Esercitazione 04

Introduzione a RISC-V

Gianluca Brilli
gianluca.brilli@unimore.it

Strumenti Consigliati

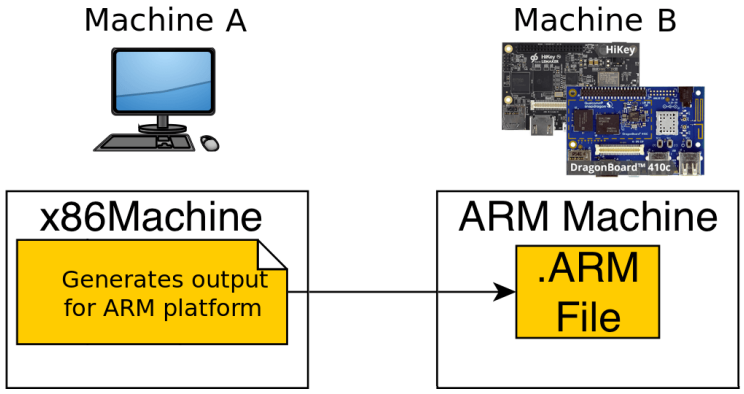
- Distribuzione Linux, **Ubuntu LTS**.
- Il vostro **editor testuale** preferito
  - Vim, Gedit, Kate, …
  - Consigliato VS Code: *https://code.visualstudio.com/*
- Un **gestore di terminali,** (dal momento che ce ne serviranno un tot aperti).
  - Es Terminator: *$ sudo apt-get install terminator*

Setup ambiente RISC-V (1)

- Diversi strumenti Necessari:
  - simulatore dell'architettura RISC-V
  - compilatore gcc,
  - assemblatore,
  - Debugger;
  - ...
- riscv-tools
- riscv-qemu

Setup ambiente RISC-V (2)

- **Cross-Compilazione**: consente di ottenere un file binario eseguibile per un elaboratore con architettura diversa da quella della macchina su cui si è lanciato il cross-compilatore stesso.
- **Emulazione**: consente di emulare via software un processore di una particolare architettura. Nel nostro caso useremo **qemu** per emulare RISC-V.



Setup ambiente RISC-V (8)

- Scaricare la macchina virtuale presente sulla pagina del corso.
- Andiamo ad estrarre il contenuto dell'archivio zip e configuriamo opportunamente VirtualBox

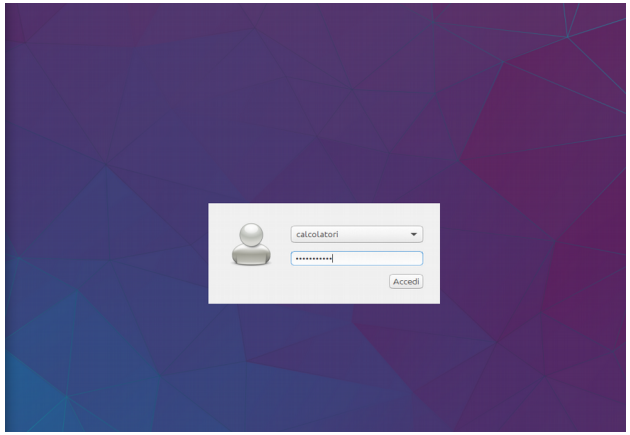
- Assegnamo un nome alla macchina.

Setup ambiente RISC-V (8)

- Assegnamo un quantitativo opportuno di RAM.

- Andiamo a specificare l'immagine di Linux che abbiamo estratto in precedenza.

Setup ambiente RISC-V (8)



- Fatto ciò andiamo ad avviare la macchina ed effettuiamo il login con le seguenti credenziali:
  - Utente: calcolatori
  - Password: calcolatori

Strumenti utili (1)

- Vediamo le cose essenziali della toolchain RISC-V:

- Compilatori gcc e g++ per RISCV:
  - *riscv64-unknown-elf-gcc*
  - *riscv64-unknown-elf-g++*

- Assemblatore e Linker:
  - *riscv64-unknown-elf-as*
  - *riscv64-unknown-elf-ld*

Strumenti utili (2)

