Assembly

Programmazione in linguaggio macchina (o meglio in assembly):
programmare utilizzando istruzioni direttamente eseguibili dal processore.

Questa parte del corsi si accompagna a lezioni in laboratorio: programmazione in assembly con uso di un simulatore.

Motivazioni

- Programmare in assembly aiuta a capire funzionamento e meccanismi base di un calcolatore.
- Fa meglio comprendere cosa accade durante l'esecuzione di un programma scritto ad alto livello.
- Insegna una metodologia, si acquisiscono delle abilità.
- In alcuni casi è utile programmare in assembly.

Programmazione Assembly

Vantaggi

controllo dell'hardware: si definiscono esattamente le istruzioni da eseguire e le locazioni di memoria da modificare.

Utile per:

- gestione hardware, risorse del computer (kernel, driver)
- ottimizzare il codice: programmi più efficienti.

Programmazione Assembly

Svantaggi:

- scarsa portabilità: i programmi ad hoc per una famiglia di processori,
- scomodi: istruzioni poco potenti, programmi lunghi,
- facile cadere in errore: programmi poco strutturati e leggibili.
- compilatori sempre più sofisticati producono codice efficiente (soprattutto per le architetture parallele), è difficile fare meglio programmando in assembly.

Processore scelto: ARM

Tanti linguaggi macchina quante le famiglie di processori: x86 (IA-32 Intel), PowerPC (IBM, Motorola), Sparc, MIPS, ...

Linguaggi simili a grandi linee, ma nel dettaglio con molte differenze.

In questo corso: processore ARM (Acorn RISC Machine), uno dei primi processori RISC (Acorn Computers, Wilson and Furber, 1985): poche istruzioni semplici e lineari.

Processore ARM

Una famiglia di architetture (ISA: Instruction Set Architecture) (e processori): ARMv1, ... ARMv8, Diversi insiemi di istruzioni:

- nel tempo si sono aggiunte istruzioni (divisione, istruzioni vettoriali)
- esistono versioni ARM che inglobano le istruzioni:
  - Java bytecode,
  - istruzioni Thumb a 16 bit
- tutte architetture a 32-bit: istruzioni, registri interni, indirizzi di memoria di memoria 32, solo l'ARMv8 gestisce registri interni e indirizzi di memoria 64-bit,

Processore ARM

Uso:

- produzione: 10^{10} pezzi l'anno.
- 95% dei smartphone, tablet
- sistemi embedded: televisori digitali, DVD, router, ADSL modem, TiVo ....

Possibili alternative

- IA-32 del Core (Pentium). Linguaggio difficile da imparare e da usare. Un aggregato di istruzioni costruito nell'arco degli anni. Problemi di legacy (compatibilità con linguaggi precedenti).
- Assembly 8088. Descritto nel libro di testo. Piuttosto vecchio, progenitore del IA-32, diverso dai linguaggi macchina attuali.
- MIPS Processore presentato sino allo scorso anno. Istruzioni simili, più semplici e lineari. Forse più adatto alla didattica, meno presente sul mercato.

ARM

Vantaggi:

- processore ampiamente usato,
- insieme di istruzioni semplice e elegante, facile da imparare,
- simile agli altri RISC.

Riferimenti

In rete disponibile ampia documentazione su ARM, problema trovare una presentazione non troppo completa e complessa.
In inglese e reperibili nella pagina web del corso:

- manuale completo delle istruzioni ARM;
- lista sintetica istruzioni ARM;
- lucidi di un corso analogo dell'Universitá di Padova;
- link ad un corso on line.

Simulatore

Necessario per eseguire i programmi, controllarne il comportamento, correggerli.

- ARM SIM simula un processore ARM-v5, sviluppato dalla University of Victoria (Canada) progetto didattico, di libero accesso, `http://armsim.cs.uvic.ca/`
  parola chiave per ricerca web: 'armsim'
- esistono altri simulatori, commerciali, più completi, più complessi da usare.

Simulatore: ARMSim

Esegue i programmi assembly. Permette di:

- caricare programmi assembly;
- compilare (controllo la correttezza sintattica, generazione codice in linguaggio macchina);
- eseguire il codice, eventualmente passo passo;
- mostrare contenuto di memoria e registri;
- fornisce semplici chiamate al sistema operativo.

