Tutorato di architettura degli elaboratori

Instruction Level Parallelism

Andrea Gasparetto – andrea.gasparetto@unive.it

IF: Instruction Fetch (memoria istruzioni)

ID: Instruction decode e lettura registri

EXE: Esecuzione istruzioni e calcolo indirizzi

MEM: Accesso alla memoria (memoria dati)

WB: Write Back (scrittura del registro risultato, calcolato in EXE o MEM)

Dipendenze RAW: Read After Write, un'istruzione legge un registro scritto da un'istruzione precedente

Esempio:

```
ADD $s0, $t0, $t1      # Write $s0
SUB $t2, $s0, $t3      # Read $s0
```

Forwarding: operazione che permette di utilizzare il risultato di una computazione senza dover aspettare che avvenga il WB.

```
ADD A B C      #A=B+C
SUB D C A      #D=C-A
```

Without operand forwarding

1	2	3	4	5	6	7	8
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result			
	Fetch SUB	Decode SUB	NOP	NOP	Read Operands SUB	Execute SUB	Write result

With operand forwarding

1	2	3	4	5	6
Fetch ADD	Decode ADD	Read Operands ADD	Execute ADD	Write result	
	Fetch SUB	Decode SUB	Read Operands SUB: use result from previous operation	Execute SUB	Write result

Un registro viene scritto nella prima parte del ciclo e legge una copia del registro nella seconda parte del ciclo.

Delayed Branch: funzionalità del vecchio processore MIPS. L'indirizzo del salto viene calcolato nello stadio ID dell'istruzione branch. L'istruzione posta successivamente al salto entra comunque nella pipeline e viene completata. Il compilatore inserisce dopo al salto una nop esplicita o un'altra istruzione che non influisce sul risultato del programma. MIPS = delay slot = 1.

Processore Pipeline a 5 stadi: ogni istruzione necessita di 1 ciclo per essere eseguita

Differenza Pipeline / Multiciclo: nelle pipeline ogni istruzione costa 1 ciclo, nel multiciclo ogni istruzione costa un certo numero di cicli prefissato.

Esercizio 1

Considerare una pipeline a 5 stadi senza forwarding, con un register file usuale, che prima legge il vecchio valore di un registro e poi lo scrive. Il processore è fornito di hazard detection unit, quindi è in grado di mettere in stallo la pipeline

Determinare le dipendenze RAW tra le istruzioni del programma assembler seguente

sub $3, $2, $5

lw $10, 4($3)

addi $3, $3, 8

add $20, $20, $10

Disegnare il diagramma temporale di esecuzione. Cosa succede all'8° ciclo di clock nei vari stadi?

Se il processore non fosse dotato di hazard detection unit, dove dovrebbero essere inserite le nop per evitare inconsistenze dovute alle dipendenze sui dati?

