Architettura degli Elaboratori

Appunti: Simone Pepi
Stesura in LaTeX: Federico Matteoni

Indice

0.1 Introduzione

Appunti del corso di **Architettura degli Elaboratori** presi a lezione da **Federico Matteoni**.

Prof.: **Maurizio Bonuccelli**, maurizio.angelo.bonuccelli@unipi.it
Riferimenti web:

- *http://pages.di.unipi.it/bonuccelli/aeb.html*
- *didawiki.cli.di.unipi.it/doku.php/informatica/ae/start*

Ricevimento: Martedì 10-12, stanza 294 DE
Esame: **scritto** (*closed book*) e **orale**. I compitini sono validi solo per la sessione invernale (gen-feb)
Libri

- M. Vanneschi *Architettura degli Elaboratori*, Pisa UniversitY Press
- D. A. Patterson *Computer Organization & Design - The Hardware/Software Interface*

0.2 Cosa riguarda il corso

Consiste in come sono fatti pc internamento da un punto di vista di sottosistemi senza scendere nei dettagli elettrici.
Il corso è diviso in quattro parti:

- Fondamenti e strutturazione firmware (I Compitino)
- Macchina assembler (D-RISC) e processi
- Architetture General-Purpose
- Architetture parallele (II Compitino)

Capitolo 1

Fondamenti di strutturazione

1.1 Struttura a livelli

Dividere Per dedicarci allo studio di un sistema complesso spesso è utile **dividerlo in pezzi**. Nel caso di un sistema di elaborazione, in alcuni casi è **interessante avere una visione vicina alla struttura fisica** in termini di componenti hardware. In altri casi è **interessante avere una visione astratta del sistema** per poterne osservare le funzionalità e le strutture più adatte alla specifica applicazione.

Astrarre Da questa necessità deriva la possibilità di strutturare un sistema a vari **livelli di astrazione** che non descrivono una reale struttura fisica, ma è **utile per ragioni specifiche** quali:

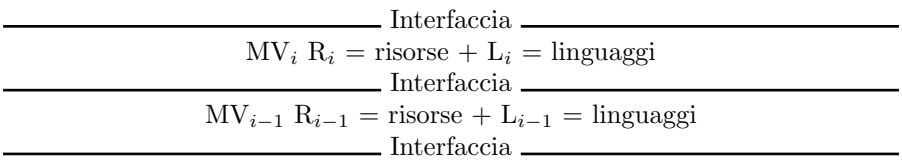
> Saper riconoscere **quale metodo di progettazione strutturata** viene seguito o conviene seguire (**top-down**, **bottom-up**, **middle-out**)
>
> Saper riconoscere **se i vari livelli rispettano una relazione gerarchica** oppure se non esiste alcun tipo di ordinamento
>
> Essere in grado di **valutare a quali livelli conviene descrivere e implementare** determinate funzioni del sistema

1.2 Macchine Virtuali

Sistema di elaborazione **Le funzionalità di un sistema di elaborazione** nel suo complesso possono essere **ripartite su un certo numero di livelli** che vengono definite **macchine virtuali**. La suddivisione può seguire **due approcci** fondamentali:

- **Linguistico**: stabilisce i livelli in base ai linguaggi usati
- **Funzionale**: stabilisce i livelli in base a cosa fanno

I vari livelli sono schematizzati come in figura:



MV$_i$ realizza politica P$_i$ con linguaggio L$_i$ e risorse R$_i$.
MV$_i$ utilizza le funzionalità che il livello MV$_{i-1}$ (cioè le sue primitive) fornisce **attraverso l'interfaccia**.
L'interfaccia definita è fondamentale per poter rendere possibile la collaborazione tra le macchine virtuali, e permettere così ai linguaggi di MV$_i$ di sfruttare funzionalità e meccanismi di MV$_{i-1}$.
Le macchine virtuali godono delle **seguenti proprietà**:

> L'**insieme** degli oggetti o risorse **R$_i$** di MV$_i$ **è accessibile soltanto da parte dei meccanismi di L$_i$**
>
> Al livello MV$_i$ **non sono note le politiche adottate dai livelli inferiori**

Supporto a tempo di esecuzione Anche detto **Runtime Support**, è l'**insieme dei livelli sottostanti**.
Nell'esempio, MV_i ha come runtime support i livelli $MV_{i-1} \dots MV_0$.

Virtualizzazione ed Emulazione Con **virtualizzazione** o astrazione intendiamo il **processo secondo cui un livello MV_i usa funzionalità dei livelli superiori**.
Con **emulazione** o concretizzazione intendiamo il **processo secondo cui un livello MV_i usa funzionalità dei livelli inferiori**.

Modularità Tutte queste funzionalità sono **alla base della strutturazione di sistemi con elevata modularià**, **modificabilità**, **portabilità**, **manutebilità** e **testabilità**.

1.2.1 Le Macchine Virtuali

MV_4 Applicazioni
L_4: Java, C, ML...
R_4: oggetti astratti, costrutti, tipi di dato definibili dall'utente

Interfaccia: chiamate di sistema

MV_3 Sistema Operativo
L_3: C, linguaggi di programmazione concorrente, linguaggi sequenziali con librerie che implementano meccanismi di concorrenza
R_3: variabili condivise, risorse condivise, oggetti astratti usati per la cooperazione tra processi e thread

Interfaccia: istruzioni assembler

MV_2 Macchina assembler
L_2: assembler (D-RISC)
R_2: registri, memoria, canali di comunicazione

Interfaccia: istruzioni firmware per l'assembler

MV_1 Firmware
L_1: microlinguaggio
R_1: sommatore, commutatore, registri, strutture di interconnessione intra-unità e inter-unità

Interfaccia: hardware

MV_0 Hardware
L_0: *funzionamento dei circuiti elettronici*
R_0: circuiti elettronici elementari (AND, OR, NOT), collegamenti fisici, reti logiche

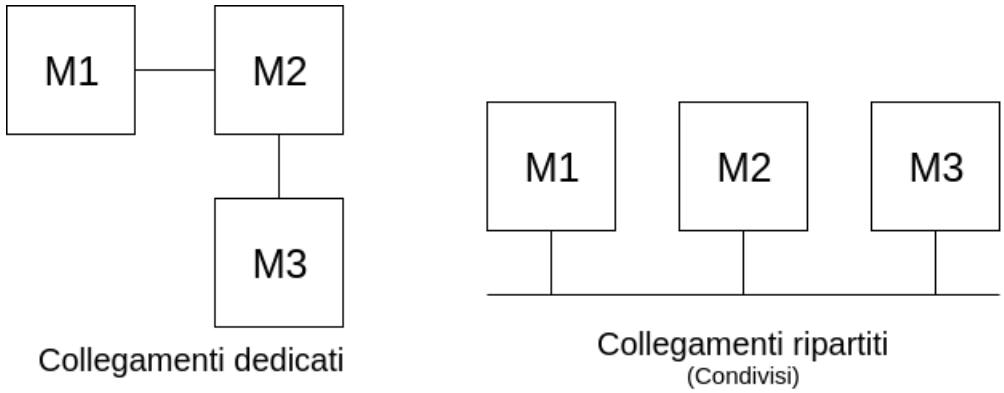
Il corso riguarderà principalmente i livelli $MV_2 \rightarrow MV_0$ inclusi, comprese le istruzioni assembler.
Il livello firmware sarà fatto da **memoria**, **processore** e **dispositivi I/O**. I dispositivi di I/O comunicano bilateralmente con la memoria e il processore comunica bilateralmente con memoria. Opzionalmente, i dispositivi di I/O comunicano bilateralmente direttamente con il processore. Questa è l'**architettura standard**, presentata in maniera **estremamente semplicistica**.
Vedremo nel dettaglio il processore e la memoria, non i dispositivi di I/O perché troppo complessi.

1.2.2 Struttura Interna

Da Verticale a Orizzontale Fin'ora abbiamo parlato delle macchine virtuali in senso **verticale**, adesso vogliamo trovare un modo concettualmente uniforme – **orizzontale** – per poter studiare i livelli **al loro interno**.

Sistema di Elaborazione Una volta scelta la struttura verticale di un sistema, dobbiamo capire come funziona l'interno di ciascun livello per poter far funzionare tutto il sistema, cioè dobbiamo capire **come funziona il sistema di elaborazione di un livello** composto da due componenti:

> **Moduli di Elaborazione**, **ad ognuno** dei quali **è affidata l'elaborazione di un sottoinsieme di operazioni** del livello.
>
> **Struttura di Interconnessione**, **con la quale i moduli** di elaborazione del livello **cooperano e comunicano tra loro**.
> Essa può essere di due tipi:



Moduli di Elaborazione Un modulo di elaborazione è definito come **un'entità autonoma e sequenziale**.

Autonomia L'autonomia è **data dal fatto che ogni modulo** di elaborazione **esegue un proprio controllo in maniera indipendente da altri moduli**.
Esso dunque **definisce le proprie strutture dati, operazioni elementari e interfacce** verso altri moduli.

