

YRT/2007/4

Modalità di interazione PPE – SPE

Roma, 25/07/2007

Redazione

Ing. Marco Pollini

Approvazione

Ing. Paolo Palazzari

Versione	Data	Redattore
1.0	25/07/2007	Ing. Marco Pollini

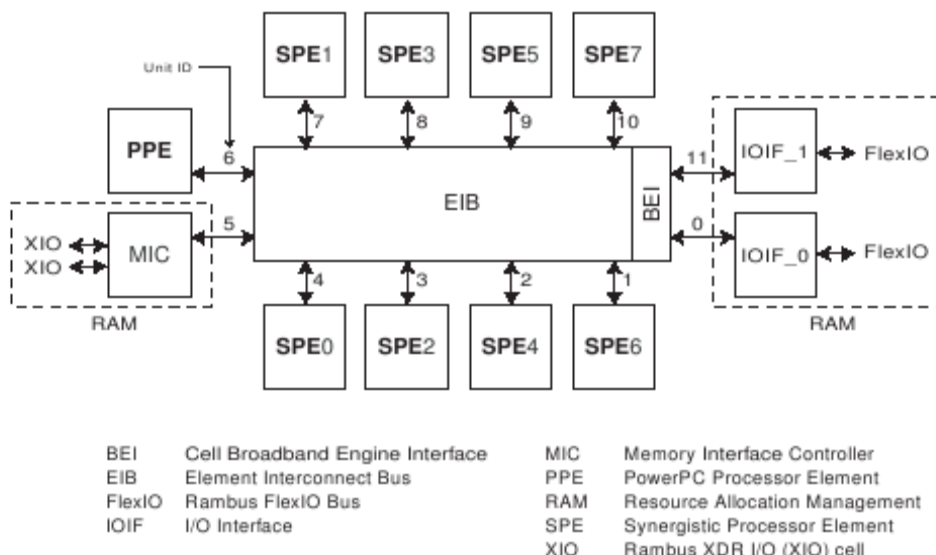
1 INTRODUZIONE.....	1
2 AMBIENTE HARDWARE.....	1
2.1 PROCESSOR ELEMENTS.....	1
2.2 ELEMENT INTERCONNECT BUS – EIB.....	2
2.3 STORAGE DOMAINS.....	3
3 COMUNICAZIONE TRA PROCESSOR ELEMENTS: DMA, MAILBOX, SIGNALING.....	4
3.1 COMANDI MFC.....	5
3.2 COMANDI DMA.....	6
3.3 DMA LIST.....	8
3.4 COMANDI DI SINCRONIZZAZIONE.....	9
3.5 COMANDI DI SINCRONIZZAZIONE.....	11
3.6 ISTRUZIONI SPU DI ORDINAMENTO.....	13
3.7 COMANDI MFC DI SINCRONIZZAZIONE.....	14
4 UTILIZZO DI MAILBOX, SIGNALING E THREAD SPE.....	15
4.1 USO DELLE MAILBOX.....	15
4.2 USO DEL SIGNALING.....	17
4.3 CREAZIONE E GESTIONE THREAD IN ESECUZIONE SU SPE.....	18
5 CODIFICA E STANDARDIZZAZIONE DELLE CHIAMATE DI SISTEMA PER LA SINCRONIZZAZIONE.....	18
5.1 SINCRONIZZAZIONE PPE – SPE.....	18
5.1.1 Sincronizzazione con PPE polling.....	20
5.1.2 Sincronizzazione con mutua esclusione.....	23
5.1.3 Sincronizzazione con variabili condizionali.....	25
5.1.4 Sincronizzazione con interrupt.....	28
5.2 SINCRONIZZAZIONE SPE – SPE.....	29
6 CODIFICA E STANDARDIZZAZIONE DELLE CHIAMATE DI SISTEMA PER IL TRASFERIMENTO DATI TRAMITE DMA.....	30
6.1 TRASFERIMENTI DMA RAM -> LS E LS -> RAM.....	30
6.2 TRASFERIMENTI DMA LS <-> LS.....	32

1 Introduzione

Questo documento analizza la struttura del processore Cell Broadband Engine mettendo in evidenza le modalità di interazione dei suoi vari componenti. In particolare sono state studiate le modalità di interazione tra PPE, SPE e i relativi spazi di memorizzazione. In una seconda fase sono state analizzate le primitive di trasferimento dati, di comunicazione e di sincronizzazione a disposizione nel RTE (Real Time Environment) del CellBE. Successivamente sono state selezionate, codificate e standardizzate le chiamate di sistema per il trasferimento dati tra PPE e SPE o anche tra singoli SPE e per la sincronizzazione di due o più processi in esecuzione su SPE differenti.

2 Ambiente Hardware

2.1 Processor Elements



La Figura 1 mostra il diagramma a blocchi ad alto livello del processore Cell Broadband Engine. Sono presenti un'unità PPE e otto identiche unità SPE. Tutte queste unità, dette anche processor elements, sono connesse le une alle altre, al controller di memoria MIC e ai controller I/O tramite il bus coerente EIB (Element Interconnect Bus).

Il PPE è costituito da un core RISC a 64 bit dual-thread con architettura PowerPC e supporta un sottosistema PowerPC per la gestione della memoria virtuale. La cache è a due livelli: la cache L1 è duplicata per dati e istruzioni ognuna con dimensione 32KB, la cache L2 è unificata con dimensione 512 KB.

Il compito principale del PPE è quello di controllo dell'elaborazione, caricamento del sistema operativo e gestione delle risorse di sistema e dei thread in esecuzione sugli SPE.

L'Instruction-set del PPE è una versione estesa dell'Instruction-set PowerPC e include le estensioni Vector/SIMD Multimedia e gli intrinsics C/C++.

Gli SPE sono processori RISC a 128 bit di tipo SIMD con 256 KB di memoria dedicata (Local Store) per i dati e le istruzioni ed un register file unificato di 128 elementi a 128 bit. Ogni SPE è indipendente dagli altri ed è ottimizzato per eseguire operazioni di elevato carico computazionale allocate dal PPE stesso. A differenza del PPE, gli SPE non sono progettati per il caricamento di un sistema operativo.

Ogni SPE supporta uno speciale instruction-set SIMD, detto Synergistic Processor Unit Instruction Set Architecture, ed un unico set di comandi per la gestione dei trasferimenti DMA e per la comunicazione e il controllo tra i vari processor elements.

C'è una dipendenza mutua tra il PPE e gli SPE; ogni SPE dipende dal PPE per il caricamento del sistema operativo e nella maggior parte dei casi per la gestione del thread principale di controllo dell'applicazione. Il PPE dipende dagli SPE per la parte più importante delle prestazioni dell'applicazione.

La differenza più significativa tra PPE e SPE risiede nel modo in cui essi accedono alla memoria. Il PPE accede al main-storage (detto effective-address space – EA) con istruzioni di load e store che spostano i dati dalla memoria in un register file interno il cui contenuto può essere tenuto in cache.

Al contrario gli SPE possono accedere al main-storage solo tramite comandi DMA che muovono i dati dalla memoria centrale ai vari Local Store. Il fetch delle istruzioni e le operazioni di load e store di un SPE avvengono solo con il proprio LS.

2.2 Element Interconnect Bus – EIB

Il bus EIB costituisce il percorso di comunicazione per i dati e i segnali di controllo tra tutti i processor elements del CBE, il controller di memoria MIC e le interfacce I/O.

E' un bus coerente nel senso che permette la condivisione di un singolo spazio di indirizzamento da parte del PPE e degli SPE. E' costituito da quattro anelli di comunicazione da 16 byte ognuno dei quali può trasferire 128 byte per volta (un'intera linea di cache del PPE).

La Figura 1 mostra anche le connessioni dei processor elements al bus e i rispettivi Unit ID. La conoscenza dell'ordine in cui sono disposti i vari elementi sul bus è fondamentale per il programmatore al fine di minimizzare la latenza dei trasferimenti su bus; infatti questa è funzione del numero di nodi di connessione cosicchè la latenza più breve si avrà per un trasferimento tra elementi adiacenti mentre quella più alta per trasferimenti tra elementi separati da un numero di nodi pari a sei.

La banda interna massima del bus EIB è di 96 byte per ogni ciclo di clock.

2.3 Storage domains

Il processore CBE ha due tipi di storage domains (o domini di memoria): un main-storage domain e otto local-store domains come indicato in Figura 2.

