UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Memoria e ISA

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 20 marzo 2025

Architettura MIPS32

32 registri, 32 bit per ogni cella di memoria, ecc.

add $10, $8, $9 ➡ 00000001000010010101000000100000

QtSpim

FORMATO R-TYPE

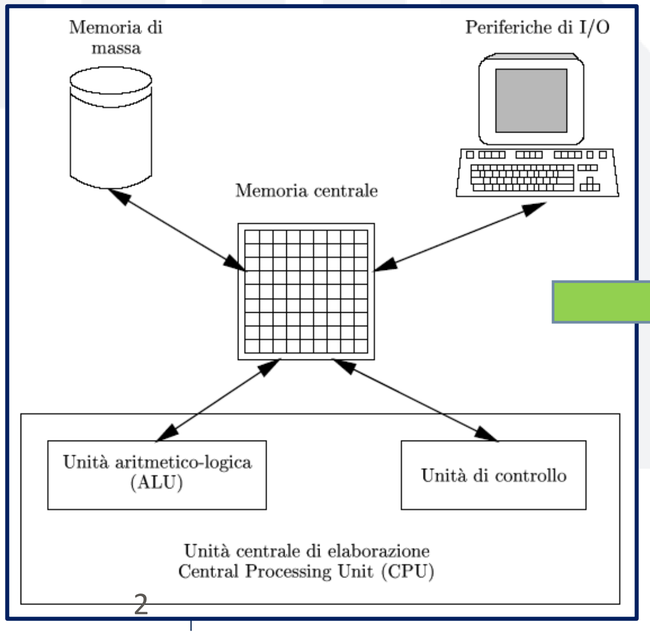
opcode	rs	rt	rd	shamt	funct

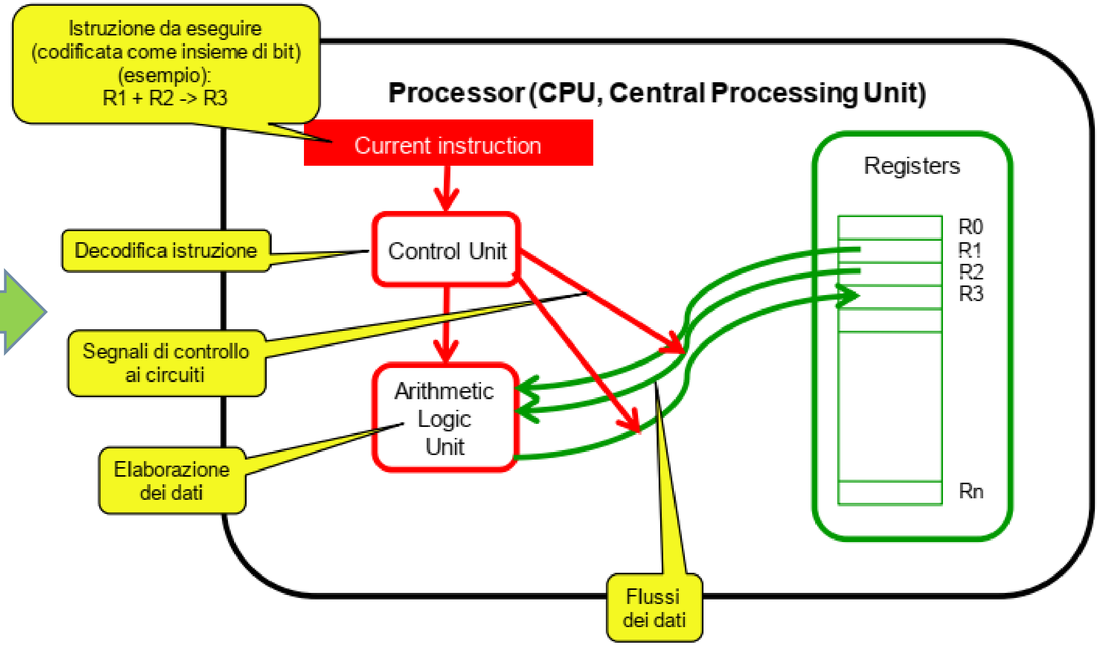
FORMATO J-TYPE

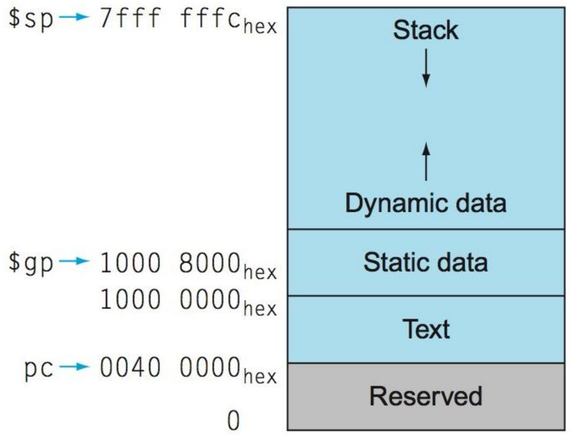
opcode	Target address

FORMATO I-TYPE

opcode	rs	rt	Immediate (offset)





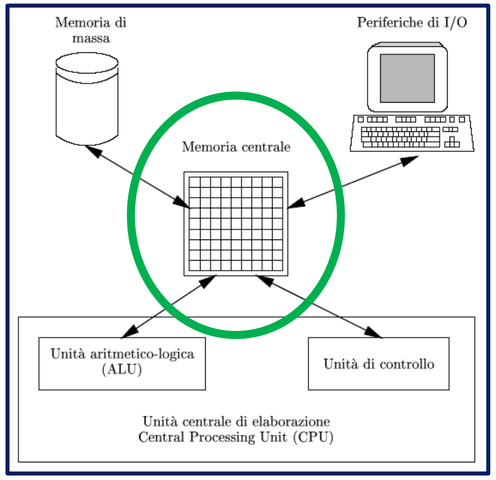


UNIVERSITÀ DEGLI STUDI DI TRIESTE

AGENDA DI OGGI/DOMANI

- Organizzazione della memoria centrale
- Istruction Set Architecture (ISA)
- Catena programmativa

