Corso di Architettura degli Elaboratori

Parallelismo a livello di istruzioni e di thread in architetture scalari e superscalari

Marco Vanneschi © - Dipartimento di Informatica, Università di Pisa

Queste note integrano il Cap. XI del libro di testo per fornire una trattazione più completa delle architetture avanzate con parallelismo a livello di istruzioni (ILP) e della loro relazione con il parallelismo a livello di thread.

L'approccio didattico sarà quello di partire dalla trattazione della CPU pipeline di base e di estenderla sia dal punto di vista dell'architettura che del modello dei costi. Al solito, la trattazione verrà sviluppata nel dettaglio per la macchina D-RISC, ma ha valore di generalità per una qualsiasi macchina Risc e, per i motivi che vedremo, anche per macchine Cisc (che adottano comunque una interpretazione degli eseguibili mediante uno pseudo-codice Risc). Occorre far notare che le macchine ILP hanno subito, a cavallo/durante gli anni 2000, un gran numero di evoluzioni e varianti architetturali. Descriverle tutte in dettaglio avrebbe uno scarso significato, tanto dal punto di vista concettuale quanto dallo stesso punto di vista tecnologico. La trattazione che verrà data ha invece l'obiettivo di catturare tutti gli aspetti essenziali e, come fatto per la CPU pipeline di base, di sviluppare un *modello dei costi* che permetta di avere una chiara idea delle prestazioni ottenibili e dell'impatto delle *ottimizzazioni statiche e dinamiche*.

Parleremo di architettura ILP **scalare**, quando ogni elemento dello stream di istruzioni contenga una sola istruzione, e di architettura ILP **superscalare** quando ogni elemento dello stream contenga più istruzioni.

Nella prima parte (sezioni 1, 2) viene approfondito l'aspetto dell'*Unità Esecutiva in parallelo (pipeline)* e le sue implicazioni sulle prestazioni e sulle direzioni tecnologiche di architetture scalari. Questo studio farà emergere le maggior parte dei problemi più significativi nel campo ILP e relative soluzioni. Un argomento importante, concettualmente e tecnologicamente, è la caratterizzazione delle ottimizzazioni statiche e dinamiche e sue ripercussioni sulle architetture e loro utilizzazione.

Nella seconda parte (sezione 3) vengono trattate le *architetture superscalari*, studiando come l'architettura scalare può essere estesa per eseguire in parallelo, a livello di ciclo di clock, istruzioni dello stesso programma. Anche in questo caso, lo studio delle ottimizzazioni statiche e dinamiche gioca un ruolo essenziale.

La terza parte (sezione 4) è dedicata ad una trattazione delle architetture ILP con *multithreading* (come sfruttare l'architettura ILP per eseguire con correntemente, a livello di ciclo di clock, più programmi o thread), ad una introduzione alle architetture *multicore* (multiprocessor su singolo chip), ed alla relazione tra architetture ILP, multithreading e multicore.

1. Architettura pipeline con Unità Esecutiva parallela

Nel Cap. XI sez. 3.4 è stato accennato alla realizzazione dell'Unità Esecutiva in parallelo per permettere di eseguire efficientemente operazioni aritmetiche complesse, sia in virgola fissa che in virgola mobile, facendo uso di tecniche di realizzazione secondo la forma di parallelismo pipeline ed altre forme.

Concettualmente, le operazioni "lunghe" in virgola fissa, come moltiplicazione e divisione, sono realizzabili con reti combinatorie aventi un ritardo di stabilizzazione dell'ordine di 1-2 cicli di clock. In pratica, però, tali realizzazioni sono molto dispendiose in termini di *area* occupata sul chip della CPU, limitando di fatto l'integrazione sul chip stesso di risorse fondamentali come le memorie cache di primo e di secondo livello ed altre. A maggior ragione, questo è vero per le operazioni aritmetiche in virgola mobile (non solo le quattro operazioni base, ma anche altre previste in alcuni processori, come la radice quadrata).

Come accennato nel Cap. XI, da tempo esistono tecniche assestate per la realizzazione in parallelo-pipeline delle operazioni aritmetiche, sia in virgola fissa che in virgola mobile. In questa sezione descriveremo il concetto alla base di tali realizzazioni e vedremo una organizzazione dell'Unità Esecutiva parallela e le sue ripercussioni sull'architettura complessiva , dopo di che estenderemo il modello dei costi dell'architettura pipeline per rilassare il vincolo della latenza unitaria dell'Unità Esecutiva (tutte le operazioni aritmetiche con latenza $t = 2\,\tau$) e per permettere anche la valutazione di programmi con istruzioni in virgola mobile.

1.1 Realizzazione in parallelo-pipeline di operazioni aritmetiche complesse

1.1.1 Operazioni lunghe in virgola fissa

Come sappiamo, la realizzazione a microprogramma dell'operazione di moltiplicazione su interi (si veda, ad esempio, Cap. IV sez. 7, esercizio 6) è caratterizzata da un tempo di servizio e da una latenza incompatibili con le esigenze di una architettura ILP. D'altra parte, se è vero che la realizzazione hardware, come rete combinatoria, ha un costo molto elevato, ben minore è il costo se gli operandi hanno un numero sostanzialmente minore di bit, ad esempio 8 bit (*byte_multiplier*). In base alla teoria sulla complessità della struttura delle reti combinatorie e sul loro ritardo di stabilizzazione in funzione del numero di variabili di'ingresso, una rete *byte_multiplier* ha, rispetto al moltiplicatore monolitico a 32 bit:

- un costo, in termini di area, molto minore di ¼,
- un ritardo di stabilizzazione minore di ¼.

Siamo dunque in grado di implementare realisticamente una unità di elaborazione specializzata ad eseguire la moltiplicazione su operandi di 8 bit, contenente il componente logico *byte_multiplier* nella parte Operativa. L'unità ha le seguenti caratteristiche:

- ciclo di clock τ tecnologicamente accettabile,
- *tempo di servizio* ideale uguale a τ,
- *latenza* uguale a τ.

Una unità-moltiplicatore a 32 bit è ora ottenibile mediante una opportuna *composizione* di unità *byte_multiplier*. Si consideri il seguente algoritmo iterativo che utilizza la funzione *byte_multiplier* come primitiva:

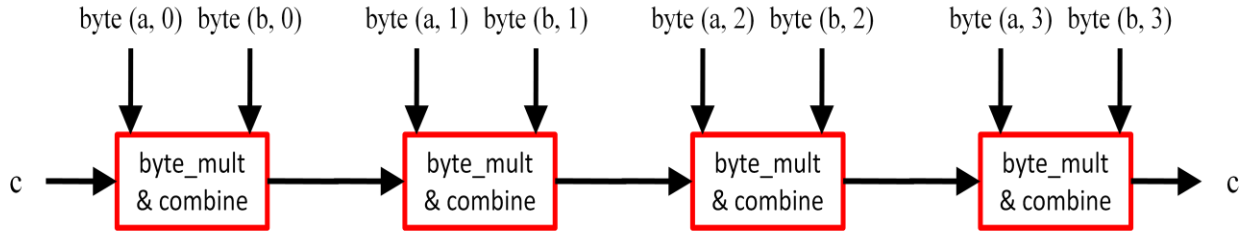
init c;

for (i = 0; i < 4; i++)

c = *combine* (c, *byte_multiplier* (*byte* (a, i), *byte* (b, i)))

L'operatore *combine* è di semplice realizzazione a partire dalle proprietà dell'operazione di moltiplicazione.

L'algoritmo iterativo precedente può essere prima trasformato mediante *loop-unfolding*, e quindi, in base a quanto studiato nel Cap. X, sez. 6.4, trasformato in una struttura *pipeline a 4 stadi*, dove ogni stadio è realizzato da una unità-moltiplicatore a 8 bit, come mostrato nella figura seguente:



Se tale struttura pipeline è integrata sullo stesso chip, si ha che il ***tempo di servizio*** effettivo della moltiplicazione su interi è uguale a

$$T_{s\text{-}lunghe} = t = 2\,\tau$$

senza alcuna degradazione rispetto ad una ipotetica realizzazione hardware di un moltiplicatore monolitico a 32 bit: applicandogli uno stream di richieste di operazioni di moltiplicazione tutte tra loro indipendenti, la realizzazione pipeline ha lo stesso tempo di servizio della realizzazione sequenziale monolitica, ma con un costo molto minore e del tutto compatibile con i requisiti dell'integrazione su singolo chip.

Del tutto analoga è la realizzazione in pipeline della *divisione* intera.

Quella vista è quindi la realizzazione da applicare all'Unità Esecutiva di una CPU pipeline.

La ***latenza*** di una moltiplicazione/divisione su interi è invece uguale a 8τ. In generale, per una realizzazione su s stadi, la latenza è data da:

$$L_{s\text{-}lunghe} = s\,t = 2\,s\,\tau$$

Per quanto sia abbastanza contenuta, tale latenza rappresenta la contropartita che dobbiamo pagare per abbassare il costo di realizzazione mantenendo costante il tempo di servizio: **la conseguenza sulle prestazioni** (tempo di servizio per istruzione) **riguarda il ritardo Δ causato da dipendenze logiche indotte da operazioni aritmetiche "lunghe"**. Questo aspetto è alla base dello sviluppo di una grande varietà di tecniche di ottimizzazione, sia statiche che dinamiche, introdotte nelle architetture ILP, come studieremo nella prossime sezioni.

Si noti che le operazioni aritmetiche "corte", così come la fase finale dell'esecuzione di istruzioni di LOAD, continuano ad avere sia tempo di servizio effettivo che tempo di latenza uguale a $t = 2\tau$:

$$T_{s\text{-}corte} = L_{s\text{-}corte} = t = 2\tau$$

Una Unità Esecutiva parallela può quindi disporre sia di una "normali" *Unità Funzionali per operazioni "corte" e per LOAD*, sia di *Unità Funzionali Moltiplicatore/Divisore*.

1.1.2 Operazioni in virgola mobile

Secondo una prima visione del livello assembler (Cap. V, sez. 3), una macchina Risc non dispone di istruzioni primitive per operazioni aritmetiche in virgola mobile. Questa visione può essere ora superata per architetture ILP, a condizione che si disponga di realizzazioni hardware per tali operazioni.

Il set di istruzioni può quindi prevedere apposite istruzioni in virgola mobile, sia per macchine Cisc che per macchine Risc. Una **estensione della macchina D-RISC** può essere la seguente:

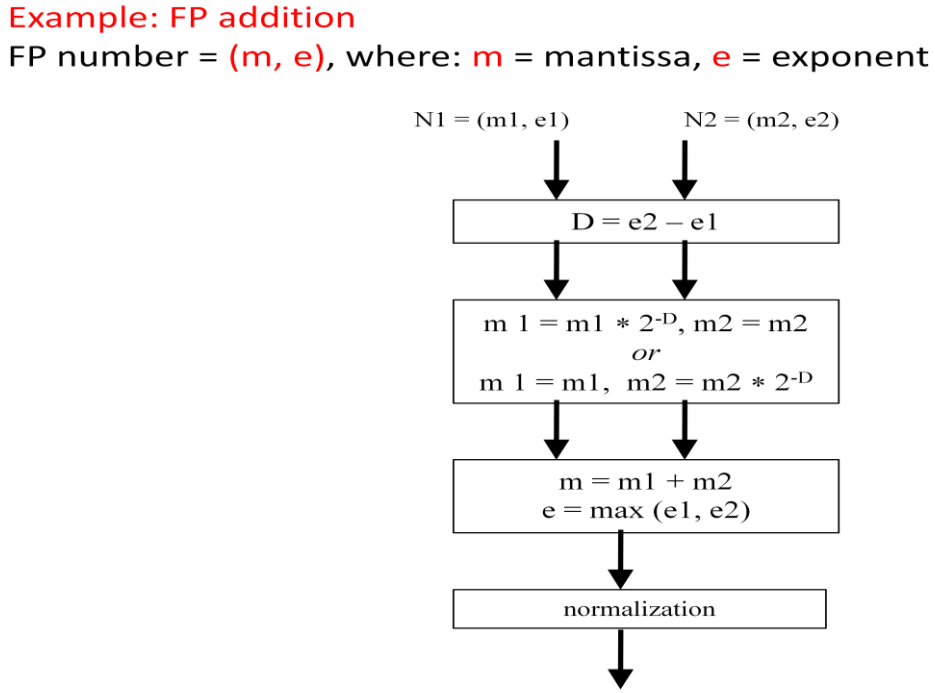
- prevediamo un apposito array di **Registri in Virgola Mobile**, *RF[64]*, ognuno a 64 bit. Essi vengono usati solo per operazioni aritmetiche in virgola mobile (non sono registri generali);
- prevediamo apposite **istruzioni aritmetiche in virgola mobile**, come

  ADDF RFi, RFj, RFk

  e analoghe (SUBF, MULF, DIVF, SQRT, ecc), così come istruzioni *LOADF* e *STOREF*, e istruzioni di salto condizionato *IFF* con predicato applicato a contenuti di registri in virgola mobile.

L'Unità Esecutiva di una architettura ILP conterrà una o più *Unità Funzionali in Virgola Mobile* e l'array dei Registri in Virgola Mobile.

La realizzazione in pipeline delle operazioni in virgola mobile è completamente nota. A loro volta esse fanno uso, per gli stadi che manipolano mantissa ed esponente, di realizzazioni in pipeline del tutto analoghe a quelle viste nella sezione precedente. Ad esempio, per l'addizione in virgola mobile:



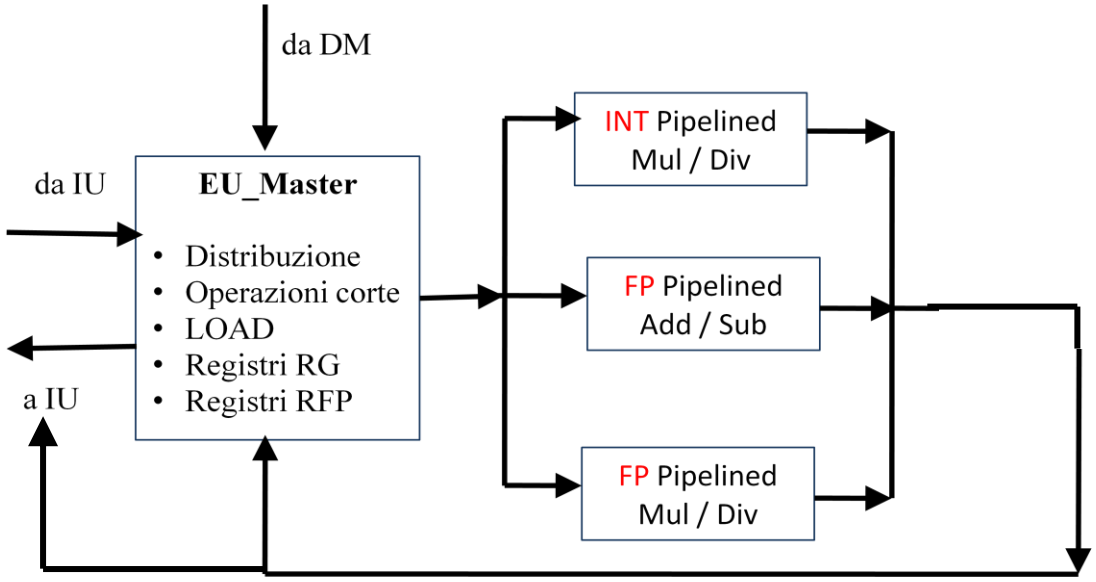
Nell'esempio dell'addizione in virgola mobile, il pipeline dispone di 3-4 stadi: il primo ed eventualmente il secondo allineano le mantisse (*m*), il secondo effettua l'addizione delle mantisse allineate e determina l'esponente del risultato (*e*), il terzo effettua la normalizzazione. Nel caso di moltiplicazione/divisione in virgola mobile, adottando realizzazioni come quelle della sezione precedente per la parte centrale dell'esecuzione, si possono avere 8 - 12 stadi.

Come visto nella sezione precedente, il *tempo di servizio* è quello minimo ($2\ \tau$), mentre la *latenza* è proporzionale al numero *s* degli stadi ($2\ s\ \tau$).

1.1.3 Unità Esecutiva parallela

Limitatamente all'architettura base della CPU pipeline (per l'architettura superscalare vedremo in seguito), l'Unità Esecutiva può essere definita secondo lo schema seguente, contenente:

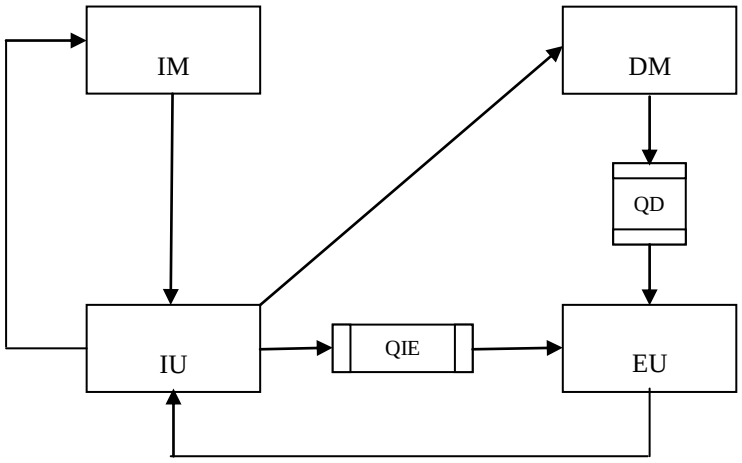
- una Unità Funzionale pipeline *moltiplicatore/divisore in virgola fissa*,
- una Unità Funzionale pipeline *addizionatore in virgola mobile*,
- una Unità Funzionale pipeline *moltiplicatore/divisore in virgola mobile*, che può anche includere altre operazioni in virgola mobile, come la radice quadrata,
- l'unità *EU_Master*.



L'unità indicata con *EU_Master* interfaccia IU e DM e *contiene la copia principale dei registri generali (RG) ed i registri in virgola mobile RF*. Ricevendo una istruzione dalla IU, il funzionamento è il seguente:

- se l'operazione richiesta da IU è una aritmetica corta o una LOAD, *la esegue direttamente* (per la LOAD attende il dato da DM) e invia a IU il valore del registro generale destinazione ed il suo indirizzo;
- se l'operazione è lunga in virgola fissa, oppure è in virgola mobile, la distribuisce all'Unità Funzionale corrispondente *insieme ai valori degli operandi* letti dai registri RG o RF rispettivamente.

Alla luce dei tempi di elaborazione variabili, è ora conveniente che i collegamenti con IU e DM abbiano un grado di asincronia maggiore di uno (tipicamente dell'ordine di qualche unità, ad esempio 8), e quindi che siano realizzati mediante **Unità Coda** (Cap. IV, sez. 5.5.2):



Lo schema complessivo della CPU è mostrato nella figura, dove le sono indicate le code tra IU ed EU (QIE) e tra DM ed EU (QD), cioè sui collegamenti per i quali è più probabile si possano formare accumuli momentanei di messaggi, mentre si è mantenuto grado di asincronia uguale a uno per tutti gli altri collegamenti (pur non escludendo che anch'essi possano contenere code, se conveniente).

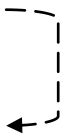
I risultati delle operazioni eseguite dalle Unità Funzionali sono attesi da EU_Master *nondeterministicamente* e scritti nei registri in virgola fissa o in virgola mobile. I risultati in virgola fissa sono inviati dalle Unità Funzionali direttamente anche alla IU insieme all'indirizzo del registro destinazione.

Dato che le operazioni realizzate in pipeline hanno latenza diversa tra loro e maggiore delle aritmetiche corte, *più operazioni aritmetiche possono essere in esecuzione contemporaneamente*, ad esempio una sequenza di operazioni tra loro *indipendenti*, della quali una sia una operazione corta in virgola fissa o Load e le altre operazioni lunghe in virgola fissa o operazioni in virgola mobile.

Il parallelismo tra operazioni aritmetiche introduce un ulteriore tipo di dipendenze logiche, quelle *all'interno della EU* stessa. Useremo il termine **dipendenze logiche EU-EU**, per distinguerle da quelle studiate nel Cap. XI, che verranno d'ora in poi dette *dipendenze logiche IU-EU*.

Ad esempio nella sequenza di istruzioni:

1. MUL Ra, Rb, Rc
2. MULF RFx, RFy, RFz
3. INCR Rc



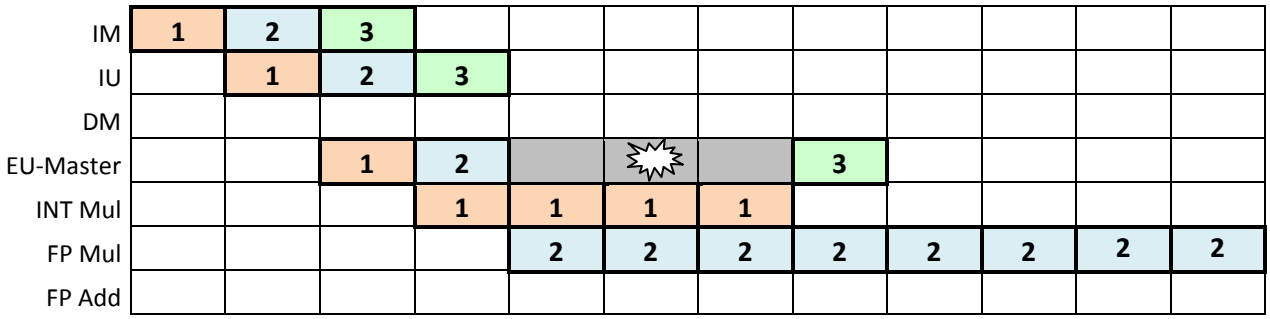
la MUL e la MULF, così come la MULF e la INCR, sono indipendenti e possono essere eseguite in parallelo da parte di unità diverse (Unità Funzionali distinte per MUL e MULF, e la stessa EU_Master per la INCR). Inoltre, esiste una dipendenza logica EU-EU di *distanza* 2 tra la MUL e la INCR.

Nella sezione successiva studieremo l'effetto delle dipendenze logiche EU-EU sulle prestazioni. A livello di architettura firmware, occorre che EU_Master disponga di *meccanismi per la sincronizzazione dei registri RG e dei registri RF*, del tutto simili a quelli semaforici adottati nella IU (Cap. XI, sez. 3.3).

Nell'esempio, EU_Master, ricevendo la richiesta di MUL, verifica che i semafori associati ai registri di indirizzo Ra e Rb siano a zero. In tal caso, incrementa il semaforo associato a Rc e invia all'Unità Funzionale l'istruzione stessa e i valori RG[Ra] e RG[Rb]. Ricevendo la richiesta di INCR, verifica che il semaforo associato al registro di indirizzo Rc sia a zero prima di poterla eseguire. Ricevendo dall'Unità Funzionale la coppia (valore *x*, indirizzo Rc), scrive *x* in RG[Rc], e decrementa il semaforo associato a tale registro. In parallelo la coppia (valore *x*, indirizzo Rc) è stata inviata dall'Unità Funzionale anche alla IU.

Si capisce come il potenziale parallelismo tra operazioni all'interno della EU introduca una prima, semplice forma di funzionamento *out-of-order*: la fase di esecuzione delle istruzioni in EU viene *iniziata* nello stesso ordine con il quale sono inviate dalla IU, ma il loro *completamento* può avvenire in un ordine diverso.

Nell'esempio precedente, l'esecuzione delle istruzioni MUL e MULF procede in parallelo e, al completamento della MUL, può iniziare la INCR anche se la MULF non è ancora terminata. Questa situazione è mostrata nella simulazione della figura seguente, supponendo che l'Unità Funzionale per la moltiplicazione in virgola fissa abbia 4 stadi e l'Unità Funzionale per la moltiplicazione in virgola mobile ne abbia 8 (come nel Cap. XI, ogni intervallo è di durata $t = 2\tau$):

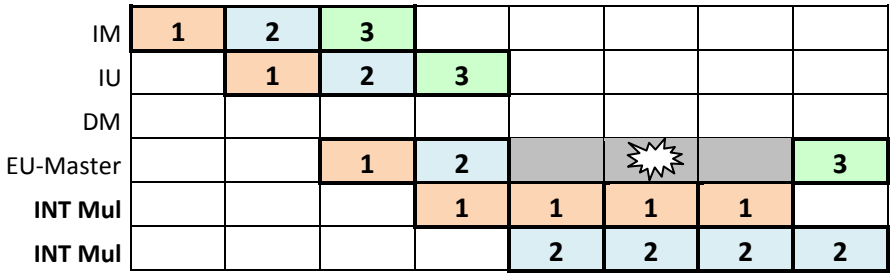


Nell'esempio, la dipendenza logica causa una *bolla* di ampiezza *3t nell'elaborazione della EU_Master*. A seconda della struttura complessiva del programma, questa bolla potrà non avere effetto oppure ripercuotersi, in tutto o in parte, sul tempo di servizio e quindi sul tempo di completamento. In ultima analisi, **l'effetto dovrà essere valutato sul tempo di servizio della IU.**

Si noti che *il parallelismo tra istruzioni aritmetiche indipendenti si ha anche tra istruzioni che utilizzano una stessa Unità Funzionale*, grazie alla struttura pipeline. Ad esempio in:

1. MUL Ra, Rb, Rc
2. MUL Rx, Ry, Rz
3. INCR Rc

le istruzioni indipendenti 1 e 2 utilizzano la stessa Unità Funzionale, nella quale le coppie di operandi (RG[Ra], RG[Rb]) e (RG[Rx], RG[Ry]) "entrano" distanziate di un intervallo *t*:



In questa rappresentazione grafica, si intende che l'istruzione 1 e l'istruzione 2 sono eseguite in pipeline nella *stessa* Unità Funzionale.

Nota: forme di parallelismo adottate nell'Unità Esecutiva

In accordo alla trattazione del Cap. X, la parallelizzazione della EU fa uso del paradigma con *partizionamento funzionale* (Cap. X, sez. 9), dove ogni worker è a sua volta realizzato in pipeline.

Una Unità Esecutiva parallela potrebbe anche essere realizzata secondo altre forme di parallelismo su stream, ad esempio secondo il paradigma *farm*. Ciò avrebbe però comportato la completa replicazione di Unità Funzionali generali (cioè capaci di eseguire qualsiasi operazione), e questo avrebbe nuovamente introdotto il problema del costo in termini di area del chip.

D'altra parte, il potenziale svantaggio del paradigma con partizionamento funzionale, cioè il possibile sbilanciamento del carico tra worker, in questa applicazione non si presenta: infatti, poiché ogni worker è realizzato in pipeline, anche un accumulo di richieste *indipendenti* alla stessa Unità Funzionale non provoca degradazioni del tempo di servizio rispetto alla soluzione farm con lo stesso grado di parallelismo, come evidenziato dall'ultimo esempio appena sopra.

Questa considerazione vale rigorosamente per l'architettura pipeline di base, ma potrebbe non essere vera per le architetture superscalari (sez. 3).

Esempi di Unità Esecutive di processori commerciali

Le famiglie dei processori Risc (come MIPS, Power PC, DEC Alpha), e Cisc x-86 (Intel, AMD) sono normalmente caratterizzate da configurazioni di EU comprendenti più Unità Funzionali (da 2 a 8) per operazioni in virgola fissa e LOAD, delle quali almeno una moltiplicatore/divisore, e più Unità Funzionali (da 1 a 4) per operazioni in virgola mobile.

Come visto, il numero di stadi dipende dall'algoritmo utilizzato e dalla lunghezza di parola per i dati.

Parallelismo ridondante per lo stesso tipo di unità si ha per le architetture superscalari.

1.2 Funzionamento in-order *vs* out-of-order: una prima analisi

L'aspetto dell'Unità Esecutiva parallela può essere interpretato in un modo più generale. Nel Cap. X, sez. 7, è stato introdotto il **modello data-flow** come una forma computazionale di base per computazioni in parallelo. Nel nostro caso, per quanto l'intera CPU sia realizzata secondo il tradizionale modello di Von Neumann, *l'Unità Esecutiva può essere pensata come un sottosistema data-flow*, dove la EU_Master si occupa

- di determinare le condizioni di abilitazione delle istruzioni ricevute,
- in caso di abilitazione, di inviarle alle opportune Unità Funzionali assieme ai valori degli operandi,
- di ricevere i risultati delle istruzioni e di renderli disponibili ad altre istruzioni da abilitare.

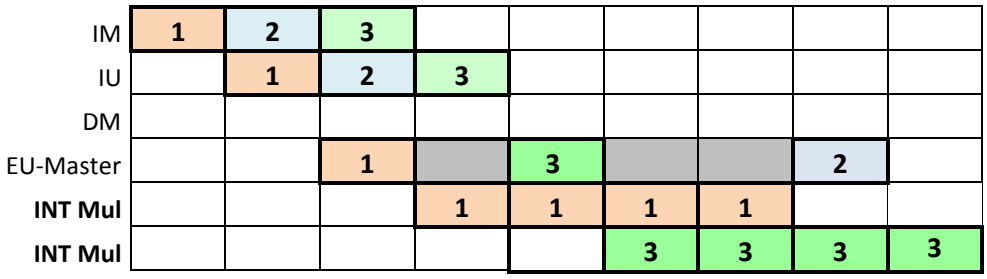
(Nel Cap. XI, sez. 7.2 è accennato a possibili schemi architetturali interamente non Von Neumann che applicano il paradigma data-flow a livello di macchina assembler non imperativa).

In altre parole, quello che viene chiamato funzionamento *out-of-order* altro non è – se applicato alla sola Unità Esecutiva – che un *ordinamento parziale* delle istruzioni aritmetiche secondo le dipendenze sui dati.

L'ordinamento delle istruzioni da parte di un compilatore ottimizzante permette di ottenere buone prestazioni facendole iniziare *in-order*, cioè *prelevandole dalla coda d'ingresso in ordine FIFO*, ma permettendo il loro completamento *out-of-order*, come abbiamo supposto in precedenza.

Questa prima, limitata accezione di out-of-ordering permette già di ottenere ottimizzazioni significative. Ad esempio, supponiamo che la EU_Master rilevi (attraverso i semafori associati ai registri) che una istruzione I1 prelevata dalla coda non è abilitata in quanto non ha ancora tutti gli operandi pronti. Se la prossima istruzione in coda, I2, è abilitata, essa può essere eseguita (direttamente da EU_Master o da una Unità Funzionale). EU_Master ha registrato che la I1 è momentaneamente bloccata e, ad ogni scrittura nei registri operandi di I1, verifica che gli operandi stessi siano pronti (i semafori valgano zero); quando ciò avviene, la I1 può essere eseguita. Ad esempio:

1. MUL Ra, Rb, Rc
2. INCR Rc
3. MUL Rx, Ry, Rz



In generale, può darsi che dopo aver prelevato la I2, la EU_Master rilevi che in coda *ci siano ancora, secondo l'ordine FIFO, una o più istruzioni abilitate*: anch'esse possono essere eseguite, mentre viene continuamente verificata (ad ogni ciclo di clock) l'abilitazione di I1.

In conclusione, in alcuni casi, le bolle introdotte a causa di dipendenze logiche EU-EU possono essere *mascherate,* e quindi non avere effetto in tutto o in parte, prelevando dalla coda d'ingresso, in ordine FIFO, una o più istruzioni abilitate.

Il funzionamento out-of-order può essere implementato anche secondo una accezione più vasta: *la coda d'ingresso alla EU potrebbe essere esaminata non in ordine FIFO, bensì in modo associativo* da parte della EU_Master, allo scopo di scoprire istruzioni abilitate indipendentemente dal loro ordine. Potenzialmente, l'ottimizzazione delle prestazioni è maggiore, a scapito di una ben maggiore complessità dell'architettura firmware.

La maggiore complessità del progetto e della struttura della CPU deriva dall'esame associativo della coda d'ingresso (delle code d'ingresso) e dal fatto che, in generale, le istruzioni fuori ordine bloccate possono essere in numero significativo e non noto a priori.

