Calcolatori Elettronici B
a.a. 2004/2005

Tecniche di Controllo:
Esercizi

Massimiliano Giacomin

Calcolo CPI e Prestazioni nei sistemi a singolo ciclo e multiciclo

1) Calcolo prestazioni nei sistemi a singolo ciclo

$CPI = 1$

$T_{clock} = \max\{T_a + \dots T_k\}$

ovvero la serie più lenta di "operazioni atomiche" [cammino critico]

$T_{esecuzione} = \#istruzioni * CPI * T_{clock} = \#istruzioni * T_{clock}$

2) Calcolo CPI e prestazioni nei sistemi multi-ciclo

Dato un certo carico di lavoro con frequenze relative delle istruzioni $f_1, \dots, f_n$

$CPI = f_1*CPI_1 + f_2*CPI_2 + \dots + f_n*CPI_n$

$T_{clock} = \max\{T_1, \dots, T_m\}$ [operazioni atomiche eseguite in una fase]

$T_{esecuzione} = \#istruzioni * CPI * T_{clock}$

3) Confronto di prestazioni tra sistemi diversi [su un carico/prog. determinato]

$$\frac{T^1_{esecuzione}}{T^2_{esecuzione}} = \frac{CPI^1 * T^1_{clock}}{CPI^2 * T^2_{clock}}$$

Esercizio [NB: cfr. Patterson pag. 323 e seguenti, pag. 342 e seguenti + esercizio di confronto macchine M1, M2 e M3 in CALC_A]

Si ipotizzi un **carico di lavoro** che preveda:

lw:	31%		Eseguite secondo le fasi viste
sw:	21%		
Tipo-R:	27%		
beq:	5%		
j:	7%		Come Tipo-R ma ALU richiede
somma virgola mob:		7%	4 operaz. in sequenza anziché 1
moltipl. Virgola mob:		7%	Come Tipo-R ma 8 operaz. ALU

Si supponga che le **operazioni atomiche** che coinvolgono le unità funzionali principali (di cui si tiene conto per il calcolo dei tempi) richiedano:

Unità di memoria (lettura e scrittura):	2 ns
Register File (lettura e scrittura):	1 ns
Operazione-ALU:	2 ns

➡ Confrontare le prestazioni di una implementazione a ciclo singolo vs. multi-ciclo (vincolo di non mettere in serie due operazioni atomiche) vs. multi-ciclo in cui lettura in memoria/ALU e scrittura Reg.File nello stesso ciclo.

Soluzione

Ricordando le fasi delle istruzioni si hanno le seguenti operazioni atomiche:

Istruzione	Fetch	Lettura Registri/ Decode	ALU	Accesso Memoria	Scrittura Registri
lw	1	1	1	1	1
sw	1	1	1	1	
Tipo-R	1	1	1		1
beq	1	1	1		
j	1	1			“1”
ADD V. MOB.	1	1	4		1
MUL V. MOB.	1	1	8		1

M1 (Macchina a singolo ciclo): tutte le operazioni in serie nello stesso ciclo di clock

Istruzione	Fetch	Lettura Registri/ Decode	ALU	Accesso Memoria	Scrittura Registri	Tempo di esecuz.
lw	1	1	1	1	1	8
sw	1	1	1	1		7
Tipo-R	1	1	1		1	6
beq	1	1	1			5
j	1	~~1~~			~~"1"~~	2
ADD V. MOB.	1	1	4		1	12
MUL V. MOB.	1	1	8		1	20
Tempo per operaz. atomica	2	1	2	2	1	

Operazione più lunga determina T_{clock} = 20 ns

⇒ CPI = 1

T_{clock} = 20 ns

T_{M1} = I * 20 ns (I è il numero di istruzioni)

M2 (Macchina multi-ciclo): in un ciclo di clock NO serie di operazioni atomiche

T_{clock} = 2ns (operazione ALU oppure accesso in memoria)

CPI = somma pesata dei CPI singole istruzioni con le frequenze relative
= (vedi lucido seguente) 5.09

⇒ T_{M2} = I * 5.09 * 2 ns = I * 10.18 ns

M2 (Macchina multiciclo): CPI per istruzione

Istruzione	Fetch	Lettura Registri/ Decode	ALU	Accesso Memoria	Scrittura Registri	CPI
lw	1	1	1	1	1	5
sw	1	1	1	1		4
Tipo-R	1	1	1		1	4
beq	1	1	1			3
j	1	1			“1”	3
ADD V. MOB.	1	1	4		1	7
MUL V. MOB.	1	1	8		1	11

