Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria – Sede di Modena

Corso di Laurea in Ingegneria Informatica – *Nuovo Ordinamento*

Analisi comparativa tra processori Intel ad architettura IA-32 e IA-64

Relatore: Candidato:

Prof. Rita Cucchiara Daniele Oronzo Velardi

Anno Accademico 2003-2004

Sommario

Indice delle figure

Ringraziamenti

I miei più sentiti ringraziamenti vanno a tutti coloro che hanno permesso la realizzazione di questa tesi di laurea.

In particolar modo mi preme ringraziare il mio relatore e tutti i dottorandi che mi hanno pazientemente supportato durante questi mesi di lavoro.

E da ultimi, ma non per questo meno importanti, ringrazio i miei genitori, Angelo e Caterina, che hanno avuto fiducia in me in tutti questi anni.

1 Introduzione

1.1 Scopo

Questa tesi si propone lo scopo di analizzare le differenze presenti tra le due architetture Intel IA-32 e IA-64.

Si comincia analizzando l'evoluzione del mercato delle CPU per mettere in risalto le motivazioni che hanno spinto l'Intel ad introdurre una nuova architettura proprio durante il passaggio delle attuali architetture dai 32 ai 64 bit.

Dopodichè si evidenziano le principali caratteristiche dell'attuale architettura IA-32 (senza alcuna pretesa di esaustività, perché ampiamente diffusa negli attuali sistemi) per andarle poi a confrontare con quelle della nuova architettura IA-64, che si prefigge lo scopo di risolvere tutti gli annosi problemi che l'IA-32 si trascina ormai da anni. Dunque, ci si sofferma maggiormente sugli elementi di questa nuova architettura, perché ritenuti maggiormente interessanti ed innovativi, andando ad esaminare per ogni caratteristica dell'IA-32 gli attuali problemi e la corrispondente soluzione adottata nell'IA-64.

Infine, per avvalorare tutte le innovazioni apportate da questa nuova architettura, attraverso esperienze di laboratorio viene effettuata l'analisi prestazionale delle due architetture per trarre delle conclusioni significative sui vantaggi effettivamente portati dal passaggio alla nuova architettura proposta da Intel.

1.2 Organizzazione

La tesi è divisa in diversi capitoli contenenti tutte le tematiche analizzate al fine di poter confrontare le due architetture:

- Il capitolo che segue introduce le motivazioni che hanno portato l'Intel ad introdurre una nuova architettura come l'IA-64.
- Nel terzo capitolo vengono analizzate rapidamente le principali caratteristiche dell'IA-32, ormai molto diffusa sul mercato.
- Nel quarto capitolo viene analizzata più in dettaglio l'architettura IA-64, con particolare attenzione a tutte quelle caratteristiche che la contraddistinguono dalla vecchia IA-32, mettendo spesso in relazione le differenze tra le due architetture.
- Nel quinto capitolo vengono mostrati i risultati che sono stati ottenuti dallo studente durante dei test prestazionali svolti su un simulatore, aventi lo scopo di verificare la validità delle scelte progettuali dell'IA-64.
- Nel sesto ed ultimo capitolo sono state espresse le proprie conclusioni riguardanti la transizione da IA-32 a IA-64, basando il proprio giudizio sulle differenze rilevate a livello architetturale ed i risultati ottenuti dai test prestazionali.

Oltre a questi capitoli è presente un'appendice, che mostra come è stato possibile svolgere i test in laboratorio, un glossario, che contiene tutti i termini inglesi di natura tecnica non comprensibili ai non addetti ai lavori, e una bibliografia, che riporta tutte le fonti da cui si è attinto per redigere questa tesi.

2 Le motivazioni della transizione verso una nuova architettura

La storia del mercato dei microprocessori a partire dall'inizio degli anni '90 potrebbe riassumersi come una sostanziale mescolanza delle caratteristiche della filosofia di progettazione **RISC (Reduced Instruction Set Computing)** e delle caratteristiche della filosofia di progettazione **CISC (Complex Instruction Set Computing)**, con maggiore attenzione al raggiungimento delle migliori prestazioni che ai 'precetti' che questi due acronimi sottendono [3].

Come riferimento, riportiamo qui di seguito una tabella riassuntiva delle differenze delle due filosofie così come si erano delineate negli anni '70:

	Caratteristiche CISC	Caratteristiche RISC
Approccio fondamentale	La complessità si sposta dal codice all'hardware	La complessità si sposta dall'hardware al software, in pratica al compilatore che deve essere molto efficiente
Conseguenze della scelta per il programmatore	Il codice è molto compatto e occorre poca memoria per contenerlo; è l'hardware che s'incarica di decodificare istruzioni anche molto compatte e molto complesse	La dimensione del codice aumenta in favore della semplificazione dell'hardware.
Conseguenze della scelta a livello hardware	-Pochi registri. -Presenza di una ROM di decodifica. -ISA molto articolato con centinaia di istruzioni. -Modalità di indirizzamento memoria-memoria	-Molti registri -Non esiste la modalità di indirizzamento memoria-memoria, ma alla memoria si accede solo con il load e lo store -ISA con qualche decina di istruzioni soltanto -Direct execution -Uso della pipeline per diminuire il ritardo del critical path.

Figura 1: CISC vs RISC [1]

Naturalmente le differenze strutturali tra i due approcci permangono, una su tutte: il numero di registri indirizzabili dai rispettivi set di istruzioni che consente ad esempio una gestione del compilatore per l'ottimizzazione più aggressiva da parte dei processori RISC, potendo questi disporre di un numero di registri nettamente superiore [3].

Durante gli ultimi anni, spinte dalle evoluzioni del mercato sia nella crescente domanda di migliori prestazioni, soprattutto in campo grafico e multimediale, che dall'evoluzione di Internet, le principali aziende produttrici, Intel® e AMD®, si sono date battaglia e i vantaggi portati da questo regime di concorrenza "perfetta" si sono visti soprattutto in termini di aumento di prestazioni e di calo dei prezzi, con grossa soddisfazione dell'utente finale.

Ma il problema che si sta riscontrando negli ultimi anni è che questa continua esasperazione nella ricerca della velocità di esecuzione maggiore per mezzo delle tecniche usate negli ultimi 10 anni ha sì permesso di superare abbondantemente la tanto agognata soglia del GHz di clock (al momento della redazione della tesi sono presenti sul mercato microprocessori operanti alla ragguardevole frequenza di 3,6 GHz), ma ciò ha distolto l'attenzione dei progettisti dalle altre parti fondamentali dell'architettura della CPU. Come si può ben capire, a distanza di alcuni anni questa grave mancanza sta venendo fuori prepotentemente limitando lo sviluppo dei prossimi microprocessori.

Infatti, a parte l'esecuzione più veloce delle singole operazioni, che attiene prettamente alla tecnologia impiegata nella realizzazione degli elementi logici e alla loro organizzazione, la strada seguita dalla maggior parte dei costruttori è stata quella di eseguire un numero maggiore di operazioni in parallelo per incrementare le prestazioni ossia di perseguire lo sviluppo a **livello di parallelismo delle istruzioni (ILP)**. Questo è stato possibile tramite la realizzazione di **pipeline superscalari** sempre più profonde, che devono però essere

accompagnate da tecniche di controllo, di **predizione delle diramazioni** e di riordino delle istruzioni per poter essere adeguatamente parallelizzate.

L'esecuzione in pipeline è naturalmente favorita da istruzioni di ridotta complessità (approccio RISC) e questo è uno dei motivi per cui gran parte delle risorse dei moderni processori x86 sono spese in fase di decodifica per scomporre le complesse (e di lunghezza variabile) istruzioni dell'*ISA* x86 in istruzioni like-RISC (ad esempio le micro-operations del Pentium® 4) da inviare alle diverse unità di esecuzione. Analogamente, tecniche come la predizione delle diramazioni o **l'esecuzione fuori ordine** contribuiscono a spostare la complessità dal software all'hardware, aspetto che sarebbe in contrapposizione con la filosofia RISC, ma che ormai ha preso piede in moltissimi processori RISC, G5® compreso, perché consentono di ottenere alte prestazioni [3].

A sua volta, l'implementazione dell'algoritmo di esecuzione fuori ordine per la gestione delle false dipendenze dei dati, ha richiesto anche l'uso della tecnica di **ridenominazione dei registri**, che consente di utilizzare un numero maggiore di registri fisici rispetto a quanto previsto dall'ISA originale; questo è il caso tipico delle CPU con architettura x86 che dovendo mantenere la compatibilità all'indietro ha dovuto fare i conti con gli insufficienti 8 registri general purpose a 16 bit; ma non deve stupire neanche che un processore tipicamente RISC come il G5 che possiede un numero notevole di registri indirizzabili abbia fatto ricorso alla predetta tecnica.

La latenza di memoria continua ad essere una delle grandi sfide dei moderni processori e anche in questo campo le novità riguardano essenzialmente le possibilità offerte dall'integrazione [3], che hanno permesso di integrare *on-die* cache di primo e secondo livello di notevoli dimensioni (fino ad 1 MB di cache di secondo livello nel Pentium 4). Di questo aspetto si sono avvantaggiate soprattutto le architetture x86 dato che necessitano

mediamente di un 40% in più di cache istruzioni per la decodifica delle istruzioni x86, contribuendo a colmare il gap di cui godevano i processori RISC nella computazione su interi.

Per quanto riguarda la computazione intensiva per floating point, usata nelle applicazioni multimediali, si è andato affermando negli ultimi anni la tendenza ad estendere il parallelismo ai dati di questa natura per processare un numero maggiore di elementi in minore tempo. Questa tecnica adottata dalle istruzioni MMX, 3DNow!, SSE, 3DNow2!, SSE2 e SSE3 (nei microprocessori x86) è conosciuta con il nome di Single Instruction Multiple Data (SIMD) [3]; ma deve far riflettere come a questa scelta, ancora una volta, siano giunti anche i processori nativi RISC sebbene l'introduzione di nuove istruzioni e di nuove unità a livello hardware non appartenga certo ai canoni RISC.

Attualmente i maggiori produttori stanno mettendo a punto processi produttivi a 0.09 micron a 7 livelli di metallizzazione con connessioni in rame o alluminio con l'effetto di [3]:

- Aumentare il numero dei transistor a parità di dimensioni del chip e quindi più unità funzionali;
- Ridurre le tensioni di alimentazione dei core e quindi la potenza assorbita ma soprattutto quella dissipata (un sentito problema degli ultimi anni);
- Incrementare le frequenze di lavoro dei processori.

Nel caso dell'architettura x86 o IA-32 (Intel Architecture 32 bit) queste ed altre innovazioni porteranno sicuramente ad aumentare ancora le prestazioni di velocità, ma da questo punto di vista negli ultimi anni si è assistito ad un netto rallentamento nella crescita, ciò è dovuto al fatto che ormai questa architettura è vecchia di oltre 20 anni e le sue potenzialità sono state sfruttate ormai al massimo.

I motivi di tale limite sono [3]:

- **Lunghezza variabile delle istruzioni:** ciò comporta una fase di decodifica molto complessa che per compensazione richiede frequenze di lavoro più alte o pipeline più profonde, con tutti i problemi che ne derivano dallo svuotamento o dallo stallo di quest'ultime;
- **Carenza dei registri:** gli indirizzi dei registri nell'architettura x86 sono di soli 3 bit che consentono quindi l'indirizzamento di soli $2^3 = 8$ registri di uso generale. I moderni processori hanno cercato di superare questo limite per mezzo di tecniche come la ridenominazione dei registri che aggiungono però ulteriore complessità;
- **Floating Point Stack:** le istruzioni x87 usano per le proprie operazioni uno stack del quale viene fatto un uso intensivo con spreco di molti cicli per posizionare i dati occorrenti alla sua cima e che costituisce la ragione principale della sostanziale inferiorità delle CPU x86 rispetto alle CPU RISC nella computazione in virgola mobile.
- **Istruzioni che referenziano indirizzi di memoria:** con evidenti rallentamenti dovuti alle latenze associate ai chip di memoria in confronto a quelle che caratterizzano i registri interni della CPU (mentre le CPU RISC adottano un modello di memoria Load/Store che prevede che le istruzioni operino solo tra registri e gli accessi alla memoria siano espliciti e deputati solo alle istruzioni di load e store);
- **Dimensioni del die:** i buffer sempre più grandi, sempre più sviluppate tecniche di branch prediction e la decodifica di istruzioni x86 in microistruzioni RISC sono tutti accorgimenti per estrarre grandi prestazioni dalle CPU x86 mantenendone la

back-compatibility, ma ciò richiede molti più transistor e quindi a parità di processo produttivo molto più spazio *on-die* e quindi costi maggiori.

Ed è per questo che l'Intel ha ritenuto che, per riuscire a garantire l'incremento delle prestazioni che ha visto la recente storia delle CPU x86, occorresse cambiare radicalmente il set di istruzioni anche a costo di penalizzare la compatibilità con l'architettura precedente ma offrendo nel contempo prestazioni tali da giustificare un simile passo [3].

E per dare una svolta decisiva nel mondo dei PC, questa importante decisone è stata presa proprio in concomitanza del passaggio dalla tecnologia a 32 bit a quella a 64 bit. Molti dubbi sono sorti riguardo la reale necessità di una tale rivoluzione, a tale riguardo basti pensare a quello che è accaduto circa 10 anni fa, quando si è passati da 16 a 32 bit: pian piano superando alcuni problemi abbiamo vissuto la transizione da Windows for Workgroup 3.11 a Windows 95, fino a giungere all'attuale Windows XP, con notevoli benefici per il mercato degli utilizzatori. Oggi, lo scenario che si presenta è molto simile, e le potenzialità di una tale transizione sono enormi, si pensi ad esempio al tanto agognato supporto dei comandi vocali e tanto altro ancora che oggi risulta addirittura difficile da immaginare!

Si noti che già nei tardi anni '90 sono stati introdotti i primi processori RISC a 64 bit, la cui applicazione è stata strettamente di tipo server, rimanendo perlopiù sconosciuti al grande pubblico. Questo non è assolutamente un dettaglio da trascurare, al contrario la storia delle CPU ci ha dimostrato più volte che la completa affermazione di un processore su di un altro non dipende solamente da parametri tecnologici ma anche dall'economicità del prodotto e dal momento di immissione sul mercato (*time-to-market*): infatti il dominio dell'architettura x86 nel mercato dei Personal Computer è dovuta in gran parte ai microprocessori Intel 8086 ed 8088, il primo dei quali fu immesso sul mercato con un anno

e mezzo di anticipo rispetto agli avversari a 16 bit di Motorola® e Zilog®, mentre il secondo fu l'unico microprocessore con bus dati esterno a 8 bit a poter indirizzare fino ad 1 Mbyte di memoria ad un prezzo estremamente basso mantenendo la compatibilità con i programmi del fratello maggiore 8086, e contribuendo così notevolmente alla diffusione dell'ISA x86.

E proprio facendo tesoro di tali esperienze, il produttore più pronto ad attuare la transizione ai 64 bit è stata l'AMD che ha introdotto l'Athlon 64, che si può considerare il primo processore a 64 bit per soluzioni di tipo desktop.

Ciò, in verità non sembra aver danneggiato particolarmente l'Intel che ha scelto una strada completamente diversa da quella dell'AMD: come abbiamo già accennato, l'Intel forte della sua posizione sul mercato ha deciso di cambiare completamente l'architettura del processore, al contrario di quanto fatto da AMD che ha deciso "semplicemente" di ampliare la vecchia ISA x86, probabilmente perché non era sicura di poter imporre da sola una nuova *instruction set architecture* sul mercato.

I principali vantaggi derivanti da una semplice transizione dell'architettura da 32 a 64 bit sono l'aumento della grandezza dei registri, che era stato già adottato anni addietro tramite le estensioni SIMD, e l'aumento della memoria principale indirizzabile che sale dagli attuali 2^{32} = 4.3 GByte ad un massimo teorico di 18 milioni di Terabyte, questo rappresenta un evidente vantaggio per i server che possono per esempio caricare direttamente in memoria interi database di dimensioni superiori ai 4.3 GByte (anche se i processori Intel Xeon© a 32 bit erano già riusciti ad indirizzare fino a 64 GByte grazie a particolari tecniche), ma potrà tornare molto utile in un futuro non molto lontano (3 o 4 anni) anche nelle applicazioni multimediali per sistemi desktop.

Dunque ho ritenuto particolarmente interessante andare ad analizzare la transizione architetturale decisa dall'Intel, che in concomitanza con la migrazione dai 32 ai 64 bit non si è limitata ad ampliare semplicemente la grandezza dei registri e dei bus (come fatto dall'AMD), ma è andata a stravolgere completamente l'architettura interna adottando una nuova ISA incompatibile (almeno in parte come vedremo nel par. 4.8) con la vecchia e gloriosa x86.

Per un approfondimento di questa prima parte introduttiva si consiglia di leggere [3].

3 IA-32 Intel Architecture

3.1 Introduzione

Le basi dell'architettura IA-32 sono state poste nel 1978 con il processore 8086 a 16-bit, macchina CISC tradizionale, che si impose subito sul mercato; sull'onda del successo di questo processore nacquero altri processori a 16-bit, come l'8088 e l'80286, che mantennero la compatibilità dell'ISA x86. Ma il primo processore ad architettura IA-32 è stato sicuramente l'80386 che ha segnato anche la grande transizione dai 16 ai 32 bit. Già nell'80486 si sono cominciate a vedere le prime influenze di progettazione di tipo RISC con l'introduzione del concetto di pipeline (una pipeline a cinque stadi: *prefetch*, decodifica, generazione dell'indirizzo, esecuzione, *Write-Back*), ma ancora senza alcun elemento superscalare. Infatti, è stato con l'introduzione del Pentium che si è introdotto il concetto di superscalarità con l'adozione di due unità di esecuzione ad interi poste su due pipeline distinte (denominate u-pipe e v-pipe), capaci di eseguire insieme due istruzioni per clock. Per maggiori informazioni consultare il manuale Intel [4].

In questa trattazione prenderemo spesso in considerazione le interessanti caratteristiche introdotte nell'ultimo nato dell'Intel per sistemi desktop: il Pentium 4.

Le operazioni compiute dal Pentium 4 possono essere riassunte così [20]:

1. Il processore carica le istruzioni di lunghezza variabile (fino ad un massimo di una word pari a 32 bit) dalla memoria nell'ordine previsto dal programma;
2. Ogni istruzione del Pentium IV viene tradotta in una o più istruzioni *like-RISC* di lunghezza prefissata, chiamate *micro-ops* (un'istruzione contiene da 1 a 4 micro-ops di tipo RISC, ognuna delle quali è lunga 118 bit);
3. Il processore esegue le micro-ops in una pipeline superscalare grazie ad un algoritmo di esecuzione fuori ordine;
4. Alla fine viene restituito il risultato di ogni micro-op al corrispondente registro nell'ordine originale del programma.

In effetti, l'architettura del Pentium IV è composta da una parte esterna di tipo CISC con un cuore di tipo RISC. Le *micro-ops* passano attraverso una pipeline lunga ben 31 stadi (nel Pentium 4 con *core* Prescott); in alcuni casi, le micro-ops richiedono più stadi di esecuzione, rendendo ancora più lunga la pipeline. Perciò possiamo indicare questo tipo di filosofia di progettazione con il termine **Post-CISC [1]**.

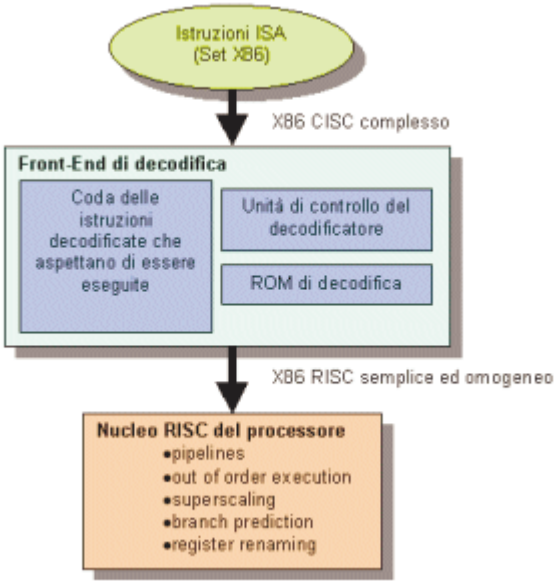

Figura 2: Post-CISC [1]

Questa soluzione ha permesso di sfruttare la bontà dell'approccio RISC pur mantenendo la *back-compatibility* con i processori precedenti e decretando così il successo di questa architettura.

Ora analizzeremo la microarchitettura denominata NetBurst che è stata implementata nel Pentium IV.