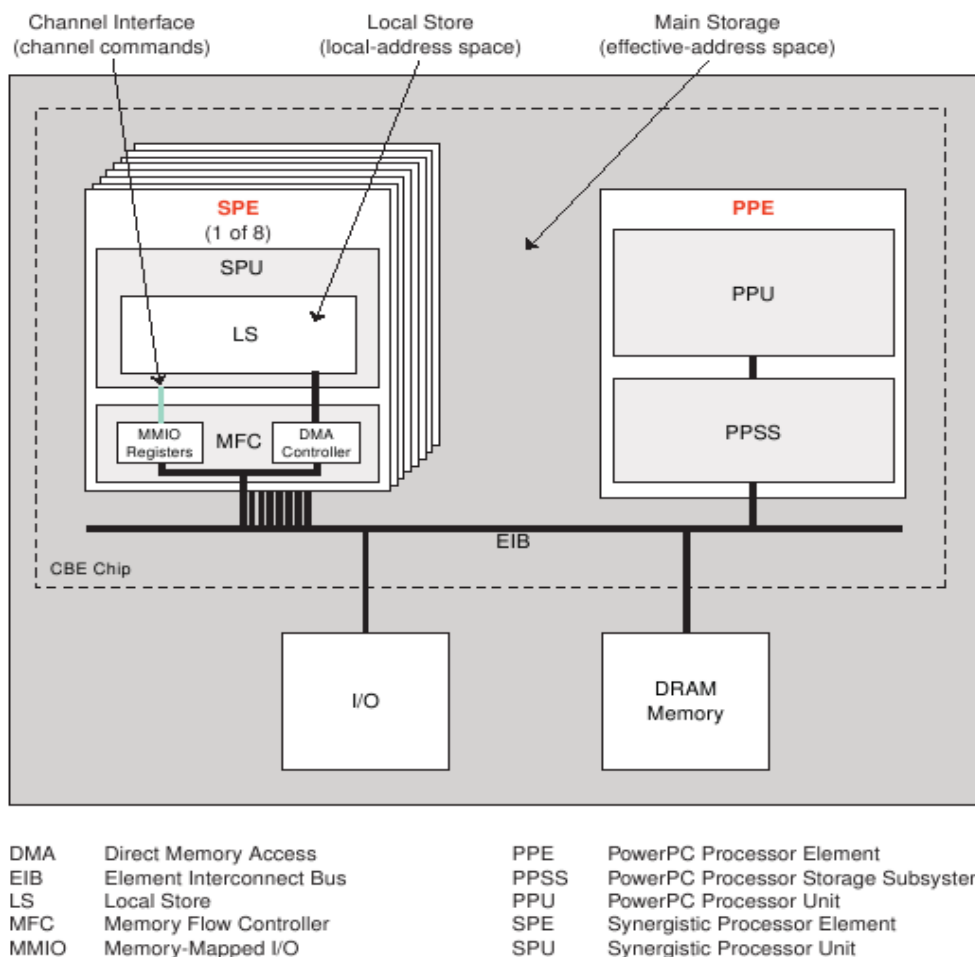


Figura 2 – Storage domains e interfacce

In più ogni SPE ha un proprio “canale di comunicazione” (channel interface) che permette il collegamento tra l'unità di elaborazione SPU (Synergistic Processor Unit) e il modulo MFC (Memory Flow Control). Questo canale rappresenta un'interfaccia unidirezionale per lo scambio di messaggi.

Il main-storage domain, che rappresenta l'intero spazio effective-address (EA), può essere configurato dal PPE per essere condiviso da tutti i processor elements e da altre periferiche mappate in memoria. Per ogni programma, l'effective-address space (264 bytes) è un sottoinsieme del più grande virtual-address space (VA – 265 bytes). L'hardware di gestione della memoria nel PPE e quello presente nel MFC di ogni SPE traduce un indirizzo EA in VA e successivamente in un indirizzo Real Address (RA – 242 bytes).

La trasformazione da indirizzi EA a VA è detta *segmentazione*.

Ogni segmento di memoria è costituito da un'area di memoria virtuale protetta e non sovrapposta di indirizzi contigui di dimensione 256 MB. Il meccanismo di traslazione da VA a RA è detto *paging*.

Una pagina è costituita da un'area di memoria dello spazio RA, protetta, non sovrapposta e rilocabile di dimensioni comprese tra 4 KB e 16 MB.

3 Comunicazione tra processor elements: DMA, mailbox, signaling

Il processore CBE possiede molte delle caratteristiche di un sistema a memoria condivisa ma tradizionalmente non è un sistema multiprocessore a memoria condivisa. Infatti i singoli SPE possono eseguire programmi e caricare dati e istruzioni solo dalla propria memoria locale LS. In un sistema multiprocessore tradizionale a memoria condivisa il trasferimento dati e la sincronizzazione tra i vari processori risulta essere in parte una conseguenza del fatto che tutti i processori utilizzano lo stesso spazio di memoria.

Nel CBE gli SPE comunicano esplicitamente con gli altri elementi del sistema utilizzando tre principali meccanismi di comunicazione: trasferimenti DMA, messaggi mailbox e messaggi signal-notification.

Tutti questi meccanismi di comunicazione sono implementati e gestiti dal MFC di ogni SPE.

Meccanismo	Descrizione
Trasferimento DMA	Utilizzato per muovere dati dal main-storage ai LS. I DMA sono asincroni e permettono di nascondere la latenza della memoria e l'overhead di trasferimento spostando i dati in parallelo con la computazione degli SPU
Mailbox	Utilizzate per comunicazioni di controllo tra SPE e PPE o altri device (anche SPE). Ogni messaggio mailbox è di 32 bit. Ogni SPE è fornito di due mailbox per l'invio di messaggi e di una mailbox per la ricezione.
Signal notification	Utilizzata per comunicazioni di controllo dal PPE o da altri device (anche SPE). La signal-notification (detta anche <i>signaling</i>) utilizza registri a 32 bit che possono essere configurati per signaling di tipo one-sender / one receiver oppure many-senders / one-receiver.

Tabella 1 : Metodi di comunicazione tra processor elements

Sebbene i metodi di signaling e mailbox appaiano simili nella struttura e nel loro utilizzo, ci sono alcune importanti differenze, mostrate in Tabella 2.

Attributi	Mailbox	Signaling
Direction	1 mail in ingresso, 2 mail in uscita accessibili tramite channel interface	2 signal register in ingresso accessibili tramite channel interface
Interrupts	1 mailbox con interrupt	2 signal-notification interrupts
Message accumulation	No	Modalità sovrascrittura in one-to-one signaling. Modalità logical OR in many-to-one signaling
Unique SPU Commands	Utilizzo di read e write su channel interface	Comandi: sndsig, sndsigf e sndsigb
Destructive Read	La lettura della mailbox consuma un elemento della mail	La lettura resetta tutti i bit del registro a '0'.
Channel Count	Indica il numero di elementi disponibili della mailbox	Indica un segnale in attesa
Number	Tre mailbox: una in entrata con 4 elementi, una in uscita con 1 elemento, una in uscita con interrupt	Due signal register

Tabella 2 : Differenze tra mailbox e signaling

3.1 Comandi MFC

L'MFC di un SPE può essere considerato come co-processore dell'SPU associato che esegue autonomamente le operazioni richieste durante l'esecuzione di un programma. L'MFC è dotato di due code indipendenti che memorizzano la sequenza dei comandi da eseguire; una prima coda di 16 elementi detta MFC SPU Command Queue e una seconda coda di 8 elementi detta MFC Proxy Command Queue. I comandi da eseguire su richiesta dell'SPU sono inseriti esclusivamente nella SPU Command Queue, mentre quelli da eseguire su richiesta del PPE o di un altro device sono inseriti nella Proxy Command Queue. Ovviamente ogni SPU accede al proprio MFC tramite la già menzionata channel-interface.

Gli altri device nel sistema, incluso il PPE, utilizzano registri memory-mapped, detti MMIO register, per accedere al MFC di un SPE e inviare comandi.

Ogni MFC ha un proprio controller DMA che implementa i trasferimenti DMA tra main-storage e Local Store. I comandi DMA contengono al loro interno i valori degli indirizzi di memoria del LS (LSA – Local Store Address) e del main-storage (EA – Effective Address). Ogni MFC traduce i comandi DMA istanziati da un SPU nella coda SPU Command Queue in trasferimenti DMA effettivi e può iniziare e gestire trasferimenti multipli.

Ad ogni comando MFC viene associato un attributo numerico a 5 bit detto Tag Group ID che viene utilizzato dal software dell'SPE per monitorare l'esecuzione e il completamento di una particolare sequenza di comandi in coda che abbiano un determinato Tag Group.

L'MFC supporta l'esecuzione out-of-order dei comandi ma, se richiesto dal software in esecuzione, può essere mantenuto un particolare ordinamento utilizzando le seguenti modalità:

- Utilizzare un comando con lo specificatore “**barrier**” per ottenere un ordinamento locale del comando rispetto a tutti gli altri comandi in coda, precedenti e seguenti, appartenenti allo stesso Tag Group.
- Utilizzare un comando con lo specificatore “**fence**” per ottenere un ordinamento locale del comando rispetto a tutti gli altri precedenti comandi in coda, appartenenti allo stesso Tag Group.
- Utilizzare un comando “**barrier**” separato per ottenere un ordinamento locale del comando rispetto a tutti gli altri comandi in coda, precedenti e seguenti senza alcun riferimento al Tag Group.

I comandi MFC sono divisi in due tipi : comandi DMA e comandi di sincronizzazione. Tutti i comandi DMA sono accodabili, solo tre dei comandi di sincronizzazione sono immediati.

