ARCHITETTURE DEGLI ELABORATORI
Architetture e Sistemi Operativi - I Semestre

Corso A

Autore
Giuseppe Acocella
2024/25
https://github.com/Peenguino

Ultima Compilazione - May 7, 2025

Contents

1 Introduzione **4**

1.1 Astrazione a livelli 4
1.2 Sistema Binario 5
1.2.1 Complemento a Due 5
1.3 Porte Logiche 6
1.3.1 Composizionalità di porte 7

2 Logica Combinatoria e Sintesi Standard **8**

2.1 Parità di Stringa di Bit 8
2.2 Numero Porte e Costo in Tempo 9
2.3 Multiplexer e Composizionalità 9
2.4 Composizione/Elementi Ex-Novo 10
2.5 Valutazione del costo di stabilizzazione approssimata 12
2.6 Full Adder 12
2.6.1 Valutazione costo Full Adder 8 bit in ingresso 13
2.7 Demultiplexer 14
2.8 Mappa di Karnaugh 15
2.9 Alea 16
2.10 Full Adder e Sottrazione 16
2.11 Shifter 16
2.12 ALU 17
2.13 Arbitro 17

3 Logica Sequenziale **18**

3.1 SR Latch 18
3.2 D Latch 19
3.3 D Flip Flop 19
3.4 Enabled Flip Flop 20
3.5 Esempio di Sintesi Automa con Memoria 20
3.6 Reti di Mealy/Moore 22
3.6.1 Automa Parità Bit Mealy/Moore 23
3.7 Memorie Statiche/Dinamiche 25
3.7.1 Registri - Memorie Statiche 25
3.7.2 RAM - Memorie Dinamiche 26
3.8 Classificazione di Memorie 26

4 Assembler ARM v7 **28**

4.1 Architettura ARM v7 28
4.1.1 Descrizione istruzioni da 32 bit 29
4.1.2 Opzione THUMB 29
4.1.3 Descrizione Istruzioni per Categoria (OP, MEM, BRANCH) 30
4.2 Ciclo del Processore 32
4.3 Tipologie di Istruzioni 33
4.4 Schemi di Traduzione: Pseudocodice - ASM ARM 36

4.5 Branch a Contenuto di Registro 38

5 Microarchitetture **39**

5.1 Tipologie di Parallelismo 39

5.1.1 Costi in Tempo nel Parallelismo 40

5.1.2 Latenza e Tempo di Servizio in Parallelismo Temporale 41

5.2 Esempi di Implementazione Istruzioni Su Processore Single Cycle 41

5.3 Processore Single Cycle 43

5.4 Processore Multi Cycle 44

5.5 Processore Pipeline 45

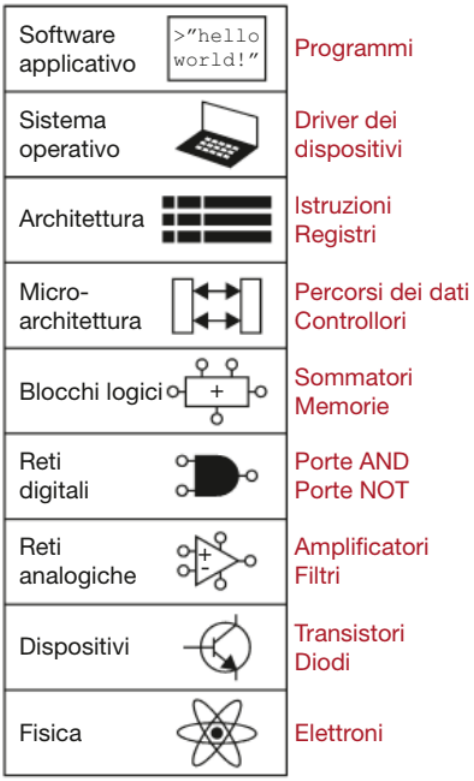
5.5.1 Dipendenze Logiche - Condizioni di Bernstein 46

5.5.2 Dipendenze sul Controllo 46

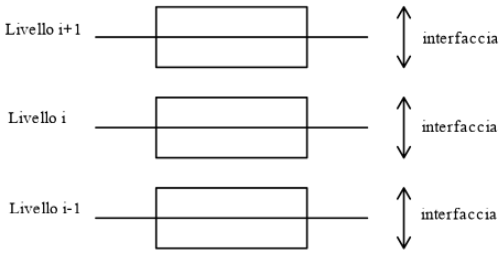
1 Introduzione

1.1 Astrazione a livelli

Un calcolatore elettronico si basa su dei livelli di astrazione:



Interfacce Questi livelli permettono di creare una gerarchia, stabiliscono una comunicazione tra i livelli stessi grazie a delle **interfacce**. Un livello $n+1$ infatti si basa su elementi composti del livello n. Ad esempio, le porte logiche del livello *circuiti* si basa sulla composizionalità di elementi del livello *device*.



Gerarchia, modularità e regolarità Definita quindi l'astrazione, che ci permette di ragionare più ad alto livello, allontanandoci dall'implementazione e realizzazione fisica di un calcolatore, dobbiamo introdurre tre regole che vanno rispettate per mantenere una lineare crescita ed evoluzione dei calcolatori:

1. **Gerarchia**: Ogni livello ha uno specifico posto.
2. **Regolarità**: Ogni elemento ha una struttura regolare costante che permette una composizione scalabile.
3. **Modularità**: Ogni elemento può essere utilizzato in composizioni diverse, mantenendo la sua struttura.

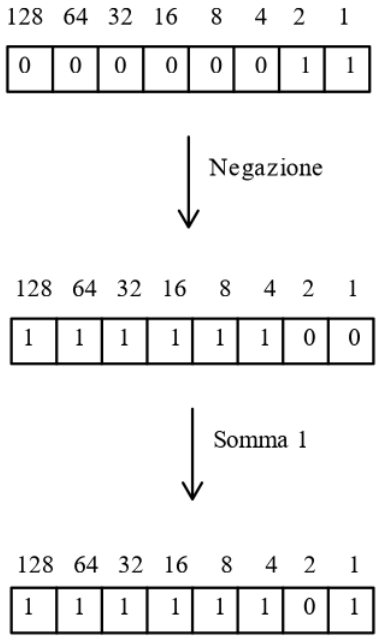
1.2 Sistema Binario

Il **sistema binario** è un sistema **posizionale**, ha base 2. che utilizza come cifre { 0,1 }.

$$2^2 * 1 + 2^1 * 1 + 2^0 * 0 = 6_{(10)}$$

1.2.1 Complemento a Due

Il complemento a due nel sistema binario permette la rappresentazione dei numeri relativi. Il bit più significativo indica il segno (0 segno positivo, 1 segno negativo). Mostriamo un esempio su 1 byte dedicato alla rappresentazione di un numero in base 10. Anche per poter leggere un numero post complemento a due, e convertirlo in base 10, va ricordato il bit più significativo applicando gli stessi passi. Mostriamo il complemento a due di 3_{10}.

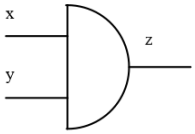


Numeri con virgola mobile Simile è la rappresentazione dei numeri con la virgola. Il numero stesso sarà infatti rappresentato da una mantissa e da un esponente. Verranno quindi dedicati dei bit alla mantissa, un bit al segno e i restanti all'esponente, in base a quale sia il focus del tipo con virgola mobile in questione.

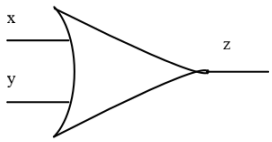
1.3 Porte Logiche

Introduciamo le porte logiche, ossia la rappresentazione in livello componenti degli operatori logici AND, OR, NOT:

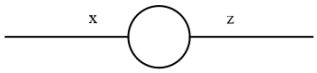
1. **Porta AND**:



2. **Porta OR**:



3. **Porta NOT**:



Ricordiamo che questo tipo di porte accetta due input $\{x, y\}$ e produce un output z.

Vedremo più avanti in che occasioni è possibile aumentare il numero di input e l'effetto sul circuito complessivo.

Map dei valori 0-1 E' necessario stabilire una correlazione tra voltaggio e bit rappresentato. Solitamente:

$$0 \mapsto 0\, volt$$

$$1 \mapsto 3/5/12\, volt \tag{1}$$

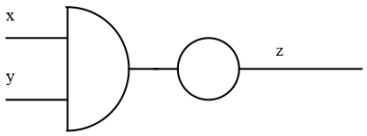
Map operatori Per poter esprimere più facilmente espressioni articolate, anche gli operatori vengono mappati:

$$\begin{aligned} AND(x, y) &\mapsto x * y \\ OR(x, y) &\mapsto x + y \\ NOT(x) &\mapsto \overline{x} \end{aligned} \tag{2}$$

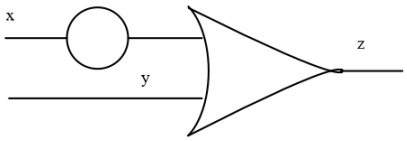
1.3.1 Composizionalità di porte

Possiamo comporre le porte per rappresentare espressioni logiche più complesse:

$$NOT\,(x\;AND\;y) \tag{3}$$



$$NOT(x)\;OR\;y \tag{4}$$



2 Logica Combinatoria e Sintesi Standard

Come realizziamo un circuito basandoci sulla descrizione ad alto livello di un problema? Bisogna seguire un preciso algoritmo che sviluppi una tabella di verità rispetto alle variabili in questione e che interpreti i dati prodotti.

2.1 Parità di Stringa di Bit

Mostriamo l'algoritmo in questione su questo esempio:

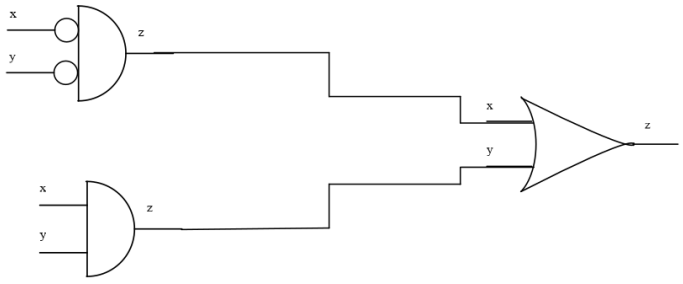
1. **Descrizione funzione**: Se numero di bit settati a 1 è pari allora torna 1, altrimenti 0.
2. **Tabella di verità**: Sviluppo la tabella e noto quali valori mi permettono di produrre 1 in output.

b1	b0	z
0	0	1
0	1	0
1	0	0
1	1	1

3. **Espressione associata**: Seleziono le variabili delle stesse righe dei vari 1 in output e
   (a) Nego la variabile se è settata a 0.
   (b) Non la nego se la variabile è settata a 1.

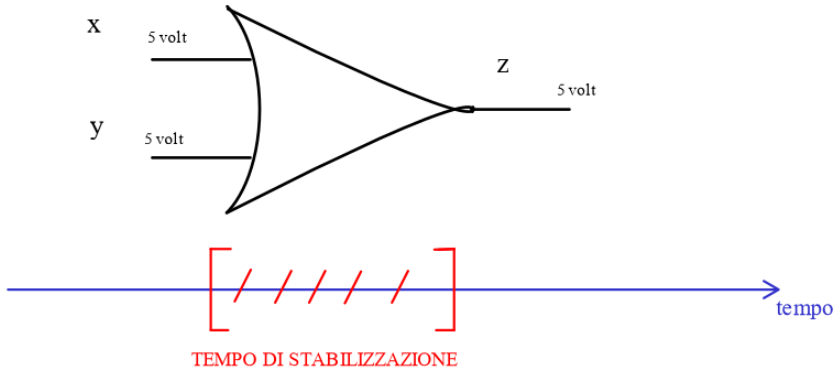
$$z = \overline{b_1}\,\overline{b_0} + b_1\,b_0 \tag{5}$$

4. **Rappresentazione Circuito**: Possiamo dunque rappresentare l'espressione in questione in formato circuito:



2.2 Numero Porte e Costo in Tempo

A livello elettrico, per poter raggiungere il voltaggio richiesto durante il map dei bit, sarà necessario attendere del tempo:



Questo viene chiamato **tempo di stabilizzazione**. Questo dipende anche dal numero di ingressi delle porte in questione. Oltre gli 8 ingressi è dimostrata l'inefficienza del circuito composto da porte di questo tipo. Solitamente indichiamo il costo dell'attraversamento di una colonna di porte con Δt.