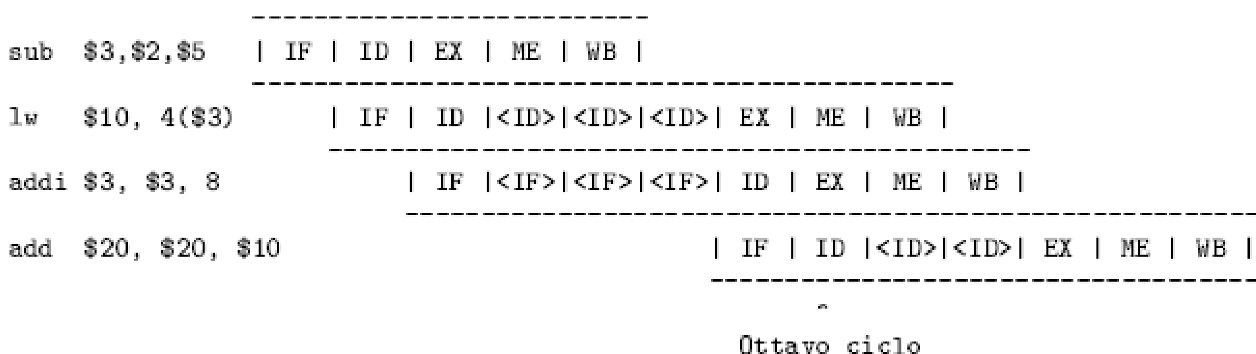
Soluzione

Dipendenze

1 → 2

2 → 4

Diagramma



All’8° ciclo lo stadio IF sta eseguendo l’eventuale quinta istruzione, lo stadio ID sta eseguendo la quarta istruzione (add), lo stadio EXE sta eseguendo la terza istruzione (addi), lo stadio MEM sta eseguendo la seconda istruzione (lw), mentre lo stadio WB sta eseguendo a vuoto (bolla).

Se le inconsistenze dovute alle dipendenze devono essere evitate inserendo nop, il programma corretto diventa:

sub $3, $2, $5

nop

nop

nop

lw $10, 4($3)

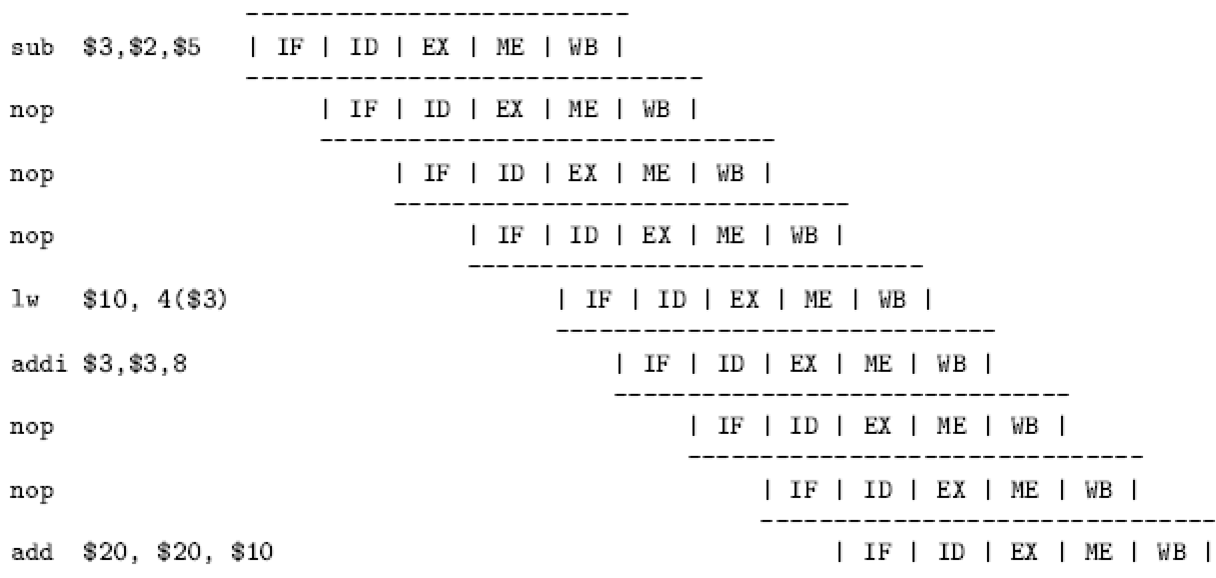
addi $3, $3, 8

nop

nop

add $20, $20, $10

Diagramma



```
sub  $3,$2,$5    | IF | ID | EX | ME | WB |
nop                   | IF | ID | EX | ME | WB |
nop                        | IF | ID | EX | ME | WB |
nop                             | IF | ID | EX | ME | WB |
lw   $10, 4($3)                       | IF | ID | EX | ME | WB |
addi $3,$3,8                              | IF | ID | EX | ME | WB |
nop                                            | IF | ID | EX | ME | WB |
nop                                                 | IF | ID | EX | ME | WB |
add  $20, $20, $10                                       | IF | ID | EX | ME | WB |
```

Esercizio 2

Programma

ori $6, $0, 0

ori $7, $0, 1000

```
loop:
    sll $5, $6, 2 #moltiplico per 4
    add $8, $4, $5
    lw $9, 0($8)
    and $9, $9, $0
    sw $9, 0($8)
    addi $6, $6, 1
    bne $6, $7, loop
```

Considerando che $4 contiene l'indirizzo di un vettore di interi, cosa fa questo codice?

