Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Laurea Magistrale in Ingegneria e Scienze Informatiche

PSE

Progettazione di Sistemi Embedded
riassunto dei principali argomenti

Autore:
Danzi Matteo
Matricola VR424987

Anno Accademico 2017/2018

Indice

1 Sistemi Embedded - Introduzione

Definizione: Sistema di Elaborazione specializzato, integrato in un dispositivo fisico in modo tale da controllarne le funzioni tramite un apposito programma SW dedicato.

1.1 Storia

- **Computer (’60 - ’80)**: sistemi general purpose.
- **Sistemi di controllo digitale (’80 - ’90)**: dedicati al controllo e automazione.
- **Sistemi distribuiti (’90 - ’00)**: sistemi general purpose e/o dedicati che cooperano attraverso la rete.
- **Sistemi Embedded (’00 -)**: sistemi distribuiti integrati in oggetti non computazionali e nell’ambiente fisico.
- **Sistemi Ciber-fisici (’10 -)**: sistemi embedded integrati con processi fisici.

1.2 Smart System

Definizione: Sistema Embedded, integrato in una componente nel silicio, che controlla il mondo fisico. Uno smart system ha le seguenti caratteristiche:

- Miniaturizzato - Autosufficiente (autonomo dal p.d.v. energetico)
- Incorpora funzioni di sensore, attuatore, controllore
- Descrive, analizza situazioni, prende decisioni in base ai dati raccolti.
- Ha un comportamento predittivo e attuativo.

Pretective Maintenance: consiste nel riempire la catena di produzione con sensori che controllano e monitorano, cercando di prevenire e predirre guasti in base ai dati rilevati.

2 Come progettare

Per progettare un S.E. non si seguono le stesse direttive e gli stessi principi con cui si progettano sistemi distribuiti general purpose. Ad esempio per quest’ultimi si tende a costruire CPU sempre più veloci, mentre nei sistemi embedded la CPU esiste come mezzo di implementazione di algoritmi di controllo che comunicano con sensori e attuatori.

2.1 Vincoli di progettazione

Durante la progettazione di S.E. ci sono i seguenti vincoli:

- Dimensioni: hand-held electronics (alla portata di mano).
- Peso, potenza: utilizzo di batterie anziché corrente.
- Budget.
- Resistenza del sistema alle condizioni avverse in cui deve operare.

Nella progettazione sono da unire le proprietà di **Calcolo, Controllo, Comunicazione**.

3 Modellazione di Sistemi Embedded

I sistemi elettronici consistono di:

- Piattaforma HW: board di sviluppo;
- Strati di applicazioni SW: in nessun contesto posso permettermi di progettare software da zero, devo riusare parti di software progettate da altri possibilmente opensource;
- Interfacce;
- Componenti analogiche: hano fatto la differenza tra la potenza di calcolo di smartphone della vecchia generazione e quelli di adesso che sono equipaggiati con componenti come accelerometro, riconoscimento impronte, giroscopio ecc.;
- Sensori e trasduttori.

Principalmente si tende a spostare tutta l'elaborazione da analogica a digitale, inoltre si vuole una più ampia integrazione a livello di sistema per supportare l'approccio System-On-a-Chip (SOC). Questo approccio consiste nell'integrare in un chip tutto ciò che normalmente si vede in una board di un componente hardware. Il downside di questo approccio è rappresentato dal fatto che una volta effettuata l'integrazione sul chip non si può più cambiare il componente.

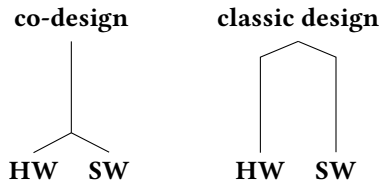
3.1 Sfide nella progettazione di sistemi embedded

- Aumentare la complessità delle applicazioni anche in prodotti standard e di grandi dimensioni
- Aumentare la complessità dei sistemi target

4 Co-design di Hardware/Software

Consiste nella progettazione combinata della parte hardware e della parte software. Gli obiettivi principali del codesign sono:

- Ottimizzazione del processo di progettazione: aumento della produttività
- Ottimizzazione del design: aumento della qualità del prodotto



I compiti del co-design sono:

- co-specification e co-modeling
- co-verification
- co-design process integration e ottimizzazione
- design optimization e co-synthesis

4.1 Vantaggi del co-design

- Permette di esplorare diverse alternative di progettazione nello spazio di progettazione architetturale.
- Permette di adattare l'Hardware al Software e viceversa.
- Riduce il tempo di progettazione del sistema.
- Supporta una coerente specifica del progetto a livello del sistema.
- Facilita il riuso delle parti HW e SW.
- Permette di provvedere di un ambiente integrate per la sintesi e la validazione delle componenti HW e SW.

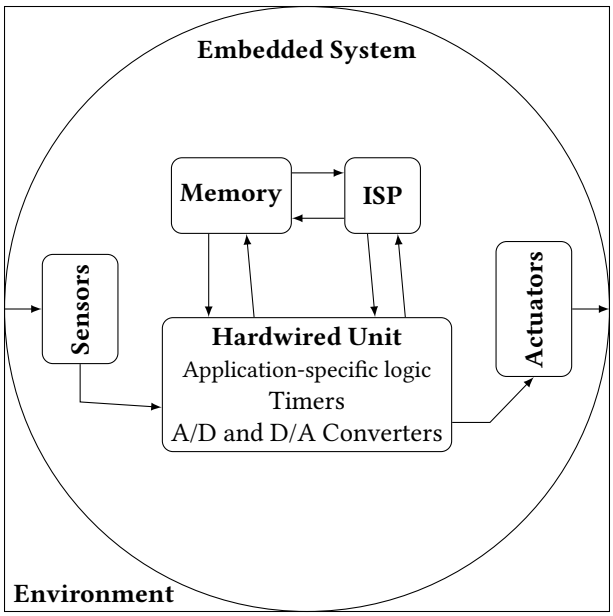
4.2 Co-design di sistemi embedded

Co-progettazione, realizzare un sistema embedded portando avanti il più possibile in contemporanea la sua progettazione hw e sw.
Abbiamo un minor spreco di tempo, è più semplice passare da una descrizione hw a una sw (più facile scegliere cosa diventerà hw e cosa sw), rende più agevolato il riuso di parti sw e hw e rende possibile la co-validazione e la co-simulazione.
Progettazione dedicata di parti HW può comprendere:

- Diversi stili di progettazione: co-processori, embedded core, Application Specific Instruction Processor (ASIP).
- Una scala di progettazione ampiamente variabile.

Progettazione dedicata di parti SW può comprendere:

- Sistemi operativi specializzati (special-purpose).
- Dirver di dispositivi periferici.



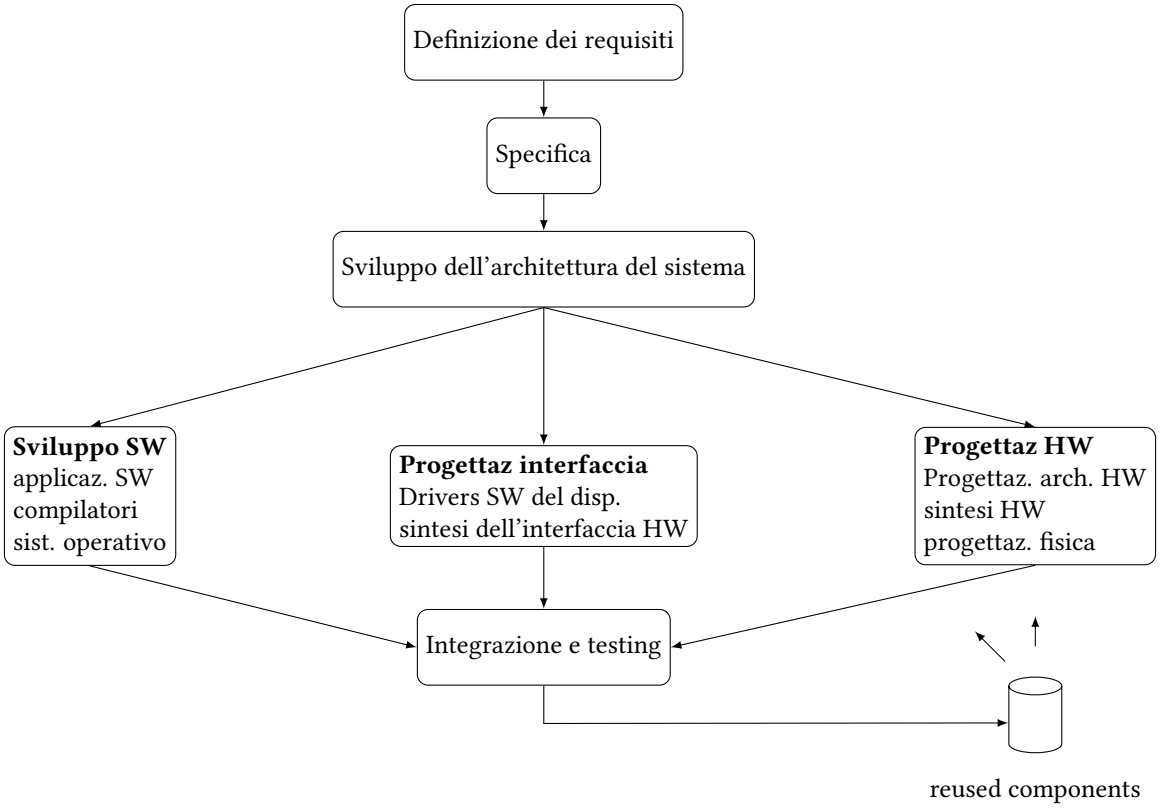
4.3 Array-based design (design basto su array)

ci sono due tipi di dispositivi che si possono progettare con questa tecnica di progettazione:

- **MPGA**: Pre-Diffused Array o più comunemente Mask Programmable Gate Array.
- **FPGA**: Pre-wired Array o più comunemente Field Programmable Gate Array. Sono dispositivi HW (circuito integrato digitale) generici che possono essere programmati con SW dedicato/specifico.

