Esame di Architetture – Canale MZ – Prof. Sterbini – 2/9/15 – Parte 1

Esercizio 1 (12 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 **MemWrite** attivo se e solo se è attivo **RegDst**.
Si assume che **RegDst** sia asserito solo per le istruzioni di tipo R, che **MemRead** e **MemToReg** siano asseriti solo per l'istruzione lw e che il segnale **Branch** sia asserito solo per l'istruzione beq.

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

I normali valori di **RegWrite** e di **RegDst** sono i seguenti, quindi le istruzioni che funzionano male sono **R** e **sw**

Istruzione	MemWrite	RegDst	Funziona o no?
tipo R	0 -> 1	1	NO (memorizza)
lw	0	0	SI
sw	1 -> 0	0	NO (non memorizza)
beq	0	0	SI
j	0	0	SI

In particolare le istruzioni R memorizzano nell'indirizzo che corrisponde al risultato della operazione (calcolato dalla ALU) il valore che si trova nel registro indicato dal campo **rt** della istruzione, ovvero il secondo argomento della istruzione.

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

Soluzione

Possiamo sfruttare o le istruzioni R che memorizzano un valore oppure la lw che non memorizza. Ad esempio

```
# usando sw che non memorizza
.data
N: .word 1
.text
sw $zero, N     # cerco di azzerare N
lw $s0, N       # se rotta torna 1
```

```
# usando add che scrive in memoria
.data
A:                  .word 0
indirizzo_di_A:     .word A
.text
lw $s0, A                   # si assume che la CPU sia OK
lw $t0, A                   # leggo il valore vecchio
lw $t1, indirizzo_di_A      # la $t1, A  ,di tipo R, può non funzionare
add $t1, $zero, $t1         # se CPU rotta scrive $t1 in A
lw $t2, A                   # lo rileggo per confrontarlo
beq $t0, $t1, fine          # se non è cambiato basta
li $s0, 1                   # altrimenti metto 1 in $s0 (CPU rotta)
fine:
```

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

Esercizio 2 (18 punti). Considerate l'architettura MIPS a ciclo singolo in figura (diagramma precedente). Vogliamo aggiungere l'istruzione di tipo J `jsr etichetta` (jump to subroutine) che:

- alloca una word su stack (ovvero sottrae 4 a $sp)
- salva l'indirizzo della istruzione successiva (PC+4) su stack all'indirizzo indicato da $sp.
- esegue un salto incondizionato all'indirizzo assoluto indicato dalla etichetta **(come per J)**.

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

NOTA: il registro $sp è il registro numero 29

2) indicate sul diagramma i valori di tutti i segnali di controllo 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 **100ns**, 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

Soluzione

1) per prima cosa dobbiamo vedere se sono necessarie unità funzionali aggiuntive (oppure se vanno modificate quelle esistenti). In questo caso potremmo usare:

- l'ALU per sottrarre 4 a $sp (già presente)
- il sommatore a valle del PC per calcolare PC+4 (già presente)
- la memoria in cui memorizzare PC+4 (già presente)

2) quindi dobbiamo individuare i trasferimenti dei dati da aggiungere al datapath:

- $sp ← $sp+4 (dai registri alla ALU ai registri, ingresso dati)

  bisogna indicare che il registro da leggere dal blocco dei registri è $sp, ovvero **bisogna mandare 29 alla porta #Reg1** del blocco dei registri

  bisogna fornire -4 come secondo argomento della ALU

  alla ALU bisogna indicare di svolgere una somma, quindi **AluOP=somma**

  il percorso dalla ALU alla porta dati in dei registri è già presente, **basta settare MemToReg=0**

  bisogna indicare che il registro destinazione deve essere $sp (registro 29), ovvero **bisogna mandare 29 sulla porta #RegDst** dei registri

- MEM[$sp] ← PC+4 (dal sommatore +4 all'ingresso dati della memoria)

  nota: il risultato della ALU è anche l'indirizzo in cui dobbiamo memorizzare PC+4

  questo percorso è già presente

- PC ← Destinazione del salto preso dalla istruzione di formato J (è lo stesso datapath di J)

  questo percorso è già presente, **basta settare Jump=1**

3) per ciacuno dei NUOVI percorsi aggiunti bisogna aggiungere un MUX nei punti in cui bisogna scegliere tra il normale valore e quello che abbiamo aggiunto

