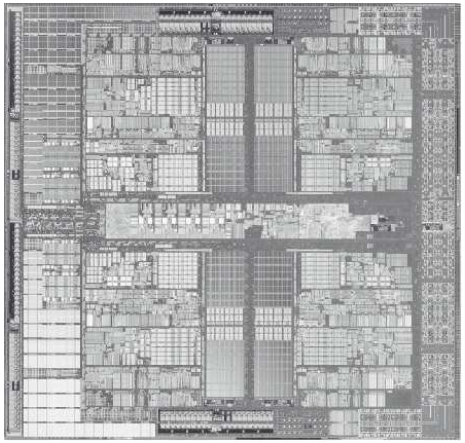
Università degli Studi di Cassino e del Lazio Meridionale



Corso di Calcolatori Elettronici

Realizzazione del Data path a ciclo singolo

Anno Accademico 2011/2012

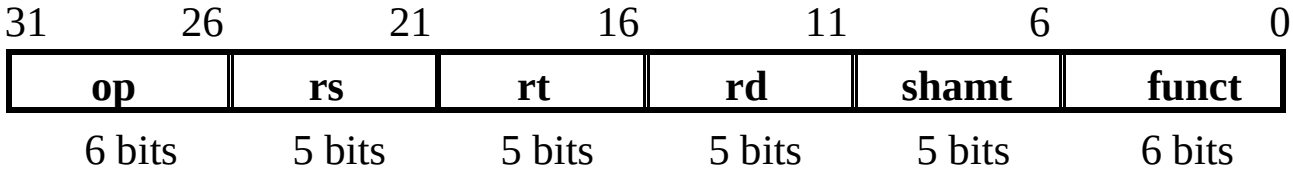
Francesco Tortorella

Realizzazione del data path

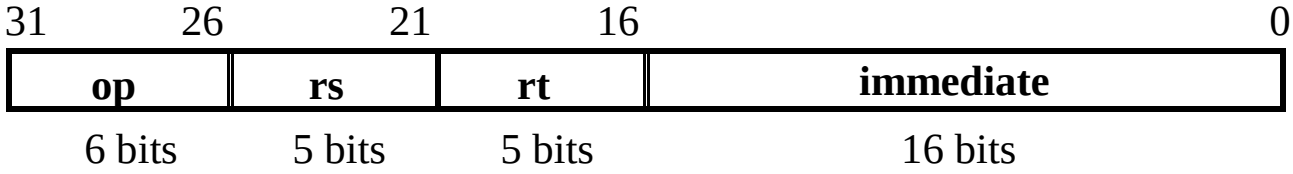
- 1. Analizzare l'instruction set => Specifiche sul datapath
  - il significato di ciascuna istruzione è dato dai *register transfers*
  - il datapath deve includere elementi di memoria per i registri necessari alla ISA
  - il datapath deve supportare ciascun trasferimento tra registri
- 2. Selezionare l'insieme di componenti del datapath e stabilire una metodologia di tempificazione
- 3. Costruire il datapath rispettando le specifiche
- 4. Analizzare l'implementazione di ciascuna istruzione per determinare i punti di controllo che abiliteranno i trasferimenti
- 5. Costruire la control logic

Formati Istruzioni del MIPS

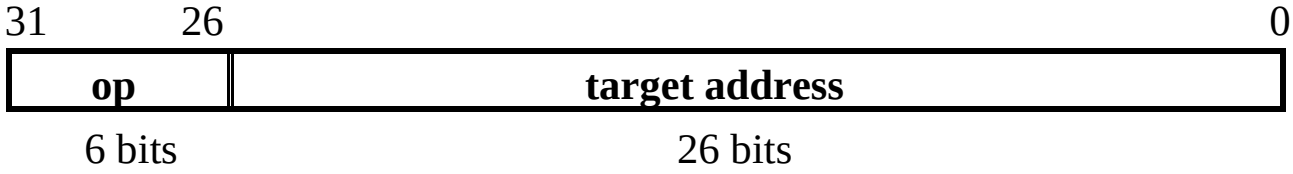
- R-type



- I-type

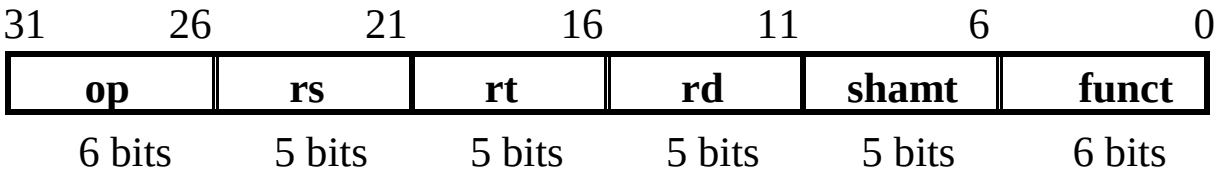


- J-type

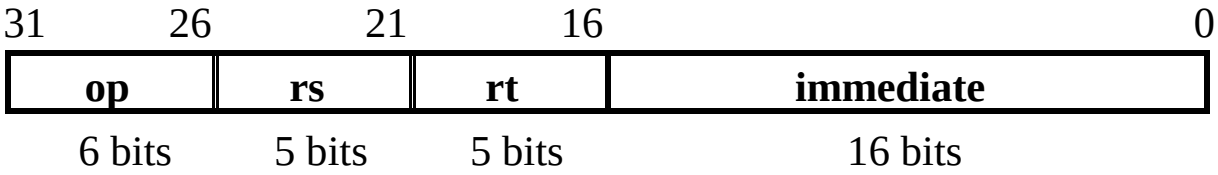


Passo 1a: Il sottoinsieme MIPS-lite

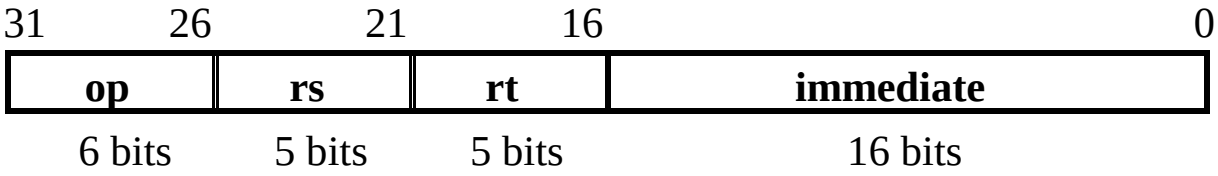
- ADD e SUB
  - addU rd, rs, rt
  - subU rd, rs, rt



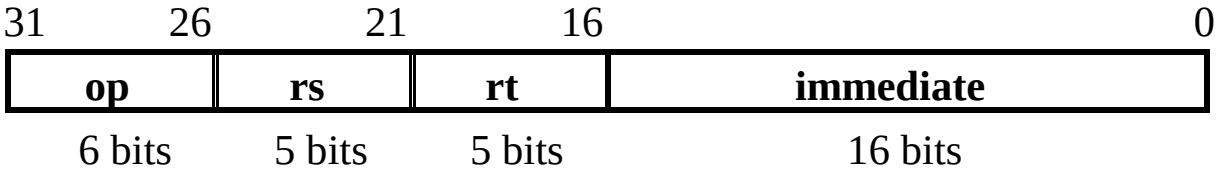
- OR Immediato:
  - ori rt, rs, imm16



- LOAD e STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16



- BRANCH:
  - beq rs, rt, imm16



Trasferimenti tra Registri

- Forniscono il “significato” delle istruzioni
- Tutto comincia con il “fetch”

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

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

inst Register Transfers

ADDU R[rd] ← R[rs] + R[rt]; PC ← PC + 4

SUBU R[rd] ← R[rs] -- R[rt]; PC ← PC + 4

ORi R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4

LOAD R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4

STORE MEM[R[rs] + sign_ext(Imm16)] ← R[rt]; PC ← PC + 4

BEQ if (R[rs] == R[rt]) then PC ← PC + 4 + sign_ext(Imm16)] || 00
else PC ← PC + 4

Passo 2: Componenti del Datapath

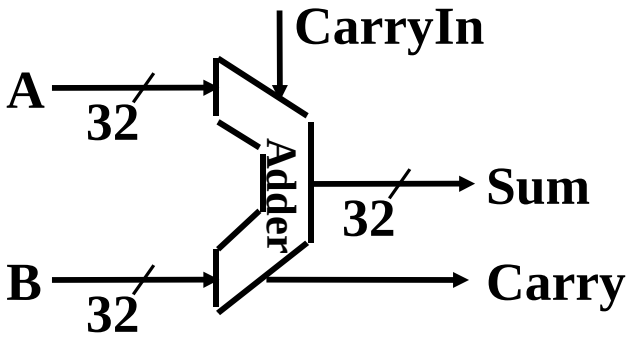
- Memoria
  - instruzioni & dati
- Registri (32 x 32)
  - read RS
  - read RT
  - Write RT o RD
- PC
- Extender (estensione in segno)
- Addiz e Sottraz di registri o di immediati (estesi)
- Somma 4 o immediati estesi al PC

Passo 2: Componenti del Datapath

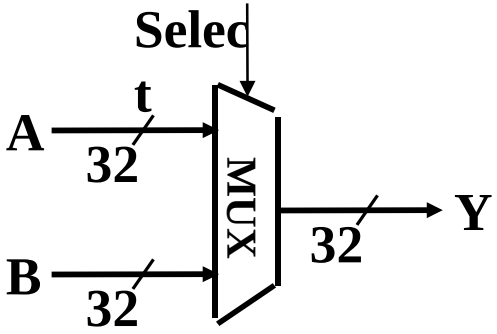
- Elementi Combinatori
- Elementi di Memoria
  - tempificazione (Clocking methodology)

Elementi Combinatori

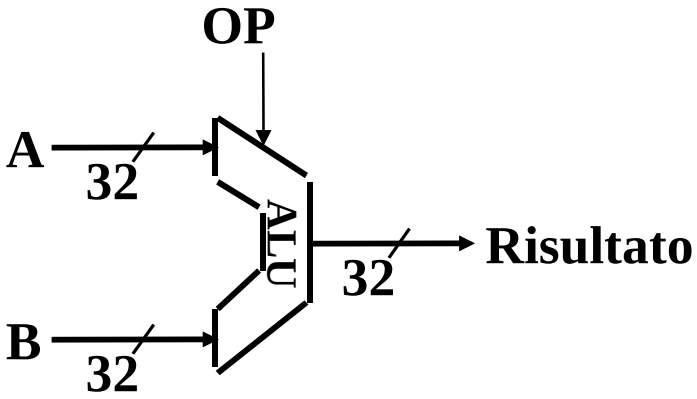
- Adder



- MUX

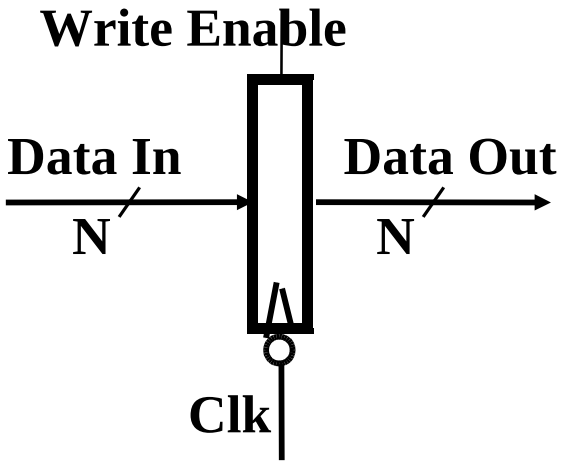


- ALU



Elementi di Memoria: Registro

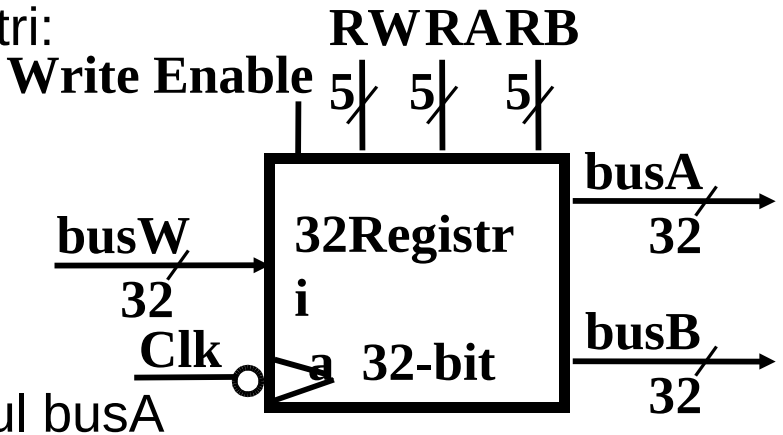
- Registro
  - –Ingressi
    - N-bit
    - Write Enable
  - –Write Enable:
    - negato (0): Data Out non cambia
    - asserito (1): Data Out diventa Data In



