

.

Es.: alternativa tra due istruzioni

Es.: `if (i == j) then i = i + 1 else i = j fi`

con `i` $\Rightarrow$ `r1` , `j` $\Rightarrow$ `r2` :

```
        cmp r1, r2
        beq then
        mov r1, r2
        b fine
then:   add r1, r1, #1
fine:
```

Alternativa senza salti, più efficiente ed elegante:

```
cmp r1, r2
addeq r1, r1, #1
movne r1, r2.
```

Costrutto `if-then-else` generale

```
if Bool then Com1 else Com2 fi
```

viene tradotta in:

```
        Eval Bool
        b_cond then
        Com2
        b fine
then:   Com1
fine:
```

Costrutto `if-then` generale

```
if Bool then Com fi
```

viene tradotta in:

```
        Eval not Bool
        b_cond fine
        Com
  fine:
```

Es.: `while` di due istruzioni

Es.: `while (i != 0) do i = i - 1, j = j + 1 od`

con `i` $\Rightarrow$ `r1` , `j` $\Rightarrow$ `r2` :

Es.: `while` di due istruzioni

Es.: `while (i != 0) do i = i - 1, j = j + 1 od`

con `i` $\Rightarrow$ `r1` , `j` $\Rightarrow$ `r2` :

```
while:  cmp r1, #0
        beq fine
        sub r1, r1, #1
        add r2, r2, #1
        b while
fine:   instr..
```

Es.: `while` di due istruzioni

Es.: `while (i != 0) do i = i - 1, j = j + 1 od`

con `i` $\Rightarrow$ `r1` , `j` $\Rightarrow$ `r2` :

```
while:  cmp r1, #0
        beq fine
        sub r1, r1, #1
        add r2, r2, #1
        b while
fine:   instr..
```

Alternativa senza salti, più efficiente ed elegante:

```
while:  cmp r1, #0
        subne r1, r1, #1
        addne r2, r2, #1
        bne while
        instr..
```

Costrutto `while` generale

```
while Bool do Com od
```

viene tradotta secondo lo schema:

```
  while:  Eval not Bool
          b_cond fine
          Com
          b while
  fine:
```

Assembly e linguaggio macchina

La traduzione da un'istruzione assembly nell'istruzione macchina a 32 bit corrispondente è immediata.

Es.: l'istruzione `subs r1, r2, r3` corrisponde alla seguente sequenza di bit

`cond`	`opcode`	`S`	`rn`	`rd`	`shift`	`2arg`
`al`	`sub`	`t`	`r2`	`r1`	`lsl 0`	`r3`
`1110`	`0000010`	`1`	`0010`	`0001`	`00000000`	`0011`

Es.: l'istruzione di branch `b` riserva 24 bit per specificare il salto relativo; a `r15` (program counter) viene sommato un numero intero codificato 23 bit con segno. L'intervallo di indirizzi raggiungibili quindi è di $\pm 2^{23} \cdot 4 = \pm 32$ MB.

Esercizi

- Calcolare e scrivere nel registro `r1` l'n-esimo numero di Fibonacci $F(n)$, con n contenuto nel registro `r0`. Allo scopo si ricordi che è
  $F(0) = 0$, $F(1) = 1$,
  $F(n) = F(n-1) + F(n-2)$, $n > 1$.
- Calcolare e scrivere nel registro `r1` la somma degli elementi di un array V di 10 elementi, avente indirizzo base contenuto in `r0`.

Funzioni, procedure, subroutine

Sono un costrutto fondamentale della programmazione: il programma chiamante invoca una porzione di codice (subroutine) a cui passa eventualmente dei parametri, e resta in attesa di acquisire il risultato dell'esecuzione della stessa.

Vantaggi riassumibili in

- modularità del codice: ogni subroutine può essere invocata più volte
- riutilizzo del codice: una subroutine scritta correttamente non deve più essere modificata
- semplificazione del codice: un programma strutturato in subroutine è organizzato in blocchi che possono essere analizzati individualmente.

