

ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Facoltà di Ingegneria

Corso di Laurea in INGEGNERIA INFORMATICA

Progetto di CALCOLATORI ELETTRONICI M

Progetto di una memoria cache per il processore DLX

Componenti Gruppo:

Andrea Grandi

Filippo Malaguti

Massimiliano Mattetti

Gabriele Morlini

Thomas Ricci

Anno Accademico 2009/2010

Indice

Introduzione	5
1 Caratteristiche della memoria cache	7
1 Politica di rimpiazzamento	8
2 Struttura e interfacce	9
2 Realizzazione del componente	15
1 Strutture dati	15
2 Implementazione	17
2.1 cache_dlx	17
2.2 cache_ram	19
2.3 cache_snoop	19
2.4 cache_replace	20
2.5 Comunicazione tra processi	20
3 Procedure interne	22
3.1 cache_hit_on	22
3.2 cache_inv_on	23
3.3 get_way	23
3 Integrazione con DLX	25
1 Modifiche al Memory_stage	25
2 Connessione del componente	27
2.1 Istruzione load	28
2.2 Istruzione store	29
4 Testbench	31
1 Testbench del componente	31

1.1	Cache_test_ReadAndWrite.vhd	32
Fase 1: Letture d' inizializzazione	32	
Fase 2: Scritture	32	
Fase 3: Letture di verifica	34	
1.2	Cache_test_ReadAndReplacement.vhd	34
Fase 1: Letture di inizializzazione	35	
Fase 2: Invalidazione	35	
Fase 3: Verifica meccanismo contatori	35	
1.3	Cache_test_snoop.vhd	36
2	Assembler per DLX	37
2.1	Dal codice all'esecuzione	38
2.2	provaReplacement123.vhd	39
2.3	provaFU.vhd	40
5	Block RAM	43
1	Caratteristiche e segnali della Block Ram	44
2	Configurazione della Block RAM	47
2.1	Inizializzazione del contenuto e attributi di configuraione	49
3	Operazioni della Block RAM	54
4	Conflitti d'accesso in Block RAM Dual-Port	55
5	Utilizzo della Block Ram in un progetto su FPGA	58
6	Progetto d'esempio: BlockRam_cmp	61
1	Specifiche del progetto	61
2	Implementazione	62
2.1	ram_cache	64
2.2	blockram_sequential_access	65
2.3	end_blockram_access	68
3	Testbench	68
4	Considerazioni e Sviluppi Futuri	70
Conclusioni		73
Bibliografia		75

Introduzione

Un processore durante il suo funzionamento accede in scrittura o in lettura ai dati presenti nella memoria a valle: tale operazione richiede tipicamente diversi cicli di clock che costringono il processore (più veloce della memoria) ad attendere il dato. Ciò comporta l'introduzione di cicli di wait che ovviamente causano un peggioramento delle performance del processore, il quale attende che la memoria gli presenti il dato richiesto segnalato dal segnale di ready. Per superare tale problema si utilizzano pertanto delle memorie cache, vicine al processore, di piccole dimensioni e molto veloci (tanto che possono avere tempi d'accesso simili a quelli dei registri interni al processore) da cui vengono reperiti i dati necessari all'esecuzione, consentendo in caso di HIT, ovvero nel caso in cui il dato si trovi in cache, di recuperarlo quasi senza ritardo.

Le cache si posizionano nella gerarchia delle memorie (insieme ai registri) tra i livelli più prossimi al processore e ciò comporta da un lato la rapidità nell'accesso e dall'altro le dimensioni limitate che fanno sì che una cache contenga un subset delle linee di memoria del dispositivo a valle (memoria o un livello superiore di cache se presente).

Pertanto quando si accede a una cache possono verificarsi due casi:

1. il dato si trova nella cache (HIT);
2. il dato non è presente e deve essere recuperato da un dispositivo a valle (MISS).

Ovviamente in caso di MISS si deve pagare un costo temporale per il reperimento del dato assente, detto *miss penalty*, dato dalla somma del tempo d'accesso al dispositivo a valle e dal tempo di trasferimento della linea col dato cercato.

Ciononostante, è dimostrato che l'hit rate e quindi l'efficienza delle cache è tipicamente molto alta (oltre il 95%) grazie alla validità del Principio di Località spaziale e temporale, per il quale un programma in esecuzione tende ad eseguire temporalmente istruzioni eseguite di recente e ad accedere a dati acceduti di recente. Quindi sulla base di tali considerazioni, l'uso di cache contenenti le linee di memoria più recentemente accedute (working set) consente di migliorare notevolmente il tempo di reperimento dei dati necessari all'esecuzione, evitando quindi i ritardi che si avrebbero per ogni accesso diretto in memoria.

Abbiamo scelto questo progetto per approfondire le tematiche e le problematiche legate alla progettazione di un componente cache da affiancare al processore DLX visto a lezione.

In particolare l'attività di progetto svolta si prefigge i seguenti obiettivi:

1. **Realizzazione memoria cache:** progetto di un component VHDL che realizza il funzionamento di una memoria cache generica.
2. **Integrazione con DLX:** modifica del progetto DLX per consentire l'integrazione del component realizzato con il processore
3. **Testbench del component:** progetto di una suite di test per il component.
4. **Block RAM:** analisi del funzionamento della RAM integrata all'interno della FPGA.

Capitolo 1

Caratteristiche della memoria cache

Si è scelto di progettare una cache di tipo set-associative, la cui schematizzazione è mostrata in Fig. 1.1. Questa tipologia di cache rappresenta un buon compromesso tra flessibilità e costo in termini di silicio.

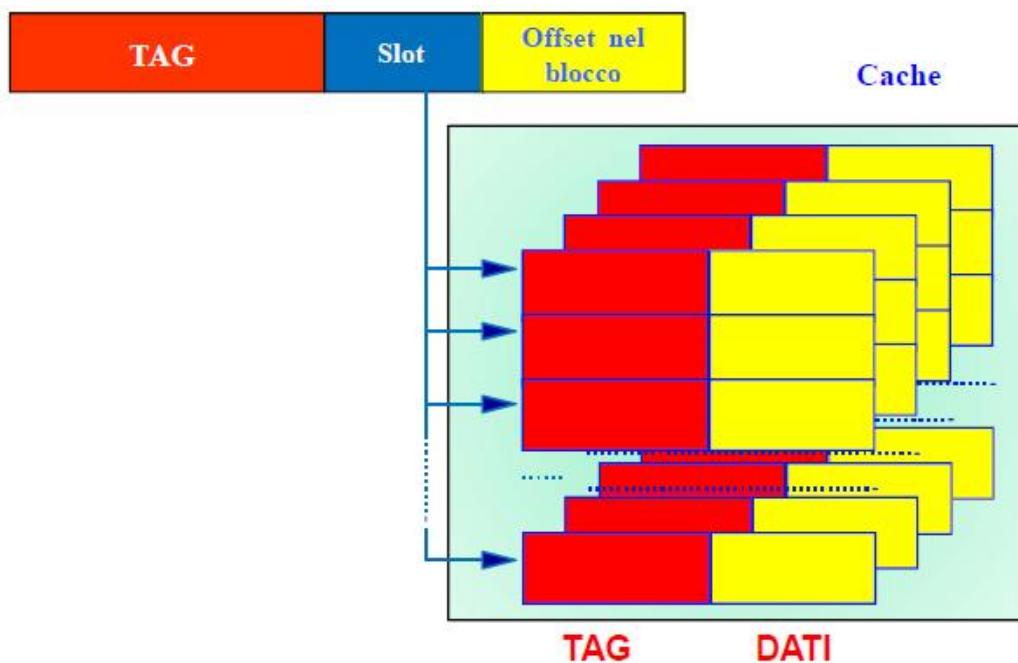


Figura 1.1: Schematizzazione di una cache set-associativa

L'indirizzo di partenza del blocco è diviso in TAG (parte alta), INDEX e OFFSET (parte bassa). Il TAG consente di identificare univocamente una linea all'interno di un sottoinsieme di linee, detto SET. L'INDEX individua immediatamente il SET all'interno del quale è possibile recuperare la linea corrente tramite il confronto del TAG. In questo modo si limita il numero di confronti tra TAG accettando il fatto che ogni linea possa appartenere ad un singolo set. La parte meno significativa dell'indirizzo rappresenta l'OFFSET che consente di individuare il dato all'interno di una linea.

Per garantire maggiore flessibilità si è scelto di parametrizzare alcune delle caratteristiche statiche della cache, quali ad esempio:

- la dimensione dei blocchi
- il numero di vie
- il numero di linee

1 Politica di rimpiazzamento

Nel caso in cui si debba caricare una nuova linea e tutte le vie siano occupate è necessario determinare quale linea rimpiazzare. Un buon algoritmo di rimpiazzamento dovrebbe cercare di individuare la linea vittima che meno probabilmente verrà riutilizzata in seguito.

Il criterio scelto per effettuare il rimpiazzamento è basato su contatori, che implementa una politica LRU (Least Recently Used). Tale politica è tipicamente implementata poiché statisticamente si verifica il principio di località. È quindi presente un contatore per ogni via di ogni set tramite il quale si tiene traccia di quanto recentemente si è acceduti a ciascuna linea: un valore basso del contatore indica un accesso recente mentre un valore alto indica un accesso *vetero*. Evidentemente la linea candidata al rimpiazzamento risulta essere quella alla quale è associato il contatore di valore più elevato.

Nel caso di HIT su una linea, sono incrementati i contatori di valore più basso rispetto al valore di quello della linea HIT mentre quest'ultimo viene resettato. Nel caso di MISS si procede con un rimpiazzamento e poi si agisce come nel caso di HIT sulla nuova linea. Infine, in caso di invalidazione di una linea, si porta al valore massimo il contatore della linea invalidata e si decrementano di 1 tutti i contatori con valore più elevato di quello della linea invalidata.

2 Struttura e interfacce

La memoria cache si interfaccia con i dispositivi esterni attraverso 4 tipi di interfacce, come mostrato in Fig. 1.2.

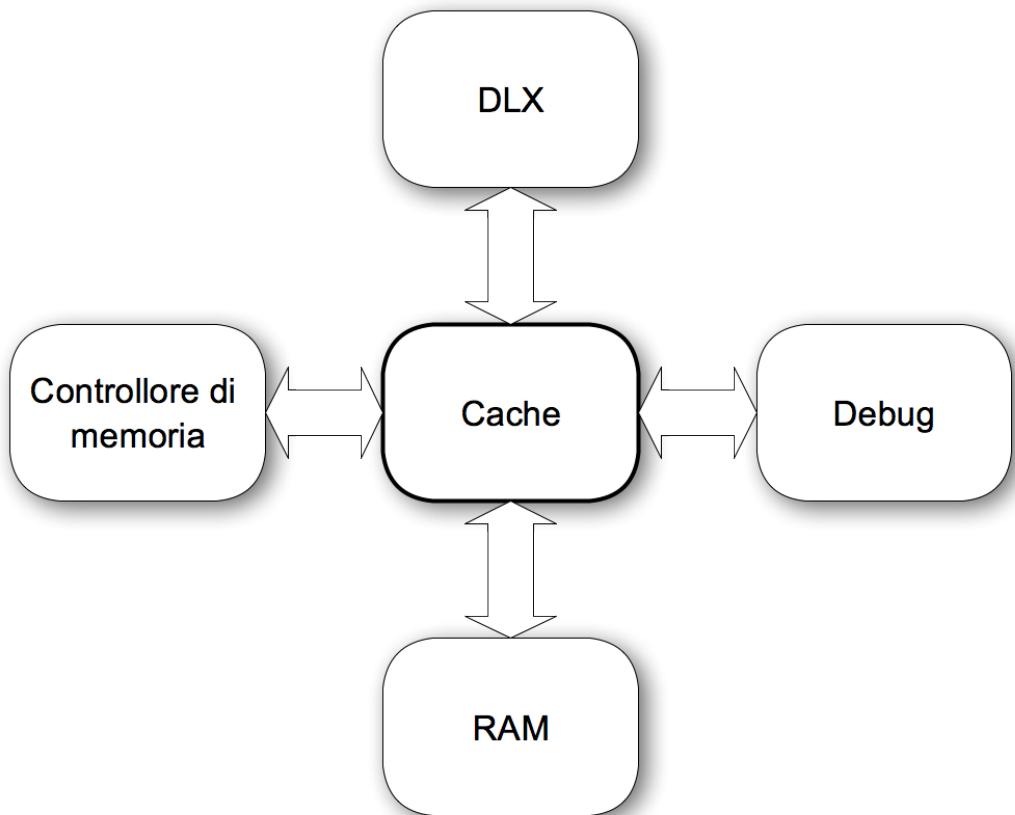


Figura 1.2: Interfacce della memoria cache

L'interfaccia verso il microprocessore, mostrata in Fig. 1.3, consente a quest'ultimo di accedere ai dati memorizzati all'interno della cache.

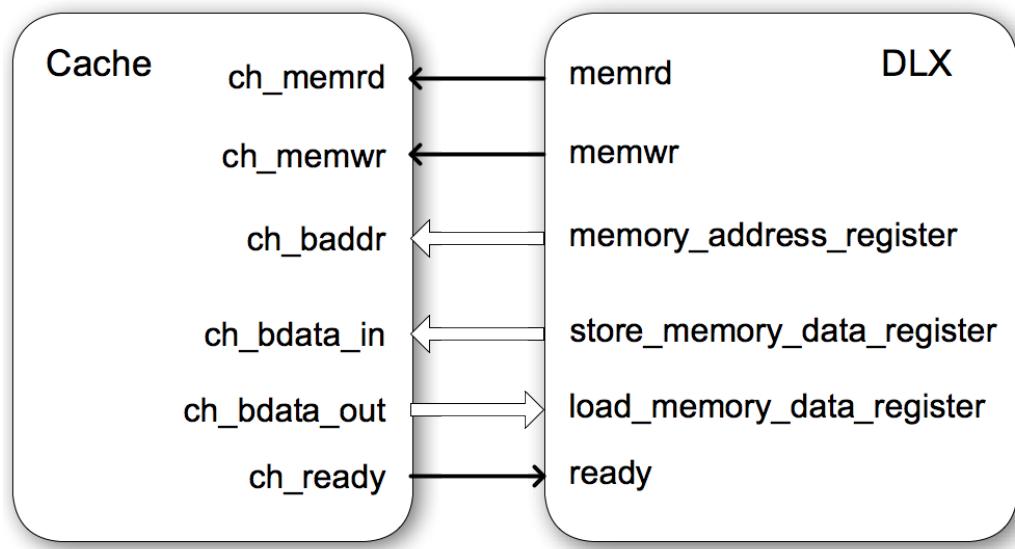


Figura 1.3: Interfaccia della memoria cache verso il processore DLX

In particolare sono presenti i seguenti segnali:

- **ch_baddr(31-2)**: indirizzi a 32 bit emessi dal microprocessore
- **ch_bdata(31-0)**: bus dati con parallelismo 32 bit
- **ch_memwr**: segnale per il comando di scrittura in cache
- **ch_memrd**: segnale per il comando di lettura da cache
- **ch_ready**: segnale che indica il termine dell'operazione di lettura/scrittura corrente

Per quanto riguarda gli scambi di dati tra processore e memoria cache, si ipotizza che siano sempre lette e scritte parole di lunghezza fissa a 32 bit.

Anche se la memoria cache progettata non verrà impiegata in sistemi multmaster, si è comunque deciso di affrontare alcune delle

problematiche derivanti dalla presenza di un controllore di memoria. Tramite l'opportuna interfaccia è ad esempio possibile effettuare l'invalidazione delle linee e lo snooping dei dati presenti in cache.

L'interfaccia verso il controllore di memoria, mostrata in Fig. 1.4, consente di testare e modificare lo stato delle linee.

