Indice

Introduzione **1**

1 Simulazioni di fisica e calcolo parallelo **5**

1.1 Problemi di fisica 5

1.1.1 QCD 7

1.1.2 Vetri di spin 9

1.2 Architetture di calcolatori paralleli 10

1.2.1 Prestazioni di un calcolatore parallelo 12

1.2.2 Classificazione dei calcolatori paralleli 15

1.2.3 Cenni sull'architettura dei processori 18

2 Compilatori **22**

2.1 Linguaggi formali 22

2.1.1 Grammatiche 27
2.1.2 Classificazione di Chomsky 30
2.2 Compilatori 33
2.2.1 Fasi della compilazione 34
2.2.2 Analisi lessicale, sintattica e semantica 36
2.2.3 Rappresentazioni intermedie 38
2.2.4 Generazione del codice oggetto 41
2.3 Ottimizzazioni 41
2.3.1 Trasformazioni che lasciano invariate le funzioni 42
2.3.2 Eliminazione delle sottoespressioni comuni 45
2.3.3 *Copy-propagation* 48
2.3.4 Eliminazione del "codice morto" 49
2.3.5 Ottimizzazione sui cicli 50
2.3.6 Spostamento del codice 51
2.3.7 Eliminazione delle variabili di induzione e riduzione in forza 52

2.3.8 *Loop unrolling* 54

3 Il calcolatore parallelo APEmille 57

3.1 Il progetto APE 57

3.2 L'architettura di APEmille 60

3.3 Architettura e funzionalità dei dispositivi VLSI 66

3.3.1 Il controllore Tarzan 66

3.3.2 Il processore aritmetico Jane 73

3.3.3 Il dispositivo di comunicazione Cheetah 78

3.4 Il software di sistema 83

3.4.1 Il simulatore 83

3.4.2 La catena di compilazione 85

3.4.3 Il sistema operativo 93

4 Il Compilatore TAOmille 96

4.1 L'Ambiente di sviluppo 97

4.1.1 Il sistema operativo Linux 97

4.1.2 Il parser dinamico Z_Z 98

4.1.3 Il linguaggio C++ 103

4.2 Il linguaggio TAOmille 104

4.2.1 TAO 104

4.2.2 TAOmille: l'estensione di TAO per APEmille 114

4.3 L'assembly di APEmille 126

4.3.1 La struttura sintattica 128

4.3.2 Tipi di istruzioni 133

4.4 La struttura del compilatore 134

4.5 Il front-end 138

4.6 Pico 142

4.6.1 Alberi e grafi 143

4.6.2 Il formato del codice intermedio 146

4.6.3 Algoritmo per la costruzione del DAG 150

4.6.4 Algoritmo per la visita del DAG 161

4.7 Il back-end 165

4.7.1 Il generatore di codice assembly 166

4.7.2 L'algoritmo "normalizzatore" 173

4.8 Algoritmi per la gestione dei tipi strutturati del linguaggio . . . 177

Conclusioni **183**

Appendici **185**

A Istruzioni assembly di APEmille **186**

Bibliografia **212**

Introduzione

La simulazione al calcolatore del comportamento dei sistemi fisici ha reso possibile lo studio di problemi che fino ad oggi non erano stati affrontati a causa delle difficoltà analitiche che essi presentano. In particolare questo metodo di indagine è adottato con successo nello studio dei problemi di meccanica statistica la cui caratteristica comune consiste nella necessità di ripetere le stesse sequenze di calcolo sui vari gradi di libertà del sistema fisico.

Questo lavoro riguarda la progettazione e la realizzazione del compilatore-ottimizzatore per il calcolatore APEmille ed è stato svolto nell'ambito del progetto APE dell'Istituto Nazionale di Fisica Nucleare, il cui obiettivo è la costruzione di un calcolatore parallelo con potenza di calcolo dell'ordine del Teraflop dedicato alla simulazione dei sistemi fisici.

Nelle applicazioni di calcolo intensivo è essenziale implementare gli algoritmi di simulazione nella maniera più efficiente possibile allo scopo di minimizzare i tempi di esecuzione dei programmi. Con un compilatore "tradizionale" la produzione di codice che soddisfi questo requisito di efficienza avviene, per un calcolatore parallelo come APEmille, in due passi distinti. In primo luogo, è necessario adattare l'algoritmo di simulazione e le sue strutture dati all'architettura parallela della macchina. Questa è una fase critica, dal momento che non tutti gli algoritmi si prestano ad essere realizzati in maniera efficiente su un calcolatore con architettura SIMD. In secondo luogo, nella scrittura del programma di alto livello bisogna utilizzare degli accorgimenti per forzare il compilatore a generare codice *assembly* ottimizzato, la cui esecuzione sfrutti al massimo le risorse di calcolo fornite dalla macchina.

Il primo di questi passi, analogamente al problema della scelta dell'algoritmo di simulazione di minore complessità possibile, deve essere lasciato al programmatore, dal momento che non esiste un metodo secondo cui svolgere questo compito in maniera automatica.

Il secondo passo può invece essere affrontato da un compilatore-ottimizzatore che effettui una serie di controlli sul codice sorgente e adotti automaticamente gli accorgimenti necessari alla produzione di codice *assembly* ottimizzato.

TAOmille è il linguaggio di programmazione di alto livello di APEmille. La struttura del compilatore TAOmille è formata da tre moduli *software*: il *front-end*, *Pico* ed il *back-end*. Il *front-end* analizza le istruzioni del programma e richiama le funzioni del modulo *Pico* che si occupano di costruire una rappresentazione intermedia del codice sorgente. Questa è data da un diagramma di flusso i cui nodi, che corrispondono a sequenze di istruzioni TAOmille, sono dei *grafi orientati aciclici* o DAG (*directed acyclic graph*). Su questa rappresentazione intermedia vengono operate delle ottimizzazioni *hardware independent*.

Il *back-end*, infine, è il modulo che genera, a partire dalla rappresentazione intermedia, il codice *assembly* ottimizzato per APEmille.

Il contributo originale di questo lavoro è consistito nella progettazione della struttura generale del compilatore e nella realizzazione dei moduli *Pico* e *back-end*, oltre che dei comandi del *front-end* necessari all'interfacciamento tra questo e *Pico*. L'attività di progettazione, preceduta dallo studio delle tecniche e degli algoritmi attualmente in uso nella realizzazione dei compilatori, ha riguardato in modo particolare la scelta del tipo della rappresentazione intermedia del codice sorgente.

La tesi si articola in quattro capitoli:

- Nel primo capitolo, dopo avere trattato due esempi di problemi di fisica il cui studio può essere condotto attraverso la simulazione su calcolatori paralleli, vengono trattate le architetture di tali calcolatori, con riferimento alle loro prestazioni e alla loro classificazione; infine vengono esposte alcune tecniche realizzative per i microprocessori.

- Il secondo capitolo è diviso in tre parti. Nella prima parte sono formalizzati i concetti di linguaggio e grammatica che verranno usati estensivamente nel quarto capitolo. Nella seconda parte si trattano i compilatori fornendo una descrizione generale delle fasi del processo di compilazione e delle tecniche realizzative. Nella terza parte viene fatta una rassegna delle principali tecniche di ottimizzazione che possono essere utilizzate per migliorare la qualità del codice prodotto da un compilatore.

- Nel terzo capitolo si descrivono l'architettura ed il software del calcolatore APEmille.

- Il quarto capitolo è quello centrale della tesi. In esso viene descritto il compilatore TAOmille con particolare riguardo al lavoro originale svolto. Il capitolo è suddiviso in otto sezioni e, partendo dalla descrizione del linguaggio sorgente TAO, del linguaggio *assembly* APEmille e dell'ambiente di sviluppo, si descrivono poi, in sezioni dedicate, il *front-end*, *Pico*, il *back-end* ed alcuni algoritmi per la gestione dei tipi strutturati del linguaggio.

- In appendice è riportato l'insieme delle istruzioni *assembly* di APEmille.

Capitolo 1

Simulazioni di fisica e calcolo parallelo

1.1 Problemi di fisica

Gli enormi passi avanti fatti nella costruzione di computer ad altissime prestazioni hanno reso possibile lo studio numerico di problemi di fisica teorica che fino ad oggi erano considerati "irrisolvibili". Si fa riferimento in particolare ai problemi di meccanica statistica la cui caratteristica comune è la necessità di ripetere le stesse sequenze di calcolo sui vari gradi di libertà del sistema fisico. Gli esempi discussi nei successivi paragrafi sono tratti da problemi reali il cui studio è stato affrontato grazie alla disponibilità di risorse di calcolo parallelo di tipo APE (1, 100, mille). Questa discussione è rivolta ad evidenziare quali debbano essere le caratteristiche *software* e *hardware* adeguate per queste investigazioni.

Il tipico problema che bisogna affrontare in meccanica statistica è il calcolo dei valori medi delle osservabili, definiti come

$$< A(C) >= \int dCA(C)p(C). \tag{1.1}$$

dove $A(C)$ rappresenta la generica osservabile (energia, magnetizzazione, propagatore fermionico), C è una possibile configurazione che il sistema fisico può assumere in uno spazio di configurazioni che d'ora in avanti denoteremo con Γ, e $p(C)$ è la funzione peso con cui fare la media. La caratteristica peculiare in questo calcolo risiede nel fatto che lo spazio delle configurazioni è vastissimo (si pensi ad un sistema di N spin, dove il numero di configurazioni è 2^N) e che un calcolo numerico esatto, (cioè portato avanti enumerando tutte le possibili configurazioni) non è attuabile. Si ricorre allora ad una stima Montecarlo della (1.1). Questa stima consiste nel valutare la (1.1) a partire da un campione limitato di C, cioè estraendo a caso K configurazioni $\{C_i\}$ da Γ e calcolando la quantità:

$$< A(C) >= \frac{1}{N} \sum_{i=1}^{K} A(C_i) p(C_i) \ .$$

Affinché una stima ottenuta nel modo precedente sia significativa, occorrerebbe estrarre un numero di configurazioni proporzionali a quelle in Γ, soprattutto perché il fattore peso $p(C)$ è una funziona apprezzabilmente diversa da zero soltanto su un sottospazio molto piccolo di Γ. Per ovviare a questo inconveniente si effettua un campionamento ad importanza che consiste nell'estrarre le K configurazioni non in modo casuale, bensì secondo un peso di probabilità fissato a priori che al fine di minimizzare l'errore sulla stima viene preso proprio pari a $p(C)$. In questo modo la stima può essere scritta semplicemente come:

$$< A(C) >= \frac{1}{N} \sum_{i=1}^{K} A(C_i) \ .$$

La maggior difficoltà è rappresentata ora dalla generazione delle configurazioni C secondo il peso di probabilità $p(C)$. L'algoritmo utilizzato per questa generazione non è unico e dipende dal particolare problema fisico che si sta studiando. In tutti i casi quanto più numeroso è il campione delle $\{C_i\}$ estratte, tanto migliore sarà la stima di $< A(C) >$, cioè tanto minore sarà l'errore ad essa associato. Per tutti questi problemi è quindi cruciale la ripetizione di un certo algoritmo di calcolo per la generazione di un campione quanto più vasto possibile di $\{C_i\}$.

Nei successivi paragrafi sarà trattato brevemente il caso degli Ising Spin-Glass e quello della QCD.

1.1.1 QCD

La QCD (CromoDinamicaQuantistica) è la teoria che descrive le interazioni forti alla base della aggregazione nucleare e sub-nucleare della materia. Sebbene questa teoria abbia avuto numerose verifiche sperimentali sopratutto nel campo delle alte energie, dove è possibile fare previsioni analitiche con metodi perturbativi (si ricorda che la costante di accoppiamento per questa teoria è funzione descrescente dell'energia del processo fisico), uno studio non perturbativo è necessario per analizzare proprietà come lo spettro di massa della teoria e in genere questioni riguardanti il confinamento dei quark negli adroni.

L'approccio non perturbativo più promettente è quello proveniente dalla teoria "discretizzata" secondo la formulazione tramite integrale funzionale.

Il processo di discretizzazione consiste nel suddividere lo spazio-tempo in un insieme discreto di punti, che costituiscono un reticolo, abitualmente ipercubico. Le variabili dinamiche della teoria sono definite sui punti del reticolo nel caso dei campi fermionici, e sui legami (*link*) per i campi gluonici. Le osservabili fisiche sono descritte da valori medi del tipo (1.1) in cui la C è una generica configurazione di campi gluonici e fermionici, l'osservabile A è in genere il propagatore fermionico e il fattore peso è quello generato dall'azione discretizzata.

La maggiore difficoltà per la teoria interagente è la generazione di configurazioni di campo che siano distribuite con un peso del tipo e^{-S} dove S è l'azione della teoria. Infatti si può mostrare che per la teoria interagente il calcolo dell'azione S prevede il calcolo dell'inversa di una matrice che ha ordine pari al volume dello spazio-tempo. Questo calcolo richiede tempi molto elevati e peggiora quanto più i parametri della teoria vengono avvicinati ai valori fisici che loro competono. Per essere più specifici [1] si pensi ad un sistema definito su un reticolo di dimensioni $16^3 \cdot 32$. L'ordine della matrice di cui si cerca l'inversa è circa 10^5. Il numero di volte in cui è necessario riprodurre questo calcolo per avere un numero di configurazioni ragionevole è anch'esso dell'ordine di 10^5, con parametri fisici non realistici ma numericamente favorevoli. In sintesi una tipica simulazione in queste condizioni richiede tempi dell'ordine di qualche mese con il computer APE100-Torre (25Gigaflops). Il guadagno di un'ordine di grandezza per quanto riguarda la potenza massima di calcolo, permetterebbe di condurre delle simulazioni con valori dei parametri più realistiche, ed il conseguente studio di numerose proprietà della teoria in regime non perturbativo.

1.1.2 Vetri di spin

Un Ising Spin-Glass [2] è un sistema fisico definito su un reticolo d-dimensionale. In ogni punto del reticolo è definita una variabile s_i che può assumere valori ± 1. Per ogni coppia di siti i, j è definita una variabile J_{ij}, detta accoppiamento, il cui valore è estratto in modo casuale da una distribuzione di probabilità definita a priori che in genere viene presa di tipo gaussiano o discreto, cioè $\pm J$ con probabilità $\frac{1}{2}$. La hamiltoniana del sistema è scritta per ogni particolare realizzazione delle J_{ij} come:

$$H(J) = -\sum_{i,j=1}^{Volume} s_i J_{ij} s_j \ .$$

Per questo tipo di problemi si è in genere interessati alle proprietà del sistema mediate oltre che secondo la (1.1), anche rispetto alle possibili realizzazioni delle J_{ij}. Il parallelismo di APE può quindi essere sfruttato in due modi differenti. Da un lato è possibile studiare sistemi con reticoli relativamente piccoli, in cui la memoria richiesta è minore di quella del singolo nodo, e ciascun processore simula una diversa realizzazione delle J_{ij}. Dall'altro è possibile suddividere un reticolo molto grande sui vari nodi della macchina come si è soliti fare per altri problemi tipo la QCD o la meccanica dei fluidi. Si è visto che il punto debole della macchina APE100 per questo tipo di applicazioni è

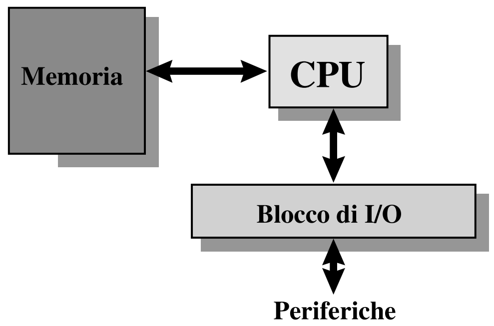


Figura 1.1: Architettura di Von Neumann di un elaboratore

dato dall'impossibilità di implementare algoritmi standard di tipo multispin per la totale assenza di aritmetica intera locale. Con APEmille si risolverà questo problema con l'uso degli interi locali (cfr. § 4.2.2).

1.2 Architetture di calcolatori paralleli

Un calcolatore parallelo prevede la divisione dei compiti ad esso assegnati nel programma tra più unità di elaborazione che provvedono ad eseguirli in maniera concorrente. Questo approccio contrasta con quello derivante dalla classica architettura di Von Neumann, che è un buon modello per descrivere i calcolatori sequenziali.

Essa prevede 4 elementi funzionali mutuamente interagenti: un controllore

(CU), una unità di calcolo logico-aritmetico (ALU), la memoria (unica per i dati ed il programma) e i dispositivi di I/O. Generalmente, come mostrato in figura 1.1, il controllore e l'unità di calcolo sono integrate in un unico dispositivo, detto CPU (*Central Processing Unit*). La CPU legge le istruzioni contenute in memoria, le decodifica e le esegue. In questo schema di funzionamento del sistema si considerano operazioni (accessi in memoria, operazioni di calcolo etc.) strettamente sequenziali.

L'uso di più processori concorrenti non porta automaticamente ad un incremento significativo delle prestazioni di un elaboratore. L'opportunità di utilizzare un certo calcolatore parallelo dipende in modo determinante dalla natura della applicazione che si vuole realizzare. Inoltre bisogna considerare che la programmazione parallela è in genere più complessa di quella sequenziale e la scrittura ed il *debugging* del codice richiedono tempi più lunghi. Un altro fattore determinante nella scelta dell'uso di un calcolatore parallelo è quello economico, dati i costi elevati che, a seconda del caso, sono necessari a produrre, acquistare o affittare una macchina parallela.

1.2.1 Prestazioni di un calcolatore parallelo

Si consideri una CPU che esegua un certo programma in un tempo T_1. Sia T_N il tempo necessario ad un calcolatore parallelo dove la stessa CPU sia replicata N volte ad eseguire lo stesso programma, o più esattamente lo stesso compito assegnato dal programma, dal momento che l'implementazione di questo sarà in genere piuttosto differente nei due casi. Si definiscono [3] il fattore di *speed-up* come indice di incremento delle prestazioni passando da una architettura sequenziale ad una di tipo parallelo il rapporto:

$$S_N = \frac{T_1}{T_N} \tag{1.2}$$

e l'efficienza:

$$E = \frac{S_N}{N} \cdot 100 \tag{1.3}$$

che indica la percentuale di tempo in cui i processori sono occupati nella elaborazione del programma. Il migliore risultato si otterrebbe riducendo i tempi di elaborazione di un fattore N ma, per una serie di cause che elencheremo nel seguito, si ha in generale:

$$S_N < N \tag{1.4}$$

Le cause che limitano il valore di S_N sono:

- Il tempo di latenza T_C dovuto alla comunicazione tra i processori, che va a sommarsi al tempo di esecuzione del programma (*communication overhead*). La durata della elaborazione parallela diventa quindi:

  $$T_N = \frac{T_1}{N} + T_C \tag{1.5}$$

  che comporta un fattore di *speed-up*:

  $$S_N = \frac{T_1}{\frac{T_1}{N} + T_C} < N \tag{1.6}$$

L'*overhead* per le comunicazioni diventa quindi trascurabile se $\frac{T_1}{N} >> T_C$, ovvero, con un tempo di latenza fissato, per un numero di processori:

$$N << \frac{T_1}{T_C} \tag{1.7}$$

Da questo si vede che aumentare arbitrariamente il numero di processori non aumenta automaticamente l'efficienza del calcolatore parallelo, dal momento che la latenza introdotta dalle comunicazioni può diventare critica.

- Il grado di parallelismo di un algoritmo è generalmente inferiore al 100%. Una parte del codice del programma che implementa l'algoritmo è quindi seriale e l'uso di un calcolatore parallelo non ne riduce i tempi di esecuzione. Assumendo nullo l'*overhead* per le comunicazioni, il tempo di esecuzione di un programma su un calcolatore parallelo può essere espresso come somma di un tempo T_P, necessario ad eseguire la frazione f di codice parallelizzabile, e di un tempo T_s relativo all'esecuzione della frazione seriale $(1-f)$. Si ha quindi:

  $T_P = f \cdot \frac{T_1}{N} \qquad T_s = (1-f) \cdot T_1$

$$S_N = \frac{T_1}{T_s + T_P} = \frac{N}{(1-f) \cdot N + f} \tag{1.8}$$

L' equazione 1.8 è nota come *legge di Amdahl*. In figura 1.2 è mostrato l'andamento del fattore di *speed-up* in funzione di f e del numero di processori N.

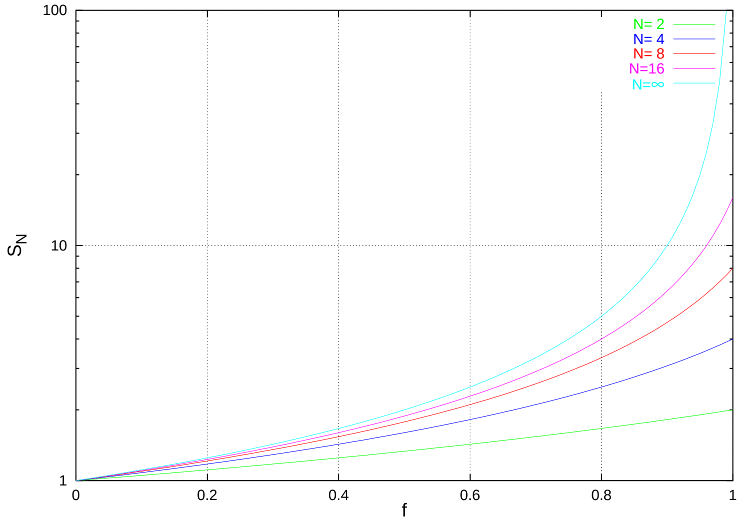


Figura 1.2: Fattore di *speed-up*

Dal grafico si nota come un miglioramento delle prestazioni, rispetto al caso seriale, abbia comunque luogo, ma affinché questo sia sostanziale è necessario che la frazione f di codice parallelizzabile sia significativa. Dalla legge di Amdhal risulta infatti che, con un numero infinito di processori, S_N è limitato superiormente da $\frac{1}{1-f}$. Inoltre questa è una stima ottimistica, dal momento che nel calcolo si è trascurato l'*overhead* introdotto dalle comunicazioni.

- L'attività dei processori richiede operazioni di sincronizzazione. Ciò introduce tempi "morti" in cui alcuni processori non svolgono elaborazioni.

- Durante l'esecuzione del programma si può esaurire lo spazio di memoria riservato ai dati. In questo caso verranno utilizzate pesantemente le memorie di massa come memorie virtuali. Questo evento, che può ovviamente verificarsi anche in un elaboratore seriale, ha in genere nel caso di un calcolatore parallelo delle conseguenze maggiormente dannose ai fini delle prestazioni (si pensi ad una architettura con memoria dati replicata per ogni processore). Gli sforzi mirati a ridurre questo genere di degrado delle prestazioni consistono nella implementazione di canali di comunicazione a larga banda passante tra le memorie dati e i dispositivi di memoria di massa.

1.2.2 Classificazione dei calcolatori paralleli

Flynn (1966) ha effettuato una classificazione delle architetture dei calcolatori paralleli in base al tipo di flusso di dati ed istruzioni in esse presenti [3]. In questa ottica l'elaboratore a singolo processore viene definito SISD(*Single Instruction stream - Single Data stream*). Si parla invece di calcolatore parallelo di tipo SIMD (*Single Instruction stream - Multiple Data stream*) o di *array computer* riferendosi ad una architettura basata su n processori aritmetici (CE1...CEn), ognuno corredato della propria memoria dati (Mem1...Mem$n$), come mostrato in figura 1.3. Il controllore CU gestisce il flusso del programma ed invia la stessa istruzione (letta dalla memoria programma) a ciascuno dei processori CE che provvedono ad eseguirla usando i dati della propria memoria locale. Una rete di interconnessione permette lo scambio dei dati tra i processori aritmetici.

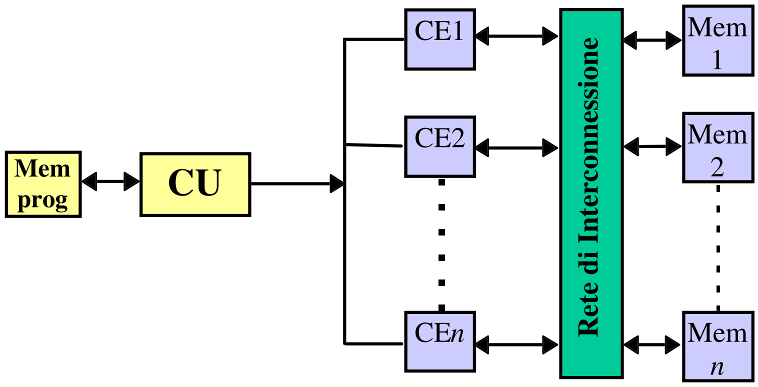


Figura 1.3: Architettura SIMD

L'architettura di tipo MIMD (*Multiple Instruction stream - Multiple Data stream*) prevede m processori (P1...Pm) e n blocchi di memoria (M1...Mn), come mostrato in figura 1.4. Ciascun processore può eseguire una istruzione differente su flussi di dati differenti. Questo tipo di architettura è più flessibile rispetto alle applicazioni che possono esservi implementate in modo efficiente (*general purpose*), ma di contro la realizzazione di un calcolatore MIMD presenta notevoli difficoltà che derivano essenzialmente dalla necessità di sincronizzare il funzionamento dei processori durante l'esecuzione dei programmi.

La *granularità* di una architettura parallela fornisce un altro parametro di classificazione. Si parla di macchine *coarse grained* quando esse sono costituite da un numero contenuto di processori di elevata potenza di calcolo e di mac-

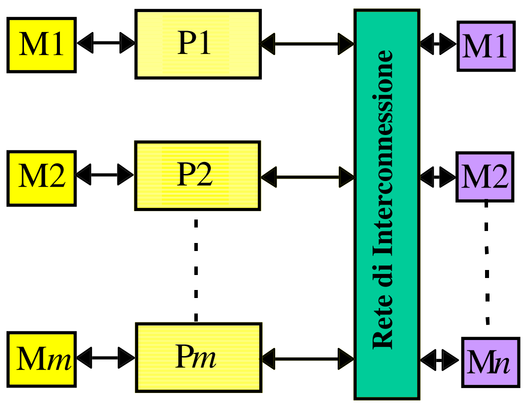


Figura 1.4: Architettura MIMD

chine *fine grained* quando si ha un numero elevato di processori relativamente semplici. L'architettura di un calcolatore parallelo si dice inoltre *scalabile* se essa prevede più configurazioni possibili con un diverso numero di processori. La topologia delle connessioni tra i processori deve essere, per macchine scalabili, di tipo modulare. Diversi tipi di connessioni possono essere considerati, a partire da quella diretta mostrata in figura 1.5 dove ogni processore comunica con ogni altro.

Una topologia di questo genere è molto complicata da gestire, richiede infatti un numero di collegamenti pari a $\frac{p}{2} \cdot (p-1)$ per un calcolatore con p processori. Solitamente si adottano altre topologie per i collegamenti, quali reticoli cubici e alberi.

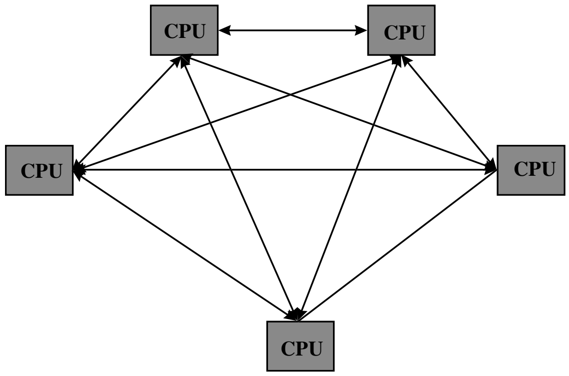


Figura 1.5: Connessione diretta

1.2.3 Cenni sull'architettura dei processori

In questo paragrafo vengono descritte alcune tecniche, a cui si farà riferimento negli altri capitoli, utilizzate nella realizzazione di microprocessori ad alte prestazioni.

- *Microprocessori RISC*

  Uno studio condotto dal Alexander e Wortman (1975) ha mostrato che su un sistema IBM con compilatore XLP circa l'80% del codice macchina eseguibile era costituito da un insieme di 10 istruzioni ed il 99% del codice da un insieme di 30 istruzioni. Questo dato è alla base dell'approccio RISC(*Reduced Instruction Set Computer*) per il progetto di microprocessori che dispongano di un numero limitato di istruzioni di basso livello, ma che siano in grado di eseguirle velocemente (nel caso ideale in un solo ciclo macchina). Il tempo di esecuzione di un programma può essere espresso dall'equazione:

$$T_{ex} = N_{ist} \cdot N_{cicl/ist} \cdot T_{clock} \tag{1.9}$$

Dove N_{ist} è il numero di istruzioni macchina del programma, $N_{cicl/ist}$ è il numero di cicli macchina necessari ad eseguire una istruzione (nell'ipotesi semplificatrice che sia lo stesso per tutte) e T_{clock} è il periodo del ciclo macchina.

Nel progetto di un processore RISC si cerca di ridurre T_{ex} minimizzando $N_{cicl/ist}$, in quello di un processore CISC(*Complex Intruction Set Computer*) si cerca invece di ridurre N_{ist}.

- *Tecniche di Pipelining*

  L'obiettivo di eseguire una istruzione in un ciclo macchina è reso possibile dall'uso di tecniche di *pipelining* nel progetto del processore: l'hardware necessario per eseguire una certa operazione viene suddiviso in più stadi (di *pipeline*) ciascuno dei quali elabora i dati prodotti dal precedente e passa i propri risultati al successivo, come mostrato in figura 1.6.

  La profondità n della *pipe* dipende dalla complessità dei singoli stadi, che deve essere tale da permettere a ciascuno di essi di completare il proprio compito in un ciclo macchina. Ogni stadio può così accettare

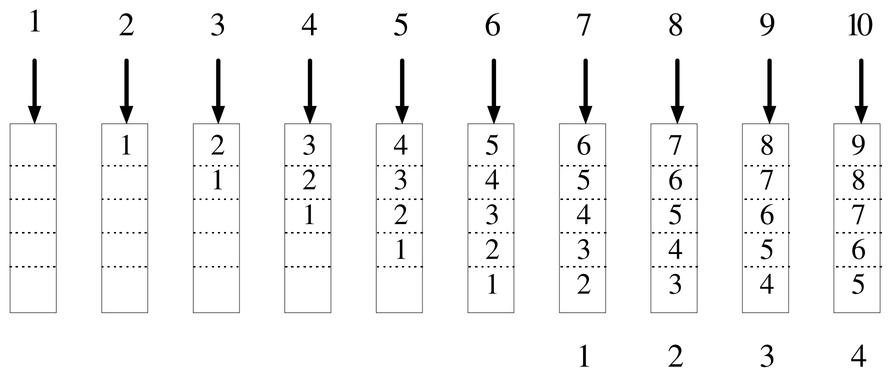


Figura 1.6: Schema di *pipeline* a 5 stadi

in ingresso una parola per ciclo. Dopo una latenza iniziale necessaria a riempire la *pipeline* il processore è in grado di eseguire una istruzione per ciclo macchina, almeno fino a quando una istruzione di salto non causa lo svuotamento della *pipeline*.

- *Microprocessori VLIW*

  Nei processori convenzionali alla fase di *fetch* (prelievo) dell'istruzione di microcodice segue la fase di decodifica della stessa. La decodifica dell'istruzione genera i segnali di controllo per i dispositivi interni al processore per la successiva fase di esecuzione. Nei processori VLIW(*Very Long Instruction Word*) la fase di decodifica non avviene, dal momento che la parola di microcodice (*control word*) prevede tanti campi di bit quanti sono i dispositivi da controllare; ogni campo controlla direttamente il dispositivo ad esso associato, come mostrato in figura 1.7.

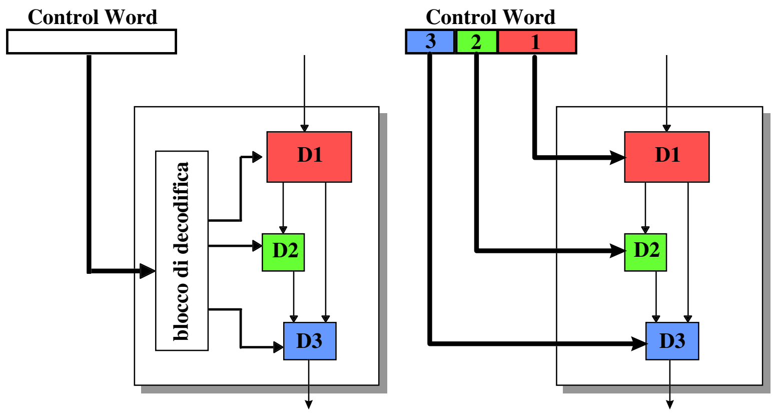


Figura 1.7: Confronto tra un processore tradizionale ed uno VLIW

Dal momento che le istruzioni non sono codificate, le parole di microcodice hanno una lunghezza, generalmente la stessa per tutte le istruzioni, dell'ordine delle centinaia di bit.

Capitolo 2

Compilatori

In questo capitolo vengono riportate alcune nozioni relative ai linguaggi formali (con particolare riferimento ai linguaggi di programmazione), ai compilatori ed al modo in cui questi possono effettuare delle ottimizzazioni, nel senso che verrà spiegato, sul codice prodotto. Molti dei concetti e dei termini qui riportati saranno utilizzati nel capitolo 4 descrivendo il linguaggio TAOmille ed il suo compilatore.

2.1 Linguaggi formali

Intuitivamente un linguaggio può essere visto come un insieme di parole (*stringhe*) ottenute concatenando, secondo le regole espresse dalla sintassi del linguaggio, alcuni simboli tratti da un insieme prefissato detto *alfabeto* del linguaggio.

In maniera più formale si possono definire un *alfabeto* come un insieme non vuoto e finito di elementi chiamati *simboli*, ed una *stringa* (o frase) su un alfabeto A come giustapposizione (o concatenazione) di un numero finito di simboli di A; tale numero può essere zero, nel qual caso la stringa si dirà vuota e sarà denotata con il simbolo ϵ. Si indica con A^* l'insieme di tutte le stringhe costruite sull'alfabeto A e con A^+ l'insieme $A^* - \{\epsilon\}$ delle stringhe di lunghezza non nulla sull'alfabeto A. Sia A^* che A^+ sono insiemi infiniti aventi la potenza del discreto.

A questo punto è possibile dare una definizione di *linguaggio*.
Un *linguaggio* L su un alfabeto A è un insieme di stringhe su A, ovvero

$$L \subseteq A^* \tag{2.1}$$

La *concatenazione* $L_1 \cdot L_2$ di due linguaggi L_1 ed L_2 è definita come il linguaggio le cui stringhe sono date dalla concatenazione di una stringa di L_1 con una stringa di L_2:

$$L_1 \cdot L_2 \ = \ \{xy \mid x \in L_1 \wedge y \in L_2\} \tag{2.2}$$

La *potenza* $n - esima$ $(n \geq 0)$ di un linguaggio L, indicata con L^n, è definita induttivamente come segue:

$$L^0 \ = \ \{\epsilon\}; \quad L^{n+1} \ = \ L \cdot L^n \tag{2.3}$$

Una volta definito cosa sia un linguaggio si pone il problema di cosa significhi descriverlo e di come esso possa essere descritto. In prima approssimazione si può affermare che descrivere un linguaggio significa fornire un criterio che consenta di stabilire se una frase appartiene o meno ad esso. Circa la modalità della descrizione consideriamo il caso più semplice che si possa presentare, quello cioè in cui un linguaggio L consiste in un numero finito (e non eccessivamente grande) di frasi. In questa situazione il modo più semplice di descrivere il linguaggio consiste nell'elencare tutte le frasi che gli appartengono (descrizione per *enumerazione*). Ad esempio si può considerare il linguaggio L costruito sull'alfabeto $A = \{a, b\}$ costituito dalle frasi w con $|w| \leq 2$, ovvero dalle stringhe lunghe al massimo due simboli:

$$L \;=\; \{\epsilon,\, a,\, b,\, aa,\, ab,\, bb,\, ba\}$$

Un problema notevole nasce invece nella descrizione dei linguaggi costituiti da un numero infinito di frasi. Gli strumenti attualmente disponibili per risolvere tale problema sono fondamentalmente tre:

1. *Descrizione insiemistica*: si stabilisce quale proprietà P deve essere soddisfatta da w per appartenere ad certo linguaggio L:

   $$L = \{w \mid P(w)\}$$

   Un esempio di questa descrizione può essere quella del linguaggio L sull'alfabeto $\{a, b\}$ costituito dalle stringhe composte da una successione di simboli a concatenata ad una successione di simboli b avente la stessa lunghezza:

   $$L = \{a^n b^n \mid n \geq 0\} = \{\epsilon, ab, aabb, aaabbb, ...\}$$

2. *Descrizione generativa*: si descrive un dispositivo che, quando è messo in funzione, produce le frasi valide per un certo linguaggio.

3. *Descrizione riconoscitiva*: si descrive un dispositivo che riceve in ingresso una stringa costruita con simboli di un certo alfabeto e la elabora. Il risultato della elaborazione è un messaggio di avvenuto riconoscimento o meno.

Il secondo metodo si presenta come un insieme finito di *regole* che consentono di produrre solo le frasi che appartengono ad un certo linguaggio. A questo insieme di regole viene dato il nome di *grammatica*. Ad esempio nella lingua italiana una frase ha spesso una struttura di questo tipo:
- un *soggetto* seguito da un *predicato*;
- un *soggetto* a sua volta può essere o un *articolo* seguito da un *aggettivo* seguito da un *nome* o semplicemente un *articolo* seguito da un *nome*;
- un *predicato* è un *verbo* oppure è un *verbo* seguito da un *soggetto*.

Soggetto, *articolo*, *predicato* etc. sono le parti del discorso dette anche **categorie sintattiche** (o **simboli non terminali** o **sintagmi**). Usando la convenzione di racchiudere tra parentesi angolari ogni categoria sintattica si può scrivere:

1. $< frase > \rightarrow < soggetto >< predicato >$

Con il significato che si può riscrivere quello che sta a sinistra della freccia con quello che sta a destra.

2. $< soggetto > \rightarrow < articolo >< aggettivo >< nome > | < articolo >< nome >$

Dove il simbolo "|" indica un'alternativa tra più riscritture possibili.

3. $< predicato > \rightarrow < verbo > | < verbo >< soggetto >$

Per completare la definizione della $< frase >$ bisogna specificare quali parole del linguaggio (in questo caso l'italiano) si possono leggitimamente sostituire alle categorie sintattiche. Alla fine del processo di generazione avremo solo frasi valide del linguaggio costituite dai simboli dell'alfabeto; per questo motivo i simboli dell'alfabeto vengono detti **simboli terminali**, dal momento che non si possono più riscrivere. Ad esempio avremo:

4. $< articolo > \rightarrow\ il \mid lo \mid la \mid un \mid una \mid uno \mid\ ...$

5. $< aggettivo > \rightarrow\ bello \mid grande \mid\ ...$

6. $< verbo > \rightarrow\ mangia \mid sogna \mid\ ...$

7. $< nome > \rightarrow\ mare \mid pesce \mid strada \mid gatto \mid bimbo \mid\ ...$

Il linguaggio generato da queste regole grammaticali è l'insieme delle frasi che si possono derivare da $< frase >$ applicando un numero finito di volte le regole da 1 a 7. Ad esempio la frase "il bimbo sogna il mare" può essere ottenuta a partire da $< frase >$ applicando nove volte le regole da 1 a 7.