3.2 Comandi DMA

I trasferimenti DMA possono essere istanziati sia da comandi eseguiti da PPE che da SPE. Ogni operazione DMA può trasferire fino a 16 KB di dati tra main-storage e LS. L'unità MFC supporta trasferimenti, con allineamento dei dati, di dimensioni 1, 2, 4, 8, 16 byte e multipli di 16 byte.

Le prestazioni di un DMA possono essere incrementate quando sorgente e destinazione del trasferimento hanno lo stesso allineamento a 128 byte (quadword-offset); in questo modo si generano sul bus richieste di un'intera linea di cache (128 byte).

Al contrario le prestazioni di un trasferimento con allineamento differente sono circa la metà di quelle ottenibili con allineamento quadword in quanto ogni richiesta del bus trasferisce solo una parte di un'intera linea di cache. Le prestazioni ottimali nel trasferimento si ottengono quando sia l'indirizzo del main-storage (EA) sia l'indirizzo del LS (LSA) sono allineati a 128 byte e la dimensione del trasferimento è un multiplo di 128 byte.

Comando	SPU	PPE	Descrizione
put	*	*	Trasferimento dati da LS a EA
puts		*	Trasferimento dati da LS a EA e start SPU a completamento del DMA
putf	*	*	Trasferimento dati da LS a EA con fence (il comando viene ordinato in base a tutti gli altri precedenti comandi in coda con lo stesso tag group)
putb	*	*	Trasferimento dati da LS a EA con barrier (il comando e tutti i precedenti con stesso tag group vengono ordinati in base a tutti gli altri comandi precedenti in coda con lo stesso tag group)
putfs		*	Trasferimento dati da LS a EA con fence (il comando viene ordinato in base a tutti gli altri precedenti comandi in coda con lo stesso tag group) e start SPU a completamento del DMA
putbs		*	Trasferimento dati da LS a EA con barrier (il comando e tutti i precedenti con stesso tag group vengono ordinati in base a tutti gli altri comandi precedenti in coda con lo stesso tag group) e start SPU a completamento del DMA
putl	*		Trasferimento dati da LS a EA con utilizzo di una DMA List
putlf	*		Trasferimento dati da LS a EA con utilizzo di una DMA List e con fence
putlb	*	*	Trasferimento dati da LS a EA con utilizzo di una DMA List e con barrier

Tabella 3 : Comandi DMA Put

Comando	SPU	PPE	Descrizione
get	*	*	Trasferimento dati da EA a LS
gets		*	Trasferimento dati da EA a LS con start SPU a completamento del DMA
getf	*	*	Trasferimento dati da EA a LS con fence (il comando viene ordinato in base a tutti gli altri precedenti comandi in coda con lo stesso tag group)
getb	*	*	Trasferimento dati da EA a LS con barrier (il comando e tutti i precedenti con stesso tag group vengono ordinati in base a tutti gli altri comandi precedenti in coda con lo stesso tag group)

getfs		*	Trasferimento dati da EA a LS con fence (il comando viene ordinato in base a tutti gli altri precedenti comandi in coda con lo stesso tag group) e start SPU a completamento del DMA
getbs		*	Trasferimento dati da EA a LS con barrier (il comando e tutti i precedenti con stesso tag group vengono ordinati in base a tutti gli altri comandi precedenti in coda con lo stesso tag group) e start SPU a completamento del DMA
getl	*		Trasferimento dati da EA a LS con utilizzo di una DMA List
getlf	*		Trasferimento dati da EA a LS con utilizzo di una DMA List e con fence
getlb	*		Trasferimento dati da EA a LS con utilizzo di una DMA List e con barrier

Tabella 4 : Comandi DMA Get

3.3 DMA List

Una DMA List è costituita da una sequenza di elementi di 8 byte, ognuno dei quali rappresenta un trasferimento DMA, memorizzata nel LS di un SPE. Un comando DMA List (getl,putl) può essere chiamato solamente dagli SPU, non da altri device nè dal PPE stesso. Tuttavia la creazione della lista stessa può essere effettuata dal PPE.

Una DMA List può contenere fino a 2048 elementi, cioè trasferimenti DMA, ognuno capace di muovere fino a 16 KB di dati; ciò significa che si possono trasferire fino a 32 MB di dati con un solo comando DMA List. La dimensione massima di una lista è 16 KB (2048 elementi di 8 byte).

Ogni elemento di una lista contiene un campo che indica la dimensione del trasferimento, un altro campo che contiene la parte meno significativa dell'indirizzo EA e un bit , detto di “stall-and-notify”, che permette di sospendere l'esecuzione dei trasferimenti DMA dopo l'ultimo elemento della lista che abbia tale bit a 1.

Un comando DMA List specifica un indirizzo LS di partenza per l'intera lista e i dati nel LS sono acceduti in modo sequenziale a passi di 128 byte. Ogni elemento della lista specifica un nuovo indirizzo EA di partenza ma l'indirizzo LS di un dato elemento è quello immediatamente successivo all'indirizzo finale del precedente elemento. Unica eccezione è data da una DMA List con trasferimenti compresi tra 1 e 8 bytes, nel qual caso l'indirizzo del LS viene incrementato a passi di 128 byte per caricare l'elemento successivo della lista; in questo modo rimangono spazi vuoti nel LS tra gli elementi della lista.

La creazione della lista può avvenire dal parte del PPE e il caricamento nel rispettivo LS deve essere fatto necessariamente con allineamento a 8 byte. Come rappresentato in Figura 2, una lista è della forma {LTS, LEAL}.

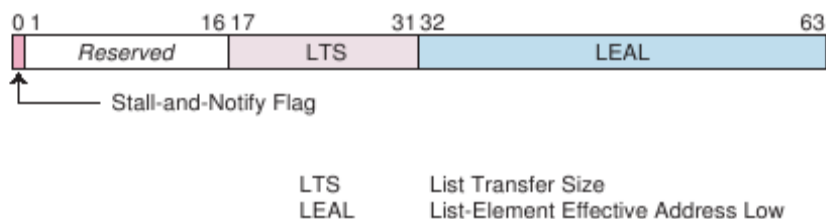


Figura 2 - Struttura DMA List

Il primo campo LTS indica la dimensione di trasferimento, il cui bit più significativo viene utilizzato opzionalmente come bit di stall-and-notify. Il secondo campo LEAL rappresenta la parte meno significativa a 32 bit dell'indirizzo EA. Gli elementi della lista sono processati sequenzialmente nell'ordine in cui sono memorizzati. Se un certo elemento della lista ha il bit di stall-and-notify settato, l'MFC interrompe i trasferimenti DMA associati alla lista dopo aver eseguito l'ultimo DMA associato a questo elemento.

3.4 Comandi di sincronizzazione

Sebbene il processore CBE esegua le istruzioni secondo l'ordine del programma eseguito, il modello di storage dei dati in memoria è “debolmente consistente” (weakly consistent) cioè l'ordine in cui un processor element (PPE o SPE) esegue gli accessi in memoria può essere diverso dall'ordine in cui questi accessi sono eseguiti rispetto ad altri processor elements e diverso dall'ordine con cui questi sono eseguiti nel main storage.

Questo modello di storage permette di riordinare gli accessi in memoria dinamicamente con l'opportunità di aumentare le prestazioni generali e ridurre gli effetti sul throughput istruzioni dovuti alla latenza della memoria.

Per questo motivo in un ambiente di memoria condiviso da altri thread o periferiche di I/O, è richiesto che il programma stesso esegua esplicitamente un ordinamento degli accessi al fine di ottenere la corretta sequenza indicata nel codice.

Per garantire che gli accessi in memoria (considerati come accessi al main storage, che includono anche quelli esterni ai Local Store degli SPE) siano eseguiti secondo la sequenza data dal programma, il software deve inserire istruzioni particolari dette di “memory barrier” tra le varie istruzioni di memoria, come richiesto dal Cell Broadband Engine Architecture (CBEA).

Un programma che non contenga istruzioni di memory barrier può funzionare correttamente in un certo ambiente di esecuzione ma può creare problemi in un altro o su una diversa implementazione del CBEA.

Il termine “accesso in memoria” (storage access) sta a significare un accesso al main storage dovuto ad un'istruzione di load, store, una lettura o scrittura DMA.

Ci sono due ordinamenti da considerare:

Ordine di esecuzione delle istruzioni

Le istruzioni possono essere eseguite in-order o out-of-order. In un processore in-order, come il CBE, ogni istruzione è completata prima dell'inizio dell'istruzione successiva e dal punto di vista del programma ciò appare come un'esecuzione sequenziale delle istruzioni specificata dal programma stesso. Un processore out-of-order invece non garantisce che le istruzioni siano eseguite nell'ordine specificato nel programma e il software stesso non può rilevare questo diverso ordine di esecuzione.

Ordine degli accessi in memoria

L'ordine con cui gli accessi in memoria vengono effettuati può essere differente sia da quello indicato dal programma sia da quello in cui le istruzioni che effettuano gli accessi sono eseguite. Ad esempio consideriamo un programma che contenga nell'ordine tre istruzioni di load dalle locazioni di memoria A,B,C. Il processore può eseguire queste istruzioni in un ordine differente (B,A,C) ed effettuare gli accessi effettivi in memoria in un altro ordine ancora (C,B,A).

