



Politecnico di Milano

Dip. di Elettronica, Informazione e Bioingegneria

prof. Luca Breveglieri
prof. Gerardo Pelosi

prof.ssa Donatella Sciuto
prof.ssa Cristina Silvano

AXO – Architettura dei Calcolatori e Sistemi Operativi

PRIMA PARTE – lunedì 17 luglio 2023

Cognome _____ **Nome** _____

Matricola _____ **Firma** _____

Istruzioni

Si scriva solo negli spazi previsti nel testo della prova e non si separino i fogli.

Per la minuta si utilizzino le pagine bianche inserite in fondo al fascicolo distribuito con il testo della prova. I fogli di minuta, se staccati, vanno consegnati intestandoli con nome e cognome.

È vietato portare con sé libri, eserciziari e appunti, nonché cellulari e altri dispositivi mobili di comunicazione. 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 **1 h : 30 m**

Valore indicativo di domande ed esercizi, voti parziali e voto finale:

esercizio 1 (6 punti) _____

esercizio 2 (2 punti) _____

esercizio 3 (6 punti) _____

esercizio 4 (2 punti) _____

voto finale: (16 punti) _____

CON SOLUZIONI (in corsivo)

esercizio n. 1 – linguaggio macchina

prima parte - traduzione da C ad assembler

Si deve tradurre in linguaggio macchina simbolico (assemblatore) **RISC-V** il frammento di programma C riportato sotto. Il modello di memoria è quello **standard RISC-V** e le variabili intere sono da **64 bit**. Non si tenti di accoppiare od ottimizzare insieme istruzioni C indipendenti. Si facciano le ipotesi seguenti:

- il registro “frame pointer” *fp* **non è in uso**
- le variabili locali sono allocate nei registri, se possibile
- vanno **salvati** (a cura del chiamante o del chiamato, secondo il caso) **solo i registri necessari**
- l’allocazione delle variabili in memoria **è non allineata** (non c’è **frammentazione** di memoria)

Si chiede di svolgere i quattro punti seguenti (usando le varie tabelle predisposte nel seguito):

1. **Si descriva** il segmento dei dati statici indicando gli indirizzi assoluti iniziali delle variabili globali e **si traducano** in linguaggio macchina le dichiarazioni delle variabili globali.
2. **Si descriva** l’area di attivazione della funzione `mirror`, secondo il modello RISC V, e l’allocazione dei parametri e delle variabili locali della funzione `mirror` usando le tabelle predisposte.
3. **Si traduca** in linguaggio macchina **il codice degli statement riquadrati nella funzione** `main`.
4. **Si traduca** in linguaggio macchina il codice **dell’intera funzione** `mirror` (vedi tab. 4 strutturata).

```
/* costanti e variabili globali */
#define N 24 /* costante da 32 bit */
typedef long long int LONG
LONG code
char STRING [N]
LONG errcode = -2
/* testata procedura ausiliaria - è una procedura foglia */
LONG putchar (char alpha) /* scrive un carattere su output */
/* funzione mirror */
LONG mirror (LONG size, char * buffer) {
    LONG status
    size = size - 1
    if (size >= 0) {
        status = putchar (buffer[size])
        if (status != errcode) {
            return mirror (size, buffer)
        } /* if 2 */
    } /* if 1 */
    return size
} /* mirror */
/* programma principale */
void main ( ) {
    code = mirror (N, STRING) + 1
} /* main */
```

punto 1 – segmento dati statici

contenuto simbolico	indirizzo assoluto iniziale (in hex)	
		indirizzi alti
ERRCODE	<i>0x 0000 0000 1000 0020</i>	
STRING	<i>0x 0000 0000 1000 0008</i>	
CODE	0x 0000 0000 1000 0000	indirizzi bassi

punto 1 – codice della sezione dichiarativa globale (numero di righe non significativo)			
	.eqv	N, 24	// costante numerica
	.data	0x 0000 0000 1000 0000	// seg. dati statici standard
CODE:	.space	8	// varglob CODE (64 bit non inizializ.)
STRING:	.space	24	// varglob STRING (vettore non inizializ.)
ERRCODE:	.dword	-2	// varglob ERRCODE (64 bit inizializ.)

punto 2 – area di attivazione della funzione MIRROR		
contenuto simbolico	spiazz. rispetto a stack pointer	
<i>ra salvato</i>	<i>+8</i>	indirizzi alti
<i>s0 salvato</i>	<i>+0</i>	← <i>sp (fine area)</i>
<i>reg a2 (param SIZE) salvato</i>		<i>max estensione pila di MIRROR</i>
		indirizzi bassi

La funzione `mirror` riutilizza l'argomento `SIZE` (reg `a2`) dopo avere chiamata la funzione ausiliaria `putchar`, la quale ne fa uso per il suo argomento `ALPHA`, dunque la funzione `mirror` salva l'argomento `SIZE` (reg `a2`) in pila e lo ripristina a ogni chiamata di `putchar`.

punto 2 – allocazione dei parametri e delle variabili locali di MIRROR nei registri	
parametro o variabile locale	registro
<i>size</i>	<i>a2</i>
<i>buffer</i>	<i>a3</i>
<i>status</i>	<i>s0</i>

punto 3 – codice dello statement riquadrato in MAIN (num. righe non significativo)	
// code = mirror (N, STRING) + 1	
MAIN: <i>li</i>	<i>a2, N // prepara param SIZE</i>
<i>la</i>	<i>a3, STRING // prepara param BUFFER</i>
<i>jal</i>	<i>MIRROR // chiama funz MIRROR</i>
<i>addi</i>	<i>t0, a0, 1 // calcola espr mirror (...) + 1</i>
<i>la</i>	<i>t1, CODE // carica ind varglob CODE</i>
<i>sd</i>	<i>t0, (t1) // aggiorna varglob CODE</i>

punto 4 – codice della funzione MIRROR (numero di righe non significativo)

```
MIRROR:  addi  sp, sp, -16           // COMPLETARE - crea area attivazione
// direttive EQV - DA COMPLETARE
.eqv  RA, 8                     // spi di reg ra salvato
.eqv  S0, 0                     // spi di reg s0 salvato
// salvataggio registri - NON VA RIPORTATO
// size = size - 1
addi   a2, a2, -1              // aggiorna arg SIZE

IF1:     // if (size >= 0)
blt    a2, zero, ENDIF1        // se size < 0 size vai a ENDIF1

// status = putchar(buffer[size])
add    t0, a3, a2              // calcola ind di elem BUFFER[SIZE]
addi   sp, sp, -8              // salva arg SIZE - push a2
sd     a2, (sp)                // fine push
lb     a2, (t0)                // prepara arg ALPHA (con load byte)
jal    PUTCHAR                 // chiama funz PUTCHAR
ld     a2, (sp)                // ripristina arg SIZE - pop a2
addi   sp, sp, 8               // fine pop
mv     s0, a0                  // aggiorna varloc STATUS

IF2:     // if (status != errcode)
la     t0, ERROCODE            // carica ind di varglob ERROCODE
ld     t1, (t0)                // carica varglob ERROCODE
beq    s0, t1, ENDIF2          // se status = errocode vai a ENDIF2

// return mirror(size, buffer)
jal    MIRROR                  // chiama (ricorsivamente) funz MIRROR
j      RETURN                  // vai a conclusione

ENDIF2:  // fine IF2
ENDIF1:  // fine IF1
// return size
mv     a0, a2                  // prepara valusc

RETURN:  // ripristino registri, elim. area e rientro - NON VA RIPORTATO
```

Seconda parte - Rappresentazione di struttura dati in assembler

Si consideri la dichiarazione, in linguaggio C, di un *array* di **struct** con campi di tipo scalare:

```
#define N 4

struct {                                // array di N struct
    int number                          // un campo di tipo int
    char c1, c2, c3, c4                // quattro campi di tipo char
} BLOCK [N]
```

Si chiede di tradurre in linguaggio assembler RISC V (con direttive se opportuno) l'istruzione C di assegnamento seguente, che usa l'*array* `BLOCK` e la **variabile globale** `idx` (l'inizializzazione di `idx` non è rilevante). In memoria, i campi di una **struct** sono rappresentati in successione, nell'ordine in cui figurano nella **struct**.

L'indirizzamento di memoria non è allineato, ossia non si lasciano spazi vuoti tra variabili.

```
BLOCK[idx].c2 = '@'    // codice ASCII di '@' = 64
```

Si risponda alle domande seguenti:

- 1) Calcolare l'ingombro della **struct**, in byte: $\text{sizeof(struct)} = 4 + 1 + 1 + 1 = 8 \text{ byte}$
8 byte // 4 byte (number) + 4 byte (c1,c2,c3,c4)
- 2) Calcolare lo spiazzamento, in byte, del campo `c2` nella struct: $\text{spi di c2} = 4 + 1 = 5 \text{ byte}$
5 // dovuto a 4 byte(number)+1 byte c1
- 3) Si completi la tabella seguente (alcune righe sono già compilate), compresa la direttiva `.space`, scrivendo il codice assembler RISC V che realizza l'istruzione di assegnamento:

	<code>.eqv N, 4</code>	<code>// N = 4</code>
	<code>.align 0</code>	<code>// indirizzamento NON ALLINEATO (no spazi vuoti)</code>
IDX:	<code>.dword ...</code>	<code>// varglob IDX</code>
BLOCK:	<code>.space 32</code>	<code>// (8 byte di ingombro per la struct) * 4 = 32 byte</code>
	<code>// BLOCK[idx].c2 = '@'</code>	
	<code>li t0, 64</code>	<code>// carica cost ASCII '@'</code>
	<code>la t1, BLOCK</code>	<code>// carica ind. iniziale di array BLOCK</code>
	<code>la t2, IDX</code>	<code>// carica ind. di varglob IDX</code>
	<code>ld t3, (t2)</code>	<code>// carica varglob IDX</code>
	<code>slli t3, t3, 3</code>	<code>// allinea indice - sizeof (struct) = 8 byte</code>
	<code>add t1, t1, t3</code>	<code>// calcola ind. di elem BLOCK [idx]</code>
	<code>sb t0, 5(t1)</code>	<code>// aggiorna elem. BLOCK [idx].c2 - si usa store byte</code>

*Nota: invece della singola istruzione **sb** `t0, 5(t1)`, si può anche usare la sequenza **add** `t1, t1, 5` e **sb** `t0, (t1)`.*

esercizio n. 2 – logica digitale

logica sequenziale

Sia dato il circuito sequenziale composto da due bistabili master-slave di *tipo D* (D1, Q1 e D2, Q2, dove D è l'ingresso del bistabile e Q è lo stato / uscita del bistabile), un ingresso **I** e un'uscita **U**, e descritto dalle equazioni nel riquadro.

$$D1 = Q2 \text{ nand } I$$

$$D2 = Q1 \text{ xor } I$$

$$U = \text{not } (D1 \text{ xor } D2)$$

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

- si devono trascurare completamente i ritardi di propagazione delle porte logiche e i ritardi di commutazione dei bistabili
- i bistabili sono il tipo master-slave la cui struttura è stata trattata a lezione, con uscita che commuta sul fronte di discesa del clock

tabella dei segnali (diagramma temporale) da completare

- per i segnali D1, Q1, D2, Q2 e U, **si ricavi**, per ogni ciclo di clock, l'andamento della forma d'onda corrispondente riportando i relativi valori 0 o 1
- notare che nel primo intervallo i segnali Q1 e Q2 sono già dati (rappresentano lo stato iniziale)

I	0	0	1	1	1	0	0	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	1	0	0	1	1
D1	1	1	1	1	0	1	1	0	1	1	1	1	0	1	1	0	1	1	1	1	0	0	1	0	1	1	1	1
Q1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
D2	0	0	1	1	0	1	1	0	0	1	0	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	1
Q2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
U	0	0	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0	0	1	1
CLK	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

esercizio n. 3 – microarchitettura del processore pipeline

prima parte – pipeline e segnali di controllo

Sono dati il seguente frammento di codice **macchina RISC V** (simbolico), che inizia l'esecuzione all'indirizzo indicato, e i valori iniziali per alcuni registri e parole di memoria – **notazione: 0⁴ = 0000, e così via.**

indirizzo hex a 64 bit	codice RISC V	registro	contenuto iniz - hex 64 bit
0 ⁴ 0 ⁴ 0040 0800	ld t1, 0x 08B(t0)	t0	0 ⁴ 0 ⁴ 1001 4021
0 ⁴ 0 ⁴ 0040 0804	add t2, t3, t3	t1	0 ⁴ 0 ⁴ 0001 CCCC
0 ⁴ 0 ⁴ 0040 0808	sd t3, 0x 0AB(t0)	t2	0 ⁴ 0 ⁴ 0001 80AA
0 ⁴ 0 ⁴ 0040 080C	add t4, t1, t3	t3	0 ⁴ 0 ⁴ 0010 800A
0 ⁴ 0 ⁴ 0040 0810	sd t1, 0x 0B3(t0)	memoria contenuto iniz - hex 64 bit	
0 ⁴ 0 ⁴ 0040 0814		0 ⁴ 0 ⁴ 1001 4004	0 ⁴ 0 ⁴ 1234 AA00
		0 ⁴ 0 ⁴ 1001 4008	0 ⁴ 0 ⁴ 1001 1B1B
		0 ⁴ 0 ⁴ 1001 40AC	0 ⁴ 0 ⁴ 1001 1A1A (t1 finale)
		0 ⁴ 0 ⁴ 1001 40CC	0 ⁴ 0 ⁴ 1001 FFCC

La pipeline è ottimizzata per la gestione dei conflitti di controllo, e si consideri il **ciclo di clock 5** in cui l'esecuzione delle istruzioni nei vari stadi è la seguente:

		ciclo di clock										
		1	2	3	4	5	6	7	8	9	10	11
istruzione	1 – ld	IF	ID	EX	MEM	WB						
	2 – add		IF	ID	EX	MEM	WB					
	3 – sd			IF	ID	EX	MEM	WB				
	4 – add				IF	ID	EX	MEM	WB			
	5 – sd					IF	ID	EX	MEM	WB		

1) Calcolare il valore dell'indirizzo di memoria dati nell'istruzione **ld t1 (load)**:

$$1001\ 4021 + 0000\ 008B = 1001\ 40AC$$

2) Calcolare il valore del risultato (t3 + t3) dell'istruzione **add t2, t3, t3**:

$$0010\ 800A + 0010\ 800A = 0021\ 0014\ (t2\ finale)$$

3) Calcolare il valore dell'indirizzo di memoria dati nell'istruzione **sd t3 (store)**:

$$1001\ 4021 + 0000\ 00AB = 1001\ 40CC$$

4) Calcolare il valore dell'indirizzo di memoria dati nell'istruzione **sd t1 (store)**:

$$1001\ 4021 + 0000\ 00B3 = 1001\ 40D4$$

Completare le tabelle.

I campi di tipo *Istruzione* e *NumeroRegistro* possono essere indicati in forma simbolica, tutti gli altri in esadecimale (prefisso 0x implicito). Utilizzare **n.d.** se il valore non può essere determinato. N.B.: **tutti** i campi vanno completati con valori simbolici o numerici, tranne quelli precompilati con *****.

segnali all'ingresso dei registri di interstadio (subito prima del fronte di SALITA del clock --- ciclo 5)			
IF	ID	EX	MEM
(sd)	(add)	(sd)	(add)
registro IF/ID	registro ID/EX	registro EX/MEM	registro MEM/WB
	.WB.MemtoReg 0	.WB.MemtoReg X	.WB.MemtoReg 0
	.WB.RegWrite 1	.WB.RegWrite 0	.WB.RegWrite 1
	.M.MemWrite 0	.M.MemWrite 1	
	.M.MemRead 0	.M.MemRead 0	
	.M.Branch 0	.M.Branch 0	
.PC 0 ⁴ 0 ⁴ 0040 0810	.PC 0 ⁴ 0 ⁴ 0040 080C	.PC *****	
.istruzione sd	.(Rs1) (t1) finale 0 ⁴ 0 ⁴ 1001 1A1A		
	.(Rs2) (t3) 0 ⁴ 0 ⁴ 0010 800A	.(Rs2) (t3) 0 ⁴ 0 ⁴ 0010 800A	
	.Rd t4 1D	.Rd *****	.Rd t2 07
	.imm/offset est. 64-bit *****	.ALU_out ind mem sd 0 ⁴ 0 ⁴ 1001 40CC	.ALU_out (t2) finale 0 ⁴ 0 ⁴ 0021 0014
	.EX.ALUSrc 0	.Zero *****	.DatoLetto *****

segnali relativi al RF (subito prima del fronte di DISCESA interno al ciclo di clock – ciclo 5)		
RF.RegLettura1 t1 add	RF.DatoLetto1 0 ⁴ 0 ⁴ 0001 CCCC (t1) iniz.	RF.RegScrittura t1 ld
RF.RegLettura2 t3 add	RF.DatoLetto2 0 ⁴ 0 ⁴ 0010 800A (t3) iniz.	RF.DatoScritto 0 ⁴ 0 ⁴ 1001 1A1A (t1) fin.

seconda parte – gestione di conflitti e stalli

Si consideri la sequenza di istruzioni sotto riportata eseguita in modalità pipeline:

			ciclo di clock									
istruzione			1	2	3	4	5	6	7	8	9	10
1	ld	t2, 0x 140 (t3)	IF	ID 3	EX	MEM	WB 2					
2	add	t2, t2, t2		IF	ID 2	EX	MEM	WB 2				
3	add	t4, t2, t4			IF	ID 2, 4	EX	MEM	WB 4			
4	addi	t3, t3, 8				IF	ID 3	EX	MEM	WB 3		
5	sd	t4, 0x 0CC (t3)					IF	ID 3, 4	EX	MEM	WB	

Si risponda alle domande seguenti:

punto 1

- Definire **tutte le dipendenze di dato** completando la **tabella 1** della pagina successiva (colonne "**punto 1a**") indicando quali generano un conflitto, e per ognuna di queste quanti stalli sarebbero necessari per risolvere tale conflitto (stalli teorici), considerando la pipeline **senza** percorsi di propagazione.
- Disegnare in **diagramma A** il diagramma temporale della pipeline senza propagazione di dato, con gli stalli **effettivamente** risultanti, e riportare il loro numero in **tabella 1** (colonne "**punto 1b**").

diagramma A

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1. ld	IF	ID 3	EX	M	WB 2											
2. add		IF	ID stall	ID stall	ID 2	EX	M	WB 2								
3. add			IF stall	IF stall	IF	ID stall	ID stall	ID 2, 4	EX	M	WB 4					
4. addi						IF stall	IF stall	IF	ID 3	EX	M	WB 3				
5. sd									IF	ID stall	ID stall	ID 3, 4	EX	M	WB	

punto 2

Si faccia l'ipotesi che la pipeline sia **ottimizzata** e dotata dei seguenti percorsi di propagazione: **EX / EX**, **MEM / EX**, **MEM / MEM**:

- Disegnare in **diagramma A** il diagramma temporale della pipeline, indicando i **percorsi di propagazione** che devono essere attivati per risolvere i conflitti e gli eventuali **stalli** da inserire affinché la propagazione sia efficace.
- Indicare in **tabella 1** le dipendenze, i percorsi di propagazione attivati con gli stalli associati, e il ciclo di clock nel quale sono attivi i percorsi di propagazione.

diagramma B

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1. ld	IF	ID 3	EX (2)	M (2)	WB 2										
2. add		IF	ID stall	ID 2	EX (2)	M (2)	WB 2								
3. add			IF stall	IF	ID (2, 4)	EX (4)	M (4)	WB 4							
4. addi					IF	ID 3	EX (3)	M (3)	WB 3						
5. sd						IF	ID 3, 4	EX	M	WB					

tabella 1

punto 1a					punto 1b	punto 2b	
N° istruzione	N° istruzione da cui dipende	registro coinvolto	conflitto (si/no)	N° stalli teorici	N° stalli effettivi	stalli + percorso di propagazione	ciclo di clock in cui è attivo il percorso
2	1	2	si	2	2	1 stallo + MEM / EX	5
3	1	2	no	--	--	--	--
3	2	2	si	2	2	EX / EX	6
5	3	4	si	1	assorbito	MEM / EX	8
5	4	3	si	2	2	EX / EX	8

esercizio n. 4 – logica combinatoria

prima parte – semplificazione algebrica

Si consideri la funzione booleana di quattro variabili $F(a, b, c, d)$ descritta dall'espressione seguente:

$$F(a, b, c, d) = (a + !(b \cdot c)) \cdot (a + b + c) \cdot (a + !b + d)$$

Si trasformi – tramite le proprietà dell'algebra di commutazione – l'espressione di F in modo da ridurre il costo della sua realizzazione, indicando per nome la singola trasformazione svolta oppure la forma della proprietà utilizzata. Allo scopo si usi la tabella seguente (il numero di righe non è significativo)

$(a + !(b \cdot c)) \cdot (a + b + c) \cdot (a + !b + d)$	<i>De Morgan $!(b \cdot c) = !b + !c$</i>
$(a + b + !c) \cdot (a + b + c) \cdot (a + !b + d)$	<i>Distributiva $(x+y) \cdot (x+z) = x + y \cdot z$, considerando $x == (a+b)$, $y == !c$, $z == c$</i>
$(a + b + !c \cdot c) \cdot (a + !b + d)$	<i>Inverso: $!x \cdot x == 0$ considerando $x == c$</i>
$(a + b) \cdot (a + !b + d)$	<i>Distributiva $(x+y) \cdot (x+z) = x + y \cdot z$ considerando $x == a$, $y == b$, $z == !b+d$</i>
$a + b \cdot (!b + d)$	<i>Distributiva: $x \cdot (y+z) = x \cdot y + x \cdot z$ considerando $x == b$, $y == !b$, $z == d$</i>
$a + b \cdot !b + b \cdot d$	<i>Inverso: $x \cdot !x == 0$ considerando $x == b$</i>
$a + b \cdot d$	

Si consideri una rete combinatoria a 3 ingressi A, B e C, e due uscite U1, U2, la cui tabella della verità è riportata sotto:

A	B	C	U1	U2
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

- 1) Si sintetizzino le uscite U1 e U2 in prima forma canonica (somma di prodotti – SoP) e si riportino le espressioni booleane corrispondenti, senza ricorrere a semplificazioni o minimizzazioni.

$$U1_{SoP} = \neg A \cdot B \cdot \neg C + \neg A \cdot B \cdot C + A \cdot \neg B \cdot \neg C$$

$$U2_{SoP} = \neg A \cdot \neg B \cdot C + \neg A \cdot B \cdot C + A \cdot \neg B \cdot \neg C$$

- 2) Si calcoli il costo (C_{SoP}) e il ritardo (R_{SoP}) della rete combinatoria che realizza congiuntamente il calcolo di U1 e U2, entrambe sintetizzate tramite la prima forma canonica.
Si supponga di avere a disposizione soltanto porte AND e OR a tre ingressi, con identico ritardo (si supponga che il ritardo sia unitario), e porte NOT con ritardo trascurabile.
Per il costo si consideri il numero di porte AND e OR a tre ingressi necessarie per l'intero circuito.

Il circuito combinatorio congiunto vede 2 porte AND a tre ingressi le cui uscite possono essere utilizzate per sintetizzare un risultato parziale valido sia per U1 sia per U2 (cioè il risultato di: $\neg A \cdot B \cdot C + A \cdot \neg B \cdot \neg C$).
Quindi il circuito finale sarà composto da (3 porte NOT) 4 porte AND a tre ingressi e 2 porte OR a tre ingressi con costo e ritardo complessivi come segue:

$$C_{SoP} = 6, R_{SoP} = 2$$

spazio libero per continuazione o brutta copia