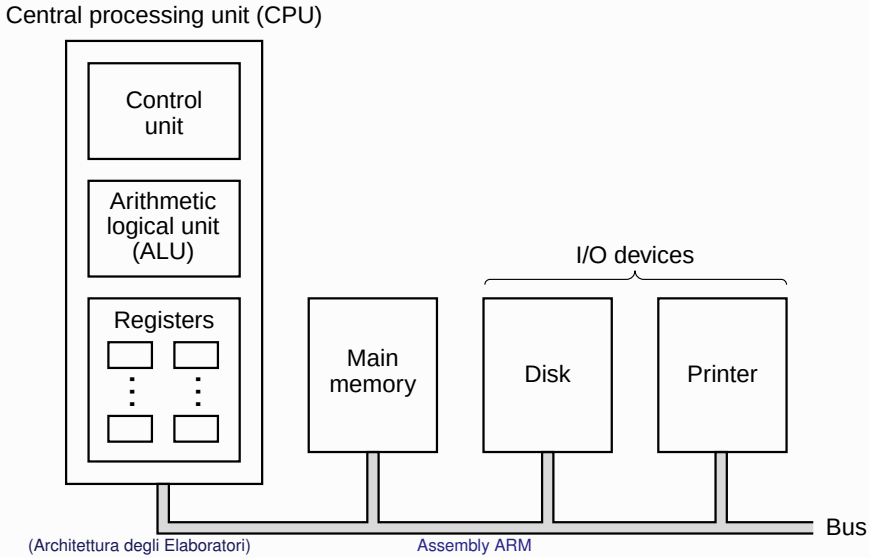
Struttura schematica di un calcolatore

Le istruzioni di un linguaggio macchina agiscono direttamente sul hardware.

Fanno riferimento ad un modello di calcolatore.

Mostriamo a grandi linee questa modello.

Struttura schematica di un calcolatore



Semplificazione rispetto al hardware

Dal punto di vista dell'assembly, il calcolatore si compone di principalmente di:

- Processore, con un certo numero di registri.
- Memoria principale.

Nella realtà, l'hardware è piuttosto complesso, questa complessità non direttamente visibile dall'assembly.

Semplificazione rispetto al hardware

Elementi dell'hardware nascosti all'assembly.

- Pipeline, processori superscalari.
- Processore con registri ombra, più registi di quelli effettivamente visibili al programmatore,
- Memoria cache.
- Memoria principale diversa da quella modellata. Memoria virtuale.
- Periferiche.

Nel corso verranno presentati questi aspetti..

Struttura della memoria

Istruzioni assembly fanno riferimento a:

- una memoria, in cui sono presenti dati e codice da eseguire,
- un insieme di registri, contenuti nel data path, argomento e destinazione delle operazioni aritmetiche logiche

Struttura memoria ARM

- Memoria principale: composta da 2^{32} locazioni, della dimensione di un byte. Ogni locazione individuata da un indirizzo di 32 bit.
- Registri processore: 16 registri *generici* di 4 byte (32 bit), `r0, ...r15`.
  Rappresentazione simbolica alternativa: i registri `r13, r14, r15` possono essere indicati, nell'ordine come `sp, lr, pc`,
  le sigle indicano l'uso tipico del registro
  Stack Pointer, Link Register, Program Counter

Nota: l'assembler non distingue il maiuscolo dal minuscolo (`r0` = `R0`)

Registri ARM

- registri `r0 ... r13`
  perfettamente analoghi per l'hardware;
- registro `r15` contiene il Program Counter, incrementato ad ogni istruzione;
- registro `r14` contiene il Link Register, modificato dalle istruzioni di chiamata procedura;
- registro `r13` per convenzione contiene lo Stack Pointer;
- registri specializzati: `cprs`
  (Current Program Status Register),
  contiene informazioni sullo stato del programma.

Operazioni aritmetiche in ARM

Agiscono sui registri, tre argomenti.

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

Operano su numeri interi, in complemento a due.
In ogni istruzione, il terzo argomento può essere una costante o un registro.

