# Esame di Architetture – Canale MZ – Prof. Sterbini – 30/6/15

## *Parte 1 (per chi non ha superato l'esonero – 1 ora)*

**Esercizio 1 (14 punti).** In una partita di CPU a ciclo di clock singolo (vedi sul retro) la Control Unit potrebbe essere rotta, producendo il segnale di controllo **Branch** attivo se e solo se NON è attivo **RegDst**.
Si assume che **RegDst** sia asserito solo per le istruzioni di tipo R e che **MemRead** e **MemToReg** siano asseriti solo per l'istruzione lw.

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

**Soluzione**
le uniche configurazioni prodotte dal guasto sono RegDst=0 e Branch=1 oppure RegDst=1 e Branch=0 quindi funzioneranno male le istruzioni che hanno RegDst=0 e Branch=0 (lw e sw) oppure RegDst=1 e Branch=1 (nessuna). Le istruzioni R e beq e j funzioneranno bene. Le lw e sw invece, oltre a eseguire l'accesso in memoria correttamente, si comporteranno come un salto condizionato, ovvero salteranno alla istruzione che si trova X istruzioni prima o dopo di PC+4 solo se l'ALU produrrà il segnale Zero.

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.
Se lo ritenete necessario potete usare la sezione **.data** per inizializzare il contenuto della memoria.

**Soluzione**

Dobbiamo far eseguire un salto ad una lw o a una sw, quindi usare un registro base ed una costante immediata, che sommate diano zero. Il salto sarà di un numero di istruzioni uguale alla parte immediata. Usiamo 1 come parte immediata per saltare l'istruzione successiva alla lw, e mettiamo nel registro base -1 in modo che la somma sia 0.

```
nor $a0, $zero, $zero    # inizializzo $a0 con 1111111111 che in Ca2 è -1
lw $a1, 1($a0)           # se errato salto la prossima istruzione
j giusto
li $s0, 1                # è errato
j fine
giusto:
li $s0, 0                # è giusto
fine:
```