Chiamata di e rientro da subroutine

Una chiamata di subroutine passa il controllo della CPU al codice chiamato. La subroutine quindi come ogni programma macchina deve disporre

- dei registri (al limite tutti!)
- di un sufficiente spazio di memoria dedicata.

In più, occorre che il programma chiamante e la subroutine condividano della memoria per

- il passaggio di parametri,
- la restituzione del risultato.

La stessa memoria dev'essere interamente recuperata al termine dell'esecuzione di ogni subroutine (rientro).

Chiamata di subroutine: salto

Assembly per ARM dedica un'istruzione di salto e delle istruzioni di copia multipla che agevolano la programmazione di subroutine:

- `bl #const` branch and link
  salta di `#const` word e salva in `lr` l'indirizzo di rientro (cioè quello dell'istruzione successiva a `bl`), che in tal modo resta memorizzato per tutta l'esecuzione della subroutine
- il rientro avviene eseguendo una `mov pc, lr`.

N.B.: l'istruzione `bl` permette in particolare a una subroutine di richiamare sè stessa. Ciò permette di realizzare una chiamata ricorsiva.

Esempio di salti alle subroutine

```
.text
main:           ...
                bl fibonacci
                ...
                bl fibonacci
                bl fattoriale
                ...
fattoriale:     ...
                ...
                mov pc, lr
fibonacci:      ...
                ...
                mov pc, lr
```

Chiamata di subroutine: parametri e risultati

Nei casi più semplici di passaggio dei parametri è sufficiente adoperare i registri. In tal caso si adopera la seguente convenzione sul loro uso:

- `r0, ..., r3` sono utilizzati per passare argomenti a una subroutine
- `r0, r1` sono utilizzati per restituire al chiamante i risultati calcolati dalla subroutine.

Nei casi in cui quattro registri non siano sufficienti occorrerà effettuare il passaggio dei parametri attraverso la memoria.

N.B.: `lr` memorizzerà al più un indirizzo di rientro.

Esempio: passaggio di parametri

```
.text
main:           ...
                mov r0, #5 ; parametro in r0
                bl fattoriale
                ...
fattoriale:     ...
                movs r1, r0  ; uso r1 nei calcoli
                beq end_fatt
                ...
                mul r0, r1, r0 ; !n = n*!(n-1)
end_fatt:       mov pc, lr ; risultato in r0
fibonacci:      ...
```

Interferenza sui registri

Poichè programma chiamante e subroutine possono a doperare gli stessi registri, bisogna evitare interferenze sul loro uso:

```
main:           ...
                add r4, r4, #1
                mov r0, #5
                bl fattoriale
                ...
fattoriale:     ...
                ...
                move r4, r0
                ...
                mov pc, lr
```

Salvataggio dei registri

L'interferenza si risolve salvando i registri in uso alla subroutine prima dell'inizio della sua esecuzione. Quindi, ogni subroutine che vuole utilizzarli

- come prima operazione dopo la chiamata, salva in memoria i registri `r4-r14`
- come ultima operazione prima del rientro, ripristina il valore originale di `r4-r14`.

Dualmente, il programma chiamante può avere bisogno di conservare il contenuto dei registri. In tal caso, quando chiama una subroutine

- salva i registri `r0-r3` in memoria prima di chiamare una procedura
- ripristina `r0-r3` dopo il rientro.

Istruzioni di `load` e `store` multipli

Assembly per ARM mette a disposizione le istruzioni `ldm` e `stm` per la copia multipla di registri.

Es.: `stmfd sp!, {r0, r4-r6, r3}` multiple store

- salva in locazioni decrescenti di memoria, a partire dall'indirizzo `[sp]` -4 (`[r13]` -4), il contenuto dei registri `r0, r4, r5, r6, r3`
- aggiorna `sp` alla locazione contenente l'ultimo valore inserito:

  ```
  r1 = r13 - 5*4
  ```

Es.: `ldmfd sp!, {r0, r4-r6, r3}` multiple load ripristina il contenuto di tutti i registri (compreso `sp`).

