



NoiStudiamo

DISPENSA DI TEORIA



Architettura degli elaboratori

La teoria completa della materia, dal modello di Von Neumann
ai circuiti digitali. Fatto da studenti, per studenti.

A CURA DI

Angelo Faro

Ottobre 2026

Indice dei capitoli

Dispensa di sola teoria, organizzata per argomenti nell'ordine in cui si sviluppa il corso.

1. Introduzione e modelli di calcolo	3
2. Memoria, bus e input output	8
3. Il processore	12
4. Rappresentazione dell'informazione	16
5. Linguaggio macchina e assembly	19
6. Il livello software	27
7. Pipeline e prestazioni	31
8. Struttura base del processore e controllo	36
9. Algebra booleana e reti logiche	40
10. Circuiti aritmetici	44
11. Tecnologia dei circuiti digitali	48

Introduzione e modelli di calcolo

Che cos'è un algoritmo, come lo si formalizza con la macchina di Turing e come la macchina di Von Neumann ne rende possibile la realizzazione fisica.

Algoritmo e calcolatore

DEFINIZIONE

Un **algoritmo** è un insieme di istruzioni, scritte e codificate in un certo linguaggio, che stabilisce come eseguire un dato lavoro. In forma equivalente, un algoritmo è una particolare macchina di Turing oppure un programma della macchina di Von Neumann.

DEFINIZIONE

Un **computer** è una macchina che computa, cioè esegue un dato lavoro in modo automatico eseguendo un algoritmo specificato in un linguaggio comprensibile alla macchina.

Per architettura di un calcolatore si intende l'insieme delle principali unità funzionali e le modalità con cui interagiscono tra loro. Le funzioni di base di un calcolatore sono quattro:

- memorizzare i dati;
- elaborare i dati;
- trasferire i dati;
- controllare i dati.

Esistono molti tipi di computer: mainframe, server, desktop, laptop (notebook), tablet, palmari (PDA), smartphone e lettori multimediali. Un personal computer comprende CPU, memoria, disco, input e output.

La macchina di Turing

La macchina di Turing è un modello teorico di calcolo. Non fu mai realizzata fisicamente, ma Turing dimostrò che il suo automa può risolvere qualsiasi problema per il quale sia possibile scrivere un algoritmo composto da un numero finito di operazioni. L'attenzione di Turing è rivolta al processo di calcolo, indipendentemente da come avvenga fisicamente: una macchina di Turing è un dispositivo ideale, indipendente da ogni sua possibile realizzazione fisica.

DEFINIZIONE

La **macchina di Turing** è un dispositivo di calcolo in grado di operare, mediante una successione finita di passi, secondo determinate regole (il programma), su un numero finito di simboli, facendo astrazione dai limiti di spazio (memoria), di tempo (lunghezza della computazione) e da possibili errori di calcolo.

COMPONENTI

- **Nastro:** di lunghezza infinita, diviso in celle che formano una sequenza lineare. Ogni cella contiene al più un simbolo, altrimenti è vuota.
- **Testina:** si sposta lungo il nastro e può leggere, scrivere o cancellare i simboli.
- **Unità di controllo:** stabilisce il comportamento della macchina in base allo stato interno.
- Un insieme finito di stati, un alfabeto finito (che comprende il simbolo nullo) e un insieme finito di istruzioni.

FUNZIONAMENTO

La macchina analizza il nastro una cella alla volta. Ad ogni passo, in accordo con il suo stato corrente:

1. legge un simbolo sul nastro;
2. decide il suo prossimo stato interno;
3. scrive un simbolo sul nastro;
4. decide se spostare la testina di una posizione.

Il comportamento può essere programmato con un insieme di regole dette **quintuple**. Ad esempio la quintupla (0, A, 1, B, ·) indica che la macchina, trovandosi nello stato interno 0 e leggendo il simbolo A, passa allo stato interno 1, scrive B sul nastro e non sposta la testina. Il quinto elemento indica infatti il movimento: destra, sinistra oppure nessuno spostamento. La quintupla (1, B, 0, A, >) indica invece che, nello stato 1 e leggendo B, la macchina passa allo stato 0, scrive A e sposta la testina di una posizione a destra. Una quintupla ha dunque la forma (stato corrente, simbolo letto, prossimo stato, simbolo scritto, direzione).

CONDIZIONI PER LA FINITEZZA

1. Il numero di simboli è fissato e finito: con infiniti simboli potrebbero esistere simboli talmente simili da non essere distinguibili.
2. Il numero di caselle del nastro osservabili in una volta è finito.
3. È possibile ricordare distintamente solo un numero finito di stadi precedenti del processo di calcolo.
4. Le operazioni possibili sono limitate: cambiare il contenuto della casella osservata, cambiare la casella osservata, cambiare il proprio stato mentale, osservare nuove caselle a distanza al più prefissata.

CONDIZIONE DI DETERMINATEZZA

Le azioni della macchina, ad ogni istante, dipendono solo dal simbolo contenuto nella casella osservata in quell'istante e dallo «stato mentale» corrente, cioè da ciò che la macchina ricorda in quel momento (per Turing, l'influenza degli stadi precedenti del calcolo su quello attuale). Un procedimento così caratterizzato è simulabile da una macchina di Turing. Se l'analisi di Turing è corretta, ogni funzione parziale a valori interi che un essere umano può calcolare rispettando le condizioni di finitezza e determinatezza è Turing calcolabile.

FUNZIONI PARZIALI E MACCHINA UNIVERSALE

Una **funzione parziale** è una funzione non necessariamente definita per ogni possibile input: per alcuni input la computazione può non terminare mai e quindi non restituire alcun output. Le funzioni totali rappresentano invece le computazioni che terminano sempre. Una funzione parziale si dice **Turing calcolabile** se esiste una macchina di Turing in grado di calcolarne il valore (quando esiste) in un numero finito di passi. Esistono funzioni che nessuna macchina di Turing può calcolare.

Le funzioni parziali sono usate nella definizione della macchina universale per quattro ragioni:

- rappresentano la **computabilità reale**: non tutte le funzioni descrivibili sono calcolabili in tempo finito e alcune computazioni potrebbero proseguire indefinitamente (ad esempio nel tentativo di risolvere il problema della fermata di un'altra macchina di Turing);
- garantiscono la **generalità**: la macchina universale deve simulare qualsiasi macchina di Turing, comprese quelle che per certi input non terminano;
- modellano con **accuratezza i problemi di decisione**: per molti di essi, come quello della fermata, non esiste un algoritmo che risponda in tutti i casi;
- rendono il modello più realistico e potente, capace di rappresentare anche i processi che continuano senza produrre un risultato.

DEFINIZIONE

Una **macchina di Turing universale** è una macchina in grado di calcolare tutte le funzioni calcolabili da ogni singola macchina di Turing, cioè di simulare qualsiasi altra macchina di Turing, comprese quelle che non terminano mai.

DEFINIZIONE

Tesi di Turing: ogni funzione parziale calcolabile con un algoritmo è una funzione parziale calcolabile da una macchina di Turing.

Il problema della fermata

Il problema della fermata (Halting Problem), formulato da Turing nel 1936, chiede se esista un algoritmo che, dati in input un programma e un input, stabilisca se quel programma si fermerà oppure continuerà indefinitamente.

Turing dimostrò che **un tale algoritmo non esiste**, con una dimostrazione per assurdo di tipo autocontraddittorio:

1. Si suppone che esista un algoritmo H che, dati un programma P e un input x , risponde «sì» se P si ferma su x e «no» altrimenti.
2. Si costruisce un programma Q che, preso in input un programma P' , lo esegue su se stesso: se $H(P', P')$ risponde «no», Q si ferma; se risponde «sì», Q entra in un ciclo infinito.
3. Si esegue Q su se stesso. Se $H(Q, Q)$ risponde «no», Q dovrebbe fermarsi, ma per costruzione entra in un ciclo infinito. Se $H(Q, Q)$ risponde «sì», Q non si ferma, il che contraddice la risposta di H .

La contraddizione dimostra che il problema della fermata è **indecidibile**: esistono problemi per i quali nessun algoritmo generale può fornire la risposta in tutti i casi. Sono invece decidibili i problemi per cui tale algoritmo esiste.

VERSIONE CON LE MACCHINE MF E MFL

Si suppone di costruire una macchina di Turing MF che, dati in ingresso i dati e le istruzioni di una generica macchina M , calcola se M si ferma. Si costruisce poi MFL , uguale a MF tranne che, dopo aver scoperto che M si ferma, esegue un ciclo infinito: se MF esiste, anche MFL si può costruire facilmente.

Si prende come macchina generica $M = MFL$, cioè si chiede a MFL di calcolare se essa stessa si ferma. Se MFL si ferma, vuol dire che ha scoperto che non si ferma; se non si ferma, vuol dire che ha scoperto che si ferma. In entrambi i casi si arriva a una conclusione assurda.

In conclusione, la macchina di Turing «non può capire se stessa»: non può calcolare fino in fondo il comportamento di un'altra macchina di Turing né il proprio.

Il test di Turing

Turing formulò la domanda se una macchina possa pensare in termini di gioco, detto gioco dell'imitazione. Il gioco coinvolge tre persone: A (uomo), B (donna) e un interrogante, chiuso in una stanza separata e che conosce gli altri due solo con le etichette X e Y . L'interrogante deve stabilire chi sia l'uomo e chi la donna ponendo domande; A cerca di ingannarlo, B cerca di aiutarlo. Turing si chiese cosa accadrebbe se una macchina prendesse il posto di A . Il test consiste quindi nell'indurre un essere umano a credere di conversare con un altro essere umano e non con una macchina.

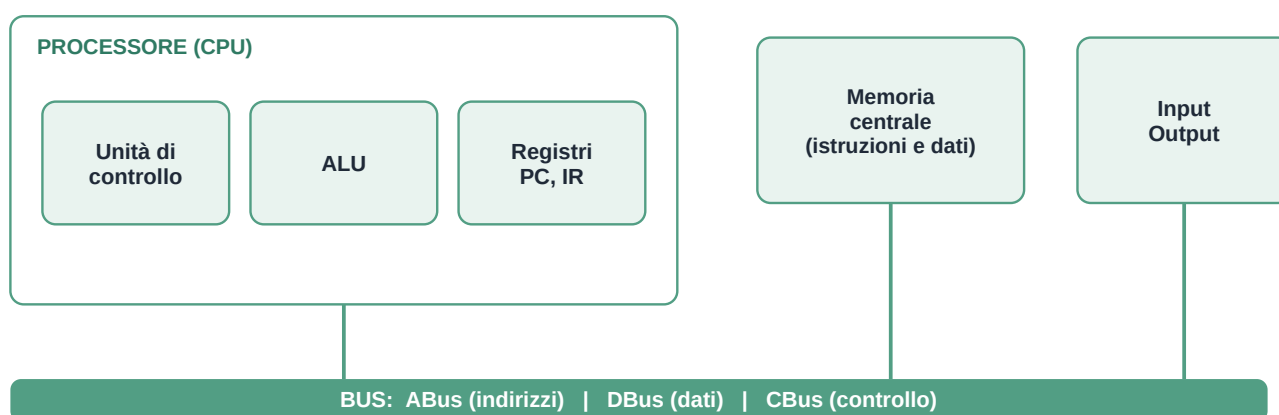
Le risposte sono dattiloscritte, così che né il tono della voce né la calligrafia aiutino l'interrogante. Il **criterio di similitudine** del test è la capacità di interagire linguisticamente, cioè di usare un insieme di simboli (le lettere) e di regole lessicali, grammaticali e logiche per combinarli in parole e frasi. Il programma più noto dei primi tentativi è **Eliza** (1966, Joseph Weizenbaum), che simula uno psicoterapeuta: poco convincente, ma capace di ingannare

persone ingenuie (il test prevede invece un operatore esperto). Secondo uno studio citato dal prof (Mei et al., PNAS 2024), ChatGPT 4 avrebbe superato il test.

La macchina di Von Neumann

Il modello della macchina di Turing (1936) è il primo modello di calcolo definito ed è tuttora la definizione teorica di algoritmo; il modello di Von Neumann (1946) ne traduce in pratica il concetto: tutti i computer moderni sono ancora strutturati come una macchina di Von Neumann. La sua geniale intuizione è usare la memoria per conservare sia le istruzioni sia i dati dei calcoli. Dal punto di vista formale è una terna (N, IS, P), dove:

- **N** = {0, 1, 2, 3, ...} è l'insieme dei numeri naturali, interpretabile come linguaggio della macchina;
- **IS** = {ZERO, INC, SOM, SOT, MOL, DIV, UGUALE, MINORE, SALCOND, ALT} è l'*instruction set*, l'insieme delle istruzioni generiche della macchina;
- **P** = {I0, I1, I2, ..., I|P| meno 1} è una sequenza finita e non vuota di istruzioni specifiche, prese da IS, con particolari valori delle variabili. Questa sequenza è il **programma** della macchina.



Schema a blocchi del modello di Von Neumann: processore, memoria e input output collegati dal bus.

Nel modello la **memoria** contiene sia le istruzioni sia i dati dei calcoli. Ad ogni locazione di memoria è associato l'indice della locazione, detto **indirizzo**. Il programma è registrato nelle locazioni con indirizzi da 0 a |P| meno 1, in modo che la i-esima istruzione sia nella locazione i.

IL PROCESSORE

1. Il **contatore di programma** (PC): locazione che contiene l'indirizzo dell'istruzione da eseguire.
2. Il **registro delle istruzioni** (IR): locazione che contiene l'istruzione da eseguire.
3. L'**unità aritmetico logica** (ALU): esegue le istruzioni.
4. Il **controllo**: tramite una sequenza di cambiamenti di stato fa avvenire l'esecuzione dell'istruzione.

LA COMPUTAZIONE

Un programma è una lista di istruzioni registrata in memoria, eseguite una alla volta nell'ordine specificato, fino a quando non si incontra l'istruzione speciale di arresto (ALT). Il ciclo di esecuzione è il seguente:

1. si legge il contenuto del contatore di programma, cioè l'indirizzo dell'istruzione da eseguire;
2. si preleva (*fetch*) l'istruzione nel registro delle istruzioni;
3. si decodifica l'istruzione, identificandola nell'insieme IS;
4. si inviano i comandi all'ALU affinché esegua l'istruzione;
5. si recuperano dalla memoria i dati necessari, tramite gli indirizzi delle locazioni;
6. si attende che l'ALU completi l'operazione;

7. si inserisce il risultato in una locazione di memoria, specificata dall'operando più a sinistra dell'istruzione (ad esempio, per SOM(M1, M2) il risultato è registrato in M1). Se l'istruzione è un salto condizionato, il controllo registra nel contatore l'indirizzo della prossima istruzione;
8. si incrementa il contatore, a meno che l'istruzione sia un salto condizionato o ALT;
9. il ciclo dal passo 1 all'8 si ripete fino a quando si incontra un'istruzione ALT.

COMPONENTI DELL'HARDWARE

La CPU (*central processing unit*) possiede memorie di lavoro proprie, i **registri**; la sua velocità si esprime in MHz o GHz, cioè nel numero di cicli al secondo del processore. La **memoria centrale** fa da deposito di dati e istruzioni da eseguire ed è composta da ROM, RAM e cache.

ESEMPIO DI PROGRAMMA

Programma per la macchina di Von Neumann che stabilisce se il numero n registrato in M0 è divisibile per 3: scrive 1 in M1 se lo è, 0 altrimenti.

INDIRIZZO	ISTRUZIONE	EFFETTO
0	ZERO(M1)	Scrivi 0 in M1
1	INC(M1)	Scrivi 1 in M1
2	INC(M1)	Scrivi 2 in M1
3	INC(M1)	Scrivi 3 in M1
4	ZERO(M2)	Scrivi 0 in M2
5	SOM(M2, M0)	Copia n in M2
6	DIV(M2, M1)	$M2 = n / 3$ (divisione intera)
7	MOL(M2, M1)	$M2 = 3 \cdot (n / 3)$
8	SOT(M0, M2)	$M0 = n$ meno $3 \cdot (n / 3) = r$, il resto
9	SALCOND(M0, 13)	Se $r \neq 0$ salta all'istruzione 13
10	ZERO(M1)	Scrivi 0 in M1
11	INC(M1)	Scrivi 1 in M1
12	ALT	Si ferma: $M1 = 1$, il numero è divisibile
13	ZERO(M1)	Scrivi 0 in M1
14	ALT	Si ferma: $M1 = 0$, il numero non è divisibile

Programma tratto dalle slide del corso e riportato in forma coerente (nella slide l'istruzione 5 e i commenti delle righe 6 e 7 non corrispondono).

L'algebra di Boole

Nel 1854 George Boole inventò un nuovo tipo di algebra, l'algebra booleana, in cui le procedure di calcolo si effettuano con tre operatori logici: **AND**, **OR** e **NOT**. Su di essa si fonda la progettazione dei circuiti digitali, trattata nei capitoli successivi.

Memoria, bus e input output

I tre elementi che, insieme al processore, compongono il modello di Von Neumann: dove si conservano i dati, come viaggiano e come il calcolatore comunica con l'esterno.

