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).

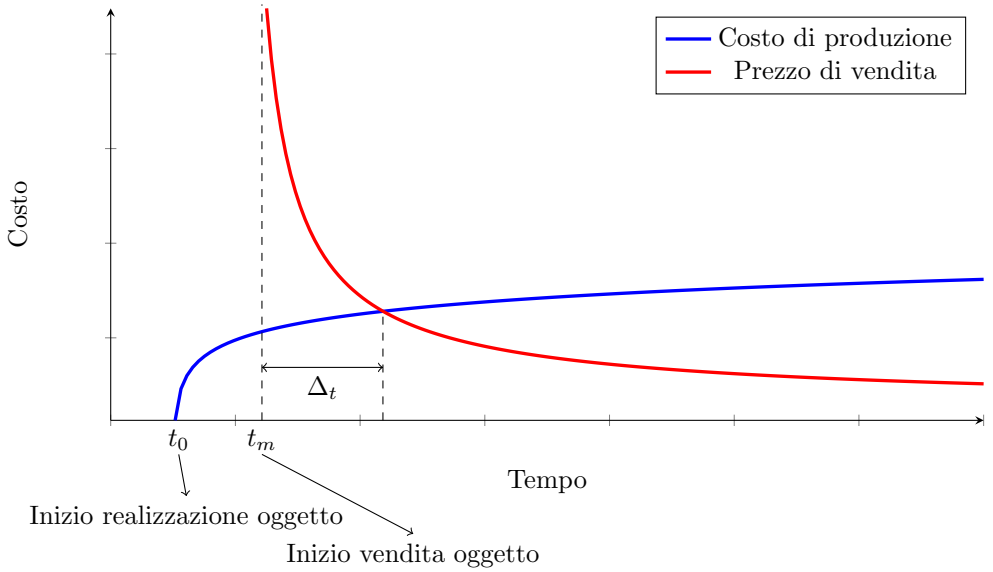
In generale, un sistema embedded è costituito dalle seguenti componenti:

- **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`.

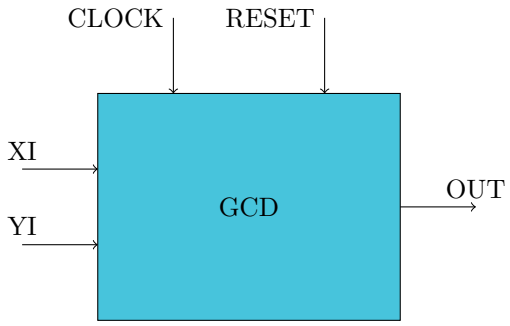
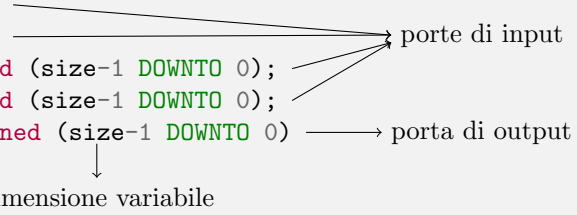


Figura 2.2: Schema di un modulo GCD

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:

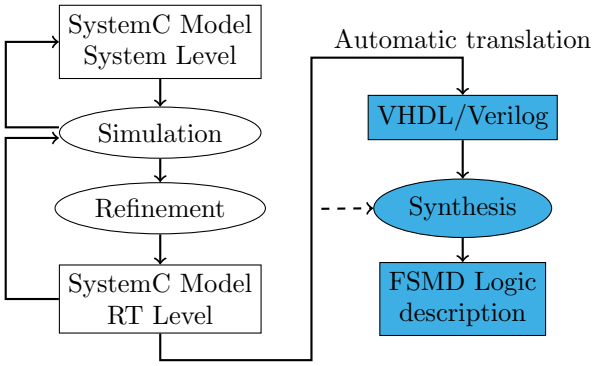


Figura 3.1: SystemC design flow

- **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) {                    → dichiarazione del modulo
  sc_in<bool> ds;
  sc_in<sc_bv<N>> a;                    } porte di input
  sc_in<bool> i0;
  sc_out<sc_bv<N>> o;                   → porta di output
  void shift(); ⟶ dichiarazione metodo interno

  SC_CTOR(shifter) {
    SC_METHOD(shift);
    sensitive << ds << a << i0; ⟶ dichiarazione sensitivity list
  }
};
```

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);
}
```

Nell'header si definisce il nome del modulo con la macro `SC_MODULE`, le sue porte

di input e di output (con tipo di dato e ampiezza in bit quando richiesto), i metodi che implementano il comportamento del modulo e la *sensitivity list*, ovvero la lista dei segnali a cui il modulo è sensibile. Tramite le macro `SC_METHOD`, `SC_THREAD` e `SC_CTHREAD` identifico rispettivamente i metodi, i thread e i clocked thread (deprecati).

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*. La progettazione platform based consiste nella creazione di un'architettura basata su microprocessore che può essere estesa rapidamente per un ampio range di applicazioni in tempi ridotti. In TLM si descrive il sistema a livello comportamentale, in cui non si sa esattamente cosa succede a ogni ciclo di clock. Ci si focalizza sulle transizioni tra le varie componenti, che nel concreto diventeranno delle interfacce.

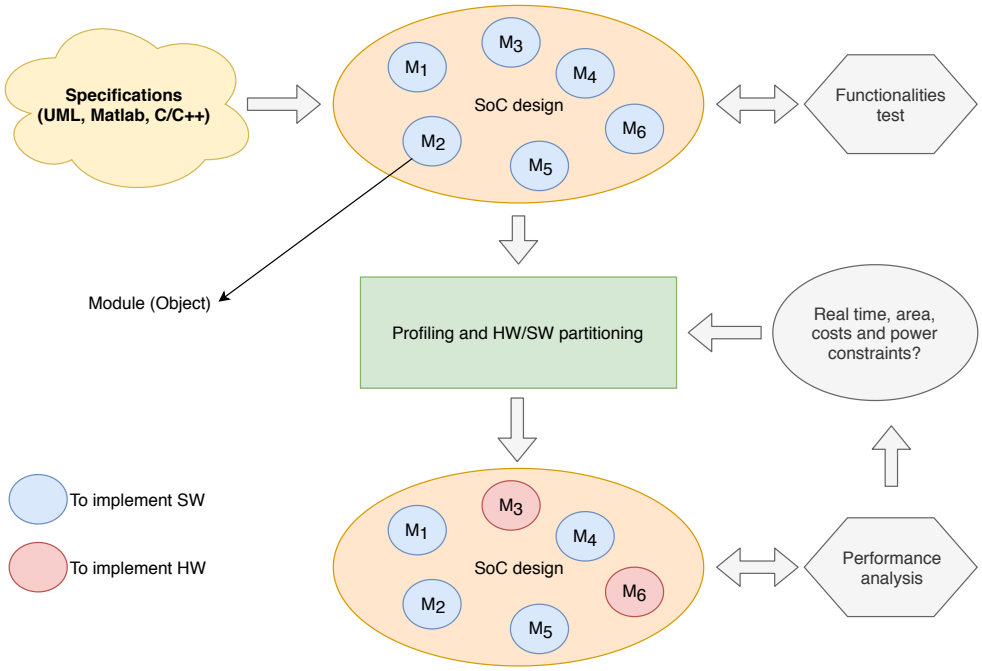


Figura 3.2: Visualizzazione della progettazione *platform based*

Date le specifiche, che possono essere in un linguaggio ad alto livello come UML, MATLAB o C/C++, si codificano i vari moduli e le interazioni fra di essi e si passa al profiling, in cui si decide cosa diventa software e cosa diventa hardware. È pratica comune riutilizzare componenti già codificate da altri, tanto che esistono portali

dedicati all'IP re-use (*Intellectual Properties re-use*). I vantaggi principali offerti dalla progettazione TLM sono la velocità di simulazione (fino a 1000 volte più veloce rispetto a RTL), la semplificazione del design e una riduzione drastica dei tempi di sviluppo.