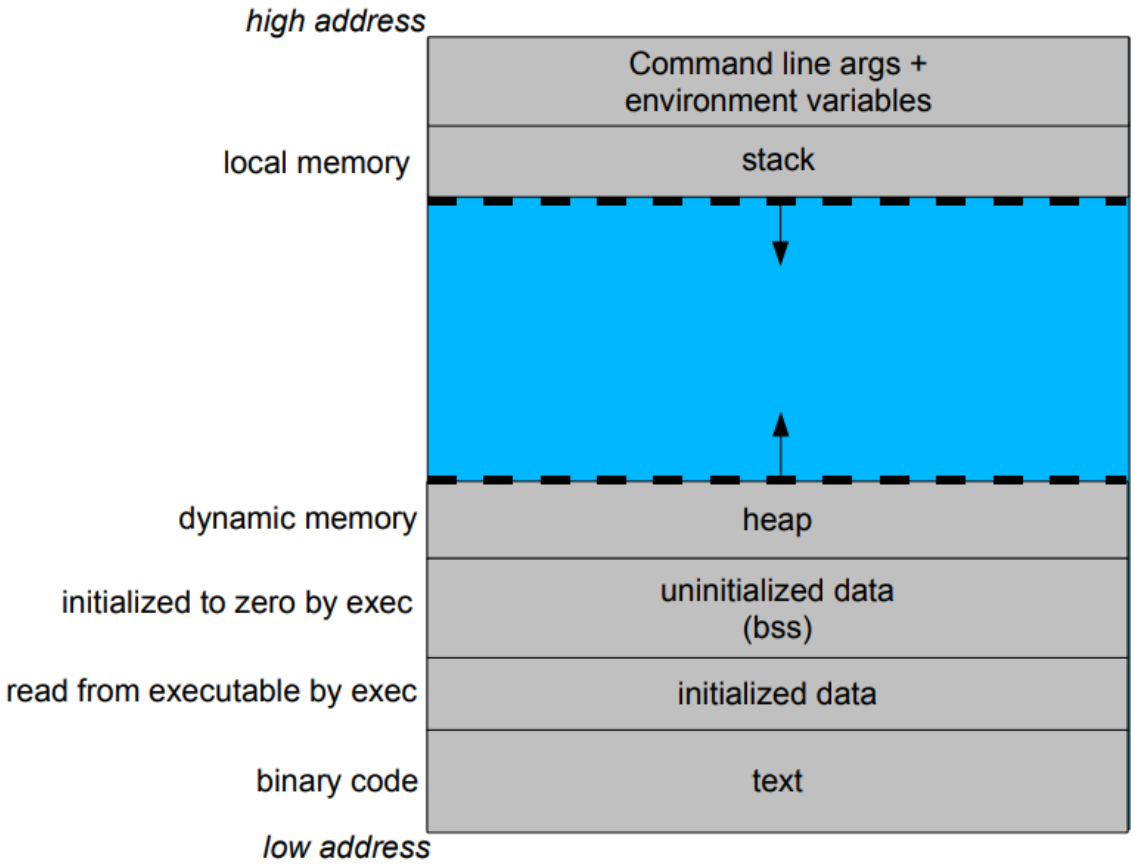
- Strumento che permette di ottenere informazioni da file .elf:
  - *riscv64-unknown-elf-readelf*
- Permette di ottenere informazioni da file oggetto e disassemblare eseguibili:
  - *riscv64-unknown-elf-objdump*

Strumenti utili (3)

- GNU debugger, permette di eseguire il programma passo passo e altre cose avanzate.
  - *riscv64-unknown-elf-gdb*
- Permette di ottenere informazioni sulle performance ottenute dai nostri programmi:
  - *riscv64-unknown-elf-gprof*

Memoria di un programma

- In Linux la memoria viene segmentata come in figura.



Registri

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Struttura di un programma Assembly (1)

```
.equ _SYS_EX, 93
.equ _SYS_WR, 64

.global _start

.section .bss
.section .data
.section .rodata
        msg: .string "Hello ASM\n"

.section .text

_start:

        li a0, 0
        la a1, msg
        li a2, 10

        li a7, _SYS_WR
        ecall

        li a7, _SYS_EX
        ecall
```

- Scriviamo lo stesso programma in Assembly

Struttura di un programma Assembly (2)

```
.equ _SYS_EX, 93
.equ _SYS_WR, 64

.global _start

.section .bss
.section .data
.section .rodata
        msg: .string "Hello ASM\n"

.section .text

_start:

        li a0, 0
        la a1, msg
        li a2, 10

        li a7, _SYS_WR
        ecall

        li a7, _SYS_EX
        ecall
```

