DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Dopo aver affrontato negli scorsi anni due argomenti fondanti dell'Informatica – il modo di codificare l'informazione digitale e la concreta possibilità di risolvere problemi mediante gli elaboratori elettronici – con questa terza serie andiamo ad esplorare "*Come parlano i calcolatori*".
La teoria dei linguaggi e la creazione di linguaggi di programmazione hanno accompagnato di pari passo l'evolversi delle architetture di calcolo e di gestione dei dati, permettendo lo sviluppo di applicazioni sempre più complesse, svincolando il programmatore dall'architettura dei sistemi e consentendogli quindi di concentrarsi sull'essenza del problema da risolvere.
Lo sviluppo dell'Informatica distribuita ha comportato la nascita, accanto ai linguaggi per l'interazione tra programmatore e calcolatore, anche di linguaggi per far parlare i calcolatori tra di loro – i protocolli di comunicazione. Inoltre, la necessità di garantire la sicurezza e la privatezza delle comunicazioni, ha spinto allo sviluppo di tecniche per "non farsi capire" da terzi, di quì l'applicazione diffusa della crittografia.
Di questo e di altro parleranno le monografie quest'anno, come sempre affidate alla penna (dovrei dire tastiera!) di autori che uniscono una grande autorevolezza scientifica e professionale ad una notevole capacità divulgativa.

Linguaggi per la progettazione dell'hardware

William Fornaciari, Mariagiovanna Sami

1. Introduzione

La crescita della complessità dei sistemi digitali negli ultimi 30 anni è stata possibile grazie alla messa a punto di metodologie di progetto in grado di automatizzare molte fasi realizzative. Un ruolo fondamentale è ricoperto dai linguaggi di descrizione dell'hardware, di cui il presente articolo traccia lo stato presente e le possibili evoluzioni, in relazione anche ai mutati scenari tecnologici. Ormai non esiste più una netta distinzione fra progettazione Hw e Sw: siamo nell'era della progettazione concorrente a livello sistema.

2. Realizzazione di un sistema digitale

La crescita esponenziale della "densità" dei circuiti digitali, rappresentata dal numero dei transistori su un *chip* (ben prefigurata dalla cosiddetta "legge di Moore") ha portato alla necessità di definire fasi e modelli del processo di progettazione e realizzazione che consentissero di dominarne la complessità (oggi su un unico chip si possono integrare dalle decine di milioni ai miliardi di transistor, in architetture che possono comprendere diversi microprocessori oltre a unità di elaborazione dedicate e a memorie, con geometrie del singolo transistor dell'ordine dei 45 nm). Tale strutturazione, esemplificata ad alto livello nella figura 1, ha diversi vantaggi. Innanzitutto consente di *specializzare* le competenze dei progettisti su alcuni passi soltanto dell'intero processo; grazie ad una *formalizzazione* dei modelli consente poi di usare rappresentazioni utili alla *automatizzazione* del processo di ottimizzazione; infine, ma non meno importante, consente di *simulare* il sistema a livelli di astrazione e di accuratezza variabili, in modo da garantire che il comportamento della realizzazione fisica finale sia effettivamente coerente con il progetto originale. A partire da una specifica del sistema, che verrà espressa con un opportuno linguaggio di descrizione o modello rappresentativo, si seguiranno trasformazioni che faranno evolvere la descrizione rendendola sempre meno astratta e sempre più vicina ad una formulazione finale sufficientemente dettagliata da potere guidare il processo