CPI = 0.31*5 + 0.21*4 + 0.27*4 + 0.05*3 + 0.07*3 + 0.07*7 +0.07*11
= 5.09

M3: in un ciclo di clock ho al più:

lettura memoria | ALU [2 ns] + scrittura Reg.File [1 ns]

⇒ T_{clock} = 3ns

CPI = per TIPO-R, lw e virgola-mobile CPI si riduce di 1

Istruzione	Fetch	Lettura Registri/ Decode	ALU [ev. Scr. Reg.]	Accesso Memoria [ev. Scr. Reg.]	CPI
lw	1	1	1	1	4
sw	1	1	1	1	4
Tipo-R	1	1	1		3
beq	1	1	1		3
j	1	1	"1"		3
ADD V. MOB.	1	1	4		6
MUL V. MOB.	1	1	8		10

CPI = 0.31*4 + 0.21*4 + 0.27*3 + 0.05*3 + 0.07*3 + 0.07*6 +0.07*10
= 4.37

⇒ T_{M3} = I * 4.37 * 3 ns = I * 13.11 ns

Riepilogo

	CPI	T_{clock}	T_{esecuz}	$T_{istruzione\text{-}più\text{-}costosa}$
M1	1	20 ns	I * 20 ns	20 ns
M2	5.09	2 ns	I * 10.18 ns	11*2 = 22 ns
M3	4.37	3 ns	I * 13.11 ns	10*3 = 30 ns

Confronto delle prestazioni

$$\frac{T_{M1} = I * 20 \text{ ns}}{T_{M2} = I * 10.18 \text{ ns}}$$

Quasi 2 volte più veloce

$$\frac{T_{M3} = I * 13.11 \text{ ns}}{T_{M2} = I * 10.18 \text{ ns}}$$

Circa 1.3 volte più veloce

NB: la riduzione di CPI non paga rispetto all'aumento di T_{clock}

La macchina multiciclo con divisione "equilibrata" delle fasi è la più efficiente.

Notare però che l'istruzione più costosa (mul in V.MOB.) è più veloce nell'implementazione a singolo ciclo.

Altra Tipologia di Esercizio: implementazione istruzioni nuove

Si consideri il DataPath per l'implementazione multiciclo del MIPS
[che avete a disposizione!]. Si considerino le due ipotetiche istruzioni assembler:

```
j   (rs)          ; PC ← rs
lw rt, (rs)++     ; rt ← M[rs]  / rs=rs+4
```

il cui significato è illustrato a commento.

Per ognuna di queste istruzioni:

- si dica se (e come) deve essere modificato il **datapath** per permetterne l'implementazione rispettando il vincolo usuale su elementi funzionali;
- supponendo che vi sia un codice operativo "libero" (non impegnato in altre istruzioni) si fornisca un possibile **formato** della corrispondente istruzione macchina e si specifichi come deve essere modificato il **diagramma a stati finiti** dell'unità di controllo per implementarla;
- supponendo T_{clock} = 2ns, calcolare i tempi di esecuzione della nuova istruzione.

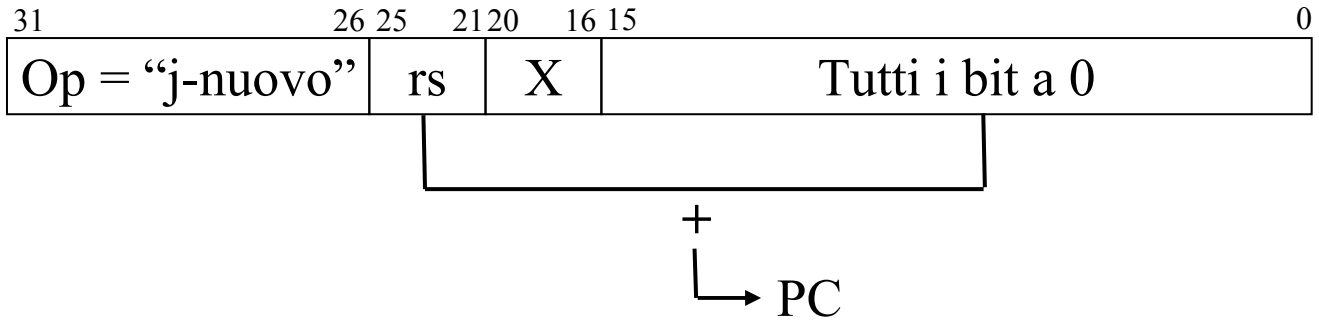
j (rs)

Osservando il datapath, si vede che:

- è possibile forzare la scrittura di PC con l'uscita del MUX [ALUout| ALU| indirizzo j] ponendo PCWrite = 1
- visto che ALUout proviene da ALU, consideriamo ALU!
- in ingresso alla ALU, il primo operando deve essere necessariamente A=rs; per ottenere rs in uscita alla ALU, l'unica possibilità è avere sign-extend(offset) [ev. shiftato di 2 bit] purchè offset=0 e forzare ALU ad effettuare una somma:

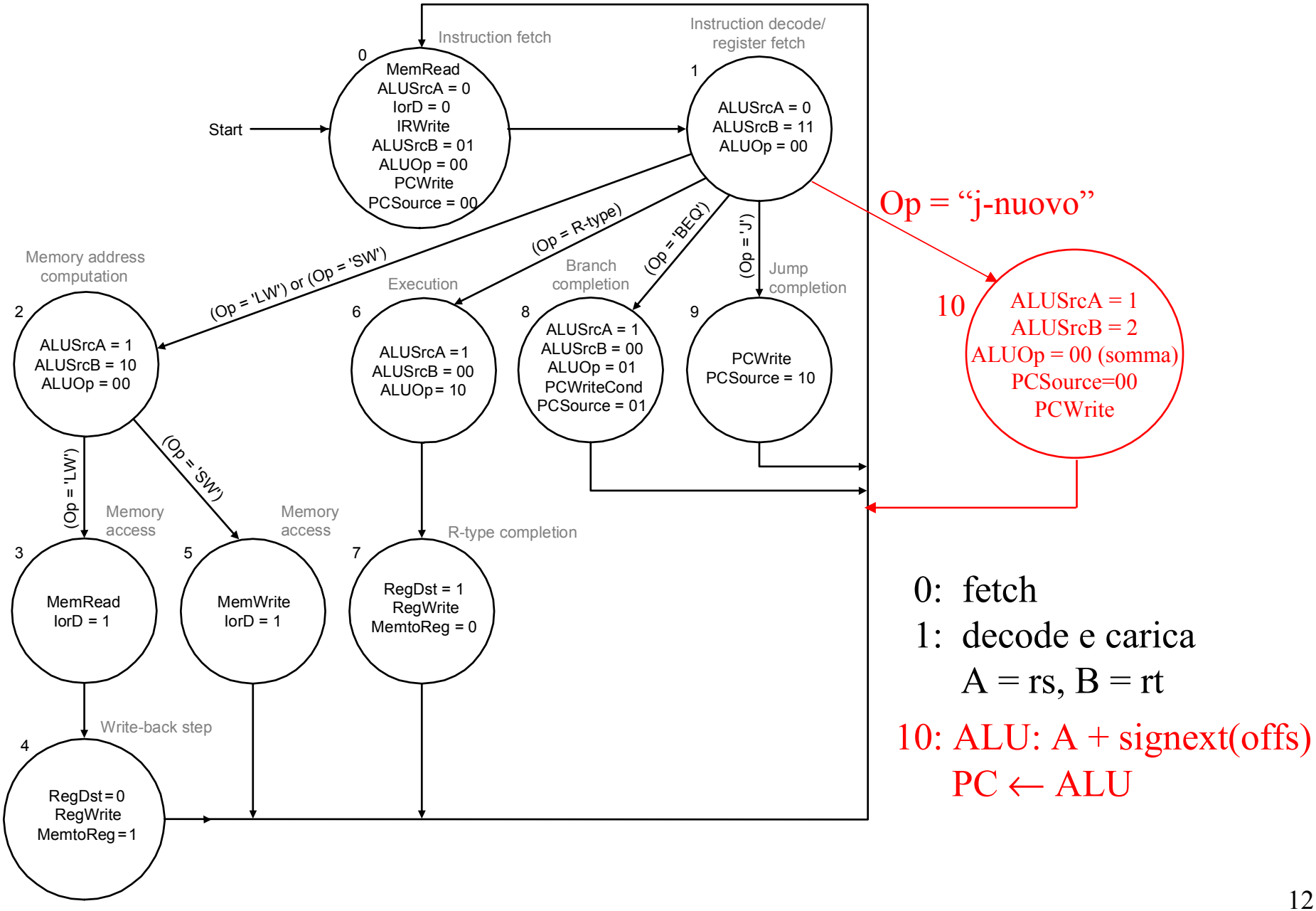
  rs + sign-extend(0) = rs

⇒ E' possibile implementare l'istruzione con il **formato Tipo-I**:



X: don't care

Vediamo il controllo con il diagramma a stati, partendo da quello originario…



0: fetch

1: decode e carica
A = rs, B = rt

10: ALU: A + signext(offs)
PC ← ALU

NB: in pratica abbiamo modificato l'ultimo stato dell'istruzione j
in modo da selezionare l'uscita della ALU (che calcola l'indirizzo)
al posto dell'usuale indirizzo derivante dall'offset a 26 bit.
Notare che in questo stato il cammino creato è A-IR → ALU → PC:
come nel caso di beq, non poniamo in serie più di un elemento "critico"

Tempo di esecuzione: 3 * 2 ns = 6 ns [come j e beq!]