La memoria principale

La memoria principale è un contenitore ordinato di celle. Nelle celle vengono immesse o prelevate le istruzioni del software e i dati di I/O. Ogni cella è ampia un byte e possiede un preciso indirizzo (*address*). Gli indirizzi partono da zero e l'ultimo è pari al numero totale di celle meno uno. L'insieme di tutte le celle è detto **spazio di indirizzamento** della memoria.

L'ampiezza dello spazio di indirizzamento è determinata dall'ampiezza del bus indirizzi. Se il bus indirizzi ha n linee, lo spazio di indirizzamento vale 2^n celle e gli indirizzi vanno da 0 a $2^n - 1$. Il valore di una cella è conservato fintanto che la memoria è correttamente alimentata: la memoria principale è quindi **volatile**, cioè perde i propri contenuti ogni volta che la macchina viene spenta.

RAM, DRAM E SRAM

La memoria è realizzata in gran parte con tecnologia **RAM** (*random access memory*). I bit della RAM sono microcondensatori, per questo si parla anche di **DRAM** (RAM dinamica): per evitare la perdita dei dati la memoria deve essere continuamente «rinfrescata» da un segnale elettrico a frequenza costante per tutto il tempo in cui il sistema è alimentato. Le DRAM sono prodotte in banchi di dimensione fissa e montati sulla scheda madre su schede DIMM (*dual inline memory module*) a due facce; la tecnologia più diffusa fino al 2011 è la DDR3 SDRAM (*synchronous dynamic RAM*).

Lettura e scrittura delle celle non sono immediate: richiedono un tempo tecnico detto **tempo di accesso alla memoria**, molto alto rispetto ai tempi del calcolatore. Questo squilibrio è all'origine del problema del **collo di bottiglia** (*bottleneck*).

Per contrastare il ritardo è nata la **SRAM** (RAM statica), in cui al posto dei microcondensatori si usa la tecnologia dei flip flop. È molto più veloce, ma costosa e voluminosa a parità di cella: il suo uso è limitato a memorie di transito tra memoria e processore, le **memorie cache**.

BIOS E ROM

Nella memoria di sistema deve esistere un'area che non perde i valori allo spegnimento. Per avviarsi (*bootstrap*) il sistema deve infatti immettere sul bus le istruzioni iniziali per configurare i dispositivi di base (**POST**, *power on self test*) e caricare il sistema operativo da una memoria secondaria. Questa area, riservata all'interno dello spazio di indirizzamento pur conservando la struttura tipica di indirizzo e cella, è denominata **BIOS** (*basic input output system*).

Le regioni che contengono il BIOS sono realizzate in tecnologia **ROM** (*read only memory*, ad esempio le memorie Flash ROM), che mantiene il contenuto anche senza alimentazione. Codice e programmi contenuti in memorie non volatili si chiamano **firmware**. Gran parte della ROM è realizzata con tecnologie **EPROM** o **EEPROM**, che mantengono i dati senza alimentazione ma possono essere riprogrammate: per questo le caratteristiche di avvio e alcune caratteristiche funzionali del sistema si possono modificare o aggiornare con il programma di setup del BIOS oppure sostituendo l'intero BIOS con speciali procedure di configurazione.

TIPO	TECNOLOGIA	CARATTERISTICHE
DRAM	Microcondensatori	Economica e densa, richiede il refresh, accesso lento. Memoria principale.
SRAM	Flip flop	Molto veloce, costosa e voluminosa. Memoria cache.
ROM, EPROM, EEPROM	Non volatile	Mantiene i dati senza alimentazione. BIOS e firmware.

ORGANIZZAZIONE DELLA MEMORIA

La memoria si organizza in una gerarchia che va dalle memorie più veloci, piccole e costose a quelle più lente, capienti ed economiche: **registri**, **cache**, **memoria principale** e, tra le memorie secondarie (di massa), **dischi magnetici**, **nastri** e **dischi ottici**. Il confronto tra i principali tipi di memoria è il seguente.

	SRAM	DRAM	NAND FLASH	NOR FLASH
Non volatile	No	No	Sì	Sì
Prezzo per GB	Alto	Basso	Molto basso	Basso
Velocità di lettura	Molto veloce	Veloce	Lenta	Veloce
Velocità di scrittura	Molto veloce	Veloce	Lenta	Lenta
Scrittura minima	Byte	Byte	Pagina	Byte
Lettura minima	Byte	Pagina	Pagina	Byte
Consumo	Alto	Alto	Medio	Medio

Il bus

Nel modello di Von Neumann il bus è l'unità che interconnette le altre unità, come una strada percorsa dai dati per andare da un componente a un altro. È un fascio ordinato di linee, ciascuna delle quali può assumere il significato di un bit. I moduli del processore, della memoria e dell'I/O si «affacciano» sul bus e possono impostare, prelevare o modificare i dati presenti sulle linee. Gran parte del lavoro di un calcolatore è il trasferimento di bit tra i moduli.

Durante un trasferimento il processore è il soggetto **master**, mentre memoria e I/O sono soggetti **slave**.

Un'operazione che porta un dato dal processore alla memoria è di **scrittura** (*write*); nel verso opposto è di **lettura** (*read*).

COMPONENTI DEL BUS

BUS	NOME	FUNZIONE
ABus	Bus degli indirizzi (<i>address bus</i>)	Specifica la locazione di memoria o di I/O coinvolta nel trasferimento.
DBus	Bus dei dati (<i>data bus</i>)	Trasporta i dati; specifica la quantità di informazione trasferibile in una singola operazione.
CBus	Bus di controllo (<i>control bus</i>)	Specifica il verso (lettura o scrittura), sincronizza il trasferimento e verifica che il bus sia libero.

Il bus possiede un orologio costante, il **clock di bus**, che scandisce le operazioni e si esprime in MHz. Il clock di bus è molto inferiore a quello del processore: la CPU lavora in GHz, mentre anche il miglior bus si ferma a qualche centinaio di MHz.

LINEE DI CONTROLLO

- **Linea I/O memoria:** se vale 1 il trasferimento riguarda processore e I/O; se vale 0 riguarda processore e memoria.
- **Linea R/W:** se vale 1 si tratta di una lettura, se vale 0 di una scrittura.
- **Linea wait:** indica se il trasferimento è stato completato (1) oppure è ancora in corso (0), nel qual caso il bus è occupato e non può accettare altri dati.

DIMENSIONE DI ABUS E DBUS

Il numero di linee di ABus e DBus non coincide necessariamente: dipende dalle caratteristiche del processore e di norma è una potenza di due (in alcuni processori si trovano però valori come 20, 36 o 80). La dimensione dell'**ABus** determina la quantità di memoria raggiungibile dai programmi (lo spazio di indirizzamento, 2^n con n numero di linee). La dimensione del **DBus** determina il **grado di parallelismo** del processore, cioè la massima quantità di dati trasferibile in un unico trasferimento di bus.

ARCHITETTURA	ABUS	DBus
32 bit	32 linee: $2^{32} = 4.294.967.296$ locazioni, circa 4 GB indirizzabili	32 linee: 32 bit (4 byte) per operazione
64 bit	64 linee: 2^{64} locazioni, circa 16 exabyte	64 linee: 64 bit (8 byte) per operazione

Il numero di linee di ABus e DBus non è sempre legato strettamente ai bit dell'architettura. Un sistema a 64 bit può avere ad esempio un ABus a 48 linee, in grado di indirizzare fino a 256 TB di memoria fisica, mantenendo il DBus a 64 linee. La scelta dipende da efficienza, costo e requisiti del sistema.

CHIPSET, NORTHBRIDGE E SOUTHBRIDGE

Il bus non è facilmente individuabile sulla scheda madre: l'insieme di circuiti, linee e chip prende il nome complessivo di **chipset**. Esso realizza il bus interconnettendo due aree distinte: il **NorthBridge** (processore e memoria) e il **SouthBridge** (processore e I/O).

TECNOLOGIE DI BUS

STANDARD	CARATTERISTICHE
ISA	Il più vecchio: DBus a 8 bit, clock a 8,33 MHz.
EISA	DBus a 16 bit, clock a 8,33 MHz.
PCI	Peripheral Component Interconnect: DBus a 32 bit, clock a 33 MHz, 133 MB/s.
PCI X, PCI Express	Evoluzione più recente: 32 o 64 bit, 66 MHz, da 528 a 1035 MB/s.
VESA, AGP	Bus dedicati al trasporto di dati dalla scheda grafica alla CPU, in disuso a favore del PCI.

Esistono anche altri bus meno fortunati, tra cui Microchannel, SCSI e PCMCIA (quest'ultimo dedicato ai dispositivi portatili).

Input e output

Il sistema di **I/O** ha il compito di acquisire dati e programmi e, una volta elaborati, di rappresentarli in varie forme: video, stampa, salvataggio su memorie secondarie. Può essere immaginato come un contenitore di celle analogo alla memoria: ogni periferica possiede un proprio intervallo di indirizzi riservato, i cui registri sono chiamati **porte di I/O**. Lo spazio degli indirizzi di I/O è molto più ridotto di quello della memoria. Alcuni dispositivi che richiedono grandi spazi sono invece mappati direttamente in memoria.

SEGNALI DI INTERRUZIONE

Per sincronizzare le proprie attività con i limiti del bus, il sistema di I/O usa linee dedicate dette **linee di interruzione**. Nel CBus sono presenti due segnali tipici:

- **INTR**: richiesta di interruzione;
- **INTA**: completamento dell'interruzione.

Con questi segnali una periferica chiede al processore di sospendere temporaneamente l'esecuzione di un altro programma, a priorità inferiore, per eseguire la propria **ISR** (*interrupt service routine*). Il meccanismo è essenziale per l'I/O **asincrono**, che può intervenire senza preavviso in ogni momento (ad esempio il movimento del mouse). La periferica invia il segnale, l'*interrupt controller* lo interpreta, smista le richieste e comunica alla CPU quale processo sospendere.

DMA

I calcolatori moderni prevedono modalità di trasferimento che evitano di occupare il processore: i valori di I/O sono veicolati direttamente verso e dalla memoria. La tecnologia si chiama **DMA** (*direct memory access*) o *bus mastering*.

DEFINIZIONE

La **scheda controller della periferica** è la circuiteria che si affaccia sul bus e rende disponibili i propri indirizzi di I/O alla sincronizzazione con i trasferimenti del CBus. Oggi è in genere integrata nel chipset della scheda madre (ad esempio per tastiera, scheda di rete e scheda video), così come l'*interrupt controller* e il DMA.

PRIMA E DOPO IL PCI

Prima del PCI ogni scheda controller doveva essere configurata a mano, con ponticelli hardware, per specificare indirizzi di I/O, numero di interruzione e canale DMA. Con il PCI si ha una tecnologia **plug and play**: BIOS, sistema operativo e firmware della scheda stabiliscono all'avvio (o all'installazione a caldo di un dispositivo) canali e linee di ogni controller, eliminando conflitti e configurazioni manuali.

PERIFERICHE ESTERNE E STANDARD DI CONNESSIONE

Le periferiche opzionali, installate dall'utente, sono dette **esterne** e si collegano al bus con connessioni standard e connettori appositi (lo slot PCI, ad esempio, per le schede di rete). Se un componente non è *hot swappable* va sostituito a calcolatore spento.

STANDARD	NOTE
USB (universal serial bus)	Molto diffuso per l'ampia larghezza di banda (480 Mbit/s per USB 2.0) e per l'hot swap. Consente connessioni multiple in catena tramite hub.
Firewire (IEEE 1394)	Larghezza di banda di 400 Mbit/s e hot swap. Consente connessioni multiple in catena, direttamente in modo passante.
Ethernet (802.x)	Connessione di rete.
RS232, parallela Centronics, PS/2	Standard storici in via di dismissione, sostituiti da standard più moderni (tipicamente convertitori USB). Anche le connessioni IDE ed EIDE (ATA e ATA2) per hard disk, e ATAPI su IDE per i DVD, sono in via di superamento.

Alcuni standard trasportano sullo stesso cavo anche una certa quantità di corrente, rendendo alcuni dispositivi autonomi dall'alimentazione.

Il processore

Struttura della CPU, ciclo di esecuzione delle istruzioni e le architetture che ne determinano le prestazioni: CISC, RISC, cache, prefetch e pipeline.

Struttura del processore

DEFINIZIONE

Il **processore** (CPU) è un singolo circuito integrato in grado di effettuare operazioni decisionali, di calcolo o di elaborazione dell'informazione.

UNITÀ	FUNZIONE
CU, unità di controllo	Si affaccia direttamente sul bus e lo arbitra, impostando valori e stati sulle linee di ABus, DBus e CBus. Prende le decisioni.
ALU, unità aritmetico logica	Unità esecutiva del processore: contiene microprogrammi cablati (hardware) scritti in «microcodice», con le relative microistruzioni.
Registri	Memorie di lavoro proprie del processore: contengono i dati letti dalla CU sul bus, predisponendoli per l'esecuzione nell'ALU, oppure i risultati dell'ALU in attesa di tornare alla CU e quindi al bus.

INSTRUCTION SET

Ogni processore è progettato con un insieme di istruzioni chiamato **ISA** (*instruction set architecture*); per ciascuna istruzione è implementato nell'ALU un preciso microprogramma. Ogni istruzione è contraddistinta da un numero specifico, l'**OP code** (codice operativo). Ogni istruzione con un OP code richiede un numero preciso di parametri, detti **operandi**, che insieme all'OP code determinano la lunghezza dell'istruzione in byte. A ogni OP code è associata una breve descrizione in linguaggio naturale che ne ricorda la funzione, detta **codice mnemonico**.

Esiste un registro speciale, il **PC** (*program counter*), che si incrementa automaticamente della lunghezza dell'istruzione appena eseguita.

DATA PATH

ALU, registri e molti dei bus formano il **data path**. I registri di input alimentano l'ALU e il risultato è scritto in un registro di output. Il ciclo del data path consiste nel passaggio di due operandi attraverso l'ALU e nella memorizzazione del risultato nei registri. Le istruzioni del data path sono di due tipi: registro memoria (trasferisce un dato tra memoria e registri) e registro registro (trasferisce un dato da un registro a un altro).

Le istruzioni dell'ISA si implementano in due modi: via hardware, fisicamente dentro la CPU, oppure via software, tramite un interprete.

Il ciclo di esecuzione

Ogni processore ripete la stessa sequenza di passi per tutto il tempo in cui la macchina è alimentata.

FASE	DESCRIZIONE
Fetch	La CU pone sull'ABus il valore del registro PC, imposta la lettura dalla memoria e carica l'OP code.
Decode	Dall'OP code la CU determina la lunghezza dell'istruzione, cioè il numero di parametri necessari, e avvia il caricamento degli operandi (<i>operand fetch</i>), che si trovano in ordine negli indirizzi adiacenti a quello dell'OP code. Gli operandi caricati sono registrati nei registri.
Execute	Si avvia il microprogramma corrispondente all'OP code, che usa i dati già nei registri. Le operazioni durano un intervallo di tempo fisso, dettato dalla frequenza di clock della CPU (espressa in GHz).
Store	I risultati, se presenti, sono posti nei registri e trasportati dalla CU attraverso il bus verso la memoria o il sistema di I/O.

Il ciclo termina consultando il segnale **INTR**, per capire se il controller di una periferica ha richiesto un'interruzione, cioè la sospensione temporanea dell'esecuzione per servire il codice associato all'interruzione pendente.

TEMPI DI ESECUZIONE

Ogni istruzione dell'ISA ha un proprio OP code, una determinata lunghezza (calcolata dal numero di operandi) e un numero di cicli di bus. Il tempo necessario a completare il microprogramma influisce pesantemente sulla durata dell'istruzione, perché il clock di una CPU è di almeno un ordine di grandezza superiore a quello di un bus.

Lo schema è semplificato: un microprocessore reale contiene, oltre all'ALU, una **FPU** (*floating point unit*) per i calcoli in virgola mobile e varie unità per istruzioni complesse (ad esempio i calcoli vettoriali).

CISC e RISC

CISC

Nei processori **CISC** (*complex instruction set computer*, sostenuti da IBM, VAX e Intel) ogni istruzione dell'ISA è interpretata tramite microprogrammi e microcodici: il processore è detto «a interprete». Ogni istruzione deve essere decodificata, prelevata, eseguita e il suo risultato salvato, con uno svolgimento che coinvolge più cicli. L'insieme di istruzioni è corposo e la dimensione delle istruzioni varia molto in base alla complessità della fase di decode.

- **Pro:** architetture estremamente portabili; il software scritto per un CISC è facilmente portabile anche sui processori di nuova generazione. Sono ideali per la programmazione in assembly.
- **Contro:** la fase di decode può essere molto lunga. Se il processore impiega un tempo Δp per elaborare un pacchetto di istruzioni e il bus, con clock inferiore e minor capienza, impiega Δb per trasportarle, con $\Delta p > \Delta b$ si ha un *bottleneck*: la CPU non può elaborare finché il bus non si libera.

