Cap. 4

4. COMPONENTI INFORMATICHE PER ELABORAZIONI IN TEMPO REALE

4.1 INTRODUZIONE

Il comportamento complessivo di un sottosistema informatico di automazione risulta, come è naturale che sia, dai comportamenti "locali" delle varie componenti (processori, memorie, interfacce, SW, ecc.) e dalle loro reciproche interazioni.

Possiamo chiamare "risorse" le componenti appena citate e che sono parzialmente prefigurate (o imposte) in sede di specifica e completamente determinate in sede di progetto. Su tali risorse in sede di progetto vanno "mappate" le funzionalità locali in cui si scompone la funzionalità globale richiesta.

La soluzione al problema di effettuare la scomposizione in funzionalità locali e l'attribuzione di queste alle risorse non è unica e anzi comporta numerose alternative che privilegiano diversi fattori di merito (costo, ingombro, assorbimento, velocità, robustezza, ecc.) e spostano vincoli di progetto da alcune risorse ad altre.

In questo capitolo cominciamo con un’analisi della tipica dicotomia tra funzionalità intrinseca e funzionalità programmata, per poi passare ad una breve rassegna delle classi di risorse tipiche del mondo informatico e delle loro caratteristiche significative rispetto alle scelte di progetto.

Nella classificazione che segue si è preferita una sequenza di tipo "bottom up" perchè:

- ogni categoria di risorse dipende in qualche modo da quelle precedenti su cui si basa o di cui deve arricchire o completare le funzionalità;
- con questa sequenza le prime categorie sono quelle più probabilmente vincolate da specifiche, disponibilità sul mercato, capacità produttive del fornitore, ecc., mentre si procede in successione verso le categorie in cui si aprono più alternative per il progettista.

4.2 FUNZIONALITA' INTRINSECA E FUNZIONALITA' PROGRAMMATA

O approccio hardware e software alla funzionalità.

Anche senza voler qui approfondire questo argomento in modo sistematico, sembra importante fare alcune considerazioni che risultano essenziali per la problematica di cui si sta trattando.

Chiamiamo **funzionalità intrinseca** di un dispositivo quella che deriva dal modo in cui esso è fatto, cioè dalla sua struttura, che mette in gioco i particolari fenomeni fisici che tale funzionalità determinano.
Il tipico esempio è costituito dai circuiti analogici, come amplificatori, derivatori, integratori, sommatori, ecc. Ognuno di essi ha un'evoluzione che dipende esclusivamente dai suoi ingressi, ed è quindi descrivibile in termini "locali" al singolo elemento, indipendentemente da quanti elementi compongano un sistema e da come essi sono interconnessi. (Es. legge di Ohm su un resistore, che non dipende dal resto del circuito in cui il resistore è inserito). Quanto appena detto vale anche per i circuiti digitali come porte logiche, contatori, ecc.

In un certo senso si può parlare di comportamento "*context-free*" che presenta diversi vantaggi per la descrizione di sistemi paralleli in tempo reale: tutti gli effetti di mutua influenza sono espliciti nelle interconnessioni che trasferiscono i segnali (sono detti anche sistemi a logica cablata). Nell'analisi di un sistema descritto in forma modulare, il comportamento interno è più facilmente comprensibile perchè non ci sono altre influenze che quelle "visibili" dai collegamenti.

Chiamiamo **funzionalità programmata** quella funzionalità che si ottiene "guidando" la funzionalità intrinseca di una macchina **programmabile** (tipicamente un calcolatore), mediante un programma. Alla macchina sono presentati cioè oltre ai segnali esterni, anche dei "comandi" (le istruzioni), che essa si procura (*fetch*) secondo un flusso con cadenza propria completamente scorrelato, a livello elementare, con i segnali esterni. Si individuano in tal modo dei "flussi" temporali di esecuzione (o sequenze) in numero ridotto (pari al numero di CPU presenti nel sistema) in cui l'effetto di dati di ingresso ed istruzioni dipende dal reciproco intreccio temporale, a priori casuale, rendendo quindi molto più complessa l’analisi del funzionamento.

La notevole flessibilità ottenibile mediante la funzionalità programmata rende interessante questo approccio talora anche per realizzare funzioni che sarebbero elementari in un approccio cablato. Ad esempio la funzione logica combinatoria AND è ottenibile da semplici circuiti a funzionalità intrinseca, con tempi di ritardo di pochi ns. La stessa funzione nella realizzazione programmata mette invece in gioco i noti meccanismi di un calcolatore (fetch di un’istruzione, lettura degli operandi, attivazione dell’ALU) che sono assai più complessi e, in ultima analisi, circa 1000 volte più lenti della corrispondente soluzione cablata.

E' importante notare che la descrizione dei fenomeni può assumere aspetti anche molto diversi cambiando la scala dei tempi, in modo analogo a come cambia la descrizione del

comportamento di un fluido se visto a livello di meccanica molecolare (analitico) o di grandezze globali (sintetico).

