Note introduttive sul
Livello della Logica Digitale

Con le presenti brevi note introduciamo lo studio del livello della Logica Digitale, il Livello 0 dei sistemi di calcolo.

In particolare, ci interessiamo alle problematiche relative alla realizzazione in Hardware delle Macchina Astratte.
Un esempio di realizzazione Hw completa e dettagliata lo avremo quando, studiando il Cap.4 del testo, vedremo come realizzare in Hw la macchina associata al linguaggio Mic-1.

Realizzare in Hw una macchina astratta significa realizzare strutture dati ed algoritmi.
Una struttura dati, in linea di pricipio, si puo' vedere come una funzione. Per esempio, un vettore di numeri e' una funzione che, preso un indice, ci restituisce un numero. Un algoritmo e' la descrizione di un processo composto da vari passi di trasformazione di informazioni.
Per implementare in Hw macchine astratte quindi e' indispensabile riuscire ad implementare **funzioni**.

Ma funzioni con quale particolare dominio e codominio?
In realta' possiamo semplificarci di molto la vita, poiche', come diceva Pitagora:

"*Ogni cosa e' numero*".

Infatti, dal punto di vista dell'informatico (teorico) qualsiasi cosa si puo' rappresentare con un numero.
Quindi consideriamo numeri come dominio e codominio delle nostre funzioni.

Il raggiungimento del nostro obiettivo poi si semplifica ulteriormente se riflettiamo sul fatto che di ogni numero se ne puo' dare una rappresentazione binaria.
Il nostro problema si "riduce" quindi alla possibilita' di realizzare in Hw funzioni del tipo

$$f : \{0,1\}^* \rightarrow \{0,1\}^*$$

dove con $\{0,1\}^*$ si indica l'insieme delle sequenze binarie di qualsiasi lunghezza.

Come primo passo per realizzare funzioni di questo tipo, cerchiamo di realizzare per prima cosa funzioni piu' semplici, e cioe' del tipo

$$f : \{0,1\}^n \rightarrow \{0,1\}^m$$

dove con $\{0,1\}^n$ e $\{0,1\}^m$ indichiamo gli insiemi di stringhe binarie di

lunghezza, rispettivamente, n e m.

E' facile capire come una funzione $f : \{0,1\}^n \rightarrow \{0,1\}^m$ si possa vedere come un *insieme di m funzioni* del tipo

$$f_i : \{0,1\}^n \rightarrow \{0,1\} \qquad 1 \leq i \leq m$$

Ogni f_i restituisce un elemento della sequenza di output di f.
Funzioni come le f_i si chiamano **funzioni booleane**.

Leggendo la sezione 3.1.1 del testo si puo' vedere come, utilizzando dei semplici transistor, sia possibile realizzare delle porte logiche, componenti elettroniche cioe' che corrispondono as alcune semplici funzioni booleane binarie e unarie.
E per tutte le altre?

Per risolvere il problema iniziamo definendo il semplicissimo concetto di circuito.
Un **circuito** e' un insieme di porte logiche connesse tra di loro.
Un circuito si dice **combinatorio** se e' privo di cicli (non esiste cioe' alcun percorso del segnale elettrico che dopo aver attraversato un certo numero di porte logiche ritorna su se stesso).
E' immediato notare come ogni circuito combinatorio sia rappresentabile da una espressione algebrica formata da variabili e funzioni booleane unarie e binarie, e viceversa.

Vediamo ora come ogni funzione booleana sia rappresentabile da una espressione algebrica contenente solo funzioni booleane dell'insieme **{**and, or, not**}**.
Nella sezione 3.1.2 e' mostrato come, data la descrizione esplicita, sotto forma di Tabella di Verita', della funzione booleana ternaria M(A,B,C), detta "funzione di maggioranza", sia possibile arrivare alla corrispondente espressione algebrica.
Poiche' il procedimento indicato per tale esempio non dipende dalla particolare funzione booleana presa in esame, esso si puo' utilizzare per produrre la rappresentazione algebrica di qualsiasi funzione booleana data, e quindi per costruire un circuito che la calcoli.
Questo procedimento e' anche la dimostrazione che l'insieme di operatori **{**and, or, not**}** e' funzionalmente completo.
Un insieme di operatori si dice **funzionalmente completo** quando ogni funzione booleana si puo' rappresentare con un un'espressione algebrica contenente solo variabili e operatori appartenenti all'insieme.
Si possono trovare molti insiemi di operatori funzionalmente completi, come mostrato nella sezione 3.1.3 del testo.
Per esempio, **{**and, not**}**, **{**or, not**}**, **{**nand**}**, **{**nor**}** sono tutti funzionalmente completi.

