Architettura degli Elaboratori

Esercizi sull'architettura CPU: istruzioni e malfunzionamenti

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Checco

checco@di.uniroma1.it

Special thanks and credits:

Andrea Sterbini, Iacopo Masi,
Claudio di Ciccio

[S&PdC]
4.1-4.4

Riepilogo

SAPIENZA
UNIVERSITÀ DI ROMA

Istruzione jr (Jump to Register)

L'istruzione **jr rs** è di tipo **R**

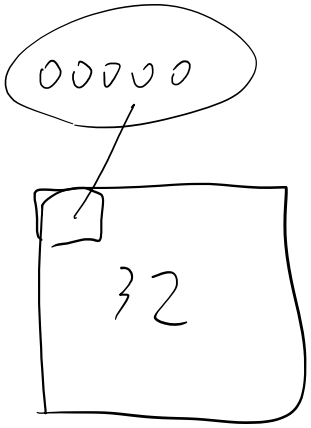
Cosa fa: trasferisce nel **PC** il contenuto del registro **rs**

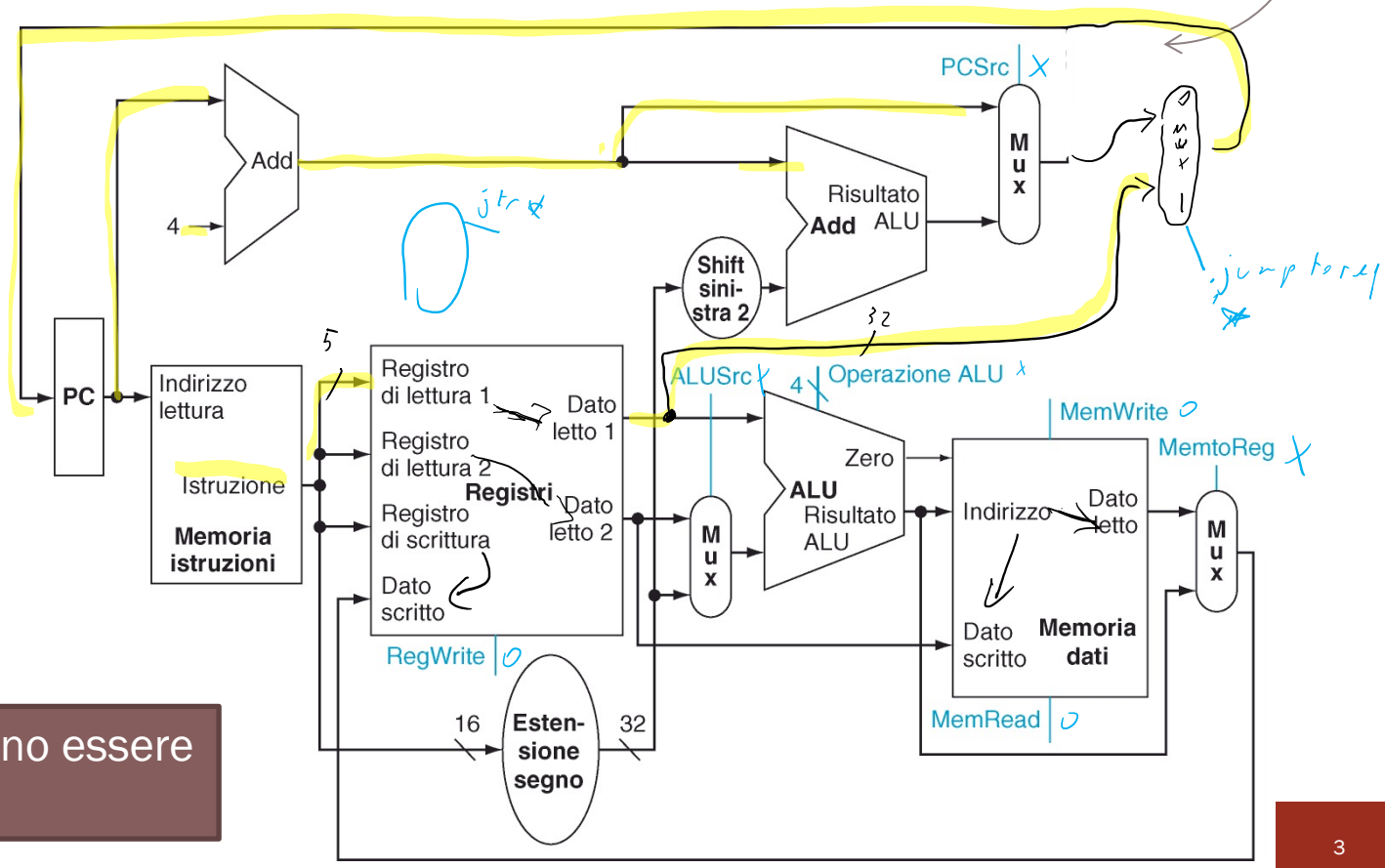
Unità funzionali: **MUX** per selezionare il PC dall'uscita del blocco registri

Flusso dei dati: **Registri[rs] → PC**

Segnali di controllo: **JumpToReg** che abilita il MUX per inserire in PC il valore del registro

Qui: scarabocchi





Tempo necessario: **Fetch+Reg**

Alcuni controlli possono essere Don't care, quali?

