POLITECNICO DI MILANO
Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Elettronica

PROGETTO DI UN'INTERFACCIA HARDWARE PER SOTTOPORRE A TEST UN PROCESSORE IN UN SISTEMA MULTIPROCESSORE ON-CHIP

Relatore: Chiar.ma Prof.ssa Mariagiovanna Sami
Correlatore: Ing. Leandro Fiorin

Tesina di laurea Specialistica di:
Andrea Ferrario
Matr. 739814

Anno Accademico 2011-2012

Indice

Lista delle figure

Lista delle tabelle

Lista delle abbreviazioni

ALFSR	Autonomous linear feedback shift register
BILBO	Built-in logic block observer
BIST	Built-in self-test
DAG	Directed acyclic graph
DUT	Device under test
FIFO	First in first out
FSM	Finite state machine
FPGA	Field programmable gate array
ISA	Instruction set architecture
ISE	Integrated synthesis environment
HBST	Hardware-based self-test
LFSR	Linear feedback shift register
LUT	Look-up table
MISR	Multiple input signature register
MPSoC	Multiprocessor system-on-chip
NI	Network Interface
NoC	Network-on-Chip
RISC	Reduced instruction set computer
RTL	Register transfer level
RV	Response verifier
SBST	Software-based self-test
SISR	Single input signature register
SoC	System-on-Chip
TG	Test generator

Capitolo 1

Introduzione

La crescente miniaturizzazione dei dispositivi elettronici, con la conseguente possibilità di integrare su un unico chip numeri sempre più grandi di transistori, e la richiesta da parte degli utenti di prestazioni sempre più elevate e funzionalità sempre maggiori hanno spinto i progettisti ad integrare diversi componenti complessi sullo stesso chip. La riduzione delle dimensioni e delle tensioni di alimentazione ha consentito alle industrie di semiconduttori di fabbricare un numero sempre maggiore di circuiti complessi, indicati col termine "System-on-Chip" (SoC), utilizzando un'area limitata e dissipando poca potenza. Tuttavia, questi componenti altamente integrati sono molto sensibili all'insorgenza di guasti statici e dinamici a causa della continua riduzione delle dimensioni dei transistori, del crescente numero di livelli di metallizzazione nei circuiti e dell'elevata frequenza cui operano i processori.

D'altro canto, la richiesta di continuità di funzionamento dei sistemi è notevolmente aumentata, dato che i sistemi in questione sono usati in una vasta gamma di applicazioni, a volte anche in ambienti dove i fattori esterni conducono a insorgenza di guasti. Per questo motivo i sistemi stessi presentano requisiti di affidabilità sempre più stringenti e quindi occorre rilevare i guasti. Là dove gli errori conseguenti ai guasti potrebbero essere particolarmente dannosi, i guasti devono essere rilevati e corretti con tecniche specifiche, in modo pienamente concorrente con la funzionalità nominale. Nel caso di altri sistemi per i quali una presenza limitata e circoscritta di errori nei risultati può essere accettabile, così da non richiedere un controllo continuo, il test può essere eseguito solo in particolare momenti come l'accensione o lo spegnimento o durante i periodi di inattività.

L'oggetto della presente trattazione sono i sistemi multiprocessore (MPSoC) eterogenei, che nello scorso decennio si sono affermati come un'importante classe di sistemi altamente integrati. Si tratta di SoC che utilizzano come componenti del sistema più processori programmabili con le loro CPU

e memorie. Il loro impiego è esteso a numerose applicazioni complesse [1] in cui vengono eseguiti diversi tipi di operazioni con requisiti di elevate prestazioni e basso consumo di potenza che non consentono l'uso di sistemi convenzionali.

Il collegamento tra i processori, la gerarchia di memoria e le periferiche di I/O avviene tramite una "Network-on-Chip" (NoC), modello emergente per le comunicazioni nell'ambito dei sistemi altamente integrati implementati su un unico chip. Infatti, i sistemi a bus sono stati a lungo utilizzati per la comunicazione tra i vari blocchi funzionali all'interno dei chip, ma si sono presto rivelati limitativi con l'avvento di sistemi complessi in cui più flussi di dati devono essere trasmessi in contemporanea. Per ovviare a questo limite ci si è ispirati alle caratteristiche, oltre che agli standard, delle reti di calcolatori, definendo così architetture in cui è possibile poter effettuare più trasferimenti di dati contemporaneamente. Si farà qui riferimento ad una struttura di NoC a "maglia bidimensionale", nella quale ad ogni processore è associato un router che presenta quattro porte globali connesse ai quattro router vicini e una porta locale connessa al core del processore, interfacciato con l'infrastruttura di comunicazione tramite un'interfaccia di rete ("Network Interface", NI), come mostrato in figura 1.1. L'informazione può essere trasmessa da qualsiasi sorgente a qualsiasi destinazione imponendo opportuni comandi ai router mediante un apposito protocollo di comunicazione. Viene raggiunto un elevato livello di parallelismo poiché tutti i collegamenti possono operare simultaneamente su diversi pacchetti di dati. Pertanto, poiché la complessità dei sistemi integrati è in continua crescita, la NoC offre migliori prestazioni e maggiore scalabilità rispetto alle precedenti architetture di comunicazione basate sulla soluzione a bus. Inoltre, la struttura regolare dei suoi collegamenti consente di contenere l'occupazione di area e il consumo di potenza delle interconnessioni.

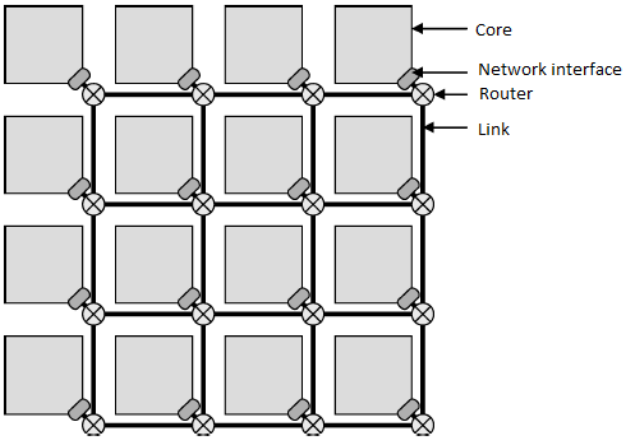

Figura 1.1: Struttura di una generica NoC [43]

In questo contesto si inserisce il problema dell'insorgenza dei guasti nei core dei processori, negli elementi di memoria e nella rete di comunicazione [2].

I guasti che si presentano al termine della produzione possono essere rilevati impiegando tecniche di test basate sull'uso di apparecchiature esterne ("Automatic Test Equipment", ATE), che però non consentono di individuare i guasti che sorgono durante il funzionamento del dispositivo. Inoltre, gli ATE sono particolarmente complessi e costosi e risulta difficile connetterli ai dispositivi collocati all'interno del chip.

Di conseguenza, è sorta l'esigenza di sviluppare tecniche di ricerca dei guasti che possano essere applicate anche durante il normale funzionamento del dispositivo e che non utilizzino apparecchiature esterne. Le soluzioni che sono state introdotte vengono indicate con i termini "self-checking" e "self-testing".

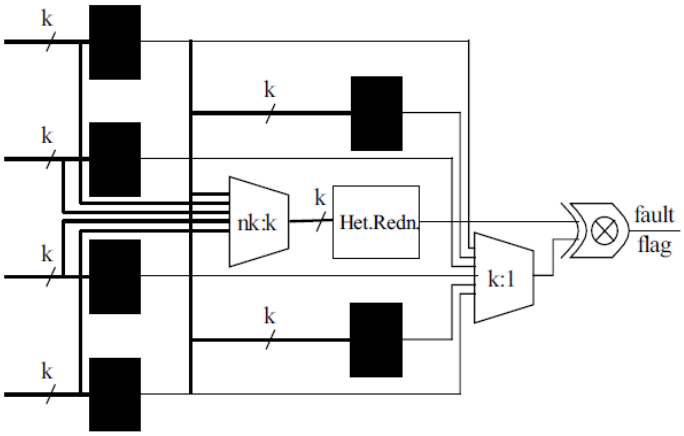

Figura 1.2: Ridondanza eterogenea per la rilevazione dei guasti [3]

Il self-checking si basa sulla suddivisione del circuito in blocchi e sull'inserimento di una o più unità hardware, ognuna delle quali in grado di implementare le funzionalità di un certo numero di blocchi [3][4]. Come mostrato in figura 1.2, un multiplexer seleziona gli ingressi di un certo blocco e li applica all'unità ridondante, configurata per implementare la funzionalità appropriata. Un altro multiplexer seleziona l'uscita del blocco considerato che viene confrontata con quella dell'unità ridondante. Un'eventuale discrepanza indica la presenza di un guasto. La percentuale di tempo in cui ogni blocco viene esaminato è data dal rapporto tra il numero di unità ridondanti e il numero di blocchi in cui viene suddiviso il circuito, quindi un eventuale guasto può non essere rilevato per un certo tempo. Un limite di questo approccio è l'incremento di area e di consumo di potenza del

circuito a causa dell'introduzione di unità hardware addizionali; inoltre, dato che si usano procedure e dati nominali, il rischio che il guasto non venga attivato e quindi rilevato è sempre presente.

Il self-testing, soluzione adottata in questo lavoro, si basa invece sul trasferimento di vettori di test progettati ad hoc al dispositivo sotto test e sulla verifica autonoma dei risultati del test compiuta da hardware addizionale inserito appositamente nel circuito originale. E' stata proposta la tecnica "software-based self-testing" (SBST) [5] che prevede l'applicazione di opportune routine di test al processore che le esegue alla propria velocità nominale durante il normale funzionamento. Viene raggiunta un'elevata capacità di rilevare i guasti minimizzando l'incremento di area e di consumo di potenza poiché si utilizzano le risorse programmabili e le istruzioni del processore per eseguire programmi che testino il processore stesso. L'SBST è una tecnica appropriata per testare i sistemi non convenzionali come gli MPSoC eterogenei, a differenza delle tecniche convenzionali basate sulla struttura del dispositivo sotto test, poiché non sempre si conosce il dettaglio strutturale di un core in un MPSoC.

Le routine di test sono salvate in memoria, programmate dal sistema operativo e applicate al processore sotto test che le esegue. I risultati vengono inviati ad un'interfaccia hardware, chiamata tester, che li compatta opportunamente ottenendo una *signature* che viene poi confrontata con quella corretta per verificare l'eventuale presenza di errori nel dispositivo. Per il caso qui discusso, il tester è stato precedentemente sviluppato da uno studente di altro Ateneo nel suo lavoro di tesi, i cui obiettivi erano la definizione di un'architettura hardware semplice e robusta che supportasse la tecnica SBST verificando i risultati del test in uscita dal processore, e la sua integrazione sul chip col minimo impatto sulle prestazioni, l'area e il consumo di potenza del sistema. Il tester è stato poi impiegato per sottoporre a test il processore MicroBlaze [54] interfacciato sul bus PLB. In questo lavoro si vuole invece testare il processore SecretBlaze che comunica con le periferiche tramite il bus Wishbone, quindi il tester è stato opportunamente adattato per essere connesso come dispositivo slave al nuovo bus che impiega protocolli di lettura e scrittura differenti.

Nel capitolo 2 si presentano le tecniche attualmente utilizzate per testare i circuiti integrati, evidenziandone anche i punti di debolezza. Si descrivono in particolare quelle software. Il tester supporta infatti la tecnica SBST per i motivi sopra citati.

Nel capitolo 3 si descrive l'architettura MPSoC in cui tutti i processori devono essere interfacciati opportunamente per poter essere sottoposti a test. In seguito, si analizza l'architettura di nodo

presentando le specifiche del processore SecretBlaze e del bus Wishbone. Infine, si discute del trasferimento dei programmi di test al nodo e dei risultati del test alla rete.

Nel capitolo 4 si illustrano la struttura del tester e la sua implementazione su FPGA, quindi si discutono i risultati delle simulazioni svolte per verificare il corretto funzionamento dell'interfaccia.

Nel capitolo 5 si espongono le conclusioni e i possibili sviluppi futuri.

In appendice si forniscono i file VHDL che sono stati scritti per interfacciare il tester sul bus Wishbone e per verificarne il corretto comportamento. In seguito, si includono i file scritti in linguaggio C per accedere agli spazi di memoria contenuti nel tester tramite software in modo tale da realizzare un ulteriore test bench.

Capitolo 2

Stato dell'arte

Secondo quanto esposto nel capitolo precedente, l'utilizzo di apparecchiature esterne per sottoporre a test i dispositivi integrati su un singolo chip è problematico e costoso, comunque non adottabile per rilevare guasti che insorgono in esercizio, quindi sono state proposte le tecniche "built-in self-test" (BIST) [6], che prevedono l'integrazione sul chip delle unità hardware necessarie per l'esecuzione delle operazioni di test. Ci si focalizzerà qui su quest'ultima classe di soluzioni.

Algoritmi per la generazione di vettori di test, ossia delle sequenze di configurazioni di ingresso da applicare ad un dispositivo per sottoporlo a test, vengono impiegati per sviluppare sequenze di vettori per la rilevazione dei guasti opportunamente modellati in un circuito. I modelli di guasto devono risultare congruenti con il livello di astrazione utilizzato per descrivere il circuito e con la tecnologia impiegata per realizzarlo. La simulazione di guasto consente poi di determinare la copertura di guasto (*fault coverage*), ossia la percentuale di guasti dell'insieme considerato che vengono rilevati dalla sequenza di vettori. La generazione di tali vettori può essere esaustiva o, più frequentemente, orientata al guasto o ancora pseudo - casuale a seconda che si generino o meno tutti i vettori possibili. Nel secondo caso si seleziona la lunghezza del test mediante simulazioni in modo da ottenere una buona copertura. Il test esaustivo è una tecnica di test off-line, ossia basata sull'interruzione delle operazioni del sistema per effettuarne il test, che non richiede un modello di guasto e che non implica la memorizzazione dei vettori di test sul chip; tali vettori possono essere creati da un generatore di vettori di test ("test generator", TG), che può essere realizzato mediante un contatore binario, ed applicati durante la modalità test. Considerando un dispositivo sotto test ("device under test", DUT) di tipo combinatorio dotato di n ingressi, è necessario generare tutti i 2^n possibili vettori, quindi, per valori crescenti di n, la complessità e il ritardo combinatorio di un tale contatore tendono a crescere oltre una soglia di accettabilità. Si ricorre pertanto ai registri a scorrimento autonomi a retroazione lineare ("autonomous linear feedback shift register", ALFSR) [52], una cui possibile struttura è rappresentata in figura 2.1, per poter generare sequenze pseudo -

casuali di vettori di bit di lunghezza prefissata, ma arbitraria. Nello specifico si ricorre a LFSR a lunghezza massima, ossia in grado di generare tutte le possibili combinazioni di n bit a partire da uno stato iniziale, come avviene nell'esempio mostrato in figura 2.2, in cui i valori alle uscite dei flip-flop costituiscono i vettori di test. Tuttavia, quando il numero di ingressi del DUT è molto elevato, il test esaustivo non può essere applicato poiché richiederebbe un tempo eccessivo. Per superare questo limite, sono state proposte altre tecniche come il test pseudo - esaustivo [7][8], il "verification testing" [9] e il test con l'uso di dati in ingresso casuali. Secondo la tecnica pseudo - esaustiva si suddivide il DUT in sezioni e si procede al test esaustivo di ogni sottosezione riducendo notevolmente la complessità.

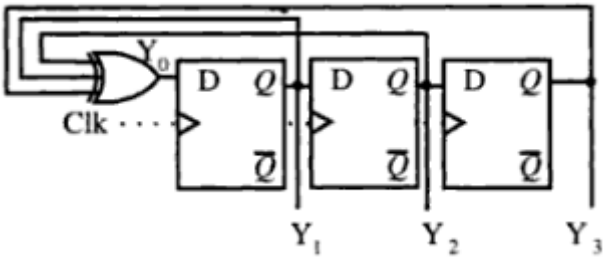

Figura 2.1: Registro a scorrimento autonomo a retroazione lineare [52]

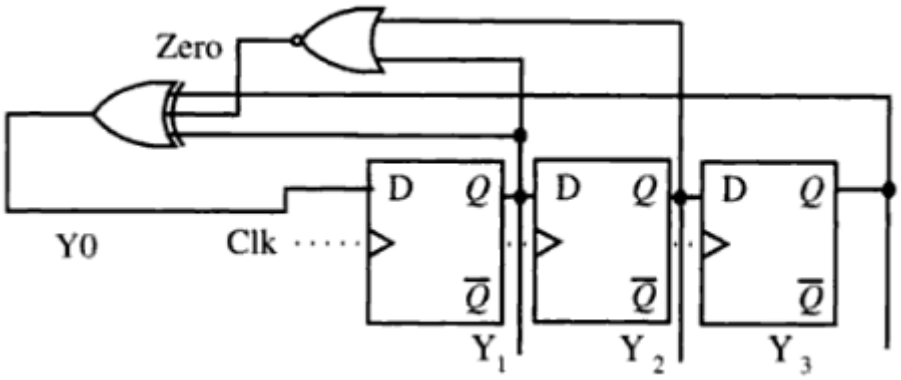

Figura 2.2: ALFSR a lunghezza massima [52]

Durante l'applicazione dei vettori di test al DUT, viene prodotto un insieme di dati di uscita che vengono raccolti e compressi dal verificatore di risposta ("response verifier", RV), in modo tale da ottenere una *signature* che viene poi confrontata con quella prevista per un circuito privo di guasti così da verificare l'eventuale presenza di errori nel DUT. Per ogni guasto e per gli errori associati

nella risposta del circuito, la tecnica di compressione impiegata deve generare una signature diversa da quella del circuito sano, altrimenti può accadere che un guasto non venga rilevato. A seconda del numero di bit di ingresso da cui vengono alimentati, gli analizzatori della signature, introdotti nel 1977 [10], si dividono in due categorie principali: "single input signature register" (SISR) e "multiple input signature register" (MISR), i cui esempi sono riportati rispettivamente nelle figure 2.3 e 2.4. Il SISR è un LFSR con un solo segnale d'ingresso, il cui registro memorizza il resto di una divisione modulo due, mentre il MISR è un LFSR con più di un ingresso che permette di raccogliere e comprimere più di un valore d'uscita. Grazie al meccanismo della signature, è possibile ridurre considerevolmente il numero di controlli e i requisiti di memoria per le risposte attese rispetto al caso in cui si vogliano confrontare tutti i valori ottenuti con i corrispondenti valori attesi.

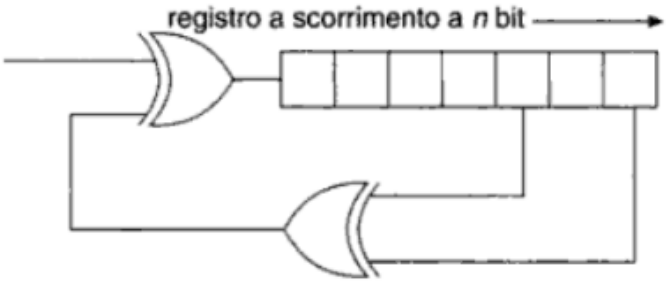

Figura 2.3: SISR

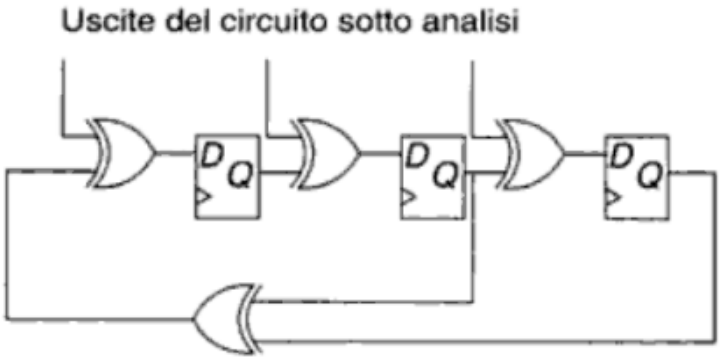

Figura 2.4: MISR

La tecnica "syndrome testing" [11] è tipica di circuiti sequenziali, ma può essere applicata anche ad un circuito combinatorio che viene sottoposto a test in modo esaustivo mentre vengono compressi i dati in uscita. In questo caso, la compressione dei risultati del test è indipendente dall'ordine in cui i vettori di test sono applicati agli ingressi, a differenza delle altre tecniche basate sul calcolo della signature, ma il meccanismo di compressione è molto più complesso. Il DUT può essere reso idoneo al "syndrome testing" incrementandone il numero di ingressi. Durante il funzionamento normale del circuito, agli ingressi aggiunti, che devono essere validi durante la modalità test, devono essere assegnati valori tali da non alterare la funzione svolta dal circuito originario.

La tecnica "built-in logic block observer" (BILBO) [12], tipica di reti sequenziali, combina le funzioni di TG e RV e può essere impiegata nell'ambito di un test pseudo - casuale, esaustivo o pseudo - esaustivo. Come illustrato in figura 2.5, la struttura è costituita da un banco di flip-flop e da alcune porte logiche. In base ai valori logici assunti dai due ingressi di controllo, sono disponibili quattro diversi modi di funzionamento, tra cui quello normale, ove i registri aggiunti sono registri funzionali del sistema.

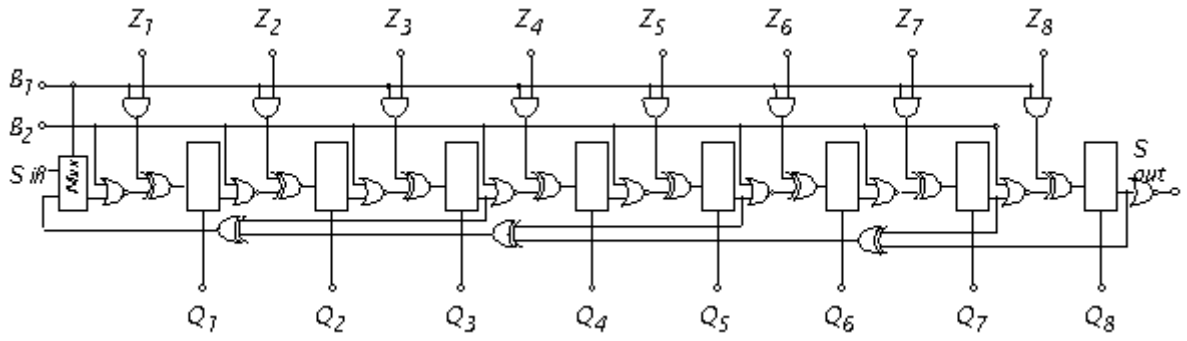

Figura 2.5: Possibile implementazione della struttura BILBO [12]

Si illustrano ora dettagliatamente le tre principali tecniche attualmente utilizzate per testare i circuiti integrati: le due principali tecniche BIST e il "software-based self-test" (SBST). In seguito, si discutono i motivi che hanno portato ad impiegare le tecniche software in questo lavoro.

2.1 BIST

La struttura dello schema BIST [13] è mostrata in figura 2.6.

La tecnica prevede che il test venga eseguito periodicamente o in particolari momenti durante il funzionamento del sistema, e consente di rilevare non solo i guasti permanenti, ma anche quelli intermittenti con elevata probabilità.

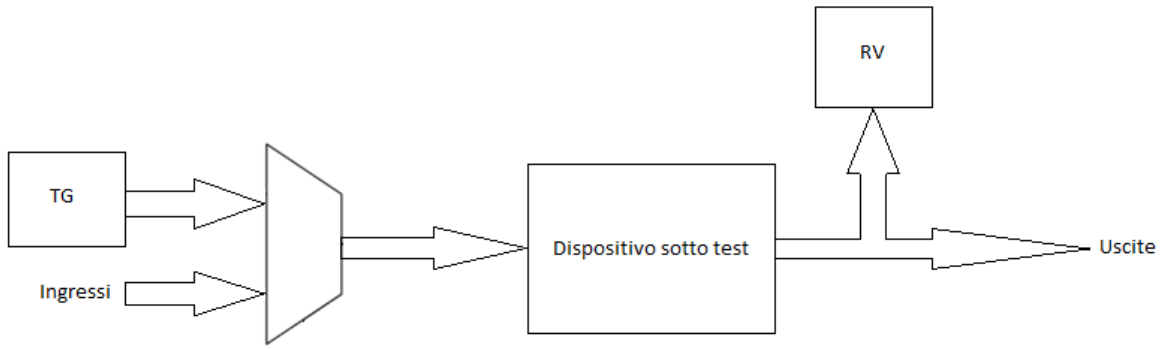

Figura 2.6: Struttura dello schema BIST

Le due modalità di funzionamento, test e normale, si svolgono come segue.

2.1.1 Modalità test

Si tratta della modalità test off-line, in cui i vettori di test generati dal TG vengono applicati al DUT tramite il multiplexer e i risultati vengono compressi dall'RV. Al termine dell'applicazione di tutti i vettori, il contenuto dell'RV viene esaminato per verificare l'eventuale presenza di errori nel DUT.

2.1.2 Modalità normale

Durante questa modalità, al DUT vengono forniti gli ingressi nominali e il circuito funziona normalmente. Non viene eseguito alcun test, quindi il TG e l'RV rimangono inattivi.

2.2 Concurrent BIST

La struttura dello schema C-BIST è mostrata in figura 2.7.

La tecnica consente di rilevare anche i guasti transitori e quelli intermittenti il cui effetto scompare rapidamente, ma determina una maggiore occupazione di area rispetto alla soluzione precedente ed è quindi più costosa. Infatti, l'architettura differisce dalla precedente per l'aggiunta di un comparatore che confronta gli ingressi nominali del DUT con le uscite del TG, e di un circuito che impartisce opportuni comandi al TG e all'RV.