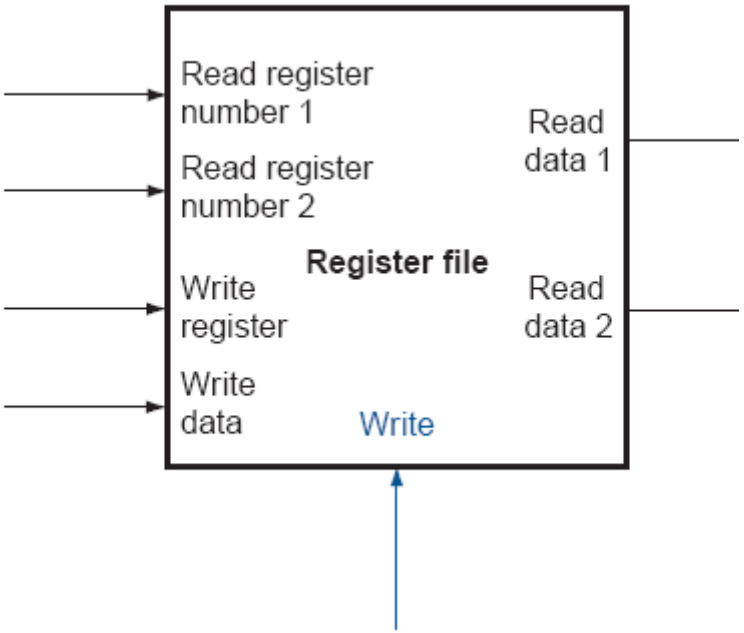
Elementi di Memoria: Banco di Registri

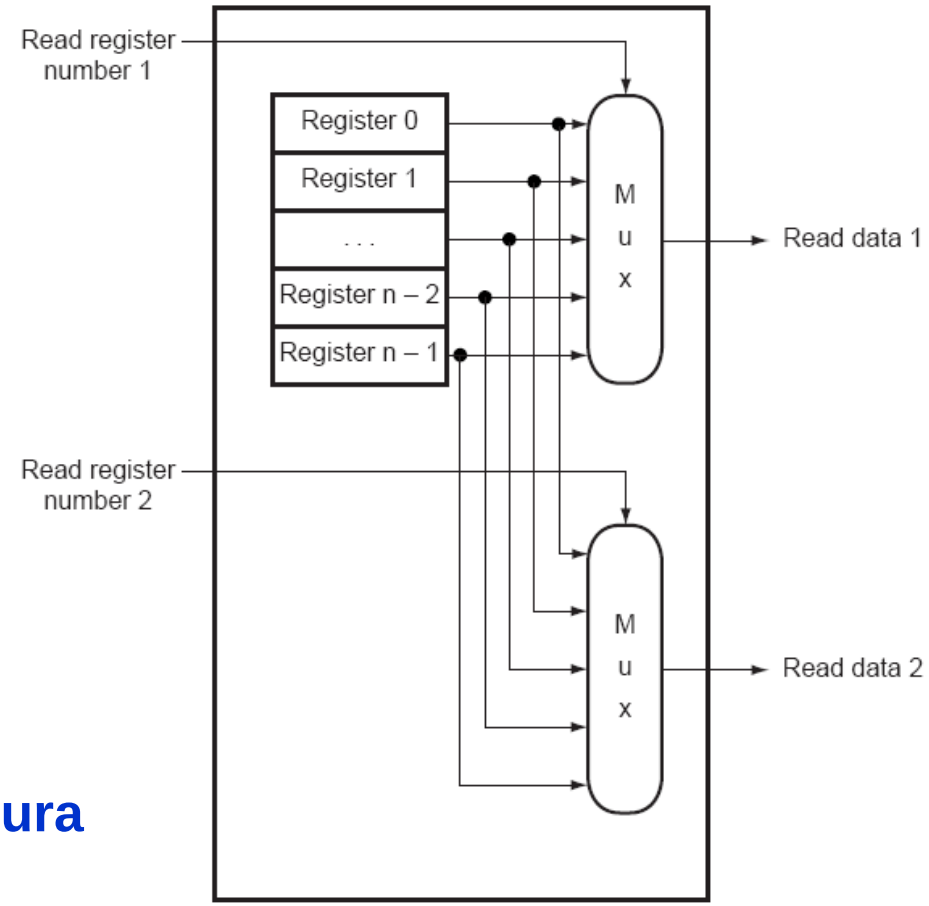
- Il Banco di Registri consiste di 32 registri:
  - Due bus a 32-bit di output:

    busA e busB
  - Un bus di input a 32-bit : busW
- Un Registro viene selezionato da:
  - RA seleziona il registro da mettere sul busA
  - RB seleziona il registro da mettere sul busB
  - RW seleziona il registro da scrivere mediante il busW quando Write Enable è 1
- Il Clock (CLK)
  - Il CLK è significativo SOLO durante le operazioni di write
  - Durante le operazioni di read, il Banco si comporta come se fosse combinatorio:
    - RA o RB validi => busA o busB validi dopo un "tempo di accesso" (access time)



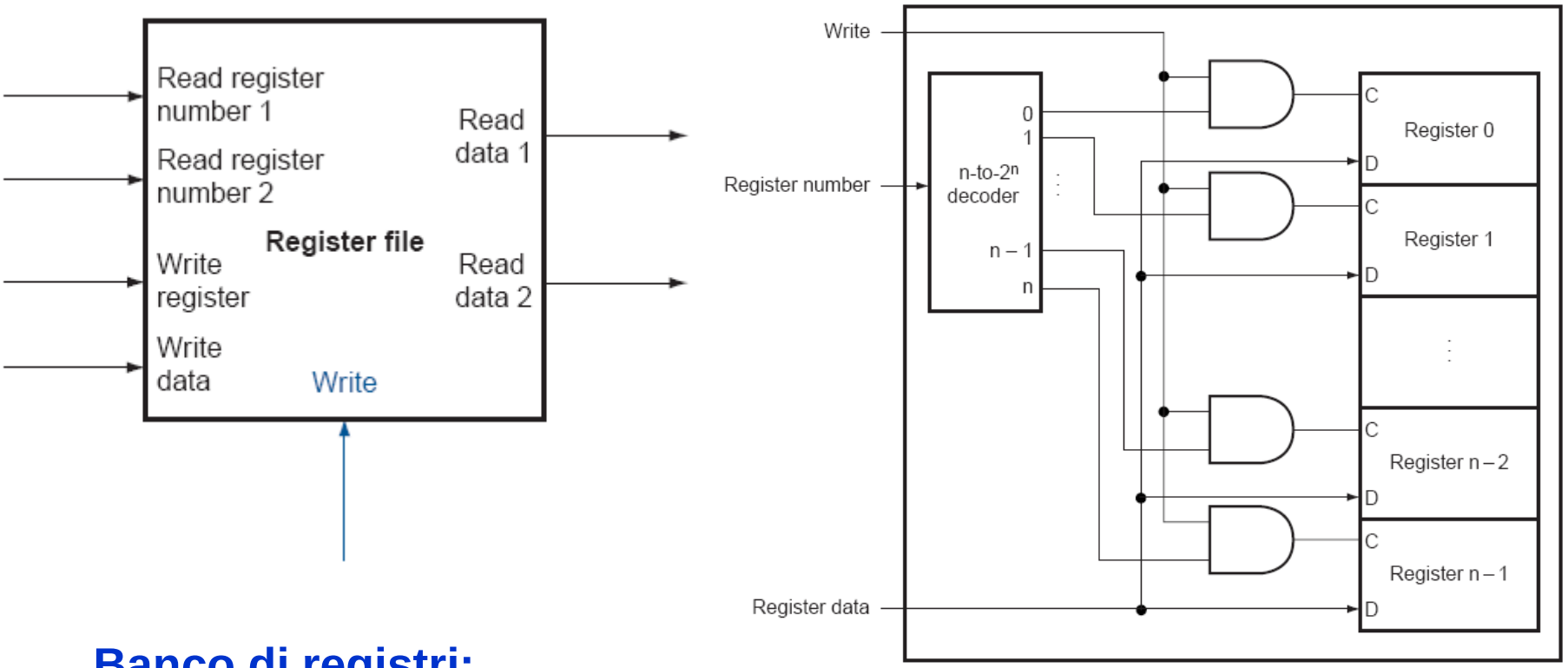
Elementi di Memoria: Banco di Registri





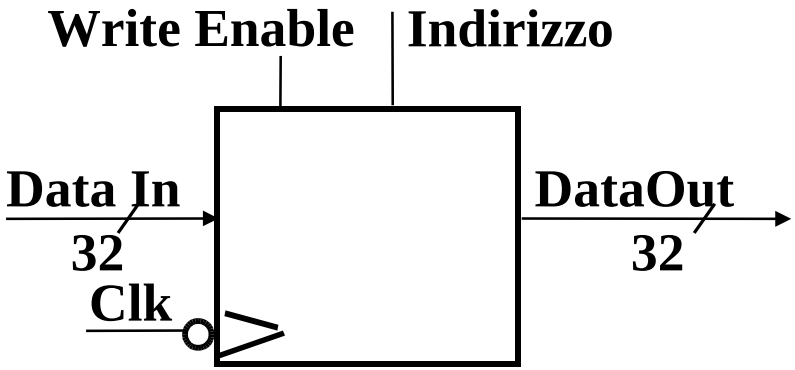
Banco di registri:
schema di accesso in lettura

Elementi di Memoria: Banco di Registri



Banco di registri:
schema di accesso in scrittura

Un modello ideale di Memoria



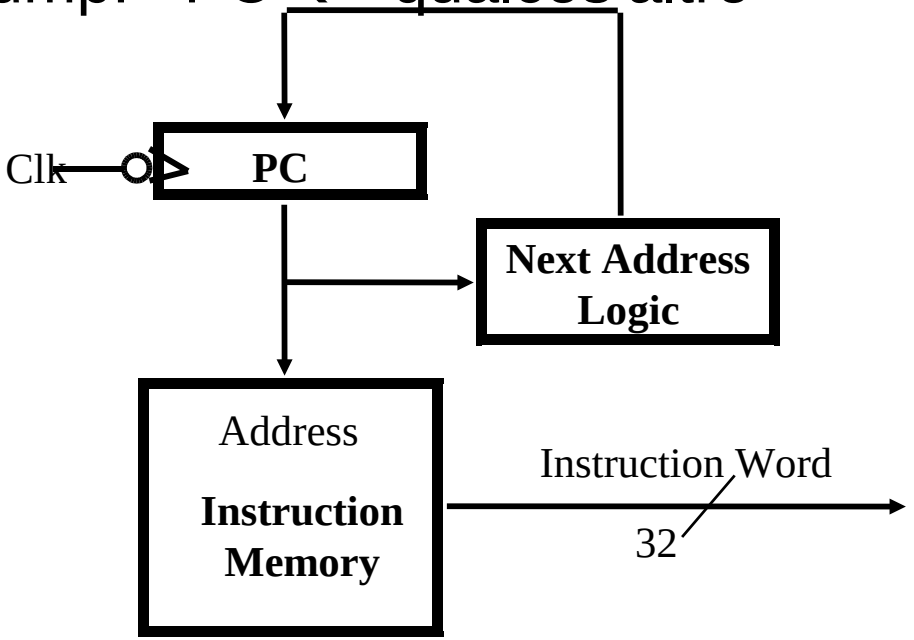
- Memoria (ideale)
  - Un bus di input: Data In
  - Un bus di output: Data Out
- Accesso a una parola di Memoria :
  - Indirizzo seleziona la parola da inviare su Data Out
  - Se Write Enable = 1: indirizzo seleziona la parola di memoria da scrivere mediante il bus Data In
- Clock input (CLK)
  - Il CLK è significativo SOLO durante le operazioni di write
  - Durante le operazioni di read, la Memoria si comporta come se fosse combinatoria:
    - Indirizzo valido => Data Out valido dopo un “tempo di accesso”

