Università degli Studi di Milano
Dipartimento di Informatica "Giovanni Degli Antoni"
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II
Laboratorio

1

I "nomi" dei 32 registri utente

- I registri utente sono denotati come sappiamo con la sintassi `$0` a `$31`
  - e sono usabili intercambiabilmente: uno vale l'altro
- Tuttavia, il loro uso è soggetto ad alcune utili convenzioni d'uso
  - L'adesione a queste convenzioni agevola l'interazione fra programmi scritti da persone (o complilatori) diversi
  - Ad esempio: «il registro $8 è usato per valori temporanei»
  - Nel corso di queste lezioni, impareremo molti di questi usi standard
- Per agevolare l'uso di queste convenzioni, l'assembler consente di riferirsi ad un registro non solo con il suo numero (per es: `$12`) ma anche con il suo «**nome**» (per es: `$ra`),
  - Ogni registro ha un nome standard, di due lettere (quattro, in un caso)
  - E' un sinonimo che richiama all'uso standard di quel registro
- Nel corso di queste lezioni, impareremo gli usi standard di molti registri, e i corrispondenti «nomi»
  - Ne abbiamo già visto uno: il registro `$0` che l'unico speciale perché non può contenere altro che il valore 0, ha nome `$zero`

2

Nomi dei registri

Registro:	$0	$1	$2	$3	$4	$5	$6	$7
Sinonimo:	$zero	$at	$v0	$v1	$a0	$a1	$a2	$a3
Registro:	$8	$9	$10	$11	$12	$13	$14	$15
Sinonimo:	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
Registro:	$16	$17	$18	$19	$20	$21	$22	$23
Sinonimo:	$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
Registro:	$24	$25	$26	$27	$28	$29	$30	$31
Sinonimo:	$t8	$t9	$k0	$k1	$gp	$sp	$s8	$ra

3

Registri per dati comuni: $t e $s

- I registri denominati
  `$t` (temporanei) ed
  `$s` (save)
  non hanno un uso specifico
- Sono usati comunemente usati
  per i valori su cui lavorare con i comuni comandi
- Vedremo la differenza più di preciso
  quando studieremo le funzioni

4

Moltiplicazioni e divisioni fra interi in MIPS

5

Istruzione mult

- Il risultato di un prodotto fra word a 32 bit è (potenzialmente) un numero a 64 bit.
- In MIPS l'operazione di moltiplicazione `mult $t3 $t4` produce il quindi risultato quindi due registri speciali:
  - «HI» i 32 bits più significativi
  - «LO» i 32 bits meno significativi
  - Questo fa eccezione alla regola «il primo registro è il risultato dell'operazione» (che vale per quasi tutte le altre operazioni logiche e matematiche)
- I due registri HI e LO *non* sono parte dei 32 registri utente.
- Vengono scritti solo dalle operazioni, e possono essere acceduti copiando il loro valore corrente in un registro a scelta, per es:

```
mfhi $t1  #copia in $t1 il registro HI
mflo $t1  #copia in $t2 il registro LO
```

6

Istruzione div

- Il risultato di un divisione fra interi produce contemporaneamente due risultati: il quoziente (intero) e il resto
  - Per es: 44 diviso 6 = 7 con il resto di 2
- In MIPS l'operazione di divisione `div $t3 $t4` produce il quindi risultato quindi due registri speciali:
  - «HI» il resto
  - «LO» il quoziente
  - Anche questo fa eccezione alla regola «il primo registro è il risultato dell'operazione» (che vale per quasi tutte le altre operazioni logiche e matematiche)
- I due registri HI e LO vengono acceduti con le operazioni viste sopra

```
mfhi $t1  #copia in $t1 il registro HI
mflo $t1  #copia in $t2 il registro LO
```

7

Pseudo-istruzione mul

Quando il risultato del prodotto è minore del massimo valore esprimibile, e basterebbe usare solo LO.

