APPUNTI di
PRINCIPI DI ARCHITETTURE DEI CALCOLATORI
(corso del Prof. Zaccaria Vittorio)
Pietro Giannoccaro

ATTENZIONE

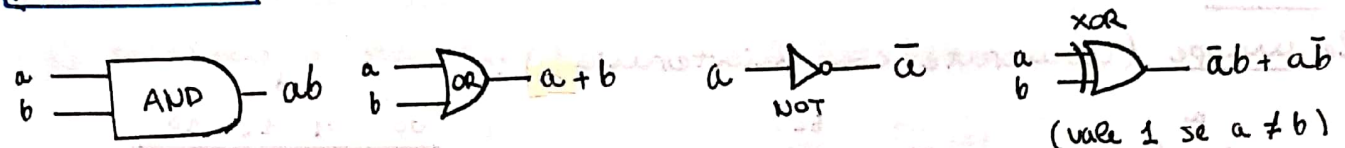
Quello che segue è un compendio del corso di Principi di Architetture dei Calcolatori, presso la facoltà di Ingegneria Elettronica al Politecnico di Milano. Il documento si propone di riassumere le nozioni necessarie allo svolgimento degli esercizi del corso. NON garantisce tuttavia la presenza di tutto ciò che è stato spiegato a lezione, né la correttezza del contenuto. Si invitano pertanto i lettori a consultare il testo di riferimento.

Il seguente documento non sostituisce in alcun modo lo studio dal libro, dagli appunti e la frequenza delle lezioni.

Se desiderate condividere questo documento o per eventuali reclami o correzioni contattate la seguente mail:
pietro.giannoccaro@mail.polimi.it

LOGICA BOOLEANA e CIRCUITI LOGICI BASILARI

PORTE LOGICHE:



PROPRIETÀ LOGICHE PIÙ USATE:

ELEM. NEUTRO	$a+0=a$; $a\cdot 1=a$
DISTRIBUTIVITÀ OR	$a+bc=(a+b)(a+c)$
COMPLEMENTO	$a+\bar{a}=1$; $a\bar{a}=0$
IDEMPOTENZA	$aa=a$; $a+a=a$ → molto usata!
ASSORBIMENTO	$a+ab=a$; $a(a+b)=a$
DE MORGAN	$\overline{ab}=\bar{a}+\bar{b}$; $\overline{a+b}=\bar{a}\cdot\bar{b}$ → molto usata!

COSTI E PRESTAZIONI:

CARDINALITÀ: numero PORTE LOGICHE utilizzate (anche a più di 2 ingressi)

LIVELLI: numero LIVELLI gerarchia

Nº INPUT: OGNI PORTA LOGICA a N INGRESSI VALE N

POS e SOP:

Rappresentazioni di espressioni booleane:

- SOMMA DI PRODOTTO (SOP): $ac+cb$
- PRODOTTO DI SOMMA (POS): $(x+y)(z+a)$

PRIMA FORMA CANONICA:

è una SOP di tutti (e soli) termini prodotto delle variabili di ingresso corrispondenti agli 1 della funzione

MINTERMINE: m_i FUNZIONE LOGICA BOOLEANA a N INGRESSI che vale 1 in corrispondenza della sola configurazione di ingresso che vale i.

*NOTA: $x=0 \Rightarrow \bar{x}$ nel mintermine
$x=1 \Rightarrow x$

	a	b	$g(a,b)$	$m_1(a,b)$	$m_3(a,b)$
	0	0	0	0	0
m_1 →	0	1	1	1	0
	1	0	0	0	0
m_3 →	1	1	1	0	1

➡ $g = m_1 + m_3 = \bar{a}b + ab$

SECONDA FORMA CANONICA:

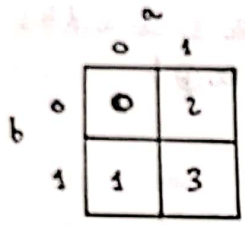
è una POS con maxtermini M_i (dove g è zero)

	a	b	$g(a,b)$	M_0	M_2
→	0	0	0	0	1
	0	1	1	1	1
→	1	0	0	1	0
	1	1	1	1	1

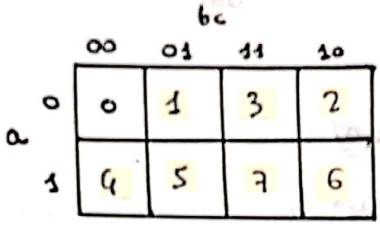
➡ $g = M_0 \cdot M_2 = (a+b)(\bar{a}+b)$

→ MINIMIZZAZIONE MAPPE DI KARNAUGH :

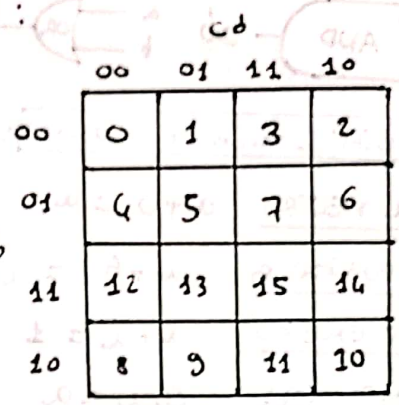
- e' ON_{SET} ci da i mintermini (dove f vale 1)
- le mappe (numerate con mintermini):



2 bit



3 bit



4 bit

IMPLICANTE: RAGGRUPPAMENTO RETTANGOLARE CONTENENTE ALMENO DUE UNO CONTIGUI (PUÒ CONTENERE SOLO 1!)
($\bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}b\bar{c}\bar{d}$ 2 mintermini, $\bar{a}\bar{c}\bar{d}$ implicante)