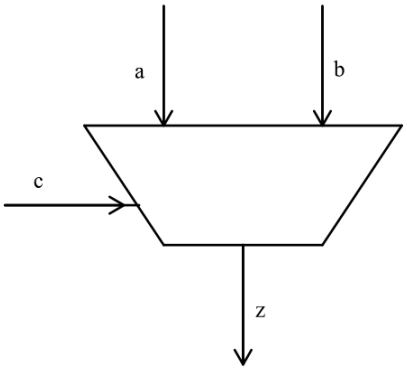
2.3 Multiplexer e Composizionalità

Mostriamo un multiplexer (**livello circuiti**) formato dalla composizione di varie porte logiche (**livello componenti**) secondo le regole della **logica combinatoria**.

1. **Descrizione Funzione e Tabella**: Descriviamo la funzione che vogliamo rappresentare in circuito e successivamente produciamo la relativa tabella di verità.

   (a) Se $c = 1 \Rightarrow z \mapsto a$

   (b) Se $c = 0 \Rightarrow z \mapsto b$



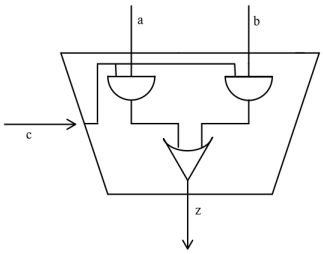
c	a	b	z
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1

2. **Espressione associata**: Descriviamo dunque l'espressione osservando la tabella di verità. Notiamo che nella prima riga e nella quarta, rispettivamente b e a possono assumere qualsiasi valore.

$$z = \overline{c}\, a\, (b + \overline{b}) + c\, (a + \overline{a}) =$$

$$= \overline{c}\, a + c\, b \tag{6}$$

3. **Rappresentazione Circuito**: Osserviamo il circuito corrispondente.



Sarà possibile d'ora in poi utilizzare il **multiplexer** come unità del livello di astrazione **circuiti**. Abbiamo però avuto la dimostrazione di come si compone di unità del livello sottostante **componenti**.

2.4 Composizione/Elementi Ex-Novo

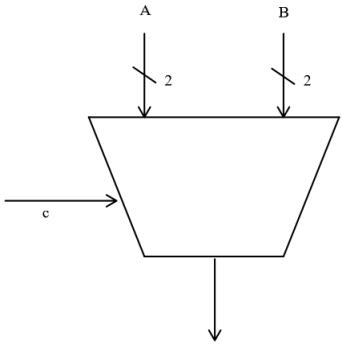
Dato che abbiamo introdotto il tempo di stabilizzazione Δt possiamo valutare quale approccio sia più efficiente tra:

1. Composizione di oggetti già esistenti del livello sottostante nella gerarchia dei livelli di astrazione.
2. Progettazione di nuove componenti, aumentando il livello di porte e costruendolo su misura alla formula corrente.

Mostriamo un esempio nella pagina successiva.

Multiplexer a 2 entrate a 2 bit Definiamo prima la sua formula e la corrispondente tabella di verità. Successivamente lo rappresenteremo con due reti diverse.

1. **Descrizione Funzione e Tabella**: Assumiamo di avere due entrate da due bit a, b, un entrata di controllo c ed un uscita da due bit. Vogliamo quindi che se $c = 0$, allora in output avremo il valore di a, mentre quello di b se $c = 1$.



c	A1	A0	B1	B0	Z1	Z0
0	0	0	-	-	0	0
0	0	1	-	-	0	1
0	1	0	-	-	1	0
0	1	1	-	-	1	1
1	-	-	0	0	0	0
0	-	-	0	1	0	1
0	-	-	1	0	1	0
0	-	-	1	1	1	1

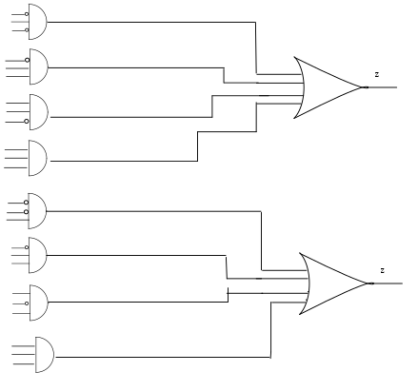
2. **Espressione Associata**: Osservando la tabella, produciamo l'espressione, composta dalla rappresentazione di $b1, b2$

   (a) $z_1 = \overline{C}A_1\overline{A_0} + \overline{C}A_1A_0 + CB_1\overline{B_0} + CB_1B_0$

   (b) $z_0 = \overline{CA_1}A_0 + \overline{C}A_1A_0 + C\overline{B_1}B_0 + CB_1B_0$

3. **Reti**: Possiamo rappresentarne diversi tipi.

   (a) **Rappresentazione Nuova**: Si basa su porte AND da tre ingressi (prima colonna) e porte OR da quattro ingressi (seconda colonna). Tempo di stabilizzazione $2\Delta t$.



   (b) **Rappresentazione Composta**: Potremmo anche utilizzare 2 multiplexer in prima colonna e uno nella seconda per poter ricavare lo stesso risultato. Questo è un raro caso in cui le due implementazioni causano lo stesso costo. Solitamente un implementazione nuova permette un ottimizzazione del costo di stabilizzazione provocato.

2.5 Valutazione del costo di stabilizzazione approssimata

A volte è necessario valutare quanto ci costa implementare ex novo un espressione logica. Dunque eseguiamo questi passaggi:

1. **Numero righe e termini in AND**

   Visualizziamo la tabella logica, notiamo quanti k input abbiamo. Possiamo dunque stabilire che la tabella produrrà $2^k - 1$ termini in and composti tra gli input. Dunque possiamo stabilire questa come prima "colonna" del costo complessivo di stabilizzazione.

2. **Altezza dell'albero**

   Ricordando che il massimo di entrate ottimali in una porta è pari ad 8 allora possiamo determinare l'attuale numero di input k come base di un logaritmo, dove l'argomento sarà appunto $2^k - 1$.

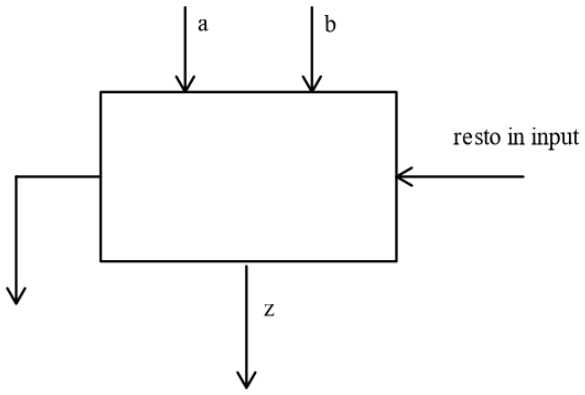
$$ceil\,[\,log_k(\,2^k - 1\,)\,]$$

Sommando quindi questi due livelli, otteniamo un approssimazione del costo di stabilizzazione totale della tabella di verità implementata ex novo.

2.6 Full Adder

Il Full Adder permette di eseguire somme con riporto in colonna. Questo avrà quindi:

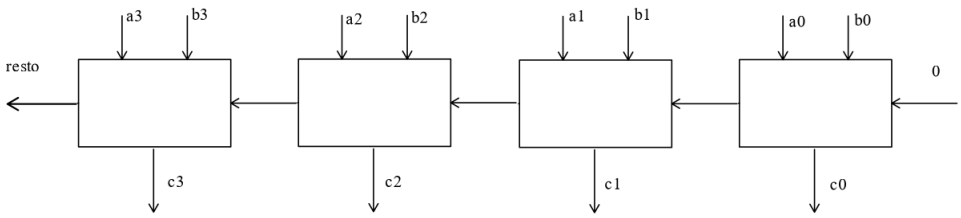
1. **Input**: Due bit in input che vogliamo sommare.
2. **Riporto in ingresso**: Di default settato a 0.
3. **Riporto in uscita**: Se la somma dei due bit "sfora" viene prodotto in output un riporto.
4. **Output**: Somma dei due bit in ingresso.



2.6.1 Valutazione costo Full Adder 8 bit in ingresso

Valutiamo il costo delle implementazioni di un full adder con 8 bit in ingresso.

1. **Composizione di Full Adder**: Assumiamo che un Full Adder singolo costi $2\Delta t$. Dobbiamo dunque impostare questa rete:



Questa implementazione costa quindi $8\Delta t$.

2. **Implementazione Ex Novo**: Abbiamo stabilito di avere 8 bit in ingresso.

   (a) **Colonna degli AND**: Dunque al primo livello abbiamo 8 termini in AND, ed avremo nella peggiore delle ipotesi 255 oggetti simili. Dunque paghiamo il Δt della prima colonna degli AND.

   (b) **Restanti colonne e altezza albero**: Consideriamo quindi la formula

$$ceil\,[\,log_8(\,2^8 - 1\,)\,] =$$

$$= ceil\,[log_8 255] =$$

$$= ceil\,[2, 2...] = 3$$

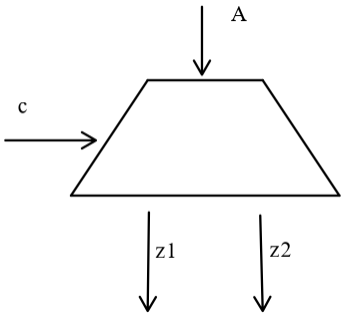
   (c) **Costo Complessivo**: Possiamo quindi sommare i due costi ricavati, quindi:

$$\Delta t + 3\Delta t = 4\Delta t$$

2.7 Demultiplexer

Questo elemento ci permetterà di scegliere la direzione di uscita di uno specifico ingresso. Dunque non ci interessa la parte di tabella che ipotizza le casistiche nelle quali l'input è nullo. Mostriamo una rappresentazione.

1. **Funzione e Tabella Verità**:



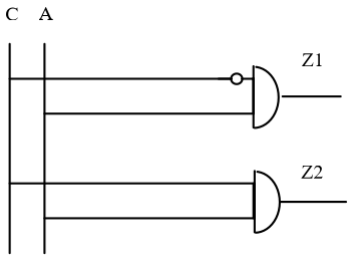
C	A	Z1	Z2
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

2. **Espressione Logica**:

$$z_1 = \overline{c}a$$

$$z_0 = ca$$

3. **Rete**:



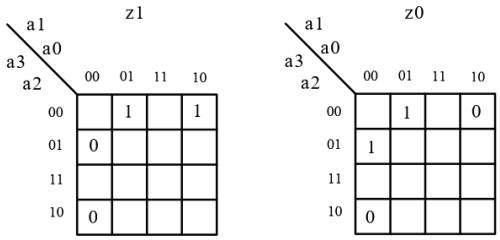
2.8 Mappa di Karnaugh

La Mappa di Karnaugh è un metodo alternativo che permette la produzione di espressioni ottimizzate. Questa è componibile interpretando la tabella di verità. Mostriamo un esempio:

1. **Tabella di Verità**: Assumiamo di essere arrivati a questa fase della logica combinatoria di una funzione arbitraria e di aver ottenuto questa tabella:

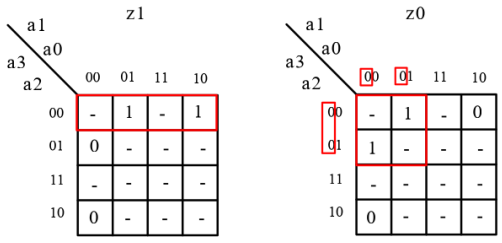
a3	a2	a1	a0	z1	z0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

2. **Mappe di Karnaugh**: Ogni colonna in output sarà rappresentata da una singola mappa. Sulle righe avremo le due variabili di input a_3, a_2 mentre sulle colonne le restanti variabili di input a_1, a_0.



