**Introduzione**

La stragrande maggioranza dei computer in uso è poco visibile. Alcuni esempi:

* far funzionare il motore, i freni, le cinture di sicurezza, l'airbag e l'impianto audio dell'automobile;
* codificare digitalmente la voce e costruire un segnale radio per inviarlo dal telefono cellulare a una stazione base;
* gestire stampanti;
* comandare i robot in una fabbrica, la produzione di energia in una centrale elettrica, i processi in un impianto chimico e i semafori in una città;
* controllare aerei e treni;
* eseguire molti altri compiti "invisibili".

Questi computer meno visibili sono chiamati **sistemi embedded** e il software che eseguono è chiamato **software embedded**.

Sebbene i sistemi embedded siano in uso fin dagli anni '70, per la maggior parte della loro storia sono stati visti semplicemente come piccoli computer. Il problema ingegneristico principale era quello di far fronte a risorse limitate (potenza di elaborazione limitata, fonti di energia limitate, memorie piccole, ecc.). La sfida ingegneristica consisteva nell'ottimizzare i progetti. Poiché tutti i progetti traggono vantaggio dall’ottimizzazione, la disciplina non si distingueva da qualsiasi altro aspetto dell'informatica. Doveva solo essere più aggressiva nell'applicare le stesse tecniche di ottimizzazione.

Di recente, la comunità ha capito che le sfide principali dei sistemi embedded derivano dalla loro interazione con i processi fisici e non dalle loro risorse limitate. Il termine **sistemi cyber-fisici** (CPS) è stato coniato per riferirsi all'integrazione del calcolo con i processi fisici.

Nei CPS, i computer e le reti integrate monitorano e controllano i processi fisici, di solito con anelli di retroazione in cui i processi fisici influenzano i calcoli e viceversa. La progettazione di tali sistemi, quindi, richiede la comprensione delle dinamiche congiunte di computer, software, reti e processi fisici. È proprio lo studio della dinamica congiunta che contraddistingue questa disciplina.

Quando si studiano le CPS, emergono alcuni problemi chiave che sono rari nel cosiddetto **calcolo general-purpose**. Ad esempio, nel software di uso generale, il tempo necessario per eseguire un'attività è un problema di prestazioni, non di correttezza. Non è scorretto impiegare più tempo per svolgere un compito. È semplicemente meno e quindi meno prezioso. In CPS, il tempo necessario per eseguire un'attività può essere critico per il corretto funzionamento del sistema.

I processi fisici sono composizioni di molte cose che avvengono contemporaneamente, a differenza dei processi software, che sono profondamente radicati in passi sequenziali (procedurali). Nel mondo fisico, invece, i processi sono raramente procedurali.

I processi fisici sono composizioni di molti processi paralleli. Misurare e controllare la dinamica di questi processi orchestrando le azioni che li influenzano sono i compiti principali dei sistemi embedded. Di conseguenza, la concorrenza è intrinseca alle CPS. Molte delle sfide tecniche nella progettazione e nell'analisi del software embedded derivano dalla necessità di collegare una semantica intrinsecamente sequenziale con un mondo fisico intrinsecamente concorrente.

I meccanismi con cui il software interagisce con il mondo fisico stanno cambiando rapidamente. Oggi la tendenza è verso sensori e attuatori "intelligenti", dotati di microprocessori, interfacce di rete e software che consentono l'accesso remoto ai dati del sensore e l'attivazione remota dell'attuatore. Chiamato in vari modi, Internet of Things (IoT), Industry 4.0, the Industrial Internet, Machine-to-Machine (M2M), the Internet of Everything, the Smarter Planet, TSensors (Trillion Sensors) or The Fog , la visione è quella di una tecnologia che connette profondamente il nostro mondo fisico con il mondo dell'informazione. Nel mondo IoT, le interfacce tra questi mondi sono ispirate e derivate dall'informatica, in particolare dalla tecnologia web. Le interfacce IoT sono comode, ma non ancora adatte a interazioni strette tra i due mondi, in particolare per il controllo in tempo reale e per i sistemi critici per la sicurezza. Le interazioni strette richiedono ancora una progettazione tecnicamente complessa e di basso livello.

I progettisti di software embedded sono costretti a confrontarsi con controllori di interrupt, architetture di memoria, programmazione a livello assembly (per sfruttare istruzioni specializzate o per controllare con precisione i tempi), progettazione di driver di dispositivi, interfacce di rete e strategie di schedulazione, piuttosto che concentrarsi sulla specificazione del comportamento desiderato.

Nell'informatica general-purpose, la varietà di instruction set architectures è oggi limitata, con l'architettura Intel x86 di Intel che domina su tutte le architetture. Non esiste una tale predominanza nell'informatica embedded.

Quando vengono installati in un prodotto, i processori embedded hanno in genere una funzione dedicata. Non sono chiamati a svolgere funzioni arbitrarie con software definito dall'utente. Di conseguenza, i processori possono essere più specializzati. Vantaggi:

* possono consumare molta meno energia, e di conseguenza essere utilizzabili con piccole batterie per lunghi periodi di tempo;
* possono includere hardware specializzato per eseguire operazioni che sarebbero costose da eseguire su un hardware generico, come ad esempio l'analisi delle immagini.

Quando si valutano i processori, è importante capire la differenza tra:

* **Instruction set architecture** (ISA): le istruzioni che il processore può eseguire e alcuni vincoli strutturali (come la dimensione delle parole). x86 è una ISA.
* **Una realizzazione di processore o un chip**: un pezzo di silicio venduto da un fornitore di semiconduttori. Esistono molte realizzazioni dell'ISA x86.

Un'ISA è un'astrazione condivisa da molte realizzazioni. Una singola ISA può apparire in molti chip diversi, spesso realizzati da produttori diversi e spesso con profili di prestazioni molto diversi.

Il vantaggio di condividere un'ISA in una famiglia di processori è che gli strumenti software, che sono costosi da sviluppare, possono essere condivisi, e (a volte) gli stessi programmi possono essere eseguiti correttamente su più realizzazioni. Quest'ultima proprietà, tuttavia, è piuttosto insidiosa, dal momento che un'ISA non include normalmente alcun vincolo sulla tempistica. Quindi, sebbene un programma possa essere eseguito logicamente allo stesso modo su più chip, il comportamento del sistema può essere radicalmente diverso quando il processore è incorporato in un sistema cyber-fisico.

Data l'enorme varietà di applicazioni embedded, esiste una grande varietà di processori utilizzati. Si va da dispositivi molto piccoli, lenti, economici e a bassa potenza, a dispositivi ad alte prestazioni e per scopi speciali.

Un **microcontrollore** (μC) è un piccolo computer su un singolo circuito integrato che consiste in un'unità di elaborazione centrale (CPU) relativamente semplice combinata con dispositivi periferici, come ad esempio memorie, dispositivi di I/O e timer.

Più della metà di tutte le CPU vendute nel mondo sono microcontrollori, anche se tale affermazione è difficile da dimostrare. I microcontrollori più semplici funzionano con parole a 8 bit e sono adatti per applicazioni che richiedono piccole quantità di memoria e semplici funzioni logiche (rispetto a funzioni aritmetiche ad alte prestazioni). Possono consumare quantità di energia estremamente ridotte e spesso includono una modalità di sospensione che riduce il consumo di energia a nanowatt. È stato dimostrato che esistono nodi di reti di sensori e dispositivi di sorveglianza, in grado di funzionare con una piccola quantità di energia.

I microcontrollori possono essere molto elaborati. L'**Intel Atom**, ad esempio, è una famiglia di CPU x86 utilizzata principalmente nei netbook e in altri piccoli computer portatili. Questi processori sono stati progettati per consumare relativamente poca energia senza perdere troppe prestazioni rispetto ai processori utilizzati in computer di fascia più alta. Sono adatti per alcune applicazioni embedded e per i server in cui il raffreddamento è problematico. **Geode** di AMD è un altro esempio di processore che si trova vicino al labile confine tra i processori di uso generale e i microcontrollori.

Il Motorola 6800 e l'Intel 8080 sono microcontrollori a 8-bit apparsi sul mercato nel 1974. I discendenti di queste architetture sopravvivono oggi, ad esempio sotto forma di Freescale 6811.

Lo Zilog Z80 è un discendente pienamente compatibile dell'8080, che è diventato uno dei microcontrollori più prodotti e utilizzati di tutti i tempi. Un derivato dello Z80 è il Rabbit 2000, progettato da Rabbit Semiconductor.

L'Intel 8051 è un microcontrollore a 8 bit sviluppato da Intel nel 1980. L'ISA dell'8051 è oggi supportato da molti fornitori, tra cui Atmel, Infineon Technologies, Dallas Semiconductor, NXP, ST Microelectronics, Texas Instruments, e Cypress Semiconductor.

Il microcontrollore Atmel AVR a 8 bit, sviluppato da Atmel nel 1996, è stato uno dei primi microcontrollori a utilizzare la memoria flash on-chip per la memorizzazione dei programmi. Sebbene Atmel affermi che AVR non è un acronimo, si ritiene che l'architettura sia stata progettata da due studenti dell'Istituto norvegese di tecnologia, Alf-Egil Bogen e Vegard Wollan, quindi potrebbe essere nata come come RISC di Alf e Vegard.

Molti microcontrollori a 32 bit implementano una qualche variante del set di istruzioni ARM, sviluppato da ARM Limited. ARM era originariamente l'acronimo di Advanced RISC Machine e prima ancora di Acorn RISC Machine ma oggi è semplicemente ARM. I processori che implementano l'ISA ARM sono ampiamente utilizzati nei telefoni cellulari per realizzare le funzioni dell'interfaccia utente e in molti altri sistemi embedded. I produttori di semiconduttori concedono in licenza il set di istruzioni ad ARM Limited e producono i propri chip. Altre architetture di microcontrollori embedded degne di nota sono:

* il Motorola ColdFire (poi Freescale ColdFire)
* Hitachi H8 e SuperH
* il MIPS (originariamente sviluppato da un team guidato da John Hennessy dell'Università di Stanford)
* il PIC (originariamente Programmable Interface Controller, di Microchip Technology)
* il PowerPC (creato nel 1991 da un'alleanza tra Apple, IBM e Motorola)
* RISC-V (Open Source)

**Reduced Instruction Set Computer** (in [acronimo](https://it.wikipedia.org/wiki/Acronimo) **RISC**), nell'[elettronica digitale](https://it.wikipedia.org/wiki/Elettronica_digitale), indica un'idea di [progettazione](https://it.wikipedia.org/wiki/Progettazione) di architetture per [microprocessori](https://it.wikipedia.org/wiki/Microprocessore) 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](https://it.wikipedia.org/wiki/Set_di_istruzioni) in tempi minori rispetto a una architettura [CISC](https://it.wikipedia.org/wiki/CISC).