IMPLICANTE PRIMO: ie RAGGRUPPAMENTO CORRISPONDENTE è di DIMENSIONE MASSIMA (DIM: 1x2, 2x2, 1x4, 2x4)

IMPLICANTE PRIMO ESSENZIALE: se è l'UNICO A CONTENERE ALCUNI MINTERMINI (COPRE UN 1 NON COPERTO DA NESSUN ALTRO)

COPERTURA OTTIMA: insieme di implicanti che
a) contiene tutti gli implicanti primi essenziali
b) non contiene implicanti primi totalmente coperti da una unione di quelli essenziali
c) ie più piccolo dei sottoinsiemi

FORMA MINIMA: somma implicanti copertura scelta (si osservano i termini che non variano)

DON'T CARE: CONDIZIONE DI INDIFFERENZA (VALORE NON NOTO)
- non deve essere necessariamente coperta da un implicante, ma può essere se conviene

NON IMPLICANTE: contiene solo DON'T CARE.

Per trovare insieme più conveniente si analizza **ALBERO DI DECISIONE** e si sceglie copertura con meno letterali.

MINIMIZZAZIONE con METODO QUINE-MCCLUSKEY:

FASE 1: RICERCA IMPLICANTI PRIMI

1) seleziono mintermini funzione e li intabello

a	b	c	f
0	0	0	1 ←
0	0	1	0
0	1	0	0
0	1	1	1 ←
1	0	0	0
1	0	1	0
1	1	0	1 ←
1	1	1	1 ←

→ ON_{SET} ; DC_{SET}

impl. numero	a	b	c
(0)	0	0	0
(3)	0	1	1
(6)	1	1	0
(7)	1	1	1

2) cerco le coppie che differiscono di 1 bit. Ottengo per riduzione gli implicanti di ordine 1. (conviene partire dalla prima e quelle dopo, poi 2ª e quelle dopo e così via!)
Gli implicanti che NON partecipano alla riduzione sono primi, e li marchio con •.

		a	b	c
•	(0)	0	0	0
	(3)	0	1	1
	(6)	1	1	0
	(7)	1	1	1

IMPL. ORDINE ZERO

RIDUZIONE

(3,7) → - 1 1
(6,7) → 1 1 -

		a	b	c
•	(3,7)	-	1	1
•	(6,7)	1	1	-

IMPL. ORDINE 1

FASE 2: COPERTURA OTTIMA

3) Creo tabella con

INDICI RIGA: IMPLICANTI PRIMI
INDICI COLONNA: MINTERMINI APPARTENENTI a ON_{SET}
ELEMENTI MATRICE:
- • impl. essenziale
- o impl. copre mintermine ma non è l'unico a farlo

vuoto altrimenti

impl.	0	3	6	7
(3,7)		•		o
(6,7)			•	o
(0)	•			

4) semplifico:

→ CRITERIO ESSENZIALITÀ: rimuovo dalla tabella IMPLICANTI ESSENZIALI (riga) e le colonne da essi coperte
Gli IMPLICANTI RIMOSSI faranno parte della soluzione.

→ **CRITERIO DOMINANZA**: si eliminano RIGHE (implicanti) o COLONNE (mintermini) completamente contenute in altre righe/colonne.
(Mintermini possono diventare essenziali•!)

DOMINANZA DI RIGA:

impe	m1	m4	m8	m10
P_0	o			o
P_1	o		o	
P_2		o		o
P_4	o	o	o	

diventa essenziale •

Applica tutti quelli possibili prima di passare alla iterazione successiva

DOMINANZA TRA COLONNE:

impe.	m1	m4	m8	m10
P_0	o			o
P_1	o		o	
P_2		o		o
P_4	o	o	o	

m_8 domina m_1

m_i domina m_j se m_j è coperto da tutti gli implicanti che coprono anche m_i e qualcuno in più.

5) Arrivati ad una tabella non riducibile, si procede con le **DIAGRAMMA AD ALBERO di DECISIONE**

CONDIZIONI DI INDIFFERENZA - DON'T CARE:

- si considerano DC come 1
- nella TABELLA di COPERTURA compaiono come indici colonna solo i mintermini appartenenti all'ON-SET
- termini con solo DC non sono implicanti della funzione.

FUNZIONI SEQUENZIALI - SIMULAZIONE CIRCUITI:

Nella simulazione dei circuiti è fondamentale analizzare la TOPOLOGIA per ricavare le RELAZIONI LOGICHE TRA I SEGNALI IN GIOCO.

Di solito si ASSUMONO NULLI i RITARDI DI PROPAGAZIONE.

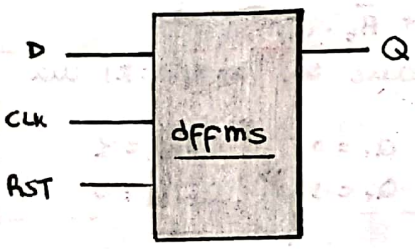
È indispensabile poi conoscere alcuni componenti circuitali "complessi" riassunti sotto.

Sfruttando dunque segnali di partenza e segnali parziali si ottiene poco alla volta la simulazione completa, seguendo un ordine cronologico.

Si noti come si disegna sempre salita/discesa leggermente inclinate.