- sulla porta #Reg1 dei registri per leggere il registro $sp dobbiamo inserire 29
- sulla porta #RegDst dei registri per scrivere in $sp dobbiamo indicare 29
- sul secondo argomento della ALU per ricevere il valore -4 da sommare a $sp
- sulla porta di ingresso dati della memoria per ricevere PC+4 da salvare su stack

4) i segnali di controllo necessari sono quindi i seguenti (compreso il nuovo segnale JSR che deve essere prodotto dalla Control Unit quando viene riconosciuta la nuova istruzione)

AluSrc	AluOp	Jump	Branch	MemToReg	MemRead	MemWrite	RegWrite	RegDst	JSR
X	add	1	X	0	X	1	1	X	1

3) supponendo che l'accesso alle memorie impieghi **50ns**, l'accesso ai registri **25ns**, 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 della nuova istruzione e se la durata totale del ciclo di clock necessario è aumentata rispetto alla CPU senza la nuova istruzione

Soluzione

Per calcolare il tempo di esecuzione della istruzione dobbiamo considerare che contemporaneamente:

- viene calcolato PC+4

questo avviene in 100ns

- l'istruzione viene caricata, decodificata, eseguita, PC+4 viene scritto in memoria e $sp-4 viene scritto in $sp

Fetch:	50ns	
Decodifica e lettura $sp dai registri:	25ns	
Esecuzione di $sp-4 nell'ALU:	100ns	
Memorizzazione di PC+4 in MEM:	50ns	(il PC+4 a questo punto è già presente, non c'è attesa)
Memorizzazione di $sp-4 in $sp:	25ns	(questo può essere sovrapposto alla scrittura in MEM)

In totale si impiegano 225ns se si esegue il Write Back in parallelo alla scrittura in memoria, altrimenti 250ns

Dato che l'istruzione più lenta è **lw**, che impiega 50+25+100+50+25=250ns, la CPU mantiene lo stesso periodo di clock anche con la nuova istruzione.

