Esame di Architetture – Canale AL – Prof. Sterbini – 8/7/13 – Compito A

Esercizio 1A. Si ha il dubbio che in una partita di CPU a ciclo di clock singolo (vedi sul retro) la Control Unit sia rotta, producendo il segnale di controllo **RegWrite** attivo **anche quando** è attivo il segnale di controllo **MemWrite**.

a) Si indichino *qui sotto* quali delle istruzioni base (**lw, sw, add, sub, and, or, xor, slt, beq, j**) funzioneranno male e perché.

Svolgimento

L'unica istruzione che ha MemWrite=1 è la sw, che però ha RegWrite=0, se invece la CPU è rotta RegWrite=1, e l'istruzione sw, oltre a memorizzare il valore in memoria, memorizza nel registro destinazione il valore trasmesso dal MUX MemToReg.
Assumendo che MemToReg=0 e che RegDst=0 (vedi sotto), nel registro destinazione verrà memorizzato l'indirizzo (registro base + offset) usato dalla sw. Dato che la sw è una istruzione di tipo I il registro destinazione sarà quello dal quale è preso il dato da memorizzare.

b) si scriva *qui sotto* un breve programma assembly MIPS (senza pseudoistruzioni) che termina valorizzando il registro $s0 con il valore 1 se il processore è guasto, altrimenti con 0. Si assume che RegDst sia asserito solo per le istruzioni di tipo R e che MemToReg sia asserito solo per l'istruzione lw.

Svolgimento

E' sufficiente sfruttare la modifica del registro che contiene il dato da memorizzare facendo in modo che l'indirizzo calcolato sia il valore 1. Quindi il programma potrebbe essere:

```
add $s0, $zero, $zero   # azzero $s0
sw $s0, 1($zero)        # se sw non funziona scrive 1 in $s0
```

Esercizio 2A. Considerate l'architettura MIPS a ciclo singolo in figura (diagramma sul retro).
Si vuole aggiungere l'istruzione di tipo I **`bali rd, label`** (branch and link immediato) che salta all'indirizzo della label (relativo come nei beq) e salva nel registro **rd** l'indirizzo della prossima istruzione.

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

2) indicate sul diagramma i segnali di controllo necessari a realizzare l'istruzione

3) supponendo che l'accesso alle memorie impieghi 133ns, l'accesso ai registri 66ns, le operazioni dell'ALU 200ns, 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.

Svolgimento

L'istruzione bali deve:
- PC = PC + 4 + 4*(offset esteso a 32 bit) (cioè lo stesso che fa un salto condizionato se viene preso)
- salvare nel registro il valore PC+4

Quindi servono:
- la Control Unit deve riconoscere l'istruzione bali e produrre un segnale (che chiameremo Bali)
- dopo l'AND tra il segnale Branch e il segnale Zero bisogna aggiungere un OR che attiva il salto se Bali=1
- l'uscita PC+4 del primo adder deve poter essere inserita in un registro, quindi serve un MUX all'ingresso del blocco registri (Dati da scrivere ...) che permette di inserire il PC+4 se Bali=1 altrimenti fa passare i soliti dati se Bali=0

Tempi:
Se assumiamo che i due Adder che calcolano i nuovi PC impieghino tempo 0, il tempo necessario è 199ns:
- 133ns per il fetch dell'istruzione
- 66ns per la memorizzazione del PC+4 nel registro

Se invece assumessimo che gli adder impiegano 200ns come una ALU allora il tempo sarebbe 400ns:
- 200ns PC+4 e contemporaneamente fetch della istruzione
- 200ns PC=(PC+4)+(4*(offset esteso)) e contemporaneamente scrittura di PC+4 nel registro