- Definizione di costanti

Struttura di un programma Assembly (3)

```
.equ _SYS_EX, 93
.equ _SYS_WR, 64

.global _start

.section .bss
.section .data
.section .rodata
        msg: .string "Hello ASM\n"

.section .text

_start:

        li a0, 0
        la a1, msg
        li a2, 10

        li a7, _SYS_WR
        ecall

        li a7, _SYS_EX
        ecall
```

- Riferimento all'etichetta _start
- Punto di ingresso del programma

Struttura di un programma Assembly (4)

```
.equ _SYS_EX, 93
.equ _SYS_WR, 64

.global _start

.section .bss
.section .data
.section .rodata
        msg: .string "Hello ASM\n"

.section .text

_start:

        li a0, 0
        la a1, msg
        li a2, 10

        li a7, _SYS_WR
        ecall

        li a7, _SYS_EX
        ecall
```

- Segmento Block Started by Symbol (**bss**).
- Contiene variabili statiche o globali.
- Inizializzate a 0 o non inizializzate.

Struttura di un programma Assembly (5)

```
.equ _SYS_EX, 93
.equ _SYS_WR, 64

.global _start

.section .bss
.section .data
.section .rodata
        msg: .string "Hello ASM\n"

.section .text

_start:

        li a0, 0
        la a1, msg
        li a2, 10

        li a7, _SYS_WR
        ecall

        li a7, _SYS_EX
        ecall
```

- Segmenti **data** e Read Only Data (**rodata**).
- Rispettivamente per variabili in lettura/scrittura e sola lettura.

Struttura di un programma Assembly (6)

```
.equ _SYS_EX, 93
.equ _SYS_WR, 64

.global _start

.section .bss
.section .data
.section .rodata
        msg: .string "Hello ASM\n"

.section .text

_start:

        li a0, 0
        la a1, msg
        li a2, 10

        li a7, _SYS_WR
        ecall

        li a7, _SYS_EX
        ecall
```

- Segmento **text**.
- Contiene il codice del programma.

Logica del programma (1)

```
li a0, 0
la a1, msg
li a2, 10

li a7, _SYS_WR
ecall

li a7, _SYS_EX
ecall
```

- Utilizziamo due chiamate di sistema (syscall) tramite istruzione **ecall**.
- Una per scrivere (**sys_write**).
- Una per uscire (**sys_exit**).
- Il resto si occupa di preparare i registri per le syscalls.

Logica del programma (2)

```
li a0, 0
la a1, msg
li a2, 10

li a7, _SYS_WR
ecall

li a7, _SYS_EX
ecall
```

- Caricare in **a7** il codice numerico della syscall:
  - sys_write → 64
  - sys_exit → 93
- Per exit non serve altro. Mentre per write serve anche:
  - **a0** → dove scrivere: 0 stdout
  - **a1** → indirizzo di memoria
  - **a2** → quanti byte scrivere

Assemblamento

- Invochiamo l’assemblatore as e il linker:
  - *$ riscv64-unknown-elf-as -o helloWorld.o helloWorld.s*
  - *$ riscv64-unknown-elf-ld -o helloWorld helloWorld.o*
- Eseguiamo con:
  - *$ qemu-riscv64 helloWorld*

Disassemblamento

- Operazione contraria all'Assemblaggio, permette di ottenere un file assembly a partire da un file eseguibile.

```
calcolatori@calcolatori-VirtualBox:~/Scrivania/Esercizi$ riscv64-unknown-elf-objdump -d
hello

hello:     formato del file elf64-littleriscv


Disassemblamento della sezione .text:

0000000000010078 <_start>:
   10078:       04000893                li      a7,64
   1007c:       00000513                li      a0,0
   10080:       00000597                auipc   a1,0x0
   10084:       02158593                addi    a1,a1,33 # 100a1 <msg>
   10088:       00000617                auipc   a2,0x0
   1008c:       01860613                addi    a2,a2,24 # 100a0 <num>
   10090:       00060603                lb      a2,0(a2)
   10094:       00000073                ecall
   10098:       05d00893                li      a7,93
   1009c:       00000073                ecall
calcolatori@calcolatori-VirtualBox:~/Scrivania/Esercizi$
```