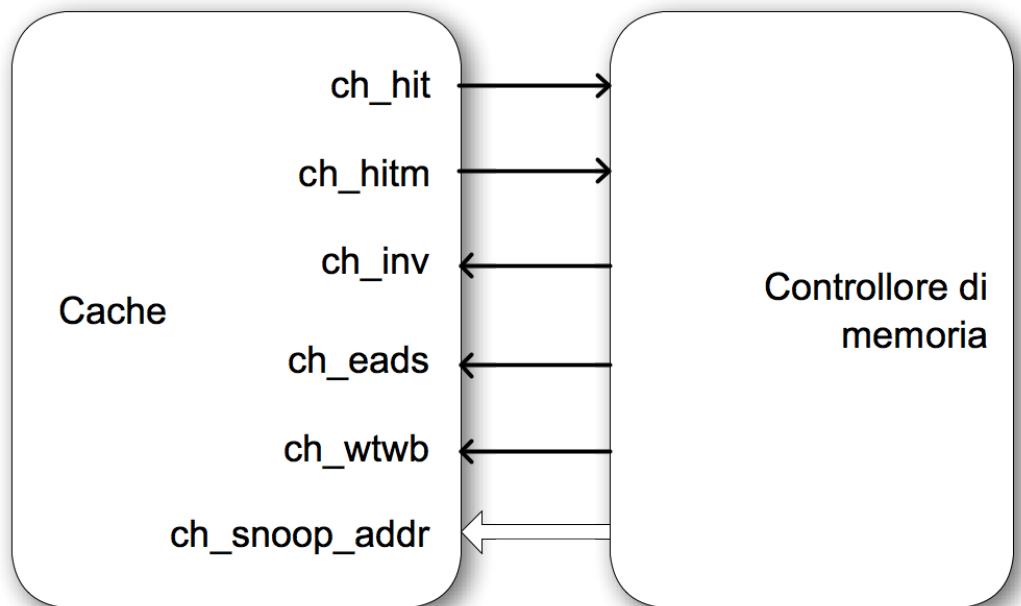


Figura 1.4: Interfaccia della memoria cache verso il controllore di memoria

In particolare sono presenti i seguenti segnali:

- **ch_eads**: inizia il ciclo di snoop
- **ch_inv**: richiede l'invalidazione della linea
- **ch_hit**: indica che la linea richiesta è presente in memoria
- **ch_htm**: indica che la linea richiesta è presente in memoria in stato modified
- **ch_wtwb**: indica se la linea appena letta deve essere caricata in stato MESI_S oppure MESI_E

- **ch_snoop_addr**: indirizzo al quale effettuare lo snoop

L'interfaccia verso la RAM è mostrata in Fig. 1.5 e consente alla cache di recuperare i blocchi dal livello sottostante.

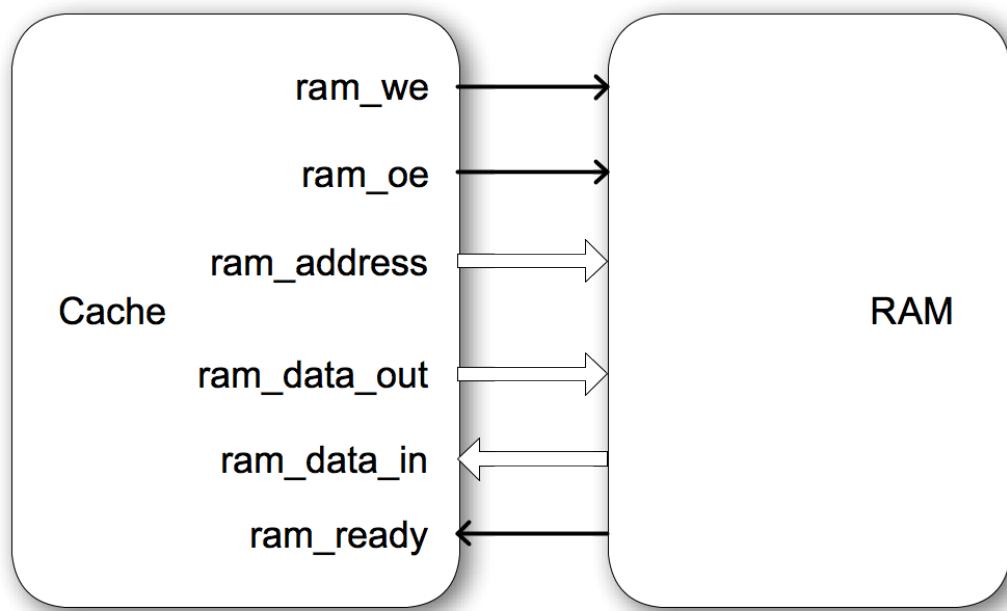


Figura 1.5: Interfaccia della memoria cache verso la RAM

In particolare sono presenti i seguenti segnali:

- **ram_address(31-2)**: indirizzi a 32 bit emessi dalla cache
- **ram_data_out**: bus dati di uscita con parallelismo pari alla dimensione di una linea
- **ram_data_in**: bus dati di ingresso con parallelismo pari alla dimensione di una linea
- **ram_we**: segnale per il comando di scrittura in RAM
- **ram_oe**: segnale per il comando di lettura dalla RAM
- **ram_ready**: segnale che indica il termine dell'operazione di lettura/scrittura corrente

È presente infine una quarta interfaccia verso l'esterno, utilizzata per monitorare lo stato interno della cache e poter quindi eseguire il debug.

Tale interfaccia, mostrata in Fig. 1.6, non è indispensabile per il corretto funzionamento del dispositivo.

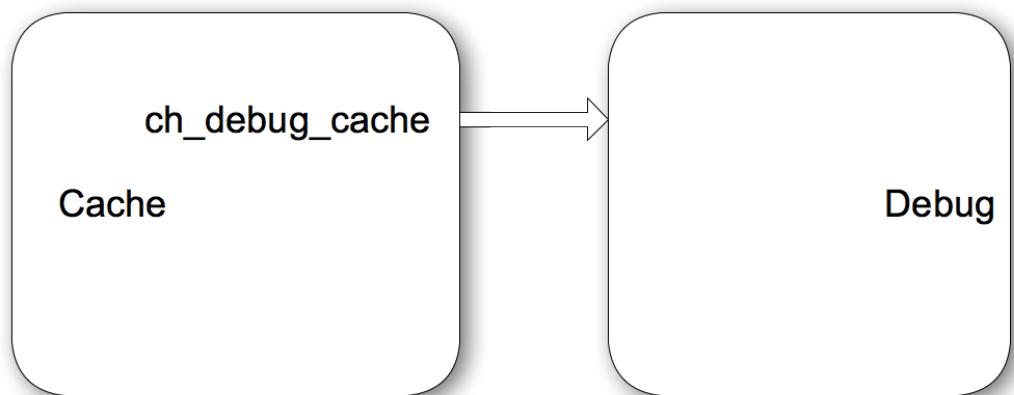


Figura 1.6: Interfaccia utilizzata per il debug della memoria cache

Il segnale `ch_debug_cache` rende visibile all'esterno la struttura dati in cui sono memorizzate le informazioni contenute in cache. Le strutture dati saranno descritte nel dettaglio nel capitolo successivo.

Capitolo 2

Realizzazione del componente

La cache è stata realizzata come componente indipendente, detto Cache_cmp. In questo capitolo saranno mostrate le caratteristiche principali di tale componente.

1 Strutture dati

I tipi di dato utilizzati sono definiti nel file Cache_lib.vhd.

Listing 2.1: Costanti e tipi di dato definiti nel file Cache_lib.vhd

```
CONSTANT OFFSET_BIT : natural := 5;
CONSTANT INDEX_BIT : natural := 2;
CONSTANT TAG_BIT : natural := PARALLELISM - INDEX_BIT -
    OFFSET_BIT;
CONSTANT NWAY : natural := 2;

CONSTANT MESI_M : natural := 3;
CONSTANT MESI_E : natural := 2;
CONSTANT MESI_S : natural := 1;
CONSTANT MESI_I : natural := 0;

TYPE data_line IS ARRAY (0 to 2**OFFSET_BIT - 1) OF
    STD_LOGIC_VECTOR (7 downto 0);
```

```

TYPE cache_line IS
  RECORD
    data : data_line;
    status : natural;
    tag : STD_LOGIC_VECTOR (TAG_BIT-1 downto 0);
    lru_counter : natural;
  END RECORD;

TYPE set_ways IS ARRAY (0 to NWAY - 1) of cache_line;

TYPE cache_type IS ARRAY (0 to 2**INDEX_BIT - 1) of
  set_ways;

TYPE RAM IS ARRAY (integer range <>) of data_line;

```

Il numero di bit di offset, indice e tag è stato parametrizzato per rendere più flessibile l'utilizzo del componente.

All'interno di Cache_lib.vhd sono poi stati definiti i seguenti tipi di dati:

- **data_line**: contiene i dati per una linea della cache, la cui dimensione è calcolata in base al numero di bit di offset;
- **cache_line**: record contenente le informazioni su dati e stato di una linea;
- **set_ways**: array di NWAY vie che compongono un set al cui interno saranno collocate le linee;
- **cache_type**: array di set il cui numero dipende dal numero di bit di index e costituisce l'intera cache.

Per ogni **cache_line** si tiene quindi traccia di:

- **data**: **data_line** relativa alla linea corrente;
- **status**: indica lo stato MESI della linea;
- **tag**: bit dell'indirizzo che rappresentano il tag della linea;
- **lru_counter**: contatore usato dalla politica di rimpiazzamento.

In Fig. 2.1 è mostrata una schematizzazione delle strutture dati utilizzate all'interno della cache.

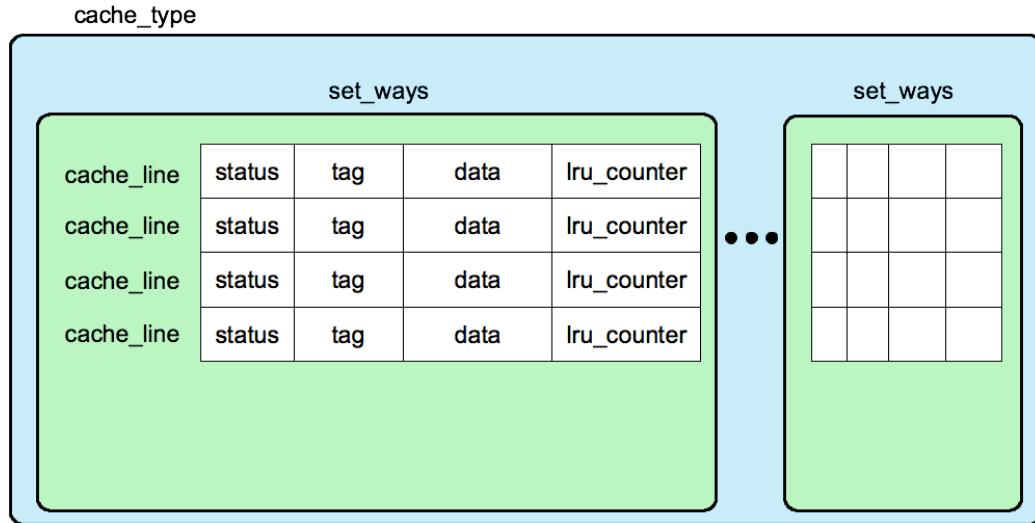


Figura 2.1: Schematizzazione delle strutture dati della cache

2 Implementazione

Il componente Cache_cmp può concettualmente essere diviso in tre parti, ognuna delle quali si interfaccia rispettivamente con DLX, RAM e controllore di memoria.

Per questo motivo si è deciso di implementare il componente con 3 process indipendenti, i quali utilizzano segnali interni per sincronizzarsi, più un quarto processo che si occupa nello specifico di eseguire il rimpiazzamento delle linee.

2.1 cache_dlx

Il process cache_dlx si occupa dell’interfacciamento con il DLX eseguendo le operazioni di lettura e scrittura richieste attraverso gli opportuni segnali di controllo. I compiti di questo process riguardano quindi i seguenti aspetti:

- gestione della lettura di dati dalla cache;
- gestione della scrittura dei dati provenienti dal DLX nella cache;
- attivazione del meccanismo di rimpiazzamento di una linea;

- generazione del segnale di ready per il DLX;

La sensitivity list del processo comprende sia segnali esterni provenienti dal DLX, che segnali interni utilizzati per la sincronizzazione tra i diversi process.

In particolare sono preseti:

- ch_memrd: segnale esterno per una richiesta di lettura;
- ch_memwr: segnale esterno per una richiesta di scrittura;
- ch_reset: segnale esterno per effettuare il reset del contenuto della cache;
- line_ready: segnale interno che indica il termine di un rimpiazzamento;
- rdwr_done: segnale interno che indica, in caso di write-through, il completamento della scrittura in RAM.

I passi seguiti durante una lettura sono:

1. Lettura dell'indirizzo dal bus separando index, tag e offset;
2. Verifica della presenza della linea in cache attraverso
`get_way();`
3. In caso di MISS, attivazione del process per la politica di rimpiazzamento;
4. Aggiornamento dei contatori attraverso `cache_hit_on();`
5. Lettura del dato dalla cache ed emissione sul bus
`ch_bdata_out.`

Per quanto riguarda invece la scrittura, si eseguono le seguenti operazioni:

1. Lettura dell'indirizzo dal bus separando index, tag e offset;
2. Verifica della presenza della linea in cache attraverso
`get_way();`
3. In caso di MISS, attivazione del process per la politica di rimpiazzamento;
4. Scrittura del dato presente in `ch_bdata_in` nella cache;
5. Aggiornamento dei contatori attraverso `cache_hit_on();`
6. Aggiornamento dei bit di stato ed eventuale write-through nel caso di scrittura su linea in stato MESI_S.

2.2 cache_ram

Questo process si occupa dell’interfacciamento con la RAM. In particolare, attraverso segnali interni di controllo, possono essere attivati i meccanismi di scrittura e di lettura di un dato.

Durante la realizzazione si è ipotizzato che fosse disponibile un segnale di `ram_ready` proveniente dall’esterno per indicare il completamento dell’operazione richiesta. Tale segnale è importante poichè le istruzioni all’interno di uno stesso process vengono eseguite in modo parallelo. Nel nostro caso non sarebbe quindi possibile emettere l’indirizzo per la RAM e leggere immediatamente di seguito i dati sul bus `ram_data_in`.

Nel nostro progetto si è supposto che tutti i componenti, compresa la RAM, eseguissero le operazioni in tempo nullo. Tuttavia il segnale `ram_ready` diviene indispensabile nel caso in cui si decida di tenere in considerazione i ritardi introdotti da una RAM reale.

2.3 cache_snoop

Il process `cache_snoop` si attiva con il segnale esterno `ch_eads` proveniente dal controllore di memoria e consente a quest’ultimo di operare sullo stato delle linee.

In particolare è possibile sapere se una determinata linea si trova in cache e se il suo stato è MESI_M.

Tramite il segnale `ch_inv` il controllore di memoria può inoltre forzare l’invalidazione di una particolare linea.

Il process `cache_snoop` ha il seguente comportamento: se l’indirizzo richiesto non è presente in cache i segnali `ch_hit` e `ch_hitm` vengono portati al valore logico ‘0’. In caso contrario il comportamento varia in base allo stato della linea che contiene l’indirizzo:

- stato MESI_E: `ch_hit` viene portato al valore ‘1’ e la linea passa in stato MESI_S;

- stato MESI_S: ch_hit viene portato al valore '1' e lo stato della linea resta invariato;
- stato MESI_M: sia ch_hit che ch_hitm vengono portati al valore '1', viene forzata la scrittura della linea in RAM e il suo stato viene portato a MESI_S.

Nel caso in cui il segnale ch_inv sia attivo il comportamento resta invariato, ma lo stato della linea diventa sempre MESI_I.

2.4 cache_replace

I meccanismi per il rimpiazzamento delle linee sono eseguiti dal process `cache_replace`. In particolare questo process implementa la politica di rimpiazzamento basata sui contatori, stabilendo di volta in volta quale linea rimpiazzare.

Il meccanismo non può eseguire tutte le operazioni in un unico ciclo, quindi per poter effettuare la sostituzione di una linea in cache con dei dati presenti in RAM è stato realizzato un *sequencer* che compie le seguenti operazioni:

1. determina la riga da sostituire;
2. nel caso in cui tale linea sia in stato MESI_M effettua il write-back sulla RAM;
3. attende eventualmente il termine della scrittura;
4. attiva il process per la lettura della nuova linea dalla RAM;
5. attende il termine della lettura;
6. comunica attraverso il segnale interno `line_ready` che il rimpiazzamento è terminato.

2.5 Comunicazione tra processi

I quattro processi si scambiano segnali che consentono la sincronizzazione delle operazioni da svolgere.