Homework

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio per casa: jrr

Aggiungere alla CPU l'istruzione **jrr rs** (Jump Relative to Register)

di tipo R, che salta all'indirizzo (relativo al PC) contenuto nel registro **rs**

Ovvero che esegue come prossima istruzione quella che si trova all'indirizzo

PC+4+Registri[rs]

a) Modificare lo schema per realizzare l'istruzione

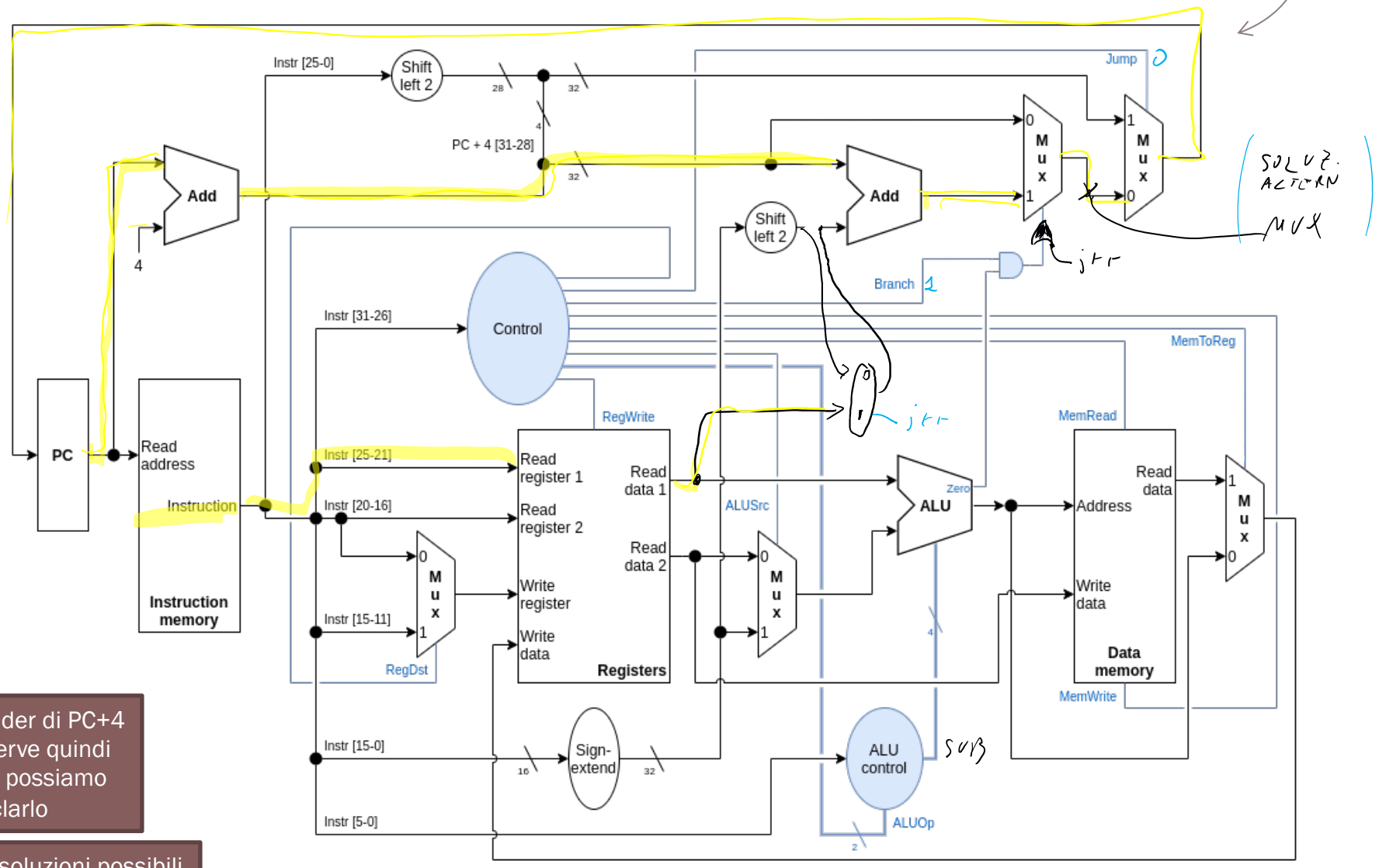
b) Indicare tutti i segnali di controllo che la CU deve generare

c) Calcolare il tempo di esecuzione della istruzione assumendo che:

Accesso a memorie = 66ns, accesso ai registri = 33ns, ALU e sommatori = 100ns

Esercizio per casa: jrr (soluzione /a) PC+4 + Reg[rs]

Qui: scarabocchi



L'adder di PC+4 ci serve quindi non possiamo riciclarlo

Più soluzioni possibili

Esercizio per casa: jrr (soluzione /b, c)

(b) Segnali dalla Control Unit										
JRR	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
1	X	X	X	0	0 (X)	0	X	0	X	X

Accesso a memorie = 66ns, accesso ai registri = 33ns, ALU e sommatori = 100ns

(c) Tempo di esecuzione				
Fetch 66ns	Reg[rs] 33ns			
PC+4 100ns		salto: PC+4 + Reg[rs] 100ns	PC ← salto 0ns	
Totale: 200ns				

Non è necessario incrementare il periodo di clock

Istruzione	Instr. Fetch	Instr. Decode	Execution	MEM	Wr. Back	Totale
add	66	33	100		33	232
lw	66	33	100	66	33	298