4.4 Flusso di co-progettazione

1. Modellazione, validazione e sintesi: simulazione a livello di sistema.
2. Modellazione omogenea:
   (a) Partizionamento HW/SW
   (b) Spostamento HW/SW o SW/HW
3. Modellazione eterogenea: implementazione diretta e reindirizzamento
4. Co-sintesi
   (c) Sintesi HW e dell'interfaccia
   (d) Compilazione del SW e generazione del codice
5. Co-simulazione



5 Requisiti per un sistema embedded

Un sistema embedded deve essere:

- **REATTIVO**: Non fermarsi mai ed essere sempre pronto a rispondere a segnali provenienti dall'ambiente esterno.
- **REAL TIME**: Rispettare vincoli temporali (HRT, SRT, FRT).

6 Time to market

Tempo che intercorre da quando un prodotto viene ideato a quando viene commercializzato, messo sul mercato per la prima volta.

7 Platform-based design (Progettazione basata sulla piattaforma)

La filosofia platform-based consiste nel progettare architetture HW (piattaforme) stabili, basate su microprocessore, che possano essere facilmente espanse a livello di componenti HW e che sono configurabili a livello SW; quindi sono piattaforme che nonostante abbiano componenti hw ben definite rimangono "a scopo generico" per la loro facile espansibilità, sia HW che SW.

- Dispositivi con gradi di configurabilità (? SW ?)
- Posso usare lo stesso oggetto per più applicazioni
- ASIC
- FPGA

Concetti principali a Livello di Sistema:
Concorrenza, Gerarchia dei moduli, Comunicazione tra sottosistemi, Sincronizzazione.

8 SystemC

SystemC è una libreria che estende il linguaggio di programmazione C++, linguaggio strettamente SW.

Permette di fornire una descrizione del sistema a diversi livelli di astrazione: un livello più vicino all'HW, in cui si definiscono i dettagli implementativi come porte, segnali e interfacce (RTL) e un livello in cui si definisce uno standard per la comunicazione tra i moduli e ci si concentra sulla funzionalità (TLM).

Tali astrazioni rendono possibile fare co-design, cioè una co-progettazione del sistema in cui si cerca di portare avanti il più possibile in parallelo la progettazione hw e quella sw; il co-design diminuisce il tempo di progettazione, rende più facile il passaggio da una componente hw a una sw (quindi la scelta di quale componente implementare come hw e quale come sw) e permette una co-simulazione e una co-validazione.

- Permette di fare co-design.
- Diversi livelli di astrazione (RTL-TLM).
- Linguaggio di definizione dell'hardware (HDL).
- Fornisce una descrizione hw a livello RT (RTL), dove ho un controllo sui segnali, sui registri e sugli operatori utilizzati. Descrivo un sistema con una FSMD.
- Le sue caratteristiche permettono di gestire gli aspetti più importanti per un sistema embedded -> gerarchizzazione dei moduli, concorrenza (processi sync/async), comunicazione (porte/segnali), timing (clock/fasi/tempo), reattività (eventi), tipi di dato hw (oltre ai tipi c++), kernel di simulazione, debug (c++).
- Ha introdotto una standardizzazione del modello transazionale (modo in cui si definiscono le transazioni tra moduli) attraverso il livello di astrazione TLM.
- Riuso di moduli/componenti già pronti messi in comunicazione grazie allo standard per le transazioni definito da SystemC.

9 SystemC RTL

Con SystemC RTL (Register Transfer Level) è possibile descrivere il funzionamento di un circuito digitale in termini di segnali, elementi di memoria dei segnali (registri) e di operazioni logiche tra segnali.

9.1 Moduli

I moduli sono i blocchi di costruzione di base per partizionare un design. Sono classi C++, permettono di partizionare sistemi complessi in componenti più piccole. Nascondono la rappresentazione dei dati interni, utilizzano interfacce. Contengono porte, segnali, dati locali, altri moduli, processi, costruttori e distruttori. Segue un esempio:

Listing 1: module

```
sc_module(module_name)
{
    // dichiarazione di porte
    // dichiarazione di segnali
    // costruttore del modulo: SC_CTOR
    //      SC_METHOD
    // creazione di sottomoduli e sensitivity list
    // inizializzazione dei segnali
}
```

Il costruttore del modulo:

- Inizializza e dichiara tutti i processi contenuti nel modulo e le regole per attivarli
- Inizializza le variabili ai valori di default oppure a valori definiti dall'utente.

Listing 2: full adder constructor

```
SC_CTOR( FullAdder )
{
    SC_METHOD( doIt );
        sensitive << A;
        sensitive << B;
}
```

9.2 Processi

I processi sono funzioni he sono identificate dal kernel di SystemC:

- I processi sono molto simili a funzioni C++ o metodi
- I processi implementano le funzionalità dei moduli
- Sono chiamate se un segnale della sensitivity list cambia il suo valore.

I processi possono essere **Metodi**, **Threads** o **CThreads (deprecated)**.

- **Metodi**:
  - Quando attivati, vengono eseguiti e ritornano.
  - **SC_METHOD**(process_name).
- **Threads**:
  - Possono essere sospese a riattivate.
  - La funzione `wait()` sospende.
  - Un evento sulla sensitivity list riattiva.
  - **SC_THREAD**(process_name)
- **CThreads (DEPRECATED)**:
  - Sono attivate all'incremento del clock
  - **SC_CTHREAD**(process_name, clock_value)

type	**SC_METHOD**	**SC_THREAD**	SC_CHTREAD
Attivaz. all'exec.	Evento	Evento	Incremento clock
Sosp. dall'exec.	No	Sì	Sì
Loop infinito	No	Sì	Sì
sosp/riattiv da	n.d.	`wait()`	`wait()` o `until()`

Costruttore e sensitivity def	**SC_METHOD**`(call_back);`
	`sensitive(signals);`
	`sensitive_pos(signals);`
	`sensitive_neg(signals);`
		SC_THREAD`(call_back);`
		`sensitive(signals);`
		`sensitive_pos(signals);`
		`sensitive_neg(signals);`
			SC_CTHREAD`(...)`

Tabella 1: Tabella riassuntiva

10 SystemC TLM

SystemC TLM (Transaction Level Modeling) è una libreria del C++ che permette di rappresentare i principali componenti architetturali di piattaforme hardware.

È uno standard per la comunicazione tra i moduli (sia HW che SW) di un sistema, attraverso transazioni con caratteristiche ben definite. L'aver definito uno standard per il modo in cui si interfacciano diverse componenti ha facilitato il riuso dei componenti, cioè la possibilità di integrare facilmente nel proprio sistema moduli provenienti da fonti esterne e di interfacciarli tra loro semplicemente conoscendo le specifiche del protocollo di comunicazione.

Il riuso dei componenti porta a grandi vantaggi anche nel cercare di diminuire il TTM, infatti posso risparmiare del tempo utilizzando componenti già implementate, di cui conosco le prestazioni e sono sicuro del loro funzionamento, piuttosto che ricrearle da zero e cercare di interfacciarle col mio sistema ad un livello più basso di astrazione.

(Con TLM è inoltre possibile fornire velocemente una descrizione SW del comportamento, in modo da poter proporre ad un eventuale cliente un primo prototipo del sistema con tempistiche brevi.)

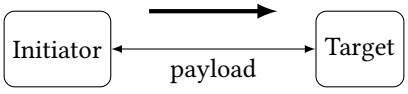
- Permette una esplorazione architetturale e modellazione delle performance.
- Consente l'esecuzione su modelli virtuali di piattaforme hardware.
- È disponibile prima dell'RTL.
- Viene simulato più velocemente dell'RTL.
- Consente di modellare sistemi a livello transazionale.

10.1 Transazione TLM

TLM si basa sul concetto di **transazione** cioè il trasferimento di dati tra moduli che comprende operazioni di scrittura/lettura. La transazione è rappresentata da un oggetto **payload**: viene scambiato con chiamate primitive, contiene sia dati che informazioni di controllo.
Gli attori della transazione sono:

- **Initiator**: fa partire la transazione tramite socket
  - Crea un oggetto transazione.
  - Chiama o si connette con il metodo target per inviare il payload.
- **Target**: agisce come destinazione finale della transazione. Elabora il payload.

