**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.

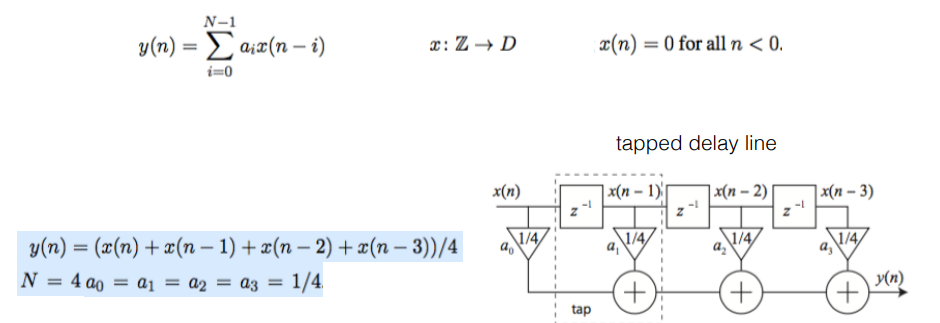
Il video utilizza in genere frequenze di campionamento di 25 o 30 Hz per i dispositivi consumer e frequenze molto più elevate per le applicazioni di misura speciali. Ogni campione, ovviamente contiene un'intera immagine (chiamata fotogramma), che a sua volta ha molti campioni (chiamati pixel) distribuiti 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.

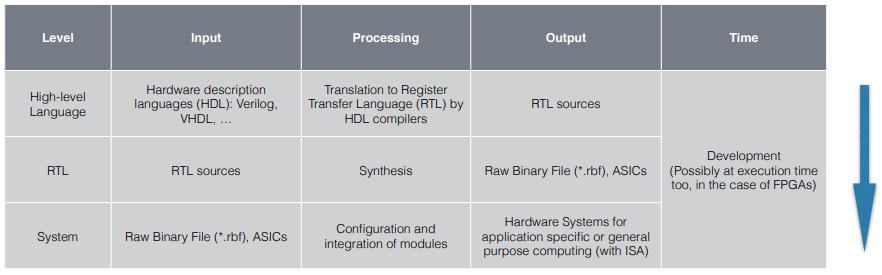
L'esecuzione del software applicativo richiede diverse fasi di elaborazione in **fase di sviluppo** e **di esecuzione**.

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.**

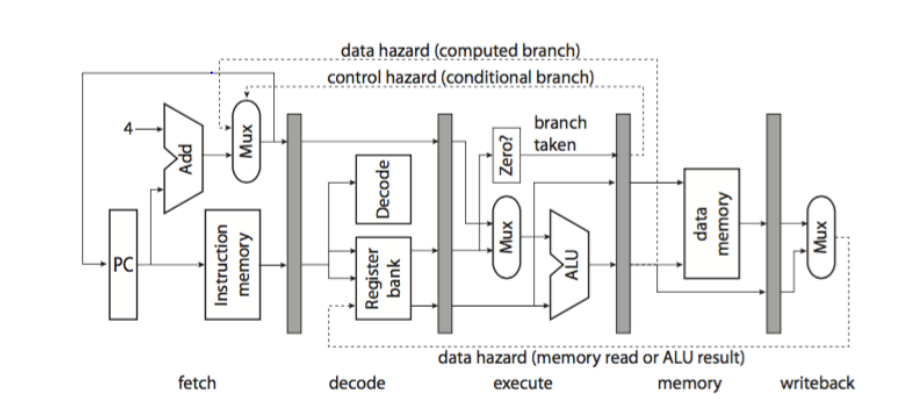


Immagine che contiene testo

Descrizione generata automaticamente

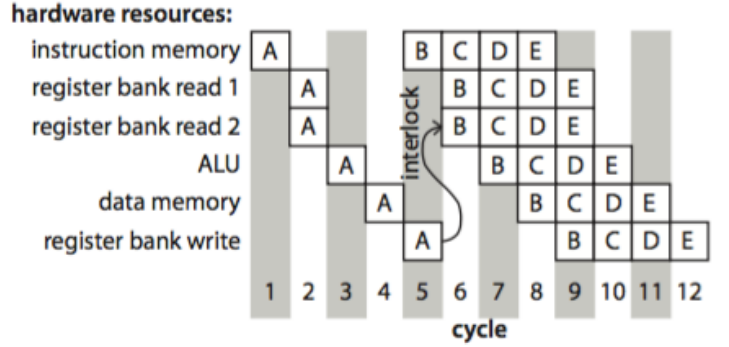
**Addressing data hazards: explicit pipeline**

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:**

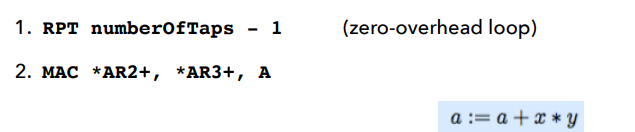
Tranne che per le pipeline esplicite e le ramificazioni ritardate, tutte le tecniche introducono una variabilità nei tempi di esecuzione di una sequenza di istruzioni. L'analisi dei tempi di un programma può diventare estremamente difficile. Le pipeline esplicite sono relativamente comuni nei processori 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.

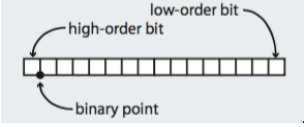
**Multicore Architectures**

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.

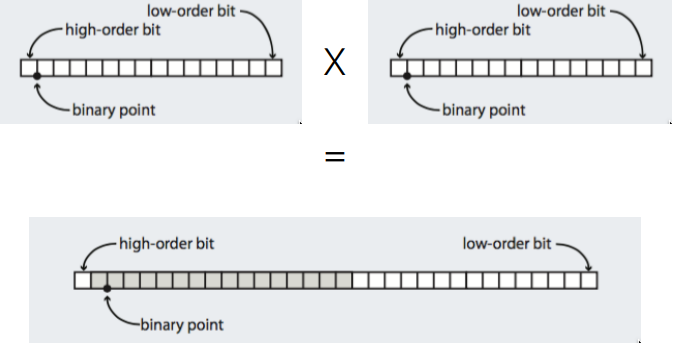
Questa mancanza di interferenza è più problematica nelle architetture multicore generiche. È comune, ad esempio, l'uso di cache multilivello, in cui il secondo livello o quello superiore è condiviso tra i core. Sfortunatamente, tale condivisione rende molto difficile isolare il comportamento in tempo 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**

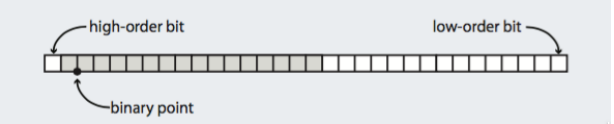
Molti processori embedded forniscono hardware solo per l'aritmetica dei numeri interi. L'aritmetica dei numeri interi può essere utilizzata per i numeri non interi, con qualche accortezza. Dato, ad esempio, un numero intero a 16 bit, un programmatore può immaginare un punto binario, che è come un punto 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.



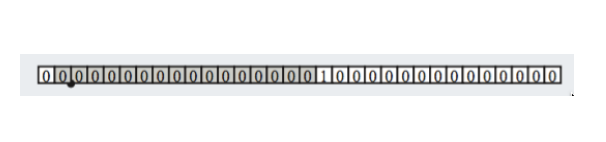
Tuttavia, l'estrazione di 16 bit da un risultato di 32 bit comporta una perdita di informazioni. In primo luogo, c'è la possibilità di un overflow, perché scartiamo il bit di ordine superiore. Supponiamo che i due numeri moltiplicati siano entrambi -1:

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**

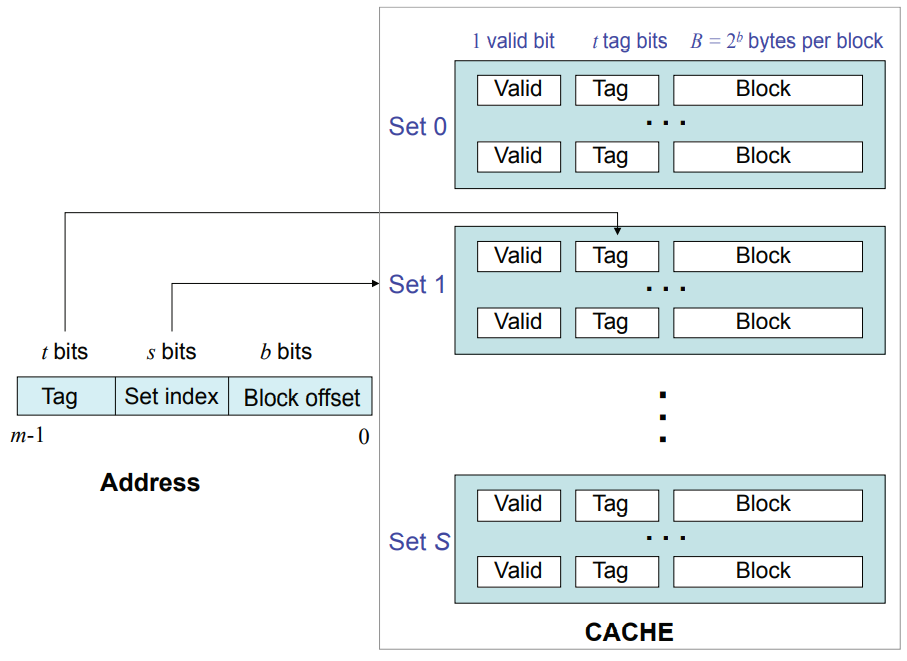
Molte applicazioni embedded combinano tecnologie di memoria. Alcune memorie vengono accedute prima di altre; diciamo che le prime sono più "vicine" al processore rispetto alle seconde. Per esempio, una memoria vicina (SRAM) è tipicamente utilizzata per memorizzare temporaneamente i dati di lavoro mentre 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.

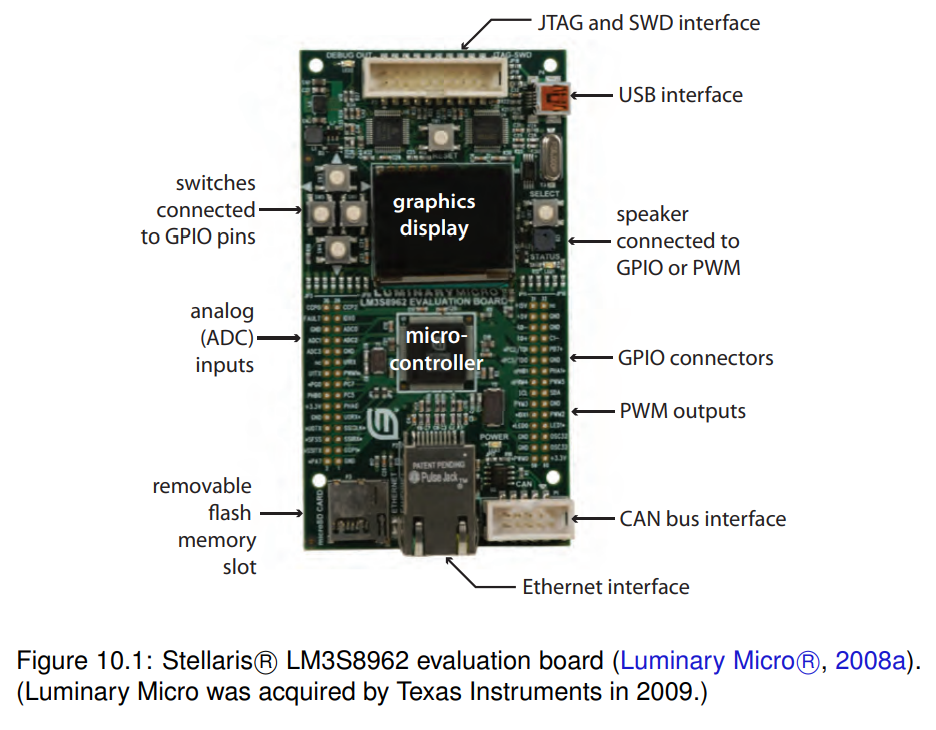
Un progettista di sistemi embedded deve comprendere l'architettura di memoria del computer di destinazione e il modello di memoria del linguaggio di programmazione. Un uso errato della memoria possono portare a errori estremamente sottili, alcuni dei quali non si manifesteranno durante i test. Errori 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.

**Input and Output**

Poiché i sistemi cyber-fisici integrano dinamiche informatiche e fisiche, i meccanismi dei processori che supportano l'interazione con il mondo esterno sono fondamentali per qualsiasi progettazione. Il progettista di un sistema deve affrontare una serie di questioni. Tra questi, le proprietà meccaniche ed elettriche delle interfacce sono importanti. L'uso non corretto delle parti, come l'assorbimento di troppa corrente da un pin, può causare il malfunzionamento del sistema o ridurne la vita utile. Inoltre, nel mondo fisico accadono molte cose contemporaneamente. Il software, invece, è per lo più sequenziale. Conciliare queste due proprietà disparate è una sfida importante e spesso rappresenta il principale fattore di rischio nella progettazione di sistemi embedded. Le interazioni errate tra il codice sequenziale e gli eventi concomitanti del mondo fisico possono causare guasti drammatici al sistema.

I processori embedded, siano essi microcontrollori, processori DSP o processori generici, includono tipicamente una serie di meccanismi di ingresso e uscita (I/O) sul chip, esposti ai progettisti come pin del chip. In questa sezione vengono esaminate alcune delle interfacce più comuni, illustrandone le proprietà.

La Figura 10.1 mostra una scheda di valutazione per il Luminary Micro Stellaris R, un processore ARM CortexTM - M3 a 32 bit. Il microcontrollore stesso si trova al centro, sotto il display grafico. Molti dei pin del microcontrollore sono disponibili sui connettori mostrati su entrambi i lati del microcontrollore e nella parte superiore e inferiore della scheda. Una scheda di questo tipo verrebbe utilizzata per prototipare un'applicazione embedded e nel prodotto finale verrebbe sostituita da una scheda di circuito personalizzata che includa solo l'hardware richiesto dall'applicazione. Un ingegnere svilupperà il software per la scheda utilizzando un ambiente di sviluppo integrato (IDE) fornito dal fornitore e caricherà il software sulla memoria flash nello slot sul fondo della scheda. In alternativa, il software potrebbe essere caricato sulla scheda attraverso l'interfaccia USB, situata nella parte superiore, dal computer di sviluppo.



**Pulse Width Modulation**

La modulazione dell'ampiezza degli impulsi (PWM) è una tecnica che consente di erogare in modo efficiente una quantità variabile di potenza a dispositivi hardware esterni. Può essere utilizzata per controllare, ad esempio, la velocità dei motori elettrici, la luminosità di una luce LED e la temperatura di un elemento riscaldante. In generale, può fornire quantità variabili di energia a dispositivi che tollerano variazioni rapide e brusche di tensione e corrente. L'hardware PWM utilizza solo circuiti digitali ed è quindi facile da integrare nello stesso chip con un microcontrollore. I circuiti digitali, per loro stessa natura, producono solo due livelli di tensione, alto e basso. Un segnale PWM commuta rapidamente tra alto e basso a una frequenza fissa, variando la quantità di tempo in cui mantiene il segnale alto. Il duty cycle è la percentuale di tempo in cui la tensione è alta. Se il duty cycle è del 100%, la tensione è sempre alta. Se il duty cycle è dello 0%, la tensione è sempre bassa. Molti microcontrollori forniscono periferiche PWM (vedi Figura 10.1). Per utilizzarle, un programmatore scrive un valore in un registro mappato in memoria per impostare il duty cycle (anche la frequenza può essere impostabile). Il dispositivo fornisce quindi energia all'hardware esterno in proporzione al ciclo di lavoro specificato. Il PWM è un modo efficace per erogare quantità variabili di potenza, ma solo a determinati dispositivi. Un elemento riscaldante, ad esempio, è un resistore la cui temperatura aumenta con l'aumentare della corrente che lo attraversa. La temperatura varia lentamente rispetto alla frequenza di un segnale PWM. La temperatura varia lentamente, rispetto alla frequenza di un segnale PWM, per cui la tensione del segnale, che varia rapidamente, viene mediata dal resistore e la temperatura sarà molto vicina alla costante per un ciclo di lavoro fisso. Allo stesso modo, i motori mediano le rapide variazioni della tensione di ingresso. Lo stesso vale per le luci a incandescenza e a LED. Qualsiasi dispositivo la cui risposta alle variazioni di corrente o tensione è lenta rispetto alla frequenza del segnale PWM è un candidato per essere controllato tramite PWM.

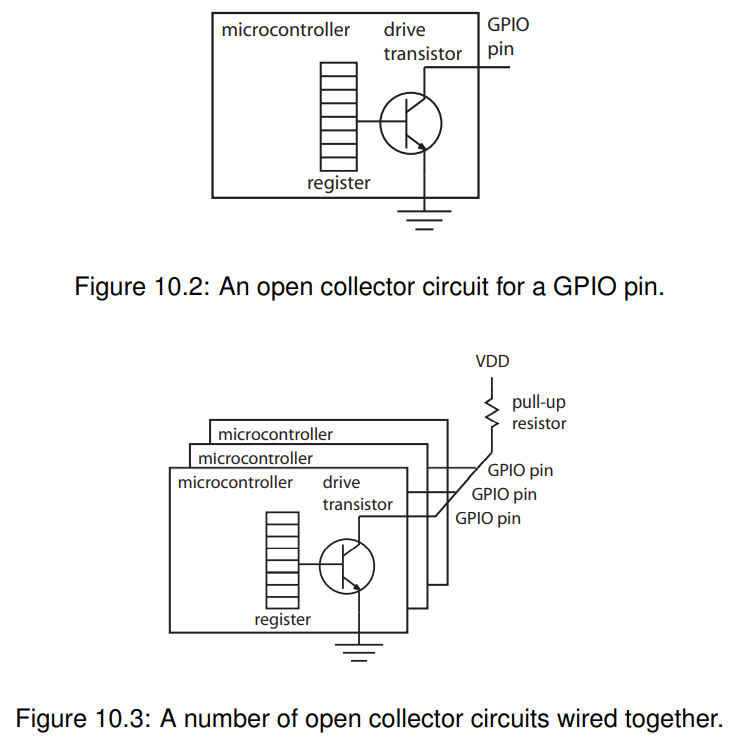
**General-Purpose Digital I/O**

I progettisti di sistemi embedded hanno spesso la necessità di collegare hardware digitale specializzato o personalizzato ai processori embedded. Molti processori embedded dispongono di una serie di pin di I/O generici (GPIO), che consentono al software di leggere o scrivere livelli di tensione che rappresentano uno zero o un uno logico. Se la tensione di alimentazione del processore è VDD, in logica active high una tensione vicina a VDD rappresenta un uno logico, mentre una tensione vicina a zero rappresenta uno zero logico. In logica attiva bassa, queste interpretazioni sono invertite. In molti progetti, un pin GPIO può essere configurato come uscita. Ciò consente al software di scrivere su un registro mappato in memoria per impostare la tensione di uscita come alta o bassa. Con questo meccanismo, il software può controllare direttamente i dispositivi fisici esterni. Tuttavia, è necessario prestare attenzione. Quando si interfaccia l'hardware ai pin GPIO, il progettista deve conoscere le specifiche del dispositivo. In particolare, i livelli di tensione e corrente variano a seconda del dispositivo. Se un pin GPIO produce una tensione di uscita pari a VDD quando gli viene assegnato un valore logico pari a una, il progettista deve conoscere i limiti di corrente prima di collegarvi un dispositivo ad esso. Se, ad esempio, vi si collega un dispositivo con una resistenza di R ohm, la legge di Ohm ci dice che la corrente in uscita sarà I = VDD/R.

È essenziale mantenere questa corrente entro le tolleranze specificate. Il superamento di queste tolleranze potrebbe causare il surriscaldamento e il guasto del dispositivo. Potrebbe essere necessario un amplificatore di potenza per fornire una corrente adeguata. Un amplificatore può essere necessario anche per modificare i livelli di tensione.