3.2 Micro-architettura NetBurst™

Questa architettura ha permesso l'introduzione di nuove tecnologie che si vanno ad aggiungere a quelle già apportate negli ultimi venti anni. Molti di questi miglioramenti si sono resi disponibili grazie al raffinamento dei processi produttivi (attualmente si produce a 90 nm) e delle tecniche di progettazione dei circuiti. Gli obiettivi raggiunti da questa architettura sono:

a) Esecuzione di codice scritto sia con istruzioni native x86 che con istruzioni SIMD;
b) Scalare sempre più verso maggiori frequenze di clock.

Per raggiungere proprio quest'ultimo obiettivo, ed arrivare a frequenze di clock sempre maggiori (attualmente 3,6 GHz) si è dovuto diminuire il *critical path* facendo ricorso all'aumento smisurato delle pipeline che nel Pentium IV hanno raggiunto la considerevole profondità di 31 stadi. Da sola questa soluzione non avrebbe portato effettivi benefici se si fosse scontrata con problemi di stallo e svuotamento delle pipeline, ma ciò è stato evitato adottando raffinate tecniche di **branch prediction** (par. 3.2.4) e **data prefetch** (par. 3.2.5). Seguono gli elementi caratterizzanti di questa microarchitettura.

3.2.1 Architettura superscalare

L'implementazione di un'architettura **superscalare** in un processore permette di processare ed eseguire più istruzioni indipendenti contemporaneamente in unità funzionali diverse. Purtroppo però tale implementazione introduce un alto numero di problemi di progettazione abbastanza complessi riguardanti la pipeline [15].

Di norma il concetto di superscalarità viene associato alle architetture di tipo RISC, ma la stessa filosofia di progettazione può essere applicata anche alle macchine CISC, come è accaduto alle famiglie dei vari Pentium.

Il termine superscalare si riferisce a macchine progettate per aumentare le prestazioni di esecuzione di istruzioni scalari. In molte applicazioni, la maggior parte delle operazioni sono in quantità scalare; di conseguenza, l'approccio superscalare rappresenta il prossimo passo nell'evoluzione dei processori *general purpose* [15].

Il punto fondamentale dell'approccio superscalare è la possibilità di eseguire contemporaneamente istruzioni indipendenti in differenti pipeline. Questo concetto può essere sfruttato permettendo alle istruzioni di essere eseguite in un ordine diverso da quello originale. Per fare ciò si utilizzano più unità funzionali, ognuna delle quali viene implementata come una pipeline, che rendono possibile l'esecuzione di istruzioni diverse [15].

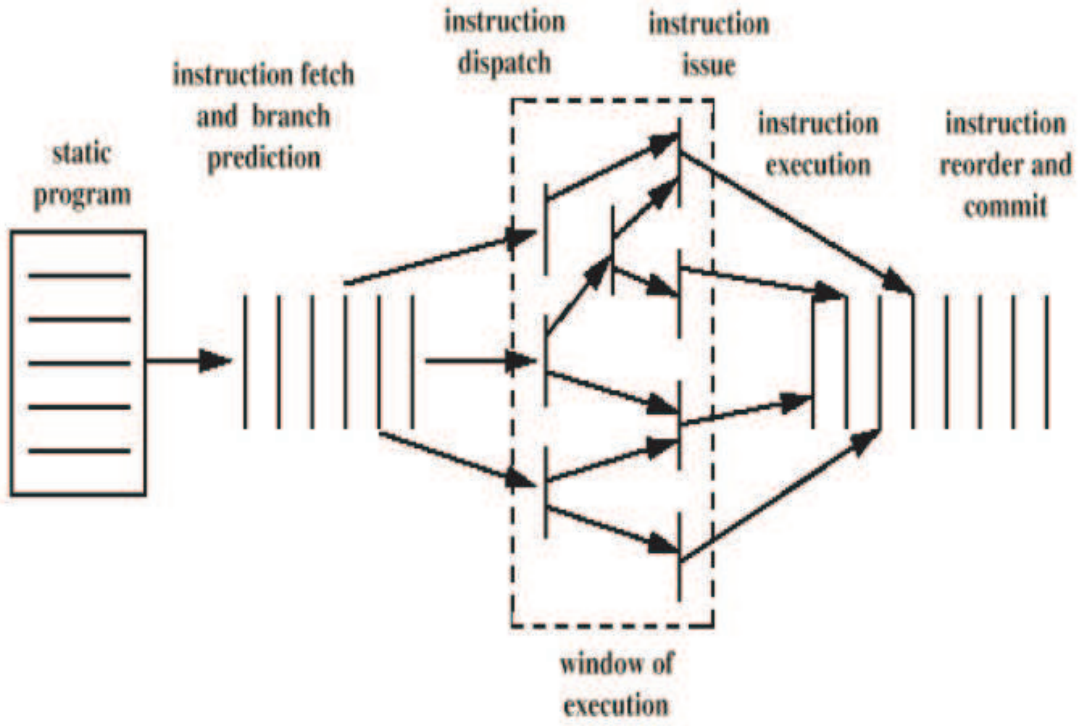

Figura 3: Architettura superscalare

Dunque la fattibilità dell'approccio superscalare dipende dalla possibilità di eseguire più istruzioni in parallelo. Il termine **parallelismo a livello istruzione (ILP)** si riferisce al numero medio di istruzioni di un programma che possono essere eseguite in parallelo senza sovrapporsi. Per massimizzare questo numero (e minimizzare così i *clock per instruction*) si sfrutta una combinazione di ottimizzazioni a livello di compilazione e di tecniche hardware. Infatti perché una sequenza di istruzioni possa essere eseguita in parallelo senza sovrapposizioni, queste debbono essere indipendenti fra di loro. Il parallelismo a livello istruzione è deciso dalla frequenza di dipendenza dei dati e delle procedure nel codice. Questi fattori dipendono in parte dall'*instruction set architecture* ed in parte dall'applicazione che si vuole eseguire. Inoltre, il parallelismo a livello istruzione

dipende dalla latenza di un operazione, cioè il tempo necessario a rendere disponibile il risultato di un operazione per l'operazione seguente [15].

L'hardware dedicato a garantire il corretto funzionamento di tale architettura si basa sui seguenti concetti che verranno illustrati di seguito: **out of order speculative execution** (par. 3.2.2), **register renaming** (par. 3.2.3), **branch prediction** (par. 3.2.4) **e data prefetch** (par. 3.2.5).

3.2.2 Out of order speculative execution

Come già accennato il parallelismo della macchina non si può ottenere semplicemente aumentando il numero di istanze per singolo stadio della pipeline, cioè aumentando il grado di parallelismo delle unità di esecuzione. Il processore deve anche essere capace di identificare il parallelismo a livello istruzioni e organizzare al meglio le fasi di fetch, decode ed execute delle istruzioni in parallelo.

In sostanza, il processore deve provare a "guardare oltre il corrente punto di esecuzione" per localizzare le istruzioni che possono essere portate nella pipeline ed eseguite. Dunque assume maggiore importanza l'ordinamento delle istruzioni.

Per ottimizzare l'utilizzazione dei vari elementi della pipeline, il processore deve alterare l'ordine di una o più istruzioni rispetto all'ordine originale. L'unico vincolo da rispettare è che il risultato ottenuto debba essere corretto. Per fare ciò il processore deve risolvere le varie dipendenze e conflitti presenti [15].

La politica utilizzata nel Pentium IV è quella di **esecuzione fuori ordine**. Per fare ciò bisogna disaccoppiare gli stadi di decodifica da quelli di esecuzione, ciò è possibile utilizzando un buffer chiamato **instruction window**. In questo modo, dopo che il processore ha finito di decodificare un istruzione, questa viene posta nell'instruction window. Fino a quando questo buffer non si riempie, il processore può continuare a fare il fetch ed il decode delle nuove istruzioni presenti nel codice del programma eseguito. Quando un'unità funzionale diventa disponibile nello stadio di esecuzione, un istruzione viene spostata dall'instruction window allo stadio di esecuzione, solo se essa necessita

della particolare unità funzionale liberatasi e soprattutto se non esistono conflitti o dipendenze non ancora risolte con altre istruzioni presenti nell'instruction window [15].
Alla fine, però, i dati ottenuti andranno riordinati per essere restituiti nell'ordine originale previsto dal programma.

3.2.3 Register renaming

Quando si utilizza l'algoritmo di esecuzione fuori ordine non è possibile sapere a cosa corrisponde il valore assunto da ogni registro in un qualsiasi momento, perché non è stata rispettata la sequenza originale di istruzioni prevista dal programma. Perciò i valori contenuti possono essere in conflitto con l'uso del registro, e il processore deve risolvere i conflitti fermando uno stadio della pipeline [15].

Nel Pentium IV il metodo utilizzato per risolvere questo tipo di conflitti è basato sulla soluzione di duplicare le risorse. Questa tecnica prende il nome di **register renaming** [15].

La ridenominazione dei registri distingue fra registri logici e registri fisici; i registri logici sono mappati dinamicamente nei registri fisici attraverso apposite tabelle che vengono aggiornate ogni volta che un'istruzione viene decodificata [3]. Quando un nuovo valore deve essere memorizzato, viene allocato un nuovo registro per quel valore. Le istruzioni seguenti che richiederanno quel valore come sorgente di un operando otterranno il riferimento al nuovo registro allocato dal meccanismo di register renaming. Quindi il riferimento ad uno stesso registro in istruzioni differenti può riferirsi a registri fisici diversi, nel caso in cui si cerchino valori diversi [15].

Il programmatore è consapevole dell'esistenza dei soli registri logici, mentre l'implementazione dei registri fisici è nascosta [3].

La ridenominazione dei registri semplifica il controllo delle dipendenze fra i dati. In una macchina come il Pentium IV, che può eseguire istruzioni fuori ordine, i numeri dei registri logici possono diventare ambigui, poiché ad uno stesso registro può essere assegnata una

successione di valori diversi. Ma dato che i numeri dei registri fisici identificano in modo univoco ogni risultato, il controllo delle dipendenze non risulta più ambiguo [3].

3.2.4 Branch prediction

Nella programmazione comune le diramazioni interrompono il flusso della pipeline, facendo calare vertiginosamente le prestazioni, pertanto per minimizzare il numero di diramazioni sono necessari degli schemi di **branch prediction** [3].

Le diramazioni accadono frequentemente, in media ogni sei istruzioni; quindi è facile capire come in un'architettura superscalare, dove vengono eseguite anche quattro istruzioni per ciclo, la predizione di tali diramazioni diventa importante [3].

Il Pentium IV usa una strategia dinamica di predizione dei salti basata sulla cronologia delle più recenti istruzioni di salto eseguite. Queste informazioni vengono mantenute nella **Trace cache BTB** (vedi par. 3.3.1). Quando si incontra un salto nel flusso di istruzioni da eseguire, questo viene inserito in questo speciale buffer che può contenere fino a 512 linee a 4 vie associative. Se si incontra un salto già contenuto in tale buffer, allora si effettua una predizione del salto concordemente a quanto riportato in tale tabella. In caso di predizione di salto, l'indirizzo di destinazione del salto viene usato per caricare la nuova istruzione nel **Front-End BTB (**vedi par. 3.3.1). Una volta eseguita l'istruzione la cronologia di quella istruzione viene aggiornata in base al risultato del salto effettuato [15].

Se un'istruzione non è presente nella trace cache BTB, allora l'indirizzo dell'istruzione viene caricato cancellando, se necessario, la linea più vecchia della tabella. Inoltre, non esistendo una cronologia su cui basarsi viene usato un algoritmo di predizione statico [15].

Nel caso di errore di predizione (o *misprediction*) sarà necessario svuotare l'intera pipeline per lasciar posto all'esecuzione delle istruzioni corrette. Nelle ultime versioni del Pentium IV, se non si facesse uso di raffinate tecniche di predizione, a causa della lunghezza

smisurata delle pipeline una misprediction porterebbe ad un calo delle prestazioni del 20-30%.

3.2.5 Data prefetch

Nel corso di pochi anni le CPU hanno avuto un progressivo e vertiginoso aumento delle prestazioni soprattutto in termini di istruzioni processate al secondo (grazie anche all'adozione di più unità di esecuzione in parallelo). Per le memorie invece l'incremento di prestazioni è stato molto più limitato, al punto da costituire ben presto una vera e propria limitazione per i processori. Il problema è sostanzialmente dovuto alla così detta **latenza delle memorie**: il tempo necessario alla memoria (tempo di accesso + tempo di trasferimento) per fornire un'istruzione al processore è significativamente maggiore del singolo ciclo di clock in cui più istruzioni vengono eseguite in un'architettura superscalare (lo stesso discorso vale per i dati da leggere e da scrivere) [2].

Non potendo la tecnologia sopperire direttamente a questa limitazione, il problema è stato affrontato in modo diverso. Il processore è stato dotato di memoria cache, una memoria locale di dimensione contenuta (a causa del suo alto costo), ma molto veloce. La memoria centrale continua ad avere la sua latenza, però essa viene di fatto "mascherata" dalla memoria cache, che conserva una copia delle istruzioni e dei dati maggiormente utilizzati (secondo il modello *pseudo-LRU, least recently used*), così da non dover bloccare l'esecuzione del programma in attesa della memoria centrale [2].

Un successivo aumento delle prestazioni dei processori ha aggravato il problema, in particolare l'adozione di pipeline particolarmente lunghe ha reso necessario ridurre ulteriormente gli effetti della latenza, perché se uno qualsiasi degli stadi richiede un dato o un'istruzione non presente in cache (*cache miss*), tutta la pipeline va in stallo in attesa della risposta della memoria. L'accorgimento adottato è stato l'adozione di memorie cache più

capienti e organizzate su più livelli (oggi si utilizzano tra i due e i tre livelli di cache) a dimensioni decrescenti e velocità crescente scendendo di livello. La contemporanea evoluzione delle memorie non ha portato alcun miglioramento significativo nella riduzione della latenza, ma c'è stata una specializzazione nell'aumentare la quantità di byte trasferiti al secondo (vedi le attuali memorie DDR e RamBus, associate a bus di tipo QuadPumped), in modo da riuscire a riempire rapidamente le memorie cache quando sopravviene una richiesta [2].

Con pipeline ancora più lunghe e frequenze di clock ancora più elevate (fino a 31 stadi nell'ultimo processore della famiglia dei Pentium IV con core Prescott operante a 3,6 GHz), la limitazione imposta dall'accesso alla memoria ha cominciato a non essere sufficientemente risolta neanche dall'adozione delle memorie cache. Infatti negli ultimi processori a 32 bit usciti sul mercato è stato introdotto un meccanismo chiamato **data prefetch** (o esecuzione speculativa) che permette l'esecuzione di un blocco di istruzioni prima di sapere se tale blocco debba essere effettivamente eseguito. Per fare ciò è necessario il supporto sia del compilatore che di hardware dedicato.

Infatti il data prefetch può essere attuato in due modi: usando le quattro istruzioni di prefetch introdotte dal SSE (vedi par. 3.2.6) oppure attraverso un meccanismo automatico di hardware prefetch che tenta di predire quali dati e istruzioni saranno richiesti ed effettua un precaricamento in cache di tali dati, in modo da renderli subito disponibili e non tenere in stallo l'intera pipeline.

Come potrete certamente immaginare, l'implementazione di un tale automatismo direttamente all'interno del processore risulta particolarmente oneroso [2].

3.2.6 Tecnologia SIMD

Sempre per quanto riguarda la computazione in parallelo, ci si è accorti subito che l'instruction set x86 risultava inadatto a questo scopo, ma dovendo essere mantenuto per ragioni di compatibilità all'indietro si è proceduti ad un progressivo aumento di istruzioni e di registri specializzati (pratica in auge anche negli attuali processori RISC), in modo da consentire di eseguire più computazioni con una singola istruzione. Il termine **Single Instruction Multiple Data (SIMD),** che significa proprio istruzioni che permettono di manipolare i dati con un singolo comando, sta proprio ad indicare tutte quelle espansioni (MMX, SSE, SSE2 e SSE3) che si sono succedute nel tempo. In particolare le quattro generazioni di SIMD sono:

- Con il Pentium **MMX** furono introdotti 8 registri a 64 bit chiamati appunto MMX e le relative istruzioni per potere eseguire operazioni su word e double-word;
- Col Pentium III è stato inserito il primo **SSE (SIMD Extensions)**, che metteva a disposizione 8 registri a 128 bit chiamati XMM e le relative istruzioni per compiere operazioni in virgola mobile a singola precisione. Inoltre sono state inserite le istruzioni per il data prefetch (vedi par. 3.2.5);
- Col Pentium IV è arrivato anche l'**SSE2** che ha permesso di compiere operazioni in virgola mobile a doppia precisione (ben 144 istruzioni);
- Sempre nel Pentium IV (core Prescott) è arrivata l'ultima estensione **SSE3**, che aggiunge altre 13 nuove istruzioni;

L'intero set di estensioni SIMD dà ai programmatori la possibilità di sviluppare algoritmi capaci di elaborare numeri interi ed in virgola mobile a doppia precisione a 128 bit.

Le SIMD si propongono l'obiettivo di aumentare le prestazioni di applicazioni grafiche 3D, di riconoscimento vocale, di editing video e multimediali che abbiano come caratteristica:

- parallelismo intrinseco;
- modelli di accesso in memoria regolari e ricorrenti;
- operazioni ricorrenti sui dati;
- flusso di controllo senza dipendenze dei dati.

E' importante evidenziare anche come l'Intel sia sempre riuscita ad imporsi sugli altri costruttori, diventando lo standard di facto. Un esempio lampante si è avuto con l'introduzione delle SSE2 in concomitanza con il 3DNow! di AMD. La vasta adozione dell'estensione SSE2 presso le maggiori software house ha costretto l'AMD a tornare sui propri passi ridimensionando le proprie ambizioni espansionistiche e adottando suo malgrado le estensioni SIMD dell'Intel (pagandone le *royalty*).

Per maggiori informazioni consultare il manuale [7].

3.3 Organizzazione interna del Pentium IV

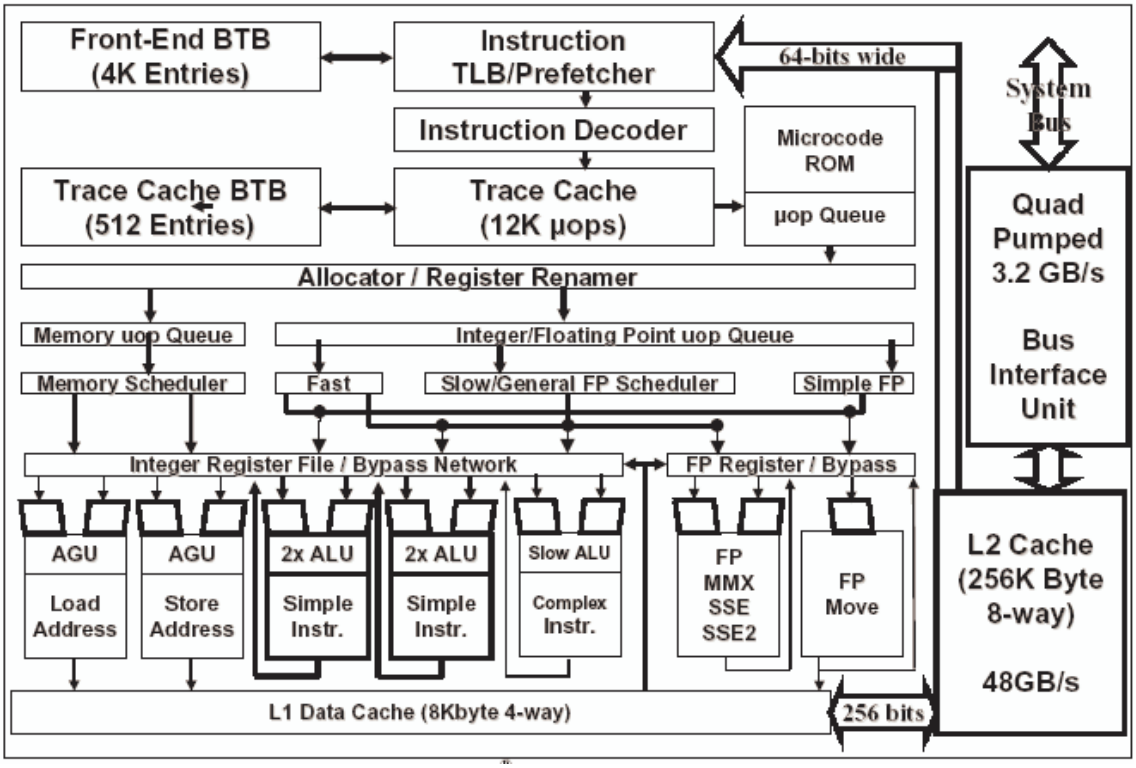

Figura 4: Block diagram del Pentium IV [8]

La maggior parte dei transistor presenti nel die del Pentium IV sono dedicati alla memoria cache. Tale memoria si sviluppa su due livelli [4]:

- Cache L1 on-chip: 12 K µops di **Trace Cache** a 8 vie associative e 8 KB di **Data Cache** a 4 vie associative con linee a 64 byte;
- Cache L2 on-chip: da 256 KB a 1 MB a 8 vie associative con linee a 64 byte.

