Università degli Studi di Roma "La Sapienza"

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

TESI DI LAUREA

Ipotesi di Assemblatore e di Sistema Operativo locale ad un nodo di una rete *apeNEXT*

Laureando: Federico Di Carlo

Matr. 11103355

Relatore: Prof. Nicola Cabibbo

Correlatore: Prof. Giuseppe De Biase

Anno Accademico 1999-2000

Indice

Elenco degli acronimi

APE Array Processor Experiment

BFD Binary File Descriptor

BN Bridge Node

ELF Executable and Linking Format

Flops Floating point operations Per Second

GAS GNU Assembler

GASP GNU Assembler Prepocessor

INFN Istituto Nazionale di Fisica Nucleare

IOP Input/Output Packet

LD GNU Linker

MIMD Multiple Instruction-Multiple Data

NAS Next Assembler

NASP Next Assembler Preprocessor

NLD Next Linker

PB Procesing Board

PLD Programmable Logical Device

QCD Cromodinamica Qauntistica

SF Simulatore Funzionale

SIMD Single Instruction-Multiple Data

VHDL VHSIC Hardware Definition Language

VLIW Very Long Instruction Word

Introduzione

Nell'ambito delle architetture degli elaboratori elettronici ha rivestito un ruolo importantissimo negli ultimi anni lo studio di eventuali allontanamenti dal modello di Von Neuman. Esso si basa sull'idea di una singola unità di controllo che segue un flusso di istruzioni scritto sequenzialmente in una memoria. Si può però pensare di far eseguire all'unità le istruzioni secondo le dipendenze sui dati che devono essere manipolati, come nelle macchine a flusso di dati (*data flow*). Oppure si può aumentare il numero di unità di controllo facendo seguire ad ognuna di esse un flusso differente di istruzioni o di dati, arrivando agli *elaboratori paralleli*.

Proprio con quest'ultima prospettiva in mente esistono dei progetti per sfruttare tale tecnologia in vari campi in cui la potenza computazionale risulta un elemento chiave.

Uno di questi progetti è **APE**, nato e svolto quasi interamente in Italia. Esso ha lo scopo di costruire degli elaboratori paralleli dedicati alla simulazione di sistemi fisici complessi e, negli ultimi due decenni, ha raggiunto risultati notevoli nel campo divenendo ben presto un polo universalmente riconosciuto per quel che riguarda la ricerca nel settore della *parallel computing*.

Il lavoro descritto in questa tesi si è svolto nei laboratori della sezione Roma 1 dell'**Istituto Nazionale di Fisica Nucleare** (INFN).

Scopo del lavoro è stato studiare le macchine della famiglia APE, ideate e realizzate principalmente nei laboratri dell'INFN, per realizzare una ipotesi di sistema operativo per l'ultima generazione di questi elaboratori: *apeNEXT*.

Ovviamente la fase preliminare del lavoro è consistita, per lo più, nella formazione del background cognitivo essenziale per poter lavorare su questo progetto. Quindi, in particolar modo, l'approfondimento delle tematiche legate alle architetture parallele per gli elaboratori e lo studio delle caratteristiche delle macchine APE precedenti, con un occhio di riguardo all'aspetto software, ma senza tralasciare totalmente la progettazione hardware delle macchine, rivelatasi altamente stimolante. Il tutto si è svolto grazie ad una interessante scelta bibliografica (ovviamente qui riportata) e alla partecipazione al lavoro del gruppo, sia in laboratorio sia nei numerosi incontri avuti con i partecipanti al progetto in tutta Europa.

Vista la recente nascita del progetto *apeNEXT* (inizio 2000) arrivare alla ideazione di un sistema operativo per esso passa non solo attraverso la formazione qui accennata ma anche per un continuo aggiornamento ed una attenta partecipazione alla fase di definizione della architettura della nuova macchina. Questo perché durante la fase di progettazione sono numerose le modifiche apportate ad ognuno degli aspetti del progetto, coinvolgendo, inevitabilmente, anche la stesura del sistema operativo.

Inoltre gran parte degli strumenti necessari alla effettiva stesura di una componente software per *apeNEXT* non erano disponibili al momento dell'avvio di questi lavori. Ad esempio era appena stata definita una ipotesi di linguaggio macchina, aspetto sul quale avrebbero poi poggiato numerosi altri strumenti necessari a questo lavoro, come l'assemblatore, il simulatore software, i compilatori, etc. In particolare il primo di essi risultava fondamentale per l'avvio dei lavori, introducendo la necessità di definirlo ed implementarlo per poter succcessivamente sviluppare gli altri strumenti accennati e, quindi, effettuare dei test sul nascente sistema operativo. È per questo che la prima parte del lavoro qui descritto riguarda proprio la creazione di un assemblatore *open source* per *apeNEXT*. Esso è un adattamento per la piattaforma APE dell'assemblatore GNU presente in numerosi sistemi e quindi notevolmente diffuso.

Una volta creato l'assemblatore è possibile utilizzarlo per scrivere codice per *apeNEXT* e quindi provare ad ideare il sistema operativo, partendo da un suo studio di fattibilità. Si è partiti dallo studio dei sistemi APE preesistenti e del loro sistema software per poi approdare sull'architettura in fase di definizione della nuova rete di calcolo. Le stesse scelte da effettuare in una fase così cruciale della definizione del software avranno poi un effetto sui dettagli hardware da dover poi implementare nelle ultime fasi di definizione.

Il risultato del lavoro è una ipotesi ragionevole e funzionante per la parte *on node* (cioè locale ad un singolo nodo della rete) di questo sistema operativo. Da essa e dalle specifiche pronte per gli aspetti rimanenti del sistema (quasi esclusivamente l'interazione con le macchine esterne) si può partire per valutare i pro e i contro di un approccio del genere per scegliere, poi, se fornire o meno i prototipi di *apeNEXT* che verranno prodotti del supporto per questo sistema operativo.

Struttura della tesi

La tesi si articola in quattro capitoli:

1. Il primo capitolo tratta della storia del progetto APE. Inoltre vengono esposte le caratteristiche dell'ultima macchina prodotta in seno ad esso: *APEmille*. In conclusione vengono poi specificate le scelte alla base dell'architettura di *apeNEXT*

2. Il secondo capitolo tratta dell'assemblatore GNU per *apeNEXT*, partendo dalle considerazioni che sono alla base della sua scelta, per poi soffermarsi sulla sua implementazione e sulle decisioni prese in merito ai formati dei file e alla compilazione di basso livello

3. Nel terzo capitolo si trova la descrizione del lavoro svolto nella ideazione dell'ipotesi di sistema operativo. In primis viene descritto il sistema operativo di *APEmille*, per poi trarne le dovute considerazioni in merito

ad una sua possibile evoluzione. Vengono inoltre descritte le modifiche hardware necessarie ad una corretta implementazione

4. In ultimo si trovano le conclusioni nelle quali vengono considerati i risultati ottenuti rigurdo allo sviluppo del progetto *apeNEXT*

- L'Appendice A riporta una parte descrittiva del sorgente dell'assemblatore e delle *APEAPI*, cioè delle routine create per accedere alle strutture dati utilizzate nei file oggetto ideati per l'assemblatore

- L'Appendice B contiene tutte le istruzioni assembler di livello utente di *apeNEXT*

- L'Appendice C riporta, infine, parti significative del codice del kernel su nodo che rappresenta il risultato dello studio di ideazione del sistema operativo

Ringraziamenti

Gradirei ringraziare il prof. Nicola Cabibbo per la fiducia mostratami e per il supporto offerto a tutto il mio lavoro. Gradirei altresì ringraziare il prof. Giuseppe De Biase per l'importante sostegno didattico senza il quale questa tesi non sarebbe mai stata completata.

Gradirei inoltre ringraziare tutti i ragazzi del laboratorio APE della sezione di Roma 1 dell'INFN; in particolar modo il dott. Davide Rossetti, il dott.

Alessandro Lonardo ed il dott. Piero Vicini, per la capacità che hanno avuto di condurmi attraverso questi mesi di studio.
Un ringraziamento particolare va al dott. Sandro Tassa per il suo importante aiuto in tutto il lavoro svolto.

Capitolo 1

Il progetto APE

1.1 Il Progetto Ape

Il progetto *APE* (*Array Processor Experiment*) nacque alle fine del 1984. Alcuni fisici dell'INFN, ricercatori nel campo della *Lattice Gauge Theory* (o LGT), videro la possibilità di costruire un elaboratore elettronico con capacità computazionale tale da renderlo fino a dieci volte più veloce delle macchine vettoriali di allora. L'introduzione sul mercato, poi, dei primi processori dedicati al calcolo in virgola mobile rese più semplice la costruzione da zero di una macchina dalle capacità richieste.

Il primo incontro dei potenziali membri del gruppo APE si tenne presso il CERN nell'inverno 1984, e già pochi mesi dopo le varie componenti del sistema erano in fase di test. L'INFN approvò nel marzo del 1985 ufficialmente il progetto. Nel 1986 era pronta la prima macchina con quattro nodi di calcolo (*APetto*), mentre nell'estate 1987 si toccarono le velocità del GFlops con la macchina successiva.

Fino a tutto il 1991 l'APE era considerata una macchina estremamente preziosa ed utile in un panorama altrimenti povero di simili strumenti, ma, già da alcuni mesi, le necessità sempre maggiori e la forte voglia di superare i propri stessi risultati, portarono alla creazione del secondo progetto in seno all'APE: $APE100$. La nuova macchina disponeva di un nodo di cacolo interamente costruito "in casa" con il vantaggio di essere più coerente con le richieste di prestazioni e sincronizzazioni dell'intero sistema. Dal 1992 ad oggi le macchine APE100 sono state largamente utilizzate nei centri di calcolo di tutta Europa, favorendo notevoli sviluppi e ricerche in numerosi campi. Il principale è quello della QCD (CromoDinamica Quantistica), cioè la teoria che descrive le interazioni forti alla base dell'aggregazione nucleare e sub-nucleare dela materia. Altri campi in cui le macchine APE sono state largamente utilizzate sono, ad esempio, la fluidodinamica e la meteorologia.

L'ultima macchina della famiglia APE attualmente in produzione è *APE-mille* con prestazioni teoriche che raggiungono il TFlops. I primi prototipi della macchina sono stati assemblati nell'estate del 1999, e la produzione in serie è già stata avviata, vista la notevole mole di richieste di queste macchine da tutto il mondo.

Raggiungere i 10 TFlops sembra ora essere l'obiettivo per la futura macchina della famiglia APE: *apeNEXT*. Essa si contraddistingue per aver aumentato la frequenza di clock dei processori matematici di un fattore 4 (circa) e per le numerose modifiche architetturali rispetto agli elaboratori precedenti. Il progetto attualmente si avvale del prezioso supporto di altri organi accademici e di ricerca europei, come il DESY di Zeuthen, Germania. La progettazione della macchina è in fase di completamento ed il test di alcune sue componenti attualmente è già cominciato.

1.2 *APEmille*

Per comprendere l'attuale situazione nella famiglia delle macchine APE è necessario analizzare l'ultima macchina attualmente in fase di distribuzione [1, 2]. *APEmille* ha segnato infatti l'apertura del progetto ad importanti partnership con enti di ricerca di tutta Europa, ed il risultato è una macchina che si sta imponendo in tutto il mondo per le proprie caratteristiche. La progettazione di *apeNEXT* deve quindi necessariamente passare per una analisi critica del lavoro fatto con *APEmille*.

1.2.1 Architettura

APEmille è un elaboratore parallelo di tipo *SIMD* (Single Instruction-Multiple Data) [3]. Questo vuol dire che tutti i processori che compongono la macchina (i *nodi*) eseguono lo stesso programma, ma lavorano su insiemi di dati differenti. Quindi ogni nodo possiede una propria memoria ad accesso casuale.

La topologia è quella di un reticolo tridimensionale *toroidizzato*, cioè con i nodi che si trovano ai limiti della macchina chiusi con quelli a loro opposti, e questo per tutte le direzioni. L'indirizzamento permette oltre all'accesso alla memoria interna ad un nodo anche a quella dei nodi primi vicini.

I processori dei nodi di calcolo sono di tipo *VLIW* (Very Long Instruction Word). Questo vuol dire che la *parola di istruzione* compresa dai processori è abbastanza lunga da contenere le stringhe di comando per ognuna delle sotto-unità dei processori, evitando quasi completamente la fase di *decode* necessaria in altre architetture (vedere figura 1.1).

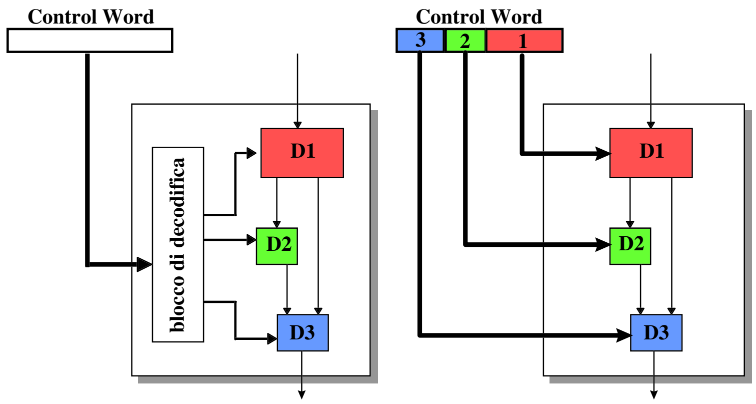

Figura 1.1: Very Long Instruction Word

I processori sono in grado di effettuare calcoli interi a 32 bit, calcoli in virgola mobile a 32 e 64 bit e calcoli su coppie da 32 bit, sia come dati di tipo complesso sia come dati vettoriali. Tutti i dispositivi della macchina forniscono un elevato grado di *canalizzazione* (pipelining) fornendo una potenza di calcolo di 528 MFlops (milioni di istruzioni in virgola mobile al secondo) con un clock di 66 MHz. La macchina presenta due moduli fondamentali: uno sincrono (*kernel*), che si occupa dell'esecuzione delle istruzioni, ed uno asincorno (*shell*), il cui compito è quello di comunicare con l'interfaccia verso l'esterno. Infatti la macchina è controllata da più *host* (personal computer con architettura Linux) su cui risiede il sistema operativo distribuito di *APEmille*. Se il controllo della macchina è affidato al kernel si parla di *run-mode*, altrimenti di *system-mode*.

Ogni elemento del multiprocessore è costituito da tre differenti unità che costituiscono i centri di calcolo e controllo della macchina: il processore di calcolo *J1000* (predisposto ai soli calcoli aritmetici), il controllore *T1000* (che controlla il flusso del programma ed invia le parole di istruzione e gli indirizzi agli J1000 che controlla) ed il processore di comunicazione *C1000* (che si occupa dei trasferimenti di dati tra nodi adiacenti).

La macchina ha come elemento fondamentale la *Processing Board* (PB) su cui risiedono 8 J1000 (disposti a cubo) con le rispetive memorie dati, un T1000 con la propria memoria, ed un C1000, come mostrato in figura 1.5.

Quattro PB disposte in una singola direzione formano una *unit*; quattro unit formano un *crate*; due crate formano un *armadio*; otto armadi formano un sistema *APEmille* completo ($32\times8\times8 = 2048$ nodi). Le linee delle dimensioni Y e Z sono limitate ad 8 nodi dal *backplane*: esso rappresenta il piano su cui corrono le linee di comunicazione (vedi figura 1.8). Grazie alla libertà della dimensione X, però, la macchina risulta scalabile, potendo presentare numerose

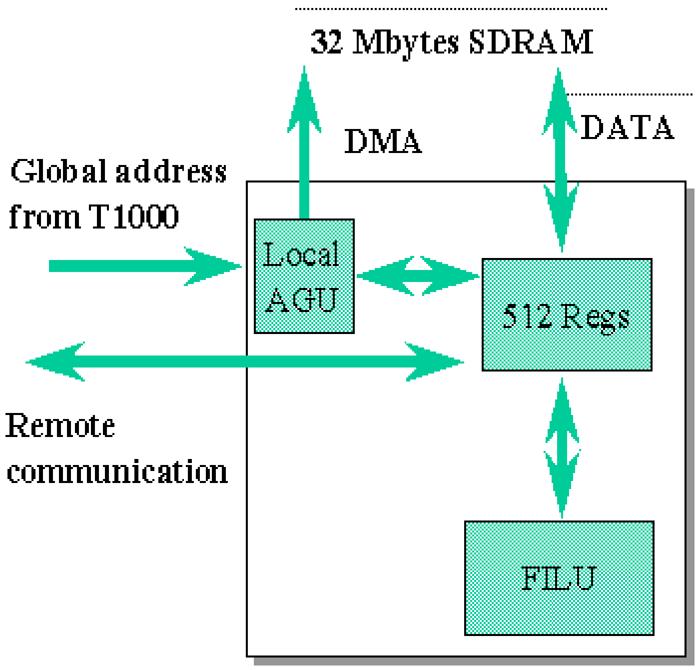

Figura 1.2: J1000

configurazioni fino al massimo rappresentato dal sistema ad otto armadi (32 nodi lungo X).

Un sistema completo *APEmille* ha una potenza di picco di 1 TFlops (mille miliardi di istruzioni in virgola mobile al secondo) con un clock di 66 MHz.

1.2.2 Software

La scrittura di software per *APEmille* può avvenire (allo stato attuale del progetto) con l'utilizzo di tre linguaggi [4]:

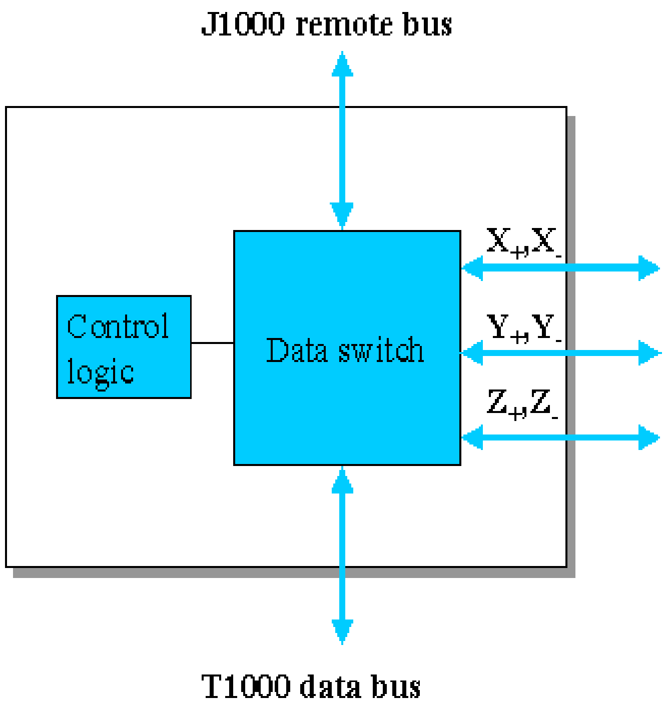

Figura 1.3: C1000

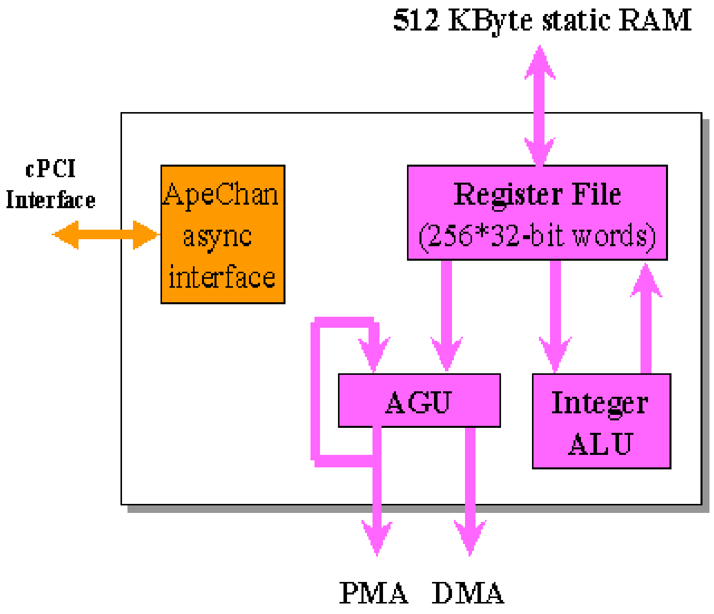

Figura 1.4: T1000

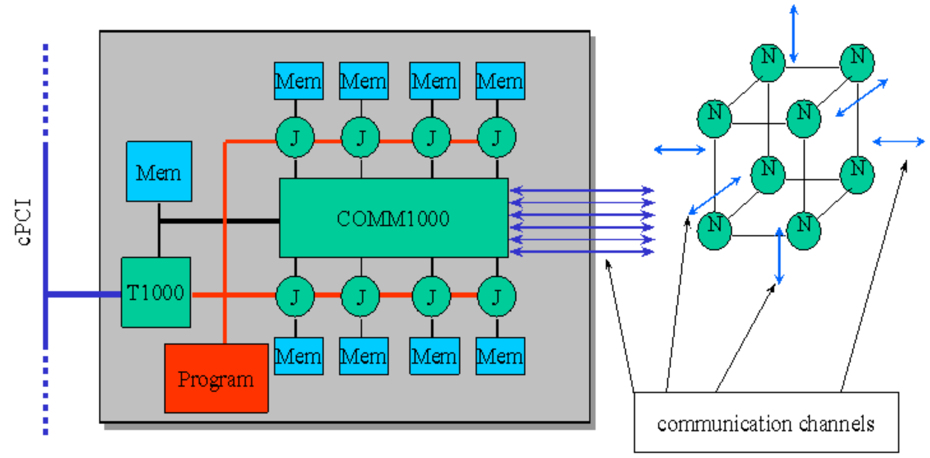

Figura 1.5: Processing Board di *APEmille*

- *TAOmille*: un linguaggio di alto livello *Fortran-like* che rappresenta l'ultima versione ideata per *APEmille* di TAO, utilizzato nel mondo APE dal modello APE100 in poi, per la creazione di applicazioni utente.

- *C*: esiste una versione preliminare del compilatore *GNU-C* per *APEmille* il quale supporta alcune estensioni al linguaggio per incorporare importanti costrutti della programmazione SIMD, come le condizioni globali.

- *Jasm*: è il linguaggio assembly ed è, quindi, il risultato delle compilazioni in altri linguaggi, ma viene anche utilizzato per ottimizzare *manualmente* il codice prodotto o per scrivere routine chiave ad alte prestazioni.

