

---

# Introduzione al MIPS64

# Interfaccia Hardware/Software

High Level Language  
Program (e.g., C)

```
int a;  
a = vett[0] +7;
```

↓ *Compiler*

Assembly Language Program (e.g.,  
MIPS)

```
LW R2, vett(R0)  
ADDI R1,R2,7
```

↓ *Assembler*

Machine Language  
Program

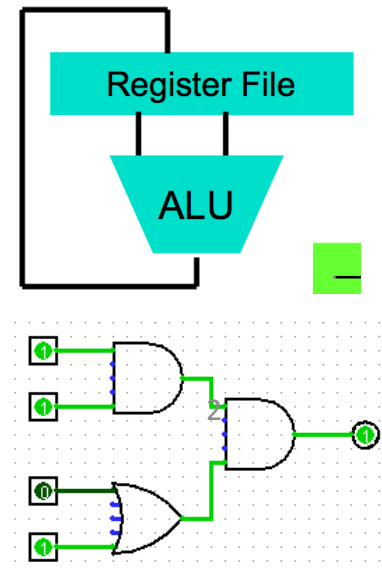
```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

↓ *Machine Interpretation*

Hardware Architecture Description  
(e.g., block diagrams)

↓ *Architecture Implementation*

Logic Circuit Description  
(Circuit Schematic Diagrams)

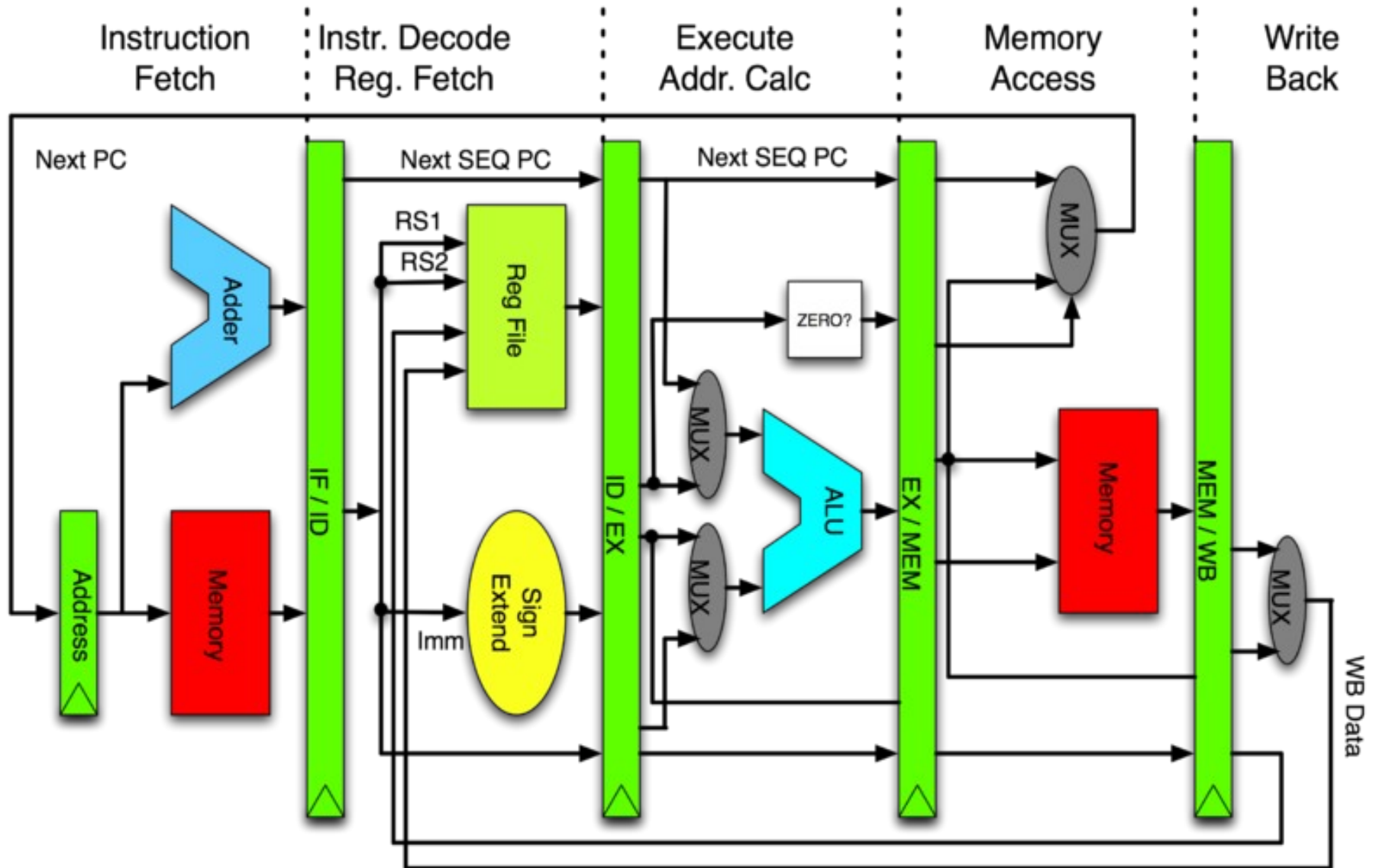


# MIPS64: caratteristiche

---

- Nel 1981 *John L. Hennessy* della Stanford University avviò un gruppo di ricerca sulle architetture RISC
- **RISC (vs CISC)**: Istruzioni complesse non implementate direttamente
- **Pipelining**: possibile grazie a istruzioni semplici che eseguono in un ciclo di clock.
- **Load/Store**: ogni dato va caricato dalla memoria con un'istruzione prima di essere manipolato

# MIPS Pipelining



# MIPS 64 Instruction Set Architecture

---

## ▪ **Concetti architetturali:**

- Semplicità del load/store IS
- Semplicità nella decodifica (istruzioni a lunghezza fissa da 32 bit)

## ▪ **Caratteristiche**

- Compatibilità con MIPS32 ISA
- 32x64-bit General Purpose Registers
- 32x64-bit Floating Point Registers
- La lunghezza di una word è di 64 bit.
- Supporta variabili a 8, 16, 32 e 64 bit
- Indirizzamento della memory a byte, Little Endian

# Registri

---

- L'ISA del MIPS64 contiene 32 (R0-R31) general-purpose registers da 64 bit
- I registri R1-R31 sono dei reali GP registers
- R0 contiene sempre il valore 0 e non può essere modificato
- R31 è utilizzato per conservare l'indirizzo di ritorno per le istruzioni JAL e JALR

# Note sull'uso dei registri

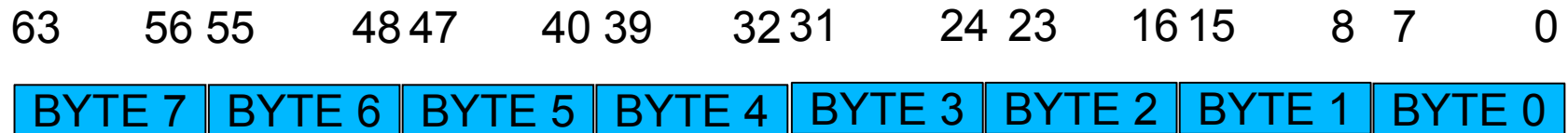
---

- I registri sono "omogenei", possiamo utilizzarli per allocare i valori scegliendoli come ci piace (eccetto R0, che vale sempre 0).
- In realtà, i compilatori adottano delle consuetudini per scegliere i registri: es. R29 come stack-pointer, R31 per l'indirizzo di ritorno dalle procedure, etc
- In quanto "compilatori-umani", cercare di fare attenzione e cercare di adottare uno stile: Es:
  - R2-R3: valori restituiti
  - R4-R7: parametri
  - R8-R15: indici, valori temporanei (temp, pippo, etc..)
  - R16-R27: deposito valori salvati (somma= .....

# Registri

---

- I bit dei registri sono numerati come 63-0, da destra a sinistra.
- L'ordinamento dei byte è fatto in modo simile



- Un registro può essere caricato con
  - × un byte (8-bit)
  - × un halfword (16-bit)
  - × una word (32-bit)
  - × Una double word (64-bit)



# Registri speciali

---

- **PC**, Program Counter, contiene l'indirizzo dell'istruzione da leggere dalla memoria (32 bit)
- HI e LO, due registri interni alla CPU a cui si accede mediante le istruzioni MFLO and MFHI

# Struttura programmi assembly EduMIPS64

---

- **Direttive:** Danno indicazioni sull'interpretazione del testo. Non corrispondono ad alcuna operazione dal punto di vista della semantica del programma assembly
- **Direttiva `.data`** Il codice generato dopo questa direttiva viene allocato nel segmento dei dati.
- **Direttiva `.code`** Il codice generato dopo questa direttiva viene allocato nel segmento testo.
- **Etichette:** associate ad un indirizzo di memoria
- **Commenti:** iniziano con `;`

# Struttura programmi assembly EduMIPS64

---

```
; This is a comment
```

```
.data
```

```
Label: .word 15 ;This is a comment
```

```
.code
```

```
daddi r1,r0,0
```

In questo esempio:

**Direttive**, **etichette**, **commenti**, **Istruzioni**

# Tool di Simulazione: EduMIPS64

---

- Sviluppato da alcuni studenti di Laboratorio di Calcolatori (anno 2006) presso la Facoltà di Ingegneria di Catania
- Multi-piattaforma (Java based)

<http://www.edumips.org>

Consiglio: scaricare al più presto e se possibile eseguire direttamente in aula gli esempi proposti

# Sezione .data

---

- Nella sezione .data possono essere memorizzati dati

<b>Tipo</b>	<b>Direttiva</b>	<b>N. Byte</b>
byte	.byte	1
Half word	.word16	2
Word	.word32	4
Double Word	.word o .word64	8

Es

.byte 2

.word16 524

.word 32 586

# Allineamento automatico

- Attenzione all'allineamento a 64 bit:

.data

.byte 1, 2

.byte 3

.byte 4



- **Esercizio:** provare a digitare sezioni `.data` e a prevedere cosa compare nella finestra "Data"

# Altre direttive

---

- `.space n`: lascia `n` bytes di spazio libero nella sezione `.data`
- `.asciiz "stringa"`: codifica la sequenza di caratteri in memoria e pone un byte nullo alla fine (terminatore C-like).
- NB: esiste anche una versione `.ascii`, senza 'z' finale, ad indicare l'assenza del terminatore.

# Sezione .code

---

- Contiene le istruzioni MIPS64 da eseguire. Tre tipologie di parametri:
  - **Registri:** 32 registri di 64 bit, indicati da  $R0 \dots R31$ . NB:  $R0$  è cortocircuitato a 0.

Esempio: `dadd R1, R2, R0`

- **Valori immediati:** espressi numericamente in decimale o esadecimale.

Esempio: `daddi R1, R1, 4`

**NB:** L'eventuale etichetta che precede un'istruzione è utilizzabile come un immediato che ha come valore l'indirizzo di memoria a cui si trova l'istruzione.



# Sezione .code

---

- **Indirizzi:** sono utilizzati dalle istruzioni load/store e sono specificati nel formato:

`offset (base)`

dove `base` è un registro che contiene il valore *base* e `offset` è un valore immediato che contiene lo *scostamento da sommare*.

Ad esempio, se R1 vale 1024

```
lw R2, 8 (R1)
```

caricherà la word che inizia all'indirizzo di memoria 8+1024.

# Classificazione delle Istruzioni

---

Le istruzioni possono essere classificate in base a due criteri:

- **Categoria:**

- aritmetico/logica (ALU),
- load/store,
- controllo di flusso,
- di sistema

- **Formato:**

- **R-type:** tutti gli operandi sono registri
- **I-type:** uno degli argomenti è un immediato (16 bit)
- **J-type:** salti che non utilizzano registri come destinazioni

# Istruzioni load/store: load double

---

## **LD Rt, offset(base)**

- Significato: *carica 64 bit sul registro Rt, leggendo a partire dal byte all'indirizzo offset+base, ossia:*

Rt = mem[offset+base] dove:

- L'argomento `base` specifica un registro
- L'argomento `offset` è un immediato (16 bit)
- **NB:** pur essendo MIPS64 un'architettura a 64bit, si usa il termine "double" per indicare 8byte, come tradizione delle usuali macchine a 32 bit

# Istruzioni Load/Store: load byte

---

**LB** Rt, offset(base)

**LBU** Rt, offset(base) (versione unsigned)

- Significato: carica il byte all'indirizzo offset+base sul registro Rt
- **NB:** la **LB** estende il bit del segno. Es: se il byte vale -2 sarà codificato come 1111 1110, caricandolo su 64 bit DEVE diventare:

1111 .... 1111 1111 1111 1111 1111 1110

**NON**

0000 .... 0000 0000 0000 0000 1111 1110

# Istruzioni load/store: LH

---

**LH** *Rt*, **offset**(**base**)

**LHU** *Rt*, **offset**(**base**) (versione unsigned)

- Significato: *carica mezza word (2 byte) a partire dall'indirizzo offset+base sul registro Rt*
- **NB:** analogamente a quanto visto con LB, la LH estende il bit del segno sui rimanenti 48 bit più alti a sinistra.

# Istruzioni load/store: load word

---

**LW** *Rt*, *offset*(*base*)

**LWU** *Rt*, *offset*(*base*) (versione unsigned)

- Significato: carica il 32 bit (4 byte) a partire dall'indirizzo  $\text{offset} + \text{base}$  sul registro *Rt*
- **NB:** analogamente a quanto visto con **LB**, la  $\text{LH}$  estende il bit del segno sui rimanenti 32 bit più alti a sinistra.
- **NB:** l'architettura è a 64 bit, ma per tradizione rispetto a MIPS32 in questo caso caricare una word significa caricare 32 bit

# Istruzioni store

---

**SD** *Rt*, *offset* (**base**)

**SW** *Rt*, *offset* (**base**)

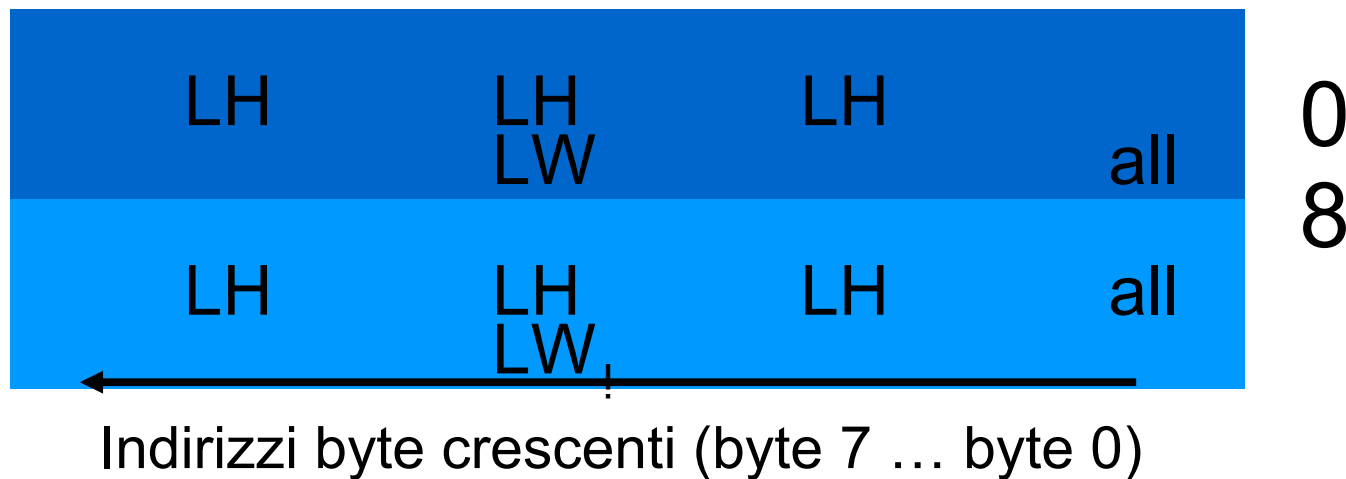
**SH** *Rt*, *offset* (**base**)

**SB** *Rt*, *offset* (**base**)

- Significato: *Memorizzano il contenuto di Rt, partendo da offset+base*
- NB: non ci sono le versioni unsigned, prendono solo i byte che servono e li scrivono
- Es: **SB** *R1*, 0 (*R2*) prende solo un byte da *R1* e lo memorizza ignorando il resto

# Load/Store: allineamento

- Ogni accesso alla memoria deve partire ad un *indirizzo allineato rispetto alla dimensione del dato*. **Es.** LD (carica 8 byte) accede gli indirizzi 0, 8, 16, 24 etc.. LW (carica 4 byte) indirizzi 0,4,8,12 etc.. LH in 0,2,4,6,8 e LB ovunque!





# Istruzioni per il flow-control

---

- Salti condizionati:

***BEQ**  $R_s, R_t, offset$*

*Salta all'indirizzo  $offset$  se  $R_t = R_s$*

- Correlate: **BNE** (registri diversi anzichè uguali)

***BEQZ**  $R_s, offset$*

*Salta all'indirizzo  $offset$  se  $R_t = 0$*

*Correlate: **BNEZ** (salta se  $R_t \neq 0$ )*

# Istruzioni per il flow-control

---

- Salti incondizionati:
  - **JR Rs**: salta all'indirizzo contenuto in Rs (ossia copia Rs nel Program Counter)
  - **J target**: salta all'indirizzo specificato nell'immediato (es. Una etichetta)
  - **JALR Rs**: come JR, in più mette l'indirizzo dell'istruzione successiva (PC+8) dentro R31, in modo che si possa ritornare dal salto mediante una JR R31
  - **JAL target**: come JALR, ma usa l'immediato invece che il registro Rs

# Istruzioni flow-control

---

- Che differenza c'è tra le due famiglie di salti ?
- Salti condizionati, solitamente *traduzione di strutture "if"*, salti relativamente vicini
- Salti incondizionati, *chiamata a procedure*, salti a regioni di codice anche molto distanti
- Le varie implementazioni di MIPS usano vari trucchetti per aumentare la raggiungibilità del codice (es. Salto relativo al PC)

# Istruzioni ALU

---

Suddivisibili in quattro sotto-categorie:

- **Aritmetiche**
- **Logiche**
- **Shift**
- **Comparazioni**

# ALU: somma

---

***ADD Rd, Rs, Rt*** (somma a 32 bit)

***DADD Rd, Rs, Rt*** (somma a 64 bit)

- Significato:  $Rd = Rs + Rt$

Correlate:

- ***ADDU, DADDU***: interpreta i valori nei registri come unsigned (in alcune implementazioni ignora overflow)
- ***ADDI, DADDI***: al posto di *Rt* specifica un valore immediato
- ***ADDUI, DADDUI***: la combinazione delle due precedenti

Domanda:

*Come si inizializza un registro?*

# ALU: sottrazioni

---

**SUB** *Rd*, *Rs*, *Rt*,            **DSUB** *Rd*, *Rs*, *Rt*

- Significato:  $Rd = Rs - Rt$
- Correlate:
  - **SUBU**, **DSUBU**: interpreta i valori dei registri come unsigned

Domanda:

*Perchè non c'è la DSUBI ??*

# ALU: moltiplicazioni

---

**DMULT** *Rs*, *Rt*

- Significato:  $HI, LO = Rs \times Rt$

NB: HI e LO sono registri speciali, riservati per alcune istruzioni

- Correlate:
  - **DMULTU**: la versione unsigned
  - **MFLO** *Rd*: copia il valore di LO nel registro *Rd*, in quanto le normali istruzioni non possono accedere a LO

# ALU: divisioni

---

## **DDIV Rs , Rt**

- Significato: *esegue  $R_s/R_t$  e pone i quoziente in LO ed il resto in HI*
- Correlate:
  - **MFHI Rd**, analoga a MFLO, per spostare il valore del registro speciale HI in Rd



# ALU: istruzioni logiche

---

## **AND Rd, Rs, Rt**

- Significato: *AND logico bit a bit tra Rs e Rt*
- Correlate:
  - **ANDI Rd, Rs, Immediato**
  - **OR, XOR** analoghe per le altre funzioni logiche

## Domanda:

- *Come usarle per controllare se un numero è pari ?*

# ALU: istruzioni di Shift

---

**DSLL** *Rd*, *Rt*, *shamt*

- Significato:  $Rd = Rt \ll shamt$

**NB:** *shamt* (shift amount) è un valore immediato.

Es: **DSLL** *R1*, *R2*, *2*

- Correlate:

- Versione con i registri: **DSLLV** *Rd*, *Rs*, *Rt*

- **NB:** Dato che lo shift è verso sinistra, la parte destra è riempita con zeri, come una moltiplicazione per potenze di due

# ALU: istruzioni di Shift

---

**DSRL Rd, Rs, shamt**

**DSRLV Rd, Rs, Rt**

- Shift verso destra, simmetriche rispetto alle DSLL.
- La parte sinistra viene riempita con zeri. Dal punto di vista logico è ok, ma dal punto di vista "aritmetico" non è più come dividere per 2

Es:  $-2 = 1111 \dots 1111 1111 1111 1111 1111 1111 1110$  diventa:  
 $0111 \dots 1111 1111 1111 1111 1111 1111$

ossia  $2^{31} - 1$ , detto anche 2.147.483.647 !

# ALU: shift aritmetico

---

**DSRA** *Rd, Rs, shamt*

**DSRAV** *Rd, Rs, Rt*

- **Estensione del segno:** Se il bit più a sinistra di *Rs* è zero, riempie a sinistra con altri zeri, altrimenti con 1
- Nel caso della slide precedente sul -2:

1111 .... 1111 1111 1111 1111 1111 1110

diventa:

1111 .... 1111 1111 1111 1111 1111 1111

ossia -1 (aritmeticamente meglio rispetto al caso precedente)

# ALU: comparazioni

---

**SLT Rd, Rs, Rt**

- Significato: *Se  $R_s < R_t$  allora  $R_d = 1$  altrimenti 0.*
- Correlate:
  - **SLTU**: interpreta i valori come unsigned
  - **SLTI**: un valore immediato al posto di Rt
  - **SLTUI**: combinazione delle due precedenti

Domanda:

*...e le altre condizioni ? (>, <=, etc..)*

## ...e basta ?

---

LUI Rt, immediate

- Carica i 16 bit dell'immediato nei 16 più alti della metà di 32 bit più bassa di Rt

Es: supponiamo Rt:

byte7 byte6 byte5 byte4 **byte3** **byte2** **byte1** **byte0**

allora dopo LUI Rt, 2049 (ossia 0000 1000 0000 0001)

0000 .... 0000 0000 **0000 1000 0000 0001** **0000 0000 0000 0000**

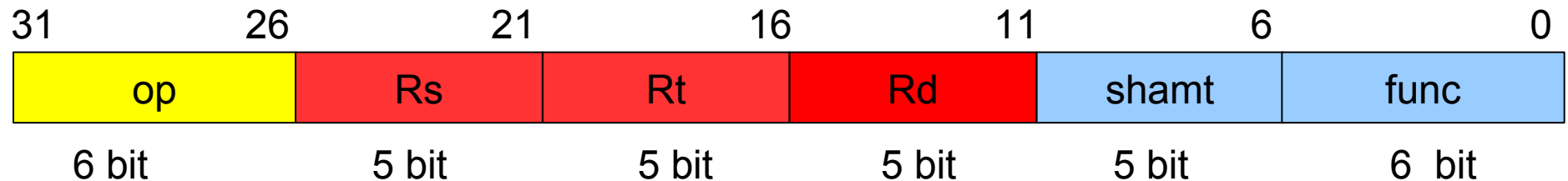
- NB: i byte 7..4 a sinistra subiscono l'estensione del segno (tutti 1 o zero a seconda del segno di Rt)

# Formato delle Istruzioni

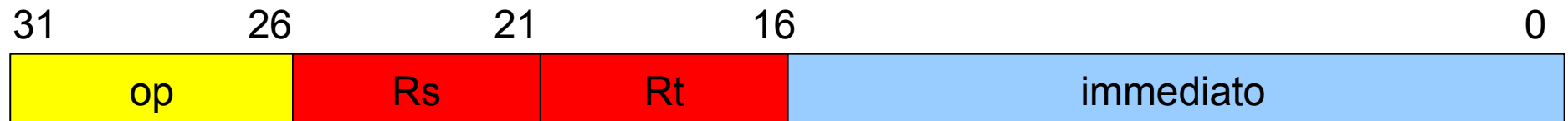
---

Tutte le istruzioni sono codificate in 32 bit

- R-type:



- I-type:



- J-type:



# Alcuni esempi di codifica

---

- Le `DADD`, `DSUB` sono R-type: `Rd` specifica il registro destinazione, `Rs` e `Rt` i sorgenti, il campo `func` insieme ad `op` il nome dell'operazione mentre `shamt` è ignorato
- Le load/store sono I-type
- Le shift utilizzano `shamt`, quindi sono R-type, anche se sembrano avere un immediato
- Le jump del tipo `J label` sono J-type, mentre le `JR Rt` sono R-type
- `beq/bne Rs, Rt label` sono I-type



# Chiamate di Sistema

---

## `syscall n`

- Utilizzate come interfaccia col sistema operativo, funzioni diverse ( $n = 0..5$ )
- **Assunzioni importanti:**
  - I parametri di una `syscall` devono essere posti, consecutivamente, in un indirizzo che va specificato in R14
  - Il valore di ritorno sarà posto in R1
- Sono simili alle chiamate `exit()`, `open()`, `close()`, `read()`, `write()`, `printf()`

# Syscall 0 – exit()

---

## `syscall 0`

- Termina il programma, non ha argomenti di input e di output

# syscall 1 - open()

---

## **syscall 1**

- Significato: *apre un file*
- Due parametri:
  - L'indirizzo di una stringa terminata con byte 0 che indica il path del file da aprire
  - Un intero che specifica la modalità di apertura (vedere manuale EduMIPS64 per l'elenco)
- Valore resituito:
  - Un intero corrispondente al file descriptor della risorsa aperta

# syscall 2 – close()

---

## **syscall 2**

- Significato: *chiude un file precedentemente aperto*
- Un parametro:
  - Un intero corrispondente al file descriptor
- Valore restituito:
  - Codice che indica il successo o meno (vedere manuale EduMIPS64)

# syscall 3: read()

---

## **syscall 3**

- Significato: *legge un certo numero di byte da un file ponendoli in una parte di memoria*
- Tre parametri:
  - Un intero corrispondente al file descriptor (NB: 0=standard input)
  - Un indirizzo di memoria dove porre i byte letti
  - La quantità di byte da leggere
- Valore restituito:
  - Il numero di bytes letti, -1 in caso di errore

# syscall 4: write()

---

## syscall 4

- Significato: *scrive su file un insieme di byte letti da una certa regione della memoria*
- Tre parametri:
  - Un intero corrispondente al file descriptor del file su cui scrivere (NB: 1=standard output)
  - L'indirizzo di memoria a partire dal quale leggere
  - Il numero di byte da scrivere
- Valore restituito:
  - Numero di byte scritti o -1 in caso di errore

# syscall 5: printf(...)

---

## **syscall 5**

- Significato: *stampa un messaggio sulla base di una stringa di formattazione (stile `printf C`)*
- Variabile numero di parametri:
  - Il primo è l'indirizzo della stringa di formattazione, terminata con byte 0
  - I successivi, per ogni segnaposto (es %d), seguono subito dopo
- Valore restituito:
  - Numero di byte stampati, -1 in caso di errore

# Esempio utilizzo syscall 5

---

**; NB: esempio con due valori, uno pronto l'altro no**  
**.data**

