Honeywell
Honeywell Information Systems Italia
Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA
Via Viotti, 5 - Milano

8

OFTWARE NOTE DI SOFTWARE

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

NOTE DI SOFTWARE

8

Ottobre - Dicembre 1979

Indice:

Autorizzazione del Tribunale di Milano n. 87 del 23 febbraio 1977

QUESTO NUMERO ...

La programmazione di sistemi interattivi, la valutazione delle prestazioni dei sistemi di calcolo, la 'language directed computer design' e l'uso di tecniche di programmazione strutturata in linguaggi di livello molto basso sono i temi a cui si rivolge questo ottavo numero di NOTE DI SOFTWARE.

Nel primo lavoro, V. Cajani descrive gli aspetti fondamentali del TPS (Transaction Processing System), un componente del GCOS del Livello 62 Honeywell, sviluppato allo scopo di facilitare all'utente la progettazione e la realizzazione di applicazioni interattive di tipo transazionale. L'articolo, di tipo intoduttivo, illustra la struttura generale di un'applicazione TPS, e i vari passi che l'utente deve seguire nello sviluppo di un'aplicazione transazionale, mettendo in evidenza le caratteristiche di facilità d'uso del prodotto. Fornisce, inoltre, alcune indicazioni sulla sua struttura interna.

Nel secondo lavoro, F. Parodi, G. Zanone, C. Firpo e M. Calvini illustrano alcune esperienze di valutazione delle prestazioni effettuate su un elaboratore di grandi dimensioni, il sistema Honeywell L66 dell'Ansaldo di Genova. La esperienza ha utilizzato un monitor software, SYMON, opportunamente integrato da programmi appositamente sviluppati. I risultati delle misure vengono visualizzati mediante plotter, utilizzando, tra l'altro, i 'grafici di Kiviat', che forniscono una visione riassuntiva particolarmente leggibile.

Nel terzo lavoro, L.G. Sartori propone un'architettura di sistemi di calcolo in cui la rappresentazione dei dati comprende informazioni sul loro tipo (architettura 'tagged'). Così, il codice operativo di un'istruzione di macchina verrebbe opportunamente specializzato dall'hardware, al momento della esecuzione, a seconda del tipo di dato con il quale interagisce: il programmatore assembler vedrebbe allora, ad esempio, non più istruzioni del tipo 'add fixed', 'add floating', 'add packed', ma un'unica istruzione 'add', che ritorna nella sua forma tradizionale solo nell'ultima fase del ciclo di esecuzione.
Il lavoro discute i vantaggi ed analizza il disegno della nuova architettura, proposta in diverse varianti, tenendo conto delle possibilità offerte dall'impiego dei microelaboratori.

Infine, M. Gualzetti, M. Motta e L. Squizzato descrivono un insieme di strumenti realizzati allo scopo di rendere possibile l'uso di strutture di controllo di alto livello in linguaggi di programmazione molto vicini alla macchina hardware: linguaggi firmware e linguaggi assembler per microprocessori.

CONTRIBUTI TECNICI

INTRODUZIONE AL TPS

Vittorio Cajani
Honeywell I.S.I., Pregnana Milanese

Riassunto

In questo lavoro vengono descritti gli aspetti fondamentali del TPS (Transaction Processing System), un componente del GCOS del Livello 62 Honeywell, sviluppato allo scopo di facilitare all'utente la programmazione di applicazioni interattive.

Abstract

This paper presents the main aspects of TPS (Transaction Processing System), a Honeywell GCOS Level 62 component, developed to allow an easy design and programming of interactive applications.

1. PERCHE' IL TPS

Considerando le statistiche sugli utenti EDP risultano evidenti due fatti.

Il primo è che l'uso di applicazioni interattive si va diffondendo rapidamente: negli USA oltre l'80% dei centri di calcolo ha almeno un terminale collegato. L'Europa segue a ruota. Inoltre il numero medio di terminali collegati ad una CPU va crescendo di pari passo e la quasi totalità dei nuovi utenti EDP richiede di poter eseguire processing interattivo oltre al tradizionale processing batch.

Il secondo fatto è il variare del rapporto tra il costo dell'hardware ed il costo del software. A parità di prestazioni offerte il costo dell'hardware diminuisce; il calo investe quasi tutti i componenti di un sistema: dalla CPU, ai dischi, ai terminali. Di contro, il costo di sviluppo del software cresce parallelamente alla mancanza di analisti e programmatori (nei soli USA si stima che ogni anno il divario tra domanda ed offerta di specialisti software cresce di 15.000 unità).

Questa tendenza del mercato impone di offrire con l'hardware un software di base che renda facilmente sviluppabili le applicazioni, e particolarmente quelle interattive. La riduzione dei costi del software è un'esigenza irrinunciabile non solo per l'utente, ma anche per le software houses e per gli stessi produttori hardware che producano anche software di base.

Questa premessa spiega l'obiettivo del TPS (Transaction Processing System), componente del GCOS del Livello 62: l'«ease to use» per la programmazione di applicazioni interattive.
Del resto l'«ease to use» è particolarmente importante nell'interattivo, perchè le problematiche sono più complesse (o quanto meno non altrettanto acquisite) di quelle delle applicazioni batch. In particolare, per quel che riguarda:

— il disegno dell'applicazione, in cui devono essere contemporaneamente eseguite le transazioni richieste dai terminalisti

— la necessità di permettere a transazioni eseguite in parallelo accesso ad una base di dati comune (gli archivi dati "in linea" all'unità centrale) controllando le interferenze negli accessi

— la programmazione degli scambi di messaggi coi terminali, periferiche estremamente sofisticate e con procedure di interfaccia non ancora ben standardizzate.
Periferiche che più potenzialità hanno, più risultano difficili da utilizzare al massimo delle loro facilities hardware e firmware

— le tecniche di integrity e di recovery degli errori

— la security dei dati degli archivi, necessaria data la decentralizzazione dell'accesso al sistema.

2. IL TPS E' «EASE TO USE»

Per raggiungere l'obiettivo il prodotto si basa su concetti semplici ed intuitivi ed è stato disegnato in maniera completamente coerente con essi.
Propone all'utente una logica di disegno e sviluppo di un'applicazione interattiva la più elementare possibile, anche a costo di minore flessibilità.
Offre al programmatore statements ad alto livello per permettere un interfaccia "logico" coi terminali e dà soluzioni implicite ai problemi di integrity e security.
Tutto ciò risulta evidente, per chi è familiare con le problematiche dell'interattivo, dai paragrafi seguenti che descrivono per sommi capi le caratteristiche di novità del prodotto.

2.1. Applicazione TPS

Un'applicazione di elaborazione dati interattiva si basa essenzialmente su tre tipi di elementi:

— una rete
— un elaboratore centrale
— un set di funzioni attivabili dai punti periferici della rete.

Nella sua schematizzazione più semplice, una rete è costituita da un elaboratore centrale (CPU più memorie secondarie) e da un insieme di linee ad essa collegate, ciascuna facente capo ad uno o più terminali. Il terminale è il punto periferico di raccolta/distribuzione delle informazioni; può essere unità di input/output o di solo output (fig. 1).

Le memorie secondarie dell'elaboratore centrale devono essere ad accesso veloce, e quindi dischi magnetici; questo per esigenze di tempi di risposta brevi e di possibilità di concorrenzialità di accessi. Non sono invece necessarie per l'applicazione interattiva altri tipi di periferiche, connesse all'elaboratore centrale.

Le funzioni eseguibili dai punti periferici della rete sono le procedure di dialogo terminalista/sistema disponibili per una data applicazione. Con riferimento alla terminologia batch, di seguito verranno chiamati Interactive Transactional Program (ITP).

L'insieme di una rete, degli archivi dati e degli ITP costituisce un "environment" TPS.
Il primo passo nella definizione di una applicazione TPS è costituito dalla descrizione dell'environment.

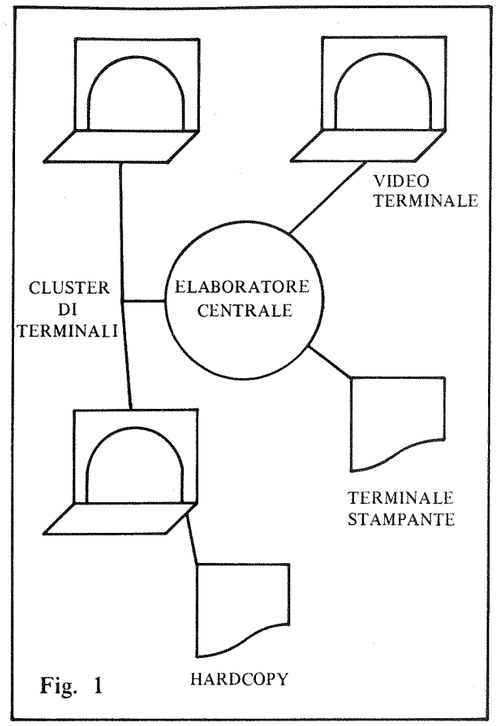


Fig. 1

2.2. ITP

L'ITP è la funzione richiamabile dal terminalista, con cui l'ITP stesso deve pilotare il dialogo una volta attivato. Caratteristica base di un ITP è quella di essere un programma conversazionale, disegnato in termini di procedure di dialogo o cicli logici di dialogo. Frasi del dialogo sono i messaggi scambiati con il terminalista.

Per semplificare al massimo la struttura di un ITP in una rete comunque complessa, il TPS stabilisce una biunivoca associazione tra ITP e terminale che lo ha richiamato: se più terminali richiedono contemporaneamente lo stesso ITP, si può intendere che siano eseguite dal sottosistema copie logicamente distinte dello stesso ITP. Un ITP opera quindi su dati caratteristici del terminale. Le periferiche accessibili da un ITP sono mostrate in fig. 2.

Keyboard, video ed hardcopy sono periferiche dello stesso terminale che ha richiamato l'ITP: il terminale "master".

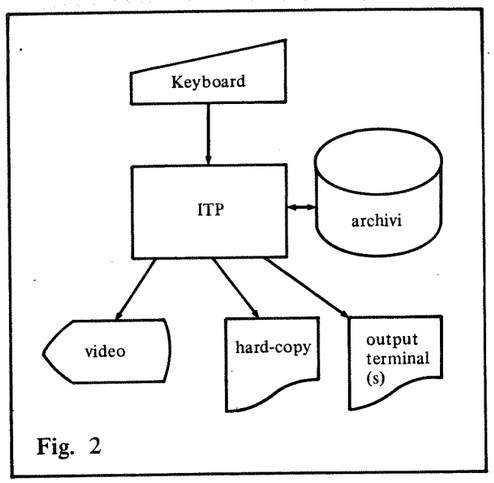


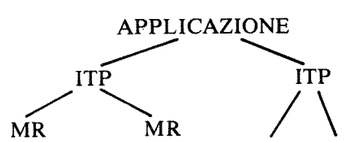
Fig. 2

2.3. Message Routine

L'informazione primaria in una applicazione interattiva è il messaggio inviato da terminale. E' primaria nel senso che determina ogni azione dell'attività real time, in continuo stato di attesa di messaggi dalla rete.
Supponendo di poter identificare ogni messaggio e di classificarlo come appartenente ad un certo "tipo", si può dire che un'attività TPS elabora un insieme predefinito di tipi di messaggi.
Ad ogni tipo di messaggio si può biunivocamente associare il processing che richiede; il codice che esprime questo processing è la Message Routine (MR).

Un insieme logicamente connesso di Message Routines costituisce un ITP. La logica che connette le MR è quella del ciclo di dialogo e ad ogni frase del ciclo di dialogo è associata una MR.
In altri termini, la MR è l'unità base di processing, l'ITP quella di dialogo sistema/terminalista.

Ne segue la struttura gerarchica di un'applicazione:

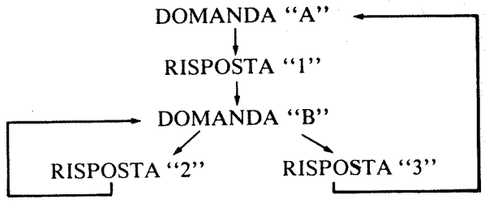


Questa gerarchia propone il disegno e lo sviluppo di un'applicazione in termini di elementi distinti, connessi tra loro dalla descrizione dell'environment e dai cicli di dialogo.
Più avanti verrà indicato come ciò consente una programmazione integrata di ITP e MR, con notevoli benefici per la loro trasferibilità in environment diversi.

2.4. Ciclo logico di dialogo

E' stato detto che la caratteristica principale di un ITP è quella di essere un programma conversazionale, cioè che conduce un dialogo con il terminalista che lo ha richiamato. Si è detto che frasi del dialogo sono i messaggi e che le MRs sono le procedure di elaborazione dei diversi tipi di messaggi.

Punto di partenza per la realizzazione di un ITP è quindi quello della definizione del dialogo che esso deve condurre col terminalista.
Ad esempio:



è la schematizzazione di un semplice ciclo logico di dialogo. Le domande sono poste dall'ITP, le risposte fornite dal terminalista. Ci possono essere più tipi di risposte alla stessa domanda. Il dialogo può essere ripetitivo, come dimostrano le frecce di ritorno; in questo senso è detto ciclo. Per un ITP di inquiry su un archivio magazzino, il ciclo di dialogo precedente può essere:

DOMANDA "A" : NOME PRODOTTO SU CUI IL TERMINALISTA VUOLE INFORMAZIONI

RISPOSTA : NOME PRODOTTO

L'ITP riconosce il nome del prodotto, accede alle informazioni contenute negli archivi, visualizza al terminale una descrizione del prodotto e propone la

DOMANDA "B" : TIPO DI INFORMAZIONI DI DETTAGLIO SUL PRODOTTO

RISPOSTA : NESSUNA O TIPO DI DETTAGLIO
- se la risposta è "NESSUNA" viene riproposta la domanda iniziale "A"
- se è una richiesta di dettaglio l'ITP accede alle informazioni contenute negli archivi, visualizza al terminale i dettagli richiesti e ripropone la domanda di tipo "B".

Definito un ciclo logico di dialogo, risultano pure definiti i tipi di messaggi che l'ITP deve elaborare; nell'esempio, le risposte di tipo "1" e "2". Noti i tipi di messaggi, vanno associate ad essi le MR. Con riferimento all'esempio precedente, andranno definite le MR che elaborano i due tipi di risposte, più eventualmente una di inizializzazione. Questa è necessaria se al momento di lancio dell'ITP oltre che porre la domanda "A" è necessaria qualche azione di inizializzazione sui dati; in caso contrario è sufficiente definire l'ITP associandogli una domanda di inizio dialogo.

Dal ciclo di dialogo precedente si ricava una struttura dell'ITP del tipo rappresentato in fig. 3.

Si noti che la differenza tra le risposte di tipo "2" e "3" è definita dal contenuto del messaggio digitato dal terminalista (è ovvio che si potrebbero avere più message routines, ognuna per elaborare un tipo di dettaglio richiesto; sarebbe sufficiente estendere l'esempio moltiplicando le uscite in risposta a "B").

3. LE CARATTERISTICHE DI UN ITP E DEI SUOI COMPONENTI

Segue da quanto detto che sviluppare un ITP significa definirne il ciclo di dialogo col terminalista, descrivere i messaggi che esso invia e codificare le MR che elaborano le risposte del terminalista ai vari tipi di messaggi di inquiry.

Va sottolineato che la programmazione di un ITP è scissa nella programmazione dei messaggi e delle MR:

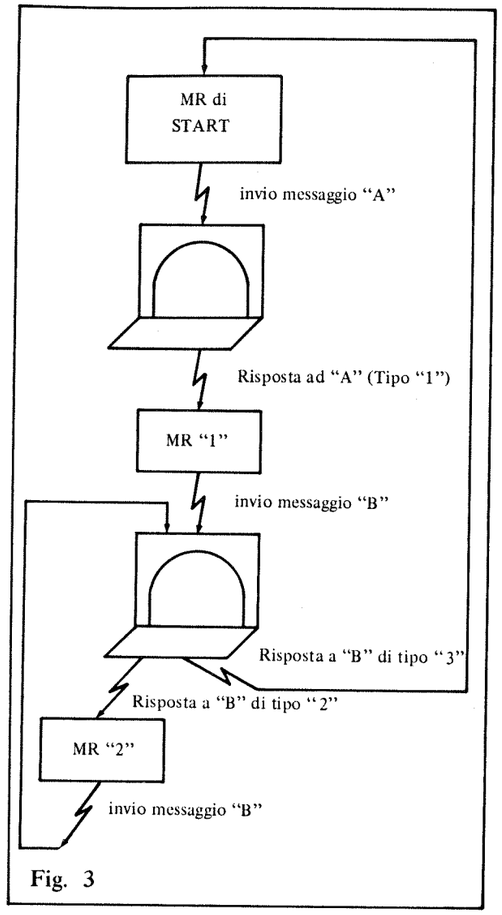
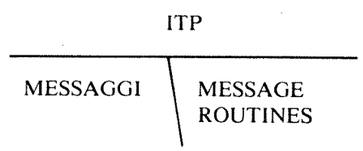


Fig. 3



Le due attività non sono scorrelate: la MR deve conoscere la struttura del messaggio che riceve. E' quindi buona regola di programmazione definire prima i messaggi e poi le Message Routines.

3.1. I messaggi

Il TPS distingue tra due classi di messaggi: quelli che propongono una domanda al terminalista (messaggi di inquiry) e quelli che sottopongono al

terminalista delle informazioni (messaggi di output). La distinzione non è rigida: un messaggio di inquiry può contenere anche delle informazioni. A seconda del tipo di ITP, una delle due classi può essere preponderante: in ITP di data entry i messaggi di informazione sono utilizzati solo per notificare errori; in ITP di puro inquiry i messaggi di interrogazione servono solo a guidare le richieste del terminalista (come nell'esempio precedente).
Non può comunque essere definito un ITP senza messaggi di interrogazione, cioè di guida del dialogo.

E' inoltre limitazione essenziale alle semplicità del disegno di un ITP e delle MR l'imporre che una MR possa inviare un solo messaggio di inquiry; questo perchè le MR che ricevono la risposta possano elaborare solo messaggi "semplici".

3.2. I modi di scambio messaggi

Dovendo entrare nel merito della definizione dei messaggi, va chiarita la distinzione base che il TPS propone nel "modo di scambio":

- il "form mode"
- il "line mode".

I modi di dialogo dipendono, più che da scelte dell'utilizzatore, dalle funzionalità hardware offerte dai terminali:

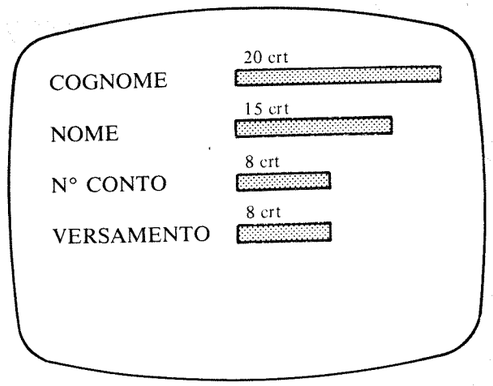
- il form mode è utilizzabile su terminali video sofisticati e con un minimo di intelligenza
- il line mode su terminali economici, video o stampanti, assimilabili nell'utilizzo a telescriventi.

3.2.1. Il form mode

In form mode il video è prospettato al terminalista in modo simile ad un questionario da compilare: la maschera.

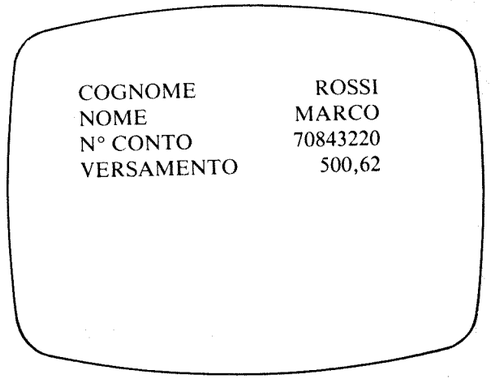
La maschera divide lo schermo in parti da compilare, e quindi accessibili al terminalista, ed in parti che guidano il terminalista alla compilazione delle zone variabili e quindi sono definite come fisse, inaccessibili al terminalista.

Una maschera su video può essere:



dove le zone tratteggiate sono le sole accessibili al terminalista. Questi può compilarle nell'ordine che vuole, saltarne, correggerne e così via; solo quando avrà deciso di avere compilato correttamente la maschera, trasmetterà il messaggio, che comprenderà tutti i caratteri visualizzati nelle zone variabili, più alcuni caratteri, aggiunti dall'hardware del terminale, che indicano il passaggio dal testo di una zona variabile a quello della successiva.
Questo è il messaggio che viene trasmesso in linea.

Supponendo che il terminalista abbia compilato così la maschera:



il messaggio in linea sarà:

CS ROSSI $\frac{H}{T}$ MARCO $\frac{H}{T}$ 70843220 $\frac{H}{T}$ 500,62 CS

Dove CS è un carattere di controllo della trasmissione, e $\frac{H}{T}$ è un carattere di separazione delle zone variabili.

Questo messaggio viene passato alla MR incaricata di elaborarlo dopo un "filtro" eseguito da routines dell'esecutivo di TPS: questo al fine di convalidare e normalizzare il messaggio.
La convalida consiste in una serie di controlli del tipo:

— esito della operazione di linea (per es., corretta trasmissione del messaggio)
— convalida del contenuto di certi campi del messaggio di risposta, ad esempio che un campo numerico contenga solo cifre e segni aritmetici, che un campo definito come "must be filled" sia stato compilato dal terminalista, eccetera.

La normalizzazione consiste nel dare al messaggio un tracciato di tipo "record like", completando con spazi i campi alfabetici, con zero quelli numerici, sistemando le posizioni decimali, eccetera. Facendo riferimento all'esempio precedente e supponendo che l'ultimo campo numerico abbia tre posizioni decimali, il messaggio arriva alla MR nella forma:

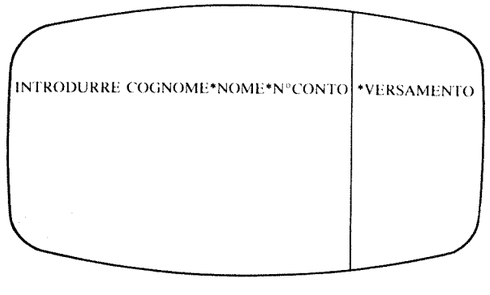
ROSSI␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣MARCO␣␣␣␣␣␣␣␣␣␣
(20) (15)

70843220 00500620
(8) (5+3)

3.2.2. Il line mode

Il line mode è il modo di scambio messaggi che il TPS propone per terminali su cui non è possibile definire "forme", cioè dividere lo schermo in zone fisse e variabili.

In questo caso, il messaggio inviato dall'ITP continuerà a guidare il terminalista nella risposta. Il terminalista stesso dovrà però inserire dei caratteri di separazione tra i vari elementi del messaggio di risposta. Facendo riferimento all'esempio precedente il messaggio inviato dall'ITP può essere:



e il terminalista dovrà rispondere nella riga successiva, fornendo di seguito le quattro informazioni, separandole una dall'altra con un asterisco (od un qualsiasi altro carattere indicato come separatore per quel messaggio). Il TPS prima di attivare la MR che elabora quella risposta opererà gli stessi controlli e le stesse normalizzazioni operate per il form mode.

E' quindi evidente che la differenza tra line mode e form mode non è rilevante per la logica di definizione di una applicazione, ma è invece determinante per facilitare al massimo l'introduzione dei dati da parte del terminalista.

3.3. La descrizione dei messaggi

Un messaggio è descritto in termini di "campi" che lo compongono.
Di ogni campo vanno definite le caratteristiche; in particolare (per il form mode):

— il tipo di campo, cioè se esso definisce una zona di schermo accessibile o meno al terminalista

— le coordinate dello schermo a cui il campo deve essere visualizzato e la sua lunghezza

— il contenuto del campo; questo può essere definito "a freddo" (è cioè una costante nota a momento di definizione del messaggio) o da attualizzare "a caldo" (è noto solo a momento di esecuzione dell'ITP ed il contenuto deve essere fornito dalla MR che richiede l'invio del messaggio)

— le caratteristiche di convalida, nel caso sia accessibile al terminalista. Ad esempio se è numerico, il numero di posizioni decimali, se deve essere obbligatoriamente compilato, eccetera

— le caratteristiche visive del campo (per i terminali il cui hardware lo permette): se ad alta o bassa luminosità, se sottolineato, se lampeggiante, eccetera.

3.4. Il processing eseguito dalle MR

La Message Routine è stata definita come il codice che descrive il processing richiesto da un tipo di messaggio. Il messaggio in input alla MR è un messaggio convalidato e normalizzato secondo i criteri precedentemente illustrati.

Il processing di una MR ha come caratteristiche principali:

— non è interattivo. Una MR ha in input dal terminale unicamente il messaggio che l'ha attivata e non può chiedere al terminalista ulteriori informazioni per completare il processing. Se sono necessarie ulteriori informazioni per il prosieguo dell'ITP, esse vanno richieste secondo la logica espressa nel ciclo di dialogo, cioè inviando al terminalista un messaggio di inquiry, la risposta al quale verrà analizzata da una successiva MR. Ciò semplifica notevolmente la codifica delle MR: esse elaborano un tipo ben preciso di dato in input;

— ha le risorse (cioè i flussi archivi del Data Base, le periferiche del terminale e i terminali di solo output) implicitamente disponibili. Non è quindi necessaria alcuna operazione di apertura/chiusura flussi od abilitazione/disabilitazione terminali

— è transazionale (vedi più oltre)

— opera su dati caratteristici del terminale (vedi più oltre).

— è "terminal independent" (vedi più oltre).

3.4.1. La transazione in ambiente TPS

Come transazione è inteso un insieme coerente e completo di modifiche agli archivi, eseguito da una MR o più in generale da un insieme di MR dello stesso ITP attivo ad un terminale.

Ad esempio, in un ITP di gestione magazzino una transazione è costituita da tutte le variazioni che comporta la consegna di un prodotto: aggiornamento flusso "bolle di consegna", aggiornamento quantità materiale nel flusso "prodotti", aggiornamento della situazione sul flusso "clienti", eccetera.

L'inizio di una transazione è implicito e coincide con la richiesta del primo record di un archivio trattato in aggiornamento. La fine transazione è dichiarata esplicitamente dalla MR.

Durante una transazione la MR (o l'ITP se la transazione è a cavallo di più MR dello stesso ITP) ha una visibilità **esclusiva** dei record dei flussi archivi elaborati. Ciò significa che se un record è in fase di aggiornamento (cioè "sotto transazione") da parte di un ITP attivo ad un terminale, una transazione innescata da un altro terminale non può accedere in aggiornamento a quello stesso record finchè la transazione precedente non è stata completata.

E' ovvio che in tali condizioni la programmazione è estremamente semplificata, soprattutto considerando che il TPS gestisce automaticamente (salvo richiesta contraria del programmatore) la sospensione di una MR che richiede un record sotto transazione, e gestisce la sua continuazione alla fine della transazione concorrente. Il TPS controlla inoltre i casi di dead-lock, cioè situazioni come quella illustrata, ad esempio, in fig. 4.

