%Come visto nel Capitolo~\ref{cap:capitolo2}, il protocollo di comunicazione tra i moduli in Cellsim prevede quattro fasi:

%\begin{itemize}

% \item[$\circ$] \textbf{start\\_of\\_cycle}

% \item[$\circ$] \textbf{on\\_data}

% \item[$\circ$] \textbf{on\\_accept}

% \item[$\circ$] \textbf{end\\_of\\_cycle}

%\end{itemize}

%La prima, \emph{start\\_of\\_cycle}, viene utilizzata per mandare in uscita i dati relativi alla precedente fase di elaborazione; le due fasi intermedie, \emph{on\\_data} e \emph{on\\_accept}, sono utilizzate per la sincronizzazione; infine l'ultima fase, \emph{end\\_of\\_cycle}, è quella in cui tutti i segnali sono stabili e possono essere elaborati.\\

%$\times$pipes, d'altro canto, campiona i segnali sul fronte positivo del clock. Questi devono essere stabili per un intervallo di tempo precedente al fronte, detto \emph{tempo di setup} e per un intervallo successivo, detto \emph{tempo di hold}.\\

%Prima di proseguire è necessario fare una precisazione; la network on chip prevede un doppio dominio di clock, uno per gli elementi interni alla rete e per il back-end delle network interface, \emph{$\times$pipes\\_clock} e uno per il front-end delle network interface, \emph{OCP\\_clock}, identico a quello degli IP core ad esse connessi. Cellsim non supporta domini multipli di clock, per tale motivo è stato utilizzato, anche per la network on chip, un solo dominio prelevando il clock direttamente da quello del k-bus e ponendo a uno il divisore del clock, \emph{Clock\\_div}, di $\times$pipes.\\

%Poiché i memory access sono considerati stabili sul fronte negativo del clock, le conversioni di questi in transazioni OCP per la network interface possono essere effettuate solamente nella fase di \emph{end\\_of\\_cycle}. Nell'implementazione del wrapper, perciò, la creazione dei comandi OCP relativi ai memory access in arrivo dagli IP core è stata realizzata tramite apposite funzioni che vengono richiamate all'interno della funzione \emph{route()} del k-bus, opportunamente modificata.

\subsubsection{Integrazione di xpipes\\_interconnect all'interno di k-bus}

L'inserimento di \emph{xpipes\\_interconnect} all'interno del modulo del k-bus prevede:

\begin{itemize}

\item[$\circ$] l'inclusione dell'header file \emph{xpipes\\_interconnect.h};

\begin{lstlisting}[caption=Inclusione del file di header xpipes\\_interconnect.h contenente la NoC, label=lst:header-file-noc]

#include "unisim/unisim.h"

#include "types.h"

#include "math.h"

/\* include la network on chip \*/

#include "/home/gcongiu/NetworkOnChip\_my\_advanced/xpipes\_interconnect.h"

using namespace std;

...

\end{lstlisting}

\item[$\circ$] la creazione dell'istanza della network on chip;

\begin{lstlisting}[caption=Creazione dell'istanza della NoC, label=lst:noc-instance]

...

/\* Istanza della NoC \*/

xpipes\_interconnect \*xpipes\_ic;

...

EIB(char \*name) : module(name)

{

...

xpipes\_ic = new xpipes\_interconnect("xpipes\_IC",/\* N\_MASTERS\*/ 0,

/\*N\_SLAVES\*/ PORT\_COUNT, tf, tracing);

...

}

...

\end{lstlisting}

\item[$\circ$] la definizione dei segnali corrispondenti agli ingressi OCP delle network interface;

\begin{lstlisting}[caption=Definizione dei segnali SystemC per la NoC, label=lst:noc-signals]

...

/\* Segnali OCP da mandare alla Network Interface slave \*/

sc\_signal <bool> OCP\_Clock;

sc\_signal <bool> xpipes\_Clock;

sc\_signal <sc\_uint<COUNTERWD> > Clock\_div;

sc\_signal <bool> rst;