Tutte le celle vuote rappresentano valori arbitrari che non variano l'esito nella selezione degli implicanti. Potremmo inserire anche dei trattini $-$

3. **Valutazione implicanti ed espressione prodotta**: Bisogna trovare degli **implicanti**, ossia "blocchi" di 1 all'interno della mappa in una quantità 2^k. L'esponente k ci permetterà di capire quanti termini stiamo risparmiando rispetto all'espressione originale.



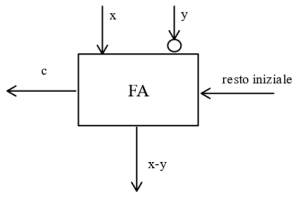
Dunque, l'implicante nella prima mappa si basa sulla riga in cui le prime due variabili sono negate allora $z_1 = \overline{a_3}\,\overline{a_2}$, invece nella seconda mappa selezioniamo gli elementi su righe e colonne in comune (cerchiati in rosso), quindi $z_0 = \overline{a_3}\,\overline{a_1}$.

2.9 Alea

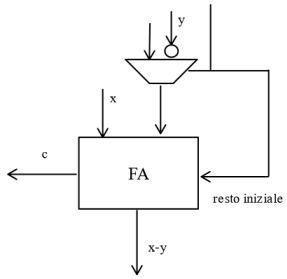
Generando espressioni ottimizzate con i metodi enunciati è possibile causare a livello pratico il fenomeno dell'alea. Dato che ogni colonna di porte genera un tempo di stabilizzazione Δt allora è possibile ottenere dei risultati errati per un breve periodo di tempo. Questo fenomeno è risolvibile aggiungendo un implicante "transitorio" alla mappa che non permetta la sovrapposizione delle operazioni in periodi di tempo non corretti.

2.10 Full Adder e Sottrazione

Come possiamo effettuare una sottrazione con i componenti che abbiamo già a disposizione? Basterebbe negare (complemento a due) il secondo termine dell'operazione.

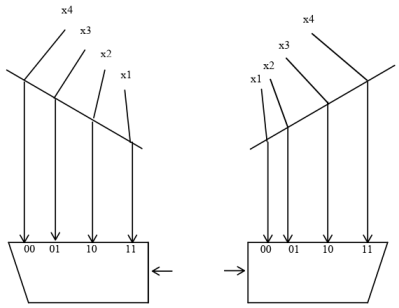


Somma o differenza e Full Adder La soluzione ottimale è quella di utilizzare un multiplexer al secondo input, stabilendo quindi se negare o meno il termine y.



2.11 Shifter

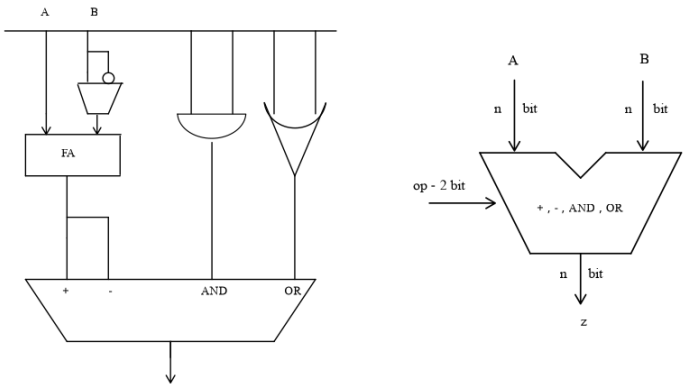
Mostriamo la rete che ci permette di shiftare a sinistra o a destra i bit in ingresso.



Sceglieremo dunque una posizione di output caratterizzata da quanti zeri di default possiede. Il complemento degli zeri saranno quindi i bit shiftati. Per poter permettere una scelta di direzione (sx, dx) si pone un multiplexer alla colonna successiva dei due shifter, in questo modo si seleziona quale fosse la direzione dello shift effettuato.

2.12 ALU

Grazie alla composizione di elementi ricavati dalla gestione di reti basilari e porte logiche possiamo ricavare un nuovo componente, ossia un **ALU** (Aritmetic Logic Unit). Mostriamo due rappresentazioni:



Come rappresentato, questa unità ci permette di eseguire operazioni logiche e aritmetica.

2.13 Arbitro

L'arbitro è un ulteriore componente che, dato un numero k ingressi, stabilisce quali di questi "vince" in base a se i bit in questione valgono 1 oppure 0. In caso di occorrenza di più bit in ingresso ad uno, vince il primo da sinistra a destra.

a b c d	za zb zc zd
0 0 0 0	0 0 0 0
1 0 0 0	1 0 0 0
0 1 0 0	0 1 0 0
0 0 1 0	0 0 1 0
0 0 0 1	0 0 0 1
0 0 1 1	0 0 1 0
0 1 1 0	0 1 0 0
1 1 0 0	1 0 0 0
0 1 0 1	0 1 0 0
1 0 0 1	1 0 0 0
1 0 1 0	1 0 0 0
1 1 1 0	1 0 0 0
0 1 1 1	0 1 0 0
1 0 1 1	1 0 0 0
1 1 0 1	1 0 0 0
1 1 1 1	1 0 0 0

Figure 1: Se osserviamo l'espressione generata osservando gli 1 nelle colonne di output possiamo determinare un upper bound del costo a $2\Delta t$

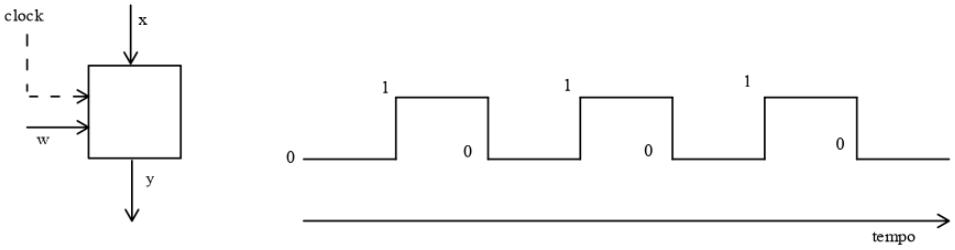
3 Logica Sequenziale

In questo capitolo si affronta il tipo di logica che tiene conto dei risultati passati grazie ad una memoria. Introduciamo il suo funzionamento di una memoria come rete e successivamente assumeremo ognuno di questi componenti come componenti noti.

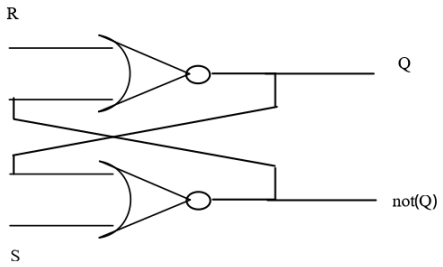
Come nella logica combinatoria definiremo quindi delle proprietà/funzioni, ma in questo capitolo queste avranno degli stati, proprio come degli automi.

3.1 SR Latch

Immaginiamo una rete che permette due operazioni, ossia **set** e **reset**. Quindi se $w = 1$ e $clock = 1$ scrive x.



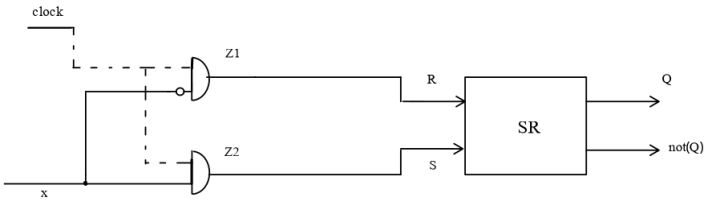
Rete Corrispondente La rete che ci permette di effettuare queste operazioni è:



1. **Set**: $S = 1$ il circuito ricorda 1.
2. **Reset**: $R = 1$ il circuito ricorda 0.

3.2 D Latch

E' necessario imporre che non si possa dare contemporaneamente i valori $S = 1, R = 1$ nella SR Latch. Dunque possiamo impostare questa condizione con delle porte AND in ingresso:

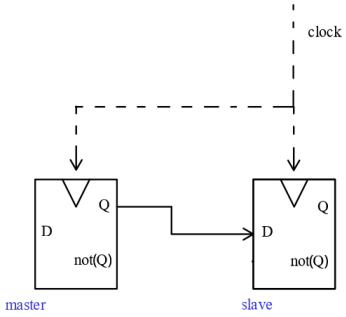


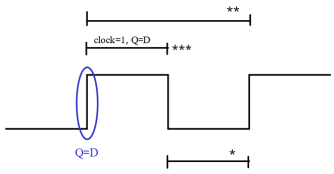
Abbiamo definito quindi un nuovo componente, ossia il **D Latch**.

3.3 D Flip Flop

Il **D Flip Flop** si compone di due **D Latch**:

1. **Master**: Primo D Latch, ha il clock negato in ingresso.
2. **Slave**: Primo D Latch, ha il clock affermato in ingresso.



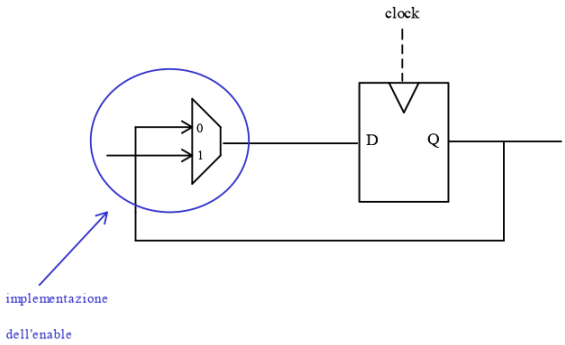


Funzionamento Descriviamo la figura, assumendo che appena prima del periodo (******) è necessario conoscere cosa vogliamo scrivere in input (**D**).

1. *****: $clock = 0$, ma lo **slave** ha già acquisito il valore nella frazione di periodo precedente. Dunque il valore viene mantenuto dato che lo **slave** ha il clock in ingresso affermato.
2. ******: Intero periodo in cui verrà mantenuto il valore in ingresso al **master** poco prima dell'inizio del periodo stesso.
3. *******: Frazione di periodo dove il $clock = 1$, dunque il **master** ha già portato il suo ingresso **D** alla sua uscita **Q**.

3.4 Enabled Flip Flop

Ponendo un multiplexer all'ingresso del primo **D Latch** del **D Flip Flop** possiamo scegliere di scrivere il nuovo valore in ingresso oppure riscrivere quello corrente. Questo permette allo schema attuale di essere un tipo di memoria **statica**, essendo che riscriverà il suo valore, indipendentemente da se verrà letta o meno.

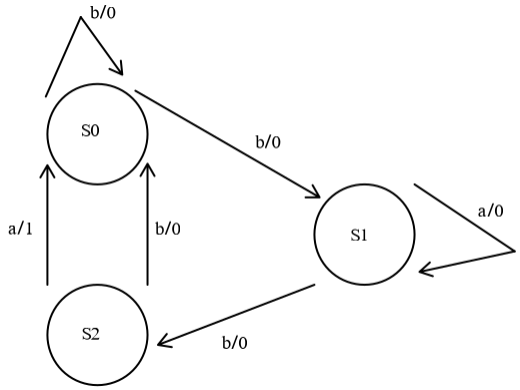


Questo ci permetterà quindi di generare da ora in poi registri da 1 bit.

3.5 Esempio di Sintesi Automa con Memoria

Vogliamo rappresentare una funzione che ricordi lo stato precedente per poter scegliere tra le possibili scelte.

1. **Automa e Map Stati**: Assumiamo di voler rappresentare questo automa con una tabella di verità per poterlo convertire in una rete.



Mappiamo in questo modo gli stati per rappresentarli nella tabella di verità:

$S_0 \mapsto 00 \quad S_1 \mapsto 01 \;\; S_2 \mapsto 10$

2. **Tabella di Verità**: Rappresentiamo la tabella osservando le possibili scelte dell'automa dato.