1 $< frase > \Rightarrow < soggetto >< predicato >$

2 $\Rightarrow < articolo >< nome >< predicato >$

4 $\Rightarrow$ *il* $< nome >< predicato >$

7 $\Rightarrow$ *il bimbo* $< predicato >$

3 $\Rightarrow$ *il bimbo* $< verbo >< soggetto >$

6 $\Rightarrow$ *il bimbo sogna* $< soggetto >$

2 $\Rightarrow$ *il bimbo sogna* $< articolo >< nome >$

4 $\Rightarrow$ *il bimbo sogna il* $< nome >$

7 $\Rightarrow$ *il bimbo sogna il mare*

Si è giunti ad una stringa che contiene solo simboli terminali, il processo di generazione è dunque terminato. Si osservi che usando le regole della grammatica di questo esempio si possono generare anche frasi sintatticamente scorrette in italiano, del tipo "la grande gatto mangia una pesce".

2.1.1 Grammatiche

Una **grammatica** è una quadrupla

$$(V_N, V_T, P, S)$$

dove :

- V_N è un insieme *finito* di **simboli non terminali** (detti anche *sintagmi* o *categorie sintattiche*)

- V_T è un insieme *finito* di **simboli terminali** disgiunto da V_N

- P è un insieme *finito* di coppie chiamate **produzioni** del tipo:

$$(\alpha, \beta) \text{ e scritte } \alpha \to \beta$$

$$\text{con} \quad \alpha \in \{V_N \cup V_T\}^* \cdot V_N \cdot \{V_N \cup V_T\}^*$$

$$\beta \in \{V_N \cup V_T\}^*$$

  α è detta *parte* **sinistra** *della produzione* ed è una stringa *non vuota* contenente almeno un non terminale

  β è detta *parte* **destra** *della produzione.*

- $S \in V_N$ è detto **simbolo distinto** o **assioma**.

L'uso delle parentesi angolari $<$ e $>$ per distinguere le categorie sintattiche (*metasimboli*) dai simboli terminali fu proposto da John Backus nel 1959 per descrivere l'*ALGOL 60* nella notazione appunto nota come **Backus-Naur Form (BNF)**.

Allo scopo di snellire la notazione, è abitudine adottare le seguenti convenzioni:

- Le lettere latine maiuscole *A, B, C, D, E, ...* stanno per i simboli non terminali; il simbolo distinto è implicitamente S.

- Le lettere latine minuscole *a, b, c, d, e, f* e in generale, tutti i simboli esclusi dalle presenti convenzioni stanno per i simboli terminali.

- Le ultime lettere minuscole *u, v, w, x, y* stanno per stringhe di terminali.

- Le lettere greche α, β, γ, δ, ... stanno per stringhe composte di terminali e non terminali.

Grazie alle prime due convenzioni è possibile descrivere completamente una grammatica fornendo semplicemente le sue produzioni. Vediamo ora, in modo più rigoroso, come sia possibile usare la grammatica per generare frasi, cominciando a formalizzare il concetto di *derivazione immediata*:

sia $G = (V_N, V_T, P, S)$ una grammatica, se $\alpha \to \beta$ è una produzione di P e $\gamma\ \alpha\ \delta$ è una stringa di $\{V_N \cup V_T\}^+$ allora possiamo *riscrivere* $\gamma\ \alpha\ \delta$ come $\gamma\ \beta\ \delta$ espresso come:

$$\gamma\ \alpha\ \delta \Rightarrow \gamma\ \beta\ \delta$$

dicendo che $\gamma\ \beta\ \delta$ **deriva immediatamente** da $\gamma\alpha\delta$ in G (γ e δ possono anche essere vuote).

Se le produzioni sono identificate con degli indici, l'espressione

$$\phi \underset{i}{\Rightarrow} \psi$$

andrà interpretata come "ψ deriva immediatamente da ϕ applicando la i-esima produzione di G". Con $\phi \overset{k}{\Rightarrow} \psi$ intenderemo invece una successione di esattamente k derivazioni immediate; si ammette $\phi \overset{0}{\Rightarrow} \phi$. In generale faremo riferimento al concetto di **derivazione** in accordo alla seguente definizione: si dice che una stringa ψ **deriva** da una stringa ϕ (e si scrive $\phi \overset{*}{\Rightarrow} \psi$ o anche che una stringa ϕ *genera* una stringa ψ, se esiste una successione di derivazioni immediate

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$$

con $n \geq 0$ tali che

$$\alpha_0 = \phi \quad \text{e} \quad \alpha_n = \psi$$

e per ogni $0 \leq i < n$ si abbia $\alpha_i \Rightarrow \alpha_{i+1}$ con $\alpha_i \in \{V_T \cup V_N\}^+$.

A questo punto possiamo fornire una definizione precisa di frase generata da una G: chiamiamo **forme di frase** (*sentential forms*) di una grammatica $G = (V_N, V_T, P, S)$ le stringhe $\alpha \in (V_N \cup V_T)^*$ derivabili da S, tali cioè che $S \stackrel{*}{\Rightarrow} \alpha$ e $\alpha \in V_T^*$.

Si definisce **linguaggio L generato da una grammatica G** , e si denota con $L(G)$, l'insieme:

$$L(G) = \{w \mid w \in V_T^*, S \stackrel{*}{\Rightarrow} w\}$$

L'insieme $L(G)$ è cioè costituito da tutte le frasi su V_T che si possono ottenere applicando le produzioni di G a partire da S.

2.1.2 Classificazione di Chomsky

Sino ad ora, ci siamo poco interrogati sulla "forma" delle produzione delle grammatiche. Si può pensare che esistano **tipi** *diversi di grammatiche (e quindi di linguaggi) a seconda di certe particolarità della forma delle produzioni.* Sembra cioè possibile riunire in "classi" le grammatiche che presentano produzioni tutte con la stessa struttura.

La classificazione che proponiamo è quella introdotta da Chomsky nel 1957; essa riunisce le grammatiche (e quindi i linguaggi da esse generati) in classi che differiscono per le restrizioni imposte alla *forma delle produzioni*:

Grammatiche e linguaggi di tipo 0

In esse non vi sono restrizioni o vincoli sulla struttuta formale delle produzioni che, in generale, sono del tipo:

$$\alpha \rightarrow \beta.$$

I linguaggi corrispondentemente generati sono molto più generali di quelli di programmazione e degli stessi linguaggi naturali.

Grammatiche e linguaggi di tipo 1

In esse si introduce una *prima restrizione nelle produzioni* che sono del tipo:

$$\gamma\ A\ \delta \quad \rightarrow \quad \gamma\ \beta\ \delta$$

con

$$\begin{aligned} \gamma, \delta &\in (V_N \cup V_T)^* \\ A &\in V_N \\ \beta &\in (V_N \cup V_T)^+ \end{aligned}$$

Grammatiche di questo tipo vengono dette **contestuali** (*context sensitive*) e la ragione di tale denominazione si fa risalire al fatto che *il simbolo* *non terminale A si può riscrivere come β* solo nel **contesto** (γ, δ) cioè solo se A compare con a sinistra γ e con a destra δ.

Questo lascia dedurre che possono esistere forme di frase in cui A può non essere riscritto come β.

Grammatiche e linguaggi di tipo 2

Le grammatiche di questo tipo sono particolarmente interessanti per i nostri scopi, in quanto la maggioranza degli attuali linguaggi di programmazione è generata da grammatiche di questo tipo o riconducibili ad esse.

Nelle grammatiche di tipo 2 le produzioni sono della forma:

$$A \quad \rightarrow \quad \alpha$$

con

$$\begin{aligned} A &\in V_N \\ \alpha &\in (V_N \cup V_T)^+ \end{aligned}$$

Le grammatiche di tipo 2 sono dette anche **libere dal contesto** (*context free*). La denominazione deriva dal fatto che A può *sempre* essere riscritto come α *indipendentemente dal posto in cui A compare in una forma di frase*.

Si noti come le grammatiche di tipo 2 siano una caso particolare di grammatiche di tipo 1: il contesto è $(\varepsilon, \varepsilon)$.

In ogni derivazione immediata si sostituisce esattamente un non terminale con una stringa α parte destra di una produzione $A \to \alpha$.

Grammatiche e linguaggi di tipo 3

Nelle grammatiche di questo tipo, le produzioni hanno la forma:

$$A \to aB \quad \text{oppure} \quad A \to a$$

con $A, B \in V_N$ e $a \in V_T$.

Anche queste grammatiche rivestono notevole importanza per il ruolo che giocano sia nella *teoria dei linguaggi* sia nella *teoria degli automi* e sono un caso particolare di grammatiche libere. I linguaggi generati da queste grammatiche sono chiamati **linguaggi regolari**.

Se indichiamo con L_0, L_1, L_2, L_3 le classi dei linguaggi generati rispettivamente da grammatiche di tipo $0, 1, 2, 3$, si dimostra che la classificazione di Chomsky induce la seguente **gerarchia**:

$$L_0 \supset L_1 \supset L_2 \supset L_3.$$

2.2 Compilatori

Gli elementi di teoria dei linguaggi formali cui abbiamo brevemente accennato nelle pagine precendenti costituiscono le prime basi teoriche su cui si fondano le moderne metodologie di progettazione dei compilatori. In particolare, la definizione formale della sintassi di un linguaggio sorgente mediante una grammatica libera costituisce il primo importante passo nell'implementazione di un linguaggio.

Oggi esistono conoscenze e strumenti che permettono la progettazione dei compilatori in tempi molto più brevi anche rispetto a pochi anni fa. La scrittura di un buon compilatore resta pur sempre un compito abbastanza arduo, al di fuori del comune lavoro di programmazione, specie se la distanza fra il linguaggio sorgente e quello oggetto è notevole.

Un compilatore ha il compito di effettuare la traduzione da un linguaggio di programmazione ad alto livello ad un linguaggio oggetto interpretabile da una determinata macchina.

Il compilatore è esso stesso un programma scritto in un linguaggio di programmazione e può essere eseguito, in linea di principio, anche su macchina diversa da quella sulla quale saranno eseguiti i programmi tradotti (in questo caso si parla di *cross-compiler*).

2.2.1 Fasi della compilazione

Il processo di compilazione può essere suddiviso in diversi sottoprocessi logicamente distinti detti *fasi della compilazione*; è difficile fornire una loro classificazione valida per tutti i compilatori dal momento che questi ultimi si possono differenziare notevolmente e possono prevedere fasi particolari dipendenti dal linguaggio o da altri fattori.

Tuttavia, a grandi linee, è possibile fare riferimento allo schema di massima di figura 2.1.

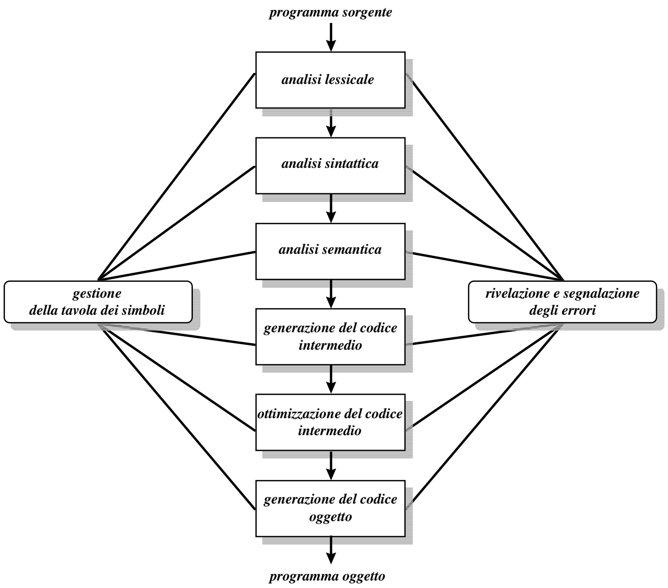


Figura 2.1: Fasi della compilazione

Il testo sorgente subisce tre tipi di analisi successive (lessicale, sintattica, semantica) il cui compito è quello di verificare il rispetto delle specifiche sintattiche e semantiche del linguaggio; altro compito, non meno importante, è quello di raccogliere tutte le informazioni necessarie al successivo processo di traduzione.

Queste sono costituite dall'albero di derivazione del programma e dalla cosidetta *tabella dei simboli* che si compone dei simboli scelti dal programmatore (identificatori, costanti, etichette, ecc.) presenti nel programma. A ciascun simbolo viene associata una serie di attributi che ne specificano il risultato:

genere (variabile, costante, parametro formale, procedura, ecc.), tipo (*integer, real, char*, ecc.) ed altri attributi dipendenti dal genere e dal tipo di simbolo; ad esempio, se si tratta di un parametro formale, bisognerà specificare la modalità di passaggio; se invece si tratta di un array occorrerà specificare l'ordine, il tipo, il numero, e i limiti degli indici di ogni dimensione, ecc.

La gestione delle operazioni sulla tabella dei simboli deve essere particolarmente efficiente per non compromettere la velocità complessiva del compilatore.

Il concetto di fase di compilazione deve essere distinto da quello di *passata* che va invece intesa come lettura del testo sorgente (o di una versione intermedia) dalla memoria di massa. Da questo punto di vista il caso ottimo si ha con i compilatori ad un'unica passata; il *Pascal* è uno dei linguaggi progettati tenendo presente anche questa esigenza.

Si osservi che per contenere il numero di passate è necessario che alcune fasi avvengano in "parallelo" o, meglio, che si susseguano alternativamente nel tempo.

Queste fasi sono tipicamente quelle di analisi lessicale, sintattica, semantica e generazione del codice intermedio (od oggetto se si opta per una traduzione diretta).

2.2.2 Analisi lessicale, sintattica e semantica

Durante la fase di *analisi lessicale* il programma sorgente viene letto carattere per carattere da sinistra verso destra e vengono riconosciute determinate sequenze di caratteri come elementi del lessico del linguaggio sorgente (parole chiave, identificatori, operatori, simboli di interpunzione, costanti, etichette, ecc.). Questi elementi vengono codificati internamente, dando luogo ad una rappresentazione interna più astratta e compatta del testo sorgente; ogni simbolo definito dall'utente viene inserito nella tabella dei simboli (se non esiste già) ed il suo valore viene rappresentato mediante un puntatore alla voce che lo contiene nella tabella.

La prima elementare traduzione effettuata durante l'analisi lessicale, va in ingresso alla fase di *analisi sintattica*.

In ogni caso l'analizzatore sintattico è basato su una grammatica libera per cui eventuali vincoli, dipendenti dal contesto, devono essere controllati in una successiva fase di *analisi semantica* che si fa carico di una serie di verifiche e di scelte come ad esempio:

a) verificare che ogni identificatore non sia dichiarato più di una volta nello stesso ambiente;

b) verificare che tutte le variabili siano state dichiarate prima del loro uso;

c) verificare (nei linguaggi con *goto*) che non manchino etichette ad istruzioni a cui si salta e che siano presenti una sola volta;

d) verificare che le interfacce di chiamata a sottoprogramma siano conformi a quelle richieste in posizione, numero e tipo degli argomenti;

e) verificare che in ogni assegnamento $X \leftarrow e$ il tipo di e sia compatibile con quello di X;

f) stabilire il tipo di un'espressione o di un'operazione in caso di sovraccarico degli operatori o di uso di espressioni miste.

2.2.3 Rappresentazioni intermedie

Alla fase di analisi semantica può seguire quella di *traduzione in codice oggetto*; questa soluzione è adottabile soltanto se non si ha alcuna pretesa sulla qualità del codice prodotto, cioè sulla sua efficienza in tempo di esecuzione e spazio occupato in memoria.

Questo è uno dei principali motivi per i quali è generalmente preferibile fare effettuare prima una *traduzione in un linguaggio intermedio* fra quello sorgente e quello oggetto. La versione intermedia del programma può infatti essere analizzata più efficientemente del codice oggetto al fine di apportare le trasformazioni necessarie per arrivare alla produzione di un codice più efficiente. Questa fase è denominata ottimizzazione.

Esistono diversi tipi di linguaggio intermedio. Fra i più comuni ricordiamo i seguenti:

- DAG (*Directed Acyclic Graph*) rappresentati con strutture dinamiche;

- *notazione polacca postfissa*;

- *liste di triple* $(operatore,\ operando_1,\ operando_2)$;

- *liste di quadruple*;

- *P-code.*

Della rappresentazione intermedia sotto forma di DAG parleremo estensivamente nel capitolo 4, dal momento che è quella adottata dal compilatore TAOmille.

Un linguaggio intermedio che risulta essere adatto alla sucessiva fase di ottimizzazione è anche quello delle liste di quadruple; una *quadrupla* è un'istruzione elementare del tipo:

(*operatore, operando*$_1$*, operando*$_2$*, destinazione*)

dove gli operandi possono essere solo variabili o costanti. Esse rappresentano istruzioni con un massimo di tre indirizzi di una macchina astratta. Ad esempio l'istruzione di assegnamento $A \leftarrow (A+B)*(C-D)$ potrebbe essere compilata come:

$$(+, A, B, T1)(-, C, D, T2)(*, T1, T2, A)$$

che sta per: $T1 \leftarrow A+B$; $T2 \leftarrow C-D$; $A \leftarrow T1 * T2$.

Le variabili $T1$ e$T2$ sono di transito o temporanee e vengono introdotte dal generatore di codice secondo opportuni criteri intesi anche a riutilizzare quelle che si rendono inattive e a sfruttare sottoespressioni comuni.

Il linguaggio intermedio *P-code* (Pascal-code) prodotto da quasi tutti i compilatori *Pascal* è il linguaggio di una macchina astratta nota come *macchina Pascal*.

Oltre a permettere una più efficace ottimizzazione, la scelta di passare per un linguaggio intermedio presenta molti vantaggi che vale la pena di rilevare:

a) Riduzione della distanza fra linguaggio sorgente e linguaggio oggetto con conseguente semplificazione del traduttore.

b) Maggiore *portabilità* del compilatore (con ciò si intende l'attitudine del compilatore ad essere adottato per un'altra macchina). Basterà infatti cambiare solo la parte di generazione del codice oggetto dal momento che le altre fasi sono indipendenti dalla macchina oggetto.

c) Può essere conveniente saltare la fase di traduzione in linguaggio macchina e accontentarsi di far eseguire il codice intermedio da un interprete scritto in linguaggio macchina. È quanto avviene per molti linguaggi generalmente interpretati come *BASIC, Snobol, Apl*; non mancano però esempi di *Fortran* e *Pascal* interpretati.

Come abbiamo già detto, alla generazione del codice intermedio segue generalmente una *fase di ottimizzazione*; più che di "ottimizzazione" sarebbe opportuno parlare di "*miglioramento del codice*" dal momento che la teoria della computabilità assicura che non possono esistere algoritmi generali in grado di trovare la versione ottima di un programma, cioè tale che non ne esistano di migliori.

Un criterio generalmente adottato per stabilire se il codice prodotto da un compilatore è "ottimo" è di confrontarlo con quello prodotto da un buon programmatore assembler.

Se manca la fase di generazione del codice intermedio, l'eventuale ottimizzazione viene eseguita direttamente sul codice oggetto ed è quindi fortemente dipendente dalla particolare macchina (ottimizzazione *hardware dependent*); in molti compilatori le azioni di ottimizzazione possono essere intraprese in più fasi della compilazione.

2.2.4 Generazione del codice oggetto

Nell'ultima fase, il codice intermedio (eventualmente ottimizzato) viene tradotto in codice oggetto; in qualche caso ciò può avvenire semplicemente attraverso un processo di macroespansione; i migliori risultati si hanno però con un programma traduttore specializzato.

La forma più comune del codice oggetto è quella rilocabile, cioè quella che richiede una successiva fase di collegamento (*link*) con librerie, sottoprogrammi o moduli esterni. In qualche caso, però, si può desiderare di disporre di un programma oggetto assoluto (cioè in forma direttamente eseguibile) o, viceversa, in linguaggio assembler.

```
    ...
    i = m-1; j = n; v = a[n];
    while(1)  {
     do i = i+1; while ( a[i] < v );
     do j = j-1; while ( a[j] > v  );
     if ( i >= j ) break;
     x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    ...
}
```

Figura 2.2: Codice C dell'esempio

2.3 Ottimizzazioni

In questa sezione introdurremo alcune delle più importanti trasformazioni per l'ottimizzazione del codice [4, 5].

Definiamo *blocco* una sequenza di istruzioni consecutive nella quale il flusso del programma entra all'inizio ed esce alla fine senza interruzioni o possibilità di salto.

Diremo che una trasformazione è *locale* se può essere effettuata guardando soltanto le istruzioni nel singolo blocco; altrimenti diremo che è globale. Molte trasformazioni possono essere effettuate sia a livello locale sia a quello globale. In genere le trasformazioni locali vengono effettuate per prime.

Nelle sezioni seguenti verrà svolto un esempio in cui si illustrano una serie di ottimizzazioni applicate al codice riportato in figura 2.2, successivamente alla sua trasformazione nella rappresentazione intermedia mostrata in figura 2.3.

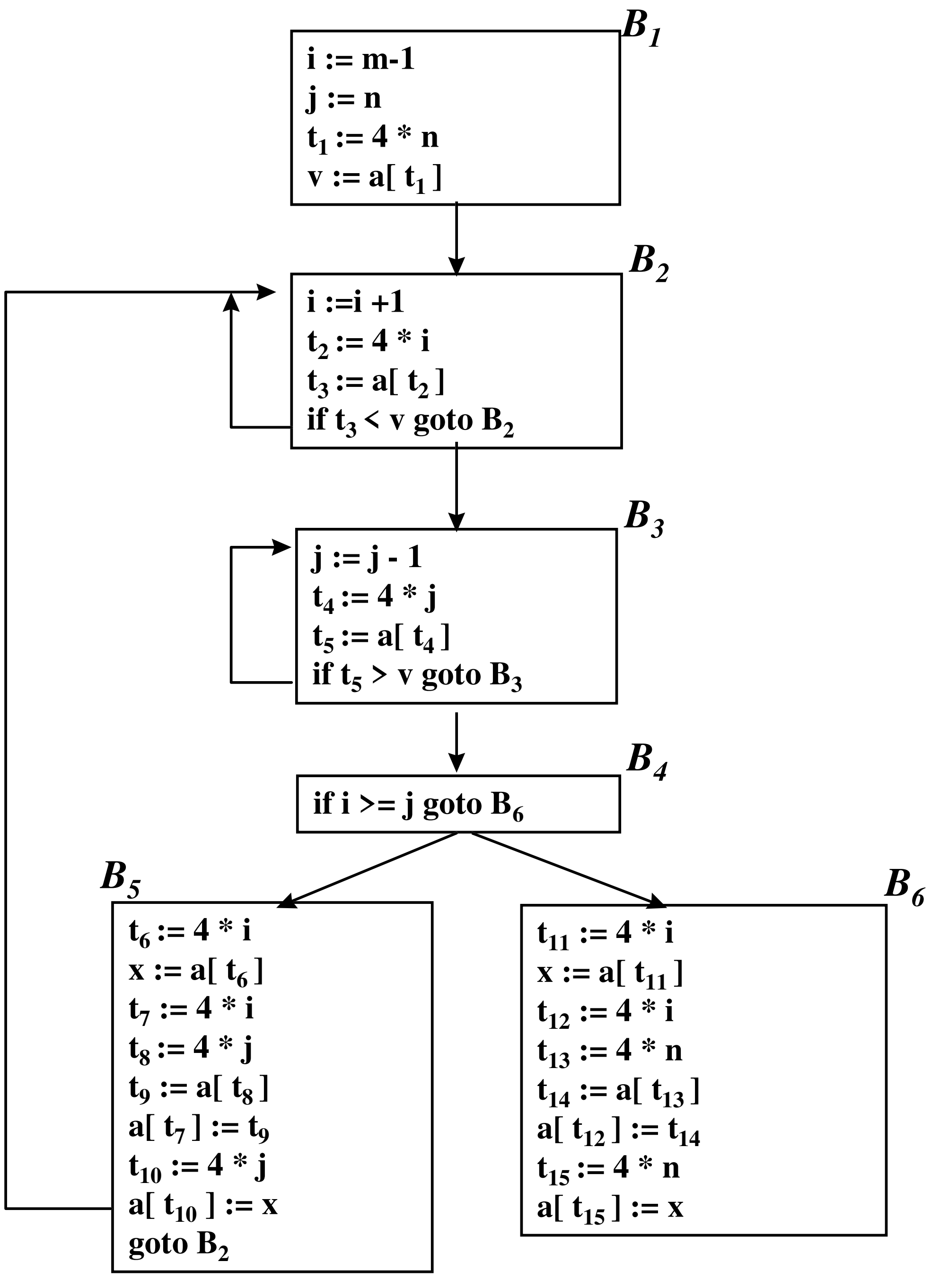


Figura 2.3: Diagramma di flusso

2.3.1 Trasformazioni che lasciano invariate le funzioni

Il compilatore può migliorare un programma senza cambiare le funzioni che calcola in molti modi differenti. Fra questi ricordiamo ad esempio l'eliminazione delle sottoespressioni comuni, il *copy-propagation*, l'eliminazione del codice non raggiungibile e il *constant-folding*. Nella sezione 4.6 vedremo come i primi due punti possano essere portati avanti durante la costruzione dei DAG dei blocchi di programma, mentre gli altri sono effettuati durante l'ottimizzazione globale del codice.

Spesso in un programma una stessa espressione viene calcolata molte volte, ad esempio ciò accade per il calcolo dello spiazzamento di un array. Purtroppo il programmatore in molti casi non è in grado di evitare queste duplicazioni perché giacciono ad un livello che non è accessibile dal sorgente. Ad esempio, il blocco B_5 nella figura 2.4(a) ricalcola `4*i` e `4*j`.

2.3.2 Eliminazione delle sottoespressioni comuni

In genere definiremo *sottoespressione comune* un'espressione E che è stata calcolata precedentemente nel programma e riappare successivamente senza che il suo valore sia cambiato nel frattempo. In questo caso, si può evitare di ricalcolarla e si può usare il valore calcolato precedentemente. Ad esempio, le assegnazioni di t_7 e t_{10} contengono, rispettivamente, le sottoespressioni `4*i` e `4*j`. Nella figura 2.4(b) sono state eliminate usando t_6 al posto di t_7 e t_8 al posto di t_{10}. Questo cambiamento sarebbe lo stesso che si otterrebbe

```
t6  := 4*i
x   := a[t6]
t7  := 4*i
t8  := 4*j
t9  := a[t8]
a[t7]  := t9
t10 := 4*j
a[t10]  := x
goto B2
```

```
t6  := 4*i
x   := a[t6]
t8  := 4*j
t9  := a[t8]
a[t6]  := t9
a[t8]  := x
goto B2
```

(a) *(b)*

Figura 2.4: Eliminazione delle sottoespressioni comuni

ricostruendo il codice intermedio a partire dalla rappresentazione a DAG del blocco iniziale.

Facciamo un altro esempio. Nella figura 2.5 è mostrato il risultato della eliminazione sia globale sia locale delle sottoespressioni dal blocco B_5 e B_6 del grafico di flusso di figura 2.3. Dapprima discutiamo la trasformazione del blocco B_5 e poi evidenzieremo alcune sottigliezze relative agli array.

Dopo aver eliminato le sottoespressioni locali comuni, in B_5 rimane la valutazione delle espressioni `4*i` e `4*j` come mostra la figura 2.4(b). Entrambe sono sottoespressioni comuni, in particolare, le tre istruzioni:

$$t_8 := 4 * j; \quad t_9 := a[t_8]; \quad a[t_8] := x$$

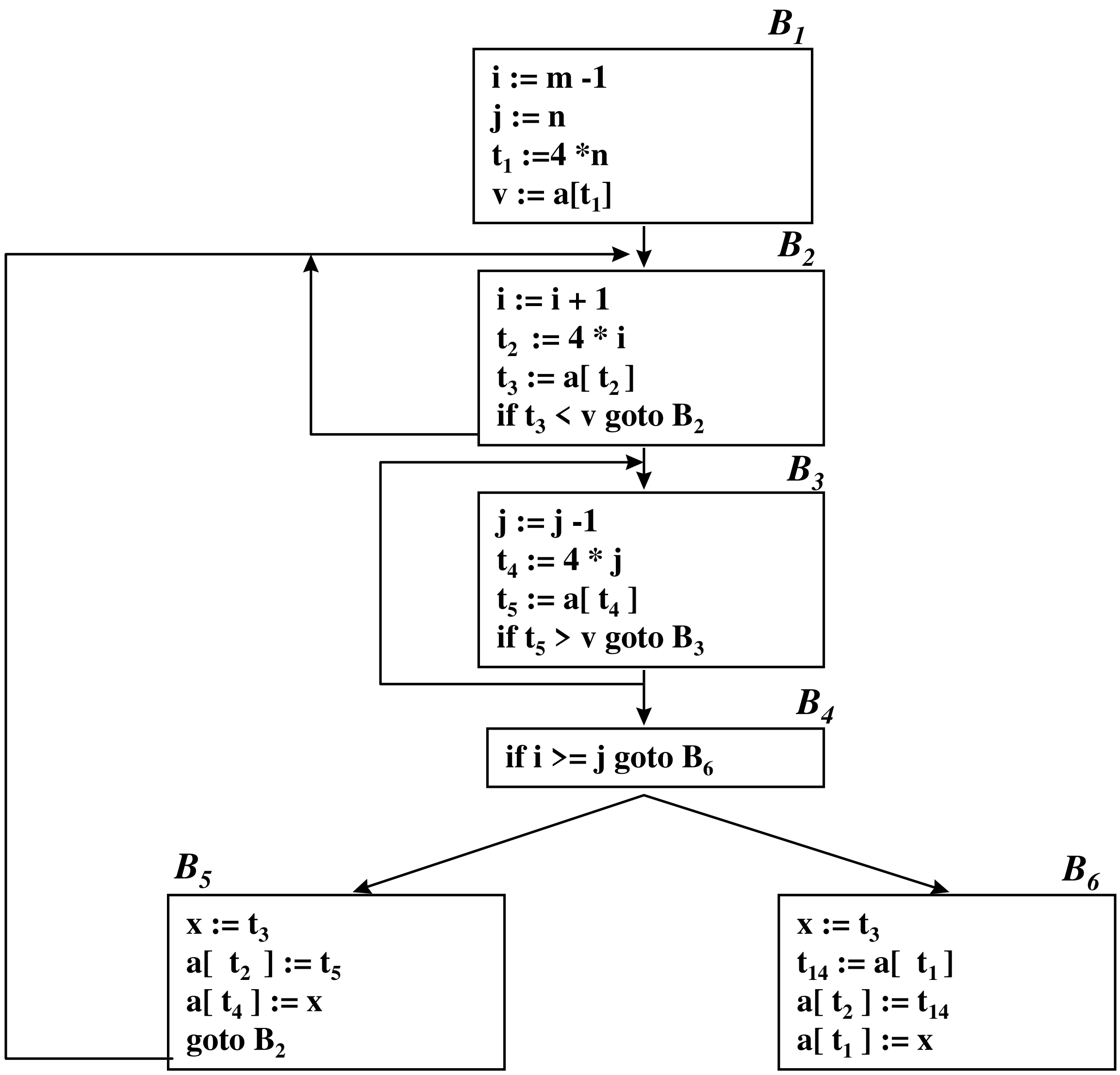


Figura 2.5: B_5 e B_6 dopo l'eliminazione delle sottoespressioni comuni

in B_5 possono essere sostituite con

$$t_9 := a[t_4]; \quad a[t_4] := x$$

usando la variabile t_4 calcolata nel blocco B_3. Infatti, in figura 2.5 appare chiaro che la variabile j non cambia valore da B_3 a B_5 e pertanto t_4 contiene ancora il giusto valore per l'espressione `4*j`. Un'altra sottoespressione comune appare in B_5 dopo che t_4 sostituisce t_8. La nuova espressione $a[t_4]$ corrisponde al valore $a[j]$ sul programma sorgente. In particolare, sia j, sia $a[j]$ mantengono il loro valore nel passaggio da B_3 a B_5 perché non ci sono assegnazioni all'array a. Quindi, le istruzioni

$$t_9 := a[t_4]; \quad a[t_6] := t_9$$

in B_5 possono essere sostituite con

$$a[t_6] := t_5 \ .$$

Analogamente, il valore assegnato ad x nel blocco B_5 di figura 2.4(b) è lo stesso di quello assegnato a t_3 nel blocco B_2. Il blocco B_5 di figura 2.5 è il risultato dell'eliminazione di tutte le sottoespressioni comuni dalle assegnazioni delle variabili $a[i]$ e $a[j]$ del programma sorgente dal blocco B_5 di figura 2.4(b). Le stesse procedure sono state effettuate sul blocco B_6 in figura 2.5.

L'espressione $a[t_1]$ nei blocchi B_1 e B_6 di figura 2.5 non viene considerata una sottoespressione comune benché t_1 possa essere utilizzata in entrambi i posti. Infatti, quando il flusso lascia il blocco B_1 e prima di raggiungere il blocco B_6 potrebbe passare per il blocco B_5 in cui è presente un'assegnazione all'array a. Perciò $a[t_1]$ potrebbe non avere più lo stesso valore del blocco B_1 una volta raggiunto il blocco B_6, e non è da considerarsi sicuro trattare $a[t_1]$ come una sottoespressione comune.

2.3.3 *Copy-propagation*

Il blocco B_5 può essere ulteriormente migliorato eliminando l'uso della variabile x usando due nuove trasformazioni. La prima riguarda gli assegnamenti della forma `f:=g` che sono noti come *istruzioni di copia*. Le *copie* sono presenti sia perché lo sono già nel codice sorgente, sia perché alcuni algoritmi di eliminazione delle sottoespressioni comuni ne introducono di nuove. Ad esempio, quando viene eliminata la sottoespressione comune presente in `c:=d+e` in figura 2.6, l'algoritmo usa una nuova variabile t per conservare il valore `d+e`. Poiché il flusso può raggiungere l'istruzione `c:=d+e` sia dopo l'assegnazione ad a, sia dopo l'assegnazione a b, sarebbe sbagliato sostituire direttamente `c:=d+e` con `c:=a` o `c:=b`.

L'idea sottostante la trasformazione relativa alla *copy-propagation* è di usare g invece di f ogni volta che compare dopo l'istruzione `f:=g`. Ad esempio, l'assegnazione $x := t_3$ nel blocco B_5 di figura 2.5 è una copia. Applicando la *copy-propagation* al blocco B_5 si ottiene:

$$
\begin{aligned}
x &:= t_3 \qquad (2.4)\\
a[t_2] &:= t_5
\end{aligned}
$$

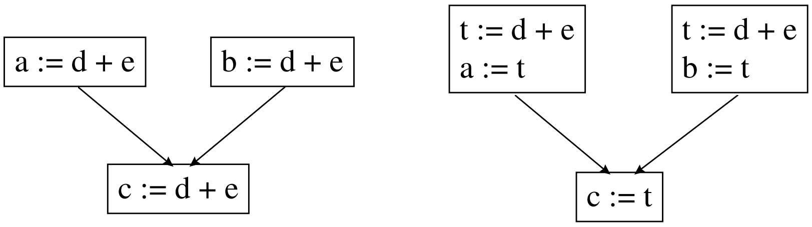


Figura 2.6: Copie introdotte durante l'eliminazione delle sottoespressioni comuni

$$a[t_4] := t_3$$

$$\text{goto } B_2$$

A prima vista potrebbe sembrare che non si sia ottenuto nessun miglioramento ma come vedremo oltre, grazie a questa modifica saremo in grado di eliminare del tutto l'assegnamento alla variabile x

2.3.4 Eliminazione del "codice morto"

Ad un certo punto del programma, una variabile si dice viva se il suo valore può essere utilizzato successivamente. Un concetto simile è quello del cosidetto "codice morto", cioè di istruzioni che valutano espressioni che non vengono mai usate nel resto del programma. Benché in genere un programmatore non inserisca mai volontariamente del codice morto, le trasformazioni discusse precedentemente possono farlo. Ad esempio, in istruzioni del tipo

```
if (debug) print ...
```

potrebbe risultare noto già a priori che la variabile `debug` abbia valore falso. Solitamente ciò accade perché nel codice c'è una istruzione del tipo

```
debug := 0
```

e il flusso del programma fino al momento dell'istruzione di controllo non ne cambia il valore. Se la trasformazione di *copy-propagation* sostituisce `debug` con 0 allora l'istruzione di `print` non può essere raggiunta. Pertanto, sarà possibile eliminare sia la stampa che la `if` dal codice oggetto. In generale, dedurre a tempo di compilazione che una espressione è una costante e usare la costante al posto dell'espressione è una trasformazione nota come *constant folding*.

Uno dei vantaggi della trasformazione *copy propagation* è che spesso istruzioni di copia diventano codice morto. Tornando all'esempio precedente, la trasformazione di *copy propagation* e l'eliminazione del codice morto rimuovono l'assegnazione a x e trasformano la (2.4) in

$$a[t_2] := t_5$$

$$a[t_4] := t_3$$

$$\text{goto } B_2$$

2.3.5 Ottimizzazione sui cicli

Segue ora una breve introduzione alle tecniche di ottimizzazione dei cicli, in particolare modo dei cicli annidati più interni dove il programma tende a passare la maggior parte del tempo. È nota la regola empirica secondo la quale i programmi passano approssimativamente il 90% del loro tempo di esecuzione eseguendo il 10% del loro codice [6]. In genere, il tempo di esecuzione di un programma diminuisce se viene ridotto il numero di istruzioni nel ciclo più annidato, anche a costo di aumentare quelle nei cicli più esterni. Le tecniche usate più frequentemente sono quattro:

1. *spostamento del codice* in cui il codice viene trasferito fuori dal ciclo;
2. *eliminazione delle variabili di induzione* con cui nell'esempio di figura 2.5 elimineremo le variabili i e j dal ciclo più annidato dei blocchi B_2 e B_3;
3. *riduzione in forza* in cui le operazioni "costose" in termini di tempo vengono sostituite da operazioni "economiche", come ad esempio la sostituzione delle moltiplicazioni intere con degli *shift*.
4. *loop unrolling* in cui viene espanso il corpo dei cicli.