/\* OCP Request Signals \*/

sc\_signal <sc\_uint<MCMDWD> > MCmd[PORT\_COUNT];

sc\_signal <sc\_uint<MATOMICLENGTHWD> > MAtomicLength[PORT\_COUNT];

sc\_signal <sc\_uint<MBURSTLENGTHWD> > MBurstLength[PORT\_COUNT];

sc\_signal <bool> MBurstPrecise[PORT\_COUNT];

sc\_signal <sc\_uint<MBURSTSEQWD> > MBurstSeq[PORT\_COUNT];

sc\_signal <bool> MBurstSingleReq[PORT\_COUNT];

sc\_signal <bool> MDataLast[PORT\_COUNT];

sc\_signal <bool> MReqLast[PORT\_COUNT];

sc\_signal <sc\_uint<MADDRWD> > MAddr[PORT\_COUNT];

sc\_signal <sc\_uint<SOURCEWD> > MSource[PORT\_COUNT];

sc\_signal <sc\_uint<MADDRSPACEWD> > MAddrSpace[PORT\_COUNT];

sc\_signal <sc\_uint<MDATAWD> > MData[PORT\_COUNT];

sc\_signal <bool> MDataValid[PORT\_COUNT];

sc\_signal <sc\_uint<MBYTEENWD> > MByteEn[PORT\_COUNT];

sc\_signal <bool> SCmdAccept[PORT\_COUNT];

sc\_signal <bool> SDataAccept[PORT\_COUNT];

/\* OCP Receive Signals \*/

sc\_signal <sc\_uint<MCMDWD> > MCmd\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MATOMICLENGTHWD> > MAtomicLength\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MBURSTLENGTHWD> > MBurstLength\_recv[PORT\_COUNT];

sc\_signal <bool> MBurstPrecise\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MBURSTSEQWD> > MBurstSeq\_recv[PORT\_COUNT];

sc\_signal <bool> MBurstSingleReq\_recv[PORT\_COUNT];

sc\_signal <bool> MDataLast\_recv[PORT\_COUNT];

sc\_signal <bool> MReqLast\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MADDRWD> > MAddr\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<SOURCEWD> > MSource\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MADDRSPACEWD> > MAddrSpace\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MDATAWD> > MData\_recv[PORT\_COUNT];

sc\_signal <bool> MDataValid\_recv[PORT\_COUNT];

sc\_signal <sc\_uint<MBYTEENWD> > MByteEn\_recv[PORT\_COUNT];

sc\_signal <bool> SCmdAccept\_recv[PORT\_COUNT];

sc\_signal <bool> SDataAccept\_recv[PORT\_COUNT];

...

\end{lstlisting}

\item[$\circ$] il binding dei segnali locali, definiti nel k-bus, con le porte di \emph{xpipes\\_interconnect.h}.

\begin{lstlisting}[caption=Binding delle porte, label=lst:ports-binding]

...

for(int i = 0; i < PORT\_COUNT; i++)

{

...

/\* NI target binding \*/

xpipes\_ic->target\_clock[i](OCP\_Clock);

xpipes\_ic->target\_div[i](Clock\_div);

xpipes\_ic->MCmd\_in[i](MCmd[i]);

xpipes\_ic->MAtomicLength\_in[i](MAtomicLength[i]);

xpipes\_ic->MDataLast\_in[i](MDataLast[i]);

xpipes\_ic->MAddr\_in[i](MAddr[i]);

xpipes\_ic->MSource\_in[i](MSource[i]);

xpipes\_ic->MAddrSpace\_in[i](MAddrSpace[i]);

xpipes\_ic->MData\_in[i](MData[i]);

xpipes\_ic->MByteEn\_in[i](MByteEn[i]);

xpipes\_ic->MDataValid\_in[i](MDataValid[i]);

xpipes\_ic->MBurstLength\_in[i](MBurstLength[i]);

xpipes\_ic->MBurstSeq\_in[i](MBurstSeq[i]);

xpipes\_ic->MBurstPrecise\_in[i](MBurstPrecise[i]);

xpipes\_ic->MReqLast\_in[i](MReqLast[i]);

xpipes\_ic->MBurstSingleReq\_in[i](MBurstSingleReq[i]);

xpipes\_ic->SCmdAccept\_out[i](SCmdAccept[i]);

xpipes\_ic->SDataAccept\_out[i](SDataAccept[i]);

xpipes\_ic->MCmd\_recv\_out[i](MCmd\_recv[i]);

xpipes\_ic->MAtomicLength\_recv\_out[i](MAtomicLength\_recv[i]);

xpipes\_ic->MDataLast\_recv\_out[i](MDataLast\_recv[i]);

xpipes\_ic->MAddr\_recv\_out[i](MAddr\_recv[i]);

xpipes\_ic->MSource\_recv\_out[i](MSource\_recv[i]);

