Esame di Architetture – Prof. Sterbini – 9/6/17 – compito A

Parte 1 (per chi non ha superato l'esonero)

Esercizio 1A (12 punti). Una CPU a ciclo di clock singolo (vedi sul retro) potrebbe avere la Control Unit rotta, che produce il segnale di controllo **MemToReg** attivo **solo quando** è attivo il segnale di controllo **Branch**.
Si assume che RegDst sia asserito solo per le istruzioni di tipo R, che MemToReg sia asserito solo per l'istruzione lw, che AluSrc sia asserito solo per le istruzioni lw e sw e di tipo immediato.

a) Si indichino *qui sotto* quali delle istruzioni base (**lw, sw, di tipo R, beq, j, R immediate**) funzioneranno male e qual'è il comportamento anomalo in caso di CU guasta.

Soluzione

	MemToReg	Branch	
lw	1->0	0	$rt ← $rs+ext(imm) = address
sw	0	0	OK
R	0	0	OK
R-imm	0	0	OK
beq	0->1	1	OK (perché non modifica registri)
j	0	0	OK

l'unica istruzione che non funziona è **lw** che si comporta come una **la**

b) si scriva *qui sotto* un breve programma assembly MIPS che termina valorizzando il registro $s0 con il valore 1 se il processore è guasto, altrimenti con 0.

Soluzione

```
.data
OK: .word 0
.text
lw      $s0, OK         # tento di leggere lo zero che sta in memoria, se errato ottengo il suo indirizzo che è != 0
beqz    $s0, funziona   # se funziona evito di caricare 1 in $s0
li      $s0, 1          # carico 1 se $s0 diverso da 0 (indirizzo di OK)
funziona:
...
```

Esercizio 2A (18 punti). Considerate l'architettura MIPS a ciclo singolo in figura (diagramma sul retro).
Si vuole aggiungere l'istruzione di tipo I `jr_min rs, rt` (salto a registo con valore minimo)
che fa un salto **incondizionato** alla destinazione che ha valore minimo tra i due valori contenuti nei due registri **rs** e **rt**
Nella soluzione non modificate gli ingressi/uscite o le funzionalità della ALU. Nota, il salto è assoluto.

1) modificate il diagramma mostrando gli eventuali altri componenti necessari a realizzare l'istruzione

2) indicate sul diagramma tutti i segnali di controllo che la CU genera per realizzare l'istruzione

3) supponendo che l'accesso alle memorie impieghi **66ns**, l'accesso ai registri **33ns**, le operazioni dell'ALU e dei sommatori **100ns**, e ignorando gli altri ritardi di propagazione dei segnali, indicate sul diagramma la durata totale del ciclo di clock per permettere l'esecuzione anche della nuova istruzione.

Soluzione

Si veda il diagramma che segue, in cui si usa il bit MSB alla uscita della ALU (usata come comparatore, che fa una sottrazione) per vedere se il risultato è negativo e comandare un MUX che seleziona il valore del registro minore, che poi va al PC.

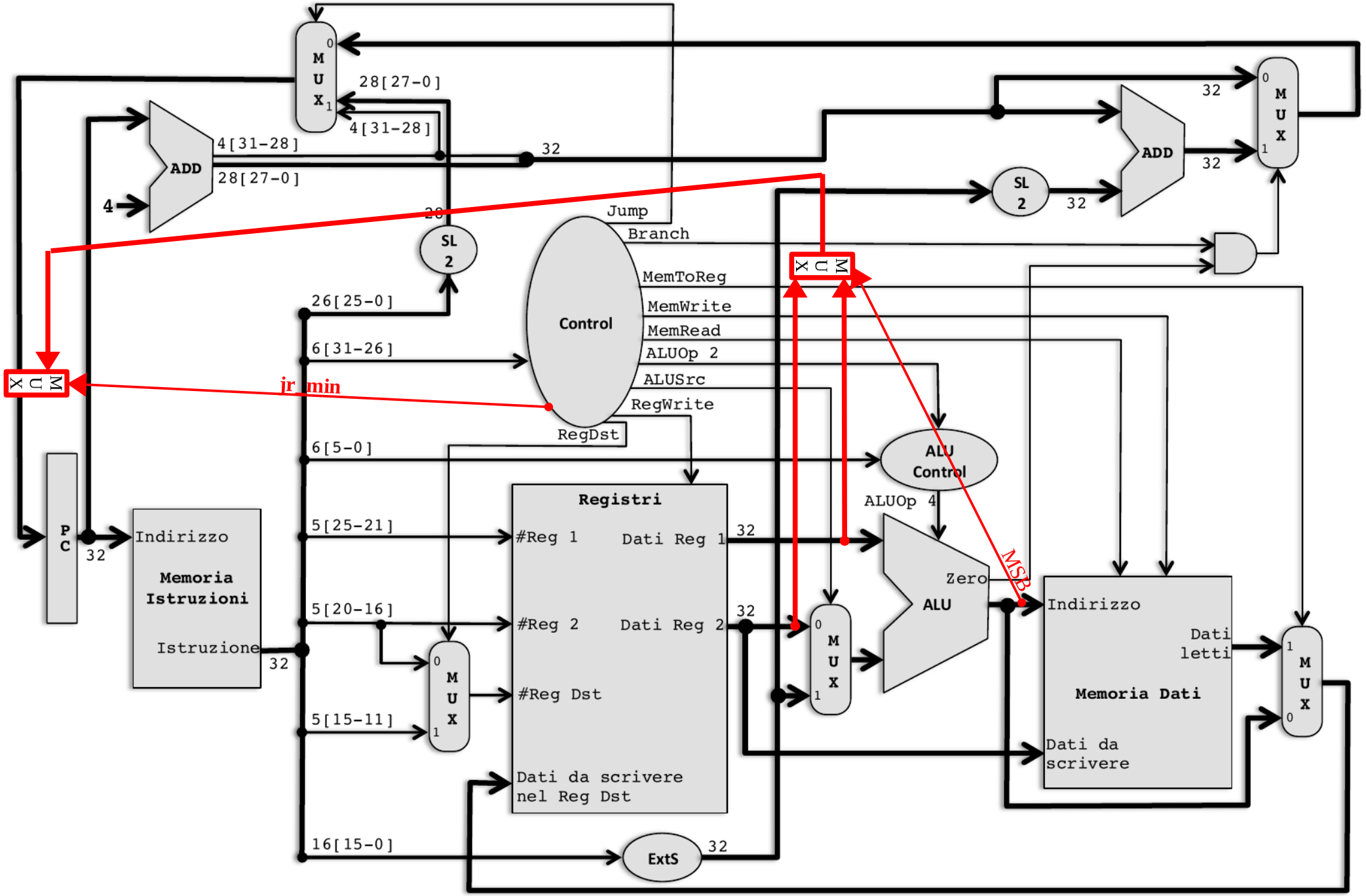
Sono necessari i segnali: **AluOp=sub, AluSrc=0, RegWrite=0, MemWrite=0, jr_min=1,** e il resto don't care

