Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Ingegneria Informatica

Appunti di Progettazione di Sistemi Embedded

Autori:
Matteo Iervasi
Linda Sacchetto
Leonardo Testolin

Anno Accademico 2018–2019

Indice

Prefazione

Questa dispensa si basa sugli appunti di Linda Sacchetto e Leonardo Testolin durante il corso di *Progettazione di Sistemi Embedded* dell'anno accademico 2018/2019. Nonostante sia stata revisionata in corso di scrittura, potrebbe contenere errori di vario tipo. In tal caso potete segnalarli inviando una mail all'indirizzo matteoiervasi@gmail.com.

Matteo Iervasi

1 Introduzione

Citando Wikipedia, un sistema embedded, nell'informatica e nell'elettronica, identifica genericamente tutti quei sistemi elettronici di elaborazione digitale a microprocessore progettati appositamente per una determinata applicazione (special purpose), ovvero non riprogrammabili dall'utente per altri scopi, spesso con una piattaforma hardware ad hoc, integrati nel sistema che controllano ed in grado di gestirne tutte o parte delle funzionalità richieste.

Storicamente, sono nati prima i sistemi embedded rispetto ai sistemi *general purpose*, basti pensare ai grandi calcolatori degli anni '40. Essi infatti erano costruiti per un utilizzo specifico, anche se in quanto a dimensioni non erano di certo ristretti. Tuttavia il primo vero sistema embedded, in tutti i sensi, fu l'*Apollo Guidance Computer*, che doveva contenere una notevole potenza computazionale per il tempo in spazi ristrettissimi. La produzione di massa di sistemi embedded cominciò con l'*Autonetics D-17* nel 1961 e continua fino ai nostri giorni.

Non possiamo progettare i sistemi embedded come facciamo con i sistemi *general purpose*, perché abbiamo dei vincoli di progettazione e degli obiettivi differenti. Se ad esempio nei sistemi *general purpose* la ricerca si focalizza nel costruire processori sempre più veloci, nei sistemi embedded la CPU esiste solamente come un modo per implementare algoritmi di controllo che comunica con sensori ed attuatori, e diventa invece più interessante trovare processori che usano sempre meno energia.

I vincoli principali ai quali bisogna attenersi durante la progettazione di un sistema embedded sono:

- **Dimensione e peso**
  Si pensi ai dispositivi che devono poter essere tenuti in una mano
- **Energia**
  Molto spesso i dispositivi embedded devono funzionare con una batteria
- **Ambiente esterno ostile**
  Bisogna dover tenere conto di eventuali fluttuazioni di energia, interferenze radio, calore, acqua, ecc.
- **Sicurezza e operazioni *real time***
  Vi sono casi in cui è necessario che il sistema debba garantire sempre il funzionamento, oppure che garantisca un tempo costante per ogni operazione
- **Costi contenuti**
  Oltre a tutto il resto, bisogna tenere i costi bassi altrimenti si rischia di non poter vendere il prodotto

2 Modellazione dei sistemi embedded

Nel momento in cui ci accingiamo a pensare a come si progetta un sistema embedded, salta subito alla mente un problema, ovvero come faccio a verificarne il corretto funzionamento? Quando progettiamo del software, abbiamo a disposizione una miriade di strumenti per assicurarci di tenere il numero dei bug il più basso possibile: *debugger*, *unit testing*, *analisi statica*, ecc. In hardware invece non possiamo certamente metterci a rifare tutto ogni volta che sbagliamo, ricordiamoci che dobbiamo tenere i costi bassi! Si pone quindi il problema della *simulazione*, strumento fondamentale per la verifica del nostro sistema. Spesso infatti l'architettura di riferimento è differente da quella del calcolatore che usiamo per lo sviluppo (caso tipico: noi sviluppiamo su architettura X86 per un'architettura di destinazione ARM).

