Architettura di un processore RISCV

Vittorio Polci

25 luglio 2023

Indice

1 Introduzione

Il progetto CustomCPU tratta della realizzazione di un primo prototipo di un processore RISCV con set di istruzioni a 32bit, sviluppato e sintetizzato interamente in codice VHLD tramite Vivado.
In particolare, il progetto prevede l'implementazione delle seguenti funzionalità:

- Quattro stadi operativi che eseguono operazioni in più cicli di clock.
- Tutto il set di istruzioni basilare, completo delle istruzioni di base di tipo R, I, S, J, U e B.
- Un sistema di pipelining che permette di eseguire al massimo quattro operazioni alla volta e che risolve eventuali situazioni di stall.
- Un interfaccia AXI per comunicare con memorie e dispositivi I/O memory mapped
- Una cache di primo livello di tipo write-trought.

Lo sviluppo dell'intero sistema prevede l'esecuzione di testbench per verificare se tutte le parti funzionano correttamente el'esecuzione di un programma finale dimostrativo per verificare la corretta esecuzione delle operazioni.

1.1 Struttura generale

Il processore è quindi formato da quattro stadi per l'esecuzione delle istruzioni, un sistema di controllo del pipeline e un'interfaccia per la comunicazione con le memorie principali. Nella figura 1 viene rappresentata una schematizzazione generale dei vari blocchi principali dell'architettura.

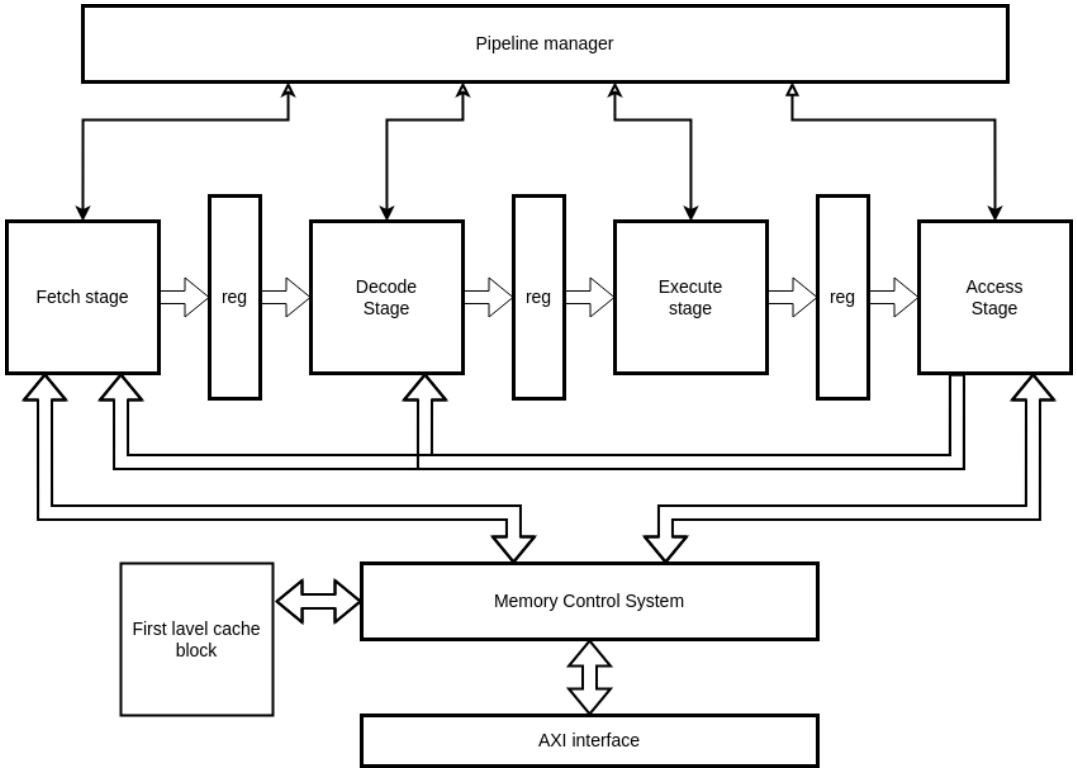

Figura 1: Schematizzazione generale della struttura del processore

Alla base di tutta la struttura è presente il *Memory Control System*, che ha il compito di gestire il flusso dei dati in lettura e in scrittura dal processore alla memoria, secondo l'architettura *von Neumann* (quindi memoria dati e istruzioni condivisa). Questa sezione provvede quindi ad eseguire tutte le richieste di lettura e di scrittura dal sistema principale e a allo stesso tempo provvede a controllare la

cache di primo livello associata e un bus AXI master, diretto verso l'esterno dell'architettura. Questa struttura è completamente scollegata dal sistema di pipelining del processore e si sincronizza con gli stadi *fetch* e *access* tramite un sistema di handshaking.

Il vero e proprio cuore del processore si trova invece nei quattro stadi di pipeline:

- *Fetch stage*
  Questo stadio ha il compito di gestire il Program Counter PC (il registro che contiene il puntatore all'istruzione corrente) e di richiedere ad ogni step di pipeline la prossima istruzione dal *memory control system*.

- *Decode stage*
  Questo stadio provvede alla decodifica dell'istruzione inviata dallo stadio di *fetch* in più dati utili allo stadio *execute* per eseguire le operazioni. La sua implementazione prevede, oltre alla decodifica dell'instruzione, a gestire tutta la parte dell'architettura relativa al register file, a ricavare i dati contenuti al suo interno e ad eseguire eventuali operazioni di writeback richieste dallo stadio *access*.

- *Execute stage*
  Questo stadio provvede ad eseguire le operazioni aritmetiche o logiche, a verificare eventuali condizioni e a calcolare eventuali indirizzi di salto, che verranno poi eseguiti nello stadio *Access*. Al suo interno è presente quindi una ALU, che verrà utilizzata anche più volte durante l'esecuzione di un singolo step di pipeline.

- *Access stage*
  Questo stadio provvede ad eseguire eventuali accessi nella memoria dati, a richiedere il salvataggio di eventuali dati all'interno del register file inoltrando le richieste allo stadio decode e controllare lo stadio *fetch* per eseguire eventuali salti. Esso permette quindi di accedere in lettura o in scrittura alla memoria dati richiedendo trasferimenti al *memory control system* e a reindirizzarli nei registri del processore.

Tutti gli stadi devono poi essere sincronizzati tramite il *pipeline manager* e registri intermedi per garantire la corretta esecuzione delle istruzioni e procedere a risolvere eventuali situazioni di stall provocati dal programma.
Il processore, infine, utilizza quindi singola *Interfaccia AXI* per comunicare con gli altri dispositivi, ricevere reset esterni e sincronizzarsi con il clock.

2 Memory control system

Durante la loro esecuzione, due stadi della pipeline potrebbero richiedere la lettura o la scrittura di dati all'interno della memoria. Per fare questo viene introdotta un'interfaccia dedicata proprio alla gestione del flusso dei dati.
Consideriamo la memoria come un oggetto capace di salvare dati al suo interno e renderli disponibili in tempi successivi su richiesta. In particolare, il processore deve avere a disposizione una memoria istruzioni, che fornisce periodicamente informazioni sulle operazioni da eseguire, e una memoria dati, utilizzata direttamente dai programmi per manipolare i dati.
In base proprio a come vengono organizzate le due memorie è possibile progettare due diversi tipi di architetture:

- Architettura Harvard, che è dotata di una memoria istruzioni separata da quella dei dati. In questo caso, mentre la prima è di sola lettura, la seconda può anche eseguire operazioni di scrittura al suo interno.

- Architettura von Neumann, che prevede l'utilizzo di una sola memoria che contiene sia le istruzioni che i dati. In questo caso, il processore è capace di interagire in scrittura anche sulle istruzioni.

Da un punto di vista architetturale, la prima soluzione è la più comoda, dato che la gestione delle due memorie viene associata a due rispettivi stadi di pipeline del processore. L'architettura dei processori RISCV è infatti basata sulla soluzione Harvard ma, in molti casi, questa suddivisione è apparente e serve per dare semplicità di rappresentazione e di design.
Molto spesso invece i processori utilizzano un'architettura von Neumann per evitare di utilizzare due diverse memorie e permettere il passaggio dei dati su un unico bus. Nel nostro caso, infatti, si è scelto di utilizzare un bus AXI per comunicare con i dispositivi esterni e, di conseguenza, il *memory control system* dovrà adattare la rappresentazione Harvard del core in modo da garantire un ordine nell'accesso dei dati.

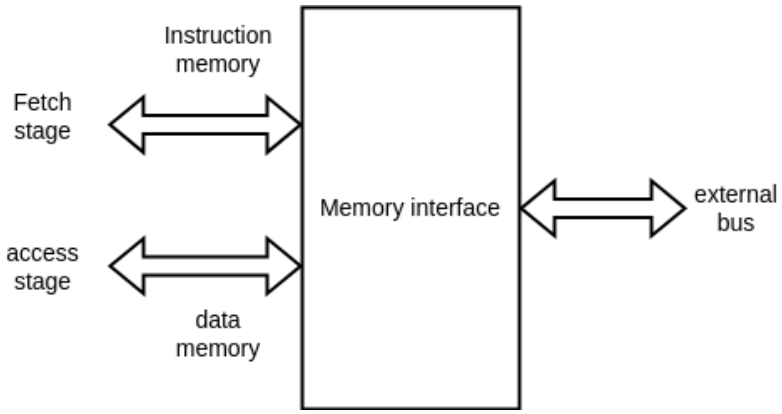

Figura 2: Schematizzazione generale del memory control system

Nella figura 2 è presente una schematizzazione generale del sistema da realizzare. In particolare, la macchina a stati dovrà poter ricevere ed elaborare richieste provenienti anche da due stati diversi di pieline del processore.
Altri due requisiti che questo sistema deve avere è, come detto precedentemente, una porta AXI master e un sistema in grado di rendere i trasferimenti più veloci tramite una cache.

Per realizzere il *Memory control system* è necessario prima creare delle componenti fondamentali:

- *memory bus interface*, che rappresenta il vero e prorio stadio che interagisce con l'interfaccia AXI.
- *cache system*, che controlla il *memory bus interface* appoggiandosi su una memoria cache per salvare le informazioni provenienti dai trasferimenti precedenti.
- Un doppio registro di pagina per eseguire la manipolazione dei dati secondo diverse casistiche.

Questi tre componenti, utilizzando un'altra macchina a stati, verranno poi utilizzati per formare il vero e proprio *memory control system*.

2.1 Memory bus interface

Il memory bus interface (MBI) è un sistema formato da due macchine a stati che riceve in input dati o richieste di accesso e traduce le informazioni sotto forma di trasferimenti AXI.
Il dispositivo è composto quindi da una parte di segnali I/O, che andranno poi utilizzati all'interno del *memory control system* per richiedere trasferimenti, ricevere esiti e fornire dati per la scrittura.
Le comunicazioni esterne seguono invece gli standard del protocollo AXI full e verranno unicamente utilizzate per transazioni in burst, dato che le comunicazioni prevederanno l'invio o la ricezione di pagine da più di 8 byte di dimensioni (nella prossima sezione verrà spiegato il motivo di questa scelta).
Tra le porte I/O si possono quindi trovare:

- *AXI master bus*, che rappresenta tutto l'insieme dei collegamenti che compongono il vero e proprio bus AXI.

- *Trasmission read start e Transmission write start*, due segnali in ingresso per ricevere eventuali rschieste di lettura o scrittura.
- *Transmission read started e Transmission write started*, un segnale output che indicare quando inizia una transmissione in scrittura o in lettura.
- *Transmission read address e trasmission write address*, utilizzati per ricevere gli indirizzi dei dati di partenza per le transazioni.
- *Transmission data in e Transmission data out*, utilizzati la comunicazione dei dati in ingresso o in uscita.

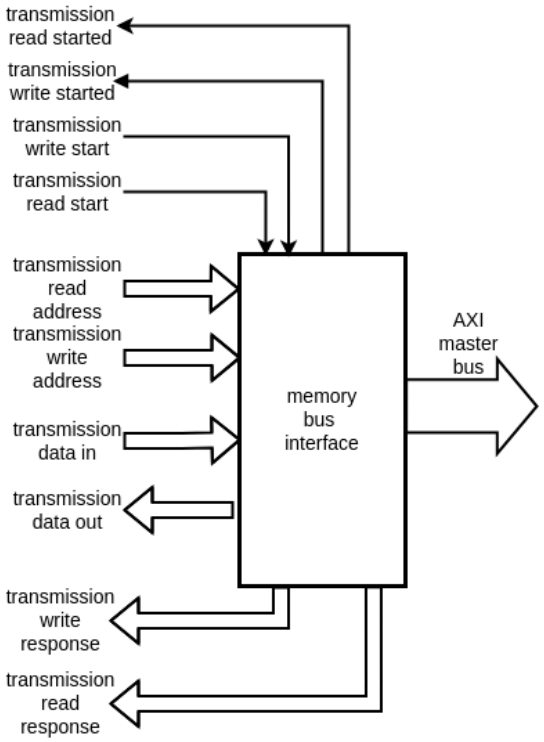

Figura 3: Rappresentazione del memory bus interface

Trasferimento in lettura

Quando il *memory control system* vuole eseguire una transazione in lettura, abilita l'ingresso *transmission read start* fornendo allo stesso tempo un indirizzo di 32 bit di memoria per indicare la posizione del primo dato da leggere. l'MBI, dopo aver ricevuto la richiesta, dovrà quindi procedere a impostare i valori corretti nel bus AXI per inviare l'indirizzo ricevuto e iniziare una transazione burst.
Una volta che il dispositivo AXI slave risponde alla richiesta di trasmissione, l'MBI potrà procedere ad abilitare il segnale *transmission read started*, per indicare a sua volta al *memory control system* che i dati stanno per essere trasmessi sulla porta *transmission data out* in parole da 32bit alla volta. Durante la trasmissione AXI burst l'MBI provvederà a reindirizzare tutti i dati in sequenza ad ogni ciclo di clock, fino a quando la transmissione non termina.
Al termine della trasmissione l'MBI provvederà poi a disattivare la linea *transmission read started* per indicare la fine trasmissione e provvederà a rendirizzare la risposta dello slave sulla linea *transmission read response*

Trasferimento in scrittura

Quando il *memory control system* vuole eseguire una transazione in scrittura, abilita l'ingresso *transmission write start* fornendo allo stesso tempo un indirizzo di 32 bit di memoria per indicare la

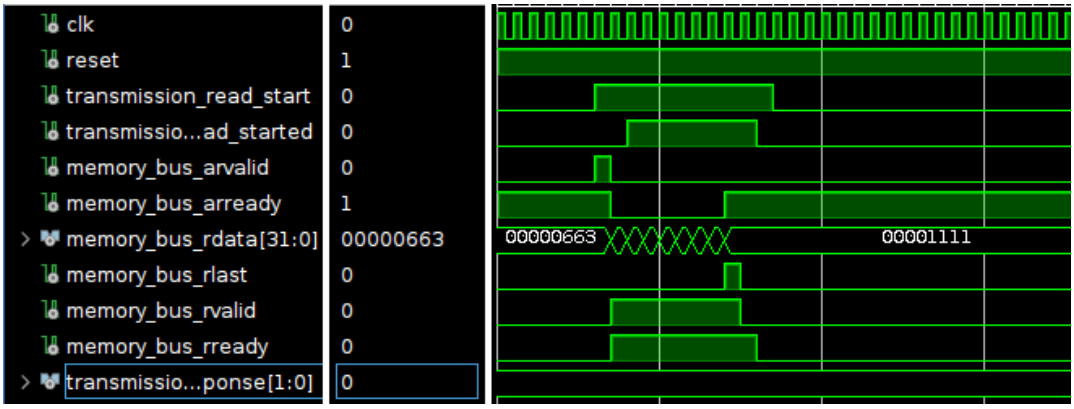

Figura 4: Trasferimento di dati in lettura sul bus AXI

posizione del primo dato da scrivere. l'MBI, dopo aver ricevuto la richiesta, dovrà quindi procedere a impostare i valori corretti nel bus AXI per inviare l'indirizzo ricevuto e iniziare una transazione in scrittura di tipo burst.
Una volta che il dispositivo AXI slave risponde alla richiesta di trasmissione, l'MBI potrà procedere ad abilitare il segnale *transmission write started*, per indicare a sua volta al *memory control system* che la porta *transmission data in* è pronta per ricevere i dati da scrivere in sequenza.
Durante la trasmissione AXI burst l'MBI provvederà a reindirizzare tutti i dati ricevuti in sequenza ad ogni ciclo di clock dal *memory control system* , fino a quando la transmissione non terminerà.
Al termine della trasmissione l'MBI provvederà poi a disattivare la linea *transmission write started* per indicare al *memory control system* la fine trasmissione e provvederà a rendirizzare la risposta dello slave sulla linea *transmission write response*

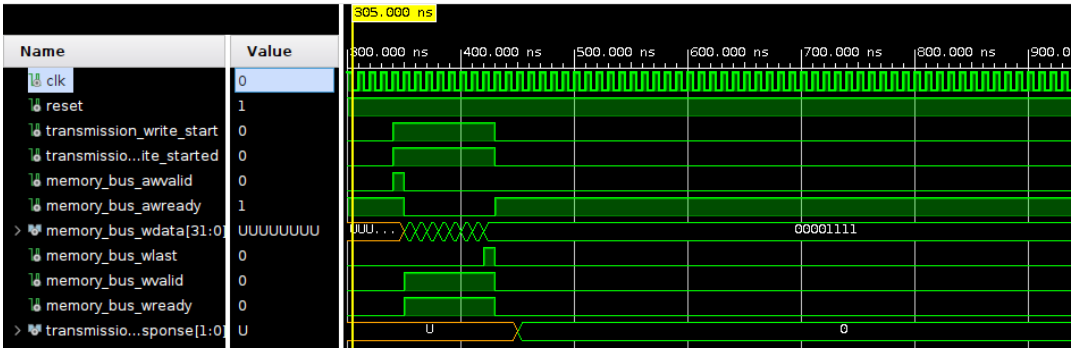

Figura 5: trasferimento di dati in scrittura sul bus AXI

Progettaione delle macchine a stati

Tutto il sistema viene implementato in due macchine a stati separate, in modo da poter garantire trasferimenti in scrittura e in lettura allo stesso momento (considizone che comunque non si verificherà mai).