Inoltre, può essere importante mantenere l'isolamento elettrico tra i circuiti del processore e i dispositivi esterni. I dispositivi esterni possono avere caratteristiche elettriche disordinate (rumorose) che rendono il processore inaffidabile se il rumore si riversa sulle linee di alimentazione o di terra del processore. Oppure il dispositivo esterno potrebbe funzionare in un regime di tensione o di potenza molto diverso da quello del processore. Una strategia utile è quella di dividere un circuito in domini elettrici, possibilmente con alimentazioni separate, che hanno relativamente poca influenza l'uno sull'altro. Dispositivi di isolamento come optoisolatori e trasformatori possono essere utilizzati per consentire la comunicazione tra domini elettrici. I primi convertono un segnale elettrico in un dominio elettrico in luce, e rilevano la luce nell'altro dominio elettrico e la riconvertono in un segnale elettrico. I secondi utilizzano l'accoppiamento induttivo tra domini elettrici. I pin GPIO possono anche essere configurati come ingressi, nel qual caso il software sarà in grado di reagire ai livelli di tensione forniti dall'esterno. Un pin di ingresso può essere attivato da Schmitt, in questo caso hanno un'isteresi. Un pin di ingresso azionato da Schmitt è meno vulnerabile ai disturbi. Prende il nome da Otto H. Schmitt, che lo inventò nel 1934 mentre era studente universitario e studiava la propagazione degli impulsi neurali nei nervi di calamaro. In molte applicazioni, diversi dispositivi possono condividere una singola connessione elettrica. Il progettista deve assicurarsi che questi dispositivi non pilotino simultaneamente la tensione di questo singolo collegamento elettrico a valori diversi, provocando un cortocircuito che porta al surriscaldamento e al guasto del dispositivo.

Le uscite GPIO possono utilizzare circuiti a collettore aperto, come mostrato nella Figura 10.2. In un circuito di questo tipo la scrittura di un uno logico nel registro (mappato in memoria) accende il transistor, il quale fa scendere la tensione sul pin di uscita fino a (quasi) zero. La scrittura di uno zero logico nel registro spegne il transistor, lasciando il pin di uscita non collegato, o "aperto". È possibile collegare un certo numero di interfacce a collettore aperto, come mostrato nella Figura 10.3. La linea condivisa è collegata a un resistore di pull-up, che porta la tensione della linea fino a VDD quando tutti i transistor sono spenti. Se un transistor viene acceso, la tensione dell'intera linea diventa (quasi) zero senza creare un cortocircuito con gli altri pin GPIO. Logicamente, tutti i registri devono contenere degli zeri perché l'uscita sia alta. Se uno qualsiasi dei registri contiene un uno, l'uscita sarà bassa. Assumendo logica attiva alta, la funzione logica eseguita è NOR, quindi un circuito di questo tipo si chiama un NOR cablato. Variando la configurazione, si possono creare analogamente OR cablati o AND cablati. Il termine "collettore aperto" deriva dal nome del terminale di un transistor bipolare. Nelle tecnologie CMOS, questo tipo di interfaccia viene generalmente chiamato interfaccia open drain. Il funzionamento è essenzialmente lo stesso. Le uscite GPIO possono anche essere realizzate con logica tristata, il che significa che oltre a produrre una tensione di uscita alta o bassa, il pin può essere semplicemente spento. Come un'interfaccia a collettore aperto, questo può facilitare la condivisione degli stessi circuiti esterni tra più dispositivi. A differenza di un'interfaccia a collettore aperto, un progetto a tristato può affermare sia tensioni alte che basse, anziché solo una delle due.



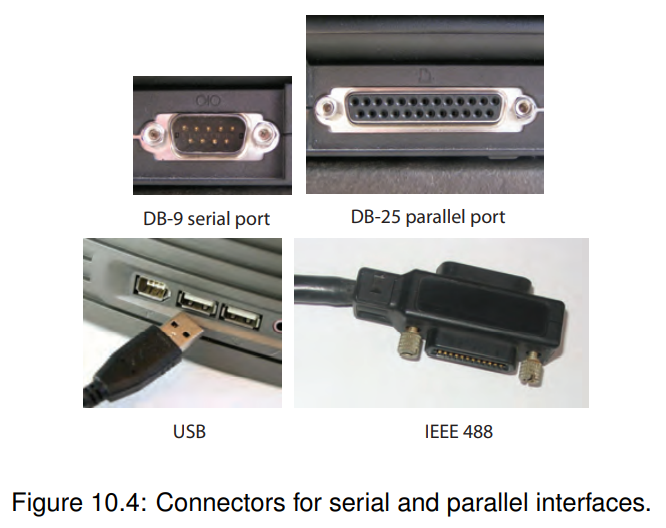
ESEMPI:

1. I pin GPIO del microcontrollore Luminary Micro StellarisR illustrati nella Figura 10.1 possono essere configurati per fornire o assorbire quantità variabili di corrente fino a 18 mA. Le combinazioni di pin in grado di gestire correnti così elevate sono limitate. Ad esempio, Luminary Micro R (2008b) afferma: "I pin del pacchetto GPIO ad alta corrente devono essere selezionati in modo che ce ne siano solo un massimo di due per ogni lato del pacchetto fisico ... con il numero totale di uscite GPIO ad alta corrente non sia superiore a quattro per l'intero pacchetto". Questi vincoli sono stati pensati per evitare il surriscaldamento del dispositivo.
2. I pin GPIO del microcontrollore mostrati nella Figura 10.1, quando sono configurati come ingressi, sono ad attivazione Schmitt.
3. Consideriamo una fabbrica in cui diversi microcontrollori indipendenti sono in grado di spegnere un macchinario affermando uno zero logico su una linea GPIO di uscita. Un progetto di questo tipo può garantire una maggiore sicurezza perché i microcontrollori possono essere ridondanti, in modo che il guasto di uno di essi non impedisca il verificarsi di uno spegnimento di sicurezza. Se tutte le linee GPIO sono collegate a un unico ingresso di controllo del macchinario, occorre prendere le dovute precauzioni per garantire che i microcontrollori non si cortocircuitino a vicenda. Questo si verificherebbe se un microcontrollore cercasse di pilotare la linea condivisa ad alta tensione, mentre un altro tenta di pilotare la stessa linea a bassa tensione.
4. I pin GPIO del microcontrollore mostrati nella Figura 10.1, quando sono configurati come uscite, possono essere specificati come circuiti di scarico aperti. Essi possono anche fornire una resistenza di pull-up, riducendo così il numero di componenti discreti esterni necessari su un circuito stampato.

**Serial Interfaces**

Uno dei vincoli principali che i progettisti di processori embedded devono affrontare è la necessità di avere pacchetti fisicamente piccoli e un basso consumo energetico. Una conseguenza è che il numero di pin sul circuito integrato del processore è limitato. Pertanto, ogni pin deve essere utilizzato in modo efficiente. Inoltre, quando si cablano i sottosistemi, il numero di fili deve essere limitato per mantenere l'ingombro e il consumo di energia. Inoltre, quando si cablano i sottosistemi, il numero di fili deve essere limitato per tenere sotto controllo l'ingombro. Pertanto, anche i fili devono essere utilizzati in modo efficiente. Un modo per usare in modo efficiente i pin e i fili è quello di inviare le informazioni in modo seriale, sotto forma di sequenze di bit. Un'interfaccia di questo tipo è chiamata interfaccia seriale. Sono stati sviluppati diversi standard per le interfacce seriali, in modo da poter collegare dispositivi di diversi produttori. Uno standard vecchio ma persistente, l'RS-232, standardizzato dall'Electronics Industries Association (EIA), è stato introdotto per la prima volta nel 1962 per collegare i telescriventi ai modem. Questo standard definisce i segnali elettrici e i tipi di connettore; persiste per la sua semplicità e per la continua diffusione di apparecchiature industriali obsolete che lo utilizzano. Lo standard definisce il modo in cui un dispositivo può trasmettere un byte a un altro dispositivo in modo asincrono (cioè senza che i dispositivi condividono un segnale di clock). Sui PC più vecchi, la connessione RS-232 può essere un connettore DB-9, come illustrato nella Figura 10.4. Un microcontrollore in genere un ricevitore/trasmettitore asincrono universale (UART) per convertire il contenuto di un registro a 8 bit in una sequenza di bit da trasmettere su un collegamento seriale RS-232. Per il progettista di un sistema embedded, un problema importante da considerare è che le interfacce RS-232 possono essere piuttosto lente e possono rallentare il software applicativo, se il programmatore non è molto attento.

Il meccanismo della RS-232 è molto semplice. Il mittente e il destinatario devono innanzitutto concordare la velocità di trasmissione (che è lenta per gli standard moderni). Il mittente inizia la trasmissione di un byte con un bit di avvio, che avverte il ricevitore dell'arrivo di un byte. Il mittente, quindi, invia la sequenza di bit alla velocità concordata, seguiti da uno o due bit di stop. L'orologio del ricevitore si azzera alla ricezione del bit di avvio e si prevede che segua l'orologio del mittente abbastanza da poter campionare il segnale in arrivo in modo sequenziale e recuperare la sequenza di bit. Esistono molti discendenti di questo standard che supportano velocità di comunicazione più elevate, come RS-422, RS-423 e altri ancora. I dispositivi più recenti, progettati per collegarsi ai personal computer, utilizzano in genere interfacce Universal Serial Bus (USB), standardizzato da un consorzio di fornitori. L'interfaccia USB 1.0 è stata introdotta nel 1996 e supporta una velocità di trasmissione dati di 12 Mbit/sec. USB 2.0 è apparso nel 2000 e supporta velocità di trasferimento dati fino a 480 Mbit/sec. USB 3.0 è apparso nel 2008 e supporta velocità di trasferimento dati fino a 4,8 Gbit/sec. L'USB è elettricamente più semplice dell'RS-232 e utilizza connettori più semplici e robusti, come mostrato nella Figura 10.4. Tuttavia, lo standard USB definisce molto di più del trasporto elettrico di byte e per supportarlo è necessaria una logica di controllo più complessa. Poiché i moderni dispositivi come stampanti, unità disco e dispositivi audio e video, sono tutti dotati di microcontrollori, il supporto del protocollo USB più complesso è ragionevole per questi dispositivi. Un'altra interfaccia seriale ampiamente implementata nei processori embedded è nota come JTAG (Joint Test Action Group). o più formalmente come porta di accesso ai test e architettura boundary-scan standard IEEE 1149.1. Questa interfaccia è apparsa a metà degli anni '80 per risolvere il problema che i pacchetti di circuiti integrati e la tecnologia delle schede a circuito stampato si erano evoluti al punto che il test dei circuiti con le sonde elettriche era diventato difficile o impossibile. I punti del circuito a cui era necessario accedere diventavano inaccessibili alle sonde. La nozione di boundary scan consente di rilevare lo stato di un confine logico di un circuito (quelli che tradizionalmente sarebbero stati i pin accessibili alle sonde) di essere letto o scritto in modo seriale attraverso i pin resi accessibili. Oggi, le porte JTAG sono ampiamente utilizzate per fornire un'interfaccia di debug ai processori embedded, consentendo a un ambiente di debug ospitato dal PC di esaminare e controllare lo stato di un processore incorporato. La porta JTAG viene utilizzata, ad esempio, per leggere lo stato dei registri del processore, per impostare punti di interruzione in un programma e per eseguire un singolo passo attraverso un programma. Una variante più recente è il debug a filo seriale (SWD), che fornisce funzionalità simili con un numero inferiore di pin. Oggi sono in uso diverse altre interfacce seriali, tra cui, ad esempio, la I2C (interintegrated circuit), SPI (serial peripheral interface bus), PCI Express (peripheral component interconnect express), FireWire, MIDI (musical instrument digital interface) e le versioni seriali di SCSI (descritte di seguito). Ognuna di queste ha il suo utilizzo. Inoltre, le interfacce di rete sono tipicamente seriali.



**Parallel Interfaces**

Un'interfaccia seriale invia o riceve una sequenza di bit in sequenza su una singola linea. Quella parallela utilizza più linee per inviare simultaneamente i bit. Naturalmente, ogni linea di un'interfaccia parallela è anche un'interfaccia seriale, ma il raggruppamento logico e l'azione coordinata di queste linee è ciò che rende l'interfaccia un'interfaccia parallela. Storicamente, una delle interfacce parallele più utilizzate è la porta per stampanti IEEE-1284, che sul PC IBM utilizzava un connettore DB-25, come mostrato nella Figura 10.4. Questa interfaccia è nata nel 1970 con la stampante Centronics modello 101, e per questo motivo viene talvolta chiamata porta per stampanti Centronics. Oggi le stampanti sono tipicamente collegate tramite USB o reti wireless. Con un'attenta programmazione, è possibile utilizzare un gruppo di pin GPIO per realizzare un'interfaccia parallela. In effetti, i progettisti di sistemi embedded si trovano talvolta a utilizzare i pin GPIO per emulare un'interfaccia non supportata direttamente dall'hardware. Sembra intuitivo che le interfacce parallele debbano fornire prestazioni più elevate rispetto alle interfacce seriali, perché vengono utilizzati più fili per l'interconnessione. Tuttavia, questo non è necessariamente così. Una sfida significativa con le interfacce parallele è il mantenimento della sincronia tra i vari fili. Questo diventa più difficile con l'aumentare della lunghezza fisica dell'interconnessione. Questo fatto, unito alla necessità di cavi più ingombranti e di un maggior numero di pin di I/O, ha fatto sì che molte interfacce tradizionalmente parallele siano state sostituite da interfacce seriali.

**Buses**

Un bus è un'interfaccia condivisa tra più dispositivi, a differenza di un'interconnessione punto-punto che collega esattamente due dispositivi. I bus possono essere interfacce seriali (come l'USB) o interfacce parallele. Un bus parallelo molto diffuso è lo SCSI (pronunciato come scuzzy, per small computer system interface), comunemente utilizzato per collegare dischi rigidi e unità a nastro ai computer. Le recenti varianti delle interfacce SCSI, tuttavia, si discostano dalla tradizionale interfaccia parallela per diventare interfacce seriali. Lo SCSI è un esempio di architettura di bus periferico, utilizzato per collegare i computer a periferiche quali schede audio e unità disco. Altri standard di bus periferici ampiamente utilizzati sono il bus ISA (architettura standard del settore, utilizzata nell'onnipresente architettura del PC IBM), PCI (interfaccia dei componenti periferici) e Parallel ATA (advanced technology attachment). Un tipo di bus periferico un po' diverso è l'IEEE-488, originariamente sviluppato più di 30 anni fa per collegare le apparecchiature di test automatizzate ai computer di controllo. Questa interfaccia è stata progettata da Hewlett Packard ed è nota anche come HP-IB (Hewlett Packard interface bus) e GPIB (general purpose interface bus). Anche molte reti utilizzano un'architettura a bus. Poiché un bus è condiviso da diversi dispositivi, qualsiasi architettura di bus deve includere un protocollo di controllo dell'accesso ai media (MAC) per arbitrare gli accessi concorrenti. Un semplice protocollo MAC prevede un singolo master del bus che interroga gli slave del bus. L'USB utilizza questo meccanismo. Un'alternativa è un bus a tempo, in cui ai dispositivi vengono assegnate fasce orarie in cui possono trasmettere (o non trasmettere, se non hanno nulla da inviare). Una terza alternativa è un token ring, in cui i dispositivi sul bus devono acquisire un token prima di poter utilizzare il mezzo condiviso e il token viene passato tra i dispositivi secondo un certo schema. Una quarta alternativa è l'utilizzo di un arbiter del bus, ovvero un circuito che gestisce le richieste per il bus in base ad alcune priorità. Una quinta alternativa è rappresentata dall'accesso multiplo (CSMA), in cui i dispositivi rilevano la portante per determinare se il mezzo è in uso prima di iniziare a usarlo, rilevano le collisioni che potrebbero verificarsi quando iniziano a usarlo e riprovare in un secondo momento quando si verifica una collisione. In tutti i casi, la condivisione del mezzo fisico ha implicazioni sulla tempistica delle applicazioni.

Un bus periferico fornisce un meccanismo di comunicazione tra dispositivi esterni e CPU. Se un dispositivo esterno dovesse trasferire una grande quantità di dati alla memoria principale, potrebbe essere inefficiente e/o dannoso richiedere alla CPU di eseguire ogni trasferimento. Un'alternativa è l'accesso diretto alla memoria (DMA). Nello schema DMA utilizzato sul bus ISA, il trasferimento viene eseguito da un dispositivo separato, chiamato controller DMA, che prende il controllo del bus e trasferisce i dati. In alcuni progetti più recenti, come il PCI, il dispositivo esterno prende direttamente il controllo del bus ed esegue il trasferimento dei dati senza l'ausilio di un controller DMA dedicato. In entrambi i casi, la CPU è libera di eseguire il software durante il trasferimento ma se il codice eseguito ha bisogno di accedere alla memoria o al bus periferico, la tempistica del programma viene disturbata dal DMA. Questi effetti sulla tempistica possono essere difficili da analizzare.

**Sequential Software in a Concurrent World**

Quando il software interagisce con il mondo esterno, la tempistica di esecuzione del software può essere fortemente influenzata. Il software è intrinsecamente sequenziale, e di solito viene eseguito il più velocemente possibile. Il mondo fisico, invece, è concomitante, con molte cose che accadono contemporaneamente e il cui ritmo è determinato dalle loro proprietà fisiche. Colmare questo disallineamento semantico è una delle principali sfide che un progettista di sistemi embedded deve affrontare.

**Interrupts and Exceptions**

Un'interruzione è un meccanismo che consente di mettere in pausa l'esecuzione di ciò che un processore sta facendo ed eseguire una sequenza di codice predefinita, chiamata routine di servizio di (ISR) o gestore di interrupt. Tre tipi di eventi possono innescare un'interruzione. Uno è un'interruzione hardware, in cui un hardware esterno modifica il livello di tensione di una linea di richiesta di interrupt. Nel caso di un'interruzione software, il programma in esecuzione attiva l'interrupt eseguendo un'istruzione speciale o scrivendo in un registro di memoria. Una terza variante è chiamata eccezione, in cui l'interrupt viene attivato da un hardware interno che rileva un errore, come ad esempio un errore di segmentazione. Per le prime due varianti, una volta completato l'ISR, il programma interrotto riprende da dove era stato interrotto. Nel caso di un'eccezione, una volta completata l'ISR, il programma che ha innescato l'eccezione non viene normalmente ripreso. Invece, il contatore del programma viene impostato in una posizione fissa dove, per esempio, il sistema operativo può terminare il programma incriminato. Al verificarsi di un trigger di interrupt, l'hardware deve innanzitutto decidere se rispondere. Se gli interrupt sono disabilitati, non risponde. Il meccanismo per abilitare o disabilitare gli interrupt varia a seconda del processore. Inoltre, è possibile che alcuni interrupt siano abilitati e altri no. Gli interrupt e le eccezioni hanno in genere delle priorità e un interrupt viene servito solo se il processore non sta già servendo un interrupt con una priorità più alta. In genere, le eccezioni hanno la priorità più alta e vengono sempre servite. Quando l'hardware decide di servire un interrupt, di solito prima disabilita gli interrupt, inserisce nello stack il contatore di programma corrente e i registri di stato del processore, e un indirizzo designato che di solito contiene un salto a un ISR. L'ISR deve memorizzare sullo stack i valori attualmente presenti nei registri che utilizzerà e ripristinarne i valori prima di tornare dall'interrupt, in modo che il programma interrotto possa riprendere il suo cammino da dove si era interrotto. La routine di servizio di interruzione o l'hardware devono anche riattivare gli interrupt prima di rientrare dall'interruzione.

I fornitori di processori offrono molte varianti dei meccanismi utilizzati nell'esempio precedente, pertanto è necessario consultare la documentazione del fornitore per il particolare processore utilizzato. Poiché il codice non è portabile (non funzionerà correttamente su un processore diverso), è bene isolare questo codice dalla logica dell'applicazione e documentare con attenzione ciò che deve essere re implementato per un nuovo processore.

