Architettura degli Elaboratori

Progetto CPU (ciclo singolo)

slide a cura di Salvatore Orlando e Marta Simeoni

Processore: Datapath & Control

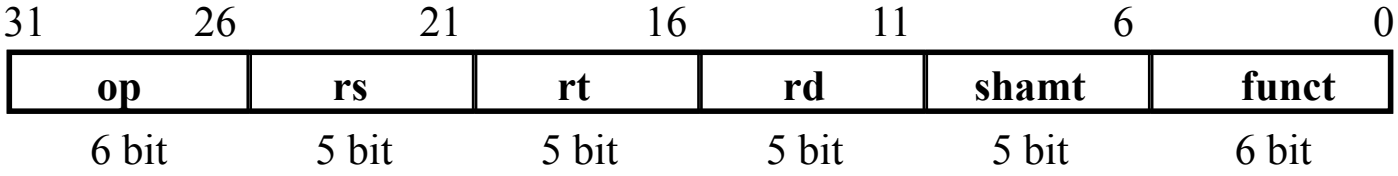
- Progetto di un processore MIPS-like semplificato

- In grado di eseguire solo:
  - istruzioni di memory-reference: `lw`, `sw`
  - istruzioni arithmetic-logic: `add`, `sub`, `and`, `or`, `slt`
  - istruzioni di control-flow: `beq`, `j`

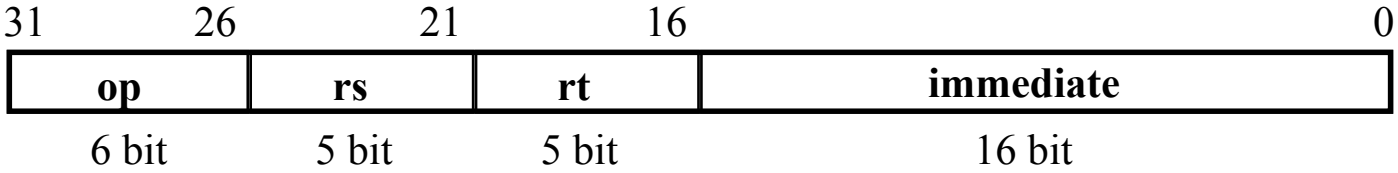
Rivediamo i formati delle istruzioni

Le istruzioni MIPS sono tutte lunghe 32 bit. I tre formati che considereremo:

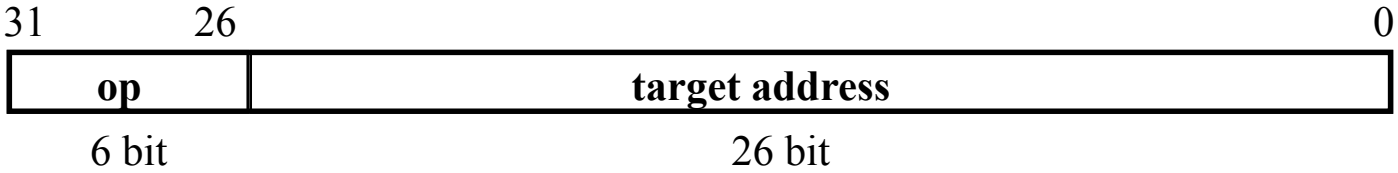
- R-type



- I-type



- J-type



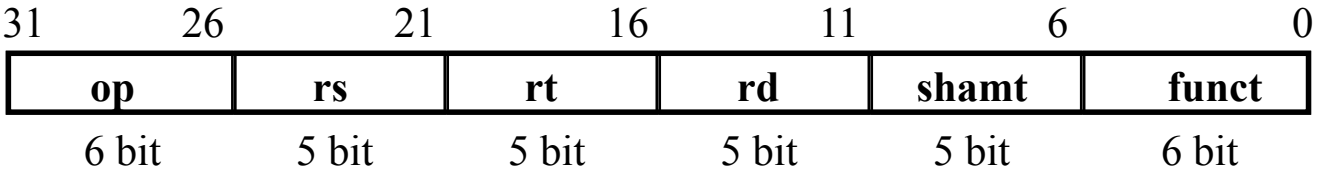
- Campi:
  - op: codice operativo dell'istruzione
  - rs, rt, rd: dei registri sorgente (rs, rt) e destinazione (rd)
  - shamt: shift amount (è diverso da 0 solo per istruz. di shift)
  - funct: seleziona le varianti dell'operazione specificata in op
  - immediate: offset dell'indirizzo (load/store) o valore immediato (op. aritmetiche)
  - target address: indirizzo target di un'istruzione di jump

ISA di un MIPS-lite

ADD e SUB

add rd, rs, rt

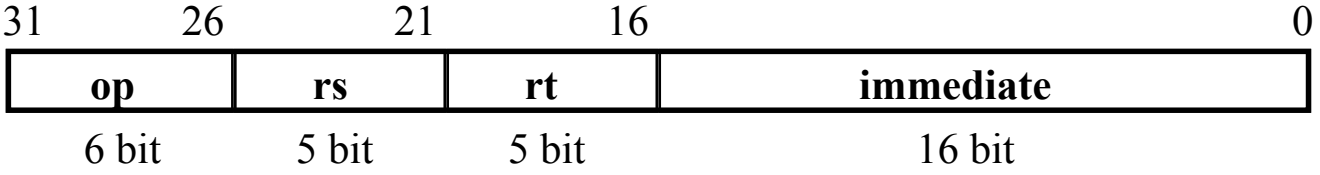
sub rd, rs, rt



LOAD and STORE Word

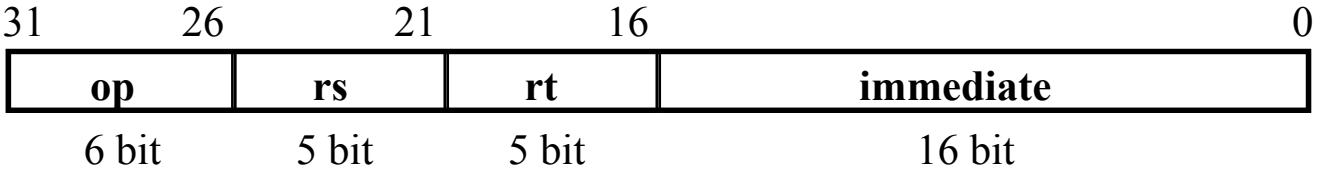
lw rt, imm16 (rs)

sw rt, imm16 (rs)



BRANCH:

beq rs, rt, imm16



Esempi

add $8, $17, $18 Formato istruzione:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

lw $1, 100($2) Formato istruzione:

35	2	1	100
op	rs	rt	16 bit offset

Passi di progetto

1. Analizza il set di istruzioni => verifica i requisiti del datapath
   - il datapath deve includere gli elementi di memoria corrispondenti ai registri dell'ISA
     - tipicamente sono necessari altri registri, usati internamente o non referenziabili direttamente attraverso l'ISA
       - **es.: PC (Program counter)**
   - analizza la semantica di ogni istruzione, data in termini di *trasferimenti tra registri*, ed eventuali *operazioni tra i registri*
     - Il datapath deve fornire i cammini per permettere tutti i *register transfer* necessari, e gli accessi alla memoria
2. Seleziona i vari componenti del datapath (es. ALU) e stabilisci la metodologia di clocking
3. Assembla il datapath in accordo ai requisiti, aggiungendo i segnali di controllo
4. Analizza l'implementazione di ogni istruzione per determinare il *setting* dei segnali di controllo che provocano i vari *register transfer*
5. Assembla la *logica di controllo* in accordo al punto 4.