2.2 Cache system

Il cache system ha due funzioni principali: controllare i trasferimenti fra il *memory control system* e il *memory bus interface* e memorizzare i dati letti nei blocchi della cache.
Consideriamo come cache un elemento di memoria molto veloci ma piccole, che vengono utilizzate per salvare informazioni che potrebbero essere riutilizzate in futuro. Nel nostro caso, la cache utilizzata

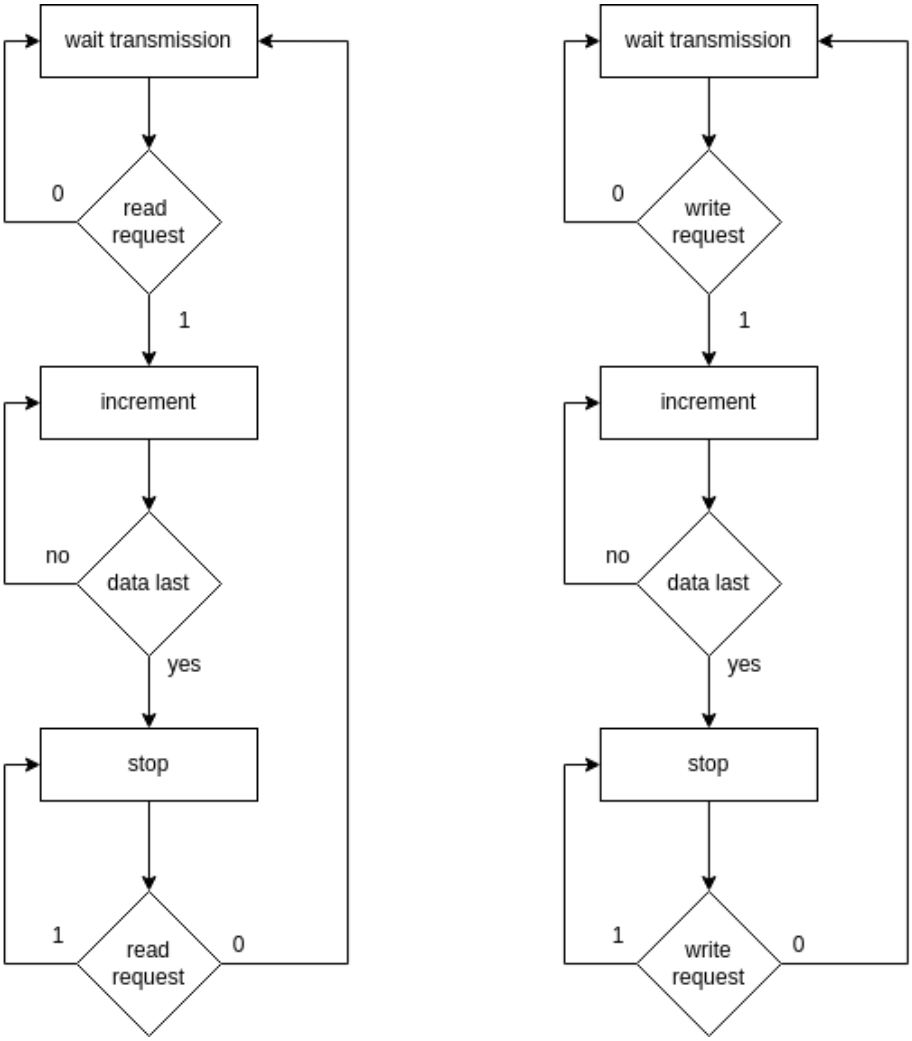

Figura 6: Macchine a stati utiizzate per coordinare le operazioni

è una *distribuited memory* generata automaticamente da Vivado tramite la funzione *Distribuited Memory Generator* presente nell'*IP catalog*.
Per la creazione della memoria cache viene scelta una dimensione dei dati maggiore rispetto a quella della lunghezza di una singola istruzione. Questo perchè, dato che il contenuto viene aggiornato mediante transazioni sul bus della memoria, conviene richiedere la trasmissione di più dati per volta, in modo da non dover ripetere l'operazione in seguito. Per questo motivo, la cache utilizzata è da 64 pagine di 256byte ciascuna.

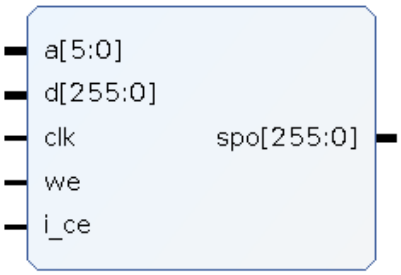

Figura 7: Rappresentazione della memoria cache utilizzata

La memoria generata può quindi memorizzare 32byte per volta, ovvero 8 istruzioni nel caso di una pagina contenente un programma. Questo garantisce quindi continuità dal punto di vista dei dati richiesti in sequenza.
Un altro elemento che verrà utilizzato all'interno del *cache system* sarà un altra mamoria distribuita formata sempre da 64 blocchi da 19bit ciascuno, che verrà poi utilizzata per identificare in tempo reale quali pagine sono memorizzate all'interno della cache. Il sistema di caching prevede l'utilizzo di un

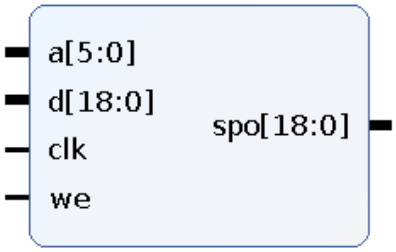

Figura 8: Rappresentazione della memoria indirizzi utilizzata

algoritmo di hashing per stabilire in che posizione della cache le pagine devono essere memorizzate.
Supponendo di prendere in considerazione l'indirizzo del dato che si desidera leggere (di una parola da 32bit o di un byte), è possibile suddividerlo di tre sottosezioni, in base a come abbiamo deciso di creare la cache. In particolare, si considerano i 18 bit più significativi *block identifier* dell'indirizzo

Figura 9: Suddivisione dell'indirizzo di memoria

come identificativi della pagina di memoria da voler caricare, i bit *block address* per ricavare l'indirizzo dove dovrebbe essere memorizzata la pagina nella cache e i restanti bit per identificare l'offset all'interno della pagina caricata si ottiene una cache a mappatura diretta.
In questo modo viene utilizzata la cache per memorizzare più sezioni dello spazio di indirizzamento secondo i bit *block address* mentre la memoria identificativa viene utilizzata per memorizzare i bit *block identifier* nella stessa cella di memoria relativa ai bit *block address*, in modo da tenere traccia di quale sezione ella memoria è stata caricata.
Il bit più significativo della memoria identificative viene invece utilizzato per indicare se la cella di cache è stata inizializzata con una pagina o è ancora vuota.
La costruzione del *cache system* prevede quindi la realizzazione di una macchina a stati che gestisce le due cache generate per memorizzare i dati e il *memory bus interface* per eseguire le transazioni con il bus AXI.
Nell'immagine 10 è possibile notare i seguenti collegamenti per la comunicazione con il *memory control system*:

- *Address in*, che rappresenta l'indirizzo della pagina di memoria (i 24 bit più significativi dell'indirizzo originale).
- *Data in e Data out*, che vengono usati per inviare o ricevere i dati di una pagina richiesta.
- *ready*, che viene usato dal *Cache system* per segnalare il sistema è pronto per una nuova transazione.
- *write*, che viene usato per segnalare al *cache system* che la transazione è in scrittura (o in lettura).
- *enable*, che viene uato per richiedere una transazione di dati.

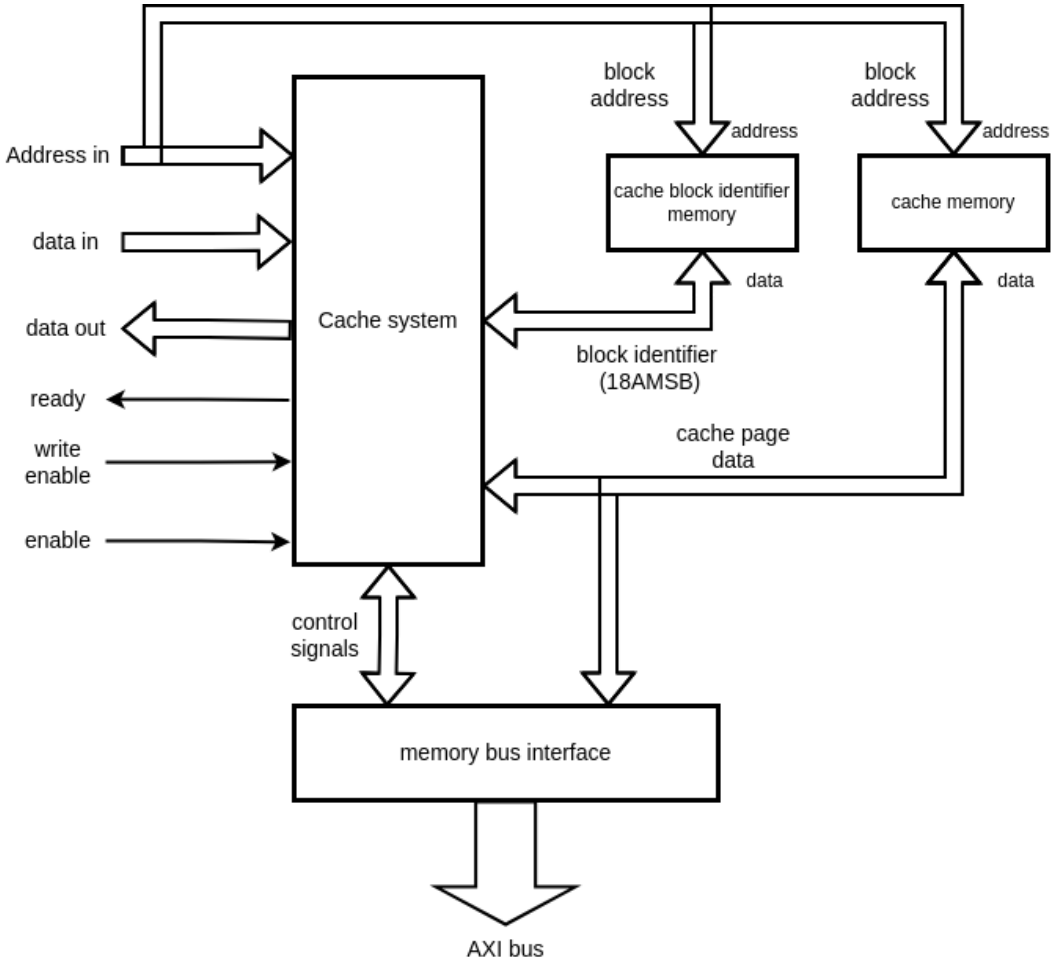

Figura 10: Rappresentazione del cache system (non vengono rappresentati i segnali per la gestione delle due memorie)

All'interno dello schema sono presenti inoltre altre linee per il passaggio dei dati all'interno della macchina a stati. Gli altri elementi presenti sono quindi:

- *Cache memory*, che memorizza le pagine su richiesta del *cache system*. Gli indirizzi vengono forniti direttamente dalla linea *address in*.
- *Cache block identifier*, che memorizza su richiesta la parte più significativa degli indirizzi ricevuti ed è indirizzata allo stesso modo della *Cache memory*
- *Memory control bus*, utilizzato nel caso in cui la pagina richiesta non viene trovata all'interno della cache.

Il *cache system* è quindi in grado di eseguire al massimo una transizione alla volta, contrariamente a come è stata sviluppato il *Memory control bus*. Il risultato è quindi molto simile a quello di una normale memoria dal punto di vista del suo utilizzo.

lettura di una pagina

Per leggere una pagina il *memory control system* deve abilitare il segnale *enable*, impostare la linea *write enable* a zero e segnalare nello stesso momento l'indirizzo della pagina che si desidera leggere. Una volta ricevuta la richiesta, il *cache system* disabilita il segnale *ready* e procede a verificare se la pagina è stata già caricata precedentemente richiedendo il dato contenuto nella memoria identificativa.

Se i dati corrispondono con i bit più significativi *address in* e la pagina della cache è stata inizializzata, allora verrà restituita nel ciclo di clock successivo l'intera pagina nella linea *data out*.

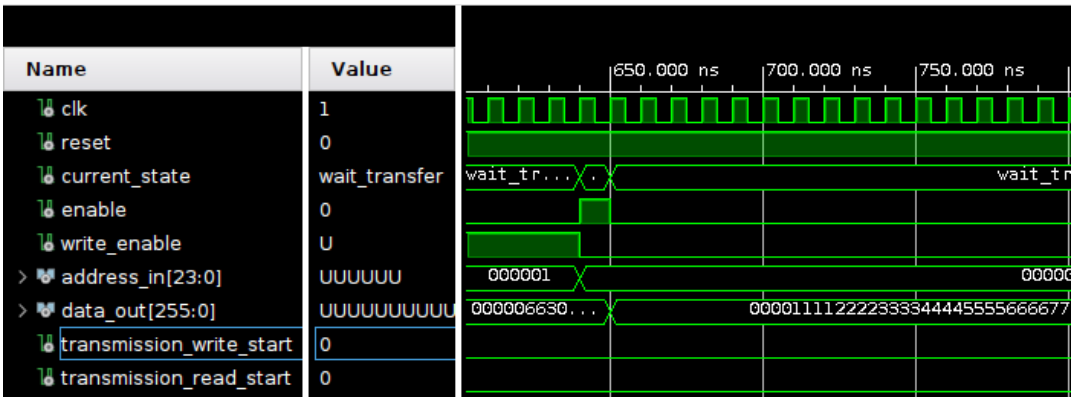

Figura 11: lettura di dati dalla cache

In caso contrario provvederà ad eseguire una transazione sul bus AXI tramite il *memory control bus* per richiedere i dati alla memoria esterna, li salverà all'interno della cache, aggiornerà l'indirizzo contenuto nella memoria identificativa e poi procederà a restituire i dati su *data out*.

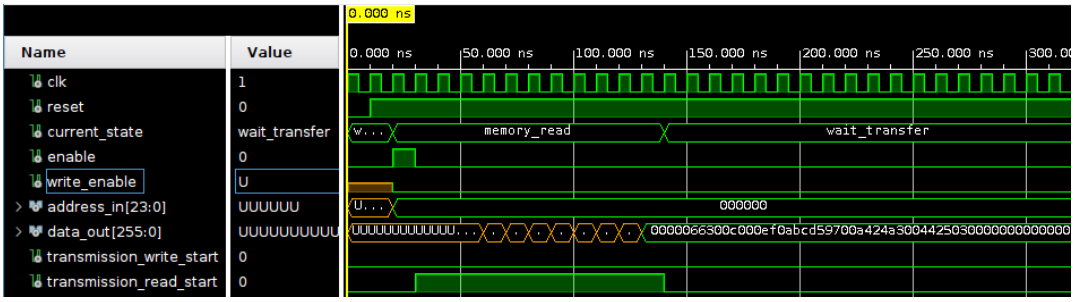

Figura 12: lettura dei dati dalla memoria

Una volta restituiti i dati, il *cache system* tornerà in attesa di un nuovo trasferimento riabilitando il segnale *ready*.

scrittura di una pagina

Per scrivere una pagina il *memory control system* deve abilitare il segnale *enable*, impostare la linea *write enable* a uno, segnalare nello stesso momento l'indirizzo della pagina che si desidera leggere e inviare i dati da scrivere sulla linea *data in*.
Una volta ricevuta la richiesta, il *cache system* disabilita il segnale *ready* e procede prima a scrivere nella pagina della cache i dati e poi ad eseguire una transazione sul bus AXI tramite il *memory control bus*, aggiornando anche i dati presenti nella memoria identificativa.
Una volta terminata la scrittura dei dati, il *cache system* tornerà in attesa di un nuovo trasferimento riabilitando il segnale *ready*.

Macchina a stati

La realizzazione della macchina a stati prevede quindi il controllo di tutti i segnali diretti ai sotto-componenti e il reindirizzamento dei dati.