Alla base del TLM vi è il concetto di transazione, che permette il trasferimento di dati da un modulo all'altro. La comunicazione avviene tramite la chiamata di una primitiva del ricevente (che chiameremo *target*) da parte del mittente (*initiator*), alla quale viene agganciato un *payload*, che contiene sia informazioni utili sia di controllo.

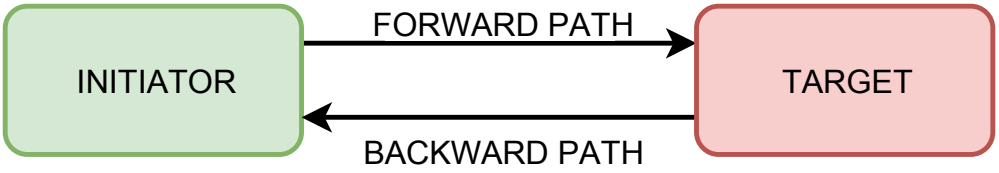


Figura 3.3: Schema forward/backward path

SystemC ha definito due standard diversi di TLM. Nello standard 1.0 venivano proposti 3 diversi livelli di astrazione, ovvero

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

tuttavia nello standard 2.0 sono stati abbandonati per concentrarsi invece nella relazione tra dati e tempo. Sono stati definiti quindi 3 modi livelli di accuratezza:

- **Untimed**

  Interfacce bloccanti

  Punti di sincronizzazione predefiniti

- **Loosely timed**

  Interfacce bloccanti

  Due punti di sincronizzazione (invocazione e ritorno)

- **Approximately timed**

  Interfacce non bloccanti

  Annotazione del tempo con fasi multiple durante la transazione

Le interfacce bloccanti, supportate dalle descrizioni UT e LT utilizzano solamente il forward path. L'initiator chiama il metodo `b_transport` del target, che ritorna il valore nel payload stesso. Lo stile *untimed* e il *loosely timed* sono sufficientemente dettagliati per poter eseguire il boot di sistema operativo sull'architettura simulata. Permettono

anche meccanismi complessi come il *temporal decoupling*, nel quale due processi che lavorano in contemporaneo a volte possono andare per conto proprio oltre il tempo attuale della simulazione, per poi arrivare al punto di sincronizzazione. Con questo stile posso solamente fare una stima del tempo trascorso con un minimo e un massimo.

Nello stile *approximately timed* le transazioni vengono divise in più fasi. Il protocollo base usa 4 fasi (per questo lo stile viene chiamato AT4):

- Inizio richiesta (`BEGIN_REQ`)
- Fine richiesta (`END_REQ`)
- Inizio risposta (`BEGIN_RESP`)
- Fine risposta (`END_RESP`)

La sincronizzazione avviene più frequentemente quindi di LT, per questo il tempo di simulazione è più lungo, ma ottengo una stima del tempo molto accurata.

3.3 SystemC AMS

SystemC AMS è un set di librerie che estendono SystemC in modo da poter supportare la simulazione di sistemi analogici, sia in tempo continuo che in tempo discreto. La descrizione di un sistema misto può essere fatta in tempo continuo o in tempo discreto.

In tempo discreto i segnali e le quantità fisiche sono definite in punti precisi e si assumono costanti nel mezzo. Il comportamento viene descritto in maniera procedurale utilizzando segnali campionati. Questo tipo di descrizione si adatta particolarmente a sistemi dove il *signal processing* è abbondante. In descrizioni a tempo continuo i segnali e le quantità fisiche sono descritti come funzioni reali del tempo, considerato come un valore continuo. Il comportamento viene descritto da equazioni algebriche o differenziali, risolte da complessi algoritmi. Queste descrizioni sono adatte a descrivere sistemi fisici dinamici.

Oltre alla distinzione tempo continuo/tempo discreto vi è la differenza tra descrizione conservativa e non conservativa. In una descrizione conservativa il comportamento del sistema segue le leggi della fisica (ad esempio nella descrizione di un circuito elettrico devono essere rispettate le leggi di Kirchhoff), mentre in una non conservativa non valgono ed è possibile descrivere dinamiche non lineari. Una descrizione conservativa è computazionalmente più pesante di una non conservativa. Le due descrizioni possono essere usate nella stessa simulazione.

SystemC AMS prevede 3 stili di modellazione:

- **Timed Data Flow** (TDF)

  Tempo discreto, non conservativo

  Scheduling statico basato su dataflow

- **Linear Signal Flow** (LSF)

  Tempo continuo, non conservativo

Basato su equazioni algebriche e differenziali

Risolutori simbolici o numerici

- **Electric Linear Network** (ELN)

  Tempo continuo, conservativo

  Modellazione di reti elettriche seguendo le leggi di Kirchhoff

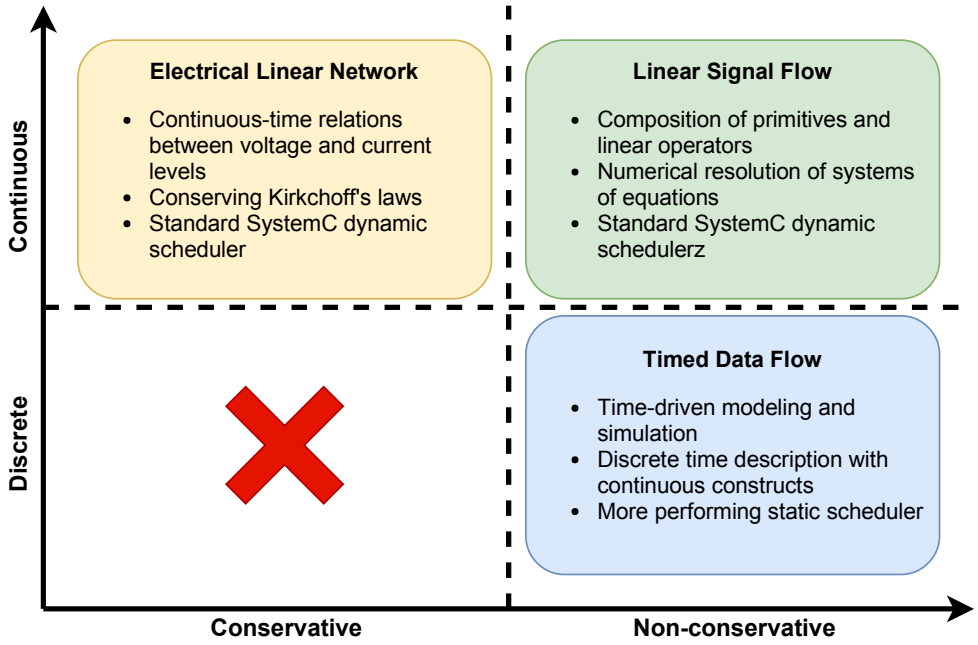


Figura 3.4: Gli stili di SystemC AMS

3.3.1 Timed Data Flow

Questo stile si basa su un *synchronous data flow* (SDF), che descrive un sistema a eventi discreti. Ogni modulo di evento discreto contiene un metodo C++ che calcola una funzione matematica. Una funzione viene eseguita se e solo se ci sono un numero di campioni sufficiente per ogni porta di input, dove il numero di campioni sufficiente è un valore fissato a priori.