Il CBEA definisce inoltre quattro attributi di controllo storage che sono memorizzati nella page-table:

W – Write-Through : quando un accesso è di tipo write-through, se il dato è in cache (L1 o L2 cache del PPE), un'operazione di store aggiorna il valore del dato sia in cache che nel main storage.

I – Caching-Inhibited : gli accessi in memoria sono effettuati solo con riferimento al main storage bypassando la cache.

M – Memory-Coherence : in questo caso viene inviato un segnale agli altri elementi del processore a indicare che l'accesso in memoria è di tipo globale e tutti gli elementi coinvolti nell'operazione devono rispondere con un segnale di acknowledge.

G – Guarded : questo attributo è utilizzato quando un'area del main storage deve essere protetta da accessi estranei a quelli specificati nel programma in esecuzione.

Il main storage possiede l'attributo memory-coherence, i LS degli SPE e le periferiche di I/O che normalmente sono mappati entrambi nel main-storage (EA - effective-address) del processore hanno gli attributi cache-inhibited e guarded.

3.5 Comandi di sincronizzazione

Il PPE supporta le “barrier instructions” per l'ordinamento degli accessi in memoria e l'esecuzione delle istruzioni. La PowerPC Architecture definisce due tipi di barrier instructions:

Storage barriers – sync, lwsync, eieio

Instruction barrier – isync

Possono essere utilizzate all'interno di una sequenza di istruzioni per l'accesso in memoria e definire una memory barrier che divida le istruzioni in due gruppi, quello che precede la memory barrier e quello che la segue. In questo modo gli accessi di memoria richiesti dalle istruzioni che precedono la barrier instruction sono eseguiti prima degli accessi richiesti dalle istruzioni che seguono la barriera.

Sync instruction

Assicura che tutte le istruzioni che la precedono siano completate e che tutte le istruzioni seguenti non siano iniziate prima del completamento dell'istruzione di sync. Tuttavia ciò non significa che gli accessi in memoria precedenti ad un'istruzione sync siano esauriti prima del completamento della sync.

La sync esegue inoltre le seguenti operazioni:

- Previene le operazioni di store simultanee su una memoria con attributi caching-inhibited o write-through.
- Previene le load simultanee su una memoria caching-inhibited.

E' cumulativa, nel senso che tutti gli elementi del processore vedranno nel sistema la stessa sequenza con barriera che è stata inizializzata dal processore element che ha eseguito l'istruzione di sync (in questo caso il PPE).

La sync può essere utilizzata, ad esempio, per assicurarsi che i risultati di tutte le operazioni di store su una struttura dati, generate da una sezione critica del programma in esecuzione, siano visibili da tutti gli altri processor element prima che tale struttura sia resa di nuovo disponibile per altre operazioni.

Lwsync instruction

Crea lo stesso tipo di barriera dell'istruzione di sync per accessi memory-coherence. Tuttavia la lwsync non crea una barriera per accessi di memoria caching-inhibited o write-through.

A differenza della sync, l'istruzione lwsync gestisce solamente l'ordinamento degli accessi al main-storage del PPE; non ha alcun effetto sugli accessi al main-storage effettuati da altri processor element.

Dovrebbe essere utilizzata solo nel caso di ordinamento memory-coherent in quanto è più veloce della corrispondente istruzione di sync.

Eieio instruction

L'istruzione eieio (enforce in-order execution of I/O) assicura che tutti gli accessi al main storage precedenti alla eieio siano completati prima di quelli iniziati successivamente all'istruzione stessa.

L'istruzione di eieio ordina le load e le store secondo le due seguenti modalità:

1. Load e store su memoria caching-inhibited e guarded, store su memoria write-through. Tutti gli accessi di memoria che rientrano in questo insieme sono ordinati come singolo insieme cioè non c'è un ordinamento per le load e le store caching-inhibited e guarded e un altro diverso per le store write-through.
2. Store su memoria con memory-coherence ma senza caching-inhibited nè write-through.

In più l'istruzione eieio ha le seguenti caratteristiche:

- Previene le store simultanee su memoria caching-inhibited e guarded o su memoria write-through.
- Previene le load simultanee su memoria caching-inhibited

E' cumulativa per memoria che non sia caching-inhibited nè write-through.

Il PowerPC Architecture specifica che le operazioni di store su memoria che siano caching-inhibited che guarded vengano eseguite secondo l'ordine dato dal programma; dunque l'istruzione eieio è necessaria per quelli accessi in memoria dove le load devono essere ordinate rispetto alle store o rispetto ad altre load, oppure quando si deve prevenire la combinazione di load e store assieme.

L'istruzione eieio è normalmente utilizzata per la gestione di strutture dati condivise, nelle operazioni I/O memory-mapped e per evitare la presenza combinata di load e store .

Isync instruction

Questa istruzione assicura che tutte le istruzioni PPE precedenti la isync siano completate prima della fine della isync.

Un'istruzione isync determina uno stallo e blocca tutte le altre istruzioni di entrambi i thread PPE fino al suo completamento. Le istruzioni già caricate per il thread in fase di issue vengono scartate mentre quelle dell'altro thread non vengono modificate, quelle che sono già state caricate in precedenza e pronte per l'esecuzione non vengono ricaricate dopo il completamento dell'istruzione isync.

3.6 Istruzioni SPU di ordinamento

Un Local Store può interagire in modo asincrono con le seguenti operazioni:

- Instruction fetch dal SPU locale.
- Load e Store dal SPU locale.
- Trasferimenti DMA dal MFC locale o dall'MFC di un altro SPU.
- Load e Store nel main-storage da parte di altri processor element o periferiche.

L'ISA di un SPU prevede tre tipi di istruzioni di sincronizzazione: **sync**, **dsync**, **synccc**.

Con in termini “External Read” e “External Write” indichiamo gli accessi di memoria effettuati durante ogni trasferimento DMA o da qualsiasi altro processor element (compreso il PPE) o periferica escluso l'SPU che esegue la sincronizzazione.

Con i termini “SPU Load” e “SPU Store” indichiamo gli accessi effettuati dall'SPU che esegue la sincronizzazione.

Sync

Questa istruzione determina uno stallo del SPU che rimane in attesa del completamento di tutte le operazioni di store nel LS pendenti prima di eseguire il fetch dell'istruzione successiva. In pratica crea un ordinamento per l' instruction fetch, le load, le store e gli accessi al channel interface.

Dopo che un SPU ha eseguito una sync, le operazioni di fetch delle istruzioni ritorneranno i dati memorizzati dalla più recente istruzione di store o scrittura esterna (External Write). In più, l'SPU avrà completato tutte le precedenti load, store, e channel access e non avrà iniziato l'esecuzione di alcuna istruzione di load o store successiva.

In questo momento una read esterna leggerebbe il dato messo in memoria dalla più recente store.

Le operazioni di load successive alla sync ritornano i dati scritti nell'istante prima del completamento della sync.

L'istruzione di sync ha effetto solo sulla sequenza di istruzioni dell'SPU e sulla coerenza delle operazioni di fetch, load e store rispetto allo stato attuale del LS e non influisce sullo stato dei dispositivi esterni.

Il buffer dell'instruction-fetch e le pipeline dell'SPU vengono svuotati quando si esegue una sync la quale non ha inoltre alcun effetto sulle operazioni DMA controllate dal MFC.

Questa istruzione è spesso utilizzata nel caso di codice self-modifying su SPU, per cui la sync è inserita prima del caricamento, ad es. tramite DMA, del nuovo codice da eseguire.

Dsync (Data synchronization)

L'istruzione dsync permette al software SPE di assicurarsi che i dati siano stati effettivamente messi nel LS prima che questi diventino visibili al MFC locale o ad altri device. Questa istruzione ordina load, store, accessi al channel interface ma non i fetch delle istruzioni, e non influisce in nessun modo sull'eventuale prefetching delle istruzioni.

Dopo il completamento di una dsync, l'SPU avrà completato tutte le load, store, accessi al channel interface precedenti e non avrà iniziato quelle immediatamente successive. In questo momento una lettura esterna da un indirizzo LS prenderà i dati memorizzati dalla più recente operazione di store. Le load dell'SPU dopo la dsync ritornano i dati scritti nell'istante precedente al completamento della dsync.

L'istruzione dsync ha effetto solo sulla sequenza di istruzioni dell'SPU e sulla coerenza delle operazioni di load e store rispetto allo stato attuale del LS e non influisce sullo stato dei dispositivi esterni.

Syncc (channel synchronization)

L'istruzione syncc esegue un'operazione di sincronizzazione del canale costringendo tutte le channel instructions a terminare prima dell'esecuzione della successiva istruzione SPU. Assicura che gli effetti sullo stato del SPU, causati da precedenti istruzioni **wrch (Write Channel)**, siano propagati ed influenzino l'esecuzione delle istruzioni successive.