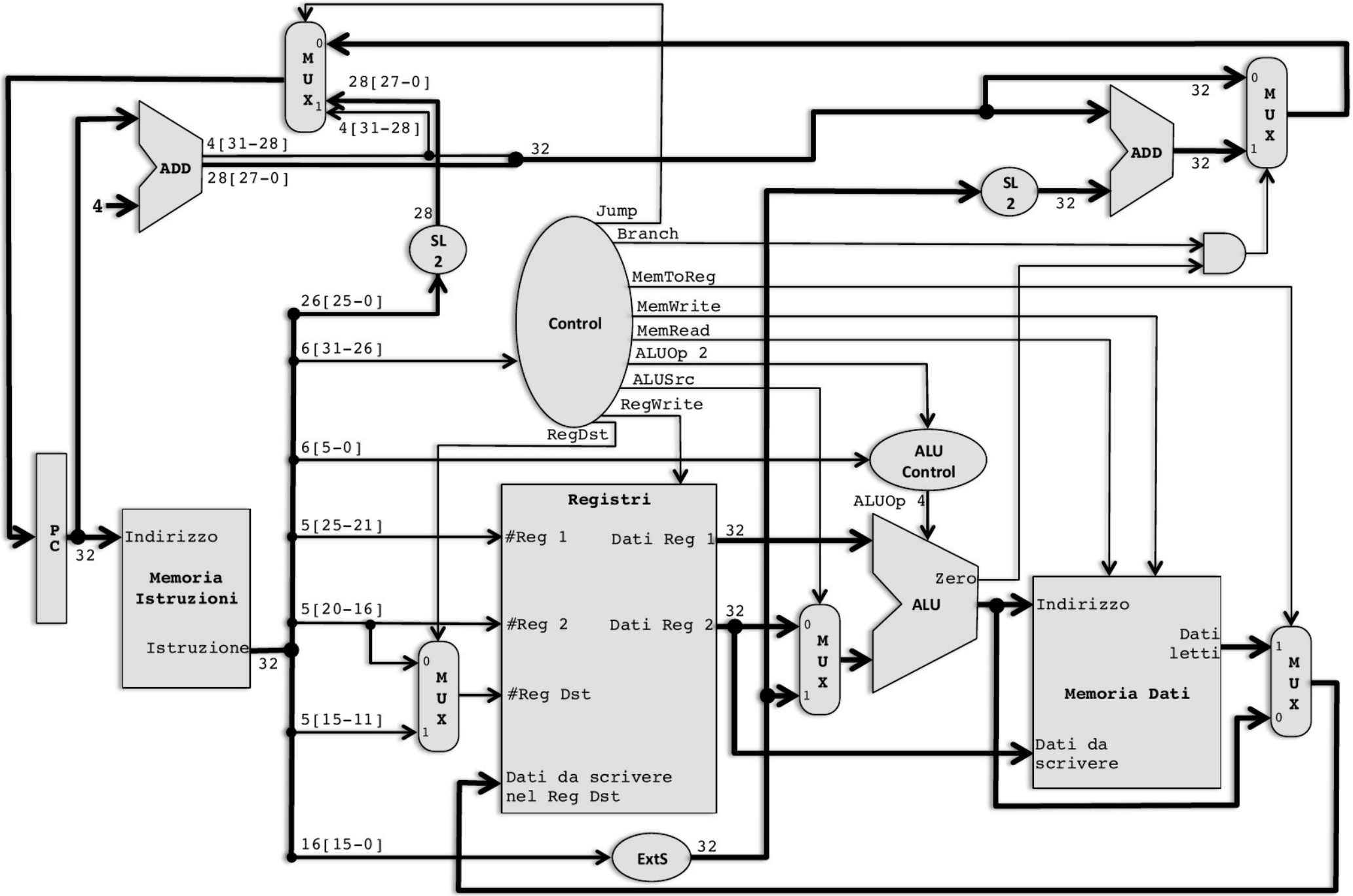
**Esercizio 2 (16 punti).** Considerate l'architettura MIPS a ciclo singolo in figura (diagramma sul retro).
Vogliamo aggiungere l'istruzione di tipo I **`jaddl rs, rt`** (jump and link to sum) che esegue un salto incondizionato all'indirizzo (assoluto) corrispondente alla somma $rs+$rt. Inoltre salva in $ra l'indirizzo della istruzione successiva (PC+4).

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 **75ns**, 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 della nuova istruzione e se la durata totale del ciclo di clock necessario è aumentata rispetto alla CPU senza la nuova istruzione

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



**Soluzione**

1) L'istruzione deve realizzare contemporaneamente:

PC ← $rs + $rt (la somma la posso realizzare con l'ALU)

$ra ← PC + 4 (la somma è già presente)

Non servono nuove unità funzionali perché la prima somma la posso realizzare con la ALU e la seconda è già fatta dal primo sommatore.

Vanno aggiunti i due trasferimenti di dati:

- dalla uscita della ALU al PC

- dalla uscita del sommatore PC+4 all'ingresso di scrittura del blocco registri

Inoltre bisogna indicare l'indice del registro $ra (il numero 31) sulla porta del registro di destinazione del blocco dei registri

Quindi sono necessari 3 MUX da attivare usando un nuovo segnale di controllo (jaddl) generato dalla CU:

- il primo subito a monte del **PC** per permettere di inserire l'output della ALU se il segnale jaddl è asserito

- il secondo subito a monte della porta **dati da scrivere** dei registri, che permette di inserire il valore PC+4

- il terzo subito a monte della porta **#regDst** del blocco registri, che permette di inserire il valore costante 31

2) I segnali di controllo che la CU deve generare per l'istruzione jaddl sono:

**RegWrite=1** **RegDst=X** **MemRead=X** **MemWrite=0** **AluSrc=0** **AluOp=add**

**MemToReg=X** **J=X** **Branch=X** **jaddl=1**

3) L'istruzione esegue contemporaneamente i due flussi (dati ed aggiornamento del PC):