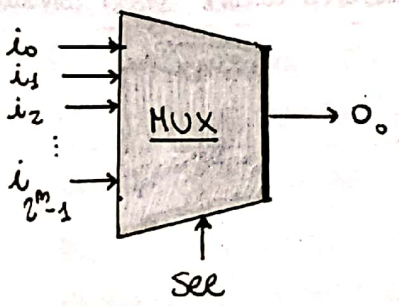
COMPONENTI CIRCUITALI:

- **FLIP/FLOP:**



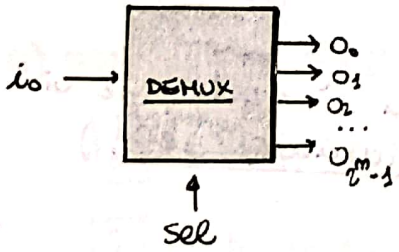
- Q assume il valore di D all'ISTANTE DI DISCESA DEL CLOCK e LO MANTIENE PER TUTTO IL PERIODO DI CLOCK SUCCESSIVO. Se D cambia durante discesa clock non ha effetto.
- CLK è il clock
- RST: SEGNALE DI RESET, se 1 PORTA D e Q A ZERO

- **SELETTORE - MULTIPLEXER:**



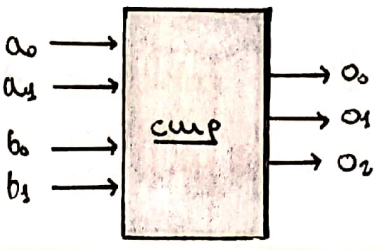
- sel è un entrata a m bit che seleziona un determinato ingresso, che viene fornito in uscita.

- **DESELETTORE - DEMULTIPLEXER:**



- l'uscita o_i selezionata da sel assume valore i_0
- se $i_0 = 1$ il DEMUX è anche detto DECODER

- **COMPARATORE:**



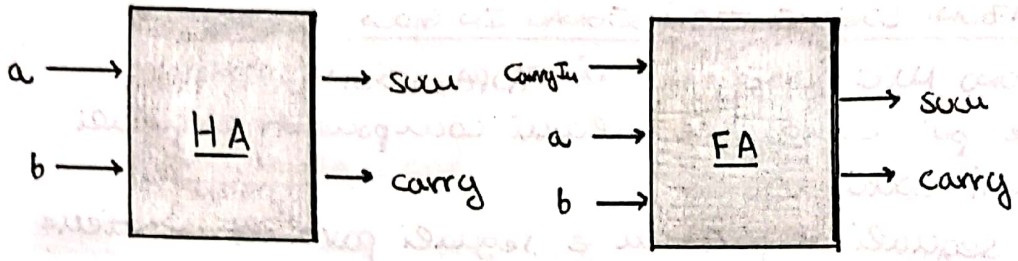
- dati due operandi a e b

O_0 è vero se $a < b$

O_1 è vero se $a = b$

O_2 è vero se $a > b$

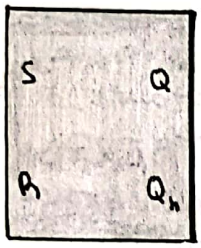
- **HALF ADDER - FULL ADDER:**



- dati due bit, a e b in ingresso viene effettuata la somma e riportato carry.
- FA tiene conto di carry In in ingresso

ELEMENTI DI MEMORIA

- **BISTABILE SR:**



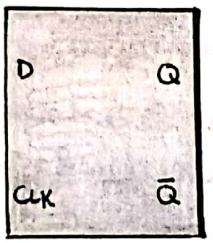
S_t	R_t	$Q_{t+\Delta t}$
0	0	Q_t
0	1	0
1	0	1
1	1	Non def

- SET, RESET hanno effetto su Q dopo Δt

$$Q_{t+\Delta t} = S_t + \bar{R}_t \cdot Q_t$$

- **FLIP/FLOP JK:** come SR (S:J, R:K) ma per $J,K = 1,1$ se $Q_t = 0$ → $Q_{t+\Delta t} = 1$; se $Q_t = 1$ → $Q_{t+\Delta t} = 0$

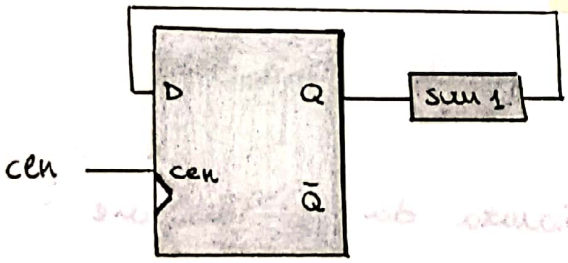
- **BISTABILE TIPO D:**



Clk	D_t	$Q_{t+\Delta t}$
0	0	Q_t
0	1	Q_t
1	0	0
1	1	1

- ha effetto se ie clk è alto
- **FLIP/FLOP T:** come D, però quando T = 1 inverte valore stato corrente

- **CONTATORE**



*NOTA: il rst alto porta i Q a ZERO, ma non è detto che vi porti anche i D (i D non sono necessariamente zero!)

MACCHINE A STATI

- MACCHINA DI MEALY:

1) Osservo il grafo, dove possiamo trovare gli stati possibili e le transizioni da uno stato all'altro con i parametri IN/OUT ovvero i valori che devono assumere l'ingresso e l'uscita.
   ↳ AD OGNI STATO Si è ASSOCIATA UNA CODIFICA BINARIA (di solito corrispondente all'indice)

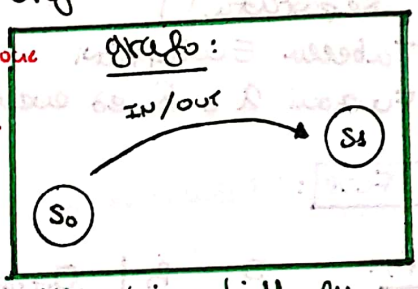
2) Ricavo TABELLA TRANSIZIONI e USCITE, così organizzata:

(esempio)

Q_0	Q_1	I_0	Q_0^*	Q_1^*	O_0
0	1	0	1	0	0
CODIFICA BINARIA STATO INIZIALE		BIT DI INGRESSO (IN)	CODIFICA BINARIA STATO FINALE		BIT DI USCITA ASSOCIATO (OUT)

← attenzione ordine indici!

per tutte le COMBINAZIONI DEL GRAFO



macchina di Mealy

Se non è presente una data transizione tra due stati, in corrispondenza della transizione sulla tabella si pone una X di DON'T CARE.