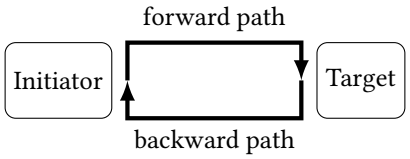
È un tipico comportamento master/slave:



10.2 Cammini di transazione

Il payload può seguire due diversi path:

- **Forward Path**: percorso di chiamata dall'initiator nella direzione del target. Il master chiama lo slave.
- **Backward Path**: percorso di chiamata attraverso cui il target ritorna indietro il payload nella direzione dell'initiator. Lo slave risponde al master.



10.3 Stili di programmazione TLM

In TLM 2.0 non ci sono livelli di astrazione standardizzati, come avveniva in TLM 1.0:

- Program View (PV)
- Program View with Time (PVT)
- Cycle Accurate (CA)

Questi livelli possono essere usati per esprimere diverse astrazioni dello stesso sistema.
In TLM 2.0 si è deeciso di standardizzare come il tempo e i dati sono collegati:

- **Untimed (UT)**: senza tempo.
  - Interfaccia bloccante.
  - Punti di sincronizzazione predefiniti.
- **Loosely Timed (LT)**:
  - Interfaccia bloccante.
  - Due punti di sincronizzazione (invocazione e return).
  - Temporal Decoupling
- **Approximately Timed (AT4)**: si dettaglia maggiormente il sincronismo tra Initiator e Target
  - Interfaccia transport non bloccante
  - Annotazione del tempo e fasi multiple durante il tempo di vita della transazione.
  - Protocollo 4-handshaking: inizio/fine richiesta, inizio/fine risposta.

UT è il livello più alto, **AT** è il livello più basso.

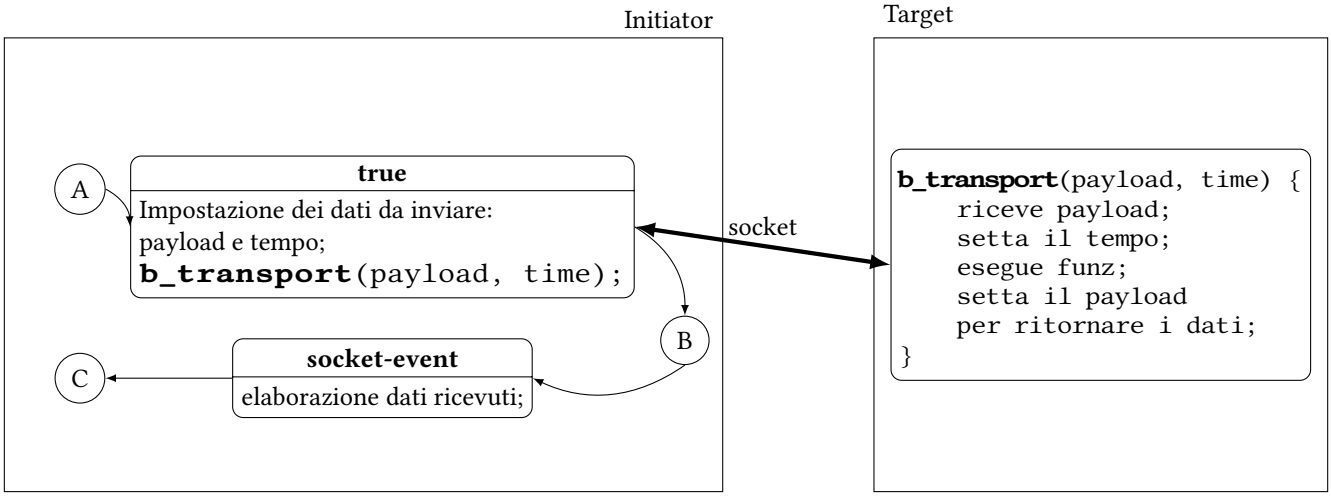
La comunicazione tra master e slave può essere *bloccante* (master rimane in attesa finché lo slave non ha finito i calcoli) o *non bloccante* (il master passa i dati allo slave ma può andare comunque avanti), in questo modo si può decidere il livello di parallelismo del software. Possono esserci componenti di interconnessione tra initiator e target, e servono per modellare bus astratti.
Se aumento i punti di sincronizzazione (arrivando a sincronizzare ogni ciclo di clock) posso arrivare ad avere una sincronizzazione cycle accurate.

Temporal decoupling

Initiator e target quando si sono de-sincronizzati possono andare avanti con tempi diversi e sono poi riallineati nel momento in cui si sincronizzano (comunicazioni non bloccanti), il più veloce aspetterà il più lento e si sincronizza. Il T.D. permette ai due moduli di evolvere insieme senza sincronizzarsi, rendendo più veloce la simulazione.

10.4 Interfaccia bloccante

Questa interfaccia è supportata da UT e LT. L'initiator completa la transazione con il target in una chiamata a funzione. Ci sono 2 punti di sincronizzazione: l'invocazione e il return. Utilizza solamente il **forward path**.



10.5 Interfaccia non bloccante

Questa interfaccia è supportata da AT4. Permette di dettagliare la sequenza di interazioni tra l'initiator e il target. È formata da due fasi:

1. Inizio/fine richiesta
2. Inizio/fine risposta.

Utilizza sia il forward che il backward path. Utilizza il classico protocollo di handshaking a 4 fasi.

10.6 Transactor TLM: (transattore)

- Modulo systemC che permette di far comunicare modelli a diversi livelli di astrazione (Es. RTL-TLM).
- Utile perché posso raffinare singolarmente prima i componenti e poi farli comunicare col transattore.
- Diversi livelli di astrazione: PV, PVT, AC.
- In PVT nel momento in cui devo temporizzare (non rispettando precisamente i cicli di clock) la comunicazione devo passare attraverso un bus (sia per moduli HW che SW) come AMBA (adottato da ARM).
- In PVT posso già decidere cosa far diventare HW (moduli che impiegano più tempo per essere calcolati) e cosa SW, e analizzare il traffico sul bus per unire per esempio moduli che hanno un alto traffico tra loro.
- In CA ho sempre bisogno di un bus ma che rispetti esattamente i cicli di clock.

10.7 Standard TLM 2.0

- Non ha senso standardizzare i livelli di astrazione. Si è passati da livelli di astrazione diversi a stili di codifica (LT e AT), lasciando al progettista la possibilità di decidere che tipo di flusso di progettazione seguire.
- La comunicazione avviene con un campo (payload) ben definito e standardizzato.
- TLM standardizza la comunicazione e non la funzionalità.
- **Initiator** fa partire la transazione, **Target** riceve la transazione (tramite socket)

- I cammini di transazione sono forward path e backward path
- Coding style NON sono specifici livelli di astrazione.
- Si possono definire coding style propri, ma potrei non riuscire a far comunicare moduli.
- In ogni transazione LT ci sono 2 punti di sincronizzazione (inizio e fine) e non ci sono molte informazioni sugli aspetti temporali.
- I processi hanno assegnati dei quanti di simulazione.
- Si spezza la sincronizzazione in più fasi, avendo più punti di sincronismo
- Per avere un modello UT prendo un modello LT e metto il tempo a 0.
- Per LT e AT4 si definiscono delle interfacce: bloccante (UT e LT) e non bloccante (AT4).
- Un altra interfaccia è la **DMI** usata per accedere ad aree di memoria condivisa tra Initiator e Target e modificarne i dati permette di far comunicare codici SystemC con codici di altri linguaggi (co-simulazione).

11 SystemC AMS

È una estensione del SystemC, permette la modellazione System-lever per sistemi con segnali analogici misti (Analog Mixed-Signal Systems).
Metodologie e casi d'uso per sistemi con segnali mixed:

- Specifiche eseguibili: verificare la correttezza dei requisiti del sistema utilizzando la simulazione.
- Prototipizzazione virtuale: modello ad alto livello (untimed/timed) dell'architettura hardware.
- Esplorazione di architetture: valutazione del mapping tra il comportamento e l'architettura di sistema.
- Validazione euristica integrata: verificare la correttezza delle componenti integrate.