Aggiungere l'istruzione jral

Vogliamo aggiungere l'istruzione di tipo R

`jral $rs, $rt` (Jump to **R**egister **a**nd **L**ink to rt)

che salta incondizionatamente all'indirizzo contenuto nel registro **rs** e memorizza nel registro **rt** l'indirizzo della istruzione successiva.

a) Modificare lo schema per realizzare l'istruzione

b) Indicare tutti i segnali di controllo che la CU deve generare

c) Calcolare il tempo di esecuzione della istruzione assumendo che:
Accesso a memorie = 100ns, accesso ai registri = 50ns, ALU e sommatori = 150ns

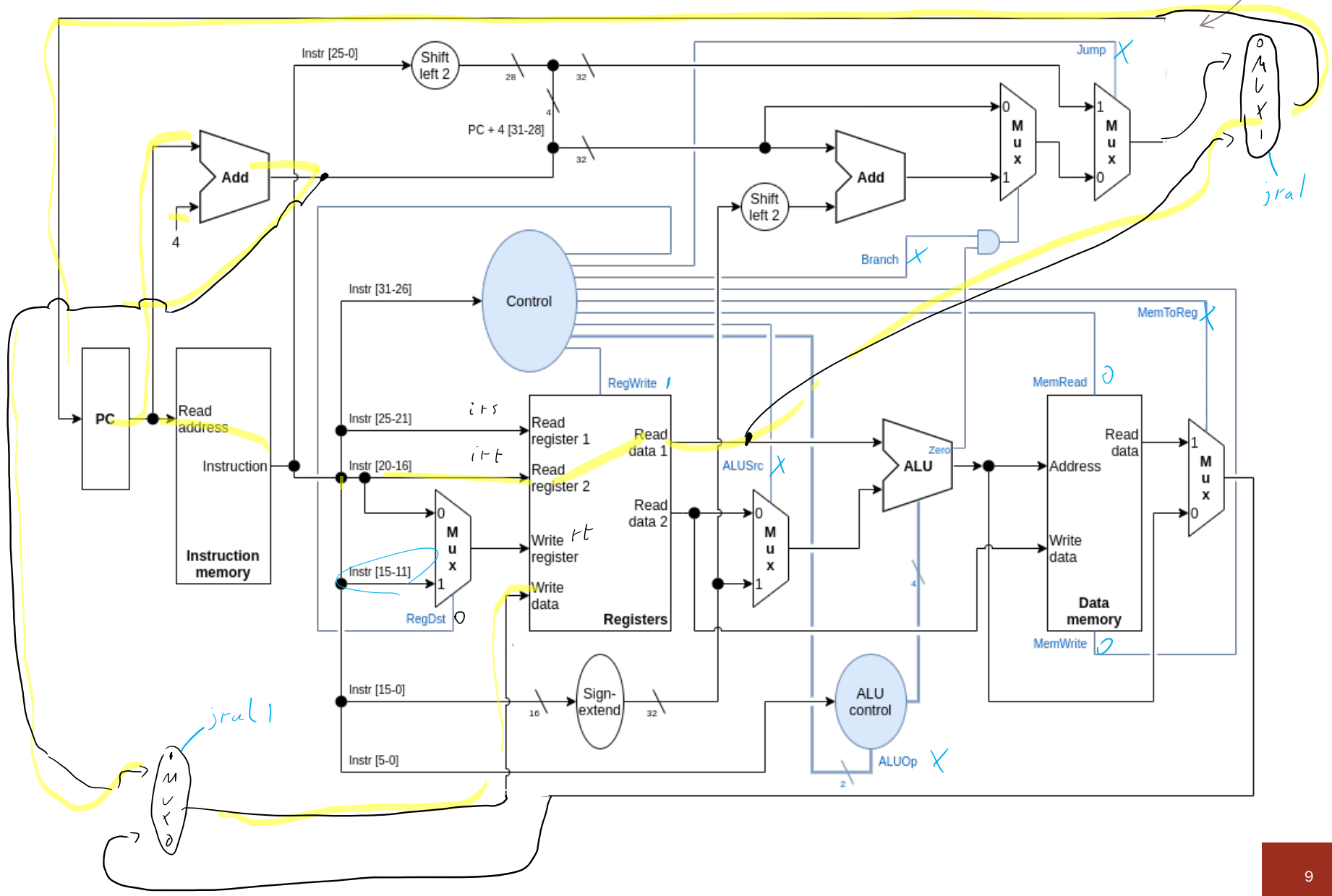
a) Le operazioni da compiere sono:

PC	← Registri[rs]	(salto incondizionato)
Registri[rt]	← PC + 4	(link)

Non c'è bisogno di unità funzionali nuove (a parte i necessari MUX).
Modifiche al datapath:
- Inviare il PC+4 all'ingresso di scrittura del blocco registri (con un MUX)
- Inviare il campo rt dell'istruzione come registro destinazione (come in una lw)
- Inviare l'uscita del primo argomento del blocco registri a PC (con un MUX)

Modifiche all'architettura per jral (rs->PC, PC+4->rt)

Qui: scarabocchi



Segnali e tempo di computazione per jral

(b) Segnali dalla Control Unit										
JRAL	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
1	0	X	X	1	X	0	X	X	X	X

Accesso a memorie = 100ns, accesso ai registri = 50ns, ALU e sommatori = 150ns

(c) Tempo di esecuzione		
Fetch 100ns	Reg[rs] 50ns	PC ← Reg[rs] 0ns
PC+4 150ns		Reg[rt] ← PC+4 50ns
Totale: 200ns		