il tempo necessario è

datapath principale: 66ns fetch + 33ns id + 100ns ALU = 199ns < lw

aggiornamento PC: non appena il datapath principale ha calcolato la differenza il valore minimo viene scritto nel PC

Implementazione ad un ciclo di clock di MIPS (solamente le istruzioni: `add`, `sub`, `and`, `or`, `xor`, `slt`, `lw`, `sw`, `beq`, `j`)



Esame di Architetture – Prof. Sterbini – 9/6/17

Parte 2 (per tutti)

Esercizio 3 (16 punti). Si consideri l'architettura MIPS con pipeline mostrata in figura (sul retro) ed il frammento di programma qui a destra che somma i valori della diagonale secondaria di una matrice 10x10 di half word, scandendo la matrice per puntatori dall'inizio alla fine a passi di N-1 elementi.

```
.data
matrice: .space 200
DIM:        .word 10  # lato della matrice
.text
main:    xor  $s4, $s4, $s4   # somma=0          5
         lw   $s2, DIM        # N                6
1 stallo DH
         subi $s2, $s2, 1     # N-1              8
         sll  $s2, $s2, 1     # 2(N-1)           9
         la   $s0, matrice    #                  10
loop:    addi $s0, $s0, $s2   # prox.el          11
         lh   $s3, ($s0)      # x=M[i]           12
1 stallo DH
         add  $s4, $s4, $s3   # somma+=x         14
         lw   $s3, DIM        # N                15
1 stallo DH
         mul  $s3, $s3, $s3   # N^2              17
         sll  $s3, $s3, 1     # 2(N^2)           18
1 stallo DH
         blt  $s0, $s3, loop  # non fine?        20
1 stallo CH se salta
end:     li   $v0, 10
         syscall
```

NOTA: c'è un errore nel codice e vengono eseguiti 11 cicli invece che i 10 necessari. Nella soluzione ne conto 11.

Si indichino qui sotto o sul codice:
1) tra quali istruzioni sono presenti data hazard,
Soluzione
vedi colori

2) tra quali istruzioni sono presenti control hazard,
Soluzione
il salto verso "loop", se viene eseguito

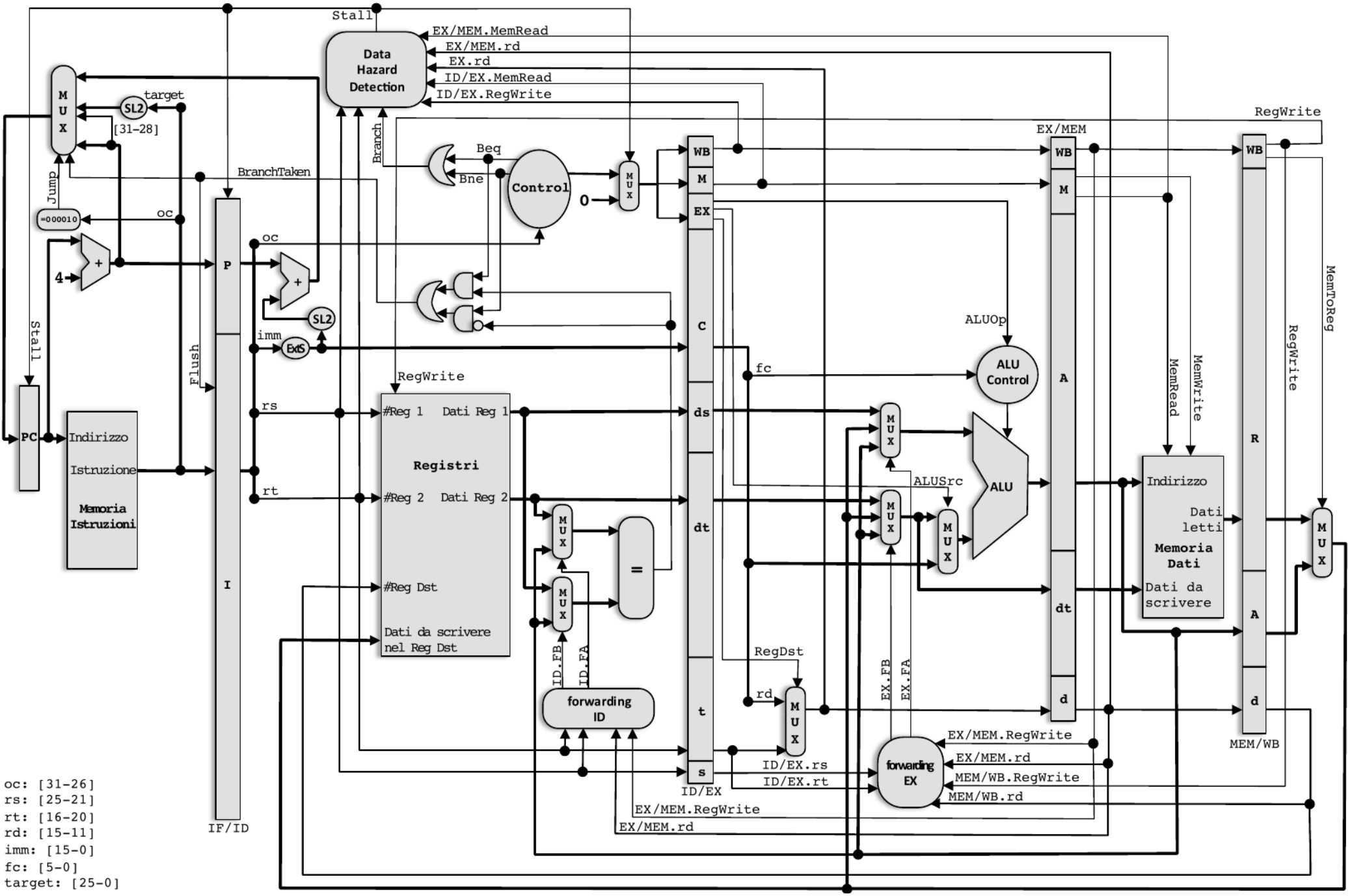
3) quanti cicli di clock sono necessari a eseguire il programma con il forwarding (PASSAGGI INCLUSI)
Soluzione
4 riempimento pipeline + 5 istruzioni prima del loop + 1 stallo per DH + 11 * (7 istruzioni + 3 stalli per DH + 1 stallo per CH) -1 stallo per CH nell'ultimo ciclo che non salta + 2 istruzioni = 11 + 121 = 132 colpi di clock