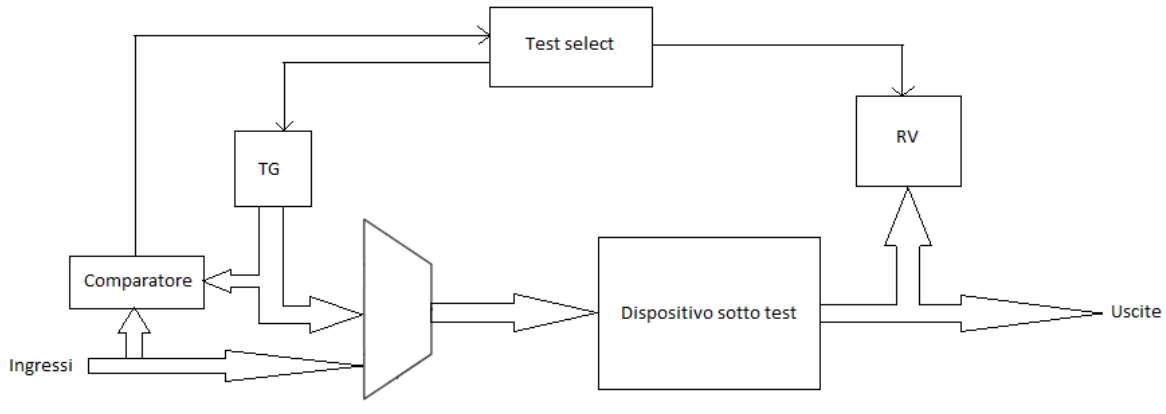

Figura 2.7: Struttura dello schema C-BIST

Il sistema opera mentre viene sottoposto a test, quindi si tratta di una tecnica di test on-line. L'RV deve comprimere solo i risultati corrispondenti all'applicazione dei vettori di test agli ingressi, quindi una logica confronta gli ingressi nominali con quelli di test per abilitare correttamente l'RV. Durante il funzionamento normale, le uscite del TG, ossia i vettori di test, vengono confrontate bit per bit con gli ingressi nominali del DUT tramite il comparatore, che genera un segnale d'uscita alto se e solo se l'esito del confronto è un'uguaglianza. In questo caso, l'RV è abilitato a comprimere le uscite del circuito e il TG può applicare il vettore successivo. Una volta che tutti i vettori di test sono stati generati e applicati al DUT, il contenuto dell'RV viene esaminato per determinare lo stato del DUT.

2.3 SBST

Le tecniche "hardware-based self-test" (HBST), come il "built-in self-test" (BIST), consentono di raggiungere un'ottima qualità del test poiché permettono di testare il processore alla sua velocità nominale. Tuttavia, nel caso in cui i processori presentino requisiti stringenti in termini di prestazioni, occupazione di area e consumo di potenza, l'applicazione delle soluzioni HBST è limitata, quindi sono state proposte le tecniche "software-based self-test" (SBST) [14] come alternativa economica. Tali strategie prevedono l'utilizzo delle risorse programmabili e delle istruzioni del processore per l'esecuzione dei programmi di test, quindi determinano un impatto minimo sulle prestazioni, l'area e il consumo di potenza del sistema.

2.3.1 Tecniche SBST funzionali

Le tecniche SBST funzionali vengono applicate a processori in quanto usano il set di istruzioni del processore ("instruction set architecture", ISA) per generare un programma di test composto da sequenze di istruzioni randomizzate. Tale approccio comporta un costo ridotto grazie al suo elevato livello di astrazione, ma necessita di un numero consistente di sequenze di istruzioni e quindi di un lungo tempo di esecuzione del programma di test per ottenere un'elevata fault coverage.

Il metodo descritto in [15] si basa sull'ISA del processore e sulle operazioni svolte dal processore in risposta ad ogni istruzione per generare una sequenza casuale di istruzioni che elenca tutte le combinazioni delle operazioni e degli operandi sistematicamente selezionati. La sua applicazione al processore GL85 prevede l'esecuzione di un programma di test composto da 360.000 istruzioni ad ottenere una copertura di guasto del 90.2%.

La tecnica "instruction randomized self test" (IRST) [16] è un metodo BIST che combina l'esecuzione di istruzioni del microprocessore con l'impiego di hardware per generare una sequenza pseudo - casuale di istruzioni, e quindi un programma di test randomizzato. Considerando il core di un processore a 16 bit con architettura RISC, si ottiene una copertura di guasto del 94.8% al termine di un processo iterativo ripetuto per 220.000 cicli.

La tecnica "functional random instruction testing" (FRIT) [17] è basata sulla generazione di sequenze casuali di istruzioni con dati pseudo - casuali creati da LFSR implementati in software, e sull'applicazione dei programmi di test tramite la cache. Al generatore vengono imposti determinati vincoli per garantire la creazione di sequenze di istruzioni valide, evitando che si verifichino cache miss e accessi al bus. Il metodo, applicato al processore Intel Pentium 4 [53], consente di ottenere una fault coverage del 70%.

2.3.2 Tecniche SBST strutturali

Le tecniche SBST strutturali prevedono la generazione di una routine di test per ogni componente del processore, sulla base di una strategia di generazione di vettori di test ("test pattern generation", TPG). L'insieme di tutte le routine di test costituiscono un programma di test che viene eseguito in momenti specifici o durante i periodi di inattività per rilevare eventuali guasti strutturali, sia permanenti che intermittenti. Tale approccio comporta costi di sviluppo maggiori rispetto a quello funzionale, ma richiede meno istruzioni per ottenere un'elevata fault coverage, quindi il programma di test generato presenta un tempo di esecuzione inferiore.

Un metodo automatico per la generazione di un programma di test [18] utilizza un grafo aciclico diretto ("directed acyclic graph", DAG) per rappresentare il flusso sintattico di un programma, e una libreria di istruzioni per descrivere la sintassi dell'ISA del processore, come mostrato in figura 2.8. Ogni nodo del DAG comprende un puntatore all'interno della libreria di istruzioni e, quando richiesto, i suoi parametri, come illustrato in figura 2.9. I programmi di test sono generati tramite un

algoritmo che modifica la topologia del DAG e i parametri all'interno dei nodi. Gli ingegneri si limitano ad elencare tutte le istruzioni disponibili e tutti i loro possibili operandi. Il metodo è stato applicato ad un microcontrollore Intel, basato su core 8051, in cui sono presenti circa 12.000 porte logiche, e ha consentito di ottenere una fault coverage del 90.7%.

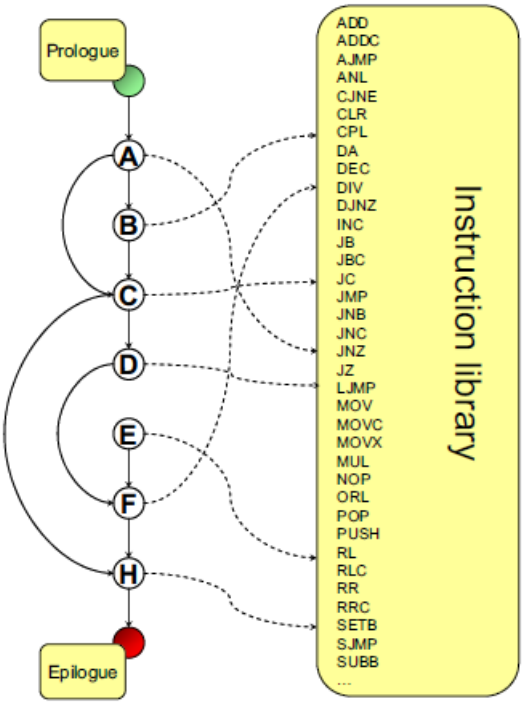

Figura 2.8: DAG e libreria di istruzioni [18]

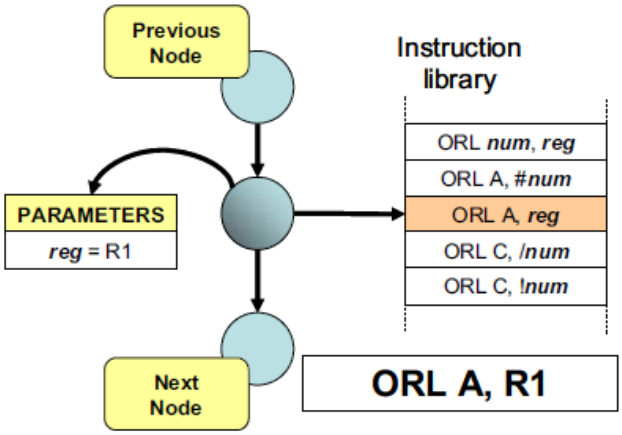

Figura 2.9: Nodo sequenziale [18]

La tecnica descritta in [19], il cui funzionamento è mostrato in figura 2.10, prevede lo sviluppo di test strutturali per ogni componente del processore con un metodo iterativo, sulla base di vincoli, estratti manualmente, che vengono imposti dal set di istruzioni del processore. Le sequenze di test così ottenute vengono compattate in modo tale da formare signature che vengono salvate in memoria ed espanse tramite LFSR implementati in software. Si ricavano vettori di test pseudo - casuali che vengono applicati al componente alla velocità nominale del processore. La compressione dei risultati del test genera una signature che viene salvata in memoria e successivamente scaricata e analizzata da un tester esterno. L'applicazione della tecnica al processore Parwan, composto da 888 porte logiche e 53 flip-flop, consente di ottenere una fault coverage del 91.4% impiegando un programma di test di 1.129 byte eseguito in 137.649 cicli. L'approccio pseudo - casuale non considera la struttura regolare dei componenti critici del processore, quindi determina una notevole occupazione di memoria e un eccessivo tempo di esecuzione del programma di test. Inoltre, l'estrazione manuale dei vincoli per i componenti considerati è un'operazione poco pratica.

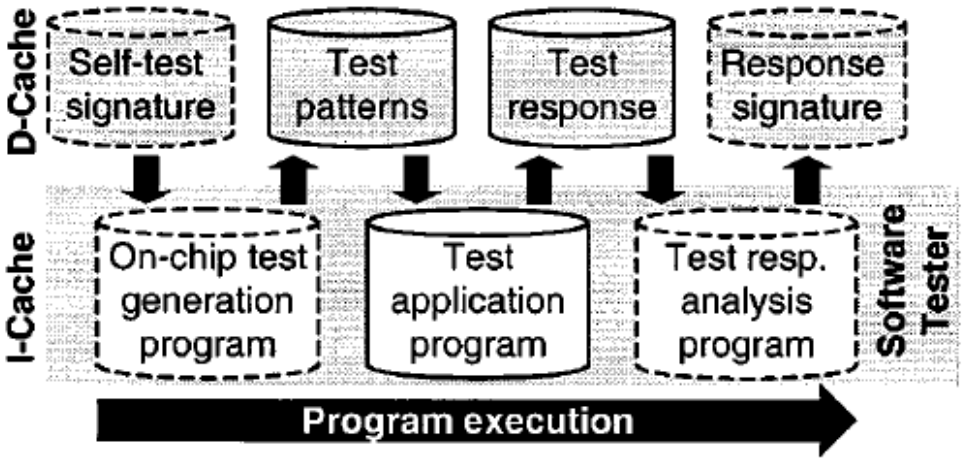

Figura 2.10: Metodo SBST [19]

E' stato pertanto proposto un metodo migliorativo [20] che automatizza la complessa fase di estrazione dei vincoli e che si basa sulla generazione automatica dei vettori di test ("automatic test pattern generation", ATPG) per lo sviluppo dei programmi di test. Tale tecnica, fondata sull'implementazione a livello di porta logica del processore, è applicabile soltanto ai componenti combinatori, di cui non si sfrutta l'eventuale struttura regolare. Inoltre, il programma di test è lungo,

e quindi il costo del test è elevato, a causa della notevole quantità di vettori di test che vengono salvati in memoria. L'applicazione del metodo ad un componente combinatorio del processore RISC Xtensa con 24.962 guasti prevede la generazione di un programma di test di 20.373 byte, e consente di ottenere una fault coverage del 95.2%.

Le tecniche SBST basate sui dettagli a livello di porta logica del processore consentono di raggiungere un'elevata fault coverage, ma presentano un notevole costo di sviluppo del test a causa dell'incremento delle dimensioni dei circuiti e della complessità degli attuali processori.

Di conseguenza, è stato proposto un metodo strutturale ad alto livello [21] che prevede uno sviluppo delle routine di test basato sulla descrizione a livello RTL ("register transfer level") del processore e sul suo ISA, in modo tale da ottenere una strategia di sviluppo del test a basso costo e indipendente dalla tecnologia impiegata. Le routine vengono salvate in memoria e successivamente eseguite alla velocità nominale del processore per generare vettori di test deterministici che testino il completo set di operazioni svolte da ogni componente. La tecnica viene applicata al semplice processore Parwan impiegato anche nel metodo [19], e consente di ridurre la dimensione del programma di test del 18.2% e il tempo di esecuzione del test dell'87.9%. Nonostante venga impiegato un approccio deterministico per lo sviluppo del test, quando il metodo viene applicato ai moderni processori complessi, che comprendono componenti funzionali e banchi di registri di grandi dimensioni, il costo del test cresce.

La tecnica descritta in [22] dimostra che lo sviluppo del test ad alto livello basato sulla descrizione a livello RTL del processore e sul suo ISA permette di mantenere basso il costo del test senza compromettere l'elevata fault coverage, indipendentemente dalla complessa implementazione del sistema. Come mostrato in figura 2.11, i componenti del processore sono suddivisi in tre classi che vengono ordinate in base alla priorità con cui le routine di test vengono sviluppate per ogni componente. Gli elementi funzionali hanno la priorità più alta poiché occupano gran parte dell'area del sistema e presentano ottime caratteristiche di controllabilità e osservabilità. La controllabilità misura la facilità con cui la logica interna di un circuito può essere controllata a partire dagli ingressi primari, mentre l'osservabilità misura la facilità con cui i valori presenti sulle linee interne di un circuito possono essere resi visibili alle uscite primarie. La classe dei componenti funzionali comprende i componenti combinatori che eseguono operazioni aritmetiche e logiche, i componenti di interconnessione che dirigono il flusso di dati e istruzioni e i componenti di memoria. La struttura regolare della maggior parte di tali elementi consente lo sviluppo di algoritmi che permettono di ottenere routine di test composte da un numero limitato di istruzioni, quindi si riduce lo spazio di memoria occupato dal programma di test, che viene eseguito in un tempo limitato. Di conseguenza,

l'approccio deterministico sembra essere il più appropriato per il test periodico on-line. Il metodo viene applicato ad un processore Plasma/MIPS con 3 stadi di pipeline e ad un processore MIPS R3000 con 5 stadi di pipeline, e consente di ottenere una fault coverage di circa il 94% utilizzando programmi di test rispettivamente di 853 e 1.728 parole.

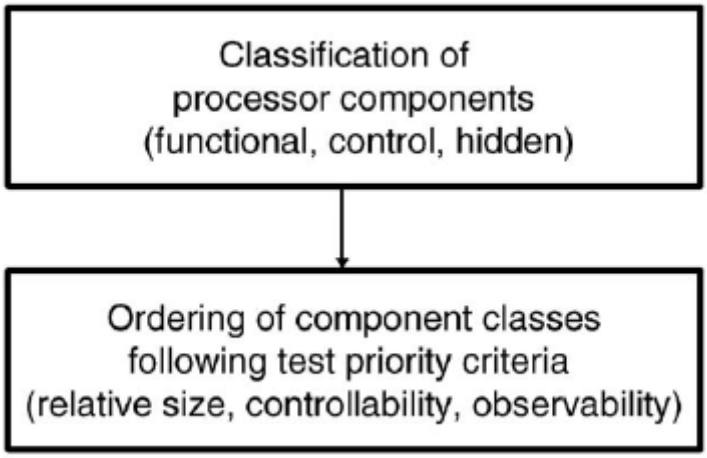

Figura 2.11: Classificazione dei componenti del processore e disposizione delle classi [22]

La presenza di componenti funzionali privi di una struttura regolare talvolta non consente di ottenere una fault coverage accettabile se si sottopongono a test soltanto tali elementi. E' stato pertanto proposto un metodo [23] che consente di testare anche i componenti di controllo tramite routine di test generate con un processo iterativo, che, per ogni componente, viene ripetuto fino ad ottenere un elevato livello di copertura in tutte le metriche funzionali. La fault coverage viene valutata al termine di ogni iterazione, e il processo viene applicato ai componenti non ancora testati fino al raggiungimento di un valore percentuale soddisfacente. Se la fault coverage ottenuta dopo aver considerato tutti i componenti non fosse ancora accettabile, si ricorrerebbe alla generazione di programmi di test casuali ("random test program generation", RTPG) per individuare i residui guasti non rilevati.

2.4 Scelta delle soluzioni impiegate in questo lavoro

Secondo quanto esposto nei paragrafi precedenti, sono state sviluppate numerose tecniche per testare i circuiti anche durante il loro normale funzionamento, e di ogni soluzione occorre considerare la fault coverage, il tempo di latenza del test e l'impatto sulle prestazioni, l'area e il consumo di potenza del sistema.

Nella tecnica BIST i vettori di test sono generati casualmente o pseudo - casualmente, quindi potrebbero non essere applicabili durante il funzionamento normale del DUT. Inoltre, non essendo tali vettori specifici per un determinato processore, la fault coverage ottenuta è veramente scarsa, pertanto in alcuni tipi di applicazioni è opportuno ricorrere ad un approccio differente.

Nella tecnica C-BIST l'elevato tempo di latenza del test condiziona la velocità del circuito durante il suo normale funzionamento, e il consistente impiego di logica di controllo determina occupazione di area e consumo di potenza.

Questi approcci non sono indicati per il nostro fine, anche se oggigiorno sono i più comuni. Come già illustrato nel capitolo precedente, l'oggetto della trattazione sono i sistemi multiprocessore (MPSoC) eterogenei, costituiti da diversi tipi di core con caratteristiche e istruzioni differenti, quindi non è possibile utilizzare un metodo convenzionale per testarli. Inoltre, è necessaria un'elevata fault coverage, superiore al 90%, ed è richiesto il minimo incremento di occupazione di area e di consumo di potenza, poiché si tratta di sistemi integrati molto costosi. Di conseguenza, non essendo possibile ottenere tali risultati con le soluzioni BIST tradizionali, sono state impiegate le tecniche software.

Come già esposto nei paragrafi precedenti, le tecniche SBST rappresentano un'alternativa economica alle soluzioni hardware e presentano alcune caratteristiche che le rendono particolarmente interessanti:

- utilizzo delle risorse programmabili e delle istruzioni del processore per eseguire programmi che testino il processore stesso, senza la necessità di appositi tester o di generatori di vettori di test;
- esecuzione delle routine di test alla velocità nominale del processore;
- assenza di overtesting per i guasti che non si manifestano durante il normale funzionamento del circuito;
- applicazione del test mentre il processore opera nelle sue normali condizioni d'esercizio.

L'efficienza di un approccio SBST dipende dalla metodologia di sviluppo del programma di test e dall'efficacia delle routine. Il fattore chiave per il suo successo, e quindi l'obiettivo principale di molti studi recentemente svolti nell'ambito dell'SBST, risiede nell'automatizzazione del processo di generazione dei programmi di test per ridurre i costi di sviluppo del test.

Tuttavia, l'obiettivo di questo lavoro non è definire le routine, bensì adattare ad un caso specifico un'interfaccia hardware semplice e robusta, chiamata tester, precedentemente sviluppata per supportare le tecniche software, anche se normalmente nell'SBST i risultati del test vengono caricati su un tester esterno e controllati a posteriori. In questa sede si intende integrare sul chip un supporto hardware che verifichi i risultati del test in uscita dal processore col minimo impatto sulle prestazioni, l'area e il consumo di potenza del sistema. Per lo sviluppo dell'interfaccia sono stati combinati due concetti:

- i vettori di test vengono applicati al processore e i risultati del test sono raccolti e controllati da un RV, come avviene nella tecnica BIST;
- un software esterno genera routine di test che vengono poi salvate direttamente in memoria e programmate dal sistema operativo, quindi non si utilizza il TG impiegato nella tecnica BIST.

L'idea fondamentale è che venga sollevata un'eccezione se si riscontra una discrepanza tra i risultati effettivi e quelli attesi. In questo caso, un'ulteriore logica interviene assegnando i compiti del processore guasto ai processori funzionanti, sulla base del carico di lavoro e delle prestazioni, e trasferendoli fisicamente.

Capitolo 3

Problema specifico

3.1 Architettura MPSoC

In questa sezione si discutono le motivazioni che hanno portato allo sviluppo dei sistemi multiprocessore su singolo chip ("multiprocessor system-on-chip", MPSoC), in cui sono presenti numerosi processori di diversa natura, memorie, interfacce di periferiche e unità dedicate. In queste architetture ci si è scontrati con il limite fisico di un unico bus di essere efficiente al crescere del numero di dispositivi, quindi si è passati ad utilizzare come infrastruttura di comunicazione una network-on-chip, i cui componenti fondamentali sono i router e le interfacce di rete.

3.1.1 Introduzione

I progressi tecnologici degli ultimi anni hanno determinato un punto di svolta per quanto riguarda i paradigmi di computazione; le classiche architetture single-core sono state progressivamente abbandonate e sostituite da quelle multi-core. Questo cambiamento è stato causato dal continuo aumento delle prestazioni richieste ai sistemi integrati, unitamente al fatto che tali prestazioni non potevano più essere garantite dalle architetture single-core, ormai giunte al loro limite fisico in termini di potenza dissipata, calore generato e frequenza di clock [24]. In questo contesto un cambio di direzione verso i sistemi multi-core si è reso obbligatorio ed è stato senza dubbio favorito dai progressi tecnologici quali la continua riduzione delle dimensioni dei transistori; analizzando i dati delle microarchitetture prodotte da Intel negli ultimi anni, riportati in tabella 3.1, risulta evidente quanto affermato.

Microarchitettura	Anno di rilascio	Processo di fabbrica
Core	2006	65 nm
Nehalem	2008	45 nm
Sandy Bridge	2011	32 nm
Haswell	2013	22 nm

Tabella 3.1: Processo di fabbrica delle microarchitetture Intel [25][26]

L'evoluzione del processo produttivo dei processori e la loro continua miniaturizzazione ha permesso di integrare su un unico chip più elementi processanti. Ciò ha spinto la ricerca ad esplorare architetture sempre più complesse e con un numero sempre maggiore di core, in grado di accelerare le applicazioni al fine di garantire le prestazioni loro richieste: Intel ha proposto un'architettura ad 80 core [27], AMD propone in commercio sistemi fino a 16 core, mentre progetti più ambiziosi hanno portato alla creazione di architetture con 1024 core [28]. La continua miniaturizzazione dei transistori ha quindi permesso di aggregare sempre più componenti su un singolo chip andando dunque a realizzare sistemi chiamati appunto "multiprocessor system-on-chip" (MPSoC) [30]. La famiglia delle architetture multiprocessore, in continua espansione sul mercato [31], si divide in due grandi tipologie di architetture caratterizzate l'una dall'avere gli elementi processanti tutti uguali tra loro, identificate appunto come architetture omogenee, l'altra dal possedere una differente topologia di elementi processanti e generalmente differenti caratteristiche nella gestione della memoria, identificate con il termine di architetture eterogenee. Le soluzioni sopra richiamate fanno tutte riferimento al caso di un'architettura omogenea, ma anche per il caso di architetture eterogenee si è assistito ad una continua ricerca nel settore per ottenere prestazioni sempre migliori; basti pensare al sempre maggiore utilizzo di unità di elaborazione grafica ("graphical processing unit", GPU), che da semplici acceleratori per grafica tridimensionale si sono evolute in vere e proprie unità programmabili da utilizzare per effettuare grandi quantità di calcoli in virgola mobile [33]. La scelta di sostituire sempre più parti del sistema, progettate specificatamente per una particolare applicazione ("application specific integrated circuit", ASIC), con architetture programmabili [32] ha consentito ai progettisti di riutilizzare moduli funzionali, detti IP cores, che implementano le funzionalità principali del sistema e che possono essere unità di calcolo, processori, memorie, periferiche o blocchi che svolgono compiti più specifici. L'approccio basato sull'impiego di librerie di componenti preesistenti fornisce la soluzione alle problematiche stringenti del progettista: si pensi alla riduzione del time to market dovuta all'utilizzo di componenti già esistenti e generalmente già testati, all'abbattimento dei costi di progetto garantito dalla

riusabilità dei blocchi funzionali, all'incremento delle prestazioni e alla riduzione dell'area e della dissipazione di potenza conseguenti alla ripetuta ottimizzazione di ciascun modulo. Inoltre, i concetti di riutilizzo e di alta riconfigurabilità pongono il progettista nella condizione di poter lavorare ad un livello molto più alto rispetto ai precedenti flussi di progetto, e quindi possono essere considerati come i punti di forza di questa rivoluzione nella progettazione VLSI.

Lo stesso andamento si è avuto nel campo dei dispositivi riconfigurabili, ossia che non hanno una funzionalità decisa al momento della fabbricazione, ma che possono essere di volta in volta riconfigurati secondo le necessità dell'utente, eventualmente anche durante l'esecuzione dell'applicazione. La loro dimensione ora non è più un vincolo così stringente da permettere di realizzare solo sistemi relativamente semplici. Infatti, tali dispositivi possono essere utilizzati per realizzare anche sistemi molto complessi quali ad esempio architetture multi-core molto avanzate o addirittura architetture con un numero di elementi processanti che può adattarsi dinamicamente al carico di lavoro del sistema [29], realizzando così MPSoC riconfigurabili a run-time.

3.1.2 Scelta della modalità di interconnessione

La progettazione degli MPSoC si è da sempre scontrata con due importanti problematiche: la modalità di gestione dello scambio di dati tra diversi IP e l'interfacciamento degli IP con il mezzo di comunicazione.