Passo 3

- Specifiche dei Trasferimenti tra Registri
  → Assemblaggio del Datapath
- Instruction Fetch
- Leggi gli Operandi ed Esegui l'Operazione

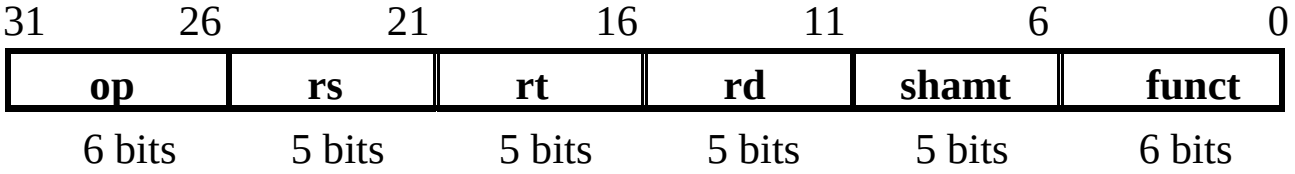
3a: Unità di Fetch

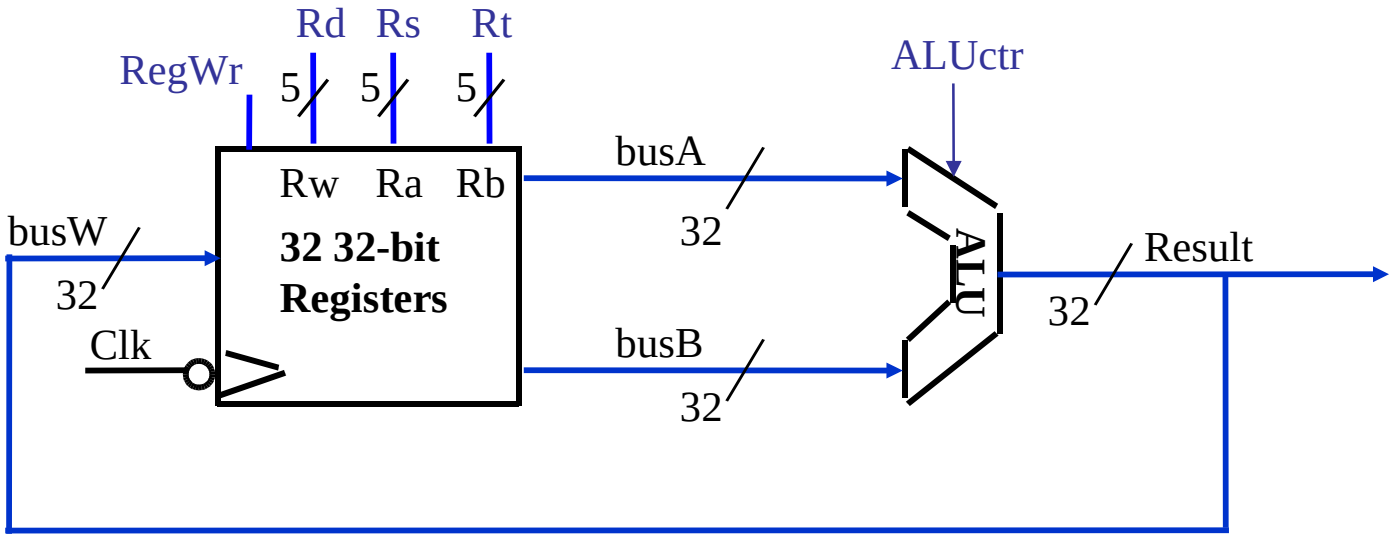
- Operazioni previste
  - Fetch dell'Istruzione: mem[PC]
  - Aggiornamento del program counter:
    - Codice Sequenziale: PC ← PC + 4
    - Branch e Jump: PC ← "qualcos'altro"



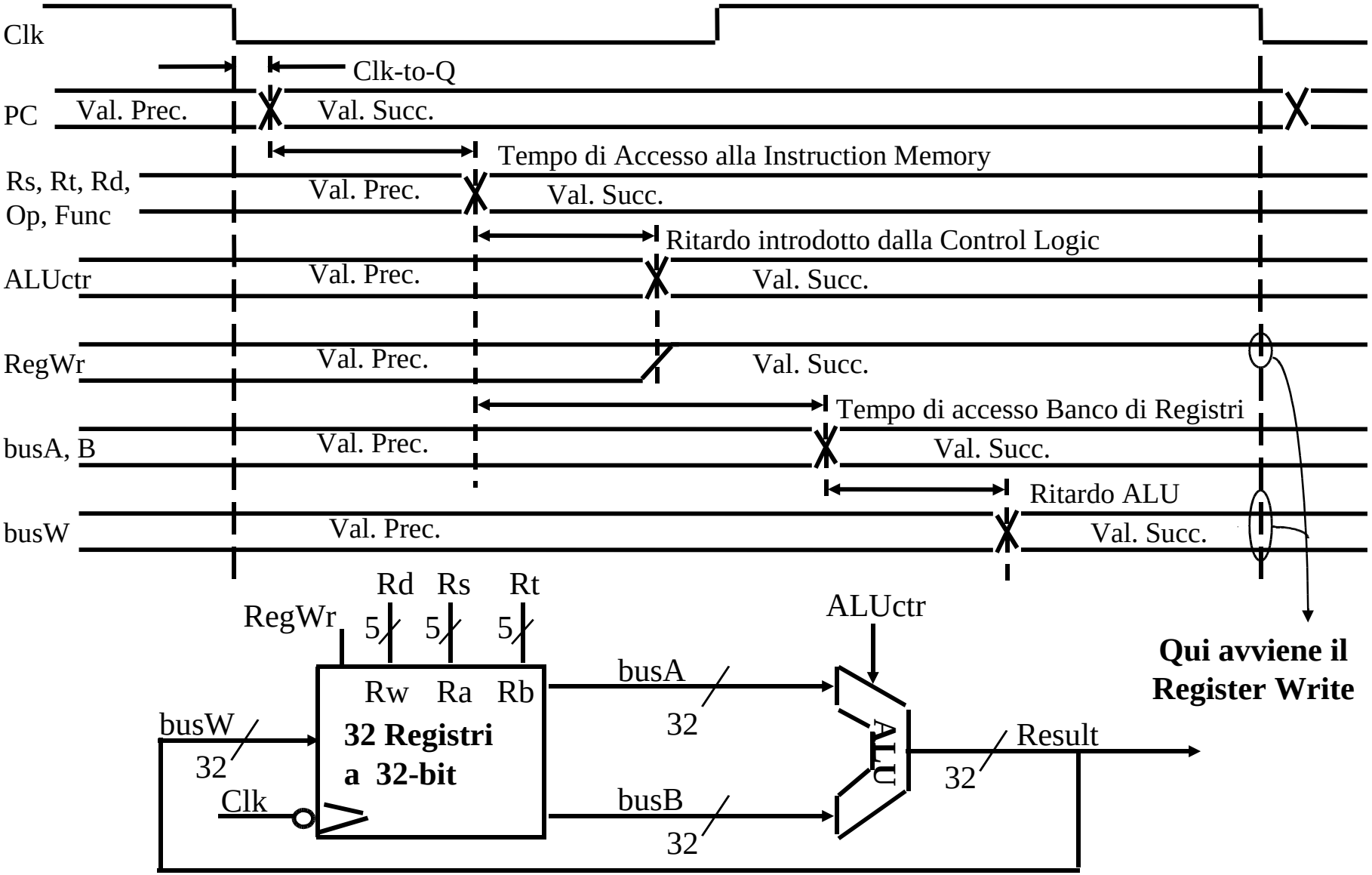
3b: Add & Subtract

- R[rd] ← R[rs] op R[rt] Esempio: addU rd, rs, rt
  - Ra, Rb e Rw dai campi rs, rt e rd dell'istruzione
  - ALUctr e RegWr: dalla control logic dopo la decodifica dell' istruzione