Operazioni aritmetiche in ARM

- `mul r0, r2, r3` multiply, `r0 = r2 * r3`
  `mul` non ammette argomento costante, (solo registri),
  esisto altre istruzioni di moltiplicazione;
- `adc r0, r2, r3` add carry, somma anche il bit di riporto generato dall'op. precedente,
  `r0 = r2 + r3 + c`, dove `c` è il bit di carry,
  permette somme su 64 bit;
- `sbc r0, r2, r3` subtract carry, considera anche il bit di carry `r0 = r2 - r3 + c - 1`,
  permette sottr. su 64 bit;
- `rsc r0, r2, r3` reverse subrtract carry,

Esercizi:

Scrivere pezzi di codice ARM che inseriscano in `r1` il valore delle espressioni:

- `r1 = r1 + r2 + r3` ,
- `r1 = r1 - r2 - 3` ,
- `r1 = 4` $\times$ `r2` ,
- `r1 = - r2` ,
- `r0 r1 = r2 r3 + r4 r5` somma a 64 bit
- `r0 r1 = r2 r3 - r4 r5` sottrazione a 64 bit

Operazioni logiche

estese ai registri (sequenze di bit),

- `and r0, r2, r3` and bit a bit tra due registri,
- `and r0, r2, #5` secondo argomento costante
- `orr r0, r2, r3` or
- `eor r0, r2, #5` exclusive or
- `bic r0, r2, r3` bit clear (r1 = r2 and (not r3))
- `mvn r0, r2` not, move negate
- `mov r0, r2` funzione identità, move

Esercizio: calcolare il resto della divisione per 16.

Costanti rappresentabili

L'ultimo argomento di ogni operazione aritmetica-logica (esclusa la moltiplicazione) può essere una costante numerica,
scritta in

- decimale:
  `sub r0, r2, #15`
- o in esadecimale:
  `sub r0, r2, #0xf`

Costanti numeriche — Immediate_8r

Costanti rappresentate in memoria con 8 + 4 bit

- 8 bit di mantissa
- 4 bit rappresentano lo spostamento a sinistra, il valore dello spostamento viene moltiplicato per 2, solo spostamenti pari.

Non tutte le costanti sono rappresentabili

- Costanti rappresentabili (valid immediate 8_r)
  `0xFF 255 256 0xCC00 0x1FC00`
- Costanti non rappresentabili:
  `0x101 257 0x102 258`

Istruzioni con constanti non rappresentabili generano errore

Esercizi

Scrivere pezzi di codice ARM che inseriscano in `r1` il valore delle espressioni:

- `r1 = 57 ,`
- `r1 = 1024 ,`
- `r1 = 257 ,`
- `r1 = #0xAABBCCDD ,`
- `r1 = -1`

Operazione di shift - rotate

Per ogni istruzione aritmetica-logica, se l'ultimo argomento è un registro, a questo si può applicare un operazione di shift o rotate:

```
add r0, r1, r2, lsl #2
 r0 = r1 + (r2 « 2)
```

la sequenza di bit in `r2` viene traslata di due posizioni verso sinistra, prima di essere di essere sommata.

costante di shift contenuta nell'intervallo `[0..31]`

Operazione di shift - rotate

`mov r0, r2, lsl r3`

è possibile specificare il numero di posizioni da traslare mediante una registro (`r3`),
solo gli 8 bit meno significativi del registro (`r3`) sono esaminati.

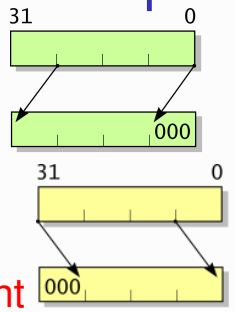
L'esecuzione di:

```
mov r0,#1
mov r1,#0x102
add r2,r0, r0, lsl r1
```

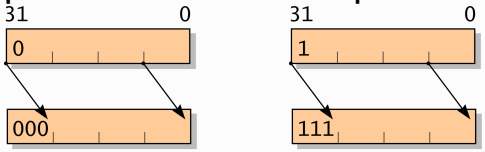
assegna a `r2` il valore 5.

5 tipi di shift

- `lsl` logical shift left

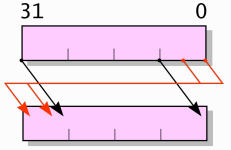


- `lsr` logical shift right
- `asr` arithmetic shift right, si inserisce a sinistra il bit di segno, esegue una divisione per una potenza di 2 in complemento a 2.

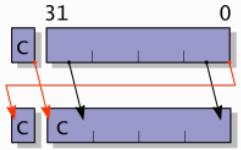


5 tipi di shift

- `ror` rotate right, i bit eliminati a destra rientrano a sinistra



- `rrx` rotate right extended, ruota a destra di una singola posizione coinvolgendo il bit di carry, non ammette argomento.



Esercizi

Scrivere pezzi di codice ARM che inseriscano in `r1` il valore delle espressioni:

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

Istruzioni trasferimento dati

Solo 16 registri: necessario usare la memoria principale.

