Programmazione e Architetture (Modulo B)

Lezione 6
Architettura ARM (32 e 64 bit)

Luca Manzoni

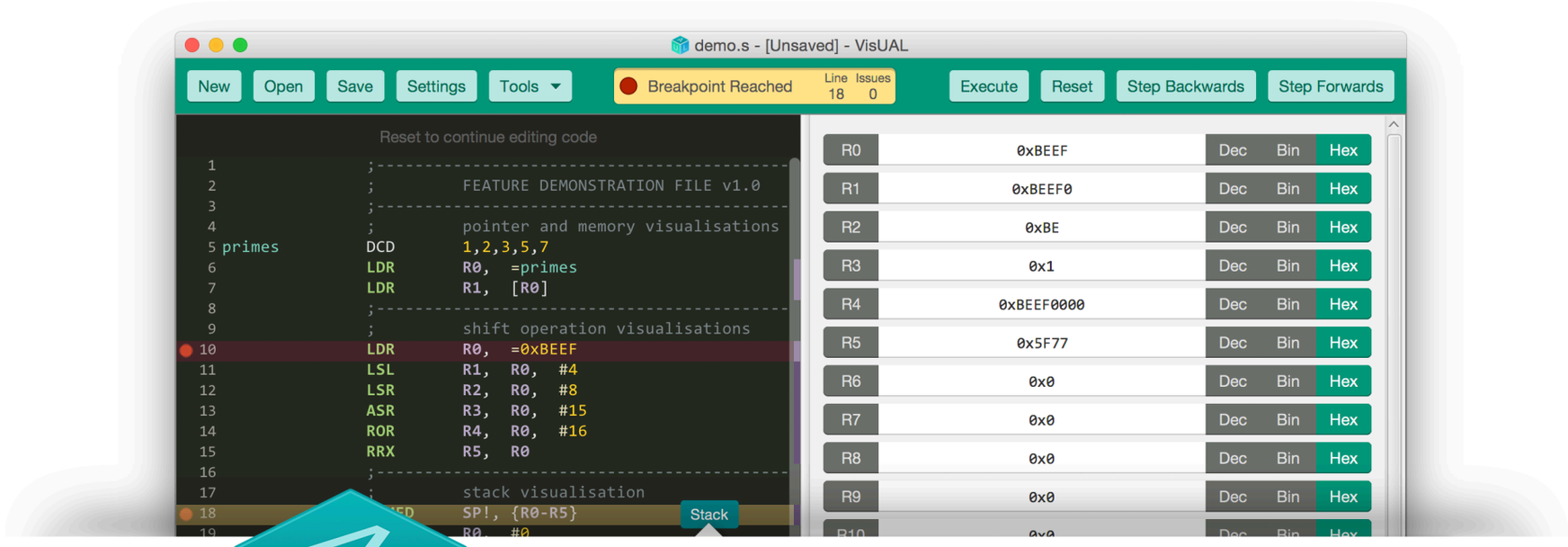
Alcuni cambiamenti

Da 64 a 32 bit

- Per scopi didattici e per rendere possibile a tutti di seguire da remoto si è scelto di cambiare l'architettura di esempio
- Passiamo da ARM a 64 bit a ARM a 32 bit…
- …ma parliamo anche delle loro differenze
- *Motivazione*: gli strumenti per simulare ARM a 32 bit sono più avanzati e facili da installare
- Per quello che vediamo noi la differenza è comunque limitata

Software

VisUAL



VisUAL

A highly visual ARM emulator

- Il software che utilizzeremo è VisUAL: https://salmanarif.bitbucket.io/visual/index.html
- Supporta un sottoinsieme delle istruzioni ARM a 32 bit: https://salmanarif.bitbucket.io/visual/supported_instructions.html
- Disponibile per Windows, macOS e Linux
- Permette di eseguire le istruzioni passo passo e visualizzare il contenuto di registri e memoria

La storia di ARM

Da Acorn Risc Machine a ARM

- L’architettura ARM nasce come prodotto della Acorn Computers, società britannica che aveva prodotto il BBC Micro nel 1981

- Dopo il BBC Micro e l’Acorn Electron, la Acorn decise di sviluppare una sua architettura di processori con il progetto *Acorn RISC Machine* (ARM). Nel 1985 l’ARM1 fu il primo processore prodotto

- Nel 1990 la progettazione dei processori fu separata in una diversa compagnia fondata da Acorn, Apple e VLSI Technology