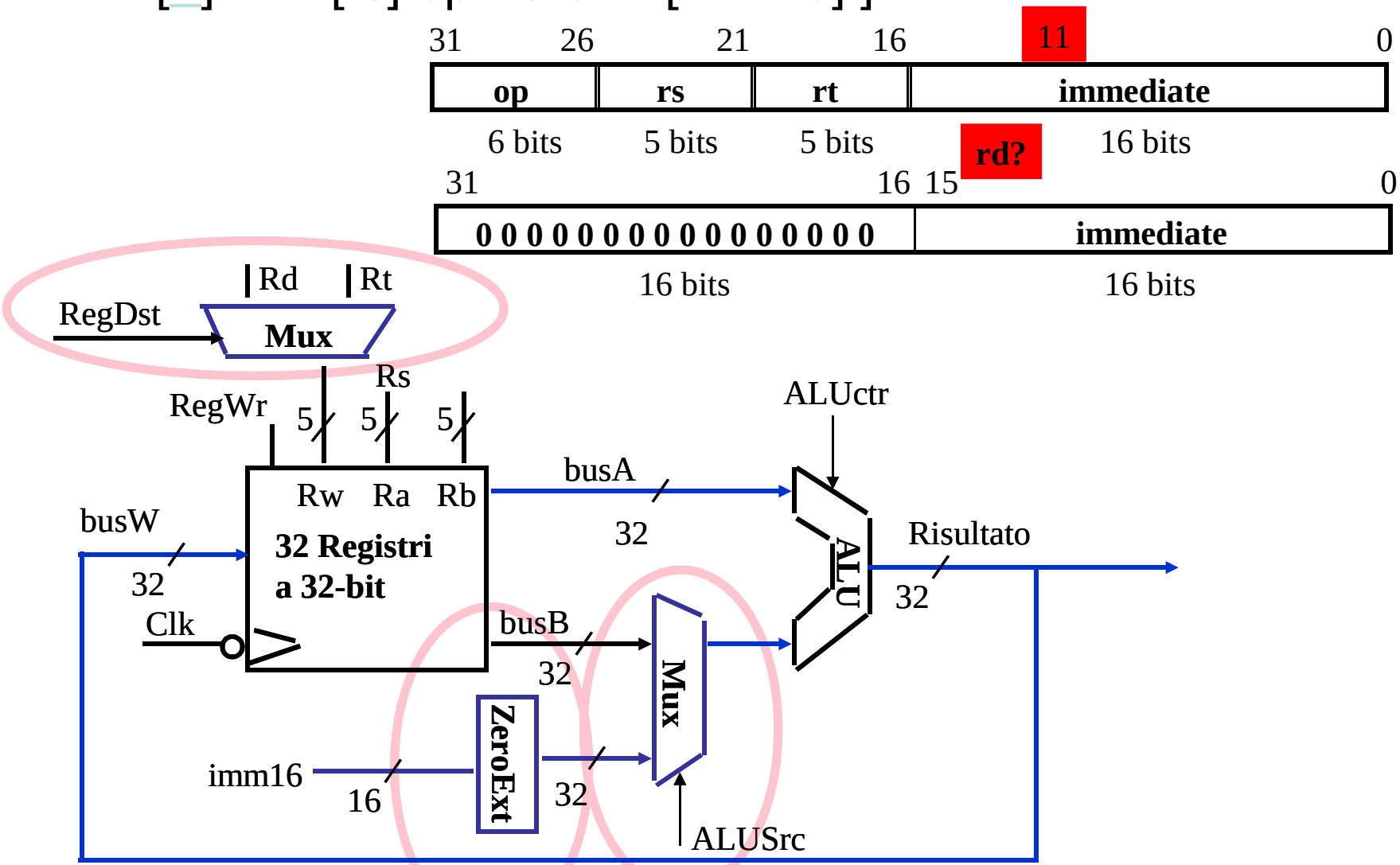


Register-Register Timing



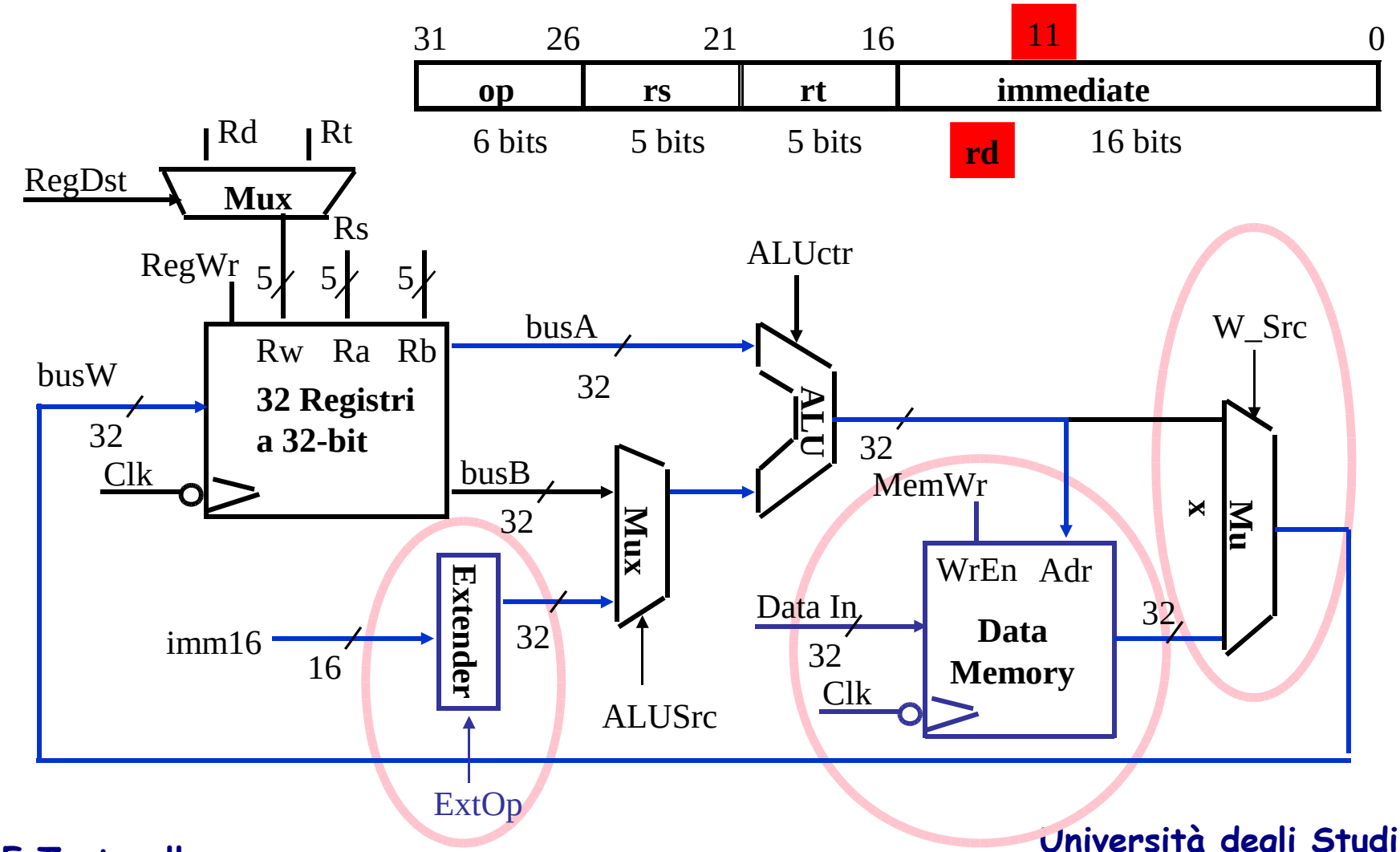
3c: Operazioni Logiche con Immediati

- R[rt] ← R[rs] op ZeroExt[imm16]]



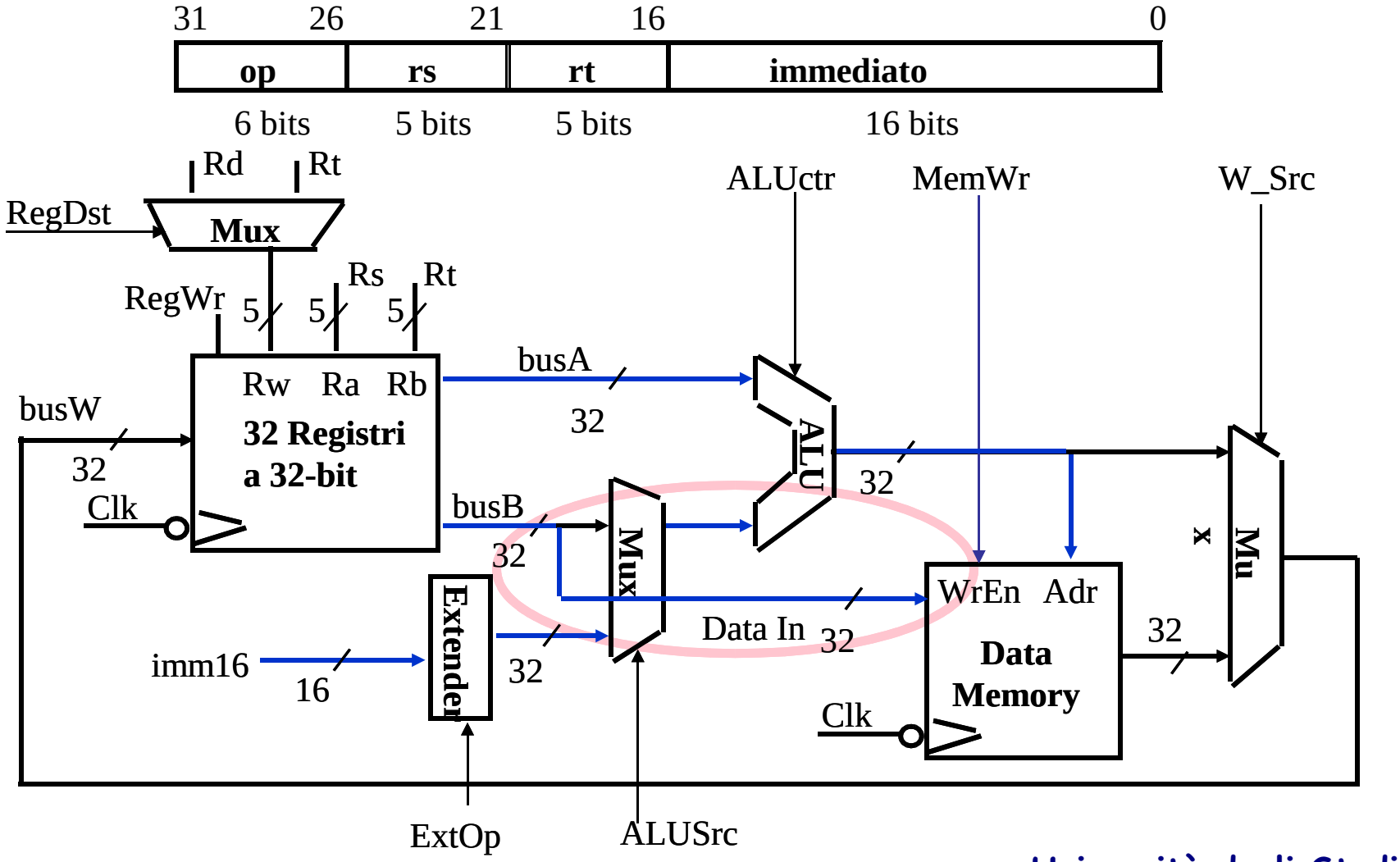
3d: Operazioni di Load

- R[rt] ← Mem[R[rs] + SignExt[imm16]] Esempio: lw rt, imm16(rs)

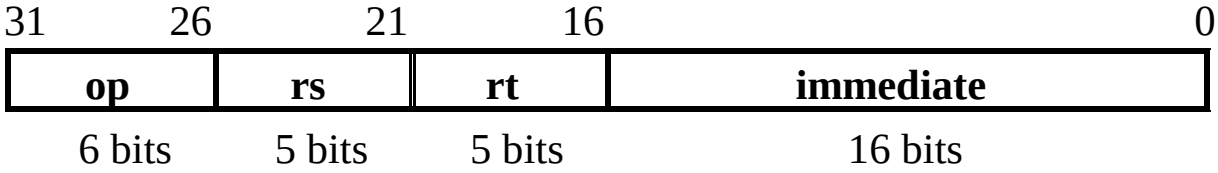


3e: Operazioni di Store

- Mem[R[rs] + SignExt[imm16] ← R[rt]] Esempio: sw rt, imm16(rs)



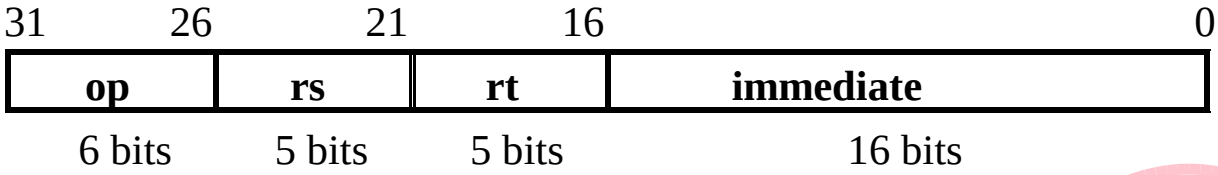
3f: L'Istruzione di Branch

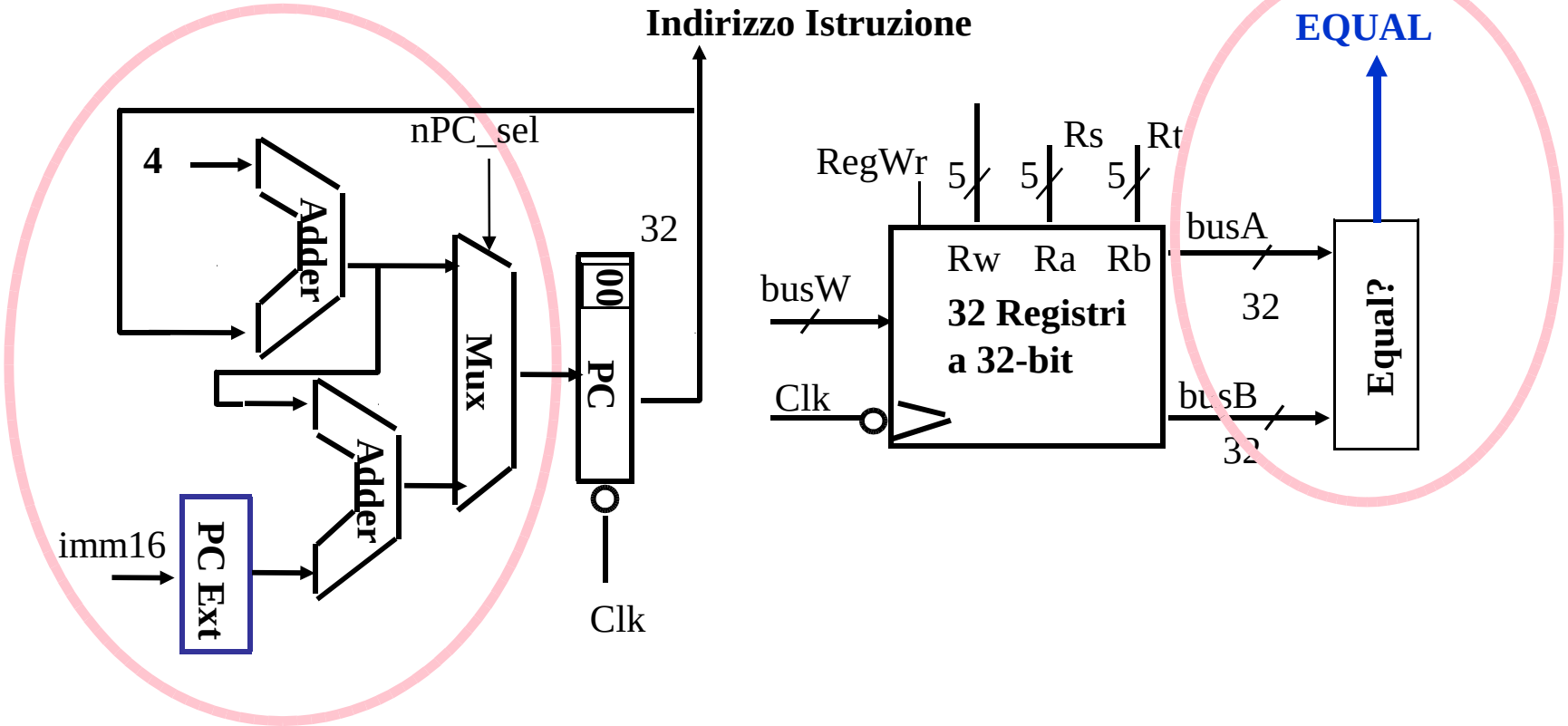


- beq rs, rt, imm16
  - mem[PC] Fetch dell'istruzione dalla memoria
  - EQUAL ← R[rs] == R[rt] Calcolo della condizione di branch
  - if (EQUAL) Calcolo indirizzo prossima istruzione
    - PC ← PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC ← PC + 4