11.1 Tempo discreto

Nelle descrizioni a tempo discreto:

- La funzione esiste solo in punti ben precisi nel tempo, negli altri non esiste e non posso valutarla è come se scrivessi una procedura. I segnali e le quantità fisiche sono definiti in punti di tempo discreti. Si assumono costanti in mezzo a questi punti.
- Vengono usate per rappresentare approssimazioni di rappresentazioni a tempo continuo, signal processing: rappresentazione di funzioni di elaborazioni di segnali (tipo filtri)

11.2 Tempo continuo

Nelle descrizioni a tempo continuo:

- La grandezza fisica che descrivo esiste in qualsiasi punto nel tempo come una equazione differenziale. I segnali e le quantità fisiche sono descritti nel come funzioni nei reali. Il tempo è considerato come un valore continuo.
- Si comportano come equazioni differenziali algebriche (DAE) o equazioni differenziali ordinarie (ODE). Vengono risolte da un solver lineare o non-lineare.
- Adatto per descrivere comportamenti fisici di sistemi dinamici,

11.3 Descrizioni conservative

Nelle descrizioni conservative:

- Il comportamento
- È tutto implementato in blocchetti di base che noi colleghiamo e che garantiscono in modo automatico che le leggi di conservazione di Kirchhoff vengano rispettate.
- ho vincoli sulla conservazione dell'energia le correnti d'entrata uguali a quelle di uscita
- Ci sono grandi quantità equazioni da risolvere, è molto dispendioso dal p.d.v. computazionale.

11.4 Descrizioni non conservative

Nelle descrizioni conservative:

- Il comportamento è espresso con flussi di segnali di tempo continuo o variabili. Vengono applicate funzioni di processing (filtering e integrazione).
- Può essere descritta la dinamica non lineare
- Non è supportata l'interazione tra componenti AMS.
- Le relazioni tra quantità non rispettano le leggi di conservazione dell'energia di Kirkhhoff.

11.5 Formalismi di modellazione

In SystemC AMS ci sono i seguenti formalismi di modellazione:

- **Time Data Flow (TDF)**: modelli a eventi discreti.
  - Tempo discreto, modellazione non conservativa
  - Scheduler statico, basato su dataflow
- **Linear Signal Flow (LSF)**: modelli a eventi continui.
  - Tempo continuo, modellazione non conservativa
  - Basato su DAE e ODE, rappresentate da connessioni di primitive per segnali reali nel dominio del tempo.
- **Electric Linear Network (ELN)**: per modellazione di reti elettriche.
  - Tempo continuo, modellazione conservativa
  - Primitive per reti lineari (resistenze, condensatori)
  - Relazioni continue tra voltaggio e corrente.

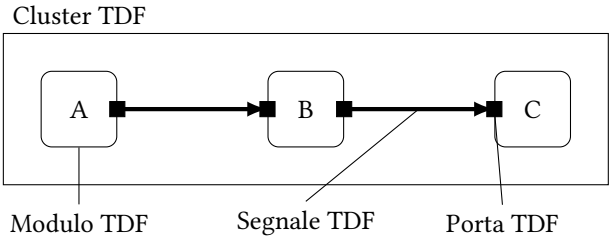
	Discreto (Scheduler Statico)	Continuo (Scheduler Dinamico)
Conservativo	$\emptyset$	**ELN**
Non-Conservativo	**TDF**	**LSF**

11.6 Time Data Flow (TDF)

Formalismo basato su Synchronous Data Flow (SDF), senza tempo, modello a eventi discreti in tempo discreto. Ogni modulo del modello a eventi discreti contiene un metodo C++:

- Elabora una funzione matematica, a seconda degli input. Può dipendere anche dai suoi stati interni.
- Composizione di moduli

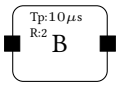
Esempio di composizione di funzioni: $f_C(f_B(f_A))$



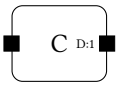
Data una funzione, questa viene eseguita se e solo se ci sono abbastanza sample disponibili nella porta di input. C'è un numero fissato di sample consumati e prodotti. Ogni sample ha il suo time stamp. L'intervallo fissato è chiamato time step.

- Time step (modulo)

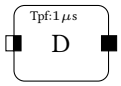
- Time step (porta)



- Rate (porta)
- Delay (porta)



- Time offset (porta specializzata)



11.7 Scheduling

Discrete Event Models Cluster: insieme di moduli di modelli di eventi discreti connessi che appartengono allo stesso scheduler statico:

- I parametri devono essere compatibili
- Definisce una sequenza in cui i moduli di modelli di eventi discreti vengono eseguiti

Nello scheduling i cicli possono essere fonte di problemi. Ogni loop deve presentare minimo una porta di delay. Le porte di delay possono portare all'inconsistenza, quindi bisogna specificare un valore iniziale.

TDF non supporta la gerarchia

Consistenza:

$$T_m = T_{p_{\text{input}}} \cdot R_{\text{input}} = T_{p_{\text{output}}} \cdot R_{\text{output}}$$

12 Sintesi di Software Embedded

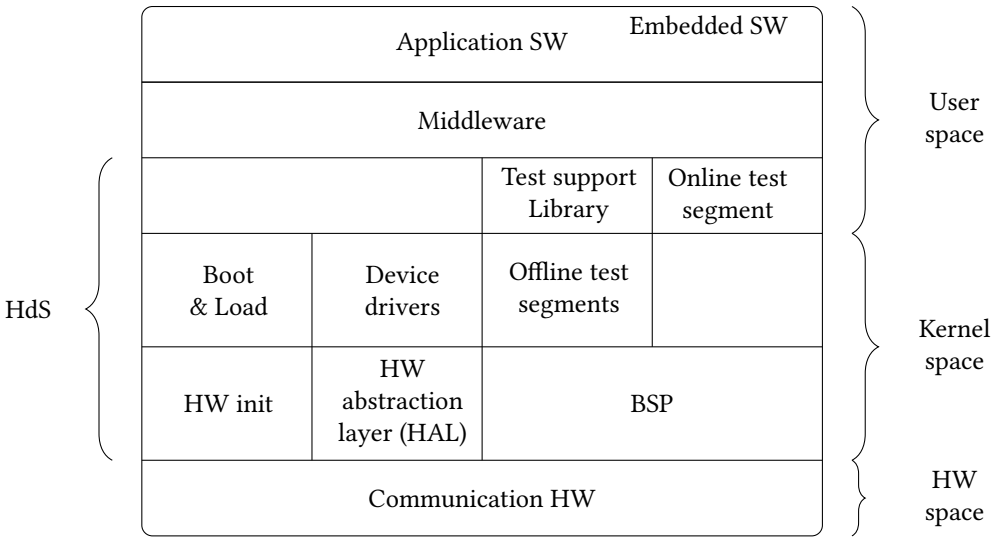
La sintesi di un software embedded è legata alla piattaforma target che deve avere un organizzazione semplice:

- Insieme di unità HW dedicate
- Porte programmabili
- Processore
- Memoria

Il software dovrà anche riuscire ad attivare l'esecuzione sui vari dispositivi hardware e ottenere risultati di tali parti. Gli algoritmi non sono mappati come processi ma come hardware, come avviene nei circuiti integrati ASIC (Application Specific Integrated Circuit).

12.1 Hardware Dependent Software

Possiamo considerare un software embedded come un **firmware** (metà tra hw e sw). Con questo termine in realtà si definiva il software che permetteva di realizzare microistruzioni all'interno dei processori negli anni '70 (microprogrammazione).
Il termine corretto non è firmware ma **HDS**: Hardware Dependent Software. Il software è hardware independent grazie ad uno strato hardware dependent. Questo strato generalmente viene fornito dal sistema operativo.



Non è detto che un sistema embedded abbia a disposizione un sistema operativo. In molti sistemi embedded infatti non c'è, se il software è realizzabile come un unico processo non mi serve un sistema operativo. Se invece ho più processi o librerie esterne devo averlo per implementare uno scheduler. Spesso nelle board ho solo una parte del kernel di linux. Se non c'è o.s., l'HDS deve comunque fare qualcosa:

- **HAL** (Hardware Abstraction Layer): contiene informazioni su dove i dispositivi hw sono mappati in memoria: mappa di indirizzi a cui non corrisponde RAM, bensì Hardware. Parte principale che permette di interfacciarmi con l'hw.
- **Device Drivers**: Se ho interrupt allora devo alzarmi nel livello di astrazione e avere a disposizione un device driver e primitive che si interfacciano con un dato dispositivo leggendo e utilizzando gli indirizzi a cui sono associati gli hw. Le primitive sono:
  - OPEN
  - READ : è solo una lettura su un indirizzo di memoria
  - WRITE
  - CLOSE (chiude la comunicazione)

- GENERIC IO : fasi specifiche del dispositivo hw (usata solo in casi particolari)

Scrivere un device driver significa scrivere le sopracitate funzioni.
La difficoltà di un device driver non risiede quindi sulla complessità delle primitive ma sulle scelte progettuali:

- bloccante o non bloccante
- Come gestire sincronismo per evitare che i dati di uno non finiscano nell'altro?

Il sistema operativo mette a disposizione una serie di funzionalità che facilitano tutto ciò. Se non ho un sistema operativo generalmente ho solo processi.

- **Boot and Load**: ogni processore ha una propria strategia di boot. Devo caricare in memoria il codice specifico per un dato processore (questa componente è generalmente venduta con il processore: non si deve scrivere).
  Il compilatore produce software sapendo qual è la mappa di memoria che si aspetta il processore per funzionare. Il linguaggio usato di riferimento è il C o C++. Posso scrivere programmi in linguaggio ad alto livello ma ho bisogno del compilatore specifico e di un o.s. (la complessità aumenta).
- **Middleware**: libreria specifica che mette a disposizione funzionalità per una particolare classe di problemi.