I file *.jasm* prodotti dalla scrittura in uno dei linguaggi sopra citati vengono letti dal modulo *JASM*, l'assemblatore, che produce dei file oggetto in formato

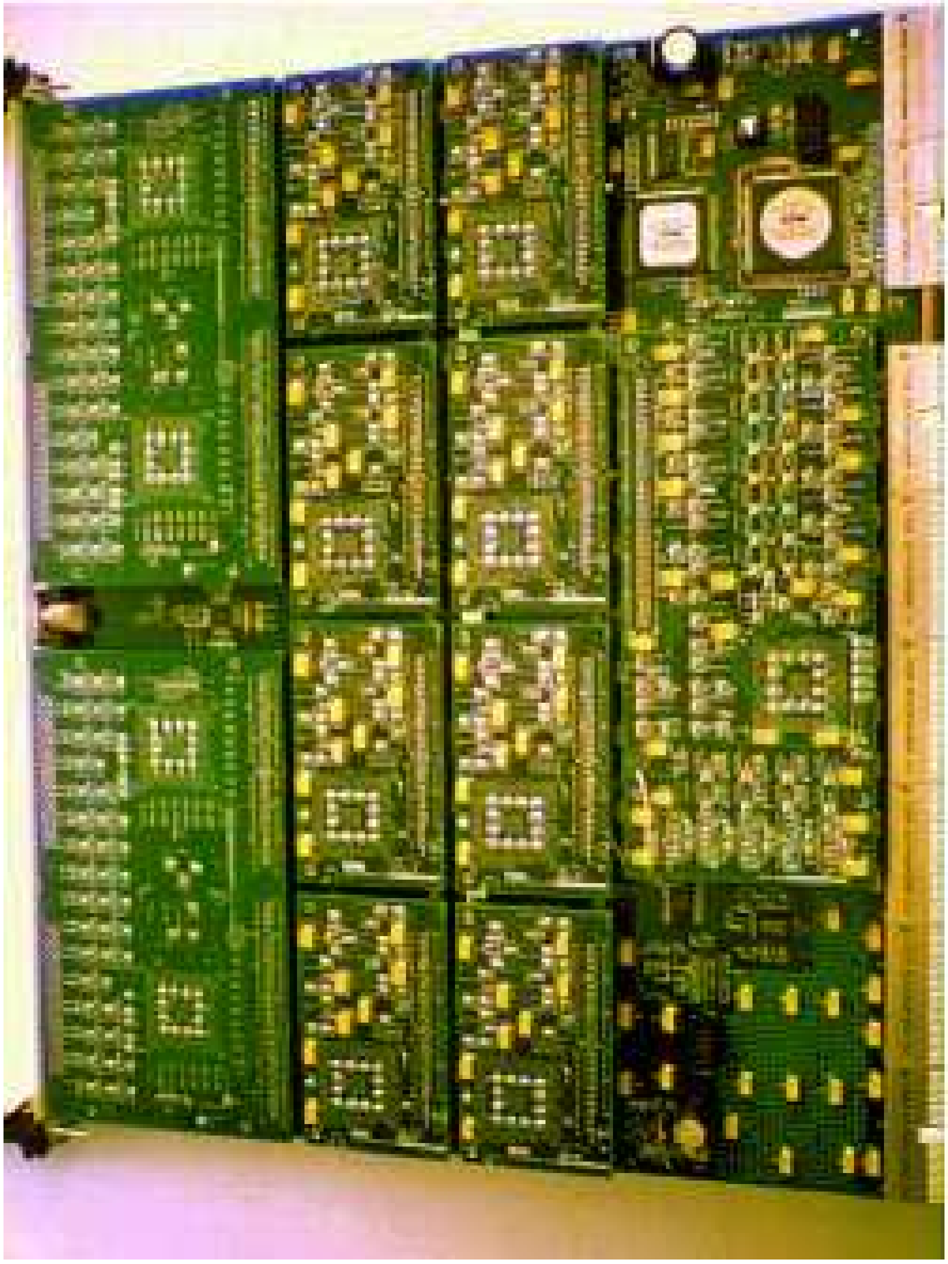

Figura 1.6: una fotografia di una PB completa

Figura 1.7: il sistema *APEmille* dei laboratori di Roma 1 dell'INFN

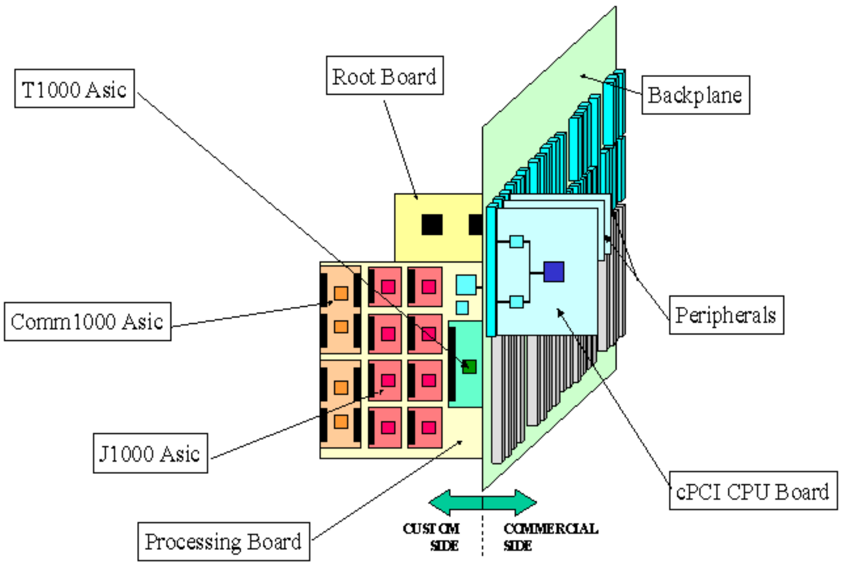

Figura 1.8: una Processing Board connessa al backplane

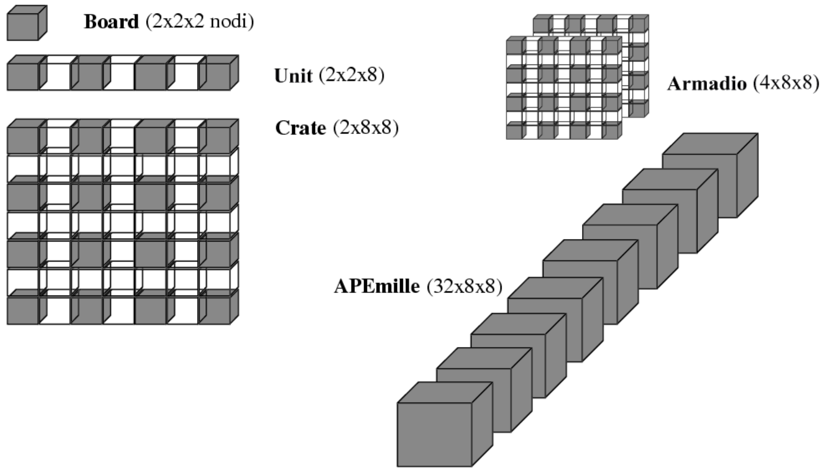

Figura 1.9: configurazioni di *APEmille*

basm. Il linker accetta in ingresso più file oggetto creandone uno non-rilocabile.

I file così prodotti, però, non sono composti dalle parole VLIW comprensibili dalla macchina. I file basm, infatti, sono una rappresentazione *espansa* delle istruzioni da eseguire e degli operatori che intervengono in esse, e devono venire in qualche modo raggruppati per generare delle parole di istruzione.

È lo *shaker* che si occupa di leggere i file basm e di generare da essi le parole VLIW equivalenti [5]. Esso effettua una ottimizzazione *hardware-dependent* legata alla architettura di *APEmille*, cercando di minimizzare il numero di parole generate. Inoltre esso tende a mantenere tutte le risorse occupate contemporaneamente, sfruttando al massimo il meccanismo di canalizzazione (*pipelining*) presente.

L'ottimizzazione si basa su una rilocazione delle istruzioni assembly compatibile con i loro vincoli di parentela, cioè con l'ordine semantico necessario a mantenere coerente il flusso delle istruzioni. La generazione dei vincoli di parentela parte dall'osservazione delle dipendenze delle varie istruzioni riguardo alle risorse a cui esse accedono. Ad esempio non è possibile effettuare un'istruzione di scrittura su una data risorsa prima che non siano state effettuate tutte le precedenti istruzioni di lettura su quella stessa risorsa.

Una volta stabiliti tali vincoli lo *shaker* effettua i due passi di *shake-up* e *shake-down*. Il primo sposta verso l'alto le stringhe VLIW che corrispondono alle operazioni da effettuare, compatibilmente con i vincoli presenti. Il secondo, contrariamente, le sposta verso il basso, compattando ulteriormente il codice.

(vedere fig 1.10).

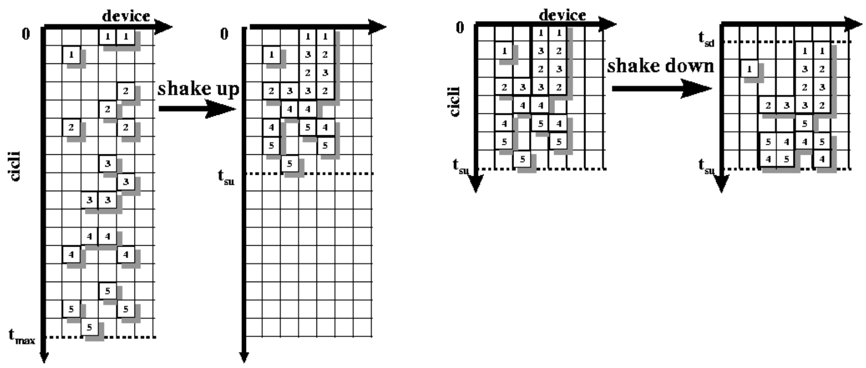

Figura 1.10: fasi dello shaker

Lo shaker si occupa anche di allocare fisicamente i *registri virtuali.* Tutti i registri utilizzati nella programmazione di *APEmille* sono infatti da considerarsi virtuali (se non contrariamente specificato), cioè non aderenti all'effettiva numerazione all'interno della macchina. Questo meccanismo permette al codice jasm di utilizzare un numero ipoteticamente infinito di registri senza occuparsi della loro effettiva rilocazione fisica. Questa rilocazione viene invece effettuata dallo shaker che assegna ad ogni registro virtuale un registro fisico secondo un rapporto *molti ad uno.* Per far ciò si procede constatando le dipendenze dei dati ed allocando un registro virtuale ad un registro fisico quando esso non viene più utilizzato.

1.3 *apeNEXT*

La necessità di creare una nuova macchina risiede nei bisogni sempre crescenti, in ambito scientifico, di innovazione e crescita. I limiti hardware di *APEmille* non permettono lo sfruttamento della macchina per sistemi di fisca quantistica quali ora ci si appresta a studiare [6]: si prevede, infatti, che per il 2002/2003 le simulazioni di *Lattice QCD* si concentreranno su sistemi completi di dimensioni medie e grandi e su approssimazioni di sistemi di grandezze ancora maggiori.

Tutto ciò impone determinate caratteristiche di efficienza che la macchina deve rispettare. In particolar modo le simulazioni previste delineano determinate richieste in fatto di prestazioni [7]:

Potenza di picco:	$\sim$ 10 TFlops
Memoria totale:	$\sim$ 1 TByte
Banda di trasferimento con l'esterno:	$\sim$ 2 GByte/sec

Esaminando le possibilità fisiche/elettroniche attualmente a disposizione le caratteristiche del singolo nodo risultano le seguenti:

Frequenza di clock:	200 MHz
Potenza computazionale:	1.6 GFlops
Memoria:	512 MBytes

Si evidenzia dunque la necessità di avere alcune migliaia di nodi per raggiungere le prestazioni richieste.

1.3.1 Il nodo di calcolo

A differenza di *APEmille* in *apeNEXT* si vuole creare un processore unico che comprenda tutte le funzionalità di T1000, J1000 e C1000. Questo per sfruttare al meglio le innovazioni introdotte nel campo della VLSI (Very Large Scale Integration) per le quali ora risulta più facile inserire un maggior numero di unità all'interno di un solo chip piuttosto che gestire la comunicazione tra più moduli. Il processore conterrà le seguenti unità:

1. *FILU* (Floating - Integer - Logic Unit)
2. *AGU* (Address Generation Unit)
3. *SU* (Sequence Unit)
4. *RF* (Register File)
5. *NI* (Network interface)
6. *LUT* (Lookup Table)

La FILU permette l'utilizzo di numeri in virgola mobile a 64 bit, è canalizzata e permette il calcolo di una *normal* $(A \times B + C)$ tra numeri complessi in 1 ciclo di clock (a regime); inoltre fornisce il supporto per i calcoli interi e logici a 64 bit. La AGU gestisce le istruzioni di generazione degli indirizzi. La SU controlla il *Program Counter*. Il RF è composto da 512 registri ognuno da 2 word (2×64 bit). Questo permette di indirizzare ogni registro secondo il banco alto (*Hi*) o basso (*Low*) oppure come vettore (*Vector*) abilitando ambuedue i banchi. La NI gestisce la comunicazione con i nodi vicini. La LUT fornisce alcune approssimazioni per complicati calcoli in virgola mobile, come $\frac{1}{x}$, $\sqrt{x}$, x^y.

Per uno schema del nodo di calcolo così come attualmente definito vedere fig. 1.11.

Seguendo il paradigma VLIW ognuna di queste unità è comandata da uno o più particolari campi nella parola di istruzione che descrivono i comandi da eseguire e gli operandi da utilizzare.

Le unità qui descritte contengono tutte un meccanismo di canalizzazione e hanno la capacità di lavorare in parallelo tra loro.

Inoltre nel processore è presente una cache per le istruzioni in grado di caricare intere sezioni di codice attraverso un meccanismo comandato da istruzioni macchina

Ogni nodo, inoltre, contiene la propria memoria ad accesso casuale, suddivisa logicamente tra una sezione per il codice (Program Memory) ed una per i dati (Data Memory).

1.3.2 Le comunicazioni

APEmille aveva un meccanismo di interconnessione tra nodi ad alta sincronia. Essendo programmati in maniera SIMD i nodi potevano prevedere in maniera esatta da quale direzione ed in quale momento avrebbero ricevuto il dato

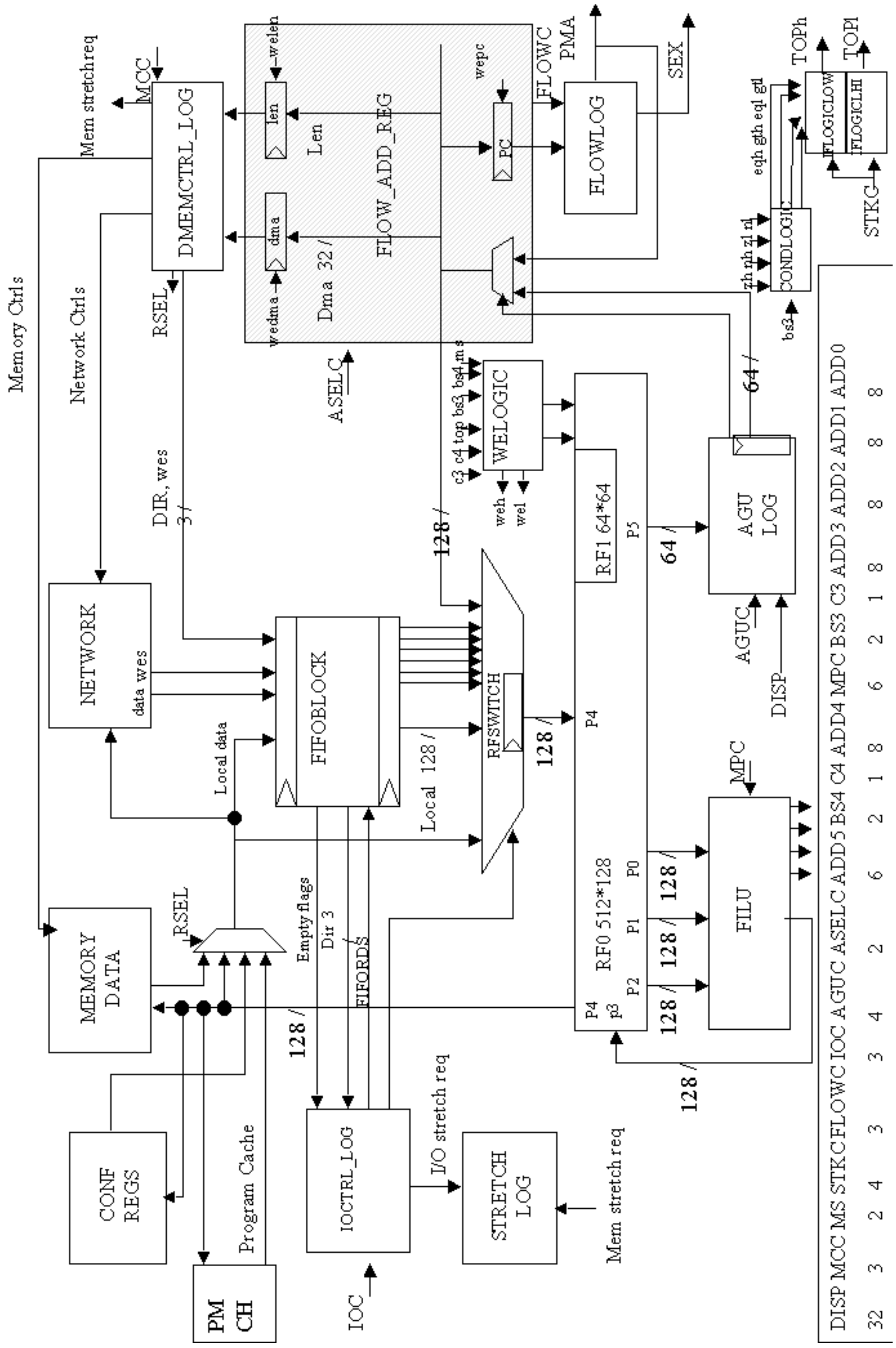

Figura 1.11: schema del nodo di calcolo di *apeNEXT*

atteso. Portando il clock da 66 a 200 MHz, tra le altre cose, si impone però un meccanismo di comunicazione più lasco che permetta piccoli ritardi nelle trasmissioni. Per questo motivo sono state introdotte delle code (*FIFO*), una per ogni direzione, che contengano i dati ricevuti dai nodi vicini e che lavorino quasi indipendentemente dalle altre unità. In questo modo è possibile inserire localmente il controllo di sincronizzazione piuttosto che fornirne uno globale (demandato a C1000 in *APEmille*), il che, alle frequenze previste, potrebbe generare più di un problema.

1.3.3 La topologia

Per quel che concerne la topologia dei nodi si è scelto di continuare con la tradizione di tutte le macchine APE, con i nodi disposti a reticolo cubico *toroidizzato*. La macchina verrà probabilmente costruita con configurazione ad armadio, come *APEmille*, con delle schede (le Processing Board) su cui verranno montati 8 o 16 nodi di calcolo ciascuna. Un host costruito da macchine su piattarfoma Linux controllerà un certo numero di queste schede per eseguire la parte *remota* del sistema operativo.

1.3.4 Software

Il meccanismo di programmazone della macchina previsto non si discosta molto da quello descritto per *APEmille*. La necessità di uno shaker che abbia caratteristiche simili al precedente risulta un retaggio necessario del gruppo APE ed un punto di forza di tutta la catena di compilazione. Inoltre il compilatore GNU-C in fase di ultimazione per *APEmille* verrà con ogni probabilità portato anche sulla nuova macchina, sfruttando l'esperienza fatta precedentemente. Per quel che riguarda il linguaggio TAO esso accompagna tutte le macchine APE da tre generazioni ed il compilatore per esso sarà presente anche in *apeNEXT*.

Una inversione di tendenza nella catena di compilazione si ha probabilmente con l'assemblatore. Nelle macchine precedenti esso era scritto completamente da zero e generava codice oggetto non standard. Ovviamente questa scelta porta anche ad una scrittura ad hoc del linker.

In *apeNEXT*, invece, si è pensato di adottare un assemblatore preesistente di facile adattamento, come lo GNU Assembler, fornendolo del supporto per un formato di file oggetto di largo utilizzo. Questo semplificherà la scrittura del linker e di tutti quegli strumenti di manipolazione dei file oggetto, oltre a rendere più semplice la scrittura di quelle applicazioni, come lo shaker, che devono ricevere in ingresso proprio questo tipo di file.

La descrizione del lavoro svolto per la creazione di questo assemblatore è oggetto del prossimo capitolo.

Capitolo 2

NAS - Next Assembler

2.1 Considerazioni teoriche sugli assemblatori

L'*assemblatore* compone uno degli ultimi anelli della *catena di compilazione*. Vuol dire che, in un insieme di strumenti atti a generare codice macchina partendo da linguaggi di programmazione di alto livello, l'assemblatore è molto vicino alla generazione di parole di istruzione comprensibili dall'elaboratore.

Esso è un *traduttore*, cioè accetta in input un testo scritto in un linguaggio e ne genera in output uno scritto in un linguaggio differente. Il linguaggio che l'assemblatore è in grado di leggere si chiama *assembly* (o semplicemente linguaggio assembler), e varia da macchina a macchina.

Il codice assembly può essere scritto da un programmatore o, più spesso, è l'output di un altro degli strumenti che compongono la catena di compilazione come i compilatori di linguaggi di alto livello (es. C o TAO).

2.1.1 Linguaggi formali e grammatiche

Un *linguaggio* L [8] viene definito su un insieme di simboli chiamato *alfabeto* A ed è un sottoinsieme di tutte le possibili *stringhe* A^* definite su A:

$$L \subseteq A^*$$

Un linguaggio si può definire con una notazione insiemistica, ad sempio indicando semplicemente $L = \{a, aa, ab, bb\}$ (un linguaggio composto da quattro stringhe su un alfabeto di sue simboli). Il modo più conveniente per indicare un linguaggio, però, risulta essere la definizione di una *grammatica* che lo genera.
Una *grammatica* G è una tupla

$$G =< T, N, P, S >$$

con:

T : insieme dei simboli terminali

N : insieme dei simboli non terminali

P : insieme delle produzioni

S : simbolo di partenza

Un simbolo terminale è composto da stringhe dell'alfabeto A e non può essere ulteriormente sostituito.
Un simbolo non terminale può essere sostituito da uno o più simboli terminali o non terminali.
Una produzione definisce una regola per sostituire un simbolo non terminale con uno o più simboli.
Il simbolo di partenza definisce quale non terminale è in grado di produrre tutte le parole del linguaggio con un insieme finito o infinito di sostituzioni.

Ad esempio il lingaggio infinito $L = \{a, aa, aaa, aaaa, aaaaa, ...\}$ può essere definito da un solo simbolo non terminale X e dalle seguenti produzioni:

$$X ::= aX \qquad X ::= a$$

ovviamente il simbolo X sarà anche quello di partenza per la grammatica.

La classificazione di Chomsky

Partendo dalle regole di definizione delle produzioni nelle grammatiche, Chomsky nel 1957 ha introdotto una classificazione:

grammatiche e linguaggi di tipo 0 : non vi sono restrizioni sul formato delle produzioni che possono prsentare sia nella parte destra che nella parte sinistra un numero arbitrario di simboli terminali o non terminali:

$$\alpha ::= \beta$$

$\alpha \in (N \cup T)^*$

$\beta \in (N \cup T)^*$

grammatiche e linguaggi di tipo 1 (*contesuali*): hanno le produzioni nella forma:

$$\gamma A \delta ::= \gamma \beta \delta$$

$\alpha, \beta, \delta \in (N \cup T)^*$

$A \in N$

grammatiche e linguaggi di tipo 2 (*liberi dal contesto*): sono più restrittivi perché le produzioni sono solo del tipo:

$$A ::= \alpha$$

$\alpha \in (N \cup T)^*$

$A \in N$

grammatiche e linguaggi di tipo 3 (*regolari*): sono i più restrittivi e le loro produzioni sono del tipo:

$$A ::= aB \qquad A ::= a$$

$a \in T$

$A, B \in N$

2.1.2 Analisi testuale

Tornando agli assemblatori essi, in quanto traduttori, devono eseguire alcune analisi testuali sul codice sorgente. In generale l'analisi di un testo da parte di strumenti simili si può dividere nelle seguenti sezioni:

1. **analisi lessicale**
2. **analisi sintattica**
3. **generazione di codice intermedio**
4. **ottimizzazione**
5. **generazione di codice oggetto**

L'**analisi lessicale** si occupa di suddividere il testo in ingresso nelle sue componenti fondamentali. È ad esempio in questo stadio che si identificano le istruzioni e i valori costanti, o che si eliminano spazi vuoti e commenti.
L'**analisi sintattica** verifica invece la correttezza semantica del codice, individuando il significato del testo fornito secondo le regole del linguaggio a cui appartiene. Ad esempio è durante questa analisi che si può verificare che ad una data istruzione siano dati un particolare numero e tipo di operandi o che un simbolo sia dichiarato prima di venire utilizzato.
La **generazione di codice intermedio** parte dall'interpretazione data al codice dalle analisi precedenti per definire le azioni corripondenti in termini di un codice di livello più basso. Esso, però, non è ancora quello risultato della

traduzione, essendo più astratto di quest'ultimo, e serve a permettere una manipolazione più efficace del codice tradotto per una migliore ottimizzazione.
Di seguito, infatti, viene l'**ottimizzazione** del codice, che segue regole ben precise per minimizzare il numero di istruzioni o il tempo di esecuzione del programma.
Per ultimo si ha la **generazione di codice oggetto** che avviene con un lavoro di traduzione dal codice intermedio in quello destinazione, come il linguaggio macchina o l'assembly.