- load register - da memoria a registro
  `ldr r3, [r0, #8]`
  copia nel registro `r3` 4 byte a partire dall'indirizzo: `r0 + 8`, (base e offset).
- `ldr r3, [r0]`
  il campo offset può essere assente

Molte modalità di indirizzamento

- `ldr r3, [r0, #8]!` pre incremento
  `r0 = r0 + 8; r3 = M32[r0]`
- `ldr r3, [r0], #8` post incremento
  `r3 = M32[r0]; r0 = r0 + 8`
- `ldr r3, [r1, - r0]` offset da registro
  `r3 = M32[r1 - r0]`
- `ldr r3, [r1, r0]!` pre inc. da registro
  `r1 = r1 + r0; r3 = M32[r1]`
- `ldr r3, [r1], r0` post inc. da registro
  `r3 = M32[r1]; r1 = r1 + r0`
- `ldr r3, [r1, r0, lsl #2]` registro scalato
  `r3 = M32[r1 + 4 * r0]`

Parole allineate

`ldr` legge 4 byte consecutivi in memoria (per comporre una parole),
legge solo a parole allineate, ossia l´indirizzo del primo byte è un multiplo di 4,

Endianness

Processori ARM, normalmente little-endian: si leggono 4 byte alla volta, bit meno significativi negli indirizzi piccoli.
Ma possono funzionare in modalità big-endian, definiti anche bi-endian

Operazione di store

Per inserire dati in memoria:

- store register - da registro a memoria
  `str r0, [r4,#8]`

Tutte le modalità di indirizzamento di `ldr` valide anche per `str`

Vettori

Tipi di dati complessi.
Sequenza di valori dello stesso tipo.
Es. Una coordinata cartesiana nello spazio $\langle 0, 3, 2 \rangle$

Supportati dai linguaggi alto livello.
Implementati *a mano*, in assembly.

- allocati in memoria principale, in locazioni consecutive;
- si accede, identifica, il vettore mediante l'indirizzo base.

Vettori

I diversi elementi di uno stesso vettore sono identificati da un indice.

Per operare sull'elemento *i*-esimo del vettore bisogna:

- valutare la sua posizione in memoria:
  = "indirizzo base" + "displacement"
  "displacement" := "indice" $\times$ "dimensione elemento".
- leggerlo o scriverlo con `ldr, str`

Esercizio

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
```


Esercizio

Scrivere in `r1` il valore A[`r2`] + A[`r2` + 1] + A[`r2` + 2] del vettore con indirizzo base `r0`, l´indirizzo base del vettore `A` è contenuto in `r0`.

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


Trasferire parti di una parola

Dalla memoria è possibile leggere o scrivere anche:

- un singolo byte, sequenza di 8 bit:
  load register byte `ldrb`
  load register signed byte `ldrsb`
  store byte `strb`,
- un half-word, sequenza di 16 bit, 2 byte :
  load register half word `ldrh`
  load register signed half word `ldrsh`
  store half word `strh`.

per gli half-word l'indirizzo deve essere allineato, multiplo di 2.

Sintassi

Un programma assembly, file `.s`, non solo sequenza di istruzioni ma anche:

- etichette `item:` permettono di associare un nomi ad un indirizzo.
- direttive `.text` indicazioni al compilatore (assembler)
  le direttive sono parole precedute da un punto, es. `.data`, `.globl`

Direttive principali

- `.text` quello che segue è il testo del programma.
- `.data` quello che segue sono dati da inserire in memoria
- `.bss` specifica la sezione contenente dati non inizializzati
- `.globl` rende l'etichetta che segue visibile agli altri pezzi di codice
- `.end` specifica la fine del modulo sorgente

Direttive rappresentazione dati

Specificano che tipo di dati inserire in memoria:

- `.word 34, 46, 0xAABBCCDD, 0xA01`
  ogni numero intero scritto con 4 byte
- `.byte 45, 0x3a`
  ogni numero intero scritto con un byte
- `.ascii "del testo tra virgolette "`
  i codici ascii dei caratteri della stringa,
- `.asciiz "altro esempio"`
  si aggiunge un byte 0 per marcare fine stringa.
- `.skip 64`
  vengo allogati 64 byte, inizializzati a 0.

Eventuali etichette usate per dare un nome alla

Esempio di programma

```
.data
primes:  .word 2, 3, 5, 7, 11
string:  .asciiz "alcuni numeri primi"

