Lezioni del 05/03/14, 06/03/14, 07/03/14

PRESENTAZIONE DEL CORSO

Tre parti:

- Programmazione avanzata.
- Architettura dei calcolatori.
- Sistemi digitali.

Prova scritta (3h): programmazione in C/C++, progetto di circuiti digitali, domande teoriche sulla parte di architettura. Massimo 20/30.
Progetto: in C++, consegnare relazione in pdf, consegnare codice sorgente ed eseguibile. Massimo 10/10.
DA CONSEGNARE ENTRO IL 23 GIUGNO!

TREND INDUSTRIA MICROELETTRONICA

Just for fun: provare a guardare su http://public.itrs.net quelle che sono le roadmap.

LEGGE DI MOORE: fondatore di IBM, nel 1965 intuisce attraverso dati sperimentali che il numero di transistor che si possono mettere per unità di superficie raddoppia ogni due anni-> andamento esponenziale. Oggi ha senso di continuare a pensare a un trend esponenziale per sempre? Forse non proprio… Giusto per avere due numeri, dall'Intel 4004 al core i7 siamo passati da 2k a 2billion transistor. Se pensiamo a un ARM1 del 1986 e a un processore del 2006, si ha avuto un guadagno 800x per la potenza e di 50x per le dimensioni usate. In 5 generazioni di tecnologie si è quindi aumenta di 20 volta la densità di dispositivi ma la potenza assorbita non è diventata un ventesimo di quella consumata originariamente -> problemi di consumo.

SYSTEM-on-CHIP and IP CORES

Oggi un Integrate Chip (IC) ha 9 layers e 7Km di connessioni fra i vari transistor. Il trend attuale è quello di portare concettualmente e fisicamente l'intero sistema su di un unico chip -> **system-on-chip (SoC)**. Per esempio su di un chip ho ROM, RAM, USB, MPEG; GSM, etc.
Tutti i vari blocchetti di cui è composto un SoC sono chiamati EMBEDDED CORES-> moduli funzionali di blocchi pre-progettati o analogamente IP core (Intellectual Property). Infatti in genere io compro i singoli cores e poi li metto insieme e li assemblo sul chip (altrimenti lo sviluppo da zero di tutto sarebbe troppo costoso). Gli IP CORES compiono il lavoro che prima svolgevano i singoli chip/interfacce sui PCB (che invece sono chiamati **system-on-board**). A sua volta fra un po' di anni i singoli SoCs potranno essere utilizzati per circuiti più complessi (Moore's law).

Definizione di **IP-cores**: un circuito hardware che viene fornito al designer assembler dall' IP provider. Posso decidere di utilizzare gli IP-core per varie ragioni:perché non ho know-how, troppo sbattimento, problemi di compatibilità (trovo tutto già fatto e i protocolli sono già rispettati), per il time-to-market (il tempo che intercorre fra la decisione di fare nuovo prodotto al primo esemplare sul mercato, se riduco il time-to-market riesco ad avere profitti maggiori, la perdita che ho non è lineare al tempo in ritardo). Gli IP-core vengono forniti come soft-core e come hard-core.

- ***Soft-core***: compro dalla ditta il sorgente, una descrizione di come è fatto il chip e non il chip in se e attraverso un tool di sintesi riesco a generare il codice per realizzare il mio SocS. Ovviamente sono criptati.
- ***Hard-core***: compro il pezzo così com'è e lo devo mettere come mi viene fornito.

All'interno di un IP-core posso avere un altro IP-core (possono essere annidati)-> avrò una gerarchia.

Gli ip-cores stanno diventando molto diffusi e il loro valore aggiunto deriva dai driver a basso livello, dai programmi che ci posso fare girare.

"The processor is the NAND gate of 21th century".

POLLACK'S RULE: regola sperimentale, in una tecnologia le prestazioni sono proporzionali alla radice quadra della complessità. Questo potenzialmente è un problema, una conseguenza è il **MPSoC** (Multi Processor System on Chip)-> non usiamo più un singolo processore ma architetture che sono fatte da più processori in modo da garantire una certa linearità fra performance e superficie occupata. Per questo conviene utilizzare un dual/quad core, perché forniscono potenza maggiore e prestazioni molto migliori rispetto a un unico mega-processore. Ok, ma se devo fare un sistema da 1G di transistor, faccio 10 da 100M, 100 da 10M? questo è il dilemma attuale dei sistemisti…

Multi-core: quando su uno stesso chip ci sono n copie dello stesso processore.
Many-core: quando su uno stesso chip ho più processori diversi (configurazione più comune).

Esempio: il Titancray XK7 (secondo calcolatore più potente al mondo) è formato da 18K CPU AMD Opteron più 18K acceleratori grafici (CGPU Nvidia Tesla).

