Architettura degli Elaboratori

Informatica per la Comunicazione Digitale

Università degli Studi di Milano

Lezione 10:

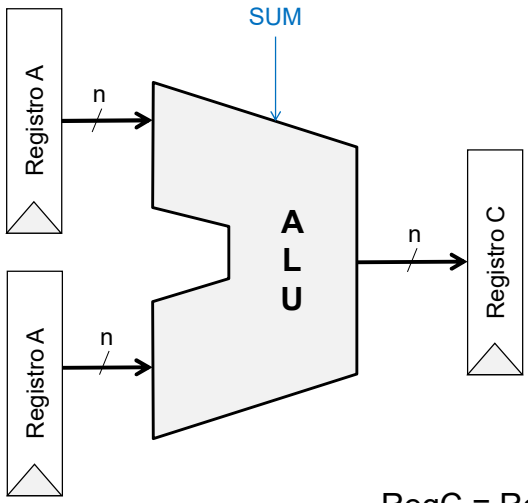
# Banco dei registri e datapath

(parte A)

Marco Tarini

1

## Idea: momorizzare il risultato di una computazione della ALU su dei registri




RegC = RegA + RegB



2

## Register File

- Il **register file** è un blocco funzionale essenziale dei processori (CPU)
- E' costituito da un piccolo insieme di registri (per es, 16, 32, o 64) che contengono i dati su cui il programma in esecuzione sta lavorando
  - sono gli operandi e i risultati (di tipo intero, o float, etc) delle istruzioni aritmetico-logiche eseguite del processore
  - cioè gli input e l'output dei conti eseguiti dalla ALU
  - la ALU è fisicamente collegata, in input e in output a questo Register File
- **Sintassi:** qui, ci riferiremo agli *m* registri, numerari da 0 a *m* – 1, come
  - reg[0], reg[1], reg[2] … reg[ m-1 ]
    oppure
  - R0, R1, R2 …
    oppure
  - $0, $1, $2 … (adottando una sintassi spesso usata per i registri)



3

## Banco di registri (Register File)

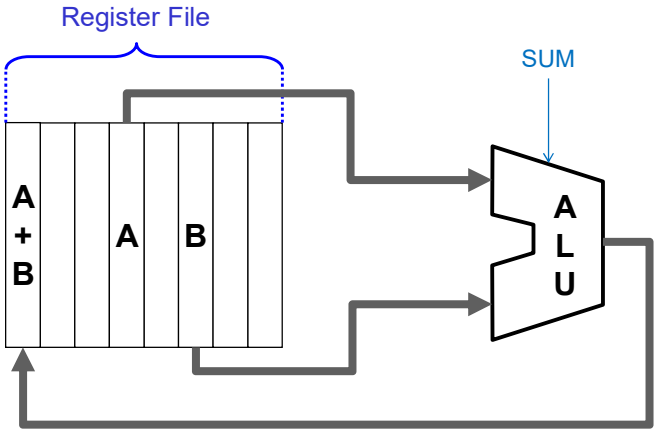
- In un elaboratore, è utile avere a disposizione un certo numero di registri paralleli, tutti aventi le stesse dimensioni e le stesse funzioni, da usare intercambiabilmente durante l'elaborazione.
- Questa idea è implementata dal **Banco dei registri**
  - Eng: **Register File**
  - (nota linguistica: «File» nel senso di «filiera», «fila»:
    I «file» di un File System non centra nulla)
- Un Banco di registri M×N è composto da
  - un insieme di M registi da N bit
  - i circuiti di controllo, per accedervi in lettura e scrittura

  Ed è provvisto di
  - 1 (o più) canali di ingresso
    per immettere i valori da memorizzare in uno dei registri
  - 1 (o più) canali di uscita
    che espongono i valori attualmente memorizzati in uno dei registri



