



Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2018-2019

Architettura degli Elaboratori II Laboratorio 2018-2019

Turno A

Nicola Basilico
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4008
nicola.basilico@unimi.it
+39 02.503.16289

Turno B

Marco Tarini
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4010
marco.tarini@unimi.it
+39 02.503.16217

Info (turno A)

- Nicola Basilico, nicola.basilico@unimi.it
- Ufficio S242, Dipartimento di Informatica, Via Celoria 18 - 20133 Milano (MI),
- **Ricevimento su appuntamento** o in aula a valle delle sessioni di laboratorio
- Home page del corso per **materiale e avvisi** <http://teaching.basilico.di.unimi.it/>

Info (turno B)

- Marco Tarini,
- Mi trovate ... su google. Oppure:
- marco.tarini@unimi.it
- <http://tarini.di.unimi.it>
- Ufficio: 4to piano, Dipartimento di Informatica, Via Celoria 18 - 20133 Milano (MI),
- Ricevimento: Martedì 14:30-17:30

Corso di laboratorio ed esame

- 24 ore di lezione/esercitazione al computer
- Esame di laboratorio al PC
- $\text{Voto di Architettura} = \frac{2}{3}TEORIA + \frac{1}{3}LAB$
 - Una volta ottenuto, il voto di laboratorio ha validità di 18 mesi.
 - Una volta ottenuti entrambi i voti, l'esame viene verbalizzato.



Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2018-2019

Progettare e assemblare software in MIPS

Turno A

Nicola Basilico

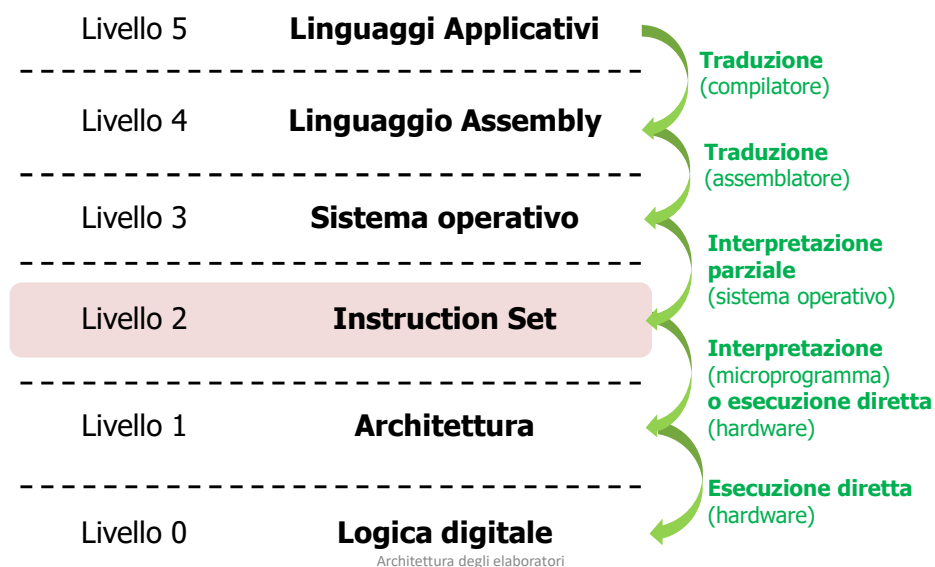
Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4008
nicola.basilico@unimi.it
+39 02.503.16289

Turno B

Marco Tarini

Dipartimento di Informatica
Via Celoria 18 - 20133 Milano (MI)
Ufficio 4010
marco.tarini@unimi.it
+39 02.503.16217

I livelli

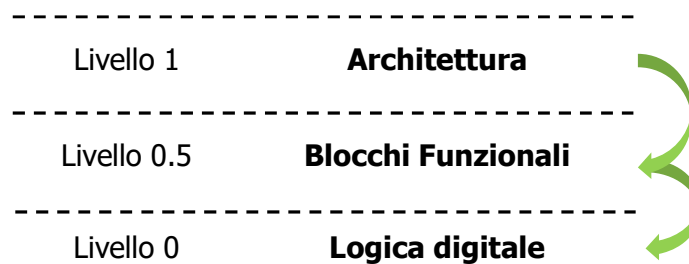


Livelli di astrazione

Ciascun livello consiste di:

- Un'interfaccia
 - cos'è visibile dall'esterno
 - usata dal livello superiore
- Un'implementazione
 - come lavora internamente
 - usa l'interfaccia del livello inferiore

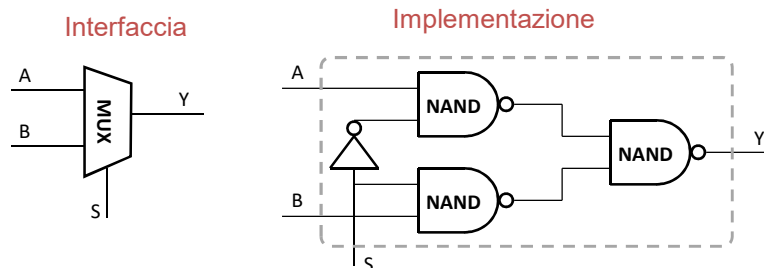
Livelli intermedi:



Livelli di astrazione

Esempio:

livello dei Blocchi Funzionali

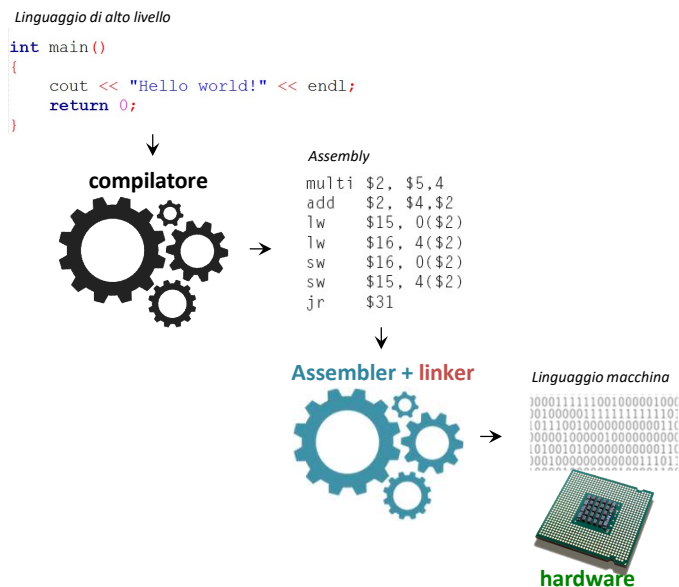


Il livello ISA

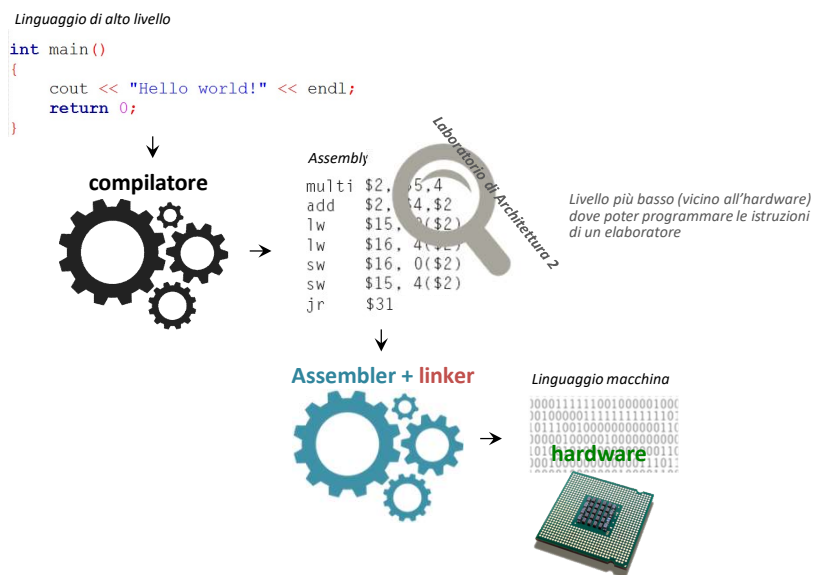
Architettura degli elaboratori

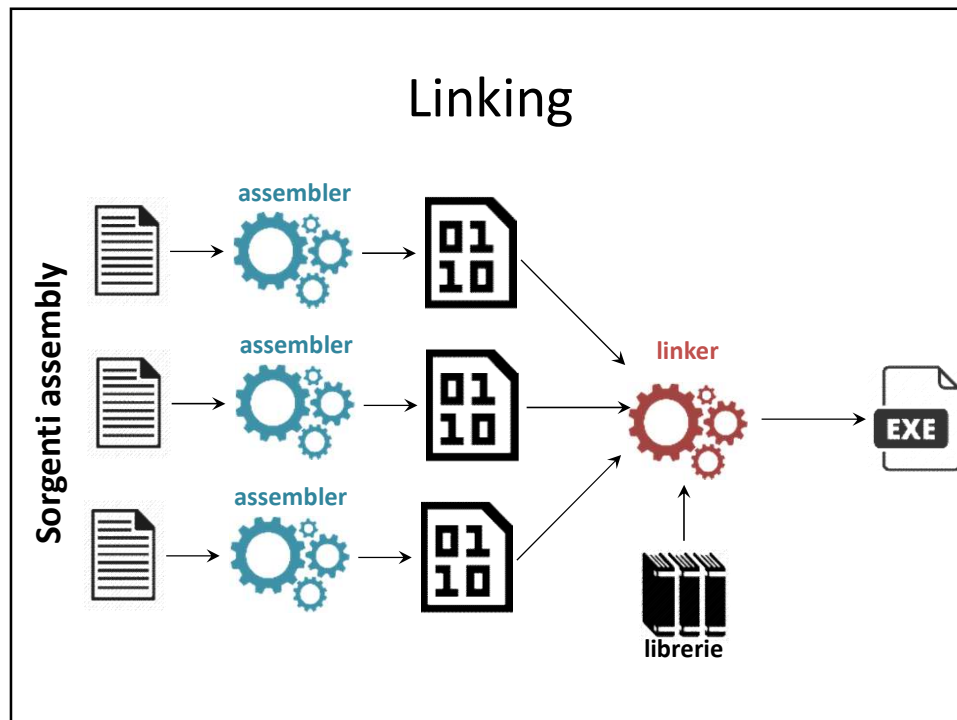
- 8 -

Compilazione



Compilazione





Assembly

È la rappresentazione simbolica del linguaggio macchina di un elaboratore.

```
add $t0 $s2 $s3
Binary: 00000010010100110100000000100000
Hex: 0x02534020
```

[MIPS instruction converter](#)

31	26 25	16 15	11 10	6 5	0
SPECIAL	\$s2	\$s3	\$t0	ADD	
000000	10010	10011	01000	00000	100000
6	5	5	5	5	6

- Dà alle istruzioni una forma *human-readable* e permette di usare **label** per referenziare con un nome parole di memoria che contengono istruzioni o dati.

