UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Catena programmativa e Assembly

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 27 marzo 2025

RECAP

Tipica struttura programma Assembly

```
# Title:                                Filename:
# Author:                               Date:
# Description:
# Input:
# Output:
################# Data segment #####################
.data
 . . .
################# Code segment #####################
.text
.globl main
main:                                   # main program entry
 . . .
li $v0, 10                              # Exit program
syscall
```

Nomi riservati

Nome Simbolico	Numero	Uso
$zero	0	Costante 0
$at	1	Assembler temporary
$v0-$v1	2-3	Functions and expressions evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Direttive all'assemblatore : .data, .text, .space, .ascii/.asciiz, …

Istruzioni per operazioni aritmetiche e logiche:

```
add rd, rs, rt
addi rd, rs, imm
and rd, rs, rt
:
```

Pseudo-istruzioni, as esempio: `li $s0, 0x12345678`

SYS-CALL: chiamata al sistema (operativo)

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

FIGURE A.9.1 System services.

ISTRUZIONI VARIE (Appendix A, da pagina A-51)

Istruzioni per operazioni aritmetiche

Istruzioni per operazioni logiche

Branch

Jump

Confronti

Manipolazione di costanti

Load/store

ISTRUZIONI VARIE (Appendix A, da pagina A-51)

Istruzioni per operazioni aritmetiche

Istruzioni per operazioni logiche

Branch

Jump

Confronti

Manipolazione di costanti

Load/store

FUNZIONI LOGICHE PRINCIPALI

AND o prodotto logico

A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

OR o somma logica

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

XOR (OR esclusivo)

A	B	A^B
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

NOT o negazione logica

A	$\overline{A}$
0	1
1	0

FUNZIONI LOGICHE DERIVATE

NAND

A	B	$A \cdot B$	A NAND B
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

NOR

A	B	$A + B$	A NOR B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

NOT o negazione logica

A	$\overline{A}$
0	1
1	0

FUNZIONI LOGICHE SU SEQUENZE DI BIT

E’ possibile applicare queste funzioni anche a sequenze di bit. Si procede al confronto bit per bit, su bit di ordine corrispondente nelle due sequenze

A = 11001100

B = 10101010

A = 11001100

AND B = 10101010

= 10001000 = 0x88

```
.text
main:
    li $t0, 0xCC          # A = 11001100
    li $t1, 0xAA          # B = 10101010
    and $t2, $t0, $t1     # $t2 = A AND B = 10001000
```

ISTRUZIONI VARIE (Appendix A, da pagina A-51)

Istruzioni per operazioni aritmetiche

Istruzioni per operazioni logiche

Branch

Jump

Confronti

Manipolazione di costanti

Load/store

AGENDA DI OGGI

- *Catena programmativa (recap)*
- *Assembly (con QtSpim)*
- **Catena programmativa**