4

## Esempio: i registri della CPU
## Schema concettuale

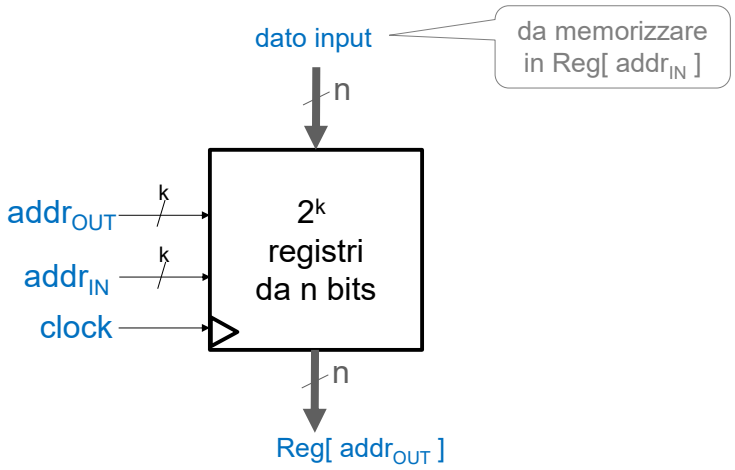



- Qui: un registro viene sovrascritto con la somma di altri due
  - (calcolata dalla ALU)



6

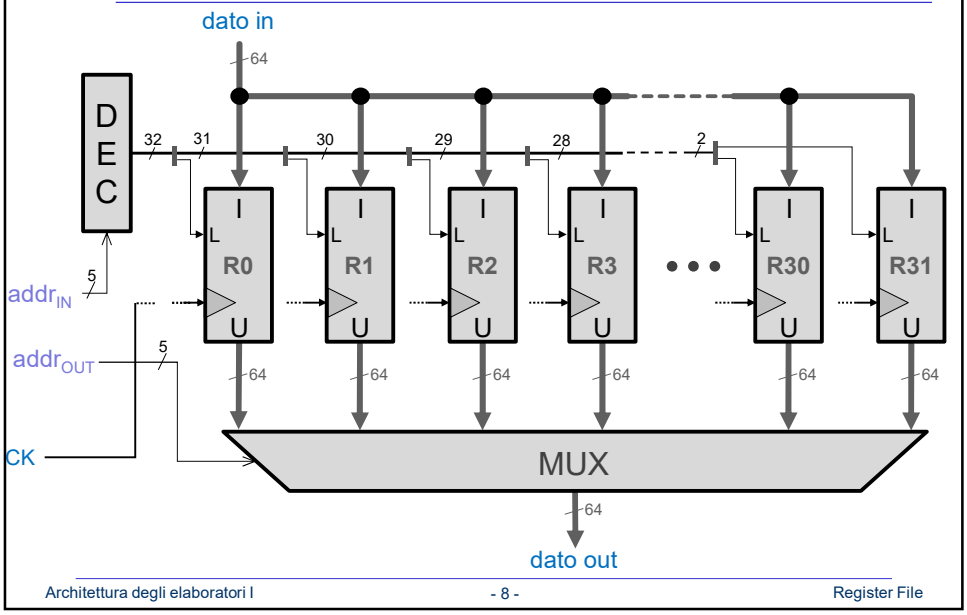
## Register file
## a 1 canale di uscita e 1 canale di entrata






7

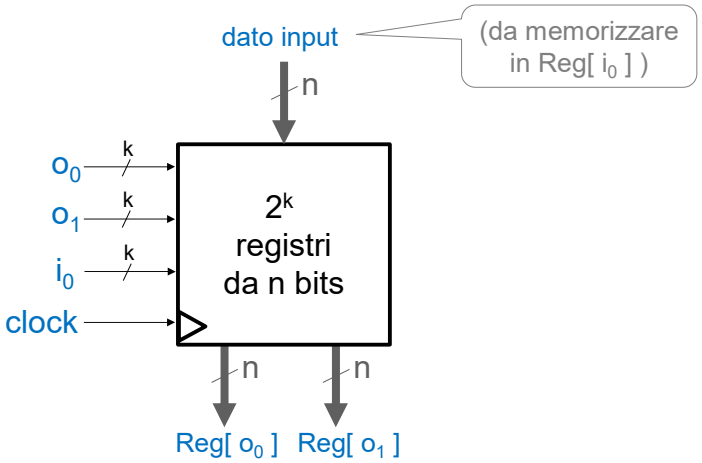
## Register file di 32x64
## Implementazione