**Complex Instruction Set Computer** (**CISC**), in [elettronica digitale](https://it.wikipedia.org/wiki/Elettronica_digitale), indica un'architettura per [microprocessori](https://it.wikipedia.org/wiki/Microprocessore) formata da un [set di istruzioni](https://it.wikipedia.org/wiki/Set_di_istruzioni) contenente istruzioni in grado di eseguire operazioni complesse come la lettura di un dato in memoria, la sua modifica e il suo salvataggio direttamente in memoria tramite una singola istruzione. Il termine è nato per distinguere l'altro paradigma dominante nei microprocessori, il paradigma [Reduced Instruction Set Computer](https://it.wikipedia.org/wiki/Reduced_Instruction_Set_Computer) (RISC).

**Processori DSP**

Molte applicazioni embedded eseguono una buona parte dell'elaborazione dei segnali. Un segnale è un insieme di misure campionate del mondo fisico, tipicamente prese a una frequenza regolare chiamata frequenza di campionamento. Un'applicazione per il controllo del movimento, ad esempio, può leggere informazioni sulla posizione o sulla localizzazione dai sensori a frequenze di campionamento che vanno da pochi Hertz (Hz o campioni al secondo) a qualche centinaio di Hertz. I segnali audio vengono campionati a frequenze che vanno da 8.000 Hz (o 8 kHz, la frequenza di campionamento utilizzata nella telefonia per la voce) a 44,1 kHz (la frequenza di campionamento dei CD). Le applicazioni a ultrasuoni (come l'imaging medico) e le applicazioni musicali ad alte prestazioni possono campionare i segnali sonori a velocità molto più elevate.

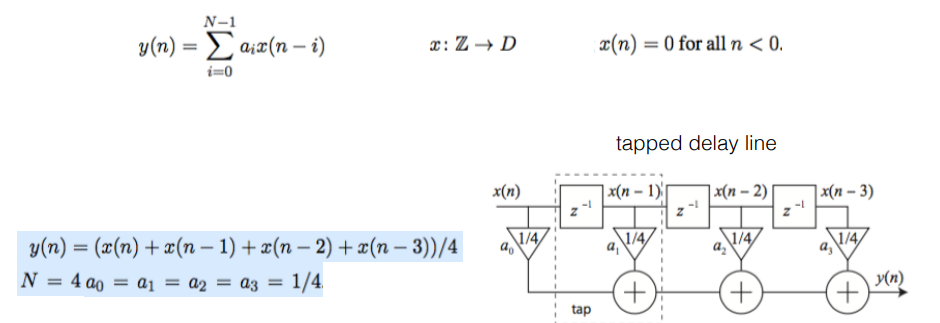
Il video utilizza in genere frequenze di campionamento di 25 o 30 Hz per i dispositivi consumer e frequenze molto più elevate per le applicazioni di misura speciali. Ogni campione, ovviamente contiene un'intera immagine (chiamata fotogramma), che a sua volta ha molti campioni (chiamati pixel) distribuiti nello spazio e non nel tempo. Le applicazioni radio definite dal software hanno frequenze di campionamento che possono variare da centinaia di kHz (per l'elaborazione in banda base) a diversi GHz (miliardi di Hertz). Altre applicazioni embedded che fanno un uso intensivo dell'elaborazione del segnale sono quelle relative a giochi interattivi, radar, sonar e sistemi di imaging LIDAR (Light Detection and Ranging); sistemi di imaging; analisi video (l'estrazione di informazioni dai video, ad esempio per la sorveglianza), sistemi di assistenza alla guida per automobili; elettronica medica e strumentazione scientifica.

Le architetture informatiche specializzate per l'elaborazione dei segnali esistono da tempo. I microprocessori DSP a singolo chip sono comparsi all'inizio degli anni '80. Le prime applicazioni di questi dispositivi comprendevano modem di dati a banda vocale, sintesi vocale, audio per i consumatori, grafica, e controller di unità disco. Le caratteristiche principali dei DSP includono un'unità di multiply-accumulate hardware; diverse varianti dell'architettura Harvard (per supportare più fetch simultanei di dati e programmi) e modalità di indirizzamento che supportano l'incremento automatico, buffer circolari e l'indirizzamento con inversione di bit (quest'ultimo per supportare il calcolo della FFT).

La maggior parte supporta precisioni di dati a virgola fissa di 16-24 bit, in genere con accumulatori molto più ampi (40-56 bit), in modo da poter eseguire un gran numero di istruzioni successive di istruzioni di moltiplicazione-accumulazione successive senza overflow. Alcuni DSP con hardware in virgola mobile non hanno dominato il mercato. I DSP sono difficili da programmare rispetto alle architetture RISC, soprattutto a causa di istruzioni specializzate complesse, pipeline esposta al programmatore e un'architettura di memoria asimmetriche. Fino alla fine degli anni anni '90, questi dispositivi erano quasi sempre programmati in linguaggio assembly. Ancora oggi, i programmi C fanno largo uso di librerie che sono in linguaggio assembly per sfruttare le caratteristiche più esoteriche delle architetture.

**Finite Impulse Response (FIR)**

Un algoritmo canonico di elaborazione del segnale, utilizzato in qualche forma in tutte le applicazioni sopra descritte, è il filtraggio a risposta impulsiva finita (FIR). La forma più semplice di questo algoritmo è semplice, ma ha profonde implicazioni per l'hardware. Nella forma più semplice, un segnale di ingresso x consiste in una sequenza molto lunga di valori numerici, così lunga che ai fini della progettazione dovrebbe essere considerata infinita. Un tale ingresso può essere modellato come una funzione x: N → D, dove D è un insieme di valori di un certo tipo di dati. Per esempio, D potrebbe essere l'insieme di tutti i numeri interi a 16 bit, nel qual caso x(0) è il primo valore di ingresso (un numero intero a 16 bit), x(1) è il secondo valore di ingresso, ecc. Per comodità matematica, si può aumentare questo valore a x: Z → D definendo x(n) = 0 per tutti gli n < 0. Per ogni valore di ingresso x(n), un filtro FIR deve calcolare un valore di uscita y(n) secondo la formula,



dove N è la lunghezza del filtro FIR e i coefficienti ai sono chiamati valori di tap. Da questa formula si può capire perché è utile aumentare il dominio della funzione x, poiché il calcolo di y(0), ad esempio, coinvolge i valori x(-1), x(-2), ecc.

Questo diagramma può essere letto come un diagramma di flusso di dati. Per ogni n ∈ N, ogni componente del diagramma consuma un valore di ingresso da ogni percorso di ingresso e produce un valore di uscita su ogni percorso di uscita. Le caselle etichettate z-1 sono ritardi unitari. Il loro compito è quello di produrre sul percorso di uscita il valore precedente dell'ingresso (o un valore iniziale se non c'era un ingresso precedente). I triangoli moltiplicano il loro ingresso per una costante e i cerchi sommano i loro ingressi.

La velocità con cui i valori di ingresso x(n) vengono forniti e devono essere elaborati è detta frequenza di campionamento. Se si conoscono la frequenza di campionamento e N, è possibile determinare il numero di operazioni aritmetiche da calcolare al secondo.

Supponiamo che un filtro FIR campioni alla frequenza di 1 MHz (un milione di campioni al secondo) e che N = 32. Le uscite devono essere calcolate a una velocità di 1 MHz e ogni uscita richiede 32 moltiplicazioni e 31 addizioni. Un processore deve essere in grado di sostenere una velocità di calcolo di 63 milioni di operazioni aritmetiche al secondo per implementare questa applicazione. Per sostenere la velocità di calcolo: l'hardware aritmetico deve essere sufficientemente veloce; i meccanismi per l'inserimento e l'estrazione dei dati dalla memoria e per l'inserimento e il disinserimento dei dati dal chip devono essere sufficientemente veloci.

Un filtro FIR necessita di una linea di ritardo. Un'implementazione ingenua prevede l'allocazione di un array in memoria e ogni volta che arriva un campione in ingresso, si sposta ogni elemento dell'array nella posizione superiore per fare spazio per il nuovo elemento nella prima posizione. Ciò comporterebbe un enorme spreco di banda di memoria. Un approccio migliore è quello di utilizzare un buffer circolare.

Immagine che contiene testo, orologio

Descrizione generata automaticamente

* L'implementazione accetta prima un nuovo valore di input e poi calcola la sommatoria a ritroso, iniziando dal termine i=N-1, nell’esempio N=8.
* Quando arriva l’n-esimo input, il valore di p è pi compreso {0,1,…7}.
* Il programma scrive il nuovo input x(n) nella locazione data da p e incrementa p calcolando p = pi+1.
* Tutti i calcoli aritmetici su vengono eseguiti con modulo 8.
* Il calcolo del filtro FIR viene quindi letto dalla posizione x(n-7) dalla locazione p = pi+1 e moltiplicato per a7.
* Il risultato viene memorizzato in un registro di accumulo (precedentemente azzerato). Il registro p si incrementa nuovamente di uno, settando p = pi+2.
* Successivamente legge x(n-6) dalla locazione p = pi+2, la moltiplica per a6, e aggiunge il risultato all'accumulatore.
* Continua fino a leggere x(n) dalla locazione p = pi+8, la stessa in cui è stato scritto l'ultimo ingresso x(n), e moltiplica il valore per a0 , e aggiunge il risultato all'accumulatore.
* Incrementa p, ponendo p = pi+9= pi+1
* Al termine di questa operazione, il valore di p è pi+1 , che indica la posizione in cui deve essere scritto il prossimo ingresso x(n+1).

Un'immagine può essere modellata in modo analogo come una funzione. Un pixel (o elemento dell'immagine) è un campione che rappresenta il colore e l'intensità di un punto in un'immagine. Esistono molti modi per farlo, ma tutti utilizzano uno o più valori numerici per ogni pixel. I set e dipendono dalla risoluzione dell'immagine.

ATSC è un insieme di standard sviluppati dall'Advanced Television Systems Committee. DVB-T è lo standard europeo . L'ATSC supporta una serie di frame rate che vanno da poco meno di 24 Hz a 60 Hz e una serie di risoluzioni. Il video ad alta definizione secondo lo standard ATSC supporta, ad esempio, una risoluzione di 1080 per 1920 pixel a una frequenza di fotogrammi di 30 Hz. Questa risoluzione è chiamata 1080p nel settore. Oggi le apparecchiature video professionali raggiungono una risoluzione quattro volte superiore (4320 x 7680). La frequenza dei fotogrammi può essere molto superiore a 30 Hz. Frequenze di fotogrammi molto elevate sono utili per catturare fenomeni estremamente veloci al rallentatore.

**Programmable Logic Controllers (PLC).**

Un controllore logico programmabile (PLC) è una forma specializzata di microcontrollore in un pacchetto robusto con interfacce di I/O adatto al controllo industriale. I PLC sono spesso programmati utilizzando la logica ladder, una notazione grafica originariamente utilizzata per specificare la logica costruita con relè e interruttori. Un relè è un interruttore il cui contatto è controllato da una bobina. Quando una tensione viene applicata alla bobina, il contatto si chiude, consentendo il passaggio di corrente attraverso il relè. Interconnettendo i contatti e le bobine, i relè possono essere utilizzati per costruire controllori digitali che seguono schemi specifici. Nella notazione comune, un contatto è rappresentato da due barre verticali e una bobina da un cerchio.