Esame di Architetture – Canale MZ – Prof. Sterbini – 2/9/15 – Parte 2

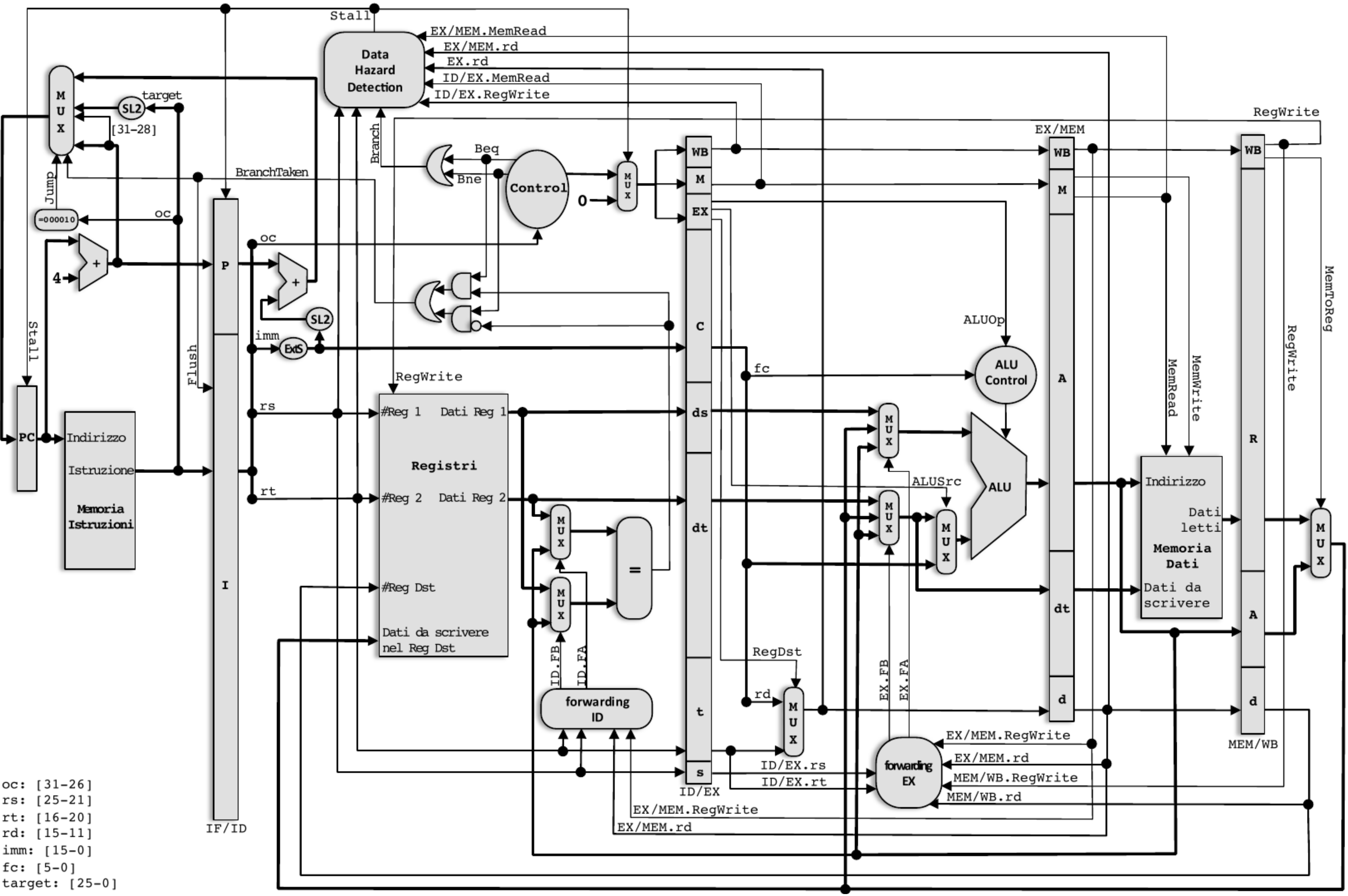
Esercizio 3 (16 punti). Considerate l'architettura MIPS con pipeline mostrata in figura (sul retro) ed il frammento di programma qui sotto che rovescia un testo usando indici.
NOTA: sono presenti solo le unità di forwarding presenti nella figura (sul retro).
NOTA: assumete che tutte le istruzioni usate nel programma siano di base (nessuna pseudoistruzione).
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) (con forwarding)
4) indicate il contenuto della pipeline (quali istruzioni si trovano in quali fasi) nel **16°** colpo di clock (con forwarding)
5) quanti cicli di clock sono necessari a eseguire tutto il programma (con forwarding)
6) quanti ne sarebbero necessari se esistesse una unità di forwarding anche nella fase MEM
7) quanti ne sarebbero necessari se il forwarding non esistesse per niente
8) riordinate le istruzioni per ridurre al massimo gli stalli (mantenendo invariata la sua semantica)
9) calcolate quanti cicli di clock sono necessari a eseguire il programma così ottimizzato (con forwarding ma non nella fase MEM)

Soluzione

```
.data
TESTO:  .asciiz   "vediamo se oggi me la cavo"
N:      .word     26           # numero di caratteri
.text                          #          esegue WB al
                               #          colpo di clock
main:   move $a0, $zero        # FDEMW          5
        lw   $a1, N            #  FDEMW         6
        subi $a1, $a1, 1       #   >FDEMW       8         1 stallo con FW, 2 senza FW
next:   lb   $t2, TESTO($a1)   #     FDEMW      9   19    0 stalli con FW, 2 senza FW
        lb   $t1, TESTO($a0)   #      FDEMW     10  20
        sb   $t1, TESTO($a1)   #       >FDEMW   12  22    1 stallo con FW -> E, 2 senza FW
        sb   $t2, TESTO($a0)   #         FDEMW  13  23
        addi $a0, $a0, 1       #          FDEMW 14  24    0 stalli con FW, 0 senza FW
        subi $a1, $a1, 1       #           FDEMW 15 25    0 stalli con FW, 2 senza FW
        blt  $a0, $a1, next    #            >FDEMW 17 27  1 stallo con FW, 2 senza FW
fine:   li   $v0, 10           #
        syscall
```