I sistemi a bus sono stati a lungo utilizzati per la comunicazione tra i vari blocchi funzionali all'interno dei chip e, non esistendo uno standard universalmente riconosciuto e adottato in questo ambito, ne sono state sviluppate diverse tipologie. Per sistemi di comunicazione semplici potrebbe essere previsto un semplice bus comune a tutte le parti del sistema, mentre per sistemi più complessi risulta necessario realizzare una struttura con più bus collegati da bridge, come mostrato in figura 3.1. Questo tipo di soluzione è ispirata a quella utilizzata a livello di scheda e, in un primo momento, è risultata vantaggiosa grazie all'esperienza maturata dai progettisti in tale contesto. I vantaggi derivati da sistemi basati su bus sono poi: la facilità di utilizzo, il basso costo, la compatibilità con le interfacce proposte dai processori e la bassa latenza che offrono una volta terminata la procedura di risoluzione di eventuali conflitti. La soluzione a bus risulta però inadeguata al crescere del numero di dispositivi di calcolo che hanno la possibilità di accedervi

[34][35], quindi i moderni MPSoC prevedono l'adozione di reti di comunicazione integrate all'interno del chip ("Network-on-Chip", NoC). Infatti [36]:

- in un bus l'aggiunta di un'unità provoca un degrado delle prestazioni e un incremento di area e di consumo di potenza a causa della sua capacità parassita che si somma a quella dei nodi già presenti, mentre in una rete tutte le connessioni sono punto a punto, quindi questo problema non sussiste;
- il bus presenta difficoltà nella gestione delle temporizzazioni, a differenza della rete in cui il protocollo è globalmente asincrono ed è quindi possibile introdurre la pipeline;
- la testabilità di un bus risulta problematica e lenta se confrontata con una metodologia BIST con cui una rete può essere testata;
- le tempistiche di arbitraggio nel bus sono influenzate dal numero di risorse presenti, mentre nella rete le decisioni per il routing sono distribuite e indipendenti;
- il bus è intrinsecamente non scalabile in termini di banda passante, condivisa da tutti i nodi a divisione di tempo. Tale banda risulta poi limitata anche in virtù della frequenza massima cui un bus è in grado di operare, limitazione data soprattutto dalla lunghezza delle interconnessioni. La rete presenta invece una banda che cresce linearmente con il numero di sistemi di cui è composta [37], mantenendo al contempo l'architettura molto più scalabile rispetto ad un'implementazione a bus [38].

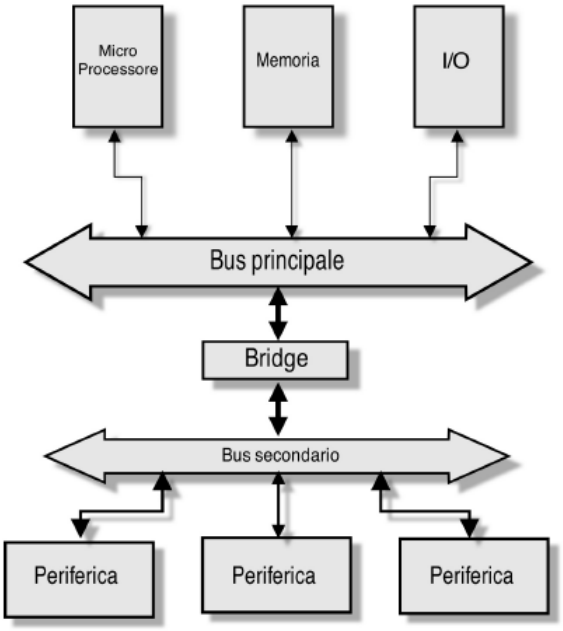

Figura 3.1: Sistema di comunicazione basato su bus

Esiste quindi un punto in cui risulta conveniente abbandonare il bus a favore della rete, al fine di ottenere le migliori prestazioni in termini di latenza. Un confronto tra un circuito integrato che gestisce la comunicazione tramite bus e lo stesso sistema realizzato con un'architettura NoC mostra come l'efficienza in termini di latenza della NoC rispetto al bus cresca all'aumentare del numero di IP cores costituenti il sistema [39].

3.1.3 Network-on-Chip

Secondo quanto esposto in precedenza, l'utilizzo del bus come mezzo di comunicazione tra gli IP si è presto rivelato limitativo con l'avvento di sistemi complessi in cui più flussi di dati devono essere trasmessi in contemporanea. Per ovviare a questo limite ci si è ispirati alle caratteristiche, oltre che agli standard, delle reti di calcolatori, definendo così architetture in cui è possibile poter effettuare più trasferimenti di dati contemporaneamente. Si giunge così all'introduzione del concetto di Network-on-Chip (NoC) [40][41][42], cioè di una micro - rete integrata all'interno del chip flessibile e scalabile con comunicazione a pacchetti, progettata secondo una metodologia a livelli e gestita secondo un opportuno protocollo.

Nella progettazione di una NoC devono essere tenuti in considerazione diversi aspetti:

- la riusabilità dei componenti è fondamentale poiché la possibilità di riutilizzare gli stessi micro-apparati in contesti che variano consente di tenere bassi i costi di produzione e quindi, in un secondo momento, di aumentare l'efficienza e le prestazioni;
- è richiesto un basso consumo poiché minore calore dissipato dai componenti significa maggiore efficienza energetica, tasso di integrazione più alto facilmente raggiungibile e quindi minore latenza;
- la semplicità della topologia di connessione dei componenti è indispensabile per mantenere un alto grado di flessibilità e contenere i costi di produzione e la complessità degli algoritmi di routing. Inoltre, la complessità di un circuito influisce notevolmente sulla massima frequenza utilizzabile, e quindi sulla massima velocità ottenibile dal circuito stesso. Tuttavia, circuiti e topologie semplici difficilmente raggiungono l'efficienza e le funzionalità di componenti più avanzati.

Una NoC può essere organizzata secondo diverse topologie [43]. Ognuno degli approcci proposti comporta un trade-off tra diverse metriche che caratterizzano una NoC [44], tra cui: complessità

della rete e dell'algoritmo di routing, latenza, throughput, dissipazione di energia e occupazione di area. La scelta di una particolare topologia dipende dall'aspetto tecnologico più rilevante nel progetto del particolare MPSoC.

Alcuni esempi di topologie regolari e irregolari sono mostrati rispettivamente nelle figure 3.2 e 3.3. Una struttura regolare, impiegata per connettere core omogenei in un MPSoC, comporta notevoli vantaggi dal punto di vista progettuale, e permette un facile controllo su tutti i parametri elettrici delle linee di comunicazione e sui disturbi che le affliggono. Tuttavia, una simile implementazione potrebbe causare un utilizzo sbilanciato della rete, con alcune sue parti congestionate ed altre sottoutilizzate, quindi occorre adottare un'opportuna politica di instradamento dei messaggi. Una topologia irregolare, adatta a sistemi MPSoC eterogenei, presenta invece notevoli vantaggi in termini di ottimizzazione della congestione del traffico, ma presuppone un iter di progettazione più difficoltoso poiché si presenta la necessità di valutare a priori i nodi di rete maggiormente critici e, in fase di definizione del layout, si perdono i vantaggi derivanti dall'impiego di una struttura regolare.

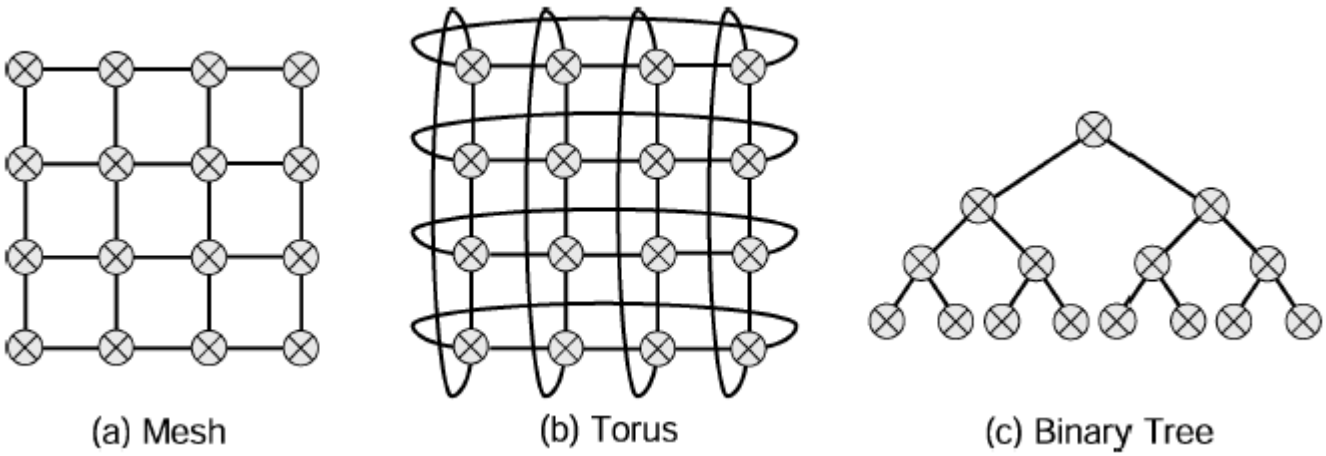

Figura 3.2: Topologie di NoC regolari [43]

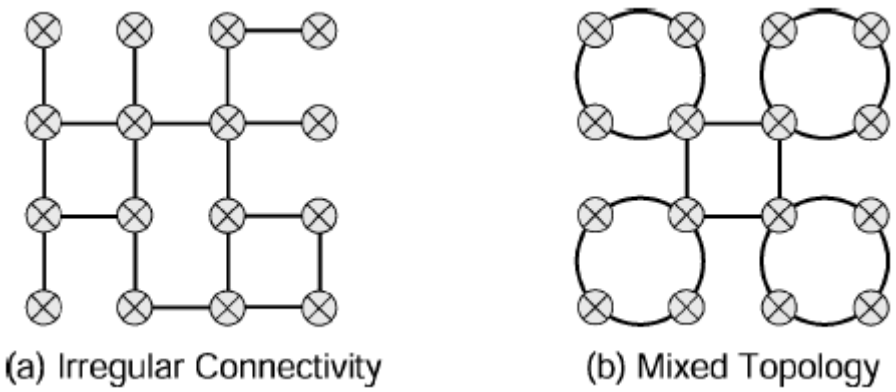

Figura 3.3: Topologie di NoC irregolari [43]

La topologia regolare cui si fa più spesso riferimento, illustrata in figura 3.4, può essere rappresentata da una maglia [45]. Come mostrato in figura 3.5, ad ogni core è associato un router composto da una matrice di interruttori, una logica di controllo, quattro porte globali connesse ai quattro router vicini e una porta locale connessa al core, interfacciato con l'infrastruttura di comunicazione tramite un'interfaccia di rete. Ogni porta presenta un buffer d'ingresso e d'uscita per la memorizzazione dei pacchetti; la dimensione di ciascun buffer dipende dalla specifica implementazione del flusso di controllo [46]. Le quattro porte globali dei router vengono considerate come parti dei link di comunicazione ed escluse dal blocco computazionale, come illustrato in figura 3.6, allo scopo di considerare tutti i blocchi uguali indipendentemente dalla loro posizione nella maglia. Un link di comunicazione è composto da porte di ingresso e uscita e da un bus, costituito da fili di dato, di controllo, di parità ed eventualmente di ricambio.

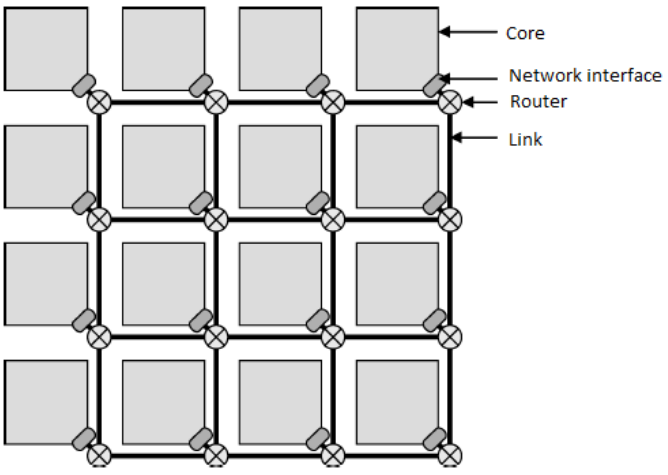

Figura 3.4: Topologia a maglia [43]

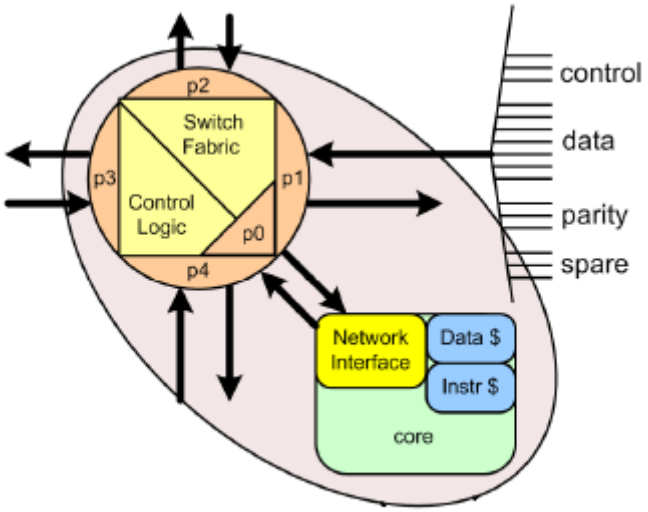

Figura 3.5: Core, router e network interface [45]

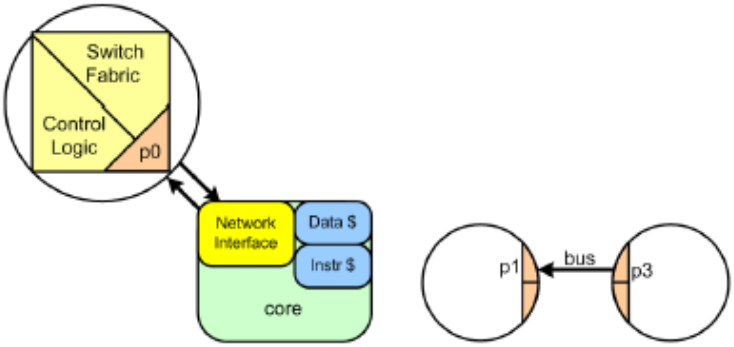

Figura 3.6: Blocco computazionale e link di comunicazione [45]

3.1.3.1 Router

Il router, la cui struttura è rappresentata in figura 3.7, è un componente fondamentale della NoC poiché supporta l'instradamento dei pacchetti da sorgente a destinazione permettendo il parallelismo di comunicazione fra diverse coppie di nodi. Ciascun core è interfacciato con l'infrastruttura di comunicazione tramite un canale interno che implementa l'interfaccia di rete, mentre ogni router è connesso ai quattro router vicini tramite un canale esterno, costituito dal mezzo di comunicazione, da circuiti che gestiscono il flusso dell'informazione ("link controller", LC) e da un certo numero di buffer d'ingresso e d'uscita per la memorizzazione dei flit in cui i pacchetti vengono suddivisi

dall'interfaccia di rete, come avviene nella tipica soluzione di instradamento chiamata "wormhole switching" [47]. Tale soluzione prevede l'instradamento dei flit da sorgente a destinazione tramite percorsi in cui la connessione tra router vicini avviene tramite interruttori opportunamente comandati dall'unità di routing e arbitraggio. Il numero di flit all'interno di un pacchetto deve essere scelto accuratamente perché se fosse eccessivo determinerebbe un'elevata latenza ed un probabile intasamento della rete, mentre un partizionamento limitato renderebbe eccessivo il costo della rete in termini di area. Infatti, i buffer occupano la maggior parte dell'area dei router, essendo la logica di controllo molto semplice [48]. Il flit di testa identifica un percorso, la cui lunghezza è proporzionale al numero di flit che formano il pacchetto, e definisce in corrispondenza un canale virtuale lungo cui vengono poi instradati tutti i flit successivi, fino all'ultimo, sfruttando quindi il meccanismo della pipeline nella comunicazione. I buffer d'ingresso vengono connessi a quelli d'uscita tramite interruttori opportunamente comandati dall'unità di routing e arbitraggio, che implementa il protocollo di routing garantendo il corretto instradamento dei dati verso la giusta destinazione. Se il flit di testa non riuscisse a procedere lungo il cammino per la presenza di un canale d'uscita occupato, l'intera catena di flit andrebbe in stallo, occupando i buffer e quindi bloccando altre possibili comunicazioni lungo il percorso. La latenza di una comunicazione priva di conflitti è proporzionale alla somma della lunghezza del percorso e della dimensione del pacchetto.

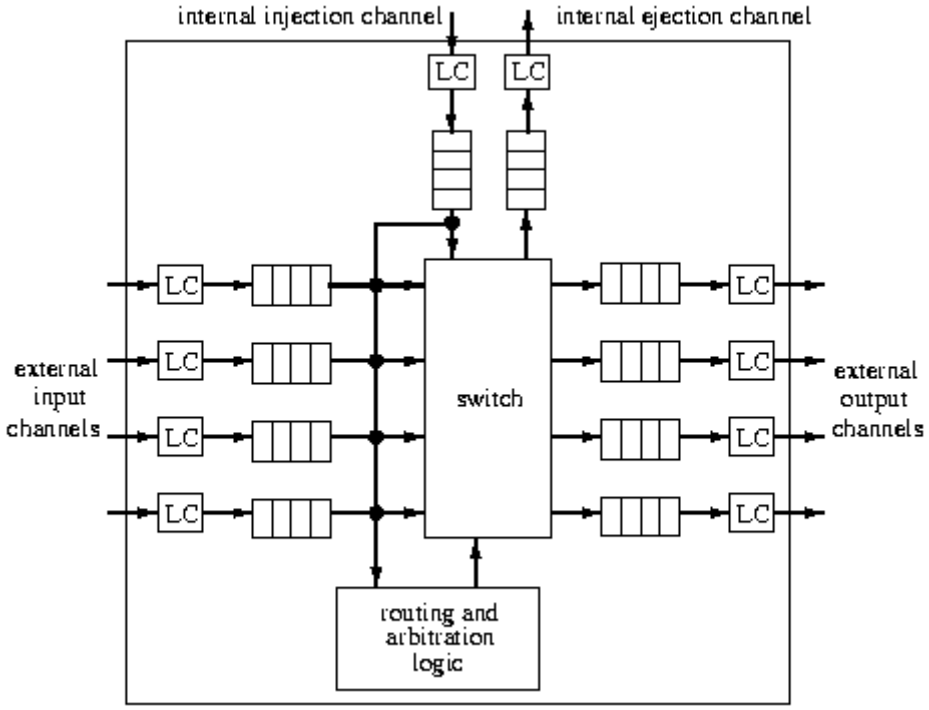

Figura 3.7: Struttura di un router [61]

3.1.3.2 Network interface

Secondo quanto esposto precedentemente, l'architettura MPSoC ha come infrastruttura di comunicazione una NoC su cui si affacciano core di varia natura. Per poter interfacciare tali dispositivi eterogenei con una rete di interconnessione progettata in modo indipendente dai dispositivi stessi si introduce un'interfaccia di rete ("network interface", NI), ossia un blocco hardware che acquisisce l'informazione da un nodo e la spedisce alla rete o viceversa. I compiti fondamentali della NI sono dunque di convertire il protocollo di comunicazione della risorsa in quello della rete e viceversa, e di nascondere i dettagli del protocollo di rete ai nuclei IP, che quindi possono essere sviluppati indipendentemente dall'infrastruttura di comunicazione.

La struttura generica di una NI, mostrata in figura 3.8, comprende l'adattatore OCP/IP, il nucleo e i buffer d'ingresso e d'uscita [49].

L'adattatore OCP/IP implementa il protocollo di comunicazione OCP ("open core protocol") [50], scelto come interfaccia standard per gli IP, che disaccoppia il funzionamento dei core dal processo di comunicazione, garantendo una proprietà intellettuale completamente riutilizzabile ed indipendente dall'architettura in cui viene impiegata. La comunicazione tra due IP prevede che un master impartisca comandi ad uno slave che risponde accettando i dati provenienti dal master o fornendo dati ad esso [51], quindi l'adattatore deve poter implementare sia un'interfaccia master che un'interfaccia slave. Come mostrato in figura 3.9, al master che presenta i comandi ed eventualmente i dati corrisponde un'interfaccia slave che converte la richiesta in un messaggio da trasmettere sulla rete, mentre un'interfaccia che converte i messaggi in comandi viene associata allo slave che, dopo aver ricevuto le istruzioni, svolge le operazioni richieste.

Il nucleo, che riceve dall'adattatore o gli trasmette dati e informazioni di controllo, svolge diversi compiti: suddividere i messaggi da inviare in pacchetti o comporre un messaggio con i pacchetti ricevuti, programmare ed effettuare l'inserimento dei pacchetti nei buffer d'uscita, estrarre i pacchetti dai buffer d'ingresso e implementare il meccanismo del flusso di controllo. Il nucleo contiene una look-up table (LUT) per la memorizzazione delle informazioni di routing da inserire nei pacchetti che vengono trasmessi sulla rete. Solitamente, infatti, le NoC implementano la soluzione di instradamento chiamata "wormhole switching" precedentemente richiamata, in cui l'informazione sul percorso da seguire per raggiungere il nodo destinazione viene fornita ai pacchetti sotto forma di una sequenza di bit che permette di selezionare le porte d'uscita degli interruttori incontrati. La LUT presenta un numero di celle di memoria per la memorizzazione dei

percorsi di routing che cresce con il numero di nodi, ed è programmabile, ossia l'informazione memorizzata può essere riscritta, per poter eventualmente modificare i percorsi di routing, evitando quindi i link o i router guasti rilevati lungo un determinato percorso.

I buffer d'ingresso e d'uscita, che memorizzano l'informazione tramite l'utilizzo di code (FIFO), immagazzinano rispettivamente i pacchetti che giungono al ricevitore e i pacchetti che il trasmettitore intende inviare alla NoC. Oltre agli elementi di memoria, è necessaria una logica di controllo per muovere i puntatori di lettura e di scrittura, implementati tramite contatori, che selezionano rispettivamente il prossimo elemento da estrarre e la prima posizione libera nella FIFO. Un'ulteriore logica implementa i segnali di controllo della FIFO, che indicano se la struttura è piena o vuota e quindi se sia possibile o meno eseguire operazioni di scrittura o lettura.

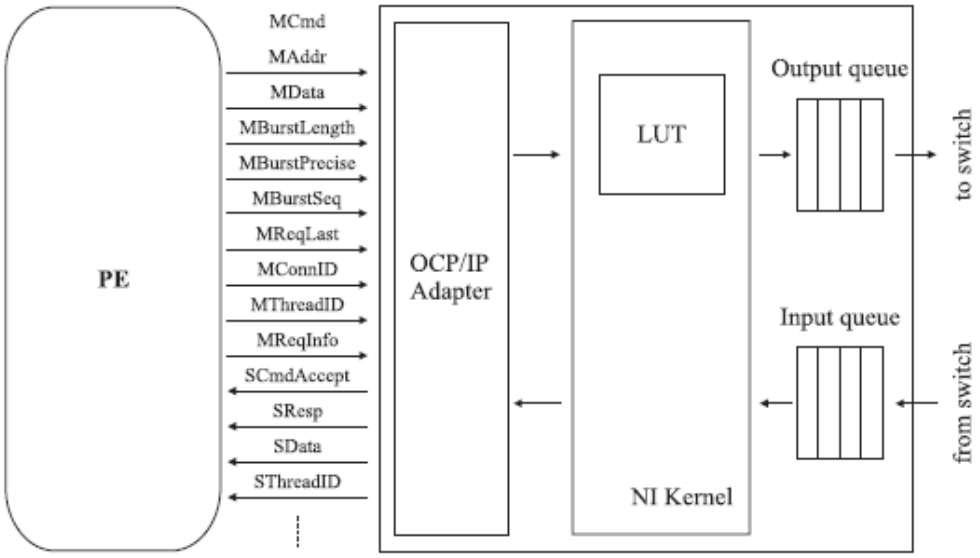

Figura 3.8: Struttura di una generica NI [49]

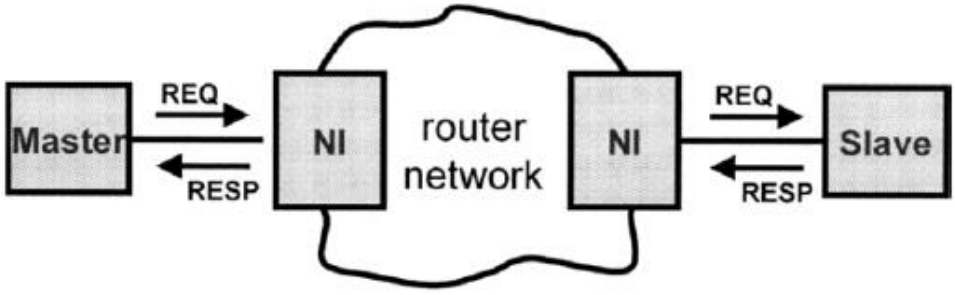

Figura 3.9: Esempio di comunicazione tra blocchi IP [51]

3.2 Architettura di nodo

In questo paragrafo si analizza l'architettura di nodo presentando le specifiche del processore SecretBlaze e del bus Wishbone. Dopo aver illustrato nel dettaglio la struttura e le prestazioni del processore SecretBlaze, si descrivono le principali caratteristiche del bus Wishbone, i segnali utilizzati per connettere i dispositivi master a quelli slave e i protocolli di lettura e scrittura.

3.2.1 Processore SecretBlaze

Secondo quanto esposto nel paragrafo precedente, negli ultimi anni si è sviluppata la tendenza di sostituire un numero crescente di parti del sistema, progettate specificatamente per una particolare applicazione, con architetture programmabili, basandosi sul riutilizzo di unità pre - progettate e generalmente già testate dette IP core. In questo contesto sono stati sviluppati i processori soft-core [55], il cui utilizzo si è notevolmente diffuso nell'ultimo decennio. Questi microprocessori, la cui architettura e il cui funzionamento sono completamente descritti ad un elevato livello di astrazione tramite un linguaggio di descrizione dell'hardware ("hardware description language", HDL), possono essere sintetizzati per svariate tecnologie, tra cui l'FPGA ("field programmable gate array"). La loro flessibilità consente ai progettisti di adattare facilmente le loro strutture ad applicazioni specifiche.