* Il diagramma è composto da due pioli.
* La bobina del motore sul piolo inferiore accende o spegne un motore.
* I contatti di Start e Stop rappresentano interruttori a pulsante.
* L'avvio è un contatto normalmente aperto.
* Il contatto Stop è normalmente chiuso, indicato dalla barra, cioè diventa aperto quando l'operatore preme il pulsante.
* Quando l'operatore preme Start, la corrente passa alla bobina Run, causando la chiusura di entrambi i contatti Run.
* Il motore funziona anche dopo il rilascio del pulsante di avvio.
* Quando l'operatore preme Stop, la corrente si interrompe ed entrambi i contatti di marcia si aprono, causando l'arresto del motore.
* I contatti cablati in parallelo eseguono una funzione logica OR, mentre i contatti cablati in serie eseguono una funzione logica AND.
* Il piolo superiore è dotato di retroazione; il significato del piolo è una soluzione a punto fisso dell'equazione logica implicita nel motore.

**Graphics Processors**

Un'unità di elaborazione grafica (GPU) è un processore specializzato progettato appositamente per eseguire i calcoli richiesti dal rendering grafico. I processori di questo tipo risalgono agli anni '70, quando venivano utilizzati per il rendering di testo e grafica, per combinare più modelli grafici e per disegnare rettangoli, triangoli, cerchi e archi, Le moderne GPU supportano la grafica 3D, l'ombreggiatura e il video digitale. I fornitori dominanti di GPU sono oggi Intel, NVIDIA e AMD. Alcune applicazioni embedded, in particolare i giochi, si adattano bene alle GPU. Inoltre, le GPU si sono evolute verso modelli di programmazione più generali, e quindi hanno iniziato a comparire in altre applicazioni ad alta intensità di calcolo, come la strumentazione. Le GPU sono in genere piuttosto affamate di energia e quindi oggi non sono adatte alle applicazioni embedded con vincoli energetici.

**The x86 Architecture**

ISA dominante per i computer desktop e portatili. Nasce con l'Intel 8086, un microprocessore a 16 bit progettato da Intel nel 1978. Una variante dell'8086, denominata 8088, è stata utilizzata nel PC IBM originalee da allora questa famiglia di processori ha dominato il mercato dei PC. I processori successivi di questa famiglia sono stati chiamati con nomi che terminano con "86" e in genere hanno mantenuto la retrocompatibilità. L'Intel 80386 è stata la prima versione a 32 bit di questo set di istruzioni, introdotta nel 1985. Oggi, il termine "x86" si riferisce solitamente alla versione a 32 bit, mentre le versioni a 64 bit sono denominate "x86-64". L'Intel Atom, introdotto nel 2008, è un processore x86 con un consumo energetico notevolmente ridotto. Sebbene sia destinato principalmente ai netbook e ad altri piccoli computer mobili, è un'opzione interessante anche per alcuni computer embedded. L'architettura x86 è stata anche introdotta in processori di AMD, Cyrix e di molti altri produttori.

**Parallelism**

La maggior parte dei processori oggi offre varie forme di parallelismo. Questi meccanismi influenzano fortemente la tempistica dell'esecuzione di un programma; quindi, i progettisti di sistemi embedded devono comprenderli.

**Parallelism vs. Concurrency**

La concorrenza è centrale nei sistemi embedded. Un programma è detto concorrente se diverse parti del programma vengono eseguite simultaneamente a livello concettuale. Un programma si dice parallelo se diverse parti del programma vengono eseguite fisicamente in modo simultaneo su hardware distinti (ad esempio su macchine multicore, server in una server farm o microprocessori distinti).

I programmi non concorrenti specificano una sequenza di istruzioni da eseguire. Un linguaggio di programmazione che esprime una computazione come una sequenza di operazioni è chiamato linguaggio imperativo. Il C è un linguaggio imperativo. Quando si usa il C per scrivere programmi concorrenti, bisogna uscire dal linguaggio stesso, tipicamente utilizzando una libreria di thread. Una libreria di thread utilizza strutture non fornite dal C, ma dal sistema operativo e/o dall'hardware. Java è un linguaggio per lo più imperativo esteso con costrutti che supportano direttamente i thread. Pertanto, è possibile scrivere programmi concorrenti in Java senza programmi concomitanti E senza uscire dal linguaggio. Ogni esecuzione (corretta) di un programma in un linguaggio imperativo deve comportarsi come se le istruzioni fossero eseguite esattamente nella sequenza specificata. Tuttavia, spesso è possibile eseguire le istruzioni in parallelo o in un ordine diverso da quello specificato dal programma e ottenere comunque un comportamento corrispondente all'esecuzione sequenziale.

**Dataflow analysis**

Un compilatore può analizzare le dipendenze tra le operazioni in un programma e produrre codice parallelo, se la macchina di destinazione lo supporta. Molti microprocessori oggi supportano l'esecuzione in parallelo, utilizzando flussi di istruzioni o architetture VLIW (very large instruction word). I processori con flussi di istruzioni multi-emissione possono eseguire istruzioni indipendenti simultaneamente. L'hardware analizza le istruzioni al volo per individuare eventuali dipendenze e, se non ci sono dipendenze, esegue più di un'istruzione alla volta.

Le macchine VLIW dispongono di istruzioni a livello ASSEMBLY che specificano più operazioni da eseguire insieme. In questo caso, è necessario che il compilatore produca le istruzioni parallele appropriate. L'analisi delle dipendenze viene effettuata a livello di linguaggio assembly o a livello di singole operazioni, non a livello di righe di C. Una riga di C può specificare più operazioni, o anche operazioni complesse come le chiamate a procedure. In entrambi i casi (multi-issue e VLIW), un programma imperativo viene analizzato per la concorrenza al fine di consentire l'esecuzione parallela. L'obiettivo generale è accelerare l'esecuzione del programma e le prestazioni, dove si presume che finire un compito prima sia sempre meglio che finirlo dopo.

Nel contesto dei sistemi embedded, la concorrenza ha un ruolo molto più importante del semplice miglioramento delle prestazioni. I programmi embedded interagiscono con i processi fisici e nel mondo fisico molte attività si svolgono contemporaneamente. Un programma incorporato deve spesso monitorare e reagire a più fonti simultanee di stimoli e controllare simultaneamente più uscite. I programmi embedded sono quasi sempre programmi concorrenti e la concorrenza è una parte intrinseca della logica dei programmi. Non è solo un modo per migliorare le prestazioni. In effetti, terminare un'attività in anticipo non è necessariamente meglio che terminarla in ritardo. Le azioni eseguite nel mondo fisico spesso devono essere eseguite al momento giusto (né in anticipo né in ritardo). Immaginiamo ad esempio un controllore per un motore a benzina. Accendere le candele prima non è certamente meglio che accenderle più tardi. Devono essere accese al momento giusto.

Proprio come i programmi imperativi possono essere eseguiti in modo sequenziale o in parallelo, i programmi concorrenti possono essere eseguiti in modo sequenziale o parallelo. L'esecuzione sequenziale di un programma concorrente è oggi tipicamente effettuata da un sistema operativo multitasking, che intreccia l'esecuzione di più compiti in un unico flusso sequenziale di istruzioni. Naturalmente, l'hardware può parallelizzare l'esecuzione se il processore è dotato di un'architettura architettura multi-emissione o VLIW. Quindi, un programma concorrente può essere convertito in un flusso sequenziale da un sistema operativo e di nuovo in un programma concorrente da parte dell'hardware. Queste traduzioni multiple complicano notevolmente il problema di garantire che le cose avvengano al momento giusto.

Il parallelismo nell'hardware esiste per migliorare le prestazioni di applicazioni ad alta intensità di calcolo. Dal punto di vista del programmatore, la concomitanza nasce come conseguenza dell'hardware progettato per migliorare le prestazioni, non come conseguenza del problema applicativo da risolvere. L'applicazione non richiede (necessariamente) che più attività procedano simultaneamente, ma richiede solo che le cose siano fatte molto molto velocemente.

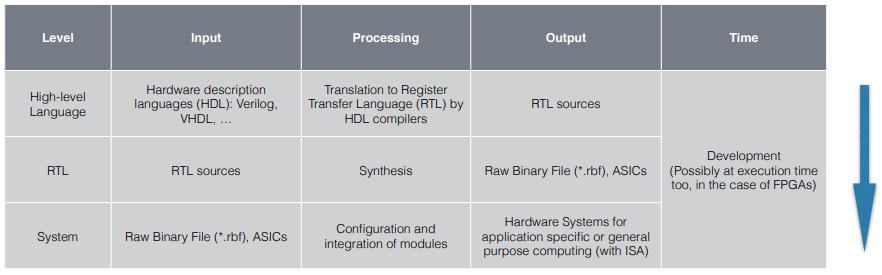
L'esecuzione del software applicativo richiede diverse fasi di elaborazione in **fase di sviluppo** e **di esecuzione**.

Immagine che contiene tavolo

Descrizione generata automaticamente

La logica digitale spesso include più unità funzionali simili (ad esempio ALU, MAC, controllori di memoria) per l'esecuzione in parallelo. Oltre al parallelismo hardware, vengono utilizzate diverse tecniche per migliorare le prestazioni nelle CPU general-purpose: pipeline lunghe, esecuzioni fuori ordine e speculative, predizione dei rami. Queste tecniche sono evitate nei processori embedded (perché?).

Il livello hardware può essere realizzato anche attraverso le specifiche software. La configurazione dell'hardware sta diventando una pratica comune nello sviluppo embedded. Esistono persino CPU di uso generale che includono hardware configurabile (FPGA). Gli HDL includono le nozioni di tempo, eventi e concorrenza nel loro formalismo.