TRANSAZIONE AL TERMINALE A	TRANSAZIONE AL TERMINALE B
.	.
.	.
.	.
Richiesta record XY	.
.	.
.	Richiesta record YZ
.	.
Richiesta record YZ	.
(stato di sospensione)	.
per lock	.
.	.
.	Richiesta record XY
.	(stato di dead-lock)

fig. 4

In un'applicazione TPS il primo momento di integrity è a livello transazione. Ciò significa che se una transazione abortisce gli archivi sono riportati in uno stato coerente, cioè di inizio transazione, registrando le "before images" dei record aggiornati durante la transazione. Ciò è estremamente importante, perchè evita al programmatore di essere coinvolto in problematiche di integrity laddove la loro soluzione è essenziale. Infatti in un ambiente interattivo i casi di abort sono molto più frequenti, considerando che i dati sono introdotti ed elaborati in tempo reale da personale generalmente non specializzato e che le possibilità di errori di trasferimento dati sono molto più frequenti sulle linee che non, ad esempio, su canali disco.

Inoltre, il "lock" dei record richiesti da una transazione permette di disegnare e programmare le MR di un ITP come se esse avessero una visione esclusiva degli archivi, cioè senza dover tener conto di eventuali transazioni attivate da altri terminali ed operanti sugli stessi dati di archivio.

3.5. I dati processati dalle MR

I dati implicitamente di input ad una MR sono:

— il messaggio che la ha attivata
— la "transaction storage".

La transaction storage è un'area di memoria caratteristica del terminale. Vi è cioè una transaction storage per ogni terminale dichiarato nell'environment dell'applicazione TPS.
La transaction storage è utilizzata per passare informazioni (se necessario) da una MR alle altre dello stesso ITP.
L'allocazione di una transaction storage per ogni terminale garantisce che le MR di uno stesso ITP, contemporaneamente attivo a più terminali, operino su dati caratteristici del terminale stesso.

Inoltre, durante la sua esecuzione la Message Routine può leggere records dagli archivi ed operare su dati locali: ha a disposizione una "working storage", cioè un'area di memoria che ha validità e tempo di vita solo per l'esecuzione del processing del messaggio che l'ha attivata.

Dati in output da una MR sono i messaggi inviati al terminalista, modifiche alla transaction storage ed aggiornamenti a records degli archivi.

La transaction storage è inizializzata a momento di lancio della prima MR di un ITP, la working storage (che fisicamente è comprensiva delle "input message area") a momento di lancio di una MR.

3.6. Programmazione integrata delle MR

Programmazione integrata significa che una MR non deve richiedere o definire esplicitamente le risorse cui accede: l'allocazione o la definizione avviene implicitamente in base alle descrizioni date nell'environment.

Ad esempio, è implicito che un ITP dialoghi con il terminale che lo ha richiamato, quindi nell'indirizzare i messaggi non deve specificare il nome della stazione ricevente. E' definibile a livello ambiente l'associazione terminale master (quelli che possono richiedere l'esecuzione di funzioni) e terminali di output: quindi non è necessario indirizzarli esplicitamente.
Può essere definita a livello environment la struttura di un archivio (tipi di records e loro contenuti): non è necessario in tal caso ridefinire i records nelle MR. E così via.

Va notato che questa integrazione comporta notevoli benefici:

— si evitano ridondanze di programmazione, e quindi possibilità di errori

— le risorse sono definite al giusto livello di competenza; cioè da chi ha studiato il problema sistemistico e ne ha espresso la soluzione in termini di descrizione dell'ambiente e non da chi programma le MR

— dà un notevole grado di flessibilità, sia a tempo di esecuzione dell'applicazione, sia per la trasferibilità degli ITP. Infatti ad esempio è possibile, in caso di malfunzionamento di un terminale di output, variare a tempo di esecuzione l'abbinamento master/output senza che l'ITP debba prevedere questa flessibilità. Si può cambiare (entro certi limiti) la struttura di un archivio senza dover fare altro che ricompilare gli ITP che fanno riferimento a quell'archivio.

3.7. La "terminal independence" delle MR

Dire che una MR è "terminal independent" significa dire che il codice scritto per processare un messaggio non è funzione del tipo di terminale che ha inviato il messaggio (e che riceverà i messaggi di output od inquiry inviati dalla MR).

Il fatto è estremamente significativo, se si tiene conto che il peso implementativo di gran lunga più rilevante nello sviluppo di un'applicazione TPS è appunto quello di scrittura delle MR; che si possono cambiare i tipi di terminali connessi alla rete (od aggiungere ai terminali esistenti nuovi tipi di terminali) senza dover riscrivere od adattare le MR.
Quello che al più va fatto connettendo nuovi terminali è creare nuove descrizioni di messaggi, che meglio sfruttino le caratteristiche hardware dei nuovi tipi di terminali connessi.

L'obiettivo è stato di primaria importanza, considerata la continua evoluzione nel campo dei terminali. E' stato possibile raggiungerlo grazie alla impostazione di sviluppo di un ITP (programmazione separata dei messaggi e delle MR) e da logica aggiunta all'esecutivo del TPS, in fase di decodifica e normalizzazione del messaggio di input e di invio dei messaggi di output (vedi paragrafo 4.3).

4. COME SI SVILUPPA UN'APPLICAZIONE TPS

Gli steps di sviluppo di un'applicazione TPS nell'ordine logico di esecuzione sono:

1. Definizione dell'environment
2. Definizione dei cicli logici di dialogo degli ITP
3. Descrizione dei messaggi
4. Codifica delle MR
5. Link dei componenti dell'applicazione.

A tale punto l'applicazione può essere lanciata ed eseguita.

4.1. La definizione dell'environment

Definire l'environment di un'applicazione TPS significa definire l'ambiente in cui verrà svolta l'attività real time, in termini di:

— descrizione della rete, cioè dei terminali collegati e delle loro caratteristiche, del tipo:
  - nome simbolico con cui il terminale è riconosciuto
  - tipo di modello del terminale
  - password di accesso al terminale
  - eccetera

— descrizione degli archivi del data base, cioè:
  - nomi dei flussi e loro organizzazione
  - tipi di accessi consentiti
  - tracciato dei records
  - eccetera

— descrizione degli ITP eseguibili dai terminali, cioè:
  - tipo di linguaggio usato per la programmazione
  - se è richiesta password per poterli eseguire
  - eccetera

Possono essere fornite inoltre informazioni di altro genere, del tipo:

— modi di esprimere il separatore decimale (punto o virgola)
— "currency sign"
— eccetera.

E' chiaro che la definizione dell'environment è il momento più importante, perchè definisce l'organizzazione del centro di calcolo, la struttura degli archivi dati e più in generale richiede di impostare logicamente l'attività che si vuole eseguire nel centro a tempo reale, cioè in modo interattivo.

Di contro, il linguaggio che il TPS propone per descrivere l'environment è estremamente semplice, basato sul concetto di keyword (eccetto che per la definizione dei flussi, in cui è disponibile anche l'uso di tracciati schede RPG).

La descrizione dell'environment è convalidata sintatticamente da una utility di program preparation del TPS e, se corretta, memorizzata su di un flusso disco, da cui preleverà informazioni la utility di program preparation delle MR.

4.2. Definizione dei cicli logici di dialogo

La definizione del ciclo logico di dialogo di un ITP è una fase analoga a quella della definizione del diagramma a blocchi di un programma batch. Nel tracciare il diagramma a blocchi va tenuto conto, come spiegato nei paragrafi precedenti, della principale caratteristica di un ITP: di essere un programma interattivo, cioè in grado di pilotare un dialogo col terminalista.

Una volta definito il ciclo di dialogo risultano altresì definiti l'insieme dei messaggi di inquiry e di associate MR che sarà necessario sviluppare per realizzare l'ITP.

Tale definizione resta uno step puramente logico se il linguaggio di programmazione delle MR è COBOL, si traduce in codifica di dichiarative del tipo:

IL MESSAGGIO XY E' INPUT ALLA MR Y2
[SE .....]

se il linguaggio di programmazione delle MR è del tipo a logica generata, cioè RPG-like (vedi § 4.4.).

4.3. Definizione dei messaggi

Per la definizione dei messaggi è usato un metodo "a tracciato schede", del tipo della modulistica per la programmazione RPG. E' infatti il metodo più idoneo, considerando il tipo di informazioni che si devono fornire (vedi § 3.3.). Ogni messaggio è identificato con un nome.

La descrizione viene analizzata formalmente da una utility di program preparation e, se corretta, memorizzata su di una libreria di messaggi in forma pre-interpretata. Va rilevato che se il messaggio

può essere indirizzato a tipi di terminali con caratteristiche hardware o firmware differenti, la utility genera più oggetti nella libreria messaggi, ciascuno personalizzato per un tipo di terminale. A momento di esecuzione della MR che richiede l'invio del messaggio, le routines dell'esecutivo TPS selezioneranno l'oggetto opportuno a seconda del tipo di terminale connesso. In tal modo è conseguita la "terminal independence" del codice delle MR.

4.4. La programmazione delle MR

Il TPS offre due linguaggi di programmazione:

— RPG-like
— Macro COBOL

Ciò non solo in ragione della loro diffusione sul mercato, ma anche perchè hanno caratteristiche complementari. Più facile da usare ma meno flessibile e potente RPG; con un set di statements molto più vasto ma complesso COBOL.

Per tutti e due i linguaggi sono state introdotte delle variazioni rispetto allo standard di programmazione batch. Questo perchè entrambi si presentano inadeguati alla programmazione interattiva. RPG II perchè il ciclo logico fisso non è praticamente mai utilizzabile; cosa del resto ovvia essendo stato pensato e realizzato per elaborazioni di tipo sequenziale di archivi dati e produzione di reports "continui". A COBOL sono state invece tolte alcune dichiarative (tipo DATA DESCRIPTION per i flussi disco, dato che tale descrizione è centralizzata a livello environment e permetterla non avrebbe altro significato che permettere ridondanza di programmazione e possibilità di errori) ed alcuni statements procedurali (tipo OPEN/CLOSE, per quanto precedentemente detto: ogni MR ha le risorse implicitamente in linea).

Quello che invece si è voluto mantenere è la completa aderenza agli standard per tutto quello che non riguarda quel set (minimo) di variazioni. Ciò perchè coesistendo la programmazione interattiva con quella per il batch si sarebbero introdotti costi addizionali di formazione per i programmatori, ed i programmi stessi non sarebbero stati facilmente trasferibili ad altri sistemi. Per cui ad esempio tutto quel che riguarda i verbi di "procedure division" (salvo alcuni di I/O) è standard ANSI.

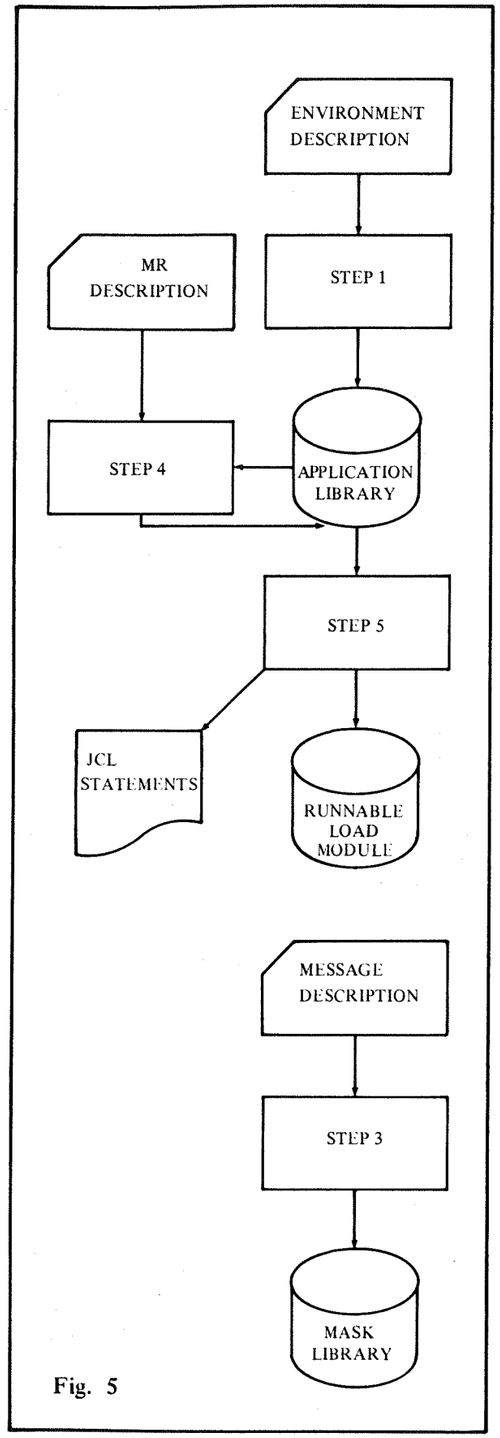


Fig. 5

4.5. Flow di program preparation

Riepilogando quanto detto, il flow "della program preparation" di un'applicazione TPS è indicato in figura 5.

Va notato che per facilitare la messa a punto della applicazione non è necessario eseguire lo STEP 5 solo quando tutti gli ITP (cioè le relative MR e messaggi) sono stati definiti, ma è possibile generare dei load modules contenenti un set di ITP ridotto.

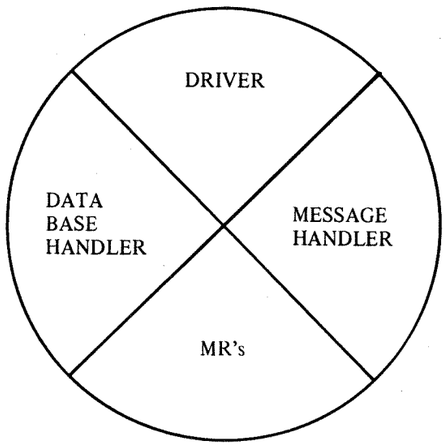
5. COME E' STATO REALIZZATO IL TPS

Il TPS è stato realizzato come un sottosistema del GCOS di Livello 62, in grado di cooperare col sottosistema batch esistente. Come è stato detto nel primo capitolo l'obiettivo del sottosistema è la facilità d'uso.
Per raggiungere tale obiettivo si sono dovuti risolvere dei problemi caratteristici dell'interattivo e principalmente:

— processare concorrenzialmente i messaggi inviati dalla rete
— fornire un interfaccia logico coi terminali
— fornire capacità transazionali al sottosistema.

La soluzione è stata quella di sviluppare del codice di sistema, orientato a risolvere quei problemi e collegato alle MR scritte dall'utente in fase di program preparation.

Da un punto di vista logico il sottosistema è schematizzabile nel seguente modo:



Dove:

— il DRIVER ha funzioni di coordinamento dell'applicazione: gestisce le fasi di start-up e di shut-down e smista i messaggi in arrivo dalla rete attivando le appropriate MR (a seconda del tipo di messaggio ricevuto)

— il MESSAGE HANDLER si incarica di gestire l'interfaccia logico tra MR e terminali: convalida e normalizza i messaggi in arrivo dalla rete ed esegue le richieste di invio logico di messaggio da parte delle MR, preparando il testo da inviare in linea e richiedendo l'invio fisico del messaggio in linea (quando necessario)

— il DATA BASE HANDLER controlla lo sharing degli archivi e gestisce le transazioni: è un filtro tra le richieste di records da parte delle MR ed il Data Management standard.

Il DRIVER ed il DATA BASE HANDLER sono personalizzati per il particolare environment durante lo STEP 5 di program preparation e vengono collegati alle MR scritte dall'utente ed al MESSAGE HANDLER sempre durante lo STEP 5.
Durante lo STEP 4 di program preparation vengono invece costruiti gli interfaccia tra le MR, il MESSAGE HANDLER e il DATA BASE MANAGER per le richieste di I/O.

5.1. La logica di funzionamento del TPS

Un'attività TPS è schematizzabile col diagramma di fig. 6.

La "linea di asincronismo" tratteggiata sta ad indicare che le operazioni in linea (cioè l'invio fisico dei messaggi e la loro ricezione) sono eventi asincroni rispetto al processing di un messaggio. Ciò permette di avere tra l'altro una completa sovrapposizione tra i tempi di trasmissione/ricezione ed i tempi di processing; cioè mentre l'attività TPS sta processando un messaggio per un terminale, tutti gli altri terminali della rete possono essere in fase di trasmissione o ricezione.
Le operazioni fisiche di linea sono gestite dal CS/B (Communications Supervisor/Basic).

L'asincronismo è reso possibile da due code logiche: una dei messaggi inviati da CPU verso terminale, una dei messaggi ricevuti da terminale a CPU.
Dalla seconda coda il DRIVER preleva un messaggio, riconosce da che terminale è stato inviato, riconosce di che tipo è il messaggio e quindi seleziona la MR appropriata, la carica in memoria e le passa il controllo.

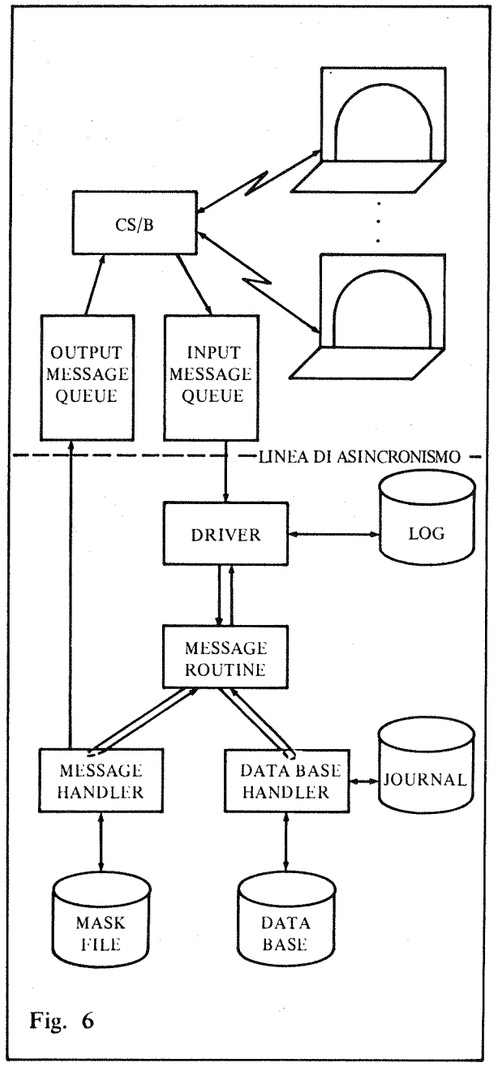


Fig. 6

La MR (dopo che il messaggio è stato convalidato e normalizzato dal Message Handler) processa il messaggio ed a fine processing ritorna il controllo al DRIVER (per semplicità non sono discussi i casi di sospensione di una MR per TIMEOUT o altro).
Durante il processing di un messaggio ogni richiesta di Input/Output è passata dalla MR al MESSAGE HANDLER o al DATA BASE HAN-

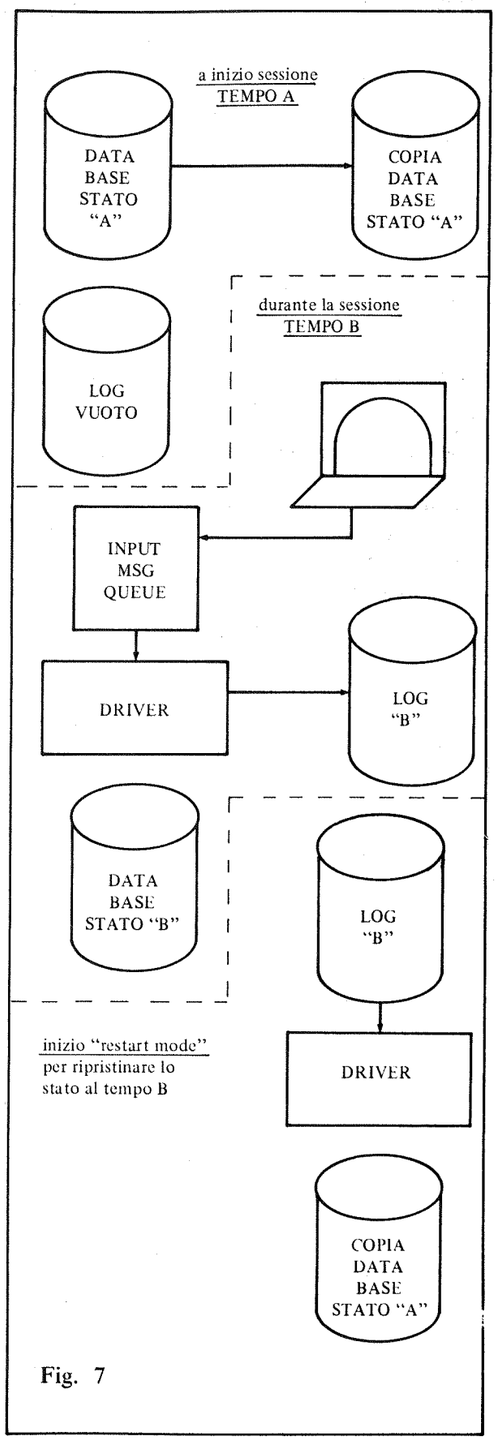


Fig. 7

DLER. Le richieste di output verso il terminale sono sempre e solo logiche ed asincrone per cui la MR non attende l'invio di un messaggio in linea per continuare il suo processing.

Il DATA BASE HANDLER si appoggia ad un flusso "JOURNAL" in cui sono registrate le "before images" dei record aggiornati per poter eseguire i ripristini a livello transazione.

Un discorso a parte merita, infine, il flusso di LOG.

5.2. La logica di integrity a livello sessione TPS

Si è già detto dell'integrity a livello transazione. Vale la pena di considerare come viene invece salvaguardata l'integrity di un'intera sessione TPS, quando eventi improbabili, ma comunque possibili, hanno provocato ad esempio la distruzione dei flussi archivi del Data Base.

Il ripristino in tale caso è ottenuto attraverso un flusso di "LOG". Su tale flusso il DRIVER durante una sessione TPS registra i messaggi che preleva dalla INPUT MESSAGE QUEUE, creando così una storia dell'intera sessione (ciò in prima approssimazione).
Ciò permette di ricostruire lo stato del Data Base al momento in cui la sessione era stata interrotta, partendo da un Data Base di "Back-up", dal flusso di "LOG" e rieseguendo il lavoro fatto a real time in "restart mode" (vedi fig. 7).

Durante il "restart mode" i terminali non sono in linea e i messaggi sono prelevati dalla coda creata sul flusso di LOG.

I tempi di ripristino sono molto ridotti (si può considerare un rapporto di 3 min. di restart/1 ora di real time) tenendo conto che le operazioni in linea non sono eseguite né vengono rilanciati gli ITP di "puro inquiry", cioè che non alterano lo stato degli archivi.

UN'ESPERIENZA DI PERFORMANCE EVALUATION IN ANSALDO

F. Parodi (+), G. Zanone (+), C. Firpo (°), M. Calvini (°)

Presentazione

Questo studio è stato presentato al VII Workshop dell'HLSUA (Honeywell Large Systems Users Association) a Salisburgo (novembre 1978) su mia proposta a quel Comitato Organizzatore quando, oltre a dirigere il Servizio Sistemi ed Elaborazione dell'Ansaldo S.p.A., rappresentavo l'Italia nel Board dell'HLSUA Europe.
Dato l'interesse suscitato in quella sede, e dato l'interesse intrinseco che ritengo debba essergli riconosciuto anche per il fatto di riferirsi ad una realizzazione a scopo non di pura ricerca ma operativo in un sistema pressato da necessità di ottimazione sia nel proprio normale esercizio che in vista di modificazioni anche profonde, lo ripropongo ora ai lettori di NOTE DI SOFTWARE.
Degli autori, i primi due, l'Ing. Giovanni Parodi e il P.I. Guido Zanone, fanno parte dei servizi EDP dell'Ansaldo di Genova; gli altri due, Sigg. Claudio Firpo e Marco Calvini, hanno dato valido contributo soprattutto nell'impostazione del modello matematico e nella realizzazione dei programmi di sovrapposizione, facendo nel contempo lavoro utile per le loro tesi di laurea in ingegneria presso l'Università di Genova.
Ho infine il piacere di segnalare il prezioso aiuto dato dal Sig. Pierino Bonifazio, della Honeywell I.S.I.

Silvio Barigozzi

1. INTRODUZIONE

Un problema pratico di grande interesse che si presenta agli utilizzatori ed ai progettisti di sistemi per l'elaborazione delle informazioni è quello di adeguare le "prestazioni" di tali sistemi alle reali esigenze che questi debbono soddisfare.

A tal fine è necessario:

- Definire delle grandezze che siano rappresentative delle "prestazioni" del sistema, sia esso da verificare od ottimizzare, sia esso da progettare
- Definire una serie di strumenti che consentano di effettuare tali misure (monitors hardware e software)
- Determinare i valori obiettivi delle grandezze selezionate
- Definire una serie di strumenti di predizione (modelli del sistema e del carico).

A questa semplice schematizzazione non corrisponde, però, altrettanta chiarezza sulle tecniche e sugli strumenti da utilizzare in generale.
La "performance evaluation" è, a tutt'oggi, un argomento non ancora ben consolidato, e non è neppure detto che possa mai esserlo, data la sua complessità.
Esistono, peraltro, in materia, alcuni diversi possibili approcci teorici, ognuno dei quali può fornire i migliori risultati in alcune specifiche aree di applicazione.
Si sente, però, la necessità di razionalizzare e di unificare: si può ricordare, qui, il notevole tentativo in tal senso di K. W. Kolence che, con la sua "Software Physics", tende ad unificare tutte le misure in una teoria consistente.
Sulla base di tre sole grandezze fondamentali (lavoro, tempo e memoria), la "Software Physics" cerca di definire in modo non ambiguo le varie grandezze in gioco affinchè i risultati forniti dai vari possibili approcci diventino fra loro paragonabili.

2. IL SISTEMA IN ESAME

Il sistema in esame è quello centrale della Società Ansaldo, con sede a Genova, e unità produttive dislocate in diversi punti del territorio nazionale (fig. 1).

L'Ansaldo opera nel campo elettromeccanico, nei settori di generazione, distribuzione ed utilizzazione di energia elettrica, con produzioni che vanno dalla generazione di vapore (caldaie) alle grandi

(+) Ansaldo, Genova
(°) laureando presso l'Università di Genova

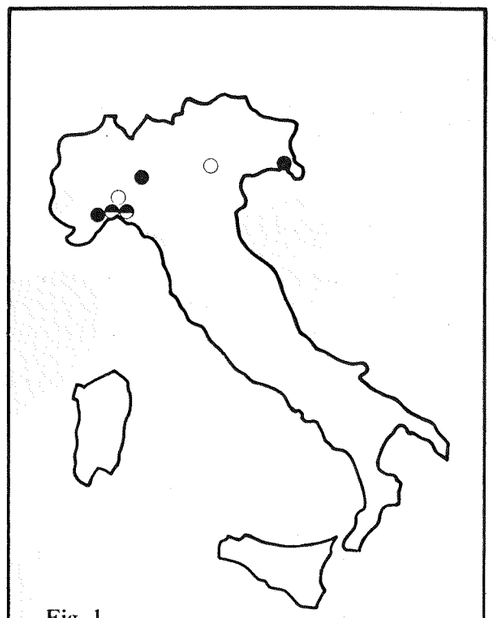


Fig. 1

macchine rotanti (turbine, alternatori), all'impiantistica, ai trasporti, all'automazione, fino ai motori e generatori di media e piccola potenza, compresa l'elettronica industriale ad esse associata.

L'impatto di questa articolazione, con la conseguente varietà di tecniche sia gestionali che scientifiche ha prodotto i suoi riflessi sull'architettura del sistema centrale di elaborazione dati, provocando una evoluzione che lo ha portato, nel giro di 6 anni, alla configurazione della fig. 2.

3. PERCHE' "COMPUTER PERFORMANCE EVALUATION" IN ANSALDO

Il sistema è pervenuto alla sua attuale configurazione mediante modifiche ed ampliamenti graduati nel tempo, coerentemente con gli sviluppi delle applicazioni e del sistema informativo.

