Guida alla preparazione all'esame orale di Embedded System

Sommario

Guida alla preparazione all'esame orale di Embedded System 1
COMUNICAZIONE 3
Domanda1 : Il Bus I2C e caratteristiche. SLIDE3 3
Domanda2 : Indirizzamento a 10 bit. SLIDE3 4
Domanda21 : **Lo Slow Star byte** 4
Domanda16 : Bus parallelo. 5
Domanda22 : Bus SPI. SLIDE3 5
Domanda14 : Il Frame buffer. 6
Domanda5 : GPIO. SLIDE3 6
Domanda38 : Registri PI. 7
Domanda27 : Pull up e down 7
ARCHITETTURE 7
Domanda6 : L'architettura Harvard. SLIDE1 7
Domanda7 : Architettura RISC SLIDE1 8
Domanda40 : Architettura ARM SLIDE1 8
Domanda8 : Architettura MIPs SLIDE1 8
Domanda3 : Processori DSP. SLIDE1 8
Domanda36 : ISA. 9
Domanda41 : Differenza tra parallelismo e concorrenza. 9
Domanda42 : Cosa è un Tap. 9
Domanda32 : Hazard. 10
Domanda23 : Timing. 10
Domanda24 : Processore vettoriale. 10
Domanda33 : Architettura multi core. 11
Domanda31 : Filtro FIR (Finite impulse response) 11
Domanda3 : Buffer circolare 12
Domanda19 : Processori Embbedded vs General Purpose (tempo medio). 12
Domanda13 : Rappresentazione formato in virgola fissa. 13
ARM 14
Domanda15 : I registri ARM. 14
Domanda38 : Program Counter. 14
Domanda51 : Status Register. 14
Domanda20 : Il Link Register. 15

Domanda18 : Gestione dello Stack. 15
Domanda34 : Barrel Shifter 16
Domanda12 : ARM Data Processing Instructions e Format. SLIDEARM 16
Domanda26 : Formato istruzione ARM. SLIDEARM 16
Domanda37 : Per codificare un riferimento ad un registro quanti bit servono? 18
Domanda4 : Istruzioni di somma in ARM. SLIDEARM 18
Domanda28 : Arm moltiplicazione con e senza MUL. 18
Domanda30 : Salto condizionato. 19
Domanda9 : La pipeline ARM. SLIDEARM 19
Domanda43 : Problema dell'allineamento degli indirizzi in ARM. ? 20
Domanda29 : Interruzione software (SWI) 20
Domanda10 : Interlocks. SLIDE2 20
Domanda11 : Branch e Branch with Link e BX exchange 21
Domanda17 : Load/Store e Multiple. 23
Domanda25 : Caratteristiche del secondo operando in ARM. 24
Domanda25 : Preindicizzazione e postinticizzazione in ARM. 25
HARDWARE 26
Domanda40 : LCD1602. 26
Domanda45 : KEPAY PROGETTO. 27
Domanda46 : FRAMEBUFFER PROGETTO. 28
Domanda47 : I2C PROGETTO. 31
Domanda48 : TIMER PROGETTO. 32
FORTH 33
Domanda44 : Comportamento runtime in forth : run-time behavior. 33
Domanda42 : Execution Token – esecuzione vettoriale in forth. 34
Domanda50 : Variabili, costanti e array in Forth. 35
NUCLEO 36
Domanda38 : NUCLEO64. 36
Domanda35 : Bit Banding. 37
Domanda39 : GPIO del NUCLEO64. 37
PRESENTAZIONE PROGETTO 40
Domanda49 : Introduzione. 40

COMUNICAZIONE

Domanda1 : Il Bus I2C e caratteristiche. SLIDE3

Il bus I2C è un canale di comunicazione seriale sincrono, multi master, multi slave, con comunicazione a pacchetto e terminazione singola.

È un canale di comunicazione bidirezionale (fullduplex) e condiviso.

Ciascun dispositivo connesso al bus è identificato tramite un indirizzo, e il clock è scandito dal Master.

È dotato di un meccanismo di controllo delle collisioni e non necessita di un rigoroso bit-rate.

È utilizzato per connessioni tra circuiti integrati a breve distanza e a velocità che vanno dai 400 kbit/s ai 5 Mbit/s.

Il trasporto dei dati avviene tramite due collegamenti: SDA Serial Data e SCL Serial Clock.

Sfrutta le connessioni OPEN-DRAIN che consentono di supportare più master sul canale condiviso, l'indirizzamento di più slave e consente l'allungamento del clock in caso di slave più lenti che tengono giù SCL.

Funzionamento:

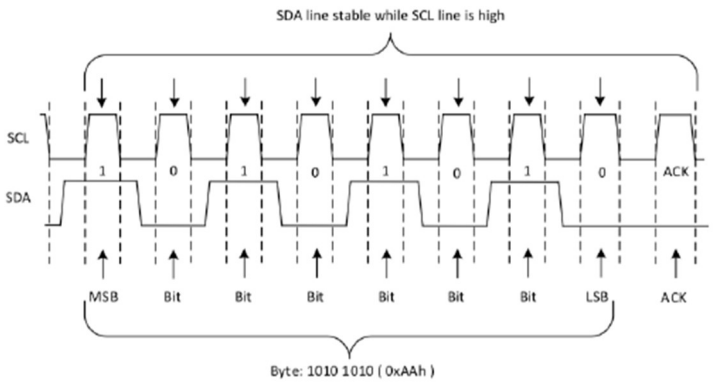
In trasmissione: il livello logico basso viene attivato dal FET di pull-down cortocircuitato a massa. Per il livello logico alto viene spento il FET pull-down e la linea viene rilasciata flottante sul resistore di pull-up che porta la tensione a salire.

Lo slave non può trasmettere dati a meno che non venga etichettato come master. Ogni driver connesso al bus ha un codice univoco identificativo al proprio interno: due cifre in Hex.

Per l'invio dati il Master invia sul canale la START CONDITION che è una transizione **da alto a basso** sulla linea SDA mentre sulla linea SCL il segnale è alto. Quindi trasmette il dato e completa la trasmissione con la STOP CONDITION ovvero la transizione **da basso a alto** sulla linea SDA mentre sulla line SCL il segnale è alto.

Esiste anche la condizione di START RIPETUTA che è equivalente ad aprire, chiudere e riaprire la connessione, utile quando il Master vuole ritrasmettere senza perdere il controllo del bus.

Un bit di dati viene trasferito ad ogni impulso di clock della SCL. Ogni byte di dati è composto da otto bit trasmessi sulla linea SDA. Vengono trasmessi prima i bit più significativi (MSB).



Tra la Start e la Stop condition si può trasferire qualsiasi numero di byte.

Alla trasmissione di ciascun byte il ricevitore trasmette un ACK non appena il trasmettitore avrà rilasciato la linea SDA. Per fare ***ciò tira giu la linea SDA durante il nono ciclo di clock per confermare la ricezione***. Se invece resta alto equivale al trasmettere un NACK ovvero l'errata ricezione.

Ovviamente sono molteplici le condizioni che portano al NACK come ad esempio: Il ricevitore non è in grado di ricevere o trasmettere, il destinatario non può ricevere più byte di dati, il master ha finito di leggere i dati e lo indica allo slave tramite un NACK oppure il ricevitore riceve dati o comandi che non comprende.

In dettaglio per **iniziare a scrivere** sul bus, il master invia una condizione di avvio con l'indirizzo dello slave seguito dal bit R/W impostato a 0; quindi lo slave invia l'ACK di conferma; Il master invia l'indirizzo di registro del registro a cui desidera scrivere; lo slave conferma; a questo punto inizia la trasmissione dei dati che termina con la condizione di Stop.

Per leggere i dati sul bus il master invia la condizione di START seguita dall'indirizzo dello slave con il bit R/W impostato a 0 e dall'indirizzo del registro da cui vuole leggere; se lo slave riconosce l'indirizzo del registro, il master invia nuovamente la Start Condition (START ripetuto), con il bit R/W impostato ad 1; se lo slave conferma la richiesta il master rilascia SDA e continua ad fornire il clock allo slave: a questo punto Il master diventa il master-ricevitore e lo slave lo slave-trasmettitore. Ad ogni byte ricevuto il master invia l'ACK e al termine della ricezione invia un NACK e quindi la Stop condition.

Domanda2 : Indirizzamento a 10 bit. SLIDE3

Al fine di evitare conflitti di indirizzi, a causa della gamma limitata degli indirizzi a 7 bit, è stato introdotto un nuovo schema di indirizzamento a 10 bit. Questo miglioramento può essere combinato con l'indirizzamento a 7 bit e aumenta l'intervallo di indirizzi disponibile di circa dieci volte. Dopo la condizione di avvio, un "**11110**" iniziale introduce lo schema di indirizzamento a 10 bit. Gli ultimi due bit di indirizzo del primo byte concatenati con gli otto bit del secondo byte dell'intero indirizzo a 10 bit. I dispositivi che utilizzano solo l'indirizzamento a 7 bit ignorano semplicemente i messaggi con '11110' iniziale.

Domanda21 : Lo Slow Star byte

I microcontrollori non dotati di circuiti di controllo I2C devono osservare le linee I2C in modo permanente per rilevare una trasmissione I2C (polling). Per ridurre lo spreco di potenza della CPU, è possibile stabilire un trasferimento I2C con un metodo di arbitrato più lento. Quando il master trasmette la condizione di START, seguita dal byte di avvio ('00000001'), un impulso di conferma fittizio e una condizione di avvio ripetuta.

Indirizzo	Scopo
0000000 0	Chiamata generale
0000000 1	Byte di inizio
0000001 X	indirizzi CBUS
0000010 X	Riservato a diversi formati bus
0000011 X	Riservato per scopi futuri
00001XX X	Codice Master Alta Velocità
11110XX X	Indirizzamento slave a 10 bit
11111XX X	Riservato per scopi futuri

Il microcontrollore di osservazione deve rilevare solo uno dei sette zeri su SDA per rilevare una trasmissione I2C. Questo può essere fatto con un tasso di polling relativamente lento. Non appena il controller rileva che l'SDA è basso, può passare a una velocità di polling più elevata per attendere la condizione di avvio ripetuto e il successivo trasferimento I2C.

Al termine del trasferimento, è possibile tornare alla velocità di polling lenta per il risparmio energetico della CPU e al fine di rilevare la successiva trasmissione.

Domanda16 : Bus parallelo.

La caratteristica peculiare della trasmissione parallela è che vengono utilizzati più conduttori per trasmettere simultaneamente informazioni: un cavo che effettua una trasmissione parallela a n bit è formato da almeno n conduttori separati. Ogni linea accende o spegne uno specifico bit. Nella realtà il cavo sarà dotato quasi sicuramente di un cavo aggiuntivo per la massa e anche di altri cavi di controllo come quello di clock. Rispetto alla trasmissione seriale la trasmissione parallela risulta avere prestazioni più elevate in termini di velocità di trasmissione a parità di frequenza, ma ovviamente sarà anche più ingombrante e costosa.

Domanda22 : Bus SPI. SLIDE3

Il Serial Pheriferal Interconnect è un tipo di collegamento che mette in comunicazione le cpu con altri dispositivi, sia on board che off board, con pochi segnali, anche solo due.

I dati sono inviati un bit alla volta da un TX ad un RX. Sono collegamenti molto economici rispetto ai collegamenti paralleli, ma a parità di clock sono più lenti. Ma molto spesso però sono sufficienti a gestire comunicazioni con applicazioni vincolate da fattori esterni (come limite di bitrate o unità di tempo macroscopiche di attuatori, tempo di lettura o scrittura su una memoria esterna o caratteristiche del canale).

I bit sono organizzati in **Frame** composti anche da singoli byte. Nel frame è possibile aggiungere bit per il controllo errori. Questa tecnica di *framing* semplifica la sincronizzazione. Inoltre hardware specializzato può alleggerire il lavoro della cpu nella trattazione del Frame (assemblaggio, disassemblaggio, crc ..)

Le trasmissioni seriali possono essere di due tipi:

Asincrone: La trasmissione può iniziare in qualunque momento. TX e RX hanno ognuno il proprio clock che viene preconfigurato per il tick alla stessa velocità (es 115200). È richiesta solo la sincronizzazione all'inizio della trasmissione. La dimensione del frame è nota sia a TX e sia a RX.

Sincrone: La trasmissione è sincronizzata da un **clock condiviso**. Non è necessaria la sincronizzazione iniziale, ma serve una linea in più per il Clock.

Trasmissione Asincrona: Es UART (TX e RX-clk) La linea dati viene mantenuta ALTA quando il collegamento è inattivo. Quando TX vuole trasmettere un frame, invia lo **Start bit (BASSO).** Il SYNC LOGIC in RX riconosce ALTO-BASSO come condizione di inizio e avvia il proprio clock.

Ad ogni impulso di TX CLOCK, il *serializzatore* in TX **trasmette** un bit del frame, contenuto nel PARALLEL IN REGISTER, attraverso la linea SERIAL DATA. A ciascun fronte di salita di RX CLOCK, il deserializzatore in RX preleva un bit da SERIAL DATA e lo accoda nel registro PARALLEL OUT REGISTER.

