Sistemi a Microcontrollore

1. Microcontrollori

Anno Accademico 2020/2021

Indice

- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

Indice

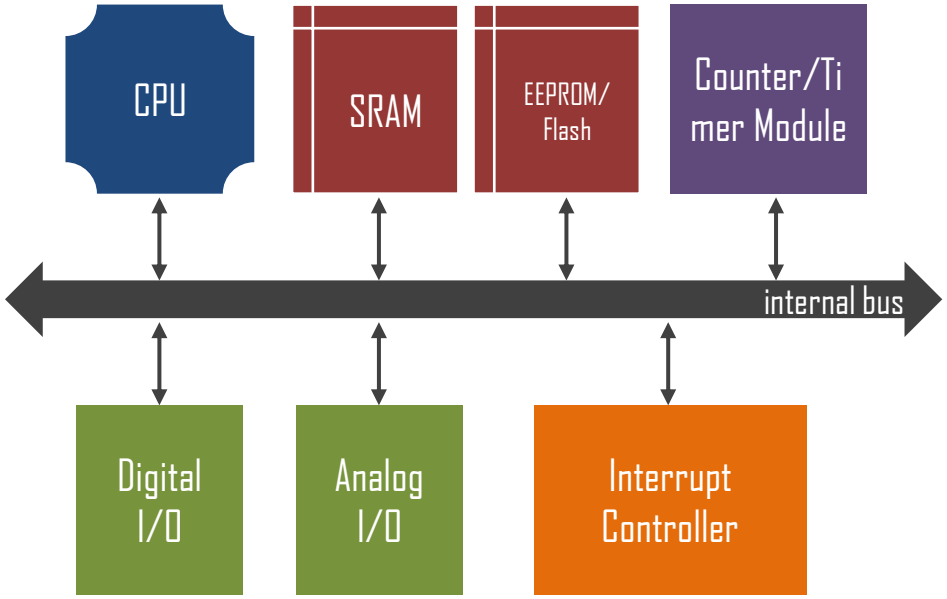
- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

Cosa è un Microcontrollore

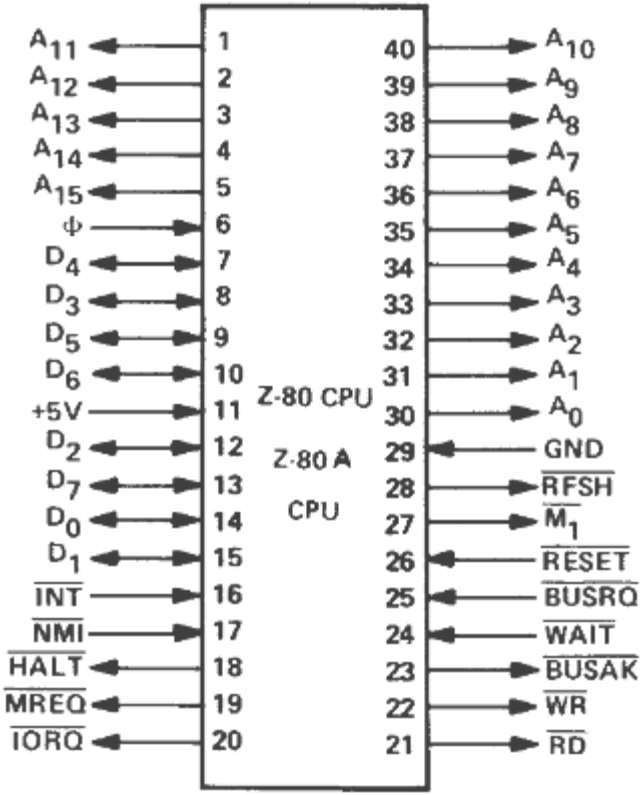
- Un microprocessore è un **processore integrato in un unico chip**
- Se in un microprocessore oltre al processore anche i circuiti di supporto, la memoria e le periferiche di input/output sono integrate in un unico chip, si parla di **microcontrollore**

Componenti di un Sistema a Microcontrollore

- Componenti di un **Sistema a Microcontrollore**
  - Processore
  - Memoria
  - I/O Digitali
  - I/O Analogici
  - Interrupt
  - Timer
  - Altro (Watchdog, etc.)

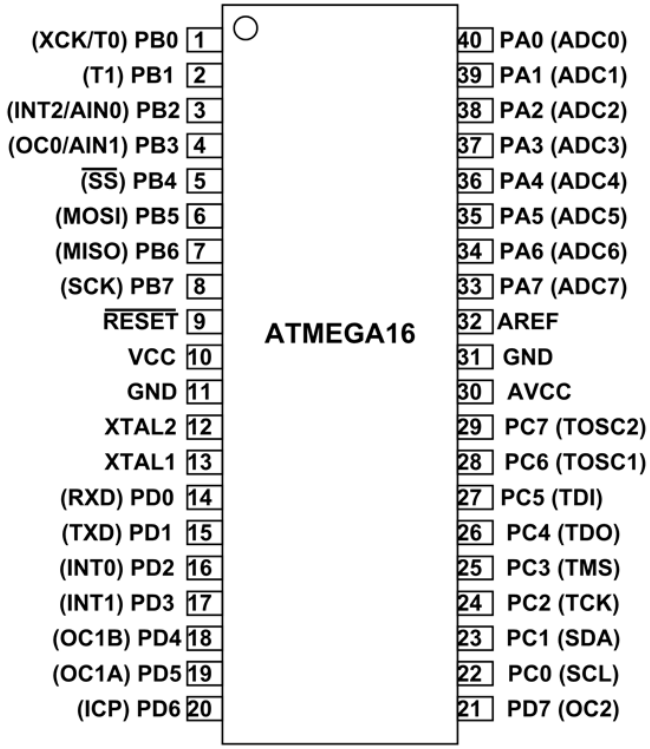


Cosa è un Microcontrollore

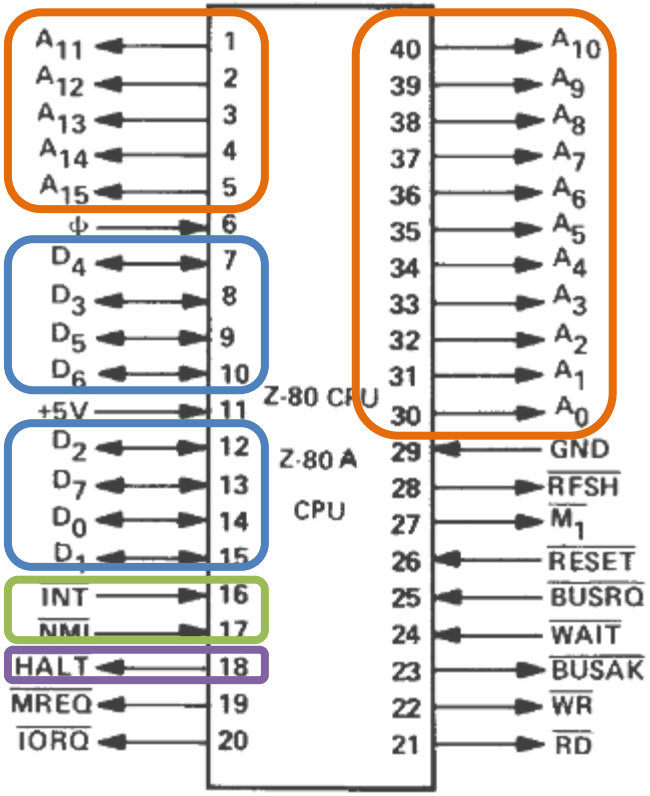


Microprocessore (MPU)
Zilog Z-80

Microcontrollore (MCU)
Atmel AVR
Atmega16



Cosa è un Microcontrollore



Microprocessore
(MPU)
Zilog Z-80

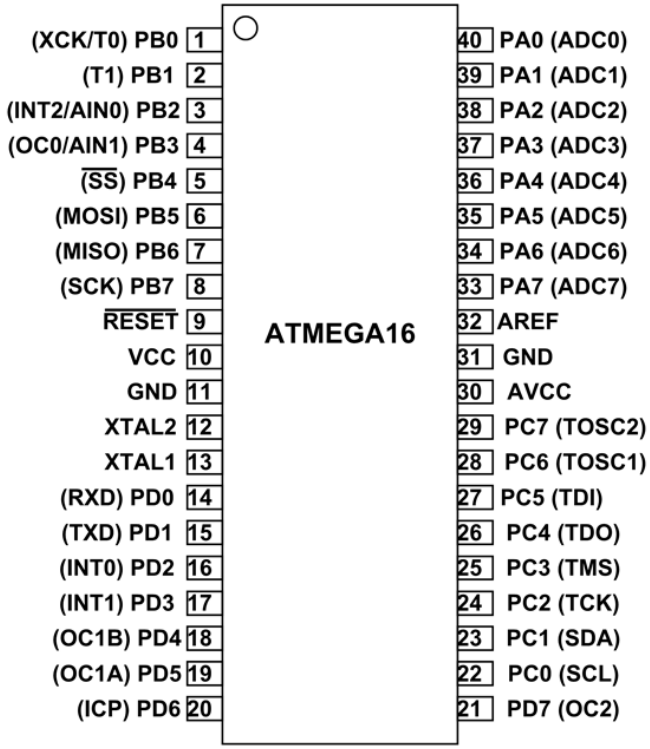
indirizzi

dati

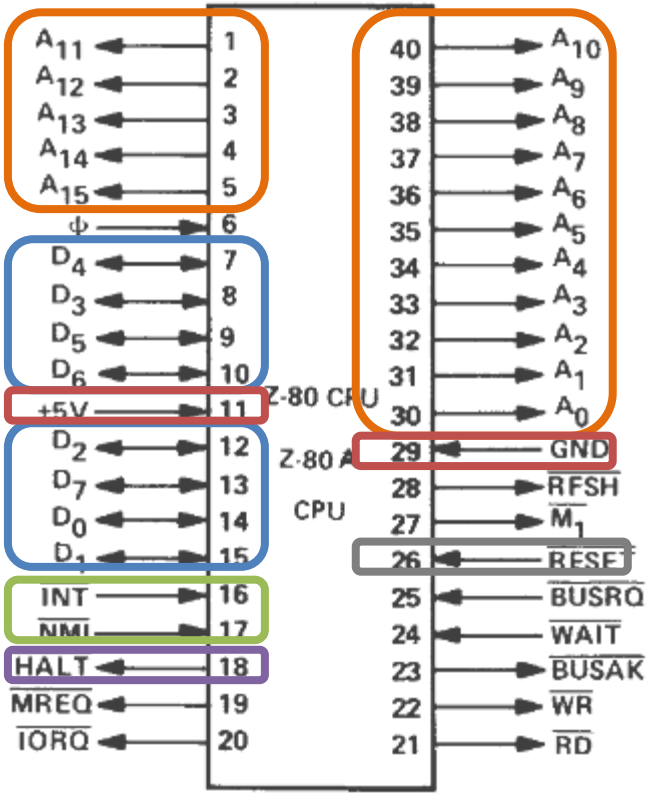
interrupt

attesa

Microcontrollore (MCU)
Atmel AVR
Atmega16



Cosa è un Microcontrollore



Microprocessore
(MPU)
Zilog Z-80

indirizzi

dati

interrupt

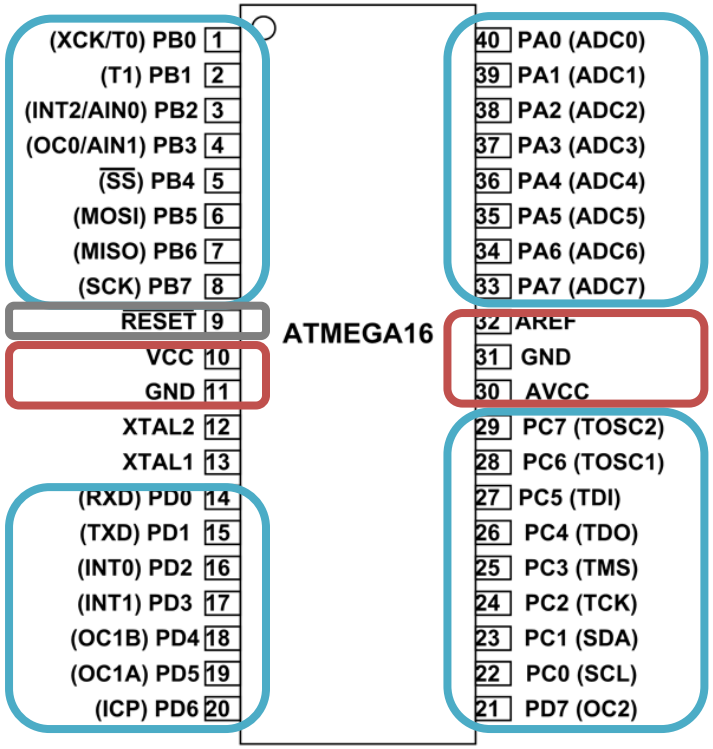
attesa

generic I/O

VCC/GND

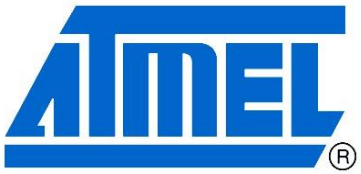
reset

Microcontrollore (MCU)
Atmel AVR
Atmega16



Venditori di Microcontrollori

- I principali **venditori** di **microcontrollori** sono:
  - Freescale
  - Atmel
  - Microchip Technology
  - Infineon
  - Texas Instruments
  - STMicroelectronics
  - NXP Semiconductors
  - Samsung Electronics
  - Fujitsu



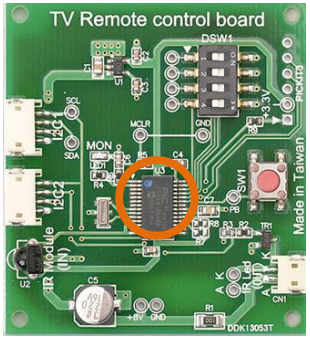
Campi Applicativi

- **Elettronica di consumo** (smartphone, tablet, orologi, registratori, calcolatrici, mouse, tastiere, modem,...)

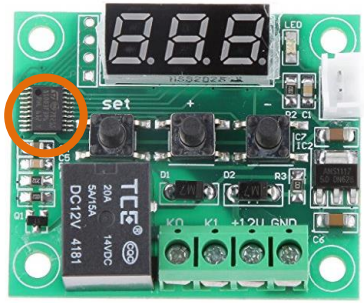
Smart Watch
ARM MCU

Mouse
PIC MCU

- **Automazione edifici** (serrature, allarmi, termostati, condizionatori, telecomandi, elettrodomestici, ...)



Remote Control
PIC MCU



Thermostat
STMicroelectronics MCU

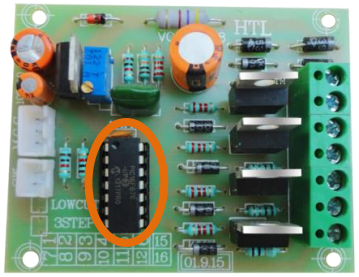
Campi Applicativi

- **Produzione di automobili** (centraline elettroniche, ABS, navigatori satellitari, sistema di intrattenimento,...)

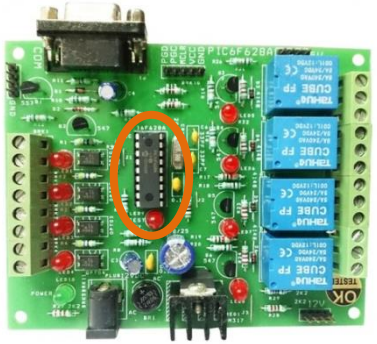
ABS control
Freescale MCU

Satellite
Navigator
ARM MCU

- **Automazione industriale** (controllo posizione e velocità assi, regolatori ON-OFF, regolatori PID, ...)