A tali modifiche si è sempre pervenuti, però, sulla base di considerazioni fondate su dati che, per quanto riguardava la macchina, erano quelli di accounting.
Ora, è chiaro che dati rivolti all'aspetto soprattutto economico del sistema, in termini di attribuzione di costi, non consentono di fare considerazioni sul suo comportamento dinamico.

Se è ancora accettabile utilizzare dati di accounting per considerazioni di "performance" in presenza di un sistema con architettura e carichi non molto complessi, e dotato di una schedulazione rigida, ben controllata e controllabile, non è più accettabile quando si è in presenza di una architettura articolata, di un decentramento di risorse e di una varietà di applicazioni e di cadenze elaborative quale è quella raggiunta a questo punto in Ansaldo.

La necessità di osservazioni dinamiche sul sistema diventa, poi, irrinunciabile quando si consideri l'articolazione delle modalità di lavoro e degli orari su cui sono distribuite (fig. 3).

Da tutto ciò la decisione, assunta recentemente, di aprire, con mezzi limitati ma con una certa sistematicità, un discorso di "computer performance evaluation" con l'obiettivo di:

- mettere a punto strumenti di verifica e di progetto della configurazione hardware;
- analizzare le interazioni carico-sistema in relazione ai diversi tipi di accesso (local-batch, remote-batch, time-sharing);
- analizzare l'andamento e la tipologia del carico per la messa a punto di modelli da utilizzare in eventuali future elaborazioni di simulazione.

Questa presentazione è relativa al primo di questi punti, ed accenna ai problemi legati alla soluzione dei successivi, che costituiscono l'argomento di lavoro da affrontare nei prossimi mesi.

4. IL MONITOR

Le "performances" di un sistema di elaborazione non possono essere stabilite da misure istantanee ed improvvisate, ma solo osservando il sistema per un periodo di tempo sufficientemente lungo, rilevandone la gestione delle risorse e la loro utilizzazione mentre il sistema stesso sta funzionando.

Un Monitor è uno strumento che facilita le misure analitiche necessarie per una valutazione di prestazioni.

La struttura di un monitor è rappresentata in figura 4. L'implementazione di questi elementi può essere realizzata con tecniche diverse, come si può rilevare dalle due tabelle in figura 5. La prima tabella descrive un monitor software, la seconda un monitor hardware.

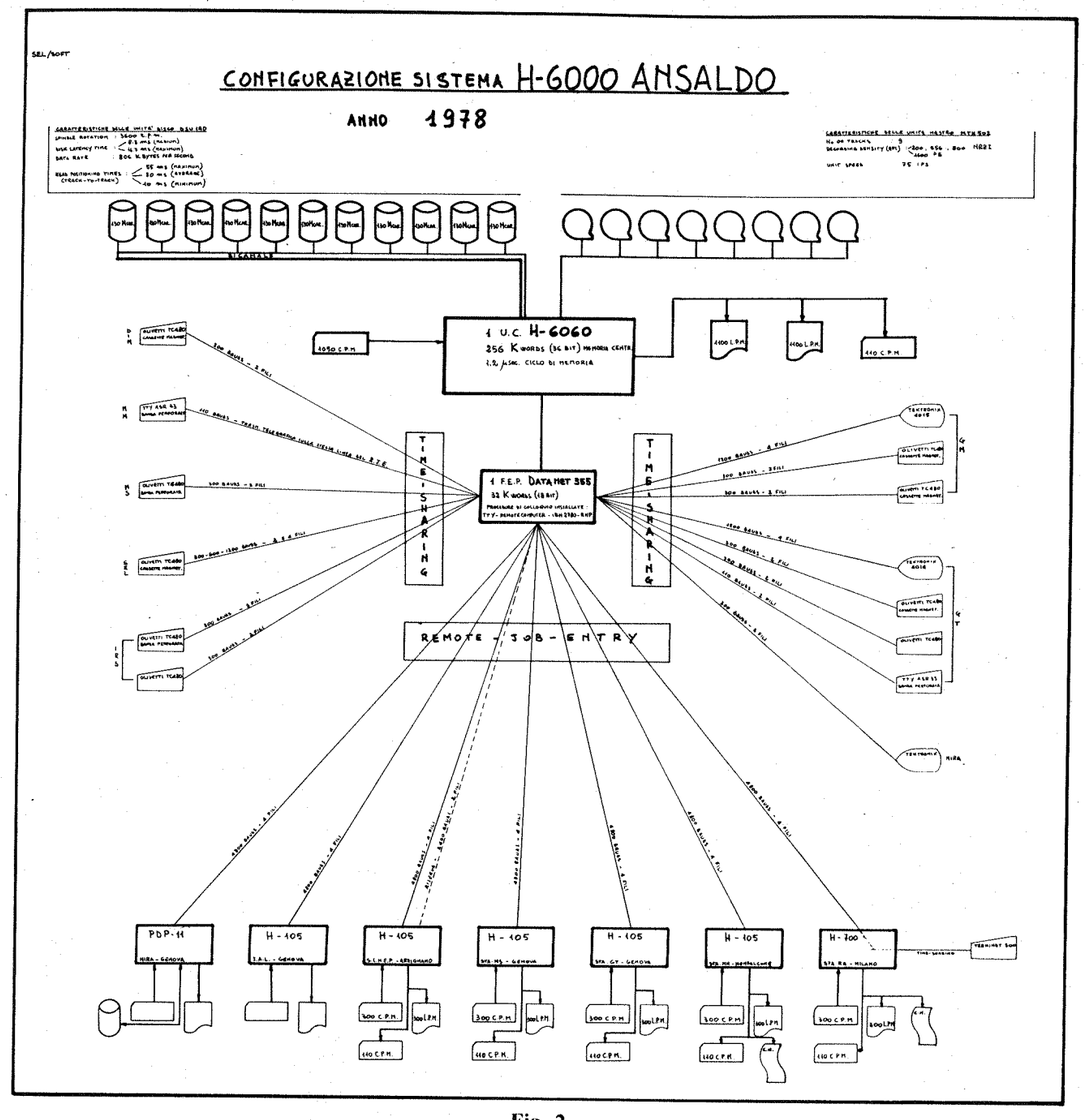


Fig. 2

Lo strumento da noi adottato per affrontare il lavoro sulla valutazione delle prestazioni del sistema di elaborazione dell'Ansaldo è appunto un monitor software fornito dalla Honeywell, chiamato SYMON.

SYMON rispecchia la struttura prima vista (figura 4). Possiamo qui parlare di due elementi: l'ELEMENTO ESTRATTORE e l'ELEMENTO INTERPRETATORE.

Il primo è stato implementato in GMAP e deve lavorare in parallelo agli altri lavori nel sistema, per cui deve portare poco overhead.
L'elemento interpretatore è stato implementato in Fortran, e va in esecuzione in tempo differito, a mezzanotte, quando il carico sul sistema è sensibilmente ridotto e quindi non ci sono problemi eccessivi di disturbo sul normale funzionamento.
La figura 6 riassume l'organizzazione di SYMON.

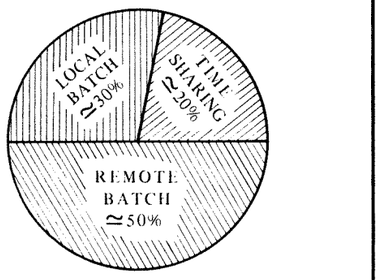


Fig. 3

5. CARATTERISTICHE DEL SYMON

Avendo definito come "attività" il funzionamento di un singolo componente del sistema o di un intero livello del sistema, una attività può essere rappresentata da una funzione logica così definita:

$$a_k = \begin{cases} 1 & \text{presenza dell'attività k-esima} \\ 0 & \text{assenza dell'attività k-esima} \end{cases}$$

SYMON fornisce misure di ATTIVITA' RELATIVA, cioè

$$r_k = \frac{1}{t-t_o} \int_{t_o}^{t} a_k(t)dt$$

Ciò viene ottenuto con una tecnica di campionamento: siccome l'occorrenza e la durata delle varie attività di sistema sono variabili casuali, il campionamento può essere eseguito a intervalli regolari di tempo.

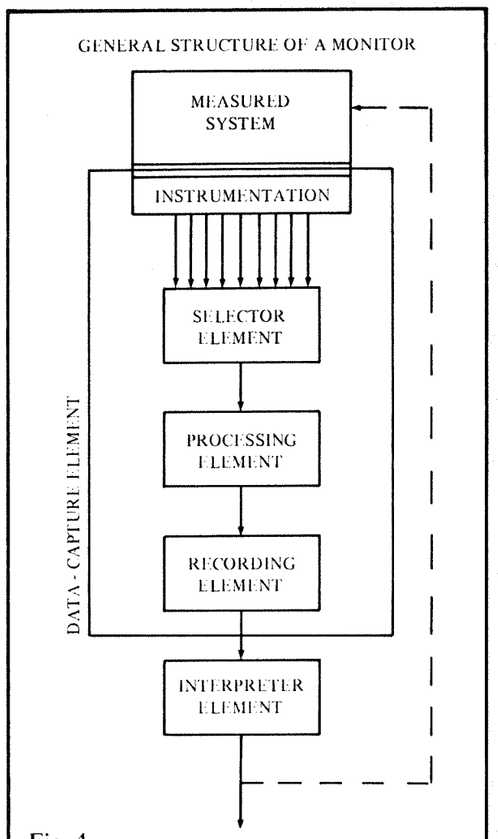


Fig. 4

Più lungo è il periodo di campionamento, più bassa può essere la frequenza di campionamento.
SYMON campiona con la frequenza desiderata i parametri relativi alle attività di sistema (figura 7).

6. LA SINTESI DEI DATI DI CAMPIONAMENTO

Uno dei problemi maggiori nell'uso dei monitors è di adottare una rappresentazione efficace e sintetica per la moltitudine di dati forniti dal monitor stesso, onde permettere una analisi facilitata ed immediata.

SYMON fornisce grafici su tabulato che risultano abbastanza confusi e di non facile lettura.
Noi abbiamo deciso di adottare la configurazione grafica ideata e descritta da Kiviat, Kolence e Merrill: si tratta di grafici cui ci si riferisce spesso col nome di "GRAFICI DI KIVIAT".

IMPLEMENTATION OF STRUCTURAL ELEMENTS OF A SOFTWARE MONITOR

INSTRUMENTATION	TRAPS SAMPLING INTERCEPTIVE
SELECTOR ELEMENT	SOFTWARE SWITCH SPECIAL HARDWARE
PROCESSING ELEMENT	SOFTWARE
RECORDING ELEMENT	SYSTEM MAIN MEMORY SYSTEM SECONDARY STORAGE DEVICES
INTERPRETER ELEMENT	POST-PROCESS (SOFTWARE) REAL-TIME (SOFTWARE)

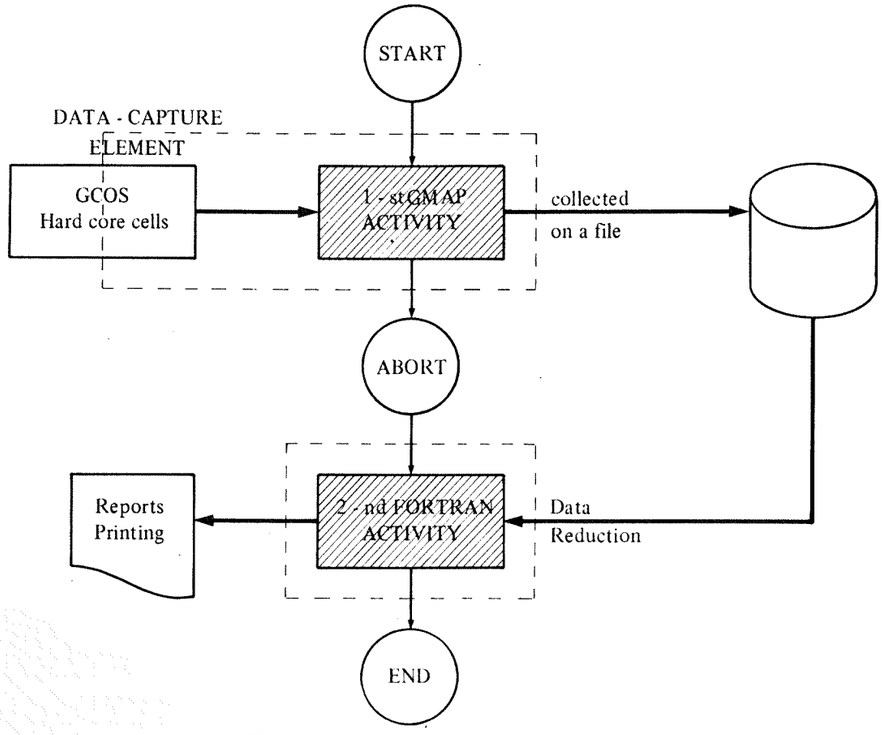
IMPLEMENTATION OF STRUCTURAL ELEMENTS OF A HARDWARE MONITOR

INSTRUMENTATION	ELECTRONIC PROBES (SENSORS) PLUG INTERFACE WIRED INTO THE SYSTEM
SELECTOR ELEMENT	LOGICAL PLUGBOARD ASSOCIATIVE MEMORY SOFTWARE
PROCESSING ELEMENT	LOGICAL PLUGBOARD SOFTWARE
RECORDING ELEMENT	COUNTERS RANDOM ACCESS MEMORY SECONDARY STORAGE DEVICE
INTERPRETER ELEMENT	SOFTWARE-POST-PROCESS SOFTWARE-REAL-TIME HARDWIRED DISPLAY

Figura 5

La scelta dei parametri da rappresentare sugli assi è arbitraria: per avere un punto di riferimento abbiamo optato per quella che compare sempre sulla letteratura relativa alla "performance evaluation".

La figura 8 mostra due tipici grafici di Kiviat. Come si può facilmente intuire tali rappresentazioni si basano su valori medi ricavati dalle misure sul sistema.



SYMON GENERAL DESCRIPTION

Fig. 6

I parametri qui riportati sono particolarmente orientati a mettere in risalto il comportamento in multiprogrammazione del sistema. Si riferiscono infatti ad attività di CPU, di I/O e alla sovrapposizione di tali attività.
Per noi, I/O significa attività di canale disco e/o canale nastro.
Tale scelta è stata dettata da considerazioni sul carico del sistema e da osservazioni, in base ai valori, forniti da SYMON, dell'attività del canale stampante, risultata insignificante.

Consideriamo nuovamente la figura 7.
Come si vede, i parametri forniti da SYMON non sempre corrispondono a quelli da riportare sul grafico di Kiviat. Questi ultimi sono stati quindi ricavati in base alle relazioni riportate in figura 9.

7. L'OVERLAP

Tutte le considerazioni sin qui fatte sui grafici di Kiviat sono imperniate sul concetto di SOVRAPPOSIZIONE DI ATTIVITA', vero e proprio elemento di base della multiprogrammazione.
SYMON NON RILEVA ALCUNA SOVRAPPOSIZIONE DI ATTIVITA'.
Il fatto è dovuto sia alle caratteristiche del SYMON, sia a quelle del H-6000 che, per quanto sia a nostra conoscenza, non ha, contrariamente ad altri sistemi, circuiti cablati per mettere a disposizione questo parametro.

L'ostacolo potrebbe essere superato tramite un monitor hardware, che si è rivelato però uno strumento estremamente costoso, fin dai primi preventivi richiesti a ditte specializzate.

MAIN PARAMETERS FROM SYMON

- Percentage idle time total and for any processor
- Percentage GCOS overhead, total and for any processor
- Available memory at the sampling time
- Average seeks per second
- Average disc channels busy
- Average tape channels busy
- Average printer channels busy
- Percentage processor time for CALC
- Percentage processor time for PALC
- Percentage processor time for SKED
- Percentage processor time for GEOT
- Number of TSS users at the sampling time
- Core size used by TSS at the sampling time
- Processor time of the jobs TSS
- I/O time for the jobs TSS
- Zero urgency memory at the sampling time
- Percentage processor time for PRIVITY jobs
- Total jobs terminated
- Total activities performed

Figura 7

La strategia da noi adottata è stata quella di ristrutturare completamente il secondo elemento di SYMON (interpretatore), trasformandolo da modulo passivo, che si limita a visualizzare i dati raccolti e registrati dall'elemento estrattore, a modulo attivo, in grado di calcolare a programma la sovrapposizione di due attività generiche, e in grado di visualizzare conseguentemente dei grafici di Kiviat completi (uniti a grafici normali sull'andamento delle varie attività durante la giornata).

L'insieme dei due elementi, estrattore (da SYMON) ed interpretatore (nostro) costituisce il programma che abbiamo battezzato MONIT.

8. CALCOLO DELL'OVERLAP

Vediamo brevemente l'impostazione data al problema del calcolo del tempo di sovrapposizione fra attività.

Supponiamo di essere in grado di ricavare una distribuzione di probabilità di sovrapposizione in funzione del tempo di sovrapposizione (figura 10 a).
Le ascisse si estendono per tutta la fascia oraria in cui c'è stato monitoraggio di sistema.

Supponiamo poi di riuscire a sintetizzare una funzione della forma mostrata in figura 10 b.
In tal caso l'ascissa corrispondente al massimo della funzione corrisponde al tempo di sovrapposizione che ha la massima probabilità di essersi verificato nell'arco di tempo in cui il sistema è stato monitorato.

Si possono benissimo considerare altri criteri per ricavare la stima migliore del tempo di sovrapposizione. Abbiamo scelto questo perchè semplice e veloce. Per avere una idea dell'accuratezza di questa soluzione si può calcolare l'intervallo di ascisse per il quale l'integrale abbraccia il 90% dell'area racchiusa dalla funzione: se questo intervallo è minore o uguale al 10% dell'estensione della fascia oraria di monitoraggio, il valore di sovrapposizione così ottenuto si può considerare ad alto grado di affidabilità.

La nostra esperienza ha dimostrato che, in base a certe ipotesi di partenza, la curva ottenuta è proprio come quella considerata in figura 10 b.

L'appendice riporta l'illustrazione del procedimento adottato per il calcolo dell'overlap.

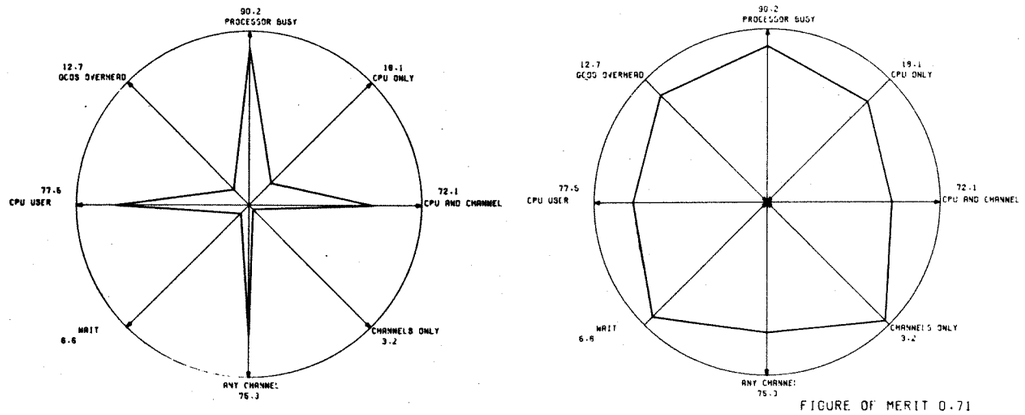
9. ORGANIZZAZIONE DI MONIT

L'organizzazione generale di MONIT è quella indicata in figura 11.

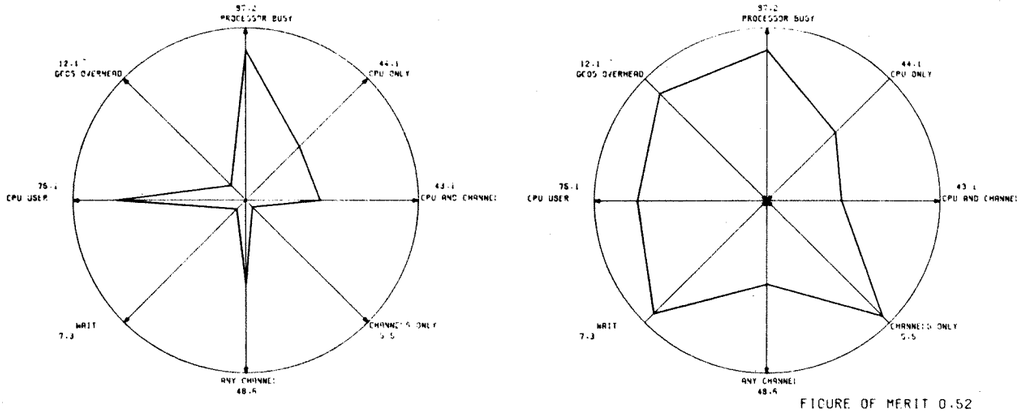
I programmi che lo compongono svolgono le seguenti funzioni:

— SYMON
E' costituito dal solo elemento estrattore di SYMON, che provvede ad eseguire il campionamento dei parametri caratteristici (44) del sistema.

— ORMED
Seleziona i parametri desiderati, li trasforma in



I grafici di Kiviat si ottengono riportando su degli assi i valori dei parametri in esame; ciò consente una visualizzazione immediata del comportamento del sistema.
Il criterio più diffuso nella letteratura relativa è quello di disporre sugli assi dispari i parametri "buoni" (quelli, cioè, che in linea di massima dovrebbero avere un valore alto) e sui pari, viceversa, i parametri "cattivi".
I parametri e la disposizione da noi usati sono tra i più tipici.
I due grafici in figura sono rappresentazioni diverse degli stessi dati. Il secondo si differenzia dal primo in quanto i valori dei parametri "cattivi" sono riportati partendo dal bordo della circonferenza anzichè dal centro. In questo modo il risultato sarà una configurazione poligonale anzichè stellare; l'area compresa dalla poligonale acquista così un significato fisico perchè il suo incremento è proporzionale all'incremento delle prestazioni "buone" del sistema e al decremento di quelle "cattive".
A questo punto è quindi possibile definire un indice di prestazione pari al rapporto tra l'area del grafico ottenuto e l'area del grafico ideale di riferimento.



Il grafico qui riportato mostra una configurazione stellare tipica di un sistema "CPU BOUND" (sbilanciato cioè sulla CPU) e nota come "CPU SAILBOAT" o "Vela di CPU".

Figura 8

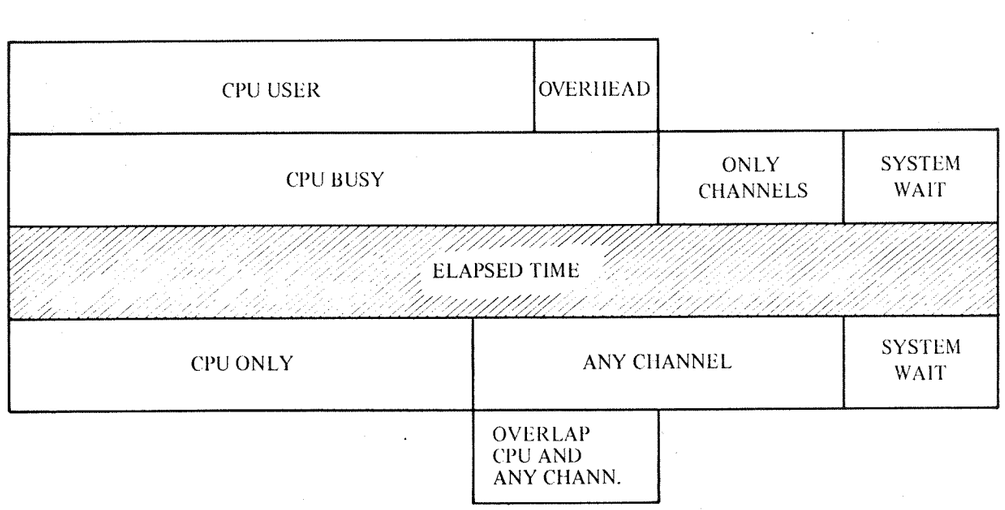
INFORMATION FROM DATA-CAPTURE ELEMENT

— CPU BUSY
— GCOS OVERHEAD
— DISC CHANNELS BUSY
— TAPE CHANNELS BUSY

DATA BY OVERLAP PROGRAM

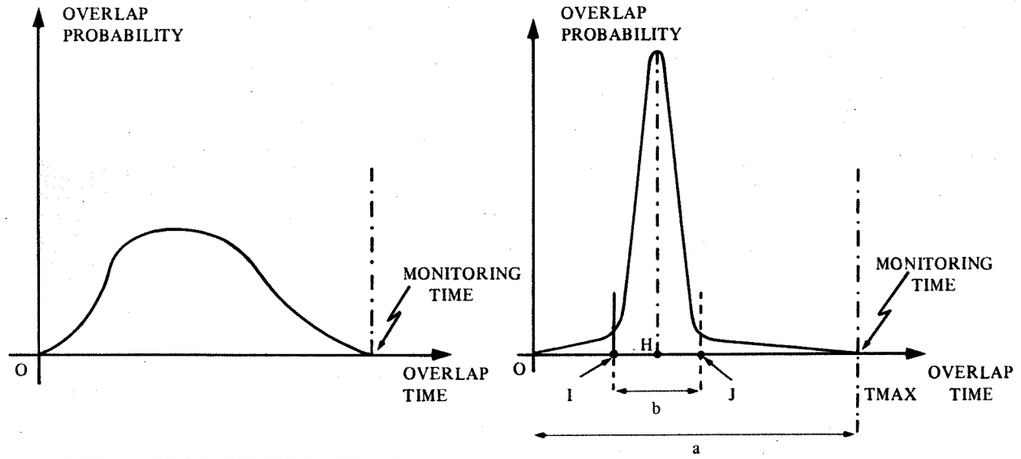
ANY CHANNEL = **NOT** [**NOT** (DISC CHANNEL) **AND NOT** (TAPE CHANNEL)]

— CPU BUSY **AND** ANY CHANNEL
— CPU ONLY = CPU BUSY — (CPU BUSY **AND** ANY CHANNEL)
— CPU USER = CPU BUSY — GCOS OVERHEAD
— CHANNELS ONLY = ANY CHANNEL — (CPU BUSY **AND** ANY CHANNEL)
— CPU WAIT = 100 — [CPU BUSY + ANY CHANNEL + (CPU BUSY **AND** ANY CHANNEL)]



I/O AND CPU ACTIVITIES OVER THE ELAPSED TIME

Fig. 9



OVERLAP PROBABILITY DIAGRAM

a)

b) H = MOST PROBABLE OVERLAP TIME IF

$$\int_I^J f\,dt = \frac{90}{100}\int_o^{TMAX} f\,dt \;\&\; b \leqslant \frac{10}{100}a$$

Fig. 10

valori percentuali ed esegue le medie richieste dall'input primario.
E' scritto in linguaggio Fortran.

— PLOSYM
Elabora per il plotter (Calcomp 936) i parametri di cui si desidera visualizzare l'andamento nel tempo.
Il programma è in grado di selezionare un particolare giorno da dati archiviati, plottare solo un arco di tempo desiderato, visualizzare da uno a tre parametri per grafico, campionando come desiderato i dati di SYMON.
E' scritto in Fortran.

— OVERLAP
Calcola i valori di sovrapposizione fra i parametri, e le grandezze derivate (figura 8) per i grafici di Kiviat.
I valori calcolati vengono anche stampati per l'eventuale costruzione a mano dei grafici di Kiviat.
E' scritto in Fortran.

— KIVIAT
Elabora per il plotter i grafici di Kiviat.
E' scritto in Fortran.