- ARM attualmente non produce direttamente processori ma li progetta e vende i diritti di proprietà intellettuale

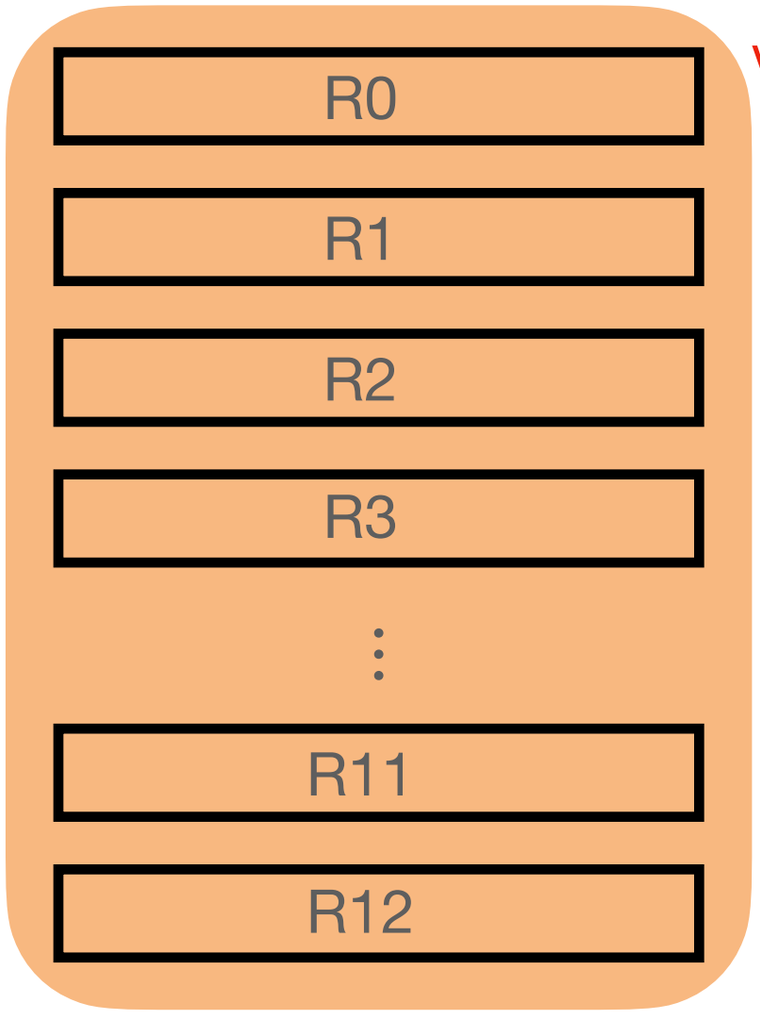
- Nel tempo il significato di ARM è passato a “Advanced Risc Machine” per poi essere, semplicemente, ARM

Arm 32 bit

Arm-v7a

- 13 registri per l’uso generale, ognuno di 32 bit
- 1 registro per lo stack pointer
- 1 link register (per le chiamate a subroutine)
- 1 program counter
- 1 program status register (PSR)
- Ogni istruzione richiede 32 bit
- Permette una modalità “ridotta” (Thumb mode) in cui ogni istruzione è a 16 bit (che non vedremo)

Arm: registri



Viene salvato l'indirizzo dello stack (quello delle chiamate a funzione)

R13

Link register viene usato per salvare un indirizzo per alcune operazioni

LR

Contiene l'indirizzo dell'istruzione da eseguire

PC

Contiene diversi bit che vengono impostati come risultato di comparazioni, operazioni aritmetiche, etc

PSR

Nessuno di questi registri ha un utilizzo specifico, possiamo usarli come vogliamo*

* esistono delle convenzioni quando si lavora con altro codice, ma sono solo convenzioni

Arm 64 bit

Arm-v8a

- Il passaggio ai 64 bit è stato l’occasione per rendere più consistenti le istruzioni (ma è ancora riconoscibile come ARM)
- 31 registri per uso generale, ognuno da 64 bit (accessibili anche come registri a 32 bit)
- Registro zero (costante 0) o stack pointer (dipende dall’istruzione)
- Program Counter non più direttamente accessibile
- Le istruzioni rimangono codificate in 32 bit

