ARCHITETTURA MIPS

CARATTERISTICHE → **architettura RISC**, riduce le istruzioni eseguibili alle più semplici e ha solo 3 formati di istr diversi (J, R, I).

→ **architettura LO/STORE**, gli operandi dell'ALU possono provenire solo da registri e non da mem, quindi sono nec. 2 azioni: → LOAD;
→ STORE;

→ **architettura pipeline**, migliora le prestazioni sovrapponendo dove possibile l'esecuzione di istr diverse appart a flusso seq.

→ **DIFFERENZE RISC vs CISC**

	RISC	CISC
istruzioni	poche/sempl.	tante/complex
archi	LO/STR	"in memoria"
indirizz.	poche/semplici 'originali'	tante/complex
registri	tanti/omog.	pochi/eterog.
HW	poco cost e basso cons.	costosa e complex

- **INSIEME RIDOTTO ISTR** → fanno parte del reduce istr-set, suddivise in: → istr aritm-log
  → istr lo/store
  → istr salto
  (coprono i 3 formati di esec.)

1 → **ar-log**: `add $s1, $s2, $s3` oppure `slt $s1, $s2, $s3`

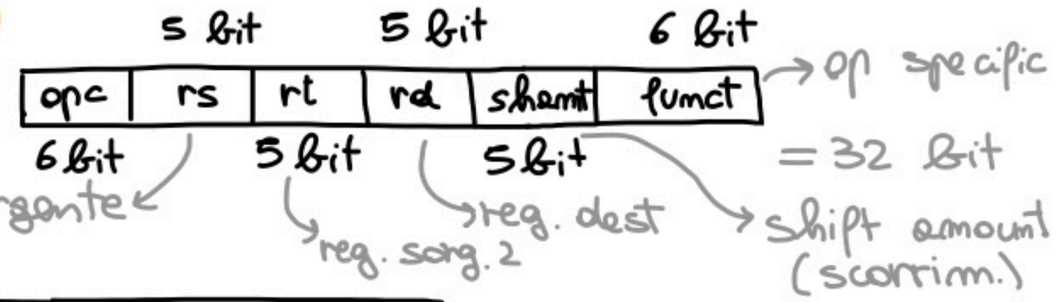
2 → **trasf lo/store**: `lw $s1, offset($s2)`

3 → **salto cond**: `beq $s1, $s2, L1` → go to L1 if ($s1==$s2)

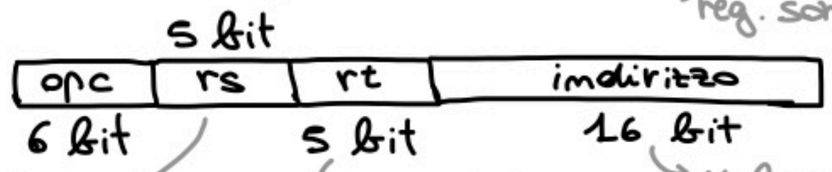
incond: `j L1` → jump to L1

→ i diversi formati sono distinti da **opcode** (primi 6 bit)

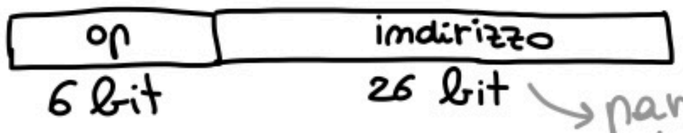
- **ISTR R** → arit-log a 3 reg. (rs, rt, rd)



- **ISTR I** → arit-log immed
  o lo/store
  o salto cond



- **ISTR J** → salto incondiz.



- **ESECUZIONE DI ISTR** → 1. **aritm. logiche** (tipo R), `add $x, $y, $z`
  - prelevo istr da mem e increm. PC;
  - lettura 2 reg sorgente dal banco dei regi
  - operaz. dell'ALU sui dati letti dal banco dei reg., utilizzando campo functions;
  - scrittura del risultato dell'ALU nel banco dei reg su reg dest;

→ simile a store

1/7

↳ simile a store

banco dei reg su reg dest;

2. **istr. load**, tipo I
   - prelevo istr da mem e increm PC;
   - lettura reg base(rs);

`lw $x, offset($y)`
   - operaz. dell'ALU per calcolare somma valore letto da rs dai 16 bit meno significativi;
   - prelevo dato da mem utilizzando come indr di lettura il ris dell'ALU;
   - scrittura dato prov. dalla mem nel reg. di dest;

3. **istr addi**: tipo I
   - prelevo istr. dalla mem e increm PC;
   - lettura reg. sorg;

`addi $x, $y, 22`
   - operaz. dell'ALU per calcolare somma valore letto dal reg. sorg e da immed;
   - scrittura risultato in ALU nel reg. dest;