8

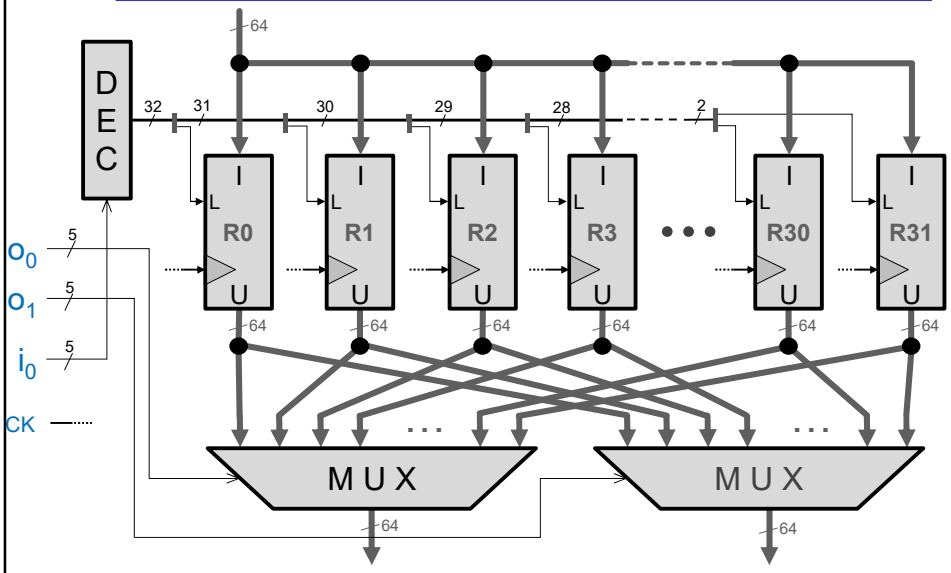
## Register File con 2 uscite
## Funzionamento






9

## Register File 32x64 (32 reg a 64 bit)
## 1 bus di input, 2 bus di output: implementaz.



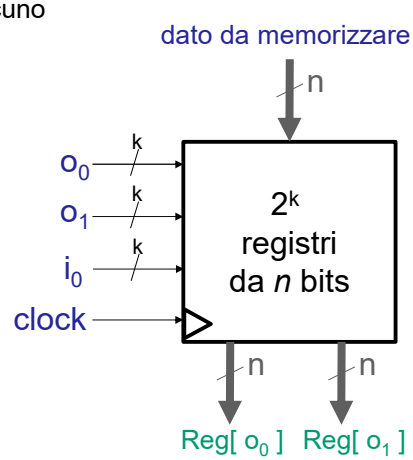



10

## Register file come blocco logico: funzionamento

Un **Register File** a 2 canali di uscita e 1 di entrata, per m = $2^k$ a n bit

- contiene: m = $2^k$ registri da n bit ciascuno
  numerati da 0 a $2^k - 1$
  (per es, se k=5, da 0 a 31)
- in output, restituisce
  il valore attuale
  di due dei registri:
  specificati dagli indici $o_0$ e $o_1$
  (presi in ingresso)
- in input, prende anche
  la parola (di n bit)
  che verrà memorizzata
  nel registro di indice $i_0$
  - appena il clock passa da 1 a 0
- e il clock, per la sincronizzazione
  (come al solito)






