Capitolo 1

MODELLI DI CALCOLO PER IL PARALLELISMO

La pratica quotidiana di produzione di software applicativo, sebbene orientata all'utilizzo dei tradizionali linguaggi di programmazione sequenziali [1] pone il programmatore di fronte a parecchi esempi di sezioni di codice nei quali é naturale immaginare una possibile esecuzione parallela delle istruzioni componenti. Queste forme di parallelismo riscontrabili nel software sono riconducibili a due tipi facilmente distinguibili. La prima forma di parallelismo si verifica quando una o piú operazioni[2] vengono applicate ad insiemi separati di dati. Un esempio tipico é quello della somma elemento per elemento di vettori della stessa dimensione, come nel caso del seguente programma PASCAL

```
for i := 1 to N do
    begin
        F[ i ] := A[ i ] + C[ i ];
        G[ i ] := A[ i ] - B[ i ]
    end
```

La seconda forma di parallelismo si presenta quando differenti operazioni[3] vengono applicate a piú insiemi di dati, che possono anche essere comuni. Tale situazione é identificabile in molti blocchi di istruzione di assegnazione come nel caso del seguente esempio di codice PASCAL

```
with A do
    with X do
        begin
            X1 := A1 - A3;
            X2 := A4 * A5;
            X3 := A1 * A4 + A2;
            X4 := A1 + A3
        end
```

dove le variabili A e X hanno l'ovvia struttura a *record* che si evince dal codice.

[1] e, tra questi, COBOL, FORTRAN, BASIC, PASCAL per citare i piú comuni

[2] o, insiemi di operazioni

[3] o, insiemi di operazioni

Queste forme di parallelismo sono note da molto tempo e la loro importanza nell'influenzare architetture per il parallelismo é ampiamente riconosciuta. Seguendo Flynn chiameremo *SIMD*[4] la prima forma di parallelismo, mentre indicheremo con *MIMD*[5] la seconda forma di parallelismo. Questa classificazione appare essere troppo semplicistica sebbene non sia emersa nessun'altra tassonomia alternativa largamente accettata. La difficoltá dipende dal tipo di *granularitá* a cui ci si pone rispetto a quella ovvia espressa negli esempi precedenti cosicché, considerando i livelli piú fini di granularitá si arriva a definire un parallelismo al *livello dell'hardware*.

Ad esempio, in un *pipeline* di istruzioni, una piccola parte dell'esecuzione di istruzioni successive viene processata concorrentemente per sovrapposizione parziale, indipendentemente da ogni parallelismo del programma al livello delle istruzioni [6] o a livelli superiori. In assenza di una tassonomia dei sistemi paralleli, che sia indipendente dal livello di granularitá, puó essere utile distinguire fra parallelismo **regolare** e parallelismo **irregolare**, eventualmente confrontando fra di loro i vari sistemi possibilmente in riferimento a quelli piú convenzionali.

Per quanto riguarda il parallelismo regolare si puó dare la seguente definizione.

Si dice che si é in presenza di **parallelismo regolare** *ogniqualvolta lo stessa sequenza di computazione puó essere ripetutamente eseguita su dati che sono solitamente disgiunti.*

Nel caso di dati connessi le forme parallelismo regolare si ottengono per mezzo di *pipeline*, come nell'esempio appena citato comprendente un pipeline di istruzioni. Nel caso di dati separati, come la prima forma di parallelismo software introdotta (SIMD), possono essere impiegate architetture hardware vettoriale a passo bloccato, come nei calcolatori DAP e ILLIAC IV. In altri casi, le azioni da eseguire concorrentemente sono altamente regolari cosicché le prestazioni del sistema dipendono fortemente dal fatto che i programmi coinvolti forniscano sufficiente lavoro rispetto alla quantitá della richiesta forma di regolaritá. Molte architetture parallele esplorano l'una o l'altra forma di parallelismo regolare sebbene sia molto difficile organizzare i programmi in modo da fornire con continuitá il desiderato parallelismo.

Per il cosiddetto parallelismo irregolare, invece, é possibile dare la seguente definizione.

Si dice che si é in presenza di **parallelismo irregolare** *tutte le volte che computazioni distinte possono essere eseguite potenzialmente in modo parallelo, anche su dati comuni.*

Questa forma di parallelismo corrisponde alla forma *MIMD* di parallelismo riscontrato nel software e hanno come supporto architetturale hardware un vettore di elementi computazionali paralleli indipendenti. Tra questi si possono citare il CDC 6600[7] o C.mmp e il multiprocessore Cm[8]. Quando i dati coinvolti dalle istruzioni sono condivisi, é necessario introdurre un complesso di meccanismi di *interlocking* al fine di prevenire accessi indesiderati ai dati[9]. Chiaramente i meccanismi hardware che esplorano il prallelismo irregolare sono in grado di trattare anche quello regolare; ma non é vero il viceversa.

Esistono alcuni sistemi che sono stati costruiti per trattare forme di parallelismo irregolare su larga scala fino ad arrivare alla realizzazione di molte interessanti architetture sperimentali nelle quali il parallelismo viene trattato al *livello dei processi*. Molte di queste architetture sono derivate da linguaggi di programmazione come **Concurrent Pascal**, **Modula**, **Distributed Process** e **Communicating Sequential Process**.

[4] single-instruction-stream, multiple-data-stream
[5] multiple-instruction-stream, multiple-data-stream
[6] é questo il caso dell'attuale tecnologia *RISC*
[7] su piccola scala
[8] su larga scala
[9] lettura di dati prima della loro definizione, scrittura prima del completamento di una precedente lettura o scrittura

1.1 GRAFI DI DIPENDENZA DEI DATI

Da quanto si é detto, indipendentemente dal fatto che il parallelismo sia regolare o irregolare, il punto chiave consiste nello sviluppo di un sistema che esplori il parallelismo mediante una notazione efficace che sia in grado di esprimere il potenziale parallelismo presente nei programmi. In questa sezione verrá presentata una notazione, per il parallelismo irregolare al *livello delle istruzioni*, con lo scopo di esaminare piú da vicino la natura dell'inerente parallelismo che puó essere presente in porzioni limitate di codice PASCAL.

Si consideri, allora, il gruppo di istruzioni PASCAL

```
begin
   T1 := A + B;
   T2 := B - C;
   E := T1 * T2
end
```

per la valutazione della seguente espressione

$$e \quad \leftarrow \quad (a+b) \times (b-c)$$

avendo avuto cura di indicare esplicitamente le operazioni aritmetiche coinvolte, nella loro forma canonica. Per individuare il potenziale parallelismo nel software é necessario abbandonare il tradizionale punto di vista dei programmi come lista di istruzioni che manipolano dati in locazioni fisse di memoria secondo una sequenza predefinita. E' necessario, invece, concentrarsi sul ruolo che le locazioni individuali di memoria giocano come *buffer* temporanei contenenti i valori dei dati che vengono passati fra le operazioni componenti il programma. Il meccanismo degli accessi alla memoria, che é determinato dalla sequenza di attivazioni delle istruzioni, é controllato dal programmatore stesso in relazione alle combinazioni di dati con le opportune operazioni dettate dal particolare problema da risolvere.

Un modo alternativo di combinare dati e operazioni puó essere ottenuto costruendo un *grafo di dipendenza dei dati* per il programma in esame. Algoritmi di questo tipo sono di uso comune nella fase di ottimazione nei compilatori convenzionali. Nell'esempio in questione il programma é rappresentato da un certo numero di *archi orientati*, un arco per ogni variabile. La parte *uscente* dell'arco mostra dove la variabile assume valore; la parte *entrante* indica dove la variabile é **consumata**[10]. In alcuni casi particolari si richiede che vi sia piú di un arco per variabile quando questa appare nella parte destra di istruzioni di assegnazione multiple. In situazioni di questo tipo[11] le variabili vengono *ridenominate* sistematicamente in modo da generare una versione del programma senza assegnazioni multiple e, quindi, mantenendo per la variabile lo stesso significato originale. Le variabili che compaiono solamente nella parte destra di istruzioni si dicono anche dati di ingresso.

Per gli scopi della discussione che segue é necessario introdurre una rappresentazione astratta del codice per una ipotetica macchina che esegue programmi nella forma di grafi di dipendenza dei dati. Tale rappresentazione é ottenuta come un insieme di istruzioni caratterizzate da sequenze di **argomenti**[12] delimitati da parantesi

$$(arg_0 \quad arg_1 \quad \dots \quad arg_n)$$

[10] comparendo nella parte destra di una istruzione di assegnazione
[11] dove le variabili hanno assegnato un valore in piú punti del programma
[12] operatori, letterali, operandi, riferimenti

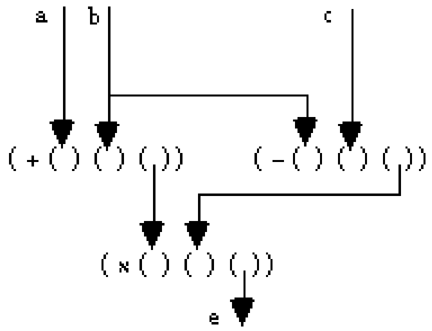

Figure 1.1: esempio di grafo di dipendenza dei dati

dove arg_0 definisce normalmente un operatore. Nell'esempio in esame il grafo di dipendenza dei dati puó rappresentarsi nel modo indicato dalla fig 1.1 la cui forma bidimensionale rafforza il potenziale parallelismo rilevabile dall'analisi del codice.