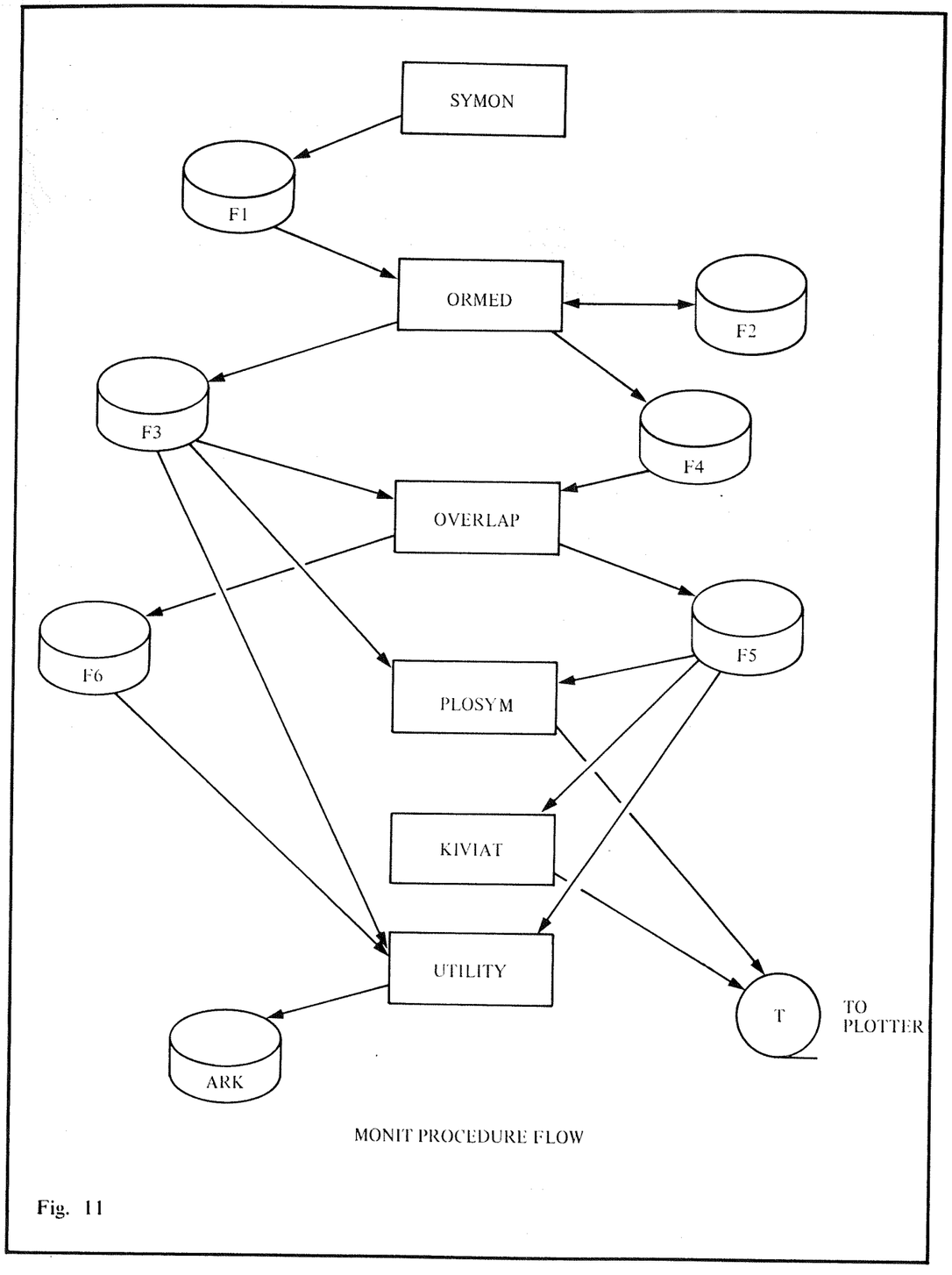
— UTILITY
Esegue l'accumulo dei dati per eventuali elaborazioni in tempo differito.

MONIT può essere composto in diversi jobs, per l'esecuzione sia totale che parziale, con o senza output grafici, in successione automatica all'estrazione dei dati o in tempo differito, qualora, volendo proseguire ininterrottamente i campionamenti, non si vogliano arrecare disturbi al sistema sotto misurazione.

L'eventuale obsolescenza di SYMON come elemento estrattore può essere ovviata facilmente adottando un qualsiasi altro elemento estrattore e realizzando una semplice interfaccia che produca un output di formato identico a quello di SYMON.

10. CONCLUSIONI E PROSPETTIVE

Siamo convinti di esserci dotati di uno strumento di analisi le cui caratteristiche principali sono la flessibilità (archiviazione dei parametri di campionamento, esecuzione "on-line" o in tempo differito, selezionabilità del periodo da analizzare), ottima leggibilità dei risultati (output su plotter a più



MONIT PROCEDURE FLOW

Fig. 11

colori) e disponibilità dei valori di sovrapposizione.
E' quest'ultima caratteristica soprattutto a rendere apprezzabile MONIT: non è infatti, a nostra conoscenza, disponibile alcuno strumento software che, per la linea H-6000, metta a disposizione tali valori.

Ci necessita peraltro, sui valori di sovrapposizione, una validazione: anche se i valori che calcoliamo statisticamente sono più che accettabili, sarebbe opportuno verificarli facendo eseguire MONIT in parallelo su un sistema monitorato in hardware.

Un altro problema è quello di determinare in maniera abbastanza sicura le configurazioni dei grafici di Kiviat e i valori delle cifre di merito da adottare come riferimenti di buon funzionamento.

Solo dopo aver fatto ciò si potrà condurre una analisi completa che, abbinando per esempio le considerazioni suggerite da MONIT a quelle derivanti da qualche programma di "trace" del carico, permetterà di decidere se eventuali interventi debbano essere operati sull'architettura del sistema piuttosto che sulla composizione del carico, o viceversa.

11. RIFERIMENTI

[1] L. Svobodova, COMPUTER PERFORMANCE MEASUREMENT AND EVALUATION METHODS: ANALYSIS AND APPLICATIONS, Elsevier Scientific Publishing Co., 1976, pagg. 146

[2] I. Epstein, PERFORMANCE EVALUATION - METHODOLOGY, TOOLS AND APPLICATIONS, Internal Report, Honeywell I.S., LISD, Phoenix, Arizona

[3] K. W. Kolence, P. J. Kiviat, SOFTWARE UNIT PROFILES AND KIVIAT FIGURES, ACM/Sigmetrics Performance Evaluation Review, giugno 1973, vol. 2, n. 3, pagg. 31-36

[4] B. H. E. Merrill, A TECHNIQUE FOR COMPARATIVE ANALYSIS OF KIVIAT GRAPHS, ACM/Sigmetrics Performance Evaluation Review, marzo 1974, vol. 3, n. 1, pagg. 34-39

[5] M. F. Morris, KIVIAT GRAPHS -CONVENTIONS AND FIGURES OF MERIT, ACM/Sigmetrics Performance Evaluation Review, ottobre 1974, vol. 3, n. 3, pagg. 2-8

APPENDICE

1. Introduzione

In questa appendice vediamo come si può calcolare la probabilità di sovrapposizione fra due grandezze qualsiasi, purchè si abbiano a disposizione dei valori relativi al funzionamento delle grandezze stesse, ottenuti con una tecnica di campionamento a intervalli regolari di tempo.
Risultato di questo studio sarà un grafico che riporta sulle ascisse il tempo di sovrapposizione, e sulle ordinate la probabilità di sovrapposizione.

2. Ipotesi avanzate:

a) Le grandezze considerate siano statisticamente indipendenti.

b) All'interno dell'intervallo di tempo tra due campionamenti le attività abbiano un andamento continuo nel tempo.

Noi abbiamo applicato questo procedimento allo studio delle prestazioni di un sistema di elaborazione. In tal caso, le caratteristiche del carico (miscela di programmi ad elevata eterogeneità, applicazioni a carattere scientifico e gestionale) garantiscono che la richiesta di allocazione delle risorse, corrispondenti alle attività, rispetti la prima ipotesi.
La seconda ipotesi, in realtà, non sempre è verificata, tuttavia essa si riferisce a una situazione peggiore di quella reale; quindi può essere accettata in quanto i risultati reali saranno sicuramente più ottimistici di quelli ricavati in base a questa ipotesi.

3. Noi abbiamo a disposizione, per ogni attività, un numero percentuale che riassume tutta la storia di quella attività nell'arco di tempo che intercorre tra due campionamenti.
Siano date due misure di attività relativa, r_k e r_h,che caratterizzano rispettivamente l'attività k-esima e quella h-esima, di cui si vuole calcolare la sovrapposizione.
Chiamiamo Δ_t il periodo di campionamento. Suddividiamo la fascia oraria in esame in n Δ_t e iniziamo a costruire la distribuzione di probabilità corrispondente a un Δ_t.
Considerando r_k e r_h possiamo già fissare il massimo e il minimo tempo di sovrapposizione possibili; infatti si ha:

massima sovrapposizione = $\min (r_k, r_h)$

minima sovrapposizione = $r_k + r_h + \Delta_t$

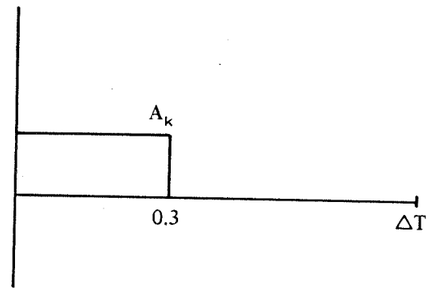
Se il risultato è negativo la minima sovrapposizione vale zero.

4. Esempio

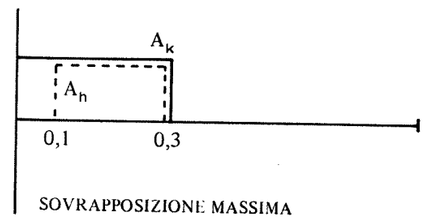
$r_k = 0.3$
$r_h = 0.2$
$\Delta_t = 1$

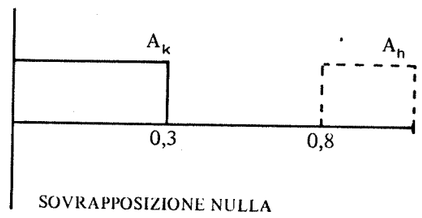
Di conseguenza si ha:
sovrapposizione massima = 0.2
sovrapposizione minima = 0

5. Riferendoci sempre allo stesso esempio, vediamo di analizzare quanto vale la probabilità di avere sovrapposizione minima e sovrapposizione massima. Riportiamo in grafico l'intervallo di tempo Δ_t e la grandezza a_k:

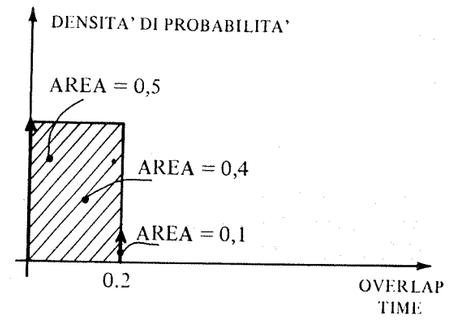


La grandezza a_h può cadere in una posizione casuale rispetto alla grandezza a_k nell'intervallo Δ_t; affinchè si abbia sovrapposizione massima bisogna che la grandezza a_h abbia il fronte di salita compreso tra 0 e 0.1.
Viceversa, per avere sovrapposizione nulla il fronte di salita deve cadere tra 0.3 e 0.8.





E' evidente che la probabilità di avere sovrapposizione massima è 0.1, la probabilità di avere sovrapposizione nulla è 0.5.
In tutti i casi intermedi la probabilità è uniformemente distribuita.
La funzione densità di probabilità avrà un andamento di questo tipo:



6. Il problema è di collegare tutte le distribuzioni di probabilità relative ad ogni Δ_t, per costruire una funzione relativa alla fascia oraria considerata.
Vediamo intanto come collegare due intervalli Δ_t; dobbiamo cioè costruire una funzione definita tra 0 e $2\Delta_t$.
Chiamate f_1 e f_2 le due distribuzioni di probabilità relative rispettivamente al 1° e al 2° intervallo di tempo, chiamiamo f_{12} la funzione risultante. Si ha:

$$f_{12}(t) = \int_{-\infty}^{+\infty} f_1(s) \cdot f_2(t - s)\, ds$$

che corrisponde all'integrale di convoluzione.
Estendendo questo procedimento a tutti i Δ_t si ottiene la funzione desiderata, relativa a tutta la fascia oraria.

7. Eseguendo graficamente il prodotto di convoluzione, si può notare facilmente che la convoluzione fra due funzioni rettangolari (distribuzioni uniformi) origina una funzione triangolare o trapezoidale. Pertanto è logico pensare che eseguendo tante convoluzioni, partendo da distribuzioni uniformi, si ottenga una funzione molto appuntita.
In realtà, mantenendo la stessa scala, la funzione finale non è più appuntita di quelle di partenza; tuttavia, normalizzando la scala ad un valore percentuale, si vede che con poche centinaia di convoluzioni tutta l'area della funzione è concentrata in un intervallo di tempo non superiore al 5% dell'estensione della fascia oraria.

ARCHITETTURA "TAGGED": PUO' L'HARDWARE CONTRIBUIRE ALL'AFFIDABILITA' DEL SOFTWARE?

Luca Gianmaria Sartori

Ist. di Cibernetica — Univ. di Milano

"Language characteristics should lead the hardware people to reflect the fundamental elegance of modern languages into the hardware."

W. M. Mc.Keeman, 1967

Riassunto

L'affidabilità del software può venire incrementata impiegando un sistema dotato di un nuovo tipo di codifica dei dati, che rende possibile un repertorio al tempo stesso ristretto e potente.
Questo lavoro discute i vantaggi ed analizza il disegno della nuova architettura, tenendo conto delle possibilità offerte dall'impiego dei microcalcolatori.
Un problema insorto nell'implementazione delle primitive vettoriali viene risolto attraverso un nuovo concetto firmware, suscettibile di numerose applicazioni nel campo dell'informatica distribuita: i linguaggi locali.

Abstract

Software reliability can be improved with hardware architectures using new techniques for data encoding ('tagged architectures'), in which the type (integer, real, character, array, ...) of each operand is specified within the operand itself, by means of a suitable 'tag'.
This paper discusses the advantages of tagged architectures, and describes a particular architecture based on microprocessor technology.

INTRODUZIONE

L'affidabilità dei sistemi di elaborazione dati è in generale molto alta, ma quando viene commesso un errore, il danno è rilevante. La crescente complessità delle strutture di calcolo ha comportato un aumento della probabilità di malfunzionamenti nonchè della difficoltà dei controlli. Il problema assume una configurazione duplice: l'hardware ha una dinamica precisa, ma è soggetto a deterioramente fisico; il software risulta difficile da mettere a punto, ma è inalterabile.

Fin dal 1956 sono state proposte numerose tecniche per prevenire o correggere guasti hardware, tutte basate sulla introduzione di elementi ridondanti nel disegno dei componenti operativi. Si è passati dalla "modularità tripla" (Triple Modular Redundancy) di von Neumann alla "ridondanza di rivelatori" (Detector Redundancy) di recente concezione (HanY76): il progresso è scaturito dall'impiego crescente di sofisticati codici rivelatori, associati ad adeguati dispositivi di controllo, accanto alla semplice duplicazione dei componenti. Tale duplicazione può rivelarsi inutile in certi casi: il solo uso di codifiche ridondanti e rivelatori locali, con trascurabili modifiche dell'unità di controllo, trasforma un minielaboratore convenzionale in un calcolatore interamente autodiagnostico (SarL77). E' altresì possibile intervenire al livello circuitale con la "logica quadruplicata" (Quadded Logic), un tipo di struttura che i progressi della tecnologia integrata rendono sempre più allettante, nonostante la sua intrinseca complessità.
Questa molteplicità di metodi non è rimasta senza conseguenze sulla pratica della progettazione, specie nel campo aerospaziale, ove ai calcolatori di controllo sono richieste prestazioni di alto livello: il successo delle proposte teoriche è stato pienamente rispondente alle aspettative.

Non altrettanto si può dire nel caso dell'affidabilità del software. I molti sforzi compiuti nel campo della verifica di correttezza dei programmi hanno sortito risultati interessanti solo sul piano astratto, poichè la complessità delle dimostrazioni è molto superiore ai livelli accettabili nel caso di applicazioni di qualche entità. Maggiore successo ha riportato lo stile di programmazione top-down, che introduce una sorta di ridondanza procedurale nel metodo di lavoro, e l'utilizzazione di nuovi linguaggi "problem-oriented", che presuppongono un esteso supporto software per la traduzione automatica.
Tuttavia anche queste innovazioni hanno lasciato aperti molti problemi, come dimostra la quotidiana esperienza dei programmatori. E' infatti noto che ogni nuova copia di sistemi operativi, pronti per la consegna, contiene errori e che, ad esempio, il software per il volo dell'Apollo 14, per quanto accuratamente controllato, rivelò 18 difetti durante i 10 giorni della missione (BoeA73). Nonostante i progressi nell'organizzazione dei team di lavoro, in un grande progetto di programmazione ciascun sistemista produce in media da 5 a 10 istruzioni corrette al giorno, se si tiene conto di tutte le fasi di lavorazione (BakF73). C'è di che essere insoddisfatti.

I problemi del software scaturiscono da un'incongruenza di fondo. L'architettura di von Neumann, che informa la filosofia costruttiva di quasi tutte le CPU dell'ultima generazione, non è il supporto più adatto per i moderni linguaggi di programmazione, nè per le crescenti applicazioni in campi assai diversificati. Ne risultano costrizioni ed inefficienze che comportano sia la complessità degli organi di traduzione, sia l'imbarazzo dei sistemisti, impegnati ad utilizzare un repertorio di macchina lontano dalla loro psicologia e povero rispetto alle loro esigenze.

Occorre dunque che l'hardware si adegui alle istanze poste dal software e si sobbarchi una frazione maggiore del carico computazionale, cercando di riflettere il più possibile la logica dei problemi sui quali è chiamato ad operare. Il trasferimento di funzioni risponde al bisogno di porre il maggior numero possibile di primitive sotto la garanzia di quei dispositivi di protezione la cui efficiacia è stata documentata; l'adeguamento delle strutture alla filosofia delle applicazioni scaturisce da una esigenza più profonda.
L'architettura di von Neumann è "machine-oriented", nel senso che fu concepita con il solo intento di funzionare, cioè di rendere possibile l'uso automatico di una calcolatrice da tavolo secondo i passi di un programma memorizzato. Le applicazioni moderne dell'informatica sono andate ben al di là dell'automatizzazione dell'aritmetica, e per tenerne il passo non ci si può più accontentare dell'interfaccia costituita dai linguaggi "problem-oriented", soggetti alle manchevolezze che conosciamo: occorre un'architettura "problem-oriented".

Il veicolo attraverso il quale si esercita l'influenza della struttura hardware sulla programmazione è l'insieme di istruzioni-macchina. Cercare l'alternativa all'architettura tradizionale equivale nel nostro caso a delineare le caratteristiche del repertorio più desiderabili dal punto di vista della programmazione. Se ne possono individuare 3:

a) Dimensioni ridotte. Ciò comporta la facilità d'impiego delle risorse disponibili e consente al programmatore di essere preciso senza perdersi nei dettagli, con sensibile vantaggio nella fase di correzione.

b) Istruzioni potenti. I programmi sono sintetici e quindi facili da scrivere, da capire, da controllare. Buona parte del lavoro elaborativo si trova, già uniformizzato e verificato, nelle istruzioni stesse.

c) Alta risoluzione diagnostica. Si tratta della capacità di evidenziare gli errori, cioè di rendere possibili approfondite verifiche.

Esistono da tempo alcuni saggi (HanF68) sulla determinazione del repertorio ottimale per una certa applicazione, e così pure è stata recentemente studiata la capacità diagnostica delle istruzioni. Noi seguiremo un'altra via, muovendo dalla rappresentazione "tagged" dei dati: gli operandi vengono resi autoidentificanti con un piccolo campo annesso, detto "tag", che ne indica il tipo. Le modifiche da apportare alla macchina per renderla consistente con questo tipo di codifica e ricavarne organicamente i benefici potenziali individuano l'architettura "tagged".

1. I VANTAGGI DELL'ARCHITETTURA "TAGGED"

L'utilità della rappresentazione "tagged" dei dati è connessa ai vantaggi delle dichiarazioni di tipo, comuni a molti linguaggi ad alto livello. Esse permettono al programmatore di affrontare i problemi applicativi senza doversi preoccupare delle caratteristiche della macchina, ad esempio manipolando stringhe di caratteri invece che configurazioni di bit.
Inoltre, un controllo di tipo sugli operandi in un'espressione consente di rivelare numerosi errori, secondo alcuni fino al 50% del totale (GanJ77). Risulta dunque semplificato sia il processo di codifica, sia la fase di correzione. Un recente studio statistico (GanJ77) rivela che la risoluzione diagnostica è la proprietà più spiccata dei linguaggi dotati di dichiarazioni statiche di tipo. Ne traggono il

massimo vantaggio i programmatori meno esperti, ma la sua utilità è indubbia per tutti.

L'architettura "tagged" realizza l'implementazione cablata delle dichiarazioni di tipo e si propone si superare le inefficienze connesse alla diagnostica in fase di compilazione, per sfruttare appieno le possibilità dei linguaggi ad alto livello. Molti problemi nascono dal fatto che il programma sorgente, usualmente in linguaggio "problem-oriented", è ricco di tipi di dati e deve venire reso eseguibile su una macchina che riconosce un solo tipo, la parola o il byte. Una alta percentuale del lavoro di compilazione o di esecuzione viene assorbito dal compito di costruire numeri reali, vettori e stringhe "virtuali", a partire dai campi di memoria. Sul CDC 7600 alcuni codici Assembler vengono eseguiti al ritmo di quasi 34 milioni di istruzioni al secondo, mentre programmi simili in Fortran "ottimizzato" raggiungono al più il 40% di questa velocità (FeuE73).

Con la rappresentazione "tagged" questi problemi vengono superati, poichè i tipi di dati sono riconosciuti dai circuiti e la compilazione delle strutture di alto livello non comporta alcuna forzatura. I controlli vengono compiuti dall'hardware durante il ciclo di esecuzione mettendo pienamente a frutto la risoluzione diagnostica delle analisi di tipo. Nei casi in cui coesistono operandi diversi nella stessa espressione, in una configurazione ammissibile, vengono eseguite le conversioni in modo automatico: altrimenti compare una segnalazione d'errore, ed il calcolo viene arrestato.

Le dimensioni del repertorio — e quindi quelle del compilatore — vengono ridotte in modo significativo, poichè è sufficiente una sola istruzione per ciascuna classe di operazioni aritmetiche o condizionali. Il codice operativo viene adeguatamente specializzato dal dato col quale interagisce, cosicchè l'unità aritmetica può eseguire l'istruzione completa. I mnemonici "add fixed", "add floating", "add packed", ... vengono rimpiazzati dal solo ADD, che ritorna alla sua forma tradizionale solo nell'ultima fase del ciclo di esecuzione.
Questa concentrazione di molteplici funzioni su poche istruzioni rappresenta un primo passo nell'aumento di potenza delle primitive. Andremo tuttavia molto più avanti considerando i dimensionamenti dei vettori come dichiarazioni di tipo: "**real array** A (0::100)" è concettualmente analogo a "**real** A", e la loro identificazione comporta rilevanti conseguenze pratiche. Ai controlli sull'omogeneità degli operandi nelle espressioni si aggiungono ora quelli sul debordamento degli indici, che consentono di individuare una cospicua frazione degli errori di programmazione. Sul piano della potenza delle singole istruzioni, il progresso compiuto con la incorporazione della somma vettoriale in ADD, del prodotto scalare in MUL, e così via, è indubbio. Altrettanto evidenti sono i vantaggi ottenuti introducendo una serie di operazioni su variabili strutturate, come ricerche o ordinamenti per vettori e primitive per la gestione delle liste: il repertorio acquisterebbe così la potenza di una piccola biblioteca software, adeguata alla complessità delle applicazioni da affrontare.

Le conseguenze di queste innovazioni si ripercuotono sulle prestazioni generali del sistema, oltre che sull'affidabilità dei programmi. Ad esempio nel caso di macchine dotate di una "cache-memory" per una breve sequenza di istruzioni, come i CDC 6600 e 7600, la velocità di esecuzione aumenta di quasi un ordine di grandezza se il tratto di codice costituente un "loop" può essere tutto contenuto nei registri rapidi. In un programma normale, molte istruzioni gestiscono l'accesso alle strutture di dati, per cui difficilmente un ciclo può essere contenuto nella limitata memoria "cache". Se tuttavia si utilizza l'architettura "tagged", i programmi divengono sintetici, facendo aumentare la probabilità di occorrenza di cicli sufficientemente corti e quindi la velocità media di esecuzione.

Si può obiettare che le modifiche proposte per la gestione dei dati possono essere — ed in effetti spesso lo sono — implementate con tecniche software, senza bisogno di supporti circuitali. Ciò che si ottiene con la programmazione, tuttavia, non è altro che l'interpretativo di una macchina ad architettura "tagged", e come tale è inferiore all'originale sul piano delle prestazioni, a fronte di un costo di esercizio nettamente superiore. Sulle motivazioni economiche del nostro progetto torneremo in seguito.

2. TRE TIPI DI ARCHITETTURA "TAGGED"

L'architettura "tagged" fu ideata da John Iliffe nel 1968 per la sua Basic Language Machine (IliJ68): egli propose di utilizzare alcuni bit delle parole di memoria per contenere informazioni sul dato annesso o sul suo stato (ad es. eventuali protezioni) rispetto al sistema operativo. Già in epoca antecedente erano stati costruiti calcolatori, come il Burroughs B 5500, che utilizzavano in modo non convenzionale alcuni bit dei dati, ma i primi esempi di architettura "tagged" nel senso da noi inteso comparvero fra il '69 ed il '70, e furono il B 6500, il

Telefunken T 4 ed il Rice Research Computer R-2 (FeuE72).
I successori della BLM di Iliffe, per quanto realizzati tutti sul medesimo schema, si sono evoluti lungo due direttrici principali, alle quali ne abbiamo aggiunto una terza, che ne compendia i vantaggi con le possibilità offerte dalla tecnologia dei microcalcolatori.

2.1. Architettura "tagged" di base

Le macchine dotate di questa struttura sono le più vicine alla concezione originale; la loro unità centrale è composta da un'unità aritmetico-logica (ALU) "general-purpose", un verificatore di legalità ed un interfaccia di output (fig. 1).
Il verificatore controlla la reciproca compatibilità dei dati in funzione dell'operazione da compiere ed invia all'ALU sia il codice operativo sia il tipo degli input, definendo così completamente l'istruzione. Il "tag" degli operandi viene poi associato al risultato, originariamente di tipo indefinito, da un circuito di interfaccia. Nelle operazioni algebriche può capitare che gli operandi siano di tipo diverso senza che si presenti una configurazione illegale: è il caso delle moltiplicazioni di scalari per vettori. Il verificatore elabora allora separatamente il "tag" del risultato e lo invia all'ALU ed all'interfaccia. Il tipo dell'operazione, in un certo senso, è dunque quello del risultato.
A carico del verificatore sono pure eventuali conversioni automatiche: se un intero viene moltiplicato per un numero in virgola mobile, il primo viene preventivamente convertito in notazione normalizzata, e pure il risultato compare in tale forma. Altre combinazioni di tipi, giudicate inammissibili, danno origine ad un'interruzione, che provvede ad inviare un opportuno messaggio all'utente. La scelta delle associazioni di "tag" che producono una conversione automatica e di quelle che generano un allarme va operata in sede di progettazione, in base al grado di risoluzione diagnostica desiderata. L'architettura si presta ad implementare qualunque livello di severità dei controlli, manifestando così la sua generalità.