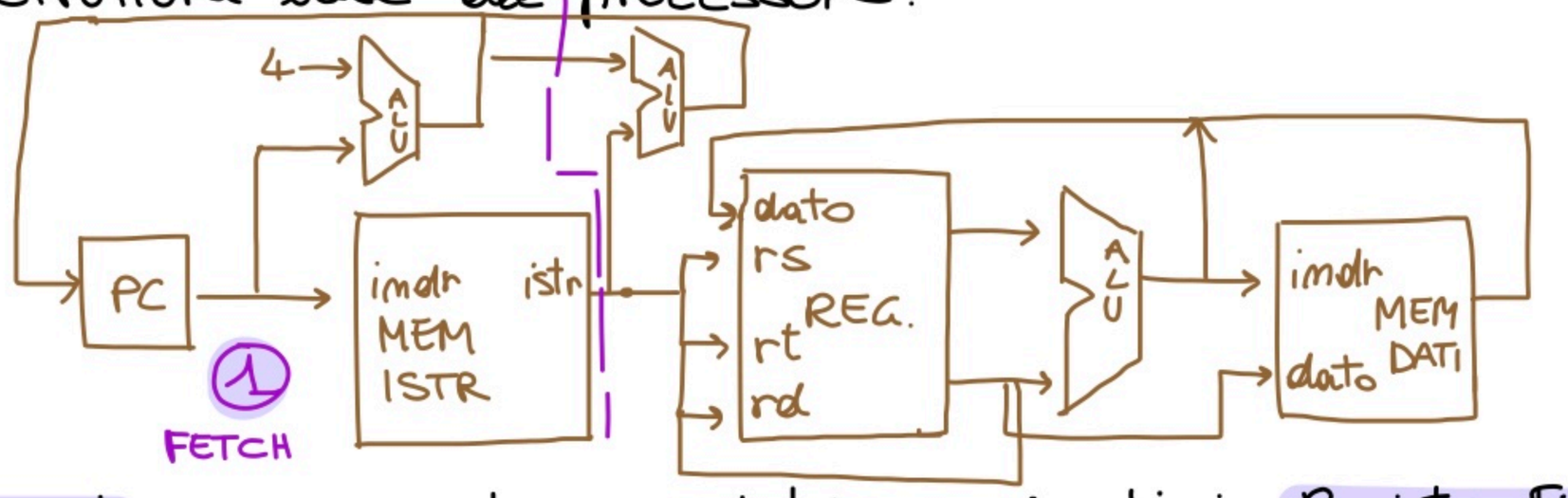
4. **istr BEQ**: `beq $x, $y, offs` tipo I
   - prelievo istr da mem e increm di PC;
   - lettura 2 reg sorg;
   - operaz. dell'ALU per effettuare la sottraz. tra i valori letti dal banco. Se val (PC+4) sottratto da offset;
   - Uscita Zero dell'ALU viene utilizz. per decidere quale val va memorizzato in PC; (PC+4 o PC+4+offset)

↳ riassumo dicendo: – per ogni ISTR inizio inviando contenuto di PC a Mem Istr;
– leggere 1 o 2 reg. utilizzando campi istr;

→ dopodiché dipende da tipo ISTR:
- **arit-log**, usa ALU per esec operaz;
- **LO/STR**, usano ALU per calcolo indr effettivo;
- **salti cond**, usano ALU per esito cond;

→ dopo l'utilizzo della ALU:
- **arit-log**, scrivono nel reg di dest il risultato;
- **LO/STR**, accesso in lett/scrittura a Mem Dati e carica dato in reg. dest o eseguono mem del dato sorg;
- **salti cond**, cambiano indr istruzione succ. a seconda del confronto;

→ **ARCHITETTURA**: struttura base del processore:



→ **Mem Istr** e **Mem Dati** sono separate e registri organizzati in **Register File** con 2 porte di lettura e 1 di scrittura (rs, rt, rd)

→ utilizzo delle ALU specificato sopra in descr. istr.

- **ESEC. CARICAMENTO (FETCH)** → avviene nella prima parte comprendente MEM ISTR, PC e ALU per indr succ.

- **ESEC. ISTR AR-LOG** → avviene nella parte con Register File e ALU (da 32 bit); Utilizza i 3 reg: per accedere ai 2 in lettura sono nec. 2 ing e 2 uscite a banco registri.
  - 5 bit per 1 ← 32 bit per 1
  - ↳ RegWrite: segnale controllo per scrittura
  - ↳ OP_ALU: segnale che specifica operazione

- **ESEC. ISTR LOAD** → avviene tra banco registri e mem. dati con estensione aggiun per estendere il valore dello spiazzamento da 16 bit a 32 bit con segno.
  - Simile per store
  - ↳ offset

→ lettura RF → calcolo ALU di indr. lett/scritt per accedere a Mem Dati → lett/scrit Mem Dati
- ↳ nel caso load: scrittura RF (in reg. dest)
- ↳ load scrive da Mem Dati in reg. dest
- ↳ store legge da reg sorg e scrive in Mem Dati

- **ESEC. SALTO COND** → avviene tra ALU, sommatore, Unità est. segno (16→32), logica rispetto a uscita ALU (zero)

→ calcolo indr dest del salto sommando (PC+4) e valore istr estesa (dopo scorr a sx di 2 bit) → confronto ALU dei reg. operand. letti da RF
- → uscita zero asserita → cond verif. indr dest=PC
- → Condi. non verif → PC increm sost attuale

→ Mem istr e Mem (cache) dati distinte per evitare che si riutilizzi una risorsa durante una esec. di istr.
↳ oppure si usa **multiplexer** per condividere risorse;
↳ prende due ing e in base a linee di controllo consid. uno o l'altro.
es. util in reg scrittura se ho lo/str o istr aritm/log

- **SEGNALI DI CONTROLLO** →
  - **RegDst**, se il segnale dest è RT o RD;
  - **RegWrite**, dato scritto in RF nel reg. individuato;
  - **ALUSrc**, se sto prendendo da reg o da imm 2° operando;
  - **MemtoReg**, dato per scrittura viene da MemDati o da ALU;
  - **MemRead/Write**, dato in Mem dati o é da leggere o da scrivi;
  - **PCSrc**, se salto é eseguito o meno;

↳ il risultato zero per i salti mi dice se i reg confrontati sono identici o meno;
↳ se utilizzo CPU a ciclo unico (ogni ISTR eseguita in 1 clock) può essere che il tempo di clock diventi eccessivamente lungo.