4) quanti ne sarebbero necessari se il forwarding non esistesse (PASSAGGI INCLUSI)
Soluzione
in questo caso servono 2 stalli per ogni DH (tranne per la **sll** che viene allontanata abbastanza da **addi** quando si inseriscono i 2 stalli dopo il **la**)
4 riempimento pipeline + 5 istruzioni prima del loop + 6 stalli per DH + 11 * (7 istruzioni + 10 stalli per DH + 1 stallo per CH) -1 stallo per CH nell'ultimo ciclo che non salta + 2 istruzioni = 16 + 198 = 214 colpi di clock

5) quali sono le istruzioni contenute nei registri della pipeline durante il **17°** ciclo di clock (con FW)
Soluzione
Vedi il numero del colpo di clock aggiunto a destra del commento

WB: **mul** MEM: **sll** EXE: **stallo** ID: **blt** IF: **li** che diventa **stallo** per CH

Implementazione pipeline di MIPS (solamente le istruzioni: `add`, `addi`, `sub`, `and`, `andi`, `or`, `ori`, `xor`, `xori`, `nor`, `slt`, `slti`, `lw`, `sw`, `beq`, `bne`, `j`).



oc: [31-26]
rs: [25-21]
rt: [16-20]
rd: [15-11]
imm: [15-0]
fc: [5-0]
target: [25-0]

6) come riordinare le istruzioni per ridurre il numero di stalli al massimo in presenza di FW (su foglio a parte)
Soluzione
Parecchie istruzioni possono essere portate fuori dal ciclo rinominando il registro $s3 ad esempio in $s5
Inoltre è possibile spostare **`addi $s0, $s0, $s2`** dopo la lh mantenendo (quasi esattamente) la semantica del programma facendo una copia della addi prima del ciclo, (se non si sposta la addi allora restano alcuni stalli nel ciclo che non sono eliminabili)

```
.data
matrice: .space 200
DIM:     .word 10  # lato della matrice
.text
main:    lw   $s2, DIM        # N              5
         xor  $s4, $s4, $s4   # somma=0        6
         mul  $s5, $s2, $s2   # N^2            7
         sll  $s5, $s5, 1     # 2(N^2)         8
         subi $s2, $s2, 1     # N-1            9
         sll  $s2, $s2, 1     # 2(N-1)         10
         la   $s0, matrice    #                11
         addi $s0, $s0, $s2   # prox.el        12
loop:    lh   $s3, ($s0)      # x=M[i]         13 18 23
         addi $s0, $s0, $s2   # prox.el        14 19 24
         add  $s4, $s4, $s3   # somma+=x       15 20 ...
         blt  $s0, $s5, loop  # non fine?      16 21
end:     li   $v0, 10
         syscall
```

7) quanti colpi di clock sarebbero necessari nel codice così ottimizzato (PASSAGGI INCLUSI)
Soluzione
4 riempimento + **8** istruzioni + 11 * (**4** istruzioni + 1 CH) -1 CH + 2 istruzioni = 13 + 55 = **68 colpi di clock**

8) quali sono le istruzioni contenute nei registri della pipeline durante il **13°** ciclo di clock nel codice ottimizzato
Soluzione
WB: **lh** MEM: **addi** EXE: **add** ID: **blt** IF: **li che poi diventa stallo per CH**