Alla fine del frame il TX invia il **bit di Stop (HIGH)** e RX riconosce la fine della trasmissione. La linea SERIAL DATA viene mantenuta ALTA fino al frame successivo.

Trasmissione Sincrona: Es USART (TX , RX , CLK) Il segnale di Clock è esplicito e condiviso da TX e RX. Il SERIAL DATA e il CLOCK vengono mantenuti ALTI quando il collegamento è inattivo. Quando TX vuole trasmettere avvia il proprio clock. Semplicemente ad ogni fronte di discesa del clock il TX trasmette un bit del frame su la SERIAL DATA.
Ad ogni fronte di salita del clock, il deserializzatore in RX esegue il campionamento a bit che accoda al PARALLEL OUT REGISTER. Le linee SERIAL DATA e CLOCK vengono mantenute ALTE fino al frame successivo.

Esempi di SPI: UART(Asincrono), USART(Sincrono), IrDA, Modbus…

Autobus: Modbus (RS-485), Circuito interintegrato bus I²C (Philips,NXP), 1-Wire® (Dallas Semiconductor), CAN (Controller Area Network) per applicazioni automotive.

UART (universale) servono 2 segnali TX ed RX del trasmittente sono collegati a RX e TX del ricevente. Le velocità in bit variano da poche centinaia a pochi milioni al secondo.

Esistono standard elettrici come RS-232 e RS-485. (Tensioni da 3 a 15V) Questi collegamenti sono chiamati Transistor-Transistor Logic TTL

SPI è "full duplex" (invio separato e linee di ricezione) così, facoltativamente, i dati possono essere contemporaneamente trasmessi e ricevuti.

Il campionamento può essere implementato semplicemente in hardware con uno shift register o tramite bit banging.

In termini di collegamento, quello seriale può essere di tre tipi: a **singolo Slave**, **MultiSlave con lo Slave Select** (SS ogni slave è indirizzato indipendentemente) o **Multi slave con Daisy-Chaning** (a Margherita ovvero un'unica linea SS è condivisa da tutti gli slave. Una volta inviati tutti i dati, viene sollevata la linea SS, che provoca l'attivazione simultanea di tutti i chip (registri a scorrimento a catena).

Domanda14 : Il Frame buffer.

(Il display Pi HDMI è generato dal Videocore.) Un pixel (picture element) si trova attraverso le sue coordinate xy (da (0,0) a (-1,-1)). Il colore di un ***pixel è controllato da un valore codificato in bit color_depth***. Quando si utilizza 32 bit per pixel (bpp), il colore del pixel è codificato come valore **ARGB : 8 bit per ciascuno dei componenti red, green, blu colors 8 bit per il canale alfa utilizzato per gli effetti di trasparenza** .

I valori dei pixel sono memorizzati in un framebuffer, che è una regione di memoria condivisa dal VC e dalla CPU. Il framebuffer è organizzato come la riga principale della matrice di color_depth.

I valori dei pixel vengono compressi uno dopo l'altro in una matrice lineare di valori di bit di color_depth. Il primo elemento dell'array controlla il pixel in alto a sinistra. In generale, il colore del pixel x,y è controllato dall'i-esimo valore nell'array lineare: i = width*y+x.

Ad esempio il pijFORTHos-se configura il display HDMI per 1024x768x32bpp all'avvio e il colore del pixel x,y è controllato dal valore ARGB nella parola all'offset del byte 4*(1024*y+x) dall'indirizzo di base del framebuffer (per il primo pixel bianco 00FFFFFF 3E8FA000 !).

Domanda5 : GPIO. SLIDE3

I General Purpose I/O (GPIO) consentono il collegamento di dispositivi come microprocessori e microcontrollori con altre periferiche. Un GPIO oltre all'ingresso e all'uscita digitali, può essere commutato anche su funzioni alternative (AFs) o gestire periferiche interne.

Il Digital Input/Output è gestito dai bit dei Pin Level Register che sono dei registri che servono a pilotare il comportamento di questi Pin.

Solitamente i Microcontrollori SoC embedded hanno moltissimi GPIO.

La selezione delle funzioni per i pin GPIO è gestita tramite **una matrice di routing configurabile** tramite registri. Ogni configurazione di bit di questi registri di selezione abilita rispettivamente l'input, output, o le funzioni alternative: ad esempio il PI ha **6 funzioni alternative per 54 GPIO**. Ovviamente alcune GPIO sono

riservate per il collegamento del SoC con l'hardware a bordo, mentre i 40 pin dell'Header contengono parecchi GPIO disponibili per il collegamento con hardware esterno.

Domanda38 : Registri PI.

Per General Purpose I/O i registri GPLEVn sono usati per leggere il livello dei singoli pin e per configurarli come ingresso digitale. Un circuito esterno collegato a un pin come ad esempio un pulsante, genera un corto (livello logico 0) se viene premuto e fa risalire la tensione VCC se viene rilasciato (livello logico 1).

Nel circuito del PI una resistenza di pull-up assicura che un pin sia dotato di una connessione stabile a Vcc o GND.

È possibile attivare resistori pull-up o pull-down interni per ciascun pin GPIO:

Sull'RPi 3B+ viene utilizzata una procedura che coinvolge i registri GPPUD e GPPUDCLKn per abilitare la stessa modalità (pull up, pull-down, no-pull) per un set di pin contemporaneamente.

Mentre sull'RPi 4B+ la modalità di input di ogni pin è controllato, inoltre, da un valore a 2 bit in uno dei GPIO_PUP_PDN_CNTRL_REGn registri.

La CPU può essere sollevata da gran parte dell'onere di attendere le modifiche desiderate negli ingressi digitali programmando i circuiti GPIO per rilevare un cambio di livello o eventi di fronte per ciascun pin:

I registri GPEDSn registrano gli **eventi rilevati** per ciascun pin. Inoltre questi possono rilevare autonomamente gli eventi:

Fronti di salita (GPRENn), **Fronti di discesa** (GPFENn), **Alto livello** (GPHENn), **Basso livello** (GPLENn), **Fronti di salita asincroni** (GPARENn), **Fronti di discesa asincroni** (GPAFENn).

Domanda27 : Pull up e down

Entrambe le resistenze di pull-up e pull-down sono usate nei circuiti logici elettronici per garantire che gli ingressi di un sistema logico stabilito siano a livelli logici previsti se i dispositivi esterni sono scollegati o ad alta impedenza. L'idea di una tale resistenza è strettamente correlata al concetto di logica tri-state. Questa debole resistenza "tira" la tensione del filo senza tensione di origine, ed è collegata verso il suo livello di tensione di origine quando gli altri componenti della linea sono inattivi (come ad esempio i 5V in caso di un resistore di pull-up, o basso 0V come nel caso della resistenza di pull-down).

ARCHITETTURE

Domanda6 : L'architettura Harvard. SLIDE1

L'architettura Harvard, in informatica, è un tipo di architettura hardware per computer digitali in cui vi è separazione tra la memoria contenente i dati e quella contenente le istruzioni. Il termine inizialmente indicava l'architettura del computer Harvard Mark I, un computer basato su relè che memorizzava le istruzioni su un nastro perforato mentre i dati venivano memorizzati in un contatore elettromeccanico a 23 cifre. Questa macchina non era dotata di un'unità di immagazzinamento dei dati, i quali erano interamente memorizzati dalla CPU e il loro caricamento e salvataggio era un processo eseguito in modo manuale agendo sui contatori. Quindi **In un'architettura Harvard le memorie per i dati, e per le istruzioni possono essere anche differenti**, con tecnologia d'implementazione e timing diversi, in particolare in alcuni sistemi l'ampiezza degli indirizzi o la larghezza di parola delle istruzioni è superiore a quella dei dati; oppure in altri sistemi i programmi sono memorizzati in una memoria a sola lettura (ROM) mentre i dati transitano in una memoria a lettura-scrittura (RAM).

Oggi, **la maggior parte dei processori implementa percorsi di segnale separati per motivi di prestazioni**, **ma in realtà implementano un'architettura Harvard modificata**, quindi possono supportare attività come il caricamento di un programma dalla memoria del disco come se fossero dati ed eseguirlo.

Domanda7 : Architettura RISC SLIDE1

Reduced Instruction Set Computer (in acronimo RISC), nell'elettronica digitale, indica un'idea di progettazione di architetture per microprocessori che predilige lo sviluppo di un'architettura semplice e lineare. Questa semplicità di progettazione permette di realizzare microprocessori in grado di eseguire il set di istruzioni in tempi minori rispetto a una architettura CISC. I più comuni processori RISC sono AVR, PIC, **ARM**, DEC Alpha, PA-RISC, SPARC, **MIPS**, RISC-V, POWER e PowerPC.

Domanda40 : Architettura ARM SLIDE1

L'architettura ARM (precedentemente Advanced RISC Machine, prima ancora Acorn RISC Machine), indica una famiglia di microprocessori RISC a 32-bit e 64-bit sviluppata da ARM Holdings e utilizzata in una moltitudine di sistemi embedded. Grazie alle sue caratteristiche di basso consumo elettrico, rapportato alle prestazioni, l'architettura ARM domina il settore dei dispositivi mobili dove il risparmio energetico delle batterie è fondamentale.

Domanda8 : Architettura MIPs SLIDE1

L'architettura MIPS (acronimo dell'inglese microprocessor without interlocked pipeline stages) è un'architettura informatica per microprocessori RISC sviluppata dalla MIPS Computer Systems Inc. (oggi MIPS Technologies Inc.). Il disegno dell'architettura e del set di istruzioni è semplice e lineare, spesso utilizzato come caso di studio nei corsi universitari indirizzati allo studio delle architetture dei processori; tale architettura ha influenzato le architetture di molti altri processori RISC tra i quali si segnala la famiglia DEC Alpha: è utilizzata ad es. nel campo dei computer SGI (grafica 3d in pipeline), ed ha trovato grossa diffusione nell'ambito dei sistemi embedded, dei device di Windows CE e nei router di Cisco e anche le console Nintendo 64, Sony PlayStation, PlayStation 2 e PlayStation Portable utilizzano processori MIPS. Nel 1981 il professore John L. Hennessy della Stanford University avviò un gruppo di ricerca sulle architetture RISC. Le ricerche del team di sviluppo portarono allo sviluppo della prima generazione di processori MIPS. Allora era noto che per incrementare le prestazioni dei processori si sarebbe potuto utilizzare la tecnica delle pipeline. Questa tecnica sebbene fosse semplice da ideare non era semplice da implementare. Le pipeline per funzionare correttamente richiedono che le varie unità siano sincronizzate e che i dati delle varie istruzioni non si sovrappongano, quindi all'interno delle pipeline vengono posti dei blocchi (interblocco) che sorvegliano il completamento delle varie istruzioni e fanno procedere la pipeline solamente quando tutti gli stadi sono pronti. Questo meccanismo garantisce la corretta esecuzione del programma ma introduce spesso stalli nella CPU che deprimono le prestazioni.

Domanda3 : Processori DSP. SLIDE1

I processori DSP sono processori dedicati e ottimizzati per eseguire in maniera estremamente efficiente sequenze di istruzioni ricorrenti e specifiche. Molto usati nelle applicazioni embedded come nell’elaborazione di segnali di campionamento.

Un set di segnali generato da misurazioni campionate del mondo fisico, generalmente prese a una frequenza regolare chiamata frequenza di campionamento, viene elaborato da questa tipologia di processori. Le frequenze di campionamento vanno dai pochi Hz fino a diversi GHz.

I microprocessori DSP a chip singolo (SoC) sono apparsi per la prima volta all'inizio degli anni '80, e le caratteristiche centrali dei DSP includono **un'unità hardware ad accumulazione multipla**; diverse varianti dell'architettura di Harvard (per supportare il caricamento simultaneo di data e program); e modalità di indirizzamento che supportano l'incremento automatico, i buffer circolari e l'indirizzamento con inversione di bit (quest'ultimo per supportare il calcolo FFT).

La maggior parte di questi processori supporta la precisione dei dati in **virgola fissa di 16-24 bit**, in genere con accumulatori molto più ampi (40-56 bit) in modo che un gran numero di istruzioni di accumulazione multipla successive possano essere eseguite senza overflow.

I DSP sono difficili da programmare rispetto alle architetture RISC, principalmente a causa di istruzioni specializzate complesse, di una pipeline esposta al programmatore e da architetture di memoria asimmetriche.

Domanda36 : ISA.

L'instruction set, in informatica ed elettronica, è **l'insieme di istruzioni macchina** che descrive quegli aspetti, visibili a basso livello al programmatore, dell'architettura di un calcolatore, definita in inglese come *instruction set architecture* o in acronimo ISA.

Domanda41 : Differenza tra 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.

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 (Cyber Physical Systems ovvero i sensori, monitor, attuatori…), 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.

Domanda42 : Cosa è un Tap.

Nel filtro FIR (8-tap) un tap è il tempo di allocazione di una porzione di memoria e della sua elaborazione, in termini di istruzioni, nel buffer circolare. Generalmente nei processori DPS permette di scandire in tempi certi il numero di istruzioni per ciclo.

Domanda32 : Hazard.