The diagram illustrates the assembly process. On the left, three source assembly files (Sorgenti assembly) are shown. Each file is processed by an assembler (represented by blue gears) to produce an object file (represented by a document with 01 and 10). These object files are then combined by a linker (represented by red gears) along with libraries (librerie, represented by a stack of books) to produce the final executable file (EXE).

- Programmi coinvolti:
 - assembler**: «traduce» le istruzioni assembly (da un **file sorgente**) nelle corrispondenti istruzioni macchina in formato binario (in un **file oggetto**);
 - linker**: combina i files oggetto e le librerie in un **file eseguibile** dove la «destinazione» di ogni label è determinata.

Assembly

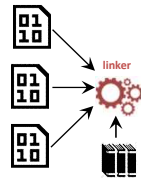
- Il codice Assembly può essere il risultato di due processi:
 - *target language* del compilatore che traduce un programma in linguaggio di alto livello (C, Pascal, ...) nell'equivalente assembly;
 - *linguaggio di programmazione* usato da un programmatore.
- Assembly è stato l'approccio principale con cui scrivere i programmi per i primi computer.
- Oggi la complessità dei programmi, la disponibilità di compilatori sempre migliori e di memoria rendono conveniente programmare in linguaggi di alto livello.
- Assembly come linguaggio di programmazione è adatto in certi casi particolari:
 - ottimizzare le performance (anche in termini di prevedibilità) e spazio occupato da un programma (ad es., sistemi embedded);
 - eredità di certi sistemi vecchi, ma ancora in uso, dove Assembly rappresenta l'unico modo conveniente per scrivere programmi;
 - rendere più efficienti certe istruzioni che hanno una semantica di basso livello.

Assembler



- Assembler traduce il sorgente Assembly in linguaggio macchina:
 1. associa ad ogni label il corrispondente indirizzo di memoria (label locali, cioè che danno il nome a oggetti referenziati solo dallo stesso file sorgente);
 2. associa ad ogni istruzione simbolica l'*opcode* e gli *argomenti* in codice binario.
- In generale, il file oggetto generato non può essere eseguito: Assembler non è in grado di risolvere le label esterne (quelle che danno il nome ad oggetti che possono essere referenziati da **altri** file sorgenti).
- Formato del file oggetto:
 - **segmento testo**: contiene le istruzioni;
 - **segmento dati**: contiene la rappresentazione binaria dei dati definiti nel file sorgente (ad esempio stringhe, costanti);
 - **informazione di rilocalizzazione**: dice quali sono le istruzioni che usano indirizzi assoluti (ad esempio una chiamata ad una procedura esterna);
 - **tabella dei simboli**: per ogni label esterna, la tabella dice quale è l'indirizzo associato ed elenca le label usate nel file che sono *unresolved*;
 - (**informazioni di debug**: informazioni riguardo al modo con cui si è svolta la compilazione.)

Linker



Il Linker combina tutti i file oggetto in un unico file che **può essere eseguito**.

- Determina quali librerie vengono usate e che quindi vanno incluse nel file eseguibile finale.
- Determina gli indirizzi di memoria a cui, nel file eseguibile, staranno le procedure e dati, «aggiusta», usando le informazioni di rilocazione, le istruzioni che fanno uso di indirizzi assoluti.
- Risolve le *unresolved* labels.



Il codice finale assemblato contiene ora in modo completo tutte le informazioni che servono per poterlo eseguire.

Fase di load

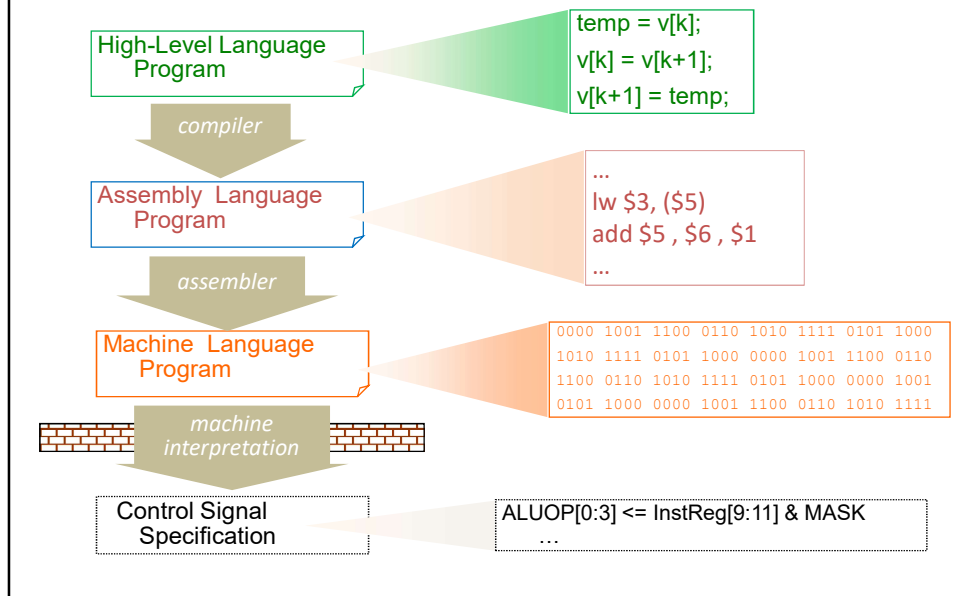


Il file eseguibile di solito risiede su una memoria di massa (o memoria secondaria), quando se ne invoca l'esecuzione deve essere caricato in memoria primaria.