Esame di Architetture – Prof. Sterbini – 9/6/17 – compito A

Matricola: _____

Esercizio A4 (14 punti).
In un sistema in cui la CPU è quella semplice ad un ciclo di clock (cioè senza pipeline) ed in cui la gerarchia di memoria contiene solo una cache **CPU <=> CACHE <=> MEM**
con parametri: dimensione blocco = **2 word** numero di set = **2** numero di vie = **2** politica di rimpiazzo **LRU**
Considerate il programma seguente, che somma gli elementi sulla diagonale di una matrice quadrata di half word, usando puntatori. (considerate che le istruzioni siano di base e non pseudoistruzioni)
1) individuate e scrivete sulla tabella tutti gli indirizzi richiesti dalla CPU (fetch ed accessi alla memoria).
2) una volta individuata la sequenza di indirizzi richiesti dalla CPU calcolate quali sono le HIT e MISS e qual'è il loro tipo (**H**=hit **L**=miss caricamento, **C**=miss conflitto, **X**=miss capacità)

```
.data       500         # indirizzo iniziale del blocco dati
M:          .half 0:16  # matrice 4x4 di half
DIM:        .word 4     # lato della matrice
.text       300         # indirizzo iniziale del blocco programmi
```

	Ciclo 1					Ciclo 2					Ciclo 3					Ciclo 4				
Istruzione	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**
`Main: move $s4, $zero`																				
`lw $s1, DIM`																				
`mul $s1, $s1, $s1`																				
`sll $s1, $s1, 1`																				
`subi $s1, $s1, 2`																				
`loop: lh $s3, M($s1)`																				
`add $s4, $s4, $s3`																				
`lw $s2, DIM`																				
`addi $s2, $s2, 1`																				
`sll $s2, $s2, 1`																				
`sub $s1, $s1, $s2`																				
`bgez $s1, loop`																				
`end: li $v0, 10`																				
`syscall`																				

3) calcolate la **percentuale di HIT:** _____ e la **percentuale di MISS:** _____

4) calcolate il tempo medio di accesso se un HIT risponde in **3ns** e la memoria risponde in **25ns** **tempo medio:** _____

5) suggerite quale/i parametro/i della cache va/vanno modificato/i (e come) per ottenere un maggior numero di HIT: _____

Soluzione

Ogni istruzione occupa una word, mentre i dati della matrice M occupano 2 byte, quindi le posizioni sono quelle indicate

.data	500		Indirizzi dei dati																
M:	.half	0:16		500	502	504	506	508	510	512	514	516	518	520	522	524	526	528	530
DIM:	.word	4		532															
.text	300																		

	Ciclo 1					Ciclo 2					Ciclo 3					Ciclo 4				
Istruzione	**Address**	**# block**	**Tag**	**Index**	**H/M**	**Address**	**# block**	**Tag**	**Index**	**H/M**	**Address**	**# block**	**Tag**	**Index**	**H/M**	**Address**	**# block**	**Tag**	**Index**	**H/M**
`Main: move $s4, $zero`	300	37	18	1	L															
`lw $s1, DIM`	304	38	19	0	L															
	532	66	33	0	L															
`mul $s1, $s1, $s1`	308	38	19	0	H															
`sll $s1, $s1, 1`	312	39	19	1	L															
`subi $s1, $s1, 2`	316	39	19	1	H															
`loop: lh $s3, M($s1)`	320	40	20	0	L	320	40	20	0	X	320	40	20	0	X	320	40	20	0	X
	530	66	33	0	C	520	65	32	1	L	510	63	31	1	L	500	62	31	0	L
`add $s4, $s4, $s3`	324	40	20	0	H	324	40	20	0	H	324	40	20	0	H	324	40	20	0	H
`lw $s2, DIM`	328	41	20	1	L	328	41	20	1	X	328	41	20	1	X	328	41	20	1	H
	532	66	33	0	H	532	66	33	0	X	532	66	33	0	X	532	66	33	0	X
`addi $s2, $s2, 1`	332	41	20	1	H	332	41	20	1	H	332	41	20	1	H	332	41	20	1	H
`sll $s2, $s2, 1`	336	42	21	0	L	336	42	21	0	X	336	42	21	0	X	336	42	21	0	X
`sub $s1, $s1, $s2`	340	42	21	0	H	340	42	21	0	H	340	42	21	0	H	340	42	21	0	H
`bgez $s1, loop`	344	43	21	1	L	344	43	21	1	X	344	43	21	1	X	344	43	21	1	H
`end: li $v0, 10`																348	43	21	1	H
`syscall`																352	44	22	0	L
				HIT	6				HIT	3				HIT	3				HIT	6
Cache	word	byte		MISS	9				MISS	6				MISS	6				MISS	5
dim blocco	2	8																		
# vie	2			TOT HIT		18								**L**	caricamento					
# set	2			TOT MISS		26								**C**	conflitto					
														X	capacità					
Tempo medio di accesso	**16.00**	ns																		
HIT	3	ns		HIT%		**41%**														
MEM	25	ns		MISS%		**59%**														

Per aumentare il numero di hit possiamo:

* aumentare il numero di linee per ridurre le miss di capacità
* aumentare l'associatività per eliminare la miss di conflitto
* oppure aumentare la dimensione del blocco per ridurre le miss di caricamento

Esame di Architetture – Prof. Sterbini – 9/6/17 – compito B

Parte 1 (per chi non ha superato l'esonero)

Esercizio 1B (12 punti). Una CPU a ciclo di clock singolo (vedi sul retro) potrebbe avere la Control Unit rotta, che produce il segnale di controllo **RegDst** attivo **solo quando** è attivo il segnale di controllo **MemToReg**.
Si assume che RegDst sia asserito solo per le istruzioni di tipo R, che MemToReg sia asserito solo per l'istruzione lw, che AluSrc sia asserito solo per le istruzioni lw e sw e di tipo immediato.

a) Si indichino *qui sotto* quali delle istruzioni base (**lw, sw, di tipo R, beq, j, R immediate**) funzioneranno male e qual'è il comportamento anomalo in caso di CU guasta.