11

## Register File: note sulle dimensioni

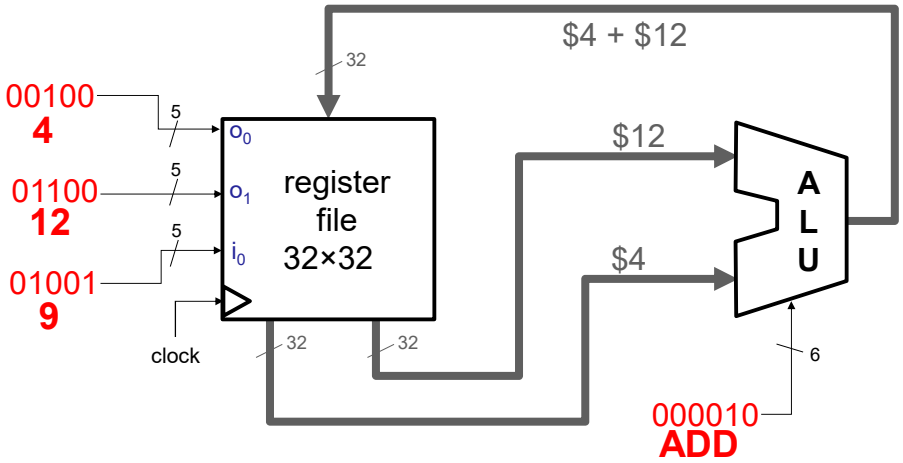
- Nella nostra implementazione, per **m** registri a **n** bit, ci vogliono:
  - **k** = $\lceil \log_2 \mathbf{m} \rceil$ bit di input per ciascun indice
  - due MUX a **m** ingressi
  - **m** branching del dato in input (e un DEC da **k** a **m** bit)
  - **m** registri paralleli, con comando di caricamento (L)
  - (quindi ben **m** x **n** Flip-Flop ☹ )
  - 3**k** bit nelle **istruzioni in linguaggio macchina binario** necessari solo per specificare i tre indici dei registri

  } un prezzo molto caro, soprattutto al crescere di **m**

- Dimensioni tipiche:
  - **n** è la dimensione della «parola» (un *word*) dell'architettura.
    Macchine «a 32 bit»: **n** = 32
    Macchine «a 64 bit»: **n** = 64
  - **m** in quasi tutte le architetture reali è piccolo (es da 8 a 64)
  - valore molto popolare: **m** = 32 (**k** = 5)
  - **m** piccolo = Istruzioni Corte ☺ + Register File piccolo ☺

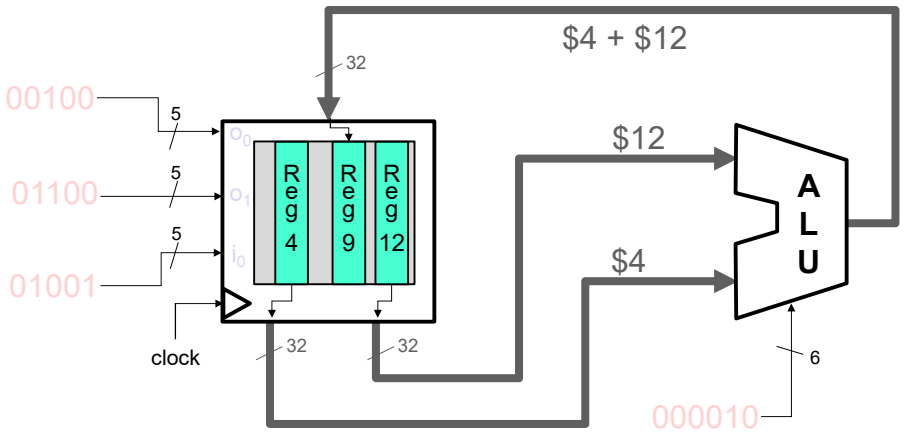


## Uso di un Register File esempio




Reg[9] ← Reg[4] + Reg[12]

## Uso di un Register File esempio




000010 01001 00100 01100
**ADD 9 4 12**



14

## Torniamo un secondo ad una visione complessiva dei livelli di astrazione

Istruzione in Linguaggio Macchina :

