
Esercitazione su Instruction Level Parallelism

Salvatore Orlando

Pipeline con e senza forwarding

- Si considerino due processori MIPS (processore A e B)
 - entrambi con pipeline a 5 stadi
 - il processore A adotta il forwarding, e quello B no
 - entrambi scrivono sul register file durante la prima parte del ciclo di clock, e leggono durante la seconda parte dello stesso ciclo
 - entrambi sono dotati di hazard detection unit per bloccare la pipeline
- Rispetto alla seguente porzione di codice

```
add $5, $6, $7
and $2, $5, $8
lw $3, a($0)
or $2, $3, $4
sub $10, $11, $12
```

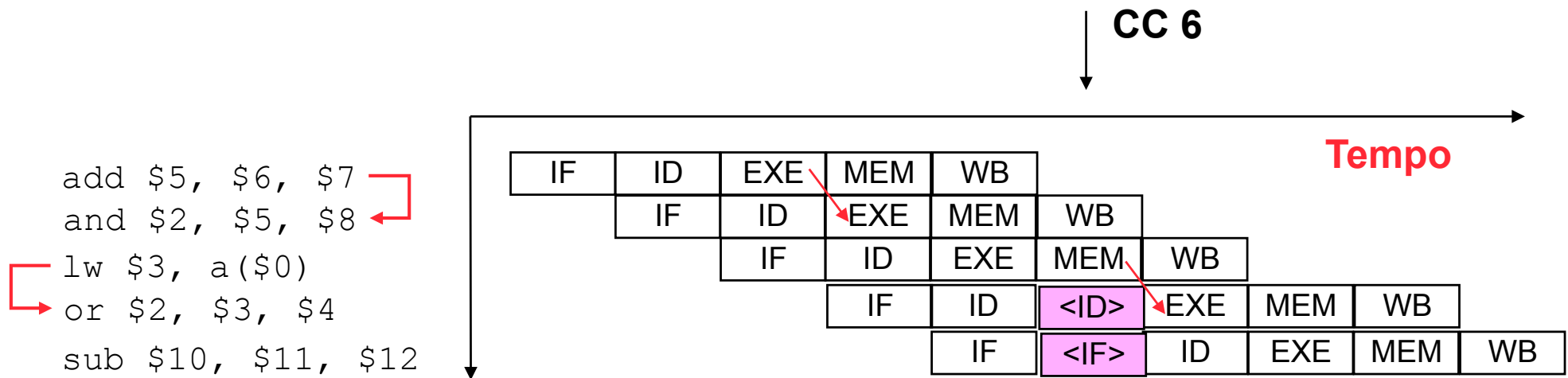
determinare

- quali sono le dipendenza RAW
 - i relativi diagrammi temporali di esecuzione, indicando con una freccia gli eventuali forwarding
- Cosa succede al ciclo 6 di clock rispetto nei due processori?

Pipeline con forwarding (A)

```

add $5, $6, $7
and $2, $5, $8
lw $3, a($0)
or $2, $3, $4
sub $10, $11, $12
    
```



- **Al ciclo 6:** IF in stallo (ripete `sub`), ID in stallo (ripete `or`), in EXE passa bolla (nop), MEM esegue `lw`, WB esegue `and`

Pipeline senza forwarding (B)

add \$5, \$6, \$7

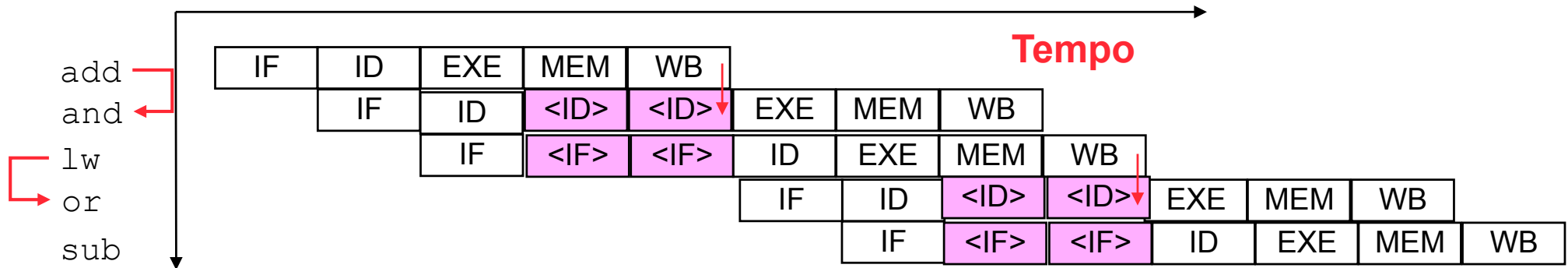
and \$2, \$5, \$8

lw \$3, a(\$0)

or \$2, \$3, \$4

sub \$10, \$11, \$12

CC 6



- **Al ciclo 6:** IF esegue or, ID esegue lw, EXE esegue and, in MEM passa bolla (nop), in WB passa bolla (nop)

