Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2017-2018

Architettura degli Elaboratori II
Laboratorio 2017-2018

Homepage del corso

Turno A
Nicola Basilico
Dipartimento di Informatica
Via Comelico 39/41 - 20135 Milano (MI)
Ufficio S242
nicola.basilico@unimi.it
+39 02.503.16294

Turno B
Jacopo Essenziale
Dipartimento di Informatica
Via Celoria 20 - 20133 Milano (MI)
AISLab
jacopo.essenziale@unimi.it
+39 02.503.14010

Info (turno A)

- Nicola Basilico, nicola.basilico@unimi.it
- Ufficio S242, Dipartimento di Informatica, Via Comelico 39/41 - 20135 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)

- Jacopo Essenziale, jacopo.essenziale@unimi.it

- AISLab, Dipartimento di Informatica,
  Settore Didattico - Via Celoria 20 - 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/

Corso di laboratorio ed esame

- 24 ore di lezione/esercitazione al computer
- Progetto individuale: proposta, approvazione, consegna.
- Una volta consegnato, il progetto va discusso con il docente.
- 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.
- È obbligatorio seguire le direttive riportate nella Guida all'esame di Laboratorio disponibile sul sito del corso (i progetti che non rispettano tali modalità non saranno valutati)
- Il calendario delle discussioni elenca una deadline di consegna e una data di discussione. Una volta consegnato il progetto viene inviata una convocazione per mail.

Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2017-2018

Progettare e assemblare software in MIPS


Homepage del corso

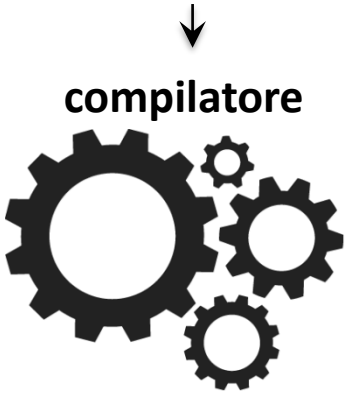
Turno A
Nicola Basilico
Dipartimento di Informatica
Via Comelico 39/41 - 20135 Milano (MI)
Ufficio S242
nicola.basilico@unimi.it
+39 02.503.16294

Turno B
Jacopo Essenziale
Dipartimento di Informatica
Via Celoria 20 - 20133 Milano (MI)
AISLab
jacopo.essenziale@unimi.it
+39 02.503.14010

Introduzione

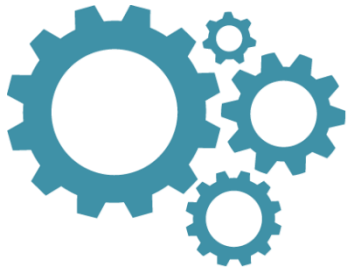
Linguaggio di alto livello

```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```



Assembly

```
multi $2, $5,4
add   $2, $4,$2
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
jr    $31
```



Linguaggio macchina

Introduzione

Linguaggio di alto livello

```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

compilatore

Assembly

```
multi  $2, $5,4
add    $2, $4,$2
lw     $15, 0($2)
lw     $16, 4($2)
sw     $16, 0($2)
sw     $15, 4($2)
jr     $31
```

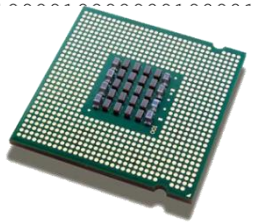
Laboratorio di Architettura 2

Livello più basso (vicino all'hardware) dove poter programmare le istruzioni di un elaboratore

Assembler + linker

Linguaggio macchina

hardware

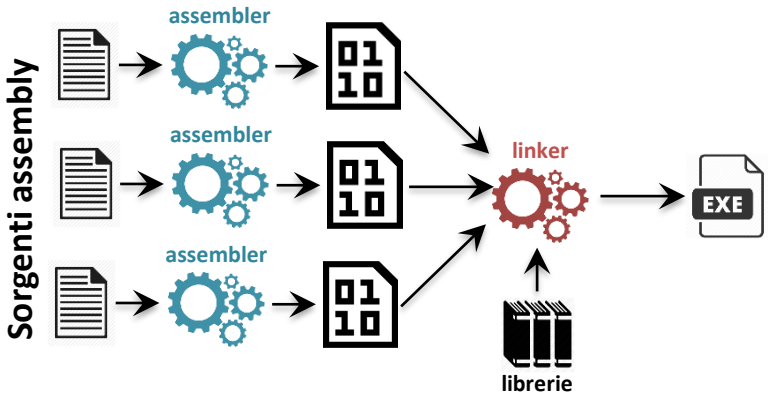


Assembly

- E’ la rappresentazione simbolica del linguaggio macchina di un elaboratore.

```
1000110010100000  <-->  add A,B
```

- 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.