S		IN	S'		Z
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	0	0	0

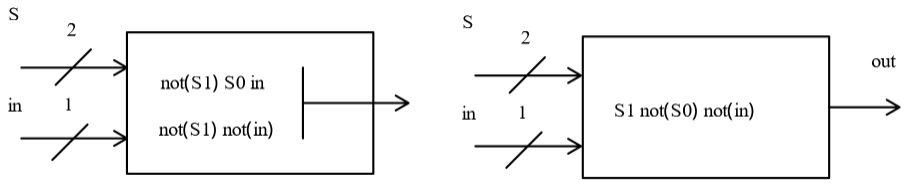
3. **Espressione**: Rappresentiamo 3 espressioni, rispettivamente prima, seconda colonna per gli stati e output.

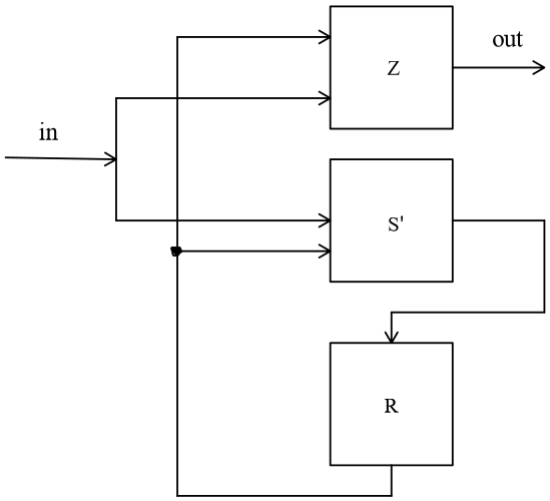
   (a) $S_1' = \overline{S_1}\, S_0\, in$

   (b) $S_0' = \overline{S_1}\, \overline{S_0}\, \overline{in} + \overline{S_1}\, S_0\, \overline{in} + \overline{S_1}\, \overline{in}$

   (c) $S_1' = \overline{S_1}\, S_0\, in$

4. **Rete e Componenti**: Rappresentiamo prima i componenti singolarmente, rispettivamente stati di transizione e output, e successivamente mostriamo la complessiva rete generata.





3.6 Reti di Mealy/Moore

Grazie all'introduzione dei registri definiti precedentemente, possiamo rappresentare non più solo funzioni ma veri automi a stati finiti. Prima dei registri era infatti impossibile mantenere le informazioni riguardanti i risultati ottenuti precedentemente. Distinguiamo però due diversi tipi di automi e corrispondenti reti:

1. **Reti di Mealy**: Reti il cui output dipende sia da **stato** (informazione mantenuta nei registri), sia da **input**
2. **Reti di Moore**: Reti il cui output dipende esclusivamente dallo stato.

Parlando quindi di semplici rappresentazioni grafiche, le reti di **Mealy** hanno input/output sugli archi mentre le reti di **Moore** hanno solo input, dato che gli output dipendono esclusivamente dallo stato.

Spesso nelle reti si preferisce utilizzare entrambi gli schemi, sia **Mealy** sia **Moore** dato che la seconda tipologia in questione "forza" il passaggio per i registri, evitando in ogni modo la non stabilizzazione della rete.

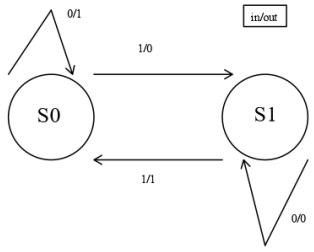
Mostriamo un esempio sulle differenze tra **Mealy** e **Moore** sull'automa di parità di bit alla successiva pagina.

3.6.1 Automa Parità Bit Mealy/Moore

Mostiamo l'intera implementazione di un automa che dati n di bit ritorna 1 se $\#\,di\,1$ è pari e 0 se $\#\,di\,1$ è dispari.

1. **Versione Mealy**: Seguiamo passo passo la creazione della rete:

   (a) **Rappresentazione Automa**: La proprietà descritta sopra sarà così rappresentata in automa:



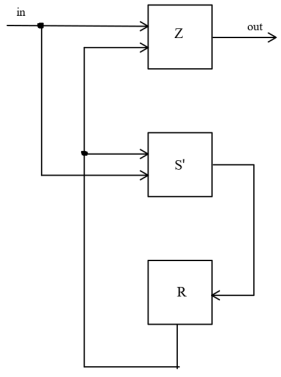
   (b) **Tabella di Verità**: Notiamo che $S_0 \in \{0,1\}$ e anche $in \in \{0,1\}$

S	IN	S'	Z
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

   (c) **Espressione**: L'espressioni prodotte dall'interpretazione della tabella saranno:

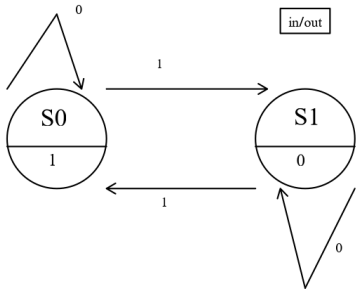
$$s^I = \overline{s}\,in \;+\; s\,\overline{in} \quad , \quad z = \overline{s}\,\overline{in} \;+\; s\,\overline{in}$$

   (d) **Rete**: Mostriamo la rete prodotta.



2. **Versione Moore**: Seguiamo il secondo sviluppo notando le differenze:

   (a) **Rappresentazione Automa**: Mostriamo la versione **Moore** dell'automa.



   (b) **Rappresentazione Automa**: Per rappresentare l'automa saranno necessarie due tabelle di verità:

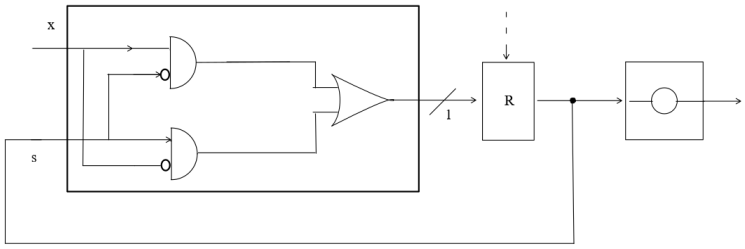
s	x	s'
0	0	0
0	1	1
1	0	1
1	1	0

s	z
0	1
1	0

   (c) **Espressioni**: Interpretiamo la tabella:

$$s^I \;=\; \overline{s}\,x \;+ s\,\overline{x} \quad , \quad z \;=\; \overline{s}$$

   (d) **Rete**: Mostriamo la rete prodotta:



   (e) **Tempi di Stabilizzazione e Clock**: E' necessario che i tempi di stabilizzazione dell'intero circuito siano minori di un ciclo di clock. In questo caso il circuito che produce s^I (la prima rete combinatoria) ha un tempo di stabilizzazione di $2\Delta t$, invece la produzione di z appena dopo il registro si basa solo su una negazione, dunque assumeremo che costi $0\Delta t$. Dunque elenchiamo tutti i dati in questione:

$$r \;=\; max\{2\Delta t\,,\,0\Delta t\} \;+\; t_{write} \quad , \quad lunghezzaClock \;\geq\; r$$

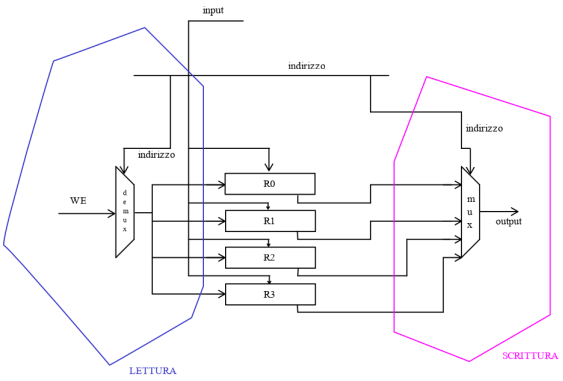
3.7 Memorie Statiche/Dinamiche

Le reti fino ad ora descritte si basavano su dei **registri** da 1 bit ciascuno. Questo tipo di memorie hanno particolari caratteristiche. Elenchiamo diversi tipi di memorie:

3.7.1 Registri - Memorie Statiche

1. 1 bit di memoria costa circa 20 **transistor**.
2. Effettuano operazioni nell'ordine di tempo dei **picosecondi** ossia 10^{-12} *sec*.
3. Solitamente, si compongono raggruppamenti di questo tipo di memorie massimo da 1024/2048 bit.

Lettura e Scrittura su Registri : Possiamo immaginare una memoria come un array di registri da un bit. Sarebbe però necessario, dato un **indirizzo**, stabilire delle operazioni di **lettura** e **scrittura**.



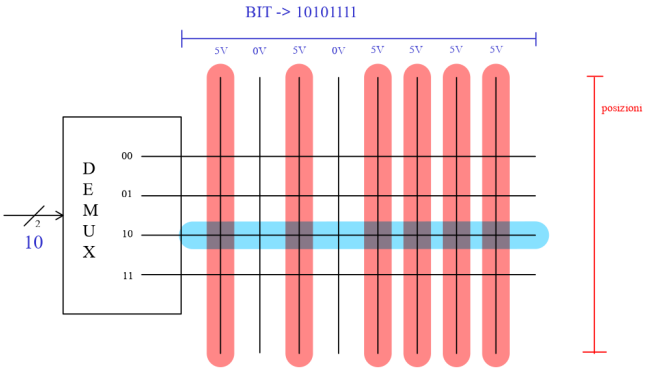
1. **Scrittura**: Possiamo scrivere nei registri in questo modo:
   (a) Inviamo l'input ad ogni registro (simil broadcast).
   (b) Poniamo un **demultiplexer** con in ingresso il **write enable (WE)** ed il segnale di controllo sarà l'**indirizzo**.

   In questo modo il nuovo input verrà inserito solo nel registro a cui è stato settato il **write enable** a 1.
2. **Lettura**: La lettura si baserà invece su un concetto leggermente più semplice:
   (a) Tutte le uscite dei registri entrano in un **multiplexer**
   (b) L'**indirizzo del registro** di cui vogliamo acquisire il valore sarà utilizzato come **segnale di controllo** nel multiplexer

Osservando infine le tabelle di verità richieste per rappresentare lettura e scrittura, verrebbe fuori che la **scrittura** è più veloce della **lettura**, dato che la prima richiede 1 livello di logica e la seconda 2 livelli di logica. Questo perchè ogni colonna di output della scrittura è "indipendente" dalle altre, non abbiamo un OR tra loro nelle colonne successive, mentre la lettura prende il massimo tra i tempi delle due componenti.

3.7.2 RAM - Memorie Dinamiche

Fino ad ora abbiamo rappresentato memorie che mantenevano nel tempo il loro valore. Le **memorie dinamiche** invece hanno bisogno di un costante **refresh** dato che di default non manterranno all'infinito il loro dato, mostriamo il perchè:



1. **Scrittura**: La scrittura avviene ponendo un voltaggio alto $5v$ sulle colonne e un bit in ingresso rispetto alla posizione sulle righe scelto dal multiplexer. Negli "incroci" di 1 in riga e colonna viene caricato un condensatore che è posizionato in quella zona. (Righe rosse nella rappresentazione).

2. **Lettura**: La lettura avviene selezionando la riga che vogliamo leggere, e laddove il condensatore è carico verrà letto 1, altrimenti 0 (Riga blu nella rappresentazione).

Vengono definite **memorie dinamiche** perchè se i condensatori non vengono periodicamente **refreshate**, queste perderanno la loro carica e quindi il bit che stavano immagazzinando.

3.8 Classificazione di Memorie

1. **RAM - Random Access Memory**: Tipologia di memoria caratterizzata dalla possibilità di poter leggere ogni posizione in memoria, come già mostrato in precedenza.

2. **ROM - Read Only Memory**: Tipologia di memoria stampata in maniera standard, non permette dunque alcun tipo di scrittura ma solo lettura. Vengono dunque saldati fisicamente i contatti che rappresenteranno 1 bit di memoria.