Piú in generale si puó dare la seguente definizione di *grafo di dipendenza dei dati*.

Si dice **grafo di dipendenza dei dati** *di un programma un* grafo etichettato orientato *i cui nodi rappresentano le istruzioni del programma e i cui archi sono i* **canali** *entro cui i dati vengono trasferiti dalle istruzioni che li producono verso quelle che li consumano. Le etichette degli archi prendono il nome di* **variabili**.

Come é facile rendersi conto le variabili di un grafo di dipendenza dei dati assumono un ruolo attivo in quanto associano al valore della variabile [13] il portatore del valore stesso [14]. La struttura che cosí si ottiene prende il nome di **data token** o, piú semplicemente, **token** dal nome che enti simili hanno nella teoria delle *reti di Petri*.

1.2 ORGANIZZAZIONE DELLA COMPUTAZIONE

Il grafo di dipendenza dei dati, nell'esempio citato nel paragrafo precedente, fa vedere come ogni modello di esecuzione parallela deve, necessariamente, permettere la valutazione contemporanea delle due istruzioni di *somma* e *differenza*. La soluzione migliore per ottenere tale effetto desiderato é rinunciare definitivamente all'uso del *program counter*. Il punto chiave é che il grafo di dipendenza dei dati, di per se, mostra in che modo le **istruzioni sono dipendenti dai dati**: *non é possibile eseguire un'istruzione prima che siano disponibili tutti i dati richiesti*. D'altra parte, non appena é terminata l'esecuzione di un'istruzione, tutte le altre istruzioni che stanno aspettando il suo dato di uscita possono, eventualmente, essere attivate[15]. Pertanto, l'esecuzione di un programma in forma di grafo che obbedisca a tale tipo di regole deve prevedere l'invio diretto dei dati da un'istruzione all'altra seguendo gli archi del grafo.

Appare evidente, a questo punto, che il modo piú naturale per esprimere programmi paralleli é quello di utilizzare una rappresentazione bidimensionale in forma di grafo dove le interconnessioni fra nodi istruzioni mettono in luce il minimo livello di sequenzialitá nell'esecuzione delle varie istruzioni. Naturalmente, la sola rappresentazione non é sufficiente ad evidenziare completamente il

[13] che diremo *data value*

[14] che diremo anche *data carrier*

[15] dopo aver verificato la disponibilitá degli altri dati di ingresso

parallelismo in essa implicito. E' necessario, in effetti, associare alla forma sintattica del programma[16] l'interpretazione di esso da parte di una potenziale macchina astratta[17] che operi sulla struttura a grafo in modo che le operazioni su di esso siano realmente interpretabili come realizzazioni di passi computazionali.

Si tratta, in effetti, di identificare un'opportuna struttura astratta di dati che possa vedersi come la definizione di una macchina universale che esegue tutti i possibili algoritmi rappresentabili nel modo che si é detto. Appare chiaro fin d'ora che tale struttura astratta di dati si ottiene a partire dal grafo introducendo un'opportuna operazione[18] che consiste nella ripetizione indefinita di un ciclo computazionale che, ad ogni passo, compie un insieme specificato semplice di operazioni su un'opportuna struttura dinamica, associata al grafo, del quale ne rappresenta lo *stato computazionale*, e che ne caratterizza univocamente il modo con cui la *computazione* progredisce.

Tale associazione puó ottenersi in due modi distinti: *statico* e *dinamico*. Nel primo caso il grafo che rappresenta il programma non cambia durante la computazione e si comporta come una struttura topologica che controlla e mantiene su di esso il flusso dei *data-token*. Nel secondo caso, anche la struttura topologica del grafo cambia ad ogni ciclo computazionale il quale é caratterizzato da un insieme di operazioni che trasformano[19] il grafo fino a quando ció é possibile; dopodiché la computazione termina producendo come risultato il grafo non ulteriormente riducibile.

L'**organizzazione delle elaborazioni** permette, a questo riguardo, di precisare il modo con cui é realizzata la computazione, esplicitando i cambiamenti di stato dell'elaboratore al termine dell'esecuzione di ciascuna istruzione e questo puó essere fatto considerando la computazione come una ripetizione continua di tre fasi.

- **Select** - In questa fase si sceglie, mediante una **regola di computazione**, un insieme di istruzioni per una possibile esecuzione, in modo tale che queste siano le uniche possibili candidate ad andare in esecuzione, anche se la selezione non garantisce l'esecuzione. Le regole di computazione possono essere:

  - imperativa, se le istruzioni selezionate dipendono dal contenuto del registro **contatore di programma**,
  - innermost, se le istruzioni selezionate sono quelle col maggior livello di **annidamento**,
  - outermost, quando le istruzioni selezionate sono quelle col minimo livello di **annidamento**.

  Per quanto riguarda le ultime due si faccia riferimento alla valutazione delle espressioni. La seconda richiede che *per valutare un'espressione é necessario aver valutato tutte le sue sottoespressioni componenti*[20]. L'ultima regola, piú debole delle precedente, *richiede solamente la valutazione di quelle sottoespressioni necessarie a valutare l'espressione data* e realizza quella che si chiama **lazy-evaluation**.

- **Examine** - Ciascuna delle istruzioni scelte nella fase precedente viene esaminata per verificarne l'eseguibilitá. La decisione, o **regola di attivazione**, é basata sull'ispezione degli argomenti attuali di ciascuna istruzione. Ad esempio, puó essere richiesto che tutti gli operandi di un'istruzione siano stati valutati. Se un'istruzione é eseguibile viene passata alla fase successiva per l'esecuzione; altrimenti, viene intrapresa qualche altra azione come ritardare l'esecuzione oppure costringere la valutazione degli argomenti per permetterne l'esecuzione.

[16] espresso in forma di grafo
[17] semantica operazionale
[18] l'interprete che esprime la sopracitata macchina universale
[19] o, riducono
[20] si tenga presente, a questo proposito, il funzionamento di un interprete LISP

- **Execute** - In questa fase *target* le istruzioni vengono realmente eseguite, determinando un cambiamento dello stato della macchina. Il risultato é reso disponibile e passato alle altre parti del programma. L'esecuzione puó produrre cambiamenti globali quando viene modificata la memoria condivisa globalmente, oppure locali quando un'espressione é sostituita dal suo valore.

La fase *select* di un elaborazione sequenziale corrisponde alla parte *fetch* del ciclo di controllo fetch-execute. Ogni elemento della computazione ha un contatore di programma che identifica la prossima istruzione che deve essere eseguita. Nella fase *select* il contatore di programma é utilizzato per scegliere le istruzioni che devono essere utilizzate. Una volta scelte, queste non vengono esaminate nella fase *examine*, ma vengono automaticamente passate alla fase successiva per l'esecuzione.

1.3 ORGANIZZAZIONE DEL PROGRAMMA

Con il termine *organizzazione del programma* si intende il modo con cui i programmi scritti in codice macchina sono rappresentati ed eseguiti nell'architettura della macchina. Due sono i meccanismi di calcolo basilari per l'esecuzione di un programma, e cioé,

- il meccanismo dei dati, e
- il meccanismo del controllo.

Con il termine **meccanismo dei dati** si intende il modo con il quale un particolare argomento é usato dalle istruzioni che lo richiedono. Esistono essenzialmente tre sottoclassi, per cui si parla di accesso ai dati

- letterale - L'argomento é noto durante la compilazione ed una copia separata é posta in ogni istruzione che accede a quell'argomento.
- per valore - L'argomento, generato durante l'esecuzione é condiviso per duplicazione, assegnando una copia separata ad ogni istruzione che accede a quell'argomento. La copia viene immagazzinata nell'istruzione come valore.
- per riferimento - L'argomento é condiviso per mezzo di un riferimento (puntatore) immagazzinato in ogni istruzione che accede a quell'argomento.