Il SecretBlaze [56][57] è un processore soft-core a 32 bit altamente configurabile e open - source, che implementa il set di istruzioni del processore MicroBlaze. Il suo sviluppo è stato condotto tramite un approccio modulare per fornire un'implementazione efficiente e garantire ottime opportunità di riutilizzo dell'IP core in diversi progetti di ricerca.

Questo processore RISC ("reduced instruction set computer") implementa un'architettura Harvard che sfrutta il parallelismo a livello di istruzione impiegando una pipeline a cinque stadi, descritti in entità VHDL distinte ed interconnessi come mostrato in figura 3.10, dove viene rappresentata la struttura generale del data path. Nelle fasi di fetch (IF), decode (ID) ed execute (EX) l'istruzione viene rispettivamente letta dalla cache, decodificata ed eseguita, mentre le fasi successive prevedono l'accesso alla memoria da parte delle istruzioni load e store (MA) e la scrittura dei risultati nel registro destinazione (WB). Grazie all'impiego della pipeline, la maggior parte delle

istruzioni richiede un ciclo di clock per l'esecuzione, quindi si ottengono elevate prestazioni a basso costo. Tuttavia, la presenza nella pipeline di più istruzioni simultaneamente attive può portare a vari tipi di conflitti, che possono essere causati da istruzioni che mostrano una dipendenza sui dati o che modificano il flusso di controllo. Nella struttura proposta, la tecnica di predizione delle diramazioni si basa sulla predizione che il salto non sia mai effettuato, mentre i conflitti di dati vengono completamente risolti impiegando la tecnica di data forwarding. Viene quindi implementato un circuito di controllo, che fornisce segnali di stallo e di flush sincroni a determinati stadi della pipeline.

Il core del processore è altamente configurabile e può essere adattato per soddisfare i vari requisiti delle applicazioni. Il data path supporta infatti numerose istruzioni, tra cui lo shift, il confronto tra vettori, la moltiplicazione e la divisione intere. La flessibilità del set di istruzioni è un aspetto fondamentale del progetto del SecretBlaze poiché consente di ottimizzare le prestazioni computazionali e il costo in termini di area, e quindi di soddisfare i requisiti dei sistemi integrati.

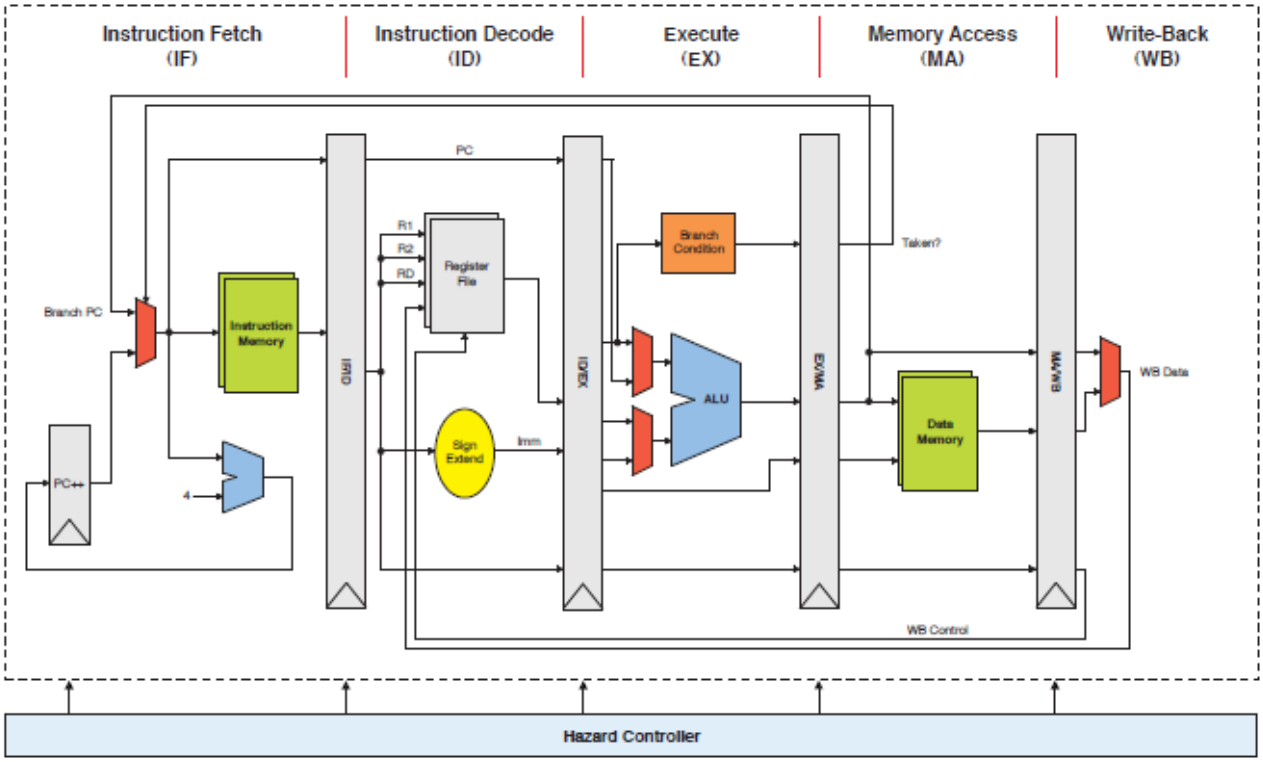

Figura 3.10: Struttura generale del data path del SecretBlaze [57]

Il sottosistema di memoria, che svolge un ruolo chiave nelle prestazioni globali, si occupa di gestire gli accessi a memoria e i dispositivi di I/O richiesti dal processore. Il SecretBlaze presenta bus separati per istruzioni e dati, ed impiega indirizzi a 32 bit fornendo uno spazio di indirizzamento di 4 gigabyte.

La struttura del sottosistema di memoria, mostrata in figura 3.11, è composta dalle memorie locali, dalle cache e dalle interfacce per memorie esterne.

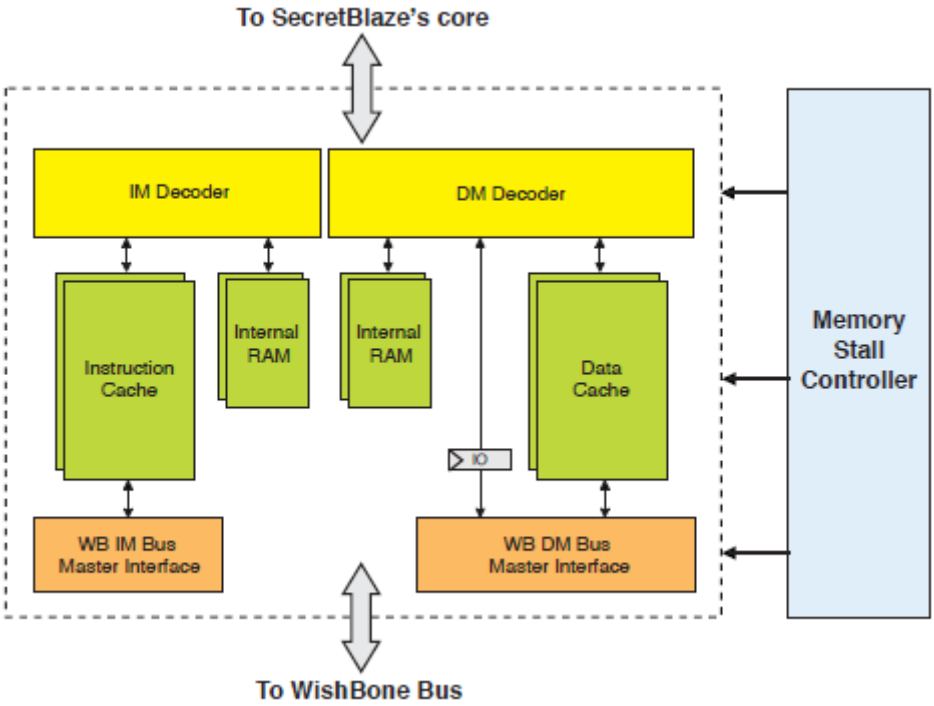

Figura 3.11: Struttura del sottosistema di memoria del SecretBlaze [57]

Nelle memorie locali, che operano alla frequenza del processore, possono essere caricati numerosi programmi semplici. L'implementazione avviene tramite moduli di memoria RAM con due porte di accesso.

Per migliorare le prestazioni, il SecretBlaze è dotato di una cache istruzioni ed una cache dati di primo livello separate, che vengono descritte come macchine a stati finiti con segnali di controllo distinti. La dimensione della memoria fisica e quella di un blocco di cache sono configurabili sia per la cache istruzioni che per la cache dati, che può adottare una politica write-through o write-back. Anche in questo caso, l'implementazione avviene tramite moduli di memoria RAM con due porte di accesso per consentire operazioni di lettura e di scrittura simultanee.

L'ultima componente chiave del sottosistema di memoria è costituita dalle interfacce per memorie esterne efficienti, che consentano la connessione di periferiche addizionali. Il bus Wishbone, scelto principalmente per la flessibilità e la facilità di utilizzo, supporta la modalità pipeline per lo scambio dei dati, secondo cui un'interfaccia master non deve attendere il segnale di acknowledge prima di porre sul bus l'indirizzo e il dato successivi, in modo tale da aumentare il throughput dei moderni circuiti integrati di memoria. Il SecretBlaze presenta due interfacce master poiché sono presenti bus separati per istruzioni e dati.

E' infine presente un circuito di controllo che ha il compito di fornire segnali di stallo quando un dato non è disponibile nella cache, ossia si verifica un cache miss, o quando un'operazione di I/O non è terminata.

La tabella 3.2 indica la massima frequenza operativa e l'utilizzo di risorse, espresso in termini di numero di flip-flop, LUT a quattro ingressi e blocchi di memoria RAM, che si ottengono implementando il SecretBlaze con una tecnologia a 90 nm.

# of Flip-Flops	638
# of LUTs used as logic	1179
# of LUTs used as RAMs	384
# of BRAMs	8
fMAX in MHz	90.9

Tabella 3.2: Utilizzo di risorse e massima frequenza operativa [57]

Le prestazioni del SecretBlaze sono confrontabili con quelle del MicroBlaze e, come mostrato in figura 3.12, superiori a quelle di altri due processori open - source, l'OpenRISC 1200 e l'MB-lite, che implementa anch'esso il set di istruzioni del MicroBlaze.

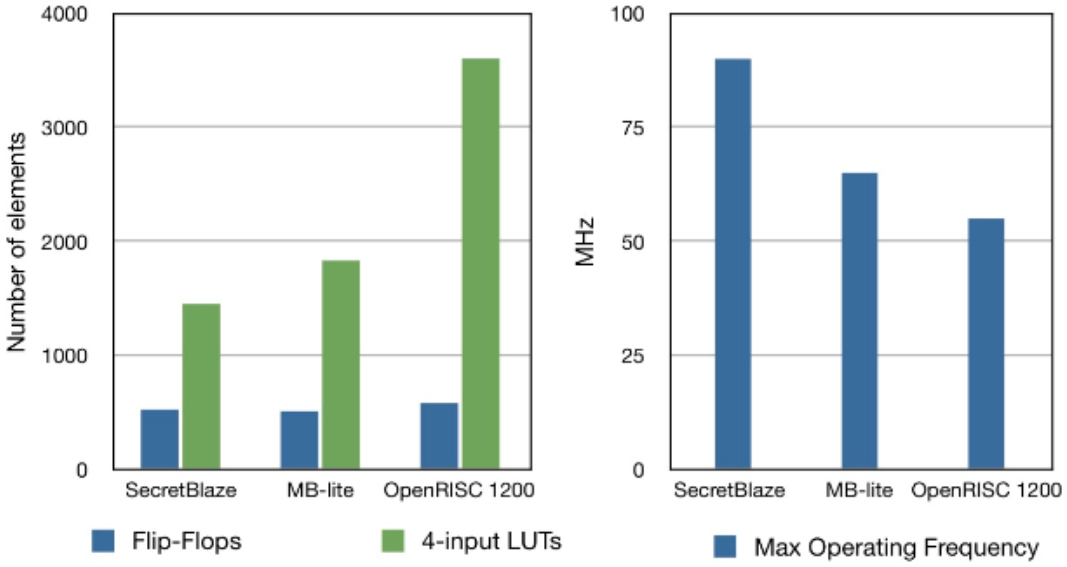

Figura 3.12: Confronto tra le prestazioni di tre processori [62]

3.2.2 Bus Wishbone

Il bus Wishbone [58] è un canale di comunicazione flessibile e open - source che consente lo scambio di dati tra diversi componenti all'interno di un circuito integrato. Uno dei principali obiettivi dei progettisti è la creazione di un'interfaccia comune per gli IP core che sia indipendente dalla tecnologia impiegata, in modo tale da consentire il riutilizzo di queste unità. In questo modo vengono migliorate la portabilità e l'affidabilità del sistema, e viene ridotto il time to market per l'utente finale, che può integrare gli IP core facilmente e velocemente, mentre in precedenza veniva utilizzato uno schema di interconnessione non standard che rendeva difficile tale integrazione. I progettisti hanno quindi cercato di creare una specifica abbastanza robusta da assicurare una completa compatibilità tra gli IP core, ma non eccessivamente dettagliata per non vincolare troppo la creatività dello sviluppatore del core e dell'utente finale.

3.2.2.1 Caratteristiche del Wishbone

Vengono ora riportate le principali caratteristiche del bus Wishbone, che definisce un protocollo standard per lo scambio dei dati tra i vari moduli.

- Vengono supportati metodi di progettazione strutturata utilizzati dai team di progetto. In questo modo ogni membro del team può costruire e testare piccole parti del sistema riferendosi alle specifiche del Wishbone, per poi passare all'integrazione del sistema completo una volta che tutte le unità sono pronte.
- La dimensione del bus dati e degli operandi non è fissa e può arrivare a 64 bit.
- Il sistema è in grado di supportare numerosi dispositivi master e slave con un efficiente meccanismo di arbitraggio.
- Vengono supportati vari metodi di interconnessione tra gli IP core in modo tale da consentire all'utente finale di adattare il SoC alle sue esigenze, tra cui:
    - punto a punto, in cui un master è connesso ad un singolo slave;
    - bus condiviso, in cui sono necessari la decodifica degli indirizzi e la risoluzione dei conflitti per garantire il corretto funzionamento della connessione, rappresentata nell'esempio di figura 3.13;
    - crossbar switch, che consente la comunicazione simultanea tra più coppie di dispositivi master e slave tramite una rete di interconnessione i cui interruttori vengono opportunamente comandati per connettere determinati componenti, come mostrato in figura 3.14;
    - data flow, particolarmente adatto ad un'implementazione basata sull'impiego della pipeline, in cui diversi IP core operano in cascata sullo stesso flusso di dati, come illustrato in figura 3.15.
- Il protocollo sincrono prevede che tutte le transazioni sul bus siano coordinate da un unico segnale di clock, e garantisce la portabilità, l'affidabilità e la facilità di utilizzo del sistema. Il meccanismo di handshake consente ai dispositivi master e slave di regolare la propria velocità di trasferimento dei dati.
- Il meccanismo di timing è variabile per consentire di regolare la frequenza di clock, e quindi di contenere il consumo di potenza del circuito integrato.

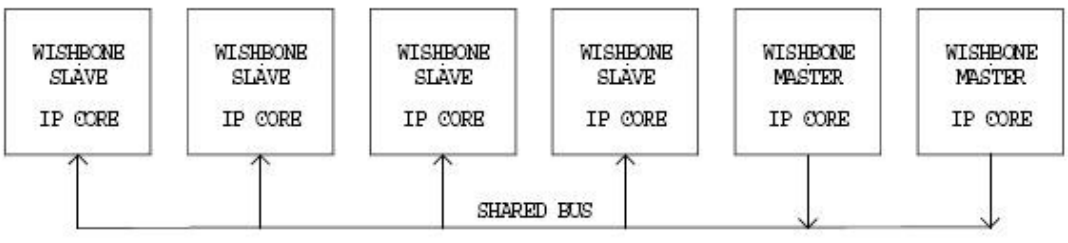

Figura 3.13: Esempio di rete a bus condiviso

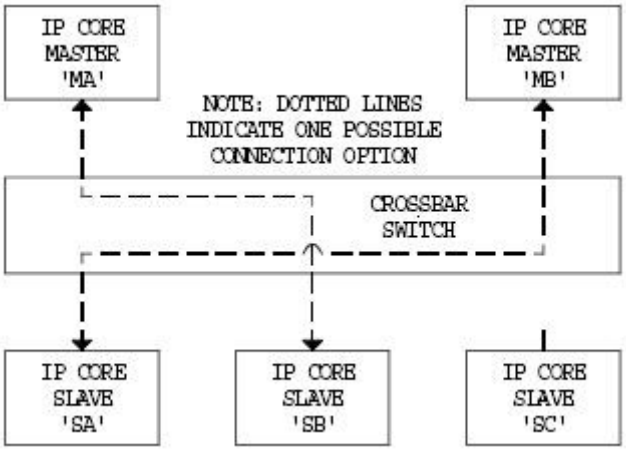

Figura 3.14: Rete di connessione crossbar

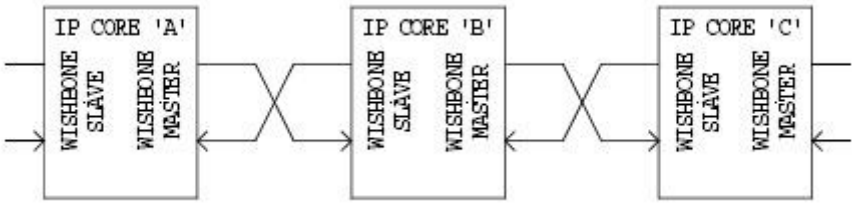

Figura 3.15: Metodo di interconnessione data flow

3.2.2.2 Segnali del Wishbone

In figura 3.16 è rappresentato un possibile schema di connessione tra un dispositivo master e un dispositivo slave, i cui segnali sono identici ma hanno direzione opposta, ad eccezione di quelli di clock e reset, che vengono generati da un modulo e forniti agli ingressi di entrambi i dispositivi.

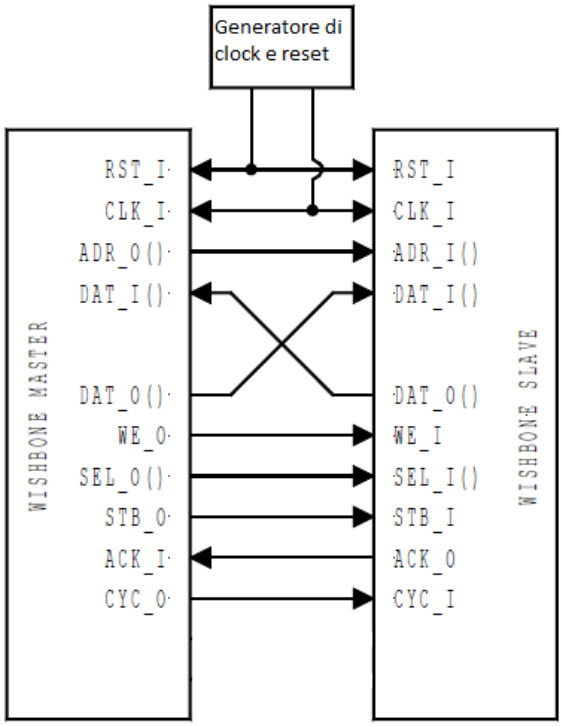

Figura 3.16: Possibile schema di connessione tra un dispositivo master e un dispositivo slave [58]

Di seguito viene riportata la descrizione di ciascun segnale:

- RST: tale segnale, quando alto, riporta tutte le macchine a stati presenti all'interno dei dispositivi nel loro stato iniziale;
- CLK: tutti i segnali sono campionati sul fronte di salita del clock, che coordina tutte le transazioni sul bus;
- ADR: il master fornisce l'indirizzo binario del registro dello slave cui vuole accedere in lettura o in scrittura;
- DAT: durante un'operazione di scrittura il master pone sul bus il dato binario che vuole scrivere in un determinato registro dello slave, mentre durante un'operazione di lettura lo slave pone sul bus il dato binario richiesto dal master, quindi il bus dati è bidirezionale;
- WE: il master asserisce questo segnale per indicare un ciclo di scrittura, mentre lo nega per indicare un ciclo di lettura;
- SEL: durante le operazioni di lettura e di scrittura, tale segnale indica quali byte del dato trasferito sono validi, quindi, essendo il bus dati a 32 bit, è formato da 4 bit;
- STB: tale segnale, quando alto, indica che i segnali di controllo, l'indirizzo e l'eventuale dato emessi dal master sono stabili;

- ACK: tale segnale viene emesso dallo slave in risposta alla ricezione di un'informazione completa;
- CYC: il master asserisce questo segnale per indicare l'inizio di un ciclo di lettura o di scrittura, e lo nega per indicarne la fine.

3.2.2.3 Protocolli di comunicazione

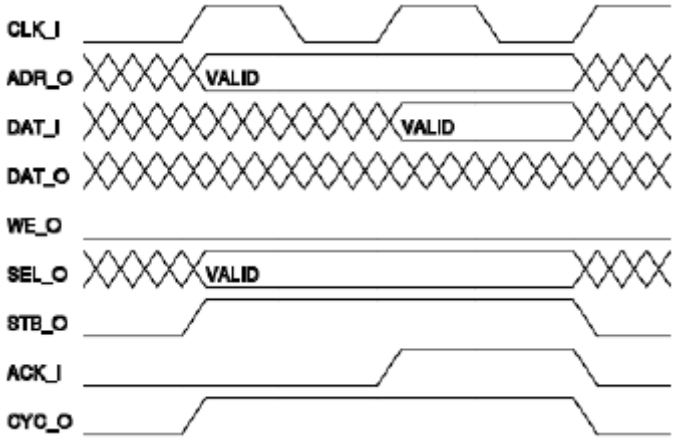

Figura 3.17: Ciclo di lettura [58]

In figura 3.17 è riportato l'andamento delle forme d'onda dei segnali durante un ciclo di lettura, che si svolge come segue:

- sul primo fronte di salita del clock il master emette un indirizzo valido, indica quali byte del dato letto sono validi tramite il segnale SEL, nega il segnale WE ed asserisce i segnali CYC e STB;
- sul secondo fronte di salita del clock lo slave indirizzato pone sul bus il dato richiesto ed asserisce il segnale ACK;
- sul terzo fronte di salita del clock il master legge il dato e nega i segnali CYC e STB. Di conseguenza, lo slave nega il segnale ACK.

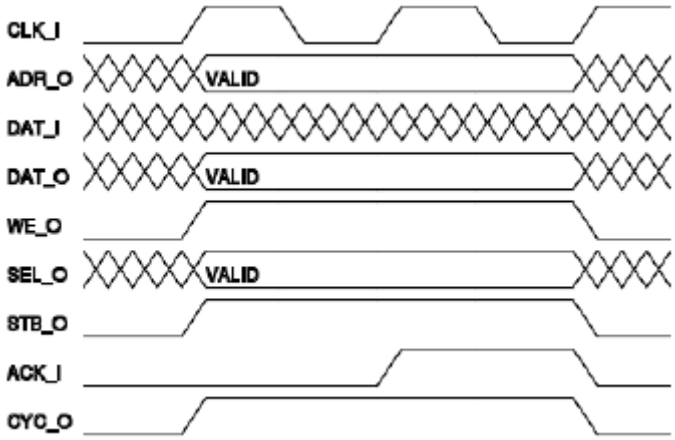

Figura 3.18: Ciclo di scrittura [58]

In figura 3.18 è riportato l'andamento delle forme d'onda dei segnali durante un ciclo di scrittura, che si svolge come segue:

- sul primo fronte di salita del clock il master emette un indirizzo valido e un dato, indica quali byte del dato d'uscita sono validi tramite il segnale SEL, ed asserisce i segnali WE, CYC e STB;
- sul secondo fronte di salita del clock lo slave indirizzato legge il dato e asserisce il segnale ACK;
- sul terzo fronte di salita del clock il master nega i segnali WE, CYC e STB. Di conseguenza, lo slave nega il segnale ACK.

3.3 Trasferimento dei programmi di test al nodo e dei risultati del test alla rete

In questo paragrafo si analizzano i compiti del tester, interfaccia hardware semplice e robusta precedentemente sviluppata per supportare le tecniche software, e la sua posizione all'interno dell'architettura MPSoC, in cui tutti i processori devono essere interfacciati opportunamente per poter essere sottoposti a test.

3.3.1 Funzioni del tester

Secondo quanto esposto nel capitolo precedente, per testare i sistemi non convenzionali come gli MPSoC eterogenei è necessario impiegare le tecniche SBST, che rappresentano un'alternativa economica ed efficiente alle soluzioni hardware, poiché consentono di ottenere un'elevata copertura di guasto col minimo impatto sulle prestazioni, l'area e il consumo di potenza del sistema. Un software esterno genera routine di test che vengono poi salvate in memoria, programmate dal sistema operativo ed applicate al processore che le esegue alla sua velocità nominale. E' necessario integrare sul chip un supporto hardware, chiamato tester, che verifichi i risultati del test in uscita dal processore e, nel caso in cui si riscontri una discrepanza tra i risultati effettivi e quelli attesi, attivi un'ulteriore logica, che assegna i compiti del processore guasto ai processori funzionanti e li sposta fisicamente, sulla base del carico di lavoro e delle prestazioni. La prima istruzione di una routine di test consente quindi di attivare il tester affinché possa ricevere i dati in uscita dal processore, che vengono opportunamente compressi in modo tale da ottenere una signature che viene poi confrontata con quella corretta per verificare l'eventuale presenza di errori nel dispositivo. Ciò avviene per due motivi fondamentali:

- lo spazio di memoria necessario per salvare tutti i risultati corretti delle routine di test è eccessivo;
- la logica richiesta per il controllo di ogni singolo risultato determina una notevole occupazione di area.