I microcontrollori includono quasi sempre un certo numero di periferiche chiamate timer. Un timer programmabile a intervalli (PIT), il tipo più comune, conta semplicemente da un certo valore a zero. Il valore iniziale viene impostato scrivendo in un registro di memoria e quando il valore raggiunge lo zero, il PIT genera una richiesta di interrupt. Scrivendo su un registro di controllo con mappatura in memoria, un timer può essere impostato per attivarsi ripetutamente senza dover essere resettato dal software. Tali attivazioni ripetute saranno più precisamente periodico rispetto a quello che si otterrebbe se l'ISR riavviasse il timer ogni volta che viene invocato. Questo perché il tempo che intercorre tra il momento in cui il conteggio raggiunge lo zero nell'hardware del timer e il momento in cui il contatore viene riavviato dall'ISR è difficile da controllare e variabile. Ad esempio, se il timer raggiunge lo zero in un momento in cui gli interrupt sono disabilitati, allora ci sarà un ritardo prima che l'ISR venga invocato. Non può essere invocato prima che gli interrupt vengano riattivati.

**Atomicity**

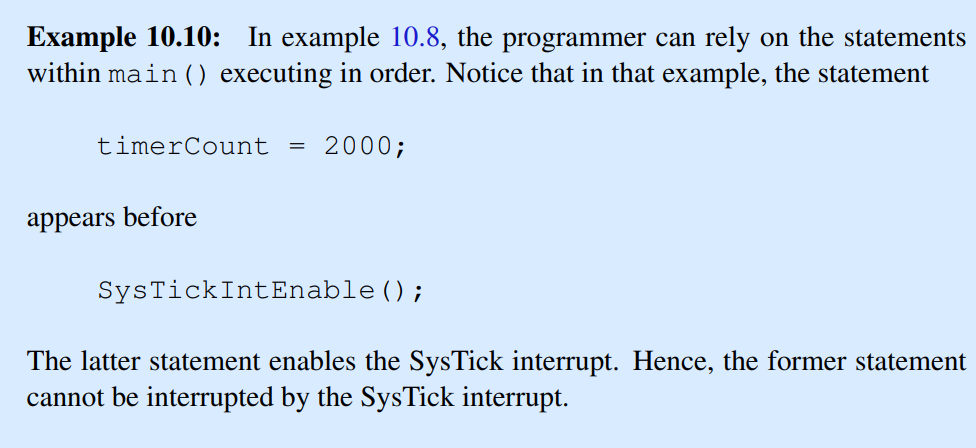
Una routine di servizio di interrupt può essere invocata tra due istruzioni qualsiasi del programma principale (o tra due istruzioni qualsiasi di un ISR a priorità inferiore). Una delle principali sfide per i progettisti di software embedded è che ragionare sulle possibili interleavings di istruzioni può diventare estremamente difficile. Nell'esempio precedente, la routine di servizio e il programma principale interagiscono attraverso una variabile condivisa, vale a dire timerCount. Il valore di questa variabile può cambiare tra due qualsiasi operazioni atomiche del programma principale. Sfortunatamente, può essere piuttosto difficile sapere quali operazioni sono atomiche. Il termine "atomico" deriva dalla parola greca "indivisibile" ed è tutt'altro che ovvio per un programmatore quali siano le operazioni atomiche. Se il programmatore sta scrivendo un codice assembly, può essere sicuro che ogni istruzione del linguaggio assembly è atomica, ma molte ISA includono istruzioni a livello di assembly che non sono atomiche.

Il set di istruzioni ARM comprende un'istruzione LDM, che carica registri multipli da posizioni di memoria consecutive. Può essere interrotta a metà del caricamento (ARM Limited, 2006).

A livello di un programma C, può essere ancora più difficile sapere quali operazioni sono atomiche. Consideriamo una singola istruzione dall'aspetto innocente:

timerCount = 2000;

Su un microcontrollore a 8 bit, questa istruzione può richiedere più di un ciclo di istruzione per essere eseguita (una parola a 8 bit non può memorizzare sia l'istruzione che la costante 2000; infatti, la costante da sola non entra in una parola a 8 bit). Potrebbe verificarsi un'interruzione durante l'esecuzione di questi cicli. Supponiamo che l'ISR scriva anche sulla variabile timerCount. In questo caso, il valore finale della variabile timerCount può essere composto da 8 bit impostati nell'ISR e dai restanti bit impostati, ad esempio, dalla riga C precedente. Il valore finale potrebbe essere molto diverso da 2000 e anche diverso dal valore specificato nella routine di servizio di interrupt. Questo bug si verifica su un microcontrollore a 32 bit? L'unico modo per saperlo con certezza è comprendere a fondo l'ISA e il compilatore. In tali circostanze, non c'è alcun vantaggio nell'aver scritto il codice in C invece che in linguaggio assembly. Bug di questo tipo in un programma sono estremamente difficili da identificare e correggere. Peggio ancora, le interleavings problematici è abbastanza improbabile che si verifichino e quindi potrebbero non apparire nei test. Per i sistemi critici per la sicurezza, i programmatori devono impegnarsi al massimo per evitare tali bug Un modo per farlo è quello di costruire i programmi utilizzando modelli di calcolo concorrente di livello superiore, come discusso nel Capitolo 6. Naturalmente, l'implementazione di questi modelli di calcolo deve essere corretta, ma presumibilmente tale implementazione è costruita da esperti di concorrenza, piuttosto che da ingegneri applicativi. Quando si lavora a livello di C e di PVR, il programmatore deve ragionare attentamente sull'ordine delle operazioni. Anche se sono possibili molte interleavings, le operazioni fornite come una sequenza di istruzioni C devono essere eseguite in ordine (più precisamente, devono comportarsi come se fossero state eseguite in ordine, anche se si utilizza l'esecuzione fuori ordine).



**Interrupt Controllers**

Il controllore di interrupt è la logica del processore che gestisce gli interrupt. Supporta un certo numero di interrupt e un certo numero di livelli di priorità. Ogni interrupt ha un vettore di interrupt, che è l'indirizzo di un ISR o un indice in un array chiamato tabella dei vettori di interrupt che contiene gli indirizzi di tutti gli ISR.

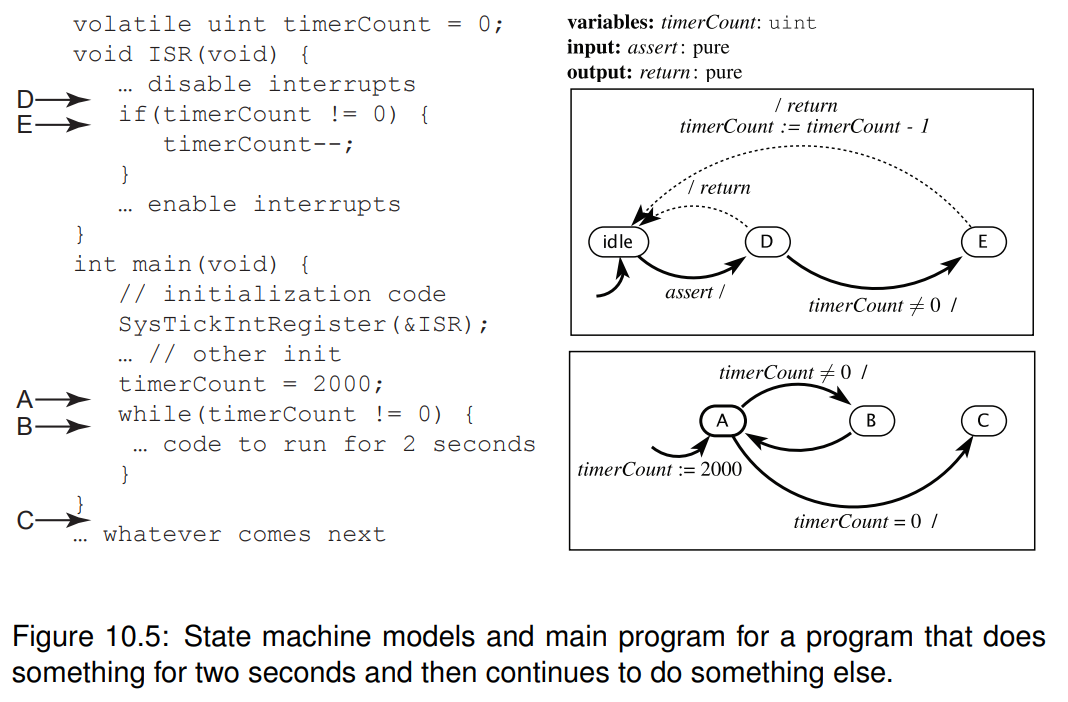
Quando si afferma un interrupt modificando la tensione su un pin, la risposta può essere **level triggered** o **edge triggered**. Per gli interrupt attivati a livello, l'hardware indica che l'interrupt è stato gestito. Per gli interrupt attivati dal bordo, l'hardware che afferma l'interrupt cambia la tensione solo per un breve periodo di tempo. In entrambi i casi, viene utilizzata linee a collettore aperto, in modo che la stessa linea fisica possa essere condivisa tra più dispositivi (naturalmente, l'ISR richiederà un meccanismo per determinare quale dispositivo ha richiesto l'interruzione, per esempio leggendo un registro di memoria in ogni dispositivo che potrebbe aver richiesto l'interrupt). La condivisione degli interrupt tra i dispositivi può essere complicata, e occorre prestare attenzione per evitare che gli interrupt a bassa priorità blocchino quelli ad alta priorità. L'asserzione degli interrupt scrivendo a un indirizzo designato su un bus ha il vantaggio che lo stesso hardware può supportare un numero maggiore di interrupt distinti, ma lo svantaggio è che i dispositivi periferici diventano più complessi. I dispositivi periferici devono includere un'interfaccia con il bus di memoria.

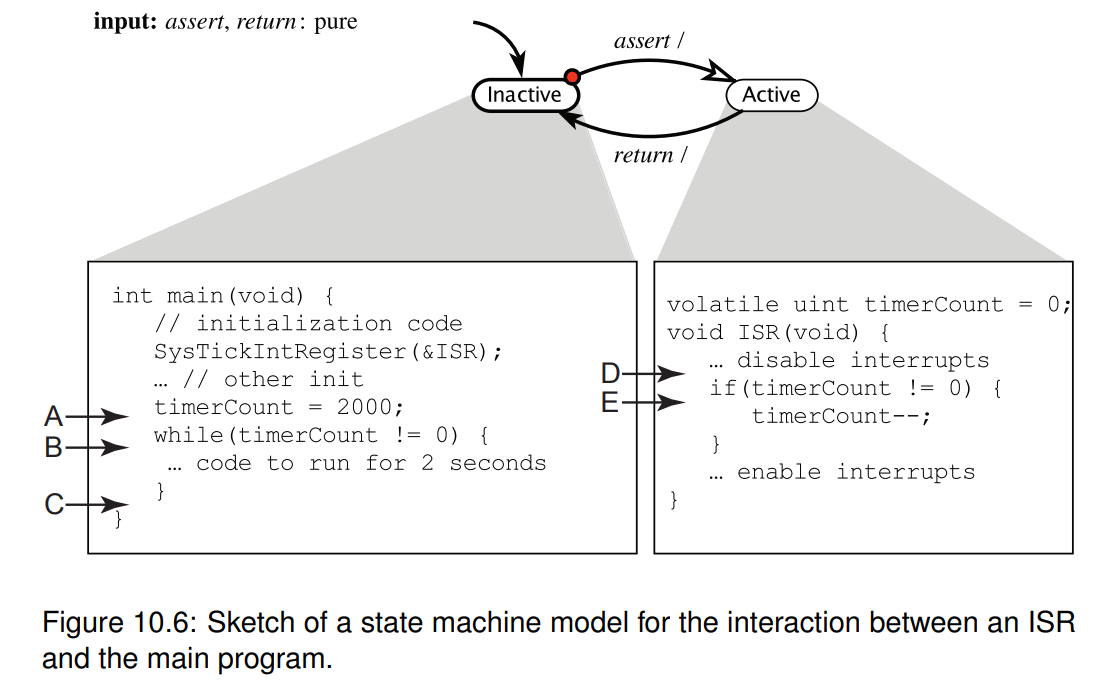
**Modeling Interrupts**

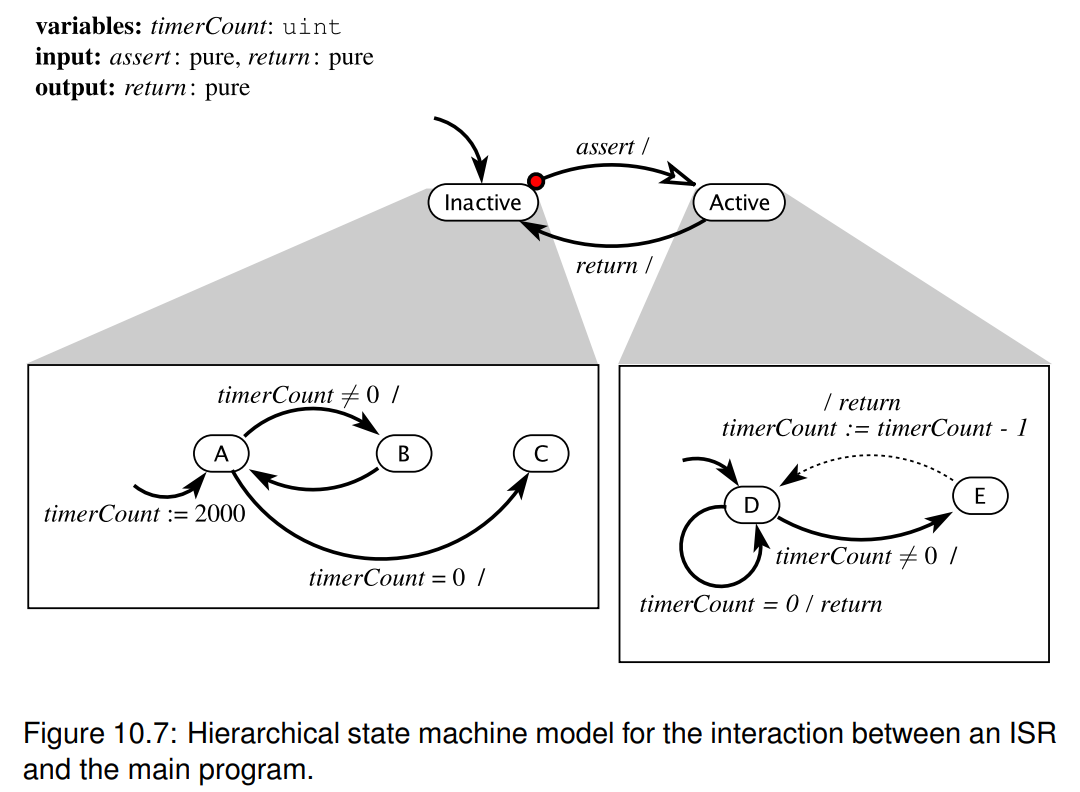
Il comportamento degli interrupt può essere difficile da comprendere appieno e molti guasti catastrofici del sistema sono causati da comportamenti inaspettati. Sfortunatamente, la logica dei controllori di interrupt è spesso descritta nella documentazione del processore in modo molto impreciso, lasciando molti comportamenti possibili non specificati. Un modo per rendere questa logica più precisa è quello di modellarla come un **FSM(finite-state machine)**.

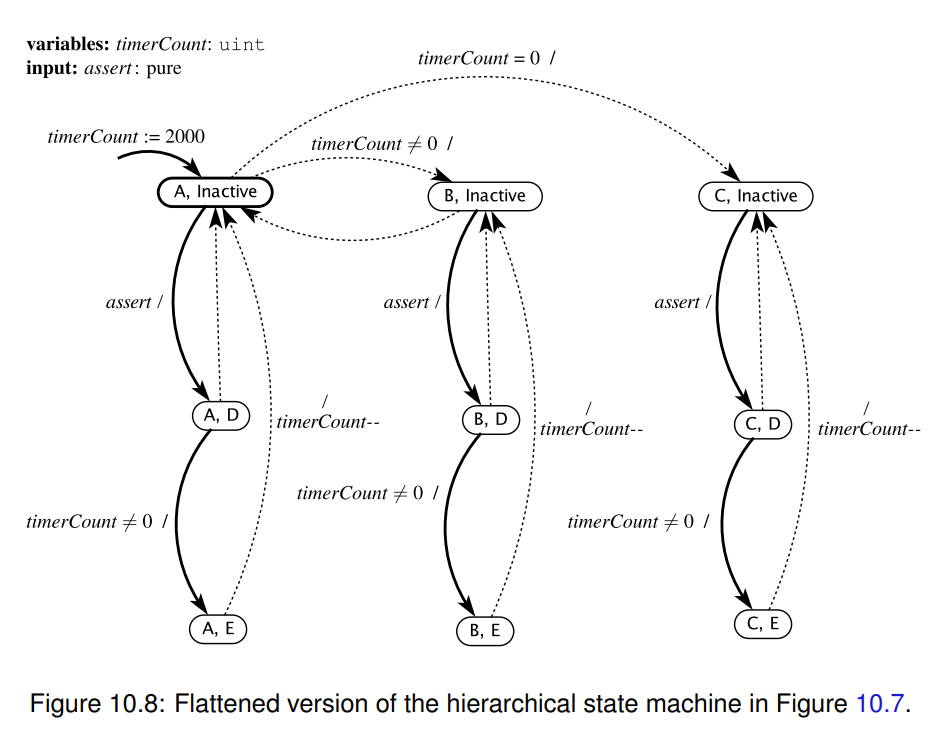
ESEMPIO.

*Il programma dell'Esempio 10.8, che esegue un'azione per due secondi, è mostrato nella Figura 10.5 insieme a due macchine a stati finiti che modellano il PVR e il programma principale. Gli stati delle FSM corrispondono a posizioni nell'esecuzione etichettate da A a E, come mostrato nel listato del programma. Queste posizioni si trovano tra gli enunciati C, quindi assumiamo che questi enunciati siano operazioni atomiche (un'ipotesi discutibile in generale). Si potrebbe voler determinare se il programma è in grado di raggiungere sempre la posizione C. In altre parole, è possibile determinare se il programma è in grado di raggiungere la posizione C? In altre parole, possiamo affermare con sicurezza che il programma alla fine andrà oltre qualsiasi calcolo che doveva eseguire per due secondi? Un modello di macchina a stati ci aiuterà a rispondere a questa domanda. La domanda chiave diventa ora come comporre queste macchine a stati per modellare correttamente l'interazione tra le due parti di codice sequenziale nelle procedure ISR e main. È facile capire che la composizione asincrona non è la scelta giusta, perché le interleavings non sono arbitrarie. In particolare, main può essere interrotto da ISR, ma ISR non può essere interrotto da main. La composizione asincrona non riuscirebbe a cogliere questa asimmetria. Assumendo che l'interrupt venga sempre servito immediatamente dopo essere stato richiesto, desideriamo avere un modello simile a quello mostrato nella Figura 10.6. In questa figura, un FSM a due stati modella se un interrupt viene servito. La transizione da Inattivo ad Attivo è innescata da un'asserzione di ingresso puro, che modella il timer hardware che richiede il servizio di interrupt. Quando l'ISR completa la sua esecuzione, un altro ritorno di ingresso puro attiva il ritorno allo stato Inattivo. Si noti che la transizione da Inattivo ad Attivo è una transizione preventiva, indicata dal piccolo cerchio all'inizio della transizione, che suggerisce che dovrebbe essere eseguita immediatamente quando si verifica l'asserzione, e che si tratta di una transizione di reset, che suggerisce che il perfezionamento dello stato di Active dovrebbe iniziare nel suo stato iniziale al momento dell'ingresso. Se combiniamo le figure 10.5 e 10.6 otteniamo l'FSM gerarchico della figura 10.7. Si noti che il segnale di ritorno è ora sia un ingresso che un'uscita. È un'uscita prodotto dal raffinamento di stato di Active, ed è un ingresso all'FSM di livello superiore, dove innesca una transizione a Inattivo. Avere un'uscita che è anche un ingresso fornisce un meccanismo che consente a un raffinamento di stato di innescare una transizione nella sua macchina a stati contenitore.Per determinare se il programma raggiunge lo stato C, possiamo studiare la macchina a stati appiattita mostrata nella Figura 10.8. Studiando attentamente questa macchina, vediamo che in realtà non c'è alcuna garanzia che lo stato C venga raggiunto! Se, ad esempio, l'asserzione è presente in ogni reazione, allora C non viene mai raggiunto.*





–



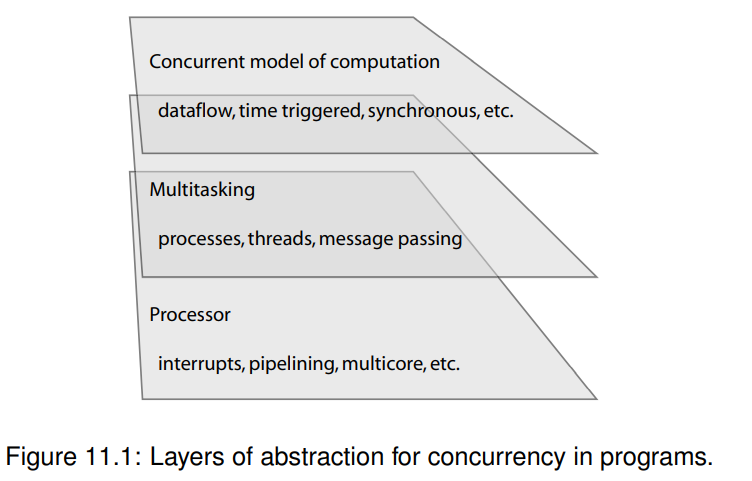
*Potrebbe accadere nella pratica? Con questo programma è improbabile, ma non impossibile. Potrebbe accadere se l'ISR stesso impiegasse più tempo per l'esecuzione tra un interrupt e l'altro. C'è qualche garanzia che ciò non accada? Sfortunatamente, la nostra unica garanzia è una vaga idea che i processori siano più veloci. Non c'è alcuna garanzia.*

Nell'esempio precedente, la modellazione dell'interazione tra un programma principale e una routine di servizio di interrupt espone una potenziale falla nel programma. Sebbene sia improbabile che il difetto si verifichi in pratica in questo esempio, il fatto che la falla sia presente è preoccupante. In ogni caso, è meglio sapere che il difetto è presente e decidere che il rischio è accettabile, piuttosto che non saperlo. I meccanismi di interruzione possono essere piuttosto complessi. Il software che utilizza questi meccanismi di I/O a un dispositivo esterno è chiamato driver di dispositivo. Scrivere driver di dispositivo che siano corretti e robusti è un compito ingegneristico impegnativo che richiede una profonda comprensione dell'architettura e una notevole abilità nel ragionare sulla concorrenza. Molti guasti nei sistemi informatici sono causati da interazioni impreviste tra i driver dei dispositivi e altri programmi.

