NUMERO: 1797A - ANNO: 2015

A P P U N T I

STUDENTE: Massara Andrea

MATERIA: Algoritmi e calcolatori, Riassunto - prof. Prinetto

SYSTEMS: DEFINITION AND TAXONOMIES

- Sistema: entità interagente con altre entità e con il modo circostante
- System Boundary: frontiera comune tra sistema e il suo ambiente

Proprietà del sistema:

- Funzionalità
- Performance
- Affidabilità e sicurezza → • Consente l'affidabilità al sistema rendendolo degno di fiducia
  • Libero da pericolo
- Costi

INTRODUCTION TO ADTs

La programmazione è un processo creativo che consiste nel dare istruzioni ad un computer. Il programma è l'insieme delle istruzioni. Esistono vari paradigmi di programmazione, tra cui:

- Procedurale: Algoritmi + dati
  - TIPI DI DATO: Formati standard che possono contenere determinati tipi di valori o dati
  - STRUTTURE DATO: Modo particolare di organizzare e memorizzare dati in modo efficiente

→ ADT: Abstract data types

Modello formale che definisce:

- struttura dati
- set di operatori che ad essa possono essere applicati (creazione, manipolazione, ...)

Esempi di ADT:

- Code (queues): inserimenti di nuovi elementi avvengono dal fondo e le estrazioni dall'inizio
  → FIFO! → OPERAZIONI: · create · front · is_empty · delete · dequeue · is_Full · make_empty · enqueue
- Pile (stacks): inserimento ed estrazione avvengono dalla cima (top)
  → LIFO → OPERAZIONI: · create · top · is_empty · delete · pop · is_full · make_empty · push

ALGORITMI DI ORDINAMENTO

Algoritmi mirranti a disporre gli elementi secondo una sequenza stabilita da una relazione d'ordine in modo che ogni elemento sia $>$ / $<$ di quello che lo segue

Problema: vettore di n elementi in serie crescente
Input: " " " "
Output: " " " " ordinato

Ipotesi:
- non occupare spazio aggiuntivo
- si può accedere ad un generico elemento del vettore in un tempo $O(1)$
- gli elementi del vettore possono avere strutture e vengo ordinate secondo una CHIAVE

La complessità dell'algoritmo è valutata in funzione di due parametri:

- # confronti
- # scambi

NON ESISTE un algoritmo di ordinamento con complessità minore di $O(n \log n)$
Parliamo di STABILITÀ se un algoritmo conserva l'ordine originale degli elementi con uguale chiave.

Parliamo di IN PLACE se l'algoritmo non crea copie del vettore per ordinarlo

Dividiamo gli algoritmi in:

- ITERATIVI
- RICORSIVI

ALGORITMI ITERATIVI

① • Insertion sort:
1) Divide il vettore input in 2 parti: ordinata / non ordinata
2) Prendo il I elemento della parte non ordinata e lo colloco nel posto giusto
3) Inizialmente la parte ordinata contiene solo il I elemento

All'i-esimo passo si inserisce l'i-esimo elemento nel vettore nella posizione corretta tra gli $(i-1)$ elementi già ordinati.
Per individuare la "posizione giusta" in cui inserire l'elemento $x[i]$ si opera così:

- si salva $x[i]$ in una variabile temporanea t
- si confronta t con gli $(i-1)$ elementi ordinati
- si spostano tutti gli elementi di una posizione a dx

Complessità: il caso peggiore si ha quando l'i-esimo elemento t è il più piccolo tra quelli già ordinati. In tal caso il # dei confronti è $O(\frac{n^2}{2})$
Il migliore è quando il vettore è già ordinato. # confronti $(n-1)$

② • Selection sort:
1) Divide il vettore input in 2 parti: ordinata / non ordinata
2) Si prenda il minimo della parte non ordinata e lo si scambia con il primo della parte non ordinata

Complessità: - il # di confronti non dipende dal contenuto del vettore ed è pari a $O(\frac{n^2}{2})$
- il # di scambi è $(n-1)$

Conviene quando il costo dello scambio è maggiore di quello del confronto

La scelta del pivot influenza fortemente la convenienza del metodo:

- elemento di mezzo: scelta migliore per vettore ordinato, scelta media per vettore non ordinato
- elemento casuale: migliorano nel caso di vettore mediatamente ordinato. Si ha un significativo miglioramento se il vettore è completamente disordinato
- 1° o ultimo elemento: partizioni buone, ma degenerano in caso di vettore già ordinato

Complessità: la complessità dipende dal pivot ma se scegliamo l'elemento in mezzo:

$$T(n) = n + 2T\left(\frac{n}{2}\right) \quad \text{con } n \geq 1$$

quindi

$$T(n) = O(n \log n)$$

ALGORITMI A CONFRONTO

	# cfr (peggiore)	# cfr (medio)	# cfr (meglio)	# scambi (peggiore)	# scambi (medio)	# scambi (migliore)
INSERTION	$n^2/2$	$n^2/4$	n	$n^2/2$	$n^2/8$	$\varnothing$
SELECTION	$n^2/2$	$n^2/2$	$n^2/2$	n	n	$\varnothing$
BUBBLE	$n^2/2$	$n^2/2$	n	$n^2/2$	$n^2/2$	$\varnothing$
MERGE	$n \log n$	$n \log n$	$n \log n$	$n \log n$	$n \log n$	$\varnothing$
SORT	n^2	$n \log n$	$n \log n$	$n \log n$	$n \log n$	$\varnothing$

LISTE

= Sequenza di zero o più elementi dello stesso tipo

$n=0$ → Lista vuota

a_1 (= primo elemento) → testa, "head"

a_n (= ultimo elemento) → coda, "tail"

eol(L) → end of list, elemento dopo l'ultimo elemento

Operazioni nelle liste:

- Insert(x, p, L)
- Delete(p, L)
- Locate(x, p, L) (ritorna la prima occorrenza x dopo la posizione p.)
- Retrieve(p, L)
- Next/Previous(p, L)
- Makenull(L)
- First(L)

③ Implementazione tramite indici:

Utile per linguaggi privi di puntatori. Con questo metodo emulo il loro comportamento attraverso record che contengono la posizione nel vettore dell'elemento successivo

Occorre gestire la lista delle posizioni libere (free list), utilizzando due variabili aggiuntive: • va per l'indice della cella che contiene il 1° elemento della lista
• va per l'indice della prima cella della free list

LISTE DOPPIE

- Con puntatore doppio: questo serve rendere la lista nei 2 sensi di percorrenza è utile dare due puntatori da ogni elemento. Uno per il precedente e uno per il successivo

  Ergo: è necessario avere due puntatori esterni, uno header e uno tail

- Circolari: liste in cui l'ultimo elemento punta al primo

Esistono poi le liste multiple in cui ogni elemento contiene più puntatori

ALBERI (TREES)

Particolare tipo di ADT utilizzato per memorizzare un insieme di elementi tra i quali sia possibile stabilire una relazione gerarchica

Formalmente è una coppia ordinata (V,E) (ordinata) di insiemi, dove V è finito e non vuoto e contiene oggetti (nodi) tra i quali esiste una certa parentela, mentre E è l'insieme degli archi che collega i nodi. I nodi che non hanno figli sono detti FOGLIE

(Il primo è detto "RADICE")

→ Si dice cammino l'insieme dei nodi legati dalla relazione padre-figlio che collega due generici nodi! In tal caso, se esiste un cammino tra A e B allora A è ascendente di B come B è discendente di A.

In teoria esiste l'unicità del cammino tra A e B se uno esiste! Più cammini = grafi!

- Profondità: lunghezza dell'unico cammino esistente fra radice e nodo
- Altezza: lunghezza del massimo cammino da quel nodo ad una foglia. Altezza di un albero è l'altezza della radice

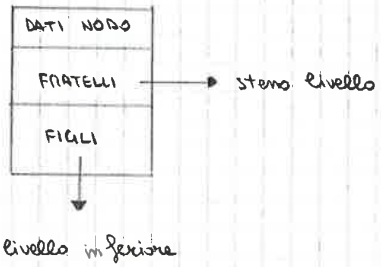
Se tutte le foglie hanno stessa profondità e l'altezza dell'albero è H allora vale:

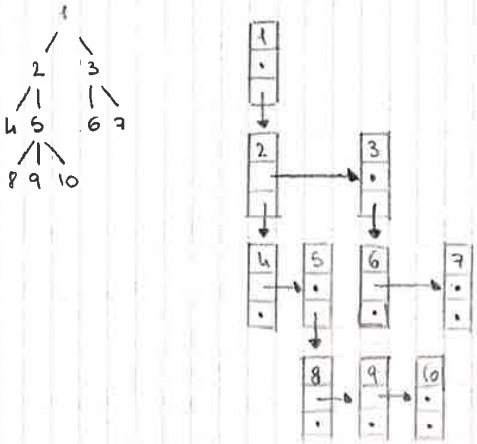
$$p(i) + h(i) = H$$

- Grado: # di discendenti diretti di quel nodo. Il grado dell'albero è il max grado dei suoi nodi
- Ordine: Solitamente i figli si radunano da sx a dx