L'oggetto della trattazione sono gli MPSoC eterogenei, quindi è richiesto il minimo incremento di occupazione di area e di consumo di potenza, poiché si tratta di sistemi integrati molto costosi. Il circuito scelto per il calcolo della signature impiega dunque un meccanismo molto semplice per comprimere i risultati del test, ed è ampiamente utilizzato nelle telecomunicazioni per controllare l'integrità dei dati. Il suo funzionamento, che sarà illustrato nel capitolo successivo, prevede che il calcolo della signature inizi non appena il primo risultato viene scritto nella memoria del tester, in modo tale da ridurre i tempi delle operazioni di test, mentre in un'implementazione standard del meccanismo di test occorre attendere che il tester abbia ricevuto tutti i risultati prima di effettuarne la compressione.

3.3.2 Inserimento del tester nell'architettura MPSoC

In un MPSoC sono presenti numerosi processori con le loro memorie, ciascuno interfacciato con l'infrastruttura di comunicazione tramite un'interfaccia di rete, e connesso alle rispettive periferiche tramite un bus. I processori devono poter essere sottoposti a test indipendentemente l'uno dall'altro, quindi ad ognuno di essi è associato un tester la cui implementazione non dipende dal particolare core, a differenza della generazione delle routine di test. Tali routine vengono programmate dal sistema operativo in modo tale che il test venga effettuato in particolari momenti come i periodi di inattività, ed eseguite dal processore, che invia i risultati al tester e poi riprende l'attività precedente, senza che il suo normale funzionamento ne venga interessato. Il tester viene quindi mappato sul bus in modo tale da essere direttamente indirizzabile dal processore, e la sua implementazione dipende fortemente dal tipo di bus utilizzato poiché le periferiche, che fungono da slave, necessitano di un'interfaccia per comunicare col bus, e quindi col processore. A seconda delle necessità, il processore può inviare comandi e dati direttamente al tester, senza coinvolgere le altre periferiche e la memoria principale. Nel tester sono quindi presenti due spazi di memoria, la cui struttura sarà illustrata dettagliatamente nel capitolo successivo: uno è composto da registri cui vengono inviati i comandi, l'altro è una memoria in cui vengono salvati i risultati delle routine di test.

Capitolo 4

Lavoro

4.1 Struttura dell'interfaccia

In questo paragrafo, dopo aver elencato e descritto dettagliatamente i principali componenti del tester, vengono illustrati la struttura ed il funzionamento dell'interfaccia che verifica i risultati del test in uscita dal processore.

4.1.1 Componenti del tester

Il tester è composto fondamentalmente da:

- due spazi di memoria, uno dei quali è composto da due registri, il cui scopo è ricevere e riconoscere i comandi e i dati inviati dal processore, che saranno descritti in seguito, mentre l'altro è una memoria in cui vengono salvati i risultati delle routine di test in uscita dal processore, che vengono poi letti e compressi in modo tale da ottenere una signature;
- due macchine a stati finiti ("finite state machine", FSM), che gestiscono la lettura dei risultati scritti nella memoria del tester, il loro invio al circuito che ne effettua la compressione, ed il confronto tra la signature effettiva e quella attesa;
- un registro a scorrimento a 32 bit, che consente di serializzare i risultati del test in modo tale che all'ingresso del dispositivo per il calcolo della signature venga fornito un bit per ogni ciclo di clock;
- il dispositivo per il calcolo della signature, che viene opportunamente attivato affinché possa comprimere i risultati del test.

Nel tester sono presenti due spazi di memoria, composti rispettivamente da due registri e una memoria FIFO, che vengono opportunamente mappati sul bus indirizzi in modo tale da essere direttamente accessibili dal processore. L'indirizzo base del registro 0 è 0x60000000, il cui incremento di 4 e di 8 fornisce rispettivamente gli indirizzi base del registro 1 e della memoria.

Il processore può inviare al registro 0 due possibili comandi tramite la scrittura delle stringhe di bit 0xFFFF0000 e 0xFFFFFFFF, per indicare che sta per iniziare o ha terminato l'invio dei risultati del test all'interfaccia. In questo modo il tester viene attivato affinché possa ricevere i risultati, effettuarne la compressione e, una volta che il processore ha terminato l'invio dei dati, confrontare la signature effettiva con quella attesa.

Nei 16 bit meno significativi del registro 1 il processore indica il numero di parole da 32 bit che devono essere scritte nella memoria del tester, mentre in quelli più significativi pone la signature corretta, che deve essere confrontata con quella ottenuta dalla compressione dei risultati del test.

Nella memoria FIFO, utilizzata essenzialmente per il salvataggio dei dati provenienti dal processore, possono essere memorizzate fino a 512 parole da 32 bit. La quantità di memoria può essere aumentata o diminuita secondo le necessità, ma dalla letteratura in materia si evince che 2 kilobyte di memoria dovrebbero essere sufficienti per contenere tutti i risultati di un tipico programma di test. Oltre agli elementi di memoria, è necessaria una logica di controllo per muovere i puntatori di lettura e di scrittura, implementati tramite contatori, che selezionano rispettivamente il prossimo elemento da estrarre e la prima posizione libera nella FIFO. Un'ulteriore logica implementa i segnali di controllo della FIFO, che indicano se la struttura è piena o vuota e quindi se sia possibile o meno eseguire operazioni di scrittura o lettura. La FIFO può essere configurata con domini di clock comuni o indipendenti per le due diverse operazioni, che in questa particolare implementazione sono sincronizzate da un unico segnale di clock.

La prima macchina a stati finiti, detta FSM di controllo, si trova inizialmente nello stato IDLE, come mostrato nel diagramma di figura 4.1. Una volta che la logica ha letto il comando di avvio dal registro 0, viene generato il segnale *start_process* e la FSM passa allo stato WAIT_END, in cui i segnali *error* e *check_done* vengono riportati a livello logico basso, mentre il segnale *start* viene asserito per abilitare la seconda macchina a stati. Quando il comando di fine viene letto dal registro 0, viene generato il segnale *end_process* e la FSM entra nello stato READ_RAM, in cui rimane finché il segnale *stop* non viene asserito dalla seconda macchina a stati per indicare che tutti i dati presenti nella memoria del tester sono stati letti e che la signature è stata calcolata. A questo punto il

segnale start viene negato e la FSM può passare allo stato CHECK_RES, in cui la signature effettiva viene confrontata con quella attesa, che viene letta dal registro 1. Se l'esito del confronto è un'uguaglianza il segnale error rimane basso, altrimenti viene portato a livello logico alto per segnalare la presenza di un errore nel dispositivo. Il segnale check_done viene asserito per indicare che il controllo è terminato, e la FSM entra nello stato FINISH, in cui rimane finché la lettura di un nuovo comando di avvio dal registro 0 non fa iniziare un altro ciclo.

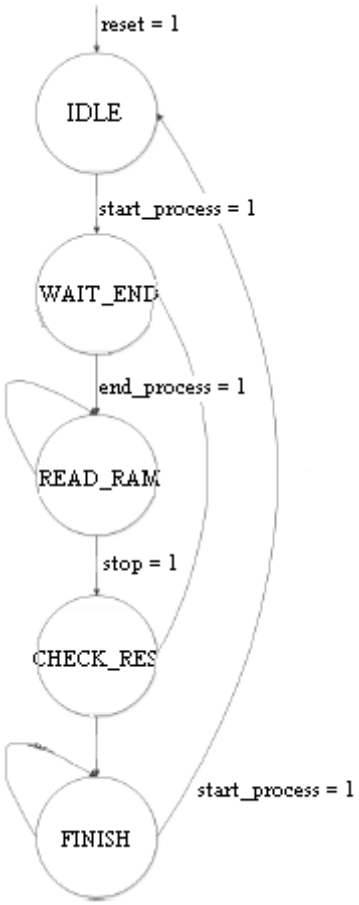

Figura 4.1: Diagramma degli stati della FSM di controllo

Come illustrato nel diagramma di figura 4.2, la seconda macchina a stati finiti, detta FSM di lettura, si trova inizialmente nello stato IDLE, quindi passa allo stato READ_D quando la FSM di controllo asserisce il segnale start. A questo punto i segnali stop e *counter* vengono riportati a livello logico basso, mentre il segnale *load* viene asserito per indicare che il primo dato a 32 bit è stato letto dalla memoria e caricato nel registro a scorrimento. La FSM entra quindi nello stato SERIALIZE, in cui il segnale load viene negato, mentre il segnale *shift* viene asserito affinché i bit contenuti nel

registro a scorrimento vengano traslati e inviati al circuito per il calcolo della signature. Il contatore viene incrementato ad ogni ciclo di clock e, quando raggiunge il valore 31, indica che tutti i bit nel registro sono stati traslati, quindi il segnale *counter_flag* viene asserito e la FSM passa allo stato FINISH, in cui il segnale shift, che ha anche la funzione di abilitare il circuito per il calcolo della signature, viene negato. Se il segnale end_process è basso la FSM torna allo stato READ_D poiché non tutte le parole sono state lette dalla memoria, altrimenti asserisce il segnale stop ed entra nello stato DUMMY, in cui rimane finché l'altra FSM non ha asserito il segnale check_done per indicare che il confronto tra la signature effettiva e quella attesa è terminato. A questo punto la FSM torna allo stato IDLE ed è possibile iniziare un nuovo ciclo.

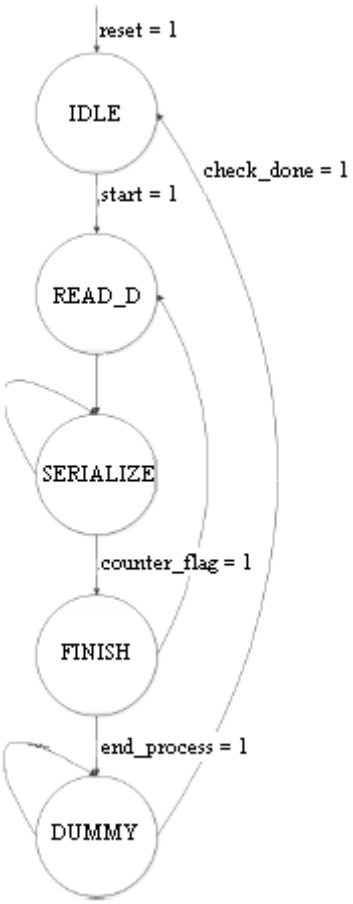

Figura 4.2: Diagramma degli stati della FSM di lettura

Il registro a scorrimento, la cui struttura è mostrata in figura 4.3, consente il caricamento dei dati in forma parallela. Infatti, i 32 bit della parola letta dalla memoria del tester vengono applicati ai terminali di ingresso dati del registro e, quando il segnale load è alto, vengono memorizzati dai flip-

flop alla prima transizione attiva del segnale di clock. La FSM di lettura entra quindi nello stato SERIALIZE, in cui il segnale load viene negato, mentre il segnale shift viene asserito affinché avvenga la traslazione dei bit contenuti nel registro, che vengono prelevati in forma seriale sull'uscita dell'ultima cella di memoria ed inviati al circuito per il calcolo della signature, che ne riceve uno ad ogni ciclo di clock. Quando il contatore raggiunge il valore 31 l'operazione termina, ed è quindi possibile leggere un nuovo dato dalla memoria del tester.

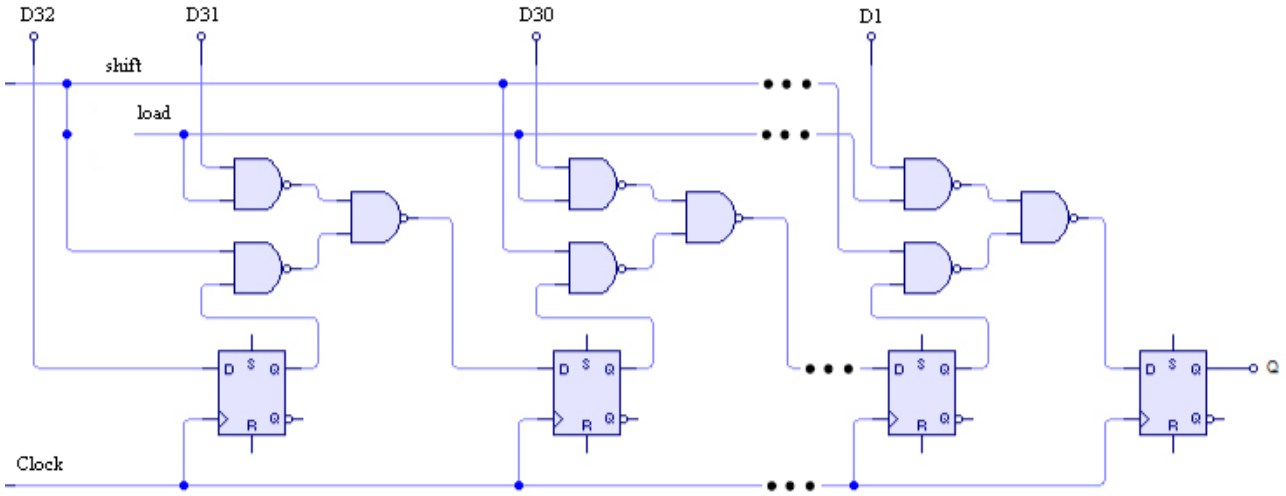

Figura 4.3: Struttura del registro a scorrimento

L'analisi della signature, centrata sulla compressione dei risultati del test, si basa sull'uso di registri a scorrimento a retroazione lineare ("linear feedback shift register", LFSR), costituiti da flip-flop di tipo D e porte logiche XOR, e sul concetto di controllo a ridondanza ciclica ("cyclic redundancy check", CRC), secondo cui i dati d'uscita sono ottenuti elaborando i dati d'ingresso, che vengono fatti scorrere ciclicamente in una rete logica. L'algebra in cui si opera è quella dei campi di Galois binari, quindi variabili e costanti possono assumere solo i valori 0 e 1, e le operazioni sono tutte in modulo 2, dunque non esistono riporti. Il circuito per il calcolo della signature, la cui struttura è rappresentata in figura 4.4, riceve i risultati del test in forma seriale e li elabora quando il segnale shift abilita i flip-flop, le cui uscite sono tutte a livello logico alto quando la macchina si trova nello stato iniziale. Ad ogni ciclo di clock alcuni bit influenzano lo stato successivo tramite le funzioni logiche XOR, mentre gli altri vengono semplicemente traslati. Una volta che la FSM di lettura ha asserito il segnale stop per indicare che tutti i risultati del test sono stati inviati al registro, l'uscita a 16 bit rappresenta la signature effettiva, che viene confrontata con quella attesa. Il circuito

impiegato è molto semplice ed è composto solo da 16 flip-flop e 3 porte logiche XOR a due ingressi, ma il tempo necessario per il calcolo della signature è lungo poiché i dati vengono forniti al registro in forma seriale, quindi, se nella memoria del tester fossero memorizzate 512 parole da 32 bit, sarebbero necessari 16384 cicli di clock per completare il processo. Nelle simulazioni svolte la logica opera a 200 MHz, quindi nel caso peggiore occorrono 81.92 µs per terminare il calcolo della signature.

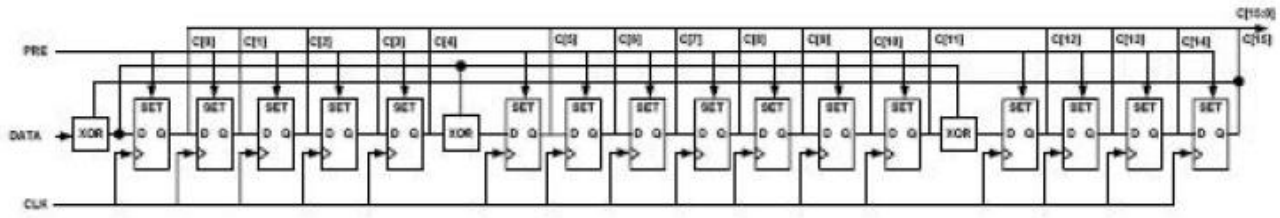

Figura 4.4: Struttura del circuito per il calcolo della signature

4.1.2 Funzionamento del tester

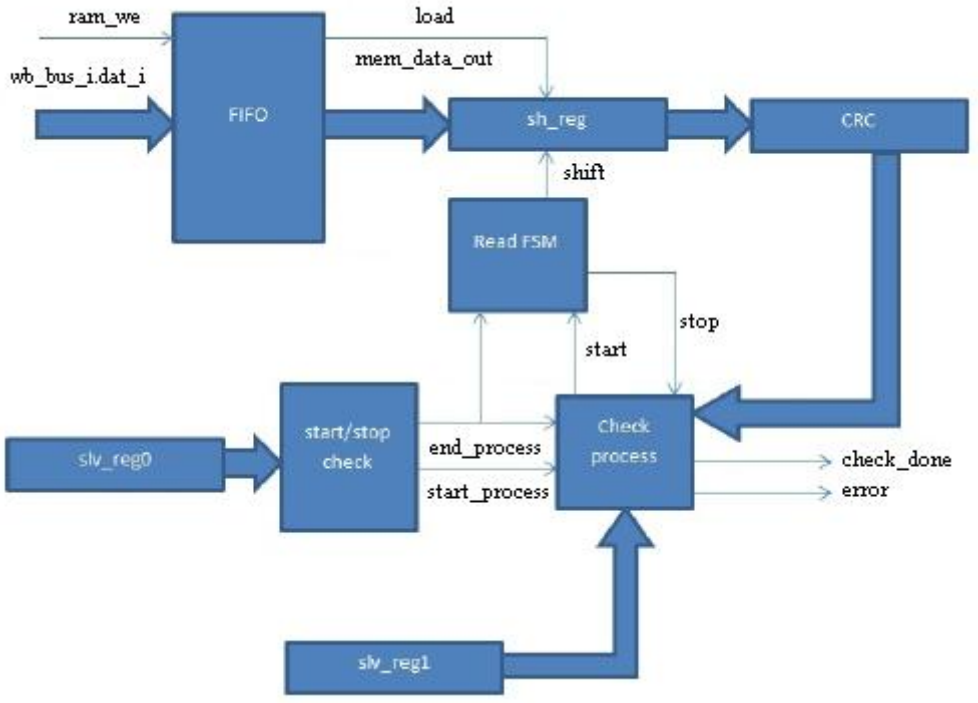

Figura 4.5: Struttura del tester

La prima istruzione di una routine di test consente di attivare il tester, la cui struttura è illustrata in figura 4.5, affinché possa ricevere i dati in uscita dal processore, che scrive il comando di avvio nel registro 0, mentre nei bit più significativi e in quelli meno significativi del registro 1 vengono posti rispettivamente la signature corretta ed il numero di parole da 32 bit che devono essere scritte nella memoria del tester. Viene generato il segnale start_process, quindi la FSM di controllo asserisce il segnale start per abilitare la FSM di lettura, che gestisce il caricamento dei risultati del test nel registro a scorrimento e la traslazione dei bit da inviare in forma seriale al circuito per il calcolo della signature. La FIFO aggiorna la posizione del puntatore di lettura ad ogni iterazione del processo, che viene ripetuto finché il processore non scrive il comando di fine nel registro 0 per indicare che ha terminato l'invio dei risultati del test all'interfaccia. A questo punto la FSM di lettura asserisce il segnale stop, quindi la FSM di controllo confronta la signature effettiva con quella attesa, asserisce il segnale check_done e, nel caso in cui l'esito del confronto non sia un'uguaglianza, porta a livello logico alto anche il segnale error. In seguito, tutti i segnali vengono riportati a livello basso e, quando viene asserito nuovamente il segnale start_process, può iniziare un altro ciclo.

4.2 Implementazione dell'interfaccia su FPGA e simulazioni

In questo paragrafo, dopo aver descritto ad alto livello la struttura del progetto open - source utilizzato in questo lavoro, si analizza l'implementazione del tester, che dipende fortemente dal tipo di bus utilizzato poiché i dispositivi slave necessitano di un'interfaccia per comunicare col bus, e quindi col processore. In seguito, si descrivono le simulazioni svolte per verificare il corretto funzionamento del tester, e si analizza l'andamento delle forme d'onda dei segnali interni, le cui transizioni avvengono come descritto nel paragrafo precedente.

4.2.1 Implementazione su FPGA

Nell'architettura ad alto livello il processore SecretBlaze, la cui struttura è stata illustrata nel capitolo precedente, è l'unico dispositivo master sul bus Wishbone, cui vengono connesse le seguenti periferiche slave, come mostrato in figura 4.6 [59][60]:

- un modulo di memoria RAM statica ("static random access memory", SRAM);
- un controllore di interruzioni ("interrupt controller", INTC), che può gestire fino a 8 interruzioni con meccanismi di mascheramento e arbitraggio;
- un timer, implementato con un contatore a 32 bit che può generare un'interruzione entro una finestra temporale configurabile;
- un'interfaccia UART ("universal asynchronous receiver transmitter"), che converte flussi di bit da un formato parallelo ad un formato seriale o viceversa per consentire la comunicazione tra il processore e le periferiche;
- un'interfaccia GPIO ("general purpose input/output"), composta da un registro a sola lettura ed un registro accessibile sia in lettura che in scrittura.

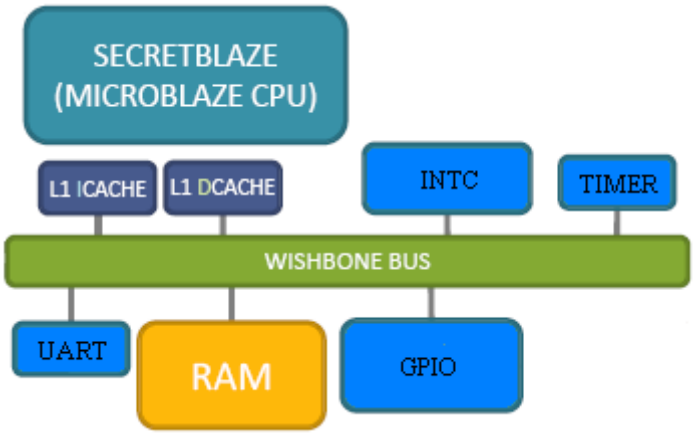

Figura 4.6: Architettura ad alto livello

In questo lavoro si vuole sottoporre a test il processore SecretBlaze che comunica con le periferiche tramite il bus Wishbone, quindi il tester, che riceve dal processore opportuni comandi e i risultati delle routine di test, deve essere connesso al bus come un ulteriore dispositivo slave. Occorre pertanto definire una nuova entità VHDL facendo riferimento alle specifiche del bus Wishbone, che

descrivono i segnali impiegati per connettere i dispositivi master e slave a questa infrastruttura di comunicazione. Essendo il tester una periferica slave, la sua connessione al bus avviene tramite i seguenti segnali d'ingresso e d'uscita, come mostrato in figura 4.7:

- *clk_i*: tutti i segnali sono campionati sul fronte di salita del clock, che coordina tutte le transazioni sul bus;
- *rst_i*: tale segnale, quando alto, riporta le due macchine a stati finiti nel loro stato iniziale, mentre tutti i segnali interni citati nel paragrafo precedente vengono riportati a livello logico basso;
- *cyc_i*: questo segnale viene asserito per indicare che è in corso un ciclo di lettura o di scrittura;
- *stb_i*: ogni volta che tale segnale viene asserito il dispositivo slave indirizzato dal master deve portare a livello alto uno dei segnali d'uscita ack_o, err_o e rty_o;
- *we_i*: questo segnale viene negato durante un ciclo di lettura, mentre viene asserito durante un ciclo di scrittura;
- *sel_i*: durante le operazioni di lettura e di scrittura, tale segnale indica quali byte del dato trasferito sono validi, quindi, essendo il bus dati a 32 bit, è formato da 4 bit;
- *adr_i*: il processore fornisce l'indirizzo binario del registro dello slave cui vuole accedere in lettura o in scrittura, quindi, se volesse inviare un comando al registro 0 del tester, dovrebbe emettere l'indirizzo 0x60000000, il cui incremento di 4 e di 8 consente di accedere rispettivamente al registro 1 e alla memoria, come già illustrato nel paragrafo precedente. La mappa di memoria, la dimensione del decodificatore degli indirizzi ed il numero di dispositivi master e slave possono essere definiti dall'utente tramite un file di configurazione;
- *dat_i*: durante un'operazione di scrittura il master pone sul bus dati il dato binario che vuole scrivere in un determinato registro dello slave, quindi, se volesse scrivere il comando di avvio o quello di fine nel registro 0 del tester, dovrebbe emettere rispettivamente i dati 0xFFFF0000 e 0xFFFFFFFF;
- *dat_o*: durante un'operazione di lettura lo slave pone sul bus il dato binario contenuto nel registro indirizzato dal master;
- *ack_o*: tale segnale viene asserito dallo slave per indicare che il ciclo di lettura o di scrittura è terminato correttamente;

- *err_o*: lo slave asserisce questo segnale per segnalare al master che il ciclo di lettura o di scrittura non è terminato correttamente poiché si è verificato un errore nel trasferimento dei dati;
- *rty_o*: lo slave non è pronto a ricevere o ad emettere dati, quindi asserisce tale segnale per indicare che il ciclo di scrittura o di lettura deve essere posticipato;
- *stall_o*: questo segnale, usato in modalità pipeline, indica che lo slave non può accettare ulteriori transazioni.