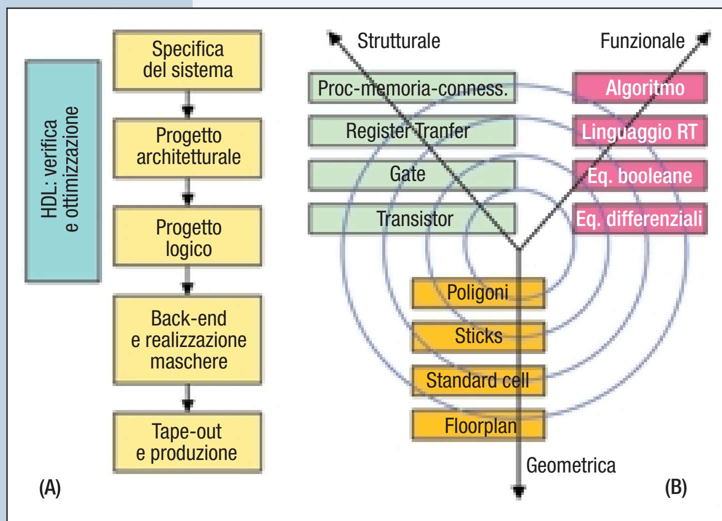


FIGURA 1

Passi del processo di progettazione di un sistema digitale (A), domini e livelli della modellazione e ottimizzazione (B). "Il cosiddetto "diagramma a Y" è stato introdtto da Gajski e Kuhn all'inizio degli anni '80"

di fabbricazione del silicio. Il percorso verso i livelli di astrazione più bassi non è una semplice traduzione, ma un processo di *design* complesso che prevede molte ottimizzazioni, (costo, velocità, potenza dissipata e tempo di progetto per citarne alcune) e, ad ogni passaggio, coinvolge attività di simulazione e verifica per garantire che il circuito continui a rispettare obiettivi e vincoli di progetto iniziali.

Per un sistema digitale, con buona approssimazione possiamo affermare che il processo di sintesi della macrofase detta *front-end* si arresta quando vengono identificati gli elementi digitali che compongono il sistema (esempio, porte logiche, memorie, registri) e le loro connessioni. A partire da questa descrizione inizia l'attività di *back-end*, che ottimizzerà ulteriormente il sistema cercando di disporre tali elementi in modo da usare la minore area di silicio possibile (floorplan) e arriverà a definire le caratteristiche geometriche e di drogaggio delle varie zone di silicio, informazioni che porteranno alla creazione di opportune "maschere" che saranno usate per la fase di produzione vera e propria dell'integrato. Oggi, una linea di produzione per un comune processo tecnologico digitale richiede in genere non meno di 4-5 settimane per la fase di *back-end* e di realizzazione di un insieme di prototipi su silicio, con un costo per la realizzazione delle maschere difficilmente inferiore al milione di euro. È quindi ovvio che l'identificazione e la messa a punto di eventuali errori dopo la fase di produzione in molti casi può significare il fallimento di un progetto; di conseguenza, molto sforzo è stato indirizzato verso lo sviluppo di strumenti di progettazione automatica (*Electronic Design Automation*, EDA), per consentire al progettista non solo di automatizzare molte fasi di sintesi ma anche (se non soprattutto) di effettuare una verifica approfondita e di lavorare il più possibile in modo *top-down*, pur mantenendo la possibilità di confrontare in tempi ragionevoli varie alternative di progetto.

Un progetto, come visto nella figura 1, può quindi essere visto a diversi livelli di astrazione, sia sotto il profilo *funzionale*, sia per quanto concerne la *struttura* e l'organizzazione dei blocchi che ne andranno a comporre l'architettura, sia per quanto attiene più da vicino le informazioni *geometriche* usate nel processo di produzione del silicio.

I linguaggi di descrizione dell'hardware (HDL), come il Verilog e il VHDL, hanno un ruolo fondamentale, perché hanno favorito la nascita di strumenti di *progetto automatico*, ridotto drasticamente i *tempi di progetto*, consentito la creazione di *figure professionali* con competenze sempre *meno verticali* e permesso la collaborazione di gruppi di progettisti a volte geograficamente distanti (passaggio pressoché inevitabile visti i tassi di crescita della complessità dei sistemi e delle loro tecnologie di realizzazione).