Inoltre, nelle macchine *Cisc,* poiché la complessità aumenterebbe ancora nel tentativo di applicare le ottimizzazioni out-of-order al codice originario, la stessa struttura firmware provvede, in maniera del tutto invisibile al codice da eseguire, a *interpretare* le istruzioni complesse mediante sequenze di istruzioni più semplici (Risc-like Operations), con non pochi overhead aggiuntivi.

Si può quindi avere una doppia accezione del funzionamento parallelo della EU:

- **in-order**: più precisamente con *verifica dell'abilitazione delle istruzioni secondo l'ordine (FIFO) con cui sono state inviate* da IU, ed eventuale completamento out-of-order. Questa situazione include il caso significativo che, trovando una istruzione bloccata da dipendenza logica, successive istruzioni abilitate, esaminate una alla volta in ordine FIFO, possano iniziare;
- **out-of-order**: più precisamente la cui *abilitazione sia verificata secondo un ordine* (*associativo) diverso rispetto all'ordine con cui sono state inviate* da IU.

Questa distinzione può essere messa in relazione con il problema del compromesso tra **complessità degli strumenti di compilazione *vs* complessità dell'architettura firmware**:

- l'approccio in-order (*verifica dell'abilitazione in-order*) semplifica il progetto firmware, anzi praticamente non introduce alcuna complessità aggiuntiva significativa. Agli effetti delle prestazioni, si confida, oltre che nel parallelismo che è possibile ottenere dall'esecuzione di istruzioni iniziate in ordine, anche su **ottimizzazioni statiche** del codice introdotte a compilazione;
- l'approccio out-of-order (*verifica dell'abilitazione out-of-order*) complica notevolmente il progetto firmware, con ripercussioni sulla maggiore area necessaria sul chip. Ali effetti delle prestazioni, si confida sul fatto che le **ottimizzazioni dinamiche** così ottenute siano sufficienti, senza utilizzare tecniche particolarmente spinte per le ottimizzazioni statiche o addirittura *senza* applicare alcuna trasformazione statica.

Molte architetture ILP hanno privilegiato un approccio dinamico alle ottimizzazioni. La ragione principale risiede nell'esigenza di garantire la *compatibilità binaria* di codici commerciali/legacy esistenti, spesso anche vecchi di molti anni, a fronte dell'evoluzione delle architetture firmware (a parità di macchina assembler). Per definizione, se di tali codici è disponibile solo l'eseguibile binario, non è possibile, o è estremamente difficile, introdurre delle ristrutturazioni statiche.

D'altra parte, l'esigenza di ottimizzare nuove applicazioni, o comunque applicazioni delle quali sia disponibile il codice sorgente, è ugualmente molto sentita nel mondo industriale.

Inoltre, l'evoluzione tumultuosa delle architetture superscalari out-of-order ha portato la complessità dei chip CPU a limiti estremi, con conseguenze negative tanto in termini di rapporto prestazioni/costo (una nuova versione molto più complessa ha talvolta comportato un guadagno poco significativo in performance), quanto anche e soprattutto di **consumo energetico**. Questi fenomeni hanno bloccato l'evoluzione verso architetture con sempre più sofisticati meccanismi di out-of-ordering: l'"arresto della legge di Moore" per uniprocessor (applicata al ciclo di clock) è in stretta relazione con questo aspetto, portando ad una rivalutazione delle validità di *architetture in-order*, o con una "ragionevole" utilizzazione di out-of-ordering, e della conseguente validità di *strumenti di compilazione ottimizzanti*.

L'attuale tendenza delle emergenti architetture *multicore* è in stretta relazione con questa problematica: sistemi con un elevato numero di processori on-chip, ma con processori relativamente semplici (pipeline o superscalari in-order).

Approfondiremo questa analisi nelle sezioni 2, 3, 4.

Per tutta la trattazione delle sezione 1, supporremo che la EU abbia il funzionamento con *verifica dell'abilitazione* in-order.

Anche per l'Unità Istruzioni potremmo avere un funzionamento *out-of-order*, in particolare prevedendo che tale unità non si blocchi in caso di dipendenza logica e che continui a preparare ed eseguire/inoltrare istruzioni successive che siano abilitate. Per il seguito della sezione 1 supporremo che la IU abbia un funzionamento rigorosamente *in-order*: la preparazione di una istruzione non ha luogo finché non si è completata quella dell'istruzione precedente.

Queste ipotesi ci consentono di comprendere la natura dei problemi, studiare alcune soluzioni concrete e adottate in macchine esistenti, e sviluppare un modello dei costi. Nella sezione 2 verranno discusse le conseguenze derivanti dal rilassare tali assunzioni.

1.3 Modello dei costi di architettura pipeline con Unità Esecutiva parallela

Nell'espressione del tempo di servizio per istruzione:

$$T = (1 + \lambda)\, t + \Delta$$

nessuna modifica viene apportata alla valutazione della degradazione dovuta a salti, mentre deve essere resa più generale la valutazione della degradazione dovuta a dipendenze logiche:

- oltre alle dipendenze *IU-EU*, vanno considerate anche quelle *EU-EU*. Le seconde possono avere influenza sulla valutazione delle prime, che in ultima analisi sono quelle che interessano nel modello dei costi;
- deve essere valutato l'effetto della *latenza* di operazioni aritmetiche in pipeline, in quanto essa può influenzare direttamente il ritardo introdotto nell'elaborazione della IU quando si verifica una dipendenza logica.

Ricordiamo che, nella formula del tempo di servizio, adottata nel Cap. XI nell'ipotesi di EU con latenza unitaria (ogni operazione aritmetica abbia latenza uguale a $t = 2\tau$):

$$\Delta = t \sum_{k=1}^{\bar{k}} d_k \ (N_{Q_k} + 1 - k)$$

l'effetto della latenza delle operazioni aritmetiche era rappresentato dal termine "*1*" nella sommatoria, mentre l'effetto della distanza delle dipendenze logiche è rappresentato dal termine "$- k$".

Manterremo il significato del termine N_{Qk}, per tenere conto della latenza introdotta dalla DM (a causa di eventuali istruzioni di LOAD nella sequenza di istruzioni, tutte eseguibili dalla EU, che si conclude con l'istruzione che induce la dipendenza logica).

Nota sul fattore di utilizzazione dell'Unità Esecutiva

Nel Cap. XI, il tempo di servizio per istruzione è assunto essere quello dell'Unità Istruzioni, in quanto il tempo di servizio T_{EU} della EU di latenza unitaria è uguale a *t*. Ora questa valutazione del tempo di servizio continua ad essere valida in molti casi, ma non in generale.

Nel *caso peggiore*, in cui il programma consti di tutte istruzioni aritmetiche e che ognuna sia legata alla precedente da dipendenza EU-EU, il tempo di servizio della EU è uguale alla latenza media delle operazioni aritmetiche eseguite. In tal caso, il fattore di utilizzazione della EU

$$\rho_{EU} = \frac{T_{EU}}{T_{interarrivo}}$$

sarebbe maggiore di uno, per cui il tempo di servizio della CPU sarebbe in realtà uguale a quello della EU. Infatti, indipendentemente dal numero di posizioni previste per la realizzazione delle code, a regime queste (o almeno QIE) sarebbero mediamente piene e la IU dovrebbe adeguarsi al tempo di servizio della EU a causa della sincronizzazione sulle interfacce ed alla condizione di coda piena.

In generale il tempo di servizio per istruzione è dato da:

$$T = max\ (T_{IU}, T_{EU})$$

dove T_{IU} è valutato secondo la formula del modello dei costi:

$$T_{IU} = (1 + \lambda)\ t + \Delta$$

Comunque, nella maggior parte dei casi, il fattore di utilizzazione della EU è minore di uno, non solo perché il tempo di interarrivo è maggiore di T_{IU} (non tutte le istruzioni sono eseguite dalla EU), ma soprattutto in conseguenza del *funzionamento cliente-servente* (Cap. X, sez. 12) quando si verifichino delle dipendenze logiche IU-EU.

Per questi motivi, a meno che non si presentino casi eccezionali da evidenziare, nel seguito della sez. 1 continueremo a valutare il tempo di servizio come

$$T = T_{IU}.$$

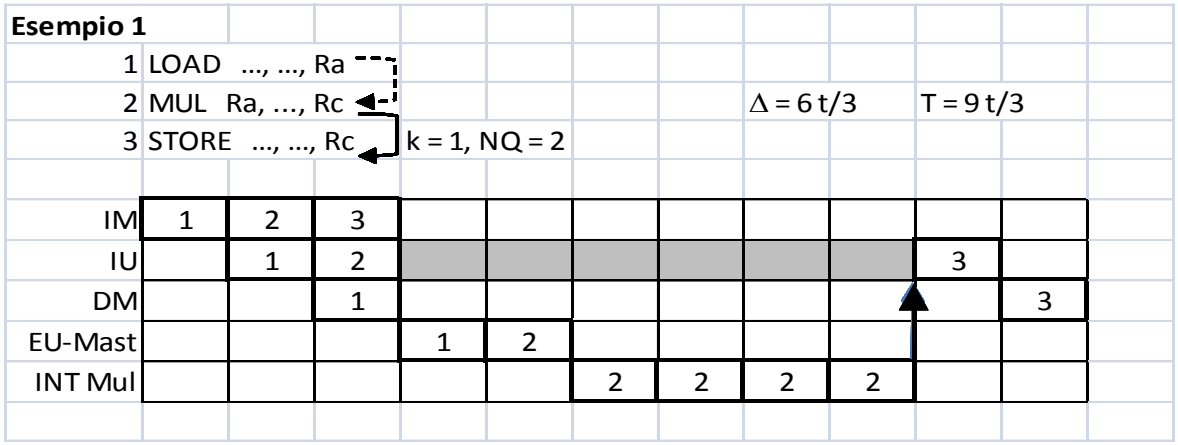
Nota sul comportamento delle memorie cache

In tutti gli esempi di questa sezione trascureremo l'overhead sul tempo di servizio causato da eventuali fault di cache. In ogni caso, dal Cap. XI sappiamo come correggere la valutazione dei tempi di completamento.

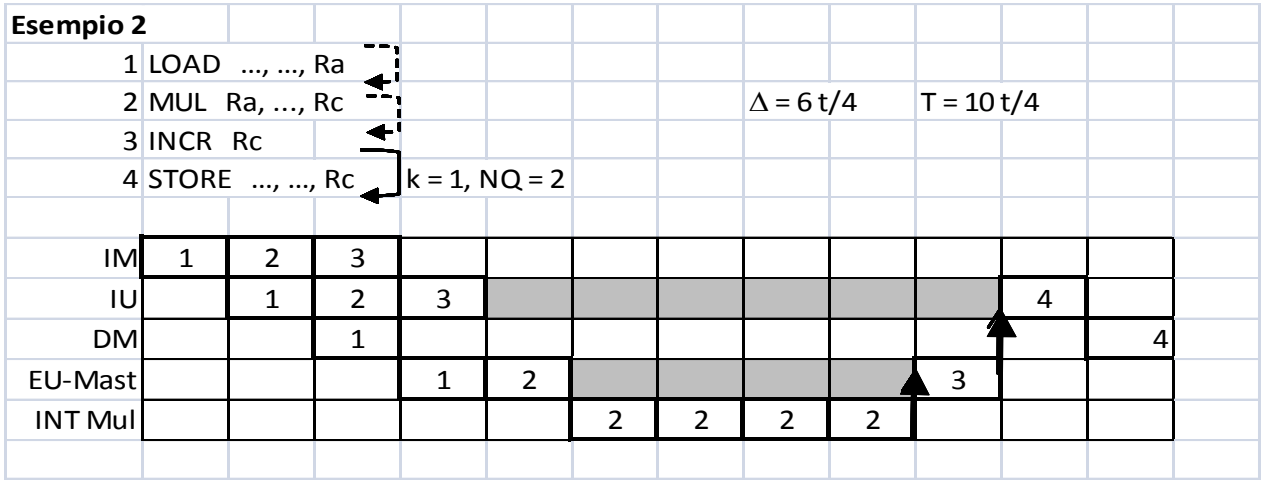
Poiché la presenza di gerarchie di memoria rappresenta un aspetto da approfondire ulteriormente, il problema verrà discusso in vari punti delle sez. 2 e 3.

1.3.1 Primi esempi

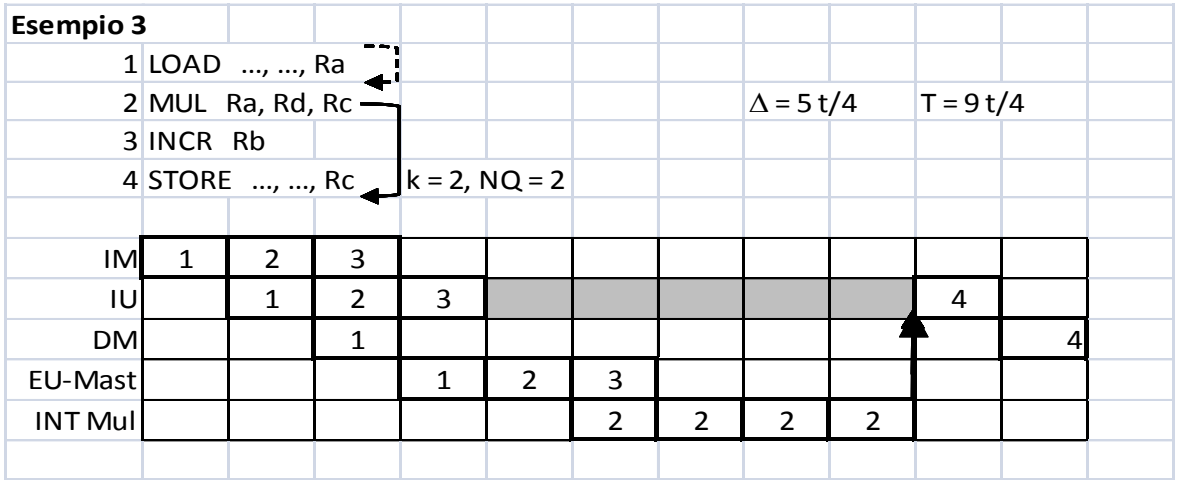
Nelle figure seguenti sono riportate le simulazioni di alcuni casi significativi di dipendenze logiche IU-EU (indicate con frecce continue) influenzate dalla latenza della EU e da dipendenze logiche EU-EU (indicate con frecce tratteggiate). Senza perdere in generalità, gli esempi fanno tutti uso, per le operazioni lunghe, di moltiplicazione in virgola fissa, e si suppone che questa venga realizzata con una Unità Funzionale pipeline a 4 stadi.



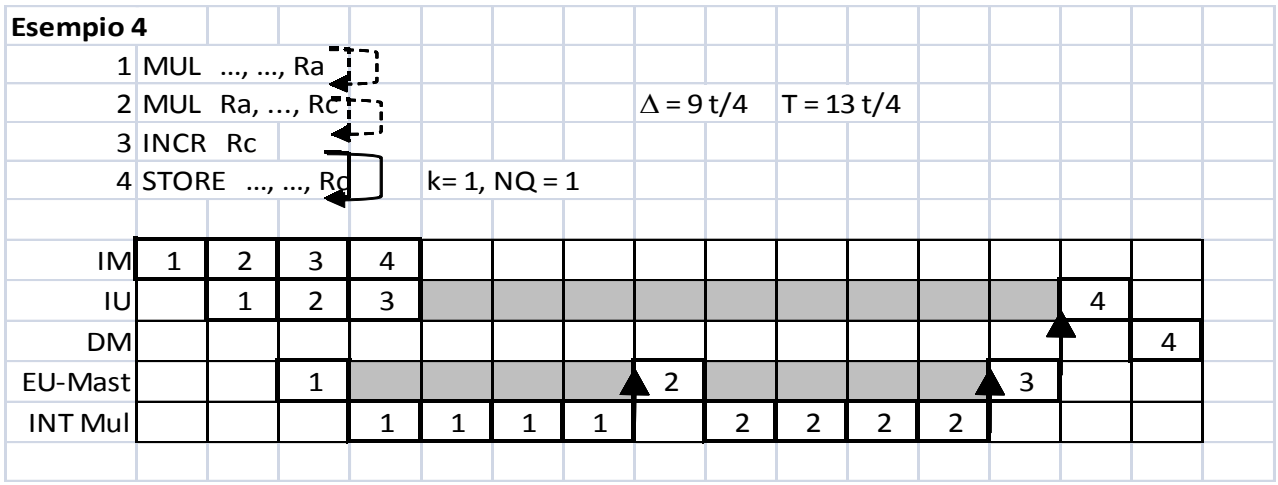
Esempio 1bis: senza la LOAD si ha: $\Delta = 5\ t/2$, $T = 7\ t/2$.



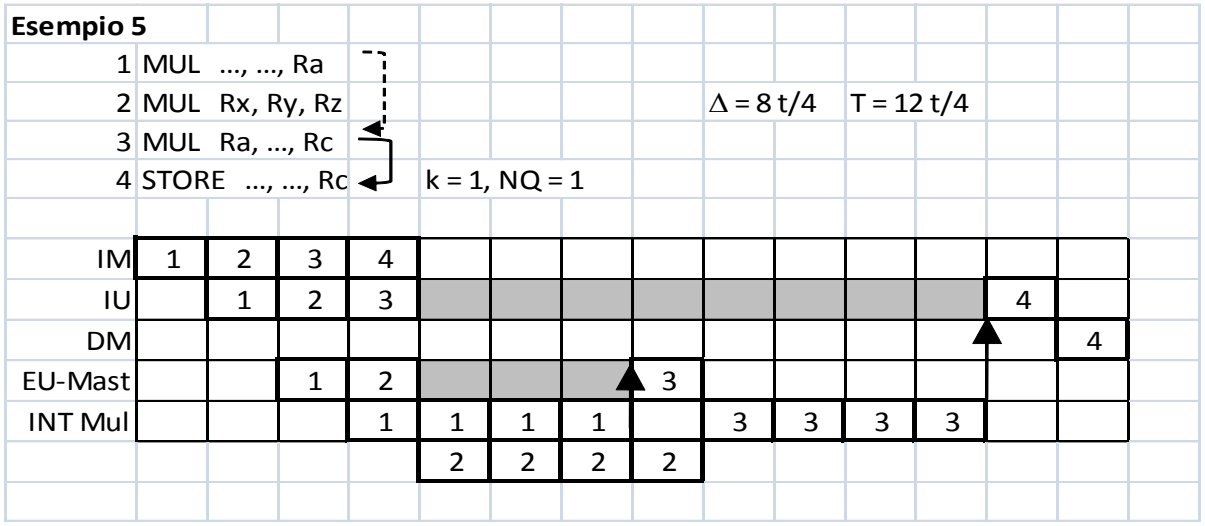
Esempio 2 bis: senza la LOAD si ha: $\Delta = 5\ t/3$, $T = 8\ t/3$.



Esempio 3 bis: senza la LOAD si ha: $\Delta = 4\ t/3$, $T = 7\ t/3$.



Esempio 4bis: se l'istruzione 3 non modifica Rc, quindi $k_{IU} = 2$: $\Delta = 8\ t/4$, $T = 12\ t/4$.



Esempio 5bis: se l'istruzione 2 è una aritmetica corta, che non modifica Rc (Esempio, INCR Rb), niente cambia. Quindi $\Delta = 8\ t/4$, $T = 12\ t/4$.

Nell'Esempio 1, la dipendenza logica EU-EU non ha di per sé alcun effetto, in quanto la LOAD è eseguita da EU_Master, che quindi è subito pronta alla preparazione della MUL. La LOAD continua ovviamente ad avere impatto su N_Q. La bolla introdotta nel funzionamento della IU è di ampiezza *6t*: cioè, alla bolla di *2t* (che comunque si sarebbe avuta essendo $N_Q = 2$, anche se la moltiplicazione avesse avuto latenza unitaria), si aggiunge la latenza di *4t* dovuta all'esecuzione in pipeline della moltiplicazione. L'Esempio 1bis differisce per il fatto che $N_Q = 1$ (bolla ampia *5t*).

L'Esempio 2 mostra come, se anche la dipendenza IU-EU è indotta da una istruzione aritmetica corta (3), occorre ugualmente tenere conto della *latenza di eventuali precedenti istruzioni lunghe* (2). Precisamente, l'istruzione lunga 2 appartiene alla sequenza di istruzioni, tutte eseguite dalla EU, che si concludono con l'istruzione 3 che induce la dipendenza IU-EU. Nell'Esempio 2 tale sequenza include una LOAD, a differenza dell'Esempio 2bis.

L'Esempio 3 è relativo ad una dipendenza IU-EU di distanza 2 con $N_Q = 2$ (con $N_Q = 1$ nell'Esempio 3bis) indotta dall'istruzione aritmetica lunga 2 sulla 4. Poiché la distanza è uguale a 2,

necessariamente l'istruzione 3 è indipendente dalla 2. Come si vede, la 3 è quindi eseguita in parallelo alla 2, e si conclude prima: come discusso in precedenza, questo limitato fenomeno di *out-of-ordering* non provoca alcun problema di consistenza, ed ha conseguenze positive sulle prestazioni contribuendo ad aumentare la distanza delle dipendenze IU-EU (la bolla è di ampiezza *5t)*.

Dal punto di vista architetturale, l'unità EU_Master, dopo avere inoltrato l'istruzione 2 all'Unità Funzionale (moltiplicatore in virgola fissa), riceve l'istruzione 3 e, *verificando che i suoi operandi sono pronti* (il registro Rb è aggiornato), la può eseguire immediatamente. In questo caso, la 3 poteva anche essere inserita tra la 1 e la 2, aumentando la distanza della dipendenza logica EU-EU. Il compilatore ha invece scelto di sfruttare l'istruzione 3 per aumentare la distanza della dipendenza logica IU-EU, confidando sul fatto che comunque, a tempo di esecuzione, la EU stessa sarebbe stata capace di eseguirla in parallelo alla 2.

In questo caso, è la EU_Master stessa ad eseguire direttamente la 3, *ma niente sarebbe cambiato se fosse stata inoltrata ad una Unità Funzionale, inclusa la stessa unità che ha iniziato in pipeline la 2*. Infatti, trattandosi di operazioni indipendenti, esse possono "entrare" nella stessa struttura pipeline distanziate della minima distanza di tempo possibile ($t = 2\tau$).

L'Esempio 4 mostra il caso di una sequenza di istruzioni, l'ultima delle quali induce la dipendenza logica IU-EU, e che contiene *più istruzioni aritmetiche lunghe tra loro dipendenti*. Nel valutare la latenza che contribuisce a formare la bolla nella IU, occorre quindi considerare *la latenza complessiva* spesa negli stadi delle Unità Funzionali in pipeline, nel nostro caso *8t*. Nell'Esempio 4bis, l'istruzione 3, indipendente dalla 2, viene eseguita immediatamente dopo che la 2 è stata inoltrata all'Unità Funzionale, questo fa accorciare la bolla di *1t*.

L'Esempio 5 mostra esplicitamente quanto affermato alla fine del commento dell'Esempio 3: *l'Unità Funzionale moltiplicatore in pipeline esegue contemporaneamente l'istruzione 1 e l'istruzione 2*, essendo esse indipendenti. Inoltre, alla conclusione della 1, la 3 viene sbloccata nella EU_Master, e quindi si ha parziale contemporaneità anche tra la 2 e la 3. Come fatto notare nell'esempio 5bis, niente sarebbe cambiato se, invece che una aritmetica lunga, l'istruzione 2 fosse stata una aritmetica corta in virgola fissa.

1.3.2 Il modello

Alla luce degli esempi e delle considerazioni precedenti, il ritardo Δ causato da dipendenze logiche verrà espresso in modo da *separare gli effetti* delle dipendenze logiche IU-EU da quelli delle dipendenze logiche EU-EU.

Nel seguito

- si continuerà ad indicare con k la distanza delle dipendenze logiche IU-EU e con d_k la rispettiva probabilità;
- con h si indicherà la distanza delle dipendenze logiche EU-EU e con d_h la rispettiva probabilità.

Per brevità useremo il termine **sequenza critica** per esprimere il concetto già utilizzato più volte: *la sequenza di istruzioni, tutte eseguibili dalla EU, che si conclude con l'istruzione che induce la dipendenza logica*. Questo concetto servirà, oltre che a valutare N_Q come nel Cap. XI, a separare gli effetti delle dipendenze IU-EU da quelle EU-EU.

Il modello è caratterizzato come segue:

1. si valuta il ritardo Δ_1 sulla IU, causato dalle dipendenze logiche IU-EU: per ognuna si considera *soltanto* la presenza dell'istruzione che induce la dipendenza logica e si tiene conto della sequenza critica *soltanto* per la valutazione del ritardo N_Q (come nel Cap. XI), ignorando l'eventuale impatto delle dipendenze EU-EU contenute nella sequenza critica;

2. per ognuna delle dipendenze IU-EU, si valuta il ritardo Δ_2 sulla EU_Master, causato dalle dipendenze logiche EU-EU che
    a. *appartengono alla sequenza critica* della dipendenza IU-EU in questione,
    b. *contribuiscono a indurre tale dipendenza*.

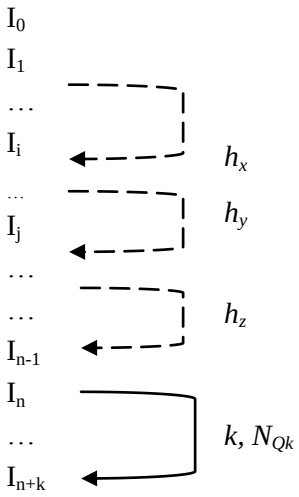
    Inoltre:

    c. nel caso che più istruzioni che contribuiscono a indurre la dipendenza IU-EU siano tra loro *indipendenti*, occorre considerare la *latenza che deriva dalla loro esecuzione in parallelo*;
3. avendo separato gli effetti delle dipendenze IU-EU ed EU-EU come ai punti 1 e 2, e tenendo conto delle latenze della EU in gioco, il termine Δ_1 può anche risultare negativo;
4. il ritardo complessivo è la somma algebrica dei due ritardi:

$$\Delta = \Delta_1 + \Delta_2$$

    in quanto le bolle introdotte nel funzionamento di EU_Master, con le caratteristiche del punto 2, contribuiscono ad aumentare ulteriormente, *e in maniera additiva*, la latenza delle istruzioni che inducono dipendenze logiche su IU.

In generale, consideriamo una sequenza critica di istruzioni I_0, I_1, …, I_n, con I_n che induce una dipendenza logica IU-EU sulla I_{n+k}:



Nel seguito ricaveremo il modello dei costi attraverso i seguenti passi:

i) *valutazione di* Δ_1

ii) *valutazione di* Δ_2

iii) *dipendenze logiche annidate e intrecciate*

i) Valutazione di Δ_1

Se la sequenza critica *non* contiene dipendenze logiche EU-EU, oppure esse *non contribuiscono* a indurre la dipendenza IU-EU, allora $\Delta_2 = 0$, quindi la valutazione di Δ_1 rappresenta la valutazione definitiva di Δ. Δ_1 tiene conto non solo di N_{Qk}, ma anche della *latenza di esecuzione dell'istruzione* I_n, quindi è espresso dalla seguente generalizzazione della formula usata nel cap. XI:

$$\Delta_1 = t \sum_{k=1}^{\bar{k}} d_k \ (N_{Q_k} + L_{pipe\ -k} + 1 - k)$$

dove con $L_{pipe\text{-}k}$ si indica la latenza di esecuzione *dell'istruzione* I_n, espressa in funzione del numero degli stadi di Unità Funzionali in pipeline; ad esempio $L_{pipe\text{-}k} = 4$ per un moltiplicatore in virgola fissa, come usato negli esempi della sezione precedente. Nel caso che tale istruzione sia *eseguita direttamente dalla sola EU_Master* (LOAD o aritmetica corta in virgola fissa):

$$L_{pipe\text{-}k} = 0$$

in quanto, per tali classi di istruzioni, della latenza di I_n si tiene già conto con il termine "1".

Il punto 3 del procedimento mette in evidenza che può essere $\Delta_1 < 0$, e che tale termine va comunque mantenuto nella somma algebrica con Δ_2. La cosa non ha un significato fisico, ma solo modellistico: la distanza della dipendenza logica può essere tale da annullare in parte il ritardo dovuto a Δ_2.

ii) **Valutazione di Δ_2**

Se la sequenza critica contiene dipendenze logiche EU-EU che contribuiscono a indurre la dipendenza IU-EU, allora deve essere valutato Δ_2. Ragionando analogamente a quanto fatto finora, si può scrivere:

$$\Delta_2 = t \sum_{h=1}^{\bar{h}} d_h \ (L_{pipe\ -h} + 1 - h)$$

dove i termini $L_{pipe\text{-}h}$ sono definiti e calcolati in maniera del tutto analoga agli $L_{pipe\text{-}k}$.

In tutti i casi in cui l'istruzione che induce la dipendenza logica EU-EU è eseguibile direttamente dalla EU_Master, e quindi nei casi in cui

$$L_{pipe\text{-}h} = 0$$

la dipendenza non ha alcun effetto. Per brevità di applicazione del modello, in questi casi porremo

$$d_h = 0$$

In particolare, si ha

$$\Delta_2 = 0$$

nel caso che la sequenza critica contenga *solo* istruzioni eseguite direttamente dalla EU_Master.

Si verifica così che, nel caso il programma contenga solo istruzioni aritmetiche corte in virgola fissa e LOAD, il modello dei costi fornisce la stessa valutazione del Cap. XI.

È importante ribadire le caratteristiche del punto 2 del procedimento, secondo la quale le dipendenze EU-EU da considerare

a. *appartengono alla sequenza critica* della dipendenza IU-EU in questione,
b. *contribuiscono a indurre tale dipendenza*.

Se una delle due caratteristiche *a), b)* non si verifica, allora le dipendenze EU-EU non hanno effetto sulla dipendenza IU-EU. Quindi, anche in questo caso porremo:

$$d_h = 0$$

Infatti istruzioni, facenti parte di una sequenza critica, che siano *indipendenti* da quelle che contribuiscono a indurre la dipendenza logica IU-EU, non hanno effetto sulla valutazione di Δ_2, in quanto la loro esecuzione è interamente sovrapposta all'esecuzione delle altre istruzioni.

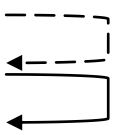
Quanto alla caratteristica

c. nel caso che più istruzioni che contribuiscono a indurre la dipendenza IU-EU siano tra loro *indipendenti*, occorre considerare la *latenza che deriva dalla loro esecuzione in parallelo*,

essa è una conseguenza della struttura parallelo-pipeline della EU.

Ad esempio nel codice:

1. MUL Ra, Rb, Rx
2. DIV Ra, Rb, Ry
3. ADD Rx, Ry, Rz
4. STORE …, …, Rz



l'istruzione 3 induce sulla 4 una dipendenza IU-EU (distanza $k = 1$, con probabilità $d_{k1} = 1/4$, $N_Q = 1$). Le istruzioni 1 e 2 appartengono alla sequenza critica di tale dipendenza (caratteristica *a*), contribuiscono a indurre la dipendenza stessa (caratteristica *b*) in quanto inducono una dipendenza EU-EU sulla 3 (distanza $h = 1$, probabilità $d_{h1} = 1/4$), e sono tra loro *indipendenti*, quindi eseguite in parallelo (caratteristica *c*). Si ha:

- per la valutazione di Δ_1: $L_{pipe\text{-}k} = 0$,
- per la valutazione di Δ_2: $L_{pipe\text{-}h} = 4$.

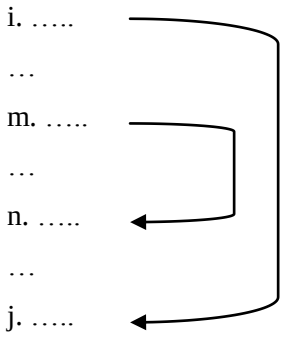
Il risultato è:

$$\Delta_1 = t/4 \qquad \Delta_2 = 4\ t/4 \qquad \Delta = 5\ t/4 \qquad T = 9\ t/4$$

Infine, nei casi in cui la valutazione di Δ_2 fornisca un risultato minore di zero, deve essere posto $\Delta_2 = 0$, in quanto le sequenza critica non avrebbe effetto sulla dipendenza logica IU-EU.

iii) Dipendenze logiche annidate e intrecciate

Consideriamo una sequenza di istruzioni con le seguenti dipendenze logiche *annidate*:



Mentre nel caso di EU con latenza unitaria la dipendenza esterna non aveva mai effetto, ora occorre valutare i ritardi $\Delta_{esterna}$ e $\Delta_{interna}$ delle due dipendenze in funzione delle latenze in gioco.