La dinamica del calcolo mette in evidenza l'invariante fondamentale della concezione "tagged": la semantica dei dati è statica e dichiarativa, mentre in una macchina di von Neumann è dinamica ed operativa. In un calcolatore tradizionale, la medesima configurazione di bit può venire interpretata come numero complesso, come stringa di caratteri o come istruzione, a seconda delle operazioni a cui viene sottoposta; in una macchina "tagged", il tipo di una variabile viene fissato durante l'assemblaggio, e sono le istruzioni che vengono interpretate in modo diverso a seconda dei dati sui quali operano. La generalità perduta costituisce uno svantaggio solo teorico, perchè in pratica era inutile: incrementare un'istruzione, trattandola come un numero intero, poteva essere utile sui calcolatori della prima generazione, privi com'erano di registri indice, ma oggi è considerato un pessimo vizio di programmazione. D'altronde, tale generalità ricompare nel meccanismo di decodifica delle istruzioni, portando con sè il concreto vantaggio di una contrazione del repertorio.

L'architettura di base presenta due svantaggi, che ne riducono velocità ed affidabilità e ne aumentano il costo: la complessità dell'ALU "general-purpose" e la pesantezza del ciclo di macchina. La sola fase di esecuzione dell'istruzione si presenta infatti come nel diagramma di flusso della fig. 2. Il superamento di questi difetti fornisce lo spunto per gli ulteriori sviluppi.

2.2. Architettura "tagged" ibrida

Un metodo per ridurre la complessità dell'ALU "general-purpose" è sostituirla con una batteria di unità aritmetiche specializzate, in accordo col principio di attribuire funzioni logicamente distinte ad unità fisicamente separate (fig. 3). Si tratta della filosofia sottostante progetti come quello del sistema SYMBOL (CheG71), che pur non rientrando nella categoria "tagged", discende da un'analoga impostazione: i suoi 11 processori centrali realizzano funzioni usualmente demandate al software dei sistemi operativi.
La prima parte del processo di calcolo non ha subito modifiche; a valle del verificatore, tuttavia, è stato inserito un decodificatore che interpreta il "tag" globale come un indirizzo, e seleziona con un "enable" l'unità aritmetica da attivare. I dati veri e propri ed il comando da eseguire possono indifferentemente venire inviati a tutte le unità o solo a quella prescelta, attraverso il decodificatore. Inoltre, sono presenti tante interfacce d'uscita quante sono le ALU: esse non comunicano col verificatore ed aggiungono sempre lo stesso "tag" al risultato.
La chiave di questa struttura è il decodificatore: a monte, il "tag" è una parte del dato e ne modifica la semantica; a valle, diviene un indirizzo e ne influenza solo la destinazione. Per questo si parla di architettura ibrida: il "tag" assume alternativamente il ruolo di parte del dato e di indirizzo dell'organo operativo. Il ciclo di esecuzione

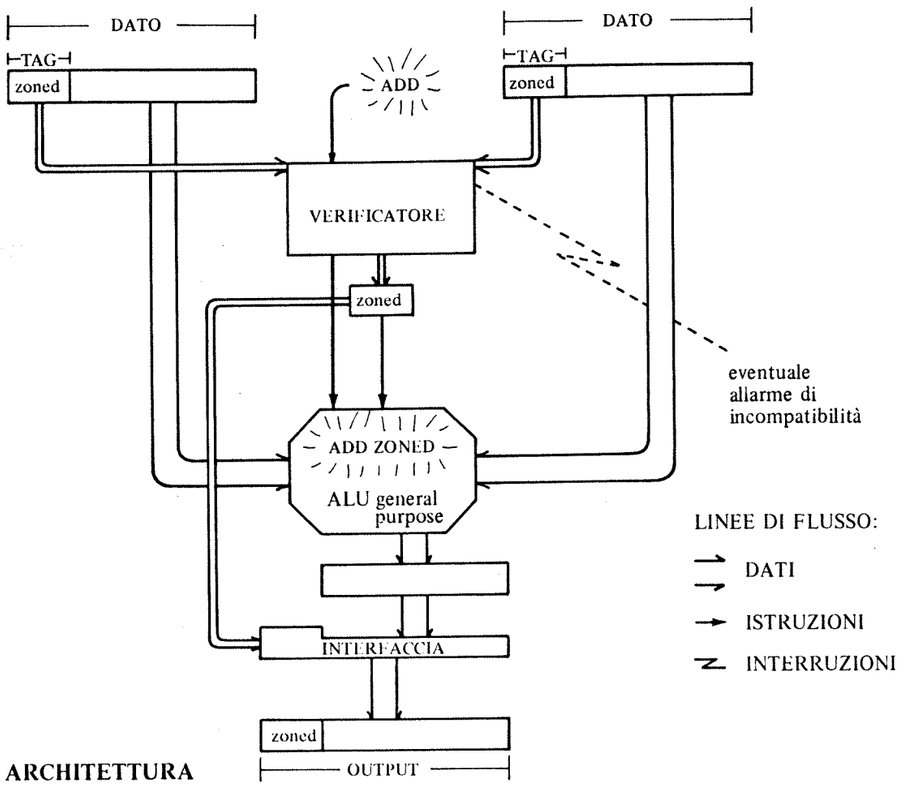


Fig. 1 ARCHITETTURA 'TAGGED' DI BASE

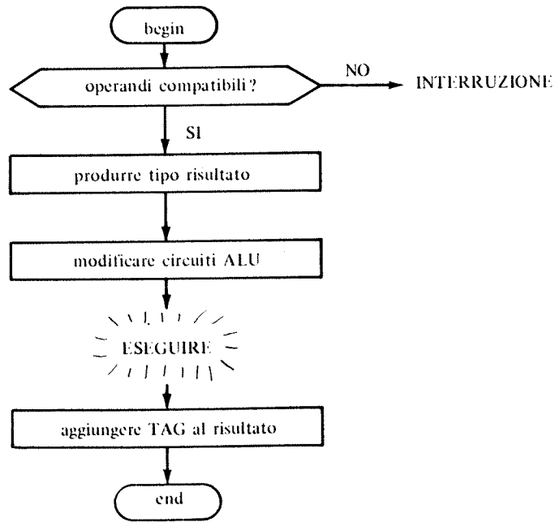
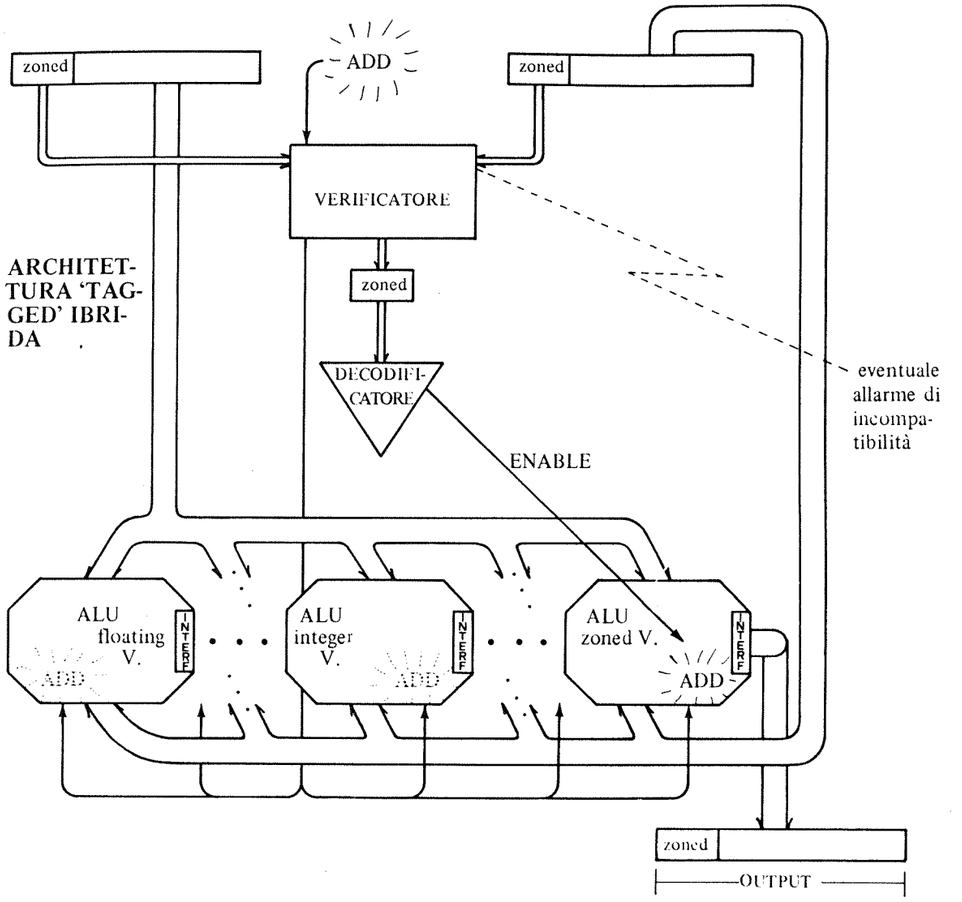


Fig. 2 SUO CICLO DI MACCHINA

Fig. 3 ARCHITETTURA 'TAGGED' IBRIDA



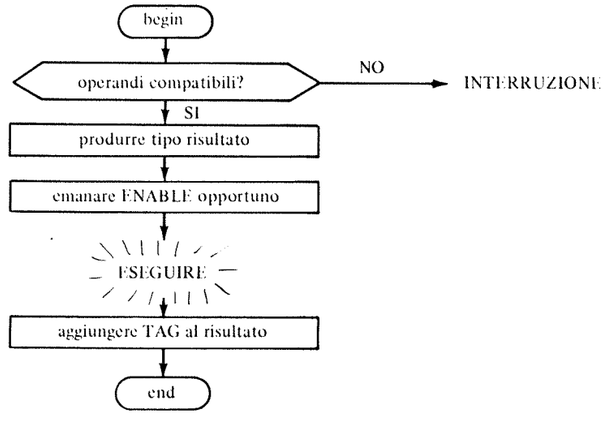


Fig. 4 SUO CICLO DI MACCHINA

dell'istruzione è sostanzialmente uguale al precedente (fig. 4).
Assai simile nelle macchine ad architettura "tagged" di base ed ibrida è pure il repertorio. Esso deve contenere codici operativi come Load Real Tag (of), Load Integer Tag (of) etc., da cui derivano mnemonici come LRT, LIT etc. Un caratteristico estratto di codice Assembler verrebbe così tradotto in linguaggio macchina:

```
                    A RES 1 (riservare una parola)
integer A, B        B RES 1 (riservare una parola)
     •                LIT A
     •                LIT B
     •                 •
     •                 •
move A,B             MOV A,B
```

Le dichiarazioni di tipo originano quindi vere e proprie istruzioni eseguibili e non solo direttive di assemblaggio.

2.3. Architettura ad indirizzi "tagged" o stellare

Analizzando la zonatura logica della memoria nei sistemi precedenti, si nota che è suddivisa in compartimenti disgiunti. Se a tale distinzione fa riscontro una separazione reale, si ottiene una configurazione nuova, poichè si può associare ad ogni processore la sezione di memoria che effettivamente gestisce. Il sistema risulta così costituito da calcolatori locali specializzati indipendenti tra loro: verificatore ed interfaccia di output non servono più e scompaiono.
Il ruolo del "tag" in questa architettura è interamente diverso da quello svolto nelle macchine precedenti. In una memoria ove tutti i dati sono dello stesso tipo, non ha senso definire un campo che lo dichiari, mentre si rende indispensabile un'appendice dell'indirizzo che specifichi quale calcolatore locale vada attivato. Il precedente estratto di codice sorgente verrebbe reso in linguaggio macchina da

MOV (int,A),(int,B)

In questo senso parleremo di macchina ad indirizzi "tagged"; il termine "stellare" deriva dalla configurazione grafica delle varie sezioni, collegate fra di loro soltanto attraverso il controllo (fig. 5).
La semantica del dato ha così compiuto una metamorfosi completa: era funzionale nella macchina di von Neumann, dichiarativa nell'architettura "tagged" di base ed ibrida, ed è divenuta topologica in un calcolatore stellare.

Il ciclo di macchina di un elaboratore così concepito differisce da quello dei sistemi convenzionali solo per la diramazione iniziale di un "enable" alla sezione che si vuole attivare. Il carico di operazioni svolto negli altri schemi da verificatore ed interfaccia di output è stato assorbito in parte dall'assemblatore, che compila gli indirizzi, ed in parte dalla logica dell'architettura. E' stato così superato anche il secondo svantaggio attribuito ai sistemi "tagged". Le istruzioni vengono divise in due parti prima dell'esecuzione: il "tag" serve ad impostare il decodificatore, mentre il resto perviene alle unità operative del sistema, grazie ai bit di informazione "persi" lungo il tragitto. Il campo "tag" può venire anch'esso trasmesso oltre il decodificatore per effettuare alcuni controlli, ma ciò non è strettamente necessario.
Il linguaggio Assembler di questa macchina non può contenere alcun riferimento esplicito ai registri, poichè la posizione di quelli effettivamente utilizzati dipende dal tipo dei dati, ed il programmatore non dovrebbe occuparsene. Il formato più adatto è quindi quello a tre indirizzi di memoria per le operazioni algebriche (OPER src 1, src 2, dst), poichè è in grado di specificare completamente ogni trasformazione usando solo i nomi simbolici delle variabili. Tutti i passaggi intermedi risultano trasparenti per il programmatore.
La struttura modulare di una macchina stellare la rende particolarmente adatta ad un'implementazione con microcalcolatori, come risulterà chiaramente dall'esame di un caso concreto.

3. LA RAPPRESENTAZIONE DELLE VARIABILI STRUTTURATE

L'implementazione delle variabili strutturate in una macchina ad architettura stellare non è immediata, poichè le soluzioni intuitive portano a pesanti complicazioni. Non si può, ad esempio, impiegare una memoria autonoma per i vettori, poichè in tal modo cadrebbe la distinzione fra vettori reali, vettori interi etc., che è fondamentale in un disegno basato sull'analisi dei tipi. D'altronde sarebbe assurdo utilizzare tante memorie quanti sono i vettori di diverso tipo, poichè non si otterrebbe che un doppione della macchina di base, con grossi problemi di coordinamento e comunicazione.
Analogo, ma più complesso ancora, è il caso delle liste, strutture di cui i vettori sono casi particolari. Queste difficoltà scaturiscono da un problema di fondo. La variabile definita dalla dichiarazione

real array A (1::100)

é dotata di **due** tipi, distinti ma connessi: è un "ar-

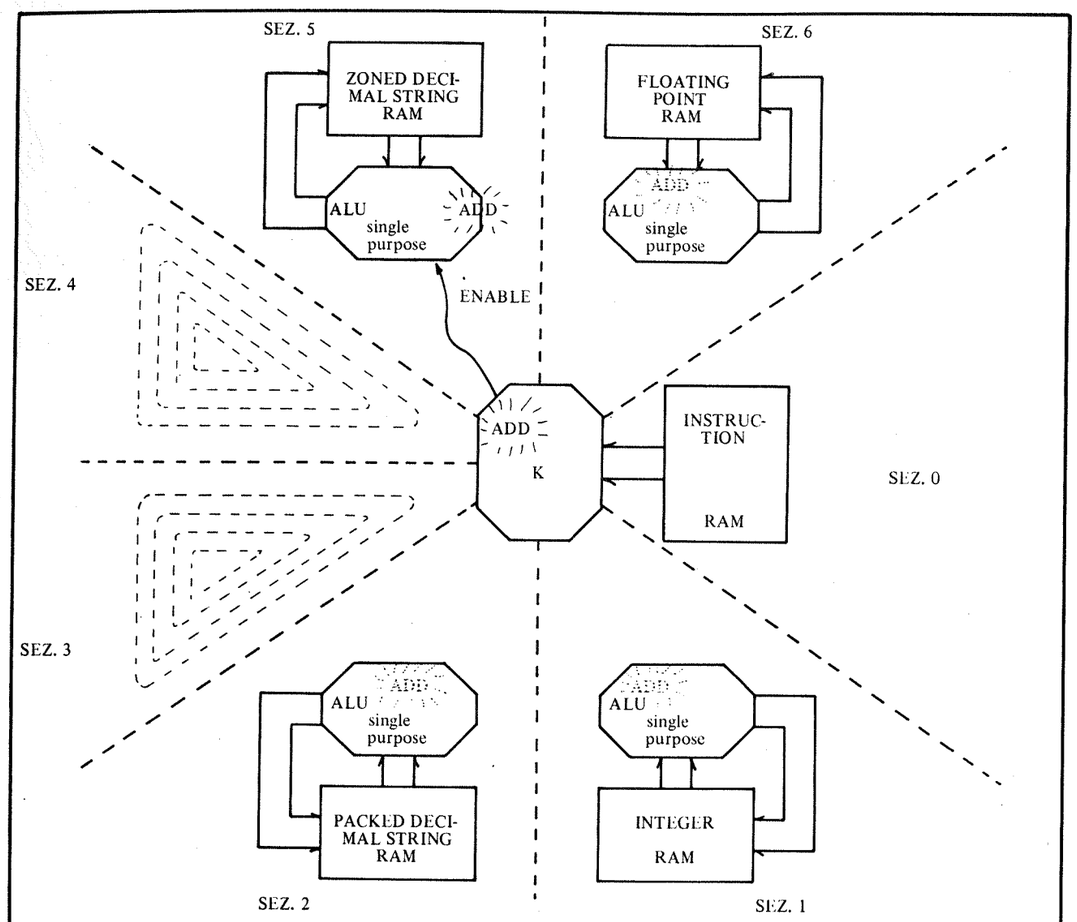


Fig. 5 ARCHITETTURA STELLARE

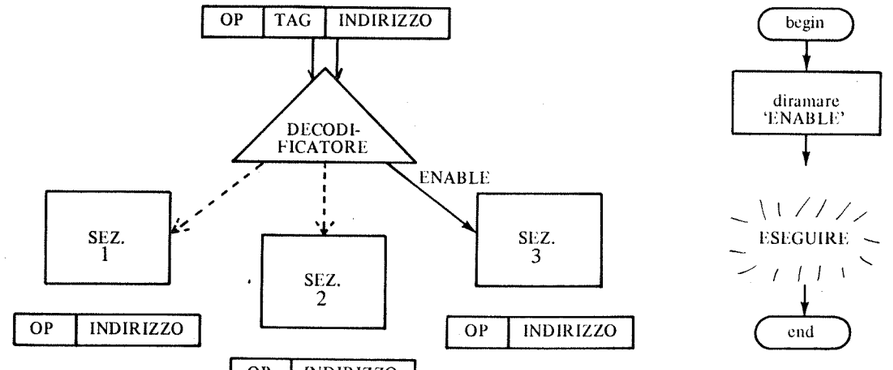


Fig. 6 DINAMICA DI DECODIFICA E CICLO DI MACCHINA RELATIVI

ray'', cioè un'entità organizzata su di una struttura interna, ed è ''real'', cioè i suoi componenti si presentano in notazione normalizzata. Due ''tag'', e quindi due processori locali, sono necessari per gestire la variabile secondo la nostra logica: uno di essi è presente nel disegno di base, il secondo va aggiunto per funzionare come interfaccia. Le operazioni concorrenti delle due sezioni rendono possibile la gestione della variabile nel contesto dell'architettura stellare, purchè nel loro coordinamento si tenga conto della gerarchia fra i due tipi.
La struttura indiciata di A è infatti prioritaria rispetto al tipo dei suoi elementi, poichè ne definisce l'esistenza: occorre calcolare l'indirizzo di A(i), prima di trasformarlo algebricamente. ''Array'' viene prima di ''real'': questa precisazione ci permette di distinguere tra le operazioni che un processore vettoriale può mutuare dalle sezioni già esistenti e quelle che gli sono proprie. Esse si riducono ai calcoli sugli indici ed alla trasmissione di indirizzi alle sezioni numeriche, ove il vettore è materialmente riportato. La memoria della sezione vettoriale contiene un record logico, che chiameremo ID (identificatore), con tutte le informazioni necessarie alla gestione degli indici, per cui le funzioni di indirizzamento da calcolare sono sempre caratterizzate dalla modalità indiretta indiciata. Il puntatore ID è suddiviso in 5 campi, che indicano rispettivamente dimensione, tipo, indirizzo, limite inferiore e superiore degli indici del vettore identificato. L'indirizzo simbolico A della struttura corrisponde all'indirizzo locale di questo record logico nella memoria per vettori, riferito alla prima parola occupata.
Matrici e volumi vengono rappresentati sequenzialmente nelle memorie numeriche, ma il puntatore globale e gli algoritmi di indirizzamento del processore vettoriale rendono possibile fare riferimento alla loro struttura matematica, senza occuparsi della configurazione dei dati in macchina (fig. 7).

La rappresentazione di una lista (fig. 8) generalizza quella dei vettori e si basa sul fatto che un record logico può essere considerato un vettore ad elementi non uniformi, ed una lista come un vettore di record uniformi. La possiamo pertanto scomporre in un record di vettori uniformi, in ognuno dei quali riuniamo gli elementi omologhi dei record primitivi. Ciascuno di questi vettori ''rimescolati'' trova posto in una memoria numerica e la sua posizione si può ricostruire con l'informazione contenuta in un identificatore, analogamente al caso dei vettori semplici. La generalità di funzionamento dei puntatori rende possibile l'occorrenza di un vettore ordinario come elemento di un record.
L'identificatore è composto da 2 + 2N campi, se N sono gli elementi del generico record primitivo. Occorre infatti memorizzare N stesso ed il numero di record presenti; inoltre, per ognuno di essi, va immagazzinato il tipo comune e l'indirizzo locale d'inizio del vettore uniforme derivato.
Eventuali contrassegni di fine messaggio, negli identificatori per vettori e per liste, dipendono dall'organizzazione hardware delle rispettive memorie.

4. PANORAMICA DI UN CALCOLATORE AD ARCHITETTURA STELLARE

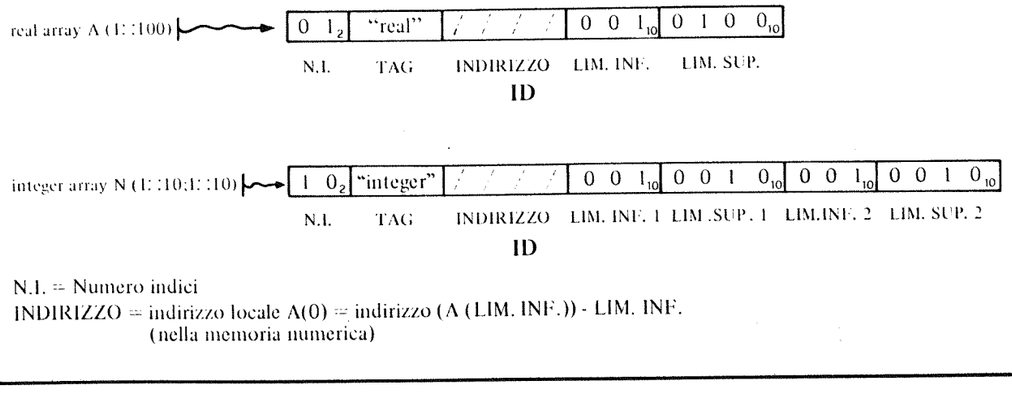
Analizziamo ora una macchina stellare concreta, capace di elaborare variabili reali ed intere, vettori e liste, nonchè stringhe del codice BCD (fig. 10). Essa è stata dotata di numerosi canali per sottolineare la possibilità del suo impiego in applicazioni richiedenti elevate prestazioni, e di un processore di conversione sprovvisto di memoria indirizzabile, una sorta di canale interno, che trasforma stringhe in numeri con metodi tabulari e reali in interi con le consuete routines.

La novità maggiore di questo sistema rispetto a quello teorico consiste nella sua struttura ''doppiamente'' stellare: le sezioni scalari sono organizzate intorno a CVP, le altre intorno ad un canale collegato asimmetricamente alla memoria degli interi. Ciò è dovuto al fatto che le macroistruzioni e gli identificatori di ''array'' vengono assemblati come numeri in virgola fissa e trasmessi alle rispettive sezioni in gruppi di lunghezza variabile. In altri termini, un identificatore vettoriale può essere costituito da una sequenza di quattro interi, che VP è in grado di interpretare correttamente senza conversioni intermedie: pertanto CVP resta fuori dal flusso di informazioni e CH 5 deve solo gestire gli allineamenti. Questo è l'unico caso di operazioni al di fuori della logica ''tagged'' nel sistema: l'eccezione è resa indispensabile dalla separazione funzionale che abbiamo fatto seguire alla distinzione dei tipi. MK, VP ed LP non dispongono dell'hardware necessario a compiere le operazioni di assemblaggio nemmeno per quanto si riferisce alle variabili di loro competenza.

Altra innovazione rispetto allo schema di fig. 5 è la perdita della collocazione centrale da parte del processore di controllo. Si tratta tuttavia di un mutamento apparente, dovuto all'evidenziamento del flusso dei dati invece che del controllo. Nella macchina reale, istruzioni ed interruzioni si muovono lungo alcuni bus che non sono stati indicati.