Informazioni Eseguibile

- Analizziamo l'eseguibile generato tramite *readelf* e notiamo il tipo di ISA *RISC-V.*

```
gian ~ riscv64-unknown-elf-readelf -h hello
Intestazione ELF:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF64
  Dati:                              complemento a 2, little endian
  Versione:                          1 (current)
  SO/ABI:                            UNIX - System V
  Versione ABI:                      0
  Tipo:                              EXEC (file eseguibile)
  Macchina:                          RISC-V
  Versione:                          0x1
  Indirizzo punto d'ingresso:        0x100b0
  Inizio intestazioni di programma   64 (byte nel file)
  Inizio intestazioni di sezione:    17912 (byte nel file)
  Flag:                              0x5, RVC, double-float ABI
  Dimensione di questa intestazione: 64 (byte)
  Dimens. intestazioni di programma:  56 (byte)
  Numero intestazioni di programma:  2
  Dimens. intestazioni di sezione:   64 (byte)
  Numero di intestazioni di sezione: 14
  Indice della tabella di stringhe delle intestazioni di sezione: 13
```

Direttive Assembly (1)

Directive	Arguments	Description
`.2byte`		16-bit comma separated words (unaligned)
`.4byte`		32-bit comma separated words (unaligned)
`.8byte`		64-bit comma separated words (unaligned)
`.half`		16-bit comma separated words (naturally aligned)
`.word`		32-bit comma separated words (naturally aligned)
`.dword`		64-bit comma separated words (naturally aligned)
`.byte`		8-bit comma separated words
`.asciz`	"string"	emit string (alias for .string)
`.string`	"string"	emit string

Direttive Assembly (2)

Directive	Arguments	Description
`.globl`	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
`.local`	symbol_name	emit symbol_name to symbol table (scope LOCAL)
`.equ`	name, value	constant definition

Directive	Arguments	Description
`.text`		emit .text section (if not present) and make current
`.data`		emit .data section (if not present) and make current
`.rodata`		emit .rodata section (if not present) and make current
`.bss`		emit .bss section (if not present) and make current
`.section`	[{.text,.data,.rodata,.bss}]	emit section (if not present, default .text) and make current

Debugging con GDB (1)

- **GNU debugger** (talvolta chiamato semplicemente GDB) è un programma libero sviluppato dal progetto GNU. È il debugger predefinito del sistema operativo GNU, gira su molte piattaforme (tra cui i sistemi Unix-like e Microsoft Windows) ed è capace di analizzare numerosi linguaggi di programmazione.
- Per RISC-V: *riscv-unknown-elf-gdb*

Debugging con GDB (2)

- Compiliamo il nostro codice specificando all'assemblatore di aggiungere informazioni di debug:
  - *$ riscv64-unknown-elf-as **-g** -o helloWorld.o helloWorld.s*

    $ riscv64-unknown-elf-ld -o helloWorld helloWorld.o

Debugging con GDB (3)

- Necessario “collegare” in qualche modo l’eseguibile sull’ambiente RISC-V emulato con il debugger gdb in esecuzione sulla macchina Host.
- Collegamento svolto tramite **socket TCP.**
- Apriamo un terminale e lanciamo **qemu** con:
  - *$ qemu-riscv64 -g 2223 helloWorld*

Debugging con GDB (4)

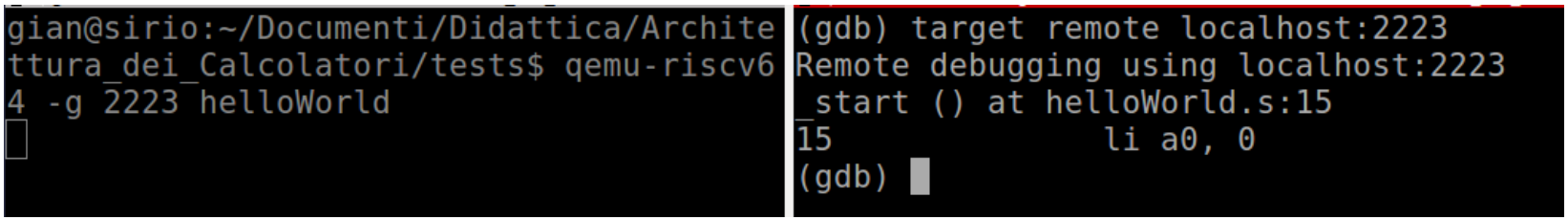
- Notiamo adesso che il programma è bloccato, In questo caso **è in ascolto** sulla porta 2223.
- Apriamo quindi un altro terminale e lanciamo il debugger con:
  - *$ riscv64-unknown-elf-gdb helloWorld*