Esame di Architetture – Canale AL – Prof. Sterbini – 8/7/13 – Compito A

Esercizio 3A (16 punti). Considerate l'architettura MIPS con pipeline mostrata in figura (sul retro) ed il frammento di programma qui a destra che somma gli elementi a coordinate x+y pari di una matrice di word.
Indicate:
1) tra quali istruzioni sono presenti data hazard,
2) tra quali istruzioni sono presenti control hazard,
3) tra quali istruzioni sono necessari stalli (data e control)
4) quanti cicli di clock sono necessari a eseguire il programma
5) quanti ne sarebbero necessari se il forwarding non esistesse
6) riordinate le istruzioni per ridurre al massimo gli stalli (mantenendo invariata la sua semantica)
7) calcolate quanti cicli di clock sono necessari a eseguire il programma così ottimizzato

```
.globl main
.data
matrice:    .space 400 # 100 words
DIM:        .word 10  #lato della matrice
.text
main:       xor  $s4, $s4, $s4 # somma=0
            xor  $s2, $s2, $s2 # x=0
            xor  $s0, $s0, $s0 # y=0
            lw   $s1, DIM
loopY:      beq  $s0, $s1, end
loopX:      bge  $s2, $s1, nextY
            mul  $t1, $s0, $s1    # y*DIM
            add  $t1, $t1, $s2    # +x
            muli $t1, $t1, 4      # *4
            lw   $t2, matrice($t1)
            add  $s4, $s4, $t2
            addi $s2, $s2, 2      # x+=2
            j loopX
nextY:      andi $s2, $s0, 0x0001 # x%=2
            addi $s0, $s0, 1      # y+=1
            # qui x=1 se y pari, else x=0
            j loopY
end:        li $v0, 1    # per la syscall
            add $a0, $s4, $zero   # move
```

Svolgimento
Sono presenti **data hazard** ogni qualvolta una istruzione produce un valore e lo mette in un registro (evidenziati in giallo) da cui verrà letto in una delle 2 istruzioni successive.
Si noti che esistono due data hazard tra gli incrementi dei contatori e i salti condizionati dei cicli.
Sono presenti **control hazard** quando un salto condizionato esegue il salto (evidenziati in verde).

La maggior parte dei data hazard viene eliminata dal forwarding, tranne per le due istruzioni:
- lw $s1, DIM — che ha bisogni di **2 stalli** prima della istruzione successiva (e anche di 1 per quella dopo ma i 2 già inseriti sono sufficienti)
- lw $t2, matrice($t1) — che ha bisogno di **1 stallo** prima della istruzione successiva

Per eliminare questi stalli basta spostare:
- lw $s1, DIM — di almeno 2 istruzioni indietro
- addi $s2, $s2, 4 — subito dopo lw $t2, matrice($t1) cioè al posto dello stallo

I due control hazard (salti condizionati) inseriscono **1 stallo** (non eliminabile visto che non abbiamo il delayed branch) quando fanno il salto.

Se il forwarding non funziona sono presenti **2 stalli** per ogni data hazard tranne per gli incrementi dei loop in cui uno dei due stalli è occupato dalla istruzione jump.

Svolgendo il conteggio del numero di colpi di clock si ottiene: (data e control hazard)

Con forwarding:	9 + 2 + 10 (5 * (8 + 1) + 1 + 5) + 1 + 2	= 524	
Ottimizzato:	9 + 0 + 10 (5 * (8 + 0) + 1 + 5) + 1 + 2	= 472	(tolgo gli stalli sui dati)
Senza forwarding:	9 + 2 + 10 (5 * (8 + 9) + 1 + 5 + 1) + 1 + 2	= 934	(stalli su tutti i data hazard)

Dal che si vede che il programma ottimizzato impiega circa il 50% del tempo rispetto a quello ottimizzato

Esercizio 4A (14 punti). Si consideri un sistema dotato di due livelli di cache: CPU <=> L1 <=> L2.
- L1 è una cache 4-way set-associative con 4 linee e blocchi grandi 4 word e strategia di rimpiazzo LRU.
- L2 è una cache 2-way set-associative con 8 linee e blocchi grandi 16 word e strategia di rimpiazzo LRU.

1) Supponendo che gli indirizzi siano da 32 bit (indirizzamento al byte) e che all'inizio nessuno dei dati sia in memoria, indicate quali dei seguenti accessi in memoria sono hit o miss in ciascuna delle due cache:

10 28 200 540 330 210 220 535 190 560 30 36 12 520

2) calcolate le dimensioni in bit delle due cache L1, L2.

3) assumendo che il processore vada a 1Ghz con 1 CPI (Clock Per Instruction), che gli accessi in memoria impieghino 100ns, che gli hit nella cache L1 impieghino 1ns e gli hit nella cache L2 impieghino 20ns, calcolate il tempo totale e il CPI per questa sequenza di accessi.