3. **EPROM - Electrically Programmable Read Only Memory**: Tipologia particolare di **ROM** che permette di "bruciare" in contatti sulla memoria per poter creare una **ROM** personalizzata.

4. **EEPROM - Erasable Electrically Programmable Read Only Memory**: Simile alla precedente categoria, ma offre la possibilità di effettuare più volte una "bruciatura" dei contatti. Memorie simili vengono utilizzate per i **BIOS**.

5. **Memoria Associativa**: Immaginiamo di voler implementare una memoria basata su $< chiavi, valori >$. E' necessario impostare tutti i livelli necessari in modo tale da poter effettuare un esempio di lettura:

   (a) **Livello Comparazione**: Assumendo di avere in memoria delle coppie $< chiave, valore >$ allora date delle chiavi abbiamo bisogno di un **comparatore** che possa stabilire se le chiavi in input siano presenti o meno in memoria.

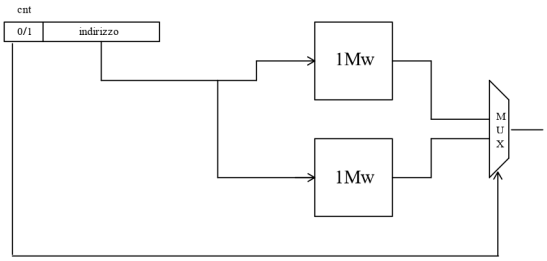
   (b) **Porta OR**: La porta OR ci permette di stabilire se le chiavi in input sono presenti o meno, ancora prima di cercare il valore corrispondente.

   (c) **Codificatore**: Il codificatore prende in input gli esiti dei comparatori e "interpreta" la posizione dell'1, generando un output che verrà utilizzato come segnale di controllo nel multiplexr finale.

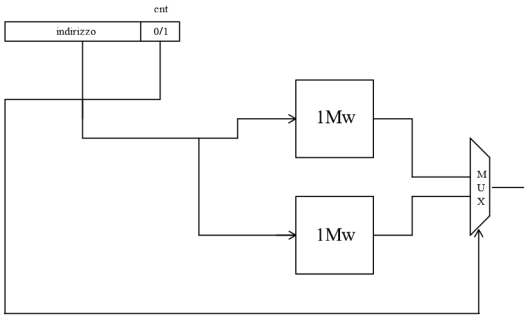
   (d) **Multiplexer**: Preso come **controllo** la posizione dell'1, che rappresenta la chiave presente, possiamo utilizzare questo per scegliere grazie al multiplexer il valore in output corrispondente al valore collegato alla chiave iniziale.

6. **Memoria Modulare**: Immaginiamo di avere due zone di memoria ed n bit dedicati agli indirizzi. Determineremo le sottocategorie in base a quale bit utilizziamo per scegliere a quale zona di memoria stiamo facendo riferimento (più/meno significativo).

   (a) **Sequenziale**: La Memoria Modulare Sequenziale utilizza il bit più significativo per indicare a quale zona di memoria fa riferimento quell'indirizzo. Questo provoca un utilizzo sequenziale delle celle di memoria disponibili.



   (b) **Interallacciata**: La Memoria Modulare Interallaciata utilizza il bit meno significativo per indicare a quale zona di memoria fa riferimento quell'indirizzo. Questo provoca un utilizzo alternato tra le due zone delle celle di memoria disponibili.



4 Assembler ARM v7

In questo capitolo analizzeremo comandi a livello macchina in uno specifico linguaggio *Assembly* ossia quello dell'architettura **ARM v7**.

4.1 Architettura ARM v7

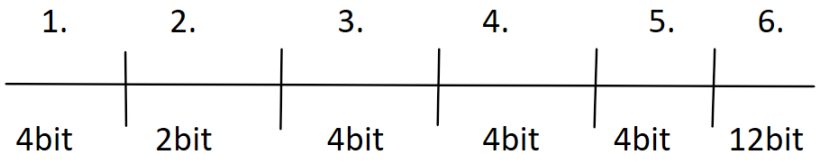
Descriviamo gli elementi fondamentali di questo tipo di architettura:

1. **Registri**: I registri sono un tipo di memorie statiche che operano nell'ordine dei picosecondi. In questo tipo specifico di architettura, ogni registro è da 32 bit. In totale sono presenti 16 registri, che si suddividono in due sottocategorie:
   (a) **Registri General Purpose**: Sono registri generici, utilizzati per effettuare operazioni di **load** e **store** che permettono l'interazione tra **memoria** e **registri**.
   (b) **Registri Speciali**: I registri speciali mantengono valori ricavati da particolari istruzioni oppure contengono informazioni utili al proseguimento dell'esecuzione del programma:
      i. **R15 - PC**: Il **Program Counter** permette di mantenere l'indirizzo della corrente istruzione. Dunque per andare all'istruzione successiva si effettua:
      $$PC = PC + \text{"1"}$$
      $$PC = PC + 4$$
      ii. **R14 - LR**: Il **Link Register** mantiene l'indirizzo di ritorno.
      iii. **R13 - SP**: Lo **Stack Pointer** mantiene l'informazione sullo stack corrente.
      iv. **CPSR**: Il **Current Program Status Register** contiene varie flag ottenute dalla valutazione di alcuni comandi di comparazione:
         A. **N**: Negative Condition Flag ottenuta da una CMP.
         B. **Z**: Zero Condition Flag ottenuta da una CMP.
         C. **C**: Carry Condition Flag.
         D. **V**: Overflow Condition Flag.

2. **Memoria**: La memoria possiamo immaginarla come vettore di celle, ognuna da 32 bit e si suddivide in due grandi categorie:
   (a) **Memoria Dati**: Mantiene i dati in memoria.
   (b) **Memoria Istruzioni**: Mantiene le istruzioni da seguire. Ogni **istruzione** è rappresentata in 32 bit:

4.1.1 Descrizione istruzioni da 32 bit

Ogni istruzione da 32 bit è così composta:



1. **Condizioni sull'istruzione**: Bit riservati alle condizioni (indicate nelle slide con XX oppure X).
2. **Tipo di Operazione**:
   (a) 00: Istruzione Operativa
   (b) 01: Istruzione Memoria
   (c) 00: Istruzione di Salto
3. **Registro Destinazione**: Indirizzo del registro in cui scriveremo il risultato dell'istruzione corrente.
4. **Registro Sorgente 1**: Indirizzo del registro in cui troveremo una sorgente su cui effettuare il calcolo
5. **Func, tipo operazione**: Descrive il "titolo" dell'istruzione, dunque a che tipo di istruzione stiamo facendo riferimento.
6. **Rappresentazione Costanti**: Questi bit sono dedicati alla rappresentazione di costanti, sono infatti 12 bit che ci permettono di rappresentare, anche grazie agli shift, anche numeri molto grandi.

Rappresentazione Esadecimale I 32 bit verranno rappresentati in 8 cifre esadecimali, ogni cifra rappresenterà "metà" byte di un istruzione.

```
50:   e2877001        add     r7, r7, #1
```

4.1.2 Opzione THUMB

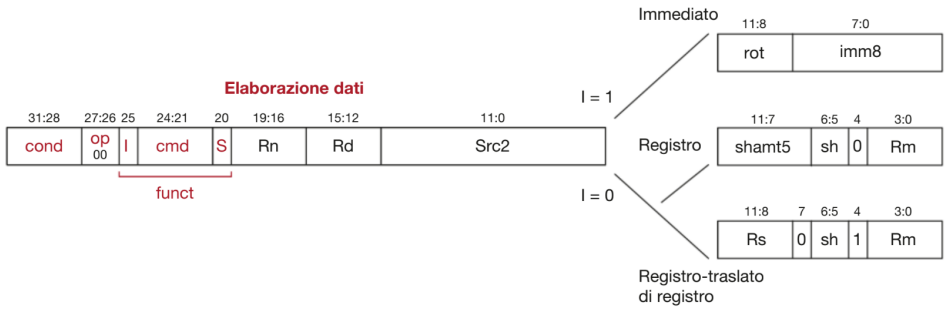
Abbiamo stabilito che le istruzioni dell'**ARM v7** sono rappresentate in 32 bit. Questa architettura però supporta anche una opzione **THUMB** (attivabile/disattivabile nella CPSR), che riduce le istruzioni ad una rappresentazione a 16 bit. Questo può risultare utile su **microcontrollori** che dispongono di memorie grandi 16/32 kb. Questo chiaramente ha delle conseguenze, infatti: solo 8 registri sono considerati utilizzabili, registro dest. e registro *src*1 sono lo stesso, non possiamo utilizzare istr. operative condizionali ecc...

Rappresentazione Esadecimale Se le istruzioni da 32 bit necessitavano di 8 cifre esadecimali, ora le istruzioni in linguaggio macchina saranno rappresentate solo in 4 cifre.

4.1.3 Descrizione Istruzioni per Categoria (OP, MEM, BRANCH)

Mostriamo la rappresentazione accurata delle istruzioni in base alla loro tipologia:

1. **Operazionale**: La composizione dei 32 bit in questione è:



   (a) **[31:28] bit**: Indica se l'istruzione è o meno condizionale.

   (b) **[27:26] bit**: Indica a quale categoria appartiene.

   (c) **[25] bit**: Indica se i bit $[11:0]$ rappresenteranno o meno un immediato.

   (d) **[24:21] bit**: Indica a quale istruzione stiamo facendo riferimento. (add, sub ...)

   (e) **[20] bit**: Indica se l'istruzione setterà o meno le flag, assumendo lo stesso comportamento di una **cmp**.

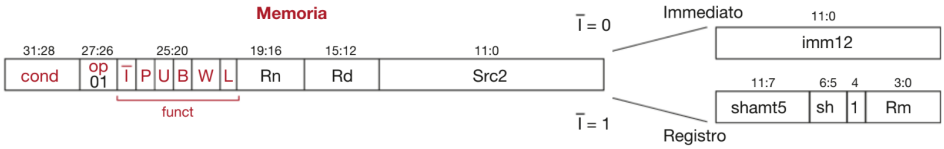
   (f) **[19:16] bit**: Primo registro sorgente.

   (g) **[15:12] bit**: Registro destinazione.

   (h) **[11:0] bit**: Sorgente 2, che può essere rappresentata da un registro o da un immediato.

   Notiamo nella rappresentazione di *registro-traslato* che quando il bit 4 è settato ad 1 abbiamo anche la possibilità di shiftare il contenuto del registro dato.

2. **Memoria**: La composizione dei 32 bit in questione è:



(a) **[31:28] bit**: Indica se l'istruzione è o meno condizionale.

(b) **[27:26] bit**: Indica a quale categoria appartiene.

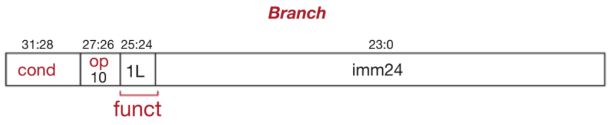
(c) **[25:20] bit**: Compongono l'effettiva istruzione di **ldr/str**, in particolare, la *L* ci dice se l'istruzione è di load o store, la *B* invece dice se vogliamo effettuare operazioni grandi 1 solo byte ecc... Di conseguenza questi 6 bit compongono la tipologia di istruzione effettiva.

(d) **[19:16] bit**: Primo registro sorgente.

(e) **[15:12] bit**: Registro destinazione.

(f) **[11:0] bit**: Sorgente 2, che può essere rappresentata da un registro o da un immediato.

3. **Salto**: La composizione dei 32 bit in questione è:



(a) **[31:28] bit**: Indica se l'istruzione è o meno condizionale.

(b) **[27:26] bit**: Indica a quale categoria appartiene.

(c) **[25:24] bit**: Il primo bit è sempre settato ad 1, mentre il secondo stabilisce se **B** oppure **BL**.

(d) **[23:0] bit**: 24 bit dedicati all'immediato a verso cui si effettuerà l'operazione di salto (solitamente un etichetta).