Debugging con GDB (5)

- Adesso possiamo connetterci al programma in esecuzione (aperto sull'altro terminale) andando a digitare sulla shell di gdb:
  - *(gdb) target remote localhost:2223*

Debugging con GDB (6)

- A questo punto dovremmo vedere qualcosa di simile a quanto mostrato in figura:



- Notiamo quindi la prima istruzione dopo l'indirizzo *_start*.

Debugging con GDB (7)

- Ora possiamo eseguire il programma **istruzione per istruzione** andando a digitare **step** (oppure s) e premendo invio.
- Premendo successivamente invio, gdb eseguirà l'ultimo comando dato: in questo caso step.
- Tramite il comando **info registers** possiamo sempre monitorare lo stato dei registri.

Debugging con GDB (8)

```
21              ecall
(gdb) info registers
ra             0x0000000000000000       0
sp             0x00000040007ffdc0       274886294976
gp             0x0000000000000000       0
tp             0x0000000000000000       0
t0             0x0000000000000000       0
t1             0x0000000000000000       0
t2             0x0000000000000000       0
s0             0x0000000000000000       0
s1             0x0000000000000000       0
a0             0x0000000000000000       0
a1             0x0000000000010098       65688
a2             0x000000000000000a       10
a3             0x0000000000000000       0
a4             0x0000000000000000       0
a5             0x0000000000000000       0
a6             0x0000000000000000       0
a7             0x0000000000000040       64
s2             0x0000000000000000       0
s3             0x0000000000000000       0
s4             0x0000000000000000       0
s5             0x0000000000000000       0
s6             0x0000000000000000       0
s7             0x0000000000000000       0
s8             0x0000000000000000       0
s9             0x0000000000000000       0
s10            0x0000000000000000       0
s11            0x0000000000000000       0
t3             0x0000000000000000       0
t4             0x0000000000000000       0
t5             0x0000000000000000       0
t6             0x0000000000000000       0
pc             0x000000000001008c       65676
priv           [Invalid]
(gdb)
```

- Notiamo l'effetto dell'istruzione precedente.
- Scrittura di 64 in a7.

Debugging con GDB (9)

- Contenuto dello stack:
- *(gdb) x/100x $sp*

```
(gdb) x/100x $sp
0x40007ff978:   0xffffffff      0xeeeeeeee      0x00000001      0x00000000
0x40007ff988:   0x007ffc9c      0x00000040      0x00000000      0x00000000
0x40007ff998:   0x007ffcac      0x00000040      0x007ffcda      0x00000040
0x40007ff9a8:   0x007ffcf8      0x00000040      0x007ffd0b      0x00000040
0x40007ff9b8:   0x007ffd2b      0x00000040      0x007ffd7b      0x00000040
0x40007ff9c8:   0x007ffd9c      0x00000040      0x008003fb      0x00000040
0x40007ff9d8:   0x00800428      0x00000040      0x00800451      0x00000040
0x40007ff9e8:   0x00800470      0x00000040      0x008004a6      0x00000040
0x40007ff9f8:   0x008004b3      0x00000040      0x008004df      0x00000040
0x40007ffa08:   0x008004f1      0x00000040      0x008004f9      0x00000040
0x40007ffa18:   0x00800508      0x00000040      0x0080054d      0x00000040
0x40007ffa28:   0x00800581      0x00000040      0x008005a2      0x00000040
0x40007ffa38:   0x008005b4      0x00000040      0x008005c8      0x00000040
0x40007ffa48:   0x008005da      0x00000040      0x008005ea      0x00000040
0x40007ffa58:   0x008005fe      0x00000040      0x0080060f      0x00000040
0x40007ffa68:   0x0080061c      0x00000040      0x00800649      0x00000040
0x40007ffa78:   0x00800665      0x00000040      0x0080067d      0x00000040
0x40007ffa88:   0x00800698      0x00000040      0x008006d4      0x00000040
0x40007ffa98:   0x00800727      0x00000040      0x0080073c      0x00000040
0x40007ffaa8:   0x0080074f      0x00000040      0x00800762      0x00000040
0x40007ffab8:   0x00800777      0x00000040      0x00800787      0x00000040
0x40007ffac8:   0x008007b6      0x00000040      0x008007e9      0x00000040
0x40007ffad8:   0x0080080a      0x00000040      0x0080081c      0x00000040
0x40007ffae8:   0x00800833      0x00000040      0x0080083d      0x00000040
0x40007ffaf8:   0x0080084e      0x00000040      0x00800884      0x00000040
(gdb)
```

```
_start:

    addi sp, sp, -8

    li a0, 0xFFFFFFFF
    li a1, 0xEEEEEEEE

    sw a0, 0(sp)
    sw a1, 4(sp)

    addi sp, sp, +8

    li  a7, 93
    ecall
```