Ma la legge di Moore può durare per sempre? No, ma il per sempre può essere spostato in là, si cerca di mantenere l'andamento esponenziale in tre modi:

- ***MORE MOORE (MM)***: la tecnologia normalmente avanza (e.g. miniaturizzazione dei componenti).
- ***MORE-THAN-MOORE***: la tecnologia può mettere su uno stesso chip tecnologie diverse (ottica, meccanica, etc.).
- ***BEYOND CMOS***: significativa rivoluzione dei materiali usati, fare dispositivi in modo diverso. Per esempio le universal memory-> prestazioni di una RAM ma non volatili-> l'architettura cambierà notevolmente (non ci sarà più bisogno di una memoria centrale veloce e una di massa).

Lezione del 06-03-2014

PACKAGING

Siamo passati dai **PCB** (Printed Board Circuit) a doppia faccia con vari layers (per i test di fine produzione si fanno radiografie e TAC per capire se le saldature sono fatte bene, non solo analisi visuale, anche perché spesso i piedini non sono di fianco ma sotto), agli MCM (multi-chip module) dove su un supporto di silicio sono collegati altri chip, ai **SiP (System in a package)**. Con il termine SoC indichiamo un intero sistema su di un unico chip, il SiP indica invece l'intero sistema su di un unico componente (sul SiP posso vedere i vari moduli che lo compongono, in un SoC tutto è dentro il package) . Il passo successivo (ed è la situazione attuale) è anziché mettere i componenti uno di fianco all'altro è il metterli **fisicamente uno sopra l'altro**. Il problema di questa soluzione è la *dissipazione di calore* (una soluzione può essere quella di fare dei camini

di rame come dissipatori). Adesso si parla di circuiti integrati in 3D-> dispositivi uno sopra l'altro, i vantaggi sono: occupano meno spazio, piste meno lunghe-> velocità maggiori-> posso andare a frequenze maggiori. In applicazioni per il mobile è normale avere sotto la memoria e sopra il processore. In questo ambito si parla anche di TSV (through silicon via)-> connessioni verticali per comunicare fra i pin esterni e il dentro.

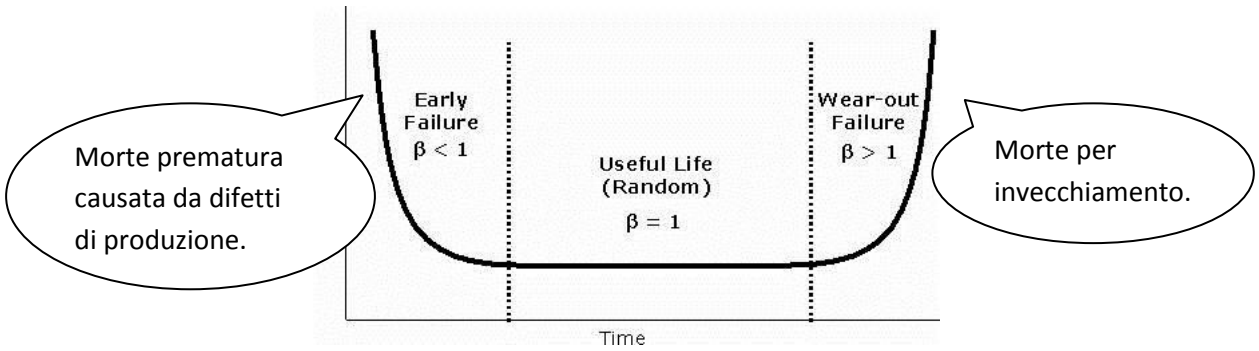
COMUNICAZIONI SUL CHIP

In un SoC si possono i disporre i dispositivi per risparmiare spazio e poi fare le connessioni ma è un casino (grande garbuglio di connessioni), ora si è passati ai **NoC (Net-on-Chip)** -> standard secondo il quale i componenti vengono messi su di una griglia e poi collego i componenti a una rete locale attraverso switch e router (alcuni utilizzano proprio il protocollo ethernet per comunicare fra i vari componenti sul chip). (http://interscience.in/IJIC_Vol1Iss4/48-52.pdf). In questo caso vengono utilizzati gli stessi algoritmi di routing (instradamento dei pacchetti) e sono presenti gli stessi problemi che avrei su una rete locale (e.g. come posso recuperare i pacchetti?).

NEW THREATS

Hardware Trojans: possono essere aggiunte a bordo del mio sistema delle backdoor per malicious modification. Nel 2007 una nazione ha deciso di bombardare una centrale nucleare di un'altra nazione ma i radar della nazione target non hanno rilevato l'attacco-> li hanno fregati. Il problema è capire dov'è il baco, a quale livello del progetto di un IC c'è stato il casino. In pratica potrebbe essere chiunque (chi fisicamente fa le maschere, chi progetta, il tool di sintesi, gli ip-cores comprati da altri, etc).
La soluzione sarebbe il farsi tutto in casa ma funziona più o meno, i costi sarebbero altissimi e non tutti i costruttori possono affrontare questi tipi d'investimenti.

