

# *CALCOLATORI*

## *Cenni ad Assembly ARM*

Giovanni Iacca  
[giovanni.iacca@unitn.it](mailto:giovanni.iacca@unitn.it)

*Lezione basata su materiale preparato  
da Luca Abeni, Luigi Palopoli, Fabiano Zenatti e Marco Roveri*



UNIVERSITÀ DEGLI STUDI DI TRENTO

---

Dipartimento di Ingegneria  
e Scienza dell'Informazione

# Architettura ARM

- Utilizzata sulla maggior parte di tablet, smartphone ed in molti sistemi embedded
- Architettura nata negli anni '80
  - 1987: Acorn Archimedes
  - ARM = Acorn Risc Machine
  - 1999: Advanced Risc Machine (nuova azienda!)
- Evoluzione meno complicata rispetto ad Intel
  - ARM “nasce” a 32 bit
  - Solo recentemente nuova architettura a 64 bit...
  - ...Ma non è troppo compatibile col passato!
- Architettura RISC (Reduced Instruction Set Computer) abbastanza “pragmatica”, che cerca di prendere il meglio dal mondo RISC e dal mondo Intel
  - Forse questa è la chiave del suo successo?

# Assembly ARM

- “Strano” RISC con molte modalità di indirizzamento anche potenti
- ISA sviluppato per ovviare ai classici problemi delle architetture RISC
- Altro obiettivo: risparmio energetico
  - Densità del codice...
- “Solo” 16 registri, general purpose
- In realtà ARM è una *famiglia* di CPU, con ISA leggermente differenti
  - Non entreremo nei dettagli, ma considereremo solo gli aspetti in comune fra tutti gli ISA
  - Al solito, assembler gnu

# Registri ARM

- 16 registri a 32 bit, quasi tutti general purpose
- Nomi: da `r0` a `r15`
  - Tecnicamente, `r15` non è un registro general purpose
- Alcuni registri accessibili tramite un nome simbolico che ne identifica l'utilizzo
  - `r13 == sp` (stack pointer)
  - `r14 == lr` (link register)
- Application Program Status Register (`apsr`) / Current Program Status Register (`cpsr`)
  - Flags register

# Utilizzo dei Registri

- Registri general purpose:  $r_0 \dots r_{14}$ 
  - Utilizzo non forzato da ISA, ma dipendente da ABI
- $r_{13}$ : spesso usato come stack pointer ( $sp$ ), ma l'utilizzo dipende da ABI
  - Differenza da Intel: in ISA Intel, lo stack pointer **deve** essere  $r_{sp}$
- $r_{14}$ : talvolta usato come link register ( $lr$ : indirizzo di ritorno da subroutine)
  - Analogo a  $ra / x_{1}$  di RISC-V
- Registro  $r_{15}$ : contiene program counter ( $pc$ ) e flags (bit 31...28)
  - Distinzione chiara fra flag usati per esecuzione condizionale ( $ap_{sr}$ ) ed altri flag

# Flag Register

- `eq` (equal): esegui se la prec. op. era fra numeri uguali (flag `Z` vale 1);
- `ne` (not equal): esegui se la prec. op. era fra numeri diversi (flag `Z` vale 0);
- `hs` (higher or same) o `cs` -carry set- (flag `C` vale 1);
- `lo` (lower) o `cc` -carry clear- (flag `C` vale 0);
- `mi` (minus): esegui se risultato ultima op. è negativo (flag `N` vale 1);
- `pl` (plus): esegui se risultato ultima op. è positivo (flag `N` vale 0);
- `vs` (overflow): esegui se ultima op. è risultata in un overflow (flag `V` vale 0);
- `vc` (overflow clear): opposto di `vs` (esegui se il flag `V` vale 0);
- `hi` (higher): esegui se in un'op. di confronto primo operando è maggiore del secondo, assumendo unsigned (flag `C` vale 1 e `Z` vale 0);
- `ls` (lower or same): esegui se in un op. confronto primo operando è minore o uguale al secondo, assumendo unsigned (flag `C` vale 0 o `Z` vale 1);
- `ge` (greater or equal): esegui se `N` vale 1 e `V` vale 1, o se `N` vale 0 e `V` vale 0;
- `lt` (less than): esegui se `N` vale 1 e `V` vale 0, o se `N` vale 0 e `V` vale 1;
- `gt` (greater than): come `ge`, ma con `Z` che vale 0;
- `le` (less or equal): come `lt`, ma esegue anche se `Z` vale 1.

# Convenzioni di Chiamata

- Non fanno propriamente parte dell'architettura
  - Data una CPU / architettura, si possono usare molte diverse convenzioni di chiamata
  - Servono per “mettere d'accordo” diversi compilatori / librerie ed altre parti del Sistema Operativo
- Tecnicamente, sono specificate dall'ABI, non dall'ISA!!!
- Come / dove passare i parametri
  - Stack o registri?
- Quali registri preservare?
  - Quando un programma invoca una subroutine, quali registri può aspettarsi che contengano sempre lo stesso valore al ritorno?
- Nota: per ARM, esistono molte ABI diverse (anche **molto** diverse!)