- *(gdb) x/nf addr:*
- *x/:* stampa contenuto stack
- *n:* blocchi da 4 bytes
- *f:* formato (hex, dec..)

Debugging con GDB (10)

- Contenuto dei segmenti:
- *(gdb) x/10x &A*
- Stampa le prime 10 word a partire dall'indirizzo A.

```
.global _start

.section .data
        A: .word 1, 6, 1, 1, 3
           .word 0, 1, 1, 4, 3
           .word 1, 5, 9, 1, 3
           .word 1, 1, 1, 7, 3
           .word 2, 2, 0, 1, 9

        b: .word 1, 2, 1, 1, 3
        n: .word 5

.section .bss
        C: .space 100

.section .text
```

```
(gdb) x/10x &A
0x112cc:        0x00000001      0x00000006      0x00000001      0x00000001
0x112dc:        0x00000003      0x00000000      0x00000001      0x00000001
0x112ec:        0x00000004      0x00000003
(gdb)
```

Debugging con GDB (11)

- Possiamo inoltre eseguire fino al termine del programma andando ad usare il comando **continue** (oppure c)**.**
- Tramite **quit** possiamo uscire dalla shell di gdb.
- A parte alle cose viste probabilmente non ci servirà altro per questo corso.

Debugging con GDB (12)

- Ulteriormente a quanto visto, è possibile anche specificare dei **breakpoints**, ovvero dei punti in cui l'esecuzione del programma si interrompe:
  - es: se dobbiamo ispezionare il contenuto della memoria/registri in un determinato punto del programma.
  - Non ancora supportato nella toolchain di RISC-V.
  - Dovremo farne a meno.

Esercizi per casa

- **Esercizio 01**: Copiare il seguente file sorgente ed inserire le direttive opportune per permettere l'assemblamento.

```
strcpy:
addi    sp,sp,-8        #   adjust stack for 1 dw
sd      x19,0(sp)       #   push x19
add     x19,x0,x0       #   i=0
L1:
add     x5,x19,x11      #   x5 = addr of y[i]
lbu     x6,0(x5)        #   x6 = y[i]
add     x7,x19,x10      #   x7 = addr of x[i]
sb      x6,0(x7)        #   x[i] = y[i]
beq     x6,x0,L2        #   if y[i] == 0 then exit
addi    x19,x19, 1      #   i = i + 1
jal     x0,L1           #   next iteration of loop
L2:
ld      x19,0(sp)       #   restore saved x19
addi    sp,sp,8         #   pop 1 dw from stack
jalr    x0,0(x1)        #   and return
```

Esercizi per casa

- Il programma precedente è un'implementazione assembly della seguente funzione C:

```
void strcpy (char x[], char y[]) {

        size_t i;
        i = 0;

        while ((x[i]=y[i])!='\0')
                i += 1;
}
```

- Inserire le stringhe x e y nel segmento dati.
- Aggiungere le direttive viste nell'esempio precedente.
- Aggiungere la chiamata alla funzione exit.