Un altro aspetto sono i ICs contraffatti, i costi stimati legati agli Ics contraffatti dei paesi all'interno del G20 sono circa 1.5 trillion $. Per esempio una siliconfab non fa 10^6 pezzi ma un po' di più e poi li mette sul mercato nero, oppure ci sono gruppi che prendono dalle vecchie board i pezzi, li dissaldano e poi li rimettono in giro cambiando l'intestazione sul packaging. Al di là dell'aspetto morale, ma se chi mi vende il componente di seconda mano prima di darmelo fa bene i test ad arte, che problema c'è? Il problema è che il tasso di guasto in funzione del tempo di un componente ha un andamento a vasca da bagno.



Fonte: http://cdn2.content.compendiumblog.com/

Il problema è se lo compro prima della sua morte, sembra funzionare ma poi presto mi morirà (la marina USA ha perso un paio di caccia perché ha comprato componenti che erano alla fine del loro ciclo vita).

TECHNOLOGY'S IMPACTS ON SOCIETY

Il dato di fatto è che l'evoluzione tecnologica ha fatto si che la produzione dell'industria microelettronica sia qualcosa facilmente reperibile sul mercato e utilizzata largamente sull'elettronica di consumo.

HIGH VOLUME: la SanDisk vende 10 milioni device/week (remember il time-to-market applicato a questo caso). Nel 2005 l'ARM vendeva 54 pezzi al secondo (problemi di collaudo, se la catena di produzione ha un guasto devo riuscire ad accorgermene subito).

PRESENZA PERMEASIVA: i computer non saranno più percepiti come tali, ambiente intelligente, cyber-physical systems (per esempio gestione delle acque in modo integrato di una città come Torino, gestione sensori e informatica).

Un sistema embedded è un sistema nel quale l'utente non percepisce che sta interagendo con un sistema di elaborazione. C'è un sistema di elaborazione che è nascosto all'utente finale. Nel 2010 c'erano 3 sistemi embedded per persona sulla terra. Una macchina di oggi ha una capacità di elaborazione molto maggiore dell'Apollo che andò sulla Luna. Una macchina moderna ha fino a 100 processori (e non scrausi, tutti uguali e tutti potenti così ho un unico ambiente di sviluppo e non 3000).

COSTI: oggi per la progettazione di un SoC avanzato a 28 nm ha un costo stimato fra 80-100 milioni di euro solo di NRE (investimenti non riciclabili per altri dispositivi). Questo pone parecchi problemi, certi investimenti sono giustificati da avere un bel po' di mercato potenziale alle spalle. Le library per gli IP-cores sono aumentate di costo del 125% nelle ultime due generazioni. Una linea di produzione per un SoC a 14 nm costa più di 10 miliardi di euro. Il problema è che il 36% delle fabbriche e il 45% della capacità di produzione sono in zone a rischio (sociale o geografico). Ci sono solo 4 silicon companies (intel, samsung e altre due). Quindi i costi di produzione sempre più elevati ma i prezzi finali sono sempre più bassi. Siamo passati da 5.52$ nel 1954 a 1n$ nel 2004 per un transistor.

AFFIDABILITA': **dependability**: proprietà di un sistema di garantire l'erogazione del servizio richiesto. Per essere dependable un sistema deve avere meno guasti e meno problemi delle soglie prefissate. Le lampadine a filamento potrebbero durare una vita, ma dopo un po' nessuno le comprerebbe un po', le ditte che producono lampadine hanno fatto notevoli investimenti per far durare molto meno i filamenti. Il tasso di guasto sul campo passa dal 10% per elettronica di consumo, al 1% per industrial e zero failure per automotive (ingegneristicamente è molto un problema).

Tutto questo riduce la market windows (lasso di tempo in cui vendo quel prodotto) e di conseguenza bisogna drasticamente ridurre il product-life cycle (tempo di progettazione e messa in produzione del prodotto). Per questo non posso più progettare a manina i sistemi digitali (troppo complicato e troppo tempo).

SISTEMI:DEFINIZIONI e TASSONOMIA.

SISTEMA : APPROCCIO GERARCHICO

Def. **Un sistema è un'entità che interagisce con altri entità (hard, soft, essere umani, mondo esterno)**.
L'insieme delle entità con cui il sistema interagisce si chiama **ambiente**.
La **frontiera** è la linea di demarcazione fra il sistema e il suo ambiente.
In ambito informatico e telecomunicazione un sistema è caratterizzato da 4 proprietà fondamentali:

- Funzionalità: cosa fa il sistema e il fatto che funzioni correttamente.
- Prestazioni del sistema.
- Il fatto che sia applicabile a sempre più sistemi, affidabilità e sicurezza (safety è la sicurezza verso le persone che li usano e ai danni che possono causare agli utilizzatori, security è la sicurezza informatica).
- Costo.