Il **meccanismo del controllo**, invece, definisce il modo con il quale le istruzioni determinano l'esecuzione di altre istruzioni e, quindi, il *flusso* esecutivo all'interno del programma. Anche in questo caso esistono tre sottoclassi per cui si parla di controllo

- sequenziale - Una singola linea di controllo segnala alle istruzioni la loro eseguibilitá.
- parallelo - Il controllo segnala la richiesta degli argomenti e l'istruzione viene eseguita quando tutti gli argomenti sono disponibili.
- ricorsivo - Il controllo segnala la necessitá degli argomenti e l'istruzione viene eseguita quando uno degli argomenti-risultato, generati dalla sua esecuzione, viene richiesto dall'istruzione che l'ha attivata. Appena eseguita, il controllo passa all'istruzione che l'ha invocata.

E' interressante notare come la classificazione precedente sia giá in grado di determinare, in quanto tale, almeno cinque possibili tipi di architetture secondo quanto appare nella tabella seguente

	per riferimento letterale	per valore letterale
sequenziale	Macchina di Von Neumann	
parallelo	parallel control flow	data flow
ricorsivo	riduzione a grafo	riduzione a stringa

dove le righe identificano il tipo di controllo e le colonne il modo con cui il programma accede ai dati. Si vede immediatamente che non risulta definita alcuna architettura in corrispondenza di controllo sequenziale e accesso ai dati per valore.

Gli elaboratori identificati dall'ultima riga della tabella si dicono, anche, macchine a riduzione perché eseguono programmi in forma di grafo che si modificano ad ogni passo computazionale. In questo caso le istruzioni, che ne etichettano i nodi, sono tutte realizzate come *riduzioni* nel senso che agiscono sulla topologia del grafo nel tentativo di renderla pi semplice. Nel seguito, comunque, verranno considerate solo le *macchine dataflow* per avere la caratteristica di eseguire programmi lasciando immutata la struttura topologica del grafo corrispondente. Questo ci permette di affermare che, anche nel caso parallelo, viene mantenuta la proprietá di *codice rientrante* per i programmi.

1.4 RAPPRESENTAZIONE DEI PROGRAMMI

La valutazione dell'espressione considerato precedentemente non é un esempio generale della pratica programmativa. Sono presenti solamente alcune operazioni aritmetiche e non c'é alcuna struttura di controllo, come le istruzioni *condizionali* o i *cicli*. In questa sezione verranno considerate alcune notazioni *dataflow* che permettono di realizzare programmi in forma piú generale. Poiché qualunque tipo di istruzione macchina puó essere rappresentata come un nodo di un grafo dataflow, il quale evidenzia le istruzioni che possono essere eseguite in parallelo, tale proprietá rende il grafo uno strumento ideale per esplorare il parallelismo irregolare presente nel software.

Il vantaggio nella valutazione di espressioni aritmetiche generali appare ovvio se si utilizzano le operazioni aritmetiche di base. In effetti, queste espressioni vengono facilmente convertite in grafi come giá accade nella maggior parte dei compilatori convenzionali che generano *alberi di valutazione delle espressioni* durante la fase di analisi del codice sorgente prima di produrre il codice oggetto equivalente. Il modello di esecuzione dataflow fa vedere come valutare direttamente tali alberi in un tempo proporzionale alla loro profonditá per mezzo dell'esecuzione parallela delle istruzioni.

Si ottiene ulteriore parallelismo quando si utilizzano le strutture di controllo, la piú semplice delle quali é l'istruzione condizionale e che viene realizzata introducendo nel grafo *archi di dipendenza condizionata*. Tali archi sono controllati dalla valutazione, al tempo di esecuzione, di predicati booleani e sono implementati utilizzando due istruzioni macchina di *scambio*, note come branch e merge. La

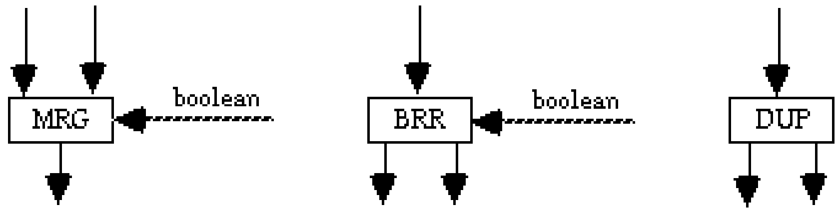

Figure 1.2: istruzioni di instradamento

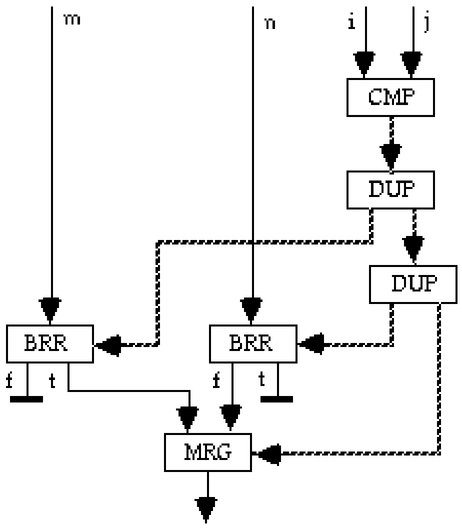

Figure 1.3: istruzione condizionale

fig. 1.2 visualizza la situazione dove BRR e MRG sono due interruttori che selezionano una delle due possibili strade che i token possono percorrere.

L'instradamento é determinato dal valore del *token di controllo* booleano e che assieme al *token dato* attende che entrambi siano disponibili come dati d'ingresso dell'istruzione di scambio, esattamente nello stesso modo delle istruzioni aritmetiche binarie e ternarie.

Laddove si é certi che sará disponibile uno solo dei dati d'ingresso per l'istruzione merge in corrispondenza dell'appropriato valore booleano associato[21], l'istruzione *merge* puó essere omessa dal codice macchina e i due archi congiunti direttamente.

Utilizzando l'estensione del grafo di dipendenza di dati con *archi di dipendenza condizionata* é possibile implementare l'istruzione condizionale del Pascal

$$\textbf{if}\ \ i = j\ \ \textbf{then}\ \ x \leftarrow m\ \ \textbf{else}\ \ x \leftarrow n$$

con il grafo che appare in fig.1.3 in cui il simbolo $\perp$ sta ad indicare che i token che vi giungono saranno distrutti.

L'istruzione etichettata con DUP, duplicate, ha l'effetto di copiare il token di ingresso su entrambi gli archi di uscita mentre CMP, compare genera un valore booleano che indica se i due token in ingresso hanno, o meno, lo stesso valore[22].

Le istruzioni di *commutazione* sono molto potenti quando sono utilizzate per implementare cicli e funzioni. Queste strutture sono fondamentali perché permettono di ricondurre computazioni, anche molto complesse, a programmi relativamente piccoli allo stesso modo dei convenzionali cicli, procedure e funzioni. Tali costrutti *rientranti*, comunque, pongono sostanziali problemi implementativi dovuti alla possibilitá di attivazione simultanea del codice rientrante.

Si consideri, ad esempio, il seguente frammento programma Pascal per il calcolo della funzione fattoriale in forma iterativa.

[21] prodotto dalla precedente istruzione branch usando lo steso valore di controllo

[22] per una lista di istruzioni assembler di una possibile macchina dataflow si veda piú avanti

```
k    := 1;                      k    := 1;
i    := n;                      i    := n;
while ( i > 0 ) do          1:  if  ( i = 0 ) then goto 2;
   begin
      k   := k * i;             k    := k * i;
      i   := i - i;             i    := i - 1;
   end;                         goto 1;
Fact := k;                  2:  Fact := k;
```

Per facilitá di comprensione, accanto ad esso, compare lo stesso programma in cui l'istruzione *while* é espressa solamente in termini di salto condizionato. La sua traduzione in grafo di dipendenza dei dati a livello macchina appare in fig.1.4.