Svolgimento

Per calcolare gli Hit e Miss di una cache sappiamo che:
- l'offset O dell'indirizzo A nel blocco è — O=A % dimensione blocco
- il numero di blocco **b** corrispondente all'indirizzo **A** è — b=A // dimensione blocco
- l'indice della linea **L** in cui verrà messo il blocco è — L=b % numero di linee
- il Tag **T** che identifica il blocco nella linea L è — T=b // numero di linee

La cache L1 ha dimensione blocco = 4*4 = 16 bytes e numero di linee 4 a 4 vie
La cache L2 ha dimensione blocco = 16*4 = 64 bytes e numero di linee 8 a 2 vie

A	**10**	**28**	**200**	**540**	**330**	**210**	**220**	**535**	**190**	**560**	**30**	**36**	**12**	**520**
b1	0	1	12	33	20	13	13	33	11	35	1	2	0	32
L1	0	1	0	1	0	1	1	1	3	3	1	2	0	0
T1	0	0	3	8	5	3	3	8	2	8	0	0	0	8
H/M	M	M	M	M	M	M	H	H	M	M	H	M	H	M
b2	0	0	3	8	5	3	-	-	2	8	-	0	-	8
L2	0	0	3	0	5	3	-	-	2	0	-	0	-	0
T2	0	0	0	1	0	0	-	-	0	1	-	0	-	1
H/M	M	H	M	M	M	H	-	-	M	H	-	H	-	H

Il numero di vie è sufficiente in entrambi i casi e per questa sequenza di accessi non perdiamo nessuna Hit.
Il tempo necessario a eseguire questa sequenza di accessi è dato da:
- 1ns per ogni Hit in L1 (e non viene fatta nessuna richiesta a L2)
- 20ns per ogni Hit in L2
- 100ns per ogni accesso in memoria, ovvero per ogni miss della cache L2

Quindi: 5*100ns + 5*50ns + 4*1ns = **604ns**
Dato che la CPU ha il clock a 1GHz e CPI=1 il tempo di una istruzione è 1ns
Il CPI di questa successione di accessi è quindi 604ns / 14*1ns = **43.14**

Esame di Architetture – Canale AL – Prof. Sterbini – 8/7/13 – Compito B

Esercizio 1B. Si ha il dubbio che in una partita di CPU MIPS come quella in figura la Control Unit sia rotta, producendo il segnale di controllo **MemToReg** attivo **se e solo se** è attivo il segnale di controllo **MemWrite**.

a) Si indichino *qui sotto* quali delle istruzioni base (**lw, sw, add, sub, and, or, xor, slt, beq, j**) funzioneranno male e perché.

Svolgimento

L'unica istruzione che ha MemWrite=1 è la sw, mentre l'unica istruzione che ha MemToReg=1 è la lw, quindi:
- sw avrà anche MemToReg=1 ma questo non ne modifica il comportamento, visto che RegWrite=0

- lw avrà MemToReg=0 per cui il valore letto dalla memoria verrà perso, e nel registro destinazione finirà l'indirizzo della locazione letta invece che il valore.

b) si scriva *qui sotto* un breve programma assembly MIPS (senza pseudoistruzioni) che termina valorizzando il registro $s0 con il valore 1 se il processore è guasto, altrimenti con 0. Si assume che RegDst sia asserito solo per le istruzioni di tipo R e che MemtoReg sia asserito solo per l'istruzione lw.

Svolgimento

Possiamo sfruttare il fatto che la lw mette nel registro destinazione l'indirizzo invece che il valore:

```
sw $zero, 1($zero)      # memorizzo 0 nella locazione 1
lw $s0, 1($zero)        # leggo 0 se la CPU è OK, altrimenti leggo 1
```

Esercizio 2B. Considerate l'architettura MIPS a ciclo singolo in figura (sul retro). Si vuole aggiungere l'istruzione di tipo I **`baddz rs, rt, label`** che incrementa il registro **`rs`** del valore contenuto nel registro **`rt`** (rs=rs+rt) e salta alla destinazione se la somma è zero.

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

2) indicate sul diagramma i segnali di controllo necessari a realizzare l'istruzione

3) supponendo che l'accesso alle memorie impieghi 100ns, l'accesso ai registri 50ns, le operazioni dell'ALU 150ns, e ignorando gli altri ritardi di propagazione dei segnali, indicate sul diagramma qual è la durata del ciclo di clock per permettere l'esecuzione anche della nuova istruzione.