Quando progettiamo un sistema dobbiamo tenere conto di tutti questi 4 aspetti. In termini di securitity non è tanto il problema proteggere un dato ma come distruggere definitivamente quel dato.

Lo spazio di un progetto è l'insieme delle cose che devo tener conto mentre lo progetto. Alcune di queste sono evidenti, altre un po' meno (il fatto che il progetto sia fattibile, sia manufacturability cioè che qualcuno possa costruirlo (DFM-> design for manufacturability); la recyclability (in che modo il sistema deve essere riciclato); la reusability (oggi non ha senso progettare solo per risolvere un singolo problema, devo progettare affinché il progetto sia riutilizzabile in un altro progetto)).

Funzione di un sistema: ciò che il sistema si supponga che faccia.
Behavior of a system: what the system does to implement its
Struttura di un sistema: com'è fatto, quali sono i blocchetti che lo compongono (behavior != structure).
Servizio di un sistema: come viene percepito da chi lo utilizza (umano o un altro sistema).

Approccio top-down: al primo passo si vede il sistema prima come un'unica entità, al passo successivo si vede il sistema come un'opportuna connessione di altri blocchi e poi avanti così. Vado per diversi livelli di astrazione, tipicamente mi fermo quando il mio sistema è descritto in blocchetti elementari (building blocks, dipende molto dal contesto in cui sono).

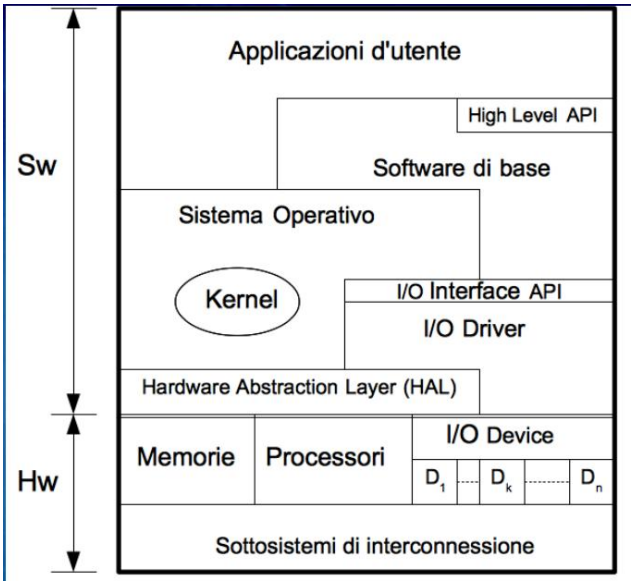
Approccio bottom-up: parto dai blocchetti elementari e li compongo via via per fari blocchi più grandi. Parto dai componenti e li assemblo.

In questo corso utilizzeremo molto l'approccio top-down.

PROCESSING ELEMENTS

Processore, memoria e I/O collegati con un sistema di interconnessioni (uno fra i possibile è il bus, un NoC invece non usa il bus). Nelle celle di memoria memorizzo solo 0 e 1 e nel sistema comunico con questi. Questo è il modo naturale per un sistema di ragionare. Per il software il livello più basso è **l'HAL (Hardware Abstaction Layer)** (e.g. il driver è il programma che pilota un dispositivo di I/O). Il sistema operativo comunica con l'HAL e il suo compito è gestire tutta la parte che sta sotto, tipicamente le applicazioni utenti

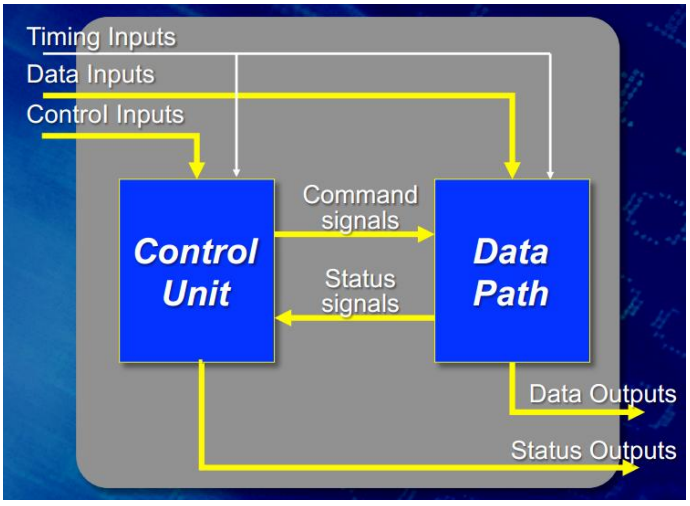
non parlano con l'hardware ma con il sistema operativo, il SO virtualizza il mio sistema. Al limite le applicazioni utente parlano con il software di base (linker, loader, etc.).