Supponiamo di dover esprimere $f_C(f_B(f_A))$.

Lo scheduling in TDF è statico, in quanto viene definita a priori una sequenza di eventi discreti che viene eseguita al di fuori dello scheduler *event-based* del SystemC puro, portando ad un aumento dell'efficienza.

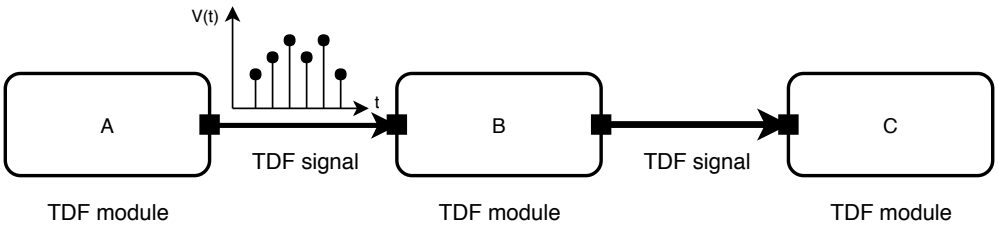


Figura 3.5: $f_C(f_B(f_A))$

Un modulo TDF è strutturato in questa maniera:

my_tdf_module.h

```
SCA_TDF_MODULE(my_tdf_module) ——→ dichiarazione del modulo
{
    // port declaration
    sca_tdf::sca_in<double>      in; ——→ porte di I/O
    sca_tdf::sca_out<double>     out;

    SCA_CTOR(my_tdf_module){ ——→ metodo costruttore
    }

    void set_attributes(){ ⟶ definizione attributi delle porte e del modulo
        // module and port attributes
    }

    void initialize(){ ⟶ definizione valori iniziali
        // initial values of ports with a delay
    }

    void processing(){ ⟶ implementazione funzionalità effettive
        // time-domain signal processing behaviour or algorithm
    }

    void ac_processing(){ ⟶ implementazione funzionalità e del rumore
        // small-signal frequency-domain behaviour
    }
};
```

3.3.2 Linear Signal Flow

In LSF, il comportamento del modello viene definito come relazioni tra variabili di un insieme di equazioni differenziali. A differenza di TDF, il segnale viene rappresentato da un valore reale. In LSF, il flusso del segnale viene rappresentato con l'ausilio di un diagramma a blocchi, dove le parti elementari come la moltiplicazione o l'addizione costituiscono i blocchi e le linee di connessione i segnali. I blocchi elementari sono stati predefiniti in SystemC AMS e non è possibile estenderli per crearne di nuovi. I segnali per l'interconnessione dei blocchi sono segnali reali, per tanto per interfacciare un modulo LSF serve discretizzare.

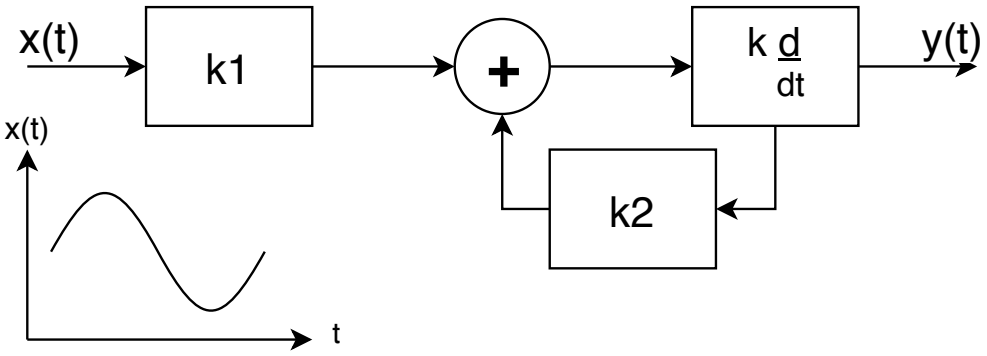


Figura 3.6: Diagramma a blocchi dell'equazione $y(t) = k_1 \frac{dx(t)}{dt} + k_2 \frac{dy(t)}{dt}$

Si presenta di seguito l'implementazione in SystemC AMS LSF del diagramma a blocchi appena mostrato.

my_structural_lsf_model

```
SC_MODULE(my_structural_lsf_model)
{
    sca_lsf::sca_in     x;
    sca_lsf::sca_out    y;

    sca_lsf::sca_gain   gain1, gain2;
    sca_lsf::sca_dot    dot;
    sca_lsf::sca_add    add;

    my_structural_lsf_model(sc_core::sc_module_name name,
                            double k1,
                            double k2):
        x("x"),
        y("y"),
        gain1("gain1", k1),
        gain2("gain2", k2),
        dot("dot"),
        add("add"),
        sig1("sig1"),
        sig2("sig2"),
        sig3("sig3")
    {
        gain1.x(x);
        gain1.y(sig1);
        gain1.set_timestep(1, sc_core::SC_MS);

        add.x1(sig1);
        add.x2(sig3);
        add.y(sig2);

        dot1.x(sig2);
        dot1.y(y);

        gain2.x(y);
        gain2.y(sig3);
    }

private:
    sca_lsf::sca_signal sig1, sig2, sig3;
};
```

4 VHDL

Il VHDL, che sta per *VHSIC Hardware Description Language*, dove *VHSIC* è la sigla di *Very High Speed Integrated Circuits* è un linguaggio di descrizione dell'hardware nato nel 1987 per un progetto del Dipartimento di Difesa degli Stati Uniti d'America. Esso costituisce uno standard (IEEE 1076) e viene utilizzato sia per la simulazione sia per la sintesi di modelli hardware.

VHDL supporta sia una descrizione del modello di tipo comportamentale, che chiameremo *behavioral modeling*, sia una descrizione più accurata a livello strutturale che chiameremo appunto *structural modeling*. Entrambe le descrizioni sono sintetizzabili, a patto che si utilizzi un sottoinsieme specifico di costrutti e che si seguano degli stili predefiniti quando si usa la descrizione behavioral.

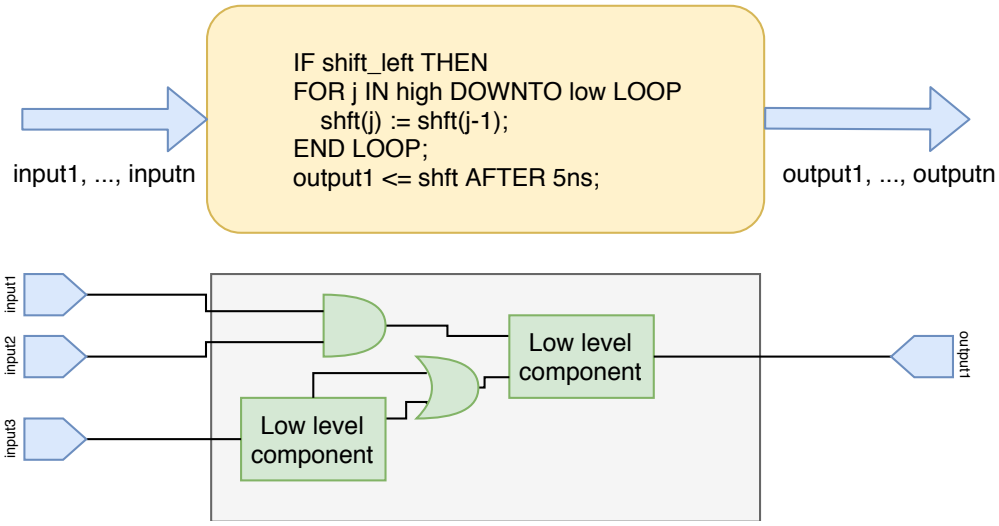


Figura 4.1: Differenza tra *behavioral modeling* e *structural modeling*

4.1 Design units

VHDL mette a disposizione 4 unità base per la modellazione:

- **Entity**: costituisce "l'interfaccia" del modello (si può anche pensare come un simbolo in uno schema di circuito)

- **Architecture**: costituisce la funzione effettiva del modello (si può pensare come all'effettivo schema di componenti primitivi)
- **Configuration**: si utilizza per associare un'interfaccia ad un'architettura specifica
- **Package**: collezione di informazioni che possono essere caricati in altri modelli, come le librerie

4.1.1 Entity

Una *entity* possiede la seguente struttura:

```
ENTITY <entity_name> IS
    -- generic declarations
    -- port declarations
END ENTITY;
```

Il nome dell'interfaccia può essere una qualsiasi stringa alfanumerica. Nella sezione "`generic declarations`" possiamo passare valori alla `ENTITY` a tempo di compilazione, in modo da poterla rendere parametrizzabile quando viene utilizzata in più occasioni, mentre nella sezione "`port declarations`" si definisco le diverse porte di I/O.

La sezione `generic` presenta il seguente modello:

```
ENTITY <entity_name> IS
    GENERIC (
        CONSTANT tplh, tphl : TIME        := 5 ns;
        -- Note: CONSTANT is assumed and isn't required
        tphz, tplz          : TIME        := 3 ns;
        default_value       : INTEGER     := 1;
        cnt_dir             : STRING      := "up"→ valore della costante
    );                                        → tipo della costante
    -- port declarations                      → nome della costante
END ENTITY;
```

La sezione comincia con la parola chiave `GENERIC`, il cui ambiente viene indicato con delle parentesi tonde seguite da un punto e virgola. Le varie costanti vengono dichiarate seguendo lo schema

NOME COSTANTE : TIPO := VALORE;

e vengono separate l'una dall'altra tramite il punto e virgola (notare che nell'ultima riga è assente). È possibile inoltre dichiarare più costanti in una singola riga, come si vede nella prima e nella seconda, se sono dello stesso tipo e presentano lo stesso valore iniziale. I valori dichiarati all'interno della sezione possono essere sovrascritti durante la compilazione, ad esempio passandoli come parametri di al compilatore.

La sezione "`port declarations`" definisce tutto l'I/O e si piazza dopo le `generic`. Il modello è anche simile:

```
ENTITY <entity_name> IS
    -- generic declarations
    PORT (
        SIGNAL clk, clr : IN    BIT;
        -- Note: SIGNAL is assumed and isn't required
        q               : OUT   BIT
    );
END ENTITY;
```

→ tipo della porta
→ direzione della porta
→ nome della porta

Come per la sezione `GENERIC` anche la `PORT` comincia con un identificativo seguito dalle parentesi, all'interno delle quali si definiscono le diverse porte. Una porta può avere una modalità fra le seguenti:

- `IN`: definisce una porta di solo input
- `OUT`: definisce una porta di solo output
- `INOUT`: definisce una porta bidirezionale
- `BUFFER`: definisce una porta di output, che però può anche essere anche letta dall'interno

4.1.2 Architecture

Una *architecture* descrive la logica interna di un modello, il quale poi esporrà le sue porte di I/O tramite una *entity*. È obbligatorio associare una *architecture* con almeno una *entity*, tuttavia nulla vieta di associarne n.

Tutto i blocchi che vengono dichiarati all'interno vengono eseguiti in parallelo. Questi blocchi vengono denominati *processi*. I processi possono essere scritti in 3 differenti stili:

- *Behavioral*, che esprime come il modello opera
- *Structural*, che esprime il modello direttamente con una netlist
- *Hybrid*, un misto fra i due stili sopra

Lo stile *behavioral* può a sua volta espresso in *RTL* (come in SystemC) oppure in stile *functional*, senza il tempo.

La struttura di una *architecture* è la seguente:

```
ARCHITECTURE <identifier> OF <entity_identifier> IS
    -- Architecture declaration section
    SIGNAL temp     : INTEGER := 1;      -- signal declarations
    CONSTANT load   : BOOLEAN := true;   -- constant declarations
    TYPE states IS (S1, S2, S3, S4);     -- type declarations
    -- Component declarations
    -- Subtype declarations
    -- Attribute declarations
    -- Attribute specifications
    -- Subprogram declarations
    -- Subprogram body
BEGIN
    -- Process statements
    -- Concurrent procedural calls
    -- Concurrent signal assignment
    -- Component instantiation statements
    -- Generate statements
END ARCHITECTURE;
```

4.1.3 Configuration

La sezione *configuration* viene utilizzata qualora vi siano più di una *architecture* associata ad un'*entity*. Questo costrutto permette di assegnare un nome unico alla coppia *entity-architecture*, andando ad indentificare singolarmente una specifica architettura. Nonostante sia supportato dalla sintesi, è più comunemente utilizzato nella simulazione, dove è più probabile avere più architetture definite per una singola *entity*.

La struttura è la seguente:

```
CONFIGURATION <identifier> OF <entity_name> IS
    FOR <architecture_name>
        FOR <instance_name> : <component_name> USE
            <entity>(<architecture>)
        END FOR;
        FOR <instance_name> : <component_name> USE
            <configuration_name>
        END FOR;
    END FOR;
END CONFIGURATION;
```

4.1.4 Package

I *package* sono un modo per immagazzinare informazioni per futuri modelli e possono essere pensati come a delle "librerie" (in maniera analoga al linguaggio Java). Più propriamente, si definisce libreria una directory che contiene uno o più *package*. Il VHDL mette a disposizione due *package* fondamentali sui quali poi vengono costruite le altre librerie, che sono `Standard` e `TEXTIO`, i quali vengono automaticamente inclusi. In aggiunta, è disponibile anche la libreria `IEEE`, definita dallo stesso ente per il linguaggio VHDL e altre librerie dei singoli produttori.

Per utilizzare una libreria all'interno di un progetto sono necessarie due istruzioni, ovvero `LIBRARY` e `USE`. La prima definisce il nome della libreria a cui ci si riferirà in seguito mentre la seconda carica effettivamente la libreria.

Un *package* è costituito da una dichiarazione obbligatoria, nella quale si mettono le dichiarazioni dei tipi e dei sottoprogrammi, e un corpo opzionale nel quale si danno le definizioni vere e proprie dei sottoprogrammi.