3. Linguaggi per la modellazione hardware: il VHDL

Per esemplificare l'organizzazione e l'utilizzo di un linguaggio per la descrizione dell'hardware, faremo riferimento al VHDL, che è il più diffuso in Europa. Il VHDL nasce negli USA nell'ambito del progetto VHSIC (*Very High Speed Integrated Circuits*) sponsorizzato dal Dipartimento delle Difesa (DoD) americano. La prima versione risale al 1984, la prima standardizzazione IEEE avviene nel 1987 e i successivi aggiornamenti sono stati compiuti negli anni 1992, 1997 e 2002. Gli obiettivi iniziali del progetto dovevano consentire al DoD di standardizzare le procedure di progettazione e documentazione degli apparati digitali e di svincolarsi da eventuali dipendenze da un particolare fornitore di sistemi elettronici. La struttura e l'utilizzo del linguaggio hanno forti similarità con un altro ben noto standard, il lin-

guaggio ADA, e si accompagnano ad una versatilità che gli consente di essere utilizzato sia per fare progettazione e verifica funzionale, sia come riferimento per l'intero processo di sintesi che porta alla realizzazione del sistema finale.
Nonostante l'apparente similarità con un normale linguaggio di programmazione, il VHDL ha particolarità che derivano dall'essere concepito per rappresentare sistemi hardware, dove non esiste un unico esecutore sequenziale delle operazioni e dove la sincronizzazione degli accessi alle varie unità non può essere implicitamente considerata come un dato di fatto. Le caratteristiche salienti di un HDL dotato di buona potenza rappresentativa, quale il VHDL, sono le seguenti:

- *Concorrenza*. Deve essere possibile rappresentare senza difficoltà l'intrinseco parallelismo dei sistemi hardware, così come, se necessario, forzare un ordinamento nell'esecuzione di gruppi di operazioni.
- *Astrazione*. Si vuole lavorare con diversi livelli di astrazione sia nella fase di specifica sia durante la simulazione.
- *Viste*. Si vuole operare con viste differenti dello stesso sistema, che possono per esempio avere le medesime interfacce ma avere poi rappresentazioni comportamentali piuttosto che strutturali.
- *Strutturazione*. Deve essere possibile rappresentare, per i diversi livelli di astrazione a cui si considera il sistema, i collegamenti fra i vari blocchi componenti. Si possono pertanto identificare gerarchie del sistema digitale; diviene possibile la progettazione modulare di sistemi complessi, grazie al supporto al riuso di unità di progetto o di librerie sviluppate anche da terze parti.
- *Tempo*. È necessario specificare precisi parametri temporali per rappresentare le caratteristiche dei sistemi reali che si otterranno dopo la sintesi, così come la sincronizzazione fra diversi sottosistemi, inclusa la gestione dei *clock*.

Nel caso del VHDL, c'è una separazione netta fra la specifica delle interfacce di un componente e il suo "corpo". Questa caratteristica, ereditata dal linguaggio ADA, consente di associare *ad una sola specifica* sia differenti viste, sia anche progetti alternativi da utilizzarsi in base alla fase del progetto o a specifiche esigenze applicative.
In VHDL ogni entità da modellare si chiama *design entity*, e si compone di una *entity declaration* e di una o più *architecture*. Per esemplificare l'uso del VHDL consideriamo un semplice sommatore completo (*full adder* nella Figura 2)

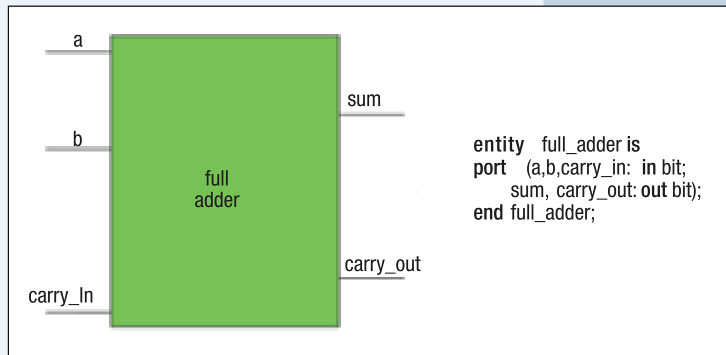


FIGURA 2
Descrizione in VHDL delle interfacce di un sommatore completo (entity declaration), dove si identificano le "porte" di ingresso e uscita

