Lezione 20
CPU con pipeline

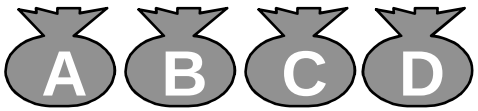
http://www.dii.unisi.it/~giorgi/didattica/arcal1

All figures from Computer Organization and Design: The Hardware/Software Approach, Second Edition, by David Patterson and John Hennessy, are copyrighted material. (COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED.) Figures may be reproduced only for classroom or personal educational use in conjunction with the book and only when the above copyright line is included. They may not be otherwise reproduced, distributed, or incorporated into other works without the prior written consent of the publisher.

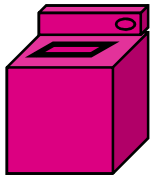
Other material is adapted from CS152 Copyright (C) 2000 UCB

Il Pipelining e' Naturale!

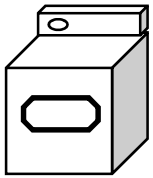
- Esempio della Lavanderia
- Ann, Brian, Cathy, Dave devono:



- Lavare



- Asciugare

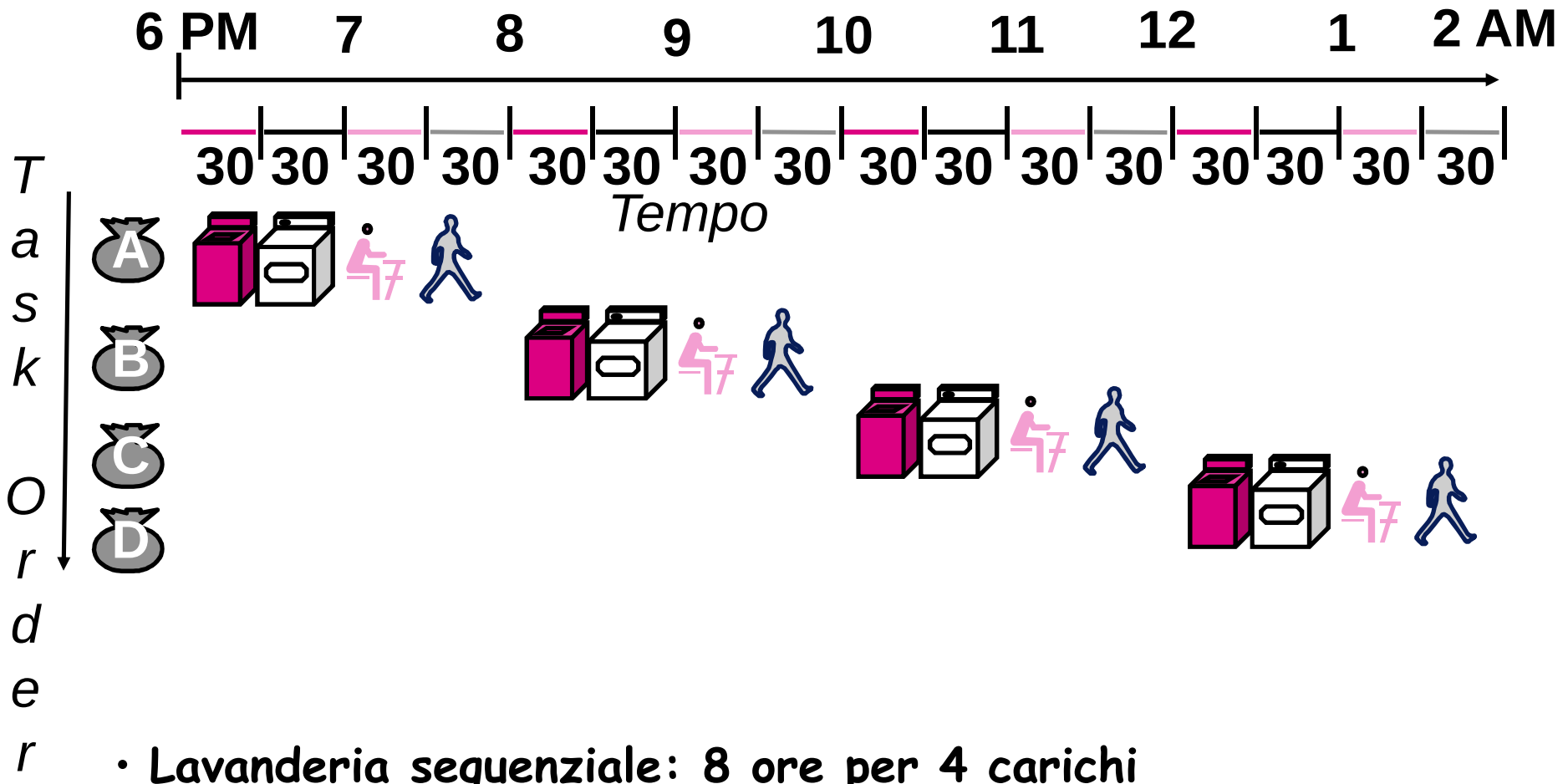


- Stirare

- Mettere a posto

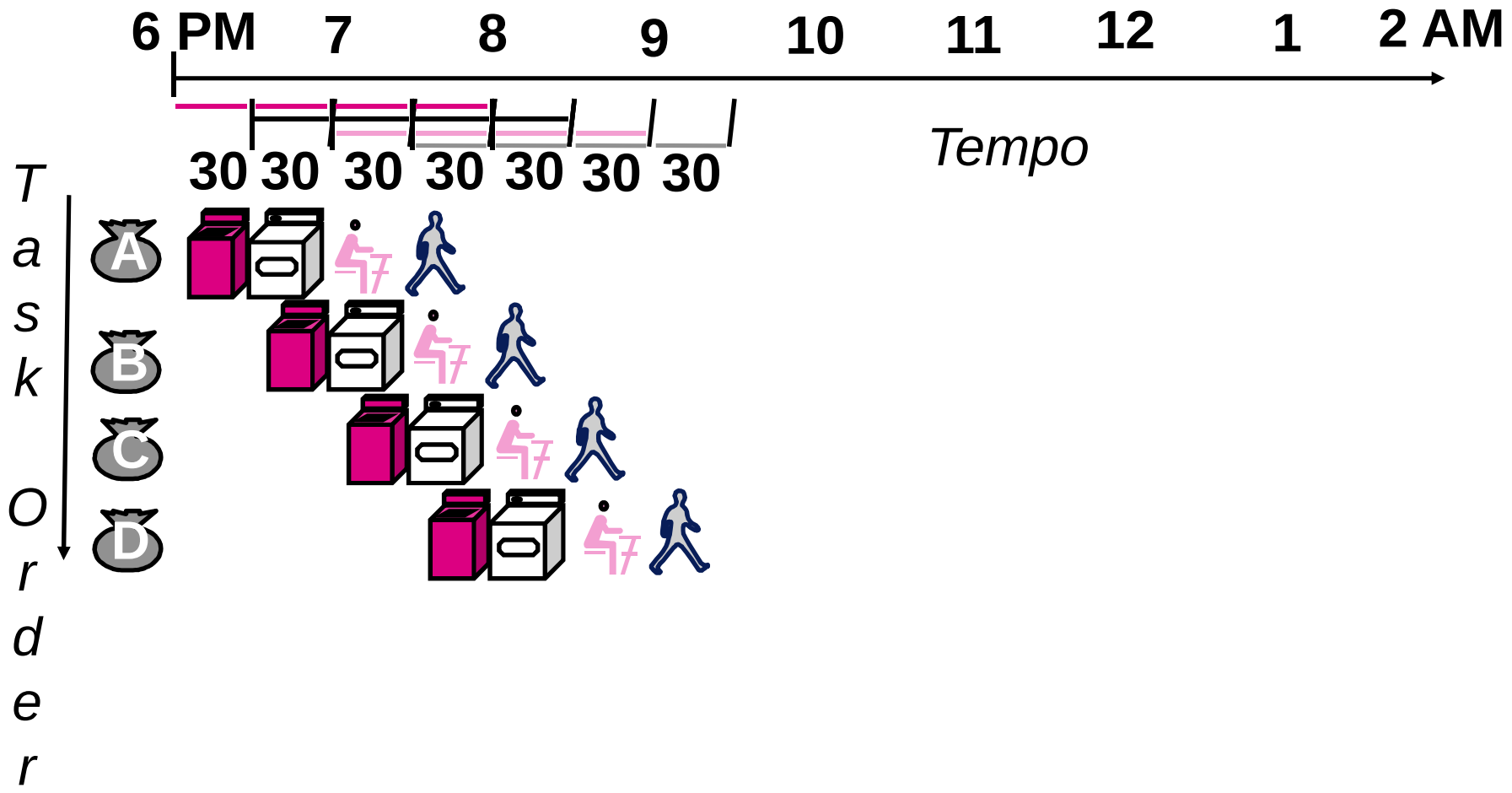
Supponiamo che ognuna di queste operazioni richieda 30 minuti
e che ci sia una sola lavatrice, una asciugatrice, un ferro da stiro, un banco

Lavanderia Sequenziale



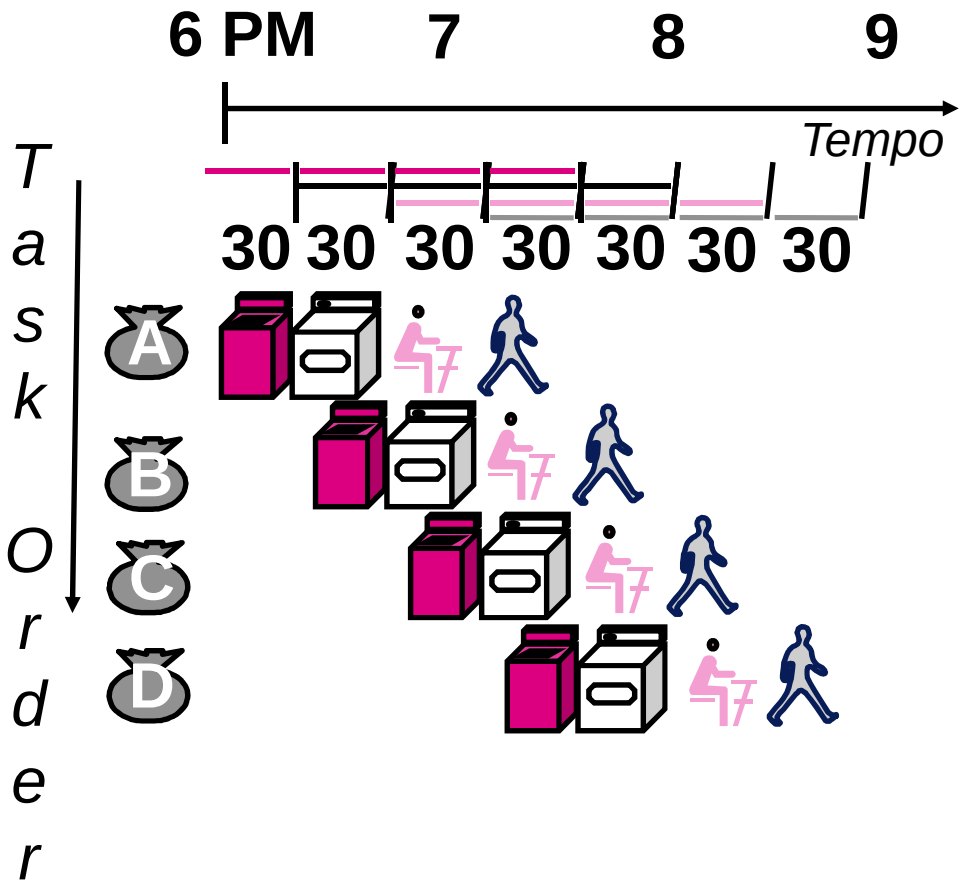
- Lavanderia sequenziale: 8 ore per 4 carichi
- Se utilizzassero il pipelining, quanto impiegherebbero?

Lavanderia a Pipeline: "Start work ASAP"



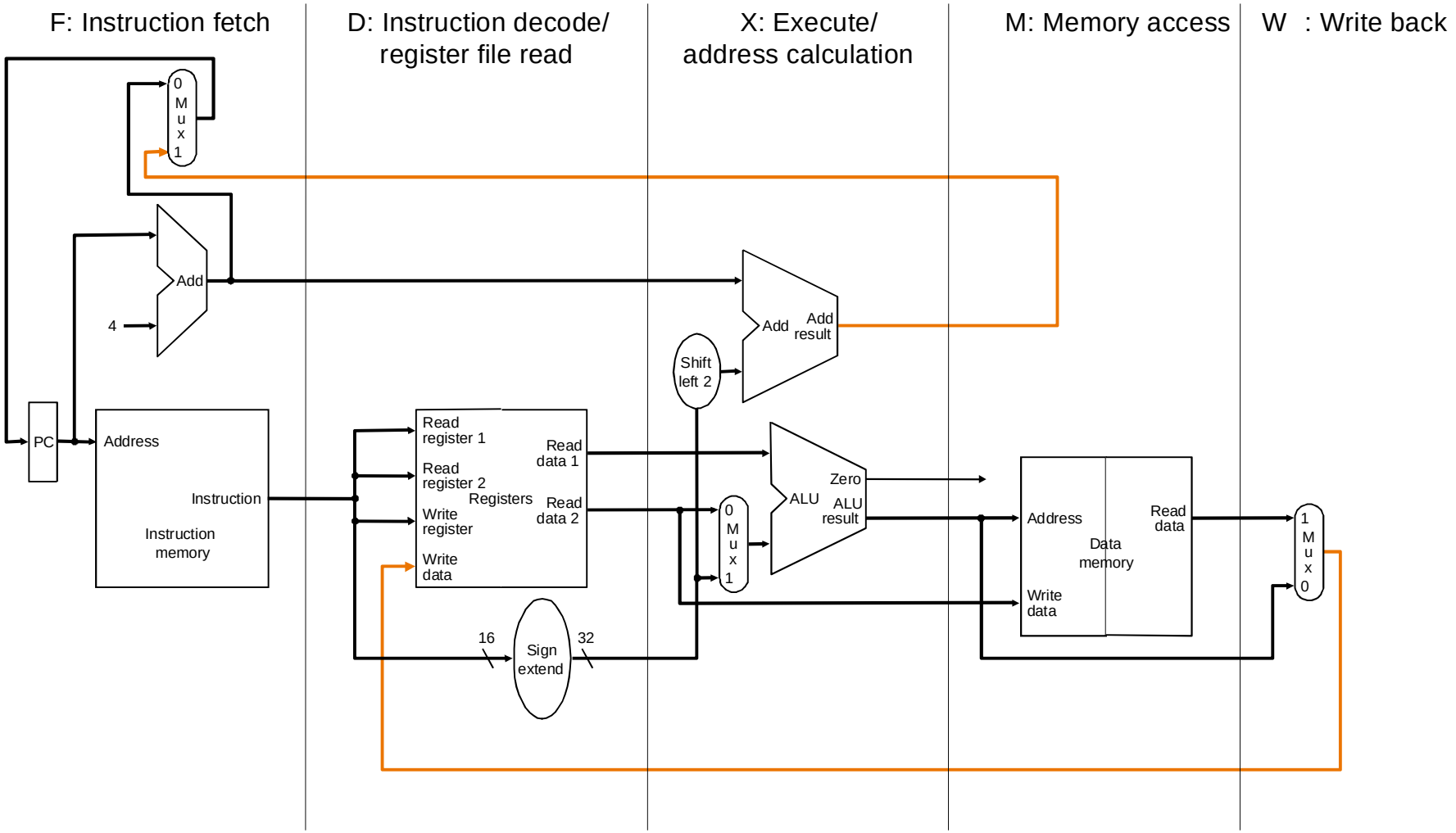
- Lavanderia a pipeline: 3.5 ore per 4 carichi!

Le lezioni di pipelining



- Il pipelining non riduce il tempo di esecuzione di ogni singola istruzione, bensi' aumenta il loro throughput
- Tutti gli stadi lavorano in contemporanea utilizzando risorse separate
- Incremento potenziale di velocita' = numero di stadi della pipeline
- La velocita' della pipeline e' limitata dallo stadio piu' lento
- Gli stadi della pipeline non bilanciati riducono l'incremento di velocita'
- Il tempo per riempire la pipeline ed il tempo per svuotarla riducono l'incremento di velocita'
- Stallo per dipendenze

Idea di base: suddivisione in stadi della CPU



Cosa dobbiamo aggiungere per suddividere il datapath in stadi?

I cinque passi per eseguire un'istruzione

- F = Prelievo dell'istruzione (Instruction Fetch)
- D = Decodifica dell'istruzione (Instruction Decode) e lettura dei registri (Register Fetch)
- X = Esecuzione, calcolo dell'indirizzo di memoria, completamento del salto (Branch Completion)
- M = Accesso alla memoria o completamento di istruzione di tipo R
- W = Scrittura del risultato in un registro (Write-Back)

LE ISTRUZIONI RICHIEDONO 3 - 5 CICLI!

Passo 1: Instruction Fetch

- Viene letta dalla memoria l'istruzione puntata dal Program Counter (PC)
- L'istruzione viene posta nell' Instruction Register (IR)
- PC viene incrementato di 4 e riscritto nuovamente in PC
- Tali operazioni si possono descrivere utilizzando un linguaggio RTL (Register-Transfer Language)

```
IR = Memory[PC];
PC = PC + 4;
```

- Da IR si determinano i valori dei segnali di controllo

Qual e' il vantaggio di aggiornare il PC in questo passo?

