

# Lab 5:

Stringhe - Procedure (parte 1)

# Esecuzione di una procedura

Per l'esecuzione di una procedura, un programma deve eseguire questi sei passi:

1. Mettere i **parametri** in un luogo accessibile alla procedura;
2. **Trasferire il controllo** alla procedura;
3. **Acquisire le risorse** necessarie per l'esecuzione della procedura;
4. **Eseguire** il compito richiesto;
5. Mettere il **risultato** in un luogo accessibile al programma chiamante;
6. **Restituire il controllo** al punto di origine, dato che la stessa procedura può essere chiamata in diversi punti di un programma.

# Parametri e Indirizzo di Ritorno

- registri `a0–a7` (`x10–x17`) sono 8 registri per i parametri, utilizzati cioè per passare valori alle funzioni o restituire valori al chiamante
- registro `ra` (`x1`) contiene l'indirizzo di ritorno

## jal e jalr: Passaggio di Controllo

- L'istruzione **jal** (**jump and link**) serve per la chiamata di funzioni: produce un salto a un indirizzo e salva l'indirizzo dell'istruzione successiva a quella del salto nel registro **ra** (indirizzo di ritorno, detto appunto link)

```
jal ra, ProcAddress # salta a ProcAddress e salva indirizzo di ritorno in ra  
[jal ProcAddress]
```

- Il ritorno da una procedura utilizza un salto indiretto, **jump and link register (jalr)**

```
jalr zero, 0(ra) # salta indietro all'indirizzo di ritorno presente in ra  
[jr ra] oppure [ret]
```

## `jal` e `jalr`: Passaggio di Controllo

Lo schema è quindi il seguente:

- la funzione chiamante mette i parametri in `a0–a7` e usa `jal x` per saltare alla funzione `x`
- la funzione chiamata svolge le proprie operazioni, inserisce i risultati negli stessi registri e restituisce il controllo al chiamante con l'istruzione `jr ra`

## Esempio (sum)

```
int sum(int a, int b){  
    return a+b;  
}
```

```
int main(int argc, char** argv) {  
    int a = 1;  
    int b = 2;  
    int result = 0;  
    result = sum(a,b);  
    printf("result: %d", result);  
    exit(0);  
}
```

## Esempio (sum)

```
int sum(int a, int b){  
    return a+b;  
}
```

```
int main(int argc, char** argv) {  
    int a = 1;  
    int b = 2;  
    int result = 0;  
    result = sum(a,b);  
    printf("result: %d", result);  
    exit(0);  
}
```

**\_start:**

```
li a0, 1 # a  
li a1, 2 # b  
li s1, 0 # result
```

```
jal sum  
add s1, a0, zero  
...
```

**sum:**

```
add a0, a0, a1  
jr ra
```

# Esempio (sum)

```
int sum(int a, int b){
```

**registri per passaggio parametri  
(FUNZIONE CHIAMANTE)**

```
int main(int argc, char** argv) {  
    int a = 1;  
    int b = 2;  
    int result = 0;  
    result = sum(a, b);  
}
```

**registri per passaggio parametri  
(FUNZIONE CHIAMATA)**

**\_start:**

```
li a0, 1 # a  
li a1, 2 # b  
li s1, 0 # result
```

```
jal sum  
add s1, a0, zero  
...
```

**sum:**

```
add a0, a0, a1  
jr ra
```



# Esempio (sum)

```
int sum(int a, int b){  
    return a+b;  
}
```

**#ra = MemAddress, jump sum**

```
    result = sum(a,b);  
    printf("result: %d", result);  
    exit(0);  
}
```

**\_start:**

```
li a0, 1 # a  
li a1, 2 # b  
li s1, 0 # result
```

```
jal sum  
add s1, a0, zero  
...
```

**sum:**

```
add a0, a0, a1  
jr ra
```

## Esempio (sum)

**domanda:** perché usiamo `jr ra`, e non semplicemente `jump`?

**risposta:** perché la funzione può essere chiamata da molti punti del programma, anche dall'interno di altre procedure. C'è quindi bisogno di **un meccanismo per tornare all'istruzione successiva alla chiamata**. Serve un meccanismo che tenga conto dell'indirizzo salvato sul registro `ra`.

```
_start:
    li a0, 1 # a
    li a1, 2 # b
    li s1, 0 # result

    jal sum
    add s1, a0, zero
    ...

sum:
    add a0, a0, a1
    jr ra
```