La struttura di un *package* è la seguente:

```
PACKAGE <identifier> IS
    -- type declarations
    -- function declarations
END PACKAGE;
PACKAGE BODY <identifier> IS
    -- function definitions
END PACKAGE BODY;
```

Riportiamo anche un possibile esempio:

```
PACKAGE filt_cmp IS
    TYPE state_type IS (idle, tap1, tap2, tap3, tap4);
    FUNCTION compare (variable a, b: integer) RETURN boolean;
END PACKAGE;
PACKAGE BODY filt_cmp IS
    FUNCTION compare (variable a, b: INTEGER) IS
        VARIABLE temp: BOOLEAN;
    BEGIN
        IF a < b THEN
            temp := true;
        ELSE
            temp := false;
        END IF;
        RETURN temp;
    END FUNCTION compare;
END PACKAGE BODY;
```

4.2 Tipi predefiniti

In VHDL esistono una serie di tipi predefiniti. Quelli messi a disposizione dallo standard package (quello incluso di default) sono:

- `BIT`
  Rappresenta semplicemente un bit. Può essere usato:
  - Come valore booleano semplice
    ```
    SIGNAL a_temp : BIT;
    ```
  - Come array di valori booleani
    ```
    SIGNAL temp: BIT_VECTOR(3 DOWNTO 0);
    SIGNAL temp: BIT_VECTOR(0 TO 3);
    ```
- `BOOLEAN`
  Rappresenta un valore booleano. Può assumere solo valore `true` o `false`.
- `INTEGER`
  Rappresenta un numero intero (positivo o negativo).
  ```
  SIGNAL int_tmp: INTEGER; -- 32-bit number
  SIGNAL int_tmp1: INTEGER RANGE 0 TO 255; -- 8-bit number
  ```
- `NATURAL`
  Rappresenta un intero con un range tra 0 e 2^32
- `POSITIVE`
  Rappresenta un intero con un range tra 1 e 2^32
- `CHARACTER`
  Rappresenta un carattere ASCII
- `STRING`
  Rappresenta un array di caratteri
- `TIME`
  Rappresenta un unità di tempo (ps, ns, sec, hr, ecc.)
- `REAL`
  Rappresenta un numero a virgola mobile con doppia precisione

I tipo messi a disposizione dalla libreria IEEE sono principalmente:

- `STD_LOGIC`
  Rappresenta un tipo logico che può assumere i seguenti valori:
  - `U`
    Stato non inizializzato
  - `X`
    Stato sconosciuto

  - `0`
    Stato logico 0
  - `1`
    Stato logico 1
  - `Z`
    Stato di alta impedenza
  - `W`
    Segnale debole, impossibile distinguere 0 da 1
  - `L`
    Segnale debole, probabilmente è 0
  - `H`
    Segnale debole, probabilmente è 1
  - `-`
    Don't care

- `NUMERIC_STD` Famiglia di tipi per aritmetica senza segno o con segno. I più usati sono i tipi `UNSIGNED` e `SIGNED`.

4.3 Un contatore a 4 bit

Vediamo un esempio completo di un contatore a 4 bit in VHDL:

counter.vhdl

```
-- Including IEEE library for the std_logic types
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- Defining the entity
entity counter is
    -- Don't need any generic, straight to the port declaration
    port(
        clk     :  in   std_logic;
        reset   :  in   std_logic;
        enable  :  in   std_logic;
        count   :  out  std_logic_vector(3 downto 0)
    );
end counter;

-- Defining the architecture
architecture behav of counter is
    -- Declaration section
    signal pre_count: std_logic_vector(3 downto 0);
begin
    -- Definition section
    process(clk, enable, reset)
    begin
        if reset = '1' then
            pre_count <= "0000";
        elsif (clk='1' and clk'event) then
            if enable = '1' then
              pre_count <= pre_count + "1";
            end if;
        end if;
    end process;
    count <= pre_count;
end behav;
```

5 Internet of Things (IoT)

5.1 Cos'è IoT

Internet of Things è una realtà ormai consolidata che vede l'unione di architetture cloud e sensori posti nell'ambiente per raccogliere dati ed elaborarli. I sistemi embedded "tradizionali" non sono stati sempre progettati per la comunicazione verso internet. Ora invece durante la progettazione non si pensa più alla singola istanza dell'oggetto ma a un sistema distribuito in grado di far comunicare più oggetti tra loro e di poter raccogliere e condividere i dati tramite la rete. In particolare, IoT è definita come un sistema provvisto di numerosi dispositivi, identificati da un codice univoco, in grado di trasferire dati verso internet senza l'ausilio d'interazione umana, ovvero, ci sono sensori che permettono l'acquisizione di dati dall'ambiente.

5.1.1 Scenari di applicazione

Gli scenari in cui i sistemi IoT sono presenti sono molti, in particolare:

- **Per la salute**: in questo primo scenario vediamo che dispositivi come uno *smartband* oppure uno *smartwatch* possono essere considerati come sistemi IoT. In questo caso, infatti, l'oggetto è in grado di rilevare senza l'intervento dell'uomo i passi, i movimenti, il battito cardiaco, ecc..., computando tutto in tempo reale e salvando le informazioni raccolte in cloud.

Figura 5.1: IoT Health

- **Per la casa (Smart Home)**: il secondo scenario è rivolto alla domotica e all'uso di dispositivo per interagire con l'ambiente, come ad esempio l'apertura di una porta attraverso attuatori, controllo automatico della temperatura oppure il comando remoto delle luci tramite lo smartphone.

Figura 5.2: IoT SmartHome

- **Per la mobilità (Smart mobility)**: il terzo scenario prevede l'utilizzo dell'IoT per la creazione di smart mobility. L'individuazione di parcheggi liberi è un esempio di sistema IoT dove, attraverso sensori e/o telecamere ti aiutano a trovare parcheggio. Tutto questo, infatti, è basato su sensori che in tempo reale acquisiscono dati e trasmettono all'utente i dati richiesti.

Figura 5.3: IoT smart mobility

- **Per l'energia (Smart grid/ energy)**: il quarto scenario vede l'IoT in un contesto in cui si ha una gestione intelligente dell'energia. Questo sistema in maniera autonoma è in grado di indirizzare l'energia in tempo reale a dispositivi in cui in quel momento serve più energia rispetto ad altri, o eventualmente cercare di equilibrare l'energia nel migliore dei modi.

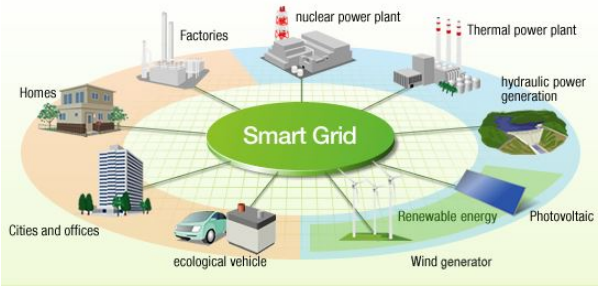


Figura 5.4: IoT SmartGrid

- **Per l'agricoltura**: il quinto scenario prevede l'utilizzo di sistemi IoT per l'agricoltura smart. Attraverso sensori di temperatura il sistema riesce a gestire il flusso d'acqua uscente per le piantagioni oppure riesce a capire quando accendere o spegnere l'erogazione di acqua.

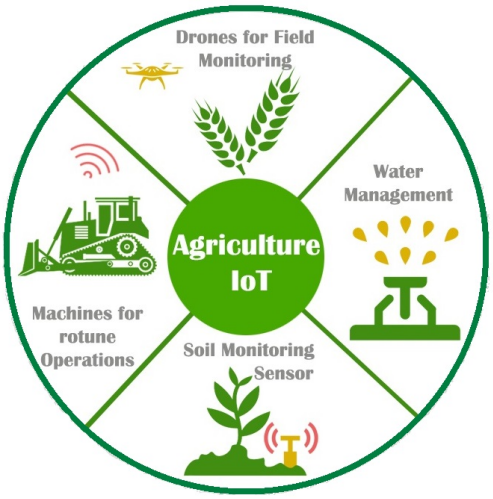


Figura 5.5: IoT Agricolture

- **Per le città smart (Smart Cities)**: il sesto scenario prevede l'applicazione di smart cities, ovvero un insieme di strategie di pianificazione urbanistica che mirano all'ottimizzazione e all'innovazione dei servizi pubblici così da mettere in relazione le infrastrutture materiali delle città.