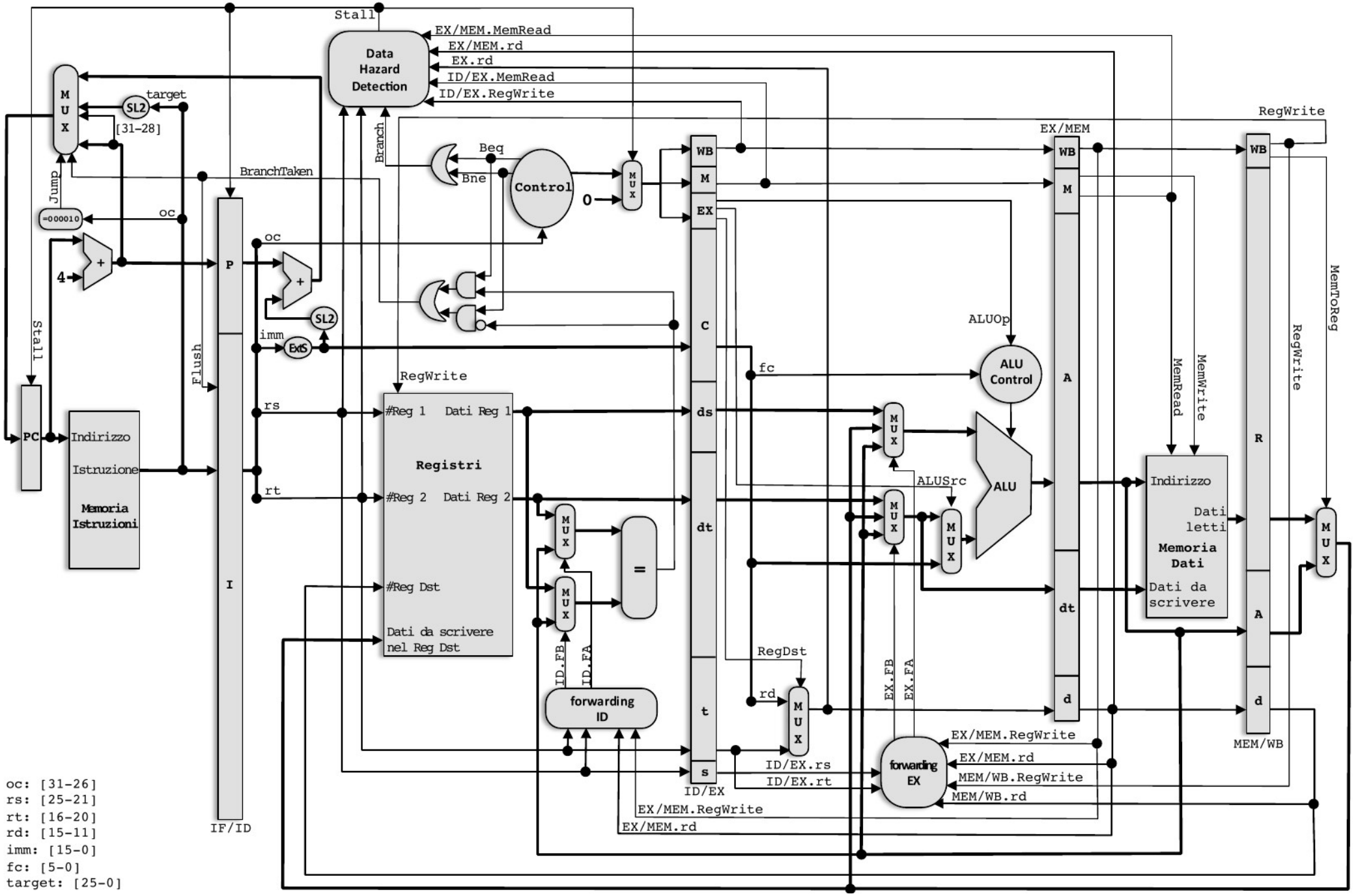
<table>
<tr><td>Fetch (75ns)</td><td>Decode (25ns)</td><td colspan="2">ADD (150ns)</td><td>Agg. PC (0ns)</td></tr>
<tr><td colspan="2">PC+4 (150 ns)</td><td>WB (25ns)</td><td></td><td></td></tr>
</table>