- 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.

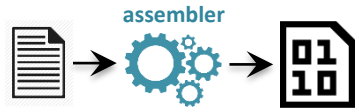
Assembly

- 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, l'invenzione di compilatori sempre migliori e la disponibilità crescente 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.

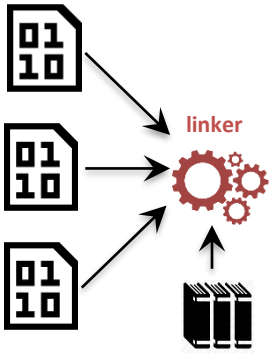
Assembler 

- 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).

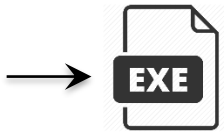
- Formato del file oggetto:
  - **segmento testo**: contiene le istruzioni;
  - **segmento dati**: contiene la rappresentazione binaria dei dati definiti nel file sorgente (ad esempio stringhe, costanti);
  - **informazione di rilocazione**: dice quali sono le istruzioni che usano indirizzi assoluti (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`.

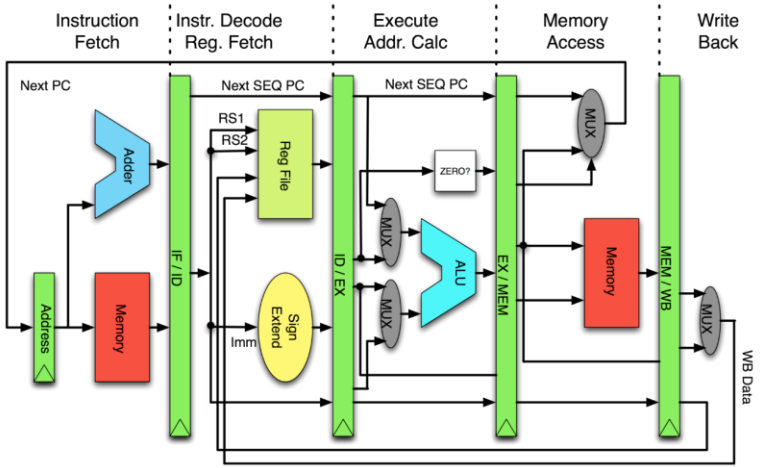
MIPS

- 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
- 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_wold.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)

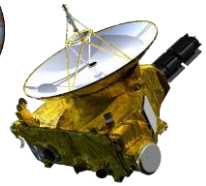
```
        .data
        hello: .asciiz "\nHello, World!\n"

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

        li $v0, 10
        syscall
```

MIPS: passato e presente

NASA New Horizons,
$\sim 6180 \times 10^6$ Km
dalla Terra



Sony Playstation I

Nintendo 64

MIPS I
R2000 ,R3000

1985

MIPS II
R6000

1990

MIPS III
R4000, … R4700

1991

SGI server
Power Challenge

MIPS IV
R5000, … R18000

1996

Home routers

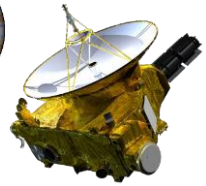
Tesla S

MIPS32, MIPS64
4K, …, 20K

1999

MIPS: passato e presente

NASA New Horizons,
$\sim 6180 \times 10^6$ Km
dalla Terra



Noi siamo qui!