RISC

Le architetture **RISC** (*reduced instruction set computer*, nate negli anni Ottanta, ad esempio DEC Alpha) sono opposte: l'insieme di istruzioni è ridotto e le istruzioni hanno sempre lunghezza costante, con un numero fisso di operandi. La fase di decode è molto più breve, senza microprogrammi da eseguire, e ogni istruzione è eseguita direttamente in hardware in un numero limitato di cicli di clock.

- **Pro:** velocità di elaborazione nettamente superiore (di almeno un ordine di grandezza). Un'istruzione CISC a più passi equivale a numerose istruzioni RISC con data path singolo.
- **Contro:** i programmi sono più lunghi dell'analogo programma CISC, meno portabili, con compilatori più complessi e difficili da programmare in assembly per la mancanza di istruzioni ISA di alto livello.

CRISC

Nessuna delle due ISA è l'architettura «totale»: molte CPU di ultima generazione scelgono un'opzione ibrida, la **CRISC** (*complex reduced instruction set computer*), basata su un CISC dotato di sottosistemi interni di tipo RISC, dedicati alla computazione delle istruzioni semplici e più comuni dell'ISA adottata. Oggi anche il Pentium è ibridato con tecnologia RISC.

CARATTERISTICA	CISC	RISC
Istruzioni	Molte e complesse	Poche e semplici
Lunghezza	Variabile	Costante, numero di operandi fisso
Esecuzione	Interpretata con microprogrammi	Diretta in hardware
Programmi	Brevi e portabili	Lunghi e meno portabili
Fase di decode	Lunga	Breve

Il modello SISD/CISC di Von Neumann ha due limiti strutturali. Eseguire una sola istruzione su un solo dato per unità di tempo (**SISD**, *single instruction single data*) è inaccettabile, e molte soluzioni mirano a parallelizzare l'esecuzione. Il ritardo di accesso alla memoria (bus in MHz, CPU in GHz) penalizza il modello CISC, con una fase di decode lenta e complessa: molte soluzioni mirano a fornire istruzioni e operandi alla stessa velocità con cui il processore li esegue.

Cache

L'accesso al bus è un problema non trascurabile. La **cache** è una memoria «tampone» tra CPU e bus: man mano che la CPU legge dalla memoria, gli indirizzi prossimi a quello in esecuzione sono caricati nei registri della cache, chiamati **linee di cache**. Funziona per il **principio di località spazio temporale**: in un breve intervallo di tempo un programma usa solo una piccola parte del proprio spazio di indirizzamento, composta da indirizzi numericamente vicini. La cache, realizzata in SRAM e molto più veloce della DRAM, può quindi fornire nell'immediato futuro (per esempio la prossima istruzione) i valori richiesti senza passare per il bus di sistema.

A ogni operazione di lettura l'informazione è cercata prima nella cache. Se è presente si ha un **hit** e non serve passare dal bus. Se non è presente si ha un **miss**: si accede alla memoria e, oltre all'informazione richiesta, si carica una nuova linea di cache, sovrascrivendo la linea di cache meno usata di recente.

Nei calcolatori moderni esistono tre livelli di cache: il primo interno al processore, il secondo collegato al processore, il terzo sulla scheda madre. Dal primo al terzo la memoria è più capiente e più lenta.

Prefetch

Un altro modo per aumentare il parallelismo è caricare nella CPU più istruzioni di quelle richieste. I processori sono dotati di una **coda di prefetch**, un buffer interno che memorizza i successivi 6 o 8 byte rispetto a quello appena letto: con un singolo accesso alla memoria aumenta il numero di valori che arrivano alla CPU senza ripassare dal bus.

Il prefetch ha difficoltà quando l'istruzione corrente è un salto distante dall'istruzione successiva: la coda viene persa e con essa la cache caricata.

Pipeline e superscalarità: panoramica

In un elaboratore un'istruzione passa per cinque passaggi fondamentali:

STADIO	NOME	COMPITO
IF	Instruction fetch	Lettura dell'istruzione dalla memoria.
ID	Instruction decode	Decodifica dell'istruzione e lettura degli operandi dai registri.
EX	Execution	Esecuzione dell'istruzione.
MEM	Memory	Attivazione della memoria (solo per certe istruzioni).
WB	Write back	Scrittura del risultato nel registro risultato.

Alla coda di prefetch si è affiancata la **pipeline**, che applica il concetto di catena di montaggio: l'istruzione successiva si avvia subito dopo che la precedente è avanzata nel data path, così mentre un'istruzione è in decode la successiva può già essere in fetch. Una pipeline a 5 stadi trasporta fino a 5 istruzioni insieme. Poiché lo stadio di esecuzione è il più lento, le CPU hanno più ALU: i processori con più unità aritmetico logiche sono detti **superscalari** (approfonditi nel capitolo sulla pipeline).

Salti e dipendenze: le soluzioni del processore

Pipeline, superscalarità e prefetch sono vanificate da due situazioni: le istruzioni di salto e le dipendenze di dato tra istruzioni. Se l'istruzione corrente è un salto (*branch*) distante dall'istruzione già in pipeline, la pipeline viene persa del tutto e con essa la cache caricata. Se l'istruzione successiva richiede come operando il risultato finale della precedente, la pipeline deve interrompersi: con Istruzione 1: $A = B + C$ e Istruzione 2: $D = A + 1$, la pipeline dell'istruzione 2 si ferma allo stadio di operand fetch finché A non è disponibile.

ESECUZIONE PREDICATIVA E SPECULATIVA

Sono implementate in moduli detti **unità di previsione dei salti** (*dynamic branch prediction*): con vari algoritmi, che usano tabelle simili a quelle della cache, cercano di capire se un salto avverrà. La previsione non è deterministica ma statistica, basata sullo storico, perché l'esito si calcola solo durante l'esecuzione. Ad esempio alcuni criteri considerano sempre «presi» (*taken*) i salti all'indietro, tipici dei cicli. L'esecuzione predicativa è detta anche **speculativa**, perché elabora anche codice che potrebbe non essere mai utilizzato. Se la previsione è sbagliata, le istruzioni eseguite inutilmente vengono scartate e lo stato della macchina viene ripristinato.

ESECUZIONE FUORI ORDINE

Con l'**esecuzione fuori ordine** (*out of order*) si evita di svuotare la pipeline per una dipendenza: invece di buttarla e attendere l'operando mancante, il processore salta l'istruzione e prosegue con istruzioni «future», che in teoria andrebbero eseguite dopo. Resta bloccato un solo stadio fino all'arrivo dell'operando e, risolta la dipendenza, altre istruzioni sono già state eseguite, con una risalita delle prestazioni. Le istruzioni future si eseguono solo se non hanno a loro volta dipendenze con istruzioni in corso; appena le istruzioni con dipendenze terminano, il processore torna all'esecuzione in ordine (*in order*). La condizione più critica si ha con un interrupt durante la fase fuori ordine: lo stato del sistema potrebbe non essere coerente, quindi il processore lo ripristina ritirando «in ordine» tutte le fasi fuori ordine.

La dipendenza più comune e complicata è la **RAW** (*read after write*), quella dell'esempio precedente. Per riordinare il flusso dopo aver saltato un'istruzione, i processori usano registri d'appoggio interni e invisibili al programmatore, su cui memorizzano i calcoli temporanei. Al riordinamento rinominano i registri interni nei nomi dei registri effettivi, risparmiando il tempo di trasferimento dei valori: è il **register renaming**.

Queste tecniche cercano di realizzare, anche se non esplicitamente, un modello di esecuzione parallela detto **VLIW** (*very long instruction word*), in cui, oltre alla parallelizzazione in hardware, si presuppone che il codice generato dai compilatori sia «pre cucinato» per essere parallelizzato in modo ottimale dalle CPU.

Rappresentazione dell'informazione

Come un calcolatore, che conosce solo due stati, rappresenta numeri naturali, interi relativi e numeri reali.

Sistemi di numerazione posizionali

Nella notazione posizionale decimale la quantità rappresentata da una cifra dipende dalla sua posizione. Il sistema decimale (arabo) è economico e flessibile: con soli 10 simboli permette di rappresentare qualunque quantità. Un esempio di sistema non posizionale è la numerazione romana, che è additiva.

DEFINIZIONE

Dato un intero B detto **base** e un insieme di B simboli $S_B = \{s_0, \dots, s_{B \text{ meno } 1}\}$ che rappresentano le quantità $0, 1, \dots, B \text{ meno } 1$, un numero a n cifre $p_{n \text{ meno } 1} \dots p_1 p_0$ è rappresentabile come **somma di potenze della base**:

$$\text{valore} = \sum p_i \cdot B^i \quad \text{con } i \text{ da } 0 \text{ a } n \text{ meno } 1$$

CONVERSIONE DA BASE 10 A BASE B

Si usa la tecnica delle **divisioni successive**:

1. sia N il numero in base 10 da convertire;
2. si calcola la divisione intera N / B e si mette da parte il resto R ;
3. se il quoziente è maggiore di 0 si ripete il passo 2 sul quoziente;
4. quando il quoziente è 0 si riportano i resti letti dall'ultimo al primo: essi formano il numero in base B .

Esempio: 13 in base 2. $13 / 2 = 6$ resto 1; $6 / 2 = 3$ resto 0; $3 / 2 = 1$ resto 1; $1 / 2 = 0$ resto 1. Leggendo i resti dall'ultimo al primo si ottiene 1101. In base 5: $13 / 5 = 2$ resto 3; $2 / 5 = 0$ resto 2; si legge 23, cioè $(13)_{10} = (23)_5$.

BASI DI USO COMUNE

Si può pensare a un abaco in cui ogni asticella contiene al più $B \text{ meno } 1$ palline: superata la capacità, le palline si raggruppano e passano come una sola all'asticella successiva. Lo stesso numero si scrive con regole diverse nei vari sistemi: con 12 palline un abaco a base 10 mostra 12, uno a base 8 mostra 14 (un otetto più 4 unità).

BASE	SISTEMA	SIMBOLI	IL NUMERO DECIMALE 12
2	Binario	{0, 1}	$(1100)_2$
8	Ottale	{0, 1, 2, 3, 4, 5, 6, 7}	$(14)_8$
10	Decimale	{0, 1, ..., 9}	$(12)_{10}$
16	Esadecimale	{0, ..., 9, A, B, C, D, E, F}	$(C)_{16}$

NUMERO DI VALORI RAPPRESENTABILI

Con n cifre in base B si rappresentano B^n valori distinti, da 0 a $B^n \text{ meno } 1$ (con una cifra sono B , con due $B \cdot B$). In binario con n bit si rappresentano i naturali nell'intervallo $[0, 2^n)$. Esempio: $(54)_{10} = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = (110110)_2$.

Un sistema additivo come la numerazione romana ha bisogno di nuovi simboli al crescere della quantità: ad esempio $\text{CCLXXVIII} = 2 \cdot 100 + 50 + 2 \cdot 10 + 5 + 3 \cdot 1 = 278$.

Dall'elettricità all'aritmetica

Il calcolatore è una macchina composta da circuiti elettrici che hanno solo due stati, acceso e spento. Realizzare circuiti basati su questo paradigma è più semplice, perché si rappresentano due livelli di tensione. I concetti di spento e acceso si rappresentano con i numeri 0 e 1; definita questa tipologia di informazione, si possono stabilire regole non ambigue per rappresentare l'informazione come **sequenze binarie**.

ADDIZIONE DI NUMERI NATURALI

Per sommare numeri binari a un bit si seguono le regole $0+0=0$, $1+0=1$, $0+1=1$, $1+1=10$ (somma 0 con riporto 1). Il riporto in uscita di una cifra diventa riporto in ingresso della cifra successiva.

SOMMA MODULARE

La funzione modulo è definita come $A \bmod n = \text{resto di } (A / n)$. La somma modulare $(A + B) \bmod n = \text{resto di } ((A + B) / n)$ assume sempre valori compresi tra 0 e n meno 1.

Interi relativi

Per rappresentare gli interi relativi si usa il bit più a sinistra come **bit di segno**: 0 per i numeri positivi, 1 per i negativi. Per il valore assoluto esistono tre soluzioni.

RAPPRESENTAZIONE	COME SI OTTIENE IL NUMERO NEGATIVO
Segno e valore assoluto	Si cambia solo il bit di segno, lasciando invariato il valore assoluto.
Complemento a 1	Si invertono tutti i bit del numero positivo.
Complemento a 2	Si inverte ogni bit del numero positivo e si somma 1 al risultato.

Esempio in complemento a 2 su 4 bit: $+5 = 0101$; invertendo i bit si ha 1010 e sommando 1 si ottiene 1011, che rappresenta meno 5.

L'addizione di due numeri in complemento a 2 corrisponde alla somma di due naturali senza contare il riporto in uscita; la sottrazione corrisponde a un'addizione in cui il sottraendo è complementato a 2, purché non avvenga trabocco.

TRABOCCO (OVERFLOW)

In somme e sottrazioni in complemento a 2 il risultato è corretto se è compreso nell'intervallo da meno 2^{n-1} a 2^{n-1} meno 1. Se così non è si verifica il **trabocco** (*overflow*), cioè il risultato supera la capacità di rappresentazione del registro. Il trabocco avviene quando:

1. i due addendi sono concordi di segno;
2. il bit di segno della somma è diverso da quello degli addendi.

ESTENSIONE E RIDUZIONE DEL SEGNO

- **Estensione del segno**: si replica a sinistra il bit del segno tante volte quanto occorre.
- **Riduzione del segno**: si rimuove il bit più a sinistra tante volte quanto occorre, purché il bit successivo abbia lo stesso valore.

Numeri in virgola mobile

Un numero binario in virgola mobile è rappresentato da tre componenti:

- un **segno** s ;
- una **mantissa** m , cioè i bit significativi escluso il bit più significativo (che vale sempre 1 e non si memorizza);
- un **esponente** e con segno, in base due.

$$\text{valore} = \pm 1, m \times 2^e$$

Lo standard **IEEE 754** definisce due formati. L'esponente è memorizzato in **rappresentazione in eccesso**: si memorizza e' e l'esponente effettivo è $e = e' \text{ meno l'eccesso}$.

	PRECISIONE SINGOLA (32 BIT)	PRECISIONE DOPPIA (64 BIT)
Segno	1 bit (0 positivo, 1 negativo)	1 bit
Esponente e'	8 bit, eccesso 127, con $0 \leq e' \leq 255$	11 bit, eccesso 1023, con $0 \leq e' \leq 2047$
Mantissa m	23 bit, rappresentazione naturale	52 bit, rappresentazione naturale
Valori speciali	$e' = 0$ ed $e' = 255$	$e' = 0$ ed $e' = 2047$
Intervallo di e	da meno 126 a 127	da meno 1022 a 1023
Fattore di scala	da 2^{-126} a 2^{127}	da 2^{-1022} a 2^{1023}

VALORI SPECIALI

ESPONENTE E'	MANTISSA M	SIGNIFICATO
0	0	Zero esatto
255 (2047 in doppia precisione)	0	Infinito
0	diversa da 0	Forma non normale (denormalizzato)
255 (2047 in doppia precisione)	diversa da 0	NaN (Not a Number)

OPERAZIONI ARITMETICHE IN VIRGOLA MOBILE

OPERAZIONE	ALGORITMO
Addizione e sottrazione	1. Si sceglie il numero con esponente minore e si sposta a destra la sua mantissa di un numero di posizioni pari alla differenza degli esponenti. 2. L'esponente del risultato è l'esponente più grande. 3. Si sommano i fattori interi (1 seguito dalla mantissa) mantenendo il segno. 4. Si normalizza il risultato.
Moltiplicazione	1. Si sommano gli esponenti e si sottrae 127 dalla somma, ottenendo l'esponente provvisorio del prodotto. 2. Si moltiplicano i fattori interi, ottenendo segno e fattore intero. 3. Si normalizza se necessario.
Divisione	1. Si sottrae l'esponente del divisore da quello del dividendo e si aggiunge 127, ottenendo l'esponente provvisorio del rapporto. 2. Si calcola il quoziente dei fattori interi, ottenendo segno e fattore intero provvisori. 3. Si normalizza se necessario.

Linguaggio macchina e assembly

Le istruzioni che la CPU è in grado di eseguire, i modi di indirizzamento, la pila e i sottoprogrammi.

ISA e linguaggi

La CPU esegue un insieme di operazioni di base chiamate **istruzioni macchina**. L'insieme delle istruzioni eseguibili e le loro modalità di utilizzo costituiscono l'**ISA**. Ogni CPU ha il proprio ISA. Il **linguaggio macchina** definisce le istruzioni con un alfabeto binario (0 e 1); il **linguaggio assembler** ne è una rappresentazione simbolica leggibile.

Il programmatore scrive in linguaggio assembler e l'**assembler** lo traduce in sequenze binarie. Nei linguaggi ad alto livello il programma è scritto in codice ad alto livello e il **compilatore** lo traduce in linguaggio assembler, che a sua volta è tradotto dall'assembler.