③ Rappresentazione tramite liste multiple:

Ogni elemento contiene 2 puntatori: uno al figlio più a sx e uno al fratello più a dx





Vantaggi e svantaggi: - risulta disagevole determinare il padre

ALBERI BINARI

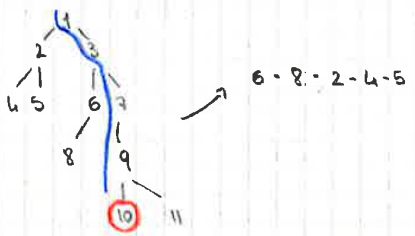
Alberi di grado 2, ovvia tali che ogni nodo può avere:

- ø figli
- un figlio dx
- un figlio sx
- un figlio dx e uno sx

Regola fondamentale: se A e B sono fratelli allora tutti figli di A sono alla sx di tutti i figli di B

Regola pratica: Dato un nodo n per trovare i nodi alla sua sx basta percorrere il cammino da n alla radice; tutti nodi che si dipartono dalla sx (con discendenti eventualmente) sono alla sx di n

Analogamente per la dx

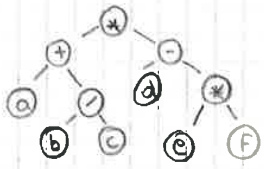


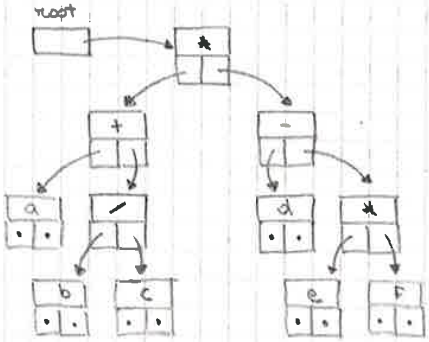
Nota: qualunque albero è trasformabile in un albero binario equivalente e viceversa

- Trasformazione normale → binario: per ogni nodo, si collegano tutti i figli di tale nodo. Poi si cancellano tutti i rami dal nodo in esame verso i figli, fatta eccezione per quello con il figlio + a sx

② Tramite liste multiple:

Ogni record contiene, oltre all'i-esimo nodo, anche i puntatori ai suoi due figli





VISITA DI UN ALBERO BINARIO

Si intende la visita di tutti i nodi dell'albero

① Visita preorder: Prima la radice, poi il sottoalbero di sx, poi quello di dx. In pratica, il padre prima di tutti i figli

Disegno una linea attorno all'albero, cammino in senso orario e scrivo il nodo la prima volta che lo incontro

1 2 4 8 5 9 10 3 6 7 11 12

② Visita inorder: Prima il sottoalbero di sx, poi radice e poi albero di dx. In pratica il padre dopo il figlio sx ma prima del dx

Disegno una linea, percorrerla in senso orario e scrivere il nodo la II volta che lo incontri

8 4 2 9 5 10 1 6 3 11 7 12

③ Visita postorder: Prima sottoalbero di sx, poi quello di dx e poi radice. In pratica il padre dopo tutti i figli

Disegno la linea, la percorro in senso orario e scrivo il nodo l'ultima volta che lo incontro

8 4 9 10 5 2 6 11 12 7 3 1

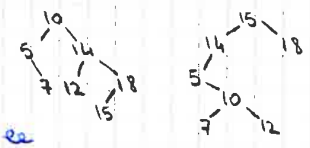
SEARCH TREES

Anche detti BST. Ogni nodo contiene un elemento x dell'insieme in modo che:

- tutti gli elementi memorizzati nel suo sottoalbero sx hanno una chiave < x.key
- tutti gli elementi memorizzati " " " dx " "
  chiave > x.key

→ La visita in order fornisce tutti gli elementi in ordine crescente!

Per un determinato insieme esistono + BST



Le possibili operazioni definibili su BST sono le stesse dei dizionari

I BST facilitano le ricerche:

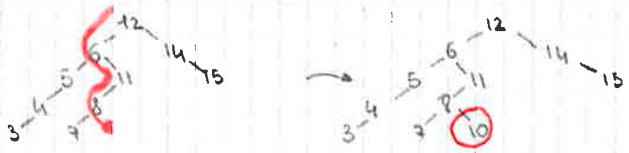
- se x.key < x.key : x è sicuramente a sx di x
- se x.key = x.key : x coincide con x
- se x.key > x.key : x è sicuramente a dx di x

La ricerca di un dato comporta al più un # di confronti pari all'altezza dell'albero

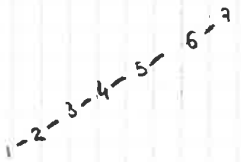
① Inserimento di nuovi elementi:

se la chiave da inserire ancora non esiste si crea un nuovo nodo e lo si aggancia come figlio dell'ultimo nodo visitato dalla ricerca!

E io?



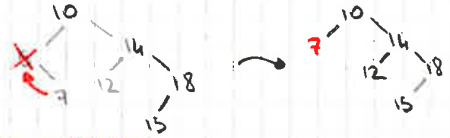
- Inserimento casuale = possibile sbilanciamento dell'albero
  Lo se l'inserimento avviene con ordine (crescente o decrescente) l'albero diventa una lista



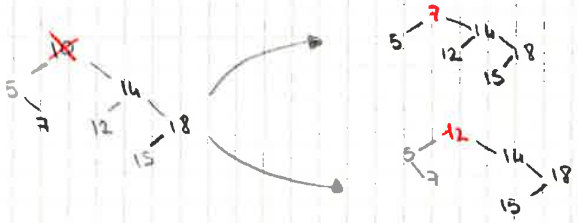
② Cancellazione di un elemento:

- Elemento senza figli : lo cancello e basta

- Elemento con un figlio: elimino l'elemento e lo sostituisco con l'unico figlio



- Elemento con due figli: elimino l'elemento e lo sostituisco con il nodo + a dx del sottoalbero di sx o con quello + a sx del sottoalbero di dx



- Implicazione:

$$a \Rightarrow b = a' + b$$

- Espansione del teorema di Boole:

$\forall f: B^m \to B$ tale che $f(x_1, x_2, \dots, x_m)$ allora

$$f(x_1, x_2, \dots, x_m) = x_1' \cdot f(0, x_2, \dots, x_m) + x_1 \cdot f(1, x_2, \dots, x_m)$$

- Regola di cancellazione:

$x + y = x + z$ con $y = z$ **NO!** Non vale in algebra booleana

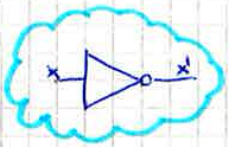
PORTE LOGICHE

① NOT

x', $\overline{x}$ oppure $not(x)$

$0' = 1$
$1' = 0$

x	x'
0	1
1	0



② AND

$x \cdot y$, xy oppure $and(x,y)$

x	y	x·y
0	1	0
0	0	0
1	1	1
1	0	0

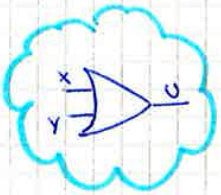
Teoremi:

$x \cdot 0 = 0$
$x \cdot 1 = x$
$x \cdot x = x$
$x \cdot x' = 0$
$x \cdot y = y \cdot x$
$x \cdot y \cdot z = (x \cdot y) \cdot z = x \cdot (y \cdot z)$

③ OR

$x + y$ oppure $or(x,y)$

x	Y	x+Y
0	0	0
0	1	1
1	0	1
1	1	1



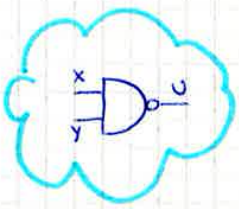
Teoremi:

$x + 0 = x$
$x + 1 = 1$
$x + x = x$
$x + x' = 1$
$x + y = y + x$
$x + y + z = (x + y) + z = x + (y + z)$

④ NAND

$nand(x,y)$

x	y	nand(x,y)
0	0	1
0	1	1
1	0	1
1	1	0



Teoremi:

$nand(x,0) = 1$
$nand(x,1) = x'$
$nand(x,x) = x'$
$nand(x,x') = 1$
$nand(x,y) = nand(y,x)$

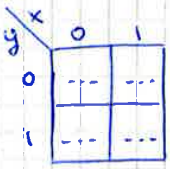
RAPPRESENTAZIONI DI FUNZIONI BOOLEANE

Serve a minimizzare le funzioni con funzioni equivalenti per ridurre i costi

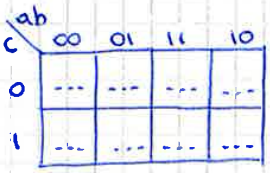
- Tavole della verità:

x	y	f(x,y)
0	0	...
0	1	...
1	0	...
1	1	...