La tecnica della pipeline è il primo passo nella ricerca di elevate prestazioni di elaborazione. La velocità effettiva è aumentata in media, ma spesso anche la latenza. Al fine di risolvere le bolle nella pipeline esistono diversi meccanismi:

Out-of-order Execution - Esecuzione fuori ordine: Dati A, B e C processi: viene fornito hardware che rileva un pericolo di dipendenza, ma invece di ritardare semplicemente l'esecuzione di B; procede a recuperare C e se C non legge i registri scritti da A o B e non scrive i registri letti da B, avverrà l'esecuzione di C prima di B. Ciò riduce ulteriormente il numero di bolle della pipeline.

Addressing Control Hazard:

Delay Branch meccanismo che documenta semplicemente il fatto che il branch verrà preso un certo numero di cicli dopo che è stato incontrato, e lascia al programmatore (o compilatore) il compito di assicurarsi che le istruzioni che dipendono dall'istruzione del brach condizionale **siano innocue (quindi vengono trattate come no-ops)** e facciano qualche operazione utile che non dipende da quel branch;

Interlocks: è hardware per inserire le bolle nella pipeline secondo necessità, proprio come il Data Hazard;

Nella *Speculative Execution* è l'hardware a stimare se è probabile che il branch venga preso, e inizia a eseguire le istruzioni che prevede di eseguire. Se le sue aspettative non vengono soddisfatte, annulla tutti gli effetti collaterali (come anche le scritture dei registri) causati dalle istruzioni che aveva *eseguito in modo speculativo*.

Data and Control Hazard:

L'analisi dei tempi di un programma può diventare estremamente difficile quando c'è una profonda pipeline con branch elaborati e speculazioni; *Le pipeline esplicite* sono relativamente comuni nei processori DSP, e sono spesso applicate in contesti in cui è essenziale una tempistica precisa; le *Speculative execution* invece sono comuni nei processori General Purpose, dove i tempi sono importanti solo in senso **aggregato**.

Un progettista di sistemi embedded deve comprendere i requisiti dell'applicazione ed evitare processori in cui il livello di precisione temporale richiesto è irraggiungibile.

Domanda23 : Timing.

Data and Control Hazard: *L'analisi dei tempi* di un programma può diventare estremamente difficile quando c'è una profonda pipeline con branch elaborati e speculazioni; *Le pipeline esplicite* sono relativamente comuni nei processori DSP, e sono spesso applicate in contesti in cui è essenziale una tempistica precisa; *Speculative execution* invece sono comuni nei processori General Purpose, dove i tempi sono importanti solo in senso **aggregato**.

Un progettista di sistemi embedded deve comprendere i requisiti dell'applicazione ed evitare processori in cui il livello di precisione temporale richiesto è irraggiungibile.

Domanda24 : Processore vettoriale.

Molte applicazioni embedded operano su tipi di dati notevolmente inferiori alla dimensione della parola del processore (es. dati RBG). In questi casi un'ampia ALU è suddivisa in sezioni più strette che consentendo operazioni aritmetiche o logiche simultanee con parole più piccole: le SUBWORD.

Intel ha introdotto il parallelismo delle subwords nel processore Pentium per General Purpose con MMX. Molte architetture di processori progettate per applicazioni embedded, inclusi molti processori DSP supporta anche il parallelismo delle subwords.

Un **processore vettoriale** è *quello in cui il set di istruzioni include operazioni su più elementi di dati contemporaneamente*.

Il parallelismo delle subwords è una forma particolare di **elaborazione vettoriale.**

Domanda33 : Architettura multi core.

Una macchina multicore è una combinazione di più processori su un singolo chip. Macchine multicore eterogenee combinano una varietà di tipi di processore su un singolo chip, rispetto a più istanze dello stesso tipo di processore.

Per le applicazioni embedded, le architetture multicore presentano un potenziale vantaggio significativo su architetture single-core **perché le attività in tempo reale e quelle critiche per la sicurezza possono avere una funzione dedicata per processore**.

Questo è il motivo per il quale si utilizzano architetture eterogenee per i telefoni cellulari, poiché le funzioni di elaborazione radio e vocale sono funzioni **hard real-time** con notevole carico computazionale.

Questa architettura può vantare cache multilivello, dove il secondo livello o superiore la cache è condivisa tra i core.

Abbiamo anche i **FPGA** sono chip la cui funzione hardware è programmabile utilizzando strumenti di progettazione hardware. Sfortunatamente, tale condivisione rende molto difficile isolare il comportamento in tempo reale del Architetture multicore.

I **Soft Core** sono processori implementati su FPGA. Il vantaggio dei soft core è che possono essere accoppiati strettamente all'hardware personalizzato più facilmente rispetto ai processori standard.

Domanda31 : Filtro FIR (Finite impulse response)

I DSP sono tipicamente macchine CISC e includono istruzioni che supportano specificamente il filtraggio FIR e spesso altri algoritmi come FFT (fast trasformate di Fourier) e la decodifica di Viterbi. Per qualificarsi come DSP, un processore deve essere in grado di eseguire FIR filtraggio in un ciclo di istruzioni per **tap** (fase del campionamento).

Ad esempio un filtro FIR può processare campioni alla frequenza di 1 MHz (un milione di campioni al secondo), e con N=32 bit, in uscite deve essere calcolata ad una frequenza di 1 MHz e ciascuna uscita richiede 32 moltiplicazioni e 31 addizioni quindi un processore deve essere in grado di sostenere una velocità di calcolo di 63 milioni di operazioni aritmetiche al secondo per implementare questa applicazione

Ad esempio il DSP TMS320c54x di Texas Instruments è concepito per essere utilizzato in applicazioni embedded con limitazioni di potenza che richiedono elevate prestazioni di elaborazione del segnale.

Il ciclo interno di un calcolo FIR è:

1. RPT numero di tap – 1 (ciclo zero overhead)
2. 2. MAC *AR2+, *AR3+, A

a := a + x * y

I registri AR2 e AR3 possono essere allestiti per implementare buffer circolari.

Il processore c54x include una sezione di memoria su chip che supporta due accessi in un unico ciclo, e fintanto che gli indirizzi si riferiscono a questa sezione della memoria, l'istruzione MAC verrà eseguita in un unico ciclo. Ogni ciclo, il processore esegue due recuperi di memoria, una moltiplicazione, un'addizione e due incrementi di indirizzo.

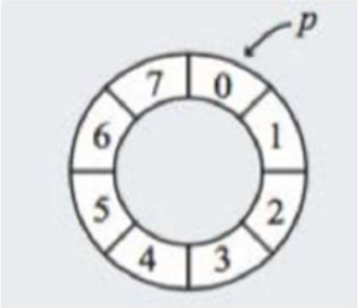
Il tempo per eseguire un filtro FIR ora si riduce a 1/2 ciclo per tap!

Domanda3 : Buffer circolare

Per realizzare un filtro FIR (Finite Impulse Response) che richiede una linea temporale, piuttosto che allocare in un array ciascun campione, uno in una nuova cella di memoria, è più conveniente usare un **buffer circolare** come struttura dati per l'elaborazione di tali campioni.

L'implementazione accetta prima un nuovo valore di input x(n) e poi calcola la somma a ritroso, iniziando con il termine i = N-1 termini, dove ad esempio N=8.

Quindi il buffer avrà sempre dentro otto campioni.



All'arrivo di ogni campione viene incrementato p dove p_i il contatore ={ 0, … 7}, quindi per il primo input abbiamo la cella x(0) e p_i =0.

Il sistema scrive il nuovo input *x(n)* nella posizione data da e quindi incrementa, impostando *p = pi +1.*

Ad ogni ciclo, il nuovo campione x[n] è inserito nella locazione relativa all'ultimo campione x[n-7] (quello più vecchio e l'indice di ognuno viene scalato)

Così facendo si genera una aritmetica in modulo 8.

~~Il calcolo del filtro FIR legge quindi x(n-7) dalla posizione *p* = p_i + 1 e lo moltiplica per a7. Il risultato viene memorizzato in un registro accumulatore (precedentemente azzerato). P viene aumenta di nuovo di uno… e così via..~~

Continua finché non legge x(n) dalla posizione p = pi + 8 scritto a stessa posizione dell'ultimo input *x(n) e moltiplica quel valore per a0.*

Ad esempio un filtro FIR sia dotato di campioni alla frequenza di 1 MHz (un milione di campioni al secondo), e che N = 32 e uscite devono essere calcolate a una frequenza di 1 MHz e ciascuna uscita richiede 32 moltiplicazioni e 31 addizioni quindi un processore deve essere in grado di sostenere una velocità di calcolo di 63 milioni di operazioni aritmetiche al secondo per implementare questa applicazione.

Il tempo per eseguire un filtro FIR ora si riduce a 1/2 ciclo per tap!

Domanda19 : Processori Embbedded vs General Purpose (tempo medio).

Le CPU General Purpose sono in media molto veloci perché sono superpipeline, superscalari e sfruttano ottimizzazioni come l'esecuzione fuori ordine, la previsione del brach che viene supportata da memorie cache veloci (tempo aggregato).

Per fare una valutazione del tempo medio **viene misurato il tempo totale trascorso e viene calcolato il tempo per l'elaborazione del campione**. In questo aspetto una CPU embedded, avendo hardware dedicato a svolgere l'elaborazione del campione senza fronzoli, è la scelta migliore; inoltre è meno costosa, spesso

con un consumo minore di risorse, ed anche il software è più semplice poiché realizzato su hardware che non deve fare diverse operazioni generiche, ma sempre le stesse… quindi con istruzioni ben definite allo scopo e tempi certi: per esempio i DSP.

I processori embedded utilizzano spesso architetture VLIW anziché architetture superscalari **per ottenere tempi più ripetibili e prevedibili**. I processori VLIW includono più unità di funzione, come i processori superscalari, ma invece di determinare dinamicamente quali istruzioni possono essere eseguite contemporaneamente, ciascuna istruzione specifica cosa dovrebbe fare ogni unità di funzione in un ciclo particolare. Un set di istruzioni VLIW combina più operazioni indipendenti in un'unica istruzione.

Infatti i processori superscalari presentano uno **svantaggio significativo per i sistemi embedded**, ovvero **che i tempi di esecuzione possono essere estremamente difficili da prevedere** e, nel contesto del multitasking (interrupt e thread), potrebbero non essere nemmeno ripetibili.

Gestione di sistemi real-time richiedono vincoli temporali. Meccanismi hardware come le pipeline esplicite sono spesso applicate in contesti in cui è essenziale una tempistica precisa;

Per i sistemi embedded vanno privilegiati i meccanismi che rendono le tempistiche precise e prevedibili come:

Data and Control Hazard: *L'analisi dei tempi* di un programma può diventare estremamente difficile quando c'è una profonda pipeline, con branch elaborati e speculazioni; *Le pipeline esplicite* sono **relativamente comuni nei processori DSP**, e sono spesso applicate in contesti in cui è essenziale una tempistica precisa; *Speculative execution* invece sono comuni nei **processori General Purpose**, dove i tempi sono importanti solo in senso aggregato.

Un progettista di sistemi embedded deve comprendere i requisiti dell'applicazione ed evitare processori in cui il livello di precisione temporale richiesto è irraggiungibile.

Domanda13 : Rappresentazione formato in virgola fissa.

Molti processori embedded forniscono hardware solo per l'aritmetica di interi. Per rappresentare un numero decimale, la strategia è quella di usare **il punto binario,** che è come un punto decimale, tranne per il fatto che separa i bit anziché le cifre decimali dal numero. Consideriamo ad esempio un numero intero a 16 bit.

Senza il punto binario, un numero rappresentato dai 16 bit è un numero intero x { -2^15, …, 2^15 -1} (complemento a due).

Invece con il punto binario, interpretiamo i 16 bit per rappresentare un numero y = x/2^15. Quindi l'intervallo sarà y { -1, …, 1 – 2^15}. Questo è noto come numero a virgola fissa. **Il formato di questo numero a virgola fissa può essere scritto 1,15** che indica che c'è un bit a sinistra del punto binario e 15 a destra**.**

Con questa metrica è possibile rappresentare i numeri compresi tra -1 e 1 (approssimativamente) posizionando un punto binario (concettuale) appena sotto il bit di ordine superiore del numero.

Quando due di questi numeri vengono moltiplicati alla massima precisione, il risultato è un numero a 32 bit. La posizione del punto binario segue dalla legge di conservazione dei bit (con i formati nm e pq, il risultato ha il formato (n+p).(m+q)).

I processori spesso supportano tali moltiplicazioni ad alta precisione, in cui il risultato va in un registro accumulatore che ha almeno il doppio dei bit rispetto ai normali registri di dati. Per riscrivere il risultato in

un registro dati, però, dobbiamo estrarre 16 bit dai 32 bit del risultato. Esiste la possibilità di overflow, perché stiamo scartando il bit di ordine superiore (es -1 *-1).

C'è una perdita di informazioni qui: se scartiamo semplicemente i 15 bit di ordine inferiore, la strategia è nota come **troncamento;** se invece aggiungiamo un bit alla parte più significativa del risultato a 32 bit, il risultato è noto come **arrotondamento**. L'arrotondamento sceglie il risultato più vicino al risultato di precisione completa, mentre il troncamento sceglie il risultato più vicino che è di grandezza minore.