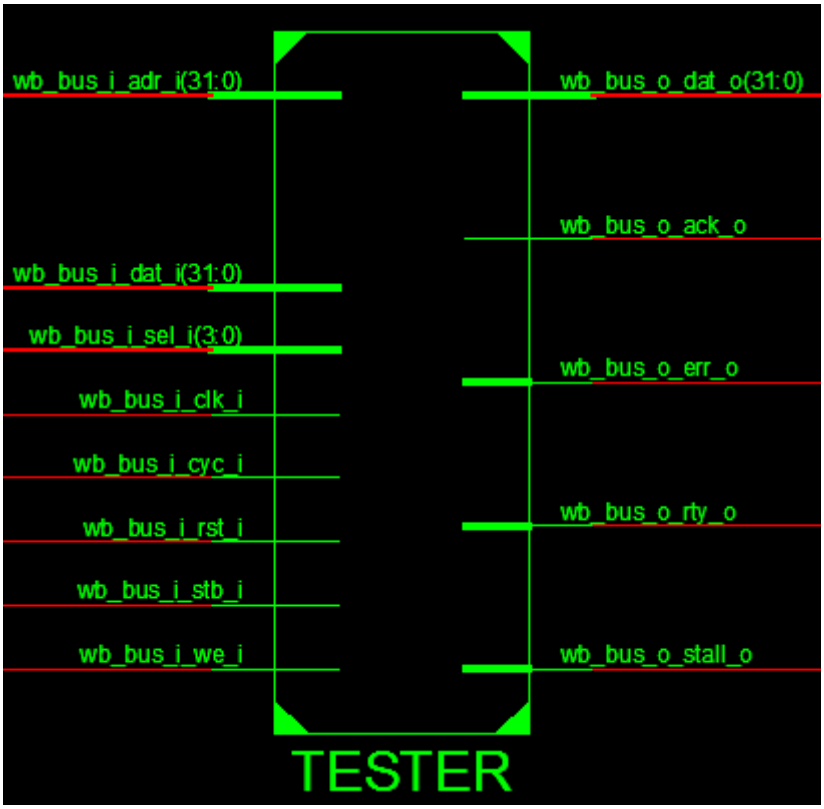

Figura 4.7: Segnali d'ingresso e d'uscita del tester

Dopo aver definito i segnali utilizzati per connettere al bus la periferica, che va istanziata nel file in cui viene descritto il sistema ad alto livello, si descrive la funzionalità del tester, presentata nel paragrafo precedente, tramite la scrittura di codice VHDL.

Il tester è stato precedentemente sviluppato per supportare la tecnica SBST, ed è stato impiegato per sottoporre a test il processore MicroBlaze, che comunica con le periferiche tramite il bus PLB,

quindi la sua implementazione in VHDL è stata opportunamente modificata in quanto fortemente dipendente dal tipo di bus utilizzato, che impiega determinati segnali e protocolli di comunicazione.

Entrambe le implementazioni utilizzano i segnali interni citati nel paragrafo precedente poiché impiegano gli stessi componenti, che vengono dichiarati ed istanziati nel file in cui viene descritta la funzionalità del tester, ed il loro funzionamento è analogo. Nell'architettura modificata, però, il calcolo della signature inizia non appena il primo risultato viene scritto nella memoria del tester, in modo tale da ridurre i tempi delle operazioni di test, mentre nell'altra implementazione occorre attendere che il tester abbia ricevuto tutti i risultati prima di effettuarne la compressione, quindi i processi che descrivono il funzionamento delle macchine a stati finiti sono diversi nei due casi. La differenza fondamentale tra le due implementazioni è invece legata al tipo di bus utilizzato poiché, come già accennato in precedenza, i bus PLB e Wishbone impiegano protocolli di lettura e di scrittura differenti, quindi occorre fare riferimento alle specifiche del bus Wishbone per scrivere i processi la cui esecuzione consente al processore di accedere in lettura o in scrittura ai due registri e alla memoria del tester. I due bus impiegano segnali differenti per connettere le periferiche slave all'infrastruttura di comunicazione, quindi le specifiche dei bus PLB e Wishbone sono state analizzate e confrontate in modo tale da poter modificare il codice VHDL, sostituendo i segnali utilizzati dal PLB con quelli corrispondenti impiegati dal Wishbone, che sono stati elencati in precedenza. In entrambi i casi i protocolli di trasferimento sono sincroni e prevedono che lo slave cui il master vuole accedere in lettura o in scrittura asserisca il segnale di acknowledge oppure indichi che si è verificato un errore nel trasferimento dei dati tramite l'asserzione del segnale d'errore. I segnali d'uscita rty_o e stall_o, non previsti nell'implementazione precedente del tester, vengono mantenuti a livello logico basso come avviene nell'implementazione degli altri dispositivi slave connessi al bus Wishbone, il cui codice VHDL è stato analizzato poiché esemplifica il comportamento dei segnali ed il funzionamento dei protocolli di comunicazione impiegati dal bus. Il tester viene connesso al bus come un ulteriore dispositivo slave, quindi è necessario modificare il file di configurazione che consente di stabilire la mappa di memoria, il numero di periferiche slave e la dimensione del decodificatore degli indirizzi affinché il tester sia direttamente accessibile dal processore. L'indirizzo base del registro 0 è 0x60000000, il cui incremento di 4 e di 8 fornisce rispettivamente gli indirizzi base del registro 1 e della memoria, quindi occorre definire un segnale interno a 2 bit che, a seconda della configurazione assunta, consente l'accesso in lettura o in scrittura ad un registro o alla memoria quando il processore indirizza il tester.

4.2.2 Simulazioni

Dopo aver implementato il tester, è necessario verificarne il corretto funzionamento, quindi occorre creare un ulteriore file VHDL, detto test bench, tramite cui si applicano al dispositivo opportune configurazioni di ingresso, ed analizzare il corrispondente andamento delle forme d'onda dei segnali interni e d'uscita in modo tale da verificare se tale andamento coincide con quello atteso, e quindi se la periferica funziona esattamente come voluto. Nella definizione di entità non vengono definiti né ingressi né uscite poiché il test bench non modella un dispositivo fisico, e quindi non risulta connesso ai componenti del sistema, mentre la descrizione dell'architettura è costituita dalle righe di codice che istanziano il circuito da simulare e definiscono l'insieme degli stimoli da applicare ai suoi ingressi. In particolare, vengono dichiarati il componente di cui si vuole simulare il funzionamento ed i segnali impiegati per connetterlo agli altri dispositivi, quindi il componente viene istanziato associandone ingressi ed uscite ai segnali precedentemente dichiarati. In seguito, il segnale di clock viene generato inizializzandolo a zero ed invertendo periodicamente il suo valore oppure assegnandogli esplicitamente i valori all'interno di un processo tramite l'utilizzo del costrutto *wait for*, che introduce un ritardo temporale tra due istruzioni di assegnamento. Una volta che l'esecuzione delle istruzioni è terminata, tale processo inizia un nuovo ciclo, quindi la generazione del clock si ripete periodicamente, mentre il segnale di reset, inizialmente a livello logico alto, viene negato dopo un certo tempo e mantenuto a livello basso, quindi il relativo processo termina con l'istruzione *wait* che lo sospende indefinitamente. Con un procedimento analogo viene definito l'andamento delle forme d'onda degli altri segnali d'ingresso, le cui commutazioni vengono sincronizzate con i fronti di salita del segnale di clock in modo tale da riuscire a tenere traccia del numero di cicli di clock in cui un segnale è attivo.

Il tester ha il compito fondamentale di controllare i risultati del test in uscita dal processore tramite il confronto tra la signature effettiva e quella attesa: occorre quindi innanzitutto verificare che esso stesso funzioni correttamente. A questo scopo, per verificarne il corretto funzionamento si scrive una signature errata nel registro 1 e, una volta che i risultati del test sono stati compressi, si verifica che la FSM di controllo asserisca il segnale error. La generazione del segnale di clock avviene come descritto in precedenza, mentre il segnale di reset, inizialmente alto affinché le due macchine a stati finiti vengano riportate nel loro stato iniziale, viene portato a livello logico basso dopo un certo tempo. A questo punto è possibile assegnare il valore binario 0xFFFF0000 al segnale dat_i in modo tale da scrivere il comando di avvio nel registro 0, il cui indirizzo base viene assegnato al segnale adr_i. Con un procedimento analogo viene indirizzato il registro 1, nei cui bit più significativi viene

posta una signature errata composta da una stringa di zeri, mentre nei bit meno significativi si indica che stanno per essere scritte due parole da 32 bit nella memoria del tester. A questo punto viene indirizzata la memoria del tester, in cui vengono scritti due valori arbitrari da 32 bit, la cui compressione avviene come descritto nel paragrafo precedente. In seguito, si assegna il valore binario 0xFFFFFFFF al segnale dat_i in modo tale da scrivere il comando di fine nel registro 0, il cui indirizzo base viene assegnato al segnale adr_i. Secondo quanto esposto nel capitolo precedente, all'inizio di ogni ciclo di scrittura, oltre ad emettere un indirizzo ed un dato, è necessario indicare che tutti i byte del dato d'ingresso sono validi portando a livello alto i 4 bit del segnale sel_i, ed asserire i segnali we_i, cyc_i e stb_i, che vengono negati dopo due cicli di clock. Una volta terminata la scrittura del test bench, è possibile visualizzare il corrispondente andamento delle forme d'onda dei segnali interni e d'uscita, che sarà discusso in seguito, per verificare che l'esito del confronto tra la signature effettiva e quella attesa non sia un'uguaglianza.

Un ulteriore test bench è stato realizzato scrivendo alcuni file in linguaggio C per poter accedere agli spazi di memoria contenuti nel tester tramite software. Il programma scritto consente di controllare il funzionamento dell'intero sistema poiché viene compilato e caricato nella memoria istruzioni del processore SecretBlaze, che lo esegue inviando comandi e dati agli spazi di memoria del tester, mentre nel test bench realizzato in precedenza viene verificato il corretto funzionamento della singola entità, cui vengono applicate opportune configurazioni di ingresso tramite la scrittura di un file VHDL.

Occorre innanzitutto modificare il file in cui vengono definiti la mappa di memoria del sistema e gli indirizzi dei registri dei dispositivi slave poiché all'architettura ad alto livello è stato aggiunto il tester, ai cui spazi di memoria vengono assegnati gli indirizzi base indicati nel paragrafo precedente. In seguito, viene creato un file in cui viene richiamata una funzione che riceve in ingresso un indirizzo ed un dato a 32 bit, e consente di scrivere il valore binario nel registro indirizzato. Occorre pertanto definire tre funzioni di questo tipo, cui vengono forniti rispettivamente gli indirizzi base del registro 0, del registro 1 e della memoria del tester, per consentire al processore di inviare comandi e dati ai due spazi di memoria dell'interfaccia. Il programma che il processore deve eseguire viene indicato in un altro file, in cui vengono richiamate le tre funzioni precedentemente definite in modo tale da scrivere opportuni dati nei registri e nella memoria del tester. Dopo aver inviato il comando di avvio al registro 0 tramite la scrittura del valore binario 0xFFFF0000, viene richiamata la funzione che indirizza il registro 1, nei cui bit meno significativi si indica che stanno per essere scritte due parole da 32 bit nella memoria del tester, mentre in quelli più significativi

viene posta una signature formata da una stringa di zeri, per verificare se la logica è in grado di stabilire che si tratta di una signature errata. A questo punto la funzione che indirizza la memoria viene richiamata due volte in modo tale da scrivere due valori arbitrari da 32 bit e, una volta che entrambi i dati sono stati salvati, viene inviato il comando di fine al registro 0 tramite la scrittura del valore binario 0xFFFFFFFF.

Il programma scritto viene compilato tramite un file script, che carica il file risultante dalla compilazione nella memoria istruzioni del processore in modo tale che il programma venga eseguito. Prima di lanciare le simulazioni, i cui risultati saranno discussi in seguito, occorre modificare il file che funge da test bench per l'architettura ad alto livello istanziando il tester, ai cui ingressi vengono forniti i segnali di clock e reset.

4.3 Risultati delle simulazioni

Dopo aver realizzato i test bench presentati nel paragrafo precedente, occorre analizzare l'andamento delle forme d'onda dei segnali interni e d'uscita del tester in modo tale da verificare se tale andamento coincide con quello atteso, e quindi se la periferica funziona correttamente.

Come mostrato in figura 4.8, il segnale di clock presenta un duty cycle del 50%, il segnale di reset è inizialmente a livello logico alto affinché le due macchine a stati finiti vengano riportate nel loro stato iniziale, e gli altri segnali d'ingresso del tester non vengono inizializzati, mentre i segnali interni citati nel paragrafo precedente vengono inizializzati a zero. Una volta che il segnale di reset è stato portato a livello logico basso, il processore può iniziare a trasmettere i comandi e i dati al tester, quindi indirizza il registro 0 e vi scrive il comando di avvio per segnalare che sta per iniziare l'invio dei risultati del test all'interfaccia. Di conseguenza, come illustrato in figura 4.9, viene generato il segnale start_process, e il tester asserisce il segnale di acknowledge per indicare che il ciclo di scrittura è terminato correttamente. La FSM di controllo passa quindi allo stato WAIT_END ed asserisce il segnale start per abilitare la FSM di lettura, che entra nello stato READ_D e vi rimane finché il segnale *ram_empty* non viene portato a livello logico basso per indicare che il primo dato può essere letto dalla memoria del tester. In seguito, come mostrato in figura 4.10, il processore indirizza il registro 1, in cui scrive una signature errata ed il numero di parole a 32 bit che stanno per essere scritte nella memoria del tester, indicati rispettivamente dai

segnali *crc_comp* e *addr_limit*. A questo punto il processore indirizza la memoria e vi scrive due valori arbitrari da 32 bit, indicati dal segnale *mem_data_out* in figura 4.11, che vengono successivamente caricati nel registro a scorrimento ed inviati in forma seriale al circuito per il calcolo della signature. Una volta che ambedue i dati sono stati salvati in memoria, il processore indirizza nuovamente il registro 0 e vi scrive il comando di fine per indicare che ha terminato l'invio dei risultati del test all'interfaccia, quindi viene generato il segnale end_process, mostrato in figura 4.12, e la FSM di controllo entra nello stato READ_RAM. Quando il primo dato viene scritto in memoria, il segnale ram_empty viene portato a livello logico basso e la FSM di lettura asserisce il segnale load per indicare che il primo dato a 32 bit è stato letto dalla memoria e caricato nel registro a scorrimento, come illustrato in figura 4.13. La FSM di lettura entra quindi nello stato SERIALIZE, in cui il segnale load viene negato, mentre il segnale shift viene asserito affinché i bit contenuti nel registro a scorrimento vengano traslati e inviati al circuito per il calcolo della signature, come mostrato in figura 4.14. Il contatore viene incrementato ad ogni ciclo di clock e, quando raggiunge il valore 31, indica che tutti i bit nel registro sono stati traslati, quindi il segnale counter_flag viene asserito e la FSM passa allo stato FINISH, in cui il segnale shift, che ha anche la funzione di abilitare il circuito per il calcolo della signature, viene negato, come indicato in figura 4.15. A questo punto la FSM torna allo stato READ_D in modo tale da leggere il secondo dato dalla memoria del tester, che viene anch'esso caricato nel registro a scorrimento ed inviato in forma seriale al circuito per il calcolo della signature. Una volta terminati la lettura dei risultati del test ed il calcolo della signature, la FSM di lettura asserisce il segnale stop e la FSM di controllo può passare allo stato CHECK_RES, in cui la signature effettiva, indicata dal segnale *crc_out* in figura 4.16, viene confrontata con quella attesa. Secondo quanto esposto in precedenza, la signature posta nei bit più significativi del registro 1 è errata, quindi la FSM di controllo asserisce il segnale error per indicare che l'esito del confronto non è un'uguaglianza, mentre il segnale check_done viene portato a livello logico alto per segnalare che il controllo è terminato. Di conseguenza, l'andamento delle forme d'onda dei segnali interni e d'uscita del tester dimostra che la periferica funziona correttamente, poiché in grado di riconoscere che la signature attesa è errata.