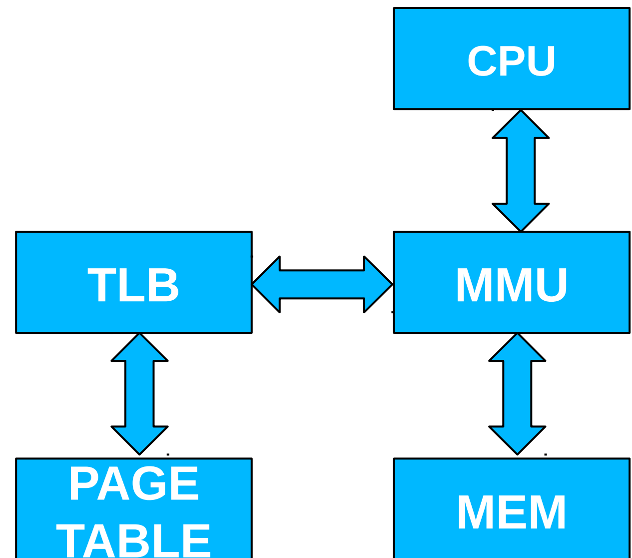
1) I data hazard sono indicati dai colori sui nomi dei registri coinvolti.
2) Un control hazard (in rosso) è presente se il salto condizionato viene effettuato, in tal caso va perso un colpo di clock perché l'istruzione successiva che era stata già caricata viene eliminata
3) Gli stalli sono indicati sopra. Nota: Il data hazard tra **lb** e **sb** introduce 1 stallo perché non è presente una unità di forwarding nella fase MEM, quindi il forwarding va fatto dal registro MEM/WB dopo la lettura di **lb** verso la fase EXE della istruzione **sb** (come indicato dal colore).
4) Al 16° colpo di clock: **WB=stallo, MEM=blt, EXE=stallo per salto a next, ID=lb, IF=lb**
5) Visto che il ciclo viene svolto 13 volte (ad ogni ciclo si avanza il puntatore al primo carattere e si indietreggia quello all'ultimo) i tempi di esecuzione sono:
FW: 4 (load pipeline) + 3 istruzioni + 1 stallo + 13 * (7 istruzioni + 2 stalli + 1 per salto) + 2 =10+130= **140**
6) Con FW in MEM si elimina uno stallo nel ciclo: 4 + 3 + 1 + 13 * (7 + 1 + 1) + 2 = 10 + 117 = **127**
7) SENZA FW: 4 + 3 + 4 + 13 (7 + 6 + 1) + 2 = 11 + 182 = **193**
8) Il codice può essere ottimizato eliminando gli stalli:
- tra **lw** e **subi** spostando la **lw** prima della **move**
- tra **lb** e **sb** scambiando le due **sb** tra di loro (oppure come indicato sotto)
- prima del **blt** spostando almeno una **sb** prima della **blt** ma solo se la scriviamo **sb $t1, TESTO+1($a1)**
9) Quindi il programma ottimizzato esegue: 4 + 3 + 0 + 13 * (7 + 0 + 1) + 2 = 9+104 = **113** colpi di clock

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



Esame di Architetture – Canale MZ – Prof. Sterbini – 2/9/15

Esercizio 4 (14 punti). VM con TLB



Sia data la gerarchia di memoria descritta in figura:
- il **TLB** è
  - **set-associativo** a **2 vie**
  - **4 set** per ogni via
  - con **blocchi da 1 word (1 linea della page table)**
  - politica di sostituzione **LRU**
- la **RAM**
  - contiene **8** pagine fisiche, ciascuna da **1024** 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 = **2,5 ns**
- tempo di accesso a RAM o a Page Table = **25 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 **2** colpi di clock. (ignorate il tempo necessario per un Page Fault)

Indirizzo (A)	**1058**	**2100**	**1600**	**6400**	**11200**	**13300**	**11000**	**15000**	**12900**	**15700**	**11800**	**16000**	**6700**
#pag. Virt. (A/1024)	1	2	1	6	10	12	10	14	12	15	11	15	6
Page Offset (A%1024)	34	52	576	256	960	1012	760	664	612	340	536	640	556
TLB Tag (#p/4)	0	0	0	1	2	3	2	3	3	3	2	3	1
TLB Index (#p%4)	1	2	1	2	2	0	2	2	0	3	3	3	2
TLB H/M	M	M	H	M	M	M	H	M	H	M	M	H	M
Tipo MISS	L	L		L	L	L		L		L	L		Conf.
Page Fault?	SI	SI	NO	SI	SI	SI	NO	SI	NO	SI	SI	NO	NO
Ind. Fisico (#p.fis.*1024 + off)	34	1076	576	2304	4032	5108	3832	5784	4708	6484	7704	6784	2604