Il ritardo che caratterizza la sequenza è dato da:

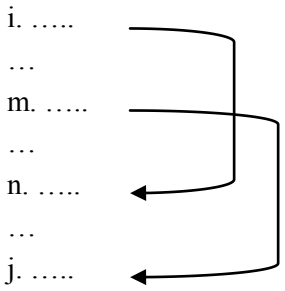
$$\Delta = max\ (\Delta_{esterna}, \Delta_{interna})$$

Inoltre, nella valutazione di $\Delta_{esterna}$ e $\Delta_{interna}$ *eventuali sequenze critiche a comune* (precedenti la i.) *sono considerate solo nella valutazione di* $\Delta_{esterna}$.

Infatti, sia *p t* la dimensione della bolla causata dalla dipendenza esterna, e la sequenza interna sia costituita da *q* istruzioni. Se le *q* istruzioni non contengono dipendenze logiche, del valore di *q* si è tenuto conto nella distanza per la valutazione della bolla esterna. Se la sequenza interna contiene dipendenze logiche, può verificarsi che

- la latenza della sequenza interna sia egualmente minore o uguale a *p t*, oppure
- pur con $q < p$, tali dipendenze siano tali da creare una bolla maggiore di *p t*.

Tutto quanto detto per il caso di annidamento vale anche le dipendenze logiche *intrecciate*:



Gli esempi di dipendenze logiche annidate e intrecciate sono riportati nella sezione 1.2.4.

1.3.3 Esempi

L'applicazione agli esempi della sezione 1.2.1 fornisce i seguenti risultati, identici a quelli ottenuti con il procedimento di simulazione:

Esempio 1:

- $k = 1$, $d_{k1} = 1/3$, $N_{Qk} = 2$, $L_{pipe\text{-}k} = 4 \rightarrow \Delta_1 = 6\ t/3$
- $d_{h1} = 0 \rightarrow \Delta_2 = 0$
- $\Delta = \Delta_1 + \Delta_2 = 6\ t/3$

Esempio 1bis:

- $k = 1$, $d_{k1} = 1/2$, $N_{Qk} = 1$, $L_{pipe} = 4 \rightarrow \Delta_1 = 5\ t/2$
- $d_{h1} = 0 \rightarrow \Delta_2 = 0$
- $\Delta = \Delta_1 + \Delta_2 = 5\ t/2$

Esempio 2:

- $k = 1$, $d_{k1} = 1/4$, $N_{Qk} = 2$, $L_{pipe\text{-}k} = 0 \rightarrow \Delta_1 = 2\ t/4$
- $h = 1$, $d_{h1} = 1/4$, $L_{pipe\text{-}h} = 4 \rightarrow \Delta_2 = t$
- $\Delta = \Delta_1 + \Delta_2 = 6\ t/4$

Esempio 2bis:

- $k = 1$, $d_{k1} = 1/3$, $N_{Qk} = 1$, $L_{pipe\text{-}k} = 0 \rightarrow \Delta_1 = t/3$
- $h = 1$, $d_{h1} = 1/3$, $L_{pipe\text{-}h} = 4 \rightarrow \Delta_2 = 4\ t/3$
- $\Delta = \Delta_1 + \Delta_2 = 5\ t/3$

Esempio 3:

- $k = 2$, $d_{k2} = 1/4$, $N_{Qk} = 2$, $L_{pipe\text{-}k} = 4 \rightarrow \Delta_1 = 5\ t/4$
- $d_{h1} = 0 \rightarrow \Delta_2 = 0$
- $\Delta = \Delta_1 + \Delta_2 = 5\ t/4$

Esempio 3bis:

- $k = 2$, $d_{k2} = 1/3$, $N_{Qk} = 1$, $L_{pipe\text{-}k} = 4 \rightarrow \Delta_1 = 4\ t/3$
- $d_{h1} = 0 \rightarrow \Delta_2 = 0$
- $\Delta = \Delta_1 + \Delta_2 = 4\ t/3$

Esempio 4:

- $k = 1$, $d_{k1} = 1/4$, $N_{Qk} = 1$, $L_{pipe\text{-}k} = 0 \rightarrow \Delta_1 = t/4$
- $h = 1$, $d_{h1} = 2/4$, $L_{pipe\text{-}h} = 4 \rightarrow \Delta_2 = 2\ t$
- $\Delta = \Delta_1 + \Delta_2 = 9\ t/4$

Esempio 4bis:

- $k = 2$, $d_{k2} = 1/4$, $N_{Qk} = 1$, $L_{pipe\text{-}k} = 4 \rightarrow \Delta_1 = 4\ t/4$
- $h = 1$, $d_{h1} = 1/4$, $L_{pipe\text{-}h} = 4 \rightarrow \Delta_2 = 4\ t/4$
- $\Delta = \Delta_1 + \Delta_2 = 8\ t/4$

Esempio 5:

- $k = 1$, $d_{k1} = 1/4$, $N_{Qk} = 1$, $L_{pipe\text{-}k} = 4 \rightarrow \Delta_1 = 5\ t/4$
- $h = 2$, $d_{h2} = 1/4$, $L_{pipe\text{-}h} = 4 \rightarrow \Delta_2 = 3\ t/4$
- $\Delta = \Delta_1 + \Delta_2 = 8\ t/4$

Esempio 5bis:

- $k = 1$, $d_{k1} = 1/4$, $N_{Qk} = 1$, $L_{pipe\text{-}k} = 4 \rightarrow \Delta_1 = 5\ t/4$
- $h = 2$, $d_{h2} = 1/4$, $L_{pipe\text{-}h} = 4 \rightarrow \Delta_2 = 3\ t/4$
- $\Delta = \Delta_1 + \Delta_2 = 8\ t/4$

Come ulteriori esempi si considerino i seguenti;

Esempio 6:

una sequenza critica di 7 istruzioni tutte con $L_{pipe\text{-}h} = 4$, tutte indipendenti, e la settima che induce una dipendenza IU-EU di distanza 1 con $N_Q = 1$. Si ha: $\Delta_1 = 5\ t/7$, $\Delta_2 = 0$.

Esempio 7:

lo stesso risultato dell'Esempio 6 si otterrebbe anche se la prima istruzione inducesse una dipendenza EU-EU sulla sesta.

Esempio 8:

una sequenza critica di 9 istruzioni tutte con $L_{pipe\text{-}h} = 4$, tutte indipendenti oppure con la prima che induce una dipendenza EU-EU sulla nona, e la nona che induce una dipendenza IU-EU di distanza 1 con $N_Q = 1$. Si ha: $\Delta_1 = 5\ t/10$, $\Delta_2 = 0$.

Esempio 9:

consideriamo il loop più interno di un programma per calcolare il prodotto matrice-vettore, senza introdurre ottimizzazioni eccetto l'applicazione della tecnica del delayed branch all'istruzione di salto:

```
1.          LOAD   RA, Rj, Ra
2. LOOPj:   LOAD   RB, Rj, Rb
3.          MUL    Ra, Rb, Rx
4.          ADD    Rc, Rx, Rc
5.          INCR   Rj
6.          IF <   Rj, RM, LOOPj, delayed_branch
7.          LOAD   RA, Rj, Ra
```

La dipendenza IU-EU di distanza *k =1*, con $N_Q = 2$, indotta dalla INCR sulla IF, è preceduta da una sequenza critica che contiene la sottosequenza di istruzioni tra loro dipendenti MUL-ADD, ma queste istruzioni sono indipendenti dalla INCR e quindi *non* contribuiscono a indurre la dipendenza IU-EU. Si ha $\Delta_2 = 0$, e quindi $\Delta = \Delta_1$. Tenendo conto che $L_{pipe\text{-}k} = 0$ per l'istruzione INCR che induce la dipendenza IU-EU, si ottiene lo steso risultato che avremmo avuto se anche la MUL avesse avuto latenza unitaria, come avveniva nel Cap. XI:

$$\Delta = 2\ t/6 \qquad\qquad T = 8\ t/6$$

Il risultato è dunque significativo dal punto di vista del rapporto prestazioni/costo.

La latenza della sequenza MUL-ADD (con latenza della moltiplicazione intera uguale a *4t*) è completamente mascherata dalle rimanenti istruzioni. Esaminando la simulazione grafica:

IM	1	2	3	4	5	6	7			2	3	4	
IU		1	2	3	4	5			6	7	2	3	
DM			1	2							7	2	
EU-Mast				1	2	3		5			4	7	2
INT Mul							3	3	3	3			

si nota come la EU_Master, dopo aver inoltrato l'esecuzione della MUL all'Unità Funzionale in pipeline, rilevi che l'istruzione ADD non è abilitata (deve attendere il completamento della MUL); invece di bloccarsi, EU_Master *rileva che la prima istruzione in coda (la INCR) è eseguibile e la esegue*. Solo a questo punto attende che si sblocchi la ADD, non avendo altro lavoro da fare in base al funzionamento in-order.

Nota sul riuso dei registri

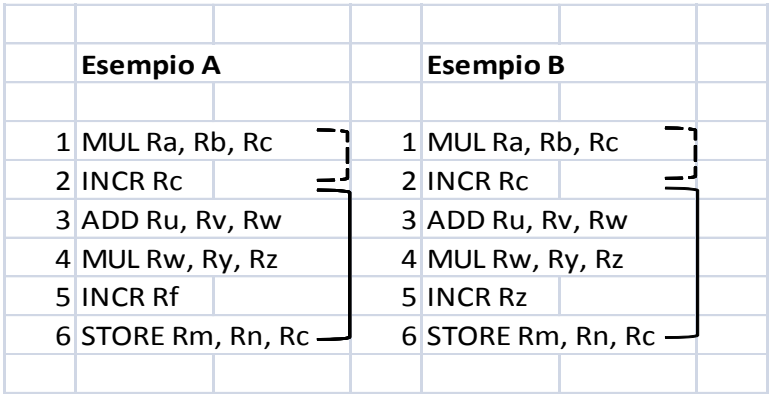
Nella compilazione del programma precedente non si è applicato, o si è limitato al massimo, il riuso di registri per valori semanticamente distinti.

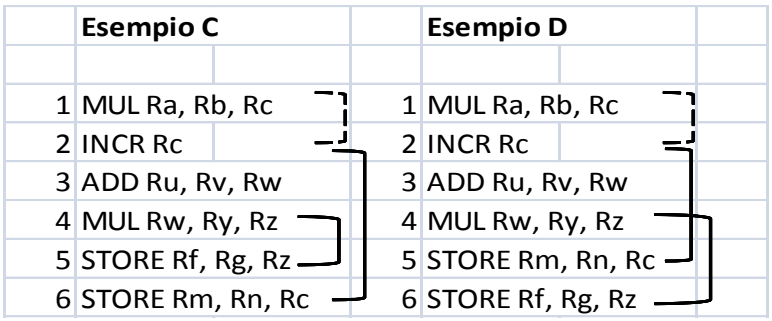
Ad esempio nella MUL Ra, Rb, Rx, potevamo anche riusare Ra al posto di Rx. In generale, il riuso di registri può portare a *ulteriori dipendenze sui dati* che degradano sensibilmente le prestazioni e complicano il modello dei costi. Il problema verrà discusso nella sezione 2.

Nel seguito ci atterremo alla regola di *non applicare riuso dei registri*, nei limiti della loro disponibilità.

1.3.4 Esempi di dipendenze logiche annidate e intrecciate

Si considerino i quattro esempi seguenti:





Il *caso A* mostra una situazione di *non* annidamento: le istruzioni 3, 4, 5 hanno solo effetto sulla distanza della dipendenza IU-EU indotta dalla 2 sulla 6. Applicando il modello dei costi:

$\Delta_1 = -2$ t/6 $\quad \Delta_2 = 4$ t/6 $\quad \Delta = \Delta_1 + \Delta_2 = 2$ t/6 $\quad T = 8$ t /6

Il *caso B* è analogo al caso A: la dipendenza indotta dalla 4 sulla 5 non dà luogo ad una situazione di annidamento in quanto si tratta di una dipendenza EU-EU. Quindi:

$\Delta_1 = -2$ t/6 $\quad \Delta_2 = 4$ t/6 $\quad \Delta = \Delta_1 + \Delta_2 = 2$ t/6 $\quad T = 8$ t /6

Il *caso C* mostra una situazione di reale annidamento, in quanto anche la dipendenza interna, indotta dalla 4 sulla 5, è di tipo IU-EU. Applicando il modello dei costi:

- per la dipendenza esterna: $\Delta_1 = -2$ t/6 $\quad \Delta_2 = 4$ t/6 $\quad \Delta_{esterna} = 2$ t/6
- per la dipendenza interna: $\Delta_1 = 5$ t/6 $\quad \Delta_2 = 0$ $\quad \Delta_{interna} = 5$ t/6
- l'effetto complessivo è: $\Delta = \max(\Delta_{esterna}, \Delta_{interna}) = \Delta_{interna} = 5$ t/6 $\quad T = 11$ t/6

Il *caso D* è una ottimizzazione statica del caso C, contenente una dipendenza intrecciata vera. La valutazione si effettua come nel caso C, ottenendo:

- per la dipendenza esterna: $\Delta_1 = -1$ t/6 $\quad \Delta_2 = 4$ t/6 $\quad \Delta_{esterna} = 3$ t/6
- per la dipendenza interna: $\Delta_1 = 4$ t/6 $\quad \Delta_2 = 0$ $\quad \Delta_{interna} = 4$ t/6
- l'effetto complessivo è: $\Delta = \max(\Delta_{esterna}, \Delta_{interna}) = \Delta_{interna} = 4$ t/6 $\quad T = 10$ t/6

In tutti i casi si verifica perfetta concordanza con la simulazione grafica.

Esempio 10:

Consideriamo ancora il programma dell'Esempio 9 (loop interno del prodotto matrice-vettore sez. 1.2.3) e apportiamo ulteriori ottimizzazioni statiche. Allo scopo di eliminare l'effetto della dipendenza logica IU-EU indotta dalla INCR sulla IF, anticipiamo la INCR prima della MUL:

```
1.          LOAD   RA, Rj, Ra
2. LOOPj:   LOAD   RB, Rj, Rb
3.          INCR   Rj
4.          MUL    Ra, Rb, Rx
5.          ADD    Rc, Rx, Rc
6.          IF <   Rj, RM, LOOPj, delayed_branch
7.          LOAD   RA, Rj, Ra
```

La dipendenza IU-EU, indotta dalla 3 sulla 6, ha $k = 3$ con $N_Q = 2$ e $L_{pipe\text{-}k} = 0$, e la sequenza critica non contiene istruzioni lunghe che contribuiscono a indurre la dipendenza stessa, quindi:

$$\Delta_1 = 0 \qquad \Delta_2 = 0 \qquad \Delta = 0 \qquad T = t$$

come nel caso di MUL con latenza unitaria:

IM	1	2	3	4	5	6	7	2	3	4		
IU		1	2	3	4	5	6	7	2	3		
DM			1	2					7	2		
EU-Mast				1	2	3	4			7	2	5
INT Mul								4	4	4	4	

Le istruzioni MUL e ADD non danno luogo a dipendenze annidate.

2. Ottimizzazioni statiche e ottimizzazioni dinamiche

2.1 Unità Istruzioni in-order *vs* out-of-order

2.1.1 Esempio

Analizziamo la computazione su array di interi:

$$\forall\ i = 0\ ..\ N\text{–}1 :\quad C[i] = A[i] * B[i] + A[i] / B[i]$$

Il codice con la sola ottimizzazione delayed branch è:

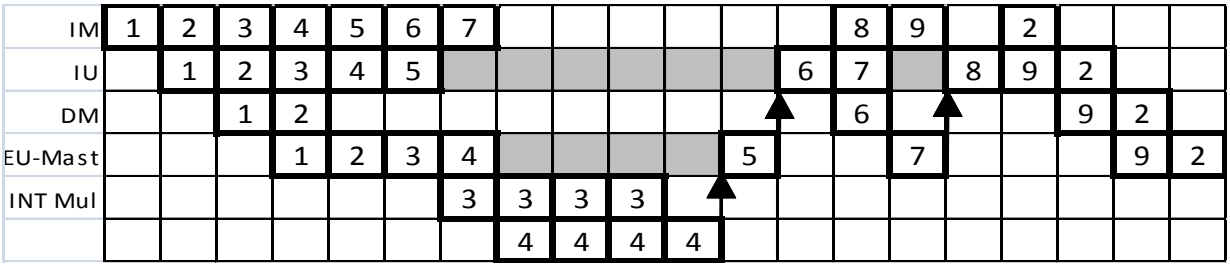
```
1.          LOAD   RA, Ri, Ra
2. LOOP:    LOAD   RB, Ri, Rb
3.          MUL    Ra, Rb, Rx
4.          DIV    Ra, Rb, Ry
5.          ADD    Rx, Ry, Rc
6.          STORE  RC, Ri, Rc
7.          INCR   Ri
8.          IF <   Ri, RN, LOOP, delayed_branch
9.          LOAD   RA, Ri, Ra
```

L'analisi rileva:

1. una dipendenza IU-EU di distanza $k = 1$, probabilità 1/8, $N_Q = 1$, indotta dalla INCR sulla IF, con $L_{pipe\text{-}k} = 0$ e sequenza critica contenente la sola istruzione INCR che induce la dipendenza;
2. una dipendenza IU-EU di distanza $k = 1$, probabilità 1/8, $N_Q = 2$, indotta dalla ADD sulla STORE, con $L_{pipe\text{-}k} = 0$ e sequenza critica costituita da LOAD RB, MUL, DIV, e la ADD stessa. Questa sequenza contiene una dipendenza EU-EU di distanza $h = 1$, con probabilità 1/8, indotta dalla DIV sulla ADD, quindi *contribuisce* a indurre la dipendenza IU-EU. Poiché MUL e DIV sono indipendenti, $L_{pipe\text{-}h} = 4$.

Quindi,	per la dipendenza 1:	$\Delta_1 = t/8$	$\Delta_2 = 0$
	per la dipendenza 2:	$\Delta_1 = 2\ t/8$	$\Delta_2 = 4\ t/8$
	complessivamente:	$\Delta = 7\ t/8$	$T = 15\ t/8$

come si può verificare anche con la simulazione grafica:



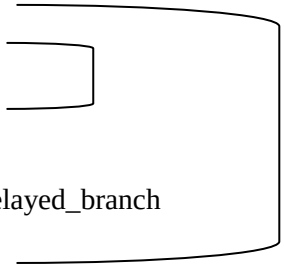
Si noti il *funzionamento in-order della IU*. Con un funzionamento out-of-order di IU, l'istruzione 7 avrebbe anche potuto essere preparata prima della 6, e addirittura eseguita dalla EU. Questo avrebbe

portato ad anticipare la risoluzione della dipendenza della stessa 7 sulla 8, con un significativo guadagno sul tempo di servizio.

Vediamo cosa è possibile fare con sole *ottimizzazioni statiche, conservando la IU in-order*.

Consideriamo il codice ottimizzato anche per le dipendenze logiche, spostando le due istruzioni di LOAD dopo la INCR ed applicando il delayed branch con lo spostamento della STORE:

```
1.          LOAD   RA, Ri, Ra
2.          LOAD   RB, Ri, Rb
3. LOOP:    MUL    Ra, Rb, Rx
4.          DIV    Ra, Rb, Ry
5.          ADD    Rx, Ry, Rc
6.          INCR   Ri
7.          LOAD   RA, Ri, Ra
8.          LOAD   RB, Ri, Rb
9.          IF <   Ri, RN, LOOP, delayed_branch
10.         STORE  RC, Ri, Rc
```



Esistono due dipendenze logiche IU-EU annidate, della quali

- quella esterna (indotta dalla 5 sulla 10, distanza $k = 5$, probabilità $d_k = 1/8$, $N_Q = 2$, $L_{pipe\text{-}k} = 0$, $h = 1$, probabilità $d_h = 1/8$, $L_{pipe\text{-}h} = 4$) è caratterizzata da Δ_1 negativo:

  $\Delta_{1\text{-esterna}} = -2$ t/8 $\qquad$ $\Delta_{2\text{-esterna}} = 4$ t/8 $\qquad$ $\Delta_{\text{esterna}} = 2$ t/8

- quella interna (indotta dalla 6 sulla 7, distanza $k = 1$, probabilità $d_k = 1/8$, $N_Q = 2$, $L_{pipe\text{-}k} = 0$, probabilità $d_h = 0$):

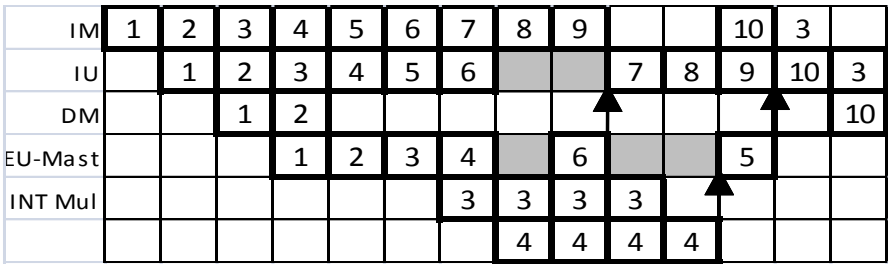
  $\Delta_{1\text{-interna}} = 2$ t/8 $\qquad$ $\Delta_{2\text{-interna}} = 0$ $\qquad$ $\Delta_{\text{interna}} = 2$ t/8

  (secondo la regola, la sequenza critica a comune è stata valutata per la dipendenza esterna).

Quindi:

$$\Delta = 2 \text{ t/8} \qquad T = 10 \text{ t/8}$$

come si verifica anche con la simulazione grafica:

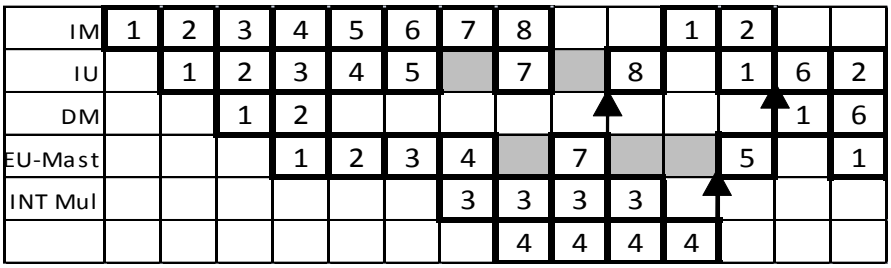


Si osservi come, alla luce dell'annidamento delle dipendenze logiche e del ruolo che giocano le latenze in una EU parallela, non ci sia stato bisogno di minimizzare ulteriormente la dipendenza interna.

Consideriamo lo stesso programma compilato *senza ottimizzazioni statiche*:

```
1. LOOP:   LOAD   RA, Ri, Ra
2.         LOAD   RB, Ri, Rb
3.         MUL    Ra, Rb, Rx
4.         DIV    Ra, Rb, Ry
5.         ADD    Rx, Ry, Rc
6.         STORE  RC, Ri, Rc
7.         INCR   Ri
8.         IF <   Ri, RN, LOOP
```

Esaminiamo il funzionamento con una IU *out-of-order*:



Una volta rilevato che l'istruzione 6 è bloccata (in attesa che si completi l'esecuzione della 5), IU può esaminare le istruzioni successive, man mano che si presentano. L'istruzione 7 (INCR Ri) è aritmetica, quindi può essere inoltrata alla EU.

Si noti che l'istruzione 6, bloccata (STORE RC, Ri, Rc) in attesa dell'aggiornamento di RG[Rc], ha nel suo dominio il registro RG[Ri]. Questo potrebbe far credere di essere in presenza di una possibile situazione di inconsistenza inoltrando INCR Ri in esecuzione, ma non è così: *quando IU prepara la STORE, RG[Ri] è aggiornato, quindi IU può calcolare l'indirizzo RG[RC] + RG[Ri] in maniera consistente* (oppure avrebbe potuto memorizzare RG[Ri] in un registro temporaneo).

Da quando registra che l'istruzione 6 è bloccata, IU controlla continuamente (nondeterministicamente ad ogni ciclo di clock) l'eventuale aggiornamento di RG[Ri]. Nel frattempo, inoltra correttamente la 7, e successivamente esegue anche la 8 che, una volta sbloccata dalla 7, provoca una bolla in seguito a salto. La stessa 1 è preparata e inoltrata prima della 6.

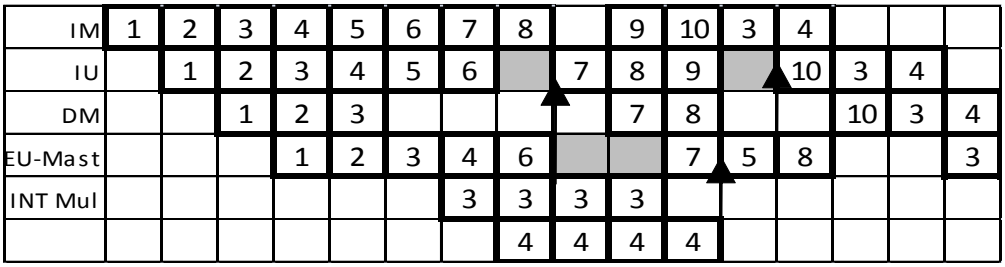
Come si vede:

$$\Delta = 2\ t/8 \qquad\qquad T = 10\ t/8$$

che rappresenta *lo stesso risultato ottenuto con sole ottimizzazioni statiche e IU in-order*. In effetti, l'evoluzione delle istruzioni è simile nei due casi, anche se non identica.

La complessità di una tale IU out-of-order aumenta, anche non in modo eccessivo in quanto IU continua ad analizzare le istruzioni nello stesso ordine con cui sono inviate da IM.

Se consideriamo la versione con ottimizzazioni statiche, con una IU out-of-order non si ottiene alcun ulteriore miglioramento (*T = 10 t/8*):



In questo esempio, le ottimizzazioni dinamiche si rivelano ugualmente efficaci di quelle statiche.

Il risultato non è generalizzabile in forma rigorosa o, almeno, *non è dimostrabile in generale*. Esistono programmi in cui si possono avere differenze in prestazioni tra le due implementazioni, oppure talvolta le ottimizzazioni statiche hanno effetto anche in architetture con ottimizzazioni dinamiche. Inoltre, un confronto rigoroso dovrebbe essere basato su un ampio insieme di ipotesi sia architetturali che relative alle ottimizzazioni statiche.

L'esempio è significativo in quanto:

a) pur non essendo generalizzabile la possibile perfetta equivalenza in prestazioni, si può comunque affermare che le due implementazioni – una con IU out-of-order e ottimizzazioni dinamiche, l'altra con IU in-order e ottimizzazioni statiche – hanno prestazioni del tutto paragonabili;

b) ci si rende conto del fatto che, specialmente quando si intenda avere compatibilità binaria, la scelta delle ottimizzazioni dinamiche e del funzionamento out-of-order ha una sua giustificazione.

2.1.2 Modello dei costi per IU out-of-order

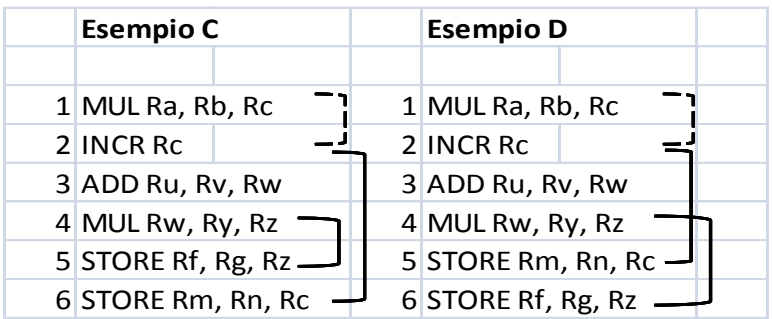
Il modello dei costi studiato per architetture con IU in-order produce risultati attendibili anche per architetture con IU out-of-order.

Infatti, il caso in cui la IU può modificare l'ordine delle istruzioni, in presenza di istruzioni bloccate, corrisponde a quello in cui, con ottimizzazioni statiche e IU in-order, si utilizza spostamento di codice per aumentare la distanza di dipendenza logica.

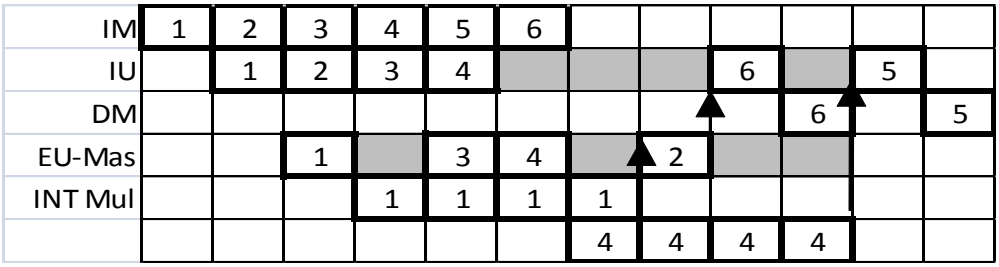
In altri termini, il modello dei costi viene applicato rigorosamente ad architetture con IU in-order.

Per ottenere una valutazione attendibile per l'architettura corrispondente out-of-order, viene valutata la computazione equivalente con le migliori ottimizzazioni statiche possibile, eventualmente utilizzando *loop-unfolding* per evidenziare le relazioni di dipendenza tra istruzioni appartenenti a iterazioni diverse di programmi iterativi.

Come ulteriore applicazione, si considerino gli esempi C e D utilizzati nella sez. 1.2.4 per le dipendenze annidate e intrecciate.



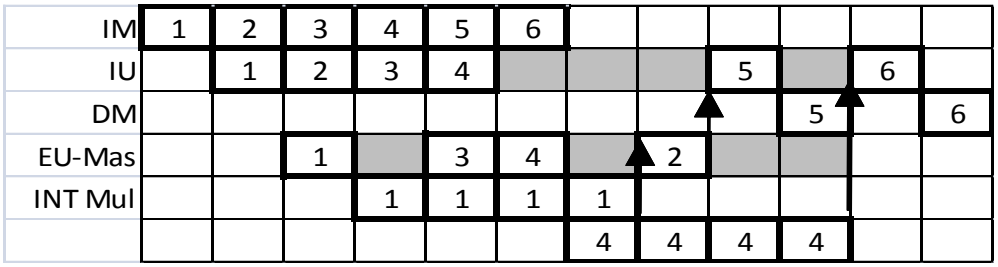
Eseguendo il programma C su una architettura con IU out-of-order e sole ottimizzazioni dinamiche



si ottengono le stesse prestazioni del caso D con IU in-order e sole ottimizzazioni statiche:

$$\Delta = 4\ t/6 \qquad T = 10\ t/6$$

In effetti, la simulazione grafica del caso in-order con ottimizzazioni statiche mostra la completa equivalenza delle due soluzioni (si consideri che le istruzioni 5 e 6 sono scambiate nei due casi):



Ribadiamo quindi gli importanti risultati finora discussi:

1. *la sostanziale equivalenza tra ottimizzazioni dinamiche con architettura out-of-order e ottimizzazioni statiche con architettura in-order,*
2. *l'applicabilità del modello dei costi ad architetture out-of-order, passando attraverso la (una) rappresentazione equivalente con ottimizzazioni statiche per l'architettura in-order.*

2.2 Mascheramento delle latenze: gerarchie di memoria

Come sappiamo, l'obiettivo principale delle ottimizzazioni è quello di mascherare il più possibile le latenze che degradano il tempo di servizio. Lo studio finora condotto si è concentrato sulle latenze delle operazioni aritmetiche in pipeline che hanno influenza sulle dipendenze logiche.