In un assemblatore, solitamente, si hanno solo le fasi 1, 2 e 5, non essendo necessaria la generazione di un codice intermedio tra l'assembly ed il linguaggio macchina. Se un assemblatore è in grado di ottimizzare il codice lo farà direttamente sul codice macchina risultato, senza richiedere la fase 3.

Le fasi 1 e 2 sono strettamente legate alla sintassi del linguaggio sorgente e quindi richiedono una rappresentazione di tale linguaggio per poter essere effettuate sul codice fornito. Grazie alla teoria dei linguaggi formali è possibile definire delle grammatiche per generare un *lexer* (analizzatore lessicale) ed un *parser* (analizzatore semantico).
Per il primo viene fornita una grammatica *regolare* che definisca le varie componenti sintattiche da cui è formato il sorgente.
Per il secondo la grammatica fornita è, solitamente, di tipo *libero dal contesto* poiché risulta avere maggiore espressività del precedente.

I linguaggi assembly sono considerati di *basso livello*; vuol dire che sono strettamente legati all'hardware della macchina per cui sono progettati. Ciò li differenzia dai linguaggi di *alto livello* che godono di una astrazione tale da permettere loro di poter essere adattati a macchine profondamente diverse, semplicemente fornendo un compilatore per ognuna delle architetture su cui si vuole adattarli.

È per questo motivo che una rappresentazione intermedia di codice assembly non ha molto senso, essendo quest'ultimo già una semplice rappresentazione simbolica *uno-a-uno* delle istruzioni macchina. Ciò vuol dire che la generazione di codice oggetto partendo da un assembly si tramuta semplicemente in una *espansione* dei comandi in una, o più, parole di istruzione per la macchina destinazione.

Per gli stessi motivi le analisi lessicali e sintattiche di un assembly sono necessariamente ridotte al minimo, essendo il linguaggio per sua natura estremamente rigido e composto da pochi e ben delineati elementi.

2.2 Un Assembler per *apeNEXT*

Nelle fasi iniziali del progetto *apeNEXT* si è subito evidenziata la necessità di avere un assemblatore che permettesse di generare del codice oggetto da provare sul *simulatore funzionale* (SF) della nuova macchina. Seguendo, poi, il procedimento adottato già in *APEmille* tale codice avrebbe potuto essere successivamente ottimizzato da uno shaker per generare delle parole VLIW pronte ad essere simulate sulla descrizione hardware *VHDL*. Differentemente da quanto accaduto per le evoluzioni precedenti dell'APE si è voluto affrontare il problema partendo da un assemblatore "standard" per farne un adattamento (*porting*) per la nuova macchina. Questo avrebbe permesso, tra le altre cose, di avere un formato universale per i file oggetto generati, e quindi la possibilità di utilizzare una serie di strumenti già disponibili e ben collaudati. Nel caso specifico si è scelto di porre l'accento sulla politica di supportare gli strumenti GNU per le future versioni di APE. Un compilatore GNU C/C++, infatti, è già in avanzata fase di implementazione, e ciò ha dimostrato il valore di questi strumenti e l'importanza di un loro porting. L'assemblatore GNU (*GAS*) [9] è già uno strumento molto utilizzato in numerose implementazioni del sistema operativo Unix (e derivati) ed esiste per una grande quantità di processori e di architetture. Eseguirne un porting per la futura macchina implica una uniformità nella catena di compilazione che ne rende più semplice ed efficace l'implementazione, oltre a rappresentare una valida scelta anche per una continuità con le future generazioni degli elaboratori APE.

Lo GNU assembler è uno strumento distribuito, ovviamente, sotto la GNU General Public Licence (GPL), quindi è di tipo Open Source. Questo vuole dire che i suoi sorgenti sono liberamente distribuibili e modificabili a patto di mantenere la stessa licenza anche nei prodotti derivati. Esso è contenuto in un insieme di strumenti per la gestione di archivi di file oggetto chiamata *Binutils* [10]. Essi sono totalmente scritti in linguaggio C [11] e tutti distribuiti con la stessa licenza GPL. Tra gli strumenti contenuti nelle Binutils ce ne sono alcuni il cui porting su *apeNEXT* risulta conveniente per costituire lo scheletro necessario ad una uniformità nella catena di assemblamento:

gas & gasp assemblatore e relativo preprocessore

ld linker

objdump disassemblatore

Altri strumenti utili (come *readelf*) sono contenuti nelle Binutils e si cercherà di permettere il loro utilizzo nella nuova macchina.

All'interno delle Binutils esiste una libreria largamente utilizzata dagli altri strumenti (e anche da altre applicazioni esterne al pacchetto) chiamata *BFD* (Binary File Descriptor) [12, 13]. Essa ha il compito di astrarre dalle particolari definizioni dei differenti tipi di file oggetto supportati, fornendo un insieme di routine generiche per l'accesso in creazione/modifica/lettura per una grande varietà di archivi. La versione utilizzata delle Binutils è la 2.9, in cui tutti gli strumenti contenuti supportano le BFD, benché sia possibile effettuare la scrittura di versioni di essi senza questo supporto. Ovviamente scrivere il supporto BFD per il proprio tipo di file oggetto offre il vantaggio di rendere uniforme e più indolore il porting dei vari strumenti.

La scrittura delle binutils per *apeNEXT* deve, quindi, passare per la preparazione del supporto BFD per i file oggetto da generare. Come formato per i file oggetto è stato scelto l'*ELF* (Executable and Linking Format). Esso risulta largamente utilizzato in molte architetture e si presta ad essere un formato unico sia per i file contententi codice oggetto non eseguibile che per i file eseguibili veri e propri.

Nelle fasi iniziali del progetto si è quindi utilizzato un supporto generico offerto dalle BFD per il formato ELF, focalizzando il lavoro sul porting del GAS per creare il **NAS** (Next ASsembler). Questo è necessario per poter scrivere l'assemblatore senza curarsi del formato di output, il quale nelle fasi preliminari del progetto non risulta ancora formalmente definito. Solo successivamente verranno apportate le opportune modifiche alle routine della libreria BFD avendo anche dei file oggetto di tipo generico su cui poter effettuare delle prove.

2.3 Porting del GAS

Il porting del GAS prevede la modifica di alcuni file di configurazione più l'aggiunta di un certo numero di file (alcuni obbligatori, altri accessori) con l'effettiva descrizione della macchina. Ecco la lista dei file creati:

gas/config/tc-apenext.h header file del codice *Target-CPU*

gas/config/tc-apenext.c file sorgente del codice *Target-CPU*

include/opcodes/apenext.h descrizione delle istruzioni macchina

include/apenof.h tipi di dato rilevanti per la lettura/scrittura di file oggetto

Di questi solo i primi due sono effettivamente necessari poiché richiesti dal meccanismo di autocompilazione (*make*) configurato per il GAS; essi contengono il codice Target-CPU, cioè un insieme di direttive del preprocessore e di definizione di variabili e funzioni, predeterminati. La corretta definizione di queste caratteristiche (che individuano l'architettura obiettivo del porting) permette all'assemblatore, predisposto a lavorare su un vasto insieme di architetture, di individuare i giusti passi da seguire. Tra gli altri i parametri specificati riguardano il formato dei file oggetto (**ELF**), i caratteri che individuano i commenti (**!!**), l'utilizzo o meno delle BFD, etc. (queste caratteristiche si possono riscontrare nel codice riportato nell'appendice A)

Una volta stabiliti i parametri da indicare in questi file di configurazione per una corretta definizione delle caratteristiche della macchina è risultato necessario passare alla definizione delle routine che gestiscono la fase di assemblamento.

Ovviamente GAS fornisce supporto al programmatore che si occupa della scrittura dell'assemblatore. In particolar modo possiamo indicare nei seguenti punti le caratteristiche del GAS che vengono ereditate al momento di scriverne un porting verso una nuova architettura:

- un insieme di strutture dati liberamente utilizzabili (come le tabelle *hash*) più tutto un insieme di librerie di pubblico dominio (le **libiberty**) che forniscono routine per la gestione di buffer di caratteri, pile, code, etc.

- supporto alla sintassi di definizione di nuovi tipi di **sezioni** di codice oggetto e per la scrittura in quelle considerate di *default* (*.text* per il codice eseguibile, *.data* per i dati inizializzati, etc.)

- gestione automatica delle **label**, cioè dell'indicazione di simboli per l'identificazione di particolari sezioni di codice o per particolari dati (inizializzati o meno) residenti in memoria

- supporto immediato per tutte le **pseudo-istruzioni** predefinite nella sintassi del GAS, come quelle per la scrittura di dati inizializzati in memoria (*.integer*, *.real*, *.string*, etc.), per l'allineamento del codice (*.align*) o per la scrittura di commenti (*.note*)

- definizione di **macro** e di sostituzione simbolica di espressioni (da aggiungere al supporto già presente nel preprocessore **NASP** - Next ASsembler Preprocessor [14])

- apertura dei file sorgente e loro *canonicizzazione* attraverso la gestione dei commenti, il passaggio a lettere minuscole, la rimozione di spazi superflui, etc.

- scrittura dei file oggetto attraverso le BFD

Vengono inoltre forniti degli strumenti per controllare con una certa accuratezza questi aspetti dell'assemblamento. Ad esempio si possono definire delle routine da richiamare quando il parser riconosce una etichetta (*tc_frob_label()*)

o alla chiusura del file sorgente (*md_cleanup()*). Nell'appendice A sarà possibile trovare la definizione di quelle routine che rientrano in questa categoria e che sono state utilizzate nel progetto, assieme ad un piccolo commento per ognuna di esse.

Quelle linee di codice sorgente che il parser di GAS non riconosce come pseudo-istruzioni o etichette vengono passate interamente al codice *machine-dependent* opportuno. Più precisamente viene invocata la funzione *md_assemble()* alla quale viene passata come parametro la stringa che rappresenta la linea di codice incontrata, già priva di commenti e spazi superflui.

Questa funzione deve quindi fornire un parser per il linguaggio macchina opportuno, ed alcuni elaboratori sono arrivati al punto di avere una sintassi talmente complicata per i loro linguaggi assembly da aver costretto i progettisti a scrivere questa parte attraverso strumenti di costruzione di analizzatori lessicali e sintattici pensati per l'ideazione di compilatori (come **Lex** e **Yacc**). Il linguaggio macchina di *apeNEXT*, definito preliminarmente in una sua forma ancora non ben delineata ereditata dai linguaggi delle macchine APE delle generazioni precedenti, non presenta una sintassi così complicata da giustificarne l'utilizzo. Quindi il parsing si è concentrato in poche routine di manipolazione di stringhe, una per ogni tipo di operando possibile (MemoryOperand [*MOP*], Register [*REG*], SetOfRegisters [*REGSET*], Length [*LEN*]). Le istruzioni sono invece stipate in una tabella hash da cui vengono estratte al momento di interpretare la prima parola presente in una riga di istruzione, assieme a tutte le informazioni riguardo a numero di operandi attesi e loro tipo. L'analisi della correttezza semantica, quindi, viene demandata alle routine di parsing dei particolari operandi: conoscendo a priori il numero di operandi attesi per istruzione ed il loro tipo si ottiene il controllo sintattico semplicemente assicurandosi che gli operandi vengano correttamente riconosciuti. Questo, ovviamente, è possibile solo grazie alla rigidità del linguaggio assembly di APE, per cui esistono pochi casi di operandi opzionali (solo LEN) e poche varianti per ogni operando. In particolar modo solo i simboli non terminali **MOP** e **REGSET** hanno una sintassi relativamente libera:

- **MOP**: ogni operando che individua un indirizzo di memoria ha questa sintassi:

  ```
  DISP[(+|-)OFFSET[([OP1],[OP2],[OP3])]
  ```

  Il significato di questa espressione indica un indirizzo di memoria di questo tipo: DISP(+|−)OFFSET + OP1 + (OP2 * OP3).

  DISP può essere un valore assoluto o un riferimento simbolico, mentre *OFFSET* può solo essere un valore intero che indica uno spiazzamento. *OP1*, *OP2* e *OP3* possono essere valori assoluti, registri o @ (accumulatore). La sintassi che regola il tipo di operando possibile è descritto attraverso la seguente grammatica regolare:

```
MOP     ::=  DISP64 | DISP64 PAR | PAR | DISP32

PAR     ::=  "(" CSO ")"

R_d     ::=  R | DISP32

R_@     ::=  R | @

CSO     ::=  R_@                     |
             R_@ "," R               |
             "," R "," R_d           |
             R   "," R "," R_d       |
             @   "," R "," DISP32
```

dove:

DISP64: `%Immediato` | `Simbolo` | `Simbolo (+|−) %Immediato`

DISP32: `%Immediato (a 32 bit)`

R: `Intero (registro)`

@: `Simbolo per l'Accumulatore`

Questa grammatica genera 20 possibili operandi, e la sua definizione nasce, ovviamente, dalle istruzioni VLIW della macchina. In particolare le micro-istruzioni della unità di generazione degli indirizzi (AGU) sono 11, il che significa che un operando di memoria solitamente corrisponde ad una sequenza di istruzioni VLIW. Ovviamente la possibilità di generare indirizzi con sequenze di istruzioni è facilitato dalla presenza dell'accumulatore, ideato proprio per questo scopo.

Questo tipo di operando è stato ispirato dal suo equivalente nella sintassi AT&T dell'assembler Intel x86, ritenuto facilmente comprensibile.

- **REGSET**: un insieme di registri viene definito nel trasferimento memoria $\leftrightarrow$ registri di più parole. Questa possibilità aumenta l'efficacia della macchina sfruttando la canalizzazione presente negli accessi alla memoria. Un set di registri può essere definito da un solo registro `[R1]`, da un insieme eterogeneo di registri `[R1,R2,R5,R8]` o da un insieme di registri contigui `[R1:NumDiRegistri]`

Questa è invece la definizione degli altri tipi di operandi possibili:

- **REG**: un registro è definito da un numero intero (in decimale, esadecimale o ottale)

- **LEN**: una lunghezza può essere solo un valore intero

Inoltre i set di registri possono avere uno specificatore di banco: **L** (Low, banco basso), **H** (Hi, banco alto), **V** (Vector, ambedue i banchi). Questi specificano quali banchi debbano essere abilitati nelle operazioni di lettura/scrittura sui registri, essendo i registri, come già indicato in (1.3.1) suddivisi in due banchi liberamente referenziabili.

Questi modificatori si applicano anche a tutte quelle istruzioni di manipolazione dei registri: ad esempio l'istruzione `ATOR.L Reg Mop` scrive il valore generato dal *Mop* nel banco basso del registro *Reg*, mentre le varianti `ATOR.H` e `ATOR.V` scrivono il valore, rispettivamente, nel banco alto o in ambedue i banchi del registro. Esempi di istruzioni che rientrano in questa categoria sono tutte quelle aritmetiche (sia in virgola mobile che intere) e quelle di test delle condizioni.

Esistono, poi, alcune pseudo-istruzioni proprie del linguaggio di *apeNEXT* così come è stato immaginato. Il GAS offre la possibilità di definire delle nuove pseudo-istruzioni semplicemente aggiungendo l'indicazione del loro nome e registrando la funzione preposta a gestirle in un array predefinito (*md_pseudo*).

La più interessante risulta essere quella per la definizione delle **Slice** di dati. Le **Slice** sono delle definizioni di sotto-spazi tridimensionali dell'insieme dei nodi della macchina per cui vengono assegnati particolari valori ai dati inizializzati. Questo vuol dire che grazie alle definizioni di slice è possibile definire dei dati inizializzati con differenti valori su differenti nodi della macchina, implementando appieno il paradigma SIMD. La pseudo-istruzione che si occupa di tutto ciò assegna ad un simbolo nella sezione dei dati inizializzati (.data) un insieme di dati e due punti nello spazio (quindi due volte tre valori interi);

questi punti definiscono univocamente il parallelepipedo di nodi che conterrà quei valori in riferimento a quel particolare simbolo. È quindi possibile definire un simbolo che fa riferimento ad un indirizzo in memoria ed un insieme di slice, ognuna delle quali contiene diversi valori per i dati inseriti. Un esempio che segue la possibilità qui descritta è il seguente:

```
.data


dato:
        .slice [0, 0, 0] [1, 1, 1]
        .real 1.0
        .slice [1, 1, 2] [2, 2, 2]
        .real -1.0
        .slice [2, 2, 3] [3, 3, 3]
        .real 2.0e3
```

Così al simbolo *dato* viene assegnato il valore reale 1.0 nei nodi compresi tra (0, 0, 0) e (1, 1, 1); il valore -1.0 in quelli compresi tra (1, 1, 2) e (2, 2, 2) e il valore 2000.0 a quelli compresi tra (2, 2, 3) e (3, 3, 3).

2.4 Generazione di codice oggetto

Come già accennato i file oggetto generati saranno scritti in formato ELF. Resta ora da specificare il contenuto di questi file, cioè il formato che descrive il risultato dell'assemblamento del codice.

Abbiamo già visto (1.3.4) che il risultato dell'assemblamento subirà un processo di ottimizzazione da parte dello shaker per generare parole VLIW comprensibili al processore. Inoltre esso potrà venir letto dal simulatore funzionale (SF) della macchina per venire eseguito in un ambiente virtuale senza venire precedentemente ottimizzato. Quindi il codice oggetto deve rispecchiare i comandi nativi del processore indicando dei *pattern* di codice VLIW da abilitare che descrivano istruzioni e operandi di ciò che deve venire eseguito.

Si è scelto di fornire un insieme di istruzioni di livello più basso che rispecchino quelle possibili nei vari campi della parola di istruzione. Tali istruzioni, assieme alle definizioni degli operandi che le accompagnano, formeranno questi pattern, indicando quali campi della parola VLIW debbano essere riempiti e con quali valori.

Si è quindi creato un linguaggio partendo da queste nuove istruzioni di basso livello, chiamato **Micro-Assembly**, dalla sintassi estremamente rigida e semplice. Esso rappresenta l'input per lo shaker ed ha il pregio di essere comunque comprensibile da un umano aprendo la strada a possibili utilizzi per debugging e/o ottimizzazioni. Il linguaggio compreso dall'assemblatore è stato quindi chiamato **Assembly Utente**, poiché dovrà essere quello generato dalla programmazione di basso livello e dai compilatori. A dispetto di quanto si possa credere, l'assemblamento del codice utente non è un semplice lavoro di "macro-espansione", e quindi non si può pensare di eseguirlo agilmente con un meccanismo di pre-processing. Questo perché le regole di traduzione macro→micro sono molto dipendenti dal tipo di operando previsto (basti pensare ai RegSet) che spesso devono generare differenti quantità di micro-istruzioni. Inoltre la gestione di simboli, slice, macro-definizioni oltre all'analisi sintattica e semantica è bene che sia fatta a livello di assembly utente. Un esempio di traduzione User→Micro Assembly è il seguente:

USER	`RTOM`	`$1(200, 450, 451)`	`300`
MICRO	`AGU_RXR`	`451`	`450`
	`AGU_DRA`	`0x0004000000000001`	`200`
	`ATODMA`		
	`RTOM`	`300`	`3`

ovvero il contenuto del registro 300 viene copiato nella locazione di memoria 1 + (contenuto del registro 200) + [(contenuto del registro 450) × (contenuto del registro 451)].

Le informazioni scritte nella sezione di codice delle istruzioni (.text) nei file ELF generati dal NAS devono essere una rappresentazione binaria di elementi di Micro-Assembler. Questi possono essere istruzioni, operandi o pseudo-istruzioni. Tra le ultime troviamo, ad esempio, la definizione delle etichette

incontrate nel codice, che servono per suddividere il programma in sezioni come richiesto dallo shaker.
Istruzioni e operandi hanno la stessa grandezza per una più semplice manipolazione: si è scelto di far rientrare tutte le informazioni per ciascun elemento in 128 bit, che è il minimo indirizzabile nella memoria di un nodo *apeNEXT*. Questa scelta evita che, nella fase di linking del codice, al momento di risolvere eventuali simboli referenziati vengano generati degli indirizzi non allineati e quindi ingestibili. Il formato creato prevedede 128 bit che identificano una istruzione o una pseudo-istruzione seguiti da zero o più operandi, secondo i seguenti schemi:

Istruzione	
codice istruzione	16 bit
numero operandi	32 bit
codice condizionamento	8 bit
linea del sorgente	32 bit
[vuoto]	40 bit

Operando	
valore	64 bit
tipo operando	16 bit
[vuoto]	48 bit

Da notare che un operando contiene anche una specifica del suo tipo, caratteristica spesso assente; questo permette un avvicinamento al modo di pensare del programmatore, facilitando la comprensione dei dati in memoria.

Per quel che concerne la sezione dei dati inizializzati (`.data`) e di definizione delle slice (`.slice`) esse sono già pronte ad essere sottoposte a linking e lette dal loader del sistema operativo senza dover essere ottimizzate. Di conseguenza anche esse seguono parimenti l'allineamento a 128 bit necessario ad un corretto indirizzamento.

Con questi risultati è stato relativamente semplice costruire degli strumenti di accesso ai file oggetto. Sono state quindi create le **apeAPI**, un insieme di routine C che astraggono dalle funzioni fornite con la libreria BFD per fornire dei meccanismi di accesso, creazione e modifica dei file oggetto secondo una semantica ragionevole per lo shaker ed il simulatore.

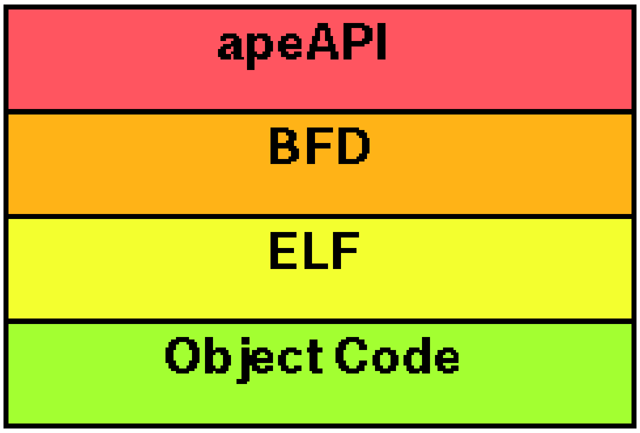

Figura 2.1: i livelli di astrazione delle apeAPI

Sono proprio le apeAPI che svolgono il lavoro di traduzione dal formato oggetto scritto nei file ELF nelle righe di micro-istruzioni comprensibili dall'ottimizzatore.