Determinare il CPI del programma nel caso in cui il processore sia multiciclo (si ipotizzi che tutte le istruzioni R e I di tipo aritmentico/logico impieghino 4 cicli di clock, lw 5, bne 3).

Rispetto all'implementazione pipeline vista a lezione (5 stadi, forwarding e delayed branch), dove si verificano eventuali stalli? Inserire le istruzioni nop opportune, e ricalcolare il CPI, senza considerare i cicli persi per il riempimento della pipeline.

Soluzione

Il programma azzera un vettore di int di 1000 elementi.

Per il processore multicilco bisogna considerare che bne impiega 3 cicli, lw 5 cicli e tutte le altre istruzioni 4 cicli. Quindi abbiamo 8 cicli per il preambolo del loop, e 28 * 1000 per il corpo del loop, per un totale di 28008 cicli. Il numero di istruzioni eseguite è IC = 2 + 7 * 1000 = 7002. Per cui abbiamo che

CPI = 28008 / 7002 = 4

Per quanto riguarda il processore pipeline, abbiamo uno stallo dopo la lw, e dobbiamo forzare uno stallo dopo la beq a causa del delayed branch. Quindi il programma modificato è il seguente, dove ad ogni ciclo viene completata un'istruzione (comprese le nop)

```
ori $6, $0, 0
ori $7, $0, 1000
loop:
    sll $5, $6, 2 #moltiplico per 4
    add $8, $4, $5
    lw $9, 0($8)
    nop
    and $9, $9, $0
    sw $9, 0($8)
    addi $6, $6, 1
```

bne $6, $7, loop

nop

Sono necessari quindi un numero di cicli pari a 2 + 9*1000 = 9002. Per cui abbiamo che

CPI = 9002 / 7002 = 1.286

Esercizio 3

Considerare il processore pipeline MIPS a 5 stadi visto a lezione, con delayed branch, forwarding, e register file speciale, e il seguente programma MIPS, che incrementa gli elementi di un array di interi, il cui indirizzo iniziale è contenuto nel registro $20, mentre $10 contiene l'indice dell'array.

Loop:

1. add $11, $20, $10
2. lw $17, 0($11)
3. addi $17, $17, 50
4. sw $17, 0($11)
5. addi $10, $10, 4
6. bne $10, $21, Loop

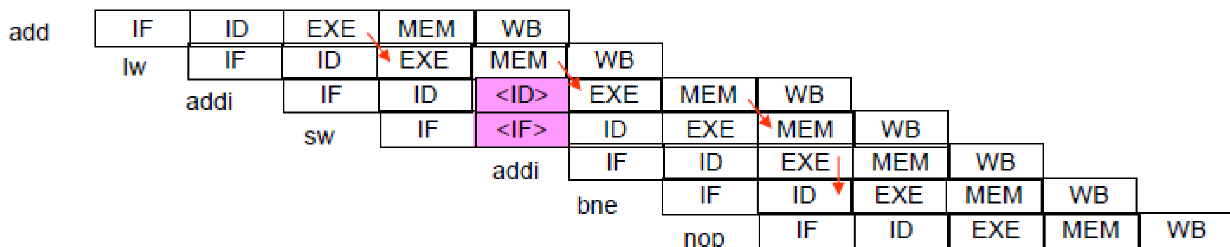
Determinare le dipendenze RAW, e il diagramma temporale di esecuzione delle istruzioni che appaiono nel corpo del loop, mettendo in evidenza i forwarding. Forzare con delle nop gli stalli che verrebbero comunque inseriti dall'hazard detection unit, e ottimizzare il codice.

Rispetto al codice non ottimizzato, determinare il diagramma temporale nel caso in cui il processore non avesse il forwarding.

Soluzione

Le dipendenze RAW sono 1 → 2 e 1 → 4 ($11), 2 → 3 ($17), 3 → 4 ($17), 5 → 6 ($10).