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Istruzione in Linguaggio Assembly :

```
ADD $9, $4, $12
```



15

## Note sulla sintassi del linguaggio assembly

- L'istruzione in linguaggio macchina segue una sintassi di un linguaggio macchina inventato per la mini architettura vista, proposta come esercizio
- L'istruzione assembly vista (e le successive) seguono la sintassi di un reale linguaggio assembly: il **MIPS-32**
- Note su questa sintassi:
  - Le virgole (che separano gli operandi) sono opzionali
  - I registri sono identificati da $ seguito da un numero da 0 a 31
  - Il  registro è quello dove viene memorizzato il risultato
  - I due registri seguenti sono gli operatori dell'operazione
  - L'operazione aritmetica o logica è identificato da un codice mnenomico di qualche lettera (come ADD, SUB, MUL, DIV, AND, OR, …)

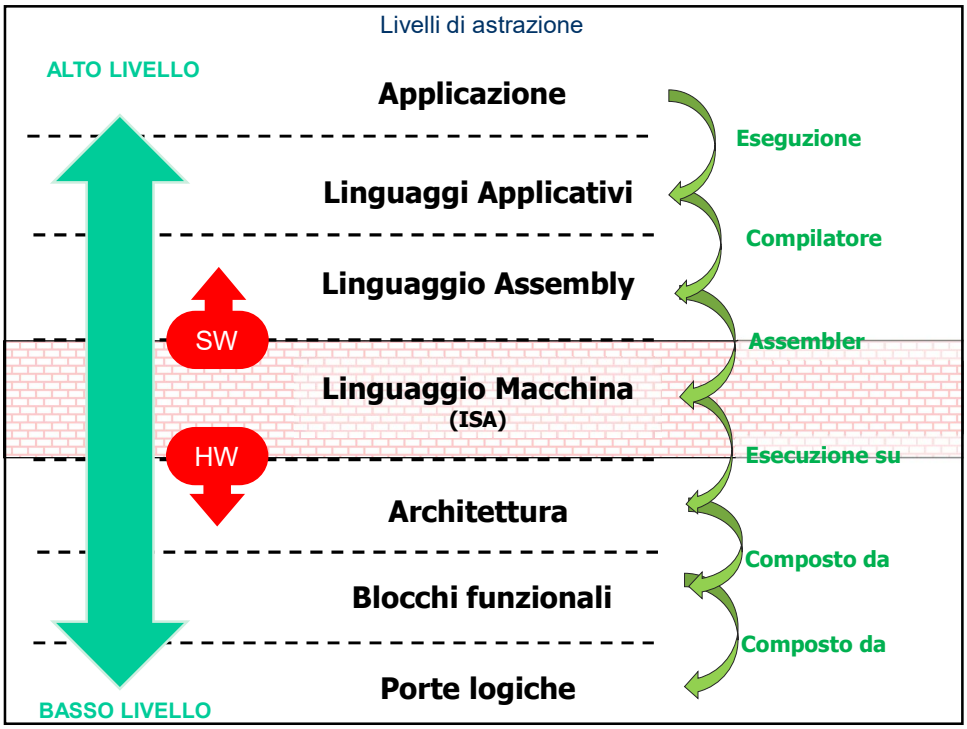
Cioè "facile da ricordare"… be', rispetto a «000010»

Bit-a-bit (in inglese: bit-wise)



16

## Livelli di astrazione




17

## Programmabilità

- Il circuito di questo esempio sta eseguendo istruzioni del tipo: «somma i registri R4 e R12 e metti il risultato in R9»
- cioè « R9 = R4 + R12 »
- in un *Linguaggio Macchina*, questo comando può apparire come:

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| codice per l'operazione «ADD» (da mandare alla ALU come OP-code) | «9», indice del registro di destinazione (da mandare al Register File come $i_0$) | «4», indice del primo operando (da mandare al Register File come $o_0$) | «12», indice del secondo operando (da mandare al Register File come $o_1$) |
|---|---|---|---|