Passo 2: Instruction Decode e Register Fetch

- I contenuti dei registri indicati nei campi `rs` e `rt` vengono letti e trasferiti nei registri di appoggio *A* e *B*
  - Non e' detto che siano successivamente utilizzati…
- Calcolo dell' "indirizzo target" del salto, nel caso in cui l'istruzione sia un salto
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- Le linee di controllo non assumono alcun valore in dipendenza dal tipo di istruzione (per il momento, dato che la decodifica e' ancora in corso)

Sign-extend effettua l'estensione da 16 a 32 bit **con segno**

Passo 3 (dipendente dall'istruzione)

- La ALU esegue una delle seguenti tre funzioni, in base al tipo di istruzione:
  - Accesso alla memoria:

```
ALUOut = A + sign-extend(IR[15-0]);
```

  - Istruzione di tipo R relativa ad una operazione op:

```
ALUOut = A op B;
```

  - Salto:

```
if (A == B) PC = ALUOut;
```

Passo 4 (R-type o accesso alla memoria)

- Le istruzioni load e store accedono alla memoria

```
MDR = Memory[ALUOut];      (load)
Memory[ALUOut] = B;        (store)
```

- Le istruzioni di tipo R possono scrivere il risultato

```
Reg[IR[15-11]] = ALUOut;
```

La scrittura avviene alla fine del ciclo, sul fronte in discesa

Passo 5: Write-Back

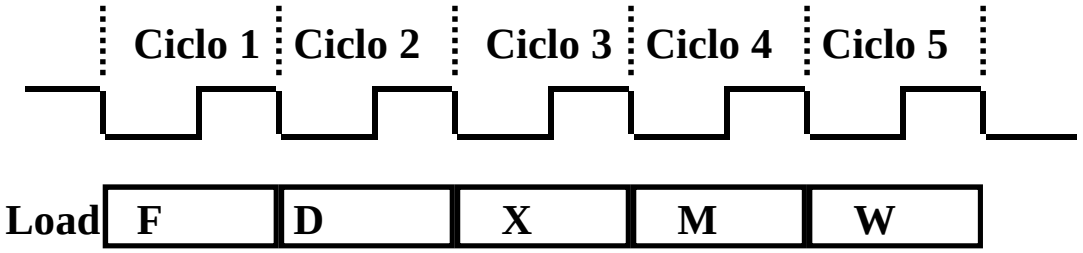
```
Reg[IR[20-16]]= MDR;
```

Cosa succede alle altre istruzioni?

Riassumendo

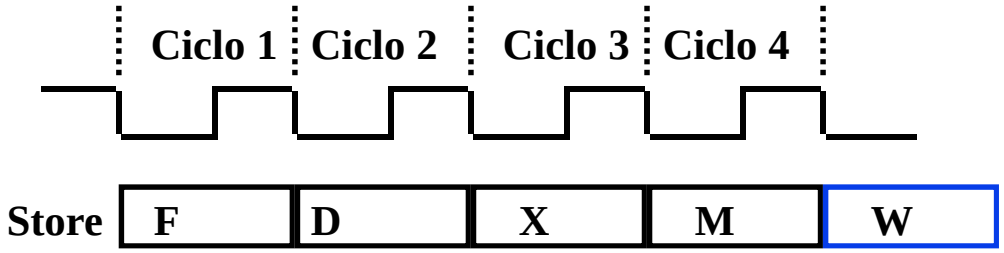
Nome del passo	Operazioni per le istruzioni di tipo R	Operazioni per le istruzioni che accedono alla memoria	Operazioni per branch	Operazioni per jump
Prelievo dell'istruzione	IR = Memory[PC] PC = PC + 4			
Decodifica dell'istruzione /lettura dei registri	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Esecuzione, calcolo del inidirizzo, completamento di branch/jump	ALUOut = A op B	ALUOut = A + sign-extend(IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Accesso alla memoria o completamento dell'istruzione di tipo R	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] oppure Store: Memory [ALUOut] = B		
Completamento della lettura della memoria		Load: Reg[IR[20-16]] = MDR		

Ciclo Multiplo: i cinque stadi della load



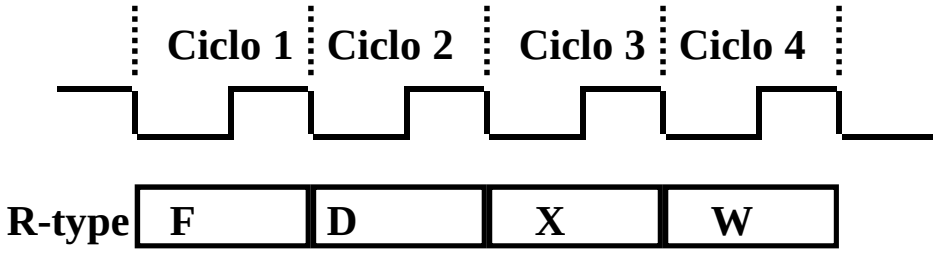
- F: Instruction Fetch
  - Prelievo dell'istruzione dalla memoria istruzioni
- D: Registers Fetch, Instruction Decode
  - Lettura dei registri e decodifica delle istruzioni
- X: Calcolo dell'indirizzo di memoria
- M: Lettura dei dati dalla memoria dati
- W: Scrittura dei dati nel register file

Ciclo Multiplo: i quattro stadi della store



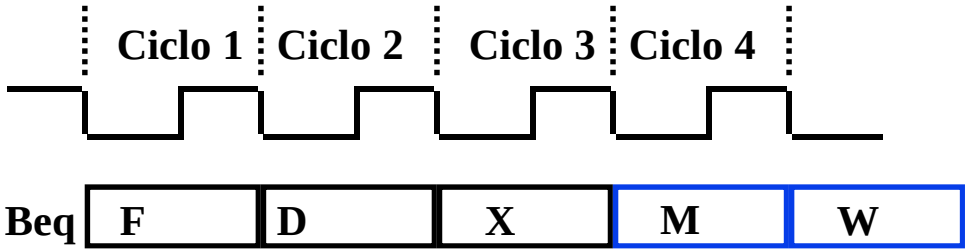
- F: Instruction Fetch
  - Prelievo dell'istruzione dalla memoria istruzioni
- D: Registers Fetch, Instruction Decode
- X: Calcolo dell'indirizzo di memoria
- M: Scrittura dei dati in memoria dati

Ciclo Multiplo: i quattro stadi dell'istruzione di tipo R



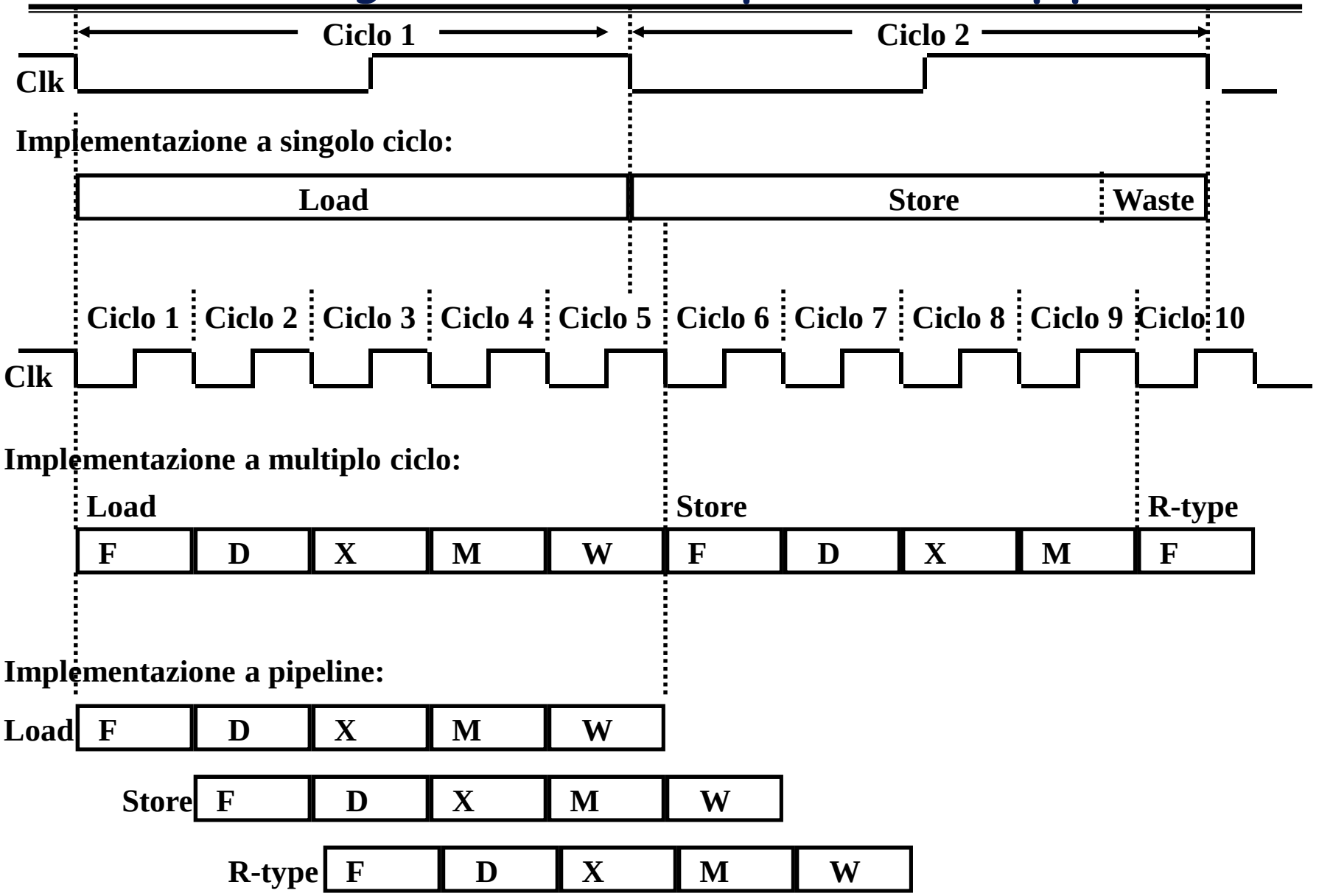
- F: Instruction Fetch
  - Prelievo dell'istruzione dalla memoria istruzioni
- D: Registers Fetch, Instruction Decode
- X:
  - ALU opera su due operandi in registri
  - Aggiornamento del PC
- W: Scrittura del risultato dall'ALU nel register file

Ciclo Multiplo: i tre stadi della Beq



- F: Instruction Fetch
  - Prelievo dell'istruzione dalla memoria istruzioni
- D:
  - Registers Fetch, Instruction Decode
- X:
  - confronto dei due operandi in registri,
  - selezione dell'indirizzo di salto corretto
  - memorizzazione dell'indirizzo in PC

Modelli a singolo ciclo e multiplo ciclo vs. pipeline

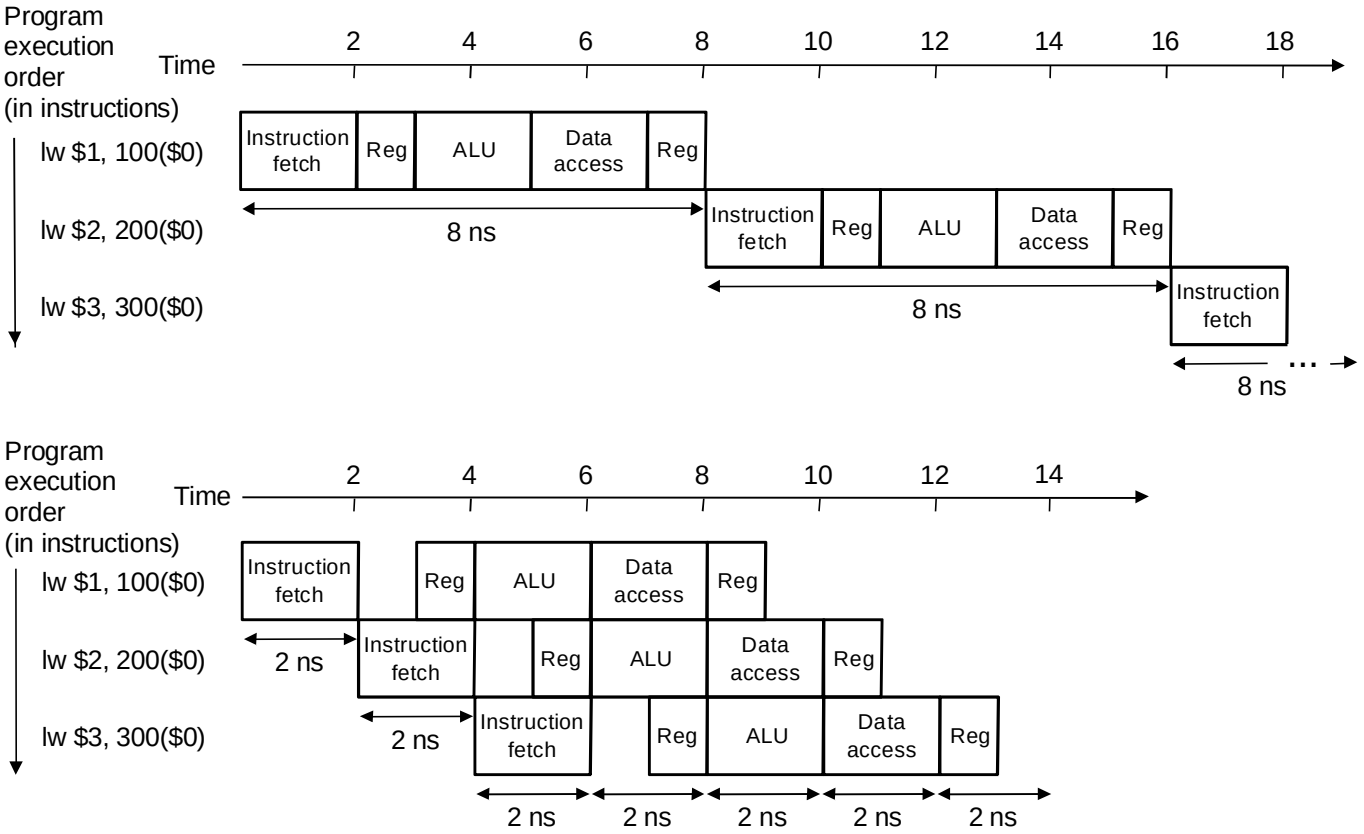


Esercizio: verificare la bonta' del pipelining

- Supponiamo che si eseguano 100 instruzioni
- Macchina a singolo ciclo
  - 45 ns/ciclo x 1 CPI x 100 inst =
    4500 ns
- Macchina a multiplo ciclo
  - 10 ns/ciclo x 4.6 CPI (dovuto al mix di istruzioni) x 100 inst =
    4600 ns
- Macchina con pipeline ideale
  - 10 ns/ciclo x (1 CPI x 100 inst + 4 cicli per riempimento + 4 cicli per svuotamento) =
    1080 ns

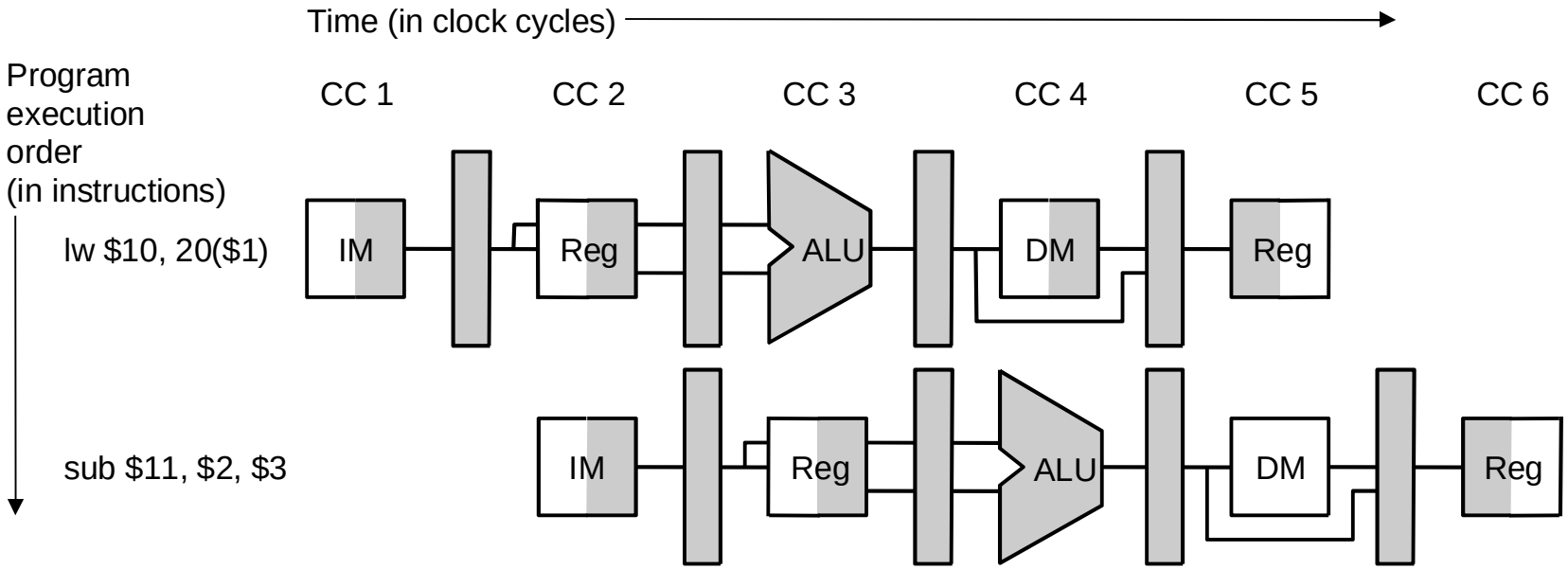
Il Pipelining

- **Migliora le prestazioni aumentando il throughput delle istruzioni**



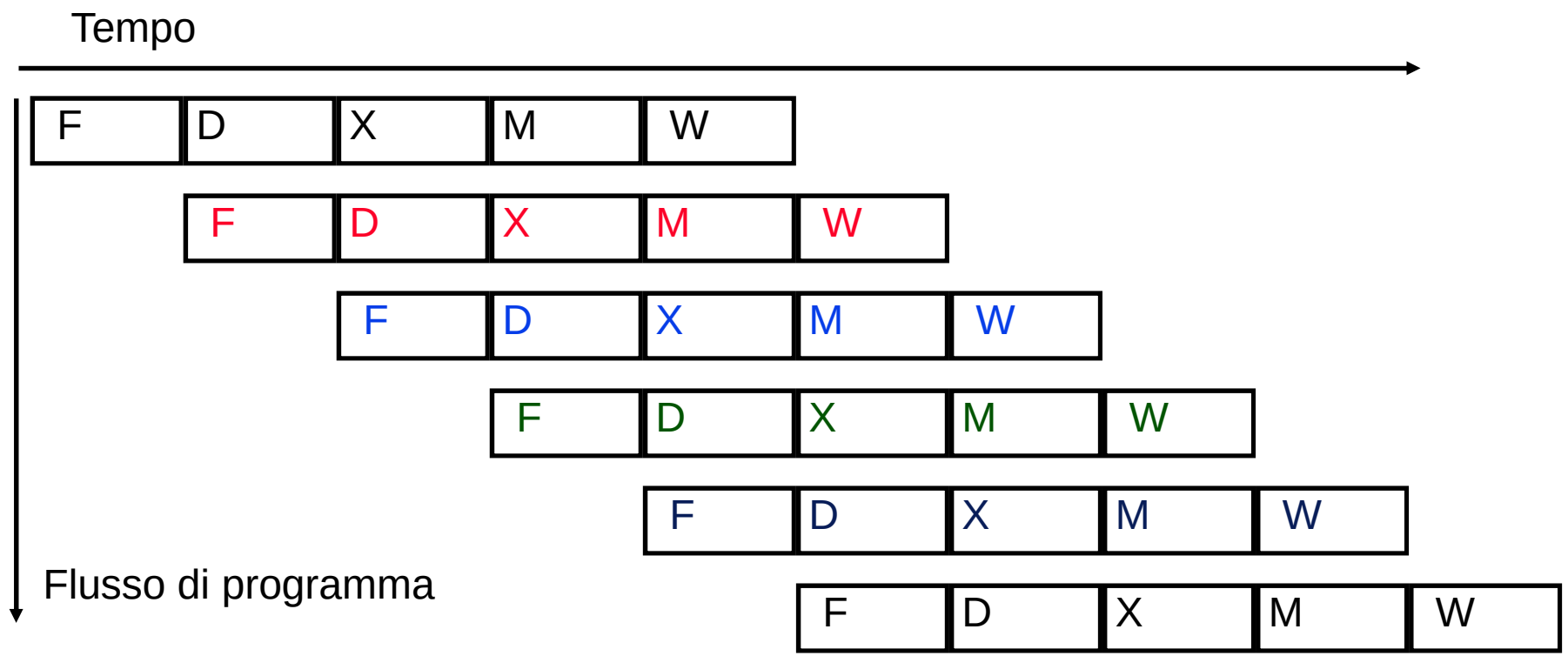
L' aumento di velocita' ideale e' pari al numero di stadi della pipeline. Riusciamo ad ottenerlo?