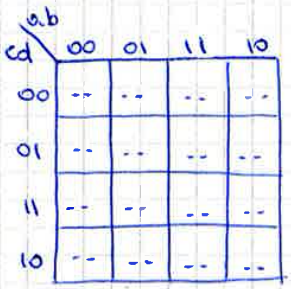
- Mappe di Karnaugh



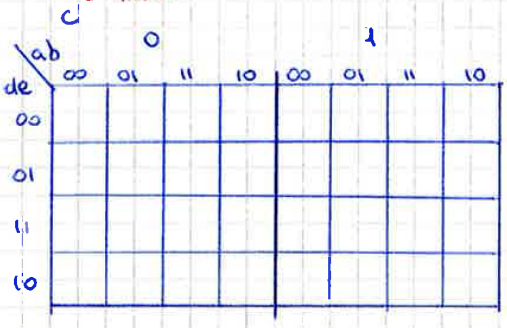
2 INPUT



3 INPUT



4 INPUT



5 INPUT

BOOLEAN FUNCTIONS IN CIRCUITS DESIGN

Funzione completamente specificata se $dcs(f) = \emptyset$, altrimenti incompletamente specificata

Come rappresento le funzioni?

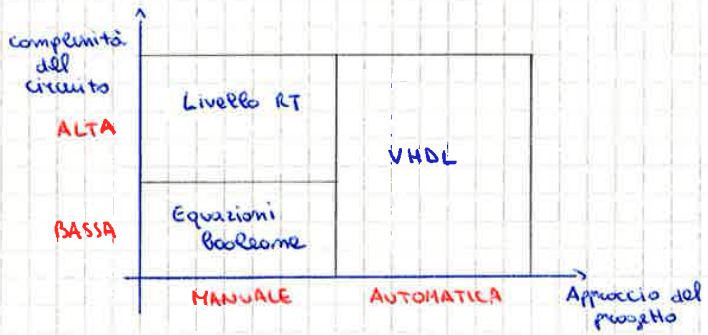
Rappresento sui k-cubes i vertici in cui la funzione assume valore "1", utilizzando spigoli, vertici e facce se vogliamo espanderle in modo da renderle più facce possibile

Dalla funzione al circuito...

OPERAZIONE	PORTA LOGICA
Somma	OR
prodotto	AND
complemento	NOT

Si cerca poi di applicare i teoremi per semplificare ulteriormente il circuito al fine di abbattere il più possibile i costi.

SINTESI MANUALE DI CIRCUITI LOGICO-COMBINATORI



- Somma completa: rappresentazione di una funzione come la somma di tutti i suoi implicanti principali

- Funzione di copertura: funzione che include tutti i vertici "1" e nessuno di "ϕ"

- Copertura non ridondante: se e solo se eliminando uno qualsiasi dei suoi elementi non è più la copertura (somma di implicanti principali essenziali)

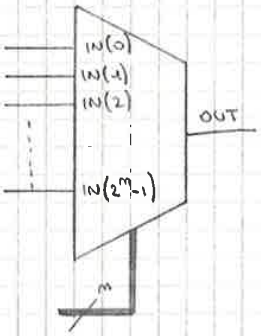
- Somma a costo minimo: se è somma di implicanti principali

- Minimizzazione di output multipli: lavorando con + funzioni contemporaneamente cerco di usare implicanti comuni per ridurre spazio e costi

② MULTIPLEXER

Blocco combinatorio capace di fornire l'uscita al valore di uno dei suoi ingressi in base ai segnali di controllo

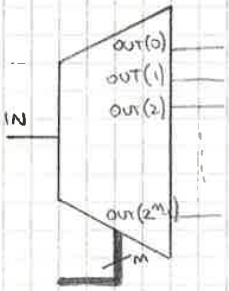
- 2^K INPUTS
- 1 CONTROLLI DI INPUT (da k bit)
- 1 OUTPUT



③ DEMULTIPLEXER

Blocco combinatorio capace di fornire il valore di uno dei suoi output a partire da quello degli input secondo i controlli

- 1 INPUT
- 1 CONTROLLO DI INPUT (da k bit)
- 2^K OUTPUT



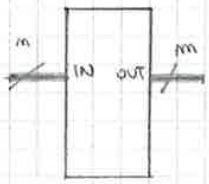
Il demultiplexer soffre di un ritardo intrinseco di trasmissione

(7) CONTATORE DI UNI

1 INPUT (n bit)

1 OUTPUT (m bit) dove $m = \log_2(n+1)$

Conta # di uni in ingresso



(8) ROM

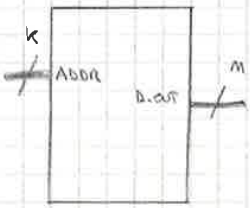
Read only memory

1 INPUT (k bit)

1 OUTPUT (n bit)

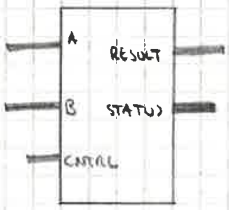
2^k CELLE (n bit ciascuna) per memorizzare i dati

In output ho il valore memorizzato nella cella J, con J valore dell'input



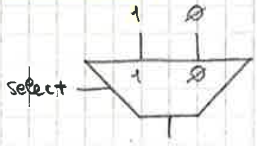
(9) ALU

Arithmetic logic unit: compie operazioni logiche o aritmetiche su due operandi di n-bit, sotto il controllo di un segnale

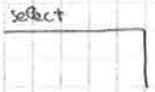


OTTIMIZZAZIONI DI BLOCCHI RT

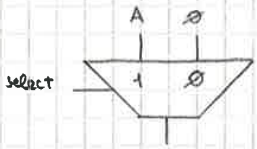
I



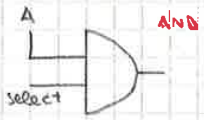
≡



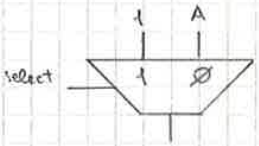
II



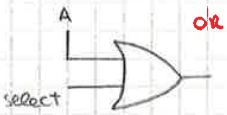
≡



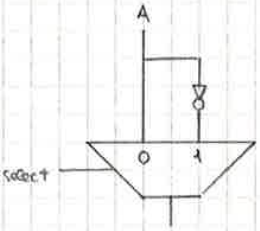
III



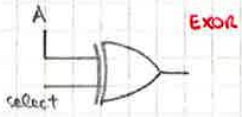
≡



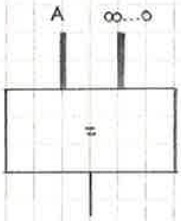
IV



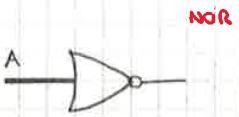
≡



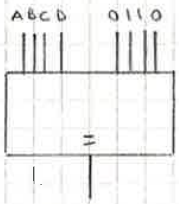
V



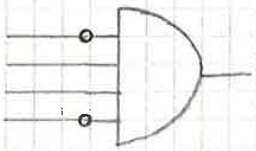
≡



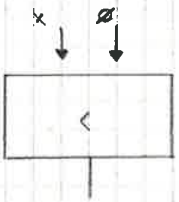
VI



≡

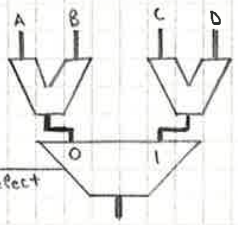


VII

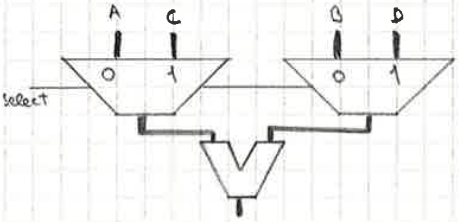


≡ MSB (most significant Bit considerado x n CAL)

VIII



≡



(2) IBM LSSD D-LATCH

Modifica di un D-Latch adottato dalla IBM

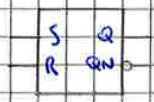
(3) SR LATCH (set-reset)

È considerato il blocco elementare per la costruzione e implementazione di circuiti sequenziali sincroni

Ci sono:

2 INPUT (set e reset)

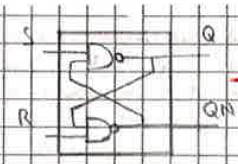
2 OUTPUT (Q e QN, tale che $QN = Q'$)



Delle quattro combinazioni di S e R:

- Una forza Q QN = 0 1 → RESET
- Una forza Q QN = 1 0 → SET
- Una lascia Q e QN invariati
- Una è proibita

NAND IMPLEMENTATION:



Q_{-1} \ SR	00	01	11	10
0	1	1	0	0
1	1	1	1	0

Q

(KARNAUGH)

S	R	Q	QN
0	0	~~1~~	~~1~~
0	1	1	0
1	1	Q_{-1}	QN_{-1}
1	0	0	1

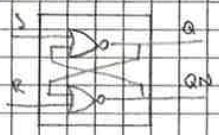
$QN = Q$! NO!