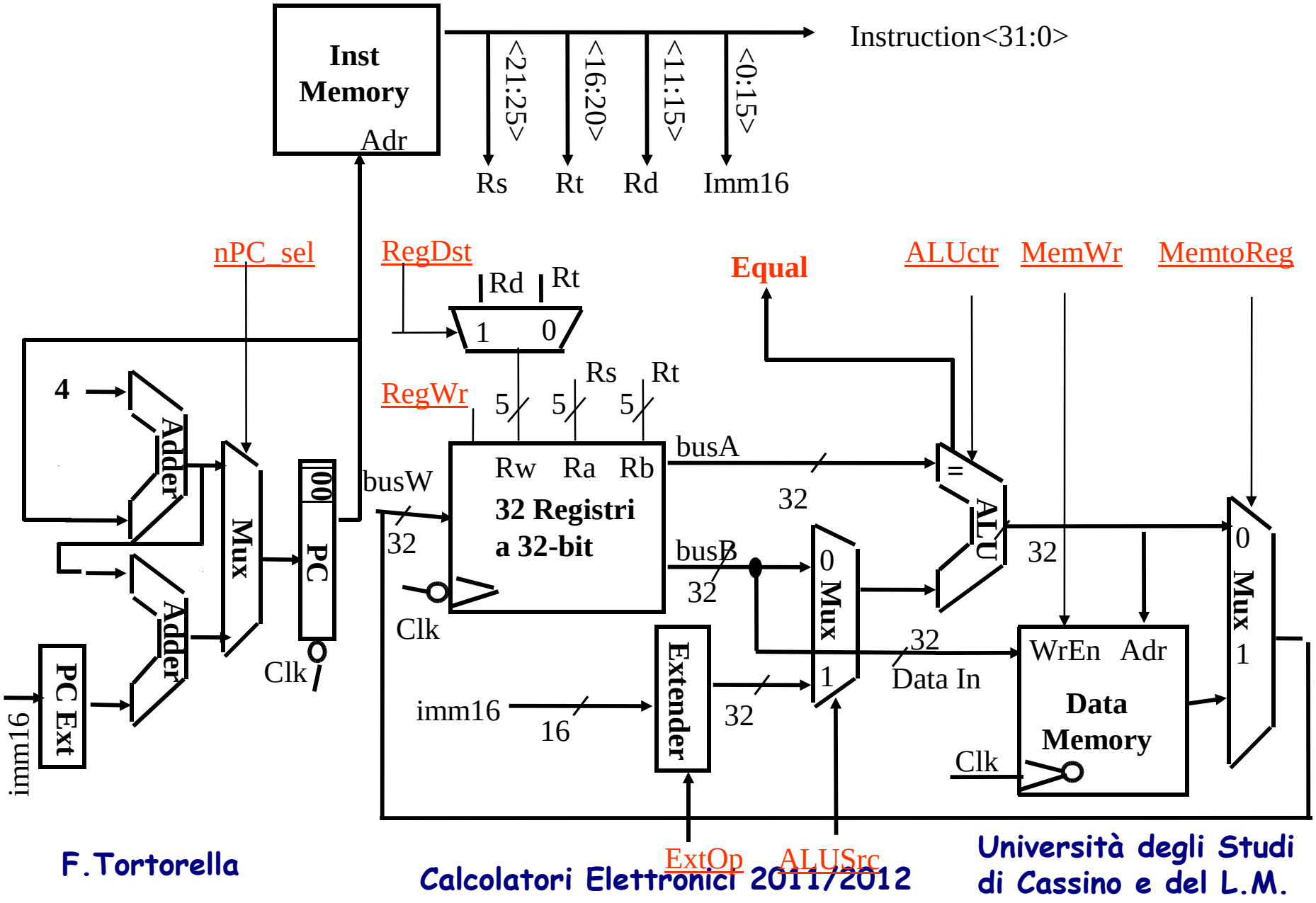
Datapath per Operazioni di Branch

- beq rs, rt, imm16

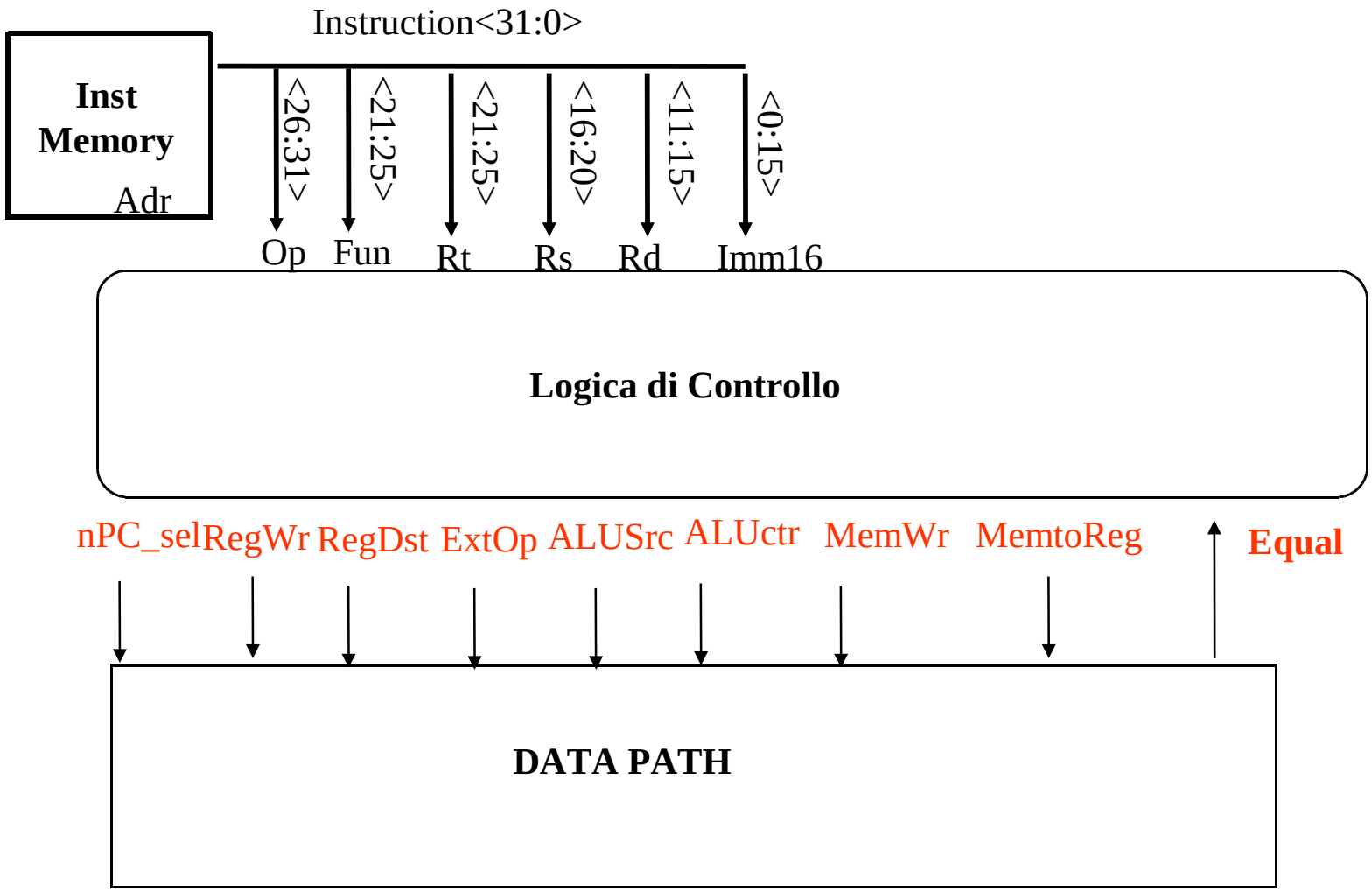




Single Cycle Datapath

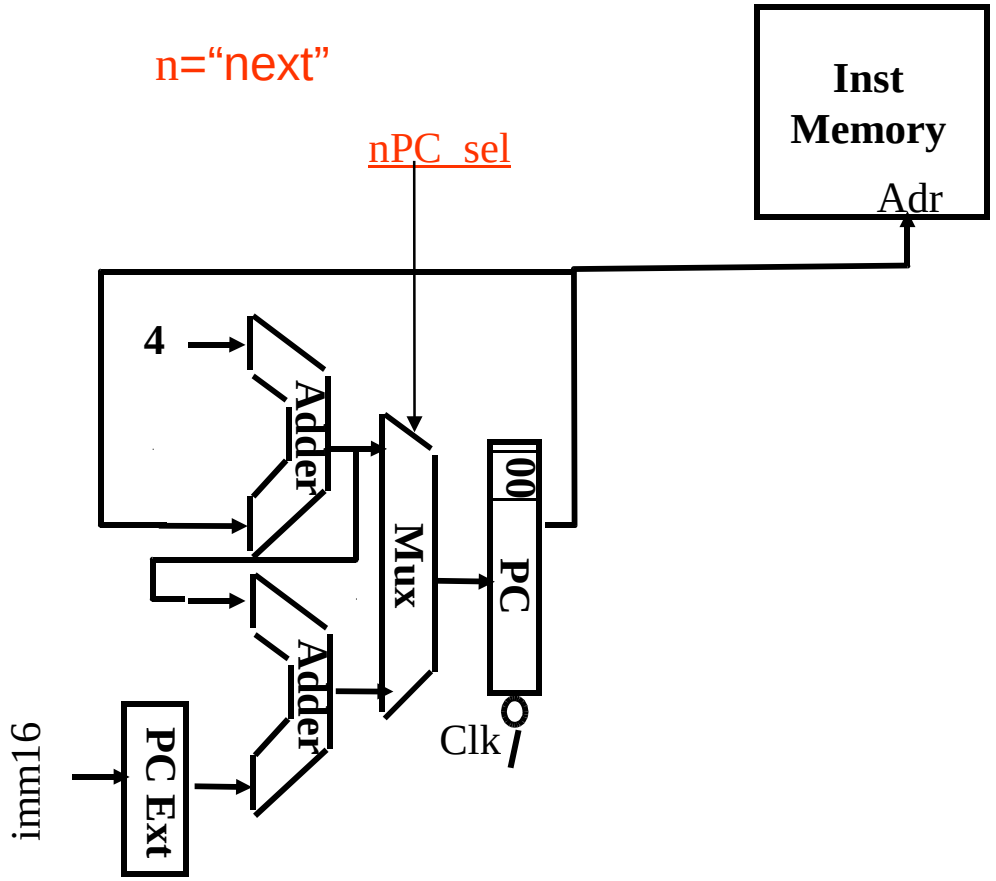


Passo 4: Realizzazione del Controllo



Significato dei Segnali di Controllo

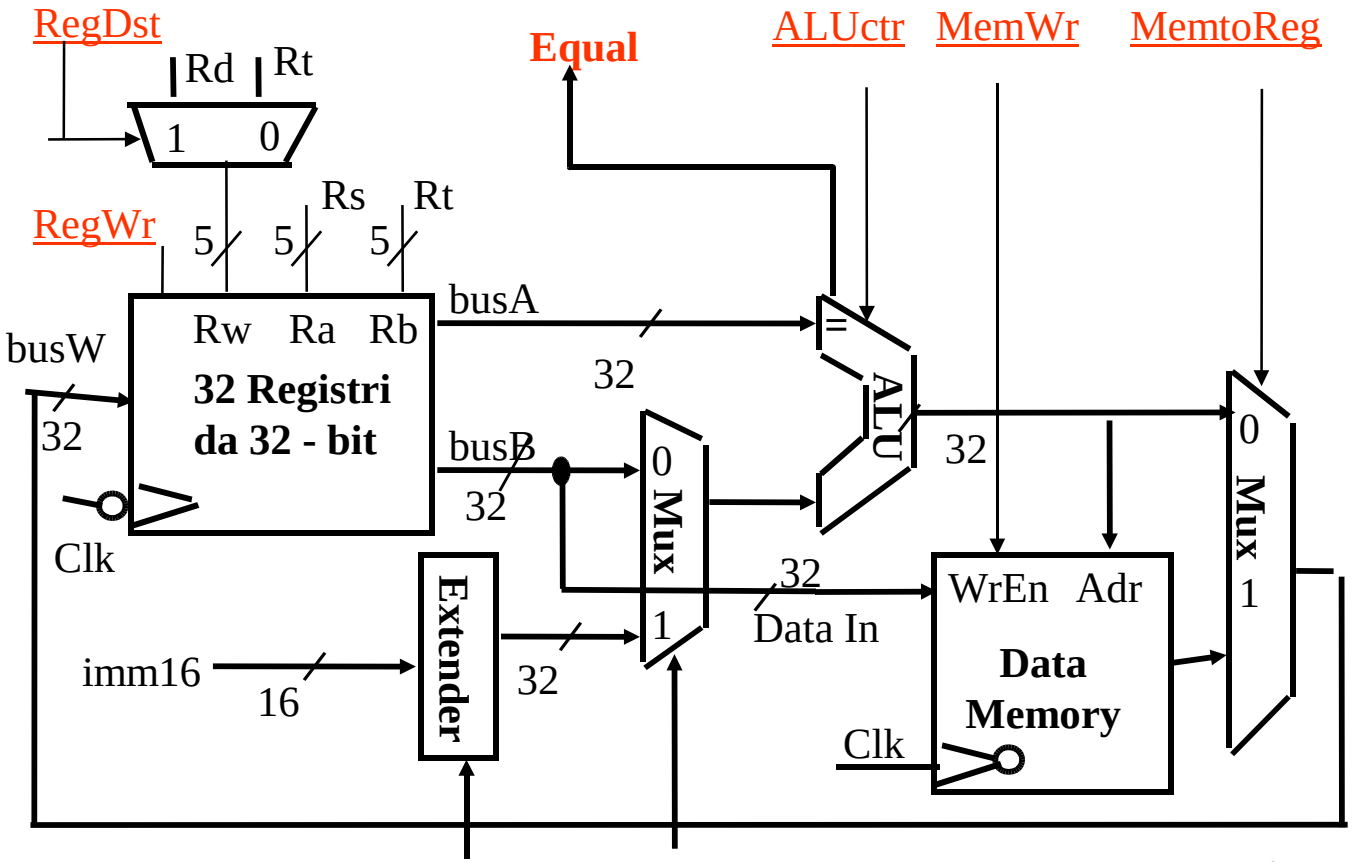
- Rs, Rt, Rd e Imm16 sono cablati nel datapath
- nPC_sel: 0 => PC ← PC + 4; 1 => PC ← PC + 4 + SignExt(Im16) || 00



Significato dei Segnali di Controllo

- ExtOp: “zero”, “sign”
- ALUsrc: 0 => regB;
  1 => immed
- ALUctr: “add”, “sub”, “or”

- **MemWr: write memory**
- **MemtoReg: 0=>ALU; 1 => Mem**
- **RegDst: 0 => “rt”; 1 => “rd”**
- **RegWr: write dest register**



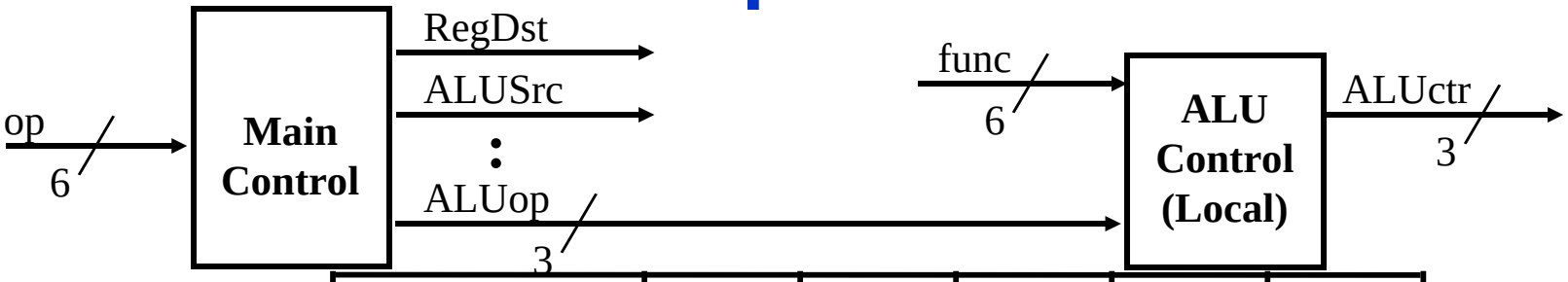
Control Signals

inst	Register Transfer	
ADD	R[rd] <-- R[rs] + R[rt];	PC <-- PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”	
SUB	R[rd] <-- R[rs] -- R[rt];	PC <-- PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <-- R[rs] + zero_ext(Imm16);	PC <-- PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”	