I dati che transitano attraverso un canale non sono mai riordinati. Un canale può essere “bloccante”, nel qual caso i dati da leggere dovrebbero essere presenti prima che l'istruzione che accede il canale sia completata oppure “non-bloccante”, nel qual caso il software deve gestire la sincronizzazione basandosi sull'ordinamento dei dati nel canale.

3.7 Comandi MFC di sincronizzazione

I comandi di sincronizzazione del Memory Flow Control (MFC) sono utilizzati per gestire l'ordine degli accessi in memoria.

Questi includono quattro comandi atomici (**getllar**, **putllc**, **putlluc**, **putqlluc**), tre comandi di tipo signal-notification (**sndsig**, **sndsigf**, **sndsigb**) e tre comandi di tipo barrier (**mfcsync**, **mfcieio**, **barrier**).

Comando	SPU	PPE	Immediato	Descrizione
getllar		*	*	Get lock line and reserve
putllc		*	*	Put lock line conditional
putlluc		*	*	Put lock line unconditional
putqlluc	*	*		Put queued lock line unconditional
barrier	*	*		Assicura l'ordinamento di tutti i precedenti comandi DMA rispetto a quei comandi che seguono la barriera nella stessa coda. Questo comando non ha effetto sui tre comandi immediati getllar,putllc,putlluc.
mfceieio	*	*		Controlla l'ordinamento dei comandi get rispetto ai comandi put e dei get rispetto ad altri comandi get che accedono memoria con attributi caching-inhibited e guarded. Inoltre controlla l'ordinamento dei comandi put rispetto ai put che accedono memoria con attributo memory-coherence.
mfcsync	*	*		Controlla l'ordinamento dei comandi DMA get e put all'interno di uno specifico tag group
sndsig	*	*		Scriva il Signal Notification Register di un altro SPU
sndsigf	*	*		Scriva il Signal Notification Register di un altro SPU, con fence
sndsigb	*	*		Scriva il Signal Notification Register di un altro SPU, con barrier

4 Utilizzo di mailbox, signaling e thread SPE

4.1 Uso delle mailbox

Le mailbox permettono l'invio e la ricezione di messaggi a 32 bit tra SPE e altri device, ad es. il PPE o altri SPE. Ogni SPE può accedere a tre mailbox differenti, ognuna delle quali è connessa ad un mailbox register nel MFC del SPU. Sono presenti due mailbox in uscita, chiamate SPU Write Outbound Mailbox e SPU Write Outbound Interrupt Mailbox ognuna capace di contenere un singolo messaggio e che permettono di inviare messaggi da SPE a PPE o altri SPE.

E' presente una mailbox in entrata, detta SPU Read Inbound Mailbox capace di contenere un massimo di quattro messaggi e che permette l'invio di messaggi dal PPE agli SPE o da SPE a SPE. Ogni Outbound mailbox ha inoltre il proprio MMIO register che può essere acceduto dal PPE o da altri device.

Un programma in esecuzione su SPE può scrivere su una delle due mailbox di uscita tramite un'istruzione di write-channel (wrch) e i dati della mailbox sono disponibili per qualsiasi processor-element che acceda il corrispondente MMIO register dell'SPE nel main-storage.

Analogamente un messaggio scritto sulla Inbound mailbox di un SPE tramite MMIO register può essere letto dal software in esecuzione sullo stesso SPE tramite un'istruzione di read-channel (rdch). Le scritture e le letture della SPU Inbound mailbox determinano rispettivamente un incremento e un decremento del contatore di messaggi. La mailbox in entrata agisce come una coda FIFO e se il PPE scrive su questa un numero di messaggi superiore alla capacità di ricezione (quattro), l'ultimo messaggio inserito nella mailbox viene sovrascritto e il contatore non si incrementa.

La scrittura su una Outbound mailbox piena causa lo stallo del rispettivo SPE fino a quando non venga effettuata una lettura; la lettura di una Inbound mailbox vuota causa ugualmente lo stallo dell'SPE. Tuttavia questo comportamento si applica solo agli SPE: la scrittura su una Inbound mailbox piena da parte del PPE non determina lo stallo del processore.

Le mailbox vengono principalmente utilizzate per lo scambio di messaggi a 32 bit che rappresentino flags di fine operazione o lo stato di esecuzione di un programma. In effetti possono anche essere utilizzate per trasferimenti di piccole quantità di dati, ad es. indirizzi di memoria, parametri di funzioni e di comandi.

Un utile impiego è quello di sfruttare i messaggi mailbox come segnali di sincronizzazione tra SPE e PPE; si può effettuare un DMA nel main-storage da SPE, attendere il completamento dell'operazione e notificare al PPE l'avvenuto trasferimento di dati tramite una scrittura nella Outbound mailbox.

Tuttavia ciò assicura solamente che l'area di memoria del LS da cui abbiamo eseguito il DMA sia pronta per essere riutilizzata, ma non garantisce che i dati siano stati scritti coerentemente nel main-storage. La soluzione a questo problema è quella di utilizzare una istruzione di lwsync subito dopo aver ricevuto la notifica di avvenuto trasferimento tramite mailbox e prima di accedere i dati nel main-storage.

Alternativamente la notifica di fine trasferimento DMA può essere effettuata tramite DMA stesso nel main-storage al quale può accedere direttamente il PPE e in questo caso si parla di “writeback DMA”.

E' necessario tuttavia ordinare l'operazione di scrittura dati in memoria rispetto al writeback e per fare ciò si può inserire un'istruzione mfceio tra il comando di trasferimento DMA e il DMA di notifica.

Sebbene le mailbox siano utilizzate principalmente per la comunicazione tra SPE e PPE, esiste la possibilità di scambiare messaggi tra le mailbox di due o più SPE. Per ottenere ciò è necessario che il software in esecuzione sul PPE effettui una mappatura della cosiddetta “problem-state area” del SPE nel main-storage; in questo modo ogni SPE conoscerà l'indirizzo effettivo di memoria dei registri memory-mapped del SPE con cui vuole comunicare o al quale deve essere inviato un messaggio via mailbox.

In assenza di ciò la comunicazione tra diversi SPE è limitata alle operazioni atomiche, prima citate, e al signal notification.

4.2 Uso del signaling

Ogni SPE possiede due registri a 32 bit per il signal-notification, ognuno dei quali ha un corrispondente registro MMIO che può essere scritto con i dati della signal-notification.

La lettura del signal-notification register provoca il reset di tale registro a differenza di una lettura del MMIO register nel main-storage.

Il PPE invia un messaggio di signal-notification agli SPE scrivendo sul registro MMIO del MFC che viene successivamente letto dal SPU eseguendo un'istruzione di read-channel (rdch).

Un SPU può invece comunicare con un altro SPU tramite signal-notification utilizzando tre comandi del MFC: sndsig, sndsigf, sndsigb. Questi tre comandi agiscono a tutti gli effetti come un comando DMA put con indirizzo di destinazione uguale all'effective address del registro MMIO.

Analogamente al comportamento delle mailbox, una lettura di un signal register senza alcun segnale pendente causa uno stallo del corrispondente SPE fino a quando non venga effettuata una scrittura sul registro medesimo.

Come per le mailbox, il signaling è utile per i messaggi di notifica verso il PPE inviati per segnalare la fine di un trasferimento DMA iniziato da un SPE. Come già detto in precedenza, questo metodo garantisce solo che l'area del LS da dove è stato effettuato il DMA possa essere riutilizzata ma non assicura che i dati trasferiti siano stati scritti coerentemente nel main-storage. In questo caso valgono le stesse considerazioni fatte precedentemente per i messaggi tramite mailbox.

I registri possono essere configurati indipendentemente dal PPE per operare in due modalità differenti: OR mode o Overwrite mode.

Nella modalità OR l'MFC del SPE combina tutti i valori del registro con un'operazione di OR logico fino a che non venga effettuata un'operazione di lettura, nel qual caso il registro viene resettato a zero.

La modalità OR permette l'accumulazione di segnali multipli in un singolo registro di signaling; ad ogni device (ad es. un SPE) che invia un segnale viene riservato un sottoinsieme dei 32 bit del registro permettendo così al device di inviare segnali in qualsiasi momento e indipendentemente da altri device che eseguono la stessa operazione. Nel momento in cui un SPE legge il proprio registro di signaling ha a disposizione l'intero set di segnali inviati a partire dall'ultima lettura effettuata.

Questa modalità è anche chiamata “many-to-one signaling”.

Nella modalità overwrite invece ogni valore scritto nel registro sovrascrive il valore precedente. In questo caso si parla di “one-to-one signaling”.

In questa ultima modalità non c'è differenza nelle prestazioni della comunicazione rispetto all'utilizzo delle mailbox.

Dal momento che il contatore del registro di signaling può assumere al massimo il valore 1, un SPE non può determinare quante scritture siano state effettuate sul registro; in questo caso deve essere configurato opportunamente un interrupt che indichi ogni volta la presenza di un nuovo segnale.