UNITÀ DI MISURA

Un **byte** è una sequenza di 8 bit. I byte si raggruppano in blocchi il cui numero di elementi è espresso con potenze di 2:

UNITÀ	SIMBOLO	VALORE	ORDINE DI GRANDEZZA
Kilobyte	KB	$2^{10} = 1024$	circa 10^3
Megabyte	MB	$2^{20} = 1024 \cdot 1024$	circa 10^6
Gigabyte	GB	2^{30}	circa 10^9
Terabyte	TB	2^{40}	circa 10^{12}

PAROLA E ORGANIZZAZIONE DELLA MEMORIA

Il calcolatore non lavora su singoli bit ma su gruppi di bit chiamati **parole**, di lunghezza da 8 a 64 bit (sempre potenze di 2), in base all'architettura. I dati occupano da un singolo byte a diverse parole; le istruzioni una o più parole. L'informazione è immagazzinata come vettore di parole, ciascuna associata a un indirizzo binario univoco: un indirizzo di m bit permette di rappresentare 2^m indirizzi. Parole consecutive hanno indirizzi consecutivi; gli indirizzi delle parole sono multipli della loro lunghezza in byte.

ORDINAMENTO DEI BYTE

L'unità minima di informazione è il byte. Esistono due schemi di indirizzamento dei byte all'interno di una parola:

- **Big endian** (crescente): l'indirizzo aumenta al diminuire del peso aritmetico del byte;
- **Little endian** (decrescente): l'indirizzo aumenta all'aumentare del peso aritmetico del byte.

RISC E CISC A CONFRONTO

RISC	CISC
Modi di indirizzamento semplici	Modi di indirizzamento complessi
Meno istruzioni, tutte di una singola parola	Molte istruzioni complesse, su più parole
Operazioni aritmetiche e logiche solo su registri	Operandi sia in registri sia in locazioni di memoria
Nessun trasferimento diretto tra due locazioni di memoria	Trasferimenti diretti tra locazioni tramite l'istruzione Move
Possibile elaborazione a stadi (pipeline)	Programmi di dimensioni ridotte
Programmi di dimensioni maggiori	

Un insieme di istruzioni deve comprendere almeno quattro tipologie: trasferimento di dati tra memoria e registri, operazioni aritmetiche e logiche sui dati, operazioni di controllo dell'ordine di esecuzione e trasferimento di dati tra unità di I/O e registri.

Registri e istruzioni di base

I registri sono identificati dal nome: registri generici (R0, R1, ..., Rn), registri speciali del processore (PC, IR, ecc.) e registri di I/O. Le locazioni di memoria sono identificate dall'indirizzo, espresso come costante numerica o come costante simbolica dichiarata in precedenza.

ISTRUZIONE	FUNZIONE
Load destinazione, sorgente	Carica un dato dalla memoria in un registro. La destinazione è un registro, la sorgente una locazione di memoria.
Store sorgente, destinazione	Salva in memoria un dato presente in un registro. La sorgente è un registro, la destinazione una locazione di memoria.
Add destinazione, sorgente1, sorgente2	Somma il contenuto di due registri; il risultato è scritto nel registro destinazione. Gli addendi sono nomi di registri o valori.
Subtract destinazione, sorgente1, sorgente2	Sottrae il contenuto di due registri; il risultato è scritto nel registro destinazione.
Branch_if condizione destinazione	Salto condizionato: salta all'istruzione indicata se la condizione è vera. La condizione confronta valori dei registri (tra parentesi quadre) o valori espliciti. Esempio: Branch_if [R2] > 0 CICLO.

MODI DI INDIRIZZAMENTO

MODO	DESCRIZIONE
Immediato	L'operando è dato in maniera esplicita nell'istruzione, e la costante è preceduta dal simbolo #. Esempio: Add R4, R6, #200 somma 200 al contenuto di R6 e scrive il risultato in R4.
Di registro	Il nome di un registro contiene l'operando o il risultato.
Assoluto (diretto)	L'indirizzo della parola che contiene operando o risultato è dato nell'istruzione stessa. Nei RISC esiste un limite al numero di bit dell'indirizzo assoluto (un'istruzione occupa una parola): in una CPU a 32 bit l'indirizzo assoluto è di 16 bit. Esempio: Load R2, NUM1.
Indiretto da registro	Il nome di un registro contiene l'indirizzo di memoria dell'operando o del risultato. Si indica con il nome del registro tra parentesi tonde. Si usa per riutilizzare più volte la stessa istruzione cambiando gli operandi.
Con indice e spiazzamento	L'indirizzo effettivo si ottiene sommando una costante (lo spiazzamento) al contenuto del registro. Si scrive X(Ri), con X spiazzamento e Ri registro che contiene l'indirizzo; è utile per gestire vettori e liste.
Con base e indice	L'indirizzo effettivo si ottiene sommando il contenuto di due registri, indicato come (Ri, Rj).

MODI DI INDIRIZZAMENTO PROPRI DEI CISC

- **Autoincremento:** l'indirizzo dell'operando è in un registro; alla fine dell'istruzione il contenuto del registro è incrementato di un'unità. Sintassi (R)+, usabile per realizzare una pop. Esempio: Move ELEMENTO, (SP)+.
- **Autodecremento:** l'indirizzo dell'operando è in un registro; prima di eseguire l'istruzione il contenuto del registro è decrementato di un'unità. Sintassi -(R), usabile per realizzare una push. Esempio: Move -(SP), NUOVOELEMENTO.
- **Relativo (a PC):** simile al modo con indice e spiazzamento, ma applicato al registro PC, con sintassi X(PC). È utile per rappresentare indirizzi di dimensione ridotta, sempre relativi al PC.

ISTRUZIONI CISC

L'istruzione **Move destinazione, sorgente** trasferisce dati tra registri e indirizzi di memoria. La destinazione può essere un registro o un indirizzo di memoria; la sorgente un registro, un indirizzo di memoria o un valore immediato. In notazione RTN: destinazione ← sorgente. Le istruzioni aritmetiche e logiche dei CISC usano un formato a due indirizzi, con operandi che possono essere registri o locazioni di memoria. Nelle operazioni a due ingressi e un risultato l'operando destinazione serve sia da ingresso sia da risultato: ad esempio Add B, A esegue $B \leftarrow [A] + [B]$.

BIT DI ESITO

I **bit di esito** (o di condizione) sono bit speciali immagazzinati nel **registro di stato**, interno al processore.

BIT	SIGNIFICATO
N (negativo)	1 se il risultato è negativo, 0 se positivo o nullo.
Z (zero)	1 se il risultato è nullo, 0 altrimenti.
V (trabocco)	1 se si è verificato trabocco in complemento a due (overflow), 0 altrimenti.
C (riporto)	1 se si è verificato trabocco in binario naturale (carry), 0 altrimenti.

Direttive dell'assemblatore

Un programma scritto in assembly è detto **programma sorgente**; il codice macchina in forma binaria prodotto dall'assemblatore è il **programma oggetto**. Per produrlo l'assemblatore deve assegnare valori numerici a nomi e simboli e decidere dove collocare in memoria le istruzioni macchina e gli operandi e risultati del programma. Il linguaggio assembleativo contiene quindi, oltre alle istruzioni, le **direttive di assemblatore**: comandi che non sono tradotti nel programma oggetto ma forniscono informazioni all'assemblatore.

DIRETTIVA	FUNZIONE
NOME EQU Valore_numerico	Dichiarazione di eguaglianza: associa un valore numerico a un nome usato nel sorgente; l'assemblatore sostituisce ogni occorrenza del nome con il valore.
ORIGIN Indirizzo_di_memoria	Indica l'indirizzo di partenza da cui inserire istruzioni e dati definiti nelle righe seguenti, ai quali sono assegnati indirizzi a partire da quello indicato.
RESERVE Spazio_in_byte	Riserva uno spazio di memoria espresso in byte; la locazione riservata non viene inizializzata.
DATAWORD Contenuto_da_assegnare	Riserva una parola di memoria e la inizializza con il contenuto indicato.

Una linea di codice assembleativo presenta, nella maggior parte dei casi, quattro campi: **Etichetta** (nome associato all'indirizzo della parola di memoria assegnata all'istruzione o al blocco riservato), **Operazione** (codice operativo o direttiva), **Operandi** (informazioni di indirizzamento per accedere agli operandi) e **Commento** (ignorato dall'assemblatore). L'etichetta è facoltativa.

I numeri si denotano in diversi formati: binario con il prefisso %, decimale senza prefisso, esadecimale con il prefisso 0x. Lo stesso operando immediato si scrive Add R2, R3, #%01011101 (binario), Add R2, R3, #93 (decimale) oppure Add R2, R3, #0x5D (esadecimale).

Pila e sottoprogrammi

DEFINIZIONE

La **pila** (*stack*) è una lista di elementi (parole) immaginati uno sull'altro. Gli elementi si aggiungono e si prelevano dalla cima, secondo il paradigma **LIFO** (*last in first out*): l'ultimo elemento inserito è il primo a essere prelevato. Il registro **SP** (*stack pointer*) punta sempre alla cima; gli indirizzi degli elementi sono in ordine decrescente.

PUSH E POP NEI RISC

Push (aggiungere un elemento in cima) richiede due istruzioni: diminuire SP di una parola e scrivere il valore nella parola puntata da SP. **Pop** (prelevare un elemento) ne richiede due: copiare in un registro il valore puntato da SP e aumentare SP di una parola.

```
push:  Subtract SP, SP, #4
       Store   Rj, (SP)

pop:   Load   Rj, (SP)
       Add     SP, SP, #4
```

SOTTOPROGRAMMI

Un **sottoprogramma** (o *routine*) è una lista di istruzioni che esegue un compito specifico e può essere richiamata in qualunque momento durante l'esecuzione di un programma. Si richiama con l'istruzione **Call INDIRIZZO**; per tornare all'istruzione successiva alla call nel programma principale si usa l'istruzione **Return**. Il **link register** è un registro speciale in cui si memorizza l'indirizzo di rientro durante l'esecuzione. La chiamata esegue due passi: salva il contenuto del PC nel link register e salta all'indirizzo di destinazione. Il Return salta all'indirizzo di rientro copiando il contenuto del link register nel PC.

Una routine ha spesso bisogno di parametri di ingresso e di restituire un risultato al chiamante. Esistono due tecniche di passaggio dei parametri: tramite i registri della CPU (numero limitato dal numero di registri) oppure tramite la pila (numero virtualmente illimitato).

Nella pila esiste un blocco di memoria riservato al sottoprogramma, l'**area di attivazione**. Il registro **FP** (*frame pointer*) punta allo *stack frame* del sottoprogramma in esecuzione, che contiene parametri, l'FP del programma chiamante, le variabili locali e i valori dei registri salvati. FP punta alla parola in cui è memorizzato l'FP del programma chiamante.

ANNIDAMENTO

Con sottoprogrammi annidati, prima di chiamare una seconda routine occorre salvare il contenuto del link register per recuperarlo in seguito; gli indirizzi di rientro si salvano nella pila, nell'area di attivazione dei programmi chiamanti. Esistono anche architetture che non usano il link register e salvano l'indirizzo di rientro solo nella pila: strategia detta **pila a modo implicito**.

Codifica delle istruzioni e operazioni

Nei RISC le istruzioni macchina sono codificate in una parola di n bit; in base al tipo di istruzione e di operandi esistono formati diversi: con operandi in registri, con operando immediato, per chiamata. Con una CPU a 32 bit e istruzioni a tre indirizzi di registro si assegnano 5 bit per ogni operando: i 17 bit meno significativi contengono il codice operativo e i 15 bit più significativi gli indirizzi dei tre registri (Rsrc1 nei bit da 31 a 27, Rsrc2 da 26 a 22, Rdst da 21 a 17).

FORMATO	CAMPI (CPU A 32 BIT)
Operandi in registri	Rsrc1, Rsrc2 e Rdst da 5 bit ciascuno (i 15 bit più significativi) e codice operativo di 17 bit. Tipico delle istruzioni aritmetiche e logiche.
Operando immediato	Codice operativo di 6 bit (i meno significativi), valore immediato, indirizzo di memoria o spiazzamento di 16 bit e indirizzi dei registri nei 10 bit più significativi. Tipico di aritmetiche e logiche con immediato, Load, Store e salti.
Chiamata	Codice operativo di 6 bit e indirizzo della prima istruzione del sottoprogramma di 26 bit.

La **notazione a trasferimento di registro** (RTN) descrive formalmente il trasferimento di informazioni tra parole di memoria e registri. Il simbolo $\leftarrow$ indica il trasferimento; a sinistra c'è un indirizzo di memoria o un nome di registro, a destra un valore o un'espressione; il contenuto di un registro o di una parola è indicato tra parentesi quadre. Esempio: $C \leftarrow [A] + [B]$.

OPERAZIONE	DESCRIZIONE
AND, OR	Istruzioni logiche che agiscono bit a bit (<i>bitwise</i>), con lo stesso formato delle aritmetiche: Rdst, Rsrc1, Rsrc2, oppure con valore immediato di 16 bit esteso a 32 con zeri. Esempio: se R2 contiene 4 caratteri ASCII da 8 bit e si vuole sapere se il carattere più a destra è la lettera Z (0x5A), si usa <code>And R2, R2, #0xFF</code> per selezionare quel byte, poi <code>Move R3, #0x5A</code> e <code>Branch_if_[R2]=[R3] TROVATOZ</code> .

OPERAZIONE	DESCRIZIONE
Scorrimento logico	Fa scorrere i bit di un registro a destra o a sinistra di n posizioni. I bit in uscita si perdono, salvo l'ultimo che finisce nel bit di riporto C; le posizioni libere si riempiono con 0. Operandi: registro destinazione, registro sorgente, numero di bit.
Scorrimento aritmetico	Per numeri in complemento a 2: preserva il bit di segno. Verso destra riempie le posizioni libere con il valore del bit più significativo; verso sinistra coincide con lo scorrimento logico.
Rotazione	I bit in uscita da un lato rientrano dall'altro; l'ultimo bit uscito è scritto nel riporto C.
Rotazione con riporto	Il bit di riporto è incluso nella sequenza ruotata (a sinistra del registro nella rotazione a sinistra, a destra nella rotazione a destra).
LoadByte, StoreByte	LoadByte legge un byte dalla memoria negli 8 bit meno significativi del registro destinazione, azzerando gli altri. StoreByte salva gli 8 bit meno significativi del registro sorgente nella locazione indicata.
Multiply	Moltiplica due numeri in complemento a 2 contenuti in registri. Servono $2n$ bit per il prodotto: in Rk si salvano i bit meno significativi; in alcune architetture quelli più significativi vanno in Rk+1.
Divide	Divisione intera tra due numeri in complemento a 2. Il quoziente è salvato in Rk; in alcune architetture il resto è salvato in Rk+1. Non tutte le architetture hanno moltiplicazione e divisione.