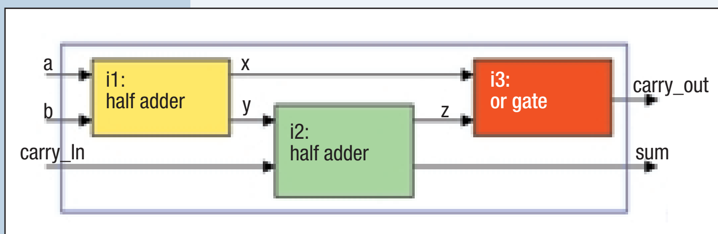
Riquadro 1

```
architecture behavior of full_adder is
begin
sum <= (a xor b) xor carry_in after 10 ns;
carry_out <= (a and b) or (a and carry_in) or
            (b and carry_in) after 10 ns;
end behavior;
```

che riceve in ingresso due bit addendi (a e b) e un eventuale riporto (carry_in) e produce il bit di somma (sum) e l'eventuale riporto in uscita (carry_out). La figura 2 mette in evidenza le interfacce del full adder che si traducono nella *entity declaration* riportata a destra.
L'architettura di una entity si compone di un *header* e di un *body* che può essere di tipo strutturale o comportamentale (behavioral). Una descrizione comportamentale fornisce informazioni sufficienti per il calcolo dei valori dei segnali di uscita a partire da quelli in ingresso oltre a eventuali informazioni di temporizzazione del circuito. Indicando con <= l'assegnamento di segnali, una possibile descrizione comportamentale del *full adder* è riportata nel riquadro 1.
Le informazioni di temporizzazione come quelle specificate con la parola chiave *after* sono utili in fase di simulazione per generare forme d'onda in risposta alla variazione degli stimoli di ingresso, tenendo conto della presenza nel circuito di ritardi reali e finiti per il calcolo delle uscite. Nel caso del riquadro 1, le uscite *sum* e *carry_out* saranno stabilizzate sul loro valore finale dopo 10 ns rispetto alla variazione degli ingressi.
Una rappresentazione del *full adder* di tipo *strutturale*, invece, focalizzerebbe l'attenzione sull'architettura di una realizzazione dello stesso circuito e non sulla sua funzionalità: si veda-

no la figura 3 e il riquadro 2, dove il *full adder* è realizzato connettendo due semisommatori (*half adder* i1 e i2) e una porta AND (i3).
I componenti debbono essere dichiarati all'interno della *component declaration*, che fornisce le informazioni salienti sul componente anche se la sua descrizione completa non è ancora presente nel database del progetto, come tipicamente avviene quando si adotta un approccio top-down alla progettazione. La corrispondenza fra le porte della entity e quelle dei componenti interni è descritta tramite il cosiddetto *port map*.
Così come col costrutto *after* si modella il ritardo nella generazione delle uscite, il VHDL consente di considerare i *livelli* (che sono un'astrazione dei valori di tensione) e la *forza* dei segnali (che invece modellano l'impedenza delle sorgenti elettriche) oltre all'eventuale indeterminatezza nei valori assunti da un segnale. In VHDL ogni uscita è associata ad un *driver*: qualora più uscite siano connesse fra loro, una funzione di *risoluzione* calcola il valore considerando i vari contributi con il loro valore e la loro forza[1]. Non meno importante, VHDL offre costrutti analoghi a quelli dei linguaggi di programmazione, come la selezione, *select/when*, la tipizzazione, i sottoprogrammi ecc. oltre a una raffinata gestione delle librerie di componenti.
La semantica del VHDL ha una formalizzazione che consente di realizzare *simulatori* pienamente deterministici, così da procedere a fasi di *verifica del progetto mediante simulazione*. Diverso è il discorso per gli strumenti di sintesi automatica, poiché ogni singolo produttore di strumento EDA, al fine di ottimizzare sia il progetto sia la complessità e l'efficienza dello strumento stesso, potrebbe far scelte che porterebbero – a partire dalla stessa descrizione VHDL – a risultati diversi (anche se funzionalmente equivalenti) a seconda dello strumento usato. Negli anni si è arrivati a definire un insieme di “buone pratiche” di specifica e di progetto per ottenere il massimo delle prestazioni dagli strumenti EDA.