Svolgimento

L'istruzione baddz fa:

- lo stesso di una beq ma sommando gli argomenti invece che sottraendoli

- memorizza il risultato della somma nel registro rs

Quindi è necessario:

- aggiungere alla Control Unit un segnale che indica che è stata riconosciuta l'istruzione, che chiamiamo Baddz

- mandare alla ALU il comando di fare una somma, come quando si esegue un accesso in memoria

- mettere Branch=1 in modo che il segnale Zero=1 faccia fare il salto se il risultato della somma è 0

- per memorizzare il risultato della somma in rs (invece che rt o rd) bisogna aggiungere un MUX (che chiamerò RsSrc) all'ingresso della porta che indica il registro destinazione al blocco registri. Il MUX serve a fornire il campo rs della istruzione come indice del registro destinazione. Come segnali dobbiamo mettere: MemToReg=0, RegWrite=1, RsSrc=1

Per i tempi servono in totale 350ns:

- Fetch: 100ns
- Decode e lettura dei registri: 50ns
- Somma (ALU): 150ns
- Write Back nel registro rs: 50ns

Esame di Architetture – Canale AL – Prof. Sterbini – 8/7/13 – Compito B

Esercizio 3B (16 punti). Si consideri l'architettura MIPS con pipeline mostrata in figura (sul retro) ed il frammento di programma qui a destra che calcola la somma degli elementi dispari di un vettore di 100 elementi.
Si indichino:
1) tra quali istruzioni sono presenti data hazard,
2) tra quali istruzioni sono presenti control hazard,
3) tra quali istruzioni sono necessari stalli (data e control)
4) quanti cicli di clock sono necessari a eseguire il programma
5) quanti ne sarebbero necessari se il forwarding non esistesse
6) riordinate le istruzioni per ridurre al massimo gli stalli (mantenendo invariata la sua semantica)
7) quanti cicli di clock sono necessari a eseguire il programma così ottimizzato

```
.globl   main
.data
vettore: .word ... (100 valori) ...
DIM:     .word 100
.text
main:    xor  $s2, $s2, $s2 # somma=0
         lw   $s0, DIM      # i=100
         subi $s0, $s0, 1   # i= 99
         muli $s0, $s0, 4   # i=396
loop:    lw   $t1, matrice($s0)
         andi $t2, $t1, 1   # val%2
         beq  $t2, $zero, pari
         add  $s2, $s2, $t1 # dispari
pari:    subi $s0, $s0, 4   # el.prec.
         blez $s0, end      # se i<0
         j loop
end:     add  $a0, $s2, $zero # move
         li   $v0, 1   # args syscall
```

Svolgimento
Sono presenti **data hazard** ogni qualvolta una istruzione produce un valore e lo mette in un registro (evidenziati in giallo) da cui verrà letto in una delle 2 istruzioni successive.
Sono presenti **control hazard** quando un salto condizionato esegue il salto (evidenziati in verde).

La maggior parte dei data hazard viene eliminata dal forwarding, tranne per le quattro istruzioni:
- `lw $s0, DIM` che ha bisogno di **1 stallo** prima della istruzione successiva
- `lw $t1, matrice($s0)` che ha bisogno di **1 stallo** prima della istruzione successiva
- `andi $t2, $t1, 1` che ha bisogno di **1 stallo** prima della istruzione successiva che è un branch
- `subi $s0, $s0, 4` che ha bisogno di **1 stallo** prima della istruzione successiva che è un branch

Di questi stalli si riesce ad eliminarne 3 spostando due istruzioni:
- `lw $s0, DIM` prima dello `xor`
- `subi $s0, $s0, 4` dopo la `lw $t1, matrice($s0)` o la `andi $t2, $t1, 1`

Il due control hazard (salti condizionati) inseriscono **1 stallo** (non eliminabile visto che non abbiamo il delayed branch) quando fanno il salto. Si noti che lo stallo del primo salto (test di parità) ha la stessa durata della istruzione che viene saltata, quindi il loop ha sempre la stessa durata, indipendentemente dai valori presenti nella matrice.