Un altro caso significativo è quello delle latenze per accessi a memorie esterne al chip, tipicamente nel caso di fault di cache e conseguenti trasferimenti di blocchi.

A livello architetturale, la situazione dei fault di cache è ancora gestibile efficientemente nel caso che il blocco risieda nella cache secondaria on-chip. Riprendendo l'analisi del Cap. VIII, sez. 3.3 (e integrazione presenta nell'Errata-Corrige), il trasferimento di un blocco da cache secondaria a cache primaria avviene in pipeline persino in una architettura seriale. In un architettura ILP, il pipeline può essere "prolungato" nella stessa fase di esecuzione (LOAD o STORE) senza degradazioni apprezzabili. Basta avere l'accortezza che le parole del blocco siano trasferite circolarmente a partire da quella su cui è stato generato il fault.

Inoltre, si può pensare che la cache dati abbia un funzionamento out-of-order: in caso di fault, la cache dati continua a servire, in parallelo al trasferimento del blocco, richieste di lettura o scrittura senza bloccarsi finché non riceve richieste per il blocco stesso in fase di trasferimento o incontra dipendenze sui dati.

La degradazione significativa di prestazioni si ha quando il blocco debba essere trasferito da memoria esterna (memoria principale o cache terziaria) in cache secondaria. Come confermato da benchmark condotti su macchine esistenti, raramente, in questi casi, il funzionamento out-of-order può aiutare a migliorare le prestazioni in maniera significativa, a fronte di un notevole aumento di complessità e di area.

Facciamo notare che, in una architettura basata su out-of-ordering, è possibile che richieste a memoria esterna possano essere generate fuori ordine, ma, come detto sopra, si tratta di richieste *indipendenti*, in quanto il funzionamento di DM assicura che vengano rispettate le dipendenze sui dati, ad esempio in una situazione di lettura-scrittura della stessa locazione.

Infine, se la sincronizzazione è applicata all'intero blocco, invece che alla singola locazione, può verificarsi la così detta situazione di *false sharing* (accessi in lettura/scrittura a locazioni distinte del blocco), che non necessariamente comporta una vera dipendenza. Quando possibile, le situazioni di false sharing si eliminano con accorgimenti di "padding" (lasciare alcune locazioni del blocco "vuote" in modo da utilizzare due blocchi distinti).

Le ottimizzazioni ottenibili staticamente attraverso le opportune annotazioni alle istruzioni, studiate Cap. VIII (riuso, prefetching, gestione delle scritture), continuano a rappresentare la soluzione più efficace in termini di prestazioni, a patto che il set di istruzioni sia predisposto. In assenza di tali facilitazioni a livello di macchina assembler, l'emulazione delle stesse ottimizzazioni a firmware, ad esempio da parte dell'unità cache, è molto più problematica e dispendiosa; ad esempio, scoprire occasioni di riuso di dati attraverso la *storia* degli accessi in memoria.

2.3 Dipendenze sui dati: un'analisi più approfondita

Come visto negli esempi della sez. 1, altre ottimizzazioni dinamiche si possono ottenere con funzionamento *out-of-order* dell'Unità Esecutiva, andando a scoprire istruzioni abilitate attraverso una analisi associativa della coda istruzioni. Anche in questo caso, un opportuno ordinamento statico delle istruzioni stesse e una allocazione opportuna dei registri può permettere di ottenere prestazioni paragonabili anche con ottimizzazioni statiche e funzionamento della EU *in-order*.

Per approfondire questo aspetto, ed in generale il tema delle dipendenze sui dati, riprendiamo lo studio delle **condizioni di Bernstein** (Cap. IV, sez. 2.2.2).

Date due operazioni presenti in una computazione sequenziale (a qualunque livello):

$$F_1\,(D_1) \rightarrow R_1 \;;\; F_2\,(D_2) \rightarrow R_2$$

con D_i dominio e R_i codominio dell'operazione F_i, esse sono *indipendenti*, e possono essere eseguite in un qualunque ordine o contemporaneamente, se valgono tutte le seguenti condizioni:

$$\begin{cases} R_1 \cap D_2 = \varnothing & \textit{vera dipendenza} \\ R_1 \cap R_2 = \varnothing & \textit{dipendenza di uscita} \\ R_2 \cap D_1 = \varnothing & \textit{antidipendenza} \end{cases}$$

Quando almeno una delle tre condizioni non sia verificata, allora le operazioni sono *dipendenti*.

Accanto alle tre condizioni è riportata una terminologia spesso adottata per caratterizzarle. La terminologia è significativa in quanto esprime subito il concetto base: la *vera* dipendenza è quella del primo tipo (detta anche *Read-After-Write*). Le altre due (rispettivamente *Write-After-Writ*e e *Write-After-Read*) sono in realtà dipendenze *artificiali*, in quanto

a) possono non avere effetto sulla correttezza di funzionamento in alcuni modelli di esecuzione,

b) possono essere sempre eliminate con una opportuna allocazione delle variabili, oppure esprimendo la computazione con un formalismo, o almeno secondo uno stile, puramente funzionale.

Il caso *a)* è stato studiato a fondo a livello firmware per le antidipendenze (Cap. IV): *il modello di funzionamento sincrono permette di eseguire utilmente in parallelo nella stessa microistruzione due operazioni legate da antidipendenza*. Le dipendenze di uscita possono sempre essere evitate con una scelta opportuna dei registri e della loro inizializzazione (caso *b*).

D'altra parte, nel modello data-flow l'ordinamento delle operazioni è espresso esclusivamente in termini delle dipendenze vere, trattandosi di un modello di programmazione funzionale (non Von Neumann) nel quale è assente il concetto di variabile, di assegnamento e di sequenza di istruzioni.

Un esempio interessante di antidipendenza, che non produce effetto, è stato incontrato nella sezione 2.1.1. a proposito della IU out-of-order. Nell'esempio di sequenza

```
6.   STORE   RC, Ri, Rc
7.   INCR    Ri
```

l'istruzione 7 può essere inoltrata in esecuzione prima del completamento della 6 in IU, in quanto IU utilizza comunque la copia corretta di RG[Ri] contemporaneamente alla sua modifica nella EU da parte della 7. Si tratta di un caso concettualmente analogo alla parallelizzazione di antidipendenze a livello di microistruzioni.

Nell'architettura ILP, oltre a questa "parallelizzazione naturale" delle antidipendenze, tanto le dipendenze di uscita quanto altre antidipendenze possono essere evitate con una opportuna **allocazione dei registri**. Si consideri di nuovo l'Esempio 9 della sez. 1:

```
1.          LOAD   RA, Rj, Ra
2. LOOPj:   LOAD   RB, Rj, Rb
3.          MUL    Ra, Rb, Rx
4.          ADD    Rc, Rx, Rc
5.          INCR   Rj
6.          IF <   Rj, RM, LOOPj, delayed_branch
7.          LOAD   RA, Rj, Ra
```

Se al posto dei registri Rx e Rc avessimo accumulato i risultati intermedi nello stesso registro Ra, nessuna conseguenza si sarebbe avuta in una esecuzione sequenziale, ma non sarebbe possibile esplicitare il massimo parallelismo con forme (pur controllate) di out-of-ordering: ad esempio, verrebbe a crearsi una *inutile antidipendenza* tra la MUL e le LOAD che impedirebbe di eseguire una LOAD dell'iterazione *i+1* in parallelo all'esecuzione della MUL dell'iterazione *i* (parallelismo che è facilmente evidenziabile con la rappresentazione loop-unfolding).

Per ottimizzare queste situazioni, *l'allocazione statica* dei registri (generali e in virgola mobile) è praticabile, ed è efficiente, nelle macchine con un buon numero di registri (come nelle macchine Risc), a condizione che si possa (ri-)compilare il programma per l'architettura a disposizione.

Nelle macchine in cui i registri siano in numero limitatissimo (come le macchine Cisc basate su x-86), o quando si voglia riusare codice binario nel quale venga fatto un riuso eccessivo degli stessi registri, allora non resta che la strada dell'*allocazione dinamica dei registri*.

La tecnica adottata prende anche il nome di **Register Renaming**: si tratta di *allocare dinamicamente i registri visibili a livello assembler in registri distinti a livello firmware* e presenti in numero superiore.

Ad esempio, la macchina D-RISC potrebbe disporre, oltre che dei 64 registri generali, di ulteriori 256 registri firmware. In linea di principio, a tempo di esecuzione un registro-assembler viene "mappato" in un registro-firmware a partire dalle prima volta che l'istruzione viene decodificata, e fatto uso di una tabella di corrispondenza (*mapping table*) ogni volta che il registro viene riferito. Come variante, invece di usare i registri-firmware come un array, alcune macchine li usano come un buffer con allocazione in ordine FIFO e con accesso associativo (*buffer di riordino*).

Concettualmente, la tecnica del Register Renaming può essere implementata nella EU_Master.

Essa rappresenta una delle principali fonti di aumento della complessità e dell'area della CPU, specialmente nelle architetture superscalari.

2.4 Trattamento di interruzioni ed eccezioni

Come sappiamo, i segnali di interruzione giungono alla IU che quindi provvede alla fase firmware del trattamento interruzione.

In un funzionamento pipeline, ricevendo l'interruzione durante una certa istruzione, altre istruzioni precedenti sono ancora in una fase dell'elaborazione e non concluse. Poiché la fase assembler (handler) è una procedura chiamata dall'istruzione interrotta, essa rappresenta una sequenza di istruzioni logicamente successiva alle istruzioni attualmente in corso.

Una situazione del tutto analoga si verifica in presenza di eccezioni, ad esempio generate dalle MMU dei sottosistemi IM e DM (fault di pagina, violazioni di protezione, guasti hardware), oppure da EU (traboccamento, sottotraboccamento, guasti hardware).

Il trattamento di interruzioni/eccezioni non comporterebbe alcun problema in un funzionamento di IU in-order : lo stato della macchina (registri generali e in virgola mobile) è sempre quello consistente con la semantica del programma, quindi è quello logicamente aggiornato dalle istruzioni precedenti. In altri termini, la procedura handler dell'interruzione verrebbe eseguita nell'ordine del programma, e quindi opererebbe o su valori dei registri temporanei propri o su valori consistenti dei registri modificati dalle istruzioni precedenti.

Un problema implementativo nasce quando il funzionamento di IU sia *out-of-order*, e deriva proprio dallo stabilire un punto della computazione che sia caratterizzato da uno stato consistente, cioè caratterizzato dai contenuti che i registri avrebbero avuto al termine dell'esecuzione sequenziale o in-order della sequenza precedente la chiamata dello handler. Ciò si può ottenere principalmente in due modi:

- con la tecnica del *checkpointing*, cioè salvando periodicamente lo stato della CPU in opportuni registri buffer (ad esempio, il buffer di riordino), e ripartendo da tale stato, oppure
- usando *copie alternative dei registri*, usate come versioni temporanee, che verranno rese definitive quando richiesto, ad esempio durante la fase firmware del trattamento.

Ancora una volta, il costo dovuto ad un eccessivo utilizzo della tecnica out-of-ordering può superare di gran lunga i vantaggi prestazionali.

2.5 Predizione del salto

Si consideri una *istruzione di salto condizionato sulla quale sia indotta una dipendenza logica* riguardante i registri sul contenuto dei quali deve essere valutato il predicato. Si tratta dell'unico caso in cui è possibile "provare" a proseguire l'elaborazione *dell'istruzione stessa* senza attendere che la dipendenza logica si sia risolta, ad esempio assumendo che la condizione sia falsa e proseguendo sul ramo che inizia dall'istruzione successiva (*branch prediction*). Quando la dipendenza logica si sarà risolta, avrà luogo una fase di *commit*: i risultati ottenuti "sotto condizione" saranno resi definitivi se la predizione del predicato si rivelerà corretta, altrimenti andranno annullati e l'elaborazione dovrà riprendere dal ramo corrispondente al valore del predicato.

Il supporto architetturale necessario è ancora quello visto nella sezione precedente: usare *copie addizionali dei registri* per salvare lo stato all'inizio della computazione "sotto condizione", oppure per i risultati temporanei del ramo stesso eseguito "sotto condizione".

In alcune macchine che adottano la predizione del salto, la predizione stessa viene effettuata con tecniche di tipo statistico implementate a firmware, ad esempio basate sulla *storia* passata dei valori assunti dal predicato. Ovviamente, questo contribuisce ad aumentare sensibilmente la complessità dell'architettura e l'area occupata.

È anche possibile la così detta *esecuzione speculativa*: eseguire entrambi i rami (tutti i rami) contemporaneamente su copie indipendenti dei registri e, al risolversi della dipendenza logica, effettuare il *commit* della copia corretta nei registri che mantengono lo stato consistente del programma. Questo funzionamento ha senso solo in architetture superscalari.

Ovviamente, la predizione del salto è efficace solo in programmi contenenti lunghe sequenze di operazioni aritmetiche complesse che inducono la dipendenza, e quando la probabilità di salto sia significativamente alta.

Infine, il *modello dei costi* continua ad essere valido anche in presenza di predizione del salto, passando attraverso una rappresentazione equivalente con ottimizzazioni statiche: si tratta di valutare le prestazioni di entrambi i rami alternativi e, in funzione della probabilità che il predicato sia vero, valutare la media tra le prestazioni in caso di predizione corretta (esecuzione di un solo ramo) e in caso di predizione errata (esecuzione di entrambi i rami sequenzialmente).

3. Architettura superscalare

3.1 Una prima visione

L'idea dell'architettura superscalare è fondamentalmente quella di aumentare la banda di tutte le unità, in termini di istruzioni elaborate ogni $t = 2\tau$. Aumentando la banda di un fattore n, il tempo di servizio ideale diminuisce del fattore n (la performance ideale aumenta del fattore n):

$$T_{id} = \frac{2\tau}{n} \qquad \wp_{id} = \frac{n}{2\tau}$$

A questo scopo, *ogni elemento dello stream di istruzioni contiene n istruzioni*.

L'architettura che così si ottiene viene detta **superscalare a *n* vie** (*n-way superscalar*).

Valori tipici vanno da $n = 2$ (caso base e una delle attuali tendenze per multicore ad alto parallelismo) fino a 4 (valore abbastanza frequente) o 8 (valore per la CPU-server tra le più spinte), anche se esistono versioni con valore superiore (12, 16).

Chiameremo **scalare** l'architettura pipeline vista finora ($n = 1$).

3.1.1 Caratteristiche architetturali

Come detto, ogni elemento dello stream generato dalla memoria istruzioni (IM) contiene n istruzioni.

Nessun problema si ha nel caso base $n = 2$, in quanto ci sono due cicli di clock a disposizione della IM in pipeline per leggere due parole consecutive dalla cache istruzioni.

Per n più elevato, l'ulteriore aumento di banda si può ottenere con una cache istruzioni *interallacciata* con n moduli, ogni modulo della quale ha locazioni ampie una parola, oppure con una cache monolitica in cui ogni locazione contiene n parole, detta memoria *a parola lunga*.

La prima soluzione (cache istruzioni interallacciata) è più generale, in quanto permette di leggere e trasmettere le n istruzioni in pipeline, utilizzando collegamenti tra IM e IU ampi meno di n parole. In alcune architetture (simultaneous multithreading, sez. 4) questa soluzione permette anche di comporre dinamicamente istruzioni in uno stesso elemento dello stream. A causa dei ritardi di sincronizzazione (ricordiamo che la latenza di comunicazione non può scendere sotto $L_{com} = 2\tau$), la soluzione interallacciata con collegamenti ampi meno di n parole introduce comunque un certo overhead dovuto al transitorio del pipeline. La soluzione a parola lunga non introduce ritardi addizionali, ma ovviamente è più dispendiosa in termini di ampiezza dei collegamenti; essa è comunque praticabile per $n = 4$, al più fino a $n = 8$.

Anche per la cache dati il caso base $n = 2$ non pone problemi, essendo richiesti fino a due accessi con tempo di servizio di due cicli di clock.

La realizzazione della cache dati a più larga banda fa di regola uso dell'organizzazione *interallacciata*, con un numero di moduli dimensionato sul massimo teorico della banda richiesta, cioè n, oppure dimensionato in funzione della media degli accessi che statisticamente possono essere effettuati contemporaneamente.

L'Unità Esecutiva a larga banda è stata oggetto della sezione 1. Passando dall'architettura pipeline scalare al caso superscalare base $n = 2$, di nuovo non ci sono problemi. Per n più grande, il numero di unità funzionali deve essere aumentato.

A differenza dell'architettura scalare, ora la parallelizzazione della EU con partizionamento funzionale più pipeline non può garantire la massima banda teorica richiesta, come invece sarebbe possibile con una soluzione farm (che, come visto nella sez. 1, viene scartata per motivi di costo e di area). Ne consegue una possibile degradazione nel tempo di servizio, che ovviamente varia da programma a programma. Dimensionamenti ragionevoli dell'Unità Esecutiva per macchine superscalari, orientate al

calcolo scientifico, portano ad avere un numero abbastanza vicino ad *n* di unità per operazioni corte in virgola fissa e LOAD, e circa la metà di unità funzionali per operazioni lunghe e in virgola mobile.

La banda richiesta all'Unità Istruzioni è, in linea di principio, sempre raggiungibile in quanto si tratta di esplicitare parallelismo a livello di ciclo di clock, o meglio di 2τ, per operazioni che in ultima analisi sono funzioni pure, dunque ottenibili con opportune reti combinatorie. Nessun problema presenta il caso base di $n = 2$: IU ha a disposizione due cicli di clock per decodificare e inoltrare/eseguire due istruzioni, cosa che non richiede tecniche particolari.

Ovviamente, la complessità della IU aumenta più che linearmente con *n* e, per valori elevati di *n*, occorre trovare altre soluzioni basate sulle forme di parallelismo. Questo problema verrà discusso in una sezione successiva.

Problemi addizionali nascono per le macchine a *lunghezza di istruzione variabile*, tipicamente Cisc. In questo caso, IU dovrebbe individuare e decodificare sequenzialmente le possibili istruzioni distinte. Il conseguente overhead è solo in parte ridotto adottando una interpretazione delle istruzioni Cisc in una sequenza di istruzioni Risc-like, facendo precedere ogni istruzione Cisc da un opportuno pseudo-opcode per individuarne la classe: a riprova che l'idea dell'architettura ILP nasce contestualmente all'idea Risc.

Un funzionamento out-of-order e le tecniche più sofisticate di ottimizzazione dinamica (Register Renaming, predizione del salto, trattamento delle interruzioni ed eccezioni fuori ordine) portano ad un significativo aumento della complessità delle varie unità, e quindi dell'area e della potenza dissipata. Anche per questi aspetti, in una successiva sezione discuteremo soluzioni architetturali e tendenze.

3.1.2 Valutazione delle prestazioni

È intuitivo comprendere come l'efficienza relativa diminuisca sensibilmente, e quindi le prestazioni aumentino sempre meno, con l'aumentare di *n*. In effetti, nelle macchine superscalari realizzate negli anni immediatamente precedenti all'arresto della legge di Moore, il guadagno in prestazioni rispetto alle versioni con più basso parallelismo talvolta non è stato significativo, a fronte di un aumento considerevole di costo e di potenza dissipata. Anche prescindendo da considerazioni di complessità architetturale, la diminuzione di efficienza relativa al crescere di *n* è causata da:

- una carenza di parallelismo a livello di istruzioni di un programma sequenziale,
- un aumento del fattore di utilizzazione, nel modello a coda cliente-servente, che comporta un aumento del ritardi dovuti a dipendenze logiche,
    - sia per la maggiore frequenza con la quale più istruzioni contemporanee chiedono servizi,
    - sia per la teorica carenza di risorse in termini di unità funzionali e banchi di registri.

Il nostro obiettivo è di avere una chiara idea di questi fenomeni attraverso lo studio di un *modello dei costi*, al solito il più possibile semplice e maneggevole, che rappresenterà l'estensione al caso superscalare del modello della sezione 1 per architetture scalari.

Nel far questo, adotteremo l'approccio della sezione 2 di distinguere criticamente tra *ottimizzazioni dinamiche con funzionamento out-of-order* e *ottimizzazioni statiche con funzionamento fondamentalmente in-order*.

I parametri oggetto di valutazione sono, come nell'architettura scalare:

- *tempo di servizio per istruzione*, *T*, e quindi tempo di completamento che è all'incirca proporzionale a *T*,
- *efficienza relativa* della CPU, ε.

Inoltre, valuteremo il *guadagno in prestazioni rispetto alla versione scalare* corrispondente. In base alle definizioni del Cap. X, sez. 3, si tratta del parametro *scalabilità*:

$$s = \frac{T_{scalare}}{T_{superscalare}}$$

Si noti che i due parametri, ε e *s*, **non** sono legati dalla relazione $s = \varepsilon n$, in quanto non è $T_{scalare} = t$, bensì $T_{scalare} \geq t$:

$$\varepsilon_{scalare} = \frac{t}{T_{scalare}}$$

$$\varepsilon_{superscalare} = \frac{t}{n\ T_{superscalare}}$$

Si può invece scrivere:

$$s = n\ \frac{\varepsilon_{superscalare}}{\varepsilon_{scalare}}$$

Riguardo all'aspetto dell'eventuale carenza di risorse, in particolare banda delle Unità Funzionali, all'inizio della sez. 1.3 è stato fatto notare come, in presenza di operazioni aritmetiche complesse, il *fattore di utilizzazione dell'Unità Esecutiva* possa rappresentare un elemento ulteriore di degradazione delle prestazioni, soprattutto per alcuni sistemi commerciali con architettura superscalare. Nel seguito, continueremo a supporre che la banda di EU sia sufficientemente alta da non introdurre questo aspetto nella valutazione delle prestazioni in quanto, altrimenti, poco senso avrebbe questo tipo di architettura.

3.1.3 Come raggruppare le istruzioni

In una prima soluzione, le istruzioni del programma assembler vengono raggruppate a *n a n* in "pacchetti" successivi nell'ordine in cui compaiono, indipendentemente dalla loro semantica e dal loro contenuto.

Se il pacchetto contiene *istruzioni tra loro legate da dipendenza logica*, in particolare IU-EU, la loro lettura contemporanea è stata inutile. Inoltre, dal punto di vista architetturale, la IU si complica in quanto deve prendere in esame tutte le *n* istruzioni in tutte le possibili combinazioni per scoprire eventuali dipendenze. Un problema analogo si ha nella EU_Master nel caso di istruzioni legate da dipendenza EU-EU presenti in uno stesso pacchetto, a meno che una non sia corta in virgola fissa o LOAD. Tale problema è particolarmente grave anche nella DM, in quanto l'esistenza di dipendenze implica l'adozione di strategie efficienti per un uso consistente dei dati in cache.

Inoltre, il pacchetto potrebbe contenere una istruzione di *salto* in una posizione diversa dalla *n*-esima, diciamo la *m*-esima con $m < n$. In questo caso, se il salto deve essere effettuato, tutte le $n–m$ istruzioni successive sarebbero state lette inutilmente, con conseguente degradazione del tempo di servizio, a meno che non si applichi la tecnica di *predizione del salto* (sez. 2.5). In effetti, tale tecniche è stata introdotta proprio per architetture superscalari, al prezzo dell'aumento di complessità discusso a suo tempo. Le degradazioni, e la complessità, aumentano ancor più che linearmente se il pacchetto contiene *più di una istruzione di salto*.

Come sappiamo, tutti questi problemi portano ad architetture complesse, con funzionamento *out-of-order*, soprattutto per far fronte al problema della compatibilità binaria: dovendo eseguire programmi di cui sia disponibile solo l'eseguibile in codice binario, ben poco altro si può fare se non raggruppare le istruzioni a *n* a *n* nell'ordine in cui compaiono nel codice stesso.

Per le architetture scalari, nella sezione 2 è stata già ampiamente discussa la convenienza, in termini di rapporto prestazioni/costo, ad adottare architetture più semplici e che facciano largo uso di ottimizzazioni a tempo di compilazione.

Nel campo delle architetture superscalari, il modello architetturale **Very Long Instruction Word (VLIW)**, pur non avendo avuto successo commerciale di per sé (nel senso della diffusione commerciale di nuovi sistemi, completamente proprietari e interamente basati su questo approccio), è stato adottato in larga parte nelle architetture più avanzate e recenti. Questo approccio fa un uso intensivo di *ottimizzazioni a tempo di compilazione*.

Per i nostri obiettivi, l'ottimizzazione VLIW-like che interessa è quello del *riconoscimento statico di gruppi di istruzioni tra loro indipendenti* dal punto di vista delle dipendenze sui dati: sono questi gruppi di istruzioni indipendenti a formare uno stesso elemento dello stream di istruzioni. Spesso, la banda necessaria è appunto ottenuta mediante l'approccio "a parola lunga" (una locazione contenente *n* parole di istruzione), da cui il nome stesso del modello architetturale.

Nel seguito useremo il termine **istruzioni lunga** al posto di pacchetto di istruzioni o elemento dello stream.

Il modello di architettura superscalare che prenderemo a base per lo studio adotta i seguenti vincoli:

1. **una istruzione lunga contiene solo istruzioni non legate da dipendenze logiche IU-EU**;
2. **una istruzione lunga contiene al più una istruzione di salto**.

Si noti che, se è presente una istruzione di salto condizionato, grazie al vincolo 1 questa è indipendente da tutte le altre presenti nella stessa istruzione lunga. L'applicazione di tecniche di delayed branch rende possibile che istruzioni scalari successive facciano parte dell'istruzione lunga. Pur non essendo strettamente necessario, porremo anche il vincolo che

3. **le dipendenze EU-EU siano assenti in una stessa istruzioni lunga**,

per ragioni di semplicità di funzionamento e del modello dei costi.

Per ottenere questa struttura del programma superscalare, *quando non sia possibile sfruttare completamente tutte le parole dell'istruzione lunga con istruzioni utili, il compilatore inserisce delle NOP*, incluso il caso che l'istruzione di salto non sia l'ultima e che non venga adottato delayed branch.

Si rifletta sul fatto che, nel caso una istruzione lunga contenesse dipendenze logiche o istruzioni di salto non ottimizzate, il risultato a tempo di esecuzione sarebbe l'introduzione di bolle. Equivalentemente, le NOP sono *bolle statiche*, cioè previste a tempo di compilazione, considerate ineliminabili.

L'effetto della codifica VLIW è di semplificare drasticamente il progetto della IU e di tutte le altre unità della CPU superscalare. Al tempo stesso, i suddetti vincoli permettono una analisi molto dettagliata delle possibili ottimizzazioni statiche e si prestano a definire un modello dei costi maneggevole, senza ledere la generalità di concetti e tecniche.

3.2 Studio del caso base: architettura superscalare a 2 vie

Per capire l'essenza del problema, inizieremo la trattazione dal caso base $n = 2$, che verrà successivamente esteso a valori di n più elevati.

3.2.1 Esempi

Esempio 1

Consideriamo un semplice esempio, senza dipendenze né salti:

1. LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. INCR Rb

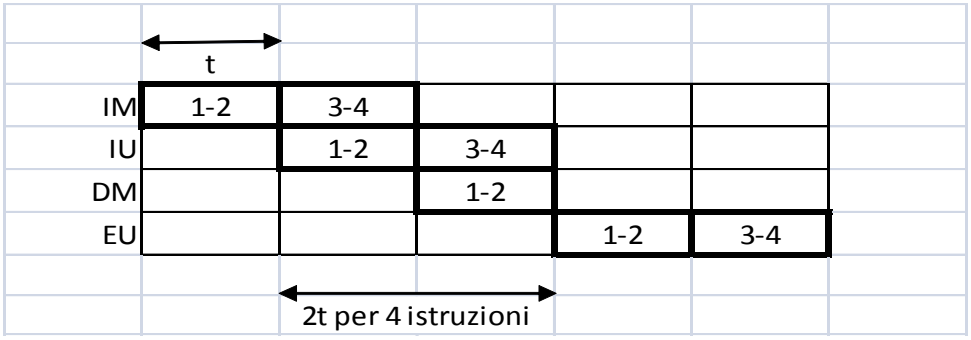
Nel caso scalare si ottiene $T = t = T_{id}$, con $\varepsilon = 1$.

Nel caso superscalare, adottiamo la seguente sintassi per le *istruzioni lunghe*:

1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb

3-4. ADD Ra, Rb, Rx | INCR Rb

Come si vede, ognuna delle istruzioni lunghe 1-2 e 3-4 contiene due istruzioni completamente indipendenti. La loro esecuzione procede come mostrato nella simulazione grafica:



Per il momento, per chiarezza considereremo la EU nel suo complesso, senza distinguere la struttura interna parallela. In seguito applicheremo tutto quanto studiato nella sezione 1.

IM legge due istruzioni alla volta con tempo di servizio $t = 2\tau$. Come detto, per $n = 2$ questo può essere ottenuto anche con una cache monolitica, essendo inserita all'interno di una IM in pipeline.

IU riceve due istruzioni in una stessa parola lunga, le decodifica e le esegue/instrada in un tempo di servizio $t = 2\tau$: questo può anche essere fatto parallelamente, o anche sequenzialmente per $n = 2$. Per l'istruzione lunga 1-2 IU calcola due indirizzi e invia a DM due richieste di lettura e alla EU due richieste di esecuzione contemporaneamente, così come avviene per la 3-4.

DM riceve contemporaneamente le due richieste di lettura, e le soddisfa entrambe in un tempo di servizio $t = 2\tau$, per $n = 2$ anche con una cache dati monolitica in quanto inserita in una DM in pipeline.

EU riceve da IU la richiesta 1-2, attende le due parole da DM, le riceve contemporaneamente o in pipeline, e quindi le scrive, in parallelo o in pipeline, nei registri destinazione indicati dalla IU. Ricevendo da IU la richiesta 3-4, EU può eseguire le due operazioni in parallelo o in pipeline o (nel caso di operazioni corte in virgola fissa come nell'esempio) anche in serie.

Il tempo di servizio per istruzione e l'efficienza relativa sono dati da

$$T = \frac{2\,t}{4} = \frac{t}{2} = T_{id} \qquad \varepsilon = 1$$

La scalabilità:

$$s = \frac{T_{scalare}}{T_{superscalare}} = 2$$

Quindi, siamo nel caso ideale in cui l'architettura superscalare a 2 vie ha effettivamente raddoppiato le prestazioni, senza scadimento di efficienza.

Esempio 2

Consideriamo ora un esempio contenente una dipendenza logica IU-EU:

1. LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. STORE RC, Ri, Rx

Essendo $k = 1$ e $N_Q = 2$, nel caso scalare si ha bolla ampia $2t$, $\Delta = 2\ t/4$,

$$T = 6\ t/4 \qquad \varepsilon = 0{,}66$$

Nell'architettura superscalare, le istruzioni sono raggruppate come segue:

1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb

3-x. ADD Ra, Rb, Rx | NOP

4-x. STORE RC, Ri, Rx | NOP

Infatti, a causa della dipendenza logica IU-EU indotta dalla 3 sulla 4, le due istruzioni sono inserite in due istruzioni lunghe distinte. Non essendoci la possibilità di riempirle con altre istruzioni, le due istruzioni lunghe contengono ognuna una NOP.

L'istruzione lunga 3-x induce sulla 4-x una dipendenza logica IU-EU di distanza $k = 1$, con $N_Q = 2$, esattamente come nel caso scalare.

Come si vede dalla simulazione grafica,

IM	1-2	3-x	4-x				
IU		1-2	3-x			4-x	
DM			1-2				4
EU				1-2	3		

IU, ricevendo l'istruzione lunga 3-x, ovviamente manda avanti la sola 3. Ricevendo la 4-x, rileva la dipendenza logica sulla 4 e *si blocca in attesa della risoluzione della dipendenza stessa, in quanto la seconda istruzione è NOP*. Nel caso avesse trovato una seconda istruzione utile, avrebbe potuto mandarla avanti e poi aspettare lo sblocco della 4.