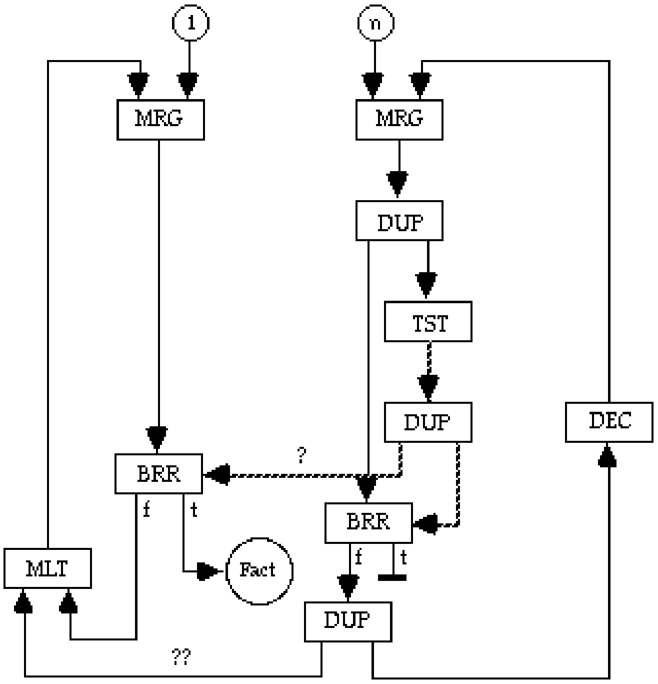

Figure 1.4: fattoriale

Un'analisi dettagliata di questo grafo rivela che sugli archi etichettati ? e ?? puó transitare piú di un token per cui é essenziale che gli archi del grafo si comportino come code FIFO[23], altrimenti il ciclo potrebbe terminare anticipatamente a causa del maggior numero di token in transito sull'arco ?. Sfortunatamente, a causa della difficoltá di realizzare code di dimensioni arbitrarie, si é soliti restringere la *regola di attivazione* dei sistemi dataflow pratici in modo che *le istruzioni possano essere eseguite solamente quando i loro archi di uscita sono vuoti.*

Questo é il metodo piú semplice, anche se non del tutto generale, di implementare i *grafi*[24] *rientranti* perché impedisce la rientranza concorrente consentendo solamente una riattivazione strettamente

[23]first-in-first-out
[24]di programma

sequenziale dei cicli. Sebbene risulti limitato il parallelismo supplementare, ottenibile dalla sovrapposizione di piú passi esecutivi di un ciclo, é conveniente disporre di uno schema implementativo piú generale al fine di mettere in luce tutto il parallelismo possibile insito nei grafi di dipendenza dei dati. Si puó, allora, dare la seguente definizione.

Si dice che un sistema di computazione dataflow é **statico** *se permette di eseguire solamente codice rientrante sequenzialmente ciclico ed é realizzato restringendo la regola di attivazione alle istruzioni i cui archi di uscita sono vuoti.*

Un caso tipico in cui é richiesta la *rientranza concorrente* é quello che si verifica quando il programmatore definisce una funzione[25] che puó essere chiamata da diversi punti del programma. Questa situazione é analoga alla definizione di procedure o funzioni nel Pascal. Naturalmente, é possibile creare diverse copie del codice macchina, che rappresenta la funzione, e inserirlo *in linea* nella posizione richiesta. Questa soluzione risulta impraticabile sia nel caso di definizioni di larghe dimensioni che in quello di chiamate frequenti. Inoltre, tale soluzione impedisce l'uso della ricorsione perché questo implica la disponibilitá di grafi-programma indefinitamente espansi.

Sono possibili al riguardo due schemi implementativi a seconda che i token condividano, o meno, lo stesso sottografo. In entrambi i casi, i sistemi dataflow che realizzano le *funzioni concorrentemente rientranti* si dicono **dinamici**.

Il primo di tali schemi consente la *rientranza concorrente* per mezzo di un'istruzione apply inserita all'inizio di un sottografo definito dall'utente, la quale *crea una nuova copia del sottografo ogniqualvolta viene attivata.* Tutti i token di ingresso all'attivazione del sottografo sono raccolti insieme dall'istruzione apply e quindi trasferiti all'unica nuova copia del codice rientrante.

L'istruzione exit, posizionata alla fine della copia del sottografo, raccoglie insieme tutti i token di uscita per l'attivazione e li restituisce lungo gli archi di uscita dell'appropriata istruzione apply. Successivamente, la copia del codice rientrante viene distrutta. Il meccanismo di funzionamento di questo schema é analogo alla *espansione di una macro* in quanto viene generato nuovo codice e allocato nuovo spazio ogniqualvolta viene chiamata la funzione. In questo modo, peró, si evita che i dati condividano concorrentemente il codice. Si puó, allora, dare la seguente definizione.

Si dice che un sistema di computazione dataflow impiega uno **schema di copiatura dinamica del codice** *se, utilizzando la coppia di istruzioni* apply *e* exit*, crea una nuova copia del sottografo ogni qualvolta se ne richiede la sua attivazione.*

Lo schema alternativo é quello che consente la condivisione del codice e che, allo scopo, impiega la segnatura[26] dei token al momento del loro ingresso e della loro uscita dalle aree rientranti. Questo sistema é molto simile all'uso della pila nell'implementazione delle procedure e delle funzioni sulle macchine convenzionali, a parte il fatto che l'attivazione concorrente di grafo-codice condiviso richiede che ciascun token sia segnato individualmente con un appropriato *nome di base* piuttosto che utilizzare registri globali della pila per identificare lo spazio dei dati correntemente attivo.

In termini visivi la segnatura puó pensarsi come il processo di coloratura dei token di dati. La regola di attivazione delle istruzioni del grafo deve, allora, essere modificata in modo tale che *solo i token dello stesso colore*[27] *possono essere raggruppati per determinare l'attivazione di un'istruzione.* Sono necessarie, quindi, speciali istruzioni per creare un nuovo colore all'ingresso del codice rientrante e ristabilire il vecchio all'uscita. Naturalmente, i token devono portare ulteriore informazione per la codifica del colore. Siamo, pertanto, in grado di dare la seguente definizione.

[25] cioé un sottografo definito dall'utente

[26] tagging

[27] cioé, quelli che hanno la stessa segnatura

Si dice che un sistema di computazione dataflow impiega uno **schema di condivisione dinamica del codice** *o* **coloratura dinamica** *se ogni token, individualmente, viene fornito di un appropriato* colore *al momento del suo ingresso nell'area di* codice rientrante.

Si noti che la coloratura dei token puó essere utilizzata per distinguere dati che appartengono a differenti passi di un ciclo. Ad esempio, nel programma esemplificato in fig.1.4, se si assume che il colore[28] di tutti i token di ingresso é zero, questo puó essere incrementato ogni volta che viene eseguito un passo del ciclo quando il token sta arrivando all'istruzione MRG e riporato a zero al momemto dell'uscita dall'ultimo ciclo, quando porta il valore della funzione *fact*. Cosí facendo non é piú necessario che gli archi si comportino come code FIFO, consentendo l'utilizzo della regola di attivazione senza restrizioni.

1.5 ARCHITETTURA DELLE MACCHINE DATA FLOW

Il modello dataflow permette di rappresentare il possibile parallelismo, presente implicitamente nei programmi sequenziali, per mezzo di grafi orientati evitando di introdurre qualunque sequenzializzazione non richiesta. La piú ovvia realizzazione di tale modello consiste nell'assegnare un processore per ciascun nodo computazionale del grafo assieme ad una rete di comunicazione riconfigurabile che fornisce le richieste interconnessioni.

Ci sono almeno due ragioni per non prendere in considerazione questa ipotesi realizzativa. La prima é dettata dal fatto che il sistema apparirebbe inevitabilmente molto inefficiente a causa dell'inattivitá di gran parte dei processori quando i corrispondenti nodi computazionali del grafo sono, essi stessi, inattivi. La seconda dipende dalla enorme complessitá della rete di comunicazione necessaria a stabilire tutte le possibili interconnessioni fra nodi.

Anche se la prima obezione venisse superata allocando molti nodi computazionali su un singolo processore, resta il fatto che é meglio procedere verso un'*astrazione dell'interconnessione fisica* dei grafi dataflow per arrivare ad architetture piú realistiche. Se ciascun nodo é allocato con un unico identificatore, le interconnessioni possono essere introdotte nella *descrizione del nodo* realizzata con l'identificatore del nodo successivo[29] verso il quale i dati verranno inviati al termine della computazione.

E' possibile, allora, esplicitare i cammini di interconnessione con una rete generalizzata di comunicazione. Il valore di un dato, quando viene computato, usa l'informazione del nodo successivo come direttiva di *instradamento*[30] per trovare il processore che contiene l'operazione richesta. Dovrebbe essere chiaro, a questo punto, come tale tecnica permetta ai programmi dataflow di essere mappati su sistemi multiprocessore convenzionali dove un certo numero di nodi vengono allocati staticamente su ciascuno dei processori.

Si tenga presente che la comunicazione é essenzialmente unidirezionale nel senso che quando un valore é stato determinato come il risultato di un'operazione le risorse utilizzate possono essere rilasciate immediatamente per eseguire un'altra operazione. Nessuna risposta é richiesta e, finché il processore che contiene il nodo destinazione ha altro lavoro da fare, il tempo di trasmissione al destinatario non é critico. Si noti, anche, che non esiste alcuna necessitá a mantenere l'ordine attraverso la rete di comunicazione. Sono questi gli aspetti del modello dataflow che rilassano i vincoli architetturali sulla struttura fisica e rendono non strettamente necessario l'approccio multiprocessore.

La realizzazione piena dei modelli computazionali dataflow si ottiene impiegando una struttura ad anello che permette di superare agevolmente la necessitá di uno schema flessibile di interconnessione

[28] tag
[29] assieme con un valore di ingresso
[30] routing

ad elevata larghezza di banda in quanto viene sfruttata *la proprietá di unidirezionalitá della comunicazione e di trasmittibilitá a velocitá contenute*. L'unico prezzo da pagare é il *ritardo di pipeline* fra i processori. Finché questi rimangono attivi e il pipeline é saturo, per il parallelismo del programma, non compare alcun deterioramento apprezzabile delle prestazioni.