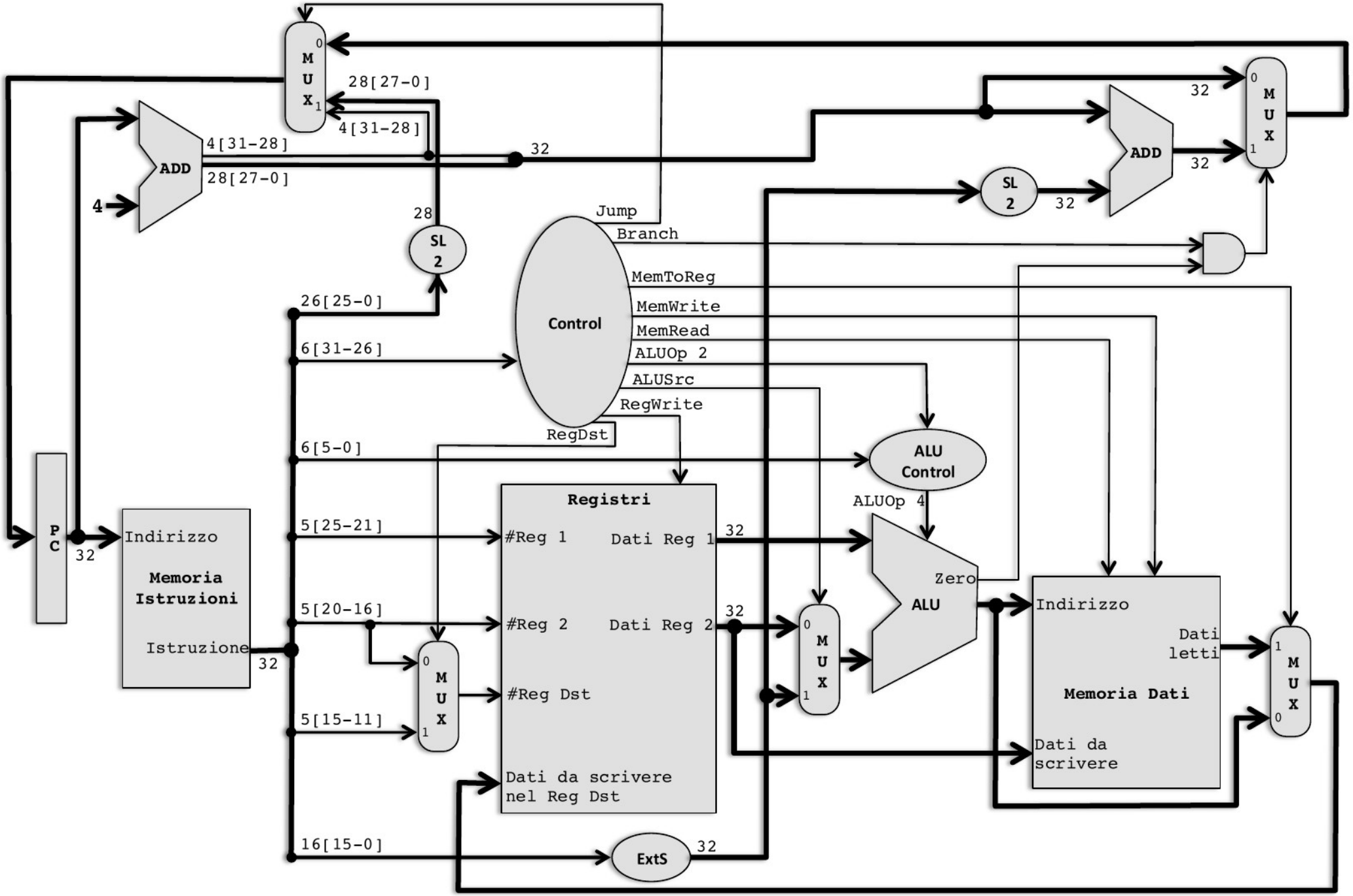
Se il forwarding non funziona sono presenti **2 stalli** per ogni data hazard.

Svolgendo il conteggio del numero di colpi di clock si ottiene:

Con forwarding:	9 + 1 + 99 * (7 + 3) + 1 + 2	= 1003	
Ottimizzato:	9 + 0 + 99 * (7 + 1) + 1 + 2	= 804	(tolgo 3 stalli)
Senza forwarding:	9 + 6 + 99 * (7 + 6) + 1 + 2	= 1305	(aggiungo gli stalli su tutti i data hazard)

Quindi senza forwarding si impiega il 30% di tempo in più

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



Esercizio 4B (14 punti). Si consideri un sistema dotato di due livelli di cache: CPU <=> L1 <=> L2.
- L1 è una cache 2-way set-associative con 16 linee e blocchi grandi 2 word e strategia di rimpiazzo LRU.
- L2 è una cache 1-way set-associative con 4 linee e blocchi grandi 8 word e strategia di rimpiazzo LRU.