La bolla continua ad essere di ampiezza $2t$, quindi

$$\Delta = 2\ t/4$$

Il tempo di servizio e l' efficienza relativa sono dati da:

$$T = 5\ t/4 \qquad \varepsilon = 0{,}4 \qquad s = 6/5 = 1{,}2$$

Il tempo di servizio è migliorato rispetto al caso scalare, ma è ben lontano dalla metà, quindi *l'efficienza relativa è peggiorata rispetto al caso scalare*.

Si osservi che il modello dei costi per l'architettura scalare non fornirebbe un risultato corretto:

$$T_{id} + \Delta = t/2 + 2\ t/4 = t \neq T$$

Infatti, anche se *in questo esempio* la valutazione di Δ è corretta, l'espressione del tempo di servizio non ha tenuto conto della presenza di due *ulteriori bolle statiche* (NOP) (nella simulazione grafica sono riconoscibili in tutti i casi in cui sull'asse IM o IU sono presenti delle "x").

Esempio 3

Questo semplice benchmark contiene un salto:

1. LOOP: LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. *GOTO LOOP*

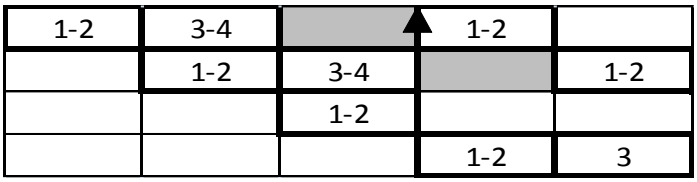
Nell'architettura scalare: $\lambda = 1/4$, con bolla ampia t, $\Delta = 0$, $T = 5\ t/4$, $\varepsilon = 0{,}8$.

Nell'architettura superscalare il codice è:

```
LOOP:   1-2.  LOAD   RA, Ri, Ra  |  LOAD   RB, Ri, Rb
        3-4.  ADD    Ra, Rb, Rx  |  GOTO   LOOP
```

La seconda istruzione lunga contiene una istruzione di salto che, per definizione, è indipendente dall'altra istruzione nella stessa istruzione lunga, ed è l'ultima.

La IU decodificando la 3-4, può inoltrare la 3 alla EU ed eseguire direttamente la 4. Come si vede dalla simulazione grafica, la bolla provocata dal salto è esattamente la stessa presente nel funzionamento dell'architettura scalare, così come la probabilità di salto λ che va valutata sul programma scalare:



Ora si ha:

$T = 3\ t/4$ $\qquad$ $\varepsilon = 0{,}66$ $\qquad$ $s = 5/3 = 1{,}66$

con un apprezzabile miglioramento del tempo di servizio e un peggioramento dell'efficienza relativa.

Esempio 4

Questo esempio contiene una dipendenza logica IU-EU ed un salto combinati:

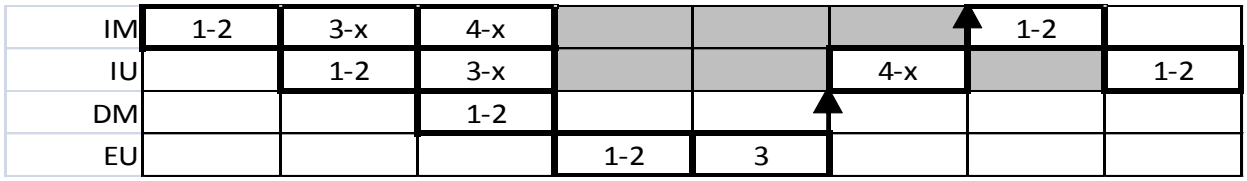
```
1. LOOP:  LOAD   RA, Ri, Ra
2.        LOAD   RB, Ri, Rb
3.        ADD    Ra, Rb, Rx
4.        IF ≠ 0   Rx, LOOP
```

Nell'architettura scalare, si ha $\lambda = 1/4$, $k = 1$ con probabilità 1/4 e $N_Q = 2$, quindi la bolla dovuta al salto è ampia t, quella dovuta alla dipendenza logica $2t$, $T = 7\ t/4$, $\varepsilon = 0{,}57$.

Nell'architettura superscalare il codice è:

```
LOOP:   1-2.  LOAD   RA, Ri, Ra  |  LOAD   RB, Ri, Rb
        3-x.  ADD    Ra, Rb, Rx  |  NOP
        4-x.  IF ≠ 0   Rx, LOOP  |  NOP
```

La simulazione grafica mostra che, *in questo esempio*, le bolle causate da dipendenza logica e salto hanno la stessa ampiezza del caso scalare, ed inoltre sono presenti due bolle statiche:



Le prestazioni sono:

$T = 6\ t/4$ $\qquad$ $\varepsilon = 0{,}33$ $\qquad$ $s = 7/6 = 1{,}17$

3.2.2 Modello dei costi

Nel modello dei costi deve ovviamente comparire il *tempo di servizio ideale*, uguale a *t/2*.

Occorre inoltre correggere il modello dei costi scalare aggiungendo la degradazione dovuta alle *bolle statiche* NOP. A questo scopo, indichiamo con θ la *probabilità di occorrenza delle NOP* valutata *rispetto al numero di istruzioni del codice scalare* (numero di occorrenze delle NOP nel codice superscalare diviso il numero di istruzioni scalari). Nell'Esempio 1 si ha $\theta = 0$, nell'Esempio 2 $\theta = 2/4$, nell'Esempio 3 $\theta = 0$, nell'Esempio 4 $\theta = 2/4$.

Il *modello dei costi del tempo di servizio per istruzione nel caso superscalare a n = 2 vie* diviene:

$$T = \frac{t}{2}(1 + \theta) + \lambda t + \Delta$$

dove Δ in generale si valuta con il metodo studiato nella sezione 2 la presenza dell'Unità Esecutiva parallela, ma tenendo conto di *caratteristiche peculiari* o *ulteriori ottimizzazioni* che sono proprie del caso superscalare e che, in generale, comportano un risultato diverso dal caso scalare (ciò non accadeva negli Esempi 1-4). Tali caratteristiche riguardano la valutazione di *parametri* utilizzati nella formula di Δ, precisamente:

1. *la distanza di dipendenza logica va valutata sul codice superscalare*;

2. se una istruzione lunga contiene una LOAD e una istruzione aritmetica I_a che induce una dipendenza logica IU-EU, e la sequenza critica non contiene ulteriori istruzioni LOAD, allora nella valutazione del ritardo della dipendenza logica va posto

   $$N_Q = 1$$

   Infatti, la LOAD non aumenta la latenza di esecuzione di I_a in quanto quest'ultima, che per definizione è indipendente dalla LOAD, viene inviata in esecuzione direttamente scavalcando la LOAD stessa.

   Ad esempio:

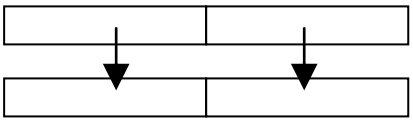
   1-2. LOAD RA, Ri, Ra | ADD Rx, Ry, Rz
   3-4. STORE RC, Ri, Rz | …….

   k = 1, N_Q = 1

   Questa caratteristica ha quindi impatto sulla *determinazione dell'ampiezza della bolla*;

3. se due istruzioni lunghe (I_1 | I_2) e (I_3 | I_4) sono legate da due dipendenze logiche, una indotta da I_1 su I_3 e l'altra da I_2 su I_4 (oppure, indifferentemente, da I_1 su I_4 e da I_2 su I_3), allora viene considerata *una sola dipendenza logica*:

   Ad esempio:

   1-2. ADD Ra, Rb, Rx | INCR Ry
   3-4. STORE RC, Ri, Rx | IF < 0 Ry, LABEL

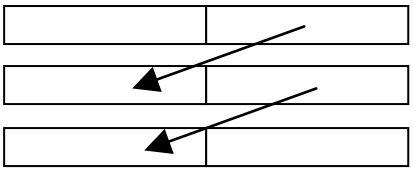
   k = 1, d_1 = 1/…

   Questa caratteristica ha quindi impatto sulla determinazione della *probabilità di dipendenza logica*, che comunque va valutata *rispetto al numero di istruzioni scalari* (numero di dipendenze logiche nel codice superscalare diviso il numero di istruzioni scalari);

4 alle *dipendenze logiche intrecciate* appartengono anche configurazioni *concatenate* del seguente tipo: una sequenza di istruzioni lunghe

$$(I_1 \mid I_2);\ (I_3 \mid I_4);\ (I_5 \mid I_6)$$

dove una delle istruzioni scalari in $(I_1 \mid I_2)$ induce una dipendenza IU-EU su una delle $(I_3 \mid I_4)$ e l'altra istruzione scalare in $(I_3 \mid I_4)$ induce una dipendenza IU-EU su una delle $(I_5 \mid I_6)$:



Ad esempio:

1-2. ADD Ra, Rb, Rx | NOP
3-4. STORE RC, Ri, Rx | INCR Ry
5-6. IF < 0 Ry, LABEL | …

k = 1, d_1 = 1/…

Anche questa caratteristica ha quindi impatto sulla determinazione della *probabilità di dipendenza logica*, sempre da valutare rispetto al numero di istruzioni scalari.

Spesso tale situazione annulla l'effetto della seconda dipendenza logica (come nell'esempio).

Inoltre:

a. *alle istruzioni lunghe possano appartenere istruzioni legate da* **antidipendenza**, *quindi di fatto eseguibili indipendentemente*. Ad esempio:

4-5. STORE RC, Ri, Rx | INCR Ri

IU utilizza il valore corrente aggiornato di RG[Ri] per valutare l'indirizzo di memoria, mentre il prossimo valore assunto da RG[Ri] sarà quello calcolato indipendentemente da EU.

Questo vale sia per antidipendenze IU-EU che EU-EU. Infatti, nel caso EU-EU, la sincronizzazione dei registri è effettuata da EU_Master;

b. in generale *la tecnica del delayed-branch può richiedere di spostare/replicare istruzioni lunghe.*

Esempio 5

Consideriamo l'esempio della somma di vettori, prima in versione non ottimizzata:

LOOP: 1. LOAD RA, Ri, Ra
2. LOAD RA, Ri, Rb
3. ADD Ra, Rb, Rx
4. STORE RC, Ri, Rx
5. INCR Ri
6. IF < Ri, RN, LOOP

Nel caso scalare, si ha probabilità di salto $\lambda = 1/6$, dipendenza logica di distanza $k = 1$ con probabilità 2/6 e in media $N_Q = 3/2$,quindi

$$\Delta = 3\ t/6 \qquad T = t + \lambda\ t + \Delta = 10\ t/6 \qquad \varepsilon = 0{,}6$$

Nel caso superscalare a 2 vie, il codice non ottimizzato è:

```
LOOP:   1-2. LOAD   RA, Ri, Ra…  |  LOAD   RA, Ri, Rb
        3-x. ADD    Ra, Rb, Rx   |  NOP
        4-5. STORE  RC, Ri, Rx   |  INCR   Ri
        6-7. IF <   Ri, RN, LOOP |  NOP
```

Come esemplificato sopra, l'istruzione lunga 4-5 contiene una antidipendenza che non ha effetto, anzi favorisce il parallelismo. Inoltre, si ha una dipendenza *concatenata* dalla 3-x alla 4-5 alla 6-7, che annulla l'effetto della seconda.

Quindi: $\theta = 2/6$ $\lambda = 1/6$ $k = 1$, $\boldsymbol{d_1 = 1/6}$, $N_{Q1} = 2$

da cui: $\Delta = 2\ t/6$ $T = 7\ t/6$ $\varepsilon = 0{,}43$ $s = 10/7 = 1{,}43$

come si può verificare anche con la simulazione grafica:

IM	1-2	3-x	4-5	6-7				1-2	
IU		1-2	3-x	5		4	6-7		1-2
DM			1-2				4		
EU				1-2	3	5			

Si noti come IU abbia un funzionamento *in-order*, in quanto ha ricevuto le istruzioni 4 e 5 in una stessa istruzione lunga e, rilevando che la 4 è bloccata e la 5 è abilitata, può inoltrare la 5 in attesa dello sblocco della 4.

Consideriamo ora il codice ottimizzato (con la base dell'array C decrementata):

```
LOOP:   1. LOAD   RA, Ri, Ra
        2. LOAD   RA, Ri, Rb
        3. INCR   Ri
        4. ADD    Ra, Rb, Rx
        5. IF <   Ri, RN, LOOP, delayed_branch
        6. STORE  RC, Ri, Rx
```

Nel caso scalare si ha $T = 7\ t/6$ e $\varepsilon = 0{,}86$.

Nel caso superscalare:

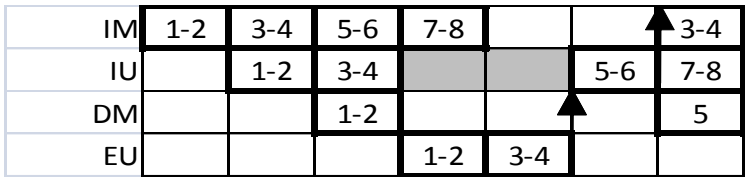
```
        1-2. LOAD   RA, Ri, Ra   |  LOAD   RA, Ri, Rb
LOOP:   3-4. INCR   Ri           |  ADD    Ra, Rb, Rx
        5-6. IF <   Ri, RN, LOOP, delayed_branch  |  STORE  RC, Ri, Rx
        7-8. LOAD   RA, Ri, Ra   |  LOAD   RA, Ri, Rb
```

La 3-4 induce sulla 5-6 *una* dipendenza IU-EU di distanza $k = 1$ e probabilità *1/6* con $N_{Q1} = 2$. La tecnica del delayed branch è stata applicata replicando l'istruzione lunga 1-2 dopo l'istruzione lunga 5-6 che contiene l'istruzione di salto.

Si ha: $\theta = 0$ $\lambda = 0$ $\Delta = 2\ t/6$,

quindi: $T = 5\ t/6$ $\varepsilon = 0{,}6$ $s = 7/5 = 1{,}4$

come confermato anche dalla simulazione grafica:



In questo esempio la valutazione di Δ è diversa nel caso scalare e nel caso superscalare.

Esempio 6

Il semplice esempio della reduce della somma permette di mostrare l'applicazione della caratteristica 2 del modello dei costi, insieme alla parallelizzazione di antidipendenze.

Nel codice scalare ottimizzato:

```
        1. LOAD   RA, Ri, Ra
LOOP:   2. INCR   Ri
        3. ADD    Rs, Ra, Rs
        4. IF <   Ri, RN, LOOP
        5. LOAD   RA, Ri, Ra
```

$N_Q = 2$

la dipendenza indotta dalla 2 sulla 4 ha $N_Q = 2$, ottenendo $\Delta = t/4$, $T = 5\ t/4$, $\varepsilon = 0{,}8$.

Nel codice superscalare:

```
        1-2. LOAD   RA, Ri, Ra  |  INCR   Ri
LOOP:   3-4. ADD    Rs, Ra, Rs  |  IF <   Ri, RN, LOOP
        5-6. LOAD   RA, Ri, Ra  |  INCR   Ri
```

$N_Q = 1$

la dipendenza della 1-2 (nella quale si ha l'antidipendenza che favorisce il parallelismo) sulla 3-4 è caratterizzata da $N_Q = 1$. La valutazione fornisce:

$\theta = 0$ $\lambda = 0$ $\Delta = t/4$

$T = 3\ t/4$ $\varepsilon = 0{,}66$ $s = 5/3 = 1{,}66$

3.2.3 Esempi con Unità Esecutiva parallela

Riprendiamo alcuni esempi delle sezioni 1, 2 per valutare programmi sull'architettura superscalare base a 2 vie contenente EU parallela.

Esempio 10, sez. 1.2.4

(loop interno della moltiplicazione matrice-vettore con ottimizzazioni statiche)

```
1.   LOAD   RA, Rj, Ra
2. LOOPj:  LOAD   RB, Rj, Rb
3.         INCR   Rj
4.         MUL    Ra, Rb, Rx
5.         ADD    Rc, Rx, Rc
6.         IF <   Rj, RM, LOOPj, delayed_branch
7.         LOAD   RA, Rj, Ra
```

L'architettura scalare era caratterizzata da $\lambda = 0$, $\Delta = 0$, $T = t$, $\varepsilon = 1$.

Il codice superscalare è:

```
                1-2. LOAD   RA, Rj, Ra  |  LOAD   RB, Rj, Rb
LOOPj:          3-4. INCR   Rj  |  MUL   Ra, Rb, Rx
                5-6. ADD    Rc, Rx, Rc  |  IF <   Rj, RM, LOOPj, delayed_branch
                7-8. LOAD   RA, Rj, Ra  |  LOAD   RB, Rj, Rb
```

Ora si ha una dipendenza logica IU-EU indotta dalla 3-4 sulla 5-6, di distanza $k = 1$, probabilità $1/6$, $N_Q = 2$, con sequenza critica con *non* contribuisce a indurre la dipendenza stessa. Quindi:

$$\Delta_1 = 2\ t/6 \qquad \Delta_2 = 0 \qquad \Delta = 2\ t/6$$

Inoltre: $\theta = 0$, $\lambda = 0$, per cui: $T = 5\ t/6$ $\varepsilon = 0{,}6$ $s = 6/5 = 1{,}2$

come confermato dalla simulazione grafica con IU in-order:

IM	1-2	3-4	5-6	7-8			3-4			
IU		1-2	3-4	5		6	7-8			
DM			1-2							
EU-Mas				1-2	3-4					5
INT Mul						4	4	4	4	

Esaminando la simulazione grafica, si possono notare/ribadire alcune cose interessanti

- relativamente al funzionamento della IU *in-order*: decodificando la 5-6, rileva che 6 è bloccata, ma inoltra regolarmente l'istruzione aritmetica 5;
- relativamente al funzionamento della EU: EU_Master riceve la doppia richiesta di una aritmetica corta e di una lunga, quindi è capace di eseguire l'operazione corta e contemporaneamente di inoltrare la lunga al moltiplicatore pipeline.

Rispetto al caso scalare, si ottiene un certo miglioramento, ma con una efficienza che scende da 1 a 0,6, in quanto il maggior parallelismo è stato in buona parte pagato con la reintroduzione di una dipendenza logica (bolla ampia 2).

L'esempio è appunto significativo per esemplificare come l'aumento di parallelismo dell'architettura superscalare, e la conseguente diminuzione del tempo di servizio, ha come (parziale) contropartita *l'aumento del fattore di utilizzazione in strutture cliente-servente (diminuzione del tempo di interarrivo al servente),* e quindi una degradazione di prestazioni percentualmente maggiore (cioè, una efficienza relativa minore).

Ovviamente, il guadagno rispetto all'architettura scalare sarebbe stato percentualmente maggiore nel caso di codice non ottimizzato, ma *le ottimizzazioni statiche giocano comunque un ruolo significativo*.

Infatti, si consideri il codice non ottimizzato, eccetto l'uso del delayed branch (Esempio 9, sez. 1.2.3):

```
1.   LOAD   RA, Rj, Ra
2. LOOPj:    LOAD   RB, Rj, Rb
3.           MUL    Ra, Rb, Rx
4.           ADD    Rc, Rx, Rc
5.           INCR   Rj
6.           IF <   Rj, RM, LOOPj, delayed_branch
7.           LOAD   RA, Rj, Ra
```

le cui prestazioni sull'architettura scalare erano $\Delta = 2\ t/6$, $T = 8\ t/6$, $\varepsilon = 0{,}75$.

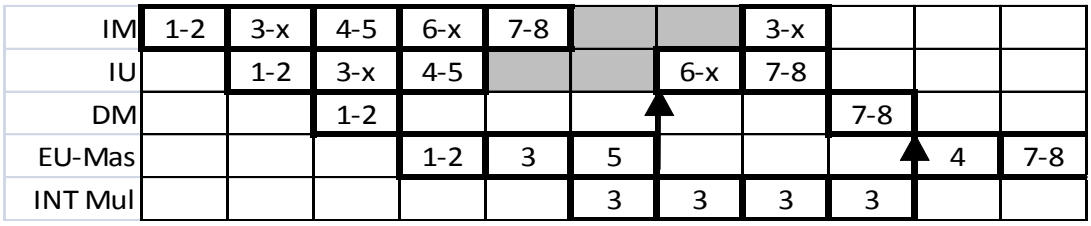
Nel caso superscalare un possibile codice non ottimizzato (eccetto l'uso del delayed branch) è:

```
              1-2. LOAD   RA, Rj, Ra   |   LOAD   RB, Rj, Rb
LOOPj:        3-x. MUL    Ra, Rb, Rx   |   NOP
              4-5. ADD    Rc, Rx, Rc   |   INCR   Rj
              6-x. IF <   Rj, RM, LOOPj, delayed_branch   |   NOP
              7-8. LOAD   RA, Rj, Ra   |   LOAD   RB, Rj, Rb
```

La presenza di due NOP comporta $\theta = 2/6$. Inoltre permane la dipendenza logica IU-EU di distanza $k = 1$, con probabilità 1/6 e $N_Q = 2$, quindi è ancora $\Delta = 2\ t/6$, da cui:

$$T = t \qquad \varepsilon = 0{,}5 \qquad s = 8/6 = 1{,}33$$

come confermato:



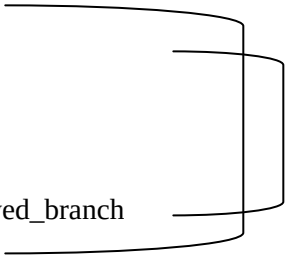
Il miglioramento rispetto al caso scalare è percentualmente maggiore nel caso non ottimizzato, ma, senza ottimizzazioni statiche, le prestazioni continuano ad essere significativamente peggiori rispetto al caso con ottimizzazioni statiche.

Esempio della sez. 2.1.1,

$$\forall\ i = 0\ ..\ N–1 :\quad C[i] = A[i] * B[i] + A[i] / B[i]$$

Versione con ottimizzazioni statiche:

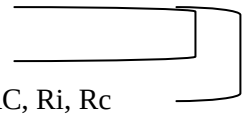
```
1.            LOAD   RA, Ri, Ra
2.            LOAD   RB, Ri, Rb
3. LOOP:      MUL    Ra, Rb, Rx
4.            DIV    Ra, Rb, Ry
5.            ADD    Rx, Ry, Rc
6.            INCR   Ri
7.            LOAD   RA, Ri, Ra
8.            LOAD   RB, Ri, Rb
9.            IF <   Ri, RN, LOOP, delayed_branch
10.           STORE  RC, Ri, Rc
```



Con l'architettura scalare si aveva: $\Delta = 2\ t/8,\ T = 10\ t/8,\ \varepsilon = 0{,}8.$

Con l'architettura superscalare:

```
              1-2. LOAD   RA, Ri, Ra  |  LOAD   RB, Ri, Rb
              3-4. MUL    Ra, Rb, Rx  |  DIV    Ra, Rb, Ry
LOOP:         5-6. ADD    Rx, Ry, Rc  |  INCR   Ri
              7-8. LOAD   RA, Ri, Ra  |  LOAD   RB, Ri, Rb
              9-10. IF <  Ri, RN, LOOP, delayed_branch  |  STORE  RC, Ri, Rc
              11-12. MUL  Ra, Rb, Rx  |  DIV    Ra, Rb, Ry
```



Si ha:

$$\theta = 0\ e\ \lambda = 0.$$

Esistono due dipendenze logiche IU-EU annidate, della quali

- quella esterna (indotta dalla 5-6 sulla 9-10, probabilità $d_k = 1/8$, distanza $k = 2$, $N_Q = 2$, $L_{pipe\text{-}k} = 0$, probabilità $d_h = 1/8$, $h = 1$, $L_{pipe\text{-}h} = 4$):

  $\Delta_{1\text{-esterna}} =$ t/8 $\qquad \Delta_{2\text{-esterna}} = 4$ t/8 $\qquad \Delta_{\text{esterna}} = 5$ t/8

- quella interna (indotta dalla 5-6 sulla 7-8, probabilità $d_k = 1/8$, distanza $k = 1$, $N_Q = 2$, $L_{pipe\text{-}k} = 0$, probabilità $d_h = 0$):

  $\Delta_{1\text{-interna}} = 2$ t/8 $\qquad \Delta_{2\text{-interna}} = 0 \qquad \Delta_{\text{interna}} = 2$ t/8

Quindi, con IU in-order:

$$\Delta = 5\ t/8 \qquad T = 9\ t/8 \qquad \varepsilon = 0{,}44$$

Come si vede, è fatalmente aumentato il ritardo dovuto a dipendenze logiche rispetto al caso scalare, pur ottenendo un certo miglioramento del tempo di servizio.

Facendo riferimento alla simulazione:

IM	1-2	3-4	5-6	7-8	9-10	11-12			5-6				
IU		1-2	3-4	5-6			7-8	9			10	11-12	5-6
DM			1-2					7-8				10	
EU-Mas				1-2	3-4	6			7-8	5			
INT Mul						3	3	3	3				
INT Div						4	4	4	4				

è da notare come, nel caso superscalare, *l'esecuzione contemporanea di più operazioni aritmetiche lunghe in altrettante Unità Funzionali indipendenti porti vantaggio in termini di latenza rispetto alla loro esecuzione in pipeline nella stessa unità, quando questo contribuisca a minimizzare la distanza di dipendenza logica*. Nell'esempio, l'esecuzione della 3 e 4 in due unità permette di iniziare prima possibile la fase di esecuzione della 5, la quale induce una dipendenza IU-EU sulla 10.

3.3 Architettura superscalare a molte vie

3.3.1 Modello dei costi

Partendo dai risultati ottenuti per il caso base $n = 2$, in questa sezione approfondiremo la trattazione delle architetture superscalari per valori più elevati di *n*.

Il *modello dei costi* del tempo di servizio per istruzione si generalizza in:

$$T = \frac{t}{n}\ (1 + \theta) + \lambda\, t\ + \Delta$$

dove Δ si determina come studiato nella sez. 1.3.2, e valgono le caratteristiche peculiari e ulteriori ottimizzazioni enunciate nella sez. 3.2.2.

Il guadagno (scalabilità) rispetto all'architettura scalare tende a *n* nel caso ideale, in genere è (sostanzialmente) minore di *n*.

3.3.2 Unità Istruzioni ad alta banda

Come già discusso, teoricamente è possibile che un'unica unità IU sia capace di decodificare e inoltrare o eseguire *n* istruzioni in 2 cicli di clock, sfruttando parallelismo nelle microistruzioni.

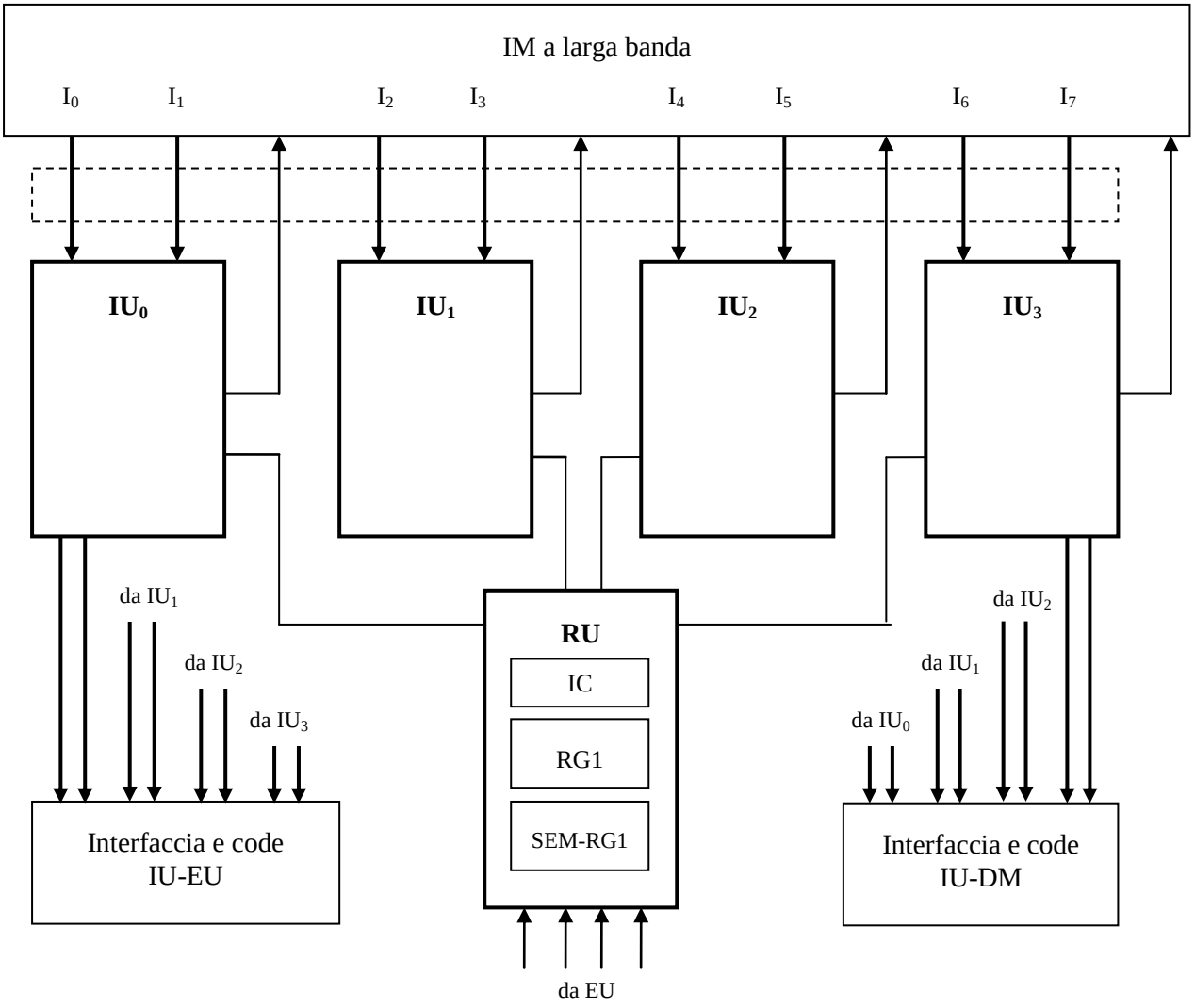
Questo approccio monolitico, o **centralizzato**, è di complessità pressoché esponenziale in *n*, per cui è praticabile solo per bassi valori di *n*. Per valori elevati di *n* è inevitabile introdurre sequenzializzazioni, e quindi aumento sia del tempo di servizio che della latenza.

Schema con IU parzialmente replicata

Un approccio concettualmente più praticabile consiste nell'adottare un certo numero di IU identiche più semplici ed affidare ad ognuna un limitato numero di istruzioni. L'approccio è favorito da una rappresentazione VLIW delle istruzioni lunghe, in quanto *le unità lavorano su istruzioni indipendenti ed al più una su una istruzione di salto*. Possiamo pensare di adottare *n* IU scalari, oppure *n/2* IU superscalari a 2 vie.

Nella figura seguente è mostrato il caso di una IU superscalare a 8 vie realizzata mediante 4 IU a 2 vie, strutturalmente identiche.

In questa prima versione, il *contatore istruzioni e la copia dei registri generali, con i semafori associati,* sono *centralizzati* in unica copia inserita in un'unità a parte, indicata con RU (*Unità Registri*).



Anzitutto conviene che i risultati siano comunicati alle IU da parte da (delle) *EU_Master*, invece che direttamente dalle Unità Funzionali. Agli effetti delle dipendenze logiche, questo aumenta di *1t* la latenza $L_{pipe\text{-}k}$ o $L_{pipe\text{-}h}$, come se ci fosse uno stadio in più nel pipeline delle Unità Funzionali stesse.

RU ascolta e serve contemporaneamente le richieste di tutte le IU ed i messaggi di aggiornamento dalla EU. Secondo un meccanismo a domanda e risposta, la latenza per inviare una richiesta e ricevere la risposta è uguale a *1t.* Tenendo conto della comunicazione di risultati via EU_Master*, l'overhead complessivo* è uguale a *2t:* si tratta di un valore *non inferiore a quella che sarebbe stata la penalità con una IU centralizzata*, ma con una complessità strutturale inferiore.

La richiesta di una IU può essere di incremento del semaforo di un registro generale o di lettura; nel secondo caso, RU può condizionare la risposta al valore del semaforo. Ovviamente, le memorie dei registri e semafori sono a larga banda, cioè capaci di più letture e più scritture contemporanee.