Mapping da pagina virtuale a pagina fisica da voi scelto

#p. virt.	1	2	6	10	12	14	15	11					
#p. fisica	0	1	2	3	4	5	6	7					

Tempo totale: 4 Hit TLB + 9 Miss TLB per leggere la page table + 13 accessi in memoria al dato

4 * 2.5ns + 9 * 25ns + 13 * 25ns = 10ns + 225ns + 325ns = **560ns**

Tempo medio di accesso: 560ns / 13 ~ **44ns**

Numero di istruzioni svolte nel tempo medio di accesso: 44ns * 4 Ghz = 176 clock = **88 istruzioni**

Esame di Architetture – ASM – Canale MZ – Prof. Sterbini – 2/9/15

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 **profonditaAlbero** che riceve come argomenti:

- **albero:** indirizzo di un vettore contenente un albero (rappresentato come descritto dopo)

- **N**: il numero di elementi del vettore (numero pari compreso tra 2 ed 100 inclusi)

- **P:** la posizione del nodo corrente (un numero pari)

e che:

- scandisce ricorsivamente **in PREORDINE** l'albero per calcolarne e **tornare come risultato la sua profondità** ovvero il numero massimo di nodi dalla radice alle foglie compresi.

- ed inoltre mentre visita l'albero in preordine **stampa gli indici delle posizioni dei nodi visitati**.

L'albero è rappresentato da nodi, ciascun nodo è una coppia di elementi consecutivi del vettore, che indicano rispettivamente quali sono il figlio sinistro e il figlio destro del nodo corrente. Il valore contenuto in un elemento del vettore può essere:

- il valore **-1** (il figlio corrispondente non è presente)

- oppure l'indice del nodo figlio (un numero pari tra 0 e N-2 compresi).

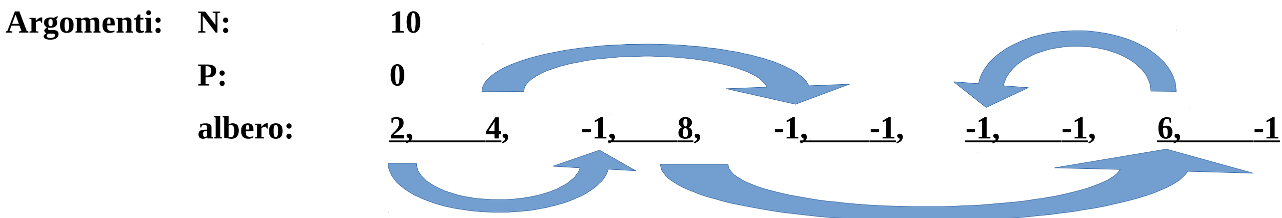
La radice dell'albero è sempre il nodo che si trova ad indice 0, (e che contiene nella posizione 0 l'indice del figlio sinistro e nella posizione 1 l'indice del figlio destro).

Esempio

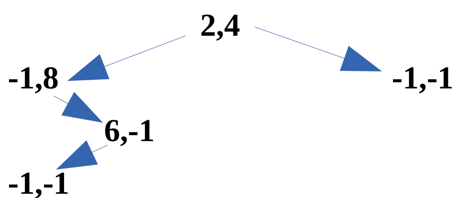
Argomenti: **N:** **10**

P: **0**

albero: **2, 4, -1, 8, -1, -1, -1, -1, 6, -1**



Ovvero l'albero che contiene i nodi:



e che tornano le profondità

4 = max(3, 1) + 1

3 = max(0, 2) + 1 **1 = max(0,0)+1**

2 = max(1, 0) +1

1 = max(0,0)+1

La visita in prordine leggerà (e stamperà) le posizioni dei nodi: **0, 2, 8, 6, 4**

Risultato da tornare e stampare in main: **4** **(dalla radice alla foglia più profonda comprese)**

NOTA: assumete che l'albero da visitare non sia mai circolare (non ripassa su un nodo già visitato)