Sequenzialità La sequenzialità è **data dal fatto che ogni modulo** di elaborazione **ha un singolo luogo di controllo**: **la sua attività è descritta** da un algoritmo di controllo costituito **da una lista sequenziale di comandi**.
La sequenzialità **non implica che un modulo non possa fare uso di forme di elaborazione concorrenti o parallele**. Alcuni o tutti i comandi di una lista sequenziale possono essere costituiti da una o più operazioni elementari **eseguite simultaneamente**.

1.2.3 Parallelismo

Sovrapporre Poichè i moduli sono autonomi fra loro, sono in grado di **operare indipendentemente l'uno dall'altro**. **Le loro attività possono quindi essere sovrapposte nel tempo** eccetto quando, per ragioni legate alla sincronizzazione, alcuni di loro devono attendere il verificarsi di certi eventi dipendenti dall'elaborazione di altri.
In alcuni **casi limite** seppur realistici, il **funzionamento di tutti i moduli è rigidamente sequenziale**.

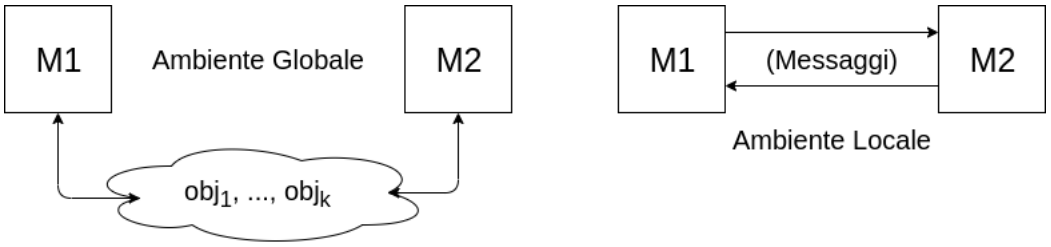
Tutto questo **vale per qualsiasi livello** o Macchina Virtuale, quindi sia per firmware che hardware.

1.2.4 Modelli di Cooperazione

Dato un sistema di elaborazione a un certo livello, **i vari moduli** presenti **possono cooperare secondo due modalità**:

Ambiente Globale: esiste un **insieme di oggetti comuni accessibili da tutti i moduli** che devono cooperare tra loro, e **tutti i moduli possono operare su tale insieme**

Ambiente Locale: **i moduli non condividono nulla**, quindi **non esiste alcun oggetto condivisio tra i moduli**. La **cooperazione avviene tramite scambio di messaggi**.



1.3 Compilazione vs Interpretazione

Programmi L'obiettivo di un calcolatore è **rendere possibile l'esecuzione di programmi** con una certa qualità di servizio. I programmi vengono **progettati mediante linguaggi di alto livello**, quindi **occorre operare una traduzione da linguaggio di alto livello a linguaggio assembler**.
Tale traduzione può essere effettuata tramite due ben note tecniche e loro combinazioni:

Compilatore: è **statico**.
Sostituisce l'intera sequenza del programma sorgente con un sequenza di istruzioni assembler. Questa traduzione viene effettuata staticamente, vale a dire in fase di preparazione e **prima che il programma passi in esecuzione**.
Uno compilatore ha **completa visione del codice** e quindi **può ottimizzarlo**. La sua attività è analoga all'opera di un traduttore, che può leggersi il testo più volte per tradurlo alla perfezione.

Interprete: è **dinamico**
Scandisce la sequenza **sostituendo ogni singolo comando** con una sequenza na di istruzioni assembler. La traduzione è effettuata dinamicamente, cioè **a tempo di esecuzione**, quindi non può ottimizzare. Il firmware riceve un'istruzione alla volta, quindi la interpreta.
Il suo svantaggio è che il **tempo di interpretazione viene pagato ogni volta che lancio il programma** e che **non può ottimizzare non avendo una visione globale** del programma.

Entrambe servono per tradurre il **codice sorgente** nel **programma oggetto** o **eseguibile**. L'esecuzione è quindi **più veloce in un programma compilato** rispetto ad un programma interpretato.

`ADD R1, R2, R3` $\longrightarrow$ *compilatore* $\longrightarrow$ **OBJ** $\longrightarrow$ Interprete Firmware (interfaccia tra MV ASM e MV FW)

Intuitivamente, dall'istruzione ad alto livello viene **compilato un programma oggetto OBJ** il quale è un insieme di bit che **viene interpretato dall'interprete firmware**.

Esempio **Suppongo programmi**:

A

```
for i=0; i++; i<n
    A[i] = A[i] + B[i];
```

B

```
for i=0; i++; i<n
    B[i] = B[i] + C;
```

Ricevendo i due blocchi di istruzioni, il **compilatore riconosce che sono diverse e le compila in modo diverso**. Però in entrambi i casi sono del tipo *oggetto = somma due oggetti*, quindi produce una sequenza di istruzioni analoga (a meno di registri e dati, ovviamente). Parte del secondo pezzo di codice, ad esempio, verrà tradotto in questa maniera:

`LOAD` R_{base}`,` R_I`,` R_1	`M[R[base] + R[I]]` $\rightarrow$ `R[1]`
`ADD` R_1`,` R_2`,` R_1	`R[1] + R[2]` $\rightarrow$ `R[1]`
`STORE` R_{base}`,` R_I`,` R_1	`R[1]` $\rightarrow$ `M[R[base] + R[I]]`
`INC` R_I	`R[I] + 1` $\rightarrow$ `R[I]`
`IF<` R_I`,` R_N`, LOOP`	**Microlinguaggio corrispondente**

Capitolo 2

MV0 – Hardware

2.1 Reti Logiche

L'**implementazione a livello hardware di funzioni "pure"** dà luogo alle **Reti Combinatorie**.
L'**implementazione** a livello hardware **di funzioni "con stato"** dà luogo alle **Reti Sequenziali**.

Famiglia Entrambe definiscono la famiglia delle **Reti Logiche** che permettono di **realizzare il livello hardware di un sistema di elaborazione**.

2.2 Reti Combinatorie

Una **rete combinatoria è** una rete logica **con `n` ingressi binari $X_1 \dots X_n$ e `m` uscite binarie $Z_1 \dots Z_m$**. Ad **ogni combinazione di valori in entrata corrisponde una ed una sola combinazione di valori in uscita**. La corrispondenza è definita secondo la funzione implementata dalla rete combinatoria.
Indichiamo $X_1 \dots X_n$ e $Z_1 \dots Z_m$ come **variabili logiche** di ingresso ed uscita. **Tutte le combinazioni possibili** delle variabili logiche **sono dette stati** di ingresso – con 2^n possibilità – e di uscita – 2^m possibilità.

Pe rdescrivere le proprietà e la struttura interna delle reti combinatorie si usa un'**algebra isomorfa a quella logica**, chiama **Algebra Booleana**.

2.2.1 Algebra Booleana

L'algebra booleana è computata su **due valori** e **tre operatori**:

`false`	`AND`
`true`	`OR`
	`NOT`

Esistono anche altri operatori, derivati dai tre precedenti: `XOR`, `NAND`, `NOR` ecc..

Proprietà Vale la proprietà distributiva anche per la somma rispetto alla moltiplicazione, oltre il viceversa, quindi: `A(B+C) = AB + AC`, ma anche `A + BC = (A + B)(A + C)`.
Inoltre si hanno le cosiddette **proprietà di DeMorgan**:

- $\overline{A+B}$ `=` $\overline{A}$ `*` $\overline{B}$
- $\overline{AB}$ `=` $\overline{A}$ `+` $\overline{B}$

AND

Anche detta **moltiplicazione logica**.

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

Anche detta **somma logica**.

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Anche detta **negazione logica**.

Y	Z
0	1
1	0

Per costruire una **rete combinatoria** esistono varie tecniche. Quella che useremo si chiama **somma di prodotti**.

2.2.2 Tecnica della Somma di Prodotti, o codifica degli 1

La tecnica nel dettaglio Partendo dalla **tabella di verità**, identifico le uscite che valgono 1. Di quelle uscite, **moltiplico (`AND`)** tra loro le entrate **sulla stessa riga**, **nego le entrate che valgono 0** e **sommo (`OR`) tra loro le diverse righe**.

Un esempio con la somma algebrica Partendo dalla seguente tabella di verità.

X	Y	Z	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sfruttando la tecnica descritta sopra ottengo le seguenti espressioni per le due uscite:

`Z =` $\overline{X}$ `* Y + X *` $\overline{Y}$

`R = X * Y`

Alternativamente, posso anche realizzare la **funzione complementare**, ovver fare il solito procedimento ma per le uscite che valgono 0 per poi negarle.