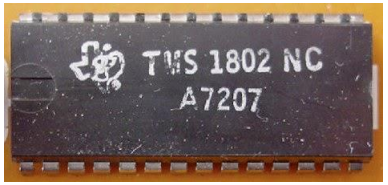
Stabilizer 3.5 Step
PIC MCU

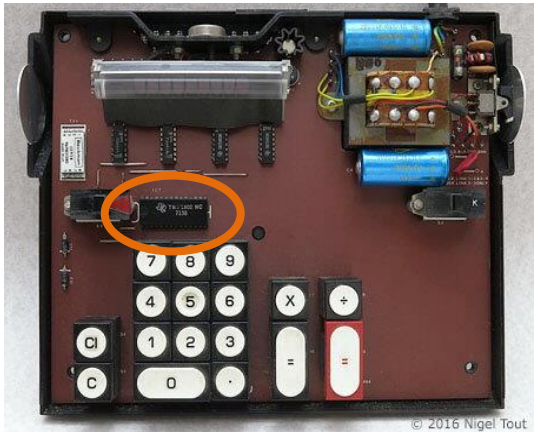


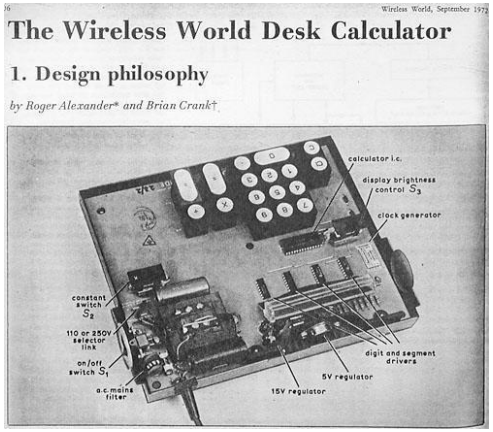
PLC board 4 in 4 out
PIC MCU

Cenni Storici

- Già quando Intel presentò il primo microprocessore nel **1971** (Intel 4004) nel mercato c'era una **domanda** per la produzione di **microcontrollori**
  - **TMS1802** progettato come microprocessore general purpose per calcolatori venne pubblicizzato per essere utilizzato in registratori di cassa, orologi e strumenti di misura







Cenni Storici

- Nel **1974** venne commercializzato il **TMS1000** che, nonostante venisse chiamato microcomputer, integrava già da allora nello stesso chip **RAM**, **ROM** e **periferiche di I/O**
- I primi microcontrollori con larga diffusione furono l'**Intel 8048** (utilizzato nelle tastiere), l'**Intel 8051** e la **famiglia 68HCxx** della Motorola

Tipologie di Microprocessore/Microcontrollore

- Così come i microprocessori, i microcontrollori possono essere caratterizzati in base a:
  - la **dimensione delle parole** manipolate
    - 8 bit, 16 bit, 32 bit, etc.
  - la struttura dell'**Instruction Set**
    - RISC (Reduced Instruction Set Computer)
    - CISC (Complex Instruction Set Computer)
  - le **funzionalità supportate**
    - elaborazione floating point

Famiglie di Microcontrollori

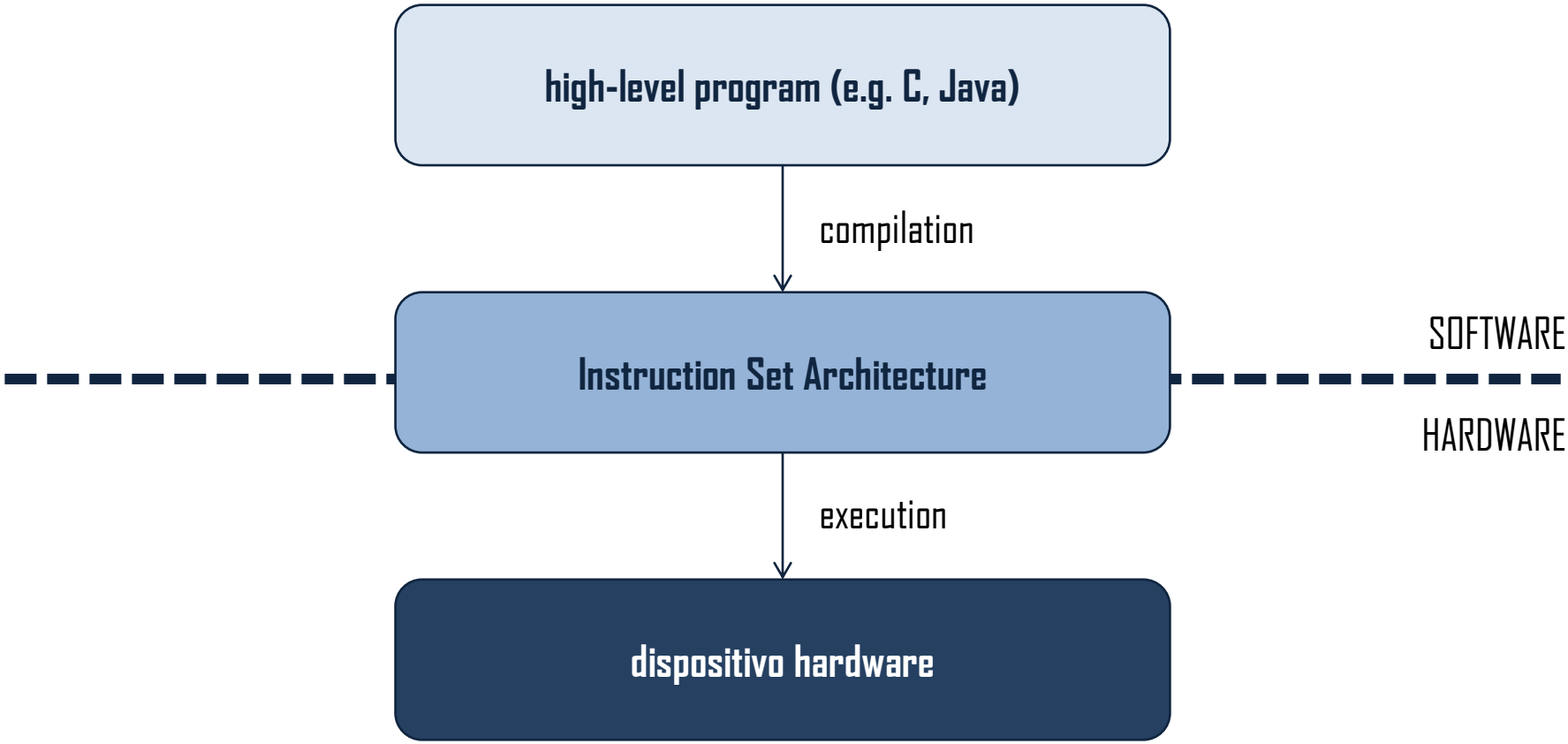
controller	Flash [KB]	SRAM [B]	EEPROM [B]	I/O pins	A/D channels	interfaces
AT90C8534	8	288	512	7	8	
AT90LS2323	2	128	128	3		
AT90LS2343	2	160	128	5	8	UART, SPI
AT90LS8535	8	512	512	32		
AT90S1200	1	64	128	15		
AT90S2313	2	160		15		
ATmega128	128	4096	4096	53	8	JTAG, SPI, IIC
ATmega162	16	1024	512	35		JTAG, SPI
ATmega169	16	1024	512	53	8	JTAG, SPI, IIC
ATmega16	16	1024	512	32	8	JTAG, SPI, IIC
ATtiny11	1		64	5 + 1 in		SPI
ATtiny12	1		64	6		
ATtiny15L	1	128	64	6	4	SPI
ATtiny26	2	128	128	11 + 8 in	16	SPI
ATtiny28L	2					

Indice

- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

Utilizzo di in un Microprocessore/Microcontrollore

Le applicazioni in un microprocessore possono esser viste attraverso diversi livelli di astrazione che vanno dal **software** all'**hardware**: l'**Instruction Set Architecture (ISA)** costituisce l'interfaccia tra i due mondi.



Instruction Set Architecture

- L'ISA fornisce le **specifiche funzionali** ai **programmatori software** per utilizzare/programmare l'hardware in modo tale da eseguire delle operazioni specifiche

- L'ISA fornisce i **requisiti funzionali** agli **sviluppatori hardware** in modo tale che l'hardware sviluppato (la micro-architettura) possa eseguire i programmi software

Instruction Set Architecture

- L'ISA specifica tutti gli aspetti del processore visibili al programmatore:
  - **istruzioni**, tipi di dato e formato delle istruzioni
  - **operandi**
    - registri
    - memorie
  - modalità di **indirizzamento**

Instruction Set Architecture

- Le **architetture** di microcontrollori sono tipicamente definite per un certo **numero di bit** (8, 16, 32 o 64 bit tipicamente)
  - il numero di bit dell'architettura ne definisce
    - dimensione dei **datapath**
    - dimensione dei **dati** manipolati in maniera nativa
    - dimensione degli **indirizzi** di memoria
  - il **numero di bit** di un'architettura **non definisce** la rispettiva **dimensione delle istruzioni**

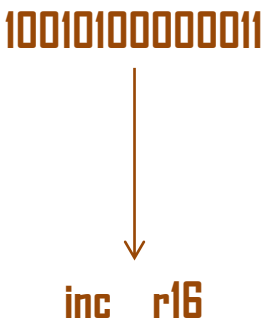
Istruzioni

- Le istruzioni sono la parte più importante dell'ISA in quanto specificano le **operazioni elementari disponibili** per il programmatore (ad esempio istruzioni aritmetiche).

- L'insieme di istruzioni è **specifico per il microprocessore** a cui ci si riferisce: la stessa operazione può essere scritta diversamente in microprocessori differenti.

Istruzioni

- Le istruzioni possono essere scritte in:
  - **linguaggio macchina**
    - composto da numeri binari
    - utilizzato dall'hardware
  - **linguaggio assembly**
    - rappresentazione testuale del linguaggio macchina (più facile da capire)
    - utilizzato dagli sviluppatori



Tipi di Dato

- Diverse tipologie di dato possono essere supportate
  - **numeriche**
    - interi di diverse dimensioni (8, 16, 32, 64 bit)
      - con o senza segno
    - floating point single (32 bit) o double (64 bit) precision
  - **non numeriche**
    - booleani
    - caratteri (ASCII, Unicode)

es. Tipi di dato supportati dal Pentium II

data type	8 bit	16 bit	32 bit	64 bit
signed integer	x	x	x	
unsigned integer	x	x	x	
floating point			x	x

es. Dimensione tipi di dato C per Linux e Windows

operating system	pointers	int	long int	long long int
Microsoft Windows	64 bis	32 bis	32 bis	64 bis
Linux, Most Unix	64 bis	32 bis	64 bis	64 bis

Formato delle Istruzioni

- È la definizione di **come le istruzioni sono rappresentate in codice binario**, ovvero in linguaggio macchina
- Tipicamente le istruzioni consistono di due tipologie di campi
  - **opcode** (operation code), definisce l'operazione da effettuare
  - **operandi**, definiscono gli oggetti dell'operazione da effettuare
- Le istruzioni solitamente hanno 0, 1, 2 o 3 operandi

Formato delle Istruzioni

- **Reduced Instruction Set Computer (RISC)**
  - piccolo numero di istruzioni
  - istruzioni semplici, spesso della stessa lunghezza e durata
  - tipicamente utilizzate nei **microcontrollori**
- Complex Instruction Set Computer (**CISC**)
  - ogni istruzione può eseguire diverse sotto-operazioni (load, exec, store)
  - è richiesto un supporto hardware complesso e tipicamente le istruzioni hanno durate molto diverse

Dimensione delle Istruzioni

- La dimensione delle istruzioni ne definisce il numero di bit
- Alcune architetture hanno istruzioni a **dimensione fissa** (ad esempio il MIPS), altre invece hanno istruzioni con **dimensioni variabili** (ad esempio l'AVR)

Codifica delle Istruzioni

- Codifica delle **operazioni**
  - 2^N operazioni diverse richiedono N bit
- Codifica degli **operandi**
  - dipendono dalle modalità di indirizzamento (come indirizzo il dato) e dagli spazi di indirizzi (dove sta il dato)
- Con istruzioni a lunghezza fissa, la presenza di più operazioni significa anche la disponibilità di meno bit per la codifica degli operandi, per cui è necessario un compromesso

Istruzioni dell'ARM

- L'ARM definisce diverse famiglie di **instruction set architecture** a cui i produttori fanno riferimento per realizzare **processori ARM proprietari**
- Ogni **ISA ARM** definisce le **istruzioni supportate** dal processore e le linee guida per definirne l'**architettura** (es. gerarchia di memoria, bus, etc.)
- Occorre far riferimento al manuale dell'ISA della famiglia ARM per la descrizione dettagliata del microcontrollore

Istruzioni dell'ARM

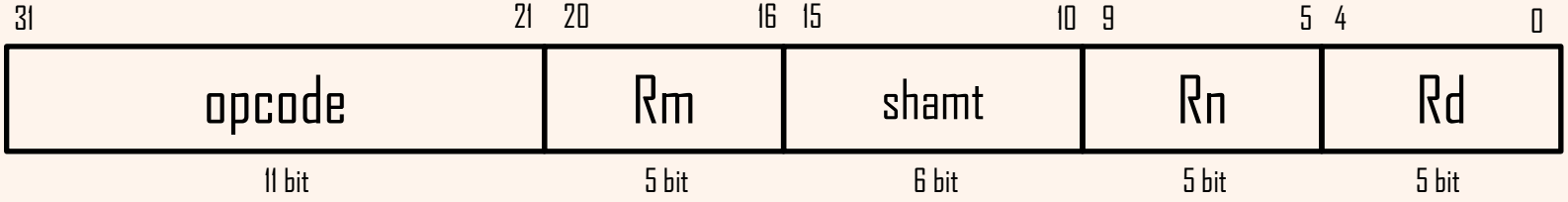
- Due principali famiglie di ISA ARM sono considerate durante il corso
  - **Progettazione** (simulazione hardware)
    - ARMv8: famiglia di ISA a 32 o 64 bit con diversi profili (tra cui anche quello M specifico per microcontrollori), a cui fanno capo, tra le altre, le architetture di microcontrollori Cortex-M32 e Cortex-M33
      - **LEGv8**: ISA derivata dall'ARMv8 a scopi didattici adottata durante il corso
  - **Programmazione** (esecuzione su dispositivo reale)
    - **ARMv6-M**: famiglia di ISA a 32 bit, concepita nello specifico per microcontrollori (M), a cui fanno capo le architetture di microcontrollori Cortex-M0, Cortex-M0+ e Cortex-M1

Istruzioni dell'ARMv8

- Le **istruzioni** dell'ARMv8 sono a **32 bit** (vi possono essere istruzioni a 16 bit, dette Thumb, volte a ottimizzare la dimensione dei programmi in memoria)
  - es. ADDS X0, X1, X2 (compilabile a 32 o a 16 bit)
- La **maggior parte** delle istruzioni dell' ARMv8 richiede **un ciclo di clock** per essere eseguita, ma alcune istruzioni richiedono **più cicli** o hanno un **numero di cicli variabile**
  - es. MOVS R0, IMM richiede 1 ciclo di clock
  - es. LDR R0, [R1, IMM] richiede 2 cicli di clock

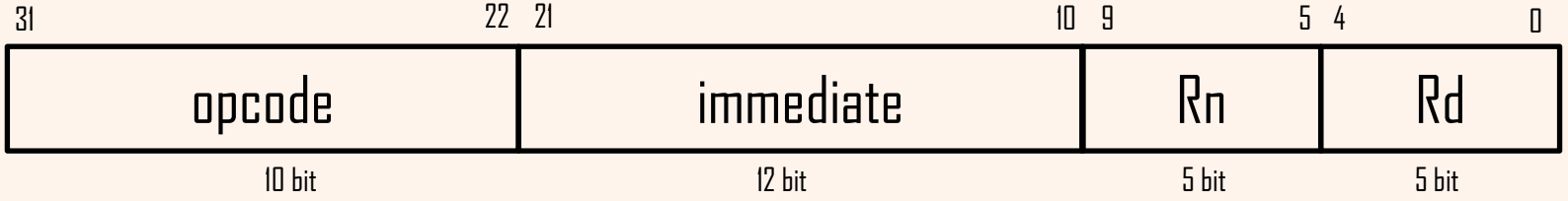
Formati di Istruzione del LEGv8

- **Tipo R** (agiscono sui registri, es. ADD X0, X1, X2)



- **Tipo D** (trasferimento dati, es. LDR X0, [X1, K])

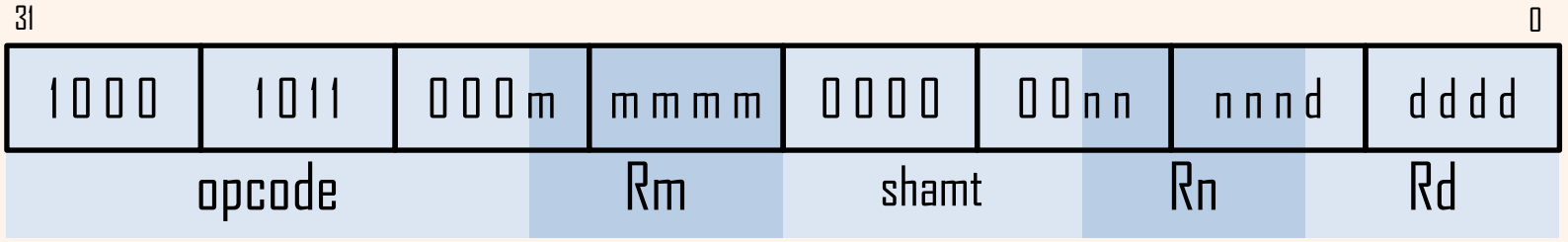
- **Tipo I** (immediate, es. ADDI X0, X1, #K)



Istruzioni Artimetico-Logiche del LEGv8

- **Aritmetiche**
  - addizione/sottrazione: ADD, ADDI, ADDS, ADDIS, SUB, SUBI, SUBS, SUBIS
  - moltiplicazione/divisione: MUL, SMULH, UMULH, SDIV, UDIV
- **Logiche**
  - AND: AND, ANDI
  - OR inclusivo: ORR, ORRI
  - OR esclusivo: EOR, EORI
  - shift logico: LSL, LSR

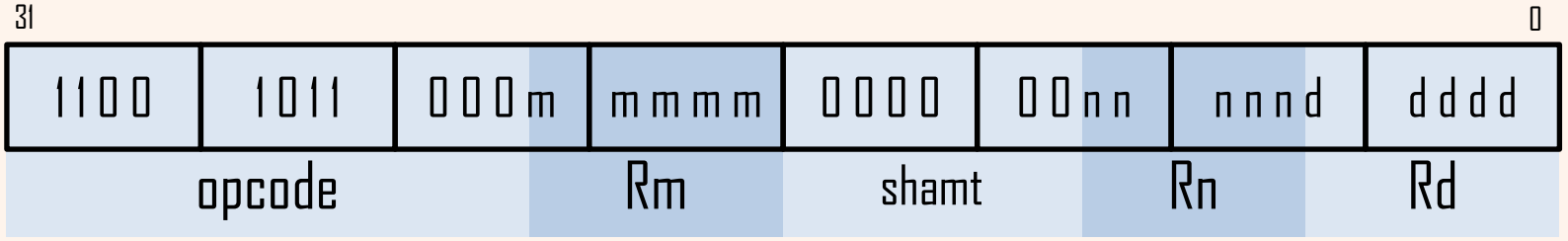
Somma



ADD – add

sintassi	`ADD Rd, Rn, Rm`
operazione	Rd ← Rn + Rm
operandi	0 ≤ d,n,m ≤ 31;
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	R

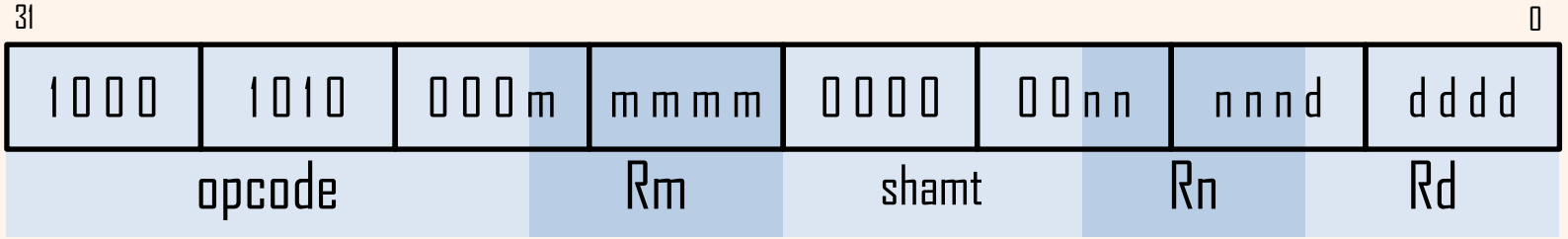
Sottrazione



SUB – subtract

sintassi	`SUB Rd, Rn, Rm`
operazione	Rd ← Rn + Rm
operandi	0 ≤ d,n,m ≤ 31;
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	R