Implementazione generica a singolo ciclo

- Proviamo a progettare una CPU in cui ogni istruzione viene eseguita all'interno di un *singolo ciclo di clock*

- Dobbiamo accedere alla Memoria e al Register file, nel seguito indicati con
  - M[x] : word all'indirizzo x
  - R[y] : registro identificato dall'id numerico y

Implementazione generica a singolo ciclo

- Implementazione generica di un'istruzione:
  - usa il registro Program Counter (PC), interno alla CPU, per fornire alla memoria l'indirizzo dell'istruzione
  - leggi l'istruzione dalla memoria (fetch)
  - interpreta i campi dell'istruzione per decidere esattamente cosa fare (decode)
  - usa l'ALU per l'esecuzione (execute)
    - `add/sub/and/or/slt` usano l'ALU per le operazioni corrispondenti, e il Register File per accedere ai registri
    - le istruzioni di `lw/sw` usano l'ALU per calcolare gli indirizzi di memoria
    - l'istruzione di `beq` usa l'ALU per controllare l'uguaglianza dei registri
  - modifica il PC e reitera il ciclo

$\Longrightarrow$ l'ALU, il Register File, il PC dovranno quindi far parte del Datapath

Anche se per comodità rappresenteremo la memoria assieme agli altri elementi del Datapath, essa non fa logicamente parte della CPU

RTL dettagliato delle varie istruzioni

- Usiamo RTL (Register-Transfer Language), un linguaggio per esprimere i *trasferimenti tra registri,* per definire la *semantica* di ogni istruzione
- Ricordiamo che BEQ adotta un indirizzamento di tipo *PC-relative*

Per tutte le istruzioni, dobbiamo come prima cosa effettuare il *fetch*

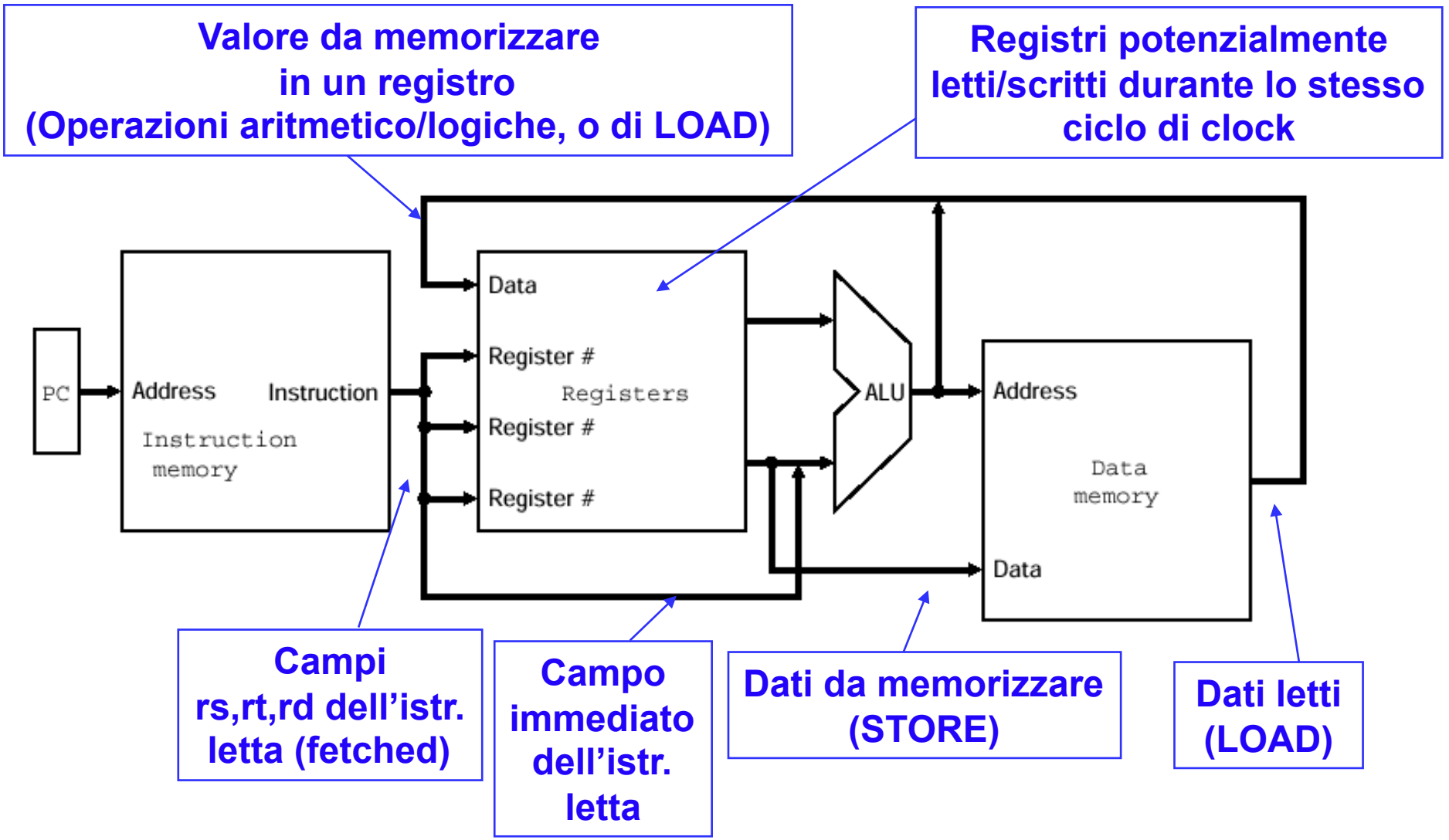
op | rs | rt | rd | shamt | funct = M[PC]

op | rs | rt | Imm16 = M[PC]

op | 26bit address = M[PC]

istruzioni	Trasferimenti tra registri	
ADD	**R[rd] <– R[rs] + R[rt];**	**PC <– PC + 4;**
SUB	**R[rd] <– R[rs] – R[rt];**	**PC <– PC + 4;**
LOAD	**R[rt] <– M[R[rs] + sign_ext(Imm16)];**	**PC <– PC + 4;**
STORE	**M[R[rs] + sign_ext(Imm16)] <– R[rt];**	**PC <– PC + 4;**
BEQ	**if (R[rs] == R[rt]) then PC <– PC + 4 + (sign_ext(Imm16) << 2);** **else PC <– PC + 4;**	