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	Casa* Alex: civico 0	
0	0	1	1	1	0	0	1	5	0x00400001	Casa Barbara: civico 1	
1	1	1	1	0	0	1	0	6	0x00400002	Casa Leo: civico 2	
0	0	1	1	0	0	1	1	7	0x00400003	Casa Penelope: civico 3	
1	0	0	0	1	1	1	1	**8**		:	
0	1	1	0	0	1	0	0	9			

*Tutte le case sono della stessa grandezza (8 bit)

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

UNIVERSITÀ DEGLI STUDI DI TRIESTE

MEMORIA RAM

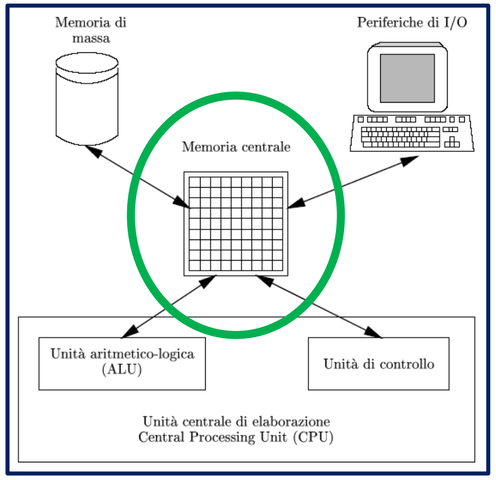
Però in realtà PER CONVENZIONE e per semplificare l'architettura, si segue un **ALLINEAMENTO A WORD**

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



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

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

UNIVERSITÀ DEGLI STUDI DI TRIESTE

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

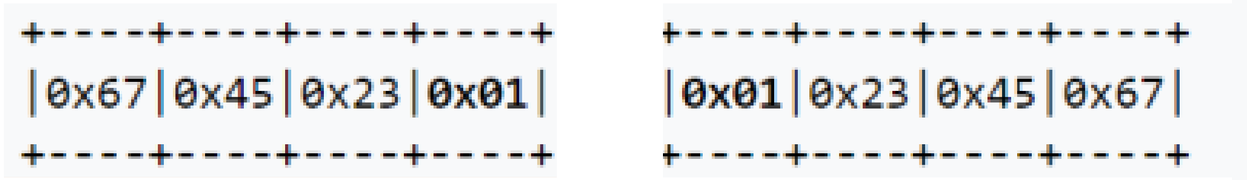
BIG-ENDIAN VS LITTLE-ENDIAN

Byte #			
3	2	1	0

Little-endian

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

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

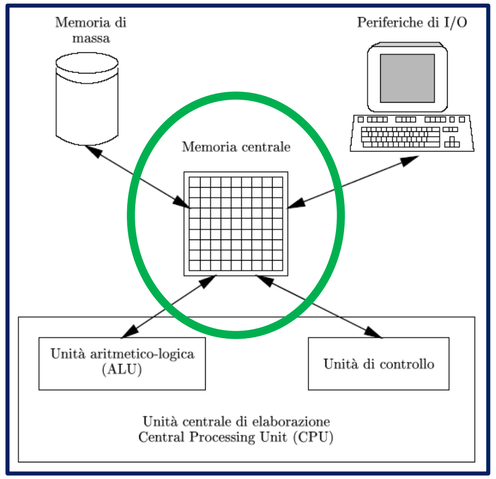


Il byte di indice più piccolo può essere il byte più a sinistra (***big-endian***) o il più a destra (***little-endian***).

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: CAPIENZA



MIPS32 è un'architettura a 32 bit, dunque può indirizzare al massimo:

2^{32} bytes = 4 294 967 296 bytes = **4 GB**

Quindi, teoricamente, un processore MIPS32 potrebbe accedere a **fino a 4 GB** di RAM.

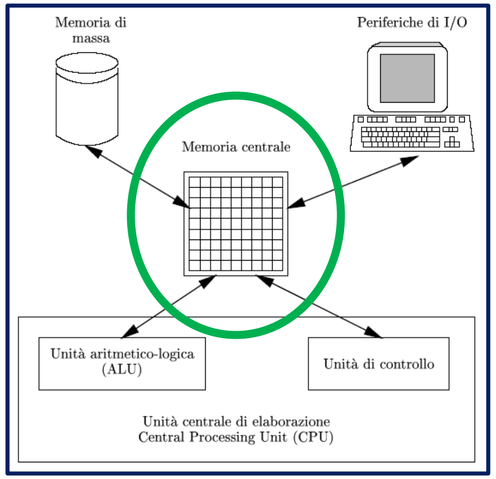
Nella pratica:

- Raramente vengono raggiunti i 4 GB massimi teorici per ragioni di costo e design.
- Sistemi più avanzati basati su MIPS32 possono avere decine o centinaia di megabyte di RAM.
- Esiste anche MIPS64, quindi può supportare memoria ben superiore a 4 GB (ARM64 o Intel x86-64).
- Se un dispositivo MIPS deve gestire grandi quantità di memoria (>4 GB), utilizzerà MIPS64, non MIPS32.
- Oggi MIPS (32 bit) non è tipicamente usata per computer desktop o smartphone consumer, ma è comune in ambito embedded, dove i requisiti di RAM sono bassi e la semplicità dell'architettura è un grande vantaggio.