2.3.6 Spostamento del codice

Con questa trasformazione, un'espressione che mantiene lo stesso valore indipendentemente dal numero di volte in cui viene eseguito il ciclo che la contiene viene spostata prima del ciclo. Notiamo che il concetto di "prima del ciclo" presuppone l'esistenza di un'entry per il ciclo. Ad esempio, la valutazione di `limit-2` è ciclo-invariante nella seguente istruzione:

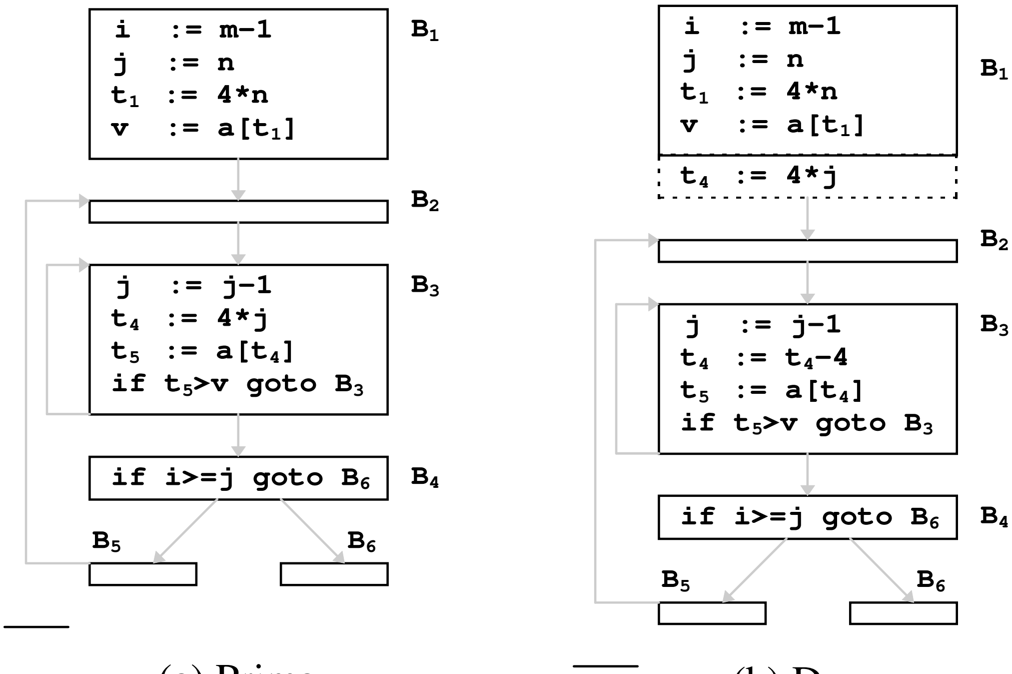
```
while ( i <= limit-2)
```

e il codice viene trasformato in:

```
t = limit-2
while ( i <= t)
```

2.3.7 Eliminazione delle variabili di induzione e riduzione in forza

Chiariamo meglio il significato delle seconde due trasformazioni della lista precedente applicandole al solito esempio. Soffermando l'attenzione sul ciclo intorno al blocco B_3 (vedi figura 2.7) notiamo che i valori di j e t_4 rimangono legati nel senso che ogni volta che j diminuisce di un'unità, t_4 viene decrementato di quattro perché $4*j$ viene assegnato a t_4 ogni volta. Questi identificatori sono noti come *variabili di induzione*. Quando in un ciclo compaiono una o più variabili di induzione, è possibile eliminarle tutte eccetto una tramite una trasformazione. Per il ciclo più annidato del blocco B_3 in figura 2.7(a) non si possono eliminare direttamente sia j sia t_4. Tuttavia, illustreremo la *riduzione in forza* e una parte del processo di eliminazione delle *variabili di induzione*. Soltanto nel momento in cui si considererà il ciclo B_2-B_5 si potrà eliminare j del tutto.



(a) Prima (b) Dopo

Figura 2.7: Riduzione in forza applicata a 4*j nel blocco B_3

Poiché la relazione $t_4 := 4*j$ è sicuramente verificata dopo l'assegnazione e t_4 non viene cambiata mai nel ciclo attorno a B_3, allora è chiaramente verificata la relazione $t_4 := 4*j-4$ subito dopo la $j := j-1$. Pertanto, sarà possibile sostituire a $t_4 := 4*j$ l'espressione $t_4 := t_4 - 4$. L'unico inconveniente è che quando il flusso del programma passa per B_3 per la prima volta t_4 non ha un valore definito. Basterà allora inizializzare la variabile t_4 laddove è inizializzata la j e il codice rimarrà invariato (vedi zona tratteggiata di figura 2.7(b)). La sostituzione della moltiplicazione con la sottrazione aumenta la velocità di esecuzione su molte macchine.

Una volta effettuata la *riduzione in forza* al ciclo attorno a B_2 e B_3, le variabili i e j sono necessarie soltanto per determinare l'esito del controllo per l'uscita dal blocco B_4. Poiché i valori di t_2 e t_4 soddisfano rispettivamente le relazioni $t_2 = 4*i$ e $t_4 = 4*j$ chiaramente il test $t_2 >= t_4$ sarà equivalente a quello $i >= j$. Una volta effettuata questa sostituzione, sia i nel blocco B_2 sia j nel blocco B_3 divengono delle variabili morte pertanto la loro assegnazione risulterà in codice morto che può essere eliminato.

Alla fine si arriverà al diagramma di flusso descritto in figura 2.8. Il miglioramento è palese, infatti il numero di istruzioni nei blocchi B_2 e B_3 è ridotto da 4 a 3, rispetto alla figura 2.3, quello di B_5 da 9 a 3, quello in B_6 da 8 a 3. Nonostante le istruzioni nel blocco B_1 siano aumentate da 4 a 6 chiaramente questo blocco è eseguito un numero di volte assai più piccolo e non influenza i tempi di esecuzione totali del codice.

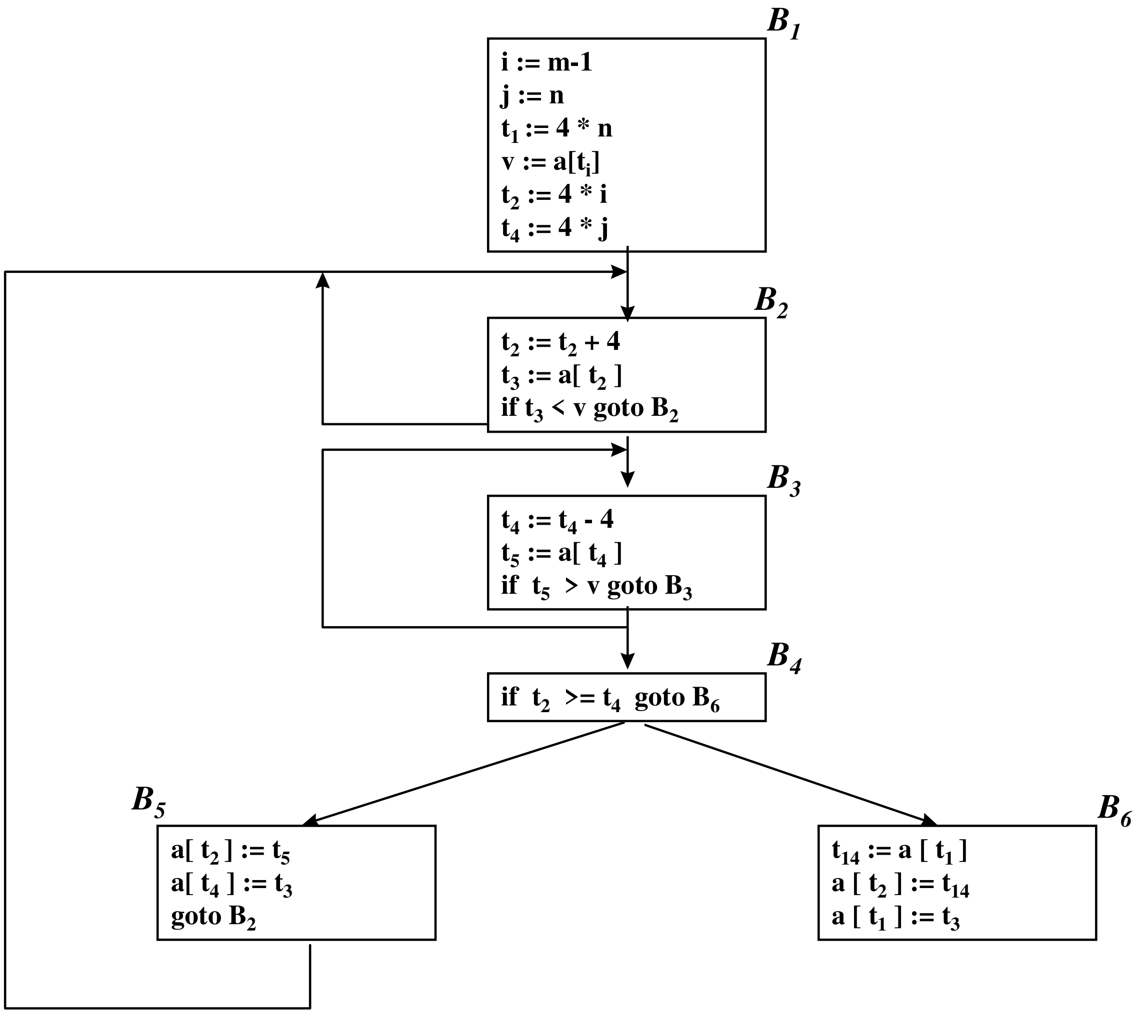


Figura 2.8: Diagramma di flusso dopo l'eliminazione delle variabili di induzione

2.3.8 *Loop unrolling*

Il *loop unrolling* consiste nel replicare il corpo originale del ciclo più volte e nell'aggiustare conseguentemente il codice di uscita dal ciclo. Gli effetti di questa trasformazione sono quattro:

1. Viene aumentata l'occupazione della memoria programma.

2. Il numero totale di istruzioni eseguite per il controllo dall'uscita del ciclo viene ridotto.

3. Con processori con architettura *pipelined* viene aumentato il riempimento delle *pipe* di calcolo.

4. Viene diminuito il rapporto B_C tra il numero di operazioni di accesso in memoria e il numero di operazioni di calcolo all'interno del ciclo [7].

Risulta che l'unico effetto indesiderato è il primo, mentre gli altri tre contribuiscono a migliorare l'efficienza del programma. È stato rilevato [8] che generalmente il corretto compromesso tra occupazione della memoria programma ed effetti benefici del *loop unrolling* si ottiene replicando il codice interno al ciclo un numero dell'ordine di 10 volte.

Il quarto punto è di importanza fondamentale nell'ambito delle applicazioni di calcolo intensivo. È possibile definire un *fattore di bilanciamento* B per una certo processore come il rapporto:

$$B = \frac{\text{massimo numero di accessi in memoria/ciclo}}{\text{massimo numero di operazioni di calcolo/ciclo}} \tag{2.5}$$

e un analogo fattore di bilanciamento per un ciclo come:

$$B_C = \frac{\text{numero di accessi in memoria}}{\text{numero di operazioni di calcolo}} \tag{2.6}$$

Il confronto tra B e B_C fornisce una misura per l'efficienza con cui le operazioni di un ciclo vengono eseguite su una certa architettura. Se $B_C > B$ l'esecuzione del ciclo richiede dati ad una velocità maggiore di quella che la macchina può fornire, e le risorse di calcolo saranno sottoutilizzate.

Capitolo 3

Il calcolatore parallelo APEmille

3.1 Il progetto APE

Il progetto APEmille dell'INFN è rivolto alla realizzazione di un calcolatore parallelo ad alte prestazioni per lo studio dei sistemi fisici. APEmille, che nella sua configurazione completa presenterà una potenza di picco di 1 Teraflop, rappresenta l'evoluzione dei precedenti modelli di calcolatori APE ed APE100 realizzati nell'ambito dello stesso gruppo di ricerca.

Il calcolatore APE, sviluppato negli anni tra il 1984 ed il 1989, era basato su una architettura di tipo SIMD (*Single Instruction stream-Multiple Data stream*) in cui i processori di calcolo erano disposti lungo un vettore monodimensionale. Questi processori, che lavoravano in aritmetica complessa floating point ed erano realizzati con dispositivi commerciali, avevano una potenza di calcolo pari a 64 Megaflop ed una memoria locale di 8/16 Megabyte. Le varie versioni realizzate prevedevano un massimo di 16 processori di calcolo con una potenza di picco di 1 Gigaflop per l'intera macchina.

Il progetto APE100, iniziato nel 1989 e concluso nel 1995, ha avuto come obiettivo la realizzazione di un calcolatore parallelo SIMD con potenza di calcolo di 100 Gigaflop (cfr. § 3.1). Per raggiungere questa potenza di picco è stato necessario ricorrere ad una architettura basata su un numero molto grande di processori, dell'ordine delle migliaia. Questo ha posto il problema della scelta del tipo di interconnessione da adottare tra i nodi di calcolo. Tale scelta è ricaduta su una topologia a reticolo tridimensionale con comunicazioni a primi vicini e condizioni al contorno periodiche, anche in base a considerazioni sulla necessità di limitare la complessità fisica della macchina entro limiti ragionevoli. Una svolta rispetto al progetto precedente è stata quella di progettare e realizzare dei chip *custom* per i nodi di calcolo (con potenza di 50 Megaflop), i controllori ed i dispositivi di comunicazione. Questo ha consentito di ridurre la dimensione fisica della macchina, il suo consumo ed i costi necessari alla sua realizzazione. L'architettura adottata per APE100 si è rivelata adeguata alla implementazione di molti algoritmi di simulazione di QCD su reticolo ed in generale per la soluzione di problemi omogenei, quali l'analisi statistica dei sistemi complessi, la fluidodinamica, le simulazioni di reti neurali.

In questi campi di applicazione risulta molto importante poter disporre di un calcolatore con potenza di calcolo dell'ordine del Teraflop. Ad esempio questa potenza è la minima necessaria per simulazioni realistiche di full QCD, o per la simulazione della dinamica globale dell'atmosfera con una risoluzione tale da poter tenere conto dei dettagli dell'orografia della superficie terrestre. Il progetto APEmille prevede a questo scopo la realizzazione di un calcolatore parallelo dalle caratteristiche architetturali simili a quelle di APE100, descritte in dettaglio nella prossima sezione. Nel progetto è stata confermata la scelta adottata per APE100 di progettare e realizzare dei chip *custom* per i dispositivi fondamentali della macchina. Questo comporta una serie di vantaggi rispetto all'uso di componenti commerciali, quali la possibilità di sfruttare appieno le possibilità fornite dalle tecnologie realizzative, una maggiore affidabilità della macchina che non ha bisogno di tutta la logica esterna che sarebbe necessaria per adattare dei dispositivi commerciali ai propri scopi, la maggiore compattezza dei dispositivi che non prevedono parti e caratteristiche inutilizzabili (come è inevitabile che sia per dei dispositivi commerciali).

Nella fase iniziale del progetto si sono individuati i requisiti necessari a soddisfare le richieste imposte dalle applicazioni di cui si è parlato precedentemente [9]. Sulla base di queste specifiche si è sviluppato un progetto di massima della macchina [10], a partire dal quale si è implementato un simulatore funzionale del calcolatore scritto in C++. I risultati forniti dal simulatore sulle applicazioni hanno suggerito delle indicazioni utili all'aggiustamento successivo del progetto [11, 12, 13, 14]. La fase successiva, già avviata, consiste nella simulazione in VHDL (linguaggio di descrizione hardware standard IEEE) dei dispositivi *custom* che devono essere realizzati: l'unità di calcolo Jane, il controllore Tarzan, il dispositivo di comunicazione Cheetah, l'interfaccia con il sistema *host* Apenic. La descrizione in VHDL della processing board permetterà di simulare il sistema nel suo complesso, tenendo quindi conto delle interazioni tra i vari dispositivi. Il confronto tra il comportamento della macchina simulato in VHDL e quello riscontrato nei prototipi che verranno realizzati consentirà una verifica delle specifiche prima della revisione finale del progetto.

3.2 L'architettura di APEmille

Facendo riferimento alle classificazioni introdotte nella sezione 1.2, APEmille è un calcolatore parallelo di tipo SIMD, scalabile, *fine grained*, con topologia a reticolo cubico tridimensionale con condizioni al contorno periodiche e nodi di calcolo costituiti da processori VLIW e dalle loro memorie dati. La memoria programma dei processori e del controllore è separata dalle rispettive memorie dati in maniera conforme all'architettura Harvard.

Accanto a queste caratteristiche ereditate da APE100, APEmille presenta nuove funzionalità che ne estendono la versatilità rispetto al numero di differenti applicazioni che potranno essere implementate in maniera efficiente su di esso. La rete di comunicazione internodo presenta nuove modalità di trasferimento: sono implementati tre tipi di comunicazione broadcast (uno a molti) ed è in fase di definizione la modalità per i trasferimenti non omogenei, in cui ogni nodo del reticolo può essere sorgente o destinatario di una trasmissione di dati (questo tipo di comunicazione è chiamato comunicazione morbida). I nodi di calcolo sono in grado di effettuare operazioni aritmetiche su dati interi a 32 bit, dati floating point in singola precisione, dati floating point in doppia precisione (64 bit), dati complessi floating point e coppie ordinate di dati in singola precisione. Una innovazione fondamentale è data dalla capacità di indirizzamento locale da parte dei nodi di calcolo: ad un indirizzo globale, cioè uguale per tutti i nodi, è possibile sommare uno spiazzamento calcolato sul nodo. Questo fa sì che l'istruzione eseguita (la stessa per tutti i nodi) possa operare su dati allocati in locazioni di memoria diverse. Per questa ragione si può classificare APEmille come un calcolatore SIMAMD (*Single Instruction stream - Multiple Address - Multiple Data stream*).

Dal punto di vista del suo stato di funzionamento, APEmille può essere descritto come l'unione di due blocchi fondamentali: un *kernel* sincrono che si occupa dell'esecuzione del programma ed una *shell* asincrona relativa all'interfaccia con la rete di PC *host* sulla quale è implementato il sistema operativo distribuito. Queste due modalità di funzionamento vengono chiamate rispettivamente *run mode* e *system mode*.

L'elemento base di APEmille è la *Processing Board* (PB), la scheda che ospita 8 processori aritmetici Jane con le loro memorie dati, il controllore Tarzan con la sua memoria dati, la memoria programma ed il dispositivo di comunicazione Cheetah, come mostrato in figura 3.1.

In *run mode* Tarzan controlla il flusso del programma, invia le parole di microcodice e gli indirizzi globali agli 8 Jane che svolgono i calcoli aritmetici, genera ed invia gli indirizzi remoti per Cheetah che si occupa dei trasferimenti

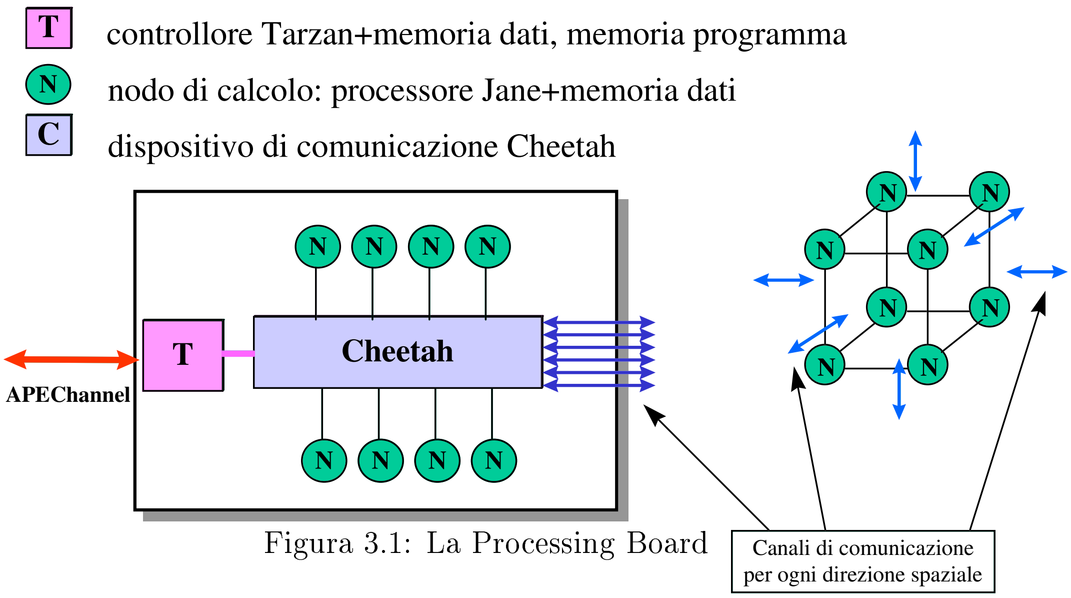


Figura 3.1: La Processing Board

dei dati tra i diversi nodi di calcolo, interfaccia la PB con l'APEChannel che è il canale di comunicazione con la rete di PC host. In *system mode* Tarzan decodifica le richieste inoltrate sull'APEChannel dalla rete di host per accedere alle risorse della PB, Cheetah fornisce il canale di accesso ai nodi di calcolo che in questo stato sono inattivi (se si escludono i controllori di memoria che effettuano le operazioni di lettura, scrittura e refresh). In figura 3.1 sono mostrati i dispositivi presenti nella PB. Ogni PB è un sottosistema di APEmille che presenta tutte le funzionalità previste dalla macchina ed in cui i nodi di calcolo sono disposti ai vertici di un reticolo cubico 2x2x2 e comunicano con i loro 6 primi vicini.

La scalabilità della macchina si ottiene replicando l'elemento di base PB in modo da formare una delle configurazioni mostrate in figura 3.2: un *subcrate* si ottiene inserendo 4 PB negli slot di alloggiamento delle schede del backplane di un crate, 16 PB formano un crate completo (mostrato in dettaglio in figura 3.3), 4 crate inseriti in un rack formano una configurazione a torre, infine 4 torri costituiscono un sistema APEmille completo.

Lungo il backplane dei crate corrono i canali di comunicazione lungo le direzioni Y e Z; lungo queste la dimensione della macchina risulta quindi limitata ad 8 nodi. Dalla parte frontale del crate si effettuano i collegamenti via cavo lungo la direzione X, che non è quindi limitata dalle caratteristiche fisiche del rack, e nella configurazione completa di APEmille (32x8x8) ha una dimensione di 32 nodi.

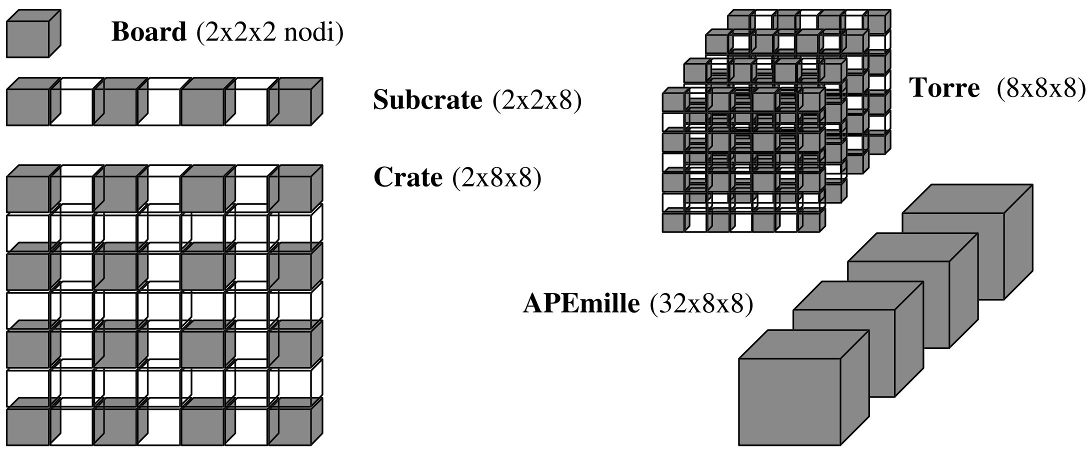


Figura 3.2: Configurazioni di APEmille

La potenza di calcolo di picco del processore aritmetico Jane si può calcolare facilmente considerando che esso è in grado di effettuare ad ogni ciclo di clock una operazione *normal* (a*b+c) con operandi complessi, che corrisponde ad 8 operazioni floating point in singola precisione. Alla frequenza di funzionamento di 66 MHz si ha quindi una potenza di 528 MFlop per il singolo nodo di calcolo. Nella tabella 3.1 sono elencate le possibili configurazioni di APEmille con le relative topologie e potenze di picco.

Configurazione	**Numero di nodi**	**Topologia**	**Potenza di picco** [GFlop]
PB	8	2x2x2	0.528
SUBCRATE	32	2x2x8	16.896
CRATE	128	2x8x8	67.584
TORRE	512	8x8x8	270.336
APEmille	2048	32x8x8	1081.344

Tabella 3.1: Configurazioni di APEmille e loro potenze di picco

Il funzionamento in *system mode* della macchina è gestito dall'esterno da una rete di host, la *Control Network*. Ogni host è una scheda PC dotata di più slot di espansione PCI (*Peripheral Computer Interconnect*), uno standard di fatto per la connessione ad alta banda passante di periferiche. Inoltre ciascun host, chiamato ACP (*Asynchronous Controller Processor*), è provvisto di una scheda di rete LAN (*Local Area Network*) e di dispositivi di memoria di massa e comunica con 4 schede PB su un bus parallelo, attraverso il protocollo dedicato APEChannel. L'interfaccia tra ACP e APEChannel è fornita da una scheda in standard PCI chiamata APEnic (*APE network interface card*) che è alloggiata

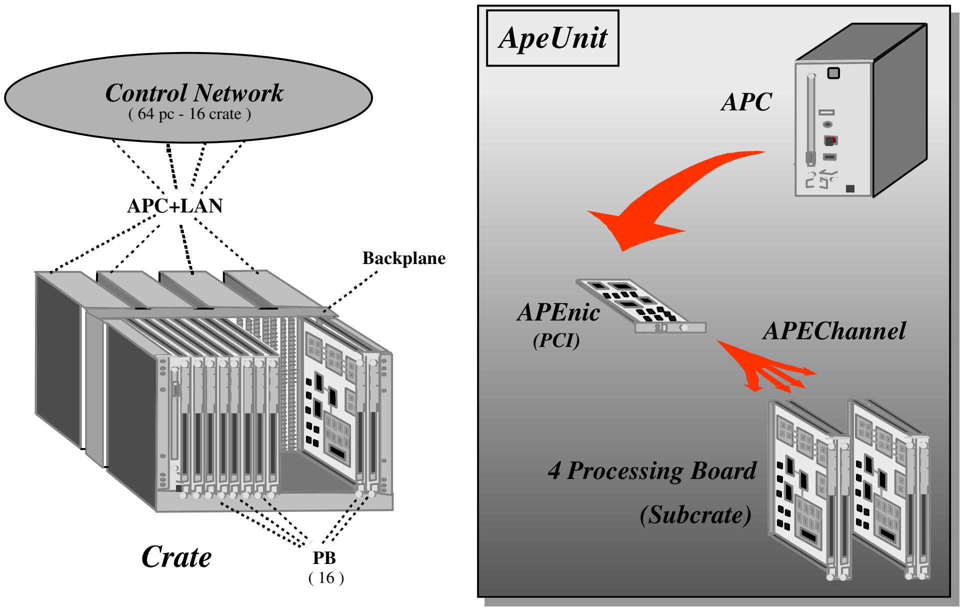


Figura 3.3: L'APEUnit, il Crate e la rete degli host

in uno slot di espansione PCI della ACP. Il bus parallelo APEChannel è largo 39 bit (32 bit di parola + 7 bit di codice Hamming per il riconoscimento e la correzione degli errori) e lavora ad una frequenza di 33 MHz, corrispondente ad una banda passante per il canale di 132 Mbyte/s. L'APEnic si occupa anche di distribuire il segnale di *clock*, alle PB.

Il più piccolo sottosistema completo di APEmille, chiamato AU(*APEUnit*), è costituito da 4 PB (un subcrate) connesse attraverso l'APEChannel ad una ACP, come mostrato in figura 3.3. I vari ACP comunicano attraverso una rete locale seriale, e per mezzo di questa fanno capo ad una scheda PCI alloggiata nel PC che presiede la rete, detta *Root Board*, che colleziona i segnali provenienti dalle AU e a partire da questi genera e distribuisce i corrispondenti segnali globali. In questo modo la *Root Board* gestisce il sincronismo dell'intera macchina.

Il sistema operativo distribuito sulla rete di host provvede a dirigere le operazioni di I/O tra memorie e registri dei processori delle PB ed i dispositivi di memorizzazione di massa presenti sulla *Control Network*. Data la struttura modulare costituita dalle AU e dal sistema operativo distribuito, è prevista la possibilità di effettuare il partizionamento della macchina sia da parte di un singolo programma che potrà avviare in questo modo più *thread* SIMD simultanei, sia per eseguire più programmi su sottosistemi diversi della macchina.

3.3 Architettura e funzionalità dei dispositivi VLSI

3.3.1 Il controllore Tarzan

Il paradigma architetturale SIMD comporta la presenza di un controllore che gestisca e coordini il funzionamento della rete dei nodi di calcolo. La struttura modulare di APEmille prevede la presenza di 1 controllore per ogni 8 nodi di calcolo, come risulta dalla descrizione della Processing Board (PB) svolta nella sezione precedente. Tarzan è il processore di controllo di APEmille ed è coinvolto in ogni operazione che avviene sulla PB. Tarzan sarà realizzato su un singolo chip al cui interno saranno contenute le *device* mostrate in figura 3.4. I due blocchi principali sono dati dall'APEcontroller e dall'APEdecoder e verrano descritti nel seguito.

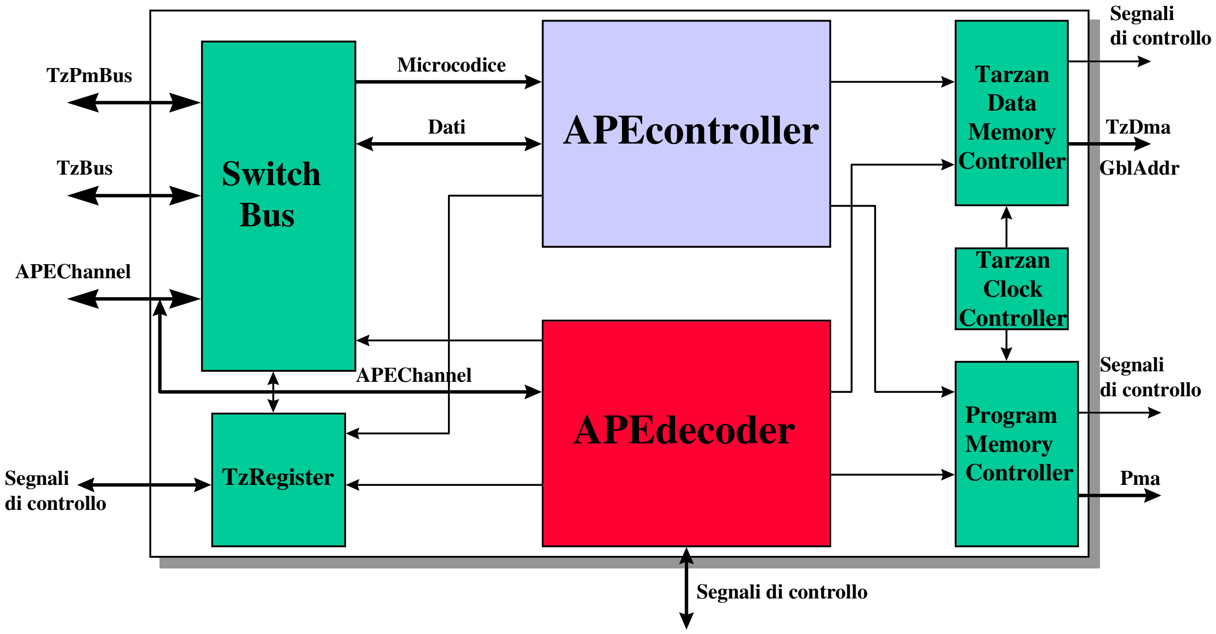


Figura 3.4: Schema a blocchi del controllore Tarzan

Lo Switch Bus effettua la opportuna connessione tra i bus esterni, ovvero il TzPmBus (bus dati della memoria programma lungo il quale viaggiano le parole di microcodice destinate a Tarzan), il TzBus (bus dati da e verso la memoria dati di Tarzan e verso Cheetah), l'APEChannel (bus di comunicazione con la rete di host) ed i dispositivi interni del controllore.

Il TzRegister contiene i registri di stato della PB e la logica necessaria al loro accesso.

Il Tarzan Clock Controller provvede alla distribuzione del segnale di clock, alla gestione del reset e delle transizioni di stato della macchina.

Il Tarzan Data Memory Controller è l'unità che gestisce il funzionamento della memoria dati di Tarzan (costituita da 512 Kbyte di memoria di tipo statico).

Il Program Memory Controller gestisce la memoria programma che è realizzata usando RAM sincrona dinamica (SDRAM). La sua dimensione è di 512 KWord, ogni word è lunga 192 bit e contiene le parole di microcodice per Tarzan e Jane.

La presenza di due grandi dispositivi principali all'interno di Tarzan risponde all'esigenza di effettuare da parte del controllore due classi differenti di operazioni a seconda dello stato della macchina. Per grandi linee si può affermare che l'APEdecoder e l'APEcontroller sono attivi rispettivamente quando APEmille si trova in *system mode* ed in *run mode*. Risulta quindi opportuno svolgere una descrizione di Tarzan a partire dallo stato di funzionamento della macchina.

In *system mode* il nucleo sincrono della macchina è fermo; l' esecuzione del programma è bloccata per consentire l'accesso dall'esterno alle risorse della PB. Il sistema operativo distribuito sulla *Control Network* può inoltrare richieste di I/O secondo il percorso che va dall'ACP, alla sua interfaccia APEnic, all'APEChannel e da questo ad ognuno dei 4 Tarzan che sono presenti sulla AU. In questo modo è possibile leggere o scrivere le *device* presenti su ognuna delle 4 PB di una AU, ovvero:

- Memoria dati di Tarzan.
- Memoria programma.

- Memorie dati degli 8 Jane.
- Registri di Tarzan.
- Registri di Cheetah.
- Registri di Jane.

L'APEdecoder è il dispositivo interno a Tarzan che riceve e decodifica i dati inviati dall'host attraverso l'APEChannel. Questi dati sono costituiti da parole di 32 bit che veicolano le richieste di I/O secondo il protocollo APEChannel. Senza inoltrarci in una descrizione dettagliata di questo protocollo di comunicazione, possiamo dire che la parola di 32 bit è divisa in più campi che contengono la codifica relativa ad informazioni quali gli identificatori della PB e della device interessate dall'operazione, gli indirizzi di memoria o il numero di registro ed il tipo di operazione da effettuare (lettura, scrittura, halt, run etc).

In *run mode* il nucleo sincrono di APEmille, essenzialmente il controllore e la rete di processori aritmetici, esegue il codice contenuto nella memoria programma. L'APEcontroller è un processore VLIW che in questa modalità di funzionamento della macchina controlla il flusso del programma, genera indirizzi per la memoria dati di Tarzan, indirizzi globali per la memoria dati dei processori aritmetici e per il dispositivo di comunicazione Cheetah, svolge operazioni logiche e aritmetiche su dati interi.

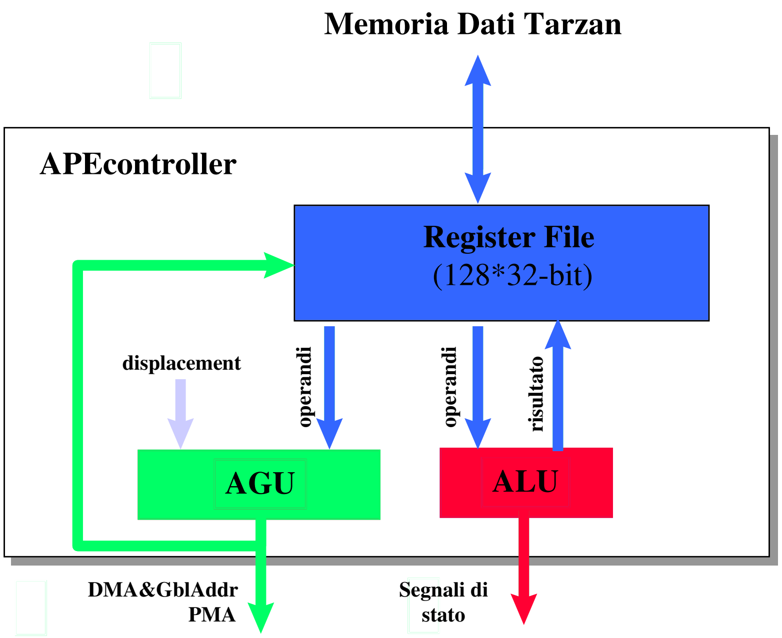


Figura 3.5: Schema a blocchi dell'APEcontroller

In figura 3.5 è riportato uno schema a blocchi di questo dispositivo; sono mostrati le unità ed i percorsi di dati e segnali di controllo principali.

Il Register File (RF) è una memoria multiporta da 128 locazioni a 32 bit. Questo dispositivo presenta 3 porte di uscita, una porta di ingresso ed una porta bidirezionale. Nella parola di microcodice di Tarzan sono presenti i 35 bit necessari ad associare, per ogni istruzione, uno dei 128 registri con ciascuna delle porte del Register File.

La ALU (*Arithmetic Logic Unit*) è l'unita aritmetico-logica preposta al calcolo intero su Tarzan. Essa è in grado di effettuare operazioni di somma, moltiplicazione, divisione, operazioni logiche *bitwise* ed operazioni di *shift* e *rotate*. Gli operandi sono prelevati dalle porte di uscita del RF e sono in numero variabile da 2 a 3, a seconda dell'operazione. Il risultato della operazione viene posto in ingresso sul RF e salvato in uno dei registri. Il progetto di questo componente prevede una *pipeline* a 5 livelli, con l'unità di calcolo in grado di svolgere, a *pipeline* piena, una operazione ad ogni ciclo di clock.

La AGU (*Address Generation Unit*) provvede alla generazione degli indirizzi per la memoria programma, per la memoria dati di Tarzan e per la memoria dati di Jane (indirizzo globale). In quest'ultimo caso per ottenere l'indirizzo di memoria dati di Jane all'indirizzo globale generato da Tarzan sarà possibile sommare nel controllore di memoria di Jane un intero locale calcolato sul nodo (*local addressing*). La AGU prevede 3 dati in ingresso, questi sono forniti da due porte di uscita del Register File (Op0 e Op1) e dal *displacement*, che è un dato intero a 32 bit contenuto nella parola di microcodice. Il *displacement* è quindi un dato noto a tempo di compilazione, ad esempio una costante o l'indirizzo di base di una struttura dati. A seconda del numero di operandi che entrano nel calcolo effettivo dell'indirizzo si parla rispettivamente di 3 tipi di Memory Operand (MOP):

- MOP1: *displacement*.
- MOP2: *displacement*, Op0.
- MOP3: *displacement*, Op0, Op1.