MIPS I
R2000 ,R3000

1985

Sony Playstation I

MIPS II
R6000

1990

MIPS III
R4000, … R4700

1991

Nintendo 64

SGI server
Power Challenge

MIPS IV
R5000, … R18000

1996

MIPS32, MIPS64
4K, …, 20K

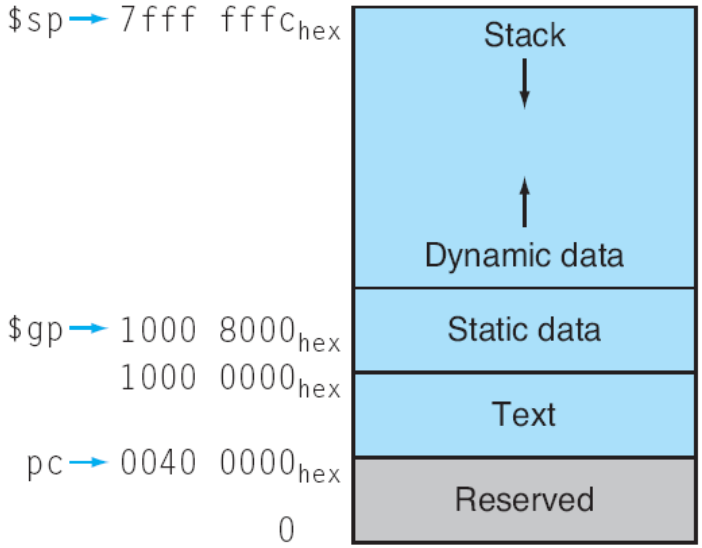
1999

Home routers

Tesla S

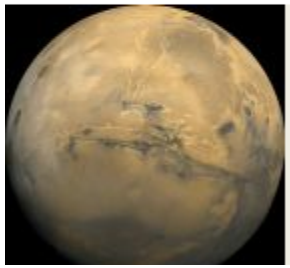
Il programma in memoria (in MIPS)

- **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

Missouri State UNIVERSITY



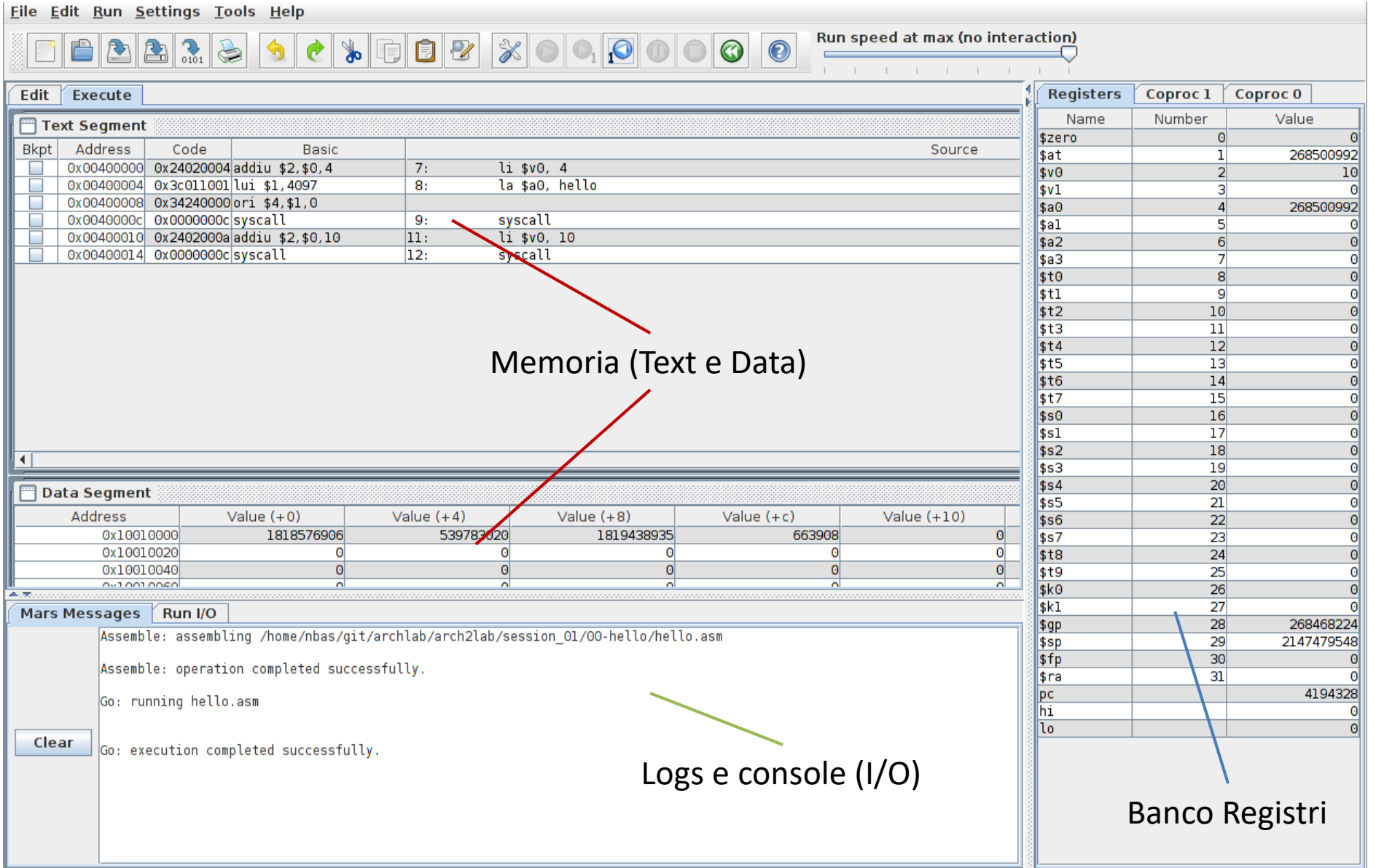
***MARS** (MIPS Assembler and Runtime Simulator)*