Pipeline, CPI, e stalli di accesso alla memoria

- Considerando che $\$s3 = 1024$, $\$t0 = 0$, e $\$s0$ contiene un indirizzo di memoria, stabilire cosa calcola questa porzione di codice MIPS

Start:

```
add $t1, $s0, $t0
lw  $t2, 0($t1)
add $t2, $t2, $t3
sw  $t2, 0($t1)
addi $t0, $t0, 4
bne $t0, $s3, start
```

- Il loop viene eseguito per $1024/4 = 256$ volte
- Il programma incrementa di $\$t3$ tutti gli elementi di un vettore di 256 interi (puntato da $\$s0$)

Pipeline, CPI, e stalli di accesso alla memoria

- Determinare le dipendenze sui dati tra le istruzioni del loop

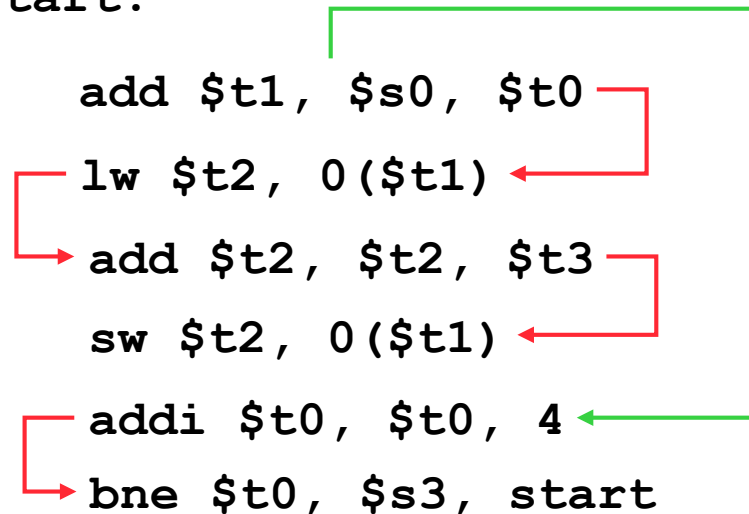
Start:

```
add $t1, $s0, $t0
lw  $t2, 0($t1)
add $t2, $t2, $t3
sw  $t2, 0($t1)
addi $t0, $t0, 4
bne $t0, $s3, start
```

Pipeline, CPI, e stalli di accesso alla memoria

- Determinare le dipendenze sui dati tra le istruzioni del loop

Start:



Le dipendenze in **rosso** sono di tipo RAW

Le dipendenze in **verde** sono di tipo WAR

Pipeline, CPI, e stalli di accesso alla memoria

- Considerando che la pipeline a 5 stadi è fornita di forwarding, mentre la tecnica per risolvere le criticità sul controllo è il salto ritardato (delay slot = 1)
 - inserire il **nop** nel branch delay slot
 - inserire i **nop** per forzare gli stalli dovuti alle dipendenze sui dati
 - calcolare il CPI senza considerare i tempi di riempimento/svuotamento della pipeline

```
Start:
    add $t1, $s0, $t0
    lw  $t2, 0($t1)
    nop
    add $t2, $t2, $t3
    sw  $t2, 0($t1)
    addi $t0, $t0, 4
    bne $t0, $s3, start
    nop
```

- 8 cicli di clock per ogni ciclo del loop, costituito da 6 istruzioni
- loop eseguito per 256 volte
- $CPI = \text{num_cicli} / IC = (8 * 256) / (6 * 256) = 8/6 = 1.33$

Pipeline, CPI, e stalli di accesso alla memoria

- Spostare istruzioni significative al posto delle `nop`, ovvero nel **load delay slot**, e nel **branch delay slot**
 - per il **load delay slot**, scegliere un'istruzione indipendente e successiva nel corpo del loop
 - per il **branch delay slot**, scegliere un'istruzione indipendente e precedente nel corpo del loop

Start:

```
add $t1, $s0, $t0
```

```
lw $t2, 0($t1)
```

```
addi $t0, $t0, 4
```

```
add $t2, $t2, $t3
```