I rimanenti transistor sono divisi principalmente in tre stadi: **In-Order Front-End**, **Out-Of-Order Superscalar Execution Core** e **In-Order Retirement Unit.**

3.3.1 In-Order Front-End

Il Pentium IV contiene un front-end di tipo in-order (in altre parole non cambia l'ordine originale delle istruzioni del programma) che può essere considerato estraneo agli scopi della pipeline, rimanendo fedele alla filosofia CISC. Il suo scopo è quello di rifornire la cache L1 delle istruzioni **(trace cache)**, che costituisce il primo stadio della pipeline. Di solito il processore comincia a lavorare dalla trace cache; quando accade un cache miss, il front-end fornisce delle nuove istruzioni alla trace cache [15].

Con l'aiuto del **branch target buffer (BTB)** e dell'**instruction lookaside buffer (I-TLB)**, vengono caricati 64 byte alla volta di istruzioni macchina del Pentium IV dalla cache L2. Di norma, queste istruzioni vengono caricate nella sequenza in cui si trovano nel codice sorgente del programma, ma con l'introduzione della predizione dei salti e dell'hardware data prefetch si tende a modificare comunque l'ordine naturale. L'unità I-TLB si occupa di tradurre l'indirizzo dell'instruction pointer in un indirizzo fisico della cache L2 [15].

Una volta che l'istruzione è stata caricata, l'**instruction decoder** controlla la lunghezza dell'istruzione (ciò è dovuto al fatto che le istruzioni del Pentium IV hanno lunghezza variabile) e ogni istruzione macchina viene tradotta in un massimo di 4 micro-operations, ognuna delle quali è lunga 118 bit. Rispetto a macchine puramente di tipo RISC, che hanno istruzioni lunghe 32 bit, nel Pentium IV è stato necessario usare micro-ops così lunghe per permettere di compiere operazioni più complesse di quelle permesse dal canonico approccio RISC [15].

Quindi le micro-ops generate vengono salvate nella trace cache [15].

I primi due stadi della pipeline sono dedicati alla selezione delle istruzioni nella trace cache impiegando un meccanismo di predizione dei salti dinamico basato sul contenuto della trace cache BTB (come già visto nel par. 3.2.4). Una volta ultimata la predizione della diramazione, la trace cache prende le micro-ops già decodificate dall'instruction decoder e li assembla in una sequenza di micro-ops chiamate tracce. Queste vengono poi trasferite alla **microcode ROM** (control unit microprogrammata) che contiene tutte le micro-ops corrispondenti ad un'unica istruzione complessa Pentium IV [15].

3.3.2 Out-Of-Order Superscalar Execution Core

Questo stadio si occupa di allocare le risorse necessarie all'esecuzione delle micro-ops in ingresso [15]:

- Se il registro richiesto da una micro-op non è disponibile viene bloccata la pipeline;
- Alloca una linea nel **reorder buffer (ROB)**, che tiene traccia dello stato di completamento di una delle 126 micro-ops che possono essere processate alla volta;
- Alloca il risultato della micro-op in uno dei 128 registri general purpose o *floating-point*;
- Alloca una linea in una delle due code dello scheduling.

Questo stadio rimappa i riferimenti degli 8 registri general purpose a 16 bit previsti dall'architettura nei 128 registri fisici realmente presenti nel processore (come già visto nel par. 3.2.3), eliminando le dipendenze causate dal numero limitato di registri previsti dall'architettura nativa x86 [15].

Dopo che le risorse sono state allocate e i registri rinominati, le micro-ops vengono poste in una delle due code, dove rimangono in attesa di passare allo scheduling. Una coda è per le operazioni in memoria (load e store) e l'altra per tutti gli altri tipi di operazioni. Ogni coda funziona in modalità FIFO (first-in first-out), ma non esiste alcun tipo di relazione tra l'ordine reciproco delle due code [15].

Gli *scheduler* delle due code si occupano di recuperare le micro-ops e di distribuirle fra le varie unità di esecuzione, nel momento in cui queste si rendano disponibili. Possono essere inviate fino a sei micro-ops a ciclo. Se più di una micro-op richiede la stessa unità di

esecuzione, allora lo scheduler le invia in sequenza dalla coda sempre secondo l'ordinamento FIFO [15].

Quattro porte collegano gli scheduler alle unità di esecuzione. La porta 0 viene usata per le istruzioni ad interi ed in virgola mobile, con l'eccezione delle operazioni ad interi più semplici e dei salti per predizione errata, che utilizzano la porta 1. Sia la porta 0 che quella 1 vengono usate dalle nuove istruzioni SIMD. Le porte rimanenti sono usate per le operazioni di load e store in memoria [15].

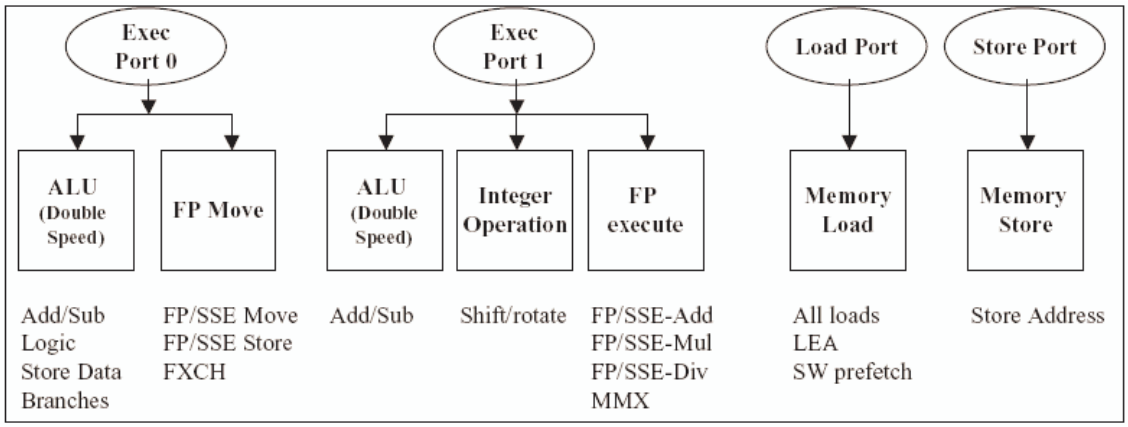

Figura 5: Porte di esecuzione del Pentium IV [8]

Si noti che le porte 0 e 1 possono di distribuire più di un'istruzione per ciclo perchè le ALU riescono ad operare al doppio della velocità delle altre unità di esecuzione, ecco perchè teoricamente è possibile eseguire sino a 6 micro-ops per ciclo pur essendoci solo quattro porte a disposizione [15].

I register file costituiscono le sorgenti delle unità di esecuzione. Le unità di esecuzione recuperano i valori cercati dai register file e dalla cache dati L1.

Uno stadio a parte della pipeline viene usato per calcolare i flag che tipicamente vengono usati per le istruzioni di salto. Lo stadio seguente della pipeline si occupa di controllare la

veridicità delle istruzioni di salto, confrontando il risultato del salto con la predizione. In caso di misprediction vengono rimosse le relative micro-ops nei vari stadi della pipeline, e le istruzione corrette da eseguire vengono fornite al **branch predictor** che riavvia la pipeline dal nuovo indirizzo di destinazione [15].

3.3.3 In-Order Retirement Unit

La sezione di ritiro riceve i risultati delle µops eseguite dall'execution core e processa i risultati in modo che lo stato dei registri sia aggiornato secondo quanto previsto dall'ordine originale del programma (le eccezioni vengono sollevate solo quando viene ritirata l'istruzione corrispondente). Quando una micro-ops viene completata e il risultato viene scritto nella destinazione, viene aggiornato il ROB. Inoltre viene tenuta traccia dei salti e si provvede ad aggiornare il BTB secondo i risultati delle diramazioni prese [4].

4 IA-64 Intel Architecture

4.1 Introduzione

Con il Pentium IV, l'IA-32 che è stata senza dubbio l'architettura per sistemi desktop di maggior successo sembra aver espresso già tutto il suo potenziale. Per questo motivo l'Intel insieme alla Hewlett-Packard (HP) ha sviluppato una nuova architettura a 64-bit (che non ha nulla a che fare con l'architettura dell'HP 64-bit PA-RISC) per raggiungere nuove vette prestazionali e allo stesso tempo per riservarle ampi margini di crescita e miglioramento (almeno come quanto dimostrato dall'IA-32). Infatti, l'IA-64 è una nuova architettura che nasce dopo anni di ricerche presso le due compagnie ed alcune università (come quella dell'Illinois). Questa nuova architettura vanta una vasta circuiteria operante ad un'alta velocità posta in un chip di ultima generazione che fa un uso sistematico del parallelismo. L'architettura IA-64 rappresenta un significativo allontanamento dal trend comune della superscalarità che ha dominato lo sviluppo dei più recenti microprocessori [15].

I concetti base da sottolineare in questa architettura sono i seguenti [15]:

- **Parallelismo a livello istruzione (ILP)** che viene esplicitato nelle istruzioni macchina, piuttosto che essere determinato *run-time* dal processore;
- **Very long instruction words (VLIW)** che contengono gruppi di tre istruzioni like-RISC;

- **Branch predication** nelle stesse istruzioni macchina, piuttosto che predizione dei salti;
- **Speculative loading** dei dati e delle istruzioni.

Intel ed HP hanno attribuito a questa combinazione di concetti l'acronimo **EPIC (Explicity Parallel Instruction Computing)**, dove appunto l'architettura IA-64 nasce secondo questa filosofia di progettazione. Diversamente da quanto fatto nei processori x86 venti anni fa, quando le unità di memorizzazione di massa erano molto piccole e costose e il software lasciava molto a desiderare, con l'IA-64 cresce l'importanza del compilatore nei riguardi delle prestazioni della macchina, quindi il compilatore è parte integrante di una CPU con approccio EPIC. Le prestazioni dipendono direttamente dalla qualità delle informazioni fornite dal compilatore. In accordo con l'approccio RISC, la complessità si sposta dall'hardware al software [15].

Il primo prodotto di questa architettura è stato l'**Itanium®**, un puro esercizio di stile che in realtà non è stato mai immesso sul mercato a causa delle sue scarse prestazioni, mentre da pochi mesi è stato messo in produzione l'**Itanium 2** per dare l'assalto definitivo al mercato. Quindi nel seguito si farà spesso riferimento a quest'ultimo processore perché all'atto pratico è l'unico esemplare rappresentativo dell'architettura IA-64 presentato sul mercato.

Comunque, per il momento l'Intel ha deciso di mantenere in produzione e soprattutto di continuare a sviluppare anche la famiglia del Pentium IV, poiché questo rappresenta ancora l'offerta di punta per il mercato dei desktop, mentre l'Itanium si prefigge lo scopo di andare a sostituire completamente in pochi anni il processore Xeon nel settore server (o almeno così si spera!).

Per l'Intel, la transizione verso una nuova architettura non compatibile all'indietro con l'ISA x86 è stata una decisione molto importante e sofferta, ma è stata imposta dall'avanzamento tecnologico stesso [15]. La motivazione di tale decisione diviene evidente osservando l'approccio scelto per continuare a sviluppare la vecchia IA-32 che, con l'aumento del numero di transistor per chip messo a disposizione dai nuovi processi produttivi, è stato quello di usare i transistor in più per aumentare la grandezza della cache on-chip, anche se ciò ha portato ad aumenti troppo contenuti dell'*hit rate*; inoltre si è scelto di aumentare il grado di superscalarità aggiungendo altre unità di esecuzione, anche se oramai si è raggiunto un livello di complessità difficile da gestire, a causa della logica di controllo per la predizione dei salti (branch prediction) e per l'esecuzione fuori ordine (out-of-order execution) che a sua volta richiede un ampio numero di registri (attraverso la tecnica di renaming register), senza parlare poi dei cali di prestazioni nel caso di *misprediction* nelle pipeline troppo lunghe. Attualmente l'utilizzo dei due approcci ha portato ad eseguire un massimo teorico di 6 istruzioni per ciclo [15], che purtroppo si realizza raramente con un grande spreco di risorse.

Quindi Intel ed HP hanno deciso di progettare insieme un'ambiziosa architettura capace di usare effettivamente un processore come un insieme di più unità di esecuzione in parallelo. Il cuore di questo approccio è il **concetto di parallelismo esplicito**. In fase di compilazione, le istruzioni vengono assemblate in modo da favorire la parallelizzazione inserendo direttamente all'interno del formato dell'istruzione le informazioni relative alla dipendenza reciproca (vedi dopo il campo di template nel par. 4.2) [2]. Quello che avviene è che il compilatore analizza il codice del programma e crea gruppi di istruzioni indipendenti fra loro che quindi possono essere eseguite in parallelo.

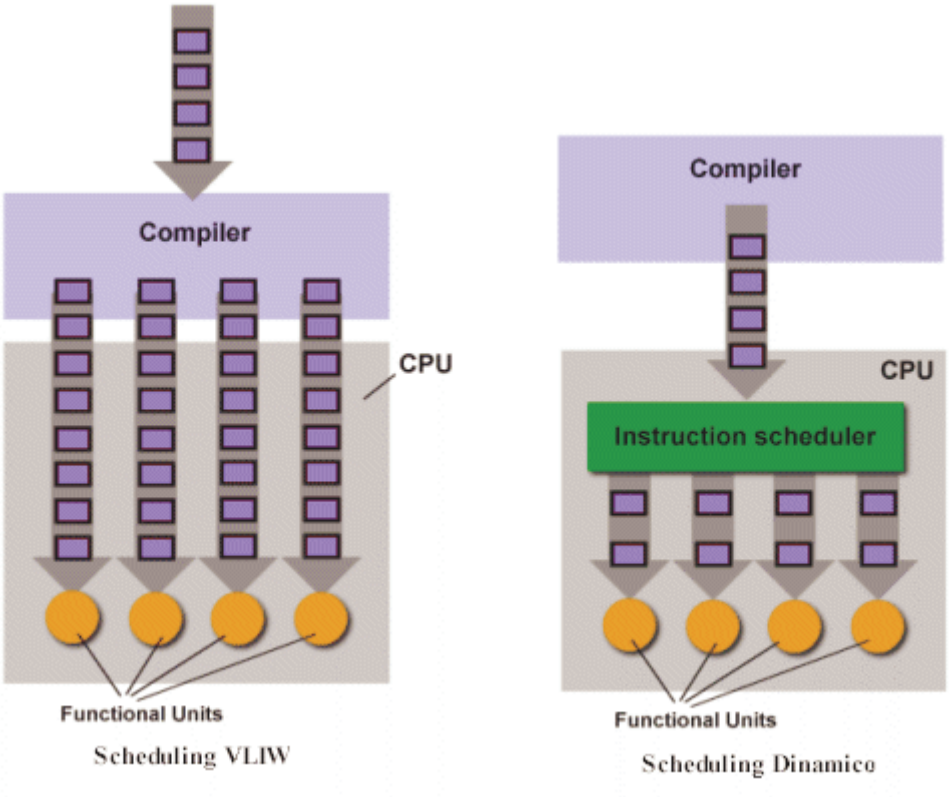

Figura 6: Scheduling VLIW e dinamico tradizionale [3]

In questo modo il compilatore **schedula staticamente** le istruzioni al momento della compilazione (*compile-time*), piuttosto che farle **schedulare dinamicamente** dal processore al momento dell'esecuzione (*run-time*). Quindi il processore sfrutta queste informazioni per parallelizzare direttamente l'esecuzione delle istruzioni. Uno dei vantaggi di questo approccio è che i processori EPIC non necessitano di una circuiteria troppo complessa come nel caso dell'esecuzione fuori ordine dei processori superscalari. Se fino ad oggi il processore aveva pochi nanosecondi per determinare una potenziale esecuzione in parallelo, oggi il compilatore può riordinare meglio il codice avendo a disposizione molto più tempo e le istruzioni di tutto il programma. Come predicato ormai da molti anni

dalla filosofia RISC, possiamo notare come anche in questo caso la complessità si sposti dall'hardware al software [15].

4.2 Formato delle istruzioni

Secondo la filosofia VLIW, l'IA-64 definisce un pacchetto da 128-bit (**bundle**) che contiene tre istruzioni, chiamate sillabe (**syllables**), più un campo di **template**.

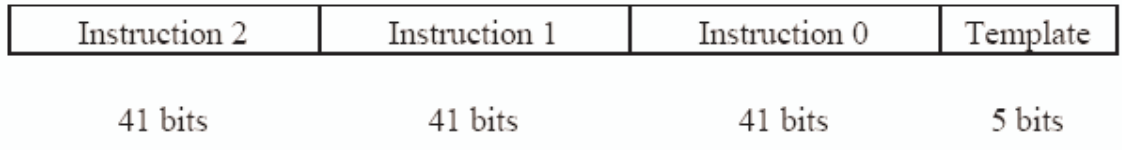

Figura 7: Bundle

Ma al contrario di quanto avviene nei processori VLIW, qui si possono prendere uno o più bundle alla volta per essere eseguiti in concorrenza fra loro. Il campo di template contiene informazioni su quali istruzioni possono essere eseguite in parallelo. Ma l'interpretazione di tale campo non rimane confinata al singolo bundle, ma va attribuita a più bundle. Per esempio, per eseguire otto istruzioni in parallelo (su altrettante unità di esecuzione parallele), il compilatore riordinerà le otto istruzioni in modo che queste si trovino in bundle attigui e setterà il campo di template in modo che il processore riesca a sapere che tutte e otto le istruzioni sono indipendenti fra loro e quindi possono essere parallelizzate su pipeline diverse. Quindi le istruzioni nei bundle non sono nell'ordine originale del programma. Inoltre, grazie alla flessibilità del campo di template, il compilatore può mischiare istruzioni indipendenti e dipendenti nello stesso bundle [15].

In particolare il campo di template può assumere i seguenti significati [15]:

1. Il campo specifica la mappatura delle istruzioni per le corrispondenti unità di esecuzione richieste;

2. Il campo indica la presenza di uno stop quando fra le istruzioni presenti all'interno dello stesso bundle esiste un certo tipo di dipendenza e quindi non è possibile parallelizzarle tutte insieme.

Ogni istruzione ha un formato di lunghezza fissata di 41 bit (approccio RISC); 14 dei quali servono ad identificare l'operazione da compiere, i successivi 21 bit servono ad indirizzare fino a tre registri come operandi, ed infine gli ultimi 6 bit possono essere usati per indicare il predicato dell'operazione.

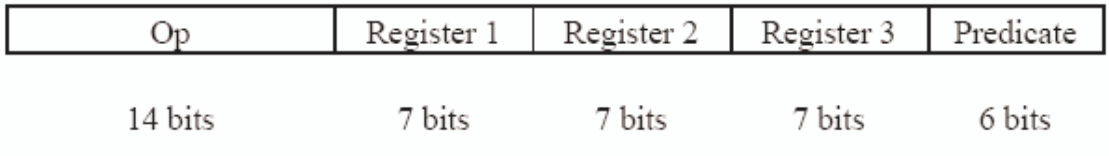

Figura 8: Syllable

Questo formato è alquanto lungo rispetto alla lunghezza tradizionale delle istruzioni delle attuali macchine superscalari RISC e CISC, anche se bisogna pensare al fatto che ogni micro-operation del Pentium IV è lunga ben 118 bit [15].

Il numero così elevato di bit necessario è legato al fatto che l'IA-64 fa uso di un numero più alto di registri rispetto ad una tipica macchina RISC (128 registri general purpose e 128 floating-point), e per la tecnica della predicazione delle diramazione sono previsti altri 64 registri per i predicati [15].

Template	Slot 0	Slot 1	Slot 2
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

Figura 9: Instruction set mapping [11]

Tutte le istruzioni includono un campo principale di operazione, detto **Op,** la cui interpretazione dipende dal campo di template e che discrimina fra 16 possibilità di processare le istruzioni in unità di esecuzione differenti (Integer, Memory, Branch e Floating-point instruction) [15].