L'assembly mips ci mette a disposizione la MUL:
(per prodotti che non fanno troppi miliardi)

```
mul $t1 $t3 $t4   # t1 = t3 x t4

tradotta in:  mult $t3 $t4
              mflo $t1
```

8

Pseudo-istruzioni div e rem

Quando ci serve solo il quoziente o solo il resto, possiamo usare …

pseudo-istruzione DIV: (divisione intera)

```
div $s1 $t3 $t4   # s1 = t3 / t4
```

tradotta in:

```
div $t3 $t4
mflo $s1
```

Nota che questa pseudo-istruzine div ha lo stesso nome dell'istruzione, e viene distinta solo in base al numero di parametri che la seguono (2 o 3).
(una strategia che non è consistente con il resto della sintassi del MIPS)

pseudo-istruzione REM: (reminder, «resto», detto anche modulo)

```
rem  $s2 $t3 t4   # s1 = t3 % t4
```

tradotta in:

```
div $t3 $t4
mfhi $s1
```

9

Memoria dati (statica) in MIPS

10

Segmento **data**

- Come sappiamo, il segmento `.data` contiene i dati del programma *statici*
  (cioè, che rimangono allocati dall'inizio alla fine dell'esecuzione del programma)

11

Etichette (label)

- Un'etichetta (label) è un "segnalibro" fisso posto in un indirizzo della memoria statica (data, o, come vedremo, text) di cui tiene traccia l'assembler

Definizione di una label (nota il due punti finale) Stiamo chiedendo all'assembler di "segnarsi" l'indirizzo di memoria al quale stiamo per mettere il dato successivo (5000)

```
.data
qui:
5000

.text
la $t1 qui
lw $t0 ($t1)
```

Pseudo istruzione: load address
Viene tradotta con il comando MIPS che assegna a $t1 l'indirizzo corrispondete alla label "qui" (nota: usata senza i due punti)

Istruzione: Load Word.
Carica in $t0 il word all'indirizzo RAM in $t1, quindi il valore 5000

12

Specificare i dati in altri formati

- Posso cambiare il formato in cui specifico i dati con apposite «direttive»
- Ognuna di queste direttive cambia il modo in cui l'assembler interpreta i dati che scrivo di seguito
  - valgono cioè fino a contrordine
- Nota: questo è indipendente dalle etichette
  - ricordare la sitassi:
    l'etichetta termina con due-punti
    la direttiva inizia con punto

13

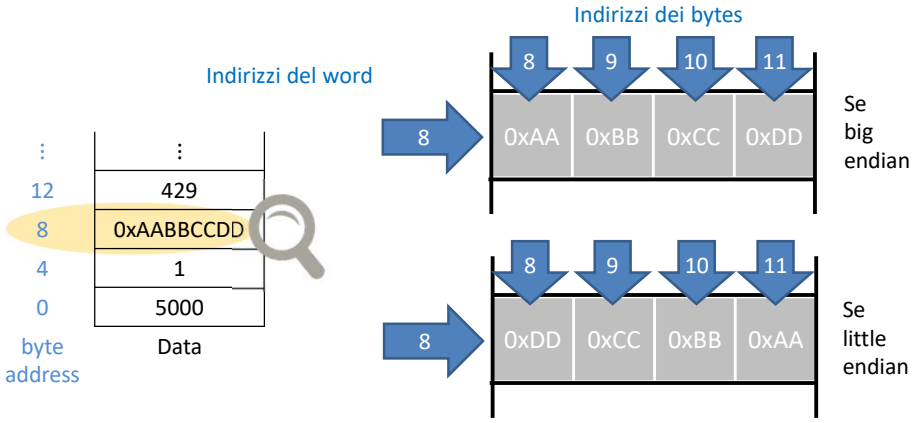
Direttive per la specifica dati

`.word`	Ogni numero che segue = un word (4 bytes) (è il default, se non si speficica alcuna direttiva)
`.half`	Ogni numero che segue = un half-word (2 bytes)
`.byte`	Ogni numero che segue = un byte
`.ascii`	Ogni *stringa* che segue, messa fra "virgolette" = 1 byte per lettera – il codice ASCII di quella lettera
`.asciiz`	Come sopra, più un ultimo ulteriore byte 0x00 per terminare la stringa
`.space n`	Lascia qui *n* byte di spazio (salta *n* byte prima di inserire il prossimo dato)
`.align n`	Lascia qui un certo numero di byte per rendere la prossima locazione di memoria divisible per 2^n

14

Endianness

- L'indirizzo di una parola di memoria è anche l'indirizzo di uno dei 4 byte che compongono quella parola