* NOTA: I_0 e O_0, ovvero IN/OUT fanno si che una transizione avvenga

3) In base al tipo di FLIP-FLOP (SR, D, T, JK) che scegliamo per realizzare la macchina, ricaviamo la TABELLA DELLE ECCITAZIONI, ovvero sostituiamo nella tabella su Q^* con i PARAMETRI CHE IL FLIP/FLOP DOVREBBE AVERE per REALIZZARE Q^*, ovvero se FLIP-FLOP è SR, gli S_0 e R_0 che realizzano Q_0^*, tenendo conto del valore iniziale Q_0 ed eventuali DON'T CARE.

*NOTA: PER OGNI BIT di Q e Q^* vi È UN FLIP-FLOP sia nella TABELLA DELLE ECCITAZIONI CHE NELLA TOPOLOGIA FINALE
(es. 4 stati: 2 bit ⇒ Q_0, Q_1; Q_0^*, Q_1^*; S_0, R_0, S_1, R_1 se FFSR)

4) osservando le tabelle appena ottenute, ricavo δ e λ (funzioni):

δ: parametro FF = funzione con Q_i, I_i (per OGNI parametro ingresso degli FF)

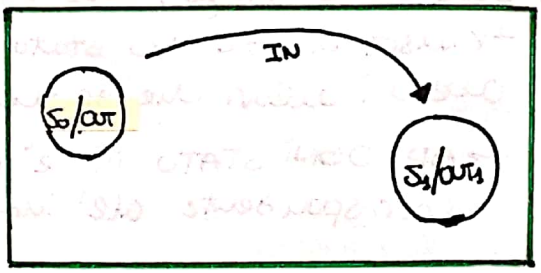
λ: bit uscita O_i = funzione con Q_i, I_i

(POTREBBE CONVENIRE USARE KARNAUGH!)

5) A partire da λ e δ, disegno la TOPOLOGIA, sfruttando le PORTE LOGICHE NOTE (AND, OR) e INCLUDENDO NEGLI INGRESSI DEI FLIP-FLOP (TANTI QUANTI SONO I BIT NECESSARI PER LA CODIFICA!) ANCHE IL SEGNALE di RESET (rst) e CLOCK (clk).

- **MACCHINA DI MOORE:**

Con la macchina di MOORE, cambia poco; sulle frecce che indicano le transizioni sono riportati gli ingressi per eseguirle.
Accanto allo stato è riportata l'UNICA USCITA.



macchina di Moore

Dunque, scelti:
- CODIFICA DEGLI STATI
- FLIP-FLOP

Ricavo
1) Tabella TRANSIZIONI
2) Tabella USCITE (questa volta conviene separata!)
3) Tabella ECCITAZIONI
4) Funzioni λ e δ ed eventualmente TOPOLOGIA.

→ **FLIP-FLOP:**

→ SR:

S	R	$Q_{t+\Delta t}$
0	0	Q_t
0	1	0
1	0	1
1	1	non def.

→ JK:

J	K	$Q_{t+\Delta t}$
0	0	Q_t
0	1	0
1	0	1
1	1	$\overline{Q_t}$

→ D:

D	$Q_{t+\Delta t}$
0	0
1	1

→ T:

T	$Q_{t+\Delta t}$
0	Q_t
1	$\overline{Q_t}$

* NOTA: conviene sempre scriversi di fianco la struttura del FLIP-FLOP utilizzato, e i valori necessari per le transizioni

es. (FLIP-FLOP SR)

Q_t	$Q_{t+\Delta t}$	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

↝ Si fa molto più velocemente a scrivere la tabella delle eccitazioni!!! Basta ricopiare SR per ogni transizione

LINGUAGGIO MACCHINA e ASSEMBLY

ISTRUZIONI:

• ARITHETICO / LOGICHE:

		SIGNIFICATO	
ADD SUB	rd, x2, x3	rd = x2 ± x3	SOMMA / DIFFERENZA
ADDi SUBi	rd, x2, imm	rd = x2 ± imm	SOMMA / DIFFERENZA
MV	rd, x2	rd = x2	COPIA
SLLI SRLi	rd, x2, imm	AGGIUNGE ZERO SHIFTA LEFT/RIGHT x2 e salva in rd ⇒ MOLTIPLICAZIONE: $rd = 2^{imm} \cdot x2$; DIVISIONE: $rd = x2 / 2^{imm}$	
AND ANDI OR ORi XOR XORi	rd, x1, x2/imm	Salva in rd il risultato 1 o 0 di x1 AND / OR / XOR x2/imm	OPERAZIONI LOGICHE
SRAi	rd, x2, imm	SHIFTA A DX MA PRESERVA SEGNO BIT PIÙ SIGNIFICATIVO	

*NOTA: una WORD è di dimensione 4 BYTE = 32 BIT (1 BYTE = 8 BIT)

• TRASFERIMENTO DATI - LOAD / STORE:

		SIGNIFICATO	
LA	rd, b	rd = &b	CARICA INDIRIZZO Salva in rd l'indirizzo di b (variabile)
LD LH LW	rd, offset(a0)	(DOUBLE WORD, WORD, HALF-WORD) rd = M[a0 + offset]	Carica in rd, il contenuto di off(a0) (di solito a0 : indirizzo) CARICA PAROLA
Li	rd, imm	rd = imm	CARICA IMMEDIATO
SW SD SH	s1, offset(a0)	(WORD, DOUBLE WORD, HALF WORD) M[a0 + offset] = s1	Salva all'indirizzo off(a0) s1 SALVA PAROLA A INDIRIZZO

• ISTRUZIONI DI SALTO:

BEQ BNE BLT BLE BGE BGT	x1, x2, label	per usare confronto con imm occorre caricarlo con li	SALTA A LABEL SE x1 == x2 (BEQ) x1 != x2 (BNE) x1 < x2 (BLT) x1 <= x2 (BLE) x1 >= x2 (BGE) x1 > x2 (BGT) SALTO CONDIZIONATO
JALR	rd, offset(a0)		SALTA A OFFSET(A0) e SALVA INDIRIZZO ISTRUZIONE SUCCESSIVA in RD
JAL	rd, offset		SALTA A OFFSET RISPETTO A PROGRAM COUNTER e SALVA INDIRIZZO ISTRUZIONE SUCCESSIVA in un REGISTRO,
J	etichetta		SALTA A ETICHETTA

→ WHILE, FOR, IF, SWITCH

FOR:

```
for (inizio; cond; fine) {
    corpo;
}
```

EQUIVALE A

WHILE:

```
inizio;
while (condizione) {
    corpo;
    fine;
}
```

IF-THEN-ELSE:

```
if (condizione) {
    corpo Then;
} else {
    corpo Else;
}
```

SWITCH:

```
switch (a) {
    case 0:  corpo; break;
    case 1:  corpo; break;
    default: corpo;
}
```

Questi esercizi sono pura logica, sfruttando le istruzioni di salto

→ CHIAMATA con INDIRIZZO COSTANTE - LUI

Quando l'indirizzo virtuale di una variabile x non è rappresentabile su 12 BIT (NEGLI ESERCIZI ACCADE SEMPRE CON VETTORI DI DATI!) ipotizzando sia rappresentabile su 32 bit si usa:

A)
```
LUI  rd, %hi(x)
ADDI rd, rd, %lo(x)
```

PRENDE i 20 BIT più SIGNIFICATIVI dell'INDIRIZZO di x e pone i 12 BIT MENO SIGNIFICATIVI pari a ZERO. Poi SOMMA i 12 BIT MENO SIGNIFICATIVI e ASSEGNA TUTTO A rd

CONTIENE INDIRIZZO di x (non valore contenuto)

B)
```
AUIPC rd, %pcrel_hi(x)
ADDI  rd, rd, %pcrel_lo(x)
```

FA LA STESSA COSA di LUI, MA più SCOMODO.

→ DIMENSIONI E TIPI DI DATO:

La specifica RISC-V prevede varie dimensioni di dato e diverse chiamate: (quando ci muoviamo su indirizzi con una addi l'unità è intesa in byte)

DIMENSIONE	NOME	SUFFISSO
8 bit	BYTE	b
16 bit	HALF-WORD	h
32 bit	WORD	w
64 bit	DOUBLE WORD	d

DIMENSIONI in C:

CHAR - 1 byte
INT - 2/4 byte
LONG INT - 8 byte

uint64_t : 8 byte

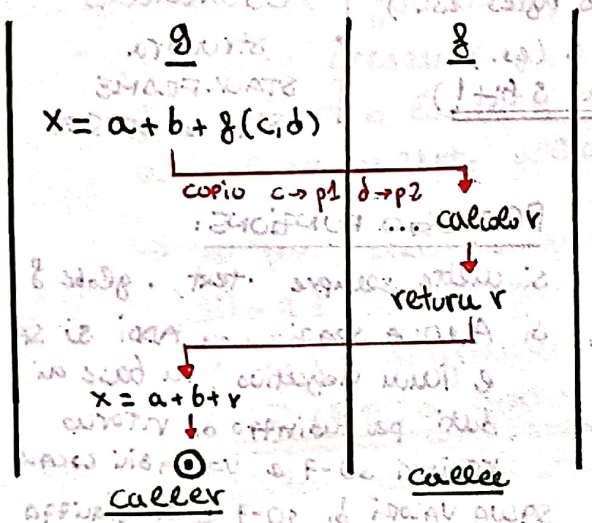
FUNZIONI in ASSEMBLY:

STACK: ZONA DI MEMORIA compresa fra l'INDIRIZZO PUNTATO da sp (STACK POINTER) e l'ULTIMA CELLA DI MEMORIA (INDIRIZZI ALTI)

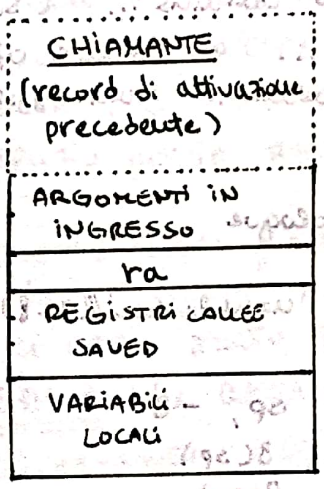
↳ manipolando registro sp possiamo allargare o restringere tale zona

↳ lo STACK PARTE VUOTO, ovvero sp punta alla parola dopo l'ultima cella di memoria, e cresce verso gli indirizzi più bassi, decrementando sp.