# Esempio (sum)

Address	Code	Basic	
0x00400000	0x00100513	addi x10,x0,1	6: li a0, 1 # a
0x00400004	0x00200593	addi x11,x0,2	7: li a1, 2 # b
0x00400008	0x00000493	addi x9,x0,0	8: li s1, 0 # result
0x0040000c	0x01c000ef	jal x1,0x0000001c	10: jal sum
0x00400010	0x000504b3	add x9,x10,x0	11: add s1, a0, zero
0x00400014	0x00900533	add x10,x0,x9	14: mv a0, s1
0x00400018	0x00100893	addi x17,x0,1	15: li a7, 1
0x0040001c	0x00000073	ecall	16: ecall
0x00400020	0x00a00893	addi x17,x0,10	19: li a7, 10
0x00400024	0x00000073	ecall	20: ecall
0x00400028	0x00b50533	add x10,x10,x11	24: add a0, a0, a1
0x0040002c	0x000008067	jalr x0,x1,0	25: jr ra

stato prima di eseguire jal sum

- pc vale 0x00000000000040000c

- ra vale 0x000000000000000000

**\_start:**

li a0, 1 # a

li a1, 2 # b

li s1, 0 # result

jal sum

add s1, a0, zero

...

**sum:**

add a0, a0, a1

jr ra

# Esempio (sum)

Address	Code	Basic	
0x00400000	0x00100513	addi x10,x0,1	6: li a0, 1 # a
0x00400004	0x00200593	addi x11,x0,2	7: li a1, 2 # b
0x00400008	0x00000493	addi x9,x0,0	8: li s1, 0 # result
0x0040000c	0x01c000ef	jal x1,0x0000001c	10: jal sum
0x00400010	0x000504b3	add x9,x10,x0	11: add s1, a0, zero
0x00400014	0x00900533	add x10,x0,x9	14: mv a0, s1
0x00400018	0x00100893	addi x17,x0,1	15: li a7, 1
0x0040001c	0x00000073	ecall	16: ecall
0x00400020	0x00a00893	addi x17,x0,10	19: li a7, 10
0x00400024	0x00000073	ecall	20: ecall
0x00400028	0x00b50533	add x10,x10,x11	24: add a0, a0, a1
0x0040002c	0x000008067	jalr x0,x1,0	25: jr ra

ra

pc

stato **dopo** aver eseguito jal sum

- pc vale 0x000000000000400028

- ra vale 0x000000000000400010

**\_start:**

li a0, 1 # a

li a1, 2 # b

li s1, 0 # result

jal sum

add s1, a0, zero

...

**sum:**

add a0, a0, a1

jr ra

# Esempio (sum)

Address	Code	Basic	
0x00400000	0x00100513	addi x10,x0,1	6: li a0, 1 # a
0x00400004	0x00200593	addi x11,x0,2	7: li a1, 2 # b
0x00400008	0x00000493	addi x9,x0,0	8: li s1, 0 # result
0x0040000c	0x01c000ef	jal x1,0x0000001c	10: jal sum
0x00400010	0x000504b3	add x9,x10,x0	11: add s1, a0, zero
0x00400014	0x00900533	add x10,x0,x9	14: mv a0, s1
0x00400018	0x00100893	addi x17,x0,1	15: li a7, 1
0x0040001c	0x00000073	ecall	16: ecall
0x00400020	0x00a00893	addi x17,x0,10	19: li a7, 10
0x00400024	0x00000073	ecall	20: ecall
0x00400028	0x00b50533	add x10,x10,x11	24: add a0, a0, a1
0x0040002c	0x000008067	jalr x0,x1,0	25: jr ra

ra

pc

stato **dopo** aver eseguito jr ra

- pc vale 0x000000000000400010

- ra vale 0x000000000000400010

**\_start:**

li a0, 1 # a

li a1, 2 # b

li s1, 0 # result

jal sum

add s1, a0, zero

...