4.2 Ciclo del Processore

Una descritta la memoria ed i registri del processore, è necessario stabilire quali operazioni effettui la **CPU**. Possiamo dunque assumere che effettui questo:

```
1    while (true){
2        M[PC] -> istruzione
3        decodIstr <- decodifica(istruzione)
4        esecuzione(decodIstr)
5        scrivi_risultati
6        aggiorna(PC)
7    }
```

Dunque, riassumendo, abbiamo a disposizione questi elementi:

1. **CPSR**: Parte dei registri che mantengono le flag su carry, overflow, risultati dei confronti.
2. **PC**: Riferimento alla corrente istruzione.
3. **REG[16]**: Registri disponibili al processore.
4. **MEM**[ ]: Memoria dove depositare sorgente e dati.

Da qui in avanti utilizzeremo questi elementi per definire le categorie di istruzioni e la loro semantica.

Semantica ADD Mostriamo un esempio su un istruzione di *ADD*:

```
1    add R0, R1, R2
```

Questa istruzione somma i contenuti di $R1$ e $R2$ e deposita il risultato in $R0$. Dietro questo si "nasconde" questa semantica:

```
1    reg[0] = reg[1] + reg[2]
2    PC = PC + 4
```

Il +4 sul **Program Counter** indica il passaggio all'istruzione successiva, è indicabile anche con +"1", e vuol dire che stiamo andando avanti di 4 byte nella lista delle istruzioni, dato che un istruzione è grande esattamente 4 byte, ossia 32 bit. Una caratteristica dell'**Assembly ARM v7** è quella di **indirizzare al byte**. Questo motiva il +1, ossia andare avanti di un unità, ossia 4 byte.

4.3 Tipologie di Istruzioni

Nell'**Assembly ARM v7** esiste una classificazioni di istruzioni in base a che genere di operazioni eseguono. Elenchiamole:

1. **Istruzioni Operative**: Questo genere di istruzioni permettono di sommare, sottrarre, dividere o moltiplicare. Mostriamo degli esempi:

   (a) **Somma**: Sommiamo il contenuto in $R2$ e $R3$ depositando il risultato in $R1$.

   ```
   1    ADD R1, R2, R3
   ```

   (b) **Shift Logico a Sx**: Shiftiamo a sinistra (moltiplicazione per 2) il contenuto di $R2$ di k posizioni contenute in $R3$, depositando il risultato in $R1$.

   ```
   1    LSL R1, R2, R3
   ```

   (c) **Shift Aritmetico a Dx**: Shiftiamo a destra (divisione per 2) il contenuto di $R2$ di k posizioni contenute in $R3$, depositando il risultato in $R1$. Questo tipo di shift è detto **aritmetico** perchè tiene in considerazione del segno del numero sorgente mantenendolo anche dopo lo shift. Dunque pusha n bit (1 oppure 0) stabiliti dal contenuto $R3$ in base a se il numero originale fosse rispettivamente negativo o positivo.

   ```
   1    ASR R1, R2, R3
   ```

   (d) **Maschera e Cancellazione**: Dato $R1$ sorgente ed $R2$ maschera, questa istruzione negherà la maschera data e la porrà in AND con il numero sorgente. Questo causerà un eliminazione dei bit in posizione degli 1 nella maschera.

   ```
   1    BIC R1, R2, R3
   ```

   A questa classificazione appartengono anche le operazioni logiche come l'AND e l'OR.

2. **Istruzioni Memoria**: Le operazioni principali sono due, ossia rispettivamente caricare nei registri **(load)** e depositare nella memoria **(store)**.

   (a) **Load**:

   ```
   1    LDR R1, [indirizzo]
   ```

   (b) **Store**:

   ```
   1    STR R1, [indirizzo]
   ```

   Notiamo che la prima posizione $R1$ svolge due operazioni diversi tra la **load** e la **store**, infatti essendo in entrambi i casi un **registro**, nel primo caso svolge il ruolo di **destinazione** dato che stiamo caricando da memoria a registro, mentre nel secondo caso svolge il ruolo di **sorgente**. Bisogna ora analizzare il secondo parametro, ossia l'**indirizzo**, che può essere espresso in vari modi in base al tipo di indirizzamento scelto. Elenchiamo a pagina successiva tutte le possibili metodologie.

Tipi di indirizzamento Possiamo indirizzare in vari modi:

(a) **Indirizzamento Secco**: Mostriamo degli esempi contestualizzati ad uno **store**:

```
1    STR R1, [R2]
```

Questa metodologia è equivalente a dire $M[R[2]]$. Dunque stiamo prendendo il valore nel registro $R1$ e lo stiamo depositando nella memoria all'indirizzo corrispondente al valore del contenuto di $R2$.

(b) **Indicizzazione**: Mostriamo degli esempi contestualizzati ad uno **store**:

```
1    STR R1,[R2,R3]
```

Questa metodologia è equivalente a dire $M[R[2] + R[3]]$. Dunque stiamo prendendo il valore nel registro $R2$ e lo poniamo come **base**, mentre $R3$ sarà un **indice**. In questo modo è possibile effettuare un indicizzazione con somma tra basi ed indici senza ricorrere ad un operazione esplicita di ADD.

Stiamo dunque prendendo il valore nel registro $R1$ e quello in $R2$ e stiamo utilizzando la loro **somma** per accedere alla **corrispondente posizione** in **memoria** per depositare il valore di $R1$.

(c) **Post Incremento**: Anche in questo caso mostriamo un operazione di store:

```
1    STR R1,[R2],R3
```

Stiamo quindi:

i. Depositando in $M[R2]$ il valore presente in $R1$.

ii. Incrementando il valore di $R2$ con quello di $R3$, in questo modo:

$$R[2] += R[3]$$

(d) **Pre Incremento**: Anche in questo caso mostriamo un operazione di store:

```
1    STR R1,[R2,R3]!
```

Stiamo quindi:

i. Incrementando il valore di $R2$ con quello di $R3$, in questo modo:

$$R[2] += R[3]$$

ii. Depositando in $M[R2]$ il valore presente in $R1$.

3. **Istruzioni Salto**: Le due istruzioni principali di salto sono **BXX** (branch) e **BLXX** (branch and link).

   Etichette Condizionali delle istruzioni di salto Queste istruzioni necessitano delle etichette (**XX** presenti nelle istruzioni sopra elencate), che spesso valutano i valori delle flag presenti nella **CPSR** ed scelgono se effettuare o meno salti. Elenchiamo le potenziali etichette:

   (a) **EQ**: Equal, si verifica se la relativa flag nella **CPSR** sia o meno alzata.

   (b) **GT**: Greater Then, si verifica lo stato delle relativa flag nella **CPSR**.

   (c) **LT**: Less Then, si verifica lo stato delle relativa flag nella **CPSR**.

   (d) **NE**: Not Equal, si verifica lo stato della flag **N** nella **CPSR**.

   Una volta definite le varie etichette (labels), possiamo descrivere i due comandi di salto effettivi:

   (a) **Branch**: Il branch esegue semplicemente un aggiornamento dell'attuale program counter. Dunque:

   $$PC = PC + offset$$

   (b) **Branch and Link**: Il branch and link esegue un aggiornamento dell'attuale program counter, ma salva nel link register la posizione successiva a dove è stato invocato, in modo da poter tornare indietro successivamente, sostituendo ad un ipotetico **PC** successivo il corrente **LR**. Dunque:

   $$PC = PC + offset$$

   $$LR = PC + 4$$

Istruzioni Etichettate Mostriamo come sia possibile, anche per altri tipi di istruzioni oltre a quelle di salto, come sia possibile essere etichettate. Mostriamo esempio sulle **LDM** e **STR** (load e store multiple).

1. **Possibili Etichette**:

   (a) **F**: Full: Partire dall'indirizzo successivo.

   (b) **E**: Empty: Partire dall'indirizzo corrente.

   (c) **A**: Ascending: Sommare 4 al corrente indirizzo.

   (d) **D**: Discending: Sottrarre 4 al corrente indirizzo.

2. **Esempio**:

```
1    STMFD R1!, {R2, R3}
```

   Stiamo discendendo (sottraendo 4) e partendo dal successivo dell'indirizzo dato, ed ogni volta inseriamo il valore dei registri dati nei parametri.

4.4 Schemi di Traduzione: Pseudocodice - ASM ARM

Elenchiamo e descriviamo schemi noti di codice (if, while, for, ecc...) come traduzione da pseudocodice all'**ASM ARM V7**.

1. **If Then**: Mostriamo lo schema generico e successivamente un esempio di traduzione.

   (a) **Schema Generico**:

```
1         @ test condizione
2         BNE not_test
3         @ cmd_then
4     not_test:
5         @ cmd_ext
```

   (b) **Pseudocodice**:

```
1     if (x == 7) then x = x + 8
```

   (c) **ASM ARM v7**:

```
1         CMP r1, #7
2         BNE not_cond
3         ADD r1, r1, #8
4     not_cond:
5         ....
```

2. **If Then Else**: In questo caso esistono due schemi di traduzione specchiati, uno che salta nel **ramo then** e l'altro che salta nel **ramo else**.

   La caratteristica in comune tra i due schemi è quella che bisogna in ogni caso porre una seconda etichetta *oltreThen/oltreElse*.

   Questo perchè eseguendo un else, dovremmo skippare il corpo del then (saltando a *oltreThen*) e lo stesso vale per l'esecuzione di un then (saltando a *oltreElse*). Mostriamo lo schema generico e successivamente un esempio di traduzione.

   (a) **Schema Generico**:

```
1         @ test condizione
2         BEQ then
3         @ corpo else
4         B oltrethen //necessario a non eseguire sia else che then
5     then:
6         @ corpo then
7     oltrethen:
8         ....
```

(b) **Pseudocodice**:

```
if (x % 2 == 0) then x++
else x--
```

(c) **ASM ARM v7**:

```
        ANDS r1,r0,#1 //and e setto FLAG Z
        BNE else
        ADD r0,r0,#1
        B oltre_else
    else:
        SUB r0,r0,#1
    oltre_else:
        ....
```

3. **For**: Mostriamo lo schema generico e successivamente un esempio di traduzione. Assumiamo che il corpo del for venga eseguito almeno una volta.

(a) **Schema Generico**:

```
        @ inizializzazione indice
    loop:
        @ corpo
        @ incremento i
        @ check, se vero loop // BMI loop
```

(b) **Pseudocodice**:

```
sum = 0;
for(i = 1; i<n; i++){
    sum += i;
}
```

(c) **ASM ARM v7**:

```
        MOV r0, 1 //variabile sum
        MOV r1, 1 //indice
        //assumiamo che r2 contenga n
    loop:
        ADD r0, r0, r1 //somma
        ADD r1, r1, #1 //incremento indice
        CMP r1, r2 //controllo
        BMI loop
```

Se non fosse possibile assumere che si svolga almeno una iterazione del for basterebbe porre un etichetta aggiuntiva appena fuori il for, per poter effettuare un salto del corpo del ciclo in caso di controllo non superato.

4. **While**: Mostriamo lo schema generico e successivamente un esempio di traduzione.

   (a) **Schema Generico**:

```
        @ check guardia //se non passa B afterloop
    loop:
        @ corpo
        @ incremento i
        @ check, se vero loop // BMI loop
    afterloop:
        ....
```

   (b) **Pseudocodice**:

```
    while (x > 1) {x = x/2}
```

   (c) **ASM ARM v7**:

```
        MOV r0,#50 //assumiamo x = 50
        CMP r0,#1
        BLT afterloop
    loop:
        LSR r0, r0, #2
        CMP r0, #1
        BGT loop
    afterloop:
        MOV pc,lr
```

4.5 Branch a Contenuto di Registro

Seguendo uno stile delle istruzioni semplici illustrate fino ad ora, non esiste un vero e proprio comando che ci permetta di saltare al contenuto di un registro invece che ad una semplice etichetta. In realtà esiste un **BX** che sta per *Branch Exchange Mode*, ma non è coperto dal contenuto di queste dispense.