# Convenzioni di Chiamata

- Primi 4 argomenti:
  - $r0 \dots r3$
  - Registri non preservati! Utilizzabili anche come registri “temporanei” da non salvare!
- Altri argomenti ( $4 \rightarrow n$ ): sullo stack
- Registri preservati:  $r4 \dots r11$ 
  - Eccezione: in alcune ABI  $r9$  non è preservato
- Valori di ritorno:  $r0$  e  $r1$
- I registri che una subroutine può utilizzare senza dover salvare sono quindi  $r0, r1, r2, r3$  ed  $r12$ 
  - Più eventualmente  $r9$  (dipende da piattaforma / ABI)



## Modalità di Indirizzamento - 1

- Istruzioni *prevalentemente* a 3 argomenti
  - Simile a RISC-V, ma meno regolare: ci sono istruzioni (`mov`, etc...) a 2 argomenti
- Operandi: sinistro e destro
  - Operando sinistro: registro
  - Operando destro: immediato o registro, eventualmente shiftato
  - No operandi in memoria!
- Uniche operazioni che accedono alla memoria: `load` e `store`
  - Load register / Store register (`ldr` / `str`)
  - Load / Store multiple (`ldm` / `stm`)

## Modalità di Indirizzamento - 2

- Operandi in memoria: varie modalità di indirizzamento
- Semplificando un po':
  - Base + Offset (immediato)
  - Base + Indice (eventualmente scalato)
- In più, possibilità di aggiornare il registro base ( `[!]` )
  - Aggiornamento prima dell'accesso (pre indexed) o dopo l'accesso (post indexed)
- $i = \langle \text{base} \rangle + \{ \langle \text{offset} \rangle \mid \langle \text{indice (shiftato)} \rangle \}$ 
  - `<base>`: valore in registro (come per RISC-V)
  - `<offset>`: costante (valore immediato) codificato su 12 bit
  - `<indice>`: valore in registro (semplifica iterazione su array)
    - Opzionalmente shiftato o ruotato
- Pre Indexed:
  - `[rb, #i] [!]`
  - `[rb, {+|-}ro [, <shift>]] [!]`
- Post Indexed:
  - `[rb], #i`
  - `[rb], {+|-}ro [, <shift>]`

## Indirizzamento - Casi Speciali

- Pre Indexed con registro indice:  $[rb, \{+|- \}ro, <shift>]$  [!]
  - $<shift>$  può indicare shift (aritmetici o logici) o rotazioni sul registro  $ro$
  - Se non ci sono shift/rotazioni,  $<shift>$  può essere omissso
    - $[rb, \{+|- \}ro]$  [!]
- Pre Indexed con offset:  $[rb, \#i]$  [!]
  - Offset = 0  $\Rightarrow$  può essere omissso
    - $[rb]$  [!]
- Post indexed con indice:  $[rb], \{+|- \}ro, <shift>$ 
  - No shift  $\Rightarrow$  si omette:  $[rb], \{+|- \}ro$
- Post indexed con offset:  $[rb], \#i$ 
  - Offset = 0  $\Rightarrow$  si omette:  $[rb]$
  - Come pre indexed con offset 0 e senza update!!!

## Indirizzamento: Esempi esplicativi

- Accesso a memoria tramite registro e offset

- **ldr** r0, [r1]                      @ r0 = mem[r1]

- Tre modi di specificare offset:

- Costante:

- **ldr** r0, [r1, #4]                      @ r0 = mem[r1 + 4]

- Registro:

- **ldr** r0, [r1, r2]                      @ r0 = mem[r1 + r2]

- Scalato:

- **ldr** r0, [r1, + r2, lsl #8]              @ r0 = mem[r1 + r2 << 8]

- **ldr** r0, [r1, - r2, lsl #8]              @ r0 = mem[r1 - r2 << 8]

## Indirizzamento: Esempi esplicativi - 2

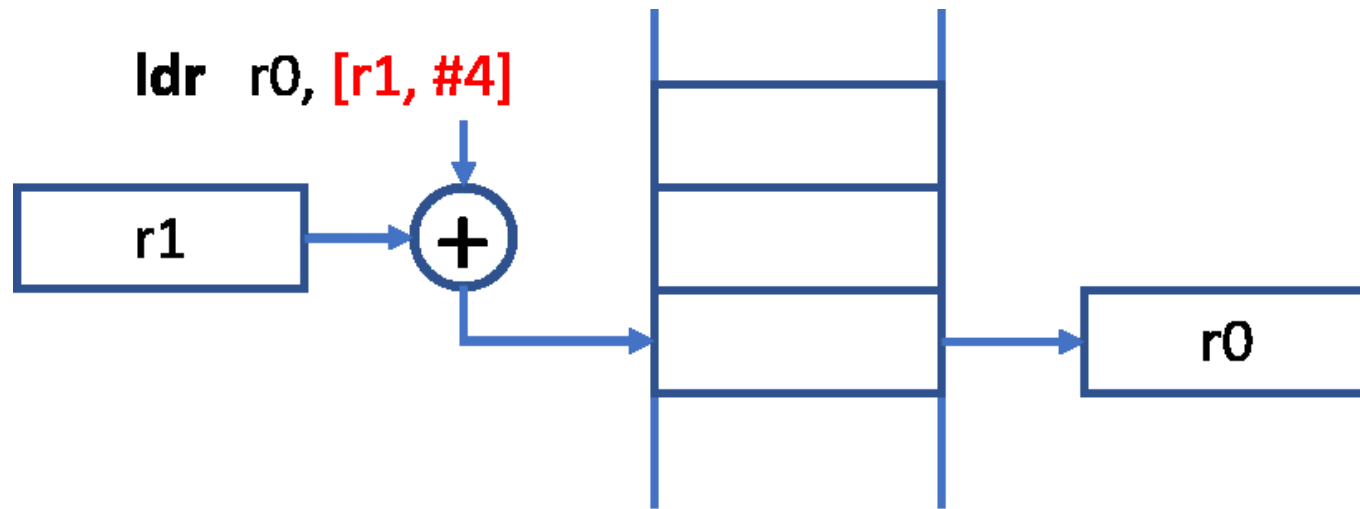
- Indirizzamento pre-indexed **senza writeback**: **ldr** r0, [r1, #4]
- Indirizzamento pre-indexed con calcolo **prima** di accedere e **writeback**:  
**ldr** r0, [r1, #4]!
- Indirizzamento post-indexed con calcolo **dopo** accesso e **writeback**:  
**ldr** r0, [r1], #4