Quindi il tempo necessario è 250 ns, che è minore del tempo per:

- una lw (350ns=75+25+150+75+25)

- o una beq (300ns=150+150 per il calcolo della destinazione del salto condizionato)

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 MZ – Prof. Sterbini – 30/6/15

## *Parte 2 (per tutti – 2 ore)*

**Esercizio 3 (16 punti).** Considerate l'architettura MIPS con pipeline mostrata in figura (sul retro) ed il frammento di programma qui sotto che rovescia un testo usando puntatori.

**NOTA:** sono presenti solo le unità di forwarding presenti nella figura (sul retro).
**NOTA:** assumete che tutte le istruzioni usate nel programma siano di base (nessuna pseudoistruzione).

```
.data
TESTO:  .asciiz   "vediamo se oggi me la cavo"
N:      .word     26          # numero di caratteri

.text                         # Pipeline        fine istr.
main:   lw   $a1, N           # FDEMW               5
                                1 stallo (fw M->E)
        subi $a1, $a1, 1      #   FDEMW             7
        la   $a0, TESTO($0)   #    FDEMW            8
                                0 stalli (fw E->E)
        add  $a1, $a1, $a0    #     FDEMW           9
                                0 stalli (fw E->E)
next:   lb   $t2, ($a1)       #       FDEMW         10 20 30
        lb   $t1, ($a0)       #        FDEMW        11 21 ...
                                1 stallo (fw M->E)
        sb   $t1, ($a1)       #          FDEMW      13 23
        sb   $t2, ($a0)       #           FDEMW     14 24
        addi $a0, $a0, 1      #            FDEMW    15 25
        subi $a1, $a1, 1      #             FDEMW   16 26
                                1 stallo (fw E->D)
        blt  $a0, $a1, next   #               FDEMW 18 28 .. 138
                                1 stallo sul salto
fine:   li   $v0, 10          #                           139
        syscall                                           140
```

Indicate:
**1)** tra quali istruzioni sono presenti data hazard,

**Soluzione:** sono evidenziati con i colori

**2)** tra quali istruzioni sono presenti control hazard,

**Soluzione:** solo il salto del **blt**

**3)** tra quali istruzioni sono necessari stalli (data e control) (con forwarding)

**Soluzione:**
1 stallo prima di **subi $a1, $a1, 1**
1 stallo prima di **sb $t1, ($a1)**
1 stallo dopo **blt $a0,$a1,next** se salta

**NOTA: la $a0, TESTO($0)** non richiede stallo perché calcola solo l'indirizzo e NON fa accesso alla memoria
**NOTA: sb $t1, ($a0)** richiede 1 stallo perché posso fare forwarding dalla fase MEM di **lb** alla fase EXE di **sb**.

**4)** indicate il contenuto della pipeline (quali istruzioni si trovano in quali fasi) nel **16°** colpo di clock (con forwarding)

**Soluzione:**
WB=subi $a1, $a1, 1 MEM=stallo EXE=blt $a0, $a1, next ID=stallo IF=lb $t2, ($a1)

**5)** quanti cicli di clock sono necessari a eseguire tutto il programma (con forwarding)

**Soluzione:**
Il ciclo viene eseguito 13 volte (si parte dalle posizioni 0 e 25 e si arriva a 13 12) quindi
**N° colpi di clock** = 4 (riempimento pipeline) + 5 (istr. fino a lb compresa) + 1 (stallo) +
+ 13 x [ 7 (istruzioni) + 1 (stallo) + 1 (stallo) + 1 (stallo blt) ] - 2 (ultimo ciclo senza stallo e lb) + 2 (istruzioni finali) = 10 + 13 x 10 = 10 + 130 = **140**