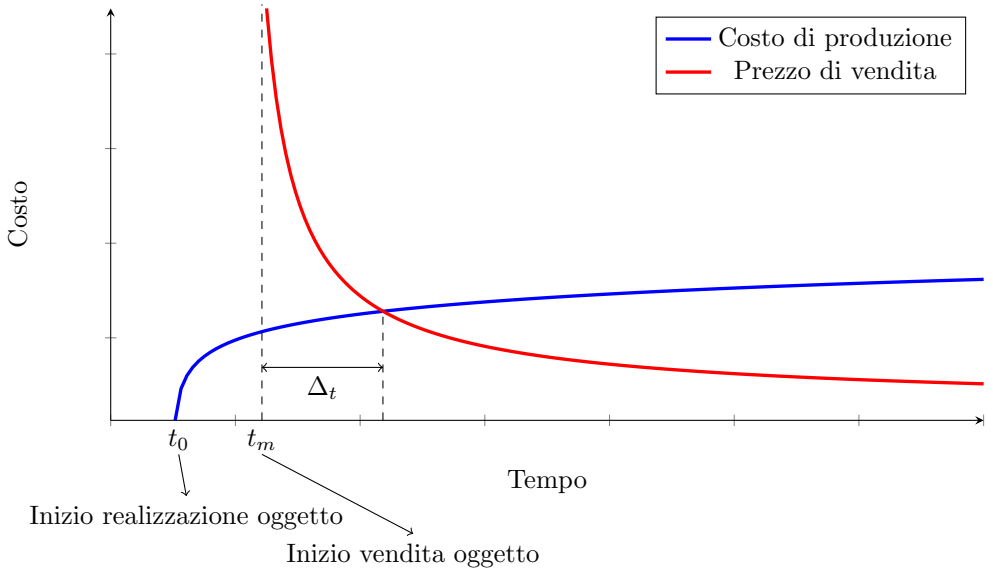
In generale, un sistema embedded è costituito dalle seguenti componenti:

- **Piattaforma hardware**
  Oltre al microprocessore, vi sono una serie di altre componenti.
- **Componenti software**
  Il software è monolitico: quando accendo il sistema, si esegue in automatico. Possono anche esserci casi in cui è necessario caricare un intero sistema operativo, e nella maggioranza di essi si fa riferimento a Linux.
- **Componenti analogiche** (es. sensori e trasduttori)
  Naturalmente se il nostro sistema dovrà interagire con l'ambiente esterno dovremmo introdurre componenti analogiche.

Il trend attuale è di portare tutto ciò in un singolo chip (SoC - *System on a Chip*), dove microprocessore, memoria e altre componenti vengono montate su un singolo chip, collegate da un bus. Esempi di SoC moderni sono i Qualcomm® Snapdragon o i Samsung® Exynos. Esiste anche un'alternativa, dove le componenti invece di trovarsi in un unico chip si trovano in una singola board (SoB - *System on a Board*).

Queste due tecnologie hanno uno scopo in comune: indurre al riutilizzo di componenti già esistenti per ridurre il più possibile il *time to market* (figura 2.1), ovvero il tempo che trascorre dall'inizio della progettazione all'immissione nel mercato. Più questo tempo è lungo, meno probabilità si ha di riuscire a vendere il prodotto in quantità tale da coprire i costi di sviluppo. Per questo motivo non si può minimamente pensare di sviluppare un sistema embedded partendo da zero, in quanto il tempo di sviluppo sarebbe improponibile.

Figura 2.1: Rappresentazione del rapporto costo produzione - guadagno



2.1 Co-design di sistemi embedded

Viste le ristrettezze imposte sui tempi dal mercato, è necessario ricorrere al co-design di hardware e software. Si parte quindi con una descrizione generale del sistema, magari aiutandosi con un prototipo, dopodiché fatte le verifiche su di esso si procede con la separazione di hardware e software, ricordandosi che è necessario "riciclare" il più possibile le parti già esistenti, si comincia lo sviluppo o l'eventuale adattamento. Durante la progettazione della parte hardware, si può decidere di affiancare l'eventuale processore da un co-processore, che svolge un compito specifico a seconda di cosa stiamo progettando. Il co-processore può essere anche una GPU.

Le tecnologie più usate per la progettazione di hardware sono:

- Microcontrollore standard o microprocessore
- ASIC (con o senza co-processore a seconda dei casi)
- FPGA

mentre per il software si possono utilizzare diversi linguaggi di programmazione (generalmente però si scelgono C/C++).

Naturalmente ci serve uno strumento per la verifica del nostro sistema, ci serve quindi co-simulare hardware e software. Si può fare su diversi livelli, ognuno con i suoi pregi e i suoi difetti.