Es.: salvataggio nella subroutine

```
main:           ...
                add r4, r4, #1
                mov r0, #5
                bl fattoriale ; non usa r0-r3
                ...
fattoriale:     ...
                stmfd sp!, {r4-r5} ; userà r4-r5
                ...
                move r4, r0 ; qui usa r4
                ...
                ldmfd sp!, {r4-r5}
                mov pc, lr
```

Es.: salvataggio dal chiamante

```
main:           ...
                add r2, r2, #1
                mov r0, #5
                stmfd sp!, {r2,r3} ; usa r2-r3
                bl fattoriale
                ldmfd sp!, {r2-r3}
                ...
fattoriale:     ...
                ...
                move r2, r0 ; non usa r4-r14
                ...
                mov pc, lr
```

Suffissi in `load` e `store` multiple

La ripetuta chiamata di subroutine impila in memoria il contenuto dei registri. Il comando `stm` fa riferimento alla cima della pila, o stack, indirizzata da `sp`, a seconda che la cima sia definita come la locazione con l'ultimo dato o la prima senza dato:

- `stmia r13!, {...}` incrementa da `sp`
- `stmib r13!, {...}` incrementa da `sp+4`
- `stmda r13!, {...}` decrementa da `sp`
- `stmdb r13!, {...}` decrementa da `sp-4`.

Esistono poi corrispondenti suffissi se lo stack è orientato nell'altro verso: `fd`, `fu`, `ed`, `eu`.

Stack frame

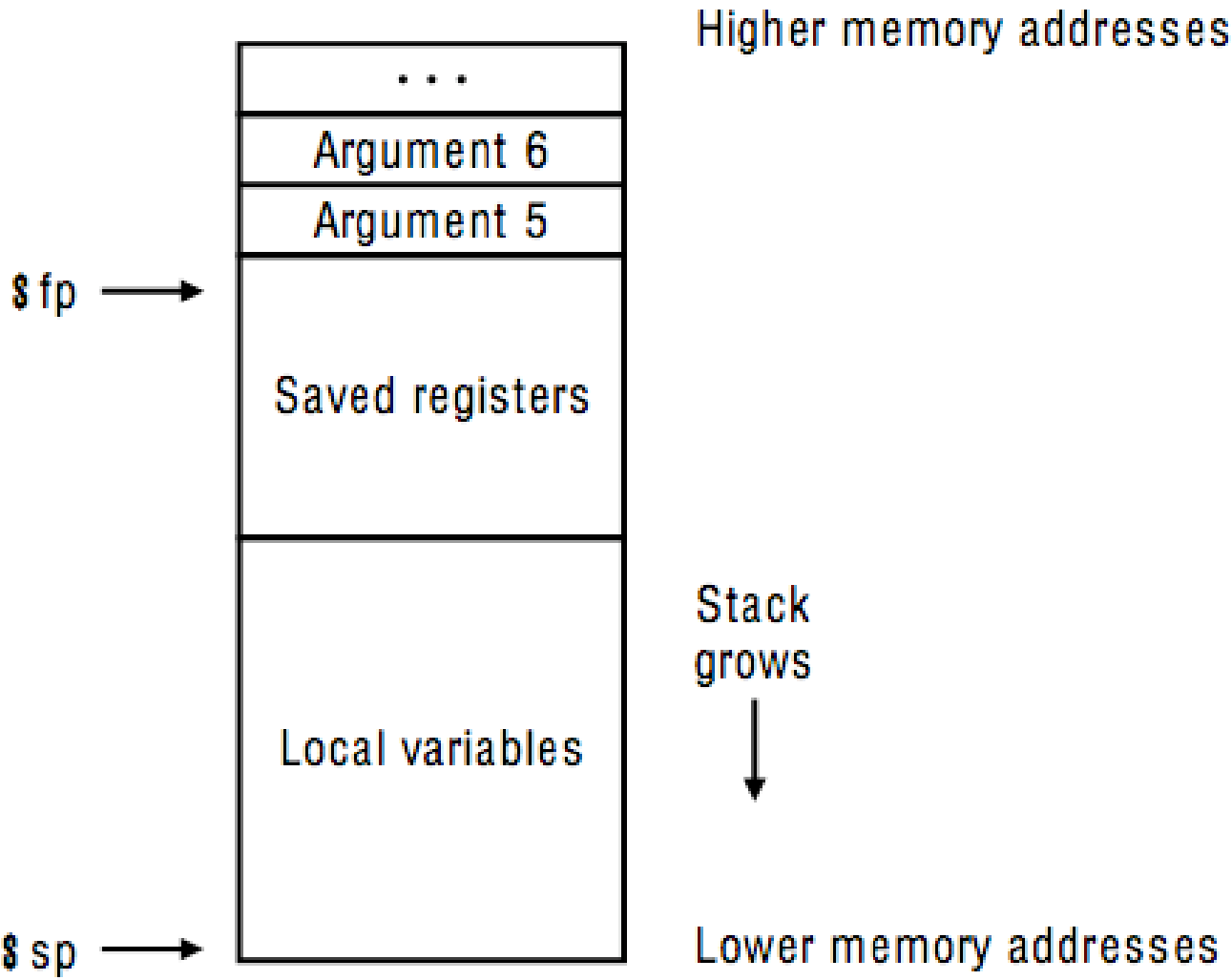
Più in generale, lo stack costituisce l'unico spazio di memoria per la subroutine. Essa avrà a disposizione un frame dello stack per