PASSAGGI LOGICI FUNZIONE:



STACK FRAME LAYOUT:



Se la funzione chiama se stessa

↳ INDIRIZZO di RITORNO

*NOTA: parametri complessi come le struct non sono inclusi nello stack frame ma sono sempre sotto il chiamante

STACK FRAME: STRUTTURA DATI MEMORIZZATA NELLO STACK CHE RENDE POSSIBILE ATTIVARE LA FUNZIONE

CONTIENE:

- VALORI DEI PARAMETRI FORMALI IN INGRESSO SE NON POTEVANO ESSERE ALLOCATI NEI REGISTRI
- VALORI ORIGINALI DEI REGISTRI CALLEE-SAVED (compreso ra)
- EVENTUALI VALORI RESTITUITI SE NON POSSONO ESSERE PASSATI TRAMITE REGISTRO
- VARIABILI LOCALI DICHIARATE NELLA FUNZIONE STESSA

CALLEE-SAVED REGISTERS: il CHIAMATO-CALLEE può usare tali registri ma deve prima salvarne il contenuto nello STACK

REGISTRI NON ASSICURATI DAL CALLEE: deve essere il chiamante a occuparsene se gli serve

INSERIMENTO DATO IN STACK FRAME - PUSH: si decrementa sp per allocare spazio e store

```
ADDI  SP, SP, -8
SD    S0, 0(SP)
```
PUSH

PRELEVAMENTO DATO DA STACK FRAME - POP: load e incremento sp per eliminare dato, riducendone le dimensioni:

```
LD    S0, 0(SP)
ADDI  SP, SP, 8
```
POP

ESECUZIONE CODICE:

*NOTA: per il PASSAGGIO DEI PARAMETRI le CONVENZIONI sono:

- caller: usa registri argomento a0 - a7 } 8 in INGRESSO, 2 in USCITA
- callee: usa a0 - a1

Se ce ne sono di più devono essere passati salvandoli sullo STACK FRAME.

PER OGNI FUNZIONE OCCORRE FARCIÒ:

```
1. # stack frame information for function "f":
2. # register a0 contains ...   (size: 8 bytes es.)
3. # parameter ...  at stack offset: 0 (es.)
4. # (per ogni parametro stack offset, ogni 8 bit!)
```

} COMMENTO struttura STACK FRAME SIZE e OFFSET

FUNZIONE - CALLEE

```
5. # function prologue
6.          .text
7.          .globe 'nome funzione' //(es. f)
8. f:
9.      ADDi    sp, sp, -8
10.     SD      ra, 8(sp)
11.     SD      s0, 0(sp)

12. # function Body
        ... corpo funzione

13: # function epilogue
14.     ADDi    sp, sp, 8
15.     RET
```

PROLOGO FUNZIONE:
si mette sempre .text, .globe f
si ALLOCA SPAZIO con ADDi su SP e il suo negativo, in base ai dati, per indirizzo di ritorno registri s0-7 e VARIABILI LOCALI
SALVA VALORI di s0-7 e INIZIALIZZA VARIABILI LOCALI su STACK

CORPO FUNZIONE:
esegue funzionalità specifica

EPILOGO:
ripristina valori registri s0-s7
DEALLOCA STACKFRAME (sp torna a valore antecedente all'invocazione)
ESEGUE RET

CALLER

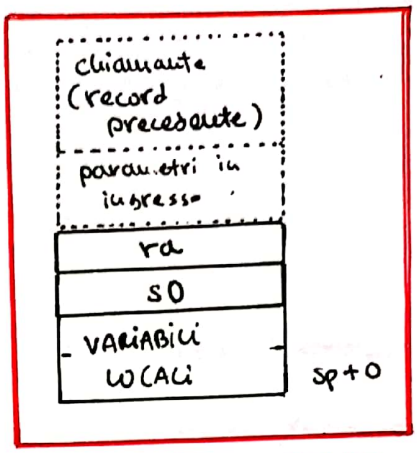
All'ATTO DELLA CHIAMATA, il CALLER deve:

- se intende usarli, salvare su stack t0-7, a0-7
- scrivere parametri attuali su a0 - a7
- invocare CALLEE con CALL f → i risultati saranno in a0,a1 vanno subito copiati!
- se necessario recuperare lo spazio dei parametri in ingresso incrementando sp opportunamente
- Ripristina a0-7, t0-7

SUI REGISTRI: a0 - a7 : INPUT | a0 - a1 : output | t0 - t7 REGISTRI TEMPORANEI A DISPOSIZIONE DELLA FUNZIONE |
| SO SI USA SE I REGISTRI a0 - a1 - ... - a7 sono richiesti nella FUNZIONE (es. per chiamare una altra funzione)

→ **PRINTF**:

```
.LC0:
    .string "stringa"
    ...
LUI  a2, %hi(.LC0)
ADDI a0, a2, %lo(.LC0)
CALL printf
```



STRUTTURA STACK FRAME

→ **STRUTTURA STACK FRAME**:

Osserviamo la funzione, e facciamo le seguenti considerazioni:

- se sono presenti all'interno di f altre funzioni, comprese la PRINTF o la FUNZIONE STESSA (RICORSIONE) allora bisognerà necessariamente allocare spazio per ie RETURN ADDRESS (ra)
  - ↳ se poi, nel chiamare queste altre funzioni, nell'usare a0 - a1 ... per passare i dati, siamo costretti a rimuovere ie dato originale che ci era stato passato in ingresso e QUESTO DOPO CI SERVE (!) allora occorre allocare spazio per REGISTRO/I CALLEE-SAVED (s0)
- se vengono dichiarate variabili locali all'interno della funzione bisogna allocare spazio anche per queste (ricordandosi delle dimensioni) (VARIABILI LOCALI)
- se in ingresso sono passati parametri complessi come un TYPEDEF STRUCT questo è al di sopra di tutto, ma non rientra nello STACK FRAME

→ **PROLOGO FUNZIONE**:

```
    .text
    .globe f
f:
ADDI sp, sp, -dim
SD   ra, OFF(sp)  // se presente
SD   s0, OFF(sp)  // se presente
```