Programmi di esempio

SOMMA DI N NUMERI

In una versione sequenziale (poco efficiente) il programma ha un Load e un Add per ogni numero della lista e, con n numeri, occupa circa $2n$ istruzioni:

```

Load  R2, NUM1
Load  R3, NUM2
Add   R2, R2, R3
Load  R3, NUM3
Add   R2, R2, R3
...
Load  R3, NUMn
Add   R2, R2, R3
Store R2, SOMMA

```

Con un salto condizionato il programma diventa più efficiente: R2 è il contatore, R3 la somma, R4 il puntatore alla lista e R5 il numero prelevato.

```

Load  R2, N           ; carica la dimensione della lista
Clear R3              ; inizializza la somma a 0
Move  R4, #NUM1       ; carica l'indirizzo del primo numero
CICLO: Load R5, (R4)   ; preleva il prossimo numero
Add   R3, R3, R5       ; aggiunge il numero alla somma
Add   R4, R4, #4       ; incrementa il puntatore alla lista
Subtract R2, R2, #1    ; decrementa il contatore
Branch_if_[R2]>0 CICLO ; salta indietro se non ancora finito
Store R3, SOMMA       ; immagazzina la somma finale

```

VOTI DEGLI STUDENTI

Una lista contiene n, il numero di elementi, e per ogni allievo la matricola e i voti di tre prove (16 byte ciascuno). Il programma calcola il totale per ciascuna prova.

```
Move R2, #LISTA      ; carica l'indirizzo della LISTA
Clear R3
Clear R4
Clear R5
Load R6, N           ; carica il valore n
CICLO: Load R7, 4(R2) ; voto della prova 1 dell'allievo
Add R3, R3, R7        ; somma parziale prova 1
Load R7, 8(R2)        ; voto della prova 2
Add R4, R4, R7
Load R7, 12(R2)       ; voto della prova 3
Add R5, R5, R7
Add R2, R2, #16       ; incrementa il puntatore
Subtract R6, R6, #1    ; decrementa il contatore
Branch_if_[R6]>0 CICLO
Store R3, SOMMA1
Store R4, SOMMA2
Store R5, SOMMA3
```

PROGRAMMA CON DIRETTIVE DI ASSEMBLATORE

Lo stesso programma di somma con le direttive: le istruzioni partono dall'indirizzo 100 (una ogni 4 byte, fino a 132), SOMMA è all'indirizzo 200, N a 204 (contiene 150) e la lista parte da NUM1 all'indirizzo 208 e arriva a 804 (600 byte).

```
ORIGIN 100
Load R2, N
Clear R3
Move R4, #NUM1
CICLO: Load R5, (R4)
Add R3, R3, R5
Add R4, R4, #4
Subtract R2, R2, #1
Branch_if_[R2]>0 CICLO
Store R3, SOMMA
...
ORIGIN 200
SOMMA: RESERVE 4
N: DATAWORD 150
NUM1: RESERVE 600
END
```

RICERCA DI UNA STRINGA: RISC E CISC

Date due stringhe ASCII, T lunga n e P lunga m (con m minore di n), il programma determina se P è contenuta in T e trova l'indice della prima occorrenza con un algoritmo a forza bruta. Ecco la versione RISC (R4 = indirizzo di T(n meno m), R5 = indirizzo di P(m), R6 e R7 scorrono T e P):

```
Move R2, #T           ; R2 punta alla stringa T
Move R3, #P           ; R3 punta alla stringa P
Load R4, N            ; preleva n
Load R5, M            ; preleva m
Subtract R4, R4, R5    ; calcola n meno m
Add R4, R2, R4         ; indirizzo di T(n meno m)
Add R5, R3, R5         ; indirizzo di P(m)
CICL01: Move R6, R2     ; R6 scandisce T
Move R7, R3           ; R7 scandisce P
CICL02: LoadByte R8, (R6) ; confronta una coppia di caratteri
LoadByte R9, (R7)
Branch_if_[R8]≠[R9] NONUGUALI
Add R6, R6, #1
Add R7, R7, #1
Branch_if_[R5]>[R7] CICL02 ; reitera se non ancora finito
Store R2, RISULTATO    ; indirizzo di T(i)
Branch FATTO
NONUGUALI: Add R2, R2, #1 ; carattere successivo di T
Branch_if_[R4]≥[R2] CICL01
Move R8, #0xFFFFFFFF ; scrive meno 1: nessuna corrispondenza
Store R8, RISULTATO
FATTO: prossima istruzione
```

La versione CISC è più corta perché usa l'autoincremento, i confronti diretti tra operandi e il Move tra locazioni di memoria:

```
Move R2, #T
Move R3, #P
Move R4, N
Move R5, M
Subtract R4, R5
Add R4, R2
Add R5, R3
CICL01: Move R6, R2
Move R7, R3
CICL02: MoveByte R8, (R6)+
CompareByte R8, (R7)+
Branch≠0 NONUGUALI
Compare R5, R7
Branch>0 CICL02
Move RISULTATO, R2
Branch FATTO
NONUGUALI: Add R2, #1
Compare R4, R2
Branch≥0 CICL01
Move RISULTATO, #0xFFFFFFFF
FATTO: prossima istruzione
```

Il livello software

Dal codice sorgente al programma in esecuzione: assembler, compilatore, linker, loader, debugger e sistema operativo.

Dal sorgente al programma eseguibile

Il passaggio dal codice scritto dal programmatore al programma in esecuzione coinvolge strumenti con compiti distinti:

STRUMENTO	COMPITO
Compilatore	Trasforma un file sorgente scritto in un linguaggio ad alto livello in un file in linguaggio assembly. Il compilatore <i>ottimizzante</i> riorganizza le istruzioni per ottimizzare il codice. Automatizza molti compiti del programmatore assembly (ad esempio la gestione delle aree di attivazione). Un programma ad alto livello può chiamare sottoprogrammi presenti in altri file assembly o scritti in altri linguaggi: il linker gestisce i collegamenti.
Assemblatore	Trasforma i file sorgente assembly in file oggetto.
Linker	Collega vari file oggetto e librerie in un unico programma oggetto.
Loader	Carica il programma oggetto in memoria centrale per l'esecuzione.

L'assemblatore

Per tradurre un file sorgente l'assemblatore compie i seguenti passi:

1. **controllo lessicale**: identifica token, etichette e operandi e li distingue;
2. **analisi sintattica**: controlla la correttezza della sintassi (e la semantica, cioè gli errori di logica);
3. **traduzione delle istruzioni**: traduce il codice operativo tenendo conto dell'architettura che ospiterà il programma;
4. **generazione del file oggetto**: compila le istruzioni assembly nel file oggetto, generando la codifica binaria;
5. riconosce le direttive per l'allocazione della memoria e ne inserisce le informazioni nell'header del file oggetto;
6. riconosce le direttive che assegnano nomi a costanti e sostituisce il valore binario a ogni occorrenza dei nomi;
7. sostituisce il valore binario a ogni occorrenza di etichette di indirizzi relativi;
8. (opzionale) **ottimizzazione**: rende il programma più veloce.

Man mano che incontra etichette e dichiarazioni di costanti, l'assemblatore tiene traccia dei nomi e dei valori corrispondenti nella **tabella dei simboli**.

ASSEMBLATORE A DUE PASSI

Il problema è il simbolo dichiarato dopo essere usato (ad esempio le etichette dei salti in avanti). Per risolverlo si usa un assemblaggio a due passate: l'assemblatore sostituisce ogni occorrenza di un nome con il valore indicato nella tabella dei simboli. Le due fasi sono:

1. **analisi e creazione della tabella dei simboli**: legge il sorgente riga per riga, identifica le etichette e associa a ciascuna un indirizzo di memoria, aggiornando il program counter;
2. **generazione del codice**: rilegge il sorgente, traduce le istruzioni, risolve i riferimenti (etichette) e genera il codice oggetto.

VANTAGGI

Gestione accurata dei simboli. Modularità: separando le fasi il debug è più semplice e gli errori sono più facili da individuare.

SVANTAGGI

Lentezza, dovuta alla scissione delle fasi, e maggiore richiesta di memoria. Tempi di elaborazione più lunghi.

Linker e librerie

Di norma un programma è distribuito in più file sorgente. L'assemblatore genera allora un file oggetto incompleto, con riferimenti a nomi esterni. Il **linker** genera un file oggetto completo combinando più file oggetto e risolvendo i riferimenti ai nomi esterni. Creare più file oggetto è utile per riutilizzare il codice, garantire la modularità e gestire meglio gli errori. Ogni file oggetto deve contenere una lista dei nomi esterni usati e una lista delle etichette da esportare: sono le liste che il linker usa per unire i file.

Le **librerie** sono file che raggruppano più file oggetto dello stesso tipo (ad esempio in `math.h` si trovano le funzioni matematiche). Sono create dal programma **archiver** e contengono le informazioni che permettono al linker di risolvere i riferimenti a nomi esterni. Devono essere specificate all'invocazione del linker.

Il loader

File sorgente, file oggetto e dati sono memorizzati in memoria secondaria (su disco); per essere elaborati dal processore devono essere trasferiti nella memoria centrale. Il **loader**, attivato da una richiesta di esecuzione da parte dell'utente, carica il programma oggetto in memoria. Legge informazioni come la lunghezza del programma e la locazione di caricamento (dall'header del file oggetto), carica il programma in memoria e salta alla prima istruzione da eseguire. Alloca inoltre lo spazio per codice, dati e stack, imposta i puntatori dello stack e gestisce i moduli e le dipendenze del programma.

TIPO DI LOADER	DESCRIZIONE
Statico	Carica il programma interamente in memoria in modo statico.
Dinamico	Carica le parti del programma a runtime, permettendo un uso più efficiente della memoria.
Bootstrap loader	Carica in memoria la porzione iniziale del sistema operativo all'avvio.

Vantaggi del loader: modularità, efficienza e velocità. Per una libreria, ad esempio, il loader carica il programma, risolve i riferimenti alla libreria, carica le parti richieste e prepara il programma all'avvio.

Il debugger

Il compilatore rileva errori sintattici e nomi sconosciuti, ma non gli errori di programmazione (*bug*). Il **debugger** permette di eseguire il programma oggetto e interromperlo in qualsiasi istante per valutarne il funzionamento, controllando anche lo stato di memoria e registri. La tecnica più semplice è la stampa di una variabile da tenere sotto controllo (in C con `printf`); gli ambienti di sviluppo integrati dispongono di debugger più precisi. Il debugger può funzionare in due modalità.

MODALITÀ	DESCRIZIONE
Trace mode	Il programma è eseguito passo passo, interrompendosi dopo ogni istruzione: al termine di ogni istruzione si genera un'eccezione e il debugger è lanciato come routine di servizio; il programma resta fermo finché il programmatore non decide di proseguire; con il comando per continuare si ha il rientro dall'interruzione e si esegue l'istruzione successiva. Utile per studiare le prestazioni, individuare cicli e, in assembly, controllare i registri, che sono pochi.
Breakpoint	L'esecuzione si interrompe in punti di osservazione scelti dal programmatore. Il debugger sostituisce le istruzioni in corrispondenza dei breakpoint con speciali interruzioni software, le trap . Il programma procede normalmente fino alla prima trap, poi si ferma; con un apposito comando il debugger riprende l'esecuzione e reinserisce la trap. I breakpoint possono essere software (inseriti dal debugger), hardware (installati sulla scheda madre) o condizionali (attivati in specifiche condizioni).

Programmare in C accanto all'assembly

Un programma che legge caratteri da tastiera e li visualizza a video tramite scansione (*polling*) accede solo a registri di I/O, che condividono lo spazio di indirizzamento delle locazioni di memoria. Si può scrivere interamente in C, conoscendo gli indirizzi dei registri di I/O, ed è notevolmente più corto dell'equivalente in assembly, perché la gestione dei registri del processore è automatizzata.

In alcuni casi occorre inserire istruzioni assembly nel codice C (ad esempio per accedere a registri di controllo del processore): alcuni compilatori lo consentono con direttive come `asm("codice assembly")`. È spesso possibile definire le routine di servizio di interruzione direttamente in C con la parola chiave *interrupt* all'inizio della definizione di funzione, ad esempio `interrupt void intserv() {...}`. Si può così scrivere in C anche un programma che riceve caratteri da tastiera tramite interruzione e li stampa sullo schermo.

Il sistema operativo

DEFINIZIONE

Il **sistema operativo** (SO) è il software che coordina tutte le attività dell'hardware, cioè dei componenti del modello di Von Neumann (CPU, I/O, RAM, memorie principali e secondarie). È formato da un insieme di routine essenziali residenti nella memoria centrale e da programmi di utilità residenti su disco, caricati in memoria quando servono.

Un processo fondamentale è il **bootstrapping**, che carica in memoria una porzione iniziale del sistema operativo. Esempi di SO sono Windows, macOS e Linux. Un sistema operativo fornisce una piattaforma **affidabile, scalabile, sicura e stabile**.

COMPITI PRINCIPALI

- controllo e allocazione della CPU tramite algoritmi di scheduling;
- controllo della memoria (RAM e non);
- gestione del multitasking;
- interfaccia utente: grafica (GUI) oppure shell, a riga di comando.

COMPONENTI

1. **Kernel**: nucleo del sistema operativo, sovrintende a tutte le altre operazioni.
2. **Interfaccia utente**: grafica oppure da riga di comando.
3. **File di sistema**: file essenziali per sicurezza, rete e così via (ad esempio quello che si occupa dello scheduling).

Le *patch* sono parti del sistema operativo corrette per risolvere falle o bug; una patch del kernel richiede il riavvio della macchina.

MULTITASKING

Un sistema operativo in grado di eseguire più programmi contemporaneamente è detto **concorrente** o **multitasking**. In un caso semplice, con un solo programma che legge dati dal disco e li stampa, il processore salta ripetutamente tra routine del sistema operativo, routine di I/O e programma; con più programmi il processo si può parallelizzare (virtualmente). Per comunicare con programmi, memoria e periferiche il SO usa un sistema di interruzioni e fornisce routine di servizio di interruzione software, ciascuna con il proprio **vettore di interruzione**. Per gestire i programmi concorrenti il SO divide l'esecuzione in quanti di tempo (*time slicing*) e un contatore lancia un'interruzione ogni quanto di tempo τ , attivando la routine **scheduler** che sceglie il prossimo programma da eseguire. Lo scheduler, residente nel kernel, è l'arbitro che decide quali programmi eseguire e in che ordine. Si parla di simulazione della contemporaneità: a un dato istante, per ogni ALU, c'è un unico processo in esecuzione (con una CPU a 12 core si hanno al più 12 processi simultanei).

Un programma può trovarsi in uno di tre stati: **RUNNING** (in esecuzione), **RUNNABLE** (pronto per essere eseguito) e **BLOCKED** (bloccato, in attesa).

Pipeline e prestazioni

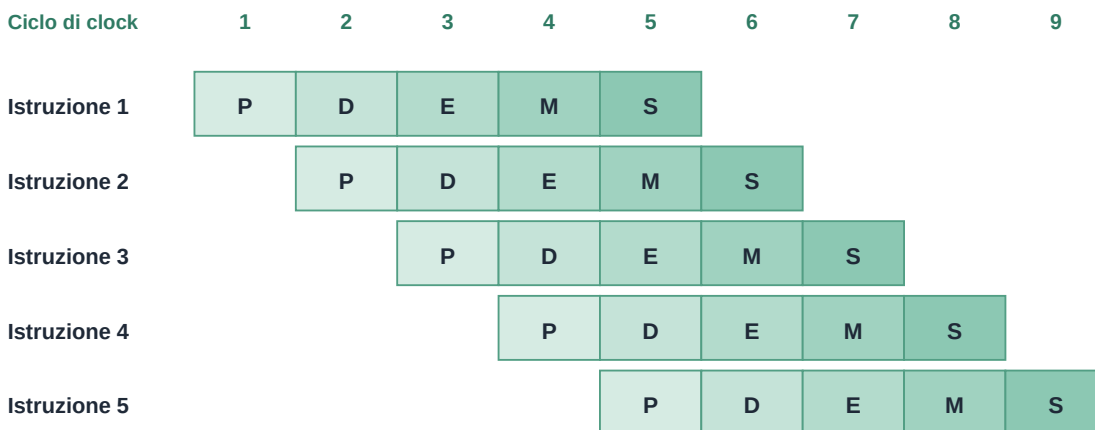
La tecnica che permette ai processori moderni di eseguire più istruzioni contemporaneamente, i conflitti che ne limitano l'efficacia e le soluzioni adottate.

La pipeline

Per svolgere compiti sempre più complessi in tempi ridotti è necessario aumentare le prestazioni del calcolatore. Una delle strategie adottate, ancora oggi in uso, è la **pipeline**, che riprende il concetto di catena di montaggio: le istruzioni sono organizzate in una sequenza continua così da essere parallelizzate, velocizzando il programma.

I processori RISC possono essere strutturati con un'architettura a **cinque stadi**: prelievo (P), decodifica (D), elaborazione (E), memoria (M) e scrittura (S). Nel caso migliore cinque istruzioni sono eseguite in parallelo, ciascuna in uno stadio diverso. La pipeline si adatta meglio ai RISC perché le loro istruzioni, più semplici e di lunghezza costante, sono più facili da parallelizzare. Oggi tutte le CPU hanno una pipeline, essendo CRISC.

Come in una catena di montaggio (un'automobile può richiedere una giornata per essere prodotta, ma lavorando su più auto in stadi diversi se ne produce una ogni pochi minuti), pur impiegando 5 cicli di clock per essere eseguita, a regime ogni istruzione si completa a ogni ciclo. Questo vale nel caso migliore, cioè senza cache miss.



Cinque istruzioni in una pipeline a cinque stadi: dal quinto ciclo, a regime, un'istruzione si completa a ogni ciclo di clock.

BUFFER INTERSTADIO

Per gestire l'esecuzione in pipeline di più istruzioni occorre mantenere informazioni tra uno stadio e l'altro. Esse sono conservate nei **buffer interstadio**, che contengono i registri interstadio (come RA, RB, RZ, PC_Temp), il registro IR per gli identificatori dei registri sorgente e destinazione, e i segnali di controllo per i vari stadi.

I conflitti nella pipeline

Non è sempre possibile eseguire in parallelo tutte le istruzioni: possono verificarsi conflitti che ritardano l'ingresso di nuove istruzioni nella pipeline. I tipi di conflitto sono quattro: dipendenze di dato, ritardi nell'accesso alla memoria, ritardi nei salti e limiti di risorse.

DIPENDENZE DI DATO

Una dipendenza di dato si ha quando un'istruzione contiene un registro sorgente non ancora aggiornato dalle istruzioni precedenti: per esempio in una somma o in una sottrazione in cui un operando è il risultato di un'altra operazione. Le dipendenze si classificano in:

TIPO	DESCRIZIONE
RAW (lettura dopo scrittura)	Per completare un'istruzione serve un dato che un'istruzione precedente non ha ancora prodotto.
WAW (scrittura dopo scrittura)	Due istruzioni cercano di scrivere nello stesso registro o nella stessa parte di memoria: la seconda sovrascrive la prima, quindi l'ordine di esecuzione è cruciale.
WAR (scrittura dopo lettura)	Un'istruzione tenta di scrivere in un registro prima che un'altra abbia completato la lettura della stessa posizione.

Esempio: nella coppia Add R2, R3, #100 e Subtract R9, R2, #30, l'istruzione Add aggiorna R2 solo alla fine della propria fase di scrittura, mentre Subtract legge R2 nella fase di decodifica, quando Add è ancora in esecuzione. Subtract deve restare in stallo finché R2 non è aggiornato (3 cicli di clock).