xpipes\_ic->MAddrSpace\_recv\_out[i](MAddrSpace\_recv[i]);

xpipes\_ic->MData\_recv\_out[i](MData\_recv[i]);

xpipes\_ic->MByteEn\_recv\_out[i](MByteEn\_recv[i]);

xpipes\_ic->MDataValid\_recv\_out[i](MDataValid\_recv[i]);

xpipes\_ic->MBurstLength\_recv\_out[i](MBurstLength\_recv[i]);

xpipes\_ic->MBurstSeq\_recv\_out[i](MBurstSeq\_recv[i]);

xpipes\_ic->MBurstPrecise\_recv\_out[i](MBurstPrecise\_recv[i]);

xpipes\_ic->MReqLast\_recv\_out[i](MReqLast\_recv[i]);

xpipes\_ic->MBurstSingleReq\_recv\_out[i](MBurstSingleReq\_recv[i]);

xpipes\_ic->SCmdAccept\_recv\_in[i](SCmdAccept\_recv[i]);

xpipes\_ic->SDataAccept\_recv\_in[i](SDataAccept\_recv[i]);

}

...

\end{lstlisting}

\end{itemize}

\subsection{Gestione delle transazioni}

Vengono di seguito elencate e descritte le funzioni che sono state sviluppate per la conversione dei memory access in comandi OCP da propagare alle network interface di $\times$pipes. Queste fanno uso di variabili globali per la gestione delle richieste di store, le quali richiedono diversi cicli per il completamento.

\subsubsection{convertFromMemoryAccessStToOcp}

La funzione viene eseguita durante l'\emph{end\\_of\\_cycle} all'interno della \emph{route()} del k-bus; essa prende in ingresso un memory access di tipo store e lo traduce in una richiesta di write da inviare alla network interface corrispondente. L'implementazione fa uso dei seguenti vettori globali:

\begin{itemize}

\item[$\circ$] \emph{wordToTransfer[PORT\\_COUNT]}. Rappresenta il numero di word, di dimensione \emph{MDATAWD}, che devono essere accettate dalla network interface per completare la transazione. Questo numero dipende dalla dimensione del dato contenuto nel memory access (indicata dal campo \emph{\\_size}). %La network interface è capace di campionare, ad ogni ciclo, dei dati in ingresso che abbiano una dimensione non superiore a \emph{MDATAWD}. Nel caso in cui il dato da trasferire sia maggiore di tale valore è necessario utilizzare una write di tipo burst, come in effetti è stato fatto nell'implementazione del wrapper. Perciò a partire dalla dimensione del memory access in ingresso bisogna calcolare il numero di cicli richiesti per il suo trasferimento; questi vengono memorizzati all'interno di \emph{wordToTransfer}, che è un vettore con numero di elementi pari al numero di IP core da interconnettere.

In Figura~\ref{fig:transaction-manage} è possibile vedere come, a partire da un dato di dimensione pari a \emph{size}, il wrapper componga delle singole richieste di dimensione \emph{MDATAWD} da inviare alla NI tramite i segnali OCP.

\begin{figure}[!htbp]

\centering

\includegraphics[scale = 0.7]{figure/transaction-manage}

\caption{Gestione delle transazioni store, aventi dimensione pari a \emph{size}, da parte del wrapper}

\label{fig:transaction-manage}

\end{figure}

\item[$\circ$] \emph{semaphore[PORT\\_COUNT]}. E' utilizzata come semaforo per segnalare che la network interface è attualmente impegnata nell'elaborazione di una transazione. Il semaforo può assumere i valori: 0, 1 e 2, che rappresentano rispettivamente lo stato di idle, elaborazione di una store ed elaborazione di una load. Questo è utile nel caso in cui il wrapper stia inviando una store che richiede diversi cicli di clock per il trasferimento; in tal caso, se una load viene caricata nel buffer di ingresso, deve aspettare che il semaforo torni a zero (idle) prima di poter accedere alla rete. In definitiva, tramite \emph{semaphore} è possibile memorizzare lo stato di ogni NI, nel corso della simulazione, evitando sovrapposizione di richieste e garantendo l'uso esclusivo della risorsa da parte di ogni transazione.

\item[$\circ$] \emph{byteTxNumber[PORT\\_COUNT]}. Memorizza, il numero di byte attualmente accettati dalla network interface.