Rappresentazione grafica delle pipeline

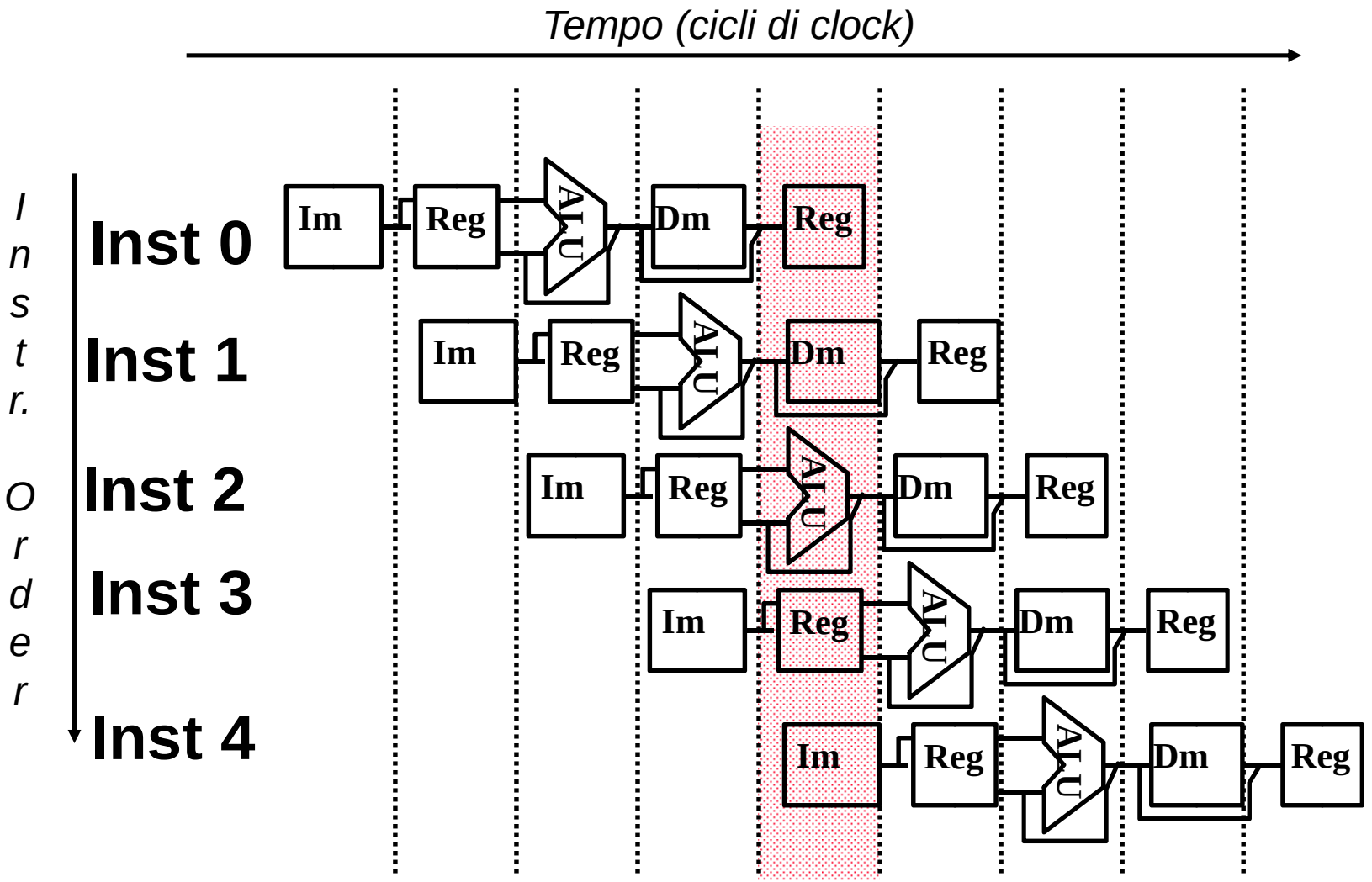


- L'uso della rappresentazione grafica aiuta a capire il datapath:
  - quanti cicli utilizza l'esecuzione di questo codice?
  - cosa fa l'ALU durante il ciclo 4?

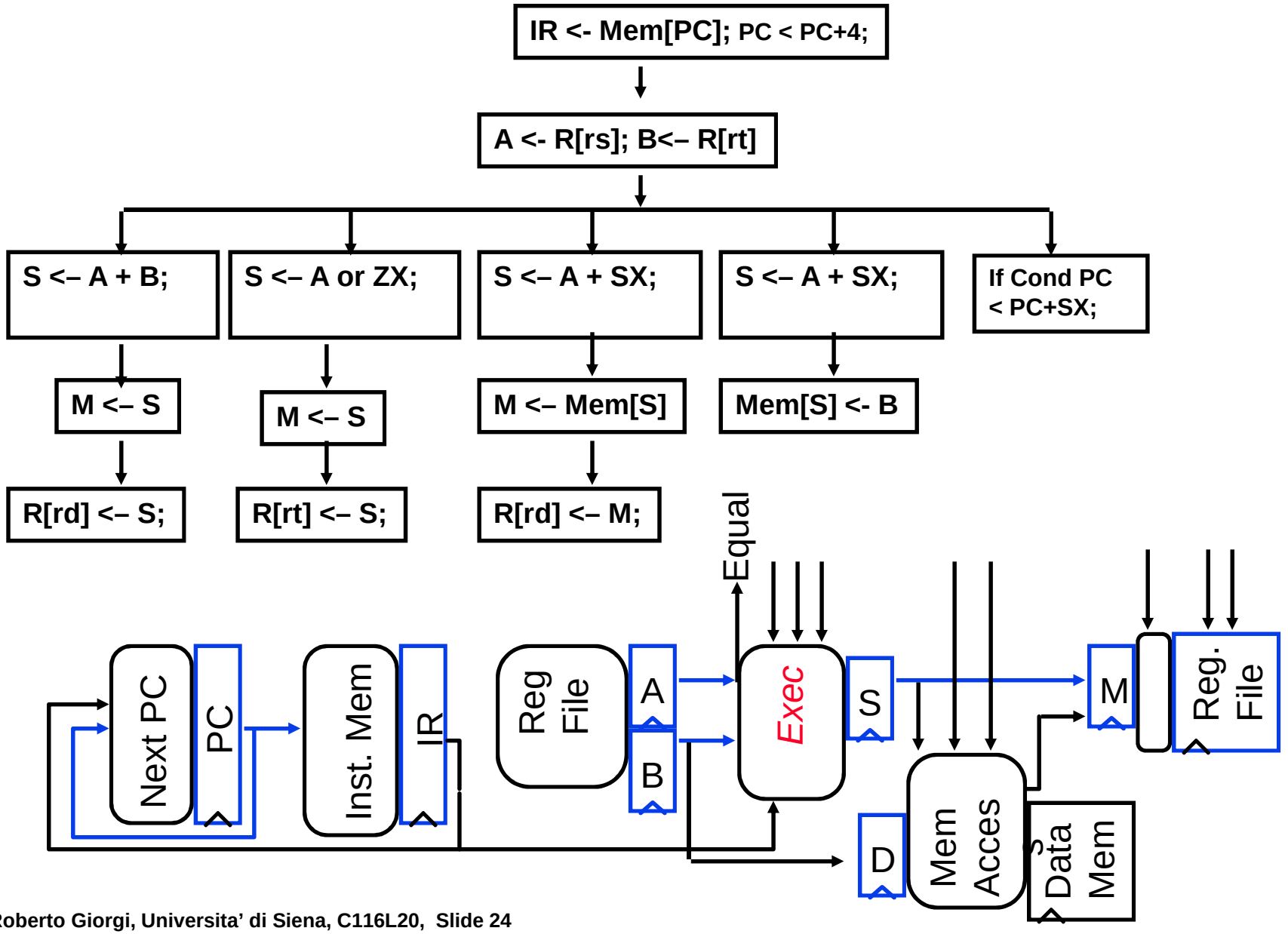
Rappresentazione convenzionale dell'esecuzione a pipeline



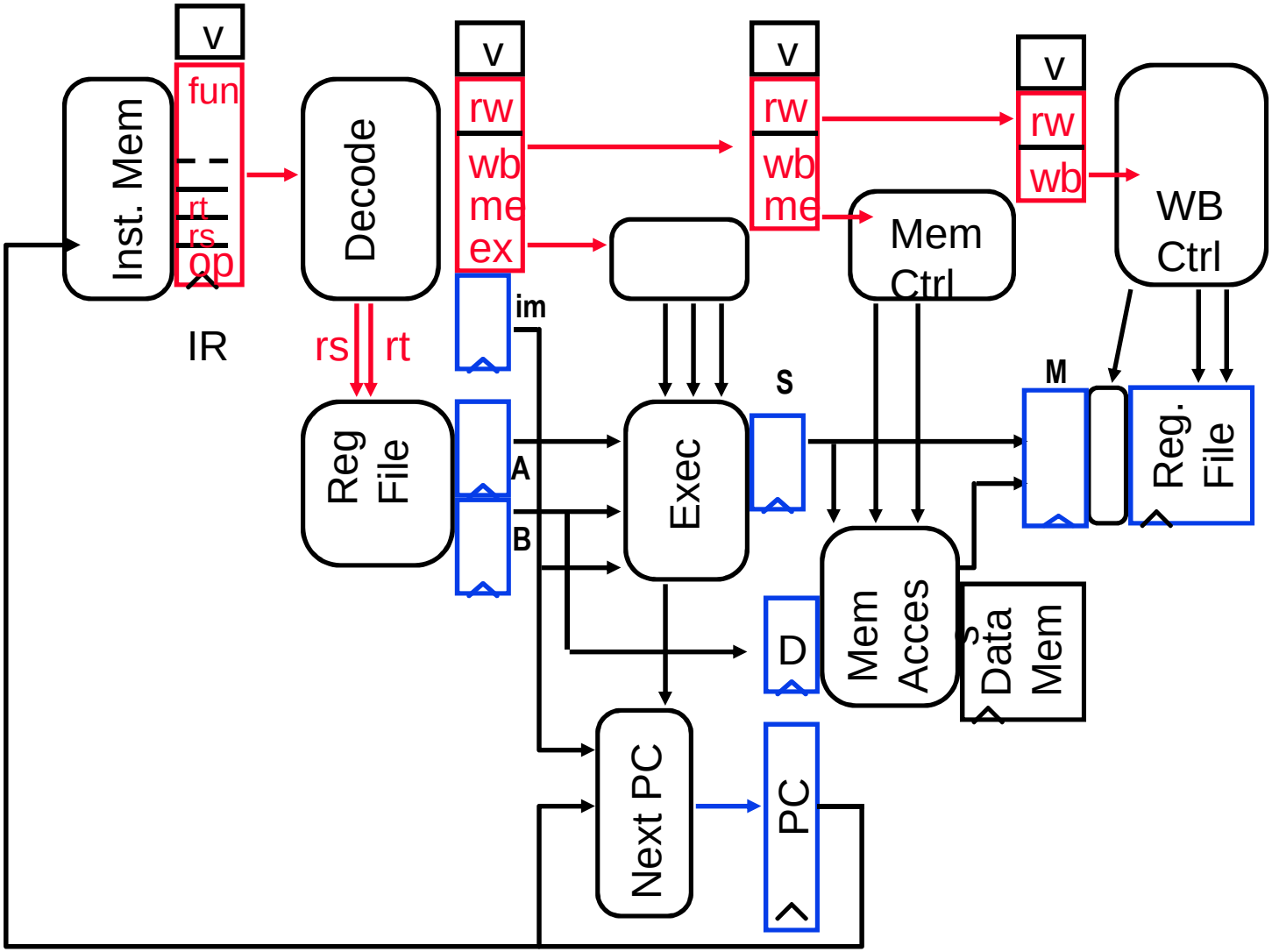
Perche' la pipeline? Utilizzo ottimale delle risorse!



Datapath semplificato



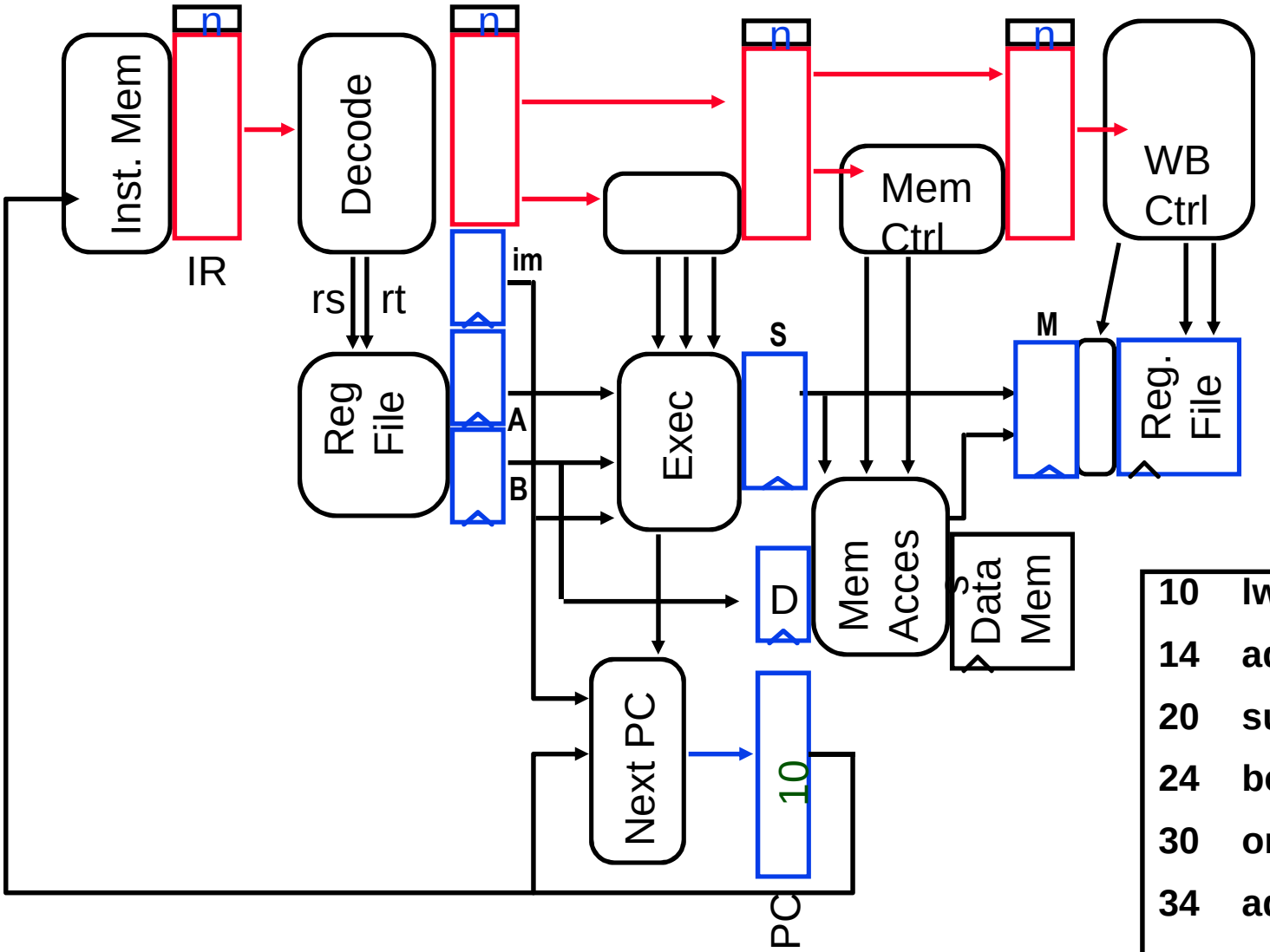
Datapath + Data Stationary Control



Proviamolo!