Le porte di HDS e software applicativo comunicano tra loro. Linux è il software embedded più utilizzato perché non ha licenze ed è aperto. È adattabile a tutte le piattaforme.
Linux inoltre è composto da tante parti. Posso prendere parti che mi servono. Esistono configuratori che capiscono quali sono le componenti che vengono effettivamente usate dalla mia piattaforma. Questi configuratori permettono di riapplicare la stessa ricetta anche per versioni di linux successive.

12.2 Flusso di co-sintesi a livello di sistema

In questa fase avviene la partizione tra *unità dedicate* e *unità programmabili*.

- **Sintesi Hardware** delle *unità dedicate* basata sulla ricerca o su tool commerciali per la sintesi standard.
- **Sintesi Software** delle *unità programmabili* basata su tecniche di compilazione specializzata.
- **Sintesi delle interfacce** in cui vengono definite le interfacce HW e SW e avviene la sincronizzazione.

12.3 Compilatori re-targetable

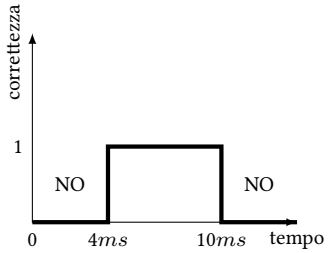
Si tratta di compilatori progettati in modo tale che risultino facili da modificare a fronte di generazione di codice per diversi instruction set di CPU.

- Risultano utili ad esempio per architetture ASIP (Application Specific Instruction-set Processor) le quali hanno un set di istruzioni specifico.
- La qualità del codice (e.g. velocità di esecuzione) è molto importante, lo è meno il tempo di compilazione.
- Sono compilatori portabili.
- Sono compilatori indipendenti dalla macchina, applicabili a diverse architetture.

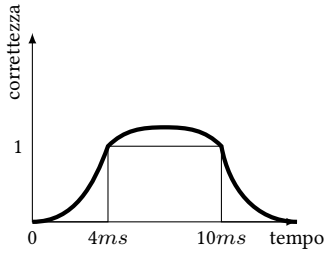
12.4 Linux RT

Linux RT: versioni *real time* di linux. Le funzionalità sono corrette quando producono risultati corretti all'interno della finestra temporale che desideriamo.

Esempio: airbag di un automobile ha requisiti real time:



Hard real time: precisione massima



Soft real time: sopporta un po' di ritardo

Nell'hard real time posso garantire che il sistema risponda in certi intervalli di tempo. Se ho un unico processo che però deve reagire a interrupt il tempo di esecuzione reale aumenta sensibilmente e si rischia di violare i tempi prefissati (non determinismo). Se voglio determinismo tolgo gli interrupt e metto polling (sebbene sia profondamente inefficiente). Ho comunque non determinismo legato alla presenza in cache del dato oppure no.

Se ho sistemi operativo con più processi il non determinismo è ancora più evidente. In sistemi operativi avanzati posso dare vincoli di real time ai processi che vengono usati dallo scheduler per capire quale processo ha priorità maggiore.

Difetti di Linux RT: è una versione obsoleta rispetto a versioni di linux più recenti. Spesso non viene usato ma si usa questa strategia: considero sempre il caso pessimo. Se non soddisfo questo requisito alzo la frequenza del processore.

12.5 Sintesi SW dal SystemC

- Le funzioni software vengono identificate da thread, poiché un singolo processore necessita della serializzazione di thread o interleaving di azioni originariamente concorrenti.
- Avviene quindi lo scheduling delle thread e delle istruzioni in modo da soddisfare i vincoli dettati dalle performance.
- C'è uno scheduler run-time a livello di sistema che sincronizza le funzioni SW e HW.
- I processi software vengono isolati dalla descrizione del sistema globale.
- Vengono generate le interfacce tra la parte HW e SW.
- Prima di andare verso l'hardware devo depurare il tutto dalle keyword di systemC: modifico il codice trasformandolo in thread che vengono fornite dal sistema operativo.
- Avvengono le misurazioni per le performance del software limitate dai vincoli hardware.

Flusso generale: TLM $\rightarrow$ generazione software $\begin{cases} \text{cache} \\ \text{HdS} \end{cases} \rightarrow$ Cross-compilazione.

Devo decidere cosa va in software e cosa va in hardware sulla base delle prestazioni richieste:

$$1000ms \begin{cases} \text{componente 1} \rightarrow 90\% \text{ del tempo di esecuzione} \\ \text{componente 2} \rightarrow 10\% \text{ del tempo di esecuzione} \end{cases}$$

Se la componente 2 viene ottimizzata di 10 volte il tempo passa da 1000 a 910. Se avessi ottimizzato di 2 volte il tempo della componente 1 lo speed up sarebbe stato notevolmente maggiore con sforzo minimo.

12.6 Misurazione delle performance Software

Uno dei tool utilizzati per la misurazione delle performance è **`gprof`**: si utilizza compilando da gcc con flag `-P`.

- Permette di calcolare quanto tempo spendo in ogni componente/sottoprogramma del sistema.
- Posso realizzare le *componenti più lente in hardware* al fine di migliorare le prestazioni.
- Per sapere quante volte è chiamata la funzione devo controllare il software ad una frequenza di campionamento maggiore.
- Occhio però perché se la frequenza è troppo alta interrompo spesso l'esecuzione e deterioro un po' i risultati.

12.7 Definizione dei testbench

L'esecuzione del programma in termini di tempi dipende fortemente dal testbench utilizzato. Il testbench è buono se rispetta alcuni vincoli di copertura:

- *Branch coverage*: percentuale di cammini che sono riuscito ad attivare
- *Statement coverage*: alcune operazioni hanno un minimo e un massimo termine di esecuzione a seconda dell'input(esempio: potenze o radici quadrate).
- *Path coverage*: i cammini di esecuzione crescono in maniera esponenziale, nel caso in cui ho processi. È necessario quindi fare delle approssimazioni. Devo cercare di individuare il cammino più lento se sono interessato al caso pessimo.

12.8 Difficoltà di test per SW Embedded

La piattaforma di esecuzione (hardware) è spesso progettata in parallelo al software embedded:

- È necessario un modello hardware.
- Non c'è una esecuzione del software easy. E quindi neanche easy testing.

Il software embedded è spesso software hardware dependent (HdS):

- La correttezza dipende dalle tempistiche
- Il comportamento del software dipende dagli eventi asincroni hardware
- È richiesta una piattaforma reale o un modello hardware complesso: no easy testing.

Una possibile soluzione approssimata:

- **Test basato su test pattern funzionali**: è un approccio comune, ma non è esaustivo, alcuni bug rimangono nascosti e non verificati.

Per migliorare l'efficienza dei test pattern:

- ATPG (generazione automatica di vettori di collaudo) funzionale: è necessario un modello di guasti hw/sw
- Restrizione dei test funzionali: è necessario un modello di guasti hw/sw
- Copertura dei cambiamenti(mutation coverage): test basato su piccole modifiche del programma, in cui ogni versione viene chiamata mutante.

Una possibile soluzione esatta:

- **Test basato sulle proprietà**: difficile da utilizzare, è esaustivo se l'insieme delle proprietà è completo.

Per migliorare l'efficienza:

- Generazione automatica delle proprietà: è necessario un modello di specifica hw/sw.
- Model checking: model checking di hw/sw è ancora un area di ricerca.

- Classificazione delle proprietà: la copertura dei cambiamenti può essere utile.

La soluzione ideale:
Generazione di software embedded che si corregge nel processo di costruzione:

- Il software viene raffinato da un modello hw/sw: è necessario un modello di specifica hw/sw.
- HdS viene generato automaticamente: il modello hardware guida la generazione del software.
- Le primitive del modello di specifica hw/sw vengono mappate automaticamente sulle primitive del sistema operativo del sistema embedded: modelli astratti di sistemi operativi.
- Non è necessario nessun test del software embedded!

13 Model Based Design e UML

Model based design si pone a un livello intermedio; si passa all'implementazione del sottosistema mediante tool automatici. Nell'approccio tradizionale scrivo software per una particolare board. Se lavoro a livello di modello sono indipendente dall'architettura.

Il software embdedded è uno specific-purpose software: è strettamente integrato con la piattaforma di esecuzione sottostante che reagisce costantemente ad eventi e unisce flusso di controllo e flusso dati.
Il suo ruolo principale non è quello di trasformare i dati, ma quello di interazione con il mondo fisico: viene eseguito su macchine che non sono solo computer, ma sono automobili, aerei, telefoni, strumenti medici, giocattoli...

