

Sistemi Embedded

Lezione 1 - 28/09/2021

- Modalità d'esame
- Introduzione ai sistemi embedded
- IoT (Internet of Things)
- Vulnerabilità (STUXNET)

- Segnali I/O
- Input asincroni

Lezione 2 - 30/09/2021

- Glitches
- Processori Embedded
- Microcontrollori
- Processori DSP

Lezione 3 - 05/10/2021 (da vedere su Google Drive)

Lezione 4 - 07/10/2021

- Esempio di processamento
- PLC (Programmable Logic Controllers)
- GPU
- Architettura x86
- Edge Computing e AI
- Modelli di programmazione
 - Parallelismo e Concorrenza
 - Analisi del dataflow

Lezione 5 - 12/10/2021

- Problemi time-bound (CPU Optimization) - Introduzione e tecniche di gestione
 - Out-of-order execution
 - Control hazard
 - Delayed branch
 - Interlock
 - Speculative execution
- Data and control Hazard
- Instruction level parallelism
 - Implementazione CISC
 - Subword parallelism
 - Architetture Very Long Instruction Word (VLIW)
 - Architetture Multicore
 - Fixed-point numbers

Livello ISA

- Address decoding

Lezione 6 - 14/10/2021

- ISA ARM32 (Advanced RISC Machine)
- ARM32 in dettaglio
- Accesso dei registri utilizzando istruzioni ARM
- PSR (Program Status Register)

Lezione 7 - 19/10/2021

- ARM32 (continua..)
- Program Counter (R15)
- Gestione delle eccezioni e il "vettore tabella"

Pipeline ARM32
Esecuzione condizionale
Esempi vari di condizioni
Istruzioni di branch

Lezione 8 - 21/10/2021
Condition field (continua...)
Branch e Branch with Link
Data processing instruction
Operazioni aritmetiche
Comparazione
Operatori Logici
Data Movement

Lezione 9 - 26/10/2021
Barrel Shifter
Istruzioni di moltiplicazione
Moltiplicazioni-Long e Moltiplicazioni-Long-Accumulate
Istruzioni Load/store
Offset dal Base Register
Modalità d'indirizzamento
Lettura in memoria (little-endian big-endian)

Lezione 10 - 27/10/2021
Esempio di codice con il post-indexing
Block Data Transfer
Stack
Operazioni dello stack
Stack e Subroutine
funzionalità dirette del block data transfer

Esercizi

Lezione 11 - 02/11/2021
Esercitazione ARM
Esercizio 1
Esercizio 2
Esercizio 3
Esercizio 5 (A)
Esercizio 5 (B)
Esercizio 5 (C)
Esercizio 5 (D)
Esercizio sul caricamento "a blocchi"
Altre operazioni in ARM
SWAP e SWAP byte
SWI (software interrupt)
PSR transfer instruction
Data processing instruction encoding
Branch instruction encoding

Lezione 12 - 04/11/2021
ARM Programming
Embedded Software Development (ripasso)
Compilazione
Interprete
Calling convention (caratteristica dell'EABI)

Lezione 13 - 16/11/2021
Esempi di codice (ARM ASM, RASPBERRY PI, GNU/LINUX)
GDB

Lezione 14 - 18/11/2021
Altri esempi di codice ARM
Dispositivi "targets"

Lezione 15 - 23/11/2021
Dispositivi target (continua)

Memory Mapping delle periferiche

Programmazione Bare metal

Lezione 16 - 25/11/2021

Bare Metal programming - esempi

Lezione 17 - 30/11/2021

Periferiche & I/O

I/O pins

Periferiche I/O

General - Purpose I/O (GPIO)

Timer

LED lampeggianti utilizzando il system timer

Lezione 18 - 02/12/2021

Configurazione GPIO

Interconnessione Seriale

Tipologia di trasporto nella comunicazione seriale

UART / USART Nel Raspberry Pi

Standard FORTH

Lezione 19 - 07/12/2021

Linguaggio Forth

Definizioni varie

Operatore modulo

Lezione 20 - 09/12/2021

Linguaggio Forth (parte 2)

Utilizzo delle costanti

LED lampeggianti in Forth

Esercizio

Double Length Variable and Costants

ESERCIZIO

Lezione 21 - 14/12/2021

FORTH Language (continua)

Array

Byte Array

Esempio

Implementazioni FORTH

Implementazioni del dizionario

Condizionale in FORTH

FORTH Data flow

Geografia di FORTH

Istruzione CREATE

Lezione 22 - 16/12/2021

FORTH (continua)

Operatori aritmetici

Prodotto-Rapporto

Return stack

Operazioni in memoria

DUMP

Lezione 23 - 21/12/2021

Hardware examples

GPIO

Ambienti di programmazione

Architettura

Mappatura della memoria

Hardware abstraction layers

Connattività Arduino

Lezione 23/12/2021

Serial Peripheal interconnect

I2C BUs

Procedure di comunicazione

Slow Start byte
Indirizzamento a 10 bit
SSD1306 OLED display
RPI HDMI display

(gli appunti sono soggetti a variazione)

Lezione 1 - 28/09/2021

L'obiettivo di questa materia è sviluppare software che permetta di gestire hardware "embedded" (dispositivi con interazione utente minima). I linguaggi più utilizzati saranno:

- Linguaggio macchina e linguaggio assembly (ARM)
- Linguaggio di basso livello interattivo ed esplorativo (Forth)
- Linguaggio ad alto livello, per sviluppo "embedded"

Proveremo (e riusciremo, si spera) a produrre il software dal più basso livello al più alto

Modalità d'esame

Ad ogni studente è richiesto di sviluppare un'applicazione "embedded" funzionante e ben documentata. Il progetto sarà discusso durante l'orale (quest'ultimo riguarda anche gli argomenti visti nel corso)

Introduzione ai sistemi embedded

La maggior parte dei computer sono molto meno visibili. Possono:

- gestire motori, airbag, freni e sistemi audio
- possono codificare digitalmente la tua voce e costruire un segnale radio trasferito dal tuo telefono alla stazione radio più vicina
- controllare il tuo forno a microonde, frigorifero e lavastoviglie
- gestire stampanti da uso domestico o da uso industriale
- gestire robot da terreno industriale, generatori di centrali elettriche, processamenti in centrali chimiche e semafori
- costruire immagini degli organi umani e misurare parametri vitali
- controllare aerei e treni
- eseguire compiti "invisibile"

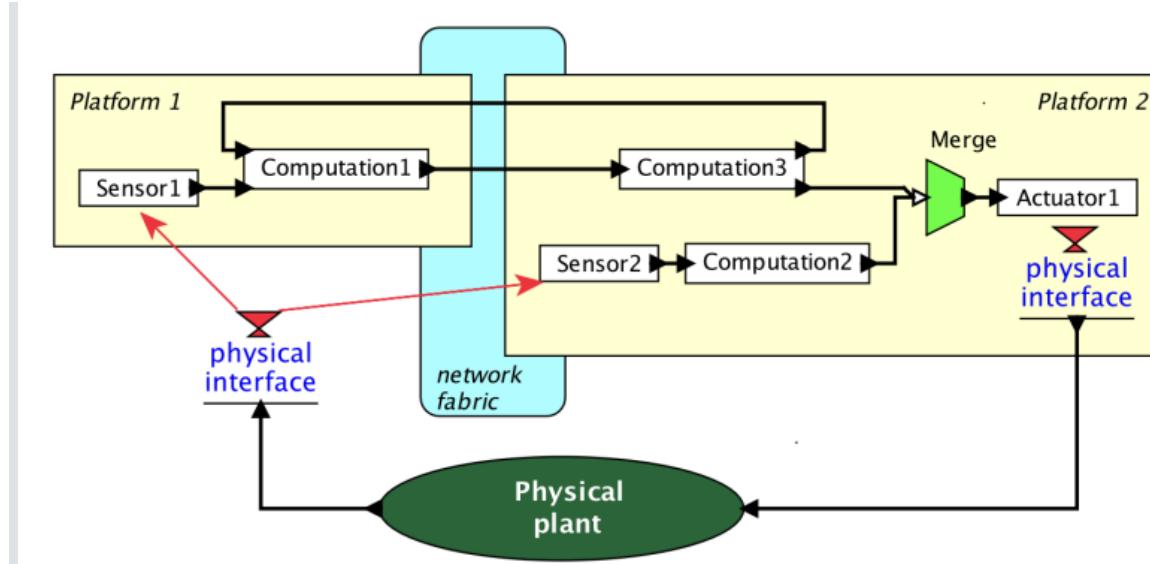
questi computer "meno visibili" sono detti "sistemi embedded", e il software che eseguono è chiamato **embedded software**. Solitamente, i dispositivi di uso quotidiano utilizzano general-purpose CPU che esegue altrettanti software anch'essi general-purpose. Nella programmazione orientata all'hardware embedded **la potenza non è tutto**, vi sono dei principi da rispettare nel design del software e nella sua realizzazione: il primo principio è la **correttezza** della computazione, il secondo è sicuramente **l'ottimizzazione** delle risorse per far sì che il dispositivo sia il più responsivo possibile.

I sistemi embedded sono utilizzati sin dal passato (seppur non molto remoto): già a partire dal 1970 si utilizzavano piccoli computer, i quali avevano (a causa del periodo) pochissime **risorse** (scarsa potenza computazionale, memoria piccola, energia limitata, ecc..); ovviamente, l'ottimizzazione era la parola chiave nella produzione del software. Recentemente, superata la

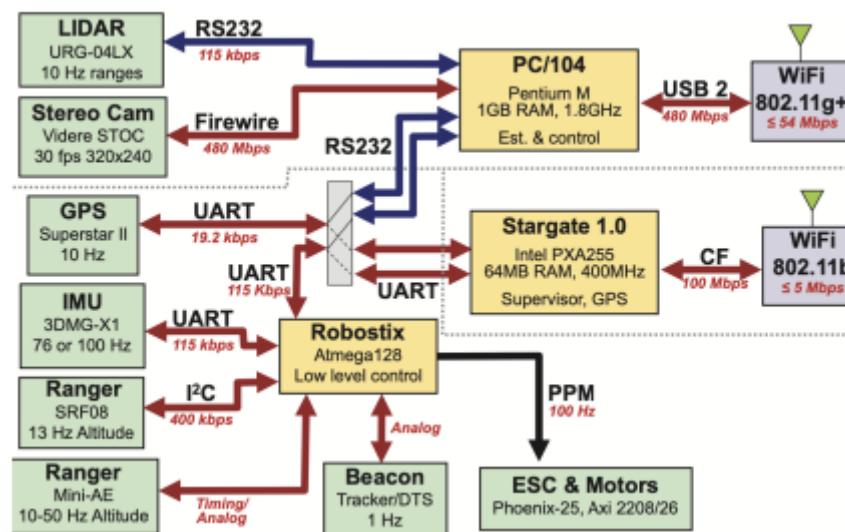
necessità del risparmio di risorse, le principali sfide dei sistemi embedded sono state reindirizzate sull'interazione tra il primo e i processi fisici (gestione dei sensori); nell'insieme, tali agglomerati di dispositivi interagenti con l'esterno vengono chiamati **CPS**

CPS : Cyber Physical Systems

nello schema dei CPS i sensori ed i monitor di rete comunicano (e di conseguenza, gestiscono) con l'esterno tramite dei **feedback loop** dove il mondo fisico influisce sul comportamento delle computazione dei dispositivi digitali e vice versa



Esempio specifico (quadro rotore STARMAC):

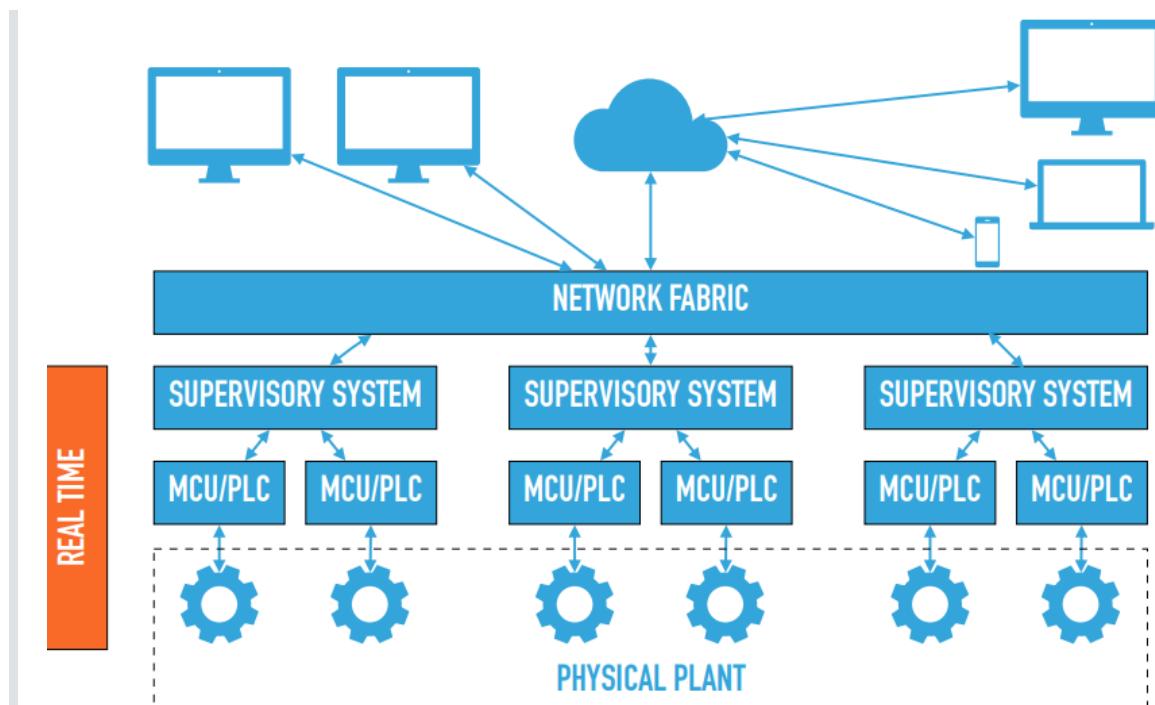


(il software viene eseguito nei box gialli)

Come spesso accade, le istruzioni software vengono eseguite **proceduralmente**: ovvero vi sono degli step **sequenziali**. Nel mondo fisico, però, i processi non sono procedurali ma **concorrenti** e non sono pochi. La sfida principale è cercare di conciliare la **semantica sequenziale** con cui solitamente vengono scritte le istruzioni, con i processi del mondo fisico (sempre **concorrenti**), realizzando e analizzando software ad-hoc per ciò

IoT (Internet of Things)

Negli ultimi anni è venuta in risalto una "nube" di dispositivi connessi ad internet che fa da "ponte fisico" al mondo digitale. Oggi la maggior parte dei sensori ed attuatori possiedono **microprocessori**, **interfacce di rete** e **software** che abilita **l'accesso remoto ai dati del sensore** e **controllo remoto** degli attuatori. La maggior parte dei dispositivi IoT non sono adatti a svolgere controlli **real-time** e ad essere utilizzati come dispositivi di **sicurezza "critici"**: il motivo è banale, più componenti ha un sistema embedded, più **latenza** possiede..per questo è necessario che tali dispositivi, specie in usi industriali, siano fisicamente il più **minimali possibili**. I designer di software embedded devono avere a che fare con **controller interrupt**, **architetture di memoria**, **programmazione a basso livello**, **design di driver per hardware**, **interfacce network** e **strategie di scheduling**.



Vulnerabilità (STUXNET)

The three-staged attack was based on several then unknown zero day vulnerabilities

- One of the 0-day vulnerabilities involved management of shared printers in Windows

Compromised digital signing certificates were also used to let the software gain access to systems



1. infection

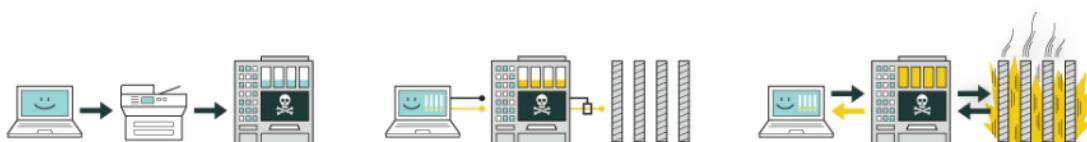
Stuxnet enters a system via a USB stick and proceeds to infect all machines running Microsoft Windows. By brandishing a digital certificate that seems to show that it comes from a reliable company, the worm is able to evade automated-detection systems.

2. search

Stuxnet then checks whether a given machine is part of the targeted industrial control system made by Siemens. Such systems are deployed in Iran to run high-speed centrifuges that help to enrich nuclear fuel.

3. update

If the system isn't a target, Stuxnet does nothing; if it is, the worm attempts to access the Internet and download a more recent version of itself.



Segnali I/O

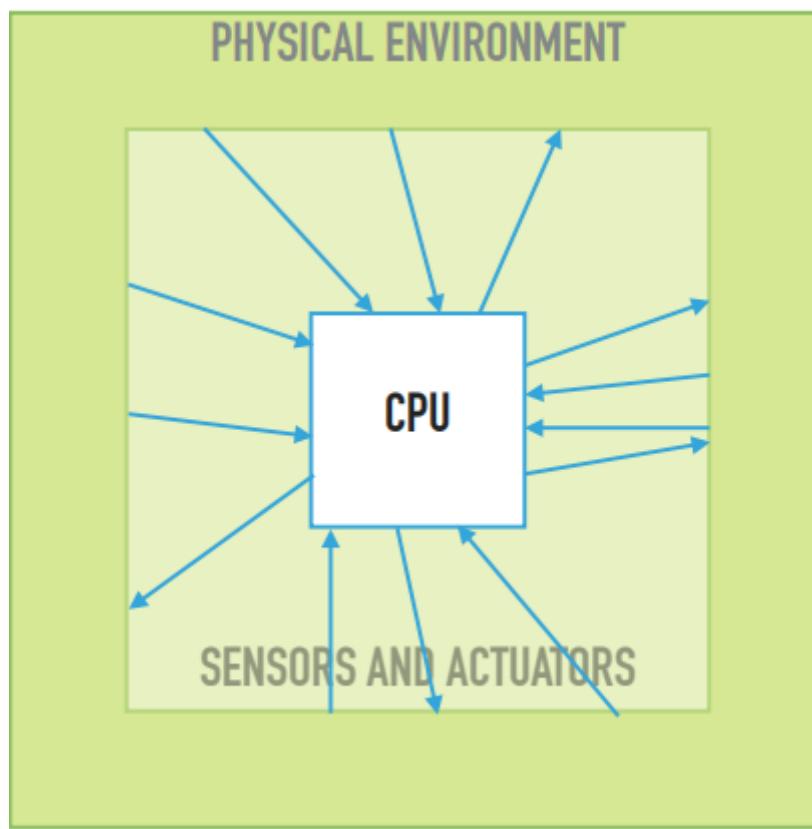
Una CPU Embedded scambia informazioni con l'ambiente fisico tramite i **segnali I/O**.

Ovviamente, tali segnali possono essere sia **digitali** che **analogici**. Dov'è la differenza?

- Un segnale digitale assume un valore **discreto** (solitamente 0 o 1, TRUE o FALSE, SET o CLEAR) oppure dei valori di tensione (0V o 5V, oppure 0V o 3.3V)
 - Questi segnali possono essere raggruppati, formando un **bus** di dati
- Un segnale analogico viene pre-elaborato da una circuiteria specializzata che lo "discretizza" (solitamente il valore TRUE corrisponde ad HIGH e il FALSE a LOW). Per esempio un dispositivo **ADC** (Analog-To-Digital Converter) si occupa di ciò

Dei segnali analogici possono essere prodotti tramite dispositivi **DAC** (Digital-To-Analog Converter) o modulando l'output digitale (modulazione sigma-delta, modulazione di larghezza d'impulso)

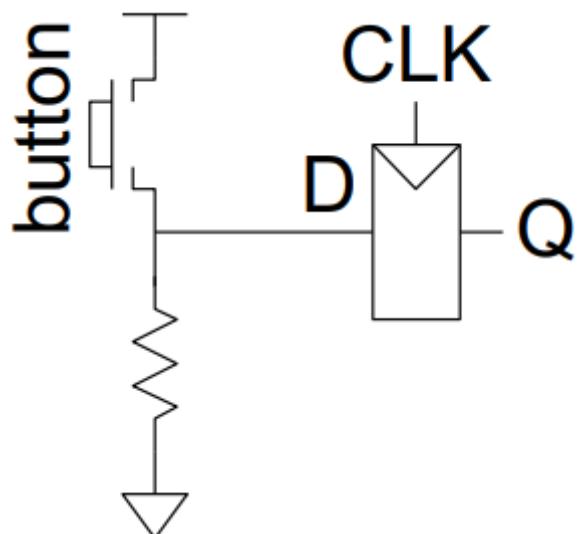
Quantità fisiche, come **temperatura**, **umidità**, **luminosità**, ecc.., vengono misurate e trasformate dai **sensori** in input digitale da inviare alla CPU. La maggior parte dei sensori sono in grado di individuare un cambio nella quantità fisica misurata (chiamato anche **event**) e generare un segnale d'input alla CPU. Come già anticipato, i segnali sono **asincroni** e quindi possono raggiungere in qualsiasi momento la CPU; i circuiti I/O devono essere in grado di gestire segnali asincroni e **scatenare** (triggering) una risposta rapida alla CPU, tramite interrupts o circuiti specializzati (come i contatori)



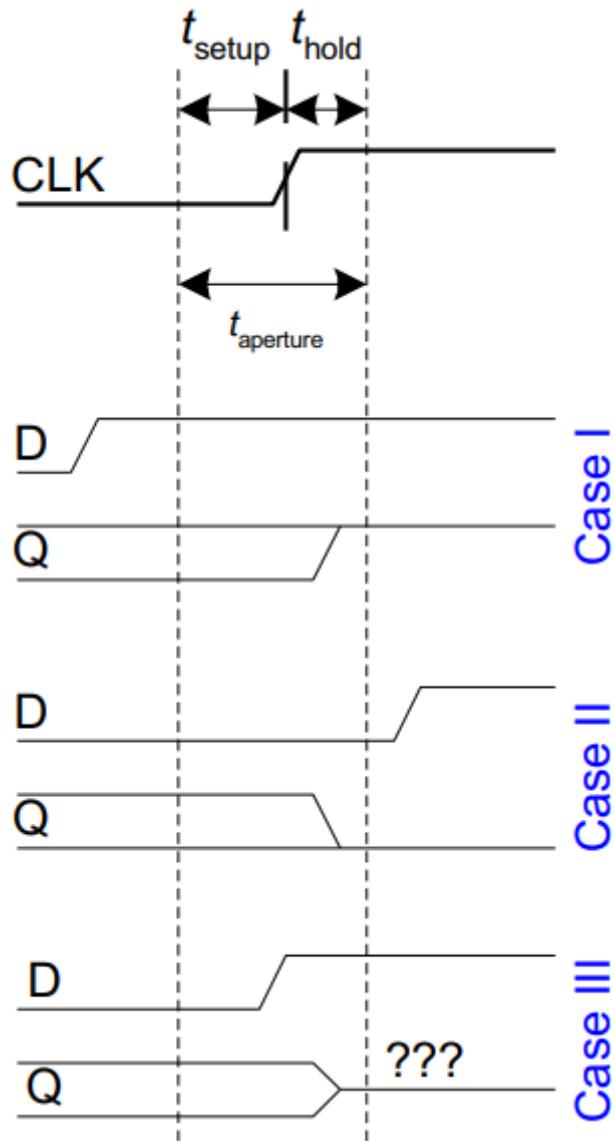
I segnali in output possono essere utilizzati per controllare dispositivi come display, motori, casse audio, servosterzi ed altri attuatori. Vengono utilizzati, molto spesso, dei dispositivi chiamati **driver**, i grado di trasformare i segnali output in bassa potenza in segnali corrispondenti alle caratteristiche elettriche richieste dagli attuatori. Altri dispositivi chiamati **controller**, "alleggeriscono" il carico della CPU, generando al suo posto pattern di segnali **complessi** per controllare dispositivi output e attuatori (esistono, ad esempio, display controller). Generalmente, tutti i dispositivi connessi ad un processore in grado di gestire segnali in input e produrre segnali output sono chiamate **periferiche I/O**.

Input asincroni

Tipicamente, i sistemi digitali sono costruiti con circuiti logici sincroni (flip-flop), poiché vi è un risparmio di costo nell'implementazione; i segnali vengono considerati validi solo se vi è un segnale di sincronizzazione che "scandisce" il tempo (clock). Tipicamente, però, gli input sono generalmente asincroni e i circuiti logici sono per natura analogici



Il tempo misurato dall'essere umano è, ovviamente molto più largo rispetto a quello misurato da un microcontrollore (si parla, nel microcontrollore, nell'ordine di millisecondi o meno): questo può generare problemi nell'interazione umano-dispositivo. Ipotizziamo di premere il tasto prima che la prossima "finestra" si possa aprire, il nostro registro cambierà il suo output fin quando vi è un clock che "scandisce" la premuta del tasto da parte dell'umano.



(casi di premuta del tasto in relazione al clock e all'uscita)

- Caso 1: D su ALTO prima del fronte di salita del clock
- Caso 2: D su BASSO prima del fronte di salita del clock
- Caso 3: D su ALTO in corrispondenza al fronte di salita del clock

Cosa succede (CASO 3 IMMAGINE) se lo stato non è ne 0 ne 1? Questo **stato** viene chiamato "metastabile" a differenza degli stati 0,1 (stabili) e può essere propagato in un'altra sequenza, influenzando il funzionamento

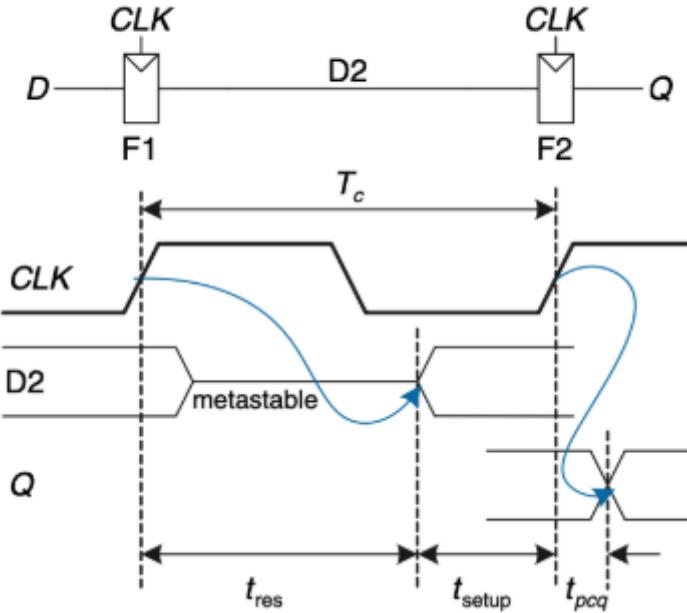
Formula di stato di metastabilità:

$$P(t_{\text{res}} > t) > \frac{T_0}{T_c} e^{-t/\tau}$$

Lezione 2 - 30/09/2021

Glitches

La misura della resilienza di un sistema che gestisce eventi asincroni è il Mean Time Between Failures (MTBF)



nell'immagine possiamo osservare l'uscita D2 in uno stato "metastabile" e questo comporta un problema: si deve utilizzare un sincronizzatore in grado di riportare tale segnale in uno dei due stati permessi. Solitamente prima di "aggiustare" tale stato, bisogna aspettare un tempo chiamato t_{setup} per aspettare il prossimo fronte di salita del CLK. Il procedimento per calcolare il MTBF è il seguente

$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c-t_{\text{setup}}}{t}}$ -> con questo termine viene calcolato il numero di "fallimenti" (ovvero quante volte si passa in metastabilità)

$$\frac{P(\text{failure})}{\text{sec}} = N \frac{T_0}{T_c} e^{-\frac{T_c-t_{\text{setup}}}{t}}$$

$$MTBF = \frac{1}{NP(\text{failure})/\text{sec}} = \frac{T_c}{NT_0} e^{-\frac{T_c-t_{\text{setup}}}{t}}$$

Se vogliamo avere sistemi affidabili per anni, questo MTBF dev'essere di grande dimensione.

Processori Embedded

Nella computazione general-purpose, la varietà di set d'istruzioni architetturali sono limitate, e l'architettura x86 di Intel la fa da padrona. Invece, nella computazione embedded non c'è questa supremazia: vi sono diverse architetture ed ognuna di esse è adatta alla produzione di dispositivi di vario genere, in base al loro **specifico funzionamento**: questi dispositivi non devono eseguire funzioni arbitrarie con software definito dall'utente. La loro specificità permette dei vantaggi assai rilevanti in termini di consumi: essi tendono a poter essere utilizzati tramite piccole quantità di energia per grosse quantità di tempo. Così come i processori general-purpose, i processi embedded posseggono il loro **ISA** (Instruction Set Architecture) che definisce le istruzioni eseguibili nativamente. Alcuni di questi sono ARM32 e x86 sono degli ISA.

gli ISA in realtà non contengono vincoli temporali sull'esecuzione delle istruzioni: queste possono essere eseguite correttamente su varie tipologie di processori ma non possiamo avere riscontro della loro esecuzione rispetto al tempo

Il vantaggio di utilizzare lo stesso ISA in una famiglia di processori è la disponibilità di strumenti software, solitamente costosi da sviluppare, che possono essere condivisi e possibilmente gli stessi programmi possono essere eseguiti correttamente in diversi dispositivi. Per quanto riguarda il tempo di esecuzione, non abbiamo modo di controllare questo tramite compilatori: necessitiamo di infrastrutture/software ad alto livello o di circuiti hardware adibiti al conteggio del tempo.

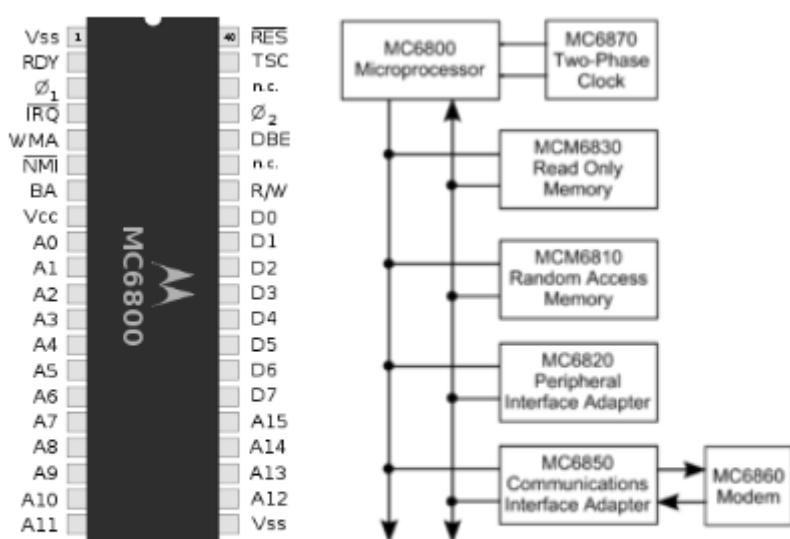
Vista la grande quantità di applicazioni embedded, possiamo avere diversi tipi di processori embedded differenti per potenza, dimensione e funzionalità.

Microcontrollori

Un microcontrollore è un piccolo computer in un singolo circuito integrato, specializzato nello svolgere "piccoli compiti" e consiste in una relativamente semplice CPU combinata con dispositivi periferici come memorie, dispositivi I/O e timer.

più della metà dei processori venduta globalmente appartiene alla categoria dei microcontrollori

I più semplici microcontrollori operano in **word di 8-bit** e sono adatti per applicazioni che richiedono piccole quantità di memoria e funzioni logiche semplici e consumano piccole quantità di energia (spesso possiedono delle modalità idle e di riposo). Tali dispositivi vengono utilizzati come sensori di nodi di rete o in sistemi di videosorveglianza.



Motorola 6800

I microcontrollori possono essere abbastanza sofisticati. Basti pensare ad Intel Atom, utilizzato nei netbook o in altri piccoli computer: questi processori sono ideati per ottimizzare l'utilizzo di energia senza però sacrificare troppa potenza in relazione ai processori utilizzati in computer di fascia più alta. Possiamo elencare altri microcontrollori (a 8 bit) come il Motorola 6800 e l'intel 8080, apparsi nel mercato nel 1974.



Zilog Z80

Processori DSP

molte applicazioni embedded fanno un po' di elaborazione di segnale. Un segnale non è nient'altro che una collezione di misurazioni campionate del mondo fisico, tipicamente prese ad una frequenza costante chiamata **frequenza di campionamento**. Un'applicazione di motion control, ad esempio, potrebbe aver bisogno di leggere la posizione o la locazione tramite sensori a **frequenze di campionamento** che vanno da pochi Hertz a poche centinaia di Hertz; i segnali audio sono campionati a frequenze che oscillano dagli 8000 Hz ai 44.1 kHz; applicazioni ad ultrasuoni operano a maggiori frequenze; i segnali video possono richiedere 25 o 30 Hz per dispositivi comuni o frequenze più alte per misurazioni particolari; i software di gestione di onde radio necessitano di frequenze dai pochi Hz ai GHz

Lezione 3 - 05/10/2021 (da vedere su Google Drive)

Lezione 4 - 07/10/2021

Esempio di processamento

In questa lezione vedremo i problemi che vanno affrontati nella scrittura di codice in CPU scalabili. Gli elementi importanti da tenere in considerazione sono:

1. La scelta del linguaggio di alto livello che sia il più efficiente possibile
2. Ottimizzazione del codice

Discuteremo anche dei tempi di esecuzione, che possono essere variabili in base ai tipi di dispositivi in cui viene eseguito il codice. Possiamo visualizzare degli esempi di codice scritto per determinati dispositivi.

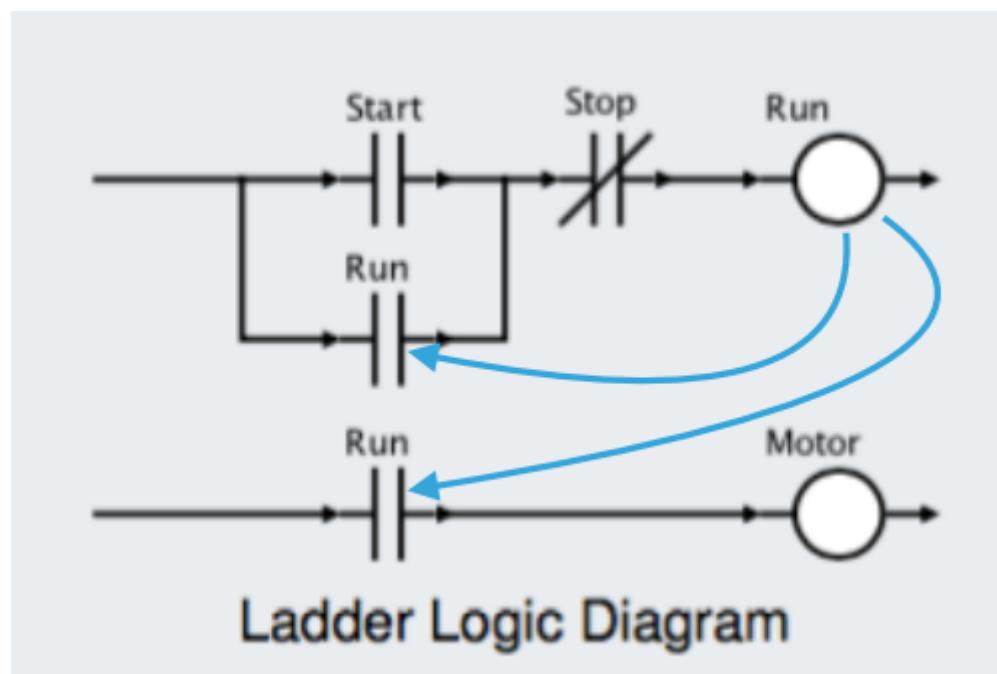
Il codice mostrato nelle slide esegue 100 milioni di calcoli in un FIR (finite input response, vedi paragrafo 8.1.2 libro) a 8 fasi; viene misurato il tempo passato dall'esecuzione del codice e il tempo di processamento per campione. Il codice mostra anche le differenze d'implementazione tra l'utilizzo dei **movimenti di memoria**, atti (presumibilmente) a migliorare il tempo di esecuzione, e il non utilizzo di essi

""*I professore mostra codice e dà consigli sulle flag da inserire nell'esecuzione: se vengono inserite, l'implementazione senza "movimenti di memoria" impiega meno tempo rispetto alla prima; se invece non vengono inserite, il contrario*""

I tempi di esecuzione non sono mai gli stessi e per natura non sono prevedibili; dobbiamo limitare quest'ultima limitando superiormente questi tempi.

PLC (Programmable Logic Controllers)

I PLC sono stati utilizzati dalle industrie per molto tempo, sin da quando venivano utilizzati dispositivi elettro-meccanici (relè, pulsanti, ecc..). Questi dispositivi sono programmati utilizzando una **notazione grafica** chiamata **ladder logic diagram** la quale si riferisce al PLC originale costruito con dispositivi elettromeccanici (relè e switch). Un **contatto** o switch è rappresentato da **due barre verticali** mentre il cerchio rappresenta il **collo** (relè connesso a un dispositivo).



nella figura, il **feedback** nella parte superiore mantiene il circuito chiuso anche dopo aver rilasciato il tasto di start, mentre il tasto stop interrompe il feedback

i PLC moderni sono costruiti utilizzando MCU al posto di relè e switch.

GPU

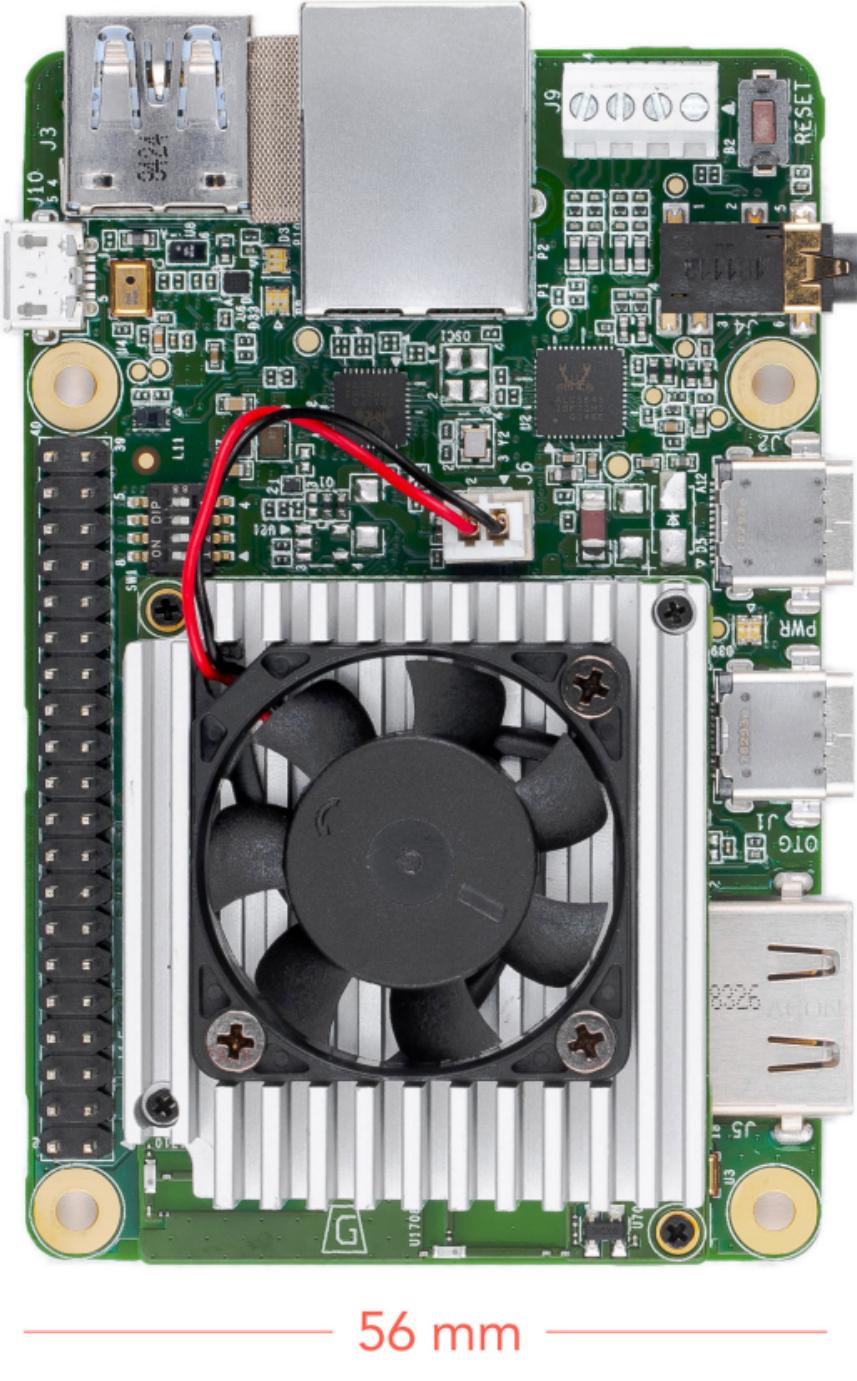
Una graphical process unit è un processore specializzato e progettato per effettuare calcoli richiesti nel rendering grafico. Questi processori vengono utilizzati dagli anni 70, quando erano utilizzati per visualizzare testo e grafica, per combinare molteplici pattern grafici e per disegnare figure geometriche. Le GPU moderne supportano grafica 3D, shading e video rendering; nel mercato dominano le GPU Intel, NVIDIA e AMD.

Architettura x86

è l'ISA dominante per pc desktop e laptop. Nasce con il processore Intel 8086, un processore a 16bit progettato da Intel nel 1978; una variante dell'8086, l'8088, fu utilizzata da IBM nel suo originale IBM PC. Tutti i processori successivi ottennero i nomi con numeri finali "86" e generalmente mantenevano la retrocompatibilità.