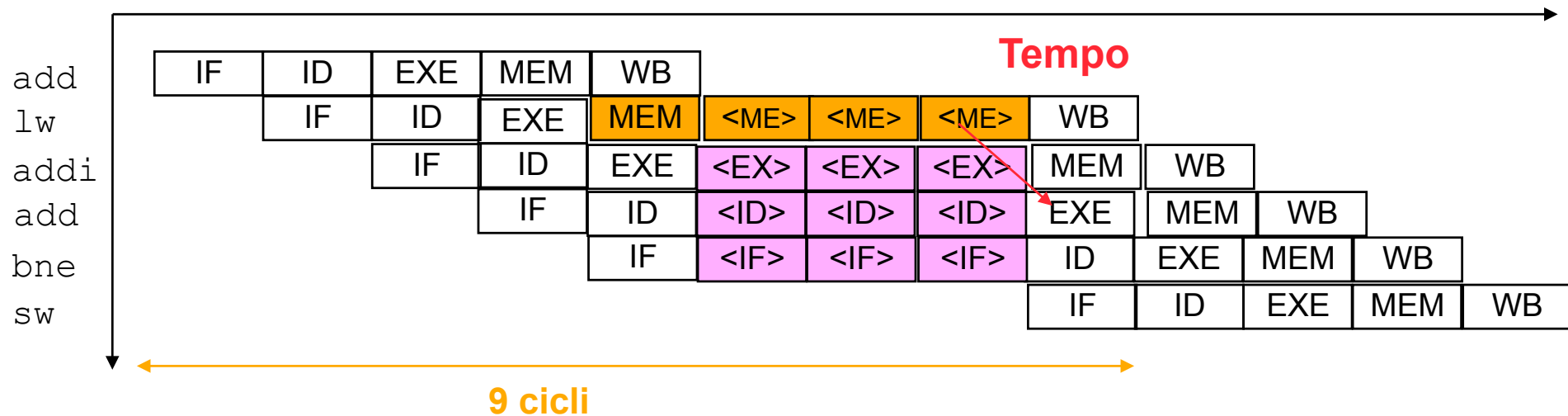
```
bne $t0, $s3, start
```

```
sw $t2, 0($t1)
```

- Qual è il numero di cicli totali per eseguire il loop ? E il nuovo CPI?
 - **Cicli totali = 6 * 256 = 1536** **CPI = 1**

Pipeline, CPI, e stalli di accesso alla memoria

- Abbiamo finora considerato una memoria dati il cui l'accesso costa 1 ciclo di clock (stadio MEM)
- Se in presenza di *miss* la pipeline andasse in stallo, poiché l'accesso costa 4 cicli, come cambierebbe il diagramma temporale di esecuzione di un ciclo del loop? (si noti che il *miss* si può avere solo sulla **lw**, mentre la successiva **sw** provoca un *hit*)



- Cicli totali per eseguire il loop in presenza di miss =
6 cicli normali + 3 cicli di stallo = 9 cicli

Pipeline, CPI, e stalli di accesso alla memoria

- Consideriamo una cache ad accesso diretto con blocchi da 16 B, che all'inizio del loop non contiene nessuna word del vettore acceduto
- Calcolare il CPI totale del loop precedente (ottimizzato)
- Soluzione:
 - Poiché i blocchi si accedono in sequenza in word da 4 B, abbiamo sempre un miss e 3 hit
 - Cicli totali per eseguire il loop in presenza di miss = 9
 - Cicli totali per eseguire il loop in presenza di hit = 6
 - Cicli totali per le 256 iterazioni = $0.25 * 256 * 9 + 0.75 * 256 * 6 = 64 * 9 + 192 * 6 = 1728$
 - $CPI = \text{cicli totali} / IC = 1728 / (256 * 6) = 1.125$
- La risposta sarebbe stata diversa se la cache avesse un'organizzazione differente rispetto all'associatività ?
- Soluzione:
 - La risposta è indipendente dal grado di associatività della cache ! Solo località spaziale, influenzata dalla dimensione del blocco. Solo miss certi (compulsory)

Loop unrolling

- Si assuma la semplice pipeline a 5 stadi, con delay branch, forwarding e hazard detection unit
- Si consideri il loop seguente:

loop:

```
lw $t0, 0($s1)
add $t0, $t0, $s2
sw $t0, 0($s1)
addi $s1, $s1, -4
bne $s1, $0, loop
```