Notare che è possibile implementare l'istruzione più generale (stesso formato Tipo-I)

```
j   offset(rs)                  // (il campo offset può essere diverso da zero)
```

dove però è più ragionevole supporre che offset sia espresso in parole di memoria!
A questo scopo, basta forzare ALUSrcB=11 per sommare a rs offset esteso e scalato.

Possibile problema: se il valore di rs non è multiplo di 4, potremmo saltare ad
un indirizzo non corrispondente ad una istruzione
(ciò non si verifica con la beq, dove il registro base è PC)
⇒ per gestire la cosa, occorre modificare leggermente DataPath
[Semplice esercizio da svolgere]

lw rt, (rs)++ ; rt ← M[rs] / rs = rs+4

Analisi del DataPath:

- possiamo prima procedere al calcolo dell'indirizzo come nel caso della lw:

  stato 0: fetch

  stato 1: lettura registri A ← rs, B ← rt

  stato 3: ALUOut ← rs + signext(offset) // offset deve essere nullo…

  stato 4: MDR ← M[AluOut]

  stato 5: rt ← MDR

- una volta usato rs, deve essere incrementato di 4 byte:

  useremo ALU per calcolare A + 4 [= rs +4] e porre risultato in ALUOut

  rs ← ALUOut : dobbiamo poter scrivere rs!

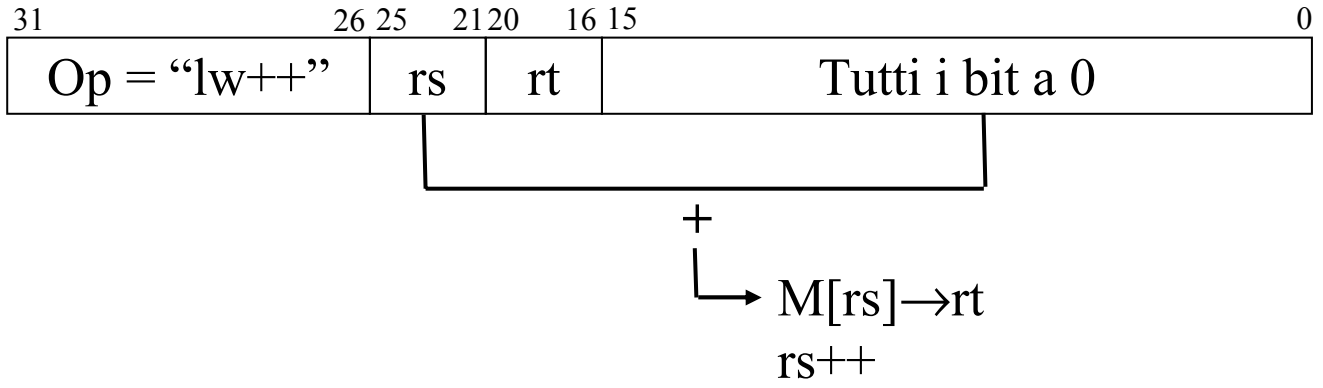
  E' necessario estendere MUX in entrata al register file,

  che attualmente prevede solo rt [per lw] e rd [per Tipo-R]

Da ciò deriva:

- il formato dell'istruzione (Tipo-I come quello della lw)
- la modifica al DataPath richiesta

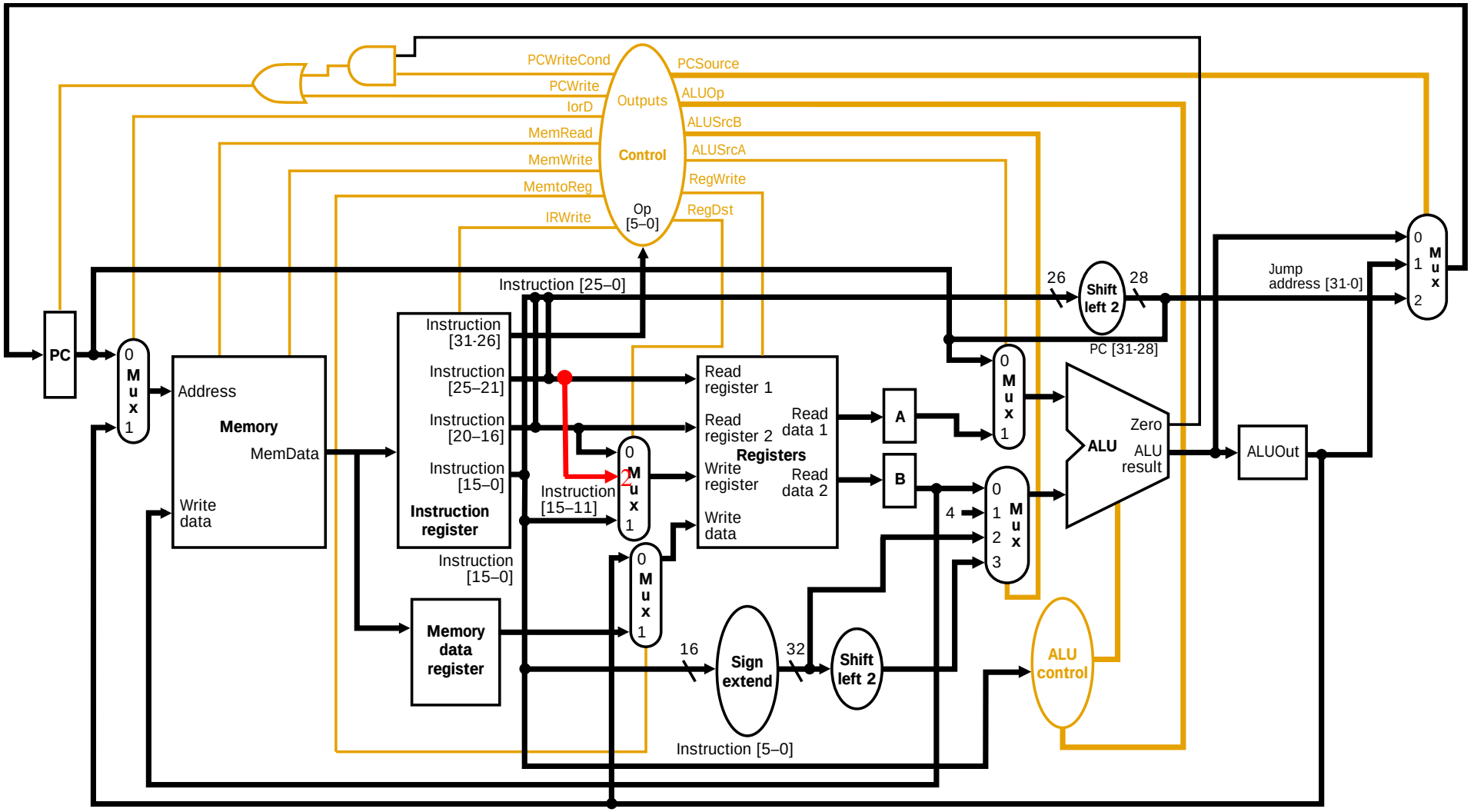
E’ possibile implementare l’istruzione con il **formato Tipo-I**:



NB: è chiaro che, permettendo offset ≠ 0, possiamo implementare una istruzione del tipo

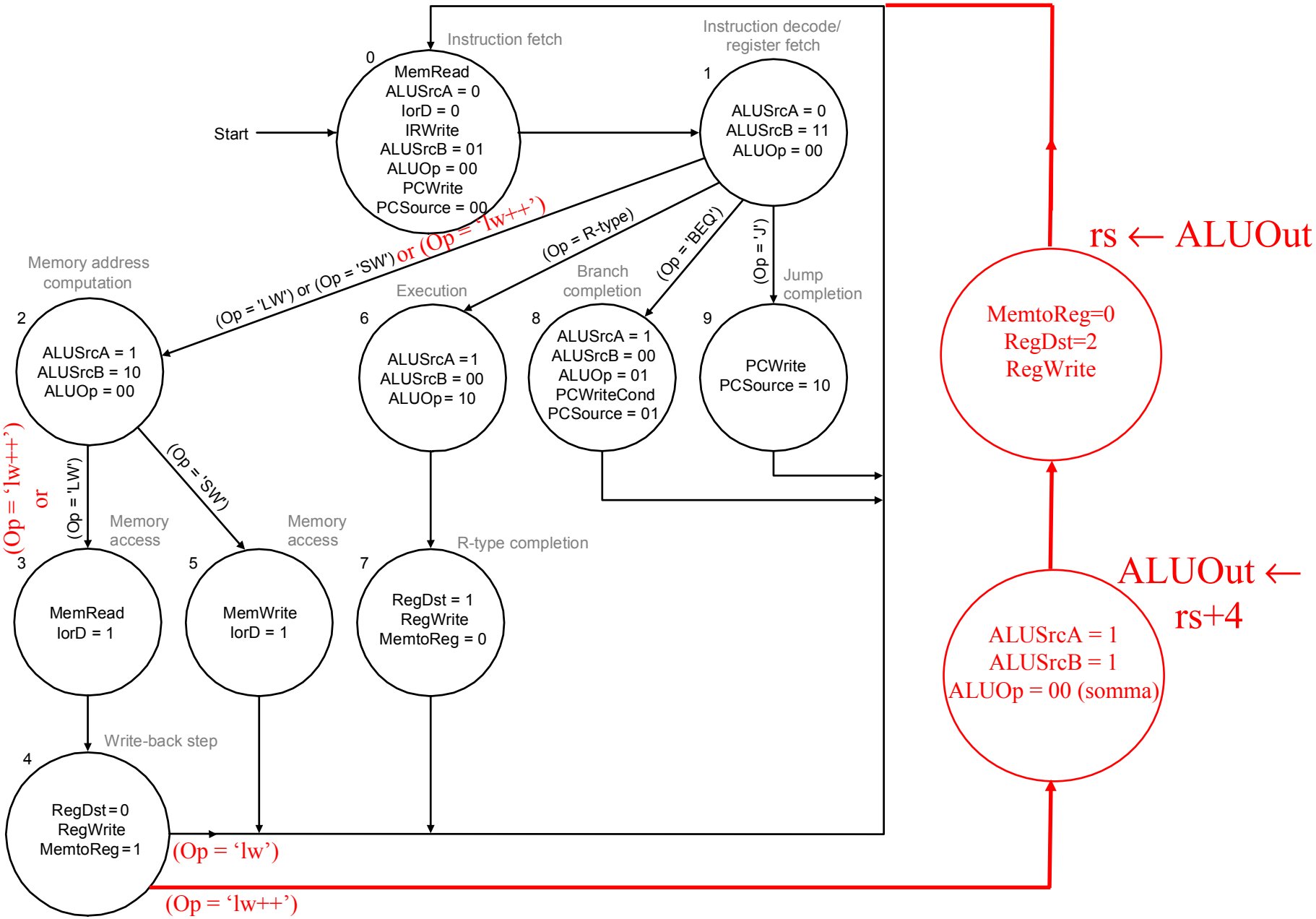
lw rt, offset(rs)++ ; rt ← M[rs+offset] / rs=rs+4

Modifica al DataPath



Aggiungiamo ingresso al MUX, con label 2 per non modificare la specifica delle precedenti istruzioni…

Possiamo allora effettuare le stesse operazioni della lw ed aggiungere in coda due stati



Questa soluzione porta a 7 cicli di clock, esecuzione in 7 * 2ns = 14 ns.

Soluzione più efficiente:

- rs è letto nel secondo stato (di decode) nel registro temporaneo A; dopo il secondo stato rs non è più letto [se lo incrementassi non ci sarebbero problemi]
- nel terzo stato A è utilizzato in ingresso ALU per calcolare l'indirizzo per la lw; dopo il terzo stato ALU non è più usata

⇒ IDEA: anticipare calcolo e scrittura di rs++ negli stati precedenti!

	Risorse per la fase "lw rt, (rs)"
Stato 0	Memoria [Fetch]
Stato 1	Lettura Reg.File [A,B←rs,rt] + ALU [ottimistica]
Stato 2	ALU [rs+offset]
Stato 3	Memoria [accesso dato]
Stato 4	Scrittura Reg. File [rt ← dato]

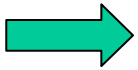
Fino a qui nessuna operazione dipende da istruzione

~~ALU [rs+4]~~

ALU [rs+4]

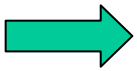
Scrittura rs

- Se calcolassimo nel penultimo stato rs+4 [posto in ALUOut] poi questo sarebbe sovrascritto nel quinto stato…
- Conviene usare la ALU per calcolare rs+4 nel quinto stato [stato 4], mentre si sta scrivendo rt con il valore prelevato in memoria.
  Posso allora aggiungere nel quinto stato (comune a lw e lw++) i segnali per il calcolo ALUOut ← rs+4; nel caso della lw, il valore non verrà usato, ma ciò non comporta una inefficienza, visto che l'operazione è svolta in parallelo!
- Basta poi aggiungere solo uno stato in più [esclusivo per la lw++] in cui si scrive rs con il valore posto in ALUOut
  [vedi diagramma a stati modificato nel lucido seguente]

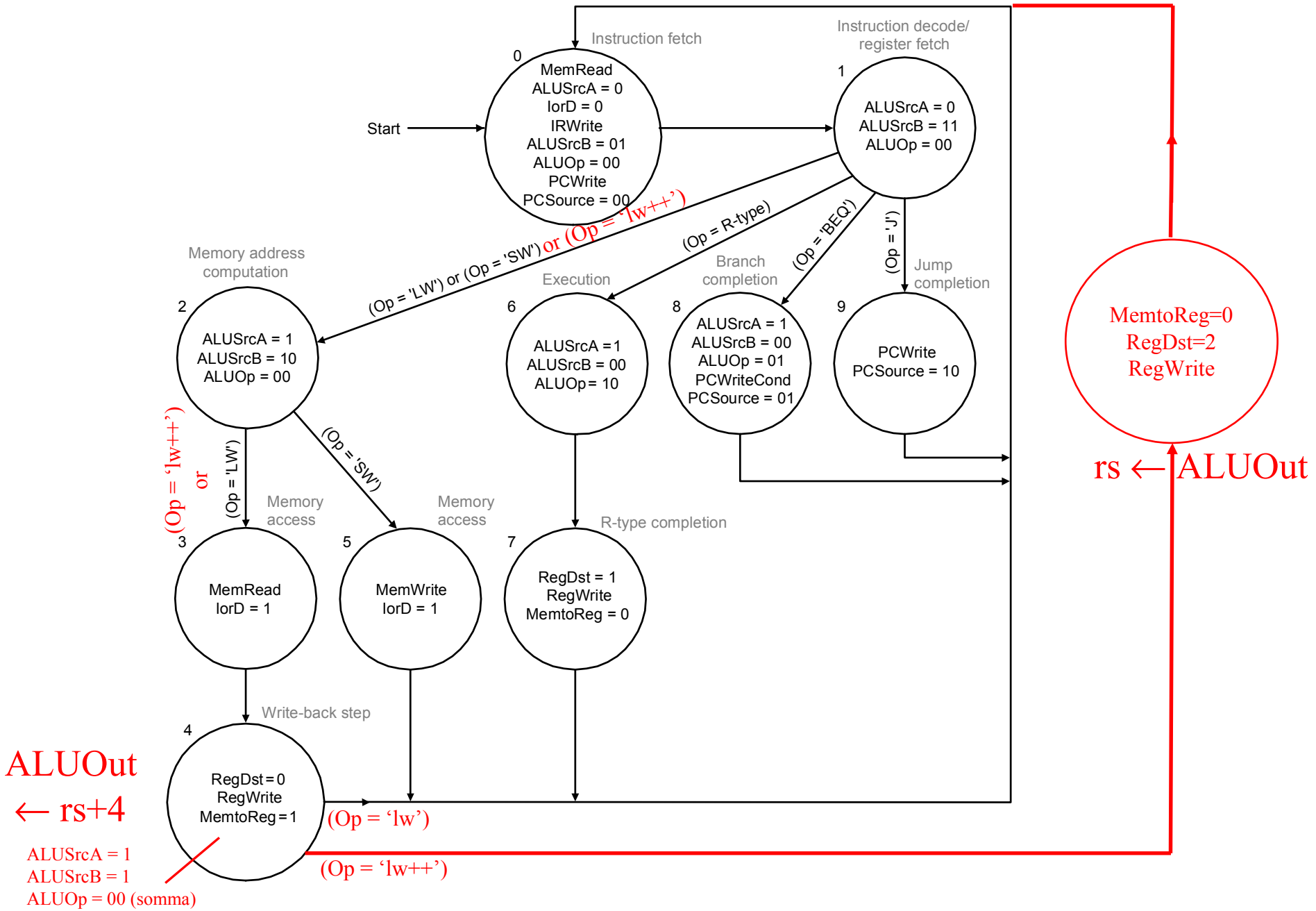


lw++ in 6 cicli di clock, pari a 6 * 2ns = 12 ns

	lw rt, (rs)++
Stato 0	Memoria [Fetch]
Stato 1	Lettura Reg.File [A,B ← rs,rt]
Stato 2	ALU [rs+offset]
Stato 3	Memoria [accesso dato] : MDR ← M[rs] A ← rs
Stato 4	Scrittura Reg. File [rs ← MDR] ALU[rs+4] : ALUout=A+4
Stato 5	Scrittura Reg. File[rs] con ALUOut



Implementazione a 6 cicli


Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
or (Op = 'lw++')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
MemtoReg=0
RegDst=2
RegWrite
rs ← ALUOut
(Op = 'lw++')
or
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1
(Op = 'lw')
(Op = 'lw++')
ALUOut
← rs+4
ALUSrcA = 1
ALUSrcB = 1
ALUOp = 00 (somma)

Confrontiamo il tempo di esecuzione dell'istruzione implementata

lw rt, (rs)++ 6 cicli di clock = 12 ns

con l'uso equivalente delle due istruzioni

lw rt, 0(rs) 5 cicli di clock +

addi rs, rs, 4 4 cicli di clock

⇒ Risparmiamo tre cicli di clock:

1 per la fase di fetch [una istruzione vs. 2]

1 per la fase di "decode" [lettura A ← rs] eseguita in parallelo

1 per la fase di calcolo rs++ eseguita in parallelo

- In questo caso, l'uso di modalità di indirizzamento più complesse (filosofia "CISC") risulta più efficiente.
- Come vedremo, con l'introduzione della **pipeline** ciò non è necessariamente vero: un certo grado di parallelismo è "automatico": ad esempio, la fase di fetch avviene sempre in parallelo con esecuzione di una o più istruzioni precedenti che si trovano ancora nella pipeline. Per contro, modalità di indirizzamento complesse rendono più difficile la gestione delle criticità…

ULTIMA DOMANDA:

Come si comporta l'istruzione implementata (nell'ultima maniera) se rs = rt?
Come dovrebbe comportarsi?

lw rs, (rs)++

Pensarci e provare a fare delle considerazioni in proposito…

lw rs, (rs)++

1) rs viene utilizzato per calcolare l'indirizzo [terzo stato: ALUOut=rs+offset]
2) Quarto stato: MDR ← M[rs] e intanto viene letto A ← rs
3) Quinto stato: rs ← M[rs] e ALUOut ← "rs vecchio" + 4
4) Sesto stato: rs ← "rs vecchio" + 4 che sovrascrive il valore precedente!

⇒ L'effetto è solo quello di sommare 4 a rs !!!

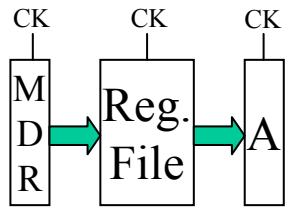
Potremmo pensare a varie possibilità:

- accettare questo comportamento
  [il programmatore non userà rt = rs per ottenere in 6 cicli il comportamento che la ADD ottiene con 4 cicli!]
- impedire l'esecuzione se rs = rt
- forzare un comportamento diverso:
  rs ← M[rs]
  rs ← rs+4

  → Notare che, dopo che rs è scritto [alla fine del quinto ciclo di clock] è necessario un altro ciclo di clock per leggere rs e caricarlo in A!

Quindi, neppure l'implementazione meno efficiente funziona!

	lw rs, (rs)++
Stato 0	Memoria [Fetch]
Stato 1	Lettura Reg.File [A,B←rs,rs]
Stato 2	ALU [rs+offset]
Stato 3	Memoria [accesso dato]
Stato 4	Scrittura Reg. File [rs ← dato] A ← rs-vecchio
Stato 5	ALU [rs-vecchio +4]
Stato 6	Scrittura Reg. File [rs-vecchio+4]



Qui rs è utilizzato in lettura con il valore vecchio!!!

Soluzione: o facciamo in modo (con HW supplementare) che, con lettura e scrittura del Register File nello stesso ciclo di clock, si legga il valore "aggiornato" oppure…

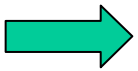
…oppure inseriamo una fase in più per leggere rs solo dopo che è stato aggiornato…

	lw rs, (rs)++
Stato 0	Memoria [Fetch]
Stato 1	Lettura Reg.File [A,B←rs,rs]
Stato 2	ALU [rs+offset]
Stato 3	Memoria [accesso dato]
Stato 4	Scrittura Reg. File [rs ← dato]
Stato 5	A ← rs
Stato 6	ALU [rs + 4]
Stato 7	Scrittura Reg. File [rs+4]

Implementazione con 8 cicli di clock vs. 9 delle due istruzioni elementari (risparmiamo solo il fetch dell'istruzione ADDI rs, rs, 4)

…oppure inseriamo un circuito che consenta di “anticipare” il dato!!!…

	lw rs, (rs)++
Stato 0	Memoria [Fetch]
Stato 1	Lettura Reg.File [A,B ← rs,rs]
Stato 2	ALU [rs+offset]
Stato 3	Memoria [accesso dato] : MDR ← M[rs]
Stato 4	Scrittura Reg. File [rs ← MDR] ALUout=MDR+4
Stato 5	Scrittura Reg. File [rs+4]



Implementazione a 6 cicli [-3 rispetto a due istruzioni]

… come si vede, le considerazioni fatte sono molto simili a quelle che riguardano la gestione delle criticità sui dati della pipeline…

… e in effetti, tutto ciò suggerisce l'idea di estendere il tentativo di parallelizzare le operazioni appartenenti ad istruzioni diverse…

⇒ CONCETTO DI PIPELINE … !