Edge Computing e AI

è un micromputer atto a svolgere operazioni di accelerazione grafica (Google TPU ML accellerator coprocessor) e gestione di crittografia (possiede un coprocessore) con un consumo irrilevante di energia (le specifiche sono 8GB di eMMC, da 1 a 4GB LPDDR4 di RAM, modulo Wi-Fi 2x2 MIMO e un processore ARM Cortex-A53 Quad Core + Cortex-M4F)



85 mm

56 mm

Modelli di programmazione

Un programma embedded necessita di **monitorare e reagire in eventi concorrenti multipli** e **simultaneamente controllare più dispositivi di output** che interagiscono con il mondo fisico; quindi, i programmi embedded sono quasi sempre programmi **concorrenti** in quanto, per loro natura e per la loro fondamentale utilità nei CPS, devono gestire e rispondere ad eventi in maniera **real-time**. Vengono proposti linguaggio di programmazione nuovi e in maniera continua per "reinventare la ruota": molti dei linguaggi di alto livello non sono totalmente adatti alla gestione di limiti severi della programmazione embedded e tutt'ora i linguaggi più adatti sono C/C++. Nuovi e vecchi paradigmi di programmazione ma rinnovati possono beneficiare la programmazione embedded e i CPS.

Parallelismo e Concorrenza

Un programma è detto **concorrente** se parti **diverse** di esso vengono **concettualmente eseguite simultaneamente**; Un programma è **parallelo** se parti diverse di esso vengono **fisicamente eseguite simultaneamente in hardware distinti** (macchine multicore, server in una server farm, o microprocessori distinti). I **programmi non-concorrenti** invece, sono definiti come sequenze di operazioni eseguite in ordine; queste sequenze vengono scritte in linguaggio imperativo come il C, poi tradotto in linguaggio macchina dai compilatori.

Analisi del dataflow

Un compilatore potrebbe analizzare le dipendenze tra operazioni in un programma e produrre codice parallelo; molte CPU oggi supportano esecuzione parallela utilizzando stream multipli o architetture VLIW (very long instruction word) e inoltre viene implementato dell'hardware, sempre all'interno della CPU, capace d'individuare al volo istruzioni indipendenti nel flusso del programma ed eseguire più istruzioni in maniera simultanea, ottimizzando l'esecuzione del codice.

Esempio di codice

- ▶ The two C assignment statements are independent and thus could be executed in parallel or in reverse order without changing the behavior of the program
- ▶ The compiler however optimized code to perform 3 multiplications instead of 4

```
#include <stdio.h>
#include <stdlib.h>

int
main(){
    double a, b, c;

    a=rand()*0.001;
    b=a*a;
    c=a*a*a;

    printf("a^2=%f, a^3=%f\n", b, c);

    return 0;
}
```

```
bl _rand
scvtf d0, w0
Lloh0:
    adrp x8, LCPI0_0@PAGE
Lloh1:
    ldr d1, [x8, LCPI0_0@PAGEOFF]
    fmul d0, d0, d1
    fmul d1, d0, d0
    fmul d0, d0, d1
    str d1, [sp]
    str d0, [sp, #8]
Lloh2:
    adrp x0, l_.str@PAGE
Lloh3:
    add x0, x0, l_.str@PAGEOFF
    bl _printf
```

Un sistema operativo multitasking potrebbe sovrapporre l'esecuzione di task multiple in un unico flusso sequenziale di istruzioni da far eseguire a una singola unità

TASK	PU
Instr ₁	Instr ₁
Instr ₂	Instr ₂
Instr ₃	Instr ₃
Instr ₄	Instr ₃
Instr ₅	Instr ₃
Instr ₆	Instr ₃
...	
Instr _n	Instr ₄
...	
...	
...	
...	

Cycle	PU
1	Instr ₁
2	Instr ₂
3	Instr ₃
4	Instr ₃
5	Instr ₃
6	Instr ₃
7	Instr ₄
8	Instr ₅
9	Instr ₅
10	Instr ₅
11	Instr ₆
...	...

NON-CONCURRENT PROGRAM (SINGLE TASK)
ON SINGLE PROCESSING UNIT

TASK 1	TASK 2
T1 Instr ₁	T2 Instr ₁
T1 Instr ₂	T2 Instr ₂
T1 Instr ₃	T2 Instr ₃
T1 Instr ₄	T2 Instr ₄
T1 Instr ₅	T2 Instr ₅
T1 Instr ₆	T2 Instr ₆
...	...
T1 Instr _n	T2 Instr _n
...	...
...	...
...	...

CONCURRENT PROGRAM (TWO TASKS)
ON SINGLE PROCESSING UNIT

Cycle	PU
1	T1 Instr ₁
2	T1 Instr ₂
3	T1 Instr ₃
4	T1 Instr ₃
5	T1 Instr ₃
6	T1 Instr ₃
7	T2 Instr ₁
8	T2 Instr ₁
9	T2 Instr ₃
10	T2 Instr ₄
11	T1 Instr ₅
...	...

Nella tabella a sinistra possiamo vedere l'esecuzione di un programma non-concorrente, a destra possiamo osservare l'esecuzione di un programma concorrente.

La scelta di una task da parte della PU, nel caso in cui si parli di esecuzione, spetta all'algoritmo di scheduling implementato

In generale, per gli umani è facile separare istruzioni indipendenti e unirle in base all'utilizzo che se ne fa; lo stesso principio non vale per una CPU. Bisogna quindi avere un qualcosa che separi il tempo di esecuzione di un'istruzione T1 rispetto a T2 in maniera tale da ordinarle in un singolo flusso; tipicamente vengono utilizzati algoritmi specifici o modelli di esecuzione.

il codice non-concorrente può essere eseguito in parallelo se l'hardware è provvisto di unità di processo multiple (vengono utilizzate VLIW o architetture multi-uso); le istruzioni possono essere seguite anche non in ordine.

TASK	Cycle	PU1	PU2	PU3	PU4
Instr ₁	1	Instr ₁	Instr ₂	Instr ₃	Instr ₅
Instr ₂	2	Instr ₆	Instr ₇	Instr ₃	Instr ₅
Instr ₃	3	Instr ₈	...	Instr ₃	Instr ₅
Instr ₄	4	Instr ₃	Instr ₉
Instr ₅	5	Instr _n	Instr _{n+1}	Instr ₄	Instr ₉
Instr ₆	6
...	7	Instr _{n+1}	Instr _{n+2}	Instr _{n+3}	Instr _{n+4}
Instr _n	8
...	9
...	10
...	11
...

NON-CONCURRENT PROGRAM (SINGLE TASK): PARALLEL EXECUTION ON MULTIPLE PROCESSING UNITS

Le istruzioni della singola task vengono "assegnate" alle PU disponibili e non vi è un ordine preciso

Un programma concorrente può essere eseguito in parallelo se l'hardware è provvisto di unità di processo multiple (vengono utilizzate VLIW o architetture multi-uso): le istruzioni possono essere seguite in ordine

TASK 1	TASK 2	Cycle	PU1	PU2	PU3	PU4
T1 Instr ₁	T2 Instr ₁	1	T1 Instr ₁	T1 Instr ₂	T2 Instr ₁	T2 Instr ₂
T1 Instr ₂	T2 Instr ₂	2	T1 Instr ₃	T1 Instr ₄	T2 Instr ₁	T2 Instr ₄
T1 Instr ₃	T2 Instr ₃	3	T1 Instr ₃	T1 Instr ₅	T2 Instr ₃	T2 Instr ₅
T1 Instr ₄	T2 Instr ₄	4	T1 Instr ₃	T1 Instr ₅	T2 Instr ₆	T2 Instr ₇
T1 Instr ₅	T2 Instr ₅	5	T1 Instr ₃	T1 Instr ₅	T2 Instr ₆	T2 Instr ₈
T1 Instr ₆	T2 Instr ₆	6
...	...	7
T1 Instr _n	T2 Instr _n	8
...	...	9
...	...	10
...	...	11
...

CONCURRENT PROGRAM (TWO TASKS): PARALLEL EXECUTION ON MULTIPLE PROCESSING UNITS

Infine, l'idea è che, utilizzando un approccio top-down, l'esecuzione di software applicativo richiede diversi passaggi di processamento (**sviluppo**) e di tempo di **esecuzione**. La tabella mostra le varie fasi che l'esecuzione del software attraversa, dal linguaggio ad alto livello alla logica digitale.

Eventualmente, anche l'hardware può essere definito da applicazioni software in quanto sta diventando una pratica comune nello sviluppo embedded. Anche le CPU general-purpose hanno hardware configurabile (FPGA)

Level	Input	Processing	Output	Time
High-level Language	High-level programming language sources	Translation to assembly by high-level language compilers	Target Assembly (ARM, RISC-V, PowerPC, MIPS, 68k, x86-32, x86-64, ...) sources	Development
Assembly Language	Target Assembly (ARM, RISC-V, PowerPC, MIPS, 68k, x86-32, x86-64, ...) sources	Translation to Machine Language by assemblers	Machine Language encoded in object files	
Target System (specific hardware)	Machine Language encoded in object files	Configuration for target, binding	Executables and libraries	
Instruction Set Architecture (ISA)	Executables and libraries	Direct execution, translation to microinstructions or both by CPUs	Direct execution: Data (operands, addresses) + digital logic control signals Translation to microinstructions:	Execution
Microarchitecture	Microinstruction streams	Execution of microinstruction streams	Data (operands, addresses) + digital logic control signals	
Digital logic	Data (operands, addresses) + digital logic control signals	Hardware (datapath) configured by control signals processes data (operands, addresses)	<u>Application results</u>	

Digital logic often **include multiple** similar **functional units** (e.g. ALUs, MACs, memory controllers) for **parallel execution**. Besides hardware parallelism, **several techniques** are used to **improve performance in general-purpose CPUs**: long pipelines, out-of-order and speculative executions, branch prediction. These techniques **are avoided in embedded processors**

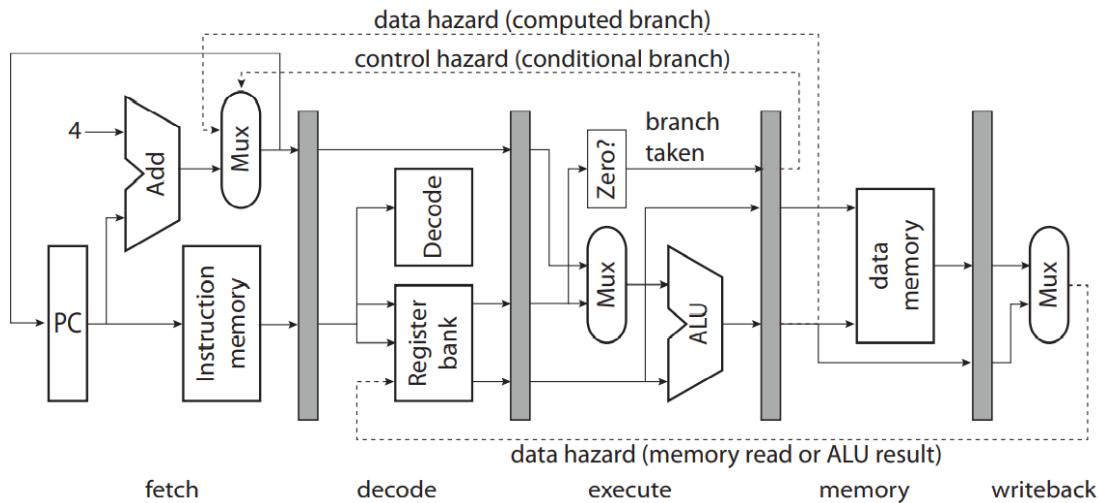
Lezione 5 - 12/10/2021

Problemi time-bound (CPU Optimization) - Introduzione e tecniche di gestione

In questa lezione ci occuperemo di "mitigare" i problemi riguardanti l'esecuzione di processi multipli e la conseguente ottimizzazione della computazione. Iniziamo a parlare del modo in cui la CPU gestisce il ciclo di computazione ("fetch-decode-execute"); per ottenere prestazioni importanti o comunque di alto livello, il **pipelining** è il primo passo per arrivare a questo "obiettivo": con questa tipologia di processamento, riusciamo ad ottenere un aumento in **"throughput"** (dati trasmessi per secondo) ma ne risente anche la latenza. Ogni pipeline è costruita generalmente come nella figura, e ne possiamo distinguere delle fasi (già citate). Nella fase **"fetch"**, il Program Counter (**PC**) "punta" all'istruzione successiva e fornisce un indirizzo all'istruzione memory, a meno che un'istruzione del **branch condizionale** fornisca un indirizzo completamente diverso; ipotizzando che la CPU accetti word a 32 bit, il PC viene incrementato di 4 byte per puntare all'istruzione successiva. Nella fase successiva, ovvero quella di **"decode"**, il componente di decodifica si occupa dell'estrazione **dell'indirizzo di registro** dall'istruzione a 32 bit e "preleva" i dati interessati dal registro specificato (tale registro ovviamente è presente nel **register bank**, l'insieme di registri). Lo stage di **"execute"** si occupa di operare i dati provenienti dal register bank o dal PC (nel caso di un "computed branch") all'interno dell'unità logico-aritmetica(**ALU**); le altre fasi, quelle di **memory** e **writeback** si occupano corrispettivamente della lettura/scrittura in memoria e della scrittura nel register bank

i processori DSP aggiungono componenti in più (e.s: componenti moltiplicative, ALU specifiche per sommare solamente gli indirizzi)

Nel contesto di pipeline, il tempo di esecuzione complessivo è lo stesso in quanto l'istruzione percorre sempre la stessa **circuiteria**. In questo caso specifico, possono essere eseguite altre istruzioni nonostante la circuiteria sia "impegnata" a processare altre istruzioni o parti di esso: se il percorso della circuiteria è ridotto possiamo usare una costante moltiplicativa per accelerare il numero di processi elaborati; idealmente, ad ogni ciclo (clock) un'istruzione viene processata (percorrendo tutte le fasi della circuiteria) ma possiamo ottimizzare caricando altre istruzioni nella pipeline. C'è un problema: la gestione dei **data branch**. Ogni qualvolta nel codice la CPU trovasse dei JUMP, i registri puntano all'istruzione a cui il primo si riferisce, ritornando effettivamente alla fase di fetch.



Modello sincrono-reattivo del comportamento di un processore

Le istruzioni ML (machine-language) sono spesso dipendenti dai risultati delle istruzioni precedenti; se B necessita un risultato da un registro scritto da A non può procedere nella pipeline: questo fenomeno è chiamato **data hazard**. Tali dipendenze vanno gestite: il compilatore (o lo sviluppatore) inserisce una o più istruzioni **no-op** tra A e B per far sì che la scrittura avvenga prima della lettura (**explicit pipeline**); queste istruzioni, se non gestite opportunamente, formano una **pipeline bubble** che si propaga nella pipeline stessa e potrebbero rappresentare un problema.

hardware resources:

instruction memory

register bank read 1

register bank read 2

ALU

data memory

register bank write

A	B	C	D	E			
A	B	C	D	E			
A	B	C	D	E			
A	B	C	D	E			
A	B	C	D	E			
A	B	C	D	E			
A	B	C	D	E			
1	2	3	4	5	6	7	8
							9
							cycle

un'esempio di "reservation table (tabella di prenotazione)" riguardante le istruzioni A,B,C,D ed E

A volte, il risultato di un'istruzione va "riportato indietro" in qualche registro nella fase di decode.

In tempi passati, nella gestione delle istruzioni si preferiva utilizzare un'architettura RISC dove la pipeline era completamente esposta al programmatore e quest'ultimo doveva essere in grado di utilizzarla ed ottimizzarne l'uso; invece nel presente viene addirittura implementato dell'hardware specifico in supporto della pipeline oppure vengono utilizzate delle tecniche complessi (**branch prediction, speculative execution, register renaming, out-of-order execution**,)

Out-of-order execution

l'hardware, capendo la presenza di una **pipeline bubble** (data hazard), invece di ritardare l'esecuzione di B, prosegue nel fetch di C, e se C non legge registri scritti da A e B, e non scrive in registri letti da B, allora procede nell'esecuzione di C prima di B e questo riduce la **pipeline bubble**. Questo in termini prestazionali è vantaggioso in quanto alleggerisce il carico del compilatore. A livello commerciale questa è la strategia vincente: il produttore può produrre processori con implementazioni hardware in grado di ottimizzare il processo di pipelining.

Control hazard

Come già indicato in precedenza, i branch condizionali cambiano il valore del PC se il registro specifico ha valore zero; il nuovo valore del PC può anche essere prodotto da un risultato di un'operazione effettuata nell'ALU (se A è un'istruzione presente nel branch condizionale, allora A deve raggiungere lo stage di memoria prima che il PC venga aggiornato): le istruzioni successive ad A (in memoria) saranno state fetcate e saranno nelle fasi di decode ed execute **DAL TEMPO** in cui è stato determinato che queste istruzioni non dovrebbero essere eseguite. Si può ottimizzare questo processo tramite la **predizione del branch**: se abbiamo una sequenza condizionale (o di loop) molto grande (ad esempio un for che deve iterare da vi sono delle operazioni **ridondanti** da eseguire e quindi in genere potrebbe essere oneroso a lungo termine per la CPU; l'hardware di predizione ha "memoria" delle istruzioni precedenti (e dei pattern) e quest'ultima può "riconoscere" una ripetizione di tale istruzione PRIMA di leggere i registri con tali istruzioni e può far andare avanti il PC con le successive istruzioni; se la valutazione dell'hardware di branch prediction è errata, la pipeline viene svuotata

Delayed branch

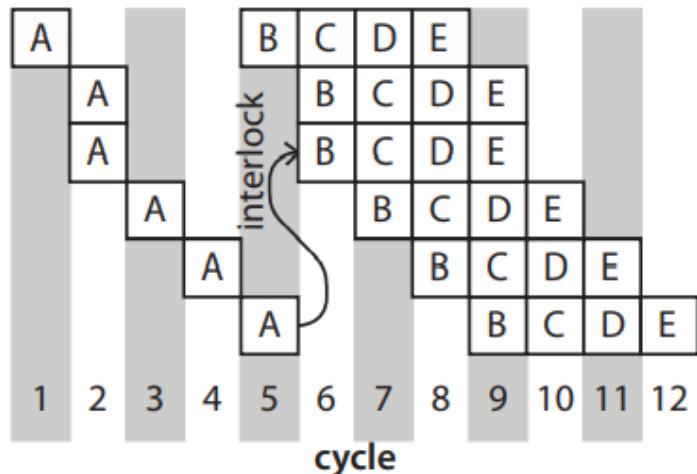
Documenta il fatto che il branch sarà "preso" dopo un certo numero di cicli ed è compito del compilatore o del programmatore stabilire se le istruzioni successive al branch condizionale siano "dannose" (ovvero che necessitino di istruzioni)

Interlock

in questa tecnica, l'hardware che decodifica le istruzioni, prima di incontrare B il quale legge un registro scritto da A, individuerà il "data hazard" e ritarderà l'esecuzione di B finchè A non abbia terminato lo stage di writeback

hardware resources:

instruction memory
register bank read 1
register bank read 2
ALU
data memory
register bank write



Speculative execution

Vengono prese multiple istruzioni (la CPU esegue "due rami"), ne valuta la correlazione e sceglie il "percorso giusto": si necessita, quindi, di una **biforcazione** hardware; il "ramo" con le istruzioni "sbagliate" viene svuotato.

Data and control Hazard

Ad eccezioni delle pipeline esplicite e dei branch ritardati, **tutte le tecniche introducono variabilità; l'analisi del tempo** può risultare quindi **estremamente difficile** quando si ha una pipeline "profonda" con salti e speculazioni elaborate. Nei modelli di embedded programming, è necessario che si abbia sempre lo stesso tempo di esecuzione o comunque approssimazioni di tempo per ogni esecuzione che siano il più simili possibili. Le pipeline esplicite sono molto comuni dei **processori DSP** e sono applicate in contesti dove la precisione temporale è necessaria e fondamentale.

Instruction level parallelism

è una metodologia di processamento implementabile tramite **istruzioni CISC, parallelismo superscalare, subword parallelism, VLIW**. (fare una ricerca per approfondire). Tipicamente i processori DSP hanno implementazioni CISC.

Implementazione CISC

è, molto banalmente, un processore che possiede un complesso set d'istruzioni

ESEMPIO TMS320c54x (il codice sotto è una possibile implementazione del FIL filtering)

```
RPT numberOfTaps -1
MAC *AR2+, *AR3+, A
          a=a + x*y
(i registri AR2 e AR3 possono essere impostati per l'implementazione di
buffer circolari)
N.B: l'operazione MAC viene eseguita in un ciclo
```

i processori c54x includono una sezione di memoria on-chip in grado di supportare due accessi in un singolo ciclo

In ogni ciclo, il processore effettua due fetch di memoria, una moltiplicazione, un'addizione e due (possibilmente in modulo) incrementi d'indirizzo (l'accumulatore è implementato in un registro a parte, evitando operazioni onerose)

Subword parallelism

Molti sistemi embedded operano con tipologie di dati le quali sono più piccole rispetto alla word size del processore. Un'ALU più grande è divisa in pezzi più piccoli i quali possono effettuare in maniera simultanea operazioni aritmetico/logiche in word più piccole: usiamo quindi la word più grande come se fosse un **vettore** suddiviso in parti, permettendo operazioni aritmetico/logiche simultanee nelle word più piccole

Prendendo in considerazione la word size come se fosse un vettore (di dimensione a 32 bit, ad esempio), la word stessa può essere "ripensata" come un vettore suddiviso in subword (ad esempio da 8 bit) le quali hanno ognuna un determinato tipo d'istruzione

La subword parallelism non è nient'altro che un caso particolare di **vector processor** (quest'ultimo ne rappresenta una generalizzazione, in quanto possono essere applicate svariate operazioni). Intel per prima ha introdotto la subword parallelism nel processore general-purpose "Pentium" (MMX).

Architetture Very Long Instruction Word (VLIW)

L'idea è che si abbia una macchina con una word molto grande, con una CPU capace di effettuare la prelevazione di tale word con più unità computazionali.

```
filtro FIR in VLIW
(istruzioni SIMD)
MAC *AR2+, *CDP+, AC0
::MAC *AR3+, *CDP+, AC1
```

AC0 e AC1 sono due registri accumulatori e CDP è il registro specializzato nel puntare ai coefficienti di filtro. La notazione :: sta ad indicare che l'esecuzione delle due istruzioni deve essere indicata e che può essere effettuata nello stesso ciclo; sta al programmatore decidere quali di queste istruzioni possono essere eseguite contemporaneamente. Assumendo che gli indirizzi di memoria siano sufficientemente grandi così che il fetch possa avvenire in maniera simultanea, queste due istruzioni MAC vengono eseguite in un singolo ciclo, dividendo effettivamente a metà il tempo richiesto per eseguire un filtro FIR. Nel pratico il numero di operazioni è ridotto di $n/2 + 1$.

Raggiungere la massima efficienza significa "riempire" tutte le unità funzionali (CU - Computational Unit). Vi sono diversi modi per effettuare il riempimento di esse:

1. Definire in software (tramite compilatore) il riempimento di tale unità: solitamente avviene in maniera sequenziale. Richiede una conoscenza profonda di ciò che viene eseguito e del funzionamento della pipeline
2. Hardware (processori superscalari): il riempimento è gestito dall'hardware

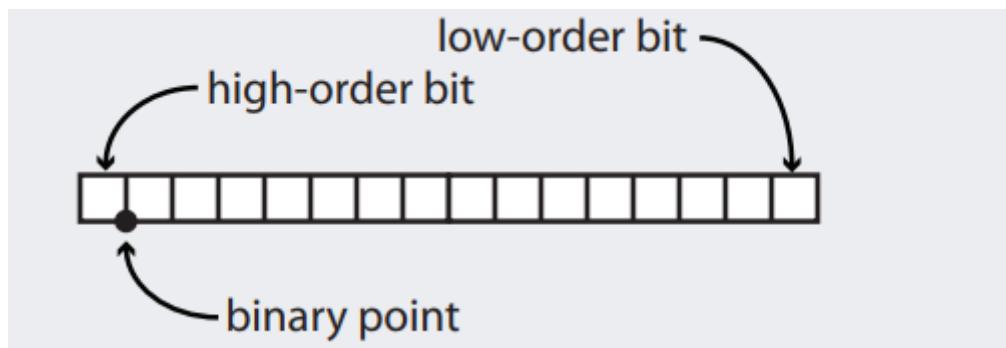
Architetture Multicore

Una macchina multicore è una combinazione di diversi processori in un singolo chip. Ogni processore può, in maniera indipendente, gestire le istruzioni in un approccio **sequenziale**, oppure in approccio **multi-threaded** utilizzando più CPU per lo stesso programma. Macchine multicore eterogenee combinano una varietà di processori diversi in un singolo chip e questo

comporta un vantaggio a livello prestazionale in quanto il "carico di lavoro" viene distribuito e la CPU principale esegue operazioni fondamentali, grazie all'esecuzione parallela; il vantaggio è ottenuto anche in ambito embedded in quanto vanno gestiti **eventi real-time**. L'ottimizzazione non avviene soltanto in ambito di calcolo ma anche a livello di consumo: se, ad esempio, l'architettura di uno smartphone include più processori, l'utilizzo di questi può risultare oneroso in ambito energetico, si può includere un processore "a basso consumo" (più semplice rispetto a quello principale) e attivarlo nel momento in cui il dispositivo non è utilizzato, tramite un'opportuna gestione delle risorse effettuata (possibilmente ma non necessariamente) anche dallo sviluppatore.

Fixed-point numbers

Molti processori embedded offrono hardware specializzato nell'aritmetica di interi; quest'ultima può essere utilizzata anche per numeri parziali, facendo particolarmente attenzione.



avendo un intero a 16 bit, un programmatore può immaginare un punto binario, approssimabile ad un punto decimale con un'unica differenza: separa i bit piuttosto che la parte intera da quella non intera

nel caso in cui avessimo, ad esempio, 1.15, possiamo stabilire che a sinistra del punto binario ci sia un bit e 15 a destra; quando questi due numeri sono moltiplicati a precisione totale, il risultato è un numero a 32 bit. Generalmente l'estrazione di 16 bit da un risultato a 32 bit comporta la perdita d'informazione; se scartiamo i 15 bit meno significativi effettuiamo un **troncamento**, se aggiungiamo il pattern di bit indicato nell'immagine effettuiamo un'**arrotondamento**. L'approssimazione ci permette di trovare il risultato più vicino al risultato espresso in precisione totale

01000000000000000000000000000000

Questo è il risultato della moltiplicazione tra -1 e -1 in 16 bit (ottenendo un valore a 32 bit). Se visualizziamo solo la parte in grigio (ovvero il pattern di bit moltiplicato), il valore di questo numero in complemento a 2 sarà -1

00000000000000000000000000000000

cambiando posizione al pattern (ovvero spostare i 16 bit, otteniamo il risultato esatto, effettuando un'approssimazione)

Livello ISA

viene illustrata in maniera generale la macchina di Von Neumann a livello ISA. Nella memoria sono presenti celle contigue raggruppabili per immagazzinare valori a più byte (nell'immagine possiamo interpretare il contenuto delle celle 0002 e 0003 come valori a 16-bit). La codifica può essere effettuata in due modi

- **little endian**: il dato viene letto dall'LSB al MSB
- **big endian**: al contrario

Solitamente la rappresentazione **big endian** è più facile da decodificare per un'essere umano, mentre la maggior parte delle architetture adotta una rappresentazione **little endian**.

Address	Value
0000	00
0001	FF
0002	27
0003	1B
0004	02
...	...
0400	03
0401	09
0402	01
0403	0F
...	...
FFFE	0C
FFFF	80

Address decoding

Solitamente la memoria è gestita diversamente in un sistema reale. Consideriamo un sistema con un bus address a n-bit; il range di 2^n bit deve essere condiviso con **TUTTE** le componenti connesse al bus: vanno scelte delle componenti con dimensione opportuna. Una circuiteria di decoding dev'essere presente e in base alle linee attivate, stabilisce quali componenti vengono attivate e quali no (ad esempio, un segnale Chip-Select (CS) connesso al pin con lo stesso nome in ogni chip)

Si può gestire la partizione dello spazio d'indirizzamento in maniera "intelligente" attivando alcune linee e sfruttando il posizionamento del MSB

Address	Value	Chip
0000	00	16 KB RAM
...	...	
3FFF	34	
...	UNUSED	
8000	40	8KB ROM
...	...	
9FFF	62	
...	UNUSED	
FFFC	0F	4B I/O
FFFD	0C	
FFFE	0C	
FFFF	80	

Lezione 6 - 14/10/2021

ISA ARM32 (Advanced RISC Machine)

Questo ISA è uno dei più vecchi presenti nel mercato; nasce come ISA RISC (dove, ricordiamolo, la complessità viene spostata ad alto livello, mantenendo semplicità a basso livello). Banalmente, ogni istruzione è codificata in word da 32 bit e memorizzate in tali word; ovviamente, la codifica può essere effettuata rispetto a vari formati i quali rappresentano **le istruzioni** (alcuni formati definisco più di una istruzione). Molte operazioni di processamento dati possono essere interpretate come se eseguissero una singola e flessibile istruzione; l'istruzione compie una specifica operazione logico/aritmetica su uno o più operandi, il primo operando è tipicamente un registro (Rn). In alcuni comandi, come MOV e MVN, il primo operando è ignorato

ARM Instruction Set Format

31	28 27 26 25 24	16 15	8 7	0	<u>Instruction type</u>	
Cond	0 0 I	OpCode	S	Rn	Rd	Operand2
Cond	0 0 0 0 0 0 A	S	Rd	Rn	Rs	1 0 0 1 Rm
Cond	0 0 0 0 1 U	A S	RdHi	RdLo	Rs	1 0 0 1 Rm
Cond	0 0 0 1 0 B	0 0	Rn	Rd	0 0 0 0 1 0 0 1	Rm
Cond	0 1 I P	U B W L	Rn	Rd	Offset	
Cond	1 0 0 P	U S W L	Rn	Register List		
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0 1 S H 1	Rm	
Cond	1 0 1 L	Offset				
Cond	0 0 0 1 0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn
Cond	1 1 0 P U N W L	Rn	CRd	CPNum	Offset	
Cond	1 1 1 0 Op1	CRn	CRd	CPNum	Op2 0	CRm
Cond	1 1 1 0 Op1 L	CRn	Rd	CPNum	Op2 1	CRm
Cond	1 1 1 1	SWI Number				

*l'istruzione ARM occupa l'intera word ed effettua una seguente suddivisione:

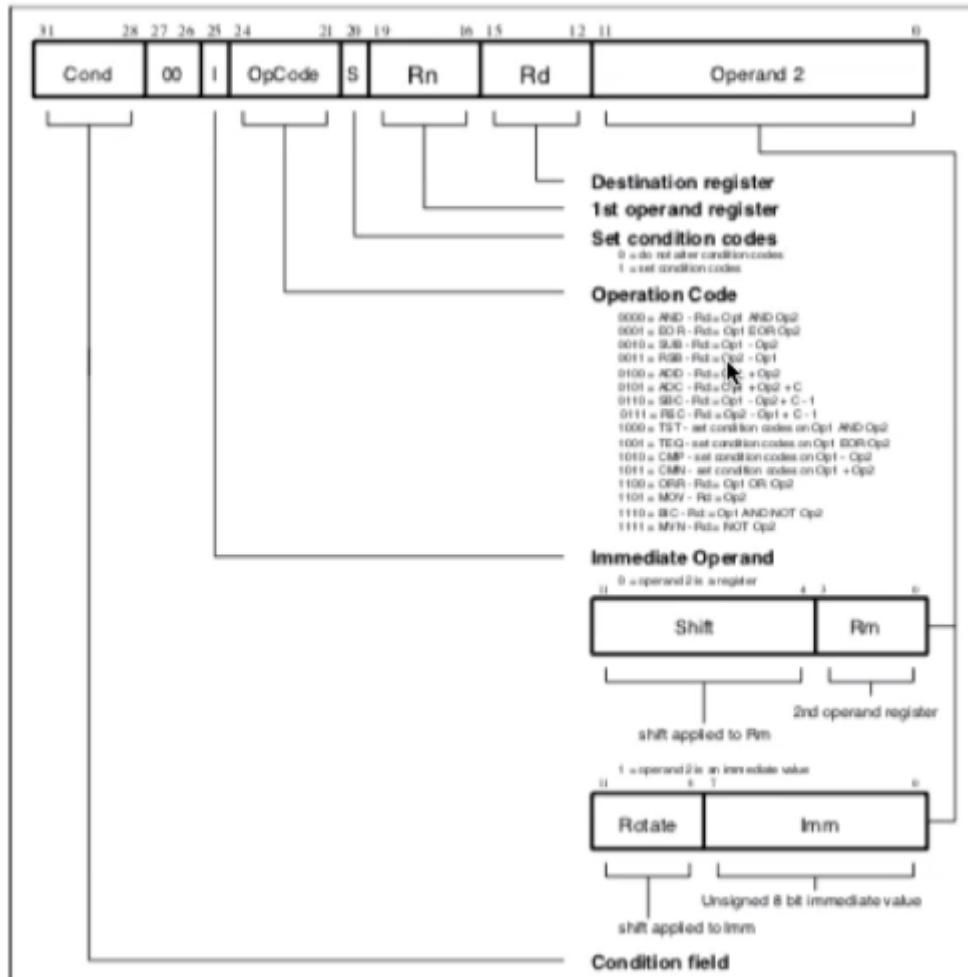


Figure 4-4: Data processing instructions

ARM32 in dettaglio

ARM possiede 6 modalità operative:

- Utente (modalità senza privilegi dove quasi ogni task viene eseguita)
- FIQ (attiva quando viene posto un interrupt ad alta priorità)
- IRQ (attiva quando viene posto un interrupt a bassa priorità)
- Supervisor (quando viene effettuato un reset e quando un'istruzione SI (Software Interrupt) viene eseguita)
- Abort (utilizzata per gestire violazioni di accesso alla memoria)
- Undef (utilizzata per gestire istruzioni non definite)

Con ARM-v4 abbiamo una "settima" modalità, ovvero questa di Sistema (modalità con privilegio, utilizza gli stessi registri della user mode). Il problema più complesso da gestire è il **context switching** (in questo caso è il cambio modalità): bisogna gestire i dati salvati dal contesto precedente per evitarne la perdita dopo aver effettuato lo switching. ARM possiede in totale 37 registri (ovviamente tutti di dimensione 32 bit):

- 1 PC
- 1 current program status register
- 5 saved program status register
- 30 registri general-purpose

Questi registri sono sistemati in diversi "banchi"; ogni modalità può accedere:

- un set particolare di registri R0-R12
- un registro R13 rappresentante lo Stack Pointer e il registro 14 (Link Register)
- R15 (il program counter)
- CPSR (il current program status register)

da questa lista viene escluso il registro SPSR (saved program status register), governato dalle modalità "privilegiate"

Register Organisation

General registers and Program Counter

User32 / System	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13 (sp)	r13_fiq	r13_svc	r13_abt	r13_irq	r13_undef
r14 (lr)	r14_fiq	r14_svc	r14_abt	r14_irq	r14_undef
r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)

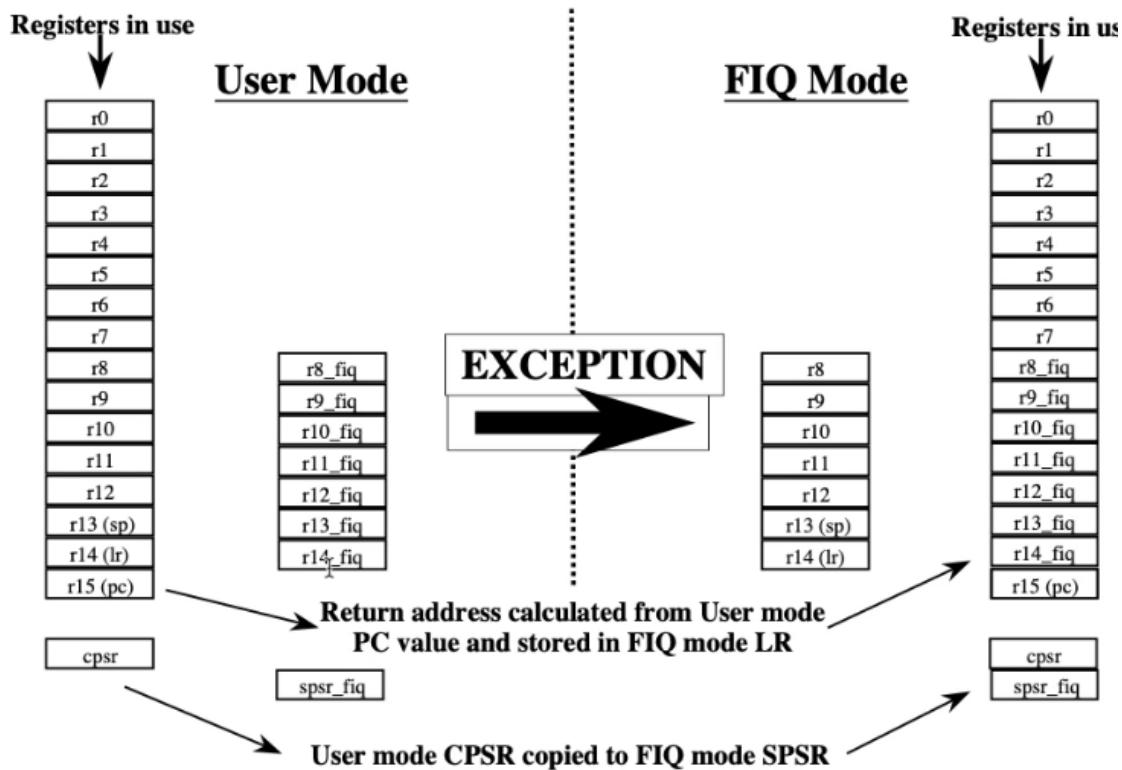
Program Status Registers

cpsr	cpsr spsr_fiq	cpsr spsr_svc	cpsr spsr_abt	cpsr spsr_irq	cpsr spsr_undef
------	------------------	------------------	------------------	------------------	--------------------

gestione ed utilizzo dei registri in base alla modalità utente

Possiamo eseguire del codice normalmente, ad esempio in modalità User32, e se dovesse succedere qualcosa (hardware interrupt, ecc..) il CPSR si occupa di "bloccare" l'esecuzione (salvando lo stato dell'esecuzione) rispetto all'utente per permettere il context switching, ad esempio, verso la modalità FIQ o IRQ

Register Example: User to FIQ Mode



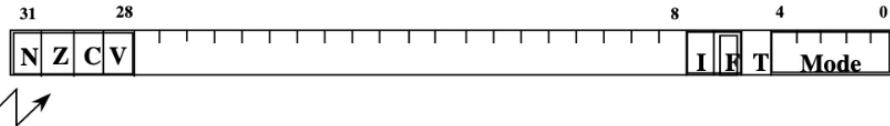
Nel momento in cui riceviamo un interrupt (che sia software che hardware), trovandoci in user mode, il contenuto CPSR viene copiato nel SPSR della modalità FIQ prima dello switching (quindi, banalmente $spsr_{fiq}=cpsr$) e vengono utilizzati gli stessi registri (nominalmente ma non fisicamente), ma in modalità FIQ, effettuando un cambio dei "banchi"; nella modalità IRQ, invece, non abbiamo lo stesso procedimento in quanto i registri "corrispondono". I registri in modalità FIQ sono generalmente più veloci rispetto alle altre modalità.

Accesso dei registri utilizzando istruzioni ARM

Tutte le istruzioni possono accedere al range R0-R14 in maniera diversa e la maggior parte delle istruzioni permettono l'utilizzo del program counter.

PSR (Program Status Register)

The Program Status Registers (CPSR and SPSRs)



- * **Condition Code Flags**
 - N = Negative result from ALU flag.
 - Z = Zero result from ALU flag.
 - C = ALU operation Carried out
 - V = ALU operation oVerflowed
- * **Mode Bits**
 - M[4:0] define the processor mode.
- * **Interrupt Disable bits.**
 - I = 1, disables the IRQ.
 - F = 1, disables the FIQ.
- * **T Bit (Architecture v4T only)**
 - T = 0, Processor in ARM state
 - T = 1, Processor in Thumb state

Di default le operazioni logico/aritmetiche non cambiano i PSR; necessitiamo di specificare a NZCV delle flag (possono essere 0 o 1)

Condition Flags

	Logical Instruction	Arithmetic Instruction
<u>Flag</u>		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

Un approfondimento sull'architettura (fisica, quindi si parla anche della famiglia Cortex) lo si può trovare [qui](#).

ARM32 (continua..)

Solitamente, il codice scritto è su un semplice file di testo: quando viene "eseguito", il compilatore si occupa di convertire tale codice in linguaggio macchina. La sintassi di ARM è molto rigida (solitamente l'istruzione inizia con il comando, poi con i registri e l'eventuale valore da memorizzare). Ricordiamo che in questo ISA, l'istruzione è "lunga" 32 bit, quindi il valore memorizzabile (sicuramente minore a 32 bit in quanto parte dei bit è necessaria per richiamare l'istruzione e il registro su cui si vuole effettuare l'operazione) sarà una parte di esso e dobbiamo gestire questa limitazione.

Program Counter (R15)

Il valore del Pc viene memorizzato nella sezione di bit [31:2] con bit [1:0] uguali a 0 (in quanto le istruzioni non possono avere la word dimezzata o byte aligned). Il pc nell'esecuzione del programma è gestito in diverse maniere: tipicamente un programma non è composto solamente da istruzioni semplici, ma contiene delle **subroutine** (porzioni di codice con scope locale e "isolate" rispetto alle altre subroutine); il program counter, quando viene invocata la subroutine, "salta" (operazione di JUMP) dalla riga in cui viene effettuata la chiamata alla prima istruzione della subroutine. In supporto al PC abbiamo il registro R14, utilizzato come subroutine link register (LR) e memorizza l'indirizzo di ritorno quando viene eseguito un branch con operazione di link. Il return viene implementato sfruttando il PC, memorizzando in esso il valore del link register (quindi l'indirizzo di return, che contiene i dati dell'istruzione precedente)

```
subprog1: instruct1;
           instruct2;
           ...
           calls subprog2
           instruct3;
           return
subprog2: instr...
           ...
           return
```

L'utilizzo del LR è necessario per l'ottimizzazione di un codice. Quando vi sono più di due subroutine, come viene gestito il Link Register? Vi è una **pila** che gestisce gli indirizzi di "link" (in cima vi sono le subroutine più interne, in basso quelle più esterne): il link register salva sempre l'ultimo indirizzo del link (ovvero l'ultima subroutine chiamata corrispondente al valore in cima della pila) e questo fa risparmiare tempo (**"chiamata di coda, o di foglia"**). La call effettua due operazioni: una è sicuramente il salto nella subroutine di destinazione e la seconda è il ritorno all'istruzione seguente alla chiamata di subroutine.