- Senza considerare gli stalli dovuti alla memoria, e quelli di riempimento della pipeline, determinare il numero di cicli totali per l'esecuzione del loop se all'inizio: $\$s1 = 8X$
- Verificare se sono possibili ottimizzazioni ri-schedulando le istruzioni

Loop unrolling

- Inseriamo le `nop` opportune (nel *branch delay slot*, e per forzare gli eventuali stalli dovuti alle `lw`):

```
loop:
  lw $t0, 0($s1)
  nop
  add $t0, $t0, $s2
  sw $t0, 0($s1)
  addi $s1, $s1, -4
  bne $s1, $0, loop
  nop
```

Le dipendenze in **rosso** sono di tipo RAW

Le dipendenze in **verde** sono di tipo WAR

- 7 cicli per ogni iterazione del loop
- Poiché `$s1` viene decrementato di 4 in 4 ad ogni ciclo, fino a diventare uguale a zero all'uscita del loop
 - 2X iterazioni, per un totale di $7 \cdot 2X = 14X$ cicli totali
- A causa delle dipendenze sopra illustrate, l'ordine di esecuzione delle istruzioni non può essere modificato, e quindi non è possibile ottimizzare il codice

Loop unrolling

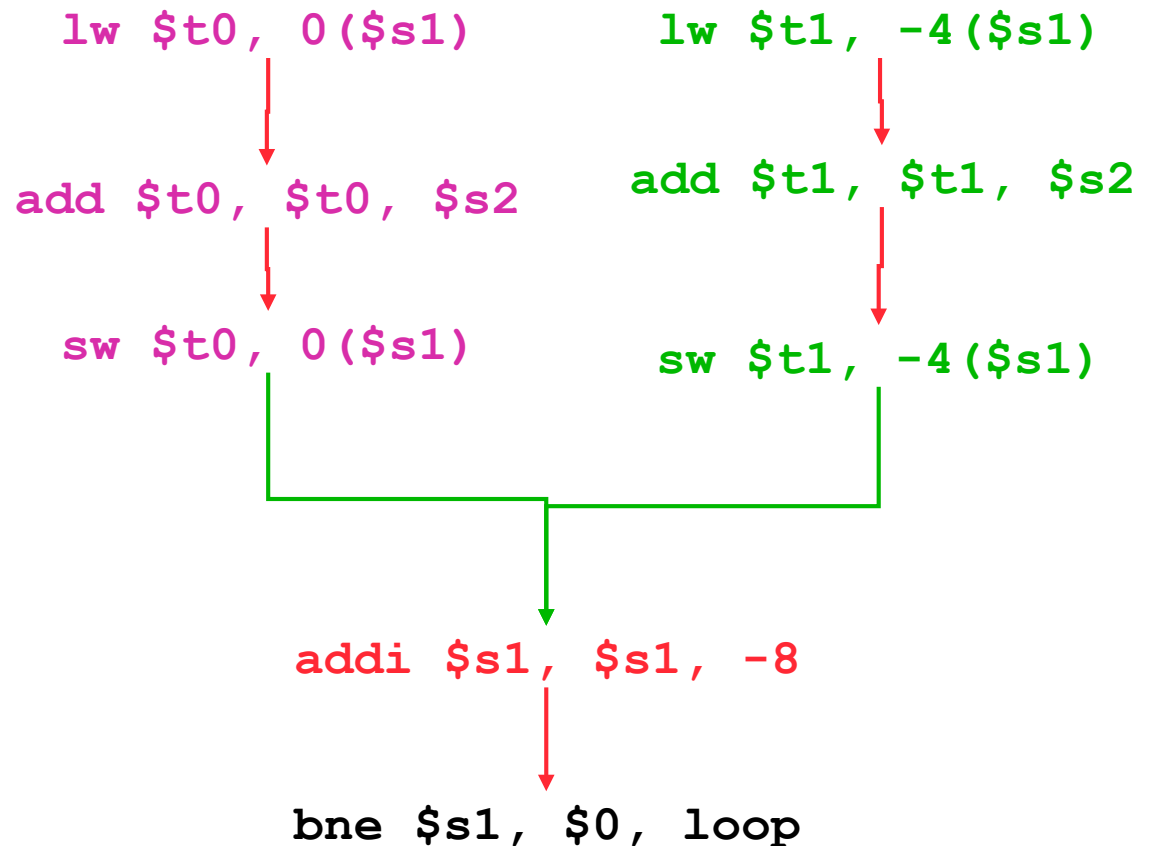
- Consideriamo la seguente ottimizzazione, nota come *loop unrolling*:
 - poiché il ciclo viene eseguito per un numero pari di volte (2X), possiamo srotolare il loop originale
 - il corpo del nuovo loop dovrà eseguire 2 iterazioni del vecchio loop
 - verranno usati **registri temporanei diversi** per accedere gli elementi del vettore puntati da \$s1

loop:

```
lw $t0, 0($s1)
add $t0, $t0, $s2
sw $t0, 0($s1)
lw $t1, -4($s1)
add $t1, $t1, $s2
sw $t1, -4($s1)
addi $s1, $s1, -8
bne $s1, $0, loop
```