Visione astratta di una possibile implementazione



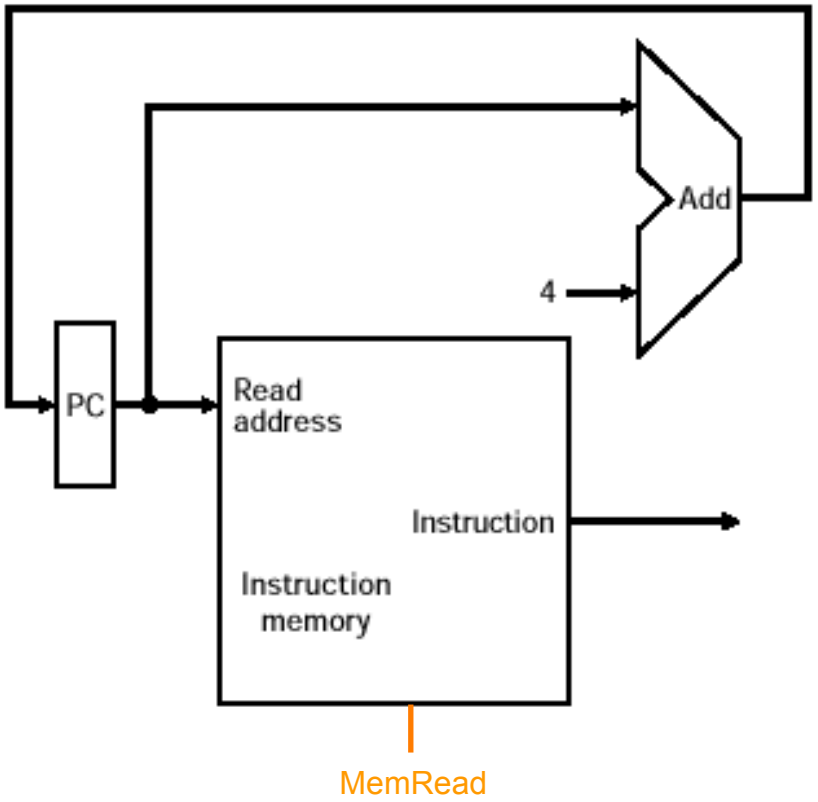
Visione astratta di una possibile implementazione

- Incremento PC ?
- Estensione campi immediati ?
- BEQ ?

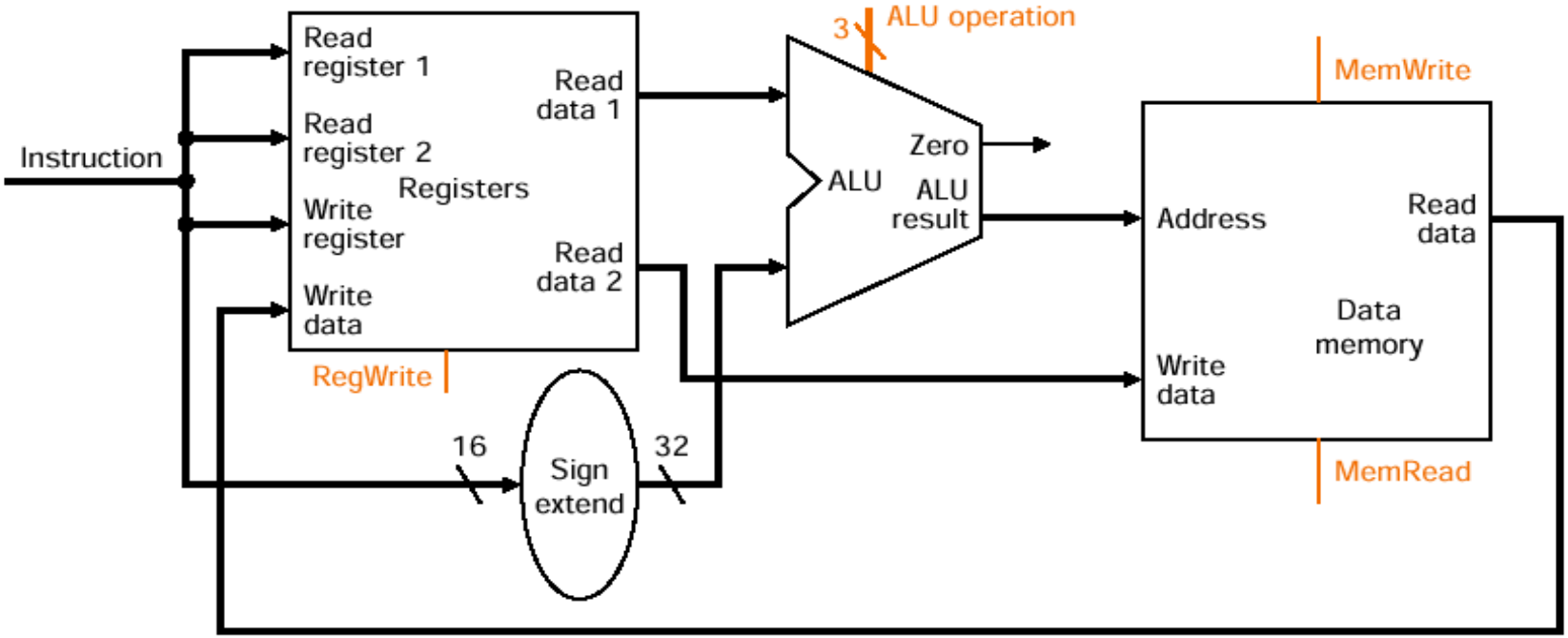
Incremento del PC

- Addizionatore aggiuntivo
  - necessario per realizzare, all'interno dello stesso ciclo di clock
    - il fetch dell'istruzione
    - l'incremento del PC
- Non possiamo usare l'ALU principale, perché questa è già utilizzata per eseguire le istruzioni
  - stiamo implementando una CPU a singolo ciclo

    ⟹ risorse replicate
- Nota che dalla memoria istruzioni viene letta una nuova istruzione ad ogni ciclo di clock
  - Il segnale di MemRead deve essere sempre affermato



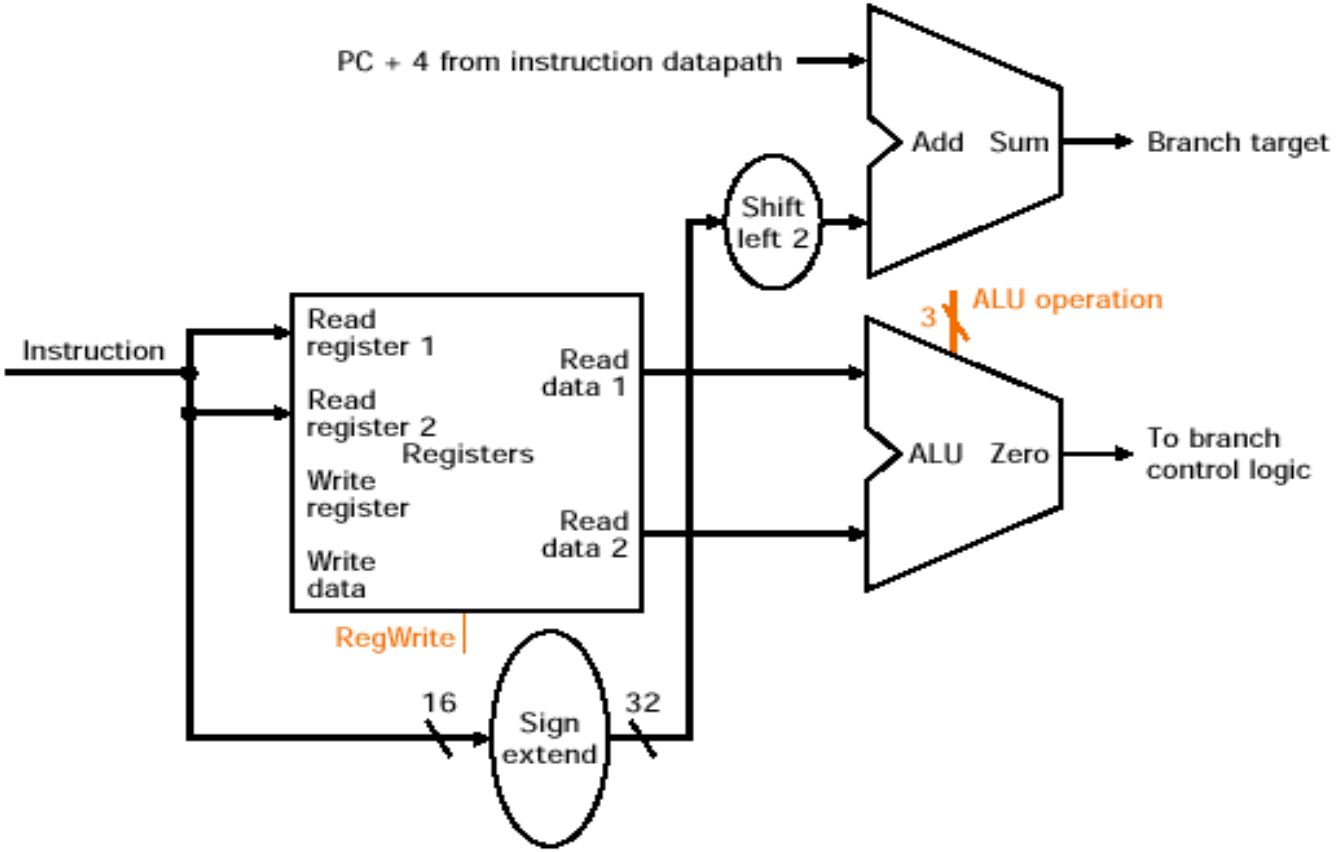
Estensione del segno di operandi immediati