CATENA PROGRAMMATIVA

Assembler, Linker

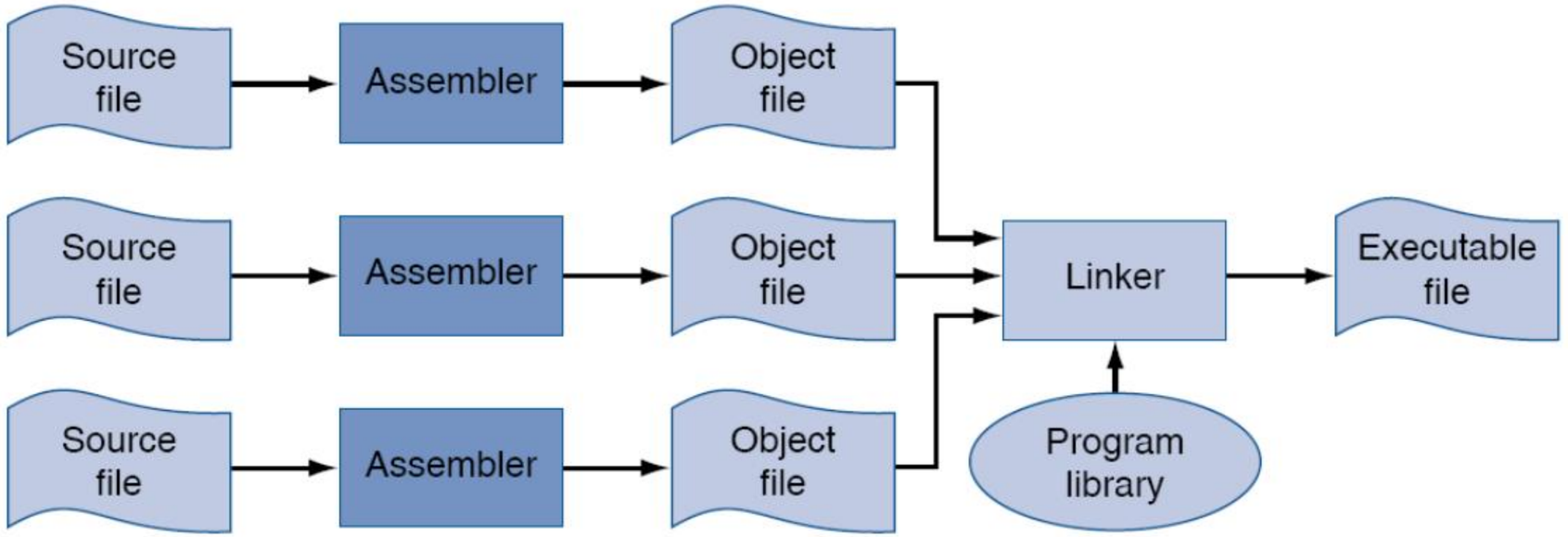


FIGURE A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

ASSEMBLER (1/3)

- Converte un programma assembler (*file sorgente*) in linguaggio macchina (*file oggetto*)
- Gestisce le etichette
- Gestisce pseudo-istruzioni
- Gestisce numeri in base diverse (e.g., binario, decimale, esadecimale)

L'assemblaggio è un procedimento sequenziale che esamina, riga per riga, il codice sorgente Assembly, traducendo ciascuna riga in un'istruzione del linguaggio macchina.

Applicato modulo per modulo al programma e costituisce per ogni modulo la **Tabella dei simboli** del modulo.

ASSEMBLER (2/3)

2 operazioni importanti:

1. Traduce i codici simbolici delle istruzioni nei corrispondenti codici binari (sequenze di 32 bit)
2. Traduce i riferimenti simbolici (variabili, registri, etichette di salto, parametri) nei corrispondenti indirizzi numerici.

Poichè le *etichette di salto* generano il problema dei riferimenti in avanti (ossia, riferimenti ad etichette successive o contenute in altri file), **l'assemblatore deve leggere il programma sorgente due volte**.

Ogni lettura del programma sorgente è chiamata **passo** e **l'assemblatore è chiamato *traduttore a due passi.***

ASSEMBLER (3/3)

Tabella dei simboli

Contiene i **riferimenti simbolici delle istruzioni** presenti nel modulo da tradurre e, al termine del primo passo, conterrà gli indirizzi numerici di tutti i simboli, tranne quelli esterni al modulo in esame.

Per le **etichette associate a direttive dell'assemblatore** che definiscono costanti simboliche, nella tabella dei simboli viene creata la coppia **< etichetta, valore >** e in ogni istruzione che fa riferimento al simbolo viene sostituito il valore.

Per **etichette che definiscono variabili** (spazio di memoria + eventuale inizializzazione), l'assemblatore riserva spazio, eventualmente inizializza la zona di memoria e crea nella tabella la coppia **<etichetta, indirizzo>**. In ogni istruzione che fa riferimento al simbolo (all'etichetta), il simbolo viene sostituito con l'indirizzo.

Nelle **etichette presenti nelle istruzioni di salto**, l'assemblatore deve generare un riferimento all'indirizzo dell'istruzione destinazione di salto.
Etichette interne (local) sono visibili solo all'interno di un modulo**.**

Etichette esterne (global) al modulo possono essere usate da moduli esterni**;** *rimangono non risolte.*
L'assember non è disturbato da questo aspetto. **Le etichette esterne saranno risolte dal linker.**

ASSEMBLAGGIO: ESEMPIO (1/3)

```
.data
array: .space 40        # Alloca spazio per 10 word (40 byte)
size:  .word 10         # Definisce una costante di valore 10

.text
main:
    li $t0, 0           # Inizializza $t0 a 0
loop:
    # Corpo del loop
    beq $t0, $t1, end # Se $t0 == $t1, salta a 'end'
    j loop              # Salta all'inizio del loop
end:
    # Fine del programma
```

- **array** è un'etichetta che rappresenta l'inizio di un'area di memoria allocata per un array.
- **size** è un'etichetta associata a una parola che contiene il valore costante 10.
- **main, loop** ed **end** sono etichette che identificano indirizzi specifici nel segmento di codice.