- **Gate level**
  Viene simulato il tutto a livello di porte logiche, ovvero il più basso livello possibile. Questa è la simulazione più lenta in assoluto.
- **RTL (Register Transfer Level)** La rappresentazione in questo caso è vista come un flusso di informazioni che percorre il sistema, i cui risultati vengono salvati nei registri. Anche questa simulazione è lenta poiché molto vicina all'hardware. Essa può essere *cycle accurate*, nella quale l'unità minima di elaborazione è il ciclo di clock dove è considerato come importante quello che avviene all'inizio e alla fine di esso, oppure può essere *instruction accurate*, nella quale l'unità minima diventa la singola istruzione.
- **Behavioural**
  Questa rappresentazione descrive le funzionalità facendo una stima dei cicli di clock che impiega
- **Transactional**
  In questo caso non ho nemmeno il dettaglio della funzionalità, ma vado solamente a descrivere le interazioni tra i singoli moduli hardware. La simulazione in questa modalità è rapida.

Ogni livello di descrizione ha dei linguaggi più adatti di altri nonostante siano stati fatti diversi sforzi di crearne uno adatto a tutti. A livello RT, quelli più diffusi sono *VHDL* e *Verilog*. *SystemC*, nonostante sia in grado di scrivere più o meno bene a livello RT, mostra la sua potenza espressiva agli altri livelli.

2.2 Hardware description languages

Gli *Hardware Description Languages* (HDL) sono nati per risolvere una serie di problemi. Prima di essi l'hardware veniva progettato a mano e senza alcuna procedura standardizzata. Questo approccio però è prono ad errori e soprattutto incompatibile con le tempistiche richieste al giorno d'oggi. Quando progetto del software mi basta pensare all'algoritmo astratto e codificarlo in un preciso linguaggio di programmazione. Può anche succedere che il linguaggio che utilizzo sia multipiattaforma, per cui non mi dovrò minimamente preoccupare di dove e come verrà eseguito, poiché so che a prescindere dall'architettura sul quale verrà eseguito darà sempre lo stesso risultato. Quando progettiamo hardware invece non possiamo permetterci questo lusso in quanto l'architettura semplicemente non c'è, siamo noi a costruirla.

Prendiamo in esempio un programma scritto in linguaggio C e osserviamo le assunzioni che facciamo senza nemmeno pensare.

```c
#include <stdio.h>

int gcd(int xi, int yi){
     int x, y, temp;

     x = xi;
     y = yi;
     while(x > 0){
       if(x <= y){
         temp = y;
         y = x;
         x = temp;
       }
       x = x - y;
     }
     return(y);
}

int main(int argc, char *argv[]){
     int xi, yi, ou;

    scanf("%d %d", &xi, &yi);
    ou = gcd(xi, yi);
    printf("%d\n", ou);

    return 0;
}
```

I requisiti hardware per poter eseguire questo programma sono:

- **Input/Output**
  SW: `printf`, `scanf`, ...
  HW: interfacce di I/O
- **Temporizzazione**
  SW: istruzioni vengono eseguite alla velocità del ciclo di clock
  HW: devono essere definiti uno o più segnali di clock (e le istruzioni possono impiegare diversi cicli di clock)
- **Dimensioni variabili**
  SW: dimensioni implicite sono nascoste
  HW: devo tenere conto delle dimensioni, in quanto sto creando qualcosa di fisico che poi corrisponderà alla variabile
- **Operazioni**
  SW: esistono librerie per ogni tipo di operazioni

HW: difficili dato che devo fare un circuito apposito. Se poi vogliamo anche i numeri in virgola mobile, il circuito aumenta molto in complessità

- **Identificazione elementi di memoria**
  SW: non guardo se una variabile va nei registri o nella RAM, la uso e basta
  HW: devo sapere che spazio andrà a occupare, c'è una bella differenza tra registro e memoria

- **Sincronizzazione moduli**
  SW: spesso lavoro in maniera sequenziale
  HW: per com'è strutturato, l'hardware lavora tantissimo in parallelo

Quando scriviamo l'algoritmo software facciamo quindi molte assunzioni, del tutto legittime, ma che non possiamo di certo dare per scontato quando invece progettiamo un modulo hardware.
La prima cosa di cui bisogna preoccuparsi è definire le porte d'ingresso e di uscita. Una volta completata l'identificazione delle porte, bisogna individuare la modalità con cui andremo a progettare il modulo, che nel nostro caso sarà una FSMD, ovvero una macchina a stati finiti combinata con un datapath. Nella definizione della FSM e del DP, bisogna tenere conto degli eventuali vincoli: se ad esempio mi interessa un circuito veloce posso permettermi un DP più grande, mentre se voglio che il mio circuito occupi il minor spazio possibile dovrò allargare la FSM. Riportiamo di seguito un possibile diagramma del modulo `gcd`.