Utilizzeremo quindi una combinazione di due comandi semplici, ossia:

```
    salto_a_contenuto_reg:
        MOV lr,pc
        MOV pc,Rx
```

Questa combinazione ci permetterà di implementare anche funzioni **higher order**, dato che potremo effettuare un salto ad un indirizzo che abbiamo ad esempio ottenuto come parametro.

5 Microarchitetture

Questo capitolo si occuperà dell'implementazione effettiva di tutta la logica del capitolo precedente. Dunque sarà necessario fornire delle implementazioni su reti sequenziali di tutte le tipologie di istruzioni mostrate nel capitolo sull'architettura (livello assembler).

5.1 Tipologie di Parallelismo

Introduciamo delle tipologie di logica parallela, ossia quel tipo di calcolo che ci permette **parallelizzare** appunto le operazioni eseguite.

1. **Parallelismo Spaziale**: Questo tipo di parallelismo si basa sulla divisione di un operazione in tante piccole sotto operazioni indipendenti. Questo ci permetterà di assegnare le piccole operazioni ad **operatori** diversi. Possiamo rendere chiara l'idea con un esempio pratico. Immaginiamo di voler tradurre un libro di 1000 pagine. Questo costerà quindi 500h in termini di tempo. Assumendo che una persona p_1 possa tradurre 1 pagina in 30 minuti, come possiamo parallelizzare il suo lavoro?

   (a) La persona p_1 può dividere ad esempio il suo lavoro in 10 parti, dunque 10 parti da 100 pagine ciascuna. Assumiamo che questa operazione costi una quantità K in termini di tempo.

   (b) Assegnando le 10 parti a 10 persone diverse, e assumendo che ciascuna persona abbia una velocita di traduzione uguale a quella di p_1, è possibile notare che in un totale di 50h si ottiene lo stesso lavoro effettuato da una persona sola in 500h.

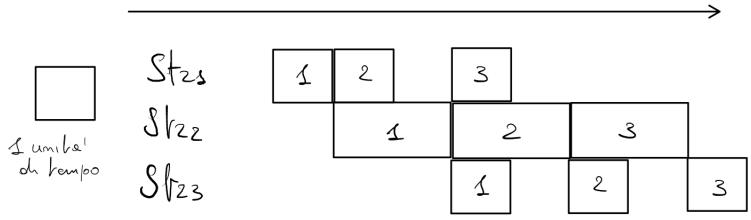
   (c) Dopo aver stabilito che il tempo impiegato per svolgere il lavoro si calcola con $\frac{T_{seq}}{n} = 50$h Bisogna però considerare anche il tempo K impiegato alla prima persona per suddividere il lavoro e quello per ricombinarlo. Dunque il tempo finale[1] sarà:
   $$\frac{T_{seq}}{n} + tempo_k + tempo_{ricombina}$$

2. **Parallelismo Temporale**: Questo tipo di parallelismo si basa invece su un concetto di **pipeline**, ossia una sorta di "catena di montaggio". Immaginiamo di avere a disposizione delle stazioni di lavoro. Ciascuna potrebbe avere un tempo di esecuzione diverso dalle altre.

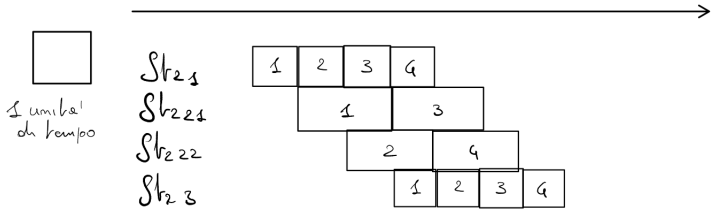
   Mostriamo e commentiamo degli esempi di pipeline a pagina successiva.

[1] Chiaramente i due tempi di divisione e ricombinazione non devono essere sostanziali, altrimenti non avrebbe alcun senso effettuare questa strategia di parallelismo.

(a) **Pipeline a 3 stazioni**: Immaginiamo di avere 3 stazioni di lavoro che elaborano operazioni. Quando un operazione viene elaborata dalla prima stazione viene scaricata nella seconda e così via fino alla completa elaborazione dell'operazione in merito.



(b) **Pipeline combinata al Parallelismo Spaziale**: Immaginiamo di splittare la seconda stazione, in due stazioni diverse, ossia Str_{21} ed Str_{22}.



Notiamo che in questa versione otteniamo un ottimizzazione dei tempi. Bisogna però categorizzare i tempi citati in questa soluzione.

5.1.1 Costi in Tempo nel Parallelismo

Analizziamo i costi delle due tipologie di parallelismo:

1. **Costo Parallelismo Spaziale**: Come già stabilito nell'esempio della traduzione del libro:

$$\frac{T_{seq}}{n} + tempo_k + tempo_{ricombina}$$

2. **Costo Parallelismo Temporale**: Definiamo un t_i ossia i-esima stazione, ed un m numero di beni da produrre. La formula sarà:

$$T \approx (max\{t_i\} * m) + (t_{inizio} + t_{fine})^2$$

[2]Queste quantità saranno trascurabili, essendo il numero di stazioni solitamente molto minore dei beni da produrre.

5.1.2 Latenza e Tempo di Servizio in Parallelismo Temporale

Stabiliamo le differenze tra queste tipologie di tempo, caratteristiche del Parallelismo Temporale:

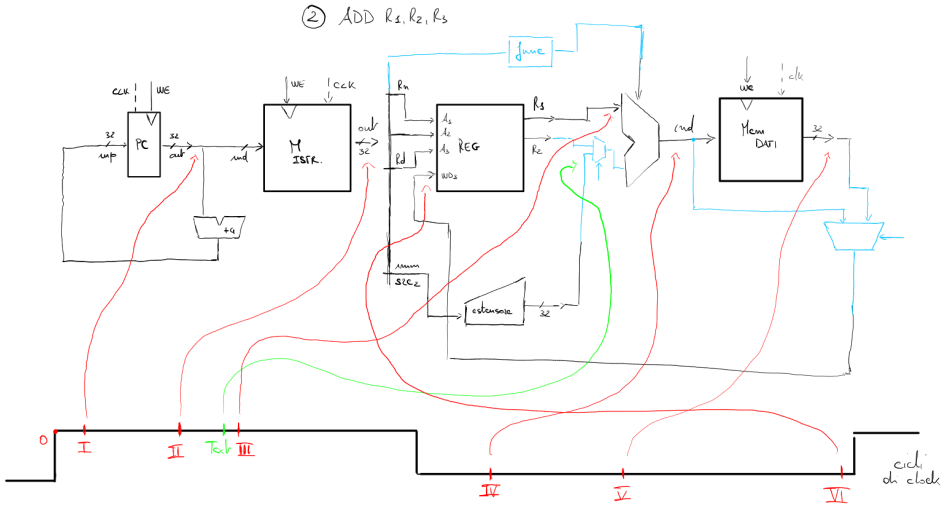
1. **Latenza (L)**: Corrisponde alla differenza tra il tempo di fine di un task e il suo tempo d'inizio:
$$L = t_{fine} - t_{inizio}$$

2. **Tempo di Servizio T_s**: Corrisponde al tempo che intercorre tra fine di un task e il suo successivo.
$$T_s = |t_{pred_i} - t_{pred_{i+1}}|$$

5.2 Esempi di Implementazione Istruzioni Su Processore Single Cycle

Mostriamo l'implementazione di alcune istruzioni del livello **assembler** analizzandone i tempi rispetto al ciclo di clock.

1. **Istruzione Operativa**: Mostriamo la rete e descriviamone lo sviluppo temporale:



   (a) **Da 0 ad 1**: Tempo di accesso al program counter.

   (b) **Da 1 a 2**: Tempo di accesso alla memoria istruzioni.

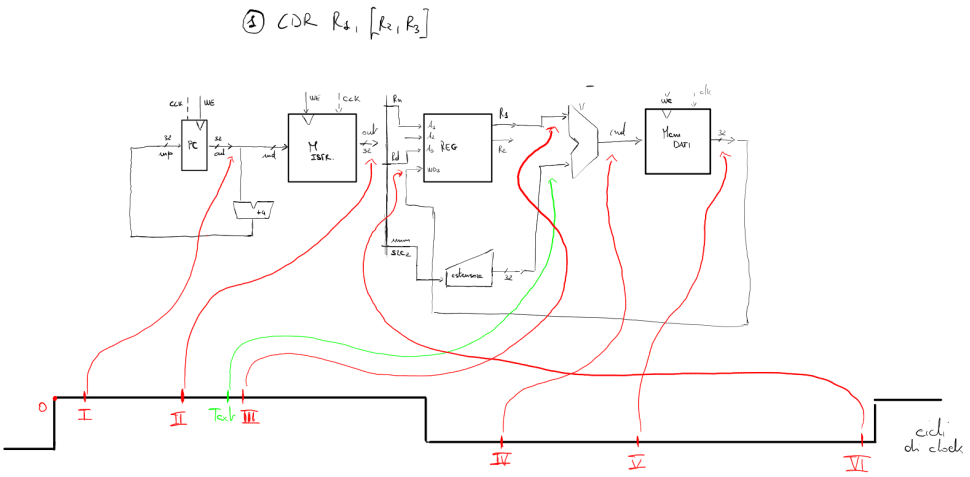
   (c) **Da 2 a 3**: Tempo di accesso ai registri e tempo di stabilizzazione dell'estensore (in verde).

   (d) **Da 3 a 4**: Tempo di stabilizzazione dell'ALU.

   (e) **Da 4 a 5**: Tempo di accesso alla memoria dati.

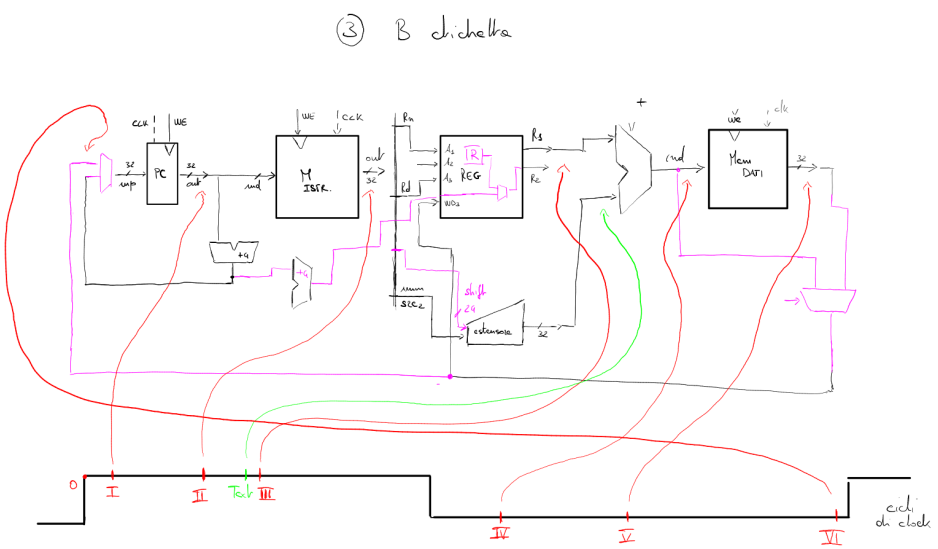
   (f) **Da 5 a 6**: Tempo in cui si fornisce in input al registri l'output della memoria dati.

2. **Istruzione di Memoria**: Le fasi evidenziate sono simili alla precedente tipologia di operazioni.



Possiamo però varie differenze sui wire e sulla mancanza di alcune componenti che nelle istruzioni operative risultavano necessarie.

3. **Istruzione di Salto**: Mostriamo la rappresentazione:



Notiamo la presenza di alcuni **multiplexer** necessari per rendere il circuito in grado di processare gli immediati, oppure per poterli shiftare.

