Lezioni del 12/03/14, 13/03/14, 14/03/14.

ALCUNE LINEE GUIDA PER LA PROGRAMMAZIONE

Nei programmi non si scrivono numeri nel codice, non si scrive for(i=1; i<7;i++) ma for(i=1; i<MAX; i++). Questo sia per maggiore leggibilità (per me e per gli altri), per maggiore manuntebilità, perché se da qualche parte non ho usato 7 ma 7-1 sono fregato. Il problema non è fare programmi che funzionino bene, ma devono essere utili (=scalabili, leggibili, facilmente modificabili,…).

Inoltre se una dato viene utilizzato da più funzioni, se devo cambiare questo dato devo cambiarlo in tutti i pezzi di codice che lo richiamano-> potrebbe essere difficile riuscire a non fare casini-> meglio avere una strettissima connessione fra dati e funzioni e non mischiare tutto insieme (meglio più classi piccole che un'unica incasinata).

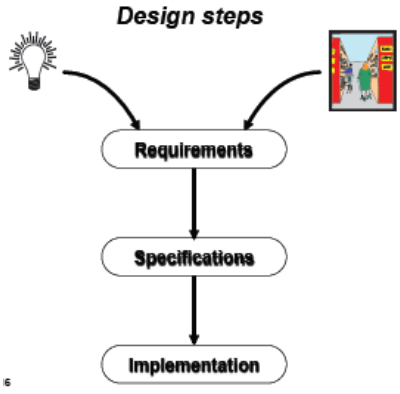
Funzioni: quando creo una funzione e la richiamo, devo assicurarmi di tornare al punto giusto quando ritorno da essa (anche nel caso in cui avessi più funzioni annidiate). Tutto questo funziona grazie al fatto che le CPU-microprocessori hanno tutti le **call** e i **return**. Sono due istruzioni macchina che permettono di usare correttamente le funzioni utilizzando uno stack. Inoltre come faccio praticamente a passare i parametri? Dipende dal linguaggio di programmazione. Per esempio nel C posso fare il passaggio by value o by reference.

Progettazione: the design cycle

Design steps

Designing=progettare=l'atto di creare un nuovo oggetto artificiale che prima non c'era partendo da concetti astratti derivanti dalla mia esperienza sulle cose esistenti.

Design process: l'insieme di operazioni che hanno l'obiettivo di portare al prodotto della progettazione.



(fonte: Test Group Polito)

Posso partire o dall'idea di un nuovo prodotto (nelle aziende più grande è lo strategic marketing che pensa a quali nuovi prodotti produrre) oppure dai bisogni dei clienti (soprattutto se l'azienda è più orientata al singolo cliente). Quindi partendo da uno dei due attraverso il design process ottengo l'implementation (= una possibile soluzione al problema target). Il design process è in genere abbastanza lungo. I tre output intermedi principali nell'ordine sono:

- **Requirements**
  In genere sono dei documenti informali (non nel senso che sono a mano, ma che sono scritti senza ricorrere a modelli-linguaggi formali). È in un linguaggio parlato e quindi è in parte ambiguo (nel senso che non è biunivoco).
- **Specification**
  Secondo la definizione dell'ISO 8402 (standard internazionale che racchiude per lo più definizioni, è quello che sta dietro l'ISO 9000). L'ISO 9000 è uno standard internazionale che ha

l'obiettivo di certificare la qualità del processo produttivo (e quindi non nel prodotto). Se invece voglio certificare la qualità del prodotto non uso la ISO 9000.
def. Specification:document stating the set of requirements a product must satisfy.
Inoltre non ci sono solo le product spec, ma anche le process spec., i marketing spec, le re cycling spec, etc. Devono essere il più omni-comprensive possibile. Per esempio l'Intel per un bridge ASIC (fra due bus) ha una spec di 400 pagine. Inoltre come faccio ad essere sicuro che nelle spec non ci siano contraddizioni o abbi saltato delle cose? Le spec sono più dettagliate e più formali dei reqs, inoltre il passaggio dai reqs alla specs non viene fatto di botta, ma viene fatto aggiungendo volta dopo volta nuovi dettagli. È più che altro un processo continuo. Inoltre quando di fermo? Non c'è una risposta, mi fermo quando raggiungo un buon livello di confidenza, ovvero quando il gruppo di progettazione arriva a dire che è pronto. Una volta le specifiche, entra in gioco il progettista.