2.3 Registro di pagina doppio

Come visto precedentemente, il *memory control system* comunica con la memoria principale e con la cache inviando e ricevendo pagine da 256byte e non singole parole o bype. Il problema principale, che

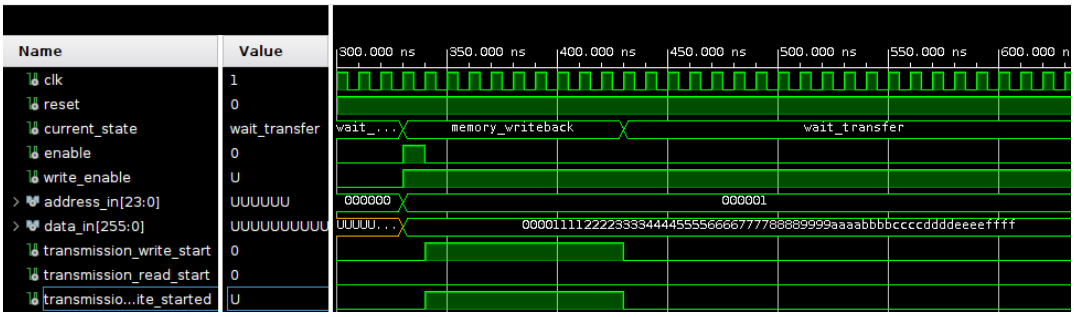

Figura 13: scrittura di dati nel cache system

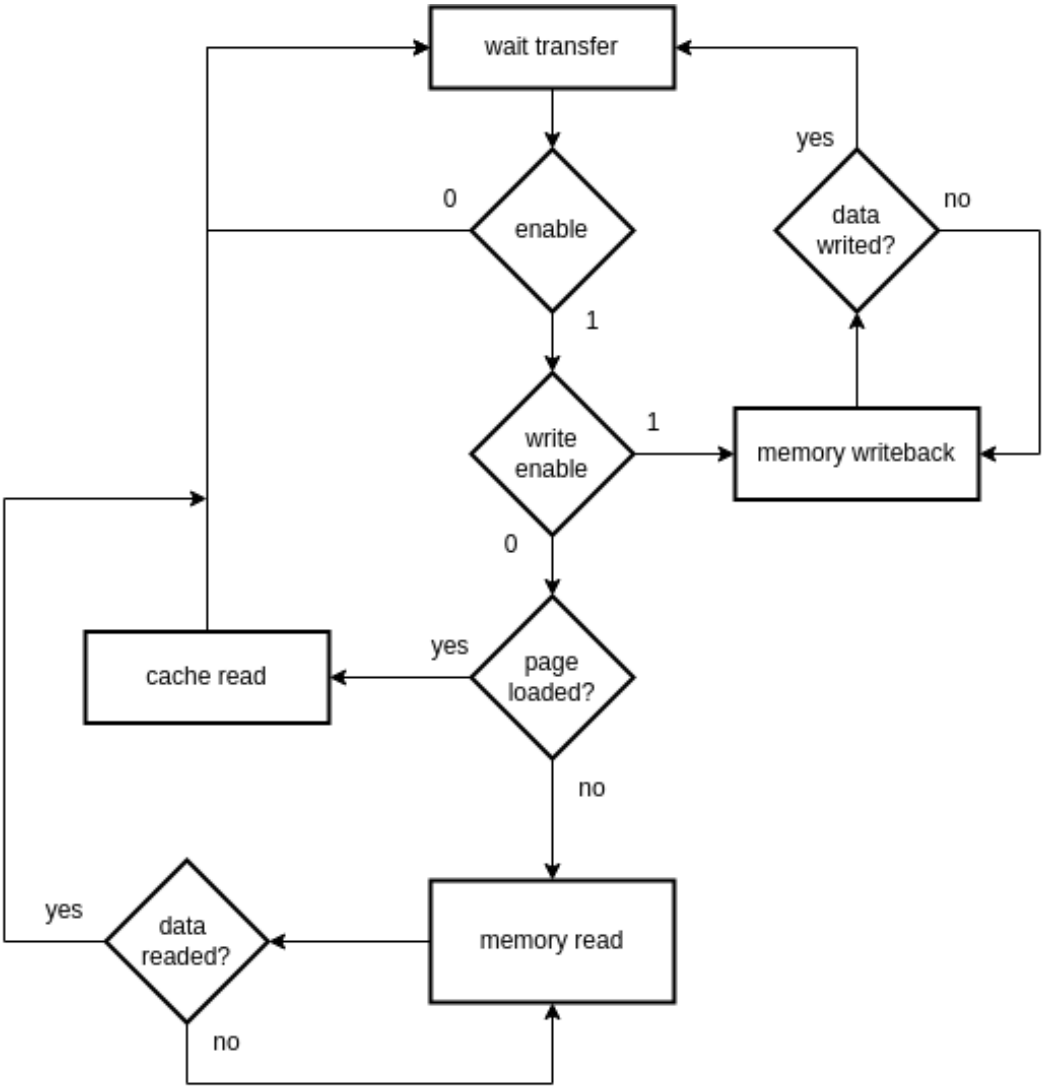

Figura 14: Rappresentazione della macchina a stati del cache system

verrà presentato successivamente dallo stadio *access*, è quello che il programma in esecuzione potrebbe richiedere un dati di dimensioni maggiori di un byte che si trovano in parte in una pagine e in parte in quella successiva.
Questo problema nasce proprio dal fatto che il processore, dal punto di vista delle istruzioni, indirizza dati da 8 bit e può salvare anche dati di dimensione differenti. Per risolvere il problema viene utilizzato

il registro di pagina doppio che, oltre a permettere la memorizzazione di due pagine concatenate, permette di leggere parole o scrivere dati in modi differenti.
Il registro di pagina doppio è dotato quindi di un elemento di memoria di dimensioni doppie rispetto ad una pagina normale (512 byte nel nostro caso) e da la possibilità di salvare le pagine concatenandole fra loro.

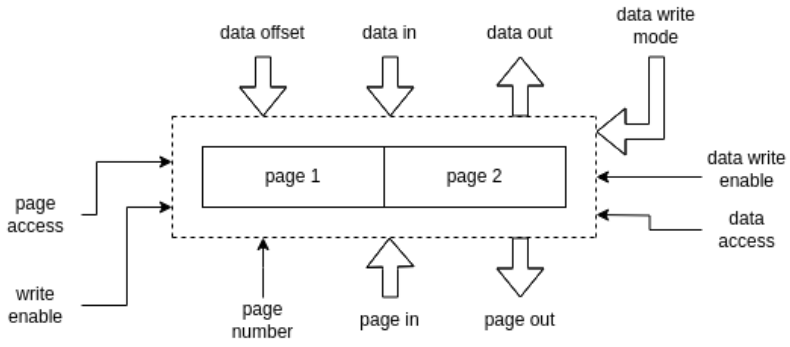

Figura 15: Rappresentazione del registro di pagina doppio

L'immagine 15 rappresenta schematicamente la struttura del registro. In particolare, l'interfaccia prevede una serie di segnali per la gestione dei dati in formato di pagina:

- *page access*, un ingresso dedicato per richiedere una lettura o una scrittura di una pagina all'interno del registro
- *page in e page out*, delle linee dedicate per inviare o ricevere i dati.
- *page number*, un ingresso che indica su quale zona del registro fare riferimento per la lettura o la scrittura della pagina.
- *page write enable*, un ingresso per insicare se l'accesso alla pagina è in lettua o in scrittura.

Il *memory control access* ha quindi la possibilità di inserire pagine all'interno del registro e rimuoverle in caso di modifica.
Un altra caratteristica del registro, come detto precedentemente, è quella di leggere parole o scrivere dati in formato differente. La struttura del registro prevede quindi diversi segnali per eseguire queste operazioni

- *data access*, un ingresso utilizzato per richiedere una lettura o una scrittura di dati nel registro.
- *data in e data out*, delle linee dedicate per inviare o ricevere dati in formato di parole dal registro.
- *data offset*, un ingresso che riceve l'offset dei dati da leggere o scrivere all'interno del registro.
- *data write mode*, un ingresso utilizzato per specificare come devono essere scritti i dati. In particolare, è possibile scrivere un byte, una parola a 16 bit o una parola a 32 bit all'interno del registro.
- *data write enable*, un ingresso utilizzato per indicare se la scrittura è in lettura o in scrittura.