FIGURA 3
Vista strutturale del full adder



Riquadro 2

```
architecture structure of full adder is
 component half_adder
  port  (in1 , in2: in bit ; carry: out bit ; sum: out bit) ;
 end component ;
 component or_gate
  port  (in1 , in2 : in bit ; o: out bit) ;
 end component ;

 signal  x , y, z : bit ;   - - segnali locali

 begin     - - connessione delle porte
  i1 : half_adder  port map (a, b, x, y) ;
  i2 : half_adder  port map (y,  carry_in, z, sum) ;
  i3 : or_gate  port  map  (x, y, carry_out) ;
 end structure;
```

4. Verso nuovi formalismi di descrizione

Molti sforzi sono attualmente legati alla messa a punto di strumenti che supportano il cosiddetto *IP-based design*, ovvero la possibilità di progettare rapidamente sistemi in modo non più verticale, ma utilizzando anche blocchi pre-progettati detti IP (*Intellectual Properties*) che possono essere eventualmente acquisiti da terze parti o sviluppati internamente allo scopo di essere facilmente riusabili. Allo stesso modo si sta cercando (con alterne fortune) di elevare il livello di astrazione dei formalismi per la descrizione dei sistemi, così da potere coprire sia le componenti hardware di un sistema sia quelle software. Probabilmente il linguaggio **SystemC** – una elaborazione del C++ nata nel 1999 e standardizzata da IEEE nel 2005 – è il rappresentante di tale filosofia con le maggiori probabilità di sopravvivenza[2]. SystemC, la cui architettura è mostrata nella figura 4, definisce una libreria di classi C++ in modo da fornire al progettista un insieme di elementi per effettuare modellazioni e simulazioni delle specifiche del sistema con l'accuratezza sino a livello di ciclo (cycle accurate). Le librerie SystemC contengono tutti quei costrutti per modellare sistemi, incluse temporizzazioni hardware, concor-

[1] Lo standard a cui si fa riferimento più comunemente, che assicura anche una ottima portabilità dei progetti, è quello IEEE 1164.

[2] Si veda www.systemc.org per maggiori informazioni circa tale iniziativa.

renza e comportamento reattivo che non sono nativi del C++. L'obiettivo è quello di consentire anche a chi ha competenze di sviluppatore esclusivamente software di muovere passi significativi nella progettazione di sistemi hw/sw complessi. La metodologia di progetto basata su SystemC fornisce la possibilità di utilizzare un unico formalismo per Hw e Sw, rimanendo al livello funzionale per la verifica del comportamento, e muovendo poi verso livelli incrementali di dettaglio sino a giungere al punto in cui esiste uno strumento di sintesi automatica. Purtroppo, almeno per ora, la mancanza di strumenti di sintesi con la maturità ed efficienza di quelli basati su VHDL porta i progettisti a ricorrere ai canonici flussi e strumenti commerciali per la sintesi di hardware reale. Ciononostante, la possibilità di creare modelli (dalla versione 2.0) con un livello di astrazione detto *transazionale* (TLM, *Transactional Level Model*), in virtù della efficienza in fase di simulazione e della potenza rappresentativa ne sta favorendo l'accettazione da parte dei progettisti. Uno stile di descrizione TLM porta a rappresentare i componenti di un sistema in modo affine all'invocazione di metodi remoti: vi sono moduli (SC_MODULE) che si interfacciano in modo più astratto rispetto al concetto di segnale, mediante invocazioni dirette (sc_port) di metodi realizzati da un modulo ed esportati (sc_export) tramite canali (sc_channel). Ovviamente, come accade per il VHDL, i costrutti utilizzabili per la sintesi delle parti hardware sono una restrizione rispetto a quelli messi a disposizione dal linguaggio. Per tale motivo la fase di sintesi prevede una serie di passaggi per raffinare la specifica così da renderla implementabile, ad esempio convertendo i tipi del linguaggio nella precisione effettivamente necessaria, eliminando le chiamate di sistema operativo, utilizzando solo costrutti per cui esiste semantica hardware ben definita (escludendo quindi allocazione dinamica della memoria, la ricorsione o l'uso del goto).
I vantaggi legati al livello di astrazione TLM sono particolarmente significativi quando si deve raffinare il modello del sistema, visto che si può iniziare l'analisi dagli aspetti puramente funzionali, rimandando la risoluzione delle questioni di dettaglio della comunicazione fra i moduli ad un momento successivo della progettazione, quando la sperimentazione di alternative è pressoché conclusa. Un semplice esempio di modellazione TLM in SystemC è riportato nel riquadro 3, dove si riporta un modello semplificato di bus che supporta operazioni *burst* di lettura e scrittura, senza entrare nei dettagli di aspetti più legati al mondo reale come l'arbitraggio, la risposta alle interruzioni o i *wait state* della memoria. Tali elementi legati all'attività del bus ad ogni ciclo di clock, non riportati per motivi di spazio nel testo, sono comunque rappresentabili rimanendo sempre al livello TLM della descrizione.
Nella sezione precedente si è accennato al linguaggio Verilog, alternativo al VHDL e più diffuso negli USA, molte caratteristiche del quale sono affini a quelle del VHDL ed il cui ruolo nella progettazione può essere considerato analogo. Le versioni a partire dalla 3.0 del 2003 sono note come **SystemVerilog** e contengono estensioni che ne ampliano potenzialità e livello di astrazio-