**Pipelining. Una semplice pipeline a cinque stadi per una macchina a 32 bit.**

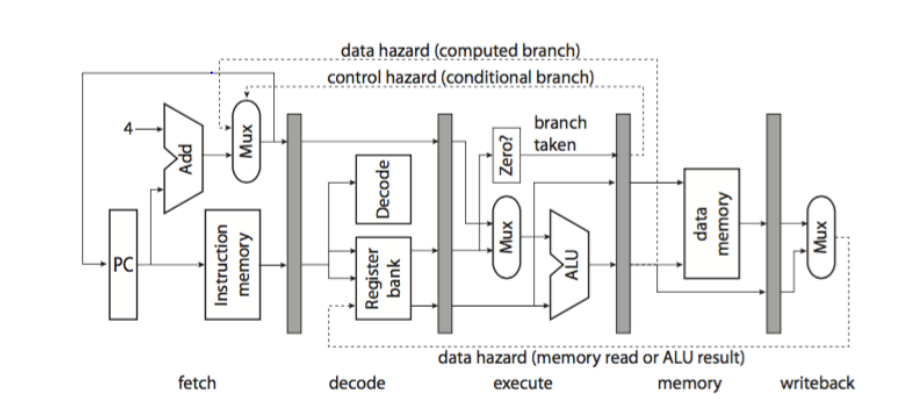


Immagine che contiene testo

Descrizione generata automaticamente

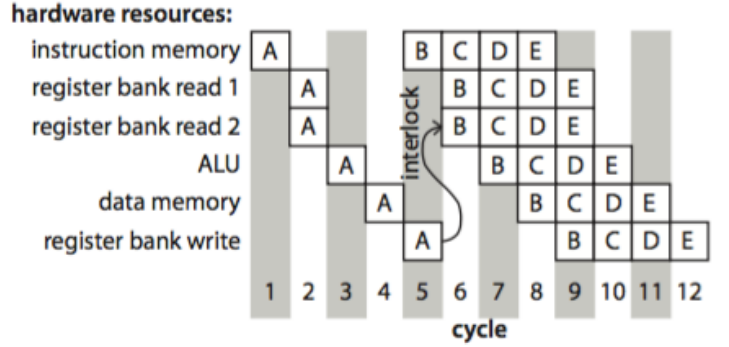
**Addressing data hazards: explicit pipeline**

Per l'esempio in cui B legge un registro scritto da A, il compilatore potrebbe, il compilatore può inserire tre istruzioni no-op (che non fanno nulla) tra A e B per garantire che la scrittura avvenga prima della lettura Queste istruzioni no-op formano una bolla della pipeline che si propaga.

**Addressing data hazards: Interlocks**

L'hardware di decodifica delle istruzioni, quando incontra l'istruzione B che legge un registro scritto da A, rileverà il pericolo e ritarderà l'esecuzione di B finché A non avrà completato la fase di writeback. Per questa pipeline, B dovrebbe essere ritardato di tre cicli di clock per consentire ad A di completare la fase di writeback. Questo può essere ridotto a due cicli se si prevede una logica di inoltro un po' più complessa, in grado di rilevare che A sta scrivendo la stessa posizione che B sta leggendo, e fornisce direttamente i dati invece di richiedere che la scrittura avvenga prima della lettura.

Gli interlocks forniscono quindi in hardware un sistema che inserisce automaticamente le bolle nella pipeline.



**Addressing data hazards: Out-of-order execution**

Esecuzione fuori ordine: viene fornito un hardware che rileva un pericolo, ma invece di ritardare semplicemente l'esecuzione di B, procede al fetch di C, e se C non legge i registri scritti da A o B, e non scrive registri letti da B, procede all'esecuzione di Cprima di B. Questo riduce ulteriormente il numero di bolle della pipeline.

**Control hazards**

Un'istruzione di diramazione condizionale cambia il valore del PC se un registro specificato ha valore zero. Il nuovo valore del PC è fornito (facoltativamente) dal risultato di un'operazione ALU. Se A è un'istruzione di diramazione condizionale, allora A deve aver raggiunto lo stadio di memoria prima che il PC possa essere aggiornato. Le istruzioni che seguono A in memoria saranno state recuperate e si troveranno già negli stadi di decodifica e di esecuzione nel momento in cui si determina che tali istruzioni non devono essere eseguite.

**Addressing control hazards:**

* **Delayed branch:** documenta semplicemente il fatto che il ramo sarà preso un certo numero di cicli dopo averla incontrata, e lascia al programmatore (o al compilatore) di assicurarsi che le istruzioni che seguono il ramo di diramazione condizionale siano innocue (come i no-op) o che svolgano un lavoro utile che non dipende dal fatto che il ramo venga eseguito o meno.
* **Interlocks:** hardware per inserire bolle di pipeline secondo le necessità, proprio come nel caso di pericoli dei dati.
* **Speculative execution:** L'hardware valuta se è probabile che il ramo debba essere eseguito e inizia a eseguire le istruzioni che si aspetta di eseguire. Se non viene soddisfatta la condizione, annulla tutti gli effetti collaterali (come la scrittura di registri) che le istruzioni eseguite in modo speculativo hanno causato.

**Data and Control hazards:**

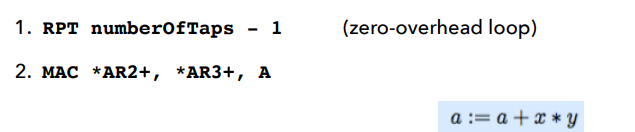
Tranne che per le pipeline esplicite e le ramificazioni ritardate, tutte le tecniche introducono una variabilità nei tempi di esecuzione di una sequenza di istruzioni. L'analisi dei tempi di un programma può diventare estremamente difficile. Le pipeline esplicite sono relativamente comuni nei processori DSP, che sono spesso applicati in contesti in cui la tempistica è essenziale. L'esecuzione fuori ordine e speculativa è comune nei processori general-purpose, dove la tempistica è importante dove la tempistica è importante solo in senso aggregato. Il progettista di un sistema embedded deve comprendere i requisiti dell'applicazione ed evitare i processori in cui il livello di precisione temporale è irraggiungibile.

**Il parallelismo a livello di istruzioni** può essere implementato come:

* ***CISC instructions***
* ***subword parallelism***
* ***superscalar***
* ***VLIW***

Un processore con istruzioni complesse (e tipicamente piuttosto specializzate) è chiamato macchina **CISC** (complex instruction set computer). La filosofia alla base di questi processori è nettamente diversa da quella delle macchine RISC (reduced instruction set computer). I DSP sono tipicamente macchine CISC, e includono istruzioni che supportano specificamente il filtraggio FIR e spesso altri algoritmi come le FFT (fast Fourier) e la decodifica Viterbi. In effetti, per qualificarsi come DSP, un processore deve essere in grado di eseguire il filtraggio FIR in un ciclo di istruzioni per tap.

La famiglia di processori DSP TMS320c54x di Texas Instruments è destinata all'utilizzo in applicazioni embedded a basso consumo energetico che richiedono elevate prestazioni di elaborazione del segnale, come i sistemi di comunicazione wireless e gli assistenti digitali personali (PDA). L'anello interno di un calcolo FIR è :



I registri AR2 e AR3 possono essere impostati per implementare buffer circolari. Il processore c54x include una sezione di memoria on-chip che supporta due accessi in un singolo ciclo e finché gli indirizzi si riferiscono a questa sezione di memoria, l'istruzione MAC verrà eseguita in un singolo ciclo. Ad ogni ciclo, il processore esegue due fetches di memoria, una moltiplicazione, un'addizione e due incrementi di indirizzo (eventualmente modulo).

Se i coefficienti del filtro FIR sono simmetrici:

▸ N è pari e ai = aN-i-1 .

▸ Il numero di moltiplicazioni può essere ridotto riscrivendo la formula come:

Immagine che contiene testo

Descrizione generata automaticamente

Il set di istruzioni TMS320c54x di Texas Instruments include un'istruzione FIRS che funziona in modo simile all'istruzione al MAC, ma con questo calcolo. Questo sfrutta il fatto che il c54x dispone di due ALU e quindi può eseguire il doppio delle addizioni rispetto alle moltiplicazioni. Il tempo di esecuzione di un filtro FIR si riduce ora a 1/2 ciclo per tap.

**Subword parallelism**

Molte applicazioni embedded operano su tipi di dati che sono considerevolmente più piccoli della dimensione di parola del processore (ad esempio i dati RBG). Un'ALU ampia viene suddivisa in fette più strette che consentono di eseguire simultaneamente operazioni aritmetiche o logiche su parole più piccole. Intel ha introdotto il parallelismo delle sotto parole nel processore generico Pentium, ampiamente utilizzato, e ha chiamato questa tecnologia MMX. Tecniche simili sono state introdotte da Sun Microsystems per i processori Sparc e da Hewlett Packard per il processore PA RISC Molte architetture di processori progettati per applicazioni embedded, tra cui molti processori DSP, supportano anche il parallelismo subword. Un processore vettoriale è un processore in cui l'insieme di istruzioni comprende operazioni su più elementi di dati simultaneamente. Il parallelismo di sottoparole è una forma particolare di elaborazione vettoriale.

**Superscalar processors**

Utilizzano set di istruzioni sequenziali abbastanza convenzionali, ma l'hardware può dividere istruzioni multiple a unità hardware distinte, quando rileva che tale simultanea non modificherà il comportamento del programma. L'esecuzione del programma è identica a quella che avrebbe avuto se fosse stata eseguita in sequenza. Questi processori supportano anche l'esecuzione fuori ordine, in cui le istruzioni più avanti nel flusso vengono eseguite prima di quelle precedenti. 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), possono anche non essere ripetibili. I tempi di esecuzione possono essere molto sensibili all'esatta tempistica degli interrupt, in quanto piccole variazioni di tale tempistica possono avere grandi effetti sui tempi di esecuzione dei programmi.

**Very Long Instruction Word (VLIW) architectures**

I processori embedded utilizzano spesso architetture VLIW anziché superscalari per ottenere tempi più ripetibili e prevedibili.

* I processori VLIW includono unità funzionali multiple, come i processori superscalari, ma invece di determinare dinamicamente quali istruzioni possono essere eseguite contemporaneamente, ogni istruzione specifica ciò che ciascuna unità funzionale deve fare in un determinato ciclo.
* Un set di istruzioni VLIW combina più operazioni indipendenti in una singola istruzione.
* Come nelle architetture superscalari, queste operazioni multiple vengono eseguite simultaneamente su hardware distinti
* A differenza delle superscalari, tuttavia, l'ordine e la simultaneità dell'esecuzione sono fissati nel programma, non decisi al momento. Spetta al programmatore (che lavora a livello di linguaggio assembly) o al compilatore di assicurarsi che le operazioni simultanee siano effettivamente indipendenti.
* In cambio di questa ulteriore complessità nella programmazione, i tempi di esecuzione diventano ripetibili e (spesso) prevedibili.

Il TMS320c55x di Texas Instruments, la generazione successiva al c54x, include due unità di moltiplicazione-accumulazione e può supportare istruzioni di questo tipo:

1. MAC \*AR2+, \*CDP+, AC0
2. ::MAC \*AR3+, \*CDP+, AC1

AC0 e AC1 sono due registri di accumulo, mentre CDP è un registro specializzato puntare ai coefficienti del filtro. La notazione :: indica che queste due istruzioni devono essere emesse ed eseguite nello stesso ciclo. Spetta al programmatore o al compilatore stabilire se queste istruzioni possono essere eseguite simultaneamente. Supponendo che gli indirizzi di memoria siano tali da consentire l'esecuzione simultanea delle due istruzioni MAC, esse vengono eseguite in un solo ciclo, dividendo di fatto a metà il tempo necessario per l'esecuzione di un filtro FIR.