La spiegazione del funzionamento della macchina a stati è più chiara nel caso di una rappresentazione in codice VHDL.

```
architecture Behavioral of cache_register is
signal data: std_logic_vector((page_dimension*2)-1 downto 0);
begin
    process (clk) is
        variable index: integer:=0;
    begin
        if (rising_edge(clk)) then
            if (reset <= '0') then
```

```
                data <= (others => '0');

-- verifica se viene richiesto un accesso di pagina
-- il funzionamento e' quello di un normale registro che salva dati con offset diversi

            elsif (page_access='1') then
                if (write_enable ='1') then
                    if (page_number = '0') then
                        data(page_dimension-1 downto 0)
                            <=page_in(page_dimension-1 downto 0);
                    else
                        data((page_dimension*2)-1 downto page_dimension)
                            <=page_in(page_dimension-1 downto 0);
                    end if;
                elsif (write_enable = '0') then
                    if (page_number = '0') then
                        page_out <= data(page_dimension-1 downto 0);
                    else
                        page_out <= data((page_dimension*2)-1 downto page_dimension);
                    end if;
                end if;

-- verifica se viene richiesto un accesso di dati
            elsif (data_access <= '1') then
                if (data_write = '0') then

.. in caso di lettura viene calcolato un index dall'indirizzo in ingresso
-- espresso in parole

                    index:=to_integer(unsigned(
                    data_address(data_address_dimension-1 downto 0)&"000"));

-- viene restiuita la parola a partire dall'index

                    data_out(7 downto 0) <= data(index+7 downto index);
                    data_out(15 downto 7)<= data(index+15 downto index+7);
                    data_out(23 downto 15) <= data(index+23 downto index+15);
                    data_out(31 downto 23) <= data(index+31 downto index+23);
                else
                    index:=to_integer(unsigned(
                        data_address(data_address_dimension-1 downto 0)&"000"));

-- in caso di scrittura viene verificata anche la modalita

                    case data_write_mode is
                        when "10" => -- write 32 bit word
                            data(index+7 downto index) <= data_in(7 downto 0);
                            data(index+15 downto index+7) <= data_in(15 downto 7);
                            data(index+23 downto index+15) <= data_in(23 downto 15);
                            data(index+31 downto index+23) <= data_in(31 downto 23);
                        when "01" => -- write 16 bit word
                            data(index+7 downto index) <= data_in(7 downto 0);
                            data(index+15 downto index+7) <= data_in(15 downto 7);
                        when others => -- write byte
                            data(index+7 downto index) <= data_in(7 downto 0);
                    end case;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

Tutte le operazioni richiedono un ciclo di clock per essere completate. Nel testbench 16 vengono inserite due pagine all'interno del registro, viene richiesta una lettura dei dati all'indirizzo 0x01 e poi viene eseguita una scrittura nell'indirizzo 0x04.

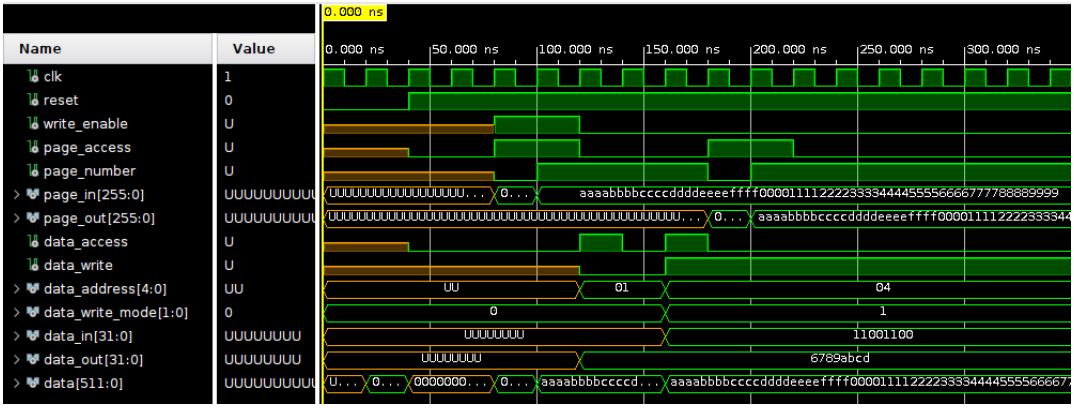

Figura 16: testbench del registro di pagina doppia

2.4 Top level

Una volta definita l'architettura di tutti i componenti necessari per il funzionamento è possibile procedere alla costruzione vera e propria del *memory control system*.

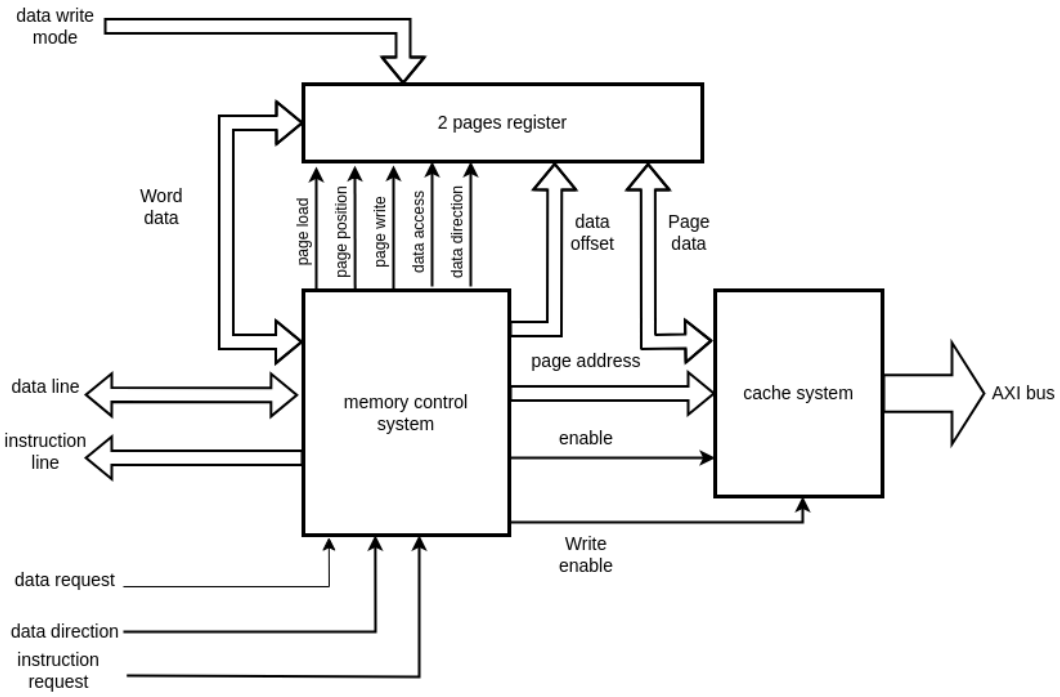

Figura 17: Architettua del memory control system

La figura 17 mostra come viene organizzata la struttura dello stadio finale utilizzato dal processore per interagire con la memoria.
Tra i segnali I/O del dispositivo si possono trovare:

- *data request*, un ingresso utilizzato per richiedere un accesso alla memoria dati.
- *instruction request*, un ingresso utilizzato per richiedere un accesso alla memoria istruzioni.
- *data direction*, un ingresso utilizzato per indicare se la richiesta alla memoria dati è in lettura o in scrittura.
- *data line*, composta da una linea dati in ingresso e una in uscita, viene utilizzata per comunicare con la memoria dati.

- *instruction line*, utilizzata per inviare una parola dalla memoria istruzioni (sola lettura).
- *data write mode*, una linea in ingresso utilizzata per indicare come scrivere nella memoria dati.
- *data ready*, un output utilizzato per indicare la disponibilià nel trasferimento dei dati dalla memoria dati.
- *instruction ready*, un output per indicare la disponibilità nel trasferimento dei dati dalla memoria istruzioni.

L'implementazione della macchina a stati (figura 18) è molto complicata, dato che richiede la gestione sia della memoria cache, sia del registro di pagina doppia.

Richiesta alla memoria dati

Il processore, quando vuole richiedere un accesso alla memoria dati, abilita l'ingresso *data request*, imposta l'indirizzo desiderato, imposta la direzione del trasferimento sulla linea *data direction* e inserisce eventuali dati per la scrittura.
Quando il *memor control system* riceve la richiesta, disabilita la linea *data ready* e provvede a verificare un eventuale offset nella pagina dati (per i modivi detti durante la spiegazione del registro di pagina doppia). Una variabile *word overflow* viene quindi impostata per ricordare agli stati futuri di operare su due pagine.
Dopo aver ricavato la variabile *word overflow*, la macchina a stati procede a caricare verificare se la pagina richiesta è stata caricata precedentemente nel registro. In caso positivo, si procederà a verificare se anche la seconda pagina richiesta è stata caricata (nel caso ci sia bisogno) e si passerà drettamente alla lettura o alla modifica dei dati.
Quando,tuttavia, manca una delle due pagine di memoria richieste, si procederà a caricarle.
Viene quindi fatta una richiesta di scrittura o lettura al registro di pagina e, solo successivamente, al posizionamento dei dati in output in caso di lettura o all'invio delle pagine modificare nel caso di una scrittura.
Successivamente, la macchina a stati imposta il segnale *data ready* e procede a verificare se è presente una richiesta alla memoria istruzioni. In caso affermativo procederà con le varie operazioni relative, mentre in caso negativo attenderà che l'ingresso *data request* torni a 0, in modo da sincronizzarsi con i stadi di pipeline del processore.

Tutte le operazioni di lettura di una pagina e di scrittura richiedono almeno l'utilizzo di due stadi differenti: uno per l'accesso dei dati nella cache e uno per l'accesso dei dati nel registro.

richiesta alla memoria istruzioni

Una richiesta alla memoria istruzioni procede in modo molto simile al caso della memoria dati. Dato che tutte le richieste vengono effettuate in sola lettura, sulla linea dedicata (*instruction request*,*instruction data*,*instruction ready* e *instruction address*).
Al completamento della richiesta la macchina a stati attenderà che gli ingressi *data request* e *instruction request* vengono disabilitati prima di tornare allo stato di idle, in modo da evitare ulteriori accessi non desiderati.

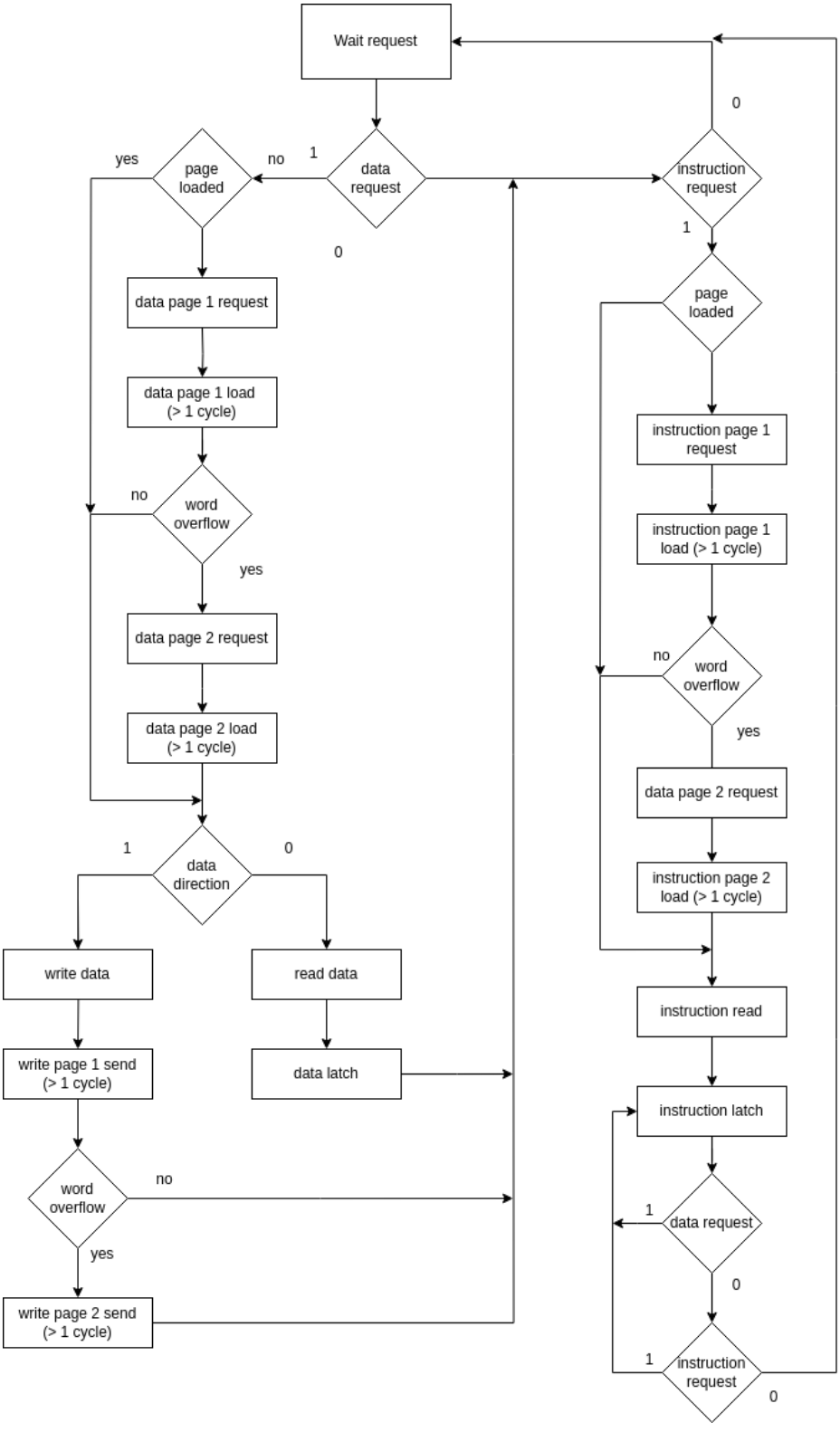

Figura 18: Diagramma di flusso del memory control system

3 Il core del processore

Per core del processore si intende l'insieme dei quattro stadi operativi che ricevono ed eseguono le istruzioni provenienti dalla memoria.

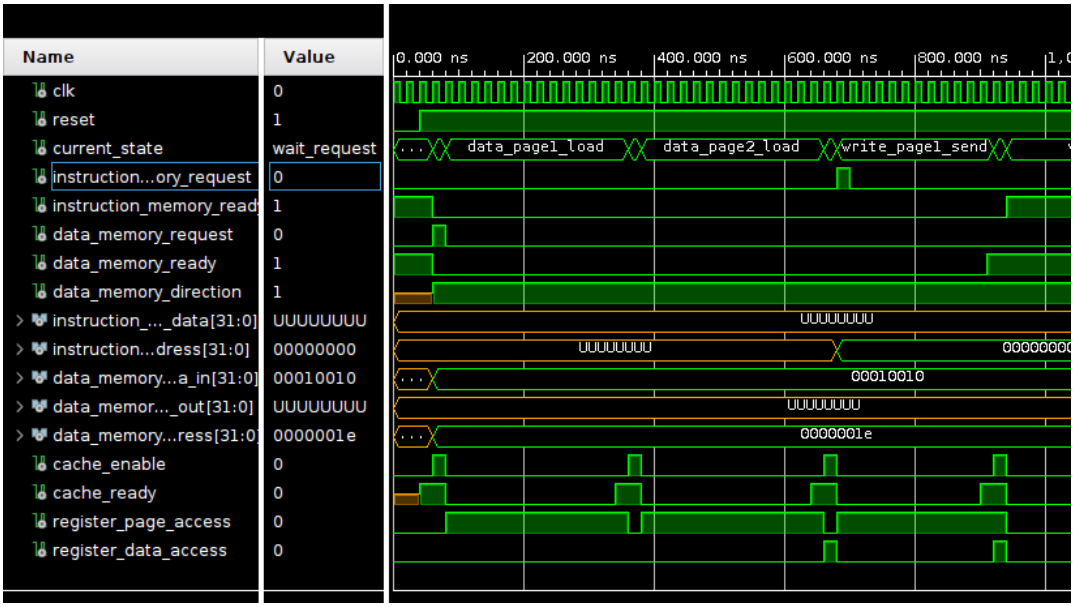

Figura 19: Scrittura di un dato nella memoria

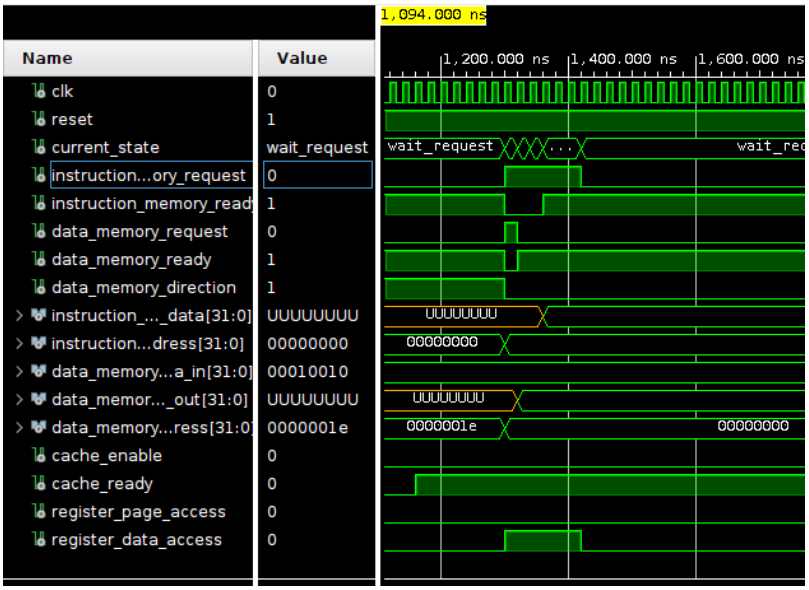

Figura 20: lettura contemporanea di dati da entrambi i dati

Come detto precedentemente, il processore è dotato di quattro stadi di pipeline che vengono sincronizzati fra loro tramite un sistema di gestione della pipeline dedicato.
La figura 21 mostra i collegamenti che collegano i quattro stadi fra loro e in quali punti vengono gestiti gli accessi in memoria. Il funzionamento dei collegamenti verrà approfondito nelle prossime sezioni e, successivamente, verranno inseriti nell'architettura anche tutti i segnali di controllo necessari alla gestione della pipeline.

3.1 Fetch stage

Il fetch stage è lo stadio che ha il compito di leggere le prossime istruzioni e gestire il registro PC. In particolare, il registro *Program Counter* è quello che tiene traccia della posizione corrente all'interno della memoria istruzioni, ovvero dell'indirizzo dell'istruzione che stata letta.

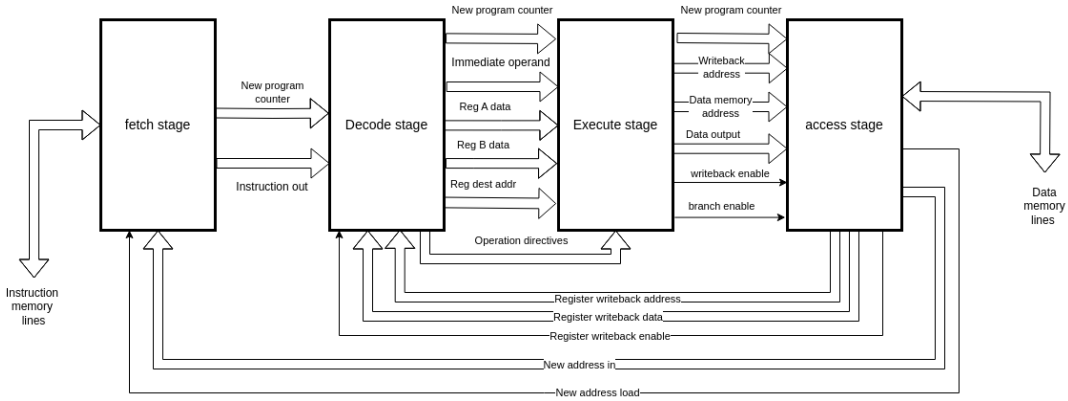

Figura 21: Struttura del core senza pipeline

Consideriamo come PC un semplice registro in cui è possibile inizializzare nuovi dati o incrementare il suo contenuto di 4 (dato che una singola istruzione è formata di 4 byte). Per descrivere il suo funzionamento si fa uso del codice VHDL descrittivo dell'architettura.

```
architecture Behavioral of program_counter is
signal current_address: unsigned(address_width-1 downto 0):=(others => '0');
begin
    process (clk, reset) begin
        if (rising_edge(clk)) then
            if (reset = '0') then
                current_address <= (others => '0');
            else
                if (load_enable='1') then
                    current_address<=unsigned(address_in);
                elsif (increment = '1') then
                    current_address<=current_address+4;
                end if;
            end if;
        end if;
    end process;
    address_out <= std_logic_vector(current_address);
end Behavioral;
```

In particolare, il registro è dotato di:

- Un ingresso *Load enable*, utilizzato per caricare dati specifici all'interno del registro.
- Un ingresso *Increment*, utilizzato per segnalare al registro di incrementare il suo contenuto di quattro.
- Un linea *address in*, utilizzata per specificare il dato da memorizzare all'interno del registro.
- Una linea *address out*, che viene usata per tenere traccia del contenuto del registro.

Tramite questo registro, lo stadio di fetch può, oltre a tenere traccia della posizione corrente nella memoria istruzioni, richiedere immediatamente i dati ad ogni step di pipeline al *memory control system* tramite le linee dedicate.
L'architettura dello stadio di *fetch* prevede quindi la costruzione di una macchina a stati in grado di eseguire tutte le operazioni descritte.
La figura 22 mostra la struttura interna dello stadio e i vari segnali necessari a controllarlo. In particolare, sono presenti:

- *address load enable*, un ingresso utilizzato dallo stadio di access per scrivere un indirizzo personalizzato all'interno del registro PC.
- *address in*, una linea in ingresso utilizzata dallo stadio *access* per indicare al registro PC il dato da registrare.

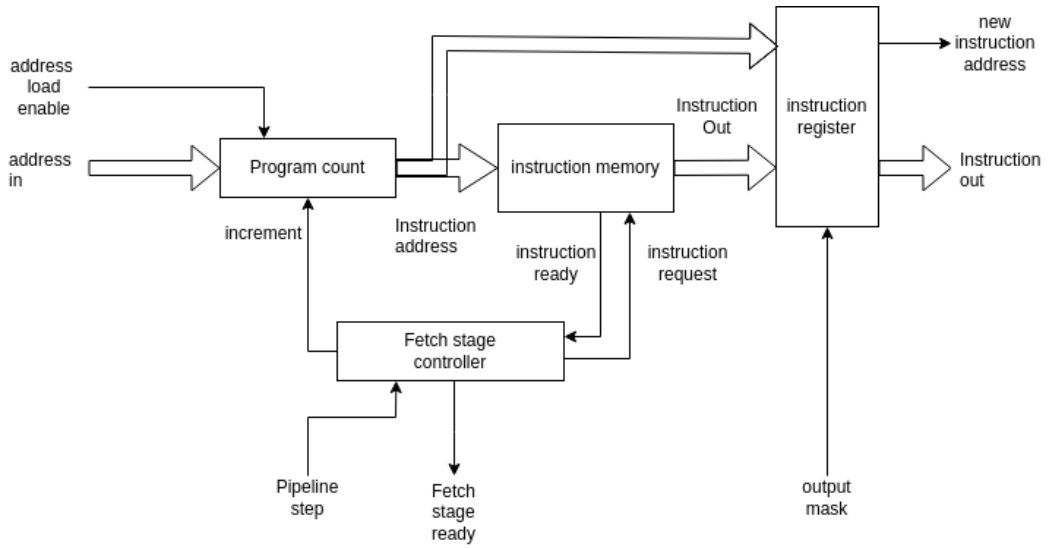

Figura 22: Struttura dello stadio di fetch

- *new instruciton address*, una linea output utilizzata per campionare il valore corrente del registro PC prima di essere incrementato.
- *instruction out*, una linea output utilizzata che prende il valore dell'istruzione appena caricata dalla memoria istruzioni.

Questi segnali serviranno poi come ingressi o uscite per la gestione delle operazioni da parte degli stadi *decode* e *access*.
Tra i segnali utilizzati dal sistema di pipeline sono presenti:

- *pipeline step*, utilizzato per segnalare allo stadio *fetch* per richiedere l'esecuzione di uno step di pipeline.
- *fetch stage ready*, utilizzato dallo stadio per indicare che è pronto per eseguire un nuovo step di pipeline.
- *output mask*, una linea input che inizializza i registri in uscita e il registro PC con dati di default.

Step di pipeline

Ad ongi step di pipeline, come detto precedentemente, lo stadio deve caricare l'istruzione all'indirizzo del registro PC richiedendo un accesso al *memory control system*, attendere che l'istruzione venga caricata, e procedere a registrarla in output con il valore di PC incrementato di quattro.
Per fare questo deve quindi attendere che la linea *pipeline step* venga abilitata e successivamente, oltre a disabilitare la linea *fetch stage ready*, procedere alla richiesta di una nuova istruzione, abilitando il segnale *instruction request* e fornendo l'indirizzo contenuto nel registro PC al *memory control system*.
A questo punto, dopo aver atteso il completamento dell'operazione da parte della memoria istruzioni (*instruction memory ready* abilitato), può disabilitare *instruction memory request*, reindirizzare l'istruzione caricata ai registri in uscita con l'indirizzo assegnato e incrementare il registro PC tramite la linea *increment*.
Dopo aver eseguito tutte queste operazioni, lo stadio di *fetch* è pronto per eseguire una nuova operazione e può riabilitare l'uscita *fetch stage ready* per segnalare al *pipeline manager* che è pronto per il prossimo step.

Inizializzione del registro

L'inizializzazione del registro può avvenire anche senza attendere lo step della pipeline. Questo perché, dato che lo stadio *access* richiede almeno un ciclo di clock per abilitare l'incresso *address load enable*,

lo stadio di *fetch* non sarebbe in grado di rilevare la richiesta e caricare l'istruzione dalla memoria (un problema verificato durante lo sviiluppo della pipeline).
Di consegenza, dopo aver ricevuto la richiesta di inizializzazione (*address load enable abilitato*), lo stadio di fetch attenderà che l'ingresso *address load enable* venga disabilitato per procedere poi a leggere nella memoria istruzioni senza attendere lo step di pipeline.

Mascheramento dei registri

Il mascheramento dei registri serve al pipeline manager per reimpostare l'indrizzo contenuto all'interno del registro PC e inizializzare i registri di output con dei valori predefiniti. In particolare, *new instruction address* viene forzato ad assumeer 0x00000000 e *instruction out* viene forzato ad assumere 0x00000013, che corrisponde alla codifica dell'istruzione NOP.
Queste funzionalità verranno poi richieste dal *pipeline manager* per predisporre gli ingressi in modo da garantire al prossimo stadio l'esecuzione di una specifica istruzione (NOP).
La reinizializzazione risulta quindi essere una richiesta di reset sincrona, che forza la macchina a stati a rimanere nello stato di attesa.

macchina a stati

La figura mostra lo sviluppo della macchina a stati dello stadio di fetch. Tra gli stati contenuti nella

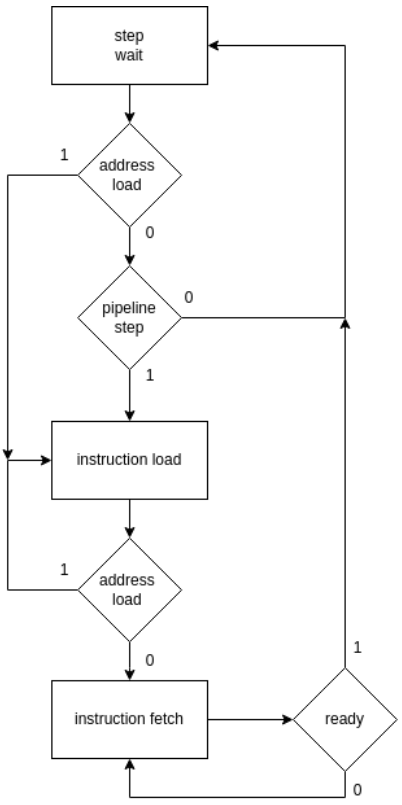

Figura 23: Struttura della macchina a stati dello stadio di fetch

macchina sono presenti:

- *step wait*
  Durante questo stato la macchima attende una richiesta di step o di inizializzatione. Nel primo caso, procederà a campionare il valore corrente del registro PC, ad abilitare il segnale *instruction request* e ad abilitare il segnale *increment* per richiedere al registro di incrementare il valore contenuto al suo interno.

Nel secondo caso procedera semplicemente a passare nello stato successivo, dato che l'inizializzazione del registro viene eseguita automaticamente.

- *instruction load*
  In questo stato la macchina provvede a disabilitare il segnale *increment* se precedentemente abilitato e a procedere allo stadio successivo.

- *instruction fetch*
  In qeusto stato, la macchina attende semplicemente il segnale *instruction ready*. Quando la memoria istruzioni sarà pronta, procederà a disabilitare il segnale *instruction memory request* e a salvare l'istruzione ricevuta nei registri di output, per poi tornare in stato di attesa.

Potrebbe succedere, nel caso in cui viene fatta una nuova inizializzazione da parte dello stadio *access*, che la macchina sii trovi ferma per più cicli di clock in *instruction load*. Questo, tuttavia, non genererà richieste multiple alla memoria istruzioni, dato che il *memory control system* deve attendere che il segnale *instruction request* venga disabilitato prima di procedere a passare nello stato di attesa.
L'istruzione viene quindi caricata una sola volta nel momento in cui viene ricevuta una richiesta di inizializzazione.
Per realizzare l'*output mask*, basta semplicemente scrivere qualche linea di codice in VHDL che salta tutto il sistema descritto precedentemente e inizializza le uscite.