1) Supponendo che gli indirizzi siano da 32 bit (indirizzamento al byte) e che all'inizio nessuno dei dati sia in memoria, indicate quali dei seguenti accessi in memoria sono hit o miss in ciascuna delle due cache:

10 28 200 540 330 210 220 535 190 560 30 36 12 520

2) calcolate le dimensioni in bit delle due cache L1, L2.

3) assumendo che il processore vada a 2Ghz con 2 CPI (Clock Per Instruction), che gli accessi in memoria impieghino 150ns, che gli hit nella cache L1 impieghino 1ns e gli hit nella cache L2 impieghino 25ns, calcolate il tempo totale impiegato e il CPI per questa sequenza di accessi.

Svolgimento

Per calcolare gli Hit e Miss di una cache sappiamo che:

- l'offset O dell'indirizzo A nel blocco è — O=A % dimensione blocco
- il numero di blocco **b** corrispondente all'indirizzo **A** è — b=A // dimensione blocco
- l'indice della linea **L** in cui verrà messo il blocco è — L=b % numero di linee
- il Tag **T** che identifica il blocco nella linea L è — T=b // numero di linee

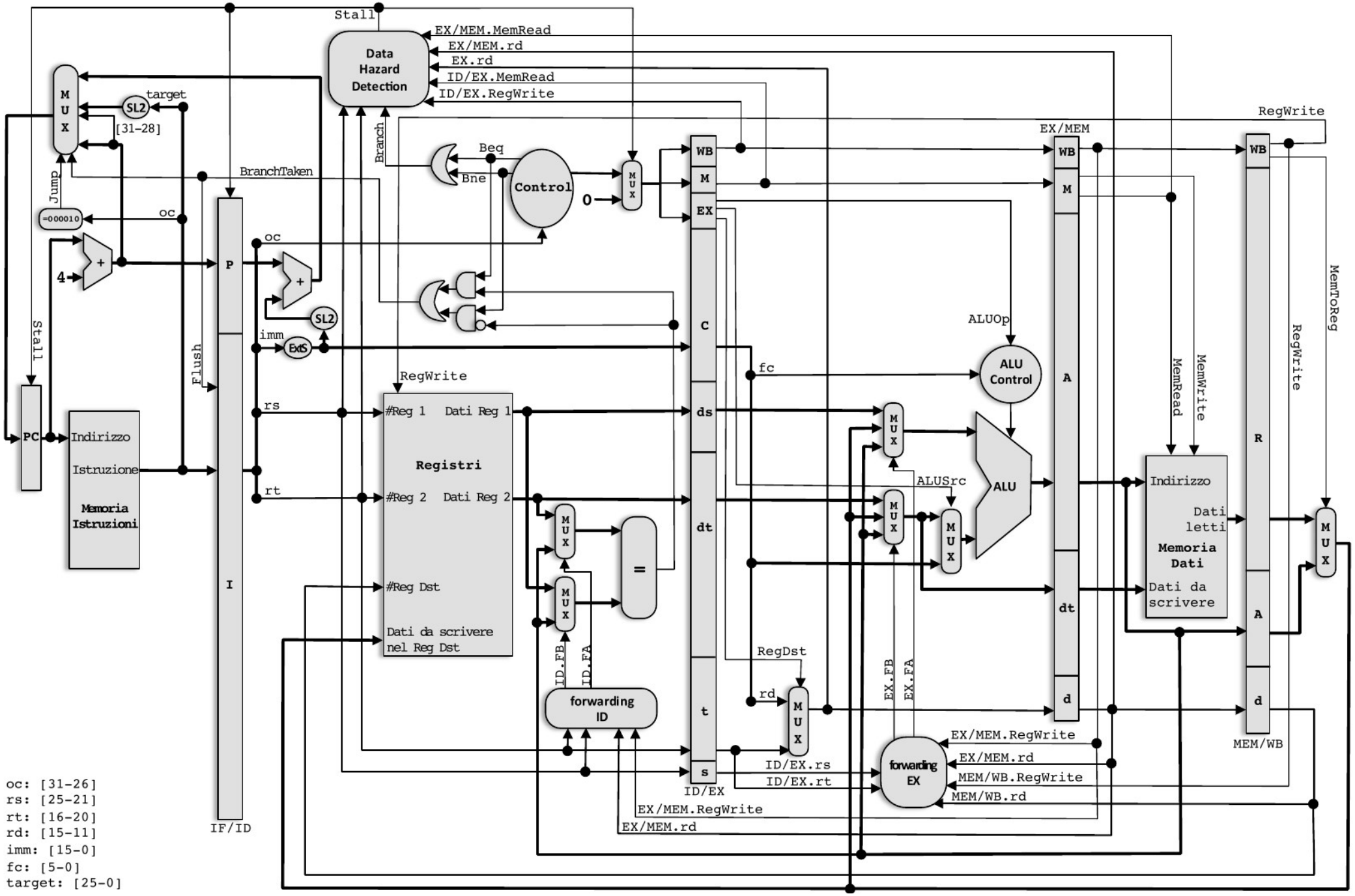
La cache L1 ha dimensione blocco = 2*4 = 8 bytes e numero di linee 16 a 2 vie
La cache L2 ha dimensione blocco = 8*4 = 32 bytes e numero di linee 4 a 1 via (direct mapped)