.text
main:    ldr r0, =primes
         ldr r1, [r0], #4
         ldr r2, [r0]
         ldr r0, =string
         ldrb r3, [r0], #1
         ldrb r4, [r0]
.end
```

Pseudo-istruzioni

`ldr r0, =primes` è una pseudo-istruzione.
inserisce in `r0` l'indirizzo associato a `primes`

Per semplificare la programmazione, l'assembly ARM contiene istruzioni extra rispetto a quelle del linguaggio macchina.

Pseudo-istruzioni vengono tradotte in sequenze di (1-4) istruzioni macchina.

Le pseudo-istruzioni utili e naturali. In programmazione, quasi indistinguibili dalle istruzioni.

Non implementate dal processore, non inserite nel linguaggio macchina, per motivi di efficienza.

Istruzioni per controllo di flusso

Programmi sono più complessi di una semplice lista di istruzioni, devono eseguire istruzioni diverse in base ai dati.

Nei linguaggi ad alto livello:

- test (`if-then-else`)
- cicli (`while, repeat`)

In assembly, meccanismi di controllo del flusso elementari e poco strutturati:

- salti condizionati
- salti incondizionati

Salto incondizionato

branch

`b etichetta`

salta all'istruzione etichettata con `etichetta`

l´assembly permette di creare un associazione etichetta – indirizzo, istruzione

```
label :  add r0, r1, r1
```


Istruzioni condizionate

- Le istruzioni ARM possono essere eseguite in maniera condizionata.
- La condizione dipende da 4 bit del registro di stato `cprs`.
- Inserendo il suffisso `s` al nome di un'istruzione questa modifica il registro di stato.
- La condizione viene specificata da un infisso di due lettere.

  `add addne addnes`

Esempi

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

L'Istruzione `subs` modifica il registro `cprs`.

Se il risultato di `subs r0, r1, r2` è uguale a zero, le istruzioni `addeq r2, r2, #1` e `beq label` sono eseguite.

Elenco condizioni

Le condizioni considerano 4 bit:

- Z zero
- N negative
- C carry
- V overflow, l'operazione, su dati signed, ha generato 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 && (N == V)
le	Signed less than or equal	Z
al	Always (default)	any

Istruzioni di confronto

Modificano solo i 4 bit (flag) del registro di stato, senza aggiornare alcune registro.

- 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`

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

salta a label solo se `r2`, come numero intero, è minore o uguale a `7`.

Esempi, istruzione di test

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

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

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

codice alternativo:

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


Schema generale di traduzione

Una generica istruzione di if-then-else

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

viene tradotta in:

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


Schema generale

Una generica istruzione di if-then

```
if Bool then Com fi
```

viene tradotta in:

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


Esempi, cicli

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

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

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

Schema traduzione dei cicli

Una generica istruzione di ciclo

`while Bool do Com od`

viene tradotta secondo lo schema:

```
while:  Eval not Bool
        b_cond fine
        Com1
        b while
fine:
```


Assembly e linguaggio macchina

Linguaggio assembly: notazione mnemonica usata dal programmatore per scrivere, analizzare e rappresentare programmi in linguaggio macchina.
Linguaggio macchina: notazione utilizzata dal calcolatore per memorizzare ed eseguire programmi, ogni istruzione è rappresentata mediante una sequenza di bit.

Assembly e linguaggio macchina

La notazione: `subs r1, r2, r3` è assembly: notazione mnemonica di istruzione macchina, sequenza di bit usati nel calcolatore (linguaggio macchina):

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

Esiste una corrispondenza 1 a 1 tra istruzioni assembly e istruzioni macchina.

In ARM, ogni istruzione macchina utilizza 32 bit.

Formato macchina istruz. di salto

- Nella rappresentazione in linguaggio macchina dell'istruzione di salto
  `beq label`
  sono riservati 24 bit per specificare l'indirizzo di salto.
- Salto relativo: si specifica di quante istruzioni saltare in avanti od indietro.
- Al registro `r15` (program counter, `pc`) viene sommato un numero intero.
- Numeri negativi salti all'indietro.
- Range di indirizzi raggiungibili: $\pm$32MB.

Esercizi

- Calcolare, e scrivere nel registro `r1`, l'n-simo numero di Fibonacci con *n* il contenuto nel registro `r0`.
- Calcolare, e scrivere nel registro `r1` la somma degli elementi di un vettore *V* con 10 elementi, con indirizzo base il valore di `r0`,

Moltiplicazione estesa

signed multiplication long: `smull r0, r1, r2, r4`
Calcola il prodotto, in complemento a due, tra `r1` e `r2`, il risultato, di 64 bit, nei due registri `r1`, cifre più significative, e `r0`, cifre meno significative.

unsigned multiplication long: `umull r0, r1, r2, r4`
Calcola il prodotto, a 64 bit e in binario puro, `r1` e `r2`, il risultato, nei due registri `r1` e `r0`.

Non esistono istruzioni per la divisione.

Procedure, funzioni

Meccanismo base della programmazione.

- Struttura il programma in parti (procedure e funzioni).
- Scrittura modulare dei programmi.
- Riutilizzo del codice.
- Necessarie per gestire la complessità dei programmi.

Utilizzate in programmazione assembly.

Supportate dai linguaggi macchina, ma non immediatamente disponibili:
una chiamata di procedura corrisponde ad una sequenze di istruzioni macchina.

Procedure, funzioni

Una chiamate di procedura comporta:

- passare il controllo ad altro codice,
- passare i parametri,
- allocare spazio di memoria per variabili locali,

L'uscita una procedura comporta:

- recuperare lo spazio in memoria,
- ritornare controllo (e dati) al chiamante.

Implementati con meccanismi elementari in ARM.

Mostriamo come le chiamate di procedura vengono tradotte in linguaggio macchina.

Chiamata di procedure

Necessità di ricordare indirizzo di ritorno:
Branch and link
`bl Label`
salta all'etichetta `Label`, salva nel registro `lr` in link register, (`r14`)
l'indirizzo di ritorno: dell'istruzione successiva (a `bl Label`).

Uscita dalla procedura:

`mov pc, lr`

Esempio: funzione fattoriale

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


Passaggi dei parametri - risultati

Si usano i registri per il passaggio dei parametri (pochi). Convenzione sull'uso dei registri:

- Registri `r0, ..., r3` utilizzati per passare argomenti ad una procedura-funzione.
- Registri `r0, r1` utilizzati per restituire, al programma chiamante, i valori risultato di una funzione.

Eventuali parametri extra si passano attraverso la memoria.

Esempio: funzione fattoriale

```
.text
main:       ...
            mov r0, #5
            bl fattoriale
            ...
fattoriale: ...
            movs r1, r0
            beq end
            ...
            mul r0, r4, r0
            mov pc, lr
fibonacci:  ...
```

Salvataggio dei registri

Programma principale e procedura agiscono sullo stesso insieme di registri (processore unico).
Bisogna evitare interferenze improprie.

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


Soluzione

- Per convenzione, i registri `r4-r14` sono considerati non modificabili dalle procedure.
- Una procedura, come primo passo, salva in memoria i registri che dovrà modificare.
- Prima di tornare il controllo al programma principale, ne ripristina il valore originale.

Esempio

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


Osservazione

- Registri `r0 - r3` sono considerati modificabili dalle procedure.
- Se è necessario preservare il valore, si salvano in memoria prima di chiamare una procedura.
- Terminata la chiamata, li si ripristina.

Esempio

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


Istruzione di load - store multiple

`stmfd sp!, {r0, r4-r6, r3}`

- salva in locazione decrescenti di memoria,
- a partire dall'indirizzo in `r13`-4,
- il contenuto dei registri `r0, r4, r5, r6, r3`,
- aggiorna `r13` alla locazione contenente l'ultimo valore inserito:
  `r1 = r13 - 5*4`

`ldmfd sp!, {r0, r4-r6, r3}`
ripristina il contenuto di tutti i registri (e dello stack pointer).

Istruzione di load - store multiple

Più versioni:

- `stmfd r8!, {r0, r4-r6, r3}`
- `stmfu r8!, {r0, r4-r6, r3}`
- `stmed r8!, {r0, r4-r6, r3}`
- `stmeu r8!, {r0, r4-r6, r3}`

Istruzione di load - store multiple

Significato dei suffissi.
Si ipotizza che `stm` venga usa per implementare uno stack.
Il registro `r8` punta alla cima dello stack.

- `d` down, lo stack cresce verso il basso, il registro viene decrementato;
- `u` up, lo stack crescite verso l'alto, il registro viene incrementato;
- `f` full, lo stack viene riempito completamente, pre (de)in-cremento;
- `e` empty, l'ultima posizione dello stack è libera, post (de)incremento.

Allocazione spazio di memoria

Ogni procedura necessita di uno zona di memoria per:

- variabili locali;
- salvataggio dei registri;
- parametri e risultati.

Questa zona di memoria allocata in uno stack.

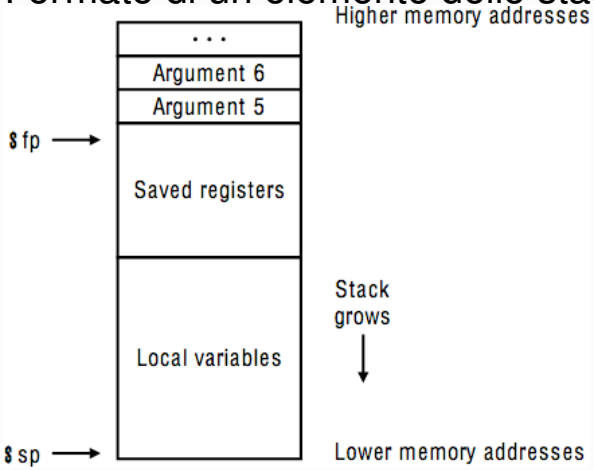
Una pila di zone di memoria consecutive:

- Chiamata di procedura: allocare spazio.
- Uscita dalla procedura: recuperare spazio.

Last In - First Out: la procedura chiamata più recentemente è la prima a terminare.

Stack chiamate di procedura

Formato di un elemento dello stack:



Uso dello stack

Per convenzione, gestito mediante

- il registro `r13`,`sp`, stack pointer, punta all'ultima parola libera dello stack,

Istruzioni iniziali di una procedura: allocazione dello stack, salvataggio registri,
Istruzioni finali di una procedura: ripristino condizioni iniziali.

Per convenzioni lo stack cresce verso il basso,
si decrementa lo stack pointer per allocare spazio.

ARMSim inizializza il registro `sp` in modo che punti alla zona di memoria realizzare lo stack.

Chiamata di funzione