Ad esempio il seguente è un frammento di codice C che apre un file oggetto e stampa a video una descrizione dei dati inizializzati contenuti al suo interno:

```
ae = apelf_open(filename);
if (!ae){
  printf("Cannot read object file:\n");
  exit(0);
}
for (i = 0; apelf_read_const_to_char(ae, &buf, i); i++) {
printf("%s\n", buf);
}
```

La definizione delle funzioni presenti nelle apeAPI si può trovare in appendice.

2.5 Linking

Una delle parti più significative del porting delle Binutils per APE riguarda il supporto al *linking* del codice oggetto. Il vantaggio di avere dei formati per i file oggetto standard risiede proprio nel poter utilizzare una serie di strumenti largamente diffusi con uno sforzo minimo di adattamento. Il linking è un esempio molto importante sotto quest'ottica.

Il linker GNU compreso nelle Binutils (**ld**) [15] accetta come input file oggetto nei formati supportati dalla versione delle BFD con cui viene compilato. Esso legge i contenuti delle sezioni presenti in questi file in maniera sequenziale e le unisce rilocandole secondo l'ordine di lettura. Ad esempio per effettuare il linking della sezione .data di tre file oggetto (*file1.rno*, *file2.rno*, *file3.rno*) partendo dall'indirizzo 0x4000 potrebbe generare un risultato come in figura 2.2.

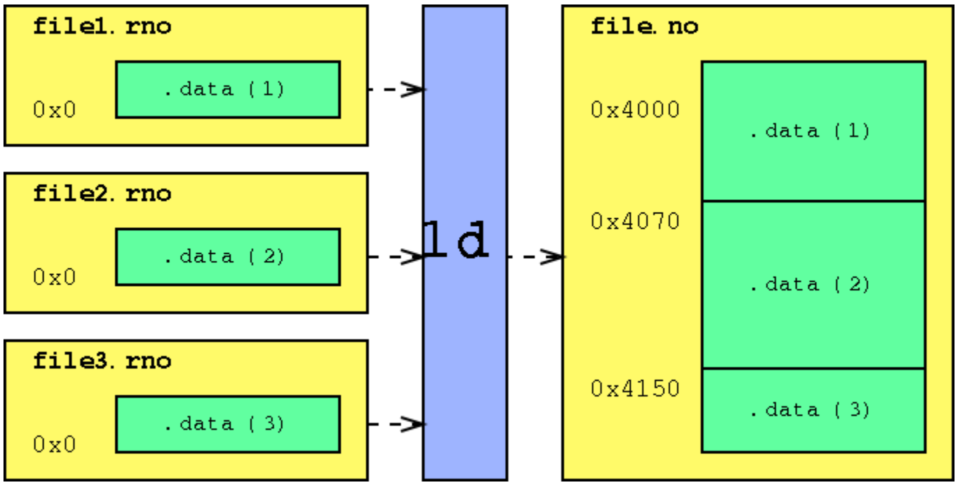

Figura 2.2: rilocazione con il linker

Ovviamente ora i riferimenti inclusi nel codice ai dati contenuti in queste sezioni presenti nei vari file subiscono una notevole modifica. Ad esempio se un'istruzione avesse acceduto al dato contenuto all'indirizzo 0x30 della sezione .data del file file2.rno ora questo riferimento dovrebbe essere aggiornato con il valore 0x40A0. Grazie al meccanismo delle *reloc* previsto nelle BFD è sufficiente indicare durante l'assemblamento che tale indirizzo in realtà è un riferimento ad una zona di memoria che può venir rilocata durante la fase di linking. Sarà compito del codice delle BFD (in parte codice ereditato dalle librerie ed in parte scritto durante il porting sulla nuova architettura), sotto il controllo del linker, gestire tale evenienza mantenendo la coerenza nel codice prodotto.

Il codice scritto per supportare le peculiarità di *apeNEXT* nelle BFD (e quindi nel linker) risulta sostanzialmente coprire i seguenti aspetti:

- la definizione dei reloc previsti dall'architettura (a 64 bit, come l'ampiezza massima di un operando di memoria)
- la modifica degli indirizzi per rispecchiare i 128 bit di unità minima indirizzabile in memoria
- l'esplicita specifica degli indirizzi di partenza per le sezioni .text (codice), .data (dati inizializzati) e .bss (dati non inizializzati)
- la definizione dei parametri della sezione .slice (contenente le specifiche di slice e gli effettivi dati di memoria in esse contenuti), come la sua persistenza dopo il linking.

La maggior parte di questo codice si trova nei file di configurazione di ld che vanno sotto il nome di *Emulation Parameters* e nei file sorgente generati per il supporto alle BFD (elf64-apenext.c), ed in vari file header.

2.5.1 La catena di assemblamento

Partendo dalle considerazioni precedenti si delinea facilmente l'intera catena di assemblamento di *apeNEXT*, cioè l'insieme di passi da effettuare per partire da un insieme di file sorgenti NAS (*.NS*) per arrivare ad un file oggetto non rilocabile da sottoporre allo shaker (*.NO*). Tali file devono prima pre-processati con il **NASP** (Next ASsembler Preprocessor), poi assemblati con il **NAS** (Next ASsembler) per divenire dei file oggetto rilocabili (*.RNO*). Più file oggetto rilocabili possono essere linkati assieme da **NLD** (Next Linker) e produrre un file oggetto non-rilocabile (*.NO*).

Tutti i passaggi qui specificati possono essere visti in figura 2.3.

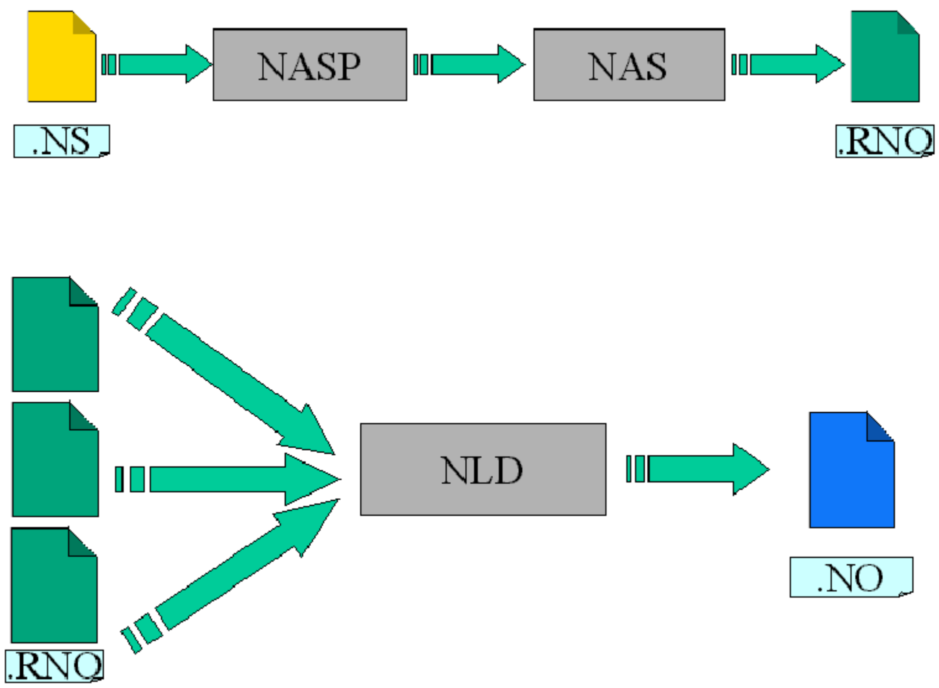

Figura 2.3: la catena di assemblamento

Inoltre i file oggetto (rilocabili o non rilocabili) possono essere salvati in file di Micro-Assembler (*.MS*) attraverso il comando **NO2MASM**. Questa piccola applicazione è stata scritta con le apeAPI e permette agli strumenti come lo shaker di leggere i contenuti dei file oggetto in maniera testuale.

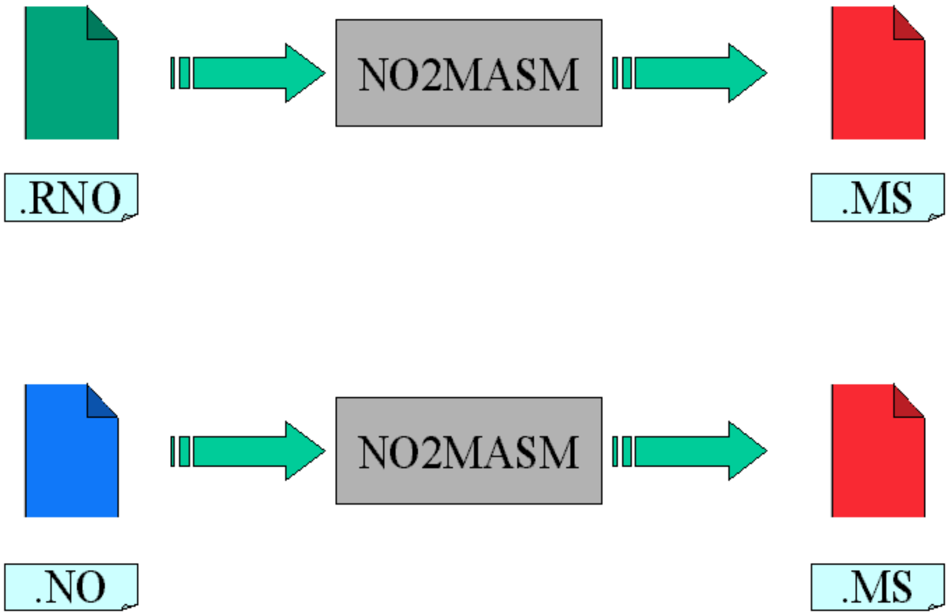

Figura 2.4: NO2MASM

Capitolo 3

Il Sistema Operativo di *apeNEXT*

3.1 I Sistemi Operativi

Un **Sistema Operativo** è un software in grado di organizzare le risorse della macchina, ottimizzandole e semplificandole. Esso astrae l'architettura della macchina su cui si trova fornendo dei metodi per accedervi e rappresenta la base su cui vengono di seguito creati gli altri livelli verso l'interazione con l'utente (vedere figura 3.1).

Figura 3.1: i livelli di astrazione di una macchina

I sistemi operativi sono stati introdotti fin dalle macchine *uniprogrammate* cioè in grado di eseguire un solo processo. Di seguito, con la *multiprogrammazione* (cioè la capacità di eseguire più processi concorrentemente), hanno poi assunto una importanza sempre crescente, fino a gestire gran parte deli problemi legati all'astrazione rispetto alla macchina [16].

Un sistema operativo moderno è in grado di gestire uno o più dei seguenti aspetti di un elaboratore:

- **gestione del processore**: controllo dei processi e delle interruzioni
- **gestione della memoria**: allocazione e deallocazione dinamica della memoria
- **gestione del file system**: astrazione del concetto di file (archivio) su memorie di massa
- **gestione dell'I/O**: interazione con dispositivi esterni alla CPU
- **gestione della rete**: comunicazione con elaboratori remoti

Gran parte della gestione del processore è demandata al *kernel* (nucleo) del sistema operativo. Esso si occupa della gestione di interruzioni e dello scheduling dei processi, e su di esso si basano gli altri moduli per eseguire le proprie operazioni [17].

Il concetto fondamentale su cui si basa il kernel di un sistema operativo moderno è quello di *interruzione*.

Le interruzioni (o *interrupt*) sono dei segnali riconosciuti dalla Control Unit di un processore e possono essere *sollevati* (cioè inviati verso la CPU) da alcuni dispositivi hardware o via software. Il sollevamento di una interruzione causa la sospensione immediata dell'esecuzione (a meno di *mascherare* una interruzione, cioè renderla invisibile alla CPU) ed il passaggio ad una routine di gestione delle interruzioni. Ogni tipo di interruzione ha una sua routine di gestione particolare e la CPU è in grado di capire quale attraverso dei meccanismi di riconoscimento delle interruzioni, come il *polling* o la *vettorizzazione* [18].

Attraverso l'utilizzo delle interruzioni è possibile effettuare efficentemente numerose operazioni a livello di sistema operativo. Ad esempio il gestore della memoria utilizzerà delle interruzioni hardware (i *page fault*) per permettere l'utilizzo della memoria virtuale.
Grazie all'esistenza delle interruzioni software (*trap*) è anche possibile creare dei meccanismi di invocazione del sistema operativo da parte di programmi utente. Le *system call* sono proprio delle routine che vengono richiamate al sollevamento di determinate trap nel codice utente e che, nei processori che la supportano, eseguono una transizione in modalità supervisore per poter accedere a particolari istruzioni o dispositivi altrimenti irraggiungibili. Le system call vengono invocate quando un programma ha la necessità di eseguire alcune operazioni demandate al sistema operativo, come l'accesso al file system, la comunicazione remota, la creazione di processi, etc.

Da ciò si intuisce anche come sia molto importante il supporto hardware fornito dalla macchina ai sistemi operativi. È solo grazie a dell'hardware specifico che risulta possibile effettuare system call e passare a modalità supervisore; oppure gestire la memoria virtuale grazie a dei particolari accorgimenti nel modulo di gestione della memoria.

Per quel che riguarda l'ipotesi di sistema operativo di *apeNEXT*, per affrontare la sua discussione è necessario partire da una piccola analisi di *APEmille*; poiché le innovazioni introdotte con la nuova macchina nascono da alcune considerazioni sulle esperienze precedenti.

3.2 Il Sistema Operativo di *APEmille*

Come già accennato *APEmille* ammette due diverse modalità di esecuzione: **Run-mode** e **System-mode** [19]. Esse sono profondamente distinte tra loro e condividono anche pochi componenti elettronici (principalmente in J1000 e C1000); ad esempio i trasferimenti remoti in Run-mode avvengono attraverso le linee inter-nodo ad alta velocità, mentre in System-mode attraverso l'interfaccia CompactPCI comandata dalle macchine host. L'esecuzione del sistema operativo della macchina avviene in System-mode. In System-mode la macchina è quasi completamente sotto il controllo degli host ad essa collegati. Essi condividono un sistema operativo distribuito in grado di accedere, attraverso i loro bus CompactPCI, alle Processing Board che comandano.

Le system call, in *APEmille*, si chiamano *System Service* e vengono eseguite in System-mode. Sono invocate da alcune macro assembler predefinite che rappresentano l'interfaccia di programmazione verso il sistema operativo.

Tutte le System Service sono delle operazioni di input/output effettuate su delle particolari risorse, che possono essere sequenziali o parallele, oppure di sola lettura, di sola scrittura o di lettura/scrittura. Come esempio di risorse sequenziali abbiamo la tastiera (sola lettura), lo schermo (sola scrittura) o un file utente (lettura/scrittura).

Una System Service viene definita da un insieme di dati di lunghezza fissa (*IOP* - I/O Packets) che descrivono:

- il comando da eseguire (Open, Read, etc.)
- la risorsa destinazione (ID di un file, schermo, etc.)
- il dispositivo interno interessato (Memoria T1000, Memoria J1000, etc.)
- indirizzo nella risorsa
- una specifica di **slice**
- numero di word da trasferire

La tipica sequenza di attivazione di una System Service è:

1. Run-mode: creazione dello IOP nella memoria e riempimento dei suoi campi
2. Run-mode: richiesta di trasferimento a System-mode eseguendo una istruzione di *trap* (HALT) che congela il clock dei nodi di calcolo

3. System-mode: il Sistema Operativo si accorge del passaggio di modalità ed esegue i seguenti passi:

    (a) distinguere tra normali operazioni di System Service e condizioni di errore (*eccezioni*) che possono verificarsi

    (b) leggere la descrizione macroscopica della System Service (come: operazioni sequenziali/parallele, etc.)

    (c) leggere lo IOP creato

    (d) decodificare l'azione richiesta

    (e) accedere al dispositivo interessato

Il sistema operativo comunica con i diversi dispositivi seguendo strade differenti:

- il PC host (master) accede agli host delle unit via LAN
- gli host delle unit comunicano con le Processing Board come una risorsa PCI *memory-mapped*
- le code di T1000 vengono accedute attraverso il canale *ApeChannel*
- T1000 accede attraverso le linee interne agli altri dispositivi della PB

Tutto il meccanismo descritto viene, come già specificato, comandato a livello di programmazione attraverso chiamate a macro assembler. Esse si

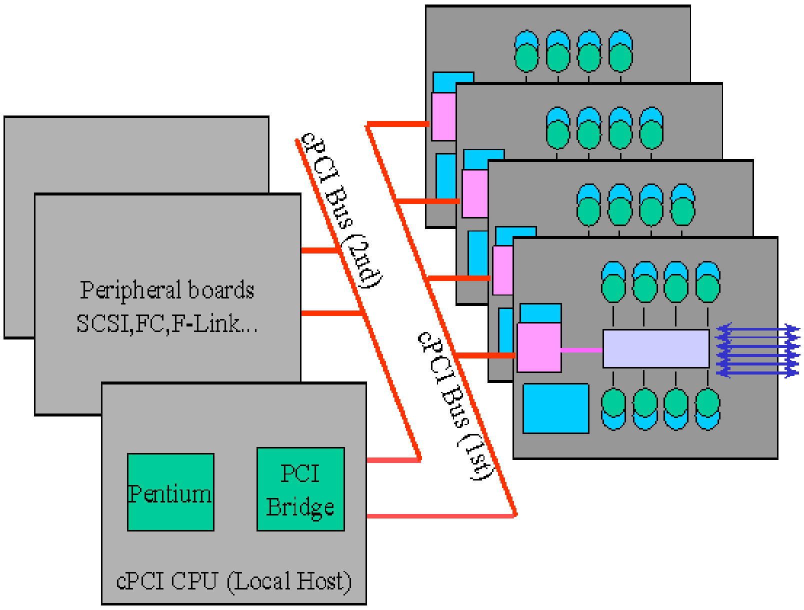

Figura 3.2: una Unit di *APEmille* e l'host a cui è connessa

occupano di creare gli IOP e riempirli con i giusti campi e di passare in System-mode.

Ad esempio possiamo vedere il seguente codice TAOmille che esegue una *normal* (A*B+C) su numeri complessi:

```
complex a,b,c
  read b
  read c
    a=b*c
  write a
```

tale codice produce il JASM (assembly) di seguito riportato:

```
$main
```

```
GLOBAL       "$rnd_tmp"  %0x400   1
!! ___   1  complex a,b,c
GLOBAL       "a"  ^0x400   2
GLOBAL       "b"  ^0x402   2
GLOBAL       "c"  ^0x404   2
!! ___   3  read b
LABEL *4096
JLOFRESET
$iolist_start
$iolist_slice_all
$iolist_packet(RES_STDIN.0,<READ_BRDCST>,DEV_JN_DATA,
               FMT_COMPLEX,0.1,^0x402.0)
$iolist_end
!! ___   4  read c
LABEL *4098
JLOFRESET
$iolist_start
$iolist_slice_all
$iolist_packet(RES_STDIN.0,<READ_BRDCST>,DEV_JN_DATA,
               FMT_COMPLEX,0.1,^0x404.0)
$iolist_end
!! ___   5
!! ___   6  a=b*c
!! ___   7
!! ___   8  write a
LABEL *4100
JLOFRESET
MEMTOJ1      20 :2 ^0x402
MEMTOJ1      22 :2 ^0x404
JCNORM_PP    24 20 22 0
JTOMEM1      ^0x400 24 :2
```

```
$iolist_start
$iolist_node_default
$iolist_packet(RES_STDOUT.0,<WRITE>,DEV_JN_DATA,
               FMT_COMPLEX,0.1,^0x400.0)
$iolist_end
!! ___   9
LABEL *4102
JLOFRESET
$stop(HS_SUCCESS)
$stop(HS_SUCCESS)
```

Le System Service in questo frammento di codice sono *iolist_start*, *iolist_node_default*, *iolist_packet*, *iolist_end*. Queste generano uno IOP di default a cui aggiungono le informazioni rilevanti il servizio effettivamente richiesto.

3.3 Il Sistema Operativo di *apeNEXT*

Per progettare il sistema operativo di *apeNEXT* è necessario partire dall'esperienza avuta con il neonato *APEmille*. Innanzitutto risulta evidente che è necessario gestire a livello di sistema operativo due tipi di eventi: *sincroni* ed *asincroni*. Gli eventi sincroni riguardano quelli chiamati in *APEmille* "**System Service**", quindi operazioni di accesso ai dispositivi esterni da parte di un nodo o a risorse interne da parte delle macchine host, ad esempio a scopo di debugging. Gli eventi asincroni riguardano invece le condizioni di errore o di notifica di condizioni impreviste, quindi delle **interruzioni**. Inoltre bisogna ricordare che la macchina deve anche poter gestire un **Debug-mode** per permetterne l'ispezione comandata dagli host esterni. In *APEmille* il Debug-mode coincideva con il System-mode e, sostanzialmente, le System Service erano implementate su di esso.

L'attivazione di questi servizi in *APEmille*, come già specificato, causava una transizione a System-mode e passava quasi tutto il controllo sui dispositivi al T1000 della PB. In *apeNEXT*, peró, i compiti svolti congiuntamente da J1000, T1000 e C1000 vengono ora accorpati nell'unico processore che implementa il nodo di calcolo. Risulta quindi evidente che gestire la transizione a System-mode implicherebbe l'aggiunta di una logica appropriata all'interno del nodo, in grado di funzionare autonomamente alla sua attivazione e di accedere a tutti gli altri dispositivi (come il Register File o la memoria).

Un notevole cambiamento rispetto a questa prospettiva consisterebbe nell'implementazione di quasi tutta la modalità di sistema operativo sulle basi di quella utente. Si dovrebbe cercare di sfruttare le normali operazioni della macchina, comprese le comunicazioni inter-nodo, in entrambi gli stati. In questo modo anche le modifiche hardware inevitabili nelle fasi di debugging e ottimizzazione diverrebbero minori e per lo più rimpiazzate dall'aggiornamento software del sistema operativo.

3.3.1 Considerazioni sull'architettura del nodo

Ci sono alcune caratteristiche del nodo di *apeNEXT* che devono essere considerate più approfonditamente per poter effettuare alcune decisioni sulla strada da intraprendere. Bisogna precisare che parte degli aspetti qui indicati costituiscono solamente delle linee-guida ipotizzate per la progettazione della macchina e sono, al momento della stesura di questa tesi, ancora in fase di discussione all'interno della collaborazione APE.

Abbiamo già specificato che il meccanismo di sincronizzazione di *apeNEXT* è *lasco* e si effettua principalmente nelle comunicazioni inter-nodo. In ciò *apeNEXT* differisce dalle macchine precedenti della famiglia APE che sono rigorosamente sincronizzate a livello di clock. Esistono quindi 6 code (FIFO), una per ogni direzione di comunicazione (X^+, X^-, Y^+, Y^-, Z^+, Z^-) (vedere fig. 3.3). Si assume che il modulo di comunicazione del nodo implementi al suo

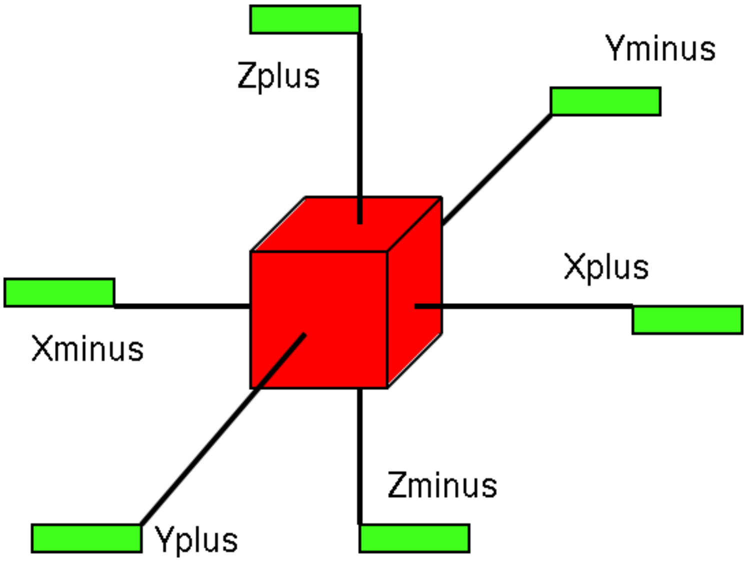

Figura 3.3: il nodo di calcolo con le sue 6 direzioni di comunicazione

interno un meccanismo per cui esso può essere considerato *affidabile*, attraverso i meccanismi di ritrasmissione e di *acknowledge* (ACK) necessari.