CISC vs RISC

CISC

Complex Instruction Set Computers

- Nell’approccio CISC si vuole completare un’operazione nel minor numero di operazioni in assembly
- Questo significa che, per esempio, è possibile nella stessa operazione accedere alla memoria, sommare un valore, scrivere il valore in memoria
- Generalmente le istruzioni richiedono più di un ciclo di clock per essere completate
- Generalmente le istruzioni sono di lunghezza variabile, sono molte e più difficili da decodificare

RISC

Reduced Instruction Set Computers

- Nell’approccio RISC si vuole mantenere basso il numero di istruzioni
- Ogni istruzione compie una cosa sola. Si hanno istruzioni separate per caricare dalla memoria in un registro, sommare un valore e salvare il risultato in memoria
- Si tende ad avere istruzioni che completano in pochi cicli di clock
- Le istruzioni sono di lunghezza fissata, rendendo la decodifica più semplice
- In realtà le architetture RISC sono andate complicandosi e quelle CISC hanno preso molte idee dai RISC

Istruzioni ARM

Assembly

Cosa è l'assembly?

- Le istruzioni, come abbiamo visto, sono codificate come sequenze di bit
- Per semplicità associamo dei nomi più facili da ricorda alle istruzioni a seconda di quello che le istruzioni svolgono
- Il passaggio da una rappresentazione come "ADD R0, R0, R1" e la rappresentazione come sequenza di bit "1110 1100 0000 0000 0000 0000 0000 0001" può essere svolto in modo semplice
- Questa rappresentazione testuale che ha una conversione diretta (o, comunque, semplice) alle istruzioni in codice macchina è detta **assembly**

MOV

Spostare valori tra registri

L’istruzione MOV permette di copiare un valore da un registro a un altro
o caricare un valore numerico (chiamato immediato) nel registro

Alcuni esempi di istruzione MOV:

```
MOV R0, #45
```

Mette il valore 45 nel registro R0

```
MOV R0, R2
```

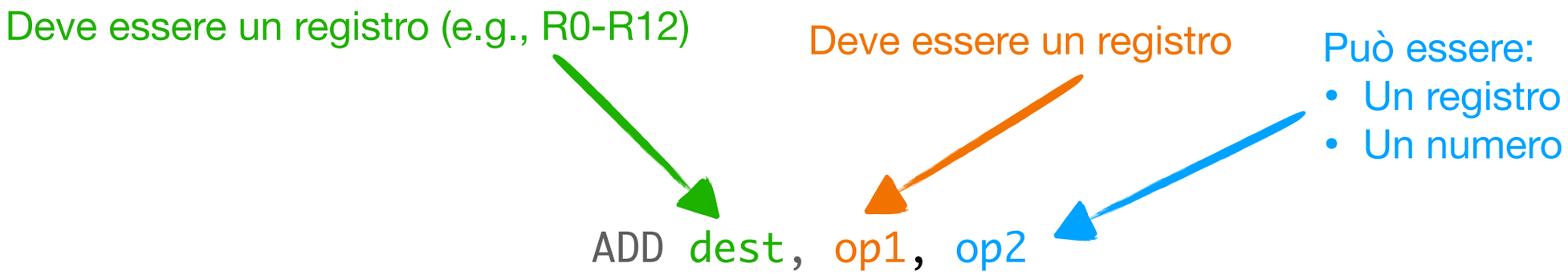
Copia il valore contenuto nel registro R2 nel registro R0

Esercizio: scambiare il valore del registro R1 e R2 usando R0 come registro di appoggio

ADD

Sommare Valori

L’istruzione ADD somma i valori contenuti in due registri o il valore contenuto in un registro e un numero.
Il risultato viene poi salvato in un registro



Alcuni esempi di istruzione ADD:

```
ADD R0, R0, #1
```

Incrementa il valore contenuto in R0 di 1

```
ADD R0, R1, R2
```

Mette in R0 il risultato della somma dei valori contenuti in R1 e R2

Esercizio: Mettere in R1 un valore a scelta. Usando istruzioni ADD mettere in R0 il triplo del valore contenuto in R1

SUB e altre istruzioni

ADC, SBC, RSB, RSC, AND, ORR, EOR, BIC, ORN

L’istruzione SUB sottrae i valori contenuti in due registri o il valore contenuto in un registro e un numero.
Il risultato viene poi salvato in un registro