X	Y	$\overline{Z}$	R
0	0	1	0
0	1	0	0
1	0	0	0
1	1	1	1

`Z =` $\overline{\overline{X} * \overline{Y} + X * Y}$

`R = X * Y`

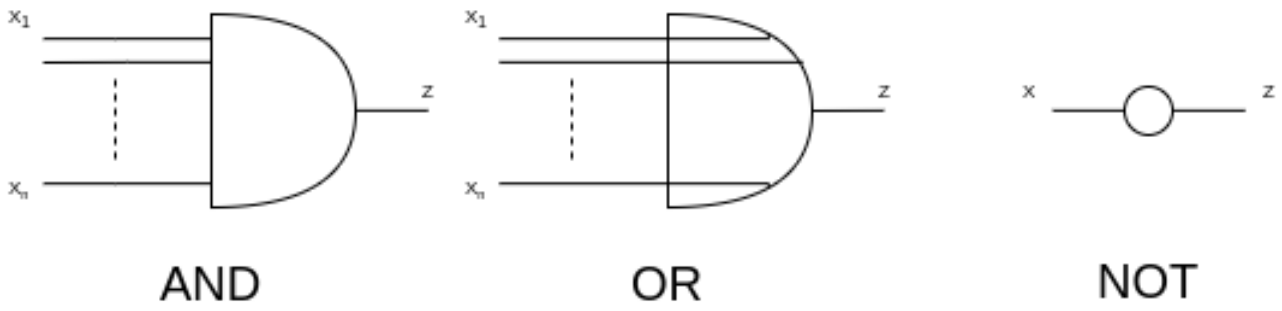
Esempio

S1	S2	X	Y	S1*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

S1* = $\overline{S1}$*$\overline{S2}$*X*Y + $\overline{S1}$*S2*X*Y + S1*$\overline{S2}$*$\overline{X}$*$\overline{Y}$ + S1*$\overline{S2}$*X*$\overline{Y}$ + S1*$\overline{S2}$*X*Y + S1*S2*$\overline{X}$*Y + S1*S2*X*$\overline{Y}$ + S1*S2*X*Y

2.2.3 Porte Logiche

Una volta ricavata l'espressione logica dalla tabella di verità, è **immediato realizzare lo schema logico utilizzando le componenti hardware elementari**, dette anche **porte logiche**:

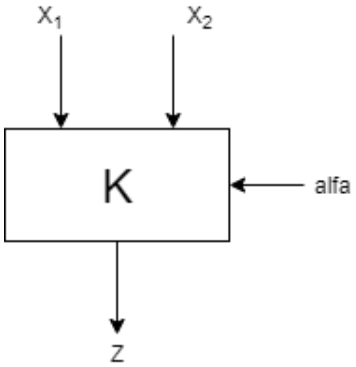


Ogni porta logica AND e OR **comporta un ritardo nel calcolo di 1 $\mathbf{T}_p$**. Inoltre, ogni AND e OR può avere **massimo 8 ingressi**, quindi se ho più di 8 segnali in ingresso devo avere *almeno* due livelli: un livello con tante porte logiche quando n/8 con n numero di segnali in ingresso, e *almeno* un livello in cui "unire" i segnali in uscita in una porta logica analoga.

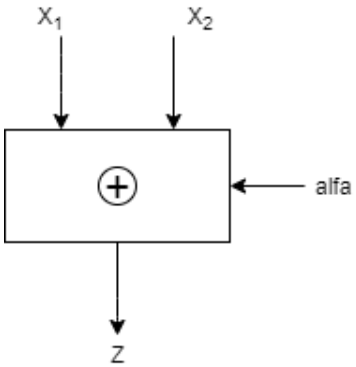
2.2.4 Componenti Standard

Di seguito sono le specifiche di alcune **reti combinatorie** che verranno supposte come **standard**, ovvero come componenti utilizzabili come blocchi elementari nella progettazione di strutture più complesse.

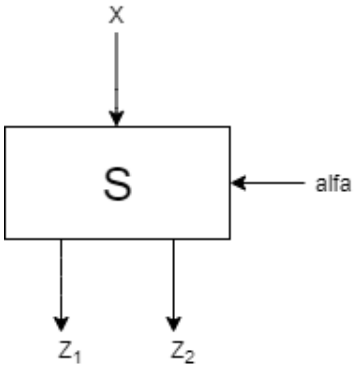
Commutatore



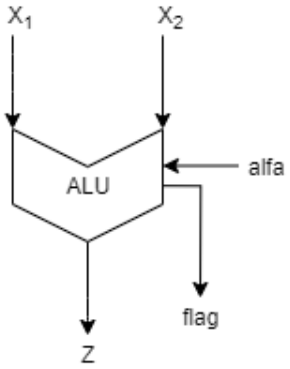
Selezionatore



Confrontatore



ALU



2.2.5 Ritardo di Stabilizzazione

Prestazioni Per valutare le prestazioni di un sistema, occorre saper **valutare le prestazioni delle reti combinatorie**. Ogni rete reale è **caratterizzata da un ritardo $\mathbf{T}_r$**, necessario affinché **a seguito di una variazione dello stato d'ingresso si produca la corrispondente variazione dello stato in uscita**.
Solo dopo questo tempo si dice che **la rete è stabilizzata**.

$\mathbf{T}_p$ Per una porta logica indichiamo con T_p **il ritardo di stabilizzazione** – ad oggi è di circa 10^{-2} millisecondi.
Supponiamo che le **porte NOT** abbiano un **ritardo nullo**, pari a 0 T_p, mentre per le **porte AND/OR** il valore T_p dipende dal numero di ingressi n della porta. Per $n \leq 8$ supponiamo che le porte AND/OR abbiano un **ritardo di stabilizzazione di 1 $\mathbf{T}_p$**.
Il costo in T_p sarà quindi pari ai livelli di AND/OR presenti. Ad esempio, se ho una tabella di verità con n termini ed m variabili, avrò $\log_8 n$ livelli di OR e $\log_8 m$ livelli di AND. Il costo in T_p sarà quindi = $(\log_8 n + \log_8 m)$ T_p

2.2.6 Registri e memorie

2.3 Reti Sequenziali

Una **rete sequenziale** è un oggetto con **un ingresso ed una uscita**, capace di **mantenere uno stato interno** – ecco perché si parla di funzioni con stato. A livello hardware, possiamo identificare una rete sequenziale con un **automa a stati finiti**.

ASF Un **automa a stati finiti** è caratterizzato da:

n variabili di ingresso $\Rightarrow$ h = 2^n **stati di ingresso** $X_1 \dots X_h$

m variabili di uscita $\Rightarrow$ k = 2^m **stati di uscita** $Z_1 \dots Z_k$

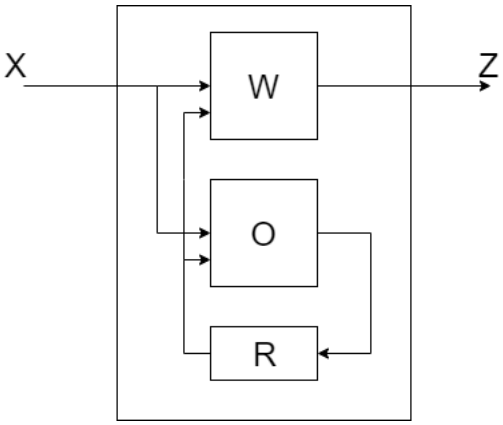
r variabili logiche dello stato interno $\Rightarrow$ p = 2^r **stati interni** $S_1 \dots S_p$

una **funzione di transizione** dello stato interno σ: X x S $\rightarrow$ S che **definisce il passaggio tra gli stati**

una **funzione delle uscite** ω: X x S $\rightarrow$ Z che **calcola le uscite**

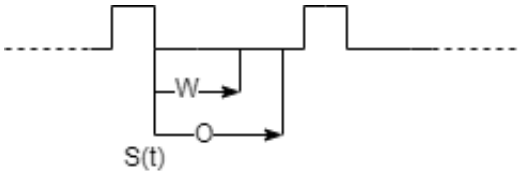
Una **rete sequenziale** è quindi **composta da due reti combinatorie** σ **e** ω, che rispettivamente calcolano la variazione dello stato e l'uscita, **e da un registro R** che contiene lo stato interno.