\end{itemize}

\begin{lstlisting}[caption=Funzione convertFromMemoryAccessStToOcp, label=lst:CFMASTO]

void convertFromMemoryAccessStToOcp(MemoryAccess \*access, int inportn)

{

int outportn = get\_port\_number(access);

...

semaphore[inportn] = 1;

if(inportn == 1 || inportn == 0 || outportn == 0 || outportn == 1)

wordToTransfer[inportn] = ceil((double)access->get\_size()/8);

else

wordToTransfer[inportn] = ceil((double)access->get\_size()/

((MDATAWD\*K)/8));

...

if(inportn == 0 || inportn == 1 || outportn == 0 || outportn == 1)

{

if(byteTxNumber[inportn] < wordToTransfer[inportn]\*8)

{

if(byteTxNumber[inportn] == 0)

{

...

MData[inportn].write(0);

//definire tutti gli altri segnali OCP

MBurstLength[inportn].write((sc\_uint<MBURSTLENGTHWD>)

(wordToTransfer[inportn]));

MBurstPrecise[inportn].write(true);

MBurstSeq[inportn].write(5); //trasferimento streaming

MBurstSingleReq[inportn].write(true);

MCmd[inportn].write(1);

MDataValid[inportn].write(true);

MReqLast[inportn].write(true);

MAddr[inportn].write((sc\_uint<MADDRWD>)

(access->get\_target()));

uint32\_t maccess = (uint32\_t)access;

MSource[inportn].write((sc\_uint<SOURCEWD>)(maccess));

if(wordToTransfer[inportn] == 1)

{

MDataLast[inportn].write(true);

MByteEn[inportn].write((sc\_uint<MBYTEENWD>)

(access->get\_size()));

}

else

{

MDataLast[inportn].write(false);

MByteEn[inportn].write((sc\_uint<MBYTEENWD>)(8));

}

}

else

{

MData[inportn].write(0);

//definire tutti gli altri segnali OCP

MCmd[inportn].write(0);

MDataValid[inportn].write(true);

//generazione dei segnali MByteEn, MDataLast, MReqLast

if(wordToTransfer[inportn]\*8 - byteTxNumber[inportn] > 8)

{

MByteEn[inportn].write((sc\_uint<MBYTEENWD>)(8));

MDataLast[inportn].write(false);

}

else

{

if((access->get\_size() % 8) != 0)

MByteEn[inportn].write((sc\_uint<MBYTEENWD>)

(access->get\_size() % 8));

else

MByteEn[inportn].write((sc\_uint<MBYTEENWD>)(8));

MDataLast[inportn].write(true);

}

}

}

}

...

}

\end{lstlisting}

\begin{itemize}

\item[$\circ$] Nell'implementazione della NoC, come anche in quella di EIB, è stato scelto di utilizzare una banda di 128-bit/ciclo per tutte le SPE e una banda di 64-bit/ciclo per PPE e memoria. \texttt{outportn} viene impiegato per determinare chi è il destinatario della transazione; nel caso in cui questo sia la PPE o la memoria, la banda utilizzabile, anche da parte di una SPE, per la comunicazione è pari a 64-bit/ciclo (ovvero quella supportabile da PPE e memoria).

\item[$\circ$] Il numero minimo di cicli di clock, necessari al trasferimento di una transazione dal wrapper alla NI, dipende dalla dimensione del campo \emph{MDATAWD}, che è un parametro di progetto della network on chip. Il calcolo di tale quantità viene fatto a partire dalla dimensione del dato, all'interno del memory access (informazione contenuta nel campo \\_size) e da \emph{MDATAWD} della network interface; di default è stato scelto un valore, per quest ultimo pari a 64-bit. Nel caso in cui il destinatario della transazione sia la memoria o la PPE, il numero di richieste da fare alla NI, viene calcolato dividento il campo \\_size del memory access per 8-byte. In caso contrario, \\_size viene diviso per \emph{MDATAWD} moltiplicato per il fattore K. Tale fattore è stato introdotto per evitare di dover riconfigurare la NoC ogni volta che viene cambiato il parametro \emph{MDATAWD}. Da notare, infine, che attraverso la NoC non vengono effettivamente trasferiti dati, ma word composte da zeri. Infatti il fine ultimo della NoC è quello di simulare il trasferimento, producendo delle latenze che possono essere poi misurate per la valutazione delle prestazioni. A tale scopo, ridefinire il campo \emph{MDATAWD} a 128-bit o moltiplicarlo per il fattore K = 2 è assolutamente equivalente.