La configurazione esemplificativa che abbiamo scelto è più che minimale: il sistema è in grado di

Fig. 7 – RAPPRESENTAZIONE VETTORI E MATRICI



N.I. = Numero indici
INDIRIZZO = indirizzo locale A(0) = indirizzo (A (LIM. INF.)) - LIM. INF.
(nella memoria numerica)

record : : = name : string; value : real; next : integer
file : : = array of record (1 : : 100)

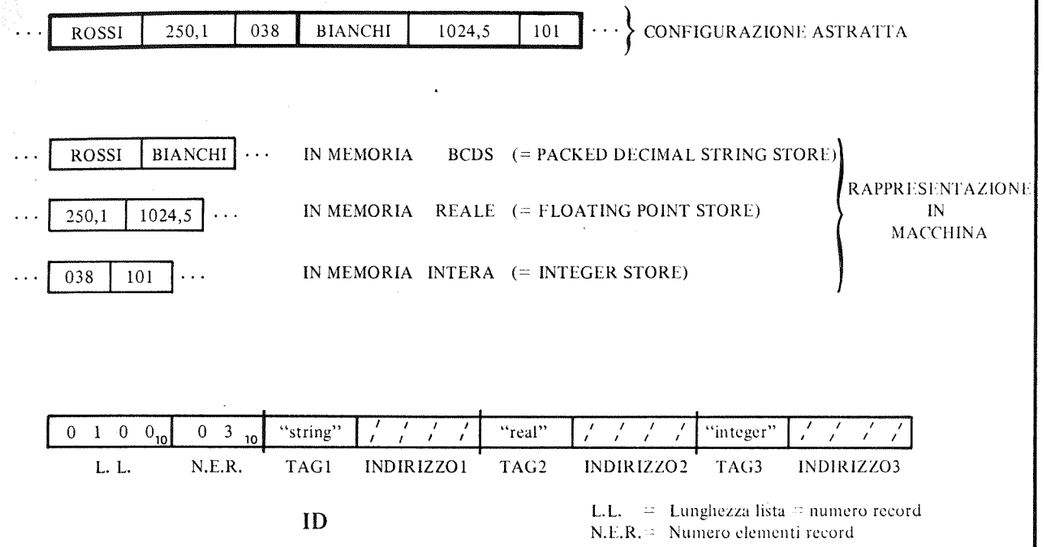


Fig. 8 – RAPPRESENTAZIONE LISTE

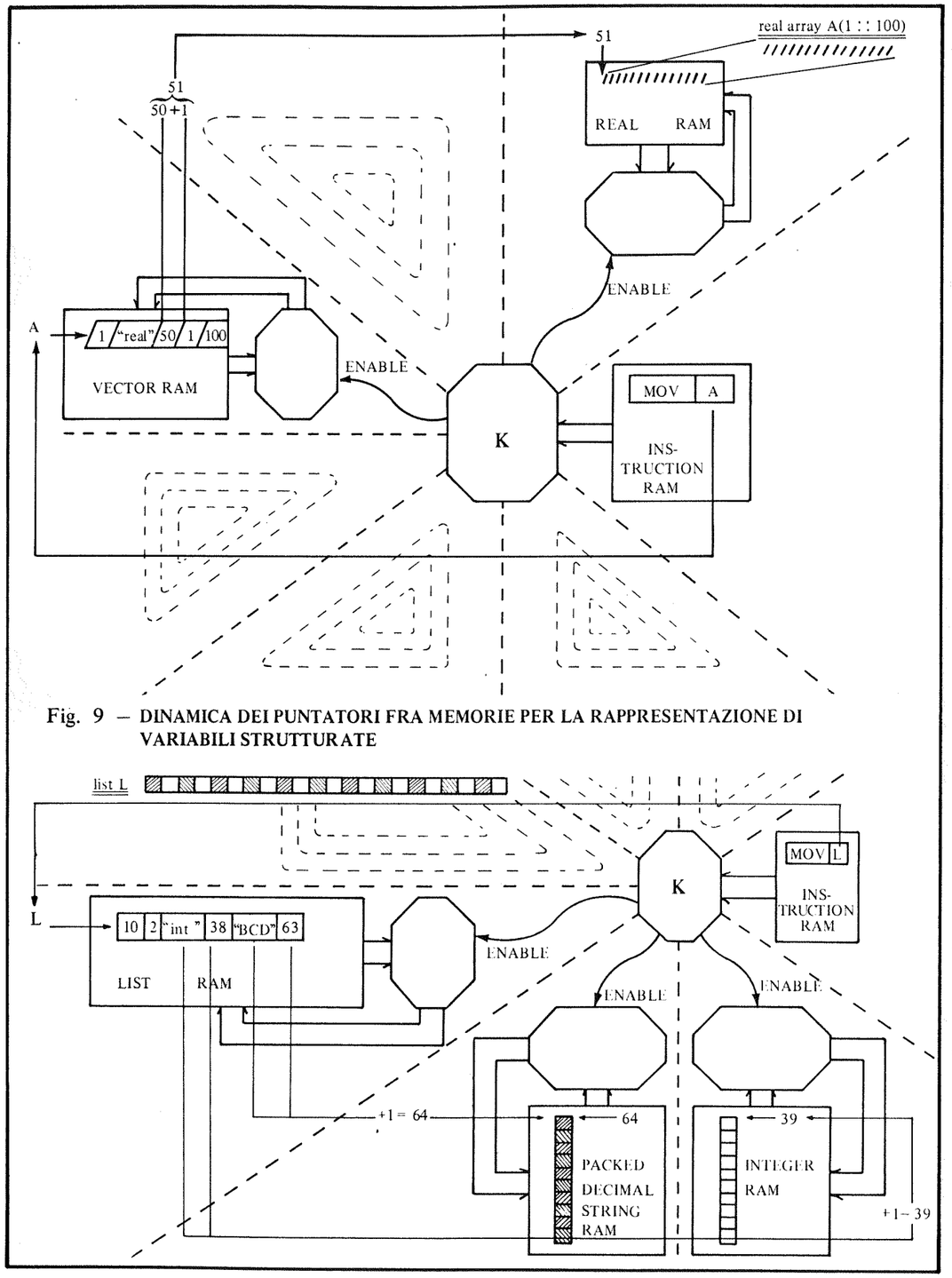
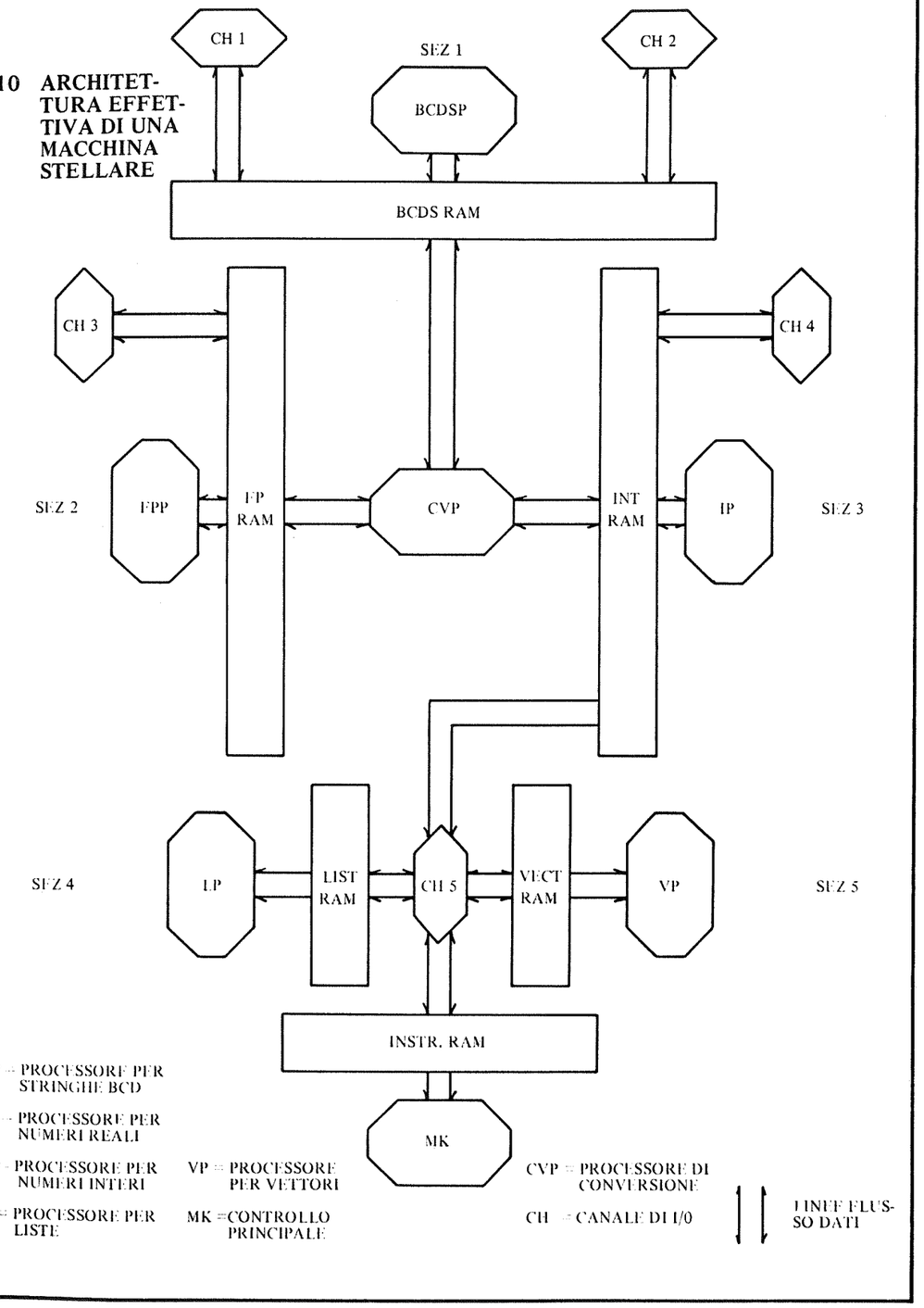


Fig. 9 — DINAMICA DEI PUNTATORI FRA MEMORIE PER LA RAPPRESENTAZIONE DI VARIABILI STRUTTURATE

Fig. 10 ARCHITETTURA EFFETTIVA DI UNA MACCHINA STELLARE



svolgere tutte le operazioni fondamentali della computazione, dagli assemblaggi alla gestione delle interruzioni, e di adempiere sia ai compiti di un calcolatore commerciale sia a quelli di una macchina per applicazioni scientifiche. Con questa generalità abbiamo inteso illustrare le vaste potenzialità dell'architettura stellare.

4.1. Relazioni di priorità e loro implementazione

La realizzazione di una macchina a struttura stellare presuppone la soluzione di un problema di coordinamento. I processori specializzati, dal cui funzionamento concorrente scaturisce l'elaborazione globale di alto livello, sono sottoposti a precise relazioni di priorità: ad esempio, il processore vettoriale opera in modo privilegiato rispetto alle sezioni scalari. Più precisamente, conveniamo che il processore A sia gerarchicamente superiore al processore B se e solo se la struttura di dati gestita da A è più generale di quella gestita da B.
In tal modo, l'ordinamento d'astrazione delle variabili si riflette in modo semplice sulle priorità fra le sezioni corrispondenti, ed i 6 calcolatori considerati si collocano su 4 piani distinti (fig. 11). Il livello più alto, associato al controllo centrale, si può pensare riferito alla variabile "programma".
Com'è possibile realizzare nella pratica queste relazioni di priorità fra processori distinti?

Faremo riferimento ad un'implementazione del sistema basata su microprocessori, poichè questo supporto tecnologico è il più adatto alla modularità di concezione del progetto. Otteniamo dunque una rete di microprocessori, dotati di propri linguaggi firmware locali: sono trasparenti per il programmatore e ne chiameremo microordini le istruzioni.

I linguaggi locali coincidono nelle sezioni scalari limitatamente alle operazioni comuni e sono assai lontani dal linguaggio globale del sistema, formato da macroordini. Tale differenza è tanto più spiccata quanto più siamo riusciti nell'intento originale di costruire un repertorio semplice e potente al tempo stesso.

I compiti di calcolo sono così suddivisi fra i vari processori:
— MK gestisce istruzioni di controllo ed interruzioni globali;
— VP ed LP calcolano le funzioni di indirizzamento più complesse;
— FPP, BCDSP, IP eseguono operazioni aritmetiche e test.

La dinamica del calcolo è relativamente semplice nelle sue linee generali. MK inizia ad eseguire un macroordine in un macrociclo asincrono; se non si tratta di un'istruzione di controllo, lo trasmette in blocco alla sezione interessata, che provvede a ricercare nella propria biblioteca firmware la microroutine corrispondente e ad eseguirla. Al termine dell'esecuzione di queste istruzioni, la sezione restituisce a MK eventuali risultati logici nel registro di condizione, accessibile a tutti i processori, ed invia il segnale di terminazione ad MK, restituendole il controllo.

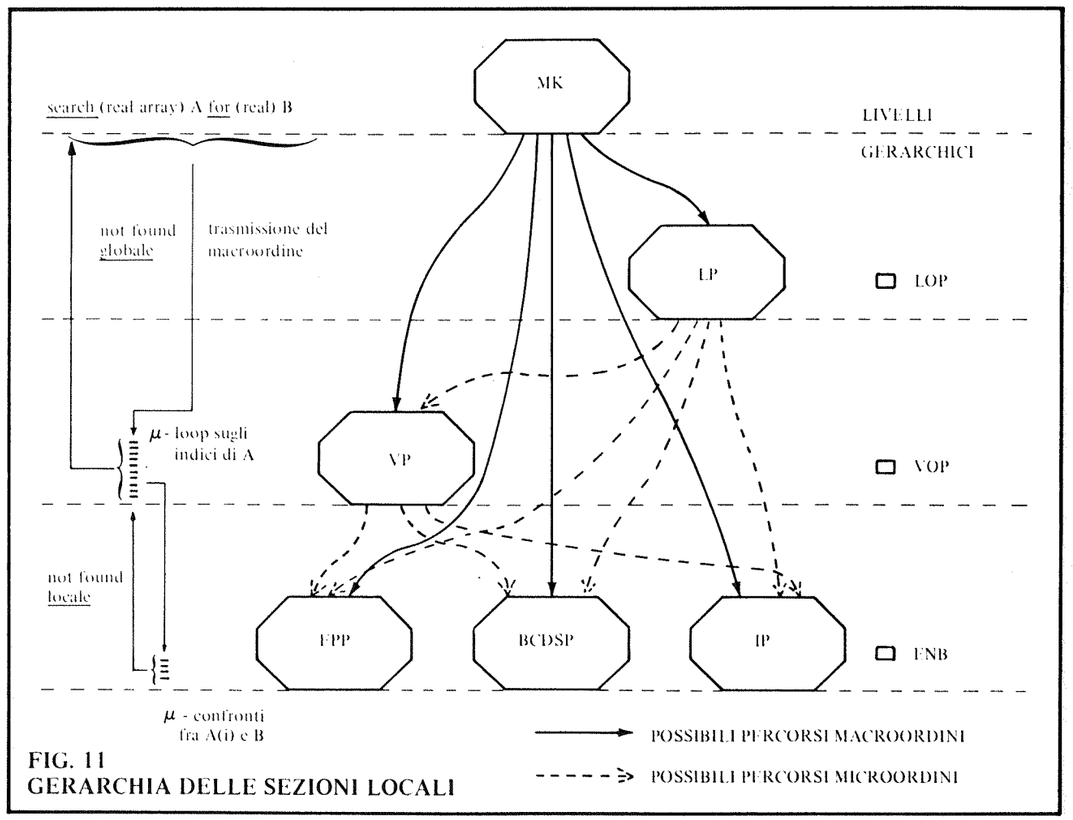
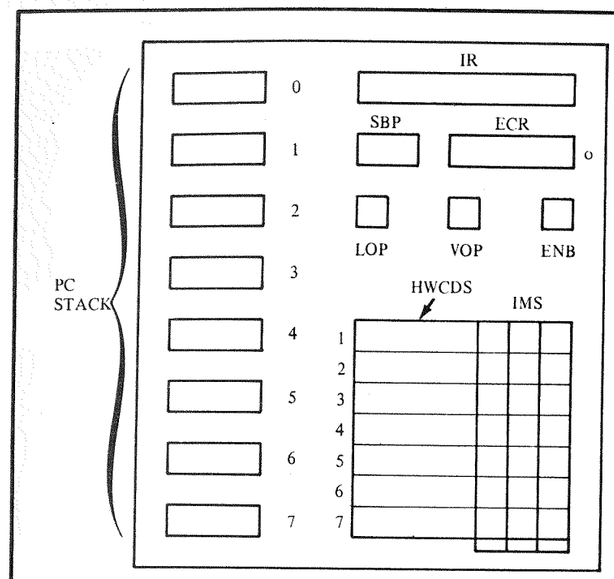


FIG. 11
GERARCHIA DELLE SEZIONI LOCALI



IR = REGISTRO ISTRUZIONI GLOBALI

PC STACK = CATASTA DI CONTATORI ISTRUZIONI

HWCDS = CATASTA HARDWARE PER REGISTRO DI CONDIZIONE

SBP = PUNTATORE FONDO CATASTE

ECR = REGISTRO DI CONDIZIONE ESTESO

LOP = OPERAZ. SU LISTA IN ESECUZIONE

VOP = " " VETTORE IN "

ENB = " " SCALARE " " (ENABLE)

IMS = REGISTRI PER ARMARE, ABILITARE ED ATTIVARE INTERRUZIONI

FIG. 12 REGISTRI DI MK

Questa semplicità è tuttavia vanificata dal problema cui accennavamo in precedenza. In conseguenza delle diverse strutture di dati ammissibili, si possono stabilire varie connessioni tra le sezioni di calcolo, ferme restando le rispettive relazioni di priorità. Tranne LP, tutti i processori possono venire comandati sia da MK sia da altre unità, e nei due casi i risultati delle elaborazioni eseguite localmente devono seguire itinerari distinti. In qual modo demandare questa differenziazione ai circuiti?

Sarebbe troppo costoso espandere i repertori locali per tener conto delle diverse destinazioni, come pure sarebbe complesso affidarsi ad una gerarchia di interruzioni. Abbiamo adottato la soluzione di comandare i processori con un macroordine nel caso che l'istruzione provenga da MK, e con un microordine del loro repertorio nel caso provenga da un processore intermedio. Questo metodo ci è sembrato il più semplice concettualmente per la simmetria che introduce tra il livello dei linguaggi impiegati e la rilevanza dei risultati ottenuti. La sua implementazione è resa più semplice dalla natura microprogrammata dei microcalcolatori impiegati e dalla disponibilità di registri distinti per le istruzioni: IR per i macroordini, IEXR per i microordini provenienti da LP ed IIEXR per quelli generati da VP. Le unità che possono trasmettere microordini ad altre sezioni, cioè VP ed LP, sono dotate di un altro registro per formarli, chiamato OEXR.

Per quanto emessi da unità intermedie, i microordini hanno la precedenza sugli altri segnali di controllo: se una sezione abilitata riceve simultaneamente un'istruzione locale ed una globale, esegue la prima accantonando la seconda. Anzi, se riceve soltanto un'istruzione globale contenente un "tag" corrispondente ad un processore di priorità maggiore della propria, si pone in stato di attesa finchè non le perviene una microroutine, ed in nessun caso tenta di eseguire il macroordine.
Ciò si rende necessario perchè nel caso di operazioni su "array", il processore di priorità maggiore coinvolto assume il controllo del sistema, e le unità sottoposte non devono iniziare esecuzioni indipendenti, pur ricevendo un macroordine ed un "enable", per non porre insolubili problemi di coordinamento: il macroordine assume per esse il significato di un avviso sull'imminente arrivo di più precise direttive.

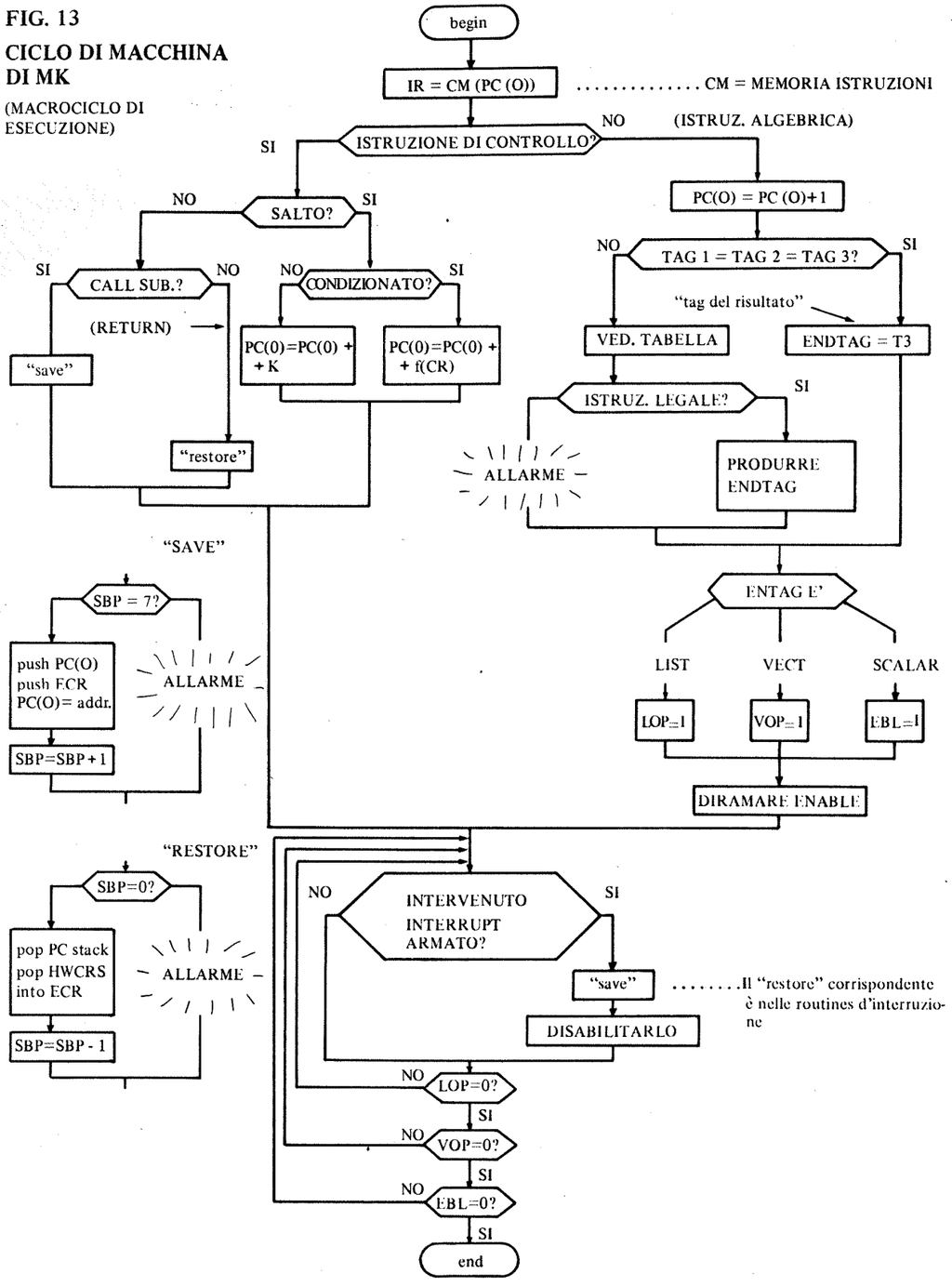
4.2. MK, il processore "master"

La struttura distribuita del sistema viene resa organica da pochi flip-flop, riuniti nel registro di condizione esteso ECR, nonchè dai tre "flag" logici LOP, VOP ed EBL. Tutte le sezioni sono collegate ad ECR: quelle scalari solo ai suoi ultimi tre bit, che fungono da segno dell'accumulatore, carry ed overflow; VP al quarto, che indica il verificarsi di condizioni logiche, ed al quinto, che segnala eventuali debordamenti di indici; LP ad altri 3, che svolgono compiti analoghi. Un'unica sezione, MK, è invece in grado di "leggere" ECR, cioè di modificare le proprie operazioni in base ai suoi contenuti: a questa proprietà deve la sua collocazione gerarchica prioritaria. In altri termini, MK è l'unico processore in grado di eseguire le istruzioni di controllo del codice sorgente; le altre unità possono eseguire cicli di program-

FIG. 13

CICLO DI MACCHINA DI MK

(MACROCICLO DI ESECUZIONE)



ma nell'ambito dei microordini, senza riferimento alcuno con le diramazioni condizionali del linguaggio globale.

MK non detiene il controllo in permanenza, bensì lo cede temporaneamente ad altre sezioni per l'esecuzione di operazioni algebriche, emettendo alcuni segnali di abilitazione attraverso il decodificatore previsto teoricamente: trasmette inoltre il macroordine in corso di esecuzione a tutti i processori, senza privarlo dei "tag" che contiene. I bit LOP, VOP ed ENB servono a gestire i tempi di attesa all'interno del macrociclo di esecuzione; vengono posti ad 1 da MK stesso ed azzerati dalle altre unità al termine dei loro cicli. (fig. 13). La condizione ENB = 1 indica che è stata abilitata una sezione scalare, mentre VOP si riferisce a VP e LOP ad LP. Tra questi bit si instaurano le stesse relazioni di priorità che sussistono tra i relativi processori: così, se ENB viene posto a 0 ma VOP rimane ad 1, il macrociclo rimane in condizione di attesa. Tale configurazione significa infatti che l'operazione vettoriale in corso ha terminato l'utenza di una sezione scalare senza essere essa stessa conclusa. Eventuali risultati logici intermedi compaiono regolarmente in ECR, ma non possono venire utilizzati finchè non divengono definitivi, cioè VOP viene riportato a zero.

Mentre la dinamica interna dei vari processori può essere sincrona o asincrona, il funzionamento globale della macchina deve essere asincrono: il tempo di esecuzione dei macroordini è molto irregolare, per cui l'impiego di un orologio per la generazione di impulsi comporterebbe gravi inefficienze. Il coordinamento delle funzioni viene ottenuto con opportune interruzioni, raggruppate in 4 tipi:

— segnali di "enable" e relative risposte;

— interruzioni globali;

— allarmi;

— interruzioni locali.

Messaggi di abilitazione e relative risposte vengono generati e ricevuti in tutti i macrocicli che non eseguono istruzioni di controllo e fanno quindi parte del funzionamento standard di MK. Le interruzioni globali segnalano circostanze straordinarie e vengono gestite con il metodo delle subroutines: il supporto hardware necessario è rappresentato dall'usuale catasta di registri, in cui l'elemento di testa funge da contatore delle istruzioni. In un'altra catasta, impiegata parallelamente alla prima, vengono salvati i contenuti del registro di condizione.

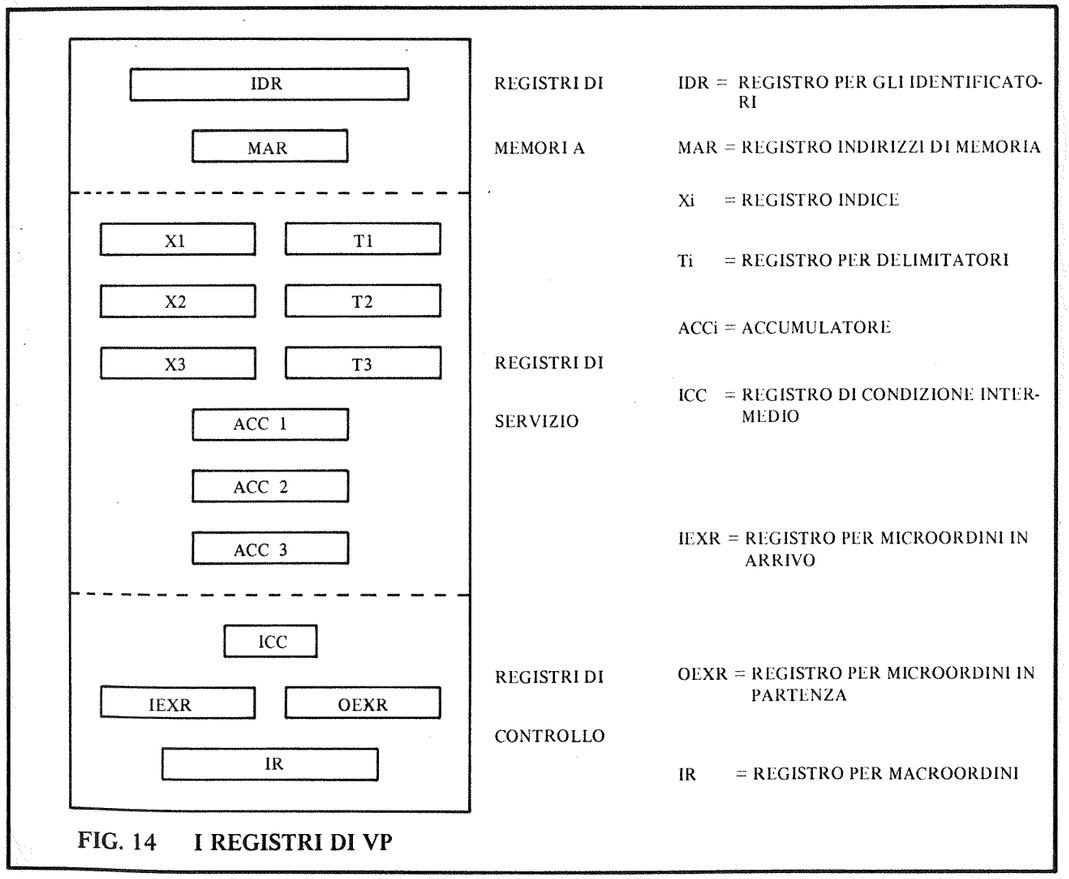


FIG. 14 I REGISTRI DI VP

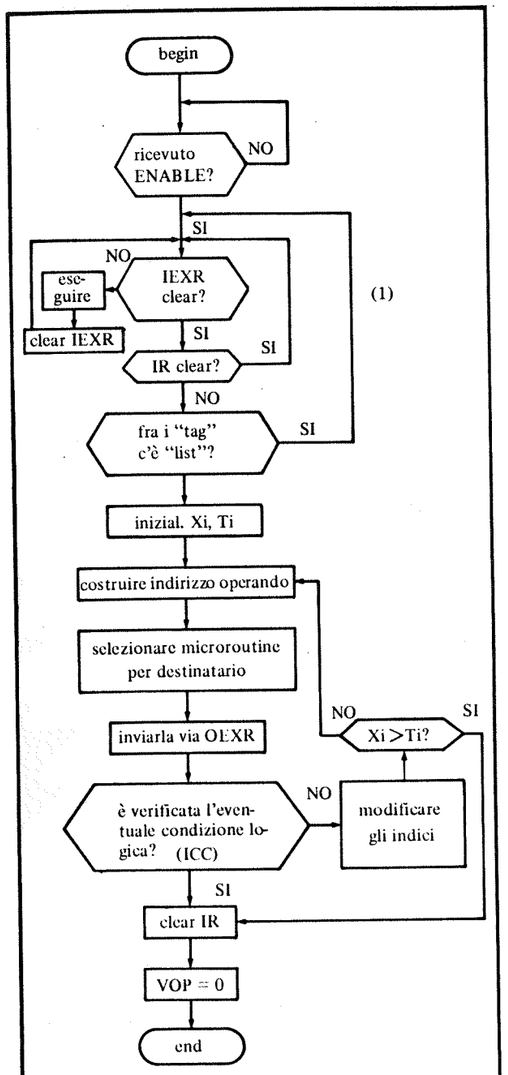


Fig. 15 – CICLO DI MACCHINA VP

La tecnica delle pile hardware per la gestione di subroutines ed interruzioni è piuttosto diffusa: ciò che differenzia il sistema stellare dagli altri è il fatto che le cataste devono essere obbligatoriamente contenute per intero nel processore di controllo. Infatti il tipo dei dati da salvare in esse è "boolean", che non rientra fra i "tag" riconosciuti: dunque nessuna sezione di memoria è predisposta a ricevere le immagini del registro di condizione o del contatore delle istruzioni. Ciò può sembrare una limitazione, ma in realtà non lo è.