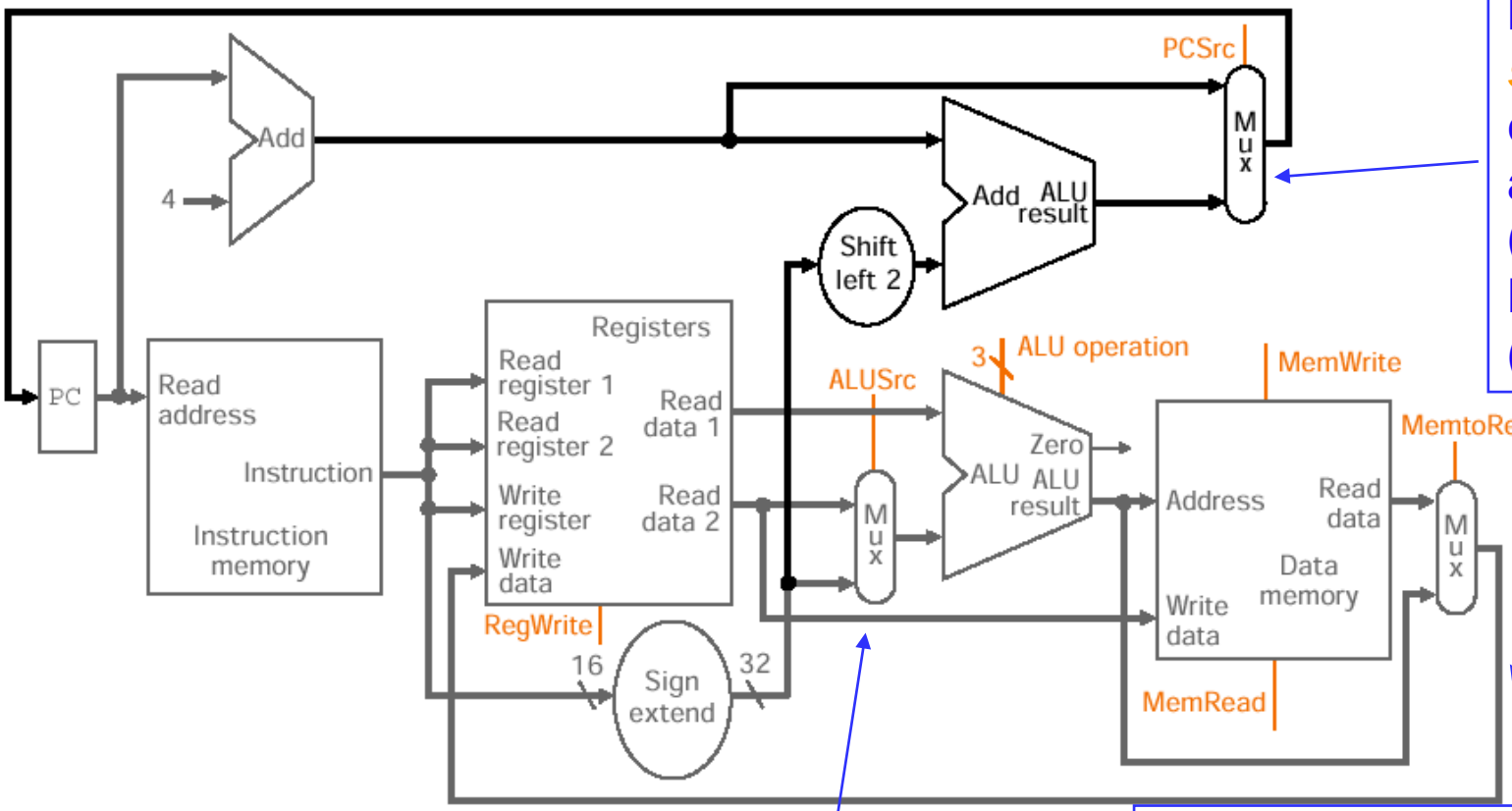
- I 16 bit del campo *immediato* dell'istruzione (es. istruzioni di LOAD/STORE) sono estesi di segno (16b->32b) prima di essere sommati con il registro R[rs]
- L'indirizzo così calcolato (R[rs] + sign_ext(Imm16)) viene usato per accedere alla Memoria Dati in lettura/scrittura
  - M[R[rs] + sign_ext(Imm16)]

Calcolo dell'indirizzo di BRANCH

- Ulteriore *addizionatore* ⟹ ancora risorse replicate
- Necessario per realizzare il calcolo dell'indirizzo di salto dei branch
  PC <– PC + 4 + (sign_ext(Imm16) << 2)
- Non possiamo usare l'ALU, perché viene già utilizzata per eseguire l'operazione di confronto (sottrazione)