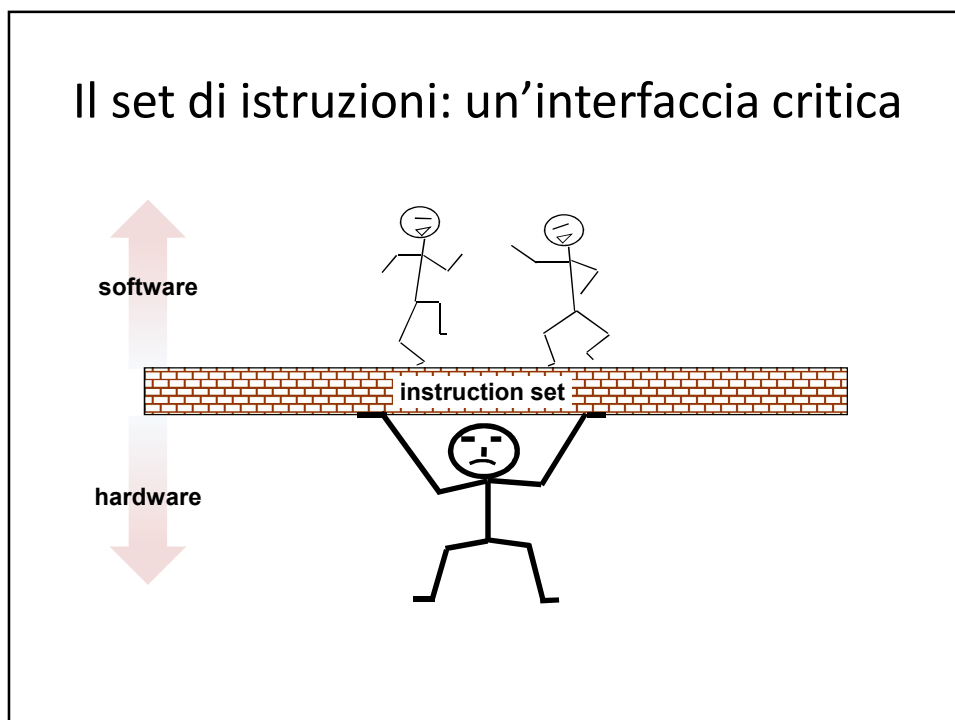
Fase di load:

1. lettura dell'header per estrarre la dimensione dei vari segmenti;
2. creazione dello spazio degli indirizzi in memoria e caricamento dei vari segmenti;
3. procedure di inizializzazione (clear dei registri, load sullo stack dei parametri, inizializzazione dello stack pointer, ...);
4. chiama di una routine che invocherà a sua volta [main](#).

Il codice da un livello all'altro



Il set di istruzioni: un'interfaccia critica



ISA: Instruction Set Architecture

- Il livello visto dal programmatore assembly o dal compilatore.
- Comprende:
 - **Instruction Set**
(quali operazioni possono essere eseguite?)
 - **Instruction Format**
(come devono essere scritte queste istruzioni? cioè la loro *sintassi*)
 - **Data storage**
(dove sono posizionati i dati?)
 - **Addressing Mode**
(come si accede ai dati?)
 - **Exceptions**
(come vengono gestiti i casi eccezionali?)

Il set di istruzioni: un'interfaccia critica

- È un difficile compromesso fra:
 - massimizzare le prestazioni
 - massimizzare la semplicità di uso
 - minimizzare i costi di produzione
 - minimizzare i tempi di progettazione
- Definisce la sintassi e la semantica del linguaggio

Criteri di progettazione per un IS

- La scelta di un insieme delle istruzioni prende in considerazione:
 - la semplicità della realizzazione dell'HW richiesto
 - filosofia RISC: favorire questo
 - l'espressività e la semplicità di uso
 - la potenza delle istruzioni
 - filosofia CISC: favorire questo
 - la facile programmabilità
 - l'efficienza
 - velocità di esecuzione
 - cioè:

$$(\text{vel di ogni istruzione}) \times (\text{quante se ne rendono necessarie})$$

filosofia RISC:
abbassare questo

filosofia CISC:
abbassare questo

Alcuni Instruction Set popolari oggi

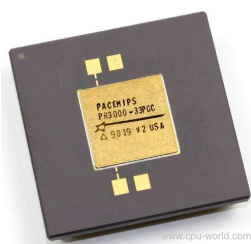
MIPS	}	filosofia RISC
ARM		
X86	}	filosofia CISC
X64		

[per i curiosi: cercare sulla Wikipedia...](#)

Criteri di definizione di un IS

- Oltre a questi criteri obiettivi, molte altre forze più capricciose hanno plasmato gli Instruction Sets oggi usati
 - La loro storia (soprattutto il bisogno di back compatibility) (non tutte le scelte sono logiche a posteriori)
 - Fattori economici / legali
 - Considerazioni tecniche pertinenti al contesto del loro sviluppo

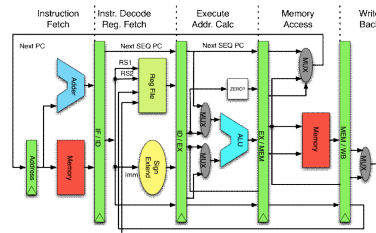
MIPS



- In questo laboratorio lavoreremo con **MIPS** (Multiprocessor without Interlocked Pipeline Stages): un'ISA di tipo RISC
- Nasce a metà anni '80 come architettura *general purpose*;
- Inizialmente è un progetto accademico (Stanford), poco dopo diventa commerciale
- Oggi è impiegata prevalentemente nell'ambito dei *sistemi embedded*

MIPS

- La maggior parte dei corsi accademici di architetture adotta MIPS, perché?