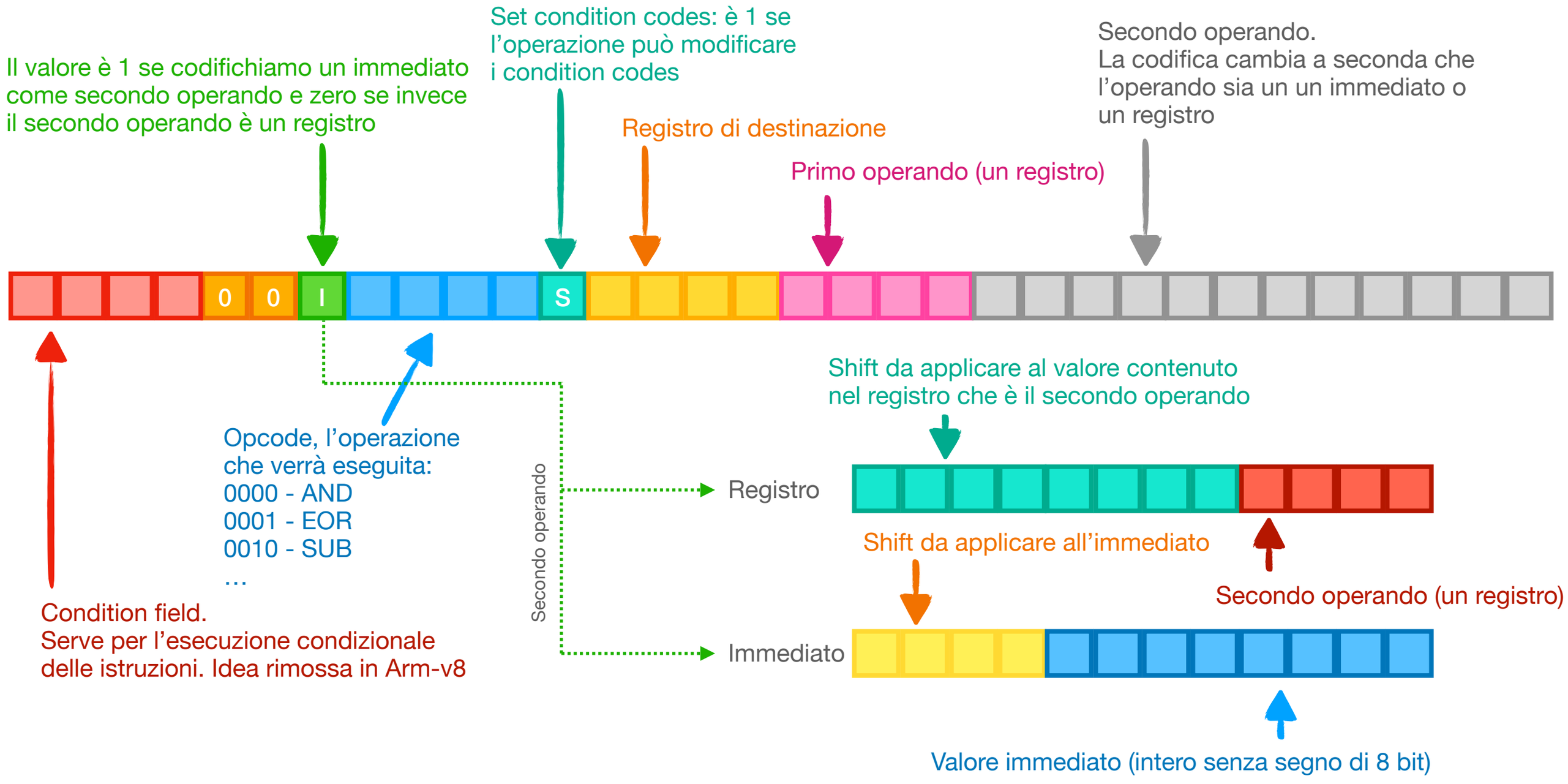
Altre istruzioni con formato simile:

`ADC dest, op1, op2`	ADD con bit di carry a 1
`SBC dest, op1, op2`	SUB con bit di carry a 1
`RSB dest, op1, op2`	Reverse SUB
`RSC dest, op1, op2`	Reverse SUB con carry a 1
`ORR dest, op1, op2`	OR dei bit di op1 e op2
`EOR dest, op1, op2`	XOR dei bit di op1 e op2
`BIC dest, op1, op2`	Calcola op1 AND NOT op2
`ORN dest, op1, op2`	Calcola op1 OR NOT op2

Codifica delle istruzioni

32 bit per istruzione

Abbiamo detto che ogni istruzione è codificata in 32 bit, vediamo un esempio per le istruzioni cosiddette di “Data Processing” (e.g., ADD, SUB, etc.)