**6)** quanti ne sarebbero necessari se esistesse una unità di forwarding anche nella fase MEM

**Soluzione:** in questo caso lo stallo tra lb e sb non sarebbe necessario, quindi si impiegano **13 cicli di meno**

**7)** quanti ne sarebbero necessari se il forwarding non esistesse per niente

**Soluzione:** TUTTI i data hazard necessitano di 2 stalli, i control hazard restano uguali
**N° colpi di clock** = 4 + 5 + 2 + 2 + 2 + 13 x [ 7 + 2 + 2 + 1 ] - 2 + 2 = 15 + 13 x 12 = 15 + 156 = **171**

**8)** riordinate le istruzioni per ridurre al massimo gli stalli (mantenendo invariata la sua semantica)

**Soluzione:**
Con il solo spostamento delle istruzioni è possibile eliminare sia lo stallo iniziale che quello tra lb e sb.

Se si ammette anche la modifica delle costanti immediate delle istruzioni si riesce ad eliminare anche lo stallo prima del blt spostando la coppia **addi subi** subito prima delle due **lb** a patto di modificare le istruzioni di accesso alla memoria cambiando la loro parte immediata in -1 o +1 in modo da leggere il byte dall'indirizzo corretto, ovvero:

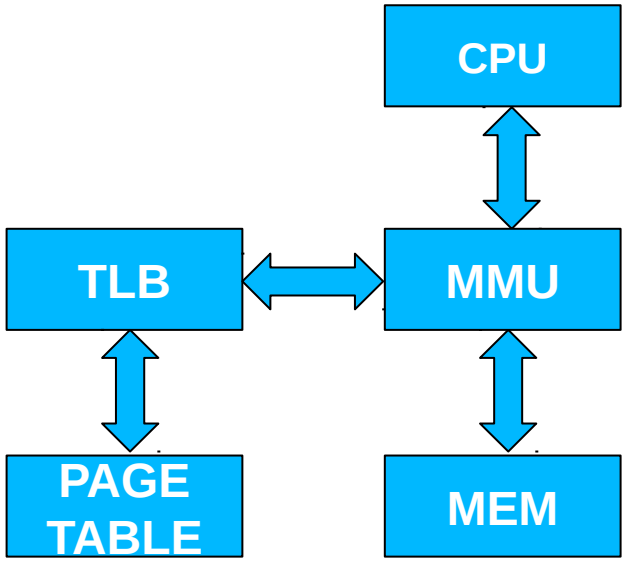
```
.data
TESTO:  .asciiz    “vediamo se oggi me la cavo”
N:      .word      26          # numero di caratteri

.text
main:   lw    $a1, N           # FDEMW                5
                                 0 stalli (fw M->E)
        la    $a0, TESTO($0)   #  FDEMW               6
        subi  $a1, $a1, 1      #   FDEMW              7
                                 0 stalli (fw M->E ed E->E)
        add   $a1, $a1, $a0    #     FDEMW            8
                                 0 stalli (fw E->E)
next:   addi  $a0, $a0, 1      #      FDEMW           9  17 25
        subi  $a1, $a1, 1      #       FDEMW          10 18 ...
                                 0 stalli (fw M->E)
        lb    $t2, +1($a1)     #        FDEMW         11 19
                                 0 stalli (fw M->E)
        lb    $t1, -1($a0)     #         FDEMW        12 20
                                 0 stalli (fw M->E)
        sb    $t2, -1($a0)     #          FDEMW       13 21
                                 0 stalli (fw M->E)
        sb    $t1, +1($a1)     #           FDEMW      14 22
        blt   $a0, $a1, next   #             FDEMW    15 23 ... 111
                                 1 stallo sul salto
fine:   li    $v0, 10          #                               112
        syscall                                                113
```

**9)** calcolate quanti cicli di clock sono necessari a eseguire il programma così ottimizzato (con forwarding ma non nella fase MEM)