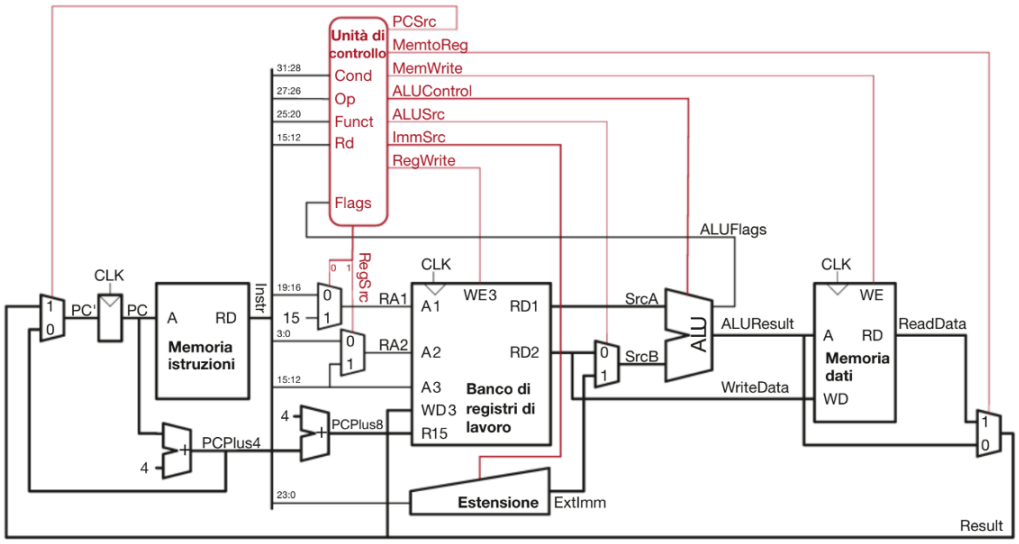
5.3 Processore Single Cycle

Immaginiamo un istruzione *ist* in ASM ed elenchiamo tutte le fasi attraversate dal processore per elaborarla:

1. **Fetch**: Si cerca l'istruzione ASM.
2. **Decode**: Decodifica per il processore (ad esempio da istruzione ASM ad esadecimale).
3. **Execute**: Elaborazione dell'istruzione in questione.
4. **Memory**: Modifiche della memoria apportate dall'elaborazione dell'istruzione (se necessario).
5. **Write Back**: Fase che permette la "transizione" all'istruzione successiva.

La caratteristica della **CPU Single Cycle** è quella di coprire tutte queste fasi in un **singolo ciclo** di clock. Questo causa cicli di clock molto ampi:

$$\tau_{SINGLE} \approx 740 \text{ picosecondi}$$



$$\tau \text{ (ciclo di clock)} = \{2t_a + t_{ALU} + 2t_{REGISTRI} + 4t_{MUX}\}$$

Ogni istruzione sarà dunque eseguita in un τ, ma un τ molto ampio.

5.4 Processore Multi Cycle

Questo tipo di processore mira a risolvere delle problematiche del processore single cycle, ossia:

1. Cicli di clock **molto ampi**, che coprivano tutti i tipi di operazioni, anche quelle più lente come le LOAD.
2. Tre circuiti sommatori **diversi**.
3. **Memoria** dati ed istruzioni **separate**.

Questo tipo di **CPU** suddivide le operazioni principali in vari **"stadi"** grazie a dei registri non architetturali che permettono di preservare i dati durante la transizione da uno stadio all'altro. Questo approccio renderà il costo in τ dipendente dal tipo di operazione effettuata. La parte di controllo è ancora una rete combinatoria.

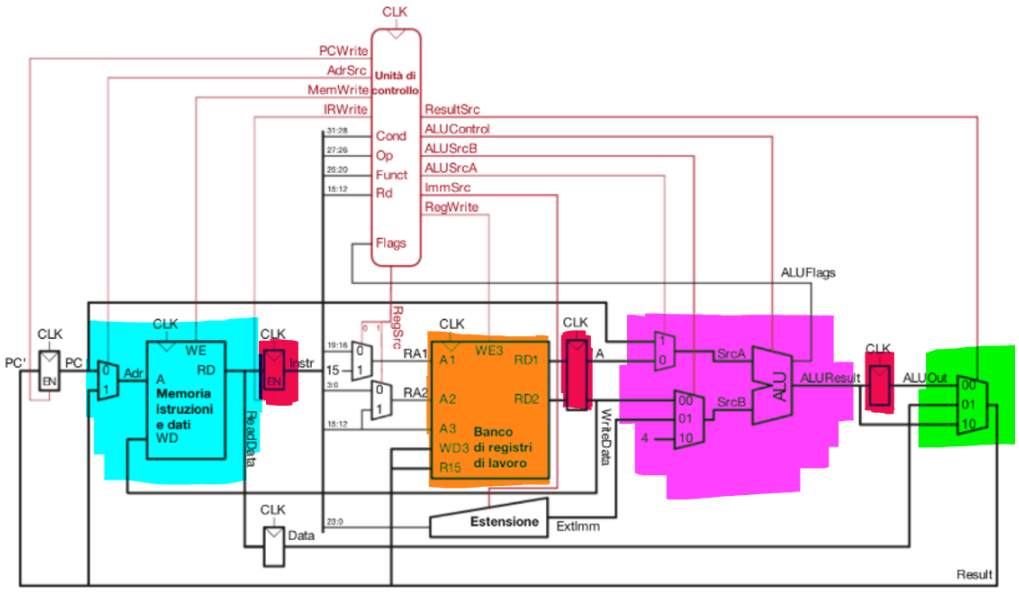


Figure 2: Illustrati in rosso i registri non architetturali, mentre tutti gli altri stadi in colori differenti tra loro

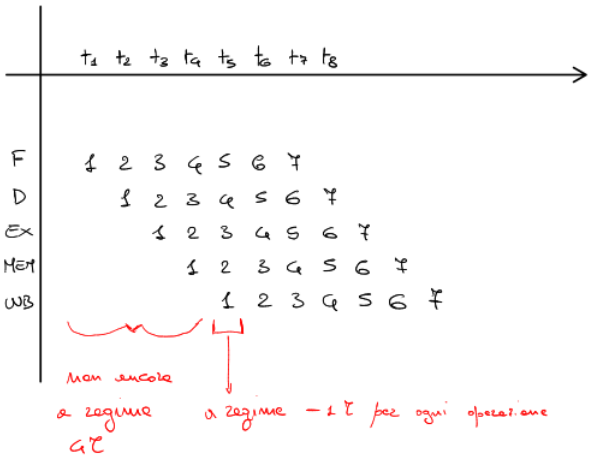
$$\tau = max\{t_a,\ t_{PC} + t_{MUX} + t_{REGISTRI},\ t_a,\ t_{MUX} + t_{REGISTRI}\}$$

$$\tau_{MULTI} \approx 340 \text{ picosecondi}$$

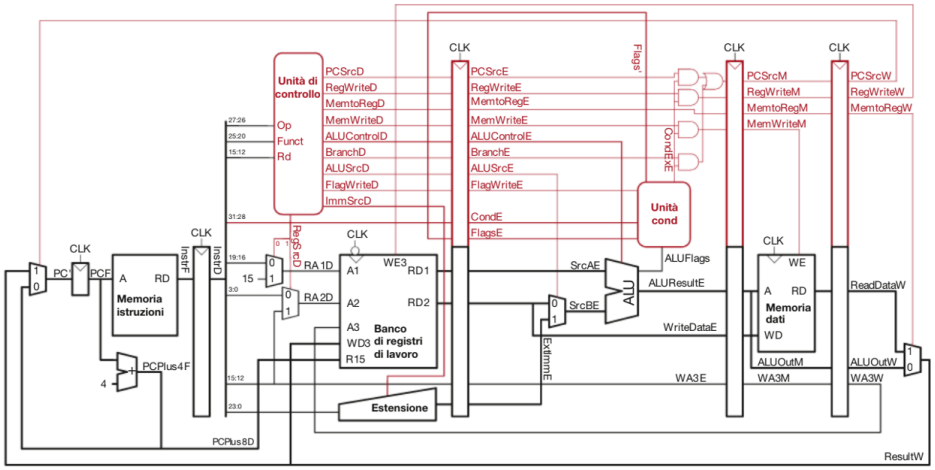
tipo operazione	costo in cicli di clock
STR	4τ
LDR	5τ
BR	4τ
OP	4τ

5.5 Processore Pipeline

Questo tipo di processore cerca di sfruttare il parallelismo temporale, permettendo alle varie istruzioni di essere elaborate in stadi "paralleli". Mostriamo una rappresentazione dello schema logico di pipeline:



Questa tipologia di schema permette, una volta raggiunta un esecuzione **a regime**, di elaborare un istruzione in $1\overline{\tau}$ ciascuna, indipendentemente dal tipo. Ovviamente $\overline{\tau} << \tau$, quindi il ciclo di clock della CPU **pipeline** è molto meno ampio di quella in **single cycle**.



Notiamo che nel processore a pipeline la zona di controllo è una **rete sequenziale**. E' infatti necessario che anche la zona di controllo possa mantenere informazioni tra i vari stadi grazie a dei registri non architetturali.

Ricordiamo che ogni istruzione causa un cambio di stato alla memoria complessiva. Di conseguenza due istruzioni potrebbero essere **dipendenti** tra loro in questo schema. Vanno dunque gestite queste situazioni.

5.5.1 Dipendenze Logiche - Condizioni di Bernstein

Stabiliamo le dipendenze tra istruzioni consecutive grazie alle **Condizioni** di **Bernstein**:

$$\text{Se } \; R_1 \cap W_2 = \emptyset \;\; , \;\; R_2 \cap W_1 = \emptyset \;\; , \;\; W_1 \cap W_2 = \emptyset$$

$$\text{allora } \; OP_1; OP_2 \equiv OP_2; OP_1$$

Ossia, possiamo invertire l'ordine di due istruzioni generiche se queste non spezzano le Condizioni di Bernstein.

Forwarding - Stalli in risoluzione alle dipendenze Bisogna dunque gestire il problema delle dipendenze logiche nella pipeline. La più semplice delle soluzioni è quella di "distanziare" le istruzioni dipendenti tra loro, dando tempo alla prima istruzione di essere completamente elaborata. Mostriamo un implementazione fisica della rete che permette la creazione di questi stalli:

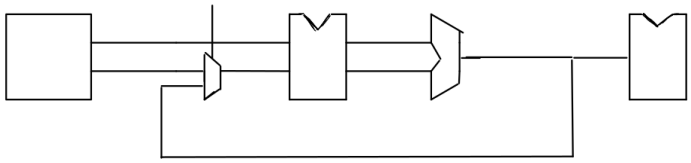
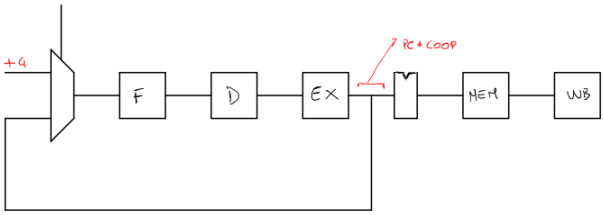


Figure 3: L'esito dell'ALU viene portato in ingresso ad un MUX che decide o meno di portare in ingresso al registro non architetturale in base al segnale fornito.

Grazie alla creazione di stalli ed alla scrittura o meno dei **write enable** riusciamo a risolvere la gran parte delle dipendenze logiche.

5.5.2 Dipendenze sul Controllo

Anche le **istruzioni** di **salto** creano problemi alla pipeline, dato che si starebbero elaborando **istruzioni da ignorare** successivamente.



Possiamo eseguire una **sorta di forwarding** tra l'uscita dell'Execute e l'entrata della Fetch per **stabilire o meno** l'**inserimento** nella **pipeline** delle nuove istruzioni. Questo permette la risoluzione delle dipendenze sul controllo.

$$\text{Tempi senza Dipendenze: } \; m\tau + 4\tau$$

$$\text{Tempi con Dipendenze: } \; m\tau + (\#stalli\_dip\_logiche)\tau + (\#salti\_presi)2\tau$$

In realtà tutti i segnali di controllo inviati ai **MUX** per regolare l'utilizzo dei forwarding vengono gestiti dall'**hazard unit**.