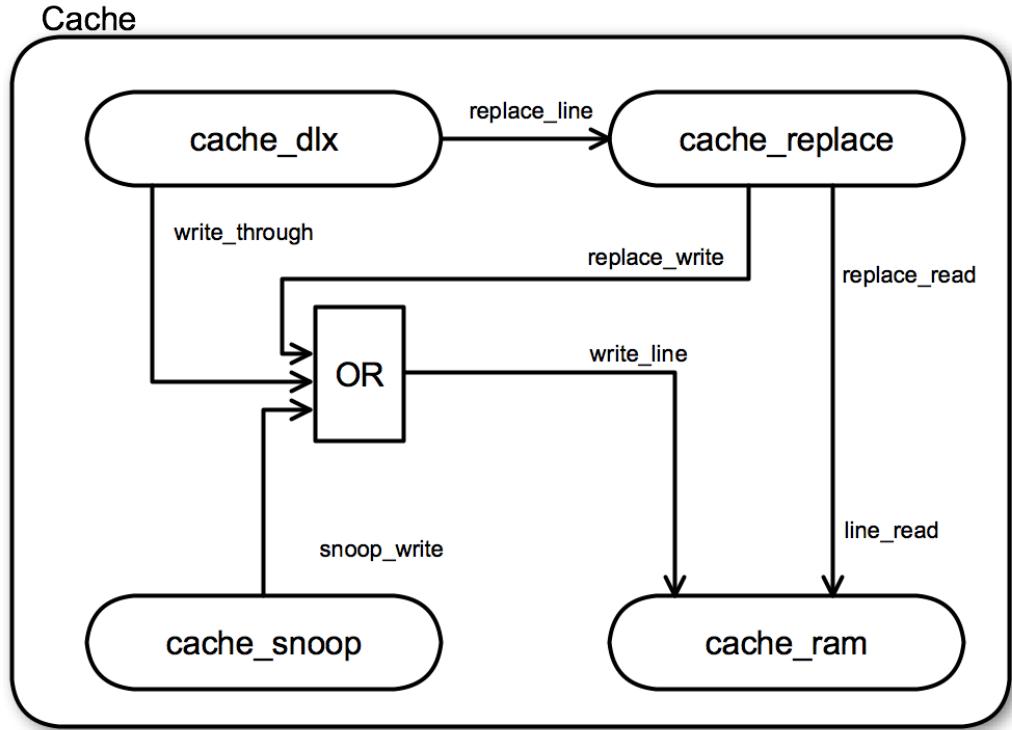


Figura 2.2: Collegamenti tra processi

La Fig. 2.2 mostra come sono collegati i seguenti segnali:

- **replace_line**: attiva il processo che gestisce il rimpiazzamento di una linea;
- **write_through**: attiva la propagazione della modifica di una linea in stato **MESI_S** in memoria RAM;
- **replace_write**: attiva la scrittura di una linea da rimpiazzare in stato **MESI_M** in memoria RAM;
- **snoop_write**: attiva la scrittura di una linea in stato **MESI_M** in memoria RAM in seguito ad uno snoop.

Ogni processo notifica il completamento dell'operazione richiesta attivando un opportuno segnale di ready, come mostrato in Fig. 2.3.

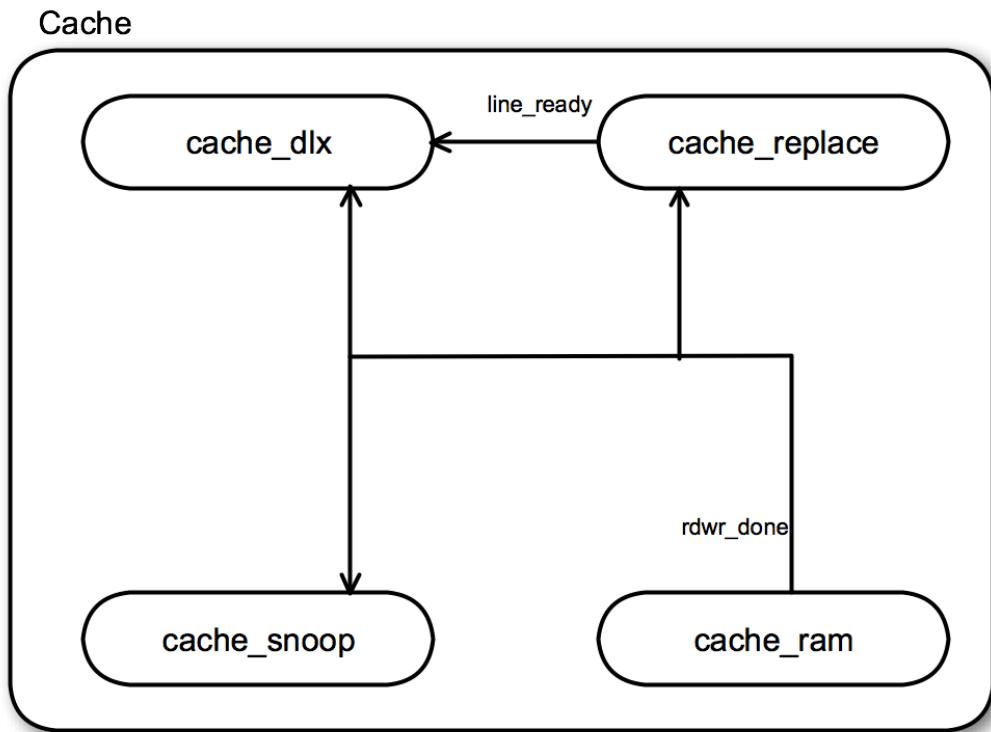


Figura 2.3: Collegamenti tra processi

3 Procedure interne

Di seguito saranno brevemente descritte le procedure invocate all'interno dei diversi processi.

3.1 `cache_hit_on`

Parametri di input:

1. `hit_index`: indice al quale si è verificato l'hit
2. `hit_way`: via nella quale si è verificato l'hit

Descrizione:

Applica la politica di invecchiamento aggiornando i contatori nel set di indice `hit_index`, in particolare:

1. incrementa i contatori di valore più basso della via corrente specificata da `hit_way`

2. resetta il contatore della via corrente su cui c'è hit

3.2 cache_inv_on

Parametri di input:

- inv_index: indice che contiene la via da invalidare
- inv_way: via da invalidare

Descrizione:

Applica la politica di invecchiamento aggiornando i contatori nel set di indice `inv_index`, in particolare:

1. decrementa i contatori di valore più alto della via corrente specificata da `inv_way`
2. porta al valore massimo il contatore della via corrente da invalidare

3.3 get_way

Parametri di input:

1. index: indice
2. tag: tag da controllare

Parametri di output:

- selected_way: via nella quale è presente il dato

Descrizione:

1. Verifica se il dato è in cache, cioè se esiste all'interno del set individuato dall'index una linea con tag uguale a quello specificato il cui stato è diverso da `MESI_I`
2. Se il dato non è presente restituisce `selected_way = -1`
3. Se il dato è presente restituisce il numero della via

Capitolo 3

Integrazione con DLX

In questo capitolo saranno descritte le problematiche affrontate durante l'integrazione del componente cache all'interno del progetto del DLX già esistente.

1 Modifiche al Memory_stage

Nella realizzazione del componente si è supposto che gli accessi alla cache avvengano in un unico ciclo di clock; così facendo si è potuto evitare di modificare la pipeline del DLX per inserire degli stalli ogni qualvolta si trovasse nello stadio di memory una operazione di load o una di store. Le modifiche da apportare al dlx si sono rivolte in questa maniera più esigue e si sono concentrate nel solo stadio di memory essendo la nostra una cache dati.

In particolare sono stati aggiunti al componente Memory_Stage i seguenti segnali:

Listing 3.1: Segnali aggiunti allo stadio di Memory

```
ready: in std_logic;  
memrd: out std_logic;  
memwr: out std_logic;  
load_memory_data_register: in std_logic_vector(PARALLELISM  
-1 downto 0);
```

```
store_memory_data_register: out std_logic_vector(  
    PARALLELISM-1 downto 0);  
memory_address_register: out std_logic_vector(PARALLELISM-1  
    downto 0);
```

È stata inoltre eliminata la variabile Ram.

Le immagini seguenti mostrano come è stato modificato lo schema del DLX in seguito all'integrazione della Cache.

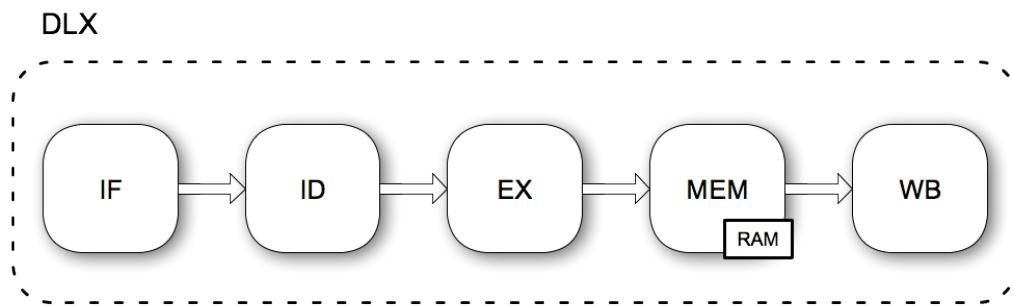


Figura 3.1: Schema della pipeline del DLX originale

La Fig. 3.3 mostra nel dettaglio il nuovo stadio di MEM.

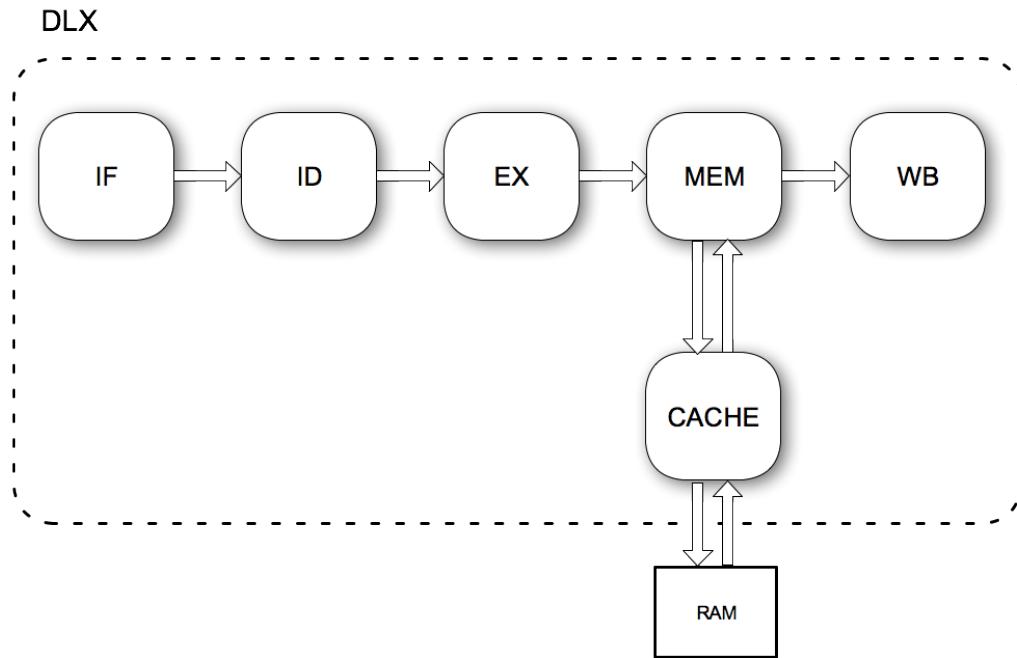


Figura 3.2: Schema della pipeline del DLX con cache integrata

2 Connessione del componente

I segnali `store_memory_data_register` e `load_memory_data_register` sono connessi rispettivamente al `ch_bdata_in` e al `ch_bdata_out` della cache mentre il `memory_address_register` è collegato al bus indirizzi della cache (`ch_baddr`).

Il segnale `ready` (connesso al `ch_ready` della cache) viene asserito alla fine di ogni ciclo di lettura e scrittura ed indica al processore che il dato proveniente dalla cache è disponibile per la lettura o che la scrittura è terminata e può avere luogo un nuovo ciclo di bus.

I rimanenti due segnali `memrd` e `memwr` sono rispettivamente collegati ai segnali della cache `ch_memrd` e `ch_memwr`.

A livello di codice nel processo `async` sono stati modificati i rami del case `a_opcode_high is` inerenti la load e la store.

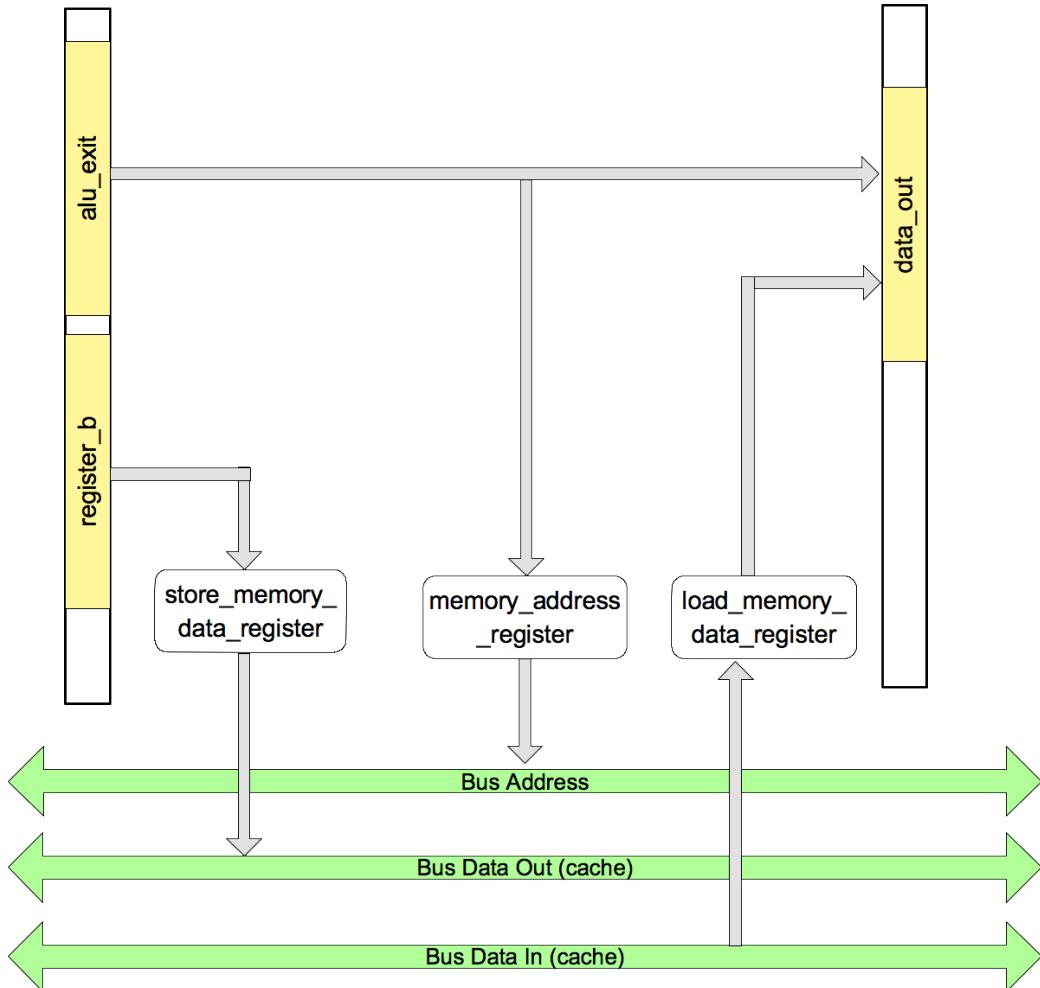


Figura 3.3: Schema dello stadio di MEM

2.1 Istruzione load

Nel caso di un'istruzione load, il codice è stato modificato come illustrato di seguito:

Listing 3.2: Codice dell'istruzione load

```
memory_address_register <= alu_exit_buffer;
memrd <= '1';
wait until ready = '1' and ready'event;
memrd <= '0' after TIME_UNIT/3;
dest_register <= a_rd_i;
```

```
dest_register_data <= load_memory_data_register;
data_out <= load_memory_data_register;
```

L'uscita dell'ALU viene inviata al bus indirizzi e viene attivato il segnale `memrd` che sveglia il processo `cache_dlx` della cache, dopodichè l'istruzione `wait until` pone il processo in attesa di un fronte del segnale `ready`.

Appena il `ready` viene attivato il dato proveniente dalla cache viene inviato alla barriera dei registri dello stadio di Write-Back. Infine il segnale `memrd` viene riportato a 0, ma con un ritardo di `TIME_UNIT/3` necessario per poter rendere visibile l'impulso del segnale in fase di simulazione.

L'utilizzo dell'istruzione `wait until` si è reso necessario in alternativa alla realizzazione di un processo separato che sul fronte del `ready` effettuasse la scrittura dei dati provenienti dal bus sui registri di uscita, in quanto con quest'ultima soluzione si avrebbero due processi distinti in grado di modificare i valori dei segnali `data_out` e `dest_register_data` cosa che dà luogo a dei conflitti in fase di simulazione. Impiegare la `wait until` ha comportato come unico effetto collaterale lo spostamento dei segnali presenti nella sensitivity list del processo `async` nella lista dei parametri della `wait on` posta come prima istruzione del processo.

2.2 Istruzione store

La struttura della store risulta simile a quella della load:

Listing 3.3: Codice dell'istruzione store

```
store_memory_data_register <= memory_data_register_buffer;
memory_address_register <= alu_exit_buffer;
memwr <= '1';
wait until ready = '1' and ready'event;
memwr <= '0' after TIME_UNIT/3;
```

In questo caso oltre all'indirizzo viene mandato sul bus dati verso la cache il dato da memorizzare e viene attivato il `memwr`.