Soluzione

	RegDst	MemToReg	
lw	0 → 1	1	lw scrive nel registro $rd invece che $rt (con $rd preso dalla parte immediata)
sw	x0	x0	OK
R	1 → 0	0	salva il risultato nel registro $rt invece che $rd
R imm	0	0	OK
beq	x0	x0	OK
j	x0	x0	OK

b) si scriva *qui sotto* un breve programma assembly MIPS che termina valorizzando il registro $s0 con il valore 1 se il processore è guasto, altrimenti con 0.

Soluzione
Ad esempio posso usare una istruzione di tipo R (non immediata) che dovrebbe dare 0 come risultato ma che se la CU è errata lo mette nel registro sbagliato

```
.text
        li      $s0, 1          # carico 1 in $s0 (immediata), se non viene sostituito la CU è rotta
        li      $t0, 1          # carico 1 in $t0 (immediata), un valore qualsiasi diverso da 0
        sub     $s0, $t0, $t0   # se CU errata il risultato (1-1=0) va in $t0 invece che in $s0 lasciando $s0=1
```

Esercizio 2B (18 punti). Considerate l'architettura MIPS a ciclo singolo in figura (diagramma sul retro).
Si vuole aggiungere l'istruzione di tipo I **`beqal rs, rt, label`** (branch if equal and link) che fa un salto **condizionato** alla destinazione indicata dalla label se i due valori contenuti nei due registri **rs** e **rt** sono uguali e contemporaneamente salva nel registro $ra (ovvero $31) l'indirizzo della istruzione seguente.
Nella soluzione non modificate gli ingressi/uscite o le funzionalità della ALU.

1) modificate il diagramma mostrando gli eventuali altri componenti necessari a realizzare l'istruzione

2) indicate sul diagramma tutti i segnali di controllo che la CU genera per realizzare l'istruzione

3) supponendo che l'accesso alle memorie impieghi **50ns**, l'accesso ai registri **25ns**, le operazioni dell'ALU e dei sommatori **150ns**, e ignorando gli altri ritardi di propagazione dei segnali, indicate sul diagramma la durata totale del ciclo di clock per permettere l'esecuzione anche della nuova istruzione.

Soluzione

la parte di branch è già presente, dobbiamo solo salvare PC+4 nel registro $rs ($31) portando il bus PC+4 alla porta dati in del blocco registri e il valore 31 ($ra) alla porta #RegDst del blocco registri.
Il segnale RegWrite è normalmente 0 ma solo se **BranchTaken and beqal** lo si attiva (inserendo un OR)