- acquisire i parametri
- salvare i registri
- operare sui dati
- restituire i risultati.

Poichè queste necessità sono sequenziali, un frame è sufficiente a soddisfarle tutte. Ogni chiamata di procedura alloca un nuovo frame. Ogni rientro da procedura libera l'ultimo frame. Infatti, l'ultima procedura chiamata è la prima a terminare. Ciò rende possibile la crescita indefinita del frame in uso.

Formato di un frame

Può essere utile un ulteriore indirizzo: frame pointer.



Es.: chiamata di `fattoriale` $f(n) = n!$

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

```
; calcolo di fattoriale(n) = f(n) = n!
.data
valore: .word 4
.text
main:   ldr r4, =valore
        ldr r0, [r4]        @ carica il dato n in r0
        bl fattoriale       @ chiamata di funzione
        str r0, [r4]        @ salva il risultato
        swi 0x11            @ termina il programma
.end
```

Realizzazione iterativa di `fattoriale`

```
fattoriale:
        mov r1, #1          @ iniz. contatore
        mov r2, #1          @ iniz. risultato parziale
loop:   cmp r1, r0          @ inizio ciclo
        bge exit
        add r1, r1, #1      @ agg. contatore
        mul r2, r1, r2      @ agg. risul. parz.
        b  loop
exit:   mov r0, r2
        mov pc, lr
```

Realizzazione ricorsiva di `fattoriale`

```
fattoriale:
        stmfd sp!, {r4, lr} @ salva registri
        mov r4, r0
        sub r0, r0, #1
        cmp r0, #1          @ se r0 == 1 ...
        beq skip            @ va a svuotare lo stack
        bl fattoriale       @ chiamata ricorsiva
skip:   mul r0, r4, r0      @ accumula risultato
        ldmfd sp!, {r4, lr} @ ripristina registri
        mov pc, lr          @ rientra dalla procedura
```

La memoria in ARMsim

ARMsim organizza la memoria come segue:

- `0x0000 - 0x0FFF`: riservata al sistema operativo
- `0x1000 - 0xNNNN` : riservata a programma (`.text`) e costanti (`.data`)
- `0xNNNN - 0x5400`: stack
- `0x5400 - 0x11400`: heap per allocazione di strutture dati dinamiche.

Valori modificabili con opportune direttive di assemblaggio.

Il registro `r13`, `sp` viene inizializzato a $0x5400$.
Il registro `r15`, `pc` viene inizializzato a $0x1000$.