```
.data
valore: .word 4
.text
main: ldr r0, =valore
        ldr r0, [r0]
        bl fattoriale  @ chiamata di funzione
        swi 0x11       @ s.o. ferma il programma
```

Funz. fattoriale ricorsiva

```
fattoriale:
        stmfd sp!, {r4, lr} @salva registri
        cmp r0, #2
        ble skip
        mov r4, r0
        sub r0, r0, #1
        bl fattoriale       @chiamata ricorsiva
        mul r0, r4, r0
skip:   ldmfd sp!, {r4, lr} @ripristina registri
        mov pc, lr          @uscita procedura
```


Fattoriale iterativo

```
fattoriale:
        mov r1, #1
        mov r2, #1
loop:   cmp r1, r0
        bge exit
        add r1, r1, #1
        mul r2, r1, r2
        b  loop
exit:   mov r0, r2
        mov pc, lr
```


Uso della Memoria

ARMSim prevede il seguente uso della memoria:

- $0 - FFF_{hex}$: riservata al sistema operativo.
- $1000_{hex} - xx$: codice programma (.text), costanti (.data)
- $xx - 5400_{hex}$: stack per chiamate di procedura
- $5400_{hex} - 11400$: heap per dati dinamici.

Valori modificabile, attraverso le preferenze.
Il registro `r13`, `sp` viene inizializzato alla cima dello stack 5400_{hex},
Il registro `r15`, `pc` viene inizializzato alla prima istruzione 1000_{hex}

Procedura per la divisione

```
divisione: @ divide r0 per r1 (interi positivi)
           @ resto in r0, risultato in r1
        mov r3, #31
        mov r2, #0
loop:   mov r2, r2, lsl #1
        cmp r1, r0, lsr r3
        suble r0, r0, r1, lsl r3
        addle r2, r2, # 1
        subs r3, r3, # 1
        bge loop
        mov r1, r2
        mov pc, lr
```

Esercizi

- Scrivere una funzione che determini se un numero è divisibile per 2 o per 3. (argomento e risultato in `r0`)
- Scrivere una procedura che, in un vettore, azzeri tutti i valori divisibili per 2 o per 3.
- Scrivere una procedura che inserisca, in un vettore, di dimensione *n*, i primi *n* numeri naturali.
- Scrivere una procedura che, elimini tutti gli elementi uguali a 0 contenuti in un vettore.

Es. programma su stringhe

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


Procedura su stringhe

```
scambia:ldrb r3, [r0], #1
        cmp r3, #0
        beq exit
        cmp r3, r1
        streqb r2, [r0, #-1]
        cmp r3, r2
        streqb r1, [r0, #-1]
        b  scambia
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 contenuta in memoria all'indirizzo base `r0` e di lunghezza `r1`, è palindroma.
3. Scrivere una procedura che determini se il contenuto del registro `r0` è una sequenza binaria palindroma.

Chiamate al sistema operativo

Software interrupt

`swi 0x12`

Simile alle chiamate di procedura. Salta ad una procedura del SO identificata dal parametro.

Distinta della chiamata di procedure per implementare meccanismi di protezione:
Il processore possiede due modalità di funzionamento.

- System: esegue qualsiasi operazione,
- User: alcune operazione non lecite per implementare meccanismi di protezione.

Passaggio controllato tra le due modalità

ARMSim software interrupt

ARMSim fornisce attraverso dei plugin la possibilità utilizzare una serie software interrupt di base.

Per fare questo i plugin devono essere abilitati:
File → Preferences → Plugins → SWIInstructions

operazione	cod.	argomento
`print_char`	0x00	`r0` char
`print_string`	0x02	`r0` string address
`exit`	0x11	
`allocate`	0x12	`r0` size ⇒ address

Software interrupt for files

operazione	cod.	argomento
`open`	0x66	`r0` name ⇒ 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 ⇒ integer

File aperto in modalità input (mode 0), o output (1 – cancella contenuto), o append (2).
`open` restituisce un handle, identifica il file.
Il file di uscita standard (StdOut) ha handle 1.
ARMSim non implementa lo standard input.

Altre direttive assembly

Costanti definite con la direttiva .equ

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

Direttiva da inserire nella parte “.data”.

Nomi simbolici per i registri definiti da .req:

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

L’uso di nomi simbolici migliora la leggibilità.

Istruzione iniziale

Denotata con l'etichetta

```
_start:   mov r0,
```

forza l'inizio da una qualsiasi istruzione.

Assembler dipendente, non universale.
Funziona con ARMSim, non necessariamente con altri simulatori.

Strutture dati: matrici

Le matrici vengono linearizzate: ricondotte a vettori.

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

Per riga:

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

o per colonna:

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

Matrici

Data M una matrice con m righe ed n colonne, indichiamo gli elementi con $M[i,j]$, ($i \in [0, m-1]$ e $j \in [0, n-1]$).
Nella linearizzazione di M, l'elemento $M[i,j]$ occupa la posizione:

- memorizzazione per righe: $i \times n + j$
- memorizzazione per colonne: $i + j \times m$

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

- l'elemento $M[i+1,j]$ occupa l'indirizzo $x + n \times d$,
- l'elemento $M[i,j+1]$ occupa l'indirizzo $x + d$

Esercizi

1) Scrivere una procedura che ricevuti in `r0, r1` indirizzo base e lunghezza di un vettore 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 suo numero di righe in `r2` il numero di colonne e ricevuto in `r3` l'indirizzo base di un vettore, inserisca nel vettore `r3` le somme degli elementi delle righe (delle colone) del vettore.

3) Scrivere una procedura che data una matrice quadrata, calcola la somma degli elementi pari sulle sue diagonali.

Strutture dati dinamiche, puntatori.

Dati di dimensione variabile.

Dati distribuiti in locazione non consecutive di memoria.

Collegati tra loro tramite puntatori (indirizzi di memoria, visti come dati, unsigned word).

Esempi tipici: liste, alberi.

Implementazione

Nel caso di una lista:
ogni elemento di una lista formato da una coppia:

- Puntatore: riferimento al punto successivo, indirizzo in memoria.
- Valore, dipendente dal tipo di lista: interi, caratteri ...

Si accede alla lista mediante un puntatore primo elemento.

Nel caso di alberi binari:
ogni nodo formato da una terna,
valore e due puntatori a due figli.

Creazione di nuovi elementi

I puntatori permettono la definizione di varie strutture dati dinamiche.

Elementi creati con una chiamata di sistema
`swi 0x12`

alloca uno spazio di memoria libero
nel heap: zona di memoria in cui allocare dati dinamici.

`r0` = numero di byte da allocare,
restituisce in `r0` l'indirizzo spazio di memoria.

Esercizi liste

- Scrivere una procedura per calcolare la somma degli elementi di una lista di interi.
- Scrivere una procedura che dato *n*, costruisce la lista dei primi *n* numeri naturali, inseriti in ordine decrescente.
- Scrivere una procedura che data un lista di interi, la modifichi eliminando gli elementi pari e duplicando gli elementi dispari. Si richiede inoltre che il primo elemento 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.