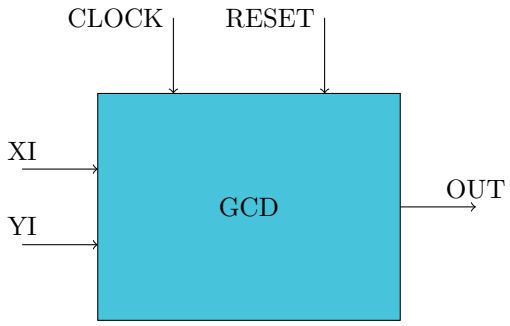
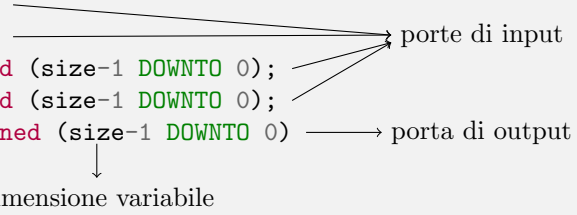


Figura 2.2: Schema di un modulo GCD

Questa è una possibile implementazione in VHDL del modulo GCD, descritta dal punto di vista ingressi-uscite:

```
ENTITY gcd IS
    PORT (
        clock : IN bit;
        reset : IN bit;
        xi : IN unsigned (size-1 DOWNTO 0);
        yi : IN unsigned (size-1 DOWNTO 0);
        out : OUT unsigned (size-1 DOWNTO 0)
    );
END gcd;
```



mentre questa è la descrizione comportamentale del modulo hardware:

```
ARCHITECTURE behavioral OF gcd IS
BEGIN
    PROCESS
        VARIABLE x, y, temp : unsigned (size-1 DOWNTO 0);
    BEGIN
    WAIT UNTIL clock = '1';
    x := xi;
    y := yi;
    WHILE (x > 0) LOOP
        IF (x <= y) THEN
            temp := y;
            y := x;
            x := temp;
        END IF;
        x := x - y;
    END LOOP;
    ou <= y;
    END PROCESS;
END behavioral;
```

3 SystemC

SystemC è un insieme di classi e macro del linguaggio C++ che forniscono un ambiente di simulazione *event-driven*. Queste classi permettono al progettista di simulare processi concorrenti, che possono anche comunicare in un ambiente real-time simulato, utilizzando segnali di qualsiasi tipo forniti da C++/SystemC o dall'utente. Sebbene sia per certi versi simile a linguaggi come VHDL o Verilog, è più corretto definire SystemC un linguaggio di modellazione di sistemi. Lo standard è definito dalla *Open SystemC Initiative* (OSCI), ora *Accellera*, ed è stato approvato dall'IEEE. Le caratteristiche salienti del SystemC sono:

- **Concorrenza**
  Processi sincroni e asincroni
- **Comunicazione**
  IPC tramite segnali e canali
- **Nozione di tempo**
  Possibilità di avere cicli di clock multipli con fasi arbitrarie
- **Reattività**
  Possibilità di attesa su eventi
- **Tipi di dato hardware**
  Vettori di bit, interi a precisione arbitraria, ecc.
- **Simulazione**
  Kernel di simulazione incluso nella libreria
- **Debugging**
  Possibilità di utilizzare i debugger disponibili per C/C++ come GNU GDB

Quando si progettano sistemi complessi, viene naturale dividere il progetto in sotto parti, che chiamiamo *moduli*, ognuno dei quali svolge una specifica funzione e comunica con altri. In SystemC i moduli sono rappresentati nientemeno che da delle classi C++ e si specificano con la keyword `SC_MODULE`. Un modulo contiene la definizione delle porte di input e di output, i segnali interni con la loro eventuale inizializzazione e i sottomoduli, che sono rappresentati nella loro forma minima dalle funzioni. Inoltre ogni modulo contiene un metodo costruttore, identificato dalla macro `SC_CTOR`, che contiene la dichiarazione di tutti i processi contenuti nel modulo e la sensitivity list associata a tali metodi, che specifica i segnali ai quali ciascun metodo deve reagire.