Si supponga, ora, di realizzare le tre funzioni fondamentali dei sistemi dataflow, *determinazione dell'istruzione successiva*, *matching dei token* ed *esecuzione delle istruzioni*, utilizzando tre moduli hardware separati fra loro. Le singole azioni di tali moduli dipendono soltanto dal modulo di *input dei dati* cosicché le azioni successive possono sovrapporsi se si connettono i moduli in pipeline. La struttura complessiva di questa realizzazione appare in fig.1.6.

L'unitá fondamentale di dato nello *switch* é il token a pacchetto[31], che rappresenta un token segnato[32] in transito su un arco del grafo di programma, e che é costituito da un tipo, un valore e un colore. L'arco é rappresentato da un indirizzo[33] dell'istruzione puntata dall'arco, noto come destinazione. Il token a pacchetto é la piú piccola unitá di informazione presente nel sistema cosicché il modulo di accodamento é posizionato in prossimitá dello switch, all'ingresso dell'anello.

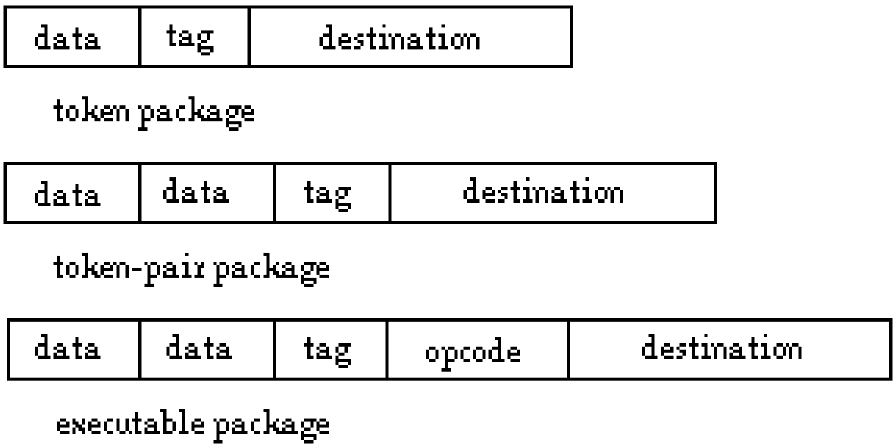

Figure 1.5: impacchettamento dei token

Uno per volta i token presenti nella coda vengono rimossi e presentati all'unitá di *matching*, che é responsabile del raggruppamento dei token aventi lo stesso colore di intestazione per la stessa istruzione di destinazione. Limitando le istruzioni ad un numero massimo di operandi pari a due, é sufficiente raggruppare i token singolarmente o a coppie. I token che aspettano di trovare il partner, ma che giungono all'unitá di matching prima del suo arrivo, sono temporaneamente immagazzinati nell'unitá dei token spaiati[34]. A questo punto, o nel caso di istruzioni a ingresso singolo quando arriva il primo ed unico token, tutti i dati di ingresso richiesti, aventi lo stesso colore per la stessa destinazione, sono inviati all'unitá che gestisce il prelievo delle istruzioni in forma di *pacchetto di token appaiati*[35].

Il grafo che definisce il programma é immagazzinato come una sequenza di istruzioni ciascuna delle quali rappresenta un operatore e i corrispondenti archi di uscita. Il campo *destinazione* di una

[31] token package
[32] tagged token
[33] nella memoria in cui é immagazzinato il grafo di programma
[34] unmatched-token unit
[35] token-pair package

pacchetto di token appaiati in arrivo é utilizzato come indirizzo per prelevare la prossima istruzione contenente il codice dell'operazione e fino a due campi destinazione. Tutto ció produce un pacchetto eseguibile completo che viene inviato all'unitá di elaborazione. A questo punto viene eseguita l'operazione relativa al corrispondente codice utilizzando come operandi le coppie dati-colore preventivamente collezionati producendo come risultato uno o piú token che vengono instradati verso lo *switch*. La fig.1.6 schematizza tale meccanismo di funzionamento.

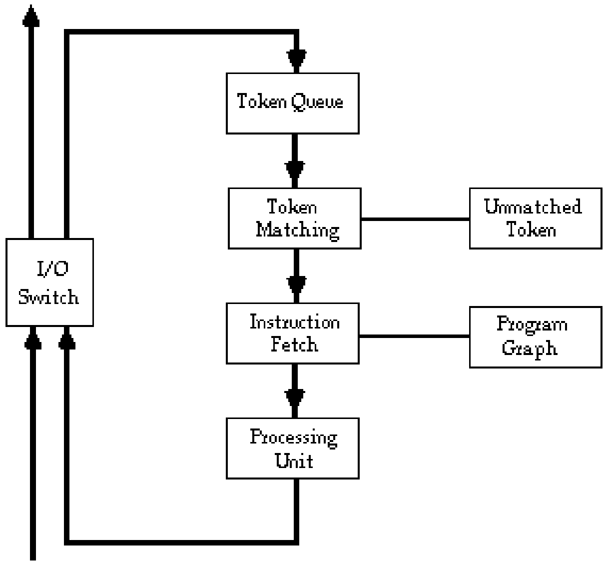

Figure 1.6: architettura di una macchina dataflow

Ovviamente, la parte critica del sistema é l'unitá di matching in quanto l'operazione di accoppiamento dei token é un atto tipicamente associativo per cui la memoria in cui vengono temporaneamente immagazzinati i token spaiati é naturalmente associativa e deve essere realizzata con un meccanismo hardware di hashing. La scelta tecnologica dell'unitá di matching determina, quindi, il limite della velocitá di esecuzione delle istruzioni lungo l'anello.

Per evitare il passaggio di un eccessivo numero di token all'interno dell'anello é necessario immagazzinare in una memoria a parte le grosse strutture di dati le quali, pertanto, sono rappresentate mediante token puntatori che sostituiscono i corrispondenti valori nel transito lungo l'anello stesso.

1.6 ISTRUZIONI DI UNA MACCHINA DATAFLOW

Come si è visto nella sezione precedente, le macchine dataflow eseguono grafi orientati i cui nodi sono istruzioni e i cui archi sono canali di comunicazione per trasmettere dati da un'istruzione all'altra in forma di pacchetti, detti token. Per permettere il massimo parallelismo possibile ciascun token viene segnato per distinguerlo da altri token che eseguono il medesimo codice ma in condizioni diverse. La segnatura, o *tag*, è realizzata con un **nome di attivazione**[36], o colore, per separare le diverse attivazioni della stessa funzione o del medesimo ciclo, e un **indice** per distinguere gli elementi di una stessa struttura dati.

[36]activation name

Nel caso più semplice tale macchina è costituita da un solo anello di unità elaborative ed, inoltre, il grafo di programma è costituito da nodi che possono avere uno o due archi di ingresso o uscita, eventualmente, nessuno in uscita. La restrizione sul numero di archi di ingresso consente di realizzare l'unita di matching in modo semplice e diretto. Ci sono solamente pochi casi in cui non è richiesto alcun valore di uscita per cui, in pratica, verranno considerate solo nodi con una o due destinazioni.

Il campo *destinazione* di un'istruzione identifica il nodo a cui l'arco è connesso, a quale dei due possibili input si riferisce e la corrispondente *funzione di matching* da eseguire. Quest'ultima si rende necessaria perchè deve istruire l'‘it unità di matching sull‘azione da svolgere in corrispondenza al token in transito su quell'arco, cioè se tentare o meno di appaiare il token con l‘eventuale suo corrispondente diretto verso lo stesso nodo.

Quindi, durante la normale esecuzione di un programma dataflow, sono necessarie solamente due funzioni,

- **BY**, *bypass*, per gli operandi che vanno verso operatori unari oppure verso operatori binari con un operando letterale;

- **EW**, *extract and wait*, per gli operandi che devono essere appaiati in istruzioni binarie.

Altre funzioni di matching possono risultare utili ma, in generale, sono impiegate per scopi speciali. Tutte le funzioni di matching, a parte BY, sono realizzate nel seguente modo. Quando un operando arriva alla *memoria di matching*[37] viene eseguita una ricerca pseudoassociativa per identificare l'operando complementare della coppia di operandi che si riferisce all'istruzione indicata nel campo destinazione. Sia in caso di successo che di insuccesso, viene richiesta l'esecuzione di una certa azione da parte della *memoria di matching.*

Le possibili azioni sono, nel caso di matching con successo:

- extract, per rimuovere l'operando appaiato dalla memoria di matching;

- preserve, per produrre una copia dell'operando appaiato da lasciare nella memoria di matching;

- increment, che lascia una copia, con l'indice incrementato, dell'operando appaiato, utilizzata per accedere agli elementi di un array;

- decrement, che lascia una copia, con l'indice decrementato, dell‘operando appaiato.