In base al vincolo che le istruzioni di una istruzione lunga sono *indipendenti* (e quindi che non esistono "dipendenze di uscita" in istruzioni aritmetiche o LOAD di una stessa istruzione lunga),

- le richieste di modifica semaforo da parte di più IU sono relative a semafori indipendenti,
- le richieste di modifica registro e modifica semaforo da parte di più EU sono relative a registri e semafori indipendenti.

In base al vincolo che *al più una istruzione scalare per istruzione lunga può essere di salto*, il *contatore istruzioni* verrà letto e modificato da una sola delle IU, la quale provvederà a comunicare le richieste di salto alla IM. Questa IU sarà anche quella incaricata del trattamento *interruzioni*. Potremmo anche realizzare le IU in modo asimmetrico, prevedendo che una ben determinata di esse (la IU_0) gestisca il contatore istruzioni e tratti le interruzioni, ma ciò non è strettamente necessario.

Ogni IU inoltra istruzioni e richieste di accesso in memoria a EU e a DM attraverso opportune interfacce e code.

Allo scopo di limitare il numero di collegamenti, può essere posto *un limite superiore alla banda di comunicazione*, in base a considerazioni statistiche. Ad esempio, prevedere tutte le *n* potenziali richieste contemporanee verso EU, ma limitare a una per IU le richieste a DM.

Per avere una esecuzione consistente delle istruzioni di LOAD, le richieste da parte di IU, sia a DM che a EU, sono accompagnate da un *identificatore unico*, che viene propagato da DM a EU. Questo può essere dato dal valore del contatore istruzioni o, più semplicemente, dalla coppia (identificatore dell'istruzione lunga, posizione dell'istruzione scalare nell'istruzione lunga) in quanto ogni IU può generare indipendentemente il valore della prima componente della coppia contando le istruzioni lunghe ricevute.

Schema con IU completamente replicata

Nella figura seguente è mostrato il caso in cui contatore istruzioni e registri generali sono replicati: *ogni IU dispone di una propria copia dei registri generali, con i relativi semafori, e di una copia del contatore istruzioni.*

Interconnessione

Le strutture di interconnessione devono essere tali da limitare il numero di collegamenti a valori ragionevoli.

Occorre una struttura *n x n*, capace di mettere in comunicazione le *n* EU_Master con le *n* IU, ed anche ogni IU con qualunque altra IU. Nel campo delle architetture parallele, una struttura di interconnessione *n x n* va sotto il nome di *crossbar*. In generale, il suo costo è inaccettabile per valori elevati di *n* e quando i collegamenti siano inter-chip; nel nostro caso, valori di *n* dell'ordine di 4-8 sono ancora accettabili, specie per una struttura *intra-chip*. In figura è mostrata una *Unità di Switch* (SU) il cui compito è appunto la semplice implementazione di un interfacciamento *n x n* con tutte le possibili comunicazioni in parallelo.

La presenza di SU introduce una latenza di *1t*, che sostituisce quella che, nella schema parzialmente replicato, era introdotta da RU.

Nel caso che il costo di un crossbar sia troppo elevato, esistono strutture di interconnessione, dette *di grado limitato*, aventi un costo molto minore e latenze tipicamente di ordine *O(lg n)*, ad esempio un albero binario o quaternario. In questi casi la latenza aumenta di $t\ lg_2 n$ o $t\ lg_4 n$.

Sincronizzazione

Studiamo come, in presenza di completa replicazione, i contenuti dei registri assumono valori consistenti.

La modifica di IC avviene da parte di una sola IU, quindi non ci sono problemi di consistenza.

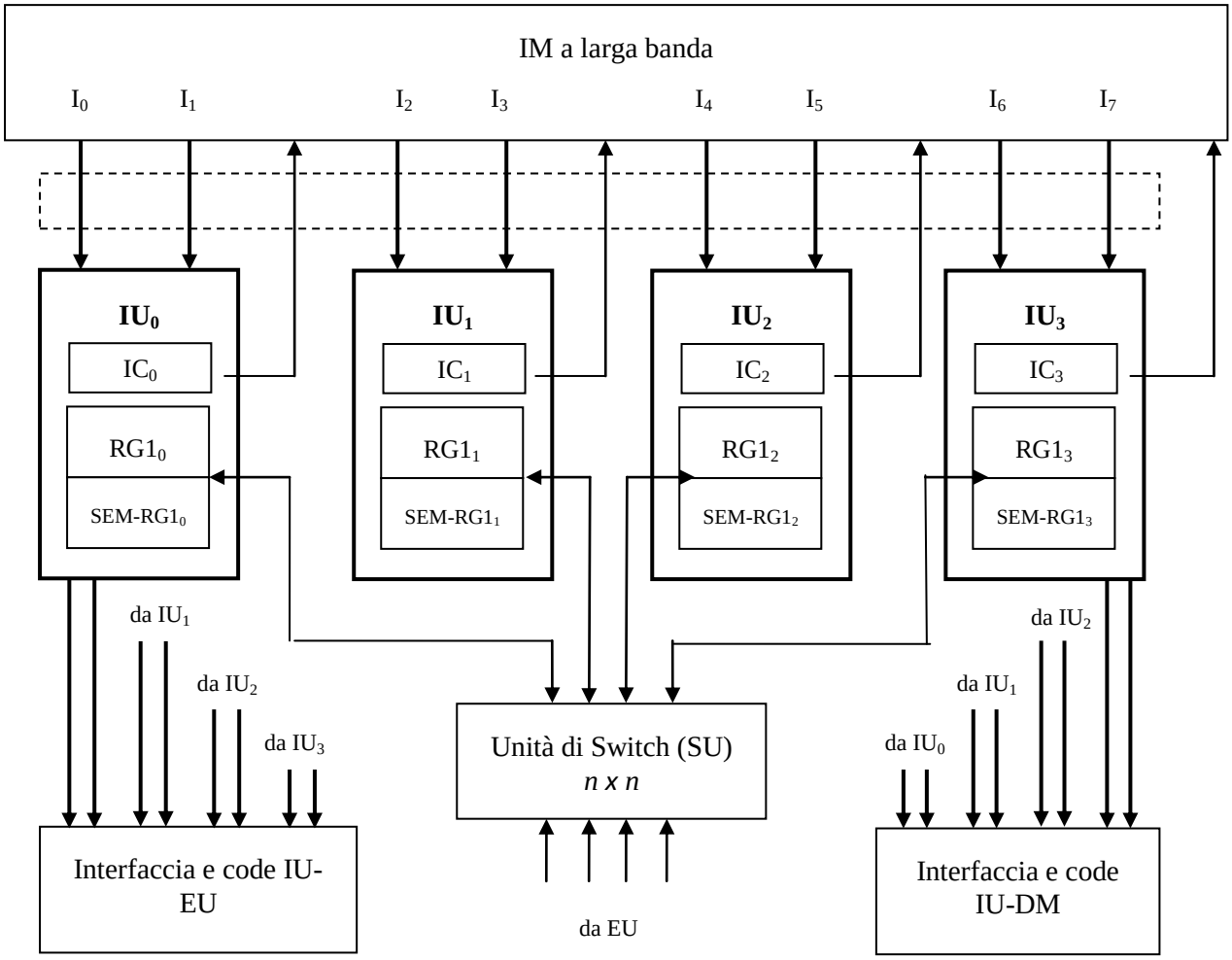
La modifica dei semafori avviene : *1)* su richiesta di EU (decremento), o *2)* da parte di IU stessa (incremento). Ogni modifica è comunicata per diffusione a tutte le altre IU. Il caso *1)*

non pone problemi: come per i registri generali, al più potrà accadere che, per qualche motivo, un semaforo in IU_i sia aggiornato in ritardo rispetto allo stesso semaforo in IU_j, ma questo non può portare a inconsistenza. Nemmeno il caso *2)* pone problemi, in quanto i casi critici sarebbero:

a) più istruzioni aritmetiche scrivono nella stesso registro (lo stesso semaforo dovrebbe essere incrementato da più IU): si tratta di un caso di dipendenza di uscita (seconda condizione di Bernstein) che è evitata del tutto, o comunque è evitata nella stessa istruzione lunga;

b) una istruzione aritmetica che scrive in un registro ed una istruzione (eseguita da IU) che legge lo stesso registro; per appartenere alla stessa istruzione lunga l'unico caso è che siano legate da antidipendenza che, come visto, non crea problemi di parallelismo e di sincronizzazione.

La modifica dei registri generali avviene solo in seguito a richieste da parte di EU: la richiesta vien inviata per diffusione a tutte le IU, quindi ogni IU aggiorna la propria copia senza problemi di consistenza, in quanto l'uso dei registri è sincronizzato dai semafori.



In linea di principio, l'architettura con IU replicata comporta il minimo overhead possibile nell'uso dei registri e contatore istruzioni, comunque superiore all'overhead nell'architettura scalare.

3.3.3 Unità Cache Dati ad alta banda

Come detto, nell'unità DM la cache dati è realizzata in modo *interallacciato*, tipicamente con *n* moduli. Questo permette di accedere a *dati singoli* in parallelo, quando non vi sia conflitto tra richieste contemporanee, ed anche di accedere a *blocchi* di dati in un unico ciclo di clock allo scopo di velocizzare i trasferimenti in caso di fault di cache.

DM è interfacciata verso IU da una unità *DM_Master*, che smista le richieste contemporanee agli opportuni moduli di cache in un unico ciclo di clock e, in caso di conflitto, impone un ordinamento che comporta la perdita di *1t* alla richiesta non servita.

Ovviamente la banda ideale di una memoria interallacciata è uguale a *n* accessi per tempo di accesso, ma la sua banda effettiva risulterà minore a causa dei *conflitti* di più richieste a uno stesso modulo. Vogliamo dare una valutazione attendibile della banda effettiva.

Per una memoria interallacciata è universalmente riconosciuto un modello statistico in base al quale, considerando uno stream di richieste di accesso tra loro indipendenti, la probabilità che una qualunque richiesta sia diretta ad un qualunque modulo è costante e uguale a *1/n*.

Valutiamo la probabilità *P(k)* che una sequenza di *k* richieste consecutive non generi conflitto (cioè, siano dirette a moduli distinti) e la *(k+1)*-esima sia in conflitto con una delle *k* precedenti.

Il caso base è *P(2)*. Sia la prima richiesta diretta al modulo M_i. La probabilità che la seconda richiesta sia diretta ad un modulo M_j, con $j \neq i$, è data da

$$p' = 1 - \frac{1}{n}$$

La probabilità che la terza richiesta sia diretta a M_i oppure a M_j è data da

$$p'' = \frac{1}{n} + \frac{1}{n}$$

Quindi:

$$P(2) = p' * p'' = \frac{2\,(n-1)}{n^2}$$

Generalizzando per induzione ad una sequenza di *k* richieste indipendenti, con $k \leq n$:

$$P(k) = \frac{k\,(n-1)!}{n^k\,(n-k)!}$$

La banda della memoria interallacciata è quindi data da:

$$B_M = \sum_{k=1}^{n} k * P(k) = \sum_{k=1}^{n} \frac{k^2\,(n-1)!}{n^k\,(n-k)!}$$

Si può verificare che tale espressione può essere approssimata a:

$$B_M = n^{0,56}$$

con un errore non superiore al 4% per $n \leq 45$, quindi ampiamente accettabile in tutti i casi pratici.

In conclusione, con una sequenza di *n* richieste indipendenti, poco più della radice quadrata di *n* moduli sono effettivamente sfruttati.

Questo rende ragione del fatto di limitare la banda massima delle richieste contemporanee da parte di IU per ridurre il costo dei collegamenti.

3.3.4 Unità Esecutiva ad alta banda

Lo schema di base della sezione 2 è ancora valido: un certo numero di Unità Funzionali specializzate e in pipeline, ed una unità EU_Master che

- contiene la copia principale dei registri generali ed in virgola mobile, ed eventuali copie aggiuntive o buffer di riordino per ottimizzazioni dinamiche out-of-order e register renaming,
- riceve le richieste da IU, verifica la loro abilitazione e smista quelle abilitate alle apposite Unità Funzionali insieme ai valori degli operandi in registri.

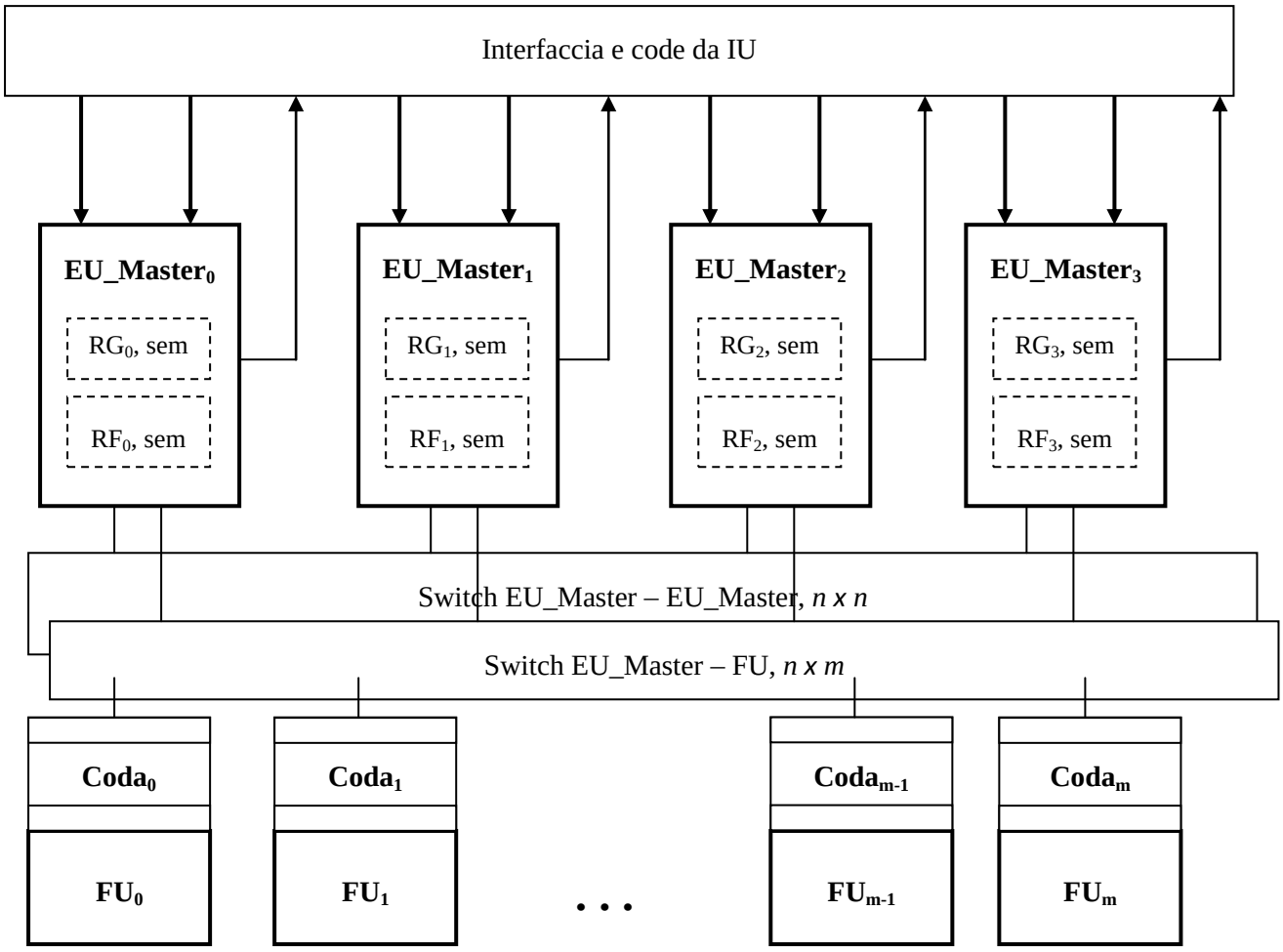
Tutto questo va potenziato per l'architettura superscalare con un numero di Unità Funzionali maggiore di *n*.

EU_Master, ricevendo fino ad *n* richieste indipendenti contemporaneamente, deve poterne verificare l'abilitazione e smistarle in parallelo.

Come per la IU sono possibili gli schemi

1. **centralizzato**,
2. **parzialmente replicato**, con *n* EU_Master, una unità registri generali RU ed una unità registri in virgola mobile RFU, dotate dei semafori di sincronizzazione,
3. **completamento replicato**, con *n* EU Master, ognuna contenente una propria copia dei registri generali, dei registri in virgola mobile e dei rispettivi semafori.

Nella figura seguente è mostrato lo schema completamente replicato.



L'interconnessione è realizzata da due *Unità di Switch*, una per le comunicazioni tra EU_Master e Unità Funzionali (FU), l'altra per le comunicazioni tra EU_Master. La seconda è utilizzata per mantenere la consistenza dei registri e semafori. La struttura di queste Unità di Switch è analoga a quella usata per le IU: si tratta di strutture a *crossbar*, con latenza costante, o strutture *di grado limitato*, con latenza logaritmica.

Ogni EU_Master opera sui semafori in maniera analoga alla IU: quando riceve una istruzione controlla i semafori dei registri sorgente e incrementa il semaforo del registro destinazione; quando riceve un risultato decrementa il semaforo del registro destinazione. Ogni modifica di un semaforo è comunicata per diffusione a tutte le EU_Master. La consistenza è basata sugli stessi principi visti per la IU.

La modifica dei registri, in base ai risultati ricevuti dalla FU delegata, viene anch'essa effettuata per diffusione a tutte le EU_Master, le quali dispongono quindi di copie consistenti in quanto il loro uso è sincronizzato dai semafori.

Inoltre, per potenziare il funzionamento data-flow delle istruzioni aritmetiche abilitate, ma senza avere code associative, una soluzione è quella di dotare ogni Unità Funzionale di una propria coda FIFO d'ingresso (chiamata *reservation station*, nel gergo di alcuni costruttori). Questo permette ad ogni EU_Master di smaltire meglio le istruzioni in ingresso, e di scoprire un numero maggiore di possibili istruzioni abilitate.

4. Multithreading

4.1 Parallelismo tra istruzioni e parallelismo tra programmi

Rivendendo criticamente le architetture ILP studiate, possiamo affermare che il passaggio dall'architettura pipeline scalare alle architetture superscalari non permette di ottenere miglioramenti drastici di prestazioni, principalmente a causa del *limitato grado di parallelismo insito in un singolo programma sequenziale*, normalmente dell'ordine di poche se non pochissime unità.

Da questo discende come, in una architettura Von Neumann, l'esecuzione in parallelo di istruzioni di uno stesso programma sia caratterizzata da degradazioni principalmente dovute a dipendenze sui dati. Le ottimizzazioni sono quindi tutte basate sul tentativo di *mascherare* il più possibile le *latenze* che causano tali degradazioni, tentativo che, in diversi casi, fornisce risultati inferiori alle attese proprio a causa del fatto che il parallelismo tra istruzioni è basso.

Nel settore dell'elaborazione ad alte prestazioni (Cap. X) il problema del parallelismo è principalmente affrontato a livello di *moduli della "grana" di programmi o processi* su architetture parallele di tipo *multiprocessor* (a memoria condivisa) o *multicomputer* (a memoria distribuita). In una architettura multiprocessor più CPU condividono fisicamente la memoria principale (più precisamente, uno o più livelli della gerarchia di memoria) e, istante per istante, sono contemporaneamente attive nell'esecuzione di più processi. Nel caso più semplice questi ultimi possono essere tra loro indipendenti, mentre in generale i programmi paralleli sono costituiti da una collezione di processi cooperanti (mediante messaggi o variabili condivise).

Il settore della programmazione parallela e delle architetture parallele, che ha proprie basi metodologiche sufficientemente assestate, ha ricevuto negli ultimi anni un impulso mai registrato in precedenza grazie allo sviluppo della tecnologia *multicore*, o *Chip MultiProcessor* (*CMP*), cioè la possibilità di integrare su singolo chip una architettura multiprocessor con un certo numero di CPU e alcuni livelli di gerarchia di memoria condivisa. L'impulso decisivo a questo sviluppo è stato causato dalla più volte discussa crisi della legge di Moore per uniprocessor, che ha bloccato l'ulteriore evoluzione delle architetture superscalari e di conseguenza il miglioramento periodico della frequenza del clock. La *legge di Moore* verrà invece applicata al numero di CPU (core) per chip, che quindi ci si aspetta raddoppiare a distanza di 18-24 mesi.

Contemporaneamente allo sviluppo della programmazione parallela applicata ai multiprocessor, e di recente ai CMP, la linea dell'architettura della *singola CPU ILP* si è anch'essa sviluppata nella direzione del *parallelismo tra programmi*, nel tentativo dei costruttori di preservare i pesanti investimenti effettuati su questa linea. L'idea di base è: visto che un singolo programma è caratterizzato da limitato parallelismo tra istruzioni, proviamo ad eseguire contemporaneamente su una stessa CPU ILP istruzioni appartenenti a più programmi. A condizione di disporre di un opportuno supporto architetturale alla concorrenza tra programmi, questo significa che le unità dell'architettura pipeline sono attive ad elaborare più istruzioni appartenenti a programmi distinti *durante uno stesso slot* (*t*) o comunque in un intervallo di pochissimi slot. In altri termini: *usare il parallelismo tra istruzioni per avere anche parallelismo tra programmi*.

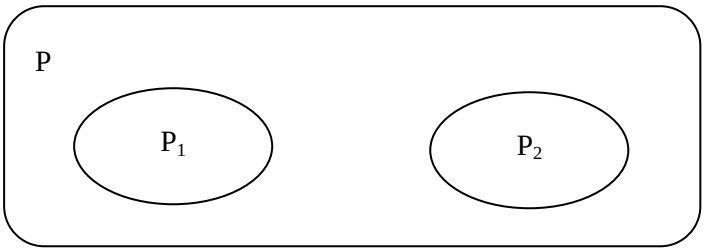
Questa idea, denominata *multithreading* (*MT*) per i motivi che esporremo in seguito, può essere interpretata in due modi distinti e in qualche modo interrelati:

1. come una *alternativa tecnologica alle architetture multiprocessor*: anche nel caso MT l'area del chip viene usata per permettere l'esecuzione di più programmi contemporaneamente,
2. come una *evoluzione tecnologica delle architetture ILP*: sfruttare meglio le risorse integrate sul chip per ridurre le cause di degradazione delle prestazioni delle CPU ILP.

Prima di discutere e approfondire dove possono portare queste due interpretazioni, vediamo di chiarire le idee con semplici esempi.

4.1.1 Esempio: multiprocessor con *n* CPU scalari *vs* CPU multithreaded superscalare a *n* vie

Consideriamo la computazione parallela P schematizzata in figura:



P consta di due moduli P_1, P_2, completamente *indipendenti*, definiti dai seguenti codici:

P_1::

1. MUL Ra1, Rb1, Rx1
2. DIV Rc1, Rd1, Ry1
3. ADD Rx1, Ry1, Rz1
4. STORE RC1, Ri1, Rz1

P_2::

a. LOAD RA2, Rj2, Rp2
b. LOAD RB2, Rj2, Rq2
c. SUB Rp2, Rq2, Rr2
d. INCR Rr2
e. STORE RD2, Rj2, Rr2

Eseguiamo questa computazione su architetture diverse, realizzate con tecnologie compatibili.

1) La prima architettura è un **multicore** CMP con $n = 2$ CPU identiche **scalari**. Supponendo che P_1 sia interamente caricato nella cache primaria di una delle due CPU e P_2 nell'altra, i tempi di servizio per istruzione e di completamento sono dati da:

$$T_1 = 9\ t/4 = 2{,}25\ t \qquad T_{c1} \sim 9\ t$$

$$T_2 = 7\ t/5 = 1{,}4\ t \qquad T_{c2} \sim 7\ t$$

La computazione P nel suo complesso è quindi caratterizzata dalle seguenti prestazioni su questa architettura:

$$T_c = max\ (T_{c1}, T_{c2}) \sim 9\ t$$

Numero di istruzioni complessivamente eseguite: $N = 9$

$$T \sim T_c / N = t$$

CMP è equivalente a una CPU capace di eseguire mediamente una istruzione ogni t secondi.

2) La seconda architettura è una singola CPU **superscalare** a $n = 2$ vie con **multithreading**. Questo permette di eseguire, nelle stessa istruzione lunga, due istruzioni appartenenti o solo al modulo P_1, o solo al modulo P_2, o una di P_1 e l'altra di P_2. Senza addentarci su quali caratteristiche in più debba avere l'architettura superscalare per essere anche multithreading, per il momento è sufficiente supporre che sia stato generato staticamente un unico codice superscalare, ad esempio:

P::

1-a. MUL Ra1, Rb1, Rx1 | LOAD RA2, Rj2, Rp2
2-b. DIV Rc1, Rd1, Ry1 | LOAD RB2, Rj2, Rq2
3-c. ADD Rx1, Ry1, Rz1 | SUB Rp2, Rq2, Rr2
4-d. STORE RC1, Ri1, Rz1 | INCR Rr2
x-e. NOP | STORE RD2, Rj2, Rr2

Le prestazioni sono:

$$T = 10\ t/9 \qquad T_c \sim 10\ t$$

Come si vede, le prestazioni sono paragonabili a quelle del multiprocessor. Addirittura, potrebbero essere rese uguali con ottimizzazioni statiche sul codice superscalare, ad esempio:

1-2. MUL Ra1, Rb1, Rx1 | DIV Rc1, Rd1, Ry1
a-b. LOAD RB2, Rj2, Rq2 | LOAD RA2, Rj2, Rp2
3-c. ADD Rx1, Ry1, Rz1 | SUB Rp2, Rq2, Rr2
4-d. STORE RC1, Ri1, Rz1 | INCR Rr2
x-e. NOP | STORE RD2, Rj2, Rr2

In questo caso si ha proprio:

$$T = t \qquad T_c \sim 9\ t$$

In conclusione, sulle due architetture il parallelismo tra moduli è stato esplicitato in modo diverso, ma ottenendo un risultato paragonabile, o addirittura lo stesso risultato, in termini di prestazioni.

L'architettura multiprocessor esegue i due processi in modo completamente disgiunto sulle due CPU disponibili. Ognuna di queste esegue il modulo assegnatole secondo le caratteristiche della sua architettura interna, nel nostro caso pipeline scalare. In particolare, l'esecuzione su ognuna delle due CPU incontrerà le proprie degradazioni (bolle). Se osserviamo l'evoluzione contemporanea di P_1 e P_2 sulle due CPU:

IM	1	2	3	4							
IU		1	2	3						4	
DM											4
EU-Mas			1	2					3		
INT Mul				1	1	1	1				
					2	2	2	2			
IM	a	b	c	d	e						
IU		a	b	c	d			e			
DM			a	b					e		
EU-Mas				a	b	c	d				
INT Mul											

vediamo che ci sono degli slot (*t*) durante i quali le IU di entrambe le CPU sono attive ognuna su una propria istruzione, mentre durante altri slot una delle due è attiva e l'altra è bloccata in attesa che si risolva una dipendenza logica, oppure entrambe le IU sono bloccate.

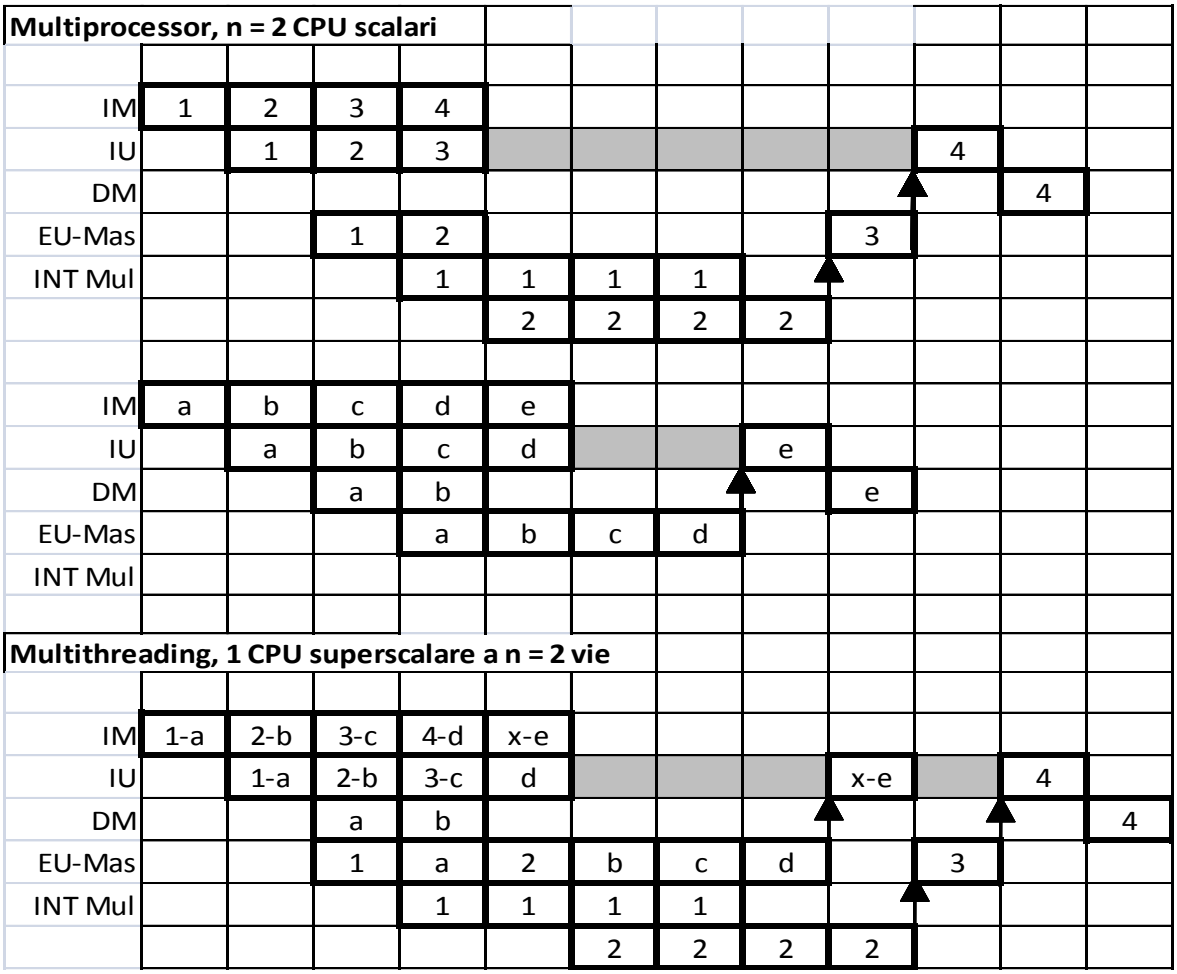
Questa osservazione è la ragione dell'esistenza della linea multithreading. Nell'esecuzione sull'architettura multithreading ci sono degli slot durante i quali IU elabora istruzioni di entrambi i moduli P_1, P_2 e altri slot in cui solo istruzioni di uno dei due moduli viene eseguita mentre una istruzione dell'altro modulo è bloccata, oppure slot durante i quali entrambi i moduli hanno istruzioni bloccate. In altri termini:

a) che il parallelismo tra istruzioni dia luogo a parallelismo tra moduli è ovvio nel caso che istruzioni di entrambi i moduli siano portate avanti da IU,

b) ma altrettanto significativo è il fatto che, quando in uno dei due moduli si verifica una bolla, è possibile che istruzioni dell'altro modulo riescano a coprire la bolla stessa, almeno in parte.

I punti *a)*, *b)* mostrano le due "anime" del parallelismo sui quali fa affidamento una architettura multithreading: *a)* trovare *istruzioni indipendenti* da eseguire simultaneamente, *b) mascherare le latenze*.