I processori DSP in genere eseguono **l'estrazione con arrotondamento o troncamento** nell'hardware quando i dati vengono spostati da un accumulatore a un registro generico o alla memoria.

ARM

Domanda15 : I registri ARM.

ARM ha 37 registri in totale, tutti lunghi 32 bit. 1 è Program Counter Register (r15), 1 è Current Program Status Register, 5 sono lo Saved Program Status Register e 30 General Purpose Register. Questi registri sono organizzati in diversi banchi, l'accessibilità ad ogni banco è regolata dalla modalità processore (USER, FIQ, IRQ, SUPERVISOR, ABORT, UNDEF, in V4 anche SYSTEM).

Ogni modalità può accedere: ai registri da r0-r12, a un particolare registro r13 (Stack Pointer), r14 (Link Register), al registro r15 (Program Counter) e al CPSR (Current Program Status Register).

Le modalità privilegiate possono accedere anche: al SPSR (Saved Program Status Register).

Invece tutte le istruzioni possono accedere direttamente a r0-r14. La maggior parte delle istruzioni consente anche l'uso del PC (r15).

Domanda38 : Program Counter.

Il Program counter è il registro che, nella fase di fecth della pipeline, memorizza la prossima istruzione da eseguire. Il Program Counter punta 8 byte (2 istruzioni) oltre l'istruzione corrente (fase di execute); con istruzioni a 32 bit e memoria indirizzata al byte, l'incremento necessario a indirizzare l'istruzione successiva sarà pari a 4 pertanto il valore del PC viene memorizzato nei bit [31:2] con i bit [1:0] uguali a zero (poiché l'istruzione non può essere allineata a metà parola o byte).

~~Lo stack deve rimanere sempre allineato a 4 byte e deve essere allineato a 8 byte in qualsiasi limite di funzione.~~

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).

Domanda51 : Status Register.

Status Register (SREG o SR) contiene dei bit associati alle operazioni del processore e viene utilizzato per controllare il flusso di esecuzione del programma. I bit di stato tipici sono:

- ▪ overflow (V) – il risultato dell'operazione non è rappresentabile
- ▪ carry (C) – l'operazione ha dato luogo ad un riporto
- ▪ zero (Z) – il risultato dell'operazione è zero
- ▪ negative (N) – il risultato dell'operazione è negativo

Domanda20 : Il Link Register.

Il Link Register è un registro usato per scopi speciali che contiene **l'indirizzo di ritorno** quando viene completata una chiamata di funzione. ~~Questo è più efficiente dello schema più tradizionale di memorizzazione degli indirizzi di ritorno in uno stack di chiamate, a volte chiamato stack di macchine~~. Il Link Register non richiede scritture e letture della memoria contenente lo stack che può far risparmiare una notevole percentuale di tempo di esecuzione con ripetute chiamate di piccole subroutine(soprattutto nei loop) VEDI STORE MULTIPLE O BRANCH WITH LINK.

Il Link Register è una ottimizzazione di ARM per ridurre le chiamate a funzione nei loop.

Domanda18 : Gestione dello Stack.

In ARM 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 essa, 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":

1. Stack pointer: punta sempre alla cima dello stack
2. base pointer: punta sempre la base

solitamente 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.

Lo Stack è un'area di memoria che cresce quando i nuovi dati vengono "pushati" sulla "parte superiore" di esso e si restringe quando i dati vengono "spuntati" dall'alto. Due puntatori definiscono i limiti correnti dello stack*: Base pointer* e *lo Stack pointer.*

Un uso degli stack è quello di creare un'area di lavoro di registro temporanea per 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 ripristinarlo prima di ritornarlo al chiamante.

ARM usa lo STMFD / LDMFD: **stack discendente completo** ovvero lo Stack Pointer punta sempre all'ultimo indirizzo occupato (Full stack).

Tutti i registri necessari possono essere inseriti nello stack all'inizio della subroutine e estratti di nuovo alla fine in modo da ripristinarli prima di tornare al chiamante: STMFD sp!,{r0-r12, lr} impilare tutti i registri e l 'indirizzo di ritorno.

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

Se l'istruzione pop ha anche il bit '**S**' impostato (**usando '^'**), il trasferimento del PC in una modalità privilegiata causerebbe **anche la copia dell'SPSR nel CPSR** (vedi modulo di gestione delle eccezioni).

Domanda34 : Barrel Shifter

Arm non ha una vera e propria istruzione di spostamento, ma circuiteria specifica che crea un meccanismo di spostamento, il Barrel Shift : il quale permette di fare shift con riporto oltre alle istruzioni.

Esistono diverse tipologie di SHIFT:

Logical Shift Left: produce spostamenti a sinistra per l'immediato specificato (moltiplica per potenze di due) es: **LSL** #5 = moltiplica per 32.

Logical Shift Right: Sposta a destra della quantità specificata (divide per potenze di due) es: **LSR** #5 = dividere per 32

Arithmetic **S**hift **R**ight: sposta a destra (divide per potenze di due) e conserva il bit del segno, per le operazioni in complemento a 2. per esempio ASR #5 = dividere per 32

ROR è simile a un ASR ma i bit ruotati. Quindi partendo da LSB appaiono come MSB. L'ultimo bit ruotato è utilizzato anche come Carry Out. ROR #5

Rotazione a destra estesa (**RRX**): Questa operazione utilizza il flag CPSR come 33° bit. ~~Ruota a destra di 1 bit. ROR #0.~~

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.

La quantità la quale il registro dev'essere "shiftato" può essere 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.

Nel Barrel Shift il *secondo operando* può essere~~: in un **registro** con un valore intero senza segno a 5 bit oppure di un valore specificato in un altro registro (byte inferiore); un **Immediato** (numero da 8 bit) che può essere ruotato a destra di un numero pari di posizioni~~. È l'assembler a calcolare la rotazione a partire da una costante per un numero pari di posizioni.

Domanda12 : ARM Data Processing Instructions e Format. SLIDEARM

I Processori ARM condividono tutti lo stesso formato di istruzione. Queste istruzioni possono essere: Operazioni **Aritmetiche, Confronti, Operazioni Logiche, Spostamento tra registri**. Le istruzioni funzionano esclusivamente sui registri e non sulla memoria. Infatti ARM è un'architettura LOAD/STORE. Il primo operando è sempre un registro (Rn); il secondo operando che è inviato all'ALU attraveriso il barrel shifter, può essere: **omesso**, un **registro**, o un **immediato**.

Domanda26 : Formato istruzione ARM. SLIDEARM

In ARM32 tutte le istruzioni sono lunghe 32 bit. Tutte le istruzioni devono essere allineate a WORD. La maggior parte viene eseguita in un singolo ciclo; ogni istruzione può essere eseguita condizionalmente.

Poiché si tratta di un'Architettura load/store:

◦Le istruzioni di processamento dati possono agire solo sui registri;

◦Istruzioni specifiche per l'accesso alla memoria con modalità di indirizzamento ad auto-indicizzazione.

La maggior parte degli *istruction set* permette l'esecuzione condizionale del branch solo posponendo l'appropriato campo di condizione; **ogni istruzione contiene un campo di condizione** che determina se la cpu eseguirà o meno l'istruzione stessa; tutto ciò causa l'incrementato del numero di istruzioni che, se non esguite consumano comunque un ciclo; questo tipo di approccio rimuove il bisogno di molti branch (ogniuno dei quali richiederebbe 3 cicli per riempire la pipeline). ~~Codice denso, senza branch;~~

La perdita di tempo di non eseguire alcune istruzioni condizionali è minore di quella per gestire una chiamata a branch o subroutine.

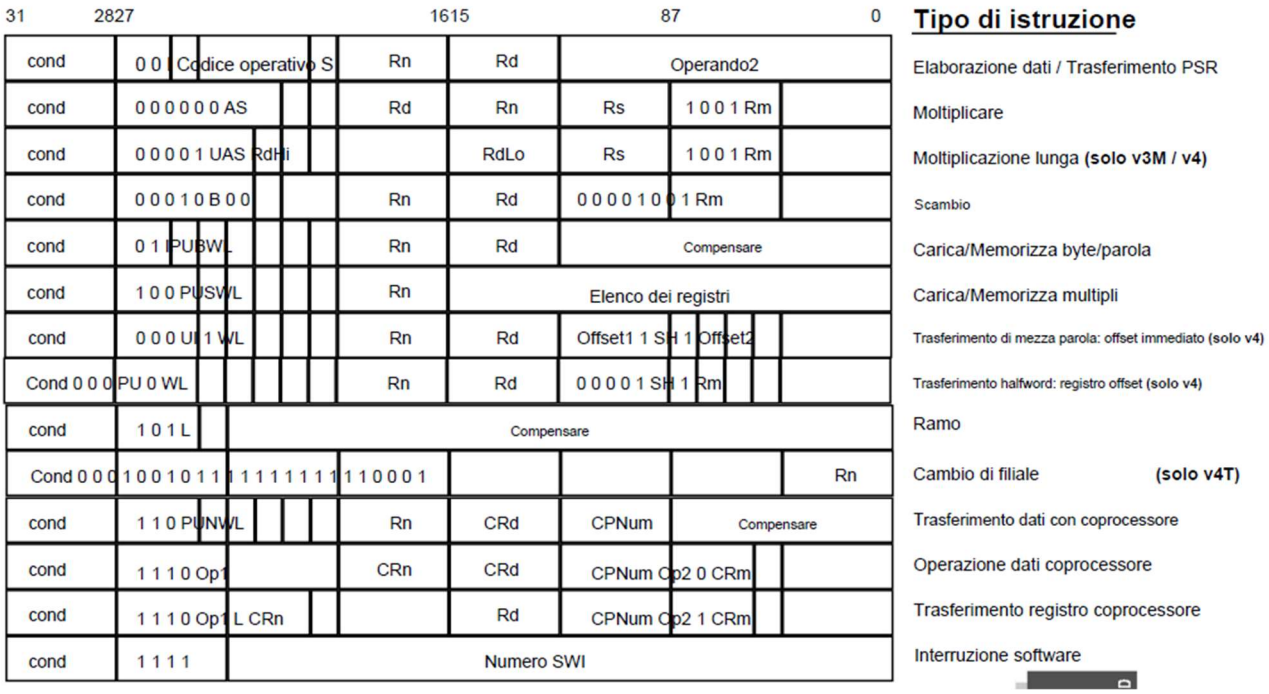
I tipi di dati sui quali le istruzioni ARM operano sono:

1 Byte -> 8 bit; Half word 2 byte, 16 bit; Full word o **word 4 byte**, 32 bit.

Le istruzioni si dividono nelle seguenti categorie:

1. ◦Data processing
2. ◦Branch ◦Load-store
3. ◦Software interrupt
4. ◦Program status register.

Abbiamo visto 14 tipi di istruzioni arm:



Per creare una istruzione condizionata è semplicemente necessario post-fissare la condizione.

Tutte le istruzioni contengono un campo condizione (i bit dal 31 al 28).

Il bit 24 è il Bit di BRANCH (0 Branch, 1 Branch con Link).

La differenza tra l'istruzione branch e l'indirizzo target deve essere minimo 8 (per consentire la pipeline). Il bit 26 shiftato a destra di 2 bits.

OFFSET da 24 bit per il conteggio shiftato LSL 2

Domanda37 : Per codificare un riferimento ad un registro quanti bit servono?

Generalmente servono 4 bit. Ma nel caso di una load/store multiple (LDM / STM) basta 1 bit (da 0 a 15).

Domanda4 : Istruzioni di somma in ARM. SLIDEARM

Ogni istruzione Arm esegue un'operazione specifica su uno o due operandi. Il primo operando è sempre nel registro Rn mentre il Secondo operando è inviato all'ALU tramite barrel shifter e può essere un immediato, o un registro.

Abbiamo 2 istruzioni di somma e sono ***ADD*** *Operando1 + Operando2* e ***ADC*** *Operando1 + Operando2+Carry*

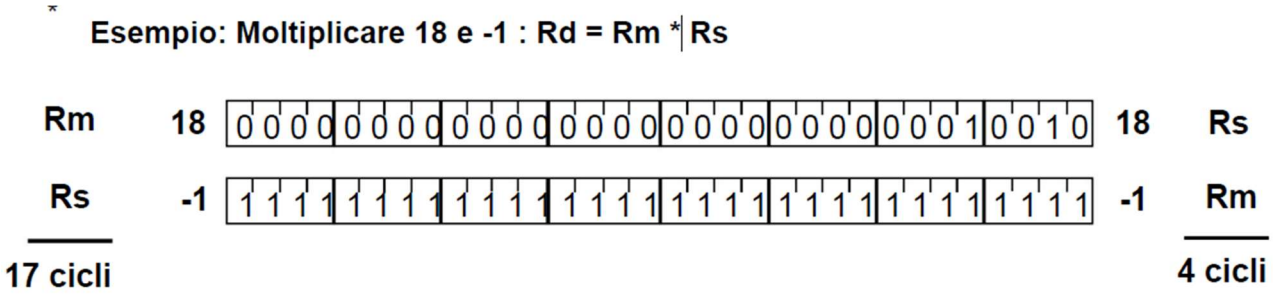
La sintassi è <Operazione>{<cond>}{S} Rd, Rn, Operando2 dove il secondo operando può essere pure un Immediato #Imm.