Nella sezione 3.1.4 si fa vedere come ci possano essere moltissimi circuiti combinatori (in realta' infiniti) che calcolano la stessa funzione booleana. Sceglierne uno piuttosto che un altro dipende dalle priorita'

che diamo ai vari parametri che intendiamo ottimizzare nel nostro progetto di circuito:

- velocita' (tempo di stabilizzazione del segnale di output)
- numero di porte logiche
- numero di intersezioni tra linee
- ecc. ecc.

Quando l'ottimizzazione dei costi (come accade spesso) e' un parametro di rilievo in un progetto, e' abbastanza comune fare quello che si fa solitamente in molte attivita' umane: anziche' costruire in modo costoso qualcosa partendo da zero, si cerca di ottenere un risultato soddisfacente componendo dei moduli che si hanno gia' a disposizione. E' un approccio valido anche per la programmazione, ampliamente supportato dalla programmazione ad oggetti.
Nella sezione 3.2.2 del testo e' possibile trovare la descrizione dei piu' comuni *moduli combinatori*, alcuni dei quali saranno utilizzati quando andremo a realizzare in Hw la macchina astratta Mic-1.

Vediamo un esempio di come, componendo semplici moduli gia' esistenti, sia possibile ottenere un circuito per calcolare una funzione complessa in modo economico (a scapito dell'efficienza).
Supponiamo di voler realizzare un circuito per il calcolo della funzione

$$somma : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$$

la funzione cioe' che somma due numeri rappresentabili in binario con n cifre.
Per far questo potremmo utilizzare il metodo visto per la funzione di maggioranza per ottenere n espressioni algebriche, una per ogni funzione booleana

$$somma_i : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\} \qquad 1 \leq i \leq n$$

Il circuito finale avra' un costo notevole, sia per il costo progettuale che per il numero di porte logiche utilizzate, ma avra' sicuramente un vantaggio: sara' molto veloce.
Infatti i circuiti ottenuti con il metodo che abbiamo visto sono circuiti a due livelli: i segnali di input devono cioe' attraversare sempre solo due porte logiche (se non teniamo in considerazione i not). Questo significa che il segnale di output si stabilizzera' molto in fretta.

Vediamo ora invece una soluzione **modulare** (ed economica) al problema della realizzazione di un circuito per *somma*.
Sappiamo che per sommare due numeri rappresentati in binario possiamo utilizzare uno dei primi algoritmi che impariamo da bambini: sommiamo cifra per cifra, tenendo in considerazione l'eventuale riporto.
L'unica cosa che dobbiamo saper fare per eseguire tale algoritmo e' dunque poter sommare tre cifre binarie (due cifre piu' il riporto proveniente dalla somma della cifra precedente) e poter produrre il riporto per la cifra successiva.

Questa cosa si puo' realizzare in Hw in molti modi. Uno e' il semplice circuito della figura 3-18 del testo (chiamiamo FA, Full Adder, tale modulo).
Questo circuito e' estremamente semplice e si puo' produrre con costi limitati.

Per realizzare ora la nostra funzione *somma* : $\{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ basta prendere n moduli FA collegando l'output Carry-out (riporto in uscita) di ogni modulo con l'input Carry-in (riporto in ingresso) del modulo successivo.
Il costo dell'addizionatore cosi' ottenuto, chiamato anche **Addizionatore Parallelo** e' molto contenuto.
Lo svantaggio pero', lo si puo' intuire, e' dal punto di vista dell'*efficienza*. Questo tipo di addizionatore infatti, pur chiamandosi "addizionatore parallelo", ha ben poco di parallelo. Anche se i vari moduli sembra possano lavorare indipendentemente uno dall'altro, cosi' non e': l'output di ogni modulo non e' corretto finche' non si e' ricevuto il giusto riporto da parte del modulo precedente.
E' quindi chiaro che affiche' si possa considerare corretto l'output dell'ultimo modulo bisogna attendere che l'eventuale riporto si propaghi attraverso tutti gli n-1 moduli precedenti.

L'Addizionatore Parallelo e' un esempio di come ottenere un circuito che lavora su uno o piu' input lunghi m, partendo da m semplici *moduli* che lavorano su singoli bit.
Tale procedimento, in generale, si puo' utilizzare non solo per costruire addizionatori, ma intere **ALU**, come si puo' vedere nella parte dedicata alle Arithmetic Logic Unit della sezione 3.2.3 del testo.
E' proprio l'ALU descritta in tale sezione che verra' utilizzata per implementare la Componente Operazioni della macchina astratta Mic-1.