A livello di istruzioni assembly è possibile specificare come effettuare il calcolo degli indirizzi scegliendo il formato degli operandi corrispondente a uno dei tre possibili(cfr. § 4.3.1). La AGU calcola la somma dei 24 bit meno significativi

(LSB) dei dati che riceve in ingresso; gli 8 bit più significativi (MSB) sono ottenuti semplicemente copiando quelli corrispondenti del *displacement* o di Op0, a seconda di quanto contenuto in un apposito campo della parola di microcodice. Il formato dell'indirizzo globale generato da Tarzan è infatti dato da due campi: i 24 bit LSB indirizzano le memorie dati dei Jane, gli 8 bit MSB costituiscono il *remote address* che deve essere inviato al dispositivo di comunicazione Cheetah in caso di trasferimenti di dati tra nodi di calcolo (trasferimenti remoti). Il *remote address* costituisce un puntatore alla tabella RLUT (*Remote Look Up Table*) contenuta in Cheetah che viene usata per determinare il nodo di destinazione del trasferimento remoto secondo lo schema mostrato in figura 3.9.

La AGU effettua il controllo del flusso del programma in esecuzione calcolando ad ogni ciclo l'indirizzo della memoria programma PMA (*Program Memory Address*) che contiene la parola di microcodice da eseguire al ciclo successivo. Il calcolo dell'indirizzo avviene incrementando il valore del PMA di una unità nel caso di flusso sequenziale di istruzioni, o calcolando il nuovo indirizzo a partire da un MOP2 nel caso di istruzioni di salto. Queste possono essere di tipo incondizionato o condizionato, dove la condizione per il salto può essere valutata sulla base di operazioni logiche con operandi di tipo intero, eseguite nella ALU, o sulla base di condizioni locali ai nodi rese globali dagli operatori ANY, ALL e NONE (cfr. §4.2.1). In quest'ultimo caso la valutazione della condizione per il salto avviene considerando l'operatore applicato alla condizione locale ed il segnale GBIf (*Global If*) che giunge in ingresso all'APEcontroller. L'operatore ALL viene trattato a basso livello convertendolo in ANY e negando la condizione alla quale è applicato. Il GBIf indica come viene valutata una condizione da tutta la macchina, ovvero se essa è vera per alcuni nodi di calcolo oppure per nessun nodo di essi. Questo segnale viene inviato all'APEcontroller dalla Root Board che lo genera a partire dai segnali PBIf che essa colleziona tramite gli APEnic di ogni AU. Il segnale PBIf è infatti generato da ogni PB sulla base del valore che la condizione assume sui suoi nodi ed è inviato all'APEnic della AU a cui essa appartiene.

3.3.2 Il processore aritmetico Jane

Jane è il processore VLIW *pipelined* che, assieme alla sua memoria dati, costituisce il nodo di calcolo di APEmille. Esso svolge operazioni aritmetiche e logiche su dati floating point ed interi. I tipi di dato floating point che il processore è in grado di elaborarare sono il singola precisione (32 bit), il doppia precisione (64 bit), il complesso (64 bit) ed il *vector* (64 bit). I dati floating in singola e doppia precisione sono rappresentati in formato standard IEEE. Il tipo complesso è costituito da una coppia ordinata di dati in singola precisione che rappresentano la parte reale ed immaginaria del numero complesso. Il tipo *vector* è formato da una coppia ordinata di dati in singola precisione sui quali Jane è in grado di effettuare operazioni aritmetiche simultaneamente (architettura superscalare del processore). In figura 3.6 è mostrato uno schema a blocchi di questo dispositivo.

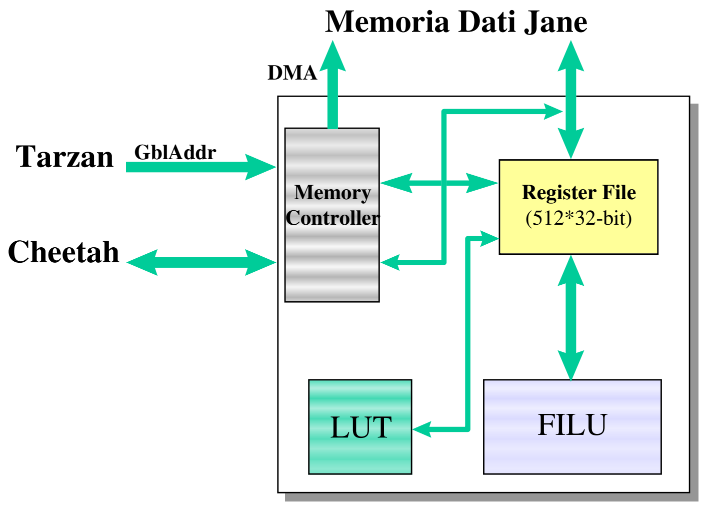


Figura 3.6: Schema a blocchi del processore Jane

Il Memory Controller genera i segnali necessari a gestire la memoria dati SDRAM (lettura, scrittura, refresh). Questa avrà nella prima versione di APEmille una capacità di 8 MByte per passare a 32 MByte nelle versioni successive della macchina. Oltre a questi segnali, il controllore provvede a generare l'indirizzo della memoria dati Jane per i trasferimenti tra questa ed il Register File. Il DMA viene calcolato sommando all'indirizzo globale proveniente da Tarzan un *offset* calcolato precedentemente dallo stesso Jane ed immagazzinato in una locazione del Register File. Questo meccanismo consente ad ogni nodo di calcolo di accedere a locazioni della memoria dati in principio differenti da nodo a nodo (*local addressing*). Il Memory Controller interviene anche nei trasferimenti remoti, cioè quelli tra il Register File e la memoria dati di un altro Jane. In questo caso il Memory Controller agisce sotto il comando del dispositivo di comunicazione Cheetah il quale viene connesso al Register File a partire dal bus che collega questo alla memoria locale del nodo.

Il Register File è una memoria multiporta con 512 locazioni da 32 bit ognuna. Questo dispositivo deriva dall'unione di due blocchi analoghi da 256 locazioni già usati per il MAD di APE100, ma dal punto di vista delle sue funzionalità può essere visto come un unico blocco con 3 porte di uscita, una porta di ingresso ed una porta bidirezionale. Il bit LSB di indirizzo del Register File seleziona uno dei due blocchi, che per questa ragione vengono detti banco pari e banco dispari. I dati floating point in doppia precisione, quelli complessi e i *vector* sono allocati in due locazioni contigue del register file. L'allocazione nel Register File di dati di questi tipi deve inoltre soddisfare i seguenti requisiti di allineamento: i 32 bit meno significativi dei dati in doppia precisione devono occupare una locazione del banco pari, la parte reale di un dato complesso deve occupare una locazione del banco pari, il primo degli elementi di un *vector* deve essere allocato in una locazione del banco pari.

La FILU (*Floating point Integer Arithmetic Logic Unit*) è l'unità che effettua i calcoli aritmetici e logici sui dati floating point ed interi. L'operazione aritmetica fondamentale che essa è in grado di eseguire sui dati floating point è la *normal*, che corrisponde al calcolo dell'espressione:

$$a * b + c \tag{3.1}$$

Gli operandi sono prelevati dal Register File attraverso le 3 porte di uscita, il risultato dell'operazione viene immagazzinato in una locazione dello stesso dispositivo attraverso la sua porta di ingresso. La FILU prevede tre *pipeline* di calcolo, come mostrato in figura 3.7.

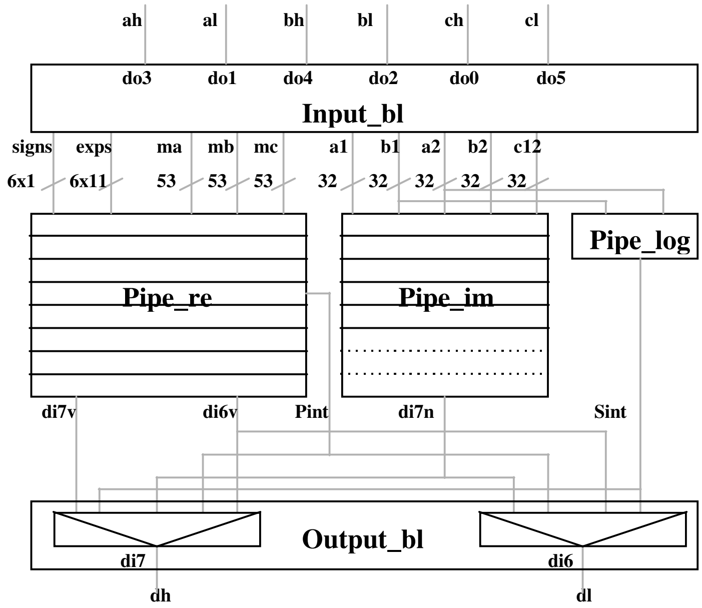


Figura 3.7: La FILU

La *pipeline* chiamata Pipe_im è quella relativa al calcolo in singola precisione. La combinazione di Pipe_re e Pipe_im consente di effettuare le operazioni sui dati in doppia precisione, sui complessi e sui vector. Nel caso di dati di tipo vector le due *pipeline* agiscono indipendentemente ciascuna su tre operandi in singola precisione, raddoppiando la potenza di picco rispetto al caso in cui si usi il tipo di dato in singola precisione. Una terza *pipeline* di calcolo, chiamata Pipe_log, è dedicata all'esecuzione delle operazioni logiche ed all'aritmetica sui dati interi. I tempi di esecuzione per una singola operazione normal sono di 8 cicli di clock per dati complessi e in doppia precisione, di 6 cicli di clock per dati in singola precisione e vector e di 2 cicli di clock per i dati interi. La struttura delle *pipeline* è tale da garantire, quando sono piene, l'esecuzione di una operazione normal ad ogni ciclo di clock.

Tipo degli operandi	**Potenza di picco** [MFlop]	B
Complesso	528	1:8
Vector	264	1:4
Doppia precisione	132	1:2
Singola Precisione	132	1:2

Tabella 3.2: Potenze di picco di Jane e rapporto tra operazioni di accesso in memoria e operazioni di calcolo floating point in un ciclo per i diversi tipi di dato

Questo corrisponde, alla frequenza di funzionamento del processore di 66 MHz, alle potenze di picco riportate in tabella 3.2, calcolate tenendo conto che una operazione normal su dati complessi corrisponde a 8 operazioni floating point e che su dati in doppia precisione corrisponde comunque a 4 *flop*. L'altro dato riportato in tabella è il rapporto $B = \frac{A}{F}$ tra il massimo numero possibile di accessi in memoria dati A e quello di operazioni di calcolo floating point F in un ciclo di clock. Questo dato fornisce un indice dell'efficienza con cui un certo algoritmo può essere implementato su APEmille, tenendo conto del bilanciamento tra gli accessi in memoria ed i calcoli floating point richiesti dall'algoritmo stesso. Nel calcolo di B si è considerato che Jane sia in grado di accedere ad un nuovo dato nella memoria dati SDRAM ad ogni ciclo. Questo è vero nella condizione in cui la lettura o la scrittura riguardi un insieme di dati allocati in maniera contigua in memoria (lettura o scrittura a *burst*).

La FILU è in grado di eseguire una serie di istruzioni di comparazione per i vari tipi di dato. L'effetto di queste istruzioni è di memorizzare il risultato booleano della comparazione nel *top* dello stack previsto per la logica dell'IF. Condizioni più complesse sono valutate applicando gli operatori logici alla coppia di valori booleani in cima allo stack, estraendo questi valori e ponendo il risultato nel *top* dello stack. Questo valore rappresenta l'Ifstatus del nodo che viene inviato a Tarzan il quale provvede ad eseguire l'OR logico tra gli Ifstatus degli 8 nodi di calcolo che gli competono generando il segnale PBIf.

L'unità LUT (*Look Up Table*) è un dispositivo collegato alla porta bidirezionale del Register File che fornisce il primo termine di approssimazione in singola precisione per le funzioni $\log x$, $\exp x$, $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$.

3.3.3 Il dispositivo di comunicazione Cheetah

Cheetah è un dispositivo replicato in ogni PB che fornisce in *run mode* le funzionalità richieste alle comunicazioni tra i nodi di calcolo. In *system mode* esso interviene nelle operazioni di I/O, fornendo il collegamento tra l'APEChannel ed i nodi di calcolo per caricare o scaricare memorie dati e registri.

In APEmille le modalità per il trasferimento di dati tra nodi di calcolo, detto anche trasferimento remoto, sono (vedi figura 3.8):

- *Comunicazioni omogenee o rigide.* Questo è il tipo fondamentale di trasferimento remoto previsto in APEmille. Esso può essere schematiz-

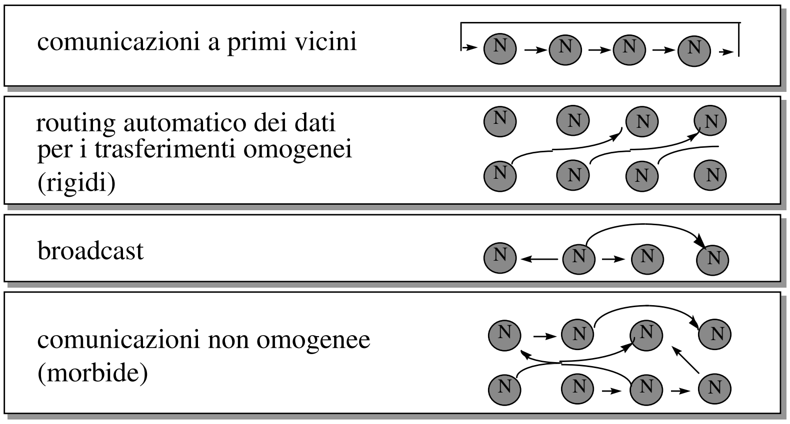


Figura 3.8: Trasferimenti remoti in APEmille

zato considerando la situazione in cui tutti i nodi inviano un pacchetto di dati ad un nodo di destinazione posto a distanza arbitraria da essi. Il nodo di destinazione viene specificato fornendo le sue coordinate relativamente al nodo sorgente. Data la natura SIMD della macchina, tutti i nodi saranno sorgenti e destinazioni del trasferimento rigido, che costituisce una estensione di quello a primi vicini implementato su APE100.

- *Broadcast.* Nella modalità broadcast due sottoinsiemi distinti dei nodi del reticolo, selezionati nel programma, diventano rispettivamente sorgente e destinazione della trasmissione. Sono previsti tre tipi di broadcast: globale, di piano o lungo un asse. Nel broadcast globale un solo nodo del reticolo è sorgente della trasmissione, tutti gli altri sono destinazione; in figura 4.2 è mostrato il percorso dei dati lungo la sezione del reticolo che corrisponde al piano contenente il nodo sorgente per la comunicazione in modalità broadcast globale. Nel broadcast di piano ogni nodo sorgente invia dati ai nodi che giacciono su uno dei tre piani (xy, xz, yz) a cui esso stesso appartiene. Infine è possibile specificare una direzione (x, y, z) lungo la quale i nodi sorgenti possono inviare i loro dati. La trasmissione broadcast utilizza il meccanismo di *pipeline*, nel senso che il nodo sorgente invia il dato senza attendere che il dato inviato precedentemente sia giunto a destinazione. In questo modo la banda passante per le comunicazioni broadcast è la stessa che per quelle a primi vicini. Il tempo di latenza, ovvero il tempo necessario affinché il primo dato inviato giunga a destinazione, dipende però dalle dimensioni del reticolo dei nodi. Con un reticolo di dimensioni (N_x, N_y, N_z) in un broadcast globale il dato dovrà viaggiare infatti attraverso un numero $N_x + N_y + N_z$ di nodi prima del completamento della trasmissione.

- *Comunicazioni non omogenee o morbide*. In questa modalità di comunicazione ogni nodo sorgente seleziona indipendentemente la destinazione della trasmissione dei dati. La rete di comunicazione, ovvero i dispositivi Cheetah che la gestiscono, provvede ad instradare i dati a destinazione risolvendo tutti i possibili casi di conflitto per l'uso dei canali di comunicazione da parte dei dati che viaggiano su di essa. Il caricamento dei dati nei nodi avviene solo successivamente all'arrivo del dato più lento sul Cheetah che corrisponde al suo nodo di destinazione.

Come mostrato in figura 3.1 Cheetah è collegato direttamente agli 8 nodi di calcolo della PB e alle 6 PB ad essa adiacenti attraverso i relativi Cheetah. Il canale di collegamento al nodo è dato da due bus bidirezionali da 20 bit ognuno, mentre i canali verso le PB sono dei bus bidirezionali da 80 bit. Limitandoci a considerare solo questi segnali risulta che il dispositivo non può essere realizzato con un solo chip, dal momento che questo dovrebbe presentare un numero di *pin* dell'ordine degli 800. Un chip con un tale numero di pin è fuori dagli standard commerciali, il progetto prevede quindi una struttura *bitsliced* in cui il dispositivo è realizzato con 4 chip identici, ciascuno dei quali gestisce un quarto dei segnali complessivi.

Le comunicazioni tra i nodi di calcolo avvengono con un trasferimento di dati tra la memoria dati di un nodo sorgente remoto ed il Register File del nodo di destinazione e corrispondono ad operazioni di lettura dalla memoria da parte del nodo di destinazione. Quello che cambia rispetto al caso di una lettura dalla memoria locale è il contenuto dell'indirizzo della locazione che deve essere letta. L'indirizzo della locazione del Register File viene scritto nella parola di microcodice di Jane assieme alla sequenza di bit che identifica l'operazione di lettura, mentre il nodo remoto e l'indirizzo della locazione della sua memoria dati sono specificati dall'indirizzo globale generato da Tarzan. Come già visto nella sezione 3.3.1, il formato dell'indirizzo globale prevede due campi distinti: gli 8 bit MSB rappresentano l'indirizzo remoto che identifica il nodo sorgente, i 24 bit LSB sono usati per specificare l'indirizzo di memoria dati di questo nodo. La figura 3.9 mostra il meccanismo con cui gli 8 bit di indirizzo remoto sono usati per individuare il nodo coinvolto nel trasferimento remoto.

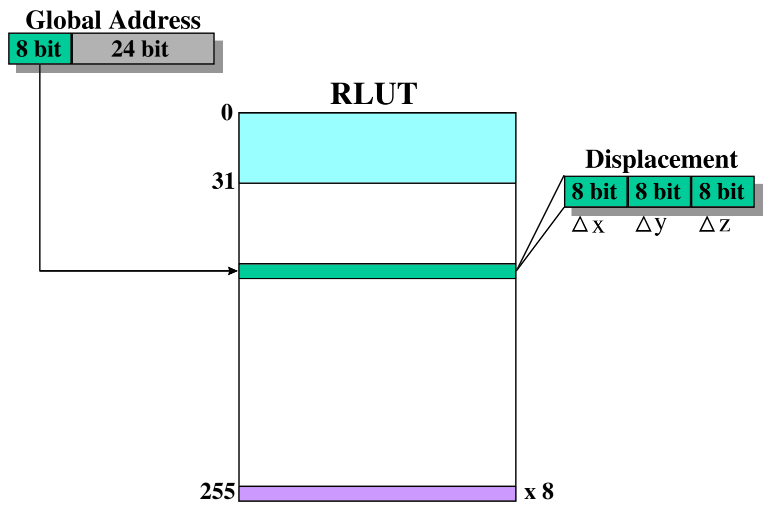


Figura 3.9: La RLUT

L'indirizzo remoto è il puntatore ad una tabella contenuta dentro Cheetah, la RLUT (*Remote Look Up Table*). Questa è formata da 256 locazioni a 24 bit contenenti i 3 campi di *displacement* a 8 bit (ΔX, ΔY, ΔZ) che identificano le coordinate del nodo sorgente relativamente a quello di destinazione nelle comunicazioni rigide, o informazioni sul tipo di broadcast da eseguire nel caso di comunicazioni di questo tipo. Le prime 32 locazioni della RLUT sono occupate da valori fissati a livello hardware. Le locazioni dalla 2 alla 19 sono occupate dai *displacement* corrispondenti ai trasferimenti rigidi usati più comunemente, come le coordinate relative dei primi vicini $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$, $(0, 0, \pm 1)$ e le cosiddette *squadre* bidimensionali $(0, \pm 1, \pm 1)$, $(\pm 1, 0, \pm 1)$, $(\pm 1, \pm 1, 0)$. Altre locazioni (dalla 20 alla 26) sono riservate per specificare il tipo di trasferimento broadcast (di linea, di piano, globale). Tra le restanti locazioni con contenuto *hardwired* si hanno la 0 che corrisponde ad una operazione nulla e la 1 che indica un finto trasferimento remoto dove il nodo sorgente e quello di destinazione coincidono, ed è utile in fase di test; le locazioni dalla 27 alla 31 sono per il momento inutilizzate. Le locazioni che vanno dalla 32 alla 254 sono scrivibili in *system mode* e sono destinate a contenere *displacement* costanti, diversi da quelli predefiniti, che possono essere utili nei programmi quando si effettuino dei trasferimenti rigidi. Questi *displacement* costanti, noti a tempo di compilazione, andranno caricati nella RLUT prima dell'esecuzione del programma. Infine la locazione 255, replicata 8 volte, è scrivibile in *run mode* ed è utilizzata nei trasferimenti morbidi. Ogni nodo della PB scrive in questa locazione le coordinate relative del nodo di destinazione.

3.4 Il software di sistema

3.4.1 Il simulatore

Nel progetto di APEmille il simulatore funzionale della macchina riveste un ruolo chiave, dal momento che consente di verificare rapidamente gli effetti delle scelte progettuali che vengono effettuate. Il simulatore Jungle, scritto in C++, descrive a livello funzionale APEmille ed i sottosistemi di cui esso è composto. L'architettura di Jungle è, secondo il paradigma della programmazione ad oggetti proprio del C++, basata su una libreria di classi che descrivono le varie componenti di APEmille secondo una struttura gerarchica. A partire da una classe generale che rappresenta l'intero sistema, questo può essere scomposto nelle sue componenti derivando le classi che le descrivono a partire dalla classe più generale; iterando questo procedimento si può aumentare l'accuratezza della descrizione del sistema.

Lo stesso sistema, ad esempio un processore aritmetico, può essere rappresentato nel simulatore in due modalità differenti. La prima è una rappresentazione di alto livello contenente delle funzioni matematiche che legano le uscite agli ingressi del sistema e alle sue variabili di stato. Il secondo tipo di rappresentazione possibile prevede una descrizione dettagliata dei sottosistemi (il Register File, la ALU etc.) e delle loro interazioni, ed è realizzata con il meccanismo illustrato precedentemente. Il primo tipo di rappresentazione risulta utile quando si voglia simulare, impiegando tempi ragionevoli, l'interazione di più sistemi dotati di una struttura interna complicata. Il secondo tipo di rappresentazione consente, al costo di tempi di simulazione più elevati, di verificare l'architettura interna dei dispositivi e di generare dei *test vector* (file contenenti gli output dei dispositivi in funzione del ciclo di clock) da confrontare con quelli ottenuti dal simulatore comportamentale scritto in VHDL e parzialmente già sviluppato.

Allo stato attuale Jungle è in grado di simulare il funzionamento di una PB incluse le operazioni di I/O tra questa e l'APEChannel. Il simulatore è corredato dal sistema operativo e dal compilatore, al fine di permettere la scrittura in memoria di dati e programmi, l'esecuzione di questi e la lettura dei risultati. Singoli blocchi di codice scritto in linguaggio TAOmille sono stati compilati ottenendo microcodice che caricato nella memoria programma simulata è stato eseguito con successo dal simulatore. A partire da codice scritto nell' assembly di APEmille, è stato possibile verificare il comportamento della catena assemblatore - sistema operativo - simulatore su programmi di test più complessi.

3.4.2 La catena di compilazione

TAOmille (cfr. § 4.2.1 e § 4.2.2) è il linguaggio di programmazione di alto livello progettato per lo sviluppo delle applicazioni su APEmille.

Il compito di tradurre i programmi scritti in TAOmille nelle corrispondenti istruzioni di microcodice eseguibile da APEmille è affidato ad una sequenza di quattro moduli software, che nel loro insieme vengono denominati catena di compilazione (vedi figura 3.10).

Il compilatore TAOmille costituisce l'argomento di questa tesi ed è descritto in dettaglio nel capitolo 4. Esso riceve in ingresso un file contenente un programma scritto in linguaggio TAOmille e dopo averlo analizzato ne produce una rappresentazione in assembly nel file *.jasm*. Tale rappresentazione è

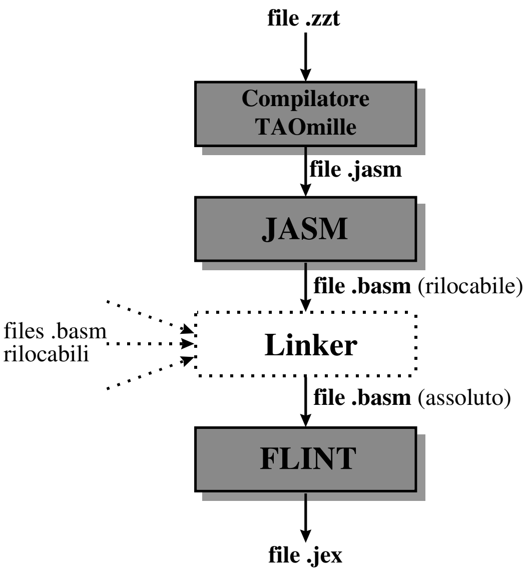


Figura 3.10: La catena di compilazione

ottenuta effettuando anche una serie di ottimizzazioni *hardware-independent* che mirano a ridurre i tempi di esecuzione del programma.

Il file *.jasm* contenente il codice assembly viene letto dal modulo JASM, l'*assembler*, che dopo aver sostituito ai simboli il loro valore ed aver espanso le macroistruzioni, traduce le istruzioni assembly in un formato intermedio binario (codice oggetto). Ad ogni istruzione assembly viene così a corrispondere un record contenente il codice operativo dell'istruzione, i suoi operandi ed una serie di informazioni riguardo alla rilocabilità che verranno utilizzate dal linker. I record prodotti vengono scritti in un file con estensione *.basm* che a questo livello contiene codice oggetto rilocabile.

Il linker può entrare in modo opzionale nel processo di compilazione ed è in grado di produrre un unico file oggetto *.basm* a partire da diversi file *.basm* rilocabili. Il linker riloca i simboli tenendo conto delle informazioni inserite nel codice assembly (vedi tabella 4.5), cosicchè nel file *.basm* che esso genera tutti gli indirizzi sono assoluti.

FLINT è il modulo che genera, corrispondentemente al codice oggetto che riceve in ingresso, le parole VLIW di microcodice di APEmille. Ogni parola di microcodice è data dall'unione delle due *control word* che contengono le microistruzioni per il controllo delle *device* interne al processore aritmetico Jane e all'APEcontroller.

FLINT effettua una ottimizzazione *hardware-dependent* del codice eseguibile per mezzo di un algoritmo che ne riduce la lunghezza basandosi sull'architettura VLIW dei processori di APEmille. Questa consente di avere, a livello di codice eseguibile, il controllo totale su tutti i dispositivi interni ai due processori; essendo questi privi di moduli che effettuino la decodifica del microcodice vi è infatti una corrispondenza biunivoca tra i campi di bit che costituiscono la parola di microcodice ed i dispositivi da essi controllati.

L'ottimizzazione viene realizzata alterando l'ordine originale delle istruzioni assembly compatibilmente con i loro vincoli di parentela che mantengono l'ordinamento necessario a preservare la semantica del programma. L'algoritmo prende in considerazione una sezione (o blocco) di codice alla volta; la sezione corrisponde a livello di assembly ad una sequenza di istruzioni comprese tra due LABEL e ad alto livello ad una parte del programma in cui il flusso sia sequenziale. Lo scopo a cui tende l'algoritmo è di fare in modo che in un dato istante tutte le *device* siano attive, mantenendo piene le *pipeline* di calcolo e sfruttando al massimo le risorse dei processori.

Le operazioni svolte da FLINT sono possono essere raggruppate nelle seguenti fasi:

- *Inizializzazione delle tavole.*

  FLINT fa uso di un insieme di strutture dati per memorizzare i dati necessari ad interpretare ed elaborare il codice oggetto. Queste strutture dati, che vengono inizializzate in questa fase, contengono le temporizzazioni delle istruzioni, ovvero i *pattern di occupazione*, cioè l'informazione su quali *device* intervengono nell'esecuzione delle istruzioni stesse. In figura 3.11 è mostrata la tabella di temporizzazione dell'istruzione TADD che esegue la somma tra due interi sull'APEcontroller.

  Sull'asse verticale sono disposti i campi della parola VLIW di microcodice dell'APEcontroller, sull'asse orizzontale sono indicati i cicli di clock a partire da quello di inizio della esecuzione dell'istruzione. La figura mostra le *device* dell'APEcontroller occupate nell'esecuzione della operazione ai vari cicli di clock: nel primo ciclo vengono forniti gli indirizzi dei due operandi della somma al Register File; nel ciclo successivo viene inviato alla ALU il codice relativo al tipo di operazione da svolgere (add); al ciclo 3 viene inviato al Register File l'indirizzo in cui andrà salvato il risultato della somma ed inoltre ingresso alla ALU vengono posti gli operandi

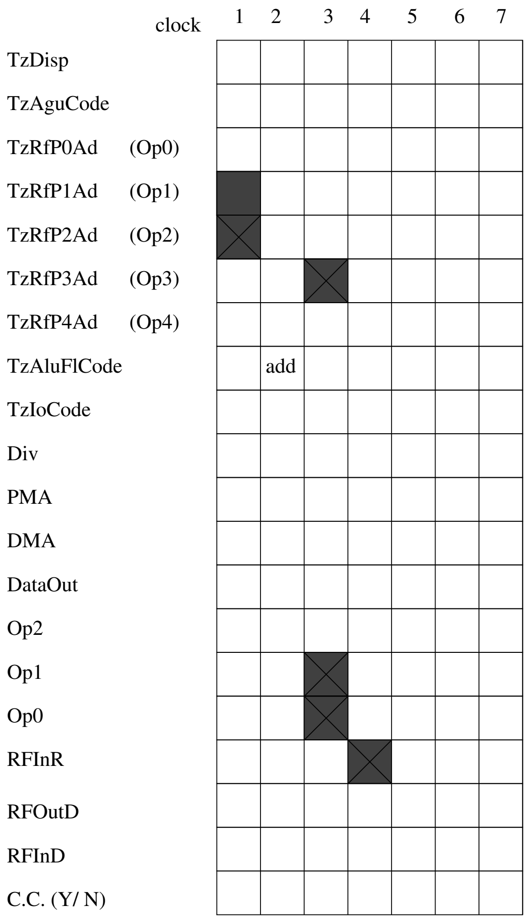


Figura 3.11: Esempio di tabella di temporizzazione di una istruzione

prelevati dal Register File; nel ciclo 4 il risultato dell'operazione viene scritto nel Register File.

- *Generazione delle parentele.*

  Viene generato un grafo che rappresenta le relazioni di parentela tra le istruzioni all'interno di una sezione. Una parentela viene generata tenendo conto che:

  1. Non è possibile eseguire una istruzione che comporti la lettura da una *device* se prima non sono state eseguite tutte le istruzioni che la precedono nel codice e che scrivono quella *device* (scrittori). In questo caso l'istruzione che effettua la lettura diventa figlia di tutti gli scrittori.

  2. Non è possibile eseguire una istruzione che comporti la scrittura su una *device* se prima non sono state eseguite tutte le istruzioni che la precedono nel codice e che leggono quella *device* (lettori). Analogamente al caso precedente l'istruzione che effettua la scrittura diventa figlia di tutti i lettori.

  Per ogni coppia di istruzioni che hanno una parentela viene calcolato, in base alle tabelle di temporizzazione, l'intervallo minimo di cicli di clock che deve separare l'esecuzione delle due istruzioni.

- *Temporizzazione.*

A questo livello FLINT tenendo conto delle informazioni sulle temporizzazioni delle singole istruzioni e sulle loro relazioni di parentela cerca, con un meccanismo che ricorda il gioco del Tetris, di "incastrare" i *pattern* di occupazione relativi alle istruzioni nel modo più compatto possibile. Questo avviene applicando successivamente le due operazioni di *shake-up* e *shake-down* ai pattern di occupazione, come mostrato in figura 3.12.

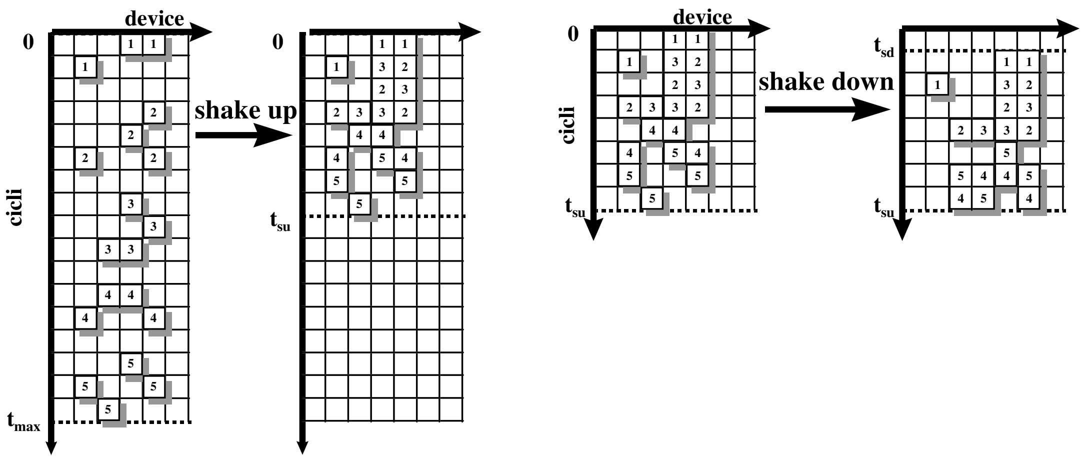


Figura 3.12: Compattamento del microcodice

Lo *shake-up* viene effettuato temporizzando le istruzioni in maniera da farle eseguire il prima possibile. Analogamente nello *shake-down* si cerca di temporizzare le istruzioni cercando di farle eseguire il più tardi possibile. In entrambe le operazioni si tiene conto dei vincoli imposti dalle parentele, che essenzialmente rappresentano la minima distanza temporale (espressa in cicli macchina) che deve essere mantenuta in fase di temporizzazione. L'effetto delle due operazioni è di compattare il microcodice: si riducono così i tempi di esecuzione del programma e si mantengono piene le *pipeline* di calcolo dei processori. Nell'esempio di figura 3.12, i tempi di esecuzione della sezione di programma, espressi in cicli macchina, si riducono da t_{max} a $t_{su} - t_{sd}$.

- *Allocazione dei registri.*

  Il compilatore di alto livello genera codice assembly nel quale figurano come operandi delle istruzioni registri *virtuali* (cfr. § 4.3.1). Questi sono rappresentati da numeri interi che non corrispondono ai registri *fisici* dei processori. In particolare il compilatore non tiene conto della limitatezza del numero di registri fisici disponibili, demandando il compito della allocazione dei registri virtuali nei registri fisici a questo modulo.

  A causa del fatto che il compilatore di alto livello alloca in ciascun blocco di programma le variabili nei registri *virtuali*, la situazione in cui ci si trovi con una occupazione elevata del Register File sarà prevedibilmente piuttosto frequente, data anche la natura delle applicazioni che saranno sviluppate su APEmille. Per ovviare a questo tipo di problema questo modulo effettua una allocazione dinamica dei registri: viene fatta una analisi dei tempi di nascita e di morte di ciascun registro virtuale, ovvero vengono rilevati i cicli macchina in cui esso viene scritto per la prima volta ed in cui viene letto per l'ultima dalle istruzioni del blocco. Ciò è possibile dal momento che a questo livello la temporizzazione del microcodice è già stata effettuata, ed è quindi nota la sequenza temporale delle istruzioni che verranno eseguite. Queste informazioni consentono di allocare più registri virtuali nello stesso registro fisico all'interno del medesimo blocco di programma, considerando libero un registro fisico nel momento in cui muore il corrispondente registro virtuale.

Allo stato attuale è implementata una strategia piuttosto elementare per il *recovery* dalla situazione di completa occupazione del Register File di Jane. Questa consiste nel dividere in due parti la sezione di codice dove si verifica il problema. Sono allo studio strategie più efficienti per lo *spilling* dei registri in memoria [15, 16, 17] (salvataggio e successiva lettura di un certo numero di registri in memoria, al fine di liberare locazioni nel Register File).

3.4.3 Il sistema operativo

Il sistema operativo (S.O.) di APEmille fornisce i seguenti servizi:

1. Gestione centralizzata della rete di PC host, tramite cui è possibile utilizzare la *Control Network* come un unico sistema, consentendo la condivisione delle risorse distribuite sulla rete.

2. Riconoscimento ed esecuzione delle richieste di servizio inoltrate dal *kernel sincrono* della macchina, quali ad esempio operazioni di I/0.

3. Interfaccia utente che permette a questi di impartire sul PC master comandi atti ad effettuare operazioni quali il caricamento sulle memorie del *kernel* sincrono dei dati e dei programmi, l'esecuzione dei programmi, la loro interruzione, il loro *debugging*, lo scaricamento delle memorie dati verso dispositivi di memoria di massa.

4. Gestione degli errori

Il S.O. di APEmille è scritto in C++, la sua struttura può essere schematizzata in tre livelli: alto, medio e basso livello. Il S.O. è distribuito sulla rete degli host, nel senso che le parti di codice relative ai livelli basso e medio sono replicate su ogni ACP (host PC Linux) che controlla una AU, ovvero 4 PB. Il codice del livello alto del S.O. risiede invece solo sul PC master della *Control Network*.

Il codice di basso livello consiste nel *driver* per la scheda APEnic di interfaccia tra l'ACP e l'APEChannel. Questo *driver* è scritto in modo tale da essere incluso nel *kernel* Linux (cfr. § 4.1.1) che è installato sull'ACP. A questo livello è inoltre definita una libreria C++ di classi che incapsula il *driver* per APEnic, interfacciandolo con il medio livello.

La parte di medio livello del sistema operativo usa il codice di basso livello per offrire, a livello di singola ACP, due funzionalità. La prima è la gestione delle *system service* richiamate nel programma TAOmille, ad esempio l'apertura di un file. La seconda è il supporto ai comandi impartiti dall'utente, provenienti dal PC master tramite la rete.

Il codice di alto livello comprende una libreria di classi C++ per la gestione di APEmille nella sua totalità; questa assicura la sincronizzazione del funzionamento delle varie AU. A questo scopo il codice provvede alla gestione della rete degli host, definendo il protocollo di comunicazione tra il PC master e la rete degli host ACP.

Capitolo 4

Il Compilatore TAOmille

Il linguaggio TAOmille rappresenta l'estensione del linguaggio TAO usato per APE100 (cfr. § 3.1), del quale mantiene tutte le caratteristiche fondamentali e a cui sono aggiunti nuovi tipi di dato e nuovi costrutti che riflettono le innovazioni apportate all'architettura della macchina.

Il compilatore, cioè il programma che traduce codice scritto in linguaggio TAOmille in codice assembly di APEmille, è stato invece completamente ripensato. Oltre ad estendere la sintassi di TAO e a produrre codice scritto con un diverso insieme di istruzioni assembly, il compilatore è in grado, attraverso la generazione e l'elaborazione di una rappresentazione intermedia del codice sorgente, di effettuare delle ottimizzazioni volte a ridurre i tempi di esecuzione dei programmi.