Rispetto ad *APEmille* la macchina ora si avvicina notevolmente ad abbracciare il paradigma MIMD (Multiple Instruction Multiple Data). La sincronizzazione della macchina ora risiede esclusivamente nelle comunicazioni, quindi non c'è più la necessità di eseguire su ogni nodo la stessa istruzione nello stesso ciclo di clock. È sufficiente che vengano rispettate, in intervalli ragionevoli di attesa, le comunicazioni tra i nodi.

Per quel che riguarda la memoria vale la pena ricordare che quella ad accesso casuale stiperà sia dati che istruzioni. La cache prevista, però, è l'unico modo per accedere a quelle locazioni che contengono le istruzioni VLIW. Quindi, a livello logico, possiamo dividere le risorse di memoria in due: la **Data Memory** e la **Instruction Cache**. Vanno poi considerati il **Register File** e i **Configuration Register**, i quali implementano, tra le altre cose, un meccanismo di indirizzamento del tutto differente dai MOP (*Memory Operand*) già visti.

3.4 Progettazione del kernel

Nell'ottica di minimizzare (o annullare) le modifiche hardware necessarie per differenziare il comportamento della macchina durante l'esecuzione di programmi utente (lo *User-mode* di *APEmille*) e durante l'esecuzione di routine di sistema operativo (il *System-mode*) è necessario tentare di utilizzare le caratteristiche del nodo indipendentemente in ambeude gli stati. I due stati sono differenti a livello logico; vediamo alcune di queste differenze:

- le comunicazioni avvengono, in User-mode, solo tra nodi della rete, senza interazione con gli host esterni

- in System-mode non ci si attende comunicazioni inter-nodo, poiché si ha solo necesità di comunicare con gli host esterni

- il System-mode è una specie di modalità *supervisore* in cui è possibile accedere a particolari risorse del nodo senza restrizioni

Per rendere minima la loro differenza nell'architettura del nodo è quindi necessario sfruttare anche in System-mode le stesse operazioni permesse nei programmi utente.

Le comunicazioni verso gli host, quindi, devono avvenire attraverso le normali code presenti. È quindi necessario un meccanismo di *routing* dei pacchetti del sistema operativo attraverso i normali nodi della rete in direzione degli host. Alcuni nodi su alcune Processing Boards devono poi contenere dei collegamenti verso le **Programmable Logical Device** (PLD) che si occupano della comunicazione con il bus condiviso con l'host (vedere fig. 3.4). In *APEmille* queste PLD erano dei dispositivi **Altera** che comunicavano con i bus CompactPCI degli host.

I nodi che contengono questi collegamenti (che possiamo chiamare **Bridge Node** o **BN**) diverranno, quindi, le destinazioni delle comunicazioni nodo→host ad alta velocità, per i nodi della PB. Le comunicazioni ad alta velocità sono quelle chiamate *parallele*, perché solitamente destinate all'I/O su memorie di massa. Un certo numero di Bridge Node si occuperà invece della comunicazione

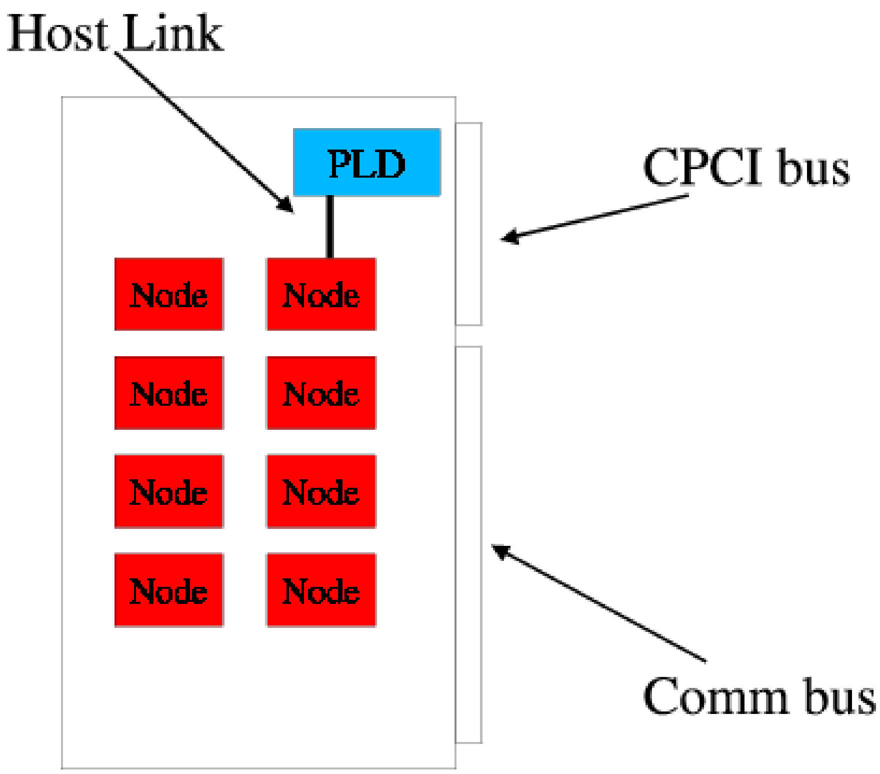

Figura 3.4: una ipotetica PB con otto nodi

nodo→host per i comandi di sistema e per le comunicazioni *seriali* (come, ad esempio, l'output su schermo); questi nodi saranno i **Default Bridge Node** e saranno collegati ai **Default PLD**, ed in prima approssimazione possiamo considerare che ne esista uno per rete (vedi figura 3.5). Da notare come si può essere ancora più restrittivi considerando collegate al bus di comunicazione con l'host solo le PB contenenti i nodi di default; ciò non modifica la funzionalità del sistema inficiando solamente le prestazioni nelle comunicazioni parallele. Potrà essere compito del programmatore indicare il rapporto di comunicazione

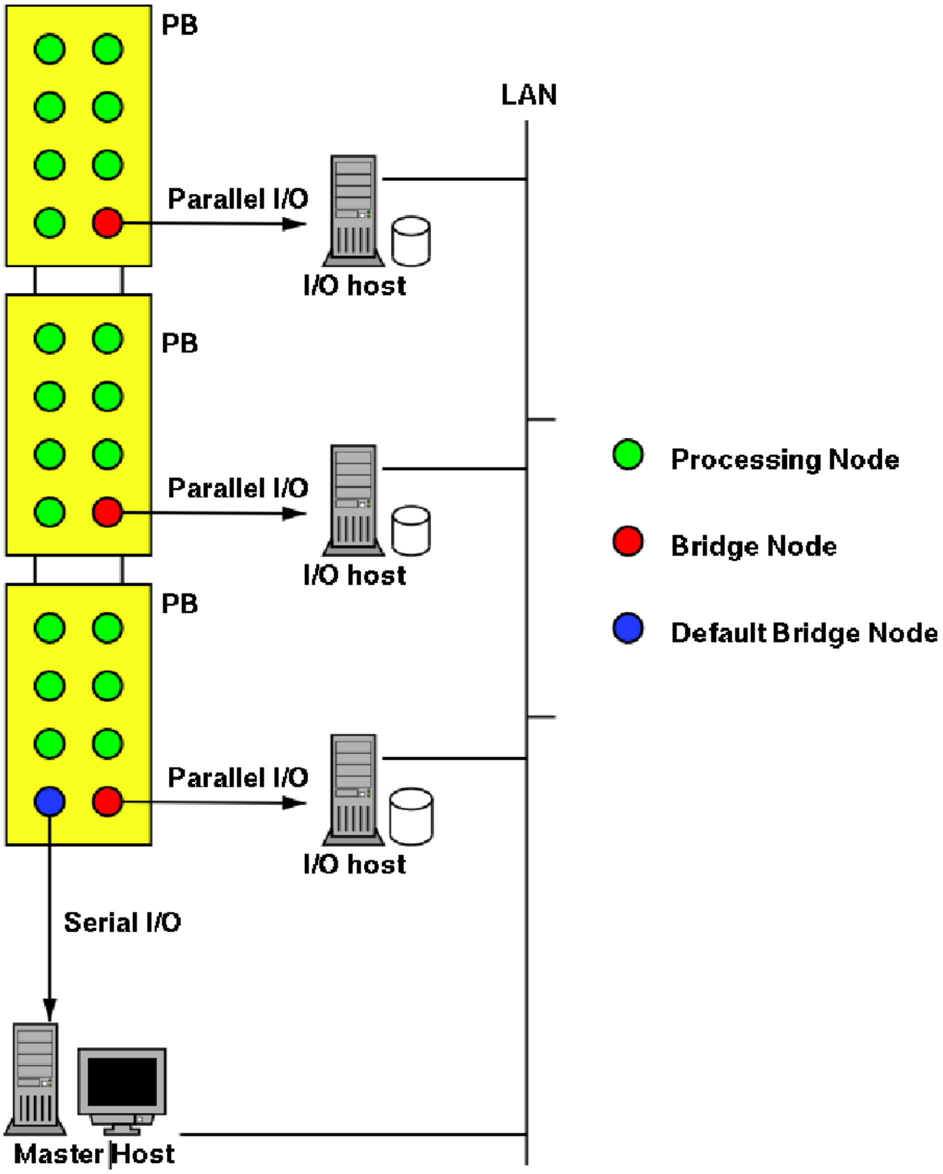

Figura 3.5: schema delle PB e dei Bridge Node

tra i BN e gli host collegati, secondo le esigenze della particolare applicazione.

I BN, seguendo i concetti esposti precedentemente, devono comunicare con i PLD attraverso le stesse linee utilizzate con gli altri nodi. Si può ad esempio immaginare che esista una settima coda di comunicazione remota (**Host Link**) che si occupi di questo. Uno schema di un nodo con la settima coda si può trovare in fig 3.6.

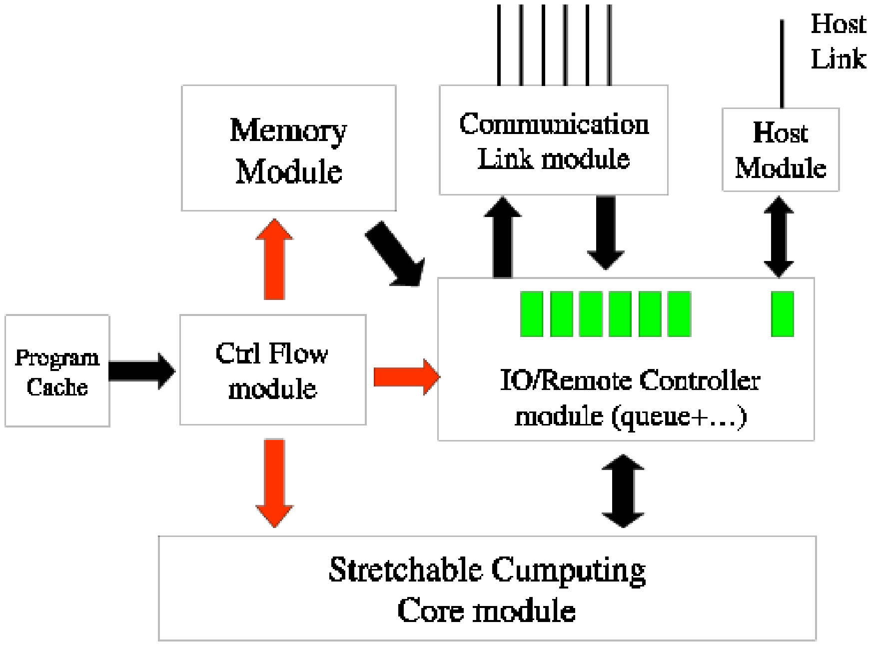

Figura 3.6: lo schema di un nodo con tutte le sue componenti più la settima FIFO

L'accesso a queste risorse deve essere possibile solo nello stato di System-Mode, e viene subito alla luce una delle differenze ipotizzate prima tra i due diversi stati di funzionamento del nodo.

Dagli schemi illustrati si può prevedere che il sistema operativo di *apeNE-XT* sarà suddiviso in 3 parti:

1. parte sul *Master Host*, cioè l'elaboratore connesso al Defult Bridge Node
2. parte sul nodo di calcolo
3. parte sull'*I/O Host*, cioè l'elaboratore connesso agli altri Bridge Node

In questo lavoro si è studiato, in particolare, il secondo aspetto del sistema operativo, fornendone anche una implementazione, come si potrà vedere di seguito.

3.4.1 Kernel-Mode e User-Mode

Essendo ora le differenze hardware tra i due stati di funzionamento molto lievi bisogna identificarli secondo parametri diversi. Chiamiamo **User-Mode** il funzionamento del nodo mentre sta eseguendo codice utente, proveniente dal programma caricato in memoria. Quando il nodo esegue codice destinato al sistema operativo, e quindi alla gestione di eventi sincroni e asincroni di sistema, si parla di **Kernel-Mode**.

Lo stato di Kernel-Mode è simile al System-Mode di *APEmille*, ma mantiene una diversità esclusivamente logica con lo stato di esecuzione normale delle operazioni, a differenza di quanto accadeva nella macchina precedente. Ciò permette la distinzione tra i due stati sarà un flag nell'hardware che indicherà l'ingresso o meno in Kernel-Mode.

In Kernel-Mode vengono eseguite delle routine di sistema operativo residenti su delle locazioni ben specifiche di memoria (e che compongono il **Kernel** su nodo). Tali routine si occupano di gestire gli eventi che si verificano durante la normale esecuzione mantenendo di conseguenza attive le comunicazioni con i nodi vicini e con gli host. Ecco elencate, brevemente, alcune delle operazioni in cui vengono richiamate routine del Kernel:

1. interruzione hardware (comprese le eccezioni dovute ad errori nelle unità di calcolo, di memoria, di comunicazione, etc.)
2. interruzione software (*Trap*)
3. gestione dell'I/O con l'esterno
4. avvio (*bootstrap*) della macchina ed inizializzazione dello stato del nodo
5. reset software della macchina

Eventi come 1 o 4 sono ovviamente generati dall'hardware specifico predisposto allo scopo. Eventi come il 2 o il 3 sono invece generati dal programma utente attualmente in esecuzione. Infatti i meccanismi per la transizione User-Mode→Kernel-Mode si possono raggruppare sotto il concetto generico di **Trap**. Sono dei Trap:

le interruzioni software (generate ad esempio in un programma di debugging)

le System-Service (richiamate, ad esempio, per eseguire I/O verso l'esterno)

Quando l'evento di passaggio a Kernel-Mode è sincrono al termine dell'esecuzione delle routine appropriate il passaggio tornerà al programma chiamante il quale non vedrà modificato il suo stato.

Ciò può somigliare alla gestione dei processi in macchine con il supporto per essi. In realtà le macchine APE non hanno mai avuto nel loro sistema operativo il supporto al multiprocessing, e neppure un hardware specializzato per questo supporto. Si capiscono le motivazioni di questa scelta considerando che, a parte le routine di sistema operativo, l'unico processo che può essere eseguito in un dato momento è il programma utente. Aggiungere un meccanismo di *context-switch* (passaggio da un *contesto*, o processo, ad un altro e viceversa) per supportare i processi multipli non ha senso in una macchina il cui scopo fondamentale è fornire le prestazioni necessarie alle applicazioni per le quali è creata.

Quando, però, il passaggio al sistema operativo avviene quasi interamente via software, come nella nostra ipotesi, un meccanismo di context-switch risulta necessario. Il supporto hardware ad esso, in prima approssimazione, si può ridurre al solo salvataggio in un registro di configurazione del valore del Program Counter, contenuto nella Sequence Unit.

3.4.2 Il Kernel su nodo

Definiamo ora la modalità di operazione della porzione di sistema operativo residente su nodo.

Questa sarà composta da un certo numero di istruzioni VLIW scritte a partire da un indirizzo di memoria predefinito ed uguale in tutti i nodi.

Il kernel ha un cuore in cui vengono eseguite tutte le sue operazioni di gestione sopra accennate che si chiama **TinyKernel**. Il TinyKernel viene invocato al riconoscimento di una interruzione e completa la sua esecuzione passando di nuovo il controllo al programma utente interrotto, se possibile.

Le funzionalità del TinyKernel vengono comandate da pacchetti informativi contenenti le operazioni da svolgere, oppure da particolari valori in determinati registri di configurazione del nodo. Il primo caso si applica per comunicazioni sincrone, cioè quando il programma utente invoca una Trap; il secondo per eventi asincroni e le informazioni riguardano per lo più la specifica dell'interruzione da gestire.

I pacchetti informativi decodificati dal TinyKernel sono analoghi a quelli descritti in (3.2) per il sistema operativo di *APEmille*, e possono quindi venire chiamati anch'essi **IOP** (I/O Packets). Questi sono interpretati allo stesso modo dalle routine di sistema interne al nodo e da quelle residenti sugli host e possono venire generati da ambedue queste entità.

Una prima approssimazione del ciclo di vita di una invocazione del TinyKernel risulta quella descritta in figura 3.7

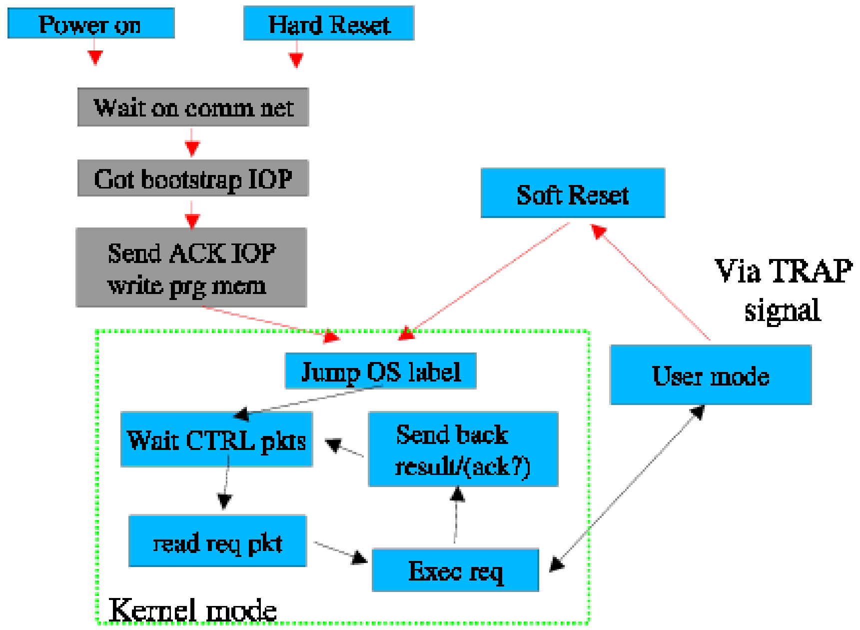

Figura 3.7: il passaggio User-Mode→Kernel-Mode

Gli IOP sono le uniche informazioni comprese sia dagli host che dai nodi e quindi vengono utilizzati anche in tutti quegli eventi asincroni generati dagli host, come il bootstrap della macchina o l'attivazione di una modalità di debugging. Essi devono quindi avere un formato quanto più completo possibile.

Nella scelta tra IOP di grandezza fissa e IOP di grandezza variabile si è optato, in questa realizzazione, per la prima possibilità. Le considerazioni che stanno alla base di questa scelta sono: la facilità di manipolazione, ed una maggiore aderenza con determinate caratteristiche dell'architettura, come l'indirizzamento del Register File che non consente operandi a lunghezza variabile. Ovviamente per implementare comunicazioni di lunghezza arbitraria con pacchetti a lunghezza fissa si è dovuto anche implementare un meccanismo di de/frammentazione dei dati da utilizzare sia sul TinyKernel che sulla parte del sistema residente sull'host. Il formato dello IOP risulta quindi quello visibile in figura 3.8. Da notare che con il termine **word** si indica una parola di *apeNEXT*, cioè 64 bit.

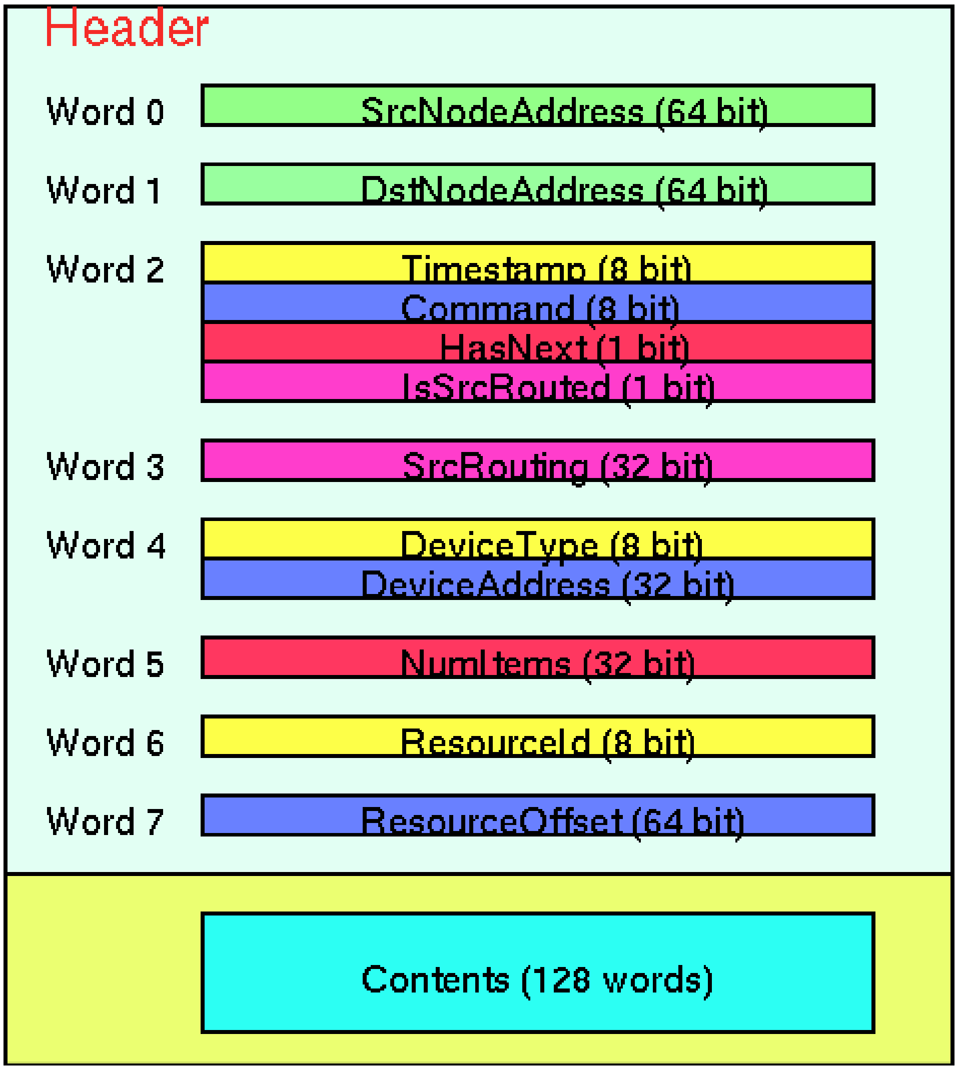

Figura 3.8: formato dell'I/O Packet

Ecco una breve descrizione dei campi:

SrcNodeAddress	indirizzo del nodo sorgente
DstNodeAddress	indirizzo del nodo destinazione
Timestamp	tempo di vita del pacchetto
Command	comando da eseguire
HasNext	se esistono altri frammenti del pacchetto
IsSrcRouted	se l'instradamento è di tipo *source routing*
SrcRouting	informazioni sull'instradamento *source routing*
DeviceType	tipo della risorsa su nodo coinvolta
DeviceAddress	indirizzo della risorsa coinvolta
NumItems	numero di parole trasferite
ResourceId	identificativo della risorsa a cui accedere
ResourceOffset	indirizzo all'interno della risorsa
Contents	contenuto del corpo del pacchetto

quindi le prime 8 parole di uno IOP formano un **header** che ne descrive le caratteristiche, mentre le 128 parole successive sono il contenuto effettivo del pacchetto.