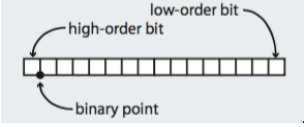
**Multicore Architectures**

Una macchina multicore è una combinazione di più processori su un singolo chip. Sebbene le macchine multicore esistano fin dai primi anni '90, solo di recente si sono penetrate nell'informatica generale. Questa penetrazione spiega gran parte dell'interesse odierno per queste macchine. Le macchine multicore eterogenee combinano una varietà di tipi di processori su un singolo chip, rispetto alle macchine multicore a più istanze dello stesso tipo di processore. Per le applicazioni embedded, le architetture multicore presentano un vantaggio potenziale significativo rispetto alle architetture single-core perché le attività in tempo reale e quelle critiche per la sicurezza possono avere un processore dedicato. Questo è il motivo delle architetture eterogenee utilizzate per i telefoni cellulari, in quanto le funzioni di radio e di elaborazione vocale sono funzioni in tempo reale con un notevole carico computazionale. In queste architetture, le applicazioni utente non possono interferire con le funzioni in tempo reale.

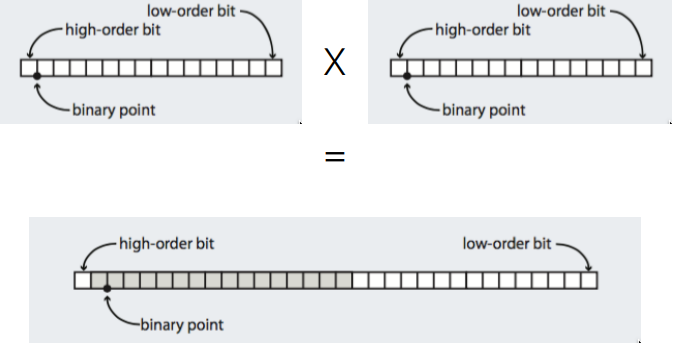
Questa mancanza di interferenza è più problematica nelle architetture multicore generiche. È comune, ad esempio, l'uso di cache multilivello, in cui il secondo livello o quello superiore è condiviso tra i core. Sfortunatamente, tale condivisione rende molto difficile isolare il comportamento in tempo reale dei programmi su core separati, poiché ogni programma può innescare miss della cache in un altro core. Tali cache multilivello non sono adatte alle applicazioni in tempo reale. Un tipo di architettura multicore molto diverso, talvolta utilizzato nelle applicazioni embedded, utilizza uno o più soft core insieme a hardware personalizzato su un fieldprogrammable gate array (FPGA). Gli FPGA sono chip le cui funzioni hardware sono programmabili mediante strumenti di progettazione hardware. I soft core sono processori implementati su FPGA. Il vantaggio dei soft core è che possono essere accoppiati all'hardware personalizzato più facilmente dei processori standard.

**Fixed-Point Numbers**

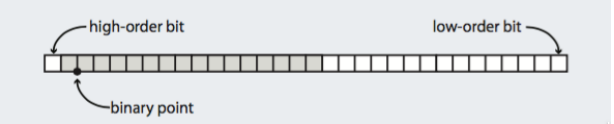
Molti processori embedded forniscono hardware solo per l'aritmetica dei numeri interi. L'aritmetica dei numeri interi può essere utilizzata per i numeri non interi, con qualche accortezza. Dato, ad esempio, un numero intero a 16 bit, un programmatore può immaginare un punto binario, che è come un punto decimale, con la differenza che separa i bit anziché le cifre del numero. Ad esempio, un numero intero a 16 bit può essere utilizzato per rappresentare numeri compresi nell'intervallo da -1,0 a 1,0 (approssimativamente) posizionando un punto binario (concettuale) appena sotto il bit di ordine superiore del numero.



Senza il punto binario, un numero rappresentato dai 16 bit è un numero intero x ∈ {-215, ..., 215 - 1} (complemento a due). Con il punto binario, interpretiamo i 16 bit per rappresentare il numero y = x/215. Quindi, y varia da da -1 a 1 - 2-15. Questo è noto come numero a virgola fissa. Il formato di questo numero a virgola fissa può essere scritto 1.15, indicando che c'è un bit a sinistra del punto binario e 15 a destra. Quando due numeri di questo tipo vengono moltiplicati a precisione, il risultato è un numero a 32 bit.



La posizione del punto binario deriva dalla legge di conservazione dei bit. Quando si moltiplicano due numeri con formato n.m e p.q, il risultato ha formato (n+p).(m+q) . I processori spesso supportano queste moltiplicazioni a precisione completa, in cui il risultato va in un registro di accumulo con un numero di bit almeno doppio rispetto ai registri di dati ordinari. Per riscrivere il risultato in un registro dati, tuttavia, è necessario estrarre 16 bit dal risultato di 32 bit. Se estraiamo l'ombreggiato, conserviamo la posizione del punto binario e il risultato rappresenta ancora un numero approssimativamente nell'intervallo da -1 a 1.



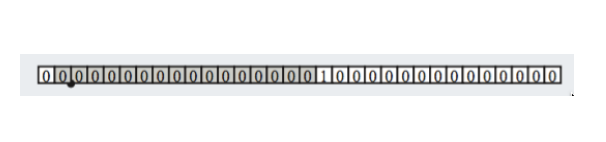
Tuttavia, l'estrazione di 16 bit da un risultato di 32 bit comporta una perdita di informazioni. In primo luogo, c'è la possibilità di un overflow, perché scartiamo il bit di ordine superiore. Supponiamo che i due numeri moltiplicati siano entrambi -1:

Immagine che contiene testo

Descrizione generata automaticamente

Questa configurazione in complemento a due rappresenta 1, il risultato corretto. Tuttavia, estraendo i 16 bit ombreggiati, il risultato è ora -1! Infatti, 1 non è rappresentabile nel formato a virgola fissa 1.15, quindi si è verificato un overflow. I programmatori devono prevenire questo problema, ad esempio assicurandosi che tutti i numeri siano strettamente inferiori a 1, vietando il -1.

Un secondo problema è che quando si estraggono i 16 bit sfumati da un risultato a 32 bit, si scartano 15 bit di ordine inferiore. In questo caso c'è una perdita di informazioni. Se scartiamo semplicemente i 15 bit di ordine inferiore, la strategia è nota come troncamento. Se invece aggiungiamo al risultato a 32 bit il modello di bit dell'immagine, il risultato è noto come arrotondamento. L'arrotondamento sceglie il risultato che più si avvicina al risultato a precisione piena, mentre il troncamento sceglie il risultato più vicino ma più piccolo in grandezza. I processori DSP eseguono tipicamente l'estrazione di cui sopra con l'arrotondamento o il troncamento in hardware quando i dati vengono spostati da un accumulatore a un registro di uso generale o alla memoria.



**Memory Architectures**

Nei sistemi embedded, i problemi di memoria sono molto importanti. La scelta delle tecnologie di memoria hanno conseguenze importanti per il progettista del sistema. Ad esempio, un programmatore potrebbe doversi preoccupare della persistenza dei dati quando si spegne l'alimentazione o si entra in una modalità di standby a risparmio energetico. Una memoria il cui contenuto viene perso quando viene tolta l'alimentazione si chiama memoria volatile. In questa sezione vengono illustrate alcune delle tecnologie disponibili e i loro relativi compromessi.

**RAM**

Oltre al file di registro, un microcomputer include in genere una certa quantità di RAM (memoria ad accesso casuale) che è una memoria in cui singoli elementi (byte o parole) possono essere scritti e letti uno alla volta in modo relativamente veloce. La SRAM (RAM statica) è più veloce della DRAM (RAM dinamica), ma è anche più grande (ogni bit occupa un'area maggiore del silicio). La DRAM conserva i dati solo per un breve periodo di tempo, quindi ogni posizione di memoria deve essere periodicamente aggiornata. La SRAM conserva i dati per tutto il tempo in cui viene mantenuta l'alimentazione. Entrambi i tipi di memoria perdono il loro contenuto in caso di interruzione dell'alimentazione, quindi entrambe sono memorie volatili, anche se probabilmente la DRAM è più volatile della SRAM perché perde il suo contenuto anche se viene mantenuta l'alimentazione. La maggior parte dei sistemi informatici embedded include una memoria SRAM. Molti includono anche una DRAM perché può essere poco pratico fornire una quantità di memoria sufficiente con la sola tecnologia SRAM. A un programmatore che si preoccupa del tempo di esecuzione di un programma deve sapere se gli indirizzi di memoria a cui si accede sono mappati su SRAM o DRAM. Inoltre, il ciclo di refresh della DRAM può introdurre una variabilità nei tempi di accesso, perché la DRAM può essere impegnata in un refresh nel momento in cui viene richiesto l'accesso. Inoltre, lo storico degli accessi può influire sui tempi di accesso. Il tempo necessario per accedere a un indirizzo di memoria può dipendere dall'ultimo indirizzo di memoria a cui si è acceduto. Il produttore di un chip di memoria DRAM specificherà che ogni locazione di memoria deve essere aggiornata, ad esempio, ogni 64 ms, e che un certo numero di locazioni (una "fila") viene aggiornato insieme. Il semplice atto di leggere la memoria aggiornerà le posizioni lette (e le posizioni sulla stessa riga) ma poiché le applicazioni non possono accedere a tutte le righe entro l'intervallo di tempo specificato, la DRAM deve essere utilizzata con un controllore che garantisca che tutte le locazioni vengano aggiornate con una frequenza sufficiente a conservare i dati. Il controller di memoria stallerà gli accessi se la memoria è occupata da un refresh quando viene avviato l'accesso. Questo introduce una variabilità nella tempistica del programma.