Effettuare scelte

Istruzioni di salto (jump o branch)

- Fino ad ora abbiamo visto come fare operazioni aritmetiche o logiche
- Non possiamo ancora alterare il flusso di esecuzione delle istruzioni
- Per fare questo abbiamo le istruzioni di salto (o branch) che cambiano il valore del program counter. Distinguiamo due categorie:
  - Salto **non condizionato** avviene sempre
  - Salto **condizionato** avviene solo se alcune condizioni sono soddisfatte (si utilizza il program status register)

B, BL ed etichette

Branch e labels

L’istruzione di branch ci permette di saltare ad un’altra istruzione nel codice. Per indicare a quale istruzione dobbiamo saltare usiamo, in assembly, della label.

Deve essere una etichetta
(un nome che precede una istruzione)

```
B label
```

Branch with link
Funziona come branch ma salva il valore del PC nel link register

```
BL label
```

etichetta

```
inizio ADD R0, R0, #1
       B inizio
```

Continuerà a sommare 1 al valore contenuto nel registro R0

Senza i salti condizionati e usando solo istruzioni di branch otterremmo solo cicli infiniti

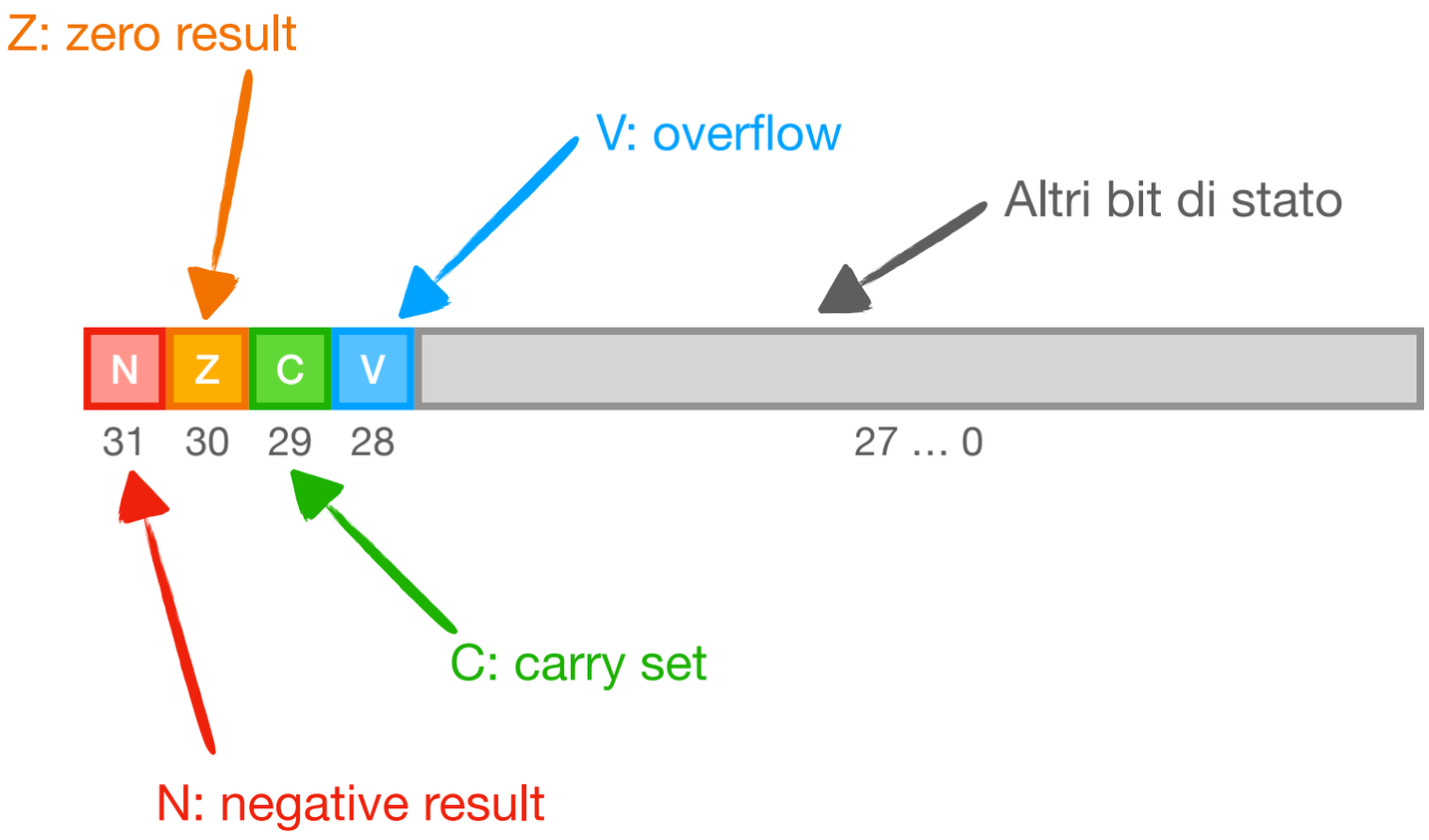
Codifica:

Link bit (distingue Branch da Branch with link)

Offset: di quanto modificare il program counter (questo valore viene calcolato dall’assembler)

Program Status Register (PSR)

Alcuni dei bit che ci interessano

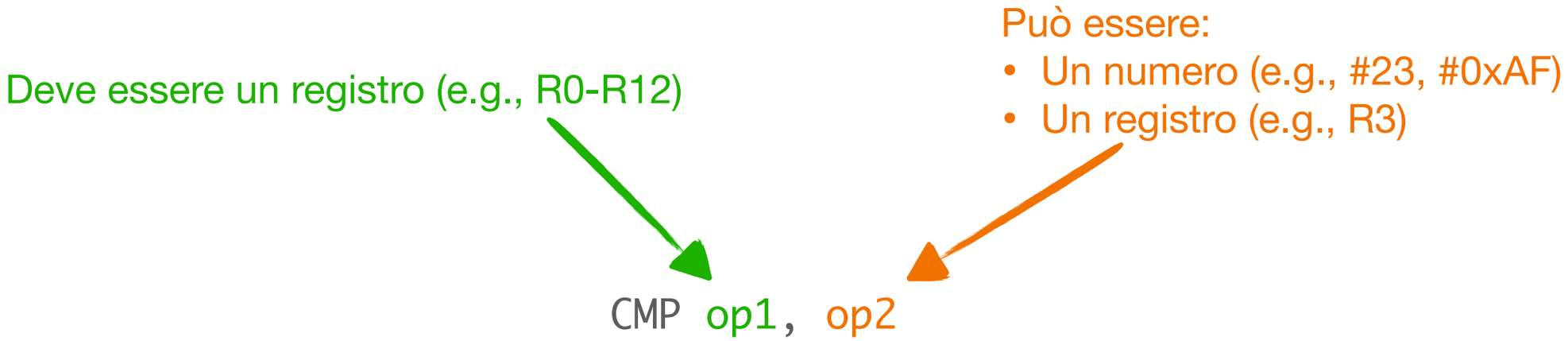


Questi bit sono aggiornati quando facciamo una comparazione e diverse configurazioni di questi bit ci forniscono informazioni sul risultato della comparazione

CMP

Comparare valori

L’istruzione CMP ci permette di comprare il valore contenuto in un registro con il valore contenuto in un altro registro o con un immediato. Il risultato della comparazione altererà il PSR



Alcuni esempi di istruzione CMP:

```
CMP R0, #45
```

Compara il valore contenuto nel registro R0 con il valore 45

```
CMP R0, R2
```

Compara il valore contenuto nel registro R0 con quello contenuto nel registro R1

Una operazione CMP cambia solo i valori contenuti nel PSR

Branch condizionale

BEQ

L’istruzione di branch condizionale ci permette di saltare ad un’altra istruzione nel codice.
Con BEQ questo accade solo se l’ultima comparazione (con CMP) era tra due valori identici

Deve essere una etichetta
(un nome che precede una istruzione)

```
BEQ label
```

etichette

```
inizio MOV R0, #10
       CMP R0, #0
       BEQ fine
       SUB R0, R0, #1
       BL inizio
fine
```

Sottrae uno al valore contenuto in R0
Continua a eseguire l’operazione a meno che
il valore in R0 non sia 0

Codifica:

Cambiano solo i bit nel condition field
(rispetto al normale branch)

Altri tipi di branch condizionale

BNE, BHS, BLO, …

A seconda dei bit nel condition field si catturano diverse condizioni, corrispondenti a diverse istruzioni in Assembly