\item[$\circ$] \texttt{byteTxNumber} come visto, viene utilizzato per tenere il conto del numero di byte correntemente accettati dalla network interface. Finché tale valore è minore del numero totale di byte da trasferire, dato dal prodotto tra \texttt{wordToTransfer} e 8, non deve essere inviato alla network interface \texttt{MDataLast} alto.

\item[$\circ$] La fase di trasferimento segue il protocollo OCP, relativamente a una write di tipo burst (Capitolo~\ref{cap:capitolo3}). Se il numero di trasferimenti richiesti è pari a uno, perché il dato ha dimensione uguale o minore a \emph{MDATAWD}, allora insieme a \texttt{MReqLast} viene posto a uno anche \texttt{MDataLast}.

\item[$\circ$] \texttt{MSource} viene utilizzato per trasferire il puntatore del memory access verso il target della comunicazione.

\end{itemize}

\subsubsection{manageStoreRequest}

La funzione \emph{manageStoreRequest} viene eseguita durante lo \emph{start\\_of\\_cycle}. Questa si occupa di aggiornare le variabili globali utilizzate da \emph{convertFromMemoryAccessStToOcp} per la conversione dei memory access.

\begin{lstlisting}[caption=Funzione manageStoreRequest, label=lst:MSR]

void manageStoreRequest(int i)

{

MemoryAccessList:: iterator it = \_inputStBuffers[i]->begin();

MemoryAccess \*access = \*it;

int outportn = get\_port\_number(access);

/\* controlla se la richiesta del wrapper è stata accettata dalla NI \*/

if(SCmdAccept[i].read() || SDataAccept[i].read())

{

if(SDataAcceptNumber[i] < wordToTransfer[i] - 1)

{

if(i == 1 || i == 0 || outportn == 0 || outportn == 1)

byteTxNumber[i] = byteTxNumber[i] + 8;

else

byteTxNumber[i] = byteTxNumber[i] + ((MDATAWD\*K)/8);

SDataAcceptNumber[i]++;

}

else

{

\_inputStBuffers[i]->remove(it);

byteTxNumber[i] = 0;

SDataAcceptNumber[i] = 0;

semaphore[i] = 0; //rilascia la risorsa (NI)

}

}

}

\end{lstlisting}

\begin{itemize}

\item[$\circ$] \emph{SDataAcceptNumber} è una variabile utilizzata dalla funzione per tenere il conto del numero di word accettate dalla network interface.

\item[$\circ$] Quando tutte le word che compongono il dato oggetto del trasferimento sono state accettate, il memory access corrispondente può essere rimosso dal buffer di ingresso e il semaforo è riportato a zero in modo da rendere la network interface disponibile per una nuova transazione.

\end{itemize}

\subsubsection{convertFromMemoryAccessLdToOcp}

La funzione viene eseguita durante l'\emph{end\\_of\\_cycle} all'interno della funzione \emph{route}. Questa si occupa di trasformare i memory access di tipo load in transazioni OCP di tipo read.

\begin{lstlisting}[caption=Funzione convertFromMemoryAccessLdToOcp, label=lst:CFMALTO]

void convertFromMemoryAccessLdToOcp(MemoryAccess \*access, int inportn)

{

semaphore[inportn] = 2;

wordToTransfer[inportn] = 1;

//definire tutti gli altri segnali OCP

MBurstPrecise[inportn].write(true);

MBurstSeq[inportn].write(5); //trasferimento streaming

MBurstLength[inportn].write(((sc\_uint<MBURSTLENGTHWD>)access->get\_size()));

MBurstSingleReq[inportn].write(true);

MCmd[inportn].write(2);

MAddr[inportn].write((sc\_uint<MADDRWD>)(access->get\_target()));

uint32\_t maccess = (uint32\_t)access;

MSource[inportn].write((sc\_uint<SOURCEWD>)(maccess));

//generazione dei segnali MByteEn, MDataLast, MReqLast

MReqLast[inportn].write(true); //faccio un solo trasferimento burst

con il campo MBurstLength uguale a size del dato da leggere

MDataLast[inportn].write(false);

MDataValid[inportn].write(false);

MData[inportn].write(0);

MByteEn[inportn].write(0);

}

\end{lstlisting}

\begin{itemize}

\item[$\circ$] Le richieste di load vengono convertite in transazioni OCP di tipo read burst.