Come in precedenza anche qui il processo attende il fronte del ready e riporta a zero `memwr` con un ritardo di `TIME_UNIT/3`.

Capitolo 4

Testbench

In questo capitolo saranno descritti i testbench realizzati per verificare il corretto funzionamento del componente.

1 Testbench del componente

Per testare nello specifico il funzionamento della cache e i meccanismi di comunicazione con la ram, sono stati realizzati 3 file:

- Cache_test_ReadAndWrite.vhd: verifica il corretto funzionamento delle scritture nella cache e nella RAM;
- Cache_test_ReadAndReplacement.vhd: verifica il corretto funzionamento della politica di rimpiazzamento mediante contatori;
- Cache_test_snoop.vhd: verifica il corretto funzionamento del protocollo MESI in caso di eventuali snoop.

Durante tutti i test eseguiti è stata collegata alla cache il componente Ram_cmp il quale implementa una semplice memoria RAM.

Per completezza e per aiutare una successiva lettura delle forme d'onda generate da ISIM durante la simulazione, si riporta fra parentesi dove necessario il valore binario di ciò a cui si fa riferimento.

1.1 Cache_test_ReadAndWrite.vhd

Questo file ha in comune con il resto dei file di testing dei componenti realizzati la dichiarazione dei componenti, il portmap e la fase di reset iniziale. In questo caso specifico il testbench si compone di 3 fasi con le quali si verifica il funzionamento delle scritture:

Fase 1: Letture d' inizializzazione

Si caricano in Cache 3 blocchi da 32 byte, 2 in modalità write-back `ch_wtwb='0'` e uno in modalità write-throght `ch_wtwb='1'`. Si occupano quindi rispettivamente la prima via del primo set ((0)(1)), del secondo set ((1)(1)) e terzo set ((2)(1)). In particolare nel terzo set si nota come l'attivazione del segnale `ch_wtwb`, porti lo stato (`.status`) del terzo blocco a 01 ovvero a `MESI_S`, mentre negli altri due casi è uguale a 10 ovvero `MESI_E`, differenza evidenziata in verde nella colonna dei valori nella figura 4.1) .

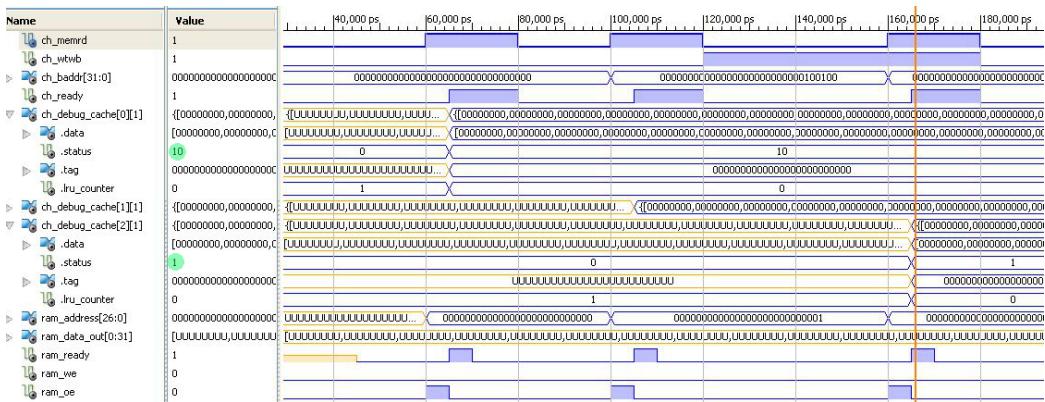


Figura 4.1: screenshot ISIM fase 1

Fase 2: Scrittura

Nella prima scrittura, evidenziata in verde nella figura 4.2 avviene un MISS, in quanto il blocco di TAG=10 non è ancora presente in cache. Verranno quindi effettuate le seguenti operazioni:

- lettura in RAM per leggere il blocco contenente il dato e portarlo nella seconda via del primo set ((0)(0));
- aggiornamento dei valori dei contatori che gestiscono l'invecchiamento per il meccanismo di rimpiazzamento;
- scrittura in cache aggiornando il valore della word di offset 00110, ovvero i byte dalla posizione (6) alla (9) e lo stato, .status , della via viene posto a MESI_M (11).

Nella seconda, avviene una scrittura su di un blocco già presente in cache, quindi si modifica semplicemente il dato all' offset specificato e si aggiorna lo stato come nel caso precedente.

Nella terza scrittura , evidenziata in azzurro nella figura4.2, a differenza dei due casi precedenti dove la via si porta in stato MESI_M, la via si porta in MESI_E in quanto il blocco è stato portato in cache in modalità write-throught di conseguenza, la scrittura oltre ad avvenire in cache, avviene anche in RAM, e la nostra cache sarà l'unica a possedere il blocco aggiornato, se qualche altro dispositivo possedeva quel blocco, il controllore di memoria dovrà preoccuparsi di procedere con l'invalidazione.



Figura 4.2: screenshot ISIM fase 2

Fase 3: Letture di verifica

In questa ultima fase vengono eseguite delle letture. Nel primo caso due letture per verificare i dati scritti nelle prime due scritture.

Nel secondo caso (riportato in figura 4.3) si obbliga la cache ad effettuare un replacement, in modo da verificare che la cache esegua correttamente il salvataggio in RAM e poi viene ricaricato il blocco originariamente modificato. Consentendoci di verificare immediatamente che il dato che ci viene restituito è quello che era stato originariamente scritto in cache.

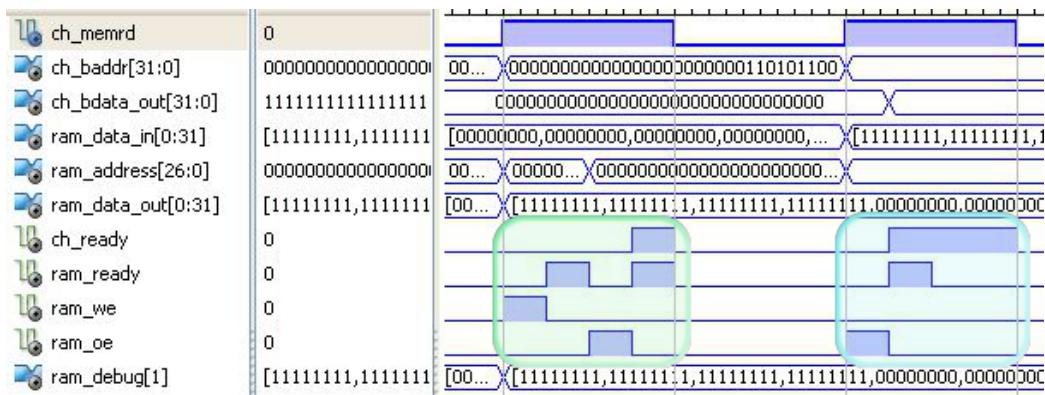


Figura 4.3: screenshot ISIM fase 3

Nella figura 4.3 si nota inoltre la sequenza di segnali che vengono utilizzati fra RAM e cache per comunicare all'attivazione del segnale ch_memrd. Nel primo caso, in verde, per scrivere in RAM un blocco modificato presente in cache e nel secondo caso, in azzurro per una lettura in RAM per portare un blocco in cache.

1.2 Cache_test_ReadAndReplacement.vhd

Questo testbench prevede 3 fasi, con le quali si verifica il corretto funzionamento del meccanismo a contatori per tener traccia di quanto recentemente si è acceduti ad una linea di cache, al fine di gestire la politica di rimpiazzamento durante una serie di letture. Si analizzerà infine anche il caso in cui la cache subisca l'invalidazione di una linea.

Fase 1: Letture di inizializzazione

Nella prima fase si effettuano 8 letture per riempire tutta la cache. Avremo quindi tutti i contatori, `lru_counter`, delle vie, dei quattro set, che sono stati caricati per ultimi a 0, mentre le altre vie avranno il contatore a 1.

Fase 2: Invalidazione

Nella seconda fase si esegue un invalidazione sul blocco di $\text{TAG}=010$ e con $\text{index}=10$, ovvero il terzo set; questo comporta il portare lo stato, `.status`, da `MESI_E(10)` a `MESI_I(0)` e contatore a 1 nella via che lo contiene mentre l'altra via del set si porta a 0.

Fase 3: Verifica meccanismo contatori

Con la prima lettura si verifica che se si effettua una lettura di una word contenuta in un blocco di $\text{TAG}=0$, presente in cache, ovvero abbiamo un hit, il contatore della via (1)(1) dove è contenuto viene resettato, mentre le altre vie con valore più basso di contatore vengono incrementate.

Con la seconda lettura si richiede un dato contenuto in un blocco non presente in cache, ovvero abbiamo miss, viene quindi selezionato il set in base al valore dell' indice, 01 in questo caso e verrà rimpiazzato il blocco con valore del contatore più elevato che in questo caso corrisponde alla via con $\text{TAG}=10$ (1)(0) nella quale si era precedentemente resettato il contatore della via, senza effettuare nessuna scrittura in RAM in quanto il blocco è in stato `MESI_E`, ovvero il blocco in cache non ha subito nessuna modifica rispetto a quanto presente in RAM.

Con la terza lettura invece si richiede un dato contenuto in un blocco non presente in cache con $\text{indice}=10$, che corrisponde al terzo set, nel quale una via era stata precedentemente invalidata. Tale

via((2)(0)) verrà quindi rimpiazzata senza scrittura in RAM (perchè la linea è in MESI_L) con il blocco con TAG=110.

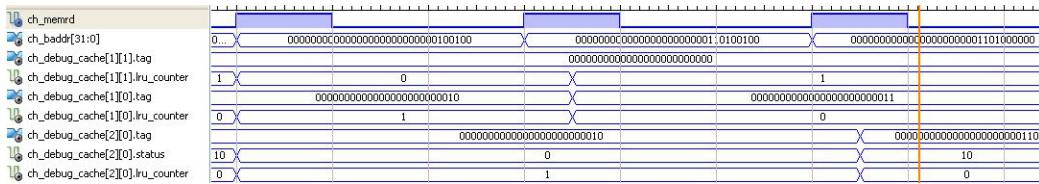


Figura 4.4: screenshot ISIM 3 letture

Infine vengono eseguite una serie di letture su blocchi presenti e non su diversi set e vie, per verificare che tutto nel complesso funzioni correttamente.

1.3 Cache_test_snoop.vhd

Per prima cosa si inizializza correttamente la cache per permettere di verificare il corretto funzionamento dello snoop nei vari casi. Nello specifico si portano prima in cache due blocchi dalla RAM e sul secondo blocco si esegue una scrittura, per portarlo in stato MESI_M.

In secondo luogo si attiva il segnale di snoop ,*ch_eds*, e si imposta l'indirizzo su cui fare lo snoop ,*ch_snoop_addr*. La cache risponderà in modo opportuno con i segnali di *ch_hit*, *ch_hitm* e modificherà in alcuni casi lo stato delle vie se il blocco è presente in cache.

Nel primo caso si effettua uno snoop su di un blocco che non è contenuto in cache, quindi la cache risponderà con *ch_hit*=0, *ch_hitm*=0.

Nel secondo caso, evidenziato in giallo in Fig.4.5, si effettua uno snoop su di un blocco presente in cache in stato MESI_E, quindi la cache risponderà con *ch_hit*=1, *ch_hitm*=0 e porta lo stato della via interessata a MESI_S(1).

Nel terzo caso, evidenziato in verde in Fig.4.5, si effettua uno snoop su di un blocco che è presente in cache in stato MESI_M(11). La cache quindi deve:

- portare `ch_hit=1` e `ch_hitm=1`
- forzare la scrittura in RAM del blocco contenuto in cache
- portare lo stato della via in MESI_S(1)

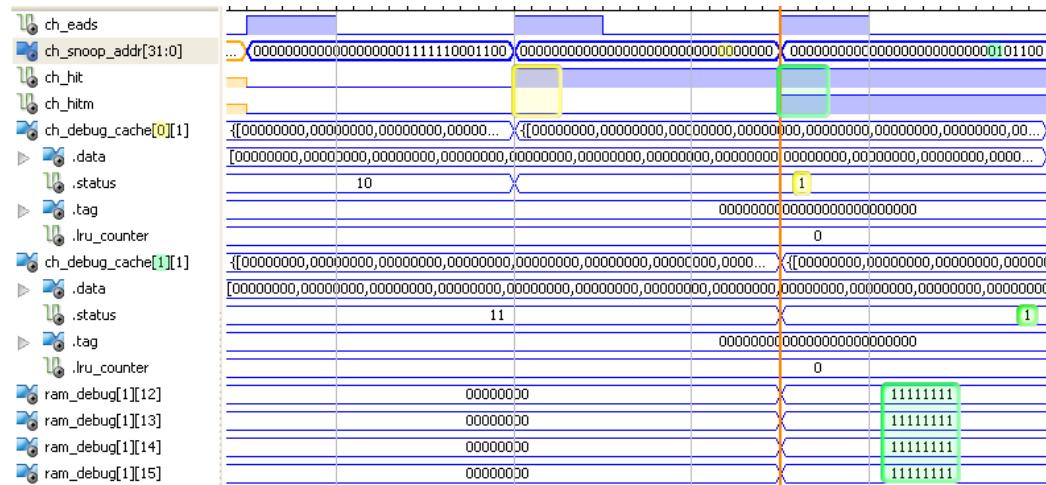


Figura 4.5: screenshot ISIM caso 2 e 3 di snoop

Infine si effettua una scrittura sul blocco precedentemente portato in MESI_S, per verificare che la cache scrivi effettivamente i dati in RAM.

2 Assembler per DLX

Dopo avere testato individualmente il funzionamento dei componenti cache e della RAM, si è passati al test del corretto funzionamento della cache inserita all'interno del progetto del processore DLX.

Per far ciò sono stati realizzati una serie di programmi in assembler, dei quali mostreremo solo i due maggiormente significativi:

- `provaReplacement123`: nel quale si verifica la corretta comunicazione tra cache e DLX e il meccanismo di rimpiazzamento.
- `provaFU`: nel quale si verifica il corretto funzionamento della Forwarding unit.

2.1 Dal codice all'esecuzione

Per completezza in questa sezione si spiegherà brevemente come poter mettere in esecuzione un codice assembler.

In primo luogo è necessario scrivere il codice in assembler all'interno di un file con estensione *.dls, che viene poi dato in pasto all'assemblatore DASM. Quest'ultimo lo converte in codice macchina mediante il seguente comando, eseguito dal prompt di comandi Windows:

```
dasm -a -l <nome_file>.dls
```

Il risultato sarà un file `<nome_file>.dlx` che a sua volta dovrà essere convertito mediante la classe `DLXConv`, eseguendo dal prompt di comandi Windows:

```
java DLXConv <nome_file>.dlx
```

per avere un file `<nome_file>.dlx.txt` contenente il codice in un formato direttamente inseribile all'interno del progetto del DLX.

In particolare quest'ultimo dovrà essere inserito nel file `Fetch_Stage.vhd` all'interno dell'array che rappresenta la EPROM contenente le istruzioni in linguaggio macchina, da far eseguire al processore:

Listing 4.1: Inserimento del codice nella memoria istruzioni

```
constant EPROM_inst: eprom_type(0 to 11) := (
— istruzioni in linguaggio macchina.
);
```

Nei paragrafi successivi si analizzeranno nel dettaglio i codici assembler dei due test più significativi. Per comodità si riporterà il codice contenuto nella `EPROM_inst` corredata di commento e codice assembler relativo.

2.2 provaReplacement123.vhd

Listing 4.2: Codice assembler per il test del meccanismo di rimpiazzamento

```
X"20010000",    --11: addi r1,r0,0 ; azzerà r1
X"20020001",    --12: addi r2,r0,1 ; imposta a 1 r2
X"AC220000",    --13: sw 0(r1),r2 ; memorizza il valore di r2
                    all'indirizzo 0+r1(via 1 dell index0)
X"20420001",    --14: addi r2,r2,1 ; incrementa r2
X"AC220100",    --15: sw 16#100(r1),r2 ; memorizza il valore di
                    r2 all'indirizzo 16#100+r1(via 0 dell index0)
X"20420001",    --16: addi r2,r2,1 ; incrementa r2
X"AC220080",    --17: sw 16#80(r1),r2 ; memorizza il valore di
                    r2 all'indirizzo 16#80+r1(replacement via 1 dell index0)
X"8C220000",    --18: lw r2,0(r1) ; ripristina valore iniziale di
                    r2 (1)
X"20210004",    --19: addi r1,r1,4 ; incremento di 4 indirizzo di
                    base in r1
X"0BFFFFE0",    --110: j l3 ;
X"FFFFFFFF",    --NOP
X"FFFFFFFF"     --NOP
```

Questo codice conta fino a tre, da qui il nome del file termina con 123, ad ogni iterazione ricomincia da uno, inoltre ogni risultato intermedio viene salvato in memoria in una locazione diversa in modo da obbligare la cache, ad ogni iterazione, a rimpiazzare un blocco, lavorando sempre nel primo set, per un numero limitato d'iterazioni. Infatti, ad esempio nella prima iterazione, quando r1=0, si nota che i due bit che identificano l'indice sono sempre uguali a 00:

- l3) 0(r1)=0+0= 000 00 00000 ovvero prima via occupata nel primo set ((0)(1)) siccome b6=0 e b7=0;
- l5) 16#100(r1)=010 00 00000 ovvero seconda via occupata nel primo set((0)(0));
- l7) 16#80(r1)= 001 00 00000 ovvero prima via rimpiazzata nel primo set((0)(1)), quindi si ha una scrittura in RAM del blocco modificato seguita da una lettura del nuovo blocco.