FONTE: http://www.testgroup.polito.it

DATA & CONTROL:

Bisogna imparare a distinguerli, questa distinzione c'è sia per HW che per SW. Devo distinguere chiaramente fra ***dati e controlli sia in termini di segnali sia in termini di blocchetti che compongono il sistema***. Tipicamente un sistema ha i primary input e i primary output, questi due si suddividono fra data input/output e control input e status output. Oltre a questi entrano in gioco i timing input (per l'hardware sono fondamentali i clock). E all'interno com'è fatto? Un qualsiasi sistema lo posso modellizzare con una parte che esegue le operazioni (**data path**) e una parte di controllo che deve comandare la parte operativa. La control unit manda i command signals alla data path e la data path manda alla control unit i status signals. I data inputs andranno nella data path, i control inputs vengono mandati alla control unit, i timing input vanno a entrambi.



FONTE: http://www.testgroup.polito.it

SYSTEM TAXONOMIES

Alcune possibili classificazioni.

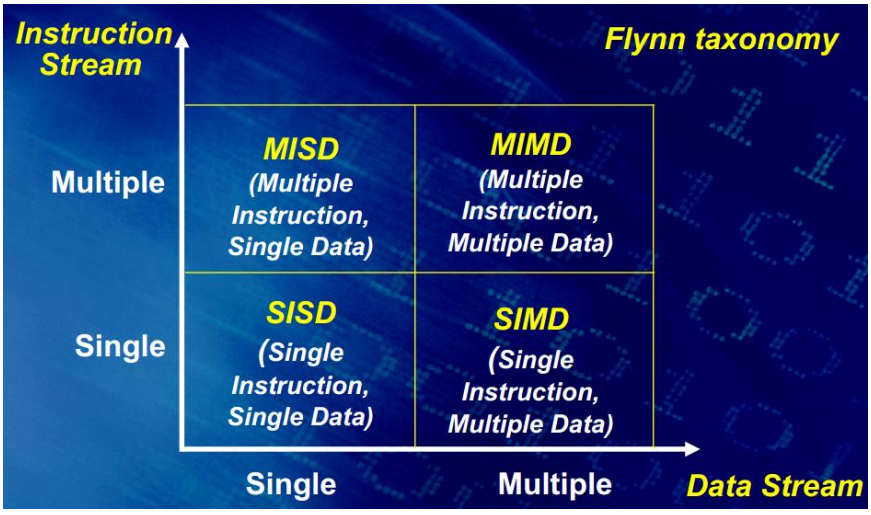
INFORMATION CODING: distinzione fra sistemi analogici e digitali. Un sistema digitale è un opportuno assemblaggio di dispostivi elettronici che trattano informazioni codificate come bits. Quando passo da analogico e digitale gioco su quantizzazione/frequenza di campionamento (la scala che scelgo sull'asse x) e la raffinatezza/precisione con cui prendo i valori (com'e graduata l'asse y, es. se uso 3 o 4 bits).

BEHAVIOR OF OUTPUT SIGNALS rispetto timing signals: distinzione fra **combinational** e **sequential** (il termine inglese "combinatory" è riferito alla combinazione matematica).
Un circuito è combinatorio ⇔ (Iff) i valori che assumono i primary outputs dipendono soltanto dai valori che assumono i primary inputs. I circuiti sequenziali dipendono invece anche dalla storia precedente, cioè dai valori degli ingressi precedenti.
Quando progetto ho tecniche di progettazione molto diverse fra le due. Per esempio un combinational può essere un sommatore, ricevo in ingresso in parallelo i due addendi e in uscita fornisco il risultato. Invece un sequenziale può essere una memoria, dipende da cosa ci ho scritto dentro. Oppure un flip-flop-> memorizzazione di un bit per volta.

BEHAVIOR OF OPERATIONS rispetto timing signals:

In ogni istante il sistema può eseguire k operazioni contemporaneamente (con k>=1, il non far niente è comunque uno status, per questo non è k>=0). Le operazioni che eseguo in un istante dipendono o dai control inputs di quell'istante o dai control inputs precedenti (è un estensione di sequential/combinational applicato alla control unit). Se Control Unit è sequenziale e data path è combinatoria ho finite state machine (FMS), viceversa ho dei RT level basic blocks.

ARCHITECTURAL MODEL: chiamata anche Flynn-taxonomy, l'idea è di avere un flusso di istruzioni e un insieme di dati e posso gestirli insieme (le schede video sono normalmente SIMD perché devo applicare la stessa istruzione a tutti i pixels, ogni elaboratore si occupa di un pixel), MIMD sono sostanzialmente i multi-cores. Le MIMD si distinguono fra multi-computer (ogni computer ha memoria e I/O) e multi-processor(parte della memoria è condivisa, in genere a partire dalla cache di secondo livello).