An IDE for MIPS Assembly Language Programming

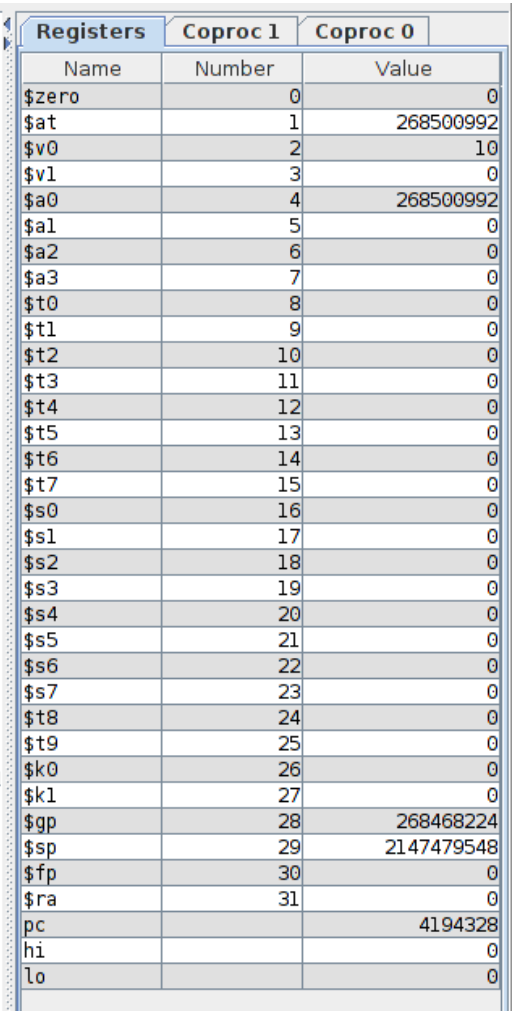
MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's *Computer Organization and Design*.

- È un simulatore di una CPU che obbedisce alle convenzioni MIPS I a 32 bit
- Perché usare un simulatore e non la macchina vera?
  - Usiamo tutti la stessa ISA indipendentemente dal calcolatore reale.
  - Ci offre una serie di strumenti che rendono la programmazione più comoda.
  - Maschera certi aspetti reali a cui non saremmo interessati (es., delays).
- Disponibile a questo URL http://courses.missouristate.edu/KenVollmar/mars/

MARS (interfaccia)



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	*Riservato MIPS*
$at	1	riservato per l'assemblatore	*Riservato Compiler*
$v0-$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	Si
$t8-$t9	24-25	registri temporanei (non salvati)	No
$k0-$k1	26-27	gestione delle eccezioni	*Riservato OS*
$gp	28	puntatore alla global area (dati)	Si
$sp	29	stack pointer	Si
$s8	30	registro salvato (fp)	Si
$ra	31	indirizzo di ritorno	No

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

Richiamo di istruzioni aritmetiche (somma, sottrazione)

- Convenzioni di notazione:
  - Identificativo con iniziale minuscola: deve essere un registro o un valore immediato (intero con segno su 16 bit);
  - Identificativo con iniziale «$»: deve essere un registro.

```
add $s1, $s2, s3 # $s1 = $s2 + s3, rileva overflow

sub $s1, $s2, s3 # $s1 = $s2 - s3, rileva overflow

addi $s1, $s2, 13# $s1 = $s2 + 13, rileva overflow

addu $s1, $s2, s3# $s1 = $s2 + s3, unsigned, non rileva overflow

subu $s1, $s2, s3# $s1 = $s2 - s3, unsigned, non rileva overflow

addui $s1, $s2, 27        # $s1 = $s2 + 17, unsigned, non rileva overflow
```

Es. 1.1

- Nome del file sorgente: *storesum.asm*

- Si scriva il codice Assembly che:
  - metta il valore 5 nel registro `$s1`,
  - metta il valore 7 nel registro `$s2`,
  - metta la somma dei due nel registro `$s0`.

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*

- Si traduca in Assembly la seguente riga di codice: A = B+C-(D+E), assegnando alle variabili A, B, C, D, E i registri `$s0, ..., $s4`.