Diagramma



Si noti che la dipendenza 1 → 4 non necessita di forwarding, in quanto lo stadio WB dell'istruzione 1 si verifica al 5o ciclo, mentre lo stadio ID dell'istruzione 4 avviene al 6° ciclo. Si noti infine che, poiché per limitare l'hazard sul controllo abbiamo anticipato allo stadio ID del branch il confronto tra i 2 registri (tramite una batteria di porte XOR), per evitare stalli il valore calcolato dall 5° istruzione ($10) nello stadio EXE deve fluire direttamente nello stadio ID della 6a istruzione. Per ottenere tale risultato dobbiamo modificare la circuiteria del processore in modo da permettere il recupero del valore calcolato dallo stadio EXE della 5° istruzione nello stadio ID della 6° istruzione.

Le uniche dipendenze che non sono risolte dal forwarding (o dal register file speciale) sono quelle tra la lw e l'istruzione successiva (2 → 3). Un'altra nop bisogna inserirla esplicitamente a causa del delay branch. Abbiamo quindi:

Loop:

add $11, $20, $10

lw $17, 0($11)

nop

addi $17, $17, 50

sw $17, 0($11)

addi $10, $10, 4

bne $10, $21, Loop

nop

Possiamo ottimizzare il codice, spostando indietro l'istruzione 5 (addi) dopo la lw, in modo da eliminare lo stallo, e l'istruzione 4 (sw) in avanti nel branch delay slot:

Loop:

add $11, $20, $10

lw $17, 0($11)

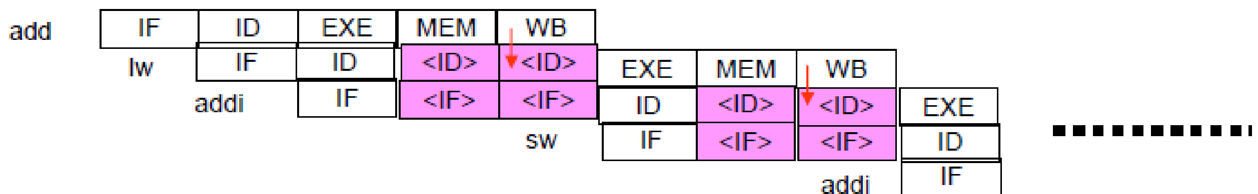
addi $10, $10, 4

addi $17, $17, 50

bne $10, $21, Loop

sw $17, 0($11)

Il diagramma relativo al codice non ottimizzato, nel caso in cui il forwarding non fosse attivo, è illustrato (solo parzialmente) di seguito:



Si noti come, grazie al register file speciale (frecce rosse), che scrive un nuovo registro nella prima parte del ciclo, e legge una coppia di registri nella seconda parte dello stesso ciclo, si risparmia un ciclo di stallo.

Esercizio 4

Dato il seguente loop:

loop:

1. lw $t10, 0($t8)
2. add $t11, $t10, $t11
3. addi $t8, $t8, 4
4. sw $t11, -4($t8)
5. bne $t8, 4096, loop

Individuare le dipendenze RAW, e disegnare il diagramma di esecuzione per un processore MIPS pipeline a 5 stadi (come quello visto a lezione, con delayed branch) nei 3 casi seguenti:

1. Senza forwarding, con un register file semplice, che durante ogni ciclo di clock legge i vecchi valori di una coppia di registri e scrive un nuovo registro.

2. Ancora senza forwarding, ma con un register file ottimizzato, che permette di eliminare qualche stallo dovuto alle dipendenze (scrive un nuovo registro nella prima parte di un ciclo, e legge una coppia di registri nella seconda parte del ciclo).

3. Con forwarding, e con il register file ottimizzato del punto 2.

Individuare un'ottimizzazione del codice per il caso 3 che riduce gli stalli.