**Multitasking**

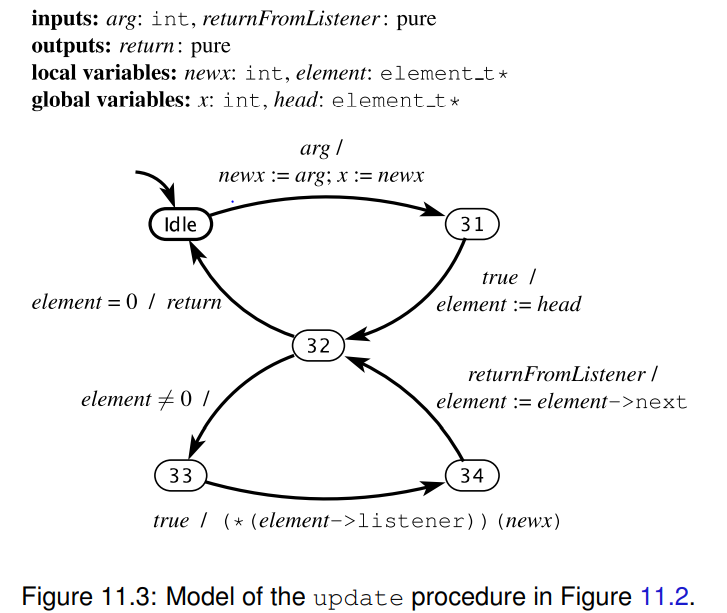
Si discutono i meccanismi di medio livello che vengono utilizzati nel software per fornire esecuzione concorrente di codice sequenziale. L'esecuzione simultanea di più programmi sequenziali multipli in modo concorrente, ma tutti riguardano la tempistica. Una ragione è quella di migliorare la reattività, evitando situazioni in cui i programmi in esecuzione prolungata possono bloccare un programma che risponde a stimoli esterni come i dati di un sensore o una richiesta dell'utente. Una migliore reattività riduce la latenza, ovvero il tempo che intercorre tra il verificarsi di uno stimolo e la risposta. Un'altra ragione è quella di migliorare le prestazioni, consentendo a un programma di essere eseguito simultaneamente su più processori o core. Anche questo è un problema di temporizzazione, in quanto presuppone che sia meglio completare i compiti prima che dopo. Una terza ragione è quella di controllare direttamente la tempistica delle interazioni esterne. Un programma può avere bisogno di eseguire alcune azioni, come ad esempio l'aggiornamento di un display, in momenti particolari, a prescindere da quali altri compiti possano essere eseguiti in quel momento.

Abbiamo già discusso la concorrenza in diversi contesti. La Figura 11.1 mostra la relazione tra l'argomento di questo capitolo e quelli di altri capitoli. I capitoli 8 e 10 coprono il livello più basso della Figura 11.1, Questo capitolo fa da ponte tra questi due livelli. Descrive meccanismi che vengono implementati utilizzando i meccanismi di basso livello e che possono fornire un'infrastruttura per la realizzazione dei meccanismi di alto livello. Collettivamente, queste tecniche di medio livello sono chiamate multitasking, ovvero l'esecuzione simultanea di più compiti. I progettisti di sistemi embedded utilizzano spesso questi meccanismi di medio livello direttamente per creare applicazioni ma è sempre più frequente che i progettisti utilizzino invece i meccanismi di alto livello. Il progettista costruisce un modello utilizzando uno strumento software che supporta un modello di calcolo (o più modelli di calcolo). Il modello viene poi automaticamente o semi-automaticamente in un programma che utilizza i meccanismi di medio o basso livello. Questo processo di traduzione viene variamente chiamato generazione di codice o autocodifica. I meccanismi descritti in questo capitolo sono tipicamente forniti da un sistema operativo, un microkernel o una libreria di procedure. Possono essere piuttosto complicati da implementare correttamente, e quindi l'implementazione dovrebbe essere fatta da esperti (per alcune delle insidie, Boehm (2005)). I programmatori di applicazioni per sistemi embedded si trovano spesso ad implementare tali meccanismi su bare iron (un processore senza sistema operativo). Per farlo correttamente, è necessaria una profonda comprensione dei problemi di concorrenza.



**Imperative Programs**

Un linguaggio di programmazione che esprime una computazione come una sequenza di operazioni viene chiamato linguaggio imperativo. Il C è un linguaggio imperativo. Un programma C specifica una sequenza di passi, dove ogni passo cambia lo stato della memoria della macchina. In C, lo stato della memoria della macchina è rappresentato dai valori delle variabili. Utilizzando le macchine a stati estese, possiamo modellare l'esecuzione di alcuni semplici programmi in C, assumendo che i programmi abbiano un numero fisso e limitato di variabili. Le variabili del programma C saranno le variabili della macchina a stati. Gli stati della macchina a stati rappresentano le posizioni nel programma e le transizioni rappresentano l'esecuzione del programma.



Il modello della Figura 11.3 non è l'unico modello che avremmo potuto costruire della procedura di aggiornamento. Nel costruire tale modello, dobbiamo decidere il livello di dettaglio e dobbiamo decidere quali azioni possono essere trattate in modo sicuro come operazioni atomiche. La Figura 11.3 utilizza le righe di codice come livello di dettaglio, ma non c'è alcuna garanzia che una riga di codice C venga eseguita atomicamente (di solito non è così). Inoltre, spesso i modelli accurati dei programmi C non sono sistemi a stati finiti. Considerando solo il codice in Figura 11.2, un modello a stati finiti non è appropriato perché il codice supporta l'aggiunta di un numero arbitrario di ascoltatori alla lista. Se combiniamo la Figura 11.2 con la procedura principale dell'Esempio 11.1, il sistema è a stati finiti perché solo tre ascoltatori vengono inseriti nella nell'elenco. Un modello a stati finiti accurato, quindi, dovrebbe includere il programma completo rendendo molto difficile il ragionamento modulare sul codice. I problemi peggiorano notevolmente quando si aggiunge la concorrenza. In questo capitolo mostreremo che un ragionamento accurato sui programmi C con meccanismi di concorrenza di medio livello come i thread è sorprendentemente difficile e soggetto a errori. È per questo motivo che i progettisti si orientano verso il livello superiore della Figura 11.1.

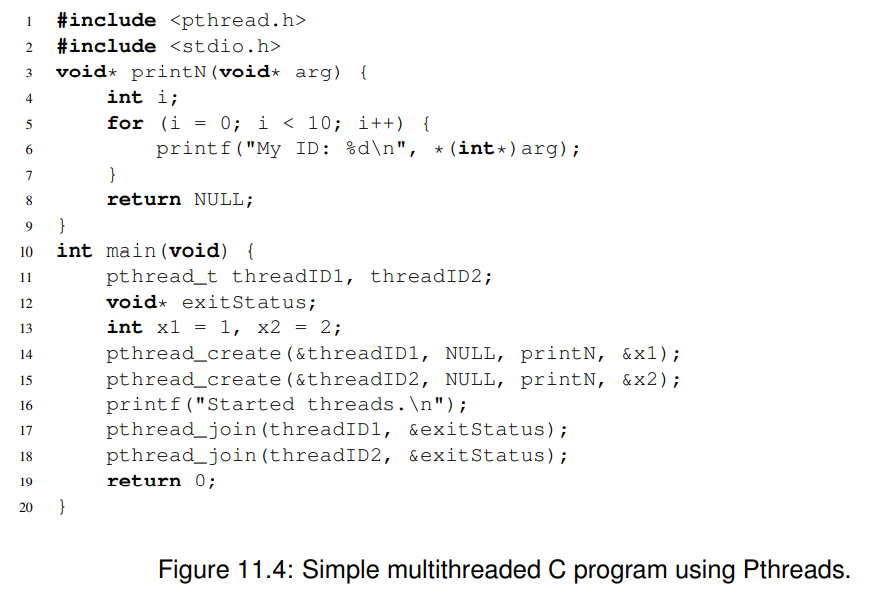
**Threads**

I thread sono programmi imperativi che vengono eseguiti simultaneamente e condividono uno spazio di memoria. Essi possono accedere alle variabili degli altri. Molti operatori del settore utilizzano il termine "thread" in senso più strettamente per riferirsi a particolari modi di costruire programmi che condividono la memoria, ma qui useremo il termine in senso lato per riferirci a qualsiasi meccanismo in cui programmi imperativi vengono eseguiti in modo concorrente e condividono la memoria. In questo senso ampio, i thread esistono sotto forma di interrupt su quasi tutti i microprocessori, anche senza alcun sistema operativo (bare iron).

**Creating Threads**

La maggior parte dei sistemi operativi fornisce un meccanismo di livello superiore agli interrupt per realizzare programmi imperativi che condividono la memoria. Il meccanismo è fornito sotto forma di un insieme di procedure che il programmatore può utilizzare. Tali procedure sono in genere conformi a API (application program interface) standardizzate, che permettono di scrivere programmi che sono portabili (possono essere eseguiti su più processori e/o sistemi operativi). Pthreads (o POSIX threads) è un'API di questo tipo; è integrata in molti sistemi operativi moderni. Pthreads definisce un insieme di tipi, funzioni e costanti del linguaggio di programmazione C. È stato standardizzato dall'IEEE nel 1988 per unificare le varianti di Unix. In Pthreads un thread è definito da una procedura C e creato invocando la procedura **pthread\_create**.

Una routine di avvio(start routine) può ritornare o meno. Nelle applicazioni embedded, è abbastanza comune definire routine di avvio che non non ritornano mai. Per esempio, la routine di avvio potrebbe essere eseguita per sempre e aggiornare periodicamente un display. Se la routine di avvio non ritorna, qualsiasi altro thread che chiama il suo join pthread sarà bloccato indefinitamente. Come mostrato nella Figura 11.4, la routine di avvio può avere un argomento e può restituire un valore. Il quarto argomento di pthread create è l'indirizzo dell'argomento da passare alla routine di avvio. È importante comprendere il modello di memoria del C, spiegato nella Sezione 9.3.5, altrimenti potrebbero verificarsi errori molto sottili, come illustrato nell'esempio successivo.



**Implementing Threads**

Il cuore di un'implementazione di thread è uno **scheduler** che decide quale thread eseguire successivamente quando un processore è disponibile per l'esecuzione di un thread. La decisione può essere basata sull'**equità**, dove il principio è quello di dare a ogni thread attivo un'uguale opportunità di esecuzione, su vincoli di tempo o su qualche misura di importanza o priorità. Gli algoritmi di sono discussi in dettaglio nel Capitolo 12. In questa sezione, ci limitiamo a descrivere come un thread scheduler, senza preoccuparsi di come si decide quale thread eseguire.La prima questione chiave è come e quando viene invocato lo scheduler. Una semplice tecnica multitasking cooperativo non interrompe un thread a meno che non sia il thread stesso a chiamare una certa procedura o una di un certo insieme di procedure. Ad esempio, lo scheduler può intervenire ogni volta che un servizio del sistema operativo viene invocato dal thread in esecuzione. Un servizio del sistema operativo viene invocato tramite una chiamata a una procedura di libreria. Ogni thread ha il proprio stack e, quando viene effettuata la chiamata alla procedura, l'indirizzo di ritorno viene spinto sullo stack. Se lo scheduler stabilisce che il thread attualmente in esecuzione deve continuare l'esecuzione, il servizio richiesto viene completato e la procedura ritorna normalmente. Se invece lo schedulatore determina che il thread deve essere selezionato un altro thread per l'esecuzione, invece di tornare indietro, lo scheduler registra il puntatore allo stack del thread attualmente in esecuzione e modifica lo stack pointer in modo che punti allo stack del thread selezionato. Quindi ritorna come di consueto, togliendo l'indirizzo di ritorno dallo stack e riprendendo l'esecuzione, ma ora in un nuovo thread. Il principale svantaggio del multitasking cooperativo è che un programma può essere eseguito per molto tempo senza effettuare alcuna chiamata di servizio al sistema operativo, nel qual caso gli altri thread saranno affamati(**starved**). Per ovviare a questo inconveniente, la maggior parte dei sistemi operativi include una routine di servizio di interrupt che viene eseguito a intervalli di tempo fissi. Questa routine mantiene un orologio di sistema, che fornisce ai programmatori di applicazioni un modo per ottenere l'ora corrente del giorno e consente di invocare periodicamente lo schedulatore tramite un interrupt del timer. Per un sistema operativo con orologio di sistema un jiffy è l'intervallo di tempo in cui viene invocato l'ISR dell'orologio di sistema.

Il valore di un jiffy è determinato dal bilanciamento delle prestazioni con la precisione di temporizzazione richiesta. Un jiffy più piccolo significa che le funzioni di pianificazione vengono eseguite più spesso, che può peggiorare le prestazioni complessive. Un jiffy più grande significa che la precisione del clock di sistema è più grossolana e la commutazione dei task avviene meno spesso, il che può causare la violazione dei vincoli del tempo reale. A volte, l'intervallo di jiffy è dettato da dall'applicazione. Oltre agli interrupt periodici e alle chiamate ai servizi operativi, lo scheduler può essere invocato quando un thread si blocca per qualche motivo. In seguito discuteremo alcuni dei meccanismi per questo tipo di blocco.

**Mutual Exclusion**

Un thread può essere sospeso tra due operazioni atomiche per eseguire un altro thread e/o una routine di servizio di interrupt. Questo fatto può rendere estremamente difficile ragionare sulle interazioni tra i thread. Il problema illustrato nell'esempio precedente è noto come condizione di gara. Due parti di codice concorrenti per accedere alla stessa risorsa e l'ordine esatto in cui si verifica l'accesso influisce sui risultati del programma. Non tutte le condizioni di gara sono come l'esempio precedente, in cui alcuni risultati della gara causano un fallimento catastrofico. Un modo per prevenire tali disastri è utilizzare un blocco di mutua esclusione (o mutex), come illustrato nel prossimo esempio. Un blocco di mutua esclusione impedisce a due thread di accedere o modificare simultaneamente una risorsa condivisa. Il codice tra il blocco e lo sblocco è una sezione critica. In ogni momento in qualsiasi momento, solo un thread può eseguire codice in tale sezione critica. Un programmatore può avere la necessità di garantire che tutti gli accessi a una risorsa condivisa siano protetti.

**Deadlock**

Quando i mutex locks proliferano nei programmi, aumenteranno il rischio di deadlock. Un deadlock si verifica quando alcuni thread si bloccano in modo permanente nel tentativo di acquisire i locks. Questo può accadere, ad esempio, se il thread A detiene il blocco1 e poi si blocca cercando di acquisire il blocco2, che è detenuto dal thread B, e poi il thread B si blocca cercando di acquisire il blocco1. Questi abbracci mortali non hanno scampo. Il programma deve essere interrotto. Il deadlock può essere difficile da evitare. In un articolo classico, Coffman et al. (1971) forniscono le condizioni necessarie perché si verifichi un deadlock, ognuna delle quali può essere rimossa per evitarlo. Una tecnica semplice è quella di utilizzare un solo blocco in un intero programma multithread. Questa tecnica, tuttavia, non porta a una programmazione molto modulare. Inoltre, può rendere difficile rispettare i vincoli del tempo reale, perché alcune risorse condivise (per esempio, i display) possono essere mantenute abbastanza a lungo da causare il mancato rispetto delle scadenze in altri thread. In un microkernel molto semplice, possiamo talvolta utilizzare l'abilitazione e la disabilitazione degli interrupt come un singolo mutex globale. Supponiamo di avere un singolo processore (non un multicore), e che gli interrupt siano l'unico meccanismo attraverso il quale un thread può essere sospeso (cioè, non vengono sospesi quando si chiamano i servizi del kernel o si blocca l'I/O). Con queste presupposti, la disabilitazione degli interrupt impedisce la sospensione di un thread. Nella maggior parte dei sistemi operativi, tuttavia, i thread possono essere sospesi per molte ragioni, quindi questa tecnica non funziona. Una terza tecnica consiste nell'assicurarsi che, in presenza di più lock mutex, ogni thread acquisisca i lock nello stesso ordine. Questo può essere difficile da garantire, tuttavia, per diverse ragioni (si veda l'Esercizio 2). In primo luogo, la maggior parte dei programmi sono scritti da più persone e i lock acquisiti all'interno di una procedura non fanno parte della firma della procedura stessa. Quindi questa tecnica si basa su una documentazione molto attenta e coerente e sulla cooperazione tra un team di sviluppo. Inoltre, ogni volta che viene aggiunto un blocco, è necessario modificare tutte le parti del programma che acquisiscono i blocchi. possono dover essere modificate. In secondo luogo, può rendere estremamente difficile una codifica corretta. Se un programmatore desidera chiamare una procedura che acquisisce il blocco1, che per convenzione nel programma è sempre il primo acquisito, deve prima rilasciare tutti i lock che detiene. Non appena rilascia blocchi, può essere sospesa e la risorsa che ha protetto può essere modificata. Una volta acquisito il blocco1, deve riacquisire i blocchi, ma deve presumere di non sapere più nulla sullo stato delle risorse e potrebbe dover rifare un lavoro considerevole. Esistono molti altri modi per prevenire i deadlock. Ad esempio, una tecnica particolarmente elegante sintetizza i vincoli su uno scheduler per prevenire i deadlock (Wang et al., 2009). Tuttavia, la maggior parte delle tecniche disponibili o impongono vincoli severi al programmatore o richiedono una notevole sofisticazione per essere applicate, il che suggerisce che il problema possa risiedere nel modello di programmazione concorrente dei thread.