In effetti una macchina programmata sequenziale può assumere “ai morsetti” un comportamento analogo a quello di una macchina a funzionalità intrinseca parallela, se osservata con una scala di tempi con la granularità dei secondi, mentre certamente saranno molto diversi i comportamenti analizzati con granularità di microsecondi.

Ciò significa che, pur di accettare una granularità temporale sufficientemente grossolana, ed in presenza di fenomeni sufficientemente lenti, è possibile scegliere tra un'implementazione HW o SW di una data funzionalità.

La linea di soglia di tale granularità dipende dalla tecnologia, che determina la velocità dei circuiti elementari, e dalla "adeguatezza" delle istruzioni SW ad eseguire le funzioni richieste.
E’ comunque opportuno notare che si suppone che la tecnologia comporti, rispetto alla velocità di esecuzione, un effetto moltiplicativo eguale per entrambi gli approcci cablato e programmato.

L'equivalenza funzionale macroscopica viene ottenuta ovviamente mediante tecniche diverse, a seconda che si utilizzi l'approccio HW o SW: è interessante fare alcune considerazioni comparative tra questi due approcci.

Nell'HW il parallelismo è "spontaneo", mentre la sequenzialità deve essere guidata con opportuni giochi di "abilitazioni" o "disabilitazioni" selettive delle varie parti, che invece spontaneamente evolverebbero tutte in parallelo con la loro funzionalità naturale (intrinseca).

Nel SW è spontanea la sequenzialità, non si pone certo il problema di disabilitare le funzioni non volute, semmai occorrono delle tecniche, talvolta macchinose, per simulare l'esecuzione in parallelo di varie funzioni, e tale parallelismo è virtualmente ottenibile solo a risoluzioni temporali di qualche ordine di grandezza più grossolane di quelle relative alle funzioni elementari interne, cioè dei tempi di esecuzione delle singole istruzioni (dell’ordine del microsecondo).

Le tipiche descrizioni HW (schemi elettrici o logici o a blocchi) seguono un paradigma dichiarativo ed evidenziano la spazialità in cui si collocano i dispositivi, mentre la successione degli istanti temporali è sottintesa come una serie di istanze dello schema con diversi valori dei segnali nei diversi istanti temporali.

Le tipiche descrizioni SW (programmi, diagrammi di flusso) seguono generalmente un paradigma imperativo ed evidenziano la successione temporale delle operazioni, con i vari percorsi alternativi, perdendo completamente gli aspetti di spazialità. Ciò rende talvolta più semplice la comprensione di un'evoluzione, dato che evidenzia solo gli aspetti in ogni istante più rilevanti: utile nei casi in cui sia predominante l'aggregazione temporale di eventi rispetto a quella spaziale.

La visione HW è basata su "operatori" continui, mentre quella SW su "operazioni", cioè attivazioni, pensate come istantanee, di operatori.

Le due visioni sono complementari, e quindi entrambe interessanti nei rispettivi domini di applicabilità. Per quanto detto sull'equivalenza temporale esiste una fascia di prestazioni per le quali si può scegliere tra i due approcci, e comunque può essere interessante indagare sulla possibilità di utilizzare le **tecniche descrittive** (o i paradigmi) tipiche di un approccio anche per rappresentare **funzionalità implementative** da realizzare mediante l'approccio complementare.

4.3 LIVELLI FUNZIONALI

In questo paragrafo si fa un'analisi delle varie risorse e del loro contributo alle prestazioni temporali complessive di un sistema.

4.3.1 HARDWARE

Intendiamo qui per HW i circuiti elettronici analogici e digitali **esterni** o **aggiuntivi** ai calcolatori, che verranno invece trattati nel punto seguente.
L'HW presenta caratteristiche interessanti che in alcuni casi ne fanno la risorsa di scelta obbligata.

-- Interagisce direttamente con fenomeni fisici di tipo elettrico o elettromeccanico, anche ad elevati livelli energetici.

-- Come corollario del punto precedente l'HW è in grado di gestire informazioni analogiche (fenomeni continui) e di effettuare su di esse elaborazioni di tipo integro- differenziale.

-- I tempi in gioco, soprattutto nelle funzioni tipicamente legate alle informazioni (quindi a bassi livelli energetici) sono molto più brevi (da 2 a 5 ordini di grandezza, cioè nell'ordine dei ns) che per altri tipi di risorse SW.

-- Il comportamento è caratterizzabile nei valori delle grandezze e nei tempi di risposta con buona od ottima precisione.

-- I "modelli di guasto" (cioè i possibili comportamenti) possono essere abbastanza precisati.

La scarsa flessibilità, la ridotta complessità delle elaborazioni implementabili e il costo di riproduzione, tipici dei dispositivi HW, ne limitano l'uso allo stretto indispensabile.

4.3.2 MACCHINA CALCOLATORE

Come calcolatore intendiamo qui la parte HW strettamente connessa con la capacità di eseguire istruzioni memorizzate in "linguaggio macchina", e quelle interfacce verso il mondo esterno che sono considerate di dotazione intrinseca.