Non è necessario incrementare il periodo di clock

Istruzione	Instr. Fetch	Instr. Decode	Execution	MEM	Wr. Back	Totale
add	100	50	150		50	350
lw	100	50	150	100	50	450

Esercizio per casa (nuovo): vj

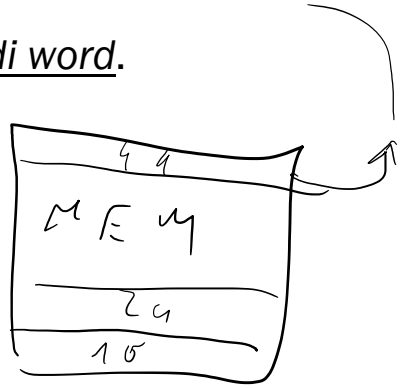
Si vuole aggiungere alla CPU l'istruzione vectorised jump (**vj**), di tipo **I** e sintassi assembly

```
vj $indice, vettore
```

che salta all'indirizzo contenuto nell'elemento $indice-esimo del *vettore* *di word*.

Esempio: se in memoria si è definito staticamente il

```
vettore: .word 16, 24, 312, 44
```



e al registro **$t0** è assegnato il valore **3** allora

```
vj $t0, vettore
```

salterà all'indirizzo **44** (che è l'elemento con indice 3 del vettore)

a) si disegnino le modifiche necessarie a realizzare la funzione, aggiungendo tutti gli eventuali MUX, segnali di controllo, bus, ALU e sommatori (ecc) che ritenete necessari.

b) indicate i valori di tutti i segnali di controllo, in modo da eseguire l'istruzione **vj.**

c) tenendo conto che il tempo di accesso ai registri (sia in lettura che scrittura) è di **25ns**, l'accesso alla memoria impiega **75ns**, la ALU e i sommatori impiegano **100ns** e ignorando gli altri ritardi, calcolate il tempo di esecuzione minimo della istruzione **vj** e indicate se è necessario aumentare il periodo di clock della CPU per poter svolgere questa nuova istruzione.

Esercizio per casa - riepilogo

Istruzione	Codice decimale	Codice binario
di tipo R	0	000000
lw	35	100011
sw	43	101011
beq	4	000100

Progettare la tabella di verità della **CU** (solo per le righe necessarie) avente

- in ingresso: i 6 bit del codice operativo dell'istruzione
- in uscita: i 9 bit dei segnali di controllo da produrre:

  (**RegDst, ALUSrc, MemtoReg, RegWrite, MemRead ,MemWrite, Branch, ALUOp1, ALUOp0**)

Soluzione

Se i codici dei 4 tipi di istruzioni sono:

istruzione	codice decimale	codice binario
di tipo R	0	000000
lw	35	100011
sw	43	101011
beq	4	000100

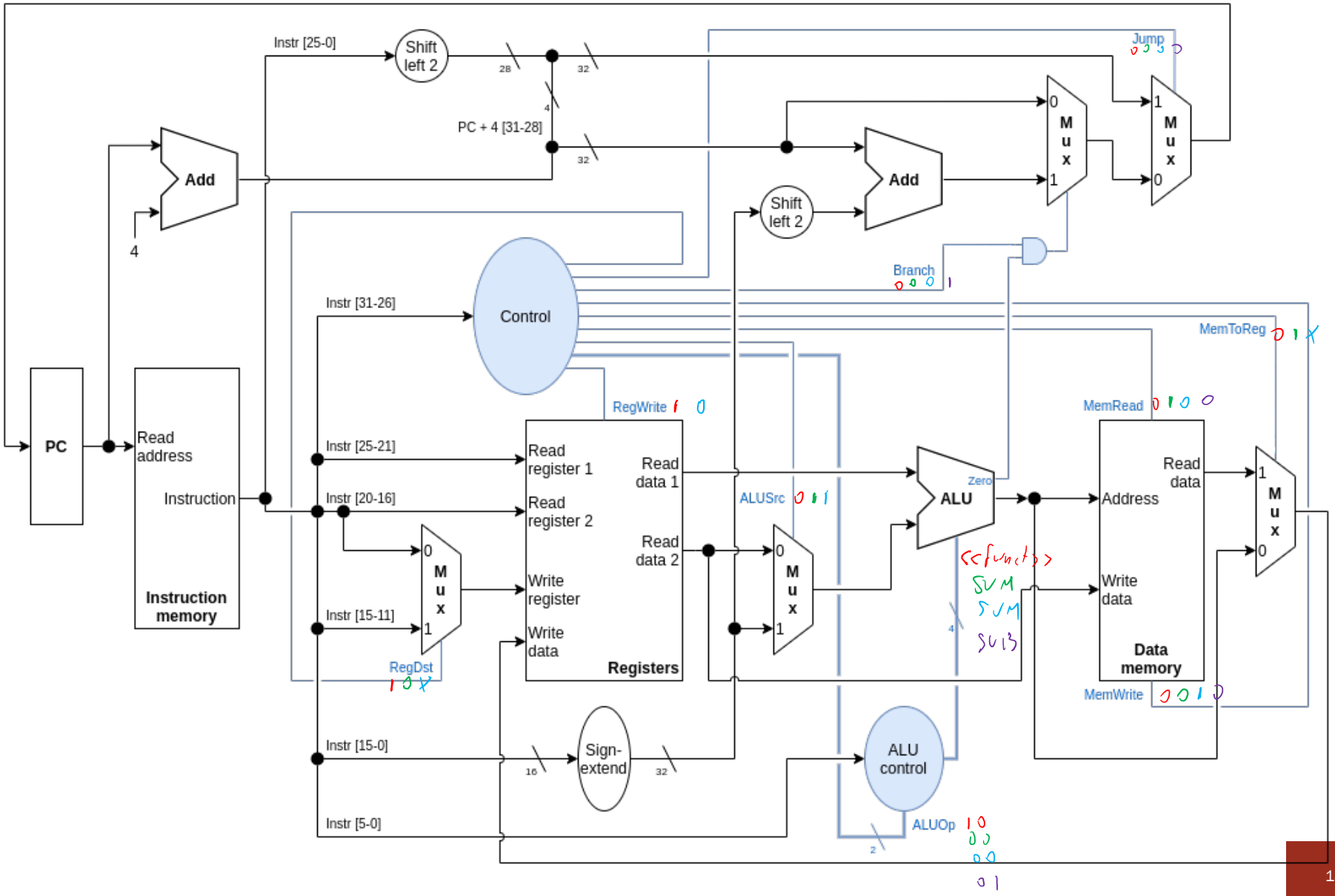
... e dobbiamo produrre i segnali dell'unità di controllo