Come in ogni altro instruction set, esiste un particolare linguaggio assembler; il compilatore, e più nello specifico l'assemblatore, traducono ogni istruzione del linguaggio di programmazione utilizzato in una serie di istruzioni IA-64 da 41 bit. Il formato generale di un istruzione in linguaggio assembler è [15]:

[qp] mnemonic[.comp] dest=srcs

dove

qp Specifica il bit del predicato usato per l'interpretazione dei salti. Se il valore è 1 (vero) durante l'esecuzione, allora il risultato dell'istruzione viene memorizzato, altrimenti questo viene scartato. Molte istruzioni IA-64 hanno bisogno del bit di predicato ma non tutte, perciò se qp=0 allora il registro dei predicati conterrà il valore costante 1.

mnemonic Specifica il nome dell'istruzione IA-64.

comp Completa l'istruzione quando necessario.

dest Specifica uno o più operandi destinazione a seconda del tipo di operazione.

srcs Specifica uno o più operandi sorgente a seconda del tipo di operazione.

4.3 Predicated Execution

Come si è visto negli ultimi anni, l'esecuzione parallela delle istruzioni è fortemente ostacolata da salti e chiamate a procedure, a causa della natura stessa delle unità di calcolo, organizzate secondo una struttura a pipeline. Infatti, in presenza di un salto tutte le istruzioni parzialmente eseguite presenti nella pipeline devono essere cancellate, si parla allora di **impredicibilità dei salti** perché fino a che l'istruzione di salto non viene completata dall'ultimo stadio della pipeline, non è noto quale sarà l'istruzione successiva e quindi non è possibile caricare nella pipeline la giusta sequenza di istruzioni da eseguire (formazione di "bolle" all'interno delle pipeline) [2].

In una architettura speculativa tradizionale, come quella dell'IA-32, si usa la predizione dei salti per stimare quale sia la path di istruzioni più probabili da eseguire. Comunque alcuni salti rimangono ugualmente difficili da predire. Una soluzione a questo problema è quella di eseguire entrambi i blocchi di istruzione che possono essere eseguiti dopo l'istruzione di salto condizionale. La **predicated execution** può essere usata per implementare l'esecuzione di più path di istruzioni contemporaneamente. Infatti una architettura a predicati può caricare ed eseguire istruzioni a cui sono associati dei predicati. Questi predicati indicano se l'istruzione può considerare valido il proprio risultato ai fini dell'esecuzione del programma. La **predication** di salti condizionati riduce il numero di salti da eseguire, eliminando la possibilità di misprediction a costo di dover eseguire istruzioni addizionali i cui risultati verranno scartati.

Da recenti studi effettuati sui processori con architettura IA-32 è risultato che oltrepassando le quattro istruzioni per ciclo si ha una diminuzione dei benefici fintanto che

il compilatore e l'architettura non migliorino. Infatti, attualmente, su un processore come il Pentium IV dove si possono eseguire fino a 6 istruzioni per ciclo, nella realtà se ne processano in media una o al massimo due [20]. Questo vuol dire che più del 70% delle potenzialità del processore spesso non vengono neanche utilizzate. Il maggiore motivo di questa degradazione delle prestazioni è dovuto alla dimensione ristretta dei blocchi di istruzione e alla grossa penalizzazione che si ha nel caso di misprediction [20]. Questo è uno dei motivi per il quale nell'IA-64 si è deciso di non implementare pipeline troppo lunghe (10 stadi nell'Itanium), anche se ciò è andato a scapito della frequenza di clock dei primi Itanium (massimo 800 MHz, mentre nell'Itanium 2 si è giunti a 1,5 MHz).

Una soluzione per mettere subito in pratica questa tecnica sarebbe quella di usare una *dynamic prediction* che assume che non ci sia supporto dei predicati nell'ISA (come nel caso dell'IA-32). I salti che possono beneficiare della predication execution vengono individuati dal compilatore che converte le istruzioni in modo tale che durante l'esecuzione del programma vengano creati dei predicati dinamici che modifichino la sequenza di esecuzione delle istruzioni [20].

Ma nell'IA-64 si è approfittati dell'abbandono dell'architettura x86 per implementare una predicated execution molto più performante direttamente nell'hardware della nuova ISA, con una logica di controllo certamente più semplice di quella di branch prediction. Il compilatore genera delle sequenze di codice che, quando eseguite, sono capaci di inserire rilevanti informazioni e statistiche di esecuzione nel bit di condizione, detto **predicato.** Quindi durante la fase di compilazione, il compilatore elimina i salti dal programma e li sostituisce con esecuzioni condizionali. Durante l'esecuzione, l'hardware utilizza queste informazioni per fare delle predizioni. E' possibile riassumere i punti fondamentali di questa nuova tecnica in tre concetti:

- I salti sono basati sui bit dei predicati a livello di architettura;
- Il meccanismo di predizione si basa sui bit dei predicati a livello di micro-architettura;
- La generazione del codice viene supportata a livello di compilatore.

A livello di architettura è stato introdotto un set di istruzioni di salto che basa le proprie decisioni sui predicati. Ogni salto richiede che sia stata eseguita precedentemente un istruzione che setti il bit del predicato corrispondente. Il costo derivante dall'uso dei predicati e dall'introduzione di appositi registri ad un bit sono quasi sempre compensati dalla riduzione dei salti.

Un tipico esempio in un linguaggio di alto livello è dato dalla sequenza di istruzione **if-then-else**. Un compilatore tradizionale inserisce un salto condizionale nel punto del costrutto in cui c'è l'istruzione if [15].

Se la condizione if restituisce come risultato il valore 1 (vero), allora non viene effettuato il salto e viene eseguito il blocco di istruzioni successivo all'istruzione if, indicato dall'istruzione then; alla fine di tale blocco si effettua un salto incondizionato sul blocco indicato dall'istruzione else. Se la condizione if restituisce come risultato il valore 0 (falso), allora si salta il blocco di istruzioni del then e si esegue direttamente il blocco else. I due flussi di istruzioni si congiungono insieme dopo la fine del blocco di istruzioni dell'else [15].

Invece, un compilatore IA-64 si comporta in questo modo [15]:

1. Nel punto dell'istruzione if, inserisce una istruzione di comparazione che crea due predicati. Se la comparazione risulta vera, il primo predicato p1 è settato vero ed il secondo p2 falso; in caso contrario, se la comparazione risulta falsa vale il viceversa;

2. Viene aggiunta ad ogni istruzione del blocco then un registro dei predicati che contiene il valore del primo predicato p1, lo stesso viene fatto per ogni istruzione del blocco else con il valore del secondo predicato p2;
3. Il processore esegue entrambi i blocchi di istruzioni then ed else su pipeline differenti. Quando si trova il risultato della comparazione if, il risultato del blocco di istruzioni con bit dei predicati settato false viene scartato (come se si trattasse di una nop – no operation), mentre il processore può memorizzare il risultato ottenuto dal blocco di istruzioni avente il bit dei predicati settato true.

Per comprendere meglio il funzionamento dei predicati, consideriamo un semplice esempio. Partendo dal codice sorgente scritto in C:

```
if ( a && b ) then
      j = j + 1 ;
else
      if ( c ) then
            k = k + 1 ;
      else
            k = k - 1 ;
i = i + 1;
```

Il corrispondente codice assembler del Pentium IV potrebbe essere:

```
      cmp a, 0 ;  //compara a con zero
      je L1 ;     //salta al punto L1 se a = 0
      cmp b, 0 ;
      je L1 ;
      add j, 1 ;  //j = j + 1
      jmp L3 ;
L1:   cmp c, 0 ;
      je L2 ;
      add k, 1 ;  //k = k + 1
      jmp L3 ;
L2:   sub k, 1 ;  //k = k - 1
L3:   add i, 1 ;  //i = i + 1
```

Le istruzioni assembler non verranno eseguite nell'ordine in cui si trovano, perchè in realtà un processore come il Pentium IV implementa via hardware un algoritmo di esecuzione fuori ordine che cerca di parallelizzare il tutto.

Quindi il corrispondente codice assembler dell'IA-64 è:

```
      cmp.eq p1 , p2 = 0 , a ;;     //compara a con lo 0, se vero pone
                                    //p1 = 1 e p2 = 0
(p2)  cmp.eq p1 , p3 = 0 , b
(p3)  add j = 1 , j
(p1)  cmp.ne p4 , p5 = 0, c         //compara c con lo 0, se vero pone
                                    //p4 = 0 e p5 = 1
(p4)  add k = 1 , k
(p5)  add k = -1 , k
      add i = 1 , i
```

Per quanto riguarda le istruzioni assembler dell'IA-64, queste sono divise in due gruppi dal doppio punto e virgola: questo significa che le istruzioni presenti in ogni gruppo sono indipendenti fra loro e quindi potranno essere parallelizzate tra loro [15].

A differenza di quanto avviene con un compilatore tradizionale, questa tecnica permette al processore di eseguire le istruzioni presenti su entrambe le diramazioni then/else contemporaneamente senza dover attendere il risultato della comparazione. Quindi non viene mai alterato il flusso originale di istruzioni, la pipeline non si blocca mai e al massimo alcune istruzioni vengono scartate, ma ciò non rappresenta un problema grazie all'alto grado di parallelismo presente [2].

4.4 Speculative Loading

Per risolvere efficacemente il problema della latenza della memoria, nell'IA-64 il nuovo set di istruzioni permette di istanziare esplicitamente un precaricamento dati ed una verifica automatica delle dipendenze di memoria, con tanto di codice di recupero in caso di perdita di coerenza del dato. Ancora una volta la differenza rispetto ai processori tradizionali è la presenza in hardware di risorse molto sofisticate, controllabili direttamente a livello di codice attraverso uno specifico set di istruzioni. Mentre prima il controllo di tali risorse doveva essere realizzato in hardware, adesso viene demandato tutto al compilatore, con risultati nettamente superiori [2].

Questo procedimento prende il nome di speculative loading; andiamo ora ad analizzare le due diverse situazioni di esecuzione speculativa: **control speculation** e **data speculation**.

4.4.1 Control Speculation

Questa tecnica permette al processore di caricare (fase di load) i dati dalla memoria prima ancora che il programma ne abbia effettivamente bisogno, evitando i lunghi ritardi di latenza delle memorie. E' inutile dire che la riduzione delle latenze produce un notevole aumento delle prestazioni, perché la memoria (sia essa una cache di primo o secondo livello) è molto più lenta del processore, e ritardi nel prelievo dei dati dalla memoria costituiscono un importante collo di bottiglia per le prestazioni della CPU. Per minimizzare tali ritardi si dovrebbe riordinare il codice dei programmi in modo da caricare i dati quanto prima possibile, questo può essere fatto da ogni compilatore limitatamente alle dipendenze dei dati che si possono incontrare: non si può anticipare l'operazione di load che si riferisce ad un salto perché il caricamento potrebbe non dover mai servire! Nell'IA-64 possiamo anticipare l'istruzione di load usando i predicati in modo che i dati possano essere recuperati dalla memoria ma non memorizzati in alcun registro fino a quando non si conosca il valore del predicato (mentre nell'IA-32 si usava la predizione dei salti). Il problema con questa tecnica è che l'operazione di load può crashare, cioè può essere generata un'eccezione dovuta ad un indirizzo di memoria non valido o a un page fault. Se questo accade, il processore dovrebbe occuparsi opportunamente di tale errore causando un ulteriore ritardo [15].

Quindi è preferibile che il processore rinvii la notifica di eventuali eccezioni fino al momento in cui questo diventi effettivamente necessario.

Perciò la soluzione implementata nell'IA-64 separa il funzionamento dell'istruzione di load dall'eccezione. L'istruzione di load originale viene sostituita da due istruzioni [15]:

1. Un istruzione di **speculative load** esegue il prelievo (fase di fetch) dei dati da memoria, eseguendo una ricerca delle eccezioni generate, ma senza notificarle. Questa istruzione, naturalmente viene eseguita in un'appropriata posizione, precedente a quella dell'istruzione originale;
2. Un **istruzione di check** rimane nel punto dell'istruzione di load originale e notifica le eventuali eccezioni. Se l'istruzione di load faceva parte di una diramazione potrebbe essere presente un predicato che indica se l'operazione dovrà essere eseguita oppure no.

Nel caso in cui l'istruzione di speculative load generi un'eccezione, viene settato vero il **Not a Thing (NaT)** bit nel registro di destinazione. Se l'istruzione di check viene eseguita e il Nat bit è settato vero, l'istruzione di check salta ad una routine del sistema operativo che impugna l'eccezione [15].

Come esempio consideriamo il seguente codice assembler:

```
(p1)  br L1
      ld8 r1 = [r5] ;;
      add r2 = r1 , r3
```

Quindi il compilatore dell'IA-64 potrà riscrivere il codice nel seguente modo:

```
      ld8.s r1 = [r5] ;;      //speculative load
      …
(p1)  br L1
      chk.s r1 , recovery     //nel caso di un'eccezione salta al
                              //punto recovery
      add r2  = r1 , r3
```

4.4.2 Data Speculation

Se con il control speculation, l'istruzione di load viene anticipata nella sequenza originale del codice del programma controllando se questa solleva un'eccezione, il data speculation anticipa l'istruzione di load prima di un istruzione di store che ha la possibilità di alterare l'informazione contenuta nella stessa locazione di memoria sorgente dell'operazione di load. Dunque, è necessario assicurarsi che l'operazione di load riceva l'informazione corretta. Come in precedenza l'istruzione di load viene sostituita da due istruzioni [15]:

1. Un istruzione di **advanced load** che carica anticipatamente i dati rispetto all'ordine originale del codice del programma. Inoltre questa istruzione scrive l'indirizzo sorgente in una struttura hardware detta **Advanced Load Address Table (ALAT)**. A questo punto ogni istruzione di store successiva controlla che il suo indirizzo di destinazione non sia presente fra quelli contenuti nell'ALAT, in caso affermativo tale indirizzo viene rimosso dall'ALAT;
2. Un **istruzione di check** rimane nel punto dell'istruzione di load originale e controlla che l'indirizzo sorgente sia ancora presente nella tabella ALAT. Se viene trovato, nessuna istruzione di store ha alterato l'indirizzo di memoria sorgente del load, e quindi il programma può continuare ad essere eseguito normalmente. Mentre, se l'istruzione di check non trova l'indirizzo corrispondente all'interno dell'ALAT, allora significa che è stata eseguita un'istruzione di store in corrispondenza dello stesso indirizzo dell'operazione di load, perciò questa deve essere eseguita nuovamente per ottenere il risultato corretto.

Consideriamo il seguente codice assembler:

```
      st8 [r4] = r12
      ld8 r6 = [r8] ;;
      add r5 = r6 , r7 ;;
      st8 [r18] = r5
```

Il compilatore potrà scegliere di riscrivere il codice nel seguente modo:

```
      ld8.a r6 = [r8] ;;      //advanced load
      …
      st8 [r4] = r12
      ld8.c r6 = [r8]         //check load
      add r5 = r6, r7 ;;
      st8 [r18] = r5
```

Stesso discorso vale nel caso di istruzione di dipendenza di dati fra l'istruzione di load ed altre istruzioni.

```
      ld8.a r6 = [r8] ;;      //advanced load
      …
      add r5 = r6 , r7
      …


      st8 [r4] = r12
      chk.a r6 , recover      //check, in caso di eccezione salta al
                              //punto recover
back: st8 [r18] = r5         //ritorno dal salto
```

Come è semplice notare, questa tecnica risulta efficace solo nel caso in cui le istruzioni di load e store abbiano poche chance di sovrapporsi (e ciò viene supposto in architetture con ampio numero di registri visibili al programmatore, come nel caso dell'IA-64) [15].

4.5 Software Pipelining

Un'altra pratica comune nella programmazione attuale è quella che fa uso di un numero considerevole di loop, perciò una loro ottimizzazione porterebbe grossi vantaggi in tutto il software esistente.

Prendendo in considerazione un ciclo di istruzioni, ci si accorge come queste non siano normalmente parallelizzabili perché l'iterazione x deve essere eseguita sempre prima che cominci l'iterazione x+1. Nell'IA-64 è possibile realizzare il parallelismo raggruppando insieme istruzioni appartenenti a iterazioni differenti ed usando registri temporanei diversi da quelli normalmente usati nel ciclo di istruzioni per evitare possibili conflitti [15].

La **software pipeline** ha tre fasi [15]:

1. Durante la **fase di prologo**, una nuova iterazione viene avviata ad ogni ciclo di clock e la pipeline gradualmente si riempie;
2. Durante la **fase di kernel**, la pipeline è piena e si raggiunge così il massimo grado di parallelismo: una nuova iterazione comincia mentre un'altra viene completata (nella figura che segue si rappresenta un ciclo composto da tre istruzioni con una pipeline della profondità di 4 stadi);
3. Durante la **fase di epilogo**, ogni iterazione viene completata ad ogni ciclo di clock successivo fino al completamento dell'ultima iterazione.

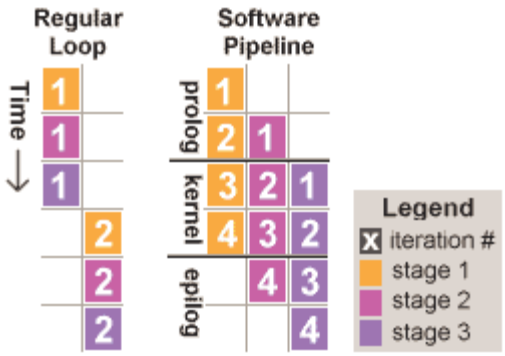

Figura 10: Software Pipeline [2]

L'IA-64 offre istruzioni e registri appositi per la gestione dei loop. Durante la fase di prologo e di kernel, ad ogni iterazione se il registro Loop Count (LC) è maggiore di zero viene decrementato e si procede ad una nuova iterazione; nella fase di epilogo, quando LC = 0 viene decrementato il registro Epilog Count (EC); EC = 0 quando la pipeline è completamente vuota.

Il software pipelining solleva il compilatore ed il programmatore dalla responsabilità di assegnare i giusti registri temporanei. Questo permette di contenere la grandezza del codice nel caso di lunghi cicli.

Quindi, a supporto di tale tecnica abbiamo [15]:

- **Automatic register renaming**, che permette di gestire una finestra dei registri necessari al ciclo;
- **Predication**;
- **Special loop terminating instruction**, che permettono di far ruotare i registri e di decrementare il registro LC ad ogni ciclo.

4.6 Instruction Set Architecture

Come già accennato, l'architettura IA-64 ha un approccio di tipo RISC col numero di registri, che è salito vertiginosamente rispetto all'IA-32. La maggiore quantità di risorse interne ha lo scopo di ridurre al minimo le dipendenze reciproche tra le istruzioni (se due istruzioni usano registri diversi non ci potranno essere problemi di dipendenza) e gli scambi di dati con la memoria (con molti registri non accadrà molto spesso di dover salvare il loro contenuto in memoria per avere qualche registro libero a disposizione).