FONTE: http://www.testgroup.polito.it

FUNCTIONALITY: 5 categorie di processore.

- **General purpouse**: processore che può eseguire qualsiasi tipo di programma (es. PC).
- **Special purpouse**: processori progettati e ottimizzati per una specifica applicazione, es. processori per i calcoli in floating-point, oppure i co-processori per trattare in modo efficiente le matrici.
- **DSP** (Digital Signal Processor): pensati e costruiti per svolgere particolari categorie di operazioni da applicare ai segnali (filtri, convoluzione, etc.) qualunque sia la natura dei segnali.
- **GPU** (Graphics processing units): processori pensati solo per accelerazioni grafiche (e.g. NVIDIA).
- I GPU sono molto potenti, potrei anche utilizzarli non solo per il video-> **GPGPU**.

IMPLEMENTATION: come il sistema è costruito.

- **System-on-rack**: il sistema è all'interno di un contenitore (e.g. server).
- **System on board**: l'intero sistema è su di una piastra.
- **System-on-chip**.
- **System-on-programmable-devices**:
  ASIC=circuito integrato progettato per risolvere uno specifico problema.
  Oppure ***FPGA***= circuito integrato progettato e costruito per essere configurato dall'utente finale magari anche sul campo. Costruisco un general purpose e poi l'utente finale mette insieme i blocchetti che ha all'interno per ottenere quel che vuole.
  Parentesi: se vendo i controllori per i cancelli automatici, come posso pensare di implementare fisicamente il mio controllore? Da un punto di vista teorico decido di costruirmi un IC, un ASIC che faccia tutto (costi immani, non è modificabile) oppure posso farmi un PCB un system-on-board (posso caricare facilmente un altro programma, però è più facile da copiare rispetto un ASIC). Queste sono le due alternative, negli ultimi anni un'altra soluzione è utilizzare un FPGA, quindi non investo nel PCB ma ho un solo chip costruito come FPGA (però sono più lenti di un ASIC perché non sono stati ottimizzati per quel specifico compito).Il passo ulteriore è che le FPGA sono così grandi che posso implementare un CPU, memoria e I/O per cui su di un unico chip posso mettere l'intero sistema senza perdermi la riprogrammabilità e la flessibilità del PCB.

Lezione 07-03-2014

PHYSICAL DIMENSIONS

- **Supercomputers** or High Performance Computing (**HPC**), le prestazioni si misurano in flop/sec (nro. Operazione in virgola mobile al secondo). Stiamo parlando di peta flop (10^15), dare un'occhiata a http://www.top500.org per classifica dei 500 calcolatori più potenti. L'anno scorso c'era al primo posto il Tianhe-2 (sviluppato in Cina, testato con dei benchmark standard, 33.86 PFlop/s, composto da 3 milioni di core, consuma 17.8MW (ca 5000 famiglie)). Al secondo posto c'è il TitanCray che raggiunge i 18PFlop/sec ma consuma "solo" 8MW. Ok, ma a cosa servono? Per tutte quelle applicazioni in cui servono un mare di calcoli: applicazioni militari, tutta la parte meteo, di simulazione finanziaria, analisi e simulazione per il rilevamento petrolifero.
- **Server**.
- **Desktop**: PC da tavolo e laptop.
- **Mobile systems** (smartphone e tablet).
- **Embedded systems**: sono immersi in altri sistemi e non sono percepiti come sistema di elaborazione dall'end-user.

Altri due concetti:

- **GRID**: infrastruttura di calcolo ottenuta collegando insieme diversi calcolatori. Non è una semplice rete locale, non serve solo per comunicare ma per condividere le risorse di calcolo.
- **CLOUD**: per esempio il problema della RAI che deve trasmettere in streaming delle partite di calcio, se devo riuscire a erogare un certo servizio anche in situazione di picco dovrei avere infrastrutture particolari solo per soddisfare quel picco di richieste (e.g. provider telefonico che deve smistare gli SMS a capodanno)-> se sovradimensiono i miei calcolatori per questo picco, poi molti di essi non saranno utilizzati. L'idea non è di comprare n macchine in più ma di affittarle, per cui quando nei hai bisogno io di affitto tutta la capacità di calcolo che vuoi (nuovo modello di business).in realtà non c'è solo l'hardware come servizio (HAAS: Hardware As A Service) ma anche il PAAS (Program as a service). -> quando mi serve affitto l' HW o il SW che mi serve.

Nota: Tutte le macchine hanno lo stesso modello di costituzione e di funzionamento, tutti funzionano alla stessa maniera ma la percezione che ha l'utente è molto diversa. Si rischia di avere sull'hard disk che compro un processore che è più potente di quello che ho sul PC.