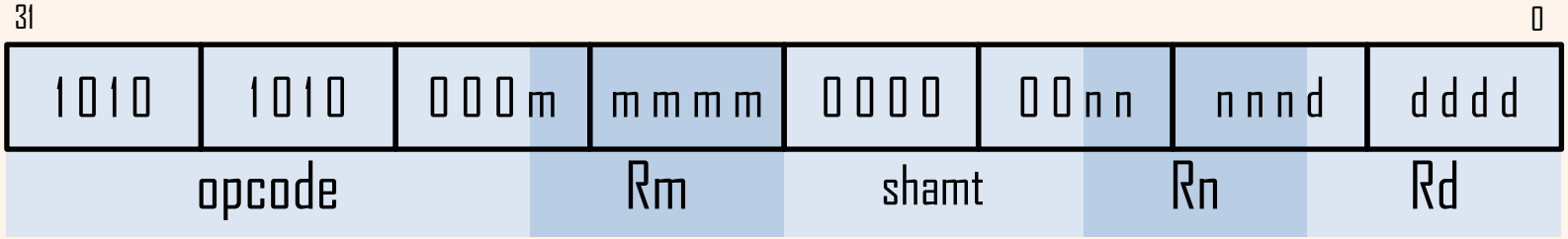
AND Bit a Bit



AND – bitwise AND

sintassi	`AND Rd, Rn, Rm`
operazione	Rd ← Rn & Rm
operandi	0 ≤ d,n,m ≤ 31;
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	R

OR Inclusivo Bit a Bit



ORR – bitwise OR	
sintassi	`ORR Rd, Rn, Rm`
operazione	Rd ← Rn | Rm
operandi	0 ≤ d,n,m ≤ 31;
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	R

Esempio: Calcolo Aritmetico (1)

C

```
// X0 ← a, X1 ← b, X2 ← c, X3 ← d , X4 ← e
a = b + c;
d = a - e;
```

Esempio: Calcolo Aritmetico (1)

C

```
// X0 ← a, X1 ← b, X2 ← c, X3 ← d , X4 ← e
a = b + c;
d = a - e;
```

assembly

```
ADD X0, X1, X2          ; a ← b + c
SUB X3, X0, X4          ; d ← a - e
```

2 cicli

Esempio: Calcolo Aritmetico (2)

C

```
// X19 ← f, X20 ← g, X21 ← h
// X22 ← i, X23 ← j
f = (g + h) - (i + j);
```

Esempio: Calcolo Aritmetico (2)

C

```
// X19 ← f, X20 ← g, X21 ← h
// X22 ← i, X23 ← j
f = (g + h) - (i + j);
```

assembly

```
ADD X9, X20, X21     ; tmp0 ← g + h
ADD X10, X22, X23    ; tmp1 ← i + j
SUB X19, X9, X10     ; h ← (g + h) - (i + j)
```

3 cicli

Indice

- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

Registri

- I registri sono il **banco di lavoro** principale del processore e costituiscono la locazione più veloce in cui memorizzare dati da elaborare
- Vi sono principalmente due **tipi di registri** nei microprocessori:
  - **general purpose**
  - **special purpose**
    - program counter (PC)
    - statur register (SR o SREG)
    - stack pointer (SP)

Registri General Purpose

- I registri general purpose sono utilizzati per **memorizzare dati e risultati temporanei**
- Possono essere organizzati diversamente a seconda del microprocessore specifico
  - Possono essere separati a seconda del fatto che contengano dati o indirizzi (ad esempio nel 68K)
  - Possono essere di **diverso numero e dimensione**
    - 32 registri a 32 bit per il MIPS
    - 32 registri a 8 bit per l'AVR
    - 16 o 32 registri a 32 o 64 bit per l'ARM

Program Counter (PC)

- Registro speciale che memorizza **l'indirizzo di memoria dell'istruzione in esecuzione**
- Può essere di **diverse dimensioni** (16 bit, 32 bit, 64 bit) a seconda della **dimensione della memoria istruzioni**
- Può **incrementare automaticamente** di una quantità pari alla dimensione dell'istruzione in relazione all'indirizzamento della memoria (es. con istruzioni a 32 bit e memoria indirizzata al byte, l'incremento necessario a indirizzare l'istruzione successiva sarà pari a 4)

Status Register (SREG o SR)

- Contiene dei **bit associati alle operazioni del processore**
- Viene utilizzato per controllare il flusso di esecuzione del programma
- **Bit di stato** tipici sono:
  - overflow (V) – il risultato dell'operazione non è rappresentabile
  - carry (C) – l'operazione ha dato luogo ad un riporto
  - zero (Z) – il risultato dell'operazione è zero
  - negative (N) – il risultato dell'operazione è negativo

Banco di Registri dell'ARMv8

- General Purpose Registers (GPRs)
  - **27 registri a 32 o 64 bit (X0-X27)**
- Special Purpose Registers (SPRs)
  - **Stack Pointer (SP o X28)**
    - memorizza l'indirizzo della cima dello stack in ogni istante di funzionamento (necessario nell'esecuzione di sub-routine assieme al **Frame Pointer, FP o X29**)
  - **Link Register (LR o X30)**
    - memorizza l'indirizzo di ritorno da una sub-routine
  - **Program Counter (PC o X31)**
    - memorizza l'indirizzo della prossima istruzione da eseguire

Status Register nell'ARMv8

- **Application Program Status Register (APSR)** (es. 32 bit)
  - registro separato rispetto al banco principale

  - N – NEGATIVE condition code flag
  - Z – ZERO condition code flag
  - C – CARRY condition code flag
  - V – OVERFLOW condition code flag

Banco di Registri del LEGv8

- Nella versione didattica iniziale adottata, la **LEGv8**, vi sono **32 registri a 32 o 64 bit (X0-X31)**:
  - per poter essere manipolati da istruzioni aritmetico logiche i dati devono risiedere in questi registri
  - il registro X31 (o XZR) è sempre pari a 0
  - **non sono presenti** i registri speciali **stack pointer (SP)**, **link register (LR)** e **status register (APSR)**
  - il **program counter (PC)** viene implementato come un **registro separato** rispetto al banco di registri principale (nell'ARMv8 è in X31)

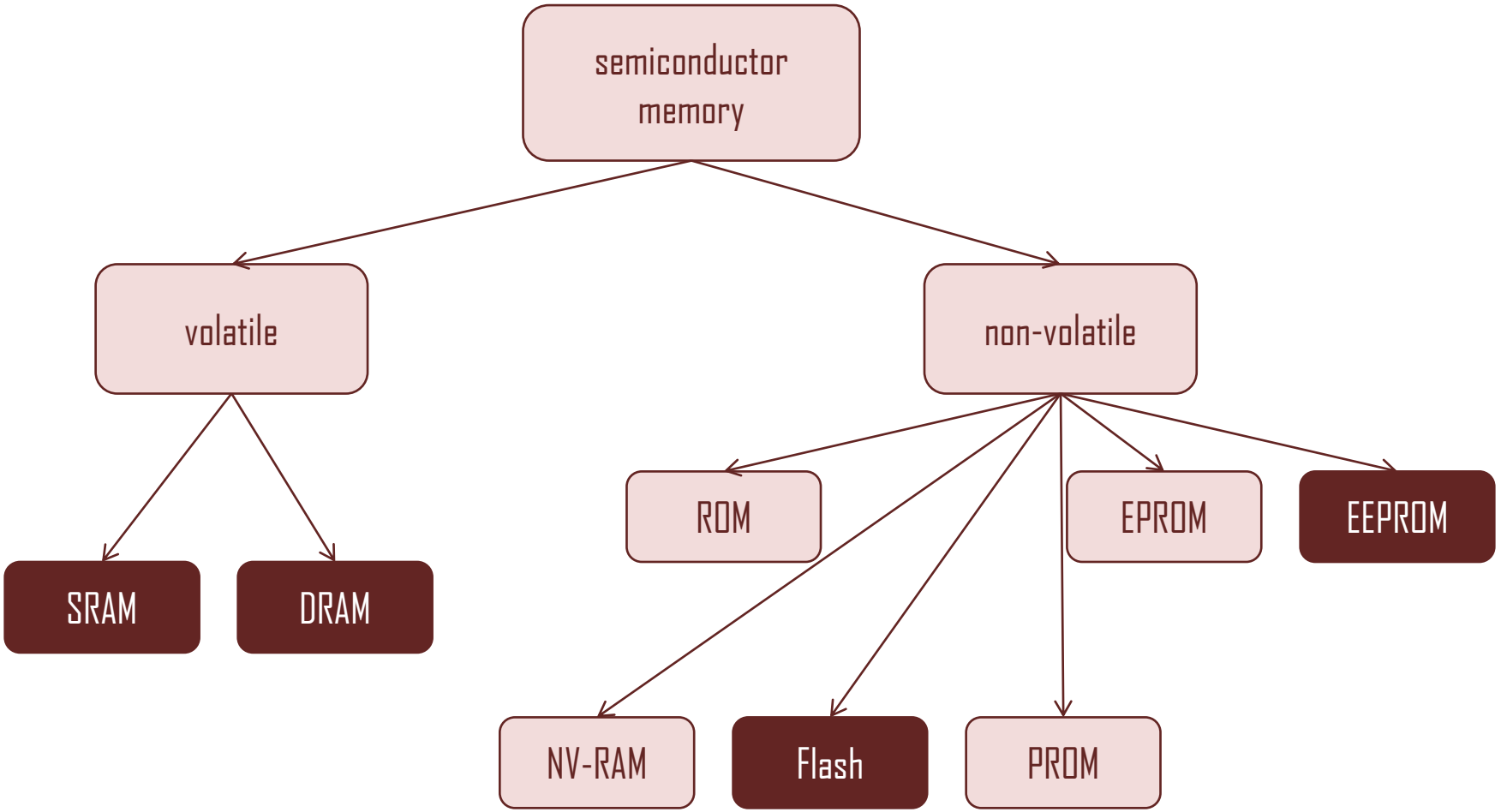
Indice

- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

Memorie

- I dati elaborati dal processore sono tipicamente tanti e non possono essere memorizzati tutti all'interno dei registri
- Sia i **dati** che il **programma** da eseguire necessitano il **slavataggio in una memoria**
- Il modello di memoria adottato dipende da come la memoria viene utilizzata per memorizzare i dati e sancisce la **dimensione di elemento indirizzabile**, l'**endianness** (ordine dei byte) e l'**allineamento**

Tipologie di Memoria



Memorie Volatili e Non Volatili

- Le **memorie volatili** mantengono il proprio contenuto finché il sistema è alimentato
  - sono **meno costose e più veloci** di quelle non volatili (tempi accesso nell'ordine di 1 ns nelle volatili e fino a qualche ms nelle non volatili)
- Le **memorie non volatili** mantengono l'informazione memorizzata anche quando l'alimentazione viene spenta
  - ma la loro **scrittura è più lenta e complicata** rispetto alle memorie volatili

Memorie nei Microcontrollori

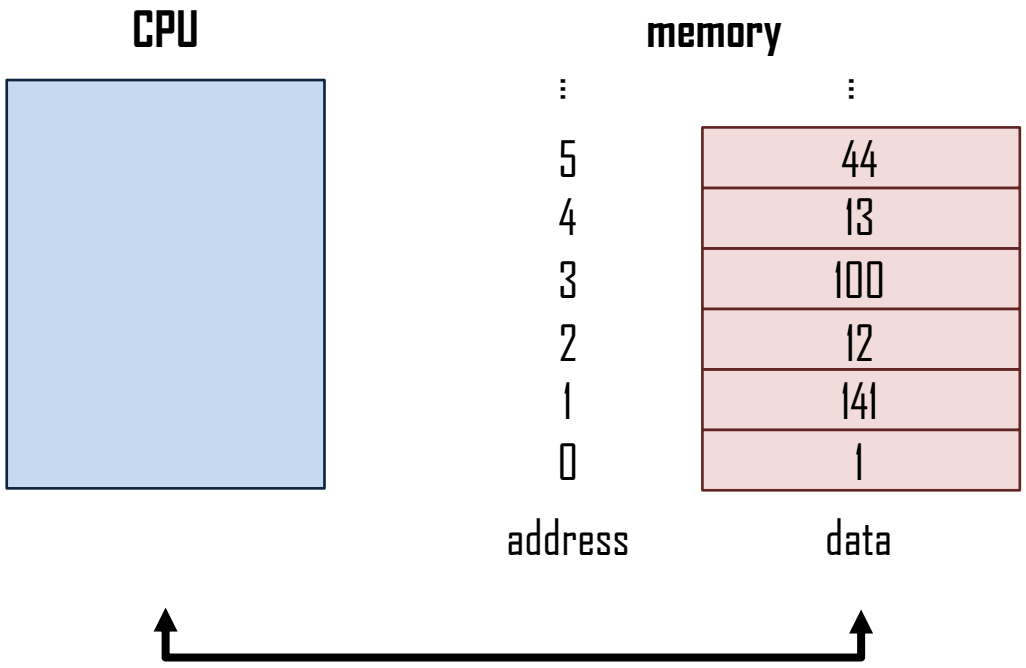
- Memorie **Volatili**:
  - **SRAM**: celle con Flip-Flop (6 transistor CMOS), tempo accesso breve, no refresh
  - **DRAM**: celle con 1 transistor e 1 capacitore, tempo accesso breve, refresh necessario (carica capacitore)
- Memorie **Non-Volatili**:
  - **EEPROM**: memorie cancellabili in maniera selettiva (cella per cella) un numero limitato di volte (100000-10000000 volte)
  - **Flash**: memorie cancellabili in maniera non selettiva (a blocchi o tutta la memoria assieme) un numero limitato di volte

Famiglie di Microcontrollori

controller	Flash [KB]	SRAM [B]	EEPROM [B]	I/O pins	A/D channels	interfaces
AT90C8534	8	288	512	7	8	
AT90LS2323	2	128	128	3		
AT90LS2343	2	160	128	5	8	UART, SPI
AT90LS8535	8	512	512	32		
AT90S1200	1	64	128	15		
AT90S2313	2	160		15		
ATmega128	128	4096	4096	53	8	JTAG, SPI, IIC
ATmega162	16	1024	512	35		JTAG, SPI
ATmega169	16	1024	512	53	8	JTAG, SPI, IIC
ATmega16	16	1024	512	32	8	JTAG, SPI, IIC
ATtiny11	1		64	5 + 1 in		SPI
ATtiny12	1		64	6		
ATtiny15L	1	128	64	6	4	SPI
ATtiny26	2	128	128	11 + 8 in	16	SPI
ATtiny28L	2					

Memorie

La memoria è vista come un **array monodimensionale** accessibile attraverso un **indirizzo** che funge da **indice** dell'array il cui primo elemento corrisponde all'indice 0. Solitamente vi sono delle **istruzioni** apposite per il **trasferimento dei dati** da registri della CPU a memoria e viceversa.



Spazio di Indirizzi

- Lo spazio di indirizzi definisce l'**intervallo di indirizzi a cui può accedere il processore**
- Ci può essere più di uno spazio di indirizzi in un processore. Se la distinzione riguarda memoria dedicata ai dati e alle istruzioni si può avere
  - **Von Neumann architecture**: prevede un singolo spazio di indirizzi lineare per la memoria dati e per la memoria istruzioni
  - **Harvard architecture**: prevede spazi di memoria sperati per la memoria dati e per la memoria istruzioni

Spazio di Indirizzi

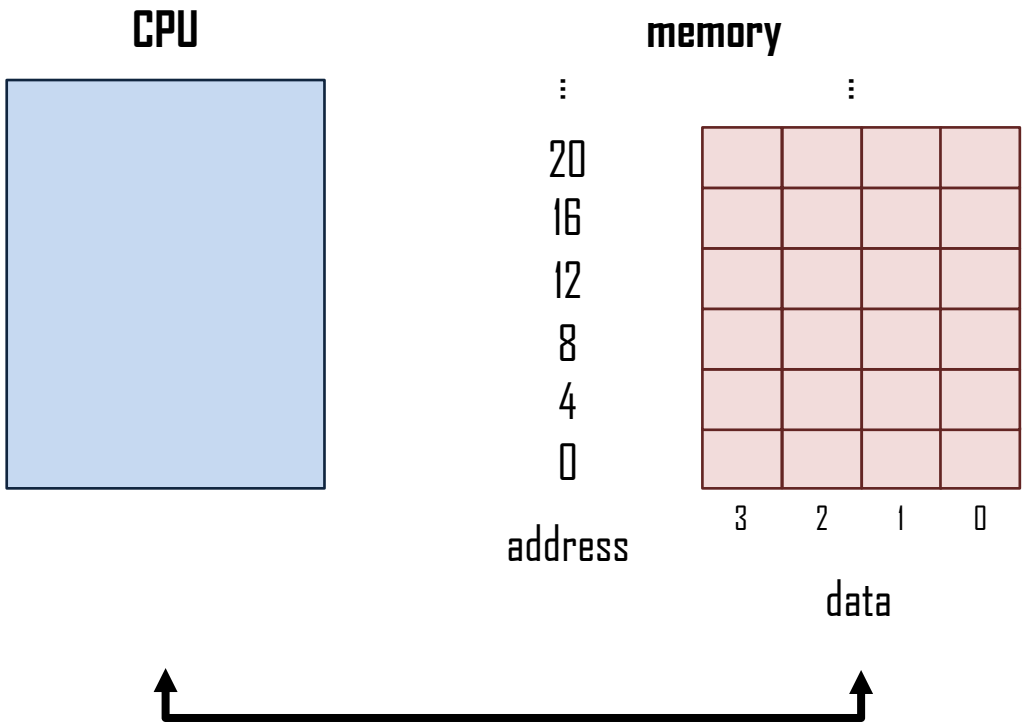
- Lo spazio di indirizzi **non viene necessariamente utilizzato solo per le memorie**
  - ad esempio nell'ATmega328P è possibile accedere a tutti i registri general purpose tramite indirizzi di memoria (mappaggio in memoria)
- Lo spazio di indirizzi è **limitato dalla profondità del bus di indirizzi**
  - se uno **spazio** di indirizzi è **condiviso da più risorse** (ad esempio memoria e registri) la **limitazione va estesa** alla somma degli intervalli dedicati a tali risorse

Dimensione di Elemento Indirizzabile

- La memoria è composta da **tanti elementi** (di dimensione maggiore a 1 bit) ciascuno dei quali corrisponde ad un **indirizzo specifico**
- La **dimensione più comune di un elemento indirizzabile** è di **8 bit** (1 byte), ma tipicamente i processori moderni manipolano parole a più byte
  - parole di 32 bit (4 byte) per la memoria istruzioni e dati nel MIPS
  - parole di 16 bit (2 byte) per la memoria istruzioni, e di 8 bit (1 byte) per la memoria dati nell'ATmega328P
- È quasi **sempre possibile indirizzare il singolo byte**

Dimensione di Elemento Indirizzabile

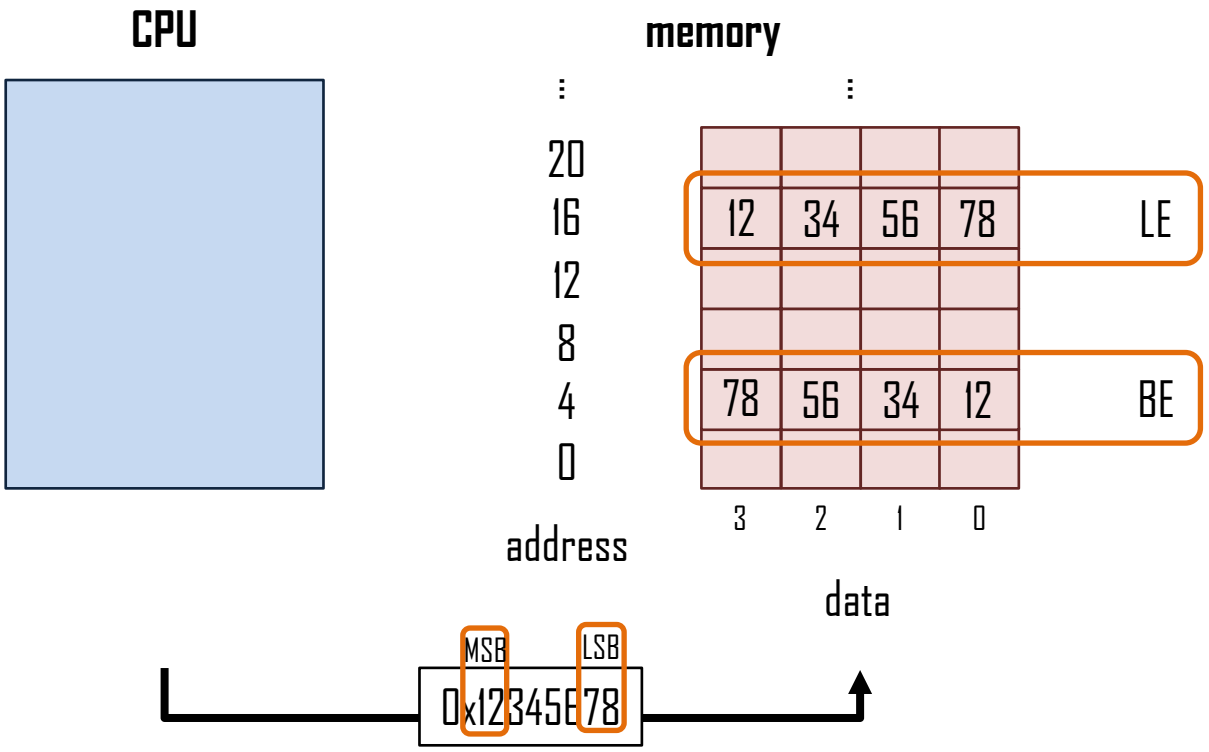
Gli **indirizzi di memoria di parole (word) contigue**, per memorie capaci di indirizzare il singolo byte ma con parole a più byte, non sono consecutivi ma **differiscono di più unità**. Nel MIPS, dove le parole sono a 4 byte, essi differiscono di 4 unità.



Endianness

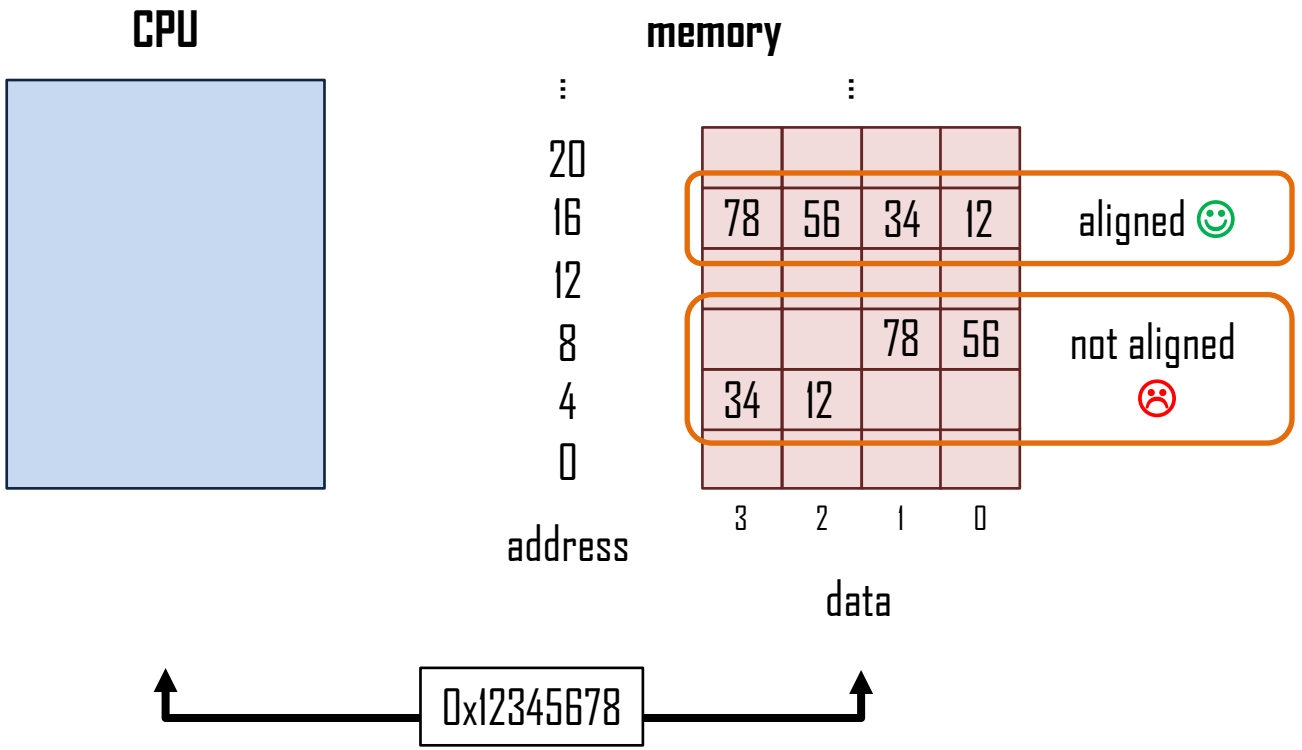
In memorie con parole a più byte e indirizzamento al singolo byte sono necessari diversi indirizzi di memoria per una parola. Vi sono due **convenzioni di ordinamento** dei byte:

- **little endian** (LE): il LSB viene memorizzato nell'indirizzo minore
- **big endian** (BE): il MSB viene memorizzato nell'indirizzo minore



Allineamento

Alcune tipologie di memoria con indirizzamento al byte e parole a più byte hanno delle **restrizioni sull'allineamento delle parole**: esse devono essere posizionate in indirizzi di memoria che sono multipli interi del numero di byte della parola per un accesso più efficiente (meno letture richieste).



Spazi di Indirizzo nell'ARMv6-M

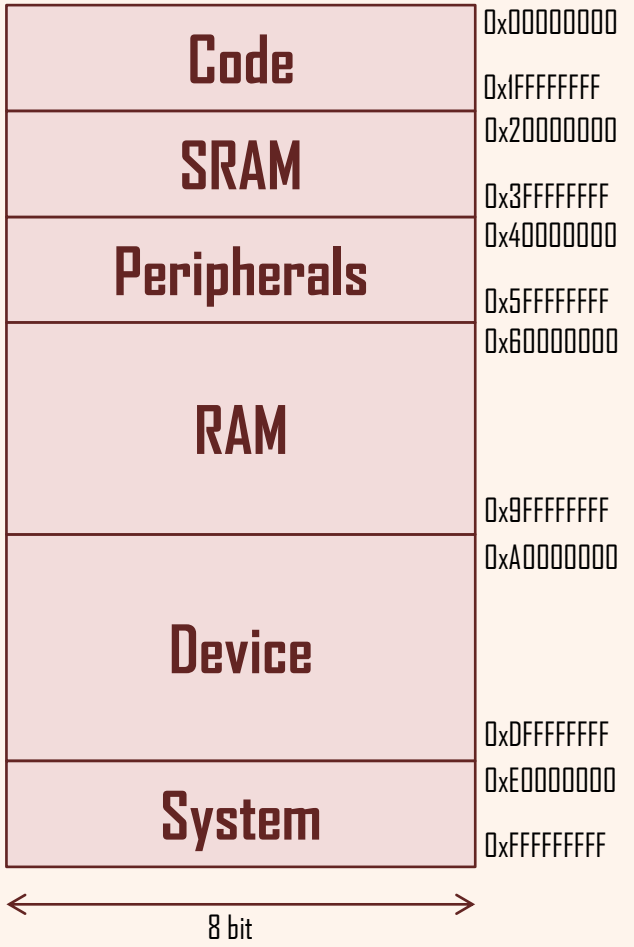
- L'ARMv6-M dispone di un **unico spazio indirizzi**
  - dimensione di elemento indirizzabile: **1 Byte**
  - profondità bus indirizzi: **32 bit** (indirizzi da 0 a 2^{32})
  - dimensione memoria (numero di Byte): **2^{32} B** (4 GB)
    - in realtà 4 GB è la dimensione massima di memoria indirizzabile dai microcontrollori della famiglia ARMv6-M, ma possono esserci dei dispositivi conformi all'ISA ARMv6-M con una quantità di memoria inferiore a 4 GB

Spazi di Indirizzo nell'ARMv6-M

- L'ARMv6-M è un'architettura a 32 bit (datapath, dati e indirizzi sono a 32 bit), ma permette l'accesso in memoria:
  - al **byte** (8 bit)
  - alla **halfword** (16 bit)
  - alla **word** (32 bit)
- È consentita **solo** la memorizzazione di **dati allineati**
- È possibile **configurare** l'allineamento come **LE o BE**, ma **alcuni sotto-spazi** hanno **restrizioni** (e.g. lo spazio riservato alle istruzioni è allineato solo in LE)

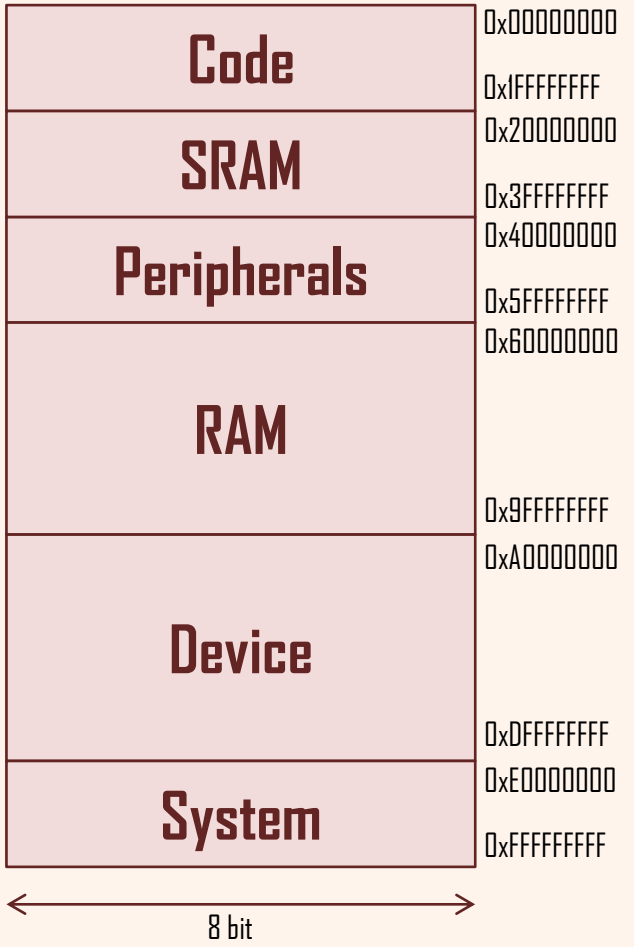
Spazi di Indirizzo nell'ARMv6-M

- Contiene diversi sotto-spazi (architettura di **Von Neumann**):
  - **Code:** spazio dedicato alla memoria istruzioni, parte sempre dall'indirizzo 0, tipicamente ROM o Flash
  - **SRAM:** spazio tipicamente utilizzato come memoria dati SRAM on-chip
  - **Peripherals:** spazio di indirizzi dedicato alle periferiche on-chip



Spazi di Indirizzo nell'ARMv6-M

- **RAM:** spazio riservato a memorie di tipo diverso, in particolare con writeback e supportate da cache
- **Device:** spazio inteso per dispositivi esterni, che può essere condiviso o meno con questi ultimi (e.g. DMA)
- **System:** spazio dedicato al periferiche private del processore e a periferiche di sistema del venditore (e.g. debug points)



Spazi di Indirizzo nel LEGv8

- Nell'**ARMv8** lo **spazio di indirizzi** è simile a quello dell'**ARMv6-M** sia per l'architettura a 32 bit che per quella a 64 bit
- Nella versione didattica iniziale adottata, la **LEGv8**, lo **spazio di indirizzi** viene **semplificato** e comprende solamente i sotto-spazi Code (**memoria istruzioni**) e SRAM (**memoria dati**), che sono tuttavia gestiti in **maniera separata ed indipendente** (architettura di **Harvard**)

Indice

- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

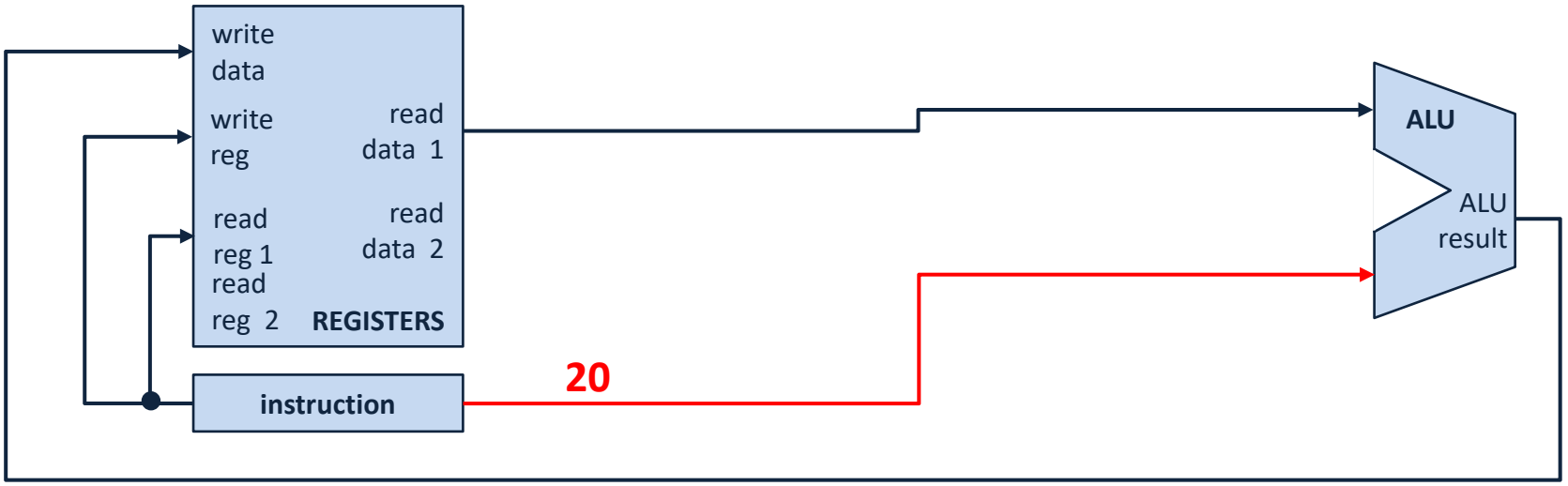
Modalità di Indirizzamento

- L'ISA definisce anche le modalità di indirizzamento, ovvero **come è possibile reperire gli operandi**
  - dall'istruzione stessa (**immediate**)
  - dai **registri**
    - l'indirizzo del registro è nell'istruzione
  - dalla **memoria**
    - l'indrizzo della memoria è nell'istruzione
    - l'indirizzo della memoria è in un registro il cui indirizzo è nell'istruzione
    - l'indirizzo della memoria è da calcolare dal valore di un registro il cui indirizzo è nell'istruzione

Indirizzamento Immediate

- L'operando è contenuto nell'istruzione stessa (è immediatamente disponibile)
  - es. 68K (d0 è un registro della CPU):

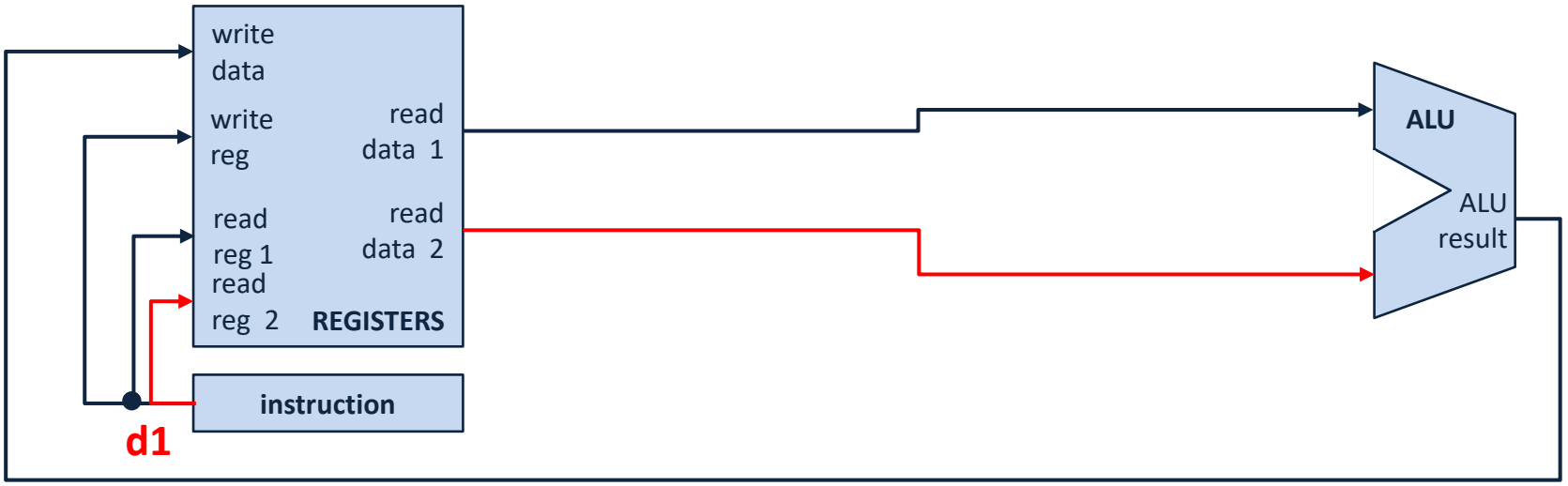
```
addw #20, d0    # d0 ← d0 + 20
```



Indirizzamento di Registro Diretto

- L'operando è un registro il cui identificativo è direttamente specificato nell'istruzione
  - es. 68K (d0 e d1 sono registri della CPU):

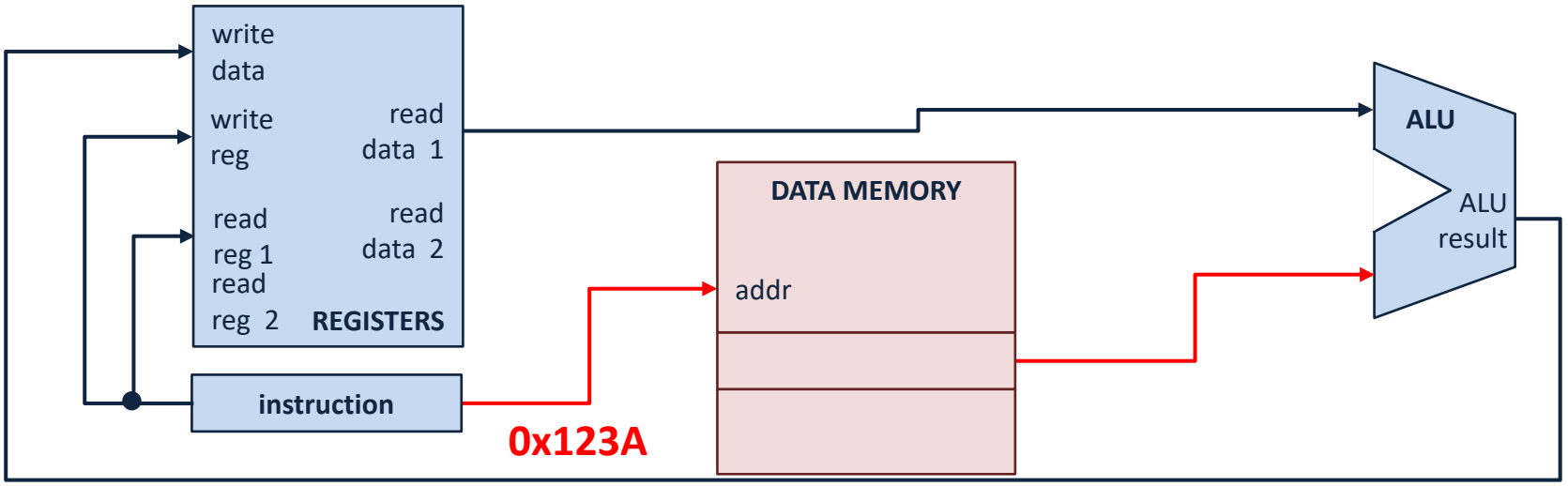
```
addw d1, d0        # d0 ← d0 + d1
```



Indirizzamento di Memoria Diretto

- L'operando è un dato in memoria il cui indirizzo è direttamente specificato nell'istruzione
  - es. 68K (d0 è un registro della CPU):

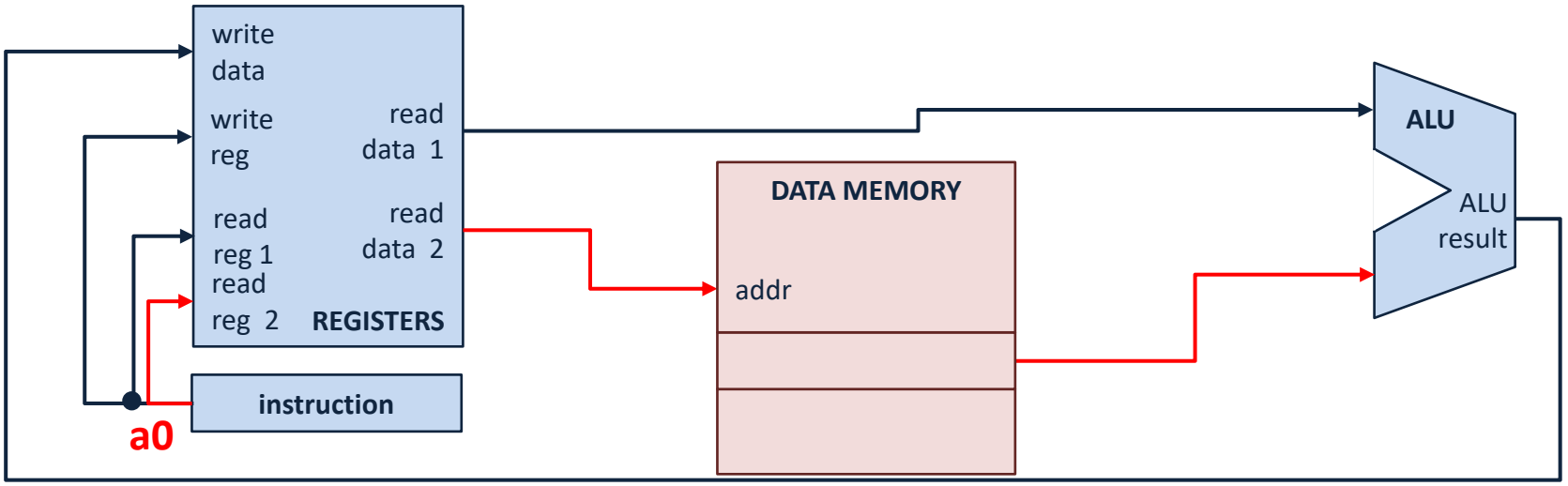
```
addw 0x123A, d0 # d0 ← d0 + mem[0x123A]
```



Indirizzamento di Memoria Indiretto

- L'operando è un dato in memoria il cui indirizzo è dato da un registro il cui identificativo è direttamente specificato nell'istruzione
  - es. 68K (d0 e a0 sono registri della CPU):

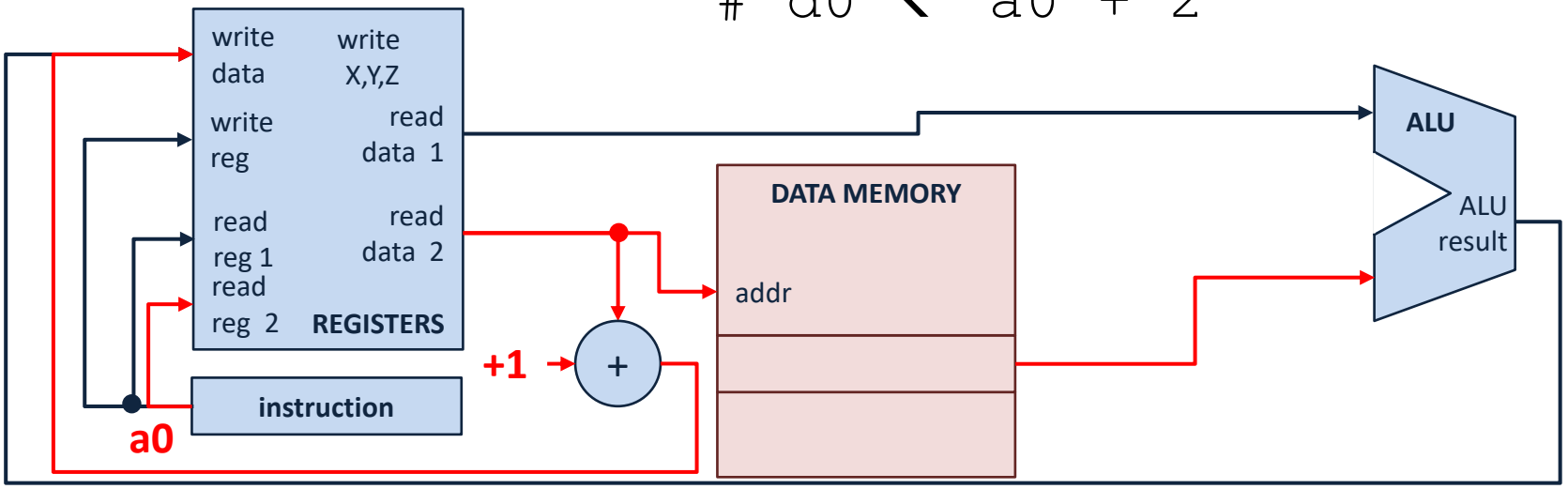
```
addw (a0), d0 # d0 ← d0 + mem[a0]
```



Indirizzamento di Memoria Indiretto con Post-Incremento/Decremento

- L'operando è un dato in memoria indirizzato indirettamente da un registro successivamente aggiornato per puntare alla locazione di memoria successiva/precedente
  - es. 68K (d0 e a0 sono registri della CPU):

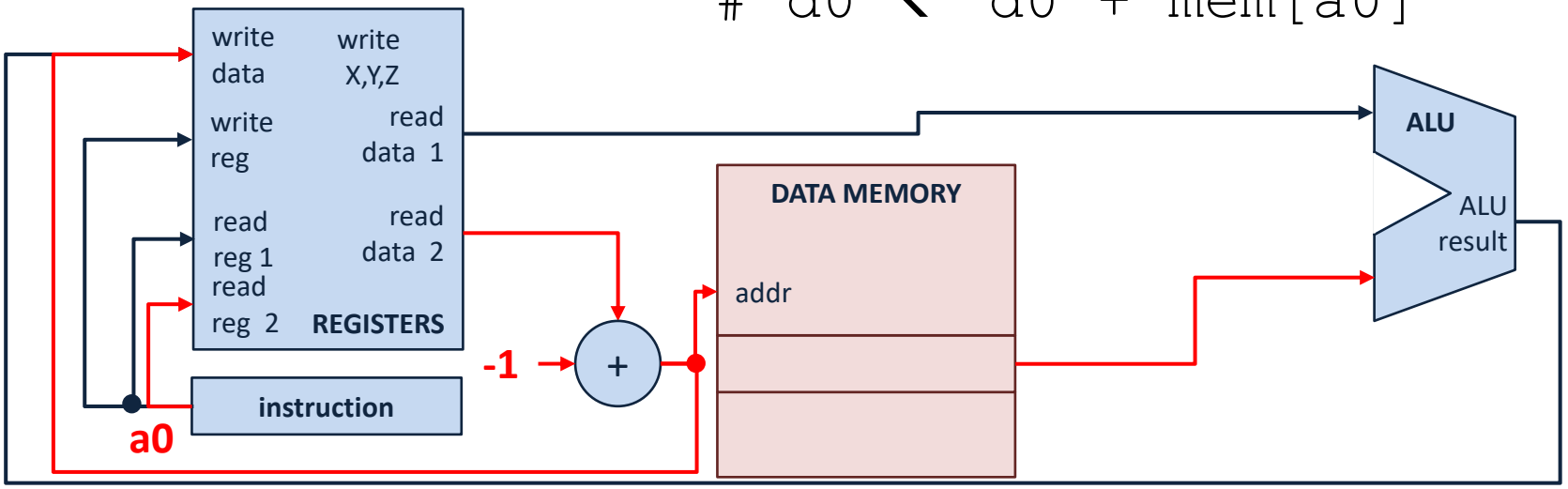
```
addw (a0)+, d0  # a0 ← d0 + mem[a0]
                # d0 ← a0 + 2
```



Indirizzamento di Memoria Indiretto con Pre-Incremento/Decremento

- L'operando è un dato in memoria indirizzato indirettamente da un registro che viene preventivamente aggiornato per puntare alla locazione di memoria successiva/precedente
  - es. 68K (d0 e a0 sono registri della CPU):

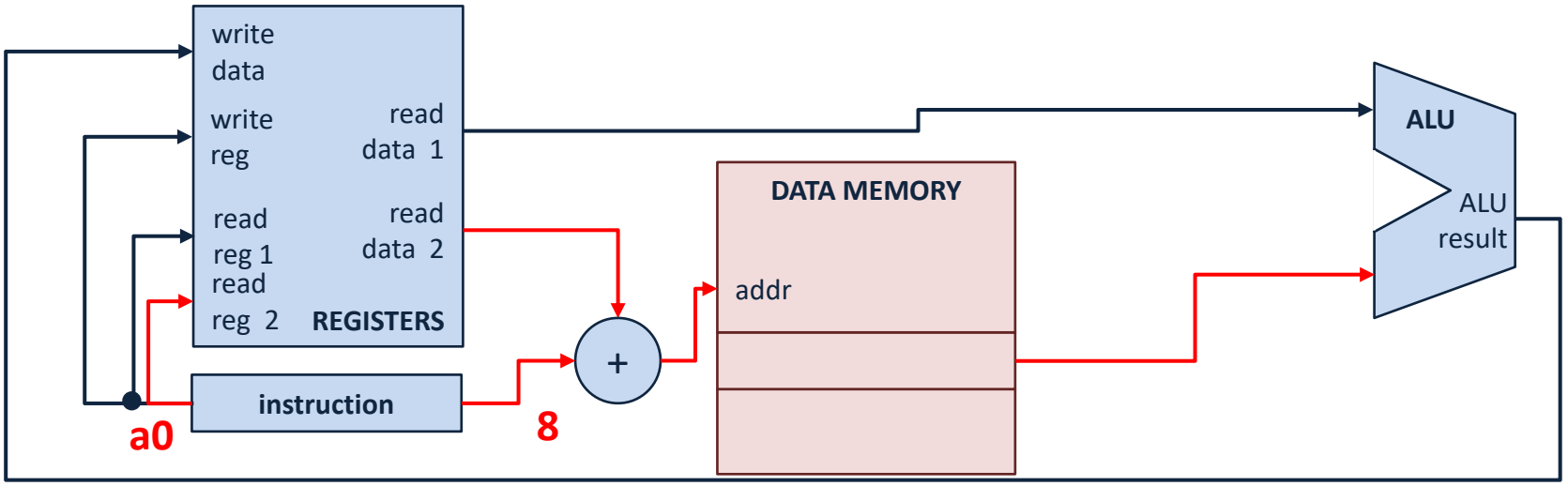
```
addw -(a0), d0  # a0 ← a0 - 2
                # d0 ← d0 + mem[a0]
```



Indirizzamento di Memoria Indiretto con Dislocamento

- L'operando è un dato in memoria indirizzato indirettamente da un registro a cui viene sommata una costante passata direttamente nell'istruzione (usato per strutture dati)
  - es. 68K (d0 e a0 sono registri della CPU):

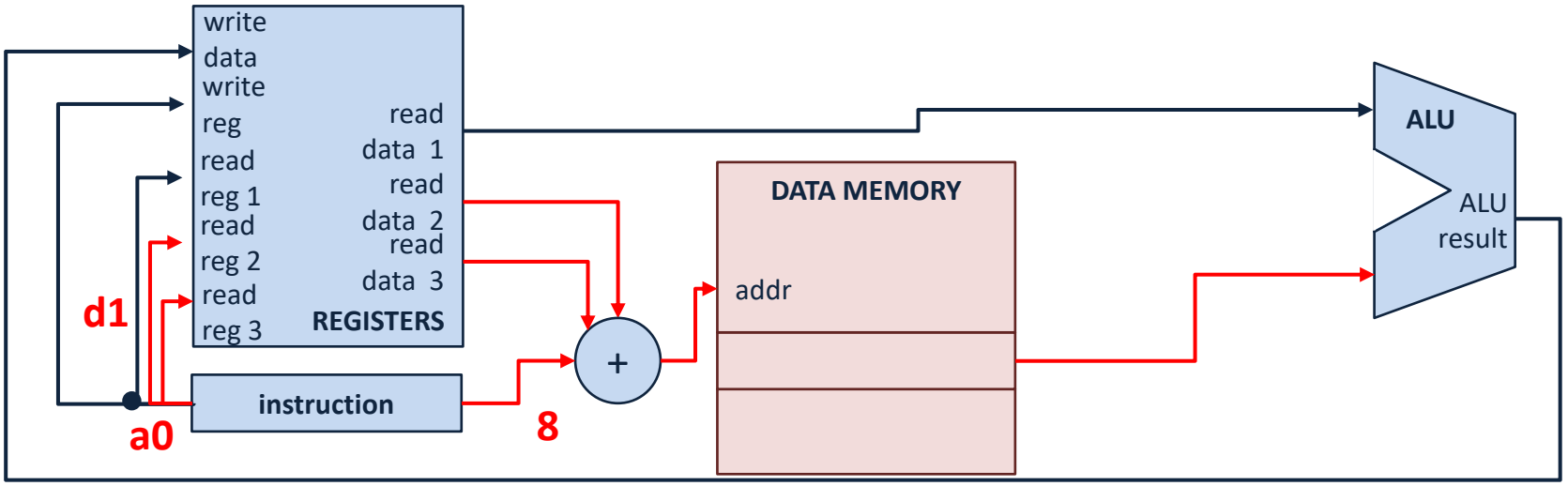
```
addw a0@(8), d0 # d0 ← d0 + mem[a0+8]
```



Indirizzamento di Memoria Indiretto con Indice e Dislocamento

- L'operando è un dato in memoria indirizzato indirettamente da un registro a cui vengono sommati un indice e una costante passati direttamente nell'istruzione
  - es. 68K (d0, d1 e a0 sono registri della CPU):

```
addw a0@(d1)8, d0 # d0 ← d0 + mem[a0+d1+8]
```



Modalità di Indrizzamento dell'ARMv6-M

- Immediate
  ```
  ADDI X22, X22, #4      ; X22 ← X22 + 4
  ```
- Banco Registri
  - diretto
    ```
    ADD X22, X22, X9     ; X22 ← X22 + X9
    ```
- Memoria Dati
  - indiretto
    - senza offset
      ```
      POP X22            ; X22 ← dmem[SP]
      ```
    - con offset
      - immediate
        ```
        LDR X22, [X9,#8]     ; X22 ← dmem[X9+8]
        ```
      - registro
        ```
        LDR X22, [X9,X10]    ; X22 ← dmem[X9+X10]
        ```

Modalità di Indrizzamento dell'ARMv6-M

- Memoria Istruzioni (imem[PC] è sempre l'istruzione seguente letta dalla memoria istruzioni)
  - diretto
  - indiretto
    - con offset

```
B #2000          ; PC ← 2000
B.EQ #20         ; PC ← PC+20
```

Modalità di Indrizzamento del LEGv8

- Immediate
  ```
  ADDI X22, X22 #4        ; X22 ← X22 + 4
  ```
- Banco Registri
  - diretto
    ```
    ADD X22, X22 X9       ; X22 ← X22 + X9
    ```
- Memoria Dati
  - indiretto
    - senza offset
      ```
      POP X22             ; X22 ← dmem[SP]
      ```
    - con offset
      - immediate
        ```
        LDUR X22, [X9,#8]     ; X22 ← dmem[X9+8]
        ```
      - registro
        ```
        LDUR X22, [X9,X10]    ; X22 ← dmem[X9+X10]
        ```

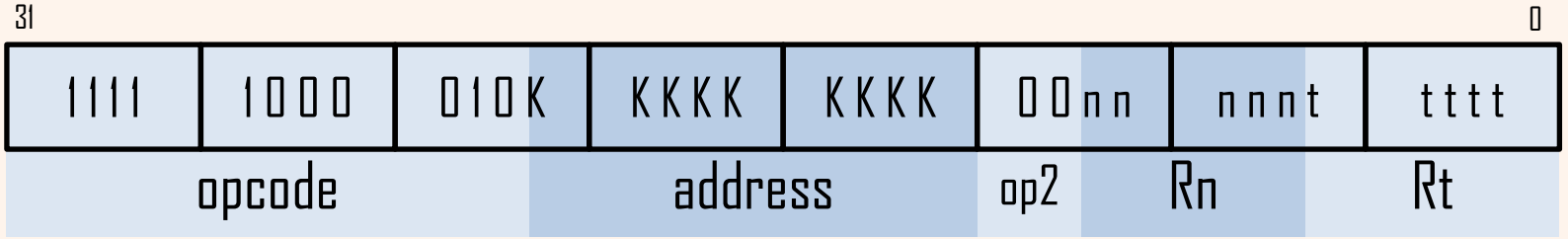
Modalità di Indrizzamento del LEGv8

- Memoria Istruzioni (imem[PC] è sempre l'istruzione seguente letta dalla memoria istruzioni)
  - diretto
    ```
    B #2000          ; PC ← 2000
    ```
  - indiretto
    - con offset
      ```
      CBZ #20        ; PC ← PC+20
      ```

Istruzioni di Trasferimento Dati del LEGv8

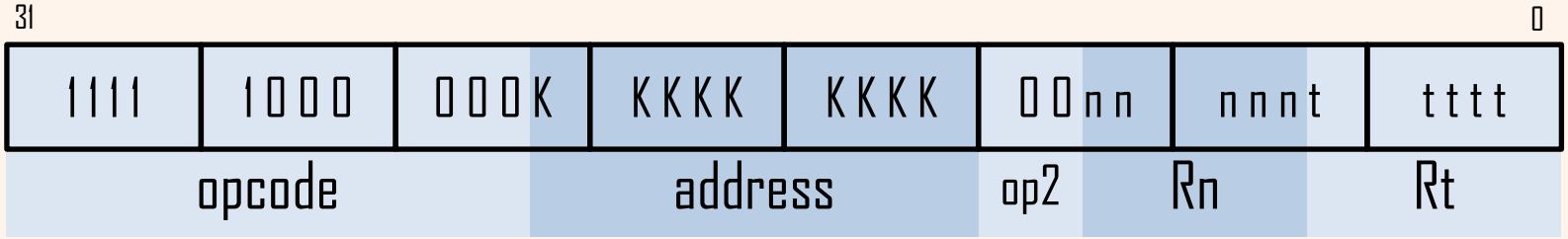
- **Tra Registri e Immediate**
  - caricamento di un immediate: MOVZ, MOVK
- **Tra Registri e Memoria**
  - trasferimento di parole (64 o 32 bit): LDUR, STUR, LDXR, STXR
  - trasferimento di parole (32 bit): LDURSW, STURSW
  - trasferimento di mezze parole (16 bit): LDURH, STURH
  - trasferimento di byte (8 bit): LDURB, STURB

Lettura di una Parola dalla Memoria



LDUR – load register	
sintassi	`LDUR Rt, [Rn, K]`
operazione	Rt ← dmem[Rn + K]
operandi	0 ≤ t,n ≤ 31; 0 ≤ K ≤ 512
flag interessati	-
cicli esecuzione	2
lunghezza istruzione	32 bit
tipo istruzione	R

Scrittura di una Parola nella Memoria



STUR – store register	
sintassi	`STUR Rt, [Rn, K]`
operazione	dmem[Rn + K] ← Rt
operandi	0 ≤ t,n ≤ 31; 0 ≤ K ≤ 512
flag interessati	-
cicli esecuzione	2
lunghezza istruzione	32 bit
tipo istruzione	D

Esempio: Trasferimento Dati

C

```
// architettura 32 bit (32 bit memory words)
// X20 ← g, X21 ← h, X22 ← addr(A)
g = h + A[8];
```

Esempio: Trasferimento Dati

C

```
// architettura 32 bit (32 bit memory words)
// X20 ← g, X21 ← h, X22 ← addr(A)
g = h + A[8];
```

assembly

```
LDUR X9, [X22,#32]   ; tmp0 ← A[8]
ADD X20, X21, X9     ; g ← h + A[8]
```

3 cicli

Esempio: Trasferimento Dati (2)

C

```
// architettura 32 bit (32 bit memory words)
// X20 ← g, X21 ← h, X22 ← addr(A)
A[12] = h + A[8];
```

Esempio: Trasferimento Dati (2)

C

```
// architettura 32 bit (32 bit memory words)
// X21 ← h, X22 ← addr(A)
A[12] = h + A[8];
```

assembly

```
LDUR X9, [X22,#32]   ; tmp0 ← A[8]
ADD X9, X21, X9      ; g ← h + A[8]
STUR X9, [X22,#48]   ; A[12] ← h + A[8]
```

5 cicli

Istruzioni di Salto nel LEGv8

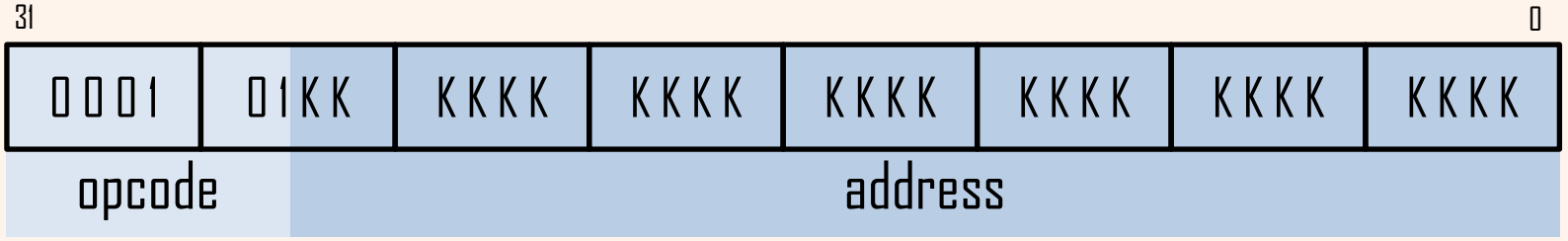
- **Salto Condizionato**
  - comparazione con 0: CBZ, CBNZ
  - con condizione sui bit di stato: B.cond
- **Salto Incondizionato**
  - senza salvataggio PC: B, BR
  - con salvataggio PC (chiamata a sub-routine): BL

Formati di Istruzione del LEGv8

- **Tipo B** (salto incondizionato, es. B K)

- **Tipo CB** (salto condizionato, es. CB K)

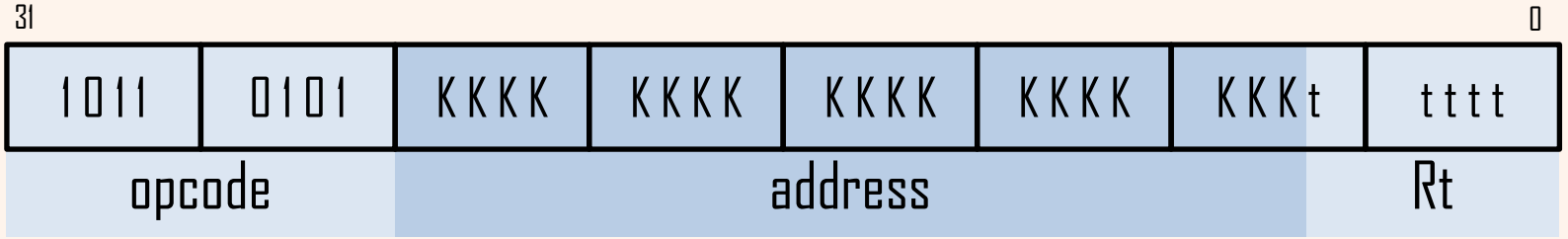
Salto Incondizionato



B – branch

sintassi	B K
operazione	PC ← PC + 4 + K*4
operandi	-32M ≤ K ≤ +32M-1
flag interessati	-
cicli esecuzione	2
lunghezza istruzione	32 bit
tipo istruzione	B

Salto Condizionato (se diverso da zero)



CBNZ – compare and branch on not equal 0

sintassi	`CBNZ Rt, K`
operazione	if(Rt!=0) PC ← PC + 4 + K*4
operandi	0 ≤ t ≤ 31; -256 k ≤ K* ≤ +256 k -1
flag interessati	-
cicli esecuzione	if (Rt!=0) then 2, else 1
lunghezza istruzione	32 bit
tipo istruzione	CB

* da 0 a 1040 nell'ARMv8

Esempio: Controllo di Flusso (1)

C

```
// X5 ← f, X6 ← g, X7 ← h, X8 ← i, X9 ← j
int f, g, h, i, j;  // 32 bit architecture
if (i == j)
      f = g + h;
else
      f = g - h;
```

Esempio: Controllo di Flusso (1)

C

```
// X5 ← f, X6 ← g, X7 ← h, X8 ← i, X9 ← j
int f, g, h, i, j;  // 32 bit architecture
if (i == j)
      f = g + h;
else
      f = g - h;
```

assembly

```
        SUB X0, X8, X9      ; tmp0 ← i - j
        CBNZ X0, Else       ; go to Else if i != j
        ADD X5, X6, X7      ; f = g + h
        B Exit              ; go to Exit
Else:   SUB X5, X6, X7      ; f = g - h
Exit:
```

4 o 5 cicli

Esempio: Controllo di Flusso (1)

C

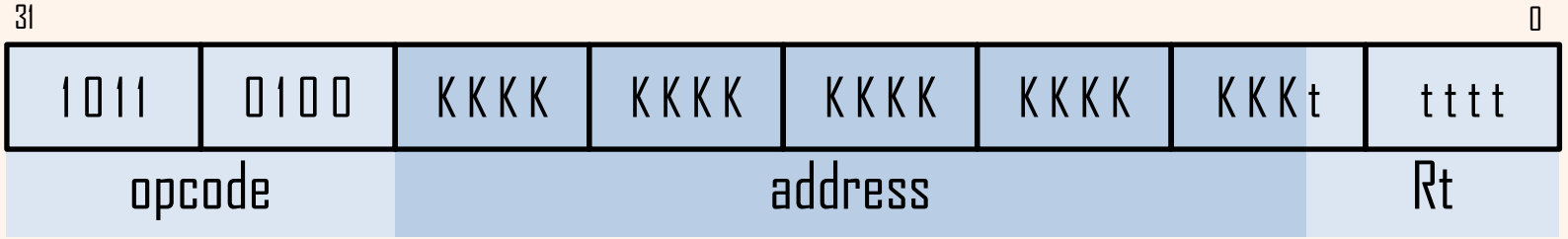
```
// X5 ← f, X6 ← g, X7 ← h, X8 ← i, X9 ← j
int f, g, h, i, j;  // 32 bit architecture
if (i == j)
      f = g + h;
else
      f = g - h;
```

assembly

```
        SUB X0, X8, X9      ; tmp0 ← i - j
        CBNZ X0, 2          ; go to Else if i != j
        ADD X5, X6, X7      ; f = g + h
        B 1                 ; go to Exit
Else:   SUB X5, X6, X7      ; f = g - h
Exit:
```

4 o 5 cicli

Salto Condizionato (se uguale a zero)

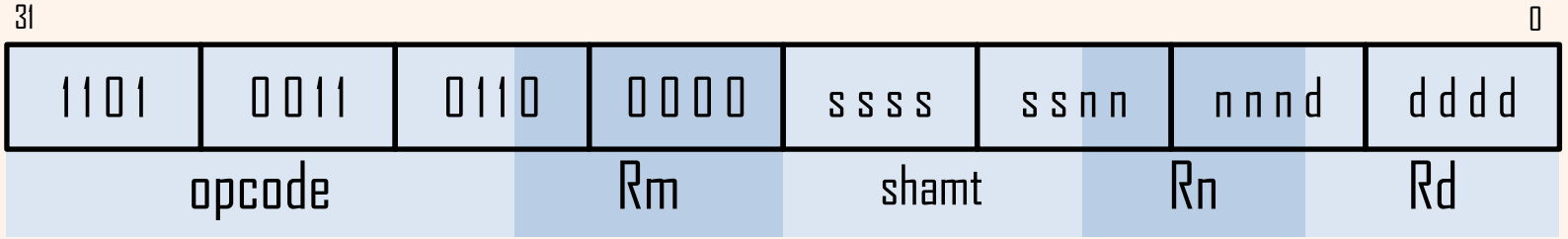


CBZ – compare and branch on equal 0

sintassi	`CBZ Rt, K`
operazione	if(Rt==0) PC ← PC + 4 + K*4
operandi	0 ≤ t ≤ 31; -256 k ≤ K* ≤ +256 k -1
flag interessati	-
cicli esecuzione	if (Rt==0) then 2, else 1
lunghezza istruzione	32 bit
tipo istruzione	CB

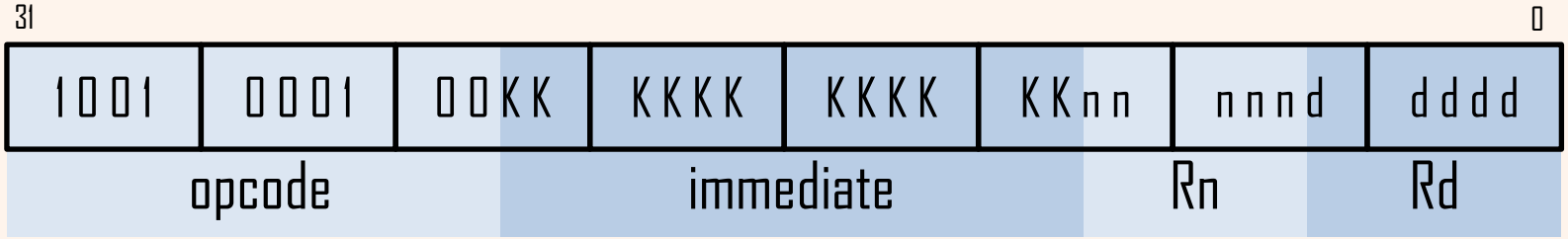
* da 0 a 1040 nell'ARMv8

Shift Verso Sinistra



LSL – logical shift left	
sintassi	`LSL Rd, Rn, s`
operazione	Rd ← Rn << s
operandi	0 ≤ d,n ≤ 31, 0 ≤ s ≤ 63
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	R

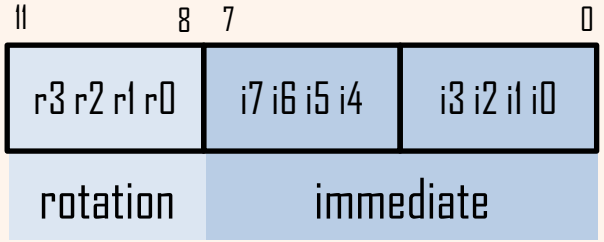
Somma con Immediate



ADDI – add immediate

sintassi	`ADDI Rd, Rn, K`
operazione	Rd ← Rn + K
operandi	0 ≤ d,n ≤ 31; 0 ≤ k ≤ 427819080 (rotate)
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	I

Immediate: Rotazioni per Archtitettura a 32 bit



r \ K	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0																									i7	i6	i5	i4	i3	i2	i1	i0
1	i1	i0																									i7	i6	i5	i4	i3	i2
2	i3	i2	i1	i0																									i7	i6	i5	i4
3	i5	i4	i3	i2	i1	i0																									i7	i6
4	i7	i6	i5	i4	i3	i2	i1	i0																								
5			i7	i6	i5	i4	i3	i2	i1	i0																						
6					i7	i6	i5	i4	i3	i2	i1	i0																				
7							i7	i6	i5	i4	i3	i2	i1	i0																		
8									i7	i6	i5	i4	i3	i2	i1	i0																
9											i7	i6	i5	i4	i3	i2	i1	i0														
A													i7	i6	i5	i4	i3	i2	i1	i0												
B															i7	i6	i5	i4	i3	i2	i1	i0										
C																	i7	i6	i5	i4	i3	i2	i1	i0								
D																			i7	i6	i5	i4	i3	i2	i1	i0						
E																					i7	i6	i5	i4	i3	i2	i1	i0				
F																							i7	i6	i5	i4	i3	i2	i1	i0		

Esempio: Controllo di Flusso (2)

C

```
// X22 ← i, X24 ← k, X25 ← addr(save)
int i, k; int[64] save;  // 32 bit architecture
while(save[i] != k)
      i += 1;
```

Esempio: Controllo di Flusso (2)

C

```
// X22 ← i, X24 ← k, X25 ← addr(save)
int i, k; int[64] save;  // 32 bit architecture
while(save[i] != k)
      i += 1;
```

assembly

```
Loop: ADDI X0, X31, #2    ; tmp0 = 2
      LSL X10, X22, X0    ; tmp1 = i<<2 = i*4
      ADD X10, X10, X25   ; tmp1 = addr(save[i])
      LDUR X9, [X10, #0]  ; tmp2 = save[i]
      SUB X11, X9, X24    ; tmp3 = save[i] - k
      CBZ X11, Exit       ; if tmp3==0 goto Exit
      ADDI X22, X22, #1   ; i += 1
      B Loop              ; go to Loop
Exit:
```

Esempio: Controllo di Flusso (2)

C

```
// X22 ← i, X24 ← k, X25 ← addr(save)
int i, k; int[64] save;  // 32 bit architecture
while(save[i] != k)
      i += 1;
```

assembly

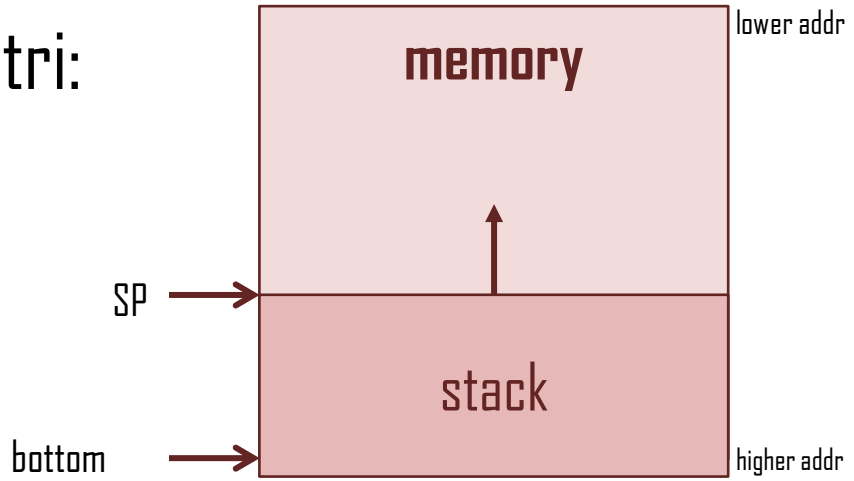
```
Loop: ADDI X0, XZR, #2     ; tmp0 = 2
      LSL X10, X22, X0     ; tmp1 = i<<2 = i*4
      ADD X10, X10, X25    ; tmp1 = addr(save[i])
      LDUR X9, [X10, #0]   ; tmp2 = save[i]
      SUB X11, X9, X24     ; tmp3 = save[i] - k
      CBZ X11, 2           ; if tmp3==0 goto Exit
      ADDI X22, X22, #1    ; i += 1
      B -8                 ; go to Loop
Exit:
```

Indice

- Generalità sui Microcontrollori
  - Definizione, campi applicativi e cenni storici
- Instruction Set Architecture
  - Istruzioni, tipi di dato e formato
  - Operandi
    - registri
    - memorie
  - Modalità di Indirizzamento
- Stack e funzioni

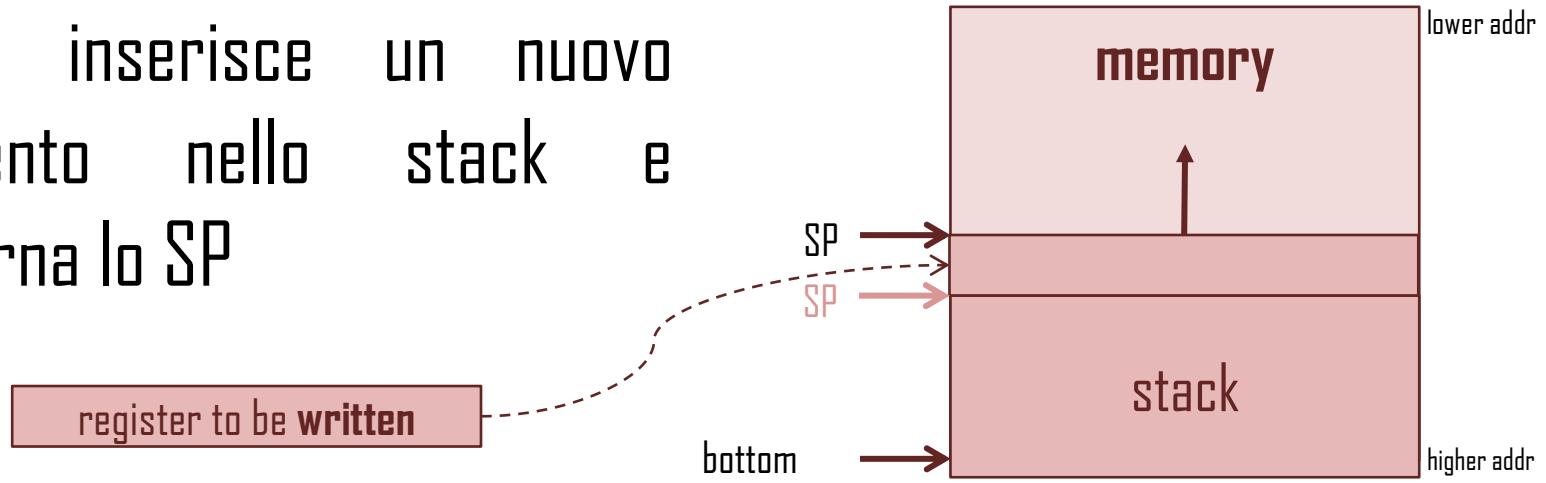
Stack

- Lo stack è una struttura dati in cui **l'ultimo elemento inserito è anche il primo ad uscire** (Last In First Out, LIFO)
- Lo stack solitamente **cresce a ritroso**, partendo da indirizzi alti verso quelli più bassi
- Lo stack ha almeno due parametri:
  - **bottom**, il fondo dello stack
  - **stack pointer** (SP), la cima dello stack

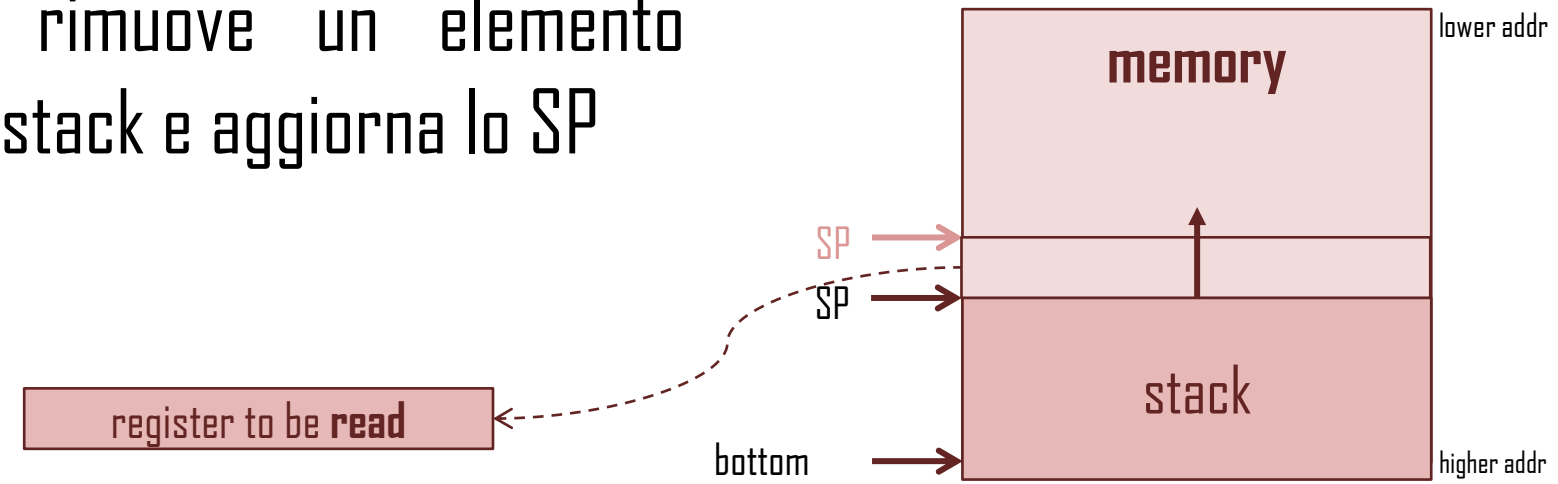


Stack Push e Pop

- **Push**: inserisce un nuovo elemento nello stack e aggiorna lo SP

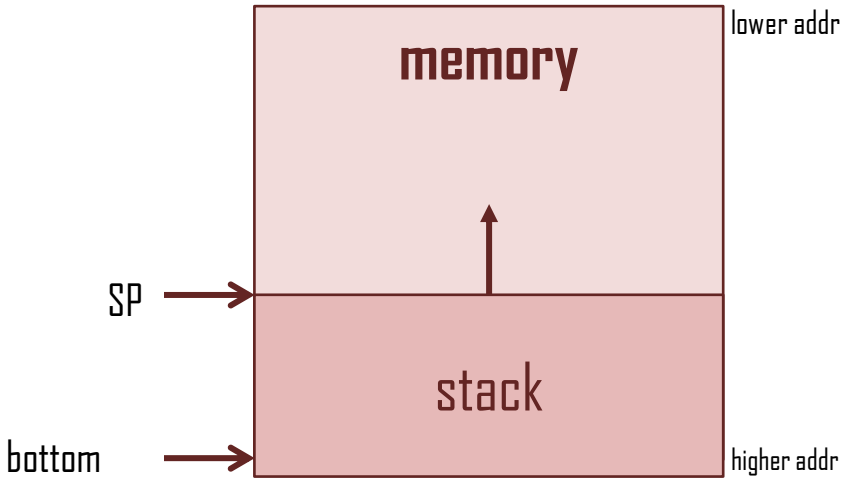


- **Pop**: rimuove un elemento dallo stack e aggiorna lo SP



Stack e Funzioni

- Lo stack è utilizzato nelle **chiamate a funzione e nelle sub-routine** per **preservare il contenuto dei registri** del processore
- Le **funzioni** sono molto utili nella programmazione per:
  - **scomposizione**
    - semplifica lo sviluppo
  - **modularità**
    - comprensibilità e manutenzione
  - **riutilizzo**
    - massimizza la produttività



Esempio di Funzione C

```
int pow (unsigned int b, unsigned int e){
      unsigned int i, p;
      p = 1;
      for(i=0;i<e;i++)
            p = p*b;
      return p;
}

int main(){
      unsigned int m, n;
      m = 2; n = 3;
      m = pow(m,n);
      return 0;
}
```

Esempio di Funzione C

```
int pow (unsigned int b, unsigned int e){
      unsigned int i, p;
      p = 1;
      for(i=0;i<e;i++)
            p = p*b;
      return p;
}

int main(){
      unsigned int m, n;
      m = 2; n = 3;
      m = pow(m,n);
      return 0;
}
```

chiamante (caller)

chiamato (callee)

parametri o argomenti (passing parameters)

valore di ritorno (return value)

Chiamate a Funzione

- Una chiamata a funzione comprende
  - **controllo di flusso tra chiamante e chiamato**
    - indirizzo di ritorno (all'istruzione da eseguire per il chiamante)
  - **passaggio di valori**
    - parametri e valori di ritorno
    - due convenzioni per le chiamate a funzione
      - **passaggio del valore corrente** di un parametro alla funzione chiamata
        - » non efficiente per le strutture e gli array
      - **passaggio di un riferimento**, o meglio dell'indirizzo del parametro corrente alla funzione chiamata
        - » efficiente anche per strutture e array

Esempio di Funzione C con Passaggio per Valore

```
void swap(int x, int y){
      int temp = x;
      x = y;
      y = temp;
}

int main(){
      int a=1, b=2;
      swap(a,b);
      printf("a=%d, b=%d", a, b);
      return 0;
}
```

il passaggio di parametri per valore **non consente al chiamante di alterare il chiamato** in quanto la modifica dei parametri x e y rimane locale rispetto alla funzione swap

Esempio di Funzione C con Passaggio per Riferimento

```
void swap(int* px, int* py){
      int temp = *px;
      *px = *py;
      *py = temp;
}

int main(){
      int a=1, b=2;
      swap(&a,&b);
      printf("a=%d, b=%d", a, b);
      return 0;
}
```

il passaggio di parametri per riferimento **consente al chiamante di alterare il chiamato** in quanto la memoria referenziata viene alterata

Conflitti nei Registri

- Se un **registro viene utilizzato sia dal chiamante che dal chiamato** e il chiamante ha **bisogno del vecchio valore dopo il ritorno** del chiamato, si ha un conflitto nei registri
- Il **compilatore** si occupa di gestire i conflitti nei registri
- I **registri in conflitto** vanno salvati all'interno dello **stack**
- Il **chiamante** o/e il **chiamato** si occupa/occupano di **salvare** i **registri in conflitto** nello **stack**

Passaggio di Parametri e Valori di Ritorno

- I **parametri** vanno passati attraverso dei **registri dedicati** e, se in conflitto, vanno memorizzati nello stack
- Il **valore di ritorno** va memorizzato in dei **registri dedicati**
- La porzione di stack occupata da una funzione, la cui dimensione dipende dalla funzione stessa, è detto **stack frame** e ha un puntatore associato chiamato **frame pointer (FP)** che punta alla base o alla cima del frame
- Lo stack frame viene **liberato quando la chiamata a funzione termina**

Contenuto di uno Stack Frame

- **Indirizzo di ritorno**
  - necessario quando la chiamata termina
- **Registri in conflitto**
  - da ripristinare quando la chiamata termina
  - un registro in conflitto è sempre il FP
- **Parametri e variabili locali** che non è possibile contenere nei registri

Identificazione Registri in Conflitto

- L'identificazione dei registri in conflitto può rivelarsi **complicata** in presenza di **funzioni annidate**
- Una possibile soluzione è quella di **salvare tutti i registri da preservare nello stack**, a prescindere dal fatto che essi siano effettivamente in conflitto o meno
- Tipicamente, i **livelli di ottimizzazione dei compilatori C** (-o0, -o1, -o2, -o3) **influiscono** sull'**identificazione dei registri in conflitto** e sui relativi salvataggi nello stack

Convenzioni di Chiamata

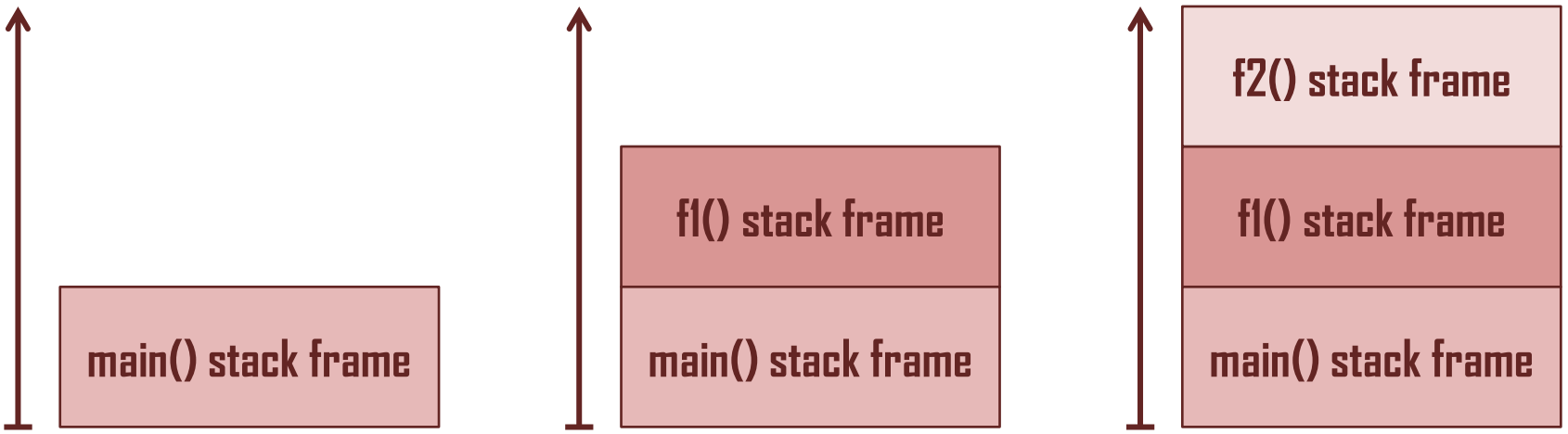
- Le **variabili locali e i parametri** hanno bisogno di essere memorizzati in **maniera contigua nello stack** per semplificarne l'accesso che avviene tramite stack o frame pointer (SP o FP)
- Occorre definire **l'ordine di memorizzazione delle variabili e dei parametri nello stack**
  - i compilatori C tipicamente li salvano in **ordine inverso** rispetto a **come essi appaiono nel programma**

Esempio di Contenuto dello Stack

```
int main(){
        ...
        f1();
        ...
}
```

```
int f1(){
        ...
        f2()
        ...
}
```

```
int f2(){
        ...
}
```



solo gli stack frame delle chiamate annidate si sommano tra di loro

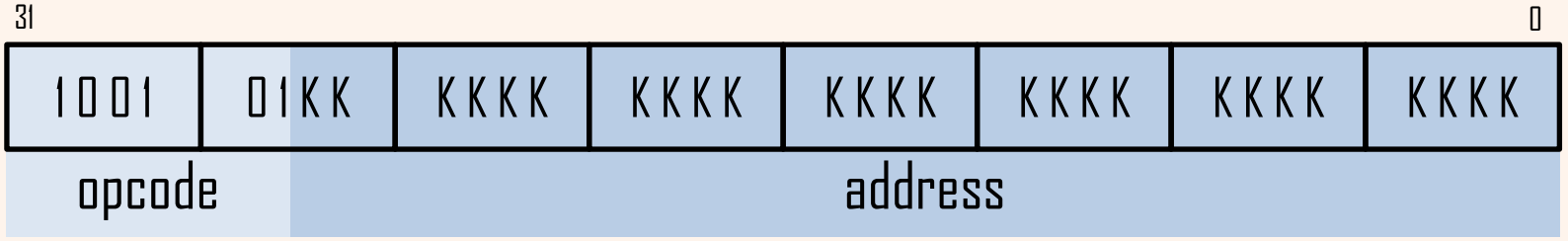
Stack nel LEGv8

- Nel LEGv8 lo stack è implementato come un **blocco di parole consecutive nella memoria dati**
- Lo **stack pointer (SP)** indica la **cima dello stack**, ovvero l'**indirizzo più basso**, dove andranno inseriti nuovi dati
- Nel LEGv8, così come nell'ARMv6-M, lo SP:
  - risiede nel banco di registri (**X28**)
  - punta all'**ultima locazione scritta** (la prima locazione che può essere scritta risulta dunque essere all'indirizzo SP-4)

Indirizzo di Ritorno nel LEGv8

- Nel LEGv8 è previsto un registro, il **Link Register (LR)**, dedicato alla memorizzazione dell'**indirizzo di ritorno** di una chiamata a funzione:
  - l'istruzione di salto incondizionato **Branch and Link (BL)** memorizza l'indirizzo dell'istruzione successiva del chiamante, ovvero l'indirizzo di ritorno, nel LR (**chiamata a funzione**)
  - l'istruzione di salto incondizionato **Branch to Register (BR)** consente di saltare all'istruzione il cui indirizzo è specificato in un registro (**ritorno da funzione** se il registro è LR)

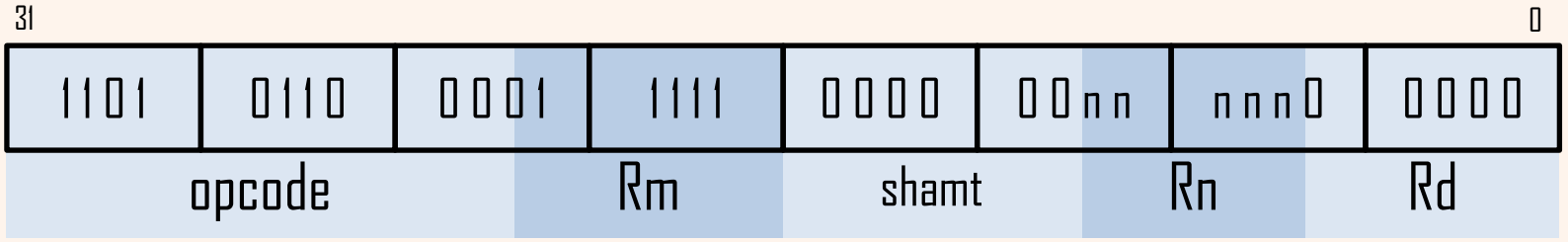
Salto con Collegamento



BL – branch with link

sintassi	`BL K`
operazione	LR ← PC + 4; PC ← PC + 4 + K*4;
operandi	-32M ≤ K ≤ +32M-1
flag interessati	-
cicli esecuzione	2
lunghezza istruzione	32 bit
tipo istruzione	B

Salto a Registro



BR – branch to register

sintassi	`BR Rn`
operazione	PC ← Rn
operandi	0 ≤ n ≤ 31
flag interessati	-
cicli esecuzione	2
lunghezza istruzione	32 bit
tipo istruzione	R

Registri e Funzioni nel LEGv8

- I registri **X0:X8** sono utilizzati per memorizzare i **parametri** e il **valore di ritorno** di una **funzione**
- I registri **X9-X18** sono **registri temporanei** che **non vengono preservati** nelle chiamate a funzione
- I registri **X19:X27** sono **registri salvati** che **vengono preservati** nelle chiamate a funzione
- I registri **SP** (X28), **FP** (X29) e **LR** (X30) sono **registri speciali** che **vengono preservati** nelle chiamate a funzione

Chiamata a Funzione nel LEGv8

- Chiamante
  - prima di chiamare il chiamato deve **memorizzare i parametri da passare nei registri dedicati** (X0:X8)
  - dopodiché può **chiamare il chiamato** attraverso l'istruzione Branch and Link (**BL**)
- Chiamato
  - **prologo** (prologue)
    - **riserva spazio** per lo **stack frame** della funzione aggiornando lo stack pointer
    - **memorizza** nello stack (**push**) lo stack frame che comprende: indirizzo di ritorno (LR, già memorizzato dal chiamante), registri in conflitto (compreso il FP) eventuali parametri o variabili locali non memorizzabili nei registri (per via del numero elevato)

Chiamata a Funzione nel LEGv8

- Chiamato
  - **corpo della funzione** (body)
    - **effettua** normalmente **l'operazione** della funzione sullo stack frame e sui registri interessati
    - memorizza il **valore di ritorno** nei registri dedicati (X0:X8)
  - **epilogo** (epilogue)
    - **rimuove** lo **stack frame** della funzione aggiornando lo stack pointer
    - **ripristina** l'indirizzo di ritorno (LR), i registri in conflitto, e gli eventuali parametri o variabili locali memorizzati nello stack frame (**pop**)
    - **ritorna al chiamante** attraverso l'istruzione Branch to Register (**BR**) con argomento il link register (LR)

Esempio: Chiamata a Funzione (1)

C

```
// X19 ← f, X0 ← g, X1 ← h, X2 ← i, X3 ← j
// 32 bit architecture
int leaf_example (int g, int h, int i, int j) {
      int f;
      f = (g + h) - (i + j);
      return f;
}
```

Esempio: Chiamata a Funzione (1)

C

```c
// X19 ← f, X0 ← g, X1 ← h, X2 ← i, X3 ← j
// 32 bit architecture
int leaf_example (int g, int h, int i, int j) {
      int f;
      f = (g + h) - (i + j);
      return f;
}
```

assembly

```
SUBI SP, SP, #12      ; adjust SP to push 3 items
STUR X21, [SP, #8]    ; save saved reg X21
STUR X20, [SP, #4]    ; save saved reg X20
STUR X19, [SP, #0]    ; save saved reg X19
...
```

Esempio: Chiamata a Funzione (1)

assembly

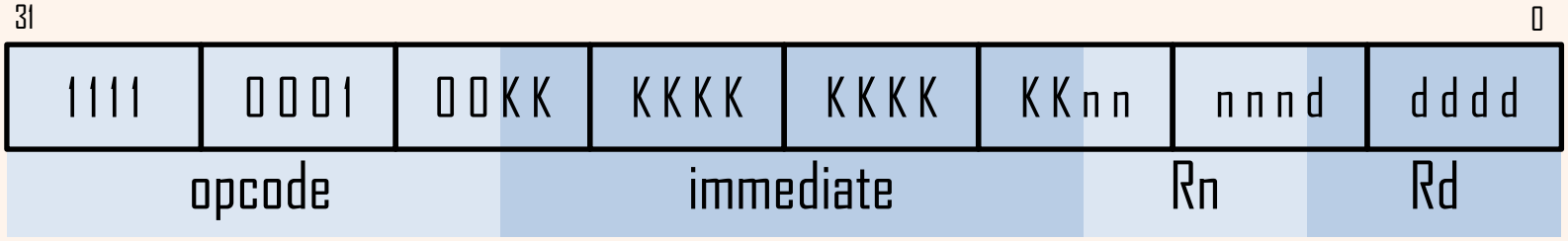
```
...
ADD X20, X0, X1        ; X20 ← g + h
ADD X21, X2, X3        ; X21 ← i + j
SUB X19, X20, X21      ; f ← (g + h) − (i + j)
ADD X0, X19, XZR       ; move f to return reg

LDUR X19, [SP, #0]     ; restore saved reg X19
LDUR X20, [SP, #4]     ; restore saved reg X20
LDUR X21, [SP, #8]     ; restore saved reg X21
ADDI SP, SP, #12       ; adjust SP to pop 3 items

BR LR                  ; return to the caller
```

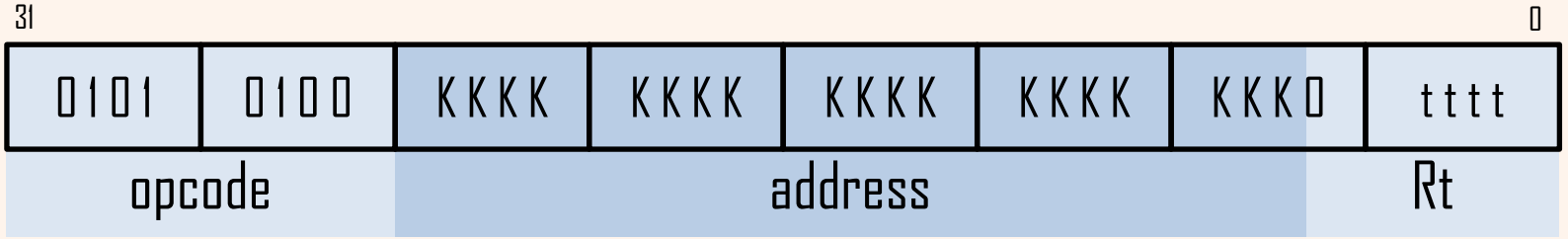
20 cicli

Sottrazione con Immediate e Settaggio Flag



SUBIS – subtract immediate and set flags	
sintassi	`SUBIS Rd, Rn, K`
operazione	Rd ← Rn – K
operandi	0 ≤ d,n ≤ 31; 0 ≤ k ≤ 427819080 (rotate)
flag interessati	N, Z, C, V
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	I

Salto Condizionato Generico



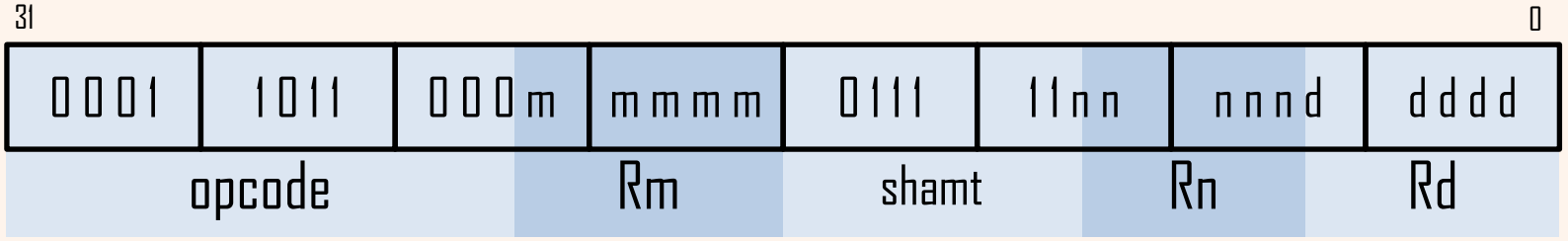
B.cond – conditional branch

sintassi	`B.cond K`
operazione	if(cond) PC ← PC + 4 + K*4
operandi	0 ≤ t ≤ 15; -256 k ≤ K ≤ +256 k -1
flag interessati	-
cicli esecuzione	if (cond) then 2, else 1
lunghezza istruzione	32 bit
tipo istruzione	CB

Salto Condizionato Generico

code	meaning	APSR flags tested (cond)
eq	equal	Z==1
ne	not equal	Z==0
cs or **hs**	unsigned higher or same (or carry set)	C==1
cc or **lo**	unsigned lower (or carry clear)	C==0
mi	negative (minus)	N==1
pl	positive or zero (plus)	N==0
vs	signed overflow (V set)	V==1
vc	no signed overflow (V clear)	V==0
hi	unsigned higher	(C==1) && (Z==0)
ls	unsigned lower or same	(C==0) || (Z==1)
ge	signed greater than or equal	N==V
lt	signed less than	N!=V
gt	signed greater than	(Z==0) && (N==V)
le	signed less than or equal	(Z==1) || (N!=V)
al (or **omitted**)	always executed (unconditional)	none tested

Moltiplicazione



MUL – multiply	
sintassi	`MUL Rd, Rn, Rm`
operazione	Rd ← Rn * Rm
operandi	0 ≤ d,n,m ≤ 31
flag interessati	-
cicli esecuzione	1
lunghezza istruzione	32 bit
tipo istruzione	R

Esempio: Chiamata a Funzione (2)

C

```
// X0 ← n, 32 bit architecture
int fact (int n) {
      if (n < 1)       return 1;
      else             return (n * fact(n-1));
}
```

Esempio: Chiamata a Funzione (2)

C

```c
// X0 ← n, 32 bit architecture
int fact (int n) {
      if (n < 1)        return 1;
      else              return (n * fact(n-1));
}
```

assembly

```
fact:SUBI SP, SP, #8        ; push 2 items
     STUR LR, [SP, #4]      ; save LR
     STUR X19, [SP, #0]     ; save saved reg

     SUBIS X9, X0, #1       ; tmp0 = n - 1
     B.GE L1                ; if n >= 1 go to L1
     ...
```

Esempio: Chiamata a Funzione (2)

assembly

```
        ...
        ADDI X1, XZR, #1      ; return 1

End:    LDUR X19, [SP, #0]    ; restore saved reg
        LDUR LR, [SP, #4]     ; restore LR
        ADDI SP, SP, #8       ; pop2 items
        BR LR                 ; return to caller

L1:     ADDI X19, X0, #0      ; tmp0 ← n
        SUBI X0, X0, #1       ; new argument n-1
        BL fact               ; call fact (n-1)
        MUL X1, X19, X1       ; return n*fact(n-1)
        B End                 ; go to End
```

Esempio: Chiamata a Funzione (2)

C

```c
// X0 ← n, 32 bit architecture
int fact (int n) {
      if (n < 1)        return 1;
      else              return (n * fact(n-1));
}
```

assembly

```
fact:SUBI SP, SP, #8        ; push 2 items
     STUR LR, [SP, #4]      ; save LR
     STUR X19, [SP, #0]     ; save saved reg

     SUBIS X9, X0, #1       ; tmp0 = n - 1
     B.GE 5                 ; if n >= 1 go to L1
     ...
```

Esempio: Chiamata a Funzione (2)

assembly

```
      ...
      ADDI X1, XZR, #1      ; return 1

End:  LDUR X19, [SP, #0]    ; restore saved reg
      LDUR LR, [SP, #4]     ; restore LR
      ADDI SP, SP, #8       ; pop2 items
      BR LR                 ; return to caller

L1:   ADDI X19, X0, #0      ; tmp0 ← n
      SUBI X0, X0, #1       ; new argument n-1
      BL -13                ; call fact (n-1)
      MUL X1, X19, X1       ; return n*fact(n-1)
      B -9                  ; go to End
```

Instruction Set LEGv8: Arithmetic

instruction	example	meaninng	comments
ADD	ADD X1, X2, X3	X1 = X2 + X3	
SUB	SUB X1, X2, X3	X1 = X2 - X3	
ADDI	ADDI X1, X2, K	X1 = X2 + K	
SUBI	SUBI X1, X2, K	X1 = X2 – K	
ADDS	ADDS X1, X2, X3	X1 = X2 + X3	APSR flags set
SUBS	SUBS X1, X2, X3	X1 = X2 - X3	APSR flags set
ADDIS	ADDIS X1, X2, K	X1 = X2 + K	APSR flags set
SUBIS	SUBIS X1, X2, K	X1 = X2 – K	APSR flags set
MUL	MUL X1, X2, X3	X1 = X2 * X3	X1 lower 32 bits
SMULH	SMULH X1, X2, X3	X1 = X2 * X3	X1 higher 32 bits (signed)
UMULH	UMULH X1, X2, X3	X1 = X2 * X3	X1 higher 32 bits (unsigned)
SDIV	SDIV X1. X2. X3	X1 = X2 / X3	signed division
UDIV	UDIV X1. X2. X3	X1 = X2 / X3	unsigned division

Instruction Set LEGv8: Data Transfer

instruction	example	meaninng	comments
LDUR	LDUR X1, [X2, K]	X1 = mem[X2 + K]	single word (64 bit)
STUR	STUR X1, [X2, K]	mem[X2 + K] = X1	single word (64 bit)
LDURSW	LDURSW X1, [X2, K]	X1 = mem[X2 + K]	single word (32 bit)
STURSW	STURSW X1, [X2, K]	mem[X2 + K] = X1	single word (32 bit)
LDURH	LDURH X1, [X2, K]	X1 = mem[X2 + K]	half word (16 bit)
STURH	STURH X1, [X2, K]	mem[X2 + K] = X1	half word (16 bit)
LDURB	LDURB X1, [X2, K]	X1 = mem[X2 + K]	byte (8 bit)
STURB	STURB X1, [X2, K]	mem[X2 + K] = X1	byte (8 bit)
LDXR	LDXR X1, [X2, 0]	X1 = mem[X2]	1st hald of atomic swap
STXR	STXR X1, X3, [X2]	mem[X2] = X1; X3 = 0 or 1	2nd hald of atomic swap
MOVZ	MOVZ X1, K	X1 = K << 2^n	16 bit constant, rest zeroes
MOVK	MOVK X1, K	X1 = K << 2^n	16 bit constant, rest unchanged

Instruction Set LEGv8: Logical

instruction	example	meaninng	comments
AND	AND X1, X2, X3	X1 = X2 & X3	
ORR	ORR X1, X2, X3	X1 = X2 | X3	inclusive OR
EOR	EOR X1, X2, X3	X1 = X2 ^ K	exclusive OR
ANDI	ANDI X1, X2, #K	X1 = X2 & K	
ORRI	ORRI X1, X2, #K	X1 = X2 | K	inclusive OR
EORI	EORI X1, X2, #K	X1 = X2 ^ K	exclusive OR
LSL	LSL X1, X2, X3	X1 = X2 << X3	
LSR	LSR X1, X2, X3	X1 = X2 >> X3	

Instruction Set LEGv8: Branch

instruction	example	meaninng	comments
CBZ	CBZ X1, K	if(X1==0) PC = PC + 4 + K	
CBNZ	CBNZ X1, K	if(X1!=0) PC = PC + 4 + K	
B.cond	B.cond K	if(cond) PC = PC + 4 + K	
B	B K	PC = PC + 4 + K	
BR	BR X1	PC = X1	return from procedure
BL	BL K	X30 = PC + 4 PC = PC + 4 + K	call procedure

Instruction Set LEGv8: Condizioni

code	meaning	APSR flags tested (cond)
eq	equal	Z==1
ne	not equal	Z==0
cs or **hs**	unsigned higher or same (or carry set)	C==1
cc or **lo**	unsigned lower (or carry clear)	C==0
mi	negative (minus)	N==1
pl	positive or zero (plus)	N==0
vs	signed overflow (V set)	V==1
vc	no signed overflow (V clear)	V==0
hi	unsigned higher	(C==1) && (Z==0)
ls	unsigned lower or same	(C==0) || (Z==1)
ge	signed greater than or equal	N==V
lt	signed less than	N!=V
gt	signed greater than	(Z==0) && (N==V)
le	signed less than or equal	(Z==1) || (N!=V)
al (or **omitted**)	always executed (unconditional)	none tested

Esercizi

- Compilazione

Compilazione (1)

- Ricavare un **codice assembly** LEGv8 capace di implementare la seguente funzione C rispettando le convenzioni di **chiamata a funzione** (tralasciare la gestione del registro FP)

C

```
// X0 ← pixel, X0 ← return, 32 bit arch.
char clip (int pixel) {
      if (pixel > 255){                  return 255;
      } else if (pixel < 0){   return 0;
      } else {
            return (char) pixel;
      }
}
```

Compilazione (2)

- Ricavare un **codice assembly** LEGv8 capace di implementare la seguente funzione C rispettando le convenzioni di **chiamata a funzione** (tralasciare la gestione del registro FP)

C

```
// X0 ← base, X1 ← fact, X3 ← return, 32 bit arch.
int mult(int base, int fact) {
      if(fact > 1) {
            return base + mult(base, fact-1);
      } else {
            return base;
      }
}
```