N.B: in realtà come già detto in precedenza, si utilizza il link register per comodità, ma il modo in cui il return è implementabile è SEMPRE a scelta dello sviluppatore; si possono utilizzare più registri

Gestione delle eccezioni e il "vettore tabella"

L'idea della gestione è la seguente: si può trovare una tabella vettore (individuabile in qualsiasi architettura, ovviamente) con determinate partizioni di memoria su di esso (quindi troveremo settori separati per ambito dove verranno gestiti i FIQ, i IRQ, reset). Ogni partizione ha 4 bit nella seguente immagine. Quando il sistema è nello stato di reset, l'indirizzo di offset del vettore tabella

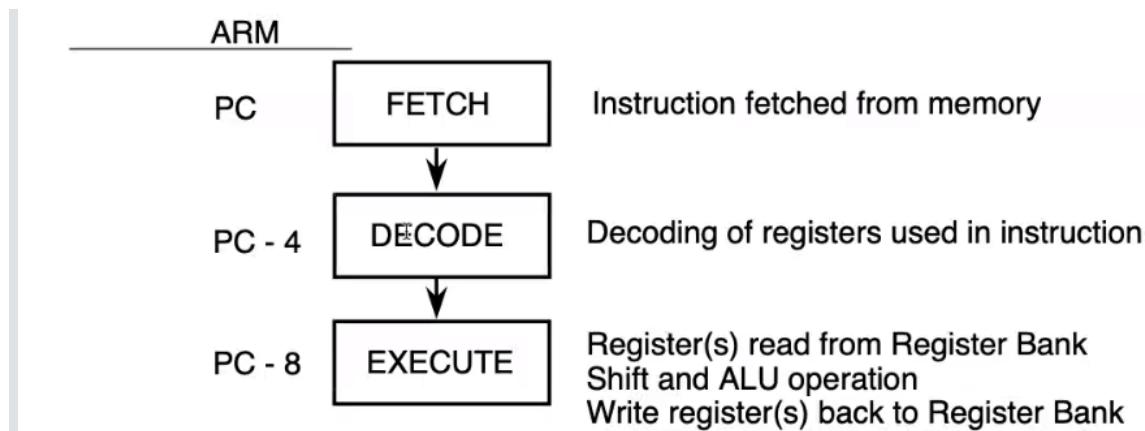
è fissato a 0x00000000; tali offset vengono memorizzati in un registro chiamato VTOR (**Vector Table Offset Register**)

0x00000000	Reset
0x00000004	Undefined Instruction
0x00000008	Software Interrupt
0x0000000C	Prefetch Abort
0x00000010	Data Abort
0x00000014	Reserved
0x00000018	I IRQ
0x0000001C	FIQ

Quando avviene un'eccezione, il processore copia il contenuto del CPSR nel SPSR_mode (alcune architetture utilizzano l'LR con la stessa finalità) e vengono settati i bit appropriati (vedi lezione 6); successivamente viene effettuato il mapping nei banchi di registro appropriati e viene memorizzato l'indirizzo di ritorno nel LR_mode; al PC, invece, viene assegnato l'indirizzo di ritorno. Per ritornare effettivamente un valore, gli handler di eccezioni necessitano di ripristinare il CPSR dal SPSR_mode e il PC dal LR_mode

Pipeline ARM32

Anche ARM utilizza una pipeline per ottimizzare la velocità del flusso d'istruzioni nel processore. Molte operazioni possono essere eseguite simultaneamente, invece che serialmente, utilizzando (presumibilmente) tecniche di Instruction Level Parallelism.



Il PC ovviamente non punta all'istruzione in esecuzione ma all'istruzione che sta per essere "prelevata" (quindi nello stadio di fetch viene prelevata l'istruzione e viene incrementato il PC), quindi nel momento di decode il pc sarà incrementato di 4. Alla fine del ciclo, sarà incrementato da 8

Esecuzione condizionale

Molti set d'istruzioni permettono solamente l'esecuzione di branch in maniera condizionale. Riutilizzando l'hardware di valutazione di condizione, ARM. Un branch crea una **pipeline bubble** (vedi lezioni precedenti) e ovviamente va gestita: solitamente, se la condizione del branch non è rispettata, il PC deve "saltare" l'istruzione da eseguire in caso di riscontro positivo della condizione e puntare alla prima istruzione che non appartiene alla condizione, "buttando via" dalla pipeline

l'istruzione non eseguita. Di conseguenza le istruzioni non eseguite fanno sprecare 1 ciclo. Per non creare pipeline bubble, bisogna evitare il più possibile i branch e l'esecuzione condizionale aiuta nell'ottimizzazione della pipeline.

The Condition Field

31	28	24	20	16	12	8	4	0
	Cond							
0000	= EQ - Z set (equal)							
0001	= NE - Z clear (not equal)							
0010	= HS / CS - C set (unsigned higher or same)							
0011	= LO / CC - C clear (unsigned lower)							
0100	= MI - N set (negative)							
0101	= PL - N clear (positive or zero)							
0110	= VS - V set (overflow)							
0111	= VC - V clear (no overflow)							
1000	= HI - C set and Z clear (unsigned higher)							
1001	= LS - C clear or Z (set unsigned lower or same)							
1010	= GE - N set and V set, or N clear and V clear (>or =)							
1011	= LT - N set and V clear, or N clear and V set (>)							
1100	= GT - Z clear, and either N set and V set, or N clear and V set (>)							
1101	= LE - Z set, or N set and V clear, or N clear and V set (<, or =)							
1110	= AL - always							
1111	= NV - reserved.							

Esempi vari di condizioni

```
mov r0, #23
mov r1, #5
addeq r2,r0,r1 -> se z è set allora somma, altrimenti salta
```

Using and updating the Condition Field

- * To execute an instruction conditionally, simply postfix it with the appropriate condition:

- For example an add instruction takes the form:
– ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)
- To execute this only if the zero flag is set:
– ADDEQ r0,r1,r2 ; If zero flag set then...
; ... r0 = r1 + r2

- * By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.

- For example to add two numbers and set the condition flags:
– ADDS r0,r1,r2 ; r0 = r1 + r2
; ... and set flags

in questa foto viene specificata come eseguire un'istruzione in maniera condizionale: basta aggiungere alla fine le condizioni già viste in precedenza. Di default le flag condizionali non sono aggiornate: per fare ciò è necessario che sia settato il bit S

Istruzioni di branch

L'offset per le istruzioni di branch è calcolato dall'assembler prendendo la differenza tra l'istruzione di branch e l'indirizzo target meno 8. Questo fa ottenere un offset da 26 bit il quale viene "shiftato a destra" di 2 bit (i due bit in fondo sono sempre 0 e le istruzioni sono word-aligned) e memorizzati nella codifica dell'istruzione. Questo ci dà un range di +- 32 Mbytes. Generalmente una subroutine è un blocco di codice che esegue delle istruzioni in base ad alcuni argomenti e optionalmente ritorna un valore. L'istruzione BL effettua le seguenti operazioni:

- piazza l'indirizzo di ritorno nel link register
- assegna al PC l'indirizzo della subroutine

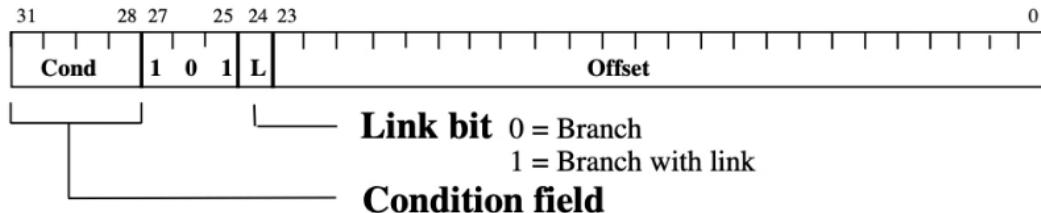
dopo che il codice della subroutine è stato eseguito, si può utilizzare l'istruzione BX lr per ritironare un valore. In realtà non è l'unico modo per effettuare il ritorno: come già affermato in precedenza, vi sono diversi modi per implementarlo ma ne studieremo solo 2

- l'istruzione BX lr effettua il ritorno
- nell'entry di subroutine, il valore di LR viene memorizzato nello stack e poi, quando effettivamente vi è necessità di ritorno, viene prelevato e memorizzato nel PC tramite l'operazione di pop

ESEMPIO: somma di due numeri

```
AREA      subrout, CODE, READONLY      ; Name this block of code
        ENTRY                      ; Mark first instruction to execute
start    MOV      r0, #10                ; Set up parameters
        MOV      r1, #3
        BL       doadd                ; Call subroutine
stop     MOV      r0, #0x18              ; angel_SWIreason_ReportException
        LDR      r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SVC      #0x123456            ; ARM semihosting (formerly SWI)
doadd   ADD      r0, r0, r1              ; Subroutine code
        BX      lr                  ; Return from subroutine
        END                      ; Mark end of file
```

- * **Branch :** B{<cond>} label
* **Branch with Link :** BL{<cond>} sub_routine_label



questo schema rappresenta la suddivisione della word in un'istruzione Branch: i bit da 27 a 25 rappresentano L'opcode e il bit 24 il link bit; se settato a 0 verrà codificato B, altrimenti BL

Condition field (continua...)

Il bit S, se settato, aggiorna le condition flag. In qualsiasi istruzione, se non viene specificata la condizione, vi sarà sempre la condizione AL (always) eseguita (forzata dall'assembler).

Branch e Branch with Link

Nel branch necessitiamo solo un JUMP e di non ritornare dopo l'esecuzione della subroutine, al contrario del BL. Il "branch with link" quindi, implementa una chiamata di subroutine "scrivendo" il contenuto del PC nel LR del banco corrente (l'indirizzo della prossima istruzione successiva al BL). Banalmente, per ritornare una subroutine, basta tale codice:

```
MOV pc, lr
```

Il pc, che avrà il valore corrente ovvero il valore della prossima istruzione, sovrascriverà il suo valore con quello del link register; la pipeline andrà riempita prima che l'esecuzione possa continuare

| N.B: l'istruzione di "Branch" non influenza LR

Generalmente, come detto prima, si preferisce utilizzare BX (utilizzabile dall'architettura 4T) per effettuare il ritorno.

Data processing instruction

è la "famiglia più grande" di istruzioni ARM, e tutte condividono lo stesso formato e contengono

- comparazioni
- operazioni aritmetiche
- operazioni logiche
- movimento di dati tra registri.

Qualsiasi sia l'operazione, queste funzionano sui dati scritti nei registri e scrivono sui registri di destinazione definiti dallo sviluppatore (**load/store**), e quindi NON FUNZIONANO NELLA MEMORIA

Operazioni aritmetiche

* **Operations are:**

- ADD operand1 + operand2
- ADC operand1 + operand2 + carry
- SUB operand1 - operand2
- SBC operand1 - operand2 + carry -1
- RSB operand2 - operand1
- RSC operand2 - operand1 + carry - 1

* **Syntax:**

- <Operation>{<cond>} {S} Rd, Rn, Operand2

* **Examples**

- ADD r0, r1, r2
- SUBGT r3, r3, #1
- RSBLES r4, r5, #5

la sintassi è la seguente:

```
<Operazione>{<cond>} {S} Rd, Rn, Operand2
```

non c'è differenza tra SUB e RSB, se non gli operandi scambiati: c'è un motivo se i designer lo hanno inserito.

Comparazione

questo tipo di operazione non contiene la destinazione, perchè in questo caso non abbiamo la necessità di memorizzare se non fare un paragone. Nel caso in cui si utilizzare questo tipo di istruzione, **le flag condizionali vengono automaticamente aggiornate**; non c'è risultato se l'informazione non viene propagato allo status register anche se le comparazioni sono semplicemente "**sottrazioni**"

- * **The only effect of the comparisons is to**
 - **UPDATE THE CONDITION FLAGS.** Thus no need to set S bit.
- * **Operations are:**
 - CMP operand1 - operand2, but result not written
 - CMN operand1 + operand2, but result not written
 - TST operand1 AND operand2, but result not written
 - TEQ operand1 EOR operand2, but result not written
- * **Syntax:**
 - <Operation>{<cond>} Rn, Operand2
- * **Examples:**
 - CMP r0, r1
 - TSTEQ r2, #5

quando si lavora con l'hardware, è necessario utilizzare operatori bit-wise

Operatori Logici

abbiamo anche qui la sintassi con 3 operatori: tutti gli operatori sono classici tranne BIC (è un'istruzione utile quando si manipola l'hardware ed è necessario nella modifica di singoli bit)

- * **Operations are:**
 - AND operand1 AND operand2
 - EOR operand1 EOR operand2
 - ORR operand1 OR operand2
 - BIC operand1 AND NOT operand2 [ie bit clear]
- * **Syntax:**
 - <Operation>{<cond>}{S} Rd, Rn, Operand2
- * **Examples:**
 - AND r0, r1, r2
 - BICEQ r2, r3, #7
 - EORS r1,r3,r0

Data movement

queste istruzioni si occupano semplicemente di spostare il contenuto dell'operando sul registro di destinazione (l'operando può essere non solo un registro ma anche valori costanti)

Data Movement

Data Movement

* Operations are:

- MOV operand2
- MVN NOT operand2

Note that these make no use of operand1.

* Syntax:

- <Operation>{<cond>} {S} Rd, Operand2

* Examples:

- MOV r0, r1
- MOVS r2, #10
- MVNEQ r1,#0

un data path è un percorso ciclico: ipotizziamo di avere una circuiteria specializzata nell'abilitare delle linee di bus (CU), attiviamo R0 nel bus A, il dato sarà propagato all'alu e successivamente abilitiamo R1 nel bus B anche lui collegato all'ALU; entrambe le istruzioni sono codificate in 32 bit. Le posizioni dell'operando 1 e 2 sono fisse. Il punto è che i dati operati dall'ALU, in base alla condizioni logiche impostate nelle flag, sono trasportati dai bus che riporteranno tali dati (eventualmente) ai registri

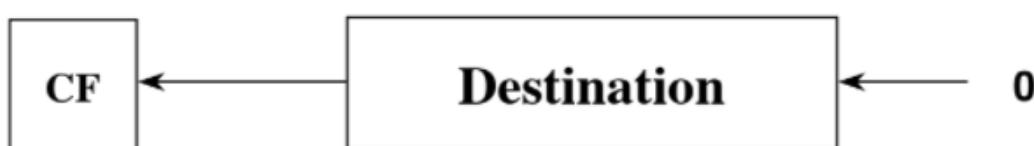
Lezione 9 - 26/10/2021

Barrel Shifter

è una scelta di design "intelligente" in quanto si tratta di circuiteria specifica la quale svolge determinate operazioni e fornisce grande "flessibilità" nonostante ci siano dei limiti. Sapendo come funziona e come è implementato il Barrel Shifter, il programmatore può sfruttarlo a suo vantaggio per svolgere operazioni in maniera più rapida ed alleggerire il carico. Di base, esso "sposta" dei bit; il left shift (LSL) è una delle operazioni possibili. Le altre operazioni sono:

- Logical Shift Right: sposta i bit a destra di una specifica quantità (divide per potenze di due)
 - LSR #5 = dividere per 32
- Arithmetic Shift Right: sposta i bit a destra di una specifica quantità e preserva il bit del segno (utile per operazioni in complemento a due)
 - ASR #5 = dividere per 32

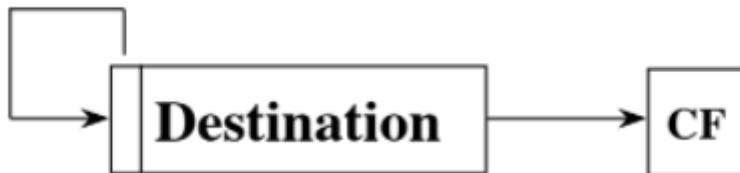
Logical Shift Left (LSL)



Logical Shift Right



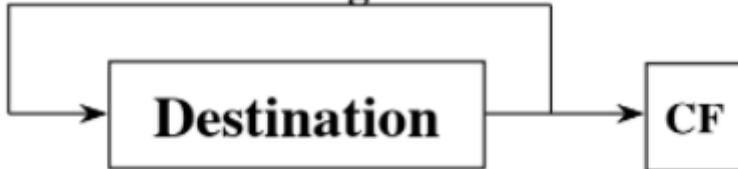
Arithmetic Shift Right



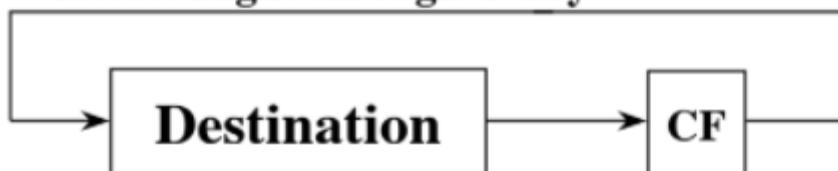
A prescindere dall'operazione svolta, il risultato dell'operazione passerà sempre per il Carry Flag (CF)

- Rotate right (ROR): simile ad ASR ma i bit si "attorcigliano" passando da LSB a MSB
 - ROR #5
- Rotate Right Extended (RRX): la rotazione usa la flag C del Current Program Status Register (CPSR) come il 33esimo bit; ruota a destra di un bit (codificato come ROR #0)

Rotate Right



Rotate Right through Carry



Possiamo aggiungere questa considerazione: il secondo elemento dell'operando passa dal Barrel Shifter prima di arrivare nell'ALU, e questo avviene **sempre** anche se non vi è necessità. I valori di shift possono essere sia dei valori interi (non negativi) a 5 bit o specificati dall'ultimo byte di un altro registro. Il valore immediato (il valore finale in output) sarà un numero a 8 bit e può essere "ruotato" in un numero dispari di posizioni; inoltre, l'assembler calcolerà la rotazione per la

costante inserita. Il circuito di BS farà sempre uno shift: infatti il programmatore (o l'assembler) specificherà sempre uno shift logico a sinistra di 0 bit.

La quantità la quale il registro dev'essere "shiftato" è specificata in diversi punti:

- Nel campo d'istruzioni da 5 bit: in questo caso non vi è nessun **overhead** e lo shift è eseguito in un unico ciclo
- Specificata dall'ultimo byte (bottom byte) di un altro registro: richiede un ciclo in più in quanto ARM non ha abbastanza porte per leggere 3 registri in un solo ciclo

Istruzioni di moltiplicazione

Nel pratico, usare un'istruzione di moltiplicazione per moltiplicare di una costante significa **PRIMA** caricare la costante in un registro e poi "aspettare" un numero di cicli interni finché l'istruzione non sia completa. Una soluzione più ottimale può essere individuata utilizzando una combinazione di MOV, ADD, SUB e RSB con shift. La moltiplicazione può essere sempre ricondotta a fattori $2^n + v$ (dove v variabile); nel caso in cui si ha un valore 2^{n+1} , la moltiplicazione può essere svolta in un ciclo.

```
r0=r1*5
r0=r1 + (r1*4) -> ADD r0,r1,r1
                      LSL #2
r2=r3*105
r2=r3 *15*7
r2=r3 * (16-1) * (8-1) --> RSB r2,r3,r3
                      LSL #4
                      RSB r2,r2,r2
                      LSL #3
```

Bisogna stare attenti all'ordine delle operazioni.

Non c'è nessuna istruzione che caricherà immediatamente una costante immediata a 32 bit in un registro prima di non aver effettuato un caricamento dati dalla memoria; bisogna ricordarsi che in ARM32 tutte le istruzioni sono lunghe 32 bit e soprattutto **il flusso delle istruzioni non è utilizzato come se fosse un dato**. Il formato di data processing instruction ha 12 bit disponibile per operand2 (se usato direttamente darebbe soltanto un range di 4096, e invece è utilizzato per salvare costanti di 8 bit, con un range [0-255]); questi bit, ovviamente, possono essere "ruotati" attraverso un numero "pari" di posizioni

```
MOV R0, #0x40,26 (shift di 26)-> MOV r0, #0x1000
```

L'assembler convertirà automaticamente un'istruzione del tipo `MOV R0,#4096` in `MOV R0, #0x1000` facendo effettuare, dal barrel shifter, una rotazione circolare di bit

Nel caso in cui avessimo `11111111` non posso codificare queste informazioni, necessito di uno shift a destra di 2 bit

quindi `0xFF` (0000 0000 0000 0000 0000 1111 1111) effettuo lo shift da destra a sinistra, quindi

-> (1100 0000 0000 0000 0000 0011 1111) `0xc000003F`, questo shift è eseguito con la seguente operazione

```
MOV R0, #FF -> MOV R0, #0xC000003F
```

soltanamente lo shifter prende in ingresso 5 bit (4 codificanti + 1 muto) per lo "shift amount", mentre il resto (8 bit) riguardano esclusivamente il valore. La filosofia RISC è quella di codificare in maniera più "semplice" possibile, e quindi è preferibile aggiungere dei bit laddove sia presente tale necessità.

Cosa facciamo se il nostro valore non è incluso "nel range"? Anche se il meccanismo MOV/MVN può caricare un grande range di costanti in un registro, qualche volta tale meccanismo non genererà la costante richiesta; a tal proposito, l'assembler offre un metodo che permette di caricare una costante da 32bit

```
LDR rd, =numeric constant
```

Se la costante può essere costruita utilizzando sia una MOV che una MVN allora questa sarà l'istruzione generata (**pseudoistruzione**), altrimenti l'assembler produrrà una istruzione LDR con indirizzo relativo al PC per leggere questa costante da un "literal pool"

```
LDR r0, =0x42 -> MOV r0,#0x42  
LDR r0, =0x55555555 -> LDR, r0, [pc, offset to lit pool]
```

ARM prevede due tipi di istruzioni per la moltiplicazione

- `MUL{}{S} Rd,Rm,Rs` ; $Rd=Rm \cdot Rs \rightarrow$ MOLTIPLICAZIONE NORMALE
- `MLA{}{S} Rd,Rm,Rn` ; $Rd=(Rm \cdot Rs)+Rn \rightarrow$ MOLTIPLICAZIONE CON ACCUMULAZIONE

Vi sono alcune limitazioni: Rd ed Rm non possono appartenere allo stesso registro; questo problema può essere aggirato "scambiando" Rm ed Rs intorno. Ciò funziona poiché la moltiplicazione è cumulativa.

Moltiplicazioni-Long e Moltiplicazioni-Long-Accumulate

Le varianti M di ARM contengono hardware con funzionalità estese rivolte alla moltiplicazione. Questo comporta vantaggi come

- Utilizzo dell'algoritmo di Booth a 8 bit
- Metodi di early termination migliorati cosicché il processore possa terminare la moltiplicazione in anticipo quando tutti i termini rimanenti sono

- tutti 0
- tutti 1
- Possono essere ottenuti risultati da 64 bit operando con operandi a 32bit

Le istruzioni sono:

- MULL -> RdHi,RdLo:=Rm*Rs
- MLAL -> RdHi,RdLo:=(Rm*Rs)

A differenza di prima, però, tutti i 64 bit del risultato "contano" e quindi bisogna specificare se gli operandi sono **signed o unsigned**

- UMULL{}{S} RdLo,RdHi,Rm,Rs
 - UMLAL //
 - SMULL //
 - SMLAL//
-

Istruzioni Load/store

L'architettura ARM è Load / Store, quindi prima di utilizzare dei dati dobbiamo "spostarli" dentro i registri (questo significa che non c'è **supporto** alle operazioni "memory-to-memory"). Le istruzioni di base sono le seguenti:

- LDR /STR/LDRB/STRB

ARMv4 aggiunge supporto alle halfword e dati "signed"

- Load and store halfword
 - LDRH /STRH
- Load signed byte / halfword
 - LDRSB / LDRSH

Queste istruzioni supportano l'esecuzione condizionale del tipo:

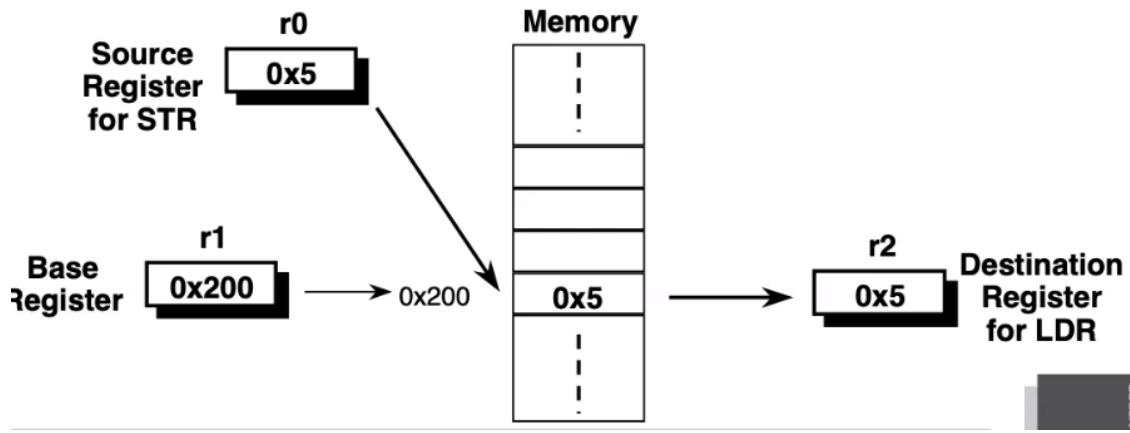
- LDREQB

Sintassi:

- <LDR|STR>{}{} Rd,

La locazione della memoria di cui si vuole ottenere accesso è contenuta in un "base register"; ho bisogno un registro in cui voglio leggere il valore o in cui voglio scrivere e poi necessito di un'espressione per calcolare l'indirizzo di memoria su cui voglio leggere/scrivere.

- STR r0,[r1] -> memorizzo il contenuto di r0 nella locazione puntata dal contenuto di r1
- LDR r2, [r1] -> carico r2 (con locazione di memoria dedicata al contenuto) con locazione puntata dal contenuto di r1



Offset dal Base Register

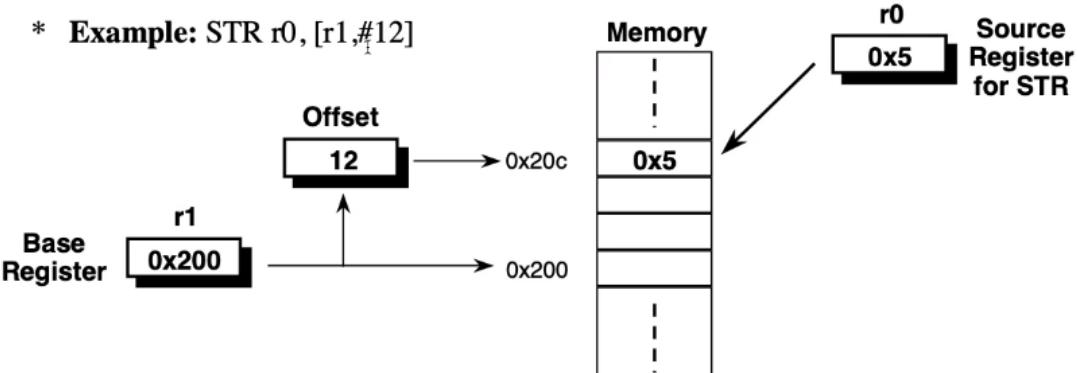
oltre a poter accedere alla locazione attuale contenuta dal base register, queste istruzioni possono accedere all'offset della locazione dal puntatore del base register. Questo offset può essere:

- un valore senza segno da 12 bit immediato
- Un registro, optionalmente shiftato da un valore immediato

questi possono essere sommati o sottratti dal base register e possono essere applicati secondo due modalità

- pre-indexing
- post-indexing

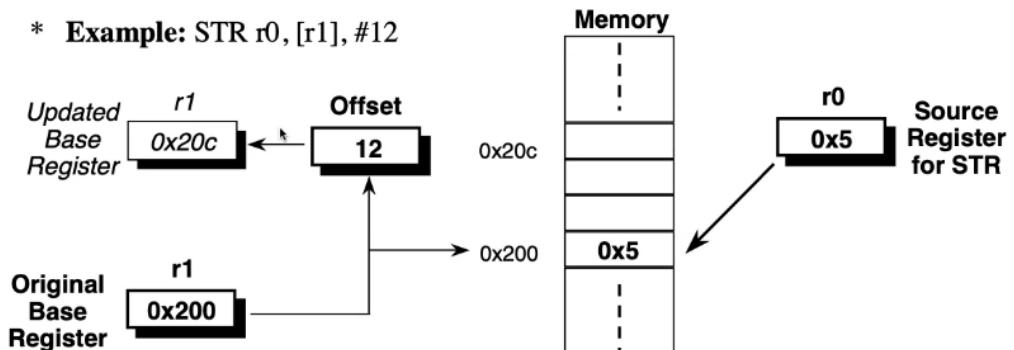
Load and Store Word or Byte: Pre-indexed Addressing



- * To store to location **0x1f4** instead use: STR r0, [r1,#-12]
- * To auto-increment base pointer to **0x20c** use: STR r0, [r1,#12]!
- * If r2 contains 3, access **0x20c** by multiplying this by 4:
 - STR r0, [r1,r2,LSL #2]

Load and Store Word or Byte: Post-indexed Addressing

* Example: STR r0, [r1], #12



* To auto-increment the base register to location 0x1f4 instead use:

- STR r0, [r1], #-12

* If r2 contains 3, auto-incremenet base register to 0x20c by multiplying this by 4:

- STR r0, [r1], r2, LSL #2

Si utilizza il pre-indexing se il base register deve "persistere" come incremento/decremento del registro base ed è effettuato **prima del trasferimento**, al contrario del post-incremento. Il funzionamento in queste due immagini è simile se non identico a quello dei puntatori.

Nel caso del pre-indexing:

- imposto come registro di memorizzazione, R0
- incremento l'indirizzo del base-register (in R1 è 0x200) con offset 12 (quindi 0x200+0x00c, ottenendo 0x20c)
- memorizzo il valore (0x5) e il base-register di R0 viene aggiornato a 0x20c

Al contrario, nel post-indexing

- imposto (identicamente) come registro di memorizzazione, R0
- a differenza del pre-indexing, se aggiorno l'indirizzo del base-register con la medesima operazione, il base-register ad essere aggiornato sarà quello di R1
- memorizzo il valore (0x5) e il base-register di R0 rimane intatto

Modalità d'indirizzamento

Possiamo immaginare di avere un'array; il primo elemento puntato per il contenuto è l'elemento 0. Se vogliamo accedere a un delemento particolare, possiamo utilizzare il pre-indexed addressing:

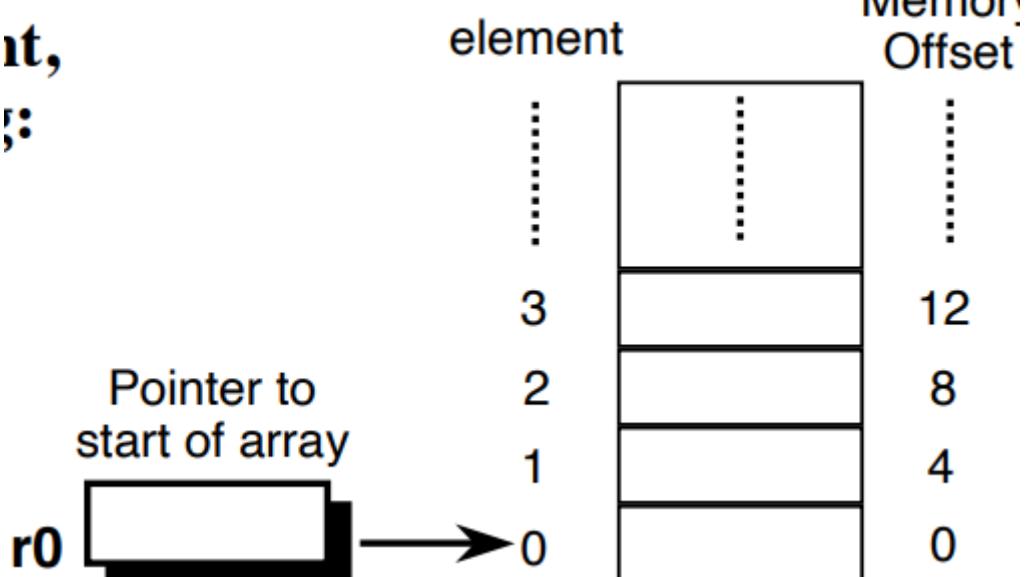
R1 è l'elemento che vogliamo
LDR r2, [r0,r1, LSL #2]

se invece vogliamo scorrere tutti gli elementi degli arrei (per produrre, ad esempio, una somma di tutti gli elementi presenti al suo interno), allora possiamo usare il post-indexed addressing

R1 è l'indirizzo dell'elemento corrente
LDR r2, [r1], #4

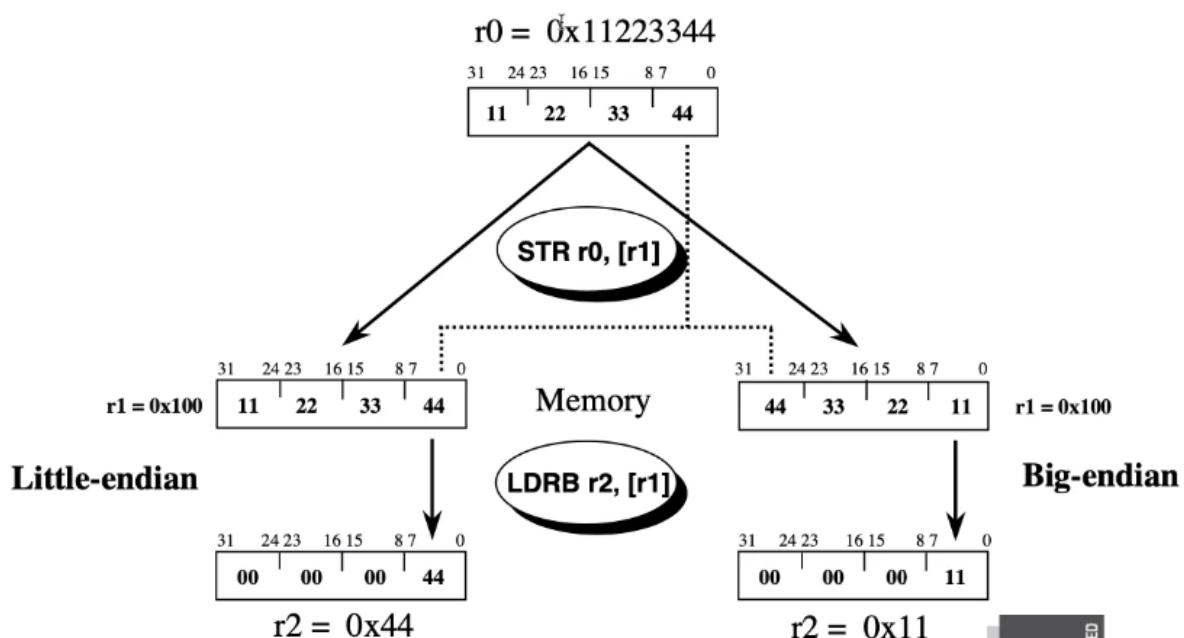
it,

;



Lettura in memoria (little-endian big-endian)

Endianess Example



Quando noi memorizziamo un valore in un registro, o "scriviamo" in little-endian o in big-endian. In Little-endian il MSB si trova all'ultimo posto, mentre in big-endian è al contrario (è come se scrivessi su carta). Quando troviamo elementi trovati in modalità big-endian, li leggiamo, ovviamente, in maniera più semplice trattandosi di lettura classica alfabetica, mentre tipicamente i dati sono salvati e letti in modalità (molto spesso) "little-endian"

Esempio di codice con il post-indexing

```
ADD r0, r0, r1, LSL#2 -> inizializzo l'indirizzo p=&array[x]
ADD r2, r0, r2, LSL#2 -> inizializzo l'indirizzo p=&array[x+1]
MOV r1, #0 -> inizializzo l'accumulatore sum a 0 -> sum=0

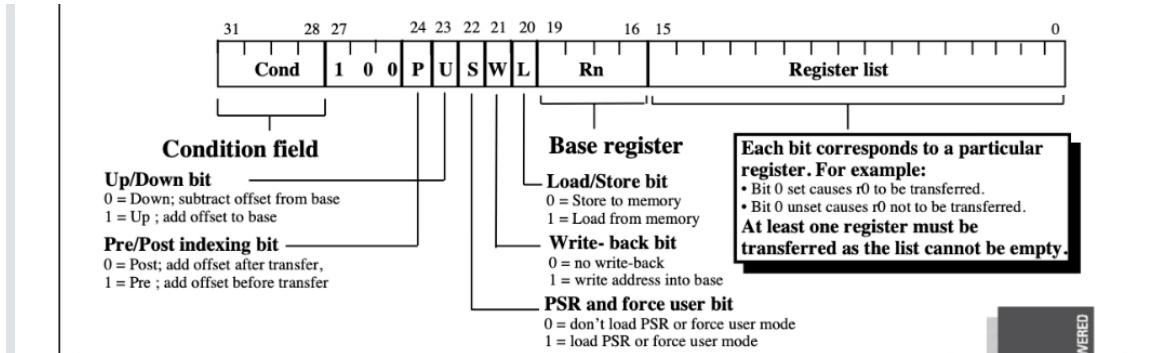
loop
    LDR r3, [r0], #4 -> post-indexing= incremento il base register di 0x04
    ADD r1, r1, r3 -> aggiungo l'indirizzo precedente all'accumulatore
    CMP r0, r2 -> effettuo una comparazione (sono arrivato ad x+n-1?)
    BLT loop -> se è falso ripetere
```

N.B: utilizzare int per i puntatori è sbagliato

Block Data Transfer

Le istruzioni LDM / STM permettono di trasferire dalla memoria al registro (o al contrario) da 1 a 16 registri. I registri trasferiti possono essere:

- qualsiasi subset del banco corrente di registri
- qualsiasi subset del banco di registri appartenente alla user mode quando si è in modalità con "privilegi" (come la modalità supervisor, ad esempio)



Il base register è utilizzato per determinare dove dovrebbe avvenire l'accesso alla memoria; vi sono 4 differenti modalità d'indirizzamento che permettono incremento/decremento inclusivo/esclusivo della locazione del base register. Quest'ultimo può anche essere aggiornato dopo il trasferimento, specificando il simbolo '!>.

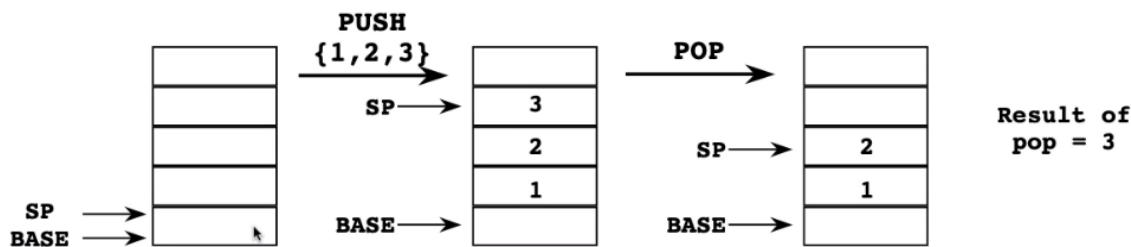
N.B: il numero di registro più basso è sempre trasferito da/alla locazione più bassa in memoria a cui si ha accesso

Stack

In ARM noi non abbiamo un registro di stack strutturato materialmente, a differenza di altri ISA. Generalmente uno stack è una porzione di memoria (approssimata anche a struttura dati) dove vengono aggiunti nuovi dati e sono "pushati" in cima ad esso, e quando vengono "estratti" dalla cima, ovviamente la dimensione dello stack diminuisce; la politica d'inserimento/estrazione dati dello stack è definita "**LIFO** (Last In / First Out)". Sono definiti quindi 2 puntatori che "limitano lo stack"

- Stack pointer: punta sempre alla cima dello stack
- base pointer: punta sempre la base

soltamente a muoversi è sempre lo stack pointer, mentre il base pointer indica o comunque definisce un limite inferiore di memoria a cui lo stack pointer non può andare oltre.