Integrazione componenti tramite *multiplexer*



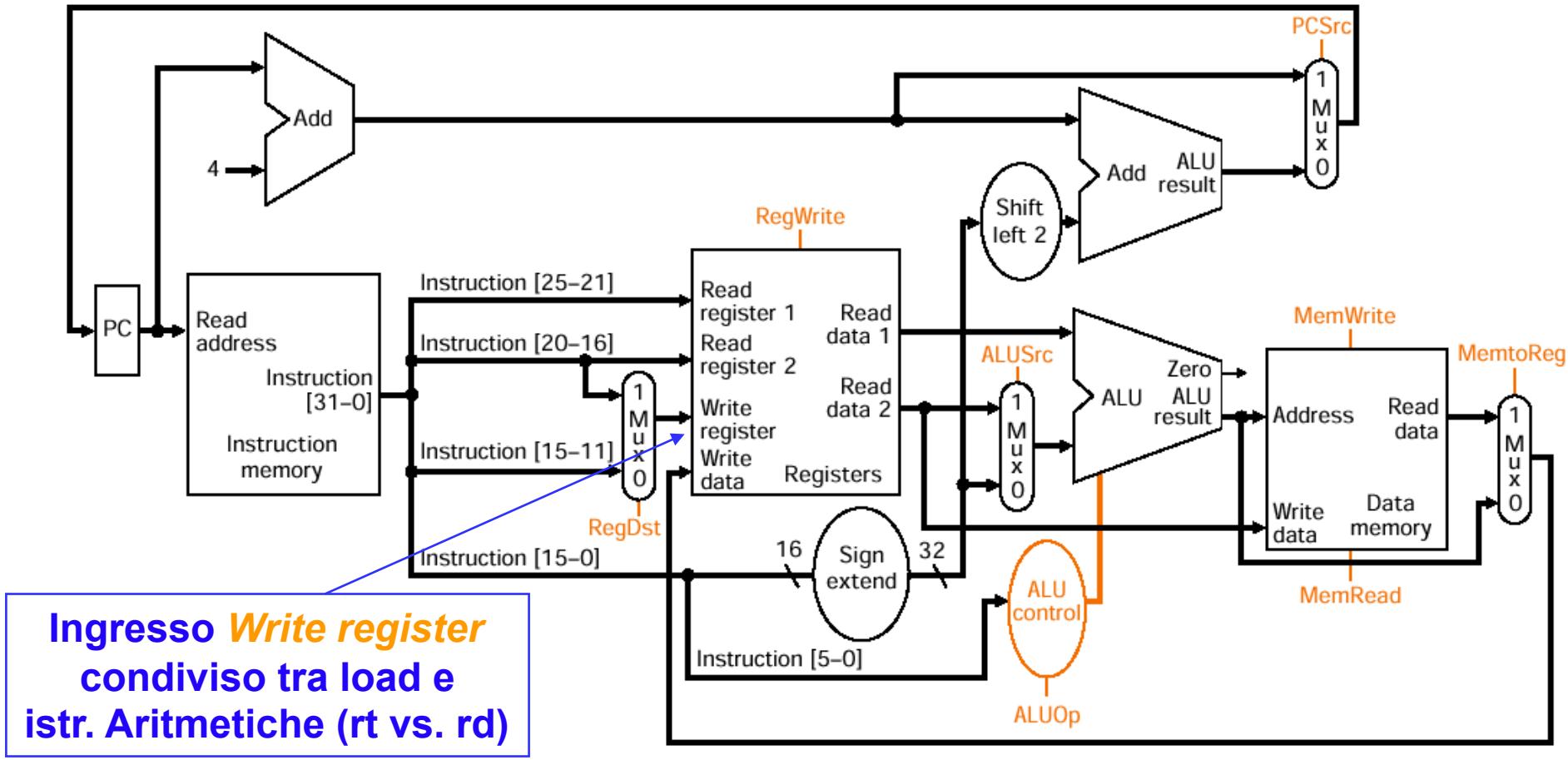
Ingresso *PC source* condiviso tra address Beq (caso taken) e le altre istr. (PC=PC+4)

Ingresso *ALU secondo operando* condiviso tra istr. R-type e I-type

Ingresso *Write data* Reg. File condiviso tra istr. di Load e altre istr. aritmetico/logico

Risorse replicate (anche memoria) per permettere l'esecuzione di una qualsiasi istruzione durante lo stesso ciclo di clock

Aggiunta multiplexer ingresso Register file



Ingresso *Write register* condiviso tra load e istr. Aritmetiche (rt vs. rd)

- rs: instr[25-21] rt: instr[20-16] rd: instr[15-11]
- LOAD: R[rt] <– M[R[rs] + sign_ext(Imm16)]; ⇒ rt usato come target
- ADD: R[rd] <– R[rs] + R[rt]; ⇒ rd usato come target

Controllo ALU

- Dobbiamo definire il circuito di controllo per calcolare i 3-bit di controllo dell'ALU (Operation) da assegnare come segue in base al tipo di istruzione, ovvero ai campi op e funct dell'istruzione:
  - 000 operazione di and
  - 001 operazione di or
  - 010 operazione di add, lw, sw
  - 110 operazione di sub e beq
  - 111 operazione di slt

  - il circuito sarà a 2 livelli:
    - Il 1° livello calcolerà ALUOp= ($ALUOp_1$ $ALUOp_0$) in base all'op code +
    - Il 2° livello calcolerà effettivamente Operation in base al campo funct e a ALUOp
- Il circuito di 1° livello dovrà semplicemente definire la configurazione dei bit ($ALUOp_1$ $ALUOp_0$) sulla base di op:
  - 00 se lw, sw (Operation=010)
  - 01 se beq (Operation=110)
  - 10 se arithmetic/logic (Operation dipende dalla specifica istruzione -> vedi campo funct)

Controllo ALU

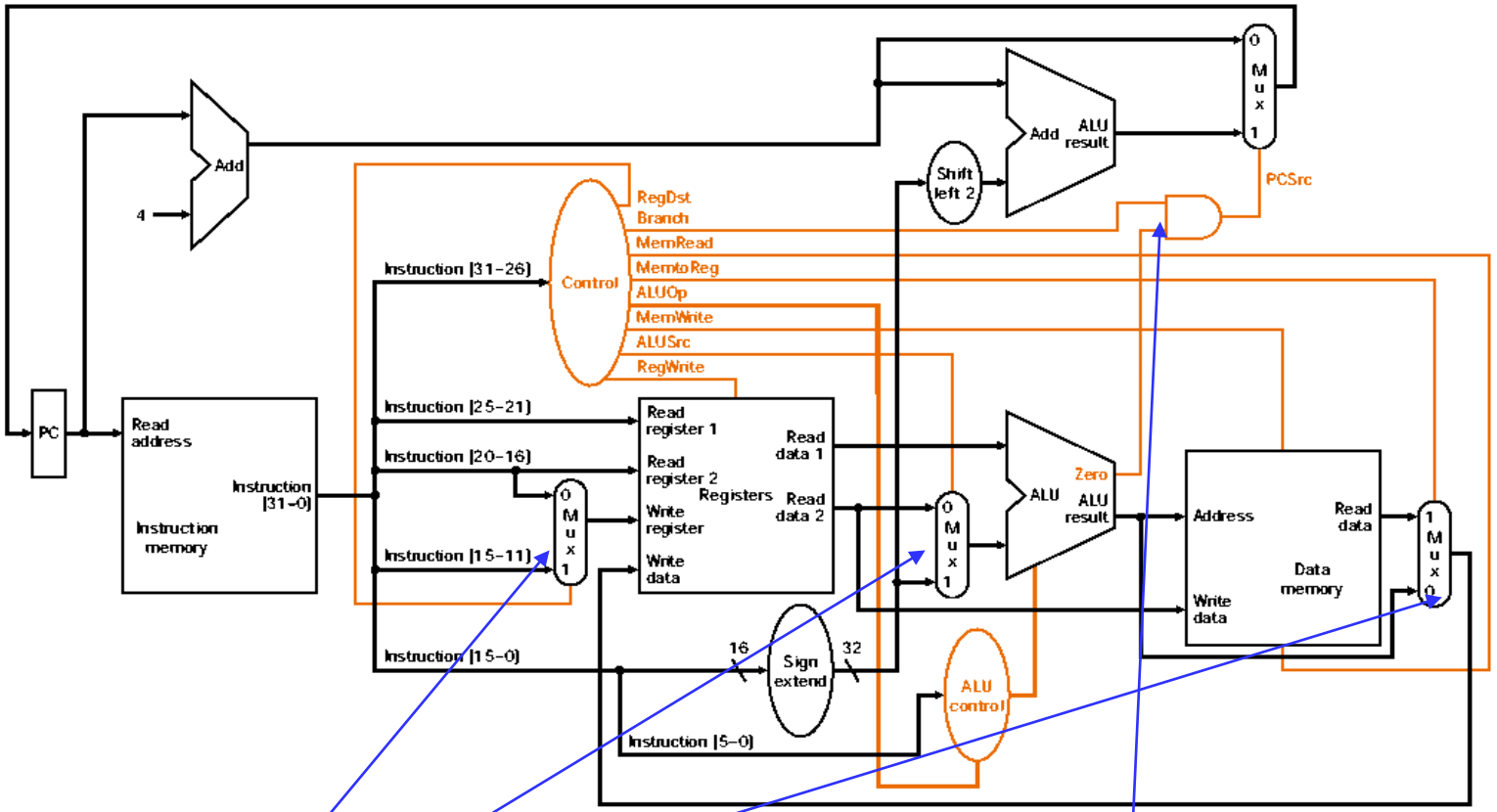
- Definiamo ora la tabella di verità che sulla base di ALUOp e funct determina i 3 bit del controllo dell'ALU (OPERATION)