2.3.1 Modello di Mealy

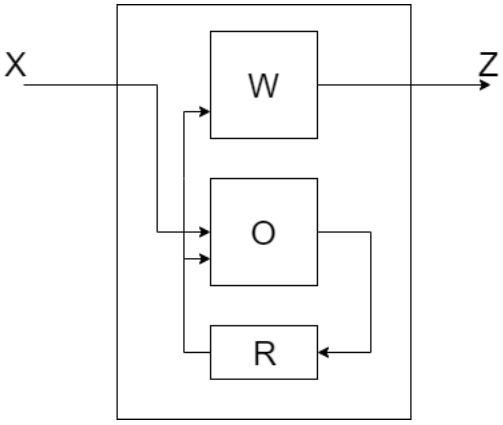


Considerando il comportamento al tempo t, lo **stato interno successivo S(t+1)** dipende sia dallo stato di ingresso al tempo t, cioè X(t), sia dallo stato interno attuale S(t).
`S(t+1) = σ(X(t), S(t))`

Lo **stato di uscita al tempo t, Z(t)**, dipende sia dallo stato di ingresso X(t) sia dallo stato interno attuale S(t).
`Z(t) = ω(X(t), S(t))`

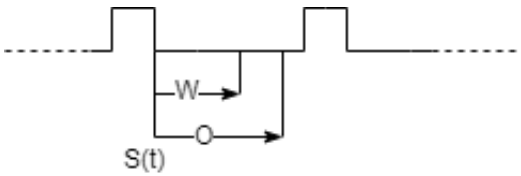


2.3.2 Modello di Moore



In maniera analoga al modello di Mealy, lo **stato interno successivo S(t+1)** dipende sia dallo stato di ingresso al tempo t, cioè X(t), sia dallo stato interno attuale S(t).
`S(t+1) = σ(X(t), S(t))`

Lo **stato di uscita al tempo t, Z(t)**, dipende solo dallo stato interno attuale S(t).
`Z(t) = ω(S(t))`

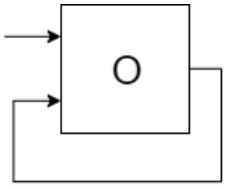


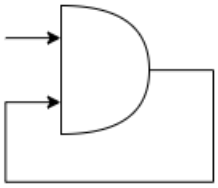
2.3.3 Reti Sequenziali di tipo Sincrono

Vediamo adesso come si comportano nel tempo le reti sequenziali e spieghiamo perché adotteremo quelle di tipo sincrono. Come riferimento usiamo una rete di Mealy.

Spezzare Abbiamo detto che lo stato al tempo successivo S(t + 1) dipende sia dall'ingresso X sia dallo stato interno attuale S(t), cioè `S(t+1) = σ(X(t), S(t))`.
Il registro R funge come un "cancello temporizzato" che **spezza la sequenza temporale degli eventi**.

Se il registro R non fosse presente, si verificherebbe la situazione in figura. In questo esempio, la porta logica o il componente σ **potrebbero non stabilizzarsi mai**.
Se per esempio mettiamo una porta AND con due variabili in ingresso che nega il proprio risultato, tale rete tenderà a non stabilizzarsi mai ma a produrre una sequenza infinita di 0 e 1 in uscita.
Quindi devo avere necessariamente **un meccanismo che mi possa aiutare a determinare il valore dell'uscita** al tempo t, t + 1...



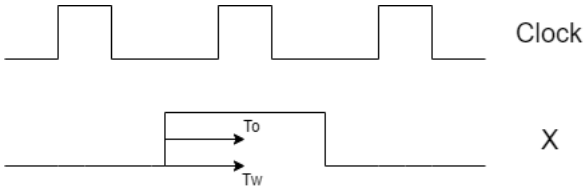


Questo strumento è il resgistro impulsato, dove **la scrittura è scandita dal ciclo di clock**.

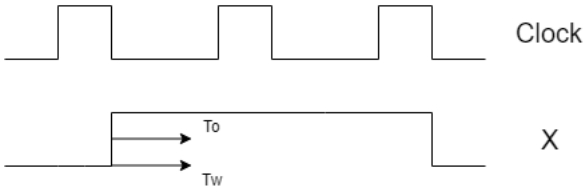
Modo Sincrono Questo modo di lavorare delle reti sequenziali con un registro impulsato che funge da cancello temporizzato grazie al ciclo di clock si chiama **Modo Sincrono**.

Quando variare Cerchiamo ora di capire quando devono variare gli ingressi e **per quanto tempo devono avere tale valore**.
Supponiamo di avere gli ingressi $X_0 = 0$ al tempo t, $X_1 = 1$ al tempo t + 1 e $X_2 = 0$ al tempo t + 2, e supponiamo che $t_\omega = t_\sigma = 2t$.

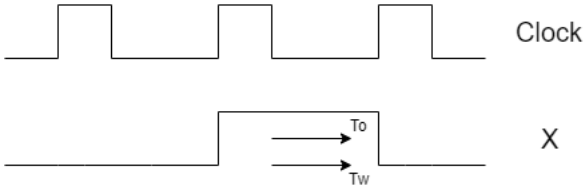
Se l'ingresso X variasse in un punto non precisato del ciclo di clock è probabile che ω e σ non abbiano il **tempo necessario per produrre un risultato** e quindi avrei un **comportamento indefinito**.



In questo caso, cambiando il valore X all'inizio del ciclo di clock do il tempo necessario a σ e ω di produrre un risultato stabile, ma al prossimo impulso del ciclo di clock (t + 2) leggerò di nuovo X = 1, che non è l'input corretto al tempo t + 2.



Questa è la soluzione giusta per il nostro esempio, che rispetta tutte le condizioni da noi elencate.



Per far funzionare le nostre reti, il ciclo di clock deve essere tale che $T = MAX(t_\sigma, t_\omega) + \delta$. Le reti funzionano anche con $T > MAX(t_\sigma, t_\omega) + \delta$, ma avrei del **tempo perso** poiché la rete non opera, **aspetta solo che il clock sia alto per andare a scrivere nel registro**.

2.3.4 Reti Sequenziali a Componenti Standard

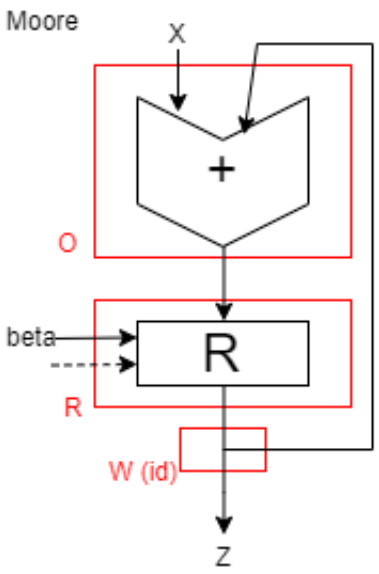
Per poter sintetizzare una rete devo prima **decidere** se **implementare un modello di Mealy o di Moore**, **derivare le tabelle di verità** di ω e σ e dire **quanti bit** ha il registro R.
Fatto questo, **ricavare le reti combinatorie** e **capire quanto valga il ciclo di clock** T (con $T = MAX(t_\sigma, t_\omega) + \delta$) che fa funzionare l'intera rete sequenziale.

Sintesi Classica ASF $\longrightarrow$ Mealy o Moore $\rightarrow$ Tabelle verità, bit di R $\Rightarrow$ Reti combinatorie $\Rightarrow$ Ciclo di clock

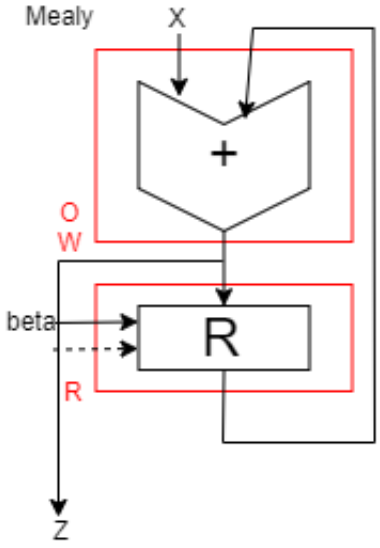
Componenti Standard In realtà per sintetizzare le reti sequenziali non usiamo questo procedimento di sintesi, ma bensì **usiamo i componenti standard**. Per esempio, prendiamo una rete che vuole calcolare il numero di persone presenti dentro una stanza con capienza massima 100 persone.

Con la sintesi classica R ha bisogno di 7 bit per contare da 0 a 100.
Se andiamo, per esempio, a fare la tabella di verità per ω, abbiamo ben 8 colonne negli ingressi, quindi 2^8 possibili combinazioni (righe).
Potrei avere $2^8/2 = 2^7$ "uni" per colonna, di conseguenza un **numero considerevole di porte logiche**.
Diventa quindi praticamente impossibile sintetizzare questo esempio con il metodo classico. Procediamo con l'alternativa: l'utilizzo delle componenti standard.

Con le componenti standard Procediamo col nostro esempio:

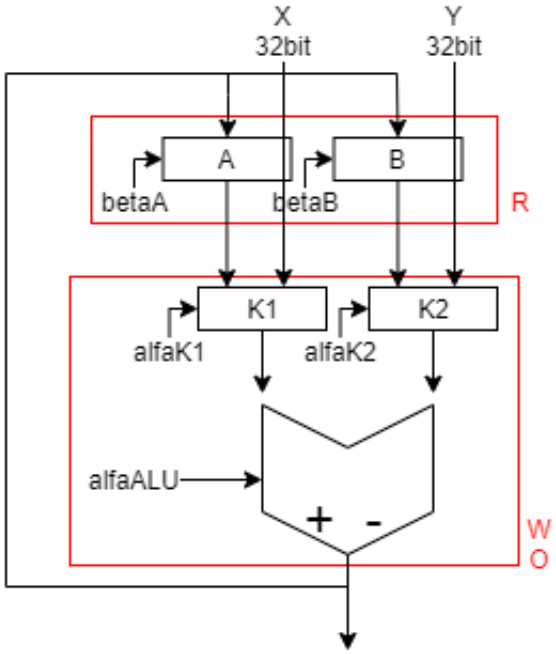


In questo caso abbiamo usato il modello di Moore. Il **risultato è disponibile al prossimo impulso del ciclo di clock**.