**Non-Volatile Memory**

I sistemi embedded hanno sempre bisogno di memorizzare i dati anche quando l'alimentazione è spenta. Esistono diverse opzioni per questo scopo. Una, ovviamente, è quella di fornire una batteria di backup in modo da non perdere mai l'alimentazione. Le batterie, tuttavia, si consumano e sono disponibili opzioni migliori, note come memorie non volatili. Una prima forma di memoria non volatile era la memoria a nucleo magnetico o semplicemente nucleo, in cui un anello ferromagnetico veniva magnetizzato per memorizzare i dati. Il termine "core" persiste nell'informatica per indicare le memorie dei computer, anche se questo potrebbe cambiare con l'avvento delle macchine multicore. La memoria non volatile più elementare oggi è la ROM (memoria di sola lettura) o ROM a maschera, il cui contenuto viene fissato nella fabbrica del chip. Questo può essere utile per i prodotti di massa che necessitano solo di un programma e di dati costanti, che non cambiano mai. Tali programmi sono noti come firmware, il che suggerisce che non sono "morbidi" come il software. Esistono diverse varianti di ROM che possono essere programmate sul campo, e la tecnologia è diventata abbastanza buona da essere quasi sempre utilizzata al posto della maschera ROM. La EEPROM, ROM programmabile elettricamente, è disponibile in diverse forme, ma è possibile scrivere su tutte. Il tempo di scrittura è in genere molto più lungo di quello di lettura e il numero di scritture è limitato durante la vita del dispositivo. Una forma particolarmente utile di EEPROM è la memoria flash. La memoria flash viene comunemente utilizzata per memorizzare il firmware e i dati utente che devono persistere quando si spegne il dispositivo. La memoria flash, inventata dal Dr. Fujio Masuoka presso Toshiba intorno al 1980, è una forma particolarmente conveniente di memoria non volatile, ma presenta alcune sfide interessanti per i progettisti di sistemi embedded. In genere, le memorie flash hanno tempi di lettura ragionevolmente veloci, ma non quanto la SRAM e la DRAM, per cui i dati a cui si accede di frequente devono essere spostati dalla flash alla RAM prima di essere utilizzati da un programma. I tempi di scrittura sono molto più lunghi di quelli di lettura e il numero totale di scritture è limitato. Esistono due tipi di memorie flash, note come NOR e NAND flash. La memoria flash NOR ha tempi di cancellazione e scrittura più lunghi, ma è accessibile come una RAM. La memoria NAND è meno meno costosa e ha tempi di cancellazione e scrittura più rapidi, ma i dati devono essere letti un blocco alla volta, dove un blocco è costituito da centinaia o migliaia di bit. Questo significa che dal punto di vista del sistema si comporta più come un dispositivo di archiviazione secondario come un disco rigido o un supporto ottico come un CD o un DVD. Entrambi i tipi di flash possono essere cancellati e riscritti solo per un numero limitato di volte, di solito inferiore a 1.000.000 per la NOR flash e a 10.000.000 per la NAND flash. I tempi di accesso più lunghi, il numero limitato di scritture e gli accessi a blocchi (per le memorie NAND) complicano il problema per i progettisti di sistemi embedded. Queste proprietà devono essere prese in considerazione non solo durante la progettazione dell'hardware, ma anche del software. Anche le memorie a disco sono non volatili. Possono memorizzare grandi quantità di dati, ma i tempi di accesso possono diventare molto elevati. In particolare, la meccanica di un disco in rotazione e di una testina di lettura/scrittura, il controller deve attendere che la testina sia posizionata sulla posizione richiesta prima di poter leggere i dati in quella posizione. Il tempo necessario è molto variabile. I dischi sono anche più vulnerabili alle vibrazioni rispetto alle memorie a stato solido discusse in precedenza, e quindi sono più difficili da usare in molte applicazioni embedded.

**Memory Hierarchy**

Molte applicazioni richiedono quantità sostanziali di memoria, più di quella disponibile su chip in un microcomputer. Molti processori utilizzano una gerarchia di memoria, che combina diverse tecnologie di memoria per aumentare la capacità di memoria complessiva, ottimizzando al contempo i costi, la latenza e il consumo energetico. In genere, viene utilizzata una quantità relativamente piccola di SRAM on-chip con una quantità maggiore di DRAM off-chip. Questi possono essere ulteriormente combinati con un terzo livello, come le unità disco, che hanno una capacità molto elevata, ma non hanno accesso casuale e quindi possono essere piuttosto lenti in lettura e scrittura. Il programmatore dell'applicazione potrebbe non essere consapevole del fatto che la memoria è frammentata tra queste tecnologie. Uno schema comunemente usato, chiamato memoria virtuale, fa sì che le diverse tecnologie appaiano al programmatore come uno spazio di indirizzi contiguo. Il sistema operativo e/o l'hardware forniscono la traduzione degli indirizzi, che converte gli indirizzi logici nello spazio degli indirizzi in posizioni fisiche.in una delle tecnologie di memoria disponibili. Questa traduzione è spesso assistita da un hardware specializzato, chiamato translation lookaside buffer (TLB) che può accelerare alcune traduzioni di indirizzi. Per il progettista di un sistema embedded, queste tecniche possono creare seri problemi perché rendono molto difficile prevedere o capire quanto tempo impiegheranno gli accessi alla memoria. Di conseguenza, i progettisti di sistemi embedded devono conoscere il sistema di memoria in modo più approfondito rispetto ai programmatori di sistemi generici.

**Memory Maps**

La **mappa di memoria** di un processore definisce il modo in cui gli indirizzi vengono mappati sull'hardware. La dimensione totale dello spazio degli indirizzi è limitata dalla larghezza degli indirizzi del processore. Un processore a 32 bit, ad esempio, può indirizzare 232 locazioni, o 4 gigabyte (GB), supponendo che ogni indirizzo si riferisca a un byte. La larghezza dell'indirizzo corrisponde in genere alla larghezza della parola, tranne che per i processori a 8 bit, dove l'ampiezza dell'indirizzo è tipicamente maggiore (spesso 16 bit). Un'architettura ARM CortexTM - M3, ad esempio, presenta la mappa di memoria mostrata nella Figura.

Immagine che contiene tavolo

Descrizione generata automaticamente

Altre architetture avranno altri layout, ma lo schema è simile. Si noti che questa architettura separa gli indirizzi utilizzati per la memoria di programma (contrassegnati con A nella figura) da quelli utilizzati per la memoria dati (B e D). Questo schema (tipico) consente di accedere a queste memorie tramite bus separati, consentendo l'acquisizione simultanea di istruzioni e dati. In questo modo si raddoppia la larghezza di banda della memoria. Questa separazione della memoria di programma da quella di dati è nota come architettura Harvard. Si contrappone con la classica architettura von Neumann, che memorizza programmi e dati nella stessa memoria. Ogni particolare realizzazione in silicio di questa architettura è vincolata da questa mappa di memoria. Ad esempio, il controllore Luminary Micro1 LM3S8962, che include un core ARM CortexTM - M3, dispone di 256 KB di memoria flash on-chip, che non si avvicina neanche lontanamente al totale di 0,5 GB che l'architettura consente. Questa memoria è mappata agli indirizzi da 0x000000 fino a 0x0003FFFF. Gli indirizzi rimanenti consentiti dall'architettura per la memoria di programma, che vanno da 0x00040000 a 0x1FFFFFFF, sono "indirizzi riservati", significa che non devono essere utilizzati da un compilatore che si rivolge a questo particolare dispositivo.

L'LM3S8962 dispone di 64 KB di SRAM, mappati agli indirizzi da 0x20000000 a 0x2000FFFF, una piccola porzione dell'area B nella figura. Include anche una serie di periferiche on-chip, ovvero dispositivi a cui il processore accede utilizzando alcuni indirizzi di memoria nell'intervallo da 0x40000000 a 0x5FFFFFFF (area C nella figura). Queste periferiche includono timer, ADC, GPIO, UART e altri dispositivi di I/O. Ognuno di questi dispositivi occupa alcuni degli indirizzi di memoria, fornendo registri. Il processore può scrivere su alcuni di questi registri per configurare e/o controllare una periferica o per fornire dati da produrre su un'uscita. Alcuni dei registri possono essere letti per recuperare i dati di ingresso ottenuti dalla periferica. Alcuni indirizzi della regione del bus periferica privata sono utilizzati per accedere al controllore di interrupt. L'LM3S8962 è montato su una scheda a circuiti stampati che fornirà dispositivi aggiuntivi come la memoria dati DRAM e altri dispositivi esterni. Come mostrato nella Figura, questi saranno mappati su indirizzi di memoria compresi nell'intervallo da 0xA0000000 a 0xDFFFFFFF (area E). Ad esempio, la scheda di valutazione Stellaris R LM3S8962 di Luminary Micro non include memoria esterna aggiuntiva ma aggiunge alcuni dispositivi esterni come un display LCD, uno slot MicroSD per memoria flash aggiuntiva e un'interfaccia USB. Questo lascia molti indirizzi di memoria inutilizzati. ARM ha introdotto un modo intelligente per sfruttar questi indirizzi inutilizzati, chiamato bit banding, in cui alcuni degli indirizzi inutilizzati possono essere utilizzati per accedere a singoli bit piuttosto che a interi byte o parole della memoria e delle periferiche. In questo modo alcune operazioni sono più efficienti, poiché le istruzioni aggiuntive per mascherare i bit desiderati non sono più necessarie.

**Register Files**

La memoria più strettamente integrata in un processore è il **register file**. Ogni **registro nel register file** memorizza una parola. La dimensione di una parola è una proprietà fondamentale dell'architettura di un processore. La parola è un byte su un'architettura a 8 bit, quattro byte su un'architettura a 32 bit e otto byte su un'architettura a 64 bit. Il file di registro può essere implementato direttamente con i flip flop nel circuito del processore, oppure i registri possono essere raccolti in un unico banco di memoria, tipicamente utilizzando la stessa tecnologia SRAM discussa in precedenza. Il numero di registri in un processore è solitamente ridotto. Il motivo non è tanto il costo dell'hardware del file di registro, ma piuttosto il costo dei bit in una parola di istruzione. Un'architettura di set di istruzioni (ISA) fornisce in genere istruzioni che possono accedere a uno, due o tre registri. Per memorizzare in modo efficiente i programmi in memoria, queste istruzioni non possono richiedere troppi bit per la loro codifica, e quindi non possono dedicare troppi bit all'identificazione dei registri. Se il register file ha 16 registri, ogni riferimento a un registro richiede 4 bit. Se un'istruzione può fare riferimento a 3 registri, richiede un totale di 12 bit. Se la parola di un'istruzione è di 16 bit, rimangono solo 4 bit per altre informazioni, come ad esempio l'identità dell'istruzione stessa, che deve essere codificata nell'istruzione. Questo identifica, ad esempio, se l'istruzione specifica che due registri devono essere sommati o sottratti, con il risultato memorizzato nel terzo registro.