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

lw/sw (somma)
beq (sottrazione)
add (somma)
sub (sottrazione)
and (and)
or (or)
slt (sottr. + slt)

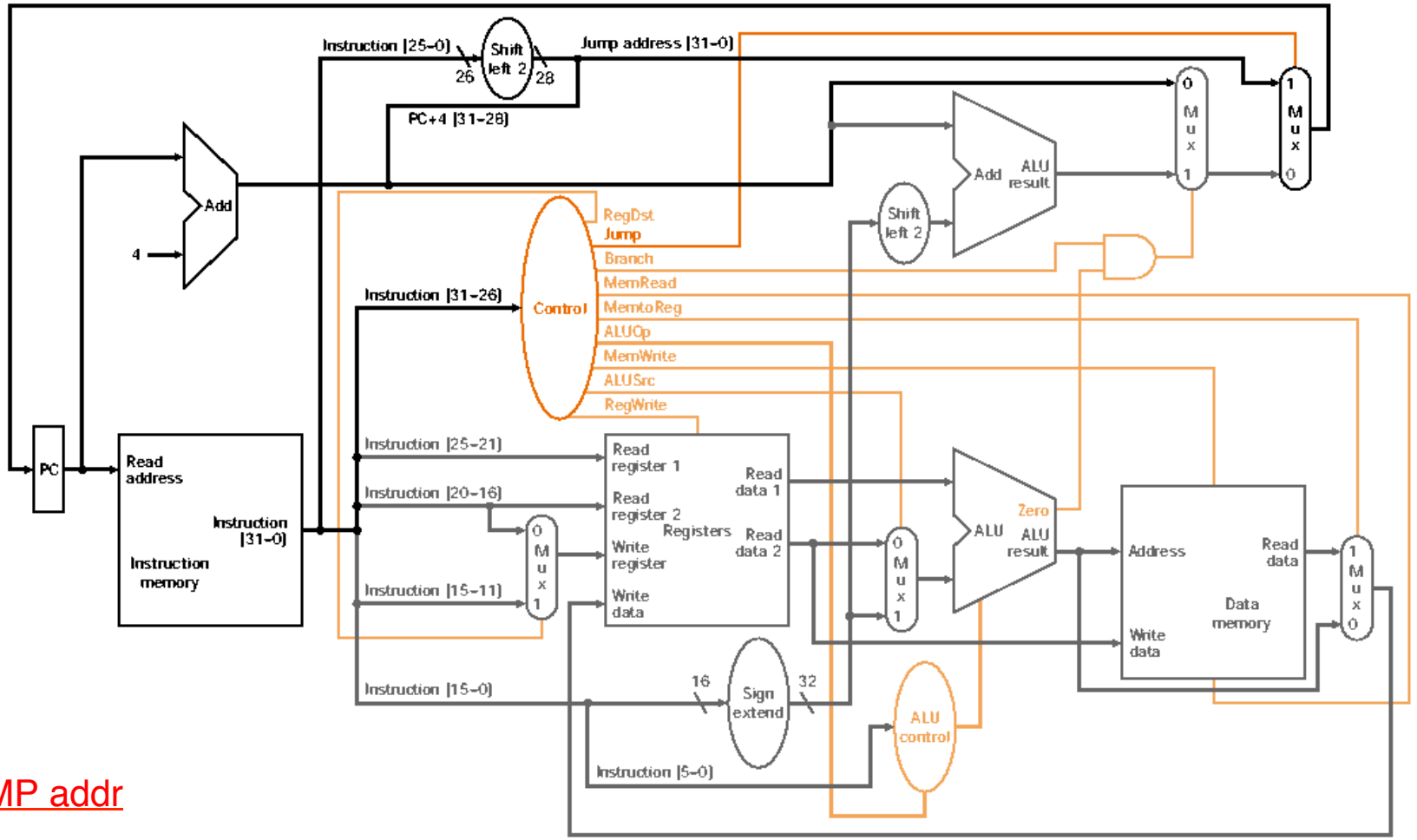
- A partire dalla tabella qui sopra possiamo definire il circuito ALUControl per il calcolo di Operation

Datapath completo con Memoria e Controllo



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Datapath esteso per l'esecuzione delle *jump*