4.3 Creazione e gestione thread in esecuzione su SPE

La creazione e la gestione dei thread che devono essere eseguiti su SPE è effettuata tramite la libreria SPE Runtime Management Library (libspe) utilizzata nel codice in esecuzione su PPE. Questa libreria fornisce una API (Application Programming Interface) che permette l'accesso alle risorse degli SPE. In generale infatti le applicazioni non hanno il controllo diretto dell'hardware di un SPE, che è invece gestito dal sistema operativo in esecuzione sul PPE, ma possono utilizzare particolari costrutti software per la gestione di un SPE detti “contesti SPE”. Un contesto è una rappresentazione logica di un SPE e costituisce l'oggetto base sul quale agisce la libreria libspe. Il sistema operativo schedula i contesti dell'applicazione in esecuzione sulle risorse fisiche degli SPE in accordo con la politica di scheduling scelta.

Lo schema base con un'applicazione che utilizza un solo SPE è il seguente:

1. Creare un contesto SPE.
2. Caricare un eseguibile SPE risultante dalla compilazione del codice SPE nel rispettivo LS
3. Eseguire il contesto SPE. Questa operazione trasferisce il controllo al sistema operativo che effettua la schedulazione del contesto nell'SPE.
4. Cancellare il contesto precedentemente creato.

Nel caso in cui invece l'applicazione necessiti l'utilizzo di più SPE simultaneamente, allora si dovranno creare tanti thread quanti sono i contesti SPE richiesti; ogni thread si occuperà dell'esecuzione di un contesto.

Se sono richiesti N contesti concorrenti, allora si dovrebbero creare un thread principale più N threads dedicati all'esecuzione dei rispettivi contesti.

5 Codifica e standardizzazione delle chiamate di sistema per la sincronizzazione

5.1 Sincronizzazione PPE – SPE

Supponiamo di avere due thread in esecuzione rispettivamente sul PPE e su un SPE qualsiasi del CBE. Il thread PPE, che chiameremo thread principale, ha il compito di creare il thread SPE e gestirne l'esecuzione oltre a inizializzare i dati richiesti dall'applicazione. Tramite DMA l'SPE carica nel proprio LS i dati letti dal main-storage e svolge le operazioni previste dal codice in esecuzione su di esso (fase di computazione); completata questa fase esegue nuovamente un DMA con destinazione main-storage per scrivere i risultati in memoria centrale. Il thread principale legge poi i dati appena scritti per verificarne la correttezza. Si presenta però subito un problema: l'esecuzione concorrente dei due thread non garantisce che la lettura in memoria del thread principale venga effettuata successivamente al DMA dei risultati nel main-storage da parte

dell'SPE. Ciò comporta evidentemente una lettura di dati errati in quanto non ancora aggiornati. Graficamente si presenta la situazione schematizzata in Figura 3.

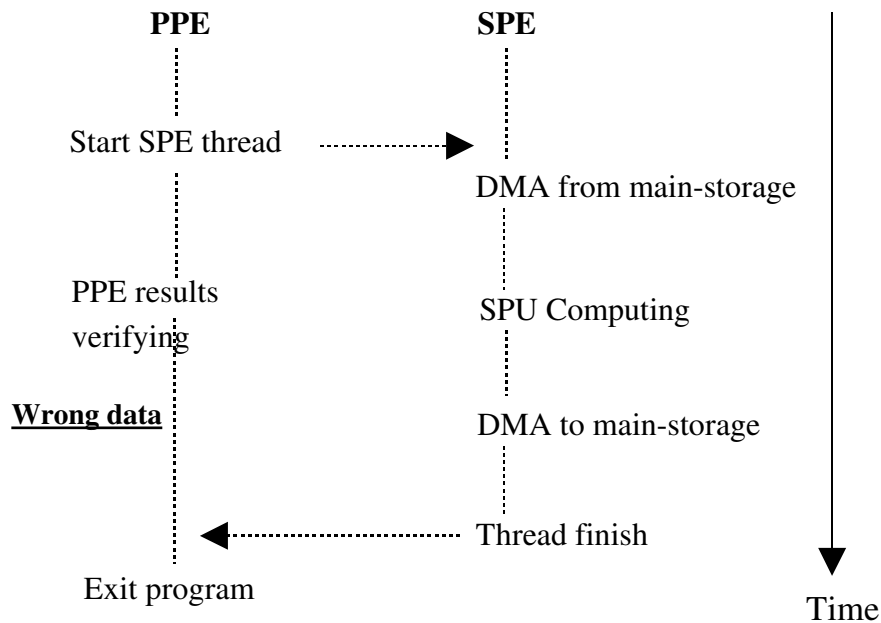


Figura 3 – Sequenza di esecuzione thread senza sincronizzazione

Per evitare ciò è necessaria una sincronizzazione tra PPE e SPE.

Una prima forma di sincronizzazione dovrebbe portare alla situazione di Figura 4 nella quale il thread PPE rimane in attesa fino a quando l'SPE non ha eseguito il DMA nel main-storage per scrivere i dati corretti.

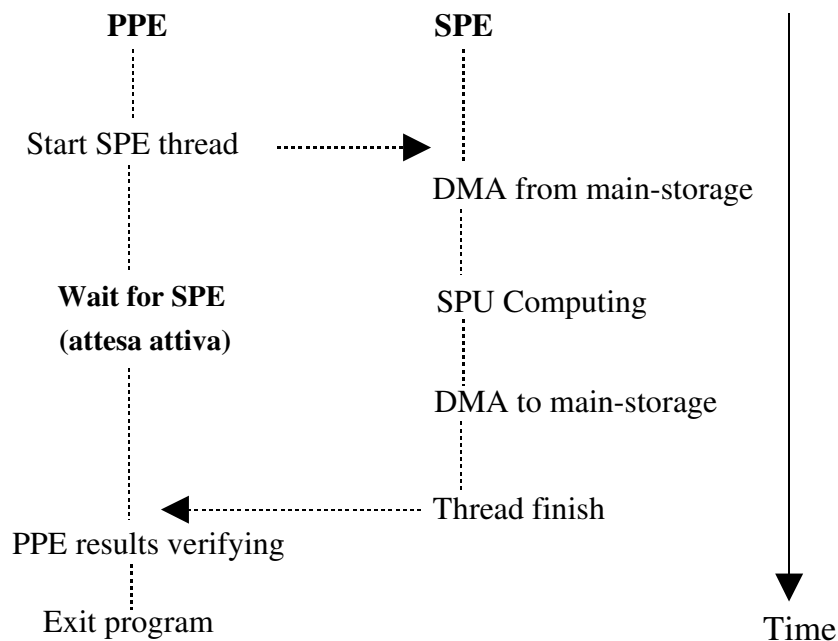


Figura 4 – Sequenza di esecuzione thread con sincronizzazione

Per realizzare ciò si può utilizzare il meccanismo delle mailbox tra PPE e SPE.

L'architettura CBE mette a disposizione del programmatore diverse funzioni per la gestione delle mailbox sia nel PPE che nell'SPE. Per il PPE sono le seguenti:

spe_in_mbox_status – Controlla lo stato della mailbox in entrata di un SPE. Ritorna il numero di messaggi disponibili per la scrittura; in caso di mailbox piena ritorna un valore pari a zero.

spe_in_mbox_write – Scrive uno o più messaggi sulla mailbox in entrata di un SPE. E' possibile scegliere un comportamento bloccante o non-bloccante.

spe_out_mbox_status – Controlla lo stato della mailbox di uscita di un SPE ritornando il numero di messaggi presenti.

spe_out_mbox_read – Legge uno o più messaggi dalla mailbox di uscita di un SPE. Ritorna il numero di messaggi letti

Per l'SPE sono disponibili le seguenti funzioni:

spu_stat_in_mbox – Controlla lo stato della mailbox in entrata e ritorna il numero di messaggi ancora disponibili. Ritorna zero se la mailbox è piena.

spu_read_in_mbox – Legge la mailbox in entrata e ritorna il valore letto. E' bloccante nel caso in cui la mailbox sia vuota.

spu_stat_out_mbox – Controlla lo stato della mailbox di uscita e ritorna il numero di messaggi ancora disponibili.

spu_write_out_mbox – Scrive un messaggio sulla mailbox di uscita. E' bloccante nel caso di mailbox piena.

5.1.1 Sincronizzazione con PPE polling

Per ottenere la sequenza di esecuzione riportata in Figura 4 nella quale il PPE effettua un polling sul bus rimanendo in attesa di una comunicazione da parte dell'SPE, si possono utilizzare le funzioni PPE *spe_out_mbox_status*, *spe_out_mbox_read* e la funzione SPE *spe_write_out_mbox*.

Nel codice PPE un ciclo while si occupa di controllare lo stato della mailbox di uscita SPE tramite la funzione *spe_out_mbox_status*.