## Indirizzamento: Esempi esplicativi - pre-indexed

- **ldr** r0, [r1, #4]

@ r0 = mem[r1 + 4]

@ r1 non modificato

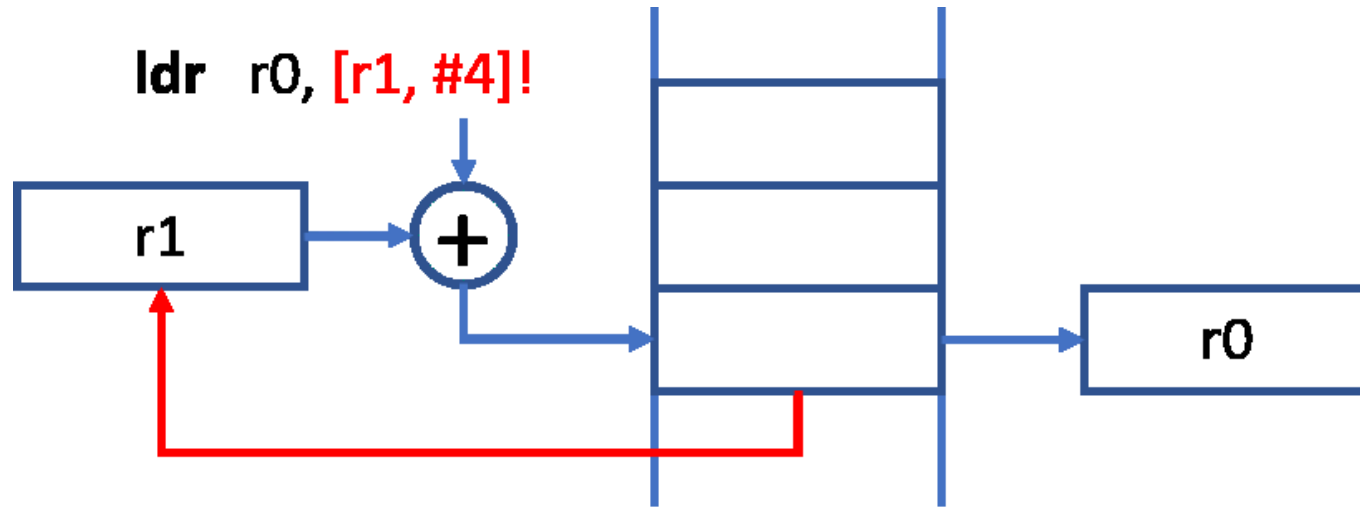


## Indirizzamento: Esempi esplicativi - pre-indexed writeback

- **ldr** r0, [r1, #4]!

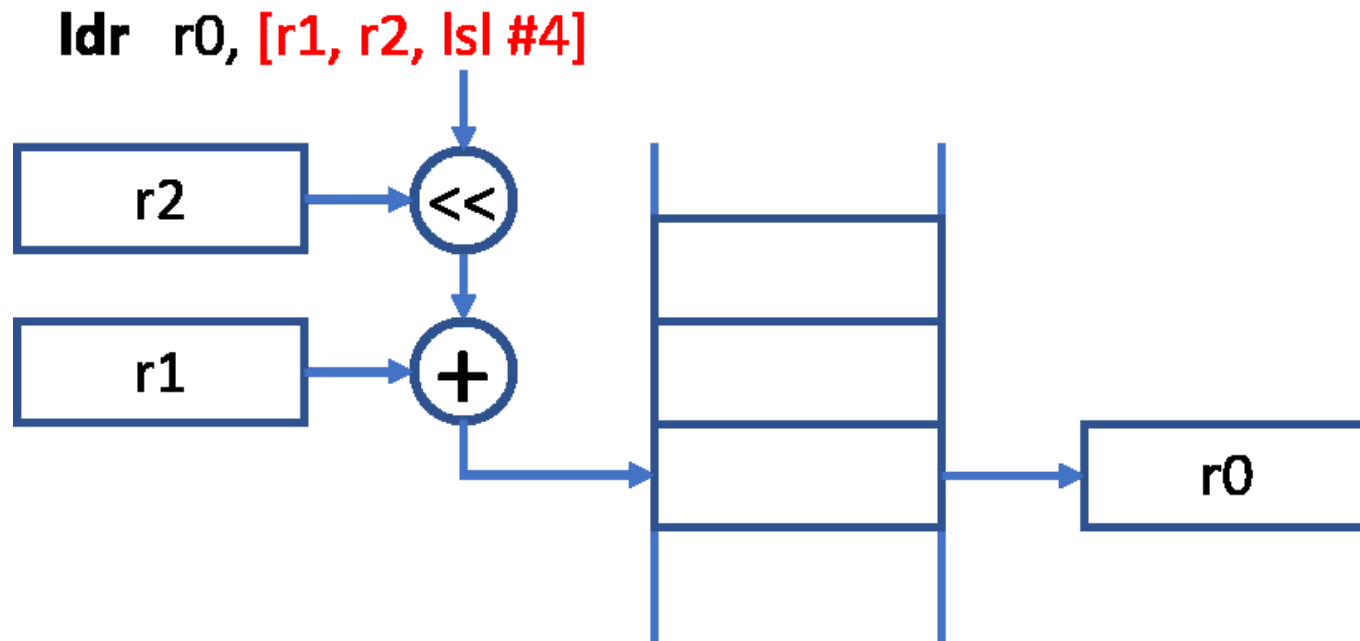
@ r0 = mem[r1 + 4]

@ r1 = r1 + 4



## Indirizzamento: Esempi esplicativi - pre-indexed shifted

- **ldr** r0, [r1, r2, lsl #4]      @ r0 = mem[r1 + r2 << 4]  
@ r1 non modificato



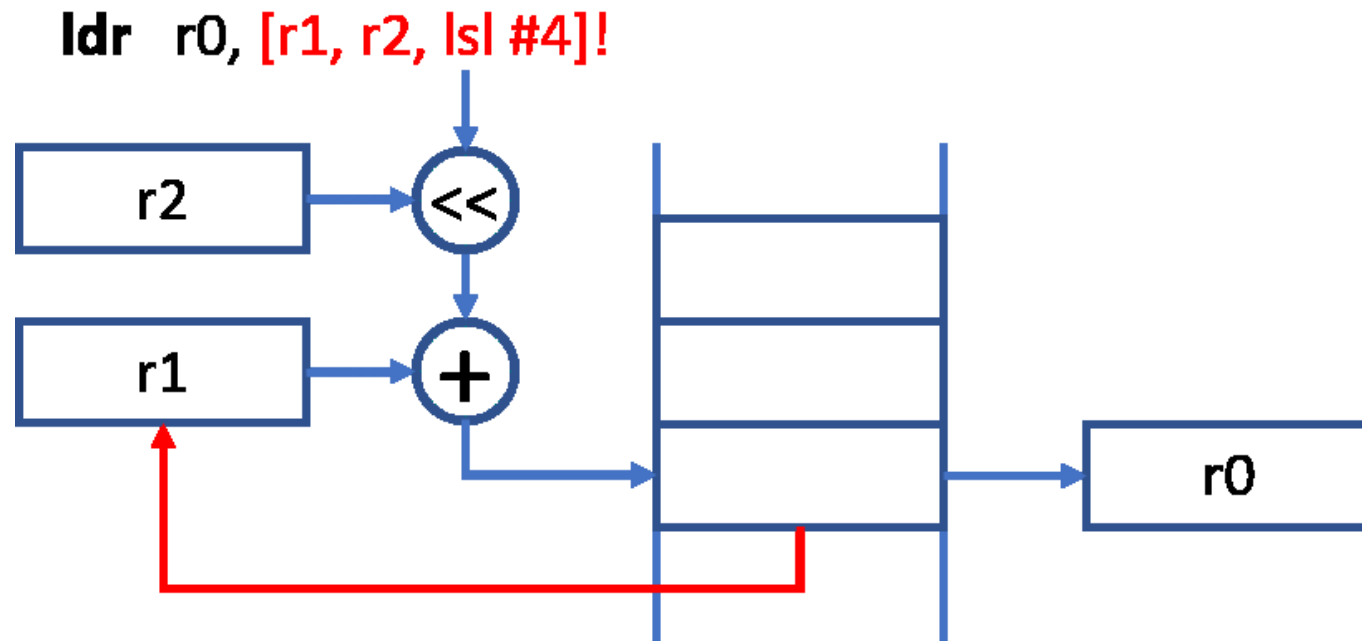


## Indirizzamento: Esempi esplicativi - pre-indexed shift writeback

- **ldr** r0, [r1, r2, lsl #4]!

@ r0 = mem[ r1 + r2 << 4 ]

@ r1 = r1 + r2 << 4



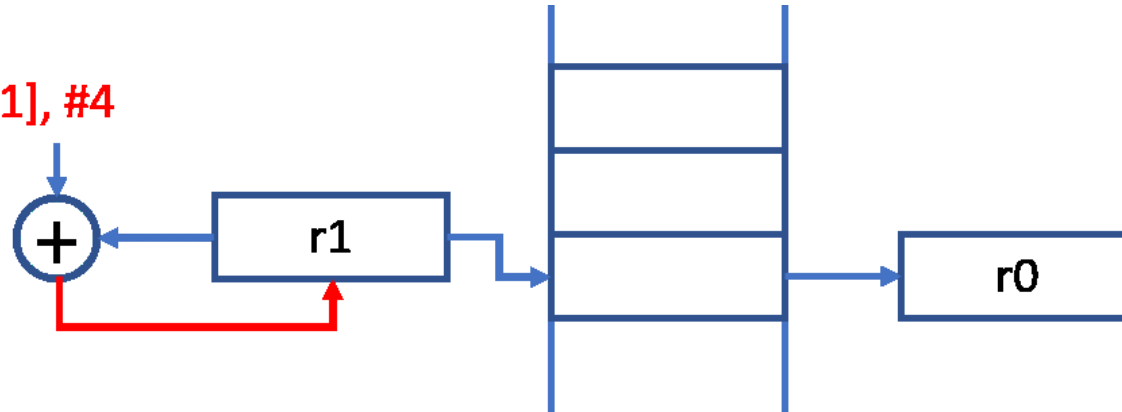
## Indirizzamento: Esempi esplicativi - post-indexed

- **ldr** r0, [r1], #4

@ r0 = mem[r1]

@ r1 = r1 + 4

**ldr** r0, [r1], #4



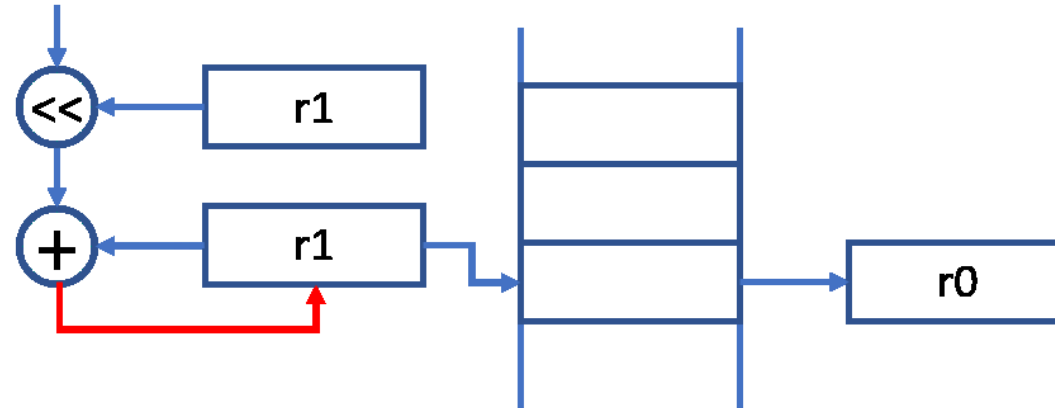
## Indirizzamento: Esempi esplicativi - post-indexed shift

- **ldr r0, [r1], r2, lsl #4**

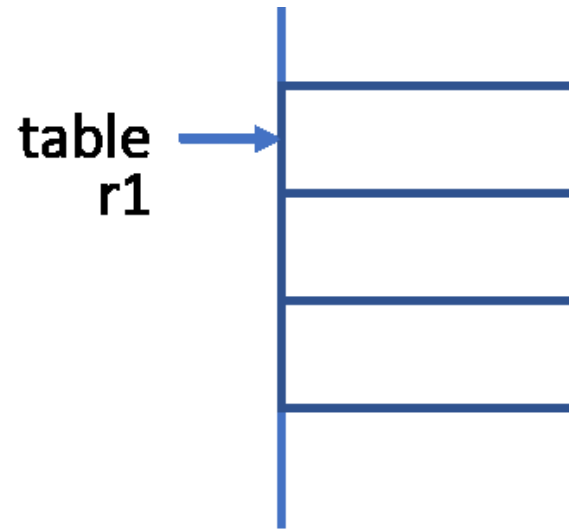
@ r0 = mem[r1]

@ r1 = r1 + r2 << 4

**ldr r0, [r1], r2, lsl #4**



## Indirizzamento: esempio applicazione



```
    adr r1, table
loop: ldr r0, [r1]
      adr r1, [r1, #4]
      @ operazioni su r0
      . . . . .
```

```
    adr r1, table
loop: ldr r0, [r1], #4
      @ operazioni su r0
      . . . . .
```

## Indirizzamento - Tirando le Somme

- Più potente di RISC-V
  - Registro indice (scalato)
  - Aggiornamento automatico del registro base
- Rispetto ad Intel
  - Ha in più l'aggiornamento automatico del registro base
  - Ha in meno la possibilità di usare contemporaneamente offset (displacement) ed indice
- Aggiornamento automatico del registro base: utile per scorrere array

# Istruzioni ARM

- Architettura RISC con **molte** istruzioni
  - Non proprio “Reduced”...
- Ampia documentazione su internet
  - e.g. <http://www.peter-cockerell.net/aalp/html/ch-3.html>
- Tutte le istruzioni permettono esecuzione condizionale
  - Suffisso `eq` (equal), `ne` (not equal), `hs` (higher or same), `lo` (lower), `mi` (minus), ...
    - Condizioni basate sui 4 flag di `apsr`: `n`, `z`, `c` e `v`
  - Flag aggiornati da
    - Istruzioni aritmetico/logiche con suffisso (opzionale) `s`
    - Da apposite istruzioni di confronto (come `cmp`)
- Nomi registri non hanno prefisso (`r0`, `r1`, ..., `r15`)
- Valori immediati (costanti) iniziano con “#”

# Istruzioni Aritmetiche e logiche

- `<opcode> [<cond>] [s] rd, rl, <r>`  
`<r> := #const | <reg> [, <sor> #const | <reg>]`
  - Come RISC-V, registro destinazione e 2 operandi
  - No argomenti in memoria
- Perché “<r>”?
  - Secondo argomento può essere immediato o registro
    - differenza da RISC-V: `add` invece che `add ed addi`
  - Se registro, può essere shiftato o rotato
    - `lsl (asl), lsr, asr, ror, rrx`
- Esempi:
  - **`add`** `r0, r1, #1`
  - **`adds`** `r0, r1, r2`
  - **`addeq`** `r0, r1, r2, lsl #2`
  - **`addeqs`** `r0, r1, r2, lsl r3`

## Istruzioni Più Comuni - 1

- `add / adc` (add with carry bit C di CPSR):  
**add** r0, r1, r2                      @ r0 = r1 + r2  
**adc** r0, r1, r2                      @ r0 = r1 + r2 + C
- `sub / sbc` (sub with carry bit C di CPSR)  
**sub** r0, r1, r2                      @ r0 = r1 - r2  
**sbc** r0, r1, r2                      @ r0 = r1 - r2 + C - 1
- `rsb / rsc` (reverse sub / reverse sub with carry)  
**rsb** r0, r1, r2                      @ r0 = r2 - r1  
**rsc** r0, r1, r2                      @ r0 = r2 - r1 + C - 1
- A che servono? Perché non usare `sub / sbc` con operandi invertiti?
  - Perché il secondo e terzo argomento di un'istruzione non sono equivalenti...
  - Terzo argomento: possibilità di fare shift / rotazione!!!



## Istruzioni Più Comuni - 2

- `and / orr / eor` : operazioni booleane bit a bit

**and** `r0, r1, r2`                      @ `r0 = r1 and r2`

**orr** `r0, r1, r2`                      @ `r0 = r1 or r2`

**eor** `r0, r1, r2`                      @ `r0 = r1 xor r2`

- `bic`

**bic** `r0, r1, r2`                      @ `r0 = r1 and not r2`

- `bic` (bit clear): calcola `r1 and not (<r2>)`, ovvero ogni 1 in `r2` mette a 0 il bit corrispondente in `r1`.

## Istruzioni Più Comuni - 3

- `mul`, `mla` & friends: varie forme di moltiplicazione

**mul** r0, r1, r2                      @ r0 = r1 \* r2

**mla** r0, r1, r2, r3                  @ r0 = r1 \* r2 + r3

- Istruzioni di shift e rotazione: tramite manipolazione del terzo argomento

**add** r0, r1, r2, `lsl #4`              @ r0 = r1 + r2 << 4

**add** r0, r1, r2, `lsl r4`                @ r0 = r1 + r2 << r4

- Nota: molti ARM core **NON** forniscono una istruzione di divisione tra interi

- L'operazione di divisione è fornita da codice:

```
int divide(int A, int B) {  
    int Q = 0; int R = A;  
    while( R >= B ) {  
        Q = Q + 1;  
        R = R - B;  
    }  
    return Q;  
}
```

## Istruzioni Più Comuni - 4

- Movimenti di registri:
  - `mov`  
`mov r0, r1`                      @ r0 = r1  
`mov r0, #21`                      @ r0 = 21
  - `mvn`: move not. Muove il complemento ad 1 di un registro  
`mvn r0, r1`                      @ r0 = not r1
- `b`: branch (anche salto condizionale, tramite suffisso...)

Branch incondizionato:

```
        b label  
        ...  
label:  ...
```

Branch condizionato:

```
        mov r0, #0  
loop:   ...  
        add r0, r0, #1  
        cmp r0, #10  
        bne loop
```

## Istruzioni Più Comuni - 5

- `bl`: branch and link (per invocazione di subroutine)
  - Salva l'indirizzo di ritorno nel link register `r14`

```
    bl sub      ; chiama sub
    cmp r1, #5 ; ritorna qui
    moveq r1, #0

sub:    ...
          ; codice funzione sub
          ...
    mov pc, lr ; return
```

- `bx`: solo su alcune CPU, equivalente a `mov r15, r`

## Istruzioni Più Comuni - 6

- `cmp`: setta flags come `sub`, ma senza risultato!
- Da usarsi prima di salti condizionali

```
if ((r0 == r1) && (r2 == r3)) r4++;
```

*; Encoding semplice*

```
cmp r0, r1
bne skip
cmp r2, r3
bne skip
add r4, r4, #1
```

skip: ...

*; Encoding piu' piccolo e veloce*

```
cmp r0, r1
cmpeq r2, r3
addeq r4, r4, #1
...
```

- Altre istruzioni simili: `tst` (esegue `and`), `teq` (`xor`), `cmn` (somma), ...

<code>tst r0, r1</code>	@ set cc a r0 and r1
<code>teq r0, r1</code>	@ set cc a r0 xor r1
<code>cmn r0, r1</code>	@ set cc a r0 + r1

## Istruzioni Più Comuni - 7

- `ldr / str`: load / store register

**ldr** r0, [r1]                      @ r0 = mem[r1]

**str** r0, [r1]                      @ mem[r1] = r0

- Valgono le regole per indirizzamento discusse in precedenza
- Esistono versioni per 32, 16, 8 bit

**ldr** r0, [r1] ; 32 bit    **ldrh** r0, [r1] ; 16 bit    **ldrb** r0, [r1] ; 8 bit

**str** r0, [r1] ; 32 bit    **strh** r0, [r1] ; 16 bit    **strb** r0, [r1] ; 8 bit

- `ldm / stm`: load / store multiple registers

**ldm** r0, {r1, r2, r3}              @ r1 = mem[r0]  
                                      @ r2 = mem[r0+4]  
                                      @ r3 = mem[r0+8]

**stm** r0, {r1, r2, r3}              @ mem[r0] = r1  
                                      @ mem[r0+4] = r2  
                                      @ mem[r0+8] = r3

- Consentono di trasferire grandi quantità di dati in modo efficiente
- Utilizzate in prologo/epilogo di funzioni per salvare/ripristinare i registri e l'indirizzo di ritorno

## Load / Store Multiple

- `ldm<mode> r [!], <register list> / stm<mode> r, <register list>`
- `<mode>` può essere
  - `ia` (increment after): registri salvati / recuperati nelle locazioni `r`, `r+4`, `r+8`, etc...
  - `ib` (increment before): registri salvati / recuperati nelle locazioni `r+4`, `r+8`, `r+12`, etc...
  - `da` (decrement after): registri salvati / recuperati nelle locazioni `r`, `r-4`, `r-8`, etc...
  - `db` (decrement before): registri salvati / recuperati nelle locazioni `r-4`, `r-8`, `r-12`, etc...
- Se `!` è specificato, `r` viene aggiornato di conseguenza
  - Nomi alternativi per suffissi: `ea` (empty ascending), `ed` (empty descending), `fa` (full ascending), `fd` (full descending)

# Load multiple

**LDM**<mode> [!] Rn, { <registers> }

IA: addr := Rn

IB: addr := Rn

DA: addr := Rn

DB: addr := Rn

foreach Ri in sorted(<registers>)

IB: addr := addr + 4

DB: addr := addr - 4

Ri := Mem[addr]

IA: addr := addr + 4

DA: addr := addr - 4

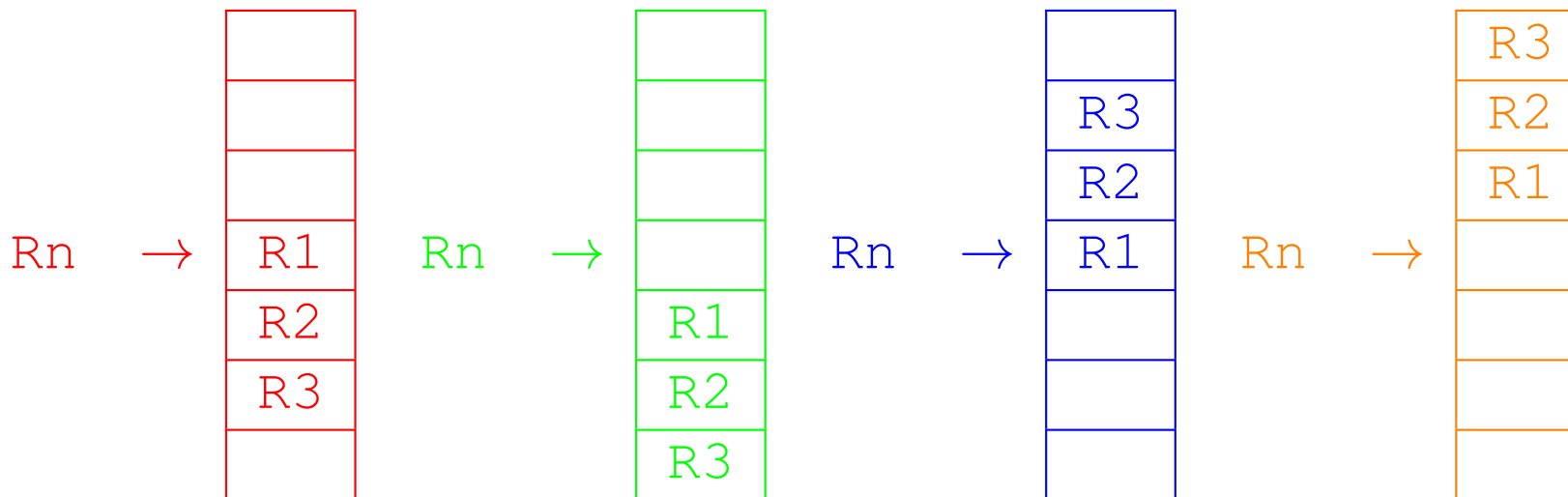
<!>: Rn := addr

IA : Increment After

IB : Increment Before

DA : Decrement After

DB : Decrement Before





## Load multiple - 2

- **ldmia** r0, {r1, r2, r3} or **ldmia** r0, {r1–r3}

r1 : 10  
r2 : 20  
r3 : 30  
r0 : 0x010

r0 →

addr	data
0x010	10
0x014	20
0x018	30
0x01c	40
0x020	50
0x024	60

- **ldmia** r0!, {r1, r2, r3} or **ldmia** r0!, {r1–r3}

r1 : 10  
r2 : 20  
r3 : 30  
r0 : 0x01c

r0 →

addr	data
0x010	10
0x014	20
0x018	30
0x01c	40
0x020	50
0x024	60

## Load multiple - 3

- **ldmib** r0!, {r1, r2, r3} or **ldmib** r0!, {r1–r3}

r1 : 20  
r2 : 30  
r3 : 40  
r0 : 0x01c

r0 →

addr	data
0x010	10
0x014	20
0x018	30
0x01c	40
0x020	50
0x024	60

- **ldmda** r0!, {r1, r2, r3} or **ldmda** r0!, {r1–r3}

r1 : 60  
r2 : 50  
r3 : 40  
r0 : 0x018

r0 →

addr	data
0x010	10
0x014	20
0x018	30
0x01c	40
0x020	50
0x024	60

## Load multiple - 4

- **ldmdb** r0!, {r1, r2, r3} or **ldmdb** r0!, {r1–r3}

r1 : 50  
r2 : 40  
r3 : 30  
r0 : 0x018

r0 →

addr	data
0x010	10
0x014	20
0x018	30
0x01c	40
0x020	50
0x024	60

## Esempio di uso di load multiple: copia blocchi memoria

- `r9`: indirizzo della sorgente
- `r10`: indirizzo della destinazione
- `r11`: indirizzo fine area della sorgente

```
loop:  ldmia r9!, {r0 - r7}  
       stmia r10!, {r0 - r7}  
       cmp r9, r11  
       bne loop
```

## Esempio

- Stringa C: array di caratteri terminato da 0
  - ASCII: caratteri codificati su byte
- Copia di una stringa:

```
void copia_stringa(char *d, const char *s)
{
    int i = 0;

    while ((d[i] = s[i]) != 0) {
        i += 1;
    }
}
```

- Esempio già visto per Assembly RISC-V ed Intel...
- Come fare con Assembly ARM?

## Accesso a Singoli Byte

- Ricordate? Necessità di copiare **byte**, non **parole**...
- RISC-V: load byte `lb` e store byte `sb` invece di `lw` e `sw`
- Intel: soluzione semplice (ed elegante? Dipende dai punti di vista!)
  - Registri composti da “sottoregistri” più piccoli
  - `al ... dl`: registri a 8 bit
  - `movb`: da memoria a `al .. dl` o viceversa
- E ARM?
- Load register (`ldr`) e store register (`str`) possono avere suffisso `b`, `h`, etc...
- Usiamo `ldrb` ed `strb`!