TECNICHE PER GESTIRE LE DIPENDENZE

TECNICA	DESCRIZIONE
Stallo con NOP e bolle	Si inseriscono istruzioni nulle (NOP, <i>no operation</i>), che occupano uno slot senza modificare lo stato del processore. Ogni NOP crea un ciclo di inattività detto bolla . Le NOP possono essere generate dal compilatore (lato software) o da circuiti di controllo più complessi (lato hardware); i compilatori ottimizzanti possono riordinare il codice spostando istruzioni utili nelle posizioni delle NOP. Funzionano ma sprecano cicli di clock e memoria.
Forwarding (inoltro degli operandi)	I registri interstadio successivi sono inoltrati a stadi precedenti: ad esempio RZ è reso disponibile direttamente nello stadio di esecuzione tramite multiplexer. Il dato passa dall'istruzione produttrice a quella consumatrice senza passare dai registri: si elimina lo stallo e migliorano le prestazioni.
Riordino delle istruzioni	Le istruzioni indipendenti sono eseguite prima di quelle dipendenti, senza rallentare la pipeline.
Speculazione	Si prevede il risultato dell'operazione da cui si dipende e si procede assumendo che la previsione sia corretta.
Pre fetching	Carica in anticipo i dati dalla memoria per ridurre la dipendenza dalla memoria.

RITARDI DELLA MEMORIA

Gli accessi alla memoria richiedono svariati cicli di clock, soprattutto per la differenza di velocità tra le diverse memorie. Se il dato da leggere non si trova nella cache si ha un **cache miss**, con ritardi di 10 o più cicli di clock. Le cause principali sono la latenza costruttiva della memoria (specialmente della RAM), la velocità del bus e il sistema. Mentre l'istruzione attende il dato, le successive restano ferme. Le strategie dei progettisti sono la gerarchia di cache (L1, L2, L3, dalla più veloce alla più lenta), il prefetching e la produzione di RAM a latenza più bassa.

RITARDI NEI SALTII

Durante l'esecuzione di un salto incondizionato l'indirizzo di destinazione è caricato nel PC al terzo passo. Le istruzioni successive già entrate nella pipeline vengono scartate: si ha una **penalità di salto** di due cicli di clock; lo stesso avviene nei salti condizionati quando la condizione è vera. Quando una previsione di salto è sbagliata (*branch misprediction*) il processore esegue il *flushing* della pipeline, cioè la svuota, e la riempie con le istruzioni del percorso corretto.

TECNICA	DESCRIZIONE
Anticipazione e valutazione del salto	Si valuta ed esegue il salto nello stadio di decodifica, riducendo la penalità a un ciclo di clock. Si anticipano due operazioni: la determinazione dell'indirizzo di destinazione (aggiungendo un sommatore nel passo 2) e la valutazione della condizione (spostando il comparatore nel passo 2, con ingressi direttamente le uscite del banco dei registri).
Salto differito	Le istruzioni successive al salto sono eseguite in ogni caso. Il compilatore riorganizza il codice posizionando dopo il salto istruzioni da eseguire comunque; se non ne trova inserisce delle NOP.
Predizione dei salti	Se si riesce a predire l'esito del salto, l'indirizzo di destinazione può essere caricato direttamente nel passo di prelievo.

PREDIZIONE STATICA E DINAMICA

Predizione statica: usa regole semplici basate su informazioni statiche, come la posizione dell'istruzione nel codice o il tipo di salto. La forma più semplice suppone che un salto condizionato non sia mai effettuato. Poiché i salti all'indietro a fine ciclo sono più probabili, mentre i salti in avanti di norma non avvengono, il processore può determinare il segno dello spiazzamento per decidere. Un'alternativa è includere nell'istruzione macchina un **bit di predizione**.

Predizione dinamica: usa il comportamento passato dei salti per influenzare la predizione. La forma più semplice ha due stati: PS (probabilmente salta) e PNS (probabilmente non salta). La forma evoluta ne ha quattro: MPS (molto probabilmente salta), PS, PNS e MPNS (molto probabilmente non salta). I processori moderni usano predittori a due livelli e predittori neurali.

BUFFER DI DESTINAZIONE DI SALTO

Per predire già nello stadio di fetch serve una memoria piccola e veloce, il **buffer di destinazione di salto**, che contiene una tabella delle istruzioni di salto del programma. Per ogni salto sono salvati: l'indirizzo dell'istruzione di salto, uno o due bit di stato per l'algoritmo di predizione e l'indirizzo di destinazione. Dopo il prelievo, l'indirizzo dell'istruzione è cercato nella tabella: se è un salto, le informazioni trovate servono ad aggiornare il PC. Nei programmi grandi la tabella non contiene tutti i salti ma si aggiorna man mano.

LIMITI DI RISORSE

La pipeline va in **stallo** quando una risorsa hardware condivisa è richiesta da più istruzioni contemporaneamente. Ad esempio a ogni ciclo si accede alla cache per prelevare l'istruzione successiva, ma quando un'istruzione Load o Store accede alla memoria si ha uno stallo. Una soluzione è usare **cache separate** per istruzioni e dati.

Valutazione delle prestazioni

Le prestazioni di un processore si valutano con il tempo di esecuzione e con il **throughput**, cioè la frequenza di operazione (istruzioni completate per unità di tempo). Siano N il numero di istruzioni macchina, S il numero medio di cicli di clock per istruzione e R la frequenza di clock del processore.

SENZA PIPELINE

$$T = (N \times S) / R \quad \text{throughput } P_{np} = R / S$$

CON PIPELINE

Nel caso ottimo si completa un'istruzione per ciclo:

$$P_p = R / 1 = R$$

In presenza di conflitti il numero medio di cicli per istruzione S aumenta. Ogni tipo di conflitto indipendente aumenta S di un delta δ , dato dalla percentuale di occorrenze del conflitto p per il numero medio di cicli di stallo

introdotti c:

$$\begin{aligned}\delta_{\text{dato}} &= p_{\text{dato}} \cdot c_{\text{dato}} \\ \delta_{\text{salto}} &= p_{\text{salto}} \cdot c_{\text{salto}} \\ \delta_{\text{miss}} &= p_{\text{miss}} \cdot c_{\text{miss}}\end{aligned}$$

Il throughput tenendo conto dei ritardi dei conflitti è quindi:

$$P_p = R / (S + \delta_{\text{dato}} + \delta_{\text{salto}} + \delta_{\text{miss}})$$

ESEMPIO SVOLTO

Si assumano queste statistiche: dipendenze di dato con stallo di 1 ciclo nel 10% dei casi; salti nel 20% delle istruzioni, con predizione accurata al 90% (stallo di 1 ciclo in caso di errore); cache miss nel prelievo con probabilità 5%; istruzioni load/store nel 30% dei casi, con cache miss nel 10%; 10 cicli di attesa per ogni cache miss.

$$S_{\text{conflitti}} = S + \delta_{\text{dato}} + \delta_{\text{salto}} + \delta_{\text{miss}} = 1 + (0,1 \cdot 1) + (0,2 \cdot 0,1 \cdot 1) + ((0,05 + 0,3 \cdot 0,1) \cdot 10) = 1 + 0,1 + 0,02 + 0,8 = 1,92$$

Il throughput scende quindi a $R / 1,92$, circa il 52% del caso ottimo R .

Processori superscalari

Nelle pipeline viste si elabora un'istruzione per ciclo di clock. I processori moderni dispongono di più ALU, ciascuna con la propria pipeline, e possono parallelizzare le istruzioni migliorando il throughput. I processori che usano l'**emissione multipla** sono detti **superscalari**. Ne è esempio un processore con due unità: una aritmetica (istruzioni aritmetico logiche) e una load/store (accesso alla memoria, su due stadi: calcolo dell'indirizzo e accesso). Aritmetiche e di accesso possono essere eseguite in parallelo, a coppie; il banco dei registri deve consentire la lettura di quattro registri e la scrittura di due in un solo passo.

Le unità di esecuzione possono essere aritmetiche (per interi, per numeri in virgola mobile e per array) e di accesso alla memoria. Nell'esempio l'unità aritmetica ha un solo stadio (esecuzione in un ciclo) e la load/store due. Con la sequenza Add R2, R3, #100; Load R5, 16(R6); Subtract R7, R8, R9; Store R10, 24(R11), le aritmetiche e gli accessi alla memoria si eseguono in parallelo a coppie, quindi nei primi due cicli di clock partono tutte e quattro le istruzioni. Un'istruzione è mandata in esecuzione solo quando tutti i suoi operandi sono disponibili.

COMPONENTI

- unità di prelievo (*fetch unit*) e coda di istruzioni;
- unità di smistamento (*dispatch unit*) e stazioni di prenotazione;
- varie unità di esecuzione e registri temporanei;
- unità di commitment e buffer di riordino.

SMISTAMENTO

Nella fase di smistamento (*dispatching*) il processore verifica che tutte le risorse necessarie a un'istruzione siano disponibili: che i registri temporanei per il risultato siano disponibili e prenotati, che ci sia spazio nella stazione di prenotazione dell'unità di esecuzione e che ci sia una locazione libera nel buffer di riordino. Un **deadlock** è il caso in cui due istruzioni restano bloccate a causa di dipendenze reciproche.

STAZIONI DI PRENOTAZIONE

Due istruzioni dipendenti possono essere inviate a unità di esecuzione differenti senza garanzia sull'ordine. Occorre allora bloccare l'istruzione con dipendenza di dato finché l'altra non è stata eseguita. Lo fanno le **stazioni di prenotazione**: buffer presenti all'ingresso di ciascuna unità di esecuzione, che contengono le istruzioni smistate in attesa di essere eseguite e le informazioni e gli operandi relativi a ciascuna. Non appena i risultati che creano le dipendenze sono disponibili, vengono inoltrati alle stazioni.

ESECUZIONE FUORI ORDINE

A causa di cache miss o eccezioni alcune istruzioni possono essere eseguite fuori ordine, con il rischio che un'istruzione che non avrebbe dovuto essere eseguita modifichi registri o locazioni di memoria (**eccezioni imprecise**). Per evitarlo i risultati delle unità di esecuzione sono memorizzati in **registri temporanei** che svolgono il ruolo dei registri permanenti (**ridenominazione dei registri**). L'**unità di commitment** trasferisce i risultati nei registri permanenti secondo l'ordine di prelievo, grazie al **buffer di riordino**, una coda in cui le istruzioni sono poste nel loro ordine di prelievo.

PIPELINE NELLE CPU CISC

I processori CISC introducono difficoltà nella pipeline per la complessità della codifica delle istruzioni, i modi di indirizzamento più complessi, l'uso di bit di esito e i trasferimenti di dati tra locazioni di memoria. Una soluzione sono i processori i7, con emissione multipla di 4 istruzioni e pipeline a 14 stadi: le istruzioni CISC sono convertite dinamicamente in **micro operazioni** di stile RISC, eseguite in pipeline.

Struttura base del processore e controllo

I blocchi che compongono la CPU, il modo in cui un'istruzione viene eseguita stadio per stadio e le due strategie per generare i segnali di controllo.

Componenti del processore

Il **processore** è un circuito elettronico integrato (chip) che fa da cervello del calcolatore: carica ed esegue le istruzioni elementari necessarie per eseguire i programmi.

COMPONENTE	FUNZIONE
ALU	Esegue le operazioni aritmetiche e logiche necessarie all'esecuzione delle istruzioni.
Circuiti di controllo	Generano i bit di controllo che gestiscono il funzionamento della CPU.
Banco dei registri	Blocco di memoria con i registri generici della CPU.
PC e IR	Registri che contengono l'indirizzo della prossima istruzione e l'istruzione in esecuzione.
Generatore di indirizzi	Aggiorna il contenuto del PC.
Interfaccia processore memoria	Gestisce il trasferimento dei dati tra memoria e CPU.

CALCOLATORE: ELEMENTI E PAROLA DI MEMORIA

Un calcolatore comprende CPU (esegue istruzioni elementari), memoria (contiene il programma e i dati), I/O (circuiti che connettono la CPU ai sistemi di input e output) e bus (collegamenti elettrici che interconnettono i componenti).

BANCO DEI REGISTRI

Il banco dei registri è a tutti gli effetti una memoria molto piccola e velocissima, composta da vari registri con un circuito di accesso in scrittura e lettura. Presenta due ingressi per gli indirizzi di lettura (indirizzo A e indirizzo B) e consente la scrittura di un singolo registro (indirizzo C). La lettura simultanea di due registri si realizza in due modi: un singolo banco con percorsi dati e circuiti di accesso duplicati, oppure due copie del banco di registri, una per la porta A e una per la B.

ALU

L'ALU esegue operazioni aritmetiche (somma, sottrazione) e logiche (AND, OR, XOR). Ha due porte di ingresso (InA e InB), che rappresentano gli operandi, e una porta di uscita con il risultato. Gli ingressi sono collegati ai registri di uscita A e B del banco; un **multiplexer** (MuxB) sceglie se passare un valore immediato come secondo operando.

GENERATORE DI INDIRIZZI DELLE ISTRUZIONI

Genera l'indirizzo della prossima istruzione da inserire nel PC. Un sommatore incrementa il PC di una parola (4 byte) e gli somma uno spiazzamento nel caso di salto. MuxINC seleziona il tipo di incremento; MuxPC seleziona se aggiornare il PC con l'incremento calcolato o con un indirizzo specifico (chiamata a sottoprogramma); il registro temporaneo PC_Temp salva il valore del PC da inserire nel link register durante una chiamata a sottoprogramma.

Esecuzione delle istruzioni

Per eseguire un'istruzione il processore compie tre passi: (1) prelievo dell'istruzione dalla memoria puntata dal PC, $IR \leftarrow [PC]$; (2) incremento del PC di 4 unità, la grandezza di un'istruzione, $PC \leftarrow [PC] + 4$; (3) esecuzione dell'istruzione prelevata. I primi due passi costituiscono la **fase di prelievo**, il terzo la **fase di esecuzione**, in cui si possono svolgere letture e scritture di memoria, letture di registri e operazioni aritmetiche e logiche.

TIPO	ESEMPIO	PASSI
Caricamento	Load R5, X(R7)	1. Prelievo e incremento del PC. 2. Decodifica e lettura di R7. 3. Calcolo dell'indirizzo effettivo. 4. Lettura dell'operando dalla memoria. 5. Caricamento nel registro destinazione R5.
Aritmetica e logica	Add R3, R4, R5	1. Prelievo e incremento del PC. 2. Decodifica e lettura dei registri sorgenti R4 e R5. 3. Calcolo della somma $[R4] + [R5]$. 4. Caricamento del risultato in R3.
Immagazzinamento	Store R6, X(R8)	1. Prelievo e incremento del PC. 2. Decodifica e lettura di R6 e R8. 3. Calcolo dell'indirizzo effettivo $X + [R8]$. 4. Immagazzinamento di R6 nella locazione $X + [R8]$.

ORGANIZZAZIONE A CINQUE STADI

Tutte le operazioni possono essere eseguite in cinque stadi: (1) prelievo dell'istruzione e incremento del PC; (2) decodifica e lettura dei registri dal banco; (3) operazione dell'ALU; (4) lettura o scrittura di dati in memoria; (5) scrittura del risultato nel registro di destinazione.

Per temporizzare i trasferimenti di dati nel circuito si usa un segnale di clock. Calcolare il risultato in un solo stadio richiederebbe un ciclo di clock molto lungo: si divide allora il circuito combinatorio in circuiti più semplici in cascata, uno per stadio, separati da registri. Questa struttura si adatta a funzionare in pipeline.

PERCORSO DATI A CINQUE STADI

STADIO	OPERAZIONI NEL PERCORSO DATI
1	MuxMA sceglie se mandare all'interfaccia processore memoria l'indirizzo in RZ o quello in PC; l'istruzione letta è salvata in IR e decodificata dai circuiti di controllo; i bit del valore immediato sono passati a MuxB.
2	Le uscite A e B del banco dei registri sono copiate nei registri temporanei RA e RB.
3	RA è il primo ingresso dell'ALU; MuxB sceglie RB o il valore immediato come secondo ingresso; il risultato dell'ALU è copiato in RZ; RB è copiato in RM.
4	Se necessario l'indirizzo in RZ è mandato all'interfaccia processore memoria e i dati in RM sono salvati in memoria; MuxY sceglie cosa salvare in RY: il risultato di un'operazione, i dati della memoria o l'indirizzo di rientro da sottoprogramma.
5	Il contenuto di RY è salvato nel banco dei registri.