Segue un elenco dei registri visibili dalle applicazioni (ma non sempre scrivibili) [15]:

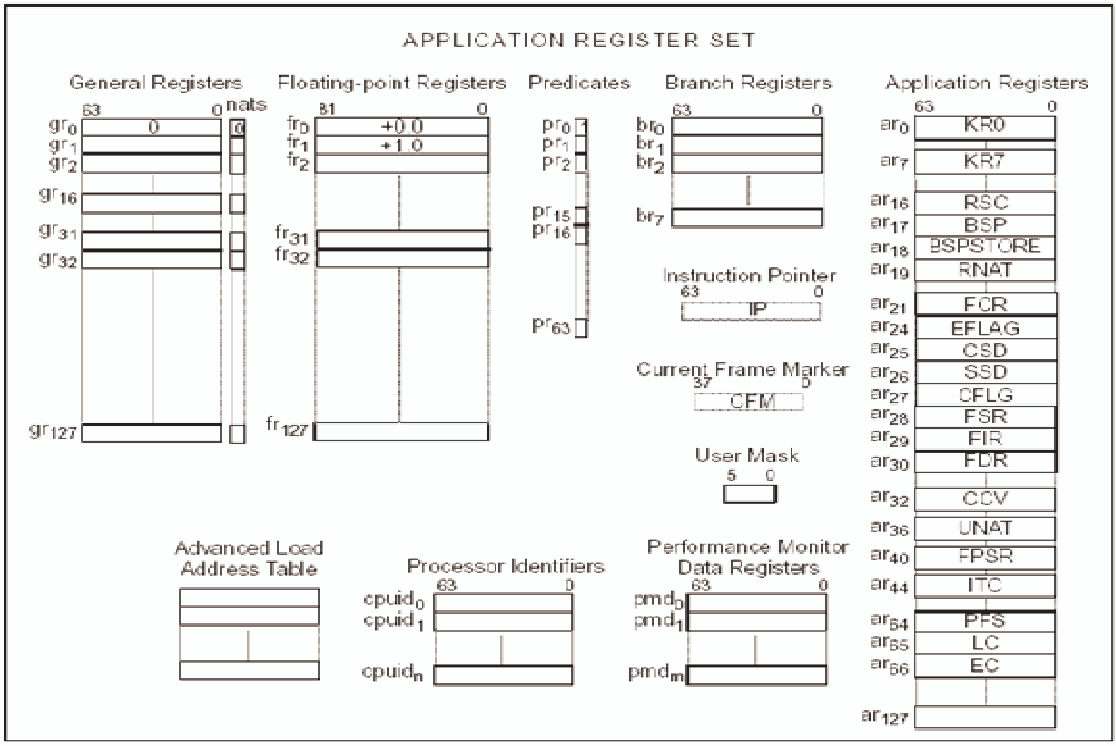

Figura 11: Registri disponibili nell'Itanium [9]

- **Integer Registers**: 128 registri a 64-bit general purpose, sia per numeri interi che per altri usi. Associato ad ogni registro c'è un NaT bit usato per la gestione delle eccezioni generate con la control speculation. I registri da gr0 a gr31 sono statici e possono essere riferiti direttamente dai programmi. I registri da gr32 a gr127 sono rotanti e vengono usati per il software pipelining e per il register stack (vedi par. 4.7), il riferimento a questi registri è del tutto virtuale e l'hardware rinomina questi registri dinamicamente;
- **Floating-Point Registers**: 128 registri a 82-bit per numeri in virgola mobile. Anche qui i registri da fr0 a fr31 sono di tipo statico, mentre i registri da fr32 a fr127 possono essere usati come registri rotanti per il software pipelining;
- **Branch registers**: 8 registri a 64 bit per i salti;
- **Predicate Registers**: 64 registri ad 1 bit usati per i predicati. Il registro pr0 è sempre settato ad 1 per le istruzioni senza predicato. I registri da pr0 a pr15 sono statici, mentre i registri da pr16 a pr63 sono rotanti e possono essere usati per il software pipelining;
- **Instruction pointer (IP)**: contiene l'indirizzo dell'istruzione corrente IA-64 in esecuzione;
- **Current frame marker (CFM)**: contiene le informazioni relative alla finestra e alla rotazione dello stack di registri di tipo general purpose, floating-point e predicate;
- **User mask**: contiene 6 bit usati per monitorare le prestazioni e l'uso dei registri in virgola mobile;
- **Performance monitor data registers**: usato come supporto al monitoraggio delle prestazioni dell'hardware;

- **Processor identifiers**: descrive le funzioni implementate dal processore;
- **Application registers**: una collezione di registri special-purpose, usati anche per register stack e software pipelining.

4.7 Register Stack Engine (RSE)

Insieme alle macchine anche i sistemi operativi ed il software in generale è cresciuto di complessità, rendendo tra l'altro necessario lo sviluppo di nuovi sistemi di programmazione di tipo modulare (programmazione ad oggetti). Questa caratteristica, ben vista dal programmatore che riesce a controllare e riusare meglio il codice che produce, ha portato a programmi che mediamente hanno molti più salti e chiamate a subroutine. Ma la chiamata a subroutine rappresenta una alterazione del flusso di esecuzione del processore, che quindi si trova a dover lavorare in una condizione non ottimale. Ma a parte il problema dovuto al blocco e al caricamento della pipeline, ci sono anche altre problematiche accessorie che il processore deve gestire, per esempio deve provvedere a salvare (per poi ripristinare) lo stato interno dei registri e permettere il passaggio di parametri fra le subroutine [2].

Nell'IA-64 è previsto il **meccanismo a stack rotante** che evita inutili spostamenti dei dati tra i registri alla chiamata e al ritorno di ogni procedura. Questo meccanismo fornisce automaticamente una nuova finestra (**frame**) di un massimo di 96 registri consecutivi (nel caso generico, da r32 a r127) ad ogni nuova procedura chiamata. Il compilatore specifica il numero di registri richiesti dalla procedura chiamata tramite un'istruzione di allocazione, la quale specifica quanti di questi siano locali (usati solo all'interno della procedura stessa) e quanti sono di output (usati per passare parametri ad altre procedure chiamate). Quando avviene la chiamata di una procedura, la finestra scorre in avanti e l'hardware rinomina i registri in modo che i registri locali della precedente finestra siano nascosti e i registri di output della procedura chiamante partano dal registro r32 nella finestra della procedura

chiamata diventando i registri locali della procedura chiamata. I registri fisici tra r32 e r127 vengono assegnati da una logica di controllo del banco circolare di registri virtuali associati alla procedura (il cerchio si chiude fra i registri r127 e r32). Al ritorno della procedura chiamante, la finestra scorre all'indietro ripristinando la finestra precedente [15]. Ciascuna procedura non ha bisogno di conoscere la posizione relativa della propria finestra, in quanto all'interno di essa i registri vengono rinominati in modo da farli apparire alla procedura virtualmente come i primi dello stack (in questo modo il codice non deve gestire la posizione della finestra) [2].

Quando necessario, l'hardware sposta il contenuto dei registri nella memoria per liberare altro spazio fra i registri quando una nuova procedura lo richieda, e carica il contenuto dalla memoria ai registri quando si ritorna alla procedura chiamante [15].

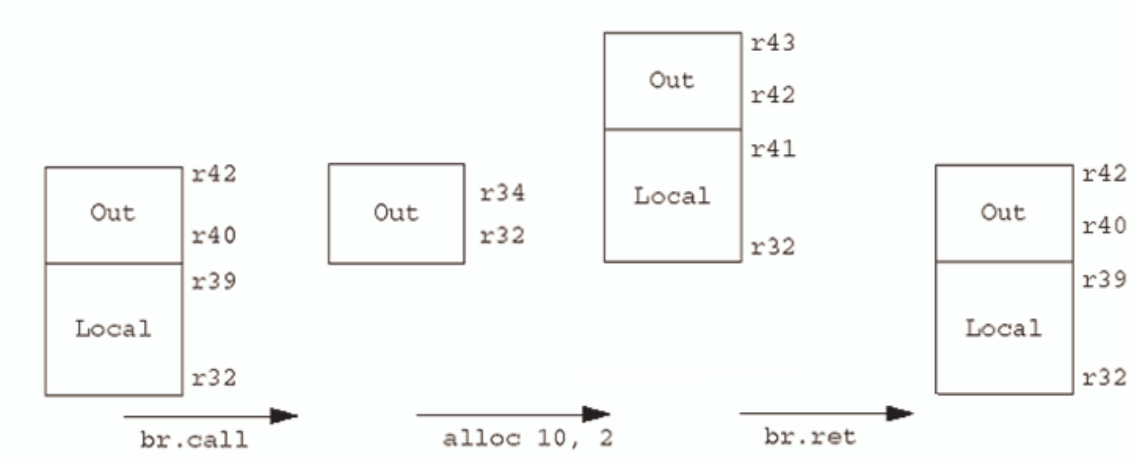

Figura 12: Register Stack

Inoltre, il meccanismo di Register Stack Engine può anche essere implementato in modo speculativo, cioè nei momenti di inutilizzo della memoria centrale si può effettuare un salvataggio/ripristino dei registri più vecchi (quelli più lontani dalla posizione della finestra

corrente). In altre parole, man mano che la finestra avanza, la logica di controllo libera dei registri davanti ad essa per permettere altri avanzamenti, viceversa man mano che torna indietro il valore dei vecchi registri viene ripristinato. In conclusione il risultato di tutto ciò è una rotazione di registri virtualmente infinita [2].

Per mettere in atto questa tecnica l'istruzione di allocazione deve utilizzare gli operandi **sof (size of frame)** e **sol (size of locals)** per specificare il numero richiesto di registri. Questi valori sono memorizzati nel registro **CFM**. Quando si chiama una nuova procedura, i valori di sof e sol vengono spostati nel registro **PFS (Previous Function State)**. Al ritorno della procedura chiamante i valori di sof e sol vengono trasferiti dal PFS al CFM. Per permettere chiamate annidate, i valori precedenti di PFS vengono sempre salvati ad ogni nuova chiamata, per essere poi ripristinati al ritorno. Questa operazione è assolta dall'istruzione di allocazione che designa un registro general purpose per contenere il valore corrente dei campi di PFS prima che questi vengano sovrascritti [15].

Il registro CFM descrive lo stato della corrente finestra dello stack di registri generici, associati con la corrente procedura attiva. I campi di questo registro sono [15]:

- **Sof**: indica la grandezza della finestra;
- **Sol**: indica la grandezza della porzione di registri locali;
- **Sor**: indica la grandezza della porzione rotante della finestra attuale, questo è un sottoinsieme della porzione locale che viene dedicato al software pipelining;
- **Register Rename Base Values**: indica la quantità di registri usati nella rotazione tra quelli general purpose, floating-point e predicate.

Mentre il registro PFS contiene i seguenti campi:

- **Pfm**: contiene il frame maker precedente, cioè tutti i campi del CFM precedente;
- **Pec**: contiene il conteggio di epilogo;

- **Ppl**: contiene la prerogativa di livello.

4.8 Organizzazione interna della famiglia Itanium

Il processore Intel Itanium è stato il primo ad implementare l'architettura IA-64. L'organizzazione interna dell'Itanium unisce le caratteristiche di un tipico processore superscalare al supporto della tecnologia EPIC. Tra le caratteristiche superscalari troviamo una pipeline a 6 vie a 8 stadi, il prefetch dinamico, branch prediction e un tabellone dei registri per ottimizzare la compilazione non deterministica. La parte hardware legata ai nuovi concetti introdotti dall'EPIC include l'esecuzione dei predicati, lo speculative loading e il software pipelining [15].

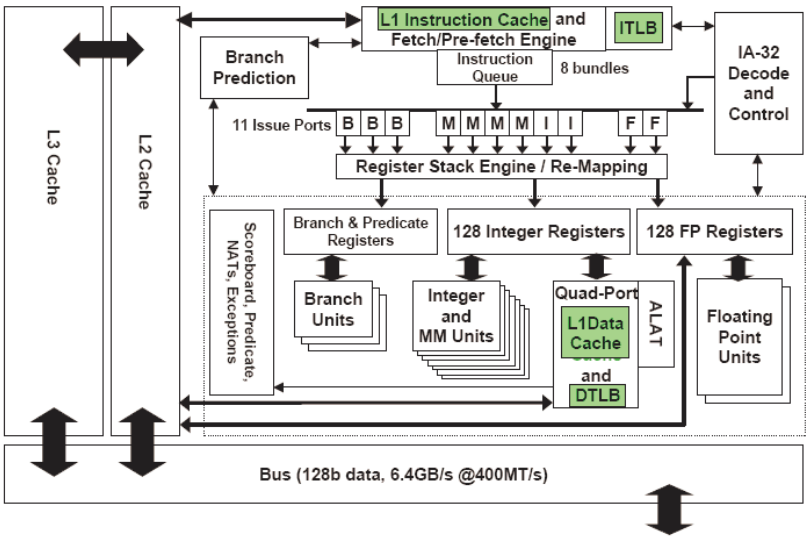

Figura 13: Block diagram dell'Itanium 2 [12]

Come evidenziato in figura, l'Itanium 2 ha undici porte per altrettante unità di esecuzione: due ad interi (I), quattro di load e store tra registri e memoria (M) (le due unità I e le quattro unità M possono processare 6 istruzioni ALU alla volta), due in virgola mobile (F)

e tre unità di esecuzione dei salti (B). Le istruzioni vengono caricate attraverso una cache di istruzioni L1 e vengono accodate in un buffer che supporta fino a 8 bundle contemporaneamente (per un totale di ben 24 istruzioni). Una volta decise le assegnazioni delle istruzioni alle rispettive unità funzionali, il processore processa almeno due bundle alla volta, per un totale di massimo 6 istruzioni per ciclo di clock [14].

L'organizzazione dell'Itanium è spesso più semplice di quella dei processori superscalari attuali, infatti la cresciuta complessità negli attuali microprocessori è uno dei motivi che ha spinto l'Intel a cambiare completamente architettura. L'Itanium non utilizza buffer di riordino perché utilizza la speculation loading. Il meccanismo di stack rotante è più semplice della tecnica di register renaming del Pentium IV. La circuiteria dell'algoritmo fuori ordine è stata sostituita dal parallelismo esplicito inserito già nel software nella fase di compilazione [15].

Usando il branch prediction, la fase di fetch/prefetch può caricare in modo speculativo nella cache di istruzioni L1 in modo da diminuire il tasso di *cache miss* (tasso che indica quante volte l'istruzione richiesta non è stata trovata all'interno della cache). Dopodichè, il flusso di istruzioni richieste viene impacchettato in bundle messi in coda verso le unità di esecuzione [15].

Si usano tre livelli di cache [14]:

- La cache L1 (on-chip) è divisa in 16 KByte per le istruzioni e 16 KByte per i dati, ognuna a 4 vie associative con linee a 32 byte;
- La cache L2 (on-chip) da 96 KByte è a 6 vie associative con linee a 64 byte;
- La cache L3 (off-chip, ma comunque nello stesso package) da 4 MByte è a 4 vie associative con linee a 64 byte (la scelta di implementare una quantità tale di cache è stata dettata dal fatto che il chipset 460GX supporta solo PC100 SDRAM).

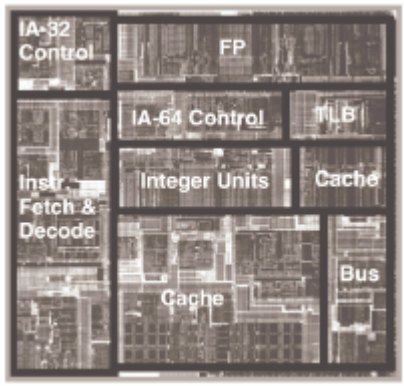

Figura 14: Core dell'Itanium [21]

Nelle versioni successive dell'Itanium 2 la cache L2 è stata portata alla dimensione di 256 KB, inoltre adottando il chipset E8870 è stato possibile utilizzare la DDR a 400 MHz, ciò ha permesso di usare un quantitativo inferiore di cache L3 (minimo 1.5 MB), anche se per alcune applicazioni server si è preferito aumentarla ancora fino a 6 MB [14].

Superato il limite di indirizzamento imposto dai vecchi registri general purpose a 32-bit sarebbe possibile indirizzare fino a 18 milioni di byte, ma gli indirizzi di memoria sono stati limitati a 44 bit perciò per ora sarà possibile indirizzare "solo" fino a 17,6 TByte, che per ora sembrano più che sufficienti.

Oltre a tutte le componenti previste dall'architettura IA-64 è stata prevista un'area dove alloggiare un'unità di decodifica e controllo IA-32, che rende l'Itanium capace di eseguire il buon vecchio codice x86 tramite emulazione. In realtà, questa è stata soprattuto una grossa mossa commerciale per cercare di garantire agli utenti finali di effettuare la migrazione verso questa nuova architettura nel modo più graduale ed agevole possibile.

Infatti, l'architettura IA-64 supporta ben tre modalità di funzionamento [14]:

- **IA-32 System Environment,** supporta sistemi operativi IA-32 a 32 bit;
- **IA-64 System Environment,** supporta sistemi operativi basati sull'IA-64;

- **IA-32 mista,** supporta su sistemi operativi sull'IA-64 con applicazioni sviluppate su IA-32.

Ma dai primi test effettuati è risultato evidente che le modalità di funzionamento che garantiscono la back-compatibility con l'architettura IA-32 sono notevolmente meno prestazionale rispetto ad un moderno microprocessore come il Pentium IV.

Infatti l'esecuzione del codice x86 viene "garantita" tramite emulazione, e come ci ha già fatto capire il Crosue® di Transmeta®, le tecnologie di emulazione possono essere anche molto avanzate ma le prestazioni che se ne otterranno potranno non essere del tutto soddisfacenti.

Per risolvere tale problema, negli ultimi mesi Intel Corporation è corsa hai ripari ed ha annunciato la disponibilità del software IA-32 Execution Layer (EL) per i sistemi basati sul processore Intel Itanium 2 e sui sistemi operativi Microsoft Windows. Questa tecnologia fornisce una maggiore flessibilità aumentando le prestazioni di diverse applicazioni a 32 bit e facilitando la migrazione verso le piattaforme basate su processori Itanium 2 e su Windows.

IA-32 EL è un pacchetto software integrato nel sistema operativo che è stato sviluppato in collaborazione con la Microsoft©. Nonostante il fatto che i processori Itanium 2 prevedano il supporto per le applicazioni a 32 bit tramite hardware on-die, con il prodotto IA-32 EL, il supporto per le applicazioni IA-32 sarà invece fornito dal software. Dopo l'installazione, l'esecuzione delle applicazioni a 32 bit viene gestita in modo trasparente tramite IA-32 EL. In fase di esecuzione questo programma converte il codice applicativo IA-32 in codice nativo dell'Architettura Itanium e ne consente l'esecuzione come codice nativo nella modalità IA-64 System Environment [14].

5 Comparazione delle prestazioni

5.1 Introduzione

Una volta analizzate le principali caratteristiche delle due architetture Intel si è voluto quantificare per quanto possibile l'efficienza e l'efficacia degli sforzi compiuti dal team di sviluppo dell'IA-64.

Al momento attuale non è stato possibile avere accesso fisico ad una macchina equipaggiata con uno dei processori della famiglia Itanium, perciò si è dovuto ricorrere ad un ambiente simulato messo gentilmente a disposizione dall'HP.

Nel 1998, poco dopo l'inizio dello sviluppo del progetto della nuova architettura IA-64, presso i laboratori dell'HP è stato avviato in parallelo il progetto Linux/ia64 (progetto Trillian) avente lo scopo di effettuare il porting di un intero sistema operativo funzionante su piattaforma Linux (Red Hat 7) in tempo utile per essere utilizzato sull'Itanium appena questo fosse stato immesso sul mercato. Ad oggi, a questo progetto si può attribuire il merito di essere riuscito a realizzare il primo sistema operativo funzionante per questa architettura, senza contare il fatto che dopo diversi annunci, in casa Microsoft, Windows XP-64 non ha ancora visto venire alla luce la sua versione finale.

Nel 2000, con la definizione delle specifiche della nuova architettura, l'HP ha deciso di rendere pubblico il progetto alla comunità Open Source, anche se in realtà non è mai stato rilasciato il codice sorgente; ma anche se il simulatore è rimasto di proprietà della Hewlett-

Packard Company, questo è stato distribuito sotto una licenza che ne permette il libero utilizzo per lo sviluppo di applicazioni, kernel e per scopi accademici come il nostro.

Lo sviluppo del kernel ha seguito di pari passo quello del kernel ufficiale (http://www.kernel.org). Per il suo sviluppo si è cercato di minimizzare il più possibile i cambiamenti delle parti indipendenti dalla machina. A partire dalla versione 2.3.42 sono state messe a disposizione delle patch contenenti il nuovo codice, la maggior parte del quale si trova nei direttori *arch/ia64* e *include/asm-ia64*; a partire dalla versione 2.6 il supporto all'ia64 è stato integrato, dunque non è più strettamente necessario applicare alcuna patch, anche se è ancora possibile trovare alcune patch specifiche. Il nuovo kernel del progetto Linux/ia64 è stato pensato sin dall'inizio come un sistema operativo a 64 bit; esso utilizza il modello di ordinamento little-endian per ovvi motivi di compatibilità con l'IA-32, naturalmente con l'aggiunta di un ulteriore supporto l'IA-64 può essere utilizzata anche con i processi big-endian. Almeno per la fase iniziale di sviluppo, non essendo ancora disponibili i primi esemplari di processori si è dovuto operare su piattaforme Linux/ia32, perciò si è dovuto conservare alcune convenzioni del passato quali l'ABI (Application Binary Interface), che per esempio ha portato a mantenere immutato il valore numerico del comando *ioctl()*.

Per maggiori informazioni consultare la guida [16].

5.2 Strumenti utilizzati

Dunque con il rilascio dell'**IA-64 software development kit for Linux/ia32 (IA64SDK)**, i laboratori HP hanno messo a disposizione un ambiente di sviluppo capace di creare applicazioni IA-64 compatibili senza disporre dell'accesso hardware a tali macchine. In questo ambiente, non si può soltanto compilare applicazioni e generare binari IA-64, ma si possono anche eseguire tali binari direttamente al prompt di shell.