Qua invece è stato usato il modello di Mealy. In questo caso si vede bene come **la rete di Mealy sia più veloce**, poiché **il risultato è subito disponibile** prima del prossimo impulso del ciclo di clock: infatti Z non viene scritto in R prima di essere pubblicato.

Di seguito un esempio di rete sequenziale a componenti standard più complesso.



Con sintesi classica Avrei:

R = {A, B}, due registri da 32 bit ⇒ 64 bit

Ingressi: X + Y + α_{K1} + α_{K2} + α_{ALU} + β_A + β_B = 32 + 32 + 3 + 2 = 69

Uscite: Z ⇒ 32 bit

La tabella di verità di ω, per esempio, avrebbe 69 colonne di ingressi, quindi 2^{69} righe, **senza considerare gli ingressi di A e B**.
Il risultato è che è molto scomodo lavorare con una tabella di circa 5.9 * 10^{20} righe.

Capitolo 3

MV1 – Firmware

3.1 Unità Firmware

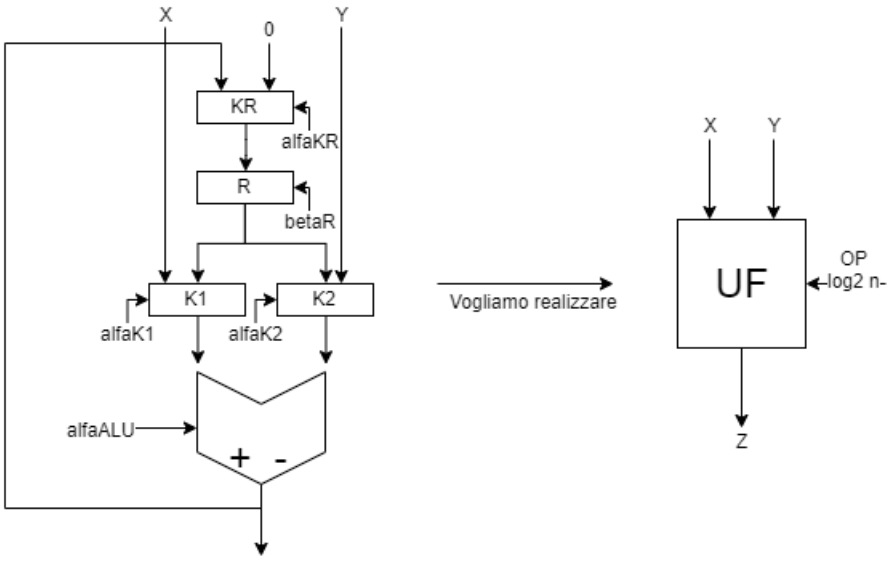
Un sistema di elaborazione, a livello Firmware, è costituito da un certo numero di **Unità Firmware** che interagiscono fra loro mediante un sistema di interconnessione. Le UF sono capaci si svolgere un certo numero di operazioni esterne.

Unità Firmware Una **unità firmware** è un **modulo di elaborazione autonomo** – cioè capace di controllare la propria operazione in modo del tutto indipendente – **e sequenziale** – cioè dal funzionamento descritto da un programma sequenziale – **capace di eseguire delle operazioni esterne** – istruzioni **assembler**.

Struttura di interconnessione Tipicamente la struttura di interconnessione tra unità firmware è **punto-a-punto** quindi a **collegamenti dedicati**.

3.1.1 PC e PO

Per capire bene cosa sono e a cosa servono le parti controllo (PC) e operativa (PO), vediamo un semplice esempio di come arriviamo a strutturare un'unità firmware.

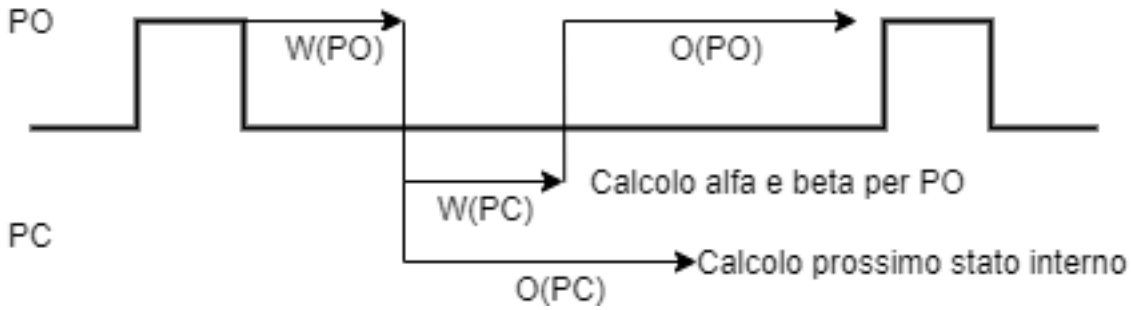


Il nostro obiettivo è quello di realizzare una unità capace di produrre un risultato Z a partire dalle variabili in input, **fornendogli solo le istruzioni per l'operazione da implementare** e **capace di gestire tutte le variabili di controllo** (α, β) **in maniera autonoma**.
L'unità firmware è quindi l'unione di due oggetti:

Parte Controllo che "comanda" l'operazione da eseguire

Parte Operativa che "esegue" l'operazione

Ciclo di Clock



PO e PC sono**reti sequenziali impulsate** dallo stesso segnale di clock, quindi aventi lo stesso ciclo di clock. Il **ciclo di clock dell'unità firmware** viene determinato in modo da **permettere la stabilizzazione di entrambe le reti per l'esecuzione di una qualsiasi microisitruzione**.
Questo modello di programmazione è **sincrono**.

Schematizzazione del diagramma del ciclo di clock sopra:

ω_{PO}: prepara i valori che servono alla PC per decidere cosa fare

ω_{PC}: prepara $\{\alpha, \beta\}$ che implementano l'operazione richiesta

σ_{PC}: decido il prossimo stato interno della PC $\rightarrow$ scrivo il registro R della PC

σ_{PO}: eseguo l'operazione pianificata $\rightarrow$ scrivo i nuovi valori nei registri che compongono lo stato interno della PO

Parte Operativa, Moore

Rete Sequenziale progettata con **componenti standard** che provvede all'esecuzione di **istruzioni** tramite commutatori, selettori, ALU e registri.

Parte Controllo, Mealy

Rete Sequenziale progettata tramite **sintesi classica** che provvede a determinare le **variabili di controllo** α e β per la parte operativa.

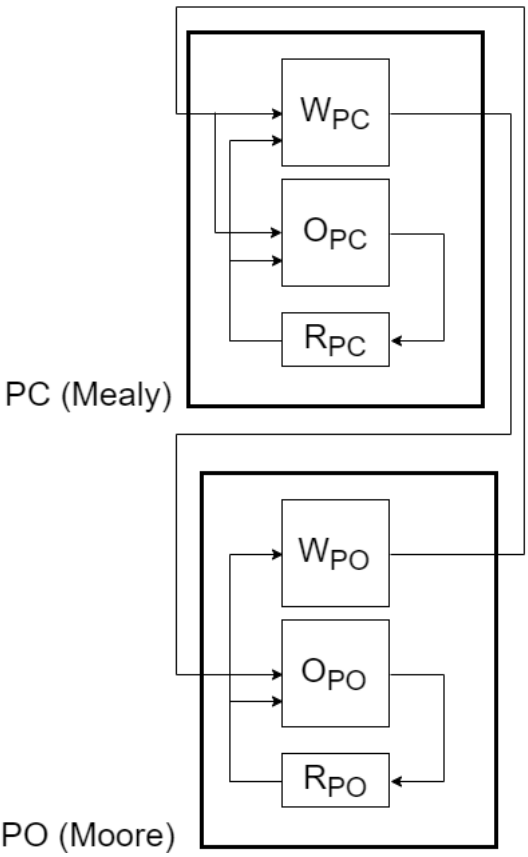
Mealy o Moore?

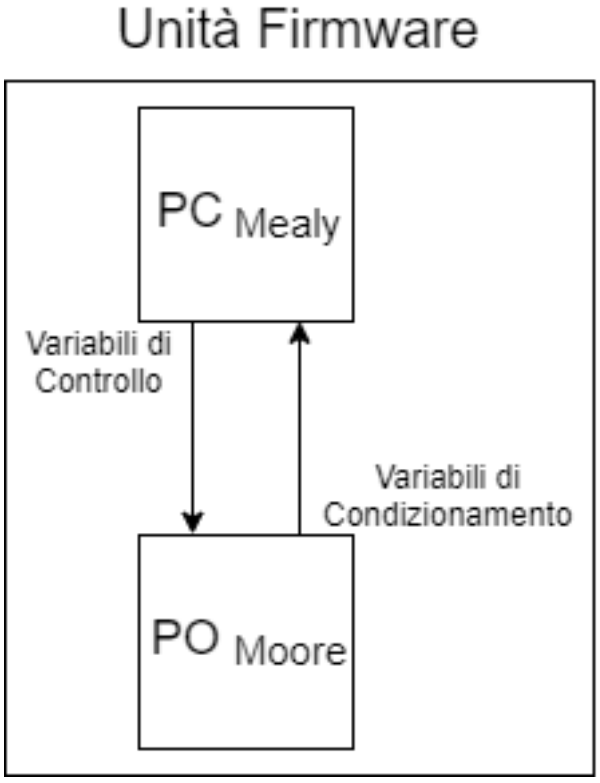
Essendo entrambe due reti sequenziali bisogna decidere quale modello usare. Analizziamo le varie combinazioni di modelli.