ASSEMBLAGGIO: ESEMPIO (2/3)

Tabella dei Simboli (va nel file oggetto)

Nome	Tipo	Sezione	Valore (indirizzo virtuale)
array	OBJECT	.data	0x10010000 (inizio area dati)
size	OBJECT	.data	0x10010028 (dopo 40 byte)
main	FUNCTION	.text	0x00400000
loop	LABEL	.text	0x00400004
end	LABEL	.text	0x0040000C

- **Nome**: Il nome simbolico definito nel codice assembly (es. etichette, variabili, costanti).
- **Tipo**: La categoria del simbolo, come etichetta (label), variabile o costante.
- **Sezione**: La sezione del programma in cui il simbolo è definito, tipicamente .text per il codice e .data per i dati.
- **Indirizzo/Valore**: L'indirizzo di memoria assegnato al simbolo o il suo valore, nel caso di una costante

ASSEMBLAGGIO: ESEMPIO (3/3)

Associazione istruzioni → codice macchina (.Text)

INDIRIZZO	**ISTRUZIONE ASSEMBLY**	**ISTRUZIONE BINARIA** (SEMPLIFICATA)
0x00400000	**li $t0, 0**	→ addi $t0, $zero, 0 (pseudoistruz.)
0x00400004	**beq $t0, $t1, end**	000100 01000 01001 0000 0000 0000 0001
0x00400008	**j loop**	000010 0000 0000 0000 0000 0000 0001
0x0040000C	**end:**	(nessuna istruzione)

AL TERMINE DELL'ASSEMBLAGGIO VIENE PRODOTTO IL **FILE OGGETTO** che contiene:

- la *tabella dei simboli* con indirizzi virtuali per etichette e variabili
- *codice macchina* nella sezione .text
- *dati* riservati nella sezione .data

FILE OGGETTO

Un **file oggetto** (*.o* oppure *.obj*) è il risultato della **compilazione o assemblaggio** di un file sorgente (*.c*, *.asm*, ecc.) ed è quindi creato dall'assemblatore.

Contiene:

- *Codice macchina* parzialmente compilato
- *Tabelle dei simboli* (etichette, variabili)
- *Dati* riservati nella sezione .data
- *Riferimenti a simboli esterni non ancora risolti* (che verranno risolti dal **linker**)

FILE OGGETTO: ESEMPIO

FILE OGGETTO: loop.o

[Symbol Table]

---

array @ 0x10010000 (**.data**)
size @ 0x10010028 (.data)
main @ 0x00400000 (**.text**)
loop @ 0x00400004 (.text)
end @ 0x0040000C (.text)

[.text Section]

0x00400000: addi $t0, $zero, 0
0x00400004: beq $t0, $t1, 0x0040000C
0x00400008: j 0x00400004

[.data Section]

0x10010000–0x10010027: spazio vuoto (40 byte per 'array')
0x10010028: .word 10 (valore di 'size')

ESEMPIO

```
main:
    li $t0, 0
loop:
    addi $t0, $t0, 1
    beq $t0, $t1, end
    j loop
end:
    li $v0, 10
    syscall
```

Durante la **prima passata**, l'assemblatore:

- Nota che `main` si trova a 0x00400000
- `loop` a 0x00400004
- `end` a 0x00400010

Nella **seconda passata**, quando trova `beq $t0, $t1, end`

...può finalmente calcolare **l'offset** corretto da inserire nell'istruzione `beq`, perché **ora conosce l'indirizzo di end**!

LINKER

Il **linker** è uno strumento che prende **più file oggetto** (generati da più file .asm) e:

1. **Unisce** il codice e i dati in un *unico programma eseguibile*.
2. **Risolve i riferimenti tra simboli** (es. se un file usa un'etichetta *func* definita in un altro).
3. **Assegna indirizzi finali** a tutte le etichette, dati e istruzioni, aggiornando la memoria.