Alcune precisazioni sui significati dei campi:

- ogni nodo ha un identificativo univoco che viene scritto in esso in maniera fisica (con un meccanismo di jumper o con un firmware apposito). Il formato di questo identificativo non è stato ancora deciso, ma per individuare l'indirizzo di un nodo si è scelto di indicare la sua posizione nel reticolo della macchina. Quindi un indirizzo, nella terminologia del sistema operativo, è una tripla di coordinate, ognuna da 8 bit

- *hasNext* indica, in quei pacchetti che data la loro grandezza hanno subito frammentazione, che il pacchetto successivo è ancora un frammento del presente. Questo non modifica la funzionalità del kernel, per il quale si cerca di costruire dei pacchetti coerenti indipendentemente dal numero di frammenti che li compongono, ma è una indicazione alla componente host del sistema. Infatti alcuni trasferimenti di dati verso le memorie di massa verranno ottimizzate per lunghi set di dati (*burst*), ed avere questo genere di informazione può migliorare le prestazioni del sistema

- il *source routing* è un mecanismo di instradamento per cui è il nodo sorgente che identifica a priori il percorso che il pacchetto deve effettuare. Ciò può servire per debugging o per evitare percorsi che si conoscere essere inutilizzabili. Questo meccanismo non viene implementato nella versione del TinyKernel qui esposta

- i *Device* sono i componenti interni indirizzabili del sistema, quali la memoria programma o i registri di configurazione. Le *Resource* sono invece le risorse accessibili dagli host, come archivi su memorie di massa, la tastiera e lo schermo.

un esempio di esecuzione del TinyKernel può essere il seguente:

1. il programma utente richiede la lettura di un dato di tipo reale da tastiera e la sua scrittura nell'indirizzo 0x1122AA della memoria dati

2. l'esecuzione passa al TinyKernel

3. il TinyKernel invia uno IOP che indica i seguenti campi:

SrcNodeAddress	identificativo del nodo corrente
DstNodeAddress	identificativo del DBN
Timestamp	0
Command	Read
HasNext	0
IsSrcRouted	0
SrcRouting	0
DeviceType	DataMemory
DeviceAddress	0x1122AA
NumItems	1
ResourceId	Keyboard
ResourceOffset	Real (il tipo di dato da leggere)
Contents	0

4. il sistema dell'host esegue la lettura da tastiera del valore 0.5 ed invia uno IOP con i seguenti campi:

SrcNodeAddress	identificativo del DBN
DstNodeAddress	identificativo del nodo corrente
Timestamp	0
Command	Write
HasNext	0
IsSrcRouted	0
SrcRouting	0
DeviceType	DataMemory
DeviceAddress	0x1122AA
NumItems	1
ResourceId	Keyboard
ResourceOffset	Real
Contents	0.5

5. il TinyKernel legge lo IOP, lo decodifica e si comporta di conseguenza, scrivendo all'indirizzo 0x1122AA della memoria dati il valore 0.5

Da questo esempio si possono subito fare alcune considerazioni:

- una operazione di lettura da una risorsa si è tramutata in una operazione di scrittura su un device. In questo modo si utilizzano gli IOP per tutte le comunicazioni in Kernel-Mode, senza il bisogno di ideare modalità di risposta ad un comando. Questo riduce al minimo le richieste di memoria del TinyKernel, che non ha necessità di ricordare il comando che ha inviato per saper decodificare la risposta ricevuta per esso

- alcuni campi, che altrimenti non verrebbero utilizzati, vengono spesso *sovraccaricati* di significato. È questo il caso di *ResourceOffset* che, non avendo senso in operazioni di lettura da tastiera, può essere utilizzato per indicare il tipo di dato coinvolto. Spesso il campo che contiene questo tipo di informazioni aggiuntive è *Contents*, ma è più semplice utilizzare l'header, quando possibile.

Come può il programma utente indicare al TinyKernel quale comando eseguire? Il modo più semplice è costruire il pacchetto proprio all'interno del programma utente. Questo può essere ottenuto creando delle routine assembler apposite che costruiscano, in opportune locazioni di memoria dedicate allo scopo, gli header degli IOP da inviare. Nella transizione al TinyKernel queste locazioni verrebbero lette, verrebbero generati i campi *Contents* necessari (utili solo per operazioni di scrittura su risorse) e verrebbero quindi inviati agli host i pacchetti completi.

Le routine di sistema lato utente, quindi, riguardano la creazione di questi header e il passaggio in Kernel-Mode. Queste coinvolgono alcuni registri di configurazione (ad esempio quello che indica la locazione di memoria in cui trovare gli header generati) e parte della memoria riservata al sistema operativo.

Non è possibile sfruttare arbitrariamente il Register File per passare informazioni tra le due modalità viste le peculiarità dell'architettura di *apeNEXT*. Il concetto di registro virtuale, infatti, obbligherebbe ad indicare i registri interessati come *fisici*, da non rilocare in fase di ottimizzazione. Questo, però è un vincolo molto forte sul numero e sulla posizione dei registri da utilizzare nella transizione, poiché se esistessero dei registri da utilizzare per questo scopo essi non potrebbero mai venire utilizzati come registri fisici in User-Mode per non rischiare che venga a mancare la coerenza semantica nel codice. Viene quindi definito un insieme di registri ad uso del sistema che non possono essere definiti come fisici in User-Mode. Ma il loro numero deve inevitabilmente essere ridotto per lasciare abbastanza spazio sul Register File per l'elaborazione dell'utente. Vedremo al momento di analizzare il codice del TinyKernel quali registri fisici sono stati definiti ad uso specifico del sistema operativo.

Per quel che concerne gli eventi asincroni deve esistere un meccanismo di attivazione del TinyKernel al loro verificarsi, nonché la possibilità di riconoscerne il tipo e, possibilmente, la causa. Visto che, come già specificato, gli eventi asincroni devono essere individuati dall'hardware apposito (come ad esempio un *watchdog* nel modulo di comunicazione in grado di riconoscere ritardi eccessivi di trasmissione) essi possono essere trattati in maniera simile alle normali interruzioni nei sistemi operativi tradizionali, cioè l'hardware deve essere in grado di effettuare un salto in Kernel-Mode indicandone anche il motivo. Il salto può avvenire verso una locazione predefinita della memoria interna al TinyKernel, dalla quale dovrebbe venire eseguito il codice necessario a leggere i registri di configurazione opportuni per riconoscere l'interruzione avvenuta (è infatti pevisto un registro di eccezione ad indicare il tipo di interruzione avvenuta, assieme ad un registro di mascheramento per inibire il sollevamento di particolari eccezioni in particolari condizioni). Il TinyKernel, di conseguenza, deve ammettere almeno due *entry-points* (cioè due divere etichette di codice su cui poter effettuare un salto): uno per le system-call da codice utente ed uno per i salti dovuti ad interruzioni. Entrambe devono poi entrare, dopo aver effettuato le operazioni necessarie, in un unico ciclo, il **TinyKernel Loop** nel quale deve essere implementato il meccanismo di invio-ricezione di pacchetti. Infatti anche la segnalazione di una eccezione hardware all'host deve avvenire attraverso la creazione di determinati IOP, così come ci si può attendere che gli host, di conseguenza, inviino dei pacchetti per comandare determinate azioni di debugging.

Si potrebbe anche decidere di avere più di un entry-point per l'ingresso nel TinyKernel da parte di programmi utente; attualmente, però, si è scelto di permettere la creazione in memoria di più header di tipo differente, quindi di poter effettuare diverse system-call con un unico ingresso in Kernel-Mode. Di conseguenza diversificare le system-call indicando per ognuna un entry-point differente, per le scelte effettuate sinora, non è importante.

3.4.3 Il routing

Abbiamo già indicato che tutte le comunicazioni in Kernel-Mode devono comunque avvenire attraverso le normali linee inter-nodo. Se, quindi, un nodo distante 3 (definendo 1 la distanza tra due nodi contigui) dal suo PLD di default deve inviare un pacchetto verso l'host è necessario che i due nodi che si trovano nel cammino effettuino l'instradamento del pacchetto. Tale azione non può che compiersi in Kernel-Mode, visto che non è possibile demandarla al codice utente.

Il loop del TinyKernel, quindi, una volta avviato, deve essere in grado di ascoltare il modulo di comunicazione per riconoscere i pacchetti inviati dai suoi nodi vicini e di smistarli secondo il percorso che devono compiere. In generale il loop è, di conseguenza, un ciclo infinito di ascolto dalla rete. Quando riceve un pacchetto esso eseguirà i seguenti passi:

1. decodifica dell'header

2. se il pacchetto è indirizzato al nodo corrente deve essere decodificato e deve venire eseguito il comando specificato, accedendo alle risorse interne al nodo interessate

3. altrimenti il TinyKernel deve individuare il nodo contiguo a quello corrente su cui instradare il pacchetto

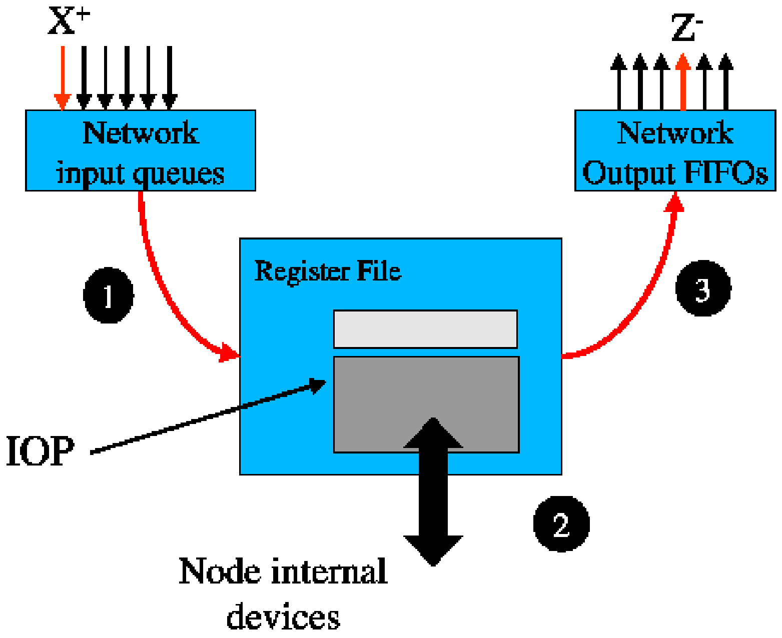

Figura 3.9: ricezione di un pacchetto in Kernel-Mode

Essendo *apeNEXT* una macchina programmabile secondo il paradigma MIMD se un nodo ha necessità di inviare uno IOP non è detto che anche i nodi ad esso prossimi abbiano la stessa necessità, e quindi entrino anch'essi in Kernel-Mode.

Stesso discorso si applica se l'evento che ha generato la necessità di passare in Kernel-Mode e di inviare un pacchetto verso la rete sia asincrono. Anche in questo caso i nodi prossimi non possono essere a conoscenza del fatto che dalle loro linee di comunicazione inter-nodo arriveranno delle informazioni che in realtà non sono attese e che costituiscono dei pacchetti da instradare.

Deve esistere quindi un meccanismo in grado di comandare il nodo contiguo coinvolto nel percorso ad entrare in Kernel-Mode. Ovviamente la soluzione più semplice consiste nella definizione di una nuova eccezione, da gestire con il metodo delle interruzioni, che faccia passare l'esecuzione a Kernel-Mode. L'hardware che deve accorgersi di questa evenienza risiede nel modulo di comunicazione che ha la facoltà, essendo esso stesso basato su pacchetti, di riconoscere il tipo di comunicazione in arrivo e di accorgersi che essa è stata generata dal kernel. In generale potremmo, considerata questa caratteristica, definire i pacchetti kernel ricevuti dalla rete come in figura 3.10.

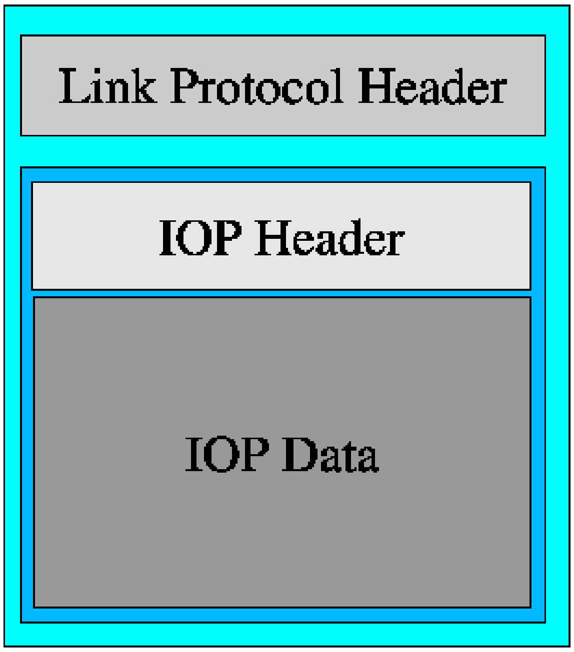

Figura 3.10: incapsulamento nei pacchetti di comunicazione di uno IOP

Si può anche pensare che possa esistere un segnale globale comandato dal PLD per l'ingresso a Kernel-Mode. Esso verrà quindi messo in OR con quello proveniente dalla rete e avrà sempre l'effetto di generare una interruzione; esso può servire per la gestione dei comandi di inizializzazione/debugging/arresto provenienti dall'host.

I pacchetti in arrivo da un nodo in modalità kernel potrebbero essere smistati verso una ulteriore coda di comunicazione, indirizzabile solo in Kernel-Mode. Questo evita che il traffico asincrono vada ad unirsi a quello sincrono, costringendo l'hardware a pulire la coda destinazione per non far giungere al TinyKernel informazioni che potrebbero riguardare la normale esecuzione di un programma utente e quindi non costituire uno IOP.

Il percorso, in una rete a cubo con condizioni al contorno periodiche come *apeNEXT*, è facile da determiare e può essere deciso in maniera locale dal kernel. Infatti esiste sempre il percorso migliore che può essere calcolato localmente; anzi in verità esistono più percorsi ottimi, e la scelta di quale di questi seguire può essere compiuta sulla base della sola ottimizzazione delle linee. Infatti per ognuna delle tre direzioni può essere presa una decisione: per schematizzare l'algoritmo di instradamento diamo alcune definizioni.

d(A, B) : distanza minima tra il nodo A e il nodo B

D(x, y, z) : diametro della rete lungo i tre assi

chiamando $A(x, y, z)$ l'indirizzo del nodo corrente e $B(x, y, z)$ quello del nodo destinazione l'algoritmo è descritto nella tabella 3.1.

$\forall$ direzione k in (x, y, z)
  se $A[k] = B[k]$ allora
    non ci muoviamo nella direzione k
  se $A[x] < B[x]$ allora
    se $d(A, B) < (D[k]/2)$
      ci muoviamo nella direzione k^+
    altrimenti
      ci muoviamo nella direzione k^-
  se $A[x] > B[x]$ allora
    se $d(A, B) < (D[k]/2)$
      ci muoviamo nella direzione k^-
    altrimenti
      ci muoviamo nella direzione k^+

Tabella 3.1: l'algoritmo di routing

Abbiamo detto che è stata anche prevista la possibilità del **source routing** grazie alla quale il nodo sorgente può specificare totalmente il cammino previsto dal pacchetto. Ciò però non si applica nelle normali esecuzioni di instradamento, in quanto l'effettivo utilizzo di tale tecnica si può riscontrare principalmente nei casi di debugging.

È possibile che ci si accorga che uno dei link inter-nodo non funziona come dovrebbe. In questo caso il TinyKernel viene richiamato per gestire una eccezione hardware oppure, ma più raramente, il nodo era già in Kernel-Mode. In questo caso il kernel deve essere in grado di saper scegliere il percorso da far effettuare al pacchetto ricevuto o da inviare sulla base della sua conoscenza di quali collegamenti siano in funzione. Se l'algoritmo di routing è in grado di gestire anche questa evenienza allora esso può essere considerato *affidabile*, cioè in grado di far giungere comunque il pacchetto a destinazione. L'unica eventualià in cui tale algoritmo fallirebbe riguarda la totale esclusione di una porzione della rete dal resto della macchina. Ciò può accadere, in un caso realistico, quando l'unità di comunicazione di un nodo risulti completamente danneggata; oppure, in una ipotesi più remota, quando tutti i collegamenti verso l'esterno di un insieme di nodi posti a formare un parallelepipedo fossero inutilizzabili. In una eventualità simile un meccanismo di segnalazione basato sui collegamenti inter-nodo sarebbe comunque fallace e si avrebbe bisogno della possibilità di fornire questi segnali con linee appropriate sulla PB. Tale possibilità non viene qui considerata.

Da notare che lo studio di affidabilità e di non-bloccanza del meccanismo di istradamento qui descritto risiede anche su alcune considerazioni effettuate sulla logica di comuicazione di *apeNEXT*. Per poter permettere questo tipo di comunicazioni è infatti necessario che il modulo di rete del nodo abbia le seguenti capacità:

- poter inviare in una direzione mentre riceve da una o più direzioni differenti
- implementare (come già specificato) i necessari meccanismi di affidabilità consistenti in timeout, ritrasmissioni e generazione di eccezioni come opportuno

- poter riconoscere lo stato di Kernel-Mode per inviare i pacchetti potendo anche indicare che si tratta di IOP

In questi aspetti non viene considerata la possibilità di poter ricevere ed inviare sulla stessa direzione, cioè avere dei canali *full-duplex*. In realtà nella programmazione SIMD questo non è assolutamente richiesto visto che se il programma indica di inviare dati in una direzione tutti i nodi invieranno solo in quella e mai verso l'opposta. Nelle comunicazioni Kernel-Mode questo può invece, in linea teorica, avvenire. Si potrebbe avere, ad esempio, la possibilità che un nodo invii nella direzione X^+ un normale pacchetto di system-call. Il nodo a lui contiguo in quella direzione, invece, avendo tutti gli altri link inutilizzabili a causa di malfunzionamenti, si vede costretto ad inviare proprio nella direzione X^-. Il risultato, allo stato attuale dei lavori, sarebbe non determinabile. Come ci si accorge, però, ciò rientra in un insieme di casi limite, oggetto di particolari eccezioni hardware; quindi è sufficiente istruire il modulo di comunicazione a generare le opportune interruzioni in queste evenienze. Il TinyKernel, quindi, interrompe tutte le proprie operazioni in caso di eccezione hardware, avendo come unica priorità far conoscere all'host le cause di tale errore. Un'attenta programmazione del TinyKernel, dunque, è in grado di risolvere anche questa eventualità.

3.4.4 Broadcast

In alcune occasioni i pacchetti in arrivo dagli host potrebbero rappresentare delle informazioni da smistare a tutti i nodi della rete; si pensi, ad esempio, alle informazioni di bootstrap. Deve quindi esistere la possibilità di effettuare **broadcasting** sulla rete (mentre il **multicasting**, cioè la comunicazione con un insieme ben specifico di nodi, non appare influente). Per specificare che un pacchetto deve subire un instradamento di questo tipo si è scelto di porre questa informazione nel campo *DstNodeAddress*. Un particolare pattern in esso (ad esempio 0xFFFFFF) potrebbe indicare al TinyKernel di effettuare più copie del pacchetto e smistarle in più direzioni. Per evitare duplicazioni di informazioni è stato adottato l'algoritmo descritto nella tabella 3.2 (dove C(x, y, z) é il Bridge Node).

Questo algoritmo smista seguendo un ordine: se il pacchetto giunge dalla direzione X allora deve continuare lungo tale direzione se non si è già raggiunta metà del percorso (per completare il diametro), in più deve essere spedito verso le direzioni Y e Z in entrambi i versi. Per la direzione Y il pacchetto deve continuare ed essere copiato su entrambi i versi di Z. Per la direzione Z esso deve solamente continuare fino coprire tutto il diametro.

Si è simulato che tale algoritmo è in grado di sfruttare l'architettura parallela ed effettuare la copertura completa di una macchina di dimensioni ($8\times8\times8$) in 12 passi e di una macchina di dimensioni ($16\times8\times8$) in 16 passi.

se il nodo attuale e' il Bridge Node allora
    invia in tutte le direzioni
se il pacchetto proviene dalla direzione X^- allora
    invia lungo le direzioni Y^+, Y^-, Z^+, Z^-
    se $|A[x] - C[x]| < D[x]/2$ allora
        invia lungo la direzione X^+
se il pacchetto proviene dalla direzione X^+ allora
    invia lungo le direzioni Y^+, Y^-, Z^+, Z^-
    se $|A[x] - C[x]| \leq D[x]/2$ allora
        invia lungo la direzione X^-
se il pacchetto proviene dalla direzione Y^- allora
    invia lungo le direzioni Z^+, Z^-
    se $|A[y] - C[y]| < D[y]/2$ allora
        invia lungo la direzione Y^+
se il pacchetto proviene dalla direzione Y^+ allora
    invia lungo le direzioni Z^+, Z^-
    se $|A[y] - C[y]| \leq D[y]/2$ allora
        invia lungo la direzione Y^-
se il pacchetto proviene dalla direzione Z^- allora
    se $|A[z] - C[z]| < D[z]/2$ allora
        invia lungo la direzione Z^+
se il pacchetto proviene dalla direzione Z^+ allora
    se $|A[z] - C[z]| \leq D[z]/2$ allora
        invia lungo la direzione Z^-

Tabella 3.2: l'algoritmo di broadcasting

Con *passo* si è definito l'invio dello IOP interessato ad un insieme di nodi e l'esecuzione locale dell'algoritmo specificato che, essendo formato da un numero di istruzioni finito senza cicli, si può considerare eseguibile in un tempo costante (*O(1)*).

Se prendiamo una macchina di dimensioni $(n \times m \times p)$ con topologia a toro in tutte le direzioni la distanza massima tra due nodi qualsiasi è $n + m + p$.

L'algortimo, quindi, nelle configurazioni provate, esegue la copertura totale della rete con un numero di passi uguale alla distanza massima tra due nodi.