Allo scopo, si confronti la simulazione grafica del caso multithreading con le due simulazioni contemporanee del caso CMP (del caso multithreading è mostrata le prima versione, non ottimizzata, in quanto più evidente):



Finora abbiamo confrontato l'architettura multiprocessor avente *n* CPU scalari con l'architettura multithreaded avente una CPU superscalare a *n* vie. Queste considerazioni possono essere generalizzate, nel senso che i casi 1), 2) diventano:

1) architettura multiprocessor avente *p* CPU superscalari a *n/p* vie,
2) architettura multithreaded con una CPU superscalare a *n* vie,

o una varietà di casi misti.

Ovviamente, **le due linee non sono confrontabili per qualunque grado di parallelismo *n***:

1) mentre **l'architettura multiprocessor è implicitamente scalabile**, e può essere realizzata con valori anche molto alti di *n* (decine, centinai, migliaia), a condizione di scrivere "buoni" programmi paralleli,
2) **l'architettura multithreaded ha senso solo per bassi valori di *n***, essendo basata su una architettura con singola CPU.

La ricerca e l'industria stanno verificando la validità e realizzabilità di soluzioni multicore con elevato numero di CPU, necessariamente caratterizzate da architettura ILP più semplice, scalare o superscalare

con limitatissimo numero di vie, fondamentalmente *in-order*. L'utilizzazione della modalità multithreading, all'interno della stessa CPU in tali architetture ad alto grado di parallelismo, sarà oggetto di ricerca e valutazione. Attualmente, diversi prodotti multicore hanno questa caratterizzazione, con ogni core capace di supportare un limitato numero di thread (2 – 4).

4.1.2 Multithreading *vs* thread singolo

Riprendiamo il semplice esempio della sez. 4.1.1:

P_1::	P_2::
1. MUL Ra1, Rb1, Rx1	a. LOAD RA2, Rj2, Rp2
2. DIV Rc1, Rd1, Ry1	b. LOAD RB2, Rj2, Rq2
3. ADD Rx1, Ry1, Rz1	c. SUB Rp2, Rq2, Rr2
4. STORE RC1, Ri1, Rz1	d. INCR Rr2
	e. STORE RD2, Rj2, Rr2

Abbiamo visto che la CPU superscalare a 2 vie, operante in multithreading, era caratterizzata da:

$$T = t \qquad T_c \sim 9\,t$$

Se, sulla stessa architettura superscalare a 2 vie, eseguiamo la computazione come processo (*thread) singolo*:

1-2. MUL Ra1, Rb1, Rx1 | DIV Rc1, Rd1, Ry1

3-x. ADD Rx1, Ry1, Rz1 | NOP

4-x. STORE RC1, Ri1, Rz1 | NOP

a-b. LOAD RA2, Rj2, Rp2 | LOAD RB2, Rj2, Rq2

c-x. SUB Rp2, Rq2, Rr2 | NOP

d-x. INCR Rr2 | NOP

e-x. STORE RD2, Rj2, Rr2 | NOP

Otteniamo:

$$T = 14\,t/9 \qquad T_c \sim 14\,t$$

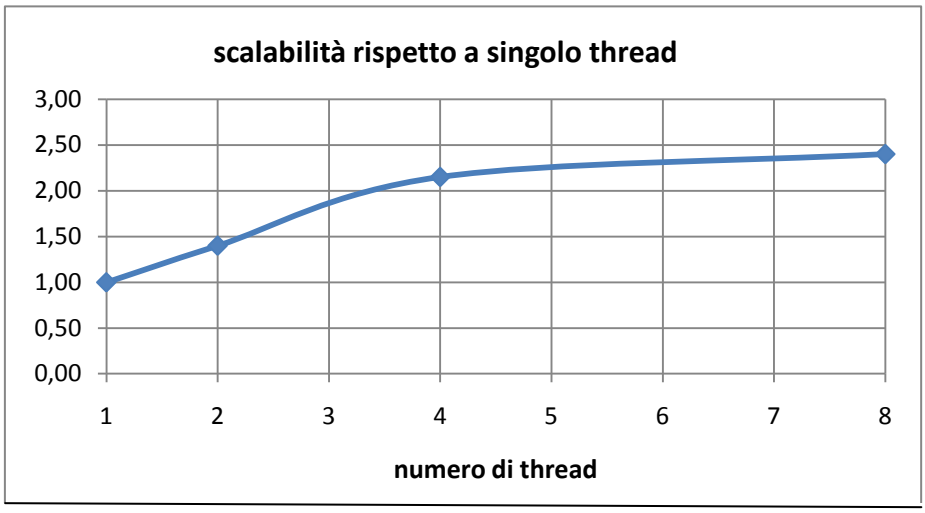
con una scalabilità della versione con due thread rispetto a quella a thread singolo:

$$s_{MT} = 14/9 = 1{,}6$$

L'esempio mostra ulteriormente il punto di vista del progettista di CPU ILP: le due caratteristiche interrelate del multithreading (aumentare il grado di parallelismo effettivo e mascherare le latenze), permettano di sfruttare meglio l'architettura superscalare.

Come si vede, pur trattandosi di un caso particolarmente limitato come singolo thread, il guadagno della modalità multithreading non è ugualmente del 100% (in questo caso 60%). In effetti, alcuni benchmark iniziali per l'architettura Intel Xeon con Hyperthreading (SMT) dichiaravano un miglioramento fino al 65% rispetto all'architettura Xeon della generazione precedente.

In realtà, la figura seguente mostra i risultati dell'applicazione della **suite di benchmark standard Spec95 e Splash2** in versione multiprogrammazione e calcolo parallelo, da cui si evince che usare una CPU con modalità multithreading rispetto alla modalità a singolo thread porta mediamente ad una scalabilità variabile all'incirca nell'intervallo 1,4 – 2,4 con un numero di thread variabile da 2 a 8:



Vale la pena di ribadire il modo di interpretare questa valutazione: si tratta di confrontare le prestazioni ottenute sulla *stessa* CPU usata nelle due modalità, a singolo thread o multithreaded (o riprogettata per funzionare anche con modalità multithreading): se l'architettura è superscalare a *n* vie, il rapporto è calcolato tra

- il tempo di servizio (o di completamento) di una computazione eseguita con singolo thread, ogni istruzione del quale è superscalare a *n* vie,
- e il corrispondente tempo della stessa computazione eseguita con un numero di thread variabile da 1 a *n*, quindi con ogni istruzione superscalare a *n* vie che può contenere una o più istruzioni scalari appartenenti ad ogni thread.

Il caso preso come punto di riferimento nei benchmark è quello in cui il numero di thread è *n* e ogni istruzione superscalare a *n* vie contiene *n* istruzioni scalari di thread distinti.

4.2 Architetture multithreading

4.2.1 Thread supportati direttamente dall'architettura firmware

Come detto, l'idea dell'approccio multithreading è quella di *sfruttare meglio le risorse di una architettura ILP per eseguire concorrentemente istruzioni appartenenti ad attività computazionali distinte*: in pratica, programmi distinti, genericamente intesi.

Tali attività vengono normalmente riferite con il termine **thread***:* questo termine viene ora usato in una accezione diversa rispetto alla nozione tradizionalmente usata per indicare multiprogrammazione supporta dal sistema operativo e/o attività concorrenti definite nello stesso spazio di indirizzamento.

Un thread va ora visto come una entità computazionale concorrente **supportata direttamente dall'architettura firmware**; talvolta viene usato il termine "hardware thread". A tale scopo, una architettura multithreaded, capace di eseguire concorrentemente fino a *m* thread, deve prevedere *a livello firmware*:

1. *m* **contesti** *indipendenti*, dove un contesto è, come di regola, rappresentato dal contatore istruzione e dai registri visibili a livello assembler,
2. un meccanismo di **tagging**, per distinguere univocamente istruzioni appartenenti a thread distinti che, durante la loro esecuzione, utilizzino contemporaneamente risorse condivise nella struttura pipeline,
3. un meccanismo che attivi la commutazione di contesto a livello di thread, o **thread switching**.

I contesti indipendenti potrebbero essere emulati sopra un unico insieme di registri, ma, in pratica, essi sono quasi sempre implementati mediante *insiemi multipli di registri*: nei nostri esempi, *m* copie dell'array RG e dell'array RF, oltre a *m* copie del contatore istruzioni IC.

Dal punto di vista del programmatore, un thread, come qui inteso, può essere definito/dichiarato come usualmente nello sviluppo di applicazioni. Un thread può, ad esempio, essere

- un programma o un processo applicativo,
- un thread di utente o di sistema (ad esempio, un Posix thread).

Inoltre un thread può anche essere una attività concorrente non esplicitamente definita dal programmatore o dai servizi di sistema, ad esempio

- una attività concorrente riconosciuta e generata automaticamente dal compilatore di un processo sequenziale (thread subordinati, microthread, nanothread, ecc),
- una attività generata direttamente a tempo di esecuzione.

Quello che caratterizza i thread, come qui intesi, è *il supporto a tempo di esecuzione*, che è sostanzialmente diverso da quello di un normale programma o processo o thread "tradizionale", in quanto tale supporto è interamente realizzato a livello firmware. Di conseguenza, il *compilatore* per una macchina multithreaded deve produrre codice in modo tale da rendere possibile l'aggancio a tale tipo di supporto.

Il supporto firmware dei thread è appunto basato sulle caratteristiche 1, 2, 3 sopra menzionate.

4.2.2 Thread switching

Per quanto riguarda la commutazione di contesto, distinguiamo tra process switching e thread switching:

a) se un thread si sospende per *cause logiche*, inerenti la semantica degli eventuali meccanismi di concorrenza usati (ad esempio, sospensione di una *receive* in condizione di "canale vuoto"), il meccanismo utilizzato è comunque sempre quello del **process switching**, consistente nel salvataggio/ripristino dello stato della computazione (registri, IC) in strutture dati in memoria condivisa, ed altre attività di gestione (Cap. VI sez. 2.1, Cap. V sez. 4.5.c). Questo è vero per qualunque thread, "tradizionale" o "hardware";

b) il meccanismo di **thread switching** è invece applicato non per ragioni che logicamente bloccano una attività concorrente, bensì solo con lo scopo di *ottimizzare l'esecuzione di computazioni su CPU ILP* sfruttandone meglio l'architettura. Il thread switching comporta il passaggio dall'esecuzione di istruzioni di un thread all'esecuzione di istruzioni di un altro thread **attivo**, cioè *il cui contesto sia già presente nei registri del processore*. Ciò comporta un overhead molto basso, di uno o pochissimi cicli di clock, talvolta effettivamente zero. Il thread switching può avvenire sostanzialmente secondo tre modi diversi di attuare la caratteristica 3 sopra menzionata:

   b1) secondo un meccanismo automatico di alternanza tra istruzioni di thread distinti,

   b2) in seguito ad un evento caratterizzato da "alta latenza", come il verificarsi di una dipendenza logica o l'accesso a memorie esterne,

   b3) in modo invisibile in quanto più istruzioni appartenenti a thread diversi fanno parte di una stessa istruzione lunga superscalare.

   Secondo la classificazione dei modelli di multithreading che introdurremo nella prossima sezione, il caso *b1* si ha nel modello detto **interleaved multithreading** (IMT), il caso *b2* nel modello **blocked multithreading** (BMT), il caso *b3* nel modello **simultaneous multithreading** (SMT). *Tutti gli esempi finora studiati usavano il modello SMT.*

4.2.3 Emissione singola ed emissione multipla

Con queste premesse, l'obiettivo di una architettura ILP multithreaded è di eseguire concorrentemente istruzioni appartenenti a thread distinti.

Tale concorrenza può manifestarsi in due modi. Usiamo il termine *emissione* (*issuing*) di istruzioni per indicare la fase di generazione di **un** elemento dello stream (scalare o superscalare) da parte di IM e sua conseguente preparazione in IU. La classificazione distingue:

I. architetture con **emissione singola,** cioè *emissione di istruzioni appartenenti ad un singolo thread*,

II. architetture con **emissione multipla,** cioè *emissione contemporanea di istruzioni appartenenti a più thread*, o **simultaneous multithreading** (SMT).

Come detto, *gli esempi presentati finora hanno utilizzato una CPU con SMT.*

Nella classe con emissione singola viene fatta l'ulteriore distinzione tra modello **interleaving** (IMT) e modello **blocking** (BMT).

Una macchina a emissione singola può utilizzare

- tanto una **architettura pipeline scalare**,
- quanto una **architettura superscalare**,

mentre una a emissione multipla può utilizzare necessariamente solo

- una **architettura superscalare**.

Per procedere per gradi, e non creare confusione tra multithreading e architettura superscalare,

- inizieremo a considerare il caso di *emissione singola* su architettura pipeline *scalare*,
- quindi passeremo al caso di *emissione singola* su architettura *superscalare*,
- infine studieremo il caso di *emissione multipla* su architettura superscalare.

In tutti i casi studiati, un requisito fondamentale è che *le computazioni concorrenti siano costituite da più thread definiti nello stesso spazio di indirizzamento*. Ciò è necessario in quanto la cache istruzioni primaria deve generare un unico stream di istruzioni (scalari o lunghe).

4.3 Multithreading con emissione singola in architettura pipeline scalare

Consideriamo una variante dell'architettura pipeline scalare, consistente nell'implementazione delle caratteristiche 1, 2, 3 discusse nella sezione precedente:

1. *contesti multipli*,
2. *meccanismo di tagging*,
3. *meccanismo di thread switching*.

Nell'architettura pipeline, avere contesti multipli significa avere più copie del contatore istruzioni (IC) e dei registri generali (RG) e floating (RF). Con l'architettura D-RISC,

- *m* copie dei registri sono presenti sia in EU, o meglio EU_Master (copie di RF e copie principali di RG), che in IU (copie secondarie di RG), con associati altrettanti semafori di sincronizzazione,
- *m* copie di IC sono presenti in IU (copie principali) e in IM/MMU_I (copie secondarie).

Per il momento non sono necessarie repliche delle unità, ma è sufficiente che IU, EU_Master e IM/MMU_I siano *singole unità* che contengano le *m* copie dei registri menzionati.

Come primo esempio, senza perdere in generalità, consideriamo i seguenti due thread computazionalmente identici:

Thread 1		**Thread 2**	
1.1	MUL Ra1, Rb1, Rc1	2.1	MUL Ra2, Rb2, Rc2
1.2	INCR Rc1	2.2	INCR Rc2
1.3	STORE RC1, Ri1, Rc1	2.3	STORE RC2, Ri2, Rc2

Per quanto non generale, un esempio con thread identici è significativo e realistico, in quanto presente in molti programmi paralleli che usino i paradigmi *farm* e *data parallel*.

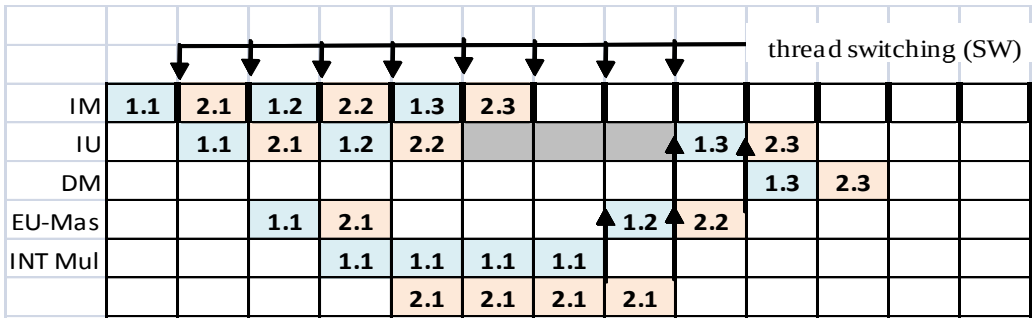
Come si vede, ognuno dei due thread usa i suoi registri ed è caratterizzato da un proprio identificatore unico. Supponiamo che i contesti dei due thread (IC1, Ra1, Rb1, Rc1, RC1), (IC2, Ra2, Rb2, Rc2, RC2) siano stati allocati e caricati in insiemi di registri disgiunti.

Come introdotto, nelle macchine a emissione singola il meccanismo del thread switching si attua secondo due modelli:

1. *interleaved multithreading*,
2. *blocked multithreading*.

4.3.1 Interleaved multithreading in architettura scalare

Un modo di eseguire i thread concorrentemente è l'*interleaving*: *ad ogni slot (t*) viene iniziata una istruzione di un diverso thread (la tecnica è anche detta *fine-grain multithreading*). Nel nostro esempio con due thread, il funzionamento è il seguente:



IM genera uno stream di istruzioni in cui sono alternate le istruzioni dei thread *attivi*. Ogni istruzione è accompagnata da un *tag* (caratteristica 2), che può essere semplicemente l'identificatore unico intero del thread (inoltre, l'istruzione è accompagnata dal valore del contatore istruzioni locale per i controlli di validità nella IU).

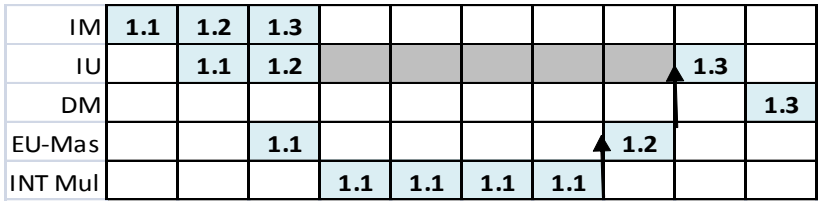
IU decodifica, esegue o inoltra ogni istruzione utilizzando i registri del contesto del thread identificato.

Tutte le comunicazioni tra unità contengono il *tag* del thread a cui si riferiscono. EU usa i *tag* per leggere e scrivere i registri del rispettivo contesto, e per associare istruzioni di LOAD inviate da IU a dati inviati da DM.

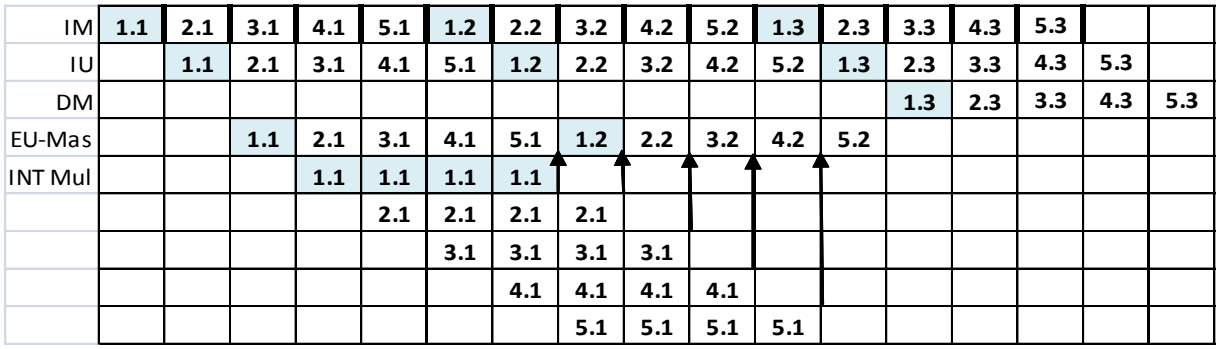
Oltre che utilizzare i propri contesti, le istruzioni dei vari thread in esecuzione *condividono* tutte le risorse architetturali. Alcune sono utilizzate alternativamente (ad esempio, IU), altre sono usate contemporaneamente, tipicamente l'Unità Esecutiva in pipeline.

Il meccanismo di *thread switching* è semplice nel modello interleaving: IM alterna la lettura di istruzioni di thread attivi secondo un ordine fisso. L'overhead è zero cicli di clock. Sulla base di questa considerazione, si riprenda il confronto tra thread e thread switching, da una parte, e processo (o thread tradizionale) e process switching, dall'altra: se le stesse due computazioni fossero state definite come processi (o thread tradizionali), il comportamento e la scala temporale sarebbero stati molto diversi.

Rispetto all'esecuzione a singolo thread:



nello stesso intervallo di tempo, in cui viene elaborato un thread, viene elaborato anche (quasi tutto) l'altro thread. In particolare, quello che interessa è che la *bolla* (ampia 5 *t*) del primo thread è stata in parte coperta da istruzioni del secondo thread, quindi le risorse sono state usate più efficientemente. Nell'esempio, se invece che 2 i thread fossero stati 5, nessuna bolla si sarebbe avuta nel funzionamento della IU:



Il numero medio ottimale *m* di thread è stimabile come l'ampiezza media della bolla (talvolta indicato con "opportunity cost" in letteratura). Trascurando le degradazioni dovute ai salti, se *N* è il numero medio di istruzioni eseguite, e Δ è il ritardo medio stimato dal modello dei costi:

$$m = \frac{N\ \Delta}{t}$$

dove le medie sono valutate sul complesso di tutti i thread attivi. In altri termini, il guadagno introdotto dal modello di multithreading a emissione singola è tanto più significativo quanto maggiori sono le degradazioni del singolo thread, fino ad annullarsi nel caso Δ = 0. Questo per confermare come la concorrenza tra thread a emissione singola sia tanto più efficace quanto minore è l'efficienza relativa dell'esecuzione del singolo thread sulla stessa architettura.

4.3.2 Meccanismi per il supporto dei thread

Diversi aspetti ora visti per il modello interleaving si applicano a qualunque modello, come le comunicazioni con *tag* e la condivisione di risorse tra thread contemporaneamente in esecuzione.

I meccanismi che, in dettaglio, occorrono per l'implementazione del supporto dei thread sono l'analogo di quelli noti per lo scheduling a basso livello dei processi, ma relativi agli eventi di interesse e realizzati direttamente a livello firmware.

I thread attivi sono caratterizzati dagli stati di avanzamento

- *in esecuzione*: una istruzione è chiamata in IM, e quindi decodificata e analizzata per l'abilitazione in IU (le fasi comuni a qualunque pipeline di elaborazione);
- *pronto*: una istruzione attende solo di essere chiamata da IM e decodificata e analizzata in IU;
- *in attesa*: una istruzione si è bloccata in IU a causa di una dipendenza logica (o altro evento ad alta latenza).

IU dispone di una *lista di thread pronti* (realizzata come vettore circolare di dimensione fissa e uguale a *m*) dove ogni posizione è caratterizzata dall'identificatore del thread e dall'indirizzo di ripristino. Questa lista è presente anche in IM (MMU_I) e viene tenuta consistente similmente a quanto accade con il contatore istruzioni in seguito a salto.

IU tiene traccia dell'eventuale *thread in attesa* (dei thread in attesa) e dei registri dei quali tale thread attende l'aggiornamento; allo scopo mantiene un *descrittore* delle istruzioni in attesa, costituto da:

(tag, indirizzo dell'istruzione, codifica dell'istruzione,

contenuto di registro 1, contenuto di registro 2, contenuto di registro 3 oppure offset di salto).

Nel modello interleaving, in uno stesso ciclo di clock MMU_I

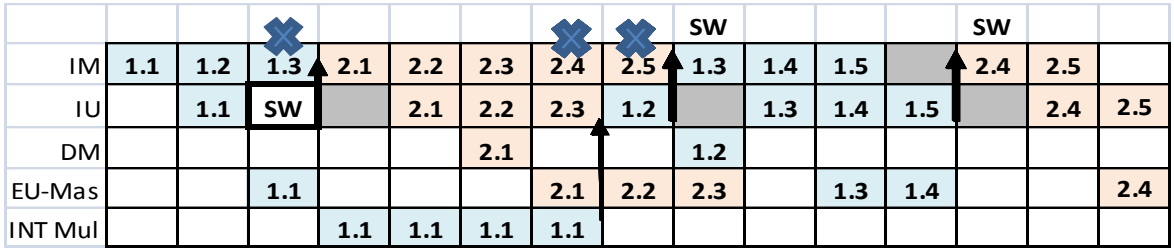
- estrae il descrittore *(tag, indir_istruzione)* del primo thread dalla testa della lista pronti,
- inserisce in fondo alla stessa lista pronti il descrittore modificato *(tag, indir_istruzione + 1)*,
- invia alla cache istruzioni la coppia *(tag, μ (indir_istruzione))*, dove μ è la funzione di traduzione degli indirizzi del thread in esecuzione.

4.3.3 Blocked multithreading in architettura scalare

In questo modello (chiamato anche *corse-grain multithreading*) non c'è un ordinamento fisso nell'esecuzione delle istruzioni dei vari thread, ma il thread switching avviene in seguito al verificarsi di un evento che blocca l'elaborazione dell'istruzione di un thread nell'Unità Istruzioni, in particolare una dipendenza logica non risolta.

Consideriamo l'evoluzione dei due seguenti thread:

Thread 1	**Thread 1**
1.1 MUL Ra1, Rb1, Rc1	2.1 LOAD RA2. Ri2, Ra2
1.2 STORE Rx1, Ry1, Rc1	2.2 ADD Ra2, Rb2, Rc2
1.3 INCR Rc1	2.3 SUB Rc2, Rd2, Re2
1.4 ADD Rc1, Rd1, Rz1	2.4 INCR Re2
1.5. END	2.5 …



Inizialmente il thread 1 passa in esecuzione, mentre il thread 2 rimane in stato di pronto.

IM continua a generare uno stream di istruzioni appartenenti al thread 1 (istruzioni 1.1, 1.2), finché IU non incontra una dipendenza logica non risolta (istruzione 1.2): in tal caso, invece di bloccarsi o di tentare di portare avanti istruzioni fuori ordine dello stesso thread, IU chiede a IM di iniziare a generare lo stream con istruzioni del primo thread in lista pronti (nell'esempio il thread 2, istruzione 2.1). Il meccanismo di thread switching è quindi concettualmente un *salto*, e come tale il suo overhead è uguale a $t = 2\tau$, tempo speso in IU per la rilevazione della dipendenza e per la comunicazione a IM. Inoltre, viene introdotta una bolla in quanto l'istruzione 1.3 è stata letta a vuoto.

Ad un certo punto, durante l'elaborazione del thread 2, si verifica la condizione che ha bloccato il thread 1 (il registro Rc1 è stato aggiornato dalla EU): questo provoca immediatamente la preparazione in IU dell'istruzione sbloccata (1.2) ed un nuovo thread switching, che riporta in esecuzione il thread 1 (il cui contesto è presente nei registri e questi sono aggiornati). Il meccanismo di thread switching consiste nel richiedere a IM di ri-iniziare a generare lo stream dall'istruzione il cui indirizzo (1.3) è quello successivo all'istruzione che si è sbloccata (1.2). Inoltre, nella stessa comunicazione IU informa IM di rimettere il thread 2 in stato di pronto all'indirizzo successivo (2.4) all'ultima istruzione del thread 2 eseguita (2.3). In questo caso, il thread switching di per sé non comporta overhead, ma viene comunque introdotta una bolla a causa di istruzioni lette a vuoto da IM (2.4, 2.5).

Quando l'elaborazione del thread 1 arriva all'istruzione 1.5 (END), il thread termina, e ciò provoca un thread switching, realizzato, analogamente al caso precedente, mediante una comunicazione a IM di riprendere a generare lo stream di istruzioni del primo thread in lista pronti (ancora il thread 2 nell'esempio, ma in generale sarebbe stato un diverso thread). Anche questo evento di per sé non comporta overhead, ma naturalmente l'evento introduce una bolla nel pipeline.

Come si vede, gran parte della bolla del thread sospeso è mascherata da istruzioni del thread che gli subentra in esecuzione, anche se vengono introdotte altre bolle, in questo caso più piccole, per la gestione del thread switching, più complicata rispetto al modello interleaving.

Rispetto al modello interleaving, il modello blocking

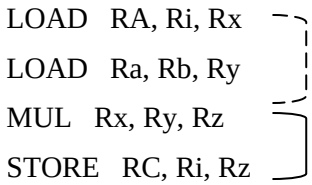
- richiede *un minor numero di thread* per il mascheramento delle latenze,
- al prezzo di un certo *overhead* complessivo per il thread switching (slot speso appositamente in IU e/o introduzione di bolle, analogamente alle istruzioni di salto).

Il modello blocking è più flessibile: un singolo thread sfrutta a pieno la CPU finché non si verifica un thread switching.

Un altro caso significativo di latenze mascherate con il modello blocked multithreading è quello di **accessi in memoria esterna** (in architetture uniprocessor, e ancor più nei multiprocessor).

Se la cache dati genera un fault, di per sé questo evento non causa degradazione di prestazioni, in quanto il trasferimento del blocco può avvenire in parallelo al prosieguo dell'elaborazione nel pipeline. L'evento si ripercuote sulla prestazioni se, e quando, il trasferimento del blocco contribuisce a indurre una dipendenza logica IU-EU.

Ad esempio nel programma:



se la prima LOAD genera un fault, la seconda LOAD (supponiamo che non generi fault) viene subito servita da DM, che ha immediatamente lanciato la richiesta esterna di trasferimento blocco. La MUL viene inoltrata da IU a EU_Master; tale istruzione però non è abilitata finché non si conclude la prima LOAD con il trasferimento del blocco. La STORE viene bloccata da una dipendenza logica indotta dalla MUL; quindi la latenza è non solo quella della MUL, ma soprattutto è determinata dalla latenza di trasferimento del blocco della prima LOAD. L'evento che causa il thread switching è il blocco della IU in seguito a questa dipendenza logica.

Finora abbiamo supposto che gli eventi che provocano il thread switching siano dipendenze logiche, cioè si ha il così detto *thread switching dinamico*. Poiché nel caso di bolla "piccola" l'overhead è paragonabile alla bolla stessa, sono state studiate delle ottimizzazioni per effettuare il thread switching *solo sotto determinate condizioni*: ad esempio in caso di fault di cache (*dynamic switch on cache miss*); oppure tenendo conto della storia delle istruzioni più recenti per capire se, in presenza di dipendenza logica, la bolla avrebbe una ampiezza significativa (*dynamic switch on condition*).

In alternativa, esiste la modalità di funzionamento con *thread switching statico*, nella quale

- è prevista una *esplicita istruzione* che provoca thread switching (thread switching *statico esplicito*): l'overhead causato dall'esecuzione di questa istruzione addizionale è in parte recuperato dal fatto che il thread switching può essere implementato direttamente in IM,
- oppure, in certe macchine, alcune classi di istruzioni – load, store, branch – provocano sempre thread switching (thread switching *statico implicito*).

4.4 Multithreading con emissione singola in architettura superscalare

I due modelli del caso I, interleaving e blocking, si applicano anche ad architetture superscalari.

Vediamo un esempio di **interleaved multithreading** su architettura superscalare base a 2 vie, per una computazione costituita da due thread identici:

Thread 1:

1.1-2 LOAD RA1, Ri1, Ra1 | LOAD RB1, Ri1, Rb1

1.3-4 MUL Ra1, Rb1, Rc1 | INCR Rs1

1.5-6 STORE RC1, Ri1, Rc1 | INCR Ri1

Thread 2:

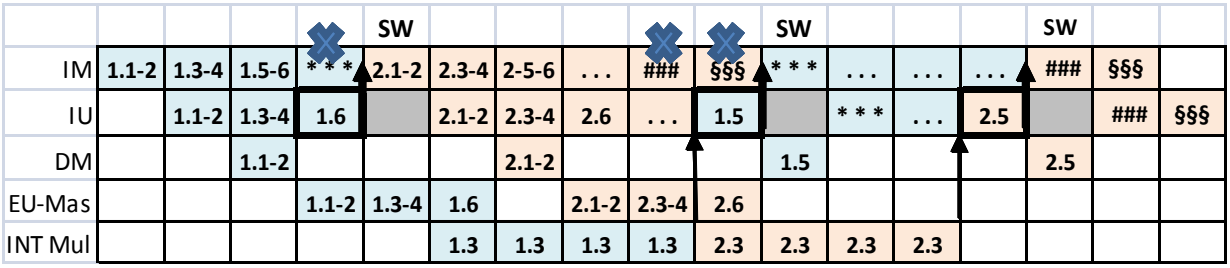
2.1-2 LOAD RA2, Ri2, Ra2 | LOAD RB2, Ri2, Rb2

2.3-4 MUL Ra2, Rb2, Rc2 | INCR Rs2

2.5-6 STORE RC2, Ri2, Rc2 | INCR Ri2

IM	1.1-2	2-1-2	1.3-4	2.3-4	1.5-6	2.5-6							
IU		1.1-2	2-1-2	1.3-4	2.3-4	1.6	2.6				1.5	2.5	
DM			1.1-2	2-1-2								1.5	2.5
EU-Mas				1.1-2	2-1-2	1.3-4	2.3-4	1.6	2.6				
INT Mul							1.3	1.3	1.3	1.3			
								2.3	2.3	2.3	2.3		

La stessa computazione per il modello **blocked multithreading** su architettura superscalare base a 2 vie:



Valgono le considerazioni fatte nella sezione precedenze.