Es.: routine per la divisione intera

Realizza la divisione intera tra binari adoperando l'algoritmo della scuola elementare.

```
divisione:      ; r0 / r1 (interi positivi)
                ; fine: resto in r0, risultato in r1
        mov r3, #31             ; contatore
        mov r2, #0              ; inizializza r2 a 0
loop:   mov r2, r2, lsl #1 ; risultato parziale
        cmp r1, r0, lsr r3 ; se r1 < (r0>>) ...
        suble r0, r0, r1, lsl r3  ; ...sottrai <<r1 a r0
        addle r2, r2, #1   ; aggiorna risultato parziale
        subs r3, r3, #1    ; aggiorna contatore
        bge loop
        mov r1, r2
        mov pc, lr
```

Esercizi

- Scrivere una funzione che sommi tutti gli elementi di indice pari di un array.
- Scrivere una procedura che azzeri tutti gli elementi negativi di un array.
- Scrivere una procedura che determini se un numero è primo.

Es.: manipolazione di stringhe

Sostituzione di caratteri in una stringa

```
.data
stringa:.asciiz "stringa da manipolare"
a:      .ascii "a"
o:      .ascii "o"
.text
main:   ldr r0, =stringa  @ inizializza registri
        ldr r1, =a
        ldrb r1, [r1]
        ldr r2, =o
        ldrb r2, [r2]
        bl scambia        @ chiamata procedura
        swi 0x11
.end
```

Es.: manipolazione di stringhe

Procedura per la sostituzione dei caratteri

```
scambia:
        ldrb r3, [r0], #1
        cmp r3, #0                  @ test fine stringa
        beq exit
        cmp r3, r1                  @ confronto caratteri
        streqb r2, [r0, #-1]        @ modifica stringa
        cmp r3, r2                  @ confronto caratteri
        streqb r1, [r0, #-1]        @ modifica stringa
        b  scambia                  @ ciclo
exit:   mov pc, lr
```

Esercizi

1. Scrivere una procedura che determini se il processore funziona in modalità big-endian o little-endian.
2. Scrivere una procedura che determini se la stringa di lunghezza come da `r1`, contenuta in memoria all'indirizzo base `r0`, è palindroma.
3. Scrivere una procedura che determini se il contenuto del registro `r0` è una sequenza binaria palindroma.

Software interrupt

L'istruzione `swi 0xCODE` software interrupt chiama una procedura eseguita dal sistema operativo identificata dall'argomento `0xCODE`.

Diversamente dalla chiamata standard, garantisce la protezione della procedura. Infatti, il processore che esegue un programma assembly prevede almeno due livelli di funzionamento

- system: qualunque procedura può essere eseguita
- user: solo i programmi utente possono essere eseguiti.

L'accesso a una procedura di sistema dal livello user è possibile solo mediante la `swi` .

Software interrupt in ArmSim

ArmSim simula una serie di procedure di sistema.

operazione	cod.	argomento
`print_char`	0x00	`r0` char
`print_string`	0x02	`r0` string address
`exit`	0x11	
`allocate`	0x12	`r0` size $\Rightarrow$ address
`open`	0x66	`r0` name $\Rightarrow$ handle, `r1` mode
`close`	0x68	`r0` handle
`write str`	0x69	`r0` handle, `r1` string
`read str`	0x6a	`r0` handle, `r1` string, `r2` size
`write int`	0x6b	`r0` handle, `r1` integer
`read int`	0x6c	`r0` handle $\Rightarrow$ integer

Gestione dei file di testo

Un file di testo può essere aperto in modalità

- input (mode 0): solo lettura
- output (mode 1): lettura e scrittura – sovrascrive il contenuto
- append (mode 2): lettura e scrittura – accoda il contenuto.

La `open` restituisce un identificatore (handle) numerico. Il flusso in uscita su schermo (stdout: standard output) ha handle = 1.

ArmSim non implementa (?) il flusso in ingresso da tastiera (stdin: standard input).

`read str`: legge al più `r2` caratteri di una riga di testo, e passa alla riga successiva.