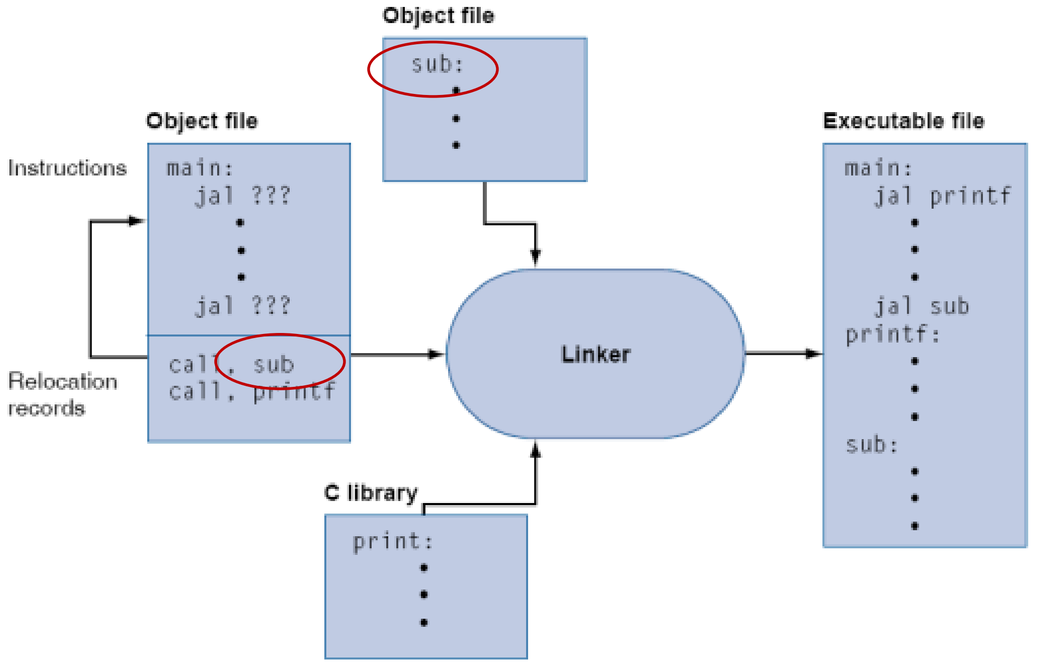


FIGURE A.3.1 The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

LINKER vs ASSEMBLER

Assemblatore (.asm → .o)	Linker (o. → .exe)
Lavora su **un file alla volta**	Lavora su **più file oggetto** insieme
Risolve **etichette locali**	Risolve **simboli globali ed esterni**
Crea codice **parzialmente assemblato**	Genera **codice finale eseguibile**
Crea **riferimenti simbolici non risolti**	Risolve quegli stessi riferimenti

LOADER

- Lettura dell'intestazione del file eseguibile per **determinare la lunghezza del segmento di testo** (cioè delle istruzioni) **e del segmento dati** (cioè le variabili)
- **Creazione di uno spazio di indirizzamento** sufficiente a contenere testo e dati
- Copia delle istruzioni e dati dal file eseguibile in memoria
- Copia nello stack degli eventuali parametri passati al programma principale
- **Inizializzazione dei registri e impostazione dello stack pointer** affinchè punti alla prima locazione libera
- **Impostare il Program Counter (PC)** al punto di inizio (es. etichetta main)
- Salto a una **procedura di startup** la quale copia i parametri nei registri argomento e chiama la procedura principale del programma
- Quando la procedura principale restituisce il controllo, la procedura di startup **termina il programma con una chiamata alla funzione di sistema exit**

LOADER: ESEMPIO (1/2)

Supponiamo che il loader segua la convenzione MIPS32:

- .text inizia da **0x00400000**
- .data inizia da **0x10010000**
- Stack da **0x7FFFFFFC** verso il basso

```
.data
array: .space 40        # Alloca spazio per 10 word (40 byte)
size:  .word 10         # Definisce una costante di valore 10

.text
main:
    li $t0, 0           # Inizializza $t0 a 0
loop:
    # Corpo del loop
    beq $t0, $t1, end # Se $t0 == $t1, salta a 'end'
    j loop              # Salta all'inizio del loop
end:
    # Fine del programma
```

Codice in slide #8

1. Caricamento del segmento .data

ETICHETTA	INDIRIZZO	CONTENUTO
array	0x10010000	40 byte (vuoti, per 10 word)
size	0x10010028	0x0000000A (valore 10)

LOADER: ESEMPIO (2/2)

```
.data
array: .space 40       # Alloca spazio per 10 word (40 byte)
size:  .word 10        # Definisce una costante di valore 10

.text
main:
    li $t0, 0          # Inizializza $t0 a 0
loop:
    # Corpo del loop
    beq $t0, $t1, end # Se $t0 == $t1, salta a 'end'
    j loop             # Salta all'inizio del loop
end:
    # Fine del programma
```

2. Caricamento del segmento .text

INDIRIZZO	ISTRUZIONE ASSEMBLY	NOTE
0X00400000	li $t0, 0	pseudoistruz. → addi $t0, $zero, 0
0X00400004	beq $t0, $t1, end	salto condizionato
0X00400008	j loop	salto assoluto
0X0040000C	# end: (nessuna istruz.)	etichetta di arrivo

L'etichetta loop è a 0x00400004, end è a 0x0040000C.

3. Loader imposta i registri

REGISTRO	VALORE	MOTIVO
$pc	0x00400000	Inizio del programma (main)
$sp	0x7FFFFFFC	Stack pointer iniziale

Appendice A

ESERCIZIO 1

Determinare a quale istruzione macchina MIPS corrisponde la sequenza binaria

00100010111010110000000001100000

1. cerco l'opcode (conversione in decimale dei primi 6 bit) nella tabella a pagina 50 dell'appendice A - in alcuni casi saranno necessari anche gli ultimi 6 bit (function code)
ATTENZIONE! Nella tabella a pag. A-50, ***op[31:26] è indicato in decimale o esadecimale*** *mentre* ***func[0:5] è SOLO in decimale!***

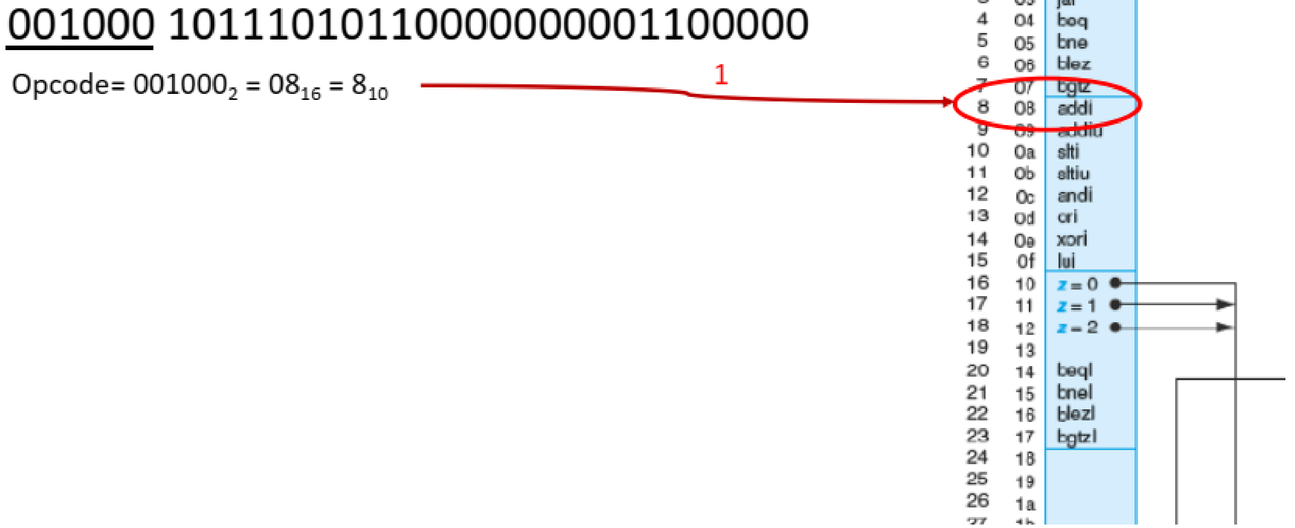
2. cerco nell'appendice A la descrizione (sintassi e semantica) dell'istruzione corrispondente
• dalla descrizione della sintassi capisco il formato (e tipo) dell'istruzione che servirà per sapere come dividere i restanti bit
• dalla descrizione della semantica dell'istruzione capisco come interpretare i restanti bit

ESERCIZIO 1

Determinare a quale istruzione macchina MIPS corrisponde la sequenza binaria

00100010111010110000000001100000

pag. 50, App.A

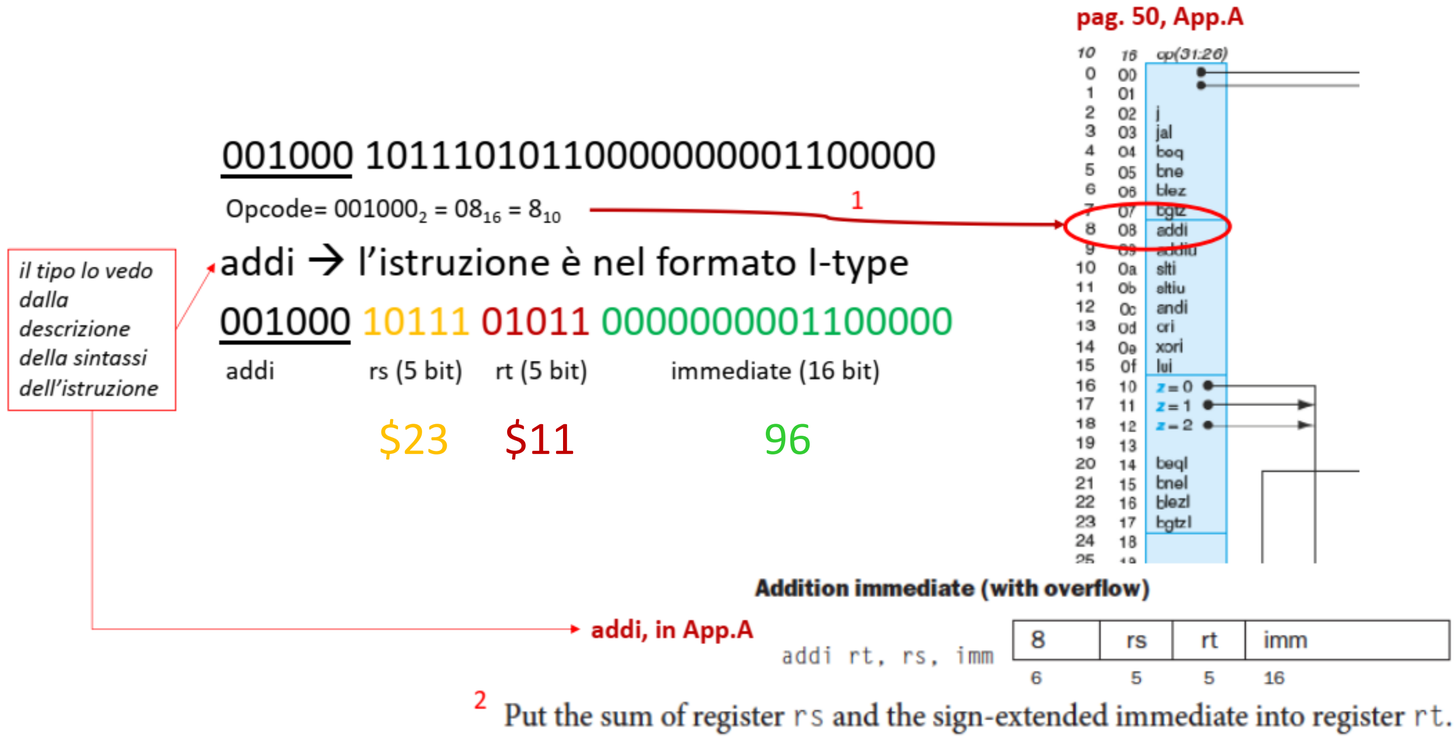


UNIVERSITÀ DEGLI STUDI DI TRIESTE

ESERCIZIO 1

Determinare a quale istruzione macchina MIPS corrisponde la sequenza binaria

00100010111010110000000001100000



UNIVERSITÀ DEGLI STUDI DI TRIESTE

ESERCIZIO 2

A quale istruzione macchina MIPS corrisponde il

codice esadecimale: 0x8fa40000

Binario: 100011 11101 00100 0000000000000000

opcode= 35_{10}= 23_{16}

1) da opcode[31:26]
a pag.A-50 App A → lw

lw $4, 0($29)

Non è registro a caso: è $sp!

Materiale per la lezione

- *Hennessy-Patterson, cap. 1 pp.14-16*
- *Appendix A.2, A.3, A.4, A.5*

Prossima lezione: 28 ***marzo, h.11:00, aula 5B***