Domanda28 : Arm moltiplicazione con e senza MUL.

CON MUL

ARM fornisce due istruzioni di moltiplicazione MUL e MLA. Rd e Rm (Risultato e Moltiplicando) non possono essere lo stesso registro e ciò può essere evitato scambiando (i fattori) Rm e Rs. Questo funziona perché la moltiplicazione è commutativa (NO PC).

ARM utilizza l'algoritmo **di Booth 8 bit** per eseguire la moltiplicazione di interi (usando il barrel shift). Per moltiplicare ogni coppia di bit occorre 1 ciclo (più 1 ciclo per cominciare) fino a quando non rimangono più bit 1 nel Rs, che porterà la moltiplicazione a terminare anticipatamente.



ARM contiene un hardware di moltiplicazione esteso. Ciò fornisce tre miglioramenti: la moltiplicazione viene eseguita più velocemente (il massimo per le istruzioni standard è ora di 5 cicli); usa il metodo di terminazione anticipata migliorato in modo da completare la moltiplicazione quando tutti i set di bit rimanenti contengono tutti zeri o tutti uno a seconda di cosa sia più conveniente; è ora possibile produrre risultati a 64 bit da due operandi a 32 bit per maggiore precisione, infatti viene usata una coppia di registri.

SENZA MUL si può fare con lo shift e le altre operazioni.

Per le moltiplicazioni spesso è possibile trovare una soluzione più ottimale utilizzando una combinazione di MOV, ADD, SUB e RSB con gli shift. Si possono fare moltiplicazioni per una costante uguale a ((potenza di 2)+- 1) in un ciclo.

Ad esempio: Usare ADD con LSL#2 per caricare 3 nel registro r1, e mettere **r1*5** in r3 e poi termini usando il valore contenuto in r3.

```
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 @Uscita
```

Domanda30 : Salto condizionato.

La maggior parte dei set di istruzioni consente l'esecuzione di branch solo in modo condizionale. Invece ARM riutilizzando l'hardware di valutazione delle condizioni e **aumenta in maniera efficente il numero di istruzioni.** Infatti tutte le istruzioni contengono un campo condizionale che determina se la CPU le eseguirà. Le istruzioni non eseguite assorbono comunque 1 ciclo. Ciò elimina la necessità di molte diramazioni, che bloccherebbero la pipeline. La penalità di tempo per non aver eseguito più le istruzioni condizionali è spesso inferiore all'overhead della chiamata a branch o a subroutine che sarebbe altrimenti necessaria.

I bit tra 31:28 rappresentano il blocco condizionale (N Z C V). Per eseguire un'istruzione in modo condizionale, è sufficiente post-fissare l'istruzione con la condizione adeguata.

B {cond} target

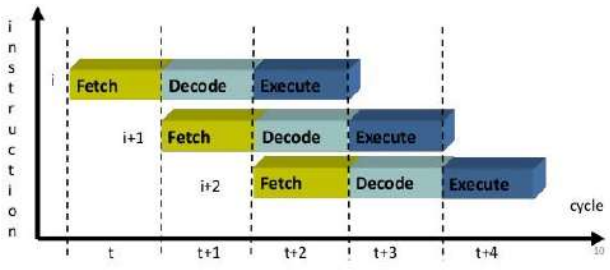
L'istruzione effettua un salto alla destinazione (relativa al PC) specificata se la condizione specificata `e vera.

ADDEQ r0,r1,r2 ; Se il flag zero è impostato, allora... r0= r1 + r2 . Per fare in modo che i flag di condizione vengano aggiornati, è necessario impostare il bit S (bit 20 per ARITMETICA) dell'istruzione.

{cond}	Flags	Significato
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=

Domanda9 : La pipeline ARM. SLIDEARM

ARM utilizza una pipeline per aumentare la velocità del flusso d'istruzioni al processore. Consente di eseguire più operazioni contemporaneamente anziché in serie. Il flusso prevede tre passaggi: fase di FETCH, fase di DECODE e fase di EXECUTE.



Nella fase di caricamento il PC punta sempre all'istruzione successiva e mai a quella in esecuzione.

Nella fase di decodifica viene effettua la decodifica dei registri utilizzati nell'istruzione e nello stesso tempo viene preparata l'istruzione per il ciclo successivo.

Nella fase di esecuzione vengono fatte diverse operazioni: Lettura dei registri coinvolti dalla Register Bank, Barrel Shift e operazione ALU, Riscrittura dei registri nella Register Bank.

In ogni momento 3 diverse istruzioni possono occupare ognuno dei tre stadi. L'hardware di ogni stadio deve poter operare in maniera indipendente. Quando il processore esegue istruzioni di processamento dati la latenza è pari a 3 cicli e il throughtput è 1 istruzione per ciclo.

Il lato negativo è che ogni istruzione di trasferimento dati causa uno stallo della pipeline. Essendo la memoria unica per dati e istruzioni non è possibile leggere l'istruzione successiva mentre si leggono dati.

Le istruzione ARM sono allineate a 4 byte.

Domanda43 : Problema dell'allineamento degli indirizzi in ARM. ?

Le istruzioni sono allineate a 4 byte perché devono gestire la latenza della pipeline. Il program counter carica le istruzioni: PC-0, PC-4, PC-8… così da essere allineate a word.

Domanda29 : Interruzione software (SWI)

Campo Condizione 1111.

Un'interruzione software o interrupt, ~~un SWI~~ è un'istruzione definita dall'utente. Provoca una trap di eccezione al vettore hardware SWI che fa passare alla modalità supervisore, più il salvataggio dello stato associato.

Utilizzando questo meccanismo ~~SWI~~, un sistema operativo può implementare un insieme di operazioni privilegiate che le applicazioni in esecuzione in modalità utente possono richiedere.

Domanda10 : Interlocks. SLIDE2

DaSlide

Le istruzioni ML spesso dipendono dai risultati delle precedenti istruzioni: Se B necessita di un risultato da un registro scritto da A (data hazard) non può procedere lungo la pipeline. Nelle macchine semplici il programmatore (o il compilatore) deve occuparsi di tali dipendenze (la cosiddetta pipeline esplicita). Il compilatore inserisce tre istruzioni no-op (che non fare nulla) tra A e B per garantire che la scrittura avvenga prima della lettura. Queste istruzioni no-op formano una bolla di pipeline che si propaga lungo la stessa. Hardware aggiuntivo può gestire dati e controllo problemi di rischi. **Interlock inserisce automaticamente delle bolle della** pipeline.
Nelle macchine più sofisticate vengono adottate tecniche, come la ridenominazione dei registri, l'esecuzione fuori servizio e speculativa, la previsione dei branch, le pipeline speculative e la superscalarità.

DaInternet

Interlocks

Il risultato di un'istruzione di caricamento della parola allineata non è disponibile fino al termine della fase di memoria della pipeline. Se la seguente istruzione richiede l'uso di questo risultato, allora deve essere interbloccato in modo che sia disponibile il valore corretto. Questo interblocco è denominato interblocco uso carico a ciclo singolo.

L'esempio seguente comporta un interblocco a ciclo singolo:

LDR r0, [r1]

ADD r2, r0, r3

ORR r4, r4, r5

L'esempio seguente non prevede un interblocco:

LDR r0, [r1]

ORR r4, r4, r5

ADD r2, r0, r3

Le istruzioni di caricamento di parole non allineate, di byte di caricamento (LDRB) e di caricamento di semiparole (LDRH) utilizzano l'unità di rotazione dei byte nella fase di scrittura della pipeline. Ciò introduce un interblocco uso-carico a due cicli, che può influenzare le due istruzioni immediatamente successive all'istruzione di carico.

L'esempio seguente prevede un interblocco a due cicli:

LDRB r0, [r1, #1]

ADD r2, r0, r3

ORR r4, r4, r5

Domanda11 : Branch e Branch with Link e BX exchange

Salto incondizionato. Abbiamo tre tipi di istruzione:

B target: L'istruzione effettua un salto alla destinazione specificata.

BL target LR [PC], *jump to target:* Salto incondizionato con collegamento (branch with link). L'istruzione effettua un salto alla destinazione specificata copiando preventivamente (PreFetch) su Link Register (LR/R14) il valore del Program Counter (PC/R15).

B Branch i bit 23-0 contengono il valore di offset (complemento a due); l'offset è relativo all'indirizzo dell'istruzione branch +2 word. L'offset viene spostato di due posizioni a sinistra e aggiunto al PC. Il bit 24 indica se il valore PC-8 deve essere copiato in LR (Link) prima della diramazione o meno. Il bit meno significativo dell'offset è sempre 0, quindi non è codificato nell'istruzione.

BL (branch and link) e MOV PC, LR sono le due istruzioni essenziali necessarie per una chiamata e un ritorno di funzione. BL svolge due compiti: memorizza l'indirizzo di ritorno dell'istruzione successiva (l'istruzione dopo BL) nel Link Register (LR), e si dirama all'istruzione di destinazione.

Teniamo presente che PC e LR sono nomi alternativi rispettivamente per R15 e R14. Per ARM il PC fa parte del set di registri, quindi è possibile eseguire un ritorno di funzione con un'istruzione MOV. Molti altri set di istruzioni mantengono il PC in un registro speciale e utilizzano un'istruzione speciale di ritorno o salto per tornare dalle funzioni.

BX Exchange. I compilatori ARM eseguono una funzione di ritorno utilizzando BX LR. L' istruzione branch e exchange BX è come una branch, ma può anche effettuare la transizione tra il set di istruzioni ARM standard e il set di istruzioni Thumb.

I processori ARM sono dotati di un set di istruzioni a 16 bit chiamato **Thumb** che utilizza sempre quattro byte per ogni istruzione. Il codice Thumb è più leggero, ma è dotato di meno funzionalità. Per esempio solo i salti possono essere condizionati e alcuni opcode non possono essere utilizzati da tutte le istruzioni. Nonostante queste limitazioni, Thumb fornisce prestazioni migliori del set di istruzioni completo nel caso di sistemi dotati di limitata larghezza di banda. Molti sistemi embedded sono dotati di un bus verso la

memoria limitato e, sebbene il processore possa indirizzare a 32 bit, spesso si utilizzano indirizzamenti a 16 bit.

Funzionamento

Qualsiasi chiamata ad un Branch ARM con operazione di collegamento richiedono tre cicli:

Durante il primo ciclo, un'istruzione branch calcola la destinazione del branch e copia i dati durante l'esecuzione di un Prefetch dal PC corrente. Questa Prefetch viene eseguita in ogni caso, perché nel momento in cui è stata presa la decisione di prendere il branch, è già troppo tardi per impedire la Prefetch. Se l'istruzione precedente ha richiesto un accesso alla memoria dati, i dati vengono trasferiti in questo ciclo.

I bit 23-0 contengono il valore di offset (complementato a due) e l'offset è relativo all'indirizzo dell'istruzione branch +2 word, l'offset viene spostato di due posizioni a sinistra e aggiunto al PC, il bit 24 indica se il valore PC-8 deve essere copiato in LR (Link) prima della diramazione o meno.

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 wordaligned) e memorizzati nella codifica dell'istruzione. Questo ci da un range di +- 32 Mbytes.