La definizione del formato e la successiva realizzazione delle funzioni di interfacciamento (*compiler kernel*) tra il *parser* Z_Z ed il modulo del compilatore che produce ed elabora la rappresentazione intermedia del codice di alto livello (*Pico*), la progettazione e realizzazione di questo e del modulo che genera il codice assembly a partire dalla rappresentazione intermedia (*back-end*), sono i contributi di questo lavoro e vengono descritti in dettaglio nelle sezioni 4.5, 4.6, 4.7 e refs:tipi$_{s}trut$.

Un vincolo essenziale fissato come specifica del progetto, e rispettato nella successiva fase di realizzazione, è stato quello relativo alla piena compatibilità del compilatore con il codice TAO scritto per APE100. In questo modo si potranno ricompilare programmi e librerie già sviluppati e, senza dover apportare modifiche ai sorgenti, ottenenere codice eseguibile per APEmille.

4.1 L'Ambiente di sviluppo

In questo paragrafo vengono descritti l'ambiente operativo e gli strumenti software utilizzati per la realizzazione del compilatore.

4.1.1 Il sistema operativo Linux

Linux è una implementazione di pubblico dominio del sistema operativo Unix disponibile per calcolatori basati su microprocessori Intel 386/486/Pentium, Dec alpha, Sun sparc, Mips, Motorola 68k, Power Pc, ed è per questo in grado di garantire un alto livello di portabilità delle applicazioni sviluppate su architetture diverse da quella originaria. Linux è organizzato in moduli (per la gestione di periferiche, per il supporto dei protocolli di rete etc.), in maniera da poter configuare il *kernel* del sistema operativo scegliendo quali di questi includervi.

Nel progetto APEmille è stato scelto Linux come ambiente per lo sviluppo di tutto il software di sistema (compilatore, sistema operativo, *debugger*, *profiler*); Linux sarà utilizzato anche come sistema operativo della rete di pc *host* (cfr. § 3.2). Linux ha offerto tutti gli strumenti necessari allo sviluppo del compilatore, dal momento che in questo ambiente sono disponibili strumenti quali *editor* di testi, compilatore C++ con relativi *debugger* simbolico e *profiler*, gestore di revisioni del software oltre che il *parser dinamico* Z_Z descritto nel paragrafo seguente.

4.1.2 Il parser dinamico Z_Z

Z_Z [18, 19] è un *parser* progettato e sviluppato all'interno del gruppo APE dell'INFN come strumento per lo sviluppo di compilatori ed altro software di sistema per i calcolatori della famiglia APE. La sua caratteristica fondamentale è che la grammatica su cui esso si basa può essere modificata, durante la fase di analisi di un programma, per mezzo di regole inserite nel programma stesso. Per questa ragione Z_Z viene definito *parser dinamico*, perché il linguaggio che esso è in grado di riconoscere viene modificato durante il *parsing*.

Per scrivere un compilatore si devono poter definire la struttura sintattica del linguaggio sorgente e la sua semantica, cioè il significato di ogni costrutto e il modo di tradurlo in termini del linguaggio di destinazione. Un problema che si incontra in questa fase è dato dal fatto che alcune istruzioni nel programma sorgente possono creare uno specifico contesto in cui vengono modificate la sintassi e la semantica riconosciute per le istruzioni che seguono. Esempi di questo tipo di istruzioni sono la dichiarazione di variabili e la descrizione della lista dei parametri per una procedura. In questi casi bisogna effettuare dei controlli semantici, ovvero bisogna verificare che le variabili siano usate solo dopo la loro dichiarazione, e che nelle chiamate a procedura il numero ed il tipo dei parametri sia quello specificato nella dichiarazione della stessa. Da queste considerazioni si capisce come la sintassi dei comuni linguaggi di programmazione non può essere completamente descritta da una grammatica libera dal contesto (cfr. § 2.1.2). Quindi solitamente questi linguaggi vengono descritti in due fasi: la prima dà una definizione puramente sintattica, la seconda è una descrizione meno formale che accomuna sintassi e semantica. Con Z_Z è invece possibile descrivere un linguaggio di programmazione fornendo una descrizione puramente sintattica anche per le parti di grammatica dipendenti dal contesto. Questo viene realizzato introducendo grammatiche libere che evolvono e si adattano al contesto creato dal programma. In questa maniera vengono risolti anche il controllo sui tipi e l'*overloading* degli operatori. Alla fine del paragrafo viene fatto un esempio di come questo venga realizzato usando Z_Z .

Z_Z è un *parser* in grado di estendere la propria grammatica per mezzo di regole di produzione espresse in formato simile al Backus-Naur Form (cfr.

§ 2.1.1) all'interno del programma. Alla partenza Z_Z è configurato con un insieme di regole di produzione che definiscono la grammatica del linguaggio $Z_Z L_0$ (linguaggio di livello 0). Questo prevede un ristretto numero di istruzioni e di tipi di dato usati per le variabili di Z_Z quali numeri interi, floating point, caratteri, stringhe, identificatori e liste formate da elementi di questi tipi. Inoltre viene riconosciuta la sintassi necessaria a definire una nuova regola di produzione.

$Z_Z L_0$ prevede istruzioni per la manipolazione dei tipi di dato predefiniti quali assegnazione a variabile (che dichiara in maniera dinamica la variabile stessa), valutazione di espressioni aritmetiche, stampa su standard output; ad esempio è possibile scrivere il seguente codice:

```
/stringa = "qstring"
/intero = 1
/carattere = _
/reale = 3.14 *2
/print stringa & carattere & intero & carattere & reale
```

ottenendo in uscita sullo standard output:

```
qstring_1_6.28
```

Dove il carattere & svolge il ruolo di operatore di concatenazione e alle variabili è stato dato per semplicità il nome corrispondente al loro tipo.

Ciò che rende Z_Z *dinamico* e la grammatica da esso riconosciuta *evolvente* è l'istruzione per l'estensione della sintassi, la cui forma è:

```
/sintagma -> frase { azione }
```

Dove il *sintagma* (o parte sinistra della produzione) è un simbolo non terminale della grammatica (cfr. § 2.1.1), la *frase* (o parte destra della produzione) è una sequenza di simboli terminali e non terminali e l'*azione*, la cui presenza è opzionale, è una sequenza di istruzioni che vengono eseguite in occasione del riconoscimento della *frase* come appartenente alla grammatica. Come esempio di questa istruzione consideriamo la seguente linea di codice estratta dal *front-end* del compilatore Taomille (cfr. § 4.5):

```
/stat -> var^v "=" expr^e { move nod(e) rap(v) }
```

In questo caso il *sintagma* **stat** indica che stiamo definendo una nuova istruzione (*statement*) la cui sintassi è espressa dalla parte destra della regola di produzione. Questa consiste in un simbolo non terminale **var** seguito dal terminale (o *token*) = e dal non terminale **expr**; questa sintassi specifica l'assegnazione a una variabile e sarà riconoscibile nel codice immediatamente successivo all'istruzione che la definisce. In generale l'*azione* consiste nell'esecuzione di una sequenza di istruzioni Z_Z appartenenti a Z_ZL_0 o la cui sintassi sia stata definita nel codice precedente l'istruzione in esame, od a chiamate di funzioni C o C++. Nell'esempio l'*azione* consiste nell'istruzione "move" il cui effetto è di richiamare una funzione C++ passandole i parametri opportuni.

Un'applicazione Z_Z si ottiene unendo alle regole di base che definiscono $Z_Z L_0$ un *kernel* di regole scritto dall'utente. In questo modo Z_Z viene configurato, estendendo la grammatica del linguaggio che esso riconosce e permettendo la chiamata a funzioni C o C++. Nel paragrafo 4.5 si descriverà con maggiore dettaglio l'applicazione Z_Z usata come *front-end* per il compilatore Taomille.

In conclusione di questo paragrafo, forniamo un esempio di come Z_Z sia in grado di effetture a livello sintattico quei controlli che in un compilatore tradizionale vengono risolti a livello semantico. Consideriamo l'istruzione di dichiarazione di una variabile intera in un certo linguaggio di programmazione che la preveda:

```
integer i
```

Questa istruzione genera un contesto nella parte successiva del programma, in cui la grammatica dovrà prevedere che il simbolo **i** sarà un "buon intero". Per descrivere questo, le istruzioni di dichiarazione possono essere descritte con Z_Z in questo modo:

```
/stat -> integer ident^var_name {
...
/integer_var -> var_name
...
}
```

Dove si è omesso il codice che in genere è necessario a gestire una variabile, ad esempio se la variabile è allocata in memoria bisognerà assegnarle un indirizzo. L'aspetto interessante è che nell'*azione* è presente un'istruzione per l'estensione della sintassi. Quando nel programma viene incontrata l'istruzione di dichiarazione "integer i", Z_Z esegue l'*azione* attaccata alla *frase* generando la regola di produzione:

```
integer_var -> i
```

In questo modo Z_Z saprà riconoscere il simbolo **i** come variabile intera nella parte di programma successiva all'istruzione di dichiarazione. In maniera analoga si risolvono il controllo sui tipi nelle espressioni, il passaggio dei parametri e l'*overloading* degli operatori.

4.1.3 Il linguaggio C++

Il C++ è un linguaggio di uso generale che si basa sul linguaggio C [20, 21, 22]. Il C++ aggiunge alle funzionalità del C le classi, le funzioni in linea, l'*overload* degli operatori predefiniti, l'*overload* del nome delle funzioni, i tipi costanti, il controllo sugli argomenti di una funzione e sulla conversione dei tipi.

Il C++ supporta l'astrazione dei dati e la programmazione orientata agli oggetti. Per supporto all'astrazione dei dati si intende la presenza di costrutti nel linguaggio che consentano al programmatore di definire nuovi tipi di dato e di trattarli alla stregua di quelli predefiniti. Nel linguaggio C++ questo è ottenibile per mezzo della definizione delle classi che descrivono i tipi di dato astratto e degli operatori che agiscono su questi. Quando si parla di supporto alla programmazione orientata agli oggetti si intende, accanto all'uso delle classi, la possibilità di esplicitare la similitudine tra le varie classi per mezzo di meccanismi quali l'ereditarietà ed il polimorfismo. Nella sezioni 4.6 e 4.7 descrivendo l'implementazione del modulo PICO e del generatore di assembly verranno fatti degli esempi di uso di queste proprietà del linguaggio.

4.2 Il linguaggio TAOmille

Il linguaggio TAOmille è l'estensione del linguaggio TAO usato per APE100. In esso si ritrovano nuove funzionalità e nuovi tipi di dato necessari per utilizzare agevolmente le nuove potenzialità introdotte nell'architettura di APEmille. TAOmille è pienamente compatibile con la precedente versione del linguaggio, del quale mantiene tutti i costrutti e le caratteristiche fondamentali.

4.2.1 TAO

TAO [23] è il linguaggio di programmazione VHLL (*Very High Level Language*) adottato per il calcolatore APE100. TAO è un linguaggio *incrementale* e supporta l'astrazione dei dati. La sua sintassi di base è molto simile a quella del Fortran e comprende costrutti specifici per il supporto alla programmazione parallela. Grazie all'architettura SIMD di APE100 è stato possibile limitare il numero di tali costrutti "paralleli". Questi riguardano il controllo del flusso del programma, l'allocazione dei dati e l'accesso ai dati su nodi remoti. TAO si presta ad implementare in modo efficiente algoritmi in cui sia presente un parallelismo che si adatta bene all'architettura SIMD della macchina; questi coincidono con quelli adottati per la soluzione di problemi relativi a sistemi che presentino un comportamento omogeneo e locale. L'omogeneità comporta che i dati usati per rappresentare il sistema in esame possano essere partizionati in sottoinsiemi allocati su diverse unità di calcolo sulle quali eseguire lo stesso algoritmo. La località implica che le varie unità non si scambino dati se non tra primi vicini.

In TAO sono presenti due tipi di dato predefiniti: il tipo intero (*integer*) ed il tipo floating-point a singola precisione (*real*); il tipo complex, disponibile senza bisogno di includere particolari librerie, non è altro che un coppia di *real* allocati in locazioni di memoria di nodo contigue. Il tipo intero e i tipi floating point svolgono ruoli ben distinti: il tipo intero è usato per la rappresentazione delle variabili globali allocate fisicamente sul controllore, i tipi floating-point per la rappresentazione delle variabili allocate sui nodi di calcolo. Dal punto di vista del programmatore, esiste quindi una sola copia di una variabile globale mentre una variabile di nodo è replicata su ciascun nodo di calcolo. Le variabili intere vengono usate in APE100 essenzialmente per il controllo del flusso del programma (ad esempio come indici di un ciclo) oppure come indici di matrici: ciò comporta che tutti i nodi di calcolo usino dati posti agli stessi indirizzi fisici della loro memoria. Le variabili di nodo sono invece usate per descrivere le grandezze fisiche del problema in esame.

Il tipo *complex*, come già detto, è formato da due dati floating-point a singola precisione che rappresentano le parti reale ed immaginaria di un numero complesso. Per le variabili *complex* sono definiti tutti gli operatori aritmetici, oltre ad alcuni operatori specifici per il tipo. Questi sono gli operatori postfissi di estrazione *.re* e *.im* che permettono di accedere rispettivamente alla parte reale ed a quella immaginaria di una variabile *complex*, e l'operatore binario circonfisso $(op1, op2)$ che restituisce una variabile *complex* con parte reale data da $op1$ e parte immaginaria data da $op2$; con questo operatore si costruiscono variabili *complex* a partire da variabili *real*.

TAO è in grado di gestire strutture dati quali ARRAY, MATRIX e RECORD. Gli ARRAY sono usati per rappresentare vettori e matrici multidimensionali i cui elementi possono essere dati di qualsiasi tipo, anche definito dall'utente per mezzo dei costrutti MATRIX e RECORD. Consideriamo il seguente segmento di programma TAO (esempio 4.2.1):

```
record v3d
  real x, y, z
end record
record particle
  v3d position, speed
end record
constant MAXSIZE = 100
particle cur_particle, particles[MAXSIZE]
```

```
DO i = 0 to particle_max_n-1
   cur_particle = particles[i]
   ...
ENDDO
```

Le variabili globali i e *particle_max_n* vengono usate per gestire il ciclo DO e per indirizzare il vettore monodimensionale particles che, come la variabile *cur_particle*, è invece allocato sulla memoria di ciascun nodo. Il tipo delle variabili di nodo di questo esempio è stato definito dall'utente per mezzo dei RECORD. Si noti che un RECORD può essere definito in termini di altri RECORD, e che una volta definito esso può essere usato come elemento di vettori e matrici.

Un altro modo per definire matrici monodimensionali e bidimensionali è l'uso delle MATRIX. L'esistenza di questo tipo di struttura dati è legata al fatto che per mezzo di essa possono essere fatte ottimizzazioni sui tempi di esecuzione dei programmi. Questo compito, che nel compilatore TAO è demandato al programmatore, viene eseguito per mezzo di istruzioni che permettono di leggere o scrivere in memoria blocchi di dati consecutivi, riducendo così i tempi di latenza (*fast I/O*). Un'altra fonte di ottimizzazione deriva dall'uso congiunto delle MATRIX con il comando /FOR, che consente di replicare una parte di codice un certo numero di volte all'interno del programma TAO. In questo modo si evita l'uso dei cicli che comportano l'esecuzione di istruzioni di salto in memoria programma e che svuotano la *pipeline* dei nodi di calcolo.

La distinzione tra variabli globali e variabili di nodo comporta l'esistenza di due diversi tipi di condizioni logiche che possono essere espresse all'interno di un programma TAO. Una condizione è globale se il suo valore logico dipende da variabili globali (intere), è locale se il suo valore logico dipende da variabili di nodo (floating-point):

```
integer i
real error
...(i < 100)...          !! condizione globale
...(error < 0.01)...     !! condizione locale
```

Una condizione locale può essere resa globale per mezzo degli operatori unari ANY, ALL e NONE:

```
(ANY(error < 0.01)) !! la condizione (globale) e' vera se ALMENO
                    !! su UN nodo di calcolo error < 0.01
(ALL(error < 0.01)) !! la condizione (globale) e' vera se
                    !! su TUTTI i nodi di calcolo error < 0.01
(NONE(error < 0.01)) !! la condizione (globale) e' vera se su
                     !! NESSUNO dei nodi di calcolo error < 0.01
```

Con l'uso di questi operatori diventa possibile combinare condizioni globali con condizioni locali:

```
((i < 100) && ANY(error > 0.01))
```

In questo esempio la condizione logica è vera se la variabile globale i ha un valore minore di 100 e se su un qualsiasi nodo di calcolo la variabile $error$ contiene un valore maggiore di 0.01 .

TAO prevede istruzioni per il controllo del flusso del programmma del tutto simili a quelle dei comuni linguaggi di programmazione quali i costrutti iterativi:

DO i=inizio,fine[,passo] blocco **ENDDO**	**WHILE**(cond_globale) blocco **ENDWHILE**	**REPEAT** blocco **AS LONG AS**(cond_globale)

il costrutto condizionale globale:

```
IF(cond_globale)
  blocco VERO
ELSE
  blocco FALSO
ENDIF
```

ed un costrutto specifico che permette di controllare il flusso del programma su ciascun nodo separatamente in base alla valutazione della condizione locale espressa:

```
WHERE(cond_locale)
  blocco VERO
ELSEWHERE
  blocco FALSO
ENDWHERE
```

Le istruzioni contenute nel blocco VERO sono eseguite su quei nodi dove la condizione locale è verificata, quelle nel blocco FALSO su quei nodi dove la condizione locale non è verificata. In realtà, data l'architettura SIMD della macchina, i nodi eseguono tutte le istruzioni contenute nei due blocchi, ma la scrittura dei risultati delle operazioni nei registri del nodo viene inibita per le istruzioni del blocco “sbagliato”; in questo modo viene preservata la sincronizzazione tra i nodi. Ad esempio, nel seguente fragmento di codice:

```
real error, accept
where (error < 0.1 )
  accept = 1.0
elsewhere
  accept = 0.0
endwhere
```

alla variabile *accept* sarà assegnato il valore 1.0 in quei nodi dove la condizione locale (error $<$ 0.1) è verificata, mentre le verrà assegnato il valore 0.0 negli altri nodi.

Le coordinate di ogni nodo nel reticolo tridimensionale sono specificate da una terna di numeri interi; per una macchina in configurazione $n_1 * n_2 * n_3$ lo spazio nelle coordinate di nodo è dato da tutte le terne di numeri interi (k_1, k_2, k_3) tali che $0 \leq k_1 \leq (n_1 - 1), 0 \leq k_2 \leq (n_2 - 1), 0 \leq k_3 \leq (n_3 - 1)$. In TAO l'accesso ai dati dei nodi primi vicini, che in APE100 è l'unico trasferimento consentito, avviene per mezzo di uno speciale indirizzamento degli array. Esistono sei costanti predefinite (RIGHT, LEFT, FRONT, BACK, UP, DOWN), che permettono di specificare una delle tre direzioni possibili (X, Y, Z) ed il verso corrispondente per il movimento dei dati: se A[i] rappresenta l'i-esimo elemento del vettore A sul nodo di coordinate (k_1, k_2, k_3), A[i+RIGHT] rappresenta l'i-esimo elemento dello stesso vettore, ma sul nodo di destra, ovvero su quello di coordinate $((k_1 + 1)\ mod\ n_1, k_2, k_3)$. Come esempio di uso di questo costrutto consideriamo, per semplicità, un'ipotetica configurazione a toro bidimensionale 4*4 ($n_1 = 4,\ n_2 = 4$) ed un programma che, dopo aver immagazzinato nella variabile e il risultato di calcoli precedenti, effettui la somma di questa variabile su tutti i nodi; dopo l'esecuzione del programma, su ogni nodo la variabile e conterrà il valore della somma dei valori calcolati sui 16 nodi:

```
constant n1 = 4, n2 = 4
integer i
real e, e_array[1]
...                                !!codice per il calcolo di e
e_array[0] = e
somma = 0.0
do i = 1, n1                       !!somma sulle righe
  e_array[0] = e_array[RIGHT]
  somma = somma + e_array[0]
enddo
e_array[0] = somma
somma = 0
do i = 1, n2                       !!somma sulle colonne
  e_array[0] = e_array[BACK]
```

```
   somma = somma + e_array[0]
  enddo
  e = somma                              !!risultato finale
```

La variabile *e_array* si rende necessaria dal momento che si può accedere a dati dei nodi vicini solo se questi sono elementi di vettori. Nella figura 4.1 sono visualizzati il reticolo bidimensionale con la direzione ed il verso in cui sono trasferiti i dati all'interno dei due cicli DO.

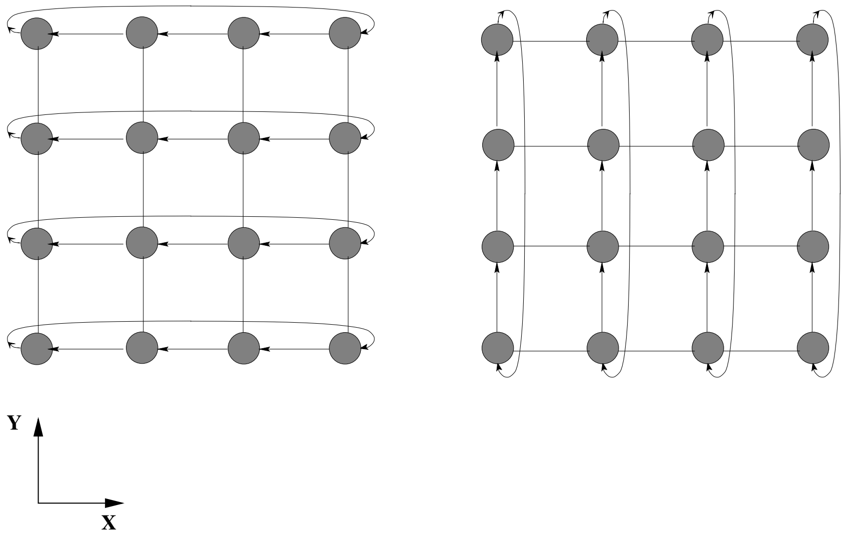


Figura 4.1: Direzione e verso dei trasferimenti rigidi internodo all'interno dei due cicli

La sintassi di TAO è definita per mezzo del *parser dinamico* Z_Z (cfr. § 4.1.2), risulta quindi agevole per il programmatore ampliare il linguaggio, che per questa ragione viene definito *incrementale*. Usando istruzioni Z_Z il programmatore TAO può definire nuovi tipi di dato e nuovi operatori oppure può sovraccaricare degli operatori già esistenti. Considerando il tipo $v3d$ definito nell'esempio 4.2.1, è possibile definire l'operatore "prodotto vettoriale" in una forma molto semplice sovraccaricando l'operatore * :

```
/v3d -> v3d^v1 "*" v3d^v2 {
temporary v3d ris
ris.x = v1.y * v2.z - v1.z * v2.y
ris.y = v1.z * v2.x - v1.x * v2.z
ris.z = v1.x * v2.y - v1.y * v2.x
return ris }
```

La prima riga della definizione descrive una regola sintattica, le linee seguenti l'azione da intraprendere quando Z_Z analizza l'espressione data (cfr. § 4.1.2). La variabile *ris* viene definita *temporary* per indicare al compilatore che essa non deve essere immagazzinata in memoria, ma che il suo uso è appunto temporaneo e che la variabile deve essere allocata solo sui registri del nodo di calcolo. Questo operatore può essere ora usato in maniera molto espressiva nel programma:

```
v3d a,b,c
a=b*c
```

Usando questa capacità del linguaggio il codice scritto diventa più comprensibile e meno soggetto ad errori da parte del programmatore. Inoltre i programmi diventano più brevi di circa un fattore 3 rispetto agli stessi scritti in FORTRAN. La documentazione relativa ai programmi può essere notevolmente ridotta dal momento che la definizione degli operatori e dei tipi risulta piuttosto eloquente. Le definizioni dei tipi di dato e degli operatori necessari in determinati campi di applicazione sono raccolti in librerie pronte per essere incluse nel programma ed usate. Ad esempio esiste la libreria di QCD che definisce tipi di dato quali matrici SU(3), matrici di Dirac, spinori etc. e gli operatori che agiscono su di essi, come il prodotto tra matrici SU(3), il prodotto di una matrice di Dirac per uno spinore etc.

4.2.2 TAOmille: l'estensione di TAO per APEmille

Le estensioni apportate al linguaggio TAO rispecchiano pienamente le innovazioni architetturali introdotte con APEmille (cfr. § 3.2).

Tipi di dato

L'architettura delle unità di calcolo JANE prevede la possibilità di effettuare operazioni aritmetiche tra numeri interi, complessi, reali in doppia precisione e tra coppie di numeri reali in singola precisione (cfr. § 3.3.2). Sono stati quindi introdotti nuovi tipi di dato predefinito:

- **localint** (numeri interi allocati sui nodi di calcolo)
- **double** (numeri floating point in doppia precisione)
- **vector** (coppia di numeri floating point in singola precisione)

Questi tipi si affiancano a quelli già presenti in TAO (cfr. § 4.2.1):

- **integer** (numeri interi allocati sul controllore)

- **real** (numeri floating point in singola precisione)

- **complex** (coppia di numeri floating point in singola precisione)

La sintassi delle istruzioni di dichiarazione e degli operatori aritmetici per questi nuovi tipi è la stessa di quella adottata per i tipi di TAO. In particolare è possibile con i nuovi tipi definire variabili *temporary*, *register* e *stack* [23]. Inoltre essi possono essere usati come elementi di array o come parti di tipi di dato definiti per mezzo delle istruzioni RECORD e MATRIX.

Il tipo *localint* rappresenta numeri interi con segno a 32 bit allocati sulla memoria e nel register file dei nodi di calcolo. L'uso di questo tipo di dato permette di accedere a due delle funzionalità introdotte con APEmille: l'indirizzamento locale ed il calcolo intero sui nodi.

Per indirizzamento locale si intende la capacità di accedere a dati differenti sui diversi nodi di calcolo della macchina. In questo modo l'unico programma che viene eseguito su tutti i nodi può operare su dati che sono diversi da nodo a nodo; questa caratteristica fa di APEmille un calcolatore parallelo con architettura SIMAMD (*Single Instruction stream-Multiple Address-Multiple Data stream*). In TAOmille l'indirizzamento locale viene specificato ponendo nell'indice di un array una espressione di variabili *localint* eventualmente sommata ad una espressione di variabili *integer* secondo la sintassi:

$indice_{array}$:

$espressione_{localint}\,[+\,espressione_{integer}]$

$espressione_{integer}\,[+\,espressione_{localint}]$

dove $espressione_{localint}$ e $espressione_{integer}$ sono costanti, variabili o espressioni aritmetiche del tipo specificato. Secondo questa sintassi sono "buoni indici" quelli formati da una espressione di *localint*, da una espressione di *integer* o dalla somma di una espressione di *localint* e di una espressione di *integer* (o il contrario). Ad esempio, se li e lj sono variabili *localint*, i e j sono di tipo *integer*, r è di tipo *real* e m è un array tridimensionale di *real*, le seguenti istruzioni sono corrette:

```
r = m[i, li, i*j+li*lj]
m[li+lj+i/j, i*2+li, i+li+1] = 1.0
```

mentre queste generano errore di sintassi da parte del compilatore:

```
r = m[i, j, i+li+j]
r = m[i, j, i*li]
```

infine la seguente istruzione non è valida a causa dell'ambiguità introdotta nella sintassi dalla rappresentazione comune delle costanti *integer* e *localint*:

```
r = m[i, j, 1+li+2]
```

Un tipico caso in cui l'indirizzamento locale risulta utile per aumentare l'efficienza degli algoritmi è quello, comune allo studio di molti sistemi fisici, in cui si richieda l'elaborazione di grandi matrici sia per righe che per colonne. Ad esempio il calcolo di una trasformata di Fourier multidimensionale (necessario per problemi di LGT, metereologia, analisi dei segnali etc.), richiede la trasposizione di una matrice i cui elementi sono disposti in sottomatrici allocate su ciascun nodo. L'indirizzamento locale consente una implementazione molto efficiente di un tale algoritmo di trasposizione su una macchina con topologia a reticolo cubico tridimensionale come APEmille [24].

Per mezzo di variabili di tipo *localint* si possono effettuare calcoli interi sui nodi. Sulle variabili di questo tipo sono definiti, oltre agli operatori aritmetici, una serie di operatori bit a bit unari e binari circonfissi che restituiscono un valore di tipo *localint* (tabella 4.1). Questi operatori *bitwise* rendono agevole

Operatori unari	
NOT(op)	complemento a 1 di op
BITCOUNT(op)	numero di bit di op a livello logico 1
LEADONE(op)	stato del MSB di op
Operatori binari	
AND(op1, op2)	and bit a bit di op1 e op2
NAND(op1, op2)	nand bit a bit di op1 e op2
OR(op1, op2)	or bit a bit di op1 e op2
NOR(op1, op2)	nor bit a bit di op1 e op2
XOR(op1, op2)	or esclusivo bit a bit di op1 e op2
XNOR(op1, op2)	nor esclusivo bit a bit di op1 e op2
Operatori di scorrimento	
SHIFT(op, n)	shift logico di n posizioni dei bit di op
ASHIFT(op, n)	shift aritmetico di n posizioni dei bit di op
ROT(op, n)	rotazione di n posizioni dei bit di op

Tabella 4.1: Operatori *bitwise* per il tipo *localint*

l'uso degli interi locali per la rappresentazione di grandezze il cui valore possa assumere un numero limitato di stati, tale da rendere conveniente la codifica di questo valore con numero n di bit piccolo rispetto a 32, ovvero alla dimensione del tipo *localint*. In questo modo una variabile *localint* può rappresentare, come vettore di bit, un insieme costituito da un numero $32/n$ di queste grandezze sulle quali effettuare dei calcoli per mezzo degli operatori *bitwise*, con evidente guadagno in velocità di esecuzione del programma e risparmio della memoria dati dei nodi di calcolo. Come esempio di algoritmo che faccia questo uso delle variabili *localint* consideriamo una simulazione Monte Carlo di un tipico sistema disordinato quale un Ising Spin Glass in n dimensioni. Nel modello si considera un reticolo d-dimensionale, dove ogni sito è descritto da uno spin $s_i = \pm 1$ e da una costante di accoppiamento con i primi vicini $J_{ik} = \pm 1$; l'energia del sito i-esimo è data da:

$$E_i = -\sum_{k=0}^{nlink-1} s_i\, J_{ik}\, s_k \tag{4.1}$$

dove $nlink = 2 \cdot d$. Nel programma lo stato dello spin viene codificato con un singolo bit: a $s_i = 1$ corrisponde il livello logico 1, a $s_i = -1$ il livello logico 0. Con un singolo dato *localint* si può quindi rappresentare lo stato dello spin di un certo sito per 32 sistemi diversi. Il seguente frammento di programma effettua parte del calcolo espresso dalla formula 4.1:

```
constant D = 3

constant NLINK = 2*D

constant VOLUME = 8*8*8
```

```
integer i, k
localint spin[VOLUME], J[VOLUME, NLINK], e_link[NLINK]
...
do i = 0, VOLUME-1
  ...
  do k = 0, NLINK-1
    e_link[k] = xnor( spin[i], xor( J[i, neighbour(i, k)], spin[k] ) )
  enddo
  ...
enddo
...
```

dove per il calcolo di $e\_link[k]$ si è fatto uso degli operatori *bitwise* or e nor esclusivi oltre che dell'operatore *neighbour()* che si assume definito in precedenza nel programma: *neighbour(i, k)* (per $0 \leq k < nlink$) restituisce un *localint* che contiene l'indice relativo all'k-esimo primo vicino del sito i-esimo. L'elemento $e\_link[k]$ contiene su ogni bit il risultato del calcolo di $s_i\ J_{ik}\ s_k$ per il sistema corrispondente. Per ottenere il valore dell'energia E_i vengono effettuate altre operazioni *bitwise* sugli $e\_link[k]$ nella successiva parte del programma non indicata nell'esempio.

Il tipo *double* rappresenta numeri floating point in doppia precisione secondo lo standard IEEE, quindi con 1 bit per il segno, 11 bit per l'esponente e 52 bit per la mantissa. Il dato *double* viene allocato su locazioni di memoria contigue e su coppie di registri adiacenti del register file. Con questo tipo di dato si possono rappresentare le variabili con una maggiore gamma dinamica e una maggiore sensibilità rispetto al caso in cui si usi il tipo floating-point a singola precisione (*real*). In particolare la maggiore sensibilità risulta di importanza primaria per quegli algoritmi in cui l'arrotondamento derivante dalla precisione finita della macchina può portare alla non convergenza di un metodo di calcolo o comunque a risultati sbagliati. Un'altra situazione, meno ovvia della precedente, in cui l'uso del tipo *double* è risolutivo, è quella in cui si voglia determinare empiricamente la dipendenza dei risultati forniti da un certo programma dalla precisione con cui vengono rappresentate le sue variabili. Ciò si può realizzare comparando i risultati forniti dal programma nei due casi in cui si rappresentino le variabili una volta con il tipo *real* e l'altra con il tipo *double*. Sulla base di questo confronto risulta agevole scegliere il tipo appropriato per le variabili.

Il tipo *vector* consiste di due floating-point a singola precisione allocati in locazioni contigue di memoria e su registri contigui del register file. È possibile accedere e riferirsi a ciascuno dei due elementi *real* di una variabile di tipo *vector* per mezzo degli operatori postfissi di estrazione *.one* e *.two* analoghi a quelli usati per il tipo *complex*. È definito l'operatore di costruzione $\{op1, op2\}$, con $op1$ ed $op2$ espressioni di tipo *real*, che restituisce un *vector*; ad esempio:

`real r, s`	equivale a:	`real r, s`
`vector v`		`vector v`

```
v.one = r                    v = {r,s}
v.two = s
```

Variabili di tipo *vector* possono essere usate per sfruttare al massimo le capacità di calcolo in singola precisione di JANE, dal momento che essa dispone di due *pipeline* di calcolo in grado di effettuare due operazioni NORMAL in parallelo (cfr. § 3.3.2). Le operazioni aritmetiche su variabili *vector* corrispondono quindi a coppie di operazioni in singola precisione:

```
vector u, v, w   equivale a:   vector u, v, w
u = v * w                      u.one = v.one * w.one
                               u.two = v.two * w.two
```

con la differenza che nel caso in cui si usa la moltiplicazione *vector* le due moltiplicazioni vengono svolte in parallelo sulle due *pipeline* di calcolo di JANE.

In TAOmille è prevista la possibilità di effettuare la conversione di tipo per le variabili di nodo, corrispondentemente alla capacità dei nodi di calcolo di effettuarle per mezzo di speciali istruzioni di microcodice. La conversione avviene in maniera esplicita per mezzo degli operatori riportati in tabella 4.2, che agiscono su variabili o espressioni appartenenti ad un certo tipo restituendo una variabile di un altro tipo.

Da A	localint	real	double
localint		LOCALINT($espr_{real}$)	LOCALINT($espr_{double}$)
real	REAL($espr_{localint}$)		REAL($espr_{double}$)
double	DOUBLE($espr_{localint}$)	DOUBLE($espr_{real}$)	

Tabella 4.2: Operatori di Conversione tra Tipi

Per quanto riguarda la conversione tra tipo *integer* e tipo *localint* questa può essere effettuata sia in maniera esplicita che in maniera implicita. Ad una variabile *localint* può essere assegnata una qualsiasi espressione *integer* senza bisogno di una conversione esplicita usando la sintassi:

$variabile_{localint} = espressione_{integer}$

La stessa assegnazione può essere effettuata usando l'operatore di conversione:

$variabile_{localint} = \mathbf{LOCALINT}(espressione_{integer})$

Ad una variabile *integer* può essere assegnata una variabile *localint*:

$variabile_{integer} = variabile_{localint}[\mathbf{of}\ nodo_{id}]$

dove $nodo_{id}$ ha uno dei possibili formati usati in TAO per specificare un determinato nodo. Se $nodo_{id}$ è omesso alla variabile *integer* viene assegnata la variabile *localint* del nodo [0, 0, 0]; lo stesso risultato si ottiene usando la conversione esplicita:

$variabile_{integer} = \mathbf{INTEGER}(variabile_{localint})$

Istruzioni per le comunicazioni internodo

TAOmille supporta le istruzioni necessarie a gestire le modalità di trasferimento internodo *broadcast* e *morbida*, descritte nella sezione 3.3.3, accanto a quelle già previste in TAO per la gestione dei trasferimenti rigidi.

Per i trasferimenti in modalità *broadcast* si fa uso di una variabile di nodo predefinita di tipo *localint* detta *broadcaster*. Il valore di questa variabile (0 o 1) viene inizializzato dal programmatore all'interno di un WHERE per indicare quali nodi del reticolo sono sorgenti per la trasmissione dei dati. Il comando che specifica quale dato inviare, la variabile a cui assegnarlo e la modalità del *broadcast* ha la seguente sintassi:

BROADCAST $[qualificatore]$ $var_{destinazione} = var_{sorgente}$ [**:** dim]

$qualificatore$: x, y, z, xy, xz, yz, xyz.

dim: $costante_{intera}$

Il campo opzionale $qualificatore$ determina la modalità della trasmissione *broadcast*: lungo un asse, su un piano o globale; se il campo non è presente si assume che il comando indica un *broadcast* globale. Le variabili sorgente e destinazione possono essere di qualunque tipo, sia predefinito che definito dal programmatore. Il campo opzionale dim specifica, nel caso in cui src sia un elemento di un array, di un RECORD o di una MATRIX, il numero di dati del tipo di src che devono essere trasmessi in blocco a partire da questo. Nel seguente esempio alle variabili $u[0]...u[6]$ di ogni nodo vengono assegnate le variabili $v[0]...v[6]$ del nodo avente coordinate [0, 0, 0]:

```
real v[100], u[100]
broadcaster = 0            !! nessun nodo e' sorgente
where (node_abs_id == 0)
  broadcaster = 1          !! il nodo [0, 0, 0] e' sorgente
endwhere
broadcast u[0] = v[0] : 7  !! effettua la trasmissione
```

Nella figura 4.2 viene visualizzato, per semplicità sul piano $z = 0$, il percorso dei dati nella trasmissione in *broadcast* globale dell'esempio.