```
if (output_mask = '0') then
                    -- macchina a stati
else
    current_state <= step_wait;
    new_address <= (others => '0');
    instruction_out <= x"00000013";
end if;
```

Vengono mostrati successivamente i risultati dei testbench relativi a due situazioni di pipeline.

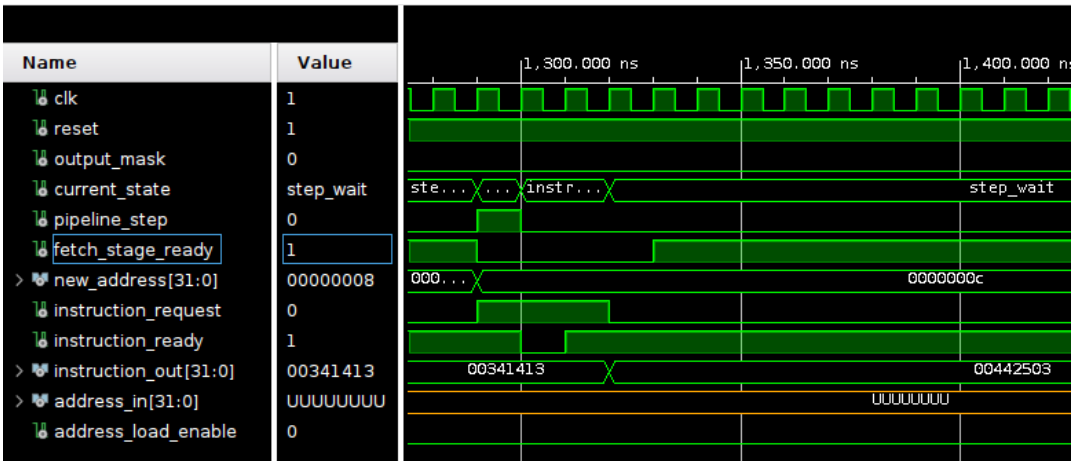

Figura 24: Simulazione di uno stato di step di pipeline normale

Nel testbench mostrato nella figura 23 viene mostrato come la macchina reagisce in condizioni normali.

Nel testbench mostrato nella figura 25 viene invece mostrato come la macchina reagisce in condizioni di branch, ovvero quando lo stadio *access* richede una modifica del registro.

3.2 Decode stage

Lo stadio *decode* ha il compito di decodificare le istruzioni che sono tate inviate dallo stadio di *fetch* in modo da ricavare da esso tutti gli operandi necessari per eseguire l'istruzione nel prossimo step.
COme detto nell'introduzione, nel processore devono essere sviluppate tutte le istruzioni base, ovvero tutte le istruzioni che prevedono operazioni aritmetiche, logiche, di salto condizionato o non. Tra le varie tipologie di istruzioni possiamo trovare quindi:

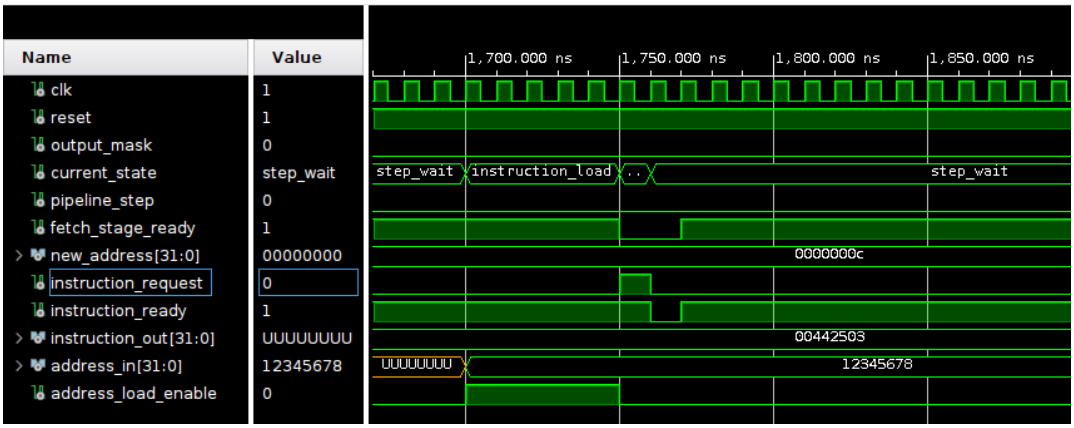

Figura 25: Simulazione di uno stato di load enable

- Tipo *R*, ovvero tutte le istruzioni aritmetiche che prevedono due operandi nei registri e un registro di destinazione.
- Tipo *I*, ovvero tutte le istruzioni aritmetche che prevedono il primo operando nel registro, il secondo codificato direttamente nell'istruzione e un registro di destinazione.
- Tipo *R shift*, ovvero le istruzioni aritmetiche di shift che prevedono uno dei due operandi come immediato (Istruzioni di tipo R ma non prevedono un registro come operando) e un registro di destinazione.
- Tipo *S*, ovvero tutte le istruzioni per salvare i dati da un registro nella memoria dati.
- Tipo *I load*, ovvero tutte le istruzioni per leggere i dati dalla memoria e salvarli in un registro.
- Tipo *B*, ovvero tutte le istruzioni di salto condizionato che richiedono due registri da confrontare.
- Tipo *J*, ovvero l'istruzione JAL per il salto incondizionato.
- Tipo *I JALR*, ovvero l'istruzione JALR utilzzata per fare un salto incondizionato a partire dall'indirizzo memorizzato in un registro.
- Tipo *U LUI*, ovvero l'istruzione LUI per inizializzare la parte più significativa di un registro con un dato immediato.
- Tipo *U AUIPC*, ovvero l'istruzione AUIPC che somma un valore immediato al program counter e salva il risultato in un registro.

Alcune istruzioni che appartengono allo stesso tipo, come AUIPC e LUI, venngono diversificate dagli operandi ma rimangono simili nella posizione dei dati.
Lo stadio di decode deve quindi ricavare le informazioni dalle istruzioni, ricavare eventuali operandi immmediati utilizzando un'estensore di segno e creare una codifica delle operazioni da eseguire, che servirà poi per indicare allo stadio di *execute* quale operazione deve svolgere.

decodifica combinatoria

La decodifica combinatoria rappresenta quella parte dello stadio di fetch che provvede a ricavare le informazioni dall'istruzione ricevuta dallo stadio di *fetch* ,restituendo le seguenti informazioni:

- *register address A*, ovvero l'indirizzo del registro del primo operando (sempre presente nelle istruzioni).
- *register address B*, ovvero l'indirizzo di un eventuale registro del secondo operando.

- *immediate value*, ovvero il valore immediato presente in alcune istruzioni, che viene codificato in modo diverso in base al loro tipo.
- *instruction type*, che distingue il tipo di operazione da eseguire secondo lo schema delle istruzioni precedente.
- *ALU control*, che semplicemente racchiude un bit del campo *funz7* e i bit del campo *funz3* per specificare la diversificazione delle operazioni della stessa tipologia.

In base quindi ai requisiti specificati, è possibile costruire una componente combinatoria in VHDL che restituisce tutte le informazioni necessarie da esegure nello stadio di decode.

```
-- gli indirizzi dei registri sorgente e di quello di destinazione
-- vengono ricarati subito, anche per le istruzioni che non li richiedono.
register_address_dest <= instruction_in(11 downto 7);
register_address_a <= instruction_in(19 downto 15);
register_address_b <= instruction_in(24 downto 20);

-- il valore immediato viene ricavato in base al tipo di istruzione ed eventualmente
-- esteso (*** implica un estensione di segno non visualizzata)

immediate_value <=  -- U type
                    (instruction_in (31 downto 12) & "000000000000")
                            when (op_code="0110111" or op_code="0010111") else

                    -- J type
                    (instruction_in(31) & ***
                        & instruction_in(31) & instruction_in(19 downto 12) &
                        instruction_in(20) & instruction_in(30 downto 21) & "0")
                        when (op_code = "1101111") else

                    -- I type JALR
                    (instruction_in(31) & ***
                            & instruction_in (31 downto 20))
                            when (op_code="1100111") else

                    -- B type
                    (instruction_in(31) & ***
                        &instruction_in(31) & instruction_in(7) &
                        instruction_in (30 downto 25) & instruction_in(11 downto 8) & "0")
                        when (op_code="1100011") else

                    --I type aritmetic
                    (instruction_in(31) & ***
                        instruction_in(31 downto 20))
                        when (op_code="0000011" or (op_code="0010011" and
                                not(funct3 = "001" or funct3 = "101"))) else

                    --I type load
                    ("00000000000000000000" & instruction_in(31 downto 25)
                        & instruction_in(11 downto 7))
                        when (op_code="0100011") else

                    -- R type shift
                    ("0000000000000000000000000000" & instruction_in(24 downto 20))
                        when (op_code="0010011" and (funct3 = "001" or funct3 = "101")) else
                    (others => '-');

-- il tipo di istruzione cambia a seconda del campo opcode e funz3 (solo in alcuni casi)
instruction_type <= "0000" when (op_code="0110111") else   -- U type LUI
                    "1001" when (op_code="0010111") else   -- U type AUIPC
                    "0001" when (op_code="1101111") else   -- J type
                    "0010" when (op_code="1100111") else   -- I type JALR
                    "0011" when (op_code="1100011") else   -- B type
                    "0100" when (op_code="0000011") else   -- I type Load
                    "0101" when (op_code="0100011") else   -- S type

                    -- I type arithmetic
                    "0110" when (op_code="0010011" and
```

```
                not(funct3 = "001" or funct3 = "101")) else

            -- R type shift
            "1000" when (op_code="0010011" and
                (funct3 = "001" or funct3 = "101")) else

            "0111" when (op_code="0110011") else    -- R type
            "____";

-- il campo alu_control e' sostanzialmente funz3, a cui vengono aggiunte eventuali
-- informazioni
with funct7 select
alu_control <= "00" & funct3 when "0000000",
               "01" & funct3 when "0100000",
               "00" & funct3 when others;
```

Le informazioni ricavate serviranno quindi nel passaggio successivo dello stadio di decode, che consiste nella lettura dei dati all'interno del register file.

Register file

Per register file si intende l'insieme di registri che sono contenuti all'interno del processore. Nel caso di un processore RISCV a 32bit, il numero di registri general purpose (quelli utilizzabili dalle istruzioni) è 32 e il primo (il registro x0) restituisce un dato fisso 0x0000000 e viene utilizzato per operazioni che non richiedono un secondo operando (ad esempio uno spostamento di dati fra due registri).
Per realizzare un register file viene utilizzata la funzione *Distribuited Memory Generator* di Vivado, che crea automaticamente un array di registri con una latenza di un solo ciclo di clock.
Per semplicità, viene quindi generato una memoria formata da 32 celle di 32bit ciascuna, che ha due porte per la lettura e una per la scrittura, in modo da poter ricavare entrambi gli operandi nei registri senza aver bisogno di due cicli di clock.

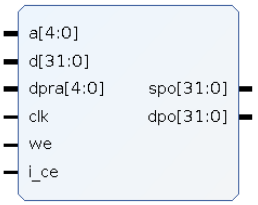

Figura 26: struttura del register file

La memoria generata viene effettivamente controllata da una macchina a stati, che coordina le operazioni di lettura dello stadio di *decode* con eventiali scritture provenienti dallo stadio *access*.
Infatti, dato che gli stadi *access* e *decode* vengono spesso eseguiti nello stesso momento, è possibile che si verifichi una lettura dei dati prima di una scrittura, operazione che potrebbe generare delle inconsistenze. Lo schema rappresentato nella figura 27 mostra come la macchina deve essere interfacciata al register file. In particolare, tra i vari collegamenti possiamo trovare:

- *read request*, un ingresso utilizzato per richiedere un trasferimento in lettura.
- *Address A*, l'indirizzo del primo operando da leggere.
- *Address B*, l'indirizzo del secondo operando da leggere.
- *Address Write*, l'indirizzo del registro da sovrascrivere.
- *data A*, la linea utilizzata per inviare l'operando A.
- *data B*, la linea utilizzata per inviare l'operando B.
- *data write*, la linea utilizzata per ricevere i dati da scrivere nel registro.
- *write request*, la linea utilizzate per richiedere un trasferimento di dati in scrittura.
- *ready*, un uscita utilizzata dalla macchina per indicare che l'operazione è stata completata.

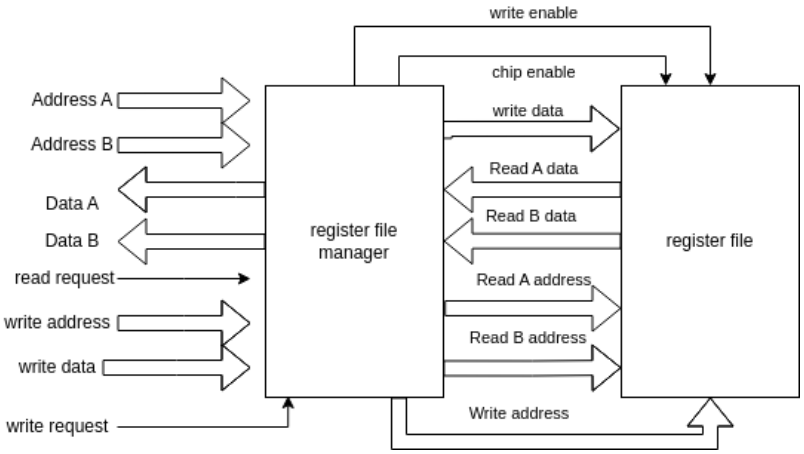

Figura 27: struttura del register file manager

lettura dei dati

Molto spesso la macchina a stati deve acconsentire a richieste di lettura di due operandi. In particolare, per eseguire un'operazione di lettura doppia è necessario abilitare l'ingresso *read request*, disabilitare *write request* e fornire gli indirizzi che si vogliono leggere.
Successivamente, la macchina riabiliterà l'uscita *ready* per indicare che ha ricevuto la richiesta e procederà a leggere il contenuto dei registri richiesti inoltrando la richiesta al register file.
Una volta eseguita la lettura (dopo un ciclo di clock), la macchina a stati reindirizzerà i dati alle uscite designate e abiliterà nuovamente l'uscita *ready* per indicare che è pronta per il trasferimento successivo.

lettura e scrittura contemporanea

Nel caso in cui il segnale *read enable* venga abilitato contemporaneamente a *write enable* la macchina a stati deve prima procedere alla scrittura dei dati e poi alla successiva lettura.
La macchina a stati disabiliterà quindi l'uscita *ready* e provvederà ad eseguire la scrittura nel primo ciclo di clock successivo e la lettura in quello seguente, per poi abilitare l'uscita *ready* una volta concluse le operazioni.
L'ordine di priorità viene dati quindi prima alla scrittura, in modo da garantire integrita nei dati salvati.

Un funzionamento molto simile si ha quando si richiede solo una scrittura (condizione che non si verificherà mai durante l'esecuzione della pipeline).

macchina a stati

La figura 28 rappresenta il funzionamento della macchina a stati. In particolare:

- Durante lo stato *wait request*, la macchina abilita l'uscita *ready* e attende un nuovo trasferimento. Se il trasferimento richiederà una lettura procederà direttamente ad andare nello stato *register read* mentre nel caso in cui è necessario eseguire anche una scrittura andrà nello stato *register write*.

- Durante lo stato *register write*, la macchina abiliterà gli ingressi *i ce* e *write enable* per permettere la scrittura dei dati ricevuti nell'indirizzo assegnato. Se è necessaria un eventuale lettura dei dati allora la macchina entrerà anche nello stato *read register* prima di tornare in *wait request*.

- Durante lo stato *register read*, la macchina abilita l'ingress *i ce* per effettuare una doppia lettura e torna nello stato *wait request*, per poi mettere in output i dati ricevuti.

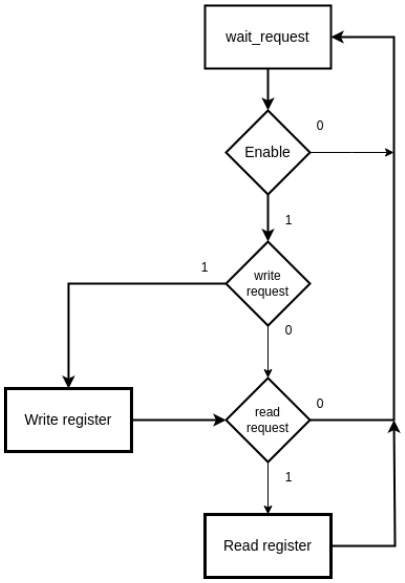

Figura 28: funzionamento del register file manager

Nel caso in cui venga inserito un indirizzo pari a 0x00000000 in lettura la macchina a stati restituirà sempre 0x00000000, come mostrato nel codice VHDL.