Generalmente una subroutine è un blocco di codice che esegue delle istruzioni in base ad alcuni argomenti e opzionalmente 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
```

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:
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.

Domanda17 : Load/Store e Multiple.

ARM è un'architettura Load/Store. Non supporta le operazioni di elaborazione dei dati direttamente da memoria a memoria. Sposta i valori dei dati nei registri prima di utilizzarli.

1 Caricare i valori dei dati dalla memoria nei registri.

2 Elaborare i dati nei registri utilizzando una serie di istruzioni di elaborazione dati che non sono rallentate dall'accesso alla memoria.

3 Memorizza i risultati dei registri in memoria.

Le istruzioni e gli mnemonici sono: **Trasferimento dati a registro singolo** (LDR / STR), **trasferimento blocco dati** (LDM/STM), **Swap** (SWP).

Single Register Data Transfer: Le istruzioni di LOAD e STORE sono LDR/STR/LDRB/STRB. Nella V4 sono supportate le mezze parole, ed ogni istruzione può essere condizionale.

Intanto la posizione di memoria a cui accedere è conservata in un **registro base**:

STR r0, [r1] Memorizzare il contenuto di r0 nella posizione indicata dal contenuto di r1

LDR r2, [r1] Carica r2 con il contenuto della posizione di memoria indicato dal contenuto di r1.

Oltre ad accedere alla posizione effettiva contenuta nel registro di base, queste istruzioni possono accedere a un offset di posizione dal puntatore del registro di base che può essere un immediato da 12 bit (0-4095) o un registro opportunamente shiftato ad un valore immediato che può essere aggiunto o sottratto dal registro di base (base + o base -).

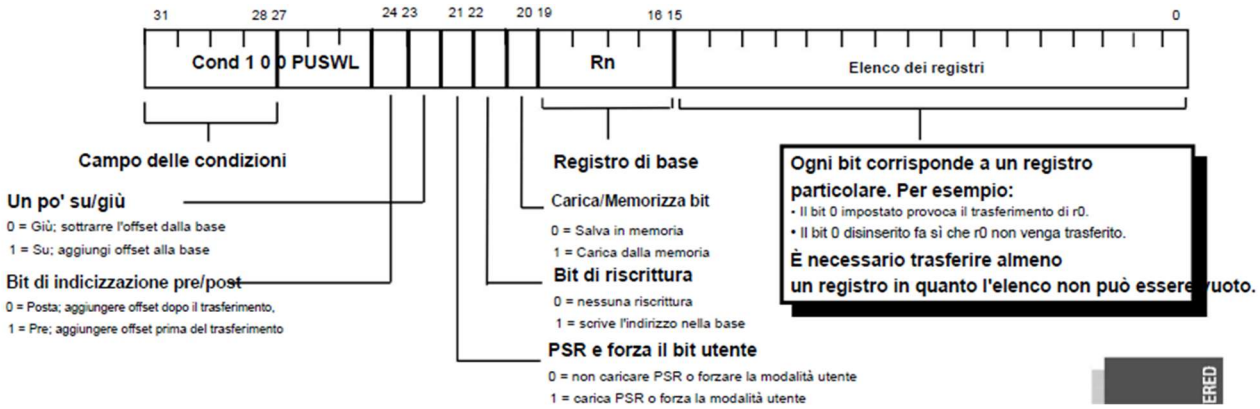
Questo offset può essere applicato prima che venga effettuato il trasferimento (Indirizzamento pre-indicizzato) oppure dopo (Indirizzamento post-indicizzato) provocando l'autoincremento del registro di base.

Pre-indicizzazione *STR r0, [r1,#-12]* oppure *STR r0, [r1, #12]!* o *STR r0, [r1, r2, LSL #2]* se r2 contiene 3.

Post-indicizzazione *STR r0, [r1], #12* oppure *STR r0, [r1], #-12* o *STR r0, [r1], r2, LSL #2* se r2 contiene 3.

Block Data Transfer: Le istruzioni di LOAD e STORE multiple (LDM / STM) sono consentite per i registri dall' 1 al 16 per il trasferimento dalla memoria, e per la memoria.

Ogni bit dall'0 al 15 corrisponde ad un registro. Il bit 0 impostato provoca il trasferimento di r0. È necessario trasferire almeno un registro in quanto l'elenco non può essere vuoto.



Il Base Register è utilizzato per determinare da dove si dovrebbe accedere alla memoria. Abbiamo 4 diverse modalità di indirizzamento consentite: incrementi e decrementi inclusi o esclusivi del registro di base.

Il registro di base può essere aggiornato facoltativamente a seguito del trasferimento (!).

Queste istruzioni sono molto efficaci per il salvataggio e ripristino del **context** e per lo spostamento di grandi blocchi di dati nella memoria.

LDM / STM vengono utilizzati anche per implementare gli stack.

SWAP: L'istruzione SWP è una operazione atomica di una lettura seguita da una scrittura, che sposta quantità di byte o parole tra i registri e la memoria.

SWP{<cond>}{B} Rd, Rm, [Rn]

Domanda25 : Caratteristiche del secondo operando in ARM.

Siamo nel Barrel Shift!

Nelle istruzioni, dove previsto, il secondo operando può essere di 2 tipo:

Se il secondo operando è un registro e il valore dello stesso oltre un immediato 5 bit senza segno, in questo caso non c'è overhead, quindi l'operazione di shift avviene nello stesso ciclo; oppure per il secondo operando lo shift può essere specificato nel byte inferiore di un registro successivo (che non sia il PC) allora servirà un ciclo in più.

Se non viene specificato alcun Shift, viene applicato quello predefinito: LSL #0 (nessun effetto sul valore)

Infatti per le moltiplicazioni spesso è possibile trovare una soluzione più ottimale utilizzando una combinazione di MOV, ADD, SUB e RSB con gli shift. Si possono fare moltiplicazioni per una costante uguale a ((potenza di 2)+- 1) in un ciclo.

Se il secondo operando è un immediato esiste una singola istruzione che caricherà una costante immediata a 32 bit in un registro senza eseguire un caricamento dati dalla memoria. **Il Data Processing instruction** ha 12 bit disponibili per il secondo operando.

Se usato direttamente questo darebbe solo un intervallo fino da 0 a 4095. Questo viene utilizzato per memorizzare costanti a 8 bit, fornendo un intervallo compreso tra 0 e 255. Questi 8 bit possono quindi essere ruotati fino a un numero pari di posizioni (cioè ROR di 0, 2, 4,..30) e ciò fornisce una gamma molto più ampia di costanti che possono essere caricate direttamente senza accedere alla memoria.

0 – 255	[0 - 0xff]
256,260,264,..,1020	[0x100-0x3fc, passaggio 4, 0x40-0xff ror 30]

1024,1040,1056,..,4080 [0x1000-0x3fc0, passaggio 64, 0x40-0xff ror 26]

4096,4160, 4224,..,16320 * [0x400-0xff0, passaggio 16, 0x40-0xff ror 28]

Ad esempio MOV r0, #0x40, 26 ; => MOV r0, #0x1000 (cioè 4096)

Per valori superiore ci sono 2 meccanismi: I complementi bit per bit possono essere formati anche utilizzando MVN: es MOV r0, #0xFFFFFFFF ; assembla in MVN r0, #0;

Caricamento Di Costanti A 32bit

oppure l'assembler fornisce anche un metodo che caricherà QUALSIASI costante a 32 bit LDR rd,=costante numerica e se la costante può essere costruita utilizzando un MOV o MVN l'assembler genera questa in caso contrario, l'assembler produrrà un'istruzione LDR con un indirizzo relativo al PC per leggere la costante da un pool letterale

LDR r0,=0x42 ; genera MOV r0,#0x42

LDR r0,=0x55555555 ; genera LDR r0,[pc, offset a pool di costanti letterali]

Domanda25 : Preindicizzazione e postinticizzazione in ARM.

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:
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'elemento che vogliamo
LDR r2, [r0,r1, LSL #2]
R1 è l'indirizzo dell'elemento corrente
LDR r2, [r1], #4

Oltre ad accedere alla posizione effettiva contenuta nel registro di base, le istruzioni di Load e Store possono accedere a un offset di posizione dal puntatore del registro di base che può essere un immediato da 12 bit (0-4095) o un registro opportunamente shiftato ad un valore immediato che può essere aggiunto o sottratto dal registro di base (base + o base -).

Questo offset può essere applicato prima che venga effettuato il trasferimento **Indirizzamento pre-indicizzato** (opzionalmente auto-incrementa il base register posticipando il simbolo '!'), oppure dopo **Indirizzamento post-indicizzato** provocando l'auto-incremento del registro di base.

Pre-indicizzazione *STR r0, [r1,#-12]* oppure *STR r0, [r1, #12]!* o *STR r0, [r1, r2, LSL #2]* se r2 contiene 3.

Post-indicizzazione *STR r0, [r1], #12* oppure *STR r0, [r1], #-12* o *STR r0, [r1], r2, LSL #2* se r2 contiene 3.

```
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
```

HARDWARE

Domanda40 : LCD1602.

Nel display LCD sono presenti diversi pin, tra cui: VSS, VDD, VEE, RS, RW, E e otto diversi D (da D0 a D7). Nello specifico ai pin VSS, CC e VEE vengono collegate rispettivamente massa, 5V ed un potenziometro per l'aggiustamento del contrasto. I pin RS ed R/W possono avere valore 0 o 1, nel caso dell'RS se questo ha valore 0 significa che è settato ad Instruction input, se ha valore 1 sarà settato su Data input; per R/W, invece, se ha valore 0 corrisponde all'operazione di "Write to LCD module", invece se settato a 1, "Read rom LCD module". Il Pin E "Enable" abilita la trasmissione dei segnali.

Ultimi, ma non per importanza, abbiamo i pin da D0 a D7, chiamati D "Data bus line", per i quali il D0 rappresenta il LSB mentre il D7 e il MSB.

5. 0 PIN ASSIGNMENT

No.	Symbol	Level	Function	
1	Vss	--	0V	Power Supply
2	Vdd	--	+5V	
3	V0	--	for LCD	
4	RS	H/L	Register Select: H:Data Input L:Instruction Input	
5	R/W	H/L	H--Read L--Write	
6	E	H,H-L	Enable Signal	
7	DB0	H/L	Data bus used in 8 bit transfer	
8	DB1	H/L		
9	DB2	H/L		
10	DB3	H/L		
11	DB4	H/L	Data bus for both 4 and 8 bit transfer	
12	DB5	H/L		
13	DB6	H/L		
14	DB7	H/L		
15	BLA	--	BLACKLIGHT +5V	
16	BLK	--	BLACKLIGHT 0V-	

La RAM di visualizzazione dei dati (DDRAM) memorizza i dati di visualizzazione rappresentati in codici di caratteri a 8 bit. L'area nella DDRAM che non viene utilizzata per la visualizzazione può essere utilizzata come RAM dei dati generali; quindi qualunque cosa inviate sulla DDRAM viene effettivamente visualizzata sul display LCD.

Come si evince dal nome, l'area CGRAM viene utilizzata per creare caratteri personalizzati nell'LCD. Nella RAM del generatore di caratteri, l'utente può riscrivere i modelli di caratteri tramite programma. Per 5 x 8 punti, è possibile scrivere otto modelli di caratteri e per 5 x 10 punti, invece, solo quattro modelli di caratteri.

Il Busy Flag è l'indicatore di stato per LCD; quando inviamo un comando o dati all'LCD per l'elaborazione, questo viene impostato (cioè BF = 1) e non appena l'istruzione viene eseguita con successo viene cancellato (BF = 0) ciò permette la produzione di una quantità esatta di ritardo per l'elaborazione LCD.
Per leggere l'indicatore di occupato, è necessario soddisfare la condizione RS = 0 e R / W = 1 e l'MSB del bus dati LCD (D7) funge da indicatore di occupato. Quando BF = 1 significa che l'LCD è occupato e non accetterà il comando o i dati successivi. Con BF = 0 l'LCD è pronto per l'elaborazione del comando o dei dati successivi.
Prima di utilizzare l'LCD per scopi di visualizzazione, l'LCD deve essere inizializzato dal circuito di ripristino interno o dall'invio di una serie di comandi per inizializzare l'LCD.
Per inviare comandi si selezionare il registro dei comandi (RS =1);
I passaggi sono:

1. Spostare i dati sulla porta LCD;
2. Selezionare il registro dei comandi;
3. Selezionare l'operazione di scrittura;
4. Inviare il segnale di abilitazione;

5. Attendere che LCD elabori il comando.

```
: 4BM>LCD  Invia i 4 bit più significativi rimasti di TOS
  F0 AND DUP ROT
  D + OR >I2C 1000 DELAY
  8 OR >I2C 1000 DELAY ;

: 4BL>LCD Invia 4 bit meno significativi rimasti di TOS
  F0 AND DUP
  D + OR >I2C 1000 DELAY
  8 OR >I2C 1000 DELAY ;
```

Se RS è HIGH 1 inviamo un Data Signal, se LOW 0 un Istruction Signal.

```
: >LCDL
 DUP 4 RSHIFT 4BL>LCD
 4BL>LCD ;

: >LCDM
  OVER OVER F0 AND 4BM>LCD
  F AND 4 LSHIFT 4BM>LCD ;
```

Per verificare che si tratti di un comando controllo se il 9 bit è 1 (lo shift elimina gli otto precedenti)

```
: IS_CMD
  DUP 8 RSHIFT 1 = ;
```

Controlla se stiamo inviando un comando o un dato a I2C
Il comando ha un 1 in più nel bit più significativo rispetto ai dati
Un input come 101 >LCD sarebbe considerato un COMANDO per cancellare lo schermo 0000 0001 altrimenti se 0 (ad esempio un input come 41 >LCD) sarebbe considerato un DATA per inviare A CHAR (41 in esadecimale) allo schermo