Per i tempi, il datapath principle, vedi il diagramma sotto

fetch 50ns + ID 25ns + ALU 150ns + WB 25ns = 250ns

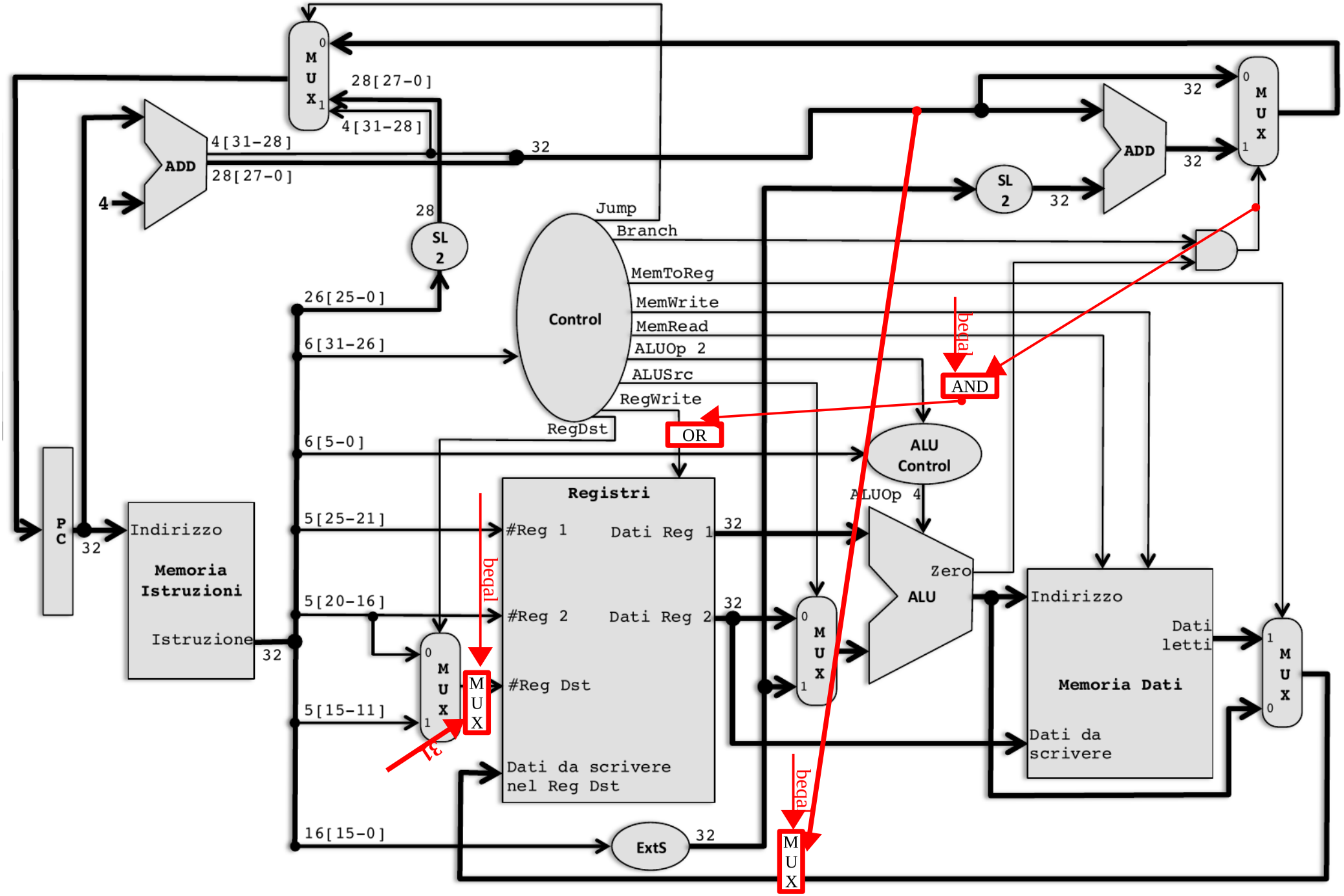
mentre il PC

PC+4 in 150ns, pronto per la fase di WB

PC+4 + perte immediata = destinazione in 300ns

Tempo massimo: 300ns = a una normale beq

Implementazione ad un ciclo di clock di MIPS (solamente le istruzioni: `add`, `sub`, `and`, `or`, `xor`, `slt`, `lw`, `sw`, `beq`, `j`)



Esame di Architetture – Prof. Sterbini – 9/6/17 – compito B

Parte 2 (per tutti)

Esercizio 3B (16 punti). Si consideri l'architettura MIPS con pipeline mostrata in figura (sul retro) **che supponiamo sia estesa con le due istruzioni JR e JAL** ed il frammento di programma qui a destra che somma i valori di un vettore di word, simulando il ciclo con una funzione ricorsiva.

NOTA: entrambe le istruzioni `jal` e `jr` eseguono il salto durante la fase ID, come le istruzioni di branch.

NOTA: ripensandoci, la **jal** può saltare già nella fase Fetch, ma svolgo l'esercizio come detto nel testo (salto in ID e stallo per CH).
NOTA: **jal** potrebbe generare un DH se una delle 2 istruzioni successive usasse $ra, ma qui non succede.

```
.data
dati:       .word 1, 2, 3, 4, 5, 6
DIM:  .word 6  # num. elementi
.text
main:
     li   $a1, 0         # offset=0        5
     lw   $a0, DIM       # N=6             6
     jal  somma_ricorsiva                  7
1 stallo per CH
     move $a0, $v0       # stampo la somma
     li   $v0, 1
     syscall
     li   $v0, 10        # fine
     syscall

somma_ricorsiva:
     bnez $a0, ancora    # se non finiti 9 19
1 stallo per CH se salta
     # altrimenti sono finiti (caso base)
     li   $v0, 0         # Somma=0
     jr   $ra            # torno
1 stallo per CH

ancora:
     subu $sp, $sp, 8    # alloc. stack    11
     sw   $t0, 4($sp)    # salvataggio     12
     sw   $ra, 0($sp)    # su stack        13
     lw   $t0, dati($a1)# M[i]             14
     subi $a0, $a0, 1    # N -= 1          15
     addi $a1, $a1, 4    # offset += 4     16
     jal  somma_ricorsiva # somma ric.     17
1 stallo per CH
     add  $v0, $v0, $t0 # Somma += M[i]
     lw   $t0, 4($sp)    # ripristino
     lw   $ra, 0($sp)    # da stack
     addu $sp, $sp, 8    # disalloc. Stack
1 stallo per DH
     jr   $ra
1 stallo per CH
```

Si indichino qui sotto o sul codice:
1) tra quali istruzioni sono presenti data hazard,
Soluzione
vedi colori
non sono necessari stalli per DH, il forwarding è sufficiente in entrambi i DH
il lw $a0 si trova abbastanza distante dal bnez (perché c'è uno stallo per CH), da non generare DH

2) tra quali istruzioni sono presenti control hazard,
Soluzione
dopo le jal (in realtà no, ma seguo quanto indicato nel testo)
dopo le jr

3) quanti cicli di clock sono necessari a eseguire il programma con il forwarding (PASSAGGI INCLUSI)
Soluzione
La funzione ricorsiva viene eseguita:
6 volte per il caso ricorsivo ($a0=6,5,4,3,2,1)
1 volta per il caso base ($a0=0)
Quindi,
(contando una coppia di CH per ciascuna chiamata/ritorno)
4 riempimento pipeline + 8 istruzioni main
+ 6*(2CH per chiamata e ritorno + 1CH per beq + 13 istruzioni caso ricorsivo + 1 stallo per DH)
+ 1*(2CH per chiamata e ritorno + 3 istruzioni caso base)
= 12 + 6*17 + 5 = 17+102 = 119 colpi di clock

4) quanti ne sarebbero necessari se il forwarding non esistesse (PASSAGGI INCLUSI)
Soluzione
Gli unici data-hazard sono nella gestione dello $sp e inseriscono 2 stalli nell'allocazione e 1 nella disallocazione (c'è già una istruzione tra lw e jr) solo nelle cinque esecuzioni della funzione che non vanno nel caso base
Quindi
4 riempimento pipeline + 8 istruzioni main
+ 6*(2CH per chiamata e ritorno + 1CH per beq + 13 istruzioni + 3 stalli per DH)
+ 1*(2CH per chiamata e ritorno + 3 istruzioni)
= 12 + 6*19 + 5 = 17+114 = 131 colpi di clock

5) quali sono le istruzioni contenute nei registri della pipeline durante il **17°** ciclo di clock (con FW)
Soluzione
WB: **jal** MEM: **stallo CH** EXE: **bnez** ID: **stallo CH** IF: **subu**