Mealy–Mealy Se uso un modello Mealy–Mealy le uscite di ω_{PO} vanno direttamente nella ω_{PC} e le uscite di ω_{PC} ritornano in ω_{PO}. Non ho un registro che ferma il ciclo continuo tra ω_{PC} e ω_{PO}, quindi non riuscirò mai a stabilizzare i segnali che si scambiano PO e PC.
Viene naturale pensare di farle entrambe Mealy–Mealy poiché, come visto in precedenza, il modello di Mealy è più veloce di quello di Moore.

Almeno una Moore Concludiamo che almeno una tra PC e PO deve essere di Moore per poter stabilizzare l'intera UF, ma quale?
La risposta corretta è usare il **modello di Mealy per la Parte Controllo** e il **modello di Moore per la Parte Operativa** in modo da avere dei **comandi veloci** ed una **esecuzione più lenta** rispetto alla PC.

Il contrario? Non scegliamo un modello Moore per PC e Mealy per PO perché è **illogico avere un esecutore veloce che deve aspettare un controllore lento**: inutile avere una macchina molto veloce se inserisco i comandi molto lentamente.
Anche un modello Moore–Moore non è comodo da usare, seppure funzionante correttamente, perché avrei entrambe le parti lente.





Condizione di Correttezza $\Rightarrow$ PO di Moore
Le uscite della Parte Operativa, cioè le variabili di condizionamento, dipendono esclusivamente dallo stato interno di PO, cioè **tutte le variabili di condizionamento devono essere prodotte senza usare alcun** α

3.1.2 Procedimento Formale

Schematizzazione dei passaggi del procedimento formale per la costruzione e l'analisi di una rete sequenziale.

1. Descrizione a parole delle operazioni esterne
2. Programma scritto in μ-linguaggio
3. Componenti

   R_{PO}: capire quali sono i registri di stato della PO

   ω_{PO}, σ_{PO}: capire le funzioni che mi servono nella PO

   R_{PC}: capire cosa è lo stato della PC

   ω_{PC}, σ_{PC}: capire cosa calcolare nella PC

   $\longrightarrow T = t(\omega_{PO}) + MAX\{t(\omega_{PC}) + t(\sigma_{PO}), t(\omega_{PC})\} + \delta$

Capitolo 4

μ-linguaggio

Si formalizza un linguaggio chiamato **μ-linguaggio** che permetta di **derivare formalmente com'è fatta la PC e la PO di una certa UF**.

4.1 Istruzioni

Nel μ-linguaggio sono presenti solamente **due tipi di istruzione**:

> `n.` μ`op`$_1$`, ...,` μ`op`$_k$`, m`
> Le `op` sono **operazioni di trasferimento tra registri**. Le varie `op` **separate da una virgola sono eseguite contemporaneamente** – cioè nello stesso ciclo di clock.
> `m` finale indica **a quale istruzione andare dopo aver eseguito questa istruzione**, la `n`.

> `n. (condizione = T)` μ`op`$_1$`, ...,` μ`op`$_k$`, m'`
> `(condizione = F)` μ`op`$_1$`, ...,` μ`op`$_h$`, m''`
> Le **condizioni** sono **date in termini di variabili di condizionamento**. Possono essere messe in sequenza, venendo **valutate in sequenza**.
> Posso considerare solo **variabili booleane** o **espressioni di cui mi interessa solo il risultato** senza memorizzarlo.

Esempio Vediamo un esempio di come scrivere un programma in μ-linguaggio. Prendiamo come esempio la divisione fra interi.

Linguaggio pseudo-C

```
Q = 0
while (A >= B) {
    Q = Q + 1
    A = A - B
}
R = A
```

μ-linguaggio

```
0.  0 -> Q, 1
1.  (segno(A - B) = 0) nop, 2
    (= 1)              nop, 4
2.  Q + 1 -> Q, 3
3.  A - B -> A, 1
4.  A -> R, 0
```

Ogni μ-istruzione è **eseguita esattamente in un ciclo di clock**. Nell'esempio, per eseguire il programma avrò bisogno di almeno 5 cicli di clock, a meno di iterazioni interne.

4.2 Ottimizzazione del codice

Dopo aver scritto il μ-codice possiamo provare ad **ottimizzarlo**, cioè **ridurre il numero di cicli di clock necessari ad eseguirlo**.
Un'ottimizzazione possibile dell'esempio precedente è la seguente.

Prima

```
0.  0 -> Q, 1
1.  (segno(A - B) = 0) nop, 2
    (= 1)              nop, 4
2.  Q + 1 -> Q, 3
3.  A - B -> A, 1
4.  A -> R, 0
```

Dopo

```
0.  0 -> Q, A - B -> T, A - B -> A, 1
1.  (T0 = 0) Q + 1 -> Q, A - B -> A, A - B -> T, 1
    (= 1)    A -> R, 0
```

Considero un registro T dove memorizzo il risultato di A - B. Di quel registro T, considero T_0 – cioè il bit più significativo – per il segno.
Inoltre elimino le `nop`, che sono *tempo sprecato*.

4.2.1 Condizioni di Bernstein

Per eseguire le ottimizzazioni sul μ-codice, dobbiamo **seguire le Condizioni di Bernstein**. Tali condizioni forniscono delle regole per verificare se due o più μ-operazioni possono essere eseguite nella medesima μ-istruzione.

Le Condizioni Per capire se
`i.` μop_A`, i+1`
`i+1.` μop_B`, k`
è equivalente a
`i.` μop_A`,` μop_B`, k`
Bisogna **valutare il dominio R(op)** – registri **letti** da op – e **il codominio W(op)** – registri **scritti** da op – delle μ-operazioni.
Nell'esempio precedente:

$R(A - B \rightarrow T) = \{A, B\}$

$W(A - B \rightarrow T) = \{T\}$

$R(Q + 1 \rightarrow Q) = \{Q\}$

$W(Q + 1 \rightarrow Q) = \{Q\}$

Le **condizioni da verificare** sono:

$W(\mu op_A) \cap R(\mu op_B) = \emptyset$
Dipendenza: non posso mettere insieme μ-operazioni tali che la prima scrive in un registro letto dalla seconda.

$W(\mu op_A) \cap W(\mu op_B) = \emptyset$
Dipendenza di output: non posso scrivere nello stesso registro con due μ-operazioni diverse nella stessa μ-istruzione.

4.2.2 Variabili di Condizionamento

Le **variabili di condizionamento** possono essere così categorizzate:

Semplici: indicano le uscite di registri **senza trasformazioni**
$\longrightarrow t_{\omega PO} = 0$

Complesse: indicano **trasformazioni delle uscite di registri** fatte tramite reti combinatorie prive di ingressi di controllo.
Esempio: `segno(A - B)`, `OR(A)`
$\longrightarrow t_{\omega PO} = k\ t_p$

4.2.3 Tempo medio di elaborazione

Il **tempo medio di elaborazione** di una UF viene valutato come: $T = \sum_{i=0}^{n-1}(p_i * k_i)$
Dove:

k_i è il numero medio di cicli di clock necessari per eseguire una generica operazione i

p_i è la probabilità di eseguire tale operazione

Quando non sono note le p_i, si assume che tutte le sottosequenze siano equiprobabili. Calcoliamo quindi T come media aritmetica dei k_i, oppure si cerca di stimare se possibile una distribuzione probabilistica attendibile.

4.2.4 Riflessioni finali sull'ottimizzazione

Bisogna prestare particolare attenzione quando si ottimizza il μ-codice. Ridurre il numero di μ-istruzioni (k_i) non è sempre qualcosa di buono.
Talvolta, **unire due o più μ-istruzioni obbliga ad aumentare il ciclo di clock T** per consentire al μ-programma di eseguirle tutte. Questa modifica, che si applica a **tutte** le μ-istruzioni, potrebbe aumentare il tempo medio di elaborazione T, rendendo il programma complessivamente più lento.