Istruzione	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
000000	1	0	0	1	0 (X)	0	0	1	0
100011	0	1	1	1	1	0	0	0	0
101011	X	1	X	0	0 (X)	1	0	0	0
000100	X	0	X	0	0 (X)	0	1	0	1

Da cui possiamo produrre la PLA oppure le funzioni booleane necessarie

Per esempio **ALUSrc = Opcode0** **Branch e ALUOp0 = Opcode2** **MemWrite = Opcode3**

R, lw, sw, beq


Instr [25-0]
Shift left 2
28
32
4
PC + 4 [31-28]
32
Jump
Add
4
Add
Shift left 2
0
M
u
x
1
1
M
u
x
0
Branch
Instr [31-26]
Control
MemToReg
RegWrite
MemRead
PC
Read address
Instruction
Instruction memory
Instr [25-21]
Instr [20-16]
Instr [15-11]
RegDst
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALUSrc
Zero
ALU
Address
Read data
Write data
Data memory
MemWrite
Instr [15-0]
Instr [5-0]
16
Sign-extend
32
ALU control
ALUOp
2

Cosa fare se la Control Unit è malfunzionante?

SAPIENZA
UNIVERSITÀ DI ROMA

Possiamo identificare guasti hardware via software?

Control Unit malfunzionante

Segnali di controllo

	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
tipo R	1	0	0	1	X	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0	0
sw	X	1	X	0	X	1	0	0	0	0
beq	X	0	X	0	X	0	1	0	0	1
j	X	X	X	0	X	0	X	1	X	X

Cosa succede se la CU genera dei segnali errati?

Dobbiamo individuare:

- **quale combinazione di segnali venga generata**
- **quali istruzioni vengano influenzate** dalle nuove combinazioni e **cosa facciano**

Una volta individuate le «nuove» funzionalità (fallaci) delle istruzioni…

… possiamo cercare di scrivere un breve programma che evidenzia se la CPU sia malfunzionante o meno

Esempio 1: RegWrite ← Branch

Si ha il dubbio che, per difetto di progettazione della CU (e.g. cortocircuito):

- il segnale **RegWrite** sia determinato dal segnale **Branch**
  (ossia, se **Branch = 0** allora **RegWrite = 0** e se **Branch = 1** allora **RegWrite = 1**)

Si assume che:

- MemToReg = 1 solo per la lw ed altrimenti valga 0 (mai X)
- RegDest = 1 solo per le istruzioni di tipo R ed altrimenti valga 0 (mai X)

1) Individuare quali istruzioni ne sono affette e perché
2) Scrivere un breve programma che scriva in $s0 un valore che distingua se la CPU sia correttamente funzionante ($s0=1), o malfunzionante ($s0=0)

(1)
Le istruzioni affette da questo **guasto** sono:
- tutte le istruzioni che **modificano un registro** (tipo R e lw): lo lasceranno invariato
- **branch**: oltre ad effettuare il salto modificherà uno dei registri
  - rt sarà sovrascritto perché RegDest = 0
  - il valore scritto sarà il risultato della differenza usata per confrontare i due operandi perché assumiamo MemtoReg=0

Le istruzioni sw e jump, invece, funzioneranno correttamente (perché hanno RegWrite a 0 comunque)

Esempio 1: RegWrite ← Branch

	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
tipo R	1	0	0	**0**	X	0	**0**	0	1	0
lw	0	1	1	**0**	1	0	**0**	0	0	0
sw	**0**	1	**0**	0	X	1	0	0	0	0
beq	**0**	0	**0**	**1**	X	0	**1**	0	0	1
j	**0**	X	**0**	0	X	0	0	1	X	X