- **Sintesi**
  Da specification a implementation arrivo attraverso la sintesi. Gli obiettivi della sintesi sono:
  - Soddisfare tutti i vincoli (inclusi i costi) imposti.
  - Deve esprimere tutto in termini di buldings blocks conformi alla mia tecnologia target.
  - Non deve violare nessuna regola di progetto (e.g. seguire le metodologie).
  - Produrre un' implementation coerente con le spec.

In questo corso la specifica è il testo d'esame. La sintesi può essere vista come una ricerca nello spazio di progetto (però a volte è difficile riuscire a tenere sotto mano tutti i vari aspetti). La sintesi consiste nel trovare il migliore compromesso (trade off) fra esigenze fra loro in conflitto (costo vs. performance, velocità vs area occupata dal silicio, etc). Conviene definire un costo globale:

$$costo=k_1x_1+k_2x_2+....$$

Dove k_i è il peso che assume quel fattore nell'insieme globale e x_i indica un generico impact-factors (e.g. area occupata, ritardo, affidabilità, etc).

Inoltre la fase di sintesi può essere vista come un sottoinsieme di fasi, parto della spec e arrivo in prima approssimazione a un' implementazione, questa implementazione diventa a sua volta la spec per una seconda implementazione e continuo a iterare. Alla fine ottengo quello che vorrei raffinando man mano le varie cose. Mi fermo nella decomposizione quando uso solo blocchetti elementari per la specifica tecnologia scelta.

Validation&verification (V & V)

Validation: (in italiano convalida) il processo che mira a capire se il sistema alla fine è conforme alle mie aspettative iniziali. Verificare la correttezza fra il semi-product e le aspettative iniziali (indipendentemente dal punto in cui sono) (definizione tratta dall' IEEE standard glossary of software engineering terminology).

Alcuni esempi: come faccio a fare la validation di un generico passo? Se la spec è ancora informale devo prendere un gruppo di esperti esterni al progetto e controllino che la spec sia coerente con l'implementation ottenuta a quel punto (visual inspection, molto usata in software), se invece la spec è formale posso fare delle simulazioni.

Il centro di research and development di una delle più grosse compagnie europee ha comunicato che nel 95% dei casi in cui in cui si è dovuto modificare il progetto la causa è dovuta a errori presenti nelle specifiche. Questo vuol dire che il processo di sintesi è ben consolidato, sono le spec a essere più critiche.

Il prodotto finale viene verificato attraverso simulazione (avere un programma che simula il comportamento del prodotto finale), emulazione (il costruire una versione che ha caratteristiche simili a quelle finali, es. un prototipo su FPGA) e la visual inspection.

Verification: (in italiano verifica) cerca di verificare se il passo i-esimo è coerente con il passo (i-1)-esimo. Se la verification fosse sempre esatta e coerente alla fine ottengo qualcosa di coerente con l'inizio, ma avendo a che fare con sistemi complessi non ho la garanzia di non perdermi nulla.

Verification: abbiamo fatto un passo giusto? Approcci più comuni:

- La simulazione: però ha dei limiti, dipende dalla raffinatezza degli strumenti che uso, ho descritto il sistema con un modello che potrebbe essere sbagliato o incompleto, ma soprattutto il problema è che simulo con un certo set di input e in output ho dei valori che possono essere coerenti con quello che vorrei oppure no, il problema è che è fisicamente impossibile riuscire a testare tutti i possibili set di valori. Il vero problema è che non riesco a applicare tutti i possibili input. Posso solo concludere che il circuito funziona con quel range di input. Sta all'esperienza del progettista scegliere con quali input simulare il mio sistema.
- Verifica formale: per circuiti non troppo grandi riesco a garantire che la specifica sia coerente con l'implementazione? Voglio dimostrare in modo matematico che specifica 1=specifica 2. Il vantaggio è che sono sicura al 100%, però va bene solo per cose piccole.
- Model Checking: in modo formale descrivo ilmio circuito e poi descrivo alcune proprietà che il circuito deve avere, poi chiedo a un tool di verificare matematicamente che queste proprietà siano sempre vere o false indipendentemente dall'ingresso e dallo stato del sistema.
  (la V&V si occupa solo della validità del progetto, è diversa dalla fase di collaudo). Però non ho nessuna garanzia di non dimenticarmi nessuna proprietà.
- Functional verification: tecnica abbastanza recente, ci sono dei linguaggi standard per descrivere in modo semplice l'andamento dei segnali. Mi aiuta ad automatizzare la simulazione.