**Memory Consistency Models**

Come se le condizioni di gara e di deadlock non fossero già abbastanza problematiche, i thread soffrono anche problemi potenzialmente sottili con il modello di memoria dei programmi. Ogni particolare implementazione di thread offre una sorta di modello di consistenza della memoria, che definisce come le variabili lette e scritte da thread diversi appaiono a questi ultimi. Intuitivamente, la lettura di una variabile dovrebbe restituire l'ultimo valore scritto sulla variabile, ma cosa significa "ultimo"? Si consideri uno scenario, ad esempio, in cui tutte le variabili sono inizializzate con il valore zero e il thread A esegue le seguenti due istruzioni:

1 x = 1;

2 w = y;

mentre il thread B esegue le due istruzioni seguenti:

1 y = 1;

2 z = x;

Intuitivamente, dopo che entrambi i thread hanno eseguito queste istruzioni, ci aspettiamo che almeno una delle due variabili w e z abbia valore 1. Tale garanzia viene definita consistenza sequenziale (Lamport, 1979). La coerenza sequenziale significa che il risultato di qualsiasi esecuzione è lo stesso che si avrebbe se le operazioni di tutti i thread fossero eseguite in un certo ordine sequenziale e le operazioni di ogni singolo thread apparissero in questa sequenza nell'ordine specificato dal thread. Tuttavia, la coerenza sequenziale non è garantita dalla maggior parte (o forse da tutte) le implementazioni di Pthread. In effetti, fornire una tale garanzia è piuttosto difficile sui moderni processori che utilizzano compilatori moderni. Un compilatore, ad esempio, è libero di riordinare le istruzioni in ciascuno di questi thread perché non c'è alcuna dipendenza tra di essi (visibile al compilatore). Anche se il compilatore non le riordina, l'hardware potrebbe farlo. Una buona tattica difensiva consiste nel proteggere con molta attenzione gli accessi alle variabili condivise usando i blocchi di mutua esclusione (e sperare che i blocchi di mutua esclusione siano implementati correttamente). Un'autorevole panoramica dei problemi di coerenza della memoria è fornita da Adve e Gharachorloo (1996), che si concentrano sui multiprocessori. Boehm (2005) fornisce un'analisi dei problemi di consistenza della memoria con i thread su un singolo processore.

**The Problem with Threads**

I programmi multithread possono essere molto difficili da capire. Inoltre, può essere difficile avere fiducia nei programmi, perché i problemi nel codice possono non essere evidenziati nei test. Un programma può avere la possibilità di un deadlock, ad esempio, ma nonostante ciò funzionare correttamente per anni senza che il deadlock si manifesti mai. I programmatori devono essere molto cauti, ma ragionare sui programmi è sufficientemente difficile da far sì che gli errori di programmazione non avvenghino.

Tutti i programmatori hanno familiarità con i thread e apprezzano la facilità con cui sfruttano l'hardware parallelo sottostante. È possibile, ma non facile, costruire programmi multithread affidabili e corretti. Si veda ad esempio Lea (1997) per un'eccellente guida all'uso dei thread in Java. Entro il 2005, le librerie standard di Java includevano strutture di dati e meccanismi basati sui thread (Lea, 2005). Librerie come OpenMP (Chapman et al., 2007) forniscono anche il supporto per i modelli multithread comunemente usati, come i costrutti di loop paralleli. Tuttavia, i programmatori di sistemi embedded raramente utilizzano Java o pacchetti di grandi pacchetti sofisticati come OpenMP. E anche se lo facessero, si verificherebbero gli stessi rischi di deadlock e di errori insidiosi. I thread presentano una serie di difficoltà che rendono discutibile la loro esposizione ai programmatori come metodo per costruire programmi concorrenti (Ousterhout, 1996; Sutter e Larus, 2005; Lee, 2006; Hayes, 2007). In effetti, prima degli anni '90, i thread non erano affatto utilizzati dai programmatori di applicazioni. È stato l'emergere di librerie come Pthreads e di linguaggi come Java e C# che hanno esposto questi meccanismi ai programmatori di applicazioni. I programmi multithread non banali sono sorprendentemente difficili da comprendere e possono produrre errori insidiosi, condizioni di gara e deadlock. I problemi possono rimanere in agguato nei programmi multithread anche dopo anni di utilizzo intensivo. Questi problemi sono particolarmente importanti per i sistemi embedded che riguardano la sicurezza e la vita delle persone. Poiché praticamente tutti i sistemi embedded coinvolgono software concorrente, gli ingegneri che progettano sistemi embedded devono affrontarne le insidie.

**Processes and Message Passing**

I processi sono programmi imperativi con spazi di memoria propri. Questi programmi non possono fare riferimento alle variabili degli altri e di conseguenza non presentano le stesse difficoltà dei thread. La comunicazione tra i programmi deve avvenire tramite meccanismi forniti dal sistema operativo, dal microkernel o da una libreria. L'implementazione corretta dei processi richiede generalmente un supporto hardware sotto forma di unità di gestione della memoria o MMU. L'MMU protegge la memoria di un processo da letture o scritture accidentali da parte di un altro processo. In genere fornisce anche la traduzione degli indirizzi, dando a ogni processo l'illusione di uno spazio di indirizzi di memoria fisso, uguale per tutti i processi. Quando un processo accede a una posizione di memoria in quello spazio di indirizzi la MMU sposta l'indirizzo in modo che si riferisca a una posizione nella porzione di memoria fisica assegnata a quel processo. Per ottenere la concorrenza, i processi devono essere in grado di comunicare. I sistemi operativi forniscono una serie di meccanismi, che spesso includono anche la possibilità di creare spazi di memoria condivisi, il che ovviamente apre la strada a tutte le potenziali difficoltà della programmazione multithread. Un meccanismo che presenta meno difficoltà è il file system. Un file system è semplicemente un modo per creare un corpo di dati che sia persistente, nel senso che sopravvive al processo che lo crea. Un processo può creare dati e scriverli su un file, e un altro processo può leggere i dati dallo stesso file. È compito dell'implementazione del file system garantire che il processo che legge i dati non li legga prima che siano stati scritti. Questo può essere fatto, ad esempio consentendo a non più di un processo alla volta di operare su un file. Un meccanismo più flessibile per la comunicazione tra processi è il passaggio di messaggi. In questo caso, un processo crea un pezzo di dati, lo deposita in una sezione di memoria accuratamente controllata e condivisa e poi notifica agli altri processi che il messaggio è pronto. Questi altri processi possono bloccarsi in attesa che i dati siano pronti. Il passaggio di messaggi richiede una memoria condivisa, ma è implementato in librerie presumibilmente scritte da esperti. Il programmatore di un'applicazione invoca una procedura di libreria per inviare un messaggio o riceverlo.

La variabile di condizione utilizzata nell'esempio precedente è una forma generalizzata di semaforo. I semafori prendono il nome dai segnali meccanici tradizionalmente utilizzati sui binari ferroviari per segnalare che un tratto di binario è occupato da un treno. Utilizzando tali semafori, è possibile usare un'unica sezione di binario per far viaggiare i treni in entrambe le direzioni (il semaforo attua la mutua esclusione, impedendo a due treni di trovarsi contemporaneamente sulla stessa sezione di binario). Negli anni '60, Edsger W. Dijkstra, professore presso il Dipartimento di Matematica della Eindhoven University of Technology, nei Paesi Bassi, prese in prestito questa idea per dimostrare come i programmi potessero condividere le risorse in modo sicuro. Un semaforo di conteggio (che Dijkstra ha chiamato semaforo PV) è una variabile il cui valore è un intero non negativo. Un valore pari a zero viene trattato come distintamente diverso da un valore maggiore di zero. In effetti, la variabile size dell'Esempio 11.13 funziona come un semaforo di questo tipo. Viene incrementata con l'invio di un messaggio e un valore pari a zero blocca l'utente finché il valore non è diverso da zero. Le variabili di condizione generalizzano questa idea supportando condizioni arbitrarie, piuttosto che solo zero o non zero, Inoltre, almeno in Pthreads, le variabili di condizione si coordinano anche con i mutex per rendere più facile la scrittura di schemi come quello dell'Esempio 11.13. Dijkstra ha ricevuto il premio Turing 1972 per il suo lavoro sulla programmazione concorrente. L'uso del passaggio di messaggi nelle applicazioni può essere più semplice dell'uso diretto di thread e variabili condivise. Ma anche il passaggio di messaggi non è privo di rischi. L'implementazione dello schema produttore/consumatore nell'Esempio 11.13, infatti, presenta un difetto piuttosto grave. In particolare non impone alcun vincolo sulla dimensione della coda di messaggi. Ogni volta che un thread produttore chiama send, viene allocata della memoria per memorizzare il messaggio e questa memoria non viene deallocata finché il messaggio non viene consumato. Se il thread produttore produce messaggi più velocemente di quanto il consumatore li consumi, il programma finirà per esaurire la memoria disponibile. Questo problema può essere risolto limitando la dimensione del buffer (si veda l'Esercizio 4), ma quale dimensione è appropriata? La scelta di buffer troppo piccoli può causare un deadlock del programma, e la scelta di buffer troppo grandi comporta uno spreco di risorse. Questo problema non è banale da risolvere (Lee, 2009b). Ci sono anche altre insidie. I programmatori possono inavvertitamente costruire programmi di messagepassing che si bloccano, in cui un insieme di thread sono tutti in attesa di messaggi l'uno dall'altro. Inoltre, i programmatori possono inavvertitamente costruire programmi di message-passing che sono nondeterminati, nel senso che i risultati della computazione dipendono dall'ordine (arbitrario) in cui i thread sono stati calcolati. La soluzione più semplice è che i programmatori di applicazioni utilizzino livelli di astrazione più alti per la concorrenza, il livello superiore della Figura 11.1, come descritto nel Capitolo 6. Naturalmente, possono usare questa strategia solo se hanno a disposizione un'implementazione affidabile di un modello di calcolo concorrente di livello superiore.

**Scheduling**

Il Capitolo 11 ha spiegato il multitasking, in cui più task imperativi vengono eseguiti contemporaneamente, sia in interleaved su un singolo processore che in parallelo su più processori. Quando ci sono meno processori che compiti (il caso normale), o quando i compiti devono essere eseguiti in un momento particolare, deve intervenire uno schedulatore. Uno scheduler prende la decisione su cosa fare in un determinato momento, come ad esempio quando un processore diventa disponibile. I sistemi in tempo reale sono collezioni di attività in cui, oltre a qualsiasi vincolo di ordine imposti dalle precedenze tra i task, ci sono anche vincoli temporali. Questi vincoli mettono in relazione l'esecuzione di un compito con il tempo reale, che è il tempo fisico nell'ambiente del computer che esegue il compito. In genere, i compiti hanno delle scadenze, che sono valori di tempo fisico entro i quali il compito deve essere completato. Più in generale, i programmi in tempo reale possono avere tutti i tipi di vincoli temporali, non solo le scadenze. Ad esempio, un task può essere richiesto di essere eseguito non prima di un determinato momento; oppure può essere richiesto di essere eseguito non più di un determinato lasso di tempo dopo l'esecuzione di un altro task; oppure può essere richiesto di essere eseguito periodicamente con un certo periodo di tempo. I task possono essere dipendenti l'uno dall'altro e possono formare un'applicazione in modo cooperativo. Oppure possono essere non correlati, tranne che per il fatto che condividono le risorse del processore. Tutte queste situazioni richiedono una strategia di schedulazione.

**Basics of Scheduling**

In questa sezione si discute la gamma di possibilità di schedulazione, le proprietà dei task che uno scheduler utilizza per guidare il processo e l'implementazione di scheduler in un sistema operativo o in un microkernel.

**Scheduling Decisions**

Uno scheduler decide quale compito eseguire successivamente quando si trova di fronte a una scelta nell'esecuzione di un programma o di un insieme di programmi concorrenti. In generale, uno scheduler può avere a disposizione più di un processore (ad esempio in un sistema multicore). Uno scheduler multiprocessore deve decidere non solo quale compito eseguire successivamente, ma anche su quale processore eseguirlo. La scelta del processore è chiamata assegnazione del processore. Una decisione di schedulazione è una decisione di esecuzione di un'attività e si compone delle seguenti tre parti:

* **assegnazione**: quale processore deve eseguire il compito;
* **ordinamento**: l'ordine in cui ogni processore deve eseguire i propri compiti; e
* **tempistica**: l'ora in cui ogni compito viene eseguito.

Ognuna di queste tre decisioni può essere presa in fase di progettazione, prima dell'inizio dell'esecuzione del programma, oppure in fase di esecuzione, durante l'esecuzione del programma. A seconda del momento in cui vengono prese le decisioni, si possono distinguere alcuni tipi diversi di scheduler (Lee e Ha, 1989). Uno **schedulatore completamente statico** prende tutte e tre le decisioni in momento della progettazione. Il risultato dello scheduling è una precisa specificazione per ogni processore di che cosa fare quando. Uno scheduler completamente statico di solito non ha bisogno di semafori o blocchi. Può invece utilizzare la temporizzazione per imporre vincoli di mutua esclusione e di precedenza. Tuttavia, gli scheduler completamente statici sono difficili da realizzare con la maggior parte dei microprocessori moderni perché il tempo necessario per eseguire un'attività è difficile da prevedere con precisione e perché le attività tipicamente hanno tempi di esecuzione dipendenti dai dati (si veda il Capitolo 16).

Uno **static order scheduler** esegue l'assegnazione e l'ordinamento dei task in fase di progettazione, ma rinvia a tempo di esecuzione la decisione su quando eseguire un'attività nel tempo fisico. Questa decisione può essere influenzata, ad esempio, dalla possibilità di acquisire un blocco di mutua esclusione, o se i vincoli di precedenza sono stati soddisfatti. Nella schedulazione statica, ogni processore riceve i suoi ordini di marcia prima dell'inizio dell'esecuzione del programma e si limita a eseguirli il più rapidamente possibile. Non cambia, ad esempio, l'ordine dei compiti in base allo stato di un semaforo o di un blocco. Un task stesso, tuttavia, può bloccarsi su un semaforo o un blocco nel qual caso esso blocca l'intera sequenza di compiti su quel processore. Uno scheduler di ordine statico è spesso chiamato **scheduler off-line.**

Uno **static assignment scheduler** esegue l'assegnazione in fase di progettazione e tutto il resto in fase di esecuzione. A ogni processore viene assegnato un insieme di compiti da eseguire e uno scheduler a tempo di esecuzione decide durante l'esecuzione quale compito eseguire successivamente.

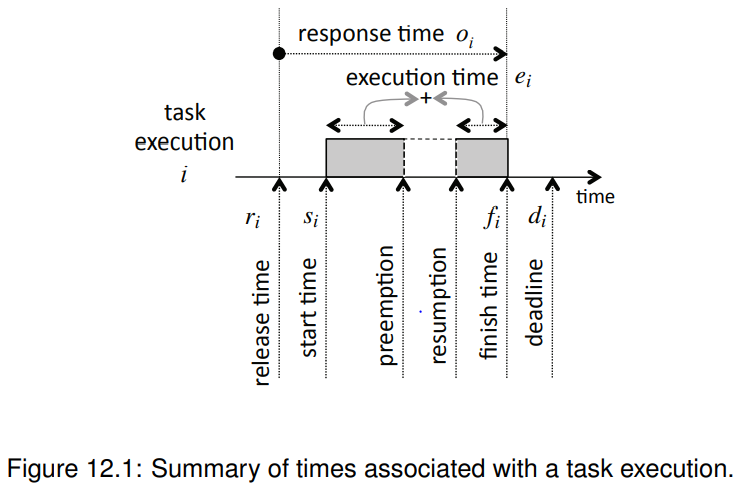
Uno **scheduler completamente dinamico** prende tutte le decisioni a tempo di esecuzione. Quando un processore diventa disponibile (ad esempio, termina l'esecuzione di un task o un task si blocca acquisendo un mutex), lo scheduler decide a quel punto quale compito eseguire successivamente su quel processore. Sia gli scheduler ad assegnazione statica che quelli completamente dinamici sono spesso chiamati **scheduler on-line**. Esistono naturalmente altre possibilità di scheduler. Per esempio, l'assegnazione di un compito può essere effettuata una sola volta per un compito, a tempo di esecuzione, appena prima della prima esecuzione del compito. Per le esecuzioni successive dello stesso task, viene utilizzata la stessa assegnazione. Alcune combinazioni non hanno molto senso. Ad esempio, non ha senso determinare il tempo di esecuzione di un'attività di un task in fase di progettazione e l'ordine in fase di esecuzione.

Uno **scheduler preemptive** può prendere una decisione di schedulazione durante l'esecuzione di un task, assegnando un nuovo task allo stesso processore. In altre parole, un'attività può essere nel bel mezzo dell'esecuzione quando lo scheduler decide di interromperla per iniziare l'esecuzione di un'altra attività. L'interruzione del primo task è chiamata **prelazione**. Uno scheduler che lascia sempre che i compiti vengano eseguiti fino al completamento prima di assegnare un altro compito da eseguire sullo stesso processore è chiamato **scheduler non preemptive**.

Nello scheduling preemptive, un'attività può essere preemptata se tenta di acquisire un blocco di mutua esclusione e il blocco non è disponibile. In questo caso, si dice che l'attività è bloccata sul blocco. Quando un altro task rilascia il blocco, il task bloccato può riprendere la sua attività. Inoltre, un'attività può essere preemptata quando rilascia un blocco. Questo può accadere, ad esempio se c'è un task a priorità più alta che è bloccato sul blocco. In questo capitolo assumeremo programmi ben strutturati, in cui ogni task che acquisisce un blocco alla fine lo rilascia.

**Task Models**

Per prendere le sue decisioni, lo scheduler ha bisogno di alcune informazioni sulla struttura del programma. Un'ipotesi tipica è che allo schedulatore venga dato un insieme finito T di compiti. Ogni compito può essere assunto come finito (termina in un tempo finito), oppure no. Un tipico scheduler di un sistema operativo non assume che i compiti terminino, ma gli schedulatori in tempo reale spesso lo fanno. Uno scheduler può fare molte altre assunzioni sui task, alcune delle quali vengono discusse in questa sezione. L'insieme delle ipotesi è chiamato modello di task dello scheduler. Alcuni scheduler assumono che tutti i task da eseguire siano noti prima dell'inizio della schedulazione, e alcuni supportano l'arrivo dei task, ovvero i task diventano noti allo scheduler mentre altri task vengono eseguiti. Alcuni schedulatori supportano scenari in cui ogni task τ ∈ T viene eseguito ripetutamente, possibilmente per sempre o periodicamente. Un task può anche essere sporadico, il che significa che si ripete e che la sua tempistica è irregolare, ma che c'è un limite inferiore al tempo che intercorre tra le esecuzioni dei task. Nelle situazioni in cui un'attività τ ∈ T viene eseguita ripetutamente, è necessario fare una distinzione tra il compito τ e le esecuzioni dei compiti τ1, τ2, - - - . Se ogni compito viene eseguito esattamente una volta, non è necessaria questa distinzione. Le esecuzioni dei task possono avere dei vincoli di precedenza, ovvero un requisito per cui un'esecuzione precede un'altra. Se l'esecuzione i deve precedere la j, si può scrivere i < j. In questo caso, i e j possono essere esecuzioni distinte dello stesso task. L'esecuzione di un task i può avere alcune precondizioni per iniziare o riprendere l'esecuzione. Queste sono condizioni che devono essere soddisfatte prima che il task possa essere eseguito. Quando le precondizioni sono soddisfatte, si dice che l'esecuzione del task è abilitata. Le precedenze, ad esempio, specificano precondizioni per avviare l'esecuzione di un'attività. La disponibilità di un blocco può essere una precondizione per la ripresa di un'attività.



Definiamo quindi alcuni termini, riassunti nella Figura 12.1. Per l'esecuzione di un task ***i***, definiamo il tempo di rilascio ***ri*** (detto anche tempo di arrivo) come il primo momento in cui un task viene abilitato. Definiamo il tempo di inizio ***si*** come il momento in cui l'esecuzione ha effettivamente inizio. Ovviamente, richiediamo che:

***si ≥ ri***

Definiamo il tempo di completamento ***fi*** il momento in cui l'attività completa l'esecuzione. Quindi,

***fi ≥ si***

Il tempo di risposta ***oi***è dato da:

***oi = fi − ri***

Il tempo di risposta, quindi, è il tempo che intercorre tra l'attivazione del task e il suo completamento.