Figura 5.6: IoT Smart Cities

- **Industria 4.0**: l'ultimo scenario che vede lo sviluppo di sistemi IoT sono le industrie 4.0, denominate IIoT, nelle quali si cerca d'incrementare la produzione di oggetti e di migliorare la gestione e l'organizzazione dell'azienda stessa. L'industria 4.0 vuole rappresentare la quarta rivoluzione industriale. L'obiettivo principale nell'industria 4.0 è arrivare a una produzione massimamente diversificata, quindi rendere la produzione molto flessibile dal punto di vista produttivo. Il concetto di linea di produzione ben organizzata e inflessibile ora, con questo modello, viene abbandonata. Si inizia a parlare di Internet of Services, ovvero, la distinzione tra oggetti e servizi è talmente sottile che tutta l'infrastruttura viene vista e usata come un servizio. All'interno dell'industria anche le macchine devono adeguarsi ai sistemi IoT, infatti, le macchine devono avere le seguenti caratteristiche:
  - Inter-operabile
  - Virtualizzabile
  - Real-time
  - Decentralizzata
  - Orientata ai servizi e modulare

  Con queste caratteristiche una macchina è in grado di dialogare con il sistema e con l'ambiente in cui si trova cercando quindi di inviare informazioni sul proprio stato. Alla fine i dati devono essere interpretati, perché se a un dato non viene data la semantica non serve a nulla. Una volta che i dati sono raccolti, vengono spediti nel cloud in cui verranno successivamente analizzati, rendendo possibile l'ottenimento di informazioni sullo stato dell'intero sistema e di conseguenza, se necessario, riuscire a modificare la produzione in tempo reale.

Figura 5.7: IoT Industry 4.0

5.1.2 Ingredienti per una soluzione IoT

La realizzazione di IoT è composta da alcuni layers:

- **Device layer**
  - Sensori
  - Attuatori
  - Tagged devices/objects
- **Communication layer**
  - Device gateway
  - Smart gateway
  - Protocollo di comunicazione
  - Protocollo applicativo
  - Protocollo industriale
- **Core platform**
  - Protocol Gateway
  - IoT Messagging Middleware
  - Data storage
  - Data aggregation/filter
- **Analytics platform**

- Processo di streaming
- Machine Learning
- Eventi e report

È importante ricordare che, oltre, all'architettura, alla raccolta e gestione dei dati è importante avere un buon livello di sicurezza per l'invio e la gestione dei dati sensibili.

Nel **device layer** ci sono i dispositivi in sè, che permettono di acquisire dati e inviarli tramite internet. Di solito questi dispositivi embedded dotati di un micro-controllore, una memoria, eventuali interfacce e un sistema operativo dedicato in base al contesto in cui sono piazzati. Il più delle volte sono dotati di architetture low power per poter risparmiare energia, in quanto alimentati da batteria. Qualsiasi dispositivo IoT è dotato di sensori, senza i quali non sarebbe possibile la raccolta dati.

I sensori possono essere di qualsiasi tipologia: elettronici, meccanici, idraulici, ecc. Ciò che li contraddistingue è il fatto di essere "smart", ovvero dotati già a livello sensoriale di componenti logiche che eseguono un minimo di data-processing.

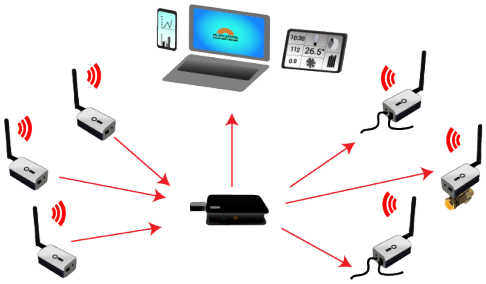


Figura 5.8: IoT sensors

Gli **attuatori** sono dispositivi che permettono d'interagire con l'ambiente in cui si trovano, in base a un'azione inviata come input.

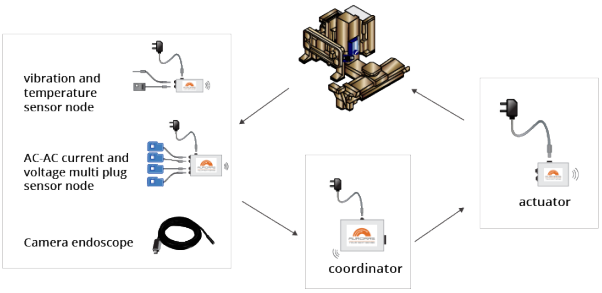


Figura 5.9: IoT actuators

Esistono anche dispositivi non alimentati che permettono però di poter interagire attivamente con l'ambiente. Spesso questo avviene tramite meccanismi come l'RFID

(Radio Frequency Identification), che tramite circuiti integrati alimentati durante l'utilizzo inviano i dati contenuti al lettore, che poi si occuperà dell'eventuale invio dei dati su internet.

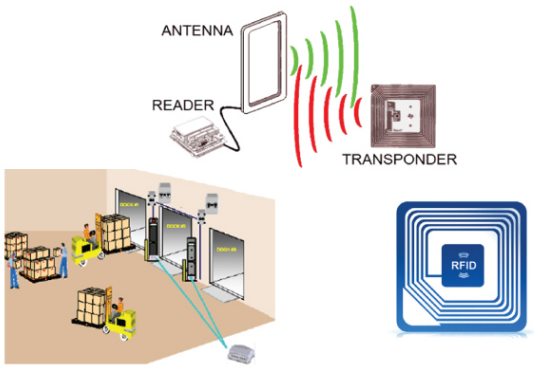


Figura 5.10: IoT RFID

Per avere invece una comunicazione **bidirezionale** si usa l'NFC (Near Field Communication), che lavorando con frequenze molto corte costringe l'interazione ad avvenire in maniera ravvicinata.

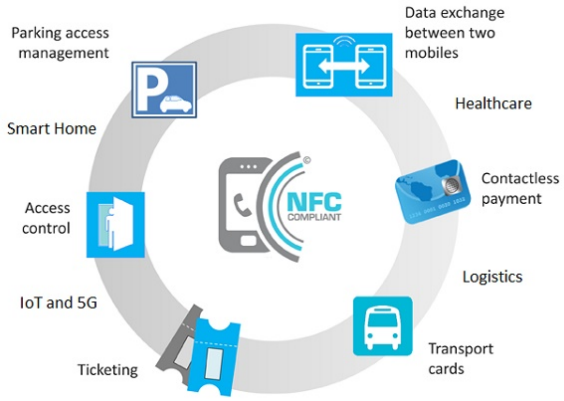


Figura 5.11: IoT NFC

Altra tecnologia molto diffusa sono i **beacons**, un'ulteriore forma di tagged devices, che sfrutta la tecnologia **bluetooth LE (Low Energy)** per trasmettere un segnale in broadcast per indicare, di solito, la posizione ad altri ricevitori bluetooth. La trasmissione bluetooth si può usare in diversi contesti.

Figura 5.12: IoT Beacon

Il **communication layer** indica la parte di comunicazione dal dispositivo locale verso la rete internet. Qui si possono distinguere dispositivi che sono in grado di connettersi alla rete, attraverso Wi-Fi o rete cellulare (3G/4G) e dispositivi che non sono in grado di connettersi direttamente a internet e quindi per poterlo fare hanno bisogno di connettersi a un **gateway o un hub**, con il quale è possibile arrivare alla rete esterna. Spesso il gateway sono anch'essi "smart": esso ha una propria memoria interna e delle specifiche appliccazioni embedded che permettono di analizzare i dati direttamente sul posto prima di instradarli verso un altro gateway o un hub centrale.