Si continuerà a operare nel primo set finchè il valore di r1 non supera trentuno, ovvero alla trentaduesima iterazione si passerà a lavorare sul secondo set, ma questo non avverrà mai nella nostra simulazione, data la durata limitata del testbench.

In seguito si effettua il caricamento in r2 del primo valore scritto in memoria (ovvero 1), ma il blocco che contiene la word di indirizzo 0 non è più presente in cache, verra quindi effettuato un ulteriore rimpiazzamento però sulla seconda via((0)(0)).

Infine si incrementa in valore di r1 di quattro unità in modo da scrivere non più nel byte di offset 00000 ma di offset 00010, ovvero la word successiva di ogni blocco.

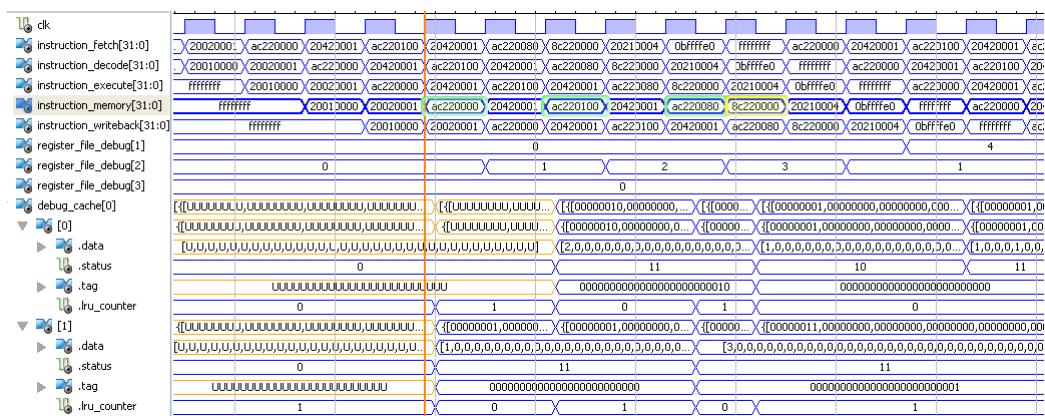


Figura 4.6: evidenziato in verde le store e in giallo le load

2.3 provaFU.vhd

In questo programma si testa il corretto funzionamento della Forwarding Unit per evitare un alea di dato.

Il codice da sostituire all'interno del EPROM_inst, al posto di quello dell'esempio precedente, è il seguente:

Listing 4.3: Codice assembler per il test della Forwarding Unit

```
X"20420001", --11: addi r2,r2,1 ; porta a 1 il valore di r2
```

```
X"AC220000", --12: sw 0(r1),r2 ; salva il contenuto di r2
X"8C230000", --13: lw r3,0(r1) ; porta in r3 il valore
    presente in r2
X"20620001", --14: addi r2,r3,1 ; incrementa r2
X"0BFFFFFF0", --15: j 12           ; salta a 11
X"FFFFFFFF", --NOP
```

La Forwarding Unit viene sfruttata nel quinto ciclo di clock per evitare un'alea si dato di tipo RAW. Si nota che l'istruzione X20620001 nello stadio di EX vuole leggere il valore contenuto in r3 per incrementarlo di uno ma è appena stato caricato dalla cache dall'istruzione (lw r3,0(r1)) presente nello stadio di MEM, pertanto il dato non è ancora disponibile nel Register File e vi sarebbe un'alea di dato. Per far fronte a ciò, la Forwarding Unit fornirà il dato presente nello stadio di MEM attraverso i segnali mem_dest_register e mem_dest_register_data allo stadio di EXE.

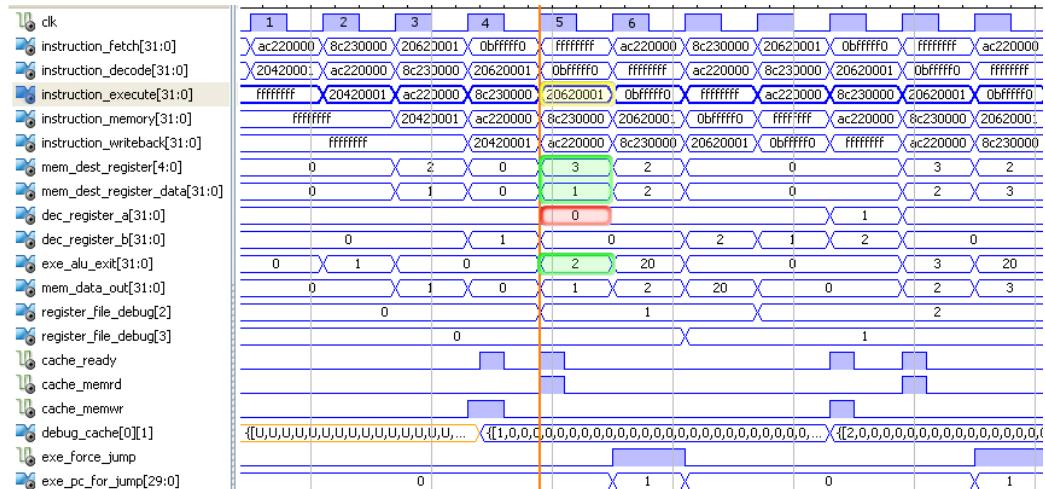


Figura 4.7: Screenshot ISIM di provaFU

Nella Fig. 4.7 sono mostrati:

- l'istruzione critica in giallo;
- i valori forniti dalla Forwarding Unit e il risultato corretto in verde;
- i valori non aggiornati forniti dallo stadio di decode in rosso.

Capitolo 5

Block RAM

Nel nostro progetto descritto in precedenza, per semplicità abbiamo considerato l’ipotesi di avere tempi d’accesso nulli alla cache e alla memoria principale. Ovviamente ciò non accadrebbe in un progetto reale che preveda l’utilizzo di cache al fine di velocizzare l’accesso ai dati evitando in caso di HIT l’accesso alla memoria RAM. Per curiosità e completezza nell’affrontare le tematiche relative al nostro progetto, abbiamo voluto approfondire le problematiche riguardanti le temporizzazioni per gli accessi in memoria che un progetto reale impone. Per far ciò abbiamo considerato i dispositivi di memoria che una FPGA dà a disposizione ad un progettista per implementare una memoria RAM e gestirne gli accessi in lettura e scrittura.

Nel nostro caso abbiamo analizzato le caratteristiche dell’FPGA della famiglia Spartan-3 di Xilinx (1), che per gestire la memorizzazione di dati consente due possibili soluzioni:

1. Memoria RAM presente sulla scheda (esternamente al chip FPGA).
2. Le Block RAM, ovvero blocchi di memoria RAM statica con tempi d’accesso non nulli e un’ampia capacità potenziale di memorizzazione, in relazione alle caratteristiche tecniche della scheda FPGA utilizzata.

1 Caratteristiche e segnali della Block Ram

La memoria RAM presente su una FPGA Spartan-3 (2) viene implementata tramite una serie di Block RAM ripartite in colonne il cui numero e capacità dipende dalle caratteristiche stesse dell'FPGA utilizzata come illustrato in Fig.5.1.

Dal punto di vista implementativo le Block RAM sono realizzate tra-

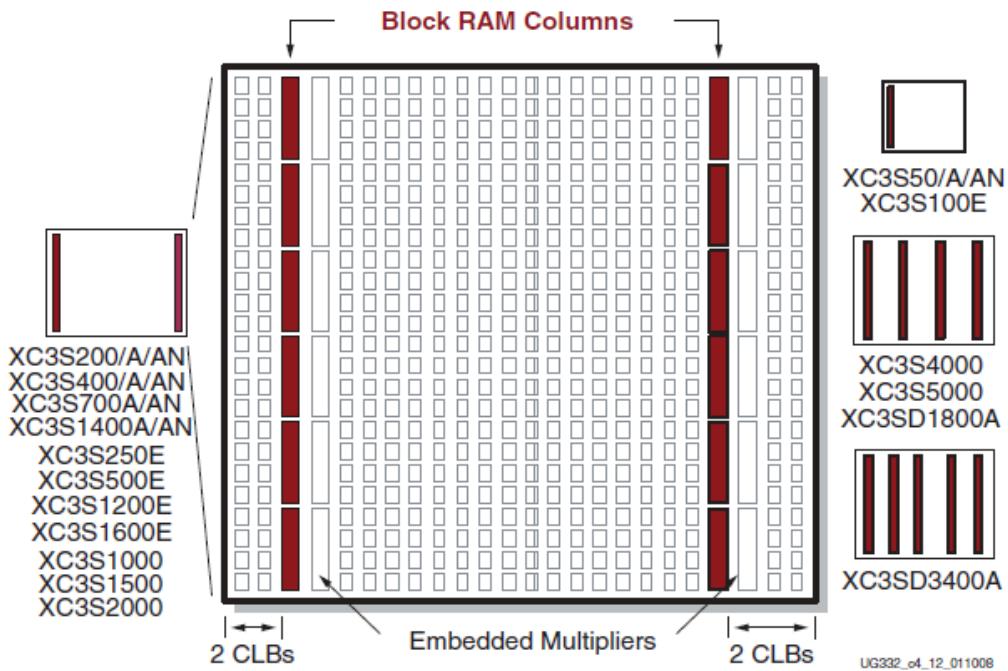
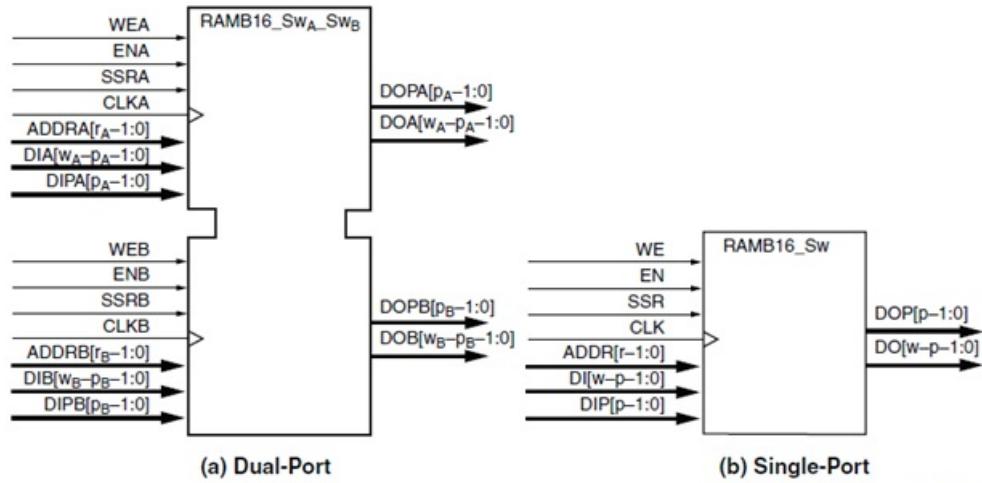


Figura 5.1: Location delle colonne di Block Ram su FPGA

mite 18,432 celle di memoria SRAM che consentono pertanto di memorizzare 18 Kbits di cui 16 Kbits di dato e 2 Kbits utilizzati tipicamente per memorizzare i bit di parità relativi ai dati memorizzati o in alternativa come spazio di memorizzazione aggiuntivo.

L'accesso alla Block RAM può avvenire o in modalità Single-Port utilizzando una sola porta dati oppure in Dual-Port tramite 2 porte indipendenti A e B che consentono di effettuare operazioni di lettura e scrittura sull'intero spazio di memoria del dispositivo (anche con sovrapposizioni).

**Notes:**

1. w_A and w_B are integers representing the total data path width (i.e., data bits plus parity bits) at ports A and B, respectively. See [Table 4-8](#) and [Table 4-9](#).
2. p_A and p_B are integers that indicate the number of data path lines serving as parity bits.
3. r_A and r_B are integers representing the address bus width at ports A and B, respectively.
4. The control signals CLK, WE, EN, and SSR on both ports have the option of inverted polarity.

X463_01_112009

Figura 5.2: Block Ram Single-Port e Dual-Port

Ogni porta della Block RAM si interfaccia con due bus dati (distinti per l'input e per l'output), con il bus degli indirizzi e dispone di una serie di segnali di comando atti ad abilitare il dispositivo e a gestire operazioni di lettura o scrittura. La tabella in Fig.5.3 racchiude i principali segnali in input e output della Block Ram sia in Single-Port che in Dual-Port.

Segnali di comando in Single-Port:

- EN: Enable consente di abilitare il dispositivo e qualora non siano asseriti WE (write enable) o SSR (synchronous set/reset), il segnale comanda la lettura del dato all'indirizzo specificato sul bus degli indirizzi ADDR sul fronte positivo del clock;
- WE: Write Enable consente di comandare un ciclo di scrittura in memoria all'indirizzo specificato sul bus degli indirizzi ADDR (con EN asserito). La peculiarità delle operazioni di scrittura è che sono sempre precedute o seguite da una lettura del dato al medesimo indirizzo sulla base del comportamento configurabile tramite l'attributo `WRITE_MODE` che analizzeremo in seguito;

Signal Description	Single Port	Dual Port		Direction
		Port A	Port B	
Data Input Bus	DI	DIA	DIB	Input
Parity Data Input Bus (available only for byte-wide and wider organizations)	DIP	DIPA	DIPB	Input
Data Output Bus	DO	DOA	DOB	Output
Parity Data Output (available only for byte-wide and wider organizations)	DOP	DOPA	DOPB	Output
Address Bus	ADDR	ADDRA	ADDRB	Input
Write Enable	WE	WEA	WEB	Input
Clock Enable	EN	ENA	ENB	Input
Synchronous Set/Reset	SSR	SSRA	SSRB	Input
Clock	CLK	CLKA	CLKB	Input
Synchronous/Asynchronous Set/Reset (Spartan-3A DSP FPGA only)	N/A	RSTA	RSTB	Input
Output Register (Spartan-3A DSP FPGA only)	N/A	REGCEA	REGCEB	Input

Figura 5.3: Segnali della Block Ram Single-Port e Dual-Port

- SSR: Syncronous Set/Reset consente di assegnare '1' o non '0' i latch di ouput sul bus dati in accordo col valore dell'attributo SRVAL specificato in fase di inizializzazione (X00000 a default);
- REGCE: Output Register Enable consente in fase di lettura da ram di salvare il dato letto in un output register;
- CLK: è il clock e si può configurare se la memoria debba essere sensibile ai fronti di salita o di discesa;
- GSR: Global Set/Reset segnale di sistema utilizzato in fase di inizializzazione del sistema per inizializzare la Block RAM (non disponibile all'esterno su un pin).

C'è inoltre la possibilità di configurare le polarità di ogni segnale di comando se da considerarsi asserito alto o basso.

Interfacciamento ai bus:

- ADDR: bus degli indirizzi la cui larghezza (NA:0) dipende dalla configurazione della Block RAM;
- DI: Data Input Bus (ND:0) (l'ampiezza del dato da trasferire dipende dalla configurazione della Block RAM);
- DO: Data Output Bus;
- DIP: Data Input Parity Bus (nei bit più significative del Bus Dati di Input);
- DOP: Data Output Parity Bus (nei bit più significative del Bus Dati di Output);

Per quanto riguarda l'ampiezza dei dati che vengono acceduti durante un ciclo di trasferimento e la memory depth associata, le possibili configurazioni della Block RAM sono illustrate in Fig. 5.4.

Ad esempio, se si volesse utilizzare la Block Ram come memoria Ram per un processore DLX, la configurazione necessaria sarebbe la 512x36. Tale configurazione dà la possibilità di accedere al dispositivo fino a 36 bit contemporaneamente, di cui 32 bit di dato e 4 bit di parità. Con tale configurazione la Block Ram (di 18 Kbit) conterebbe 512 entry (memory-depth) da 36 bit (infatti $512 \times 36 \text{ bit} = 18 \text{ Kbit}$).