I processi che vengono dichiarati all'interno di ogni modulo possono essere di tre tipi:

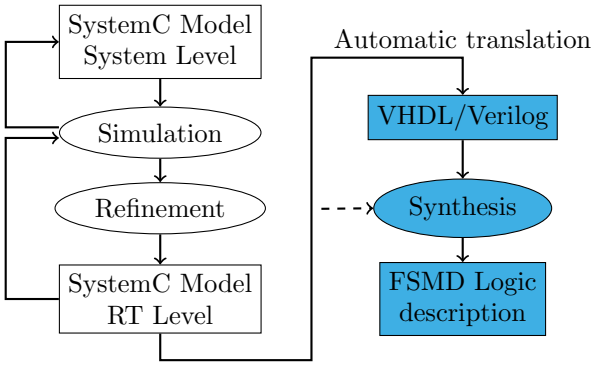


Figura 3.1: SystemC design flow

- **Metodi**
  Sono dei processi che quando vengono attivati ogni volta che arriva uno dei segnali espressi nella sensitivity list. Si identificano con la keyword *SC_METHOD*.
- **Thread**
  Sono dei processi che possono essere attivati o sospesi, mediante la funzione `wait()`. A differenza dei metodi, possono essere eseguiti una volta sola durante la simulazione. Si identificano con la keyword *SC_THREAD*.
- **Clocked threads**
  Sono dei processi sensibili al segnale di clock. Sono stati dichiarati obsoleti.

La simulazione in SystemC è gestita dal kernel, che gestisce lo scheduling nel seguente modo:

⇒ Tutti i segnali di clock vengono aggiornati

⇒ Tutti i processi sensibili ad esso vengono attivati

⇒ Si aumenta il tempo di simulazione di 1

Il segnale di clock viene generato normalmente da un thread che definisce anche il periodo del segnale. Oltre a poter eseguire una simulazione *event-driven*, è anche possibile farne una *cycle accurate*. La differenza sta nella velocità e nella precisione della simulazione, infatti in quella *cycle accurate* si osservano i cambiamenti ad ogni cambio di fronte, senza però preoccuparsi di osservare i cambiamenti che avvengono durante il ciclo.

3.1 SystemC RTL

Come detto precedentemente, in SystemC si possono descrivere sistemi hardware e software a diversi livelli di astrazione, rendendo possibile il co-design. In particolare,

si può descrivere a livello RT o TLM. Nel *Register Transfer Level* si può descrivere il funzionamento di un circuito digitale in termini di segnali, registri e operazioni logiche.

SystemC mette a disposizione dei tipi predefiniti, utili ad identificare:

- `sc_int<n>` e `sc_uint<n>`
  Con questi rappresento un valore intero con o senza segno.
- `sc_bigint<n>` e `sc_biguint<n>`
  Con questi rappresento un valore intero molto grande, con o senza segno.
- `sc_bit`
  Rappresento un singolo bit di informazione.
- `sc_logic`
  Tipo a 4 valori: 0, 1, `unknown` e `don't care`.
- `sc_bv<n>` e `sc_lv<n>`
  Vettore di bit e vettore di valori logici.
- `sc_fixed` e `sc_ufixed`
  Servono per rappresentare valori in virgola fissa.
- `sc_fix` e `sc_ufix`
  Alias per `sc_fixed` e `sc_ufixed`.

Per quanto riguarda le porte invece vi sono:

- `sc_in<PORT_TYPE>`
  Identifica una porta di input
- `sc_out<PORT_TYPE>`
  Identifica una porta di output
- `sc_signal<PORT_TYPE>`
  Codifica un segnale utile per collegare le porte (che nella sintesi molto probabilmente diventerà un filo)

Analizziamo la codifica RT tramite un esempio, nel quale implementiamo uno shifter a 8 bit.

shifter.h

```
#include <systemc.h>
#define N 8

SC_MODULE(shifter) {
  sc_in<bool> ds;
  sc_in<sc_bv<N>> a;
  sc_in<bool> i0;
  sc_out<sc_bv<N>> o;

  void shift();

  SC_CTOR(shifter) {
    SC_METHOD(shift);
    sensitive << ds << a << i0;
  }
};
```

shifter.cpp

```
#include "shifter.h"

void shifter::shift() {
  bool ds1;
  bool i01;
  sc_bv<N> a1;
  sc_bv<N> c1;

  i01 = i0.read();
  ds1 = ds.read();
  a1 = a.read();

  if (ds1 == 1) {
    c1.range(N - 2, 0) = a1.range(N - 1, 1);
    c1[N - 1] = i01;
  } else {
    c1.range(N - 1, 1) = a1.range(N - 2, 0);
    c1[0] = i01;
  }
  o.write(c1);
}
```

3.2 SystemC TLM

Anche se in SystemC è possibile progettare hardware a livello RT, quello che lo ha reso interessante rispetto a VHDL/Verilog è la capacità di poter fare *platform based design*. Una *piattaforma* è un'architettura hardware e software complessa generica, adattabile secondo

3.3 SystemC AMS

4 VHDL