Soluzione

Le dipendenze RAW sono:

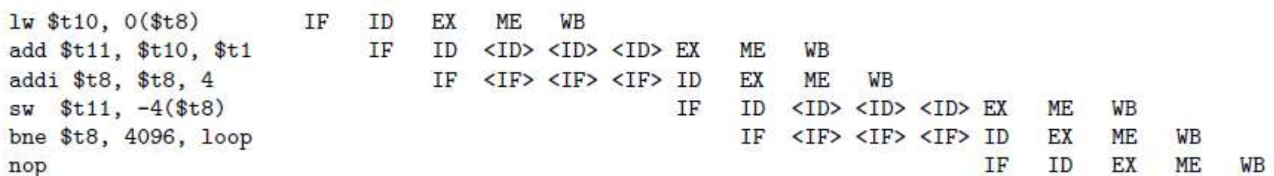
1->2 (reg. $10)

2->4 (reg. $11)

3->4 (reg. $8)

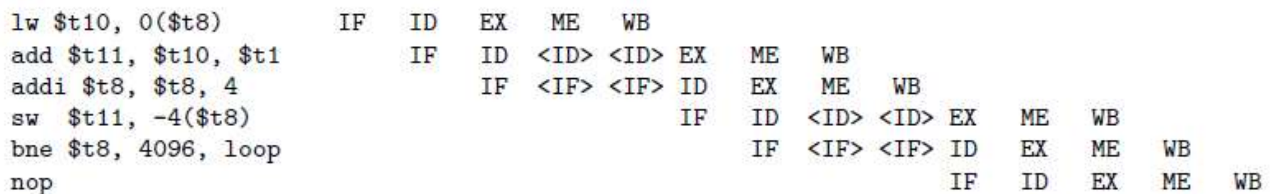
3->5 (reg. $8)

1)



lw $t10, 0($t8)	IF	ID	EX	ME	WB											
add $t11, $t10, $t1		IF	ID	<ID>	<ID>	<ID>	EX	ME	WB							
addi $t8, $t8, 4			IF	<IF>	<IF>	<IF>	ID	EX	ME	WB						
sw $t11, -4($t8)							IF	ID	<ID>	<ID>	<ID>	EX	ME	WB		
bne $t8, 4096, loop								IF	<IF>	<IF>	<IF>	ID	EX	ME	WB	
nop												IF	ID	EX	ME	WB

2)



lw $t10, 0($t8)	IF	ID	EX	ME	WB									
add $t11, $t10, $t1		IF	ID	<ID>	<ID>	EX	ME	WB						
addi $t8, $t8, 4			IF	<IF>	<IF>	ID	EX	ME	WB					
sw $t11, -4($t8)						IF	ID	<ID>	<ID>	EX	ME	WB		
bne $t8, 4096, loop							IF	<IF>	<IF>	ID	EX	ME	WB	
nop										IF	ID	EX	ME	WB

3)

```
lw $t10, 0($t8)        IF   ID   EX    ME   WB
add $t11, $t10, $t1         IF   ID   <ID> EX   ME   WB
addi $t8, $t8, 4                 IF   <IF> ID   EX   ME   WB
sw  $t11, -4($t8)                          IF   ID   EX   ME   WB
bne $t8, 4096, loop                             IF   ID   EX   ME   WB
nop                                                  IF   ID   EX   ME   WB
```

Per il caso 3, il codice ottimizzato che elimina tutti gli stalli, e sposta un'istruzione indipendente nel delay slot è la seguente:

```
loop:
        lw $t10, 0($t8)
        addi $t8, $t8, 4
        add $t11, $t10, $t11
        bne $t8, 4096, loop
        sw $t11, -4($t8)
```

Esercizio 5

Si individuino le dipendenze RAW (Read After Write) nel seguente codice MIPS:

1. lw $2, 25($27)
2. add $4, $2, $6
3. add $1, $27, $3
4. sub $5, $2, $1
5. and $5, $5, $28

Si consideri una CPU MIPS a 5 stadi come quella vista a lezione, ma senza forwarding, e con un register file normale, che durante un ciclo di clock permette di leggere i vecchi valori di una coppia di registri, e di modificare un solo registro alla fine dello stesso ciclo. La CPU in questione non ha hazard detection unit, per cui la gestione delle dipendenze è affidata al software.