Concludendo, le ottimizzazioni che si possono fare sono:

Eliminare le `nop`, tranne quelle di attesa per operazioni esterne

Raggruppare le μ-operazioni, attraverso le condizioni di Bernstein

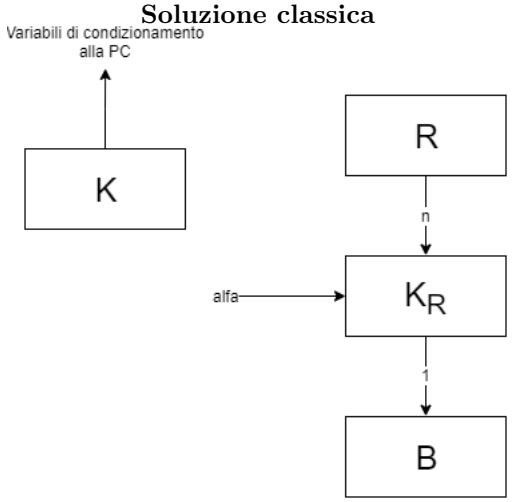
Raggruppare le condizioni logiche

4.3 Controllo Residuo

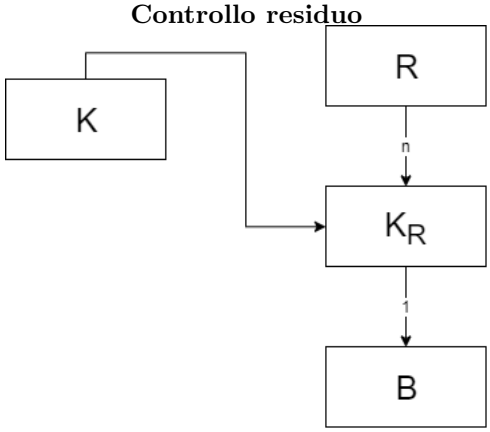
Per diminuire ulteriormente la complessità della PC possiamo **delegare alla PO alcune delle decisioni che dovrebbe prendere la PC**.
Vediamo alcuni esempi:

Leggere il k-esimo bit di un registro R di n bit

Soluzione classica



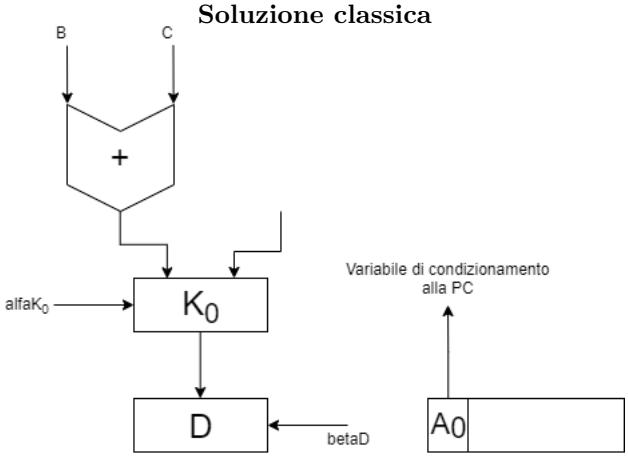
Controllo residuo



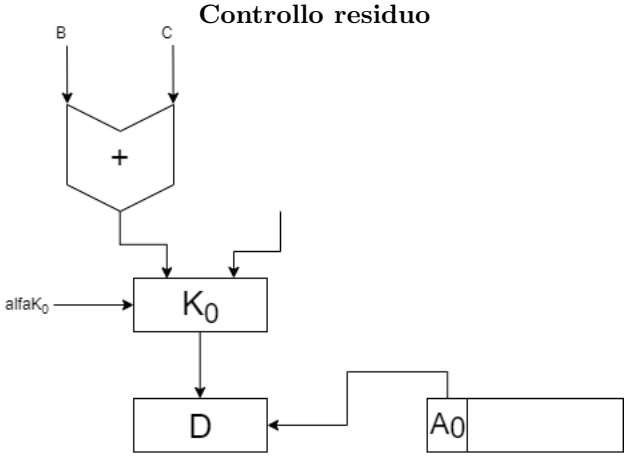
Risparmio complessità della PC e riduco il traffico di dati da PO a PC

Supponiamo una μ-istruzione da eseguire a seconda di una certa condizione, ad esempio: `(A0 = 0) B + C → D`

Soluzione classica

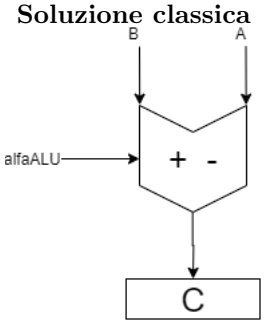


Controllo residuo

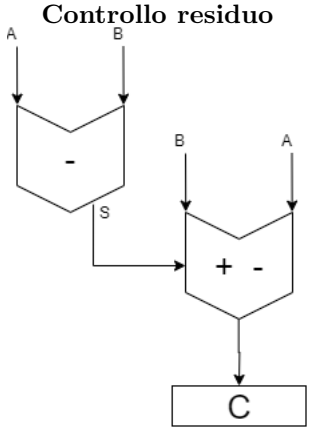


`(segno(A - B) = 0) B - A → C`
`(= 1) B + A → C`

Soluzione classica



Controllo residuo



4.4 Comunicazioni

Con **comunicazioni** si intendono le **comunicazioni fra unità firmware e mondo esterno** e viceversa.
Nell'esempio preso in esame, della divisione fra A e B interi con Q ed R risultati, **A e B provengono dal mondo esterno e Q ed R sono comunicati verso di esso**.

$$\text{A,B} \longrightarrow \text{UF} \longrightarrow \text{Q,R}$$

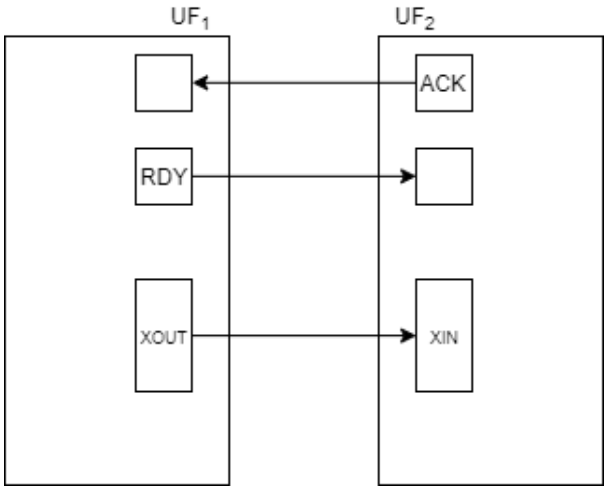
Categorie Le comunicazioni sono **classificate in due categorie**:

Simmetriche/Asimmetriche
Simmetriche: un solo mittente, un solo destinatario (**uno-a-uno**)
Asimmetriche: asimmetria in ingresso (**più mittenti**) o in uscita (**più destinatari**)

Sincrone/Asincrone
Sincrone: la comunicazione avviene "**istantaneamente**"
Asincrone: il destinatario legge il messaggio **dopo del tempo** (es. e-mail)

4.4.1 Protocollo a Livelli

Simmetrico e asincrono, il protocollo a livelli **funziona aggiungendo ai registri** XOUT di UF_1 e XIN di UF_2 **altri due registri da 1 bit ciascuno**, che **indicano quando avviene la comunicazione**: **ACK e RDY**



Di seguito i passi del funzionamento del protocollo:

1. UF_1 scrive XOUT e il primo registro da 1 bit

Situazione iniziale

0		0
1	$\longrightarrow$	0

Situazione finale

0		0
1	$\longrightarrow$	1

1 in RDY di UF_2 significa che ci sono dati significativi in XIN

2. UF_2 utilizza XIN e comunica che ha finito scrivendo nel proprio registro di OUT da 1 bit

Situazione iniziale

0	$\longleftarrow$	1
1	$\longrightarrow$	0

Situazione finale

1	$\longleftarrow$	1
1	$\longrightarrow$	1

3/4. Ritorno alla situazione iniziale con tutti i registri da un bit a 0

1	$\longleftarrow$	1	1	$\longleftarrow$	1	1	$\longleftarrow$	0	0	$\longleftarrow$	0
0	$\longrightarrow$	1	0	$\longrightarrow$	0	0	$\longrightarrow$	0	0	$\longrightarrow$	0

Si può riiniziare

Vediamo i cicli di clock necessari per usare questo protocollo:

1. UF_1 scrive 1
2. UF_2 vede 1 $\rightarrow \ldots$ UF_2 agisce$\ldots \rightarrow$ UF_2 scrive 1
3. UF_1 vede 1 di ritorno $\rightarrow$ UF_1 scrive 0
4. UF_2 vede 0 $\rightarrow$ UF_2 scrive 0
   $\Rightarrow$ Condizioni iniziali: **4 cicli di clock**

Se le due UF hanno clock sfasati uso lo stesso ragionamento, probabilmente finendo per dover usare più cicli di clock.