**sum:**

add a0, a0, a1

jr ra

Stringhe

# Codifica ASCII

- *American Standard Code for Information Interchange*
- Utilizza 8 bit (1 byte) per rappresentare i caratteri
- `load byte unsigned (lbu)` prende un byte dalla memoria mettendolo negli 8 bit di un registro, collocati più a destra
- `store byte (sb)` prende il byte corrispondente agli 8 bit di un registro, collocati più a destra, e lo salva in memoria

```
lbu x12, 0(x10) // Leggi un byte dall'indirizzo sorgente  
sb x12, 0(x11) // Scrivi il byte all'indirizzo di destinazione
```

# Codifica ASCII

Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere
32	Spazio	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL



# Codifica ASCII

Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere	Valore ASCII	Carattere
32	Spazio	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	

**Il linguaggio C termina le stringhe con un byte che contiene il valore 0 (carattere "null" in ASCII, non mostrato nella tabella)**

# Esercizio 0 – charAt

Usando il linguaggio assembly del RISC-V, scrivere una funzione **charAt** che riceva:

- sul registro a0, l'**indirizzo** in memoria di una stringa (array di byte)
- sul registro a1, un numero intero (n)

**charAt** ritorna il carattere nella posizione n della stringa str.  
Il seguente codice in C realizza charAt (convertilo in RISC-V):

```
char charAt(char *str, int n) {  
    return str[n];  
}
```

# Esercizio 1 – strlen (String Length)

Scrivere una procedura RISC-V per calcolare la lunghezza di una stringa di caratteri in C, escluso il carattere terminatore. Le stringhe di caratteri in C sono memorizzate come un array di byte in memoria, dove il byte ‘\0’ (0x00) rappresenta la fine della stringa.

```
unsigned long strlen(char *str) {  
    unsigned long i;  
    for (i = 0; str[i] != '\0'; i++);  
    return i;  
}
```

```
.globl _start  
.data  
    src: .string "This is the source string."
```

## Esercizio 2 – `digit`

Scrivere una funzione RISC-V **`digit`** che verifichi se un byte passato come parametro nel registro `a0` rappresenta un carattere cifra (0-9) nella codifica ASCII. Verificare vuol dire: restituire 1 (su `a0`) se la condizione è vera, 0 altrimenti.

## Esercizio 3 – strcmp

Scrivere una procedura RISC-V `strcmp` per confrontare due stringhe di caratteri. `strcmp(str1, str2)` restituisce 0 se `str1` è uguale a `str2`, 1 nel caso contrario.

risultato atteso, `a0 = 1`

```
.globl _start
.data
    str1: .string "first"
    str2: .string "second"
```

## Esercizio 4 – strchridx

Scrivere una procedura RISC-V `strchridx(str, c)` per restituire l'indice della prima occorrenza di **c** in **str**.

**strchridx(str, c)** restituisce -1 se c non è presente in str.

```
long long strchridx(char *str, char c) {
    long long i = 0;
    while (str[i] != '\0' and str[i] != c) {
        i++;
    }
    if (str[i] == '\0') {
        return -1;
    } else {
        return i;
    }
}
```

# Esercizio 5 – hash

Usando il linguaggio assembly del RISC-V, scrivere una funzione **hash** che riceva sul registro **a0** l'indirizzo in memoria di una stringa (array di byte).

La funzione deve calcolare il valore hash della stringa e ritornarlo sul registro a0. Il valore hash deve essere calcolato come segue:

```
long hash(char *str) {
    long hash = 5381;
    int i = 0;

    while (str[i] != '\0') {
        hash = ((hash << 5) + hash) + str[i];
        i++;
    }
    return hash;
}
```