Gli elementi rilevanti rispetto alle prestazioni di un calcolatore sono numerosi, complessi e fortemente interagenti, così da rendere difficile una loro valutazione oggettiva e generale, soprattutto in applicazioni con notevoli componenti di I/O, come sono generalmente quelle di automazione di macchine.

Inoltre esistono svariati tipi di microcontrollori e microcalcolatori su singolo chip, così da rendere vasta la casistica e non ben marcato il confine tra hardware esterno, dispositivi "intelligenti" e veri e propri calcolatori. La classificazione è spesso più condizionata dalle caratteristiche che si vogliono evidenziare (ad es. la programmabilità) che dalla effettiva struttura logica dei dispositivi.

Non è casuale che il problema del "*benchmarking*" (strumenti e tecniche di misura delle prestazioni) sia tuttora oggetto di studi e che le proposte di tecniche di misura delle prestazioni siano generalmente orientate ognuna ad uno spettro di applicazioni estremamente ridotto, al di fuori del quale le misure ottenute possono essere poco significative.

Non trarremo quindi conclusioni precise ma ci limiteremo ad annotare alcune considerazioni sulle principali caratteristiche dei calcolatori, di rilievo rispetto alle loro prestazioni temporali.

-- **Velocità elementari** (clock, CPU, memorie operative, BUS).
Le velocità elementari costituiscono un fattore moltiplicativo di quasi tutti i tempi di esecuzione SW.
Non si deve confondere la frequenza del clock con il ritmo di elaborazione della CPU, come pure non vanno confusi il clock che determina il ritmo HW (con periodo dell'ordine delle decine di ns), da quello che nei sistemi in tempo reale effettua le temporizzazioni SW (detto spesso *Real Time Clock* con periodo di diverse centinaia di μs).
-- **Registri, ALU e Set di istruzioni**
Le architetture CISC (*Complex Instruction Set Computer*) prevedono istruzioni anche lunghe e complesse, che possono provocare latenze nella gestione delle interruzioni. Diverse organizzazioni dei registri e del set di istruzioni possono consentire realizzazioni più o meno efficienti di particolari operazioni di accesso o trasformazione dei dati.
-- **Codifica e *fetch* delle istruzioni**
Il tempo di esecuzione delle singole istruzioni, la sua regolarità e soprattutto la sua predicibilità dipende molto dalla codifica e dalle modalità di *fetch* delle istruzioni adottata dalla CPU. In particolare le tecniche di ***prefetch*** basate su un parallelismo in *pipeline* tra i circuiti della CPU che accedono alla memoria e quelli (ALU) che elaborano i dati, determinano una variabilità dei tempi di esecuzione di una stessa istruzione in dipendenza dalla sequenza di istruzioni precedenti. Con questi processori non è consigliabile basare le temporizzazioni sui tempi di esecuzione delle istruzioni.
Decisamente più rilevante, e talora inaccettabile nei sistemi real-time, è l'effetto introdotto dai meccanismi con **memorie *cache*** che, in caso di *miss* (istruzione o dato non presente nella *cache*), danno luogo a tempi anche di ordini di grandezza maggiori rispetto al caso di *hit* (dati trovati nella *cache*). *Prefetch e cache* sono tecniche che aumentano la velocità media (*throughput*) a scapito della predicibilità e regolarità.
-- **MMU - Memory Management Unit**.
I meccanismi di trasformazione tra indirizzi logici visti dai programmi e indirizzi fisici in memoria, che caratterizzano i processori più potenti, presentano una duplice valenza rispetto ai nostri problemi.
Da un lato è molto interessante l'effetto di robustezza del SW dato dalla possibilità di controllare i diritti di accesso di ogni processo a ben delimitate aree di indirizzi,

evitando che errori del software o occasionali malfunzionamenti portino a corrompere informazioni appartenenti ad altri processi.
D'altra parte però la flessibilità di indirizzamento, che può giungere all'implementazione di funzioni di memoria virtuale, non trova in molti casi possibilità di applicazione, data la collocazione statica degli indirizzi fisici delle memorie RAM, RAM Tampone e ROM destinate a particolari usi.

-- **Meccanismi di interrupt**

Una maggior flessibilità nella gestione delle interruzioni comporta, a basso livello, tempi di risposta maggiori, ma può risultare comunque globalmente più efficiente.
E' interessante la possibilità di un cambiamento di contesto (registri) automaticamente effettuato ad HW in occasione delle interruzioni.
Nei calcolatori di controllo di processo una ricca ed efficiente gestione di interrupt, con priorità e annidamento, costituisce una caratteristica qualificante.
Un'analisi più approfondita delle interruzioni e dei relativi tempi in gioco viene proposta nei paragrafi 5.5.2 e 10.6.

-- **Livelli di privilegio** di esecuzione.