- È una prima e lineare implementazione del concetto di pipeline
- È costruita su una semplice assunzione: ogni stadio della pipeline deve terminare in un ciclo di clock, ogni stadio non necessita di attendere il completamento degli altri (interlock)
- (Oggi l'assunzione è rilassata per avere istruzioni come moltiplicazione e divisione, ma il nome è rimasto lo stesso)*

MIPS

- La semplicità dell'architettura emerge anche a livello di Assembly

"Hello world" in x86 (64 bit)

```
.file "hello_world.c"
.section .rodata
.LC0:
.string "Hello world!"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
call puts
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits
```

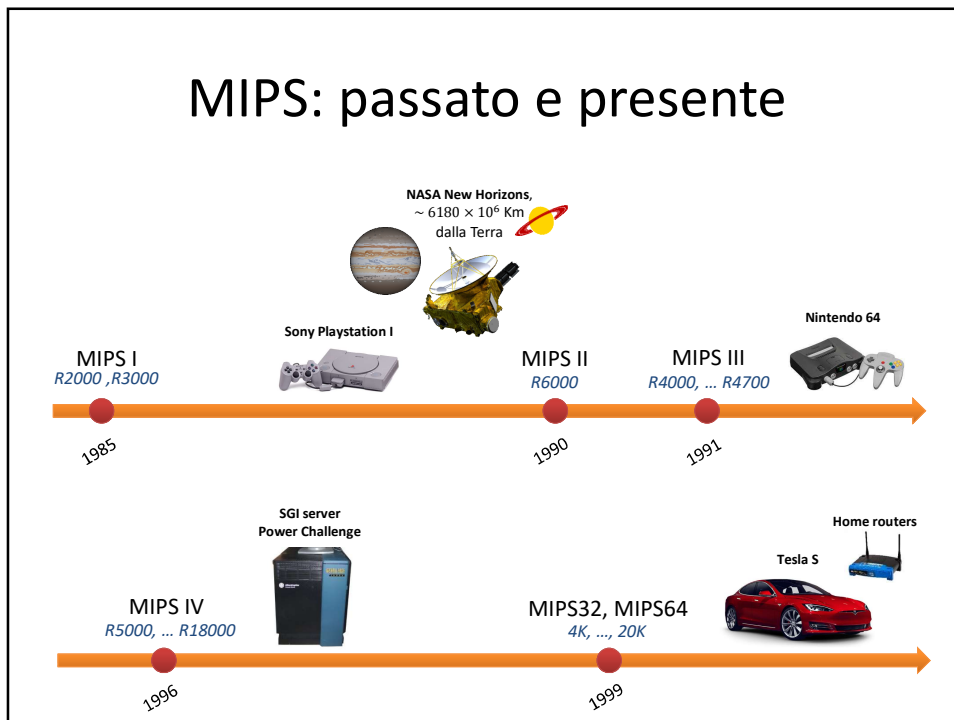
"Hello world" in MIPS (32 bit)

```
.data
hello: .asciiz "\nHello, World!\n"

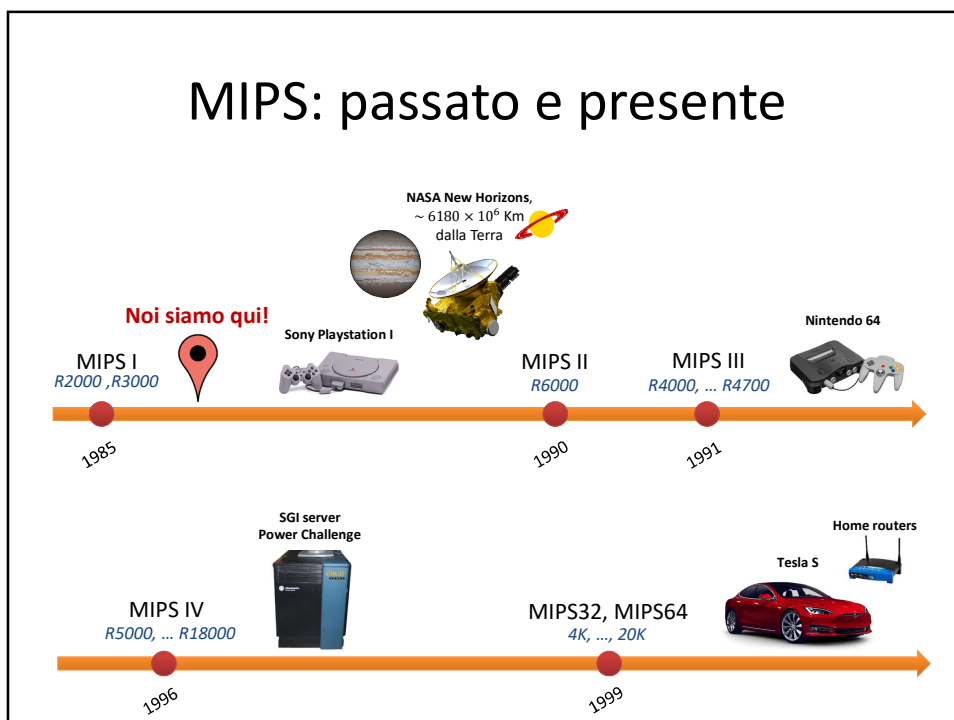
.text
.globl main
main:
li $v0, 4
la $a0, hello
syscall

li $v0, 10
syscall
```