DIFFERENT POINT OF VIEW

Diverse figure e come ciascuno si approccia a un sistema di elaborazione.

- Progettista di sistemi: approccio top-down.
- Hardware designer: interconnessione di componenti.
- Assembler level programmer: ha a disposizione solo l'insieme di istruzioni che la CPU capisce direttamente (Instruction Set Architecture-> ISA). Deve pensare come la macchina.
- High level programmer: vede il sistema come una virtual machine, la macchina riesce a capire istruzioni coerenti al linguaggio scelto e al paradigma scelto.
- Computer user: lo vede o come un eragatore di servizi (service provider) o come un'interprete di comandi.
- End user: è quello per i sistemi embedded.

ABSTRACT DATA TYPES (ADTs)

Cos'è un programma? È un insieme di algoritmi che operano su dei dati. Algoritmo deriva da il nome di un matematico arabo.
Algoritmo: sequenza finita di operazioni da compiere in un tempo finito.
Bontà dell'algoritmo (in termini di prestazioni): numero di operazioni (il tempo di esecuzione non è indipendente dalla macchina su cui la faccio girare).
Data type: il tipo di dato associato al valore. A ogni tipo sarà associato un range di valori rappresentabili.
Basic data types: tipi base nativi del linguaggio di programmazione scelto (int, float, char,boolean).
Data Structure - struttura dati: modo per memorizzare e organizzare in modo che questi dati siano utilizzabili in modo efficiente (e.g. array, matrix, records, ordered list, dictionary, trees).

In termini di programmazione, posso costruirmi nuovi tipi di dato (es. per trattare i numeri complessi).

ADTs: modello formale che definisce una struttura dati e degli operatori che possono essere applicati a quei dati. Per esempio gli interi e le 4 operazioni sono definite come ADTs. Le operazioni devono poter creare, manipolare e cancellare il tipo di dato.

QUEUE

Coda: insieme di elementi in cui tutti gli inserimenti avvengono da un'estremità (questo side si chiama rear) e tutte le estrazioni avvengono dalla parte opposta (questo side si chiama front). Gli elementi possono essere qualunque cosa (persone, bit, etc.). La coda è una struttura di tipo **FIFO** (First In First Out). Quali sono le operazioni che posso fare su una coda? Estrarre e inserire un nuovo elemento, in una coda non ha senso cercare un elemento perché non posso accedere all'i-esimo elemento, creazione e cancellazione della coda. Quando ragiono con le ADTs non m'interessa come le implemento o in quale linguaggio lo faccio, siamo su un livello più astratto. Altre operazioni: svuotamento (lasciare la coda allocata ma eliminare tutti gli elementi dentro). La queue per definizione non è reversibile (non posso scambiare rear e front). Oppure dimmi chi è il primo elemento della coda ma non estrarlo.

- *Create(Q). Ritorna ok se creazione corretta, errore altrimenti.*
- *Delete(Q) ritorna Ok o error (se non riesce a cancellarla, il perché non importa).*
- *MakeEmpty(Q): svuoto la coda senza dis-allocarla.*
- *Front(Q): leggo il primo elemento senza estrarlo.*
- *Dequeue(Q) estraggo elemento.*
- *Enqueue (x, Q): aggiungo elemento x alla coda.*
- *isEmpty(Q): indipendentemente dalla dimensione massima che dipende dalla specifica implementazione, mi dice se la coda è vuota o no.*
- *isFull(Q).*

STACK

E' una **LIFO** (Last in First out), come una pila di piatti. **Tutte le operazioni (estrazione e aggiunta) avvengono dalla stessa estremità**. Su di una pila in gergo:

- *push(x, S): aggiungo l'elemento x alla cima dello stack.*
- *pop(S): tolgo l'ultimo elemento (quello in cima).*
- *Altre…*

ADTs e LINGUAGGI DI PROGRAMMAZIONE

Per il Pascal ci sono i record, in C le struct, in OOP ho le classi. Le ADTs devono essere indipendenti dal linguaggio usato e dai dettagli implementativi. La definizione di una ADTs è funzionale ed è modulare: posso cambiare quello che c'è dietro ma il front-end riamane lo stesso, stacco in modo chiari i dati dal modo in cui sono messi.