Sono meccanismi implementati ad HW all'interno della CPU che consentono l'accesso a tutte le risorse solo ai livelli **sistema**, mentre limitano le operazioni legali (e generano *trap* in caso di violazione) eseguibili ai livelli **utente**.
Aumentano la robustezza del SW a parità di tempo di calcolo. Si noti però che nelle applicazioni di automazione molte operazioni tipicamente di sistema (ad es. operazioni di I/O) ricadono tra i compiti del SW applicativo.

-- **Supporti HW della CPU per il debug**

Consentono operazioni di *debug* "intrusivo" con un minimo sovraccarico rispetto ad una esecuzione SW. In particolare può essere molto utile la possibilità di generare una interruzione sincrona (*trap*) quando vengono eseguite istruzioni ad indirizzi prefissati o quando si accede in lettura o in scrittura a determinate variabili (questi punti di monitoraggio dell'esecuzione sono detti *break-point*). Le routine di risposta a queste interruzioni possono memorizzare o visualizzare direttamente interessanti informazioni dinamiche, come contenuto di registri o valori di variabili, senza un arresto prolungato dell'esecuzione dei processi applicativi che non può essere tollerato nelle applicazioni in tempo reale.

-- **Supporti per memorie gerarchiche**

Come si è accennato la gestione di memorie *cache*, ed ancor più di memorie virtuali, comporta incremento delle prestazioni medie ma, in genere, anche una notevole impredicibilità di tempi di accesso e di esecuzione, e quindi sono sconsigliate e talora inaccettabili per i sistemi di elaborazione in tempo reale stretto con granularità temporali fini (millisecondi o meno).

-- **Risorse di temporizzazione e interfacciamento**

Tutte le risorse che realizzano funzioni a livello HW portano interessanti vantaggi in termini di velocità e predicibilità di comportamento, anche se talora con qualche limite alla flessibilità. La disponibilità di un meccanismo di temporizzazione (*Real Time Clock*) in grado di generare richieste di interruzioni è elemento essenziale in tutti i calcolatori destinati ad applicazioni di controllo.

4.3.3 ARCHITETTURE DISTRIBUITE

Al livello architetturale più alto si collocano i sistemi multiprocessori e i sistemi distribuiti che vengono trattati più a fondo in capitoli successivi e di cui riportiamo qui alcuni importanti aspetti caratterizzanti.

L'aspetto principale dei sistemi di elaborazione dotati di più di una CPU è costituito dai **meccanismi di comunicazione** necessari per trasformare in "sistema" una costellazione di macchine.
Convenzionalmente vengono detti **multiprocessore** (ad accoppiamento stretto) i sistemi in cui le comunicazioni sono basate sull'accesso a risorse (generalmente le memorie) in varia misura condivise.

Memorie a doppia o multipla porta
Architetture a bus globale
Architetture a matrice di commutazione.

Sono invece detti **sistemi distribuiti** (ad accoppiamento lasco) quelli che basano le comunicazioni su esplicite trasmissioni di messaggi attraverso opportuni mezzi trasmissivi con tecniche (bit)seriali o (bit)parallele.

Connessioni punto a punto.
Bus seriali di campo.
Reti Locali (LAN = *Local Area Network*)

Un interessante caso di sistemi multiprocessore ad accoppiamento lasco è costituito dai **Transputer**. Si tratta di microprocessori relativamente potenti e corredati di 4 canali di comunicazione veloce con funzionalità HW strettamente integrata nella CPU in modo da realizzare comunicazioni rapide ed efficienti anche in architetture con molti processori.

Rispetto alle applicazioni di controllo di processo in tempo reale, le architetture distribuite richiedono una particolare attenzione sui seguenti punti, per le possibili criticità derivanti dal parallelismo fisico e dai ritardi di comunicazione.

Parallelismo fisico. Diversi elementi attivi (le CPU) evolvono contemporaneamente ognuna con un proprio ritmo, ponendo problemi di sincronizzazione e di mutua esclusione.

Accesso a risorse condivise. Occorre risolvere i possibili conflitti di accesso a risorse condivise, in modo da non introdurre eccessive incertezze sui tempi di accesso alle risorse.

Protocolli e tempi di comunicazione. Le comunicazioni richiedono protocolli adatti alle necessarie sincronizzazioni tra produttori e consumatori di informazioni, e al rilievo ed eventuale correzione degli inevitabili errori.
Oltre all'inevitabile *overhead* del protocollo si hanno ritardi di propagazione che spesso sono tutt'altro che trascurabili, e ciò rende ad esempio impossibile che una CPU "conosca" lo stato attuale di un'altra: quando infatti riceve eventuali informazioni di stato queste sono già relativamente "vecchie".

Distribuzione dei compiti. I problemi di affidabilità locale e globale e di rispetto di vincoli temporali, rendono necessario un accurato studio della distribuzione dei compiti, e relativi carichi di lavoro, alle varie CPU. Generalmente si ricorre ad una distribuzione **statica** o **semistatica** (in cui i sistemi di *back-up* sono prefissati) dei compiti.