Per gestire i trasferimenti *morbidi* ad alto livello, TAOmille prevede due variabili di nodo predefinite di tipo *localint*, ma logicamente di tipo booleano, chiamate *received* e *net_busy*, e due istruzioni:

SEND($nodo_{id}$, $var_{sorgente}$**)**

RECEIVE($var_{destinazione}$**)**

$nodo_{id} : var_{localint}$

Il comando SEND() permette di inviare una variabile di nodo $var_{sorgente}$ di qualunque tipo al nodo specificato dalla variabile $nodo_{id}$ di tipo *localint*. Alla variabile $var_{destinazione}$, argomento del comando RECEIVE(), verrà assegnato il dato in arrivo sul nodo. L'uso di questi due comandi abbinati alle due variabili *received* e *net_busy* viene mostrato nel seguente esempio in cui si effettua

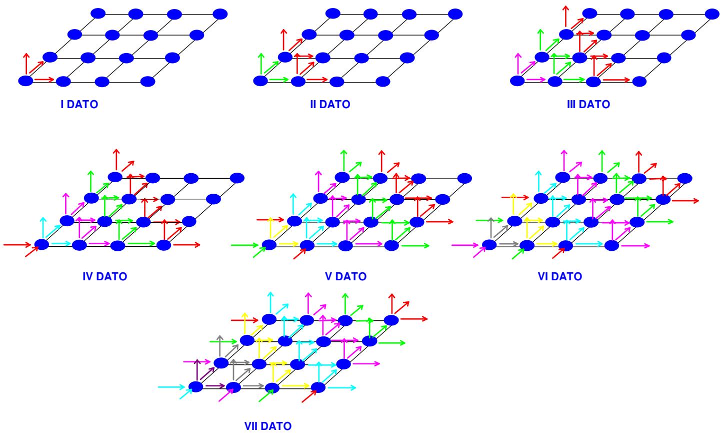


Figura 4.2: Broadcast Globale

il trasferimento *morbido* di un dato *real*; ogni nodo invia il contenuto della variabile *sent_data* sulla variabile *rec_data* del nodo indicato da *dest_addr*:

```
localint dest_addr
real sent_data, rec_data
send(dest_addr, sent_data)
!! si possono inserire altre istruzioni prima del ciclo che saranno
!! eseguite mentre la rete cerca di portare i dati a destinazione
repeat
  receive(rec_data)
  where(received)
```

```
  !! se il dato e‘ stato ricevuto dal nodo
  !! possono essere eseguite altre istruzioni, ad esempio
  !! i dati ricevuti possono essere immagazzinati in un buffer
  endwhere
as long as(any net_busy)
```

Dall'esempio appare chiaro che (any *net_busy*) e *received* sono due flag: il primo è alto fintanto che la rete di interconnessione è occupata da dati che viaggiano da un nodo all'altro, il secondo diventa alto quando un nodo riceve il dato ad esso destinato. La comunicazione *morbida* si interrompe dunque all'uscita del ciclo repeat-as long as, quando tutti i dati sono arrivati sui nodi di destinazione.

4.3 L'assembly di APEmille

Il linguaggio assembly di APEmille è composto dall'insieme delle istruzioni per il controllore, delle istruzioni per il processore aritmetico e delle istruzioni per la comunicazione tra i tre dispositivi fondamentali: controllore (TARZAN) - processore (JANE) - unità di comunicazione (CHEETAH).

L'*assembly* di APEmille segue una filosofia diversa da quella adottata per APE100. L'insieme delle istruzioni assembly di APEmille è infatti di tipo RISC (*Reduced Instruction Set Computer*), con un numero piuttosto limitato di istruzioni differenti che sono,tranne poche eccezioni, in corrispondenza uno ad uno con le istruzioni di microcodice della macchina. In questo modo l'assemblatore (cfr. § 3.4.2) si limita a tradurre ogni istruzione assembly nella corrispondente sequenza di bit che costituisce la parola VLIW (*Very Long Instruction Word*) del microcodice di APEmille, a differenza di quanto avviene per APE100 dove le istruzioni assembly sono in realtà delle macroistruzioni che devono essere espanse dall'assemblatore in un certo numero di istruzioni di microcodice.

In figura 4.3 è riportato il codice assembly prodotto dai compilatori TAO e TAOMILLE in corrispondenza di una somma di due variabili intere. Senza inoltrarci nella descrizione dettagliata della sintassi e del formato delle singole istruzioni si può notare che le istruzioni assembly di APE100 sono in realtà delle macroistruzioni che una volta espanse producono un codice simile a quello prodotto per APEmille. Questo consiste nel caricamento dei due dati dalla memoria nei registri, nel calcolo della somma tra i due registri e nell'immagazzinamento del registro contenente il risultato della somma in memoria. Nell'assembly di APE100 parte di questo codice è "nascosto" nel formato piuttosto articolato previsto per l'istruzione di somma, che accetta come operandi delle locazioni di memoria e demanda all'assemblatore la generazione delle istruzioni di caricamento dei dati dalla memoria nei registri.

Una precedente versione dell'assembly APEmille (*assembly_V0*), che seguiva la struttura di quello adottato da APE100, è stata abbandonata perché il processo di espansione delle macroistruzioni è fisso e non consente di effettuare molte ottimizzazioni a partire dal codice di alto livello, dal momento che

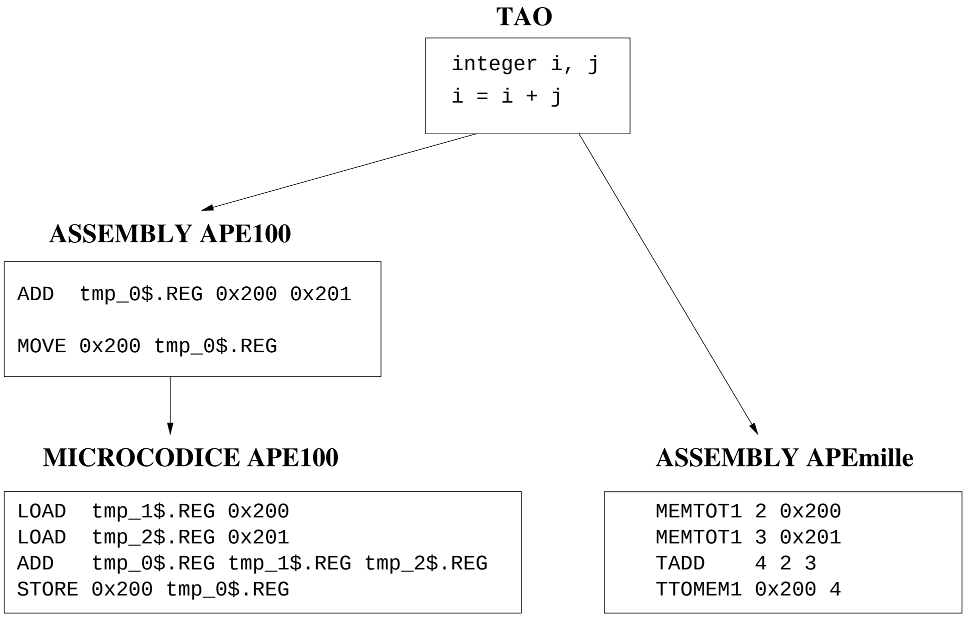


Figura 4.3: Confronto tra codice assembly di APE100 e di APEmille

diventa impossibile, per il compilatore, mappare le istruzioni ad alto livello nella sequenza di istruzioni di microcodice che le eseguano nella maniera più efficiente.

4.3.1 La struttura sintattica

Il linguaggio riconosciuto dall'assemblatore è composto dalle istruzioni assembly, dalle direttive per il macroespansore e da macroistruzioni predefinite usate essenzialmente per effettuare chiamate a routine di servizio del sistema operativo (*system service*). Una generica istruzione assembly è espressa per mezzo della sintassi:

MNEMONICO $[op_1\ \ [op_2\ \ [op_3\ \ [op_4\ ]\ ]\ ]$

Il campo corrispondente al codice mnemonico è fisso ed identifica il tipo di operazione che viene eseguita in corrispondenza dell'istruzione. Gli operandi, presenti in numero variabile da zero a quattro a seconda dell'istruzione, rientrano in uno dei tipi riportati in tabella 4.3.

Tipo dell'Operando	Descrizione
MOP	indirizzo globale di memoria (*Memory OPerand*)
reg	registro del controllore o del processore aritmetico
lab	etichetta (*label*)
const	numero intero in formato decimale o esadecimale
float	numero floating-point
str	stringa di caratteri delimitata da virgolette "
slice	sottoinsieme dei controllori o dei proc. aritm.

Tabella 4.3: Tipi degli Operandi per le Istruzioni Assembly

Il formato in cui viene espresso il codice mnemonico è quello di un stringa di caratteri codificati in maniera tale da avere in prima posizione la lettera T, J o C per indicare operazioni che riguardano rispettivamente il controllore, il processore aritmetico o l'unità di comunicazione. Le istruzioni del processore aritmetico che manipolano dati interi, in singola precisione, in doppia precisione, complessi o coppie di dati in singola precisione hanno inserita nel codice mnemonico rispettivamente la lettera I, S, D, C, V. Le istruzioni del controllore e del processore aritmetico che coinvolgono trasferimenti tra memoria dati e register file contengono la sottostringa MEM.

Il formato degli operandi dipende invece dal loro tipo. Il MOP rappresenta un indirizzo di memoria globale ed è espresso per mezzo del seguente formato:

$indirizzo_{base}$ [$.R_a$ [$.R_b$]]

$indirizzo_{base}$: const

R_a, R_b: reg

dove $indirizzo_{base}$ è un numero intero, cioè un *const*, compreso nello spazio degli indirizzi di memoria; R_a e R_b sono due interi che indicano registri del controllore. Il calcolo di un indirizzo di memoria a partire dal MOP avviene in questo modo:

$$indirizzo = indirizzo_{base} + @R_a + @R_b \tag{4.2}$$

ovvero l'indirizzo è dato dalla somma dello spiazzamento espresso da $indirizzo_{base}$ con il contenuto dei registri del controllore R_a e R_b. A seconda del numero di registri che vengono specificati nell'espressione del MOP si distinguono tre diversi tipi di operando:

- MOP1: $indirizzo_{base}$
- MOP2: $indirizzo_{base}.R_a$
- MOP3: $indirizzo_{base}.R_a.R_b$

Il calcolo 4.2 per ottenere l'indirizzo a partire dal MOP viene effettuato considerando solo i termini specificati esplicitamente. Nelle istruzioni di inizializzazione delle memorie il solo formato consentito è il MOP1.

L'operando *reg* indica, per mezzo di un numero intero, uno dei registri del controllore o del processore aritmetico. A livello di codice assembly, i registri che vengono specificati non corrispondono a quelli *fisici* dei register file, ma sono registri *virtuali* che verranno mappati dal modulo allocatore di registri della parte bassa della catena di compilazione nei registri fisici disponibili. Il numero intero che indica un registro virtuale non sarà quindi limitato superiormente dalla dimensione fisica dei register file. L'unico vincolo è dato dal fatto che determinati registri virtuali vengono mappati in modo rigido sui corrispondenti registri fisici che contengono le rappresentazioni dei numeri 0 ed 1 per i vari tipi di dato, vedi tabella 4.4.

Processore Aritmetico	
# registro	dato contenuto
0	zero per tutti i tipi di dato
2	unità immaginaria (0.0, 1.0)
4	uno singola precisione, uno reale per complessi (1.0, 0.0)
6	uno doppia precisione
8	uno vector (1.0, 1.0)
10	uno intero
Controllore	
0	zero intero
1	uno intero

Tabella 4.4: Registri riservati

Un operando *slice* ha due possibili formati:

1. $(x_1, y_1, z_1; x_2, y_2, z_2)$

2. (x, y, z)

l'effetto del primo formato è di selezionare un sottoinsieme dei nodi di calcolo o dei controllori aventi coordinate (x, y, z) tali che:

$x_1 \leq x \leq x_2$; $y_1 \leq y \leq y_2$; $z_1 \leq z \leq z_2$

quello del secondo formato è di selezionare un solo nodo o un solo controllore di coordinate (x, y, z). Le coordinate degli elementi selezionati devono essere coerenti con la configurazione della macchina sulla quale deve girare il programma (numero e topologia dei processori e dei controllori).

L'operando di tipo *const* può essere preceduto da un simbolo tra quelli indicati in tabella 4.5 per indicare il tipo di rilocabilità associata al numero intero.

Se l'operando *const* non è preceduto da nessun simbolo il *linker* assume che si tratti di un numero intero "puro" ovvero di un indirizzo assoluto. Una *label* è indicata da un numero intero preceduto da un asterisco " * ".

Simbolo	**Direttiva per il linker**
#	indirizzo assoluto
%	indirizzo rilocabile per la memoria dati del controllore
ˆ	indirizzo rilocabile per la memoria dati del proc. aritm.
*	indirizzo rilocabile per la memoria programma (label)

Tabella 4.5: Simboli per la Rilocabilità

All'interno di un programma assembly è possibile richiamare delle macroistruzioni predefinite una volta che si sia incluso il file che contiene la loro definizione. Solitamente il loro uso è associato a compiti quali l'allocazione della memoria dati e alla chiamata a *system service*. La sintassi prevista per le macroistruzioni predefinite è:

$\$nome[(\ op_1\ [\ ,op_2\ [\ ,op_3\ [\ ,op_4\ [\ ,op_5\ [,\ op_6]\ ]\ ]\ ]\ ]\ )]$

$nome$: $stringa$

op_n: $MOP1 \mid const \mid str$

il *nome* identifica la macro, la quale può avere da zero a sei operandi di tipo MOP1, *const* o *str*.

Nell'appendice A è riportato un elenco di tutte le istruzioni assembly, delle direttive per il macroespansore e delle macroistruzioni predefinite.

4.3.2 Tipi di istruzioni

Le istruzioni assembly di APEmille possono essere classificate in base alle operazioni che svolgono in tre tipi fondamentali:

1. istruzioni per l'inizializzazione dei dati
2. istruzioni eseguibili
3. metaistruzioni

Le istruzioni del primo tipo realizzano il caricamento dei dati nelle memorie dati dei controllori o dei processori aritmetici. Questo genere di istruzioni prevede un operando di tipo *slice* che indica quale sottoinsieme dei controllori o dei processori aritmetici è coinvolto nell'operazione di inizializzazione dei dati. Non sono previste istruzioni per il caricamento diretto di un dato nel register file dei processori aritmetici o dei controllori.

Nel secondo tipo rientrano le istruzioni aritmetiche, logiche, per il trasferimento dei dati, per il controllo del flusso del programma, per la conversione tra tipi di dato. A livello di istruzioni eseguibili il parallelismo della macchina non è visibile dal momento che le istruzioni di trasferimento tra nodi di calcolo sono mappate in memoria ed il partizionamento della macchina (cfr. 3.2) è effettuato per mezzo di chiamate a sistema operativo. Le istruzioni eseguibili fanno dunque riferimento ad un solo controllore e ad un solo processore aritmetico.

Le metaistruzioni, infine, non portano direttamente alla generazione di microcodice, ma costituiscono delle direttive per i vari moduli della catena di compilazione. Ad esempio all'interno di un programma assembly può comparire la seguente metaistruzione:

```
PRAGMA_JN_REGSET          [ 20  22  23 ]
```

il cui significato è di istruire la parte bassa della catena di compilazione che i tre registri virtuali (20, 22, 23) del processore aritmetico costituiscono un'insieme identificato con il simbolo REG_SET. Questo insieme di registri potrà essere usato come operando nelle istruzioni di trasferimento tra register file e memoria dati del processore aritmetico, ad esempio:

```
JTOMEM1         0x127   REG_SET
```

indicando con questa istruzione un trasferimento in blocco dei tre dati contenuti nei registri (20, 21, 22) nelle locazioni (0x127, 0x128, 0x129) della memoria dati.

4.4 La struttura del compilatore

Il compilatore è stato completamente riprogettato. Questa scelta è stata adottata successivamente alla realizzazione di una prima versione ottenuta modificando il compilatore TAO per APE100. Questa versione preliminare è in grado di compilare programmi sorgenti che usano tutti i costrutti sintattici previsti da TAOmille, generando codice *assembly_V0* (cfr. § 4.3). La necessità di ripensare l'intera struttura del compilatore è stata dettata principalmente da tre considerazioni:

- L'esigenza di integrare nel compilatore un modulo di ottimizzazione.
- Il cambiamento radicale nel linguaggio di destinazione del compilatore nel passaggio da *assembly_V0*, di tipo CISC, alla versione definitiva dell'assembly, di tipo RISC.
- La possibilità di riutilizzare una parte significativa del codice del compilatore TAOmille per il compilatore C++ che verrà sviluppato successivamente.

Il compilatore TAOmille opera con una sola passata, ovvero legge da un file il programma sorgente e, senza produrre file intermedi, scrive su un file il codice assembly relativo a quel programma. Il processo di compilazione è articolato in cinque fasi: analisi lessicale, sintattica e semantica del programma sorgente, produzione della rappresentazione intermedia del codice di alto livello, generazione del codice assembly. Le varie fasi sono realizzate per mezzo dei tre moduli che costituiscono la struttura di base del compilatore: il front-end, PICO e il generatore di codice assembly (back-end). Allo stato attuale il compilatore è in grado di produrre codice assembly a partire da singoli blocchi di codice TAOmille. Per blocco si intende una frazione di codice in cui il flusso di programma è sequenziale (cfr. § 2.3).

Il front-end è il modulo, scritto in Z_Z , che realizza le prime tre fasi della compilazione, ovvero le tre analisi lessicale, sintattica e semantica. Questo modulo riceve in ingresso il flusso contenente il programma TAOmille e, analizzando un'istruzione alla volta, effettua le chiamate alle funzioni di PICO che generano la rappresentazione intermedia relativa all'istruzione stessa. Il front-end è in grado di riconoscere errori di sintassi ed errori di semantica nel corso dell'analisi del programma e, nel caso, di generare appositi messaggi riportati sullo standard output e su file.

Il modulo PICO è scritto in C++ e fornisce al front-end tutte le funzioni e le classi necessarie alla produzione e alla manipolazione della tabella dei simboli e della rappresentazione intermedia del codice. Questa consiste in un grafo aciclico diretto (DAG) allocato, come la tabella dei simboli, nella memoria dinamica.

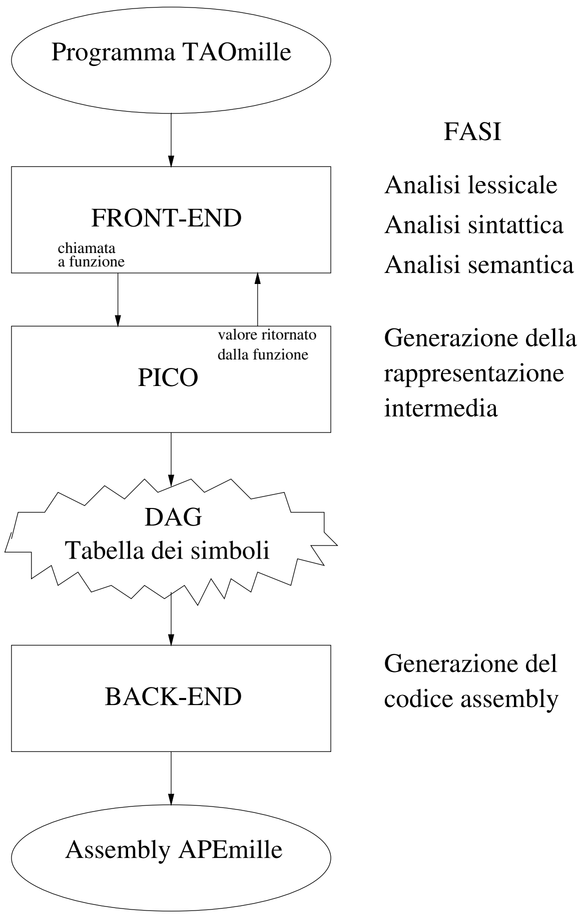


Figura 4.4: Struttura del compilatore TAOmille

La fase finale consiste nella produzione del codice assembly a partire dalla rappresentazione del programma come DAG e dalle informazioni contenute nella tabella dei simboli. Il modulo generatore di assembly, scritto in C++, effettua una serie di controlli sul DAG e sulla tabella dei simboli in modo da produrre codice assembly ottimizzato.

La struttura modulare del compilatore consentirà una agevole migrazione dello stesso verso altri linguaggi. In questo caso la gran parte delle modifiche andrà effettuata sul modulo front-end, dove viene definita la sintassi del linguaggio sorgente. Allo stesso modo si potrà agire sul modulo back-end per adattare il compilatore ad eventuali nuove versioni della macchina che prevedano variazioni nell'insieme delle istruzioni assembly.

4.5 Il front-end

Questo modulo effettua le analisi lessicale, sintattica e semantica del programma sorgente di alto livello. A questo scopo viene usato il parser dinamico Z_Z (cfr. § 4.1.2) che, oltre che come programma, è disponibile come libreria di funzioni C (zz.a). Includendo tale libreria in un file C++ diventa possibile specificare, come viene descritto in seguito, quali sono i file *header* di cui Z_Z deve effettuare il parsing per configurarsi a riconoscere la sintassi di TAOmille. Per mezzo di questa libreria è stato inoltre possibile specificare un insieme di regole grammaticali, detto *compiler kernel*, e richiamare, nelle azioni corrispondenti a queste regole, le funzioni C++ necessarie al funzionamento del compilatore. Nel nostro caso la chiamata a queste funzioni, definite nel modulo PICO, costituisce l'interfaccia tra il front-end ed il modulo che genera la rappresentazione intermedia del codice.

Nella figura 4.5 viene descritta la struttura del modulo front-end: al kernel che definisce i comandi di base riconosciuti da Z_Z si aggiunge, per mezzo

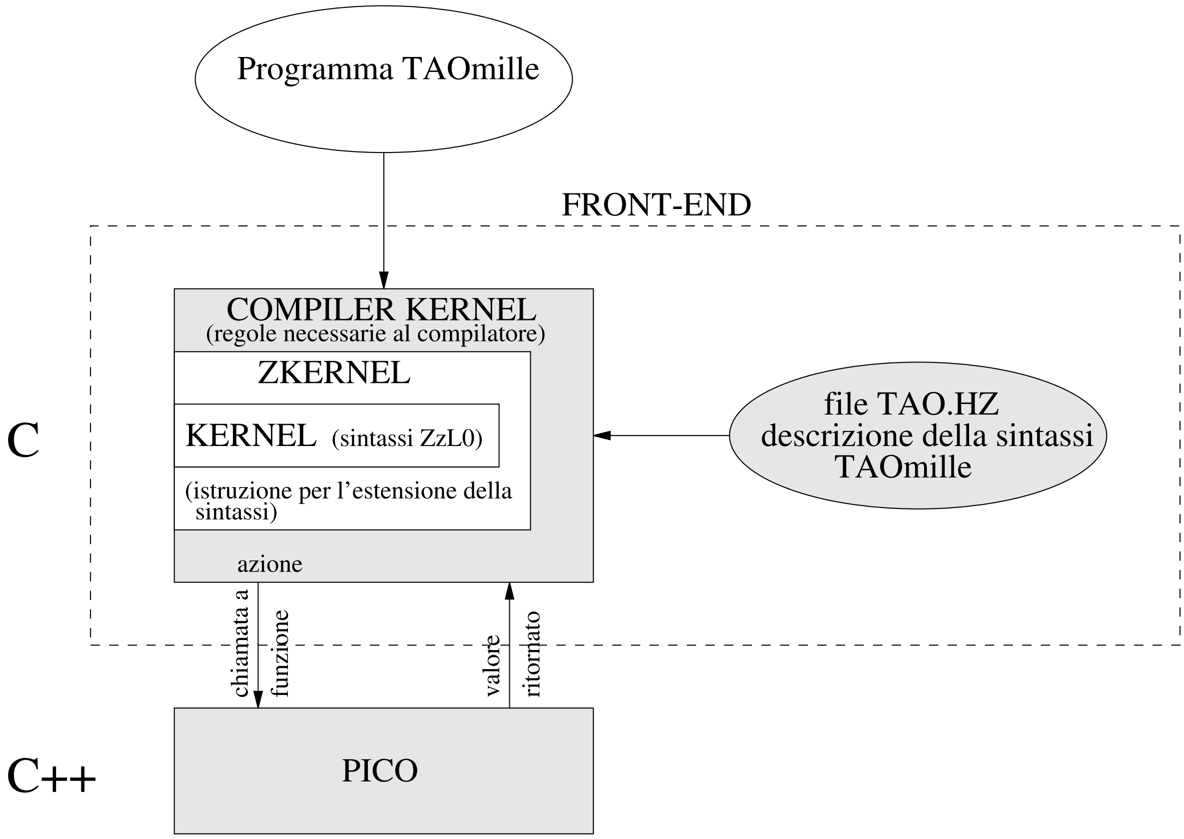


Figura 4.5: Schema del modulo Front-End

dello zkernel, l'insieme dei comandi Z_Z del compilatore, la cui azione consiste nel richiamare le funzioni C++ di PICO che generano la tabella dei simboli ed il DAG relativi al blocco di programma TAOmille che deve essere compilato. Il parsing del file tao.hz configura Z_Z a riconoscere la sintassi del linguaggio TAOmille ed a intraprendere le azioni necessarie alla compilazione in corrispondenza dei costrutti sintattici riconosciuti. Queste azioni prevedono, in ogni caso, l'esecuzione di comandi definiti nel compiler kernel, e quindi la chiamata alle funzioni C++ di PICO.

Di seguito è mostrata la sequenza di funzioni della libreria zz.a che vengono richiamate nel *main* del compilatore per utilizzare il parser Z_Z come *motore* per il front-end:

```
...
kernel();
zkernel();
compiler_kernel();
zz_parse_file("tao.hz");
...
```

La funzione KERNEL() carica la sintassi di base $Z_Z L_0$. ZKERNEL() definisce la meta-sintassi di Z_Z , ovvero il formato dell'istruzione per l'estensione della sintassi. Per mezzo della funzione ZZ_PARSE_FILE() viene specificato il file .hz che contiene le regole di produzione che descrivono la sintassi di TAOmille. In questo file la sintassi è descritta attraverso regole di produzione espresse nel formato, esposto nel § 4.1.2, che qui riportiamo per comodità:

```
/sintagma -> frase { azione }
```

COMPILER_KERNEL() è la funzione, specifica del compilatore TAOmille, che descrive la sintassi dei comandi Z_Z la cui azione consiste nel richiamare le funzioni di PICO. Per esprimere la sintassi dei comandi del compiler kernel

nel formato di regole di produzione scritte all'interno di un file C o C++ il compilatore ricorre alle seguenti funzioni della libreria zz.a:

- zOpen(*sintagma*): specifica la parte sinistra della regola di produzione.
- zKeyword($simbolo_{terminale}$): indica che nella parte destra della regola deve essere presente, nella posizione appropriata, un determinato simbolo terminale.
- zMatch($simbolo_{non\,terminale}$): ha la stessa funzione di zKeyword, ma indica la presenza di un simbolo non terminale all'interno della frase.
- zCall($nome_{funzione}$): l'azione della regola che si sta definendo consiste nella chiamata alla funzione C o C++.
- zCallfun($nome_{funzione}$, *tipo*): specificando il parametro *tipo*, al valore ritornato dalla funzione C o C++ viene associato un tipo tra quelli previsti da Z_Z .
- zClose(): indica la fine della regola.

L'uso di queste funzioni è però poco espressivo, sono state quindi definite le seguenti macro che hanno reso più immediata la scrittura delle regole:

```
#define O(tag)      zOpen(#tag);
#define M(a)        zMatch(#a);
#define K(a)        zKeyword(#a);
```

```
#define F(fun,tag) zCallFun((int(*)())fun,#tag); zClose();
#define P(proc)    zCall((void(*)())proc);       zClose();
```

Ad esempio consideriamo la regola che descrive la sintassi di TAO per l'assegnazione a variabile:

```
/stat->var^v "=" expr^e   { move nod(e) rap(v) }
```

La sequenza che il parser riconosce come un'assegnazione a variabile è data da una variabile seguita da un simbolo di uguale e da una qualsiasi espressione del tipo della variabile. L'azione viene tradotta dal compilatore nella esecuzione del comando Z_Z " move" definito per mezzo delle macro nella seguente maniera:

```
O(stat) K(move) M(nod) M(rap) P(node::move)
```

In questo modo quando Z_Z effettua il parsing di una assegnazione, viene eseguito il comando move del compiler kernel che richiama la funzione move della classe node definita in PICO, passandole i parametri appropriati.

Nel prossimo paragrafo viene ripresa la descrizione di alcune parti di codice di questo modulo allo scopo di mostrare come avvenga il processo di generazione della rappresentazione intermedia a partire dal parsing del programma sorgente di alto livello.

4.6 Pico

Nella sezione 2.2.3 si è parlato della generazione e della elaborazione di codice intermedio tra il linguaggio sorgente e quello di destinazione da parte di un compilatore. Ricordiamo qui alcune conseguenze che questa scelta comporta:

1. Si possono effettuare delle ottimizzazioni *machine independent* sulla rappresentazione intemedia in modo che il compilatore possa produrre codice di "buona qualità".

2. La maggiore portabilità di un tale compilatore, sia verso un altro linguaggio sorgente che verso un altro tipo di calcolatore (o comunque verso nuove versioni dello stesso).

Pico è il modulo del compilatore TAOmille dove vengono definite tutte le classi C++ necessarie alla generazione ed alla manipolazione del codice intermedio relativo a singoli blocchi di programma di alto livello, dove per blocco di programma si intende una parte di esso in cui il flusso sia sequenziale (cfr. 2.3). Prima di descrivere in dettaglio il tipo di rappresentazione intermedia adottata nel compilatore TAOmille, forniamo una serie di definizioni utili nel seguito della sezione.

4.6.1 Alberi e grafi

Un *albero radicato*, d'ora in poi semplicemente *albero*, è un insieme T non vuoto di nodi in cui esiste un nodo speciale r detto *radice* dell'albero. Gli altri nodi sono ripartiti in una successione ordinata di sottoinsiemi $T_1 ... T_k$ ciascuno dei quali è un *albero*. I sottoinsiemi $T_1 ... T_k$ vengono detti *sottoalberi* della radice.

Si definiscono *foglie* (o nodi terminali) di un albero quei nodi che non hanno sottoalberi. I nodi che non sono foglie vengono detti *nodi interni* o non terminali. Dato un albero o un sottoalbero T di radice r, tutti gli altri nodi che gli appartengono vengono detti *discendenti* di r. Ci si può riferire alla radice r di un albero o di un sottoalbero come al nodo *padre* delle radici dei suoi sottoalberi, tali radici sono dette nodi *figli* di r.

Un *cammino* da un nodo S ad un nodo D è una successione di nodi:

$$n_0, n_1, ..., n_i, n_{i+1}, ..., n_k$$

tale che:

1. $n_0 = S$ e $n_k = D$
2. n_i è padre di $n_{i+1} \quad \forall\, 0 \leq i \leq k$

Dato un qualsiasi albero, se esiste un cammino da un nodo n_i ad un nodo n_j questo è unico.

Una *foresta* F è una successione ordinata $T_1...T_n$ di alberi.

Con riferimento alla figura 4.6 possiamo individuare la radice nel nodo A, le foglie nei nodi {B, F, H, I, L, E}, F e G nei figli di C (ovvero C nel padre di F e G), i nodi {H, I, L, G} nel sottoalbero radicato in G.

Un tipo di albero di particolarmente interessante ai nostri scopi è l'albero binario, del quale diamo la seguente definizione ricorsiva:
un *albero binario* può essere l'albero binario vuoto oppure una coppia ordinata di alberi binari (detti sottoalbero *sinistro* e sottoalbero *destro*) cui è associato un nodo detto *radice* dell'albero binario. Un albero binario è usato come

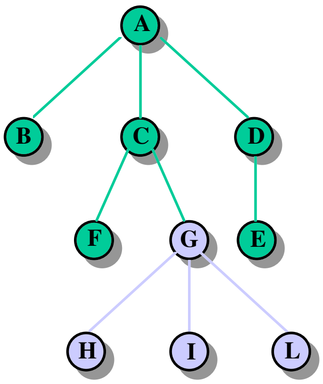


Figura 4.6: Esempio di albero

struttura dati, a ciascun nodo sarà associato un certo insieme di dati, è quindi naturale dover definire un metodo per accedere a questi dati (visita dell'albero binario). Ciò equivale a risolvere il seguente problema: dato un albero binario visitare una sola volta tutti i suoi nodi. Esistono tre algoritmi di visita di un albero binario, che vengono implementati in maniera ricorsiva. La visita dell'albero binario comincia a partire dalla sua radice:

1. *Ordine anticipato (preorder o depth-first)*: si visita la radice, poi i sottoalberi sinistro e destro in ordine anticipato.

2. *Ordine binario simmetrico (inorder)*: si visita in ordine binario simmetrico il sottoalbero sinistro, poi si visita la radice e quindi si visita il

sottoalbero destro in ordine binario simmetrico.

3. *Ordine differito (postorder o bottom-up)*: si visita in ordine differito il sottoalbero sinistro e poi quello destro; infine si visita la radice. Si tratta quindi di un ordine di visita che parte dalle foglie e risale verso la radice dell'albero (da cui la denominazione *bottom-up*).

Un *grafo* G consiste di due insiemi: $G = (V, A)$. V è un insieme non vuoto di *vertici* (o *nodi*) solitamente numerati $1, 2, ..., n$ ed A è un insieme finito di *archi*, ovvero di coppie di nodi. Se le coppie sono ordinate, cioè se $(u, v) \neq (v, u)$ allora il grafo si dice *orientato* o *diretto*, altrimenti il grafo si dice *non orientato*.

Ristringendo l'attenzione ai grafi orientati, definiamo *grado entrante* e *grado uscente* di un nodo rispettivamente il numero di archi che incidono e che si dipartono da quel nodo. Un *cammino* da v_S a v_D in un grafo G è una successione di nodi $v_0...v_n$ tali che $v_0 = v_S$, $v_n = v_D$ e inoltre, per ogni coppia v_i, v_{i+1} della successione esiste in G l'arco (v_i, v_{i+1}). Se $v_S = v_D$ il cammino viene detto *ciclo*, un grafo che non preveda cicli viene detto *aciclico*.

Un DAG (*directed acyclic graph*) è un grafo orientato aciclico. I DAG sono piuttosto diversi dai comuni grafi orientati in quanto, in un certo senso, sono in parte alberi ed in parte grafi, tanto è vero che un albero può essere pensato come un particolare DAG dove la radice è un nodo particolare con grado entrante nullo, ogni altro nodo ha grado entrante 1 ed esiste un cammino (unico) dalla radice ad ogni nodo.

È possibile avvalersi della struttura particolare dei DAG al momento della loro elaborazione. Un'operazione fondamentale definita su questo tipo speciale di grafi consiste nell'elaborare i nodi del grafo in un ordine tale per cui nessun nodo viene elaborato prima dei suoi "figli".

4.6.2 Il formato del codice intermedio

Il codice intermedio rappresenta sia le istruzioni che la tabella dei simboli (variabili e costanti) di un blocco di programma TAOmille.

La rappresentazione delle istruzioni avviene sotto forma di grafi orientati aciclici (DAG), quella delle variabili e delle costanti per mezzo di una tabella di oggetti appartenenti ad una gerarchia di classi appositamente progettata che verrà descritta nel seguito. In particolare viene utilizzato un sottoinsieme di tutti i possibili DAG, quello in cui il grado uscente di ogni nodo vale al massimo due.

I DAG sono strutture dati molto utili da manipolare per effettuare delle ottimizzazioni, dal momento che forniscono una immagine di come i valori calcolati da ogni istruzione vengono utilizzati nelle istruzioni successive.

La costruzione dei DAG a partire dalle istruzioni di alto livello è un metodo efficace per determinare le sottoespressioni comuni (cioè le espressioni calcolate più di una volta all'interno di un blocco di programma), per capire quali simboli (variabili e costanti) sono usati all'interno del blocco ma calcolati fuori da esso e per individuare quali valori calcolati all'interno del blocco potrebbero essere utilizzati fuori di esso.

Quando si parlerà degli algoritmi utilizzati su queste strutture saranno fornite delle rappresentazioni grafiche dei DAG del tipo di quella mostrata in figura 4.7, diamo quindi una serie di indicazioni sulle convenzioni che saranno adottate.

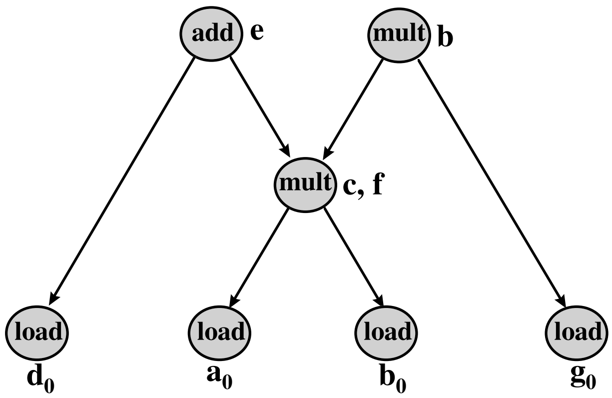


Figura 4.7: DAG per un blocco di programma

Ai nodi del DAG sono associati degli oggetti detti *etichette* che sono informazioni relative all'istruzione rappresentata:

1. I nodi sono etichettati con un operatore tra quelli previsti da Pico.

2. I nodi foglia sono etichettati con un unico identificatore che può rapp-

resentare sia il nome di una variabile sia di una costante. I nodi foglia sono sempre associati agli *r-value*, ovvero a variabili e costanti che in una istruzione compaiono alla destra di un operatore di assegnazione. In particolare le etichette associate ai nodi foglia rappresentano i valori iniziali delle variabili alla entrata del blocco di programma e saranno indicate con pedice zero.

3. I nodi possono essere etichettati con una lista di identificatori che rappresentano gli *l-value*, cioè gli identificatori delle variabili che compaiono a sinistra di un operatore di assegnazione.

In figura 4.7 è riportato il DAG relativo al seguente breve blocco di programma TAOmille:

```
c = a * b
e = d + c
f = b * a
b = f * g
```

Discuteremo il significato del DAG dopo aver fornito l'algoritmo per la sua costruzione a partire dal blocco di programma, per il momento si può comunque osservare che ogni nodo del DAG rappresenta un'operazione di calcolo in termini delle foglie, cioè dei valori posseduti dalle variabili (o dalle costanti) all'entrata del blocco.

Si nota inoltre che gli operatori con cui sono etichettati i nodi interni non

Tipo	Nome dell'operatore
Aritmetico	ADD, SUB, MUL, DIV, INC, DEC, MOD, UMINUS, MULA, INV, INVSQRT.
Logico	EQ, NE, GT, LT, GE, LE, ANDL, ORL.
Memoria/Registro	LOAD, STORE.
Speciale	FMLY, BLDCMPL, BLDVCTR, COPY, NPRT, NOP, FRST, SCND.
Conversione	STOC, STOD, STOI, DTOS, DTOC, DTOI, ITOS, ITOC, ITOD.
Bitwise	NOT, AND, NAND, OR, NOR, XOR, LSH, ASH, ROT, LEADONE, BITCOUNT.

Tabella 4.6: Operatori di Pico

corrispondono a quelli di TAOmille. Pico prevede infatti un suo insieme di operatori sui quali vengono mappati quelli del linguaggio di alto livello; questo è uno degli accorgimenti adottati per rendere la rappresentazione intermedia il più possibile indipendente dal linguaggio sorgente. In tabella 4.6 sono riportati gli operatori di Pico raggruppati in base alla loro funzione.

4.6.3 Algoritmo per la costruzione del DAG

In questa sezione descriviamo l'algoritmo per la costruzione del DAG considerando, per maggiore chiarezza, solo i tipi di dato non strutturati di TAOmille (integer, localint, real, double). Nel seguito verranno esposte le variazioni all'algoritmo generale che derivano dal trattamento dei tipi strutturati del linguaggio (complex, vector, array, record).

Negli esempi seguenti si mostrerà la costruzione dei DAG per le espressioni aritmetiche. Tralasceremo gli aspetti relativi alla rappresentazione degli oggetti contenuti nella tabella dei simboli e ci concentreremo su quelli che riguardano la costruzione del DAG.

Si possono considerare le seguenti definizioni per la node_of(), che sono concettualmente equivalenti a quelle scritte nel codice C++ :

1. *nodo node_of(rap r)* : alla funzione viene passato un oggetto *rap* che rappresenta una variabile (o una costante); essa restituisce, se esiste, il nodo del DAG che ha come etichetta quella variabile (o costante), altrimenti crea un nuovo nodo foglia e lo etichetta con la variabile (o la costante) e con l'operatore LOAD.

2. *nodo node_of(operatore op, nodo figlio_sx)* : la funzione riceve in ingresso un operatore *op* tra quelli previsti da Pico ed un nodo $figlio\_sx$. Essa ricerca nel DAG un nodo interno che sia etichettato con *op* e che abbia come figlio sinistro il nodo $figlio\_sx$ e lo restituisce. Se non esiste tale nodo nel DAG lo crea e lo restituisce.

3. *nodo node_of(operatore op, nodo figlio_sx, nodo figlio_dx)* : ricerca nel DAG un nodo con operatore *op* e con figli sinistro e destro rispettivamente uguali ai nodi $figlio\_sx$ e $figlio\_dx$ se *op* non è commutativo, oppure ignorando l'ordinamento se l'operatore è commutativo e lo restituisce. Se non esiste tale nodo, lo crea e lo restituisce.

La quarta regola tra quelle estratte dal *compiler kernel*, descrive l'assegnazione di una espressione ad una variabile e richiama, passandole i parametri

appropriati, la funzione move() di Pico che possiamo qui definire nella seguente maniera:
void move(rap r, nodo n) : etichetta il nodo n con la variabile r; se r era già associata ad un altro nodo interno, la rimuove come etichetta di quel nodo prima di etichettare n.

Durante il parsing di una istruzione, ad esempio $c = a + b$, si cercano nel DAG, per mezzo di *node_of()*, i nodi che rappresentano il valore “corrente” di a e di b. Questi nodi potrebbero essere foglie, oppure nodi interni nel caso in cui uno o entrambi i simboli a e b siano stati calcolati da espressioni precedenti nello stesso blocco di programma. Se la ricerca non ha successo si creano i nodi foglia, ciascuno etichettato con il simbolo non trovato. Se già non esiste nel DAG, si crea un nodo interno etichettato con l'operatore appropriato, in questo caso ADD, e gli si danno come figli sinistro e destro i nodi di a e di b rispettivamente; infine si etichetta il nuovo nodo con c. Se invece un nodo simile già esisteva ci si limita ad attaccare c alla sua lista di identificatori. Infine bisogna ricordare che se c (e non c_0) etichettava già un altro nodo, ovvero in una istruzione precedente del blocco compariva a sinistra di una assegnazione, questa etichetta andava rimossa prima di attaccare c al nuovo nodo.

Per illustrare l'algoritmo utilizzato per la costruzione del DAG, è utile considerare una parte di codice, piccola ma significativa, estratta dal file di configurazione tao.hz del front-end del compilatore (cfr. § 4.5) e mostrata in figura 4.8.

Queste regole di produzione della grammatica di TAOmille descrivono la sintassi delle espressioni aritmetiche e richiamano le opportune funzioni delle classi di Pico necessarie a costruire il DAG. Una volta che Z_Z sia configurato con queste regole esso è in grado di riconoscere tutte le istruzioni TAOmille del tipo:

$nome_{variabile} = espressione_{aritmetica}$

Nelle regole sono presenti i simboli non terminali $< expr >$, $< term >$ e $< fact >$. Consideriamo, per semplicità, *var* e *const* come simboli terminali. Attraverso la catena di derivazioni:

$< expr > \Rightarrow < term >$

$< term > \Rightarrow < fact >$

$< fact > \Rightarrow var$

$< fact > \Rightarrow const$

la grammatica garantisce il corretto ordine di produzione dei nodi del DAG, rispettando le precedenze degli operatori aritmetici. Effettuando una serie di derivazioni canoniche a destra sulle regole (applicando cioè le produzioni sempre a partire dal simbolo non terminale che sta più a destra nella frase) Z_Z analizza l'espressione aritmetica ottenendo di volta in volta la tripla costituita dagli identificatori delle variabili (o delle costanti) e dal simbolo (+, -, *, /) di una singola operazione aritmetica. A partire da questa tripla, svolgendo le operazioni riportate nella azione della corrispondente regola di produzione,

```
/expr->term^$ :pass             !! REGOLA 1
/term->fact^$ :pass             !! REGOLA 2
/fact->"(" expr^$ ")" :pass
/fact->typenod^$ :pass
/fact->var^v {                  !! REGOLA 3
  /nn=node_of  current_block rap(v)
  /return nn as typenod
}
/fact->const^c {
  /nn=node_of  current_block c
  /return nn as typenod
}

/expr->expr^e "+" term^t {
  /nn=node_of current_block ADD nod(e) nod(t)
  /return nn as typenod
}
/expr->expr^e "-" term^t {
  /nn=node_of current_block SUB nod(e) nod(t)
  /return nn as typenod
}
/term->term^t "*" fact^f {      !! REGOLA 4
  /nn=node_of  current_block MUL nod(t) nod(f)
  /return nn as typenod
}
/term->term^t "/" fact^f {
  /nn=node_of  current_block DIV nod(t) nod(f)
  /return nn as typenod
}

/stat->var^v "=" expr^e {       !! REGOLA 5
  move nod(e) rap(v)
}
```

Figura 4.8: Regole della grammatica di TAOmille per le operazioni aritmetiche

Z_Z richiama le funzioni di Pico che generano i nodi (con le relative etichette) e gli archi che rappresentano quella operazione aritmetica all'interno del DAG.

Il front-end prevede quattro regole Z_Z che comportano la chiamata alle funzioni di Pico che si occupano di generare il DAG. Queste sono definite nel *compiler kernel* per mezzo delle seguenti istruzioni, espresse nel formato illustrato nella sezione 4.5:

```
O(nod)  K(node_of) M(blo) M(rap) F(node_of_ptr1,nod)
O(nod)  K(node_of) M(blo) M(int) M(nod) F(node_of_ptr2,nod)
O(nod)  K(node_of) M(blo) M(int) M(nod) M(nod) F(node_of_ptr3,nod)
O(stat) K(move)    M(nod) M(rap) P(node::move)
```

Nelle prime tre istruzioni vengono definite le regole che permettono di richiamare da Z_Z la funzione node_of() di Pico rispettivamente con uno, due o tre parametri espliciti; il parametro *blo* rappresenta il blocco di programma del quale si vuole costruire il DAG ed in questo contesto (singolo blocco) può essere ignorato. Nell'implementazione in C++ la definizione di node_of() è *sovraccaricata*, è cioè ripetuta più volte con un diverso numero di parametri. Per questioni legate all'interfacciamento tra Z_Z e C++ è stato necessario esplicitare nel *compiler kernel* il sovraccarico definendo tre funzioni node_of_ptr1(), node_of_ptr2(), node_of_ptr3() diverse, ognuna per ogni definizione di node_of().

Dopo questo esempio preliminare trattiamo il caso più generale, tralasciando per il momento gli aspetti relativi alla rappresentazione degli oggetti contenuti nella tabella dei simboli e concentrandoci su quelli che riguardano la costruzione del DAG.

A questo punto si può facilmente comprendere come avvenga tutto il processo di costruzione del DAG a partire dal parsing delle espressioni. La scomposizione operata dal front-end sulla espressione attraverso le derivazioni espresse dalle regole grammaticali porta a una delle seguenti tre situazioni:

1. Riduzione ad una variabile (o a una costante) var (o $const$). In questo caso le regole del *compiler kernel* prevedono la chiamata alla node_of(rap).

2. Riduzione ad una frase del tipo $op\ var$. Viene richiamata la node_of(op, node_of(rap)).

3. Riduzione ad una frase del tipo $var1\ op\ var2$. Nell'azione viene richiamata la funzione node_of(op, node_of(rap1), node_of(rap2)).

Ad esempio illustriamo il processo relativo alla costruzione del DAG per una singola istruzione aritmetica:

$$c\ = a\ *\ b$$

partendo dalla riduzione della stessa da parte del front-end per mezzo delle regole grammaticali riportate in precedenza (vedi figura 4.8) e arrivando alle corrispondenti chiamate delle funzioni di Pico che generano il DAG. Le regole che intervengono nel parsing della istruzione in esame sono state numerate da 1 a 5 nel codice riportato in figura 4.8. Mostriamo come tali regole vengono utilizzate per ridurre l'istruzione, contrassegnando la regola che interviene in una certa riduzione con il numero ad essa assegnato:

5 $c \ = < expr >$

1 $c \ = < term >$

4 $c \ = < term > \ * \ < fact >$

3 $c \ = < term > \ * b$

2 $c \ = < fact > \ * b$

3 $c \ = \ a \ * \ b$

Il parser Z_Z esegue l'*azione* associata ad una regola solo dopo aver riconosciuto l'intera *frase*, questo comporta che la catena di derivazioni mostrata dà luogo ad una serie di azioni eseguite nell'ordine inverso che richiamano le funzioni di Pico attraverso i comandi del *compiler kernel*:

3 na = node_of(rap_a)

2

3 nb = node_of(rap_b)

4 nc = node_of(MULT, na, nb)

1

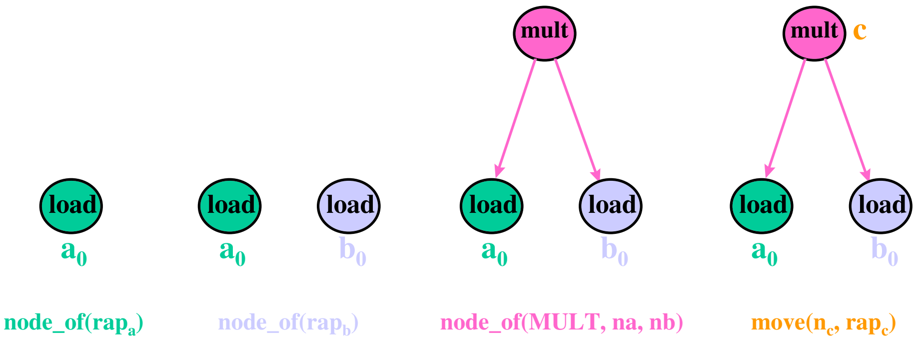


Figura 4.9: Costruzione del DAG per l'istruzione c = a * b

5 move(nc, rap_c)

Come mostrato in figura 4.9 la chiamata a node_of(rap_a) comporta la creazione, dopo una infruttuosa ricerca dello stesso (il DAG è vuoto!), del nodo foglia n_{a_0}, etichettato con a_0; allo stesso modo node_of(rap_b) crea un nodo foglia n_{b_0} e lo etichetta con b_0. Con la chiamata a node_of(MULT, n_{a_0}, n_{b_0}) viene creato un nodo n_c con operatore MULT a cui vengono assegnati come come figli sinistro e destro rispettivamente i nodi foglia associati ad a e a b. Infine l'assegnazione della espressione a c viene trattata richiamando la move(n_c, rap_c).

Possiamo ora considerare un esempio di costruzione del DAG per un piccolo blocco di programma composto da tre istruzioni aritmetiche (vedi figura 4.10).

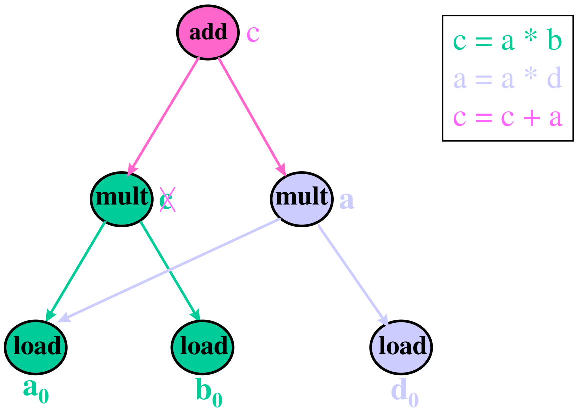


Figura 4.10: Costruzione del DAG per un blocco di tre istruzioni

Si noti che le istruzioni hanno tutte il formato $nome_{var} = nome_{var}\ op\ nome_{var}$. Questo non è assolutamente un vincolo per l'algoritmo implementato che è in grado di trattare espressioni comunque complesse, ma è utile ai fini della descrizione dell'algoritmo stesso. Con riferimento alla figura 4.10, alla prima istruzione corrisponde il "sottoalbero" colorato in verde, ed il procedimento con cui questo è stato generato è già stato spiegato in dettaglio (vedi figura 4.9).

L'elaborazione della seconda istruzione del blocco genera i nodi e gli archi del DAG colorati in violetto: la node_of(rap_a) ritorna il nodo foglia n_{a_0} etichettato con a_0 che è già presente nel DAG; la node_of(rap_d) crea il nodo foglia n_{d_0} associato a d dal momento che tale nodo non esisteva nel DAG; la node_of(MULT,

n_a, n_d) crea un nodo n_a assegnandogli n_{a_0} come figlio sinistro e n_{d_0} come figlio destro, infine la move(n_a, rap_a) etichetta il nodo n_a con l'*l-value* a, senza togliere l'etichetta a_0 dal nodo foglia su cui compare.
La terza istruzione aggiunge il nodo e gli archi di colore rosa: la node_of(rap_c) restituisce il nodo n_c che era già stato generato elaborando la prima istruzione, la node_of(rap_a) restituisce il nodo n_a etichettato con a che era stato generato elaborando la seconda istruzione, la node_of(ADD, n_c, n_a) crea un nuovo nodo n_C con figlio sinistro n_c e figlio destro n_a, infine la move(n_C, rap_c) rimuove l'etichetta c dal nodo n_c e la attacca al nodo n_C appena creato.

A questo punto si può notare come applicando l'algoritmo per la costruzione del DAG si possano ricavare una serie di informazioni sul blocco di programma. Come risulta dal DAG di figura 4.11 che rappresenta il seguente blocco di programma:

```
d = a * b + 1
e = a
g = f - b * e
```

vengono rilevate automaticamente le sottoespressioni comuni a più istruzioni, anche se queste sono abbastanza "nascoste", come in questo caso. Il nodo figlio sia del nodo associato a d che di quello associato a g rappresenta in questo caso la sottoespressione comune alla prima e alla terza istruzione del blocco, ovvero $a * b$.

Un secondo tipo di informazione che può essere ricavato dal DAG di un

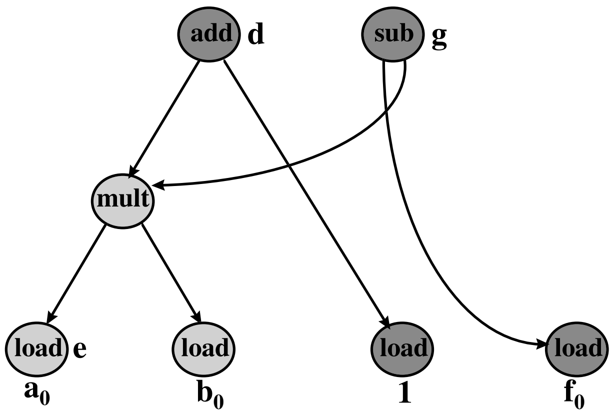


Figura 4.11: Rilevamento delle sottoespressioni comuni nel DAG

blocco di programma riguarda la determinazione di quali siano le variabili o le costanti il cui valore viene utilizzato all'interno del blocco: queste coincidono con quelle per cui sono state creati i nodi foglia. Ad esempio nel blocco corrispondente al DAG di figura 4.11 queste coincidono con le variabili a, b, f e con la costante 1. Infine, dall'analisi del DAG si può determinare quali valori calcolati dalle istruzioni del blocco potranno essere usati fuori da esso: si tratta dei valori conenuti nelle variabili che compaiono come etichetta *l-value* di un qualche nodo alla fine della costruzione del DAG. Ad esempio nel DAG di figura 4.10 la prima istruzione del blocco non soddisfa questa condizione, dal momento che l'identificatore che in essa compare come *l-value* (c), viene utilizzato nello stesso ruolo da una istruzione successiva (la terza): nel blocco rappresentato solo i valori calcolati dalla seconda e dalla terza istruzione potranno essere usati fuori da esso. Informazioni di questo tipo, ovvero quali sono le variabili o le costanti i cui valori sono usati in un blocco e quali sono le istruzioni di un blocco che calcolano valori che possono essere usati fuori da esso, sono molto utili per le cosiddette ottimizzazioni multi-blocco, di cui si parla nella sezione 2.3.

4.6.4 Algoritmo per la visita del DAG

Nella sezione precedente è stato illustrato l'algoritmo usato dal compilatore per generare la rappresentazione come DAG di un blocco di programma. Come si è visto, si ricorre spesso alla ricerca di "sottoalberi" aventi certe caratteristiche, individuando così le sottoespressioni comuni a più istruzioni. La logica generale di tale algoritmo prevede infatti sempre la ricerca dei dati all'interno del DAG prima di eventuali inserimenti di nuovi nodi ed archi.

Il DAG prodotto dal compilatore ha, come è già stato ricordato, una struttura più semplice di quella di un generico grafo orientato aciclico. Esso è infatti costituito da nodi aventi al massimo due archi uscenti, mentre non c'è limitazione al numero di archi entranti. Questo permette di affrontare il problema della ricerca nel DAG scomponendolo nella sua *foresta* di ricerca, come mostrato in figura 4.12.

Durante la fase di costruzione del DAG la funzione node_of() tiene traccia dei nodi del DAG che non hanno padri, denominati *teste* del DAG. A questo

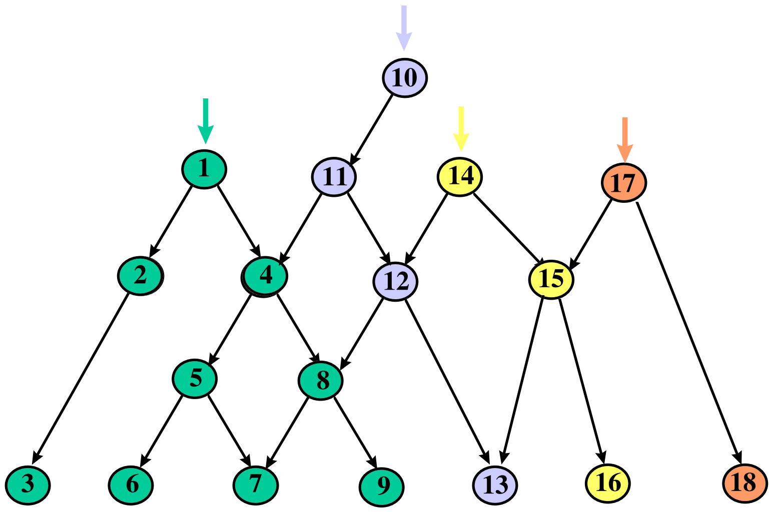


Figura 4.12: Foresta di visita del DAG

scopo la node_of() fa uso di una struttura dati a lista concatenata, implementata in C++ per mezzo di un *template*. Lo stesso tipo di struttura dati a lista concatenata viene utilizzato per etichettare i nodi con gli oggetti *rap* che rappresentano gli *l-value*, grazie al fatto che il *template* consente di parametrizzare la lista con il tipo dei suoi elementi. Nella figura i nodi *testa* sono indicati da frecce di colore uguale a quello dell'albero binario di cui essi sono *radice*.

L'algoritmo di ricerca prevede la visita in ordine anticipato (o depth-first) di ciascuno degli alberi della foresta di ricerca, nell'ordine indicato dalle *radici* contenute nella lista delle *teste* del DAG. Nel corso della visita ogni nodo viene marcato con un *flag* che lo identifica come "già visitato". In questo modo i sottoalberi comuni a più alberi binari vengono visitati solo una volta. In figura è mostrato l'ordine di visita dei nodi del DAG ottenuto applicando l'algoritmo appena descritto.

La funzione node_of() richiama la visita del DAG in uno dei due casi seguenti:

1. Ricerca di un nodo etichettato con operatore OP e con figlio sinistro N_{sx}.
2. Ricerca di un nodo etichettato con operatore OP, con figlio sinistro N_{sx} e con figlio destro N_{dx}.

La prima ricerca viene effettuata nel caso si voglia determinare se nel DAG è già rappresentata una espressione avente formato

$$OP_{unario}\ \ variabile$$

in caso affermativo la node_of(OP, N_{sx}) evita ricostruire il sottoalbero dato dal nodo che rappresenta l'operatore e dal suo nodo figlio sinistro etichettato con la variabile.

La seconda ricerca avviene per le stesse ragioni nel caso si abbia una espressione del tipo

$$variabile_{sx}\ \ OP_{binario}\ \ variabile_{dx}$$

ma viene richiesta dalla node_of(OP, N_{sx}, N_{dx}) prima della eventuale creazione del sottoalbero formato dal nodo etichettato con l'operatore e avente come figli sinistro e destro i nodi etichettati con $variabile_{sx}$ e con $variabile_{dx}$. Nel caso che l'operatore binario sia commutativo, la ricerca viene svolta senza tenere conto dell'ordinamento (sinistro, destro) dei nodi figli del nodo che rappresenta l'operatore.

Si può notare che nel caso in cui l'espressione sia formata da un solo identificatore, come nel caso di una assegnazione semplice

$$var1 = var2 \mid const$$

o come avviene alla fine della riduzione di una espressione arrivando ai singoli identificatori di variabili e costanti, la node_of(rap) non comporta la visita del DAG, dal momento che l'oggetto rap contiene l'informazione sul nodo che viene etichettato da esso (ovvero il puntatore al nodo in cui la variabile o la costante rappresentata da rap compaiono come etichetta) e quindi la ricerca comporta solo la lettura di questa informazione (basta leggere il valore del puntatore).

4.7 Il back-end

Nella sezione 4.6 si è visto come il compilatore costruisca la rappresentazione intermedia di un blocco di programma sotto forma di DAG.

In questa sezione vengono descritti gli algoritmi la cui realizzazione in C++ costituisce il back-end del compilatore, ovvero l'algoritmo "normalizzatore" e, soprattutto, l'algoritmo di generazione del codice assembly a partire dal DAG.

Come è già stato visto nella sezione 3.3.2 l'operazione fondamentale di calcolo floating-point dei processori aritmetici Jane è la *normal*, che corrisponde al calcolo della espressione:

$$a * b + c$$

Dovendo essere il più possibile indipendente dal linguaggio assembly di destinazione, il DAG generato dal compilatore non prevede la rappresentazione diretta di una tale operazione. L'algoritmo "normalizzatore" si occupa di ricombinare, quando sia possibile, una operazione di moltiplicazione con una operazione di somma, in modo da sfruttare appieno la potenza di calcolo del processore aritmetico. La descrizione di questo algoritmo viene svolta successivamente a quella per la generazione dell'assembly, contrariamente all'ordine con cui i due algoritmi prendono parte al processo di compilazione, dal momento che così facendo questa risulterà di più immediata comprensione.

4.7.1 Il generatore di codice assembly

Data una espressione E definiamo come generazione ottimale di codice la traduzione in linguaggio assembly di E che presenti il minimo numero di istruzioni. Si può dimostrare [25] che il problema della generazione ottimale di codice per una macchina con un numero infinito di registri a partire da un DAG che contenga sottoespressioni comuni è NP-completo.

Nel compilatore si è adottato un algoritmo di generazione di complessità $O(n)$ che, basandosi su tecniche euristiche, produce codice assembly avente le seguenti caratteristiche:

1. Le sottoespressioni comuni vengono calcolate una sola volta. Ad esempio il codice assembly ottenuto a partire dal blocco rappresentato dal DAG di

figura 4.11 effettua una sola volta il calcolo della sottoespressione comune alla prima e alla terza istruzione.

2. Tutte le variabili e le costanti presenti nel blocco vengono allocate sui registri virtuali. Si è adottata questa scelta partendo dalle ipotesi ottimistiche che, date le grandi dimensioni dei register file di Jane e Tarzan, l'allocatore dei registri fisici, contenuto nel modulo FLINT, si trovi ad affrontare la situazione di esaurimento dei registri fisici disponibili in una percentuale molto piccola dei casi possibili e che in tal caso la strategia di *spilling* sia efficiente.

3. Le variabili a cui venga assegnata una sottoespressione comune sono allocate sullo stesso registro (sullo stesso insieme di registri se le variabili sono di tipo complex, vector o record). In questo modo viene ridotta percentuale di occupazione del register file (*copy-propagation*).

4. La lettura dei valori delle variabili dalla memoria nei registri avviene nel momento in cui nel codice assembly questi si rendono necessari come operandi di altre istruzioni. La scrittura in memoria del contenuto dei registri avviene esclusivamente alla fine del codice assembly per il blocco di programma, e riguarda solo i registri contenenti valori calcolati all'interno del blocco che possano essere riutilizzati fuori da esso. Le istruzioni di lettura e scrittura in memoria delle variabili avvengono, dove possibile, per *burst*, ovvero leggendo o scrivendo blocchi di dati in locazioni

di memoria contigue. Questo è un punto di importanza fondamentale relativamente all'architettura di APEmille, e verrà ripreso in dettaglio nella sezione 4.8 quando si parlerà della gestione dei tipi strutturati da parte del compilatore.

5. Alcune istruzioni assembly non sono la traduzione diretta delle corrispondenti istruzioni di alto livello, ma sono generate effettuando una *riduzione in forza* di queste. La *riduzione in forza* di una istruzione corrisponde alla sua sostituzione con una o più istruzioni che comportino un minor tempo di esecuzione. Ad esempio, le costanti non vengono caricate nei registri leggendole dalla memoria (l'accesso in memoria è una operazione molto dispendiosa in termini di tempo), ma vengono fatte generare dalla AGU dell'APEcontroller (dal momento che sono note a tempo di compilazione) e vengono scritte nei registri, utilizzando un tipo di istruzione assembly la cui esecuzione avviene in tempi più brevi rispetto a quelli di una istruzione di caricamento dalla memoria.

6. Non sono presenti istruzioni che calcolino identità aritmetiche: i sottoalberi che rappresentano espressioni del tipo $x + 0$, $0 + x$, $x * 1$, $1 * x$, $x/1$ vengono riconosciuti e per essi non viene generato codice. Allo stesso modo non sono presenti istruzioni per il caricamento nei registri delle costanti 0 ed 1 per ogni tipo di dato, dal momento che esse sono allocate nei registri riportati in tabella 4.4 dal sistema operativo in fase di

inizializzazione.

Per descrivere l'algoritmo di generazione dell'assembly è necessario considerare che gli oggetti nodo, che rappresentano i nodi del grafo, e gli oggetti rap, che invece rappresentano variabili e costanti, contengono altre informazioni oltre a quelle viste finora. L'oggetto nodo può essere etichettato con un intero che rappresenta il registro virtuale ad esso associato in fase di produzione dell'assembly: tale registro conterrà il valore calcolato dalla operazione rappresentata dal nodo. Inoltre il nodo è caratterizzato da un tipo: si hanno così nodi di tipo integer, localint, real, double, complex, vector e usertype, che corrisponde ai tipi definiti per mezzo dei record.

La classe rap è in realtà una classe di base che contiene tutte le informazioni comuni alle classi derivate da essa: il tipo del simbolo, la sua dimensione (in termini di parole di memoria), il nodo su cui compare come etichetta.

Gli oggetti della classe *vrap* (derivata da rap) contengono tutte le informazioni necessarie al compilatore (oltre a quelle presenti in rap) per gestire le variabili: l'indirizzo di memoria della variabile ed altre informazioni necessarie a gestire gli array che verranno illustrate nella sezione 4.8. Un oggetto della classe vrap può rappresentare anche una variabile di tipo *register* di TAOmille (variabili che non vengono allocate in memoria), contenendo questa informazione nel campo tipo della rap. Essa viene trattata in maniera equivalente alle altre variabili tranne per il fatto che un suo uso come *r-value* all'interno di una espressione che non sia preceduto da un uso della stessa come *l-value* genera un messaggio di errore da parte del compilatore (si sta infatti cercando di utilizzare un valore indeterminato). Inoltre le vrap di tipo register non contengono l'informazione riguardo l'indirizzo di memoria e non vengono salvate in memoria alla fine del blocco di codice.

La classe *crap* (ovvero "rap costante"), è definita usando un *template*. Il risultato è che in questa classe il dato che rappresenta il valore della costante ha il tipo parametrizzato con quello della costante (integer, real, double, complex, vector).

Infine la classe *cloned_vrap* rappresenta gli elementi degli array, e verrà descritta nella sezione 4.8. Coerentemente con quello che avviene nel codice C++ nel seguito potremo riferirci ad un oggetto come appartenente alla classe rap, senza specificare a quale delle classi derivate da questa l'oggetto appartenga effettivamemente.

L'algoritmo di generazione prevede la visita del DAG con un ordinamento diverso da quello visto in precedenza illustrando il metodo di costruzione del grafo (vedi sezione 4.6.4). La generazione del codice avviene visitando ciascuno dei sottoalberi binari individuati dalle *teste* del DAG in ordine differito (*bottom-up*), marcando con un *flag* l'avvenuta generazione del codice per il nodo visitato di volta in volta in modo che la visita di uno stesso nodo non avvenga più di una volta.

In figura 4.13 è mostrato l'ordine di visita di un DAG nella fase di generazione dell'assembly. Dal momento che ciascun nodo ed i suoi figli rappre-

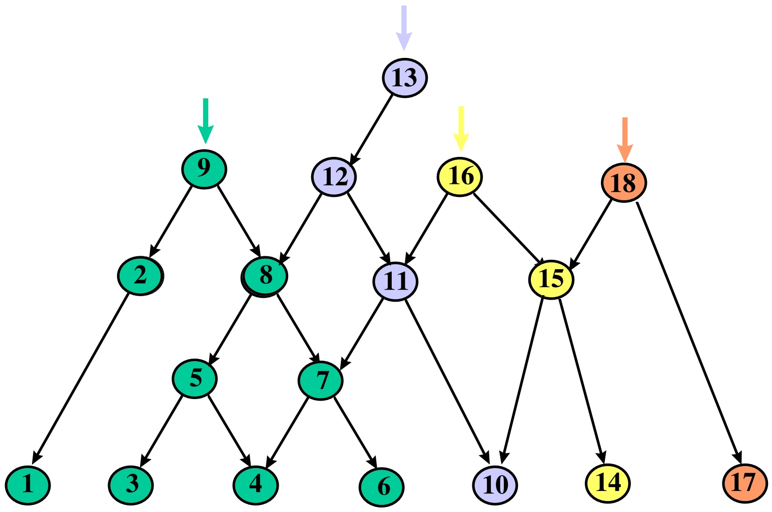


Figura 4.13: Ordinamento dei nodi del DAG nella generazione del codice

sentano rispettivamente una operazione ed i suoi operandi, si nota che l'ordinamento introdotto da questo tipo di visita garantisce che gli operandi siano valutati prima del loro utilizzo.

L'algoritmo di generazione può essere descritto nella seguente maniera:

1. Visita il DAG con l'ordinamento specificato. Per ogni nodo visitato:

   (a) Assegna un numero di registri al nodo pari alla dimesione del suo tipo richiamando la funzione new_reg().

   (b) Se il nodo rappresenta una istruzione che non deve produrre codice (ad esempio se consiste nel caricamento di uno dei valori costanti 1 o 0 dei vari tipi) termina la visita del nodo.

(c) In base al tipo del nodo ed all'operatore Pico che lo etichetta trova il corrispondente codice mnemonico e genera l'istruzione assembly secondo il formato previsto dal codice mnemonico.

2. Effettua una seconda visita con il medesimo ordine. Per ogni nodo visitato:

   (a) Richiama la funzione store() su ogni nodo, salvando in memoria le rap che etichettano il nodo come *l-value* in memoria.

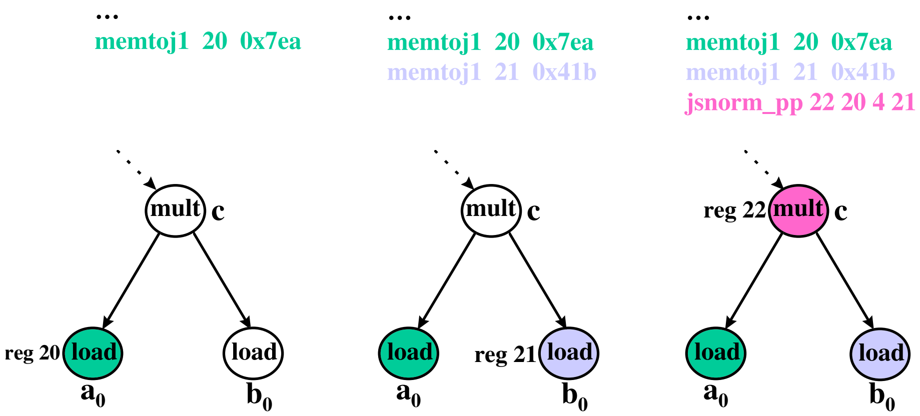


Figura 4.14: Esempio di generazione di codice

In figura 4.14 è illustrata la sequenza di generazione dell'assembly, ottenuta nel corso della prima visita, per il sottoalbero del DAG che rappresenta l'istruzione $c = a + b$. Nel corso della seconda visita del DAG viene ef-

```
MEMTOJ1         20      0x17d
MEMTOJ1         21      0x19a
MEMTOJ1         22      0x18b
JSNORM_PP       23      21      22      0
JSNORM_PP       24      20      4       23
MEMTOJ1         25      0x190
JSNORM_PP       26      23      25      0
JTOMEM1         0x23c   23
JTOMEM1         0x31f   23
JTOMEM1         0x59e   24
JTOMEM1         0x18b   26
```

Figura 4.15: Codice generato a partire dal DAG di figura 4.7

fettuato il salvataggio in memoria dell'unico *l-value* presente nel sottoalbero, attraverso l'istruzione assembly

```
JTOMEM1         0x82c   22
```

Il formato e le operazioni svolte da ciascuna istruzione sono riportate in appendice A.

Infine in figura 4.15 è riportato, come esempio, il codice prodotto a partire dal blocco rappresentato dal DAG di figura 4.7. Per semplicità di lettura si è immaginato che ciascuna variabile del blocco di programma considerato sia allocata in memoria ad un indirizzo la cui ultima cifra esadecimale sia lo stesso carattere dell'identificatore della variabile; g è allocata nella locazione 0x190.

4.7.2 L'algoritmo "normalizzatore"

L'algoritmo "normalizzatore", come è già stato detto, si occupa di modificare il DAG in modo da ricombinare le operazioni di somma e moltiplicazione per i tipi floating-point in una istruzione *normal*.

Questo algoritmo prende parte alla compilazione dopo la fase di generazione della rappresentazione intermedia e prima di quella di generazione dell'assembly. È possibile specificare in modo opzionale se esso debba essere richiamato o meno nel corso della compilazione. Ciò è stato fatto frequentemente nel corso del *debugging* del compilatore, che in questo modo produce sempre operazioni normal "ingenue" mappando direttamente somme e moltiplicazioni in questo modo:

$op_1 + op_2 \rightarrow op_1 * 1 + op_2$

$op_1 * op_2 \rightarrow op_1 * op_2 + 0$

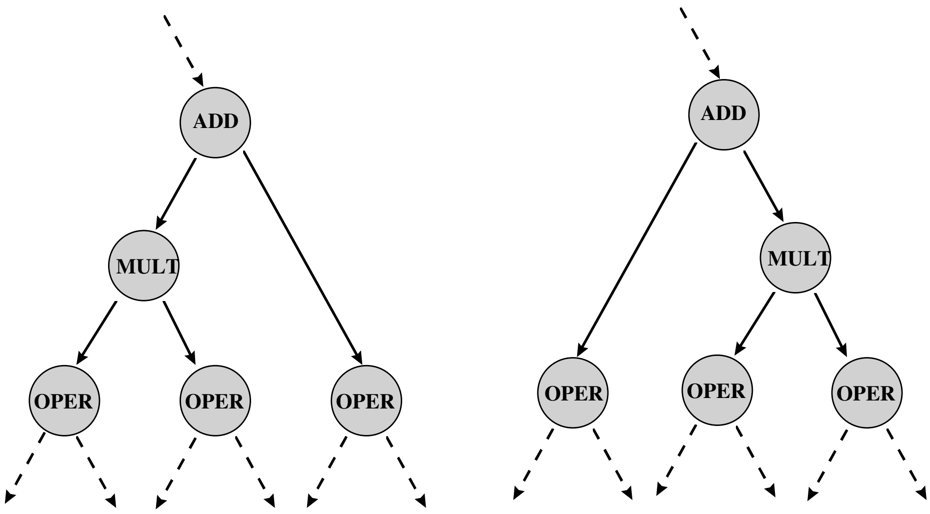


Figura 4.16: Somme e moltiplicazioni ricombinabili in operazioni normal

L'algoritmo visita il DAG ed effettua una serie di controlli sui sottoalberi del DAG, cercando quei sottoalberi, formati da tre nodi, che corrispondono a operazioni normal. Esso può essere descritto nella seguente maniera:

1. Visita ogni nodo del DAG nello stesso ordine adottato nella generazione del codice.

2. Controlla il tipo e l'operatore con cui è etichettato il nodo. Considera solo i nodi il cui tipo è real, double, complex o vector che sono etichettati con operatore ADD.

3. Se il nodo figlio sinistro del nodo considerato è etichettato con operatore MULT, ha un solo arco incidente e non ha rap che lo etichettano allora cambia il suo operatore da MULT a NPRT (parte di una normal) e quello del suo nodo padre da ADD a NORM.

   Altrimenti controlla se il figlio destro del nodo considerato è etichettato con operatore MULT, ha un solo arco incidente e non ha rap che lo etichettano; in caso affermativo cambia il suo operatore da MULT a NPRT e quello del suo nodo padre da ADD a NORM.

In figura 4.16 sono mostrati i sottoalberi contenenti una somma ed una moltiplicazione che vengono riconosciute dall'algoritmo come ricombinabili in una operazione normal.

In figura 4.17 sono mostrati due casi di sottoalberi che non possono essere tradotti in operazioni normal dall'algoritmo proposto.