Identificatori e nomi simbolici

Le costanti possono essere associate a degli identificatori mediante la direttiva `.equ`:

```
.equ   PrintInt, 0x6b
...
swi PrintInt
```

I registri possono essere associati a dei nomi simbolici mediante la direttiva `.req`, a vantaggio della leggibilità:

```
lo .req r0
hi .req r1
adds lo, lo, r2
adcs hi, r3, lo
```

Strutture dati: matrici

Le matrici vengono linearizzate in memoria, ovvero ridotte a strutture monodimensionali di tipo array.

Es.:

$$\begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 5 & 8 & 1 \\ 3 & 7 & 1 & 5 \end{pmatrix}$$

Per riga:

1 3 5 7 2 5 8 1 3 7 1 5

Per colonna:

1 2 3 3 5 7 5 8 1 7 1 5

Matrici

Data una matrice M con m righe ed n colonne, indichiamo gli elementi con $M[i, j]$, $i \in [0, m-1]$, $j \in [0, n-1]$.

Nella linearizzazione di M, l'elemento $M[i, j]$ occupa la posizione

- $i \times n + j$ (memorizzazione per righe)
- $j \times m + i$ (memorizzazione per colonne).

Nella memorizzazione per righe, se l'elemento $M[i, j]$ occupa il word d'indirizzo x allora

- l'elemento $M[i + d, j]$ occupa il word $x + n \times d$
- l'elemento $M[i, j + d]$ occupa il word $x + d$.

Esercizi

1) Scrivere una procedura che, ricevuti in `r0` e `r1` indirizzo base e lunghezza di un array di interi (word), calcoli in `r0` la somma degli elementi.

2) Scrivere una procedura che ricevuti in `r0` l'indirizzo base di una matrice, in `r1` il numero di righe, in `r2` il numero di colonne, e infine in `r3` l'indirizzo base di un array, inserisca nello stesso array le somme degli elementi delle righe della matrice. Ripetere l'esercizio stavolta sommando le colonne.

3) Scrivere una procedura che calcola la somma degli elementi con indici che hanno somma pari di una matrice quadrata.

Strutture dati dinamiche

Strutture che evolvono dinamicamente durante l'esecuzione di un programma. Non è dunque nota a priori l'occupazione in memoria dei dati.

- I dati sono contenuti in nodi collegati tra loro tramite puntatori.
- Un puntatore appartiene al nodo e contiene l'indirizzo (riferimento) di uno o più nodi.
- I nodi possono essere distribuiti ovunque nello heap.

In assembly gli indirizzi di memoria sono dati di tipo unsigned word (32 bit).

Tipiche strutture dinamiche: liste e alberi.

Strutture dinamiche: realizzazione

Ogni nodo di una lista consiste in una coppia

- dato, di tipo costante lungo tutta la lista (intero, carattere, . . .)
- puntatore: riferimento al nodo successivo.

Si accede alla lista mediante un riferimento al primo nodo.
Occorre etichettare l'ultimo nodo in qualche modo (riferimento NULL; es.: puntatore alla locazione `0x00000000`).

Ogni nodo di un albero binario consiste in una terna: dato; riferimento a un albero binario destro; riferimento a un albero binario sinistro. Nodi contenenti due riferimenti NULL sono detti foglie.

Creazione dinamica dei nodi

Ogni nodo è creato durante l'esecuzione mediante una chiamata a subroutine di sistema.

`swi 0x12` alloca uno spazio di memoria nello heap a runtime: `r0` passa alla procedura il numero di byte da allocare, e contiene dopo il rientro l'indirizzo iniziale della memoria allocata.

La libertà sulla definizione del nodo permette la costruzione di ogni possibile struttura dinamica.

Esercizi

- Scrivere una procedura per calcolare la somma dei valori contenuti in una lista di interi.
- Scrivere una procedura che, dato n, costruisce la lista dei primi n numeri naturali in ordine decrescente.
- Scrivere una procedura che, data una lista di interi, elimina i nodi in posizione pari e duplica quelli in posizione dispari. Si richiede che il primo nodo della lista resti inalterato.

Esercizi

- Scrivere una procedura per calcolare il bit di parità di una parola.
- Scrivere una procedura per calcolare il logaritmo in base 2 di un numero.