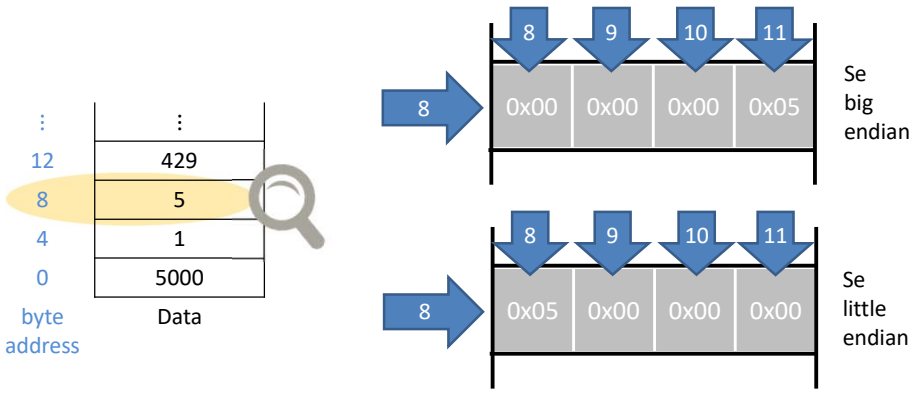


- Ma, tra i 4, **quale?** Dipende dall'ordine dei byte: la «**endianness**» dell'architettura

15

Endianness

- L'indirizzo di una parola di memoria è anche l'indirizzo di uno dei 4 byte che compongono quella parola



- Ma, tra i 4, **quale?** Dipende dall'ordine dei byte: la «**endianness**» dell'architettura

16

Direttiva .byte

- La direttiva byte specifica che i dati che seguono occupano un byte ciascuno

```
.byte 0xAA 0xBB 0xCC 0xDD
```

equivalente a:

```
.word 0xDDCCBBAA
```

(se la macchina è big endian)

La famiglia di architetture x86 è Little Endian (Intel Core i7, AMD Phenom II, FX, …).

17

Direttiva .half

- La direttiva half (half-word) specifica che i dati che seguono occupano due byte ciascuno

```
.byte 0xAABB 0xCCDD
```

equivalente a:

```
.word 0xDDCCBBAA
```

(se la macchina è big endian)

18

Dati numerici

- Posso esprimere i valori anche in esadecimale:

```
.word 0xABCD0000
.half 0xAB13  0xAC01
.byte 0xAF
```

- Posso esprimere i valori come numeri negativi (vengono interpretati in complemento a 2)

```
.word -1
```
equivalente a
```
.word 0xFFFFFFFF
```
equivalente a
```
.word 4294967295
```

```
.byte -1
```
equivalente a
```
.byte 0xFF
```
equivalente a
```
.byte 255
```

19

Direttive .ascii

```
.ascii "Derp"
```

equivalente a:

```
.byte 0x44 0x65 0x62 0x70
```

Ogni lettera della striga che segue la direttiva .ascii viene tradotta in un codice ascii di 1 byte

20

Direttive .ascii e .asciiz

```
.asciiz "Derp"
```

equivalente a:

```
.byte 0x44 0x65 0x62 0x70 0x00
```

21

Tabella ascii

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

22

Osservazione su direttive e etichette

- Combinando i meccanismi di direttive e etichette, ottengo una sintassi che somiglia superficialmente a quella di una *dichiarazione delle variabili* in un linguaggio ad alto livello (C, Java, Go…)
  - La direttiva somiglia al TIPO della variabile
  - L'etichetta somiglia all'IDENTIFICATORE della variabile
  - Il valore del dato somiglia all'INIZIALIZZAZIONE

```
peso: .word 65000
```
assembly MIPS

```
peso := int 65000 ;
```
Go

```
int peso = 65000 ;
```
C o Java

23

Osservazione su direttive e etichette

- La somiglianza è solo superficiale
- A differenza delle variabili in un linguaggio ad alto livello, i nostri dati in memoria RAM non sono associati ad alcun tipo
- Sta al programmatore (o al compilatore) MIPS usarli in modo consistente con loro semantica / tipo
- Per es
  - nulla distingue in indirizzo di memoria a cui memorizzo un array di numeri da quello in cui memorizzo in numero solo
  - posso definire 4 byte in successione, e poi usarli come un singolo word, oppure come una stringa di 4 lettere

24

Vettori (array)

- Un Array non è altro che una sequenza di *n* dati in RAM dello stesso formato (e dimensione) memorizzati in sequenza, (cioè ad indirizzi successivi)
- Per es, un vettore di interi = una successione di words (agli indirizzi n, n+4, n+8…)
- Ad alto livello, i vettori si accedono con una sintassi del tipo: `v[3]` (il quarto elemento del vettore, quello preceduto da 3 elementi)

- A basso livello, possiamo calcolare l'indirizzo di ogni elemento del vettore:
  - Se `b` è registro da cui parte un vettore `v`, cioè anche l'indirizzo del suo primo elemento `v[0];`
  - Allora l'elemento i-esimo ha indirizzo `b + 4*i`.