Codice PPE :

```
int main () {  
  
    Inizializzazioni e creazione thread;  
  
    while (spe_out_mbox_status(SPE_thread) == 0); //Attesa attiva del thread PPE  
    spe_out_mbox_read(SPE_thread);  
  
    Verifica risultati;  
  
}
```

Codice SPE :

```
int main () {  
  
    Trasferimento DMA main-storage -> LS;  
  
    Computazione;  
  
    Trasferimento DMA LS -> main-storage;  
  
    spu_write_out_mbox(flag); //Sblocca il thread PPE  
  
}
```

Il ciclo si interrompe quando il valore ritornato dalla funzione è diverso da zero indicandoci che l'SPE ha terminato il suo trasferimento DMA verso il main-storage e scritto un messaggio nella mailbox di uscita.

Il programma SPE, non appena caricato il contesto nel LS, esegue subito il trasferimento DMA dal main-storage per leggere i dati da elaborare ed effettua la fase di computazione. Nel frattempo il thread PPE è in attesa. Finita la computazione ed effettuato il DMA nel main-storage, il programma scrive un messaggio nella mailbox di uscita che sblocca il thread PPE e permette la verifica dei risultati da parte del PPE.

Nel caso di sincronizzazione di due o più thread dovrà essere modificata la procedura di polling della mailbox di uscita di ogni SPE in modo tale da sequenziare i trasferimenti DMA nel main-storage.

Nel codice PPE avremo un ciclo while come il seguente:

```
int main () {

Inizializzazioni e creazione thread;

while (count != NUM_SPE) {
    for (i=0;i<NUM_SPE;i++) {
        if (spe_out_mbox_status (SPE_thread[i]) != 0) {
            spe_out_mbox_read (SPE_thread[i]);
            if (i != NUM_SPE-1)
                spe_in_mbox_write (SPE_thread[i+1]);
            if (i !=0)
                spe_in_mbox_write (SPE_thread[i-1]);
            count++;
        }
    }
}
```

dove la variabile “count” è utilizzata per memorizzare il numero dei thread SPE completati e la costante NUM_SPE rappresenta il numero massimo di SPE utilizzati dall'applicazione.

Nel codice SPE la gestione dei trasferimenti DMA è più complessa in quanto si deve evitare la scrittura concorrente nel main storage da parte di più thread. Il codice è del tipo:

```
int main () {

Trasferimento DMA main-storage -> LS;

Computazione;

    spu_write_out_mbox(flag); // Segnala al thread PPE l'avvenuta computazione

    spu_read_in_mbox(); // Chiamata bloccante : Il thread SPE aspetta un messaggio dal PPE

    if ((SPU_id != 0)&&(SPU_id !=NUM_SPE-1))
        spu_read_in_mbox(); // Gestione dei thread intermedi con chiamata bloccante

Trasferimento DMA LS -> main-storage;

}
```

Questa forma di sincronizzazione tra PPE e SPE può essere codificata con chiamate standard del tipo “sync” inserite nel codice del PPE alle quali dovranno corrispondere le sequenze di operazioni di comunicazione trattate precedentemente.

La chiamata di sincronizzazione dal lato PPE è della forma:

```
sync_SPE_mailbox (num_SPE, *SPE_thread[0]);
```

dove num_SPE indica evidentemente il numero dei thread SPE che devono essere sincronizzati con il PPE e SPE_thread[0] rappresenta un puntatore al primo thread SPE.

Analogamente nel codice SPE avremo la chiamata:

```
sync_PPE_mailbox (SPU_id);
```

dove SPU_id è l'identificativo dell'SPE che richiede la sincronizzazione con il PPE.

5.1.2 Sincronizzazione con mutua esclusione

Si può utilizzare la sincronizzazione tramite una variabile di mutua esclusione nella situazione in cui un thread (sia PPE che SPE) esegue una sezione critica di codice e gli altri thread devono rimanere in attesa del rilascio di questa variabile. La variabile di mutua esclusione, detta “mutex”, è inizialmente inizializzata a zero e su di essa possono essere eseguite due operazioni, “lock” e “unlock”.

Le routine che implementano queste operazioni sono le seguenti:

```
mutex_lock (mutex_var)
```

Questa routine acquisisce il lock aspettando che la variabile mutex_var assuma il valore zero e successivamente setta a 1 tale variabile. Se il lock è già presente il thread che ha effettuato la chiamata viene messo in stallo.

```
mutex_trylock (mutex_var)
```

Questa routine cerca di acquisire il lock sulla variabile. Se un lock è già presente viene ritornato il valore 0 e il lock non viene acquisito, in caso contrario viene ritornato 1 e il lock viene acquisito.

```
mutex_unlock (mutex_var)
```

Questa routine rilascia il lock presente sulla variabile mutex_var.

mutex_init (mutex_var)

Questa routine inizializza la variabile mutex_var a zero.

Supponiamo di avere un thread PPE e n thread SPE che devono aggiornare i valori di un array di interi denominato “buffer”. Ogni thread SPE cerca di acquisire il lock sulla variabile mutex di mutua esclusione, aggiorna il buffer e rilascia il lock. Allo stesso tempo il PPE tenta l'acquisizione del lock aggiornando il buffer e rilasciando poi il lock.

Ciò può essere ottenuto con il seguente codice:

Codice PPE:

```
int main () {  
  
    mutex_lock (mutex); // acquisizione lock  
  
    if (mutex_trylock(mutex)==0)  
        update_buffer(); // aggiornamento valori di buffer  
  
    mutex_unlock(mutex); // rilascio lock  
  
    wait for SPE threads finish; // attende il completamento dei thread SPE  
    if (mutex_trylock(mutex))  
  
        mutex_unlock(mutex);  
        verify_buffer(); // verifica correttezza dati in buffer  
}
```

Codice SPE:

```
int main () {  
  
    DMA_GET (mutex); // caricamento della variabile mutex nel LS  
    mutex_lock(mutex);  
  
    if (mutex_trylock(mutex))  
        mutex_unlock(mutex);  
  
    DMA_GET (buffer); // caricamento buffer nel LS  
    update_buffer();
```

```
DMA_PUT (buffer); // scrittura nel main-storage di buffer
mutex_unlock(mutex);

return 0;

}
```

5.1.3 Sincronizzazione con variabili condizionali

Le routine qui riportate operano su variabili condizionali. Le due operazioni principali che si possono effettuare sono “wait” e “signal”. Quando un thread esegue una chiamata wait su una variabile condizionale, l'esecuzione è sospesa fino a quando un altro thread esegua una chiamata signal (oppure broadcast) sulla variabile.

cond_broadcast (condition_var)

E' utilizzata per sbloccare tutti i thread in attesa sulla variabile condizionale condition_var. Per sbloccare un singolo thread si utilizza la routine cond_signal.

cond_init (condition_var)

Inizializza la variabile condition_var a 0.

cond_signal (condition_var, mutex_var)

E' utilizzata per sbloccare un thread in attesa sulla variabile condition_var.

cond_wait (condition_var)

Questa routine rilascia il lock sulla variabile mutex e mette in attesa il thread sulla variabile condition_var. Tale thread può essere sbloccato da un'altra chiamata cond_broadcast o cond_signal. Vediamo un esempio in cui il thread PPE inizializza un array di valori detto “buffer”, crea n threads SPE ed esegue una chiamata cond_wait mettendosi in attesa. I thread SPE eseguono le proprie operazioni sull'array “buffer” incrementando ogni volta la variabile contatore denominata “count”; il thread SPE che raggiunge il valore limite COUNT_LIMIT su tale variabile, effettua una chiamata cond_signal che sblocca il thread PPE.

Codice PPE:

```
int main () {

cond_init (cond_var); // Inizializzazione della variabile condizionale
mutex_init (mutex); // Inizializzazione della variabile di mutua esclusione
buffer_initialize();
SPE_thread_create;

mutex_lock (mutex); // Acquisizione lock
while (count < COUNT_LIMIT) {
    cond_wait (cond_var, mutex); // thread PPE in attesa
}
mutex_unlock (mutex); // rilascio lock
verify_buffer ();
}
```

Codice SPE:

```
int main () {

DMA_GET (cond_var,mutex); // caricamento delle variabili cond_var e mutex nel LS
DMA_GET (buffer); // caricamento di buffer nel LS
count ++; // incremento contatore
do_operations();

if (count == COUNT_LIMIT) {
    mutex_lock (mutex);
    cond_signal (cond_var); // Segnala al PPE che i thread SPE hanno eseguito tutte le operazioni
    mutex_unlock (mutex);
}

return 0;
}
```

Vediamo ora l'utilizzo della routine cond_broadcast. In questo caso il PPE crea n threads SPE. Questi eseguono subito una chiamata cond_wait e si mettono in attesa. Nel stesso tempo il PPE setta a 1 la variabile “flag_start” che viene monitorata dai thread SPE per uscire dallo stato di attesa. Successivamente il thread PPE inizializza il buffer ed esegue una chiamata cond_broadcast per sbloccare i thread SPE che eseguono le loro operazioni.