**Soluzione:**
Quindi i cicli totali diventano:
- senza eliminare lo stallo prima del blt: 4 + 5 + 13 x [ 7 + 1 + 1 ] - 2 + 2 = 9 + 13 x 9 = 9 + 117 = **126**
- eliminando anche quello: 4 + 5 + 13 x [ 7 + 1 ] - 2 + 2 = 9 + 13 x 8 = 9 + 104 = **113**

# Esame di Architetture – Canale MZ – Prof. Sterbini – 30/6/15

**Esercizio 4 (14 punti). VM con TLB**




Sia data la gerarchia di memoria descritta in figura:
- il **TLB** è
  - **set-associativo** a **2 vie**
  - **2 set** per ogni via
  - con **blocchi da 1 word**
  - politica di sostituzione **LRU**
- la **RAM**
  - contiene 8 pagine fisiche, ciascuna da 512 byte
  - la politica di rimpiazzo è **LRU**

1) Per la seguente sequenza di accessi si determinino quali sono gli HIT/MISS su TLB
2) per ciascuna MISS indicate se è di tipo **Caricamento (L), Capacità (Cap) o Conflitto (Conf)**
3) indicate quali accessi generano Page Fault
4) dopo aver scelto una assegnazione #pagina virtuale → #pagina fisica calcolate gli ndirizzi fisici corrispondenti
5) calcolate le dimensioni in bit del TLB compresi i bit di controllo
6) calcolate il tempo medio di accesso (su questa sequenza) se:
- tempo di HIT su TLB = **1,5 ns**
- tempo di accesso a RAM o a Page Table = **75 ns**
7) calcolate il numero di istruzioni corrispondente al tempo medio di accesso, se la CPU ha una frequenza di clock di **4 GHz** e completa una istruzione ogni **3** colpi di clock. (ignorate il tempo necessario per un Page Fault)

**Soluzione:**
Il numero di pagina si ottiene dividendo l'indirizzo per 512 (parte intera)
L'offset si ottiene calcolando il resto dell'indirizzo diviso 512
Assumiamo che ogni linea dl TLB contenga una sola riga della tabella delle pagine, quindi il numero di pagina sarà anche il numero di blocco per il TLB.
Il tag e l'indice nel TLB si ottengono dividendo il numero di pagina per 2 ed ottenendo parte intera e resto.
Tutti i primi accessi a una pagina nel TLB sono sempre Miss di caricamento (L) e generano sempre un Page Fault.
I rimanenti accessi danno HIT se avvengono entro 2 accessi diversi (abbiamo 2 vie) allo stesso set (index) e in questo caso NON generano Page Fault perché se HIT allora la Page Table contiene il mapping e la pagina è già in memoria.
Altrimenti sono Miss di capacità se avvengono più di 4 accessi diversi prima (2 vie x 2 insiemi = 4 linee), oppure di conflitto se avvengono 3 o 4 accessi diversi prima.
Per capire se i Miss di conflitto o capacità generano Page Fault bisogna capire se la pagina richiesta è ancora in memoria fisica o è stata rimpiazzata.
La ram fisica contiene 8 pagine e gli accessi nel complesso richiedono solo 8 pagine quindi non c'è nessuna sostituzione di pagina e tutte le miss di conflitto o di capacità sul TLB NON generano Page Fault perché la corrispondente pagina è ancora in memoria fisica.
Per calcolare l'indirizzo fisico corrispondente a ciascun indirizzo virtuale bisogna concatenare l'indirizzo fisico della pagina (che potete scegliere come volete tra 0 e 7 visto che le pagine fisiche sono 8) con l'offset.
Questo corrisponde a calcolare:

512 * #pag. fisica + offset

In tabella vedete il risultato se le pagine virtuali sono assegnate alle pagine fisiche come indicato nell'ultima tabellina.