**Scratchpads and Caches**

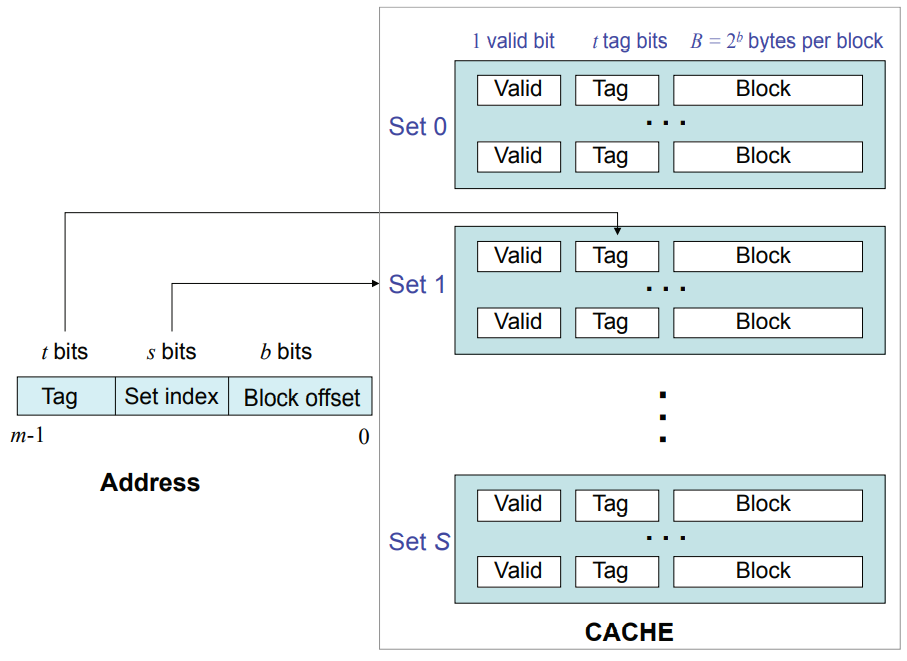
Molte applicazioni embedded combinano tecnologie di memoria. Alcune memorie vengono accedute prima di altre; diciamo che le prime sono più "vicine" al processore rispetto alle seconde. Per esempio, una memoria vicina (SRAM) è tipicamente utilizzata per memorizzare temporaneamente i dati di lavoro mentre il programma opera su di essi. Se la memoria vicina ha un insieme distinto di indirizzi e il programma è responsabile dello spostamento dei dati in essa o nella memoria lontana, allora questa operazione si chiama **scratchpad**. Se la memoria vicina duplica i dati nella memoria distante e l'hardware gestisce automaticamente la copia da e verso, allora si chiama **cache**.

Per le applicazioni embedded con stretti vincoli di tempo reale, le memorie **cache** presentano alcuni ostacoli formidabili perché il loro comportamento temporale può variare in modo sostanziale e difficilmente prevedibile. D'altra parte, la gestione manuale dei dati in una memoria scratchpad può essere piuttosto noiosa per un programmatore, e i metodi automatici guidati dal compilatore per farlo sono ancora agli albori.

Come spiegato, un'architettura in genere supporta uno spazio di indirizzi molto più ampio di quello che può essere effettivamente memorizzato nella memoria fisica del processore, con un sistema di memoria virtuale per presentare al programmatore la vista di uno spazio di indirizzi contiguo. Se il processore è dotato di un'unità di gestione della memoria (MMU), i programmi fanno riferimento a indirizzi logici e la MMU li traduce in indirizzi fisici. Ad esempio, utilizzando la mappa di memoria della Figura 9.1, un processo potrebbe essere autorizzato a utilizzare gli indirizzi logici da 0x60000000 a 0x9FFFFF (area D nella figura), per un totale di 1 GB di memoria dati indirizzabile. La MMU può implementare una cache che utilizza la quantità di memoria fisica presente nell'area B. Quando il programma fornisce un indirizzo di memoria, l'MMU determina se quella posizione è memorizzata nella cache dell'area B e, in caso affermativo, traduce l'indirizzo e completa il fetch. In caso contrario, si verifica una cache miss e la MMU gestisce il fetch e si occupa di recuperare i dati dalla memoria secondaria (nell'area D) nella cache (area B). Se anche la posizione non è presente nell'area D, la MMU innesca un page fault, il che può comportare la gestione da parte del software dello spostamento dei dati dal disco alla memoria. In questo modo, il programma ha l'illusione di avere a disposizione una grande quantità di memoria, ma i tempi di accesso alla memoria diventano piuttosto difficili da prevedere. Non è raro che i tempi di accesso alla memoria varino di un fattore 1000 o più, a seconda di come gli indirizzi logici siano distribuiti sulle memorie fisiche. Data la sensibilità del tempo di esecuzione all'architettura della memoria, è importante capire l'organizzazione e il funzionamento delle cache.

**Basic Cache Organization**

Supponiamo che ogni indirizzo in un sistema di memoria sia composto da m bit, per un massimo di M = 2m indirizzi unici. Una memoria cache è organizzata come un array di S = 2s **cache set**. Ogni set di cache comprende a sua volta E **cache lines**. Una linea di cache memorizza un singolo blocco di B = 2b byte di dati, insieme a bit di validità e tag. Il bit valido indica se la chace line contiene informazioni significative, mentre il tag (composto da t = m - s - b bit) identifica in modo univoco il blocco memorizzato nella cache line. La Figura 9.2 illustra l'organizzazione della cache e il formato degli indirizzi.



Pertanto, una cache può essere caratterizzata dalla tupla (m, S, E, B). Questi parametri sono riassunti nella Tabella 9.1. La dimensione complessiva della cache C è data da C = S × E × B byte. Supponiamo che un programma legga il valore memorizzato all'indirizzo a. Assumiamo per il resto della questa sezione che questo valore sia una singola parola di dati w. La CPU invia prima l'indirizzo alla cache per determinare se il valore è presente nella cache. L'indirizzo a può essere visto come suddiviso in tre segmenti di bit: i primi t bit codificano il tag, i successivi s bit codificano l'indice del set e gli ultimi b bit codificano la posizione della parola all'interno del blocco. Se w è presente nella cache, l'accesso alla memoria è un **cache hit**, altrimenti è un **cache miss**. Le cache sono suddivise in classi in base al valore di E. Passiamo ora in rassegna queste categorie di memorie cache e ne descriviamo brevemente il funzionamento.

Immagine che contiene testo

Descrizione generata automaticamente

**Direct-Mapped Caches**

Una cache con esattamente una riga per set (E = 1) è chiamata cache a mappatura diretta. Per tale cache, data una parola w richiesta dalla memoria, dove w è memorizzata all'indirizzo a, ci sono tre passi per determinare se w è un hit o un miss:

1. **Set Selection**: I bit s che codificano il set vengono estratti dall'indirizzo a e utilizzati come indice per selezionare il set di cache corrispondente.
2. **Line Matching**: Il passo successivo consiste nel verificare se una copia di w è presente nella linea di cache unica per questo set. A tale scopo, si controllano i bit di validità e di tag per quella riga di cache. Se il bit di validità è impostato e i bit di tag della riga corrispondono a quelli dell'indirizzo a, allora la parola è presente nella riga di cache e si ha un riscontro nella cache. In caso contrario, abbiamo una cache miss.
3. **Word Selection**: Una volta che si sa che la parola è presente nel blocco di cache, si utilizzano i bit dell'indirizzo a che codificano la posizione della parola all'interno del blocco per leggere quella parola di dati.

In caso di cache miss, la parola w deve essere richiesta dal livello successivo della gerarchia di memoria. Una volta recuperato, il blocco sostituisce il blocco che occupa attualmente la linea di cache per w. Sebbene una cache a mappatura diretta sia semplice da comprendere e da implementare, può soffrire di conflitti. Un conflitto miss si verifica quando le parole di due o più blocchi che si mappano sulla stessa riga della cache vengono accedute ripetutamente, in modo che l'accesso a un blocco eviti l'altro dando luogo a una serie di miss della cache. Le cache set-associative possono aiutare a risolvere questo problema.

**Set-Associative Caches**

Una set-associative-caches può memorizzare più di una linea di cache per set. Se ogni set di una cache può memorizzare E righe, dove 1 < E < C/B, allora la cache è detta E-way set-associative cache. Il termine "associativo" deriva dalla memoria associativa, che è una memoria che si indirizza in base al suo contenuto. Cioè, ogni parola della memoria è memorizzata insieme a una chiave univoca e viene recuperata utilizzando la chiave piuttosto che l'indirizzo fisico che indica dove è memorizzata. Una memoria associativa è anche chiamata **content-addressable memory**. Per una set-associative cache, l'accesso a una parola w all'indirizzo a consiste nei seguenti passi:

1. **Set Selection**: Questo passaggio è identico a quello di una cache a mappatura diretta.
2. **Line Matching**: Questo passaggio è più complicato rispetto a una cache a mappatura diretta, perché potrebbero esserci più righe in cui w potrebbe trovarsi; cioè, i bit di tag di a potrebbero corrispondere ai bit di tag di qualsiasi riga del suo set di cache. Operativamente, ogni insieme in una cache set-associativa può essere visto come una memoria associativa, dove le chiavi sono la concatenazione dei bit di tag e di quelli validi, mentre i valori dei dati sono il contenuto del blocco corrispondente.
3. **Word Selection**: Una volta trovata la linea di cache, la selezione della parola viene eseguita come per una cache a mappatura diretta.

In caso di miss, la sostituzione della linea di cache può essere più complessa rispetto a quella di una cache direct-mapped. In quest'ultimo caso, non c'è possibilità di scelta nella sostituzione, poiché il nuovo blocco sostituirà il blocco attualmente presente nella linea di cache. Tuttavia, nel caso di una cache set-associativa, abbiamo la possibilità di selezionare la linea di cache da cui eliminare un blocco.

Una politica comune è quella del "least-recently used" (LRU), in cui viene sfrattata la linea di cache il cui accesso più recente è avvenuto il più lontano nel tempo. Un'altra politica comune è quella del first-in, first-out (FIFO), in cui la riga della cache che viene eliminata è quella che è rimasta nella cache per più tempo, indipendentemente dall'ultimo accesso. Buone politiche di sostituzione della cache sono essenziali per ottenere buone prestazioni della cache. Si noti anche che l'implementazione di queste richiede una memoria aggiuntiva per ricordare l'ordine di accesso,e la quantità di memoria aggiuntiva varia da politica a politica e da implementazione a implementazione. Una cache completamente associativa è quella in cui E = C/B, cioè esiste un solo insieme. Per una cache di questo tipo la corrispondenza delle righe può essere molto costosa per una cache di grandi dimensioni, perché la memoria associativa è costosa. Per questo motivo, le cache completamente associative sono tipicamente utilizzate solo per cache di piccole dimensioni, come i Translation Lookaside Buffer (TLB) menzionati in precedenza.

**Memory Models**

Un modello di memoria definisce come la memoria viene utilizzata dai programmi. L'hardware, il sistema operativo (se presente), il linguaggio di programmazione e il suo compilatore contribuiscono al modello di memoria. In questa sezione vengono discussi alcuni dei problemi comuni che si presentano con i modelli di memoria.

**Memory Addresses**

Come minimo, un modello di memoria definisce un intervallo di indirizzi di memoria accessibili al programma. In C, questi indirizzi sono memorizzati in puntatori. In un'architettura a 32 bit, gli indirizzi di memoria sono numeri interi senza segno a 32 bit, in grado di rappresentare gli indirizzi da 0 a 232 - 1, ovvero circa quattro miliardi di indirizzi. Ogni indirizzo si riferisce a un byte (otto bit) della memoria.Il tipo di dati C char fa riferimento a un byte. Il tipo di dati C int fa riferimento a una sequenza di almeno due byte. In un'architettura a 32 bit, in genere fa riferimento a quattro byte, in grado di rappresentare numeri interi da -231 a 231 - 1. Il tipo di dati double in C fa riferimento a una sequenza di otto byte codificati secondo lo standard IEEE in virgola mobile (IEEE 754). Poiché un indirizzo di memoria si riferisce a un byte, quando si scrive un programma che manipola direttamente gli indirizzi di memoria, ci sono due problemi critici di compatibilità. Il primo è l'allineamento dei dati. Un int occuperà in genere quattro byte consecutivi a partire da un indirizzo che è un multiplo di quattro. In notazione esadecimale questi indirizzi terminano sempre con 0, 4, 8 o c.