2 Configurazione della Block RAM

La configurazione della Block RAM avviene tramite una serie di attributi nella sezione `generic (. . .)` dei componenti RAM disponibili nelle librerie di sistema tramite i quali si può settare in base alle specifiche di progetto l'organizzazione interna dei dati, l'ampiezza dei dati trasferiti e diverse altre caratteristiche della Block Ram come il comportamento in scrittura.

La definizione di un componente Block Ram e la relativa configurazione tipicamente possono essere gestiti in due modi:

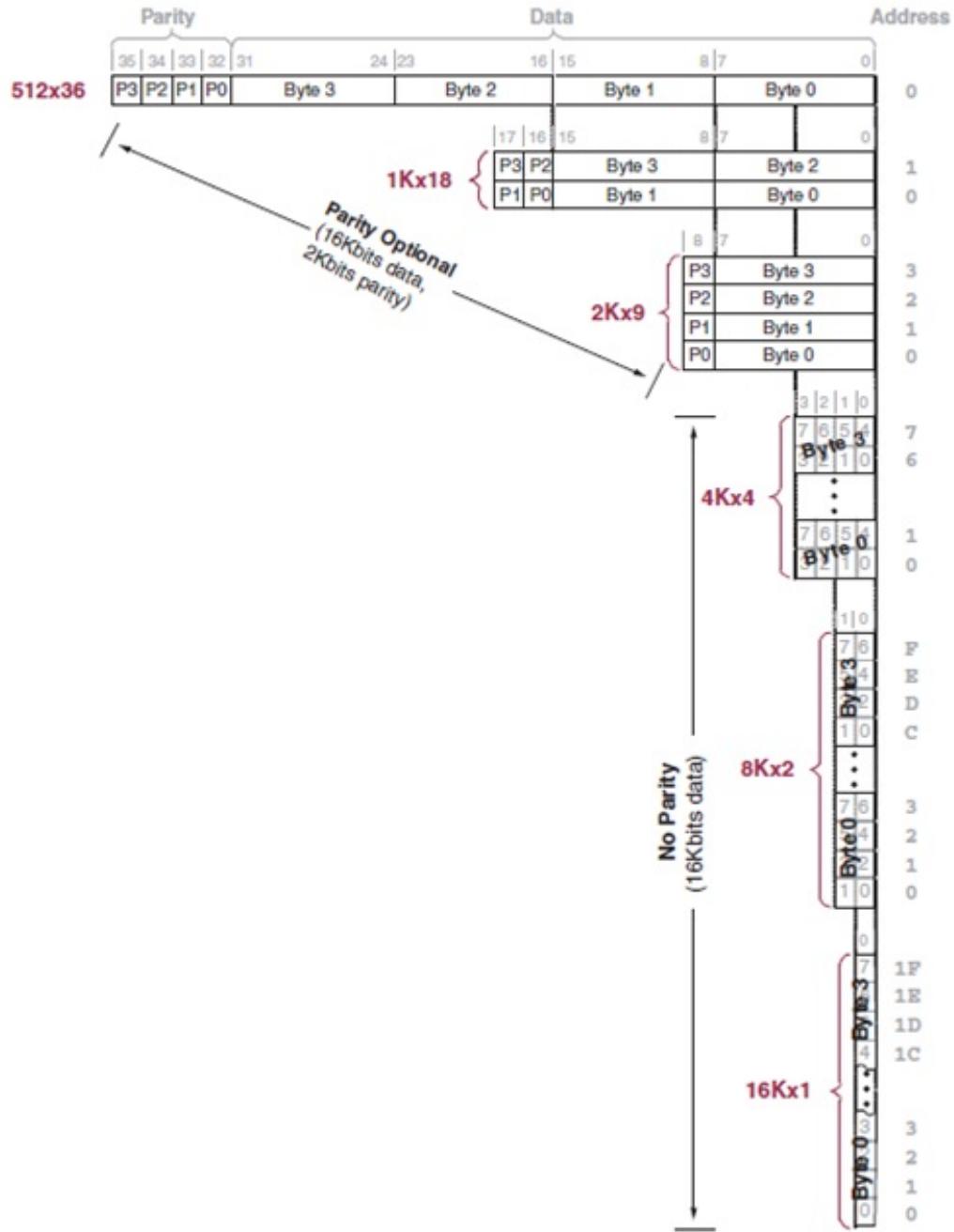


Figura 5.4: Possibili organizzazioni interne della Block Ram

1. Xilinx Core Generator.
2. Utilizzo di componenti predefiniti della libreria UNISIM della Xilinx e configurazione tramite appositi attributi.

Xilinx Core Generator consente tramite un’interfaccia grafica di configurare interamente tutte le possibili caratteristiche della Block Ram ottenendo direttamente il codice VHDL del componente ram desiderato. Tale strumento è però disponibile solo nelle versioni complete del software ISE di Xilinx.

In alternativa, si possono utilizzare i tipi VHDL già associati alla Block Ram presenti nella libreria **UNISIM** package **VComponents**. In entrambi i casi si riesce a definire direttamente il numero di porte da utilizzare per l’accesso alla Block Ram e l’ampiezza del dato acceduto (da ciò segue l’organizzazione interna dei dati).

Per Block Ram in Single-Port i componenti sono denominati **RAMB16_Sn** dove n corrisponde all’ampiezza del dato + parità. In Fig.5.5 si mostano le possibili organizzazioni dei dati e i rispettivi componenti della libreria Xilinx UNISIM che le implementano.

Organization	Memory Depth	Data Width	Parity Width	DI/DO	DIP/DOP	ADDR	Single-Port Primitive	Total RAM Kbits
512x36	512	32	4	(31:0)	(3:0)	(8:0)	RAMB16_S36	18K
1Kx18	1024	16	2	(15:0)	(1:0)	(9:0)	RAMB16_S18	18K
2Kx9	2048	8	1	(7:0)	(0:0)	(10:0)	RAMB16_S9	18K
4Kx4	4096	4	-	(3:0)	-	(11:0)	RAMB16_S4	16K
8Kx2	8192	2	-	(1:0)	-	(12:0)	RAMB16_S2	16K
16Kx1	16384	1	-	(0:0)	-	(13:0)	RAMB16_S1	16K

Figura 5.5: La tabella mostra le diverse tipologie di **RAMB_Sn** ottenibili dalla Block Ram in base all’organizzazione interna desiderata in Single-Port.

Analogamente avviene per l’utilizzo di Block Ram Dual-Port dove l’organizzazione dei dati acceduti dalle due porte è rappresentato dai suffissi mostrati in Tabella 5.6.

2.1 Inizializzazione del contenuto e attributi di configurazione

- **INIT_xx - INITP_xx** A default la Block Ram è inizializzata a tutti 0, ma è possibile inizializzarne il contenuto in diversi modi o

		Port A					
		16Kx1	8Kx2	4Kx4	2Kx9	1Kx18	512x36
Port B	16Kx1	<u>S1_S1</u>					
	8Kx2	<u>S1_S2</u>	<u>S2_S2</u>				
	4Kx4	<u>S1_S4</u>	<u>S2_S4</u>	<u>S4_S4</u>			
	2Kx9	<u>S1_S9</u>	<u>S2_S9</u>	<u>S4_S9</u>	<u>S9_S9</u>		
	1Kx18	<u>S1_S18</u>	<u>S2_S18</u>	<u>S4_S18</u>	<u>S9_S18</u>	<u>S18_S18</u>	
	512x36	<u>S1_S36</u>	<u>S2_S36</u>	<u>S4_S36</u>	<u>S9_S36</u>	<u>S18_S36</u>	<u>S36_S36</u>

Figura 5.6: Suffissi relativi alle possibili configurazioni in Dual-Port.

direttamente tramite Core Generator al momento della configurazione del componente oppure tramite opportuni attributi VHDL come INIT_xx e INITP_xx.

Nel primo caso si passa direttamente un file di coefficienti (.coe) che definisce in primo luogo la base numerica dei dati da inserire e in seguito l'elenco dei dati elencati a partire dalla parte bassa della memoria fino agli indirizzi alti. Un esempio della struttura di tale file è il seguente:

```
memory_inizialization_radix=16;
memory_inizialization_vector=80, 0F, 00, ..., 82;
```

In alternativa si utilizzano direttamente 64 attributi VHDL INIT_xx (da INIT_00 a INIT_3F in Fig.5.7) che consentono di inizializzare le 64 zone da 256 bit con cui è ripartita la memoria. Gli indirizzi del blocco di memoria da inizializzare identificati da xx sono calcolabili nel seguente modo dopo aver convertito l'indirizzo esadecimale xx nel corrispondente indirizzo decimale yy:

$$\begin{aligned} \text{indirizzo iniziale del blocco xx} &= ((yy + 1) * 256) - 1 \\ \text{indirizzo finale del blocco xx} &= yy * 256 \end{aligned}$$

INITP_xx sono attributi analoghi che consentono di inizializzare i bit di parità presenti in memoria (da INITP_00 a INITP_07).

Attribute	From	To
INIT_00	255	0
INIT_01	511	256
INIT_02	767	512
...
INIT_3F	16383	16128

Figura 5.7: Attributi di Inizializzazione del contenuto della Block Ram

- **INIT** è l'attributo utilizzato in fase di inizializzazione per settare il valore iniziale del registro di output quando viene asserito il segnale GSR.
 - **LOC** consente di specificare quale Block Ram utilizzare all'interno dell'FPGA individuandola tramite coordinate di riga e colonna. Tale operazione può essere fatta solamente tramite codice VHDL (non con il Core Generator) specificando `LOC = RAMB16_X#_Y#`.
- La Fig5.8 mostra in dettaglio come si configura una Location Constraint tramite coordinate riga-colonna. Naturalmente la scelta delle coordinate dipende dal numero di Block Ram presenti sull'FPGA utilizzata.
- **SRVAL** rappresenta il dato che viene portato in uscita alla Block Ram durante i particolari cicli d'accesso caratterizzati dal segnale SSR attivo.
 - **WRITE_MODE** è l'attributo che consente di specificare il comportamento della Block Ram e dei latch di output durante un ciclo di scrittura. I possibili valori sono i seguenti:

1. **WRITE_FIRST** è il valore di default e comporta un comportamento Read after Write della memoria, ovvero durante un ciclo di scrittura il dato in input viene contemporaneamente scritto alla locazione di memoria indicata dall'indirizzo e memorizzato e reso disponibile sui latch di output. Nel caso di utilizzo in Dual-Port si ha l'effetto collaterale di un'invalidazione del contenuto del registro di output dell'altra porta, causando pertanto potenziali problemi di

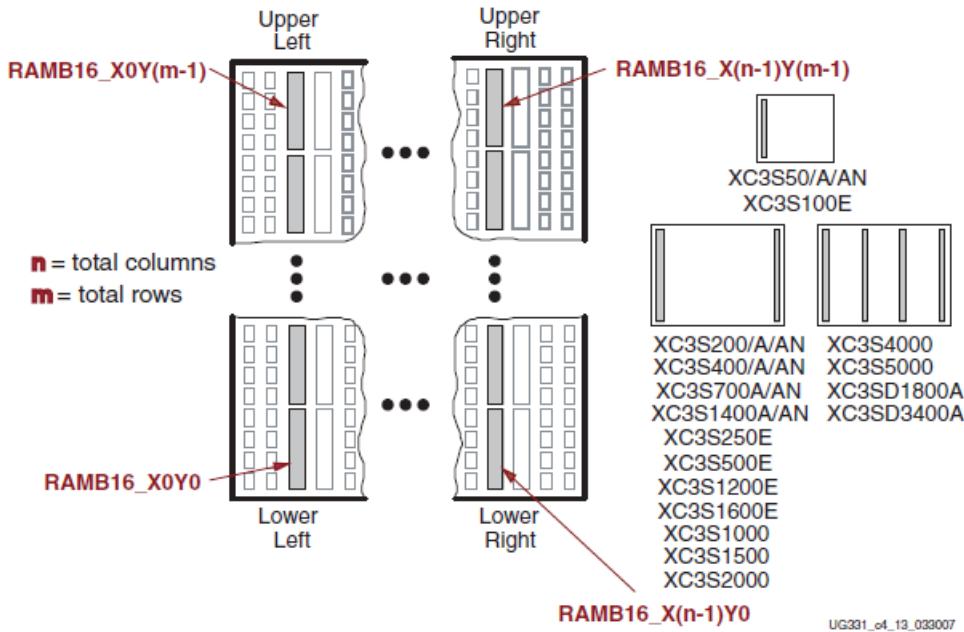


Figura 5.8: LOC: coordinante Block Ram su FPGA

conflitto in caso di scrittura e lettura contemporanei sulle due porte.(Fig.5.9).

2. **READ_FIRST** determina un comportamento Read before Write, ovvero prima si carica nei latch di output il dato presente alla locazione di memoria specificata dall'indirizzo e poi lo si sovrascrisce in memoria con l'operazione di scrittura. Le temporizzazioni e il comportamento dettagliato in tale modalità sono illustrati in Fig.5.10.
3. **NO_CHANGE** determina un comportamento classico di scrittura in memoria senza alcun aggiornamento del dato contenuto nel registro in output (temporizzazioni e funzionamento in Fig.5.11). Nel caso di utilizzo in Dual-Port si ha come side-effect l'invalidazione del contenuto del registro di output dell'altra porta.

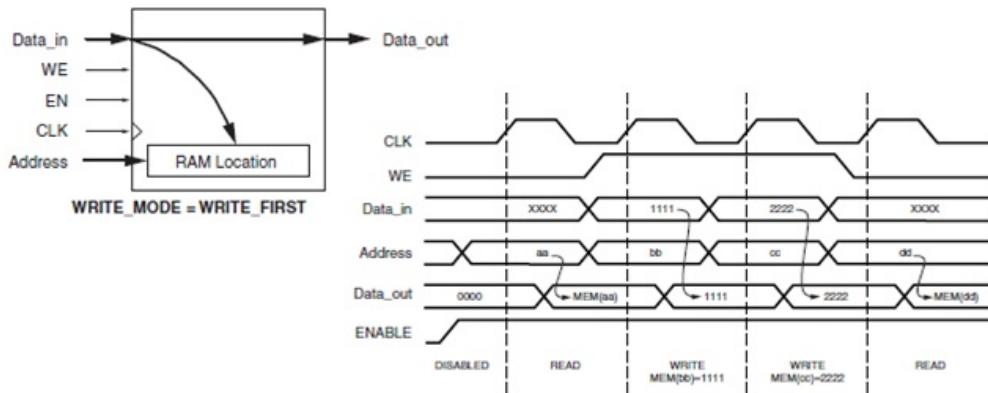


Figura 5.9: WRITE_MODE = WRITE_FIRST

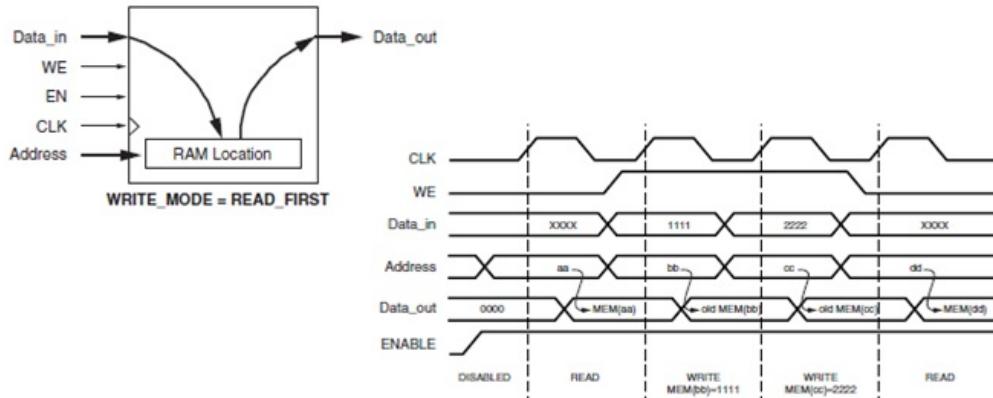


Figura 5.10: WRITE_MODE = READ_FIRST

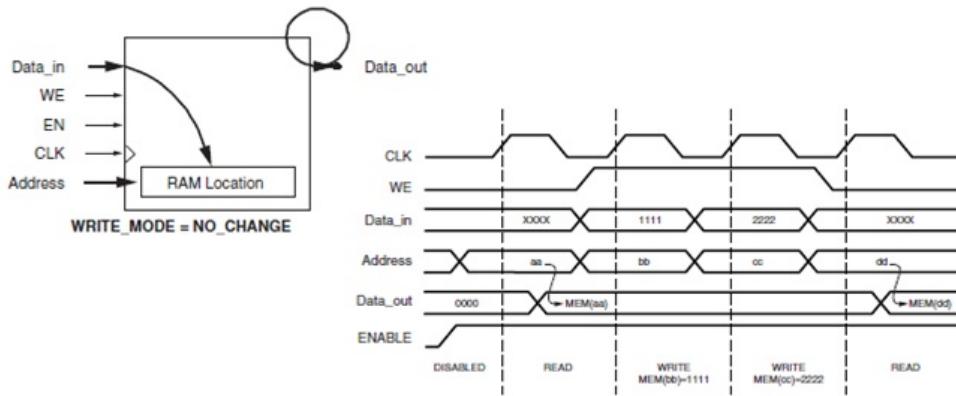


Figura 5.11: WRITE_MODE = NO_CHANGE

3 Operazioni della Block RAM

Di seguito viene riportato l'elenco delle operazioni che la Block Ram è in grado di gestire e dei relativi segnali impiegati:

- **Global Set/Reset**: segue la fase di inizializzazione iniziale del contenuto della Block Ram in cui si inizializza la ram o a tutti zeri (default) o ai valori impostati con gli attributi INIT_xx. Tale segnale serve per inizializzare lo stato dei flipflop e registri di output che vengono settati in base al valore specificato dall'attributo INIT (0 a default).
- **RAM Disabled**: se il segnale EN non è asserito la ram mantiene il proprio stato. Ogni operazione prevede che EN venga asserito affinchè la RAM sia attiva.
- **Synchronous Set/Reset**: è l'operazione conseguente all'asserzione contemporanea dei segnali EN e SSR. Tale operazione comporta la re inizializzazione dei registri di output al valore specificato dall'attributo SRVAL.
- WE + SSR comporta un ciclo di scrittura in cui il dato in input viene salvato in memoria all'indirizzo presente sul bus degli indirizzi, mentre nei latch di output viene impostato il valore dell'attributo SRVAL.
- **READ**: la lettura sulla Block RAM avviene in modo sincrono, quindi sul fronte positivo del clock qualora sia asserito il solo segnale di EN.
- **WRITE**: la scrittura sulla Block RAM avviene in modo sincrono sul fronte positivo del clock e qualora siano asseriti contemporaneamente EN + WE. La scrittura del dato in input sui pin dell'Input Data Bus avviene all'indirizzo specificato e tale operazione è affiancata contemporaneamente dalla lettura del dato alla stessa locazione di memoria che viene reso disponibile in lettura e caricato sui latch di output. Naturalmente la politica con la quale avviene tale operazione di scrittura e lettura simultanea è definita dal valore dell'attributo WRITE_MODE.