| **Indirizzo** | **1058** | **2100** | **600** | **3000** | **2200** | **3300** | **1100** | **4000** | **2900** | **3700** | **1800** | **6000** | **700** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **#pagina** | 2 | 4 | 1 | 5 | 4 | 6 | 2 | 7 | 5 | 7 | 3 | 11 | 1 |
| **Page Offset** | 34 | 52 | 88 | 440 | 152 | 228 | 76 | 416 | 340 | 116 | 264 | 368 | 188 |
| **TLB Tag** | 1 | 2 | 0 | 2 | 2 | 3 | 1 | 3 | 2 | 3 | 1 | 5 | 0 |
| **TLB Index** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| **TLB H/M** | M | M | M | M | H | M | M | M | H | H | M | M | M |
| **Tipo MISS** | L | L | L | L | | L | conf | L | | | L | L | cap |
| **Page Fault?** | SI | SI | SI | SI | NO | SI | NO | SI | NO | NO | SI | SI | NO |
| **Ind. Fisico** | 34 | 564 | 1112 | 1976 | 664 | 2276 | 76 | 2976 | 1876 | 2676 | 3336 | 3952 | 1212 |

**Mapping da pagina virtuale a pagina fisica da voi scelto**

| #p. virt. | 2 | 4 | 1 | 5 | 6 | 7 | 3 | 11 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #p. fisica | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | |

Tempo medio di accesso:

**Soluzione:**
Si ottiene sommando per ogni accesso:

1 accesso al TLB se HIT altrimenti 1 accesso alla Page Table (che sta in memoria)
più l'accesso al dato che sta in memoria (non c'è cache sul dato)

**Tempo totale senza tempo di Page Fault** = 3 HIT su TLB x 1.5 ns + 10 MISS TLB x 75ns + 13 accessi alla MEM fisica x 75ns = 4.5ns + 750ns + 975ns = **1729.5 ns**
**Tempo medio** = Tempo totale / 13 = 1729.5/13 = **133 ns** circa (poco meno di due accessi, che sono 150ns)

Numero di istruzioni svolte nel tempo medio di accesso:

**Soluzione:**
La CPU ha una frequenza di clock di 4 Ghz, quindi il **periodo di clock è di 1/4 di ns**.
Per completare una istruzione impiega 3 x 1/4 = 3/4 ns = **0.75 ns**
In 133 ns esegue 133/0.75 = **177 istruzioni** circa

# Esame di Architetture – Canale MZ – Prof. Sterbini – 30/6/15

## *Parte 3 (assembler)*

**Esercizio 5 (30 punti se corretto e ricorsivo, 18 se corretto e iterativo, 0 se non funziona).**

**Vanno svolti sia la funzione 1) che il main 2)**

**1)** Si realizzi la funzione RICORSIVA **lunghezzaLista** che riceve come argomenti:

- **lista:** indirizzo di un vettore contenente una lista (rappresentata come descritto dopo)
- **N**: il numero di elementi del vettore (compreso tra 1 ed 100 inclusi)
- **P:** la posizione del nodo corrente

e che:

- scandisce ricorsivamente la lista per calcolarne e **tornare come risultato la lunghezza** (che sarà minore o uguale ad N)
- ed inoltre mentre visita la lista **stampa gli indici delle posizioni visitate**.

La lista è rappresentata dagli elementi del vettore, ciascuno dei quali contiene:

- il valore **-1** (nodo senza successore)
- oppure l'indice del prossimo elemento della lista nel vettore (un numero tra 0 e N-1 compresi).

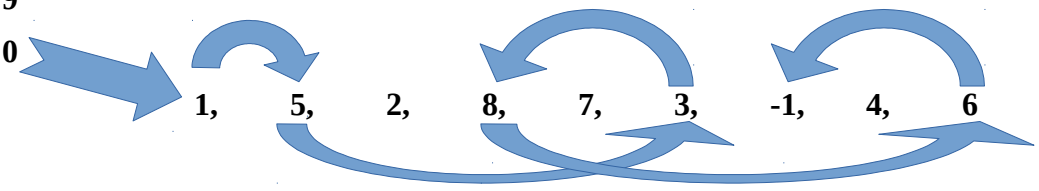
Il primo elemento della lista è sempre quello che si trova ad indice 0.