Il tempo di esecuzione ***ei***di ***τi*** è definito come il tempo totale di esecuzione del task. Non include il tempo in cui l'attività può essere bloccata o preemplificata. Molte strategie di di schedulazione assumono (spesso in modo irrealistico) che il tempo di esecuzione di un task sia noto e fisso. Se il tempo di esecuzione è variabile, è comune assumere (spesso in modo irrealistico) che il tempo di esecuzione nel caso peggiore (WCET) sia noto. Determinare i tempi di esecuzione del software può essere piuttosto impegnativa, come discusso nel Capitolo 16.

La deadline ***di***è il tempo entro il quale un compito deve essere completato. A volte, la scadenza è un vero e proprio vincolo fisico imposto dall'applicazione, dove il mancato rispetto della scadenza è considerato un errore. Una scadenza di questo tipo è detta **hard deadline**. La programmazione con scadenze rigide è chiamata schedulazione in tempo reale rigida. Spesso, una scadenza riflette una decisione progettuale che non deve essere applicata rigorosamente. È meglio rispettare la scadenza, ma il mancato rispetto della scadenza non è un errore. In genere è meglio non mancare la scadenza di molto. Questo caso è chiamato **programmazione soft in tempo reale**. Uno schedulatore può usare la priorità piuttosto che (o in aggiunta a) una scadenza. Uno scheduler basato sulla priorità prevede che a ogni task venga assegnato un numero, chiamato priorità, e che lo scheduler scelga sempre di eseguire il task con maggiore priorità. Una priorità fissa è una priorità che rimane costante per tutte le esecuzioni di un task. Una priorità dinamica può cambiare durante l'esecuzione. Uno **scheduler basato su priorità preemptive** è uno scheduler che supporta l'arrivo dei task e che in ogni momento esegue il task abilitato con la priorità più alta. Uno scheduler non preemptive è uno scheduler che utilizza le priorità per determinare quale task eseguire dopo l'esecuzione del task corrente, ma non interrompe mai un'attività durante l'esecuzione per programmare un'altra attività.

**Comparing Schedulers**

La scelta della strategia di schedulazione è regolata da considerazioni che dipendono dagli obiettivi dell'applicazione. Un obiettivo piuttosto semplice è che tutte le esecuzioni dei task rispettino le loro scadenze, ***fi ≤ di***. Una schedulazione che raggiunge questo obiettivo è chiamata **feasible schedule**. Uno scheduler che produce una schedulazione fattibile per qualsiasi insieme di task (conforme al suo modello di task) per il quale esiste una schedulazione fattibile, si dice che è ottimale rispetto alla fattibilità. Un criterio che può essere utilizzato per confrontare gli algoritmi di schedulazione è l'utilizzo del processore ottenibile. L**'utilizzo** è la percentuale di tempo che il processore dedica all'esecuzione dei compiti (rispetto all'inattività). Questa metrica è più utile per le attività che vengono eseguite periodicamente. Un algoritmo di schedulazione che fornisce una schedulazione fattibile ogni volta che l'utilizzo del processore è inferiore o uguale al 100% è ovviamente ottimale per quanto riguarda la fattibilità. Non riesce a fornire un programma fattibile solo in circostanze in cui tutti gli algoritmi di scheduling non riescono a fornire un programma fattibile. Un altro criterio che può essere utilizzato per confrontare gli schedulatori è la massima latenza, definito per un insieme di esecuzioni di task T come:

***Lmax = max i∈T (fi − di)***

Per un programma fattibile, questo numero è zero o negativo. Ma la massima latenza può anche essere utilizzata per confrontare i programmi non fattibili. Per i problemi soft in tempo reale, può essere tollerabile che questo numero sia positivo, purché non diventi troppo grande.

Un terzo criterio che può essere utilizzato per un insieme finito T di esecuzioni di task è il tempo totale di completamento o **makespan**, definito da:

***M = max i∈T fi − min i∈T ri***

Se l'obiettivo della schedulazione è quello di minimizzare il makespan, si tratta più di un obiettivo di performance che di un requisito in tempo reale.

**Implementation of a Scheduler**

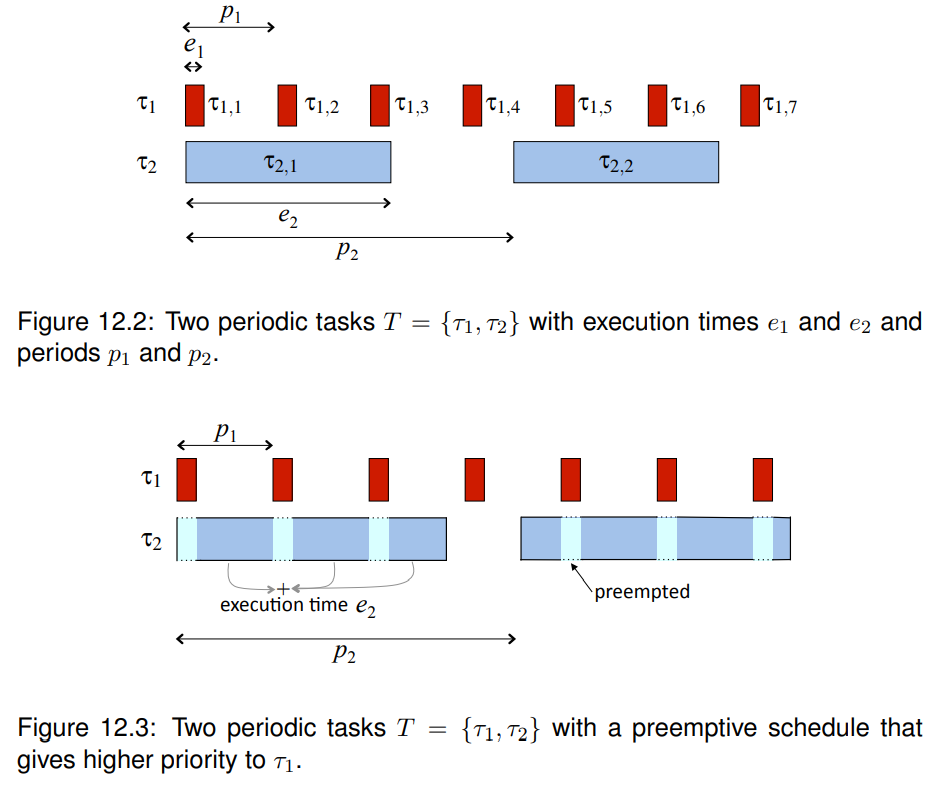
Uno scheduler può essere parte di un compilatore o di un generatore di codice (per le decisioni di scheduling prese in fase di progettazione), parte di un sistema operativo o di un microkernel (per le decisioni di scheduling prese a tempo di esecuzione), o di entrambi (se alcune decisioni di scheduling sono prese a tempo di progettazione e altre a tempo di esecuzione). Uno schedulatore a tempo di esecuzione implementa tipicamente i task come thread (o come processi, ma la distinzione non è importante). A volte lo schedulatore presuppone che questi thread si completino in un tempo finito, altre volte non fa alcuna ipotesi del genere. In entrambi i casi, lo scheduler è una procedura che viene invocata in determinati momenti. Per gli scheduler molto semplici, e non preemptive, la procedura di scheduling può essere invocata ogni volta che un task viene completato. Per gli scheduler preemptive, la procedura di scheduling viene invocata quando si verifica una delle seguenti situazioni:

* Si verifica un'interruzione del timer, ad esempio a un jiffy interval.
* Si verifica un'interruzione di I/O.
* Viene invocato un servizio del sistema operativo.
* Un task tenta di acquisire un mutex.
* Un task testa un semaforo.

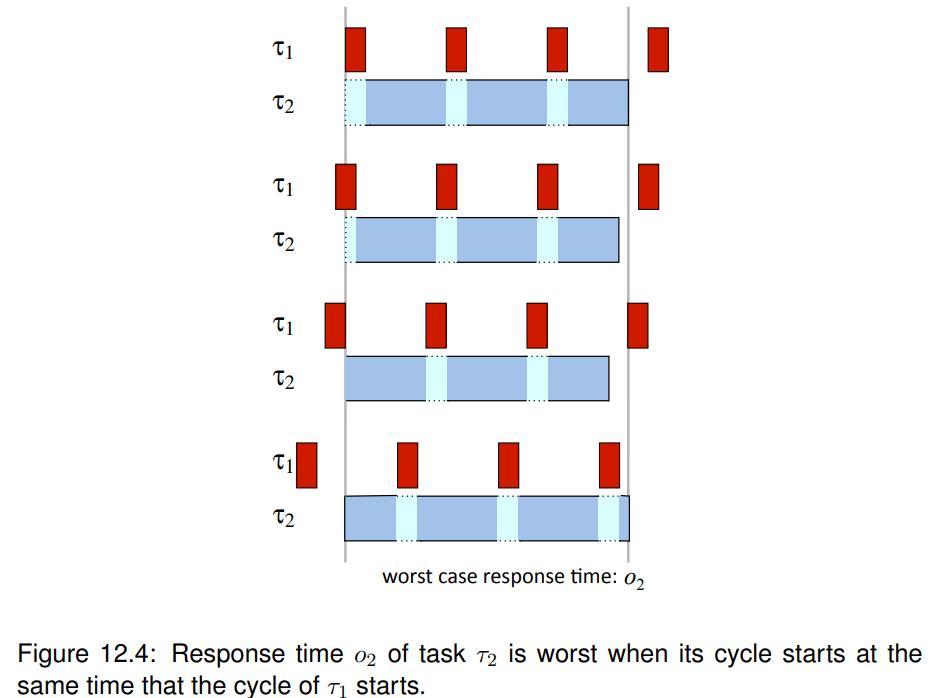
Per gli interrupt, la procedura di schedulazione viene chiamata dalla routine di servizio degli interrupt (ISR). Negli altri casi, la procedura di schedulazione viene chiamata dalla procedura del sistema operativo che fornisce il servizio. In entrambi i casi, lo stack contiene le informazioni necessarie per riprendere l'esecuzione. Tuttavia, lo scheduler può scegliere di non riprendere semplicemente l'esecuzione. Può cioè scegliere di non tornare immediatamente dalla procedura di interrupt o di servizio. Può scegliere di escludere l'attività in corso e iniziare o riprendere un'altra attività. Per realizzare questa prelazione, lo schedulatore deve registrare il fatto che l'attività viene pretermesso (e, forse, il motivo per cui è stato pretermesso), in modo da poterlo riprendere in un secondo momento. Può quindi regolare il puntatore allo stack per fare riferimento allo stato del task da avviare o riprendere. A quel punto, si esegue un ritorno, ma invece di riprendere l'esecuzione con il task che era stato preemptive, l'esecuzione riprenderà per un altro task. L'implementazione di uno scheduler preemptive può essere piuttosto impegnativa. Richiede un controllo molto attento della concorrenza. Ad esempio, potrebbe essere necessario disabilitare gli interrupt per parti significative del processo per evitare di ritrovarsi con uno stack corrotto. Questo è il motivo per cui la schedulazione è una delle funzioni più centrali di un kernel o microkernel del sistema operativo. La qualità dell'implementazione influisce fortemente sull'affidabilità e sulla stabilità del sistema.

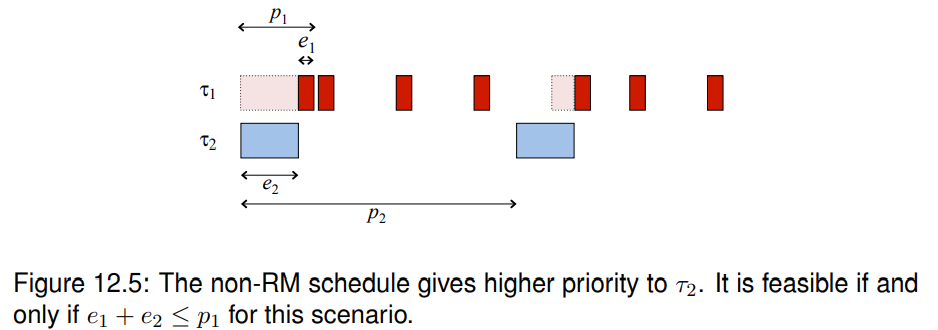
**Rate Monotonic Scheduling**

Si consideri uno scenario con T = {τ1, τ2, - - - , τn} di n task, dove i task devono essere eseguiti periodicamente. In particolare, si assume che ogni task τi debba essere eseguito esattamente una volta in ogni intervallo di tempo pi. Ci riferiamo a pi come al periodo del task. Quindi, la scadenzaper la j-esima esecuzione di τi è ***ri,1+jpi***, dove ri,1 è il tempo di rilascio della prima esecuzione. Liu e Layland (1973) hanno dimostrato che una semplice strategia di schedulazione preemptive chiamata **rate monotonico** (RM) è **ottimale rispetto alla fattibilità tra gli schedulatori uniprocessori a priorità fissa** per il modello di task sopra descritto. Questa strategia di schedulazione dà maggiore priorità a un compito con un periodo più piccolo. La forma più semplice del problema prevede solo due task, T = {τ1, τ2} con tempi di esecuzione e1 e e2 e periodi p1 e p2, come mostrato nella Figura 12.2. Nella figura, il tempo di esecuzione e2 del task τ2 è più lungo del periodo p1 del task τ1. Pertanto, se questi due task devono essere eseguiti sullo stesso processore, è chiaro che uno scheduler non preemptive non produrrà un programma fattibile. Se l'attività τ2 deve essere eseguita fino al completamento senza interruzioni, allora il task τ1 non rispetterà alcune scadenze.



La Figura 12.3 mostra un programma di prelazione che segue il principio del tasso monotono. In questa figura, il task τ1 ha una priorità maggiore, perché il suo periodo è più piccolo. Quindi viene eseguito all'inizio di ogni intervallo di tempo, indipendentemente dal fatto che τ2 sia in esecuzione. Se τ2 è in esecuzione allora τ1 lo precede. La figura assume che il tempo necessario per eseguire la prelazione, chiamato tempo di commutazione del contesto, sia trascurabile. L'ipotesi che il tempo di commutazione del contesto sia trascurabile è problematica nella pratica. Sui processori con cache, un cambio di contesto spesso causa ritardi sostanziali legati alla cache. Questo programma è fattibile, mentre se a τ2 fosse stata assegnata una priorità maggiore, il programma non sarebbe stato fattibile. Per il caso di due task, è facile dimostrare che tra tutti gli scheduler preemptive a priorità fissa, RM è ottimale rispetto alla fattibilità, sotto il modello di task ipotizzato con tempo di commutazione di contesto trascurabile. Questo è facile da dimostrare perché ci sono solo due schedulatori a priorità fissa per questo semplice caso: lo scheduler RM, che dà la priorità più alta al task τ1, e la schedulazione non-RM, che dà maggiore priorità al task τ2. Per dimostrare l'ottimalità, è sufficiente dimostrare che se la schedulazione non-RM è fattibile, lo è anche la schedulazione RM.



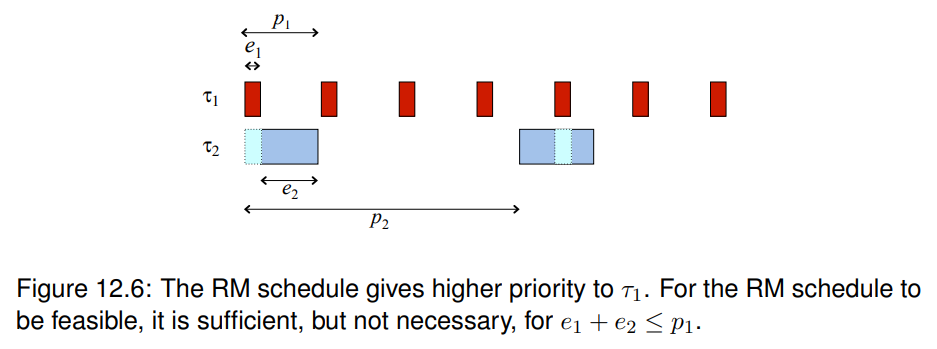


Prima di poterlo fare, è necessario considerare i possibili allineamenti delle esecuzioni dei compiti che

possono influenzare la fattibilità. Come illustrato nella Figura 12.4, il tempo di risposta del task a priorità più bassa è peggiore quando la sua fase iniziale coincide con quella dei task a priorità più alta. In altre parole, lo scenario peggiore si verifica quando tutti i task iniziano i loro cicli nello stesso momento. Pertanto, dobbiamo considerare solo questo scenario. In questo scenario peggiore, in cui i tempi di rilascio si allineano, il programma non-RM è fattibile se e solo se:

***e1 + e2 ≤ p1***

Questo scenario è illustrato nella Figura 12.5. Poiché il task τ1 è preempted da τ2, affinché τ1 non manchi la sua scadenza, si richiede che e2 ≤ p1 - e1, in modo che τ2 lasci tempo sufficiente per τ1 per eseguire prima della sua scadenza.



Per dimostrare che la RM è ottimale rispetto alla fattibilità, è sufficiente dimostrare che se il programma non RM è fattibile, anche il programma RM è fattibile. Esaminando la Figura 12.6, è chiaro che se l'equazione (12.1) è soddisfatta, allora il programma RM è fattibile. Poiché questi sono gli unici due programmi a priorità fissa, il programma RM è ottimale per quanto riguarda la fattibilità. La stessa tecnica di dimostrazione può essere generalizzata a un numero arbitrario di task, ottenendo il seguente teorema (Liu e Layland, 1973):

***“Dato uno scheduler preemptive a priorità fissa e un insieme finito di task ripetitivi T = {τ1, τ2, - - - , τn} con periodi associati p1, p2, - - - , pn e nessun vincolo di precedenza, se una qualsiasi assegnazione di priorità produce uno scheduler fattibile, allora l'assegnazione di priorità monotona del tasso produce un programma fattibile.”***

Le pianificazioni RM sono facilmente implementabili con un timer ad interrupt con un intervallo di tempo pari al massimo comune divisore dei periodi dei task. Possono anche essere implementati con più interrupt del timer. È emerso che gli scheduler RM non possono sempre raggiungere il 100% di utilizzo. In particolare Gli schedulatori RM sono vincolati ad avere una priorità fissa. Questo vincolo si traduce in situazioni in cui un insieme di task che produce una schedulazione fattibile ha un utilizzo inferiore al 100%, ma non può tollerare un aumento dei tempi di esecuzione o diminuzione dei periodi. Ciò significa che ci sono cicli di processore inattivi che non possono essere utilizzati senza causare il mancato rispetto delle scadenze. Fortunatamente, Liu e Layland (1973) dimostrano che questo effetto è limitato. Innanzitutto si noti che l'utilizzo di n compiti indipendenti con tempi di esecuzione ei e periodi pi si può scrivere:

Immagine che contiene orologio

Descrizione generata automaticamente

Se µ = 1, il processore è occupato per il 100% del tempo. Quindi è chiaro che se µ > 1 per un qualsiasi insieme di task allora quell'insieme di compiti non ha un programma fattibile. Liu e Layland (1973) dimostrano che se µ è inferiore o uguale a un limite di utilizzo dato da:



allora il programma RM è fattibile.

Per comprendere questo risultato (piuttosto notevole), consideriamo alcuni casi. Innanzitutto, se n = 1 (c'è

c'è un solo task), allora n(21/n - 1) = 1, quindi il risultato ci dice che se l'utilizzo è pari o inferiore al 100%, allora il programma RM è fattibile. Questo è ovvio, perché con un solo task, µ = e1/p1. e chiaramente la scadenza può essere rispettata solo se e1 ≤ p1. Se n = 2, allora n(21/n - 1) ≈ 0,828. Quindi, se un insieme di compiti con due compiti non tenta di utilizzare più dell'82,8% del tempo disponibile del processore, il programma RM rispetterà tutte le scadenze. Man mano che n diventa grande, il limite di utilizzo si avvicina a ln(2) ≈ 0,693. Cioè



Ciò significa che se un insieme di attività con un numero qualsiasi di attività non tenta di utilizzare più del 69,3% del tempo disponibile del processore, allora la pianificazione RM rispetterà tutte le scadenze. Nella prossima sezione, rilassiamo il vincolo della priorità fissa e dimostriamo che gli schedulatori a priorità dinamica possono fare meglio di quelli a priorità fissa, nel senso che può raggiungere un utilizzo più elevato. Il costo è un'implementazione un po' più complicata.