- Si assumano valori iniziali 1, 2, 3 e 4

Es. 1.2 Soluzione e osservazioni

```
            .text
            .globl main

main:
            addi $s1, $zero, 1  # s1=1, B=1
            addi $s2, $zero, 2  # s2=2, C=2
            addi $s3, $zero, 3  # s3=3, D=3
            addi $s4, $zero, 4  # s4=4, E=4

            add $t0, $s1, $s2   # t0=s1+s2, t0=B+C
            add $t1, $s3, $s4   # t1=s3+s4, t1=D+E
            sub $s0, $t0, $t1   # s0=t0-t1, s0=(B+C)-
(D+E)
```

- Il risultato finale ottenuto nel registro $s0 è corretto e pari a 0xfffffffc.
- Prova:
  - (1+2)-(3+4) = 3-7 = -4
  - 0xfffffffc = $[1111,1111,1111,1111,1111,1111,1111,1100]_{C2}$ = ...
  - ... = -$\{[0000,0000,0000,0000,0000,0000,0000,0011]+1\}_2$ = -$\{100\}_2$ = -4

Osservazioni

- Filosofia RISC: un'operazione che implica più di due addendi viene divisa in una sequenza di operazioni (l'hardware è più semplice se il numero di operatori è costante).

A=(B+C)-(D+E)

```
add $t0, $s1, $s2# $t0=$s1+$s2, $t0=B+C
add $t1, $s3, $s4# $t1=$s3+$s4, $t1=D+E
sub $s0, $t0, $t1# $s0=$t0-$t1, $s0=(B+C)-(D+E)
```

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

Istruzioni: moltiplicazione

- Due istruzioni:
  - `mult $rs $rt`
  - `multu $rs $rt` `# unsigned`
- Il registro destinazione è **implicito.**
- Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati **hi (High order word)** e **lo (Low order word)**.
- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit.

Istruzioni: moltiplicazione

- Il risultato della moltiplicazione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:
  - `mfhi $rd  # move from hi`
    - sposta il contenuto del registro **hi** nel registro **rd**;
  - `mflo $rd  # move from lo`
    - sposta il contenuto del registro **lo** nel registro **rd**.

Operazioni aritmetiche: divisione

```
div $s2, $s3 # $s2 / $s3, divisione intera
```

- Il risultato della divisione intera va in:
  - Lo: $s2 / $s3 [quoziente];
  - Hi: $s2 mod $s3 [resto].

- Il risultato va quindi prelevato dai registri Hi e Lo utilizzando ancora la `mfhi` e la `mflo`.

Istruzioni: pseudo-istruzioni

- Le pseudoistruzioni sono un modo compatto ed intuitivo di specificare un insieme di istruzioni.
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore.

Esempi:

- `move $t0, $t1` `# pseudo istruzione`
  - `add $t0, $zero, $t1` `# (in alternativa) addi $t0, $t1, 0`

- `mul $s0, $t1, $t2` `# pseudo istruzione`
  - `mult $t1, $t2`
  - `mflo $s0`

- `div  $s0, $t1, $t2` `# pseudo istruzione`
  - `div $t1, $t2`
  - `mflo $s0`

Esercizio 1.3

- Nome del file sorgente: *muldiv.asm*

- Si implementi il codice Assembly che effettui la moltiplicazione e la divisione tra i numeri 100 e 45, utilizzando le istruzioni dell'ISA e le pseudoistruzioni.

Esercizio 1.3 – Soluzione & Osservazioni

```
main:

addi $s1, $zero, 100                    # $s1 = 100
addi $s2, $zero, 45             # $s2 = 45

mult $s1, $s2                   # [Hi, Lo] = $s1 * $s2
mflo $s0                        # $s0 = Lo

move $s0, $zero                 # Reset $s0
mul $s0, $s1, $s2               # $s0 = $s1 * $s2

move $s0, $zero                 # Reset $s0
div $s1, $s2                    # Hi = $s1 % $s2, Lo = $s1 / $s2
mflo $s0                        # $s0 = Lo

addi $s0, $zero, 0              # Reset $s0
div $s0, $s1, $s2               # $s0 = $s1 / $s2
```

- SPIM implementa l'operazione div a tre operatori con un'eccezione (si osservino i valori di PC, ovvero le righe di memoria eseguite dal simulatore...).

- L'opzione *bare machine* deve essere disattivata per usare `div` a tre operatori.

Università degli Studi di Milano
Laboratorio di Architettura degli Elaboratori II
Corso di Laurea in Informatica, A.A. 2017-2018

Nicola Basilico
Dipartimento di Informatica
Via Comelico 39/41 - 20135 Milano (MI)
Ufficio S242
nicola.basilico@unimi.it
+39 02.503.16294

Hanno contribuito alla realizzazione di queste slides:

- Iuri Frosio
- Jacopo Essenziale