25

Vettori (array) Esempio

```
.data
voti: .word 28 21 30 27 24

.text
la $s0 voti  # s0 = l'indirizzo dell'array voti
lw $t5 ($s1) # copio voti[0] (cioè 28) in t5
lw $t5 12($s1) # copio voti[3] (cioè 27) in t5
```

26

Direttiva .space

- Con la direttiva **`.space n`**
  chiediamo all'assembler di lasciare n byte di spazio vuoti in memoria in RAM (prima di scrivere il dato successivo)
- Non è previsto che venga cancellata
- Se il programma **legge** da questo spazio prima di scriverci, può trovare qualsiasi valore
  - per es, il valore lasciato in quell'area di memoria da un programma eseguito in precedenza
  - Può dipendere da: il sistema operativo, quale programma è stato eseguito in precedenza
  - E' detta «Memoria sporca»
- Un programma corretto **scrive** nella memoria RAM riservata in questo modo, prima di leggerla

27

Comandi per letture e scritture in RAM

Istruzione	*Commento*	
sw $3, 500($4)	Store word	da registro a mem (store)
sh $3, 502($2)	Store half	
sb $2, 41($3)	Store byte	
lw $1, 30($2)	Load word	da mem a registro (load)
lh $1, 40($3)	Load halfword	
lhu $1, 40($3)	Load halfword unsigned	
lb $1, 40($3)	Load byte	
lbu $1, 40($3)	Load byte unsigned	
lui $1, 40($3)	Load Upper Immediate (16 bits shifted left by 16)	

35

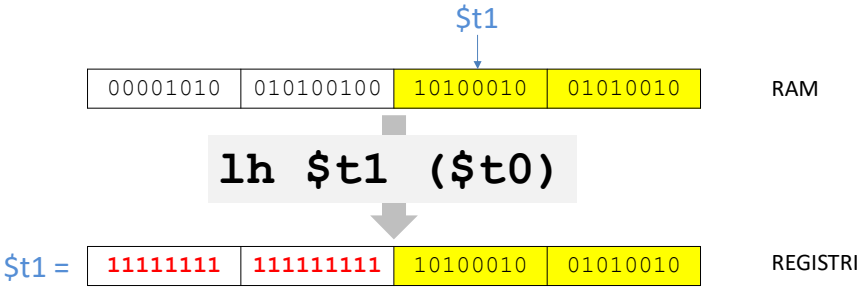
MIPS: Letture e scritture RAM

- Store: da registri --> a RAM
- Load: da RAM --> a registri
- Primo operando:
  il registro da / a quale operare
- Secondo operando, scritto fra parentesi ($14):
  il registro che *contiene l'indirizzo RAM*
  - ricordare: in MIPS 1 registro = 1 indirizzo memoria
- Numero prima della parentesi:
  «offset» Viene aggiunto all'indirizzo
  - è opzionale: se non viene specificato, allora vale 0
  - a cosa serve? (vedi dopo)

36

Load and Store di half words

- Quando si caricano sottopezzi di parola (**half words** o bytes)
  MIPS estende in segno