Il pacchetto che siamo andati ad utilizzare è dunque composto da **NUE (Native User Environment)** che ricrea l'ambiente di lavoro di un tipico sistema IA-64, con una shell dove si possono editare comandi e file, e si possono compilare ed eseguire programmi. Il cuore di questo pacchetto è l'**HP IA-64 instruction set simulator (Ski)**, che simula una CPU IA-64, ma non l'intera piattaforma (cioè non BIOS o bus PCI). Questo simulatore può operare in due modalità di esecuzione [16]:

1) **User mode**: permette di eseguire applicazioni direttamente al livello applicazione di un kernel Linux/ia32. La simulazione finisce al livello delle system call, ogni chiamata viene intercettata e tradotta nell'equivalente chiamata del sistema operativo host di tipo Linux/ia32; spesso avviene solo un passaggio di parametri da 32 a 64 bit.
2) **Kernel mode**: viene simulata l'intera CPU IA-64, compreso il comportamento della memoria virtuale e degli interrupt. In questa modalità è possibile lanciare in esecuzione un kernel Linux/ia64.

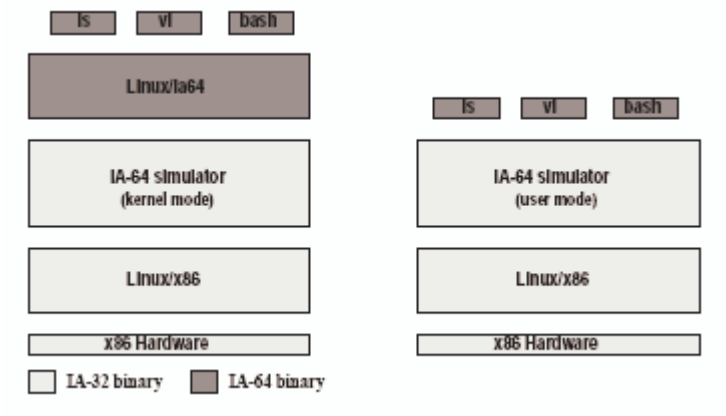

Figura 15: Modalità di esecuzione di ski [16]

Per il corretto utilizzo di ski viene fornito un modulo del kernel chiamato **binfmt_misc** (avviabile tramite il servizio **ia64fmt**), che permette di collegare dinamicamente l'interprete dei comandi ai rispettivi binari. NUE, da parte sua, fornisce una piattaforma di sviluppo incrociato, completo di compilatore gcc, linker, compilatore assembler, librerie e file di header; tutto è facilmente accessibile nelle consuete posizioni grazie all'adozione di un ambiente chrooted.

Per lo sviluppo kernel, l'IA64SDK contiene anche una piccola immagine disco di tipo raw contenente una distribuzione semplificata di TurboLinux/ia64 alpha utile al boot di kernel Linux/ia64.

Purtroppo, con l'immissione sul mercato dei primi esemplari di processori Itanium e delle prime patch ufficiali per kernel, lo sviluppo di NUE è stato fermato, perché si è ritenuto raggiunto completamente l'obiettivo di porting nella sola modalità user.

Mentre, lo sviluppo di ski è continuato incessante (l'ultima versione 0.9.81-15 risale a maggio 2004) per permettere, a tutti coloro che non abbiano ancora una macchina con architettura IA-64, di continuare a contribuire allo sviluppo dei kernel per questa architettura.

Quindi, per poter accedere alla modalità kernel si richiede di creare un cross-compilatore capace di compilare un kernel per architettura ia64 avviabile tramite il simulatore. Operazione al quanto angusta e poco supportata. Negli anni passati il kernel mode era stato possibile grazie all'utilizzo del cross-compilatore gcc messo a disposizione da NUE. Ma dato che il suo sviluppo si è fermato alla GNU libc v2.2, questo è risultato inadatto a cross-compilare un kernel sugli attuali sistemi host dotati di distribuzioni Linux con versioni di glibc superiori alla v2.3.

Perciò, essendo comunque i nostri scopi lontani da quelli sviluppare kernel per questa architettura ci siano "accontentati" di utilizzare l'IA64SDK nella sola modalità user soddisfacente i nostri scopi. E' stato utilizzato ski v0.9431-2, nue v1.1 ed il file system v1.2 di nue.

In entrambe le modalità di esecuzione, ski rende disponibile un'interfaccia grafica *user-friendly* che mostra il codice disassemblato del programma, il contenuto dei registri e quello della memoria dell'architettura simulata. Il simulatore è molto simile ad un comune debugger: si può avviare il programma dall'inizio sino alla fine in modo continuativo o attraverso step successivi che eseguano istruzione per istruzione (facendo uso di breakpoint).

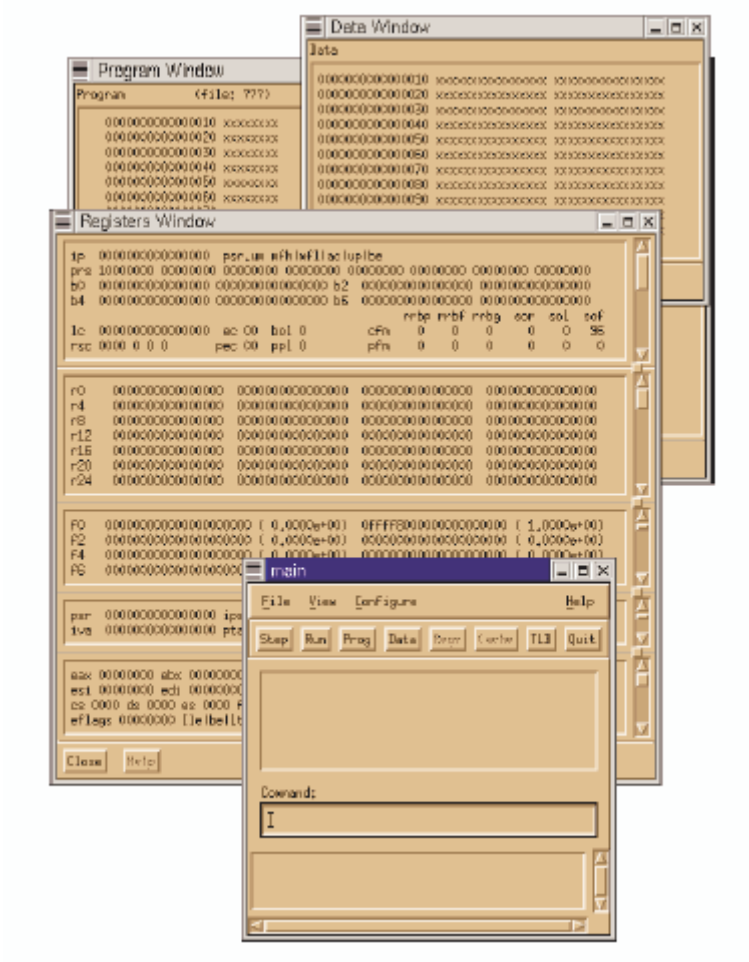

Figura 16: Interfaccia grafica di ski [17]

Naturalmente trattandosi di un simulatore, esso non può essere usato per determinare le prestazioni nel mondo reale dei programmi simulati, infatti a livello user non si possono emulare sistemi multiprocessore, cioè non è supportato il multi-threading, e non si possono eseguire processi concorrenti precludendo anche la possibilità di determinare possibili violazione di dipendenze nei programmi. Ma le statistiche di esecuzione possono comunque essere utilizzate opportunamente per avere dei riferimenti sulle prestazioni della nuova architettura [16].

5.3 Test effettuati

A questo punto si è scelto di analizzare le prestazioni delle due architetture limitatamente ad un caso di decodifica Mpeg2 su di un file video di esempio della grandezza di 900KB, risoluzione di 768x576, *framerate* 25 e durata 1 secondo.

Questa scelta, sicuramente insufficiente per trarre delle conclusioni troppo significative dal punto di vista prestazionale, trova il suo fondamento nel fatto che si tratta di un programma che utilizza in buona parte le risorse di calcolo della CPU e quindi e possibile attribuire ogni differenza prestazionale approssimativamente alle sole differenze architetturali delle CPU.

Per fare ciò, con il compilatore GCC v2.96 (della Cygnus, attuale divisione della RedHat) fornito con NUE v1.1 è stato possibile ricompilare il decoder mpeg v1.2 (http://www.mpeg.org) ed avviarlo con il predetto filmato video di prova, ottenendo dal simulatore i seguenti risultati:

numero delle istruzioni	2.310.134.138
cicli di clock	727.303.163
CPI (cicli di clock / n. istruzioni)	0,31

Per ottenere lo stesso tipo di informazioni (limitate, ma altrettanto preziose) messeci a disposizione da ski si è dovuto utilizzare un programma capace di funzionare sempre sulla medesima piattaforma Linux per mantenere una certa compatibilità a livello di binari (sempre nel formato *ELF* a 32 o a 64 bit e quindi conformi allo stesso standard *ABI*).

Allora si è proceduto a compilare il programma di decodifica video con il GCC v3.3.1 (della SuSE) ed in seguito sono stati campionati il numero di istruzioni e di cicli di clock con il programma Intel VTune Performance Analyzer 2.0 per Linux, ottenendo i seguenti risultati:

numero delle istruzioni	1.758.000.000
cicli di clock	1.371.971.750
CPI (cicli di clock / n. istruzioni)	0,78

Un termine di paragone importante risulta il numero di cicli di clock per istruzione (CPI), che sono diminuiti del 60% rispetto alle tradizionali architetture IA-32. Infatti essendo

Tempo CPU = cicli di clock / frequenza di clock

si deduce che il Tempo CPU è direttamente proporzionale ai cicli di clock; perciò, teoricamente, a parità di frequenza di clock tra le due architetture, una diminuzione dei cicli di clock del 47% determina una diminuzione proporzionale del Tempo CPU. In realtà a causa dell'eterogeneità dei programmi nell'uso delle risorse della CPU, la diminuzione del Tempo CPU sarà probabilmente meno che proporzionale.

Si sono appositamente trascurati i risultati in termini di tempo poiché irrilevanti ai fini di un confronto prestazionale tra un programma simulato ed uno no; basti pensare che con la medesima configurazione hardware il programma simulato è stato eseguito in 21,1 minuti mentre nel sistema host il corrispondente binario IA-32 ha impiegato meno di 3 secondi!

Un'altra differenza sostanziale da prendere in considerazione è la grandezza dei due binari appena compilati. Il binario IA-32 è grande 65KB, mentre quello IA-64 ben 156KB, cioè il 60% più grande. Questa grossa differenza nasce dal fatto che le istruzioni dell'IA-32 sono a lunghezza variabile e ciò permette di ottenere un certo vantaggio in fase di codifica (anche se come abbiamo già visto ciò va a penalizzare la fase di decodifica nel core), mentre nell'IA-64 la lunghezza delle istruzioni è fissa; inoltre nel binario IA-64 sono presenti molte più informazioni (salti, predicati,…) riguardanti l'esecuzione stessa del programma. Dal punto di vista dei dispositivi di memorizzazione di massa e non, ciò non sembra rappresentare un grosso problema dato che sul mercato mondiale si sta assistendo ad un rapido aumento della loro dimensione.

Inoltre, per rimarcare l'importanza del ruolo svolto dai compilatori, nell'IA-32 ed a maggior ragione nell'IA-64, si è proceduto ad utilizzare il cross-compilatore ORC v2.1 (Open Research Compiler), basato sul compilatore SGI OpenPro64, che come tanti altri compilatori attualmente sviluppati (vedi anche Intel C++ Compiler) hanno adottato la tecnica OpenMP che permette di parallelizzare il codice sorgente scritto in linguaggio C per aumentare il numero di thread dei programmi in modo da utilizzare pienamente le risorse messe a disposizione dai processori. Nel caso specifico, il compilatore scelto dichiara di essere stato realizzato con lo scopo di ottimizzare lo scheduling e i bundle di istruzioni per il processore Itanium2. Perciò si è proceduto a cross-compilare il predetto programma di decodifica video, ed eseguendolo sempre con lo stesso video di prova si è ottenuto che:

numero delle istruzioni	2.722.329.264
cicli di clock	753.497.825
CPI (cicli di clock / n. istruzioni)	0,28

Come si può osservare si è ottenuto ancora una diminuzione del CPI del 10% rispetto al compilatore GCC v2.96, mentre i cicli di clock sono aumentati del 4%.

Ciò ha permesso di eseguire la simulazione in 15 minuti contro i 21 precedenti, cioè una diminuzione del Tempo CPU del 29%; tale dato però non va inteso in senso assoluto, infatti è necessario tener conto dei limiti del simulatore utilizzato, che non simula il parallelismo a livello istruzioni e non supporta neanche il multi-threading.

Quindi questo risultato dovrebbe assumere scarso valore poiché rilevato in un ambiente simulato, ma secondo quanto dichiarato dagli stessi sviluppatori del compilatore, il guadagno medio ottenibile in termini di Tempo CPU si dovrebbe proprio aggirare attorno al 30%; ciò va a verificare la validità dei nostri test prestazionali e l'enorme potenzialità racchiusa nello sviluppo software dei compilatori degli anni a venire.

Per meglio comprendere come sia stato possibile ottenere un tale incremento di prestazioni è necessario analizzare la composizione delle istruzioni assembler. Nell'analisi del programma di decodifica è sufficiente limitarsi ad analizzare le due funzioni scritte in linguaggio C che si occupano della trasformata discreta a coseni (DCT) inversa bidimensionale, che costituisce il cuore dell'applicazione e che quindi ne determina la prestazione. Infatti si tratta di due funzioni che svolgono delle operazioni sugli interi contenuti nelle righe e nelle colonne di una matrice di ordine 8. Dunque per quanto

riguarda il codice assembler relativo a queste funzioni nel caso del compilatore gcc v2.96 si ha che:

numero di istruzioni totali	471
numero di nop totali	118 (25%)

numero di nop [M]	39 (33%)
numero di nop [I]	76 (64%)
numero di nop [F]	3 (3%)
numero di nop [B]	0 (0%)

mentre nel caso di ORC v2.1 si ha che:

numero di istruzioni totali	423
numero di nop totali	37 (9%)

numero di nop [M]	12 (32%)
numero di nop [I]	21 (57%)
numero di nop [F]	1 (3%)
numero di nop [B]	3 (8%)

La riduzione delle null operation, dal 25% al 9% del numero totali di istruzioni delle funzioni della DCT in maniera equivalente su ogni tipo di unità di esecuzione, ha

sicuramente inciso sulle prestazioni del binario compilato con ORC v2.1 che è riuscito ad assemblare meglio il codice sorgente in modo da utilizzare a pieno tutte le unità di esecuzione parallele messe a disposizione dall'architettura IA-64. E' infatti questa la strada seguita da questa architettura, che cerca di sfruttare tutti i livelli di parallelizzazione ormai lasciati inutilizzati dagli attuali processori IA-32 senza Hyper-Threading.

Per avvalorare il lavoro di ottimizzazione compiuto dal compilatore ORC v2.1, si è proceduto ad analizzare una piccola parte della trasformata a coseni utilizzata dal programma di decodifica video. Il listato in linguaggio C è il seguente:

```
000   #define W1 2841 /* 2048*sqrt(2)*cos(1*pi/16) */
001   #define W2 2676 /* 2048*sqrt(2)*cos(2*pi/16) */
002   #define W3 2408 /* 2048*sqrt(2)*cos(3*pi/16) */
003   #define W5 1609 /* 2048*sqrt(2)*cos(5*pi/16) */
004   #define W6 1108 /* 2048*sqrt(2)*cos(6*pi/16) */
005   #define W7 565  /* 2048*sqrt(2)*cos(7*pi/16) */
006
007   int main()
008   {
009     int x0, x1, x2, x3, x4, x5, x6, x7, x8;
010
011     /* first stage */
012     x8 = W7*(x4+x5);
013     x4 = x8 + (W1-W7)*x4;
014     x5 = x8 - (W1+W7)*x5;
015     x8 = W3*(x6+x7);
016     x6 = x8 - (W3-W5)*x6;
017     x7 = x8 - (W3+W5)*x7;
018   }
```

Questo è stato disassemblato dal compilatore gcc v2.96 nel seguente listato:

```
012   x8 = x7 * ( x4 + x5 ) ;
//conserva in r2 il valore dello stack pointer che punta a x8
main                    mov              r2=r12                      MII
//alloca 32 byte necessari alle nove variabili ad interi iniziali
```

```
                        adds            r12=-32,r12;;
//operazione inutile
                        adds            r15=-32,r2;;
//operazione inutile
main+0010               mov             r14=r15                     MII
//copia in r16 la posizione di x8 nello stack
                        mov             r16=r2
//operazione inutile
                        adds            r17=-32,r2;;
//operazione inutile
main+0020               mov             r14=r17                     MII
//indica la posizione di x4 nello stack
                        adds            r15=-16,r2
//operazione inutile
                        adds            r17=-32,r2;;
//operazione inutile
main+0030               mov             r14=r17;;                   MMI
//indica la posizione di x5 nello stack
                        adds            r14=-12,r2
                        nop.i           0x0
//recupera il valore della variabile x4 dalla memoria
main+0040               ld4             r15=[r15];;                 MMI
//recupera il valore della variabile x5 dalla memoria
                        ld4             r14=[r14]
                        nop.i           0x0;;
//addiziona il valore di x5 a quello di x4
main+0050               add             r15=r15,r14                 MII
//recupera il valore della costante W7
                        mov             r14=0x235;;
                        nop.i           0x0
//setta un registro floating point con il valore ( x4 + x5 )
main+0060               setf.sig        f6=r15                      MMF
//setta un registro floating point con il valore di W7
                        setf.sig        f7=r14
                        nop.f           0x0;;
main+0070               nop.m           0x0                         MFI
//esegue la moltiplicazione fra gli interi W7 e ( x4 + x5 )
                        xmpy.l          f6=f6,f7
                        nop.i           0x0;;
013   x4 = x8 + ( W1 - W7 ) * x4;
//recupera il risultato della moltiplicazione W7 * ( x4 + x5 )
main+0080               getf.sig        r14=f6;;                    MMI
//memorizza il predetto risultato in memoria
                        st4             [r16]=r14
//operazione inutile
                        adds            r15=-32,r2;;
…
014   x5 = x8 - ( W1 + W7 ) * x5 ;
…
015   x8 = W3 * ( x6 + x7 ) ;
…
016   x6 = x8 - ( W3 - W5 ) * x6 ;
…
017   x7 = x8 - ( W3 + W5 ) * x7 ;
…
//ripristina il precedente valore dello stack pointer
```

```
                        mov             r12=r2
main+0310               nop.m           0x0                         MIB
                        nop.i           0x0
//salta alla posizione dello stack pointer della funzione chiamante
//in questo caso concluderà semplicemente l'esecuzione del programma
                        br.ret.sptk.many b0;;
```

Si è evitato di riportare il codice assembler delle altre istruzioni C perché i punti salienti di queste si ripetevano ciclicamente.