## Implementazione Assembly: Prologo

- I due parametri `d` ed `s` sono contenuti in `r0` ed `r1`
- Invece di usare un registro per il contatore `i`, incrementiamo direttamente `r0` ed `r1`
  - Non sono registri preservati...
  - Non è necessario salvarli sullo stack
- Si può usare indirizzamento post indexed!!!
- Non è necessario alcun prologo; possiamo cominciare col codice C

## Implementazione Assembly: Loop

- Ciclo `while`: copia `s[i]` in `d[i]`
  - Prima di tutto, carichiamolo in un registro temporaneo
    - Usiamo `r3`
  - Per fare questo, usiamo indirizzamento post indexed (base + indice \* scala, con scala = 1 e poi incrementa base)
  - Nessuna necessità di caricare l'indirizzo dell'`i`-esimo elemento in un registro, come si faceva per RISC-V

```
ldrb r3, [r1], #1
```

- Ora memorizziamo `r3` in `d[i]`

```
strb r3, [r0], #1
```



## Implementazione Assembly: Fine del Loop

- Bisogna ora controllare se  $s[i] == 0$ 
  - Se sì, fine del loop!

<b>cmp</b>	r3, #0	<i>@ confronta r3 con 0...</i>
<b>beq</b>	L2	<i>@ se sono uguali, salta a L2</i>
		<i>@ (esci dal loop!)</i>

- Se no, cicla...

<b>b</b>	copia_stringa
----------	---------------

- La label `L2` implementerà il ritorno al chiamante

## Implementazione Assembly: Fine

- Non abbiamo salvato nulla sullo stack: non c'è necessità di epilogo!
- Si può direttamente tornare al chiamante

```
L2: mov r15, r14
```

- Mettendo tutto assieme:

```
        .text  
        .globl copia_stringa  
copia_stringa:  
        ldrb r3, [r1], #1  
        strb r3, [r0], #1  
        cmp  r3, #0  
        beq  L2  
        b    copia_stringa  
L2:  
        mov  r15, r14
```

## Vediamo GCC...

```
        .text
        .global copia_stringa
copia_stringa:
        ldrb    r3, [r1]
        strb    r3, [r0]
        cmp     r3, #0
        bxeq    lr

L3:
        ldrb    r3, [r1, #1]!
        strb    r3, [r0, #1]!
        cmp     r3, #0
        bne     L3
        bx      lr
```

# Esempio

```
for(int i = 0; i < 10; i++) {  
    a[i] = 0;  
}
```

```
mov r0, #0 ; r0 = 0  
adr r2, a ; r2 = &a  
mov r1, #0 ; i = 0  
loop: cmp r1, #10 ; i < 10  
bge end ; se >=, end  
str r0, [r2, r1, lsl #2]  
; mem[r2 + r1 << 2] = 0  
add r1, r1, #1 ; i ++  
b loop  
end:
```

```
int gcd (int i, int j) {  
    while (i != j) {  
        if (i > j)  
            i -= j;  
        else  
            j -= i;  
    }  
}
```

```
loop: cmp r0, r1  
subgt r0, r0, r1  
sublt r1, r1, r0  
bne loop  
bx lr
```

# Esempio di passaggio parametri con lo stack

```
int caller() {  
    int sum = f1(1, 2, 3, 4, 5, 6);  
    return sum;  
}  
  
int f1(int a1, int a2, int a3, int a4,  
       int a5, int a6) {  
    return a1+a2+a3+a4+a5+a6;  
}
```

```
f1:  
    add    r0, r0, r1  
    add    r0, r0, r2  
    add    r0, r0, r3  
    ldr    r3, [sp]  
    add    r0, r0, r3  
    ldr    r3, [sp, #4]  
    add    r0, r0, r3  
    bx     lr  
  
caller:  
    str    lr, [sp, #-4]!  
    sub    sp, sp, #12  
    mov    r3, #6  
    str    r3, [sp, #4]  
    mov    r3, #5  
    str    r3, [sp]  
    mov    r3, #4  
    mov    r2, #3  
    mov    r1, #2  
    mov    r0, #1  
    bl     f1  
    add    sp, sp, #12  
    ldr    lr, [sp], #4  
    bx     lr
```

# Esempio di divisione come shift e sottrazione per interi positivi

```
int divide(int A, int B) {  
    int Q = 0; int R = A;  
    while( R >= B ) {  
        Q = Q + 1;  
        R = R - B;  
    }  
    return Q;  
}
```

```
divide :  
    mov     r3, r0  
    cmp     r0, r1  
    blt     .L4  
    mov     r0, #0  
.L3 :  
    add     r0, r0, #1  
    sub     r3, r3, r1  
    cmp     r1, r3  
    ble     .L3  
    bx      lr  
.L4 :  
    mov     r0, #0  
    bx      lr
```

# Versione sofisticata di divisione tra interi positivi

<http://www.tofla.iconbar.com/tofla/arm/arm02/index.htm>

```
div:
; divs r0, r1, r2
cmp    r2, #0
beq    divide_end ; check for divide by zero!
mov    r0, #0      ; clear r0 to accumulate result
mov    r3, #1      ; set bit 0 in r3, which will be shifted left then right
start:
cmp    r2, r1
movls  r2, r2, lsl #1
movls  r3, r3, lsl #1
bls    start      ; shift r2 left until it is about to be bigger than r1,
                    ; shift r3 left in parallel in order to flag how far we
                    ; have to go

next:
cmp    r1, r2      ; carry set if r1 > r2 (don't ask why)
subcs  r1, r1, r2   ; subtract r2 from r1 if this would give a positive
                    ; answer
addcs  r0, r0, r3   ; and add the current bit in r3 to the accumulating
                    ; answer in r0
movs   r3, r3, lsr #1 ; Shift r3 right into carry flag
movcc  r2, r2, lsr #1 ; and if bit 0 of r3 was zero, also shift r2 right
bcc    next        ; If carry not clear, r3 is shifted back to where
                    ; it started, and we can end

divide_end:
; r1 holds the remainder, if any, r2 has returned to the value it held on
; entry to the routine, r0 holds the result and r3 holds zero. Both zero
; and carry flags are set.
bx    lr
```