Tutti questi approcci non sono esclusivi ma sono complementari. Oggi la parte di verifica costa fino al 70% del costo complessivo di progettazione di un ASIC moderno.

Design Rule Checking (DRC)

Una tecnologia a 28nm ha più di 2000 regole di design for manufacturing (manufattibilità del mio silicio). Come verificare che non ho violato nessuna regola?

- Layout rule checking.
- Logic rule checking (es. fan-out, massimo numero di porte che posso pilotare con un unico segnale, ampiezza minima dell'impulso, la pendenza del clock, etc.).
- High level rule checking.

Per verificare se è tutto corretto posso usare la simulazione o con dei tools ad hoc (in genere comunque chi definisce le regole mi dà anche gli strumenti per verificare se sono corrette).

Programmazione: puntatori

Definizione

Puntatore=variabile contenente un indirizzo di memoria. Se ho una variabile puntatore e gli inserisco un numero, questo numero non viene interpretato come int o come float ma come indirizzo, per cui riesco a puntare a una locazione di memoria il cui indirizzo è il valore contenuto nel puntatore.

Per definire un puntatore la sintassi è

```
int *p;      // sto definendo un puntatore che punterà a una variabile di tipo int
```

Devo esprimere il tipo della locazione di memoria a cui vado a puntare perché il compilatore deve sapere quanta memoriaallocare per quel tipo. Se è un int sarà un 32 bit, se è un char sarà un 8 bit, etc. tutti i puntatori hanno la stessa dimensione perché gli indirizzi di memoria sono codificati su n bit dove n dipende dal processore che uso, indipendentemente da cosa vado a puntare. Ciò che cambia è la locazione di memoria sulla quale vado a puntare. Perché devo specificare il tipo del puntatore? Perché voglio sapere cosa andrò a vedere quando aprirò la scatola a cui il puntatore punta (deferenzazione del puntatore, il compilatore deve sapere su cosa andrà a lavorare).

Inizializzazione

Il puntatore può essere inizializzato a

- 0 or NULL-> ho allocatore una variabile puntatore (32 bit) ma non ho ancora specificato l'indirizzo.
- Address

Operatori

- **&**: operatore di reference, torna l'indirizzo dell'operando.
  `int a=10;  int *p=NULL;  p=&a; // metto in p l'indirizzo della variabile a, a è di tipo intero, p punterà a qualcosa di intero, ok.`
  Se faccio &p ottengo l'indirizzo del puntatore, cioè dove la variabile puntatore è messa.
- ***:operazione deferenziale, serve a tornare i valore al quale il puntatore sta puntando. Quindi *p tornerà 10 (cioè il valore assunto dalla variabile a alla quale il puntatore punta). Vuol dire prendere il valore del cassetto al quale il puntatore punta. Lo uso sempre quando definisco il puntatore e poi quando devo deferenziale il puntatore (ha due usi diversi).

Esempi

```
int x=1;
int y=2;
int *ip;     // puntatore che punterà a qualcosa di intero. Se non specifico il valore alla sua inizializzazione, alcuni compilatori gli danno un valore sporco o NULL.

ip=&x; // ip ora punta a x
y=*ip; // y ora viene sovrascritta e diventa 1.
*ip=0; // x ora è 0
```

Non usare mai * e & insieme uno dopo l'altro, il loro effetto si cancella.
&*p=&(*p) -> *p torna il contenuto della variabile a, quindi è come scrivere &a, quindi è come tornare

l'indirizzo di a.
se invece faccio *&p=*(&p)=torno indirizzo di a.

Modi di referenziale una variabile:
In modo diretto-> non faccio dei salti e non uso il puntatore, in modo indiretto allora uso il puntatore.
Referenziare è sinonimo di puntare, accedere al valore di quella variabile.