Alcuni esempi reali:

- Router e dispositivi di rete basati su MIPS spesso hanno memoria nell'ordine di decine o centinaia di MB.
- Dispositivi didattici (come emulatori o simulatori MIPS usati per apprendimento) possono avere pochi megabyte.
- Dispositivi IoT (Internet of Things)
- Set-top box (ricevitori satellitari, decoder TV)
- Sistemi embedded industriali

MEMORIA RAM: UTILIZZI



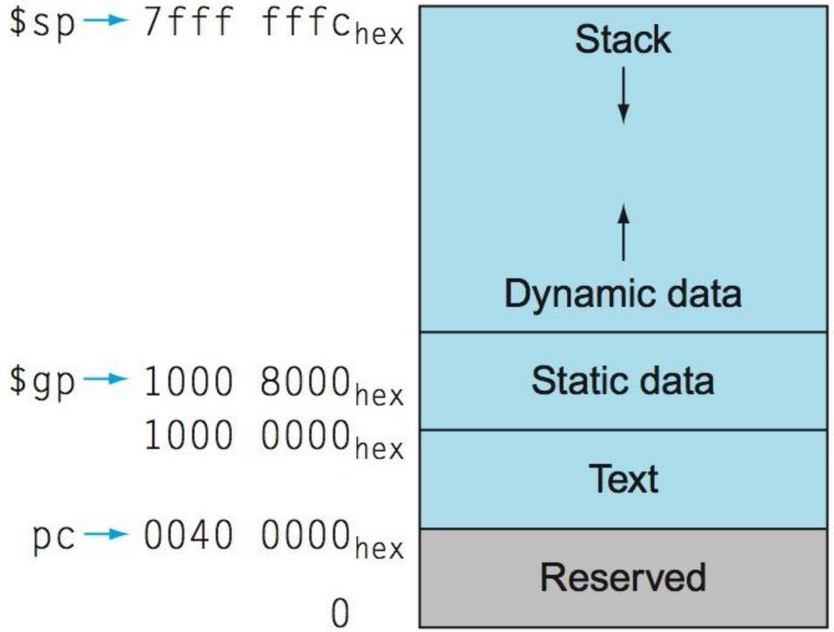
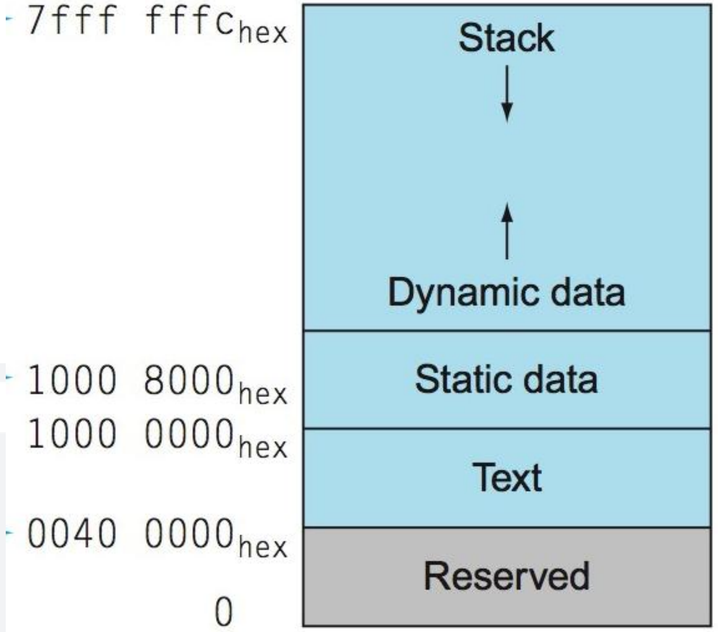
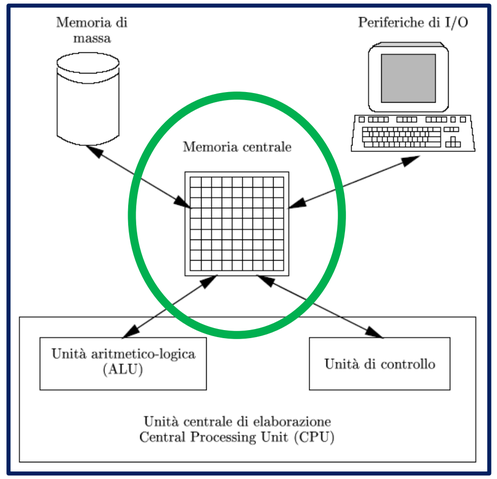


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 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.

MEMORIA RAM: UTILIZZI





Segmento testo (Text segment):

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

Segmento dati (Data segment):

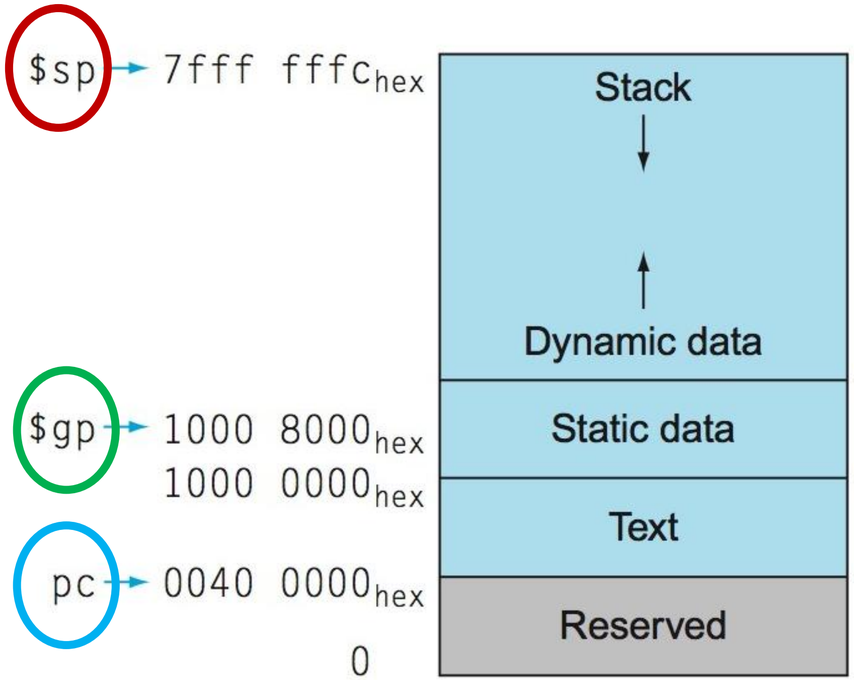
- Indirizzi da **0x10010000** a circa **0x7FFFFFFF**.
- Può utilizzare fino a circa **2 GB** per i dati.

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

KERNEL (per simulare 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: UTILIZZI



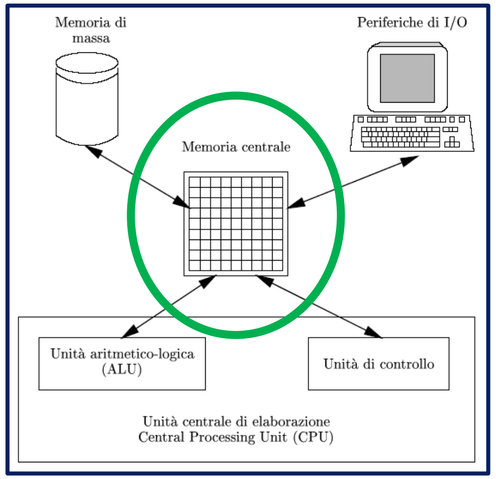
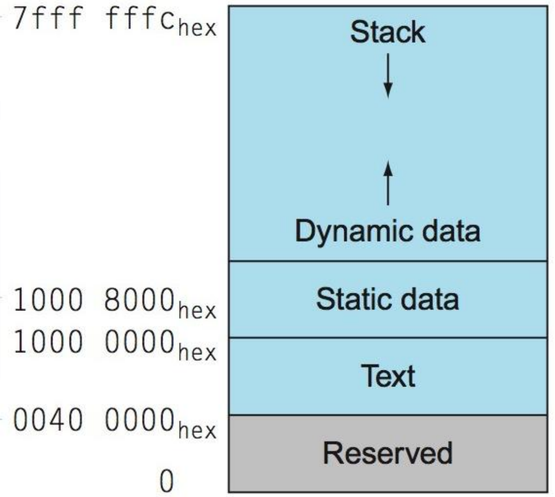


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 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.).

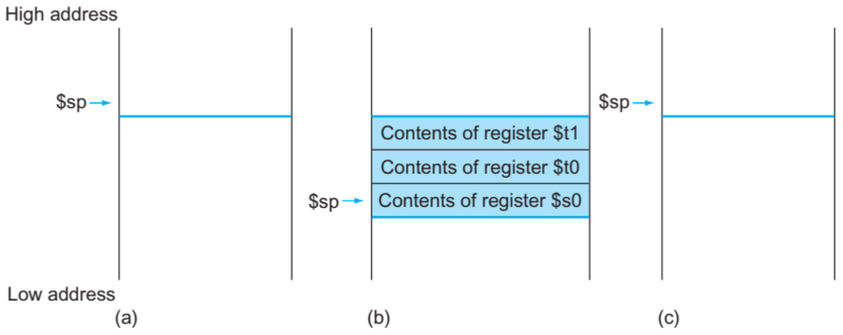
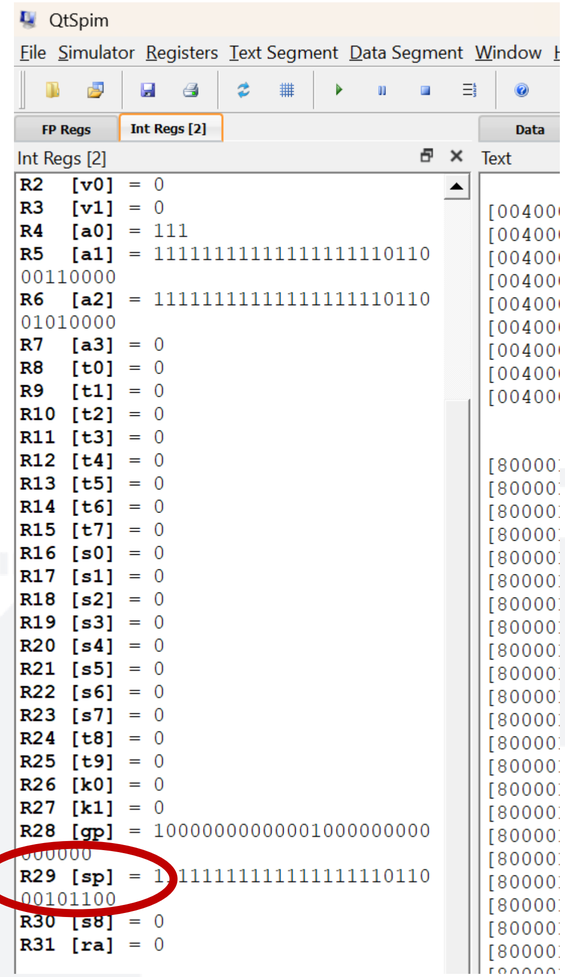


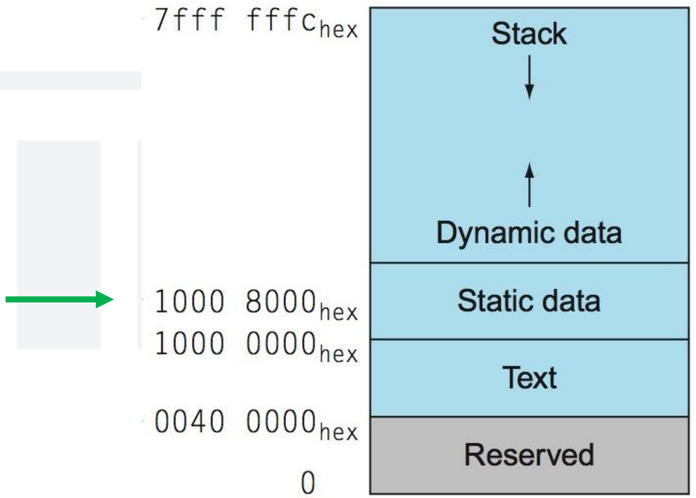
FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.



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

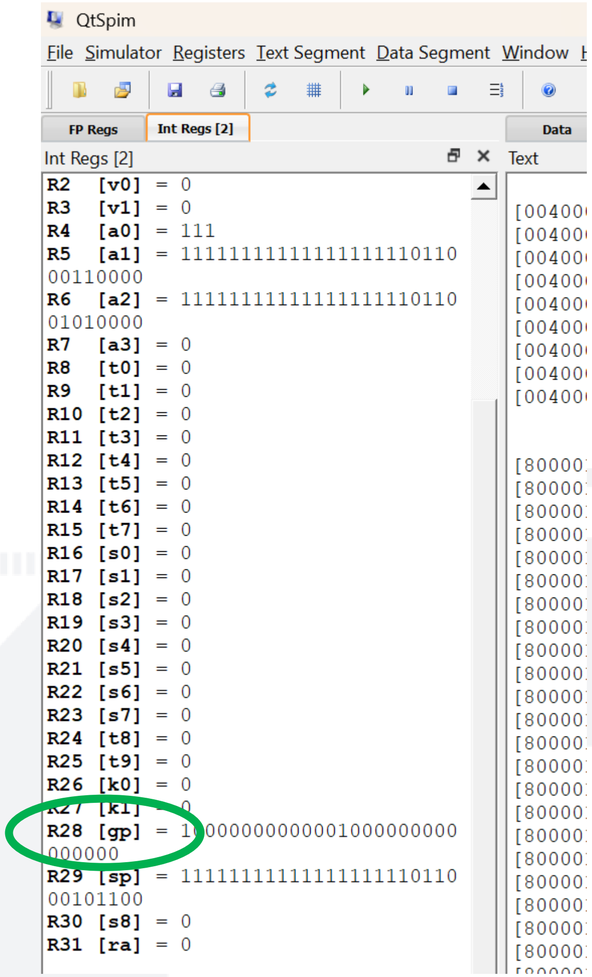
GLOBAL POINTER ($gp)

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.

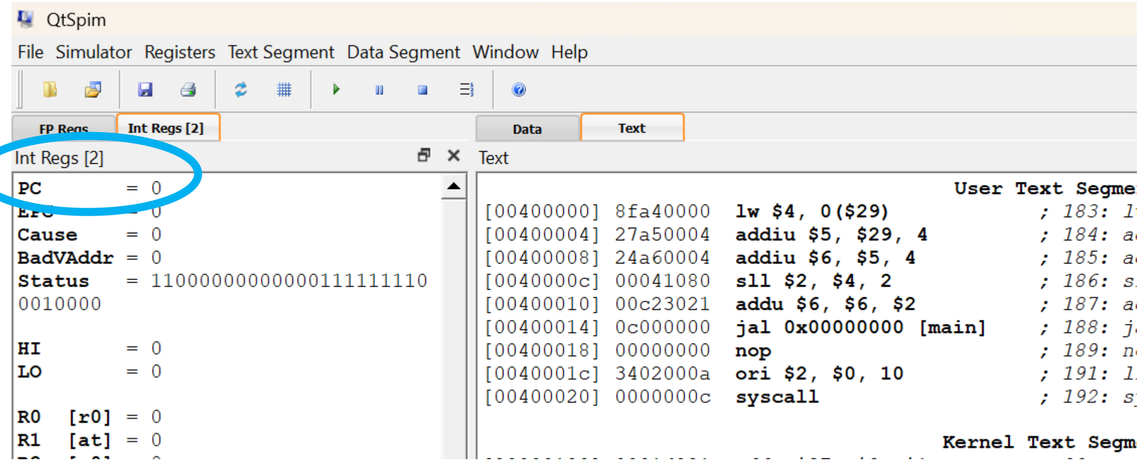


Accesso alle variabili globali più efficiente rispetto all'indirizzamento assoluto.

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.

Per provare ad eseguire questo programma come mostrato in classe, copiare e incollare il programma su un file di testo e poi salvarlo nel formato .asm. Poi andare su **QtSpim** → File → Load e caricare il programma. Eseguirlo: tutto (▶) oppure passo passo (simbolo lista).

PROGRAMMA (1/2)

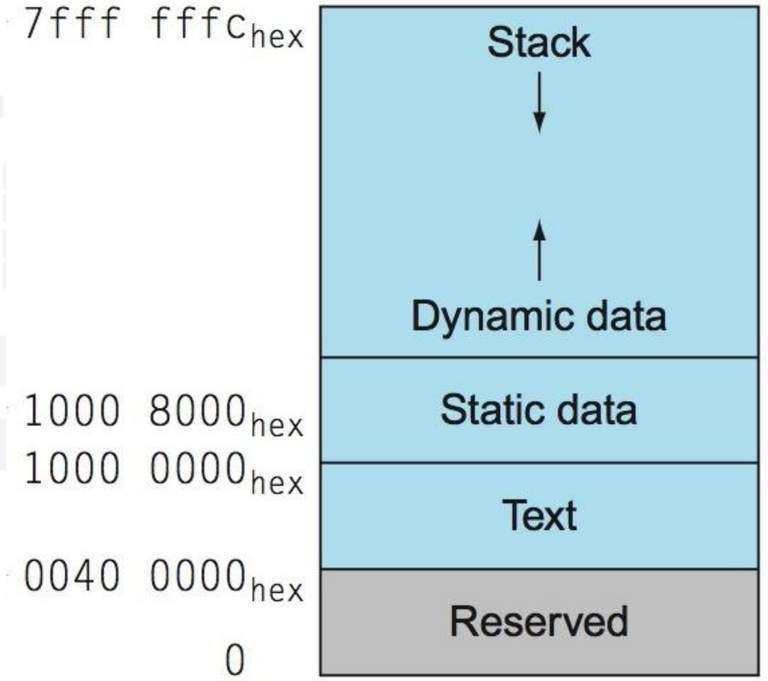
Un programma è un testo che contiene una lista di istruzioni da eseguire.

Esempio: si carichi in memoria un vettore di 5 interi e si faccia la somma del primo e del terzo valore, restituendo la somma in un registro ($t3).

```
.data
vet: .word 2, 4, 6, 8, 10  # vettore di 5 interi

.text
.globl main

main:
   la $t0, vet            # Carica in $t0 l'indirizzo base del vettore
   lw $t1, 0($t0)         # Carica primo elemento (vet[0]=2) in $t1
   lw $t2, 8($t0)         # Carica terzo elemento (vet[2]=6) in $t2
   add $t3, $t1, $t2      # Somma $t1+$t2 → risultato in $t3
                          # risultato finale in $t3 (=8)
```



UNIVERSITÀ DEGLI STUDI DI TRIESTE

PROGRAMMA (2/2)

(quasi) Ogni istruzione del programma viene tradotta in un'istruzione in linguaggio macchina.

Ci sono delle **eccezioni**:

La direttiva .data indica che il testo seguente si riferisce al segmento di memoria dedicata ai dati.

La direttiva .text indica che si scrive nella porzione di memoria dedicata al codice.

Esistono le pseudo-istruzioni: sono istruzioni che noi scriviamo con un'unica riga di codice, ma che la macchina non può eseguire tutta in un'unica volta (ad esempio: saltare ad un indirizzo di 32 bit).
Quindi, al momento del caricamento del programma, **il sistema converte automaticamente la nostra riga di codice in due istruzioni macchina.**

la $t0, vet ➡ **lui $t0, 0x1001**
ori $t0, $t0, 0x0000

la (load address) è una pseudo-istruzione (comodità per il programmatore).
lui (load upper immediate) è la reale istruzione MIPS che viene generata, insieme a **ori**, da QtSpim.

INDIRIZZAMENTO

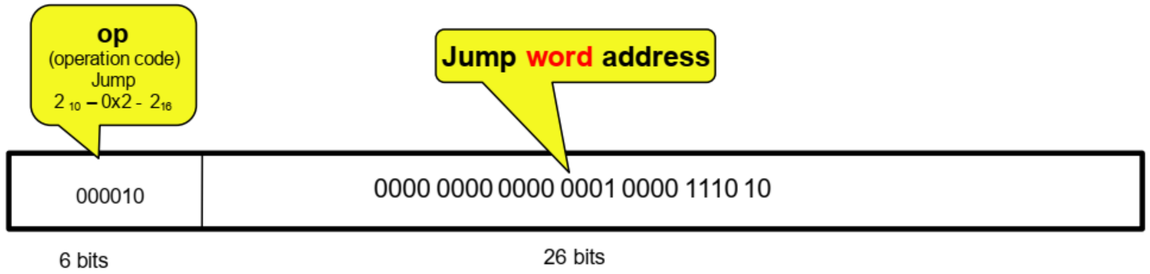
Si riferisce a tutte le volte che dobbiamo indicare al programma dove prendere la nuova istruzione da eseguire o dove prendere un dato che serve.

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

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

Problema: quanti bit occorrono?

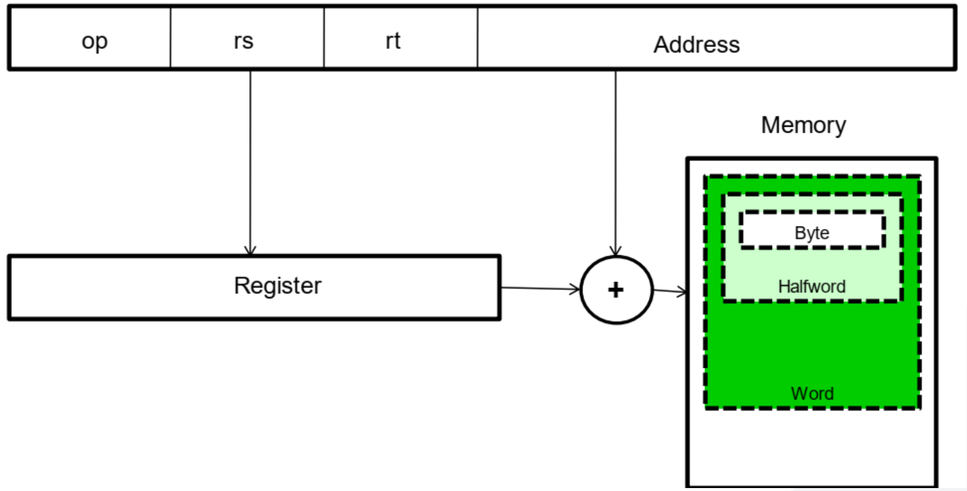
- La memoria è grande (MIPS: 2^{32} locazioni)
- Per specificare un indirizzo occorrono 32 bit …
- …che non ci stanno in un'istruzione di 32 bit



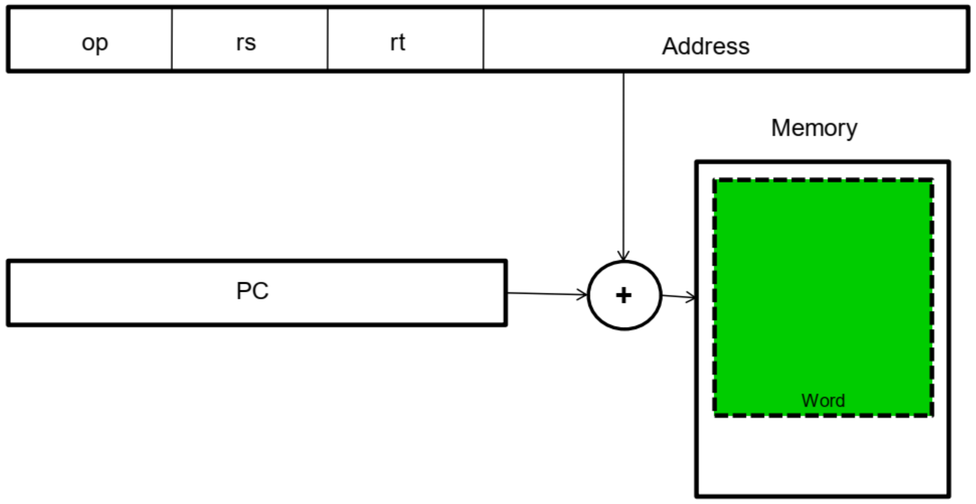
Dobbiamo trovare un modo semplice, univoco ed efficiente per ovviare a questo problema.

INDIRIZZAMENTI (GENERALE)

Base addressing



PC addressing



FORMATO J-TYPE

Per codificare operazioni di salto (assoluto) a indirizzi specifici

J-type = jump-type

opcode	Target address

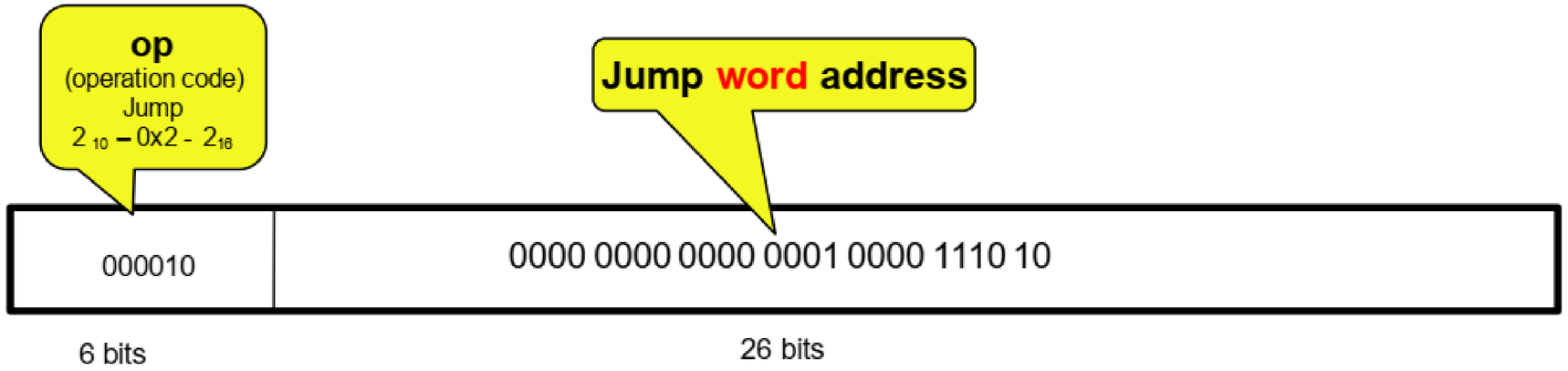
opcode (6 bit) → Indica che è istruzione di salto

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

Salti assoluti.

FORMATO J-TYPE: ESEMPIO

00001000000000000000010000111010



jump

Jump

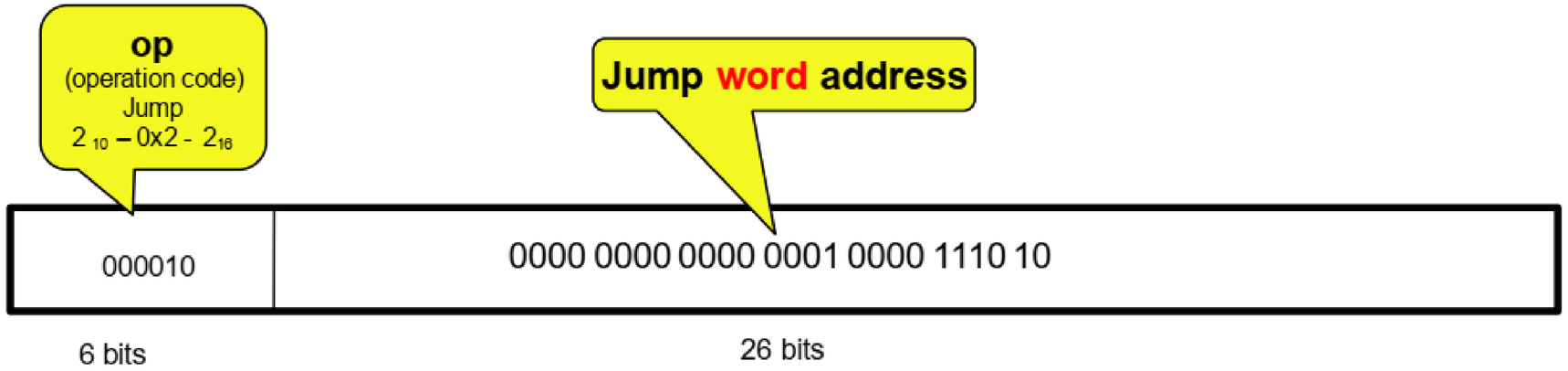
`j target`

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**.

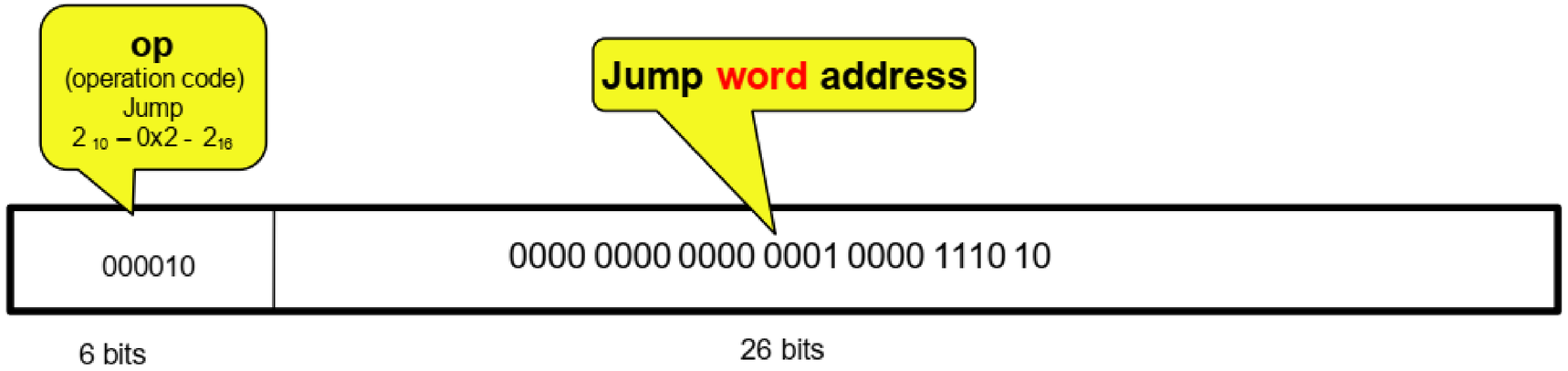
FORMATO J-TYPE: ESEMPIO



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

FORMATO J-TYPE: ESEMPIO



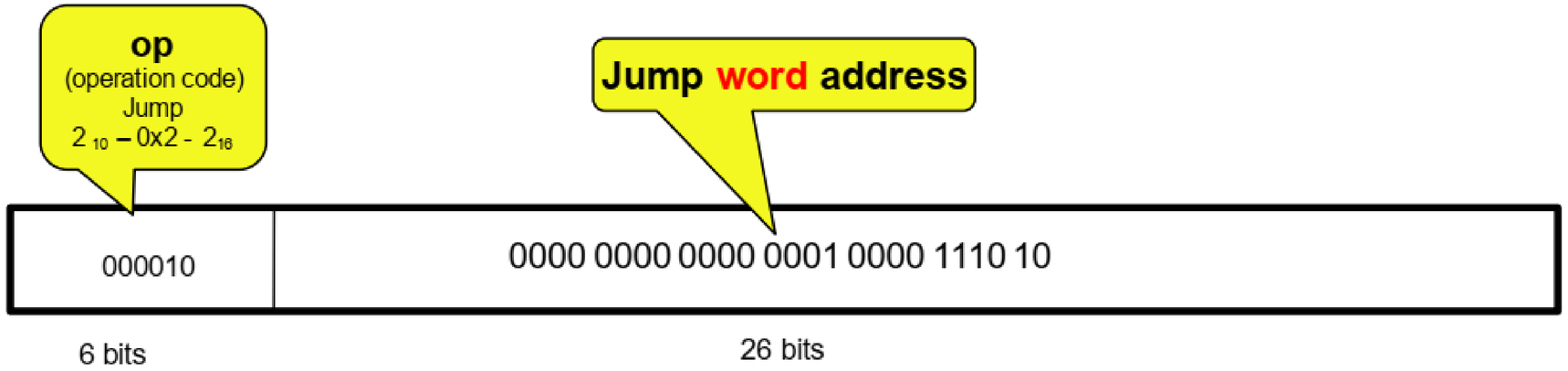
1.Shiftiamo il valore a sinistra di 2 bit (perché gli indirizzi MIPS sono allineati a 4 byte).

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 $000010E8_{16}$

FORMATO J-TYPE: ESEMPIO

Quindi **con 26 *bit*** si indirizzano **2^{28} *byte* di memoria** programma oppure 2^{26} *word*

Il range di indirizzi a cui posso arrivare in questo modo di indirizzamento (relativo al PC) sono tutti quelli del tipo XXXX 0000 0000 0000 0001 0000 1110 10**00**

26 bit

26 bit → 2^{26} configurazioni → Ciascuna di esse indirizza 1 byte ogni 4, ovvero l'inizio di word consecutive

→ 2^{28} configurazioni, se lascio variare anche gli ultimi due bit(LSB) → 2^{28} indirizzi (se ammetto anche gli indirizzi non multipli di 4 byte)

Materiale per la lezione

- Appendix A
- Capitolo 2

Prossima lezione: 21 ***marzo, h.11:00****,* ***aula 5B***