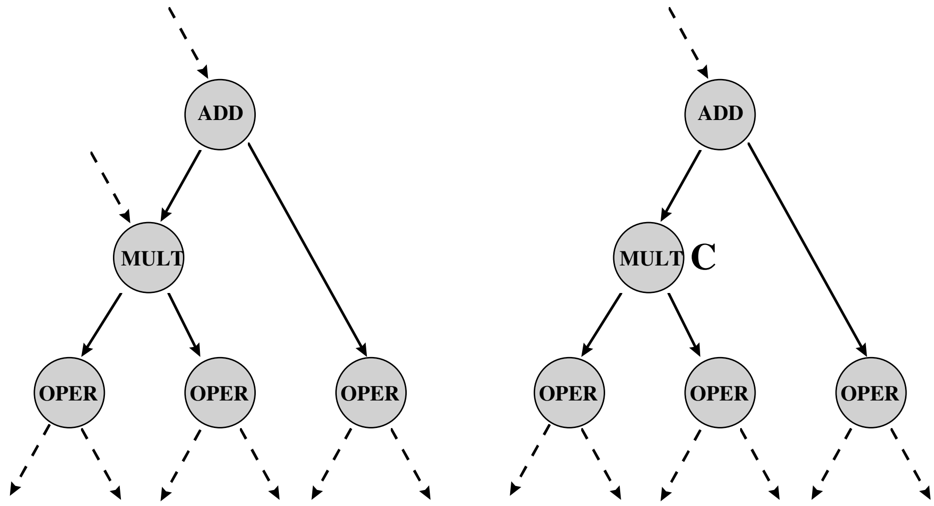


Figura 4.17: Somme e moltiplicazioni non ricombinabili in operazioni normal

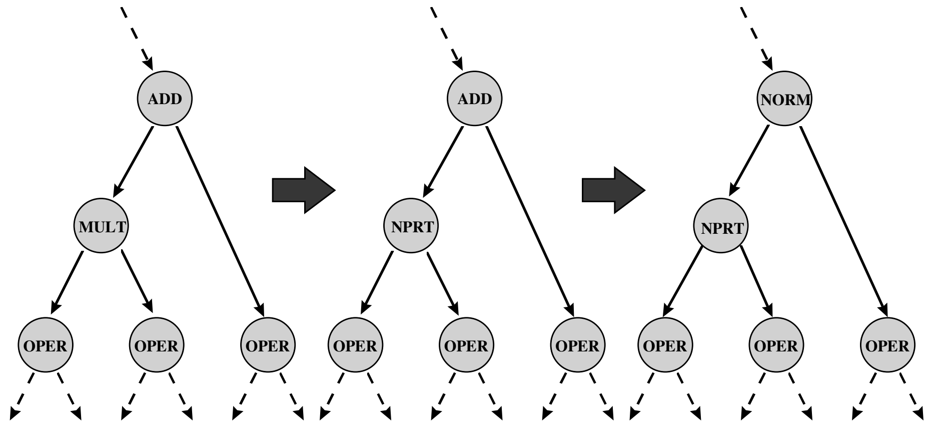


Figura 4.18: Trasformazioni sul DAG da parte dell'algoritmo

Infine in figura 4.18 vengono illustrate le trasformazioni sul DAG quando il sottoalbero sia stato riconosciuto come “normalizzabile”.

Il tipo della trasformazione effettuata dall’algoritmo rientra nelle cosidette *peephole optimization.* Queste corrispondono alle trasformazioni sul codice intermedio effettuate analizzando di volta in volta tutte le istruzioni e tenendo conto di una piccola “finestra” di codice centrata sull’istruzione in esame. Questo metodo ha il pregio di essere veloce, di contro però alcune possibili ottimizzazioni possono essere non rilevate.

L’ algoritmo generatore di assembly tratta i sottoalberi modificati dal “normalizzatore”:

1. evitando di assegnare registri ai nodi etichettati con NPRT e di produrre istruzioni assembly per essi;

2. generando una istruzione assembly normal nel caso in cui il nodo sia etichettato con un operatore NORM. I tre operatori necessari alla normal sono dati nell’ordine dai registri associati ai nodi nipoti del nodo NORM e figli del nodo NPRT e dal registro associato all’altro figlio del nodo NORM.

4.8 Algoritmi per la gestione dei tipi strutturati del linguaggio

In questa sezione vengono illustrate le principali variazioni agli algoritmi descritti nelle sezioni precedenti nel trattamento da parte del compilatore dei tipi strutturati del linguaggio TAOmille.

I tipi *complex* e *vector* vengono gestiti nella stessa maniera. Le differenze rispetto al caso generale risiedono essenzialmente nella funzione node_of(rap) per la creazione di nodi foglia, e nelle funzioni load() e store() che generano l'assembly per le letture e le scritture in memoria delle variabili. Le modifiche alla node_of(rap) sono dovute a due fattori:

1. Un nodo per un tipo complex o vector (etichettato rispettivamente con operatore BLDCMPL e BLDVCTR) contiene, oltre alla coppia di puntatori a nodo che rappresentano i figli del nodo, un'altra coppia di puntatori ai campi della variabile (.re e .im oppure .one e .two) che tiene traccia degli stessi.

2. Ai nodi di tipo real che rappresentano i campi di queste variabili devono essere assegnati registri contigui ed allineati secondo i vincoli visti nella sezione 3.3.2. Questa richiesta è soddisfatta dall'allocatore di registri new_reg() e dal fatto che, a differenza del caso generale, i registri vengono assegnati ai nodi all'atto della creazione di questi.

Il funzionamento della node_of(rap) per variabili di questo tipo può essere quindi riassunto nella seguente maniera:

1. Crea il nodo n_S etichettato con operatore BLDCMPL (BLDVCTR).

2. Richiama la node_of(rap) su entrambi i campi della variabile:

(a) se il nodo che rappresenta il campo non esiste allora lo crea assegnandogli il registro opportuno (pari o dispari).

(b) se il nodo che rappresenta il campo già esiste e non rappresenta il campo di un'altra variabile complex (vector) allora gli assegna il registro opportuno.

(c) infine se il nodo n_C che rappresenta il campo già esiste e rappresenta il campo di un'altra variabile complex (vector) allora crea un nodo etichettato con l'operatore COPY ed a questo assegna un registro opportuno e il nodo n_C come figlio sinistro.

3. Inserisce i puntatori ai nodi così trovati nella coppia di dati che nel nodo n_S rappresentano i campi della variabile complex (vector).

In questo modo nel DAG vengono introdotte ulteriori relazioni tra nodi, oltre a quelle già viste (parentela). Questa nuova informazione viene usata principalmente in due casi:

1. i campi di una variabile strutturata sono rappresentati nel DAG e su di essi possono essere eseguite tutte le ottimizzazioni.

2. i trasferimenti tra memoria e registri possono essere eseguiti in maniera più efficiente. Ad esempio qualora la visita di un nodo del DAG che rappresenta un campo di una variabile complex (vector) comporti la generazione di una istruzione di trasferimento verrà effettuato il trasferimento di tutta la variabile complex (vector) verificando nel DAG che

questa operazione sia compatibile con il resto del codice e raggruppando istruzioni di trasferimento che nel codice possono essere sparse.

Il trattamento delle variabili di tipo record è l'estensione di questi accorgimenti al caso di un numero n "qualsiasi" di campi, ciascuno di un qualsiasi tipo previsto dal linguaggio.

La gestione degli array è invece affrontata in maniera completamente differente. Ciò a causa del fatto che le variabili elementi di array presentano la caratteristica di non rendere esplicita la propria allocazione a tempo di compilazione. Un esempio potrà chiarire questa affermazione. Consideriamo il seguente blocco di istruzioni:

x = a[i]

a[j] = y

z = a[i]

Usando l'algoritmo per la costruzione del DAG illustrato nella sezione 4.6.3 la $a[i]$ rappresenterebbe una sottoespressione comune, il corrispondente blocco ottimizzato sarebbe quindi:

x = a[i]

z = x

a[j] = y

Si noti però che i due blocchi precedenti calcolano valori diversi di z nel caso che $i = j$ e $y \neq a[i]$. Il problema nasce dal fatto che quando si assegna ad un elemento di un array a, si può cambiare l'*r-value* della espressione $a[i]$ anche se

a ed i non sono cambiati nel frattempo. L'algoritmo scritto per il trattamento degli array, che fa uso di classi apposite, introduce alcune relazioni nel DAG che forzano un ordinamento nella elaborazione di questo secondo le seguenti regole:

- Qualsiasi valutazione di un elemento di un array o qualsiasi assegnazione a questo deve seguire l'assegnamento precedente a quell'array.
- Ogni assegnazione ad un elemento di un array a deve seguire qualsiasi valutazione precedente di un elemento dello stesso.

Questo significa che nel riordinamento del codice le istruzioni che usino (a destra o a sinistra) degli elementi di un array non possano scavalcarsi a vicenda.

Questo inoltre implica che, con riferimento a quanto detto nella sezione 4.7.1, lo *store* degli elementi di array venga effettuato durante la prima visita del DAG (e non insieme agli altri store nella seconda visita) nel caso che esista almeno una lettura di un qualsiasi elemento dell'array che segua, nel programma sorgente, una scrittura di un altro qualunque elemento.

Conclusioni

Questo lavoro ha riguardato il progetto e la realizzazione del compilatore ottimizzante del linguaggio TAOmille per il calcolatore parallelo APEmille.

Il linguaggio TAOmille rappresenta l'estensione del linguaggio TAO usato per APE100, del quale mantiene tutte le caratteristiche fondamentali e a cui sono stati aggiunti nuovi tipi di dato e nuovi costrutti che riflettono le innovazioni apportate all'architettura della macchina. Viceversa il compilatore, cioè il programma che traduce codice scritto in linguaggio TAOmille in codice assembly di APEmille, è stato completamente ripensato in funzione delle sue caratteristiche di ottimizzatore ed è stato realizzato usando il linguaggio di programmazione ad oggetti C++.

Esso è in grado, attraverso la generazione e l'elaborazione di una rappresentazione intermedia del codice sorgente, di effettuare delle ottimizzazioni volte a ridurre i tempi di esecuzione dei programmi. Nell'attuale stadio di sviluppo del progetto, il compilatore genera codice di basso livello a partire da singoli blocchi di programma. Le ottimizzazioni principali effettuate sul codice prodotto consistono: nella eliminazione delle sottoespressioni comuni;

nella minimizzazione dell'occupazione dei registri dei processori; nella gestione efficiente dei trasferimenti tra memoria e registri; nella sostituzione di istruzioni costose in termini di tempo di esecuzione con altre di minore impatto; nella eliminazione di codice ridondante (ad esempio il calcolo di espressioni del tipo $x + 0$ o $x \cdot 1$).

Un vincolo essenziale fissato come specifica del progetto, e rispettato nella successiva fase di realizzazione, è stato quello relativo alla piena compatibilità del compilatore con il codice TAO scritto per APE100. In questo modo si potranno ricompilare programmi e librerie già sviluppati e, senza dover apportare modifiche ai sorgenti, ottenenere codice eseguibile per APEmille.

Gli sviluppi nella realizzazione del progetto prevedono l'estensione delle ottimizzazioni locali, cioè sul singolo blocco, a quelle globali, ovvero relative a più blocchi o all'intero programma. Due aspetti particolarmente interessanti in tal senso sono l'analisi globale del flusso dei dati tra i blocchi e il *loop unrolling*. Il primo di questi consente di minimizzare il numero di accessi in memoria, il secondo, eliminando un numero consistente di operazioni di *branch*, contribuisce significativamente al mantenimento di un'alta percentuale di riempimento delle *pipeline* dei processori. La rappresentazione intermedia del codice sorgente adottata nel compilatore contiene tutte le informazioni necessarie alla realizzazione delle suddette estensioni.

La struttura modulare del compilatore consentirà, attraverso la riscrittura del *front-end*, una agevole migrazione verso un linguaggio di programmazione di alto livello orientato agli oggetti.

Appendici

Appendice A

Istruzioni assembly di APEmille

Gli operandi delle istruzioni assembly di APEmille possono essere di vario tipo, e per essi si possono usare numeri decimali o esadecimali a piacere: tutti i numeri utilizzati in un programma assembly (file.JASM) saranno implicitamente considerati decimali dall'assemblatore, mentre quelli esadecimali dovranno essere preceduti dal prefisso 0x.
Gli argomenti opzionali vengono indicati con le parentesi quadre.
Un operando puo' essere:

1) un indirizzo globale di memoria calcolato da APEController: mop1, mop2, mop3; il piu' generico Memory Operand e' mop3, che ha il seguente formato: base_address.Ra.Rb
L'indirizzo di memoria e' dato da: base_address + @(Ra) + @(Rb),

dove @(x) indica il contenuto del registro x.
Il base_address relativo ad un indirizzo viene implicitamente preso come un numero assoluto, ma cio‘ puo‘ essere scritto esplicitamente affiancandogli il simbolo ‘#’ (per es. #33.4.5); gli indirizzi rilocabili per la memoria dati di Tarzan sono preceduti dal simbolo ‘%’ (per es. %0x200.63.127), mentre quelli rilocabili per la memoria dati di Jane sono preceduti dal simbolo ‘^’ (per es. ^41.256.511).
Un mop1 puo‘ essere inteso anche come numero immediato nell’istruzione TAGU1, che appunto genera questo numero tramite la Address Generation Unit (AGU) e lo carica in un registro del Register File di Tarzan.

2) un registro di Jane o di APEController: regj, regjp, regt; regj e regjp sono numeri assoluti compresi fra 0 e 511 che indicano i registri del Register File di Jane (regjp solo quelli pari), mentre regt (numeri assoluti fra 0 e 127) indicano i registri del Register File di APEController: nell’assembly APEmille i registri vengono indicati semplicemente dal loro numero sia per Jane che per APEController, senza l’aggiunta di alcun simbolo.

3) una label: lab; il numero d’ordine di una label deve essere preceduto dal simbolo * (per es. LABEL *0x100): queste istruzioni

non generano codice eseguibile, ma possono essere utilizzate come punto di riferimento dalle istruzione di salto
(per es. JUMP *0x100.0) o caricate dalla TAGU1 in un registro (tipicamente nelle routine del sistema operativo).

5) un indirizzo di memoria dati per le costanti e le stringhe: adrt, adrj; anche per questi indirizzi possono essere specificate le informazioni di rilocabilita‘
(per es. JCONST ^100 {2.3}).

6) un offset (contenuto in un registro) per il calcolo dell'indirizzo di memoria locale: lof; un offset viene specificato indicando il registro del Register File di Jane che lo contiene.

7) il numero di dati (burst) da leggere/scrivere in una istruzione di trasferimento dati: repj; il burst deve essere preceduto dal simbolo ‘:’ (per es. MEMTOJ1 10 :3 0x100 0).

8) una costante per le memorie dati di Jane ed APEController: constj, const; le costanti possono essere interi (per es. 40), indirizzi assoluti e rilocabili per le memorie dati composti dal

solo base_address (per es. #100, %33, ^44), numeri floating point in singola e doppia precisione (per es. {2.3} e {D2.3}) o infine una label.

9) una stringa: string; le stringhe devono essere precedute e seguite dal simbolo " (per es. "Ciao").

10) le coordinate di un nodo o di un gruppo di nodi: slice; le costanti e le stringhe possono essere caricate su un solo Jane o su un solo Tarzan, oppure su un gruppo (slice) di Jane o di Tarzan. Se la slice non viene specificata, si assume implicitamente l'intera macchina. Il formato per specificare le slice e' (x1 y1 z1 ; x2 y2 z2) per un gruppo di nodi, oppure (x y z) per un solo nodo (per es. JCHAR 0x200 (0 0 0; 1 1 1) "pippo").

11) un pragma: pra; il pragma indica un gruppo di registri o di label coinvolti in una direttiva per FLINT ed e' caratterizzato dalla presenza dei simboli '[' e ']'
(per es. PRAGMA_ALLOC_JBUNDLE [10, 0x30, 45, 67]).

Le istruzioni che coinvolgono Tarzan (ovvero l'APEController), Jane o Cheetah presentano come prima lettera T, J o C rispettivamente; le istruzioni della ALU di Jane, i cui registri operandi contengono numeri reali in standard IEEE, vengono distinte poi da una successiva lettera posta in seconda posizione, ovvero:

- la lettera D indica come operandi numeri reali in doppia precisione,

- la lettera C indica come operandi numeri complessi (parte reali e parte immaginaria sono numeri reali in singola precisione),

- la lettera S indica come operandi numeri reali in singola precisione,

- la lettera V indica come operandi coppie di numeri reali in singola precisione,

- la lettera I indica come operandi numeri interi.

Le istruzioni di trasferimento tra memoria dati e Register File contengono la J per il RF di Jane e T per il RF contenuto nell'APEController di Tarzan.

Istruzioni di Tarzan

Legenda

regt0, regt1, regt2, regt3 sono di tipo regt.

(addr) indica la locazione di memoria indirizzata da addr.

addr indica il valore dell'indirizzo di memoria.

--> e <-- vengono utilizzati per indicare il caricamento di un registro o di una locazione di memoria.

Istruzioni aritmetiche

```
TADD    regt3   regt0   regt1
!! regt3 = regt0 + regt1
TSUB    regt3   regt0   regt1
!! regt3 = regt0 - regt1
TMUL    regt3   regt0   regt1
!! regt3 = regt0 * regt1
TMULA   regt3   regt0   regt1
!! regt3 = regt0 * regt1
TADD3   regt3   regt0   regt1   regt2
```

```
!! regt3 = regt0 + regt1 + regt2

TSUB3   regt3   regt0   regt1   regt2

!! regt3 = regt0 - regt1 - regt2

TDIV    regt3   regt0   regt1

!! regt3 = regt0 / regt1
```

Istruzioni logiche bitwise

```
TAND    regt3   regt0   regt1

!! regt3 = regt0&regt1

TOR     regt3   regt0   regt1

!! regt3 = regt0 | regt1

TXOR    regt3   regt0   regt1

!! regt3 = regt0 ^ regt1

TNAND   regt3   regt0   regt1

!! regt3 = ~(regt0&regt1)

TNOR    regt3   regt0   regt1

!! regt3 = ~(regt0 | regt1)

TASH    regt3   regt0   regt1

!! regt3 = regt0 shiftato aritmet. di regt1

TLSH    regt3   regt0   regt1

!! regt3 = regt0  shiftato logicam. di regt1
```

```
TROT    regt3   regt0   regt1

!! regt3 = regt0 ruotato di regt1

TEQ     regt0   regt1

!! regt0 = regt1

TNE     regt0   regt1

!! regt0 != regt1

TGT     regt0   regt1

!! regt0 > regt1

TLT     regt0   regt1

!! regt0 < regt1

TGE     regt0   regt1

!! regt0 >= regt1

TLE     regt0   regt1

!! regt0 <= regt1


Istruzioni di controllo di flusso


TINCCOMP        regt0   regt1   regt2

!! regt0 + regt1 <= regt2

TANY

!! GBIf = 1

TNONE
```

```
!! GBIf = 0

TCNB

!! non ancora implementata

TGETPC          regt

!! regt <-- PC (Program Counter)

GOSUB           mop2    regt

!! salta a mop2 e regt <-- PC  di ritorno

TLOADSR         mop3

!! Status Register <-- (mop3)

TSTORESR        mop3

!! Status Register --> (mop3)

JUMP            mop2

!! salta all'indirizzo mop2

JUMPIF          mop2

!! salta all'indirizzo mop2

JUMPIFNOT       mop2

!! salta all'indirizzo mop2

LABEL           lab

!! indica l'inizio di una nuova sez.

HALT

!! interrompe l'esecuzione del prog.
```

```
Istruzione di generazione di indirizzi

TAGU1           regt    mop1
!! regt <-- mop1
TAGU2           regt    mop2
!! regt <-- mop2
TAGU3           regt    mop3
!! regt <-- mop3

Istruzioni di trasferimento dati tra memoria e Register File

MEMTOT1         regt    mop1
!! regt <-- (mop1)
MEMTOT2         regt    mop2
!! regt <-- (mop2)
MEMTOT3         regt    mop3
!! regt <-- (mop3)
TTOMEM1         mop1    regt
!! regt --> (mop1)
TTOMEM2         mop2    regt
!! regt --> (mop2)
TTOMEM3         mop3    regt
```

!! regt --> (mop3)

Istruzioni di Jane

Legenda

regj0, regj1, regj2, regj3 indicano registri di tipo regj.
Con ln, rn e an (n = 0,1,2,3) indichiamo registri di tipo regjp contenenti la parte bassa di un dato floating in doppia precisione (l= low), la parte reale di un dato complesso (r = reale) ed infine un dato floating in singola precisione (a) rispettivamente nelle istruzioni della ALU di Jane in doppia precisione, complesse e vettoriali;
con rn, in e bn indichiamo invece i registri contenenti la parte alta del dato floating in doppia precisione (h = high), la parte immaginaria del dato complesso (i = immaginario) ed infine un dato floating in singola precisione (b). Si ha sempre hn = ln+1, in = rn+1 e bn = an+1.
TOS indica il primo bit dello stack dell'IF (Top Of Stack), mentre TOS - 1 quello immediatamente sottostante.
s.p. indica operandi in singola precisione, d.p. operandi in doppia precisione, c. operandi complessi e i. operandi interi.

Significato analogo al caso precedente assumono i simboli <-- e -->.

Istruzioni aritmetiche in singola precisione

```
JSNORM_PP       regj3   regj0   regj1   regj2
!! regj3 = +(regj0*regj1+regj2)
JSNORM_PM       regj3   regj0   regj1   regj2
!! regj3 = +(regj0*regj1-regj2)
JSNORM_MP       regj3   regj0   regj1   regj2
!! regj3 = -(regj0*regj1+regj2)
JSNORM_MM       regj3   regj0   regj1   regj2
!! regj3 = -(regj0*regj1-regj2)
JSNORMABS_PP    regj3   regj0   regj1   regj2
!! regj3 = |regj0*regj1+regj2|
JSNORMABS_PM    regj3   regj0   regj1   regj2
!! regj3 = |regj0*regj1-regj2|
JSADD3_PPP      regj3   regj0   regj1   regj2
!! regj3 = +(regj0*regj1+regj2)
JSADD3_PPM      regj3   regj0   regj1   regj2
!! regj3 = +(regj0+regj1-regj2)
JSADD3_PMP      regj3   regj0   regj1   regj2
```

```
!! regj3 = +(regj0-regj1+regj2)

JSADD3_PMM      regj3   regj0   regj1   regj2

!! regj3 = +(regj0-regj1-regj2)

JSADD3_MPP      regj3   regj0   regj1   regj2

!! regj3 = -(regj0+regj1+regj2)

JSADD3_MPM      regj3   regj0   regj1   regj2

!! regj3 = -(regj0+regj1-regj2)

JSADD3_MMP      regj3   regj0   regj1   regj2

!! regj3 = -(regj0-regj1+regj2)

JSADD3_MMM      regj3   regj0   regj1   regj2

!! regj3 = -(regj0-regj1-regj2)

JSMULT_PP       regj3   regj0   regj1   regj2

!! regj3 = +(regj0+regj1)*regj2

JSMULT_PM       regj3   regj0   regj1   regj2

!! regj3 = +(regj0-regj1)*regj2

JSMULT_MP       regj3   regj0   regj1   regj2

!! regj3 = -(regj0+regj1)*regj2

JSMULT_MM       regj3   regj0   regj1   regj2

!! regj3 = +(regj0-regj1)*regj2

JSMULTABS_PP    regj3   regj0   regj1   regj2

!! regj3 = |(regj0+regj1)*regj2|

JSMULTABS_PM    regj3   regj0   regj1   regj2
```

```
!! regj3 = |(regj0-regj1)*regj2|
```

Istruzioni aritmetiche vettoriali

(con coppie di operandi in singola precisione)

```
JVNORM_PP     a3   a0   a1   a2
!!(a3,b3)=+((a0*a1+a2),(b0*b1+b2))
JVNORM_PM     a3   a0   a1   a2
!!(a3,b3)=+((a0*a1-a2),(b0*b1-b2))
JVNORM_MP     a3   a0   a1   a2
!!(a3,b3)=-((a0*a1+a2),(b0*b1+b2))
JVNORM_MM     a3   a0   a1   a2
!!(a3,b3)=-((a0*a1-a2),(b0*b1-b2))
JVNORMABS_PP  a3   a0   a1   a2
!!(a3,b3)=|((a0*a1+a2),(b0*b1+b2))|
JVNORMABS_PM  a3   a0   a1   a2
!!(a3,b3)=+((a0*a1+a2),(b0*b1+b2))
```

Istruzioni aritmetiche complesse

```
JCNORM_PP     r3    r0    r1    r2
!!(r3+ji3)=+((r0+ji0)*(r1+ji1)+(r2+ji2))
```

```
JCNORM_PM     r3   r0   r1   r2

!!(r3+ji3)=+((r0+ji0)*(r1+ji1)-(r2+ji2))

JCNORM_MP     r3   r0   r1   r2

!!(r3+ji3)=-((r0+ji0)*(r1+ji1)+(r2+ji2))

JCNORM_MM     r3   r0   r1   r2

!!(r3+ji3)=-((r0+ji0)*(r1+ji1)-(r2+ji2))

JCNORMABS_PP  r3   r0   r1   r2

!!(r3+ji3)=|(r0+ji0)*(r1+ji1)+(r2+ji2)|

JCNORMABS_PM  r3   r0   r1   r2

!!(r3+ji3)=|(r0+ji0)*(r1+ji1)-(r2+ji2)|
```

Istruzioni aritmetiche in doppia precisione

```
JDNORM_PP     l3   l0   l1   l2

!!(l3,h3)=+((l0,h0)*(l1,h1)+(l2,h2))

JDNORM_PM     l3   l0   l1   l2

!!(l3,h3)=+((l0,h0)*(l1,h1)-(l2,h2))

JDNORM_MP     l3   l0   l1   l2

!!(l3,h3)=-((l0,h0)*(l1,h1)+(l2,h2))

JDNORM_MM     l3   l0   l1   l2

!!(l3,h3)=-((l0,h0)*(l1,h1)-(l2,h2))

JDNORMABS_PP  l3   l0   l1   l2
```

```
!!(l3,h3)= |(l0,h0)*(l1,h1)+(l2,h2)|

JDNORMABS_PM  l3    l0    l1    l2

!!(l3,h3)= |(l0,h0)*(l1,h1)-(l2,h2)|
```

Istruzioni aritmetiche fra interi locali

```
JIADD        regj3   regj0   regj1

!! regj3 = regj0 + regj1

JISUB        regj3   regj0   regj1

!! regj3 = regj0 - regj1

JIMULT       regj3   regj0   regj1

!! regj3 = regj0 * regj1
```

Istruzioni logiche e bitwise

```
JIAND        regj3   regj0   regj1

!! regj3 = regj0 & regj1

JINAND       regj3   regj0   regj1

!! regj3 = ~(regj0 & regj1)

JIOR         regj3   regj0   regj1

!! regj3 = regj0 | regj1

JIXOR        regj3   regj0   regj1
```

```
!! regj3 = regj0 ^ regj1

JILSH       regj3   regj0   regj1

!! regj3 = regj0 shiftato logicam. di regj1

JIASH       regj3   regj0   regj1

!! regj3 = regj0 shiftato aritmet. di regj1

JIROT       regj3   regj0   regj1

!! regj3 = regj0 ruotato di regj1

JIBITCOUNT  regj3   regj0

!! regj3 = # di bit ad 1 in regj0

JILEADONE   regj3   regj0

!! regj3 = primo bit piu` significativo ad 1 in regj0


Istruzioni di conversione di tipo


JSTOD     regjp   regj

!! regjp = single_to_double (regj)

JSTOI     regj3   regj0

!! regj3 = single_to_integer (regj0)

JDTOS     regj    regjp

!! regj = double_to_single (regjp)

JDTOI     regj    regjp

!! regj = double_to_integer (regjp)
```

```
JITOS     regj3   regj0

!! regj3 = integer_to_single (regj0)

JITOD     regjp   regj

!! regjp = integer_to_double (regj)

JSTKTOI   regj

!! regj <-- TOS
```

Istruzioni per la Look Up Table

```
JEXP      regj3   regj0

!! regj3 = primo termine dell'approssimazione

per il calcolo di EXP(X)

JINV      regj3   regj0

!! regj3 = primo termine dell'approssimazione

per il calcolo di 1/X

JINVSQRT  regj3   regj0

!! regj3 = primo termine dell'approssimazione

per il calcolo di 1/SQRT(X)

JLOGE     regj3   regj0

!! regj3 = esponente di LOG(regj0) (base 2)

JLOGM     regj3   regj0

!! regj3 = mantissa di LOG(regj0) (base 2)
```

```
JLSB      regj3   regj0

!! regj3 = bit meno significativo in regj0

JLUTONE   regj

!! regj  <-- 1 in singola precisione (0x3f800000)

JLUTMOVE  regj3   regj0

!! regj3 = regj0 passando per la LUT
```

Istruzioni per il controllo delle condizioni locali

e dello stack dell'IF

```
JSPUSHEQ  regj0   regj1

!! TOS <-- (regj0 = regj1) con regj0 e regj1 in s.p.

JSPUSHNE  regj0   regj1

!! TOS <-- (regj0 != regj1)      ''          ''

JSPUSHGT  regj0   regj1

!! TOS <-- (regj0 > regj1)       ''          ''

JSPUSHLT  regj0   regj1

!! TOS <-- (regj0 < regj1)       ''          ''

JSPUSHGE  regj0   regj1

!! TOS <-- (regj0 >= regj1)      ''          ''

JSPUSHLE  regj0   regj1

!! TOS <-- (regj0 <= regj1)      ''          ''
```

```
JDPUSHEQ  regjp0  regjp1
!! TOS <-- (regjp0 = regjp1) con regjp0 e regjp1 in d.p.
JDPUSHNE  regjp0  regjp1
!! TOS <-- (regjp0 != regjp1)    ``          ``
JDPUSHGT  regjp0  regjp1
!! TOS <-- (regjp0 > regjp1)     ``          ``
JDPUSHLT  regjp0  regjp1
!! TOS <-- (regjp0 < regjp1)     ``          ``
JDPUSHGE  regjp0  regjp1
!! TOS <-- (regjp0 >= regjp1)    ``          ``
JDPUSHLE  regjp0  regjp1
!! TOS <-- (regjp0 <= regjp1)    ``          ``
JCPUSHEQ  regjp0  regjp1
!! TOS <-- (regjp0 = regjp1) con regjp0 e regjp1 c.
JCPUSHNE  regjp0  regjp1
!! TOS <-- (regj0 != regj1)      ``          ``
JIPUSHEQ  regj0   regj1
!! TOS <-- (regj0 = regj1)   con regj0 e regj1 i.
JIPUSHNE  regj0   regj1
!! TOS <-- (regj0 != regj1)      ``          ``
JIPUSHGT  regj0   regj1
!! TOS <-- (regj0 > regj1)       ``          ``
```

```
JIPUSHLT  regj0   regj1

!! TOS <-- (regj0 < regj1)       ‘‘          ‘‘

JIPUSHGE  regj0   regj1

!! TOS <-- (regj0 >= regj1)      ‘‘          ‘‘

JIPUSHLE  regj0   regj1

!! TOS <-- (regj0 <= regj1)      ‘‘          ‘‘

JPOP

!! estrae il TOS shiftando tutti i bit dello

Stack di 1 verso il TOS

JSTKAND

!! TOS = TOS & (TOS-1) e shifta tutti i rimanenti

bit dello Stack di 1 verso il TOS

JSTKOR

!! TOS = TOS | (TOS-1) e shifta tutti i

rimanenti bit dello Stack di 1 verso il TOS

JSTKXOR

!! TOS = TOS ^ (TOS-1) e shifta tutti i

rimanenti bit dello Stack di 1 verso il TOS

JSTKANDBIS

!! TOS =  TOS & (TOS-1)

JSTKREPUSH

!! TOS = TOS-1
```

```
JSTKSWAP
!! scambia fra loro i valori di TOS e TOS-1
JSTKNOT
!! TOS = ~(TOS)
JSTKRESET
!! pone tutti i bit dello Stack a zero
```

Istruzioni di trasferimento dati fra memoria e Register File

```
MEMTOJ1    regj    [repj]  mop1    lof
!! repj dati presi in memoria a partire da (mop1+lof)
e posti nel RF a partire da regj
MEMTOJ2    regj    [repj]  mop2    lof
!! repj dati presi in memoria a partire da (mop2+lof)
e posti nel RF a partire da regj
MEMTOJ3    regj    [repj]  mop3    lof
!! repj dati presi in memoria a partire da (mop3+lof)
e posti nel RF a partire da regj
JTOMEM1    mop1    lof     regj    [repj]
!! repj dati presi a partire da regj e posti in memoria a
partire da (mop1+lof)
JTOMEM2    mop2    lof     regj    [repj]
```

```
!! repj dati presi a partire da regj e posti in memoria a
partire da (mop2+lof)
JTOMEM3    mop3    lof     regj    [repj]
!! repj dati presi a partire da regj e posti in memoria a
partire da (mop3+lof)
JGETSR     regjp
!! rejgp <-- Status Register (a 64 bit)
JPUTSR     regjp
!! regjp --> Status Register (a 64 bit)
```

Istruzioni di trasferimento dati fra i tre chip
e simulazione eccezioni

```
JTOC0           regj
!! primi 8 bit di regj a Cheetah
JTOC1           regj
!! secondi 8 bit di regj a Cheetah
JTOC2           regj
!! terzi 8 bit di regj a Cheetah
JSEND   mop3    regj    [repj]
!! ancora non implementata
JREC            regj    [repj]
```

```
!! ancora non implementata

CTOT            regt

!! regt <-- dato a 32 bit da Cheetah

TTOJ            regj    mop3

!! regj <-- mop3

JEX

!! Jane exception

TEX

!! Tarzan exception
```

Direttive per il compilatore

```
PRAGMA_ALLOC_JREG       pra

!! rende statico il regj virtuale in pra

PRAGMA_ALLOC_JBUNDLE    pra

!! rende statici i regj virtuali in pra

PRAGMA_FREE_JREG        pra

!! libera il regj in pra

PRAGMA_FREE_TBUNDLE     pra

!! libera i regj in pra

PRAGMA_ALLOC_TREG       pra

!! rende statico il regt virtuale in pra
```

```
PRAGMA_ALLOC_TBUNDLE    pra

!! rende statici i regt virtuali in pra

PRAGMA_FREE_TREG        pra

!! libera il regt in pra

PRAGMA_FREE_TBUNDLE     pra

!! libera i regt in pra

PRAGMA_JN_REGSET        pra

!! specifica i regj di un trasferimento dati con la memoria

PRAGMA_COMPUTE_PER      pra

!! calcola le performance nelle sezioni di codice che

iniziano con le label contenute  in pra
```

Direttive per il loader

```
TCONST  adrt    [slice] const

!! const --> (adrt) nei Tarzan in slice

TCHAR   adrt    [slice] string

!! string --> (adrt) nei Tarzan in slice

JCONST  adrj    [slice] constj

!! constj --> (adrj) nei Jane in slice

JCONSTD adrj    [slice] constj  constj

!! due constj o una constj double --> (adrj) nei Jane in slice
```

```
JCHAR   adrj    [slice] string

!! string --> (adrj) nei Jane in slice
```

Bibliografia

[1] F. Zuliani, *Metodi Numerici per il Calcolo dello Spettro di Massa nella QCD*, Tesi di Laurea in Fisica, Università di Roma La Sapienza, (Luglio 1995).

[2] M. Mézard, G. Parisi and M. A. Virasoro, *"Spin Glass Theory and Beyond"*, World Scientific (Singapore, 1987).

[3] B. Wilkinson, *Computer Architecture Design and Performance*, Prentice Hall (1996).

[4] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers Priciples, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[5] S.A. Mahlke, *Design and Implementation of a Portable Global Code Optimizer*, Thesis for the degree of Master of Science, UIUC, 1992.

[6] J.L. Hennesy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1990.

[7] S. Carr, K. Kennedy, *Improving the Ratio of Memory Operations to Floating-Point Operations in Loops*, ACM TOPLAS, Vol. 16, No. 6, November 1994.

[8] J.W. Davidson, S. Jinturkar, *Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler*, Lectures Notes in Computer Science, 1060, Springer, 1996.

[9] C. Allton et al., *Physics Projects on APEmille*, Nota Interna, Università di Roma La Sapienza, (September 1994).

[10] A. Bartoloni et al., *APEmille: A Parallel Processor in the Teraflops Range*, Nota Interna, Università di Roma La Sapienza, (September 1994).

[11] A. Bartoloni et al., *Addendum to the APEmille Proposal*, Nota Interna, Università di Roma La Sapienza, (1995).

[12] F. Aglietti et al., *The APEmille Project: a Status Report*, Nota Interna, Università di Roma La Sapienza, (May 1996).

[13] E. Panizzi for the APE collaboration, *APEmille: a parallel processor in the teraflop range*, Proceedings of International Symposium on Lattice Field Theory, St. Louis MO, USA (June 1996).

[14] F. Aglietti et al., *Architecture, design trade-offs and programmability in the APEmille teraflop supercomputer*, Nota Interna, Università di Roma La Sapienza, (September 1996).

[15] G.J. Chaitin et al., *Register Allocation and Spilling via Graph Coloring*, ACM SIGPLAN Notices 17:6, 201-207 (1982).

[16] P. Briggs, K.D. Cooper, L. Torzcon, *Improvements to Graph Coloring Register Allocation*, ACM TOPLAS, Vol. 16, No. 3, 428-455, (May 1994).

[17] R. Gupta, M.L. Soffa, D. Ombres, *Efficient Register Allocation via Coloring Using Clique Separators*, ACM TOPLAS, Vol. 16, No. 3, 370-386, (May 1994).

[18] S. Cabasino, P.S. Paolucci, G.M. Todesco, *Dynamic Parsers and Evolving Grammars*, ACM SIGPLAN Notices, Vol. 27, No. 11 (November 1992).

[19] S. Cabasino, P.S. Paolucci, G.M. Todesco, *Zz V4.0*, documentazione interna INFN, A100/ZZ/s-04.

[20] B.W. Kernighan, D.M. Ritchie, *Linguaggio C.* Gruppo Editoriale Jackson, 1989.

[21] B. Stroustrup, *Il Linguaggio C++*. Addison-Wesley, 1995.

[22] B. Eckel, *Programmare in C++*. McGraw-Hill, 1993.

[23] *TAO Reference Manual*, documentazione dei Sistemi Quadrics, Alenia Spazio.

[24] N. Cabibbo, P.S. Paolucci, *SIMD Algorithm for Matrix Transposition*, International Journal of Modern Physics C, Vol. 6, No. 2 (1995) 183-195.

[25] A.V. Aho, S.C. Johnson, J.D. Ullman, *Code generation for expressions with common subexpressions*, J. ACM 24:1, 146-160.

[26] A.V. Aho, M. and T. Ganapathi, W.K. Steven, *Code generation using tree matching and dynamic programming*, ACM TOPLAS, Vol. 11, No. 4, 491-516, (Oct. 1989).

[27] A.V. Aho, J.E. Hopcroft, J.D Ullman, *The design and analysis of computer alghorithms*, Addison-Wesley (1974).

[28] E. Horowitz, S. Sahni, *Fundamentals of computer alghorithms*, Pitman Publishing (1978, London).