**Earliest Deadline First**

Dato un insieme finito di compiti non ripetitivi con scadenze e senza vincoli di precedenza, un semplice algoritmo di schedulazione è quello della data di scadenza più vicina (EDD), noto anche come algoritmo di Jackson (Jackson, 1955). La strategia EDD esegue semplicemente i compiti nello stesso ordine di scadenza, con quello con la scadenza più lontana che viene eseguito per primo. Se due compiti hanno la stessa scadenza, il loro ordine relativo non ha importanza.

***Dato un insieme finito di compiti non ripetitivi T = {τ1, τ2, - - - , τn} con scadenze associate d1, d2, - - - , dn e nessun vincolo di precedenza, una schedulazione EDD è ottimale nel senso che minimizza il massimo ritardo rispetto a tutti gli altri possibili ordinamenti dei compiti.***

L'EDD è ottimale anche per quanto riguarda la fattibilità, perché minimizza il massimo ritardo. Tuttavia, l'EDD non supporta l'arrivo dei compiti e quindi nemmeno l'esecuzione periodica o ripetuta dei compiti. Fortunatamente, EDD è facilmente estendibile per supportare questi aspetti, ottenendo quello che è noto come **earliest deadline first** (EDF) o algoritmo di Horn (Horn, 1974).

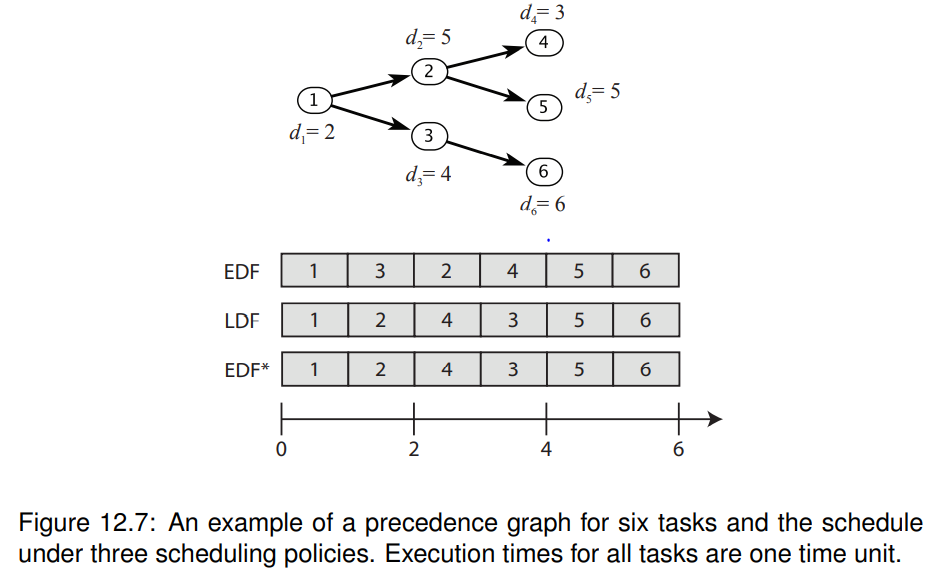
***Dato un insieme di n compiti indipendenti T = {τ1, τ2, - - - , τn} con scadenze associate d1, d2, - - - , dn e tempi di arrivo arbitrari, un qualsiasi algoritmo che in qualsiasi istante esegue il task con la scadenza più vicina tra tutti i task arrivati è ottimale rispetto alla minimizzazione del ritardo massimo.***

La dimostrazione di ciò utilizza un'argomentazione simile a quella dell'interscambio. Inoltre, il risultato è facilmente estendibile per supportare un numero illimitato di arrivi.

Si noti che EDF è un algoritmo di schedulazione a priorità dinamica. Se un compito viene eseguito ripetutamente, può essergli assegnata una priorità diversa a ogni esecuzione. Questo può rendere più complesso da implementare. In genere, per i task periodici, la scadenza utilizzata è la fine del periodo del task anche se è certamente possibile utilizzare altre scadenze per i compiti. Sebbene l'EDF sia più costoso da implementare rispetto a RM, in pratica le sue prestazioni sono generalmente superiori (Buttazzo, 2005b). In primo luogo, RM è ottimale per quanto riguarda la fattibilità solo tra gli schedulatori a priorità fissa, mentre EDF è ottimale rispetto alla fattibilità tra gli schedulatori a priorità priorità dinamica. Inoltre, EDF minimizza anche il ritardo massimo. Inoltre, nella pratica, EDF comporta un minor numero di preemptions (si veda l'Esercizio 2), il che significa un minore overhead per la commutazione di contesto. Questo spesso compensa la maggiore complessità dell'implementazione. Inoltre, a differenza di RM, qualsiasi programma EDF con un utilizzo inferiore al 100% può tollerare aumenti dei tempi di esecuzione e/o riduzioni dei periodi e risultare comunque fattibile.

**EDF with Precedences**

L'EDF è ottimale (minimizza il ritardo massimo) per un insieme di compiti senza precedenze. Cosa succede se ci sono delle precedenze? Dato un insieme finito di compiti, le precedenze tra di essi possono essere rappresentate da un grafo delle precedenze.



ESEMPIO: Consideriamo sei compiti T = {1, - - - , 6}, ciascuno con tempo di esecuzione ei = 1, con le precedenze mostrate nella Figura 12.7. Il diagramma indica che il task 1 deve essere eseguito prima che il 2 o il 3 possano essere eseguiti, che il 2 deve essere eseguito prima del 4 o del 5 e che il 3 deve essere eseguito prima del compito 6. La scadenza per ogni compito è mostrata nella figura. La schedulazione etichettata come EDF è la schedulazione EDF. Questa pianificazione non è fattibile. Il task 4 non rispetta la scadenza. Tuttavia, esiste un programma fattibile. Il programma etichettato come LDF rispetta tutte le scadenze.

L'esempio precedente mostra che l'EDF non è ottimale se ci sono delle precedenze. Nel 1973, Lawler (1973) ha fornito un semplice algoritmo che è ottimale in presenza di precedenze, nel senso che minimizza il massimo ritardo. La strategia è molto semplice. Dato un insieme fisso e finito di compiti con scadenze, la strategia di Lawler costruisce il programma a ritroso, scegliendo per primo l'ultimo compito da eseguire. L'ultimo compito da eseguire è quello da cui non dipende nessun altro compito e che ha la scadenza più recente. L'algoritmo procede a costruire la schedulazione a ritroso, ogni volta scegliendo tra i compiti le cui dipendenze sono già state schedulate quello con la scadenza più recente. Per l'esempio precedente, la schedulazione risultante, etichettata come LDF nella Figura 12.7, è fattibile. L'algoritmo di Lawler è chiamato **latest deadline first** (LDF). LDF è ottimale nel senso che minimizza il ritardo massimo e quindi è ottimale anche per quanto riguarda la fattibilità. Tuttavia, non supporta l'arrivo dei compiti. Fortunatamente, esiste una semplice modifica dell'EDF, proposta da Chetto et al. (1990). EDF\* (EDF con precedenze), supporta gli arrivi e minimizza il ritardo massimo. In questa modifica, regoliamo le scadenze di tutti i compiti. Supponiamo che l'insieme di tutti i task sia T. Per l'esecuzione di un task i ∈ T, si consideri D(i) ⊂ T l'insieme delle esecuzioni di task che dipendono immediatamente da i nel grafico delle precedenze. Per tutte le esecuzioni i ∈ T, definiamo una scadenza modificata:

Immagine che contiene testo

Descrizione generata automaticamente

L'EDF\* è quindi come l'EDF, ma utilizza queste scadenze modificate. L'EDF\* può essere considerata una tecnica di razionalizzazione delle scadenze. Invece di accettare scadenze arbitrarie come dato, questo algoritmo garantisce che le scadenze tengano conto dei compiti successivi. Nell'esempio, non ha molto senso che il task 2 abbia una scadenza più tarda, 5, rispetto ai suoi successori. Quindi EDF\* corregge questa anomalia prima di applicare EDF.

**Scheduling and Mutual Exclusion**

Sebbene gli algoritmi descritti finora siano concettualmente semplici, gli effetti che producono nella pratica sono tutt'altro che semplici e spesso sorprendono i progettisti di sistemi. Questo è particolarmente vero quando i task condividono risorse e utilizzano la mutua esclusione per proteggere l'accesso a tali risorse.

**Priority Inversion**

In linea di principio, uno scheduler preemptive basato sulla priorità esegue sempre il task ad alta priorità abilitato. Tuttavia, quando si utilizza la mutua esclusione, è possibile che un task si blocchi durante l'esecuzione. Se l'algoritmo di schedulazione non tiene conto di questa possibilità, possono verificarsi seri problemi.

L'inversione di priorità è un'anomalia di schedulazione in cui un'attività ad alta priorità viene bloccata mentre vengono eseguite attività non correlate a priorità inferiore. Il fenomeno è illustrato nella Figura 12.9.

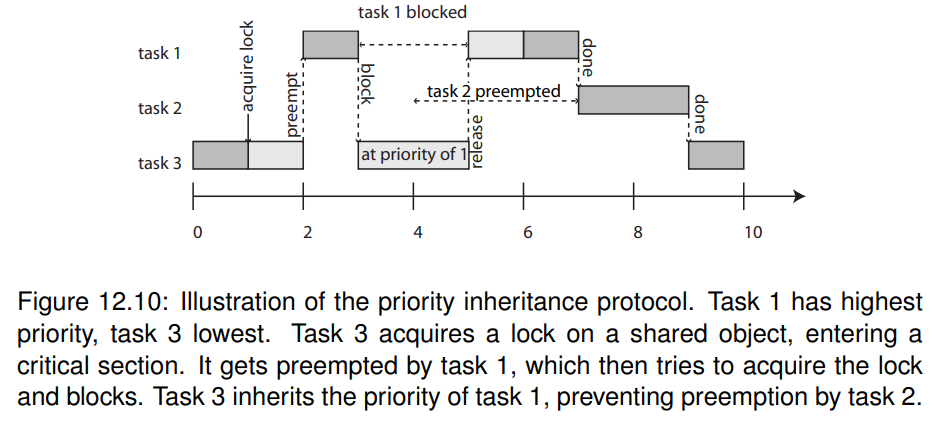


Nella figura, il task 3, un task a bassa priorità, acquisisce un blocco al tempo 1. Al tempo 2, viene preemplificato. Al tempo 2, viene preemptato dal task 1, un task ad alta priorità che poi al tempo 3 si blocca cercando di acquisire lo stesso blocco. Prima che il task 3 raggiunga il punto in cui rilascia il blocco, tuttavia, viene preentato da un task 2 non correlato, che ha priorità media. Il task 2 può essere eseguito per un tempo illimitato, impedendo di fatto l'esecuzione del task 1 a priorità più alta. Questo è quasi certamente non desiderabile.

**Priority Inheritance Protocol**

Nel 1990, Sha et al. (1990) hanno fornito una soluzione al problema dell'inversione di priorità, denominata **priority inheritance**. Nella loro soluzione, quando un task si blocca nel tentativo di acquisire un lock, il task che detiene il lock eredita la priorità del task bloccato. In questo modo, il task che detiene il blocco non può essere pretermesso da un task con priorità inferiore a quella del task che cerca di acquisire il blocco.

La Figura 12.10 illustra l'ereditarietà della priorità. Nella figura, quando il task 1 si blocca nel tentativo di acquisire il blocco detenuto dal task 3, il task 3 riprende l'esecuzione, ma ora con la priorità più alta del task. Pertanto, quando il task 2 si abilita al tempo 4, non prevale sul task 3. Al contrario, il task 3 viene eseguito fino a quando non rilascia il blocco al tempo tempo 5. A quel punto, il task 3 torna alla sua priorità originale (bassa) e il task 1 riprende l'esecuzione. Solo quando il task 1 ha terminato la sua esecuzione, il task 2 è in grado di eseguirla.

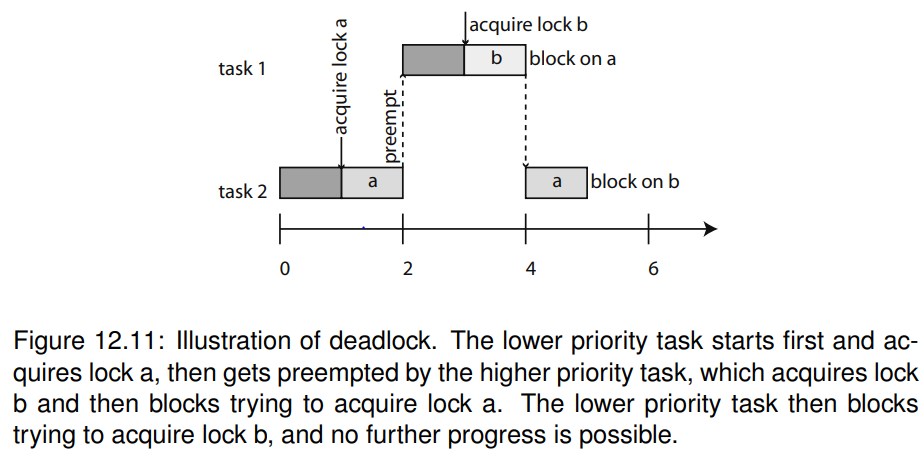


**Priority Ceiling Protocol**

Le priorità possono interagire con i blocchi di mutua esclusione in modi ancora più interessanti. In particolare, nel 1990, Sha et al. (1990) hanno dimostrato che le priorità possono essere utilizzate per prevenire alcuni tipi di deadlock. Lo stallo dell'esempio precedente può essere evitato grazie a una tecnica intelligente chiamata protocollo **priority ceiling** (Sha et al., 1990). In questo protocollo, a ogni blocco o semaforo viene dato un tetto di priorità pari alla priorità del task a più alta priorità che può bloccarlo. Un task τ può acquisire un lock solo se la sua priorità è strettamente superiore ai massimali di priorità di tutti i lock in corso attualmente detenuti da altri task. Intuitivamente, se si impedisce al task τ di acquisire il lock a, allora ci assicuriamo che il task τ non manterrà il lock nel tentativo successivo di acquisire altri lock detenuti da altri task. In questo modo si evita che si verifichino alcuni deadlock.

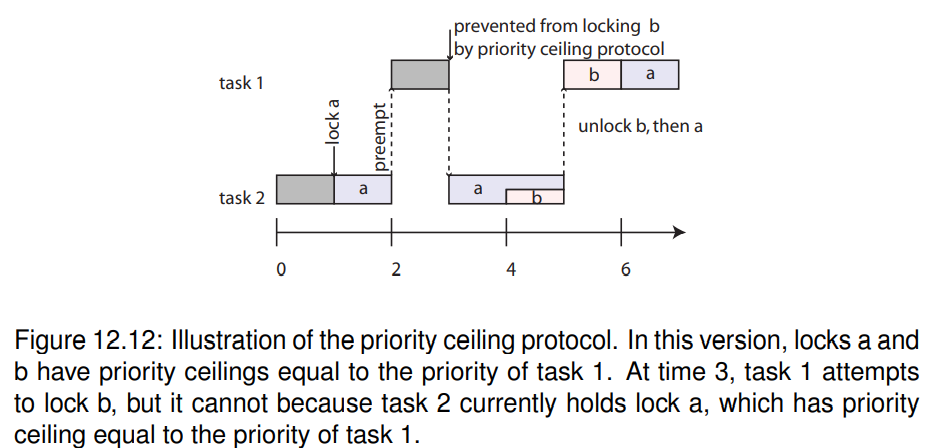
Naturalmente, l'implementazione del protocollo priority ceiling richiede la possibilità di determinare in anticipo quali compiti acquisiscono quali blocchi. Una semplice strategia conservativa consiste nell'esaminare il codice sorgente di ogni task e inventariare i blocchi acquisiti nel codice. Si tratta di una strategia conservativa perché un particolare programma può eseguire o meno una particolare linea di codice. quindi il fatto che un blocco sia menzionato nel codice non significa necessariamente che il task tenterà di acquisirlo.

ESEMPIO: La Figura 12.11 illustra uno scenario in cui due task si bloccano. Nella figura, il task 1 ha una priorità maggiore. Al tempo 1, il task 2 acquisisce il lock a. Al tempo 2, il task 1 preempie il task 2 e al tempo 3 acquisisce il lock b. Mentre detiene il blocco b, tenta di acquisire il blocco a. Poiché a è detenuto dal task 2, il task 2 tenta di acquisire il blocco a. Poiché a è in possesso del task 2, si blocca. Al tempo 4, il task 2 riprende l'esecuzione. Al tempo 5, tenta di acquisire il lock b, che è detenuto dal task 1. Stallo!



Il protocollo priority ceiling impedisce lo stallo dell'Esempio 12.5, come mostrato nella Figura 12.12. Nella figura, quando il task 1 tenta di acquisire il lock b al tempo 3, gli viene impedito di farlo. In quel momento, il blocco a è attualmente detenuto da un altro task (task 2). Il tetto di priorità assegnato al blocco a è uguale alla priorità del task 1, poiché il task 1 è il task a più alta priorità che può acquisire il blocco a.

Poiché la priorità del task 1 non è strettamente superiore a questo tetto di priorità, al task 1 non è consentito acquisire il blocco b. Invece, il task 1 si blocca, consentendo al task 2 di eseguire l'operazione fino al completamento. Al tempo 4, il task 2 acquisisce il blocco b senza impedimenti e al tempo 5 rilascia entrambi i blocchi. Una volta rilasciati entrambi i lock, il task 1, che ha una priorità più alta, non è più bloccato quindi riprende l'esecuzione, anteponendo il task 2.



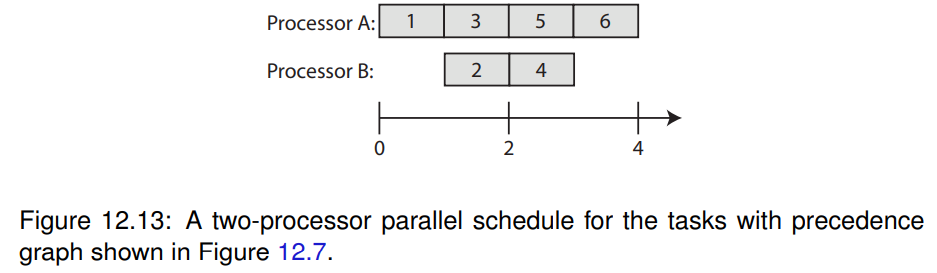
**Multiprocessor Scheduling**

Pianificare le attività su un singolo processore è già abbastanza difficile. Pianificarli su più processori è ancora più difficile. Si consideri il problema di schedulare un insieme finito di compiti con precedenza su un numero finito di processori con l'obiettivo di minimizzare il makespan. È noto che questo problema è NP-hard. Tuttavia, esistono strategie di schedulazione efficaci ed efficienti. Una delle più semplici è nota come algoritmo di schedulazione a livello Hu. Esso assegna una priorità a ogni task τ in base al livello, che è la somma maggiore dei tempi di esecuzione dei task su un percorso nel grafo delle precedenze da τ a un altro task senza dipendenze. I task con livelli maggiori hanno una priorità più alta rispetto a quelli con livelli minori.

Lo scheduling a livello di Hu fa parte di una famiglia di metodi di percorso critico, in quanto enfatizza il percorso attraverso il grafo delle precedenze con il maggior tempo totale di esecuzione. Sebbene non sia ottimale, è noto che approssima la soluzione ottimale per la maggior parte dei grafi (Kohler, 1975; Adam et al., 1974). Una volta assegnate le priorità ai compiti, uno schedulatore di liste ordina i compiti in base alle priorità e li assegna ai processori nell'ordine dell'elenco ordinato, man mano che i processori si rendono disponibili.

Nel grafico delle precedenze della Figura 12.7, il compito 1 ha livello 3, i compiti 2 e 3 hanno livello 2, mentre i compiti 4, 5 e 6 hanno livello 1,Quindi, uno schedulatore di livello Hu assegnerà al task 1 la priorità più alta, ai task 2 e 3 la priorità media e ai task 4, 5 e 6 la priorità più bassa.