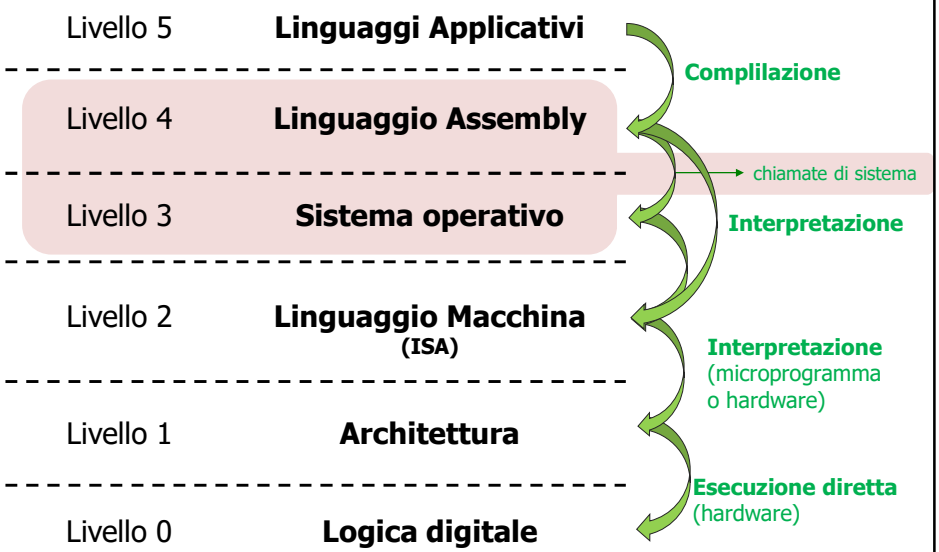
- La versione UNSIGNED **`lhu`** estende invece con zeri

37

Le chiamate di sistema in MIPS

38

Livelli di astrazione in un elaboratore



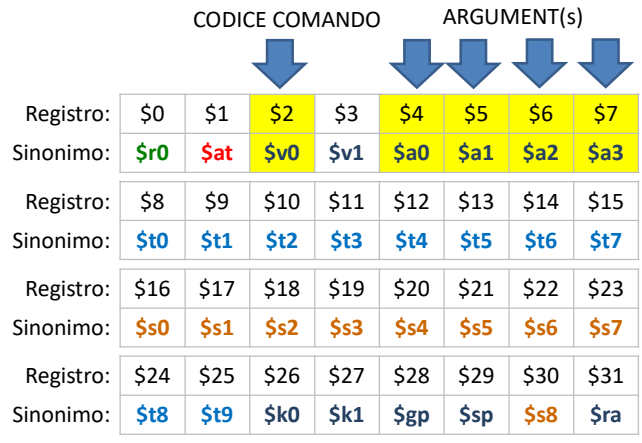
39

System Calls

- **System call**: permette di utilizzare **servizi** la cui esecuzione è a carico del sistema operativo: tipicamente operazioni di input/output e di interfacciamento con le periferiche (attraverso i drivers)
- Ogni servizio è associato ad un codice numerico univoco (un intero **K**)
  - Su MARS: la lista è disponibile da help -> system calls su Mar
- Come si utilizza una system call che ha una dato **CODICE** (numerico)?
  1. Caricare **CODICE** nel registro `$v0`;
  2. caricare gli argomenti (se necessari) nei registri `$a0, $a1, $a2, $a3`
  3. eseguire l'istruzione `syscall`
  4. leggere eventuali valori di ritorno nei registri `$v0` (e, `$v1`).
- EFFETTI COLLATERALI: dopo la chiamata di una system call, qualsiasi registro di classe `$ti` (registri «temporanei») può essere stato modificato ! (dal funzionamento interno della funzione chiamata)

40

Prima della chiamata di una System Call



43

System Calls «Canoniche»

Syscall	Codice	Argomenti	Valore di ritorno	Descrizione
print_int	1	intero da stampare in $a0	nessuno	Stampa l'intero passato in $a0
print_float	2	float da stampare in $f12	nessuno	Stampa il float passato in $f12
print_double	3	double da stampare in $f12	nessuno	Stampa il double passato in $f12
print_string	4	Indirizzo della stringa da stampare in $a0	nessuno	Stampa la stringa che sta all'indirizzo passato in $a0
read_int	5	nessuno	Intero letto in $v0	Legge un intero in input e lo scrive in $v0
read_float	6	nesuno	Float letto in $f0	Legge un float in input e lo scrive in $f0
read_double	7	nessuno	Double letto in $f0	Legge un double in input e lo scrive in $f0
read_string	8	Indirizzo nel segmento dati a cui salvare la stringa in $a0 e lunghezza in byte in $a1	nessuno	Legge una stringa di lunghezza specificata in $a1 e la scrive nel segment dati all'indirizzo specificato in $a0
sbrk	9	Numero di byte da allocare in $a0	Indirizzo del primo dei byte allocati in $v0	Accresce il segmento dati allocanto un numero di byte specificato in $a0, restituisce in $v0 l'indirizzo del primo di questi nuovi byte
exit	10	nessuno	nessuno	Termina l'esecuzione