Esame di Architetture – Prof. Sterbini – 9/6/17 – compito B

Matricola: _____

Esercizio 4B (14 punti).
In un sistema in cui la CPU è quella semplice ad un ciclo di clock (cioè senza pipeline) ed in cui la gerarchia di memoria contiene solo una cache **CPU <=> CACHE <=> MEM**
con parametri: dimensione blocco = **4 word** numero di set = **4** numero di vie = **2** politica di rimpiazzo **LRU**
Dato il programma seguente, che somma gli elementi sulla diagonale secondaria di una matrice quadrata di word, usando puntatori. (considerate che le istruzioni siano di base e non pseudoistruzioni)
1) individuate e scrivete sulla tabella tutti gli indirizzi richiesti dalla CPU (fetch ed accessi alla memoria).
2) una volta individuata la sequenza di indirizzi richiesti dalla CPU calcolate quali sono le HIT e MISS e qual'è il loro tipo (**H**=hit **L**=miss caricamento, **C**=miss conflitto, **X**=miss capacità)

```
.data       400         # indirizzo iniziale del blocco dati
M:          .word 0:16  # matrice 4x4 di word
DIM:        .word 4     # lato della matrice
.text       200         # indirizzo iniziale del blocco programmi
```

	Ciclo 1					Ciclo 2					Ciclo 3					Ciclo 4				
Istruzione	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**
`main: xor $s4,$s4,$s4`																				
`lw $s2, DIM`																				
`subi $s2, $s2, 1`																				
`sll $s2, $s2, 2`																				
`la $s0, matrice`																				
`loop: addi $s0,$s0,$s2`																				
`lw $s3, ($s0)`																				
`add $s4, $s4, $s3`																				
`lw $s3, DIM`																				
`mul $s3, $s3, $s3`																				
`sll $s3, $s3, 2`																				
`blt $s0, $s3, loop`																				
`li $v0, 1`																				
`move $a0, $s4`																				
`li $v0, 10`																				
`syscall`																				

3) calcolate la **percentuale di HIT:** _____ e la **percentuale di MISS:** _____

4) calcolate il tempo medio di accesso se un HIT risponde in **2ns** e la memoria risponde in **50ns** **tempo medio:** _____

5) suggerite quale/i parametro/i della cache va/vanno modificato/i (e come) per ottenere un maggior numero di HIT: _____

Soluzione

NOTA: il codice ha un errore ed esegue 5 cicli invece che 4, ma durante l'esame abbiamo stabilito di terminare l'esecuzione al 4°

.data	400		Indirizzi dei dati
M:	.word	0:16	
DIM:	.word	4	

400	404	408	412	416	420	424	428	432	436	440	444	448	452	456	460
			464												

.text 200

	Ciclo 1					Ciclo 2					Ciclo 3					Ciclo 4				
Istruzione	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**	**Address**	**# block**	**Tag**	**Index**	**HM/ tipo**
`main: xor $s4,$s4,$s4`	200	12	3	0	L															
`lw $s2, DIM`	204	12	3	0	H															
	464	29	7	1	L															
`subi $s2, $s2, 1`	208	13	3	1	L															
`sll $s2, $s2, 2`	212	13	3	1	H															
`la $s0, matrice`	216	13	3	1	H															
`loop: addi $s0,$s0,$s2`	220	13	3	1	H	220	13	3	1	C	220	13	3	1	H	220	13	3	1	H
`lw $s3, ($s0)`	224	14	3	2	L	224	14	3	2	H	224	14	3	2	H	224	14	3	2	H
	412	25	6	1	L	424	26	6	2	L	436	27	6	3	L	448	28	7	0	L
`add $s4, $s4, $s3`	228	14	3	2	H	228	14	3	2	H	228	14	3	2	H	228	14	3	2	H
`lw $s3, DIM`	232	14	3	2	H	232	14	3	2	H	232	14	3	2	H	232	14	3	2	H
	464	29	7	1	C	464	29	7	1	H	464	29	7	1	H	464	29	7	1	H
`mul $s3, $s3, $s3`	236	14	3	2	H	236	14	3	2	H	236	14	3	2	H	236	14	3	2	H
`sll $s3, $s3, 2`	240	15	3	3	L	240	15	3	3	H	240	15	3	3	H	240	15	3	3	H
`blt $s0, $s3, loop`	244	15	3	3	H	244	15	3	3	H	244	15	3	3	H	244	15	3	3	H
`li $v0, 1`																248	15	3	3	H
`move $a0, $s4`																252	15	3	3	H
`li $v0, 10`																256	16	4	0	L
`syscall`																260	16	4	0	H
				HIT	8				HIT	7				HIT	8				HIT	11
Cache	word	byte		MISS	7				MISS	2				MISS	1				MISS	2
dim blocco	4	16																		
# vie	2			TOT HIT		34	TOT	46			**L**	caricamento								
# set	4			TOT MISS		12					**C**	conflitto								
											X	capacità								
Tempo medio di accesso	**14.52**	ns																		
HIT	2	ns		HIT%		**74%**														
MEM	50	ns		MISS%		**26%**														

Per aumentare il numero di hit possiamo:

* aumentare l'associatività della cache per eliminare le miss di conflitto
* aumentare la dimensione del blocco per diminuire le miss di caricamento

Esame di Architetture – Prof. Sterbini – 9/6/17 – compito A

Parte 3 (assembler)

Esercizio 5 (18 punti se iterativo, 30 se ricorsivo).