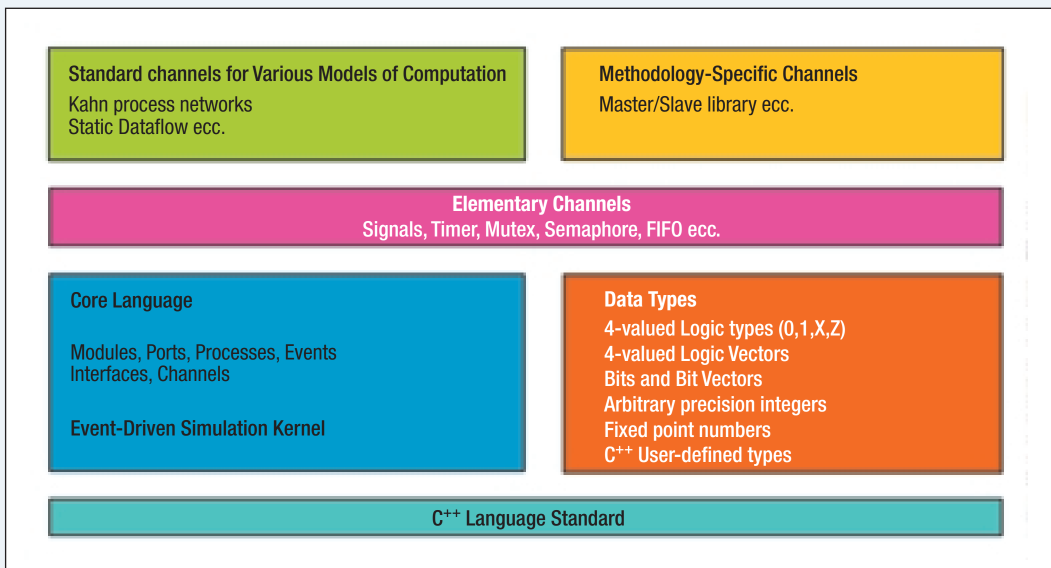


FIGURA 4
Architettura del linguaggio SystemC

Riquadro 3

```
class very_simple_bus_if : virtual public sc_interface
{
public:
      virtual void burst_read   (char  *data,
                        unsigned  addr,
                        unsigned  length)  = 0;
      virtual void burst_write   (char *data,
                        unsigned  addr,
                        unsigned  length)  = 0;
}

class very_simple_bus
      : public  very_simple_bus_if ,
      public  sc_channel
{
public:
      very_simple_bus(sc_module_name nm,  unsigned mem_size,
      sc_time  cycle_time)   :  sc_channel  (nm),
                                  _cycle_time  (cycle_time)
      {
// uso  di  un  array  per modellare l'accesso alla memoria
      _mem = new  char [mem_size] ;
// inizializzazione  a  zero  della  memoria
      memset  (_mem,  0,  mem_size) ;
      }

very_simple_bus () {delete [ ]  _mem ; }

virtual   void  burst_read (char *data ,  unsigned  addr,
                                 unsigned   length)
{
// uso di un mutex per modellare la contesa, ma senza arbitraggio

_bus_mutex.lock () ;

//  blocco  del chiamante  per  la  durata  del  burst
wait (length   *_cycle_time);
//  copia  dei  dati  dalla  memoria  a  chi  ha  fatto  la  richiesta
memcpy (data, _mem + addr ,  length);
//  sblocco   del  mutex  per  consentire  l'accesso  al  bus  ad  altri
_bus_mutex.unlock () ;
}

virtual  void  burst_write (char  *data,  unsigned   addr,
                                 unsigned length)
{
_bus_mutex. lock () ;
wait  (length    *_cycle_time) ;
//  copia   dati   dal   richiedente alla memoria
memcpy (_mem + addr, data, length);
_bus_mutex. unlock () ;
}

protected:
      char*  _mem;
      sc_time   _cycle_time;
      sc_mutex _bus_mutex;
} ;
```