- in un *Linguaggio Assembly*, questo comando può apparire come

```
ADD $9, $4, $12
```



18

## Collegamento con linguaggi a più alto livello di astrazione

- In un linguaggio ad alto livello (es. C o Java) scriveremo:

```
int tot_animali, n_mucche, n_pecore;
… /* altro codice */
tot_animali = n_mucche + n_pecore;
```

- Il compilatore potrebbe (per es) decidere che le variabili con gli identificatori «tot_animali», «n_mucche», «n_pecore» vadano memorizzati nei registri n. 10, 11, e 12 rispettivamente, e tradurre il comando di assegnamento come:

```
ADD $10, $11, $12
```

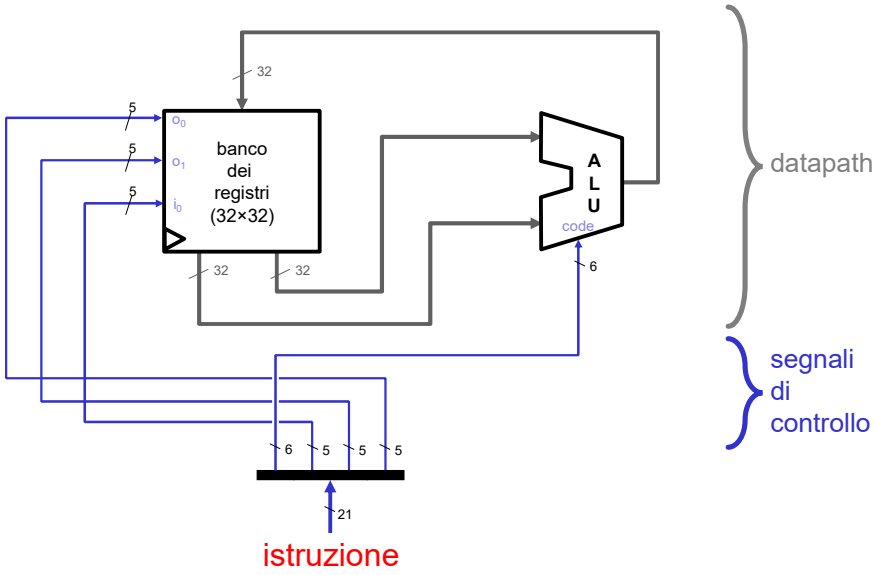
che in un linguaggio macchina binario diventa (per es):

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|



19

## Un primo elaboratore (esecutore di istruzioni)




20

## Istruzioni e programmi

- Mettiamo di voler assegnare al registro 2 ($2) il valore dell'espressione

$$\$2 = ( \$3 - \$4 ) \times ( \$1 + \$3 + \$4)$$

- Ecco una possibile sequenza di istruzioni "assembly" che, eseguite in successione, ottiengono il risultato voluto

```
SUB $10 $3 $4
ADD $11 $1 $3
ADD $11 $11 $4
MUL $2 $10 $11
```

Architettura degli elaboratori I - 21 - Register File

21

## Istruzioni e programmi: note

- Abbiamo avuto bisogno di registri addizionali per calcolare le *espressioni intermedie* come ( $3 - $4) e ($1 + $3 + $4 )
- L'espressione ($1 + $3 + $4) a sua volta va calcolata in due passi: sommando $1 a $3, e poi il risultato a $4 (questo perché la nostra ALU è solo capace di eseguire operazioni fra DUE operandi)
- Scelta arbitraria: nella nostra soluzione abbiamo usato i registri $10 e $11 per memorizzare le espressioni intermedie

- Quello che abbiamo ottenuto è un primo "**programma**" in **assembly**, cioè una **sequenza** di istruzioni assembly (da eseguire in successione)
  - di appena 4 istruzioni
- Una volta convertite in **linguaggio macchina** (task dell' **assembler**, che è a sua volta un software) queste istruzioni sono, singolarmente, eseguibili direttamente dal circuito che abbiamo visto
  - Linguaggio macchina = linguaggio binario («fatto di 0 e 1») di comandi eseguibili su una specifica macchina (come il circuito visto)