\item[$\circ$] \texttt{MBurstLenght} contiene la dimensione, in byte, del dato da leggere.

\item[$\circ$] \texttt{MSource} ha la stessa funzionalità vista nel caso di memory access store.

\end{itemize}

\subsubsection{manageLoadRequest}

La funzione \emph{manageLoadRequest} viene eseguita durante lo \emph{start\\_of\\_cycle} e si occupa di verificare che il comando di read venga accettato dalla network interface.

\begin{lstlisting}[caption=Funzione manageLoadRequest, label=lst:MLR]

void manageLoadRequest(int i)

{

if(SCmdAccept[i].read())

{

\_inputLdBuffers[i] = NULL;

semaphore[i] = 0;

}

}

\end{lstlisting}

Se il comando di read viene accettato, il buffer di ingresso, contenente il memory access, può essere liberato per lasciar spazio a una nuova richiesta. Il semaforo viene riportato a 0 in modo da liberare la network interface.

\subsubsection{convertFromOcpToMemoryAccessSt}

La funzione \emph{convertFromOcpToMemoryAccessSt} viene eseguita durante l'\emph{end\\_of\\_cycle} all'interno della funzione \emph{route} e si occupa di ricostruire, a partire dai segnali OCP in arrivo dalla network interface ricevente, il messaggio originale.

\begin{lstlisting}[caption=Funzione convertFromOcpToMemoryAccessSt, label=lst:CFOTMAS]

void convertFromOcpToMemoryAccessSt(int inportn)

{

if(!\_outputBuffers[inportn])

{

if(MCmd\_recv[inportn].read() == 1)

{

wordToReceive[inportn]=(uint32\_t)(MBurstLength\_recv[inportn].read());

source\_recv[inportn]=(uint32\_t)(MSource\_recv[inportn].read());

}

if(MDataValid\_recv[inportn])

{

if(MCmd\_recv[inportn].read() == 1)

if(MDataValid\_recv[inportn])

{

SCmdAccept\_recv[inportn].write(true);

SDataAccept\_recv[inportn].write(true);

}

else

SDataAccept\_recv[inportn].write(true);

}

if(MDataLast\_recv[inportn].read())

{

\_outputBuffers[inportn] = (MemoryAccess\*)source\_recv[inportn];

\_outputBuffers[inportn]->set\_latency(0);

\_outputBuffers[inportn]->set\_cycle\_output(cycle\_number,inportn);

byteRxNumber[inportn] = 0;

SCmdAccept\_recv[inportn].write(true);

SDataAccept\_recv[inportn].write(true);

}

}

else

{

SCmdAccept\_recv[inportn].write(false);

SDataAccept\_recv[inportn].write(false);

}

}

\end{lstlisting}

\begin{itemize}

\item[$\circ$] La funzione per prima cosa controlla che il buffer di uscita possa accogliere il memory access in arrivo dalla NoC.

\item[$\circ$] \texttt{wordToReceive} legge il numero word che sono state associate alla transazione e usa tale valore per la corretta gestione della fase di ricezione.

\item[$\circ$] \texttt{source\\_recv} viene utilizzato per memorizzare il puntatore al memory access inserito, dalla corrispondente funzione \emph{convertFromMemoryAccessStToOcp}, all'interno del segnale \emph{MSource}. Una volta terminata la ricezione dell'ultimo blocco di dati, tale puntatore viene trasferito nel buffer di uscita.

\item[$\circ$] \texttt{set\\_cycle\\_output} è un metodo della classe MemoryAccess che viene utilizzato per la valutazione della latenza del memory access all'interno della NoC.

\end{itemize}

\subsubsection{manageRecvStoreRequest}

La funzione \emph{manageRecvStoreRequest} viene eseguita durante lo \emph{start\\_of\\_cycle} e si occupa dell'aggiornamento delle variabili utilizzate dalla funzione \emph{convertFromOcpToMemoryAccessSt} per la gestione della corretta ricezione del messaggio.

\begin{lstlisting}[caption=Funzione manageRecvStoreRequest, label=lst:MRSR]

void manageRecvStoreRequest(int i)

{

if(SCmdAccept\_recv[i].read() || MDataValid\_recv[i])

{

if(!MDataLast\_recv[i] || MReqLast\_recv[i].read())

{

wordReceived[i]++;

SCmdAccept\_next[i] = true;

}

if(MDataLast\_recv[i].read())

{

wordReceived[i] = 0;

SDataAccept\_next[i] = true;

}

}

}

\end{lstlisting}

\begin{itemize}

\item[$\circ$] \emph{SDataAccept\\_next} e \emph{SCmdAccept\\_next} vengono utilizzate per far si che la NoC possa campionare i segnali \emph{SDataAccept} e \emph{SCmdAccept}. L'utilità di tali variabili sarà chiarita nel seguito del capitolo.

\end{itemize}

\subsubsection{convertFromOcpToMemoryAccessLd}

La funzione \emph{convertFromOcpToMemoryAccessLd} viene eseguita all'\emph{end\\_of\\_cycle} all'interno della funzione \emph{route} e si occupa della conversione delle transazioni OCP in arrivo dalla NoC in memory access.

\begin{lstlisting}[caption=Funzione convertFromOcpToMemoryAccessLd], label=lst:CFOTMAL]

void convertFromOcpToMemoryAccessLd(int inportn)

{

if(!\_outputBuffers[inportn])

{

wordToReceive[inportn] = 0;

SCmdAccept\_recv[inportn].write(true);

uint32\_t maccess = MSource\_recv[inportn].read();

\_outputBuffers[inportn] = (MemoryAccess\*)maccess;

\_outputBuffers[inportn]->set\_latency(0);

\_outputBuffers[inportn]->set\_cycle\_output(cycle\_number,inportn);

}

else

{

SCmdAccept\_recv[inportn].write(false);

}

}

\end{lstlisting}

\begin{itemize}

\item[$\circ$] \texttt{set\\_cycle\\_output} e \texttt{set\\_cycle\\_input} sono i metodi della classe MemoryAccess utilizzati per il calcolo della latenza del messaggio all'interno della NoC. Quando un memory access viene caricato nell'input buffer del wrapper gli viene assegnato il numero corrispondente al ciclo di sistema in corso; quando lo stesso memory access viene trasferito dalla network interface all'output buffer gli viene assegnato il valore del ciclo di sistema corrispondente. La latenza a questo punto può essere calcolata come differenza tra numero del ciclo di uscita e numero del ciclo di ingresso.

\end{itemize}

\subsubsection{manageRecvLoadRequest}

Lo scopo di questa funzione è analogo a quello di \emph{manageRecvStoreRequest}.

\begin{lstlisting}[caption=Funzione manageRecvLoadRequest, label=lst:MRLR]

void manageRecvLoadRequest(int i)

{

if(SCmdAccept\_recv[i].read())

{

SCmdAccept\_next[i] = true;

}

}

\end{lstlisting}

Tutto il codice del wrapper fin qui descritto è contenuto all'interno del file \emph{EIB\\_wrapper\\_noc.sim} nel path \emph{./cellsim-0.9/src/modules/processor}.

\subsubsection{start\\_of\\_cycle}

La funzione \texttt{start\\_of\\_cycle} gestisce l'aggiornamento del contatore \texttt{cycle\\_number}, il quale memorizza il numero di ciclo di esecuzione e che viene utilizzato per il calcolo della latenza all'interno del sistema. Inoltre, questa si occupa anche di invocare le funzioni per la gestione delle transazioni in corso, viste precedentemente, dell'inizializzazione della network on chip (\texttt{rst.write(true)}) e della generazione del segnale di clock (\texttt{xpipes\\_Clock} e \texttt{OCP\\_Clock}).

\begin{lstlisting}[caption=Funzione start\\_of\\_cycle, label=lst:start-of-cycle]

void start\_of\_cycle()

{

OCP\_Clock.write(true);

xpipes\_Clock.write(true);

cycle\_number+=1;

/\* la NoC viene resettata solo al primo start\_of\_cycle \*/

if(cycle\_number == 1)

{

rst.write(true);

}

for (int i = 0; i < PORT\_COUNT; i++)

{

if(MCmd[i].read() == 1 || MDataValid[i].read())

{

manageStoreRequest(i);

}

else

{

if(MCmd[i].read() == 2) manageLoadRequest(i);

}

if(MCmd\_recv[i].read() == 1 || MDataValid\_recv[i].read())

{

manageRecvStoreRequest(i);

}

else

{

if(MCmd\_recv[i].read() == 2)

{

manageRecvLoadRequest(i);

}

}

if (\_outputBuffers[i])

{

out\_port[i].data= \_outputBuffers[i];

}

else

{

out\_port[i].data.nothing();

}

}

...

}

\end{lstlisting}