46

System Calls «Apocrife» (fornite dalla macchina virtuale MARS)

Syscall	Codice	Argomenti	Valore di ritorno	Descrizione
Time	30	nessuno	32 bit meno significativi del system time in $a0, 32 bit più significativi del system time in $a1	Il system time è rappresentato nel formato Unix Epoch time, cioè il numero di millisecondi trascorsi dal 1 Gennaio 1970
random int	41	Id del generatore pseudo-random in $a0	Prossimo numero pseudo random in $a0	Ad ogni chiamata restituisce un numero intero in una sequenza pseudo-random
random in range	42	Id del generatore pseudo-random in $a0, massimo intero generabile in $a1	Prossimo numero pseudo random in $a0	Ad ogni chiamata restituisce un numero intero in una sequenza pseudo-random, ogni numero sarà compreso tra 0 e il massimo passato in $a1
MessageDialog	55	Indirizzo della stringa da stampare in $a0, intero corrispondente al tipo di messaggio in $a1		Mostra una finestra di dialogo con un messaggio dato dalla stringa passata in $a0. Viene anche mostrata una icona che dipende dal tipo di messaggio passato in $a1: errore (0), info, (1), warning (2), domanda(3)
InputDialogInt	51	Indirizzo della stringa da stampare in $a0	Intero letto in $a0, stato in $a1	

47

Esempio

```
        .data
msg1: .asciiz "Hello world!"
msg2: .asciiz "Inserisci un intero"

        .text
        .globl main
main:

        li $v0 4,
        la $a0, msg1
        syscall

        li $v0 55
        la $a0 msg1
        li $a1 1
        syscall

        li $v0, 51
        la $a0, msg2
        syscall

        li $v0 10
        syscall
```

Stampiamo una stringa nello standard output (la console)

Stampiamo una stringa in una finestra di diaologo

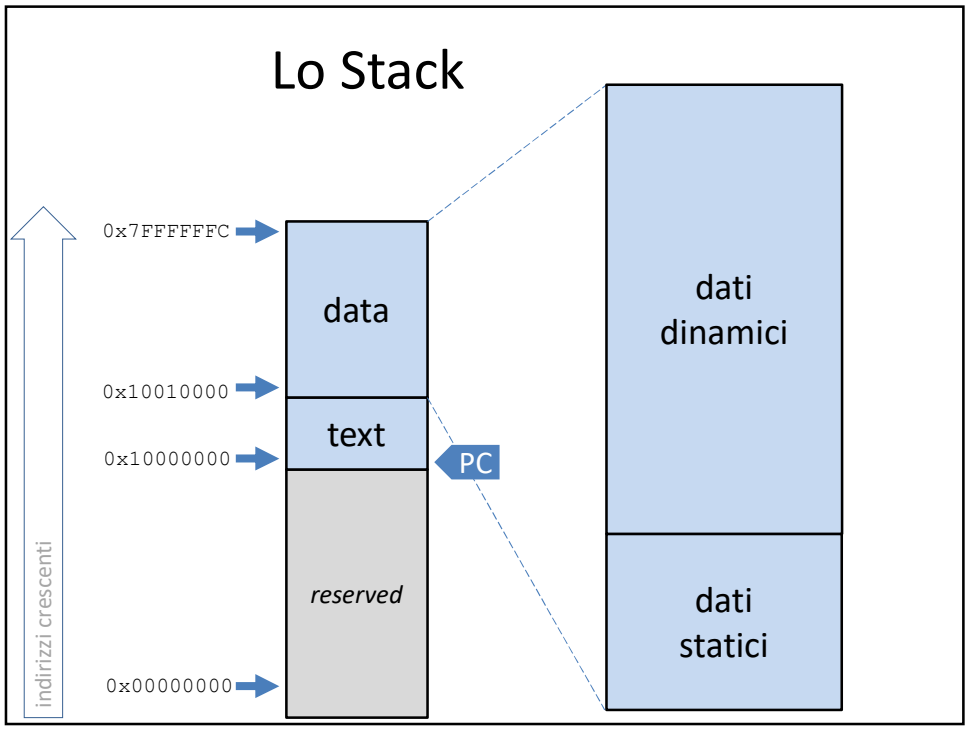
Leggiamo un intero in input con una finestra di dialogo