LOAD	R[rt] <-- MEM[R[rs] + sign_ext(Imm16)];	PC <-- PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr,	nPC_sel = “+4”
STORE	MEM[R[rs] + sign_ext(Imm16)] <-- R[rs];	PC <-- PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <-- PC + sign_ext(Imm16)] || 00 else PC <-- PC + 4	
	nPC_sel = EQUAL, ALUctr = “sub”	

Passo 5:
Logica per ciascun segnale di controllo

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <=if (OP == “000000”) then “regB” else “immed”
- ALUctr <= if (OP == “000000”) then **funct**
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- ExtOp <= if (OP == ORi) then “zero” else “sign”
- MemWr <= (OP == Store)
- MemtoReg<= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

La “tabella di verità” per il controllo

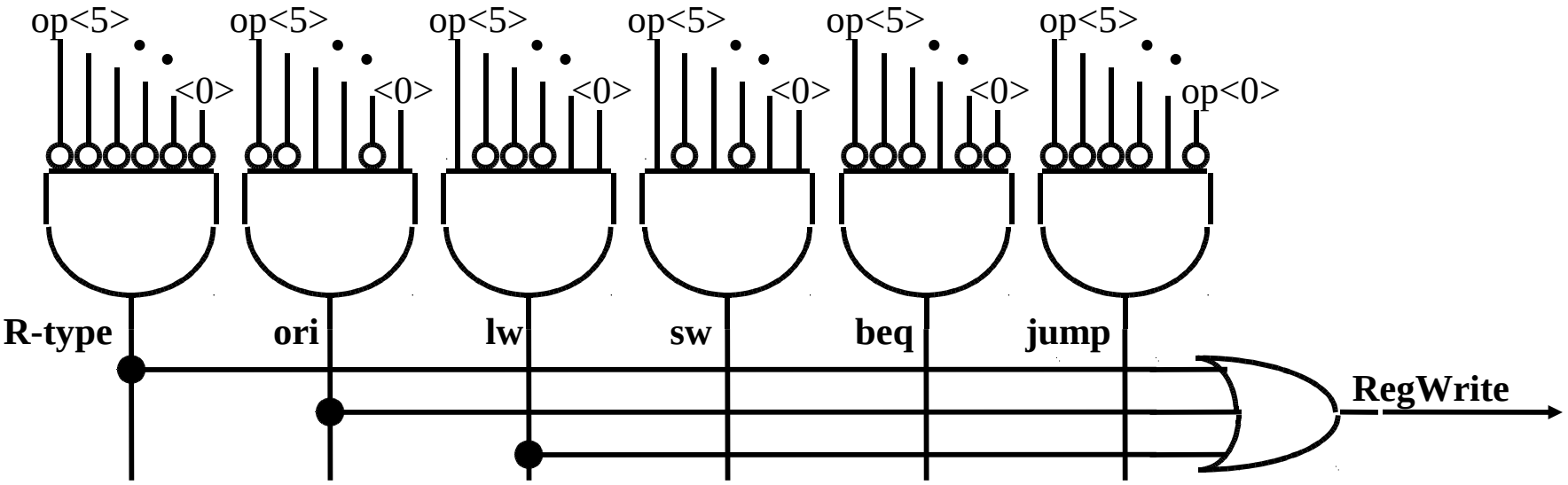


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
PCSrc	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

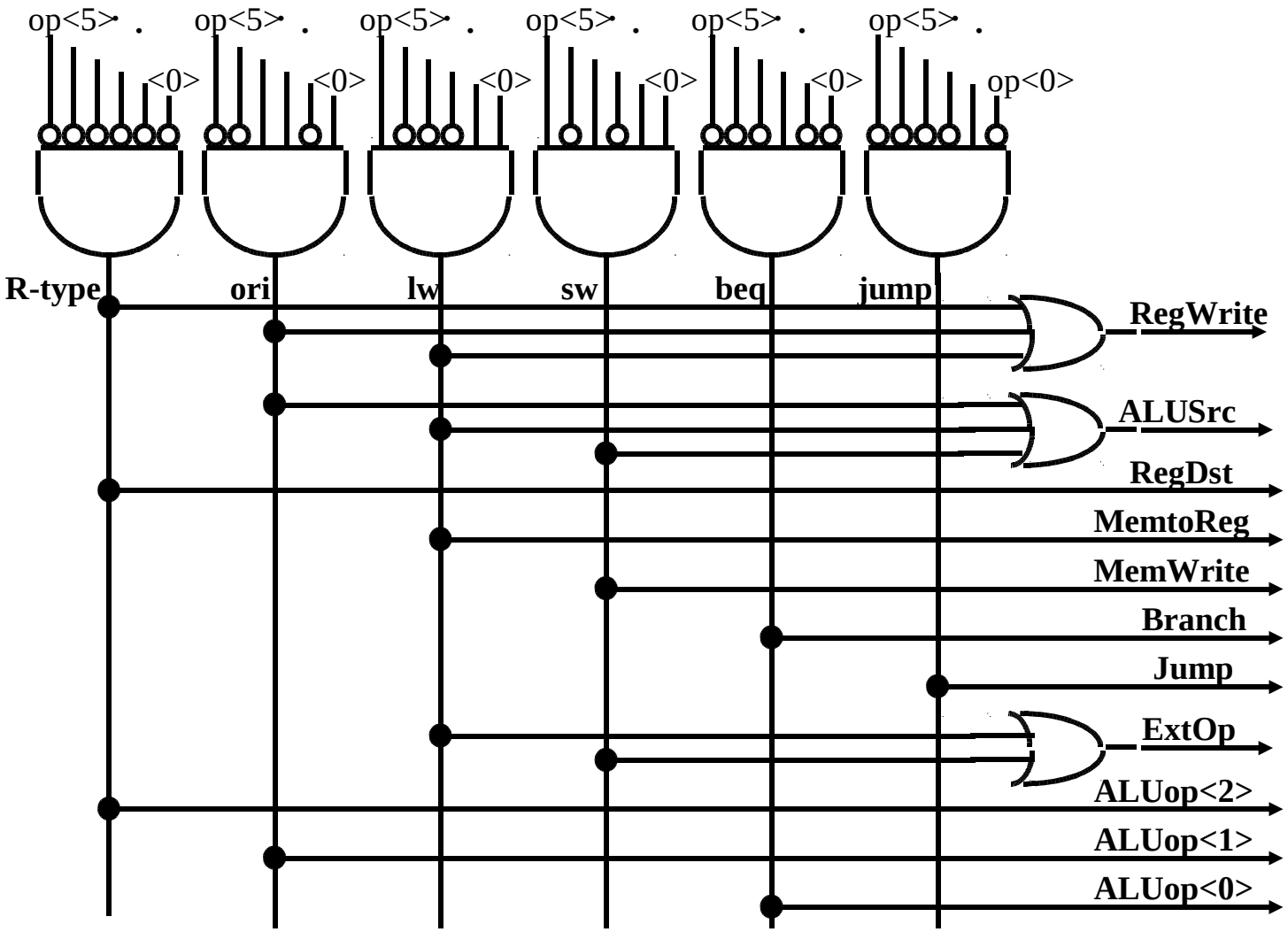
La “tabella di verità” per RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