3.4.5 Condizioni globali

Nella programmazione APE esistono alcuni concetti di programmazione che abbracciano totalmente il paradigma SIMD. Questi sono delle condizioni da verificare in un insieme di nodi. In particolar modo esistono due costrutti in grado di effettuare dei salti condizionati a seconda della condizione che viene indicata, effettuando il test su tutti i nodi della rete. Questi sono **IFANY** e **IFALL** che assumono valore *true* se, rispettivamente, la condizione indicata è verificata in almeno un nodo o in tutti i nodi. Tali costrutti vengono inseriti nei linguaggi di programmazione di alto livello presenti per la macchina; anche il compilatore C/C++ ha subito delle modifiche alla sintassi accettata per poter includere queste caratteristiche, assumendo una operatività più orientata alla programmazione SIMD.
Di seguito possiamo trovare un esempio di codice TAO che utilizza queste condizioni globali:

```
real a
    read a
    if all (a > 0)
       write a
```

In *APEmille* tali condizioni venivano propagate attraverso la rete da segnali globali che risiedevano su particolari linee della PB. Attraverso l'algoritmo delle *somme globali*, implementato in hardware sulle PB, ogni nodo avrebbe potuto conoscere il valore di queste condizioni globali.

Seguendo le linee guida che hanno portato alla definizione del kernel di *apeNEXT* come qui è descritto è necessario considerare attentamente l'utilizzo di queste caratteristiche per cercare di implementarle attraverso la normale esecuzione di codice.

Innanzitutto si parte dallo studio del codice scritto per le generazioni precedenti di APE, in particolare *APEmille*, per valutare la percentuale di utilizzo di queste condizioni. Un loro utilizzo diffuso costringerebbe comunque ad una ottimizzazione dei loro tempi di esecuzione e quindi ad una loro implementazione nella logica delle PB.

Nelle macchine precedenti l'hardware, essendo i nodi fortemente sincronizzati tra loro, era in grado di effettuare velocemente il test. In *apeNEXT*, con la sua sincronizzazione lasca, ciò comporterebbe più di un problema, introducendo un meccanismo di ritardo al momento di collezionare il valore della condizione in un insieme di nodi.

Se, però, questi costrutti venissero effettivamente sfruttate da relativamente pochi algoritmi codificati per i programmi di fisica, e per le altre applicazioni, delle macchine APE, allora si potrebbe pensare ad una implementazione della propagazione di questi segnali esclusivamente in software. L'algoritmo delle somme globali potrebbe venire codificato nei programmi utente (generando apposite librerie ottimizzate) sfruttando le normali comunicazioni inter-nodo e l'unità logica per effettuare operazioni di AND e OR sui valori da diffondere.

L'assenza di segnali globali per la gestione di queste condizioni rende la progettazione hardware della macchina estremamente più semplice, diminuendo in numero di linee da costruire e di pin sui connettori delle schede. Inoltre l'aggiornabilità software di questo genere di aspetti può portare ad ottimizzazioni e modifiche speciali per particolari programmi (si pensi ad un IFALL solo su un particolare sottoinsieme di nodi) semplicemente con un relativo sforzo programmativo.

3.5 Il TinyKernel

La programmazione del TinyKernel prevede numerosi passi:

1. la definizione di alto livello del Tinykernel Loop
2. l'implementazione del loop in linguaggio assembly
3. la creazione delle interfacce verso la programmazione utente

3.5.1 La definizione di alto livello

In questo stadio è stato utilizzato il linguaggio C per presentare un simulatore su singolo nodo del loop. Questa simulazione è servita per evidenziare alcuni aspetti macroscopici del problema e per svilupparne una soluzione ad un livello logico facilmente comprensibile.

Sono state implementate delle simulazioni delle unità del nodo accedute dal kernel, quali il Register File e la Memoria. Questo ha permesso un controllo più granulare su eventuali incongruenze nelle informazioni scritte.

In particolare risulta necessario non modificare in alcun modo locazioni di memoria che non siano dedicate al Kernel. Questo potrebbe inficiare sulla coerenza del programma utente una volta effettuato il ritorno ad esso. Allo stesso modo il Register File conterrà alcuni valori (scritti su registri dichiarati *fisici* dal programma utente) che in User-Mode possono avere dei significati particolari. Per questo l'intero Register File deve venire salvato su delle locazioni di memoria predisposte per questa operazione e opportunamente ripristinato al momento di tornare al programma utente.

Il comportamento generale del loop immaginato può essere riassunto nei passi seguenti:

```
1)  accesso da interruzioni:
2)      salvataggio del Register File in Memoria
3)      impostazione degli opportuni Configuration Register
4)      lettura del registro delle eccezioni
5)      se il registo non e' vuoto:
6)          creazione nel Register File di uno IOP che descriva l'eccezione avvenuta
7)          invio dello IOP creato
8)  accesso da system-call:
9)      salvataggio del Register File in Memoria
```

10) impostazione degli opportuni Configuration Register

11) lettura dell'indirizzo di memoria dove risiedono gli IOP creati in User-Mode

12) se esistono IOP in memoria:

13) lettura del primo header da memoria

14) creazione nel Register File di uno IOP completo (header + contenuto)

15) invio dello IOP creato

16) **ascolto della rete**:

17) lettura dei registri di stato della rete

18) se e' avvenuta la ricezione di uno IOP nella coda apposita:

19) lettura dello IOP dalla coda e salvataggio sul Register File

20) decodifica dell'header dello IOP

21) se il destinatario dello IOP e' il nodo corrente:

22) esecuzione del comando ed accesso alle unita' interessate

23) altrimenti:

24) individuazione del nodo contiguo piu' vicino a quello destinazione

25) invio dello IOP al nodo scelto

26) salto a 1.

Si possono individuare nei passaggi sopra elencati alcune componenti fondamentali che rappresenteranno dei sottoprogrammi sviluppati indipendentemente:

1. **salvataggio del register file in memoria**: questo verrà eseguito al-

l'accesso del TinyKernel qualunque sia il motivo della sua attivazione e deve essere sviluppato in modo da rappresentare il primo corpo di codice eseguito, per non rischiare di sporcare il Register File che si deve invece copiare

2. **impostazione dei Configuration Register**: in questo codice deve venire mascherato il registro delle eccezioni per evitare che all'interno del TinyKernel la sua esecuzione venga interrotta per l'attivazione di una interruzione. Il codice leggerà autonomamente l'avvenuto riconoscimento di una eccezione ed il suo tipo. Inoltre deve venire impostato correttamente il registro che indica al resto delle unità l'attivazione del Kernel-Mode, importante, ad esempio, per le corrette comunicazioni con i nodi vicini, come specificato in (3.4.3).

3. **invio di uno IOP**: questo dovrebbe leggere dal Register File l'header dello IOP calcolando il nodo contiguo migliore per effettuare l'instradamneto verso la destinazione. Di seguito l'operazione da compiere consiste nel comandare al modulo di comunicazione di effettuare l'invio attraverso le linee inter-nodo nella direzione prescelta. Ovviamente questa stessa porzione di codice deciderà se invocare il meccanismo di broadcasting, a seconda dell'indirizzo del nodo destinzione specificato

4. **esecuzione del comando descritto in uno IOP**: partendo dal coman-

do specificato viene acceduta la device interna interessata, in modalità di lettura o scrittura, e per il numero di parole appropriato

I test sul codice C prodotto, che ha permesso di creare dei veri e propri programmi con cui effettuare i controlli, sono tornati utili per verificare alcuni algoritmi importanti. Ad esempio si è potuta verificare la correttezza del codice di routing (in cui, come già accennato, la scelta è subordinata al corretto funzionamento dei link tra i nodi) o di quello di broadcasting, potendo anche fornire delle stime di prestazioni.

3.5.2 L'implementazione in assembly

Il passaggio alla scrittura di codice macchina vero e proprio ha rivelato numerosi aspetti altrimenti invisibili in un semplice ambiente di simulazione di alto livello.

La sintassi dell'assembly è risultata in continua evoluzione, durante la codifica degli algoritmi, tanto che al momento della stesura di questa tesi alcuni aspetti del linguaggio vengono ancora modificati (ad esempio l'ultimo cambiamento effettuato riguarda la sintassi di definizione delle slice). Questo ha causato numerose riletture del codice che, spesso, sono servite per costringere ad una verifica di quanto fatto; ma a volte hanno anche rappresentato un rallentamento nei lavori. In generale seguire questi cambiamenti è risultato necessario: ad esempio nelle definizioni preliminari del linguaggio non erano presenti riferimenti ai Configuration Register, oppure non era possibile comunicare con altri nodi se non attendendosi dati dal nodo opposto (come è auspicabile in una normale programmazione SIMD). Questi aspetti, come altri, hanno fatto sì che fosse necessario arrivare ad una maturazione del linguaggio tale da rappresentare il valido supporto alla stesura del TinyKernel.

Inoltre era necessario disporre di alcuni strumenti per poter effettuare i test previsti sul codice prodotto. In particolar modo si sono utilizzati il simulatore hardware VHDL (e per il funzionamento con esso il codice ha anche subito la fase di shaking) ed il simulatore funzionale **SF**. Quest'ultimo si è dimostrato di grande utilità essendo sempre aggiornato con le modifiche apportate al linguaggio assembly e avendo la capacità di essere modificato, in quanto scritto in C++, per fornire segnali di maggiore interesse per il caso specifico. Risulta quindi utile analizzare con maggiore interesse questo strumento.

Il Simulatore Funzionale è composto da un insieme di classi C++ ognuna delle quali descrive una unità interna al nodo di calcolo. Queste sono in grado di essere comandate da un decodificicatore di istruzione che prende come input un file di tipo .NO (Next Object) rilocabile o non-rilocabile. Dal file oggetto SF estrae sia le istruzioni sia i dati inizializzati (utilizzando le apeAPI); con essi costruisce una memoria dati di partenza ed una cache di istruzioni, entrambe virtualmente infinite, avviando poi un ciclo che comandi le varie unità ad eseguire le istruzioni specificate. SF è poi in grado di eseguire, ad ogni passo, un debug interno dell'unità interessata dall'istruzione, fornendo all'utente le necessarie informazioni sull'esecuzione del programma. SF offre un supporto completo per i registri virtuali (cosa che diverrà in seguito molto importante) e per la gestione delle comunicazioni inter-nodo.

Grazie alla presenza di SF e di NAS è stato possibile scrivere codice assembly, assemblarlo, linkarlo ed eseguirlo in un ambiente di test protetto.

Si è partiti da una *traduzione* del codice C scritto precedentemente nel linguaggio assembly di *apeNEXT*, individuando da subito le distinzioni fatte precedentemente, aggiungendone delle nuove e quindi suddividendo il codice nelle porzioni qui di seguito indicate:

- **TinyKernel-Loop**: il ciclo del Tinykernel in grado di invocare gli altri moduli o di restituire il controllo al programma utente
- **TinyKernel-ManageIOP**: è in grado di leggere gli header creati in User-Mode per generare degli IOP da instradare
- **TinyKernel-ManageTrap**: controlla il registri delle eccezioni e si comporta di conseguenza
- **TinyKernel-DecodeIOP**: legge gli IOP ricevuti dalla rete ed esegue i comandi specificati
- **TinyKernel-ListenNet**: ascolta la rete individuando la ricezione di nuovi IOP
- **TinyKernel-Route**: decide il nodo giusto su cui istradare uno IOP che deve venire inviato

- **TinyKernel-Broadcast**: esegue l'algoritmo di broadcasting su uno IOP

Ognuno di questi moduli risiede in un file sorgente separato e rappresenta una routine da poter invocare simbolicamente secondo l'etichetta associata.

Assieme a questi file sono stati anche scritti dei sorgenti importanti per il pre-processing:

- **TinyKernel_consts.hs**: contiene tutte le costanti referenziate simbolicamente nel codice, come la grandezza di alcuni tipi di dato o valori enumerativi

- **TinyKernel_defs.hs**: contiene valori costanti che indicano il numero ed il tipo di registri utilizzati e permettono di referenziarli simbolicamente

- **TinyKernel_macros.hs**: consiste in alcune macro del preprocessore definite per eseguire alcuni compiti semplici ma utilizzati di frequente. La loro definizione in questo file permette di avere nei sorgenti un codice più pulito e di più facile aggiornamento

- **Nasm_macros.hs**: un insieme di macro del preprocessore di utilità generale che sono state rese disponibili agli altri programmatori assembler di *apeNEXT*. Riguardano la gestione dello stack delle condizioni, i salti a subroutine e alcune operazioni logiche

- **IOP_macros.hs**: altre macro che definiscono i metodi di accesso in lettura/scrittura di uno IOP residente su Register File

3.5.3 L'interfaccia verso l'utente

Bisogna ora definire le procedure di attivazione del TinyKernel dai programmi utente, cioè il meccanismo di transizione User-Mode→Kernel-Mode.

La richiesta di chiamate al sistema operativo deve, come già specificato, creare l'header dello IOP che specifica il comando da compiere, quindi invocare la transizione a Kernel-Mode. Quest'ultima si effettua semplicemente con un *branch* (salto) incondizionato ad una specifica locazione di memoria predefinita. Questo indirizzo verrebbe specificato in qualche file header da includere nei propri programmi. La locazione verso cui si compie il salto è quella indicata precedentemente **accesso da system-call**, internamente al loop del TinyKernel.

Il TinyKernel è in grado di leggere uno o più header da una locazione di memoria specificabile attraverso la modifica di un registro fisico (o un configuration register, se nel futuro verrà considerato una opzione migliore). Deve essere quindi il programma utente a decidere quali locazioni di memoria riservare per lo scambio di dati con il kernel, dipendentemente dal numero di header che si vogliono creare. Con ogni probabilità saranno i compilatori a gestire questo tipo di scelta, visto che i sistemi operativi delle macchine della famiglia APE non implementano nessun meccanismo di allocazione dinamica della memoria, non essendocene necessità: infatti l'assenza di più processi concorrenti pone l'intera memoria del nodo a disposizione dell'unico programma utente in esecuzione, ad eccezione di eventuali zone protette ad uso del sistema.

Le invocazioni del sistema vengono codificate in subroutine da richiamare attraverso delle macro del preprocessore. L'utilizzo di macro permette di poter scrivere nel codice assembly del programma istruzioni molto comprensibili, del tipo:

```
WriteWords    mem     size    res
```

(questo comando scrive *size* parole sulla risorsa *res* leggendole dalla locazione di memoria *mem*)
Come si può vedere la sua invocazione sembra quella di una normale funzione in un linguaggio ad alto livello, benché, in realtà, le operazioni compiute da questa macro siano:

1. scrittura dei tre operandi specificati su locazioni di memoria predefinite e allocate ad uso del sistema

2. invocazione di una subroutine che, leggendo i tre operandi dalla memoria, crea l'header specificato su register file

3. esecuzione della routine di frammentazione nel caso il numero di parole da inviare sia maggiore della grandezza del campo *Contents* dello IOP

4. scrittura dell'header in memoria

Ovviamente la maggior parte dei parametri di una macro di sistema sono dei registri virtuali, in cui viene specificato il valore assoluto (per valori che rappresentano dati) o l'indirizzo (nel caso di locazioni di memoria) dell'operando. Ma i parametri possono anche essere dei valori differenti, come in `PutString` che accetta come operando una stringa di caratteri racchiusa tra virgolette o apici per inviarla in scrittura allo schermo dell'host.

Questi sono i file sorgente in cui vengono definite le chiamate a sistema:

- **User_header.hs**: il file dove sono definite le macro che individuano le chiamate a sistema
- **User_macros.hs**: contiene il corpo di ogni chiamata a sistema e delle routine accessorie, come quella di frammentazione
- **User_subs.ns**: la definizione delle subroutine previste che richiamano le macro definite nel file precedente

Si prevede che un programma utente debba includere nel proprio codice assembly solo il file **User_header.hs** per poter assemblare il proprio programma. Poi i file oggetto generati devono venire *linkati* con quello proveniente dal file **User_subs.ns** (che include **User_macros.hs**), per avere i riferimenti simbolici alle subroutine invocate. Il linker NLD verrà modificato per includere il più aggiornato file oggetto con le subroutine di sistema, evitando questo lavoro al programmatore.

Dalla scrittura delle procedure utente è emerso che esse necessitano di alcuni registri dichiarati fisici. Infatti appena lo shaker incontra una label esso azzera i puntatori dei registri virtuali ricominciando il mapping da zero. Se un registro viene dichiarato *fisico* in una sezione di codice ogni riferimento ad esso non verrà ri-mappato con altri. È però impossibile prevedere se l'utente utilizzerà o meno alcuni registri come fisici perché ne ha necessità nel prorprio codice. Quindi se il sistema operativo necessita di alcuni registri fisici per poter funzionare allora essi devono venire definiti a priori e deve essere impossibile il loro referenziamento dai programmi utente; devono divenire, cioè, dei *registri di sistema*. È chiara quindi l'importanza di minimizzare il numero di registri di sistema con ua attenta analisi del codice prodotto, al fine di ottimizzare l'utilizzo del Register File. Dopo varie riletture del codice sono stati individuati solo 7 registri da dichiarare *di sistema*, che, contro i 512 presenti, può essere un compromesso accettabile.

Di seguito presentiamo un esempio di file sorgente in assembly in cui vengono invocate delle chiamate a sistema per aprire un file sull'host:

```
.include "user_header.hs"

.globl user_test
.globl _start
_start:
.text
user_test:

        .data
        stringa:        .STRING         "nomeFile.txt"
        mode:           .INTEGER        0x1
```

```
        .text

!! vari simboli che rappresentano registri
filename        .assign         9
modo            .assign         10
handle          .assign         11
temp            .assign         13

!! puntatore al modo di apertura
        ATOR.L          modo            mode
!! puntatore all'handle del file
        ATOR.L          handle          %0x350
!! valore proposto dell'handle
        ATOR.L          temp            %7
        RTOM            %0(handle)      temp
!! puntatore al nome dle file
        ATOR.L          filename        stringa

!! dove scrivere gli header
        set_IOP_address                 %0x1010

        PutString       "Apro il file!\n"  0

        OpenFile        filename        14      handle  modo

!! salto a TinyKernel
        send_made_IOPs

        HALT %0
```

In questo codice le chiamate utilizzate sono:

- **PutString**: per scrivere su schermo la stringa specificata
- **OpenFile**: per aprire il file specificato con una particolare modalità (lettura, scrittura, etc.) ed indicando l'handler proposto

mentre le seguenti sono delle macro aggiuntive:

- **set_IOP_address**: per indicare la locazione di memoria dove verranno posti gli header
- **send_made_IOPs**: per entrare in Kernel-Mode

Conclusioni

Il lavoro qui esposto è caratterizzato da tre fasi principali. La prima riguarda l'approfondimento delle tematiche trattate nell'ambito della progettazione di un elaboratore parallelo specifico per la *computational physics*. La seconda la creazione di un assemblatore *open-source* per *apeNEXT*. La terza l'ideazione di una ipotesi di sistema operativo distribuito partendo dalla definizione delle funzionalità del singolo nodo di calcolo dell'elaboratore.

In tutte queste fasi l'esperienza effettuata ha portato a notevoli traguardi, permettendo anche la realizzazione di quanto proposto.

In particolare l'assemblatore ha subìto nel tempo una maturazione tale da renderlo lo strumento ufficiale per lo sviluppo in assembly utente per la nuova macchina. Esso ha dimostrato una efficienza ed una adattabilità alle continue modifiche della macchina tali da renderlo immediatamente utilizzabile. Con esso sono state già provate alcune routine fondamentali per gli scopi prefissi da *apeNEXT* (come l'*operatore di Dirac inverso*) oltre al codice del Sistema Operativo qui descritto. Inoltre è già avviata la scrittura delle prime versioni dei compilatori per i linguaggi previsti per la macchina ed essi producono

codice nel linguaggio dell'assemblatore.
La creazione delle apeAPI ha, poi, permesso la facile generazione di una serie di strumenti necessari al progetto, come il traduttore **MASM2NO** (che scrive un file oggetto partendo da istruzioni di micro-assembly), attualmente in fase di completamento.

Per quel che riguarda l'ipotesi di Sistema Operativo il presente lavoro ha dimostrato che determinati requisiti prefissi sono effettivamente raggiungibili partendo dalla definizione attuale della macchina. In particolare si è riuscita a dare una implementazione funzionante della parte più critica (quella residente su nodo) mantenendo esigue le richieste in fatto di aggiunte hardware necessarie al suo supporto. Si è inoltre resa estremamente adattabile la configurazione generale della macchina, in fatto di collegamenti ed interazioni verso l'esterno, e per quel che riguarda il suo avvicinarsi al paradigma MIMD.

Il lavoro qui descritto rappresenta, quindi, una possibile base per una definizione più accurata del futuro Sistema Operativo di *apeNEXT*. Ha infatti interessato sia l'aspetto hardware che quello software, generando una serie di ipotesi architetturali e di sottoprogrammi che potrebbero tornare utili nella stesura finale del codice.