A	**10**	**28**	**200**	**540**	**330**	**210**	**220**	**535**	**190**	**560**	**30**	**36**	**12**	**520**
b1	1	3	25	67	41	26	27	66	23	70	3	4	1	65
L1	1	3	9	3	9	10	11	2	7	6	3	4	1	1
T1	0	0	1	4	2	1	1	4	1	4	0	0	0	4
H/M	M	M	M	M	M	M	M	M	M	M	H	M	H	M
b2	0	0	6	16	10	6	6	16	5	17	-	1	-	16
L2	0	0	2	0	2	2	2	0	1	1	-	1	-	0
T2	0	0	1	4	2	1	1	4	1	4	-	0	-	4
H/M	M	H	M	M	M	M	H	H	M	M	-	M	-	H

Il numero di vie di L2 non è sufficiente per cui in L2 perdiamo la Hit indicata dal colore viola.
Il tempo necessario a eseguire questa sequenza di accessi è dato da:
- 1ns per ogni Hit in L1 (e non viene fatta nessuna richiesta a L2)
- 25ns per ogni Hit in L2
- 150ns per ogni accesso in memoria, ovvero per ogni miss della cache L2

Quindi: 8*150ns + 4*25ns + 2*1ns = **1302ns**
Dato che la CPU ha il clock a 2GHz e CPI=2 il tempo di una istruzione è 1ns
Il CPI di questa successione di accessi è quindi 1302ns / 14*1ns = **93.00**

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]

Esame di Architetture – Canale AL – Prof. Sterbini – 8/7/13 – Compito A

Esercizio 5A.

1) Si realizzi la funzione **ricorsiva** che calcola il massimo comun divisore (MCD) con l'algoritmo di Eulero basato sulle differenze definita qui a destra:

$$MCD(x,y)=\begin{cases} x & \text{se x = y} \\ MCD(x-y,y) & \text{se x > y} \\ MCD(x,y-x) & \text{se x < y} \end{cases}$$

2) Si realizzi un programma che usa la funzione MCD per calcolare il minimo comune multiplo (mcm) tra 2 valori positivi:

- legge due valori interi positivi X, Y

- calcola e stampa il minimo comune multiplo tra X, e Y, ovvero mcm(X,Y) = X * Y / MCD(X, Y)

Esempi del MCD:

se X=2 e Y=3	il risultato è	MCD(2,3) = MCD(2,1) = MCD(1,1) = **1**
se X=30 e Y=20	il risultato è	MCD(30,20) = MCD(10,20) = MCD(10,10) = **10**
se X=15 e Y=81	il risultato è	MCD(15,81) = MCD(15,66) = MCD(15,51) = MCD(15,36) =
		= MCD(15,21) = MCD(15,6) = MCD(9,6) = MCD(3,6) = MCD(3,3) = **3**