Aritmetica dei puntatori

Se faccio +1 su di un puntatore, mi muovo in avanti di n bit dove n=size(tipo a cui sto puntando, quello che ho usato per definire il puntatore).

```
int *p=&something;

p=p+5; // sto puntando a 5 interi (=32*5 bit ) sotto/dopo.
// l'attuale incremento dipende dal size della variabile a cui punto.
```

Con il puntatore non c'è allocazione statica di memoria, ma dinamica.

Programmazione: algoritmi di ordinamento

Visto che tutti i programmi si basano sui dati, spesso è richiesto di ordinarli per fare delle ricerche in modo più efficiente.

DEFINIZIONE

È un algoritmo che ha l'obiettivo di disporre, ordinare gli elementi di un insieme secondo una sequenza stabilita sulla base di una relazione di ordine (e.g. ordinamento crescente o decrescente).

Se voglio ordinare in modo crescente gli elementi interi di un vettore di n valori, avrò:
input: $<a_1,a_2,a_3....a_n>$
output: permutazione $<a_1', a_2'...>$ (mantengo tutti gli elementi, non ne tolgo né aggiungo nessuno) in modo modo che $a_1'<=a_2'<=... a_n'$.

Ipotesi:
- non si deve occupare spazio aggiuntivo o quanto meno devo minimizzarlo.
- per accedere al generico elemento del vettore impiego O(1)-> questo tempo non dipende dal numero di elementi del vettore.
- gli elementi del vettore possono essere strutture e vengono ordinati in base a un loro campo chiamato chiave.

La complessità degli algoritmi sarà valutata in **numero di confronti e numeri di scambi**.

Stabilità: un algoritmo di ordinamento si definisce stabile se conserva l'ordine originale degli elementi aventi ugual chiave. Cioè se faccio due ordinamenti in serie, alla fine del secondo non mi perdo il primo a parità di chiave del primo ordinamento.

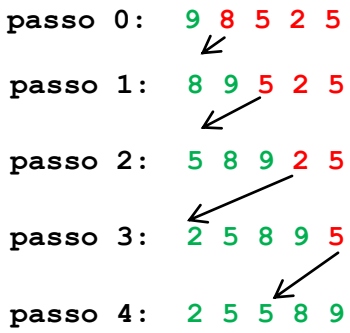
In place: algoritmo che non usa copie del vettore originale.

ALGORITMI ITERATIVI

Insertion sort

Il vettore viene diviso in due parti-> una ordinata e una non. Prendo il primo elemento della parte non ordinata e la metto al posto giusto nella parte ordinata.
Al passo 0 ho sempre parte ordinata=primo elemento, parte non ordinata=tutto il resto.

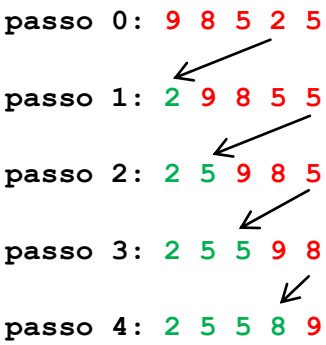
Al passo i-esimo prendo i-esimo elemento della parte non ordinata e lo confronto con tutti gli elementi della parte ordinata, quando trovo la parte giusta lo inserisco in quella posizione shiftando tutti gli elementi a destra di uno.



Questo è quello più semplice.
Worst case:quando l'i-esimo elemento è il più piccolo di quelli con cui lo confronto -> $O(n^2/2)$.
Best case: vettore già ordinato, solo confronti, no scambi ->O(n-1).

Selection sort

Ho sempre il vettore diviso in due parti, a ogni passo prendo l'elemento più piccolo della parte non ordinata e lo metto in testa della parte ordinata.



Il numero di confronti non dipende dal contenuto del vettore ed è sempre pari a $O(n^2/2)$. Il numero di scambi è O(n), è da preferirsi quando l'operazione di scambio è particolarmente costosa rispetto a quella di confronto.

Bubble sort

A ogni passo considero il sotto vettore (0, n-1-i) e considero le coppie adiacenti ed eventualmente le scambio. Man mano come una bolla faccio salire in alto il valore più alto.

Questa volta considero come ordinata la parte in cima perché so che ci sono i valori più alti.

Worst case: il vettore è ordinato in modo opposto a quello richiesto ($O(n^2)$).