Coordinamento e consistenza. La macchinosità dei protocolli e i ritardi di comunicazione richiedono tecniche particolari per coordinare attività parallele eseguite in tempo reale in modo da garantire coerenza di comportamento e consistenza delle informazioni condivise.
Si pensi ad esempio ad una comunicazione *broadcast* che invia a tutti i processori il valore di una grandezza che però sia ricevuta in modo errato da uno o più di essi.

Un altro esempio è costituito dal problema, ancora in fase di studio, di mantenere sincronizzati tra loro gli orologi locali a diverse CPU remote.

Confinamento degli effetti di errori e guasti. L'affidabilità di un sistema distribuito presenta interessanti doti di tolleranza ai guasti (*fault tolerance* e *graceful degradation*), ma solo se sono adottati provvedimenti che evitino la propagazione degli effetti negativi di errori o guasti.

4.3.4 SW DI BASE - Sistema Operativo

Tranne che in casi particolari di calcolatori con semplice struttura interna (microcontrollori su singolo chip) e applicazioni relativamente semplici, la "piattaforma" applicativa è costituita da una macchina virtuale ottenuta sovrapponendo alla macchina HW uno strato di funzionalità ottenuta con del software (di base) che chiameremo Sistema Operativo, anche se talora è molto semplice.

Questo sistema operativo potrà coprire tutte le funzionalità che solitamente vengono attribuite all'accezione usuale del termine, ma in alcuni casi potrà limitarsi ad un ridotto sottoinsieme di tali funzionalità.

In questi ultimi casi si parla di **nucleo** (*kernel*) o **supporto di esecuzione** (*executive*) che gestisce direttamente alcune risorse (tipicamente l'orologio RTC = *Real Time Clock*) fornendo per esse delle funzioni di livello superiore, mentre lascia alla gestione degli strati applicativi del SW le altre risorse (interfacce di I/O). Ciò consente al progettista, sia pure al costo di una maggiore complessità nello sviluppo di applicazioni, di avere una completa visibilità delle risorse e adottare tecniche specifiche ottimizzate per particolari esigenze applicative.

L'importanza dei sistemi operativi nei sistemi di elaborazione in tempo reale suggerisce di rimandare l'approfondimento di questo punto ad un successivo capitolo ad essi dedicato (cap. 10). Qui si vuole solo sottolineare che le prestazioni dei sistemi di elaborazione sono anche, e talvolta in modo determinante, influenzate dal sistema operativo adottato.

L'affidabilità di un sistema operativo costituisce una caratteristica estremamente importante. Anche per questo motivo lo spazio di personalizzazione dei sistemi operativi viene limitato alla loro **configurazione** (cioè scelta delle componenti funzionali e dei valori di alcuni parametri) mentre le "filosofie" e le "politiche" sono generalmente prefissate in modo relativamente rigido.

Occorre però notare che gli attuali sistemi operativi privilegiano tuttora più la correttezza logica che quella temporale e non si propongono di fornire elevate garanzie rispetto a requisiti di tempo reale stretto (*hard real-time*), che sono lasciati alla responsabilità del programmatore applicativo.

4.3.5 LINGUAGGI DI PROGRAMMAZIONE

Tradizionalmente le applicazioni con stringenti vincoli sulle prestazioni e fortemente orientate all'interazione col mondo esterno sono state spesso basate sull'uso del

linguaggio macchina, ovviamente nelle forme simboliche proposte da **Assembler** sempre più evoluti, che consente il controllo "diretto" di **tutte le risorse** del calcolatore.

Con la tendenza, più che giustificabile, ad utilizzare sempre meno il linguaggio Assembler, il livello di macchina virtuale effettivamente "visibile" al progettista applicativo è in larga misura caratterizzato dal linguaggio di livello "superiore" che viene adottato.

Il contributo di linguaggi "*general purpose*", come ad esempio il **C**, rispetto alle prestazioni di tempo reale si riduce sostanzialmente a caratteristiche di rientranza e ad una generica efficienza del codice prodotto. Questi linguaggi lasciano invece al software di librerie, o addirittura al SW applicativo (talora con parti scritte in Assembler), il compito di risolvere esplicitamente funzioni di temporizzazione, sincronizzazione e comunicazione tra le varie attività (processi).

Alcuni linguaggi sviluppati per la programmazione concorrente in tempo reale supportano direttamente alcuni costrutti tipici e, molto parzialmente, strutture di dati adatti per applicazioni di automazione. Tra questi citiamo Ada, Modula-2, Occam e PEARL.

Mentre nel cap. 3 sono stati discussi i vari paradigmi, qui si analizzano le caratteristiche di generici linguaggi di programmazione.
Le caratteristiche desiderabili per un linguaggio di programmazione di applicazioni industriali dipendono in buona parte dall'impostazione culturale e, perchè no, dai gusti del progettista applicativo. Nel seguito ne elenchiamo alcune, ricordando però che un buon linguaggio di programmazione deve essere ben supportato e diffuso, altrimenti a poco valgono le sue doti "concettualmente interessanti".