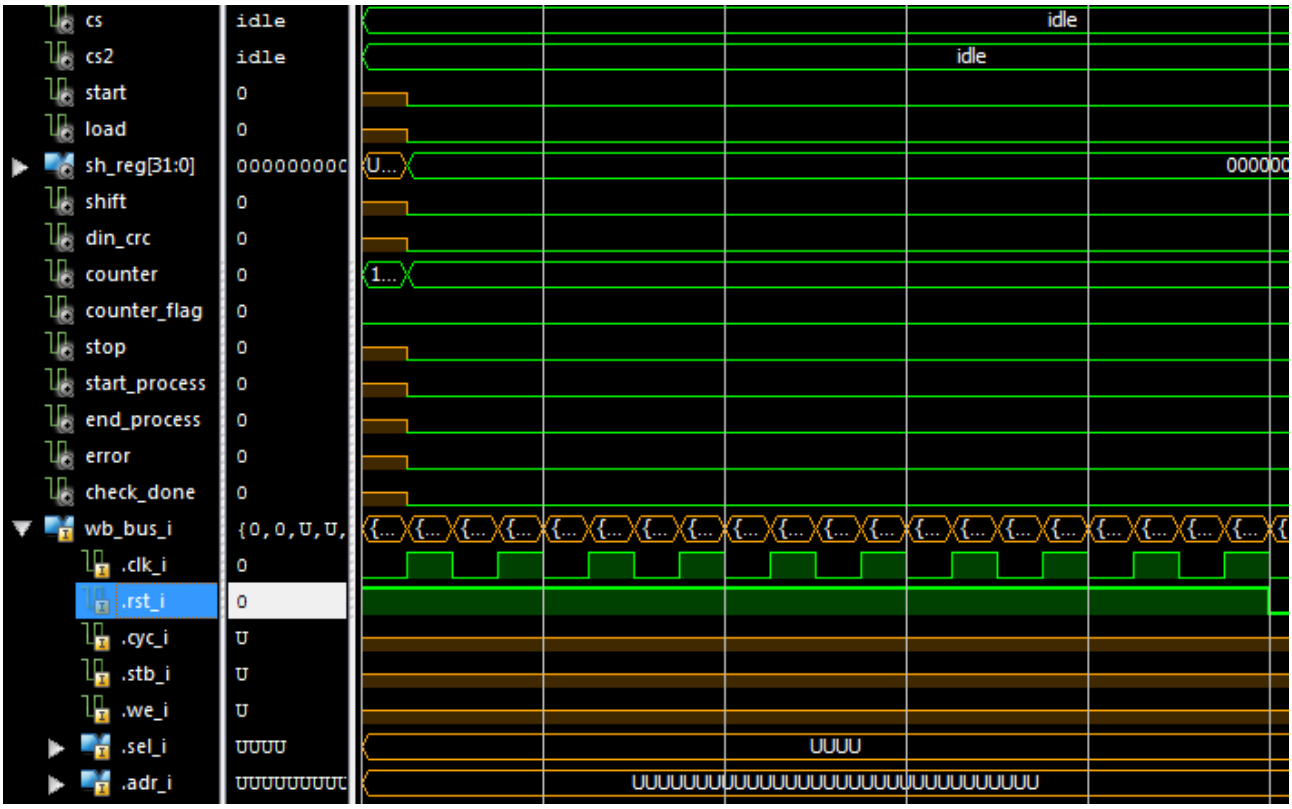

Figura 4.8: Inizializzazione dei segnali d'ingresso e interni del tester

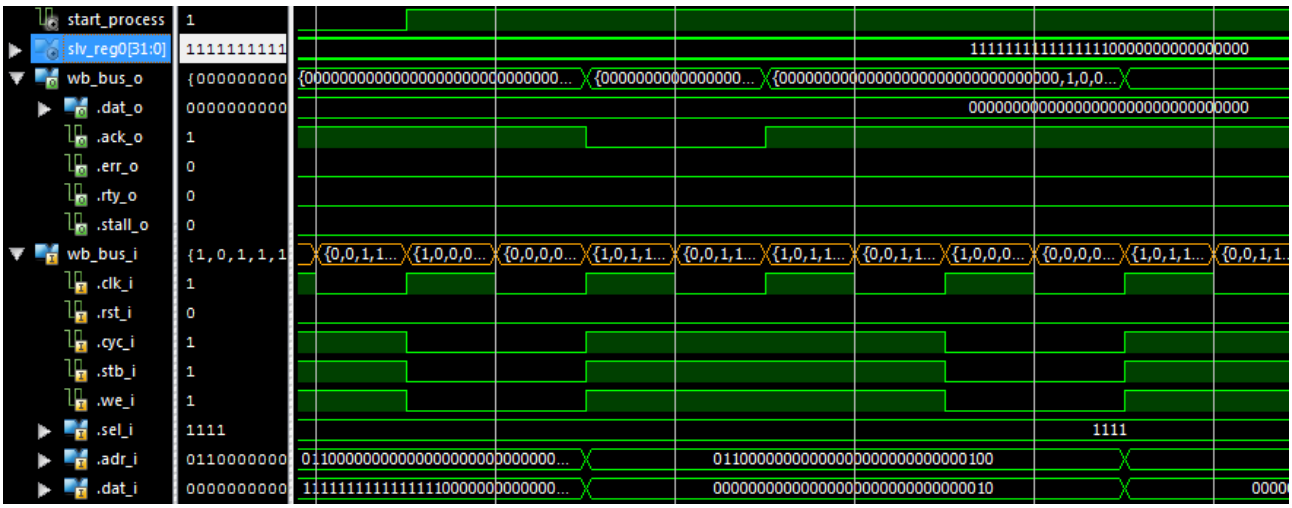

Figura 4.9: Scrittura del comando di avvio nel registro 0

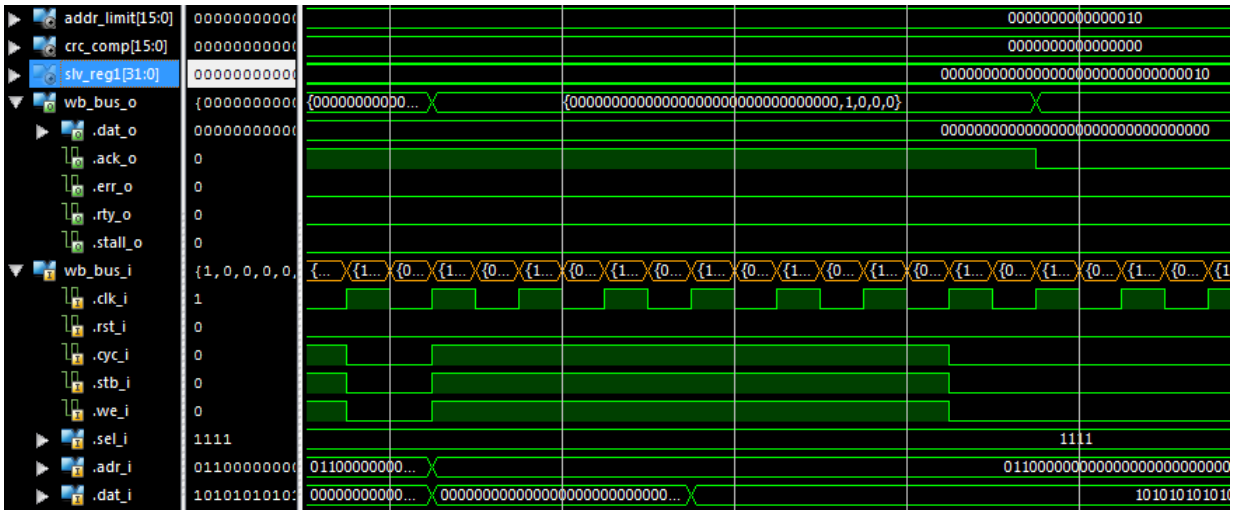

Figura 4.10: Scrittura dei dati nel registro 1

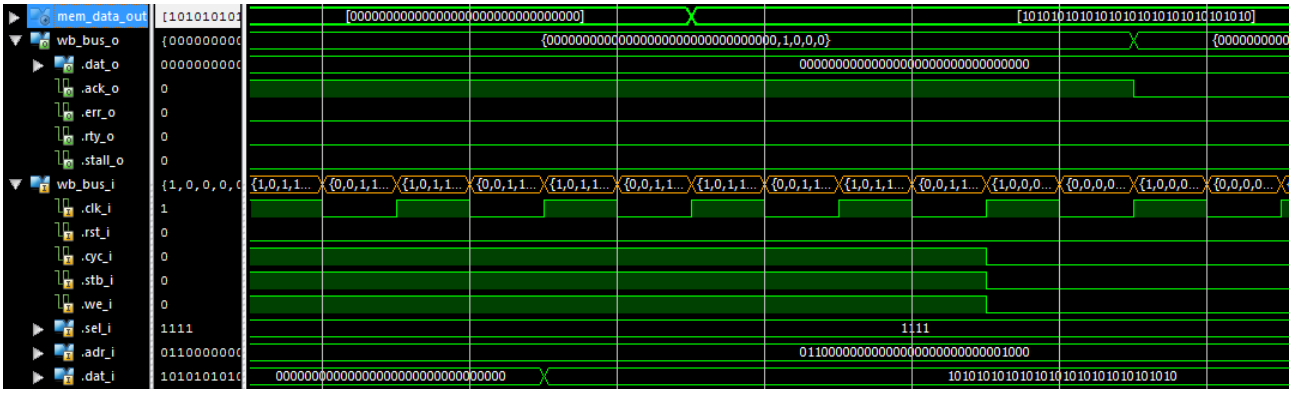

Figura 4.11: Scrittura dei dati in memoria

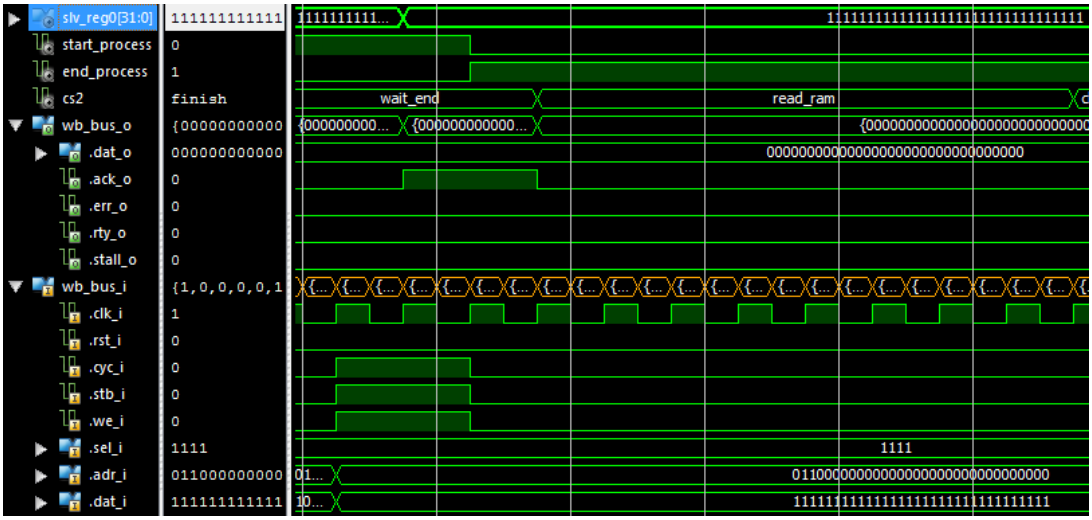

Figura 4.12: Scrittura del comando di fine nel registro 0

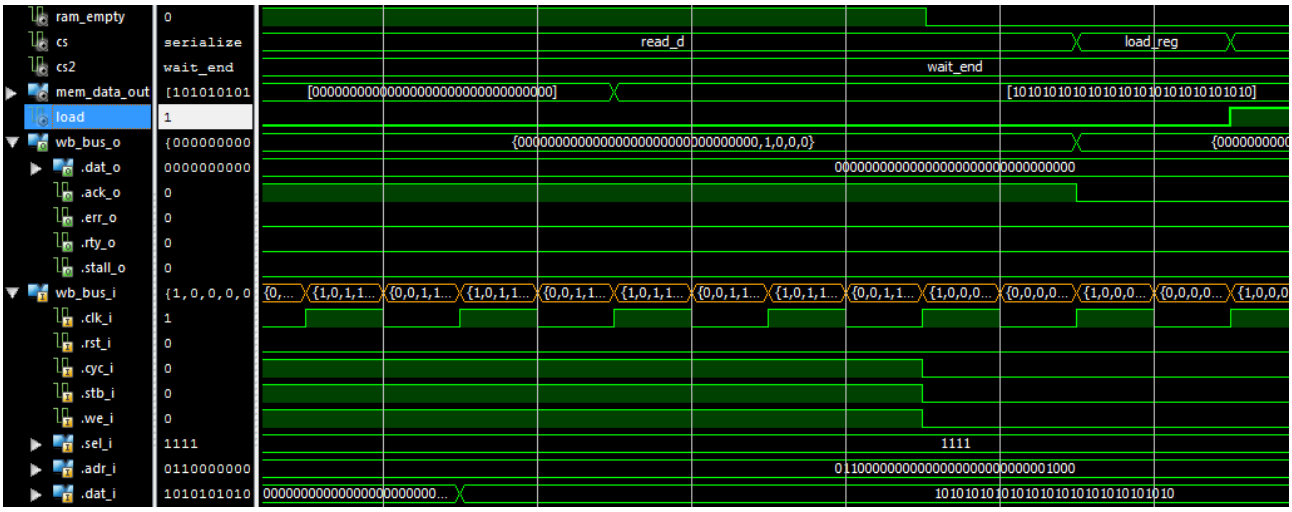

Figura 4.13: Lettura del primo dato dalla memoria

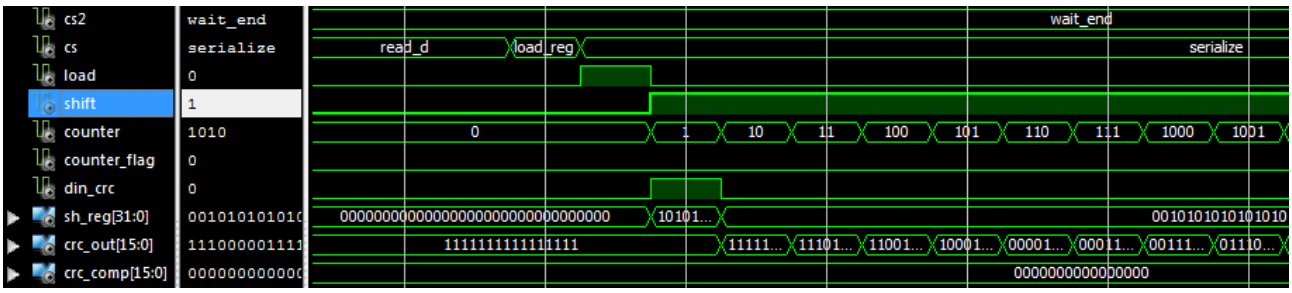

Figura 4.14: Traslazione dei bit e calcolo della signature

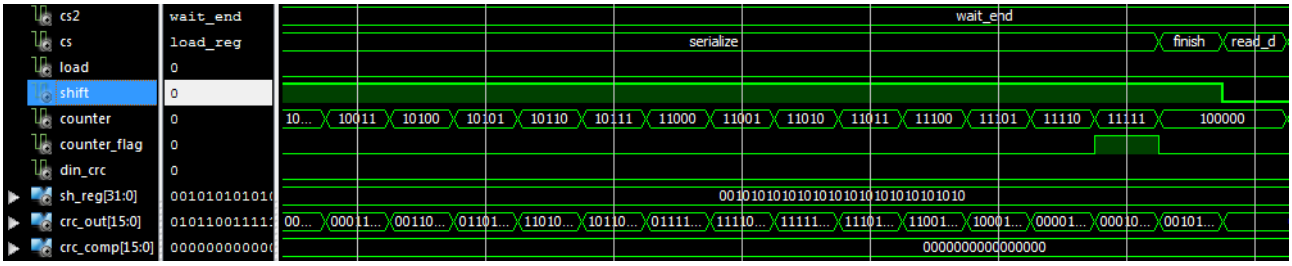

Figura 4.15: Incremento del contatore ed asserzione del segnale counter_flag

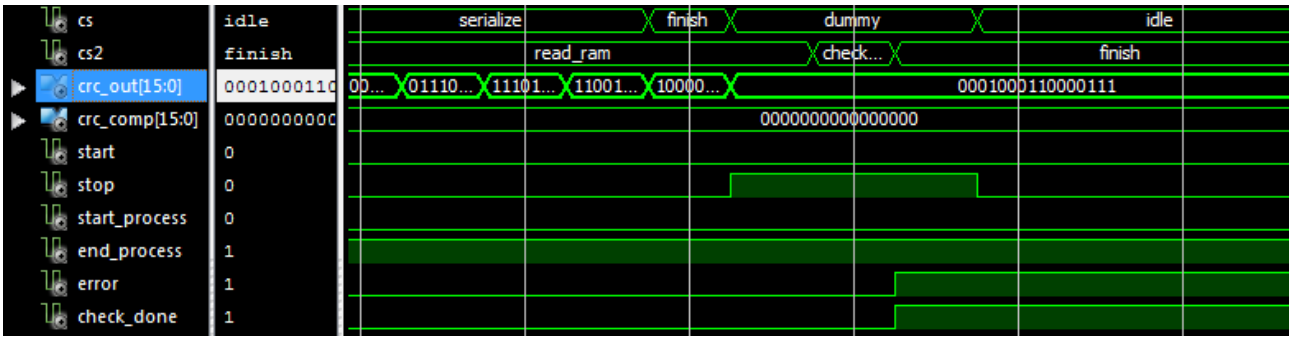

Figura 4.16: Confronto tra la signature effettiva e quella attesa

Capitolo 5

Conclusioni

In questo lavoro si è considerata un'architettura MPSoC in cui si è voluto predisporre della possibilità di sottoporre a test on-line il processore SecretBlaze, interfacciato con l'infrastruttura di comunicazione tramite un'interfaccia di rete, e connesso alle proprie periferiche tramite il bus Wishbone. A questo scopo sono state impiegate le tecniche SBST, che prevedono la generazione di opportune routine di test tramite un software esterno e la loro applicazione al processore, che le esegue alla sua velocità nominale periodicamente o in particolari momenti durante il funzionamento del sistema. I risultati del test vengono inviati ad un'interfaccia hardware, chiamata tester, precedentemente sviluppata da uno studente di altro Ateneo nel suo lavoro di tesi, i cui obiettivi erano la definizione e l'integrazione sul chip di un'architettura hardware semplice e robusta che supportasse la tecnica SBST verificando i risultati del test in uscita dal processore, poiché normalmente nell'SBST i risultati vengono caricati su un tester esterno e controllati a posteriori. Il tester è stato poi impiegato per sottoporre a test il processore MicroBlaze, interfacciato sul bus PLB, quindi in questo lavoro è stato opportunamente adattato per essere connesso come dispositivo slave al bus Wishbone, in modo tale da essere direttamente accessibile dal processore SecretBlaze, che ne indirizza gli spazi di memoria per l'invio di opportuni comandi e dei risultati delle routine di test. Infine, è stato verificato il corretto funzionamento dell'interfaccia, che asserisce il segnale d'errore quando l'esito del confronto tra la signature attesa e quella ottenuta dalla compressione dei risultati del test non è un'uguaglianza.

L'architettura può essere facilmente adattata in funzione delle esigenze del progettista. In particolare, nell'implementazione precedente occorre attendere che il tester abbia ricevuto tutti i risultati prima di effettuarne la compressione, quindi, per ridurre i tempi delle operazioni di test, è stato necessario modificare il relativo codice VHDL per fare in modo che il calcolo della signature inizi non appena il primo risultato viene scritto nella memoria del tester.

L'impiego delle tecniche SBST consente di ottenere un'elevata copertura di guasto, il cui valore, teoricamente superiore al 95%, dipende dall'efficacia delle routine di test utilizzate e può essere determinato con esattezza al termine delle operazioni di test in uno scenario reale.

Nonostante si ottenga un'elevata copertura di guasto col minimo impatto sulle prestazioni, l'area e il consumo di potenza del sistema, il tester presenta alcuni limiti, che vengono di seguito discussi.

La signature attesa e il numero di parole a 32 bit che il processore scrive nella memoria FIFO devono essere memorizzati in un apposito registro del tester non appena l'interfaccia è stata attivata dalla prima istruzione di una routine di test.

Il tester può verificare solo risultati del test da 32 bit poiché nella memoria FIFO possono essere memorizzate fino a 512 parole da 32 bit, ed il registro a scorrimento presenta 32 terminali di ingresso dati, quindi non è possibile controllare risultati più lunghi.

L'algoritmo per il calcolo della signature presenta una bassa complessità computazionale, ma il tempo necessario per il calcolo è lungo poiché i dati vengono forniti al circuito in forma seriale. Tuttavia, il processore può svolgere altre operazioni mentre viene eseguito il calcolo della signature, quindi il tempo delle operazioni di test si riduce a quello necessario per l'esecuzione delle routine.

L'implementazione del tester dipende fortemente dal tipo di bus utilizzato poiché le periferiche slave necessitano di un'interfaccia per comunicare col bus, e quindi col processore, da cui ricevono comandi e dati. Di conseguenza, il tester viene implementato tramite la scrittura di codice VHDL facendo riferimento alle specifiche del bus utilizzato, che definiscono i segnali da impiegare per connettere le periferiche slave all'infrastruttura di comunicazione, ed il funzionamento dei protocolli di lettura e di scrittura. I segnali interni e la struttura dei processi che descrivono il funzionamento delle macchine a stati finiti, del registro a scorrimento e del registro 0 del tester non vengono invece modificati poiché le varie implementazioni impiegano gli stessi componenti e presentano il medesimo funzionamento. Tuttavia, i segnali d'ingresso e d'uscita del tester dipendono dal tipo di bus utilizzato, quindi il codice VHDL va modificato sostituendo i segnali impiegati nell'implementazione precedente con quelli corrispondenti utilizzati dal nuovo bus, cui il tester viene connesso come un ulteriore dispositivo slave. E' pertanto necessario modificare la mappa di memoria del sistema, il numero di periferiche slave e la dimensione del decodificatore degli indirizzi affinché il tester sia direttamente indirizzabile dal processore. Al registro 0 viene assegnato un determinato indirizzo base, il cui incremento di 4 e di 8 fornisce rispettivamente gli indirizzi

base del registro 1 e della memoria, quindi occorre definire un segnale interno che, a seconda della configurazione assunta, consente l'accesso in lettura o in scrittura ad un registro o alla memoria quando il processore indirizza il tester.

Alcuni possibili sviluppi futuri legati a questo lavoro sono volti a cercare di risolvere i limiti precedentemente esposti, mentre altri lavori possono prevedere il trasferimento dei compiti del processore guasto ai processori funzionanti al termine dell'esecuzione delle operazioni di test.

Per realizzare un'architettura che soddisfi pienamente le esigenze degli utenti, la logica di test viene definita una volta che le routine di test sono disponibili poiché occorre conoscere la dimensione delle parole e la lunghezza delle singole routine per poter stabilire la quantità di memoria necessaria per salvare i risultati del test.

Il tempo necessario per il calcolo della signature è lungo, quindi, per anticipare il confronto tra la signature effettiva e quella attesa e l'eventuale attivazione della logica di migrazione, si dovrebbe impiegare un algoritmo di calcolo diverso da quello basato sul controllo a ridondanza ciclica, ma la logica impiegata sarebbe più complessa.

L'implementazione del tester dipende fortemente dal tipo di bus utilizzato, quindi l'applicazione del dispositivo ad altri processori che si interfacciano su bus differenti richiede una fase di ri-progettazione poiché occorre fare riferimento di volta in volta alle specifiche del bus impiegato per i motivi precedentemente esposti. Di conseguenza, è opportuno rendere più flessibile l'implementazione del tester in modo tale da poter estendere facilmente l'uso del dispositivo a diversi processori, che vengono quindi sottoposti a test ottenendo un'elevata copertura di guasto col minimo impatto sulle prestazioni, l'area e il consumo di potenza del sistema.

Il tester deve essere integrato sul chip insieme alla logica di migrazione che, nel caso in cui l'esito del confronto tra la signature attesa e quella ottenuta dalla compressione dei risultati del test non sia un'uguaglianza, assegna i compiti del processore guasto ai processori funzionanti e li sposta fisicamente, sulla base del carico di lavoro e delle prestazioni.

Bibliografia

[1] W. Wolf, A. A. Jerraya, G. Martin: "Multiprocessor system-on-chip (MPSoC) technology", 2008

[2] "Report on fault tolerant models and fault tolerance techniques", 2010

[3] M. Gao, H. M. Chang, P. Lisherness, K. T. Cheng: "Time-multiplexed online checking", 2011

[4] V.V. Kumar, J. Lach: "Heterogeneous redundancy for fault and defect tolerance with complexity independent area overhead", 2003

[5] A. Paschalis, D. Gizopoulos: "Effective software-based self-test strategies for on-line periodic testing of embedded processors", 2005

[6] W. Fornaciari, C. Brandolese: "Sistemi embedded. Sviluppo hardware e software per sistemi dedicati", 2007

[7] K. Y. Cho, S. Mitra, E. J. McCluskey: "Gate exhaustive testing", 2005

[8] E. J. McCluskey, S. Bozorgui-Nesbat: "Design for autonomous test", 1981

[9] E. J. McCluskey: "Verification testing - A pseudoexhaustive test technique", 1984

[10] R. A. Frohwerk: "Signature analysis: a new digital field service method", 1977

[11] J. Savir: "Syndrome-testable design of combinational circuits", 1980

[12] B. Konemann, J. Mucha, G. Zwiehoff: "Built-in logic block observation techniques", 1979

[13] H. Al-Asaad, B. T. Murray, J. P. Hayes: "Online BIST for embedded systems", 1998

[14] G. Xenoulis, D. Gizopoulos, N. Kranitis, A. Paschalis: "Low-cost, on-line software-based self-testing of embedded processor cores", 2003

[15] J. Shen, J. A. Abraham: "Native mode functional test generation for processors with applications to self test and design validation", 1998

[16] K. Batcher, C. Papachristou: "Instruction randomization self test for processor cores", 1999

[17] P. Parvathala, K. Maneparambil, W. Lindsay: "FRITS - A microprocessor functional BIST method", 2002

[18] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero: "Fully automatic test program generation for microprocessor cores", 2003

[19] L. Chen, S. Dey: "Software-based self-testing methodology for processor cores", 2001

[20] L. Chen, S. Ravi, A. Raghunathan, S. Dey: "A scalable software-based self-testing methodology for programmable processors", 2003

[21] N. Kranitis, A. Paschalis, D. Gizopoulos, Y. Zorian: "Instruction-based self-testing of processor cores", 2003

[22] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis: "Software-based self-testing of embedded processors", 2005

[23] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, D. Gizopoulos: "Hybrid-SBST methodology for efficient testing of processor cores", 2008

[24] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, G. I. Bourianoff: "Limits to binary logic switch scaling - A gedanken model", 2003

[25] Intel: 60 years of the transistor: 1947 - 2007. http://www.intel.com/technology/timeline.pdf

[26] Intel. Intel's vision. http://download.intel.com/newsroom/kits/idf/2011_fall/pdfs/Kirk_Skaugen_DCSG_MegaBriefing.pdf#page=21

[27] T. G. Mattson, R. Van der Wijngaart, M. Frumkin: "Programming the Intel 80-core network-on-a-chip terescale processor", 2008

[28] Adapteva. http://www.adapteva.com

[29] A. Montone, V. Rana, M. D. Santambrogio, D. Sciuto: "HARPE: a harvard-based processing element tailored for partial dynamic reconfigurable architectures", 2008

[30] A. Jerraya, H. Tenhunen, W. Wolf: "Multiprocessor systems-on-chips", 2005

[31] W. Wolf: "The future of multiprocessor systems-on-chips", 2004

[32] J. Henkel: "Closing the SoC design gap", 2003

[33] J. Nickolls, I. Buck, M. Garland, K. Skadron: "Scalable parallel programming with cuda", 2008

[34] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini: "Analyzing on-chip communication in a MPSoC environment", 2004

[35] J. Nurmi: "Network-on-chip: a new paradigm for system-on-chip design", 2005

[36] P. Guerrier, A. Greiner: "A generic architecture for on-chip packet-switched interconnections", 2000

[37] N. Saint-Jean, P. Benoit, G. Sassatelli, L. Torres, M. Robert: "MPI-based adaptive task migration support on the HS-scale system", 2008

[38] J. Pontes, M. Moreira, R. Soares, N. Calazans: "Hermes-GLP: a GALS network on chip router with power control techniques", 2008

[39] C. A. Zeferino, M. E. Kreutz, L. Carro, A. A. Suzin: "A study on communication issues for systems-on-chip", 2002

[40] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, D. Lindqvist: "Network on a chip: an architecture for billion transistor era", 2000

[41] L. Benini, G. De Micheli: "Networks on chips: a new SoC paradigm", 2002

[42] L. Benini, G. De Micheli: "Networks on chips: a new paradigm for component-based MPSoC design", 2002

[43] T. Bjerregaard, S. Mahadevan: "A survey of research and practices of network-on-chip", 2006

[44] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, R. Saleh: "Performance evaluation and design trade-offs for network-on-chip interconnect architectures", 2005

[45] S. Shamshiri, K. T. Cheng: "Yield and cost analysis of a reliable NoC", 2009

[46] G. De Micheli, L. Benini: "Networks on chips: technologies and tools", 2006

[47] Z. Shi, A. Burns: "Real-time communication analysis for on-chip networks with wormhole switching", 2008

[48] W. J. Dally, B. Towles: "Route packets, not wires: on-chip interconnection networks", 2001

[49] L. Fiorin, L. Micconi, M. Sami: "Design of fault tolerant network interfaces for NoCs", 2011

[50] "Open core protocol specification 2.2", 2006

[51] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal: "An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration", 2004

[52] S. Mourad, Y. Zorian: "Principles of testing electronic systems", 2000

[53] http://www.intel.com/p/en_US/embedded/hwsw/hardware/pentium-4

[54] *MicroBlaze processor reference guide*, Xilinx, Inc. UG081(v9.0), 2008

[55] J. G. Tong, I. D. L. Anderson, M. A. S. Khalid: "Soft-core processors for embedded systems", 2006

[56] SecretBlaze. http://janela.lirmm.fr/~barthe/index.php/page/secretblaze.html

[57] L. Barthe, L. V. Cargnini, P. Benoit, L. Torres: "The SecretBlaze: a configurable and cost-effective open-source soft-core processor", 2011

[58] WISHBONE, Revision B.4 Specification, 2010

[59] R. Busseuil, L. Barthe, G. M. Almeida, L. Ost, F. Bruguier, G. Sassatelli, P. Benoit, M. Robert, L. Torres: "Open-scale: a scalable, open-source NoC-based MPSoC for design space exploration", 2011

[60] R. Busseuil, L. Barthe, G. M. Almeida, G. Sassatelli, P. Benoit, M. Robert, L. Torres: "Design of an open-source, NoC-based MPSoC: open-scale", 2011

[61] http://pages.cs.wisc.edu/~tvrdik/7/html/Section7.html

[62] SecretBlaze. http://janela.lirmm.fr/~barthe/index.php/page/performance.html

Appendice

Linguaggio VHDL e tool utilizzati

Il flusso di progetto include la creazione di modelli tramite un linguaggio di descrizione dell'hardware ("hardware description language", HDL), la loro verifica tramite simulazioni a eventi e il trasferimento del programma su una piattaforma fisica. I linguaggi di descrizione dell'hardware consentono di definire i componenti del progetto e di istanziarli in una struttura gerarchica, e il sistema risultante può essere analizzato a diversi livelli di astrazione. Nella descrizione comportamentale vengono definite le operazioni che l'ambiente di sviluppo provvederà poi ad implementare sull'hardware nella sequenza corretta. Nell'astrazione di tipo RTL ("register transfer level") si descrive come avvengono il trasferimento e l'elaborazione dei dati all'interno del circuito, composto da reti combinatorie, che esprimono in forma esplicita le trasformazioni dei dati mediante espressioni algebriche e aritmetiche, e registri, che memorizzano i dati intermedi di elaborazioni complesse. A livello strutturale viene invece rappresentata l'architettura interna del sistema, formata dai componenti di più basso livello e dalle relative connessioni; questo stadio presuppone la scelta degli elementi che svolgono i compiti richiesti dal progetto. Indipendentemente dal livello di astrazione scelto, l'ambiente di sviluppo si occupa dell'implementazione dei passi necessari per ottenere la descrizione dell'architettura a livello di porta logica. La descrizione di un modulo VHDL comprende un'entità, in cui si definiscono i terminali d'ingresso e d'uscita del circuito, ed un'architettura, in cui si descrivono la funzionalità e la struttura interna del circuito. L'architettura è composta da una parte dichiarativa, in cui si definiscono i segnali interni e i componenti utilizzati, ed una parte assertiva, in cui si descrivono le operazioni logiche tra segnali, si specificano le istruzioni di assegnamento e si istanziano i componenti precedentemente dichiarati. Un segnale, cui vengono associati un tipo ed eventualmente un valore iniziale, è una struttura dati in grado di rappresentare forme d'onda variabili nel tempo. Un'istruzione di assegnamento, che stabilisce un legame tra i segnali a destra e a sinistra dell'operatore di assegnamento, detti rispettivamente driver e target, viene eseguita solo quando si verifica un evento, ossia un cambiamento del valore di almeno uno dei driver, quindi l'ordine temporale in cui tali istruzioni vengono eseguite è indipendente dalla loro disposizione all'interno del codice. Grazie a questa proprietà il linguaggio

VDHL riesce a riprodurre la simultaneità dell'esecuzione di operazioni propria dei circuiti elettronici. Un altro costrutto fondamentale è il processo, blocco di codice che descrive il funzionamento di un circuito sequenziale o combinatorio, e che viene eseguito solo quando si verifica un evento su uno dei segnali appartenenti alla sua *sensitivity list*.

L'ambiente di sviluppo utilizzato per l'implementazione di un progetto su FPGA è ISE ("integrated synthesis environment"), un software fornito da Xilinx che consente di gestire le varie fasi del processo di implementazione, tra cui la sintesi e la simulazione del codice VHDL. ISE mette a disposizione una serie di strumenti per la progettazione, l'analisi e l'implementazione su FPGA di circuiti digitali mediante un'interfaccia grafica semplice e intuitiva. Ogni modulo VHDL può essere descritto mediante uno schema a livello RTL, che ne rappresenta i segnali d'ingresso e d'uscita e le interconnessioni con gli altri moduli. Tale rappresentazione consente di semplificare l'analisi di strutture complesse scomponendo il sistema nei suoi blocchi costituenti, e di facilitare l'inserimento dei moduli IP, ossia unità pre - progettate messe a disposizione del progettista.

ISE integra ISim, un tool che consente di simulare il codice VHDL permettendo di verificare il corretto funzionamento di una singola entità o dell'intero sistema. Si tratta di un software che consente di applicare ad un dispositivo determinate configurazioni di ingresso, che vengono definite in un apposito file VHDL detto test bench, e di visualizzare il corrispondente andamento delle forme d'onda dei segnali d'uscita per verificare se il circuito funziona esattamente come voluto. Un evento su un segnale appartenente alla sensitivity list di un processo ne attiva l'esecuzione, ed un evento su un driver di un'istruzione di assegnazione può provocare una transizione del relativo target, il cui nuovo valore viene posto nella coda degli eventi. Una volta inserite nella coda tutte le transizioni dei target che si verificano ad un determinato istante di tempo in seguito ad un evento su un driver, il simulatore passa a considerare l'elemento successivo nella coda eseguendo nuovamente le operazioni descritte.

File

In questa sezione vengono riportati i file VHDL che sono stati scritti per interfacciare il tester al bus Wishbone e per verificarne il corretto comportamento. In seguito, si includono i file scritti in linguaggio C per controllare il funzionamento dell'intero sistema e realizzare dunque un ulteriore test bench.