Più specificamente, gli smart gateway permettono:

- **Filtraggio e aggregazione di dati**: cercano di dare una semantica ai dati ricevuti e scartare quelli inutili.
- **Buffering**: quando un dispositivo rimane offline, hanno un certo buffer per mantenere in memoria i dati trasmessi
- **Sincronizzazione**: aggiornamento dei dati mancanti o incompleti

Un'applicazione sul campo di questi smart gateway può essere la smart city. Per esempio, un lampione può fare da smart gateway e comunicare i dati da una zona a un'altra. Gli stessi smartphone possono essere considerati dei gateway, infatti nel momento in cui comunicano con una smartband oppure uno smartwatch, sono loro che ricevono i dati e li inviano alla rete.

I **protocolli di comunicazione** sono responsabili per la connettività di rete. Alcuni protocolli più usati per una connessione diretta verso internet sono:

- Wi-Fi
- Ethernet
- Rete cellulare (3G,4G, in futuro 5G)

Non tutti i protocolli sono in grado di connettere i dispositivi direttamente a internet, ad esempio:

- Bluetooth (LE)
- RFID
- NFC
- Zigbee/Z-wave

I **protocolli di applicazione** poggiano, come nei normali PC general purpose, sul protocollo TCP/IP. Nel caso dei dispositivi IoT viene data molta enfasi all'uso del cloud, motivo per cui si sono creati protocolli specifici. In particolare, i due procotolli più usati e più "famosi" sono **MQTT** e **AMQP**. **MQTT** è stato pensato per trasferire dei dati che partono da un database locale, per esempio all'interno di un'azienda. **AMQP** a differenza di MQTT garantisce una minore latenza nel trasferimento dei dati, questo perchè l'uso del protocollo AMQP è stato pensato per applicazioni IoT, infatti i dati che vengono trasferiti sono di piccole dimensioni e la trasmissione può essere asincrona.

Alla base del protocollo **MQTT** (Message Queue Telemetry Transport) si trova il classico concetto di **publisher e subscriber**. Viene usato principalmente per:

- Invio di dati con una taglia molto piccola
- Dispositivi dotati di hardware limitato
- Rete con poca banda e alta latenza

Il payload dei messaggi essendo binario risulta molto compatto. Come detto prima, questo protocollo sfrutta il concetto di **publisher e subscriber**. Tutti i dispositivi IoT che sono presenti nella rete si connettono a un **broker MQTT**, in grado di gestire molteplici connessioni. Un client, quando si connette al broker, si può iscrivere come publisher o come subscriber in uno o più topic, che è una stringa in formato UTF-8 che viene usata dal broker per filtrare i messaggi da inviare ai client sottoscritti a quello specifico topic. I topic possono essere composti da un solo livello o a più livelli, infatti un client può decidere di sottoscriversi ad un solo livello o più usando le **wildcards**. Il protocollo MQTT prevede anche il **QoS** (Quality of Services), che garantisce entro certi limiti spedizione e ricezione del messaggio. Sono previsti tre livelli di affidabilità del QoS:

- 0: Al massimo una volta
- 1: Almeno una volta

- 2: Esattamente una volta

Se vogliamo possiamo pensare all'invio del messaggio come ad un ack a livello TCP/IP, perchè anche qui ci sono sempre due parti di invio e ricezione del messaggio.

- I messaggi che partono dal client e arrivano al broker: in questo caso il livello di QoS dipende dal livello che il client ha impostato nel messaggio in cui ha appena inviato.
- I messaggi dal broker verso il client: in questo caso il livello di QoS usato dal broker **non** è quello usato nel messaggio appena inviato dal client, ma il livello di QoS che il client aveva impostato in precendeza.

Ogni volta che un client si collega al broker, oltre al messaggio di benvenuto che si scambiano all'inizio, inviano anche il testamento di fine connessione, nel quale viene definito ciò che deve fare il broker una volta che questo si disconnette dalla rete. Il broker, infatti, tiene in memoria il testamento finché il client non esce dalla rete.

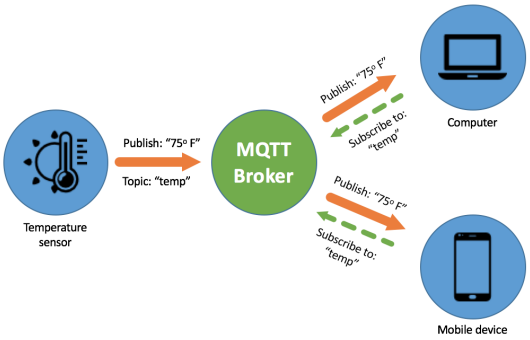


Figura 5.13: MQTT

Il protocollo **AMQP** (Advanced Message Queuing Protocol), fu creato da JP Morgan® per applicazioni business. AMQP è un protocollo binario e supporta sia il modello basato su coda che quello publisher e subscriber. In AMQP non è presente il testamento e non sono previsti metadata nei messaggi. Condivide con MQTT alcune caratteristiche come il QoS a 3 livelli.

In ambito industriale ci sono due famosi protocolli di comunicazione:

- **SCADA**
- **ModBus**

In particolare, il protocollo ModBus permette di dialogare con le macchine e di estrapolare i dati in maniera da poter attribuirgli un significato. Infatti, assieme a ModBus viene spesso utilizzato **OPC-UA** che è uno standard in cui i dati devono assumere una certa struttura per poter essere interpretati dalle applicazioni in futuro applicate.

Quindi OPC-UA mi permette di definire una certa **semantica** ai dati che la macchina produce.

Il **core platform** lo possiamo vedere come una sorta di grande hub o gateway in grado di dialogare con più protocolli diversi tra loro e di connettere moltissimi dispositivi assieme. È composto da:

- **Protocol Gateway**
- **IoT Messagging Middleware**
- **Data storage**
- **Data aggregation/filter**

Il **Protocol Gateway** è la parte principale del Core Platform perchè è in grado di garantire la trasparenza di più protocolli diversi tra loro.

L'**IoT Messagging Middleware** è un software presente nel Core Platform molto simile ai protocolli visti in precedenza (MQTT e AMQP) perchè adotta il modello publisher e subscriber. Prima abbiamo definito il Core Platform come un grande hub che è in grado di gestire moltissimi dispositivi connessi ad esso: questo è possibile grazie all'IoT Messagging Middleware che permette di avere elevate performance e tolleranza al guasto. Questo livello permette, inoltre, di lavorare con meta-modelli, che vengono definiti attraverso l'emissione di dati dai diversi dispositivi. I meta-modelli permettono di creare una soluzione dipendente dai dati inviati dai dispositivi e quelli che vengono usati piattaforma IoT. Tramite queste astrazioni il software è in grado di gestire più tipi di dati dai diversi protocolli. Infine, l'IoT Messagging Middleware permette uno storage di particolati intervalli di dati cercando di analizzare i dati che gli stanno arrivando.

Nel livello **data storage** come dice il termine stesso è il livello che si occupa di immagazzinare i dati in Database per poter avere una copia locale. I Database più utilizzati sono:

- **MongoDB**
- **CouchDB**

Nel livello **Data aggregation and filter** i dati vengono convertiti in alcuni formati specifici per poter poi essere mandati a un software in grado di creare dei modelli di dati per essere interpretati.