Le azioni eseguite, in caso di insuccesso, sono

- wait, per rinserire l'operando nella memoria degli operandi spaiati con lo scopo di tentare successivamente il matching;

- defer, per rinunciare il matching al momento e tentarlo successivamente;

- abort, per generare un pseudo-operando *vuoto* da appaiare, comunque, all'operando spaiato;

- generate, come la precedente, ma con la caratteristica di immagazzinare l'operando in arrivo nella *memoria di matching* come se stesse andando al punto di ingresso opposto del nodo in questione, utilizzata nella *lazy evaluation.*

[37]matching store

Si noti che alcune funzioni di matching sono legate al modo con cui sono rappresentati e immagazzinati i dati strutturati, e cioè, gli array e i record i quali, sono allocati staticamente in memoria[38] e, ad essi, si accede nel modo solito, con indirizzo base e offset, o indice.

La macchina dataflow in esame utilizza *token tipizzati* nel senso che ogni valore è associato ad uno specifico tipo, e le istruzioni accettano o un tipo particolare oppure uno qualunque fra quelli possibili. Tranne gli ordinari tipi di dati scalari come i boolean, gli interi e i reali, vengono definiti, ai fini del controllo, un certo numero di tipi speciali:

- nome di attivazione,
- indice,
- etichetta, una combinazione dei precedenti,
- destinazione, una combinazione di indirizzo di nodo più l'informazione del punto di ingresso, che può essere destro o sinistro,
- contesto, una combinazione di destinazione e nome di attivazione,
- puntatore, nella memoria dei dati strutturati.

L'esecuzione di un'istruzione può produrre un *singolo risultato logico*[39] oppure un doppio *risultato logico*[40].

Con il termine logico si intende che i due possibili archi risultato sono valori distinti usati con differenti propositi, senza riferimento alcuno al numero di copie prodotte del risultato. Così l'istruzione di duplicazione fornisce due copie del proprio input, ma questa è un'istruzione a risultato singolo. L'esempio classico di istruzione a risultato doppio è quella che esegue la divisione intera e che fornisce il quoziente su un arco e il resto sull'altro.

Con un'istruzione a risultato singolo il nodo può produrre una o due copie dell'unico risultato ottenuto, a seconda del numero di destinazioni che compaiono all'interno del nodo. Con istruzioni a doppio risultato bisogna tener presente che, se è richiesto uno solo dei due risultati logici, questo deve essere indicato dal prefisso dell'istruzione per cui, le due destinazioni disponibili possono essere utilizzate per fornire una o due copie del medesimo risultato. Altrimenti, il prefisso indica che sono richiesti entrambi i risultati, nel qual caso quello di sinistra è indirizzato alla prima destinazione e quello di destra alla seconda. Dunque, i possibili prefissi sono i seguenti:

N	per un'istruzione singola;
D	per un'istruzione doppia che utilizza entrambi i risultati logici;
L	per un'istruzione doppia che usa il solo risultato di sinistra;
R	per un'istruzione doppia che utilizza il solo risultato di destra

La maggior parte degli archi di un grafo dataflow sono statici, il che significa che sono fissi e congiungono permanentemente un nodo all'altro. Ci sono, però, delle situazioni per cui la destinazione verso la quale un nodo deve inviare il risultato deve essere differente in relazione ad attivazioni separate di quel nodo. L'esempio classico è quello che si ha quando una funzione restituisce un risultato alla parte del grafo che l'ha chiamata. Analogamente al caso degli elaboratori convenzionali, vorremmo poter scegliere se duplicare il codice del corpo della funzione ovunque è chiamata oppure avere soltanto

[38] e, quindi, si comportano come dati globali per le istruzioni che li utilizzano
[39] per cui si parla di istruzioni a risultato singolo
[40] per cui si parla di istruzioni a doppio risultato

un'unica copia a cui passare gli argomenti e ricevere i risultati da restituire al grafo che l'ha chiamata. Il passaggio di argomenti al corpo di una funzione è possibile purchè si possa utilizzare un arco dinamico per il risultato di ritorno. La destinazione per questo arco è data dall'operando di destra dell'istruzione SCD che crea l'arco, essendo l'operando di sinistra il valore da inviare. L'informazione di ritorno al momento della chiamata di una funzione è data dalla destinazione in cui il risultato deve essere restituito.

Come si é accennato precedentemente, l'introduzione della *segnatura dei token* consente un utilizzo piú flessibile dei grafi dataflow. Si supponga, allora, che ciascun token sia caratterizzato da tre campi etichetta: il *nome di attivazione*, il *livello di iterazione* e l'*indice*. In particolare, il secondo é particolarmente adatto a trattare correttamente l'attivazione contemporanea dei diversi passi esecutivi di un ciclo computazionale.

Le istruzioni descritte nel seguito manipolano i campi etichetta in vari modi. Ad esempio, l'istruzione *YIL*[41] consente di determinare il livello di iterazione dell'operando di ingresso, mentre l'istruzione *SIL*[42] fa si che il risultato di sinistra sia il medesimo dell'operando di sinistra tranne il livello di iterazione che é posto uguale al valore dell'operando di destra. Il risultato logico di destra potrebbe essere impiegato in situazioni d'errore che si potrebbero verificare sia perché l'operando destro non é un tipo ordinale sia perché non puó essere rappresentato nello spazio ad esso assegnato.

L'istruzione *ADL*[43] é simile alla precedente a parte l'impiego dell'operando destro, che deve essere di tipo intero, e il cui valore viene sommato al livello di iterazione corrente, invece di sostituirlo. Si possono introdurre istruzioni simili per la manipolazione del nome di attivazione e dell'indice, relativi alla segnatura dei token.

Si consideri, ora, il seguente sottoinsieme di istruzioni della macchina dataflow in esame, scelte allo scopo di illustrare la scrittura di grafi di programmi per semplici algoritmi e per schematizzare la compilazione a livello macchina dei programmi di alto livello. A meno che non sia specificamente indicato, i token prodotti dall'esecuzione di un'istruzione hanno lo stessa *segnatura*[44] o *colore*, dei rispettivi token di input. Le marcature *LV* e *RV* denotano, rispettivamente, i valori sinistro e destro dei due archi di input dell'istruzione mentre *AN* e *IX* sono utilizzati per rappresentare il nome di attivazione e l'indice.

Le seguenti sono tipiche *istruzioni aritmetiche*

Istruzione	tipo di dati		Commento
ADI	Int	Int	Somma due valori interi; il risultato è LV + RV
INC	Int		Incremento unitario; il risultato è LV + 1
ADR	Real	Real	Somma due valori reali; il risultato è LV + RV
SBI	Int	Int	Sottrae due valori interi; il risultato è LV - RV
DEC	Int		Decremento unitario; il risultato è LV - 1
SBR	Real	Real	Sottrae due valori reali; il risultato è LV - RV
MLI	Int	Int	Moltiplica due valori interi; il risultato è LV * RV
MLR	Real	Real	Moltiplica due valori reali; il risultato è LV * RV
DRM	Int	Int	Dividi i due valori interi producendo quoziente e resto; il risultato è R1 := LV **div** LR, R2 := LV **mod** RV

[41] Yields the Iteration Level
[42] Set the Iteration Level
[43] ADd to the Iteration Level
[44] tag

Istruzione	tipo di dati		Commento
TST	Int		Verifica se il valore intero è nullo; il risultato è LV = 0
CMP	Int	Int	Confronta due valori interi; il risultato è LV = LR
GTR	Int	Int	Verifica se il primo valore è maggiore del secondo; il risultato è LV > LR;
GEQ	Int	Int	Verifica se il primo valore non è minore del secondo; il risultato è LV >= LR;
LSS	Int	Int	Verifica se il primo valore è minore del secondo; il risultato è LV < LR;
LEQ	Int	Int	Verifica se il primo valore non è maggiore del secondo; il risultato è LV <= LR;

Le *istruzioni di controllo* tipicamente ricevono in ingresso un dato, che puó essere di qualunque tipo, sull'arco sinistro, e un segnale di controllo sull'arco destro, che determina cosa accade al dato. Alcune istruzioni di controllo inviano il valore presentato come dato d'ingresso verso un arco dinamico con le specifiche dell'input di controllo; altre sono del tipo BRR, oppure manipolano etichette come specificato in precedenza. Nelle istruzioni presentate di seguito il valore V, con nome di attivazione A e indice I, é denotato con $V_{<A,I>}$. Inoltre *Act* sta per *nome di attivazione* e *Dest* per *destinazione*. Laddove i nodi hanno due risultati logici, questi vengono denotati con R1 e R2, rispettivamente.