Inoltre, è da notare che l'architettura superscalare permette di *ridurre l'overhead del thread switching nel modello blocking*; ad esempio, IU, nell'inoltrare l'istruzione 6, in parallelo effettua anche il thread switching in seguito al blocco dell'istruzione 5.

Dal punto di vista dell'architettura, l'eventuale replicazione di IU ed EU_Master è una soluzione per l'esecuzione superscalare, indipendentemente dal fatto di avere multithreading, in quanto istante per istante *IM, IUe EU_Master operano su istruzioni di uno stesso thread* (si può avere parallelismo effettivo tra thread distinti solo in DM e nelle Unità Funzionali di EU).

4.5 Simultaneous Multithreading: emissione multipla in architettura superscalare

In questo modello architetturale (SMT) si ha *emissione multipla*, in una architettura *superscalare* multithreading, di istruzioni appartenenti a più thread.

Ogni elemento dello stream contiene istruzioni appartenenti a thread distinti; questo concorre ad attuare il vicolo che le istruzioni di una istruzione lunga, o comunque emesse contemporaneamente, siano indipendenti. In generale, comunque, se l'architettura è superscalare a n vie, le n istruzioni di una stessa istruzione lunga possono appartenere ad un numero m di thread

$$1 \leq m \leq n$$

ed ogni istruzione avere un suo valore di m.

La chiave dell'utilizzazione del modello SMT è *l'opportuna composizione delle istruzioni lunghe*, con l'obiettivo di avere sia *parallelismo reale tra thread*, che *mascheramento delle latenze*. In questo, sono possibili tutti i casi di combinazione,

- da quello, in cui in una istruzione lunga n, figurano n istruzioni di uno stesso thread, come nel caso superscalare senza multithreading,
- fino a quello in cui le n istruzioni appartengono a thread distinti,
- con tutte le possibilità intermedie.

Il riconoscimento delle istruzioni emesse contemporaneamente può essere effettuato *staticamente* (istruzione lunga nel vero senso della parola) o *dinamicamente*. Nel secondo caso, l'interprete firmware di IM-IU può decidere quali istruzioni emettere contemporaneamente ad ogni passo, anche sulla base di eventi non prevedibili staticamente.

Consideriamo ancora semplici esempi sul tipo delle sezioni precedenti.

Iniziamo dal caso immediato di $m = n$ in tutte le istruzioni lunghe:

Thread 1:

1.1 MUL RA1, Ri1, Rx1

1.2 INCR Rx1

1.3 STORE RB1, Ri1, Rx1

Thread 2:

2.1 MUL RA2, Ri2, Rx2

2.2 INCR Rx2

2.3 STORE RB2, Ri2, Rx2

Con una codifica superscalare per architettura a 2 vie:

1.1-2.1 MUL RA1, Ri1, Rx1 | MUL RA2, Ri2, Rx2

1.2-2.2 INCR Rx1 | INCR Rx2

1.3-2.3 STORE RB1, Ri1, Rx1 | STORE RB2, Ri2, Rx2

l'esecuzione è la seguente:

IM	**1.1-2.1**	**1.2-2.2**	**1.3-2.3**							
IU		**1.1-2.1**	**1.2-2.2**						**1.3-2.3**	
DM										
EU-Mas			**1.1-2.1**					**1.2-2.2**		**1.3-2.3**
INT Mul				**1.1**	**1.1**	**1.1**	**1.1**			
				2.1	**2.1**	**2.1**	**2.1**			

In pratica, i due thread evolvono in parallelo sincrono a livello di ciclo di clock. Il tempo di servizio per istruzione della *computazione costituita dall'insieme dei due thread* è la metà di quello che si avrebbe eseguendo i due thread sequenzialmente in una architettura non multithreaded.

L'esempio è servito a introdurre il modello SMT, ma ne ha mostrato solo una parte delle potenzialità. Ad esempio, non si è avuto vantaggio, in termini di tempo di servizio, rispetto alla stessa computazione eseguita su architettura scalare, a singola emissione, blocked:

IM	**1.1**	**1.2**	**1.3**		**2.1**	**2.2**	**2.3**							
IU		**1.1**	**1.2**	**SW**		**2.1**	**2.2**		**1.3**				**2.3**	
DM										**1.3**				**2.3**
EU-Mas			**1.1**				**2.1**	**1.2**				**2.2**		
INT Mul				**1.1**	**1.1**	**1.1**	**1.1**	**2.1**	**2.1**	**2.1**	**2.1**			

L'architettura SMT dovrebbe essere sfruttata anche per il mascheramento delle latenze. Ad esempio:

Thread 1:

1.1 LOAD RA1, Ri1, Rc1

1.2 STORE RC1, Ri1, Rc1

Thread 2:

2.1 LOAD RX2, Ri2, Rx2

2.2 ADD Rx2, Ry2, Rz2

2.3 INCR Ri2

2.4 INCR Rv2

2.5 MUL Rz2, …, …

Su una architettura superscalare a 2 vie, si possono generare le istruzioni lunghe:

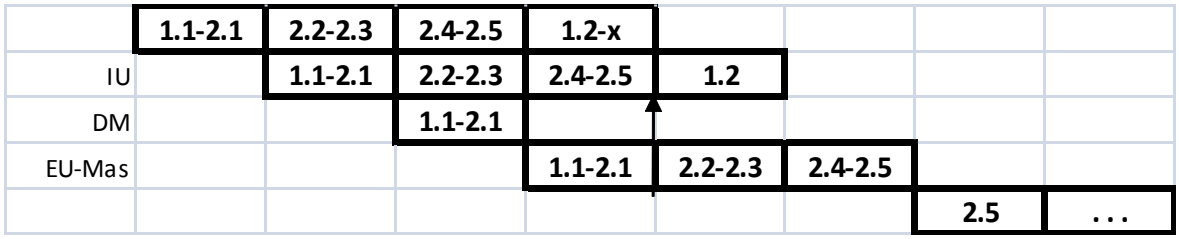
1.1-2.1 LOAD RA1, Ri1, Rc1 | LOAD RX2, Ri2, Rx2

2.2-2.3 ADD Rx2, Ry2, Rz2 | INCR Ri2

2.4-2.5 INCR Rz2 | STORE …

1.2-x. STORE RC1, Ri1, Rc1 | NOP

eseguite nel seguente modo:



La bolla del thread 1 è stata annullata da altre istruzioni lunghe (nell'esempio interamente del thread 2, ma potevano anche essere usate istruzioni di ulteriori thread).

Rispetto all'esecuzione su una architettura a emissione singola, blocked, superscalare a 2 vie,

Thread 1:

1.1-x LOAD RA1, Ri1, Rc1 | NOP

1.2-x STORE RC1, Ri1, Rc1 | NOP

Thread 2:

2.1-2.2 LOAD RX2, Ri2, Rx2 | ADD Rx2, Ry2, Rz2

2.3-2.4 INCR Ri2 | INCR Rv2

2.5-x MUL Rz2, … | NOP

si ottiene un certo guadagno dovuto al fatto che, nel caso a emissione singola, mediamente viene introdotto un *maggior numero di bolle statiche* NOP (nell'esempio tre contro una) e un *maggior overhead di thread switching* (nell'esempio, al verificarsi della dipendenza logica indotta da 1.1 su 1.2).

In termini di rapporto prestazioni/costo, è stato dimostrato che, rispetto alla versione superscalare single-threaded, l'aumento di area è abbastanza contenuto per architetture con 2 contesti.

Circa la modalità di realizzazione di IU ed EU_Master in modo monolitico o con replicazione, ora è ragionevole pensare a *soluzioni replicate dove il numero di repliche sia uguale al numero di thread* da supportare contemporaneamente, in modo che ogni coppia IU-EU_Master disponga di un proprio contesto in modo completamente indipendente.

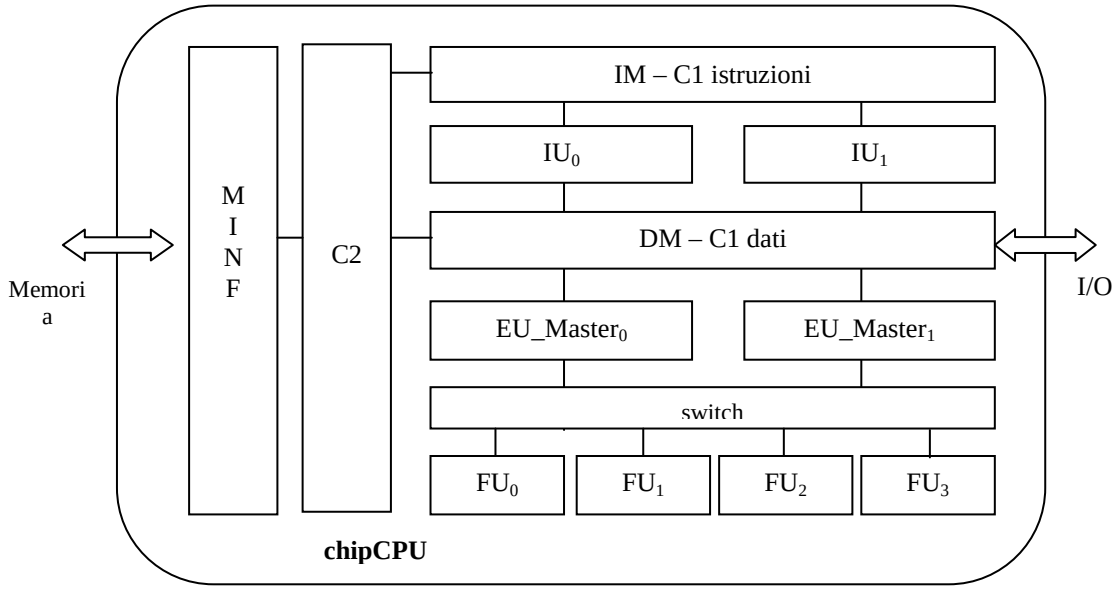
Il vero problema delle architetture SMT è lo **scheduling** dei thread a livello di ciclo di clock: per essere sfruttata adeguatamente, questa architettura richiede che vengano individuati e/o progettati thread *opportuni*, tali da poter dar luogo alla composizione di istruzioni lunghe con elevato parallelismo tra thread e con mascheramento delle latenze. La difficoltà dipende dalla capacità di definire un buon numero di attività concorrenti

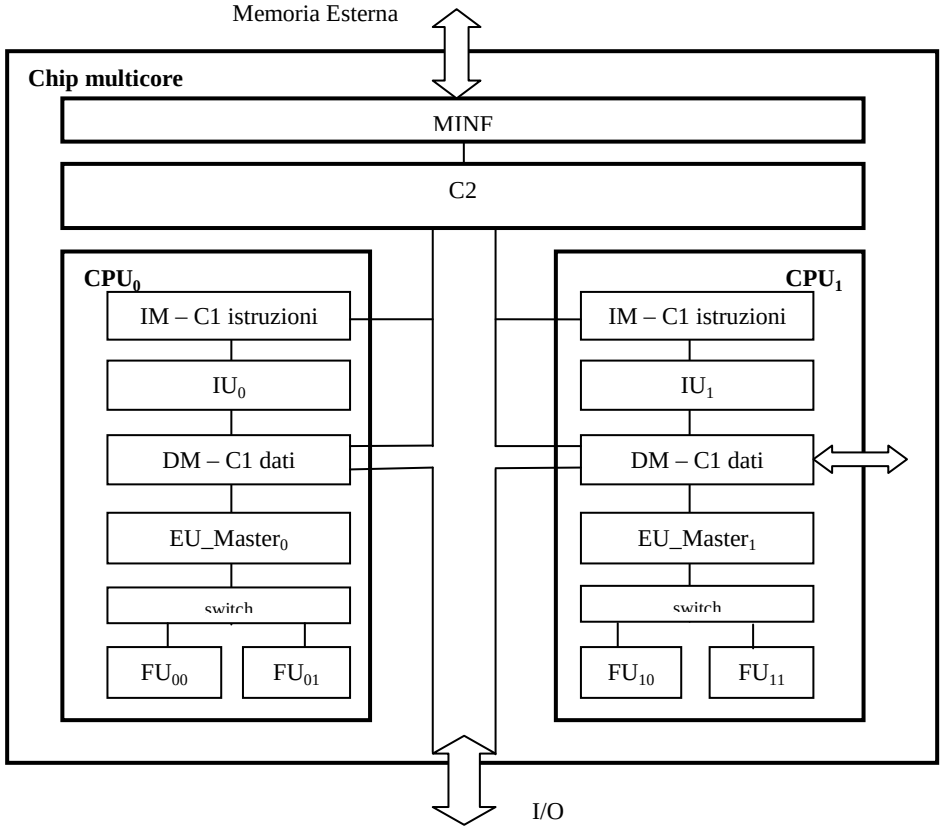
1. come appartenenti allo *stesso spazio di indirizzamento*,
2. con elevato grado di *indipendenza a livello di grana grossa*, per il mascheramento delle latenze,
3. con elevato grado di *indipendenza a livello di grana fine*, per riempire adeguatamente le istruzioni lunghe,
4. e, con riferimento al punto 3, con *ordinamento tra istruzioni scalari di ogni thread tale da dare luogo a composizioni con basso numero di bolle statiche*.

Il problema è difficile, tanto che venga affrontato con soluzioni *statiche* quanto *dinamiche*, a meno che non venga caratterizzato da *vincoli* applicabili con successo a casi che ricorrono spesso nella progettazione di applicazioni. Due esempi verranno riportati nelle sezione seguente.

4.6 Da multithreading a multicore

Le due figure seguenti mostrano rispettivamente un possibile schema di architettura di CPU multithreaded con replicazione di IU e EU_Master in due copie identiche, e la struttura di un chip *dual-core* realizzato con la stessa tecnologia.





L' architettura delle prima figura può supportare il modello SMT con *due thread*, dove i due contesti sono all'interno delle coppie IU-EU_Master, mentre le altre unità sono condivise dai thread in esecuzione.

È stata evidenziata, all'interno del chip, la *cache secondaria*, condivisa essa stessa, facente capo all'interfaccia di memoria esterna.

La seconda figura mostra una architettura multiprocessor a memoria condivisa, nella quale le due CPU, ognuna *single-threaded*, occupano all'incirca la stessa area del chip precedente, tenendo conto che le Unità Funzionali sono la metà per ogni CPU. Poiché le due CPU sono integrate sullo stesso chip, la cache secondaria può ora essere condivisa.

Nel caso di un numero più elevato di CPU per chip, le strutture di interconnessione sono del tipo indicato in precedenza: crossbar fino a grado di parallelismo medio-basso, strutture di grado limitato (mesh, alberi) per grado di parallelismo superiore.

Una volta ribadito che l'approccio CMP è comunque orientato al parallelismo elevato, mentre SMT è limitato al numero di vie di una COU superscalare, riassumiamo le potenzialità di ognuno dei due approcci:

CMP:

- non è necessario che la computazione parallela sia costituita da thread definiti nello stesso spazio di indirizzamento. Questo offre molta più flessibilità nella definizione di applicazioni, e anche nel riuso di applicazioni esistenti;
- viene supportata l'esecuzione di *programmi paralleli* nel vero senso della parola, non semplicemente una collezione di attività indipendenti. La programmazione parallela ha ormai proprie solide basi (Cap. X) e lo sviluppo di strumenti per permettere la progettazione di applicazioni parallele in modo relativamente facile e portabile è un "must" alla luce dell'evoluzione tecnologica;
- il *grado di parallelismo* può essere anche molto alto (pur se gli attuali prodotti commerciali sono ancora limitati), ed è previsto che l'evoluzione tecnologica segua ancora la legge di Moore, ma d'ora in poi applicata al numero di core, invece che al ciclo di clock;

SMT:

- potenziali possibilità di ottimizzazione attraverso *combinazioni dinamiche*, anche sofisticate, delle istruzioni dei thread. L'equivalenza tra funzionamento SMT e funzionamento CMP si ha nel caso semplice in cui le istruzioni dei thread siano combinate in base alla loro posizione e indirizzo nel programma, ma, come visto negli esempi, non è sempre questo il caso delle ottimizzazioni SMT. In altri termini, mentre l'allocazione delle risorse computazionali in un CMP è, in quanto applicata ad una grana molto più grossa, fondamentalmente statica, l'approccio SMT è capace della massima dinamicità che, da un punto di vista teorico, offre maggiori possibilità di ottimizzazione. Come detto, questo potenziale vantaggio è comunque difficile da attuare, in quanto richiede complessi algoritmi di *scheduling* a livello di thread e di singole istruzioni.

Vediamo due esempi di applicazione della tecnica multithreading a programmi eseguiti su multicore.

4.6.1 Esempio 1: parallelismo sui dati

Consideriamo la computazione della somma di vettori in versione con ottimizzazioni statiche, come visto nella sez. 3.2.2, Esempio 5. Il tempo di completamento di una iterazione vale, in assenza di fault di cache:

$$T_{iter\,,scalare} = 7\,t \qquad\qquad T_{iter\,,superscalare\;,2} = 5\,t$$

Consideriamo al versione data-parallel, di tipo *map*, nella quale si prevede un certo numero di moduli esecutori ("worker"), identici come funzione e con partizionamento dei dati: ogni worker esegue lo stesso codice della somma di vettori, ognuno su una partizione distinta degli array A, B, C. Se *M* è l'ampiezza della partizione di ogni array, ogni worker esegue la somma di vettori di dimensione *M* senza interagire con gli altri (worker indipendenti). I worker sono allocati a CPU distinte. Ad esempio, disponendo di un multicore con 8 core, se *N* è la dimensione degli array, ogni esecutore opera su *M = N/8* interi.

Consideriamo il generico worker. Se la CPU è superscalare a 2 vie operante su singolo thread, il suo tempo di completamento è:

$$T_{c,superscalare\ ,2} = M\, T_{iter\ ,superscalare\ ,2} = 5\, M\, t$$

Se la CPU è superscalare a 2 vie operante su 2 thread, possiamo implementare il worker mediante due thread indipendenti e identici, ognuno operante su metà partizione, cioè su *M/2* elementi degli array. Ricordiamo che, se i thread condividono lo stesso spazio di indirizzamento, la cache primaria contiene istruzioni e dati di entrambi i thread.

I due thread possono essere definiti nel modo più semplice: ogni istruzione lunga contiene le istruzioni corrispondenti dei due thread (si sfrutta più il parallelismo tra thread indipendenti che il mascheramento delle latenze). In tal modo, per ognuno dei due thread il tempo di completamento per iterazione vale, come nel caso scalare:

$$T_{iter\ ,SMT,2} = 7\, t$$

Il tempo di completamento dei due thread, eseguiti perfettamente in parallelo, vale quindi:

$$T_{c,SMT,2} = M\ T_{iter\ ,SMT,2} = 7\,\frac{M}{2}t = 3{,}5\, M\, t$$

Quindi, realizzando ogni singolo worker multithreaded si ottiene un guadagno corrispondente alla scalabilità multithreading *vs* superscalare:

$$s_{MT,2} = \frac{T_{c,superscalare\ ,2}}{T_{c,SMT,2}} = 1{,}4$$

In altri termini, utilizzando le CPU superscalari a 2 vie come CPU multithreaded con 2 thread, è come avere aumentato il grado di parallelismo: non del doppio, ma di quasi una volta e mezzo. Con i dati precedenti, disponendo di un chip multicore con 8 core per eseguire il programma parallelo dato, in realtà è come avere

$$8 * s_{MT,2} = 11\ core$$

Questo rappresenta un reale vantaggio se permette di ottenere, o di avvicinare meglio, il grado di parallelismo ottimo della computazione.

4.6.2 Esempio 2: funzioni di supporto

Come sappiamo, al codice di una applicazione vengono "linkate" diverse funzioni/librerie di supporto, ad esempio: supporto a tempo di esecuzione dei meccanismi di concorrenza (primitive operanti su messaggi o su variabili condivise), handler di interruzioni ed eccezioni, gestione dei dati, oltre a librerie computazionali, ed altro.

In alcuni casi, queste funzioni sono invocate in maniera asincrona rispetto al codice in esecuzione (ad esempio, handler di interruzioni), o comunque sono parallelizzabili. Un esempio del secondo caso è rappresentato da primitive di scambio messaggi su canale asincrono. Supponiamo che un modulo contenga una sezione di codice del tipo:

…; y = F(x); send (…, y); z = G(w); …

L'esecuzione della primitiva *send* può essere parallelizzata all'esecuzione della funzione *G*: ciò significa che il modulo delega l'esecuzione delle primitiva stessa ad un'altra entità computazionale, riuscendo così a sovrapporre il calcolo di *G* con la comunicazione. In alcune architetture multiprocessor, ogni processore ha associato un "processore di comunicazione" specializzato (possibilmente a firmware) ad eseguire le primitive. In alternativa, può essere utilizzato un thread della stessa CPU. A questo scopo, al momento dell'invocazione della primitiva, le istruzione di *G* sono composte (dinamicamente) con le istruzioni del supporto a tempo di esecuzione della *send*.

Se la sovrapposizione di comunicazioni a calcolo è importante agli effetti dell'ottimizzazione del programma, la potenza che viene tolta al calcolo di *G* viene ampiamente riguadagnata: l'esecuzione in serie di calcolo e comunicazione con modalità superscalare a 2 vie (ad esempio) su singolo thread è più costosa dell'esecuzione con modalità SMT su 2 thread del tutto indipendenti.

Un altro esempio dello stesso tipo riguarda la sovrapposizione di calcolo con la gestione di I/O o coprocessori.

4.6.3 Riferimenti a multiprocessor e multicore

Lo studio delle architetture CMP è basato su tutta una serie di principi e tecniche generali nel campo delle architetture parallele a memoria condivisa. Questa trattazione è presente nel Cap. XII, cui si rimanda almeno per quanto riguarda la sezione 1 introduttiva. Citiamo alcuni esempi di multicore commerciali:

- architetture general-purpose a basso parallelismo:
    - basate su x-86:
        - Intel Xeon Core 2 Duo/Quad (2-way superscalar, 1 thread/core), Nehalem (4-8 core, SMT 2 thread/core), Arrendale (2 core), Sandy Bridge(4 core)
        - AMD Athlon, 4 core, SMT 2 thread/core; AMD Opteron 6170 (6 core), 6174 (12 core), SMT 2 thread/core
    - basate su Power PC:
        - IBM Power 5, 6, (4 core, SMT 4 thread/core)
        - IBM BlueGene/P, Power 7 (8 core, SMT 4 thread/core)
        - IBM Cell (eterogeneo: un PowerPC 2 thread, 8 core, 1 thread/core)
    - basate su UltraSPARC:
        - Sun UltraSparc T2 Niagara (8 in-order core, 8 thread/core interleaved)
- architetture ad alto parallelismo orientate a video encoding e multimedia:
    - Tilera TileGX (64-100 core, 1 thread/core)
- architetture a medio parallelismo
    - Schede Intel Champ-AV8con 2 chip Sandy Bridge (8 core)
    - Schede AMD Magny Cours con 4 chip Opteron (48 core)
    - IBM WireSpeed/PowerEN (16 in-order core, SMT 4 thread/core)
- Network processor (specializzati per network processing), esempio:
    - Intel IXP (8-16 in-order core, blocked multithreading, 8 thread/core)

Una trattazione approfondita delle tematiche della programmazione parallela, architetture multiprocessor e multicore è effettuata in corsi avanzati a livello di laurea magistrale, in particolare nel corso *High Performance Computing and Enabling Platforms* della Laurea Magistrale in Informatica e Networking.

4.7 Architetture ILP e parallelismo sui dati

Come studiato nel Cap. X e ricordato nell'esempio della sez. 4.6.3, il paradigma data-parallel è caratterizzato dalla replicazione delle funzioni e dal partizionamento sui dati. Ad esempio, partendo da una computazione sequenziale come

int A[M], B[M];

$\forall\ i = 0\ ..\ M–1$: *B[i] = F(A[i])*

la parallelizzazione di tipo *map* consiste nel partizionare gli array A e B tra un certo numero di worker, ognuno dei quali esegue la funzione *F* indipendentemente sulla propria partizione. In altri casi, i worker devono anche cooperare per scambiarsi dati ad ogni iterazione (data-parallel con *stencil*), come in

int A[M], B[M];

$\forall\ i = 0\ ..\ M–1$: *B[i] = F(A[i], A[i–1], A[i+1])*

Oltre che molto potente per programmi paralleli a livello di processi, il parallelismo sui dati trova importanti applicazioni nel parallelismo a livello di istruzioni ed in alcune architetture multithreaded.

4.7.1 Vettorizzazione

La macchina assembler prevede istruzioni per operare su vettori e matrici, ad esempio

- Somma di vettori
- Reduce
- Ricerca
- Prodotto interno
- Funzioni a supporto del calcolo del gradiente
- Funzioni a supporto dell'eliminazione tri-diagonale
- Funzioni a supporto di equazioni alle ricorrenze
- Funzioni a supporto di integrazione numerica
- ecc.

Ad esempio, l'istruzione di somma di vettori (C = A + B) ha come operandi gli indirizzi base dei tre vettori e la loro dimensione *M*.

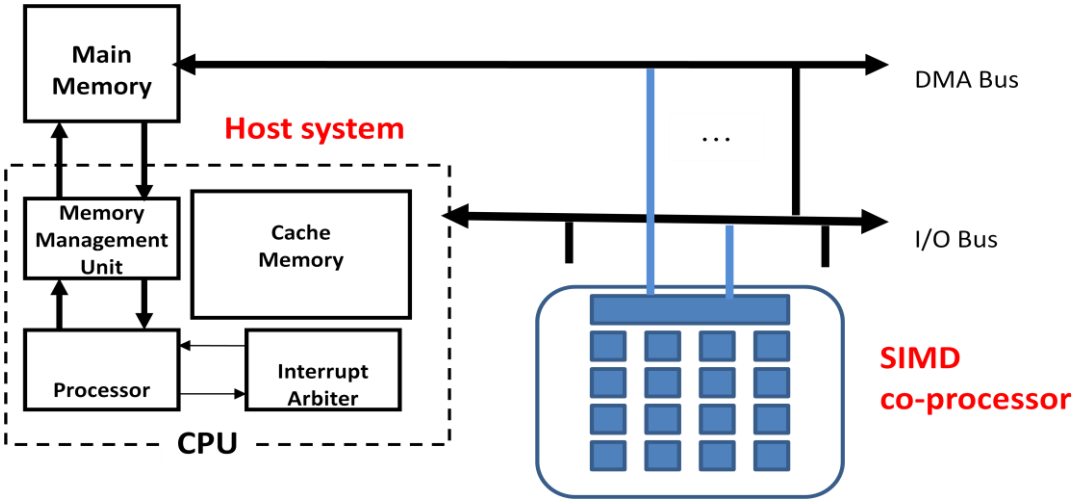
La sua esecuzione permette di sfruttare pienamente l'*Unità Esecutiva in pipeline*: la Cache Dati alimenta una Unità Funzionale con una stream di coppie di operandi, con tempo di interarrivo *t*, e riceve lo stream dei valori dell'array risultato.

Il tempo di servizio è uguale a *t*, e il tempo di completamento a *M t*, con un guadagno di almeno un ordine di grandezza rispetto all'esecuzione dello stesso algoritmo a livello assembler.

La disponibilità di una o più *Unità Funzionali dedicate* alla vettorizzazione aumenta notevolmente il parallelismo nei programmi che facciano uso di tali operazioni.

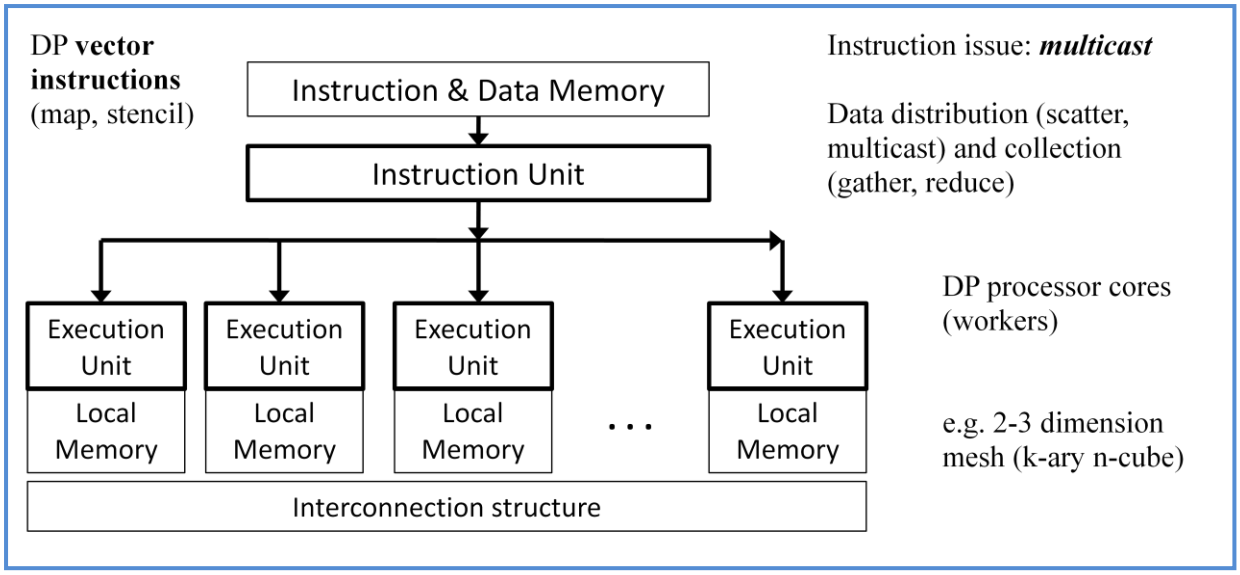
4.7.2 Architetture SIMD e GPU

Le architetture *Single Instruction Steam Single Data Stream* (*SIMD*) sono specializzate all'esecuzione di computazioni data-parallel, anche complesse. Come tali, esse sono usate come coprocessori di I/O di calcolatori ospiti:



Ogni volta che un programma sequenziale incontra una istruzione vettoriale, oppure una chiamata ad una procedura corrispondente ad una certa funzione data-parallel, viene innescata l'elaborazione dell'unità di I/O che interfaccia la macchina SIMD.

La macchina SIMD, schematizzabile come nella figura seguente:



dispone di una propria memoria locale, nella quale in particolare sono presenti i dati bufferizzati per conto delle applicazioni richiedenti. Un elevato numero di *Unità Esecutive* (EU), in genere con associata una propria memoria locale o insieme di registri, funge da insieme di worker data-parallel: tutte eseguono la stessa operazione sulla partizione dei dati loro assegnata, su indicazione di una Unità Istruzioni che le coordina.

Il problema dell'adeguamento della banda richiesta all'effettiva banda offerta dal Bus di I/O al trasferimento dati è uno dei limiti di questa architettura, assieme ad alcuni delicati aspetti di programmabilità.

Attualmente, questo approccio è proprio delle *GPU (Graphic Processing Unit):* machine SIMD su singolo chip, derivate da un processo di generalizzazione di coprocessori specializzati verso l'elaborazione real-time di grafica 3D ad alta qualità.

Le GPU applicano la tecnica del *multithreading*: ogni istanza di esecuzione da parte di una EU (core) è un thread, inteso come in tutto questo capitolo, secondo l'approccio SMT.

Prodotti GPU da parte di AMD e Nvidia includono alcune centinaia di EU on-chip, ognuna multithreaded. L'architettura delle GPU è potenziata da un approccio misto multiprocessor – SIMD: le EU sono partizionate, o partizionabili, in sottoinsiemi, ognuno dei quali al suo interno opera in modo SIMD, mentre le partizioni lavorano reciprocamente come processori di un multiprocessor.