Exit

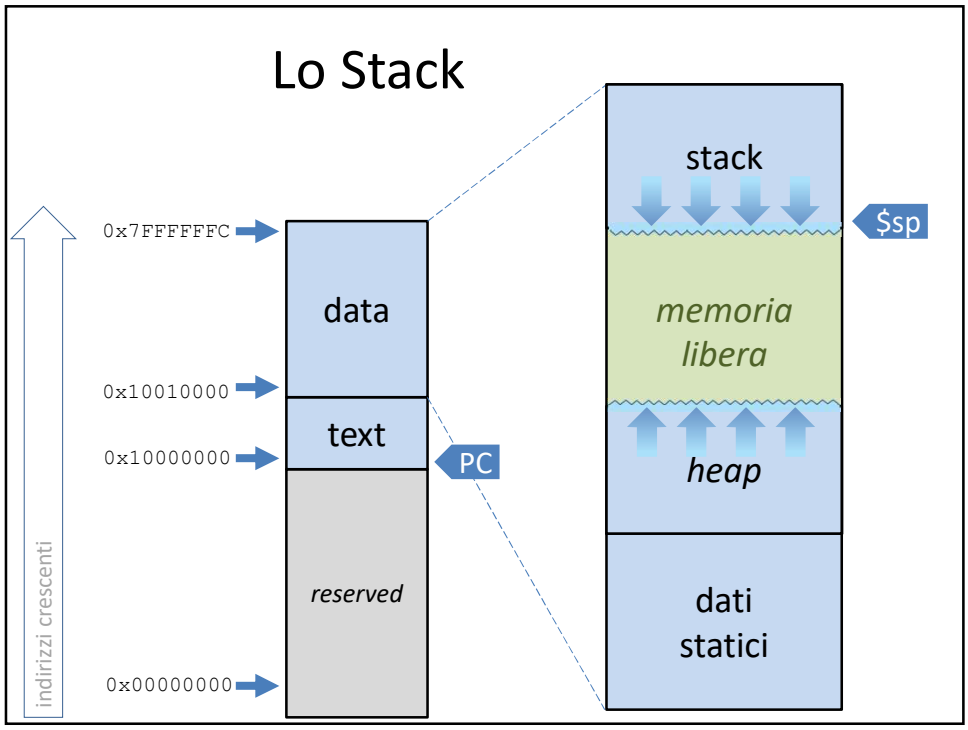
48

Uso dello Stack

49


Lo Stack
0x7FFFFFFC
0x10010000
0x10000000
0x00000000
data
text
reserved
PC
dati
dinamici
dati
statici
indirizzi crescenti

50


Lo Stack
0x7FFFFFFC
0x10010000
0x10000000
0x00000000
data
text
reserved
PC
stack
$sp
memoria
libera
heap
dati
statici
indirizzi crescenti

51

Aree di memoria

- **Area riservata**: contiene il codice per il kernel del Sistema operativo. Non usare.
- **Area text**: contiene il programma, cioè le istruzioni codificate nei segmenti .text del nostro programma.
- **Area dati statici**:
  - Non cambia di dimensione durante l'esecuzione
  - Ad alto livello: contiene le variabili globali e statiche (rimangono allocate durante tutta l'esecuzione)
  - In MIPS: contiene i dati che scriviamo nei segmenti .data
  - La sua dimensione cambia da programma a programma, a seconda di quanti dati usiamo in .data

52

Aree di memoria: area Stack

- Comincia dal "tetto" in alto della memoria
- Si espande (verso il basso) e contrae (verso l'alto) *durante l'esecuzione del programma* (cioè: dinamicamente)
- Ad alto livello: usata per le variabili locali
- In MIPS: usiamo un apposito registro, lo $sp (il numero $29) per tenere traccia di dove è arrivato lo stack
  - Lo $sp contiene *l'indirizzo dell'ultimo word usato dallo stack*
  - E' inizializzato dal sistema, ma sta al nostro programma tenerlo aggiornato!