L'ultimo livello, per una soluzione IoT, è l'**Analytics Platform Layer** il quale permette di analizzare una grande mole di dati, di predirre un dato e di eseguire una determinata azione. Quest'ultimo livello è composto da:

- **Stream processing**
- **Machine learning**
- **Actionable insights**

Nello **stream processing** vengono ricevuti i dati in tempo reale dai dispositivi e analizzati in maniera da trovare delle correlazioni tra i differenti dati per poter poi successivamente usare quest'informazione per migliorare qualcosa.

Per **machine learning** si intende l'apprendere delle nozioni sui dati che abbiamo senza esplicitamente creare degli algoritmi per farlo. Ci sono 4 fasi principali per utilizzare questo meccanismo:

- Capire il problema e trovare una soluzione
- Pulire i dati e analizzare i dati filtrati
- Creare dei modelli (es. IoT Messagging Middleware)
- Fare training sui dati

Detto questo ci sono 3 modi per eseguire machine learning:

- **Supervised learning**: i dati che arrivano dall'esterno sono già etichettati, pronti per essere interpretati e riuscire a predirre i dati successivi.
- **Unsupervised learning**: i dati che arrivano dall'esterno non sono già stati manipolati e qui è l'algoritmo stesso che capisce dove sono le correlazioni tra i dati.
- **Reinforcement learning**: il sistema impara dai dati che riceve dall'ambiente esterno, per esempio, attrverso un sensore.

Il livello **actionable insights** permette di creare report degli eventi in base ai dati che vengono analizzati.

5.1.3 Cloud

Anche se non è parte integrante di una soluzione IoT il **cloud** è la parte fondamentale per rendere il sistema IoT. Per cloud si intende un'area in cui i dati da diversi dispositivi vengono inviati attraverso internet. Il cloud può essere di due tipi:

- **Privato**: la gestione e il mantenimento del cloud è a carico di chi ha creato il cloud, senza potersi appoggiare ad altri enti. Molte aziende utilizzano questa tipologia di cloud per paura di condiviedere i dati ad altri enti.
- **Pubblico**: la gestione e il mantenimento del cloud è a carico di chi gestisce il servizio di cloud. Chi usufruisce del servizio paga lo spazio che utilizza.

La soluzione migliore è quella di creare un **cloud ibrido**, ovvero, che ha sia la parte di computazione privata, esempio dati sensibili di un'azienda, e la computazione dei dati ormai filtrati all'esterno, senza conpromettere la "privacy" dei dati.

Nel momento in cui la scelta ricade su un cloud pubblico ci possono essere delle categorie di servizi offerti come: **IaaS – PaaS - SaaS**.

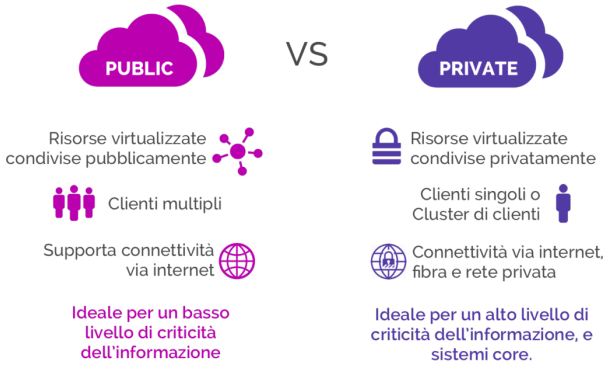


Figura 5.14: Cloud

IaaS (Infrastructure as a Service) Questo tipo di servizio include offerte come lo spazio virtuale su server, connessioni di rete, larghezza di banda, indirizzi IP e bilanciatori di carico. Fisicamente, il gruppo di risorse hardware viene estratto da una moltitudine di server e reti solitamente distribuiti presso numerosi data center, la cui manutenzione è responsabilità del provider del cloud.

PaaS (Platform as a Service) Consente agli utenti di creare applicazioni software utilizzando gli strumenti forniti dal provider. Le funzioni che possono essere incluse in un'offerta PaaS sono:

- Sistema operativo
- Ambiente di programmazione lato server
- Sistema di gestione database
- Software server
- Assistenza
- Archiviazione
- Accesso di rete
- Strumenti per progettazione e sviluppo
- Hosting

SaaS (Software as a Service) Indica qualsiasi servizio cloud tramite il quale i consumatori possono accedere ad applicazioni software tramite internet. Spesso viene chiamato "software on demand", in quanto il suo utilizzo è più simile a un noleggio del software che al suo acquisto.

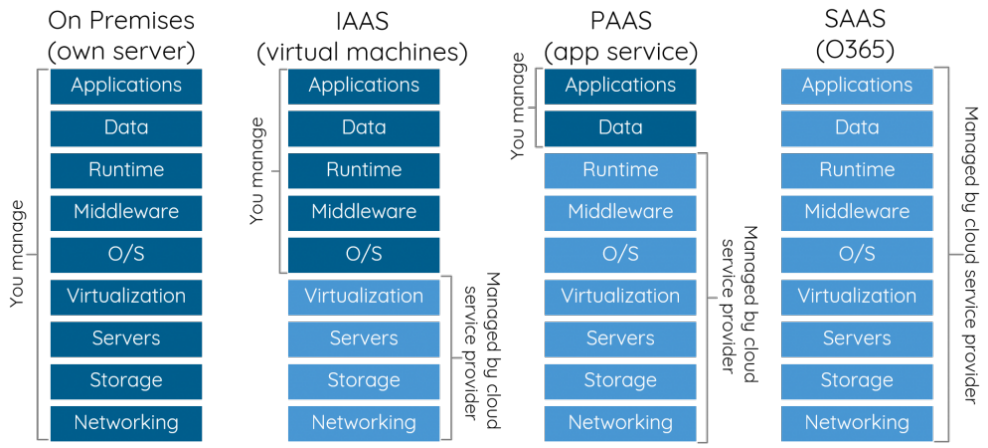


Figura 5.15: Cloud Services models

5.1.4 Architettura cloud per IoT

Alla fine, come detto all'inizio, una soluzione IoT oltre a tutti gli ingredienti visti fin'ora ha bisogno della rete internet per poter comunicare con il mondo esterno, ma soprattutto, del cloud per poter raccogliere in un unico spazio i dati raccolti e poterli esaminare e ottenere dei risultati validi.

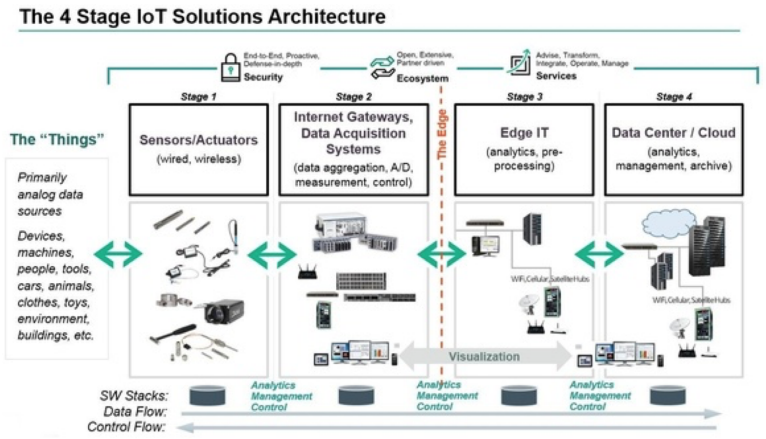


Figura 5.16: Cloud Architecture for IoT Solution