(CHARACTERISTIC TABLE)

$Q_{-1} \to Q$	S	R
0 → 0	0	-
0 → 1	1	0
1 → 0	0	1
1 → 1	-	1

(TRANSITION TABLE)

NOR IMPLEMENTATION:



Q_n \ SR	00	01	11	10
0	0	1	0	0
1	1	1	0	0

Q

(KARNAUGH)

S	R	Q	QN
0	0	Q_{-1}	QN_{-1}
0	1	1	0
1	0	0	1
1	1	~~0~~	~~0~~

(CHARACTERISTIC TABLE)

$Q_{-1} \to Q$	S	R
0 → 0	-	0
0 → 1	0	1
1 → 0	1	0
1 → 1	0	-

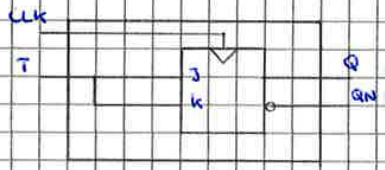
(TRANSITION TABLE)

④ T FLIP-FLOP

T	Q	QN
0	Q_{-1}	Q_{N-1}
1	Q_{N-1}	Q_{-1}

Q_{-1}	Q	T
0	0	0
0	1	1
1	0	1
1	1	0
-	0	?
-	1	?

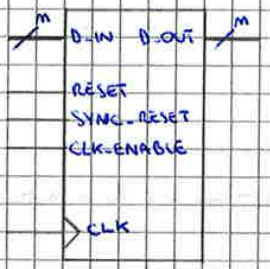
È possibile implementare un T a partire da un JK:



BLOCCHI SEQUENZIALI A LIVELLO RT

① REGISTER

Un registro a n-bit è un blocco sequenziale capace di misfiorare operazioni quali:
- caricamento completo
- scarico completo
- caricamento parallelo di dati n-bit
- memorizzare dati



= Tanti flip flop messi insieme!

② COUNTER

Ci sono due tipi di contatori: • UP-DOWN MODULO m COUNTERS
• 1 TO m UP COUNTERS

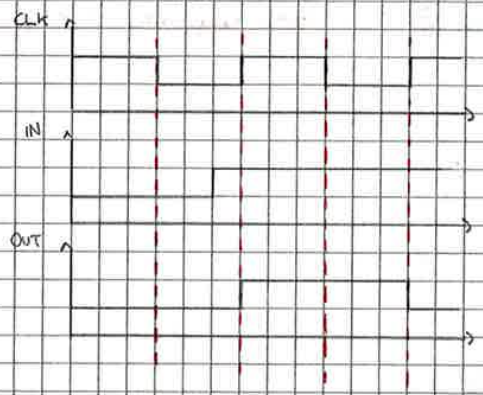
Noi vedremo solo il UP-DOWN MODULO m COUNTER, che è come un registro a n-bit ma può anche incrementare o decrementare il modulo del dato memorizzato

Ogni contatore è caratterizzato dal suo modulo come massimo numero m di diverse configurazioni di uscita che può enumerare.

Tale m deve essere minore o uguale di $2^{\#FF}$ e un contatore di modulo m fornisce valori compresi tra 0 e m-1

(6) PULSE GENERATOR

Blocco sequenziale capace di generare un impulso pari all'ampiezza del periodo del CLK ogni volta che in ingresso è rilevato un fronte di salita

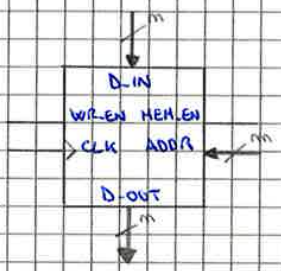


(7) RAM

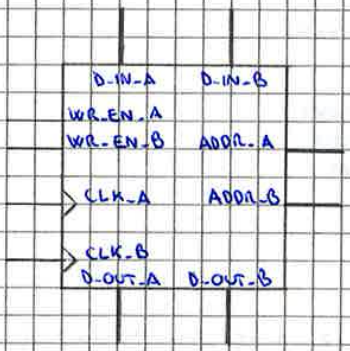
Due tipi di RAM: • porta singola
• porta doppia

Porta singola:

Una RAM a porta singola di n bit è un blocco sequenziale che:
- seleziona uno delle sue m posizioni e ci memorizza un dato di n bit
- " " " " " " " " ne legge il contenuto
- memorizza dati



Porta doppia:

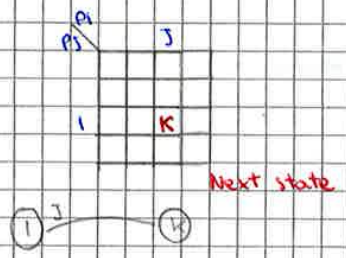


Una volta ottenuta la STG minimizzata è conveniente trattare i risultati in modo tabulare con STT oppure POT. Ambedue hanno: • # righe = # colonne (una riga per stato)

• # colonne = $2^{\#PI}$ (tante colonne quante le possibili combinazioni)

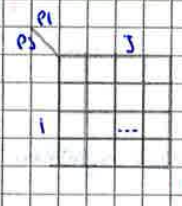
STT

La cella (i,j) memorizza il valore dello stato successivo K dello stato I quando il PI assume valore J

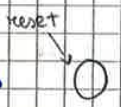


POT

La cella (i,j) memorizza il valore del PO quando la macchina è allo stato i e il PI assume valore J



→ Parliamo ora di segnale di reset e stato di reset:



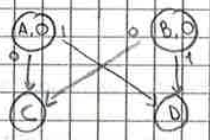
Lo stato di reset dovrebbe essere chiaro e univocamente identificato

Quando raggiunto forza a Ø l'uscita ma non è necessario implichi che PO's vanno a Ø siccome questo dipende dalla struttura della rete delle PO

STATE MINIMIZATION

Una volta terminato il STG e/o la STT si cercano stati equivalenti e si fondono/cancellano

Stati equivalenti: 2 stati sono equivalenti se e solo se essi producono stessi outputs e hanno gli stessi stati successivi. Ne me frega degli input



A≡B

Una volta trovati stati equivalenti si procede in 2 modi:

- o cancello A o B e ricollego tutti gli archi di quello eliminato all'altro
- o fondo A e B in un unico stato ed elimino gli stati ridondanti

MEMORY DEVICES

SRAM:
- Dimensioni della cella (:|)
- Scalabilità (:|)
- Tempo di accesso (:))
- Potenza di consumo (:|)
- Durata (:))
- Ridotto rischio dati (:|)
- È volatile
- Costi di produzione (:|)

Le celle composte da flip-flop e composti dalla linea Bit con 2 transistor
Quando la cella memorizza un bit, esso mantiene il valore finché il valore opposto non è scritto nella cella

DRAM:
- Dimensioni della cella (:))
- Scalabilità (:|)
- Tempo di accesso (:))
- Potenza di consumo (:|)
- Durata (:))
- Ridotto rischio dati (:|)
- È volatile
- Costi di produzione (:|)

Dynamic RAM che memorizza ogni bit usando un transistor come switch per caricare o scaricare il capacitore

FLASH:
- Dimensioni della cella (:))
- Scalabilità (:|)
- Tempo di accesso (:|)
- Potenza di consumo (:))
- Durata (:|)
- Ridotto rischio dati (:|)
- Non è volatile
- Costi di produzione (:))

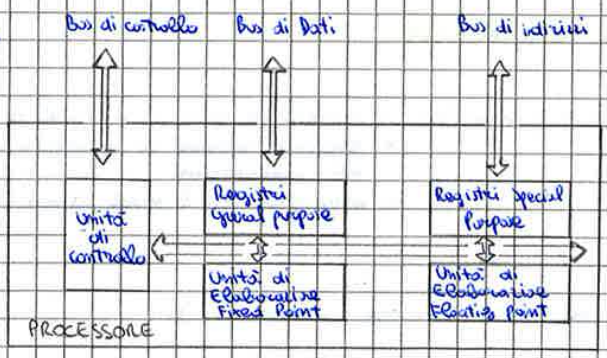
Ne esistono 2 tipi (NAND e NOR)

* NOR → Accede random alle celle; la lettura è veloce; scrive e cancella lentamente; usata maggiormente per memorizzare codice eseguibile in sistemi embedded

* NAND → Legge e scrive molto velocemente; alta densità; buono costo di memorizzazione dati; buono consumo; usata maggiormente per memorizzare altre quantità di dati

ROM:
- Read only memory (ROM)
- Erasable, Programmable Read only memory (EPROM)
- Electrically Erasable, programmable Read only memory (EEPROM)

Basic CPU architectures:



Registri:
- ogni processore ne contiene un set
- diamo dimensione del registro il n° di bit del parallelismo del processore

Tali registri sono classificati in:

- registri di "general purpose":

  accessibili dal programmatore e usati per salvare dati e/o operandi per le istruzioni.

  A loro volta si dividono in registri fixed e floating point

- registri di "specialized purpose":

  hanno scopi specifici nelle operazioni del processore, tipo:

  A) INSTRUCTION REGISTER (IR): contiene l'ultima lettura di fetch
  B) MEMORY ADDRESS REGISTER (MAR): contiene l'indirizzo a cui il processore sta attualmente accedendo
  C) MEMORY DATA REGISTER (MDR): contiene il dato letto o da scrivere in memoria
  D) STACK POINTER (SP): organizza la porzione di memoria da usare come "stack"
  E) STATUS REGISTER (SR): contiene info circa lo stato di esecuzione del processore
  F) PROGRAM COUNTER (PC): contiene indirizzi memoria da cui leggere le istruzioni

Sottosistemi di memoria:

Esistono operazioni nelle memorie:

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 39 di 76

Il sistema operativo dunque è un insieme di programmi che:

- schedulano i processi di esecuzione
- virtualizzano le risorse
- gestiscono archiviazione e accesso ai file

Al fine di lavorare in multitasking e multiprogrammazione il SO lavora in modi diversi, cioè in "Utente" ne ci sono limitazioni di protezione, "Supervisore" senza alcuna limitazione.