In notazione RTN, il passo 1 è uguale per tutte le istruzioni: indirizzo di memoria $\leftarrow [PC]$, leggi memoria, $IR \leftarrow$ dati da memoria, $PC \leftarrow [PC] + 4$. Gli altri passi sono:

ISTRUZIONE	PASSO 2	PASSO 3	PASSO 4	PASSO 5
Add R3, R4, R5	$RA \leftarrow [R4], RB \leftarrow [R5]$	$RZ \leftarrow [RA] + [RB]$	$RY \leftarrow [RZ]$	$R3 \leftarrow [RY]$
Load R5, X(R7)	$RA \leftarrow [R7]$	$RZ \leftarrow [RA] + X$	Indirizzo di memoria $\leftarrow [RZ]$, leggi memoria, $RY \leftarrow$ dati	$R5 \leftarrow [RY]$

ISTRUZIONE	PASSO 2	PASSO 3	PASSO 4	PASSO 5
Store R6, X(R8)	$RA \leftarrow [R8], RB \leftarrow [R6]$	$RZ \leftarrow [RA] + X, RM \leftarrow [RB]$	Indirizzo di memoria $\leftarrow [RZ]$, dati per memoria $\leftarrow [RM]$, scrivi memoria	Nessuna azione
Branch CICLO	Decodifica	$PC \leftarrow [PC] +$ spiazzamento	Nessuna azione	Nessuna azione
Branch_if [R5]=[R6] CICLO	$RA \leftarrow [R5], RB \leftarrow [R6]$	Confronta $[RA]$ e $[RB]$; se uguali $PC \leftarrow [PC] +$ spiazzamento	Nessuna azione	Nessuna azione
Call SUB1	Decodifica	$PC_Temp \leftarrow [PC], PC \leftarrow [PC] +$ spiazzamento	$RY \leftarrow [PC_Temp]$	$LR \leftarrow [RY]$
Return	$RA \leftarrow [LR]$	$PC \leftarrow [RA]$	Nessuna azione	Nessuna azione
Call_Register R9	$RA \leftarrow [R9]$	$PC_Temp \leftarrow [PC], PC \leftarrow [RA]$	$RY \leftarrow [PC_Temp]$	$LR \leftarrow [RY]$

ATTESA DELLA MEMORIA

Non è sempre possibile accedere alla memoria in un solo ciclo di clock: se dato o istruzione non sono nella cache, l'esecuzione si blocca finché l'operazione di memoria non è conclusa. Ogni operazione di memoria genera il segnale **MFC** (*memory function completed*). Il circuito di controllo interrompe l'esecuzione finché MFC non vale 1. L'attesa di MFC avviene nel primo passo di ogni istruzione e nel passo 4 delle istruzioni Load e Store.

Segnali di controllo

Per eseguire le istruzioni macchina il processore deve generare sequenze di segnali di controllo per ogni stadio. Essi consistono in: segnali di selezione dei multiplexer; segnali di attivazione di alcuni registri; segnali di condizione; segnali per la gestione della memoria; indirizzi, codice operativo e dati letti dall'istruzione nel registro IR; operazioni da far eseguire alla ALU.

CONTROLLO CABLATO

Approccio hardware: un **contatore modulo 5** scandisce gli stadi di esecuzione. Supponendo di avere m istruzioni, un decodificatore di istruzione genera un vettore di lunghezza m mettendo a 1 solo il bit corrispondente all'istruzione letta in IR. Il generatore dei segnali di controllo produce i segnali sulla base dell'istruzione in esecuzione, dello stadio attuale, dei segnali di condizione ed eventuali segnali esterni (tra cui le interruzioni).

Gli stadi che accedono alla memoria possono durare più cicli di clock: solo in questi stadi occorre bloccare il contatore dei passi finché l'operazione non è conclusa. Il segnale **WMFC** è inviato a ogni accesso alla memoria, mentre MFC segnala che l'operazione è conclusa: $\text{Contatore_abilita} = \neg \text{WMFC} + \text{MFC}$.

Esempi di segnali di controllo cablati, con T1, ..., T5 i passi scanditi dal contatore: abilitazione della scrittura nel banco dei registri $\text{RF_scrittura} = T5 \cdot (\text{ALU} + \text{Load} + \text{Call})$; selettore del valore immediato $\text{B_selettore} = \text{Immediato}$; abilitazione del registro PC $\text{PC_abilita} = T1 \cdot \text{MFC} + T3 \cdot \text{BR}$.

CONTROLLO MICROPROGRAMMATO

I segnali di controllo di ogni passo sono raccolti in una parola di memoria detta **microistruzione**. L'insieme delle microistruzioni di un'istruzione macchina è una **microroutine**, memorizzata in locazioni consecutive della **memoria di controllo**. Il registro μPC contiene l'indirizzo della prossima microistruzione da caricare: all'inizio di un'istruzione macchina il generatore di indirizzi delle microistruzioni carica nel μPC la prima della microroutine corrispondente; a ogni passo il μPC è incrementato.

Organizzazione del processore CISC

Le istruzioni CISC possono occupare più di una parola, quindi un approccio a stadi come nei RISC non è possibile. Un **blocco di interconnessione** permette il trasferimento di dati tra una qualsiasi coppia di componenti. Non esistono registri interstadio, ma registri temporanei per i risultati intermedi. Il blocco di interconnessione è realizzato tramite bus.

INTERCONNESSIONE VIA BUS

Un bus è formato da un insieme di linee a cui sono connessi più dispositivi tramite una porta tristate. Un solo dispositivo alla volta può trasferire dati; nel bus esiste una porta logica, il **bus driver**, che invia il dato sulla linea.

INTERCONNESSIONE A TRE BUS

Una soluzione è l'uso di tre bus: i bus A e B per le sorgenti delle operazioni e il bus C per i risultati. Il generatore di indirizzi è direttamente collegato al PC e il blocco di controllo legge dall'IR. Tutte le componenti sono connesse ai tre bus tramite porte tristate, tutte disattivate tranne il bus driver, e istruzioni diverse sono eseguite in un numero di passi differente.

ESEMPI DI ESECUZIONE SU PROCESSORE CISC

Add R5, R6 si esegue in 3 passi: (1) indirizzo di memoria $\leftarrow [PC]$, leggi memoria, attesa MFC, $IR \leftarrow$ dati da memoria, $PC \leftarrow [PC] + 4$; (2) decodifica dell'istruzione; (3) $R5 \leftarrow [R5] + [R6]$.

And X(R7), R9, prodotto logico bit a bit tra una locazione di memoria (indice e spiazzamento) e un registro, richiede 7 passi:

1. indirizzo di memoria $\leftarrow [PC]$, leggi memoria, attesa MFC, $IR \leftarrow$ dati da memoria, $PC \leftarrow [PC] + 4$;
2. decodifica dell'istruzione;
3. indirizzo di memoria $\leftarrow [PC]$, leggi memoria, attesa MFC, $Temp1 \leftarrow$ dati da memoria (lo spiazzamento X), $PC \leftarrow [PC] + 4$;
4. $Temp2 \leftarrow [Temp1] + [R7]$;
5. indirizzo di memoria $\leftarrow [Temp2]$, leggi memoria, attesa MFC, $Temp1 \leftarrow$ dati da memoria;
6. $Temp1 \leftarrow [Temp1] \text{ AND } [R9]$;
7. indirizzo di memoria $\leftarrow [Temp2]$, dati da memoria $\leftarrow [Temp1]$, scrivi in memoria, attesa MFC.

Algebra booleana e reti logiche

Il linguaggio matematico dei circuiti digitali: funzioni logiche, forme canoniche, minimizzazione e porte.

Algebra booleana della commutazione

L'algebra booleana della commutazione è un sistema algebrico in cui ogni variabile può assumere solo due valori: 0 e 1. Su tali variabili binarie sono definite tre operazioni base, dette **funzioni logiche fondamentali**:

- **somma logica** (OR);
- **prodotto logico** (AND);
- **complementazione**, negazione o inversione (NOT).

Ognuna di queste operazioni prende in ingresso una o più variabili e restituisce in uscita una sola variabile binaria.

SOMMA LOGICA, OR

Vale 1 se almeno uno degli ingressi vale 1. Si denota con «+» o « $\vee$ »: $f(x_1, x_2) = x_1 + x_2 = x_1 \vee x_2$.

PRODOTTO LOGICO, AND

Vale 1 solo se tutti gli ingressi valgono 1. Si denota con « $\cdot$ » o « $\wedge$ »: $f(x_1, x_2) = x_1 \cdot x_2 = x_1 \wedge x_2$.

COMPLEMENTAZIONE, NOT

Inverte il valore dell'unica variabile in ingresso. Si denota con « $\neg$ »: $f(x) = \neg x$. Proprietà di involuzione (doppia negazione): $\neg \neg x = x$.

PROPRIETÀ	SOMMA (OR)	PRODOTTO (AND)
Commutativa	$x_1 + x_2 = x_2 + x_1$	$x_1 \cdot x_2 = x_2 \cdot x_1$
Associativa	$x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$	$x_1 \cdot (x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3$
Elemento neutro	$0 + x = x$	$1 \cdot x = x$
Distributiva	$x + y \cdot z = (x + y) \cdot (x + z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Idempotenza	$x + x = x$	$x \cdot x = x$
Complemento	$x + \neg x = 1$	$x \cdot \neg x = 0$
Proprietà di 1 e di 0	$1 + x = 1$	$0 \cdot x = 0$
Teoremi di De Morgan	$\neg(x + y) = \neg x \cdot \neg y$	$\neg(x \cdot y) = \neg x + \neg y$

Funzioni logiche e tabelle di verità

DEFINIZIONE

Una **funzione logica** è una funzione con una o più variabili binarie di ingresso e una variabile di uscita. Si esprime con una **tabella di verità**: esiste una sola tabella di verità per ogni funzione logica. Con n variabili di ingresso la tabella ha 2^n righe e $n + 1$ colonne.

Un'espressione logica è una possibile realizzazione di una funzione logica: esistono infinite espressioni che realizzano la stessa funzione. Due espressioni sono **equivalenti** se rappresentano la medesima funzione; per verificarlo si confrontano le rispettive tabelle di verità. Per sapere quale funzione rappresenta un'espressione basta calcolarne la tabella di verità. Per forzare la precedenza di un operatore si usano le parentesi.

Esempio: le espressioni $(x_1 + x_2)(x_1 + \neg x_2)(\neg x_1 + x_2)$ e $x_1 \cdot x_2$ sono equivalenti, perché hanno la stessa tabella di verità.

x_1	x_2	$x_1 + x_2$	$x_1 + \neg x_2$	$\neg x_1 + x_2$	PRODOTTO DELLE TRE SOMME	$x_1 \cdot x_2$
0	0	0	1	1	0	0
0	1	1	0	1	0	0
1	0	1	1	0	0	0
1	1	1	1	1	1	1

PRECEDENZA TRA OPERATORI

OPERATORE	PRECEDENZA
Negazione	1 (si esegue per prima)
Prodotto	2
Somma	3

Dalla tabella di verità all'espressione

Esiste un metodo per trovare almeno un'espressione di una funzione partendo dalla sua tabella di verità.

MINIMI TERMINI E PRIMA FORMA CANONICA

Un **minimo termine** (*minterm*) è una funzione di n variabili che vale 1 solo per una specifica configurazione delle variabili. Si rappresenta come prodotto delle variabili, in forma diretta se la variabile vale 1 nella configurazione, in forma negata se vale 0. La somma logica dei minimi termini corrispondenti a tutte le configurazioni in cui la funzione vale 1 equivale alla funzione cercata. Un'espressione in questa forma è una **somma di prodotti** (SOP), detta **prima forma canonica**. Tale forma è unica.

MASSIMI TERMINI E SECONDA FORMA CANONICA

Un **massimo termine** (*maxterm*) è una funzione di n variabili che vale 0 solo per una specifica configurazione. Si rappresenta come somma delle variabili, in forma diretta se la variabile vale 0 nella configurazione, in forma negata se vale 1. Il prodotto logico dei massimi termini corrispondenti alle configurazioni in cui la funzione vale 0 equivale alla funzione. Un'espressione in questa forma è un **prodotto di somme** (POS), detta **seconda forma canonica**. Anche questa forma è unica.

FORMA MINIMA

Un'espressione è in **forma minima** quando non esiste alcuna altra espressione equivalente con costo inferiore. Si usa il **criterio di costo dei letterali**: il costo di un'espressione è il numero di comparse di variabili nell'espressione stessa. Un'espressione in forma minima è più semplice ed economica da realizzare come circuito.

Per passare dalla prima forma canonica alla forma minima si procede in tre mosse: si associano le coppie di minimi termini che differiscono per una sola variabile (diretta in uno, negata nell'altro) e si usa la proprietà distributiva; si applica la legge del complemento, trasformando in 1 le somme di variabili complementari ($x + \neg x =$

1); si usa l'idempotenza ($x + x = x$), duplicando i minimi termini quando necessario.

ESEMPIO SVOLTO

Sia f_1 la funzione di tre variabili con questa tabella di verità.

x_1	x_2	x_3	F_1
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- **Prima forma canonica:** la funzione vale 1 per le configurazioni 0, 1, 3 e 7, quindi $f_1 = m_0 + m_1 + m_3 + m_7 = \neg x_1 \neg x_2 \neg x_3 + \neg x_1 \neg x_2 x_3 + \neg x_1 x_2 x_3 + x_1 x_2 x_3$, di costo 12.
- **Seconda forma canonica:** la funzione vale 0 per le configurazioni 2, 4, 5 e 6, quindi $f_1 = M_2 \cdot M_4 \cdot M_5 \cdot M_6 = (x_1 + \neg x_2 + x_3)(\neg x_1 + x_2 + x_3)(\neg x_1 + x_2 + \neg x_3)(\neg x_1 + \neg x_2 + x_3)$.
- **Forma minima:** $m_0 + m_1 = \neg x_1 \neg x_2 (\neg x_3 + x_3) = \neg x_1 \neg x_2$ e $m_3 + m_7 = x_2 x_3 (\neg x_1 + x_1) = x_2 x_3$, perciò $f_1 = \neg x_1 \neg x_2 + x_2 x_3$, di costo 4.

METODO DI KARNAUGH

Il metodo di Karnaugh facilita la semplificazione in forma minima: la tabella di verità è rappresentata in una **mappa di Karnaugh** e la minimizzazione si ottiene raggruppando geometricamente i minimi termini. È vantaggioso per funzioni con poche variabili (tre o quattro). Per funzioni a tre variabili le colonne rappresentano le coppie di due variabili e le righe la terza; le caselle sono ordinate in modo che caselle adiacenti differiscano per una sola variabile. Le mappe a quattro variabili ne hanno due nelle righe e due nelle colonne.

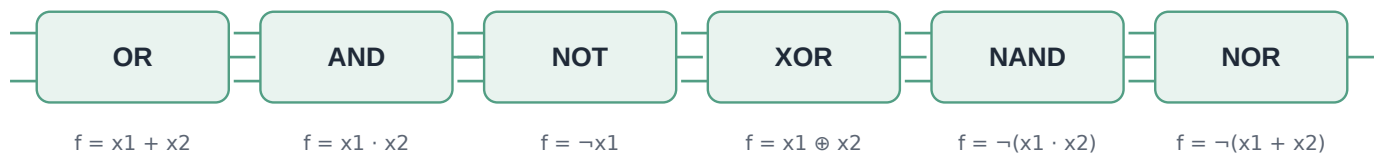
Come si usa: si raggruppano le caselle di valore 1 adiacenti orizzontalmente e verticalmente, formando gruppi di dimensione massima, con un numero di caselle multiplo di 2 (potenza di 2). Ogni gruppo rappresenta il prodotto delle variabili che al suo interno hanno lo stesso valore. Si ottiene così un'espressione SOP in forma minima, in cui ogni gruppo è uno dei prodotti.

DEFINIZIONE

Condizione di indifferenza (*don't care*): spesso una funzione non è definita per tutte le combinazioni di valori delle variabili. Le combinazioni non usate sono in condizione di indifferenza e nelle tabelle sono indicate con «X». Il loro valore può essere scelto in modo da minimizzare il più possibile la forma minima.

Circuiti logici

Le operazioni logiche di base possono essere realizzate da semplici circuiti elettronici detti **porte**. Una rete di porte logiche è una **rete combinatoria**, con n ingressi binari e m uscite binarie ($n, m \geq 1$).



Porte logiche fondamentali con la relativa funzione (rappresentazione semplificata).

Grazie alla proprietà associativa di AND e OR le porte possono essere estese a più di due ingressi: equivale a mettere in cascata, o ad albero su due livelli, porte AND o OR a due ingressi. Un'espressione logica si rappresenta come rete combinatoria con una porta per ogni operatore e collegando le porte ad albero secondo i livelli di precedenza dell'espressione. Le espressioni SOP e POS corrispondono a reti a due livelli.

XOR, DIFFERENZA SIMMETRICA

La funzione XOR vale 1 se e solo se il numero di 1 nei suoi ingressi è dispari. Si denota con « $\oplus$ »: $f = x_1 \oplus x_2 = \neg x_1 x_2 + x_1 \neg x_2$. Vale quindi $x_1 \oplus x_2 = 0$ se e solo se $x_1 = x_2$.

NAND E NOR

Due funzioni di largo uso pratico sono l'AND negato (**NAND**, operatore $\uparrow$) e l'OR negato (**NOR**, operatore $\downarrow$):

$$x_1 \uparrow x_2 = \neg(x_1 \cdot x_2) = \neg x_1 + \neg x_2 \qquad x_1 \downarrow x_2 = \neg(x_1 + x_2) = \neg x_1 \cdot \neg x_2$$

La NAND a n ingressi è la negazione della congiunzione di tutti gli ingressi: **vale 0 solo se tutti gli ingressi valgono 1**, in tutti gli altri casi vale 1. Le porte a più ingressi servono a gestire un numero diverso di variabili di input.