File tester_pack.vhd

```
--
--! The package implements useful defines & tools for TESTER.
--

--! TESTER Package
package tester_pack is

  -- //////////////////////////////////////////
  --              TESTER SETTINGS
  -- //////////////////////////////////////////

  constant TESTER_W : natural := 32; --! TESTER register width

  subtype tester_data_t is std_ulogic_vector(TESTER_W-1 downto 0); --! TESTER data type

  -- //////////////////////////////////////////
  --      TESTER WB SLAVE INTERFACE SETTINGS
  -- //////////////////////////////////////////

  --
  -- MEMORY MAP DEFINES
  --

  constant MAX_SLV_TESTER_W : natural := 32; --! TESTER WISHBONE read data buffer max width

  subtype wb_tester_reg_adr_t is std_ulogic_vector(1 downto 0); --! TESTER register memory map type
  constant SLV0_OFF      : wb_tester_reg_adr_t := "00";  -- base + 0x0
  constant SLV1_OFF      : wb_tester_reg_adr_t := "01";  -- base + 0x4
  constant MEM_OFF       : wb_tester_reg_adr_t := "10";  -- base + 0x8

end tester_pack;
```

File tester_slave_wb_bus.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library soc_lib;
use soc_lib.tester_pack.all;

library wb_lib;
use wb_lib.wb_pack.all;

--! TESTER WISHBONE Bus Slave Interface Entity
entity tester_slave_wb_bus is

  port
    (
              wb_bus_i : in wb_slave_bus_i_t; --! WISHBONE slave inputs
                    wb_bus_o : out wb_slave_bus_o_t --! WISHBONE slave outputs
            );

end tester_slave_wb_bus;

--! TESTER Wishbone Bus Slave Interface Architecture
architecture Behavioral of tester_slave_wb_bus is

  -- component that calculates the CRC
  component crc_16 is
   Port (
            clk                             : in std_ulogic;
            rst                             : in std_ulogic;
            din                             : in std_ulogic;
                    den                             : in std_ulogic;
            crc16_out           : out std_ulogic_vector(15 downto 0)
   );
            end component;

            component BRAM_fifo IS
  PORT (
    rst : IN std_ulogic;
    wr_clk : IN std_ulogic;
    rd_clk : IN std_ulogic;
    din : IN std_ulogic_VECTOR(31 DOWNTO 0);
    wr_en : IN std_ulogic;
    rd_en : IN std_ulogic;
    dout : OUT std_logic_VECTOR(31 DOWNTO 0);
    full : OUT std_ulogic;
    overflow : OUT std_ulogic;
    empty : OUT std_ulogic;
    valid : OUT std_ulogic;
    rd_data_count : OUT std_logic_VECTOR(8 DOWNTO 0);
    wr_data_count : OUT std_logic_VECTOR(8 DOWNTO 0)
  );
END component;

  -- internal signals for custom logic
type fsm_state is (IDLE, READ_D, WAIT_REG, LOAD_REG, SERIALIZE, FINISH, DUMMY);
```

```
signal cs : fsm_state;
type check_state is (IDLE, WAIT_END, READ_RAM, CHECK_RES, FINISH);
signal cs2 : check_state;
signal sh_reg : tester_data_t; -- shift register for sending data to the CRC calculator
signal counter : integer; -- counter that counts the data shifted
signal counter_flag : std_ulogic; -- flag the raises when the counter reaches the end
signal load : std_ulogic; -- load for the shift register
signal shift : std_ulogic; -- shift for the shift register
signal addr_limit : std_ulogic_vector (15 downto 0); -- address limit where to read, calculated from the external one
signal din_crc : std_ulogic; -- data input to the CRC calculator
signal crc_out : std_ulogic_vector (15 downto 0); -- output signature
signal crc_comp : std_ulogic_vector (15 downto 0); -- CRC read from the register 1 (it must be known, it is stored by
the MB)
signal start, stop : std_ulogic;
signal start_process, end_process : std_ulogic;
signal error, check_done : std_ulogic; -- final signals
signal ONE, ZERO : tester_data_t;
signal ram_we, ram_re, ram_empty, ram_full, ram_overflow, ram_valid : std_ulogic;

 -------------------------------------------
 -- Signals for tester slave model s/w accessible register example
 -------------------------------------------
 signal slv_reg0                 : tester_data_t;
 signal slv_reg1                 : tester_data_t;
 signal slv_ip2bus_data          : wb_bus_data_t;
 signal slv_ack                  : std_ulogic;

 -------------------------------------------
 -- Signals for tester memory space example
 -------------------------------------------
 type DO_TYPE is array (0 to 0) of std_ulogic_vector(tester_data_t'length-1 downto 0);
 signal mem_data_out             : DO_TYPE;
 signal mem_data_out_1                                          : std_logic_vector(tester_data_t'length-1 downto
0);
 signal mem_select               : std_ulogic;
 signal mem_read_enable          : std_ulogic;
 signal mem_read_enable_dly1     : std_ulogic;
 signal mem_read_req             : std_ulogic;
 signal mem_ip2bus_data          : wb_bus_data_t;
 signal mem_read_ack_dly1        : std_ulogic;
 signal mem_read_ack             : std_ulogic;
 signal mem_write_ack                                           : std_ulogic;

 signal wb_ack_o_r : std_ulogic;                        --! WISHBONE single read/write ack reg
 signal wb_dat_o_r : std_ulogic_vector(MAX_SLV_TESTER_W - 1 downto 0); --! WISHBONE data bus reg

 --
 -- SLAVE INTERFACE SIGNALS
 --

 signal slv_read_s                                              : wb_bus_data_t;
 signal mem_read_s                                              : wb_bus_data_t;
 signal slv_write_slv0_s                                : wb_bus_data_t;
 signal slv_write_slv1_s                                : wb_bus_data_t;
 signal slv_write_fifo_s                                : wb_bus_data_t;

 --
 -- WB SIGNALS
 --

 signal wb_we_s                                                 : std_ulogic;
 signal wb_re_s                                                 : std_ulogic;
```

```
  signal wb_reg_adr_s                : wb_tester_reg_adr_t;
  signal wb_ack_s                                        : std_ulogic;

begin

  --
  -- ASSIGN OUTPUTS
  --

  wb_bus_o.ack_o   <= wb_ack_o_r;
  wb_bus_o.dat_o   <= std_ulogic_vector(resize(unsigned(wb_dat_o_r),wb_bus_data_t'length));
  wb_bus_o.err_o   <= '0'; -- not implemented
  wb_bus_o.rty_o   <= '0'; -- not implemented
  wb_bus_o.stall_o <= '0'; -- not implemented

  --
  -- WB SIGNALS
  --

  wb_we_s       <= (wb_bus_i.stb_i and wb_bus_i.cyc_i and wb_bus_i.we_i);                                    -- write bus
operation
  wb_re_s       <= (wb_bus_i.stb_i and wb_bus_i.cyc_i and not(wb_bus_i.we_i));                               -- read bus
operation
  wb_reg_adr_s    <=    (wb_bus_i.adr_i(wb_tester_reg_adr_t'length    +    WB_WORD_ADR_OFF    -    1    downto
WB_WORD_ADR_OFF)); -- register address
  wb_ack_s     <= (wb_bus_i.stb_i and wb_bus_i.cyc_i);                                                  -- pipelined read/write ack

  ONE <= (others => '1');
  ZERO <= (others => '0');
  addr_limit <= slv_reg1(tester_data_t'length/2-1 downto 0);
  crc_comp <= slv_reg1(tester_data_t'length-1 downto tester_data_t'length/2);

  start_stop_check: process(wb_bus_i.clk_i)

  begin
          if (wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then
                  if (wb_bus_i.rst_i = '1') then
                          start_process <= '0';
                          end_process <= '0';
                  else
                          if slv_reg0 = ONE(tester_data_t'length/2-1 downto 0) & ZERO(tester_data_t'length/2-1
downto 0) then -- read the start command from register 0 (the MB is starting to send data to the RAM)
                                  start_process <= '1';
                                  end_process <= '0';
                          elsif slv_reg0 = ONE(tester_data_t'length-1 downto 0) then -- read the end command from
register 0 (MB finished sending data to the RAM)
                                  start_process <= '0';
                                  end_process <= '1';
                          else
                                  start_process <= '0';
                                  end_process <= '0';
                          end if;
                  end if;
          end if;

  end process start_stop_check;

   check_process: process(wb_bus_i.clk_i)

  begin
          if (wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then
                  if (wb_bus_i.rst_i = '1') then
```

```
                        error <= '0';
                        check_done <= '0';
                        start <= '0';
                        cs2 <= IDLE;
                else
                        case cs2 is
                                when IDLE =>
                                        error <= '0';
                                        check_done <= '0';
                                        start <= '0';
                                        if (start_process = '1') then -- when the process start get ready to read from
RAM
                                                cs2 <= WAIT_END;
                                        else
                                                cs2 <= IDLE;
                                        end if;
                                when WAIT_END =>
                                        if (end_process = '1') then -- the RAM is written
                                                cs2 <= READ_RAM;
                                        else
                                                cs2 <= WAIT_END;
                                        end if;
                                        start <= '1';
                                when READ_RAM => -- start reading from the RAM and wait for the end
                                        if (stop = '1') then
                                                cs2 <= CHECK_RES;
                                        else
                                                cs2 <= READ_RAM;
                                        end if;
                                        start <= '0';
                                when CHECK_RES =>
                                        if (crc_out = crc_comp) then -- when the CRC is calculated compare it with
the right one
                                                error <= '0';
                                        else
                                                error <= '1';
                                        end if;
                                        check_done <= '1';
                                        start <= '0';
                                        cs2 <= FINISH;
                                when FINISH =>
                                        if (start_process = '1') then
                                                cs2 <= WAIT_END;
                                                check_done <= '0';
                                                error <= '0';
                                        else
                                                cs2 <= FINISH;
                                        end if;
                                when others => null;
                        end case;
                end if;
        end if;

 end process check_process;

 ---------------------------------------
 -- Example code to read/write tester slave model s/w accessible registers
 ---------------------------------------

 -- This process implements the behaviour of a bus write operation.
 COMB_SLAVE_WRITE_REG: process(wb_bus_i, slv_reg0, slv_reg1, mem_data_out(0), wb_we_s, wb_reg_adr_s)
 begin
```

```
  -- default
  slv_write_slv0_s <= std_ulogic_vector(resize(unsigned(slv_reg0),wb_bus_data_t'length));
          slv_write_slv1_s <= std_ulogic_vector(resize(unsigned(slv_reg1),wb_bus_data_t'length));
          slv_write_fifo_s <= std_ulogic_vector(resize(unsigned(mem_data_out(0)),wb_bus_data_t'length));

  -- write enable
  if(wb_we_s = '1') then

    -- decode address
    case wb_reg_adr_s is

      when SLV0_OFF =>

        for i in 0 to (wb_bus_data_t'length/8)-1 loop
          if (wb_bus_i.sel_i(i) = '1') then
            slv_write_slv0_s(8*(i+1) - 1 downto 8*i) <= wb_bus_i.dat_i(8*(i+1) - 1 downto 8*i);
          end if;
        end loop;

      when SLV1_OFF =>

        for i in 0 to (wb_bus_data_t'length/8)-1 loop
          if (wb_bus_i.sel_i(i) = '1') then
            slv_write_slv1_s(8*(i+1) - 1 downto 8*i) <= wb_bus_i.dat_i(8*(i+1) - 1 downto 8*i);
          end if;
        end loop;

      when MEM_OFF =>

                                  for i in 0 to (wb_bus_data_t'length/8)-1 loop
          if (wb_bus_i.sel_i(i) = '1') then
            slv_write_fifo_s(8*(i+1) - 1 downto 8*i) <= wb_bus_i.dat_i(8*(i+1) - 1 downto 8*i);
          end if;
        end loop;

                    when others =>
                      null;

                  end case;

    end if;

  end process COMB_SLAVE_WRITE_REG;

  -- This process implements read/write registers.
  CYCLE_WRITE_REG: process(wb_bus_i.clk_i)
  begin

    -- clock event
    if(wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then

      -- sync reset
      if(wb_bus_i.rst_i = '1') then
        slv_reg0                        <= (others =>'0');
                    slv_reg1                                <= (others =>'0');
                    mem_data_out(0)         <= (others =>'X');

      else
        slv_reg0                        <= slv_write_slv0_s(TESTER_W - 1 downto 0);
                    slv_reg1                                <= slv_write_slv1_s(TESTER_W - 1 downto 0);
                    mem_data_out(0)         <= slv_write_fifo_s(TESTER_W - 1 downto 0);
```

```
    end if;

  end if;

end process CYCLE_WRITE_REG;

-- This process implements the behaviour of a bus read operation.
COMB_SLAVE_READ_REG: process(wb_bus_i, slv_reg0, slv_reg1, mem_data_out(0), wb_re_s, wb_reg_adr_s)

  variable slv0_v : wb_bus_data_t;
  variable slv1_v : wb_bus_data_t;
            variable fifo_v : wb_bus_data_t;

begin

  slv0_v      := std_ulogic_vector(resize(unsigned(slv_reg0),wb_bus_data_t'length));
  slv1_v      := std_ulogic_vector(resize(unsigned(slv_reg1),wb_bus_data_t'length));
  fifo_v      := std_ulogic_vector(resize(unsigned(mem_data_out(0)),wb_bus_data_t'length));

  -- default
  slv_read_s <= (others =>'0');
  mem_read_s <= (others =>'0');

  -- read enable
  if(wb_re_s = '1') then

    -- decode reg address
    case wb_reg_adr_s is

      when SLV0_OFF =>

        for i in 0 to (wb_bus_data_t'length/8)-1 loop
          if (wb_bus_i.sel_i(i) = '1') then
            slv_read_s(8*(i+1) - 1 downto 8*i) <= slv0_v(8*(i+1) - 1 downto 8*i);
          end if;
        end loop;

      when SLV1_OFF =>

        for i in 0 to (wb_bus_data_t'length/8)-1 loop
          if (wb_bus_i.sel_i(i) = '1') then
            slv_read_s(8*(i+1) - 1 downto 8*i) <= slv1_v(8*(i+1) - 1 downto 8*i);
          end if;
        end loop;

      when MEM_OFF =>

        for i in 0 to (wb_bus_data_t'length/8)-1 loop
          if (wb_bus_i.sel_i(i) = '1') then
            mem_read_s(8*(i+1) - 1 downto 8*i) <= fifo_v(8*(i+1) - 1 downto 8*i);
          end if;
        end loop;

      when others =>
        null;

    end case;

  end if;

end process COMB_SLAVE_READ_REG;
```

```
 -- This process implements WISHBONE slave output registers.
 CYCLE_TESTER_WB_OUT_REG: process(wb_bus_i.clk_i)
 begin

   -- clock event
   if(wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then

     -- sync reset
     if(wb_bus_i.rst_i = '1') then
       slv_ack <= '0';

     else
       slv_ack <= wb_ack_s;
       slv_ip2bus_data  <= slv_read_s(MAX_SLV_TESTER_W - 1 downto 0);
                        mem_ip2bus_data  <= mem_read_s(MAX_SLV_TESTER_W - 1 downto 0);

     end if;

   end if;

 end process CYCLE_TESTER_WB_OUT_REG;

 -------------------------------------------
 -- Example code to access tester memory region
 -------------------------------------------
 mem_select          <= wb_bus_i.adr_i(wb_tester_reg_adr_t'length + WB_WORD_ADR_OFF - 1) and
not(wb_bus_i.adr_i(WB_WORD_ADR_OFF));
 mem_read_enable <= mem_select and wb_re_s;
 mem_read_ack        <= mem_read_ack_dly1;
 mem_write_ack <= mem_select and wb_we_s;

 -- implement single clock wide read request
 mem_read_req    <= mem_read_enable and not(mem_read_enable_dly1);
 BRAM_RD_REQ_PROC: process(wb_bus_i.clk_i)
 begin

   if (wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then
     if (wb_bus_i.rst_i = '1') then
       mem_read_enable_dly1 <= '0';
     else
       mem_read_enable_dly1 <= mem_read_enable;
     end if;
   end if;

 end process BRAM_RD_REQ_PROC;

 -- this process generates the read acknowledge 1 clock after read enable
 -- is presented to the BRAM block. The BRAM block has a 1 clock delay
 -- from read enable to data out.
 BRAM_RD_ACK_PROC: process(wb_bus_i.clk_i)
 begin

   if (wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then
     if (wb_bus_i.rst_i = '1') then
       mem_read_ack_dly1 <= '0';
     else
       mem_read_ack_dly1 <= mem_read_req;
     end if;
   end if;

 end process BRAM_RD_ACK_PROC;
```

```
 ram_we <= wb_we_s and mem_select;

 mem_data_out_1 <= To_StdLogicVector(mem_data_out(0));

 BRAM_GEN: BRAM_fifo port map(
          rst                                => wb_bus_i.rst_i,
  wr_clk                          => wb_bus_i.clk_i,
  rd_clk                          => wb_bus_i.clk_i,
  din                             => wb_bus_i.dat_i,
  wr_en                   => ram_we,
  rd_en                   => ram_re,
  dout                            => mem_data_out_1,
  full                            => ram_full,
  overflow                => ram_overflow,
  empty                   => ram_empty,
  valid                   => ram_valid,
  rd_data_count => open,
  wr_data_count=> open
);

-- tester code

        SHIFT_REG: process(wb_bus_i.clk_i) -- implement shift register for the CRC (data must be serial)
        begin
        if (wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then
                if (wb_bus_i.rst_i = '1') then
                        sh_reg <= (others => '0');
                else
                        if (ram_valid = '1') then
                                sh_reg <= mem_data_out(0);
                        elsif (shift = '1') then
                                sh_reg <= '0' & sh_reg(tester_data_t'length-2 downto 0);
                        end if;
                end if;
        end if;
        end process SHIFT_REG;

        din_crc <= sh_reg(tester_data_t'length-1);
        counter_flag <= '1' when counter = 31 else '0'; -- check when the counter reaches 31 (32 iterations done)

        READ_FSM: process(wb_bus_i.clk_i)
        begin
        if (wb_bus_i.clk_i'event and wb_bus_i.clk_i = '1') then
                if (wb_bus_i.rst_i = '1') then
                        cs <= IDLE;
                        load <= '0';
                        shift <= '0';
                        stop <= '0';
                        ram_re <= '0';
                        counter <= 0;
                else
                        case cs is
                                when IDLE =>
                                        load <= '0';
                                        shift <= '0';
                                        ram_re <= '0';
                                        counter <= 0;
                                        if (start = '1') then -- when the start command is issued start reading from
the RAM
                                                cs <= READ_D;
                                        else
```

```
                                        cs <= IDLE;
                                end if;
                        when READ_D => -- read the data from the RAM
                                if (ram_empty = '1') then
                                        ram_re <= '0';
                                        if (end_process = '0') then
                                                cs <= READ_D;
                                        else
                                                cs <= FINISH;
                                        end if;
                                else
                                        cs <= LOAD_REG;
                                        ram_re <= '1';
                                end if;
                                load <= '0';
                                shift <= '0';
                                stop <= '0';
                                counter <= 0;
                        when LOAD_REG =>
                                ram_re <= '0';
                                load <= '1';
                                shift <= '0';
                                stop <= '0';
                                counter <= 0;
                                cs <= SERIALIZE;
                        when WAIT_REG =>
                                cs <= SERIALIZE;
                        when SERIALIZE => -- serialize the data for the CRC (32 data in the register, 32 iterations needed)
                                counter <= counter + 1;
                                shift <= '1';
                                load <= '0';
                                if (counter_flag = '1') then -- when the counter reaches 32 iterations change state
                                        cs <= FINISH;
                                else
                                        cs <= SERIALIZE;
                                end if;
                        when FINISH =>
                                shift <= '0';
                                load <= '0';
                                if (end_process = '1') then -- check if all the words have been read
                                        cs <= DUMMY; -- if yes finish
                                        stop <= '1';
                                else
                                        cs <= READ_D; -- if no read the next word from the RAM
                                end if;
                        when DUMMY =>
                                if (check_done = '1') then -- wait for the end
                                        cs <= IDLE;
                                        stop <= '0';
                                else
                                        cs <= DUMMY;
                                end if;
                        when others => null;
                end case;
        end if;
        end if;
end process READ_FSM;

CRC_CALC : crc_16 port map( -- calculate CRC16
        clk                     => wb_bus_i.clk_i,
```

```
                rst                     => wb_bus_i.rst_i,
                din                     => din_crc,
                den                     => shift,
                crc16_out       => crc_out
        );

  -- end tester code

  ------------------------------------------
  -- Example code to drive IP to Bus signals
  ------------------------------------------
  wb_dat_o_r <= slv_ip2bus_data when slv_ack = '1' else
                                          mem_ip2bus_data when mem_read_ack = '1' else
                                          (others => '0');

  wb_ack_o_r <= slv_ack or mem_read_ack or mem_write_ack;

end Behavioral;
```

File tester_tb.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

library soc_lib;
use soc_lib.tester_pack.all;

library wb_lib;
use wb_lib.wb_pack.all;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

-- Component Declaration
        COMPONENT tester_slave_wb_bus
        PORT(
                wb_bus_i: IN wb_slave_bus_i_t;
                wb_bus_o: OUT wb_slave_bus_o_t
                );
        END COMPONENT;

        SIGNAL wb_bus_i: wb_slave_bus_i_t;
        SIGNAL wb_bus_o: wb_slave_bus_o_t;

BEGIN

-- Component Instantiation
        uut: tester_slave_wb_bus PORT MAP(
                wb_bus_i => wb_bus_i,
                wb_bus_o => wb_bus_o
        );

-- Concurrent process to offer clock signal
  clk : process
  begin
                wb_bus_i.clk_i <= '0';                                  -- clock cycle 5 ns
                        wait for 2.5 ns;
                        wb_bus_i.clk_i <= '1';
                        wait for 2.5 ns;
  end process clk;

--  Test Bench Statements
  tb : PROCESS
  BEGIN

                        wb_bus_i.rst_i <= '1';
    wait for 50 ns; -- wait until global set/reset completes

                        wb_bus_i.rst_i <= '0';
                        wait for 2.5 ns;

                        -- The start command is written on register 0
                        wb_bus_i.adr_i <= "01100000000000000000000000000000";
                        wb_bus_i.dat_i <= "11111111111111110000000000000000";
```

```
                    wb_bus_i.we_i <= '1';
                    wb_bus_i.sel_i <= "1111";
                    wb_bus_i.cyc_i <= '1';
                    wb_bus_i.stb_i <= '1';
                    wait for 10 ns;

                    wb_bus_i.we_i <= '0';
                    wb_bus_i.stb_i <= '0';
                    wb_bus_i.cyc_i <= '0';
                    wait for 5 ns;

                    -- The right data is written on register 1
                    wb_bus_i.adr_i <= "01100000000000000000000000000100";
                    wb_bus_i.dat_i <= "00000000000000000000000000000010";
                    wb_bus_i.we_i <= '1';
                    wb_bus_i.cyc_i <= '1';
                    wb_bus_i.stb_i <= '1';
                    wait for 10 ns;

                    wb_bus_i.we_i <= '0';
                    wb_bus_i.stb_i <= '0';
                    wb_bus_i.cyc_i <= '0';
                    wait for 5 ns;

                    -- Signal ram_we is raised
                    wb_bus_i.adr_i <= "01100000000000000000000000001000";
                    wb_bus_i.dat_i <= "00000000000000000000000000000000";
                    wb_bus_i.we_i <= '1';
                    wb_bus_i.stb_i <= '1';
                    wb_bus_i.cyc_i <= '1';
                    wait for 15 ns;

                    -- The first data written in the FIFO is "00000000000000000000000000000000", the second is
"10101010101010101010101010101010"
                    wb_bus_i.dat_i <= "10101010101010101010101010101010";
                    wait for 15 ns;

                    -- Signal ram_we is negated
                    wb_bus_i.we_i <= '0';
                    wb_bus_i.stb_i <= '0';
                    wb_bus_i.cyc_i <= '0';
                    wait for 300 ns;

                    -- The end command is written on register 0
                    wb_bus_i.adr_i <= "01100000000000000000000000000000";
                    wb_bus_i.dat_i <= "11111111111111111111111111111111";
                    wb_bus_i.we_i <= '1';
                    wb_bus_i.cyc_i <= '1';
                    wb_bus_i.stb_i <= '1';
                    wait for 10 ns;

                    wb_bus_i.we_i <= '0';
                    wb_bus_i.stb_i <= '0';
                    wb_bus_i.cyc_i <= '0';

      wait; -- will wait forever

   END PROCESS tb;
 --  End Test Bench

 END;
```

File sb_tester.h

```
#ifndef _SB_TESTER_H
#define _SB_TESTER_H

#include "sb_types.h"
#include "sb_io.h"
#include "sb_def.h"

/* INLINE FUNCTIONS */

/**
 * \fn void reg0_write(const sb_uint32_t data)
 * \brief Write data to reg0
 * \param[in] data The data to write
 */
static __inline__ void reg0_write(const sb_uint32_t data)
{
  WRITE_REG32(TESTER_SLV0_REG,data);
}

/**
 * \fn void reg1_write(const sb_uint32_t data)
 * \brief Write data to reg1
 * \param[in] data The data to write
 */
static __inline__ void reg1_write(const sb_uint32_t data)
{
  WRITE_REG32(TESTER_SLV1_REG,data);
}

/**
 * \fn void mem_write(const sb_uint32_t data)
 * \brief Write data to mem
 * \param[in] data The data to write
 */
static __inline__ void mem_write(const sb_uint32_t data)
{
  WRITE_REG32(TESTER_MEM,data);
}

/* PROTOTYPES */

/**
 * \fn void tester_put(void)
 */
extern void tester_put(void);

#endif /* _SB_TESTER_H */
```

File sb_tester.c

```
#include "sb_tester.h"

void tester_put(void)
{
  reg0_write(4294901760);
  reg1_write(2);
  mem_write(0);
  mem_write(2863311530);
  reg0_write(4294967295);
}
```

File main.c

```
#include "sb_tester.h"

int main(void)
{
  tester_put();
}
```