22

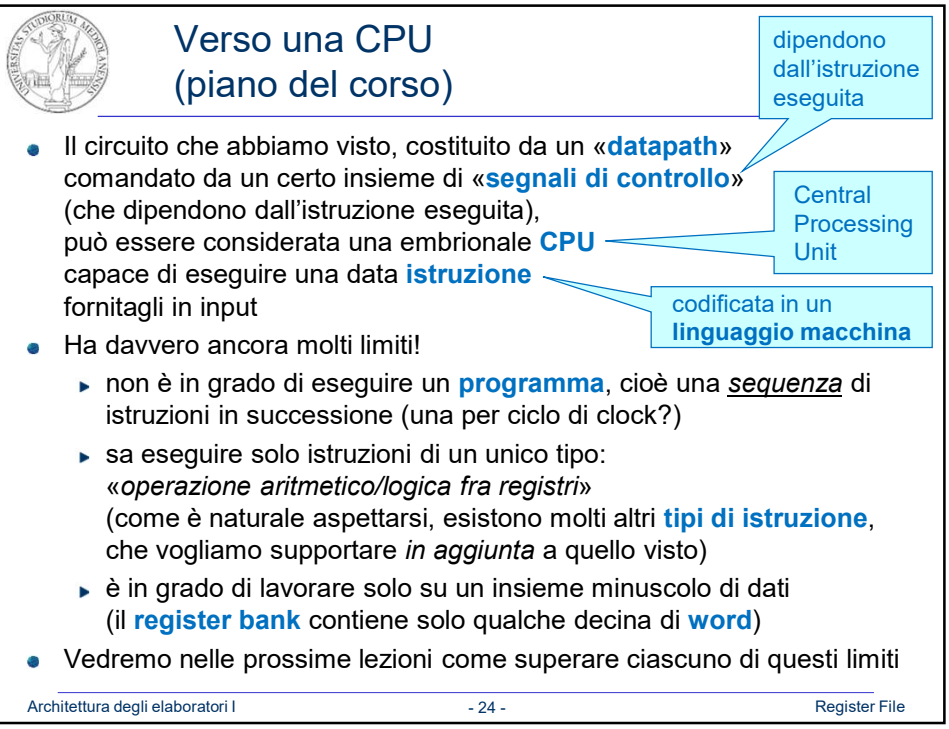
## Il datapath: note sul funzionamento e sulla durata del ciclo di clock

- Fino a che il ciclo di clock non finisce (con un fronte in discesa):
  - il Register File manda gli operandi in uscita;
  - la ALU trova questi operandi nei propri ingresso;
  - la ALU manda il risultato computato nella propria uscita;
  - il Register File trova il risultato nel proprio ingresso.
- Quando avviene il fronte in discesa, il Register File memorizza il risultato nel registro indicato
  - (nota: agli altri mandiamo L=0 e non si modificano)
- Alcune conseguenze importanti:
  - è possibile eseguire comandi del tipo R5 ← R5 + R2 infatti R5 viene sovrascritto solo alla fine del ciclo di clock. Fino ad allora la ALU vede il vecchio R5 immutato come operando.
  - Il ciclo di clock deve essere sufficientemente lungo da permettere al segnale di propagarsi dall'output del Register File al suo input, (compreso il computo della ALU) Questo mette un limite alla frequenza di clock!