Istruzione	tipo di dati		Commento
DUP	Any		DUPlicate;
SYN	Any	Any	SYNchronize; R1 := LV, R2 := LR
BRR	Any	Bool	Branch; if RV then R1 := LV else R2 := LV
BIX	Any	Bool	Branch and Increment indeX; if RV then R1 := $LV_{<AN,IX+1>}$ else R1 := $LV_{<AN,IX+1>}$
ADX	Any	Int	ADd to Index; il risultato é LV tranne l'indice IX+RV, cioé $LV_{<AN,IX+RV>}$
SLB	Any	Label	Set LaBel; il risultato é LV ma con indice RV, cioé $LV_{<RV.AN,RV.IX>}$
SAN	Any	Act	Set Activation Name; il risultato é LV ma con nome di attivazione RV, cioé $LV_{<RV,IX>}$
SAZ	Any	Act	Set Activation Name and Zero Index; il risultato é LV ma con nome di attivazione RV e indice 0, cioé $LV_{<RV,0>}$
GAN	Any		Generate Activation Name; il risultato é un nuovo nome di attivazione
PRP	AN	Dest	Prepare Access; viene creato il contesto C = [RV,AN]; il nome di attivazione del token di uscita é LV cosí il risultato é $C_{<LV,IX>}$
SCD	Any	Context	Set Colour and Destination; questa é un'istruzione di controllo per archi dinamici; una copia di LV, a parte il nome di attivazione posto uguale a quello del valore di contesto RV, é inviata alla destinazione specificata da RV
PRL	Any	Int	Proliferate; produci RV copie di LV con indici da IX a IX + RV - 1

Per accedere alla memoria strutturata[45] sono necessarie ulteriori istruzioni, le quali utilizzano puntatori[46] e valori per costruire messaggi da inviare ad essa. Nel seguito, i campi *indirizzo* e *dimensione* di un valore puntatore V sono denotati con V.A e V.S, rispettivamente.

Istruzione	tipo di dati		Commento
GSP	Int		Generate Structure Store Pointer; genera una richiesta di allocazione in memoria di una struttura di dimensione RV; il risultato é il puntatore restituito dalla memoria strutturata
WSS	Any	P	Write to Structure Store; genera una richiesta di scrittura nella memoria per scrivere LV all'indirizzo (RV.A)+IX+1; non viene restituito nulla
RSS	P	Int	Read from Structure Store; genera una richiesta di lettura nella memoria per leggere un valore dall'indirizzo (LV.A)+RV+1 e portarlo come risultato dell'istruzione
SVA	P	Int	Single Vectore Access; genera una richiesta di lettura nella memoria per leggere un valore dall'indirizzo (LV.A)+RV e portarlo come risultato
DSR	P	Int	Double Read from Structure Store; genera due richieste di lettura nella memoria per leggere i valori agli indirizzi (LV.A)+RV+1 e (LV.A)+RV+2 per portarli come risultati sinistro e destro dell'istruzione, rispettivamente
FSS	P		Fetch Stream from Structure Store; genera una serie di richieste per estrarre dalla memoria un flusso di LV.S valori a partire dall'indirizzo LV.A e restituendoli come un unico output. I token risultanti hanno indici da 0 a (LV.S)-1

Come si é giá detto, l'esecuzione del programma é ottenuta generando e sostenendo il transito di token lungo gli archi che collegano le istruzioni nel corrispondente grafo dataflow e questo viene realizzato, nella pratica, controllando i messaggi[47] che transitano lungo l'anello[48] permettendo, eventualmente, un'interscambio con l'esterno attraverso l'unitá di interfaccia denominata *I/O switch*. D'altra parte, anche i programmi sono opportunamente caricati all'interno dell'anello in forma di *messaggi speciali* cosí come avviene per altre azioni che devono essere intraprese dalle unitá che costituiscono l'anello.

Nel seguito é riportato il listing del programma, nella forma simbolica a piú basso livello possibile, per il calcolo del fattoriale in forma iterativa.

[45] memoria statica, condivisa, in cui sono memorizzati array e record
[46] tipo P
[47] che rappresentano i token
[48] si veda la fig. 1.6

0	N	DUP	1	L	BY	2	L	EW
1	N	SBI	6	L	BY	#1		
2	D	BRR	4	R	EW	10	L	BY
3	N	CEI	7	L	BY	#0		
4	N	MLI	5	L	BY			
5	L	ADL	2	L	EW	#1		
6	N	DUP	8	L	EW	3	L	BY
7	N	DUP	2	R	EW	8	R	EW
8	L	BRR	4	L	EW	9	L	BY
9	L	ADL	1	L	BY	#1		
10	R	OPT	11	L	BY	#0		
11	N	KIL	0	L	BY	0	L	BY

Come si puó notare la sua forma é piú complessa rispetto a quella di un normale listing assembler sequenziale perché in esso di deve tener conto della struttura bidimensionale insita nel codice e delle azioni che devono essere intraprese dall'unitá di matching per la sua corretta esecuzione.

1.7 GENERAZIONE DEL CODICE

I valori scalari sono rappresentati da token di tipo appropriato. Un letterale, ad esempio, é rappresentato da un'istruzione SYN che ha quel letterale come parte dell'istruzione e un input di regolazione[49]. I regolatori sono token aggiuntivi che vengono inviati al corpo di una funzione o di un ciclo per attivare parti del grafo dataflow che altrimenti non potrebbero essere, in alcun modo, attivati automaticamente dai dati.

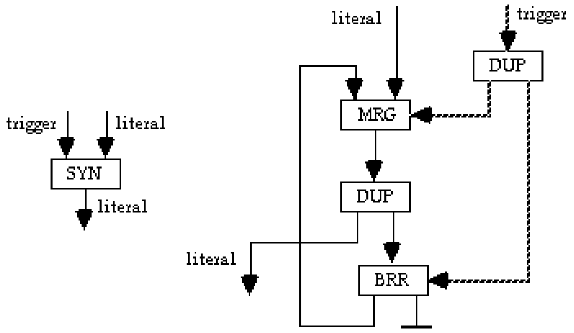

Figure 1.7: letterale

In molti casi i letterali si possono implementare senza l'uso di regolatori, semplicemente propagandoli ai nodi che li usano.

La rappresentazione di grosse strutture di dati, come gli array, gli stream o i record, si ottiene impiegando due tecniche distinte. La prima, immagazzinando la struttura e riferendola con un puntatore, puó provocare una riduzione del parallelismo. La seconda, associando alla struttura un certo numero di token distinguibili per mezzo del campo *indice* della segnatura, provoca un eccessiva copiatura di token rendendo l'accesso casuale molto inefficiente.

[49] trigger input

Seguendo il primo approccio, gli array sono rappresentabili da un descrittore che consiste di due token: un puntatore ed un intero per l'estremo inferiore.

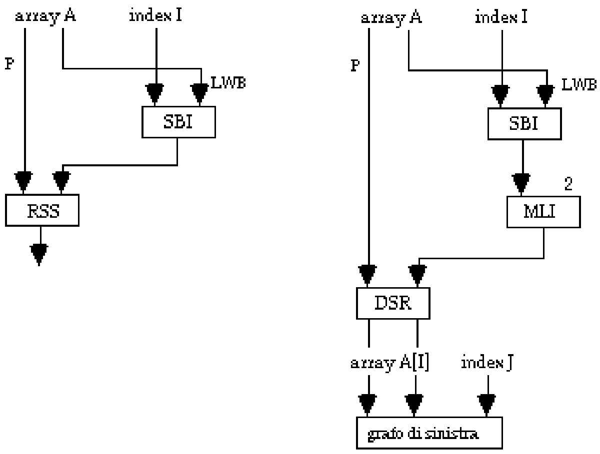

Figure 1.8: accesso a un vettore

Se gli elementi dell'array sono essi stessi strutturati, anche le sottostrutture devono essere immagazzinate e rappresentate da descrittori. Nella figura precedente la parte sinistra mostra il codice per accedere ad un vettore di valori scalari mentre la parte destra definisce il meccanismo d'accesso ad una matrice bidimensionale.

Il grafo dataflow, che appare nella figura seguente, mostra il meccanismo della chiamata di funzione nel caso di una funzione F(i) intera a valori interi.