Si richiede quindi di inserire le nop opportune nel codice, di verificare il funzionamento con un diagramma temporale, e di calcolare il n° di cicli totali spesi per l'esecuzione.

Si chiede infine di modificare l'ordine di esecuzione delle istruzioni per ridurre il numero di cicli impiegati per l'esecuzione del programma. Verificare il risparmio in cicli tramite il diagramma temporale di esecuzione.

Soluzione

Le dipendenze sono:

1->2 (reg. $2)

1->4 (reg. $2)

3->4 (reg. $1)

4->5 (reg. $5)

Per far rispettare le dipendenze di sopra, istr0 -> istr1, basta fare in modo che lo stadio ID dell'istruzione istr1 avvenga successivamente allo stadio WB di istr0. Dopo aver inserito le nop opportune, il nuovo diagramma temporale è il seguente.

```
lw :   IF   ID   EX   MEM  WB
nop:        IF   ID   EX   MEM  WB
nop:             IF   ID   EX   MEM  WB
nop:                  IF   ID   EX   MEM  WB
add:                       IF   ID   EX   MEM  WB
add:                            IF   ID   EX   MEM  WB
nop:                                 IF   ID   EX   MEM  WB
nop:                                      IF   ID   EX   MEM  WB
nop:                                           IF   ID   EX   MEM  WB
sub:                                                IF   ID   EX   MEM  WB
nop:                                                     IF   ID   EX   MEM  WB
nop:                                                          IF   ID   EX   MEM  WB
nop:                                                               IF   ID   EX   MEM  WB
and:                                                                    IF   ID   EX   MEM  WB
```

Scambiando 2 con 3 abbiamo:

1. lw $2, 25($27)

3. add $1, $27, $3

2. add $4, $2, $6

4. sub $5, $2, $1

5. and $5, $5, $28

Grazie a questa modifica, possiamo ridurre il numero di nop inserite.

```
lw :   IF   ID   EX   MEM  WB
add:        IF   ID   EX   MEM  WB
nop:             IF   ID   EX   MEM  WB
nop:                  IF   ID   EX   MEM  WB
add:                       IF   ID   EX   MEM  WB
sub:                            IF   ID   EX   MEM  WB
nop:                                 IF   ID   EX   MEM  WB
nop:                                      IF   ID   EX   MEM  WB
nop:                                           IF   ID   EX   MEM  WB
and:                                                IF   ID   EX   MEM  WB
```

Esercizio 6

Determinare il numero di cicli complessivi per l'esecuzione del seguente loop nei seguenti casi:

- Processore multiciclo
- Processore pipeline a 5 stadi con forwarding e delayed branch (non tener conto dei cicli iniziali di riempimento della pipeline, e considerare che il branch delay slot è riempito con una nop).

Il loop è il seguente

```
loop:
        lw $8, 0($5)
        add $8, $8, $9
        sw $8, 0($5)
        addi $5, $5, 4
        bne $5, $10, loop        # $10 è uguale a 1024
```

dove il valore iniziale del registro 5 è zero.

Calcolare i CPI in entrambi i casi.

Soluzione

Multiciclo:

lw (5 cicli), sw (4 cicli), add (4 cicli) e branch (3 cicli)

L'addi incrementa il contatore ($5) di 4 ad ogni loop, quindi non faccio 1024 loop, ma 1024 / 4

Ogni iterazione del loop costa: 5 + 4 + 4 + 4 + 3 = 20 cicli

Numero totale di cicli: (1024 / 4) * 20 = 5120   NumeroLoop * CicliPerLoop

CPI = 5120 / ((1024 / 4) * 5) = 4   Numero cicli / IC (= NumeroLoop * NumeroIstruzioniLoop)

Pipeline:

Nel pipeline, abbiamo solo uno stallo nella pipeline dopo la lw (a causa di una dipendenza RAW). Possiamo forzare lo stallo con una nop, e mettere una nop nel branch delay slot.

```
loop:
        lw $8, 0($5)
        nop
        add $8, $8, $9
        sw $8, 0($5)
        addi $5, $5, 4
        bne $5, $10, loop        # $10 è uguale a 1024
        nop
```