La tabella mostrata in Fig.5.12 racchiude quanto detto in precedenza e associa ad ogni operazione i valori dei segnali associati.

Input Signals								Output Signals		RAM Contents	
GSR	EN	SSR/RST	WE	CLK	ADDR	DIP	DI	DOP	DO	Parity	Data
Immediately After Configuration											
Loaded During Configuration								X	X	INITP_xx ³	INIT_xx ²
Global Set/Reset Immediately after Configuration											
1	X	X	X	X	X	X	X	INIT ³	INIT	No Chg	No Chg
RAM Disabled											
0	0	X	X	X	X	X	X	No Chg	No Chg	No Chg	No Chg
Synchronous Set/Reset											
0	1	1	0	↑	X	X	X	SRVAL ⁴	SRVAL	No Chg	No Chg
Synchronous Set/Reset during Write RAM											
0	1	1	1	↑	addr	pdata	Data	SRVAL.	SRVAL.	RAM(addr) ↔pdata	RAM(addr) ↔ data
Read RAM, no Write Operation											
0	1	0	0	↑	addr	X	X	RAM(pdata)	RAM(data)	No Chg	No Chg
Write RAM, Simultaneous Read Operation											
0	1	0	1	↑	addr	pdata	Data	WRITE_MODE = WRITE_FIRST ⁵ (default)			
								pdata	data	RAM(addr) ↔ pdata	RAM(addr) ↔ data
								WRITE_MODE = READ_FIRST ⁶ (recommended)			
								RAM(data)	RAM(data)	RAM(addr) ↔ pdata	RAM(addr) ↔ pdata
								WRITE_MODE = NO_CHANGE ⁷			
								No Chg	No Chg	RAM(addr) ↔ pdata	RAM(addr) ↔ pdata

Figura 5.12: Tabella delle operazioni ed dei segnali utilizzati sulla Block Ram

4 Conflitti d'accesso in Block RAM Dual-Port

Utilizzando la Block RAM in modalità Dual-Port si ha la possibilità di utilizzare contemporaneamente le due porte per accedere alla memoria sia in lettura sia in scrittura. Mentre da un lato ciò consente di aumentare lo throughput complessivo dei dati trasferiti, dall'altro vi sono potenziali problemi di conflitto negli accessi simultanei alle stesse celle di memoria.

Le condizioni di potenziale conflitto sono le seguenti:

1. Scrittura simultanea sulle due porte alla stessa locazione di memoria.

Tale situazione non è gestita tramite un meccanismo di arbitraggio per far fronte ad accessi in scrittura simultanei, ma l'effetto prodotto è quello di comportare l'invalidazione del contenuto dell'area di memoria coinvolta, in quanto il contenuto finale risulterà impredicibile.

2. Conflitti per temporizzazioni clock-to-clock tra le due porte.

Ciò accade a causa dei clock diversi che comandano le operazioni tra le due porte che sono troppo ravvicinati tra loro e il clock della porta in lettura non rispetta i tempi di setup per l'accesso in scrittura al dispositivo (arriva troppo presto quando ancora la scrittura in memoria non ha terminato). Un esempio è illustrato in Fig.5.13.

Nel primo caso, la porta B inizia la scrittura in memoria all'indirizzo aa del dato 3333 e poco dopo, prima che la scrittura abbia terminato, arriva il fronte del CLK_A che fa iniziare la lettura allo stesso indirizzo aa violando il tempo di setup necessario per scrivere il dato in memoria. Nel secondo caso invece si ha la scrittura da parte della porta B all'indirizzo bb del dato 4444 e in questo caso CLK_A rispetta le temporizzazioni di scrittura e la porta A legge il dato correttamente scritto in memoria.

3. Scrittura e Lettura contemporanea sulla stessa zona di memoria in funzione del WRITE_MODE impostato (Fig.5.14).

Nei casi di scrittura su una porta e lettura sull'altra, se si utilizza WRITE_MODE= NO_CHANGE o WRITE_FIRST, la scrittura su una porta invalida automaticamente il contenuto del registro di output (in lettura) dell'altra porta. Per tale motivo è consigliabile la modalità di scrittura READ_FIRST per evitare il problema dell'invalidazione del dato della porta in lettura.

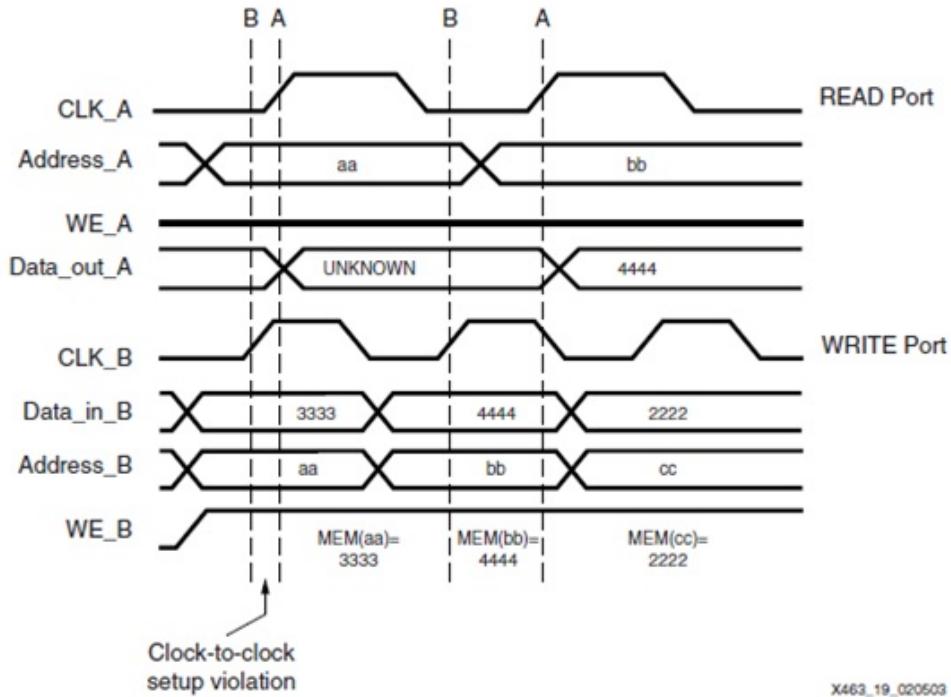


Figura 5.13: Conflitti per temporizzazioni d'accesso a Block Ram Dual-Port

Input Signals								Output Signals			
Port A				Port B				Port A		Port B	
WEA	CLKA	DIPA	DIA	WEB	CLKB	DIPB	DIB	DOPA	DOA	DOPB	DOB
WRITE_MODE_A=NO_CHANGE											
1	↑	DIPA	DIA	0	↑	DIPB	DIB	No Chg	No Chg	?	?
WRITE_MODE_B=NO_CHANGE											
0	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?	No Chg	No Chg
WRITE_MODE_A=WRITE_FIRST											
1	↑	DIPA	DIA	0	↑	DIPB	DIB	DIPA	DIA	?	?
WRITE_MODE_B=WRITE_FIRST											
0	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?	DIPB	DIB
WRITE_MODE_A=WRITE_FIRST, WRITE_MODE_B=WRITE_FIRST											
1	↑	DIPA	DIA	1	↑	DIPB	DIB	?	?	?	?

Figura 5.14: Conflitti in lettura e scrittura simultanea come side-effect della WRITE_MODE selezionata.

Per semplicità implementativa la Block Ram non implementa un sistema di arbitraggio per gestire tali conflitti che sono lasciati a cu-

ra del progettista e comunque in caso di conflitto dovuto a scritture contemporanee non si verificano danni fisici al dispositivo di memoria.

5 Utilizzo della Block Ram in un progetto su FPGA

La Block Ram può essere utilizzata in un progetto su FPGA per implementare una serie di funzionalità che coinvolgano la memorizzazione di dati. I principali possibili utilizzi sono i seguenti:

1. RAM utilizzata da un microprocessore integrato sull'FPGA per memorizzare dati accessibili in lettura e scrittura.
2. ROM realizzata attraverso l'inizializzazione del suo contenuto all'avvio del sistema e accessibile in sola lettura.
3. Memorie FIFO.
4. etc.

Tipicamente per utilizzare la Block Ram all'interno di un progetto si procede come segue:

1. Si crea un componente che utilizza la Block Ram configurandolo in base alle specifiche di progetto, settando il numero di porte volute, l'ampiezza dei dati da trasferire, la dimensione della ram voluta, etc.

Tale operazione può essere fatta in due modi:

- (a) Ricorrendo ad una serie di template presenti tra i Language Templates Ram di ISE appartenenti alla libreria UNISIM.
- (b) Configurazione ad hoc tramite Xilinx Core Generator che tramite un wizard consente di personalizzare il componente Ram di cui si ottiene infine il codice VHDL o Verilog.

2. Si integra il componente all'interno del progetto dichiarandolo nell'Architecture del componente finale e creandone un'istanza utilizzata internamente al progetto tramite degli appositi segnali interni.

Per utilizzare tali componenti implementati da Xilinx è necessario dichiare all'interno del progetto l'utilizzo della seguente libreria:

Listing 5.1: Libreria dei Components Xilinx predefiniti

```
library UNISIM;  
use UNISIM.VComponents.all;
```

3. Si utilizza il componente che rappresenta la Block Ram comandando i segnali di input e gestendo opportunamente i valori in output.

Capitolo 6

Progetto d'esempio: BlockRam_cmp

Al fine di testare il funzionamento della Block Ram e approfondire le problematiche che vi sarebbero state nel progettare una cache reale che si interfacci con una Ram esterna il cui tempo di accesso non è nullo, abbiamo realizzato un componente Ram ad hoc: `BlockRam_cmp`. Tale componente rappresenta una memoria Ram sincrona (il cui funzionamento è scandito dal clock in ingresso) realizzato con lo scopo di interfacciarsi con il nostro componente cache scambiando con questo linee di memoria di dimensione configurabile tramite un apposito parametro.

In questo caso, differentemente dall'implementazione realizzata nel `Ram_cmp` del progetto, la memorizzazione dei dati non è più gestita tramite un array di linee di memoria a cui si accede istantaneamente, ma tramite un componente interno `RAMB16_S9` capace di trasferire singoli byte ad ogni ciclo di lettura o scrittura.

1 Specifiche del progetto

- `BlockRam_cmp` è il componente che si occupa di gestire le richieste di lettura e scrittura di linee in memoria Block Ram.

- La dimensione in byte della linea di memoria è configurabile impostando opportunamente il numero di bit di OFFSET dell'indirizzo.
- Utilizzo di 1 componente Block Ram di dimensione 18 Kbits con organizzazione interna 2Kx9, ovvero ram-depth pari a 2048 ed ampiezza del dato trasferito di 9 bit (di cui 8 di dato e 1 di parità trascurato nel progetto).
- Il componente `BlockRam_cmp` ha lo scopo di interfacciarsi internamente con la Block Ram per gestire una sequenza di $2^{*\text{OFFSET_BIT}}$ trasferimenti da o verso la Block Ram al fine di leggere o scrivere in memoria un'intera linea di memoria.

2 Implementazione

Per comodità abbiamo ipotizzato che il componente `BlockRam_cmp` si interfacci alla cache sempre tramite un bus dati dell'ampiezza della linea di memoria da trasferire. Tale ipotesi semplificativa porta ad una potenziale complessità del cablaggio del bus dati, ma è tuttavia lecita dal momento che i trasferimenti tra cache e ram coinvolgono sempre linee di memoria.

Il nuovo componente prevede l'utilizzo al suo interno del componente Block Ram `RAMB16_S9` capace di leggere e scrivere dati da 8 bit. La scelta di tale organizzazione della Block Ram deriva dall'ipotesi che le linee di memoria sono sempre di dimensione multipla di 8 bit (1 Byte) e quindi il componente `BlockRam_cmp` ad ogni operazione di lettura o scrittura di una linea deve provvedere ad un ciclo di trasferimento dei singoli Byte costitutivi la linea a partire dall'indirizzo specificato in ingresso sul bus degli indirizzi che ad ogni accesso dovrà essere incrementato opportunamente.

In alternativa si sarebbero potuti trasferire dati anche maggiori (fino a 32 bit) ma l'effetto sarebbe stato quello di avere un vincolo ulteriore sulla dimensione della linea che sarebbe stata multiplo di un maggiore numero di byte (4 byte nel caso di trasferimenti a 32 bit in

Block Ram).

Listing 6.1: Il codice seguente mostra l' inizializzazione fatta nel progetto d'esempio del contenuto interno della Block Ram tramite gli attributi INIT_xx SRVAL_INIT e le modalità di scrittura WRITE_MODE.

```

ADDR: in std_logic_vector (ADDR_BIT-1 downto 0);
WE: in std_logic;
EN: in std_logic;
CLK: in std_logic;
SSR: in std_logic
);
end component;

```

La Fig. 6.1 mostra una schematizzazione dei processi utilizzati per gestire le funzionalità sopra descritte.

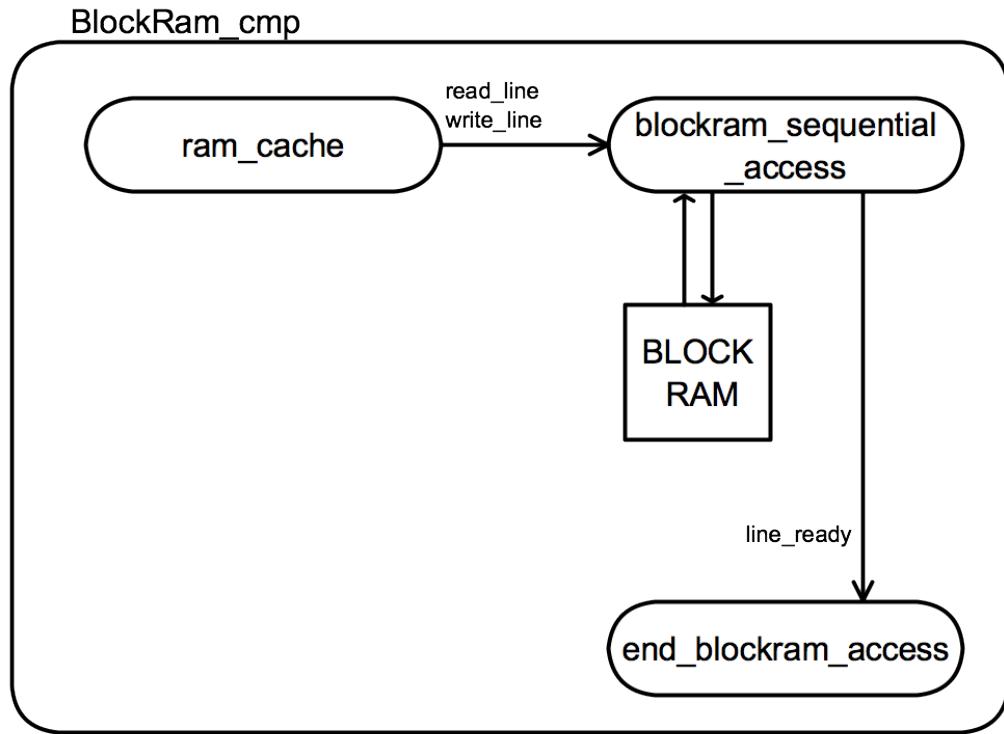


Figura 6.1: Schematizzazione dei processi che gestiscono la Block Ram

2.1 `ram_cache`

Il processo `ram_cache` è il processo principale che gestisce le richieste di trasferimento di linee di memoria provenienti dalla cache.