→ **EPILOGO FUNZIONE**:

```
LD   ra, OFF(sp)  // se presente
LD   s0, OFF(sp)  // se presente
ADDI sp, sp, +dim
RET
```

PRINCIPI DI ARCHITETTURA DI UN PROCESSORE

IDENTIFICAZIONE DEI CONFLITTI:

esempio

```
SUB x2, x1, x3   IM - REG = AW - DM - REG
SUB x3, x1, x2   IM - REG = AW - DM - REG
                    conflitto!
```

I CONFLITTI possono essere:

- STRUTTURALI: tentativo di usare la stessa risorsa da parte di ISTRUZIONI DIVERSE
  (RIGUARDA solo il REGISTER FILE)
  ↳ FASI: D, W
- SU DATI: un'istruzione che dipende dal risultato di una istruzione precedente ancora in PIPELINE (*)
- SUL CONTROLLO: quando vi è un tentativo di prendere una decisione sulla prossima istruzione da eseguire prima che la condizione sia valutata (i SALTI)
  ↳ nelle istruzioni di salto come la BEQ, la decisione relativa al salto non viene presa fino allo stadio M (Data Memory)

```
BEQ x2, x1, #ff   IM + REG ≠ AW + DM + REG
ADD x12, x2, x5      IM + REG ≠ AW + DM + REG
                     conflitto!
```

esempio

SOLUZIONE AI CONFLITTI:

CONFLITTI SU DATI

STALLI HARDWARE: si "congela" la istruzione con conflitto dati, e si attende che la prima istruzione, di cui è richiesto il dato, sia completa

```
SUB x2, x1, x3
SUB x3, x1, x2
```

F	D	E	M	W			
	F	•	•	D	E	M	W

NOP SOFTWARE: il compilatore inserisce delle NOP tra quelle con dipendenza

```
SUB x2, x1, x3
NOP
NOP
SUB x3, x1, x2
```

F	D	E	M	W			
	F	D	E	M	W		
		F	D	E	M	W	
			F	D	E	M	W

FORWARDING DEI DATI: Si sfruttano risultati temporanei, senza attendere che questi siano stati scritti nei REGISTRI.
Si PRELEVANO i DATI IN INPUT DA UN QUALSIASI REGISTRO DI PIPELINE e lo si sfrutta guadagnando cicli (ANCHE DETTO BYPASSING)

*NOTA: LA FRECCIA DI FORWARDING NON PUÒ ANDARE INDIETRO!

- Forwarding verso la ALU dei dati da tutti gli stadi:

```
SUB x2, x1, x3      IM + REG ≠ ALU + DM + REG
SUB x3, x1, x2      IM + REG ≠ ALU + DM + REG
```

FRECCIA DI FORWARDING: è risultato di x2 è già disponibile dopo la ALU, anche se ancora stato scritto nei registri, però lo possiamo usare col FORWARDING!

F	D	E	M	W	
	F	D	E	M	W

*Il dato è già prodotto dopo la ALU (E) e può essere sfruttato!

- Forwarding verso lo STADIO DI MEMORIA:

```
LW  x2, lab(x8)     IM + REG ≠ ALU + DM + REG
SW  x2, lab(x10)    IM + REG ≠ ALU + DM + REG
```

F	D	E	M	W	
	F	D	E	M	W

*Il DATO è già prodotto dopo la Data Memory (M) e può essere sfruttato!

CONFLITTI SU CONTROLLO

Riguarda soprattutto le ISTRUZIONI DI SALTO CONDIZIONATO o INCONDIZIONATO

- Nessuna ottimizzazione dei branch:

SOFTWARE: Inserisco 3 nop dopo ogni istruzione di salto condizionato
HARDWARE: inserisco 3 stalli dopo aver riconosciuto una istruzione di salto condizionato

```
BEQ  x1, x2, lab
ADD  x3, x1, x2
```

F	D	E	M	W				
				F	D	E	M	W

- Con ottimizzazione dei branch:

```
BEQ  x1, x2, lab
ADD  x3, x1, x2
```

F	D	E	M	W		
		F	D	E	M	W

*in uscita dalla ALU (E) sappiamo già se l'istruzione verrà eseguita o meno (FORWARDING!)

*Con FORWARDING, il confronto per branch ottimizzato avviene nello stadio D

→ MISURA DELLE PRESTAZIONI :

(C_L: N° ISTRUZIONI SEQUENZA)

LATENZA λ_L: NUMERO DI CICLI CHE BISOGNA ATTENDERE PER IL COMPLETAMENTO DI UNA SEQUENZA DI ISTRUZIONI (DA F della PRIMA a W della ULTIMA).

$$\lambda_L = C_L + (4)$$ → bisogna attendere si svuoti!

CICLI PER ISTRUZIONE π_L:

$$\pi_L = \frac{\lambda_L}{\text{N° ISTRUZIONI SEQUENZA}}$$

(per loop, si ricavano λ_L e N° in funzione delle ITERAZIONI I e si fa $\lim_{|I| \to \infty} \frac{C_L + 4}{C_L}$)

THROUGHPUT: τ_L: NUMERO ISTRUZIONI ESEGUITE PER ISTANTE DI TEMPO

$$\tau_L = \text{§ch} / \pi_L$$

(per loop → §ch)

Diviso 10^6 ottengo in MIPS

Diviso 10^9 ottengo in Gips

SPEEDUP $\sigma_{2,1}$: DELLA CONFIG. 2 RISPETTO ALLA CONFIG 1.

$$\sigma_{2,1} = \frac{\lambda_{1,L} \cdot \text{§c2}}{\lambda_{2,L} \cdot \text{§c1}} - 1$$ CCLOCK DIVERSO

$$\sigma_{2,1} = \frac{\pi_{1,L}}{\pi_{2,L}} - 1$$ CCLOCK UGUALE

*NOTA: per il calcolo di τ_L si considerano:

- numero iterazioni I
- numero istruzioni per iterazione m
- numero stalli per iterazione

$$\lambda_L = m \cdot I + \text{STALLI} \cdot I + 4 \quad (\text{per } I \to \infty)$$

→ RICAPITOLANDO IL FORWARDING :

ADD/SUB (ARITMETICO LOGICHE): DATO PRONTO IN ALU (E)
RICEVE DATI IN ALU (E)

LD/SD (LOAD/STORE): DATO PRONTO IN DATA MEMORY (M)
RICEVE DATI IN DATA MEMORY (M)

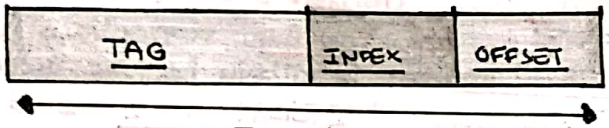
BEQ (SALTO CONDIZIONATO): IL CONFRONTO AVVIENE IN D, ANZICHÈ IN E e dunque LA DECISIONE SULLA ISTRUZIONE SUCCESSIVA è presa già in D, anziché essere presa in W

RICEVE in REGISTERS (D)

Il FORWARDING avviene sempre in "diagonale" (es. FDEMW / FDEMW NO in VERTICALE!)

→ **SIMULAZIONE CACHE**

INDIRIZZI



STRUTTURA

Dati forniti

DIMENSIONI MEMORIA DI LAVORO
DIMENSIONE CACHE
DIMENSIONI BLOCCO CACHE

Calcoliamo dunque la dimensione in BIT di ogni CAMPO: ↝ VADO A TENTATIVI!

1) **TOT BITS** (dimensione totale): DIMENSIONE MEMORIA DI LAVORO

(es. = 32 KiB ⇒ 2^{15} = 32768 ⇒ 15 BITS)

2) **INDEX**:

CACHE A INDIRIZZAMENTO DIRETTO:

$$N^\circ \text{ BLOCCHI} = \frac{\text{DIMENSIONE CACHE}}{\text{DIMENSIONI BLOCCO CACHE}}$$

↳ INDEX = n° BITS NECESSARI per poter numerare univocamente ogni blocco)

(es. 4 BLOCCHI → 2 BITS
5 BLOCCHI → 3 BITS
8 BLOCCHI → 3 BITS)

CACHE A INDIRIZZAMENTO A PIÙ VIE / COMPLETAMENTE ASSOCIATIVE:

$$\text{Calcolo } N^\circ \text{ BLOCCHI} = \frac{\text{DIMENSIONI CACHE}}{\text{DIMENSIONI BLOCCO CACHE}}$$

$$\text{Calcolo } N^\circ \text{ BLOCCHI EFFETTIVI} = \frac{N^\circ \text{ BLOCCHI}}{N^\circ \text{ VIE}}$$

INDEX = n° BITS necessari per poter numerare univocamente ogni blocco

(es. N° BLOCCHI EFFETTIVI: 1 → 0 BITS | 2 → 1 BIT | 3 → 2 BITS | 4 → 2 BITS | 5 → 3 BITS | ...)

3) **OFFSET**: DIMENSIONI BLOCCO CACHE (es 1 KiB ⇒ 2^{10} = 1024 ⇒ 10 BITS)

4) **TAG**: PER SOTTRAZIONE

TAG = TOT − INDEX − OFFSET

SIMULAZIONE CACHE

- A INDIRIZZAMENTO DIRETTO:

1) DIVIDO GLI ADDRES NEI CAMPI APPENA CALCOLATI. Siamo interessati a INDEX e TAG

2) Osservo INDEX → ci dice quale Blocco OSSERVARE

3) Se ie Blocco è:

   A) NON VALIDO (x): ci sarà sicuramente una Miss, sempre!

   B) VALIDO: se i TAG coincidono ci sarà HIT, altrimenti Miss

4) In caso di:

   I) Miss: Aggiorno ie Blocco con nuovo TAG e VALUE (M). Se ie Blocco NON ERA VALIDO ORA LO È.
   ACTION: Carica mem[value] in cache[tag] (in base 10)

   II) Hit: ie Blocco rimane invariato
   ACTION: legge mem[value] da cache[tag] (in base 10)

   VALUE = CODIFICA IN BASE 10 DI TAG - INDEX

5) Gli altri blocchi rimangono invariati. Procedo con l'indirizzo successivo considerando eventuali modifiche del blocco.

- TOTALMENTE ASSOCIATIVA:

In questo caso scompare INDEX, c'è solo TAG

1) Per ogni indirizzo confronto con tutti i blocchi ie TAG (validi!)

→ In caso di HIT, ie blocco viene letto, nessuno viene modificato.
ACTION: legge mem[TAG_{10}] da cache[0.a / b / c / d ...] (in base a dove ho la hit)

→ In caso di Miss:

- Se ci sono BLOCCHI NON VALIDI sfrutto uno di questi per caricare ie dato richiesto (modificando dunque lo stato del blocco)
- Se i BLOCCHI sono TUTTI VALIDI, allora procedo a RIMUOVERE/USARE QUELLO CHE NON VIENE USATO DA PIÙ TEMPO (UNO di quelli...)
  ACTION: carica mem[TAG_{10}] in cache[0.a / b / c / d ...] (in base al blocco scelto)

2) Procedo per gli indirizzi successivi considerando le modifiche ~~agli~~ ai blocchi.

- **INDIRIZZAMENTO A PIÙ VIE**

È un ibrido dei due indirizzamenti. si ricava logicamente il procedimento osservando i precedenti due.