13.1 Caratteristiche principali di Software Embedded

- **Tempestività** (*Timeliness*): i processi fisici evolvono lungo il tempo. Una computazione in tardiva ("late computation") non è solo in ritardo, ma è incorretta!
- **Concorrenza** (*Concurrency*): i segnali possono arrivare dall'ambiente contemporaneamente, potrebbero accadere attività disgiunte ma in parallelo che necessitano il monitoraggio.
- **Liveness**: nella visione di computazione di Turing, tutti i programmi che non terminano sono programmi difettosi. Nella computazione embedded, tutti i programmi che *terminano* sono difettosi!
- **Reattività** (*Reactivity*): E.S. sono sistemi vincolati in tempo reale e safety critical che reagiscono in modo continuo con l'ambiente che li circonda.
- **Eterogeneità** : E.S. mescolano stili di computazione con tecnologie di implementazione.

Il software embedded (ESW) deve quindi essere:

- Sviluppato rapidamente;
- Rapidamente modificabile ed estendibile;
- Conforme alle specifiche del cliente;
- Facilmente portabile tra le piattaforme di Sistemi Embedded.

13.2 ESW: vincoli di progettazione

La progettazione di software embedded implica vincoli di progettazione che vanno in conflitto tra loro:

- Efficiente, efficace, basso costo computazionale, basso utilizzo di memoria $\rightarrow$ stretta integrazione con la piattaforma ES;
- Riusabile, sviluppabile velocemente, mantenibile nel tempo, portabile $\rightarrow$ astratto e indipendente dalla piattaforma ES.

Per risolvere il conflitto di progettazione: **Model Based Design (MBD)** $\rightarrow$ progettazione basata su modelli astratti, generazione automatica di codice dai modelli astratti.

13.3 MBD: soluzioni sul mercato

Matlab permette lo sviluppo di agoritmi usando componenti già pronte assemblabili tra loro e quindi indipendentemente dal software. Saranno poi i plug-in di Matlab a fare sintesi sulla board specifica. L'unico *difetto* di Matlab è quello che la specifica scritta in Matlab non è portabile in altri ambienti.
Modelica: simile a Matlab ma open source. Utilizzato in progettazione di sistemi tecnici e.g.: sistemi meccanici, elettrici, termici, idraulici... La specifica del comportamento dinamico è descritta con equazioni differenziali, algebriche e discrete. *Difetto*: scarsa manutenzione, supporta principalmente specifica funzionale/comportamentale.
Simulink: progettazione di sistemi dinamici e di controllo. Descrizione dei sistemi tramite una libreeria di blocchi grafici o di equazioni differenziali, algebriche e discrete.*Difetto*: supporta principalmente specifica funzionale/comportamentale, supporta in parte la partizione strutturale del sistema.

Il mercato cerca qualcosa di standard: **UML**

- Specifica strutturale: componenti
- Specifica funzionale
- Specifica use case

Vantaggi: linguaggio aperto e flessibile. Le specifiche grafiche sono facilmente interpretabili. Permette la specifica a diversi livelli di astrazione:

- Livello use case
- Livello di sistema (structural view)
- Livello di integrazione (subsystems interaction view)
- Livello di unità (*Unit Level* - subsystem functional view)

13.4 Unified Modeling Language (UML)

È un linguaggio standard per:

- Specificare
- Visualizzare
- Costruire
- Documentare
- Modellare il business
- Comunicazioni

Ci permette di visualizzare secondo 3 punti di vista diversi: *Utente*, *Designer* e *Analizzatore*.
UML mette a disposizione diversi **diagrammi** per la rappresentazione dei sistemi:

- *Use Case diagram*: sono diagrammi dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.
  Usati nei E.S. solamente all'inizio con il cliente, in modo da definire un'idea astratta.
- *Class diagram*: si definiscono le classi con attributi e relazioni del sistema includendo ereditarietà, aggregazione, associazione e le operazioni e gli attributi delle classi.
  Non molto utili nei E.S. perchè troppo ad alto livello per le strutture dati dei ESW.
- *Sequence Diagram*: diagramma di interazione che spiega in dettaglio come le operazioni vengono portate avanti nel tempo. Quali messaggi vengono inviati e quando in base al tempo di vita del sistema.
  Usati nei E.S. per ritrarre le relazioni nel tempo, anche se è difficile scrivere codice in modo automatico.
- *Activities Diagram*: si permettono di definire stati e transazioni per descrivere cosa avviene nel tempo.
  Non usati nei E.S.: troppo distanti dal codice effettivo.
- ***State Machine Diagram***: mostrano i possibili stati degli oggetti e le transizioni che provocano un cambiamento dello stato degli oggetti.
  Usati nei E.S. e rappresentano un trade off perfetto tra l'astrazione e la verosimiglianza con il codice effettivo. Permettono di fare una rappresentazione astratta del ESW, pur mantenendo il contatto con il codice generato.

Vantaggi: trade-off tra astrazione e codice finale. Permette di rappresentare cosa effettivamente viene fatto.

Esistono tool che permettono di convertire modelli UML in codice. Effettivamente usati su embedded (IAR azienda che produce questi tool.)

14 A Software Cloud Architecture (IoT)

Devo mettere insieme il cloud con gli oggetti embedded. Per cloud si intende archivio che risiede in remoto.

IoT(Internet of Things): un sistema di device interconnessi, dispositivi digitali e meccanici, oggetti con identificatori univoci e con la possibilità di trasferire dati sulla rete senza che sia richiesto l'intervento umano.

14.1 Scenari di applicazione

Vengono elencati di seguito i possibili scenari di applicazione di IoT.

1. Prodotti IoT per la salute (smart healthcare): questi prodotti permettono di misurare battiti cardiaci, sudore, movimento, numero di passi, pressione del sangue, attività cerebrale, tutto in tempo reale e salvato in modo sicuro nel cloud.
2. Prodotti IoT per la casa (Smart Home): per controllare l'umidità, luci, serrature...
3. Prodotti IoT per la mobilità (Smart Mobility): dispositivi per l'industria del settore automobilistico come il controllo della pressione delle gomme o concetti più grandi tipo auto connesse per le misure di sicurezza sulla strada, sensori di parcheggio, sensori nelle strade e nei parcheggi..
4. Prodotti IoT per l'energia (Smart grid / energy): dai contatori smart a vere e proprie griglie smart.
5. Prodotti IoT per l'agricoltura : per ottimizzare i processi di manodopera, diminuire lo spreco e aumentare la qualità
6. Prodotti IoT per la città (Smart cities)
7. Prodotti IoT per l'industria (IIoT): per migliorare l'efficienza, aumentare la produzione.

14.2 Ingredienti per una soluzione IoT

Una soluzione IoT è composta da molti strati:

- **Strati di device** (*device layers*): sono le *things* dell IoT, cioè dispositivi embedded che sono equipaggiati per leggere, elaborare e inviare informazioni all'interno di internet. Solitamente un dispositivo è un micro-controllore equipaggiato di un chip di memoria, processore integrato, porte di I/O e di rete e un piccolo sistema operativo (Linux Yocto, Windows 10 IoT...). La capacità di controllare il dispositivo e di farlo comunicare con la rete è uno degli aspetti chiave di IoT.
  - **Sensori**: giocano un ruolo importante nelle soluzioni IoT. Un sensore può essere programmato per catturare i parametri dell'ambiente richiesti che necessitano di essere monitorati e successivamente questi vengono convertiti in dati utili. Difficilmente hanno un'interfaccia. Sono i più diffusi.
    Non hanno controlli di input e output o funzioni, l'informazione può essere visualizzata in una console connessa con un altro dispositivo.
    Sono low powered device, e spesso sono equipaggiati con una piccola batteria a lunga vita che dura mesi o anche anni.
  - **Attuatori**: dispositivi che interagiscono con il mondo fisico e in base all'input che ricevono svolgono l'azione richiesta. E.g.: movimento, luci, emissione di suoni, controllo di potenza ecc..
  - *RFID* (Radio Frequency Identification): tessera/ codice a barre smart che funge da chiave e identifica un device o un prodotto e lavora in connessione con il lettore RFID per tracciare l'informazione del prodotto e inviarla alla rete.
  - *NFC* (near field communication): funzionano su corte frequenze radio. Funzionano come gli RFID, ma il dispositivo con protocollo NFC può agire sia come lettore che come tag per una comunicazione a due vie. Sono utili per i pagamenti contactless.
  - *Beacons*: altra forma di dispositivi tag che trasmettono segnali o condividono in broadcast le proprie informazioni sulla posizione utilizzando bluetooth low energy(LE).