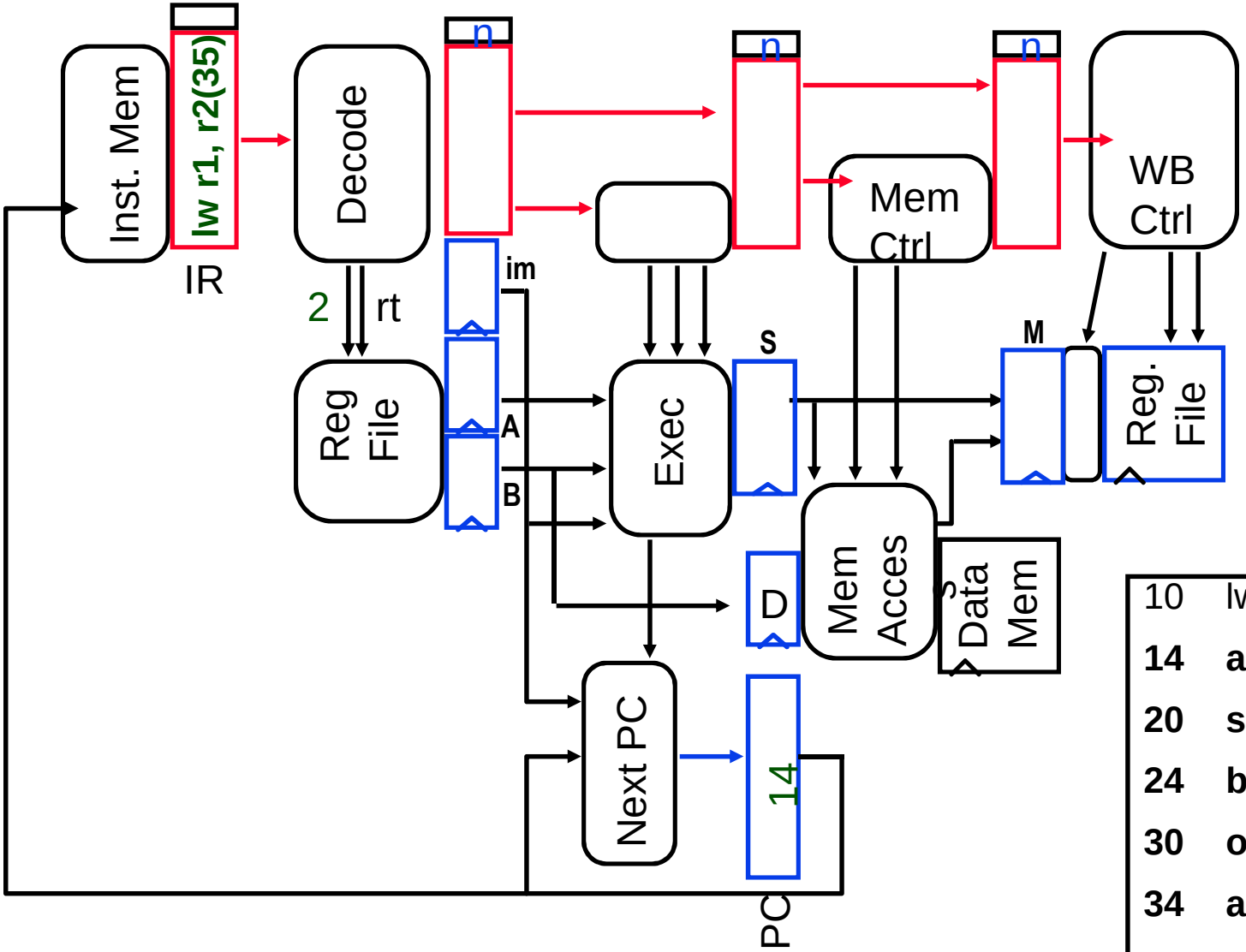
```
10     lw     r1, r2(35)
14     addi   r2, r2, 3
20     sub    r3, r4, r5
24     beq    r6, r7, 100
30     ori    r8, r9, 17
34     add    r10, r11, r12

100    and    r13, r14, r15
```

Start: Fetch 10



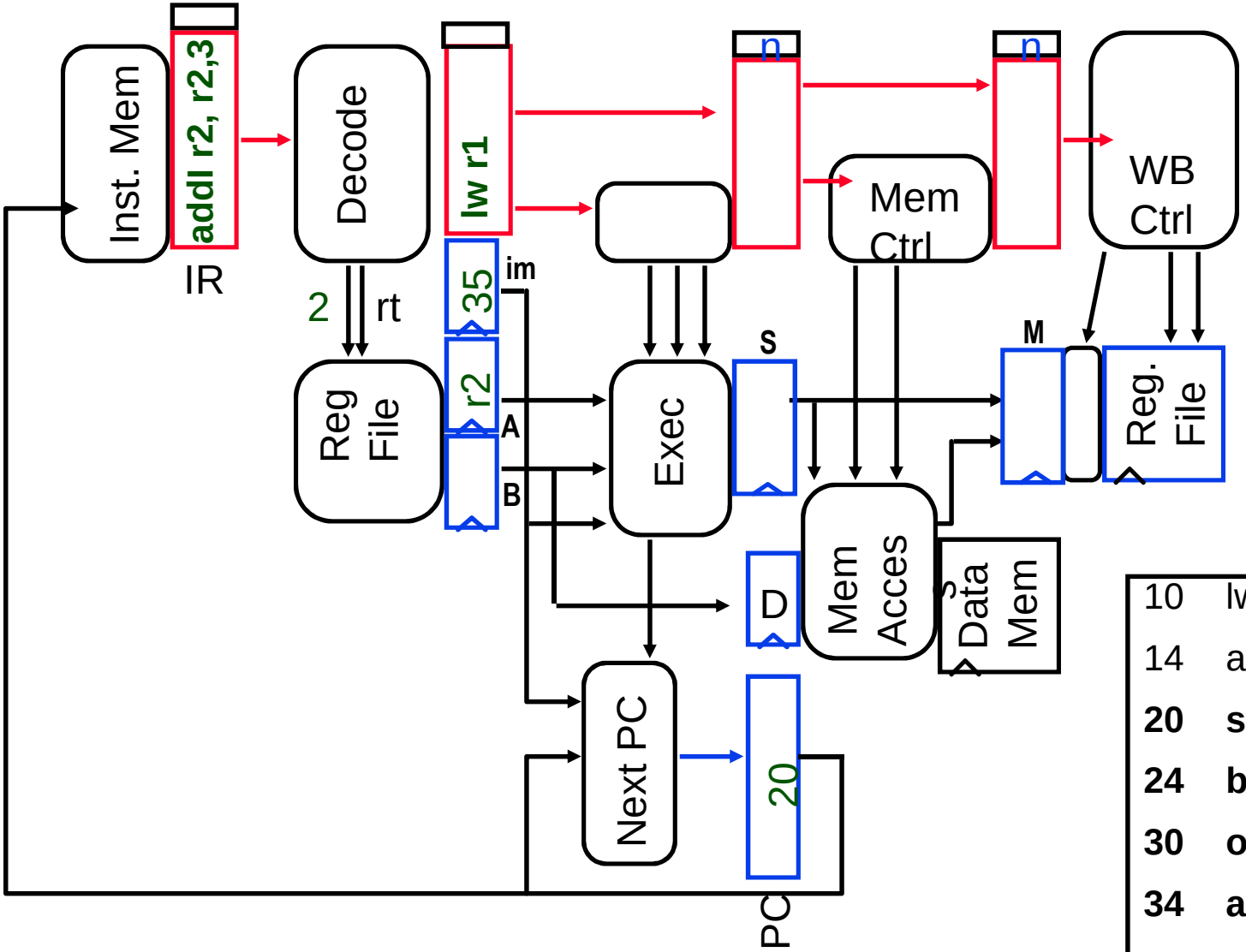
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, r15

Fetch 14, Decode 10



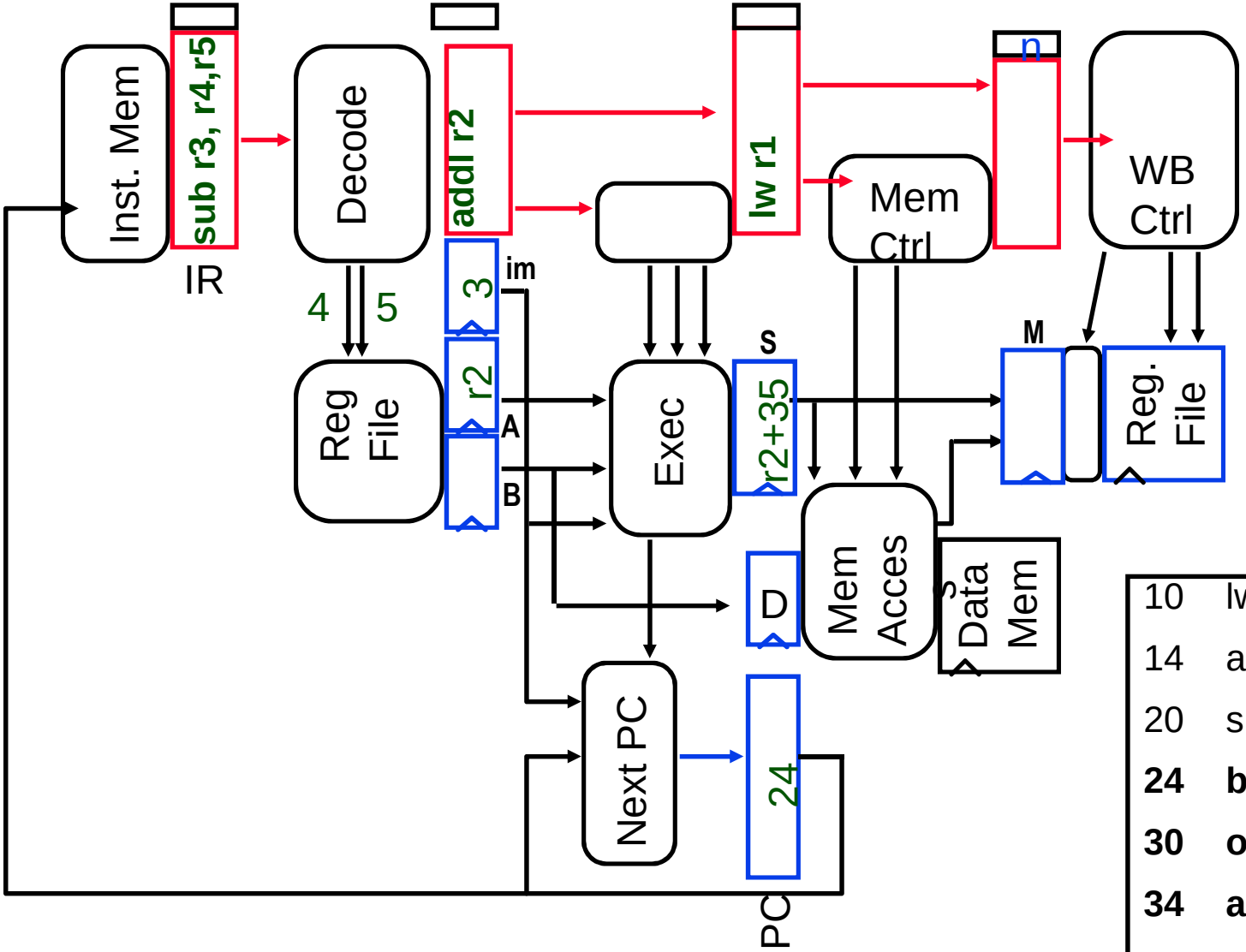
10	lw	r1, r2(35)
14	**addi**	**r2, r2, 3**
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, r15**

Fetch 20, Decode 14, Exec 10



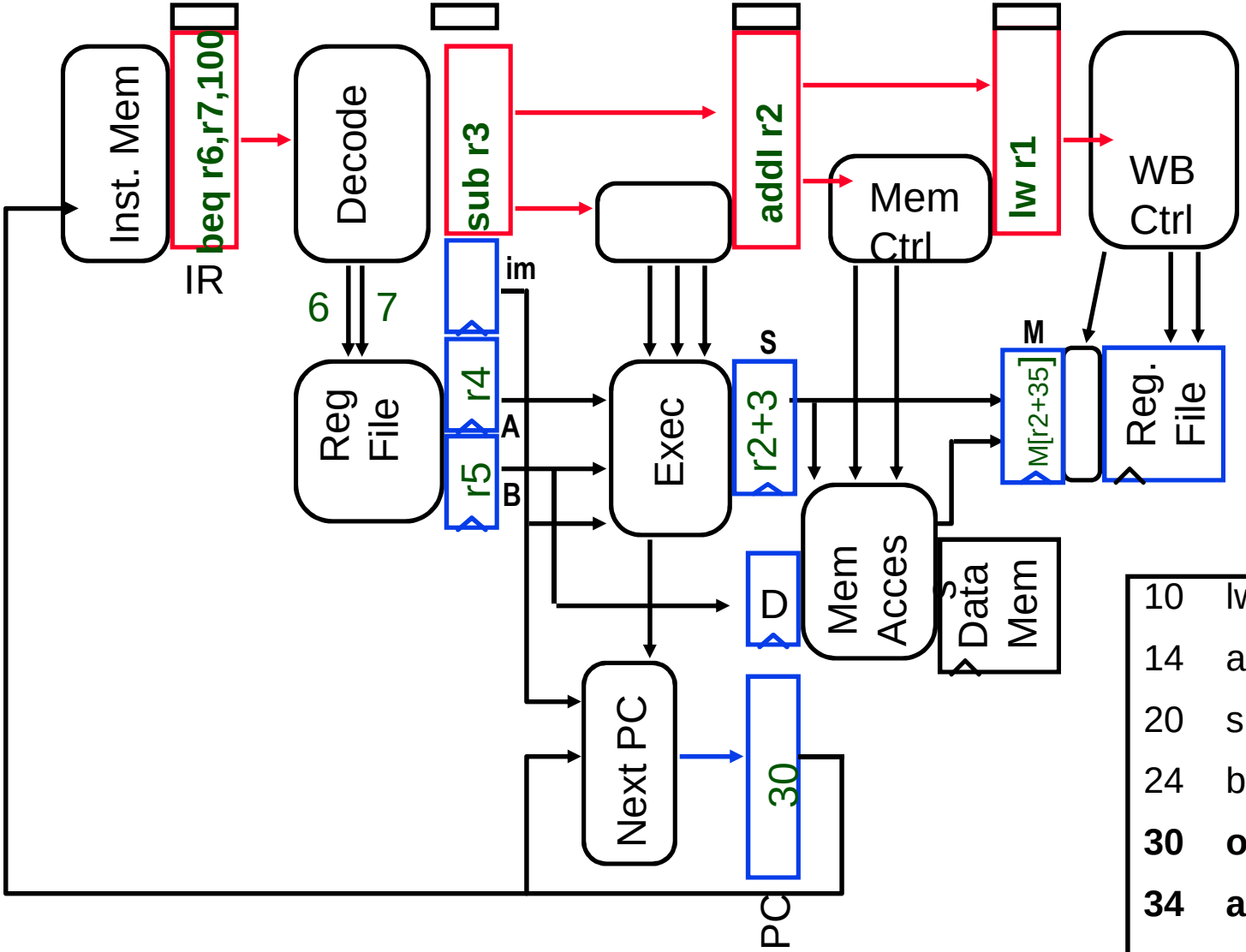
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, r15**

Fetch 24, Decode 20, Exec 14, Mem 10



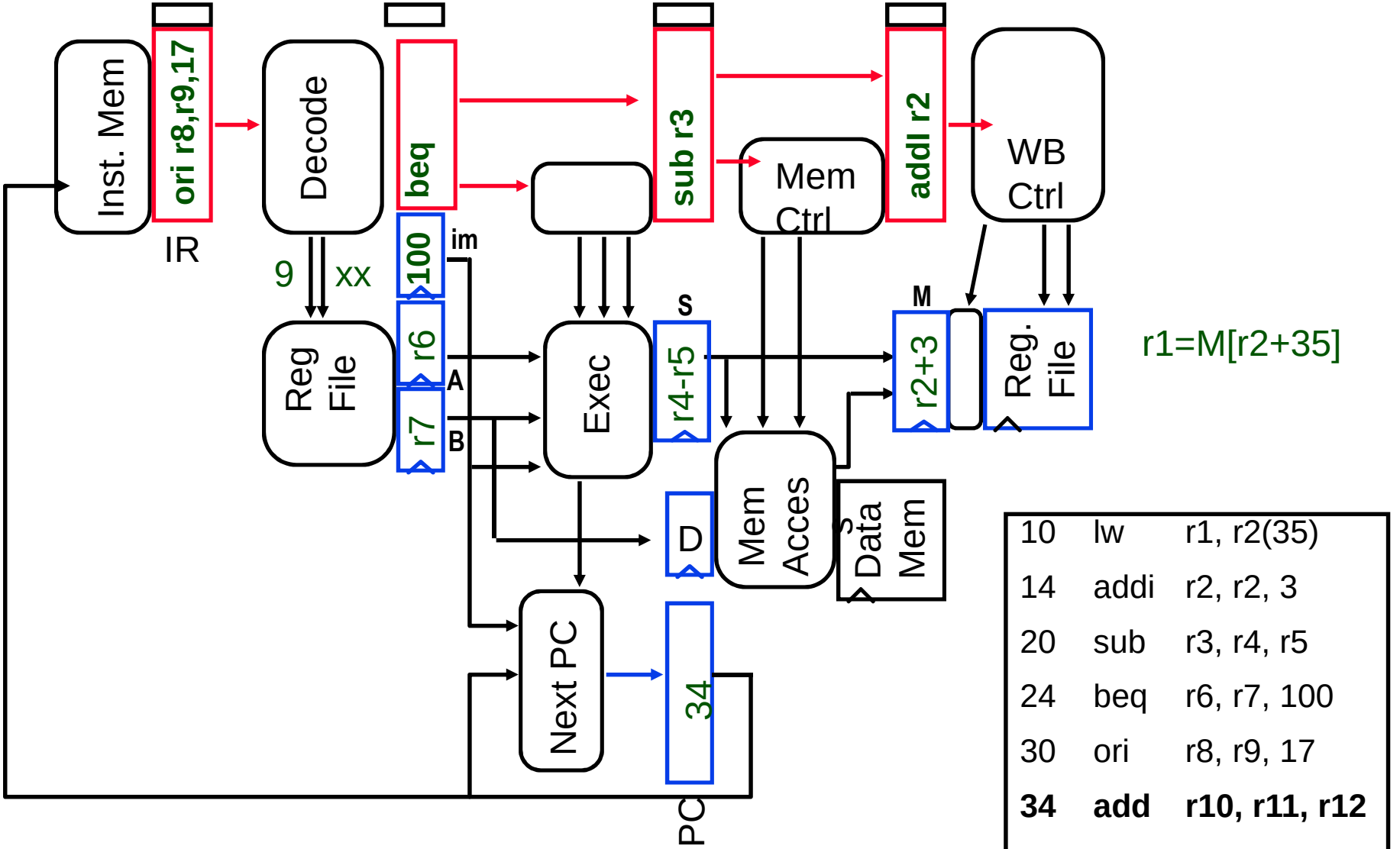
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, r15**

Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



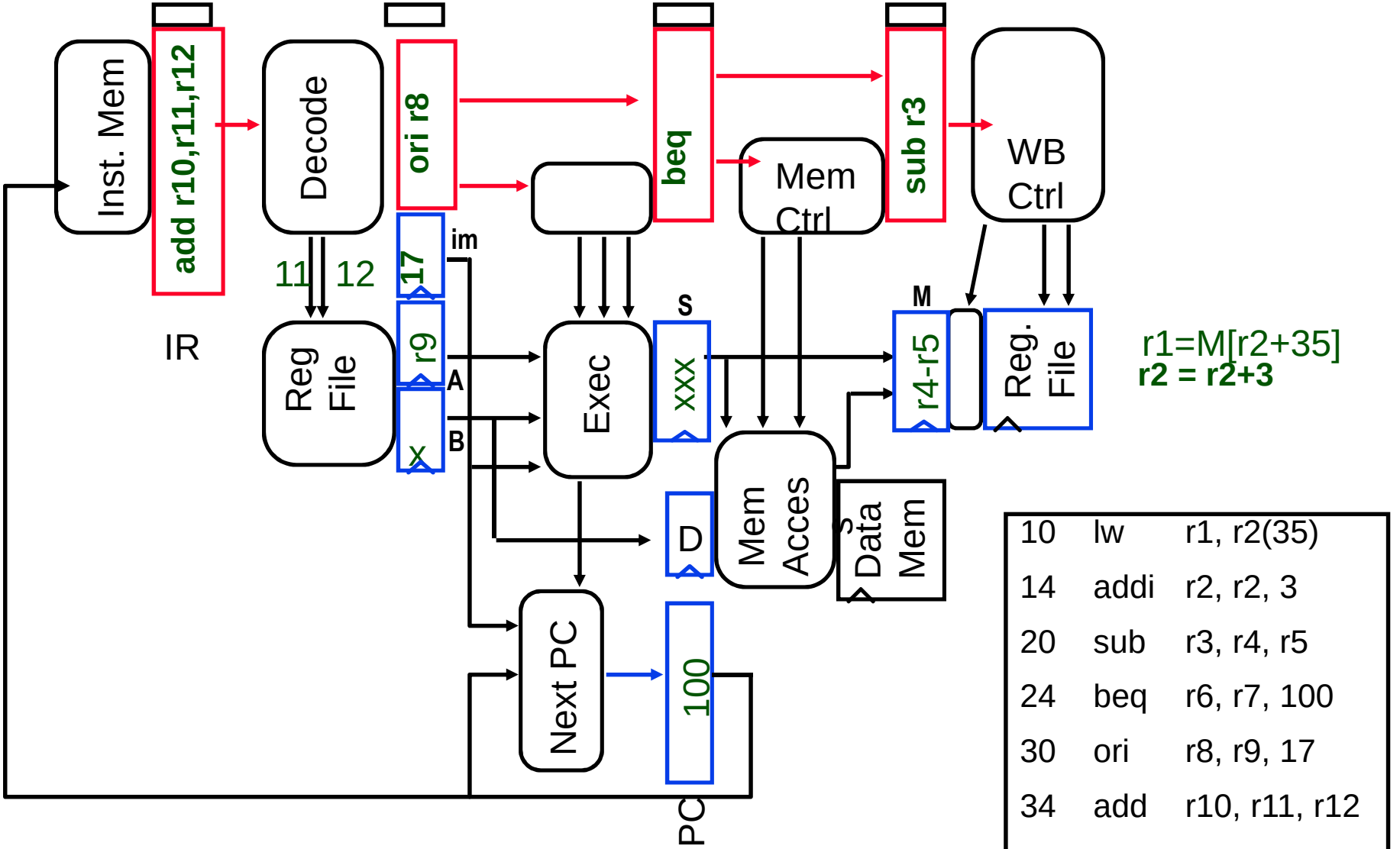
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, r15**

Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14



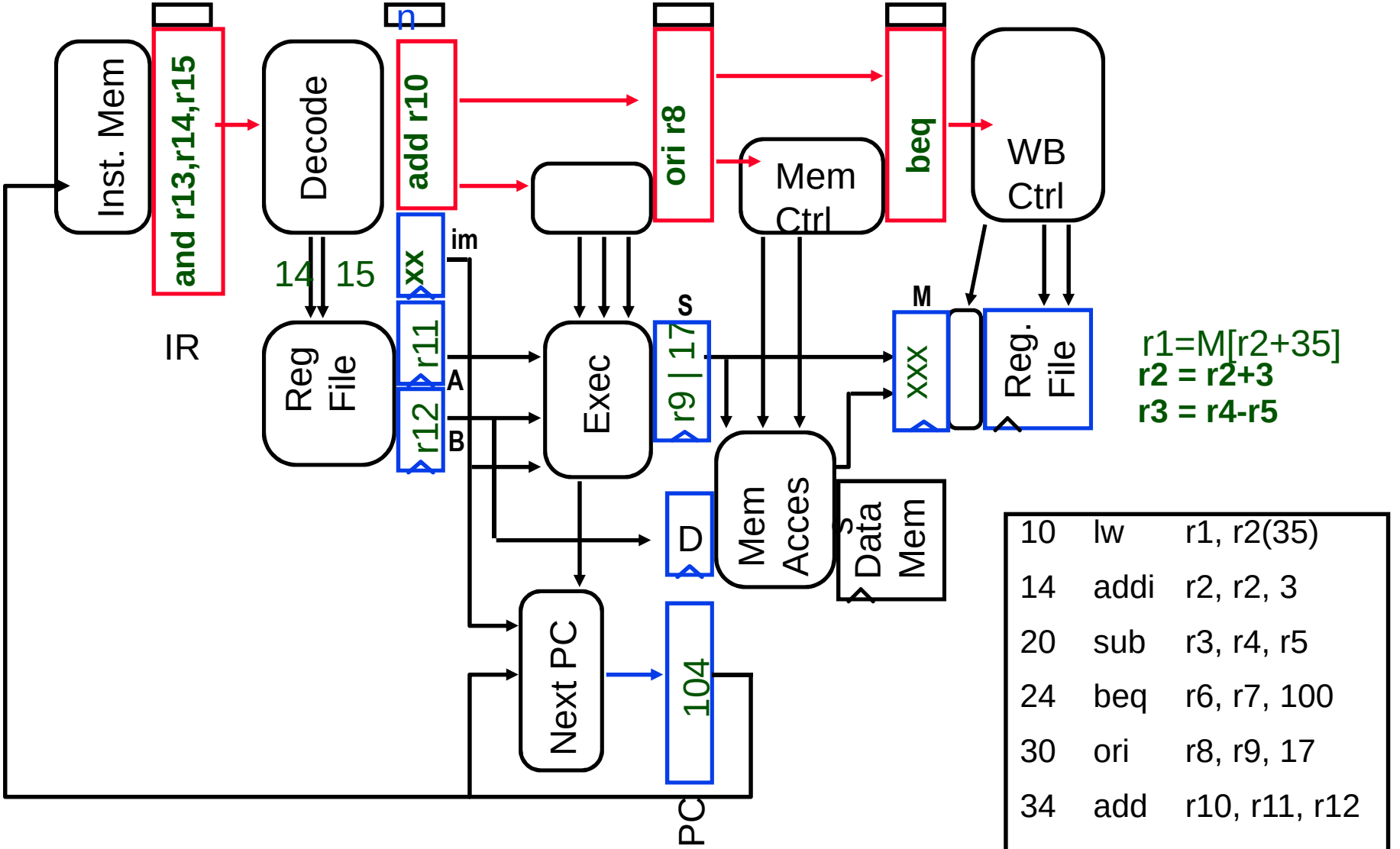
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, r15**

Fetch 100, Dcd 34, Ex 30, Mem 24, WB 20



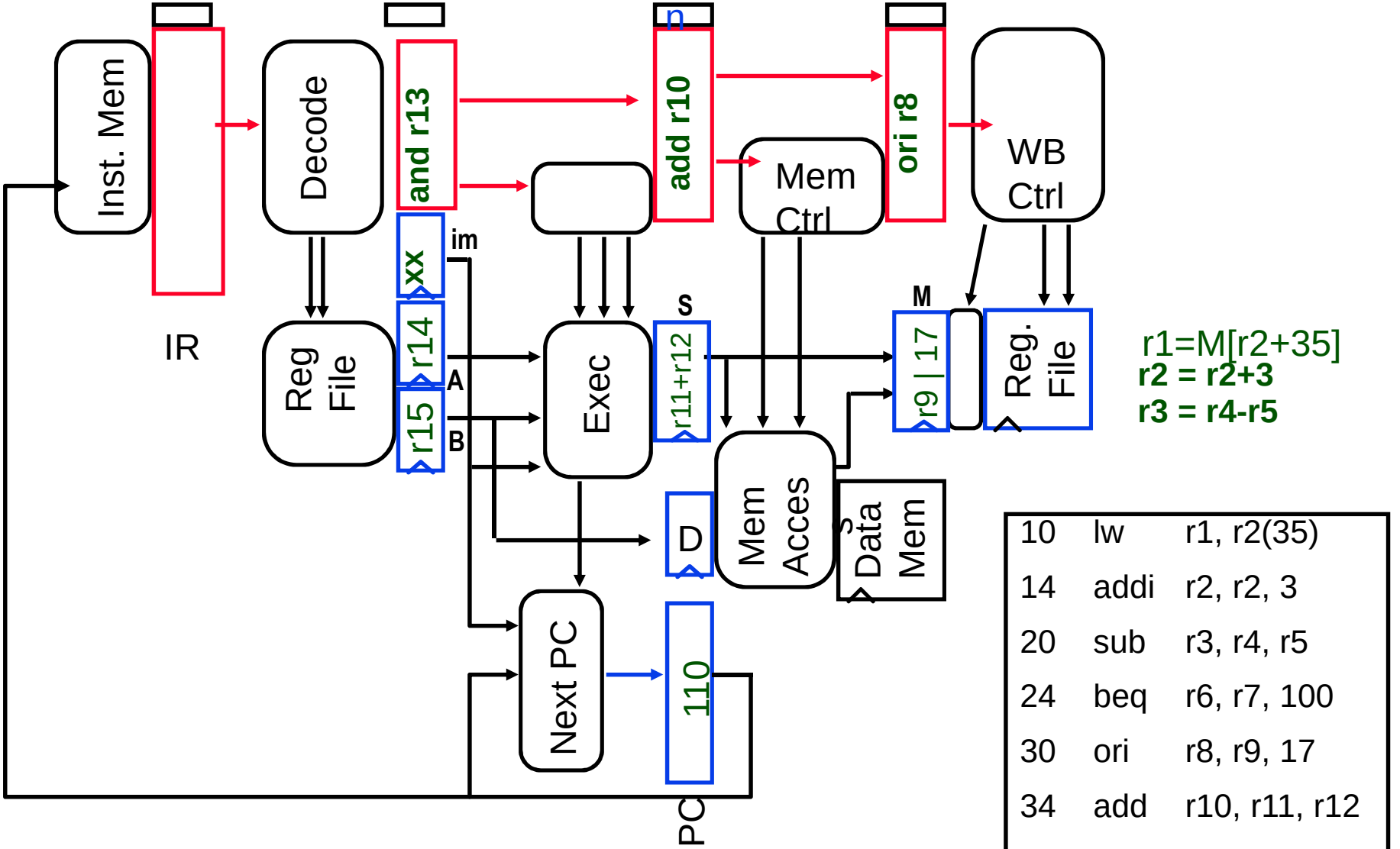
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, r15**

Fetch 104, Dcd 100, Ex 34, Mem 30, WB 24



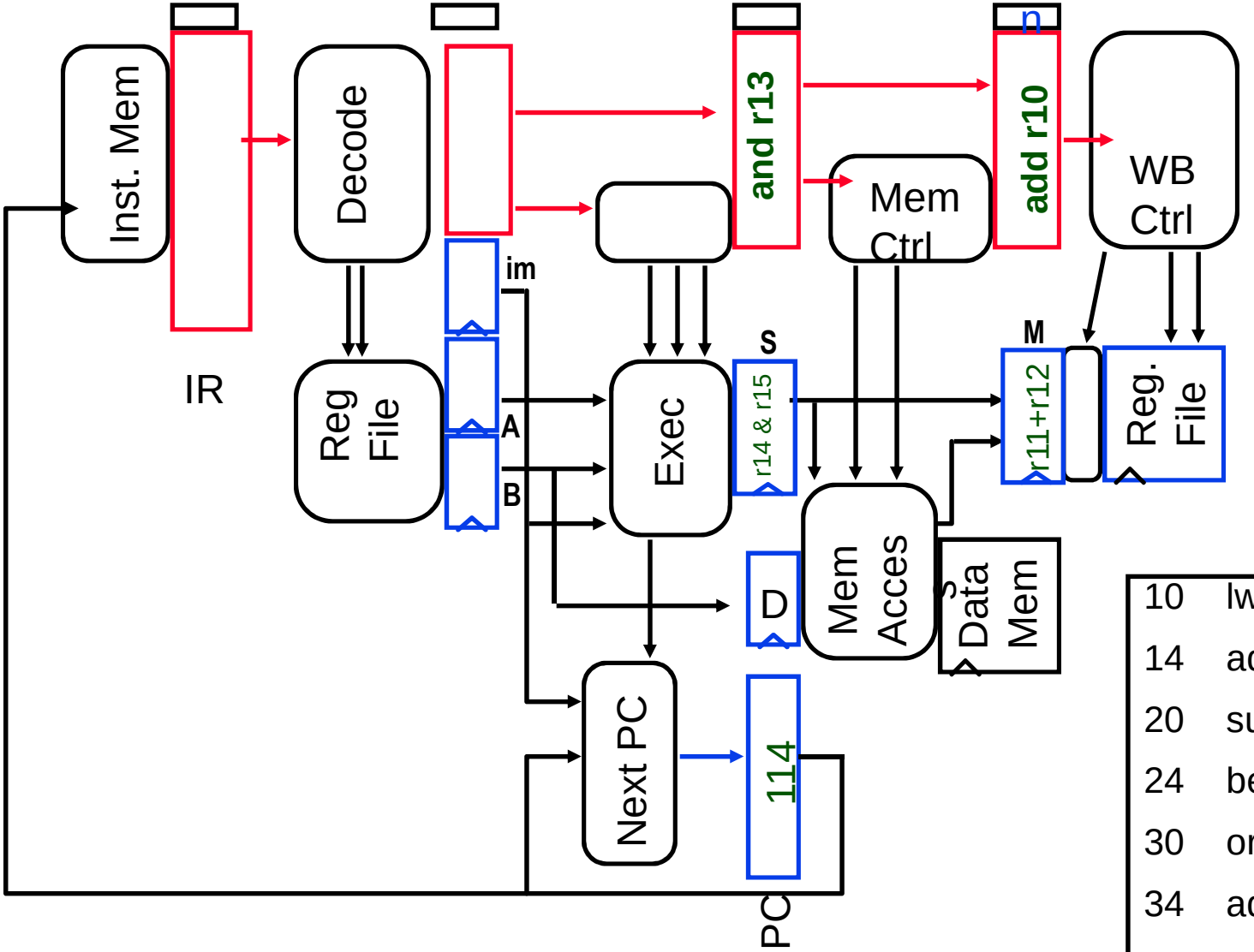
10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, r15**

Fetch 108, Dcd 104, Ex 100, Mem 34, WB 30



10	lw	r1, r2(35)
14	addi	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, r15**

Fetch 114, Dcd 110, Ex 104, Mem 100, WB 34



NO WB
NO Ovflow

r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

r8 = r9 | 17

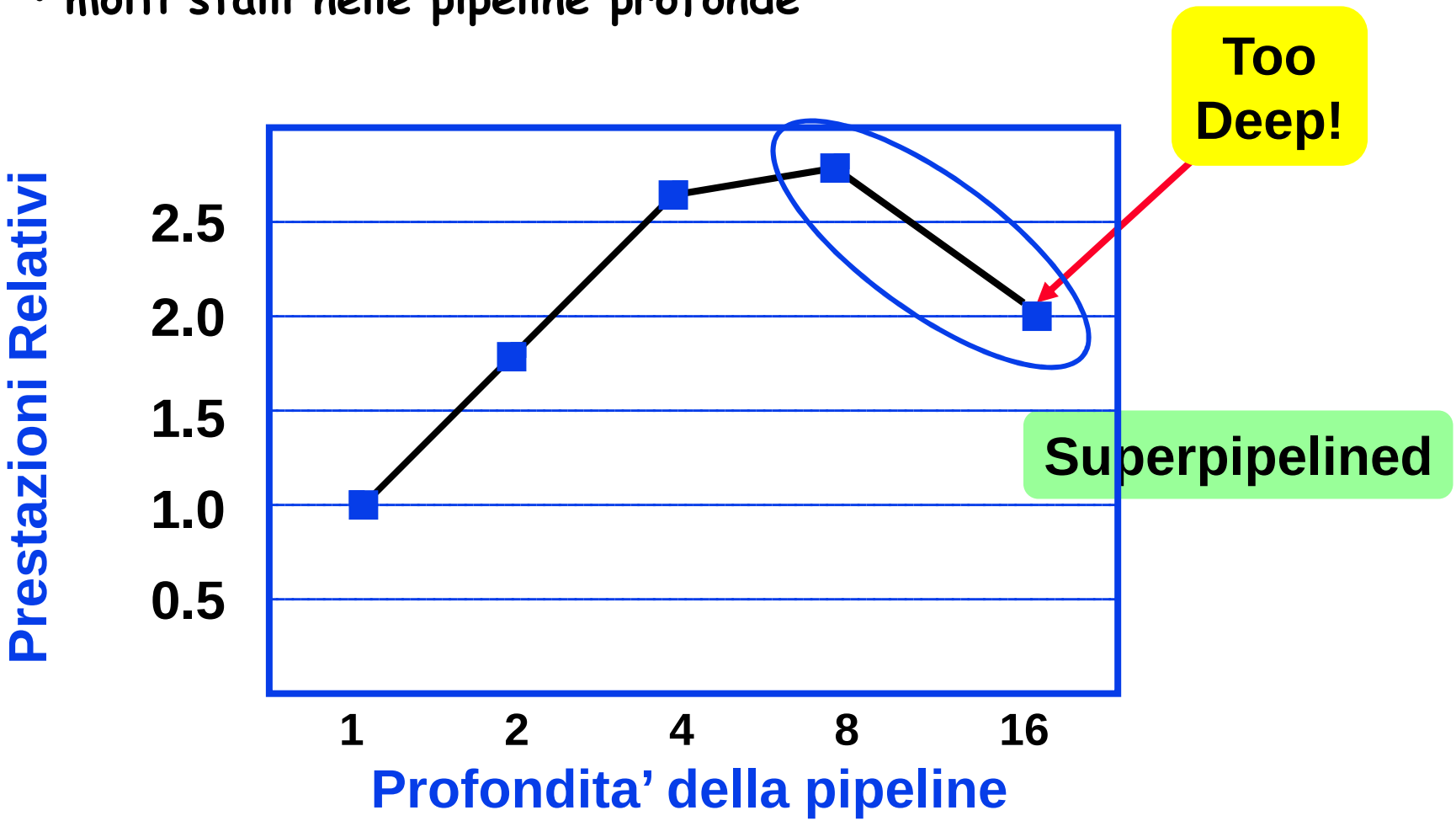
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, 15**

Il pipelining puo' provocare problemi?

- Si': Criticita' nelle pipeline
  - criticita' strutturali: tentativo di utilizzare le stesse risorse due volte allo stesso tempo
    - Esempio: utilizzo di un'unica macchina lavatrice/asciugatrice anziche' due macchine separate, oppure risorsa impegnata in qualche altro compito
  - criticita' sui dati: tentativo di utilizzare un dato prima che sia prodotto
    - Esempio: cerco di effettuare la stiratura prima di avere effettuato l'asciugatura
  - criticita' sul controllo: tentativo di saltare pur avendo caricato istruzioni del ramo non-preso
    - Esempio: cerco di fare l'asciugatura a 100 gradi invece che a 30 gradi perche' non ho verificato che tipo di capo sto asciugando
- Le criticita' si possono sempre risolvere aspettando, il controllo della pipeline deve:
  - 1) rilevare la criticita'
  - 2) gestirla (o rimandare l'operazione) per risolverla

Profondita' delle pipeline

- La pipeline non puo' essere troppo profonda
- Criticita' sono frequenti
  - molti stalli nelle pipeline profonde



Pipeline nei processori CISC (es. x86)

- Praticamente tutti i processori oggi usano la pipeline (anche i processori x86)
  - Sovrapposizione degli stadi F, X
- Ostacoli ad un buon funzionamento della pipeline
  - lunghezza variabile delle istruzioni
  - tempo di esecuzione variabile (numero dei cicli di microcodice)
    - sbilanciamento della pipeline
    - "backup" degli stadi di pipe
    - difficolta' del rilevamento delle criticita'
  - modi di indirizzamento complessi
    - auto-increment aggiorna address register
    - numerosi accessi alla memoria richiesti
- *Un flusso senza stalli nella pipeline e' molto difficile!*

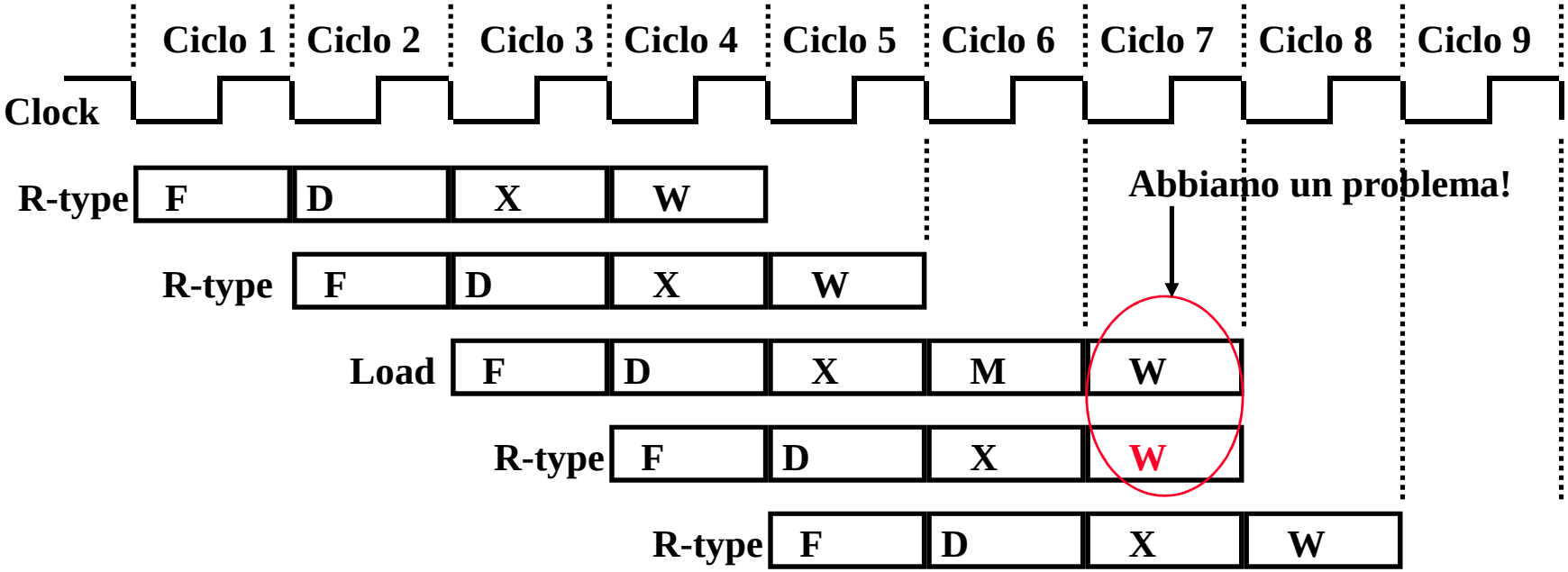
Ricapitolando

- Il pipelining e' un concetto fondamentale
  - Si cerca di non lasciare inutilizzate risorse distinte, schedulando oppurtunamente le operazioni ad ogni passo

- Per un corretto funzionamento occorre
  - limitare la lunghezza dello stadio piu' lungo
  - rilevare e risolvere criticita'

Criticita' di tipo strutturale

- Supponiamo di avere istruzioni di tipo R a 4 cicli
- Cio' puo' comportare la seguente situazione:

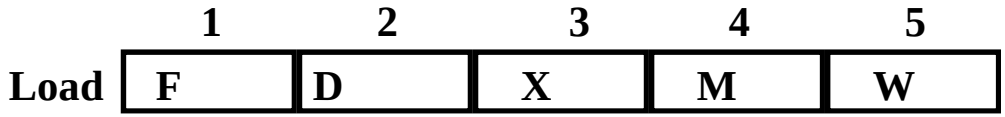


- Due istruzioni stanno tentando di scrivere nei registri contemporaneamente!
  - La porta di scrittura e' unica

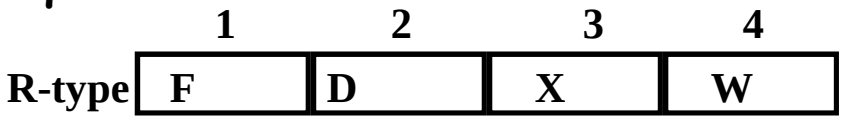
→Abbiamo una criticita' strutturale

Osservazione importante

- Ogni unita' di elaborazione puo' essere utilizzata una volta sola per istruzione
- Ogni unita' di elaborazione deve essere utilizzata nello stesso stadio per tutte le instruzioni:
  - La Load utilizza la porta di scrittura del Register File nel suo quinto stadio

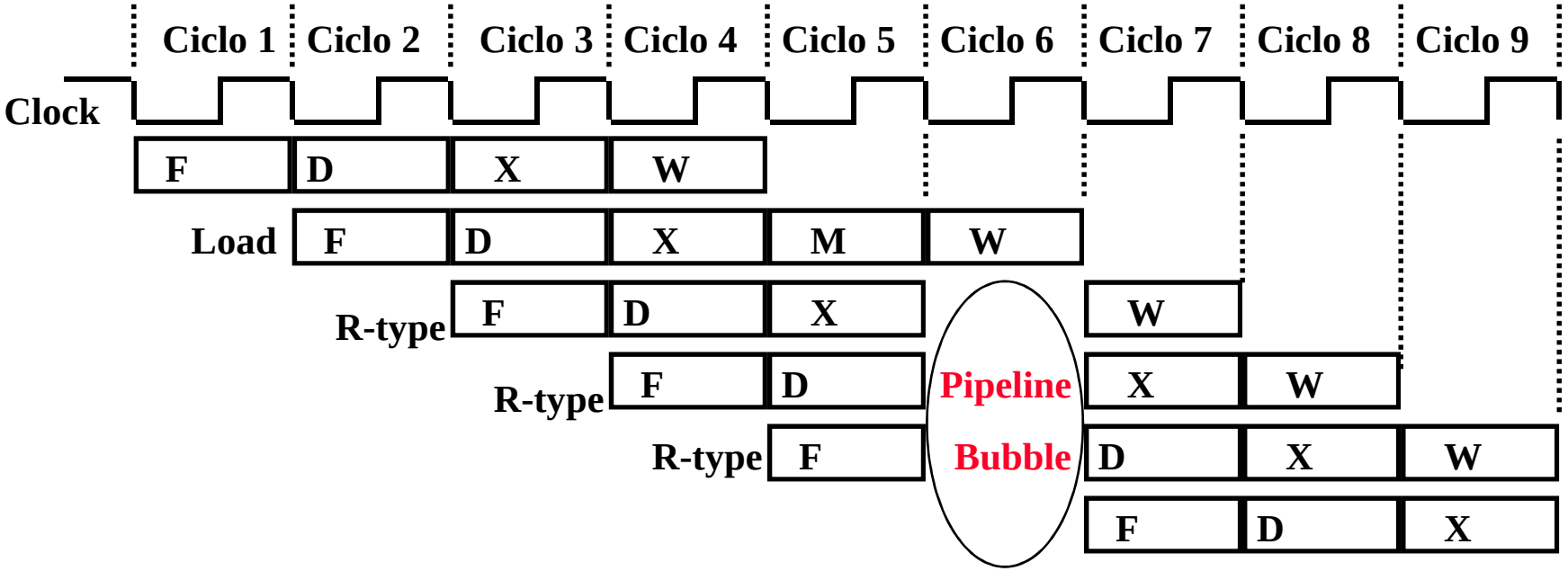


  - L'istruzione R-type utilizza la porta di scrittura del Register File nel suo quarto stadio



- *Esistono due modi per risolvere questa criticita' della pipeline.*

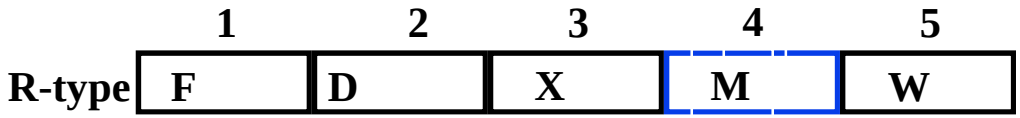
Soluzione 1: Inserire una "bolla" nella pipeline

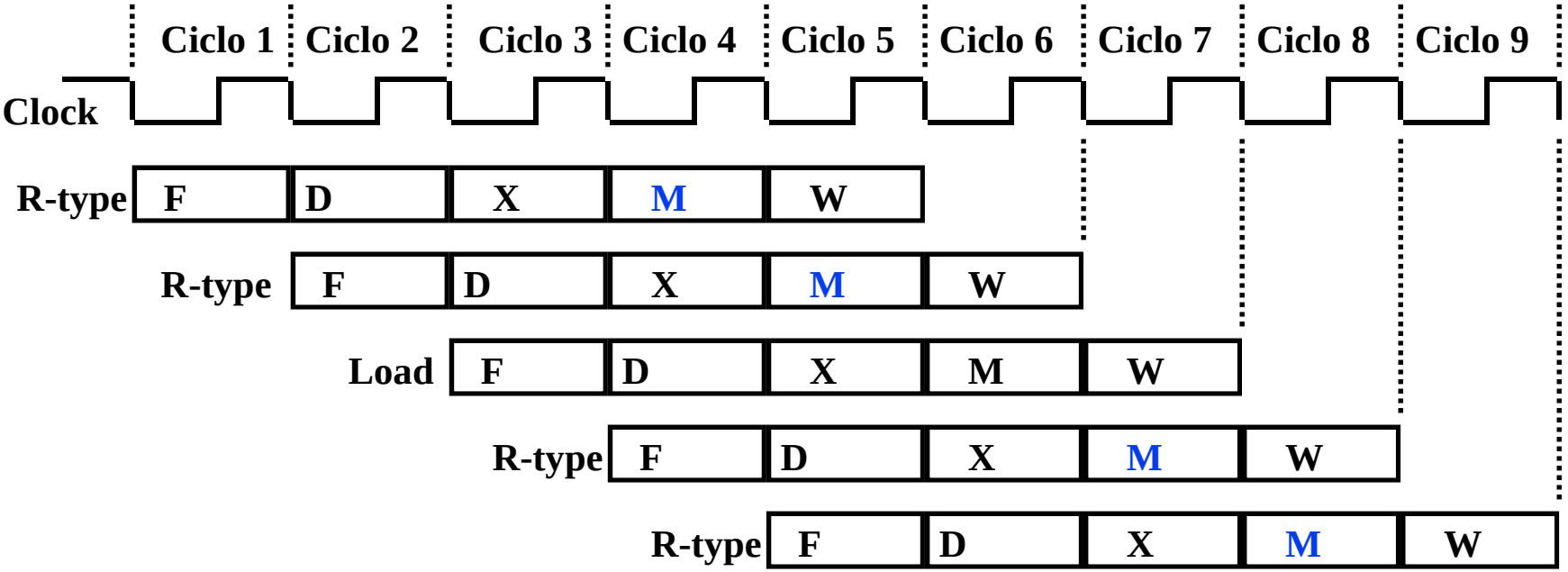


- Inserire una "bolla" nella pipeline per impedire due scritture in stesso ciclo
  - La logica di controllo puo' essere complessa.
- Nessuna instruzione comincia nel ciclo 6!

Soluzione 2: Rimandare la scrittura del registro per un ciclo (istruzioni di tipo R)

- Ora le istruzioni di tipo R possono utilizzare la porta di scrittura dei registri nello stadio 5
- Lo stadio Mem non fa niente





Criticita' sui dati

- criticita' sui dati: tentativo di utilizzare un'unita' prima che sia pronta
- un'istruzione dipende dal risultato di un'istruzione precedente, anch'essa in pipeline

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```

Data Hazard on r1

Classificazione di criticita' sui dati

- RAW
  - Un'istruzione successiva cerca di LEGGERE un registro prima che l'istruzione precedente lo abbia scritto
- WAR
  - Un'istruzione successiva cerca di SCRIVERE un registro prima che l'istruzione precedente lo abbia letto
- WAW
  - Un'istruzione cerca di SCRIVERE un registro prima che l'istruzione precedente lo abbia scritto (quando questo avviene, il valore non sara' piu' associato all'istruzione piu' recente)

Criticita' sui dati - classificazione

- Read after Write (RAW)
  - L'istruzione 1 deve scrivere *prima che* l'instruzione 2 legga
- Write after Read (WAR)
  - L'istruzione 1 legge
    L'istruzione 2 scrive (riscrive)
  - L'istruzione 2 *non deve scrivere prima che la* 1 legga
- Write after Write (WAW)
  - Entrambe le istruzioni 1 e 2 scrivono
    L'istruzione 2 deve scrivere *dopo* la 1

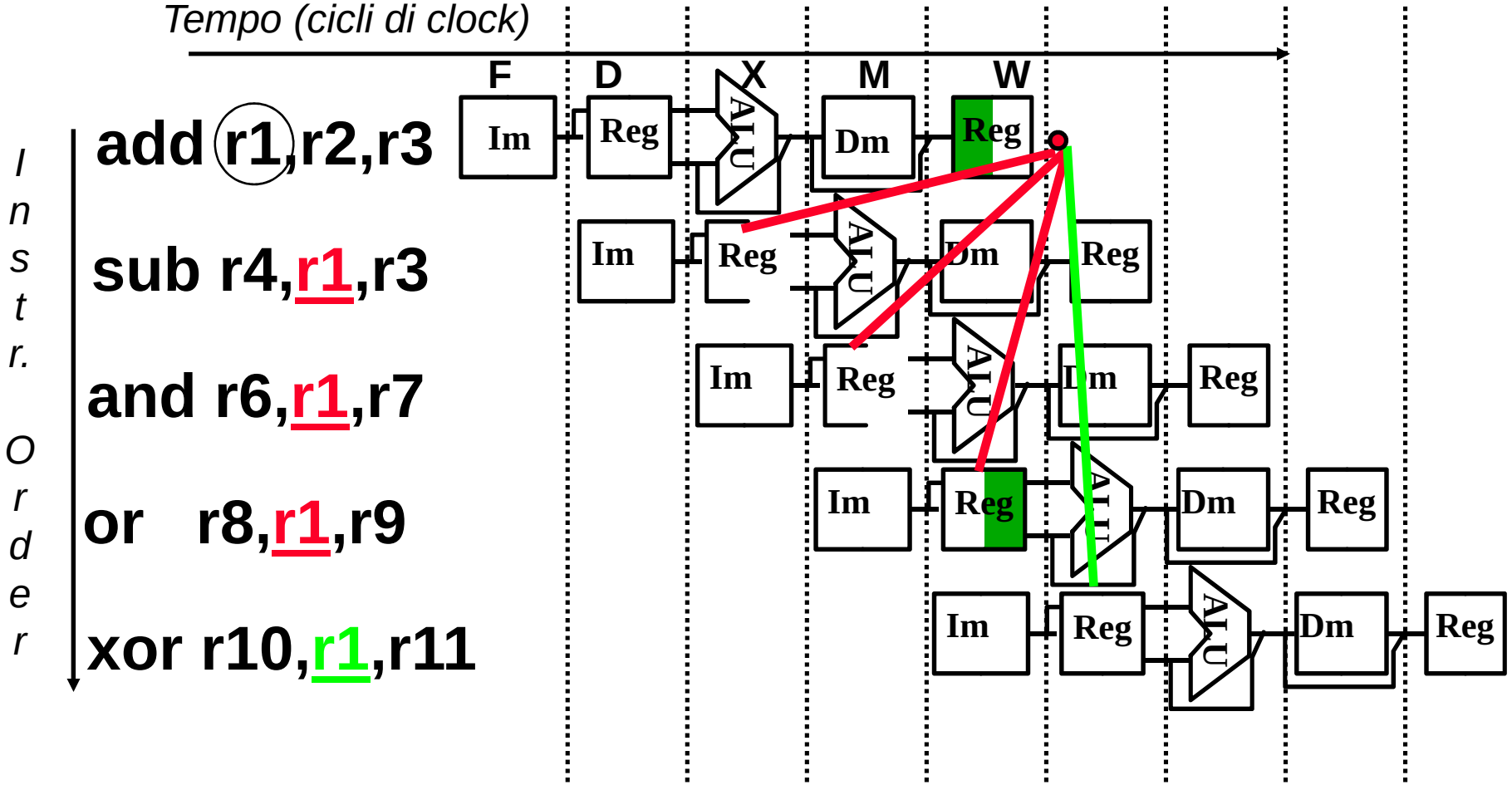
La pipeline deve considerare tutti e tre i casi!

Riassumendo

- Criticita' sui dati
  - RAW – i piu' comuni
  - WAW
  - WAR
- Il compilatore cerca le dipendenze funzionali
  - poi riordina il codice
- Hardware
  - Rete di controllo delle criticita'
    - Sorveglia le dipendenze
  - Rete di propagazione del valore
    - propaga i risultati dallo stadio X o Ma stadi precedenti

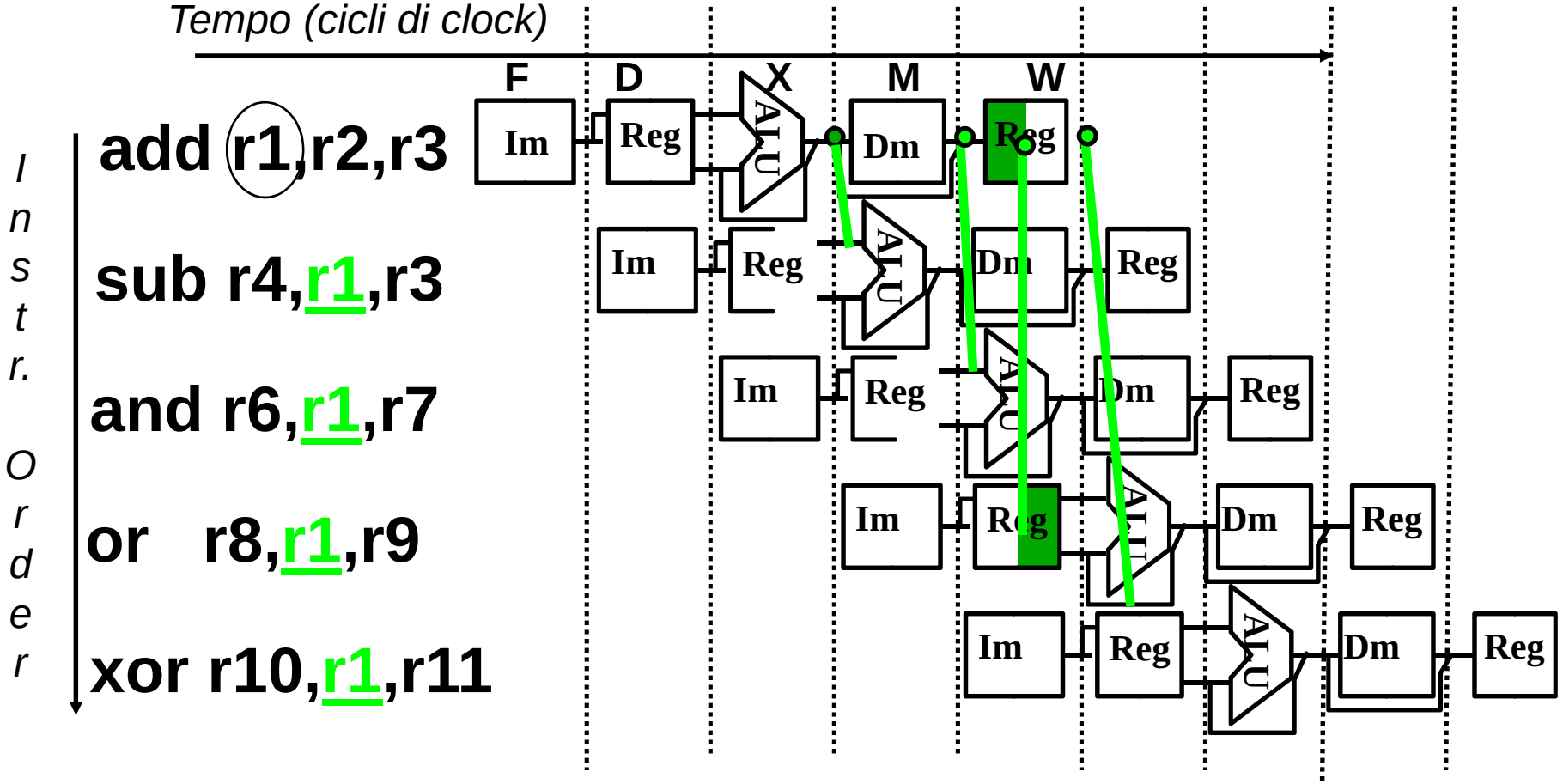
Data Hazard su r1:

- Le dipendenze "indietro nel tempo" sono criticita' di tipo RAW



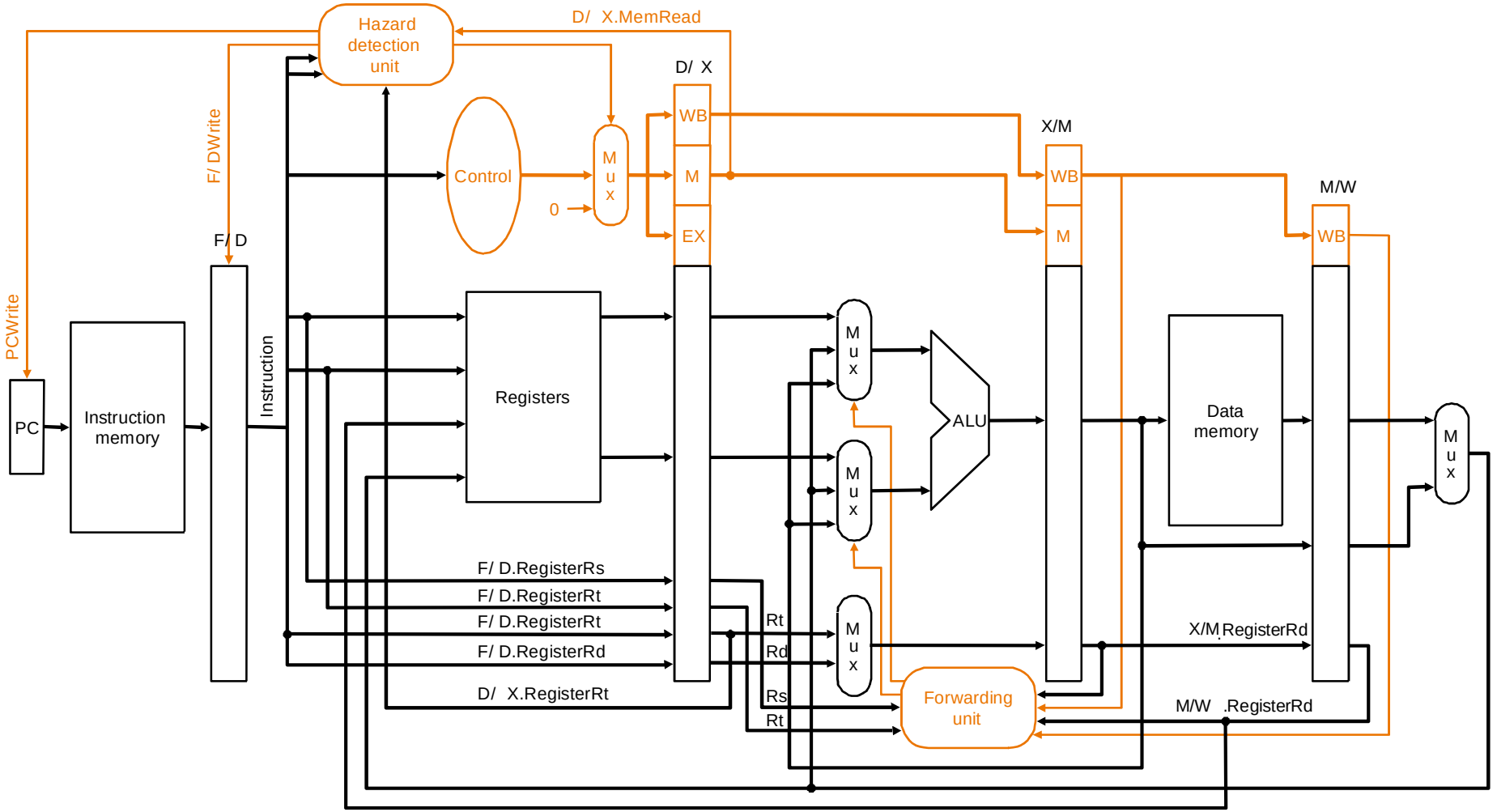
Soluzione per la criticita' sui dati

- "Propagare" il risultato da uno stadio all'altro ("forwarding")



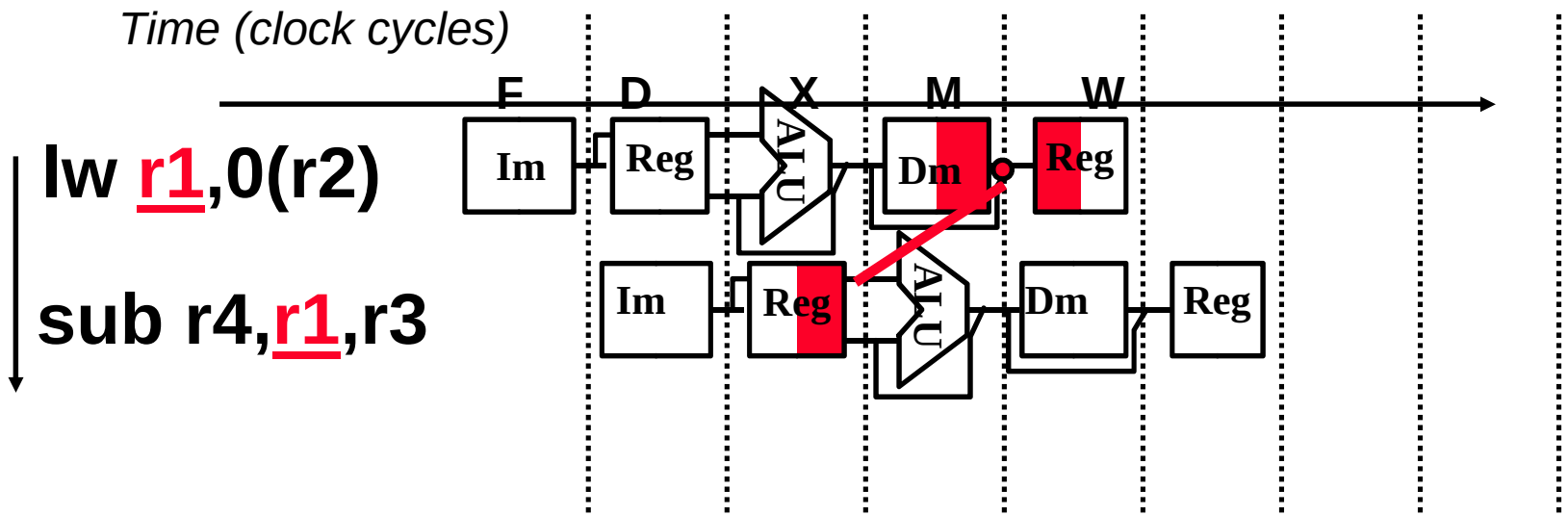
"or" OK se definisco read/write in un modo appropriato

Rete per la propagazione e rete di rilevamento criticita'



Cosa si fa con criticita' su load (detta "load/use")

- Le dipendenze "indietro nel tempo" sono criticita'



- Non si puo' risolvere con propagazione
- Le istruzioni che dipendono da una load devono essere ritardate/mandate in stallo ad hardware
- Questo si chiama interlock

Criticita' sul controllo: Branching (1/6)

- Supponiamo che la logica di decisione dei salti sia nello stadio di ALU
  - Allora due istruzioni in piu' dopo il salto saranno sempre prelevate
- Funzionalita' desiderata del salto
  - se il salto non viene effettuato, non perdere tempo e continuare con la normale esecuzione
  - se il salto viene effettuato, non eseguire istruzioni dopo il salto, ma saltare sulla etichetta desiderata

Criticita' sul controllo: Branching (2/6)

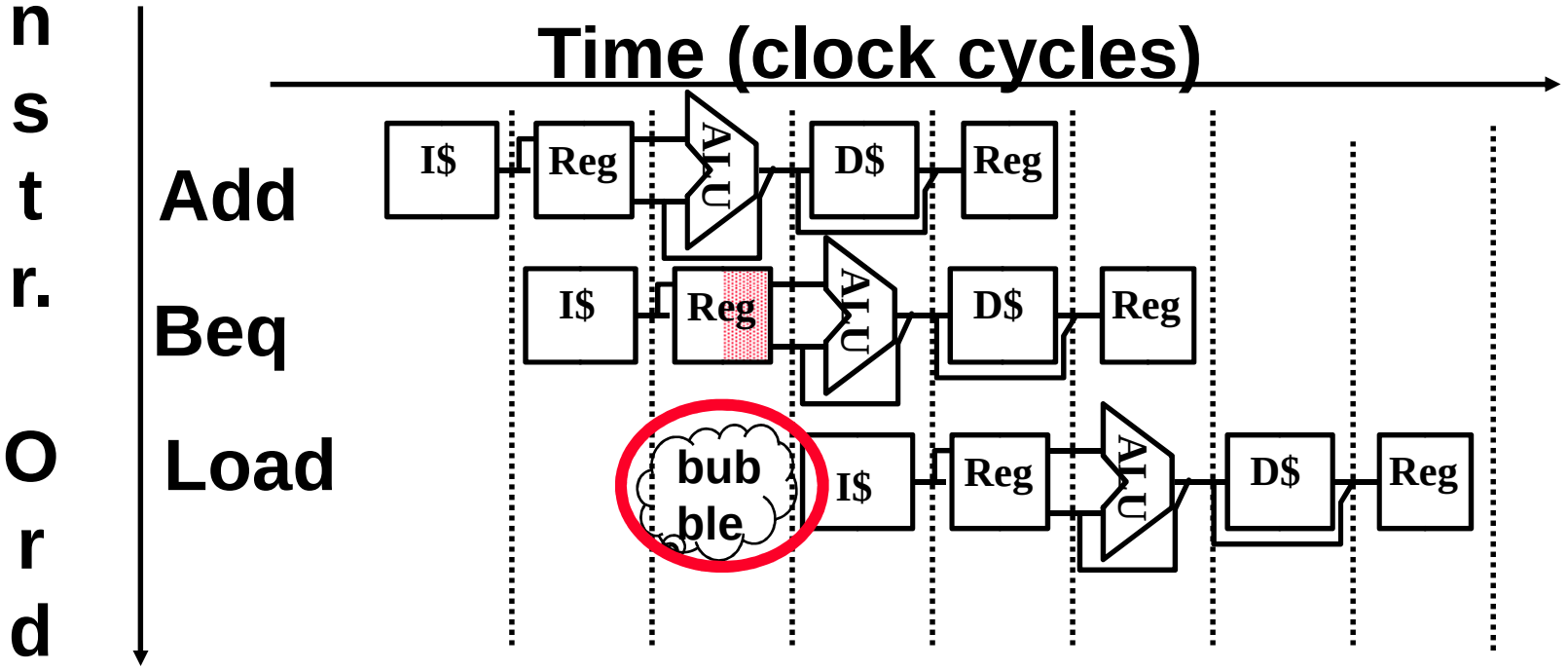
- Soluzione iniziale: sospendere l'esecuzione fino a quando non si prende la decisione
  - Inserire istruzioni "nop": non fanno nulla, solo consumano tempo
  - Svantaggio: ogni salto consuma 3 cicli (anziche' 1)
    (a patto che il comparatore sia stato messo nello stadio di ALU)

Criticita' sul controllo: Branching (3/6)

- Ottimizzazione #1:
  - Spostare il comparatore (e calcolo branch target) nello stadio 2
  - Non appena l'istruzione viene decodificata (opcode identifica un salto), prendere la decisione e impostare il valore di PC (se necessario)
  - Beneficio: poiche' il salto e' completo nello stadio 2, solo una
  - istruzione del ramo sbagliato viene prelevata, e solo una nop e' richiesta
  - Nota: in questo caso, i salti non fanno nulla negli stadi 3, 4 e 5

Criticita' sul controllo: Branching (4/6)

- Inserire una singola nop (bolla o "bubble")



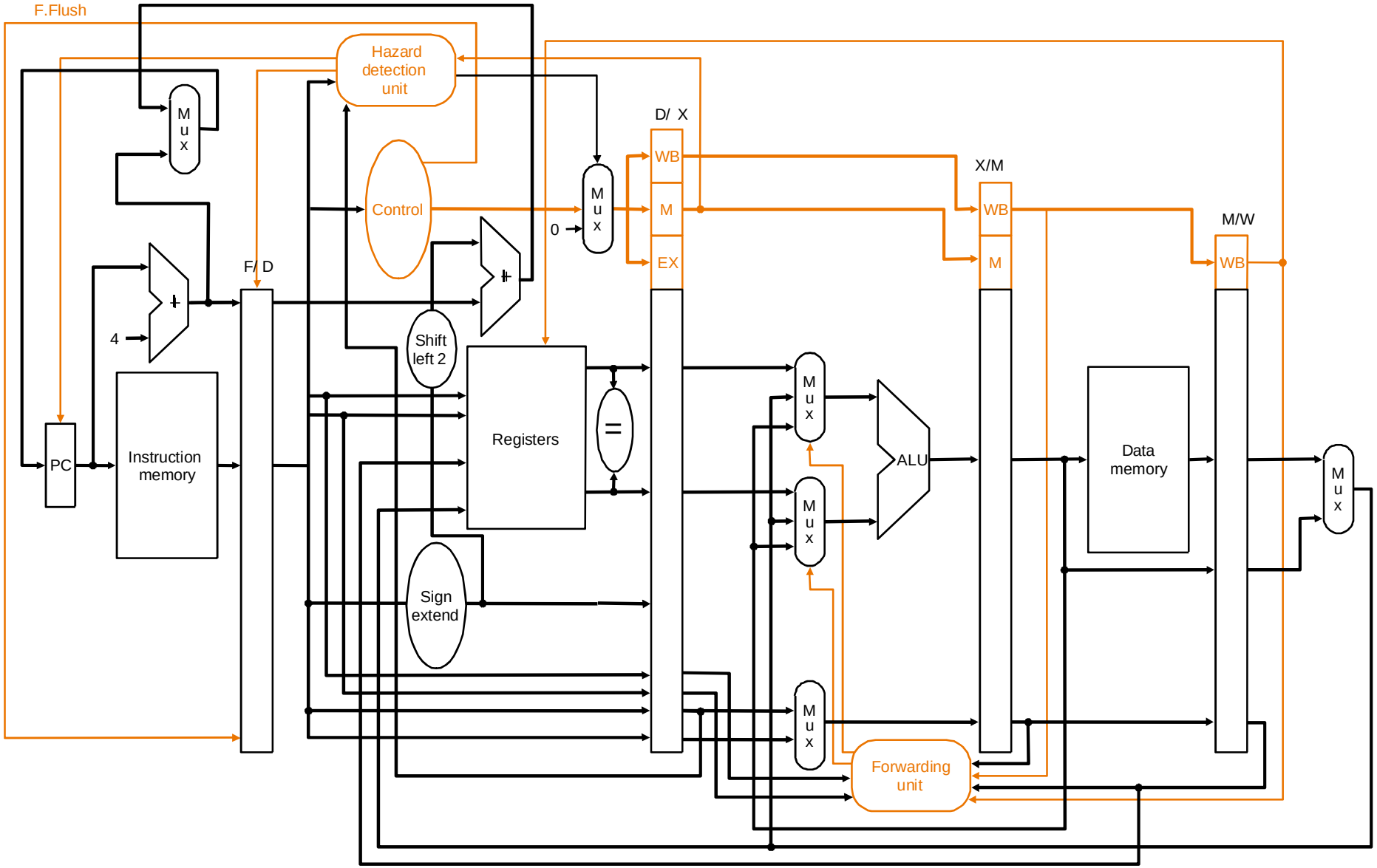
- Impatto: 2 cicli di clock per istruzione di salto $\Rightarrow$ lento

Propagazione e spostamento di decisione del salto

- La propagazione/bypassing attualmente influisce sullo stadio di esecuzione:
  - Invece di utilizzare il valore letto nei registri nello stadio di Decode, utilizzare il valore risultante dalla ALU o dalla memoria
- Lo spostamento della decisione di salto dallo stadio di esecuzione allo stadio di decodifica significa che la propagazione/bypassing deve arrivare fino allo stadio di decodifica
- Cioe', il seguente codice deve comunque funzionare:

```
addiu $s1, $s1, -4
beq   $s1, $s2, Exit
```

Rete di propagazione con branch anticipato nello stadio D



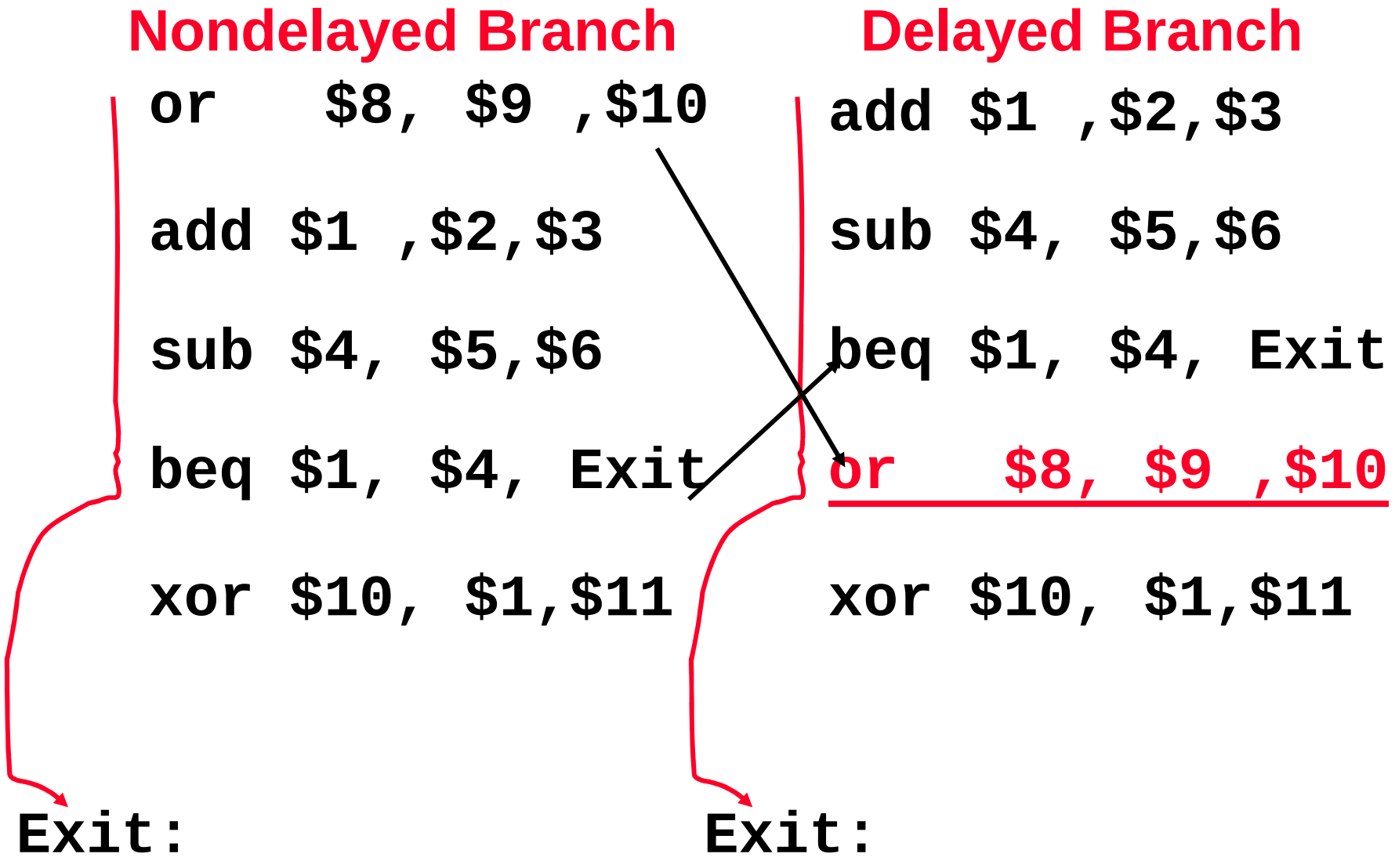
Criticita' sul controllo: Branching (5/6)

- Ottimizzazione #2: Ridefinire i salti
  - Definizione vecchia: se il salto viene effettuato, nessuna istruzione dopo il salto viene eseguita
  - Definizione nuova: sia che si faccia il salto, sia che non si faccia, viene eseguita l'istruzione che segue immediatamente il salto (detto branch-delay slot)

Criticita' sul controllo: Branching (6/6)

- Osservazioni sul Branch-Delay Slot
  - Caso peggiore: si puo' sempre inserire una nop nel branch-delay slot
  - Caso migliore: si puo' trovare un'istruzione prima del salto che puo' essere messa nel branch-delay slot senza influenzare il flusso di programma
    - il riordinamento delle istruzioni e' un metodo comunemente utilizzato per velocizzare i programmi
    - il compilatore deve essere molto furbo per trovare le istruzioni appropriate
    - tipicamente i compilatori sono in grado di riempire circa il 50% di tali posizioni

Esempio: Nondelayed vs. Delayed Branch



Quanto dura l'esecuzione?

- Assumiamo: delayed branch, pipeline di 5 stadi, propagazione/bypassing,
  interlock sulle criticita' di load non risolte

```
Loop:   lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        addiu   $s1, $s1, -4
        bne     $s1, $zero, Loop
        nop
```

- Quanti cicli di clock sono necessari per eseguire questo codice per un' iterazione di loop?

  a)<= 5 b) 6 c) 7 d) 8 e) >=9

  (dopo 1000 iterazioni, quando la pipeline e' piena)

Quanto dura l'esecuzione?

- Assumere delayed branch, la pipeline di 5 stadi, propagazione/bypassing, interlock sulle criticita' di load non risolte
- Diamo un'occhiata a questo codice:

```
Loop:   1.  lw      $t0, 0($s1)
2. (data hazard so stall)  addiu   $t0, $t0, $s2
        3.  sw      $t0, 0($s1)
        4.  addiu   $s1, $s1, -4
        5.  bne     $s1, $zero, Loop
        6.  nop     (delayed branch so execute nop)
        7.
```

- Quanti cicli di clock sono necessari per eseguire questo codice per un' iterazione di loop?

a)<= 5 b) 6 c) 7 d) 8 e) >=9

Riscrivere il ciclo per migliorare le prestazioni

- Riscrivere questo codice per ridurre al minimo i cicli di clock del loop :

```
Loop:   lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        addiu   $s1, $s1, -4
        bne     $s1, $zero, Loop
        nop
```

- Quanti cicli di clock sono necessari per eseguire il codice riscritto per un' iterazione di loop?

  a) 4 b) 5 c) 6 d) 7

Riscrivere il ciclo per migliorare le prestazioni

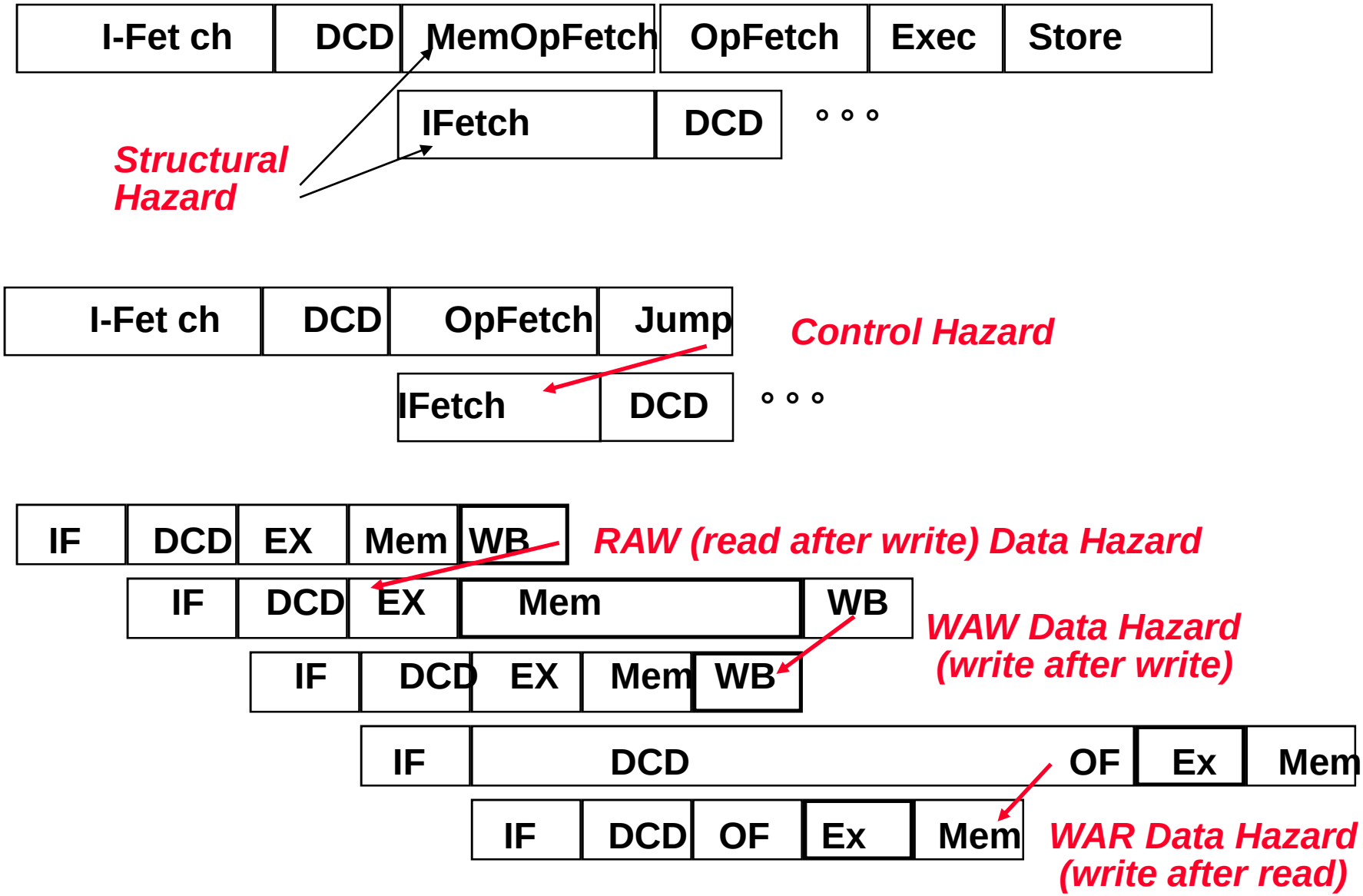
- Riscrivere questo codice per ridurre il numero dei cicli di clock al minimo per un' iterazione del loop :

```
Loop:   1.  lw      $t0, 0($s1)      (no hazard since extra cycle)
        2.  addiu   $s1, $s1, -4
        3.  addu    $t0, $t0, $s2
        4.  bne     $s1, $zero, Loop
        5.  sw      $t0, +4($s1)     (modified sw to put past addiu)
```

- Quanti cicli di clock sono necessari per eseguire il codice riscritto per un' iterazione di loop?

  a) 4 b) 5 c) 6 d) 7

Ricapitolando: Le criticita' nelle pipeline

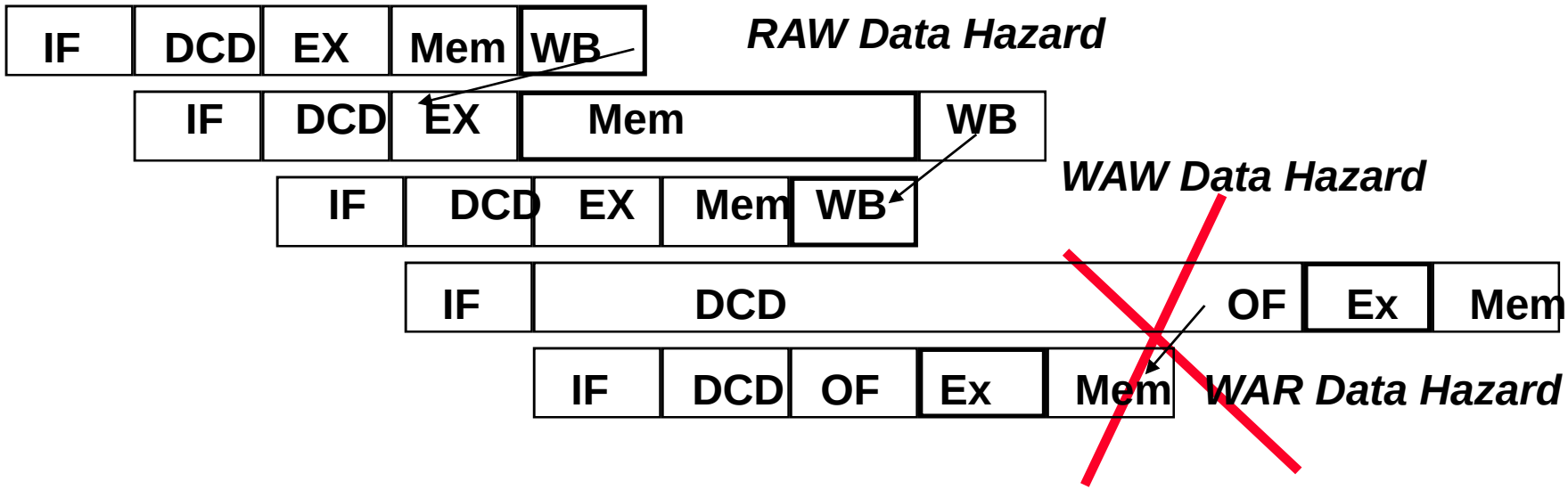


Punti principali del pipelining

- Le pipeline fanno passare informazioni di controllo al loro interno nello stesso modo in cui fanno passare i dati
- Propagazione e stalli sono gestiti dal controllo locale
- Le criticita' limitano le prestazioni
  - Strutturali: richiedono piu' risorse HW
  - Dati: richiedono propagazione, schedulazione dal compilatore
  - Controllo: valutazione preliminare, delayed branch, predizione
- Aumentando la lunghezza della pipe aumenta l'impatto delle criticita'; pipelining migliora il throughput delle istruzioni, e non la loro latenza
- Interruzioni, Instruction Set, FP rendono il pipelining piu' difficile
- I compilatori riducono il costo delle criticita' sui dati e sul controllo
  - Load delay slots
  - Branch delay slots
  - Branch prediction

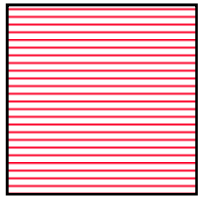
Ricapitolando: Criticita' sui dati

- Evitare alcune durante la progettazione
  - eliminare WAR prelevando presto gli operandi (DCD) in pipe
  - eleminare WAW eseguendo tutti i WB in ordine (nell'ultimo stadio)
- Rivelare e risolvere quelli rimasti
  - Mandare in stallo o propagare (se possibile)



Rivelamento delle criticita'

- Supponiamo che l'istruzione *j* stia per essere elaborata e l'istruzione precedente *i* si trovi ancora nella pipeline.
- Una criticita' RAW esiste su registro ρ se ρ ∈ Wregs(*i*) ∩ Rregs(*j*)
  - Tenere traccia delle scritture in attesa (per le istruzioni nella pipe) e confrontare con i registri di operandi della istruzione corrente.
  - Quando l'istruzione viene elaborata, prenotare il suo registro del risultato.
  - Quando l'istruzione e' completa, rimuovere la sua prenotazione di scrittura.



- Una criticita' WAW esiste su registro ρ se ρ ∈ Wregs(*i*) ∩ Wregs(*j*)
- Una criticita' WAR esiste su registro ρ se ρ ∈ Rregs(*i*) ∩ Wregs(*j*)