Le interruzioni globali intervengono al termine dell'operazione in corso e non interrompono il macrociclo di macchina. Lo interrompono invece gli allarmi, interruzioni molto veloci che vengono gestite a livello del microciclo locale di MK e fanno fronte agli incidenti di calcolo più gravi. Pure a carico dei microcicli d'esecuzione sono le interruzioni locali, legate ad esigenze particolari dei vari processori: il loro insorgere non comporta conseguenze ai livelli superiori d'astrazione.

4.3. Osservazioni

Non è stata riservata alcuna specifica risorsa di calcolo alle variabili logiche, poichè vengono gestite autonomamente da ciascun processore, secondo le proprie necessità. I flag logici del registro di condizione sono di competenza di MK; le operazioni di mascheramento vengono eseguite solo da BCDSP e CVP all'interno di procedure più complesse; gli shift di normalizzazione sono a carico di FPP, ma non vengono mai richieste isolatamente. In altri termini, le operazioni di "shift", "and", "or" etc. fanno parte del repertorio locale delle varie sezioni, ma sono troppo specializzate per comparire nel linguaggio globale, che deve risultare di livello più alto possibile.

VP ha il ciclo di macchina più complesso dopo quello di MK, poichè è l'unica unità dotata di processori sia inferiori sia superiori nella gerarchia definita; conseguentemente, la sua sezione di controllo è quella più tipica, quanto all'impiego dei registri, anche se è priva di IIEXR. Ad LP manca IEXR; ad IP, FPP e BCDSP manca OEXR, ed il ciclo di macchina relativo è molto più semplice.
Il test sul tipo che porta al "loop" di attesa (1) (fig. 15) serve a sincronizzare l'avvio del ciclo di macchina con l'arrivo dei microordini da LP, nel caso debba venire eseguita una operazione globale mista fra liste e vettori. Esso fornisce un esempio dell'utilità di trasmettere ai processori periferici anche i "tag" contenuti nelle istruzioni; ne profitta non solo la ricchezza dei test diagnostici, ma anche la semplicità dei cicli di macchina. In questo caso, l'inscatolamento dei due **whiledo** iniziali è una conseguenza della maggiore priorità dei microordini esterni rispetto agli ordini globali.

I 3 registri indice di VP, Xi, permettono di ricalcare fedelmente la struttura matematica di matrici e volumi, mentre le necessarie conversioni per accedere alla struttura lineare di macchina sono a carico del firmware locale, che utilizza a tal fine gli accumulatori ACCi. I registri Ti indicano i limiti superiori di variabilità degli indici. Gli Xi vengono inizializzati al valore del limite inferiore; al termine di ogni ciclo, allorchè Xi viene incrementato, si effettua un confronto con Ti che può portare all'interruzione del calcolo. Il controllo viene restituito ad MK anche nel caso che l'unità scalare coinvolta faccia scattare un particolare bit nel registro di condizione ICC, collegato con una sezione del corrispondente ECR di MK.

4.4. Esempi di computazioni

Per comprendere meglio il meccanismo del calcolo, seguiamo dal principio alla fine il ciclo di elaborazione di un semplice programma. Il monitor, residente nella memoria associata a MK,

viene avvisato dell'esistenza di un job e chiama attraverso CH 5 l'assemblatore, conservato ordinariamente su memoria di massa, per cedergli il controllo. Le linee di codice sorgente vengono elaborate sulla base di tavole d'assemblaggio organizzate in LP e contenute nelle sezioni dei numeri interi e dei caratteri. Le istruzioni macchina vengono costruite come numeri interi da IP e passate ad una ad una alla memoria di massa attraverso CH 5. Da qui le carica il "loader" a traduzione completata, per dare inizio all'esecuzione vera e propria.

Nel caso più complesso, tale esecuzione prevede un'operazione vettoriale, ad esempio la moltiplicazione di un numero in virgola mobile per un vettore di elementi reali. Si tratta di un'operazione fra dati di tipo misto: nell'istruzione macchina sono contenuti i "tag" "real" e "real array". Il decodificatore contenuto in MK invia i segnali di abilitazione sia a FPP sia a VP, mentre non è necessario l'intervento di CVP. Entrambi iniziano il ciclo di macchina, ma un test nel firmware di FPP, analogo al loop (1) nel ciclo di VP, lo mette in stato di attesa. Simultaneamente, VP seleziona nella sua memoria dati il vettore cercato, inizializza X1 e T1 e calcola l'indirizzo del primo elemento del vettore. Successivamente invia ad FPP microistruzioni di questo tipo:

— caricare in accumulatore l'elemento scalare;
— moltiplicarlo per l'elemento che sta all'indirizzo indicato, fornito da VP;
— scaricarlo allo stesso indirizzo.

La trasmissione di queste direttive è praticamente fattibile poichè è stato presupposto che i processori scalari abbiano lo stesso repertorio locale, limitatamente alle operazioni comuni, cosicchè non è necessario che VP disponga di una microroutine esterna per ciascuna possibile destinazione.

Dopo queste tre operazioni, il ciclo di macchina di FPP viene nuovamente posto in attesa dal test cablato, mentre VP procede a calcolare un nuovo indirizzo, previo test fra X1 e T1. Durante queste procedure, il ciclo di MK viene inibito dal settaggio di VOP, per cui VP è l'unico processore non in stato di attesa o disabilitato.

5. CONSIDERAZIONI SULL'ARCHITETTURA STELLARE

L'architettura stellare aggiunge nuovi vantaggi a quelli caratteristici di ciascun elaboratore che faccia uso della rappresentazione "tagged" dei dati.

Anzitutto, la modularità del disegno. Le risorse di calcolo sono accentrate attorno ai nodi logici individuati dalle funzioni elaborative dei vari tipi di dati e costituiscono le sezioni specializzate. Ciò consente di fornire a ciascun processore le caratteristiche più idonee a svolgere i propri compiti, come lunghezza di parola e numero di registri opportuni, senza cercare di pervenire a standardizzazioni forzate. Si può ad esempio impiegare una memoria a parole per le sezioni reale ed intera ed una memoria a byte come supporto per le stringhe BCD. In tal modo, ciascun elaboratore locale può utilizzare le proprie risorse in modo ottimale, perchè queste gli sono state attribuite a posteriori rispetto ai compiti, ed il tasso di impiego dell'hardware è ottimizzato.

Altri due fattori, sempre derivanti dalla modularità, concorrono a ridurre il costo del sistema. Come già sottolineato, le varie sezioni si prestano a venire implementate con microprocessori, e ciò anzi semplifica il cablaggio delle varie funzioni. In pratica, ogni sezione può venir realizzata con microprocessori esistenti in commercio, organizzati in batterie "bit-sliced" per ottenere la lunghezza di parola desiderata. A carico del costruttore rimane in tal modo solo la realizzazione delle interfacce e dei bus di comunicazione, compiti relativamente semplici. Occorre sottolineare che la classica architettura di von Neumann si presta assai meno a simili implementazioni rapide, in virtù della natura casuale dei suoi circuiti logici.

Inoltre, la configurazione di base della macchina si può modificare facilmente a partire dalla struttura indicativa che abbiamo presentato. Nel caso di un utente interessato ad un suo particolare impiego, la struttura si può facilmente riconfigurare per adattarla al settore desiderato, aggiungendo o rimuovendo sezioni locali ed inserendo processori di conversione opportuni: in tal modo l'utente ottiene un prodotto personalizzato ed il capitale impiegato nell'acquisto viene investito interamente nelle risorse di calcolo veramente utili. In pratica, questa elasticità si può perseguire con una gamma di modelli differenziati, dotati della stessa architettura stellare: nè risulta necessario disporre di tanti tipi di CVP ed MK quante sono le possibili combinazioni, poichè è sufficiente utilizzare sempre le configurazioni massimali per le due sezioni. Dei "tag" di volta in volta inutilizzati potrebbe tenere conto una tabella contenuta nel decodificatore.

Oltre a questa espansione orizzontale, è possibile ottenerne una verticale per la gamma di macchine stellari, incrementando prestazioni e costi dei singoli componenti. Evidentemente, modifiche strutturali locali, come l'inserimento di un'unità aritmetica a pipeline dedicata per il processore vettoriale, non farebbero che produrre un ulteriore salto qualitativo, ma la disamina della relativa problematica esula dagli scopi della trattazione: ciò che ci preme sottolineare è che l'architettura stellare si presta ad essere impiegata per i minicalcolatori come per gli elaboratori paralleli di grandi dimensioni.

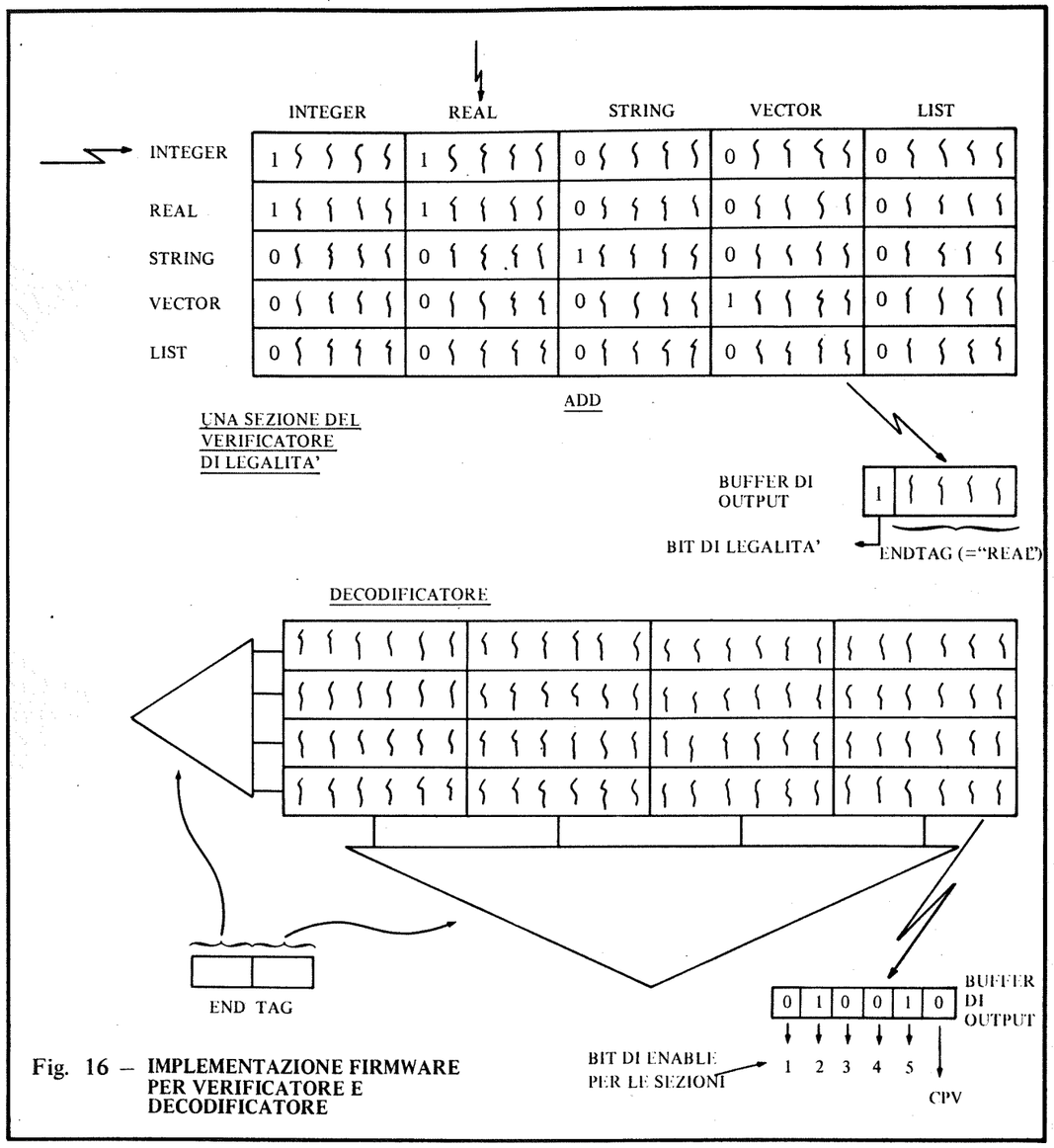


Fig. 16 – IMPLEMENTAZIONE FIRMWARE PER VERIFICATORE E DECODIFICATORE

Sul piano dell'affidabilità hardware, l'architettura stellare consente prestazioni elevate. Ai vari moduli si possono applicare le varie tecniche di ridondanza, e l'utilizzazione di opportuni codici rivelatori non pregiudica la semplicità delle strutture. Può preoccupare la complessità del macrociclo di esecuzione, ma molte funzioni risultano più semplici nella loro realizzazione hardware che nella formulazione software del diagramma di flusso. Il verificatore di legalità delle istruzioni ed il decodificatore, ad esempio, si possono facilmente realizzare con matrici booleane in una memoria a sola lettura, i cui output forniscano direttamente gli impulsi agli altri settori della macchina (fig. 16).
Generalmente, le implementazioni hardware e software di una funzione sono molto diverse. Basti pensare alla complessità programmativa della gestione di una lista: utilizzando una memoria associativa organizzata opportunamente, tale struttura di dati diviene assai maneggevole.

La filosofia costruttiva della macchina stellare e dell'architettura "tagged" in generale sembra essere in contrasto con una tendenza che si è venuta delineando nella realizzazione dei sistemi. La potenza del linguaggio base, a cui fanno riferimento compilatori ed assemblatori, viene generalmente ottenuta creando un assembler "virtuale" non legato alle caratteristiche della macchina, chiamato MOL (macchine oriented language), dotato di tutte le proprietà da noi auspicate per il repertorio. Il divario tra MOL e linguaggio macchina è colmato da opportuni supporti software, che svolgono pure compiti diagnostici. In una scala d'astrazione che includa linguaggi di ogni livello, il MOL si colloca dalla parte della minore generalità, e lo sviluppo in atto tende a spostarlo sempre più verso l'«alto», demandando ad una gerarchia di interfacce software il compito di generare i corrispondenti codici eseguibili. Nel nostro caso sembra avvenire l'opposto, poichè è l'hardware stesso ad eseguire i compiti di conversione: il linguaggio base diviene il più basso della scala.
Quest'inversione di tendenze è prevalentemente apparente; in certa misura tuttavia risponde a precise motivazioni. Le interfacce software di traduzione non sono state soppresse, bensì trasformate in firmware: anche il repertorio di una macchina stellare è un assembler "virtuale", in un certo senso programmato a partire da livelli inferiori. La cornice teorica fornita dal raggruppamento delle funzioni secondo il tipo dei dati elaborati consente tuttavia un frazionamento del carico computazionale che non solo ne riduce la complessità, ma demanda una parte della logica alla struttura stessa, cioè all'architettura del sistema. Tale concezione risulta notevolmente modulare, mentre quella tradizionale era monolitica: dato che la complessità del software è più che lineare rispetto alla lunghezza dei programmi, dalla distribuzione delle funzioni trae vantaggio l'affidabilità.

6. CONCLUSIONE

Uno dei più importanti enunciati della teoria degli automi afferma che, in presenza di un nucleo minimo di funzioni cablate, i compiti dell'hardware e quelli del software sono intercambiabili. Il nucleo fondamentale viene individuato teoricamente dalla macchina di Turing universale e praticamente da un microprocessore con un repertorio essenziale. Tale simmetria tra software e hardware sussiste altresì nei confronti del metodo che consente di perseguire l'affidabilità del funzionamento: l'introduzione di ridondanza nel disegno effettivo o nella procedura di progettazione.

Gravi divergenze compaiono tuttavia al livello dei risultati ottenuti sul piano dell'affidabilità e dei costi sostenuti per renderli possibili. Sono stati escogitati numerosi modi per controllare il buon funzionamento dei circuiti, mentre manca ancora un procedimento pratico per verificare la correttezza dei programmi. Ciò è tanto più grave in quanto le spese di programmazione assorbono una frazione sempre maggiore degli investimenti di messa a punto dei sistemi: si è assistito anzi ad un brusco rovesciamento di tendenze, poichè i costi software, un tempo assai contenuti, hanno oggi sopravvanzato di molto quelli hardware. Lo sviluppo del software richiese il 25% del budget USAF per l'EDP nel 1960, ma raggiunse l'80% nel 1973 (BoeA73).

Il tradizionale equilibrio di costi a favore del software ha prodotto la tendenza a realizzare il maggior numero possibile di funzioni attraverso la programmazione, avvalendosi di linguaggi ad alto livello e di pesanti interfacce di conversione per ricondurli a codici eseguibili. Oggi, questa tradizione va ridimensionata, ed i suoi punti di forza, come la potenza dei linguaggi "problem oriented", vanno armonizzati con l'esigenza di demandare all'hardware una frazione sempre più consistente di carico computazionale, per migliorare costi ed affidabilità dei sistemi.

Certo, non basta cablare meccanicamente una routine dopo l'altra, congelando nelle circuiterie intere biblioteche software: in tal modo non si risolverebbe alcun problema, ma se ne susciterebbero molti nuovi. Ciò che ci occorre è una cornice teorica che guidi il trasferimento di funzioni, riconfigurandole in modo coerente al supporto tecnologico da impiegare. In questo senso va intesa l'utilizzazione della classificazione dei dati in tipi e l'analisi dei differenti aspetti realizzativi della rappresentazione "tagged": al di là dei propositi diagnostici immediati si cercava una filosofia costruttiva che rendesse elastica ed "intelligente" l'architettura stessa della macchina. La disponibilità di una tecnologia a basso costo e ad elevate modularità come quella dei microprocessori ha dettato poi le ulteriori scelte di progettazione, portando con sè cospicui vantaggi. I problemi di integrazione dinamica dei vari componenti hanno prodotto a loro volta una proposta costruttiva, i linguaggi locali, che può arricchire il firmware di nuove applicazioni e nuovi sviluppi.

Il sistema che scaturisce da queste interazioni rappresenta un primo passo nella direzione auspicata. Molte funzioni ricorrenti in compilatori e sistemi operativi, come scansioni e ricerche tabellari, divengono di realizzazione immediata grazie alle primitive vettoriali disponibili, ed in generale tutta la struttura del software si alleggerisce, poichè l'architettura stessa viene incontro ai più significativi costrutti di alto livello.
E' questa la cosiddetta "language directed computer design": al di là dei suoi limiti contingenti, la nostra proposta va principalmente intesa come un contributo in questa direzione, che ci appare la più adatta a guidare l'evoluzione dei sistemi nei prossimi anni.

7. RIFERIMENTI

[BelC71] - C.G. Bell, A.Newell - Computer Structures: Readings and Examples, McGraw-Hill 71

[McKW67] - W.M. McKeeman - Language Directed Comp. Design - Fall Joint Comp. Conf., 1967

[CheG71] - G.D. Chesley, W.R. Smith - The h/w - Implemented High Level Machine Language for SYMBOL - Spring Joint Comp. Conf., 71 - p. 563

[FenE73] - E.A. Fenstel - On the Advantages of Tagged Architecture - IEEE Trans. Comp. - July 73 - p. 644

[HanY76] - Yih-Wu Han - A Systematic Study of Computer System Reliability-Berkeley Comp. Sc. PhD thesis, 76

[BoeA73] - B. Boehm et al. - Characteristics of Software Quality - TRW-SS-73-09, Dec. 7

[GanJ77] - J.D. Gannon - An Experimental Evaluation of Data Type Conventions - CACM, Aug 77 p 584

[BakF73] - F.T. Baker, H.D. Mills - Chief Programmer Teams - Datamation, Dec 73, p. 58

[FenE72] - E.A. Fenstel - The Rice Research Comp - A Tagged Architecture - Spring Joint Comp. Conf., p. 369

[IliJ68] - J.K. Iliffe - Basic Machine Principles - American Elsevier, New York, 68

[SarL77] - L.G. Sartori - La struttura di elaboratori interamente autodiagnostici - Introduzione alla diagnostica - seminario HISI - Istituto Cibernetica Univ. Milano, Apr. 77 - p. 336

[HanF68] - F.M. Haney - Using Comp. to Design Comp. Justo Sets - Carnegie - Mellon Univ., Comp. Sc. PhD thesis, 68

STRUMENTI PER LA MICROPROGRAMMAZIONE STRUTTURATA

M. Gualzetti°, M. Motta°, L. Squizzato°

Riassunto

Questo lavoro descrive alcune esperienze di estensione, attraverso macro o precompilatori, di linguaggi di programmazione di livello molto basso, allo scopo di facilitare l'adozione delle usuali tecniche di programmazione strutturata.
I linguaggi a cui ci si riferisce sono l'assembler del microprocessore Intel 8080, e due linguaggi firmware (di un sistema multiprocessor e del livello 62 Honeywell).

Abstract

In this paper some implementations are described, in which very low-level programming languages have been extended, with the use of macros or precompilers, in order to allow the adoption of the well known structured programming techniques.
The languages involved in the described projects are the assembler of the Intel 8080 microprocessor, and two firmware languages (of a multiprocessor system and of Honeywell Level 62).

1. INTRODUZIONE

Da alcuni anni i microprocessori si sono imposti come gli elementi più idonei a svolgere innumerevoli tipi di funzioni (come la gestione di processi industriali) e sono giunti anche a proporsi come elementi costitutivi di micro e minicalcolatori "general purpose" così come di terminali, intelligenti o no.

La loro duttilità ne ha favorito l'utilizzo anche da parte di utenti non necessariamente specializzati in linguaggi e programmazione firmware (F/W).
In particolare, il F/W viene oggi fortemente utilizzato non solo in applicazioni "dedicate" orientate alla gestione di specifici processi, ma anche nella costruzione di microprogrammi che, in sistemi di calcolo, svolgono funzioni, ad esempio tipiche di sistemi operativi, un tempo realizzate esclusivamente dal software (si pensi ad esempio a firmwarizzazione di routines di data management).

(°) H.I.S.I., Pregnana Milanese e Univ. di Milano, Istituto di Cibernetica.

Il crescente utilizzo di linguaggi di basso livello che ne consegue e le applicazioni a cui tali linguaggi sono rivolti suscitano esigenze di leggibilità, facilità al testing, manutenibilità.
Una risposta a tali esigenze viene dall'estensione, anche ai linguaggi di basso livello, delle tecniche della programmazione strutturata, già ampiamente diffuse per i linguaggi evoluti.
Il presente lavoro descrive specifiche esperienze che hanno permesso di raggiungere tali obiettivi, mediante l'estensione di linguaggi di basso livello con macro e con precompilatori.
Il lavoro si riferisce a linguaggi molto vicini alla macchina: l'Assembler INTEL 8080 e i linguaggi firmware di P6 (la macchina che realizza, via F/W, il Livello 62 Honeywell) e di un sistema multiprocessor.

L'orientamento seguito nella scelta dell'estensione di tali linguaggi è quello suggerito dalla programmazione strutturata: lo sviluppo che essa ha avuto in questi ultimi anni e le esperienze acquisite danno utili indicazioni sulla opportunità di avere a disposizione strutture di controllo fondamentali (selezione, iterazione etc.).
Nella definizione dell'estensione dei linguaggi, si è fatta particolare attenzione al fatto che l'innalzamento del livello di linguaggio conseguente alla introduzione delle strutture non andasse a discapito della visibilità della macchina, soprattutto in relazione alle parti hardware normalmente operabili da F/W; ci si è quindi posto l'obiettivo che le risorse della macchina, usate da una struttura, fossero visibili nella chiamata delle macrofasi (°).

2. LE STRUTTURE DI CONTROLLO

Le strutture implementate per i tre linguaggi sono quelle già diffusamente descritte in letteratura, e cioè:

(°) Nel seguito dell'articolo con il termine "macrofase" intenderemo genericamente le frasi introdotte come estensioni del linguaggio ospite, sia quando esse sono realizzate per mezzo di un precompilatore, sia quando sono realizzate per mezzo di macroprototipi.

- **selezione:**

  IF ... ELSEIF ... ELSE ...ENDIF

- **iterazione:**

  DO ... REPEATUNTIL
  DO ... REPEATWHILE
  DO UNTIL ... REPEAT
  DO WHILE ... REPEAT
  DO ... EXITIF ... EXIT ... REPEAT
  DO ... VARYING ... REPEAT

- **selezione diretta da eventi:**

  EXECUTE ... EVENT ... THENCASE ...
  ... WHEN ... ENDEXEC

- **salto a subroutine:**

  PERFORM ... SUBROUTINE ...
  ... RETURN.

Le caratteristiche dei linguaggi ospiti hanno portato alla realizzazione del set più conveniente di strutture in termini di occupazione di memoria e di tempi di esecuzione.

Nell'ambito di linguaggi di basso livello, infatti, il problema della occupazione di memoria e, soprattutto, dei tempi di esecuzione, può assumere dimensioni critiche. Ciò è vero soprattutto in quelle applicazioni, tipiche del F/W, per eui la velocità della macchina visibile al software è fondamentalmente funzione della velocità del microprogramma che le implementa (es.: interior decor, cioè il F/W per l'implementazione di istruzioni assembler di una macchina).
Queste considerazioni hanno portato, caso per caso, alla scelta delle strutture realizzabili in modo più vantaggioso (es.: DO UNTIL invece di DO WHILE).
Si noti che, anche in condizioni particolarmente critiche (tempo di esecuzione e memoria), l'impiego della programmazione strutturata resta valido, tenuto conto della possibilità di effettuare:

- ottimizzazioni locali, eventualmente realizzate automaticamente (eliminazione di istruzioni di salto che trasferiscono il controllo ad altre istruzioni di salto, eliminazione di istruzioni di "no operation";
- revisione critica degli algoritmi con l'introduzione di istruzioni che "rompono" la struttura a vantaggio delle prestazioni del programma.