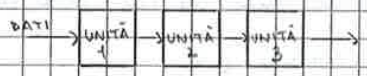
Ciò perché un SO deve:

1. GESTIRE I PROCESSI
2. " LA MEMORIA
3. " LA CPU

PIPELINES

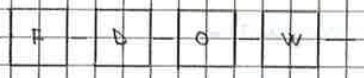
La pipeline è l'equivalente elettronico della catena di montaggio, poiché rende parallelo un processo che era originariamente sequenziale al fine di ottimizzare il tempo.

Dunque eseguo in parallelo fasi diverse sullo stesso flusso di dati



Come visto nei processori, l'esecuzione di un'istruzione è divisa in 4 fasi:

1. FETCH: carico l'istruzione dalla memoria
2. DECODE: decodifica l'istruzione e carico gli operandi sorgente
3. OPERATE: esecuzione dell'operazione
4. WRITE: scrittura del risultato



Colpo di CLK	1	2	3	4	5	6	7	8
I_1	F	D	O	W				
I_2		F	D	O	W			
I_3			F	D	O	W		
I_4				F	D	O	W	

Una pipeline è detta "piena" o "a regime" se ogni istruzione richiede K colpi di clock ma ad ogni periodo del clock viene completata un'istruzione

Le istruzioni di salto sono potenzialmente dannose per la pipeline perché interrompono il normale flusso di esecuzione sequenziale

Per attenuare gli effetti negativi che derivano dalla pipeline a salti si può anche qui utilizzare:

① TECNICHE HARDWARE

- Posso limitarsi a rilevare i salti e svuotare la pipeline dalle istruzioni erroneamente caricate
- Posso implementare tecniche + sofisticate da predire dal fatto che il salto sia condizionato o incondizionato

SALTI INCONDIZIONATI: La fetch riconosce un salto incondizionato e procede alla decodifica dell'istruzione per caricare, nella coda istruzioni, l'istruzione destinazione del salto (BRANCH FOLDING)

SALTI CONDIZIONATI: Bisogna attendere che venga eseguita l'istruzione prima del salto per sapere se il salto vada eseguito o no. Nel frattempo la coda viene caricata sequenzialmente e se eseguo il salto vorrà dire che alcune istruzioni subiranno un ritardo (BRANCH DELAY SLOT)

Posso dotare la FETCH di un'unità di previsione sul fatto che il salto vega eseguito o no al fine di ridurre la penalizzazione di ritardo del salto stesso.
Se la previsione è positiva carica dalla a partire dalla istruzione di destinazione, altrimenti continua a caricare le istruzioni successive al salto.
Dopodiché si verifica la correttezza della previsione e in caso si svuota la pipeline e per farlo bisogna preventivamente avvertire il processore

Ci sono 2 previsioni:

① STATICA: Il compilatore codifica nelle istruzioni di salto come fare la predizione

② DINAMICA: A ogni istruzione di salto condizionato sono associati uno o due flag che indicano la previsione

② TECNICHE SOFTWARE:

Basate su istruzioni NOP a partire dal compilatore e sulla riorganizzazione del codice che mira a garantire che nell'intervallo di ritardo del salto vi siano delle istruzioni NOP e delle istruzioni che vanno comunque eseguite

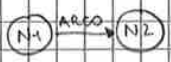
GRAFI - INTRODUZIONE e DEFINIZIONI

Il grafo permette di rappresentare le relazioni fra gli oggetti:
gli oggetti sono vertici, le linee che collegano i vertici le relazioni fra essi

Si definisce "GRAFO" una coppia ordinata (V,E) di insiemi:

- V è un insieme finito e non vuoto di oggetti, detti nodi o vertici
- E " " " di archi o spigoli, collegamento di nodi

→ $|V|$ = # vertici
→ $|E|$ = # archi



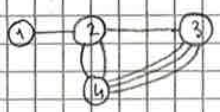
Quando gli archi sono orientati, ossia hanno una direzione, il grafo si dice orientato.

NON ORIENTATI:
Viceversa se non è orientato è rappresentato per mezzo di una coppia di vertici $e=\{v_1,v_2\}$, dove v_1 e v_2 sono detti "adiacenti" e l'arco è "incidente" su di essi.
Sono ammessi archi incidenti su vertici coincidenti e sono detti "CAPPI"

Quando invece il grafo è orientato si dice che:

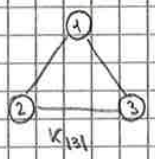
- V_1 è la CODA di e
- V_2 è la TESTA di e
- V_1 è PREDECESSORE di V_2
- V_2 è SUCCESSORE di V_1
- V_1 è ADIACENTE a V_2
- V_2 è " a V_1
- e è INCIDENTE DA V_1
- e " INCIDENTE A V_2

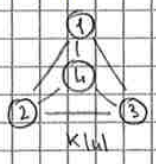
Quando un grafo collega due nodi per mezzo di + archi stiamo parlando di un "MULTIGRAFO"



GRAFO SEMPLICE: non contiene cappi

GRAFO COMPLETO: se l'insieme E di un grafo semplice contiene tutte le possibili coppie





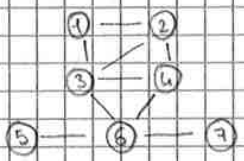
- Il generico $K_{|V|}$ avrà $|V|\cdot(|V|-1)$ spigoli se il grafo è orientato
- " " $K_{|V|}$ " $\frac{|V|\cdot(|V|-1)}{2}$ " " " " " non orientato

GRADO o VALENZA: $p(v)$ è il # di archi incidenti il nodo v.
Esistono poi grado di mezzo $(\overline{p}^0(v))$ e di uscita $(\overline{p}(v))$

GRAFO REGOLARE: grafo in cui tutti i nodi hanno lo stesso grado "r"
(Un grafo semplice $K_{|V|}$ è regolare di grado $|V|-1$)
(" " completo $K_{|V|}$ è regolare di grado $|V|$)

→ Non pesato: incidente di un grafo non pesato di dimensioni $|V| \times |V|$ dove il generico a_{ij} vale:

- 1 se $(v_i, v_j) \in E$
- ∅ altrimenti



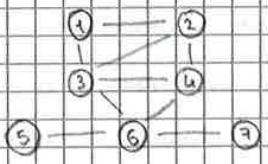
$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

② LISTE DI ADIACENZA

Usate quando si vogliono avere informazioni sugli archi esistenti

- per i grafi NON ORIENTATI: $\forall$ vertice si memorizza la lista dei vertici adiacente
- " " " ORIENTATI: $\forall$ vertice si memorizza la lista dei successori

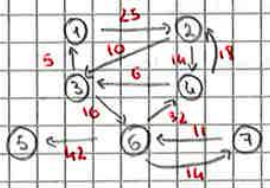
→ Non orientato:



1 → 2 → 3 •
2 → 1 → 3 → 4 •
3 → 1 → 2 → 4 → 6 •
4 → 2 → 3 → 6 •
5 → 6 •
6 → 3 → 4 → 5 → 7 •
7 → 6 •

! Non vuol dire necessariamente che 2 e 3 siano collegati !

→ Orientato:



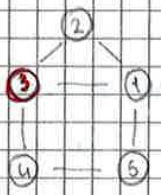
1 → 2 | 25 •
2 → 3 | 10 → 4 | 14 •
3 → 1 | 5 → 6 | 16 •
4 → 3 | 6 → 2 | 18 •
5 •
6 → 4 | 32 → 5 | 42 → 7 | 14 •
7 → 6 | 11 •

TEOREMA: L'algoritmo di visita in ampiezza permette di visitare tutti i vertici del grafo se e solo se questo è connesso

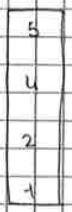
(2) VISITA IN PROFONDITÀ (Depth first)

Tra i vari vertici usati per il proseguimento della ricerca viene sempre scelto quello visitato più di recente

Al generico passo, appena visitato il nodo k si cerca un nodo non ancora esaminato e adiacente a k; se esiste un tale nodo la visita prosegue altrimenti si ritorna al nodo j, da cui si era partiti per arrivare a k e si cercano nodi adiacenti a j e non ancora visitati e così via.



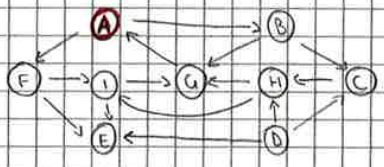
1 → 2 → 3 → 5 •
2 → 1 → 3 •
3 → 1 → 2 → 4 •
4 → 3 → 5 •
5 → 1 → 4 •



È possibile implementare la visita in profondità in modo iterativo come per l'ampiezza, però con uno stack!
Inizialmente lo stack contiene solo la "s", nodo iniziale; se da un generico nodo j che si trova al top dello stack, è possibile raggiungere un nuovo nodo k non ancora visitato, allora k viene inserito in cima allo stack, altrimenti j viene eliminato dallo stack.