Altri esempi di moduli combinatori solitamente utilizzati per realizzare circuiti complessi, si trovano nella sezione 3.2.2, come Multiplexer, Decoder, etc. Anch'essi saranno utilizzati per realizzare la macchina Mic-1.

Torniamo ora al nostro problema iniziale: la realizzazione di funzioni del tipo

$$f : \{0,1\}^* \rightarrow \{0,1\}^*$$

per esempio la funzione somma, ma questa volta su tutti i numeri naturali, le cui rappresentazioni quindi possono essere sequenze binarie di lunghezza arbitraria:

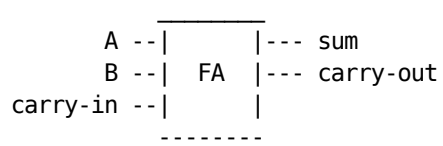
$$somma : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$$

Capiamo subito che, essendo l'insieme $\{0,1\}^*$ costituito da sequenze di lunghezza arbitraria, la possibilita' di realizzare funzioni come

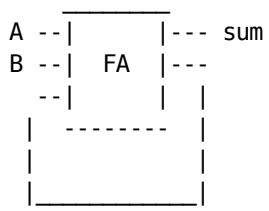
questa con circuiti combinatori viene a cadere immediatamente: un circuito combinatorio puo' avere infatti solo un numero fisso di linee di ingresso.

Potremmo pero' risolvere il problema tornando al nostro algoritmo per sommare numeri binari.
Quello che noi facciamo manualmente e' sommare le cifre in sequenza, una dopo l'altra. Per ogni coppia di cifre e riporto facciamo esattamente quello che fa il nostro modulo combinatorio FA.
Quindi in realta' potremmo utilizzare sempre lo stesso modulo FA per sommare le varie cifre, a patto di poterci ricordare il riporto dalla cifra precedente e di avere le coppie di cifre del numero da sommare non tutte insieme, ma una per volta, in sequenza.

Vediamo come questo si possa ottenere in modo molto semplice.
Se indichiamo il modulo FA come segue:



e' sufficiente collegare l'output carry-out con l'input carry-in nel modo seguente



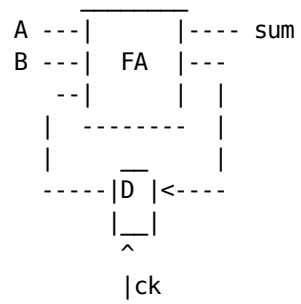
e poi presentare in A e B una dopo l'altra le coppie di cifre da sommare.
Sull'output compariranno, una dopo l'altra, le cifre del risultato.

Quello che abbiamo costruito in questo modo e' un Circuito Sequenziale.
Un **circuito sequenziale** e' un circuito che contiene cicli.
Gli output dei circuiti sequenziali in un dato istante non dipendono solo dal valore degli input, ma anche dalla storia degli input negli istanti precedenti. Nel nostro addizionatore sequenziale infatti il valore della somma di due cifre dipende anche dal riporto, e quindi dal valore delle cifre precedenti che abbiamo sommato negli istanti passati.

E' facile intuire come in ogni circuito sequenziale sia possibile identificare una parte costituita da un circuito puramente combinatorio ed una parte costituita da linee "*di feedback*". Proprio come nel nostro semplice esempio, dove la linea di feedback e' unica.

Osservando con attenzione il nostro addizionatore sequenziale, non possiamo pero' non accorgerci di un problema che abbiamo trascurato: un problema di *sincronizzazione*.
Infatti, non appena il modulo FA ha prodotto un riporto, questo viene immediatamente propagato indietro per essere sommato alle due cifre successive.
Dovremmo quindi essere capaci di produrre le due cifre successive non appena il riporto arriva in input. Se anche ci fosse il minimo scarto temporale, ci ritroveremmo in input delle configurazioni inconsistenti e quindi delle cifre spurie in output.
Inoltre c'e' da considerare il fatto che i tempi di stabilizzazione delle linee di output *sum* e *carry-out* potrebbero essere differenti. Il riporto potrebbe stabilizzarsi prima di *sum* e quindi, una volta tornato indietro, potrebbe influenzare il calcolo della somma delle cifre correnti, anziche' interessare solamente la somma delle cifre successive.