	`BNE label`	I due valori comparati erano diversi
Senza segno	`BHS label`	Il primo valore era ≥ del secondo (considerati senza segno)
	`BLO label`	Il primo valore era < del secondo (considerati senza segno)
	`BHI label`	Il primo valore era > del secondo (considerati senza segno)
	`BLS label`	Il primo valore era ≤ del secondo (considerati senza segno)
Con segno	`BGE label`	Il primo valore era ≥ del secondo (considerati con segno)
	`BLT label`	Il primo valore era < del secondo (considerati con segno)
	`BHT label`	Il primo valore era > del secondo (considerati con segno)
	`BLE label`	Il primo valore era ≤ del secondo (considerati con segno)

Architetture Load-Store

Accedere alla memoria di lavoro

- Se notate nessuna delle operazioni precedenti utilizza direttamente sulla memoria di lavoro
- Sono solo operazioni tra registri
- Se vogliamo fare operazioni su valori salvati nella memoria di lavoro dobbiamo prima copiare il valore in un registro
- ARM utilizza operazioni esplicite per interagire con la memoria:
  - LOAD per caricare dalla memoria in un registro
  - STORE per salvare il valore di un registro in memoria

Architetture Load-Store

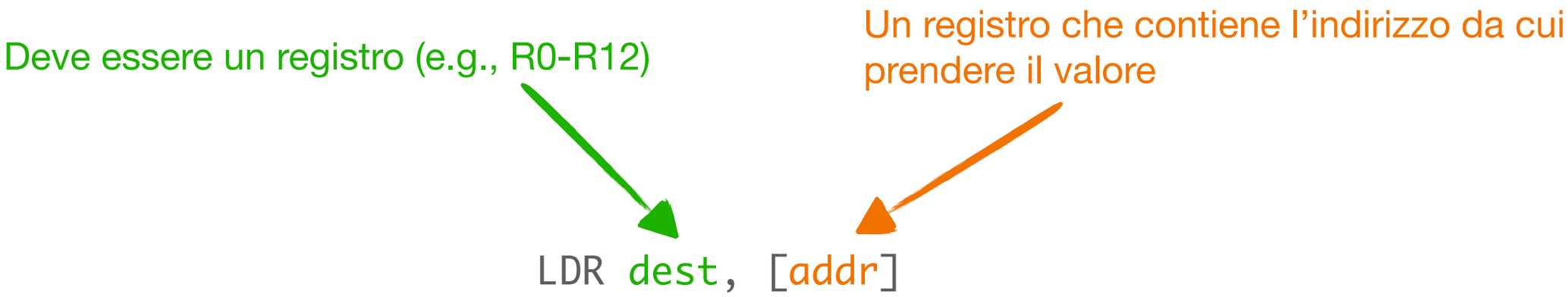
Indirizzamento

- Per accedere alla memoria ci serve un indirizzo
- Se codificassimo l'indirizzo direttamente nell'istruzione potremmo accedere solo a una frazione della memoria (e.g., se avessimo 24 bit "liberi" per l'indirizzo potremmo accedere solo a 2^{24} diverse locazioni di memoria, solitamente byte)
- Di solito si salva l'indirizzo di memoria in un registro (quindi 32 bit possibili) e si accedere all'indirizzo indicato da quel registro:
  - e.g., "Leggi in R0 il valore contenuto all'indirizzo di memoria contenuto in R1"

LOAD

Caricare valori dalla memoria

L’istruzione LDR ci permette di caricare in un registro un valore in memoria
Dato che le operazioni come ADD, SUB, … lavorano solo su registri questo è essenziale



Alcuni esempi di istruzione LDR:

```
LDR R0, [R1]
```

Carica in R0 il valore contenuto all’indirizzo di memoria che è il valore di R1

```
MOV R0,#0x100
LDR R1, [R0]
```