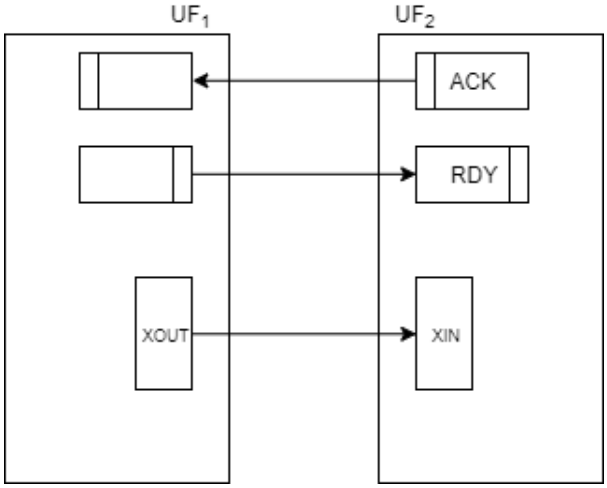
Nel programma Come rendere questo meccanismo nel μ-codice? Proviamo a scriverlo per UF_2:

```
0.  (RDY = 0) nop, 0
    (= 1)     A -> TEMPA, B -> TEMPB, 1 -> ACK, 1
1.  (RDY = 1) nop, 0
    (= 0)     0 -> ACK, <proseguo con altro>
```

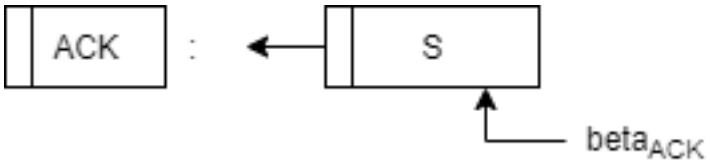
Questo protocollo è particolarmente semplice e necessita **pochissimo hardware**, ma **richiede troppi cicli di clock per comunicare**. Vediamo un'alternativa migliore.

4.4.2 Protocollo a Transizione di Livello

Simmetrico e asincrono, simile al protocollo a livelli ma **usa degli indicatori di transizione di livello**.

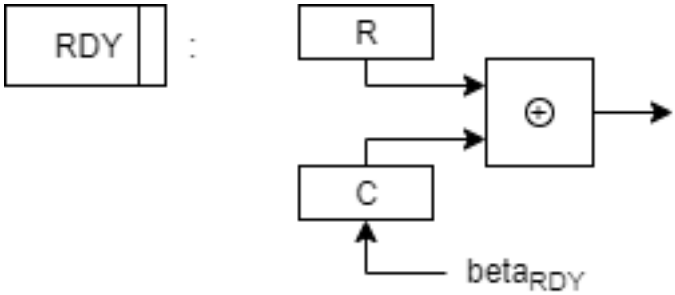


ACK **Contatore in modulo 2**, quindi cambia stato ($0 \leftrightarrow 1$) ogni volta che ci scrivo.



S	β	S'
0	0	0
0	1	1
1	0	1
1	1	0

RDY Risultato di un confrontatore fra un contatore modulo 2 e un registro in ingresso.

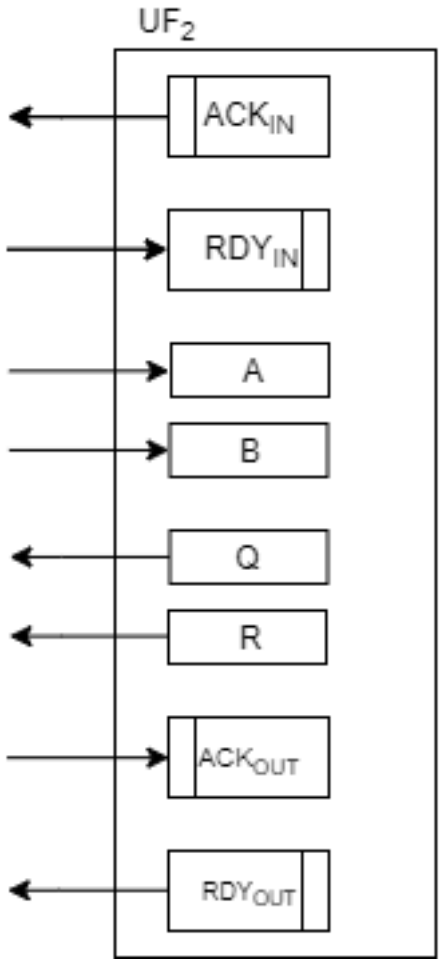


Funzionamento Dal punto di vista di UF_2.
Quando arriva un segnale da UF_1, la rete RDY diventa 1, quindi si può lavorare con i dati ricevuti.
Per comunicare a UF_1 che l'operazione è conclusa, mando un $\beta_{ACK} = 1$ al mio ACK, che diventerà 1 anche in uscita.

Questo procedimento è il medesimo del protocollo a livelli visto in precedenza. Quello che cambia è come mi riporto nelle condizioni iniziali.
Dopo aver lavorato con XIN, chiamo `reset RDY` che **riporta la rete RDY a 0**, quindi pronta ad accogliere un nuovo messaggio.
In poche parole, torno alle impostazioni iniziali nello stesso momento in cui ricevo il messaggio.

Esempio

Per capire meglio vediamo come esempio quello in esame, la divisione fra A e B interi con risultati Q, R.




```
0.  (RDYIN = 0) nop, 0
    (= 1)       A -> TA, B -> TB, set ACKIN,
                reset RDYIN, 1
1.  0 -> TQ, 2
2.  (segno(TA - TB), ACKOUT = 0-) TQ + 1 -> TQ,
                                  TA - TB -> TA, 2
    (= 11) TA -> R, TQ -> Q, set RDYOUT, reset ACKOUT ..
```

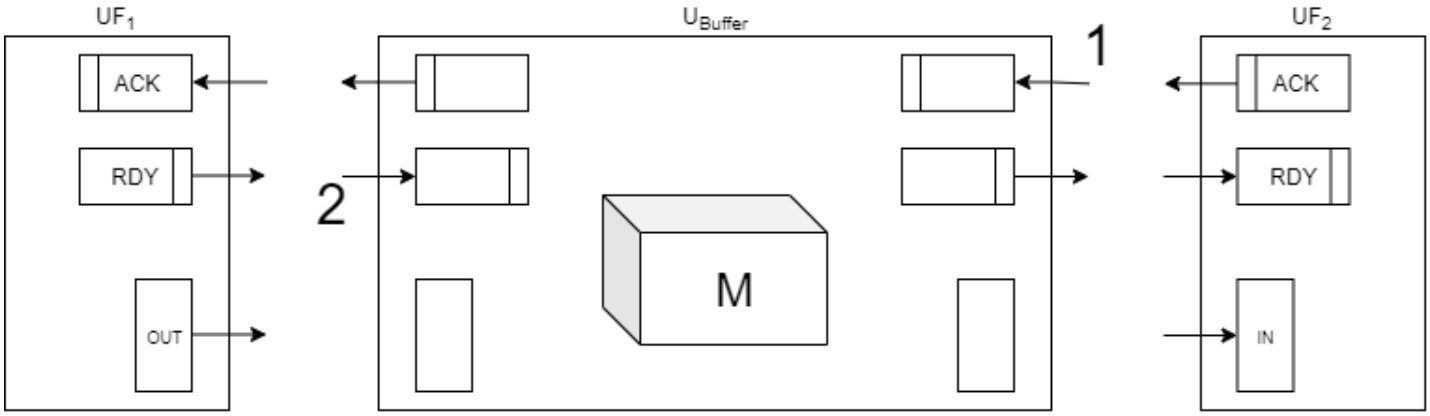
Possiamo notare che UF_1 manda i segnali A e B e **non ha altre operazioni mentre attende la risposta Q, R da UF_2**. Questa situazione si chiama **protocollo domanda/risposta** e solo in questo caso basta una coppia di indicatori di transizione.

```
UF1
0.  ... -> A, set RDY, 1
1.  (ACK = 0) nop, 1
    (= 1)     B -> ..., reset ACK
```

```
UF2
0.  (RDY = 0) nop, 0
    (= 1) A -> ..., 1
... elaborazione di UF2 ...
n.  ... -> B, set ACK, reset RDY, 0
```

4.4.3 Comunicazioni asincrone a n posizioni

In questo caso **non esiste una soluzione basata su semplici interfacce nelle due unità comunicati**, ma è **necessario introdurre una terza unità** (chiamata **unità buffer**) che **implementi una coda FIFO a n posizioni**. Il **mittente può spedire al più n messaggi senza che il destinatario effettui ricezioni**.



1 Se ci sono messaggi in memoria M, manda un messaggio a UF_2

2 Se c'è posto in memoria M, riceve un messaggio da UF_1 e lo memorizza in M

Se valgono entrambe ed M è vuota, allora il messaggio in ingresso da UF_1 viene passato direttamente ad UF_2

U_{buffer} Il codice di U_{buffer} avrà come **variabili di condizionamento RDY da UF_1, ACK da UF_2, condizione di memoria piena** e **condizione di memoria vuota**.
Le due condizioni di memoria piena/vuota possono essere gestite tramite un semplice contatore. SE per esempio abbiamo una memoria M con 2^k posizioni, useremo un contatore da k + 1 bit:

Memoria vuota `OR(CONT) = 0`

Memoria piena $\texttt{CONT}_0$ `= 1`

$\Rightarrow$ `0.  (RDY, ACK, OR(CONT),` $\texttt{CONT}_0$`, ...`

Il buffer implementa una politica **FIFO**: **il primo messaggio inviato** da UF_1 **deve essere il primo messaggio letto** da UF_2