Ogni iterazione del loop costa quindi 7 cicli

Il numero totale di cicli è (1024 / 4) * 7 = 1729

CPI = 1729 / ((1024/4)*5) = 1.4

Esercizio 7

Si individuino le dipendenze RAW (Read After Write) nel seguente codice MIPS:

1. lw $v0, 8($s0)
2. add $t5, $v0, $s7
3. add $t0, $s0, $t2
4. sub $t4, $v0, $t0
5. and $t4, $t4, $s1

Si consideri una CPU MIPS a 5 stadi come quella vista a lezione, ma senza forwarding, e con un register file speciale, che durante un ciclo di clock permette prima di scrivere un nuovi valore di un registro e poi di leggere il valore alla fine dello stesso ciclo.

La CPU in questione non ha hazard detection unit, per cui la gestione delle dipendenze `e affidata al software. Si richiede quindi di inserire le nop opportune nel codice, di verificare il funzionamento con un diagramma temporale, e di calcolare il # di cicli totali spesi per l'esecuzione.

Cosa cambierebbe se il datapath della CPU fosse dotato di forwarding?

Soluzione

Le dipendenze sono:

1->2 (reg. $v0)

1->4 (reg. $v0)

3->4 (reg. $t0)

4->5 (reg. $t4)

Per far rispettare le dipendenze di sopra, istr0 -> istr1, basta fare in modo che lo stadio ID dell'istruzione istr1 avvenga successivamente allo stadio WB di istr0. Dopo aver inserito le nop opportune, il nuovo diagramma temporale è il seguente.

```
1) lw :  IF   ID   EX   MEM  WB
   nop:       IF   ID   EX   MEM  WB
   nop:            IF   ID   EX   MEM  WB
2) add:                 IF   ID   EX   MEM  WB
3) add:                      IF   ID   EX   MEM  WB
   nop:                           IF   ID   EX   MEM  WB
   nop:                                IF   ID   EX   MEM  WB
4) sub:                                     IF   ID   EX   MEM  WB
   nop:                                          IF   ID   EX   MEM  WB
   nop:                                               IF   ID   EX   MEM  WB
5) and:                                                    IF   ID   EX   MEM  WB
```

Con il forwarding, abbiano ancora bisogno di inserire una nop.

```
1) lw :    IF   ID   EX  MEM   WB
   nop:         IF   ID   EX  MEM   WB
2) add:              IF   ID   EX  MEM   WB
3) add:                   IF   ID   EX  MEM   WB
4) sub:                        IF   ID   EX  MEM   WB
5) and:                             IF   ID   EX  MEM   WB
```

Esercizio 8

Si consideri il seguente programma:

1. ori $s0, $0, 0

init_loop:

2. add $t0, $a0, $s0

3. lw $t1, 0($t0)

4. addi $t1, $t1, 50

5. sw $t1, 0($t0)

6. addi $s0, $s0, 4

7. bne $s0, $a1, init_loop

dove $a0 contiene l’indirizzo iniziale di un array di interi, $s0 viene usato come indice dell’array, mentre $a1 contiene il numero di byte dell’array ($a0/4 `e il numero di elementi dell’array).

1. Discutere cosa calcola il codice;

2. Individuare le dipendenze RAW (Read After Write) nel codice, facendo riferimenti ai numeri che identificano le varie istruzioni e al registro che causa ciascuna dipendenza;

3. Facendo riferimento all’architettura MIPS vista a lezione con pipeline a 5 stadi con forwarding, si disegni un diagramma temporale di esecuzione di una singola iterazione del loop, mettendo in evidenza i vari forwarding necessari per risolvere le dipendenze RAW.

Soluzione

Il codice incrementa di 50 tutti gli elementi dell’array puntato da $a0. Le dipendenze RAW sono le seguenti:

1 -> 2 ($s0)

2 -> 3 ($t0)

3 -> 4 ($t1)

4 -> 5 ($t1)

1 -> 6 ($s0)

6 -> 7 ($s0)

Il diagramma di esecuzione del corpo del loop con i relativi forwarding è il seguente: