Politecnico di Milano
Dipartimento di Elettr., Inform. e Bioing.

prof.ssa Anna Antola **prof. Giuseppe Pelagatti**
prof. Luca Breveglieri **prof.ssa Donatella Sciuto**
prof. Roberto Negrini **prof.ssa Cristina Silvano**

AXO – Architettura dei Calcolatori e Sistemi Operativi

esame di martedì 9 settembre 2014

CON SOLUZIONI

Cognome _____ **Nome** _____

Matricola _____ **Firma** _____

Istruzioni

Scrivere solo sui fogli distribuiti. Non separare questi fogli.

È vietato portare all'esame libri, eserciziari, appunti, calcolatrici e telefoni cellulari. Chiunque fosse trovato in possesso di documentazione relativa al corso – anche se non strettamente attinente alle domande proposte – vedrà annullata la propria prova.

Non è possibile lasciare l'aula conservando il tema della prova in corso.

Tempo a disposizione: 1h:30m (una parte) 3h:00m (completo).

	punteggio approssimativo	I parte ☐	II parte ☐	completo ☐
esercizio 1	5			
esercizio 2	6			
esercizio 3	5			
esercizio 4	5			
esercizio 5	6			
esercizio 6	5			
voto finale				

ATTENZIONE: alcuni esercizi sono suddivisi in parti.

esercizio n. 1 – thread e parallelismo

Si consideri il programma C seguente (gli "#include" sono omessi):

```
pthread_mutex_t UP, DOWN          /* iniz. a PTHREAD_MUTEX_INITIALIZER */
sem_t QUARK;
int global = 0;

void * FIRST (void * arg) {
   int local = 0;
   pthread_mutex_lock (&UP);                                  /* istruzione A */
   global = 1;
   pthread_mutex_unlock (&UP);
   return NULL;
} /* end FIRST */

void * NEXT (void * arg) {
   int local = 0;
   pthread_mutex_lock (&DOWN);
   sem_wait (&QUARK);                                         /* istruzione B */
   global = 2;
   pthread_mutex_unlock (&DOWN);
   return 4;
} /* end NEXT */

void * LAST (void * arg) {
   int local = 0;
   pthread_mutex_lock (&DOWN);
   sem_wait (&QUARK);                                         /* istruzione C */
   global = 3;
   pthread_mutex_lock (&UP);
   sem_post (&QUARK);
   pthread_mutex_unlock (&UP);
   pthread_mutex_unlock (&DOWN);
   return NULL;
} /* end LAST */

void main ( ) {
   pthread_t TH_1, TH_2, TH_3;
   sem_init (&QUARK, 0, 1);
   pthread_create (&TH_1, NULL, FIRST, NULL);
   pthread_create (&TH_2, NULL, NEXT, &global);
   pthread_create (&TH_3, NULL, LAST, NULL);
   pthread_join (TH_1, NULL);
   pthread_join (TH_2, NULL);                                 /* istruzione D */
   pthread_join (TH_3, NULL);
   return;
} /* end main */
```

Si completi la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva ESISTE; se **non esiste**, si scriva NON ESISTE; e se può essere **esistente o inesistente**, si scriva PUÒ ESISTERE. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	**variabile locale**		
	local in **TH_1**	*local* in **TH_2**	*local* in **TH_3**
subito dopo istr. **A**	*ESISTE*	*PUÒ ESISTERE*	*PUÒ ESISTERE*
subito dopo istr. **B**	*PUÒ ESISTERE*	*ESISTE*	*PUÒ ESISTERE*
subito dopo istr. **C**	*PUÒ ESISTERE*	*ESISTE*	*ESISTE*
subito dopo istr. **D**	*NON ESISTE*	*NON ESISTE*	*PUÒ ESISTERE*

Si completi la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato. Per i MUTEX, si scriva **0** se il MUTEX è **libero**, si scriva **1** se il MUTEX è **occupato**.

condizione	**variabile globale**			
	QUARK	*UP*	*DOWN*	*global*
subito dopo istr. **A**	*0 / 1*	*1*	*0 / 1*	*0 / 2 / 3 / 4*
subito dopo istr. **B**	*0*	*0 / 1*	*1*	*0 / 1 / 3*
subito dopo istr. **C**	*0*	*0 / 1*	*1*	*0 / 1*
subito dopo istr. **D**	*0*	*0*	*0 / 1*	*1 / 4*

Il sistema va senz'altro in stallo (deadlock), in un solo caso. Qui **si indichino** le primitive dove si bloccano i thread (uno o più di uno), precisando il valore (o i valori) della variabile `global`:

TH_1	**TH_2**	**TH_3**	*global*
–	–	`sem_wait`	*1 / 2* / 4*

** Alla lunga il valore 2 viene sovrascritto dal valore 1 o dal valore 4, poiché i thread TH_1 e TH_2 non si bloccano e sono indipendenti.*

esercizio n. 2 – processi e sistema operativo

prima parte – commutazione tra processi

Sono dati due **processi P** e **S**. Il processo **P** esegue il programma **CODICE_UNO** e crea il processo figlio **Q**. Il processo **S** (che non è figlio né di **P** né di **Q**) esegue il programma **CODICE_DUE**. Nel sistema non ci sono altri processi utente oltre a **P**, **S** e **Q**.

```
/* programma CODICE_UNO.c eseguito dai processi P e Q                */
main ( ) {

(1) fd1 = open ("/acso/file1", O_RDWR);          /* 4 blocchi   */
(2) read (fd1, vet_P, 540);                      /* 2 blocchi   */
    pid = fork ( );
    if (pid == 0) {       /* codice eseguito da Q figlio di P   */
(3)    write (fd1, vet_Q, 1024);                 /* 2 blocchi   */
(4)    lseek (fd1, 30, 0);   /* 0 si riferisce a inizio file    */
(5)    fd2 = open ("/acso/file2", O_RDWR);       /* 2 blocchi   */
(6)    write (fd2, vet_Q, 1024);                 /* 2 blocchi   */
       ...
       exit (1);
    } else {              /* codice eseguito da P               */
(7)    read (fd1, vet_P, 10);                    /* 1 blocco    */
       pid = wait (&status);
       ...
       exit (0);
    } /* if */
} /* CODICE_UNO */
```

```
/* programma CODICE_DUE.c eseguito dal processo S                */
main ( ) {

(8)  fd = open ("/acso/file2", O_RDWR);          /* 4 blocchi   */
(9)  lseek (fd, 430, 1);    /* 1 si riferisce a posizione corr. */
(10) write (fd, vet_S, 1030);                    /* 2 blocchi   */
     ...
     exit (2);
} /* CODICE_DUE */
```

Ulteriori specifiche:

1. i processi utente hanno una priorità associata, il processo "idle" ha priorità minima e nel sistema non ci sono altri processi oltre a quelli utente e a "idle"
2. quando è necessario, sono indicate le priorità dei processi attivi nel sistema
3. quando un processo diventa pronto e ha priorità maggiore di quello in esecuzione, va attivato lo scheduler
4. il buffer del driver di standard output ha dimensione di **50 caratteri**
5. per le operazioni sui file che – nell'esercizio considerato – implicano trasferimento di blocchi:
    a. l'interruzione di fine DMA è associata al trasferimento di un singolo blocco di file: evento *DMA_in* per lettura di un blocco da file ed evento *DMA_out* per scrittura di un blocco su file
    b. il numero di blocchi da trasferire è indicato esplicitamente come commento nel codice
6. le chiamate di sistema "wait" e "waitpid" invocano la funzione "Sleep_on" su un evento opportuno
7. per completare la tabella delle commutazioni, si faccia riferimento alla notazione vista a lezione

Si completino le parti mancanti della tabella di commutazione dei processi. Nella tabella sono previste righe da completare, dove:

a. è specificato l'evento (con informazioni aggiuntive); qui sono da completare le parti relative allo stato dei processi e – se richiesto – ai moduli del S.O. con il contesto

b. è specificato lo stato dei processi raggiunto dopo il verificarsi dell'evento; qui sono da completare i campi relativi all'evento (con eventuali informazioni aggiuntive) e – se richiesto – ai moduli del S.O. con il contesto

NOTA BENE:

- l'evento è sempre determinabile univocamente dallo stato raggiunto, dall'evoluzione precedente dei processi, dal codice dei programmi e dalle ulteriori specifiche di sistema
- se l'evento è un interrupt, va usata la notazione "*n* interrupt", indicando esattamente il numero di interruzioni che si sono verificate

EVENTO (associato al nome del processo dove si verifica)	**informazioni aggiuntive**	**moduli del SO invocati per gestire l'evento**	**processo dov'è gestito il modulo**	**stato dei processi dopo la gestione dell'evento**		
				P	**Q**	**S**
	per scadenza del quanto di tempo			**pronto**	**non esiste**	**esecuzione**
S: lseek		*G_SVC_1* *lseek* *G_SVC_2/3*	*S* *S* *S*	*pronto*	*non esiste*	*esec*
S: write	*write ha inizializzato il DMA in scrittura*	*G_SVC_1* *write* *Sleep_on_1 (E1)* *Change* *Preempt_2* *R_int (CK)*	*S* *S* *S* *S – P* *P* *P*	esec	non esiste	attesa (E1)
P: pd = fork		*G_SVC_1* *fork* *G_SVC_2/3*	*P* *P* *P*	*esec*	*pronto*	*attesa (E1)*
P: 2 interrupt da DMA_out	*Q ha la priorità maggiore*	*R_int (DMA_in)* *Wake_up (E1)* *Preempt_1* *Change* *G_SVC_2/3*	*P* *P* *P* *P – Q* *Q*	pronto	esec	pronto
Q: open	**P < S**	*G_SVC_1* *open* *Sleep_on_1 (E2)* *Change* *Sleep_on_2 (E1)* *write* *G_SVC_2/3*	*Q* *Q* *Q* *Q – S* *S* *S* *S*	*pronto*	*attesa (E2)*	*esec*
S: exit		*G_SVC_1* *exit* *Sleep_on_2 (E1)* *write* *G_SVC_2/3*	*S* *S – P* *S* *S* *S*	esec	attesa (E2)	non esiste

TABELLA DI COMMUTAZIONE DEI PROCESSI *(fine)*

In rosso sono colorate le parti di tabella assegnate.

seconda parte – funzioni del file system

Si consideri nuovamente il codice dei programmi dati nella prima parte e in particolare le chiamate di sistema identificate dai numeri d'ordine da **1** a **10**.

Relativamente a questa seconda parte valgono le specifiche seguenti:

- l'area buffer in memoria centrale gestita dal SO per le operazioni su file è costituita da **4 buffer** di dimensione opportuna che possono essere assegnati indifferentemente a un qualsiasi file del sistema. In caso di mancanza di spazio l'area buffer viene gestita con politica **LRU**.
- per **tutte le operazioni** su file
  (a) la dimensione del blocco trasferito da o verso file tramite **DMA** è di **512 byte**
  (b) l'interruzione di fine DMA è associata al trasferimento di un singolo blocco
  (c) le chiamate di sistema sono dipendenti e pertanto i blocchi allocati/letti dalle precedenti chiamare di sistema rimangono a disposizione in memoria finché è possibile
  (d) gli *I-node*, una volta trasferiti da disco, vengono copiati in memoria centrale nella *I-lista* e il buffer occupato per il loro trasferimento viene subito considerato "libero"
- per l'**apertura** del file è **sempre** necessario accedere a:
  (a) un blocco per l'accesso allo *I-node* di ogni cartella (catalogo) o file presente nel nome-percorso (pathname)
  (b) un blocco per il contenuto di ogni cartella (catalogo) presente nel nome-percorso (pathname)
- i blocchi possono trovarsi in memoria (se già acceduti) o necessitare di trasferimento DMA da disco

Al momento dell'esecuzione il contenuto del volume è il seguente:

I-lista: ⟨ 0, dir, 4 ⟩ ⟨ 6, dir, 20 ⟩ ⟨ 32, norm, (600, 601, 602, 603, 604, 605, 606) ⟩
⟨ 48, norm, (800, 812, 700, 703, 705) ⟩

blocco 4: ... ⟨ 6, acso ⟩ ...

blocco 20: ... ⟨ 32, file1 ⟩ ⟨ 48, file2 ⟩ ...

Nota: **lo *I-node* di root è sempre in memoria** e il contenuto dei blocchi dei file normali è omesso poiché non è significativo ai fini dell'esercizio.

Si ipotizzi che la sequenza di ordine delle chiamate di sistema sia **1 2 8 9 3 10**.

Per ciascuna della chiamate, **si indichino** nella tabella predisposta:

- il numero totale di interruzioni di fine DMA che si verificano affinché l'operazione si completi
- e la sequenza di accessi:
  - agli *I-node* su disco (notazione "*I-node [X]*", con X = numero *I-node*)
  - agli *I-node* nella lista in memoria centrale (notazione "*I-lista [X]*", con X = numero *I-node*)
  - e ai blocchi (con la notazione "*blocco Y*"), specificando se a memoria (**M**) o a disco (**D**)

Nell'ultima colonna si indichi anche lo stato dei quattro buffer alla fine della gestione di ogni chiamata, indicando per ciascuno il numero di blocco presente e il file a cui appartiene, oppure se è libero (**L**).

chiamata di sistema	n. di interrupt di fine DMA	sequenza di accessi in memoria o su disco	stato dei buffer
(1) **open** ("/acso/file1", O_RDWR)	*0* *1* *1* *1* *1* *totale* *4 interrupt*	*I-lista [0] in M* *blocco 4 – D* *I-node [6] – D* *blocco 20 – D* *I-node [32] – D*	BUF_1 = *blocco 4* BUF_2 = *I_node acso, L* BUF_3 = *blocco 20* BUF_4 = *I_node file1, L*
(2) **read** (fd1, vet_P, 540)	*0* *1* *1* *totale* *2 interrupt*	*I-lista [32] in M* *blocco 600 – D* *blocco 601 – D*	BUF_1 = *blocco 4* BUF_2 = *600* BUF_3 = *blocco 20* BUF_4 = *601*
(8) **open** ("/acso/file2", O_RDWR	*0* *0* *0* *0* *1* *totale* *1 interrupt*	*I-lista [0] in M* *blocco 4 – M* *I-lista [6] – M* *blocco 20 – M* *I-node [48] – D*	BUF_1 = *blocco 4* BUF_2 = *I-node file2, L* BUF_3 = *blocco 20* BUF_4 = *601*
(9) **lseek** (fd, 430, 1)	*0* *totale* *0 interrupt*	*I-lista [48] – M*	BUF_1 = *blocco 4* BUF_2 = *I-node file2, L* BUF_3 = *blocco 20* BUF_4 = *601*

(continua)

chiamata di sistema	n. di interrupt di fine DMA	sequenza di accessi in memoria o su disco	stato dei buffer
(3) **write** (fd1, vet_Q, 1024)	*0* *0* *1* *1* *totale* *2 interrupt*	*I-lista [32] – M* *blocco 601 – M* *blocco 602 – D* *blocco 603 – D*	BUF_1 = *blocco 603* BUF_2 = *blocco 602* BUF_3 = *blocco 20* BUF_4 = *601*
(10) **write** (fd, vet_S, 1030)	*0* *1* *1* *1* *1* *1* *totale* *5 interrupt*	*I-lista [48] – M* *blocco 800 – D* *blocco 601 – D* *blocco 812 – D* *blocco 602 – D* *blocco 700 – D*	BUF_1 = *blocco 603* BUF_2 = *blocco 700* BUF_3 = *blocco 800* BUF_4 = *812*

(fine)

esercizio n. 3 – memoria virtuale

Un sistema dotato di memoria virtuale con paginazione e segmentazione tipo UNIX è caratterizzato dai parametri seguenti: la memoria fisica ha capacità di **32 K byte**, la memoria logica ha capacità di **64 K byte** e la dimensione delle pagine è di **4096 byte**.

(a) Nel sistema vengono attivati i processi P e Q. Essi eseguono i programmi X e Y, e condividono una pagina dati. La dimensione iniziale dei segmenti dei due programmi X e Y è la seguente:

CX: 8 K DX: 4 K PX: 4 K COND: 4 K

CY: 12 K DY: 8 K PY: 4 K COND: 4 K

Il segmento COND è allocato lasciando **due pagine** libere dopo il segmento dati di X e Y.

Si inserisca in tabella la struttura in pagine della memoria virtuale dei due programmi X e Y; si ricorra alla notazione: CX0, CX1, ..., DX0, ..., PX0, ..., CY0, ..., COND0, ...

indirizzo di pagina virtuale	X	Y
0	***CX0***	***CY0***
1	***CX1***	***CY1***
2	***DX0***	***CY2***
3	***(DX1)***	***DY0***
4	***(DX2)***	***DY1***
5	***COND***	
6		
7		***COND***
8		
9		
A		
B		
C		
D		
E		
F	***PX0***	***PY0***

(b) A un certo istante t_0 sono terminati, nell'ordine, gli eventi seguenti:

1. **P** viene creato ("fork" di P ed "exec" di X) *alloca CP0, DP0 e PP0*
2. **P** esegue una "fork" e crea Q *alloca CQ0 = CP0, DQ0 = DP0 e PQ0*
3. **P** esegue una "sbrk" di indirizzo massimo **40AA hex** (virtuale assoluto) *alloca DP1, e poi rilascia DP0 e alloca DP2*
4. **Q** salta all'istruzione di indirizzo **1440 hex** (virtuale assoluto) *alloca CQ1*
5. **Q** accede a COND *rilascia DQ0 e alloca COND*
6. **P** accede a COND *rilascia DP1 e alloca COND*

Considerando le ipotesi seguenti, **si compilino** le tabelle della situazione al tempo t_0 relative alla memoria fisica e al contenuto della MMU:

- il lancio in esecuzione di un programma avviene caricando solo la pagina di codice con l'istruzione di partenza, la **prima pagina dati** e una pagina di pila, in quest'ordine
- il caricamento di ulteriori pagine in memoria avviene "on demand"
- nella "sbrk" vengono caricate **tutte** le pagine **dati** fino all'indirizzo richiesto
- il numero **R** di pagine residenti è pari a **4**
- l'indirizzo di partenza del programma **X** è **01CA hex** (virtuale assoluto)
- l'indirizzo di partenza del programma **Y** è **20A0 hex** (virtuale assoluto)
- si utilizza l'algoritmo LRU per la sostituzione di pagine di memoria, considerando che almeno una pagina di pila debba sempre rimanere in memoria
- l'allocazione delle pagine virtuali nelle pagine fisiche avviene sempre in sequenza
- all'inizio della sequenza di eventi la MMU è vuota e se è richiesta una nuova riga si utilizza **sempre** la prima riga libera

situazione al tempo t_0

memoria fisica	
indirizzo fisico	**pagine allocate**
0	*CP0 = CQ0*
1	*(DP0) = (DQ0) COND*
2	*PP0*
3	*PQ0*
4	*(DP1)*
5	*DP2*
6	*CQ1*
7	

MMU			
proc.	**NPV**	**NPF**	**valid bit**
P	*CP0 / 0*	*0*	*1*
P	*(DP0 / 2) DP2 / 4*	*(1) 5*	*1 0 1*
P	*PP0 / F*	*2*	*1*
Q	*CQ0 / 0*	*0*	*1*
Q	*(DQ0 / 2) COND / 5*	*(1) 1*	*1 0 1*
Q	*PQ0 / F*	*3*	*1*
P	*(DP1 / 3) COND / 5*	*(4) 1*	*1 0 1*
Q	*CQ1 / 1*	*6*	*1*

(c) A un certo istante $t_1 > t_0$ sono terminati, nell'ordine, gli eventi seguenti:

7. **Q** esegue una "exec" e passa a eseguire il programma Y *rilascia CQ0, CQ1, COND e PQ0, e alloca CQ2, DQ0 e PQ0*
8. **P** termina ("exit" di P) *rilascia CP0, DP2, PP0 e COND*
9. **Q** esegue un accesso a COND *alloca COND*
10. **Q** esegue una "fork" e crea **R** *alloca CR2 = CQ2, DR0 = DQ0, COND e PR0*

Si completino le tabelle con la situazione al tempo t_1.

situazione al tempo t_1

memoria fisica	
indirizzo fisico	**pagine allocate**
0	*(CP0) = (CQ0) COND*
1	*(COND) PR0*
2	*(PP0)*
3	*(PQ0) CQ2 = CR2*
4	*DQ0 = DR0*
5	*(DP2)*
6	*(CQ1) PQ0*
7	

MMU			
proc	**NPV**	**NPF**	**valid bit**
(P) Q	*(CP0 / 0) COND / 7*	*(0) 0*	*1 0 1*
(P) R	*(DP2 / 4) CR2 / 2*	*(5) 3*	*1 0 1*
(P) R	*(PP0 / F) DR0 / 3*	*(2) 4*	*1 0 1*
(Q) Q	*(CQ0 / 0) CQ2 / 2*	*(0) 3*	*1 0 1*
(Q) Q	*(COND / 5) DQ0 / 3*	*(1) 4*	*1 0 1*
(Q) Q	*(PQ0 / F) PQ0 / F*	*(3) 6*	*1 0 1*
(P) R	*(COND / 5) COND / 7*	*(1) 0*	*1 0 1*
(Q) R	*(CQ1 / 1) PR0 / F*	*(6) 1*	*1 0 1*

esercizio n. 4 – logica digitale

prima parte – logica combinatoria

Un circuito combinatorio riceve in ingresso un numero binario intero senza segno da **quattro bit** (X3 X2 X1 X0), dove ogni bit rappresenta un ingresso al circuito stesso (X3 è il bit più significativo e X0 è il bit meno significativo del numero). Il dispositivo è dotato di **un'uscita** F, che vale 1 solo se il numero in ingresso è maggiore o uguale a dieci.

(a) Si disegni la mappa di Karnaugh corrispondente alla prima forma canonica (somma di prodotti – SOP) della funzione F, si mettano in evidenza gli implicanti primi, e per ogni implicante primo individuato si scriva la corrispondente forma algebrica:

*ci sono **due** implicanti primi: (10, 11, 14, 15) = X3 X2 e (12, 13, 14, 15) = X3 X1*

X3 X2 | X1 X0	*00*	*01*	*11*	*10*
00				
01				
11	*1*	*1*	*1*	*1*
10			*1*	*1*

(b) Si sintetizzi la funzione F tramite il metodo delle mappe di Karnaugh e si scriva la forma minima come somma di prodotti (SOP). Qualora esistano più forme minime le si indichino tutte (il numero di righe date sotto non è significativo):

F (*X2*, *X1*, *Y2*, *Y1*) = *X3 X2 + X3 X1* _____

F (*X2*, *X1*, *Y2*, *Y1*) = _____

*Notiamo che gli implicanti primi sono entrambi essenziali, quindi c'è **una sola** forma di costo minimo, che ovviamente li usa entrambi.*

(c) Si disegni una rete combinatoria che realizza la funzione F sintetizzata al punto precedente, utilizzando solamente porte a **due ingressi**. *lasciato al lettore*

(d) Si calcoli il ritardo massimo (percorso critico) della rete disegnata al punto precedente, supponendo che ogni porta NOT abbia ritardo pari a **1 ns**, che ogni porta AND a due ingressi abbia ritardo pari a **2 ns** e che ogni porta OR a due ingressi abbia ritardo pari a **3 ns**:

ritardo (F) = *1 × 2 ns (strato AND) + 1 × 3 ns (strato OR) = 5 ns (i NOT non sono usati)* _

seconda parte – logica sequenziale

Sia dato il circuito sequenziale avente un ingresso I e un'uscita Z, dotato di **due bistabili** master / slave di tipo D: (D1, Q1) e (D0, Q0), descritto dalle equazioni logiche seguenti:

$$D1 = (!Q1)\ Q0 + Q1\ (!Q0) \qquad D0 = (!Q1)\ (!Q0) + Q1\ (!Q0)$$

$$Z = (!I)\ (!Q1)\ Q0 + I\ Q1\ (!Q0)$$

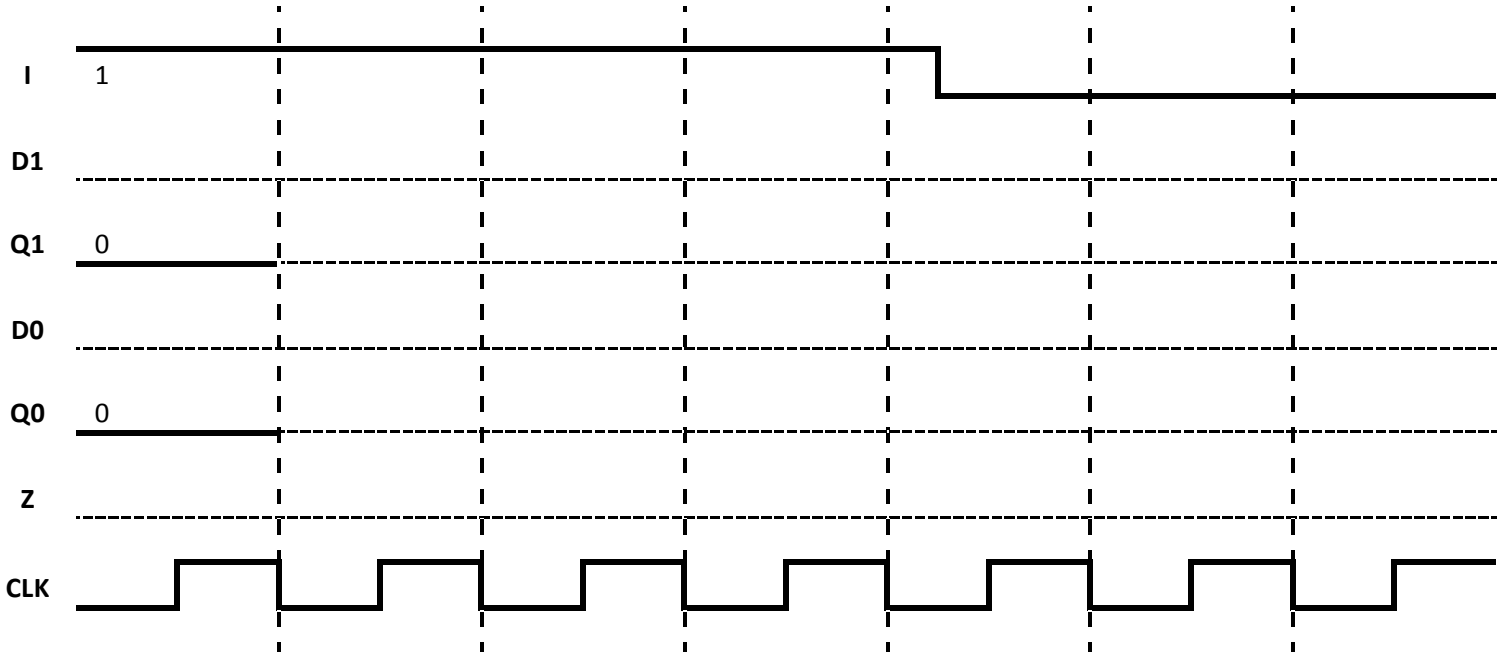
Si chiede di completare il diagramma temporale riportato qui sotto. Si noti che:

- si devono trascurare completamente i ritardi di propagazione delle porte logiche NOT, OR e AND, e i ritardi di commutazione dei bistabili
- i bistabili presenti nel circuito sono di tipo "master-slave " e hanno questo funzionamento:
    - sul **livello alto** del clock campionano e memorizzano l'ingresso
    - sul **fronte di discesa** del clock mostrano il valore memorizzato

  mentre negli altri momenti sono insensibili agli ingressi e mantengono fissa l'uscita; questo è il comportamento **usuale** del bistabile master-slave standard illustrato a lezione

---

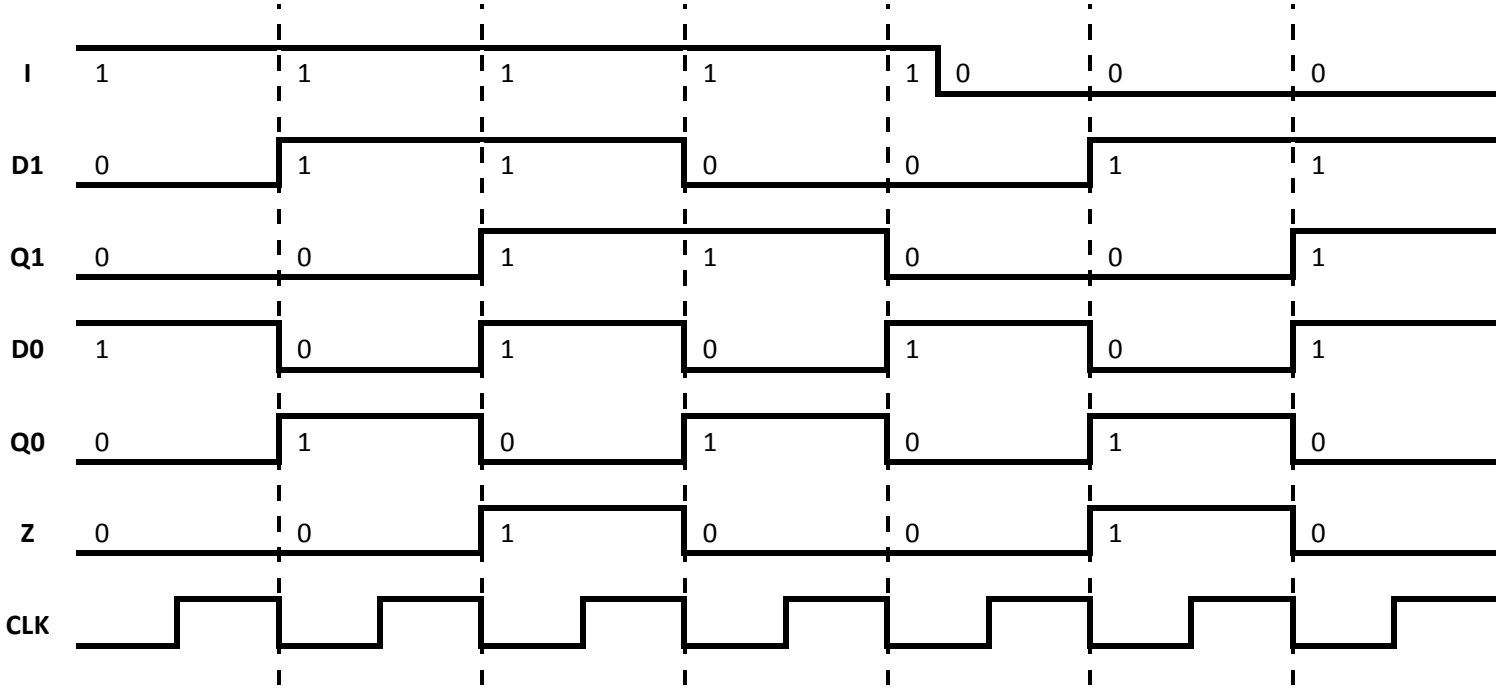
diagramma temporale da completare



soluzione

D1 = (!Q1) Q0 + Q1 (!Q0) D0 = (!Q1) (!Q0) + Q1 (!Q0)

Z = (!I) (!Q1) Q0 + I Q1 (!Q0)



Commenti

D1 e D0 dipendono dallo stato attuale e non dall'ingresso: indipendentemente da I continuano ad assumere i valori in sequenza 00, 01, 10, 11; di conseguenza (con il ritardo di un clock) anche lo stato Q1 Q0 assume in sequenza i valori 00, 01, 10, 11, e poi di nuovo 00, 01, 10, 11 ecc. ecc.

Si tratta quindi di un contatore binario naturale a 2 bit che conta gli impulsi di clock.

La sequenza di conteggio Q1 Q0 viene poi usata come base per generare:

a) se I = 0 la sequenza Z = 0, 1, 0, 0

b) se I = 1 la sequenza Z = 0, 0, 1, 0

(ovvio capire cosa succede a Z se I commuta proprio mentre la sequenza Z sta generando lo 1, più complesso capire il comportamento del circuito se I commuta quando il clock vale 1 …)

Il circuito potrebbe quindi essere usato per generare un 1 ogni 4 impulsi di clock (è una specie di divisore per 4 della frequenza del clock), ritardando lo 1 al secondo o al terzo clock in funzione del valore di I (e per semplicità potremmo imporre a chi usa il circuito di non commutare I quando il clock è alto …)

esercizio n. 5 – linguaggio macchina

prima parte – codifica in linguaggio macchina

Si deve tradurre in linguaggio macchina simbolico (linguaggio assemblatore) 68000 il programma (main e funzione *funz*) riportato qui sotto. Nel tradurre non si tenti di accorpare od ottimizzare insieme istruzioni C indipendenti.

La memoria ha parole da **16 bit**, indirizzi da **32 bit** ed è indirizzabile per byte. Le variabili intere sono da **16 bit**. Ulteriori specifiche al problema e le convenzioni da adottare nella traduzione sono le seguenti:

- i parametri di tutte le funzioni sono passati sulla pila in ordine inverso di elencazione
- i valori restituiti dalle funzioni ai chiamanti rispettivi sono passati sulla pila, sovrascrivendo il primo dei parametri passati o nello spazio libero opportunamente lasciato
- le variabili locali vengono impilate in ordine di elencazione
- le funzioni (tranne *main*) devono sempre salvare i registri che utilizzano

Si chiede di svolgere i **tre** punti seguenti:

1. **Si descriva** la struttura della memoria delle variabili globali e l'area di attivazione della funzione *funz*, secondo il modello 68000, nelle tabelle predisposte, indicando gli spiazzamenti rispetto a FP.
2. **Si dichiarino** in linguaggio macchina 68000 le variabili globali e **si scriva** il codice macchina della funzione *main*, coerentemente con le specifiche e le risposte precedenti, usando le tabelle predisposte.
3. **Si scriva** in linguaggio macchina 68000 il codice macchina della funzione *funz*, coerentemente con le specifiche e le risposte ai punti precedenti, usando le tabelle predisposte.

programma in linguaggio C

```
#define N = 5

/* variabili globali */

int dati [N];
int i;
int r = 0;

/* programma main (alcune parti sono volutamente omesse) */

void main ( ) {
   i = N - 1;
   do {
      if (dati [i] < dati [i - 1]) {
         r = r + funz (dati [i], i);
      }  /* fine if */
      i--;
   while (i >= 1);
   }  /* fine do */
} /* fine main */

/* funzione funz */

int funz (int a, int b) {
   return (a + r) * b;
} /* fine funzione */
```

domanda **1** (numero di righe non significativo)

	memoria variabili globali (domanda **1**)	
ind min	*dati [0]*	*- 2 byte*
	dati [1]	*- 2 byte*
	dati [2]	*- 2 byte*
	dati [3]	*- 2 byte*
	dati [4]	*- 2 byte*
	I	*- 2 byte*
	R	*- 2 byte*
ind max		

	area di attivazione di ***funz*** (domanda **1**)	
	registri salvati	
0	*FP precedente*	*- 4 byte*
+ 4	*indirizzo di rientro*	*- 4 byte*
+ 8	*A*	*- 2 byte*
+ 10	*B*	*- 2 byte*

dichiarazione delle **variabili globali** (domanda **2** – num. righe non signif.)

```
DATA:   ORG     1000            // indir. virt. segmento dati statici
N:      EQU     5               // costante N
DATI:   DS.W    N               // vettore DATI
I:      DS.W    1               // variabile I
R:      DC.W    0               // variabile R inizializzata a 0
```

codice 68000 di ***main*** (domanda **2** – num. righe non signif.) – *UN PO' OTTIMIZZATO*			
MAIN:	**LINK**	FP, #-0	// collega
	MOVE.W	#N, D0	// carica N
	SUBI.W	#1, D0	// calcola N - 1
	MOVE.W	D0, I	// memorizza I
	MOVEA.L	#DATI, A0	// inizializza A0
	MULS	#2, D0	// allinea indice (sizeof(int) = 2)
	ADDA.L	D0, A0	// calcola indir. ultimo elem. DATI
LOOP:	**MOVE.W**	(A0), D1	// carica DATI [I]
	SUBA.L	#2, A0	// aggiorna A0 (con allineamento)
	MOVE.W	(A0), D2	// carica DATI [I - 1]
	CMP.W	D2, D1	// confronta
	BGE	ENDIF	// se >= 0 va' a ENDIF
	MOVE.W	I, D0	// (ri)carica I
	MOVE.W	D0, -(SP)	// impila 2° param. di funz
	MOVE.W	D1,-(SP)	// impila 1° param. di funz
	BSR	FUNZ	// chiama funz
	ADDA.L	#2, SP	// abbandona 1° param. di funz
	MOVE.W	(SP)+, D3	// spila valusc di funz
	MOVE.W	R, D4	// carica R
	ADD.W	D3, D4	// addiziona
	MOVE.W	D4, R	// memorizza R
ENDIF:	**SUBI.W**	#1, D0	// calcola I - 1
	MOVE.W	D0, I	// memorizza I
	CMPI.W	#1, D0	// confronta
	BGE	LOOP	// se >= 0 va' a LOOP
	UNLK	FP	// scollega
	RTS		// rientra
	END	MAIN	// fine main

Per accedere al vettore, si ne carica l'indirizzo base in A0, poi si allinea l'indice I (gli elementi hanno taglia di due byte) e lo si addiziona ad A0; l'accesso usa l'indirizzamento indiretto da registro. Esistono soluzioni diverse. Il codice è un po' ottimizzato: si evita di ricaricare la variabile I e l'elemento DATI[I], se sono già nei registri D0 e D1 . Si può ottimizzare di più, sfruttando l'ortogonalità di alcune istruzioni.

codice in linguaggio macchina 68000 della funzione *f*unz (domanda **3**)			
FUNZ:	**LINK**	FP, #0	// area di attivazione di funz
B:	**EQU**	+10	// spiazzamento param b
A:	**EQU**	+8	// spiazzamento param a
	MOVEM.W	D0-D2, -(SP)	// salva registri
	MOVE.W	A(FP), D0	// carica param a
	MOVE.W	R, D1	// carica varglob r
	ADD.W	D0, D1	// addiziona
	MOVE.W	B(FP), D2	// carica param b
	MULS	D1, D2	// moltiplica
	MOVE.W	D2, B(FP)	// impila valusc
	MOVEM.W	(SP)+, D0-D2	// ripristina registri
	UNLK	FP	// scollega
	RTS		// rientra

seconda parte – assemblaggio e collegamento

Si supponga che la memoria abbia parole da **16 bit** e sia indirizzata per **byte**. Si svolgano i punti seguenti:

a) Nella tabella sotto, **si riportino** gli indirizzi dove collocare dati e istruzioni. Si consideri che ogni istruzione ha sempre una parola di codice operativo e, se serve, una o più parole aggiuntive. Si tenga conto che lo spiazzamento sia in istruzioni che manipolano dati sia in istruzioni di salto è sempre da **16 bit** e che indirizzi e costanti sono **corti** (16 bit) o **lunghi** (32 bit) come specifica l'istruzione.

			# parole	# byte	indirizzo (in decimale)
	ORG	5000			
ROW:	**EQU**	2			
COL:	**EQU**	3			
MAT:	**DS.L**	6	*12*	*24*	5000
SUM:	**DC.W**	0	*1*	*2*	*5024*
INIZ:	**MOVEA.L**	#MAT.L, A0	*3*	*6*	*5026*
	CLR.L	D1	*1*	*2*	*5032*
OUTER:	**CMPI.L**	#ROW.L, D1	*3*	*6*	*5034*
	BGE	FINISH	*2*	*4*	*5040*
	CLR.W	D2	*1*	*2*	*5044*
INNER:	**ADD.W**	(A0, D2), SUM.L	*3*	*6*	*5046*
	ADDI.W	#1.W, D2	*2*	*4*	*5052*
	CMPI.W	#COL.W, D2	*2*	*4*	*5056*
	BLT	INNER	*2*	*4*	*5060*
	ADDI.L	#ROW.L, A0	*3*	*6*	*5064*
	ADDI.L	#1.L, D1	*3*	*6*	*5070*
	BRA	OUTER	*2*	*4*	*5076*
FINISH:	**END**	INIZ			*5080*

b) Nella tabella data sotto, **si riportino** i simboli e i rispettivi valori numerici.

Tabella dei simboli	
nome simbolo	*valore numerico (in decimale)*
ROW	*2*
COL	*3*
MAT	*5000*
SUM	*5024*
INIZ	*5026*
OUTER	*5034*
INNER	*5046*
FINISH	*5080*

c) **Si dica** quanto vale in decimale lo spiazzamento codificato nell'istruzione BGE:

spiazzamento in BGE = *5080 – 5044 = 36*

d) **Si dica** quanto vale in decimale lo spiazzamento codificato nell'istruzione BLT:

spiazzamento in BLT = *5046 – 5064 = –18*

e) **Si dica** quale valore (decimale) verrà caricato nel registro PC al lancio del programma: 5026

esercizio n. 6 – memoria cache ed efficienza

prima parte – simulazione

Si consideri un sistema di memoria (centrale + cache) caratterizzato dalle dimensioni seguenti:

memoria di lavoro da **4 K Byte** (indirizzata a livello di byte)
memoria cache da **512 Byte**
ogni blocco di cache contiene **128 Byte**

Considerando la sequenza di richieste alla memoria riportata qui sotto, si chiede di completare la tabella che illustra il comportamento di una cache set-associativa a **due vie** (associativa a gruppi a due vie) nel rispetto delle indicazioni seguenti:

Nella colonna esito, riportare H (hit - successo) se il blocco richiesto si trova in cache, oppure riportare M (miss - fallimento) se il blocco va caricato da memoria.

Nelle colonne dati va riportato il numero del blocco di memoria che si trova nel corrispondente blocco di cache. Questi valori sono denotati come numeri decimali (base dieci), mentre le etichette sono in binario. Pertanto l'indirizzo 0000 0001 0010 individua un byte nel blocco $00000_{due} = 0_{dieci}$.

Nella colonna azione va indicato il blocco cui si accede (in caso di successo H) o il blocco in cui vengono caricati i dati della memoria (in caso di fallimento M).

Nella cache ci sono quattro blocchi, indicati con A, B, C e D, che sono organizzati in due insiemi: i blocchi A e B appartengono all'insieme 0, e i blocchi C e D appartengono all'insieme 1.

La politica di sostituzione adottata per la cache è quella LRU (Least Recently Used).

Nota: la memoria ha 12 bit di indirizzo; di questi, 7 bit servono per individuare il byte nel blocco; nella cache ci sono quattro blocchi organizzati in due gruppi, dunque lo 8° bit (da destra) indica l'insieme e i 4 bit più significativi restanti costituiscono l'etichetta.

passo	indirizzo richiesto	esito	blocco A			blocco B			blocco C			blocco D			azione
			valido	etichetta	dati	valido	etichetta	dati	valido	etichetta	dati	valido	etichetta	dati	
0			**1**	**0000**	**0**	**0**	**-**	**-**	**1**	**0001**	**3**	**0**	**-**	**-**	**situazione iniziale**
1	1110 1110 0010	*M*	*1*			*0*			*1*			*1*	*1110*	*29*	*carica blocco 29 in D*
2	0001 1110 0101	*H*	*1*			*0*			*1*	*0001*	*3*	*1*			*accedi a blocco 3 in C*
3	1111 0011 1001	*M*	*1*			*1*	*1111*	*30*	*1*			*1*			*carica blocco 30 in B*
4	0100 0000 0010	*M*	*1*	*0100*	*8*	*1*			*1*			*1*			*carica blocco 8 in A*
5	0010 1111 1110	*M*	*1*			*1*			*1*			*1*	*0010*	*5*	*carica blocco 5 in D*

seconda parte – valutazione delle prestazioni

Si consideri un sistema di memoria costituito da una memoria centrale di **1 M parole** e da una cache set-associativa a **2 vie** (associativa a gruppi a 2 vie) di **2 K parole** con blocchi da **32 parole**, che utilizza un algoritmo di sostituzione del blocco di tipo **LRU**.

Inizialmente la cache è vuota. La CPU esegue un ciclo che preleva sequenzialmente dalla memoria un array di **2208** elementi di una parola, corrispondenti a **69 blocchi**, partendo dall'indirizzo **0**. Il ciclo viene eseguito per **2** volte.

1) **Mostrare** la struttura dell'indirizzo di memoria (nella tabella seguente).

etichetta	indice	spiazz.
10 bit	*5 bit*	*5 bit*

La memoria ha 1 M parole: 20 bit per l'indirizzo.

I blocchi sono da 32 parole: 5 bit per identificare la parola nel blocco.

Nella cache (2 KB) ci sono: $2^{11} / 2^5 = 2^6 = 64$ blocchi. La memoria cache è set-associativa a due vie, dunque avrà $2^6 / 2 = 2^5 = 32$ insiemi di due blocchi. Sono pertanto necessari 5 bit per individuare l'insieme. L'etichetta è dunque di 10 bit.

2) **Completare** la tabella seguente indicando il numero decimale dei blocchi dell'array che appartengono agli insiemi indicati (parte della prima riga è già compilata).

insieme (gruppo)	blocchi
0	0, 32*, 64*
1	*1, 33, 65*
2	*2, 34, 66*
3	*3, 35, 67*
4	*4, 36, 68*

Tutti i blocchi che in memoria hanno come valore 00000 per i bit di insieme, quindi i blocchi 0, 32 e 64 finiranno nell'insieme 00000, tutti i blocchi che hanno indirizzo terminante con 00001 (1, 33 e 65) finiranno nell'insieme 00001, tutti i blocchi che terminano con 00010 (2, 34 e 66) finiranno nell'insieme 00010, e così via.

3) **Indicare** il numero di MISS che si verificano durante la prima esecuzione del ciclo (nella tabella predisposta), spiegando a lato sinteticamente il motivo.

n. di miss nella prima iterazione del ciclo
69 miss

Nel primo ciclo per ogni blocco del vettore si genera un miss, poiché inizialmente la memoria cache è vuota; pertanto si verificano 69 miss.

4) **Indicare** nella seguente tabella il numero decimale dei blocchi contenuti negli insiemi indicati al termine della prima iterazione del ciclo, la quale legge **2208** elementi.

insieme (gruppo)	blocchi
0	*32, 64*
1	*33, 65*
…	
4	*36, 68*
5	*5, 37*
6	*6, 38*
…	
31	*31, 63*

La cache si riempie con il blocco 63 che apparterrà all'insieme 31.

Quando bisogna caricare il blocco numero 64 si osserva che ha i bit di insieme pari a 00000, dunque il suo "posto" è nell'insieme 00000: tutto l'insieme è occupato e non c'è il blocco 64, dunque si verifica un miss e con l'algoritmo LRU verrà rimosso il blocco 0.

In modo analogo il blocco 65 finirà nell'insieme 00001 dove verrà rimosso il blocco 1 e così fino al blocco 68 che finirà nell'insieme 4 sostituendo il blocco 4.

5) **Calcolare** il numero di MISS che si verificano durante **la seconda iterazione del ciclo**. Si scriva il risultato nella tabella predisposta (e a lato si illustri il calcolo).

n. di miss nella seconda iterazione del ciclo
15 miss

Nella seconda iterazione, quando serve il blocco 0 si verifica un miss e nella cache viene sostituito il blocco 32; quando serve il blocco 1, si verifica un miss e viene sostituito il blocco 33; quando serve il blocco 2, si verifica un miss e viene sostituito il blocco 34; così come avviene per il blocco 3 che sostituisce il blocco 35, e il blocco 4 che sostituisce il blocco 36. Dunque si verificano 5 miss.

Invece il blocco 5 e i successivi sono già in memoria. Avremo hit fino al blocco 31, quando si verificano 5 miss di seguito che andranno a sostituire i blocchi 64, 65, …, 68 con i blocchi 32, 33, …, 36, rispettivamente. Poi avremo hit fino al blocco 63, quando si verificano 5 miss di seguito che andranno a sostituire i blocchi 0, 1, …, 4 con i blocchi 64, 65, …, 68, rispettivamente. In totale si verificano dunque 15 miss nella seconda iterazione del ciclo.

6) **Determinare** i seguenti tempi di esecuzione e derivarne il fattore di miglioramento risultante dall'utilizzo della cache facendo l'ipotesi che il tempo di accesso alla memoria cache sia di **1 ns**, il tempo di accesso alla memoria centrale sia di **10 ns**, e che il bus dati abbia la dimensione della parola. Indicare per ognuno la formula utilizzata oltre al risultato del calcolo.

Tempo senza cache =

2 iterazioni × 2208 elementi × 10 ns = 44160 ns

Tempo con cache prima iterazione =

69 blocchi × 32 elementi nel blocco × (10 + 1) ns = 24288 ns

Tempo con cache seconda iterazione =

(15 blocchi miss × 32 elementi nel blocco × (10 + 1) ns + 54 blocchi hit × 32 elementi nel blocco × 1 ns) = 5280 ns + 1728 ns = 7008 ns

Tempo con cache (totale) =

44160 ns + 7008 ns = 31296 ns

Fattore di miglioramento =

tempo senza cache / tempo con cache = 44160 ns / 31296 ns ≈ 1,41 ossia circa 41 %

spazio libero per brutta o continuazione

spazio libero per brutta o continuazione

spazio libero per brutta o continuazione