MIPS: passato e presente



MIPS: passato e presente



Problema pratico per questo lab

- MIPS = molto adatto alla didattica dei linguaggi assembly
- Vogliamo scrivere ed eseguire programmi in MIPS
- Dove trovare una macchina MIPS?
 - un HW in grado di eseguire un programmal scritto in linguaggio macchina MIPS?

senza
avere
questo →



Emulazione (= Instruction-Level Simulation)

- Ho una macchina, con Instruction Set a , ma ho programmi eseguibili scritti in diverso Instruction Set b
- Scrivo un interprete (un programma in IS a) per un un IS b
 - Questo interprete è in grado di eseguire programmi scritti in b
- Ora ho una «*macchina virtuale*» per B!
 - posso eseguire programmi per B pur non avendo una macchina fisica costruita per capire B

Emulazione (= Instruction-Level Simulation)

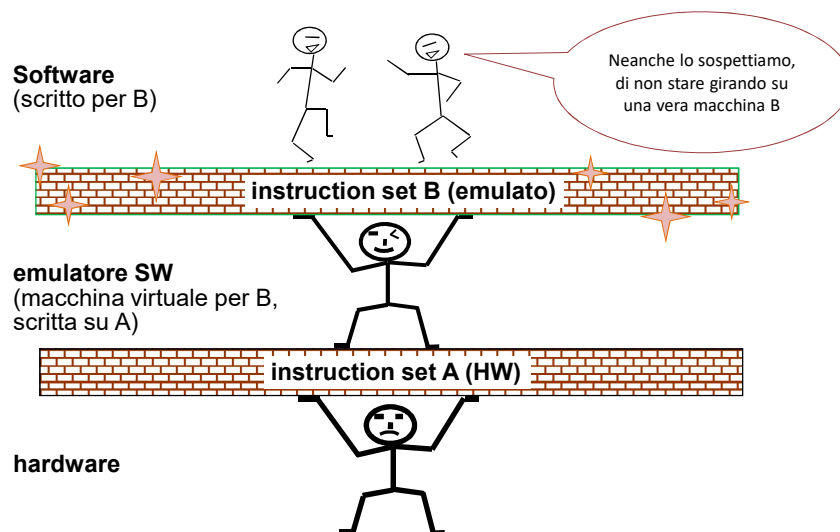
Vantaggi:

- consente di riutilizzare il programma (così com'è, alcun senza adattamento, o riscrittura) scritto per un'altra architettura B, senza avere l'HW che la esegue
 - (programma = dati + istruzioni)
- Chi scrive (o, ha scritto) il software per B non deve fare nulla di diverso dal solito

Svantaggi:

- La performance può risentirne:
Se le prestazioni di A non sono molto superiori a quelle attese per B,
il programma emulato andrà molto più lento di quello originale che giri su un HW che implementa l'IS B

Digressione: le gioie dell'emulazione SW



Emulazione: alcuni progetti interessanti

- **DosBox:**
 - emula l'IS X86, e il vecchio Sistema Operativo DOS, su varie piattaforme moderne (Windows, etc)
 - Consente di eseguire vecchi programmi DOS degli anni '80
- **WinE (Windows Emulator):**
 - emula varie piattaforme Windows per MacOS (emula il livello SO, non IS)
 - Consente di eseguire programmi Windows su Mac
- **MAME (Multi Arcade Machine Emulator)**
 - emula migliaia di IS di Arcade Machine (coin-op machine)
 - Consente di eseguire videogames arcade («da sala giochi») fine anni '70, anni '80, '90 e 2000
- **MESS (Multi Emulator Super System) --- ora parte di MAME**
 - emula centinaia di IS di home gaming console (VIC-20, C-64, ...)
 - consente di eseguire videogames da home entertainment



multi
arcade-machine
emulator

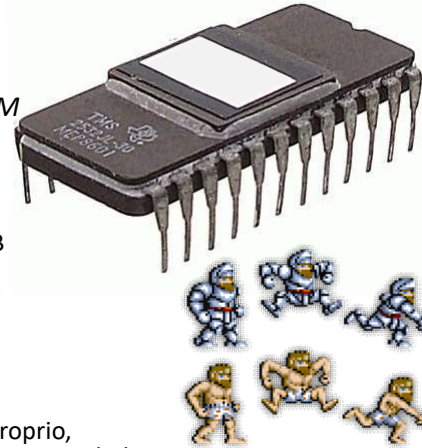


- Emulazione di migliaia di IS propri delle Architetture HW dedicate al gaming
 - coin-op dagli anni '70 ad oggi
- Software: i programmi originali per questi IS sono recuperati la' dove hanno aspettato per decenni: in chip di ROM
 - ROM dump = scaricare il contenuto di una (qui: vecchia) ROM
- Finalità: recuperare videogames, importanti pezzi della nostra storia culturale
 - è una corsa contro il tempo: ROM è memoria «persistente» sì... ma entro certi limiti temporanei!
 - quasi sempre, HW capace di eseguire quel dato IS non esiste più: Senza emulazione, molti video-games storici sarebbero perduti!



multi
arcade-machine
emulator

- I programmi da eseguire in emulazione sono estratti da un insieme (10-50) di *chip di ROM*
 - capacità complessiva: molto poco!
Per es:
pac-man (Midway, 1980): 20 KB
ghost'n'goblin (Taito, 1985): 400 KB
- Come sempre:
Programmi = DATI + ISTRUZIONI
- Nel caso dei videogame
 - DATI: includono i game asset: sprites, effetti sonori, fonts, ...
 - ISTRUZIONI: il programma vero e proprio, scritto nell'IS della rispettiva architettura dedicata



MARS

(MIPS Assembler and Runtime Simulator)

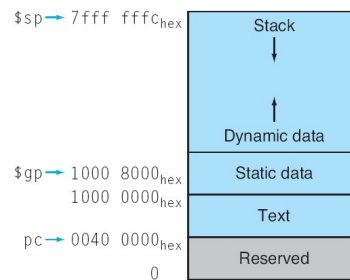


Missouri State
UNIVERSITY

- Implementa un assembler di MIPS
- Emula la CPU che obbedisce alle convenzioni MIPS I a 32 bit
- Fornisce un IDE (comodo)
- Disponibile a questo URL
<http://courses.missouristate.edu/KenVollmar/mars/>

Il programma in memoria (in MIPS)

- **Segmento testo:** contiene le istruzioni del programma.
- **Segmento dati:**
 - **dati statici:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma (e.g., *variabili statiche, costanti, etc.*);
 - **dati dinamici:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso (e.g., *liste dinamiche, etc.*).
- **Stack:** contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) (e.g., parametri di una procedura, valori di ritorno, etc.).



MARS (interfaccia)

The screenshot shows the MARS MIPS assembler interface. The main window is divided into several panes:

- Text Segment:** Displays assembly code with columns for Address, Code, Basic, and Source. A red arrow points to the 'Memoria (Text e Data)' label.
- Data Segment:** Displays data values with columns for Address, Value (+0), Value (+4), Value (+8), Value (+c), and Value (+10).
- Mars Messages / Run I/O:** Shows the execution log, including messages like 'Assemble: assembling /home/nbas/git/archlab/arch2lab/session_01/00-hello/hello.asm' and 'Go: execution completed successfully.' A green arrow points to the 'Logs e console (I/O)' label.
- Registers:** A table on the right showing the state of MIPS registers. A blue arrow points to the 'Banco Registri' label.

Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	10
\$v1	3	0
\$a0	4	268500992
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$t8	16	0
\$t9	17	0
\$s0	18	0
\$s1	19	0
\$s2	20	0
\$s3	21	0
\$s4	22	0
\$s5	23	0
\$s6	24	0
\$s7	25	0
\$s8	26	0
\$s9	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194328
hi		0
lo		0

MARS (Registri)

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	10
\$v1	3	0
\$a0	4	268500992
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194328
hi		0
lo		0

- 32 registri a 32bit per operazioni su interi (**\$0..\$31**).
- 32 registri a 32 bit per operazioni in virgola mobile sul coprocessore 1 (**\$FP0..\$FP31**).
- registri speciali a 32bit:
 - il **Program Counter (PC)** l'indirizzo della prossima istruzione da eseguire;
 - **hi** e **lo** usati nella moltiplicazione e nella divisione;
 - **EPC, Cause, BadVAddr, Status** (coprocessore 0) vengono usati nella gestione delle eccezioni.
- I registri general-purpose sono chiamati col nome dato dalla convenzione MIPS e numerati da 0 a 31
- Il loro valore è ispezionabile nel formato esadecimale o decimale

Richiamo sulle Convenzioni MIPS

Nome	Numero	Utilizzo	Preservato durante le chiamate
\$zero	0	costante zero	<i>Riservato MIPS</i>
\$at	1	riservato per l'assemblatore	<i>Riservato Compiler</i>
\$v0-\$v1	2-3	valori di ritorno di una procedura	No
\$a0-\$a3	4-7	argomenti di una procedura	No
\$t0-\$t7	8-15	registri temporanei (non salvati)	No
\$s0-\$s7	16-23	registri salvati	Sì
\$t8-\$t9	24-25	registri temporanei (non salvati)	No
\$k0-\$k1	26-27	gestione delle eccezioni	<i>Riservato OS</i>
\$gp	28	puntatore alla global area (dati)	Sì
\$sp	29	stack pointer	Sì
\$s8	30	registro salvato (fp)	Sì
\$ra	31	indirizzo di ritorno	No

Il registro **\$1 (\$at)** viene usato come variabile temporanea nell'implementazione delle pseudo-istruzioni.