- **Strati di comunicazione** (communication layer): la strategia di comunicazione è basata sulle capacità dei rete di ogni dispositivo. Un dispositivo può essere in grado di connettersi direttamente ad internet con wifi o con la rete telefonica. Al contrario un altro dispositivo può non essere in grado di connettersi direttamente ad internet e magari deve fare uso di un gateway o di un hub.
  - **Device Gateway**: un dispositivo che connette reti o protocolli incompatibili tra loro e fornisce un mezzo per connettere dispositivi ad internet. Il gateway tradurrà i dati in arrivo in dati IP che possono essere poi instradati nella rete.
  - **Smart gateway** (o edge gateway): sono device gateway dotati di archiviazione locale e applicazioni embedded che possono fare analisi sui dati che vengono trasmessi direttamente dai dispositivi. Permettono di abbattere la complessità dei nodi. Un gateway può avere capacità computazionale. Facendo così aumentano l'efficienza facendo computazione in tempo reale. un esempio è *Kura*, un progetto open source provvisto di una piattaforma per costruire IoT edge gateways.
  - **Smartphone**: un sensore o attuatore può comunicare con uno smartphone attraverso il protocollo Bluetooth, lo smartpone può a sua volta utilizzare wifi o rete telefonica per inviare i dati. Lo smartphone non automatizza il processo di comunicazione da solo, è necessario un intervento manuale per impostare il processo.
- **Protocolli di comunicazione**: responsabili della connettività della rete. Alcuni forniscono una connettività diretta con internet (Wifi, Ethernet, rete telefonica tipo 2G, 3G, 4G, LTE...).
  Altri protocolli richiedono un gateway per tradurre comandi specifici nei corrispondenti comandi IP (Bluetooth LE, RFID, NFC, Zigbee/Z-wave per dispositivi con vincoli ambientali).
- **Protocolli di Applicazione**: sono in cima allo strato TCP. Utilizzano tCP come strato di trasporto per comunicare con messaggi di applicazione specifici.
  - **MQTT** (Message Queue Telemetry Transport): protocollo di connettività machine-to-machine (M2M). È un protocollo di trasporto di messaggi di tipo publish/subscribe molto leggero.
    Utile per comunicazioni di piccoli dati, dispositivi con vincoli di risorse, reti con banda ristretta e alta latenza. Ha un payload binario ristretto.
    I dispositivi IoT sono client che si connettono all'MQTT broker, il quale supporta connessioni multiple concorrenti che coinvolgono un grande numero di dispositivi. Il dispositivo client inizia la connessione al broker MQTT.
  - **AMQP** (Advanced Message Queuing Protocol): un altro famoso middleware orientato ai messaggi che opera con il trasporto TCP. Questo protocollo supporta sia il modello a code di messaggi che il modello publish/subscribe. Assicura un'affidabile consegna dei messaggi.
- **Protocolli di industria**: protocolli progettati per implementare soluzioni connesse specifiche per le industrie. Sono protocolli proprietari e permettono l'automazione industriale (SCADA, Modbus...).
- **Piattaforma core**: fornisce un insieme di possibilità per la connessione, il controllo e il monitoraggio di milioni di dispositivi. Composta da:
  - *Protocollo Gateway*: usata per il supporto della gestione di più protocolli. Permette la conversione tra protocolli. Può agire come interfaccia per supportare diversi protocolli e convertire diversi protocolli.
  - *IoT Messagging Middleware*: è un software che permette ai publishers e ai subscribers di distribuire messaggi senza che questi siano connessi fisicamente tra loro. È molto scalabile e pò gestire un vasto numero di dispositivi connessi assicurando alte prestazioni e tolleranza ai guasti.
  - *Data Storage*: archiviazione di un flusso continuo di dati proveniente dai dispositivi. C'è bisogno di un servizio di archiviazione molto scalabile. I dati devono essere replicati attraverso i server per assicurare un'alta disponibilità e per evitare errori.
    Tipicamente vengono usati database NoSQL: database Time Series in cui i dati sono rappresentati con una sequenza di vaolri per uno specifico attributo per un certo periodo di tempo. (MongoDB, Cassandra, Hbase, CouchDB...).
  - *Aggregazione di dati e filtraggio*: l'aggregazione dei dati permette di contestualizzare i dati del dispositivo con più informazioni.
- **Strato di piattaforme di analisi** (Analytics Platform Layer): permettono di analizzare grandi quantità di informazioni, derivare approfondimenti. Formato da:

- *Stream processing*: interagiscce con il componente di Messaging Middleware di IoT ascoltando i topics specificati. Fa real time stream processing, cioè analizza flussi di dati di dispositivi in tempo reale.
- *Machine Learning*: in che modo le macchine imparano dai dati utilizzando diversi algoritmi senza programmarle in modo specifico, in modo tale che ritornino il risultato richiesto. Viene classificato in tre categorie:
  1. *Supervised Learning*: forniamo dati etichettati come input e output desiderato e alleniamo il sistema ad imparare dai dati e a predirre i risultati.
  2. *Unsupervised learning*: non vengono forniti i dati etichettati ma viene lasciato agli algoritmi il compito di trovare una struttura nascosta nei dati non etichettati.
  3. *Reinforcement learning*: i sistemi imparano attraverso l'interazione con l'ambiente.
- *Events and Reporting*: insieme di servizi che rendono più semplice l'invocazione delle azioni richieste in base ai dati analizzati. Le azioni posso essere: l'innesco di eventi, la chiamata a servizi esterni, l'aggiornamento dei report in tempo reale.

IoT dovrebbe inoltre organizzare la sicurezza e il controllo delle applicazioni IoT.

Per lavorare su IoT devo gestire il dispositivo, i protocolli di comunicazione e il cloud. È fondamentale per la sicurezza per evitare che i pacchetti vengano intercettati. Dovremmo avere un meccanismo di crittografia che parte dal sensore. Devo infatti spesso interfacciarmi con software di terze parti.

15 Networked Embedded Systems Design (NESD)

I sistemi Embedded di rete rappresentano una classe importante di dispositivi: le funzionalità della rete sono il nucleo degli obiettivi della progettazione. L'unione della modellazione e della simulazione HW-Sw e la rete è un beneficio: si ha l'integrazione di un simulatore di sistema e di un simulatore di rete.
Questioni aperte:

- Le applicazioni NES sono sviluppate senza alcun supporto di software di sistema;
- Non è semplice lo sviluppo di applicazioni NES;
- Esistono applicazioni ad-hoc, ma non sono flessibili e hanno alti costi di sviluppo.

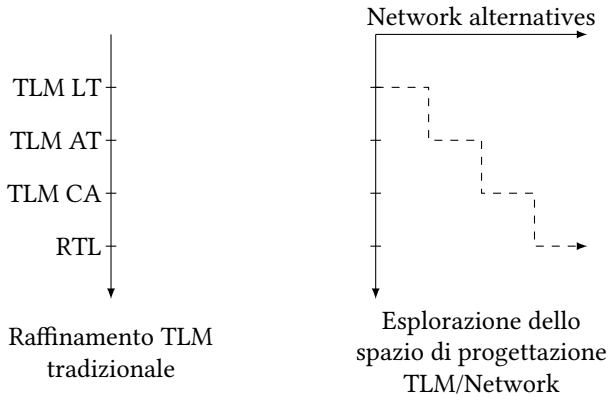
15.1 System/Network co-simulazione

- **Obiettivo:** Efficiente riproduzione di computazione (system) e trasmissione del comportamento (network) di un'applicazione.
- **Risultati:**
  - *Simulazione di rete in SystemC*: si ha l'estensione dell'insieme di primitive del SystemC per modellare le reti basate su pacchetti. I NES descritti in SystemC possono essere direttamente connessi allo scenario di rete descritto in SCNSL (solo un tool/linguaggio, simulazione più veloce).
  - *Sincronizzazione tra diversi tool di simulazione*: il simulatore dell'instruction set e il SystemC vengono sincronizzati attraverso un algoritmo e i due comunicano tra loro.
  - *Simulazione dei guasti basata sulla rete*: i guasti di un'applicazione di rete sono dovuti a un errato comportamento delle trasmissioni dati asincrone basate sullo scambio di pacchetti. I pacchetti sono soggetti a diversi guasti:
    - LOST: pacchetto scomparso dal canale;
    - CORRUPT: alcuni bit del pacchetto trasmesso sono stati girati;
    - CUT: alcuni bit adiacenti sono scomparsi dal canale;
    - DUPLICATE: il pacchetto trasmesso è stato inviato due volte;
    - CARRIER: il test sull'uso del canale è o positivo o negativo indipendentemente dalla presenza di una effettiva comunicazione.

15.2 System/Network co-progettazione

- **Obiettivo:** Unire la progettazione di unità di computazione (sistema) e la progettazione di protocolli di trasmissione (network).
- **Risultati:** Un flusso di progettazione attento alla rete per modellare i requisiti di trasmissione di una applicazione per guidare la sintesi dei protocolli di trasmissione.

TLM (Transaction Level Modeling) sta diventando un approccio comune per semplificare la progettazione di sistemi embedded: da un modello funzionale in TLM Loosely Timed a un modello accurato a Livello Transazionale Cycle Accurate.
Come combinare Hw-Sw-Network co-modellazione e co-simulazione con il TLM?



Suddivido lo spazio di progettazione in 2 dimensioni:

1. Dimensione *Verticale*: raffinamento del sistema, ottimizzazione dell'algoritmo.
2. Dimensione *Orizzontale*: diverse topologie di rete, diversi parametri di rete.

Vantaggi:

- I requisiti della rete possono essere facilmente verificati dopo ogni passo di raffinamento;
- I test pattern realistici possono essere facilmente generati;
- L'implementazione dei protocolli di rete può essere testata contro la loro specifica astratta nel simulatore di rete;
- Pianificazione di rete per le migliori performance.