Variabili e strutture dati.

Classi di allocazione in memoria con attributi

-- Deve essere prevista l'allocazione statica anche di variabili locali delle procedure, in modo che esse mantengano il loro valore tra una chiamata e la successiva.
(Es. - Stato di un automa - Accumulo dell'integrale in un PID).

-- Alcune variabili usate come parametri di configurazione (coefficienti, soglie, tempi di *time-out*, ecc.) devono poter essere dichiarate come **persistenti**, cioè dotate di una copia del loro valore mantenuta in memoria non volatile e coerentemente aggiornata.

-- Alcuni parametri possono essere molto importanti e la loro conservazione è **critica**: potrebbero essere mantenuti in duplice copia con *checksum* che consente di verificare se e quale delle due copie è corretta.

-- Devono essere definibili **tabelle contenenti valori fissi** ("costanti" in senso matematico) da collocare su memorie ROM.

-- Deve poter essere specificato, possibilmente in modo parametrico, **l'indirizzo fisico** di allocazione delle variabili perchè si possono avere memorie operative disomogenee (RAM, EPROM, memory mapped I/O).

-- Può essere necessario specificare che l'allocazione in memoria presenti **contiguità fisica** degli indirizzi (tipicamente per le tabelle), come pure in alcuni casi è desiderabile una **contiguità logica** di variabili collocate in diverse aree di memoria. Ad es. per aggregare come campi di uno stesso record variabili dinamiche collocate in RAM, variabili parametriche in RAM TAMPONE, variabili fisse in ROM)

Tipi e conversioni

Devono essere previsti tipi predefiniti come

`interi senza segno` (Byte, Word),
valori numerici in `virgola fissa`,
valori numerici adatti ad esprimere `tempi relativi`
valori numerici per i `tempi assoluti`.

E' molto comodo poter definire tipi corrispondenti alle unità di misura di grandezze fisiche e relativi multipli e sottomultipli. (pico, nano, micro, milli, kilo, mega, giga, tera).

Esempi:

```
float (W=N*m/s)     P;   potenza
float (m/s)         V;   velocità
float (N)           F;   forza
```

Verifiche sui tipi.

Un rigoroso *type-checking* è ormai considerato caratteristica tipica dei buoni linguaggi. Nelle nostre applicazioni è utile una verifica rigorosa ma anche una ampia libertà di dichiarare esplicitamente violazioni delle regole di compatibilità tra tipi quando ciò sia necessario (ad es. valori `integer` letti come due byte da due porte di ingresso di 8 bit ognuna).

Verifiche dimensionali e di compatibilità

Potrebbe essere prevista una *verifica dimensionale* nelle operazioni aritmetiche, nel senso che si verifica che le unità di misura della variabile a cui è assegnato il risultato di un'espressione sono compatibili con quelle derivanti dai calcoli dell'espressione. Ad es.

```
P(potenza) = F(forza) * V(velocita).
```

Operatori.

Sono necessari operatori logici (AND, OR, NOT, XOR) anche del tipo bit a bit, di shift dei bit di N posizioni, e operazioni su campi di bit e parole.

Controllo di parte intera e frazionaria di variabili `fixed point`.

Operazioni su variabili temporali (lettura, somma, differenza, confronto)

Costrutti di controllo.

Generalmente i costrutti previsti dagli attuali linguaggi sono sufficienti. Potrebbe essere utile disporre di un meccanismo di controllo sul numero di ripetizioni dei cicli (da cui dipende il tempo totale di esecuzione) con garanzia che non si superi una soglia prefissabile.

Un particolare costrutto potrebbe essere la `regione_critica`, variante della normale sequenza, ma protetta da mutua esclusione.

Costrutti di temporizzazione, e per la concorrenza.

Questi costrutti mancano totalmente nei classici linguaggi sequenziali, mentre sono più o meno previsti per i linguaggi adatti all'esecuzione concorrente.

Ad esempio potrebbero essere utili costrutti dei tipi seguenti.

```
wait (tempo_relativo)
at (tempo_assoluto)
every (tempo_relativo) do
when (evento)
signal (evento)
```

Sottoprogrammi.

Oltre alle normali procedure e funzioni devono essere dichiarabili particolari procedure dei tipi:

```
procedura processo
```

```
procedura interrupt
procedura privilegioN
```

Deve essere inoltre possibile specificare l'attributo `reentrant` per le procedure (o funzioni) che debbano essere rientranti.

E' molto importante notare che alcune rilevanti caratteristiche di un ambiente di programmazione sono spesso dovute al particolare compilatore più che al linguaggio in sè. Il livello qualitativo e l'affidabilità degli attuali compilatori hanno raggiunto valori ottimi a prezzi d'acquisto molto ridotti, mentre poco ancora si è fatto per rendere più flessibili e personalizzabili questi strumenti, e tanto meno per ottenere da essi indicazioni sulle prestazioni temporali del codice eseguibile da essi prodotto.