Esempi dell'output:

se X=27, Y=15	il risultato è	mcm(27,15) = 27 * 15 / MCD(27, 15) = 27 * 15 / 3 = **135**
se X=54, Y=30	il risultato è	mcm(54,30) = 54 * 30 / MCD(54, 30) = 54 * 30 / 6 = **270**
se X=54, Y=72	il risultato è	mcm(54,72) = 54 * 72 / MCD(54, 72) = 54 * 72 / 18 = **216**

Svolgimento

La funzione principale da sviluppare è MCD, che potrebbe essere scritta come segue

```
.globl main
.text
MCD:        subi $sp, $sp, 4            # alloco 1 word sullo stack
            sw $ra, 0($sp)              # ci salvo $ra
            bneq $a0, $a1, X_NEQ_Y      # se X != Y
            move $v0, $a0               # altrimenti X=Y e torno X in $v0
            j FINE                      # salto alla chiusura della funzione e pop dallo stack
X_NEQ_Y:    blt $a0, $a1, X_LT_Y        # se X<Y
            sub $a0, $a0, $a1           # altrimenti X>Y e calcolo X -= Y
            j RICHIAMA                  # passo alla chiamata ricorsiva
X_LT_Y:     sub $a1, $a1, $a0           # altrimenti X<Y e calcolo Y -= X
RICHIAMA:   jal MCD                     # chiamo MCD ricorsivamente
FINE:       lw $ra, 0($sp)              # all'uscita recupero il $ra dallo stack
            addi $sp, $sp, 4            # disalloco 1 word
            jr $ra                      # torno il valore in $v0
```

Esame di Architetture – Canale AL – Prof. Sterbini – 8/7/13 – Compito B

Esercizio 5B.

1) Si realizzi la funzione assembler **ricorsiva** che calcola il massimo comun divisore (GCD) di tre numeri X, Y, Z usando l'algoritmo di Eulero basato sui resti, definita qui a destra (si esegue la prima condizione valida)

$$GCD(x,y)=\begin{cases} x & \text{se y è zero} \\ GCD(y,x) & \text{se x < y} \\ GCD(y, x \bmod y) & \text{altrimenti} \end{cases}$$

2) si realizzi il programma assembler che usa la funzione GCD per calcolare il GCD di 3 valori positivi, e che:

- legge tre valori interi positivi X, Y, Z

- calcola e stampa il minimo comune multiplo tra X, Y e Z, ovvero GCD(GCD(X, Y), Z)

Esempi del GCD:

se X=2 e Y=3	il risultato è	GCD(2,3) = GCD(3,2) = GCD(3,1) = GCD(1,0) = **1**
se X=30 e Y=20	il risultato è	GCD(30,20) = GCD(20,10) = GCD(10,0) = **10**
se X=15 e Y=81	il risultato è	GCD(15,81)=GCD(81,15)=GCD(15,6)=GCD(6,3)=GCD(3,0)=**3**

Esempi dell'output:

se X=27, Y=150 e Z=600	il risultato è	GCD(GCD(27,150),600) = GCD(3,600) = **3**
se X=54, Y=150 e Z=600	il risultato è	GCD(GCD(54,150),600) = GCD(6,600) = **6**
se X=54, Y=90 e Z=630	il risultato è	GCD(GCD(54,90),630) = GCD(18,630) = **18**

Svolgimento

La funzione principale da sviluppare è GCD, che potrebbe essere scritta come segue

```
.globl main
.text
GCD:        subi $sp, $sp, 4          # alloco 1 word sullo stack
            sw $ra, 0($sp)            # ci salvo $ra
            bnez $a1, Y_NONZ          # se Y!=0 vado avanti
            move $v0, $a0             # altrimenti Y=0 e torno X in $v0
            j FINE                    # salto alla chiusura della funzione e pop dallo stack
Y_NONZ:     bgt $a0, $a1, X_GT_Y      # se X>Y vado avanti
            move $s0, $a1             # altrimenti X<Y e scambio X e Y
            move $a1, $a0             # resto dello scambio
            move $a0, $s0
            j RICHIAMA                # passo alla chiamata ricorsiva
X_GT_Y:     rem $a0, $a0, $a1         # altrimenti X>Y e calcolo X mod Y e lo metto in X
RICHIAMA:   jal GCD                   # chiamo GCD ricorsivamente
FINE:       lw $ra, 0($sp)            # all'uscita recupero il $ra dallo stack
            addi $sp, $sp, 4          # disalloco 1 word
            jr $ra                    # torno il valore in $v0
```