Loop unrolling

- Consideriamo dipendenze le data flow tra le istruzioni del body del loop, e vediamo quali sono gli ordinamenti imposte da tali dipendenze:

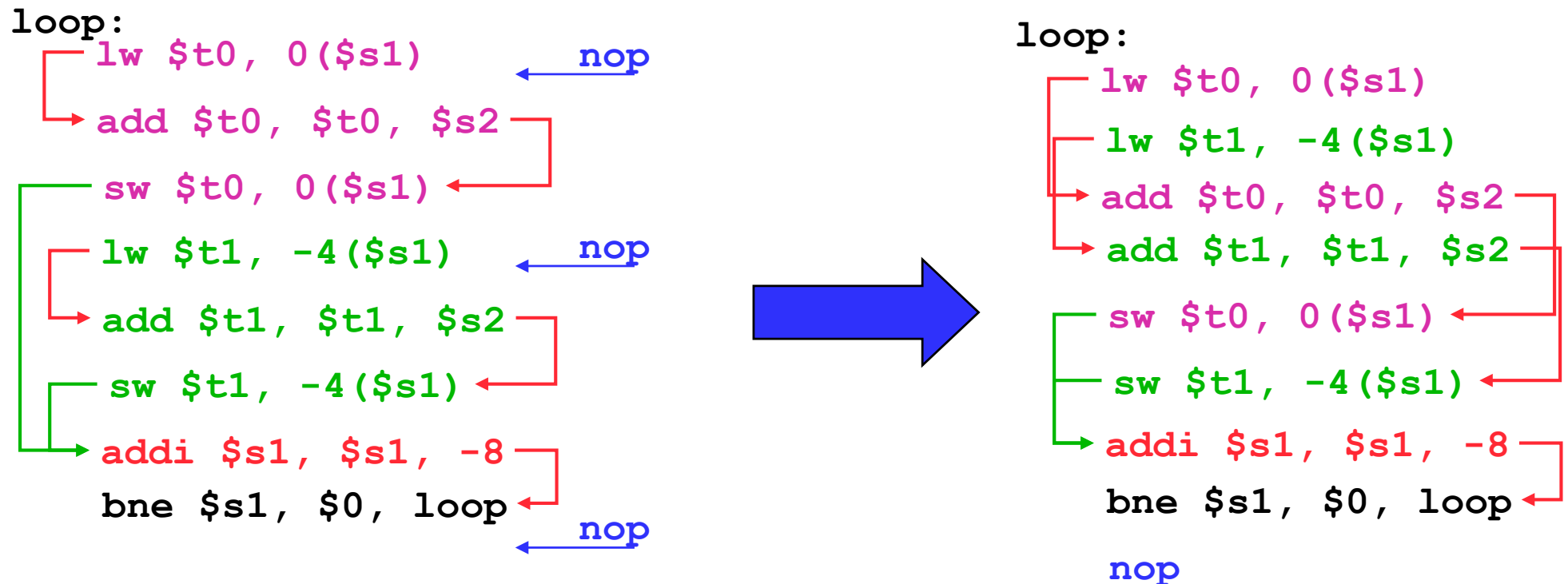


Le dipendenze in **rosso** sono di tipo RAW

Le dipendenze in **verde** sono di tipo WAR

Loop unrolling

- Considerando le dipendenze, effettuare il ri-schedulazione delle istruzioni per tentare di riempire i delay slot dovuti alle istruzioni `bne` e `lw`
Ricalcolare il numero di cicli totali per ogni iterazione del loop, e lo speedup rispetto al caso senza *loop unrolling*



- Vecchio loop eseguito per **2X volte** - Nuovo loop eseguito per **X volte**
- Ogni iterazione del nuovo loop eseguita in **9 cicli**, per un totale di **9X cicli**
- Speedup** = cicli senza loop unrolling / cicli con loop unrolling = **14X / 9X ≈ 1.56**