Esercizi per casa

- Inserire le stringhe x e y nel segmento dati:

```
.global _start

.section .data

    x: .string "rossi\0"
    y: .string "mario\0"
```

- Marcare l'inizio del segmento text:

```
.section .text
```

Esercizi per casa

- Inserire l'etichetta start, il caricamento degli indirizzi e la chiamata a funzione:

```
_start:

        la      x11, y
        la      x10, x

        jal x1, strcpy

        li      a7, 93
        ecall
```

Esercizi per casa

- **Esercizio 02**: Tramite il debugger GDB eseguire il programma passo-passo, ed al termine verificare se la stringa y è stata copiata correttamente.
- Leggere il contenuto della memoria a partire dall'indirizzo x col comando visto in classe:

  (gdb) x/10x &x

Esercizi per casa

- Memoria prima dell'esecuzione del programma:

```
(gdb) x/10x &x
0x11100:        0x73736f72      0x6d000069
```

- Memoria dopo l'esecuzione del programma:

```
(gdb) x/10x &x
0x11100:        0x6972616d      0x6d00006f
```

- Confrontare i risultati ottenuti con una tabella di caratteri ASCII e verificare che la copia sia avvenuta correttamente.

Esercizi per casa

Binario	Oct	Dec	Hex	Glifo	Binario	Oct	Dec	Hex	Glifo	Binario	Oct	Dec	Hex	Glifo
010 0000	040	32	20	Spazio	100 0000	100	64	40	@	110 0000	140	96	60	`
010 0001	041	33	21	!	100 0001	101	65	41	A	110 0001	141	97	61	a
010 0010	042	34	22	"	100 0010	102	66	42	B	110 0010	142	98	62	b
010 0011	043	35	23	#	100 0011	103	67	43	C	110 0011	143	99	63	c
010 0100	044	36	24	$	100 0100	104	68	44	D	110 0100	144	100	64	d
010 0101	045	37	25	%	100 0101	105	69	45	E	110 0101	145	101	65	e
010 0110	046	38	26	&	100 0110	106	70	46	F	110 0110	146	102	66	f
010 0111	047	39	27	'	100 0111	107	71	47	G	110 0111	147	103	67	g
010 1000	050	40	28	(	100 1000	110	72	48	H	110 1000	150	104	68	h
010 1001	051	41	29	)	100 1001	111	73	49	I	110 1001	151	105	69	i
010 1010	052	42	2A	*	100 1010	112	74	4A	J	110 1010	152	106	6A	j
010 1011	053	43	2B	+	100 1011	113	75	4B	K	110 1011	153	107	6B	k
010 1100	054	44	2C	,	100 1100	114	76	4C	L	110 1100	154	108	6C	l
010 1101	055	45	2D	-	100 1101	115	77	4D	M	110 1101	155	109	6D	m
010 1110	056	46	2E	.	100 1110	116	78	4E	N	110 1110	156	110	6E	n
010 1111	057	47	2F	/	100 1111	117	79	4F	O	110 1111	157	111	6F	o
011 0000	060	48	30	0	101 0000	120	80	50	P	111 0000	160	112	70	p
011 0001	061	49	31	1	101 0001	121	81	51	Q	111 0001	161	113	71	q
011 0010	062	50	32	2	101 0010	122	82	52	R	111 0010	162	114	72	r
011 0011	063	51	33	3	101 0011	123	83	53	S	111 0011	163	115	73	s
011 0100	064	52	34	4	101 0100	124	84	54	T	111 0100	164	116	74	t
011 0101	065	53	35	5	101 0101	125	85	55	U	111 0101	165	117	75	u
011 0110	066	54	36	6	101 0110	126	86	56	V	111 0110	166	118	76	v
011 0111	067	55	37	7	101 0111	127	87	57	W	111 0111	167	119	77	w
011 1000	070	56	38	8	101 1000	130	88	58	X	111 1000	170	120	78	x
011 1001	071	57	39	9	101 1001	131	89	59	Y	111 1001	171	121	79	y
011 1010	072	58	3A	:	101 1010	132	90	5A	Z	111 1010	172	122	7A	z
011 1011	073	59	3B	;	101 1011	133	91	5B	[	111 1011	173	123	7B	{
011 1100	074	60	3C	<	101 1100	134	92	5C	\	111 1100	174	124	7C	|
011 1101	075	61	3D	=	101 1101	135	93	5D	]	111 1101	175	125	7D	}
011 1110	076	62	3E	>	101 1110	136	94	5E	^	111 1110	176	126	7E	~
011 1111	077	63	3F	?	101 1111	137	95	5F	_					