Nodi Visitati: 3 1 2 5 4

Nel caso in cui il grafo sia un albero e il nodo s la radice di esso, allora effettuare una visita in profondità del grafo è equivalente ad una visita preorder dell'albero



PROFONDITÀ: A B C H G I E F
(D è irraggiungibile da A)

AMPIEZZA: A B F C G E I H
(D è irraggiungibile da A)

CAMMINO MINIMO: è qualsiasi altro cammino tra 2 vertici la cui lunghezza maggiore o uguale

ALBERO DEI CAMMINI MINIMI: Fissato un vertice s, l'insieme dei cammini minimi tra s e ogni altro vertice da esso raggiungibile è detto albero dei cammini minimi a partire da s

Algoritmi per la determinazione dell'albero dei cammini minimi:

- Algoritmo basato sulla visita in ampiezza
- Algoritmo greedy di Dijkstra (complessità $O(|E|+|V| \cdot \log|V|)$

Il problema della determinazione dei cammini di costo minimo tra tutte le coppie di vertici di un grafo può essere risolto:

- ripetendo gli algoritmi precedenti $|V|$ volte, facendo coincidere s con tutti i vertici del grafo
- algoritmo di Floyd che impiega una matrice di adiacenza A, il cui generico a_{ij} conterrà, alla fine delle iterazioni, la lunghezza del cammino minimo tra V_i e V_j

ALBERO RICOPRENTE: Dato un grafo non orientato, viene definito albero ricoprente quel sottografo che è un albero e contiene tutti i vertici del grafo

MINIMO ALBERO RICOPRENTE: Dato un sottografo G' di un grafo G non orientato e pesato, si definisce peso di G' la somma dei pesi degli archi appartenenti a G'.
Di questi sottografi quello con peso minimo è l'albero ricoprente minimo
La determinazione di questo albero è quella di:

- Dijkstra - Prim

PSEUDO CODICI

1) PREORDER

```
void PreOrder (nodo x)
{ if (x != NULL)
  {
    visita (x);
    PreOrder (x -> sinistra);
    PreOrder (x -> destra);
    return;
  }
}
```

2) INORDER

```
void InOrder (nodo x)
{ if (x == NULL)
    return;
  InOrder (x -> sinistra);
  visita (x);
  InOrder (x -> destra);
  return;
}
```

3) POSTORDER

```
void PostOrder (nodo x)
{ if (x == NULL)
    return;
  PostOrder (x -> sinistra);
  PostOrder (x -> destra);
  visita (x);
  return;
}
```

4) MERGE SORT

```
Mergesort (a[], left, right)
  if left < right  then
    center ← (left + right)/2;
    Mergesort (a, left, center);
    Mergesort (a, center + 1, right);
    merge (a, left, center, right)
```

Tale metodo fonde (merge) due sequenze ordinate in una ancora ordinata applicando il principio il "divide et impera"!

5) FATTORIALE (RICORSIVO)

```
double fact (double N)
{
  double sub;
  if (N == 0)
    return 1.0;
  sub fact (N-1);
  return N * sub;
}
```

6) FATTORIALE (ITERATIVO)

```
double fact (double N)
{ tot = 1.0;
  for (i=2; i<=N; i++)
    tot = tot * i;
  return tot;
}
```

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 53 di 76

(12) BFS (visita in ampiezza)

```
breath_first (vertex)
{
    visit (vertex);
    enqueue (vertex);          FIFO
    while (queue non è vuota)
    {
        x = dequeue ();
        for (ogni vertice w adiacente a x e non ancora visitato)
        {
            visit (w);
            enqueue (w);
        }
    }
}
```

(13) DFS (visita in profondità)

```
depth_first (vertex)
{
    visit (vertex);            LIFO
    for (ogni vertice w adiacente a vertex e non ancora visitato)
        depth_first (w);
}
```

(14) DIJKSTRA

```
S = empty set;
for (tutti i vertici w)
    D[i] = C[1,i]
for (i=1; i<n-1; i++)
{
    scegli un vertice w in V-S tale che che D[w] sia minimo;
    inseriscì w in S;
    for (tutti i vertici v in V-S)
        D[v] = min(D[v], D[w]+C[w,v])
}
```

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 55 di 76

19 THE KNIGHT TOUR

```
Considerare la scacchiera totalmente bianca
void mossa_cavallo (int x, int y)
{
   if (tutta la scacchiera colorata)
      exit();
   if (mossa possibile da (x,y) ad una casella bianca)
   {
      muovi il cavallo in (a;b);
      colora (a,b);
      mossa_cavallo (a,b);
   }
   else
      torna al I step con un'altra mossa possibile;
   /* NOTA: bisogna colorare le caselle già visitate */
}
```

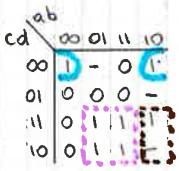
20 8 QUEENS

```
void putQueen(row)
{
   for (ogni posizione "col" nella riga "row")
   {
      if (posizione "col" disponibile)
      {
         Piazza la regina successiva in posizione "col"; /* NOTA: riga fissata da parametro */
         if (row < 8)
            putQueen (row+1);
         else
            exit(); /* successo */
      }
      else
         Togli la regina dalla posizione (row,col);
   }
}
```

ESERCIZI ALGORITMI

MAPPE KARNAUGH

- 2_1-002 del 2_1.3 pag. 48



Trovare tutti gli implicanti principali ed essenziali

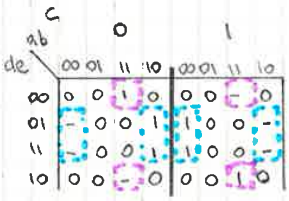
- $\bar{b}\bar{c}\bar{d}$
- bc → $\bar{b}\bar{c}\bar{d} + bc + a\bar{b}c$
- $a\bar{b}c$

- 2_1-002 del 2_1.3 pag. 87

Trovare una copertura non ridondante

Quella dell'esercizio soprastante lo è

- 2_1-006 del 2_1.3 pag. 111



- $ab\bar{e}$
- $\bar{b}e$

→ $ab\bar{e} + \bar{b}e$

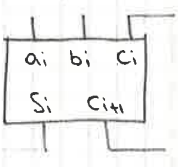
L'espressione finale non dipende da c ma da d!

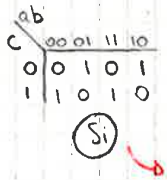
Copro i dcs con "1" e trovo che le 2 mappe si eguagliano

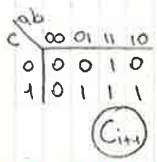
- 2_1-028 del 2_1.4 pag. 13

Disegnare un full adder di 1 bit che ha:

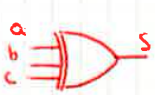
- 3 INPUTS (a_i, b_i, c_i)
- 2 OUTPUTS (c_{i+1}, S_i)







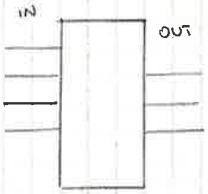
→ mappa di sinistra = EXOR(a,b,c)



- 2.1-007 del 2.1.5 pag. 56

Progettare un circuito combinatorio con 4-bit input e 3-bit output che quando riceve x fornisce in uscita $\lceil sqrt(x) \rceil$

4 bit → 0, ..., 15



x(3)x(2) \ x(1)x(0)	00	01	11	10
00	000 (0)	010 (4)	100 (12)	011 (8)
01	001 (1)	011 (5)	100 (13)	011 (9)
11	010 (3)	011 (7)	100 (15)	100 (11)
10	010 (2)	011 (6)	100 (14)	100 (10)

dividi gli output

out 1

	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	0	0	1	1
10	0	0	1	1

out 2

	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

out 3

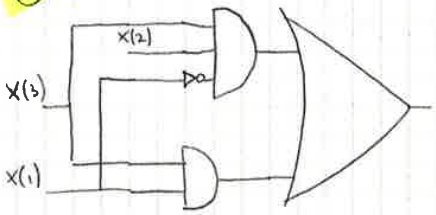
	00	01	11	10
00	0	0	0	1
01	1	1	0	1
11	0	1	0	0
10	0	1	0	0

① $x(3)x(2)\overline{x(1)} + x(3)x(1)$

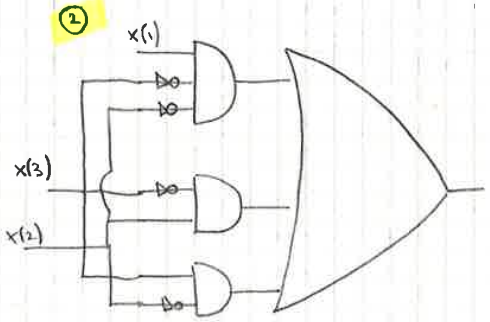
② $\overline{x(3)}\,\overline{x(2)}x(1) + \overline{x(3)}x(2) + x(3)\overline{x(2)}$

③ $\overline{x(3)}\,\overline{x(1)}x(0) + \overline{x(3)}x(1)x(2) + x(3)\overline{(x(2)}\,\overline{x(1)}$

①



②



③

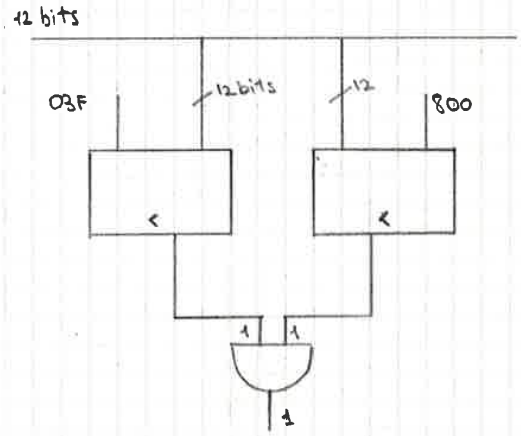
and xo on ...