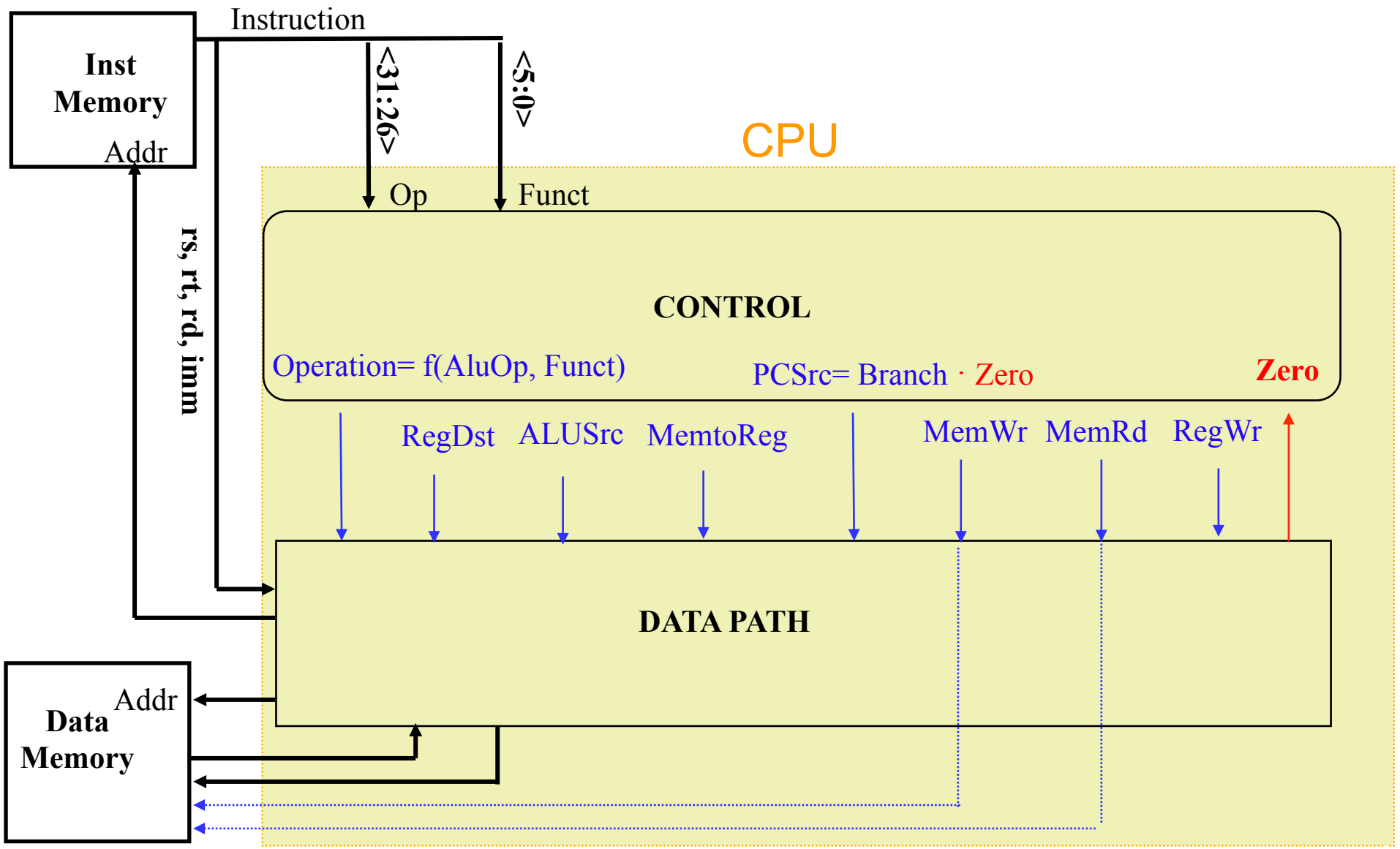


JUMP addr

op | Imm26 = M[PC]; PC <– (PC + 4)[31-28] || (Imm26 << 2);

MUX aggiuntivo, con relativo segnale di controllo Jump

Componenti CPU (Datapath+Control) e Memoria



Controllo a *singolo ciclo*

Il controllo per la realizzazione a *singolo* ciclo è molto semplice

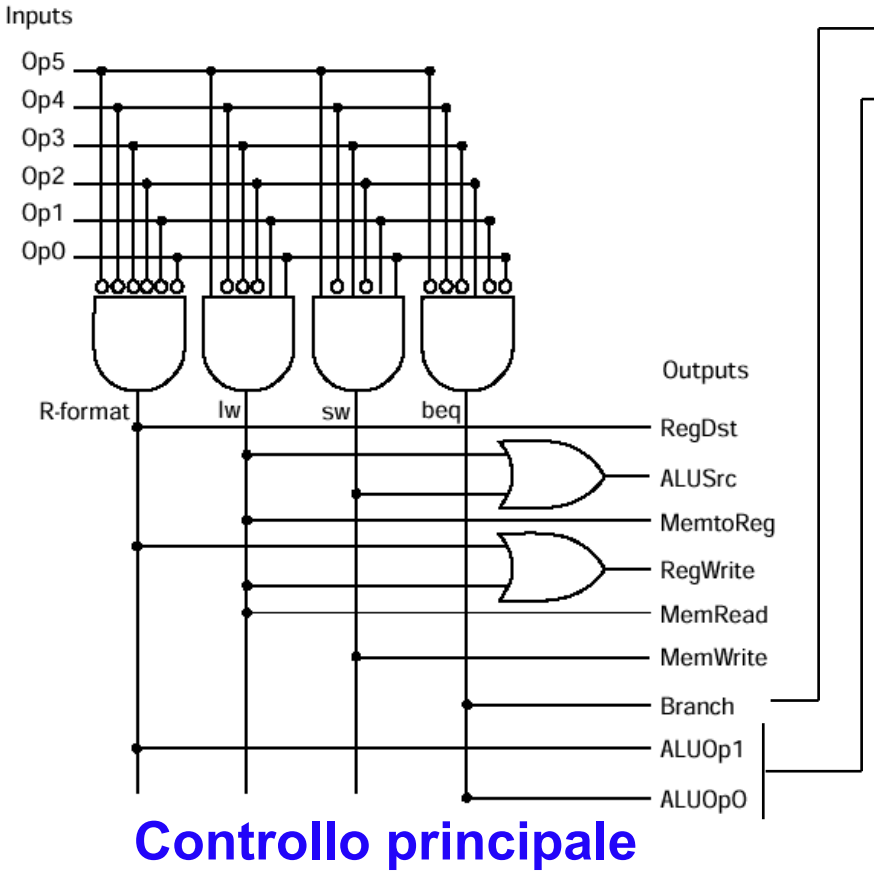
- definito da una coppia di *tabelle di verità*
- circuito combinatorio (non sequenziale !!)

Il controllo principale si basa sul codice dell'operazione da eseguire:

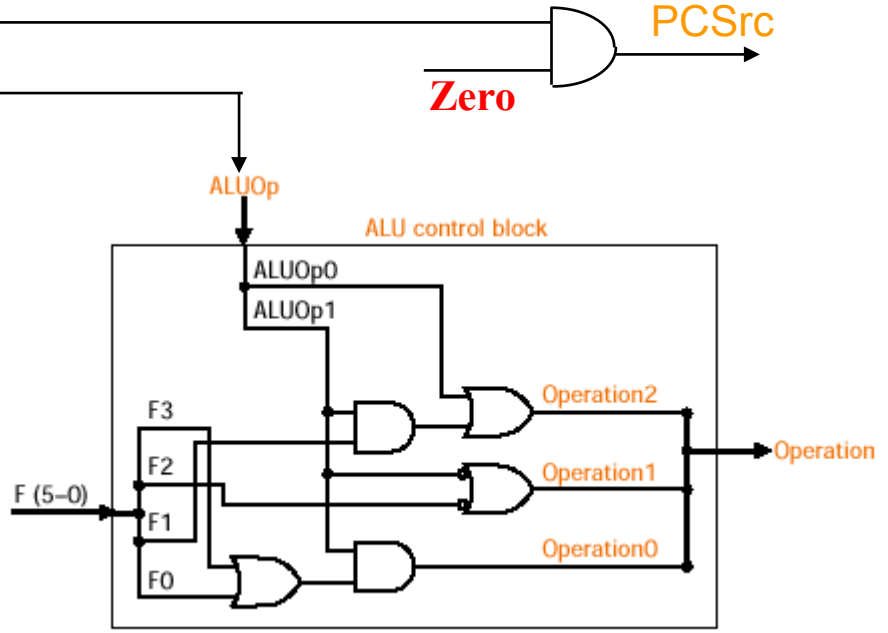
	op5	op4	op3	op2	op1	op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

Il controllo secondario determina l'ingresso all'ALU, ovvero il segnale Operation