53

Aree di memoria: **area Heap**

- Comincia da dove finisce il segmento di dati statici
- Si espande (verso l'alto) e/o contrae (verso il basso) *durante l'esecuzione del programma*
  - (cioè: dinamicamente)
- Ad alto livello: usata per le variabili allocate dinamicamente
  - Si espande quando vengono allocate con comandi come "new", "alloc", "malloc"…
  - Si contrae quando vengono deallocate con comandi come "free" "delete", o dal garbage collector
- Vedremo il suo uso in MIPS un'altra lezione

54

Aree di memoria: memoria libera

- E la zona di memoria fra Stack e Heap, che può essere occupata da uno qualsiasi dei due
  - Quello dei due che si espande per primo
  - Design furbo! Molto meglio che non riservare una zona di memoria fissa per lo Stack e una per lo Heap
- Se si esaurisce, causa un errore (a tempo di esecuzione):
  - uno "stack overflow" se lo stack scende troppo e invade lo heap
  - uno "heap overflow" se lo heap sale troppo e invade lo stack

55

Stack Pointer

- Il registro `$sp` (stack pointer) contiene sempre l'indirizzo della parola che sta in cima allo stack. Questa parola è l'ultima ad essere stata inserita nello stack e sarà la prima ad essere rimossa.

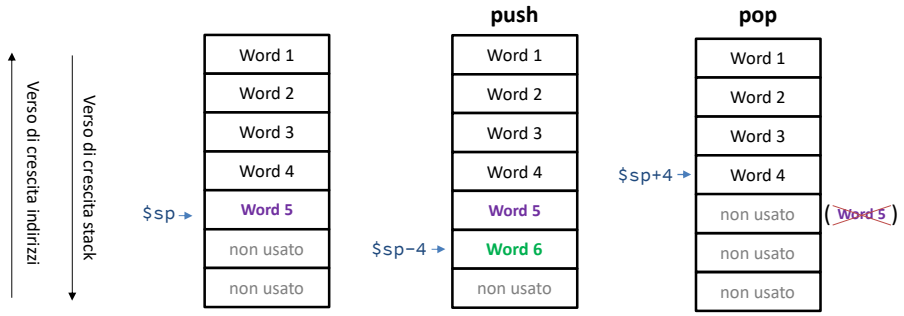
Regole per l'utilizzo dello stack

Per aggiungere un **dato** (push):

```
addi $sp $sp -4
sw $reg 0($sp)
```

Per consumare un **dato** (pop):

```
lw $reg 0($sp)
addi $sp $sp, 4
```



56

Sinonimi dei registri MIPS

Registro:	$0	$1	$2	$3	$4	$5	$6	$7
Sinonimo:	$r0	$at	$v0	$v1	$a0	$a1	$a2	$a3
Registro:	$8	$9	$10	$11	$12	$13	$14	$15
Sinonimo:	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
Registro:	$16	$17	$18	$19	$20	$21	$22	$23
Sinonimo:	$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
Registro:	$24	$25	$26	$27	$28	$29	$30	$31
Sinonimo:	$t8	$t9	$k0	$k1	$gp	$sp	$s8	$ra

STACK POINTER

58

Register Spilling

- Quando serve salvare dati sullo stack?
  Una prima risposta è «Per fare **spilling** di registri»
- In generale, un programma potrebbe lavorare su di un numero di variabili maggiore rispetto al numero di registri disponibili. Non è possibile avere tutti i dati nel banco registri allo stesso momento.
- Una possibile soluzione è questa: tengo nei registri i dati di cui ho maggior bisogno (ad esempio quelli che devo usare più volte o più spesso) mentre i dati di cui non ho bisogno urgente vengono spostati *temporaneamente* in memoria. (con una push sullo stack)
- Trasferire variabili poco utilizzate da registri a memoria è detto **register spilling**.
- L'area di memoria di solito utilizzata per questa operazione è lo stack.

59

Register Spilling

Esempio

- Supponiamo di poter utilizzare solo i registri $t0 e $t1
- Dobbiamo calcolare il prodotto di due variabili che stanno nel segmento dati e i cui indirizzi sono identificati dalle label x e y

```
        .data
x:      .word 3
y:      .word 4

        .text
        .globl main
main:
        la $t0 x
        lw $t1 0($t0)

        addi $sp $sp -4
        sw $t1 0($sp)

        la $t0 y
        lw $t1 0($t0)

        lw $t0 0($sp)
        addi $sp $sp 4

        mult $t0 $t1
        mflo $t0
```

Spilling (push)

Spilling (pop)

60