Il secondo problema è l'ordine dei byte. Il primo byte (a un indirizzo che termina con 0, 4, 8 o c) può rappresentare gli otto bit di ordine basso dell'int (una rappresentazione chiamata **little endian**), oppure può rappresentare gli otto bit di ordine alto dell'int (una rappresentazione chiamata big endian). Purtroppo, sebbene molte questioni relative alla rappresentazione dei dati siano diventate standard universali (come l'ordine dei bit in un byte), l'ordine dei byte non è una di queste. Le architetture x86 di Intel e i processori ARM, per impostazione predefinita, utilizzano una rappresentazione little-endian, mentre PowerPC di IBM utilizza la rappresentazione big endian. Alcuni processori li supportano entrambi. L'ordine dei byte è importante anche nei protocolli di rete, che generalmente utilizzano big endian.

La terminologia deriva da I viaggi di Gulliver, di Jonathan Swift, dove un editto reale a Lilliput richiede di aprire l'uovo alla coque dalla parte più piccola, mentre nel regno rivale di Blefuscu gli abitanti rompono il loro dalla parte grande.

**Stacks**

Lo stack è una regione di memoria che viene allocata dinamicamente al programma secondo uno schema last-in, (LIFO). Un puntatore allo stack (in genere un registro) contiene l'indirizzo di memoria della parte superiore dello stack. Quando un elemento viene inserito nello stack, il puntatore allo stack viene incrementato e l'elemento viene memorizzato nella nuova posizione a cui fa riferimento lo stack pointer. Quando un elemento viene tolto dallo stack, la posizione di memoria a cui fa riferimento il puntatore allo stack viene copiata da qualche altra parte (ad esempio, in un registro) e il puntatore allo stack viene decrementato. Gli stack sono tipicamente utilizzati per implementare le chiamate di procedura. Data una chiamata di procedura in C, il compilatore produce codice che spinge sullo stack la posizione dell'istruzione da eseguire al ritorno dalla procedura, il valore corrente di alcuni o tutti i registri della macchina, gli argomenti della procedura, e poi imposta il program counter uguale alla posizione del codice della procedura. I dati di una procedura che vengono inseriti nello stack sono noti come stack frame di quella procedura. Quando una procedura ritorna, il compilatore apre il suo stack frame, recuperando infine la posizione del programma dalla quale riprendere l'esecuzione. Per il software embedded, può essere disastroso se il puntatore allo stack viene incrementato oltre la memoria allocata per lo stack. Un tale overflow dello stack può portare alla sovrascrittura della memoria che viene utilizzata per altri scopi, portando a risultati imprevedibili. Limitare l'uso dello stack è quindi un obiettivo importante. Questo diventa particolarmente difficile con i programmi ricorsivi in cui una procedura richiama sé stessa. I progettisti di software embedded **spesso evitano di usare ricorsione** per aggirare questa difficoltà.

Errori più sottili possono derivare da un uso improprio o da un'errata comprensione dello stack. Si consideri il seguente programma C:

Immagine che contiene testo, persona

Descrizione generata automaticamente

La variabile b è una variabile locale, la cui memoria si trova nello stack. Quando la procedura ritorna, la variabile c conterrà un puntatore a una posizione di memoria al di sopra del puntatore della pila. Il contenuto di questa posizione di memoria verrà sovrascritto quando gli elementi verranno successivamente inseriti nella pila. Pertanto, non è corretto che la procedura foo restituisca un puntatore a b. Nel momento in cui tale puntatore viene de-referenziato (cioè, se una riga di main fa riferimento a \*c dopo la riga 8), la posizione di memoria potrebbe contenere qualcosa di completamente diverso da ciò che è stato assegnato in foo. Sfortunatamente, il C non fornisce alcuna protezione contro tali errori.

**Memory Protection Units**

Un problema fondamentale nei sistemi che supportano più attività simultanee è quello di evitare che un'attività interrompa l'esecuzione di un’altra. Questo aspetto è particolarmente importante nelle applicazioni embedded che permettono di scaricare software di terze parti, ma può anche fornire un’importante difesa contro i bug del software nelle applicazioni critiche per la sicurezza. Molti processori forniscono una protezione della memoria in hardware. Ai task viene assegnato un proprio spazio di indirizzi e se un task tenta di accedere alla memoria al di fuori del proprio spazio di indirizzi, si verifica un errore di segmentazione o un'altra eccezione. In genere questo comporta la terminazione dell'applicazione incriminata.

**Dynamic Memory Allocation**

Le applicazioni software di uso generale hanno spesso requisiti di memoria indeterminati, che dipendono dai parametri e/o dall'input dell'utente. Per supportare tali applicazioni, gli informatici hanno sviluppato schemi di allocazione dinamica della memoria, in cui un programma può in qualsiasi momento richiedere al sistema operativo l'allocazione di memoria aggiuntiva. La memoria viene allocata da una struttura di dati nota come heap, che facilita la traccia di quale memoria è in uso da parte di quale applicazione. L'allocazione della memoria avviene tramite una chiamata di sistema operativo (come malloc in C). Quando il programma non ha più bisogno di accedere alla memoria così allocata, la dealloca (chiamando free in C). Il supporto per l'allocazione della memoria spesso (ma non sempre) include il garbage collection. Per esempio, il garbage collection è intrinseco al linguaggio di programmazione Java. Un garbage è un'attività che viene eseguita periodicamente o quando la memoria si restringe e che analizza le strutture di dati che un programma ha allocato e libera automaticamente le porzioni di memoria che non sono più referenziate all'interno del programma. In linea di principio, quando si utilizza un garbage collector, il programmatore non deve preoccuparsi di liberare esplicitamente la memoria. Con o senza garbage collection, è possibile che un programma accumuli inavvertitamente memoria che non viene mai liberata. Questo fenomeno è noto come memory leak e per le applicazioni embedded, che in genere devono continuare a essere eseguite per molto tempo, può essere disastroso. Il programma finirà per fallire quando la memoria fisica sarà esaurita. Un altro problema che si presenta con gli schemi di allocazione della memoria è la frammentazione della memoria. Questo si verifica quando un programma alloca e dealloca in modo caotico memoria di dimensioni variabili. Una memoria frammentata ha pezzi di memoria allocati e liberi intervallati, e spesso i pezzi di memoria liberi diventano troppo piccoli per essere utilizzati. In questo caso è necessaria la deframmentazione. La deframmentazione e la garbage collection sono entrambe molto problematiche per i sistemi in tempo reale. Le implementazioni più semplici di questi compiti richiedono l'interruzione di tutti gli altri compiti in esecuzione mentre la deframmentazione o il garbage collection sono attivi. Le implementazioni che utilizzano queste tecniche di "arresto del mondo" possono avere tempi di pausa considerevoli, che a volte durano molti millisecondi. Altri task non possono essere eseguiti durante questo tempo perché i riferimenti ai dati all'interno delle strutture dati (puntatori) sono inconsistenti durante il task. Una tecnica che può ridurre i tempi di pausa è la garbage collection incrementale, che isola le sezioni di memoria e le raccoglie separatamente. Al momento in cui scriviamo, tali tecniche sono sperimentali e non ampiamente diffuse.

**Memory Model of C**

I programmi C memorizzano i dati sullo stack, sull'heap e in posizioni di memoria fissate dal compilatore. Consideriamo il seguente programma C:

Immagine che contiene testo

Descrizione generata automaticamente

In questo programma, la variabile a è una variabile globale perché è dichiarata al di fuori di ogni definizione di procedura. Il compilatore le assegnerà una posizione di memoria fissa. Le variabili b e c sono parametri, che vengono allocati nello stack quando viene chiamata la procedura foo (il compilatore potrebbe anche metterle nei registri piuttosto che nello stack). Le variabili d ed e sono variabili automatiche o variabili locali. Sono dichiarate all'interno di una procedura (in questo caso, main). Il compilatore alloca lo spazio sullo stack per loro. Quando la procedura foo viene richiamata alla riga 11, la posizione in pila di b acquisirà una copia del valore della variabile d assegnata alla riga 8. Si tratta di un esempio di pass by value, in cui il valore di un parametro viene copiato sullo stack per essere utilizzato dalla procedura chiamata. I dati referenziati dal puntatore e, invece, è memorizzato nella memoria allocata sull'heap, e quindi viene passato per riferimento (il puntatore ad esso, e, viene passato per valore). L'indirizzo è memorizzato nella posizione dello stack per c. Se foo include un'assegnazione a \*c, dopo il ritorno di foo, quel valore può essere letto dereferenziando e. Alla riga 1 viene assegnato un valore iniziale alla variabile globale a. C'è una sottile insidia qui, tuttavia. La posizione di memoria che contiene a sarà inizializzata con il valore 2 quando il programma viene caricato. Questo significa che se il programma viene eseguito una seconda volta senza ricaricarlo, il valore iniziale di a non sarà necessariamente 2! Il suo valore sarà quello che aveva al termine della prima invocazione del programma. Nella maggior parte dei sistemi operativi desktop, il programma viene ricaricato a ogni esecuzione, quindi questo problema non si presenta. Ma in molti sistemi embedded, il programma non viene necessariamente ricaricato a ogni esecuzione. Il programma può essere eseguito dall'inizio, ad esempio ogni volta che il sistema viene resettato. Per evitare questo problema, è più sicuro inizializzare le variabili globali nel corpo di main, piuttosto che nella riga di dichiarazione, come fatto in precedenza.

Un progettista di sistemi embedded deve comprendere l'architettura di memoria del computer di destinazione e il modello di memoria del linguaggio di programmazione. Un uso errato della memoria possono portare a errori estremamente sottili, alcuni dei quali non si manifesteranno durante i test. Errori che si manifestano solo in un prodotto in produzione possono essere disastrosi, sia per l'utente del sistema che per il fornitore di tecnologia. In particolare, il progettista deve capire quali porzioni dello spazio degli indirizzi si riferiscono alla memoria volatile e a quella non volatile. Per le applicazioni sensibili al tempo (che sono la maggior parte dei sistemi embedded), il progettista deve anche conoscere la tecnologia della memoria e l'architettura della cache (se presente) per poter capire i tempi di esecuzione del programma. Inoltre, il programmatore deve conoscere il modello di memoria del linguaggio di programmazione per per evitare di leggere dati che potrebbero essere non validi. Inoltre, il programmatore deve essere molto attenti all'allocazione dinamica della memoria, in particolare per i sistemi embedded che si prevede funzionino per un periodo di tempo molto lungo. L'esaurimento della memoria disponibile può provocare il crash di sistema o altri comportamenti indesiderati.