Assunzione | Difetto

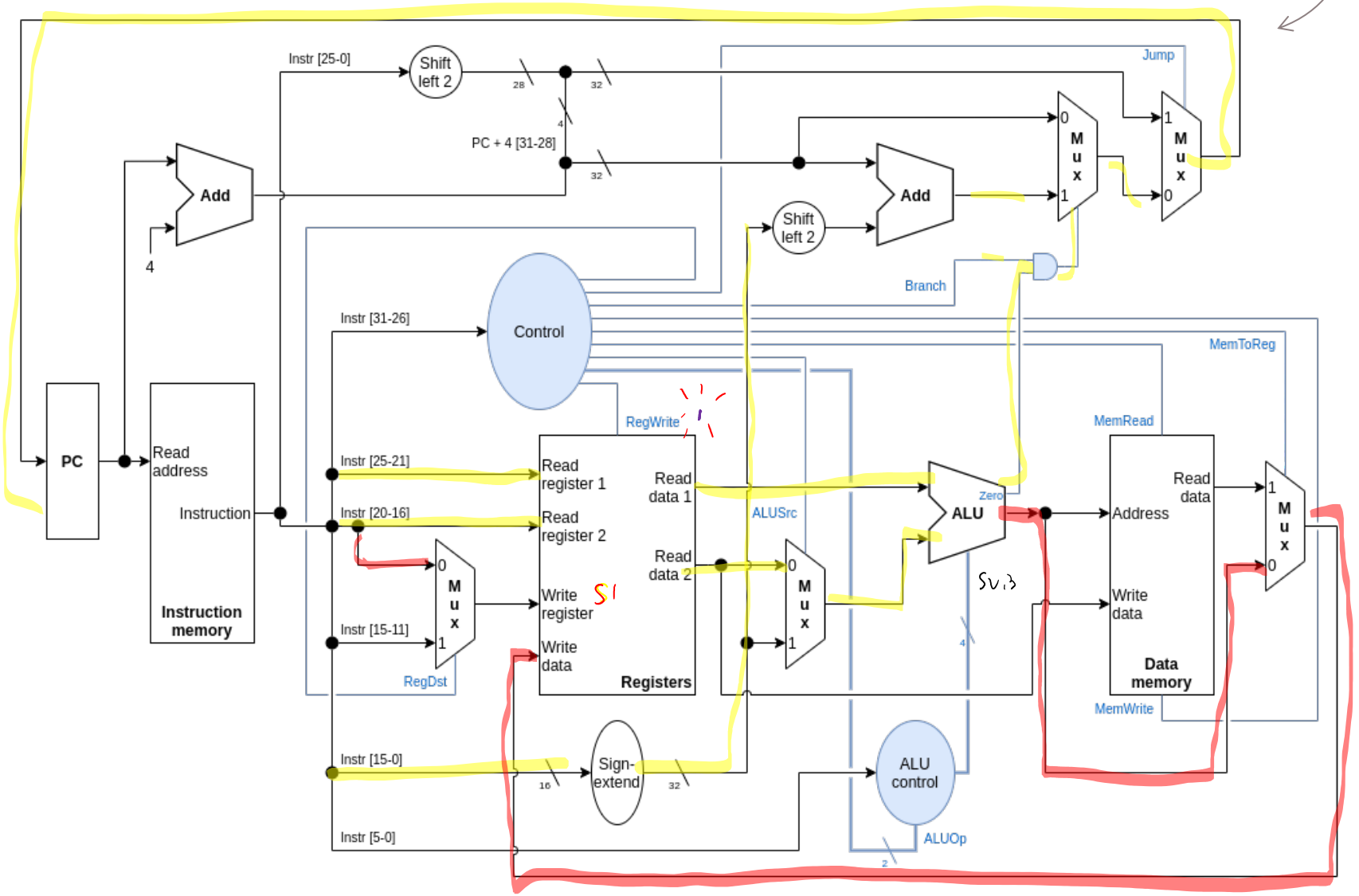
1) Individuare quali istruzioni ne sono affette e perché

Le istruzioni affette da questo **guasto** sono:
- tutte le istruzioni che **modificano un registro** (tipo R e lw): lo lasceranno invariato
- **branch**: oltre ad effettuare il salto modificherà uno dei registri
  - rt sarà il registro sovrascritto perché RegDest = 0
  - il valore scritto sarà il risultato della differenza usata per confrontare i due operandi perché assumiamo MemtoReg=0

Le istruzioni sw e jump, invece, funzioneranno correttamente (perché hanno RegWrite a 0 comunque)

RegWrite ← Branch

Qui: scarabocchi



? Evidenziare le linee attraversate dai segnali di `beq $s0,$s1,0x00000014`

Individuare il guasto: RegWrite ← Branch

(2)

Per individuare se la CPU sia malfunzionante creiamo un programma che lasci il valore **0** nel registro **$s0** se lo è, e scriva **1** se funziona correttamente, tenendo conto che:

- Non possiamo caricare un valore in un registro perché RegWrite = 0

Potremmo assumere che in memoria sia presente il valore 1 e provare a leggerlo

```
.data                          .text
uno:   .word  1                main:    lw $s0, uno
```

Oppure basta una qualsiasi istruzione che generi un valore diverso da 0 (una **tipo R** o una **li**), come una delle seguenti

```
su: nor  $s0, $zero, $zero # nega 0 e produce 11..11 = -1
    li   $s0, 1            # = 1
    addi $s0, $zero, 1     # = 1
```

Oppure potremmo costruire una **beq** che scriva 0 in **$s0** se la CPU è rotta, calcolando la differenza tra due valori uguali ($s0 e se stesso). Però, per distinguere il caso di CPU rotta da quello con CPU funzionante dobbiamo assumere che inizialmente $s0=1

```
.text                          # assumo che $s0=1
main:  beq $s0, $s0, su
```

MemWrite ← not(RegWrite)

Si ha il dubbio che in alcune CPU MIPS la Control Unit sia rotta, producendo il segnale di controllo **MemWrite attivo se e solo se NON è attivo il segnale RegWrite.**

Si assume che:

- MemToReg = 1 solo per la lw ed altrimenti valga 0 (mai X)
- RegDest = 1 solo per le istruzioni di tipo R ed altrimenti valga 0 (mai X)

a) Indicare quali delle istruzioni funzioneranno male e perché.

b) Scrivere un breve programma assembly MIPS che termini scrivendo nel registro **$s0** il valore **1 se il processore è guasto**, **altrimenti** vi scriva **0**.