**Esempio**

**Argomenti:** **N:** **9**

**P:** **0**

**lista:** **1, 5, 2, 8, 7, 3, -1, 4, 6**



**Valori da stampare in lunghezzaLista:** **0 1 5 3 8 6**

**Risultato da tornare e stampare in main:** **6**

**NOTA: assumete che la lista da visitare non sia mai circolare (non ripassa su un nodo già visitato)**

**NOTA: i nodi non visitati (che non appartengono alla lista) possono contenere sia -1 che valori tra 0 e N-1**

**Esempio di algoritmo ricorsivo da realizzare:**

- stampare la posizione corrente **P** e leggere il nodo corrispondente **X=lista[P]**
- **se X == -1** questo nodo non ha successori, la lista è finita ed ha un solo nodo (caso base)
- **altrimenti** questa lista ha un nodo in più della lista che inizia dal prossimo nodo **X** (ricorsione)

**2)** Si realizzi un programma **main** che usa la funzione **lunghezzaLista** e che:

- legge da stdin (ovvero da tastiera):
  - il valore **0<N<=100** di elementi da inserire in un vettore **lista**
  - la successione di **N** valori interi e la memorizza nel vettore **lista**
- chiama la funzione **lunghezzaLista** passando gli argomenti **N,** indirizzo di **lista**, e **P=0**
- stampa il risultato tornato dalla funzione (la lunghezza)

**Soluzione**

```
.globl main
.data
DATI:     .word     0:100     # vettore da 100 elementi
.text
main:
     li   $v0, 5              # read integer
     syscall
     move $t1, $v0            # conteggio elementi da leggere
     li   $t0, 0              # offset del primo elemento
ciclo:    beqz $t1, calcola   # termino quando il conteggio è a 0
     li   $v0, 5              # read integer
     syscall
     sw   $v0, DATI($t0)      # memorizzo l'elemento
     addi $t0, $t0, 4         # passo al prossimo elemento
     subi $t1, $t1, 1         # decremento il conteggio
     j    ciclo

calcola:
     la   $a1, DATI           # indirizzo del vettore
     move $a0, $zero          # si parte dal primo elemento
     jal  lunghezzaLista
     move $t0, $v0            # metto da parte il risultato
     li   $a0, '\n'           # stampo accapo
     li   $v0, 11             # print char
     syscall
     move $a0, $t0            # recupero il valore
     li   $v0, 1              # print integer
     syscall
     li   $v0, 10             # end program
     syscall
```

```
# $a0 = indice dell'elemento corrente
# $a1 = indirizzo del vettore
# NOTA: viste le assunzioni non è necessario passare la dimensione di DATI
lunghezzaLista:
     li   $v0, 1              # print integer (ho già P in $a0)
     syscall
     sll  $t0, $a0, 2         # P*4 = offset dell'elemento corrente
     add  $t0, $a1, $t0       # posizione in memoria dell'elemento
     lw   $t0, ($t0)          # contenuto dell'elemento   X = DATI[P]
     li   $a0, ' '            # stampo spazio
     li   $v0, 11             # print char
     syscall
     # se != 1 calcolo la lunghezza dal prossimo elemento (ricorsione)
     bne  $t0, -1, caso_ricorsivo
     li   $v0, 1              # altrimenti sono nel caso base e torno 1
     jr   $ra

caso_ricorsivo:
     subi $sp, $sp, 4         # alloco una word su stack (PRIMA di usarla)
     sw   $ra, 0($sp)         # e preservo $ra
     move $a0, $t0            # P = X
     jal  lunghezzaLista
     # la lunghezza è 1 di più della lunghezza dal prossimo elemento
     addi $v0, $v0, 1
     lw   $ra, 0($sp)         # recupero $ra
     addi $sp, $sp, 4         # disalloco la stack (DOPO averla usata)
     jr   $ra                 # e torno al chiamante
```

**NOTA:** se non si usa l'ordine corretto di allocazione e salvataggio (o di ripristino e disallocazione) una semplice eccezione (interrupt) in quel momento potrebbe distruggere il contenuto del frame di attivazione e incasinare tutto.