UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

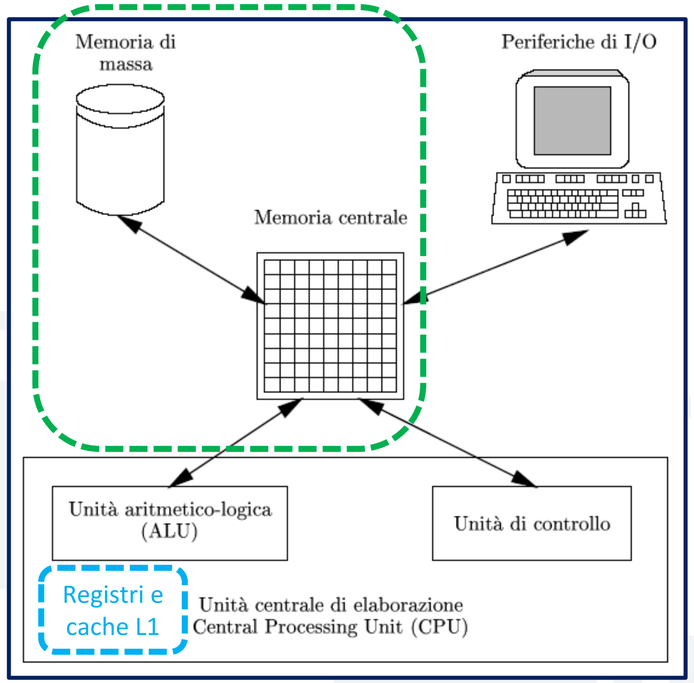
Instruction Set Architecture (ISA)

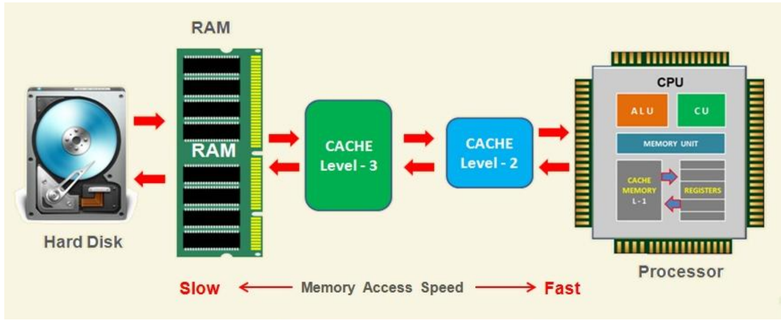
Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

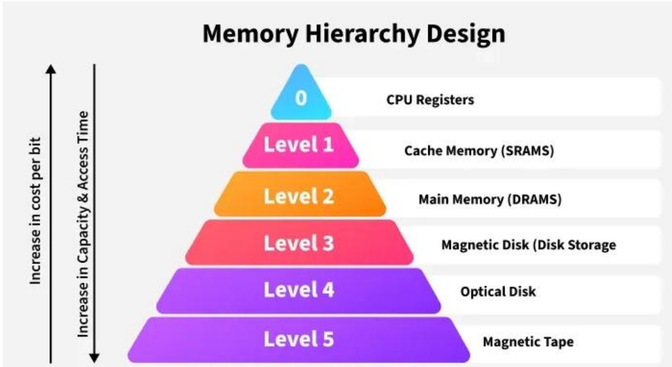
Trieste, 5 marzo 2025

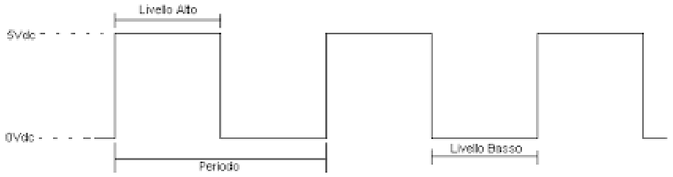
GERARCHIE DI MEMORIE E SYSTEM BUS

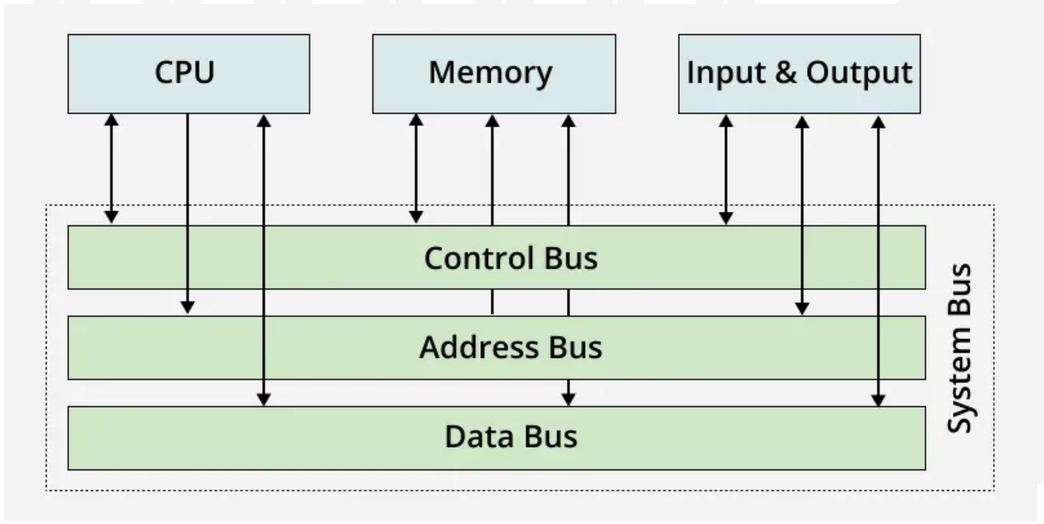




Patterson & Hennessy, Cap.5

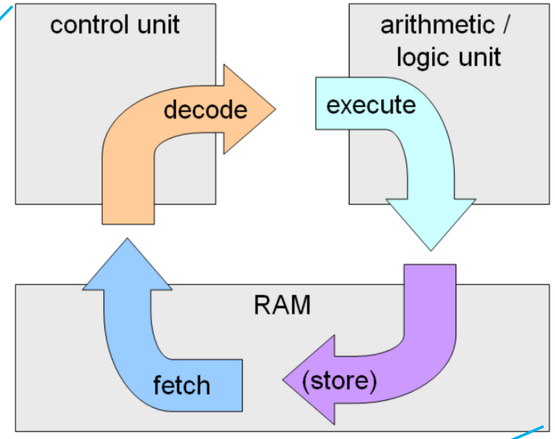


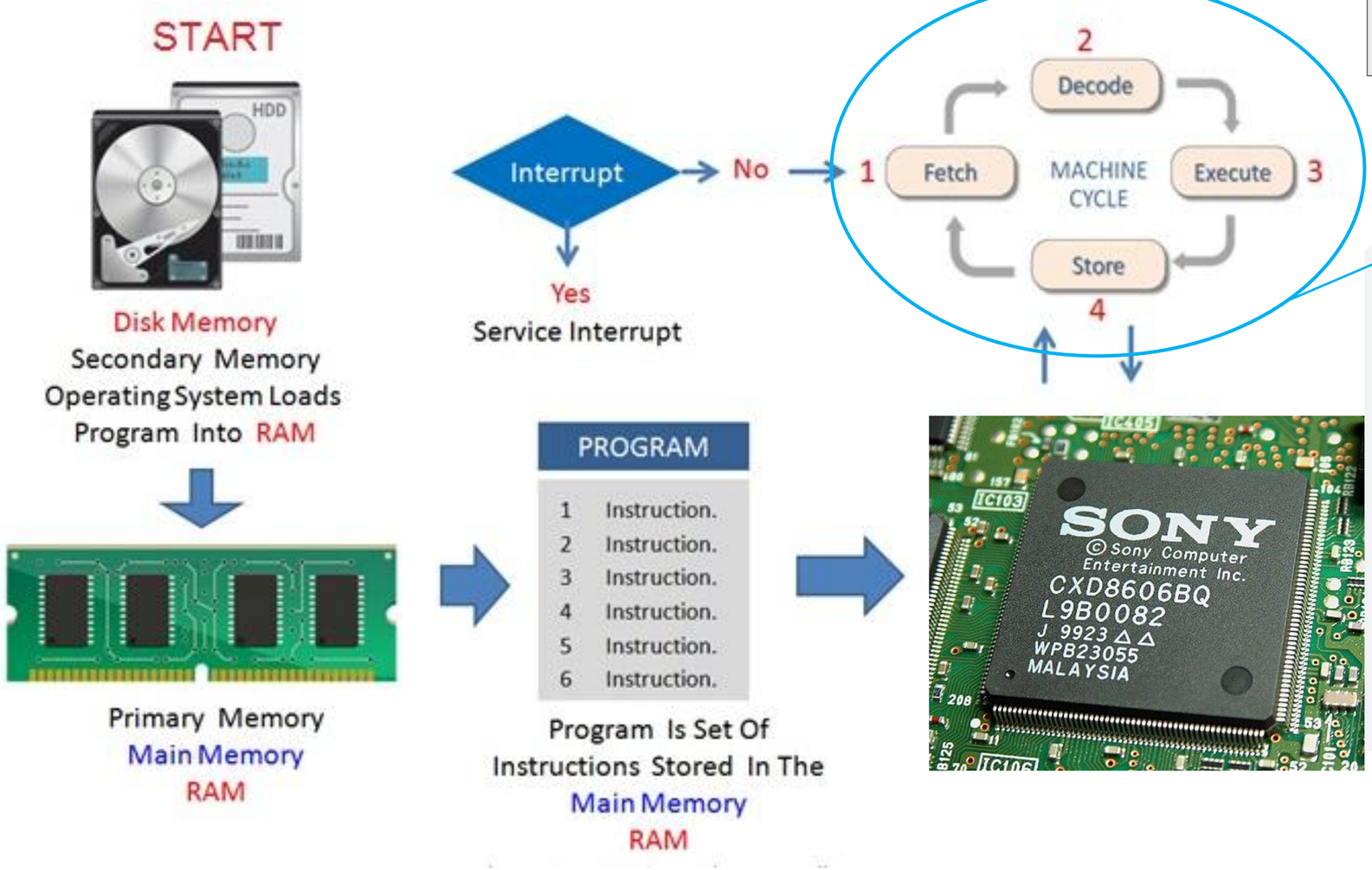




UNIVERSITÀ DEGLI STUDI DI TRIESTE

DATAPATH E
FASI DI ESECUZIONE DI UN'ISTRUZIONE





MIPS32

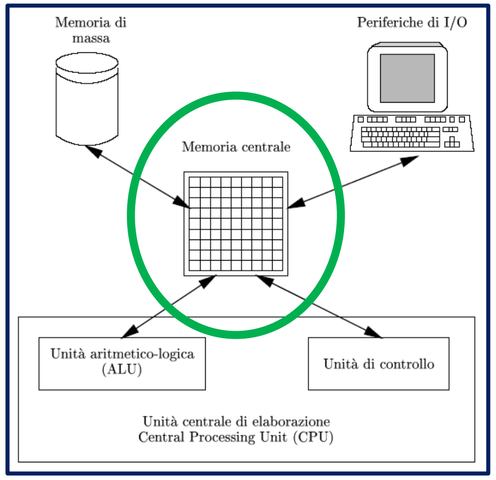
QtSpim

NOTA: nella memoria RAM si memorizzano **sia le istruzioni del programma che i dati dell'utente** durante l'esecuzione

AGENDA DI OGGI

- *Recap*
- Convenzioni sull'uso della memoria RAM e puntatori principali
- Instruction Set Architecture (ISA) per MIPS32
  - formato istruzioni
  - indirizzamento alla memoria
  - Appendix A Patterson&Hennessy

MEMORIA RAM



Con MIPS32 possiamo «**indirizzare al byte**»

Ogni byte ha un suo indirizzo individuale

Potremmo inserire un byte di informazione alla volta, *però…*

bit									
0	0	1	0	1	1	0	0	**0** ← byte	
1	0	0	1	1	1	1	0	1	
0	0	0	1	1	1	1	0	2	
1	0	0	1	0	0	1	0	3	
1	0	0	1	1	1	0	1	**4**	0x00400000
0	0	1	1	1	0	0	1	5	0x00400001
1	1	1	1	0	0	1	0	6	0x00400002
0	0	1	1	0	0	1	1	7	0x00400003
1	0	0	0	1	1	1	1	**8**	
0	1	1	0	0	1	0	0	9	

I bit che ho inserito sono del tutto ***casuali***!

MEMORIA RAM

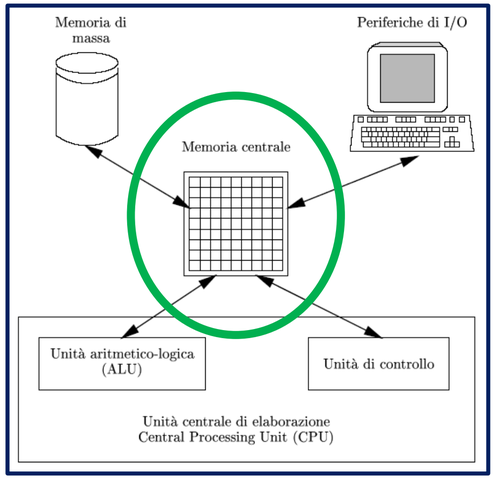
Però in realtà PER CONVENZIONE e per semplificare l'architettura, si segue un **ALLINEAMENTO A WORD**

Si scrivono le info (dati e istruzioni) **a gruppi di 4 byte** (=1word)

0	0	1	0	1	1	0	0	**0**
1	0	0	1	1	1	1	0	1
0	0	0	1	1	1	1	0	2
1	0	0	1	0	0	1	0	3
1	0	0	1	1	1	0	1	**4**
0	0	1	1	1	0	0	1	5
1	1	1	1	0	0	1	0	6
0	0	1	1	0	0	1	1	7
1	0	0	0	1	1	1	1	**8**
0	1	1	0	0	1	0	0	9

WORD

WORD



I bit che ho inserito sono del tutto ***casuali***!

MEMORIA RAM

La RAM è utilizzata in parte per memorizzare le istruzioni del programma da eseguire (***segmento TEXT***) e in parte per i dati dell'utente (***segmento DATA***).

Le **istruzioni** sono già «naturalmente» di 32 bit, quindi vengono scritte in 4 celle di memoria consecutive
e il **segmento TEXT** della memoria «è allineato» per forza

ALLINEAMENTO

1100 0000 0000 0000 0000 0100 0011 1011

Può essere l'indirizzo di un'istruzione?

Dobbiamo verificare se è «allineato», ovvero se porta ad una cella di memoria (da 8 bit) che ha come indice un multiplo di 4.

Verifica: guardiamo gli ultimi 2 bit. 11 =3 → 00

1100 | 0000 | 0000 | 0000 | 0000 | 0100 | 0011 | 1011

C | 0 | 0 | 0 | 0 | 4 | 3 | B → 0xC000043B

Un INDIRIZZO MIPS32 ALLINEATO multiplo di 4 termina sempre con una cifra tra: 0, 4, 8, c

Gli ultimi 4 bit devono essere:

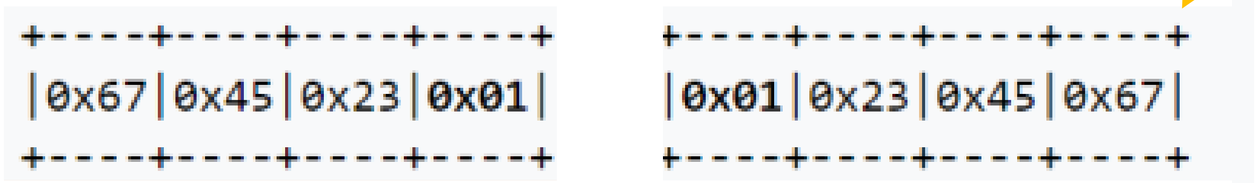
0000	0100	1000	1100
0x0	0x4	0x8	0xc

BIG-ENDIAN VS LITTLE-ENDIAN

lo standard didattico su QtSpim è **BIG-endian**

Memorizzare una word (4 byte): come numeriamo i byte?

Esempio. La word 0x01234567 può essere memorizzata in due modi:



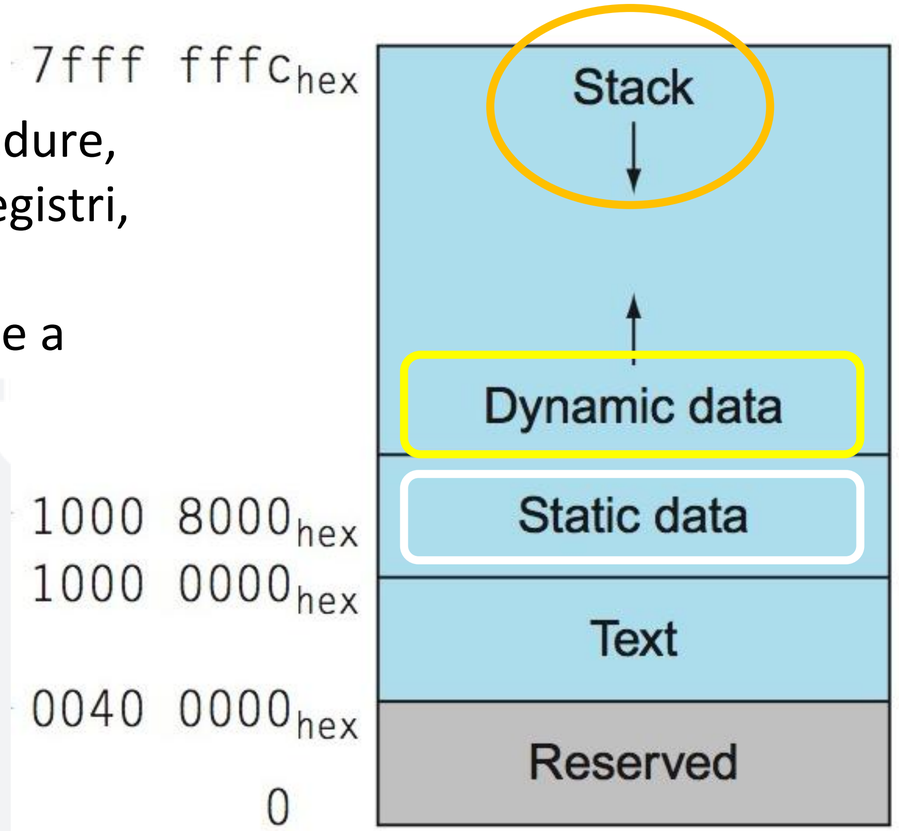
Il byte di indice più piccolo può essere il byte più a sinistra (***big-endian***) o il più a destra (***little-endian***).

La **convenzione** che regola la scelta per ogni macchina è chiamata "***byte order***".

I processor MIPS possono operare sia in un modo che nell'altro. **Nel simulatore SPIM l'ordine del byte viene "imposto" dalla macchina su cui gira il simulatore stesso.** Su Intel 80x86, SPIM è little-endian, mentre su Macintosh or Sun SPARC, SPIM è big-endian.

MEMORIA RAM: CONVENZIONE D'USO (1/2)

Stack: parametri delle procedure, variabili locali, salvataggio registri, return address, gestione ricorsione. Cresce/diminuisce a ogni chiamata funzione.

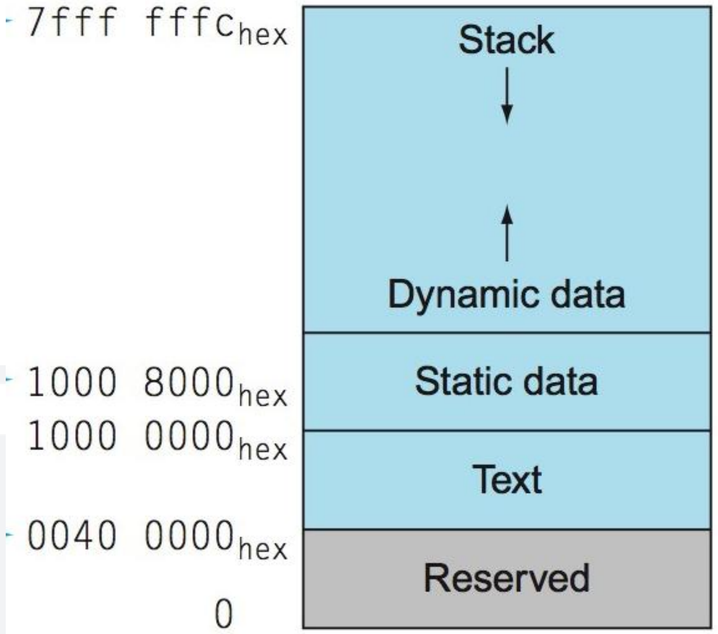


Dynamic data: strutture dati di dimensione NON nota a priori, array dinamici, liste concatenate, alberi, … Cresce dinamicamente durante l'esecuzione. È gestita dal programmatore.

Static data: variabili globali, costanti, stringhe, array di dimensione nota. La dimensione è fissata a inizio esecuzione.

Patterson & Hennessy, Capitolo 2

MEMORIA RAM: CONVENZIONE D'USO (2/2)



Segmento testo (Text segment):

- Indirizzi da **0x00400000** a circa **0x0FFFFFFF**.
- Contiene fino a circa **256 MB** per le istruzioni.

Segmento dati (Data segment):

- Indirizzi da **0x10010000** a circa **0x7FFFFFFF**.
- Può utilizzare fino a circa **2 GB** per i dati.

Stack: Inizia a partire dall'indirizzo **0x7FFFFFFC** e cresce verso indirizzi inferiori.

KERNEL (per il sistema operativo): Area riservata da **0x80000000.**

AREA DI MEMORIA	1. INDIRIZZI	DIMENSIONE MASSIMA TEORICA
SEGMENTO TESTO	0x00400000–0x0FFFFFFF	~256 MB
SEGMENTO DATI	0x10010000–0x7FFFFFFF	~2 GB
STACK	Parte da 0x7FFFFFFC	incluso nei 2 GB precedenti
KERNEL	0x80000000–0xFFFFFFFF	~2 GB

MEMORIA RAM: PUNTATORI

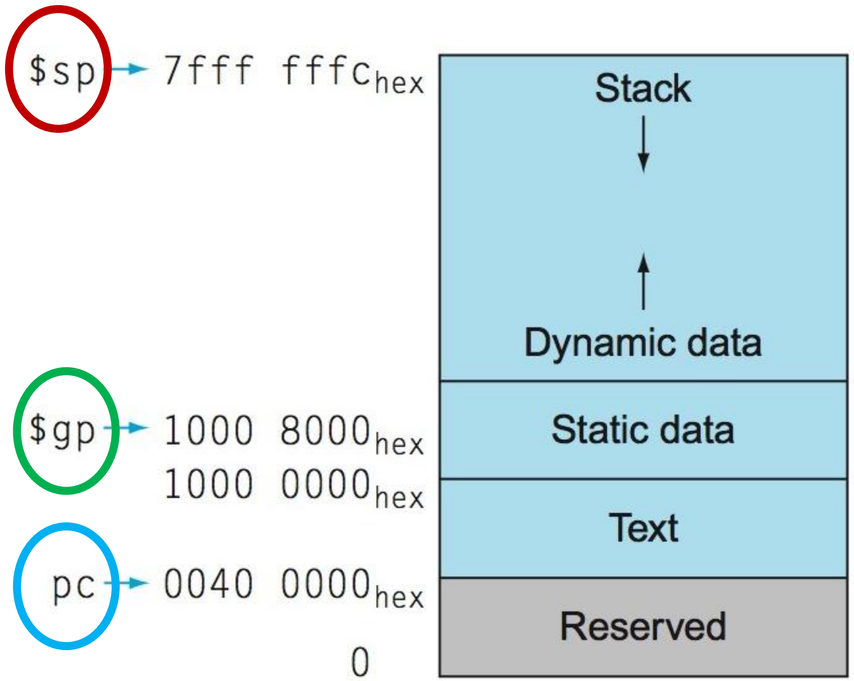
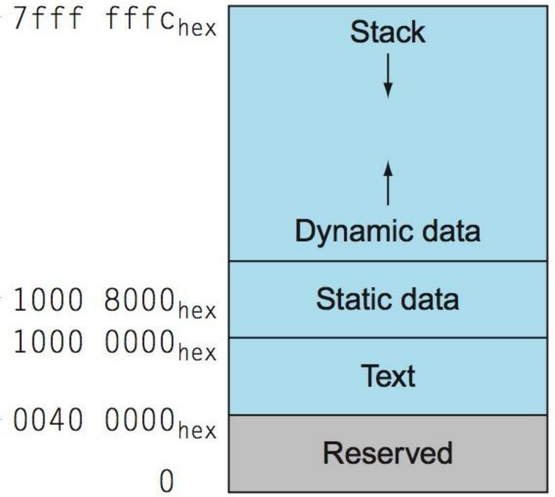


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 0000$_{hex}$. The static data starts at 1000 0000$_{hex}$. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, $gp, is set to an address to make it easy to access data. It is initialized to 1000 8000$_{hex}$ so that it can access from 1000 0000$_{hex}$ to 1000 ffff$_{hex}$ using the positive and negative 16-bit offsets from $gp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

STACK POINTER ($sp)

Lo **Stack Pointer** è un registro (in MIPS è $sp) che indica l'**indirizzo della cima dello stack in memoria**.



Lo stack è un'area di memoria utilizzata per memorizzare temporaneamente dati (ad esempio variabili locali, indirizzi di ritorno di funzioni, parametri di funzioni, ecc.).

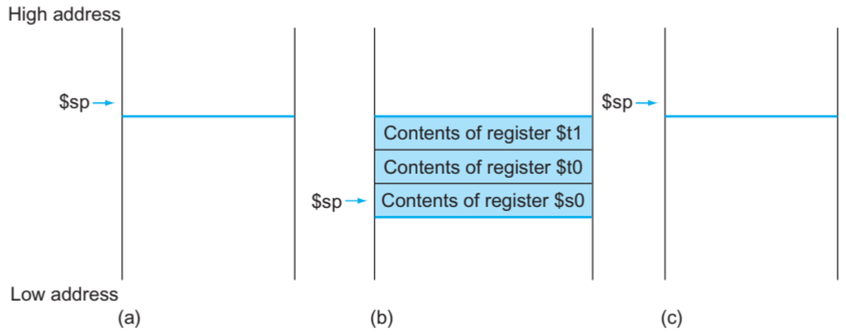
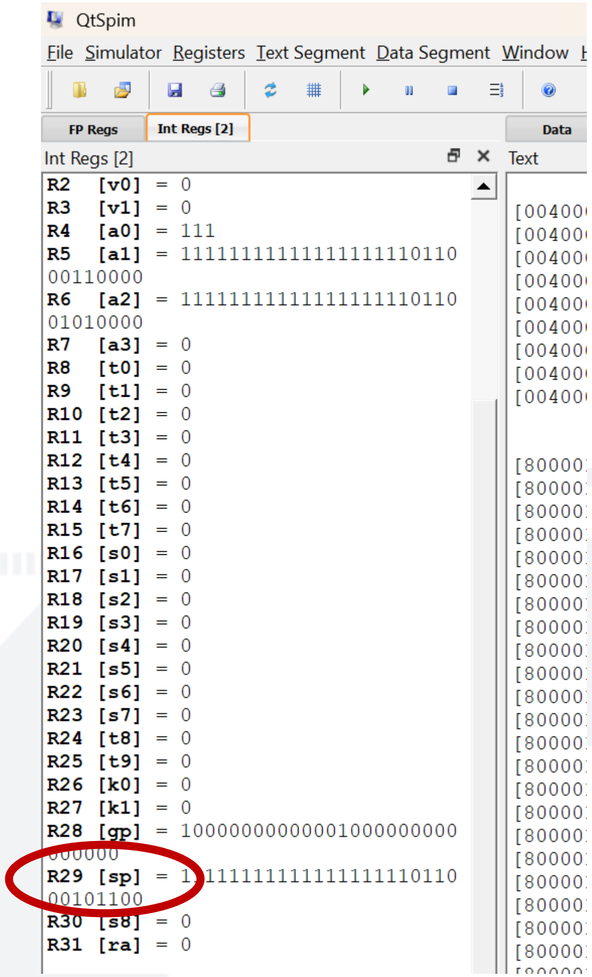


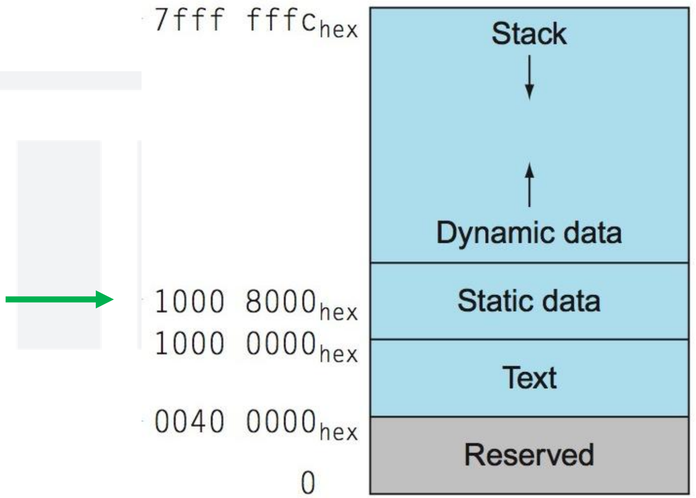
FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.



Cresce verso indirizzi più bassi: ogni volta si aggiunge qualcosa allo stack, $sp viene decrementato.

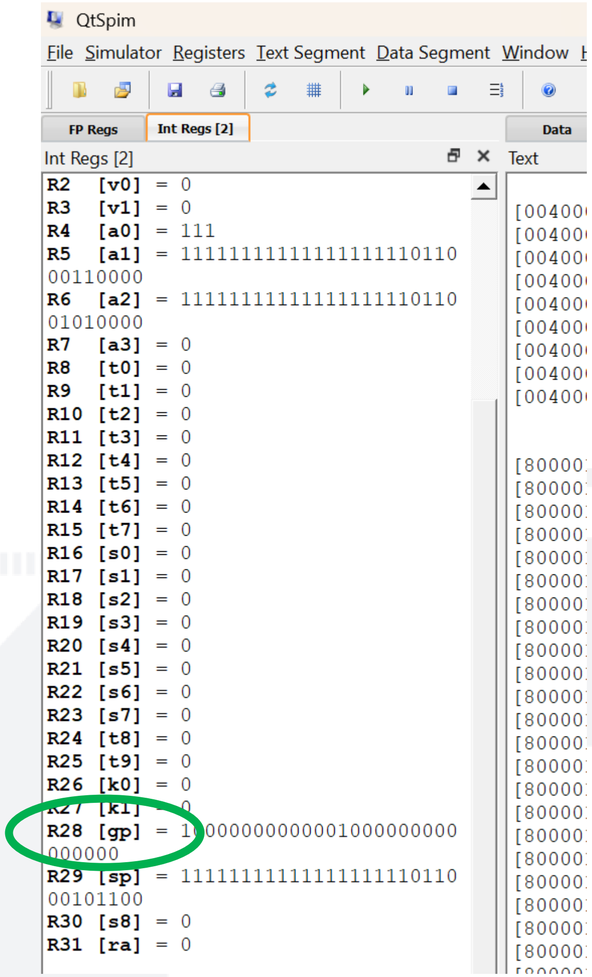
GLOBAL POINTER ($gp)

È un registro speciale in MIPS32 che contiene un **indirizzo di riferimento** per accedere rapidamente alle variabili globali in memoria.



È utile **per accedere a dati statici/globali con indirizzamento breve e veloce**.

Solitamente è inizializzato a un indirizzo centrale nel segmento dati, permettendo di accedere facilmente (con un offset piccolo) alle variabili globali più comuni.

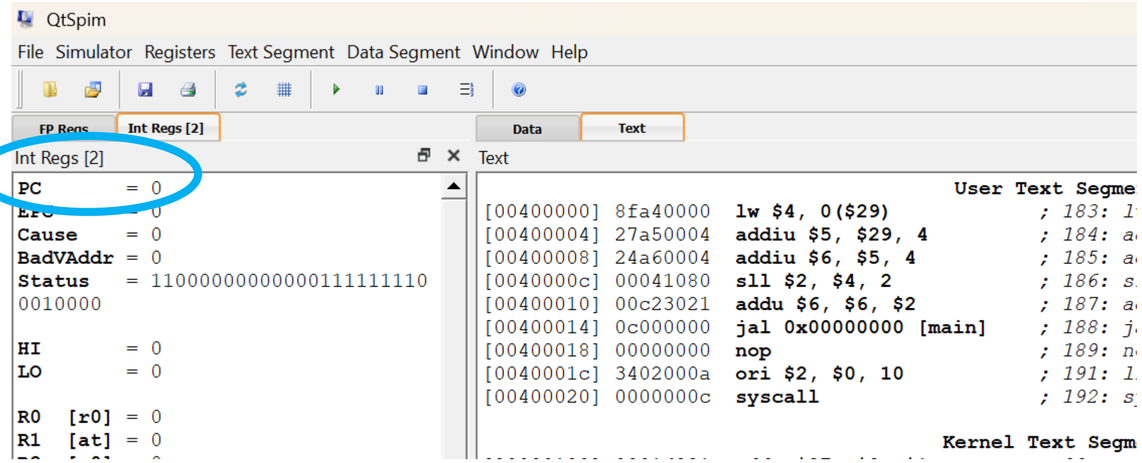


Accesso alle variabili globali più efficiente rispetto all'indirizzamento assoluto.

PROGRAM COUNTER (PC)

Il **Program Counter (PC)** in MIPS32 è un registro speciale della CPU che ha il compito di tenere traccia dell'**indirizzo della prossima istruzione** da eseguire.

- È un registro a **32 bit**.
- L'indirizzo in esso contenuto è sempre **multiplo di 4**, dato che ogni istruzione MIPS32 occupa **4 byte**.
- Dopo che un'istruzione viene letta (*fetch*), il PC viene automaticamente incrementato di **4**, passando all'istruzione successiva.



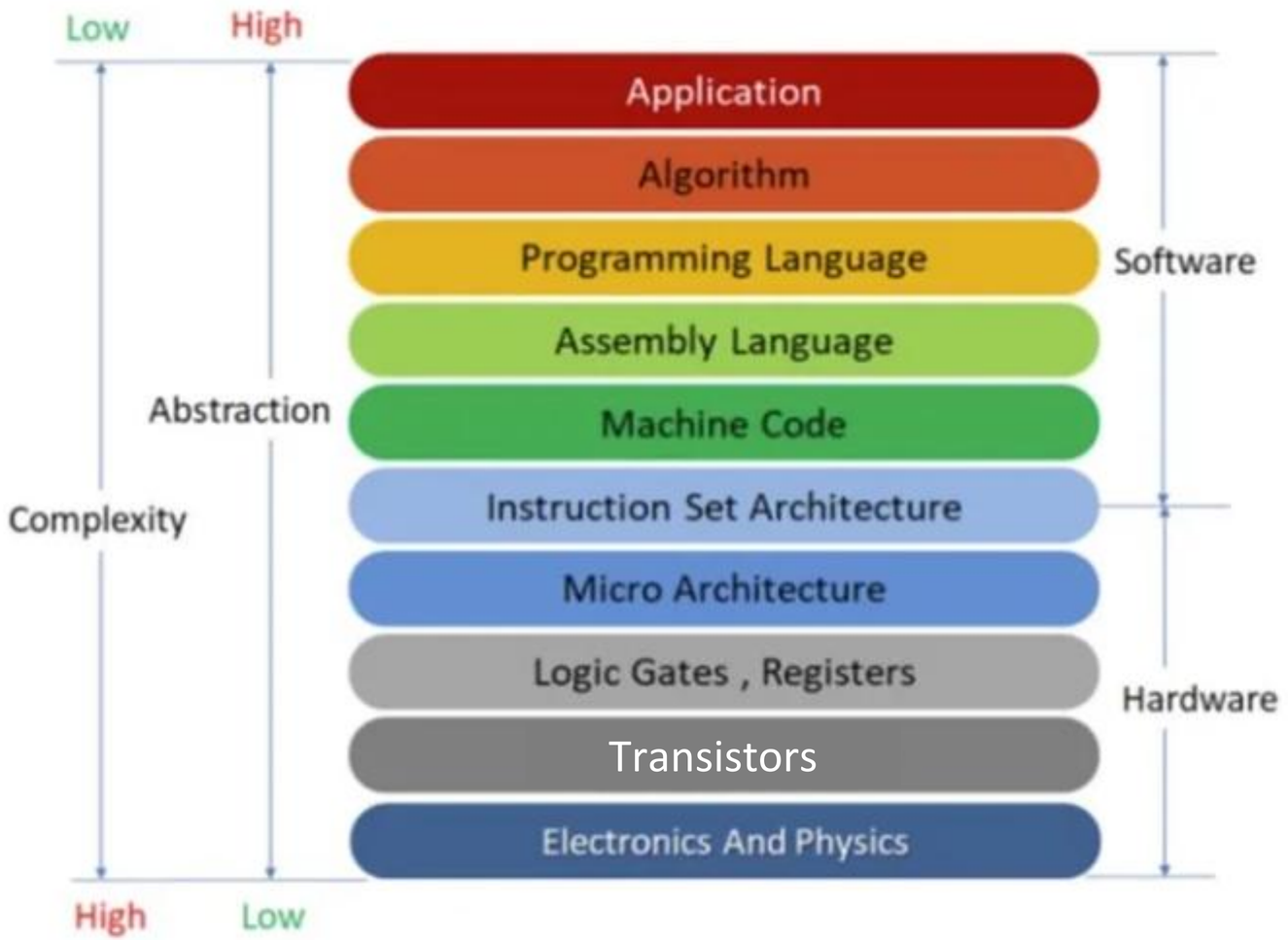
Istruzioni sequenziali (la maggior parte): PC = PC + 4

Istruzioni speciali (branch, jump, chiamate funzioni) modificano il PC diversamente.

AGENDA DI OGGI

- *Recap*
- *Convenzioni sull'uso della memoria RAM e puntatori principali*
- Instruction Set Architecture (ISA) per MIPS32
  - formato istruzioni
  - indirizzamento alla memoria
  - Appendix A Patterson&Hennessy

INSTRUCTION SET ARCHITECTURE (ISA)



dipendenza

UNIVERSITÀ DEGLI STUDI DI TRIESTE

INSTRUCTION SET ARCHITECTURE

00000001000010010101000000100000

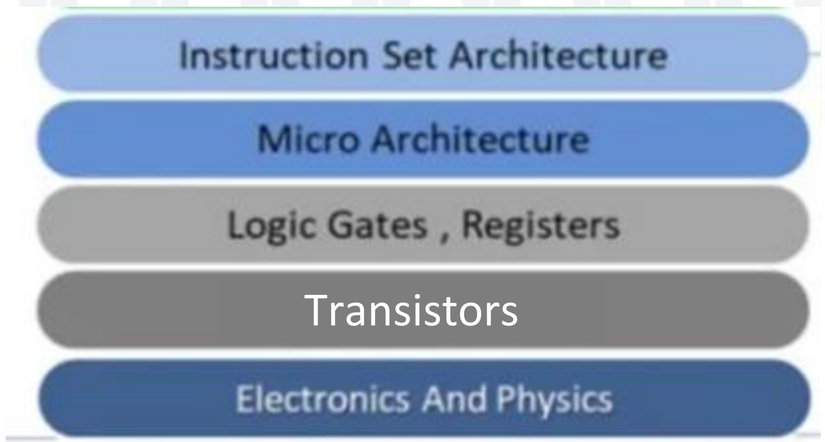
I primi 6 bit definiscono il cosiddetto OPCODE, ovvero il codice dell'operazione che va eseguita.

IL/I successivo/i *blocchi di bit* determinano i registri (indirizzo o valore contenuto) da cui prendere gli operandi.

Il *blocco di bit* seguente determina dove andrà messo il risultato (un altro registro).

Poi ci sono altri bit che danno informazioni necessarie per implementare altre operazioni o varianti delle stesse.

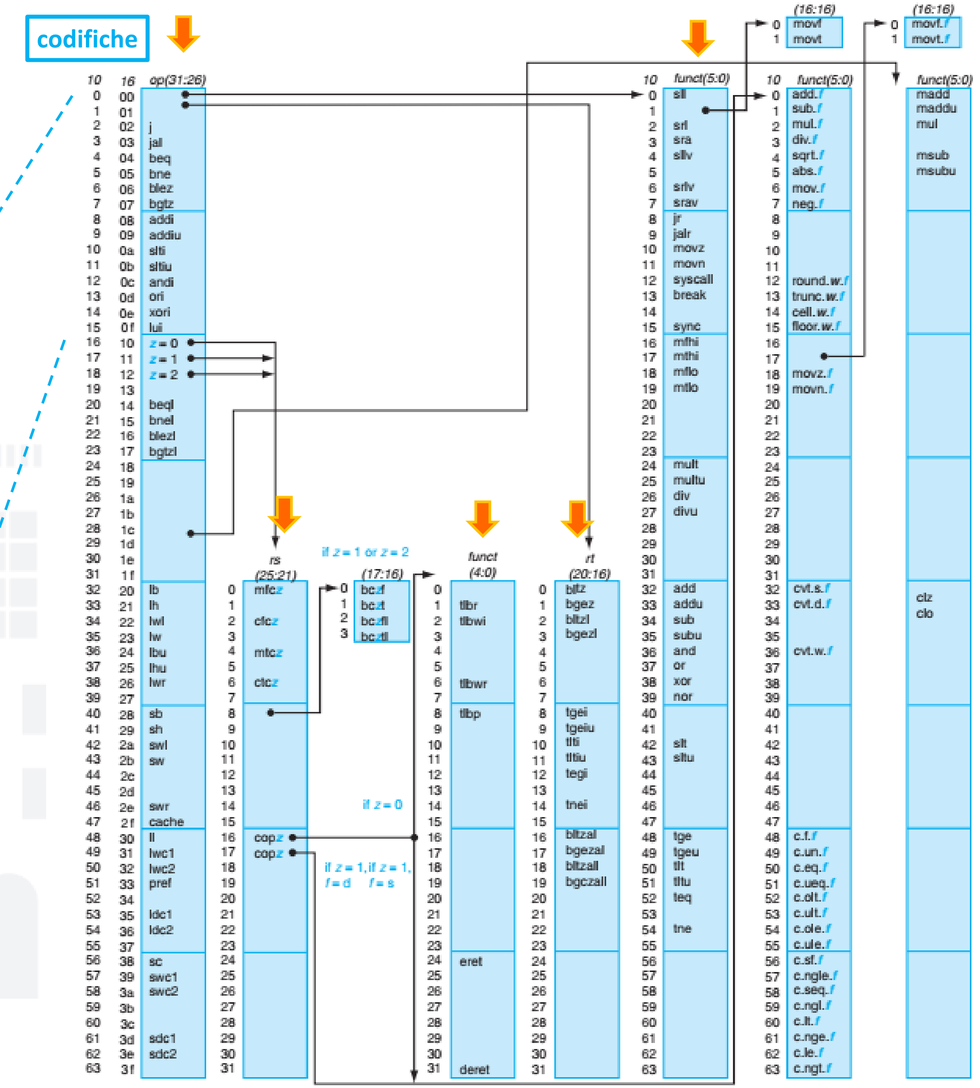
*Per esempio, ISA deve sapere che **ci sono solo 32 registri nel processore** e che le operazioni da poter fare sono solo con **un range di valori ottenibili con 32 bit**..*



INSTRUCTION SET ARCHITECTURE: OPCODE PER MIPS32

In MIPS32, si possono realizzare «solo» le operazioni codificabili tramite questa **MAPPA DEGLI OPCODE**.

00000001000010010101000000100000



Appendice del libro di testo (pagina A-50)

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Tre formati delle istruzioni: R-type, I-type, J-type

Attenzione: i tre formati non coincidono esattamente con le tre tipologie di operazioni

FORMATO R-TYPE

Per codificare operazioni aritmetico-logiche tra registri.

R-type = register-type

opcode	rs	rt	rd	shamt	funct

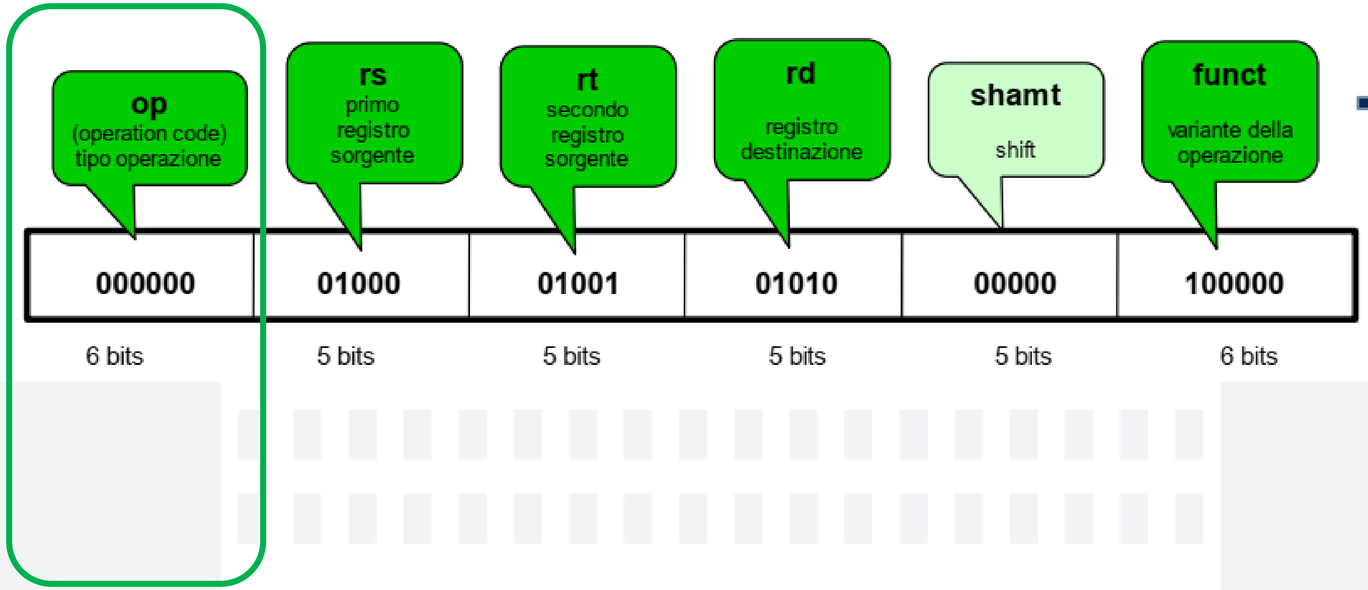
opcode = 000000 → indica che è un'**istruzione aritmetico-logica**.
rs = registro sorgente 1.
rt = registro sorgente 2.
rd = registro di destinazione.
shamt = valore di shift (solo per istruzioni di shift, altrimenti è 00000).
funct = specifica l'operazione esatta.

Operazioni aritmetiche con valori dai registri
Operazioni logiche
Operazioni di confronto (es. minore o uguale)

FORMATO R-TYPE: ESEMPIO

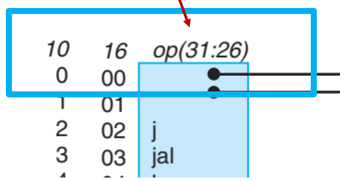
00000001000010010101000000100000

"somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10"



L'Unità di Controllo "sa" come interpretare i singoli campi

Nota: bit dell'opcode sono dal 26 al 31 (MSB) della sequenza. Il primo bit ha indice 0.



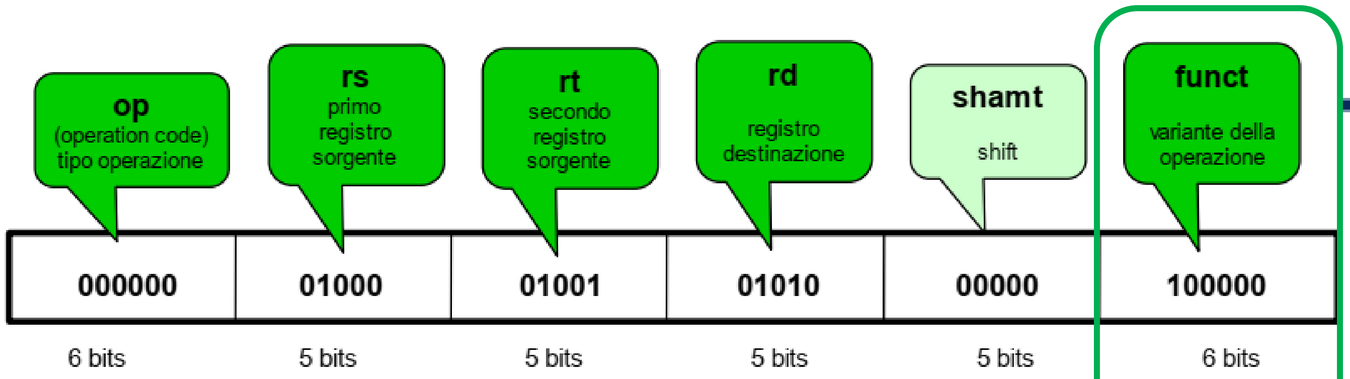
Appendice A p.50

opcode = 000000 → indica che è un'**istruzione aritmetico-logica**

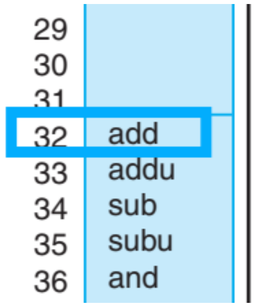
FORMATO R-TYPE: ESEMPIO

00000001000010010101000000100000

"somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10"



L'Unità di Controllo "sa" come interpretare i singoli campi



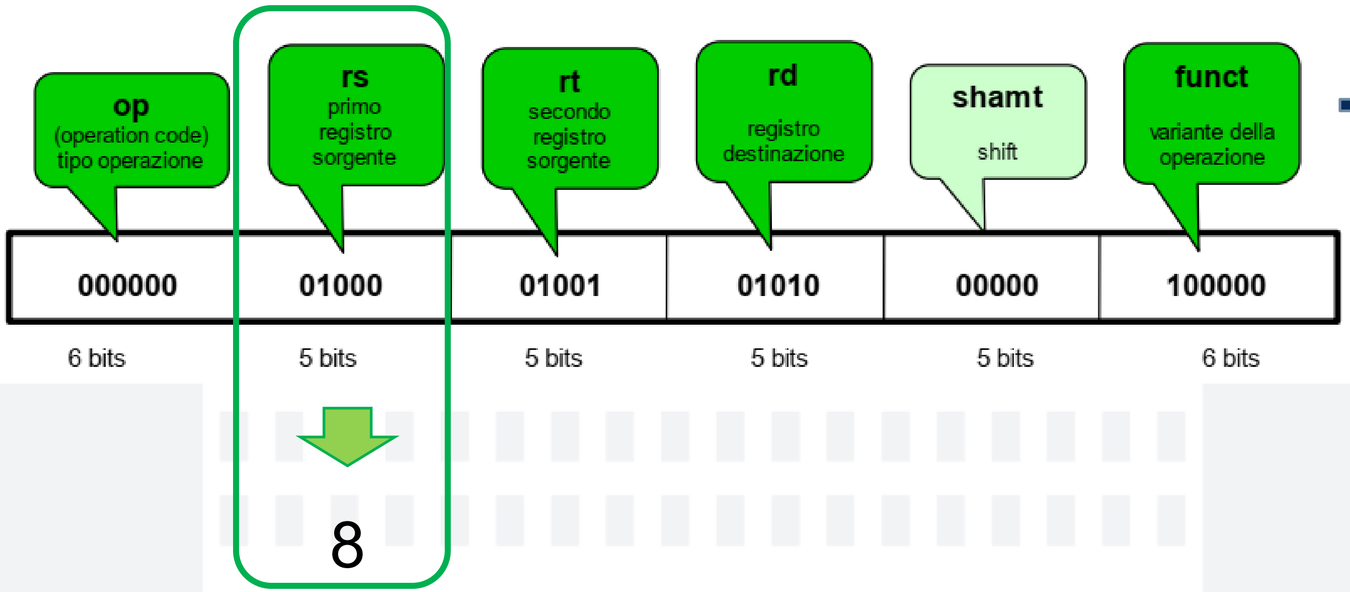
opcode = 000000 → indica che è un'**istruzione aritmetico-logica**

Leggere e decodificare il campo **funct** per determinare l'operazione esatta da eseguire

FORMATO R-TYPE: ESEMPIO

00000001000010010101000000100000

“somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10”



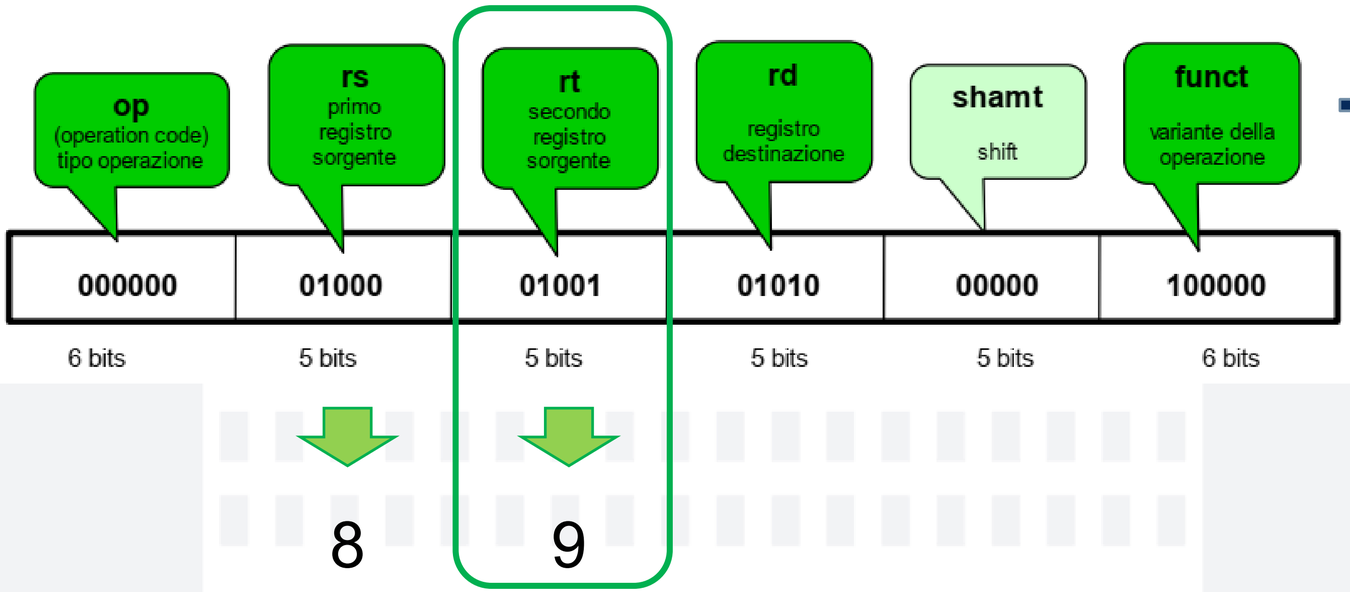
L’Unità di Controllo “sa” come interpretare i singoli campi

UNIVERSITÀ DEGLI STUDI DI TRIESTE

FORMATO R-TYPE: ESEMPIO

00000001000**01001**01010000000100000

“somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10”



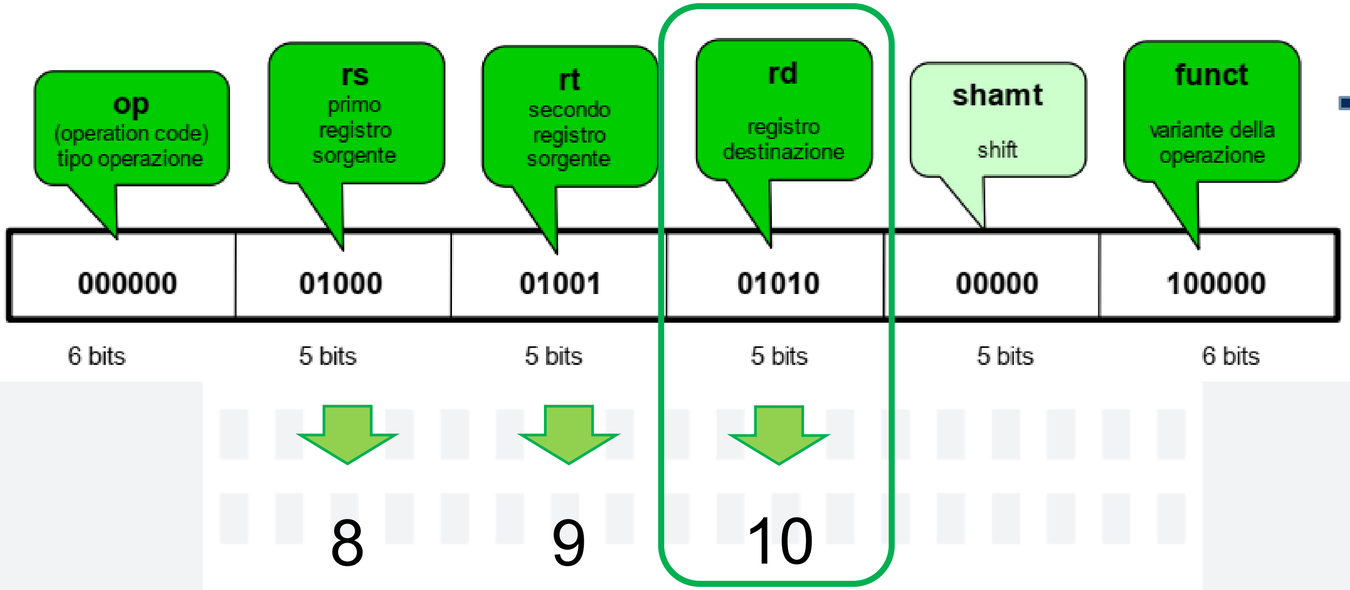
L’Unità di Controllo “sa” come interpretare i singoli campi

UNIVERSITÀ DEGLI STUDI DI TRIESTE

FORMATO R-TYPE: ESEMPIO

00000001000010010101000000100000

“somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10”



L’Unità di Controllo “sa” come interpretare i singoli campi

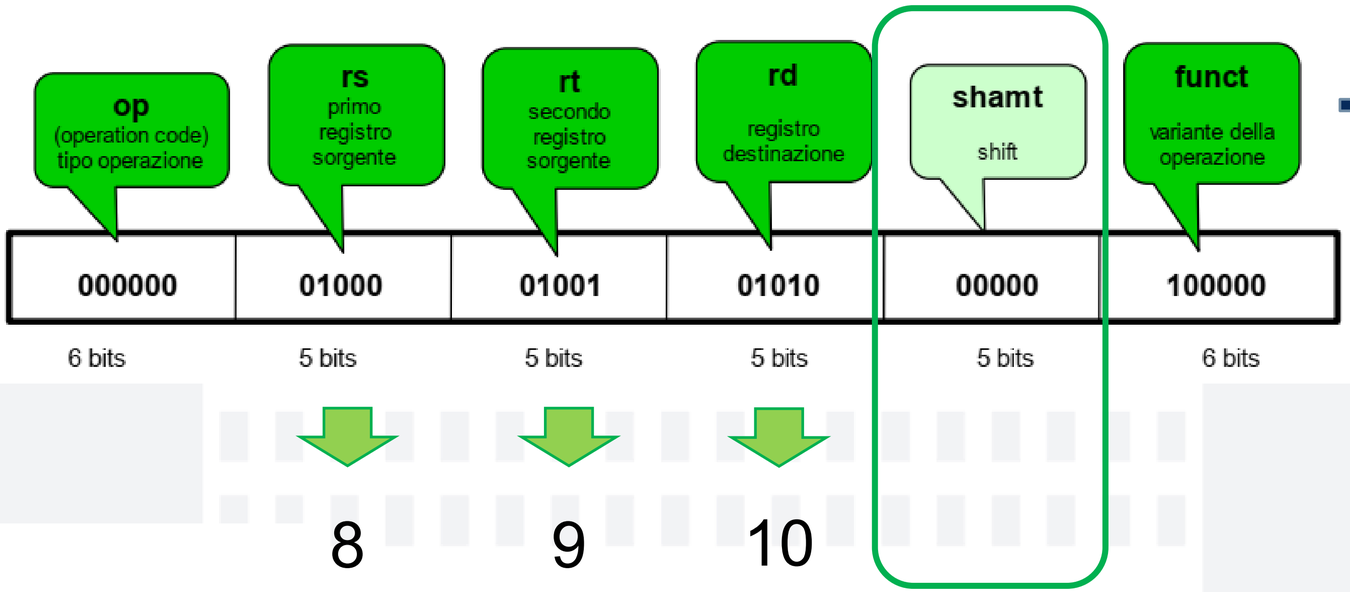
UNIVERSITÀ DEGLI STUDI DI TRIESTE

FORMATO R-TYPE: ESEMPIO

Nota. L'istruzione potrebbe essere anche scritta, in modo più compatto, in esadecimale: 0x1095020

00000001000010010101000000100000

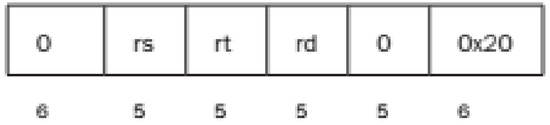
"somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10"



L'Unità di Controllo "sa" come interpretare i singoli campi

Addition (with overflow)

add rd, rs, rt



Appendice A p.51

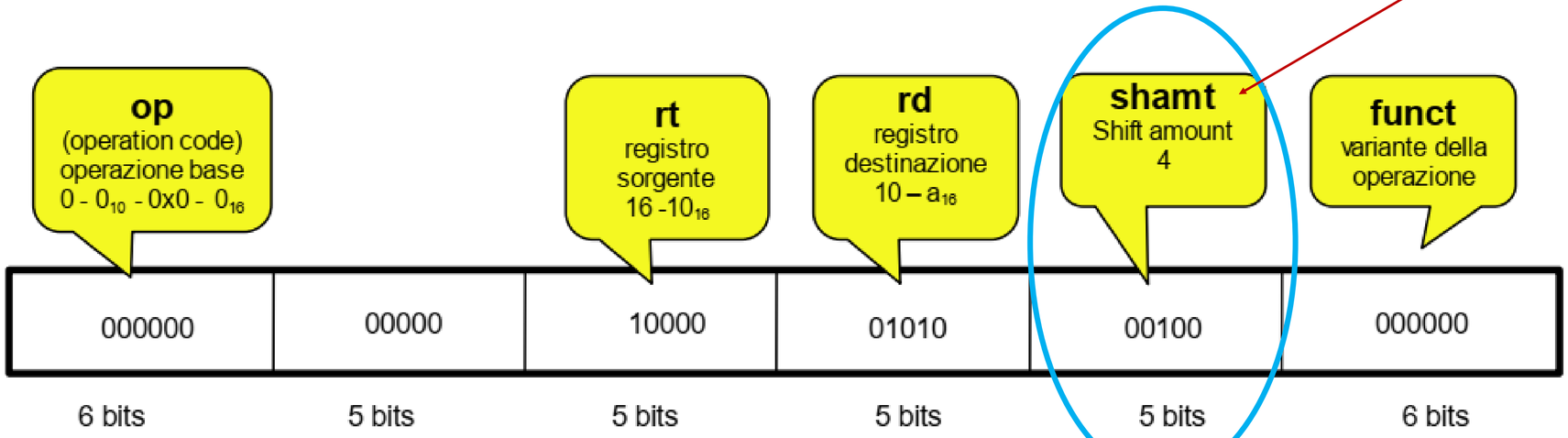
add $10, $8, $9

UNIVERSITÀ DEGLI STUDI DI TRIESTE

FORMATO R-TYPE: SHIFT LEFT

Shift di 4 bit a sinistra il contenuto del registro 16 e metti il risultato nel registro 10.

Shamt: shift amount



Tutto a zero **Tutto a zero** **Tutto a zero**

$16 $10 4

Shift left logical

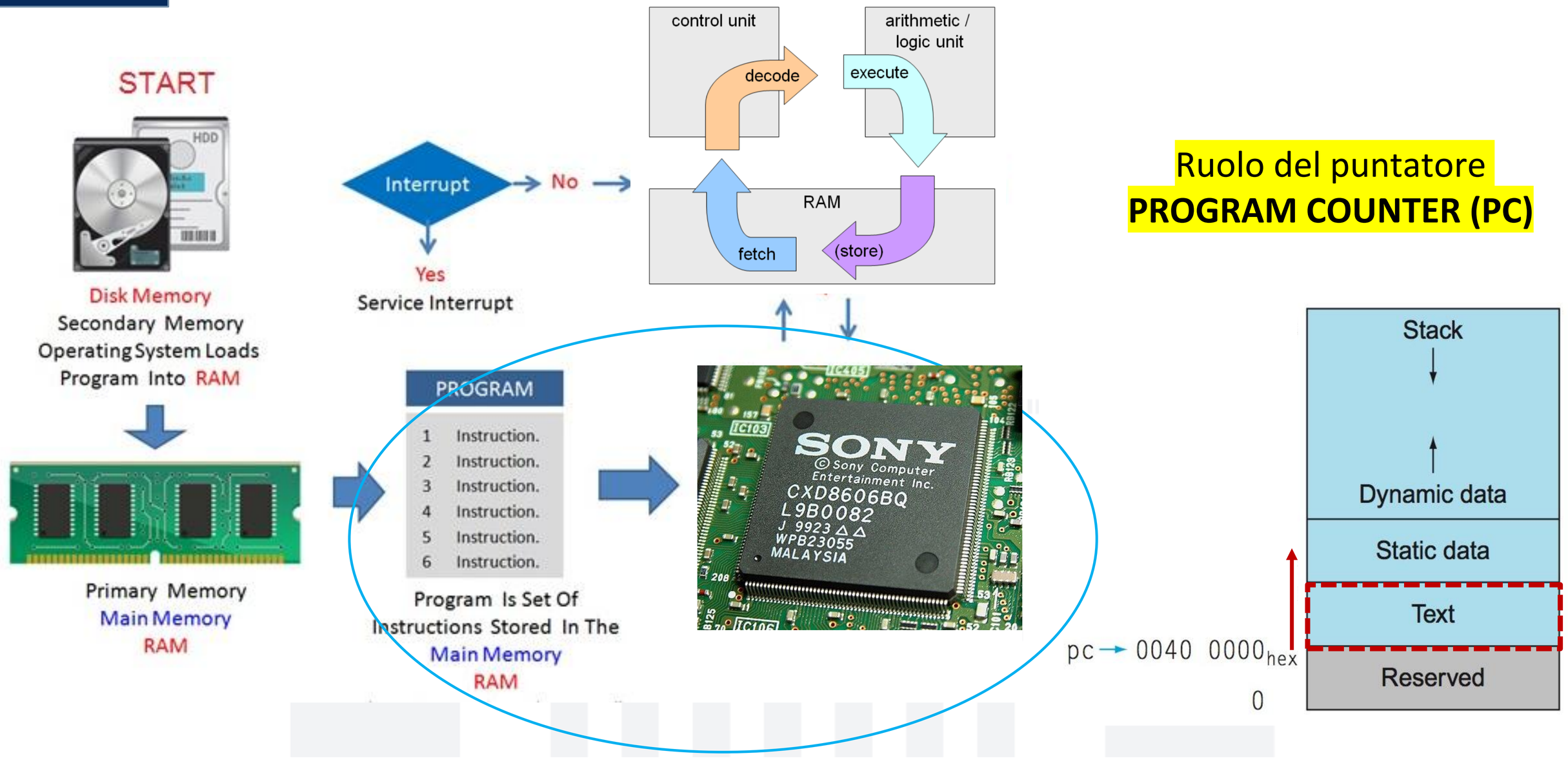
`sll rd, rt, shamt`

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

Appendice A p.55

Shift Left Logical **sll $10, $16, 4**

UNIVERSITÀ DEGLI STUDI DI TRIESTE



Ruolo del puntatore
PROGRAM COUNTER (PC)

Dopo che un'istruzione viene letta (*fetch*), **il PC viene automaticamente incrementato di 4**, passando all'istruzione successiva.

FORMATO J-TYPE

Per codificare operazioni di salto (assoluto) a indirizzi specifici

J-type = jump-type

opcode	Target address

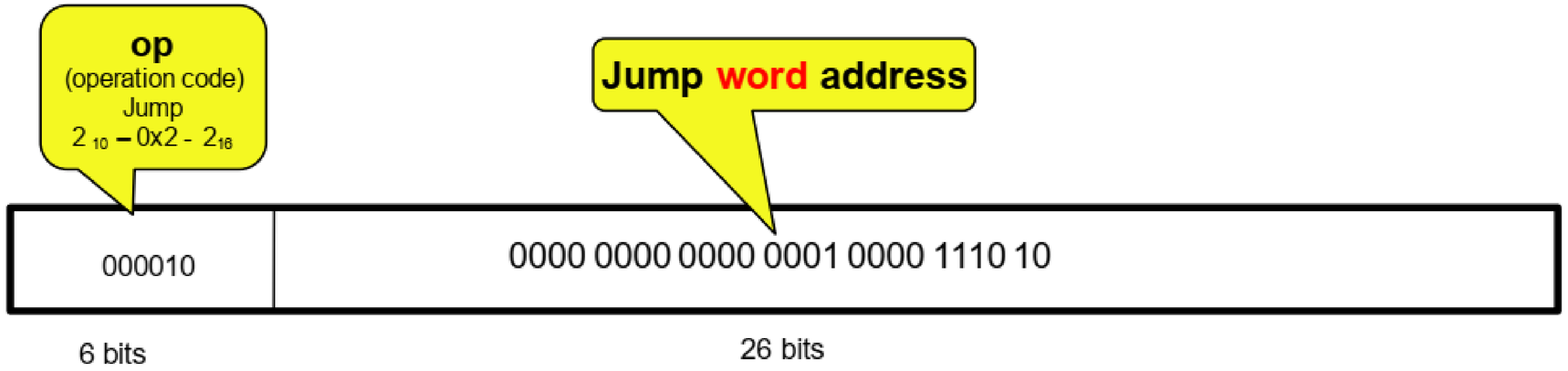
opcode (6 bit) → Indica che è istruzione di salto

Target address (26 bit) → Specifica l'indirizzo di destinazione

Salti assoluti.

FORMATO J-TYPE: ESEMPIO

00001000000000000000010000111010



jump

Jump

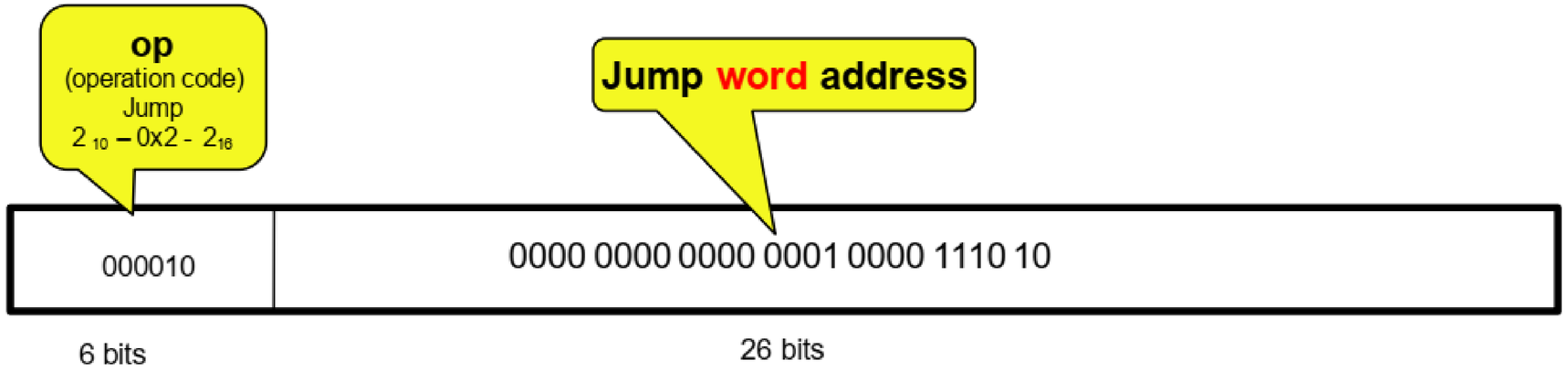
`j target`

2	target
6	26

Unconditionally jump to the instruction at target.

L'**indirizzo target** in un'istruzione J-type non è l'indirizzo completo, ma solo i **26 bit centrali**.

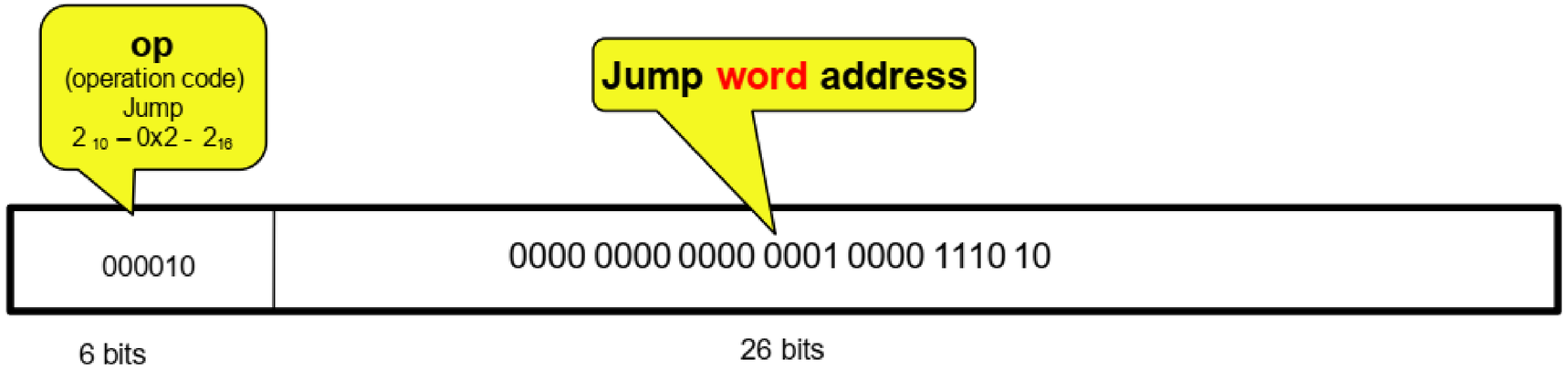
FORMATO J-TYPE: ESEMPIO



L'**indirizzo target** in un'istruzione J-type *non* è l'indirizzo completo, ma solo i **26 bit centrali**.

Per ottenere l'indirizzo effettivo bisogna **«calcolarlo»**.

FORMATO J-TYPE: ESEMPIO



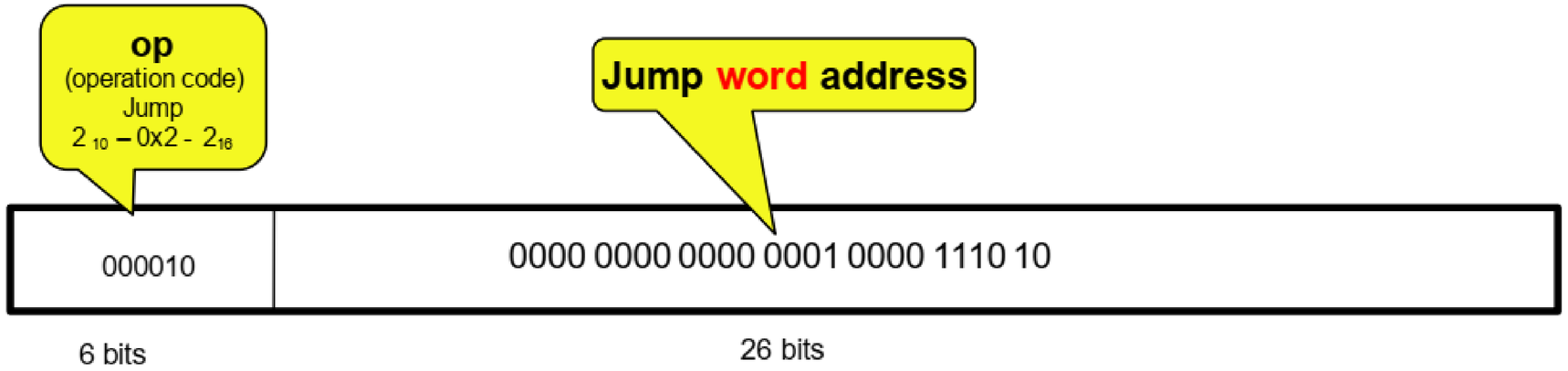
1.Shiftiamo il valore a sinistra di 2 bit (perché gli indirizzi MIPS sono allineati a 4 byte).

Codifica hex del target address: 0x00001E3A

Indirizzo finale di salto: 0x00001E3A << 2 = 0x000078E8

Nota. Ricordate che shiftare a sinistra di 2 bit equivale a **moltiplicare il numero per 4** (poiché 2^2=4).
In effetti: 0x**00001E3A** = **7738**$_{10}$
7738$_{10}$ x 4 = 30952$_{10}$ = 0x**000078E8**

FORMATO J-TYPE: ESEMPIO



2. Aggiungiamo i 4 bit più significativi del PC corrente

Leggo il PC(*) 0000 0000 0100 0000 0000 0000 0011 0100

Compongo l'indirizzo finale 0000 0000 0000 0000 0001 0000 1110 1000

Carico in PC l'indirizzo: **0000** 0000 0000 0000 0001 0000 1110 1000 = $000010E8_{16}$

Invece che eseguire l'istruzione che era prevista (*), verrà eseguita l'istruzione contenuta all'indirizzo $000010E8_{16}$

Materiale per la lezione

- Patterson & Hennessy, Cap.2
- Patterson & Hennessy, Cap.5
- Appendix A (***da*** pagina 49)

Prossima lezione: 10 ***marzo, h.14:00****, aula 4C*