PORTE UNIVERSALI

NAND e NOR sono porte **universali**: qualsiasi funzione combinatoria può essere realizzata con reti di soli NAND o di soli NOR. Mediante le leggi di De Morgan e la legge di involuzione si passa da una SOP a una rete di NAND: si sostituiscono tutte le porte AND e OR con porte NAND mantenendo gli ordini di priorità. Servono due strumenti: realizzare una porta NOT con porte NAND (una NAND con gli ingressi unificati nega la variabile in ingresso) e realizzare porte NAND con più ingressi. Le operazioni NAND e NOR sono commutative ma non associative: le porte NAND a più ingressi si ottengono con la legge di involuzione e la rappresentazione della NOR come NAND. Anche una NOR con gli ingressi unificati si comporta come una NOT.

Circuiti aritmetici

Come l'hardware esegue le operazioni: addizionatori, sottrazione, moltiplicazione e divisione.

Addizionatore a 1 bit

L'addizionatore a 1 bit somma numeri binari a un bit; il riporto in uscita delle cifre precedenti è assegnato come riporto in ingresso all'addizione successiva. Si esprime con due funzioni logiche a tre ingressi (i due bit da sommare più il riporto in ingresso): la prima calcola la somma s_i , la seconda il riporto in uscita c_{i+1} . Gli addizionatori a 1 bit sono parte integrante delle ALU delle CPU.

x_i	y_i	RIPORTO IN INGRESSO c_i	SOMMA s_i	RIPORTO IN USCITA c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i c_i + y_i c_i + x_i y_i$$

DEFINIZIONE

Fondendo in un unico circuito le reti logiche per somma e riporto in uscita si ottiene l'**addizionatore completo** (*full adder*): somma 3 bit (due da sommare più il riporto in ingresso) e produce in uscita somma e riporto.

Addizionatore a propagazione di riporto

Collegando n addizionatori completi in modo che il riporto si propaghi, si ottiene un circuito capace di sommare numeri binari fino a n bit: l'**addizionatore a propagazione di riporto** (*ripple carry adder*, RCA). Il primo stadio elabora il bit meno significativo, l'ultimo il più significativo.

TRABOCCO

Il trabocco si ha quando il risultato supera la capacità di rappresentazione del registro. L'addizione di due numeri in complemento a due corrisponde alla somma di due naturali senza contare il riporto in uscita, mentre la sottrazione è un'addizione con il sottraendo complementato a due, purché non avvenga trabocco. Il trabocco si calcola come:

$$\text{trabocco} = c_n \oplus c_{n-1}$$

ADDIZIONATORE ALGEBRICO A N BIT

L'ALU per addizione e sottrazione si ottiene con un addizionatore a propagazione di riporto, usando il bit di controllo $OpType_0$ per complementare a due il secondo addendo in caso di sottrazione. Quando $OpType_0 = 1$ si ha un riporto in ingresso al bit meno significativo e il secondo addendo è complementato attraverso una catena di porte XOR in parallelo ($y_i \oplus OpType_0$). Il trabocco è calcolato come $c_n \oplus c_{n-1}$.

RITARDI DEL RIPPLE CARRY ADDER

Il ritardo totale di un circuito dipende dal percorso più lento. Si assume che ogni porta logica semplice introduca un ritardo fisso. Un full adder genera s_i dopo un ritardo di porta e c_{i+1} dopo due ritardi di porta. In un RCA a n bit il riporto c_n è generato dopo $2n$ ritardi di porta, mentre l'ultimo bit del risultato, s_{n-1} , dopo $2n$ meno 1 ritardi di porta.

Addizionatore con anticipo di riporto

Fattorizzando l'equazione del riporto si ha $c_{i+1} = x_i y_i + (x_i + y_i) c_i = G_i + P_i c_i$, con:

$$G_i = x_i \cdot y_i \text{ (funzione di generazione)} \quad P_i = x_i + y_i \text{ (funzione di propagazione)}$$

G_i e P_i dipendono solo dagli ingressi x_i e y_i e si calcolano tutti in parallelo in un ritardo di porta. Espandendo iterativamente c_i fino a c_0 si ottiene un'equazione per c_{i+1} in funzione solo di P , G e c_0 :

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_0 c_0$$

In questo modo ciascun riporto può essere calcolato **in parallelo dopo 2 ritardi di porta**. Si modifica allora il sommatore per fornire in uscita G_i (tramite una porta AND con ingressi x_i e y_i), P_i (tramite una porta XOR con ingressi x_i e y_i) e s_i (tramite due porte XOR annidate con ingressi x_i , y_i e c_i). Con 4 celle da 1 bit si realizza un addizionatore con anticipo di riporto a 4 bit:

$$\begin{aligned} c1 &= G0 + P0 \ c0 \\ c2 &= G1 + P1 \ G0 + P1 \ P0 \ c0 \\ c3 &= G2 + P2 \ G1 + P2 \ P1 \ G0 + P2 \ P1 \ P0 \ c0 \\ c4 &= G3 + P3 \ G2 + P3 \ P2 \ G1 + P3 \ P2 \ P1 \ G0 + P3 \ P2 \ P1 \ P0 \ c0 \end{aligned}$$

Il riporto c_4 è generato dopo 3 ritardi di porta e il risultato S dopo 4 ritardi di porta. Con più di 4 celle i valori di fan in nelle porte che generano i riporti diventano troppo elevati: una soluzione è replicare l'idea dell'anticipo di riporto su più livelli. In un addizionatore a 4 bit si ha $c_4 = G_k + P_k c_0$, con $G_k = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$ e $P_k = P_3 P_2 P_1 P_0$. Per ottenere un addizionatore a 16 bit si collegano 4 addizionatori a 4 bit con un blocco di anticipo di riporto che riceve i vari G_i e P_i e genera in parallelo c_4 , c_8 , c_{12} e c_{16} :

$$\begin{aligned} c4 &= G_{I0} + P_{I0} \ c0 \\ c8 &= G_{I1} + P_{I1} \ G_{I0} + P_{I1} \ P_{I0} \ c0 \\ c12 &= G_{I2} + P_{I2} \ G_{I1} + P_{I2} \ P_{I1} \ G_{I0} + P_{I2} \ P_{I1} \ P_{I0} \ c0 \\ c16 &= G_{I3} + P_{I3} \ G_{I2} + P_{I3} \ P_{I2} \ G_{I1} + P_{I3} \ P_{I2} \ P_{I1} \ G_{I0} + P_{I3} \ P_{I2} \ P_{I1} \ P_{I0} \ c0 \end{aligned}$$

In questo caso c_{16} è generato dopo 5 ritardi di porta e il risultato S dopo 8 ritardi di porta.

Moltiplicazione

Il metodo imparato a scuola vale anche in binario: si moltiplica ogni cifra del moltiplicatore per il moltiplicando e si sommano i risultati, fatti scorrere di una posizione per cifra. Nel caso binario il moltiplicando è moltiplicato solo per 0 o per 1. Il prodotto di due numeri di n cifre è composto da $2n$ cifre.

CIRCUITO MOLTIPLICATORE SEQUENZIALE

La moltiplicazione di numeri senza segno si realizza in modo sequenziale con un solo addizionatore a n bit e due registri a scorrimento. A ogni ciclo l'addizionatore somma il moltiplicando e il prodotto parziale, fatto scorrere a destra di una posizione. All'inizio il registro A vale 0 e il registro Q contiene il moltiplicatore; alla fine la concatenazione dei registri A e Q contiene il prodotto.

```
A = 0
Q = Moltiplicatore
M = Moltiplicando
Per n cicli:
  se q0 = 1:  A = A + M
  altrimenti: A = A + 0
  c = riporto
  c || A || Q scorrono verso destra di una posizione
```

MOLTIPLICAZIONE DI NUMERI CON SEGNO

Per moltiplicare numeri in complemento a 2 occorre modificare l'algoritmo, distinguendo i casi:

- **moltiplicatore positivo**: si procede normalmente;
- **moltiplicando positivo**: si procede normalmente;
- **moltiplicando negativo**: si estende il segno quando si riporta il moltiplicando in tabella;
- **moltiplicatore negativo**: si fa il complemento a 2 di moltiplicando e moltiplicatore per ricondursi al caso di moltiplicatore positivo.

ALGORITMO DI BOOTH

La soluzione più efficiente è l'**algoritmo di Booth**, che ricodifica il moltiplicatore come somma e sottrazione di potenze di 2. Nel caso semplice in cui il moltiplicatore contiene una sequenza contigua di 1, ad esempio $Q = 0011110 = 0100000$ meno $0000010 = 2^5$ meno 2^1 , il prodotto è $P = M \cdot Q = M \cdot 2^5$ meno $M \cdot 2^1$. Il prodotto è uguale al moltiplicando fatto scorrere di 5 posizioni a sinistra più il complemento a due del moltiplicando fatto scorrere di 1 posizione a sinistra. La ricodifica vale per qualsiasi moltiplicatore: $Q = 1100111011 = -2^8 + 2^6 - 2^3 + 2^2 - 2^0$, quindi $P = -M \cdot 2^8 + M \cdot 2^6 - M \cdot 2^3 + M \cdot 2^2 - M \cdot 2^0$.

RICODIFICA BIT PAIR

La ricodifica **bit pair** raggruppa a coppie i bit del moltiplicatore ricodificato con Booth. Ciascuna coppia è riscritta come un singolo coefficiente da moltiplicare per il moltiplicando, così il numero di addizioni da eseguire è dimezzato.

COPPIA RICODIFICATA	CALCOLO	COEFFICIENTE	COPPIA EQUIVALENTE
(+1, -1)	$2M - M$	$1 \cdot M$	(0, +1)
(-1, +1)	$-2M + M$	$-1 \cdot M$	(0, -1)
(+1, 0)	$2M + 0M$	$2 \cdot M$	(0, +2)

Divisione

Anche per la divisione di interi si parte dall'algoritmo scolastico: si allinea il divisore con il dividendo a partire da sinistra; se il divisore è contenuto nel dividendo si calcola la divisione parziale tra le cifre allineate e si aggiunge il risultato al quoziente; si trascrive il resto e gli si aggiunge la cifra successiva del dividendo; si riallinea il divisore e si riprende dal secondo passo.

DIVISIONE CON RIPRISTINO

Il circuito sequenziale per la divisione si realizza con un registro M (divisore), due registri a scorrimento A e Q (Q contiene il dividendo) e un addizionatore a $n + 1$ bit. All'inizio M contiene il divisore, A vale 0 e Q contiene il dividendo. Si eseguono n volte tre passi:

1. si fanno scorrere A e Q verso sinistra di una posizione;
2. si sottrae M da A e si inserisce il risultato in A;
3. si controlla il segno: se A è negativo si pone $q_0 = 0$ e si ripristina A sommando M ($A = A + M$); altrimenti si pone $q_0 = 1$.

Alla fine A contiene il resto e Q contiene il quoziente.

DIVISIONE SENZA RIPRISTINO

L'algoritmo si può semplificare eliminando il passo di ripristino e si divide in due stadi. **Stadio 1**, da eseguire n volte: se il segno di A è 0, si fanno scorrere A e Q a sinistra di una posizione e si sottrae M da A; altrimenti si fanno scorrere A e Q a sinistra e si somma M ad A. Poi, se il segno di A è 0 si pone $q_0 = 1$, altrimenti $q_0 = 0$. **Stadio 2**: se il segno di A è 1, si somma M ad A.

Tecnologia dei circuiti digitali

Come le porte logiche vengono realizzate con i transistor e quali sono i parametri che ne determinano velocità e limiti.

Rappresentazione delle variabili binarie

Nei circuiti elettronici i valori 0 e 1 delle variabili binarie sono rappresentati con valori di **tensione elettrica**. Una **soglia di separazione** discretizza il valore della tensione: i valori superiori alla soglia rappresentano 1, quelli inferiori 0. Per evitare l'incertezza dovuta al rumore del circuito, i valori prossimi alla soglia non sono presi in considerazione: questa zona è detta **banda vietata**.

I transistori

I transistori sono componenti elettronici che svolgono la funzione di **interruttori**. A seconda della tensione in ingresso possono trovarsi in due stati: **conduzione** e **interdizione**. La tecnologia più comune è il transistor a metallo ossido semiconduttore (**MOS**). Valori tipici: $V_{cc} = 5\text{ V}$ con $V_{soglia} = 2,5\text{ V}$, oppure $V_{cc} = 3,3\text{ V}$ con $V_{soglia} = 1,5\text{ V}$.

Un transistorore ha tre collegamenti: **base** (porta), **pozzo** e **sorgente**. Se il transistorore è in conduzione la tensione nel pozzo diventa uguale a quella della sorgente.

TIPO	BASE ALTA	BASE BASSA	SORGENTE
NMOS	Conduzione	Interdizione	Collegata alla massa
PMOS	Interdizione	Conduzione	Collegata all'alimentazione

PORTE CON TRANSISTORI NMOS

- **NOT**: un transistorore NMOS con la sorgente alla massa e il pozzo all'alimentazione tramite una resistenza. Con tensione di ingresso 1 si ottiene uscita 0 e viceversa.
- **NOR**: due NMOS in parallelo; l'uscita vale 1 se e solo se entrambi i transistori sono in interdizione.
- **NAND**: due NMOS in serie; l'uscita vale 0 se e solo se entrambi i transistori sono in conduzione.

Tecnologia CMOS

Il principale problema dei transistori NMOS è che sono energivori: consumano molta energia in stato di conduzione, a causa della resistenza. Si risolve con la tecnologia **MOS complementare (CMOS)**: il circuito è composto da un ramo di transistori NMOS collegato in serie a uno di PMOS. I due rami sono complementari e in stato stabile non c'è mai continuità tra massa e alimentazione.

VANTAGGI

- consumo di potenza ridotto (solo in fase di commutazione);
- potenza elettrica dissipata proporzionale alla frequenza di commutazione;
- dimensioni molto ridotte dei transistori MOS (miliardi di transistori integrati), che consentono un'alta frequenza massima di commutazione (dell'ordine dei GHz).

PORTA CMOS	REALIZZAZIONE
NOT	Un NMOS in serie a un PMOS che condividono la stessa tensione di ingresso alla base. Quando un transistor è in interdizione l'altro è in conduzione.
NAND	Ramo NMOS con 2 transistori in serie e 2 PMOS in parallelo. Costo: 4 transistori.
NOR	Ramo NMOS con 2 transistori in parallelo e ramo PMOS con 2 in serie.
AND	Una porta NOT collegata all'uscita di una porta NAND. Costo: 6 transistori.

Ritardi in un circuito

GRANDEZZA	DEFINIZIONE
Tempo di transizione	Tempo impiegato da un segnale per passare da un livello a un altro.
Ritardo di propagazione	Tempo che l'uscita di un circuito impiega ad adattarsi ai nuovi valori di input. Il ritardo del percorso più lento tra ingresso e uscita si dice percorso critico .
Frequenza di lavoro	Numero di volte che il circuito commuta in un determinato tempo.

FAN IN E FAN OUT

Il numero di ingressi di una porta logica si chiama **fan in**. Il numero di ingressi paralleli a cui può essere collegata l'uscita di una porta si chiama **fan out**. Valori elevati peggiorano ritardo di propagazione e margine di rumore: sono limitati a circa 10 per porta.

PORTA TRISTATE

Più uscite non si possono collegare a uno stesso ingresso: si rischierebbe il cortocircuito. La **porta tristate** ha un ingresso dato e un ingresso di abilitazione, e un'uscita a tre stati: 0, 1 e Z (alta impedenza). Con abilitazione 1 l'uscita è il segnale di ingresso, con 0 è Z. In forma di tabella (abilitazione, segnale): (0, 0) dà Z, (0, 1) dà Z, (1, 0) dà 0, (1, 1) dà 1.

Circuiti integrati

Le porte logiche sono raggruppate in **circuiti integrati**: piastrelle di silicio incapsulate e dotate di morsetti (*pin*) esterni. Per scala di integrazione si distinguono quattro tipologie.

SCALA	DIMENSIONE	ESEMPI
SSI	Piccola: poche porte logiche	Porte singole
MSI	Media	Addizionatori, sottrattori, singoli registri, multiplexer
LSI	Grande	ALU, banco dei registri, piccoli processori
VLSI	Molto grande	Memorie molto capaci, processori potenti

Decodificatore e multiplexer

DEFINIZIONE

Il **decodificatore** (*decoder*) è un blocco funzionale combinatorio che decodifica un codice binario in ingresso. Ha n ingressi e 2^n uscite e attiva la linea di uscita corrispondente al numero binario in ingresso.

Il **multiplexer** è un circuito logico che seleziona uno dei suoi «ingressi dato» e lo convoglia nella sua uscita. Ha n ingressi di selezione, 2^n ingressi dato e un'uscita: l'ingresso dato è selezionato dalla configurazione degli n bit di selezione. È facilmente realizzabile come somma di prodotti degli ingressi.