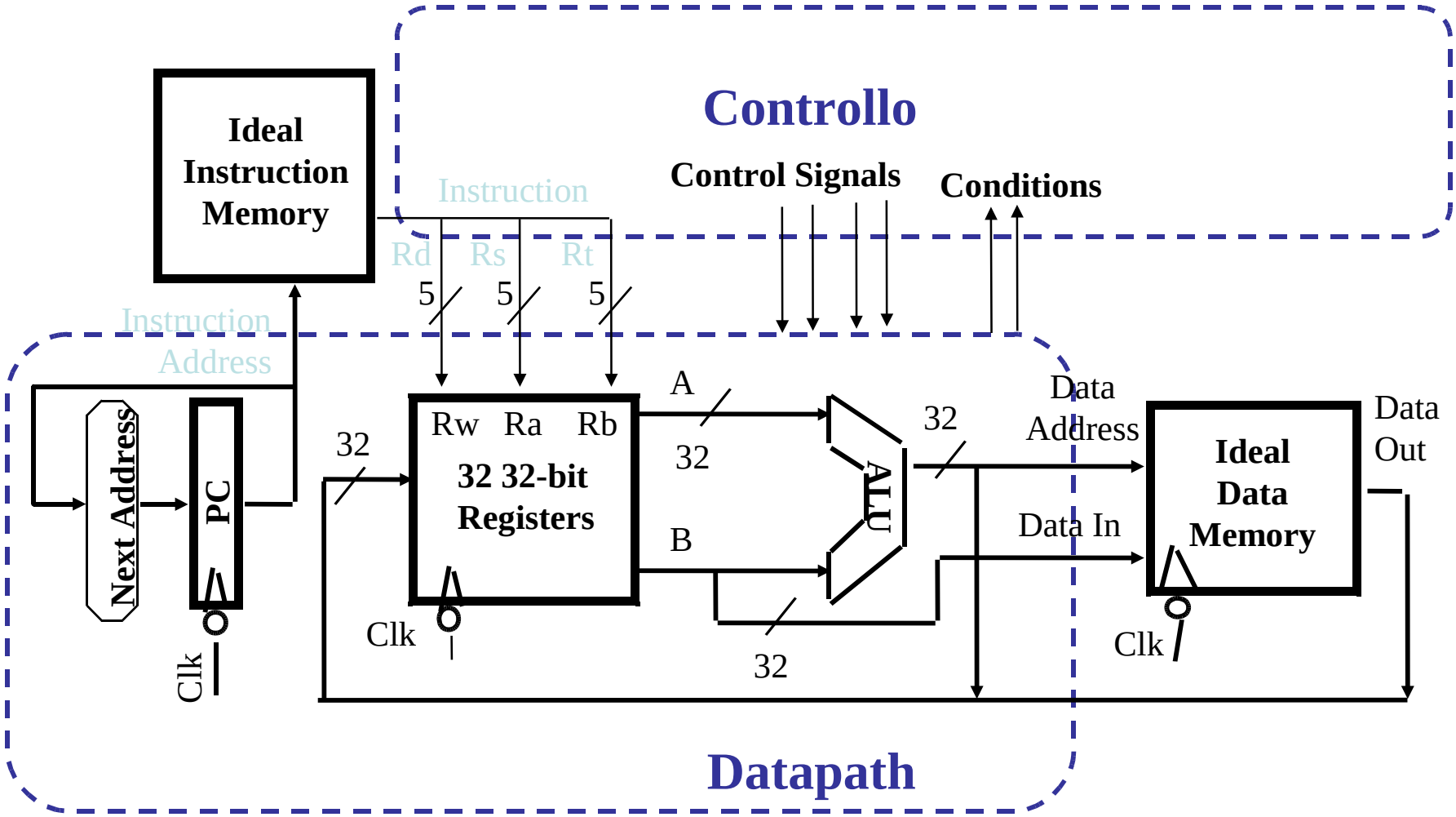
- RegWrite = R-type + ori + lw
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