Controllo a *singolo ciclo*



Controllo principale

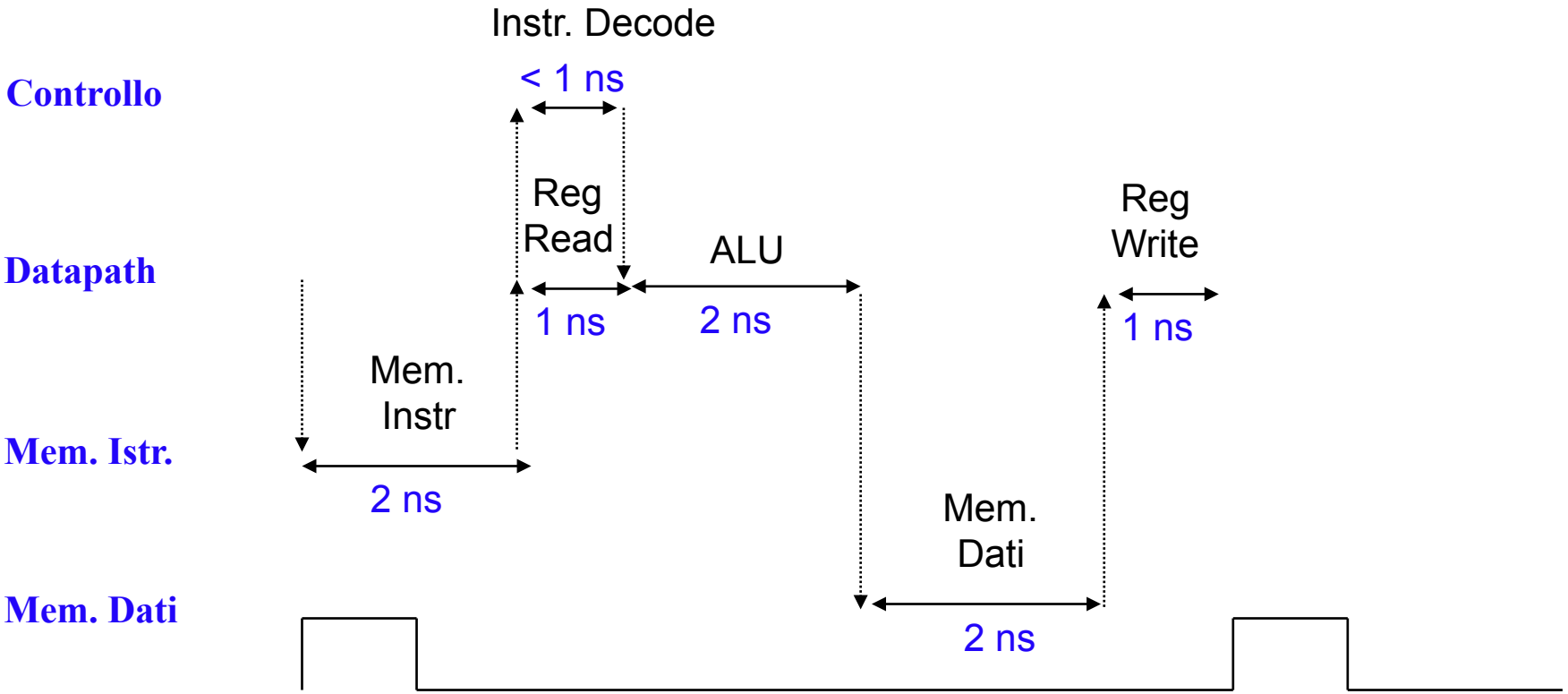


Controllo secondario che determina l'ingresso dell'ALU: *Operation*

Controllo a *singolo ciclo*

- Il controllo della CPU a singolo ciclo è combinatorio
- Il datapath è invece un circuito sequenziale
  - i suoi output dipendono anche dal valore dei registri
  - es. Zero, oppure l'indirizzo della memoria dati, oppure il valore da immagazzinare in memoria in conseguenza di una store, dipendono dai valori dello stato interno del Datapath (ovvero dal contenuto dei registri)
- Dobbiamo attendere che tutti i circuiti siano stabili, sia quelli del datapath che quelli del controllo, prima di attivare il fronte di salita/discesa del clock
- Clock in AND con i segnali di controllo di scrittura (registri/memoria)
  - i valori vengono scritti in corrispondenza del fronte di salita/discesa del clock solo se i segnali relativi sono affermati
- Ciclo di clock determinato sulla base del cammino più lungo che i segnali elettrici devono attraversare
  - es.: l'istruzione lw è quella più costosa: mem. istr. - Reg. File (Read) - ALU e Adders - Mem. Dati - Reg. File (Write)
  - i circuiti del controllo agiscono in parallelo alla lettura dei registri

Determiniamo il ciclo di clock per LW



Ipotizziamo costi (in ns) per le varie componenti

Mem. Istr/Dati: 2 ns Reg. File: 1 ns ALU: 2 ns Control: < 1 ns

Consideriamo l'istruzione LW, che abbiamo detto essere la più costosa

- è l'unica che usa sia il Register File in lettura/scrittura che la Memoria dati
- ciclo di clock lungo *8 ns*

Problemi con il singolo ciclo

- Ciclo singolo e di lunghezza fissa penalizza le istruzioni veloci
- Anche se *complesso*, si potrebbe realizzare una CPU a ciclo di clock variabile
- Quali i vantaggi?
  - istruzioni diverse dalla lw eseguite in un tempo < 8 ns
  - se il ciclo fosse fisso, sarebbero invece sempre necessari 8 ns
- Analizziamo quanto costa, in termini delle unità del Datapath usate, eseguire le varie istruzioni usando un ciclo di clock variabile

Classe istr.	Unità funzionali utilizzate					ciclo
Formato R	**Mem. Istr.**	**Reg (read)**	**ALU**		**Reg (wr)**	**6 ns**
Load	**Mem. Istr.**	**Reg (read)**	**ALU**	**Mem. dati**	**Reg (wr)**	**8 ns**
Store	**Mem. Istr.**	**Reg (read)**	**ALU**	**Mem. dati**		**7 ns**
Branch	**Mem. Istr.**	**Reg (read)**	**ALU**			**5 ns**
Jump	**Mem. Istr.**					**2 ns**
	2 ns	**1 ns**	**2 ns**	**2 ns**	**1 ns**	

Ciclo fisso vs. variabile

- Si consideri di conoscere che in un generico programma, le istruzioni sono combinate in accordo a questo mix
  - 24% load 12% store 44% formato-R
  - 18% branch 2% jump
- Qual è la lunghezza media (periodo medio) del ciclo di clock nell'implementazione a *ciclo variabile* ?
  - Periodo medio = $8 \times 24\% + 7 \times 12\% + 6 \times 44\% + 5 \times 18\% + 2 \times 2\% = 6.3$ ns
- Le prestazioni della CPU sono calcolabili rispetto a NI (Numero Istruzioni eseguite da un programma):
  - $T_{var} = NI \times periodo = NI \times 6.3$ (variabile)
  - $T_{fisso} = NI \times periodo = NI \times 8$ (fisso)
- Facendo il rapporto:
  - $T_{fisso} / T_{var} = 8 / 6.3 = 1.27$ (l'implem. a clock variabile è l' 1.27 più veloce !)
- Se consideriamo istruzioni più complesse della lw, come le istruzioni FP di moltiplicazione, l'implementazione a ciclo fisso risulta ulteriormente penalizzata
  - vedi esempio libro