An introduction to Object-Oriented Programming (OOP)

I programmi software sono fra le cose più complesse che l'uomo abbia mai fatto, più il problema è complesso più ho errori (errori che poi si ripercuotono sia su safety sia sui costi). Come posso controllare questa complessità? Negli ultimi 50 anni ci sono stati due soluzioni per minimizzare questa complessità: OOP e UML (Unified Modeling Language, è un linguaggio di modellazione che si basa sulle specifiche richieste, quando faccio il progetto devo standardizzare quello che faccio). Perché è necessaria la OOP? Nasce per superare la programmazione procedurale (es. pascal o C) dove ho una lista di istruzione che il calcolatore prende una per una e poi il calcolatore le esegue. Questo va bene per piccoli programmi ma le funzioni che io scrivo hanno restrizioni per accedere alle variabili globali del programma, e **non c'è una stretta correlazione fra funzione e dati su cui essa opera**. Se invece le funzioni possono accedere senza limitazioni e indipendentemente alle variabili (se devo cambiare il tipo di dato devo poi andarlo a cambiare in tutto il codice). L'altro problema è che **non c'è una relazione semantica fra funzione e dati**, se separo dati e strutture questo non matcha con il mondo che ci circonda.

Anziché pensare in termini di funzioni come singoli blocchetti, proviamo a pensare in termini di oggetti (come normalmente sono abituata a pensare). L'OOP nasce per modellare il mondo naturale così come lo vediamo noi. La OOP non è un linguaggio di programmazione ma un paradigma, il linguaggio di programmazione è al di sotto del paradigma. Simula e Smalltalk sono i primi OOP (anni 60-70). **L'OOP mette insieme in un'unica entità dati e funzioni che operano su questi dati-> OGGETTO**. In C non sono obbligata a fare le struct (se le faccio lo faccio per rendere più pulito il codice), mentre qui se non definisco le classi non vado da nessuna parte. Quindi ho un insieme di oggetti che cooperano fra di loro scambiandosi messaggi, come funziona il programma è determinato da come gli oggetti interagiscono fra di loro. Gli oggetti sono considerati smart (è self-consistent) ed come black-box (nasconde informazioni non utili a chi lo utilizza).

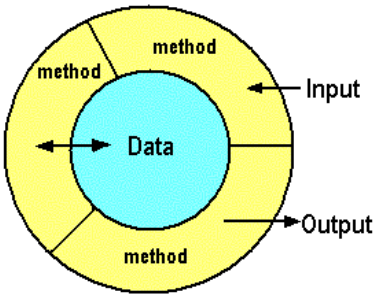
Ogni oggetto ha :

- **un suo stato: insieme di proprietà che caratterizza l'oggetto -> set di data (variabili).**
- **un suo comportamento: cosa fa l'oggetto sotto certi stimoli -> set di metodi.**

Nota ai lettori: spesso uso come sinonimo di stato di un oggetto (cioè il suo set di variabili) il termine "property".

ESEMPIO: tetris. Quali sono gli oggetti, cosa possono fare gli oggetti, quali proprietà hanno gli oggetti.

- Mattoncino: ruotare e cadere, colore, forma, dimensione.
- Tavolo di gioco: colore, dimensione,
- Preview del mattoncino successivo: property (mattoncino),
- Segnapunti: calcolare quante volte si fa riga, fa la statistica, property: label con level, score, lines,



ENCAPSULATION

Nascondere le variabili non fondamentali dall'esterno. Il problema è che se cambio lo stato di una variabile in modo involuto può generare un casino di problemi -> lo stato di

un oggetto deve essere preservato e non tutti lo possono cambiare. Ci sono delle funzioni chiamate **member function** il cui unico obiettivo è leggere-scrivere le property dell'oggetto. Quindi se voglio modificare lo stato dell'oggetto ho un unico punto d'ingresso da cui entrare e dall'altra parte ci sarà tutto il codice di controllo. *Data encapsultion* è sinonimo di data hiding. Per accedere alle property di un oggetto devo per forza utilizzare i metodi messi a disposizione, non posso accedere alle property direttamente.

Un oggetto può richiamare funzioni di altri oggetti, **i messaggi sono i parametri che passo in input alla funzione richiamata del II oggetto**.

CLASSI E ISTANZE

Il problema è che non posso fare un oggetto per ogni macchina o persona su questa terra. A noi non interessa descrivere il singolo oggetto se posso trovare proprietà comuni a due o più oggetti. Tutte le car hanno delle proprietà in comune.
Classe: astrazione di oggetti che hanno in comune proprietà e funzioni.
Quindi descrivo solo una volta proprietà e metodi e poi il singolo oggetto avrà definite queste proprietà e utilizzerà questi metodi.

- ➔ *La classe è un tipo di dato astratto. Definire una classe non significa istanziare un oggetto, quando un oggetto appartiene a una classe allora quell'oggetto è l'istanza di quella classe.*

Tutti gli oggetti di una stessa classe hanno le stesse property ma i valori delle property che essi assumono sono diverse.

Esempio in C++:

```
class CRectangle {
      intx,y;
      public:
      void set_values(int, int);          // funzione membro
      int area(void);                     // esplicita un servizio

}
CRectangle rect1;
rect1.set_values(3,5);   //<nome oggetto>.<funzione a cui voglio accedere>
```