Modularità.

In generale devono essere previsti meccanismi che facilitano la corretta programmazione "*in the large*", in modo da realizzare applicazioni complesse con l'aggregazione modulare di diverse parti, eventualmente anche scritte in diversi linguaggi.

Disponibilità di macroespansione.

Le **macro** costituiscono un interessante meccanismo di personalizzazione di un linguaggio, consentendo al programmatore di richiamare in modo chiaro e conciso aggregazioni sintattiche e sequenze di operazioni. Un buon macroespansore deve concedere flessibilità anche nel formato testuale, così da consentire un effettivo incremento di chiarezza con l'uso delle macro.

Ad esempio la verifica se un interruttore (sw1) collegato al bit 2 della porta 80H è chiuso (ON) è pensata dal progettista come elementare, ma richiede la sequenza di operazioni:

```
input (temp, 80H)          lettura del byte
and_bit (temp, 04H)        maschera bit 2
sw1_ON = (temp not= 0)
```

Questa sequenza potrebbe essere racchiusa in una macro chiamata come SW1_ON a tutto vantaggio di concisione e chiarezza.

Compilazione condizionata.

La compilazione condizionata consiste nel verificare il valore di verità di un'espressione di guardia ad una sezione di codice e nel procedere alla traduzione di tale porzione solo se l'espressione assume valore *true*. Questa possibilità facilita molto, ad esempio, la manutenzione di un programma con inserite parti utili in fase di debug che vanno eliminate nella versione di esercizio, ma che devono essere reinserite per la nuova verifica dopo ogni modifica o ricerca di errori.

Si hanno anche interessanti possibilità per i programmi destinati a girare su diverse piattaforme e che devono quindi contenere tutte le varianti di quelle parti che sono dipendenti da HW, e che vanno introdotte o meno a seconda della piattaforma *target* utilizzata..

Controllo sul codice prodotto.

Oltre alla compilazione condizionata è utile poter esercitare altri controlli sul codice prodotto dal compilatore, abilitando o disabilitando l'ottimizzazione in tempo e/o in spazio, e l'inserimento automatico di funzioni di *check* di consistenza dei dati (*checksum*) supero della pila (*stack overflow*), ecc..

Valutatori di tempi e spazi.

Un aspetto molto importante per le applicazioni di controllo è la valutazione dei tempi di esecuzione delle varie procedure e dello spazio in memoria (nelle varie aree RAM, ROM, ecc.) occupato dalle variabili. Queste informazioni dovrebbero essere fornite dal compilatore, eventualmente su richiesta.

Diagnostica (warning).

I messaggi diagnostici dei compilatori dovrebbero essere ricchi di informazioni, non solo ovviamente per indicare errori fatali, ma anche per evidenziare situazioni accettabili sintatticamente ma che è bene siano note al programmatore, come ad esempio variabili dichiarate e non citate, variabili non inizializzate, accessi ad array con indice non garantito, cicli senza limite superiore di ripetizioni, ecc.

Cross-reference.

Tra le informazioni fornite dai compilatori sono certamente interessanti le tabelle dette di *cross-reference*. Tipicamente sono utili riferimenti incrociati tra dove un simbolo (ad es. una variabile o una procedura) è dichiarato e dove è utilizzato. In ambiente multitask sono interessanti anche riferimenti di interazioni tra *task*, *task* e risorse (semafori, messaggi, ecc.), ecc..

E' molto importante notare che molte delle caratteristiche sopra citate, e che se disponibili renderebbero più facile la stesura di programmi corretti, possono essere in qualche modo, talora complicato, realizzate anche con i normali linguaggi di programmazione, come il C.

4.3.6 LIBRERIE

Le librerie costituiscono un'interessante integrazione delle funzionalità del sistema operativo sottostante e del linguaggio utilizzato, sollevando il programmatore applicativo dalla necessità di risolvere esplicitamente problemi di basso livello concettuale ma spesso complicati e richiedenti un'approfondita conoscenza di intricate funzionalità HW o di dettagli di basso livello.

Oltre alle librerie di normale corredo per i più diffusi linguaggi di programmazione, si vanno diffondendo librerie aggiuntive (o parzialmente sostitutive) che coprono in modo organico una serie di funzionalità interessanti e specifiche di vari settori applicativi, così da costituire una componente essenziale della "personalità" dei sistemi di sviluppo delle applicazioni (librerie grafiche, librerie di calcoli matematici, librerie per gestione di acquisizione dati, ecc.).

E' interessante notare che si è diffusa la pratica, da parte dei costruttori di interfacce di I/O, di fornire delle librerie specificatamente dedicate a coprire le funzioni tipiche di tali interfacce, in modo abbastanza generale da risultare utilizzabile nella maggior parte delle applicazioni. In particolare la disponibilità di queste librerie risulta estremamente vantaggiosa per i progettisti che debbano utilizzare con modalità standard interfacce anche molto complesse.