```
mess:      .ascii "Stampa i numeri %d e %d"
arg_printf: .space 8
num1:      .word 255
num2:      .space 8
str:       .ascii "stringa non usata"
```

**.code**

```
addi r14, r0, arg_printf
addi r8, r0, mess
sd r8, arg_printf(r0)
addi r9, r0, 50 ; scelta di 50 arbitraria
sd r9, num2(r0)
```

```
syscall 5
syscall 0
```

Eseguire step-by-step con F7 osservando i registri e la memoria



# Chiamata a Procedure

---

- Si utilizzano `JAL` e `JR`

```
... .                               procA:  
JAL procA                            ... .  
... .                               JR R31
```

- NB: `jal`, prima di saltare all'indirizzo indicato, setta `R31` a `PC +8` (prossima istruzione)
- NB: Alla fine, prima terminare, la procedura salta all'indirizzo indicato in `R31`

# Chiamate annidate

---

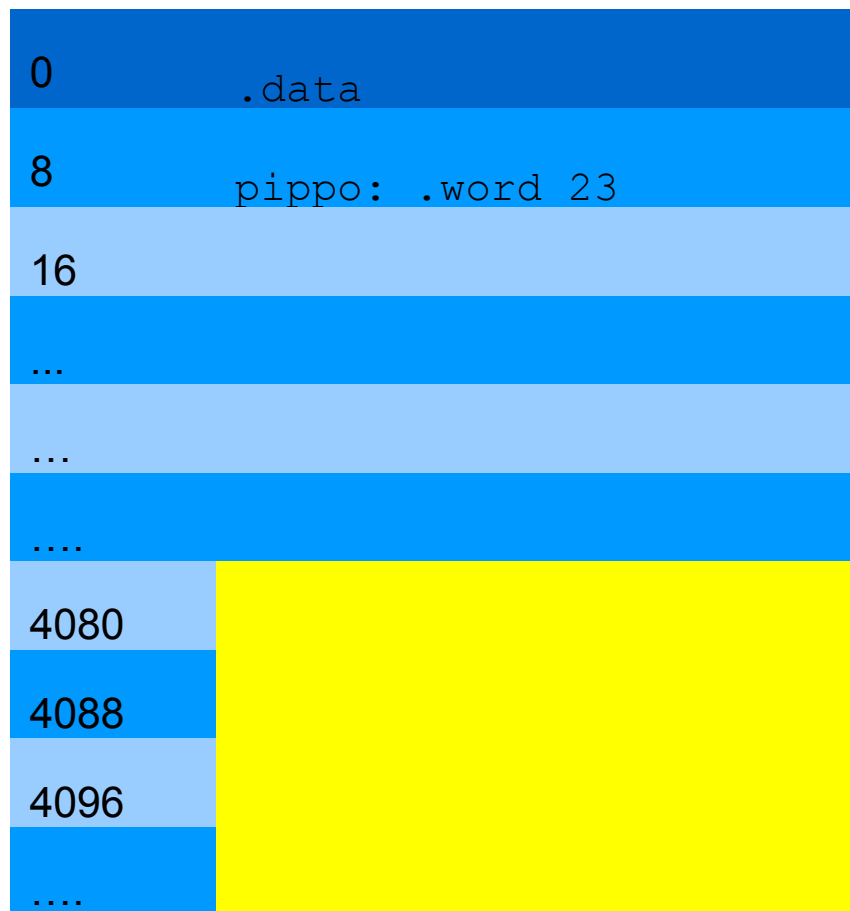
ProcA:	ProcB:
...	...
JAL procB	JR R31
...	
JR R31	

Ipotizziamo che inizialmente la procA venga invocata da una `jal procA` che si trova all'indirizzo 1024 della sezione `.code`, riusciremo mai a continuare sull'istruzione successiva che si trova a `PC+8` ossia 1032 ?

# Stack pointer

---

- In un stack (pila) vengono accumulati i valori salvati da recuperare, R29 indica la cima.
- NB: lo stack cresce verso l'alto (indirizzi più piccoli



R29 = 4080



# Chiamate con stack pointer

---

ProcA:

**DADDI R29,R29,-8**

**SD R31 0(R29)**

JAL procB

**LD R31,0(R29)**

**DADDI R29,R29,8**

JR R31

ProcB:

...

JR R31

Se la procA è invocata dall'indirizzo 1024, il valore 1032 è salvato sullo stack e poi recuperato

# Aggiornamento stack pointer

---

- Ogni procedura che chiama un'altra deve occuparsi di aggiornare lo stack pointer e salvare quello che andrà recuperato