Analizzando solo la prima istruzione si possono notare ampi sprechi compiuti in fase di compilazione: per ogni istruzione C ci sono sempre sei operazione completamente inutili ai fini pratici del programma che il compilatore produce in automatico senza tener conto che in questa specifica situazione queste non hanno affatto significato; inoltre stupisce come il compilatore abbia scelto di utilizzare i registri floating point per calcolare delle semplici moltiplicazioni fra interi; si noti inoltre la presenza di almeno sei nop per ogni istruzione C. Dunque da una prima analisi si evince subito come le possibilità di ottimizzazione siano plurime. A questo punto si può passare ad analizzare il codice assembler generato dal compilatore ottimizzato ORC v2.1:

```
//alloca una nuova finestra nello stack rotante dei registri generali
main                    alloc           r24=ar.pfs,0,2,0,0      MII
//lo stack pointer punta la variabile x8
                        adds            r12=-32,r12
                        nop.i           0x0;;
//indica all'istruzione 012 dove andare a memorizzare il valore di x8
main+0010               mov             r19=r12                 MMI
                        mov             r18=r12
                        adds            r22=40,r12
main+0020               adds            r17=4,r12               MMI
                        mov             r16=r12
                        adds            r21=4,r12;;


main+0030               ld4             r21=[r21]               MMI
//indica la posizione di x5 nello stack
```

```
                        adds             r14=12,r12
                        mov              r10=r12
main+0040               st8              [r22]=r33                  MMI
//indica la posizione di x4 nello stack
                        adds             r22=8,r12
                        adds             r15=8,r12;;
//recupera il valore della variabile x4 dalla memoria
main+0050               ld4              r22=[r22]                  MMI
//recupera il valore della variabile x5 dalla memoria
                        ld4              r14=[r14]
                        mov              r9=r12
main+0060               mov              r3=r12                     MMI
                        adds             r2=16,r12
                        adds             r8=12,r12;;
main+0070               nop.m            0x0                        MII
//esegue la somma fra interi ( x4 + x5 )
                        add              r21=r21,r22
                        adds             r23=32,r12;;
main+0080               st4              [r23]=r32                  MII
//inzia operazione di moltiplicazione per W7
                        shladd           r20=r21,4,r21;;
                        shladd           r20=r20,1,r21
main+0090               nop.m            0x0                        MII
                        shladd           r21=r21,2,r21;;
                        shladd           r20=r20,4,r21;;
//finisce operazione di moltiplicazione per W7
main+00a0               nop.m            0x0                        MII
//estende il segno del risultato ottenuto a 64 bit
                        sxt4             r20=r20
                        nop.i            0x0;;
//memorizza il risultato di x8 ottenuto con l'operazione 012
main+00b0               st4              [r19]=r20                  MII
                        adds             r20=4,r12;;
                        nop.i            0x0
//recupera il valore di x8 dalla memoria
main+00c0               ld4              r20=[r20]                  MMI
…
//salta alla posizione dello stack pointer della funzione chiamante
//in questo caso concluderà semplicemente l'esecuzione del programma
                        br.ret.sptk.many  b0;;
```

Innanzitutto, in questo secondo caso è possibile vedere come il compilatore non abbia più rispettato l'ordine originale del sorgente C, subito vengono inseriti in registri specifici le posizioni nello stack delle varie variabili; sicuramente si è avuta una diminuzione del numero delle nop; ma la differenza più significativa è l'assenza di operazioni fatte sui registri floating-point, perché come si è già evidenziato si tratta di interi e quindi risulterebbe uno spreco inutile. La cosa più interessante è che tali moltiplicazioni vengono

compiute con delle operazioni di shladd (shift left and add): la costante viene scomposta in potenze di 2 e si procede ad effettuare uno shift a sinistra per ogni moltiplicazione per due dell'operando; nel caso dell'istruzione alla riga 012 si dovrebbe avere:

```
shladd          r20=r21,4,r21;;
shladd          r20=r20,1,r21;;
shladd          r20=r20,2,r21;;
shladd          r20=r20,2,r21;;
```

cioè essendo r21 uguale alla somma dei valori delle variabili (x4+x5), per moltiplicarlo per la costante W7=565 si dovrà

1) moltiplicare il valore di r21 per 2^4=16 (shift di 4 bit) e sommarlo a r21;
2) moltiplicare il risultato precedente per 2 (shift di 1 bit) e sommarlo a r21:
3) moltiplicare il risultato precedente per 2^2=4 (shift di 2 bit) e sommarlo a r21;
4) moltiplicare il risultato precedente per 2^2=4 (shift di 2 bit) e sommarlo a r21.

Ma in realtà il compilatore non ha compiuto questa sequenza di operazioni perché tutte dipendenti fra loro e quindi non parallelizzabili, bensì utilizzando sempre solo due registri ha corretto le operazioni 3 e 4 nelle seguenti:

3) moltiplicare il valore di r21 per 2^2=4 (shift di 2 bit) e sommarlo a r21;
4) moltiplicare il valore ottenuto al punto 2 per 2^4=16 (shift di 4 bit) e sommarlo a r21.

In questo modo è stato possibile ottenere sempre lo stesso risultato ma più rapidamente perché è stato possibile parallelizzare le operazioni 2 e 3 poiché indipendenti fra loro, riuscendo ad eseguire l'operazione di moltiplicazione fra gli interi in un numero minore si cicli di clock.

Molte altre ottimizzazione potranno essere eseguite ancora, vedasi ad esempio la possibilità di eliminare due istruzioni ravvicinate di store e load della stessa variabile nel caso in cui questa compaia all'interno di due istruzioni C consecutive, eliminando i problemi di latenza dalla memoria principale.

5.4 Conclusioni

Riportiamo nella tabella che segue i risultati dell'analisi prestazionale del decoder Mpeg compilato per le due differenti architetture ed anche nella versione ottimizzata per l'IA-64. Leggendo i dati in tabella a partire dalla colonna dell'IA-32, si può procedere ad un primo confronto con l'IA-64: dai valori assoluti e dai dati percentuali (riferiti alla seconda colonna) riportati è possibile rilevare un netto aumento del numero delle istruzioni di macchina del programma dovuto al differente ISA utilizzato ed un ottimo miglioramento delle prestazioni globali della macchina, e sfruttando la solita formula:

Tempo CPU = cicli di clock / frequenza di clock

si può ipotizzare una diminuzione del Tempo CPU circa del 47%.

Per avvalorare la bontà delle prestazioni dell'architettura IA-64 è possibile confrontare la terza e quarta colonna (i dati percentuali si riferiscono alla sola terza colonna) per verificare i vantaggi prestazionali ottenibili andando ad utilizzare compilatori sempre più ottimizzati per questa architettura. Si nota infatti che le prestazioni vanno via via a migliorare, si può notare un ulteriore diminuzione dei clock per instruction del 10%, il chè fa ben sperare per il futuro. Ciò è giustificato dall'adozione della tecnica OpenMP che è riuscita ad ottenere una diminuzione del numero di nop presenti nel codice disassemblato (dal 25% al 9%) e una migliore utilizzazione delle unità di esecuzione disponibili, a tutto vantaggio dello sfruttamento del parallelismo della macchina.

	IA-32	IA-64 tradizionale	IA-64 ottimizzato
N. di istruzioni	1.758.000.000	2.310.134.138 (+31%)	2.722.329.264 (+17%)
Cicli di clock	1.371.971.750	727.303.163 (-47%)	753.497.825 (+4%)
CPI	0,78	0,31 (-60%)	0,28 (-10%)
N. di nop di DCT	-	118 (25% di 471)	37 (9% di 423)

Dunque dai test effettuati col decoder Mpeg, risulta evidente come in un'applicazione che sfrutti maggiormente la capacità di calcolo della CPU l'architettura IA-64 prevalga nettamente sull'IA-32. Naturalmente il termine "nettamente" utilizzato in questo contesto non va inteso in senso assoluto, poiché se è vero che nei nostri test sono stati rilevati grossi gap prestazionali, ciò potrebbe perdere di validità in altri ambiti, quali applicazioni con diverso utilizzo delle risorse della CPU o effettivo vantaggio derivante dal rapporto prestazioni/costi.

Come si è avuto modo di vedere, in questi test non sono state comparate soltanto le performance delle due architetture, ma ci si è spinti oltre andando a valorizzare uno dei concetti cardine dell'IA-64 (di ispirazione RISC): l'importanza di avere un compilatore molto efficiente. Perciò si è fatto uso di uno dei tanti compilatori ottimizzati per IA-64 (ORC v2.1) che ha dimostrato la maturità di tale tesi.

Dunque, grazie all'enorme flessibilità dimostrata negli ultimi anni dal mondo del software, il vantaggio offerto dal fatto che la complessità si sia spostata dall'hardware verso il software, dà la possibilità di avere in futuro aumenti di prestazioni senza dover cambiare nuovamente l'architettura del sistema.

6 Conclusioni

La scelta di mantenere attive le produzioni delle famiglie di processori basate su entrambe le architetture IA-32 e IA-64 consentirà all'Intel di procedere al loro sviluppo in parallelo, progredendo di pari passo senza che nessuna delle due architetture rimanga indietro nella continua "corsa tecnologica". Infatti per entrambe le architetture è già stato previsto l'ulteriore aumento del livello di parallelizzazione attraverso il **Thread Level Parallelism (TLP)**, che consiste nell'esecuzione di istruzioni in parallelo da parte di multiprocessore che possono essere fisici (cioè con la presenza effettiva di più CPU sul medesimo chip come nel POWER4 di IBM) o virtuali (cioè ottenuto da un singolo chip che emula un sistema multiprocessore simmetrico come l'EV8® della Compaq®) di differenti processi software (o thread). Questo approccio trova le sue ragioni e naturali applicazioni nell'ambito del mercato server dove effettivamente molte applicazioni possono essere scomposte in più threads ed assegnate ai diversi processori che costituiscono il sistema; ed è proprio in base a questo principio che trova ragione di esistere la tecnica OpenMP di compilazione che sta prendendo piede nei compilatori di entrambe le architetture.

E con la recente diffusione di sistemi operativi come Windows XP, Windows 2003 e Linux, che supportano i sistemi multiprocessore, e di linguaggi di programmazione come il Java, che predispone a sviluppare applicazioni multithread, l'implementazione del TLP anche sui sistemi desktop porterà presto a dei notevoli vantaggi.

L'approccio TLP, in sostanza, pone la sua attenzione sul design del sistema, più che sulla definizione di un nuovo ISA e in particolare, dovendo prevedere la condivisione di risorse tra i diversi processori, viene posta grande attenzione nella gestione della banda di

memoria per non creare interferenze di cache tra i diversi threads; e difatti l'IBM con il G5 ha speso molte risorse in questa direzione. I sistemi TLP inoltre potrebbero essere più liberi nella loro evoluzione rispetto alle macchine EPIC e limitati più che altro dalla tecnologia produttiva.

Recentemente l'Intel ha introdotto nella famiglia dei Pentium IV la tecnologia **Hyper-Threading** e si prepara a presentare soluzioni *dual-core* del Pentium IV e dell'Itanium 2, che promettono di raddoppiare le prestazioni ad un costo ragionevole. Soluzione sicuramente più semplice ed economica che provare a raddoppiare il numero di unità di esecuzione e di registri all'interno dello stesso chip.

Sicuramente, come anticipato già da diversi produttori come AMD e IBM, e come già osservato nei nostri test prestazionali, la mole di dati da processare in continuo aumento rende quasi obbligatoria la transizione da 32 a 64 bit nei microprocessori per PC, e quindi da questo punto di vista, l'IA-64 sicuramente non incontrerà troppi ostacoli, però il problema si pone per quanto riguarda il completo rifacimento dell'architettura di base.

Indubbiamente la nuova architettura IA-64 costituisce un passo epocale di Intel verso nuovi paradigmi di progettazione dei processori. Tuttavia abbandonare la ormai vetusta architettura x86 rappresenta sì una necessità (almeno in prospettiva futura) ma anche un grosso rischio. Abbiamo infatti più volte assistito alla dipartita di tecnologie meravigliose a causa delle severe leggi di mercato.

Un altro possibile problema potrebbe essere costituito dall'esecuzione in ordine, il raggruppamento statico delle istruzioni e l'esecuzione speculativa che non possono comunque garantire la totale assenza di cache miss. Inoltre un importante linguaggio come il Java di tipo semicompilato potrebbe incontrare alcune difficoltà dato che in questo caso

la compilazione assume un ruolo defilato rispetto a tutti gli altri tipi di linguaggi compilati considerati di solito.

Comunque un'architettura come l'IA-64 porta grandi innovazioni ed una elevata potenza specifica (cioè una elevata capacità di calcolo ed efficienza operativa a parità di clock) ma al contempo richiede una completa riscrittura dei software e dei sistemi operativi che rappresenta un grosso limite iniziale al diffondersi dell'IA-64 e quindi al suo sviluppo futuro. Basti pensare alle difficoltà di rilascio di Microsoft Windows XP-64 e al rilascio del software "salvagente" IA-32 Execution Layer (EL).

Sarà anche interessante vedere in che modo il mercato risponderà alle diverse offerte attualmente disponibili: AMD Athlon 64 (K8) e IBM Power5 (G5), che nascono da approcci completamente diversi.

La strada per l'IA-64 appare quindi ancora tutta in salita ma l'Intel ha già dimostrato di possedere le capacità per imporsi sul mercato, come quando venti anni fa riuscì ad affermare l'architettura x86 nel mercato dei PC.

Appendice – Come effettuare i test

I. Introduzione

Per poter effettuare tutti i test prestazionali riportati nel capitolo 1 è stata approntata una macchina dotata di processore Intel PIII 500 MHz. Si è proceduto ad installare sul sistema una distribuzione Linux, quale la SuSE 9.0 Professional dotata del kernel Linux v2.4.21-231-default, gcc v3.3.1, ld v2.14.90.0.5, glibc v2.3.2; questa distro è stata preferita ad altre anche più recenti ed aggiornate per motivi di stabilità e compatibilità con alcuni software utilizzati, in particolar modo con Intel VTune (a tal proposito può essere utile consultare le note di rilascio del software in questione [18]).

II. Installazione applicazioni

La prima cosa da fare è effettuare il download di tutti file necessari ai nostri scopi:

1. http://www.hpl.hp.com/research/linux/ski/clickthrough.php?file=ski-0.9431-2.i386.rpm : il simulatore ski v0.9431-2;
2. http://www.hpl.hp.com/research/linux/ski/clickthrough.php?file=nue-1.1-1.i386.rpm : il comando e la documentazione di nue v1.1;
3. http://www.hpl.hp.com/research/linux/ski/clickthrough.php?file=nue-fs-1.2-1.i386.rpm : il filesystem di nue v1.2;

4. http://ovh.dl.sourceforge.net/sourceforge/ipf-orc/orc-2.1-bin.tar.gz : il compilatore open source ORC v2.1;
5. http://www.intel.com/software/products/vtune/vlin/eval.htm : VTune Performance Analyzer 2.0 per Linux;
6. ftp://ftp.mpegtv.com/pub/mpeg/mssg/mpeg2v12.zip : il decoder Mpeg2 v1.2.

Poi, con i diritti di amministratore, si procede ad installare tutto il predetto software.

Dal direttorio dove si scaricano i predetti file, bisogna installare il pacchetto rpm di ski con il comando:

- rpm –Uhv ski-0.9431-2.i386.rpm

ma dato che si tratta di un pacchetto rpm creato nell'aprile 2001 sulla base di una distro RedHat, su una recente distribuzione come quella di SuSE 9.0 e su tutte le altre che rispettano lo standard sull'uso dei servizi unix, è necessario anche spostare lo script del servizio ia64fmt nella posizione corretta andandolo a modificare opportunamente:

- cp /etc/init.d/init.d/ia64fmt /etc/init.d/
- vi /etc/init.d/ia64fmt

ed eliminare la dicitura "|| exit 1" dalla riga 23 e 28. Riavviato il sistema, bisogna verificare che il servizio sia stato avviato correttamente con il comando:

- /etc/init.d/ia64fmt status

In caso negativo è necessario creare un link software per avviare e fermare il servizio almeno nel runlevel 5 in questo modo:

- ln –s /etc/init.d/ia64fmt /etc/init.d/rc5.d/S60ia64fmt
- ln –s /etc/init.d/ia64fmt /etc/init.d/rc5.d/Z60ia64fmt

e una volta riavviata la macchina bisogna assicurarsi nuovamente che questa volta il servizio sia stato avviato con successo; in caso il servizio non venga ancora avviato è necessario controllare il corretto avvio del modulo binfmt_misc nel file di configurazione:

- less /etc/modprobe.conf

naturalmente, il file potrebbe variare di distribuzioni in distribuzione.

Ci si raccomanda di non aggiornare il simulatore alla sua ultima versione reperibile all'indirizzo http://www.hpl.hp.com/research/linux/ski/clickthrough.php?file=ski-0.9.81-l5-i686.tar.gz perché questo renderebbe inutilizzabile nue e così anche la modalità user; in questo caso l'unica possibilità di utilizzare ski sarebbe quella in kernel mode facendo uso anche del disk image messo a disposizione all'indirizzo http://www.hpl.hp.com/research/linux/ski/clickthrough.php?file=ski-linux-rootfs-2.0-1.noarch.rpm; ma con questa soluzione si presenta il problema di cross compilare il kernel con una della configurazioni specifiche per tale simulatore.

Una volta installato correttamente l'rpm di ski, si procede con il comando nue ed il suo file system:

- rpm –Uhv nue-1.1-1.i386.rpm
- rpm –Uhv nue-fs-1.2-1.i386.rpm

A questo punto ski e nue sono installati correttamente. Per poterlo utilizzare comodamente si può creare/modificare il file di profilo che viene avviato ad ogni login:

- vi /root/.profile

inserendo le seguenti righe:

- xhost +localhost
- export DISPLAY=localhost:0.0

questo si rende necessario a causa del fatto che nue è un ambiente chrooted e quindi un'applicazione X11 come xterm e lo stesso xski non riescono ad accedere alle usuali socket, e devono quindi poter dialogare tramite l'interfaccia di loopback utilizzando il protocollo TCP/IP. In alcune distribuzioni, come la SuSE 9.1, ciò potrebbe essere stato inibito di default per motivi di sicurezza avviando il server X con l'opzione –nolisten TCP, perciò sarà necessario andare a modificare il file di avvio di X eliminando tale opzione (utilizzando come server grafico kde 3.2 di SuSE 9.1, il file in questione è /etc/X11/xdm/Xservers).

A questo punto l'IA64SDK è pienamente funzionante; nue è avviabile col comando:

- nue (comando "exit" per uscire)

mentre sia all'interno di nue che all'esterno e possibile avviare ski in versione testuale:

- ski

o nella sua versione grafica

- xski

A questo punto si procede ad installare il nuovo compilatore ORC v2.1; dopo aver decompresso l'archivio, per comodità è possibile andare a modificare nuovamente il file .profile aggiungendo le seguenti righe:

- export TOOLROOT=/root/orc
- export COMP_TARGET_ROOT=$TOOLROOT
- export PATH=$PATH:$TOOLROOT/usr/ia64-orc-linux/bin

per consentire una corretta procedura di installazione ed un semplice richiamo del nuovo compilatore semplicemente con il comando:

- orcc

a questo punto, dal direttorio dove è stato decompresso l'archivio basterà digitare:

- ./install.sh -c

Per quanto riguarda l'installazione di Vtune, una volta decompresso l'archivio sarà sufficiente digitare:

- ./install

E seguire la procedura di installazione installando tutti i componenti. Fortunatamente SuSE 9.0 risulta fra le poche distribuzioni supportate, già con una distro come SuSE 9.1 (dotata del kernel v.2.6.5) l'installazione è fallita miseramente richiedendo di utilizzare il VTune(TM) Performance Analyzer Driver Kit per ricompilare i driver a seconda del kernel in uso dal sistema operativo utilizzato. Unica eccezione è rappresentata dalla solita posizione dei servizi vtune e dcom, che come fatto già prima per il servizio ia64fmt vanno riposizionati e avviati correttamente nel runlevel 5. Inoltre, è necessario porre particolare attenzione al fatto che il software in questione non è gratuito, al momento del download è possibile ottenere una licenza della durata di 5 giorni a solo scopo dimostrativo che andrà posta nel direttorio /opt/intel/licenses. Inoltre per comodità nel solito file .profile è possibile modificare la riga:

- export PATH=$PATH:$TOOLROOT/usr/ia64-orc-linux/bin

nella nuova riga :

- export PATH=$PATH:$TOOLROOT/usr/ia64-orc-linux/bin:/opt/intel/vtune/bin

Dopodichè si procede a ricompilare il decoder Mpeg2 con i tre compilatori disponibili. Decomprimendo l'archivio in tre sottodirettori diversi di nue, in un primo direttorio, fuori dall'ambiente nue, si compila con il gcc v3.3.1 utilizzando il makefile fornito:

- make

ottenendo un file binario LSB di tipo ELF a 32-bit, mentre per cross-compilare con l'ORC v2.1 nel secondo direttorio si procede come segue:

- CC=orcc
- make

creando un file binario LSB di tipo ELF a 64-bit. Infine per compilare con il gcc v2.96 fornito con nue è necessario prima entrare nel suo ambiente per poi andare ad utilizzare semplicemente il suo compilatore nel terzo ed ultimo direttorio dove è stato decompresso l'archivio del decoder Mpeg2:

- nue
- make

Per maggiori informazioni riguardo l'installazione di ogni singolo software utilizzato si consiglia di consultare la documentazione relativa.

III. Uso applicazioni

A questo punto tutto è pronto per eseguire i test desiderati. Con vtune è sufficiente posizionarsi fuori dall'ambiente di nue ed in particolare nel primo direttorio …/mpeg2v12/src/mpeg2dec/ dove è stato compilato il binario a 32-bit ed eseguire l'analisi sul nostro ipotetico file video di prova:

- vtl activity –c sampling –o "-ec en='Instructions Retired', en='Clockticks' " –app mpeg2decode, "-b prova.m2v" run
- vtl show
- vtl view a1::r1 –processes

dove a1 rappresenta l'unica attività di analisi disponibile, come risulta dall'output del comando vtl show.

Per quanto riguarda i binari compilati a 64-bit è possibile trovare lo stesso tipo di statistiche (in realtà l'esiguità delle statistiche trattate è dovuta proprio al fatto che ski fornisce "solo" quelle) utilizzando ski nell'ambiente di nue:

- nue
- xski –forceuser …/mpeg2v12/src/mpreg2dec/mpreg2decode –b prova.m2v

per entrambi i direttori dove si è compilato con gcc v2.96 e con ORC v2.1. L'opzione -forceuser serve a forzare il simulatore ad operare in user mode (baco risolto nell'ultima versione di ski v.0.981-15, che però non potremo utilizzare per i motivi evidenziati prima).