Tale processo sulla base dei segnali di comando ricevuti (`en`, `memrd` e `memwr`) asserisce i segnali interni di sincronizzazione, abilitando le seguenti operazioni:

1. `write_line`: la scrittura di una linea in Block Ram deve provvedere al trasferimento byte per byte della linea in ingresso su `bdata_in` sulla Block Ram tramite una serie di `2**OFFSET_BIT` scritture consecutive che avvengono sul fronte positivo del clock `clk`.
2. `read_line`: la lettura di una linea da Block Ram deve prevedere un buffer (una variabile VHDL `line` di tipo `mem_line`) che viene riempito man mano attraverso `2**OFFSET_BIT` letture di byte dalla Block Ram. Al termine la linea letta deve essere restituita in uscita alla cache su `bdata_out` (bus dati di output).

2.2 blockram_sequential_access

Il process `blockram_sequential_access` si occupa di gestire tramite un contatore interno gli accessi sequenziali alla Block Ram, scanditi dal clock `clk`. Tali accessi in sequenza saranno in lettura qualora `read_line` è asserito, in scrittura se è asserito il segnale `write_line`. Per tale motivo questo process ha la responsabilità di incrementare l'indirizzo di memoria dopo ogni accesso e comandare tramite opportuni segnali interni le operazioni di lettura e scrittura di singoli byte sulla Block Ram RAMB16_S9.

Tali segnali interni sono mostrati nel port map del componente RAMB16_S9:

Listing 6.2: Port Map del componente RAMB16_S9 - collegamento ai segnali interni utilizzati per interfacciarsi alla Block Ram.

```
RAMB16_S9_inst : RAMB16_S9
  port map
  (
    DI => br_data_in (DATA_WIDTH-1 downto 0),
    DIP => br_parity_in,
    DO => br_data_out (DATA_WIDTH-1 downto 0),
```

```

DOP => br_parity_out,
ADDR => br_addr (ADDR_BIT-1 downto 0),
WE => br_we,
EN => br_en,
CLK => clk,
SSR => br_ssr
);

```

Mentre in scrittura il processo `blockram_sequential_access` gestisce correttamente la sequenza di scritture in quanto il contatore degli accessi aggiorna a ogni clock l'indirizzo in scrittura e il byte della linea da scrivere a tale indirizzo, in caso di lettura ciò non è altrettanto immediato.

Il motivo è che per leggere un byte in un ciclo di lettura, prima si fornisce alla Block Ram l'indirizzo a cui leggere il dato e si alza il segnale `en` per abilitare la lettura, ma poi bisogna attendere un tempo d'accesso in lettura per avere il dato richiesto (non essendo immediatamente disponibile sul bus dati in uscita dalla Block Ram `br_data_out`).

Per risolvere tale problema, dal momento che la Block Ram non dispone di un segnale di ready col quale avvisare che il byte letto è pronto e non potendo utilizzare un processo asincrono che si attiva con una commutazione del bus dati di output della Block Ram a causa di possibili letture di dati uguali (che non attiverebbero il processo), abbiamo determinato il tempo necessario per completare la lettura di un byte.

In questa fase abbiamo riscontrato che durante ogni operazione in cui si richiede di portare in uscita un dato presente sulla Block Ram ad un certo indirizzo (quindi durante le lettura, ma anche durante le scrittura per effetto delle `WRITE_MODE` da gestire), il dato viene reso disponibile in uscita al clock successivo e dopo un ritardo di 100 ps. L'effetto è pertanto che al primo clock si richiede una dato e lo si ottiene solo al secondo quando è in corso la richiesta del secondo dato. La Fig.6.2 mostra in dettaglio ciò che accade. `br_out` è il segnale di debug utilizzato per mostare il byte in uscita alla Block Ram

dopo ogni richiesta.

Consultando la User Guide di Xilinx (2) non abbiamo raccolto infor-

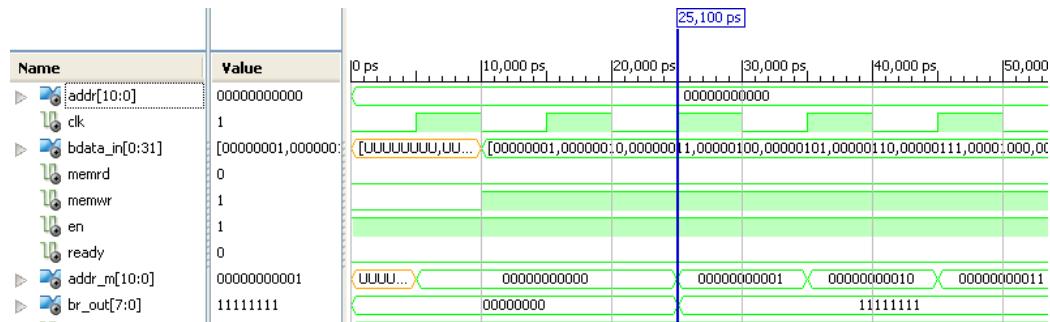


Figura 6.2: Dato richiesto ad un clock è reso disponibile in uscita al clock successivo + ritardo di 100 ps.

mazioni in merito alle temporizzazioni in lettura per avere un dato in uscita (dopo averne comandato una lettura), pertanto abbiamo cercato una spiegazione su forum del settore. Abbiamo riscontrato che i due clock per avere il dato in uscita sono necessari in quanto il process sincrono che sequenzializza l'accesso alla Block Ram utilizza lo stesso clock in ingresso alla Block Ram. Pertanto i due processi sincroni si attivano contemporaneamente e operano con assegnamenti sugli stessi segnali. Quindi il nostro process assegna i segnali di comando che dovrebbero comportare l'operazione voluta sulla Block Ram, ma per come avviene l'assegnamento dei segnali all'interno di un process, tali valori assegnati non sono ancora disponibili per la Block Ram e quindi vengono assegnati correttamente al clock successivo. Il ritardo di 100 ps è legato invece al fatto che la Block Ram è un componente reale e tale ritardo è utilizzato per simularne l'accesso con tempistiche non istantanee ai fini di simulazione.

Sulla base di tali considerazioni deriva la scelta di attendere il fronte di discesa del secondo clock prima di leggere il dato (che sicuramente sarà presente in uscita dalla Block Ram).

2.3 end_blockram_access

Il process `end_blockram_access` viene risvegliato ogni volta si attiva il segnale `line_ready` col quale il process `blockram_sequential_access` avvisa che il trasferimento della linea è stato completato. La responsabilità di tale process è quindi di attivare il segnale di `ready` e in caso di lettura fornire la linea letta all'esterno portandola in uscita su `bdata_out`.

Lo stesso, a fini didattici, accade in caso di scrittura al fine di testare il funzionamento della Block Ram con le diverse `WRITE_MODE`.

3 Testbench

Per verificare il funzionamento del componente abbiamo realizzato un semplice testbench nel quale si scrive all'indirizzo 0000h della Block Ram una linea di memoria (di dimensione 8 byte nel nostro esempio) passata in ingresso su `bdata_in`. Successivamente si effettua una lettura allo stesso indirizzo per verificare l'effettiva scrittura del dato in memoria.

Al fine di verificare le diverse modalità di funzionamento in scrittura della Block Ram descritte in precedenza, abbiamo provato lo stesso test sul componente configurando le tre differenti politiche di scrittura.

Il primo caso è quello di scrittura in modalità `READ_FIRST` in Fig.6.3. In questo caso l'operazione di scrittura di una linea sulla Block Ram, oltre all'attivazione del segnale di `ready`, comporta in uscita su `bdata_out` una linea di memoria il cui contenuto sono gli 8 byte (nel cerchio in giallo) presenti sulla Block Ram sulla base del contenuto iniziale impostato e che verranno sovrascritti.

Tale modalità consente quindi di verificare anche la corretta inizializzazione del contenuto della Block Ram fatta all'interno della sezione `generic(...)`.

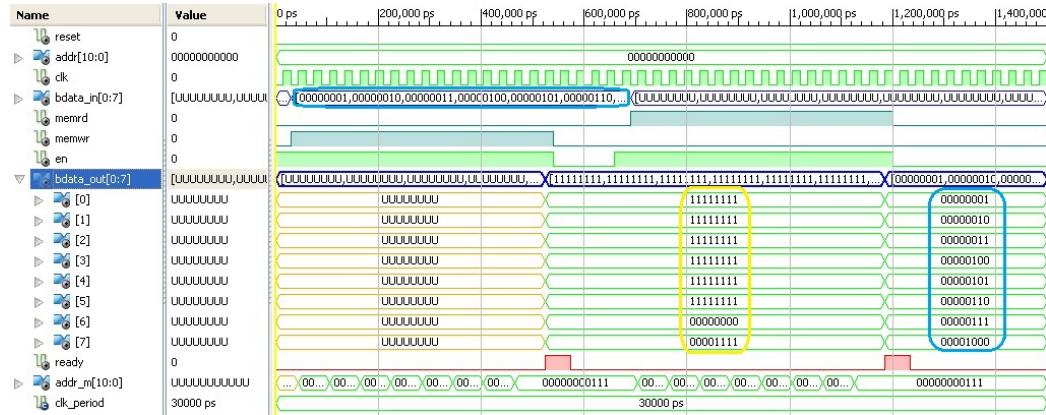


Figura 6.3: Simulazione con WRITE_MODE = READ_FIRST

Il secondo caso (in Fig.6.4) è caratterizzato dalla politica di scrittura di default, ovvero la WRITE_FIRST. Tale politica come mostra la simulazione, consente durante un ciclo di scrittura di avere contemporaneamente la scrittura del dato in ingresso sia in memoria all'indirizzo specificato sia sui latch di output.

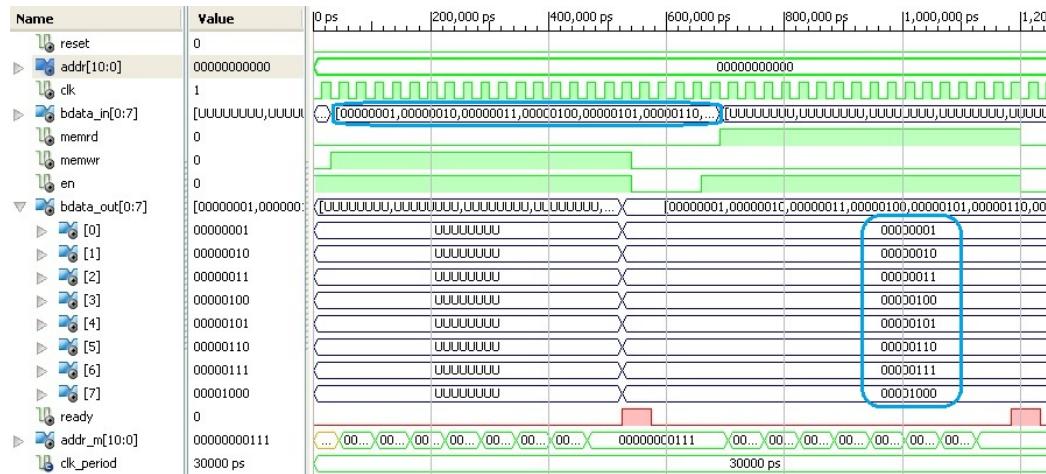


Figura 6.4: Simulazione con WRITE_MODE = WRITE_FIRST

Infine in Fig.6.5 abbiamo il caso di scrittura in modalità NO_CHANGE che non comporta alcun cambiamento del contenuto dei latch in output in caso di scrittura che rimane al valore nullo di default dal momento che non erano state fatte letture in precedenza.

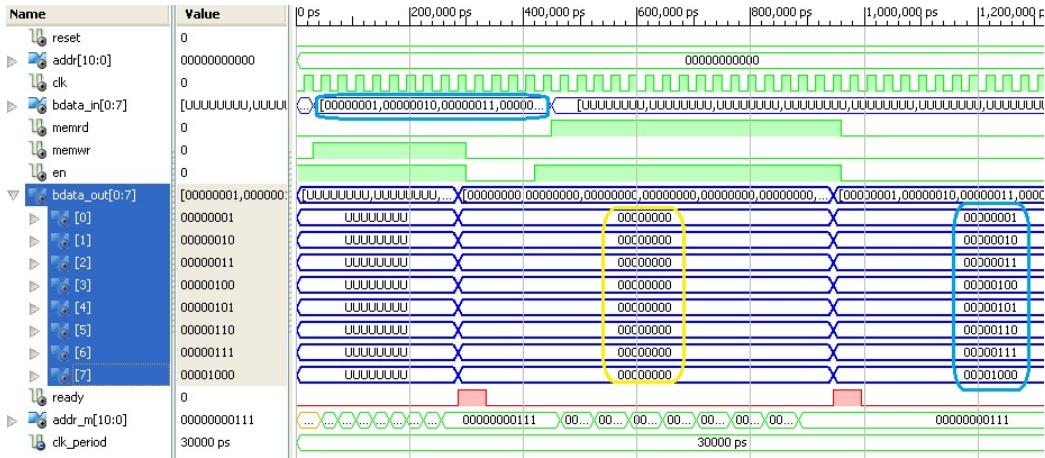


Figura 6.5: Simulazione con WRITE_MODE = NO_CHANGE

4 Considerazioni e Sviluppi Futuri

Il componente BlockRam_cmp rappresenta una memoria RAM a tutti gli effetti che prevede dei tempi d'accesso non nulli sia in scrittura che in lettura. Ciò comporta la necessità di considerare i tempi d'accesso alla memoria al fine di segnalare opportunamente (segnaletico di `ready`) alla cache quando possa leggere il dato richiesto (nel caso del nostro progetto che prevede il `ready` in ingresso alla cache per la sincronizzazione). Al termine dell'analisi del funzionamento della Block RAM e del componente realizzato abbiamo provato a integrarlo con la nostra Cache_cmp utilizzando i testbench precedenti, sostituendo il componente Ram_cmp con il nuovo. L'integrazione ha avuto esito positivo.

Un possibile sviluppo futuro del progetto potrebbe prevedere l'integrazione di BlockRam_cmp con il processore DLX. In particolare si dovrebbero gestire le temporizzazioni d'accesso alla Block RAM, contando gli stati di wait necessari e gestendo in modo opportuno il segnale di `ready`, in modo da informare il processore del completamento di un ciclo d'accesso alla memoria.

Dal punto di vista dell'implementazione interna del DLX ciò comporterebbe la necessità di stallare la pipeline qualora il `ready` non

sia asserito, operazione che in VHDL sarebbe molto semplice. Ad esempio si possono stallare tutti gli stadi della pipeline con una condizione if nei vari process che testa la presenza del ready e in caso di assenza non si fa nulla.

Conclusioni

Durante l'attività progettuale è stata realizzata una memoria cache per il processore DLX. In particolare si è realizzata una cache di tipo set-associative con numero di vie e dimensione configurabili. Il componente è stato realizzato in VHDL ed è testato sfruttando l'ambiente integrato di Xilinx.

La cache è stata poi integrata all'interno del progetto del DLX. Per verificare il corretto funzionamento della nuova versione del processore sono stati scritti diversi programmi in assembler che accedono ai dati presenti nelle memorie.

Infine si è analizzato il funzionamento della Block RAM presente all'interno dell'FPGA, la quale può essere sfruttata per il salvataggio di notevoli quantità di dati.

Per quanto riguarda le performance non sono stati ottenuti significativi miglioramenti rispetto al progetto originale del DLX poiché quest'ultimo integrava la RAM direttamente all'interno dello stadio di MEM. Tuttavia grazie alla BlockRAM integrata nell'FPGA è ora possibile realizzare un DLX dotato di una memoria RAM molto superiore rispetto al progetto iniziale.

Bibliografia

- (1) Xilinx (2009), *Spartan-3E FPGA Family: Data Sheet*
- (2) Xilinx (2009), *Spartan-3 Generation FPGA User Guide*