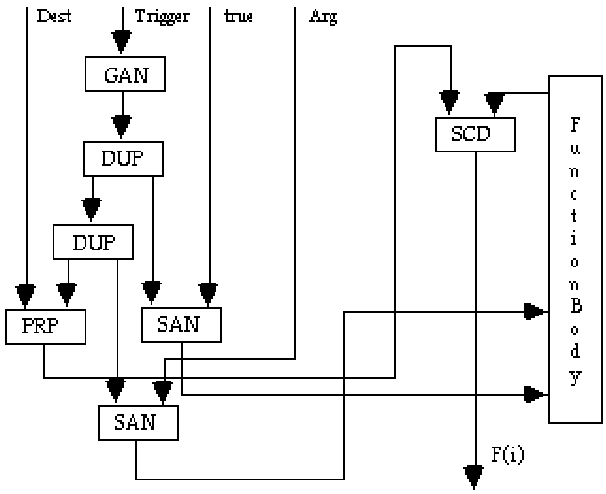

Figure 1.9: chiamata di funzione

Ogni funzione é determinata da un *grafo d'accesso* ed, inoltre, ogni chiamata di funzione é associata ad un unico *nome di attivazione* che viene generato dall'istruzione GAN[50] ed inviato all'istruzione SAN[51] nel grafo d'accesso e dar luogo al regolatore usato all'interno del corpo della funzione. Il nuovo nome di attivazione viene utilizzato anche per segnare gli argomenti e gli indirizzi di ritorno dei risultati. Si tenga presente che devono comparire tante istruzioni SAN quanti sono gli argomenti dlla funzione piú un'altra per la funzione stessa.

Il *grafo chiamato* contiene un'istruzione *PRP*[52] per ogni risultato fornito con il compito di inviare all'istruzione *SCD*[53], nel *grafo d'accesso*, l'indirizzo di ritorno nella forma di contesto, cioé, l'indirizzo del risultato e il vecchio nome di attivazione segnato col nuovo mome di attivazione. Quest'ultima istruzione, non appena riceve il risultato dal corpo della funzione, lo ricolora con il vecchio nome di attivazione e lo invia alla rispettiva destinazione nel grafo chiamante.

1.8 ESEMPI DI PROGRAMMI DATAFLOW

Sono riportati nel seguito alcuni semplici esempi di programmi in forma di grafo dataflow. Essi si riferiscono a semplici funzioni ricorsive intere la cui definizione é data in forma ricorsiva. Dopo averne determinata la corrispondente forma iterativa, utilizzando cicli strutturati while o repeat, ne viene stabilita la forma iterativa non strutturata, con l'uso di istruzioni di salto condizionato o assoluto. A questo punto, una breve analisi del codice, mettendo in luce l'eventuale parallelismo, conduce alla realizzazione del codice dataflow.

1.8.1 Divisione Intera

Assegnata la definizione ricorsiva della funzione che esegue la *divisione intera* di una coppia di numeri interi

$$IDiv(n,m) \equiv \textbf{if}(n \geq m)\textbf{then} \quad IDiv(n-m,m)+1\textbf{else}0$$

la versione in forma iterativa, utilizzando un costrutto di tipo while, é quella che compare di seguito

```
q := 0;
r := n;
while ( r ≥ m ) do
begin
    q := q + 1;
    r := r - m;
    end;
IDiv := q;
```

```
    q := 0;
    r := n;
1:  if ( r < m ) then goto 2;
    q := q + 1;
    r := r - m;
    goto 1;
2:  IDiv := q;
```

A questo punto é immediato realizzare l'algoritmo che la valuti col maggior livello di parallelismo possibile.

[50] Generate Activation Name
[51] Set Activation Name
[52] Prepare Access
[53] Set Colour and Destination

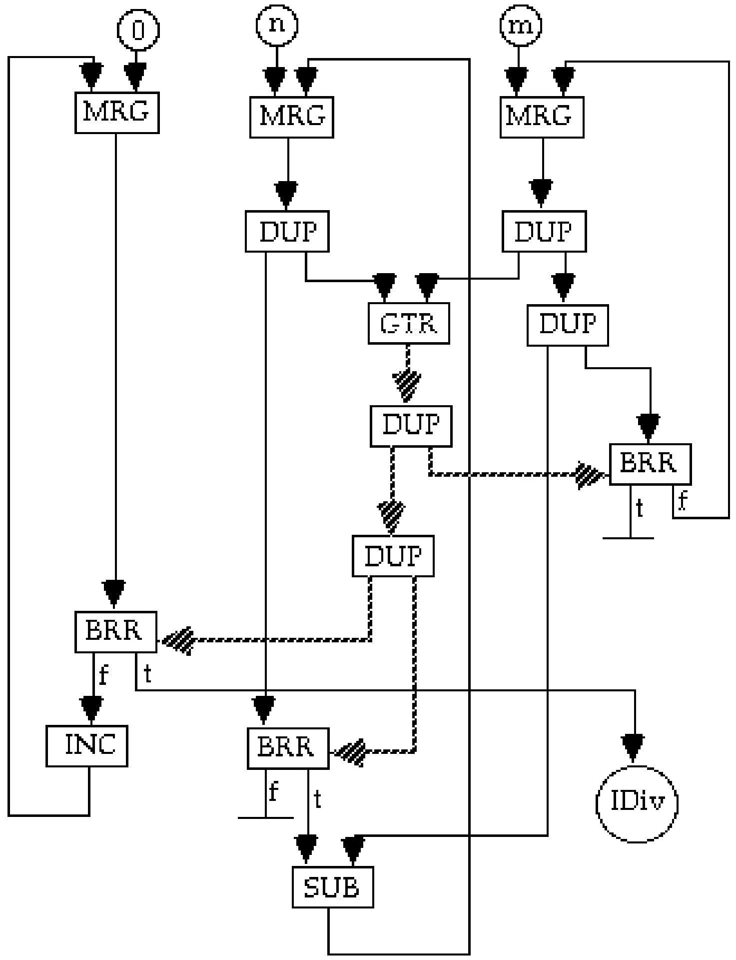

Figure 1.10: grafo dataflow per la divisione intera

1.8.2 Massimo Comune Divisore

Assegnata la definizione ricorsiva della funzione che esegue il *massimo comune divisore* di due numeri interi

$$GCD(m,n) \equiv \textbf{if } (m=n) \textbf{ then } m \textbf{ else}$$
$$\textbf{if } (m<n) \textbf{ then } GCD(m, n-m) \textbf{ else } GCD(m-n, n)$$

la versione in forma iterativa, utilizzando un costrutto di tipo while, é quella che compare di seguito

```
while ( m ≠ n ) do
   if ( m ≤ n ) then
      n := n - m else
      m := m - n;
GCD := m;
```

```
1: if ( m = n ) then goto 2;
      if ( m < n ) then n := n - m
         else m := m - n;
   goto 1;
2: GCD := m;
```

A questo punto é immediato realizzare l'algoritmo che la valuti col maggior livello di parallelismo possibile.

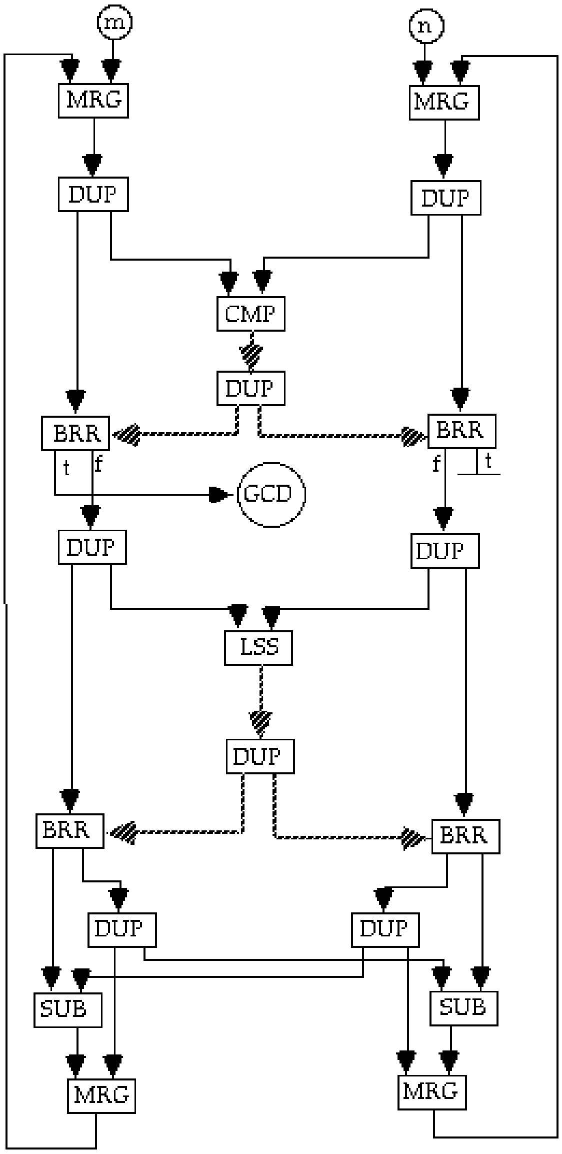

Figure 1.11: grafo dataflow del m.c.d.