3. CARATTERISTICHE DEI LINGUAGGI

P6

Il linguaggio di questa macchina permette l'utilizzo delle strutture di controllo in modo perfettamente trasparente all'utente. Questo viene realizzato mediante istruzioni che non alterano le risorse H/W della macchina ma che non permettono di controllare codici di estensione superiori a 64 bytes.
Per superare questa limitazione sono state implementate strutture che permettono di controllare porzioni di codici di 2K bytes,ma che alterano un registro hardware.
Quindi al programmatore viene offerta la possibilità di scegliere tra strutture "corte" (es.: IFS, ove la S sta per Short) e lunghe (es.: IF) a seconda delle sue esigenze.

INTEL 8080

Per questo linguaggio non esiste alcuna delle limitazioni viste per P6, in quanto il range di salto è praticamente illimitato (± 64K bytes), rispetto alle dimensioni di memoria.
L'unico problema si è presentato nell'implementazione della struttura DO ... VARYING, in cui, per lasciare visibilità completa delle risorse H/W, si è ricorso ad innumerevoli salvataggi con conseguente appesantimento del codice e dei tempi di esecuzione.

Sistema multiprocessor

Le caratteristiche di questo linguaggio hanno permesso di sfruttare appieno le esperienze precedenti (P6, INTEL 8080 e STNAL [1]) e le possibilità offerte dalle strutture.
Innanzitutto le istruzioni di salto permettono range di 16K bytes, le risorse H/W del sistema poi permettono l'utilizzo di alcune di esse per poter eseguire operazioni normalmente previste dal microprogrammatore; questo permette un innalzamento considerevole del linguaggio a scapito di una accettabile riduzione di visibilità della macchina H/W.

4. I PRECOMPILATORI

P6

Le strutture di controllo sono messe a disposizione mediante un precompilatore, che riceve in ingresso un sorgente contenente, oltre alle microistruzioni, anche le macrofrasi e fornisce in uscita un sorgente costituito di sole microistruzioni, in formato scheda (su scheda o su nastro magnetico).
Questo viene poi fornito al compilatore F/W (o, meglio, a quell'insieme di programmi che ne svolgono le funzioni) (fig. 1).

Il set di strutture implementate è il seguente:

IF ... ELSEIF ... ELSE ... ENDIF
DO ... WHILE ... REPEAT ...
DO ... EXITIF ... REPEAT
EXECUTE ... EVENT ... THENCASE ...
... WHEN ... ENDEXEC
PERFORM ... SUBROUTINE ... RETURN

Il precompilatore è scritto in NAL (Assembler del sistema L62 Honeywell).

STRUTTURA DEI PRECOMPILATORI

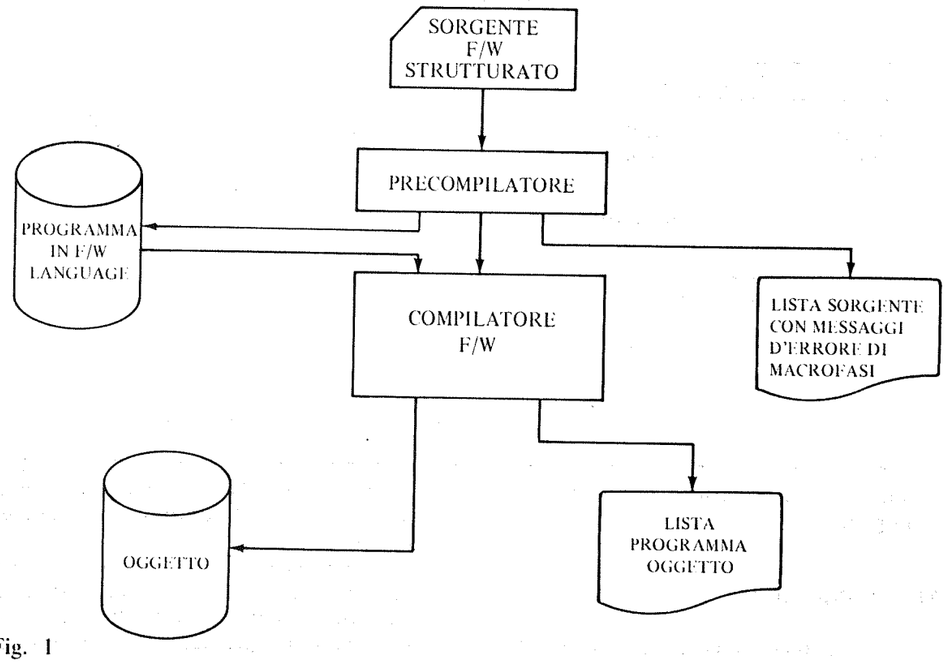


Fig. 1

In fig. 2 è riportata la lista della traduzione della struttura IF fornita dal precompilatore.

INTEL 8080

Il set di strutture implementate è il seguente:

IF ... ELSEIF ... ELSE ... ENDIF
DOWHILE ... REPEAT
DO ... REPEAT UNTIL
DO ... EXITIF ... REPEAT
EXECUTE ... THENCASE ... ENDEXEC
DOFOR ... ENDDO
(quest'ultima è la struttura di iterazione su variabili indiciate).

Anche nel caso di INTEL 8080, il precompilatore (scritto in Fortran IV) accetta in ingresso il programma sorgente attrezzato con le macrofrasi, e fornisce in uscita:

- il codice (con le macrofrasi tradotte) accettabile come sorgente dall'assemblatore 8080;
- la lista del programma con l'espansione delle

```
*L      $IF     CY      THEN

                JCCY            R0001           MMMMM
                JUR             R0002           MMMMM
R0001           NOP                             MMMMM

                < CODICE UTENTE A >

*L      $ELSEIF         SB              THEN

                JUR             R0003           MMMMM
R0002           JCSB            R0004           MMMMM
                JUR             R0005           MMMMM
R0004           NOP                             MMMMM

                < CODICE UTENTE B >

*L      $ELSEIF         G3              THEN

R0003           JUR             R0006           MMMMM
R0005           JCG3            R0007           MMMMM
                JUR             R0008           MMMMM
R0007           NOP                             MMMMM

                < CODICE UTENTE C >

*L      $ELSE

R0006           JUR             R0009           MMMMM
R0008           NOP                             MMMMM

                < CODICE UTENTE D >

*L      $ENDIF

R0009           NOP                             MMMMM
```

Fig. 2 — Esempio prodotto dal precompilatore per P6

macrofrasi e le segnalazioni di errore (un esempio in fig. 3).

5. I MACROPROTOTIPI

L'esperienza acquisita nell'implementazione delle macrofrasi e dei precompilatori per il F/W di P6 e per l'Assembler 8080 ha portato alla definizione del linguaggio F/W strutturato in ambiente multiprocessor.
Questo linguaggio, in tutto simile funzionalmente ad altri linguaggi F/W, è stato corredato delle strutture di controllo mediante la definizione di macroprototipi (realizzati cioè via macroprocessor).


```
$DO
                IN              OH
                CPI             OFFH
$EXITIF         Z
  $EXECUTE ROUT1,ROUT2,ROUT3,
                ROUT4,ROUT5,ROUT6,
                RAR
  $EVENT        ROUT1 IF C
                RAR
  $EVENT        ROUT2 IF C
                RAR
  $EVENT        ROUT3 IF C
                RAR
  $EVENT        ROUT4 IF C
                RAR
  $EVENT        ROUT5 IF C
                RAR
  $EVENT        ROUT6 IF C
                RAR
    $IF C       THEN
ROUT7:
; CODICE        G
    $ELSE
ROUT8:
; CODICE        H
    $ENDIF
  $THENCASE
  $WHEN ROUT1
; CODICE        A
  $WHEN ROUT2
; CODICE        B
  $WHEN ROUT3
; CODICE        C
  $WHEN ROUT4
; CODICE        D
  $WHEN ROUT5
; CODICE        E
  $WHEN ROUT6
; CODICE        F
  $ENDEXEC
$REPEAT
                HLT
                END
```

Fig. 3 — Esempio accettato dal precompilatore per INTEL 8080

Dal punto di vista dell'utente (per utente in questo caso si intende il progettista F/W) i macroprototipi implementati offrono:

- rilocabilità dei programmi oggetto, in quanto parametrici rispetto all'assolutizzazione dovuta alla compilazione (es. il F/W di sottosistema, che viene caricato solo al momento del suo utilizzo, non dipende dall'indirizzo d'impianto in quanto tutti i suoi collegamenti al F/W residente sono effettuati mediante salti indiretti, e non mediante salti assoluti);
- implementazione top-down dei microprogrammi utilizzando subroutines interne ed esterne al microprogramma (essendo l'ambiente F/W dotato anche di un linker che assicura i collegamenti tra vari programmi e procedure);
- razionale utilizzo delle strutture, in quanto i macroprototipi sono stati definiti in stretta collaborazione coi progettisti F/W (essi infatti hanno portato le esigenze di utenza direttamente in fase di progettazione).

Il linguaggio è stato inoltre arricchito di macro per la definizione di strutture di dati e relativo accesso, con un conseguente ulteriore innalzamento di livello [2].

Il linguaggio F/W così attrezzato viene oggi ampiamente e proficuamente utilizzato dai progettisti F/W della HISI di Pregnana, consentendo aggiunte e modifiche ai microprogrammi in tempi eccezionalmente brevi.
Le caratteristiche del linguaggio F/W e della macchina H/W hanno consentito l'ampliamento delle funzionalità dei macroprototipi rispetto alle esperienze di P6 e INTEL 8080.
Essi, infatti, consentono l'esecuzione di operazioni normalmente predisposte dal programmatore (MOVE fra registri, istruzioni operative, confronti, salti) mediante il richiamo di un solo macroprototipo.
Questa peculiarità è offerta dalla ricchezza dei parametri di chiamata che forniscono le indicazioni sulle operazioni da eseguire.
Si noti che tali operazioni sono, però, completamente trasparenti, normalmente, in quanto non comportano l'alterazione di alcun registro hardware nello sviluppo della condizione. Solo in casi eccezionali (ad es. quando siano necessarie operazioni di MOVE fra registri) esse utilizzano alcuni registri, che compaiono esplicitamente nella macrofase, dichiarati nei parametri di richiamo dal microprogrammatore stesso.

Le strutture implementate sono:

- IF ... ELSEIF ... ELSE ... ENDIF
- DO ... REPEATWHILE
- DOUNTIL ... REPEAT
- DO ... EXIT(IF) ... REPEAT
- DO ... VARYING ... REPEAT
- EXECUTE ... EVENT(IF) ... THENCASE ... WHEN ... ENDEXEC
- PERFORM ... ROUTINE ... RETURN.

Una caratteristica delle strutture è che esse possono essere utilizzate sia nell'implementazione di routines F/W allocate nella memoria di controllo della macchina, sia in quelle adibite alla gestione di sottosistemi che, normalmente, sono rilocabili.
Questo è reso possibile soprattutto dalla struttura

PERFORM ... ROUTINE ... RETURN

che può essere tradotta, di caso in caso, come un salto indiretto o come un salto relativo, realizzando così i collegamenti tra i F/W rilocabili (di sottosistema) e no.
Un'altra peculiarità è che la macro EXIT(IF) può essere inserita in qualsiasi altra struttura (posto che sia all'interno di un gruppo di DO).
Questo, seppure "rompa" la struttura, offre ulteriori possibilità al linguaggio strutturato che lo rendono competitivo rispetto alle tradizionali tecniche di microprogrammazione.

Ulteriori problemi rimangono però aperti per quanto riguarda eventuali ottimizzazioni, ad esempio nella nidificazione di strutture.
In questo caso, potrebbero presentarsi situazioni in cui compaiono istruzioni di salto ad altre istruzioni di salto, ad esempio:

```
        JUMP            PIPPO
        .....
        .....
        .....
PIPPO   JUMP            PLUTO
```

e questo implica un aumento dei tempi di esecuzione.

E', comunque, molto semplice l'implementazione di un algoritmo per la soluzione automatica di tale problema.

7. RIFERIMENTI

[1] C. Poggiali, S. Farnedi, M. Maiocchi, R. Polillo: "STNAL, un insieme di macro per la programmazione strutturata in NAL", NOTE DI SOFTWARE n° 1, gennaio 1977.

[2] C. Monducci, S. Dalla: "Strumenti di microprogrammazione nello sviluppo di grandi sistemi". NOTE DI SOFTWARE n. 6/7, aprile-settembre 1978.

IL SEGNALIBRO

In questa rubrica vengono segnalati libri, articoli o studi di recente pubblicazione che, a giudizio della redazione o dei lettori di NOTE DI SOFTWARE, rivestono un particolare interesse nell'ambito dei temi a cui questa rivista è dedicata.
Le citazioni fra virgolette, se non ne è indicata la fonte, sono tratte dai lavori stessi.

Metodologie - un'antologia

● Gries, D. (editor), PROGRAMMING METHODOLOGY - A Collection of Articles by Members of IFIP WG2.3, Texts and Monographs in Computer Science, Springer-Verlag, 1978, XIV +437

"This volume is being published for two reasons. The first is to present a collection of previously published articles on the subject of programming methodology that have helped define the field and give it direction. [...]
The second [...] is to make public the nature and work on programming methodology of IFIP Working Group 2.3. (IFIP stands for 'International Federation for Information Processing'). WG2.3 is one of many IFIP Working Groups that have been established to provide international forums for discussion of ideas in various areas. [...]"
Il volume raccoglie molti dei più rilevanti contributi nell'area delle metodologie di programmazione. L'indice è il seguente. Part I: Viewpoints on Programming: 1. THE HUMBLE PROGRAMMER, di E. W. Dijkstra; 2. SOFTWARE ENGINEERING, di J. N. Buxton; 3.SOFTWARE ENGINEERING - SOME PRINCIPLES AND PROBLEMS, di W. M. Turski; 4. THE ENGINEERING OF SOFTWARE: A STARTLING CONTRADICTION, di C.A.R. Hoare; 5. PROGRAMS, CITIES, STUDENTS -LIMITS TO GROWTH?, di M. M. Lehman; 6. ON STRUCTURED PROGRAMMING, di D. Gries; Part II: The Concern for Program Correctness: 7. CORRECTNESS CONCERNS AND, AMONG OTHER THINGS, WHY THEY ARE RESENTED, di E. W. Dijkstra; 8. AN AXIOMATIC BASIS FOR COMPUTER PROGRAMMING, di C.A.R. Hoare; 9. PROOF OF A PROGRAM: FIND, di C.A.R. Hoare; 10. AN APPROACH TO CORRECTNESS PROOFS FOR SEMICOROUTINES, di O.-J. Dahl; 11. AN AXIOMATIC PROOF TECHNIQUE FOR PARALLEL PROGRAMS, di S. Owicki e D. Gries; 12. PROGRAMMING WITH TRANSITION DIAGRAMS, di J. C. Reynolds; 13. GUARDED COMMANDS, NON-DETERMINACY, AND FORMAL DERIVATION OF PROGRAMS, di E. W. Dijkstra; 14. A SYSTEM WHICH AUTOMATICALLY IMPROVES PROGRAMS, di J. Darlington e R. M. Burstall; Part III: Harnessing Parallelism; 15. TOWARDS A THEORY OF PARALLEL PROGRAMMING, di C.A.R. Hoare; 16. STRUCTURED MULTI-PROGRAMMING, di P. Brinch Hansen; 17. MONITORS: AN OPERATING SYSTEM STRUCTURING CONCEPT, di C.A.R. Hoare; 18. THE PROGRAMMING LANGUAGE CONCURRENT PASCAL, di P. Brinch Hansen; Part IV: Data Types: 19. PROOF OF CORRECTNESS OF DATA REPRESENTATIONS, di C.A.R. Hoare; 20. THE ALGEBRAIC SPECIFICATION OF ABSTRACT DATA TYPES, di J. V. Guttag e J. J. Horning; 21. USER -DEFINED TYPES AND PROCEDURAL DATA STRUCTURES AS COMPLEMENTARY APPROACHES TO DATA ABSTRACTION, di J. C. Reynolds; Part V: Software Development: 22. PROGRAM DEVELOPMENT BY STEPWISE REFINEMENT, di N. Wirth; 23. ON A "BUZZWORD": HIERARCHICAL STRUCTURE, di D. L. Parnas; 24. ON THE DESIGN AND DEVELOPMENT OF PROGRAM FAMILIES, di D. L. Parnas; 25. SYSTEM STRUCTURE FOR SOFTWARE FAULT TOLERANCE, di B. Randell; 26. STRUCTURED ANALYSIS (SA): A LANGUAGE FOR COMMUNICATING IDEAS, di D. T. Ross.

Multiprogrammazione

● Brinch Hansen, P., REPRODUCIBLE TESTING OF MONITORS, Software - Practice and Experience, vol. 8,n. 6 (novembre-dicembre 1978), 721-729

"This paper describes a systematic method for testing monitor modules which control process interactions in concurrent programs. A monitor is

tested by executing a concurrent program in which the processes are synchronized by a clock to make the sequence of interactions reproducible. The method separates the construction and implementation of test cases and makes the analysis of a concurrent experiment similar to the analysis of a sequential program. The implementation of a test program is almost mechanical. The method, which is illustrated by an example, has been successfully used to test a multicomputer network program written in Concurrent Pascal''.

● Thorelli, L.-E., A MONITOR FOR SMALL COMPUTERS, Software - Practice and Experience, vol. 8, n. 4 (luglio-agosto 1978), 439-450

"A monitor suitable for small computers is described. It implements processes and synchronization primitives. A time queue mechanism with the possibility of time-limited waiting for signals, is also provided. Processes have fixed priorities determining allocation of processor time. The monitor allows dynamic creation and deletion of processes. Each process has a local data area and access to the registers of the processor and a global stack on a temporary basis."

Valutazione delle prestazioni

● Il numero di settembre 1978 di Computing Surveys dell'ACM (vol. 10, n. 3) è interamente dedicato a problemi di 'performance evaluation', e, precisamente, ai 'queueing network models'.

"This issue contains three long tutorial/survey papers, three short application notes, and a short note giving a perspective to the issue. The long papers introduce queueing network models, measurement procedures, and approximate solution techniques. The application notes illustrate how the models can be, and have been, used successfully. The concluding note assesses the state of the art."

I lavori presentati sono: THE OPERATIONAL ANALYSIS OF QUEUEING NETWORK MODELS, di P. J. Denning e J. P. Buzen; A MEASUREMENT PROCEDURE FOR QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS, di C. A. Rose; APPROXIMATE METHODS FOR ANALYZING QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS, di K. M. Chandy e C. M. Sauer; A QUEUEING NETWORK MODEL OF MVS, di J. P. Buzen; THE VM/370 PERFORMANCE PREDICTOR, di Y. Bard; QUEUEING NETWORK MODELING OF COMPUTER COMMUNICATION NETWORKS, di J. W. Wong; QUEUEING NETWORKS: A CRITIQUE OF THE STATE OF THE ART AND DIRECTIONS FOR THE FUTURE, di R. R. Muntz.

Correttezza dei programmi

● London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G., Popek, G. J., PROOF RULES FOR THE PROGRAMMING LANGUAGE EUCLID, Acta Informatica, vol. 10 fasc. 1, 1978, 1-26

"In the spirit of the previous axiomatization of the programming language Pascal, this paper describes Hoare-style proof rules for Euclid, a programming language intended for the expression of system programs which are to be verified. All constructs of Euclid are covered except for storage allocation and machine dependencies."

Computer Graphics

● Il numero di dicembre 1978 di Computing Surveys dell'ACM (vol. 10, n. 4) è interamente dedicato al "proposed graphics standard Core System, which has been developed by the Graphics Standard Planning Committee (GSPC) of the ACM Special Interest Group on Graphics (SIGGRAPH). The papers of this issue trace the history of the proposal, review its functions and use in interactive graphics programming, discuss the major design issues and their resolution in the proposal, and explain how three-dimensional objects may be projected onto a flat viewing surface.
The Core System as described here is a proposal for a standard. The purpose of this issue is to describe the technical aspects of the proposal so they can be discussed knowledgeably. Even if the proposal never becomes a standard, the papers of this issue can be of considerable help to the designers and users of interactive graphics systems."

I lavori raccolti nel fascicolo sono: RECENT EFFORTS TOWARD GRAPHICS STANDARDIZATION, di W. M. Newman e A. Van Dam; A FUNCTIONAL OVERVIEW OF THE CORE SYSTEM WITH GLOSSARY, di J. C. Michener e A. Van Dam; GRAPHICS PROGRAMMING USING THE CORE SYSTEM, di R.D. Bergeron, P. R. Bono e J. D. Foley; SOME MAJOR ISSUES IN THE DESIGN OF THE CORE GRAPHICS SYSTEM, di J. C. Michener e J. D. Foley; PLANAR GEOMETRIC PROJECTIONS AND VIEWING TRANSFORMATIONS, di I. Carlbom e J. Paciorek.

● Il numero di novembre 1978 di COMPUTER (vol. 11, n. 11) è anch'esso dedicato alla 'computer graphics'.

"The new user of interactive graphics systems probably doesn't know much about the tradeoffs and options involved with using interactive input/output equipment. And there are few sources of introductory material on the subject. This issue attempts to provide such a source.
In this spirit, you will find an article by Ohlson that discusses graphics input devices—joysticks, tablets, light pens, etc. Ohlson describes the basic operation of the devices and some of the applications in which they may be effectively employed.
Interactive display devices are covered in the articles by Preiss and Lucido. Preiss discusses the direct view storage tube display and its operating environment; Lucido reviews directed beam cathode ray tube display technology.
Rounding out the issue, Machover presents an informal history of computer graphics based on his personal experiences with the technology from the days of Whirlwind to the present. Finally, Capowski discusses an effective computer graphics application—an inexpensive, easily programmable display system for medical users." (A. P. Lucido, Guest Editor Introduction)

Architettura degli Elaboratori

● Myers, G. J., ADVANCES IN COMPUTER ARCHITECTURE, John Wiley & Sons, 1978, pag. xv + 314

"[...] The similarity of the architecture of today's systems to that of earlier systems can cause us to become complacent about the subject; we look around us and see tremendous software and hardware advances, but we also see that the architectures of current systems are virtually the same as those of earlier systems. Thus we might be inclined to assume that someone in the 1940s and 1950s invented all there is to be invented in the area of computer architecture. As a result, the architecture of future systems remains the same. This attitude motivated me to write this book: to destroy this complacency by showing that there are serious problems in current computer architectures, and to discuss advanced architectural concepts that will solve these problems. [...]"

Indice: PART I - THE NEED FOR ARCHITECTURAL ADVANCES: 1. A Definition of Computer Architecture; 2. A Critique of the Conventional Von Neumann Architecture; 3. A Classification of Computer Architectures; 4. Requisites for Improved Architectures; PART II - A LANGUAGE-DIRECTED ARCHITECTURE: 5. The Student-PL Machine SPLM; 6. Program Compilation and Execution on SPLM; 7. SPLM Instruction Set; PART III - A HIGH-LEVEL-LANGUAGE ARCHITECTURE: 8. System Architecture of the SYMBOL System; 9. Computer Architecture of the SYMBOL System; 10. SYMBOL Processor and Configuration Architecture; PART IV - A MULTIPLE-LANGUAGE-DIRECTED ARCHITECTURE: 11. The Burroughs B1700 System; 12. Burroughs B1700 COBOL/RPG Architecture; PART V - A SOFTWARE-RELIABILITY-DIRECTED ARCHITECTURE: 13. The SWARD Machine; 14. Program Compilation and Execution on SWARD; 15. SWARD Instruction Specifications; PART VI -RELATED TOPICS IN COMPUTER ARCHITECTURE: 16. Input/Output Architecture; 17. Architecture Optimization and Tuning; 18. The Art of Computer Architecture.

TRW Series of Software Technology

● Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. J., Merrit, M. J., CHARACTERISTICS OF SOFTWARE QUALITY, TRW Series of Software Technology, 1, North-Holland Publishing Co., 1978, pagg. xxxix + 165

"The study reported in this book establishes a conceptual framework and some key initial results

in the analysis of the characteristics of software quality. Its main results and conclusions are:

- Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs.
- The current software state-of-the-art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software.
- A definitive hierarchy of well-defined, well-differentiated characteristics of software quality is developed. [...]
- A large number of software quality evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation.
- Particular software life-cycle activities have been identified which have significant leverage on software quality.

[...] The bulk of the work reported in this book was performed in a study by TRW for the National Bureau of Standards in 1973. [...]"

● Thayer, T. A., Lipow, M., Nelson, E. C., SOFTWARE RELIABILITY, TRW Series of Software Technology, 2, North-Holland Publishing Co., 1978, pagg. xvi +311

"This document is the final technical report for the Software Reliability Study, performed by TRW for the Rome Air Development Center. It presents results of a study of data, principally error data, collected from four software development projects. These data were analyzed to determine what might be learned about various types of errors in the software; the effectiveness of the development and test strategies in preventing and detecting errors, respectively; and the reliability of the software itself.
This report also provides guidelines for data collection and analysis on other projects: data that are generally available, how project data were collected in this study, and some observed realities concerning the data collection and analysis processes.
Finally, the most recent work on TRW's Mathematical Theory of Software Reliability (MTSR), the Nelson model, is presented. This is complemented by a survey of software reliability models currently available in the software community."

Inchieste

● Couger, J. D., Zawacki, R. A., WHAT MOTIVATES DP PROFESSIONALS?, DATAMATION, vol. 24, n. 9 (settembre 1978), 116-123

"We have just concluded the first part of a survey on what motivates dp employees. The survey revealed that systems professionals have a startlingly low proclivity to social interaction. [...]
It follows that programmers and analysts are not necessarily seeking the kind of interaction that programming team concepts are imposing. [...]
Dp professionals are different from others in a variety of ways, the research shows, including having a higher need for personal growth than other professionals do."

● Fitz-enz, J., WHO IS THE DP PROFESSIONAL?, DATAMATION, vol. 24, n. 9 (settembre 1978), 125-128

"[...] We conducted research in a dozen companies in the western United States during the latter part of 1977 to attempt to find out something about the dp pro's motivation to work and his desires for communicating with the organization that employs him. Data was gathered using a survey questionnaire designed and pretested for the specific project. Some 1,500 subjects in several industries, occupations, and job levels responded to the questionnaire. [...]"

Articoli vari

● Schechter, D., THE SKELETON PROGRAM METHODOLOGY, Datamation, vol. 24, n. 11 (novembre 1978), 147-150

"[...] Where a source program library facility exists, the skeleton program is cataloged in that library. The skeleton is reproduced as many times as there are source modules to be coded. The essential task of the programmer, then, is to customize his particular version of the skeleton program until it satisfies the requirements of the program specification. [...]"

● Sneeringer, J., USER-INTERFACE DESIGN FOR TEXT EDITING: A CASE STUDY, Software-Practice and Experience, vol. 8, n. 5 (settembre-ottobre 1978), 543-557

"This paper describes the design of the user interface of a text editor named Occam with the goal of communicating by example the process of user-interface design. An attempt is made to induce principles; where the attempt fails, the raw details are presented. First Occam itself is described, and then aspects of its user interface are used to exemplify (1) power versus ease of learning, (2) the use of prototypes and user feedback, (3) the importance of planning and (4) error detection and handling."