MemWrite ← not(RegWrite)

	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
tipo R	1	0	0	1	X	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0	0
sw	0	1	0	0	X	1	0	0	0	0
beq	0	0	0	0	X	1	1	0	0	1
j	0	X	0	0	X	1	X	1	X	X

Assunzione | Difetto

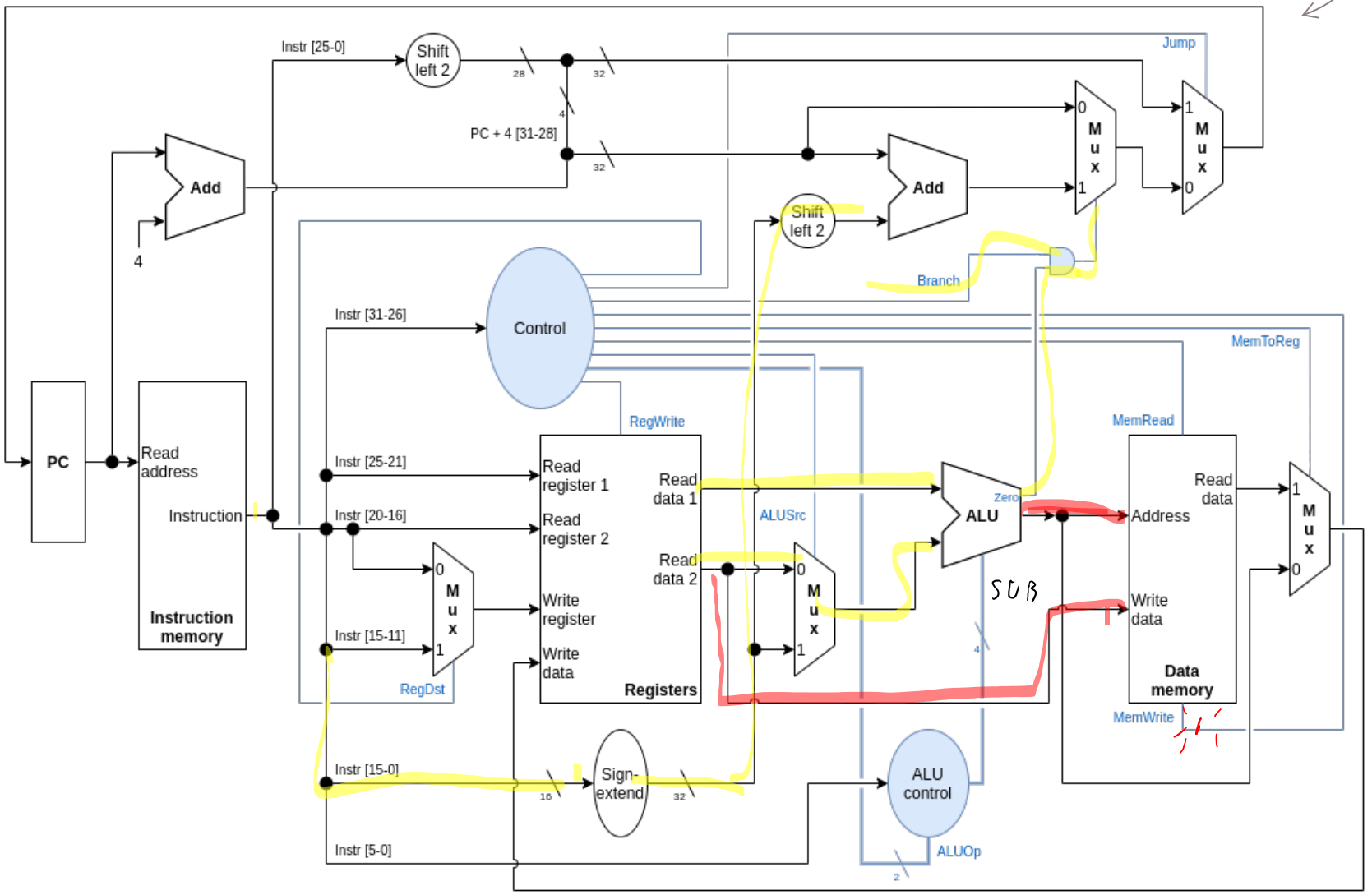
a) Indicare quali delle istruzioni funzioneranno male e perché.

a) «**MemWrite attivo se e solo se NON è attivo il segnale RegWrite»** vuol dire che
- se RegWrite = 0 allora MemWrite = 1
- se RegWrite = 1 allora MemWrite = 0

Quindi sono danneggiate le istruzioni che hanno i due segnali entrambi a 0 o entrambi a 1
- **lw:** funziona correttamente (1, 0)
- **sw:** funziona correttamente (0, 1)
- **beq:** salta correttamente MA INOLTRE SCRIVE IN MEMORIA (0, 1) – invece che (0, 0)
- **j:** salta correttamente MA INOLTRE SCRIVE IN MEMORIA (0, 1) – invece che (0, 0)
- **tipo R:** funzionano correttamente (1, 0)