Codice PPE:

```
int main () {  
  
    cond_init (cond_var);  
    mutex_init (mutex);  
  
    SPE thread create;  
  
    buffer_initialize();  
    mutex_lock (mutex);  
    flag_start = 1;  
    cond_broadcast (cond_var);  
    mutex_unlock (mutex);  
  
    wait for SPE thread finish;  
    buffer_verify();  
}
```

Codice SPE:

```
int main () {  
  
    DMA_GET (cond_var,mutex); // caricamento delle variabili cond_var e mutex nel LS  
  
    mutex_lock (mutex);  
    while ( flag_start !=1) {  
        cond_wait (cond_var);  
    }  
    mutex_unlock (mutex);  
  
    DMA_GET (buffer); // caricamento di buffer nel LS  
    do_operations ();  
    DMA_PUT (buffer); // scrittura nel main-storage di buffer  
    return 0;  
}
```

5.1.4 Sincronizzazione con interrupt

Utilizzando il meccanismo di comunicazione delle mailbox è possibile gestire gli interrupt verso il PPE. In particolare si può utilizzare la funzione di scrittura sulla mailbox di uscita **spu_write_out_intr_mbox** per generare un evento interrupt che viene catturato e gestito dal PPE. In questo modo, ad esempio, si può mettere in attesa il PPE dei dati da parte dei vari SPE senza che questo effettui continue richieste sul bus per verificare l'effettivo completamento delle operazioni degli SPE.

Codice PPE:

```
int main () {

    Initializing variables;
    Create SPE thread;

    while (count_event < NUM_SPU) {
        if (receive_interrupt) {
            count_event++;
            spu_in_mbox_write (SPE_thread);
        }
    }
}
```

La variabile `count_event` memorizza il numero degli eventi ricevuti e per ogni interrupt ricevuto da SPE sblocca il relativo thread scrivendo nella mailbox d'ingresso.

Codice SPE:

```
int main () {

    Initializing variables;
    DMA_GET (data); // caricamento dati nel LS
    Computation on data;

    spu_write_out_intr_mbox(); // genera l'evento interrupt
    spu_read_in_mbox(); // blocca il thread in attesa del segnale da PPE

    DMA_PUT (data);
    return 0;
}
```

Anche in questo caso si può prevedere una standardizzazione della sincronizzazione definendo una funzione “sync_intr” alla quale dovrà corrispondere la sequenza di operazioni sopra definita sia nel PPE che nell'SPE. Sarà dunque del tipo:

```
sync_intr (NUM_SPE);
```

dove NUM_SPE indica il numero di SPE utilizzati.

5.2 Sincronizzazione SPE – SPE

La sincronizzazione tra due o più thread SPE effettuata senza l'ausilio del PPE viene effettuata prevalentemente tramite il metodo di comunicazione del signaling.

Supponiamo di avere 4 thread SPE in esecuzione che aggiornano ognuno un array di valori detto “buffer” e di voler ordinare l'esecuzione di tali thread in modo sequenziale, cioè SPE 0, SPE 1, SPE 2 e SPE 3.

Si può utilizzare il codice SPE seguente:

```
int main () {  
  
    switch (SPU_id) {  
        case 0:  
            buffer_reset();  
            buffer_update();  
            dma_put(SPU_id+1);  
            sendsig(SPU_id+1);  
            spu_read_signal1();  
            check_buffer();  
            break;  
  
        case 1:  
            spu_read_signal1();  
            buffer_update();  
            dma_put(SPU_id+1);  
            sendsig(SPU_id+1);  
            break;  
  
        case 2:  
            spu_read_signal1();  
            buffer_update();  
            dma_put(SPU_id+1);
```

```

sendsig(SPU_id+1);
break;

case 3:
    spu_read_signal1();
    buffer_update();
    dma_put(0);
    sendsig(0);
    break;
}

```

I 4 thread iniziano l'esecuzione in modo concorrente ma è il thread con SPU_id = 0 ad avere la precedenza. Infatti gli altri thread effettuano una lettura sul registro di signaling settato a 0 (poichè non ci sono segnali pendenti) e per questo rimangono in stato di stallo in attesa di una scrittura su tale registro. Il thread 0 effettua dunque il reset del buffer, aggiorna i valori e li scrive nel LS associato al thread 1 tramite DMA put; successivamente con una scrittura sul registro di signaling del thread 1 (sendsig (SPU_id+1)) permette a tale thread di uscire dallo stallo ed eseguire le proprie operazioni, cioè l'update del buffer e la scrittura dei dati aggiornati nel LS del thread 2.

Le stesse operazioni verranno poi eseguite per sbloccare il thread 2 e 3 e infine il thread 0 iniziale che verificherà la correttezza dei valori del buffer.

6 Codifica e standardizzazione delle chiamate di sistema per il trasferimento dati tramite DMA

6.1 Trasferimenti DMA RAM -> LS e LS -> RAM

E' possibile iniziare trasferimenti DMA sia dal PPE che da un SPE, tuttavia è consigliabile scegliere la seconda modalità in quanto la MFC Command Queue di un SPE può contenere fino a 16 elementi che corrispondono ognuno a un comando DMA al contrario del PPE che ne può gestire solamente 8.

Inoltre il tempo di setup richiesto da un trasferimento DMA eseguito in un SPE è minore di quello richiesto dal PPE.

Le funzioni per i trasferimenti DMA messe a disposizione dal MFC di un SPE sono le seguenti:

mfc_get (LS_address, main_storage_address, dma_size, tag_id, tid, rid)

Muove la quantità di dati indicata da dma_size dal main_storage al LS di un SPE. Il tag_id rappresenta l'identificativo del Tag Group associato al trasferimento; i due campi "tid" e "rid" rappresentano il Transfer Class Identifier e il Replacement Class Identifier e normalmente sono entrambi settati a 0.

mfc_put (LS_address, main_storage_address, dma_size, tag_id, tid, rid)

Muove la quantità di dati indicata da dma_size dal LS di un SPE al main_storage.

Ad ognuna di queste funzioni può essere associato lo specificatore “fence” o “barrier”, come precedentemente descritto, per ottenere le seguenti funzioni: mfc_getf, mfc_getb, mfc_putf, mfc_putb

Per completezza riportiamo di seguito le funzioni per effettuare trasferimenti DMA da PPE.

spe_mfcio_get (spe_context , LS_address, main_storage_address, dma_size, tag_id, tid, rid)

Inserisce un comando DMA get nel MFC dell'SPE associato a spe_context per muovere la quantità di dati indicata da dma_size da main-storage a LS.

spe_mfcio_put (spe_context , LS_address, main_storage_address, dma_size, tag_id, tid, rid)

Inserisce un comando DMA put nel MFC dell'SPE associato a spe_context per muovere la quantità di dati indicata da dma_size da LS a main-storage.

Anche in questo caso è possibile aggiungere gli specificatori fence e barrier per ottenere gli stessi tipi di funzioni sopra indicate.

Per standardizzare le chiamate a funzioni per trasferimenti DMA tra RAM e LS possiamo definire le seguenti funzioni utilizzabili nel codice SPE che corrispondono alle due definite precedentemente.

SPE_dma_read (LS_address, main_storage_address, dma_size, tag_id)

SPE_dma_write (LS_address, main_storage_address, dma_size, tag_id)

omettendo i due campi tid e rid normalmente settati a zero.

Analogamente possiamo definire due funzioni per le chiamate a DMA effettuate dal PPE, come le seguenti:

PPE_dma_read (LS_address, main_storage_address, dma_size, tag_id)

PPE_dma_write (LS_address, main_storage_address, dma_size, tag_id)

Tuttavia a differenza delle chiamate DMA effettuate da un SPE, quelle eseguite dal PPE si limitano a inserire il comando DMA nella Proxy Command Queue dell'SPE selezionato che effettivamente inizializza e gestisce il trasferimento DMA.

6.2 Trasferimenti DMA LS <-> LS

Nello stesso modo in cui vengono effettuati i trasferimenti DMA tra RAM e LS, è possibile trasferire dati tra i LS degli SPE specificando come sorgente l'indirizzo del LS da cui leggere i dati e come destinazione l'indirizzo del LS dove scrivere i dati. E' tuttavia necessario che sia il PPE a comunicare ad ogni SPE l'indirizzo del LS dell'SPE verso cui effettuare il DMA, in quanto ai singoli SPE non è permesso accedere direttamente a queste informazioni.

Le due funzioni utilizzabili per trasferimenti LS <-> LS sono essenzialmente la `mfc_get` e `mfc_put` sopra descritte. Nel caso, ad esempio, di comunicazione tra due SPE, con la quale SPE 1 scrive nella memoria LS di SPE 2 è sufficiente scrivere:

`mfc_put (LS_addr_SPE1, LS_addr_SPE2, dma_size, tag_id, tid, rid)`

Viceversa se è SPE2 a leggere dal LS di SPE1 avremo:

`mfc_get (LS_addr_SPE2, LS_addr_SPE1, dma_size, tag_id, tid, rid)`

Lo standard per queste due funzioni può essere rappresentato dalle seguenti:

`SPE_LS_dma_write (LS_addr_SPE1, LS_addr_SPE2, dma_size, tag_id)`

`SPE_LS_dma_read (LS_addr_SPE2, LS_addr_SPE1, dma_size, tag_id)`