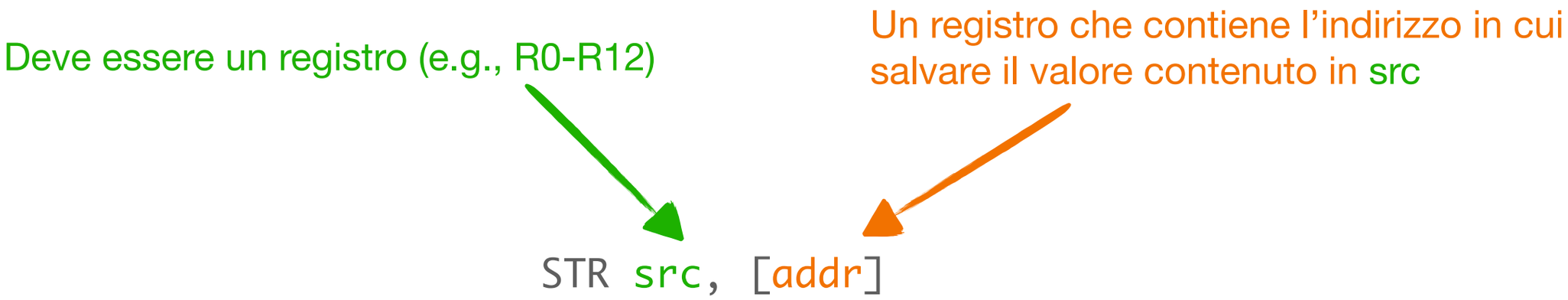
Carica in R1 il valore contenuto nell’indirizzo di memoria 0x100 (indicato in esadecimale)

Esercizio: Caricare in R1 il valore contenuto all’indirizzo di memoria 0x200

STORE

Salvare i valori in memoria

L’istruzione STR ci permette di salvare in memoria il valore contenuto in un registro.



Alcuni esempi di istruzione STR:

```
SDR R0, [R1]
```

Salva il valore di R0 nell’indirizzo di memoria che è il valore di R1

```
MOV R0,#0x100
STR R1, [R0]
```

Salva all’indirizzo 0x100 il valore contenuto nel registro R1

Esercizio: Salvare all’indirizzo di memoria 0x200 il risultato di 12+4

Load e Store con offset

Accedere ad array

- È comune dover accedere a valori contigui in memoria
- Per esempio i valori in un array sono salvati in modo contiguo in memoria
- La struttura degli indirizzi è quindi un “indirizzo di base” (primo elemento dell’array) e un offset (di quando incrementare l’indirizzo rispetto alla base)
- Questo è direttamente supportato dalle istruzioni LDR e STR
- Supportano anche il pre- e post-incremento dell’indirizzo di base del valore indicato dall’offset

LOAD

Questa volta con offset

L'offset può essere:
- Un registro (in quel caso viene considerato come valore il contenuto del registro)
- Un immediato (che rappresenta direttamente il valore dell'offset)

`LDR dest, [addr, offset]` Accede all'indirizzo addr + offset

`LDR dest, [addr, offset]!` **Pre-incremento**: somma offset a addr, salvando il valore in addr, e accede a addr (dopo che il suo valore è stato incrementato di addr)

`LDR dest, [addr], offset` **Post-incremento**: accede a addr (prima che il suo valore sia incrementato). Somma addr e offset e salva il suo valore in addr.

STORE

Questa volta con offset

L'offset può essere:
- Un registro (in quel caso viene considerato come valore il contenuto del registro)
- Un immediato (che rappresenta direttamente il valore dell'offset)

`STR dest, [addr, offset]` Accede all'indirizzo addr + offset

`STR dest, [addr, offset]!` **Pre-incremento**: somma offset a addr, salvando il valore in addr, e accede a addr (dopo che il suo valore è stato incrementato di addr)

`STR dest, [addr], offset` **Post-incremento**: accede a addr (prima che il suo valore sia incrementato). Somma addr e offset e salva il suo valore in addr.

Accesso allineato

E possibili errori con LDR e STR

- Un possibile problema con le operazioni di LOAD e STORE è che gli accessi in memoria devono essere **allineati**
- Cosa significa? La memoria viene letta a gruppi di byte contigui (nel nostro caso 4 byte o 32 bit) ma:
  - Gli accessi sono più efficienti se si legge da multipli di 4 byte (indirizzi allineati)
  - Accessi non allineati non sono concessi (in ARM). Altre architetture li concedono, ma possono essere più lenti, richiedendo due accessi allineati in memoria
- Ergo, potete accedere solo ad indirizzi che sono multipli di 4

Esercizio

Fare il lavoro del compilatore

Facciamo il lavoro di un compilatore: proviamo a convertire questo ciclo in del codice assembly in modo “plausibile”

```
int x = 5;
int y = 10;
while (y > 0) {
    x = x + y + 3;
    y = y - 1;
}
```

Due varianti dell’esercizio:

- x e y sono valori nei registri (e.g., R0 e R1)
- x e y sono da salvare alle locazioni di memoria 0x100 e 0x104

Notate che dobbiamo spezzare questa operazione di somma in due operazioni distinte!