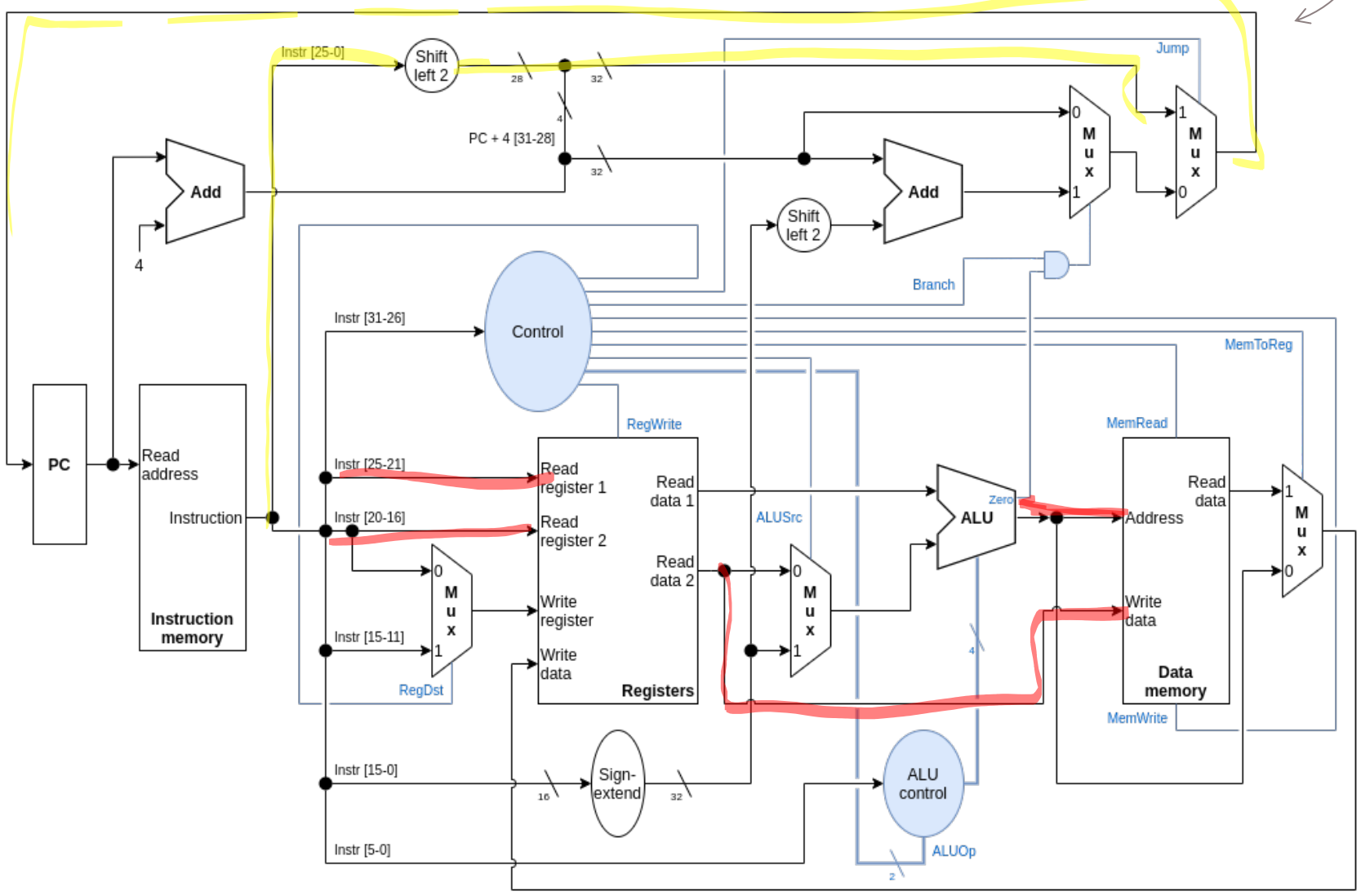
MemWrite ← not(RegWrite)

Qui: scarabocchi



MemWrite ← not(RegWrite)

Qui: scarabocchi



MemWrite ← not(RegWrite)

b) Per distinguere se la CPU è rotta dobbiamo quindi usare una delle due istruzioni di salto:

beq **$rs**, **$rt**, ***etichetta*** che memorizza

- il valore del registro rt all'indirizzo calcolato dalla ALU come la differenza dei due operandi rs e rt (per realizzare il confronto)

j ***etichetta*** che memorizza:

- il valore del registro indicato dal campo rt della istruzione (i 5 bit [20–16]) e che si trovano in mezzo all'indirizzo di destinazione

- all'indirizzo che si ottiene in uscita dalla ALU, di cui però non sappiamo quale funzione venga svolta

Conviene quindi usare la **beq** poiché possiamo controllarla meglio:

```
move    $s0, $zero      # $s0 = 0
sw      $s0, 0          # memorizzo 0 all'indirizzo 0
li      $s1, 1          # $s1 = 1
beq     $s1, $s1, On    # salto di zero istruzioni = continua
On:                     # se rotta, la CU memorizza 1 all'ind. 0
lw      $s0, 0          # carico il contenuto dell'indirizzo 0
```

WARNING