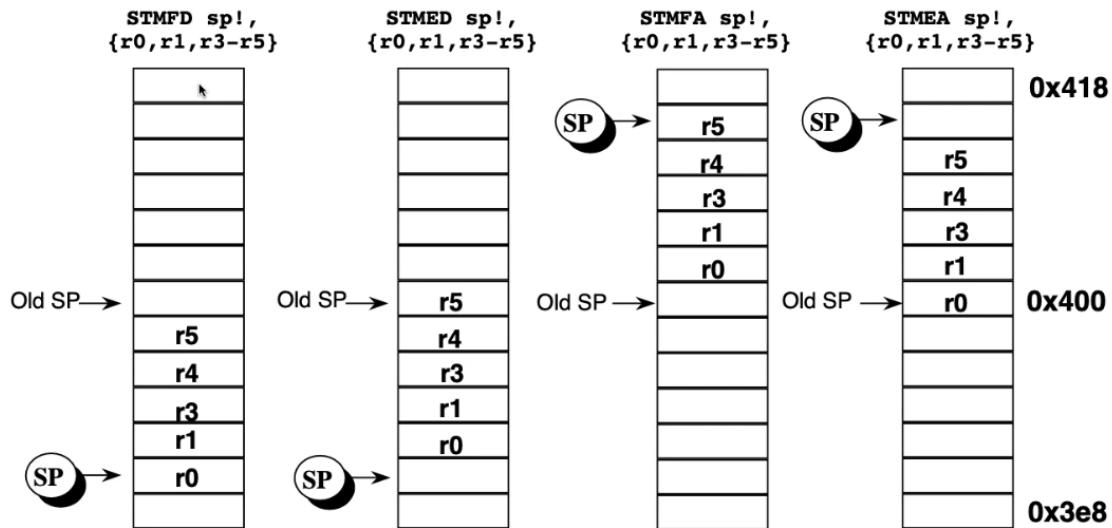
Operazioni dello stack

Soltamente, uno stack giù cresce in memoria, con l'ultimo valore "pushato" in fondo all'indirizzo più basso. ARM contiene il supporto agli stack ascendenti dove la struttura dello stack cresce in memoria (vedere la figura precedente al contrario). Il valore dello stack pointer può sia **puntare all'ultimo indirizzo occupato (full stack)** (e quindi necessita un **pre-decremento**) oppure **puntare al prossimo indirizzo occupato (empty stack)** (e quindi necessita un **post-decremento**). Il tipo di stack condiziona il tipo di istruzioni utilizzabili:

- STMFD / LDMFD : Full descending stack
- STMFA / LDMFA : Full ascending stack
- STMED / LDMED : Empty descending stack
- STMEA / LDMEA : Empty Ascending stack

N.B: il compilatore ARM utilizzerà sempre un Full descending stack

Stack Examples



Stack e Subroutine

generalmente e molto spesso, gli stack sono utilizzati per implementare delle funzioni le quali necessitano un "workspace" (ovvero un range di spazio dedicato solo all'esecuzione di subroutine). Ogni registro di cui si ha necessità può essere pushato dentro lo stack all'inizio della subroutine ed "estratto" di nuovo alla fine così da ripristinarli prima di ritornarli al caller

```

STMFD sp!, {r0-r12, lr}          ; stack all registers
.....
.....
LDMFD sp! {r0-r12, pc}          ; carica tutti i registri e ritorna
automaticamente

```

N.B: se l'istruzione di pop ha il bit S settato (utilizzando '^') il trasferimento del PC, laddove sia in una modalità privilegiata, causerà anche la copia del SPSR nel CPSR

Viene sfruttato un trucchetto: la vicinanza dei registri r14 (lr) ed r15(pc). In cima del branch salvo i valori e l'ordine in cui vengono caricati i registri; alla fine della "procedura" faccio in modo che il pc corrisponda al lr cosicché possa ritornare allo stato precedente, ovvero nel momento in cui viene chiamata l'istruzione STMFD

funzionalità dirette del block data transfer

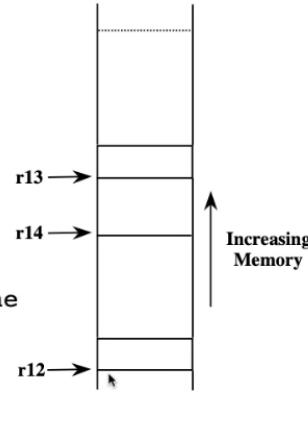
quando LDM/STM non sono utilizzati per implementare gli stack, è più chiaro specificare con chiarezza quali funzionalità possiede l'istruzione (ad esempio, indicare di quanto incrementare/decrementare il base pointer, prima o dopo l'accesso in memoria). Per poterlo fare, LDM/STM supporta un modo ulteriore (sintattica) in aggiunta a quelle dello stack

- STMIA / LDMIA : Increment After
- STMIB / LDMIB: Increment Before
- STMDA / LDMDA : Decrement After
- STMDB / LDMD: Decrement Before
- Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

```

; r12 points to the start of the source data
; r14 points to the end of the source data
; r13 points to the start of the destination data
loop    LDMIA   r12!, {r0-r11} ; load 48 bytes
        STMIA   r13!, {r0-r11} ; and store them
        CMP     r12, r14       ; check for the end
        BNE     loop           ; and loop until done

```



- This loop transfers 48 bytes in 31 cycles
- Over 50 Mbytes/sec at 33 MHz

Esempio

Le architetture sono ottimizzate per fornire dati in sequenza in maniera efficiente: l'unica penalizzazione è il primo accesso (vi è una latenza superiore rispetto agli accessi successivi); quindi è sempre meglio chiedere un "pacco" di valori "sequenziali"

Esercizi

Li carico nel drive, comunque sono svolti giù

Esercitazione ARM

Esercizio 1

```
MOV    R0, #2
MOV    R1, #3
ADD    R2, R0, R1
```

Esercizio 2

```
MOV    R0, #2
MOV    R1, R0
LSL    R0, R0, #5
ADD    R0, R0, R1
```

Esercizio 3

```
MOV    R0, #2
MOV    R1, R0
LSL    R0, R0, #4
RSB    R0, R1, R0
```

Esercizio 5 (A)

```
ldr    r0, =data ;"data" è la label riferita ai dati
ldr    r1, =data_size ;stessa cosa ma riferita alla dimensione
mov    r2,r0 ;copiamo i dati in r2
mov    r0, #0 ;reimpostiamo a 0 r0
teq    r1, #0 ; (test equivalence (si aggiornano N e Z del CPSR))
beq    end_loop ; salta se la condizione precedente è verificata
loop   ldr    r3, [r2], #4 ;carica (tramite post-indexing) il valore puntato
       dal base register r2
       add    r0, r0,r3      ;somma elemento in r3
       subs   r1,r1,#4      ;sottrai a r1 4 byte e aggiorna le flag nel CPSR
       bne    loop ;salta a loop se la flag z non è settata
end_loop
       end

data     DCD    1,2,3,4 ; int data[] ={1,2,3,4} : questa è una definizione
       della label, visual utilizza questa e varia in base all'interprete
data_end  FILL   0 ;fine dell'array
data_size EQU    data_end-data ;indirizzo
```

Esercizio 5 (B)

```
ldr    r0, =data ;"data" è la label riferita ai dati
ldr    r1, =data_size ;stessa cosa ma riferita alla dimensione
mov    r2,r0 ;copiamo i dati in r2
mov    r0, #0 ;reimpostiamo a 0 r0
loop   cmp    r1,r2; ;sottrai r1 ed r2, aggiorni le flag nel CPSR
       beq    end_loop
       ldr    r3, [r2], #4 ;carica (tramite post-indexing) il valore puntato
       dal base register r2
```

```

        add    r0, r0,r3      ;somma elemento in r3
        b     loop ;salto incondizionato a loop
end_loop
        end

data      DCD    1,2,3,4 ; int data[] ={1,2,3,4} : questa è una definizione
della label, visual utilizza questa e varia in base all'interprete
data_end   FILL      0 ;fine dell'array

```

Esercizio 5 (C)

questo modello (in specifico) ottimizza l'utilizzo dei registri: gli argomenti delle funzioni, in ARM, sono i registri stessi

```

ldr r0, =data
ldr r1, =data_end
b1 sum ;branch with link (lr=pc)
end

sum
    mov r2,r0
    mov r0, #0
loop
    cmp r1,r2
    beq end_loop
    ldr r3, [r2], #4
    add r0,r0,r3
    b loop
end_loop
    mov pc,lr ; ritorno (pc=lr)
end

data      DCD    1,2,3,4
data_end   FILL      0

```

Esercizio 5 (D)

```

ldr r0, =data
ldr r1, =data_end
mov r2,r0
mov r0,0
loop cmp r1,r2
    ldrne r3, [r2], #4
    addne r0, r0, r3
    bne loop
end_loop
end

data      DCD 1,2,3,4
data_end   FILL 0

```

Esercizio sul caricamento "a blocchi"

crea un programma che sposti:

r0 in r3

r1 in r4

r2 in r6

r3 in r5

r4 in r0

r5 in r1

r6 in r2

utilizza operazioni di tipo "full descending stack"

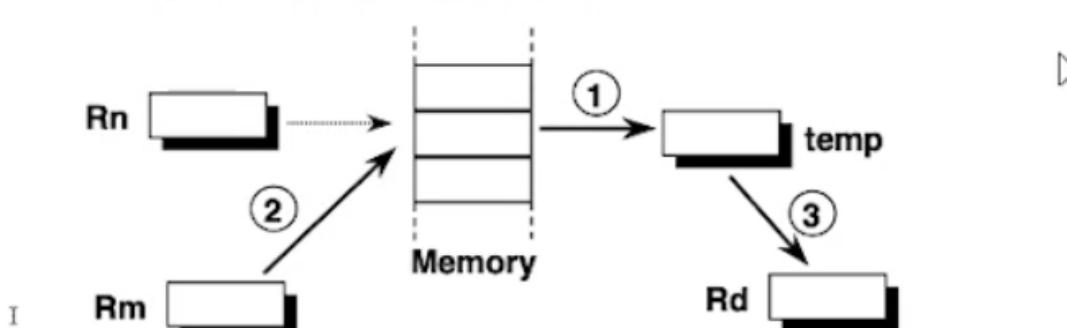
```
STMFD sp!, {r0-r6} ;stack pointer full descending non vuoto in preincremento  
(store)  
LDMFD sp!, {r3,r4,r6} ;r0 in r3, r1 in r4, r2 in r6  
LDMFD sp!, {r5} ;r3 in r5  
LDMFD sp!, {r0-r2} ;r4 in r0, r5 in r1, r6 in r2
```

Altre operazioni in ARM

SWAP e SWAP byte

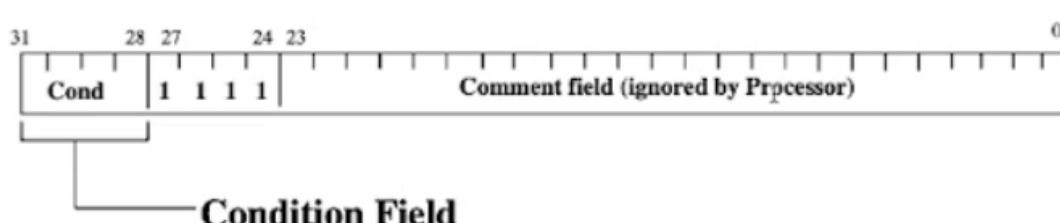
Questa operazione racchiude due operazioni atomiche, ovvero una lettura in memoria seguita un'operazione di scrittura in memoria che muove byte o quantità di word tra i registri. La sua sintassi è:

SWP{}{B} Rd, Rm, [Rn]



- * Thus to implement an actual swap of contents make Rd = Rm.
- * The compiler cannot produce this instruction.

SWI (software interrupt)



nelle ultime versioni di ARM non è utilizzato in questa forma, ma generalmente mantiene lo stesso comportamento. Questa istruzione viene utilizzata per effettuare il **context switching** in una modalità privilegiata (il meccanismo consiste nel causare una cattura dell'eccezione nell'hardware vector SWI, causando quindi lo switch in supervisor mode e la chiamata dell'SWI exception handler; da ciò ne consegue anche il salvataggio dello stato). L'handler analizza il campo di commento per stabilire quale operazione è stata richiesta. Tramite il meccanismo SWI, un sistema operativo può implementare diversi set di operazioni "privilegiate" richiesti da un'applicazione eseguita in modalità utente.

PSR transfer instruction

tipicamente il CPSR/SPSR non è un registro general purpose e per tale motivo non può essere utilizzato per operazioni di qualunque tipo. **MRS** e **MSR** permettono il trasferimento del CPSR/SPSR nel registro appropriato o in registri general purpose (può essere trasferito tutto il registro o solo le flag)

Sintassi

MRS{} Rd, ; Rd=

MSR{},Rm ; =Rm

MSR{},Rm ; =Rm

dove = CPSR, CPSR_all, SPSR, SPSR_all

dove =CPSR_flg o SPSR_flg

si può utilizzare anche

MSR{}, #immediate



Queste istruzioni possono essere utilizzate per preservare lo stato corrente e quando si vuole mantenere "intaccato" il loro contenuto

Bisogna seguire la strategia "read-modify-write" :

- trasferire il PSR in un registro gp utilizzando MRS
- modificare i "bit rilevanti"
- trasferire il valore aggiornato nel PSR utilizzando MSR

N.B: in user mode può essere letto tutto il contenuto, ma la modifica è permessa solo per i bit di flag

Data processing instruction encoding

```

Chute dip$ cat mov-movn-example.s
    mov    r0, #0xFF00FFFF
    mvn    r0, #0x00FF0000
    mvn    r0, #0xFF, 16
    mov    r0, #0x00FF0000
    mvn    r0, #0xFF00FFFF
    mov    r0, #0xFF, 16

Chute dip$ arm-none-eabi-as -g mov-movn-example.s -o mov-movn-example.o

```

```

00000000 <.text>:
 0: e3e008ff      mvn    r0, #16711680 ; 0xff0000
 4: e3e008ff      mvn    r0, #16711680 ; 0xff0000
 8: e3e008ff      mvn    r0, #16711680 ; 0xff0000
 c: e3a008ff      mov     r0, #16711680 ; 0xff0000
10: e3a008ff      mov     r0, #16711680 ; 0xff0000
14: e3a008ff      mov     r0, #16711680 ; 0xff0000
> ls -la 1-mov-movn-example.o
-rw-r--r-- 1 dip staff 576 2 Nov 10:55 1-mov-movn-example.o
> arm-none-eabi-objdump -d 1-mov-movn-example.o

1-mov-movn-example.o:      file format elf32-littlearm

Disassembly of section .text:
00000000 <.text>:
 0: e3e008ff      mvn    r0, #16711680 ; 0xff0000
 4: e3e008ff      mvn    r0, #16711680 ; 0xff0000
 8: e3e008ff      mvn    r0, #16711680 ; 0xff0000
 c: e3a008ff      mov     r0, #16711680 ; 0xff0000
10: e3a008ff      mov     r0, #16711680 ; 0xff0000
14: e3a008ff      mov     r0, #16711680 ; 0xff0000

```

Un'ABI (**Application Binary Interface**) è uno standard che definisce una sorta di "mapping" tra concetti di basso livello, in linguaggio di "alto livello" e la capacità di un specifico hardware/OS di una macchina; un ABI quindi include:

- come i **data type** vengono gestiti in memoria (per ogni tipologia di linguaggio di programmazione)
- come funzionano le **chiamate di funzioni**
- Come **avviene l'inizializzazione/avvio di un programma**

In semplici parole definisce **come le funzioni / procedure passano informazioni tra di loro utilizzando i registri**. In ARM è definita come "EABI" (embedded application binary interface).

N.B: ogni architettura ha diversi ABI dedicati a differenti linguaggi di programmazione

Le istruzioni ML per caricare un "immediato" in un registro sono MOV Rd, #Imm Rotate e MVN Rd, #Imm, Rotate. L'assemblatore accetta anche MOV Rd, #LegalValue32 (stessa cosa vale per MVN)

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
1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	x	1	1	1	1		
E		3			E			0		0		0		0		0		8		F		F									
al		i		mvn						r0		Rotate=8x2=16								Imm=0xFF											

Cond 0 0 I OpCode S Rn Rd Operand2

Assembly instruction	ML encoding	Encoded instruction in assembly
mov r0, #0xFF00FFFF		
mvn r0, #0x00FF0000	E3E008FF	mvnal r0, #0xFF, 16
mvn r0, #0xFF, 16		

'E' (1110) codifica la condizione 'AL'. I bit 27-26 sono fissati e rappresentano l'opcode (ovvero quale istruzione sta per essere codificata). Il bit 25 codifica 'i', ovvero che il secondo operando è un "immediato" (questo cambia il significato dei 12 LSB, dove da 0 a 7 viene definito l'immediato e da 8 a 11 la rotazione). Riprendendo i concetti, sappiamo sempre che Rd è il registro di destinazione (sempre da specificare) ed Rn il registro sorgente (non è codificato ma viene messo un valore arbitrario, in questo caso è 0)

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
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0		
E		1			A			0		1		0		0		E															
al		r		mov						r1		1		ROR			r0														

Cond 0 0 I OpCode S Rn Rd Operand2

Assembly instruction	ML encoding	Encoded instruction in assembly
ror r1, r0, #1		
mov r1, r0, ROR #1	E1A010E0	moval r1, r0, ROR #1

in questo caso (immagine sopra) cambiano i bit da 27 a 25: l'operand2 è un registro e quindi i 12 bit MSB vengono interpretati diversamente: i 4 bit MSB vengono utilizzati per codificare il registro d'interesse, i bit da 7 a 4 codificano ROR

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
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0		
E		1			A			0		1		1		8																	
al		r		mov						r1		3		LSL			r0														

Assembly instruction	ML encoding	Encoded instruction in assembly
lsl r1, r0, #3		
mov r1, r0, LSL #3	E1A01180	moval r1, r0, LSL #3

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
1	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0	1	1	1	1	
E		0			4					5			4			2			3				7								
al		r	sub						r5			r4			2			LSR				r7									
Cond	0	0	I	OpCode	S	Rn	Rd																								
Assembly instruction				ML encoding				Encoded instruction in assembly																							
sub r4,r5,r7,LSR r2				E0454237				subal r4, r5, r7, LSR r2																							

in questo caso viene effettuata un'operazione di sottrazione: il bit 4 è impostato 1 in manieratale da prendere il registro

Branch instruction encoding

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
1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E		A		0		0		0		0		0		0		0		0		0		0		0		0		0		0	
al		B																													
Cond		L																													
Assembly instruction				ML encoding				Encoded instruction in assembly																							
b label				EA000000				bal c																							

L'istruzione di branch effettua un salto alla label specificata. Dal bit 23 al bit 0 è contenuto il valore di offset relativo all'indirizzo dell'istruzione di branch PIU' 2 word: quindi bisogna shiftare l'offset di due posizioni a sinistra e poi viene sommato al PC. Il bit 24 rappresenta se il PC deve essere copiato nel LR (BL se 1) oppure no (B se 0)

Lezione 12 - 04/11/2021

ARM Programming

Partendo dalla codifica in linguaggio macchina, è facile decodificare in istruzioni assembly. Partendo dal MSB si può **decodificare** la condizione dell'istruzione ARM (campo Cond). Se capiamo bene come funziona l'hardware, capiremo meglio come programmare

ESERCIZIO: Decodificare le seguenti istruzioni ML

PRIMA ISTRUZIONE: 0xE0820101

Partendo dal MSB e orientandoci con la tabella di codifica, possiamo orientarci su quale istruzione viene codificata:

E è decodificato in 1110 -> condizione AL

0 è decodificato in 0000 -> generalmente possiamo orientarci tra "Multiply" e "Data processing": Multiply richiede 6 bit impostati a 0 ma il byte successivo è codificato in 9, quindi escludiamo Multiply; l'istruzione sarà di Data processing (MOV/MVN). Il terzo bit (bit I, ovvero quello che definisce se un valore sia immediato o meno) è impostato a 0 quindi il

secondo operando sarà un registro. In base al bit I, gli LSB assumono una codifica/decodifica differente

8 è decodificato 1000 -> i primi 3 bit insieme all'ultimo bit del byte precedente codificano l'**opcode**: in questo caso avremo 0100 ovvero ADD

2 è decodificato in 0010 -> il registro Rd sarà R2

0 è decodificato in 0000 -> Il registro Rm sarà R0

a questo punto sappiamo che l'operazione sarà del tipo ADD R0,R2... cosa abbiamo dopo? Partiamo dal LSB (ultimo bit a destra per poi salire verso sinistra)

1 (LSB) è 0001 -> si tratta del registro R1

0 è 0000 -> prendiamo in considerazione soltanto il secondo e terzo bit: viene codificato LSL (0x00). L'ultimo bit a destra se è uguale 1, allora significa che l'operazione (in questo caso di shift laterale a sinistra) verrà effettuata da un valore proveniente dal registro (quindi verranno codificati i successivi 4 bit e non 5, come in questo caso)

1 è 0001 -> in questo caso verranno presi i 3 LSB e il MSB del byte precedente (0010) che codificherà 2

Quindi alla fine avremo ADD, R0,R2, LSL #2

SECONDA ISTRUZIONE:

0xE12FFF1E

partendo sempre dal MSB:

E -> 1110 -> condizione AL

1 -> 0001 -> si utilizza un registro

2 -> 0010 -> prendo il LSB del byte precedente e i 3 LSB di questo byte ottenendo 1001 -> codifico BX

tutti gli F codificano 1111 -> BX è giustificato

1 -> 0001

E -> 1110 -> nel caso in cui si utilizza l'istruzione di branch (in questo caso BX), questo byte codificherà il registro in cui è contenuto l'indirizzo di ritorno (generalmente è il LR, infatti 1110 = 14 in decimale)

Embedded Software Development (ripasso)

Quando viene definito un hardware embedded e quando vengono scritti applicativi su di/per esso, vengono considerati diversi aspetti. L'hardware solitamente consiste di:

- ISA
- Organizzazione in memoria
- Periferiche (IRQ, timer, I/O, GPIO, ADC, DAC)
- Il SoC (System on a Chip) include tutti quelli precedenti come se fossero integrati, tranne per alcune periferiche

L'obiettivo è sempre scrivere codice "ottimizzato" e realizzare dispositivi che siano meno "power-hungry". Dal punto di vista software gli aspetti da considerare sono:

- linguaggio di programmazione: quelli più utilizzati sono sempre di linguaggi di medio-alto livello, ma l'utilizzo di linguaggi assemblativi è inevitabile
- Sistema operativo: è fondamentale per gestire gli eventi I/O e differenti interrupt, sia hardware che software. Tipicamente questi kernel sono minimali e di tipo "monolitico" proprio per aumentare l'efficienza energetica e la velocità (condizione necessaria in sistemi di elaborazione real-time)
- ABI (Application Binary Interface), nel caso di arm EABI (Embedded Application Binary Interface): Come già descritto in precedenza l'ABI è una sorta di "mapping" tra basso ed alto livello e descrive **come vengono chiamate le funzioni (Calling convention), le chiamate di sistema (Syscall), il formato binario dei file e le librerie e i tipi di dato (nativi) manipolabili.**

Possiamo adottare diversi modelli di programmazione; quello più familiare a noi è sicuramente la **compilazione nella macchina target**: scriviamo software nella macchina, compiliamo il sorgente nella macchina ed eseguiamo sempre sulla stessa macchina..questo quindi non garantisce il funzionamento su altre macchine se non la target stessa. Il modello di **cross-compilation** ci permette di compilare il codice e di farlo funzionare su più macchine possibile; è la soluzione più comune , mentre **l'interazione ambientale nella macchina target** permette allo sviluppatore di agire direttamente "real-time" sul funzionamento dei registri.

Compilazione

Interprete

Un interprete è eseguito nella macchina target: il codice può essere facilmente testato sulla macchina target, quindi si può sperimentare con più facilità. Questo modello, però potrebbe non essere **adatto** per macchine con **restrizioni alle risorse**. L'interprete **Forth** può essere implementato per macchine a risorse ridotte in modalità **bare-metal** (senza OS).

Calling convention (caratteristica dell'EABI)

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

nell'EABI i registri vengono definiti in questa maniera (convenzioni)

Per scrivere software su ARM (32 bit), seguiamo le **procedure standard di chiamata per l'architettura ARM** (AAPCS). Quali sono le basi?

- I primi quattro registri sono utilizzati per passare valori di argomento nelle subroutine e per ritornare valori da una funzione. Possono servire anche come variabili temporanee dentro una routine
- I registri R4-R8, R10 ed R11 (v1-v5, v7,v8) sono usati per mantenere valori delle variabili locali appartenenti ad una routine
 - (una subroutine deve mantenere il contenuto dei registri r4-r8,r10,r11 e l'SP)
- Una double-sized word è passata in due registri consecutivi (r0 ed r1, o r2 ed r3)
- Facendo riferimento allo stack, sappiamo che esso è una porzione contigua di memoria ed è utilizzata per mantenere/memorizzare variabili locali e per passare argomenti aggiuntivi alle subroutine quando vi sono registri d'argomento insufficienti. Un allineamento di stack di 8 byte è un requisito dell'AAPCS

Come viene gestito il valore di ritorno?

```
/* -- first.s */
/* This is a comment */
.global main /* 'main' is our entry point and must be global */

main:          /* This is main */
    mov r0, #2 /* Load 2 into register r0 */
    bx lr      /* Return from main */
pi@raspberrypi:~/EmbeddedSystems/2021-2022/2021-11-04/00$ cc first.c
cc: error: first.c: File o directory non esistente
cc: fatal error: no input files
compilation terminated.
pi@raspberrypi:~/EmbeddedSystems/2021-2022/2021-11-04/00$ cc first.s
pi@raspberrypi:~/EmbeddedSystems/2021-2022/2021-11-04/00$ ./a.out
pi@raspberrypi:~/EmbeddedSystems/2021-2022/2021-11-04/00$ echo $?
2
pi@raspberrypi:~/EmbeddedSystems/2021-2022/2021-11-04/00$ less first.s
/* -- first.s */
/* This is a comment */
.global main /* 'main' is our entry point and must be global */

main:          /* This is main */
    mov r0, #2 /* Load 2 into register r0 */
    bx lr      /* Return from main */
first.s (END)
```

cc è un comando presente nella toolchain del compilatore di GCC

The screenshot shows a terminal window with several tabs at the top. The active tab displays ARM assembly code. The code includes comments explaining the assembly instructions, such as the SuperVisor Call (SVC) instruction which must be in register r0. The assembly code is as follows:

```
1 @ Example 2
2 @
3 @ assemble with:
4 @ as -g 2.s -o 2.o
5 @ ld 2.o -o 2
6
7 .global _start @ export
8
9 _start:
10    mov r1, #5
11    add r3, r1, r1, LSL #2 @ r3=r1+r1*4=r1
12    mov r0, r3      @ the argument for the exit system call must be in r0
13    mov r7, #1      @ r7=1, this selects the exit system call
14    svc 0          @ SuperVisor Call
~
~
~
~
~
~
~
"2.s" 14L, 306C
```

6,0-1

Tut

esempio di codice: quando viene utilizzata la SuperVisor Call (SVC, equivalente a SWI), la disposizione dei registri cambia

Lezione 13 - 16/11/2021

Esempi di codice (ARM ASM, RASPBERRY PI, GNU/LINUX)

EXAMPLE 1 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 1
@
@ compile with:
@ cc -g 1.s -o 1

.global main      @ symbol main must be exported for linking with
                  @ the C-runtime (crt)

main:
    mov    r1, #2      @ assign 2 to r1
    mov    r2, #3      @ assign 3 to r2
    add    r3, r1, r2  @ assign r1+r2 to r3
loop:   b     loop      @ never ending loop
```

piccola variazione rispetto al codice precedente: questo è un loop infinito utilizzando le convenzioni del C. Seguendo tali convenzioni, si riesce a rendere il codice "portabile" in varie macchine

```

45 ?      S<    0:00 [bioset]
46 ?      S<    0:01 [kworker/0:1H]
48 ?      S<    0:00 [bioset]
63 ?      S     0:01 [jbd2/sda2-8]
64 ?      S<    0:00 [ext4-rsv-conver]
237 ?    Ssl   0:02 /lib/systemd/systemd-timesyncd
257 ?    Ss    0:00 avahi-daemon: running [raspberrypi.local]
259 ?    Ss    0:02 /usr/bin/dbus-daemon --system --address=systemd: --n
262 ?    S     0:00 avahi-daemon: chroot helper
264 ?    Ss    0:02 /lib/systemd/systemd-logind
266 ?    Ssl   0:00 /usr/sbin/rsyslogd -n
269 ?    Ss    0:01 /usr/sbin/thd --triggers /etc/triggerhappy/triggers..
271 ?    Ss    0:00 /usr/sbin/cron -f
377 ?    Ss    0:00 /sbin/dhcpcd -q -w
395 ?    Ss    0:00 /usr/sbin/sshd -D
398 ?    Ss    0:02 /bin/login --
400 tty1   Ss    0:00 /bin/login --
496 ?    Ss    0:02 /lib/systemd/systemd --user
500 ?    S     0:00 (sd-pam)
505 ttyAMA0 S     0:14 -bash
565 tty1   S     0:00 -bash
9632 ?   Ss    0:00 /lib/systemd/systemd-udevd
3740 ?   Ss    0:02 /lib/systemd/systemd-journald
4281 ?   S     0:00 [kworker/0:0]
4361 ?   S     0:00 [kworker/u2:1]
4374 ?   S     0:04 [kworker/0:2]
4375 ?   S     0:00 [kworker/0:1]
4384 ttyAMA0 R+   0:17 ./1
4386 tty1   R+   0:00 ps ax

```

digressione su TTY: stampa lo standard dell'input

```

@ Example 2
@
@ assemble with:
@ as -g 2.s -o 2.o
@ ld 2.o -o 2

.global _start @ export

_start:
    mov    r1, #3
    add    r3, r1, r1, LSL #2 @ r3=r1+r1*4=r1
    mov    r0, r3            @ the argument for the exit system call must be in r0
    mov    r7, #1            @ r7=1, this selects the exit system call
    svc    0                @ SuperVisor Call

```



in questa porzione di codice vengono effettuate una serie di operazioni: secondo le convenzioni, viene spostato il valore di r3 in r0 e infinite viene selezionata l'uscita tramite r7=1 e svc 0. Il programmatore utilizza liberamente R0,R1,R2,R3 e non necessitiamo di preservare il contenuto. Non stiamo utilizzando il compilatore C ma l'assembler: 'as -g 2.s -o 2.0' serve per assemblare, la parte successiva è "ld 2.0 o 2" dove viene effettuato il linking e lo effettua al CRT (C Runtime) per ottenere un eseguibile. Il processo di linking non è semplicemente ciò che è stato detto prima: bisogna effettuare una serie di operazioni come la comparazione bit-a-bit.

La regola generale è che i registri devono essere utilizzati secondo le convenzioni definite dal progettatore. Le chiamate alle funzioni non hanno sempre la stessa necessità di chiamare gli stessi registri; altri necessitano 0 argomenti, altri n-argomenti.

GDB

Il GBD è il tool di debugging per sistemi GNU e può principalmente svolgere 4 funzioni

- Eseguire il programma, permettendo di specificare dei parametri
- Fermare il programma alle condizioni desiderate dal programmatore
- Esaminare ciò che è successo quando il programma si ferma
- Cambiare elementi in corso d'opera

```
pi@raspberrypi:~/EmbeddedSystems/2021-2022/2021-11-16/02$ gdb 2
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from 2...done.
(gdb)
```

EXAMPLE 3 (ARM ASM, RASPBERRY PI, GNU/LINUX)

```
@ Example 3
@
@ assemble with:
@
@ as -g 3.s -o 3.o
@ ld 3.o -o 3
@

@
@ Definitions
@
@ void exit(int status)
SYSCALL_EXIT=1

.global _start

_start:
    ldr r0, =#4096
    ldr r1, =#6000
    mul r0, r1, r0
    mov r7, #SYSCALL_EXIT
    svc 0      @ SuperVisor Call
```

in questo codice viene definita globalmente SYSCALL_EXIT impostata ad 1: non è nient'altro che una variabile globale e che ha la stessa valenza di un'immediato.

EXAMPLE 4 (ARM ASM, RADISCOPE RTI, QEMU/LINUX)

- ▶ Write an assembly program that prints a message (e.g. "Hello world!"), checks that all the bytes have been put out, and exits with the proper code
- ▶ This is about using the write system call with **stdout**, referencing local values with synonyms, section markers, definition of constant strings in data section, and expressions

```
@ Example 4
@
@ assemble with:
@
@ as -g 4.s -o 4.o
@ ld 4.o -o 4
@

@
@ Definitions:
@
@ void exit(int status)
@
SYSCALL_EXIT=1
@
@ ssize_t write(int fd, const void *buffer, size_t size)
@
SYSCALL_WRITE=4
@
@ each process finds the three stdio file descriptors
@ open at start:
@
@ 0: stdin
```

vengono specificate le chiamate di sistema: l'exit ha 1 mentre write ha 4. Write è una funzione generica che accetta in input il file descriptor, il buffer e la dimensione della memoria su cui si andrà a scrivere.

30		30,0-1	17%
31	.text	31,1	20%
32		32,0-1	23%
33	.global _start	33,1	25%
34	_start:	34,1	28%
35		35,0-1	30%
36	mov r0, #stdout	36,1-4	33%
37	ldr r1, =message	37,1-4	35%
38	ldr r2, =message_len	38,1-4	38%
39	mov v1, r2	39,1-4	41%
40	mov r7, #SYSCALL_WRITE	40,1-4	43%
41	svc 0	41,1-4	46%
42		42,0-1	48%
43	cmp r0, v1	43,1-4	51%
44	moveq r0, \$0	44,1-4	53%
45		45,0-1	56%
46	@ exit(int status):	46,1	58%
47	mov r7, #SYSCALL_EXIT	47,1-4	61%

```

46 @ exit(int status):          46,1      58%
47     mov r7, #SYSCALL_EXIT    47,1-4    61%
48     svc 0                   48,1-4    64%
49
50 .data
51 message:                    50,1      69%
52     .ascii "Hello World!\n"  51,1      71%
53 message_len= .-message      52,1-4    74%
54 @                           53,1      76%
55 @ . evaluates to the current address: the address where th55,1  54,1      79%
56 @ or data is to be assembled. In this case the address is 43,1-4   55,1      82%
                                         56,1      84%

```

Disassemblamento della sezione .data:

```

000200a4 <message>:
200a4:    6c6c6548        cfstr64vs      mvdx6, [ip], #-288      ; 0xffff
fee0
200a8:    6f57206f        svcvs    0x0057206f
200ac:    21646c72        smccs    18114    ; 0x46c2
200b0:    L'indirizzo 0x000200b0 è fuori dai limiti.

```

disassemblamento di una parte di codice

```

00010074 <_start>:
10074:    e3a00001        mov      r0, #1
10078:    e59f101c        ldr      r1, [pc, #28]    ; 1009c <_start+0x28>
1007c:    e59f201c        ldr      r2, [pc, #28]    ; 100a0 <_start+0x2c>
10080:    e1a04002        mov      r4, r2
10084:    e3a07004        mov      r7, #4
10088:    ef000000        svc      0x00000000
1008c:    e1500004        cmp      r0, r4
10090:    03a00000        moveq   r0, #0
10094:    e3a07001        mov      r7, #1
10098:    ef000000        svc      0x00000000
1009c:    000200a4        andeq   r0, r2, r4, lsr #1
100a0:    0000000d        andeq   r0, r0, sp

```

Disassemblaggio della parte "start" di un altro codice. In r1 ed r2 vengono caricati i dati presenti agli indirizzi 1009c e 100a0. Questo codice utilizza la SYSCALL_WRITE.

```

@ Example 5
@
@ compile with:
@
@ cc -g 5.s -o 5
@
@

.text
.global main

main:
    push {ip, lr}
    @ push {ip, lr} is a pseudo-instruction that stores
    @ the values of ip and lr on the stack in a precise
    @ order that simply follows that of register numbers.
    @ This way there is no need to keep track of pushes
    @ as with other ISAs.
    @ The value of ip is not important, but it is stored
    @ too because stack must be kept 8-byte aligned when
    @ calling other functions

    ldr    r0, =#4096

```

```
    ldr    r1, =#6000
    mul    a2, r0, r1

    ldr    a1, =format
    bl     printf

    pop {ip, lr} @ restoring registers is as simple as storing them
    bx    lr

.data
format:
    .asciz "value: %u\n"
```

Non abbiamo necessità di utilizzare le syscall poichè stiamo utilizzando stdlib ed è fornito da default dal compilatore c. Otteniamo lo stesso risultato ma in maniera più complicata, facendo il paragone tra C e ARM;

Questo codice è un programma in C, però scritto in assembly e questo aggiunge complessità: per questo vengono proposti linguaggi di alto livello

Lezione 14 - 18/11/2021

Altri esempi di codice ARM

ESEMPIO 6 (scrivi un codice che stampa da 10 a 0)

```
@ Example 6
@
@ compile with:
@
@ cc -g 6.s -o 6
@
@

.text
.global main

main:
    push {ip, lr}
@
@      Equivalent C code:
@
@      int i;
@
@      i=10;
@      do{
@          printf("%i\n", i);
@          --i;
@      }while(i>=0);
```

```

@

    ldr v1, =#10          @ i=10 -> v1=10

loop:
    ldr a1, =format      @
    mov a2, v1            @     printf(format, v1)
    bl  printf            @

    subs v1, v1, #1        @     --i; -> v1=v1-1 CPSR.N=(v1<0)
    bpl loop              @     }while(!CPSR.N);

    pop {ip, lr}
    bx      lr

.data
format:
    .asciz "%i\n"

```

prima di utilizzare subroutine, dobbiamo preservare il valore del link register effettuando un push {ip,lr} -> ip è il registro di inter-procedura ed lr ovviamente il link register. Nel momento in cui viene eseguita l'istruzione bl main, il primo indirizzo della label main viene memorizzato in cima allo stack (e ovviamente il valore corrente del LR è quello puntato dallo SP).

Dopo aver eseguito completamente il loop ed "esaurita" la condizione, il pop mi permette di "svuotare" lo stack e lo stack pointer ritorna in cima allo stack (nello stato precedente all'esecuzione di push); questo meccanismo è utile per scrivere sottochiamate a routine (**nested call**)

I compilatori sono "permissivi", c'è elasticità tra le piattaforme

ESEMPIO 7 (scrivi un programma che stampi i*400000)

```

@ Example 7
@
@ compile with:
@
@ cc -g 7.s -o 7
@
@

.text
.global main

main:
    push {ip, lr}

@      Equivalent C code
@
@      long long unsigned a; /* 64-bit value */
@      unsigned c; /* 32-bit value */
@      unsigned i; /* 32-bit value */
@
@      a=0;
@      c=4000000000
@      i=10;

@      do{
@          a+=i*c;
@          --i;
@      }while(i+1>0);
@

        ldr    v1, =#10          @ i=10;
        ldr    v2, =#4000000000  @ c=4000000000;
        mov    v3, #0             @ a=0; (lowest 32 bits)
        mov    v4, #0             @ a=0; (highest 32 bits)

loop:
        umlal v3,v4,v1,v2      @ a+=i*c ->

        ldr    a1, =format1     @
        mov    a2, v1             @ printf(format1, i, c);
        mov    a3, v2             @
        bl    printf             @

        subs   v1, v1, #1         @ --i -> v1=v1-1, CPSR.N=(v1<0)
        bpl   loop               @ while(!CPSR.N)

        ldr    a1, =format2     @
        mov    a3, v3             @ printf(format2, a);

```

```

        mov a3, v3          @ printf(format2, a);
        mov a4, v4          @ the second arg to printf is 64-bit wide
        bl  printf          @ a couple of contiguous register is needed
                           @ a1 is used for the first arg so the first
                           @ (and last) usable couple of registers to
                           @ hold a 64-bit arg to printf is
                           @ a3,a4
                           @ In this case any other argument would be stored
                           @ on the stack (try adding a %x to format2 and see
                           @ what happens...

        pop {ip, lr}
        bx    lr

.data
format1:
    .asciz "%u*u\n"
format2:
    .asciz "=%llu\n"

```

è abbastanza facile, molto simile all'esempio 6 ma con l'utilizzo di UMLAL. Quando vengono utilizzati valori più grandi della word, ad alto livello il compilatore gestisce questi valori. A basso livello possiamo passare un valore da 64 bit utilizzando due coppie di registri consecutivi (è previsto dalle calling convention, il primo registro deve essere "pari" quindi r2-r3 ovvero a3,a4)

ESEMPIO 8 (implementa una funzione che copi l'array in un altro)

```

pi@raspberrypi: 08$ cat 8_main.c
/* Example 8
 *
 * compile with:
 *
 * cc -g 8.s 8_main.c -o 8
 */
#include <stdio.h>
#include <stdlib.h>

/* Protos */

void
array_copy(int *src, int *dst, size_t size);

void
array_print(int *array, size_t size);

/* Functions */

void
array_print(int *array, size_t size){

```

```

size_t i;

for(i=0; i<size; ++i){
    printf("%i ", array[i]);
}
printf("\n");

int
main(void){
    int a[]={1,2,3};
    int b[3];
    size_t size;

    size=sizeof(a)/sizeof(int);
    printf("Source array:\n");
    array_print(a, size);
    printf("Destination array before
copy:\n");
    array_print(b, size);
    array_copy(a, b, size);
    printf("Destination array after
copy:\n");
    array_print(b, size);
    return 0;
}

```

```

pi@raspberrypi: 08$ cat 8.s
@ Example 8
@
@ compile with:
@
@ cc -g 8.s 8_main.c -o 8
@
@