16 Verification and Testing

Figura 1: Flusso di Verification and testing

Solitamente i requisiti iniziali vengono scritti in un testo semi-strutturato; la descrizione non è formale. Dobbiamo quindi passare da questi requisiti a delle specifiche formali. Questo passaggio è scarsamente automatizzabile. Seguono alcune definizioni.

Definizione 1 (Test)**.** Metodi che mi permettono di capire se un sistema ha o non ha guasti. Se un chip non funziona lo butto via, non posso ripararlo (GO/ NO GO).

Definizione 2 (Fault Detection)**.** Identificazione di un malfunzionamento.

Definizione 3 (Fault Diagnosis)**.** Identificazione del guasto. Se il chip non funziona, devo identificare la posizione del guasto. Permette alla fonderia di silicio di capire quale fase della produzione ha dato problemi.

Definizione 4 (Validazione)**.** Processo in cui verifico se la specifica formale è coerente con la specifica informale Confronto tra la parte informale e quella formale. Ci sono pochi strumenti automatici in grado di fare ciò. Dalle specifiche ottengo quindi un modello simulabile; riesco a dire così se le specifiche sono conformi. Per questa fase si richiede quindi un intervento umano.

Definizione 5 (Verifica)**.** Confronto tra specifica formale e realizzazione. (esempio: il modello RTL è conforme con quello di partenza?). Una specifica formale viene sintetizzata in una realizzazione. Quando ottengo una specifica formale raffino il progetto in un livello di sintesi sottostante. La verifica è un processo automatizzabile perché basato sul confronto tra due linguaggi formali.
In sostanza si vuole assicurare che il design sintetizzato, una volta fabbricato, svolgerà le funzioni di i/o date dalle specifiche formali.

Se avessi solo software allora mi basterebbero solamente Validazione e Verifica.

Definizione 6 (Sintesi del design)**.** Data una funzione di i/o, sviluppa una procedura per fabbricare un dispositivo utilizzando materiali noti e processi.

Definizione 7 (Test di Produzione)**.** Un passo della fabbricazione che assicura che il dispositivo fisico, fabbricato dal design sintetizzato, non ha difetti di fabbricazione.

16.1 Verifica vs Test

Verifica:

- Verifica la correttezza del design
- Attuata dalla simulazione, emulazione di hardware o metodi formali
- Attuata una volta prima della fabbricazione
- Responsabile della qualità del design

Test:

- Verifica la correttezza dell'hardware fabbricato
- Processo diviso in 2 parti:
  1. **Generazione dei test**: processo sw eseguito una volta durante la progettazione
  2. **Applicazione dei test**: test elettrici applicati all'hardware.
- L'applicazione dei test viene attuata su ogni dispositivo fabbricato
- Responsabile della qualità dei dispositivi.

Il test dell'hardware è incredibilmente più complesso perché errori possono avvenire anche in fase di produzione (non è così nel software).

16.2 Test di fabbricazione (Manufacturing test)

Nei sistemi embedded abbiamo sia hw che sw. L'hardware deve essere realizzato (esempio: una FPGA deve essere fabbricata). Il test di fabbricazione viene fatto dopo la fabbricazione dell'oggetto fisico. Passo quindi dal mondo simulato della progettazione al mondo fisico.
È complicato vedere se un transistor è difettoso (i transistor in un chip sono milioni). L'osservabilità e la controllabilità sono molto difficili da valutare. Il costo del test non deve superare il costo a cui il prodotto è venduto. Spesso il test non viene nemmeno fatto.
Esempio:

- Eseguo un test su un chip, costo: 1€. Il test non è accurato ma viene comunque messo in vendita. Il chip viene quindi montato su una board.
- Viene eseguito un test sulla board, costo: 10€. La board risulta difettosa perché il chip è difettoso. La board viene comunque messa insieme ad altre board.

- Questo insieme di board viene testato, costo: 100€. L'insieme delle board risulta essere difettoso.

Ad ogni livello di integrazione il costo del test per sapere se funziona decuplica. Il 40 o 50% dei costi di sviluppo sono dovuti a validazione, verifica e test. Spesso si arriva anche al 70%.

16.3 Functional Test (test funzionale)

Verifica la correttezza del design. Un generatore automatico di testbench fa esattamente lo stesso della verifica. Il functional test fa da tramite tra i due mondi. Utilizza tecniche di test pattern per generare test per fare validazione/verifica. Un test pattern deve essere in grado di evidenziare il guasto se questo è presente. Dovrei orientare il test ai difetti ma i difetti possibili sono moltissimi.

Il test deve essere inserito nella fase di progetto. Ottengo un chip più grande, che consuma di più o più lento, ma che è testabile. Modifico il progetto al fine di poter generare più velocemente il test. Il sistema deve avere delle risorse hardware aggiuntive che permettono di avere un sistema *fault-tollerant*.

Verifica del Design: per verificare formalmente la correttezza del design.
Test interconnesso: per verificare la correttezza dei blocchi di interconnessione.
Test dei guasti (fault testing): per verificare i guasti che emergono dalla fabbricazione.

16.4 Problemi con il test ideale

I test ideali individuano tutti i difetti prodotti nel processo di fabbricazione. Tutti i dispositivi corretti funzionalmente passano i test ideali.
Il problema principale è: *quali sono tutti i guasti per i dispositivi hardware*? Tutt'ora non si conosce il modo di generare test pattern per ogni tipo di guasto.

16.5 Test Reali

- Basati su modelli di guasti analizzabili, che potrebbero non mapparsi su tutti i reali difetti.
- Incompleta copertura dei guasti modellati a causa dell'alta complessità
- Alcuni chip buoni vengono buttati: la percentuale di questi chip viene chiamata perdita di produzione (yeld loss).
- Alcuni chip difettosi passano i test: questa percentuale viene chiamata defect level.

Nella VLSI (Very large scale integration) ci possono essere difetti di fabbricazione. Il costo di un chip è dato dal seguente rapporto:

$$\frac{\text{Costo della fabbricazione e del test di un wafer}}{\text{Perdita per numero di locazioni del chip nel wafer}}$$

Defect Level (DL) è il rapporto dei chip difettosi tra i chip che passano il test.
Il livello di difettosità è misurato in *parti per milion*e (ppm).
Rappresenta una misura dell'efficacia dei test e una quantitativa misura della qualità del prodotto fabbricato.
Per chip commerciali VLSI un DL maggiore di 500ppm è considerato inaccettabile.

16.6 Design for Testability (DFT)

Si riferisce a stili di progettazione hardware che riducono la complessità della generazione dei test. Motivazione: la complessità della generazione dei test aumenta esponenzialmente con la dimensione del circuito.
Ruolo del testing:

- Rilevazione (detection): determinare se il DUT (design under test) ha o meno guasti.
- Diagnosi: identificazione di guasti specifici presenti nel DUT.
- Caratterizzazione del design: identificazione e correzione di errori nel design o nella procedura di testing.
- Failure mode analysis (FMA): identificazione di errori nel processo di fabbricazione che possono aver provocato difetti nel DUT.

Difficoltà del testing:

- Aumento delle dimensioni dei circuiti e dei sistemi;
- Aumento della densità dell'integrazione: più funzionalità necessitano di essere testate;
- Inaccessibilità degli interni dei circuiti;
- Diminuzione del rapporto tra porte / pin: minor livello di controllabilità/osservabilità;
- Aumento delle velocità: correttezza logica non è più sufficiente;
- L'analisi fisica dei difetti è troppo costosa;
- I difetti fisici sono modellati usando i guasti: modelli a livello logico come STUCK-AT

Guasto: fallimento del meccanismo fisico dovuto ad alcuni difetti.
Effetto del guasto: effetto logico di un guasto sulle linee in una rete
Errore: la condizione (lo stato) di un sistema che contiene un guasto (deviazione da uno stato corretto).

16.7 Caratteristiche dei guasti

Guasto permanente: guasto che dura per un tempo sufficiente da essere osservato nell'ambiente di test.
Guasto transiente: guasto che appare e scompare a intervalli di tempo brevi.
Guasto intermittente: guasto che appare e scompare a intervalli di tempo regolari.

16.8 Modelli di guasti

Sono astrazioni logiche che descrivono l'effetto funzionale di un guasto fisico. C'è la necessità di astrarre perché altrimenti i costi sarebbero eccessivi.
Esistono diversi livelli di modellazione dei guasti basati su diverse primitive:

- funzione (livello behavioural)
- struttura (livello RT)
- Circuito (livello gate)
- Porte di libreria (livello switch)

Esistono guasti diversi a seconda del livello:

- Livello funzionale (TLM):
  - Funzioni generiche
  - Metriche ti copertura sw: statement/condition/-branch/path coverage
- Livello RT:
  - FSM e data-path
  - Blocchi funzionali e registri: transition/stuck-at line/erroneous block functionality
- Livello Gate:
  - Porte di libreiria
  - Modello di guasto basato su linea: stuck-at/gate-delay/path-delay/bridge
- Livello switch:
  - celle standard
  - Transistor
  - modelli elettrici