CIRCUITI COMBINATORI CON BLOCCHI RT

- 2.1-029 del 2.1.6 pag.62

Progettare un dispositivo da connettere a un Bus di 12-bits, che forzi a 1 l'uscita U quando:

$$03F < A < 800$$



- 2.1-037 del 2.1.6 pag.53

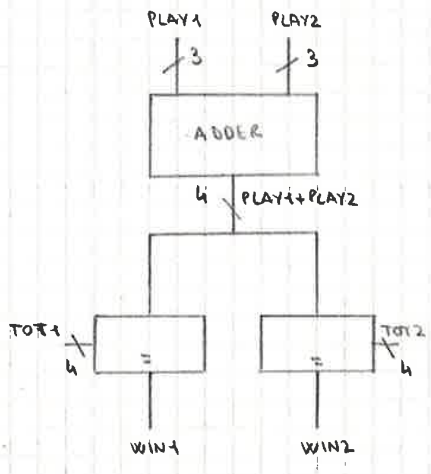
Progettare un circuito capace di individuare il vincitore del gioco "morra"

Ogni partita coinvolge 2 giocatori che:

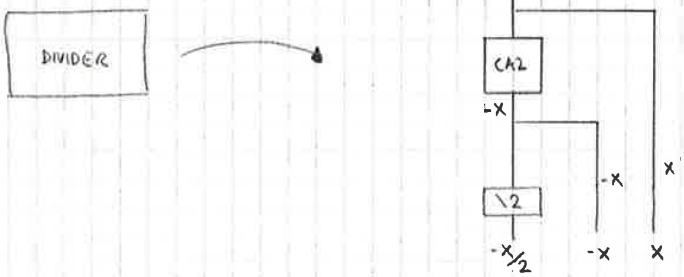
- danno un n° da 0-5 → PLAY
- " " n° " 0-10 → TOT

Se TOT1 = PLAY1 + PLAY2 → 1 vince

Se TOT2 = PLAY1 + PLAY2 → 2 vince



Devo spezzare quel "divider":



- 2-1-043 dal 2-1.6 pag. 119

3 segnali in input: R_1, R_2, R_3 , uno per ogni riga

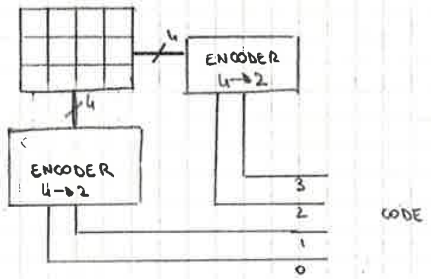
4 " " " : $C_1, \dots, C_4$ " " " colonna

Ogni segnale è attivato quando viene ricevuta una chiave alla riga o colonna premuta

La keyboard deve:

- dare un output di VALID se una sola chiave è premuta
- un codice su 4 bits della chiave premuta

0 0000	1 0001	2 0010	3 0011	00--
4 0100	5 0101	6 0110	7 0111	01--
8 1000	9 1001	* 1010	# 1011	10--
--00	--01	--10	--11	

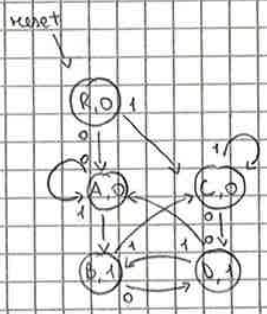


- 2.2-018 del 2.2.3 pag. 72

Progettare un circuito che:

- abbia input x
- un segnale di CLK
- un output z, raggiunto per un ciclo di clock quando in input è rilevato un fronte (salita o discesa)

"01" oppure "10"!

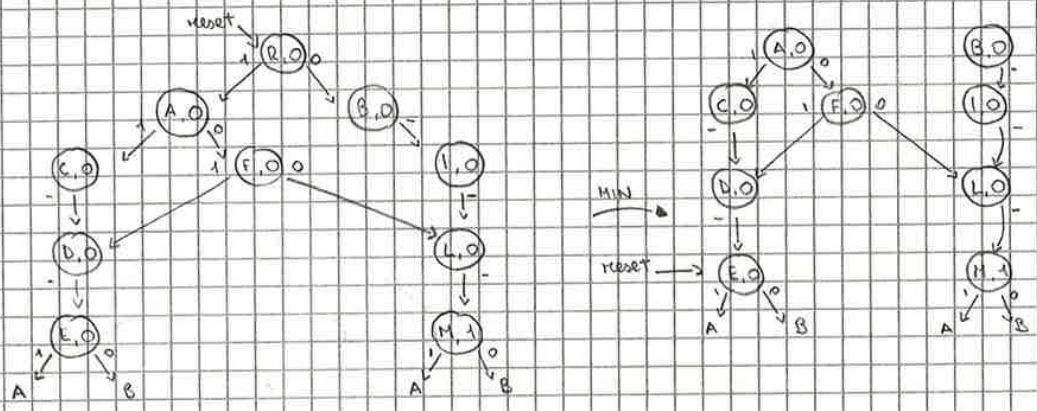


- 2.2-005 del 2.2.3 pag. 89

Vengo trasmessi gruppi di 4 bit secondo MSB first (Big Endian)

Il circuito da progettare deve avere un output z che è posto a 1 per un ciclo di clock solo in corrispondenza del 4th bit se questi quattro sono un codice BCD (0÷9)

A B C D
2^3 2^2 2^1 2^0



- 2.2-009 del 2.2.3 pag. 118

I dati sono trasmessi secondo il "2 fuori di 3"!

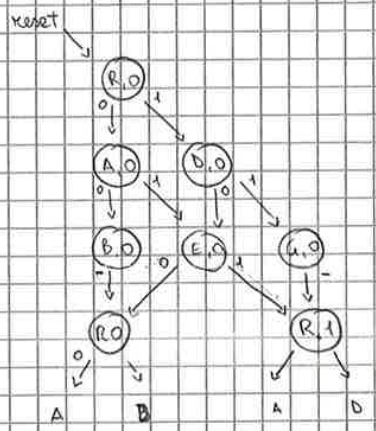
Il trasmettitore codifica ogni dato come 3 identici bit ("1" come "111")

Non ci sono bit di separazione tra 2 dati

Il ricevente decodifica come: "0" quando almeno 2 bit nel gruppo sono "0"

"1" altrimenti

Progettare un circuito tale che, ricevendo questo protocollo in input, in uscita trasmetta come viene inviato un dato



- 2.2-021 del 2.2.3 pag. 122

Progettare un circuito che sia un contex up/down modulo 8.

se UP~DN = 1 ↑

se UP~DN = 0 ↓

UP~DN	0			1			-		
0 0 0	1	1	1	0	0	1	0	0	0
0 0 1	0	0	0	0	1	0	0	0	1
0 1 0	0	0	1	0	1	1	0	1	0
0 1 1	0	1	0	1	0	0	0	1	1
1 0 0	0	1	1	1	0	1	1	0	0
1 0 1	1	0	0	1	1	0	1	0	1
1 1 0	1	0	1	1	1	1	1	1	0
1 1 1	1	1	0	0	0	0	1	1	1
	NS						PO		

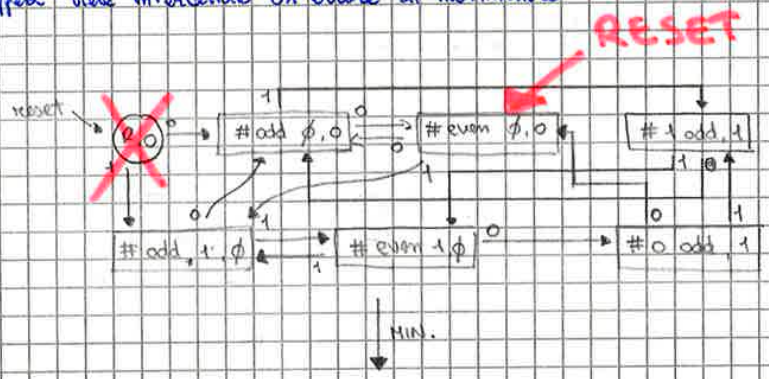
ESEMPI DI SINTESI

- 2.2-010 del 2.2.6 pag. 13

  Le sequenze possono essere di qualsiasi lunghezza ma:

  - La sequenza di "1" deve avere # di 1 dispari
  - " " " "0" " " # di ϕ pari

  Progettare un circuito tale che la sua uscita "ERROR" sia raggiunta per un ciclo di clock non appena viene intercettato un errore di trasmissione



MIN.