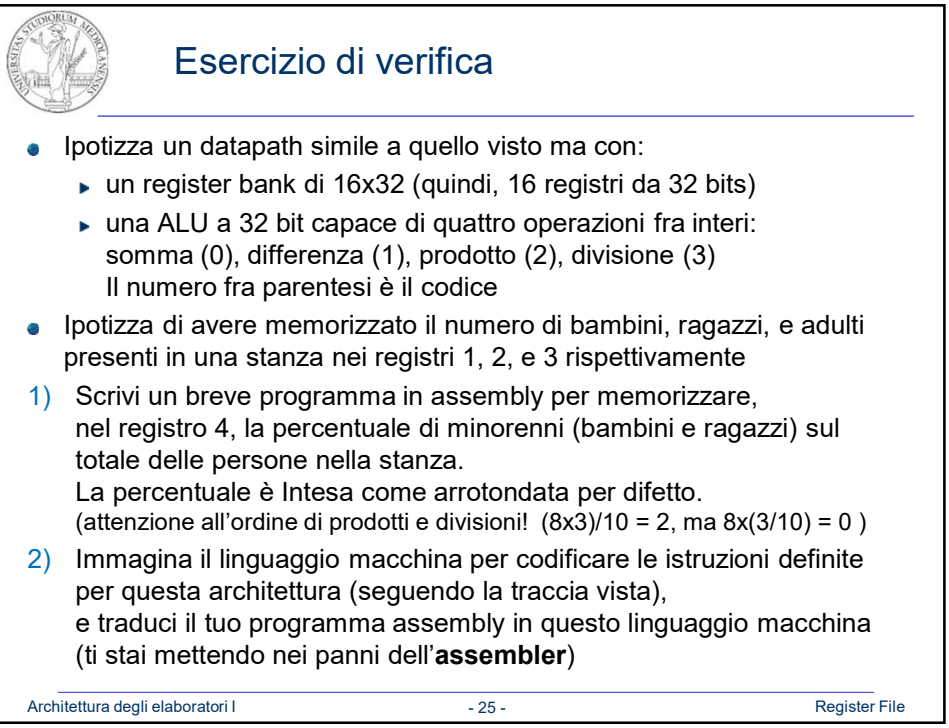
Stesso registro usato sia come operando che come risultato (viene letto e scritto nello stesso ciclo di clock!)



23


Verso una CPU
(piano del corso)
dipendono dall'istruzione eseguita
Central Processing Unit
codificata in un linguaggio macchina
Il circuito che abbiamo visto, costituito da un «datapath» comandato da un certo insieme di «segnali di controllo» (che dipendono dall'istruzione eseguita), può essere considerata una embrionale CPU capace di eseguire una data istruzione fornitagli in input
Ha davvero ancora molti limiti!
non è in grado di eseguire un programma, cioè una sequenza di istruzioni in successione (una per ciclo di clock?)
sa eseguire solo istruzioni di un unico tipo: «operazione aritmetico/logica fra registri» (come è naturale aspettarsi, esistono molti altri tipi di istruzione, che vogliamo supportare in aggiunta a quello visto)
è in grado di lavorare solo su un insieme minuscolo di dati (il register bank contiene solo qualche decina di word)
Vedremo nelle prossime lezioni come superare ciascuno di questi limiti
Architettura degli elaboratori I
- 24 -
Register File

24


Esercizio di verifica
Ipotizza un datapath simile a quello visto ma con:
un register bank di 16x32 (quindi, 16 registri da 32 bits)
una ALU a 32 bit capace di quattro operazioni fra interi: somma (0), differenza (1), prodotto (2), divisione (3) Il numero fra parentesi è il codice
Ipotizza di avere memorizzato il numero di bambini, ragazzi, e adulti presenti in una stanza nei registri 1, 2, e 3 rispettivamente
1) Scrivi un breve programma in assembly per memorizzare, nel registro 4, la percentuale di minorenni (bambini e ragazzi) sul totale delle persone nella stanza. La percentuale è Intesa come arrotondata per difetto. (attenzione all'ordine di prodotti e divisioni! (8x3)/10 = 2, ma 8x(3/10) = 0 )
2) Immagina il linguaggio macchina per codificare le istruzioni definite per questa architettura (seguendo la traccia vista), e traduci il tuo programma assembly in questo linguaggio macchina (ti stai mettendo nei panni dell'assembler)
Architettura degli elaboratori I
- 25 -
Register File

25