Esempi.

Interfacce da telecamera per acquisizione di immagini statiche o in movimento.

Interfacce per collegamento in rete.

Interfacce per comunicazioni telefoniche automatiche (es. modem).

Interfacce per acquisizione di segnali analogici.

Interfacce per controllo assi, tipiche di applicazioni di robotica, macchine a controllo numerico, macchine operatrici, ecc.

La tecnica basata sull'uso di librerie è così significativa che alcuni nuclei real- time sono addirittura realizzati proprio sotto forma di librerie di funzioni da collegare (*link*) al SW applicativo, in modo da ottenere alla fine un programma eseguibile autosufficiente, cioè che gira direttamente sulla macchina "nuda", e che contiene al suo interno le funzioni di temporizzazione e di sincronizzazione tra processi.
L'elevato grado di diffusione e riutilizzo di molte librerie, oltre che la professionalità di chi le ha prodotte, conferiscono ad esse un ottimo livello qualitativo a costi, per ogni singola copia, piuttosto contenuti.
Come tutti gli strumenti che semplificano la vita al programmatore applicativo, le librerie limitano però la visibilità dei meccanismi e quindi richiedono una attenta considerazione per valutare le prestazioni ottenute dal loro impiego.

Spesso, con eventuale sovrapprezzo, sono disponibili anche i **codici sorgenti** delle librerie: viene in tal modo offerta al programmatore applicativo una maggiore visibilità dei meccanismi adottati ed un maggior grado di libertà per eventuali personalizzazioni.

Nota a proposito delle modifiche alle librerie.
E' importante dedicare molta attenzione all'eventuale decisione di modificare librerie acquisite, soprattutto per i seguenti motivi.

- Una conoscenza incompleta delle librerie, che generalmente costituiscono un corpo organico, può portare ad introdurre errori che eventualmente si possono manifestare in funzioni apparentemente scorrelate con quelle modificate. Questi errori non sono facili da individuare e correggere.
- Le modifiche devono essere riportate in ogni eventuale nuova release delle librerie che si renda disponibile e venga adottata.

4.3.7 SW APPLICATIVO

Al SW applicativo rimane ovviamente la responsabilità di completare la realizzazione delle applicazioni con le funzionalità più o meno specifiche ma comunque non risolte dalle risorse precedentemente citate e che siano disponibili.
E' importante quindi notare che il programmatore di applicazioni in tempo reale verrà alleggerito da compiti di dettaglio se riesce a trovare strumenti che se ne facciano carico, ma manterrà sempre, almeno in parte, la responsabilità di un uso semanticamente e temporalmente corretto di tali strumenti che dovrà quindi conoscere molto bene.

Questo spiega in parte la scelta di molti progettisti che adottano strumenti meno potenti di altri disponibili ma che consentono un maggior controllo delle risorse o almeno una maggiore visibilità degli aspetti semantici; ciò è particolarmente vero quando si

debbano dominare con sufficiente padronanza i **tempi di esecuzione** delle elaborazioni e le gestioni di anomalie (*exception handling*).

La notevole libertà delle scelte, la complessità di molte applicazioni, ma soprattutto la difficoltà a riutilizzare per la massima parte programmi già ben collaudati, fanno del SW applicativo il livello più critico rispetto ai costi di sviluppo e all'affidabilità dei sistemi di automazione.

NOTA. - Sui livelli funzionali HW e SW sopra descritti si possono fare considerazioni rispetto alla "progettazione integrata" e alla scelta tra "comperare o fare".

- **Progettazione integrata** (*HW/SW Codesign*)

Uno dei problemi di progetto è la distribuzione delle funzioni tra livelli funzionali, in particolar modo tra livelli adiacenti.
Questa distribuzione assume aspetti di particolare rilievo soprattutto a cavallo del confine di massima disomogeneità, cioè tra HW e SW.
Interessanti articoli sul *Codesign* sono pubblicati su:

IEEE - Design and Test of computers. Sett.93.
IEEE - Design and Test of computers. Dic.93.

Il *Codesign* sta assumendo interesse e importanza crescente, soprattutto con l'attuale sviluppo raggiunto dagli strumenti CAD per il progetto di circuiti integrati (tra cui il linguaggio di descrizione VHDL).

- **Comperare o fare** (*buy or make*)

Ogni nuovo progetto costituisce un'opportunità per il potenziale "riuso" di soluzioni già disponibili o acquistabili.
Il problema di scegliere tra il riutilizzo di soluzioni acquistabili o la realizzazione *ex novo* di una soluzione specifica si presenta molto articolato tra diversi compromessi che qui non analizziamo ulteriormente.

4.4 BIBLIOGRAFIA

Alan Burns, Andy Wellings
Real-time systems and their programming languages
Addison Wesley Pub. Co. 1989