```
register_a_output <= (others => '0') when address_a="00000" else
                     register_array_data_out_a;
register_b_output <= (others => '0') when address_b="00000" else
                     register_array_data_out_b;
```

Top level

La figura 29 mostra come vengono uniti i componenti appena creati.
Tra i collegamenti dello stadio troviamo:

- *Instruction IN*, utilizzato per ricevere l'istruzione in ingresso da decodificare.
- *operation type e alu control*, utilizzati per segnalare allo stadio *execute* come procedere al prossimo step di pipeline.
- *Immegiate operand, Register A output e Register B output*, utilizzati per inviare gli operandi necessari per l'esecuzione.
- *Destination register address*, utilizzato per inviare l'indirizzo del registro di destinazione ai prossimi stadi.
- *write request, write register address e write register data*, che sono i collegamenti diretti con il *register file manager* per la scrittura dei dati da parte dello stadio *access*.

Come per lo stadio di *fetch*, sono presenti anche dei collegamenti relativi alla gestione della pipeline:

- *Pipeline register A e Pipeline register B*, ovvero gli indirizzi dei registri A e B che verranno usati dal controller di pipeline per identificare eventuali stall.
- *decode stage ready*, usato dallo stadio *decode* per segnalare che è pronto per un nuovo step.
- *output mask*, usato dal *pipeline manager* per inizializzare la macchina a stati e i registri di output.
- *pipeline step*, usato per segnalare alla macchina a stati di procedere un un nuovo step di pipeline.

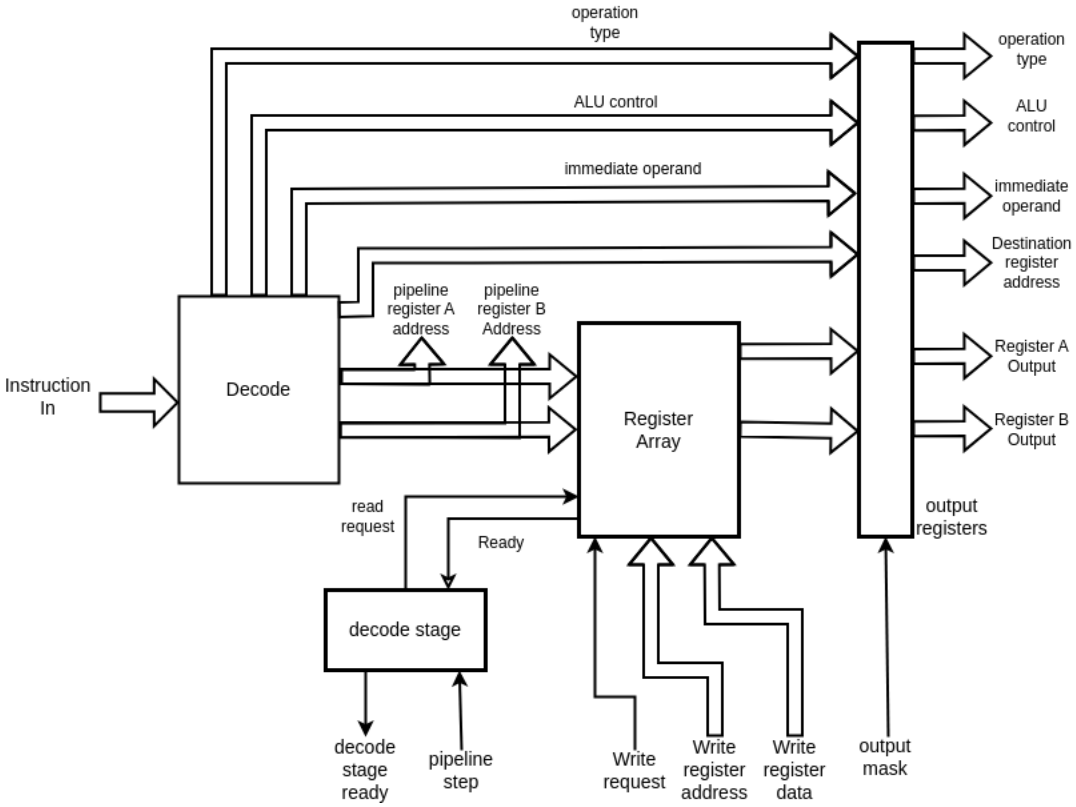

Figura 29: funzionamento del register file manager

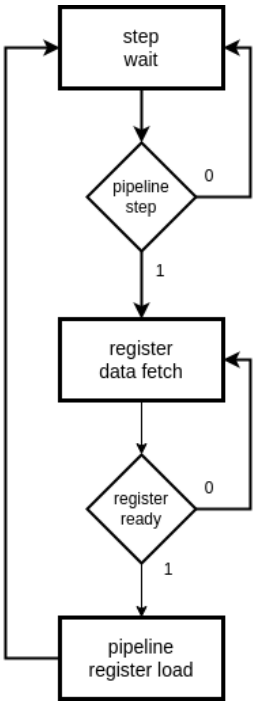

Figura 30: funzionamento della macchina a stati dello stadio decode

Nelle porte sono presenti anche *new program counter in* e *new program counter out* che rappresentano semplicemente una propagazione che avviene dagli ingressi alle uscite per portare il valore di PC allo

stadio successivo. Nella rappresentazione della macchina a stati in figura 30 è possibile notare tre diversi stati:

- *step wait*, in cui la macchina abilita il segnale *decode stage ready* per segnalare di essere pronta per un nuovo step. Alla ricezione di uno step, la macchina provvede a richiedere i dati al register file manager, nelle modalità descritte precedentemente, entrando nello stato *register data fetch*.
- *register data fetch*, in cui la macchina disabilita il segnale *decode stage ready* e attende il completamento della lettura nei registri. Una volta completata la lettura, la macchina a stati enterà in *output latch*.
- *output latch*, in cui la macchina a stati inserisce nei registri di output i dati raccolti al *decode* e dal *register file manager*. Subito dopo passerà nello stato di *step wait*, in attesa di una nuova richiesta.

La macchina a stati integra anche il segnale di *output mask*, utilizzato dal *pipeline manager* per inizializzare i registri di output e resettare la maccina a stati. Il funzionamento è analogo a quello dello stadio di fetch, con la differenza che i dati inizializzati sono differenti (*instruction type* viene impostato a 0x0111 *alu operation* a 0x00000 in modo da eseguire una NOP). Nella simulazione mostrata in

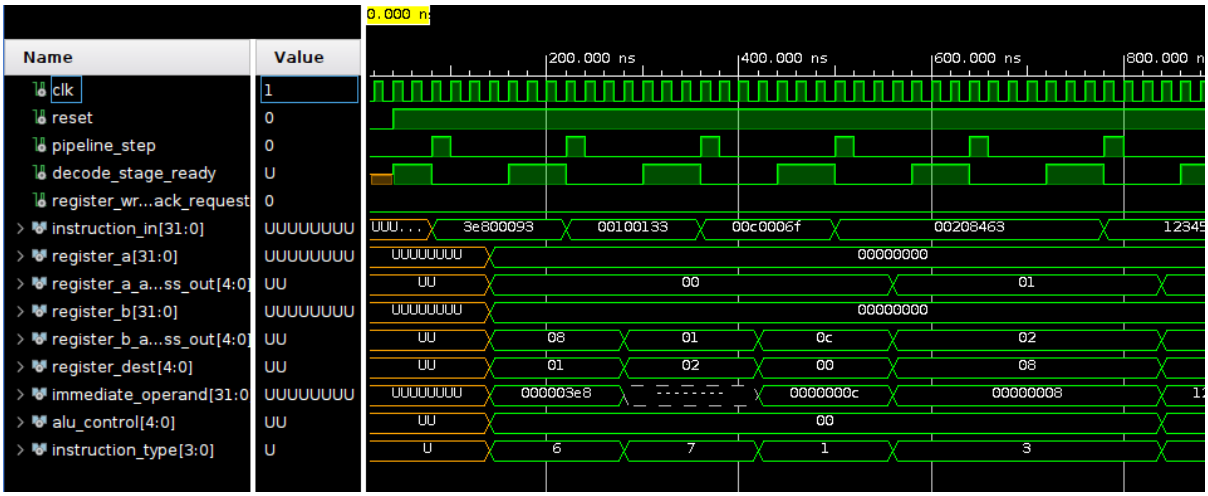

Figura 31: simulazione della macchina a stati dello stadio decode

figura 31 viene mostrato il funzionamento della macchina a stati secondo i vari tipi di istruzioni. Il funzionamento corrento del *register file manager* verrà però testato successivamente durante un'esame completo del risultato finale.

3.3 Execute stage

Lo stadio di esecuzione ha il compito di eseguire effettivamente le istruzioni, che sono state ricevute e decodificate dagli stadi precedenti. Il suo funzionamento si basa principalmente sull'utilizzo di una ALU, che viene predisposta per eseguire operazioni di tipo aritmetico-logico.

ALU

La ALU è un componente capace di eseguire operazioni aritmetiche come somme o sottrazioni e, nei casi più complessi, anche moltiplicazioni e divisioni. Nel nostro caso la alu ha due ingressi da 32bit e deve saper eseguire:

- Sottrazioni
- Somme
- Shift logici e aritmetici verso destra

- Shift logici verso sinistra
- Confronto tra due numeri (con o senza segno)
- AND, OR e XOR

Le operazioni da eseguire non sempre possono coprire un soli ciclo di clock. Per eseguire infatti uno shift logico non predeterminato è necessario utilizzare infatti uno shift register e un contatore, dato che non è fattibile con solo logica combinatoria.
La ALU in questione è quindi un componente sequenziale dotato dei seguenti segnali:

- *execute*, una linea in ingresso utilizzata per ricevere richieste da parte dello stadio *execute*.
- *ready*, una linea in uscita utilizzata per segnalare che la ALU è in attesa di nuove richieste.
- *A e B*, ovvero i due ingressi per i due operandi della ALU.
- *data out*, ovvero l'uscita della ALU.

Per descrivere in modo semplice la sua architettura viene utilizzato il codice VHDL.

```
if (reset='0') then
current_state <= execute_wait;
ready<='0';

-- nello stato execute wait la ALU imposta ready a 1 se execute = 0
-- in questo a situazione attende ancora la richiesta

-- nel caso in cui execute=1, la macchina imposta ready=0 e procede nel prossimo
-- stato (a seconda dell'operazione da eseguire).

elsif (current_state = execute_wait) then
    if (execute='1') then
        ready<='0';

-- se l'operazione richiede uno shift la macchina passa allo stato shift ed esegue
-- ad ogni ciclo l'operazione richiesta.

-- viene utilizzata la variabile tshift per memorizzare in che direzione lo schift
-- deve essere eseguito.
        if (operation = "00001" or operation="00101" or operation="01101") then
            if (operation = "00001") then
                if (b_signal >32) then
                    result <= (others => '0');
                else

-- caso shift a sinistra

                    tshift:= left;
                    shift_number:=a_signal;
                    shift_count:=b_signal(4 downto 0);
                    current_state <= shift;
                end if;
            elsif (operation = "00101") then
                if (b_signal >32) then
                    result <= (others => '0');
                else

-- caso shift logico a destra

                    tshift:= uright;
                    shift_number:=a_signal;
                    shift_count:=b_signal(4 downto 0);
                    current_state <= shift;
                end if;
            else
                if (b_signal >32) then
                    result <= (others => b_signal(31));
                else
```

```
-- caso shift aritmetico a destra

                    tshift:= aright;
                    shift_number:=a_signal;
                    shift_count:=b_signal(4 downto 0);
                    current_state <= shift;
                end if;
            end if;

        else

-- se l'operazione non e' uno shift la macchina entra nello stato other

            current_state <= other;
        end if;
    else
        ready<='1';
    end if;
elsif (current_state = shift) then
    ready<='0';
    if (shift_count=0) then
        result <= shift_number;
        current_state <= execute_wait;
    else
        case tshift is
            when left =>
                shift_number := shift_left(shift_number,1);
            when uright =>
                shift_number := shift_right(shift_number,1);
            when aright =>
                shift_number := shift_right(shift_number,1);
                shift_number(31):=shift_number(30);
        end case;
        shift_count:=shift_count-1;
    end if;
elsif (current_state = other) then
    ready<='0';
    case operation is
        when "00000" =>
            result <= (a_signal + b_signal);          --sum
        when "01000" =>
            result <= (a_signal - b_signal);          --sub
        when "00011"=>
            if (a_signal<b_signal) then               -- unsigned lower than
                result <= (others => '1');
            else
                result <= (others => '0');
            end if;
        when "00010" =>                               -- signed lower than
            if (signed(a_signal)<signed(b_signal)) then
                result <= (others => '1');
            else
                result <= (others => '0');
            end if;
        when "00110" =>                               -- or
            result <= (a_signal or b_signal);
        when "00111" =>                               -- and
            result <= (a_signal and b_signal);
        when "00100" =>                               -- xor
            result <= (a_signal xor b_signal);
        when others =>
            result <= (others => '-');
    end case;

-- alla fine dell'operazione la alu torna nello stato execute wait

    current_state <= execute_wait;
end if;
```

top level

Nello stadio di *access* la ALU provvede ad eseguire tutte le operazioni aritmetiche necessarie. Tuttavia in ingresso allo stadio sono presenti diversi possibili operandi tra cui:

- Il registro A
- Il registro A
- l'operando immediato
- Il program counter

Il compito del top level è quello, oltre di reindirizzare i dati correttamente alla ALU, quello di predisporre delle linee in uscita per l'esecuzione del prossimo stadio.

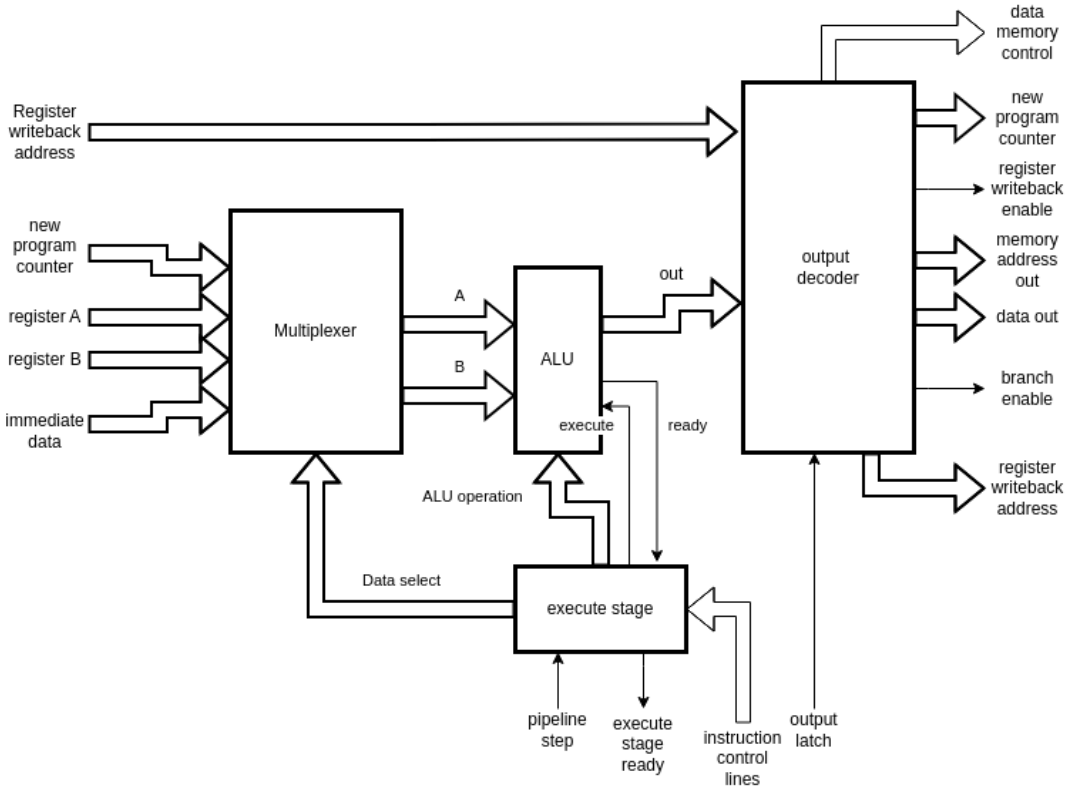

Figura 32: Struttura dello stadio execute

La figura 32 mostra la struttura interna dello stadio *execute*. Tra i segnali possiamo trovare:

- *new program counter*, che prende in ingresso il valore di PC propagato dall stadio *decode*.
- *register A e register B*, che vengono utilizzati per ricevere gli operandi dal register file.
- *immediate operand*, il dato immediato ricavato dall stadio di *decode*.
- *new program counter out*, utilizzato per inviare allo stadio *access* un eventuale nuovo valore PC da registrare.
- *memory address out*, utilizzato per specificare l'indirizzo di un eventuale lettura o scrittura in memoria dati.
- *data out*, che viene utilizzato per inviare il risultato di un'operazione aritmetica o un dato da scrivere in memoria dati.

- *branch enable*, utilizzato per indicare allo stadio *access* di eseguire un'operazione di salto.
- *data memory direction e data memory enable*, utilizzati per indicare se lo stadio *access* deve utilizzare la memoria dati.
- *alu operation e instruction type*, gli ingressi generati dallo stadio *decode* per indicare le operazioni da eseguire.

Inoltre, sono presenti i segnali *register writeback address e register writeback enable*, utilizzati per indicare un'eventuale scrittura nei registri allo stadio *access*. Mentre il primo viene direttamente propagato, il secondo viene invece calcolato in base all'operazione che lo stadio deve eseguire. Sono presenti anche i segnali relativi alla gestione della pipeline:

- *pipeline step*, usato per segnalare allo stadio un nuovo step di pipeline
- *esecute stage ready*, usato dalla macchina a stati per indicare che lo stadio è pronto per eseguire un nuovo step.
- *output mask*, usato per inizializzare la macchina a stati e il registo di output con i valori di NOP corretti.

La macchina a stati dello stadio *execute* deve eseguire operazioni diverse in base ai valori delle linee *alu operation e instruction type*. in particolare:

- caso JALR
  La macchina a stati deve mandare in ingresso alla ALU prima il registro e l'operando immediato per calcolare l'offset rispetto al PC e poi eseguire la somma con il valore in ingresso del program counter (2 passaggi). Una volta eseguite queste due operazioni, in output deve abilitare *branch enable*, abilitare *writeback enable* e inviare il nuovo valore di PC in *new program counter out*.
- caso istruzioni B
  la macchina a stati deve prima inviare alla ALU i valori dei due registri A e B, eseguire un confronto per capire se eseguire il salto e poi calcolare l'indirizzo PC, inviando nuovamente alla ALU l'offset e il program counter corrente. Se il confronto avviene a buon fine allora la macchina abilita *branch enable* e procede ad inviare il nuovo valore PC. (2 passaggi)
- caso JAL
  In queso caso la macchina a stati deve semplicemente calcolare il nuovo valore PC tramite la ALU inserendo il valore mmediato e il PC corrente e poi procedere ad abilitare *branch enable e register writeback*.
- caso operazioni I e R aritmetiche
  In questi casi la alu deve calcolare il risultato utilizzando due registri o un registro e un immediato come operandi. Il risultato viene posto in uscita e viene abilitato *writeback enable* per salvare il risultato in un registro.
- caso istruzioni S
  In questo caso la ALU deve calcolare l'indirizzo di memoria dove prelevare i dati (sommando il valore di un registo all'immediato) e procedere poi ad abilitare *memory enable e memory write enable* in modo da richiedere un accesso in scrittura. Inoltre, il dato del registro che deve essere scritto viene indirizzato direttamente in output.
- caso istruzioni I load
  In questo caso la ALU deve eseguire il calcolo dell'indirizzo di memoria come nel caso delle istruzioni S e abilitare *memory enable* in modo da richiedere una lettura nella memoria dati. Inoltre deve abilitare *regiter writeback* in modo da far scrivere il dato nel registro.

La figura 33 mostra una rappresentazione semplificata di come è strutturata la macchina a stati che controlla la ALU. Nello schema non vengono infatti visualizzati tutti i passaggi di attesa necessari alla ALU per calcolare il risultato.

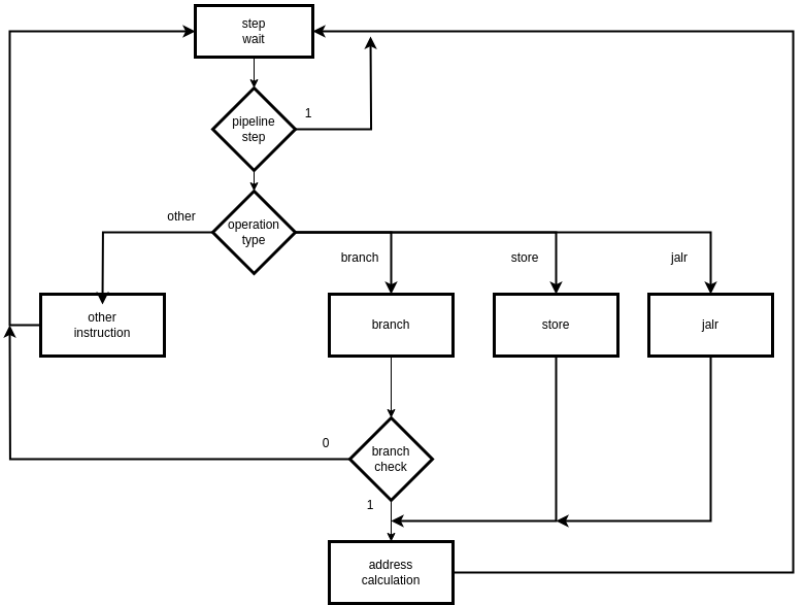

Figura 33: Struttura dell a macchina a stati dello stadio execute

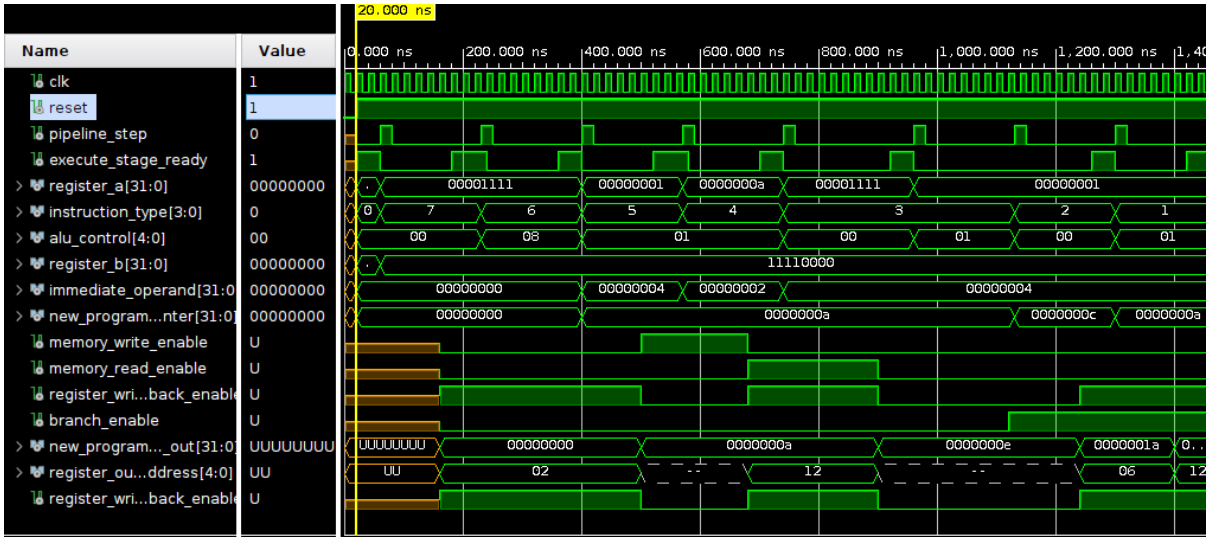

Figura 34: Simulazione della macchina a stati per alcune operazioni

3.4 Access stage

Lo stadio *access* ha il compito di completare l'esecuzione delle operazioni iniziate dallo stadio *execute*, ovvero tutte le operazioni di scrittura e lettura in memoria, scrittura nei registri e scrittura del program counter nel caso di salti.
L'implementazione di questo stadio è quindi molto semplice, dato che nella maggiorparte dei casi deve solo propagare dei segnali. Nella figura 35 viene rappresentata la struttura dello stadio *access*. tra i segnali di ingresso utilizzati dalla macchina a stati sono presenti:

- *alu output*, l'ingresso utilizzato per ricevere dati destinati ai registri o alla mmeoria.
- *data memory address*, l'ingresso che contiene l'indirizzo di memoria dove leggere o scrivere i dati.
- *data memory read* e *data memory write*, gli ingressi utilizzati per ricevere richieste di accesso alla memoria.

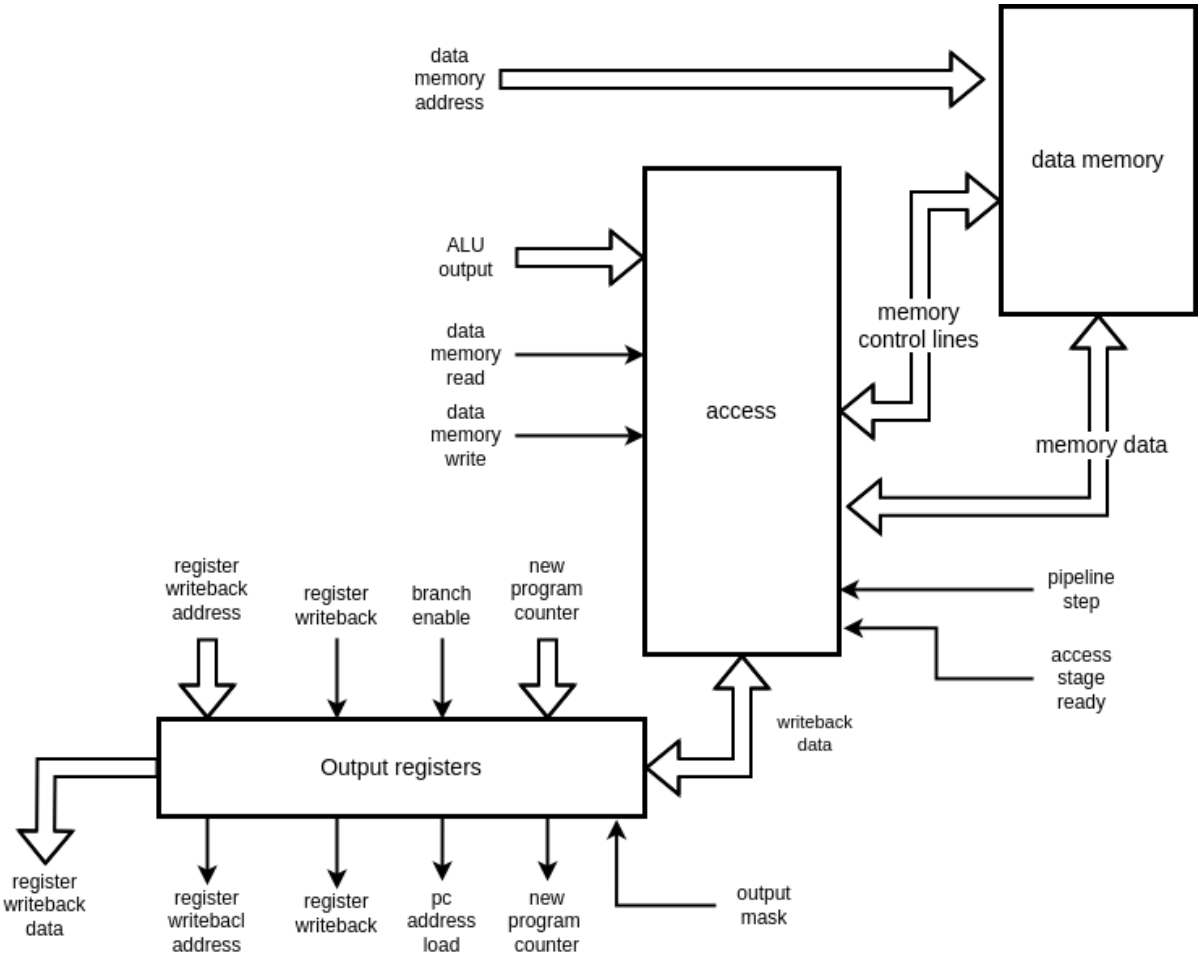

Figura 35: Struttura dello stadio access

- *registr writeback data*, la linea output utilizzata per inviare i dati ai registri.

Gli altri ingressi e le altre uscite vengono invece semplicemente propagati tramite il registro di output. In particolare:

- *branch enable* viene propagato in *pc address load*
- *new program counter* viene propagato in *new progra counter out*
- *register writeback enable* viene propagato in *regitser writeback*
- *register writeback address* viene propagato in *register writeback* address

Sono presenti anche segnali per la gestione di pieline analogi a quelli degli stadi precedenti:

- *pipeline step* è un ingresso utilizzato per stabilire quando procedere con la pipeline
- *access stage ready* è un uscita utilizzata per segnalare che la lo stadio è pronto per l'esecuzione di un nuovo step
- *output mask* è un ingresso utilizzato dal *pipeline manager* per inizializzare la macchina a stati e i registri di output con valori che rispettano l'istruzione NOP.

La macchina a stati dello stadio si trova quindi a gestire interamente la parte relativa all'accesso in memoria che, come si può notare nella figura 36, è molto simile a quella dello stadio *fetch*. Dalla macchina a stati è possibile notare i seguenti stati:

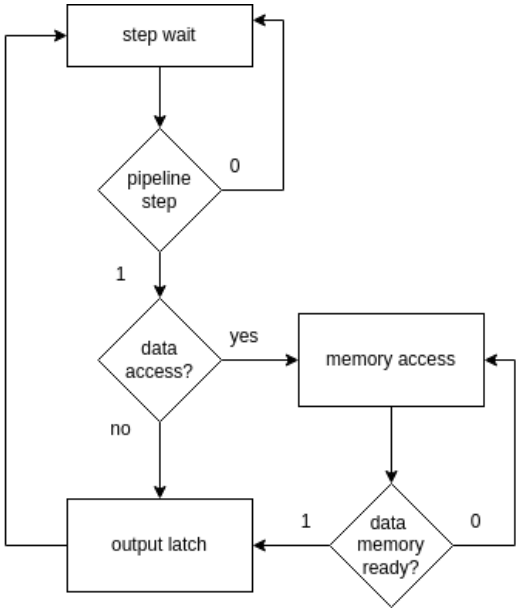

Figura 36: Struttura della macchina a stati dello stadio access

- *step wait*
  In questo stato la macchina attende uno step dal *pipeline manager* e mantiene abilitato *access stage ready* fino a quando arriva una richiesta. Nel momento in cui la richiesta arriva, la macchina a stati disabilita *access stage ready* e, se è necessario un accesso in memoria, procede allo stadio *memory access* facendo nello stesso momento una richesta alla memoria (in caso contrario a *output latch*).

- *memory access*
  Durante questo stato la macchina attende che la meoria dati fornisca il risultato o scriva il dato ricevuto. Quando la transazione viene completata, la macchina passa nello stato *output latch*.

- *output latch*
  In questo stato la macchina provvede a reindirzzare tutti i dati nel modo corretto. Se è stato necessario un accesso in memoria in lettura la macchina reindirizza il dato ricevuto in *register writeback data*, altrimenti provvede a propagare il dato ricevuto in *alu output*.

I segnali di controllo relativi all'accesso nel registro PC vegnono propagati nel momento in cui la macchina a stati rileva uno step della pipeline, in modo da dare il tempo allo stadio di *fetch* di poter leggere l'istruzione successiva.

4 Gestione della pipeline

La struttura del core, come detto precedentemente, utilizza quattro stadi per eseguire la pipeline e, di conseguenza, avere un troughput di circa quattro volte maggiore rispetto alla sua versione normale. La velocità della pipeline dipende dallo stadio con latenza maggiore che, in condizioni normali (senza accessi in memoria dati), risulta essere lo stadio di *fetch*, dato che deve periodicamente accedere alla memoria istruzioni.

Durante la descrizione dell'achitettura di ogni stadio sono stati sempre introdotti tre collegamenti per la gestione della pipeline: uno per eseguire uno step, uno per segnalare l'attività e un altro per reimpostare lo stadio. Questi tre segnali vengono utilizzati dal *pipeline manager* proprio per la gestione del flusso dei processi. In particolare, il *pipeline manager* deve:

1. Inizializzare tutti gli stadi all'accensione del processore utilizzando gli ingressi *output mask* di ogni stadio

2. Attendere che tutti gli stadi siano pronti per l'esecuzione di uno step

3. Abilitare tutti e quattro i collegamenti *pipeline step* per far progredire il flusso di pipeline

4. Ripetere il secondo e il terzo punto in sequenza, fino a quando il processore non riceve un reset

Durante l'avanzamento del programma, tuttavia, potrebbero verificarsi delle condizioni che non permettono al processore di eseguire correttamente le istruzioni. Queste condizioni, dette *stall*, si verificano per due diversi motivi:

- *Read after write*
  Questo tipo di stall viene provocato a causa del fatto che nel programma è presente un'istruzione che utilizza come operando un registro che nell'istruzione precedente viene utilizzato come destinazione. Questa situazione si verifica perché nel momento in cui la seconda istruzione viene eseguita la prima deve ancora essere processata dallo stadio *access*, che ha proprio il compito di salvare i risultati nel *register file*.

- *Branch stall*
  questo tipo di stall si verifica ogni volta che il processore deve eseguire un salto in un altro indirizzo di memoria. In questo caso, è necessario prima far terminare l'esecuzione di tutte le istruzioni precedenti per evitare che l'esecuzione del programma venga tagliato in alcuni punti.

Il *pipeline* manager deve avere quindi una macchina a stati che deve essere in grado di rilevare eventuali situazioni di *stall* e procedere alla loro risoluzione. Per fare questo è necessario ottenere informazioni dagli tadi in esecuzione. In particolare:

- Dallo stadio *decode* è necessario ottenere gli indirizzi dei due registri che rappresentano gli operandi dell'istruzione e il tipo di istruzione decodificata (l'uscita *instruciton type*). Mentre gli indirizzi degli operandi serviranno per identificare situazioni di RAW, il tipo di operazione servirà per identificare tutti i tipi di *stall*.

- Dallo stadio *execute* è necessario ottenere lo stato del segnale *branch enable*, per verificare eventuali situazioni di salto.

- Dallo stadio *access* è necessario ottenere l'indirizzo del registro di destinazione, per identificare eventuali problemi di RAW.

Tramite tutte queste informazioni è quindi possibile risalire alla situazione corrente della pipeline.

Read after Write

Supponiamo che nella pipeline siano attualmente presenti quattro istruzioni diverse, tra cui R (istruzione di lettura) e W (istruzione di scrittura). La situazione di RAW si verifica quando si è nella situazione raffigurata nell'immagine 4. L'istruzione R deve quindi attendere che W entri nello stadio

fetch	decode	execute	access
X	R	W	Y

access e venga eseguito. Di conseguenza, il pipeline *pipeline manager* deve:

1. Far progredire lo stadio *decode*, *execute* e *access* per rendere effettive le modifiche dell'istruzione Y.

2. Far progredire lo stadio *decode*, e *access* per rendere effettive le modifiche dell'istruzione W.

3. Reimpostare lo stadio di *decode*, *execute* e *access* in modo da evitare di ripetere altre tre volte la stessa istruzione R nei successivi step.

4. Procedere con la normale esecuzione del programma.

In questo caso è necessario eseguire più volte anche lo stadio *decode*, dato che è quello che ha il compito di gestire il *register file*.
Per accorgersi di una situazione RAW imminente, il *pipeline manager* verifica se gli indirizzi degli operandi (uno o due in base alle istruzioni) di *decode* sono uguali a quello di destinazione in *access* e se *write enable* è abilitato. La figura 4 mostra la situazione interna della pipeline ad ogni passaggio dell'algoritmo di stall.

	fetch	decode	execute	access
0	X	R	W	Y
1	X	R	R	W
2	X	R	R	R
3	X	NOP	NOP	R

Jump stall

Consideriamo come *jump stall* la situazione provocata da un salto incondizionato, che deve quindi essere eseguito in ogni caso.
Supponiamo in questo caso che all'interno della pipeline ci sia un istruzione J che provoca un salto incondizionato. In questo caso il *pipeline manager* deve:

fetch	decode	execute	access
X	J	Y	Z

1. Far progredire lo stadio *decode*, *execute* e *access* per rendere effettive le modifiche dell'istruzione Y.
2. Far progredire lo stadio *decode*, e *access* per rendere effettive le modifiche dell'istruzione Z e carcare l'istruzione J.
3. Reimpostare lo stadio di *decode*, *execute* e *access* in modo da evitare di ripetere altre tre volte la stessa istruzione J nei successivi step.
4. Procedere con la normale esecuzione del programma.

In questo caso non è necessario eseguire un altro step di pipeline, dato che il salto viene effettivamente fatto durante l'esecuzione pel punto 2 quando *access* riceve le direttive. nella figura 4 viene rappresentato lo stato della pipeline durante l'esecuzione dell'algoritmo. Una situazione di *jmp stall* viene

	fetch	decode	execute	access
0	X	J	Y	Z
1	X	J	J	Y
2	X	J	J	J
3	J	NOP	NOP	NOP

identificata utilizzando il campo *instruction type* dello stadio *decode*.

Branch stall

Consideriamo come *branch stall* la situazione che si verifica quando il programma deve fare un branch condizionato. In questo caso è possibile quindi che il salto non debba essere fatto.
Suppoiamo che all'interno della pipelina ci sia un istruzione B che provoca un salto condizionato. In questo caso il *pipeline manager* deve:

fetch	decode	execute	access
X	B	Y	Z
X	J	J	Y
X	J	J	J
J	NOP	NOP	NOP

1. Far progredire lo stadio *decode*, *execute* e *access* per rendere effettive le modifiche dell'istruzione Z.
2. Reimpostare lo stadio *execute* in modo da non provocare un salto non desiderato nello step successivo
3. Far progredire lo stadio *decode*, *execute* e *access* per rendere effettive le modifiche dell'istruzione Y.
4. Far progedire lo stadio *execute*, in modo da capire le il salto deve essere eseguito (tramite l'ingresso *branch enable*). In caso negativo, gli stadi *decode*, *execute* e *access* Verranno reimostati e si portà procedere direttamente con la normale esecuzione della pipeline, altrimenti è necessario continuare con il prossimo step.
5. Far progredire lo stadio *access* in modo da rendere effettivo il salto.
6. Procedere con la normale esecuzione della pipeline.

Nella figura 4 viene rappresentato lo stato della pipeline durante l'esecuzione dell'algoritmo (il punto 1 e 2 vengono eseguiti insieme). Una situazione di *branch stall* viene identificata utilizzando il campo

	fetch	decode	execute	access
0	X	B	Y	Z
1	X	B	NOP	Y
2	X	B	B	NOP
3	X	NOP	NOP	B
4	B	NOP	NOP	NOP

instruction type dello stadio *decode*.

Diagramma di flusso

Il diagramma di flusso descrive quindi il funzionamento complessivo del *pipeline manager*. Lo stato *initialize* viene utilizzato una sola volta subito dopo l'operazione di reset, mentre gli *stall* comprendono più di uno stato della macchina ma vengono rappresentati come blocchi unici per semplicità.

5 Testbench finale

Il testbench finale serve per verificare il completo funzionamento del processore in tutti i suoti aspetti. Durante questa fase viene utilizzato un programma già codificato in binario, che verrà poi posizionato all'interno di una memoria di tipo AXI in modo da poter verificare tutte le transizioni dei segnali.

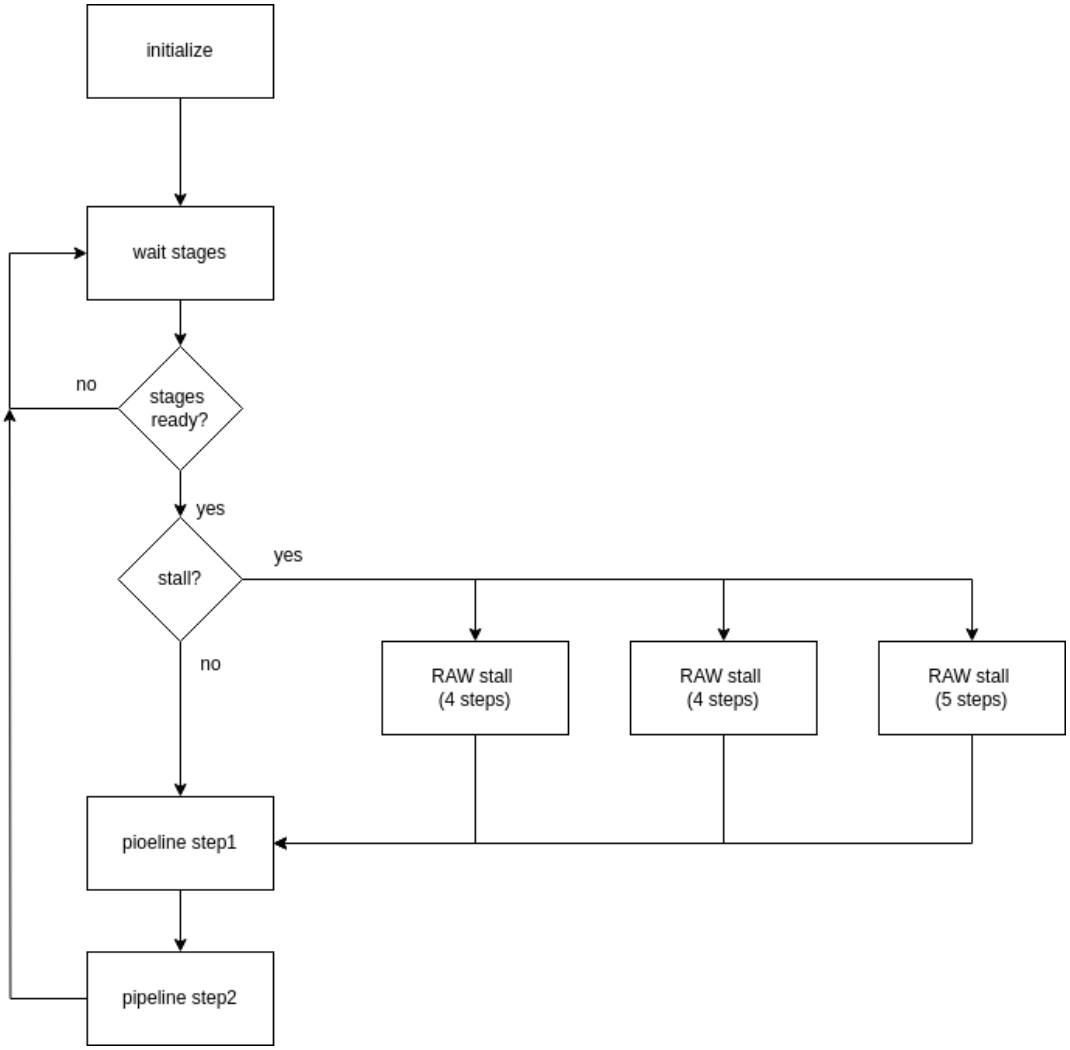

Figura 37: Diagramma di flusso semplificato del pipeline manager

Il programma scelto procede a fare una moltiplicazione tra due numero (0x0a e 0x04 in questo caso).

```
0000000000000000 <_boot>:
   0:   00440413                addi    s0,s0,4
   4:   00a48493                addi    s1,s1,10

0000000000000008 :
   8:   00048513                mv      a0,s1
   c:   00a484b3                add     s1,s1,a0        #read after write
  10:   fff40413                addi    s0,s0,-1
  14:   fe041ae3                bnez    s0,8          #branch condizionato

0000000000000018 :
  18:   000000b3                sw      s1,0(x0)
```

La prima parte del testbench mostra come il *pipeline manager* reagisce alle varie condizioni di branch e RAW (figura 38). Durante la fase di loop, infatti, si verificano periodicamente degli stall a causa dell'istruzione di salto condizionato presente in 0x00000014. Andando avanti il testbench rimane simile, fino a quando il loop finisce e il processore inizia a leggere istruzioni non valide (figura 38.

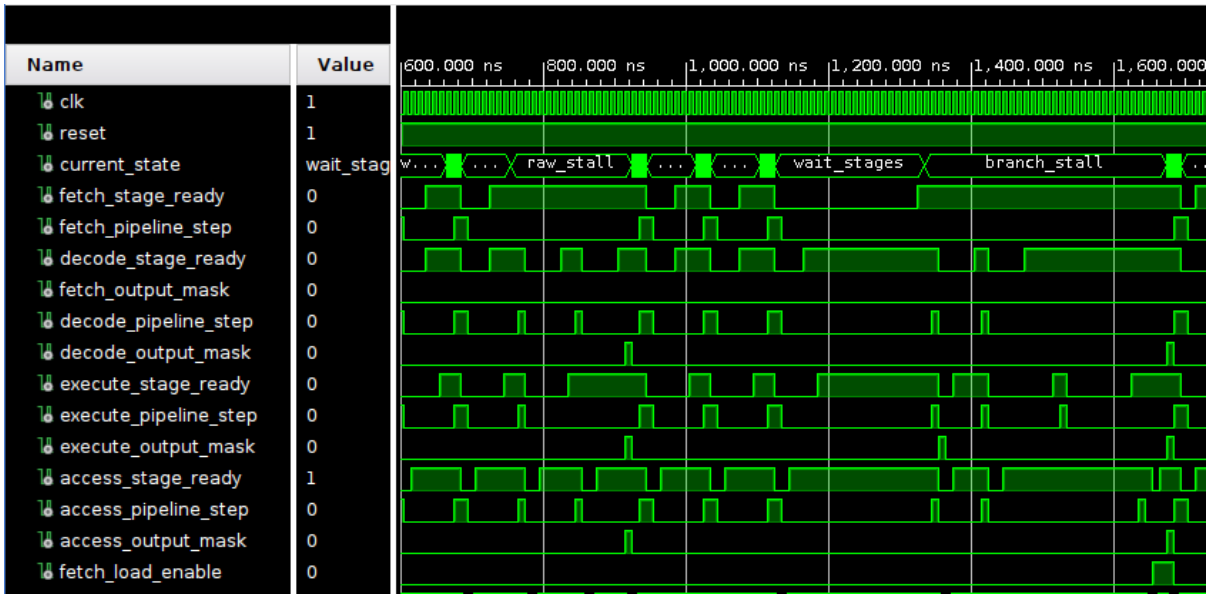

Figura 38: Testbench del loop della moltiplicazione

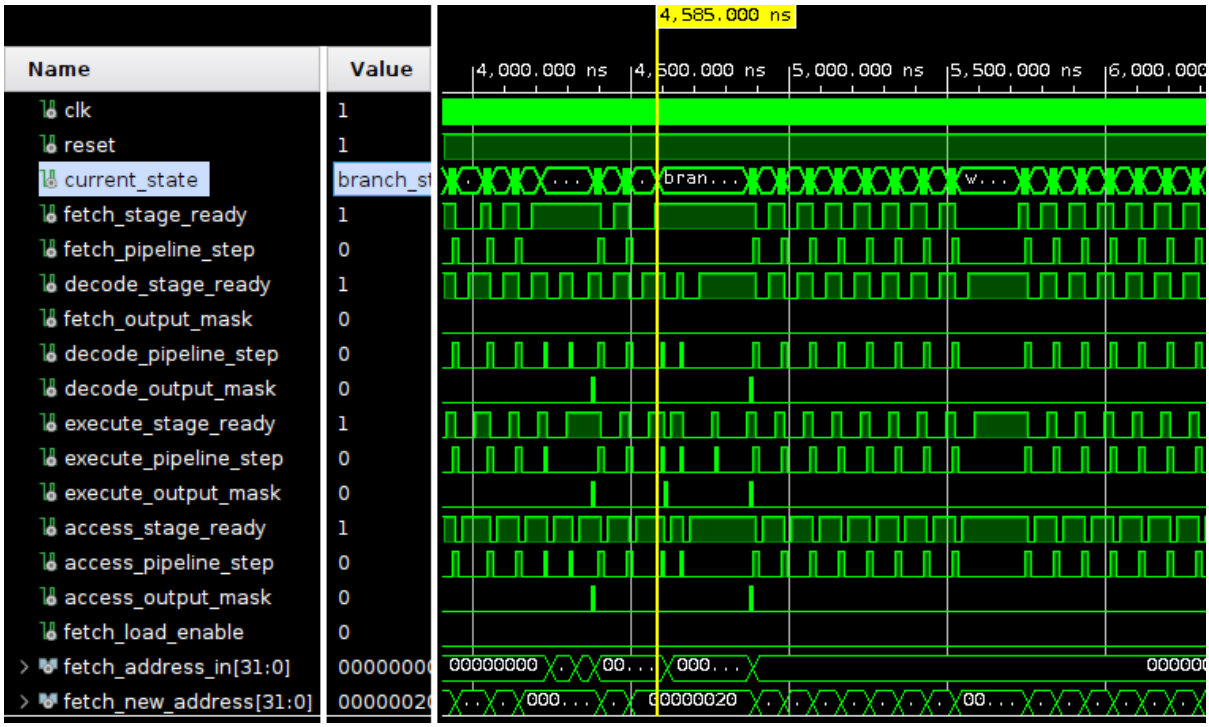

Figura 39: Testbench della fine del loop della moltiplicazione