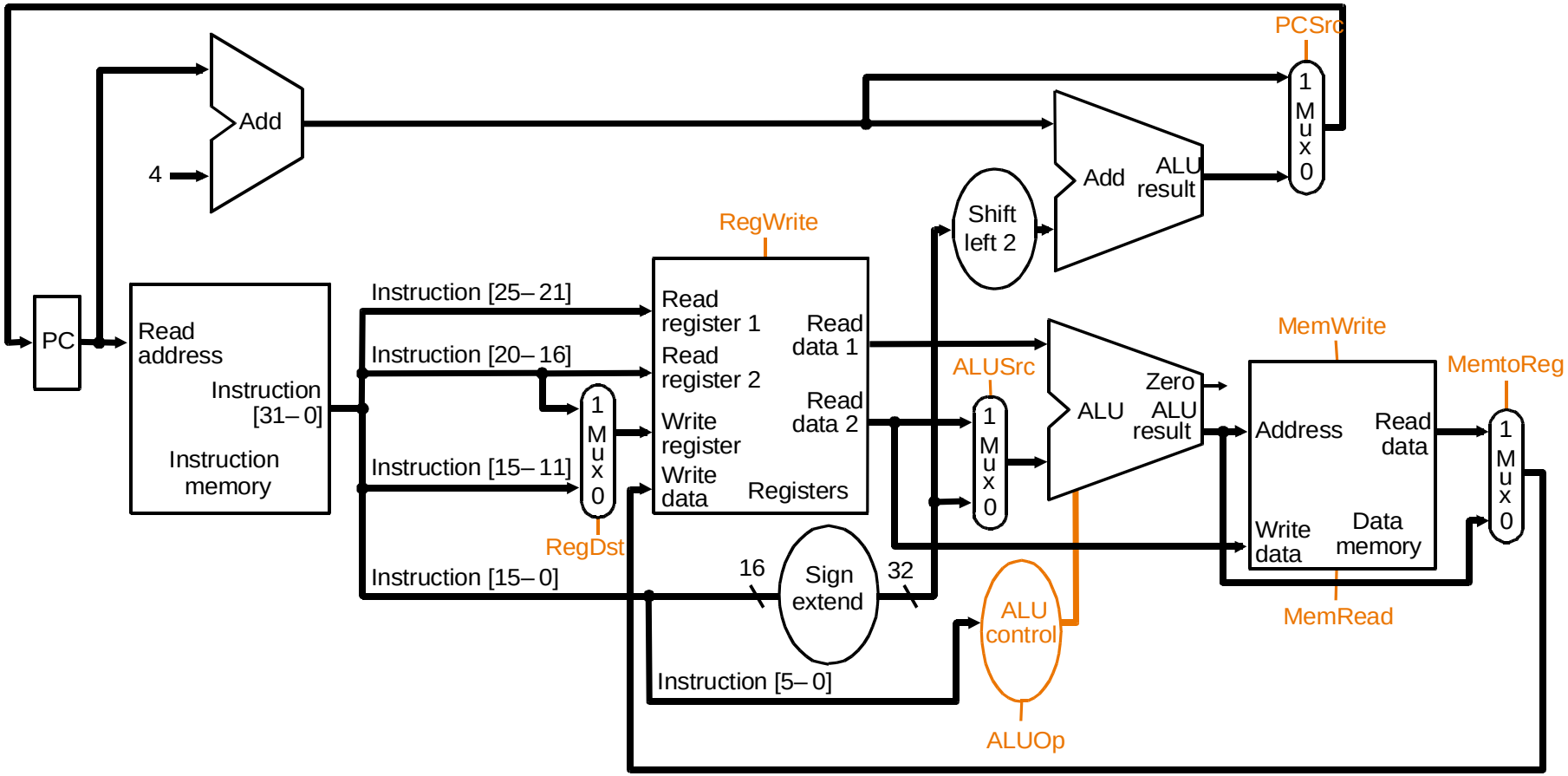
Implementazione PLA del controllo



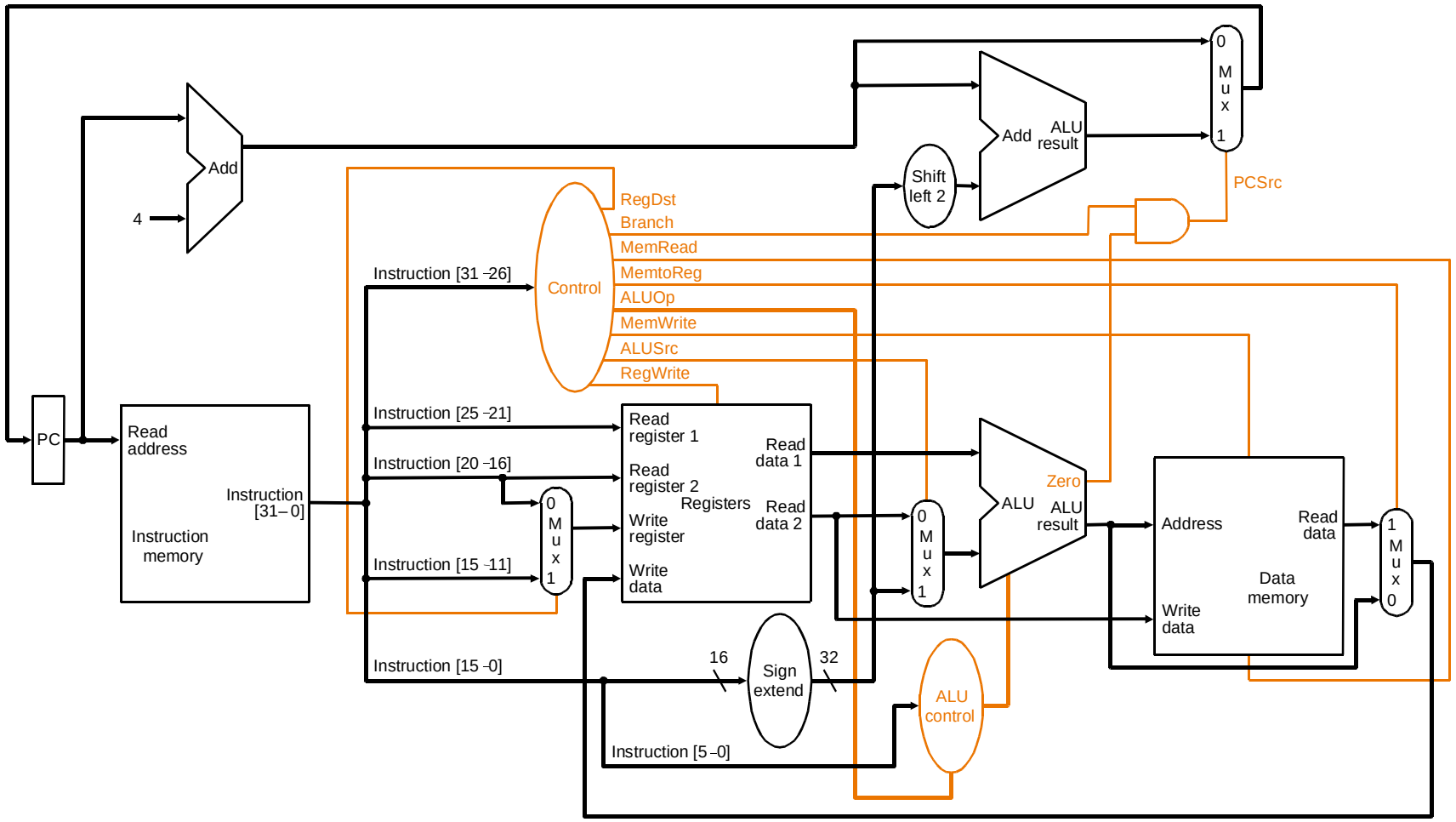
Vista Astratta dell'Implementazione



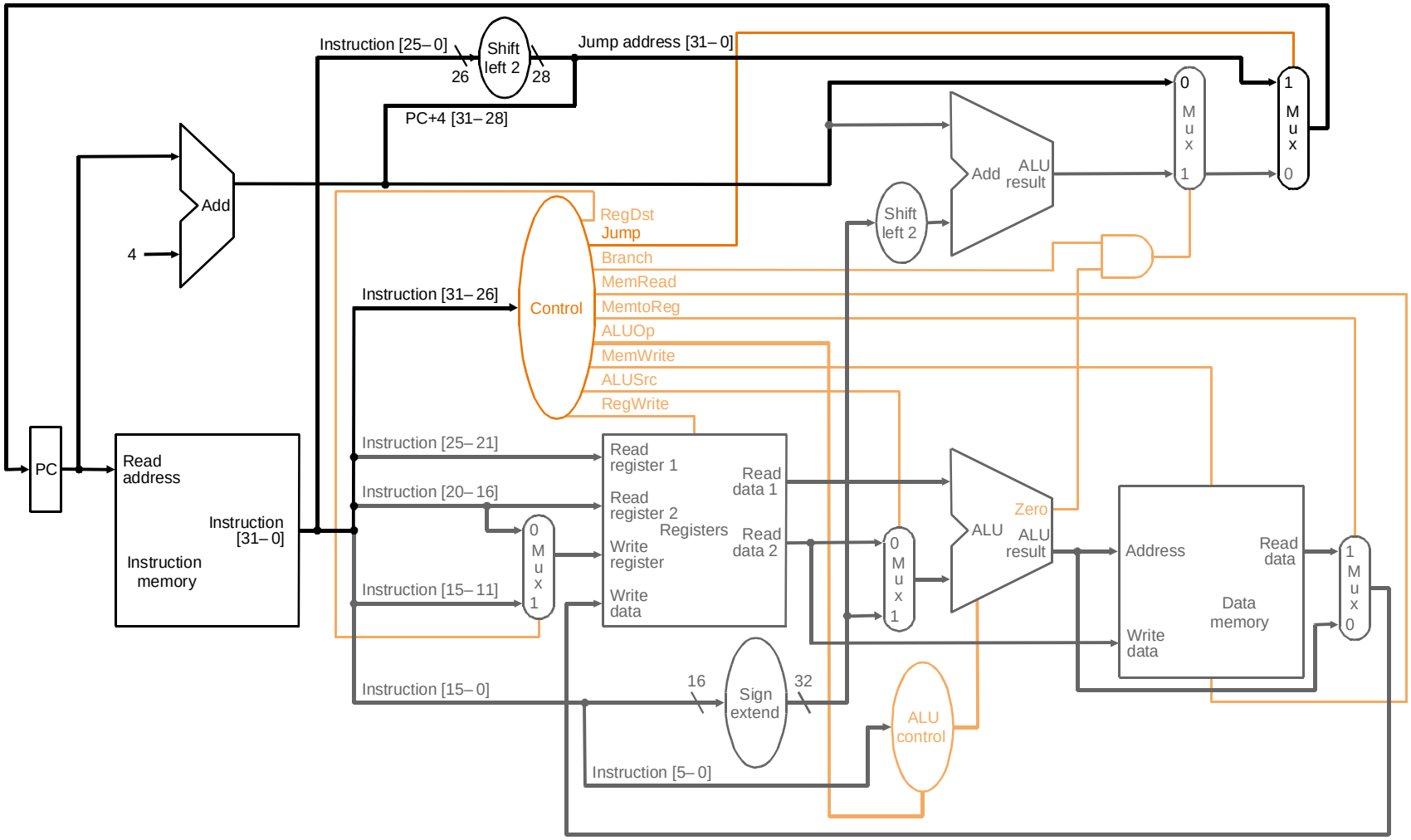
Schema del datapath con le linee di controllo



Schema del datapath con l'unità di controllo



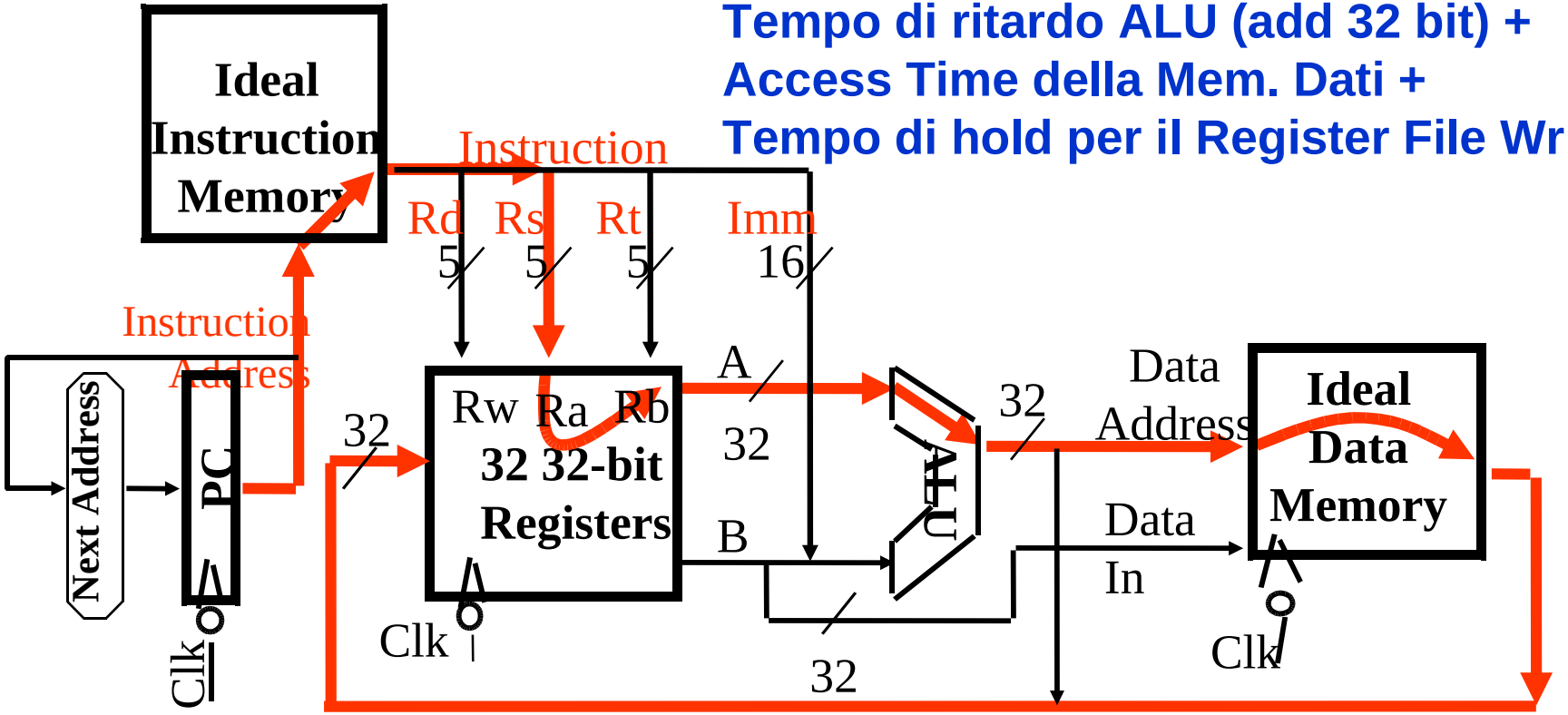
Modifica per implementare jump



Vista astratta del critical path

Definisce la max frequenza di clock

Critical Path (istr. LOAD) =
- **Ritardo tra Clk e out PC (FFs) +**
- **Access Time della Mem. Istruzioni +**
- **Access Time del banco registri+**
- **Tempo di ritardo ALU (add 32 bit) +**
- **Access Time della Mem. Dati +**
- **Tempo di hold per il Register File Write**



Riassumendo

- 5 passi per progettare un processore
  - 1. Analizzare l'instruction set => Specifiche sul datapath
  - 2. Selezionare l'insieme di componenti del datapath stabilire una metodologia di clocking
  - 3. Costruire il datapath rispettando le specifiche
  - 4. Analizzare l'implementazione di ciascuna istruzione per determinare i punti di controllo che abiliteranno i trasferimenti
  - 5. Costruire la logica di controllo

- il MIPS rende più semplice il progetto
  - Le istruzioni tutte della stessa dimensione
  - I registri sorgente sempre nello stesso posto dell'istruzione
  - Gli Immediati stessa dimensione, posizione
  - Le Operazioni sempre su registri/immediati

Che cosa abbiamo combinato ?

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g.,MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

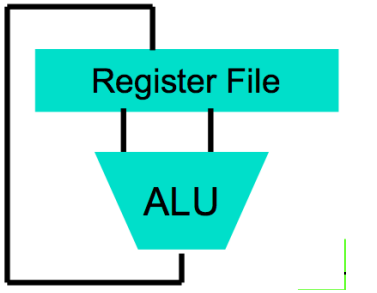
Architecture Implementation

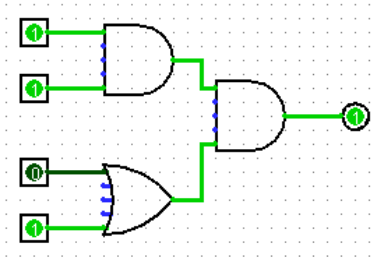
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





Valutazione del data path a ciclo singolo

- L'implementazione realizzata prevede implicitamente che occorra un solo ciclo di clock per eseguire una qualunque istruzione.
- Sebbene tale implementazione funzioni correttamente, non è utilizzata in pratica perché inefficiente (qual è il motivo ?)
- Consideriamo un esempio numerico

Valutazione del data path a ciclo singolo

- Assumiamo che i tempi di ritardo di tutti i componenti nel datapath siano trascurabili tranne:
  - Unità di memoria: 200 ps
  - ALU e addizionatori: 100 ps
  - Register file (read e write): 50 ps
- Confrontiamo le prestazioni di due diverse implementazioni:
  - Implementazione a ciclo singolo di lunghezza fissa
  - Implementazione a ciclo singolo di lunghezza variabile (soluzione solo teorica)

Valutazione del data path a ciclo singolo

- Supponiamo la seguente distribuzione per le istruzioni:
  - Load: 25 %
  - Store: 10 %
  - Istruzioni logico-aritmetiche: 45 %
  - Branch: 15 %
  - Jump: 5 %

Valutazione del data path a ciclo singolo

Tipo di istruzione	Unità funzionali usate				
Logico aritmetiche	Instr. fetch	Accesso ai registri	ALU	Accesso ai registri	
Load	Instr. fetch	Accesso ai registri	ALU	Accesso alla memoria	Accesso ai registri
Store	Instr. fetch	Accesso ai registri	ALU	Accesso alla memoria	
Branch	Instr. fetch	Accesso ai registri	ALU		
Jump	Instr. fetch				

Valutazione del data path a ciclo singolo

Tipo di istruzione	Instruction memory	Register Read	ALU operation	Data memory	Register write	Totale (ps)
Logico aritmetiche	200	50	100	0	50	400
Load	200	50	100	200	50	600
Store	200	50	100	200	0	550
Branch	200	50	100	0	0	350
Jump	200	0	0	0	0	200

Valutazione del data path a ciclo singolo

- Ciclo di lunghezza fissa: necessario allinearsi all'istruzione che richiede maggior tempo di esecuzione. $T_f = 600$ ps
- Ciclo di lunghezza variabile: occorre valutare il tempo medio di esecuzione. $T_v = 447.5$ ps
  - $T_v = 600*.25+550*.10+400*.45+350*.15+200*.05$
- Consideriamo il rapporto

$$T_f/T_v = 1.34$$

Valutazione del data path a ciclo singolo

- Problemi dell'implementazione del data path a ciclo singolo di lunghezza fissa:
  - Condizionato dal worst case (istruzione più lenta)
  - Ogni unità funzionale è usata una volta sola per ciclo (necessario replicare alcune funzionalità, es. adder)
  - Situazione anche peggiore se si considera l'introduzione di altre unità necessarie (FPU).
- Soluzioni possibili ?
  - Implementazione multiciclo (OLD): ciclo più breve legato al tempo di ritardo dell'unità funzionale più lenta. Le unità vengono usate per più compiti, ma l'esecuzione di un'istruzione richiede più cicli di clock.
  - Implementazione pipeline: data path simile all'implementazione a ciclo singolo. Permette l'esecuzione contemporanea di più istruzioni incrementando l'utilizzazione dell'hardware e aumentando le prestazioni.