Richiamo: Istruzioni aritmetiche del MIPS

Comando	Sintassi (es)	Semantica (es)	Commenti
<i>add</i>	<code>add \$1,\$2,\$3</code>	$\$1 = \$2 + \$3$	operandi: 2 registri
<i>subtract</i>	<code>sub \$1,\$2,\$3</code>	$\$1 = \$2 - \$3$	operandi: 2 registri
<i>add immediate</i>	<code>addi \$1,\$2,99</code>	$\$1 = \$2 + 99$	operandi: registro e costante
<i>add unsigned</i>	<code>addu \$1,\$2,\$3</code>	$\$1 = \$2 + \$3$	operandi: 2 registri
<i>subtract unsigned</i>	<code>subu \$1,\$2,\$3</code>	$\$1 = \$2 - \$3$	operandi: 2 registri
<i>add immediate unsigned</i>	<code>addiu \$1,\$2,99</code>	$\$1 = \$2 + 99$	operandi: registro e costante
<i>multiply</i>	<code>mult \$2,\$3</code>	$Hi \mid Lo = \$2 \times \3	prodotto con segno: (risultato in 64 bit)
<i>multiply unsigned</i>	<code>multu \$2,\$3</code>	$Hi \mid Lo = \$2 \times \3	idem, ma senza segno
<i>divide</i>	<code>div \$2,\$3</code>	$Lo = \$2 \div \$3,$ $Hi = \$2 \bmod \3	Lo = quoziente Hi = resto
<i>divide unsigned</i>	<code>divu \$2,\$3</code>	$Lo = \$2 \div \$3,$ $Hi = \$2 \bmod \3	idem, ma senza segno
<i>move from Hi</i>	<code>mfhi \$1</code>	$\$1 = Hi$	Copia Hi in un registro
<i>move from Lo</i>	<code>mflo \$1</code>	$\$1 = Lo$	Copia Lo in un registro

Richiamo: Notazione MIPS (somma e sottrazione)

- Primo argomento: risultato dell'operazione
 - sempre un registro
 - nota: eccezione per prodotto e divisione (vedi dopo)
- Secondo argomento: il primo operando
 - sempre un registro (scelta del MIPS)
- Terzo argomento: il secondo operando
 - un registro: `$...`, oppure...
 - ...un valore *immediate* (un «literal»): un numero senza `$`
 - nota: si tratta di due istruzioni diverse!
Sia in assembly MIPS, che in linguaggio macchina MIPS.
Es: `add` vs `addi`, `sub` vs `subi`
- Comandi in due versioni
 - «unsigned»: ignora overflow
 - «signed»: riporta overflow

`add` vs `addu`, `sub` vs `subu`
 dove `u` sta per «unsigned»
 L'unica differenza è l'overflow!

Es. 1.1

- Nome del file sorgente: *storesum.asm*
- Si scriva il codice Assembly che:
 - metta il valore 5 nel registro *\$1*,
 - metta il valore 7 nel registro *\$2*,
 - metta la somma dei due nel registro *\$3*.

Osservazioni: filosofia RISC in azione

- MIPS non ha un'istruzione per copiare un valore dato in un registro dato
- Motivo:
l'hardware è più semplice se l'IS prevede meno istruzioni diverse!
- Nessun problema: al suo posto, usiamo l'operazione somma fra registri
- Usando la convenzione MIPS «il registro 0 contiene sempre 0»
- Es: per memorizzare il valore 15 nel registro *\$8*, scriviamo:
«*\$8* prende la somma di *\$0* e il valore immediate 15»

Es. 1.1 Soluzione e osservazioni

```
.text  
  
addi $1    $0    5    # $1 = 5  
addi $2    $0    7    # $2 = 7  
  
add $3    $1    $2    # $3 = $1 + $2
```

Note:

.text: keyword mips che significa

“ora segue la parte istruzioni di questo programma”

La # identifica un commento (l'assembler semplicemente ignora la parte rimanente della riga)

Es. 1.1 (step by step)

1. scrivere il codice Assembly nell'editor di MARS (è anche possibile usare un editor di testo)
2. caricare il file in MARS e/o assemblarlo
3. lanciare l'esecuzione
4. osservare come variano i registri coinvolti nelle operazioni
5. ripetere mediante l'uso di un break point, aggiornare codice, re-inizializzare il simulatore e ricominciare da 2

Es. 1.2

- Nome del file sorgente: *expression.asm*
- Esercizio:
scriviamo un programma che calcola $(22-2) + (34-10)$ e mette il risultato nel registro **\$8**
- Scelte arbitrarie: il nostro programma ...
 - memorizza i valori 22, 2, 34 e 10 in quattro registri
Scegliamo **\$9, \$10, \$11, \$12**
 - Sottrae **\$10** da **\$9** (risultato ancora in **\$10**)
 - Sottrae **\$12** da **\$11** (risultato ancora in **\$12**)
 - Somma **\$12** a **\$10** per ottenere il risultato finale (risultato in **\$8**, come richiesto)

Es. 1.2 Una soluzione

```
.text

addi $9  $0  22      # A=22 , con A in $9
addi $10 $0  2       # B=2  , con B in $10
addi $11 $0  34      # C=34 , con C in $11
addi $12 $0  10      # D=10 , con D in $12

sub $9  $9  $10      # A = A-B
sub $11 $11 $1       # C = C-D
sub $8  $9  $11      # $8 = A+C
```

Osservazioni: filosofia RISC in azione

- L'hardware è più semplice se il numero di operandi è costante
- In MIPS, ogni operazione ha max due operandi
- Un'operazione che implica più di due operandi dovrà essere suddivisa in una sequenza di operazioni
- Esempio:
una *somma a tre addendi* non viene scritta:

`add $5, $6, $7, $8` # $\$5 = \$6 + \$7 + \8 **(non è MIPS!)**

ma deve essere scomposta (ad esempio) così:

`add $5, $6, $7` # $\$5 = \$6 + \$7$

`add $5, $5, $8` # $\$5 = \$5 + \$8$

- Spetta al compilatore (o al programmatore Assembly) il compito di ottimizzare la sequenza di operazioni.



Università degli Studi di Milano
Laboratorio di Architettura degli Elaboratori II
Corso di Laurea in Informatica, A.A. 2018-2019

Hanno contribuito alla realizzazione di queste slides:

- Nicola Basilico
- Marco Tarini
- Iuri Frozio
- Jacopo Essenziale