Appendici

Appendice A

GAS e apeAPI

A.1 Porting del GAS

Il codice più significativo per indicare alcune sclete effettuate durante il porting dell'assemblatore GAS per *apeNEXT* è quello incluso nel file *tc-apenext.h*. In esso sono indicati alcuni parametri dell'assemblatore e l'insieme delle routine *machine-dependent* che sono state riscritte per il pieno supporto alla nuova macchina.

```
#ifndef TC_APENEXT
#define TC_APENEXT 1

/* Little endian */
#define TARGET_BYTES_BIG_ENDIAN 0

#ifndef BFD_ASSEMBLER
#error BFD needed
#endif

#define tc_fix_adjustable(X) true
```

```
#define TARGET_ARCH bfd_arch_apenext

#ifdef OBJ_ELF
#define TARGET_FORMAT "elf64-apenext"
#endif

#define NO_RELOC BFD_RELOC_NONE

#include "apenof.h"

#define LOCAL_LABELS_FB 1

#define tc_headers_hook(a) {;} /* not used */

extern const char extra_symbol_chars[];
#define tc_symbol_chars extra_symbol_chars

const char comment_chars[] = "#/!";

const char line_comment_chars[] = "!/";

const char line_separator_chars[] = "";

const char EXP_CHARS[] = "eE";

const char FLT_CHARS[] = "fFdDxX";


/* called before assembler exit */
void APEN_cleanup();
```

```
#define md_cleanup() APEN_cleanup ()

/* called for each label defined */
void APEN_frob_label(symbolS*);
#define tc_frob_label(s) APEN_frob_label(s)

/* called while writing a section in object file */
void APEN_frob_section(asection*);
#define tc_frob_section(x) APEN_frob_section (x)

/* manages initialized strings */
void APEN_stringer (int);

/* manages initialized floating point numbers */
void s_float_cons (register int float_type);

/* main function parsing an instruction line */
void md_assemble (char*);

/* called when opening an input file */
void md_begin ();

/* array of pseudo-ops identifiers
   and  respective managing functions */
const pseudo_typeS md_pseudo_table[] =
{
  {"regset",s_regset,0},
  {"ascii", APEN_stringer, 0},
  {"asciz", APEN_stringer, 1},
  {"string", APEN_stringer, 1},
  {"slice",s_slice,0},
```

```
  {"integer",s_integer,0},
  {"vinteger",s_integer,1},
  {"real",s_real,0},
  {"vreal",s_vreal,0},
  {"complex",s_complex,0},
  {"int" ,s_bad,0},
  {"byte",s_bad,0},
  {"word",s_bad,0},
  {"long",s_bad,0},
  {"quad",s_bad,0},
  {"octa",s_bad,0},
  {0,0,0}
};

#define md_number_to_chars number_to_chars_littleendian

#endif /* TC_APENEXT */
```

A.2 apeAPI

Di seguito viene riportato il file header C `apeAPI.h`. In esso sono definite le strutture dati e le funzioni delle apeAPI per accedere ai file oggetto di *apeNEXT*.

```
typedef struct apelf *APELF;

/* apelf_const
It's a representation of an APEnext constant.
```

```
fields:
value:          a pointer to a 128 bit buffer with
                constant value
address:        the address where constant is
                allocated in object file
                      Note: only after all fixing (i.e. after
             final linking) this address is meaningful
slicevalue_p : a pointer to 6 32-bit word with slice
                value for constant in format:
                (X,Y,Z)(X,Y,Z)
*/

struct apelf_const {
  void *value;
  unsigned long address;
  long *slicevalue_p;
};


/* apelf_instr
It's a representation of an APEnect macrocode instruction.

fields:
header_p:     pointer to an ins_header_t type data.
operand_p:    array of header_p->num_ops operands_t
              type elements
*/

struct apelf_instr {
  ins_header_t *header_p;
```

```
  operands_t  *operand_p;
};


/***************** F U N C T I O N S *****************/


/* apelf_open
Opens an object file (does NOT open core files and
archive files).
It returns an APELF object, with sections .text and .data
already loaded to access with apelf_read_xxxxx
It allocates all needed space
*/

APELF apelf_open (const char*);


/* apelf_close
Closes an opened APELF object and frees all allocated memory
*/

void apelf_close (APELF);


/* apelf_read_const
Reads from an APELF object the n.th constant previously
loaded from a data section
It allocates a pointer to apelf_const structure and returns
0 if n.th constant isn't present
*/
```

```
int apelf_read_const (APELF, struct apelf_const **, int);

/* apelf_read_const_to_char
acts like apelf_read_const but translates result to a
canonical string
*/

int apelf_read_const_to_char (APELF , char **, int);


/* apelf_read_instr
Reads from an APELF object the n.th instruction previously
loaded from a text section
It allocates a pointer to apelf_inst structure and returns
0 if n.th instruction isn't present
NOTE: the allocated pointer is maintained until next call to
      function or until an apeLf_close() call
*/

int apelf_read_instr (APELF, struct apelf_instr **, int);


/* apelf_read_instr_to_char
acts like apelf_read_instr but translates result to a
canonical string
*/

int apelf_read_instr_to_char (APELF, char **, int);

/* apelf_read_instr__and_asmline_to_char
```

```
Reads from an APELF object the n.th instruction previously
loaded from a text section and converts it to a string
representation.
Moreover take the line number of the nasm file.
*/

int apelf_read_instr_and_asmline_to_char (APELF, char **,
                                                 int, int*);


/* apelf_load_const_from_section
Reads from APELF object the section with the given name
and loads from it all constants. These constants can
later be read via apelf_read_const() function
*/

int apelf_load_const_from_section(APELF, const char*);


/* apelf_load_instr_from_section
Reads from APELF object the section with the given name
and loads from it all macro instructions. These instructions
can later be read via apelf_read_instr() function
*/

int apelf_load_instr_from_section(APELF, const char*);


/* apelf_load_instr_from_buffer
Reads from buffer with specified size and loads from it all
macro instructions. These instructions can later be read
```

```
via apelf_read_instr() function
*/

int apelf_load_instr_from_buffer(APELF, const char*, int);


/* apelf_get_section_contents_by_name
Returns a pointer to all contents of the section with given name
*/

unsigned char *apelf_get_section_contents_by_name (APELF,
                                                   const char*);


/* apelf_get_section_size_by_name

   Returns the size (in bytes) of section

*/

int apelf_get_section_size_by_name (APELF, const char *);


/* apelf_get_num_const
Returns the number of constants loaded from section via
apelf_load_const_from_section()
*/

int apelf_get_num_const(APELF);
```

```
/* apelf_get_num_instr
Returns the number of macro instructions loaded from section via
apelf_load_instr_from_section()
*/

int apelf_get_num_instr(APELF);


/* apelf_get_section_names
Returns an array of strings with the names of all sections
in APELF
*/

const char **apelf_get_section_names (APELF);


/* apelf_get_section_count
Returns the number of sections in APELF object
*/

int apelf_get_section_count (APELF);


/* apelf_get_section_start_address
   Returns starting address of given section
*/

unsigned long int apelf_get_section_start_address(APELF,
                                                  const char*);
```

```
/* apelf_open_to_write
Opens an object file to write on it
*/

APELF apelf_open_to_write(const char*);


/* apelf_add_constant
Writes a constant in an object file
*/

int apelf_add_constant(APELF, struct apelf_const *);


/* apelf_add_instruction
Writes an instruction in an object file
*/

int apelf_add_constant(APELF, struct apelf_instr *);


/* apelf_write
Flushes writes to destination object file
*/

void apelf_write(APELF);
```

Appendice B

Il Linguaggio Assembly

B.1 Assembly Utente

Di seguito viene riportata la documentazione ufficiale riguardo la sintassi del linguaggio assembly di livello utente, cioè quello accettato in input dal **NAS** (Next ASsembler).

```
============================================================
NASM: APEnext User Level Assembly

This is a user-level assembly language for a machine with 1 FILU,
no static memory, no addressable queues memory interface.
Register file and memory addresses refer to 128 bit locations,
immediate operands are 64 bit wide.

============================================================
Symbols:

bq: bank qualifier V=vector, H=high, L=low
IMM: 64 bit immediate value (prefixed with %)
```

```
DISP32: 32 bit immediate value (prefixed with %)
DISP64: IMM | SYMBOL | SYMBOL + IMM
REGSET: R:LEN, [Ro ... Rn] or REGSET as defined by a
                PRAGMA_REGSET [Ro ... Rn]
R0 ... Rn: blank separated sequence of REGSET_ELEMs
REGSET_ELEM:    int | int.bq
mem[MOP]: content of the (local or remote) memory location
                addressed by MOP
rem(mem_addr): remote part of the memory address
loc(mem_addr): local  part of the memory address
@: AGU accumulator (aka feedback)
d: DISP32
D: DISP64
============================================================
Addressing modes:

The general form of a MOP is:

    X(Y,Z,W)

the resulting expression is: X + Y + Z MULA W.

There are all 20 different MOPs:

#     MOP  MOP name

1)     d  d
2*)    D  D
3)     (R0)  R
4)     (@)  @
```

```
5*)   D(R0)  DR
6*)   D(@)  D@
7)    (R0, R1)  RR
8)    (@,  R0)  @R
9)    (, R0, R1)  _RR
10)   (, R0, d )  _Rd

11*)  D(R0, R1)  DRR
12*)  D(@,  R0)  D@R
13*)  D(, R0, R1)  D_RR
14*)  D(, R0, d)  D_Rd
15)   (R0, R1, R2)  RRR
16)   (R0, R1, d)  RRd
17)   (@,  R0, d)  @dR

18*)  D(R0, R1, R2)  DRRR
19*)  D(R0, R1, d)  DRRd
20*)  D(@,  R0, d)  D@Rd


*
 These MOPs are the only allowed in data transfer instructions
 (MTOR, LMTOR, MTOQ, RTOQ, RTOM).
 Other constraints on MOP usage as instructions operands are
 explicitly written in the instruction entry.

============================================================

Mnemonic     op0    op1    op2    op3     semantic

CNORM_PP     R0     R1[~]  R2[~]  R3[~]   R0=+(R1[~]*R2[~]+R3[~])
```

```
CNORM_PM      R0    R1[~]  R2[~]  R3[~]   R0=+(R1[~]*R2[~]-R3[~])
CNORM_MP      R0    R1[~]  R2[~]  R3[~]   R0=-(R1[~]*R2[~]+R3[~])
CNORM_MM      R0    R1[~]  R2[~]  R3[~]   R0=-(R1[~]*R2[~]-R3[~])

FNORM_PP.bq   R0    R1     R2     R3
FNORM_PM.bq   R0    R1     R2     R3
FNORM_MP.bq   R0    R1     R2     R3
FNORM_MM.bq   R0    R1     R2     R3
FNORM_AP.bq   R0    R1     R2     R3      R0= |R1[~]*R2[~]+R3[~]|
FNORM_AM.bq   R0    R1     R2     R3      R0= |R1[~]*R2[~]-R3[~]|

IADD.bq       R0    R1     R2             R0=R1+R2
ISUB.bq       R0    R1     R2
IMUL.bq       R0    R1     R2
IMULA.bq      R0    R1     R2
IDIV.bq       R0    R1     R2     R3      R0=R2/R3; R1=R2%R3;
IAND.bq       R0    R1     R2
INAND.bq      R0    R1     R2
IOR.bq        R0    R1     R2
IXOR.bq       R0    R1     R2
ILSH.bq       R0    R1     R2
IASH.bq       R0    R1     R2
IROT.bq       R0    R1     R2
IBITCOUNT.bq  R0    R1
ILEADONE.bq   R0    R1

FTOI.bq       R0    R1                    R0=cast_to_integer(R1)
ITOF.bq       R0    R1                    R0=cast_to_float(R1)

FLUTLSB.bq    R0    R1
FLUTINV.bq    R0    R1
```

```
FLUTINVSQR.bq R0     R1
FLUTEXP.bq    R0     R1
FLUTLOGE.bq   R0     R1
FLUTLOGM.bq   R0     R1
FLUTMOVE.bq   R0     R1     R2             R0.lo=R1.lo, R0.hi=R2.hi
FLUTCROSS.bq  R0     R1     R2             R0.lo=R1.hi, R0.hi=R2.lo

IPUSHEQ.bq    R0     R1
IPUSHNE.bq    R0     R1
IPUSHGE.bq    R0     R1
IPUSHGT.bq    R0     R1
IPUSHLE.bq    R0     R1
IPUSHLT.bq    R0     R1

FPUSHEQ.bq    R0     R1
FPUSHNE.bq    R0     R1
FPUSHGE.bq    R0     R1
FPUSHGT.bq    R0     R1
FPUSHLE.bq    R0     R1
FPUSHLT.bq    R0     R1

CPUSHEQ       R0     R1
CPUSHNE       R0     R1

SPOP
SNOT
SAND
SOR
SXOR
SANDBIS
SREPUSH
```

```
SSWAP
SRESET


Mnemonic     op0        op1        semantic

MTOR         REGSET     MOP*       REGSET<=mem[MOP]
LMTOR        REGSET     MOP*       REGSET<=mem[loc(MOP)]
MTOQ         MOP*       [: LEN]    Q[rem(MOP)]<=mem[MOP]:LEN
RTOQ         MOP*       REGSET     Q[rem(MOP)]<=REGSET
QTOR         REGSET                REGSET<=Q[TheRightOne]
RTOM         MOP*       REGSET     mem[loc(MOP)]<=REGSET
ATOR.bq      R0         MOP        R0<=MOP
PCTOR.bq     R0

LDA          MOP                   @<=MOP
JUMP         MOP
JUMPIF       MOP
JUMPIFNOT    MOP
GOSUB        R0         MOP

ANY
NONE

SEX                                software exception

HALT         DISP32
============================================================
```

Appendice C

Il TinyKernel

C.1 Il loop

Il loop del TinyKernel è il “cuore” della sezione di sistema operativo residente su nodo. Esso si occupa di inviare gli IOP creati, di riceverne ed instradarne altri verso i nodi remoti e di comprendere eventuali stati di eccezione del processore.

Qui di seguito si può trovare il codice assembly del loop con i riferimenti a tutti i sottoprogrammi che vengono in esso richiamati per la gestione delle varie fasi del ciclo.

```
.include "tinykernel_macros.hs"

.macro saveregs
   RTOM            %RF_IMAGE          0:512
   IOR.V           SAVED_IMAGE        ZERO           ONE
.endm
```

```
!! entry-point for exceptions

.globl loop_begin
loop_begin:

   saveregs

!! loading configuration registers

   GOSUB        returnaddress_loop    loop_load_conf_regs

iops_num     .equ   100
   LMTOR            iops_num          %0(IOPS_START)


!! if we have no IOP and no Exception
!! then execute boot code

   IPUSHEQ.L        TRAP_STATUS       ZERO
   IPUSHEQ.L        iops_num          ZERO
   SAND
   jumpif_to_subroutine    returnaddress_loop        boot
   JUMP             .Lloop_start


!! entry-point for System-Service

.globl loop_from_node
loop_from_node:
   saveregs
```

```
.Lloop_start:

   GOSUB        returnaddress_loop    loop_load_conf_regs

!! check for an execption and handle it

   IPUSHNE.L        TRAP_STATUS       ZERO
   jumpif_to_subroutine  returnaddress_loop   manage_trap

.L01:

!! if something arrives from near nodes then listen for them

   IPUSHNE.L        FIFO_STATUS       ZERO
   jumpif_to_subroutine   returnaddress_loop  listen_net

.L02:

!! load number of ready IOP headers to send

   LMTOR            iops_num          %0(IOPS_START)

!! manage first IOP heder in memory

   IPUSHNE.L        iops_num          ZERO
   jumpifnot_to_label                 .L1
   IADD.L           FIRST_PARAM       IOPS_START       ONE
   GOSUB            returnaddress_loop        manage_iop
   ISUB.L           iops_num          iops_num         ONE
   IADD.L           IOPS_START        IOPS_START       FIRST_PARAM
   RTOM             %0(IOPS_START)    iops_num
```

```
.L1:

!! loop

   JUMP              .Lloop_start
```

C.2 Simboli costanti

Il file `tinykernel_const.hs`, qui riportato, contiene la defiizione di numerosi simboli dal valore costante. Essi rappresentano i tipi di dato enumerativi utilizzati (ad esempio per la specifica dell'eccezione avvenuta) oppure semplici valori gobali definiti per vari scopi (come il valore per l'indirizzo di broadcast). Come si può vedere in tutto il codice del TinyKernel viene fatto largo utilizzo del preprocessore NASP, utile in particolare per esigenze come queste, cioè per la definizione di simboli da espandere prima dell'assemblamento.

```
!! Assi
X               .assign 0
Y               .assign 1
Z               .assign 2


!! Device
Dev_Cache       .assign 1
Dev_DMemory     .assign 2
```

```
Dev_ConfRegs    .assign 3
Dev_RegFile     .assign 4


!! Risorse predefinite
StdIn           .assign 0xF0
StdOut          .assign 0xF1
StdErr          .assign 0xF2


!! Comandi
Nack            .assign 0
Ack             .assign 1
Alive           .assign 2
Ready           .assign 3
BootIOP         .assign 4
Start           .assign 5
KernelModeStart .assign 6
Write           .assign 7
Read            .assign 8
Open            .assign 9
Close           .assign 10
RoutingErrorPkt .assign 11
FiluErrorPkt    .assign 12
LinkErrorPkt    .assign 13
SexPkt          .assign 14


|| Indirizzodi broadcast
BroadcastAddress .assign 0xFFFFFF
```

```
!! Grandezza dell'header di uno IOP in termini di vettori (128 bit)
IOP_HEADER_SIZE .assign 4

!! Grandezza dell'header di uno IOP in termini di vettori (128 bit)
CONTENTS_SIZE   .assign 64

!! Grandezza di uno IOP completo
IOP_SIZE        .assign IOP_HEADER_SIZE + CONTENTS_SIZE


!! Lista delle FIFO
LOCAL           .assign 0
XPLUS           .assign 1
XMINUS          .assign 2
YPLUS           .assign 3
YMINUS          .assign 4
ZPLUS           .assign 5
ZMINUS          .assign 6
HOST            .assign 7
ALL_DIRS        .assign 255



!! Eccezioni
None            .assign 0
FiluError       .assign 1
Sex             .assign 2
NetAlert        .assign 4
RoutingError    .assign 8
LinkError       .assign 16
```

C.3 L'interfaccia verso l'utente

Qui sono riportate tutte le chiamate al Sistema Operativo possibili dai programmi utente. La maggior parte di queste creerà degli header di IOP in memoria per farli poi processare ed inviare una volta entrati nel Tinykernel, e rappresentano le system-call possibili.

Come si può notare tutte le chiamate sono "racchiuse" in delle macro che basta richiamare simbolicamente per eseguire.

```
RETURN_ADD      .assign         421


!! reserved memory address
PARAMETER1      .assign         10000
PARAMETER2      .assign         10001
PARAMETER3      .assign         10002
PARAMETER4      .assign         10003
PARAMETER5      .assign         10004


!! virtual registers
temp1           .assign         1000
temp2           .assign         1001
temp3           .assign         1002
temp4           .assign         1003
temp5           .assign         1004
```

```
!! macro used later to load parameters from memory
.MACRO load_params      P1 P2 P3 P4 P5  NUM
.AIF    \NUM    GT      0
        RTOM    %PARAMETER1     \P1
.AIF    \NUM    GT      1
        RTOM    %PARAMETER2     \P2
.AIF    \NUM    GT      2
        RTOM    %PARAMETER3     \P3
.AIF    \NUM    GT      3
        RTOM    %PARAMETER4     \P4
.AIF    \NUM    GT      4
        RTOM    %PARAMETER5     \P5
.AENDI
.AENDI
.AENDI
.AENDI
.AENDI
.ENDM


!! System-calls
.MACRO  ReadFromFile MEM,       FILE,   OFFSET,     LEN
        load_params  \MEM       \FILE   \OFFSET     \LEN    0    NUM=4
        GOSUB        RETURN_ADD sub_read_from_file
.ENDM



.MACRO  WriteToFile  MEM,       FILE,   OFFSET,     LEN
        load_params  \MEM       \FILE   \OFFSET     \LEN    0    NUM=4
```

```
        GOSUB        RETURN_ADD sub_read_from_file
.ENDM


.MACRO  Getc         MEM,       RES,    ADDR=ZERO
!!      MEM:    Register
!!      RES:    Register: Resource Id (if 0: StdIn)
!!      ADDR:   Register: Optional Resource Address
        load_params  \MEM       \RES    \ADDR       0       0    NUM=3
        GOSUB        RETURN_ADD sub_getc
.ENDM


.MACRO  Putc         MEM,       RES,    ADDR=ZERO
!!      MEM:    Register
!!      DEV:    Register: Resource Id (if 0: StdOut)
!!      ADDR:   Register: Optional Resource Address
        load_params  \MEM       \RES    \ADDR       0       0    NUM=3
        GOSUB        RETURN_ADD sub_putc
.ENDM


.MACRO  Gets         MEM,       SIZE,   RES,        ADDR=ZERO
!!      MEM:    Register
!!      SIZE:   Register with size of string (-1 for unlimited)
!!      RES:    Register: Resource Id (if 0: StdIn)
!!      ADDR:   Register: Optional Resource Address
        load_params  \MEM       \SIZE   \RES        \ADDR   0    NUM=4
        GOSUB        RETURN_ADD sub_gets
.ENDM
```

```
.MACRO  Puts         MEM,       SIZE,   RES,        ADDR=ZERO
!!      MEM:    Register
!!      SIZE:   Register with size of string
!!      RES:    Register: Resource Id (if 0: StdIn)
!!      ADDR:   Register: Optional Resource Address
        load_params  \MEM       \SIZE   \RES        \ADDR   0    NUM=4
        GOSUB        RETURN_ADD sub_puts
.ENDM


.MACRO  PutString    STRING,    RES,    ADDR=ZERO
!!      STRING: String
!!      RES:    Register: Resource Id (if 0: StdIn)
!!      ADDR:   Register: Optional Resource Address
        .data
        String\@:    .STRING    "\STRING"
        .text
length  .assigna     .LEN("\STRING") + 1
        ATOR.L       temp1      String\@
        ATOR.L       temp2      %\&length
        Puts         temp1      temp2   \RES        \ADDR
.ENDM


.MACRO  OpenFile     FILENAME,  FILENAMELEN,   RESID,    MODE
!!      FILENAME: Register=Pointer for file name (often a symbol)
!!      FILENAMELEN: Length of filename string
!!      RESID:    Register=Suggested Resource ID
!!      MODE:   Register=Mode code to open file
        load_params  \FILENAME  \FILENAMELEN  \RESID  \MODE  0  NUM=4
```

```
        GOSUB      RETURN_ADD    sub_open_file
.ENDM


.MACRO  CloseFile    RESID
!! RESID:    Register=Resource ID
        load_params  \DEVID   0   0   0   0      NUM=1
        GOSUB        RETURN_ADD sub_close_file
.ENDM


.MACRO  ReadWords    MEM,        SIZE,   RES,     ADDR=ZERO
!!      MEM:    Register
!!      SIZE:   Register with size of data
!!      RES:    Register: Resource Id (if 0: StdIn)
!!      ADDR:   Register: Optional Resource Address
        load_params  \MEM       \SIZE   \RES     \ADDR  0       NUM=4
        GOSUB        RETURN_ADD sub_readwords
.ENDM


.MACRO  WriteWords   MEM,        SIZE,   RES,     ADDR=ZERO
!!      MEM:    Register
!!      SIZE:   Register with size of data
!!      RES:    Register: Resource Id (if 0: StdIn)
!!      ADDR:   Register: Optional Resource Address
        load_params  \MEM       \SIZE   \RES     \ADDR  0       NUM=4
        GOSUB        RETURN_ADD sub_writewords
.ENDM
```

```
.MACRO  set_IOP_address          ADD
        ATOR.V                   IOPS_START       \ADD
        RTOM                     %0(IOPS_START)   ZERO

IOP_ADDRESS_SET .assigna 1
.ENDM


!! at beginning of time we have no addresses for IOPs
IOP_ADDRESS_SET .assigna 0
```

Bibliografia

[1] A. Bartoloni et al., *An Overview of the APEmille Project*, Nuclear Physics B, Proc. Suppl. 60A (1998)

[2] A. Bartoloni et al., *Progress and Status of APEmille*, Nuclear Physics B, Proc. Suppl. 63A-C (1998)

[3] M. J. Flynn, *Very High-Speed Computing System*, Proc. IEEE, 54 (1966)

[4] E. Panizzi for the APE Collaboration, *Programming Languages and Environment for the APEmille Parallel Processor*, in *Computing in High-Energy Physics*, Berlino (1997)

[5] S. Salvatori, *FLINT: l'ottimizzazione del microcodice per APEmille*, Nota Interna n. 1083 INFN (1997)

[6] F. Jegerlehner et al., *Requirements for High Performance Computing for Lattice QCD: Report of the ECFA Working Panel*, Preprint ECFA/99/200 (1999)

[7] The APEmille collaboration, *apeNEXT: a Multi-TFlops LGT Computing Project*, Proposal approvata dall'INFN (2000)

[8] John C. Martin, *Introduction to Languages and the Theory of Computation*, McGraw-Hill (1997)

[9] D. Elsner, J. Fenlason et al., *using AS*, Free Software Foundation (1998)

[10] http://www.gnu.org/software/binutils/binutils.html

[11] D. M. Ritchie, B. W. Kernighan, *Linguaggio C*, Gruppo ed. Jackson (1989)

[12] S. Chamberlain, *The Binary File Descriptor Library*, Free Software Foundation (1991)

[13] I. L. Taylor, *BFD Internals*, Free Software Foundation (1999)

[14] R. Pesch, *GASP, an Assembly Preprocessor*, Free Software Foundation (1994)

[15] P. Bother, S. Chamberlain, I. L. Taylor, D. J. Delorie, *The GNU Linker Internals*, Free Software Foundation (1998)

[16] P. Maestrini, *Sistemi Operativi*, McGraw-Hill (1994)

[17] Silberschatz & Galvin, *Operating System Concepts*, Addison Wesley (1994)

[18] V. Carl Hamacher, Zvonko G. Vranesic, Safwat G. Zaky, *Computer Organization*, McGraw-Hill (1996)

[19] APEmille collaboration, *The APEOS project: Architecture of the APEmille Operating System*, nota interna INFN (Frascati) (1998)