La Figura 12.13 mostra una schedulazione a due processori costruita con l'algoritmo di schedulazione a livello Hu per il grafico delle precedenze mostrato nella Figura 12.7. Il makespan è 4.

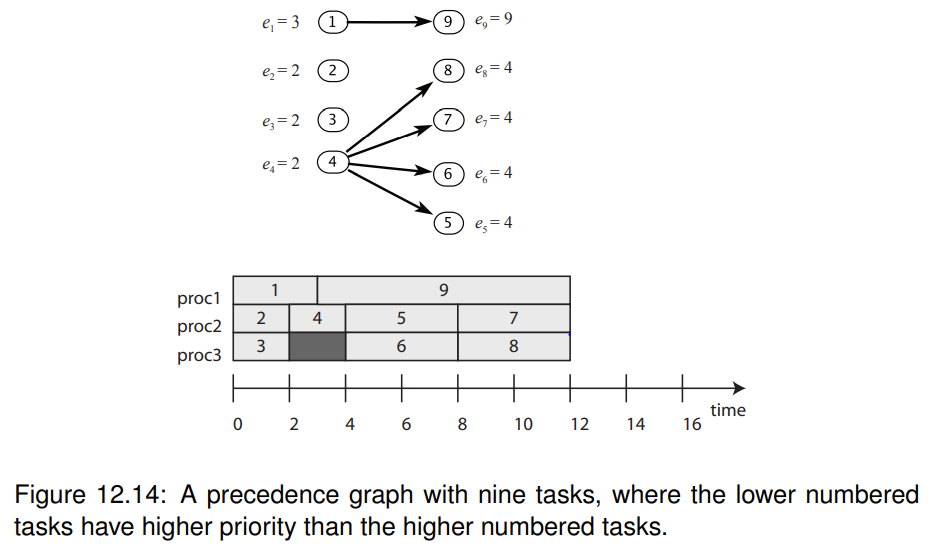


**Scheduling Anomalies**

Tra le peggiori insidie nella progettazione di sistemi embedded ci sono le anomalie di programmazione, che causano comportamenti inattesi o controintuitivi a causa di piccoli cambiamenti nelle condizioni operative di un sistema. Abbiamo già illustrato due di queste anomalie, l'inversione di priorità e il deadlock. Ne esistono molte altre. La possibile portata dei problemi che possono sorgere sono ben illustrati dalle cosiddette anomalie di Richard (Graham, 1969). Queste dimostrano che i programmi multiprocessore sono non-montonici, il che significa che i miglioramenti delle prestazioni a livello locale possono possono comportare un degrado delle prestazioni a livello globale, e fragili, nel senso che piccoli cambiamenti possono avere grandi conseguenze. Le anomalie di Richard sono riassunte nel seguente teorema.

***Se un insieme di compiti con priorità, tempi di esecuzione e vincoli di precedenza fissi viene pianificato su un numero fisso di processori in base alla priorità, aumentare il numero di processori, ridurre i tempi di esecuzione o indebolire i vincoli di precedenza può aumentare la lunghezza della schedulazione.***

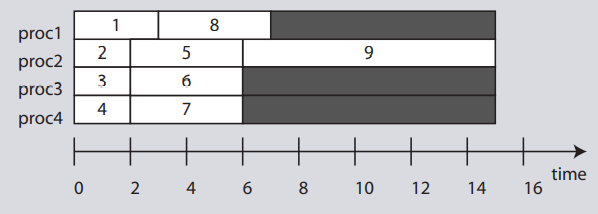
Dimostrazione: Il teorema può essere dimostrato con l'esempio riportato nella Figura 12.14. L'esempio ha nove compiti con tempi di esecuzione come mostrato nella figura. Assumiamo che ai compiti sono assegnate delle priorità, in modo che i compiti numerati più bassi abbiano una priorità maggiore di quelli rispetto a quelli più alti. Si noti che questo non corrisponde a un'assegnazione di priorità al percorso critico, ma è sufficiente per dimostrare il teorema. La figura mostra un programma a tre processor in base alle priorità. Si noti che il makespan è 12.



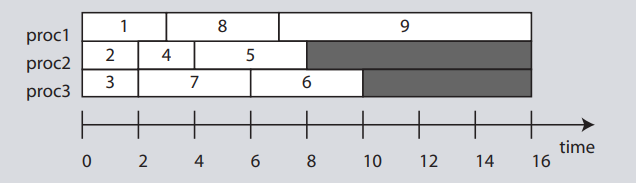
In primo luogo, si consideri cosa succede se i tempi di esecuzione vengono tutti ridotti di un'unità di tempo. Uno schema che rispetta le priorità e le precedenze è mostrato di seguito:



Si noti che il makespan è aumentato a 13, anche se la quantità totale di calcolo è diminuita in modo significativo. Poiché raramente i tempi di calcolo sono noti con esattezza, questa forma di fragilità è particolarmente preoccupante. Consideriamo poi cosa succede se aggiungiamo un quarto processore e manteniamo tutto il resto come nel problema originale. Il programma risultante è mostrato di seguito:

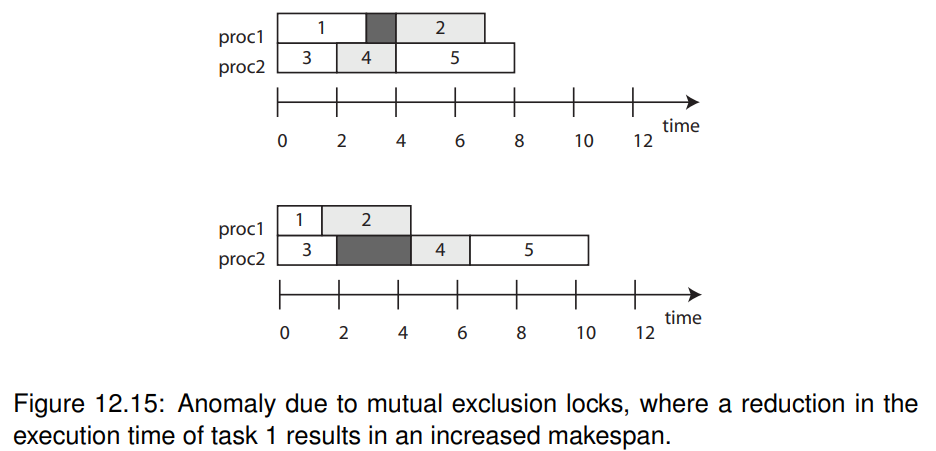


Ancora una volta, il makespan è aumentato (a 15 questa volta) anche se abbiamo aggiunto il 33% di potenza di elaborazione in più rispetto a quella originariamente disponibile. Consideriamo infine cosa succede se indeboliamo i vincoli di precedenza rimuovendo le precedenze tra il compito 4 e i compiti 7 e 8. Il programma risultante è mostrato di seguito:



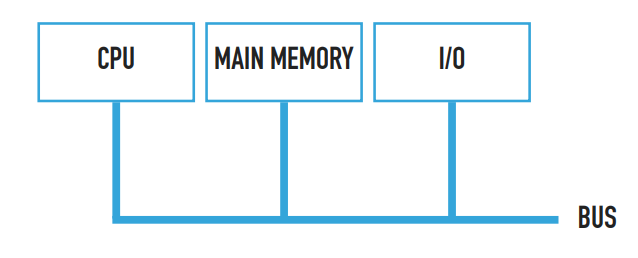
Il makespan è ora aumentato a 16, anche se l'indebolimento dei vincoli di precedenza aumenta la flessibilità di schedulazione. Un semplice schema di schedulazione basato sulla priorità come questo non sfrutta l'indebolimento dei vincoli.

Questo teorema è particolarmente preoccupante quando ci si rende conto che i tempi di esecuzione del software sono raramente noti con esattezza (si veda il Capitolo 16). Le politiche di programmazione si basano su approssimazioni e il comportamento a tempo di esecuzione può essere del tutto inaspettato. Un'altra forma di anomalia si verifica in presenza di blocchi di mutua esclusione. Un'illustrazione è Figura 12.15. In questo esempio, cinque task sono assegnati a due processori utilizzando uno scheduler di assegnazione statica. I task 2 e 4 si contendono un mutex. Se il tempo di esecuzione del task 1 si riduce, l'ordine di esecuzione dei task 2 e 4 si inverte, con un conseguente aumento del tempo di esecuzione. Questo tipo di anomalia è piuttosto comune nella pratica.

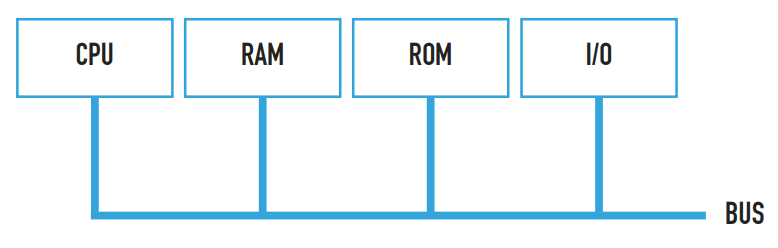


**ARM Instruction Set**

Nella Macchina di Von Neumann, a livello ISA la CPU esegue programmi in linguaggio macchina (ML) memorizzati nella memoria principale. I programmi ML sono sequenze di istruzioni che vengono eseguite dalla CPU una dopo l'altra. Ogni istruzione ML è memorizzata come configurazione specifica di bit. Le istruzioni ML sono memorizzate in modo compatto secondo l'ordine di esecuzione. Le istruzioni ML possono pilotare i dispositivi di I/O per ottenere (INPUT) o fornire (OUTPUT) dati in forma leggibile dall'uomo.



Nelle macchine attuali possono essere presenti più memorie. La più grande è solitamente la memoria ad accesso casuale, che è "volatile". Le memorie di sola lettura non volatili (ROM) sono incluse per fornire alla CPU i programmi all'accensione. Possono essere disponibili anche altri tipi di memoria.



La memoria principale è organizzata come un array di celle contenenti lo stesso numero di bit. Nella maggior parte dei casi, ogni cella di memoria contiene un byte (otto bit). A ogni cella si accede tramite un numero chiamato indirizzo. Una memoria di 64KB (65536 byte) richiede un numero di 16 bit (2^16=65536) per indirizzare una cella.

Le celle contigue possono essere raggruppate per memorizzare valori a più byte. Il contenuto delle celle 0002 e 0003 può essere interpretato come un valore a 16 bit.

16 bit

Il contenuto delle celle 0400-0403 può essere interpretato come un valore a 32 bit. Ma come? Considerando gli indirizzi dal più basso al più alto la rappresentazione little-endian codifica il valore a più byte, dal byte meno significativo (LSB), l'estremità piccola, al byte più significativo (MSB). (es. 1B27 e 0F010903) . La rappresentazione big-endian utilizza l'ordine opposto in modo che i valori a più byte vengano letti dall'MSB, l'estremità grande, verso l'LSB (ad esempio 271B e 0309010F). Il contenuto della memoria viene solitamente scaricato man mano che gli indirizzi crescono. La rappresentazione big-endian è più facile da decodificare per gli esseri umani. Tuttavia, la maggior parte delle architetture adotta la rappresentazione little-endian.

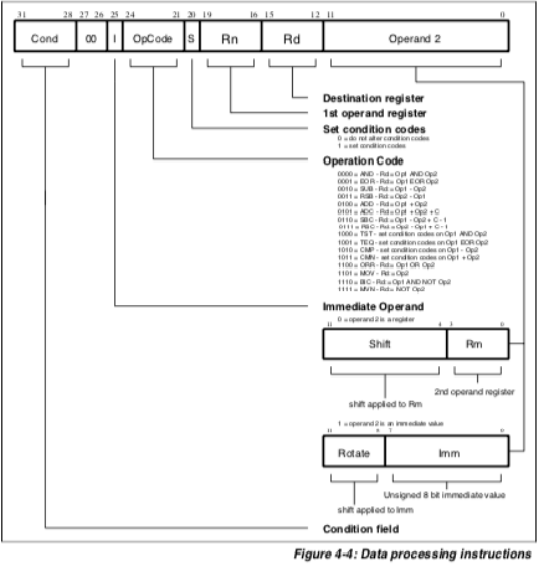
Si consideri un sistema con un bus di indirizzi a n-bit. L'intera gamma di indirizzi deve essere condivisa tra tutti i componenti collegati al bus. È necessario scegliere componenti di dimensioni adeguate. Deve essere presente un circuito di decodifica che utilizzi, ad esempio, un segnale di Chip-Select (CS) collegato al pin omonimo di ciascun chip del componente. Schemi più sofisticati (ad esempio, decodificatori programmabili o memorie) supportano la sovrapposizione degli di indirizzi per i componenti o spazi di indirizzi virtuali anche più grandi di quelli che il bus fisico può indirizzare.

**Formati del set di istruzioni ARM**

Alcuni formati definiscono più di un'operazione. Molte operazioni di elaborazione dati possono essere essere interpretate come eseguite da una singola istruzione flessibile. L'istruzione esegue una determinata operazione aritmetica o logica su uno o due operandi. Il primo operando, se presente, è sempre un registro (Rn). Nelle varianti MOV e MVN il primo operando viene ignorato.

Immagine che contiene tavolo

Descrizione generata automaticamente

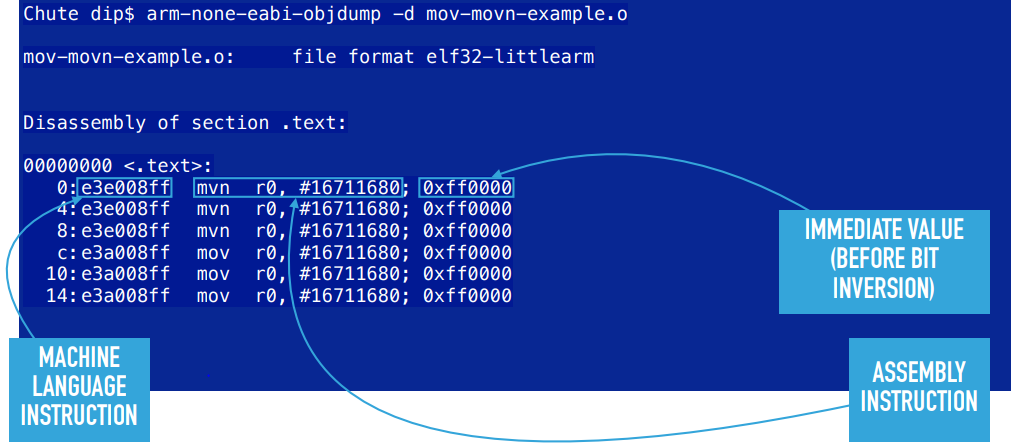


Per analizzare la codifica di queste istruzioni, il codice sorgente dell'assemblaggio (mov-movnexample.s) è stato tradotto in linguaggio macchina (ML) utilizzando un vero assemblatore (GNU armnone-eabi-as).

Immagine che contiene testo

Descrizione generata automaticamente

Il codice oggetto risultante (mov-movn-example.o) viene poi disassemblato (GNU arm-none-eabi-objdump) che rivela come le sei istruzioni sono state codificate.



**DATA PROCESSING INSTRUCTION ENCODING**

* Le istruzioni ML per **caricare un immediato in un registro** sono **MOV Rd, #Imm, Rotate** e **MVN Rd, #Imm, Rotate.**

Le pseudoistruzioni MOV Rd, #LegalValue32 e MVN Rd, #LegalValue32 sono accettate dall'assemblatore. (come nel caso di mvn r0, #16711680). L'assemblatore si occupa, se possibile, di convertirle in MOV Rd, #Imm, Rotate e MVN Rd, #Imm, Rotate. Le tre istruzioni di assemblaggio che seguono condividono la stessa codifica in linguaggio macchina (mvn r0, #0xFF, 16).

INFATTI: 0xFF0000= 15\*16^5+15\*16^4= 16711680.

Immagine che contiene tavolo

Descrizione generata automaticamente

Se l'immediato a 32 bit desiderato è 0x00FF0000, solo un bit (bit 22) cambia rispetto alla codifica di mov r0, #0xFF00FFFF. L'istruzione codificata è mov r0, #0xFF, 16

Immagine che contiene tavolo

Descrizione generata automaticamente

* **Shifted register second operand** con quantità di shift specificata dall'istruzione.

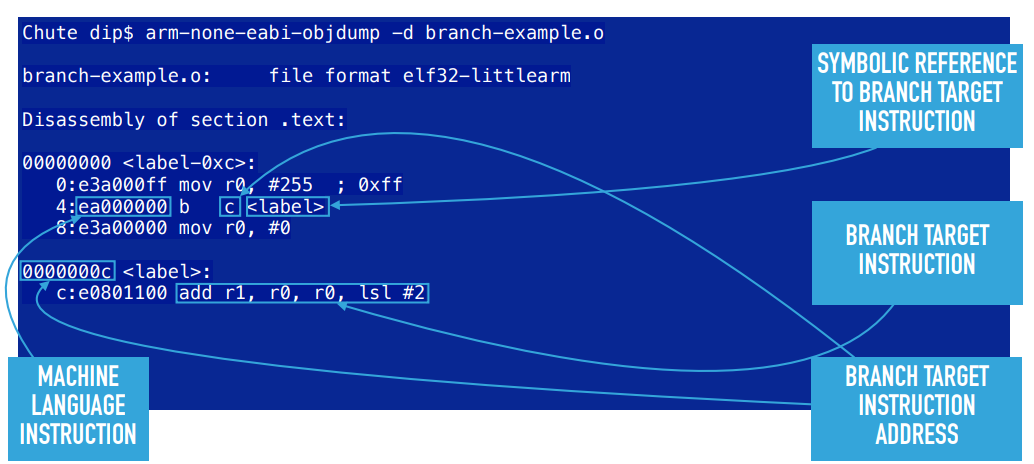
I bit 11-7 contengono la quantità di shift/rotazione.

I bit 6-5 indicano una delle operazioni di shift: LSL (0b00), LSR (0b01), ASR (0b10) e ROR (0b11).

il bit 4 è 0, i bit 3-0 indicano il registro contenente il valore da spostare o ruotare.

Immagine che contiene tavolo

Descrizione generata automaticamente



* **Branch and Branch with Link instructions**

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

Immagine che contiene tavolo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

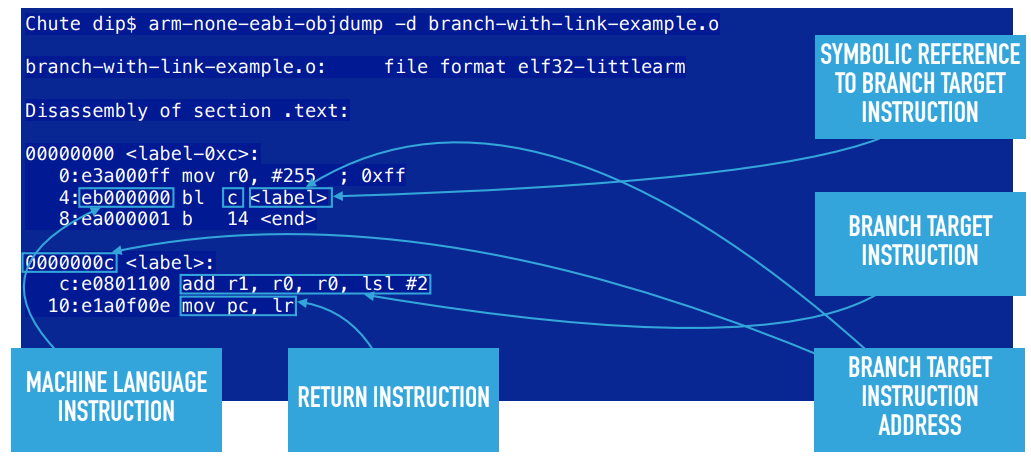


Immagine che contiene tavolo

Descrizione generata automaticamente



* **Branch eXchange instruction**

L'indirizzo di destinazione è contenuto nel registro specificato nei bit 3-0. L'esecuzione prosegue in stato ARM o in stato Thumb a seconda del valore del bit 0 dell'indirizzo di destinazione. Se il bit è impostato l'esecuzione prosegue in modalità Thumb, altrimenti prosegue in modalità ARM-32.

Immagine che contiene tavolo

Descrizione generata automaticamente