All'interno dell'interfaccia grafica sarà possibile utilizzare il comando run per avviare l'applicazione dall'inizio alla fine ed ottenere le tanto agognate statistiche. Inoltre sarà possibile analizzare il codice assembler generato per analizzarne le differenze generate tra i due compilatori, ma non con il binario a 32-bit poiché questo utilizza un codice assembler sostanzialmente differente a causa della diversa instruction set architecture utilizzata.

IV. Guida di riferimento di ski

Il modo più semplice per utilizzare il simulatore ski è sicuramente nella sua versione grafica, dotata di un'interfaccia *user-friendly*. Come già accennato, la versione grafica del simulatore ski può essere avviata digitando il comando "**xski"** al prompt dei comandi di una qualsiasi *shell* di Linux. Prima di avviare xski all'interno di NUE bisogna essere sicuri

di aver settato correttamente la variabile di sistema DISPLAY. A questo punto sarà possibile visualizzare sullo schermo le quattro finestre principali del simulatore.

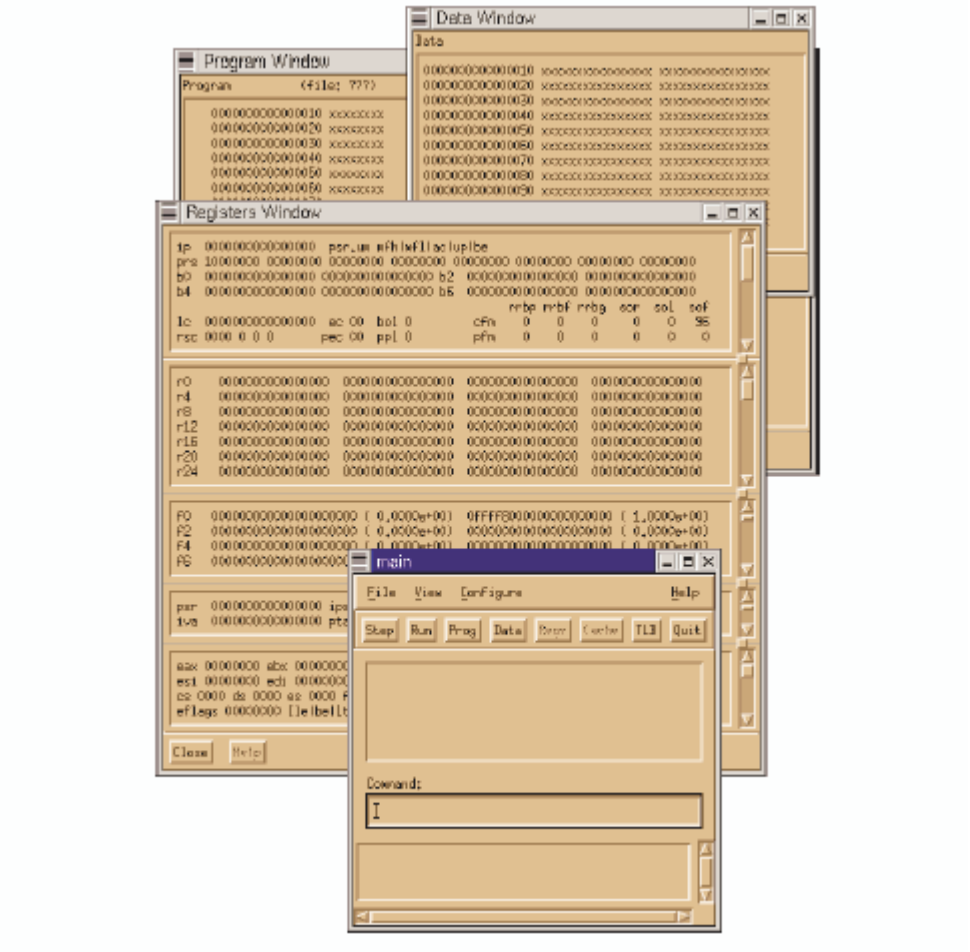

Figura 17: Le quattro finestre principali di xksi [17]

Usando la sezione **Command** della finestra **main** è possibile caricare il programma che si vuole simulare; basterà digitare il comando "**load"** seguito dal nome del programma, sempre che non lo si sia già passato tramite linea di comando nella shell.

A questo punto verrà visualizzato il codice assembler nella finestra **Program Window**, i valori delle variabile globali e statiche nella finestra **Data Window**, ed i valori dei registri della CPU simulata nella finestra **Register Window**. Per visualizzare tutti i dati contenuti

nella Register Window è necessario utilizzare l'apposito scrollbar, mentre a causa dell'enorme quantità di memoria indirizzabile dall'IA-64 non risulta conveniente utilizzare un semplice scrollbar, perciò per la Data Window sarà necessario utilizzare il comando "**dj**" ("**d**ata **j**ump") seguito dall'indirizzo di memoria che interessa visualizzare (per esempio è possibile visualizzare l'indirizzo di partenza dello stack di memoria contenuto nel registro r12). Questa finestra riporta sulla colonna di sinistra gli indirizzi di memoria in formato esadecimale, nelle due colonne centrali i valori esadecimali contenuti in memoria in corrispondenza dei rispettivi indirizzi di memoria, ed infine nell'ultima colonna a destra gli stessi valori in formato ASCII.

Figura 18: Data Window [17]

Facendo riferimento alla figura in alto, se l'indirizzo di partenza dello stack di memoria è 9fffffffffff780 allora i tutti primi 16 byte dello stack saranno inizializzati a zero. I successivi 8 byte contengono la variabile **argc** (che contiene il numero di parametri passati al nostro programma, dove il numero 1 indica che non è stato passato alcun parametro). Seguono le variabili **argv** (che contiene l'array di lunghezza variabile dei parametri passati al programma) ed **envp** (che contiene l'array delle variabili del sistema operativo). Infine è

possibile utilizzare i comandi "**df**" ("**d**ata **f**orwards") e "**db**" ("**d**ata **b**ackwards") per muoversi avanti ed indietro tra gli indirizzi di memoria contenuti nella Data Window.

Per quanto riguarda la Program Window, nella colonna di sinistra viene riportato il riferimento del codice disassemblato in riferimento alla funzione C rappresentata, nella colonna centrale è possibile visualizzare il codice disassemblato diviso per gruppi di tre istruzioni (bundle), mentre sulla colonna di destra vengono riportate le unità di esecuzione utilizzate per ogni bundle.

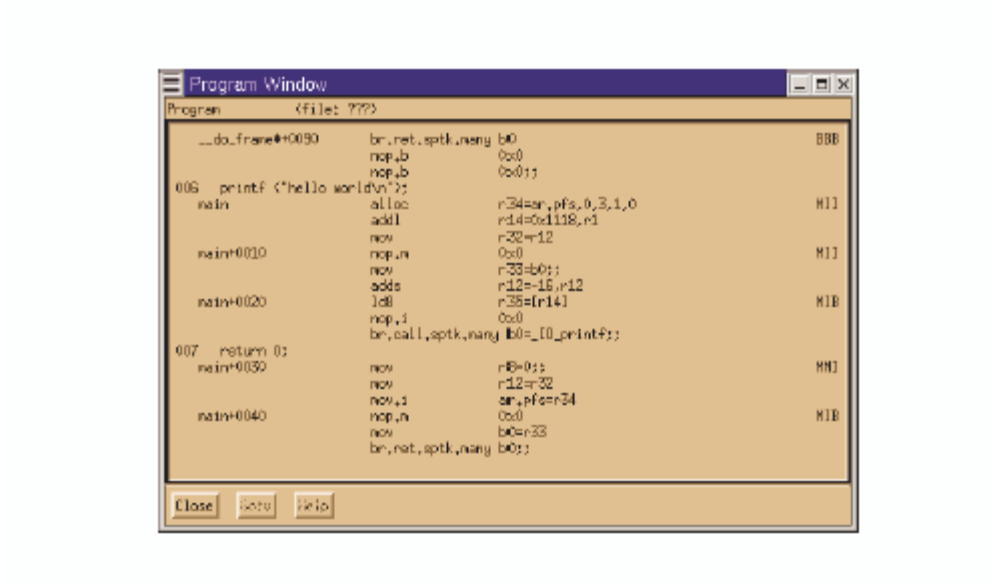

Figura 19: Program Window [17]

Una volta caricato il programma, questa finestra mostrerà il codice disassemblato partendo dalla linea **_start**; in ogni file in formato ELF, questa è la routine di partenza crt1.o, si tratta di un oggetto creato *run-time* che fornisce un'interfaccia tra l'ambiente del sistema operativo e l'ambiente ANSI C, allo scopo di poter avviare il programma semplicemente digitando il nome del binario dal prompt dei comandi, sempre che si posseggano i diritti necessari.

Alla pari della Data Window è possibile muoversi attraverso le righe di codice disassemblato utilizzando il comando "**pj**" ("**p**rogram **j**ump") seguito dal riferimento di codice a cui si vuole saltare; inoltre è possibile fare dello scrolling utilizzando i comandi "**pf**" ("**p**rogram **f**orward") e "**pb**" ("**p**rogram **b**ackwards").

Nel caso in cui il binario in formato ELF sia stato compilato in modalità debug (nella maggior parte dei compilatori in commercio si utilizza l'opzione "**–g**") sarà possibile visualizzare ogni linea di codice scritta in linguaggio C in corrispondenza delle corrispondenti linee di codice assembler; per passare ad una modalità di visualizzazione senza codice sorgente C e viceversa sarà possibile utilizzare rispettivamente i comandi "**pa**" ("**p**rogram **a**ssembly") e "**pm**" ("**p**rogram **m**ixed").

Come in ogni buon debugger, ski fornisce i breakpoint. Per settare un breakpoint si usa il comando "**bs**" ("**b**reakpoint **s**et") seguito dal punto del codice assembler in cui lo si vuole posizionare. Usando il comando "**bl**" ("**b**reakpoint **l**ist") si ottiene la lista dei breakpoint approntati. Per cancellare un singolo breakpoint si usa il comando "**bd**" ("**b**reakpoint **d**elete"), mentre per cancellarli tutti si usa "**bD**" ("**b**reakpoint **D**elete all").

L'ultima finestra che rimane da esaminare è la **Registers Window** che, dall'alto verso il basso, riporta gli user register, i general register e i floating point register.

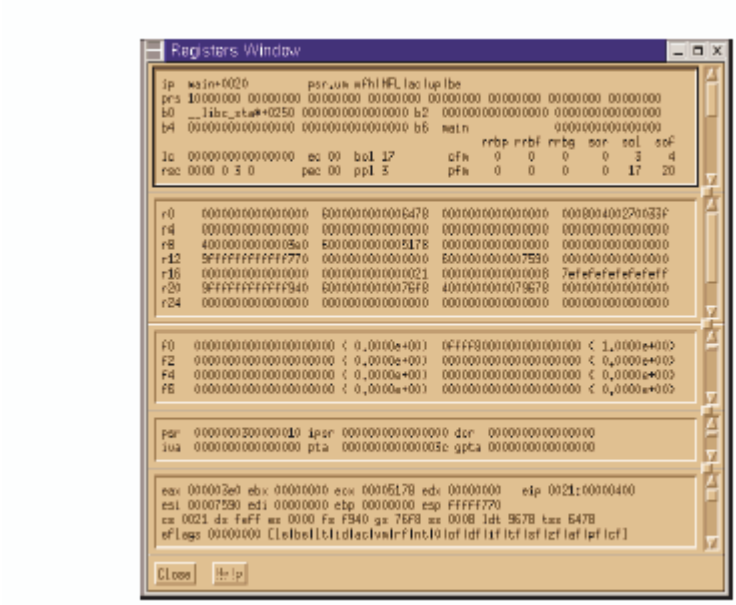

Figura 20: Register Window [17]

Una caratteristica molto utile in fase di debug è quella che permette di cambiare il contenuto di tutti i registri visibili, per esempio per cambiare il puntatore alla prossima istruzione (instruction counter) è possibile usare il comando "**= ip**" seguito da riferimento ad una determinata parte di codice assembler.

Infine per eseguire il programma caricato è possibile usare il comando "**run**" (disponibile anche il bottone sul pannello della finestra **Main**), nel caso si usino dei breakpoint è possibile usare il comando "**cont**" per continuare fino al breakpoint successivo o il comando "**step**" per eseguire un istruzione alla volta. Una volta eseguito il binario, sempre nella finestra **Main** verranno riportate tutte le statistiche inerenti l'esecuzione del programma. Per chiudere il simulatore è possibile utilizzare il comando "**exit**".

Per un tutorial completo e per maggiori informazioni riguardo l'uso del simulatore ski si consiglia di consultare il manuale [17].

Glossario

ABI è l'acronimo di Application Binary Interface; si tratta convenzioni che specificano i formati dei binari eseguibili per garantire la compatibilità a livello sorgente almeno fra le stesse architetture hardware.

Back-compatibility compatibilità all'indietro; termine utilizzato in genere per evidenziare la compatibilità dei software scritti per vecchi microprocessori come l'8086 con le nuove architetture che utilizzano l'ISA x86.

Cache miss mancanza del dato cercato all'interno della memoria cache, che provoca ulteriori latenze dovute alle basse velocità di accesso della memoria principale.

Compile-time al momento della compilazione; per esempio nello scheduling di un'istruzione.

Core o *die,* il "cuore" di un chip contenente i soli transistor e collegato al suo package sottostante tramite i bump che portano i segnali ai pin esterni.

CPU è l'acronimo di *central processing unit,* detto anche microprocessore, si tratta dell'unità di calcolo centrale di un computer.

CPI è l'acronimo di *clock per instruction*, e sta ad indicare il numero di cicli di clock necessari a processare un istruzione; questo è uno degli

	indici di riferimento nel confronto delle prestazioni dei microprocessori.
Critical path	ossia il cammino critico che indica il percorso più lungo che un segnale deve attraversare nel tempo di un ciclo di clock. Quindi il massimo ritardo di propagazione del segnale va a determinare la frequenza massima a cui una CPU può operare.
Dual-core	implementazione di CPU che prevede la presenza di due core distinti collegati allo stesso package.
ELF	formato dei file binari disponibile a 32 o 64 bit.
Fetch	operazione di caricamento nella pipeline di istruzioni.
Floating-point	in virgola mobile; termine utilizzato per indicare tutti i numeri macchina rappresentativi di numeri reali.
FIFO	è l'acronimo di *first-in first-out*, che indica una modalità di ordinamento di una coda secondo la quale il primo oggetto ad entrare è anche il primo ad uscire.
Framerate	tasso di immagini per secondo.
General purpose	per scopi generici; usato per indicare tutti quei processori costruiti per assolvere compiti generici, diversamente da particolari sistemi embedded.
Hit rate	è il tasso di successo nel reperire dati dalla cache, quindi è il risultato del rapporto tra riferimenti soddisfatti dalla cache e il totale dei riferimenti richiesti.

ISA	è l'acronimo di *instruction set architecture* e rappresenta l'insieme delle istruzioni assembler utilizzate da una particolare architettura di CPU.
Like-RISC	simile al RISC, si usa dire per quei processori che utilizzano soluzioni in accordo con la filosofia di progettazione di tipo RISC.
Like-CISC	simile al CISC, si usa dire per quei processori che utilizzano soluzioni in accordo con la filosofia di progettazione di tipo CISC.
Micro-operations	o *micro-ops* o *μ-ops*; termine utilizzato per indicare le operazioni atomiche in cui vengono decodificate le complesse istruzioni di processori come il Pentium IV.
Misprediction	predizione di salto errata con conseguente svuotamento della pipeline e calo delle prestazioni.
Off-chip	parte esterna al package del chip.
On-die	parte interna al die.
On-chip	parte del package del chip.
Prefetch	operazione attraverso la quale si caricano istruzioni prima di quanto previsto nell'ordine originale del programma.
Pseudo-LRU	è l'acronimo di *Pseudo-least recently used*; algoritmo secondo il quale viene preso in considerazione l'ultimo oggetto che è stato utilizzato.
Royalty	penale che una società si impegna a pagare per sfruttare un brevetto altrui.
Run-time	a tempo di esecuzione; per esempio la path da prendere in una diramazione nell'IA-64.

Scheduler	algoritmo di pianificazione; per esempio, usato per gestire le code di istruzioni verso le rispettive unità di esecuzione.
Shell	interprete dei comandi del sistema operativo.
Time-to-market	è il tempo necessario all'azienda per trasformare una nuova opportunità di mercato in prodotto.
User-friendly	si dice di un programma molto semplice da usare per l'utente finale; caso limite è quello di Windows dove tutti i programmi sono dotati di un interfaccia più o meno amichevole.
Write-Back	fase finale della pipeline durante la quale si memorizzano nella cache i dati risultanti dal processo di elaborazione.

Bibliografia

[1] Lorenzo Marchetti. “Le tecnologie dei processori: CISC vs RISC”
http://www.lithium.it/articolo.asp?code=20

[2] Paolo Lombardi. “Le tecnologie dei processori: IA-64, i 64 bit di Intel”
http://www.lithium.it/articolo.asp?code=21

[3] Gianni Conte. “Le sfide tecnologiche delle moderne CPU”
http://www.lithium.it/articolo.asp?code=29

[4] Intel Corporation [2004]. “The IA-32 Intel Architecture Software Developer’s Manual, Volume 1: Basic Architecture”, Order Number 25366515
http://developer.intel.com/design/Pentium4/documentation.htm

[5] Intel Corporation [2004]. “The IA-32 Intel Architecture Software Developer’s Manual, Volume 2A: Instruction Set Reference, A-M”, Order Number 25366615
http://developer.intel.com/design/Pentium4/documentation.htm

[6] Intel Corporation [2004]. “The IA-32 Intel Architecture Software Developer’s Manual, Volume 2B: Instruction Set Reference, N-Z”, Order Number 25366715
http://developer.intel.com/design/Pentium4/documentation.htm

[7] Intel Corporation [2004]. “The IA-32 Intel Architecture Software Developer’s Manual, Volume 3: System Programming Guide”, Order Number 25366815
http://developer.intel.com/design/Pentium4/documentation.htm

[8] Intel Technology Journal Q1 [2001]. “The Microarchitecture of the Pentium 4 Processor”
http://www.intel.com/technology/itj/

[9] Intel Corporation [2002]. “Intel Itanium Architecture Software Developer’s Manual, Volume 1: Application Architecture”, Order Number 245317
http://developer.intel.com/design/itanium/itanium/index.htm

[10] Intel Corporation [2002]. “Intel Itanium Architecture Software Developer’s Manual, Volume 2: System Architecture”, Order Number 245318
http://developer.intel.com/design/itanium/itanium/index.htm

[11] Intel Corporation [2002]. “Intel Itanium Architecture Software Developer’s Manual, Volume 3: Instruction Set Reference”, Order Number 245319
http://developer.intel.com/design/itanium/itanium/index.htm

[12] Intel Corporation [2002]. “Intel Itanium 2 Processor Hardware Developer’s Manual”, Order Number 25110901
http://developer.intel.com/design/itanium2/index.htm

[13] Intel Corporation [2004]. “Intel Itanium 2 Processor Reference Manual, For Software Development and Optimization”, Order Number 25111003
http://developer.intel.com/design/itanium2/index.htm

[14] Intel Corporation [2004]. “Intel Itanium 2 Processor, Specification Update”, Order Number 25114123
http://developer.intel.com/design/itanium2/index.htm

[15] William Stallings. “Computer Organization and Architecture, Designing for Performance”, Prentice Hall 2003.

[16] The Linux/ia64 Project: Kernel Design and Status Update

http://www.hpl.hp.com/techreports/2000/HPL-2000-85.html

[17] Hewlett-Packard Co. Ski IA-64 Simulator Reference Manual
http://www.hpl.hp.com/research/linux/ski/ski-manual-v1.0.pdf

[18] VTune Performance Environment User's Guide
/opt/intel/vtune/doc/users_guide/index.htm

[19] Readme For Open Research Compiler Release 2.1
http://ipf-orc.sourceforge.net/readme-release-2.1.htm

[20] Klauser A., Austin T., Grunwald D., Calder, B. . "Dynamic hammock predication for non-predicated instruction set architectures". Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on 12-18 Oct. 1998

[21] Stefan Rusu, Gadi Singer. "The First IA-64 Microprocessor". IEEE Journal Of Solid-State Circuits, Vol. 35, No. 11, Novembre 2000

Si ricorda, inoltre, che i contenuti presenti in questa tesi sono disponibili presso il sito web del corso di Calcolatori Elettronici della facoltà di Ingegneria di Modena:
http://imagelab.ing.unimo.it/Calcolatori/