C'e' solo un modo di uscirne fuori: fare in modo che gli input si presentino in sequenza ad istanti prefissati e, ancora piu' importante, impedire che il valore calcolato del riporto in output vada ad influenzare la somma delle cifre correnti.
Questo si puo' ottenere per prima cosa introducendo un "orologio" che ci permetta di identificare una sequenza di istanti temporali: un segnale impulsivo, detto appunto Segnale di Clock (ck).
E poi inserendo nella linea di feedback una componente elettronica D in grado di mantenere al suo interno (quando il ck e' 0) il valore dell'input *carry-in* impedendo nel contempo che il valore calcolato del *carry-out* si propaghi sulla linea finche' non arriva l'istante successivo, quando avremo su A e B anche le cifre successive. Quando arrivera' l'istante temporale successivo (ck diventa 1 per un breve momento) D permettera' al riporto di propagarsi e lo manterra' al suo interno per permettere il calcolo corretto delle somma delle cifre con riporto corretto.



Circuiti sequenziali di questa forma, con componenti come D che permettono il propagarsi del segnale di feedback solo all'inizio dei vari istanti temporali, memorizzando il suo segnale al loro interno, si chiamano **Circuiti Sequenziali Sincroni**. "*Sincroni*" appunto perche' il loro comportamento e' sincronizzato attraverso l'uso di un segnale impulsivo di clock.

La domanda che ci poniamo ora e': cosa potra' mai essere la componente D?
Certo non un circuito combinatorio, poiche' i circuiti combinatori calcolano *funzioni* booleane ("*funzioni*", vale a dire che producono un output dipendente in modo univoco dall'input).
D invece (quando ck vale 0) produce in output il suo "contenuto", che e' indipendente dal valore proveniente dal carry-out.
Se D non e' un circuito combinatorio, allora necessariamente deve essere un circuito sequenziale, con linee di feedback. Pero' non un circuito sequenziale sincrono, altrimenti dovremmo avere un altro D al suo interno...
D deve allora essere un circuito sequenziale **asincrono**, cioe' senza la presenza di altri segnali di clock al suo interno e senza elementi come D che bloccano le linee di feedback.
Ed infatti elementi come D sono costruiti a partire da circuiti sequenziali asincroni, come i latch S-R descritti nella sezione 3.3.1 del testo. A partire da questi si possono costruire piccoli elementi "di memoria" come D, come mostrato sempre nella sezione 3.3.1.

Se ci prendessimo la briga di separare tutta la parte combinatoria da tutte le possibili linee di feedback, si potrebbe vedere come ogni possibile CPU, Memoria e, in generale, microprocessore, per quanto grande, altro non sia che un grosso circuito sequenziale sincrono.

Tutti contenti dopo aver visto come funzioni del tipo $f : \{0,1\}^* \rightarrow \{0,1\}^*$ si possano realizzare in modo sequenziale, dobbiamo pero' capire anche i limiti dei circuiti sequenziali.
Per esempio non e' possibile costruire nessun circuito sequenziale in grado di calcolare la semplice funzione che, preso in input il numero binario 2^n, restituisca il numero 2^{n*n}.
Perche'?
2^n e' rappresentato da un 1 seguito da n zeri. Per produrre 2^{n*n} dobbiamo prima cosa ricevere tutti gli zero, contandoli, in modo da capire quale sia il valore n. Poi (cosa importante) "*ricordarci*" in qualche modo tale valore per calcolare cosi' n*n e produrre n*n zeri seguiti da un 1.
Nei circuiti sequenziali l'unico modo di "*ricordarsi*" quel che e' successo in precedenza e' utilizzando le linee di feedback, rappresentando in esse, in ogni istante j<n, l'informazione "ho incontrato j zeri". Ma il circuito, avendo un numero finito di linee di feedback, puo' ricordarsi solo un numero finito di tali informazioni, mentre il numero n di zeri che posso ricevere in input non e' limitato.
Quindi, niente da fare.
Questo significa che nessun circuito sequenziale, e quindi **nessun computer** potra' mai veramente calcolare la funzione matematica appena descritta.
Nulla di preoccupante, ovviamente. Ogni informatico si accontenta di lavorare su segmenti limitati (per quanto grandi) di valori numerici. Il

problema, se lo vogliamo chiamare cosi', risiede nella quantita finita di memoria.
E comunque, qualcuno ha mai visto un computer con una memoria infinita?...
E' chiaro che per avere computazioni piu'potenti bisogna necessariamente passare a "*memorie*" infinite (che pero' nella realta' non esistono...).
Non per niente una delle caratteristiche principali delle Macchine di Turing, e uno dei suoi punti di forza, e' quella di possedere una capacita' infinita di memoria. E questo un formalismo matematico, a differenza di un computer, puo' permetterselo....

Con questo breve inquadramento dell'argomento Circuiti possiamo andare avanti a studiare tutto quello che di circuiti e memorie tratta il Capitolo 3 del testo.