(segue dietro)

Esempio di algoritmo <u>ricorsivo</u> da realizzare:

- stampare la posizione corrente **P** seguita da spazio

- leggere il figlio sinistro **SX=albero[P]**

  - se SX == -1 questo nodo non ha figli sinistri, la profondità a sinistra è 0 (caso base)

  - altrimenti esiste il sottoalbero sinistro e parte dal nodo in posizione **SX** (ricorsione)

- leggere il figlio destro **DX=albero[P+1]**

  - se DX == -1 questo nodo non ha figli destri, la profondità a destra è 0 (caso base)

  - altrimenti esiste il sottoalbero destro e parte dal nodo in posizione **DX** (ricorsione)

- tornare come valore della profondità il massimo tra la profondità sinistra e quella destra, più 1

Esempio di algoritmo <u>iterativo</u> SENZA STAMPA IN PREORDINE: (voto 18)

- allocare staticamente un vettore di **appoggio** in cui tenere le profondità calcolate per ciascun nodo, inizializzate a zero

- per **N/2** volte (profondità massima dell'albero)

  - per tutti i nodi del vettore

    - calcola la profondità del nodo leggendo le profondità dei due sottoalberi da appoggio

    - memorizza in appoggio la profondità del nodo

Alla fine nella posizione 0 (radice) del vettore di appoggio avrete la profondità dell'albero

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

- legge da stdin (ovvero da tastiera):

  - il valore pari **2<=N<=100** di elementi da inserire in un vettore **albero**

  - la successione di **N** valori interi e la memorizza nel vettore **albero**

- chiama la funzione **profonditaAlbero** passando gli argomenti **N,** indirizzo di **albero**, e **P=0 (la radice)**

- stampa il risultato tornato dalla funzione (la profondità calcolata)

Soluzione iterativa (18 punti)

```
.globl main

.data
ALBERO:         .word   0:100       # alloco 100 word inizializzate con zero
PROFONDITA:     .word   0:100       # alloco un vettore di appoggio per le profondità calcolate
(inizialmente 0)

.text
main:
        li      $v0, 5          # read integer
        syscall
        move    $s0, $v0        # $s0 = numero di elementi N
        move    $a2, $v0        # $a2 = numero di elementi N
        move    $s1, $zero      # indice del primo valore

loop:   li      $v0, 5          # read integer
        syscall
        sw      $v0, ALBERO($s1)
        addi    $s1, $s1, 4
        subi    $s0, $s0, 1
        bgtz    $s0, loop
# fine della lettura

        li      $a0, 0          # si parte dal nodo con indice 0
        jal     profondita_iter
        move    $s0, $v0
        li      $a0, '\n'       # stampo accapo
        li      $v0, 11         # print character
        syscall
        li      $v0, 1          # print integer
        move    $a0, $s0        # stampo la profondità
        syscall
        li      $v0, 10         # fine programma
        syscall
```

```
# $a0 = indice del nodo corrente                    # $a2 = N
profondita_iter:
        div     $t0, $a2, 2                         # per N/2 volte
        aggiorna_profondita:
                subi    $t1, $a2, 2                 # per tutti i nodi del vettore a partire da N-2
                aggiorna_nodo:                      # leggo il figlio sinistro
                        sll     $t2, $t1, 2         # offset = I*4
                        lw      $t3, ALBERO($t2)    # SX = ALBERO[I]
                        bltz    $t3, mancaSX        # se -1 non c'è figlio
                        # altrimenti la profondità è nel vettore di appoggio
                        sll     $t3, $t3, 2         # offset del figlio SX
                        lw      $t4, PROFONDITA($t3)    # profondita di SX
                        j       destro
                        mancaSX:
                        li      $t4, 0              # oppure 0
                        destro:                     # leggo il figlio destro
                        addi    $t2, $t1, 1         # il figlio destro sta all'indice seguente
                        sll     $t2, $t2, 2         # offset del DX
                        lw      $t3, ALBERO($t2)    # DX = ALBERO[I+1]
                        bltz    $t3, mancaDX        # se -1 non c'è figlio
                        sll     $t3, $t3, 2         # offset DX
                        lw      $t5, PROFONDITA($t3)    # profondità di DX
                        j       max                 # calcola il massimo
                        mancaDX:
                        li      $t5, 0              # se manca DX la prof. è 0
                        max:                        # trovo il massimo
                        bgt     $t4, $t5, gia_maggiore  # voglio il maggiore in $t4
                        move    $t4, $t5
                        gia_maggiore:
                        addi    $t4, $t4, 1         # aggiungo 1 al massimo
                        # memorizzo il massimo del nodo corrente
                        sll     $t2, $t1, 2         # offset = I*4
                        sw      $t4, PROFONDITA($t2)    # aggiorno la profondita del padre
                        subi    $t1, $t1, 2         # passo al prossimo nodo
                bgez    $t1, aggiorna_nodo
                subi    $t0, $t0, 1                 # prossima iterazione
        bgez    $t0, aggiorna_profondita
        lw      $v0, PROFONDITA   # in APPOGGIO[0] c'è la profondità
        jr      $ra
```