1) Scrivere una funzione **cercaSottostringa** in linguaggio assemblativo che, ricevendo gli indirizzi di due stringhe **testo** e **query** di stessa lunghezza, cerca se la seconda stringa è presente nella prima, ne torna la posizione (o -1 se non presente), ovvero:

- partendo dalla posizione N=0 nella stringa **testo**
- e dalla posizione M=0 nella stringa **query**
- dati i due caratteri **X=testo[N+M]** e **Y=query[M]**
  - se Y è '\0' oppure '\n' la stringa **query** è finita ed è stata trovata, si torna il valore N
  - se X è '\0' oppure '\n' la stringa **testo** è finita, query non è stata trovata, si torna il valore -1
  - se X != Y la posizione è sbagliata, si incrementa N e si ricomincia con M=0
  - se X==Y la posizione potrebbe essere giusta, si incrementa M per passare al prossimo carattere

2) Scrivere il programma **main** che:

1. legge la prima stringa (di lunghezza massima 100 caratteri)
2. legge la seconda stringa (della stessa lunghezza massima)
3. chiama la funzione **cercaSottostringa** passando gli indirizzi delle due stringhe
4. stampa il risultato

NOTA: la funzione **cercaSottostringa** che scandisce le due stringhe può essere realizzata con un ciclo (**18 punti**) oppure il ciclo può essere simulato con la ricorsione (**30 punti**)

ESEMPI

testo = "Topolino va al **mare** con Topolina e le nipotine Emy, Ely, Evy"

query = "**mare**"

risultato = 15

testo = "**granturco**"

query = "**papero**"

risultato = -1

Soluzione

```
.globl main
.data
testo:  .space 101
query:  .space 101
.text
.eqv    N,      $s0
.eqv    M,      $s1
main:   li      $v0, 8
        li      $a1, 100
        la      $a0, testo
        syscall
        li      $v0, 8
        li      $a1, 100
        la      $a0, query
        syscall
        la      $a0, testo
        la      $a1, query
        li      N, 0
        li      M, 0
        jal     cercaSottostringa
        move    $a0, $v0
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall

cercaSottostringa:
        add     $t0, $a1, M
        lb      $t0, ($t0)
        seq     $t1, $t0, $zero
        seq     $t2, $t0, '\n'
        or      $t1, $t1, $t2
        beqz    $t1, query_non_finita
        move    $v0, N
        jr      $ra
query_non_finita:
        add     $t1, $a0, N
        add     $t1, $t1, M
        lb      $t1, ($t1)
        seq     $t2, $t1, $zero
        seq     $t3, $t1, '\n'
        or      $t2, $t2, $t3
        beqz    $t2, testo_non_finito
        li      $v0, -1
        jr      $ra
testo_non_finito:
        beq     $t0, $t1, avanti
        addi    N, N, 1
        li      M, 0
        subi    $sp, $sp, 4
        sw      $ra, 0($sp)
        jal     cercaSottostringa
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
avanti: addi    M, M, 1
        subi    $sp, $sp, 4
        sw      $ra, 0($sp)
        jal     cercaSottostringa
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
```

Esame di Architetture – Prof. Sterbini – 9/6/17 – compito B

Parte 3 (assembler)

Esercizio 5 (18 punti se iterativo, 30 se ricorsivo).

Sia data una lista rappresentata da un vettore in cui il primo elemento si trova ad indice 0 ed ogni elemento può contenere:

- il valore -1 per indicare che la lista è finita e non ci sono altri elementi
- oppure l'indice del prossimo elemento della lista

Esempio

il vettore	1, 4, 2, 6, 9, -1, 3, 5, 7, 8, 10
corrisponde alla lista di posizioni	0 → 1 → 4 → 9 → 8 → 7 → 5 → -1

1) Scrivere una funzione **contaPariInLista** in linguaggio assemblativo che, ricevendo l'indirizzo di un vettore contenente una lista

che inizia dall'indice 0, calcola e torna come risultato il numero di elementi della lista che si trovano ad indice pari.

Un esempio di algoritmo potrebbe essere:

- se l'indice corrente è -1 la lista è finita, si torna 0 (caso base)
- si contano gli elementi in posizione pari a partire dal prossimo elemento (caso risorsivo)
- se l'elemento corrente è pari si somma 1 al risultato
- si torna il risultato

2) Scrivere il programma **main** che:

1. legge il numero N di elementi da inserire in un vettore (max 20)
2. legge gli N interi e li mette in un vettore
3. chiama la funzione **contaPariInLista** passando l'indice iniziale 0 e l'indirizzo del vettore
4. stampa il risultato

NOTA: la funzione **contaPariInLista** può essere realizzata con un ciclo (**18 punti**) oppure il ciclo può essere simulato con la ricorsione (**30 punti**)

Esempio

il vettore	1, 4, 2, 6, 9, -1, 3, 5, 7, 8, 10
corrisponde alla lista di posizioni	**0** → 1 → **4** → 9 → **8** → 7 → 5 → -1
il risultato sarà	**3**

Soluzione

```
.data
vettore:      .word 0:20
.globl main
.text
main:
      li     $v0, 5
      syscall
      move   $t0, $v0      # conteggio
      li     $t2, 0        # offset
loop: li     $v0, 5
      syscall
      sw     $v0, vettore($t2)
      addi   $t2, $t2, 4
      subi   $t0, $t0, 1
      bgtz   $t0, loop
      li     $a0, 0
      la     $a1, vettore
      jal    contaPariInLista
      move   $a0, $v0
      li     $v0, 1
      syscall
      li     $v0, 10
      syscall

contaPariInLista:
      bne    $a0, -1, caso_ricorsivo
      li     $v0, 0        # se fine lista torno
      jr     $ra

caso_ricorsivo:
      subu   $sp, $sp, 8   # salvo su stack
      sw     $ra, 0($sp)   # $ra
      sw     $a0, 4($sp)   # e $a0 che mi serve
dopo
      sll    $t0, $a0, 2   # offset = i * 4
      add    $t0, $t0, $a1 # indirizzo+offset
      lw     $a0, ($t0)    # prossimo indice
      jal    contaPariInLista # resto della lista
      lw     $ra, 0($sp)   # ripristino $ra
      lw     $a0, 4($sp)   # e $a0
      addu   $sp, $sp, 8   # e disalloco
      andi   $t0, $a0, 1   # prendo il LSBit
      xori   $t0, $t0, 1   # lo nego
      add    $v0, $v0, $t0 # e lo sommo al
conteggio
      jr     $ra           # tono al chiamante
```