.text
.global array_copy

array_copy:

loop:   ldr a4, [a1], #4
        str a4, [a2], #4
        subs a3, a3, #1
        bne loop

        bx lr

```

il codice mostrato prevede l'implementazione di due funzioni: array print, array copy. La struttura del codice è molto simile a quelli precedenti. Lo stack pointer dev'essere allineato a "double word" e quindi necessitiamo di 8 byte

N.B i compilatori possono ottimizzare le chiamate a funzione, come ad esempio nel caso di `printf("stringa!\n")` -> in questo caso verrà chiamata la funzione `puts()` la quale permette di stampare con maggiore efficienza la stringa (probabilmente la tratta come immediato).

Dispositivi "targets"

Dopo aver studiato l'ISA ,i meccanismi di esecuzione,fondamenti di sistemi operativi e funzionamento di software di basso livello, il focus si sposta nel target: come scriviamo il codice su tali macchine?

Le nostre macchine possono essere descritte gerarchicamente come:

- una scheda (scheda madre) la quale contiene tutte le parti di sistema ed includono:
 - System of a Chip (SoC)
 - Sistemi on-board
 - Memoria
 - Storage
 - periferiche I/O
 - Networking wireless e wired
 - Connettori I/O9
 - Elettronica di potenza
 - circuiteria di programmazione "in-system" e debugging

Lezione 15 - 23/11/2021

Dispositivi target (continua)

Come sviluppiamo per le nostre macchine target? Abbiamo 3 modelli

- Compilazione nativa
- Cross-compilation
 - utilizzo di una toolchain software (GCC ad esempio) per produrre codice "oggetto"
 - l'eseguibile è caricato nella macchina target: il sistema operativo è opzionale
 - possono essere utilizzati emulatori per far eseguire codice di una macchina in un'altra con ISA differente
- Programmazione interattiva
 - Un interprete è eseguito nella macchina target in maniera tale da poter "testare" il codice, rendendo flessibile la programmazione dinamica
 - in base alla complessità dell'interprete, questo modello potrebbe non essere adatto a macchine con risorse limitate
 - Interpreti Forth possono essere implementati facilmente per macchine con risorse limitate

Nei prodotti target si possono trovare sicuramente applicazioni ad alto-livello "astratte" dall'hardware tramite del layer software come il sistema operativo: quest'ultimo agisce come "scudo" rispetto all'hardware e agisce da intermediario tra quest'applicazione e l'hardware, permettendo solamente determinate operazioni. Possiamo trovare anche HALs (Hardware Abstraction Layer) e runtime. La macchina target può essere anche programmata "bare-metal" con codice di basso livello, strettamente collegato all'hardware.

Qualsiasi esso sia il modello di sviluppo, il codice eseguibile è scritto per essere eseguito al momento giusto

1. o all'avvio nel caso dell'OS e del codice bare metal
2. o quando è richiesto dall'utente, sempre in spazio OS

In entrambi i casi, è necessario che venga eseguito del codice all'inizio, anche prima dell'OS: il target esegue del codice di inizializzazione delle componenti in una locazione fissa della memoria. Questo processo è chiamato **bootstrapping**. Queste memorie "permanenti" da cui viene preso il codice utilizzano tipicamente Flash ROM oppure possono essere incluse nell'architettura di sistema e "mappate" all'indirizzo di partenza della CPU (es: UEFI, STM32F446). E' utilizzato un meccanismo hardware per copiare codice eseguibile dall'esterno allo storage (flashing).

Se voglio registrare dei valori di misurazione dei sensori nel tempo, posso utilizzare questo meccanismo

Tipicamente i SoC sono provvisti di un bootloader pre-installato che semplifica la programmazione del target tramite varie periferiche come USB, network, Over-the-air (OTA) aggiornamenti.

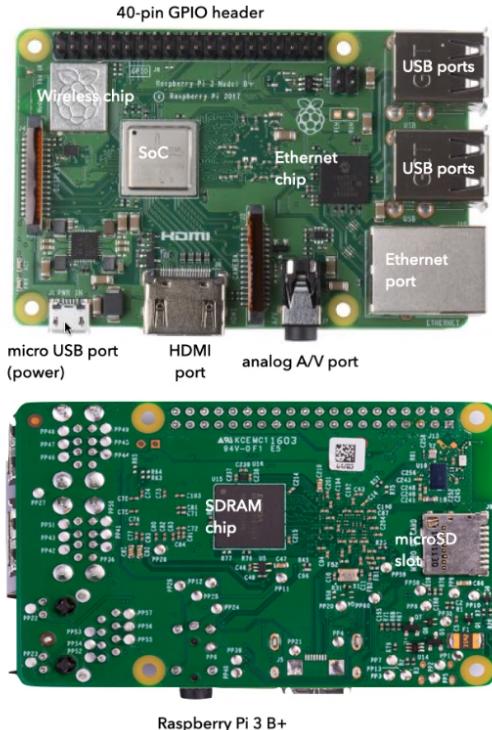
- possono essere implementate modalità di recovery basate su tecniche di bootstrapping utilizzabili in caso di danneggiamento software del dispositivo

Anche i bootloader possono essere aggiornati dal codice eseguito nella macchina target; se si utilizza uno storage separato per il bootloader se un problema avviene durante l'aggiornamento la procedura può essere ripetuta. Cosa può succedere se un problema avviene durante un upgrade del bootloader? la macchina diventa un "mattone" (bricked). Alcune interfacce hardware supportano il "programming" e bootstrapping

- JTAG(Joint Test Action Group): questo è uno standard
- SWD (Serial Wire Debug)

HARDWARE SPECS:

- ▶ SoC: Broadcom BCM2837B0 quad-core A54 (ARMv8) 64-bit @ 1.4GHz
 - ▶ GPU: Broadcom VideoCore IV
- ▶ On-board Components
 - ▶ 1GB LPDDR2 SDRAM
 - ▶ Gigabit Ethernet (via USB channel)
 - ▶ Wireless networking: 2.4GHz and 5GHz 802.11b/g/n/ac wireless LAN, Bluetooth 4.2, Bluetooth Low Energy (BLE)
 - ▶ Storage: microSD
 - ▶ One CPU controllable green LED
- ▶ Ports
 - ▶ HDMI, 3.5 mm analog audio-video jack
 - ▶ 40-pin GPIO header
 - ▶ 4x USB 2.0
 - ▶ Camera Serial Interface (CSI) and Display Serial Interface (DSI)



Raspberry Pi 3 B+

il Raspberry Pi possiede una CPU quad-core a 64 bit e supporta tutte le varianti ISA di ARM

- aarch64
- aarch32
- Thumb

include anche una toolchain per cross-compiling (GNU ARM Embedded Toolchain)

Il bootstrapping è gestito dal VC (BroadCome VideoCore IV) una parte proprietaria del sistema che include una CPU e circuiteria di generazione video

Il VC è in grado di vedere file nella prima partizione (boot partition) della microSD se è formattata in FAT32

All'avvio, prima che la CPU ARM inizi a runnare, il VC effettua il bootstrap

- il VC carica il suo codice di bootloading dal file bootcode.bin nella partizione di boot
- il bootloader controlla per la presenza del file config.txt nella partizione di boot. Se il file esiste, il bootloader esamina il file leggendo i parametri di configurazione; altrimenti, utilizza valori di default
- Il bootloader quindi carica il second-stage del bootloader da un altro file (default: start.elf) e in maniera opzionale, un file linker (fixup.dat). Il file contiene altro codice VC e dati di inizializzazione

Il codice nel second-stage bootloader è quindi eseguito dal VC, che cercherà per un file .img contenente codice eseguibile (il file .img viene copiato esattamente così com'è). I file verranno cercati nel seguente ordine:

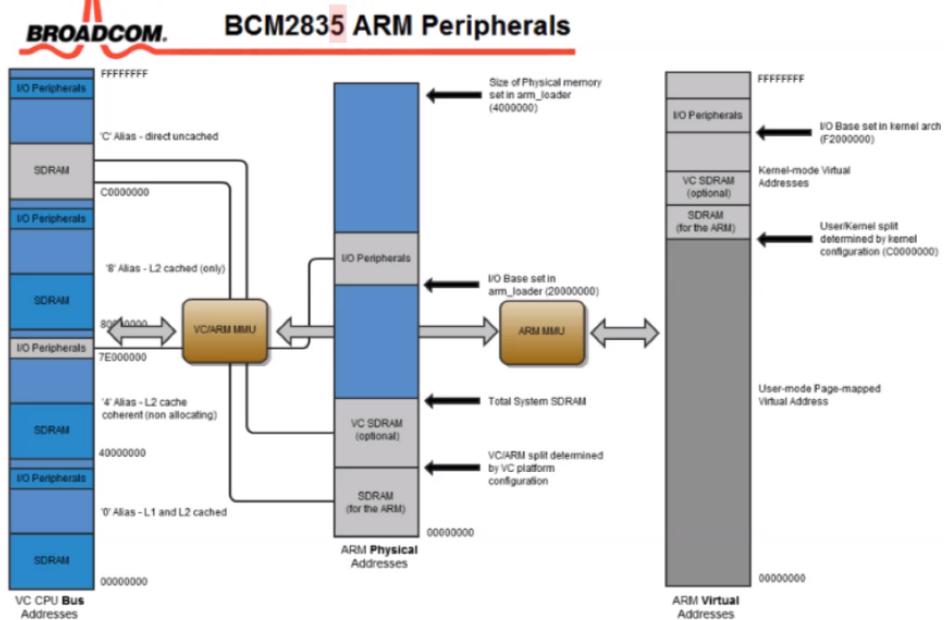
- kernel8.img
- kernel71.img
- kernel7.img
- kernel.img

kernel8.img deve contenere codice a 64-bit, mentre gli altri file devono contenere codice a 32-bit. L'indirizzo di caricamento può essere specificato tramite il parametro KERNEL_ADDRESS nel file config.txt. Gli indirizzi di caricamento di default sono solitamente 0x8000 (32 bit) oppure 0x80000 (64 bit). Dopo aver caricato l'immagine, il VC inizializza la CPU ARM (il primo core) e lo fa "saltare"

all'indirizzo di caricamento: gli altri core eseguiranno dei loop WFE(wait for event). Se il file kernel8 è trovato, allora ARM eseguirà istruzioni a 64bit

Memory Mapping delle periferiche

MEMORY MAPPING OF PERIPHERALS



06 February 2012 Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB4 0WW
© 2012 Broadcom Corporation. All rights reserved

Page 5

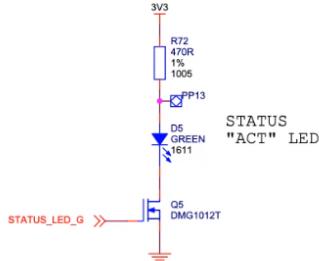
l'MMU "traduce" gli indirizzi da VC ad ARM nella maniera corretta ed è controllata dal VC. Nella seconda fase abbiamo l'ARM MMU che "traduce" l'indirizzamento richiesto dal sistema operativo linux (ARM Virtual Address): se siamo in modalità bare-metal, la seconda MMU è spenta. Vengono mappati altri spazi come quello riservato alle periferiche I/O, il quale è uno spazio "variabile" e, ovviamente l'SDRAM (si parte dall'indirizzo 0x00000000). Ogni processo di linux "vede" la memoria così come rappresentata nello spazio virtuale

Programmazione Bare metal

- ▶ Our target code is obtained through the following steps:
 - ▶ Code is written into a source (.s) file
 - ▶ vi test.s
 - ▶ The assembly file is assembled into an Executable and Linkable Format (ELF) object file (.o)
 - ▶ arm-none-eabi-as test.s -o test.o
 - ▶ The object file is processed by the linker to resolve addresses for a fixed memory model producing a finalized ELF file
 - ▶ arm-none-eabi-ld -T kernel17.ld test.o -o test.elf
 - ▶ The objdump tool can be used to obtain the finalized ML and assembly code in a listing file (.list)
 - ▶ arm-none-eabi-objdump -D test.o > test.list
 - ▶ The objectcopy tool is used to extract only the ML code from the finalized ELF file in binary format
 - ▶ arm-none-eabi-objcopy test.elf -O binary test.bin
 - ▶ The binary file is renamed to follow the bootstrap conventions
 - ▶ mv test.bin kernel17.img
- ▶ The image file can then be simply copied onto the microSD to be executed at startup

BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 01

► Let's light up the green LED



```

1 /* Example bm-01: LED on
2 *
3 * Target: Raspberry Pi 2/3
4 *
5 */
6
7 /* Pi 2/3
8 * GPIO register addresses
9 */
10 GPFSEL2=0x3F200008
11 GPSET0 =0x3F20001C
12
13 .global _start
14
15 _start:
16     ldr r0,=GPFSEL2
17     ldr r0,[r0]
18     bic r1, r0, #0x38000000
19     orr r1, r1, #0x08000000
20     ldr r0,=GPFSEL2
21     str r1, [r0]
22
23     ldr r0,=GPSET0
24     mov r1, #0x20000000 /* GPIO pin 29 on (LED ON) */
25     str r1, [r0]
26
27 1:    b 1b
28

```

viene eseguito un loop infinito in maniera tale che il led sia sempre attivo

in R0 metto l'indirizzo che mi serve, il load mi serve per modificarlo ed applico una "mascheratura" tramite BIC (tra r0 ed 0x38000000) ed ottengo 0x38000000, così riesco a controllare il PIN 9 del GPFSEL2; effettuo l'or con se stesso.

Due registri General Purpose I/O (GPIO) sono usati per accendere il LED verde: GPFSEL2 e GPSET0

- GPFSEL2 all'indirizzo 0x3F200008 contiene 3 bit (29-27) che controllano le funzionalità del pin GPIO 29, fisicamente connesso al LED
- In questo caso, la funzione di output deve essere selezionata impostando i bit 29-27 come 0b001

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL29	FSEL29 - Function Select 29 000 = GPIO Pin 29 is an input 001 = GPIO Pin 29 is an output	R/W	0
		100 = GPIO Pin 29 takes alternate function 0 101 = GPIO Pin 29 takes alternate function 1 110 = GPIO Pin 29 takes alternate function 2 111 = GPIO Pin 29 takes alternate function 3 011 = GPIO Pin 29 takes alternate function 4 010 = GPIO Pin 29 takes alternate function 5		
26-24	FSEL28	FSEL28 - Function Select 28	R/W	0
23-21	FSEL27	FSEL27 - Function Select 27	R/W	0
20-18	FSEL26	FSEL26 - Function Select 26	R/W	0
17-15	FSEL25	FSEL25 - Function Select 25	R/W	0
14-12	FSEL24	FSEL24 - Function Select 24	R/W	0
11-9	FSEL23	FSEL23 - Function Select 23	R/W	0
8-6	FSEL22	FSEL22 - Function Select 22	R/W	0
5-3	FSEL21	FSEL21 - Function Select 21	R/W	0
2-0	FSEL20	FSEL20 - Function Select 20	R/W	0

nel caso del GPIO, possiamo cambiare la direzione di ogni "linea" affinchè possano essere utilizzati sia come input digitale che come output digitale. Possiamo sfruttare la stessa linea fisica per scegliere fino a 8 "possibilità" (funzioni alternative); in particolare la funzione che c'interessa è 001

Bare Metal programming - esempi

```

▶ The disassembly obtained with
    arm-none-eabi-objdump -d led_on.elf > led_on.elf.list

▶ shows that the information about the fixed loading address (0x00008000) is contained in the ELF file and that the two register addresses (0x3f200008, 0x3f20001c) were transformed by the assembler into two data words (.word directives) at the end of code to be read by PC-relative load instructions. Even if the information about the loading address is not copied to the binary image file (arm-none-eabi-objcopy led_on.elf -O binary led_on.bin), this dump is useful to see how the code will look like when loaded into memory

led_on.elf:      file format elf32-littlearm

Disassembly of section .text:
00008000 <_start>:
 8000:   e59f0020      ldr    r0, [pc, #32] ; 8028 <_start+0x28>
 8004:   e5900000      ldr    r0, [r0]
 8008:   e3c0130e      bic    r1, r0, #939524096 ; 0x38000000
 800c:   e3811302      orr    r1, r1, #134217728 ; 0x80000000
 8010:   e59f0010      ldr    r0, [pc, #16] ; 8028 <_start+0x28>
 8014:   e5801000      str    r1, [r0]
 8018:   e59f000c      ldr    r0, [pc, #12] ; 802c <_start+0x2c>
 801c:   e3a01202      mov    r1, #536870912 ; 0x20000000
 8020:   e5801000      str    r1, [r0]
 8024:   eaaffffe      b     8024 <_start+0x24>
 8028:   3f200008      .word 0x3f200008
 802c:   3f20001c      .word 0x3f20001c

```

In the ARM ISA the offset of a PC-relative address is given with respect to the second instruction following the instruction using it, e.g:

- the first constant (0x3f200008) is stored in the word (32-bits) at address 0x00008028
- the first ldr loads its value from the address $pc+32 = pc+0x20 = 0x00008008 + 0x20 = 0x00008028$
 - 0x00008008 is the address of the second instruction after the ldr (bic)
- the second ldr accesses the same value using the PC-relative offset 16=0x10 with the address 0x00008018

.word è la position label mod. e' posta alla fine ma comunque si può spostare preservandone il suo funzionamento

```

arm-none-eabi-as led_on.s -o led_on.o
arm-none-eabi-ld -T kernel7.ld led_on.o -o led_on.elf
arm-none-eabi-objdump -d led_on.elf > led_on.elf.list
arm-none-eabi-objcopy led_on.elf -O binary led_on.bin
arm-none-eabi-objdump -d led_on.elf > led_on.elf.list
mv led_on.bin kernel7.img
rm led_on.o led_on.elf

```

è conveniente separare assembler e compilatore durante l'esecuzione in questo caso. L'assembler produce qualcosa di immediato ed è "collegato" con altri oggetti simili; accanto questo file possiamo trovare informazioni per fare del "link" con altri codici. Il codice "lavorerà" in un modulo dove la prima istruzione è caricata con indirizzo 8000

in genere all'assembler non diciamo dove deve iniziare. Tutte queste scelte sono assegnate ad altre funzionalità o componenti.

Quando programmiamo a questo livello, dobbiamo specificare l'indirizzo di inizio con "b 8024"

la seconda riga è pensata per essere specificata ad un indirizzo specifico

```

00008000 <_start>:
 0:   e59f0020      ldr    r0, [pc, #32] ; 28 <_start+0x28>
 4:   e5900000      ldr    r0, [r0]
 8:   e3c0130e      bic    r1, r0, #939524096 ; 0x38000000
 c:   e3811302      orr    r1, r1, #134217728 ; 0x80000000
10:   e59f0010      ldr    r0, [pc, #16] ; 28 <_start+0x28>
14:   e5801000      str    r1, [r0]
18:   e59f000c      ldr    r0, [pc, #12] ; 2c <_start+0x2c>
1c:   e3a01202      mov    r1, #536870912 ; 0x20000000
20:   e5801000      str    r1, [r0]
24:   eaaffffe      b     24 <_start+0x24>
28:   3f200008      .word 0x3f200008
2c:   3f20001c      .word 0x3f20001c

```

in questo caso il nostro riferimento fisso è 0. Questo è la lista dei comandi del file .o.

INSERISCI IMMAGINE

```
kernel7.img:      file format binary

Disassembly of section .data:

00000000 <.data>:
 0:   e59f0020      ldr    r0, [pc, #32] ; 0x28
 4:   e5900000      ldr    r0, [r0]
 8:   e3c0130e      bic    r1, r0, #939524096 ; 0x38000000
 c:   e3811302      orr    r1, r1, #134217728 ; 0x8000000
10:   e59f0010      ldr    r0, [pc, #16] ; 0x28
14:   e5801000      str    r1, [r0]
18:   e59f000c      ldr    r0, [pc, #12] ; 0x2c
1c:   e3a01202      mov    r1, #536870912 ; 0x20000000
20:   e5801000      str    r1, [r0]
24:   eafffffe      b     0x24
28:   3f200008      svccc 0x00200008
2c:   3f20001c      svccc 0x0020001c
```

questo è il codice disassemblato di kernel7.img: non si può effettuare il dump

Il codice che abbiamo visto nelle precedenti lezioni può essere rivisto e scritto con delle subroutine

```

1 /* Example bm-01: LED on
2 *
3 * Target: Raspberry Pi 2/3
4 *
5 */
6
7 /* Pi 2/3
8 * GPIO register addresses
9 */
10 GPFSEL2=0x3F200008
11 GPSET0 =0x3F20001C
12
13 .global _start
14
15 _start:
16     mov sp, #0x80000
17     push {ip, lr}
18     bl led_pin_enable
19
20     ldr r0,=GPSET0
21     mov r1, #0x20000000 /* GPIO pin 29 on (LED ON) */
22     str r1, [r0]
23
24     pop {ip, lr}
25 1:   b 1b
26
27
28 led_pin_enable:
29     ldr r0,=GPFSEL2
30     ldr r0, [r0]
31     bic r1, r0, #0x38000000
32     orr r1, r1, #0x08000000
33     ldr r0,=GPFSEL2
34     str r1, [r0]
35     bx lr

```

in una struttura del genere, si pone l'eventualità di chiamate foglia; per fare ciò bisogna utilizzare lo stack. Quando si programma in C oppure anche in Assembly, abbiamo la maggior parte delle "cose" contenute in questo stack. Qual è la strategia adottata per scrivere questo codice?

Settiamo il valore dello stack pointer ad 0x80000 (di solito non si inserisce un valore assoluto nello stack pointer) e sfrutto lo stack

```

1 /* Example bm-01: LED on
2 *
3 * Target: Raspberry Pi 2/3
4 *
5 */
6
7 /* Pi 2/3
8 * GPIO register addresses
9 */
10 GPFSEL2=0x3F200008
11 GPSET0 =0x3F20001C
12
13 .global _start
14 _start:
15     mov sp, #0x80000
16     push {ip, lr}
17     bl led_pin_enable
18     bl led_on
19     pop {ip, lr}
20 1: b 1b
21
22 /* led_pin_enable
23 * args: none
24 */
25 led_pin_enable:
26     ldr r0,=GPFSEL2
27     ldr r0,[r0]
28     bic r1, r0, #0x38000000
29     orr r1, r1, #0x08000000
30     ldr r0,=GPFSEL2
31     str r1, [r0]
32     bx lr
33
34 /* led_on
35 *
36 * args: none
37 */
38 led_on:
39     ldr r0,=GPSET0
40     mov r1, #0x20000000 /* GPIO pin 29 on (LED ON) */
41     str r1, [r0]
42     bx lr

```

questo codice è molto più leggibile rispetto a quelli precedenti: viene, come al solito, inizializzato lo stack ed eseguiti 2 bl

Come facciamo a scrivere del codice che ci permette di eseguire un lampeggiamento? Dobbiamo gestire il tempo e per questo dobbiamo avere chiari due concetti

1. Scaling del tempo
2. tempo di esecuzione della macchina

BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 02

▶ Blinking LED

- ▶ The pin which the LED is connected to is enabled as output by the `led_pin_enable` subroutine
- ▶ Then, in a never ending loop,
 - ▶ the LED is switched on by calling the `led_on` subroutine
 - ▶ the delay subroutine makes the CPU wait for a time proportional to the value passed in `r0`
 - ▶ the LED is switched off by calling the `led_off` subroutine
 - ▶ the delay subroutine is called again with a smaller time value in `r0`

```
1 /* Example bm02: Blinking LED
2 *
3 * Target: Raspberry Pi 2/3
4 *
5 */
6
7 /* Pi 2/3
8 * GPIO register addresses
9 */
10 GPFSEL2=0x3F200008
11 GPSET0 =0x3F20001C
12 GPCLR0 =0x3F200028
13
14 .global _start
15
16 _start:
17     mov sp, #0x80000
18     push {ip, lr}
19
20     bl led_pin_enable
21
22 1:   bl led_on
23     mov r0, #0x00400000 /* duration=64*65536 - LED stays longer (8x)
on than off */
24     bl delay
25     bl led_off
26     mov r0, #0x00080000 /* duration=8*65536 */
27     bl delay
28     b 1b    -->
29
30 /* led_pin_enable
31 * args: none
32 */
33 led_pin_enable:
34     ldr r0,=GPFSEL2
35     ldr r0,[r0]
36     bic r1, r0, #0x38000000
37     orr r1, r1, #0x08000000
38     ldr r0,=GPSET2
39     str r1, [r0]
40     bx lr
41
```

ARM_Assembly/4_Bare metal/RPi3Bn/bm_02_blinking_led.s/blinking_led.s

questa volta "b 1b" ricopre una porzione più grossa del codice. Qual è la complicazione di questo codice ? la costruzione della funzione delay. Intanto passiamo il valore di durata (rispettando le calling convention) ad `r0`.

- ▶ The `led_on` and `led_off` subroutines are very similar
- ▶ The same value, `0x20000000`, written to `GPSET0` turns the LED on, written to `GPCLR0` turns it off
 - ▶ This is because the bits that are set (in this case the 29th) in the immediate operand only carry the information on the position of the bits to be set or clear
 - ▶ The real bit write in the corresponding pin-level register bits is performed by the peripheral circuitry
 - ▶ A write to `GPSET0` sets the bits at the indicated positions
 - ▶ A write to `GPCLR0` clears the bits at the indicated positions
 - ▶ All the other pin-level bits are untouched
- ▶ To make the CPU wait for a given time, the delay subroutine simply executes a counting loop until the number of iterations reaches the value in `r0`

```
42 /* led_on
43 *
44 * args: none
45 */
46 led_on:
47     ldr r0,=GPSET0
48     mov r1, #0x20000000 /* GPIO pin 29 on (LED ON) */
49     str r1, [r0]
50
51     bx lr
52
53 /* led_off
54 *
55 * args: none
56 */
57 led_off:
58     ldr r0,=GPCLR0
59     mov r1, #0x20000000 /* GPIO pin 29 off (LED OFF) */
60     str r1, [r0]
61     bx lr
62
63 /*
64 * delay
65 *
66 * args (r0) delay (iterations)
67 */
68 delay:
69     mov r1, #0
70
71 1:   add r1, r1, #1    -->
72     nop
73     cmp r1, a1
74     bne 1b
75
76     bx lr
```

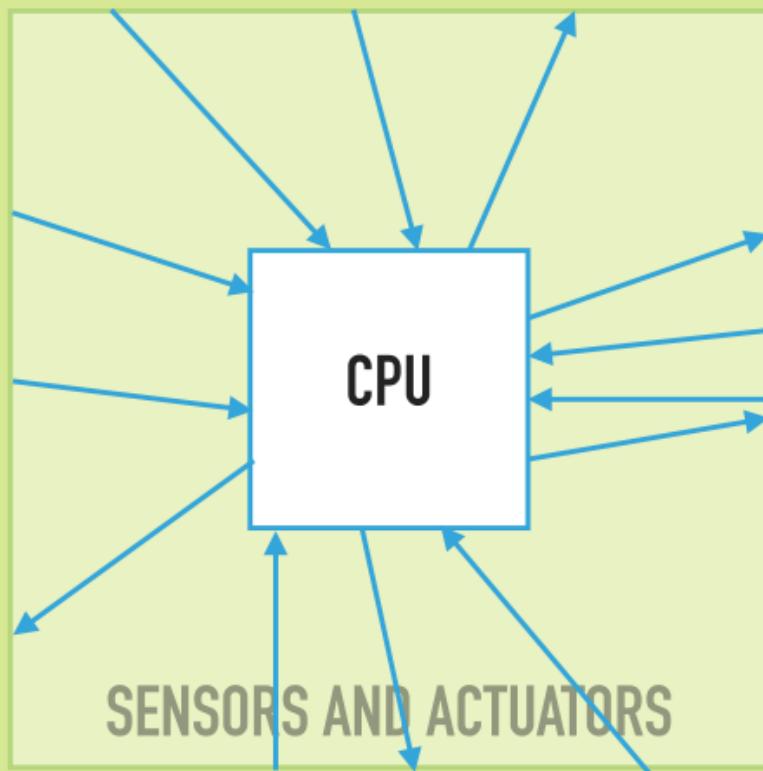
la "nop" nel branch di delay serve per "raddoppiare" il tempo di esecuzione .

Lezione 17 - 30/11/2021

Periferiche & I/O

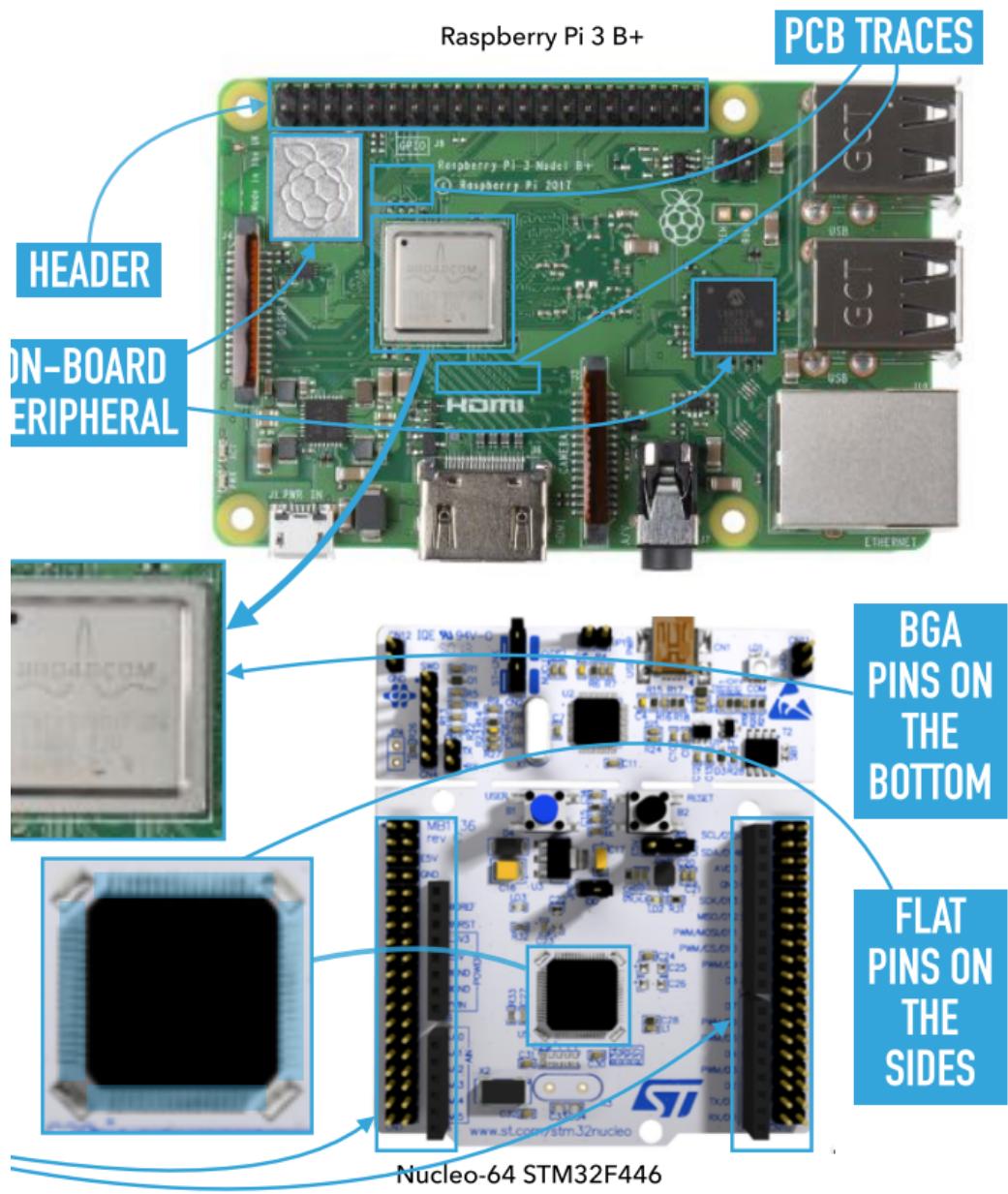
Altri dispositivi hardware, chiamate "periferiche", sono capaci di "scambiare dati" con l'ambiente. Uno dei problemi più importanti quando abbiamo a che fare dati dall'esterno (in relazione "all'attuazione" di certe azioni in base a queste informazioni) è la "traduzione" e il trasporto di questi dati (come segnali, ovviamente). Il dispositivo di computazione deve possedere dei dispositivi in grado di supportare queste operazioni per alleggerire il carico, come i **driver**: dispositivi in grado di trasformare l'output a bassa potenza dalla CPU in segnali adatti alle caratteristiche elettriche richieste dagli attuatori.

PHYSICAL ENVIRONMENT



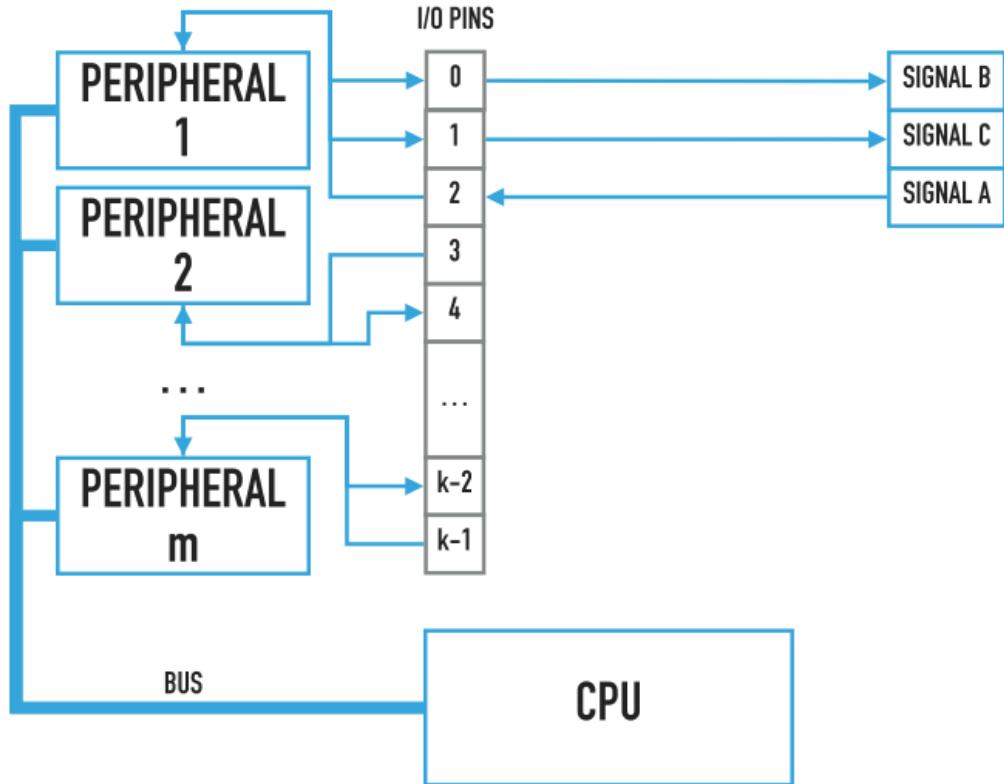
I/O pins

L'MCU e il SoC provvedono l'interfaccia elettrica per i segnali I/O tramite i cosiddetti "pin". Microcontrollori e SoC Embedded hanno abbastanza pin; alcuni sono fisicamente inon individuabili, come il Dual in-line Package o le Surface Mounted Devices (SMD) con "pin flat"



Periferiche I/O

le periferiche possono essere sia semplici che estremamente complesse. Generalmente sono connesse tramite dei bus specializzati (I/O bus) ma non tutti i processori hanno questi "bus" e molto spesso è necessario l'utilizzo di istruzioni ML speciali. Il concetto è che l'utilizzo delle periferiche è garantito tramite la **mappatura della memoria** nello spazio d'indirizzamento. Il comportamento delle periferiche può essere gestito se queste periferiche sono **programmabili**



General - Purpose I/O (GPIO)

• Tied on or off
• Active **enable**
• Lower as much as possible to save power
• Machine reset
• Tied low and must be used
• Control other pins
• (positive, negative)
• Power

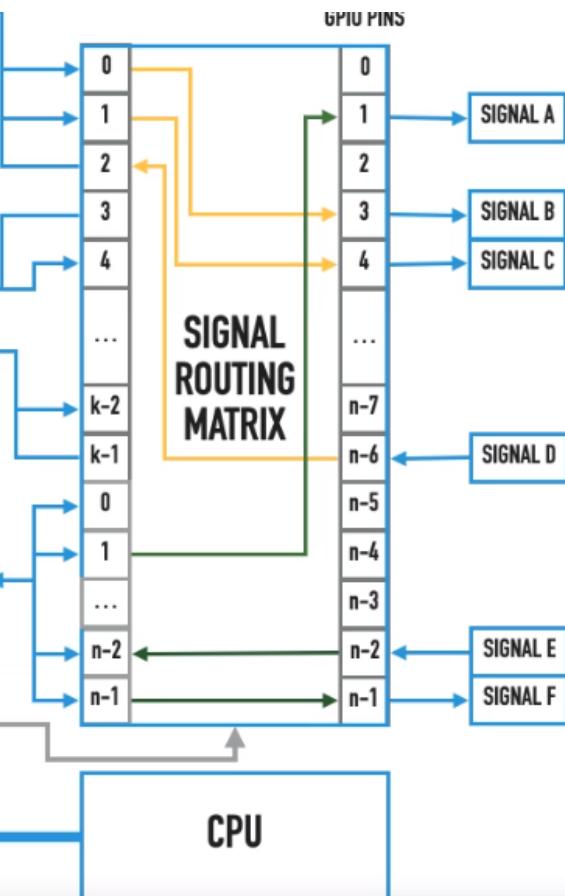
CONTROL REGISTERS

PERIPHERAL
1
2
...
m

PIN LEVEL REGISTERS

FUNCTION SELECT REGS

MEMORY BUS



il GPIO non è nient'altro che un "pin" in grado di switchare ed utilizzare le cosiddette **alternate function** (AF). Nell'immagine le frecce verdi sono segnali di input/output gestiti pin a livello dei registri, mentre le frecce gialle sono i segnali associati alle funzioni periferiche. Spesso le periferiche possono essere accese o spente, settando o resettando i corrispondenti **enable bits**

(EN) in un registro di controllo per ridurre il consumo energetico. IL GPIO inoltre, può "controllare" ,al posto della CPU se le periferiche sono accese.

Peripheral			Functions									
pin #	FSEL bits	FUNC	0	1	2	3	4	5	...	m	k-2	k-1
0	00	INP										
1	00	INP										
2	00	INP										
3	10	AF0	X									
4	11	AF1		X								
...
n-7	00	INP										
n-6	10	AF0			X							
n-5	00	INP										
n-4	00	INP				X						
n-3	00	INP					X					
n-2	00	INP						X				
n-1	01	OUT										

Il control register riesce a gestire i registri di FSEL. Gli interrupt, prima di passare dalla CPU, passano dal GPIO che saprà come gestire opportunamente (E.S: la CPU interrompe l'esecuzione del codice corrente, eseguirà un'altra indicata dal GPIO, e alla fine dell'evento, riprenderà l'esecuzione del codice in precedenza interrotto). I segnali di ogni tipo vengono indirizzati tramite la Signal Matrix Routing: una matrice di $(n-1) \times (n-1)$. Per ogni pin fisico possiamo selezionare una funzione: non possiamo selezionare ognuna perché dobbiamo mantenere l'hardware "meno appesantito" possibile. La codifica FSEL di 00 è INput, per 01 è OUTput, e le due rimanenti 10 e 11 sono rispettivamente AF0 e AF1.

6.2 Alternative Function Assignments

Every GPIO pin can carry an alternate function. Up to 6 alternate functions are available but not every pin has that many alternate functions. The table below gives a quick overview.

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			
GPIO4	High	GPCLK0	SA1	<reserved>			ARM_TDI
GPIO41	Low	PWM1	SD5	<reserved>	<reserved>	SPI2_MOSI	RXD1
GPIO42	Low	GPCLK1	SD6	<reserved>	<reserved>	SP12_SCLK	RTS1
GPIO43	Low	GPCLK2	SD7	<reserved>	<reserved>	SPI2_CE0_N	CTS1
GPIO44	-	GPCLK1	SDA0	SDA1	<reserved>	SPI2_CE1_N	
GPIO45	-	PWM1	SCL0	SCL1	<reserved>	SPI2_CE2_N	
GPIO46	High	<Internal>					
GPIO53	High	<Internal>					

tabella di assegnazione delle funzioni alternative. Il SoC utilizzato nel raspberry ha 8 configurazioni di utilizzo: Input,Output,Alt0,Alt1,Alt2,Alt3,Alt4,Alt5 e non tutte le alt sono connesse ad una periferica

A volte, abbiamo bisogno di più copie dello stesso hardware che controlla degli oggetto: le funzionalità potrebbero servire "più volte". Usare troppi bit non è consigliabile in quanto si evita la mutua esclusione.

Timer

La periferica possiede 4 registri a 32 bit e un singolo canale a 64 bit come "counter". Il timer è fondamentale perchè ci permette di scandire le esecuzioni. Ogni canale ha un compare register in output

- ▶ Peripheral of many different kinds can be included in SoCs
- ▶ All provide a set of registers located each one at a fixed offset from a base address
- ▶ The memory mapping circuitry, either a simple decoding logic or an MMU, defines the base address
 - ▶ Sometimes the mapping circuitry allows for changing the base address
- ▶ The System Timer of the Pi 3 B+ has 5 32-bit registers and one 64-bit register
- ▶ The 64-bit register, exposed to the memory bus as two consequent 32-bit memory words, provides a value counting elapsed μ s since system started

The System Timer peripheral provides four 32-bit timer channels and a single 64-bit free running counter. Each channel has an output compare register, which is compared against the 32 least significant bits of the free running counter values. When the two values match, the system timer peripheral generates a signal to indicate a match for the appropriate channel. The match signal is fed into the interrupt controller. The interrupt service routine then reads the output compare register and adds the appropriate offset for the next timer tick. The free running counter is driven by the timer clock and stopped whenever the processor is stopped in debug mode.

ST Address Map			
Address Offset	Register Name	Description	Size
0x0	CS	System Timer Control/Status	32
0x4	CLO	System Timer Counter Lower 32 bits	32
0x8	CHI	System Timer Counter Higher 32 bits	32
0xc	CO	System Timer Compare 0	32
0x10	C1	System Timer Compare 1	32
0x14	C2	System Timer Compare 2	32
0x18	C3	System Timer Compare 3	32

LED lampeggianti utilizzando il system timer

- ▶ Blinking LED using the System Timer
- ▶ The address of the word containing the lower 32 bits of the system clock value is defined as the symbol **SYSTIMER_CLO**

```

1 /* Example bm01: Blinking LED
2 *
3 * Target: Raspberry Pi 2/3
4 *
5 */
6
7 /* Pi 2/3
8 * GPIO register addresses
9 */
10 GPSEL2=0x3F200008
11 GPSET0 =0x3F20000C
12 GPCLR0 =0x3F200028
13 SYSTIMER_CLO =0x3F003004
14

```

```

15 .global _start
16 _start:
17     mov sp, #0x8000000
18     push {ip, lr}
19
20     bl led_pin_enable
21
22 1: bl led_on
23     mov r0, #0x00400000 /* duration=64*65536 us = ~ 4s */
24     bl delay_us
25     bl led_off
26     mov r0, #0x00100000 /* duration=16*65536 us = ~ 1s */
27     bl delay_us
28     b 1b
29
30 /* led_pin_enable
31 * args: none
32 */
33 led_pin_enable:
34     ldr r0,=GPSEL2
35     ldr r0,[r0]
36     bic r1, r0, #0x38000000
37     orr r1, r1, #0x08000000
38     ldr r0,=GPSEL2
39     str r1, [r0]
40     bx lr
41
42 /* led_on
43 */
44 * args: none
45 */
46 led_on:
47     ldr r0,=GPSET0
48     mov r1, #0x20000000 /* GPIO pin 29 on (LED ON) */
49     str r1, [r0]
50     bx lr
51

```

In epoca passata, dispositivi hardware utilizzati per riprodurre audio o video avevano il problema dell'esecuzione "troppo veloce/lenta" causando artefatti: solitamente s'inseriva un tasto turbo per rallentare/accelerare l'esecuzione.

```

52 /* led_off
53 *
54 * args: none
55 */
56 led_off:
57     ldr r0,=GPCLR0
58     mov r1, #0x20000000 /* GPIO pin 29 off (LED OFF) */
59     str r1, [r0]
60     bx lr
61
62 /*
63 * delay_us
64 *
65 * args (r0) delay (~microseconds)
66 */
67 delay_us:
68     delay .req r0
69     time_lo .req r2
70     time_hi .req r3
71     elapsed_time_lo .req r2
72     start_time_lo .req r4
73     push {r4, lr}           /* Save r4 along with lr */
74     ldr r1,=SYSTIMER_CLO
75     ldrd time_lo, time_hi, [r1] /* Read current time (64-bit value) */
76     mov start_time_lo, time_lo
77 1: ldrd time_lo, time_hi, [r1] /* Read current time (64-bit value) */
78     sub elapsed_time_lo, time_lo, start_time_lo
79     cmp elapsed_time_lo, delay
80     bts 1b
81     pop {r4, lr}
82     .unreq start_time_lo
83     .unreq elapsed_time_lo
84     .unreq delay
85     .unreq time_hi
86     .unreq time_lo
87     bx lr

```

questa subroutine svolge il compito di calcolare il delay, dato in input il registro r0.

Vengono utilizzati degli pseudonimi per manipolare i registri (delay .req r0 è uno di questi, a seguire tutti gli altri). Viene utilizzato lo stack per operare con i registri giusti.

Successivamente, andiamo a caricare in memoria la direttiva SYSTIMER_CL0 ovvero il primo canale per la comparazione del tempo.

Alla linea 75 viene letto il tempo corrente tramite le due istruzioni sempre alla linea 75 e 76. Subito dopo viene utilizzato una subroutine, che valuterà il tempo corrente finché la differenza tra il tempo trascorso (elapsed_time_lo) e il tempo inserito in output (delay) è 0

Lezione 18 - 02/12/2021

Configurazione GPIO

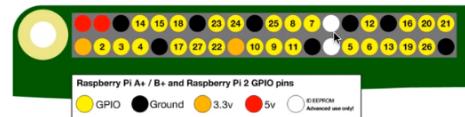
▶ Voltages

- Two 5V pins and two 3V3 pins are present on the board, as well as a number of ground pins (0V), which are not configurable. The remaining pins are all general purpose 3V3 pins, meaning outputs are set to 3V3 and inputs are 3V3-tolerant.



▶ Outputs

- A GPIO pin designated as an output pin can be set to high (3V3) or low (0V).



▶ Inputs

- A GPIO pin designated as an input pin can be read as high (3V3) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.

Sono presenti 2 pin a 5V e 2 pin a 3.3V così come i GND, non configurabili. Molte interfacce, come la USB, forniscono potenza sufficiente anche con tensioni basse. In alcuni casi, quando abbiamo bisogno di convertire segnali elettrici in segnali digitali, necessitiamo dell'hardware che trasformi quei segnali ad una tensione maggiore

▶ PWM (pulse-width modulation)

- Software PWM available on all pins
- Hardware PWM available on GPIO12, GPIO13, GPIO18, GPIO19

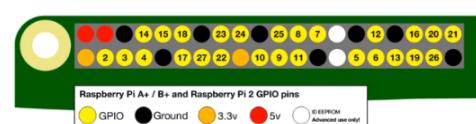


▶ SPI

- SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7)
- SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18); CE1 (GPIO17); CE2 (GPIO16)

▶ I2C

- Data: (GPIO2); Clock (GPIO3)
- EEPROM Data: (GPIO0); EEPROM Clock (GPIO1)
 - A boot ROM can be connected to these pins (white colored in the drawing)



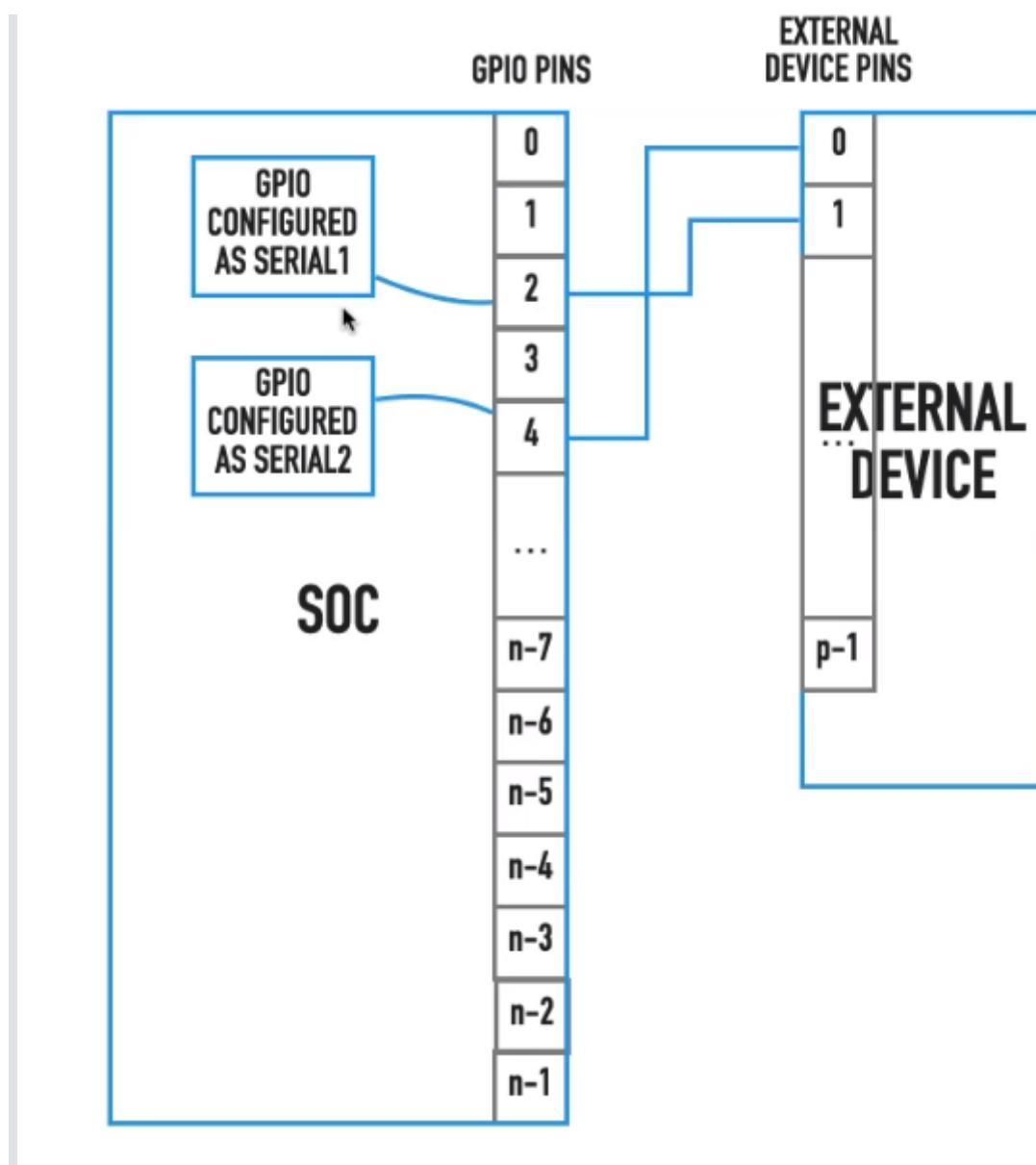
▶ Serial (UART)

- TX (GPIO14); RX (GPIO15)

Ogni pin del GPIO può essere utilizzata per delle applicazioni particolari quali PWM, SPI, I2C o Serial (UART)

Interconnessione Seriale

Nell'interconnessione seriale abbiamo una connessione point-to-point. I dati vengono inviati da un TX (trasmettitore) e ricevuti da un RX (ricevitore)



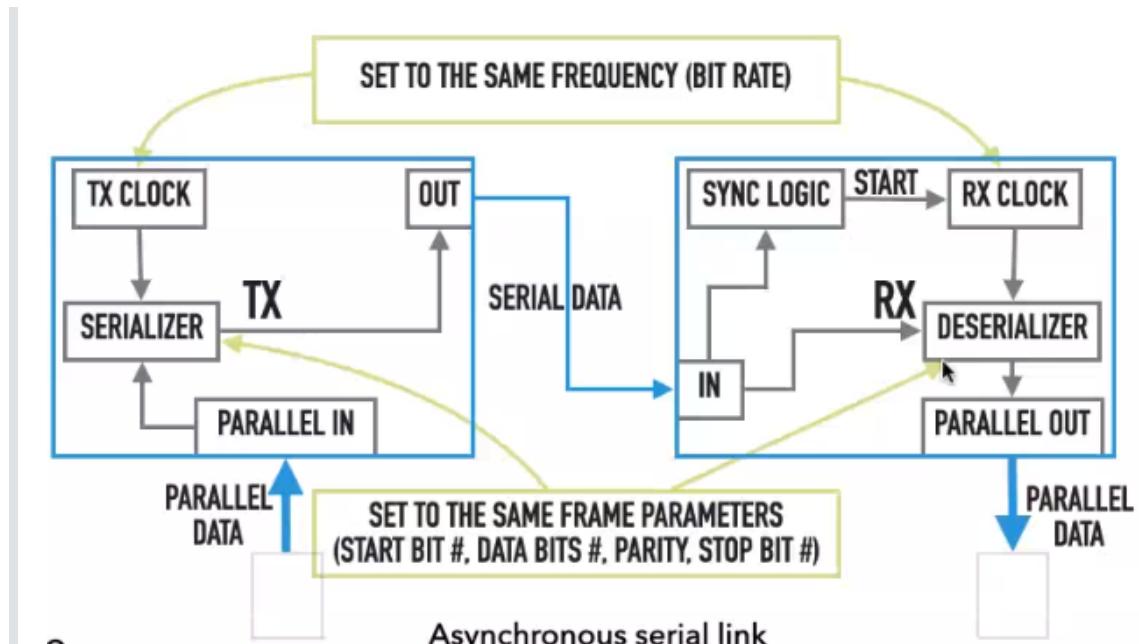
I collegamenti seriali sono molto meno costosi rispetto ai bus paralleli ma più lenti allo stesso clock, ma comunque è sufficiente per svolgere compiti con altri vincoli esterni (e.s: velocità di lettura di una memory card). Un bus parallelo può essere implementato ma richiede molti pin, per questo è meglio utilizzare il collegamento seriale. Le periferiche interne possono "alleggerire" il carico della CPU traducendo le letture parallele dei dati e scrivendole in quelle seriali, assemblando e disassemblando frame, individuando e correggendo errori e controllando il flusso dei dati. I bit trasmessi in frequenza vengono organizzati in sequenze chiamati **Frame**: possono essere presenti anche anche bit di individuazione errore.

Il trasmettitore quindi, organizza questi dati tramite frame, (per individuare l'ordine del bit), li invia e il ricevitore deve utilizzare dei registri, chiamati **shift register** per ottenere "lo scorrimento" dei dati

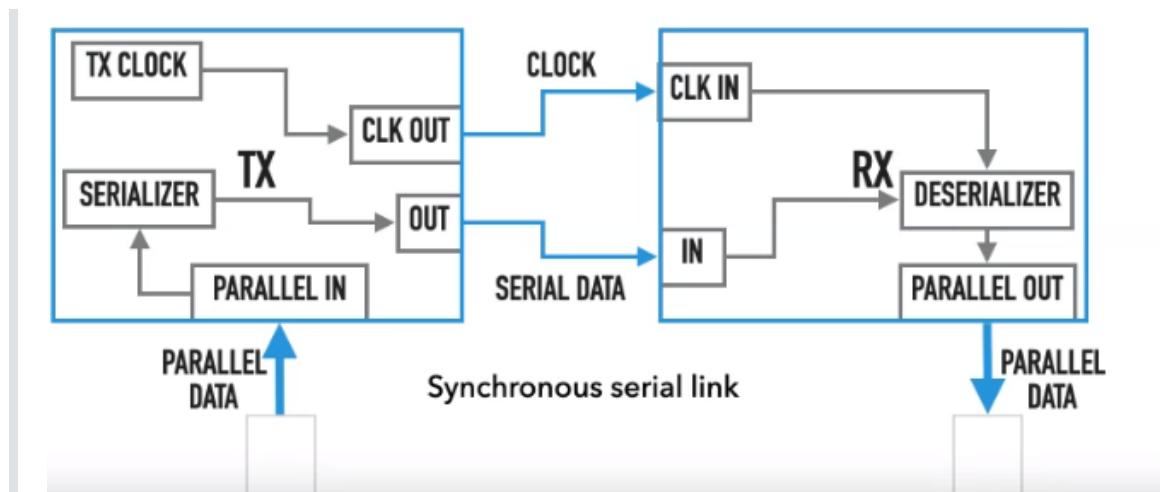
Tipologia di trasporto nella comunicazione seriale

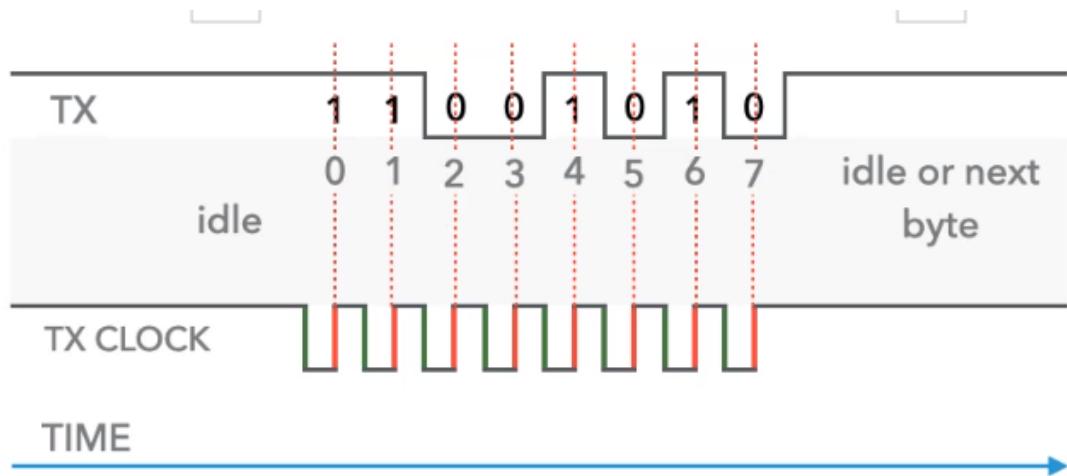
La trasmissione seriale può essere **Sincrona** o **Asincrona**

Asincrona quando i dati possono essere trasmessi in qualunque momento. Il trasmettitore e il ricevitore usano il loro clock, preconfigurato in maniera tale da avere lo stesso tick-rate, per trasmettere e ricevere dati. La dimensione del frame è conosciuta da entrambi trasmettitore e ricevitore ed è necessaria solo la sincronizzazione all'inizio della trasmissione (e dev'essere fornita dai bit trasmessi)



Sincrona quando la trasmissione è sincronizzata da un segnale di **clock** condiviso da trasmettitore e ricevitore. Non c'è la necessità di utilizzare dei bit di sincronia, ma il clock richiede un'altra linea di trasmissione





Transmission of value 0x53, ASCII character 'S', over a synchronous serial link.

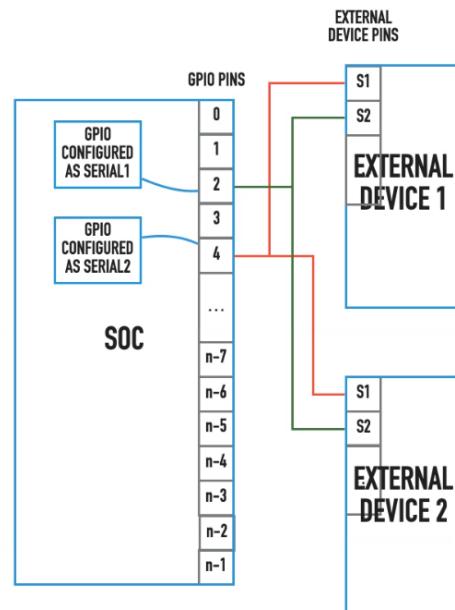
questo è un esempio di trasmissione sincrona

Vi sono differenti interfacce di connessione seriale

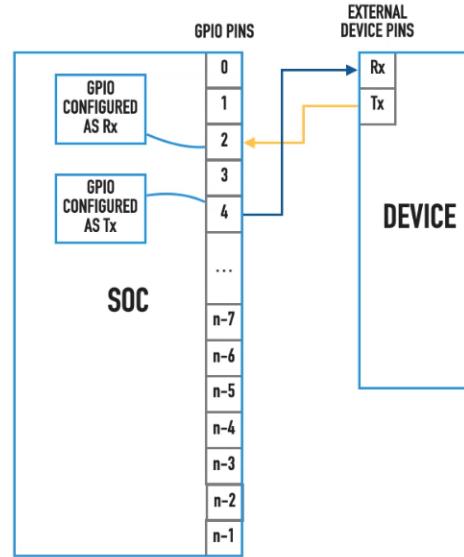
- Point-to-point
 - Universal Asynchronous Receiver / Transmitter (UART)
 - Universal Synchronous / Asynchronous Receiver / Transmitter (USART)
 - può supportare diversi protocolli point-to-point

SERIAL INTERCONNECTIONS

- ▶ Buses
 - ▶ Modbus (RS-485)
 - ▶ Inter-integrated circuit - I²C bus (Philips, NXP)
 - ▶ 1-Wire® (Dallas Semiconductor)
 - ▶ CAN (Controller Area Network) for automotive applications
- ▶ Sort of a bus
 - ▶ Serial peripheral Interconnect (SPI) (serial, point to point with optional selection signals)

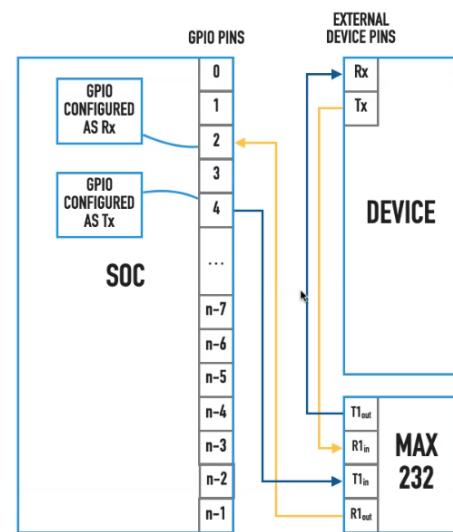


- ▶ UART and USART include optional control signals
- ▶ At least two digital signals are required:
 - ▶ Transmit (Tx) - output signal
 - ▶ Receive (Rx) - input signal
 - ▶ The Tx pin in one device is connected to the Rx pin in the other
- ▶ Bit rates range from few hundred to few million per second



Ogni sistema embedded possiede almeno una di queste periferiche specie quella UART. Queste periferiche sono versatili e supportano diversi protocolli di trasmissione che includono un grosso range di impostazioni che includono lunghezza di frame, correzione di errore, ecc..

- ▶ On SoCs and MCUs, usually, the voltage levels of UART and USART signals are those used to supply chips with power:
 - ▶ Positive power-supply voltage (**Vcc**), e.g. 5V or 3.3V, for logical one
 - ▶ **Ground (GND)** voltage for logical zero
 - ▶ These are called **Transistor-Transistor Logic (TTL)** levels
 - ▶ Voltage level translators are **required** to connect electrically different interfaces (e.g. a TTL UART to an RS-232) to avoid damages
 - ▶ Even connecting a device operating with $V_{cc}=3.3V$ to another with $V_{cc}=5V$ should be avoided



A bidirectional voltage level translator (MAX232) is used to connect the RS-232 port of the device to the TTL UART of the SoC. The same GND is used by the three devices.

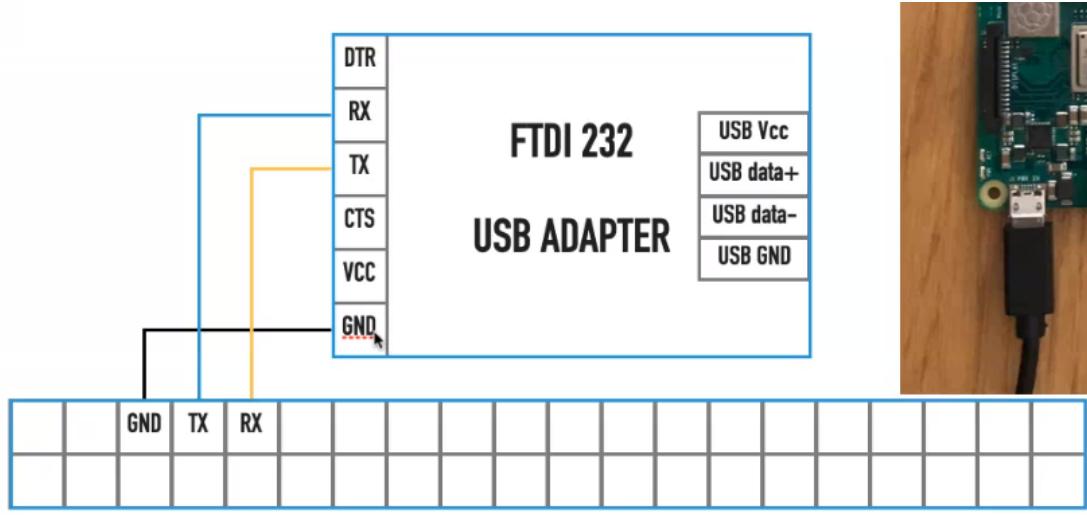
Alcune interfacce hanno necessità di alcuni livelli di tensione specifici; per questo vengono utilizzati dei traduttori TTL (transistor-transistor Logic) per evitare problemi ed eventuali danni alle componenti.

UART / USART Nel Raspberry Pi



i pin UART/USART sono GPIO14 e GPIO15

Un adattatore con TTL permette di connettere un pc provvisto di USB al Raspberry; è richiesto che i cavi di terra e di potenza siano connessi correttamente (Vcc e GND)



Tramite il FTDI possiamo utilizzare il raspberry pi in modalità "bare-metal" e interagire con esso. Un tipico utilizzo è quello della programmazione interattiva on-target dove (in questo caso) viene eseguito `pijForth`, ovvero un interprete forth per pi

queste sono le informazioni correnti della macchina che esegue forth

Standard FORTH

- ▶ The FORTH standard word ! (store) is used to write a word-sized value (32 bits) to memory. Store pops the memory address from the Top Of Stack (TOS) and the value to write from the element under the TOS
- ▶ The LED is switched on by writing a '1' to bit 29 of the GPSET0 register, which controls GPIOs 0-29, with the following code:

```
20000000 3F20001C !
```

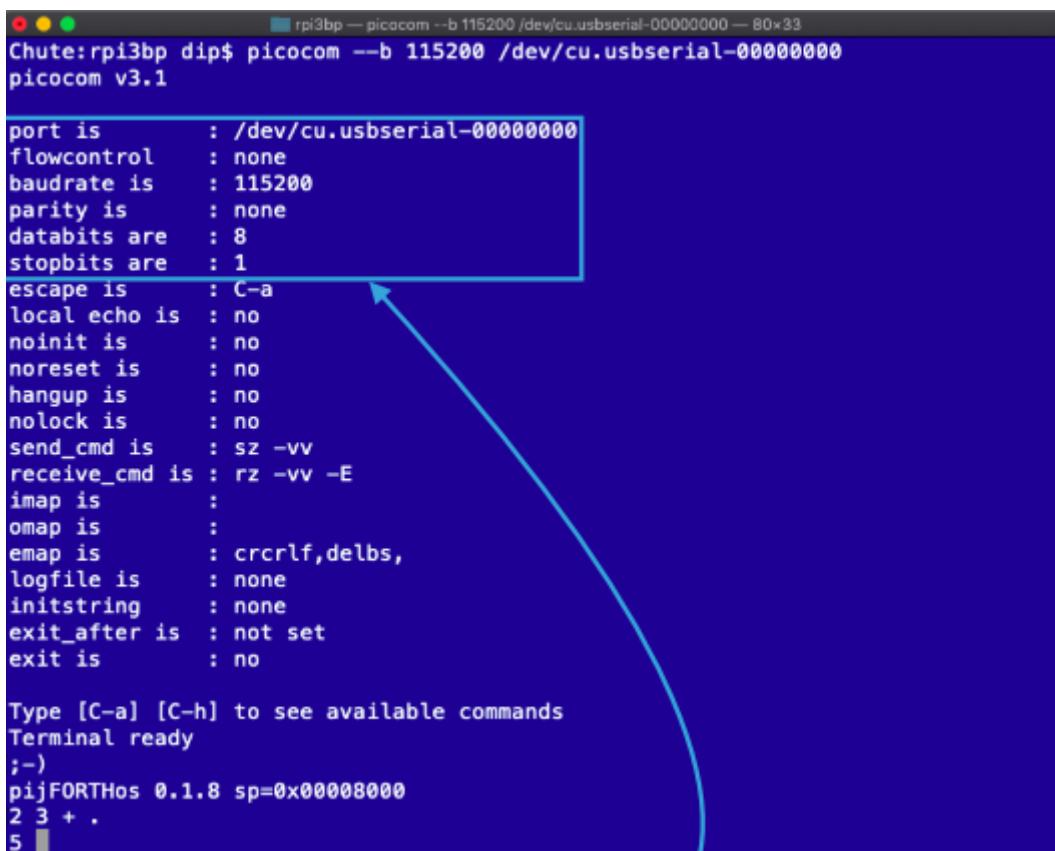
- ▶ The LED is switched off similarly by writing a '1' to bit 29 of the GPCLR0 register:

```
20000000 3F200028 !
```

Lezione 19 - 07/12/2021

Linguaggio Forth

Tra i linguaggi utilizzati per i sistemi embedded, forth è quello più ad "alto livello". Nell'immagine sotto viene caricato pijFORTHos nel raspberry Pi. Un emulatore del terminale è "lanciato" nella macchina target con gli stessi parametri VCP come l'UART Pi: 115200 bit/s, 1 bit di stop, 8 bit di dati, nessun controllo di parità, nessun controllo del flusso.



```
rpi3bp — picocom --b 115200 /dev/cu.usbserial-00000000 — 80x33
Chute:rpi3bp dip$ picocom --b 115200 /dev/cu.usbserial-00000000
picocom v3.1

port is      : /dev/cu.usbserial-00000000
flowcontrol  : none
baudrate is  : 115200
parity is    : none
databits are : 8
stopbits are : 1
escape is    : C-a
local echo is: no
noinit is   : no
noreset is  : no
hangup is   : no
nolock is   : no
send_cmd is : sz -vv
receive_cmd is: rz -vv -E
imap is      :
omap is      :
emap is      : crcrlf,delbs,
logfile is   : none
initstring   : none
exit_after is: not set
exit is      : no

Type [C-a] [C-h] to see available commands
Terminal ready
;-)
pijFORTHos 0.1.8 sp=0x00008000
2 3 + .
5
```

Il motivo per cui studiamo FORTH è perchè possiamo interagire interattivamente con la macchina.

- ▶ The FORTH standard word @ (fetch) is used to read a word-sized value (32 bits) from memory. Fetch pops the memory address from the Top Of Stack (TOS), then pushes the read value on the stack
- ▶ The value of GPFSEL2 is thus read with the following code:

```
HEX
3F200008 @ U.
80000000
```

- ▶ Register GPFSEL2 contains 0x08000000 so bits 29-27 (GPIO29) have value 0b001. The pin is configured as an output so its level drives the green LED on the Pi

L'operatore ! in FORTH è usato per scrivere un valore in memoria. Store prende l'indirizzo di memoria dal TOS (Top Of Stack) e il valore da scrivere dall'elemento in fondo al TOS

- ▶ The LED is switched on by writing a '1' to bit 29 of the GPSET0 register, which controls GPIOs 0-29, with the following code:

```
20000000 3F20001C !
```

- ▶ The LED is switched off similarly by writing a '1' to bit 29 of the GPCLR0 register:

```
20000000 3F200028 !
```

In forth si possono specificare i significati delle "word", in maniera tale da svolgere in maniera ottimale i compiti. Il vantaggio di Forth è che si può avere un interprete all'interno di macchine piccole.

Each of the words comprising the definition of WASHER has already been defined in our washing-machine application. For example, let's look at our definition of RINSE:

```
: RINSE    FILL AGITATE DRAIN ;
```

As you can see, the definition of RINSE consists of a group of words: FILL, AGITATE, and DRAIN. Once again, each of these words has been already defined elsewhere in our washing-machine application. The definition of FILL might be:

```
: FILL    FAUCETS OPEN  TILL-FULL  FAUCETS CLOSE ;
```

In this definition we are referring to *things* (faucets) as well as *actions* (open and close). The word TILL-FULL has been defined to create a "delay-loop" which does nothing but mark time until the water-level switch has been activated, indicating that the tub is full.

La differenza tra Forth ed altri linguaggi di alto livello è la mancanza di parentesi, parametri e formati e questo lo rende molto più leggibile

```

julia> [ 12 , 34 ]
2-element Vector{Int64}:
12
34

julia> [ 12.3 , 34 ]
2-element Vector{Float64}:
12.3
34.0

julia>
> pwd
/Users/dip/Documents/Didattica/Corsi attivi/Embedded Systems/ES-2021-2022/Lezioni/2021-12-02-1100-1300/ARM-Assembly/4-Bare-metal/RPi3Bp
> cd ..
> cd ..
> cd ..
> cd ..
> gforth
Gforth 0.7.3, Copyright (C) 1995-2008 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit

```

inizializzazione di GForth

Vista la natura molto flessibile di Forth, possiamo fare qualsivoglia specifica sul modo in cui vengono scritte le procedure

```

julia> [ 12.3 , 34 ]
2-element Vector{Float64}:
12.3
34.0

julia>
> pwd
/Users/dip/Documents/Didattica/Corsi attivi/Embedded Systems/ES-2021-2022/Lezioni/2021-12-02-1100-1300/ARM-Assembly/4-Bare-metal/RPi3Bp
> cd ..
> cd ..
> cd ..
> cd ..
> gforth
Gforth 0.7.3, Copyright (C) 1995-2008 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit
15 SPACES          ok
42 emit * ok
65 emit A ok
48 emit 0 ok
15 SPACES 42 EMIT 42 EMIT      ** ok
: STAR 42 EMIT ;  ok

```

L'istruzione 15 SPACES mi permette intuitivamente di stampare 15 spazi a schermo. 42 Emit mi permette di stampare un carattere (l'asterisco). Si possono definire nuove word tramite : NOMEWORD ISTRUZIONE;

In Forth, quindi, la flessibilità è massima: posso specificare quel che voglio

```

BAR BLIP BAR BLIP BLIP CR
*****
*
*****
*
*
ok
:F   BAR BLIP BAR BLIP BLIP CR ; ok
F
*****
*
*****
*
*
ok
: STAR  [CHAR] * EMIT ; redefined STAR  ok

```

in questo ultimo esempio viene definita la procedura STAR come il prodotto tra Emit e l'indirizzo di Char. Ci rendiamo conto di quanto il linguaggio forth sia "potente" dal punto di vista della manipolazione

```

42  ok
.S <1> 42  ok
65  ok
.S <2> 42 65  ok
48  ok
.S <3> 42 65 48  ok
EMIT EMIT EMIT 0A* ok
. 5024825840
:42: Stack underflow
>>>.<<<
Backtrace:
$12A804B20 throw
: GREET  ." Hello, I speak Forth " ; ok
GREET Hello, I speak Forth  ok
G
:45: Undefined word
>>>G<<<
Backtrace:
$12A808A20 throw
$12A81ECB8 no.extensions
$12A808CE0 interpreter-notfound1
CR GREET
Hello, I speak Forth  ok

```

Questa non è una funzione di "printf": questa "CR" è una parola che evoca il compilatore

Definizioni varie

where the names symbolize "yards-to-inches" and "feet-to-inches." Here's what they do:

```
10 YARDS>IN ↵360 ok  
2 FT>IN ↵24 ok
```

If we *always* want our result to be in inches, we can define:

```
: YARDS 36 * ;↵ok  
: FEET 12 * ;↵ok  
: INCHES I ;↵ok
```

So that we can use the phrase

```
10 YARDS 2 FEET + 9 INCHES + ↵393 ok
```

Notice that the word INCHES doesn't do anything except remind the human user what the nine is for. If we really want to get fancy, we can add these three definitions:

L'utilizzo di forth, in questo caso, serve per definire delle variabili "scalabili" (iarde, piedi, pollici)

Operatore modulo

/MOD	(n1 n2 — rem quot)	Divides; returns remainder and quotient
MOD	(n1 n2 — rem)	Divides; returns remainder only

Playing Doubles

The next four stack manipulation operators should look vaguely familiar:

2SWAP	(d1 d2 — d2 d1)	Reverses the top two pairs of numbers
2DUP	(d — d d)	Duplicates the top pair of numbers
2OVER	(d1 d2 — d1 d2 d1)	Duplicates the second pair of numbers
2DROP	(d1 d2 — d1)	Discards the top pair of numbers

Questo operatore permette di operare con due word contemporaneamente

```
. 3 ok
. 2 ok
12 ok
.s <1> 12 ok
12 ok
.s <2> 12 12 ok
= ok
.s <1> -1 ok
if
:106: Interpreting a compile-only word
>>>if<<<
Backtrace:
$12A807AE8 throw
[if]
:107: Stack underflow
>>>[if]<<<
Backtrace:
$12A804B20 throw
12 13 = ok
.s <1> 0 ok
: ?FULL 12 = IF ." It's full " THEN ; ok
12 ?FULL It's full ok
11 ?FULL ok
```

questo è un esempio di condizione. Può essere tradotto come If full=12 "it's full"

Lezione 20 - 09/12/2021

Linguaggio Forth (parte 2)

FORTH ON RASPBERRY PI 3 B+

- ▶ After the execution of LED the stack contains the values:

```
LED
U.
3F200028 U.
3F20001C U.
20000000
```

- ▶ So to switch the LED on or off it is possible to use the values left on the stack by LED removing the unwanted register address for each case
- ▶ To turn the LED on the TOS value (address of GPCLR0) is DROPPed

```
LED
DROP
!
```

- ▶ To turn the LED of the value under TOS is NIPped (NIP can be defined as SWAP DROP)

Questi sono i valori nello stack dopo l'esecuzione di LED.

- ▶ Now, two more words may conveniently defined

```
: ON      DROP ! ;
: OFF    NIP ! ;
```

- ▶ After these words are defined, two FORTH phrases can be used to drive the LED instead of writing to registers

LED ON

- ▶ to turn the LED on and

LED OFF

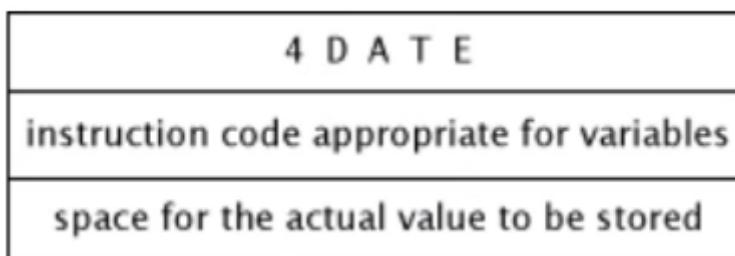
- ▶ to turn it off
- ▶ This code defines a small programming language to drive the LED

Le due definizioni vengono sfruttate per utilizzare in maniera veloce DROP(viene preso l'indirizzo di GPCLR0) e NIP(SWAPDROP GP qualcosa, approfondire)

Il professore utilizza l'interprete gforth (FORTH associato alla toolchain GCC)

```
==> Analytics
install: 328,124 (30 days), 598,314 (90 days), 3,021,251 (365 days)
install-on-request: 87,482 (30 days), 112,114 (90 days), 250,203 (365 days)
build-error: 80 (30 days)
> gforth
Gforth 0.7.3, Copyright (C) 1995-2008 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit
VARIABLE DATE ok
```

DATE non è definita, bisogna definirla. Utilizzando VARIABLE DATE, la inseriamo nel "dizionario" di forth e questa la renderà "leggibile" agli occhi dell'interprete. Nel dizionario una variabile "chiama un valore" a differenza di altri linguaggi dove la variabile è semplicemente un "container"



Viene "lasciato" un puntatore al container associato al nome "DATE"

```

DATE 10 DUMP
12C042340: 0C 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....1
ok
DATE 20 DUMP
12C042340: 0C 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....1
12C042350: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....1
ok
DATE 20 - 10 DUMP
12C042320: 04 00 00 00 00 00 00 80 - 44 41 54 45 20 20 20 20 .....DATE
ok
DATE 20 - 20 DUMP
12C042320: 04 00 00 00 00 00 00 80 - 44 41 54 45 20 20 20 20 .....DATE
12C042330: 50 E7 D7 02 01 00 00 00 - 00 00 00 00 00 00 00 00 P.....1
ok
DATE 20 - 30 DUMP
12C042320: 04 00 00 00 00 00 00 80 - 44 41 54 45 20 20 20 20 .....DATE
12C042330: 50 E7 D7 02 01 00 00 00 - 00 00 00 00 00 00 00 00 P.....1
12C042340: 0C 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....1
ok

```

Da come possiamo osservare nel dump (20 - numero è il range di indirizzi preso), nel dizionario ad ogni definizione è associato un puntatore. La struttura del dizionario è la stessa e quindi il comportamento delle variabili è identico.

1. word della procedura
2. puntatore p
3. contenuto del valore attuale

```

12C042380: 05 00 00 00 00 00 00 80 - 52 45 53 45 54 20 20 20 .....RESET
12C042390: 30 E5 D7 02 01 00 00 00 - 00 00 00 00 00 00 00 00 00 0 .....I
12C0423A0: 74 20 D8 02 01 00 00 00 - 00 00 00 00 00 00 00 00 00 0 .....t
12C0423B0: 74 20 D8 02 01 00 00 00 - 70 23 04 2C 01 00 00 00 0 .....t .....p#
12C0423C0: 80 A3 D8 02 01 00 00 00 - A4 F2 D7 02 01 00 00 00 .....t
12C0423D0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 0 .....t
12C0423E0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 0 .....t
12C0423F0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 0 .....t
12C042400: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 0 .....t

```

RESET è diversa dalle altre word: si comporta come una "calling word" perchè "decide" cosa deve fare il computer; questa distinzione è importante

```
$12C01ECB8 no.extensions
$12C008CE0 interpreter-notfound1
RESET ?
:34: Invalid memory address
RESET >>>?<<<
Backtrace:
$12C017C90 @
EGGS ? 0 ok
1 EGGS ok
.S <2> 1 12C042370 ok
+! ok
EGGS ? 1 ok
1 EGGS +! ok
EGGS ? 2 ok
1 EGGS +! ok
EGGS ? 3 ok
: EGG 1 EGGS +! ; ok
EGG ok
EGG ok I
EGG ok
EGGS ? 6 ok
EGG ok
EGGS ? 7 ok
```

la word EGGS rappresenta una variabile che può essere implementata. Nel seguente codice viene definita una procedura del tipo

EGG 1 EGGS +! --> aumenta di 1 l'occorrenza di EGGS

Utilizzo delle costanti

In forth è possibile definire delle costanti (elementi non cambiabili), utilizzando ad esempio:

```
220 CONSTANT LIMIT
```

la word limit sarà una costante con valore 220

Possiamo utilizzare questa costante per effettuare dei controlli

```
?TOO-HOT LIMIT > IF . "Danger --reduce heat" THEN;
```

```
Backtrace:
$12C008A20 throw
$12C01ECB8 no.extensions
$12C008CE0 interpreter-notfound1
45 LIMIT ok
.S <2> 45 220 ok
```

Il funzionamento delle costanti è simile alla variabile, con l'unica differenza che **non fornisce modi per cambiare il valore della word**: in parole povere, nei bit è specificato il "comportamento". Nello stack infatti i valori sono separati

```
: PHOTOGRAPH SHUTTER OPEN TIME EXPOSE SHUTTER CLOSE ;
```

Here the word SHUTTER has been defined as a constant so that execution of SHUTTER returns the hardware address of the camera's shutter. It might, for example, be defined:

```
HEX
FFFF3E27 CONSTANT SHUTTER
DECIMAL
```

The words OPEN and CLOSE might be defined simply as

```
: OPEN 1 SWAP ! ;
: CLOSE 0 SWAP ! ;
```

In questa procedura viene definita l'apertura e la chiusura dell'otturatore

In base al tipo di dispositivo e alle convenzioni del GPIO, viene utilizzato l'indirizzo che controlla quella porta. Nella convenzione, se vogliamo manipolare quell'indirizzo, tale indirizzo deve essere nel TOS per essere memorizzato: lo swap ci serve a questo.

Nel caso del raspberry PI abbiamo due registri: GPSET0 e GPCLR0

```
: LED 20000000 GPSET0 GPCLR0 ;
```

- ▶ When executed
 - ▶ the value **0x20000000** is pushed on the stack
 - ▶ **GPSET0** is executed
 - ▶ the execution of **GPSET0** pushes **0x3F00001C** on the stack
 - ▶ **GPCLR0** is executed
 - ▶ the execution of **GPSET0** pushes **0x3F000028** on the stack

ovviamente GPSET0 e CPCLR0 sono due procedure pre-definite

- ▶ Complete code of the new language to drive the green LED:

```
HEX
3F200008 CONSTANT GPFSEL2
3F20001C CONSTANT GPSET0
3F200028 CONSTANT GPCLR0

: LED    20000000 GPSET0 GPCLR0 ;
: ON     DROP ! ;
: OFF    NIP ! ;
```

- ▶ This code only works on the Raspberry Pi 3 B+ because the on-board LED is connected to GPIO 29 (Pi 4 B+ has the LED connected to GPIO 42)
- ▶ In most applications GPIOs exposed in the GPIO connector are used
- ▶ Pi models may differ in the base Peripheral address (e.g. 0x3F000000 for the Pi 3, 0xFE000000 for the Pi 4) but the register offsets are the same

3F nel Pi 4 deve essere cambiato in FE

LED lampeggianti in Forth

BLINKING LED IN FORTH

- ▶ To make the LED blink the phrases LED ON and LED OFF can be executed one after the other interleaving the executions with a delay. A possible way to do it simply requires the definition of the new word DELAY (n --):

```
LED ON 10000 DELAY LED OFF
```

- ▶ A second delay can be added to make the LED transition visible, for example by holding the LED on and off for the same amount of time:

```
LED ON 10000 DELAY LED OFF 10000 DELAY
```

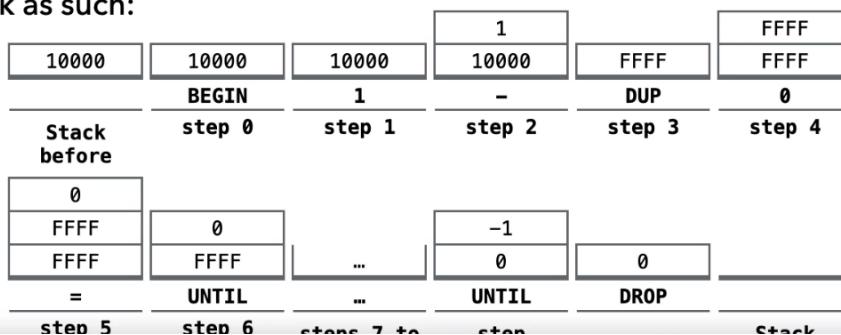


Questa è una maniera immediata per svolgere il blinking del LED. Dovremmo definire una procedura DELAY che prenda in input i millisecondi per eseguire un ciclo che non faccia nulla

- ▶ As a first attempt, the DELAY word can be defined using a busy loop

```
: DELAY BEGIN 1 - DUP 0 = UNTIL DROP ;
```

- ▶ The word consumes the TOS and leaves no values on the stack. The **steps** of the execution of DELAY with 0x10000 as TOS value modify the stack as such:

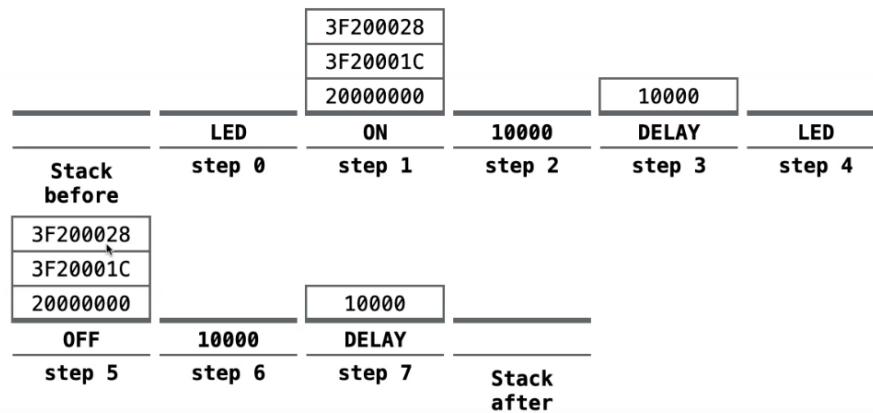


Implementazione della procedura di DELAY.
 allo step 2 viene inserito 1 nel TOS, allo step 3 viene effettuata una sottrazione dal valore inferiore allo stack al TOS. Allo step 4 viene fatta una comparazione con lo stack e ci accorgiamo che il valore non è uguale, quindi ripetiamo più volte il ciclo (step 7 to 7*FFF+5) finchè non compare, in cima allo stack, -1: valore logico per indicare FALSO. e quindi si può effettuare il drop.

► **The execution of the sequence:**

```
LED ON 10000 DELAY LED OFF 10000 DELAY
```

► **has the following effects on the stack:**



comportamento dello stack

Esercizio

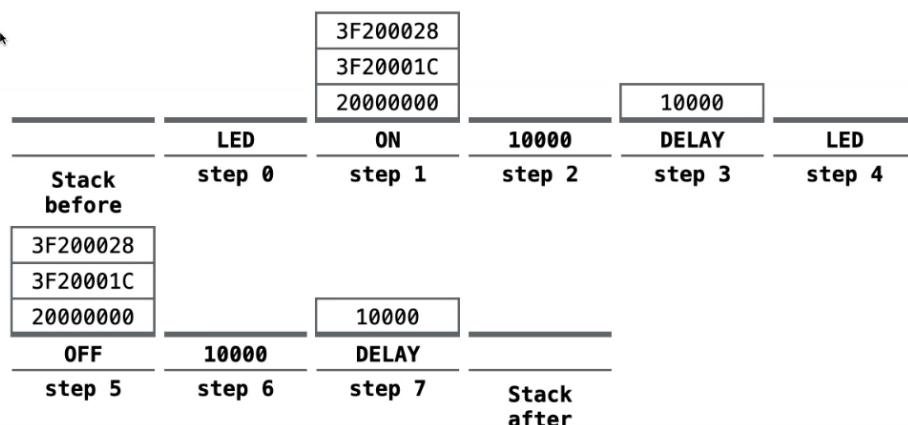
Provare a scrivere un comando che racchiuda tutta la procedura

E.S: LED BLINK

► **The execution of the sequence:**

```
LED ON 10000 DELAY LED OFF 10000 DELAY
```

► **has the following effects on the stack:**



Double Length Variable and Costants

Possiamo definire delle variabili a lunghezza doppia utilizzando la word 2VARIABLE

2VARIABLE DATE

in ogni cella abbiamo una variabile, in forth: quindi la variabile in una macchina a 64 bit può sfruttare due blocchi ed avere 128 bit come dimensione

Nel caso utilizzassimo:

800,000 DATE 2!

Quando si utilizzano le variabili a doppia lunghezza, nello stack non c'è distinzione di spaziamento: semplicemente vengono manipolate due celle adiacenti nello stack

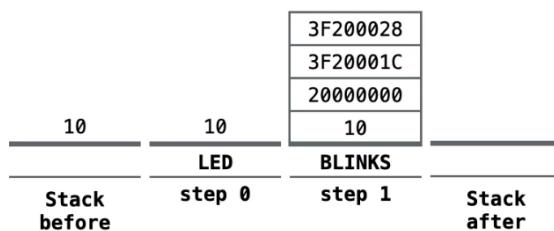
APPROFONDIMENTI 3DUP

ESERCIZIO

- ▶ The LED driving language could be extended with the word BLINKS so that the phrase:

10 LED BLINKS

- ▶ would blink the LED 10 times



Lezione 21 - 14/12/2021

FORTH Language (continua)

2VARIABLE XXX

(-) xxx: (- addr)

Creates a double-length variable named xxx; the word xxx returns its address when executed.

2CONSTANT XXX

(d -) xxx: (- d)

Creates a double-length constant named xxx with the value d; the word xxx returns d when executed.

2!

(d addr -) or (n1 n2 addr -)

Stores a double-length number (or a pair of single-length numbers) at address addr.

2@

(addr - d) or (addr - n1 n2)

Fetches a double-length number (or a pair of single-length numbers) from addr.

Queste istruzioni sono utili nella manipolazione di variabili a doppia lunghezza.

Array

tipicamente, un'array è un insieme di dati posizionati in maniera contigua rispetto alla memoria ed hanno **tutti lo stesso formato**.

Ad esempio, la frase VARIABLE DATE crea una definizione che concettualmente appare come

```
4 DATE code spazio_1
```

se scriviamo 1 CELLS ALLOT, verrà aggiunta una nuova cella

```
4 DATE code spazio_1 spazio_2
```

```
VARIABLE DATE ok
1 CELLS ok
.S <1> 8 ok
ALLOT ok
23 DATE 2!
:6: Stack underflow
23 DATE >>>2!<<<
Backtrace:
$142004B20 throw
23 0 DATE 2! ok
DATE 2@ ok
D. 23 ok
VARIABLE LIMITS 4 CELLS ALLOT ok
220 LIMITS ! ok
```

gli ultimi due comandi sono esattamente gli stessi.

```
LIMITS 16 DUMP
142042378: 20 02 00 00 00 00 00 00 - 40 03 00 00 00 00 00 00 .....@.....
142042388: 00 00 00 00 00 00 - .....@.....
ok
LIMITS 5 DUMP
142042378: 20 02 00 00 00 - .....@.....
ok
LIMITS 5 CELLS DUMP
142042378: 20 02 00 00 00 00 00 00 - 40 03 00 00 00 00 00 00 .....@.....
142042388: 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....@.....
142042398: 00 00 00 00 00 00 00 00 - .....@.....
i ok
```

Non necessitiamo per forza di dover specificare i byte: su Forth basta specificare la keyword; quindi basta fare il dump specificando "5 CELLS".

The keyword "HERE" ci da informazioni della "fine del dizionario"

```

VARIABLE LIMITS ok
HERE . 147042348 ok
147042318 ok
10 CELLS DUMP
147042318: 00 19 04 47 01 00 00 00 - 06 00 00 00 00 00 00 00 80 ...G.....
147042328: 4C 49 4D 49 54 53 20 20 - 50 E7 E5 02 01 00 00 00 LIMITS P.....
147042338: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 .....
147042348: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 .....
147042358: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 .....
147042368: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 .....
147042378: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 .....
147042388: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 .....

```

L'indirizzo 147042348 è l'ultimo indirizzo del dizionario e corrisponde all'indirizzo della cella dove viene memorizzato il valore di LIMITS

```

LIMITS 5 CELLS DUMP
147042340: 00 02 00 00 00 00 00 00 - 40 03 00 00 00 00 00 00 00 .....@.....
147042350: 50 03 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 00 P.....
147042360: 00 00 00 00 00 00 00 00 - .....ok
360 LIMITS 3 CELLS + ! ok
370 LIMITS 4 CELLS + ! ok
LIMITS 5 CELLS DUMP
147042340: 00 02 00 00 00 00 00 00 - 40 03 00 00 00 00 00 00 00 .....@.....
147042350: 50 03 00 00 00 00 00 00 - 60 03 00 00 00 00 00 00 00 P.....
147042360: 70 03 00 00 00 00 00 00 - .....p.....
ok

```

dump dei 5 dati delle celle

```

400 LIMITS 5 CELLS + ! ok
LIMITS 6 CELLS DUMP
147042340: 00 02 00 00 00 00 00 00 - 40 03 00 00 00 00 00 00 00 .....@.....
147042350: 50 03 00 00 00 00 00 00 - 60 03 00 00 00 00 00 00 00 P.....
147042360: 70 03 00 00 00 00 00 00 - 00 04 00 00 00 00 00 00 00 p.....

```

In realtà, anche se l'allocazione dell'array è di 5 celle, l'istruzione 400 LIMITS 5 CELLS + ! non fa nient'altro che aggiungere dati, quindi non verrà lanciata un'eccezione di memoria non gestita (come avviene tipicamente negli altri linguaggi di programmazione)

in genere quindi, ALLOT non fa nient'altro che spostare il puntatore che punta all'ultimo indirizzo della cella, "più in basso"

```

0 LIMIT ? 200 ok
1 LIMIT ? 340 ok
2 LIMIT ? 350 ok
3 LIMIT ? 360 ok
4 LIMIT ? 370 ok
5 LIMIT ? 147042318 ok

```

il modo in cui si accedere alle celle di un array è il seguente: NUM VARIABLE ?

FILL**(addr n char -)**

Fills n bytes of memory, beginning the address addr, with value char.

ERASE**(addr n -)**

Stores zeroes into n bytes of memory, beginning at address addr.

FILL è una keyword utile ad inizializzare un array (un metodo diverso rispetto a quelli precedenti), mentre ERASE non fa nient'altro che azzerare la memoria

```
COUNTER CELLS COUNTS +;
```

questa word ci restituisce l'indirizzo di..

APPROFONDIRE

define an additional word which expects a category number and prints an output message, like this:

```
: LABEL ( category -- )
CASE
  REJECT    OF ." reject "    ENDOF
  SMALL     OF ." small "    ENDOF
  MEDIUM    OF ." medium "   ENDOF
  LARGE     OF ." large "   ENDOF
  EXTRA-LARGE OF ." extra large " ENDOF
  ERROR     OF ." error "   ENDOF
ENDCASE ;
```

For example:

```
SMALL LABEL↔small ok
```

Now we can define EGGSIZE using three of our own words:

```
: EGGSIZE CATEGORY DUP LABEL TALLY ;
```

Questa è un'implementazione del costrutto Switch case, dove OF e ANDOF delimitano inizio e fine della lettura della label

APPROFONDIRE

Byte Array

In FORTH, una singola cella può anche contenere un solo byte: quindi devo usare CELLS per manipolare gli offset, visto che ogni elemento corrisponde ad un indirizzo

C! -> questa istruzione memorizza il byte del carattere nell'indirizzo di destinazione

C@ -> riprende il valore del byte dall'indirizzo di destinazione

Suppose we want permanent values in our LIMITS array. Instead of saying

```
VARIABLE LIMITS 4 CELLS ALLOT
```

we can say

```
CREATE LIMITS 220 , 340 , 170 , 100 , 190 ,
```

sono due istruzioni simili, solo che tipicamente la seconda viene caricato da un "file disco"

```
HERE OK
CREATE LIMITS3 ok
HERE ok
.S <2> 147042438 147042460 ok
147042438 30 DUMP
147042438: E8 23 04 47 01 00 00 00 - 07 00 00 00 00 00 00 80 .#.G.....
147042448: 4C 49 4D 49 54 53 33 20 - 50 E7 E5 02 01 00 00 00 LIMITS3 P.....
147042458: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....ok
200 , ok
147042438 30 DUMP
147042438: E8 23 04 47 01 00 00 00 - 07 00 00 00 00 00 00 80 .#.G.....
147042448: 4C 49 4D 49 54 53 33 20 - 50 E7 E5 02 01 00 00 00 LIMITS3 P.....
147042458: 00 00 00 00 00 00 00 00 - 00 02 00 00 00 00 00 00 .....ok
HERE . 147042468 ok
```

Questa sequenza di istruzioni permette di memorizzare 200 alla fine del dizionario, per poi successivamente "espandere" l'array di una cella

La keyword "," permette questa espansione

Esempio

Similarly, for instance, we could store each of the values used in our egg-surfing definition CATEGORY as follows:

```
CREATE SIZES 18 C, 21 C, 24 C, 27 C, 30 C, 255 C,
```

This would allow us to redefine CATEGORY using a DO loop rather than as a series of nested IF...THEN statements, as follows

```
: CATEGORY 6 0 DO DUP SIZES I + C@ < IF DROP I LEAVE THEN LOOP ;
```

Sfruttando la "," e il concetto di byte array si può ridefinire la keyword "CATEGORY" utilizzando un loop

Implementazioni FORTH

La struttura del codice compilato e il corrispondente modello di esecuzione è differente in base all'approccio che si vuol seguire

- Approccio indiretto
 - pijFORTHos

- Approccio diretto
 - GForth

Implementazioni del dizionario

DICTIONARY IMPLEMENTATION

- ▶ The entry contains
 - ▶ The link pointer
 - ▶ The number of characters in the name field
 - ▶ The name field, which is an array of characters padded with spaces (20) to a multiple of a cell size
 - ▶ The code pointer field, which defines the behavior of the word on execution
 - ▶ The data field (1 cell=8 bytes)

```
Gforth 0.7.3, Copyright (C) 1995-2008 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit
HEX HERE .S <1> 10DB4D2C0 ok
VARIABLE ORANGES ok
HERE .S <2> 10DB4D2C0 10DB4D2F0 ok
2DUP SWAP - . 30 ok
OVER - OVER SWAP .S <3> 10DB4D2C0 10DB4D2C0 30 ok
DUMP
10DB4D2C0: B0 BB B4 0D 01 00 00 00 - 07 00 00 00 00 00 00 80 .....
10DB4D2D0: 4F 52 41 4E 47 45 53 20 - 8A 07 A7 0D 01 00 00 00 .....ORANGES.....
10DB4D2E0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
ok
```

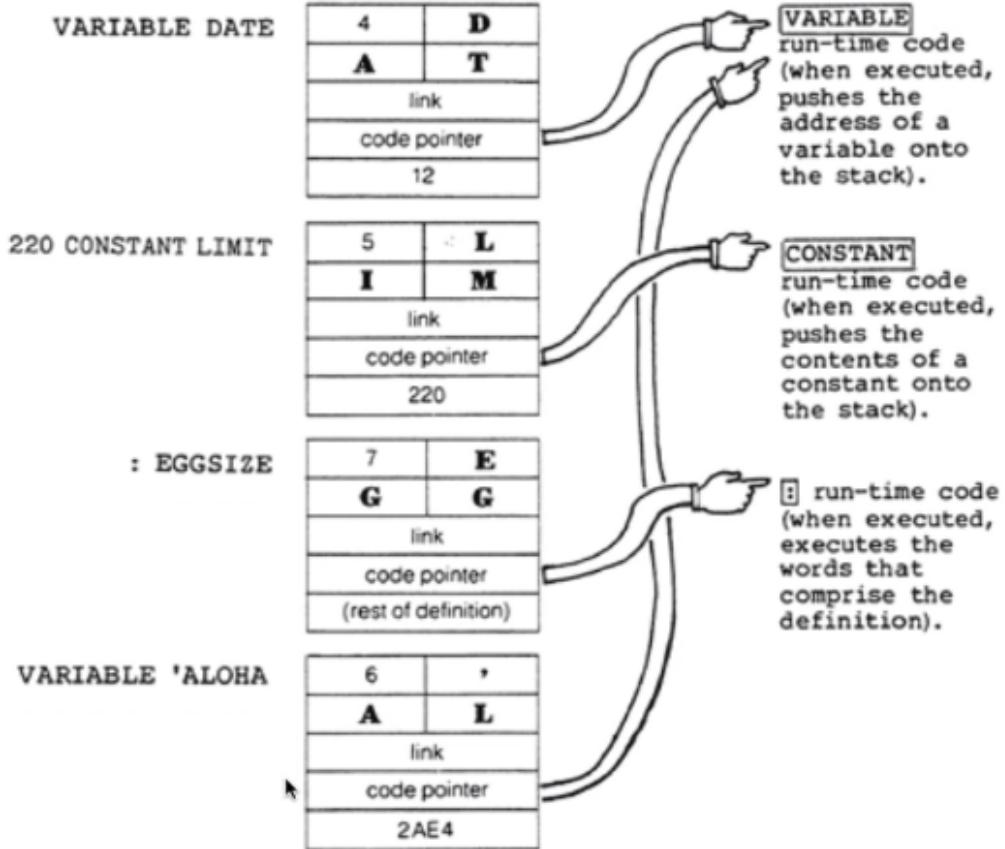
Condizionale in FORTH

Forth's text interpreter uses a word related to tick that returns a zero flag if the word is found. The name and usage of the word varies, but the conditional structure of the INTERPRET phrase always looks like this:

```
(find the word) IF      (convert to a number)
    ELSE   (executeI the word)
THEN
```

that is, if the string is *not* a defined word in the dictionary, INTERPRET tries to convert it as a number. If it is a defined word, INTERPRET executes it.

abbasta autoesplicativo: se la word viene trovata, esegui la word, altrimenti converti la stringa in un numero.



"struttura del dizionario di FORTH" -> viene mostrata la struttura di ogni keyword nel dizionario di FORTH.

FORTH Data flow

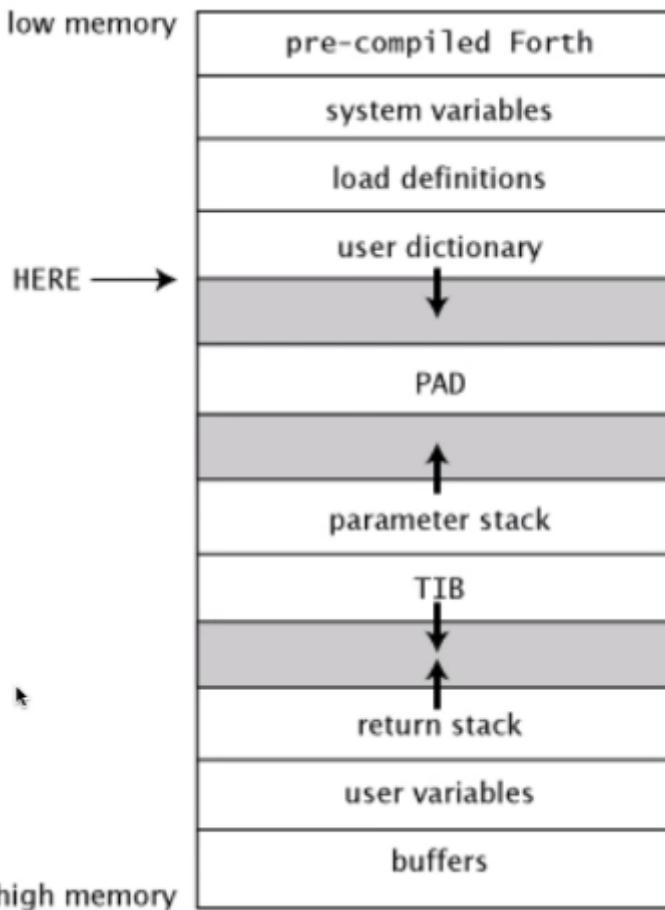
The data field of a colon definition contains the xts of the previously defined words which comprise the definition. Here is the dictionary entry for the definition of PHOTOGRAPH, which is defined as

```
: PHOTOGRAPH
    SHUTTER OPEN  TIME EXPOSE  SHUTTER CLOSE ;
```

10	P
H	O
link	
code pointer	
adr of SHUTTER	
adr of OPEN	
adr of TIME	
adr of EXPOSE	
adr of SHUTTER	
adr of CLOSE	
adr of EXIT	

un'istruzione complessa come PHOTOGRAPH viene memorizzata nel dizionario nel modo seguente: le prime keyword chiamate vengono messe in cima, subito dopo il code pointer

Geografia di FORTH



questa è la mappatura di memoria di un sistema Forth:

La sezione riguardante le variabili di sistema è creata dal core di Forth ed è utilizzata dall'intero sistema (non viene utilizzato dall'utente). La sezione riguardante l'user dictionary cresce verso indirizzi di memoria "più alti" in base alle definizioni che l'utente dà. La cella successiva disponibile nel dizionario in ogni momento è puntata dal CP

Il professore fa un riepilogo di alcune istruzioni presenti su FORTH

Istruzione CREATE

```
: VARIABLE ( -- ) CREATE 0 ,
```

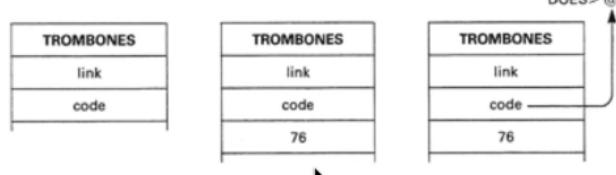
CREATE genera un entry addizionale: associa alla parola creata, un comportamento (l'indirizzo della fine della parola).

Per quanto riguarda CONSTANT, non necessitiamo di questo comportamento, quindi:

```
: CONSTANT ( n -- ) CREATE , DOES>@
```

questa istruzione aggiunge un comportamento (con il fetch verrà individuare l'indirizzo del data field)

The words that precede DOES> specify what the mold will do; the words that follow DOES> specify what the product of the mold will do.



CONSTANT crea qualcosa, specificando come questo qualcosa si comporterà quando verrà eseguito. Quindi vi è una netta distinzione tra compile time e run-time.

La procedura relativa a CREATE è eseguita in compile-time, mentre la procedura relativa a DOES è eseguita run-time.

ESEMPIO

Our next example could be useful in an application where a large number of byte (not CHAR!) arrays are needed. Let's create a defining word called STRING as follows:

```
: STRING    CREATE ALLOT DOES> + ;
```

to be used in the form

```
30 STRING VALVE
```

to create an array thirty bytes in length. To access any byte in this array, we merely say:

```
6 VALVE C@
```

in questo caso viene allocato un'array di 30 byte -> viene creata un'entry di nome VALVE e poi viene eseguita (in tempo di compilazione) ALLOT (che consuma 30). con DOES si aggiunge il comportamento (eseguito in tempo di esecuzione)

Quando viene ridefinita una parola, la nuova definizione viene inserita in cima al dizionario

Lezione 22 - 16/12/2021

FORTH (continua)

(cercare di fare questo esercizio 10 LED BLINKS)

l'istruzione MARKER "distrugge" tutte le keyword user-defined nel dizionario FORTH per evitare che ci sia una dimensione del dizionario molto grosso. Per gestire i sorgenti si potrebbe utilizzare la word INCLUDE in questa maniera:

```
INCLUDE blocks.
```

Operatori aritmetici

Prodotto-Rapporto

In Forth esiste un operatore che effettua sia divisione che moltiplicazione:

```
*/
```

```
E.S: WORD1 */ WORD2
```

vengono eseguite più moltiplicazioni e poi alla fine la divisione, per esempio: lo stack contiene i seguenti valori (225 32 100 --)

*/ moltiplicherà dapprima 225 con 32, e poi dividerà per 100. Queste operazioni sono utili per problemi quali la percentuale. Forth memorizza il valore moltiplicato in una cella "double-length" in maniera tale da evitare overflow.

Return stack

>R	(n -)
Takes a value off the parameter stack and pushes it onto the return stack.	
R>	(- n)
Takes a value off the return stack and pushes it onto the parameter stack.	
R@	(- n)
Copies the top item from return stack and pushes it onto the parameter stack.	

possiamo immaginare il "return stack" come uno spazio di memoria temporaneo dove vengono memorizzati, ad ogni ciclo di clock, i valori che vengono manipolati nello stack (valori che subiscono operazioni aritmetiche).

Quindi >R e R> trasferiscono valori da e verso lo stack. Per vedere come funziona, facciamo un esempio:

devi risolvere l'equazione $ax^2 + bx + c$

con 4 valori nello stack -> (a b c x --)

quindi compio le seguenti operazioni:

1. con >R sposto x nello stack di return -> (a b c -- PARAMETER STACK), (x return stack)
2. SWAP ROT scambio la fine con l'inizio -> (c b a -- PARAMETER STACK), (x return stack)
3. R@ (fetch) prelevo il valore dallo stack return -> (c b a x - P S), (x return stack)
4. moltiplico con * -> (c b ax - PS),(x return stack)
5. sommo + -> (c ax+b - PS), (x return stack)
6. R>* (prelevo il valore dallo stack di ritorno e moltiplico) -> (c(ax+b) x), (niente nel return stack)

```

backtrace:
:124808A20 throw
:12481ECB8 no.extensions
:124808CE0 interpreter-notfound1
).0 ok
s <2> 0 0 ok
pad 10 ok
s <4> 0 0 4907607104 10 ok
NUMBER ok
s <4> 123 0 4907607107 7 ok
drop ok
s <2> 123 0 ok
PLUS 0. BL WORD COUNT >NUMBER 2DROP DROP + ." = " . ; ok
2 13 + ok
15 ok
x
41: Undefined word
>>>xx<<<
backtrace:
:124808A20 throw
:12481ECB8 no.extensions
:124808CE0 interpreter-notfound1
: PLUS 13 = 15 ok

```

da approfondire

Operazioni in memoria

vedere delle operazioni interessanti tra cui ACCEPT, KEY

```

ARRAY ( #rows #cols -- ) CREATE DUP , * ALLOT
DOES> ( member: row col -- addr )
ROT OVER @ * + + CELL+ ;

```

qui viene data la definizione di array generale (devo cercare di estrarre più informazioni possibili).

Comportamento dello stack:

Operation	Contents of stack
	row col addr
ROT	col addr row
OVER @	col addr row #cols
*	col addr index
++ CELL+	addr

Con l'istruzione successiva viene allocato un array di 4x4 bytes

DUMP

Here's a silly example which may give you some ideas for more practical applications.
This definition let's you peek into the innards of the word itself:

```
: DUMP-THIS      [ HERE ] LITERAL 32 DUMP ." DUMP-THIS" ;
```

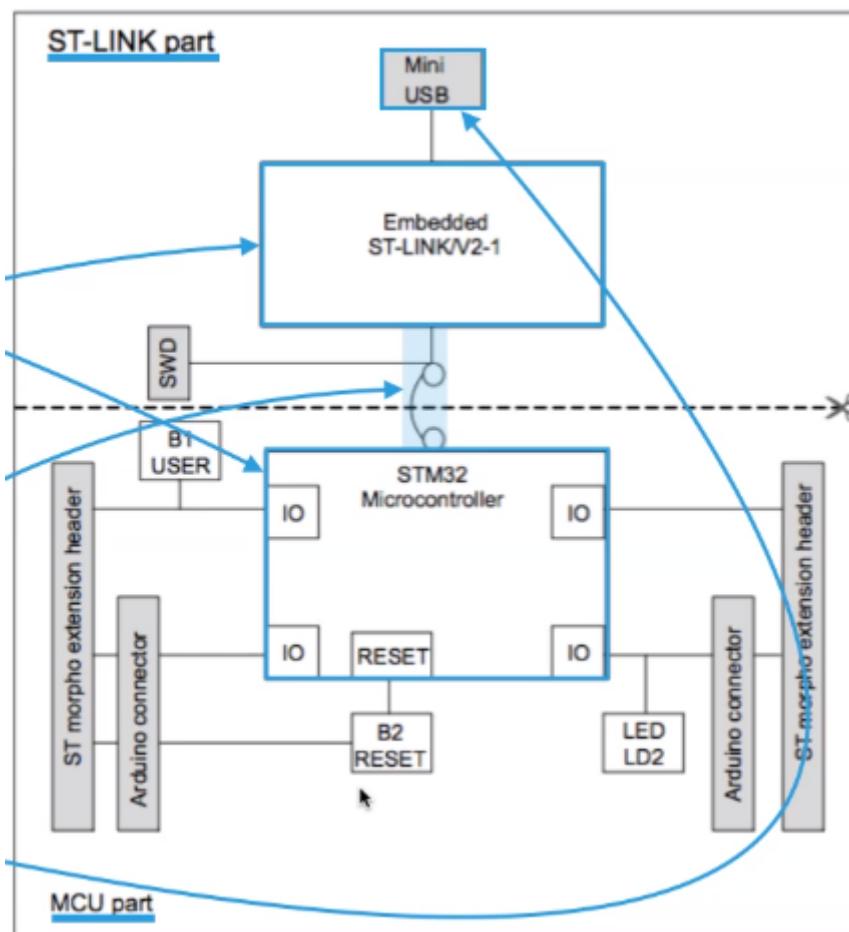
da approfondire

Lezione 23 - 21/12/2021

Hardware examples

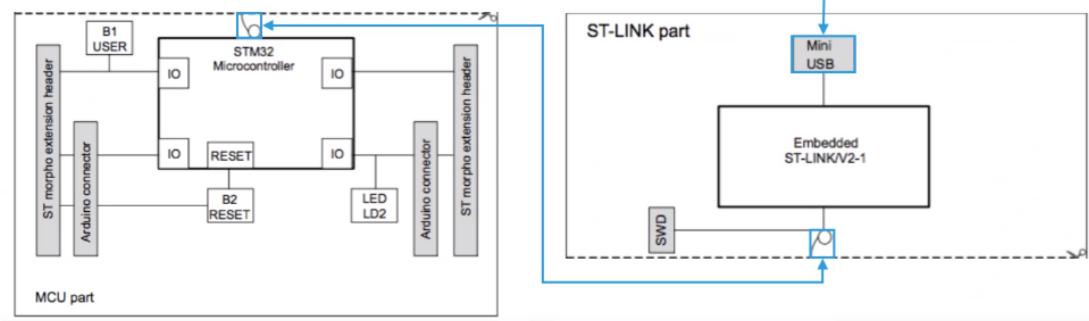
Il nucleo-64 STM32F446 appartiene alla famiglia dei nucleo-64 della ST microelectronics

- Ogni scheda è provvista di un MCU STM32
- La scheda offre una combinazione di performance, consumo ottimizzato di energia e caratteristiche
- supporta la connessione ad Arduino Uno V3
- header ST morpho
- Possibilità di effettuare debug grazie al debugger ST-LINK/V2-1

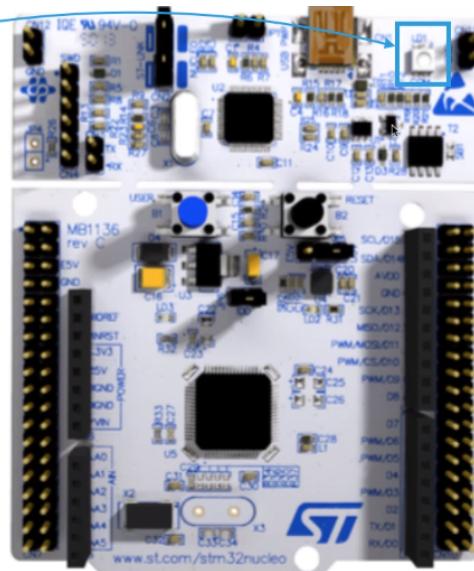


Nella figura sono indicate le componenti del nucleo-64. L'ST-LINK offre all'host una porta virtuale COM, una interfaccia di disco virtuale e un'interfaccia di programmazione in-system

- ▶ The ST-LINK part provides the host with
 - ▶ A Virtual COM Port (UART)
 - ▶ A virtual disk interface (Write-only)
 - ▶ An In-System Programming interface



- ▶ The tricolor LED (green, orange, red) LD1 (COM) provides information about ST-LINK communication status.
- ▶ LD1 default color is red. LD1 turns to green to indicate that communication is in progress between the PC and the ST-LINK/V2-1, with the following setup:
 - ▶ Slow blinking Red/Off: at power-on before USB initialization
 - ▶ Fast blinking Red/Off: after the first correct communication between the PC and ST-LINK/V2-1 (enumeration)
 - ▶ Red LED On: when the initialization between the PC and ST-LINK/V2-1 is complete
 - ▶ Green LED On: after a successful target communication initialization
 - ▶ Blinking Red/Green: during communication with target
 - ▶ Green On: communication finished and successful
 - ▶ Orange On: Communication failure



Essenzialmente il professore fa un listato delle varie componenti della scheda, nelle slide c'è altro, non mi pare il caso di inserire tutte queste immagini

GPIO

Il SoC possiede 8 porte GPIO da 16 pin

- GPIOx x=A..H

Alcune porte sono connette a pin esterni come

- connettori Arduino (CN5,CN6)
- connettori Morpho proprietari (CN7,CN10)

Ambienti di programmazione

Abbiamo molteplici ambienti di programmazione, alcuni proprietari, altri open source e web based come

- ST IDE
- Arduino IDE
- GCC C/C++ toolchain
- Mecrisp-Stellaris Forth Environment

- Mbed WEB IDE (Mbed OS)

Architettura

Il sistema principale consiste in una **matrice di bus AHB multilayer a 32-bit** che interconnette 7 master e 7 slave:

M32F446xx, the main system consists of a **32-bit multilayer AHB bus**

ix that interconnects:

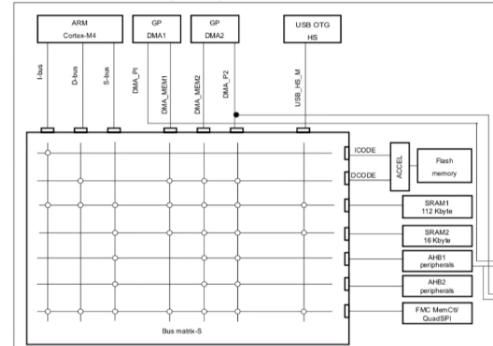
Seven masters:

- ▶ Cortex®-M4 with FPU core I-bus, D-bus and S-bus
- ▶ DMA1 memory bus
- ▶ DMA2 memory bus
- ▶ DMA2 peripheral bus
- ▶ USB OTG HS DMA bus

Seven slaves:

- ▶ Internal Flash memory **ICode** bus
- ▶ Internal Flash memory **DCode** bus
- ▶ Main internal SRAM1 (112 KB)
- ▶ Auxiliary internal SRAM2 (16 KB)
- ▶ AHB1 peripherals including AHB to APB bridges and APB peripherals
- ▶ AHB2 peripherals
- ▶ Flexible Memory Controller FMC / QUADSPI

Figure 1. System architecture for STM32F446xx devices



le SRAM sono molte piccole, in quanto questa macchina non è stata progettata per gestire grossi carichi (multimedia, NAS, server hosting, ecc..)

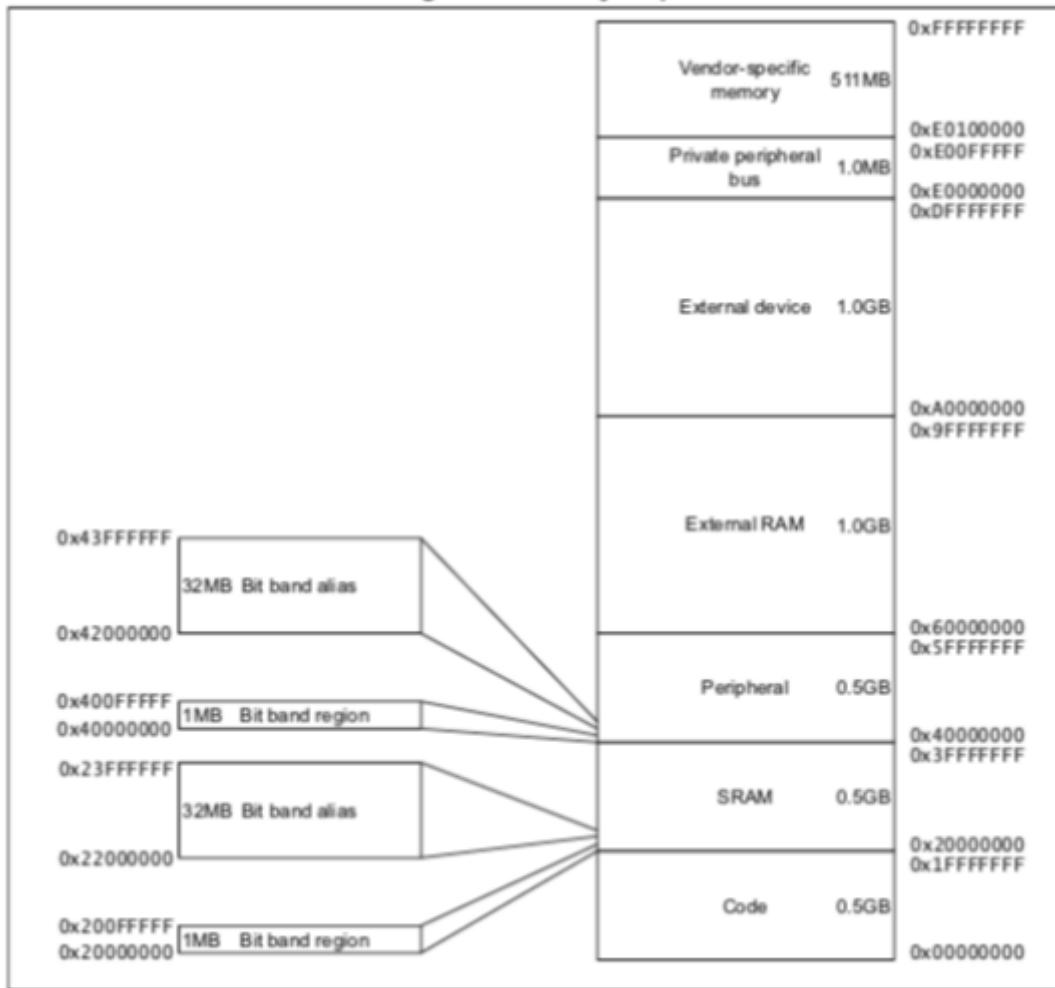
i bus del sistema sono i seguenti:

1. I-bus: connette l'i-bus del Cortex M4 con core FPU al busMatrix. Questo bus è utilizzato dal core per effettuare il fetch delle istruzioni.
2. D-bus: questo bus è utilizzato dal medesimo core (definito sopra) per il "literal load" e il debug.
3. S-bus: questo bus è usato per accedere a dati localizzati in periferiche o in una SRAM
4. DMA memory bus: connette l'interfaccia master DMA al busmatrix; è utilizzata principalmente per trasferimenti da/alla memoria
5. DMA peripheral bus: connette l'interfaccia DMA periferica master al busmatrix.

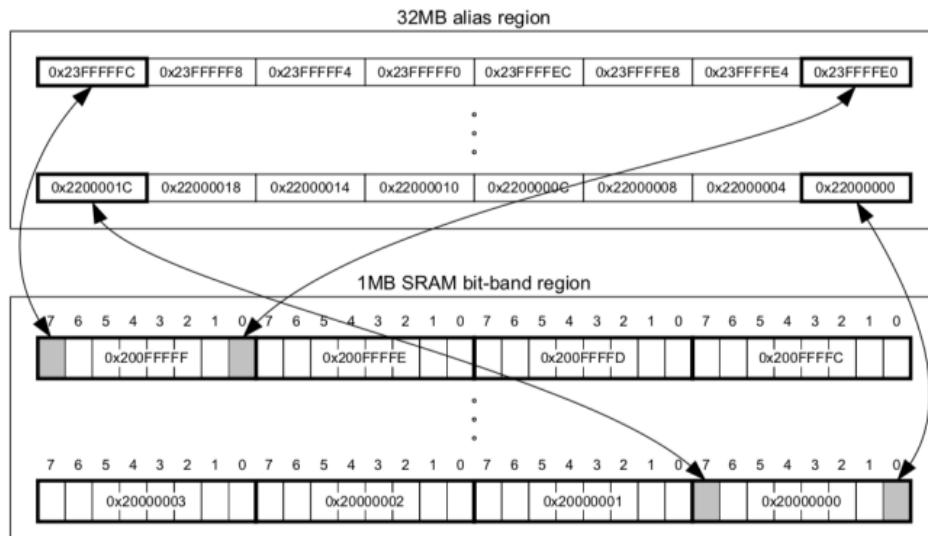
Mappatura della memoria

Il processore ha una mappatura fissa FINO a 4 gb di memoria indirizzabile

Figure 8. Memory map



Lo spazio di memoria è diviso in 8 blocchi principali da 512 MB circa ? (alcune porzioni di memoria sono da 1 GB)



Due regioni da 32 MB mappano per 2 da 1 Mbyte sia per le regioni di alias che per le regioni bit-band

Program memory, data memory, registers and I/O ports

are organized within the same linear 4-Gbyte address space

- ▶ The bytes are coded in memory in Little Endian format. The lowest numbered byte in a word is considered the word's least significant byte and the highest numbered byte the most significant
- ▶ The addressable memory space is divided into 8 main blocks, of 512 Mbytes each.
- ▶ All the memory areas that are not allocated to on-chip memories and peripherals are considered "Reserved"

Boundary address	Peripheral	Bus	Register
0x4002 1C00 - 0x4002 1FFF	GPIOH	AHB1	Section 7.4.11: GPIO register
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		
0x4001 1C00 - 0x4001 5FFF	SAI2	APB2	Section 28.5.10: SAI register
0x4001 1800 - 0x4001 5BFF	SAI1		
0x4001 1400 - 0x4001 4BFF	TIM11		
0x4001 1400 - 0x4001 47FF	TIM10		
0x4001 1400 - 0x4001 43FF	TIM9		
0x4001 3C00 - 0x4001 3FFF	EXTI		
0x4001 3800 - 0x4001 3BFF	SYSCFG		
0x4001 3400 - 0x4001 37FF	SPI4		
0x4001 3000 - 0x4001 33FF	SPI1	APB2	Section 26.7.10: SPI register
0x4001 2C00 - 0x4001 2FFF	SDMMC		
0x4001 2000 - 0x4001 23FF	ADC1 - ADC2 - ADC3		
0x4001 1400 - 0x4001 17FF	USART6		
0x4001 1000 - 0x4001 13FF	USART1		
0x4001 0400 - 0x4001 07FF	TIM8		
0x4001 0000 - 0x4001 03FF	TIM1		

La figura sopra indica lo spiazzamento per ogni periferica

- ▶ The I/O direction of each pin can be configured through a two-bit value in the port MODER register

7.4.1 GPIO port mode register (GPIOx_MODER) (x = A..H)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]	MODER14[1:0]	MODER13[1:0]	MODER12[1:0]	MODER11[1:0]	MODER10[1:0]	MODER9[1:0]	MODER8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]	MODER6[1:0]	MODER5[1:0]	MODER4[1:0]	MODER3[1:0]	MODER2[1:0]	MODER1[1:0]	MODER0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 MODERy[1:0]: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

- 00: Input (reset state)
- 01: General purpose output mode
- 10: Alternate function mode
- 11: Analog mode

il registro MODER regola la "direzione I/O" tramite la combinazione di 2 bit (4 modalità possibili); questo lo rende meno flessibile del Raspberry Pi perchè in quest'ultimo i bit combinabili sono 3 e ci permettono di avere fino a 5 funzioni di selezione

- ▶ The IDR register of a port holds the digital input values of its pins
- ▶ The whole word must be read to read even a single pin value

7.4.5 GPIO port input data register (GPIOx_IDR) (x = A..H)

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 IDRy: Port input data (y = 0..15)

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

il registro IDR possiede i valori dei digital input dei suoi pin

GPIO and Alternate Functions

- When in Alternate Function mode, a GPIO pin can be configured through either the **AFRL (lower 8 pins)** or AFRH (highest 8 pins) register to express a specific function.

7.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..H)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw												

Bits 31:0 **AFRLy**: Alternate function selection for port x bit y (y = 0..7)

These bits are written by software to configure alternate function I/Os

AFRLy selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

Hardware abstraction layers

L'idea è quella di astrarre dall'hardware utilizzando costrutti ad alto livello. Le definizioni HAL possono essere fondamentali per raccogliere informazioni per quanto riguarda i dispositivi in una maniera che possa essere eseguibile e testabile. FORTH è in grado di interagire facilmente con l'hardware e permette di scrivere codice che lo possa "dirigere"

- Let's define a HAL in FORTH using Mecrisp-Stellaris on the Nucleo-64 STM32F446
 - The code begins with GPIO definitions
 - Then GPIO words to interact with the on-board I/O devices (LD2 LED, B1 Button) are defined
 - The HAL can also be used to drive external I/O devices

```
7 \ Each GPIO register set spans $0400 bytes
8 \ See en.DM00135183.pdf page 55
9
10 : GPIO{ ( addr -- ) ;
11 : PORT ( addr -- addr ) DUP CONSTANT $0400 +
12 : }GPIO ( addr -- ) DROP ;
13
14 $40020000
15 GPIO{ PORT GPIOA PORT GPIOB PORT GPIOC
( ... ) }GPIO
16 ( More GPIO port could be added... )
17
18 : REGS{ 0 ;
19 : REG CREATE DUP , OVER + DOES> @ + ;
20 : }REGS 2DROP ;
21
22 $04 REGS{
23   REG MODER REG OTYPER REG OSPEEDR
24   REG PUPDR REG IDR REG ODR
25   REG BSSR REG LCKR REG AFRL
26   REG AFRH
27 }REGS
28
```

```

DELAY  1000 * 0 DO LOOP ;
1BIT  $1 1 ;
2BIT  $3 2 ;
4BIT  $F 4 ;
MASK  ( index mask width -- offset_mask ) ROT * LSHIFT ;
PIN   SWAP 1BIT MASK SWAP ;
MODE  OVER 2BIT MASK SWAP ;
MODE@ MODE @ AND SWAP 2 * RSHIFT ;
MODE!  >R R@ MODE @ SWAP NOT AND ROT 3 AND ROT 2 * LSHIFT OR R> ! ;
BIT  ( mask addr -- addr value mask ) DUP @ ROT ;
SET  ( addr value mask -- ) OR SWAP ! ;
CLEAR ( addr value mask -- ) NOT AND SWAP ! ;
TRUTH ( addr value mask -- value ) AND 0<> NIP ;
OUT@ ODR PIN BIT TRUTH ;
OUT!  ODR PIN BIT SWAP OVER NOT AND >R ROT AND R> OR SWAP ! ;

```

Examples:

5 GPIOA MODE@ .

Shows current mode for pin 5 of GPIO Port A (controlling LED2)

13 GPIOC MODE@ .

Shows current mode for pin 13 of GPIO Port C (connected to Button B1 USER)

1 5 GPIOA MODE!

Sets Output mode (1) for pin 5 of GPIO Port A (LED2)

```

\ 5 GPIOA ODR PIN BIT SET
\ Sets pin 5 of GPIO Port A to high (LED2 turns ON)
\
\ TRUE 5 GPIOA OUT!
\ Sets pin 5 of GPIO Port A to high (LED2 turns ON)
\

: BUTTON  $2000 GPIOC IDR ; \ Same as 13 GPIOC IDR PIN
: RELEASED  BIT TRUTH ;
: PRESSED  RELEASED NOT ;
: ?BUTTON  BUTTON PRESSED ;
: CLICKED  BEGIN 2DUP PRESSED UNTIL BEGIN 2DUP RELEASED UNTIL 2DROP ;
: LED    $20 GPIOA ODR ; \ Same as 5 GPIOA ODR PIN
: ON     ( mask addr -- ) BIT SET ;
: OFF    ( mask addr -- ) BIT CLEAR ;
: BLINK  2DUP OFF 1000 DELAY 2DUP ON 1000 DELAY OFF ;

\ Examples:
\
\ LED ON
\ LED BLINK
\

( Test code: wait for button click then blink the LED, do it n times )
: TEST  ( n -- ) 0 DO  BUTTON CLICKED LED BLINK LOOP ;

\ Configuration:
1 5 GPIOA MODE!

```

Connettività Arduino

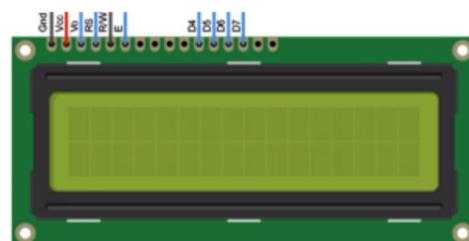
è facile definire in FORTH un semplice layer software per usare gli stessi simboli trovati nelle specifiche delle applicazioni Arduino

```

3 ( Daniele Peri, Università degli Studi
di Palermo, 17-18 )
4
5 \ Must be INCLUDED after Nucleo64-
STM32F446RE-simple-HAL.f
6
7 \ Definitions for a few Arduino pins on
8 \ connectors CN5 and CN9:
9 \
10
11 : D2    10 GPIOA ;
12 : D3    3 GPIOB ;
13 : D4    5 GPIOB ;
14 : D5    4 GPIOB ;
15 : D6    10 GPIOB ;
16 : D7    8 GPIOA ;
17 : D11   7 GPIOA ;
18 : D12   6 GPIOA ;
19 : D13   5 GPIOA ;
20
21 \ Examples:
22 \
23 \ TRUE D13 OUT!
24 \ Sets Arduino D13 pin (connector CN5)
high
25 \ This is another way to turn the LD2
LED on.
26 \

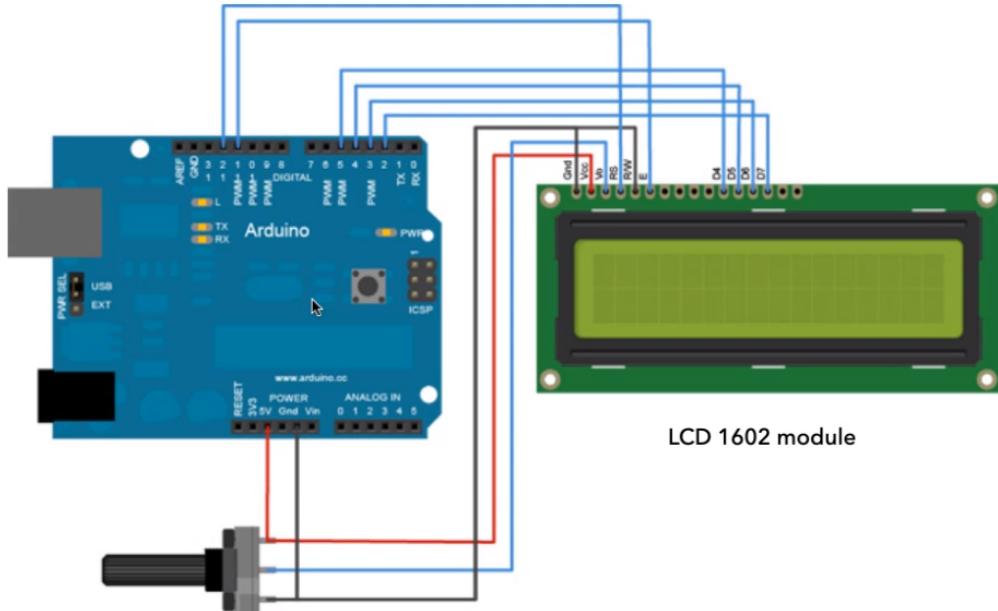
```

- ▶ The LCD 1602 is a display module
 - ▶ 16x2 dot-matrix (5x8 pixels) characters
 - ▶ It is controlled through a parallel interface with
 - ▶ 8-bit/4-bit data bus
 - ▶ 3 control signals
 - ▶ A parallel bus can be set up using a group of GPIOs



LCD 1602 module

Modulo LCD: possiede una matrice 16x2 e può essere controllato con 8bit/4bit bus di dati 3 segnali di controllo. Per evitare la connessione di tutti questi segnali, abbiamo 4 data signals

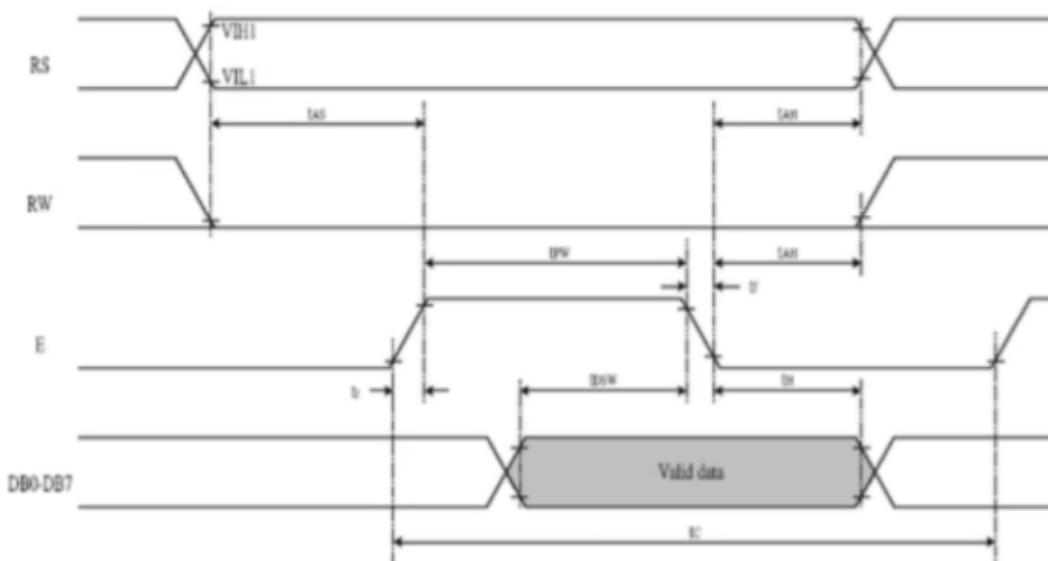


schema di connessione dell'LCD

- ▶ The LCD 1602 module is controlled through instructions (four categories) to:
 - ▶ set display format, data length, scrolling modality
 - ▶ set internal RAM address
 - ▶ perform data transfer from/to internal RAM
 - ▶ access status flags

Instruction	Instruction Code								Description	Description Time (270Hz)	
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write "0H" to DDRAM, and set DDRAM address to "00H" from AC
Return Home	0	0	0	0	0	0	0	0	1	x	Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.
Entry Mode Set	0	0	0	0	0	0	0	1	ID	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.
Display ON/OFF	0	0	0	0	0	0	1	D	C	B	D=1: entire display on C=1: cursor on B=1: cursor position on
Cursor or Display Shift	0	0	0	0	0	0	1	S/C	R/L	x	Set cursor moving and display shift control bit, and the direction, without changing DDRAM data.

Non possiamo scrivere semplicemente scrivere bit nel bus, abbiamo bisogno di "sapere che il BUS sia pronto per farlo" essendo parallelo: l'immagine sotto ci dice effettivamente come funziona la temporizzazione



RW non "scrive" finchè RS non ritorna in "salita". Il Bus "E" (enable) si attiva affinchè i dati possano essere **campionati** nei databus e possano essere mostrati nel display.

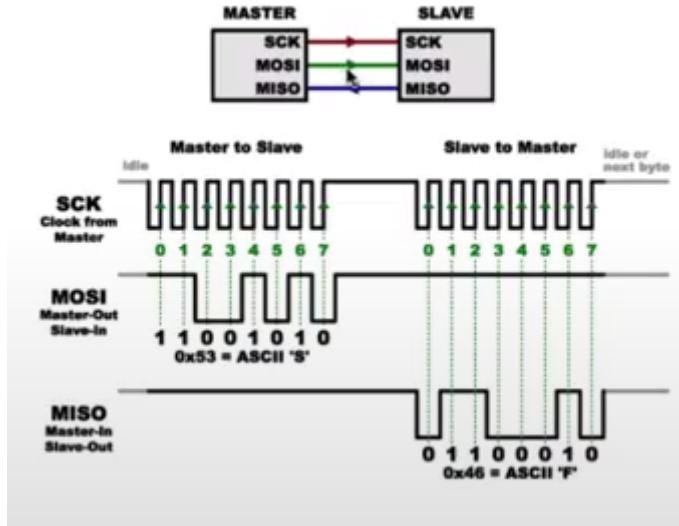
```
1 ( Embedded Systems - Sistemi Embedded - 17873)
2 ( Some code for LCD 1602 )
3 ( Daniele Peri, Università degli Studi di Palermo, 17-18 )
4
5 \ Must be INCLUDED after arduino.f
6
7 : LCDE    D11 ;
8 : LCDRS   D12 ;
9 : LCD7    D2 ;
10 : LCD6   D3 ;
11 : LCD5   D4 ;
12 : LCD4   D5 ;
13
14 1 LCD4 MODE!  1 LCD5 MODE!  1 LCD6 MODE!
15 1 LCD7 MODE!  1 LCDRS MODE!  1 LCDE MODE!
16
17 : BTST AND 0<> ;
18 : LCDREG4H!
19     DUP $80 BTST LCD7 OUT! DUP $40 BTST LCD6 OUT!
20     DUP $20 BTST LCD5 OUT!      $10 BTST LCD4 OUT! ;
21 : LCDREG4H@
22     0 LCD7 OUT@ $80 AND OR  LCD6 OUT@ $40 AND OR
23     LCD5 OUT@ $20 AND OR  LCD4 OUT@ $10 AND OR ;
24 : LCDRSH  -1 LCDRS OUT! ;
25 : LCDRSL  0 LCDRS OUT! ;
26 : LCDEH   -1 LCDE OUT! ;
27 : LCDEL   0 LCDE OUT! ;
28
```

Implementazione HAL in FORTH del display LCD.

Lezione 23/12/2021

Serial Peripheal interconnect

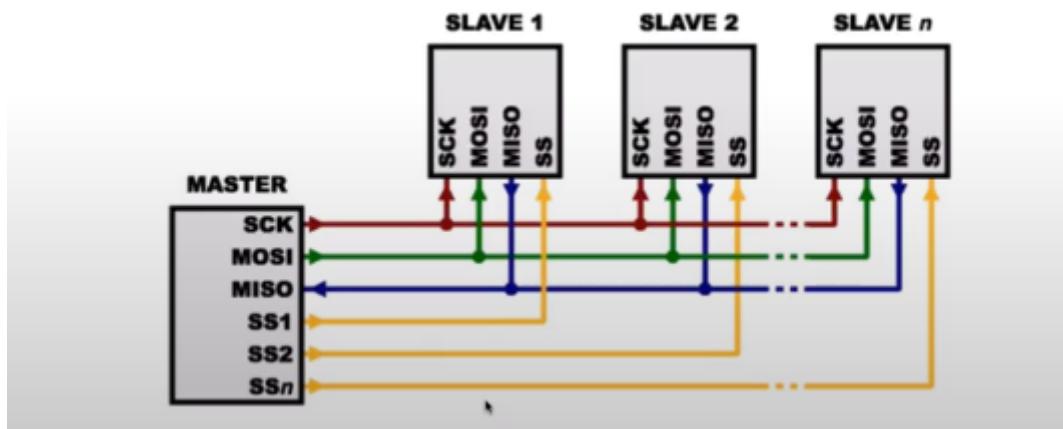
è un'interfaccia seriale che stabilisce connessione tra il "master" e lo "slave".



SPI è un'interfaccia full duplex, quindi le linee di invio e di ricevuta sono separate: i dati possono essere ricevuti allo stesso tempo. Ricordiamo anche che SPI agisce in modalità sincronizzata e il clock SCK fornisce la temporizzazione. Può essere implementato hardware tramite uno shift register o tramite il "bit banging"

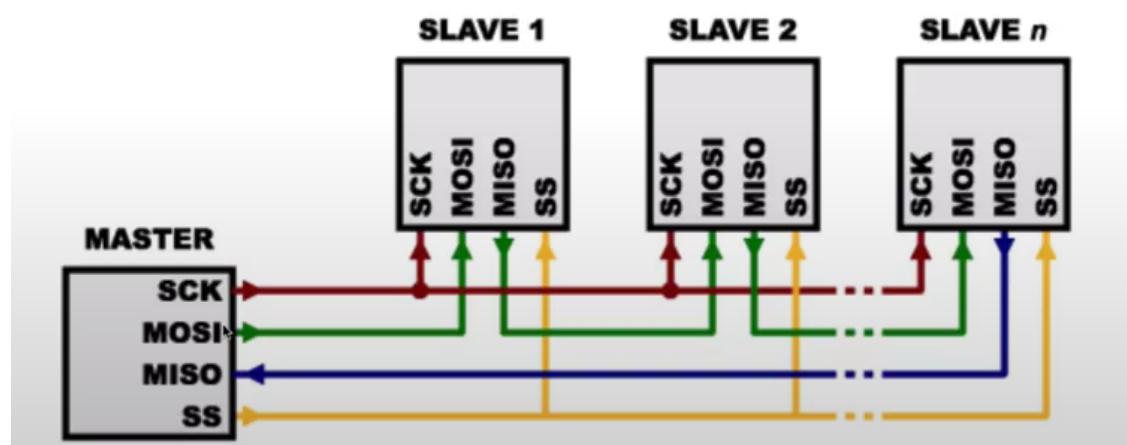


Le modalità di connessione possono variare dal "single slave", effettuando quindi un "slave select" oppure al "multiple slave select"



quindi il master, in questo caso, può comunicare con più slave e non con un singolo come avveniva nel caso precedente

- ▶ Daisy-chaining: a single SS line is shared by all the slaves. Once all the data is sent, the SS line is raised, which causes all the chips to be activated simultaneously (Daisy-chained shift registers)

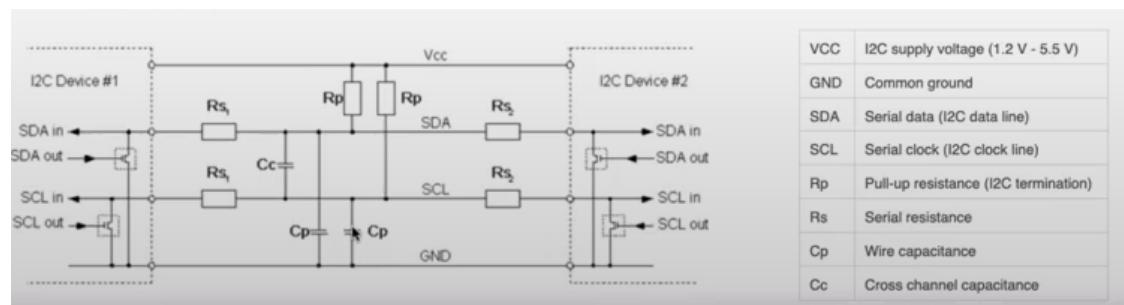


l'SS può anche essere in modalità "daisy chaining", ovvero l'SS è connesso con tutti gli slave e una volta attivata l'SS line, tutti gli slave si attivano (quando abbiamo gli stessi tipi di slave, l'utilizzo del daisy chaining è necessario, ad esempio un display che deve svolgere le stesse funzioni)

I2C BUS

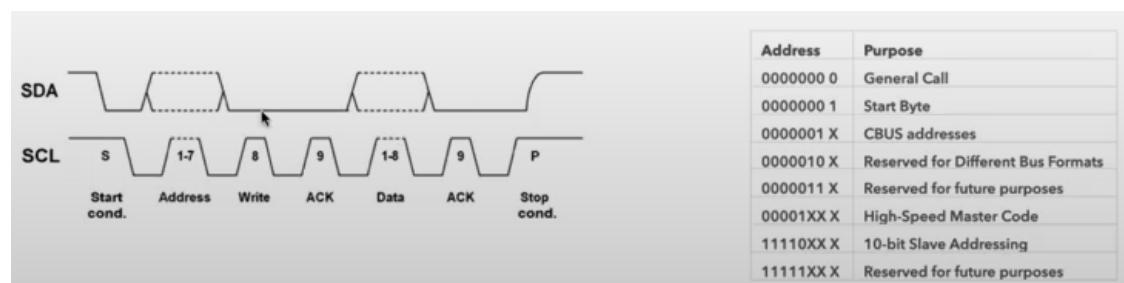
è un'interfaccia "sincrona", "multi-master", "multi-slave" ed orientata allo switching di pacchetti. Questa interfaccia è utilizzata per connettere, nella breve distanza, periferiche IC di velocità minore a processori e microcontrollori. Ogni dispositivo connesso al bus può essere associato ad un singolo ed unico indirizzo software. Può viaggiare a diverse velocità:

- Fastmode (400 kbit/s)
- High speed mode (3.4 Mbit/s)
- Fast mode plus (1 Mhz max frequency)
- Ultra fast mode (5 Mbit/s)



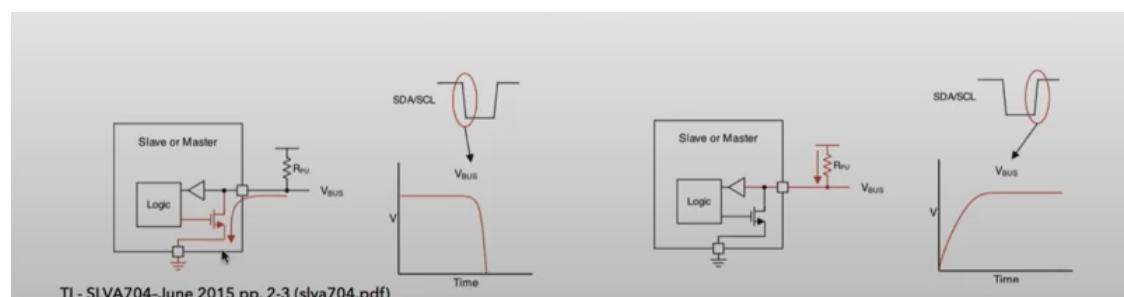
vi sono due linee: SDA e SCL (la prima trasporta i dati, la seconda il clock)

questo tipo di connessione permette di svolgere operazioni concorrenti in più di un master o anche operazioni di clock stretching (gli slave possono rallentare la comunicazione "ritardando" il SCL)



dal punto di vista logico è molto simile a SPI con la differenza che, stavolta, abbiamo una sola linea per il trasferimento di dati

per trasmettere un "low voltage", viene attivato il FET pull-down (la linea viene collegata alla "terra")



per trasmettere un "high voltage", basta rilasciare il bus spegnendo il FET pull-down (la linea viene lasciata sospesa, e il resistore pull-up fa salire la tensione)

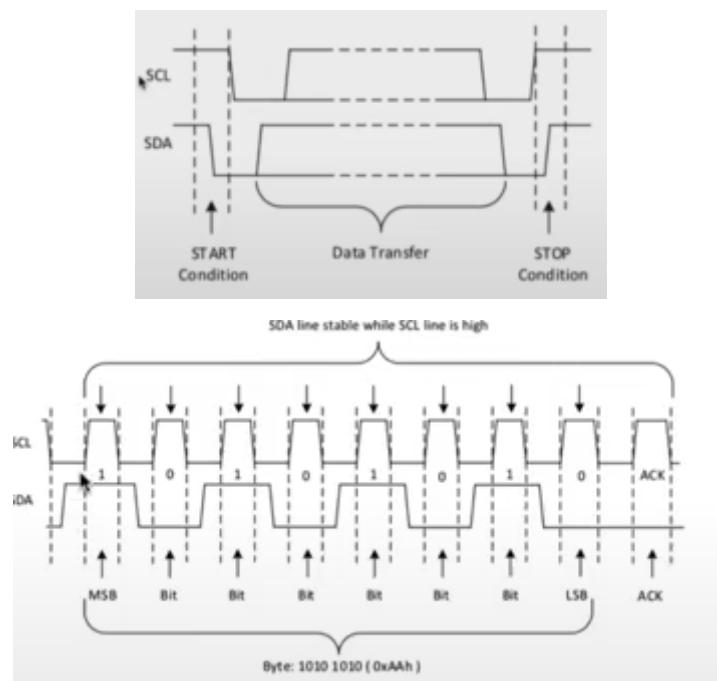
uno slave non dovrebbe trasmettere dati a meno che non sia stato "ordinato" dal master. Ogni dispositivo nel bus i2c ha un indirizzo di device specifico per differenziarli tra altri device che sono nello stesso bus i2c. Molti dispositivi slave richiedono una certa configurazione per impostare il proprio "comportamento" (ciò viene fatto quando il master ha accesso alla mappatura dei registri dello slave)

Procedure di comunicazione

1. Il master invia una condizione START e "indica" lo slave; il master invia dati allo slave e il master conclude il trasferimento con una condizione di STOP (scrittura)
2. Il master invia START allo slave, il Master "invia" l'indirizzo del registro richiesto per leggere lo slave, il master riceve dati dallo slave, il master conclude il trasferimento con una condizione di STOP (lettura)

La comunicazione è inizializzata dal master, come abbiamo detto prima, e conclusa sempre da quest'ultimo

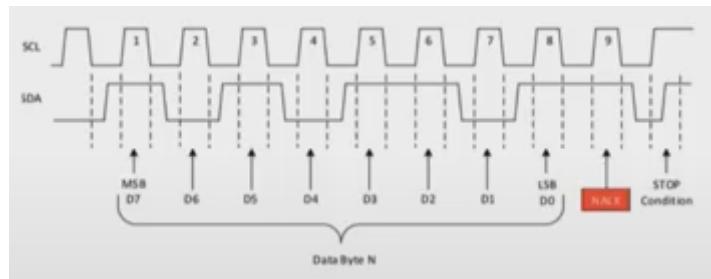
- Una transizione **alto-basso** nella linea SDA mentre l'SCL è impostato in alto, definisce una condizione di **START**
- Una transizione **basso-alto** nella linea SDA mentre l'SCL è alto, definisce una condizione di **STOP**



qualsiasi numero di byte può essere trasferito dal master allo slave tra la condizione START e la condizione STOP. Per inviare un bit ACK, il ricevitore tiene "bassa" la linea SDA durante la low phase del periodo 9, in maniera tale da avere SDA stabilmente in basso durante la high phase del periodo di ACK clock.

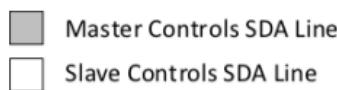
quando la linea SDA rimane alta durante il periodo 9, quest'ultima è interpretata come se fosse un **NACK** (not acknowledge)

- Il ricevitore è impossibilitato a ricevere o trasmettere perchè sta svolgendo altre operazioni e non è pronto ad iniziare la comunicazione con il master
- durante il trasferimento, il ricevitore riceve dati o comandi che non capsce
- durante il trasferimento, il ricevitore non riceve più data byte
- un master-receiver è realizzato leggendo dati e indicando questo allo slave tramite un NACK

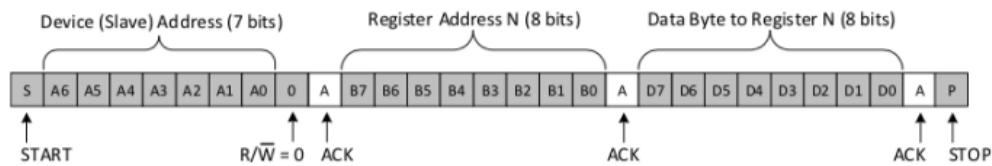


Per scrivere nel bus, il master invia una condizione di START con l'indirizzo dello slave seguito dal bit R/W settato a 0 e poi

1. lo slave invia il bit di ACK
2. il master invia l'indirizzo di registro in cui vuole scrivere
3. lo slave invia un'altro ack
4. il master inizia ad inviare i dati allo slave
5. il master conclude la trasmissione con una condizione di STOP



Write to One Register in a Device

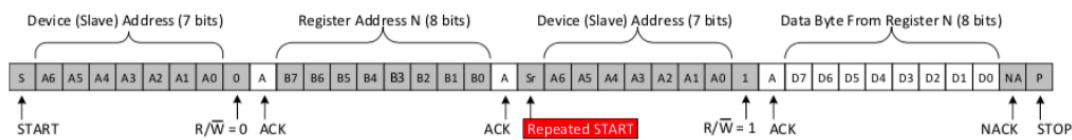


Per la lettura, invece il discorso è quasi identico ma con alcune differenze

1. se lo slave riconosce l'indirizzo del registro, il master invia un'altra condizione di START, stessa volta con l'indirizzo dello slave seguito dal bit R/W impostato a 1
2. se lo slave riconosce la richiesta di lettura, il master rilascia il bus SDA ma continua a trasmettere il clock allo slave. Il master diventa master-receiver e lo slave diventa slave-receiver.
3. Alla fine di ogni byte, il master invia un ACK allo slave, permettendo a quest'ultimo di sapere se è pronto per ricevere altri dati. Quando il master riceve tutti i byte desiderati, viene inviato un NACK



Read From One Register in a Device



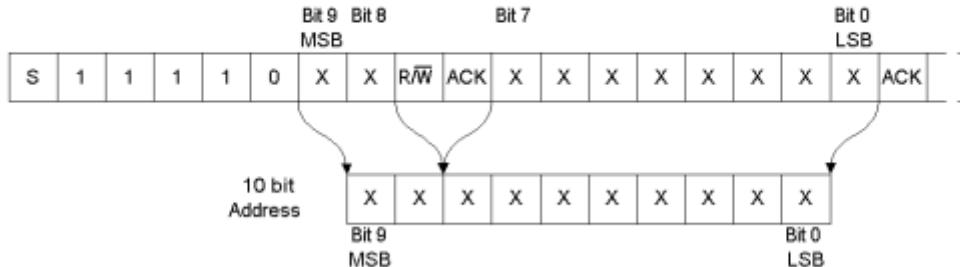
Slow Start byte

è una sorta di metodo per "svegliare" gli slave connessi a questo bus.

- il master trasmette la condizione di START, seguita dal byte di start, un ack "fasullo" e una condizione di start ripetuta. Il microcontrollore in ascolto deve individuare soltanto uno dei sette zero nell'SDA per individuare una trasmissione I2C e lo si può fare con un polling rate basso; nel momento in cui il controller individua (o rileva) il basso rateo di trasferimento, può switchare ad un polling rate più alto in maniera tale da abilitare il trasferimento I2C

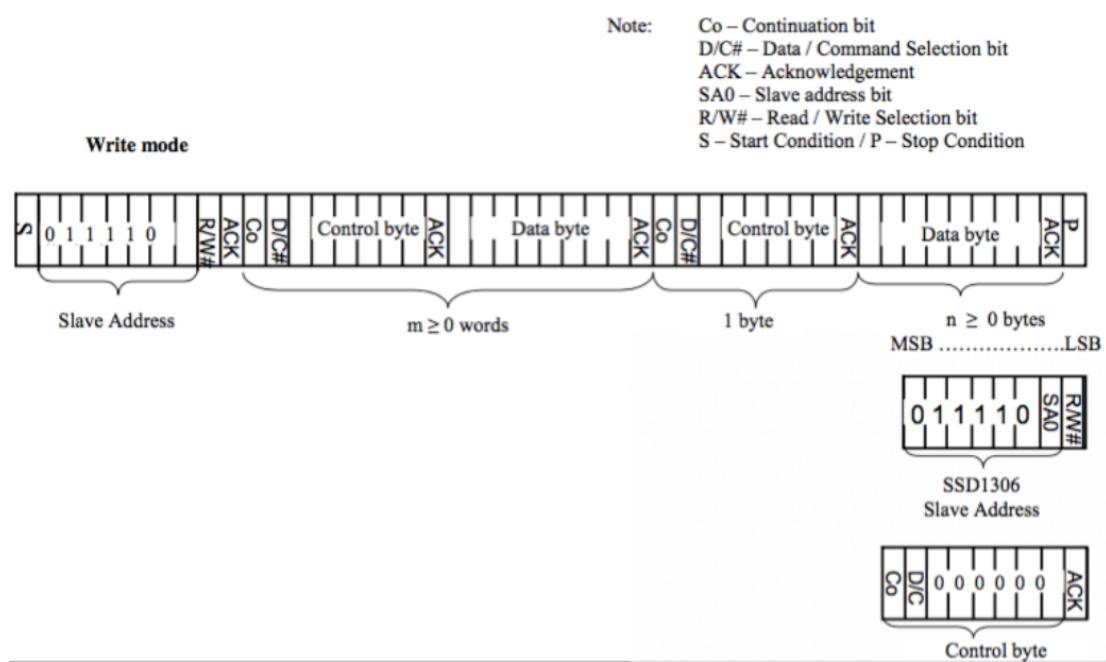
Indirizzamento a 10 bit

Per evitare collisioni d'indirizzo, è stato introdotto un nuovo schema di 10 bit. Questo miglioramento può essere "mescolato" con l'indirizzamento a 7 bit e aumenta il range di indirizzi disponibili di 10 volte.



Dopo la condizione di start, una sequenza di bit (precisamente '11110') introduce lo schema d'indirizzamento a 10 bit

SSD1306 OLED display



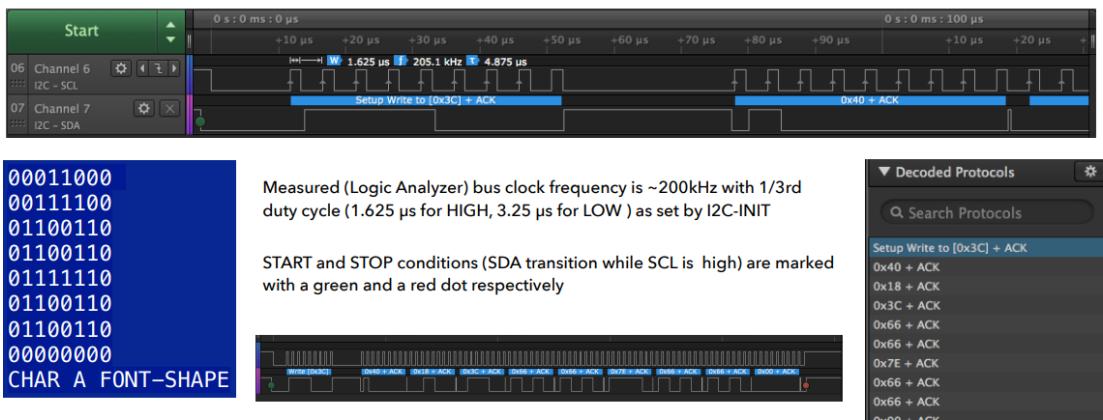
la procedura di comunicazione è la seguente:

- si inizia sempre con il bit di start, seguito dall'indirizzo dello slave e dal bit R/W (settato a 0 in modalità Read, altrimenti 1 se si parla di write).
- Dopo l'invio dell'ACK, si ha un bit di continuazione e un'altro detto "Data/Command selection bit" necessario a stabilire se leggere soltanto il Control Byte o il Data bye
 - ogni "chiusura" è seguita sempre da un ACK

Displaying an 8x8 pixel character

- Command Data Write (\$40) is sent followed by 8 bytes, one for each pixel row

CHAR A C64Font PUTUDG ok.

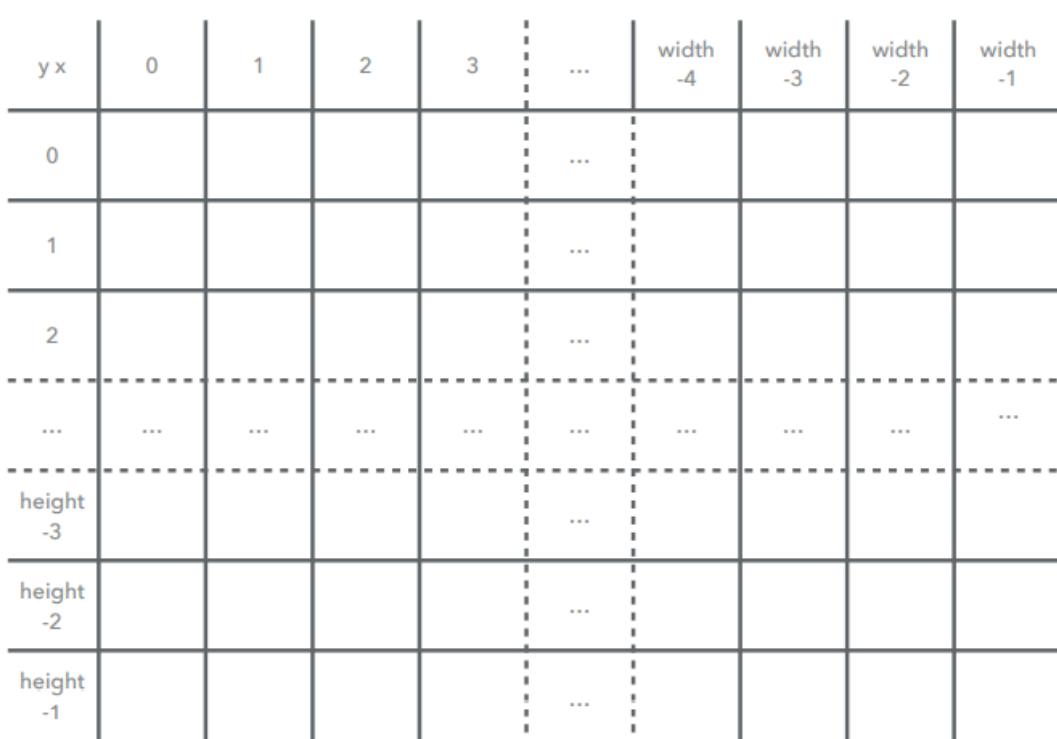


SSD1306 OLED Display Datasheet - (SSD1306 Datasheet for 096 OLED.pdf) page 33 - i2c-dip8.f SSD1306-dip-4.f fonts.f

Questo comando "scrive" A nel display.

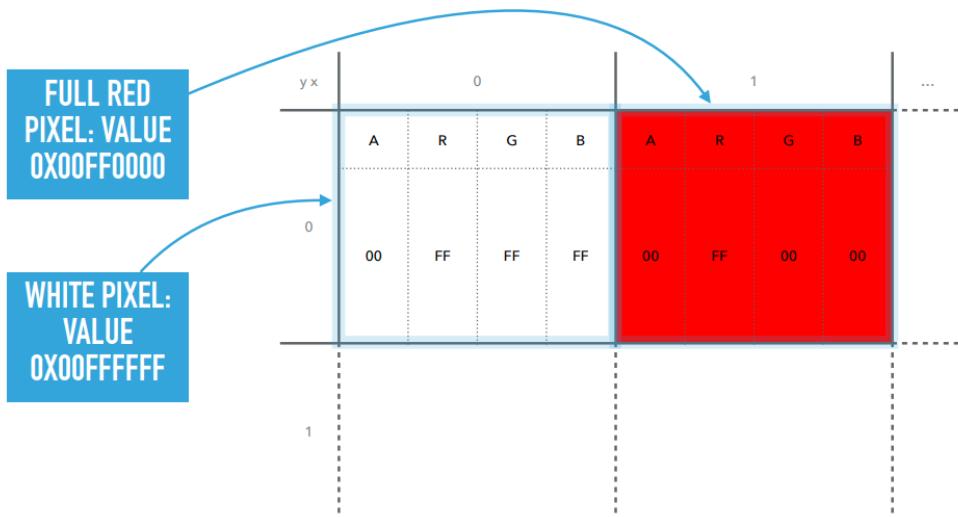
RPI HDMI display

questo display è generato dal Videocore; non è nient'altro che una matrice di pixel width*height. I parametri width ed height possono essere scelti e definisco la risoluzione spaziale del display



Ogni pixel può essere, a livello di colore, codificato dai bit presenti in color_depth e ciò varia in base al tipo di codifica.

Se si utilizzano 32 bit per pixel, il colore è codificato utilizzando un sistema ARGB (8 bit per ogni componente, dove A rappresenta l'alpha channel ed è utilizzato per indicare il livello di trasparenza)



Le slide finali sono autoesplicative