R ≡ #even ϕ, 0

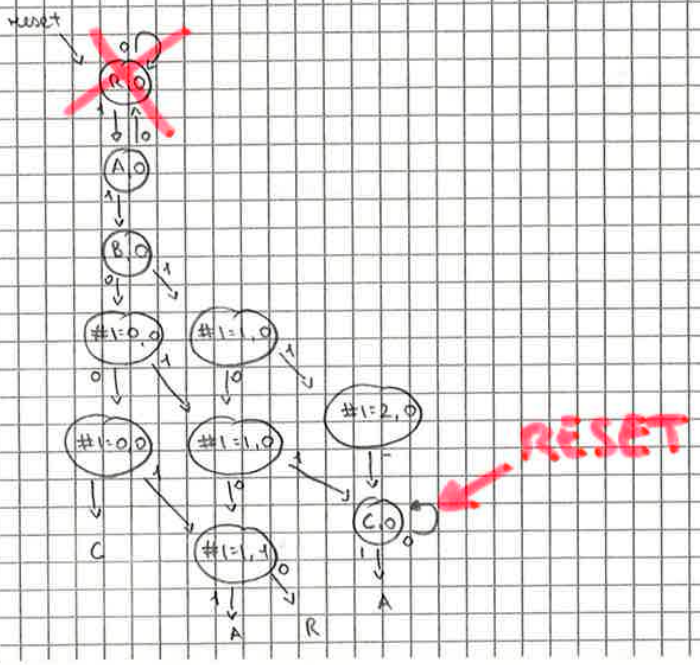
(vedi dopo le metriz)

- 2.2-008 del 2.2.6 pag. 43

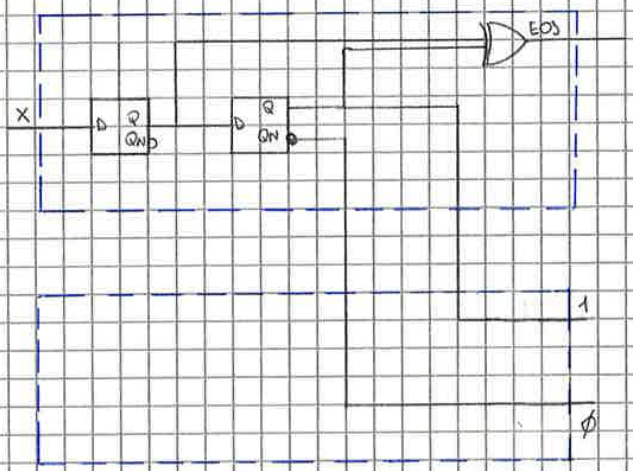
  Trasmetti dati di 3 bit

  Ogni dato è preceduto da due bit di Header "11"

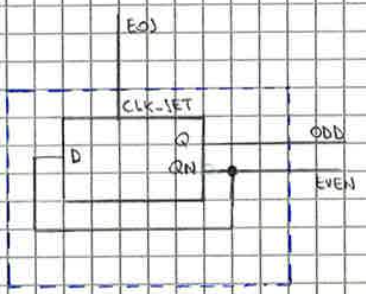
  Bisogna progettare un circuito che, per un ciclo di clock, forzi l'uscita a quando viene ricevuto un dato che contiene uno zero "1"



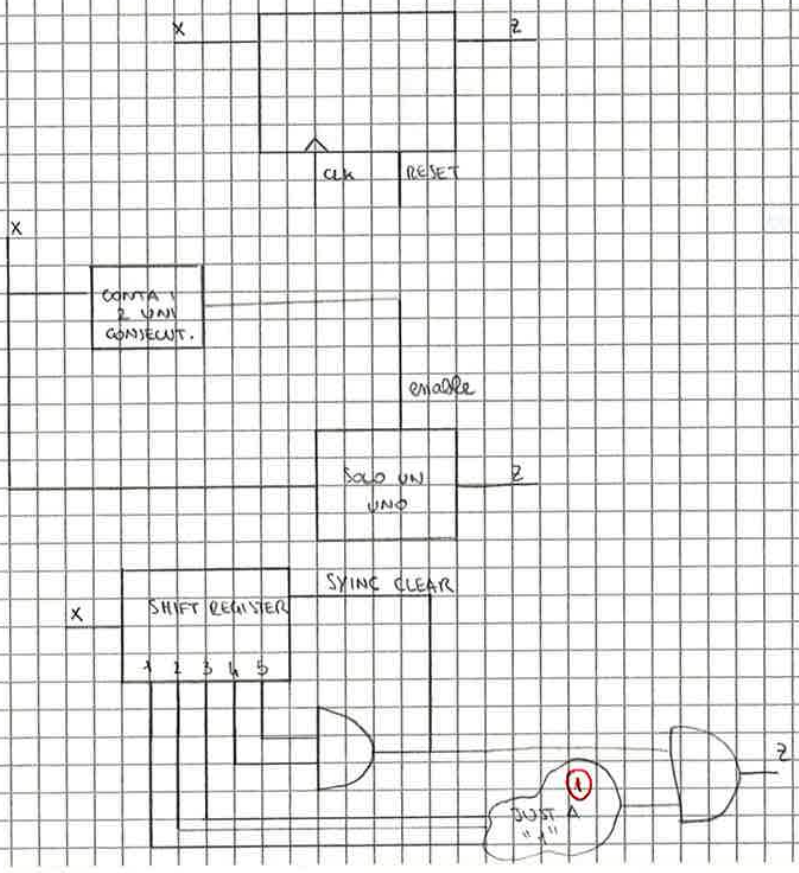
① e ②



③

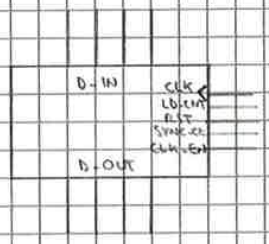


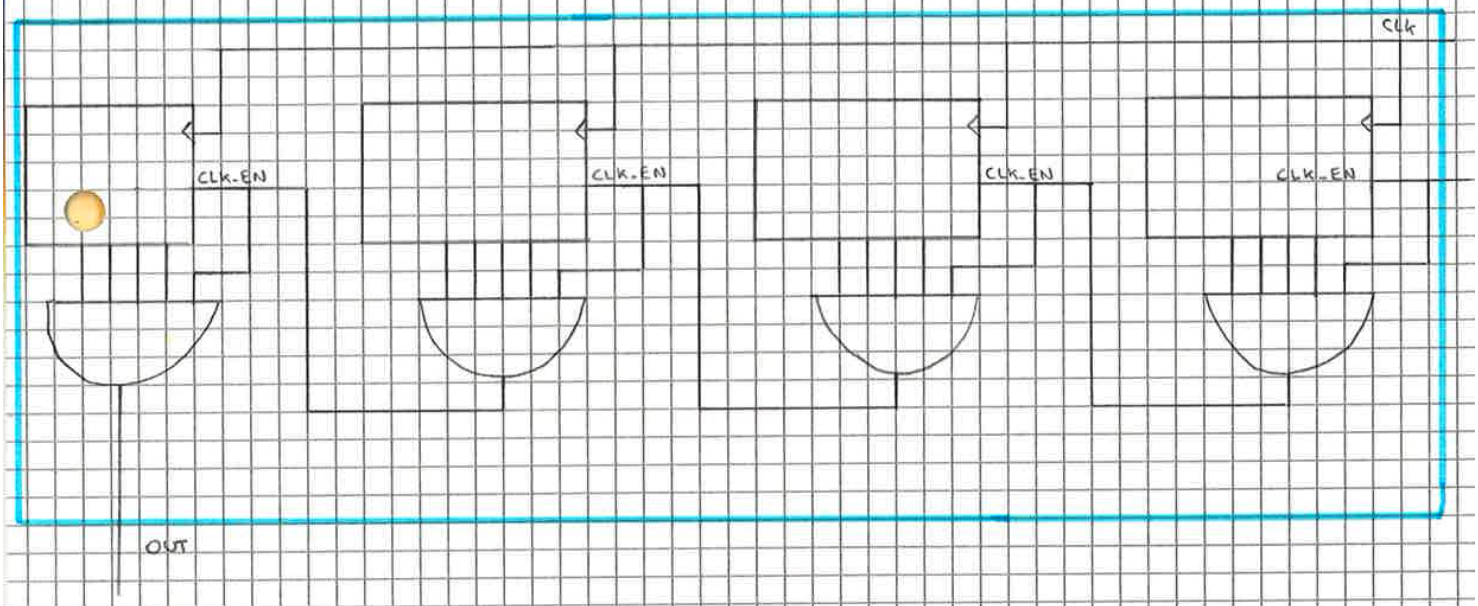
- 2.2-008 del 2.2-6 pag.43 (RT-Level)



• 2.2-022 del 2.2-6 pag. 65

Progettare un contatore da 16 Bit usando il seguente contatore a 4 bit





•