Soluzione ricorsiva

```
.globl main
.data
ALBERO:         .word   0:100       # alloco 100 word inizializzate con zero
.text
main:
        li      $v0, 5          # read integer
        syscall
        move    $s0, $v0        # $s0 = numero di elementi N
        move    $s1, $zero      # indice del primo valore

loop:   li      $v0, 5          # read integer
        syscall
        sw      $v0, ALBERO($s1)
        addi    $s1, $s1, 4
        subi    $s0, $s0, 1
        bgtz    $s0, loop
# fine della lettura

        li      $a0, 0          # si parte dal nodo con indice 0
        la      $a1, ALBERO
        jal     profondita
        move    $s0, $v0
        li      $a0, '\n'       # stampo accapo
        li      $v0, 11         # print character
        syscall
        li      $v0, 1          # print integer
        move    $a0, $s0        # stampo la profondità
        syscall
        li      $v0, 10         # fine programma
        syscall
```

```
#a0 = indice del nodo corrente                  #a1 = indirizzo del vettore
profondita:
        bgez    $a0, non_foglia                 # se la posizione è > -1 non è una foglia (caso ricorsivo)
        li      $v0, 0                          # la profondità è zero (caso base)
        jr      $ra                     # non c'è bisogno di ripristinare la stack perchè non l'ho ancora modificata
non_foglia:                                     # caso ricorsivo
                                # salvo su stack i registri che vengono modificati dalla funzione ($ra, $a0, $t0)
        subi    $sp, $sp, 12                    # alloco 3 word
        sw      $ra, 0($sp)                     # salvo $ra
        sw      $a0, 4($sp)                     # salvo $a0
        sw      $t0, 8($sp)                     # salvo $t0
        li      $v0, 1                          # print integer
        syscall                                 # stampo la posizione dell'elemento nel vettore
        li      $v0, 11                         # print character
        li      $a0, ' '  # spazio
        syscall
        # i due figli sono in posizione $a0 e $a0+1 (caso ricorsivo)
        lw      $a0, 4($sp)                     # recupero l'indice del nodo
        sll     $s0, $a0, 2                     # P * 4
        add     $s0, $s0, $a1                   # posizione in memoria
        lw      $a0, ($s0)                      # leggo il figlio sinistro
        jal     profondita                      # ricorsione sul figlio sinistro
        move    $t0, $v0                        # tengo da parte il risultato
        lw      $a0, 4($sp)                     # recupero l'indice del nodo
        addi    $a0, $a0, 1                     # il figlio destro è all'indice seguente
        sll     $s0, $a0, 2                     # P * 4
        add     $s0, $s0, $a1                   # posizione in memoria
        lw      $a0, ($s0)                      # leggo il figlio destro
        jal     profondita                      # ricorsione sul figlio destro
        bgt     $v0, $t0, gia_maggiore          # confronto le profondità dei due sottoalberi
        move    $v0, $t0
gia_maggiore:
        addi    $v0, $v0, 1                     # aggiungo 1 alla profondità massima dei 2 sottoalberi
        # ripristino da stack i valori salvati
        lw      $ra, 0($sp)                     # recupero $ra
        lw      $a0, 4($sp)                     # recupero $a0
        lw      $t0, 8($sp)                     # recupero $t0
        addi    $sp, $sp, 12                    # disalloco 3 word
        jr      $ra
```