ne: consentono per esempio di chiamare funzioni C/C^{++}, creare processi dinamicamente, standardizzare la comunicazione e sincronizzazione tra processi (inclusi i semafori) e predisporre una interfaccia standard per la verifica formale. La capacità di utilizzare anche il C/C^{++} - che dovrebbe rendere possibile l'interfacciamento verso modelli SystemC - un sistema di simulazione migliorato e l'apertura verso la verifica formale sono i pilastri principali dei fautori di tale linguaggio.

Bibliografia

[1] Ashenden P.: *VHDL Cookbook*. Testo liberamente disponibile in rete.

[2] Grotker T., Liao S., Martin G., Swan S.: *System Design with SystemC*. Kluwer Academic Publisher, 2002. ISBN: 1-4020-7072-1.

[3] Mentor Grahpics (EDA Vendor): www.mentor.com

[4] Synopsys (EDA Vendor): www.synopsys.com

[5] Lattice Semiconductor (produttore logiche programmabili): www.latticesemi.com

[6] Xilinx (produttore logiche programmabili): www.xilinx.com

[7] Altera (produttore logiche programmabili): www.altera.com

William Fornaciari si è laureato con lode in Ingegneria Elettronica (1989), ha svolto il Dottorato di Ricerca in Ing. Informatica e Automatica (1992), è stato ricercatore (1995) e dal 2001 è professore associato presso il Dipartimento di Elettronica e Informazione del Politecnico di Milano. Fra il 1993 e il 2005 è stato responsabile della embedded systems design unit (ESD) del centro di Ricerca CEFRIEL, di cui ora è mentor scientifico. È stato membro dello steering committee del VHDL user's group italiano e chairman dei primi workshop "VHDL Users Design Practice" svolti nel 1993 e 1994. Dal 1992 ricopre ruoli in committee di conferenze internazionali nell'ambito dei sistemi digitali e della progettazione a livello di sistema. È autore di oltre 100 pubblicazioni scientifiche, per cui ha ricevuto tre best paper awards (IEEE-ICONIP'95, IEEE-IJCNN'92 e IEEE-ICCD'98) e un Certification of Appreciation da parte della IEEE Circuits and Systems Society. I sui interessi scientifici sono legati alla progettazione di sistemi embedded, Hw-Sw codesign, Wireless Sensor Networks e sistemi a basso consumo di potenza.
E-mail: william.fornaciari@polimi.it

Mariagiovanna Sami è professore ordinario di prima fascia presso il Politecnico di Milano, nell'area dei Sistemi di Elaborazione. Dal 1987 al 1990 è stata Direttore del Dipartimento di Elettronica del Politecnico di Milano. È Direttore Scientifico del Master of Science in Embedded Systems Design presso l'Università della Svizzera Italiana a Lugano.
La sua attività di ricerca riguarda le architetture hardware dei sistemi digitali, con particolare riguardo alle metodologie di progetto di sistemi dedicati caratterizzati da elevate prestazioni, robustezza e basso consumo di potenza. È autrice o co-autrice di oltre duecento lavori scientifici in sede internazionale ed ha ricevuto alcuni premi per la sua attività di ricerca. È membro dell'Accademia Italiana delle Scienze (detta dei Quaranta).
E-mail: sami@elet.polimi.it