```
: >LCD
  IS_CMD SWAP >LCDM
;
```

Domanda45 : KEPAY PROGETTO.

Per ogni riga si invia un output e per ogni colonna si controllano i valori: se viene letto il bit di rilevamento dell'evento, abbiamo trovato il tasto premuto nel formato RIGA-COLONNA

MATRIX 5x4 (Righe Output + Colonne Input)

GPIO-17 -> Riga-1 (F1-F2-#-*) GPIO-16 -> Colonna-1 (*-SU-GIU-ESC-ENT)

GPIO-18 -> Riga-1 (1-2-3-SU) GPIO-22 -> Colonna-2 (#-3-6-9-DX)

GPIO-23 -> Riga-2 (4-5-6-GIU) GPIO-27 -> Colonna-3 (F2-2-5-8-0)

GPIO-24 -> Riga-3 (7-8-9-ESC) GPIO-10 -> Colonna-4 (F1-1-4-7-SX)

GPIO-25 -> Riga-4 (SX-0-DX-ENT)

Usiamo **GPFEN0** con la word **SETUP_ROWS** per abilitare il rilevamento del fronte di discesa per i pin che controllano le RIGHE scrivendo 1 nelle posizioni dei pin corrispondenti (GPIO-18, 23, 24, 25) HEX (0x03840000) che è (0000 0011 1000 0100 0000 0000 0000 0000) in BIN

I pin RIGA sono impostati come output, i pin colonna sono impostati come input.

Il campo GPFSEL1 viene utilizzato per definire il funzionamento dei pin GPIO-10 - GPIO-19

Il campo GPFSEL2 viene utilizzato per definire il funzionamento dei pin GPIO-20 - GPIO-29

Ogni 3 bit di GPFSEL rappresenta un pin GPIO e usiamo le words: SETUP_IO per settare e: CLEAR_ROWS

per cancellare i GPIO.

La word SETUP_KEYPAD serve ad inizializzare la tastiera.

Domanda46 : FRAMEBUFFER PROGETTO.

IMPOSTAZIONI DEL FRAMEBUFFER SONO DEFINITE SULL' INTERPRETE

FRAMEBUFFER

Indirizzo Base 0x3E8FA000 (3E8FA000 CONSTANT FRAMEBUFFER)

- **Larghezza** 00000400
- **Altezza** 00000300
- **Profondità** 00000020
- **Pixel per riga** 00001000

Dichiarazione dei colori in esadecimale (formato ARGB)

- 00FFFFFF CONSTANT WHITE
- 00000000 CONSTANT BLACK
- 00FF0000 CONSTANT RED
- 00FFFF00 CONSTANT YELLOW
- 0000FF00 CONSTANT GREEN
- 000000FF CONSTANT BLUE

DIM Parametro in Hex

COUNTERH Contatore utilizzato nei cicli per disegnare le linee orizzontali

: +COUNTERH Contatore utilizzato nei cicli per tenere traccia del numero di riga corrente;

: RESETCOUNTERH Azzera il contatore linee orizontali;

NLINE Numero di linea;

: RESETNLINE Restituisce l'indirizzo del punto centrale dello schermo. Coordinata ((larghezza-1)/2, (altezza-1)/2);

: +NLINE Incrementa il numero di linea.

WORD PER MUOVERSI DI PIXEL NELLE DIREZIONI E PER CENTRARE

Restituisce l'indirizzo del punto centrale dello schermo. Coordinata ((larghezza-1)/2, (altezza-1)/2) dato che il le coordinate dei pixel sono comprese tra (0,0) e (-1,-1)

```
( -- addr )
: CENTER FRAMEBUFFER 200 4 * + 180 1000 * + ;
```

Colora, con il colore presente sullo stack, il pixel corrispondente all'indirizzo presente sullo stack, dopodiché punta al pixel a destra .

```
( color addr -- color addr_col+1 )
: RIGHT 2DUP ! 4 + ;
```

Colora, con il colore presente sullo stack, il pixel corrispondente all'indirizzo presente sullo stack, dopodiché punta al pixel in basso.

```
( color addr -- color addr_row+1 )
: DOWN 2DUP ! 1000 + ;
```

Colora, con il colore presente sullo stack, il pixel corrispondente all'indirizzo presente sullo stack, dopodiché punta al pixel a sinistra

```
( color addr -- color addr_col-1 )
: LEFT 2DUP ! 4 - ;
```

*Ripristina il valore di partenza dell'indirizzo a seguito di COUNTERH * 4 spostamenti a destra*

```
( addr_endline_right -- addr )
: RIGHTRESET COUNTERH @ 4 * - ;
```

*Ripristina il valore di partenza dell'indirizzo a seguito di COUNTERH * 4 spostamenti a sinistra*

```
( addr_endline_left -- addr )
: LEFTRESET COUNTERH @ 4 * + ;
```

Disegna una linea verso destra di dimensione pari a 48 pixel

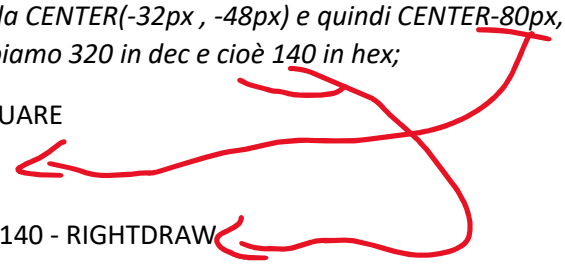
```
: RIGHTDRAW
  BEGIN COUNTERH @ DIM @ < WHILE +COUNTERH RIGHT REPEAT RIGHTRESET RESETCOUNTERH ;
```

Disegna una linea verso sinistra di dimensione pari a 48 pixel

```
: LEFTDRAW
   BEGIN COUNTERH @ DIM @ < WHILE +COUNTERH LEFT REPEAT LEFTRESET RESETCOUNTERH ;
```

Disegna o il simbolo di stop o pulisce la porzione di schermo su cui disegnare.

Partendo da CENTER(-32px , -48px) e quindi CENTER-80px, e poiché ogni spostamento di 1px su una riga vale 4, abbiamo 320 in dec e cioè 140 in hex;



```
: DRAWSQUARE
   80 DIM !
   CENTER 140 - RIGHTDRAW
\ Ciclo che disegna un simbolo di stop di altezza 105 pixel
   BEGIN NLINE @ 70 <
      WHILE
         DOWN RIGHTDRAW
         +NLINE
      REPEAT
   2DROP RESETNLINE
;
```

Partendo da CENTER(-32px , -48px) e quindi CENTER-80px, disegna il simbolo di start.

```
: DRAWSTARTWIND
   GREEN CENTER 80 -
   BEGIN NLINE @ 70 <=
      WHILE
```

Permette di rappresentare le linee del triangolo superiore del simbolo start

```
         NLINE @ 37 <= IF
         NLINE @ DIM !
```

Permette di rappresentare le linee del triangolo inferiore del simbolo start

```
            ELSE
               70 NLINE @ - DIM !
            THEN
```

Disegna una linea verso destra di dimensione variabile e dipendente dal numero di riga \ memorizzato in NLINE.

```
        DOWN RIGHTDRAW

        +NLINE

      REPEAT

   2DROP RESETNLINE

;
```

Domanda47 : I2C PROGETTO.

Ci sono otto master Broadcom Serial Controller (BSC) all'interno di BCM2711, BSC2 e BSC7 sono dedicati all'uso da parte delle interfacce HDMI, qui utilizziamo BSC1 all'indirizzo: 0xFE804000.

Per utilizzare l'interfaccia I2C, basta aggiungere i seguenti offset all'indirizzo del registro BCS1. Ogni registro è lungo 32 bit:

- 0x0 -> Control Register (usato per abilitare gli interrupt, cancellare il FIFO, definire un'operazione di lettura o scrittura e avviare un trasferimento)
- 0x4 -> Status Register (usato per registrare lo stato delle attività, gli errori e le richieste di interruzione)
- 0x8 -> Data Length Register (definisce il numero di byte di dati da trasmettere o ricevere nel trasferimento I2C)
- 0xc -> Slave Address Register (specifica l'indirizzo slave e il tipo di ciclo)
- 0x10 -> Data FIFO Register (utilizzato per accedere al FIFO)
- 0x14 -> Clock Divider Register (usato per definire la velocità di clock della periferica BSC)
- 0x18 -> Data Delay Register (fornisce un controllo accurato sul punto di campionamento/lancio dei dati)
- 0x1c -> Clock Stretch Timeout Register (fornisce un timeout su quanto tempo il master può attende lo slave, così da allungare il timeout, prima di decretarne la caduta)

I pin GPIO-2 (SDA) e GPIO-3 (SCL) devono prendere la ALTERNATIVE FUNCTION 0.

: SETUP_I2C Imposta la ALTERNATIVE FUNCTION 0

: RESET_S Ripristina lo Status Register.

: RESET_FIFO Ripristina FIFO utilizzando.

: SET_SLAVE Imposta l'indirizzo SLAVE.

: STORE_DATA Memorizza i dati nella FIFO.

: SEND Avvia un nuovo trasferimento:

- Il bit 0 è 0 -> Scrivi trasferimento pacchetti
- Il bit 7 è 1 -> Avvia un nuovo trasferimento
- Il bit 15 è 1 -> Il controller BSC è abilitato

: >I2C La parola principale per scrivere 1 byte alla volta.

Domanda48 : TIMER PROGETTO.

Clock Register BASE 003004 + CONSTANT CLO

F4240 CONSTANT SEC indica un secondo e che ha valore 1 000 000 usec in decimale o F4240 in hex.

\ Variabile che memorizza il valore attuale del CLO + 1 secondo

VARIABLE COMP0

VARIABLE TIME_COUNTER

\ Restituisce il valore attuale del registro CLO

(-- clo_value)

: NOW CLO @ ;

\ Setta un ritardo corrispondente al valore presente sullo stack

(delay_sec --)

: DELAY_SEC NOW + BEGIN DUP NOW - 0 <= UNTIL DROP ;

Converte il valore decimal in secondi a hexadecimal per darlo in pasto al tic. La word pone sullo stack il resto e il quoto della divisione per 10 moltiflica il quoto per A (10 in HEX) somma il resto ed effettua successivamente uno swap.

: PARSE_DEC_HEX (n1 -- n3 n2) 10 /MOD A * SWAP + DUP . ." >> SECONDS " ;

Memorizza il valore attuale del CLO + 1 secondo in COMP0

: INC NOW SEC + COMP0 ! ;

Segnala ogni qual volta e passato un secondo confrontando CLO con COMP0

: SLEEPS DECIMAL INC BEGIN NOW COMP0 @ < WHILE REPEAT DUP U. ." | " DROP DECIMAL ;

Words di incremento e decremento contatore

: DECCOUNT TIME_COUNTER @ 1 - TIME_COUNTER ! ;

: INCCOUNT TIME_COUNTER @ 1 + TIME_COUNTER ! ;

Word che imposta un conto alla rovescia in secondi a partire dal n passato fino a zero.

DECIMAL : TIMER PARSE_DEC_HEX TIME_COUNTER ! CR begin TIME_COUNTER @ SLEEPS DECCOUNT TIME_COUNTER @ 0 = until CR

." END EROGATION " CR CR DROP ;

FORTH

Domanda44 : Comportamento runtime in forth : run-time behavior.

Abbiamo usato il termine "tempo di esecuzione" quando ci riferiamo a cose che accadono quando una parola viene eseguita e "tempo di compilazione" quando ci riferiamo a cose che accadono quando una parola viene compilata .

In generale ci sono due classi di parole che si comportano in entrambi i modi. Ai fini di questa discussione, chiameremo queste due classi "parole di definizione" e "parole di compilazione".

Usando la parola che definisce COSTANTE come esempio, quando diciamo

```
80 MARGINE COSTANTE
```

stiamo eseguendo il *comportamento in fase di compilazione* di CONSTANT; cioè, CONSTANT sta compilando una nuova voce di dizionario di tipo costante chiamata MARGIN e memorizzando il valore 80 nel suo campo parametro. Ma quando diciamo

```
MARGINE
```

stiamo eseguendo il *comportamento di runtime* di CONSTANT; cioè, CONSTANT sta spingendo il valore 80 nello stack.

L'altro tipo di parola che possiede un duplice comportamento è la "parola di compilazione". Una parola di compilazione è una parola che usiamo all'interno di una definizione di due punti e che in realtà fa qualcosa durante la compilazione di quella definizione.

Ad esempio la parola ."", che in fase di compilazione compila una stringa di testo nella voce del dizionario con il conteggio in primo piano e in fase di esecuzione la digita.

In fase di compilazione, CREATE prende un nome dal flusso di input e crea un'intestazione del dizionario per esso.

In fase di esecuzione, CREATE inserisce l'indirizzo del corpo di EXAMPLE nello stack.

Cosa succede se utilizziamo CREATE all'interno di una definizione? Considera questo esempio, che è la definizione di VARIABLE:

```
: VARIABILE ( -- ) CREATE 0 , ;
```

Quando eseguiamo VARIABLE come in

```
ORANGES VARIABILI
```

Stiamo usando indirettamente CREATE per creare una testa di dizionario con il nome ORANGES e un xt che punta al codice di runtime di CREATE. Quindi stiamo allocando una cella per la variabile stessa (con "0 ,").

Poiché il comportamento in fase di esecuzione di una variabile è identico a quello di una parola definita da CREATE, VARIABLE non ha bisogno di avere un codice di runtime proprio, può usare il codice di runtime di CREATE.

Come specifichiamo un diverso comportamento in fase di esecuzione in una parola che definisce? Usando la parola DOES>, come mostrato qui:

```
: DEFINING-WORD CREATE (operazioni in fase di compilazione)
   DOES> (operazioni in fase di esecuzione) ;
```

Per illustrare, la seguente potrebbe essere una definizione valida per CONSTANT (sebbene in realtà CONSTANT sia solitamente definito nel codice macchina):

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

Per vedere come funziona questa definizione, immagina di usarla per definire una costante denominata TROMBONES, in questo modo:

```
76 TROMBONES COSTANT
```

Parte del tempo di compilazione:

CREARE

Crea una nuova voce del dizionario (ad es. TROMBONI)

,

Compila il valore (ad esempio, 76) per la costante dallo stack nel campo del parametro della costante.

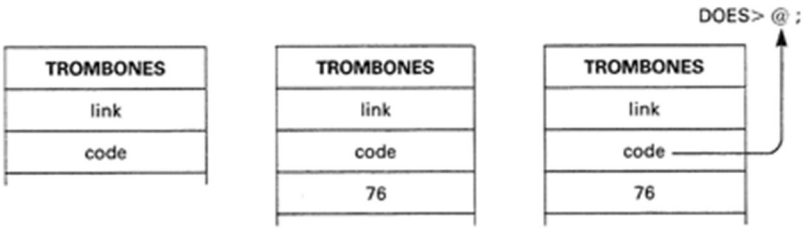
Porzione di esecuzione:

DOES >

Contrassegna la fine del comportamento in fase di compilazione e l'inizio del comportamento in fase di esecuzione. In fase di esecuzione, DOES> lascerà l'indirizzo del corpo della parola definita nello stack.

@

Recupera il contenuto della costante, usando l'indirizzo del corpo che sarà nello stack in fase di esecuzione.



Le parole che *precedono* DOES> specificano cosa farà lo stampo; le parole che *seguono* DOES> specificano cosa farà il prodotto dello stampo (riferito all'analogia delle saliere).

Domanda42 : Execution Token – esecuzione vettoriale in forth.

L'interprete di testo, il cui nome è INTERPRET, preleva le parole dal flusso di input e cerca di trovare le loro definizioni nel dizionario. Se trova una parola, INTERPRET la fa eseguire. La parola '(pronunciato tick) trova una definizione nel dizionario e restituisce il suo token di esecuzione.

L'idea dell'esecuzione vettoriale è davvero piuttosto semplice. Invece di eseguire direttamente una definizione per esempio: ' SALUTO EXECUTE possiamo eseguirlo indirettamente mantenendo il suo *XT* in una variabile, quindi eseguendo il contenuto della variabile, in questo modo:

```
' SALUTO  pointer !
pointer @ EXECUTE


VARIABLE 'aloha  ' HELLO 'aloha !
: SAY  ' 'aloha ! ;
```

Il vantaggio è che possiamo cambiare il puntatore in un secondo momento, in modo che una singola parola possa eseguire cose diverse in momenti diversi.

È una convenzione Forth usare questo prefisso per i puntatori di esecuzione vettoriali.

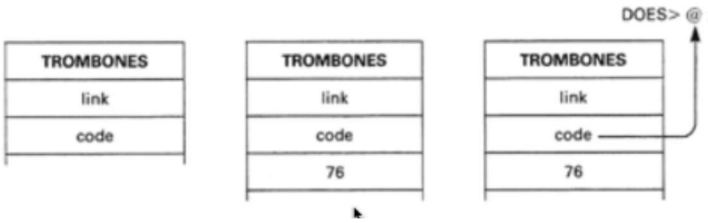
Il segno di TICK va sempre alla parola successiva nel *flusso di input*. se vogliamo che tick usi la parola successiva nella definizione dobbiamo usare la parola ['] (bracket-tick-bracket) invece di tick.

Domanda50 : Variabili, costanti e array in Forth.

Istruzione CREATE

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: questa istruzione aggiunge un comportamento (con il fetch verrà individuare l'indirizzo del data field)
: VARIABLE (--) CREATE 0 ,
: CONSTANT (n --) CREATE , DOES>@

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.

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
se scriviamo 1 CELLS ALLOT, verrà aggiunta una nuova cella
gli ultimi due comandi sono esattamente gli stessi.
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"
4 DATE code spazio_1

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"

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

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

NUCLEO

Domanda38 : NUCLEO64.

Questa scheda è dotata di un STM32 ARM con MCU(modulo di controllo integrato) che fornisce un'ottima combinazione tra prestazioni, consumo energetico e funzionalità. Ha integrato un supporto di connettività ArduinoTM Uno V3, uno standard ST Morpho e un'interfaccia integrata per il debug ST-LINK/V2-1.

Il SoC è dotato di 8 porte GPIO con 16 pin: GPIOxx=A..H. Per la programmazione possiamo usare Mecrisp-Stellaris che è un ambiente Forth, IDE Proprietario di ST o altro.

Nel STM32F446xx, il sistema principale è costituito da un **bus AHB multistrato a 32 bit** con matrice che interconnette: 7 Master e 7 Slave:

MASTER: Cortex®-M4 con nucleo FPU I-bus, D-bus e S-bus, Bus di memoria DMA1, Bus di memoria DMA2, Bus periferico DMA2 e un Bus USB OTG HS DMA.

SLAVE: Memoria flash interna ICode bus, Memoria Flash interna DCode bus, SRAM1 interna principale (112 KB), SRAM2 interna ausiliaria (16 KB), Periferiche AHB1 inclusi bridge da AHB a APB e periferiche APB, Periferiche AHB2 e Controller di memoria flessibile FMC / QUADspi.

Il processore ha una mappa di memoria fissa che fornisce fino a 4 GB di memoria indirizzabile. Memoria di programma, memoria dati, registri e porte I/O sono organizzati all'interno dello stesso indirizzo lineare di 4

Table 4. Flash module organization

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Gbyte di spazio. I byte sono codificati in memoria in formato Little Endian. Il byte con il numero più basso in una parola è considerato il byte meno significativo della parola e il numero più alto byte quello più significativo. Lo spazio di memoria indirizzabile è suddiviso in 8 blocchi, di 512 Mbyte ciascuno. Tutte le aree di memoria non sono allocate sul chip SoC.

Domanda35 : Bit Banding.

È una mappatura della memori: infatti nella **bit-band region map** ogni parola è allocata con un *alias nella bit-band region* su un singolo bit. La **bit-band region** occupa l'1° Mbyte più basso della **SRAM** e della **peripheral memory region**.

La mappa di memoria ha due alias region da 32 Mbyte che mappano due **bit-band region** da 1 Mbyte ciascuna. L'accesso all'*alias bit-band region* della **SRAM** da 32 Mbyte è mappato dalla **bit-band region** SRAM da 1 Mbyte.

L'accesso all' *alias bit-band region* **periferal** da 32 MB è mappato dalla **bit-band region** perifheral da 1 Mbyte.

Domanda39 : GPIO del NUCLEO64.

Le caratteristiche di connessione elettrica di ciascun pin possono essere configurate tramite un valore a due bit nel registro PUPDR della porta:

7.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..H)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Il registro IDR di contiene **i valori di input digitali** dei suoi pin per i specifici GPIO; è necessario leggere l'intera parola per leggere anche un singolo valore di pin:

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 ODR di una porta **dà accesso in lettura e scrittura ai valori di uscita digitali** dei suoi pin; è necessario accedere all'intera word per modificare il valore anche di un singolo pin.

7.4.6 GPIO port output data register (GPIOx_ODR) (x = A..H)

Address offset: 0x14

Reset value: 0x0000 0000

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..H).

Pin di uscita Bit Set/Reset. I registri GPIOx_BSRR consentono di controllare le uscite digitali senza operazioni di mascheramento dei bit; questi registri funzionano in modo simile ai registri GPSETn/GPCLRN nei SOC Raspberry Pi.

In modalità **Funzione alternativa**, un pin GPIO può essere configurato tramite **l'AFRL (inferiore 8 pin) o AFRH (8 pin più alti)** per esprimere una funzione specifica.

Table 11. Alternate function

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11/CEC	I2C1/2/3/4/CEC	SPI1/2/3/4	SPI2/3/4/SAI1	SPI2/3/USART1/2/3/UART5/SPDIFRX	SAI/USART6/UART4/5/SPDIFRX	CAN1/2 TIM12/13/14/QUADSPI	SAI2/QUADSPI/OTG2_HS/OTG1_FS	OTG1_FS	FMC/SDIO/OTG2_FS	DCMI	-	SYS
Port A	PA0	-	TIM2_CH1/TIM2_ETR	TIM5_CH1	TIM8_ETR	-	-	-	USART2_CTS	UART4_TX	-	-	-	-	-	-	EVENTOUT
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-	USART2_RTS	UART4_RX	QUADSPI_BK1_IO3	SAI2_MCLK_B	-	-	-	-	EVENTOUT
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-	USART2_TX	SAI2_SCK_B	-	-	-	-	-	-	EVENTOUT
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	-	SAI1_FS_A	USART2_RX	-	-	OTG_HS_ULPI_D0	-	-	-	-	EVENTOUT
	PA4	-	-	-	-	-	SPI1_NSS/I2S1_WS	SPI3_NSS/I2S3_WS	USART2_CK	-	-	-	-	OTG_HS_SOF	DCMI_HSYNC	-	EVENTOUT
	PA5	-	TIM2_CH1/TIM2_ETR	-	TIM8_CH1N	-	SPI1_SCK/I2S1_CK	-	-	-	-	OTG_HS_ULPI_CK	-	-	-	-	EVENTOUT
	PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO	I2S2_MCK	-	-	TIM13_CH1	-	-	-	DCMI_PIXCLK	-	EVENTOUT
	PA7	-	TIM1_CH1N	TIM3_CH2	TIM8_CH1N	-	SPI1_MOSI/I2S1_SD	-	-	-	TIM14_CH1	-	-	FMC_SDNWE	-	-	EVENTOUT
	PA8	MCO1	TIM1_CH1	-	-	I2C3_SCL	-	-	USART1_CK	-	-	OTG_FS_SOF	-	-	-	-	EVENTOUT
	PA9	-	TIM1_CH2	-	-	I2C3_SMBA	SPI2_SCK/I2S2_CK	SAI1_SD_B	USART1_TX	-	-	-	-	-	DCMI_D0	-	EVENTOUT
	PA10	-	TIM1_CH3	-	-	-	-	-	USART1_RX	-	-	OTG_FS_ID	-	-	DCMI_D1	-	EVENTOUT
	PA11	-	TIM1_CH4	-	-	-	-	-	USART1_CTS	-	CAN1_RX	OTG_FS_DM	-	-	-	-	EVENTOUT
	PA12	-	TIM1_ETR	-	-	-	-	-	USART1_RTS	SAI2_FS_B	CAN1_TX	OTG_FS_DP	-	-	-	-	EVENTOUT
	PA13	JTMS-SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA14	JTCK-SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA15	JTDI	TIM2_CH1/TIM2_ETR	-	-	HDMI_CEC	SPI1_NSS/I2S1_WS	SPI3_NSS/I2S3_WS	-	UART4_RTS	-	-	-	-	-	-	EVENTOUT

Le definizioni **HAL** (che racchiudono tutte le definizioni hardware) possono essere rese chiare per raccogliere conoscenze consolidate sui dispositivi in modo eseguibile e testabile.

PRESENTAZIONE PROGETTO

Domanda49 : Introduzione.

Questo progetto nasce dalla prerogativa di creare qualcosa di utile.

Ho cercato di inserire più argomenti possibili di quelli trattati a lezione.

Ho utilizzato il Raspberry pi 4B come target con il PIJForthOs come interprete.

Ho usato il metodo di programmazione dinamica su target (quindi ho un sistema Bare Metal).

Il sistema serve a gestire in maniera automatizzata la temporizzazione degli attuatori all'interno della serra e permette all'utente di avviare il processo, e di modificarne i parametri.

In questo progetto gli attuatori sono la **ventola** per l'areazione e la **lampada** per l'illuminazione della pianta.

Abbiamo inoltre un **pulsante** di accensione, il **monitor**, display **LCD** e i **led** per osservare le varie fasi del ciclo del sistema, ed una **tastiera** per inserire i parametri.

Una volta che abbiamo caricato il file sorgente tramite terminale, il sistema è già in attesa della pressione del tasto di accensione, il quale avvia il ciclo principale.

Quando premo il tasto di accensione il sistema si avvia. Sul display lcd appare il messaggio di benvenuto e il nome del dispositivo e sul monitor il logo italia.

A questo punto il sistema passa in stato di arresto: sul display abbiamo il simbolo rosso di stop ed il led rosso acceso; inoltre appare la richiesta di inserire i parametri temporali.

L'utente adesso deve inserire il valore in secondi da 01 a 99 per la temporizzazione degli attuatori: le prime due cifre per il tempo di luce e gli altri due per il tempo di ventilazione.

La temporizzazione sfrutta il timer interno del Raspberry verificando ogni quando quel valore viene incrementato di un secondo, semplicemente confrontando l'incremento con il campione "secondo in HEX".

Il sistema con tali parametri effettua un ciclo di quattro scambi (illuminazione ventilazione).

Parte lo stato di illuminazione: quindi si accende la luce, sul monitor compare il simbolo di play giallo, il led giallo si accende, sul display lcd compare la scritta "system light" e sul terminale parte il conteggio in secondi.

Finito il tempo per questo stato parte lo stato di ventilazione: quindi si accende la ventola, sul monitor compare il simbolo di play verde, il led verde si accende, sul display lcd compare la scritta "system wind" e sul terminale parte il conteggio in secondi.

Come dicevo prima il ciclo dura 4 scambi (ovviamente questo numero è compatibile con questa dimostrazione così come la scelta dei tempi in secondi).

In caso di anomalie del sistema è previsto un tasso di "arresto d'emergenza": il tasto ESC, che può essere premuto in fase d'inserimento per uscire dal ciclo e lasciare il sistema in fase di arresto.