ISA & Assembly
Register file
Introduzione a SPIM
Operazioni aritmetiche

Laboratorio

- Massimo Marchi, marchi@di.unimi.it

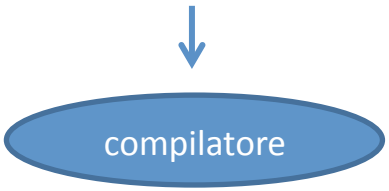
- 24 ore di esercitazione
- Modalità di esame: realizzazione e discussione di un progetto in assembly concordato con il docente

- Ricevimento da concordare via mail

Introduzione

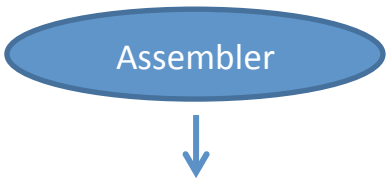
```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Linguaggio di alto livello (es. C)

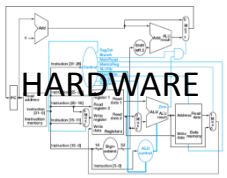


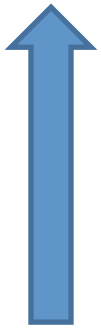

```
multi $2, $5,4
add   $2, $4,$2
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
jr    $31
```

Assembly (es. per MIPS)




```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
```





Astrazione
(complessità,leggibilità)

Instruction Set Architecture (ISA)

- Descrive l'interfaccia tra il software di basso livello e l'hardware.
- Fornisce un livello di astrazione sull'hardware
  - Cosa l'HW può fare, non come lo fa
  - Regole su come far eseguire un programma in linguaggio macchina

Cosa definisce un'ISA?

- Insieme delle istruzioni elementari
  - Classificazione in tipologie (aritmetico-logiche, memoria, salti, I/O)
  - Specifiche (pre-condizioni, post-condizioni, significato, …)
  - Formato di basso livello delle istruzioni (per la loro codifica)
- Tipi di dati nativi, loro formato
- Registri (quali, quanti, convenzioni di utilizzo)
- Uso della memoria e dell'I/O

ISA in diversi processori

- In generale, ogni architettura di processore ha un proprio linguaggio macchina.
- Due processori con lo stesso linguaggio macchina hanno la stessa ISA anche se le implementazioni hardware possono essere diverse (astrazione sull'HW).
- Grazie all'ISA sappiamo come scrivere software che, accedendo direttamente all'hardware di un calcolatore, ci permetta di programmare le sue operazioni elementari.

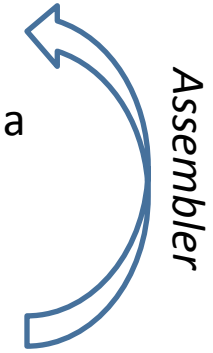
Assembly (1)

- In linguaggio macchina, una istruzione è una stringa di bit (parola di memoria), ad es.:

```
1000110010100000
```

- **Assembly** associa ad ogni istruzione in linguaggio macchina una sua **rappresentazione simbolica**, ad es.:

```
add A,B
```



- E' un linguaggio di programmazione «elementare» o «a grana fine», con limitata espressività sia in termini di operazioni che di controllo di flusso;

- La sua portabilità è soggetta alla condivisione della stessa ISA.

Assembly (2)

- I primi programmi per calcolatore venivano scritti in assembly.
- La crescente necessità di programmi più complessi ha portato all'invenzione dei linguaggi di alto livello (come il C).
- Oggi? (Di solito) è il compilatore che traduce il linguaggio di alto livello in assembly.
- *Esempio, somma dei primi 100 interi positivi in* C*:*

```
main()
{
        int i;
        int sum = 0;
        for (i = 0; i <= 100; i = i + 1)
         sum = sum + i*i;
        printf("La somma da 0 a 100 è %d\n",sum);
}
```

```
main:
        addu $t7, $t6, 1
        subu $sp, $sp, 32
        sw $ra, 20($sp)
        sw $a0, 32 ($sp)
        sw $0, 24($sp)
        sw $0, 28($sp)
loop:
        lw  $t6, 28($sp)
        …
```

Assembly (3)

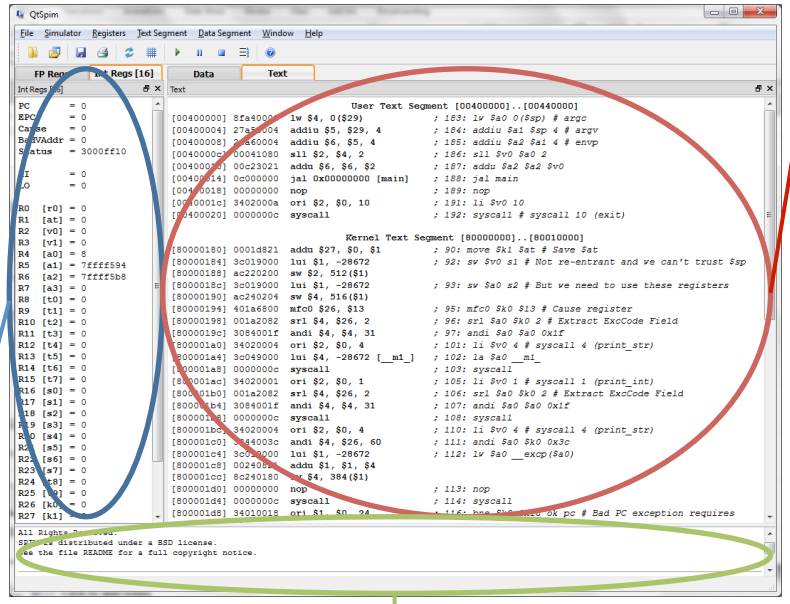
- Durante questo laboratorio:
  - Scriveremo codice assembly per MIPS e lo faremo eseguire ad un simulatore (SPIM), tipicamente partendo da una specifica di alto livello.
  - Aspetti da tenere presente: maggiore numero di linee, molto più difficile identificare i bug.
- Perché programmare in assembly?
  - E' uno strumento che permette un controllo fine sulle prestazioni e potenzialità dell'hardware
  - «hands on» il livello più basso di un calcolatore
- Strumenti che utilizzeremo:
  - ISA MIPS e suo simulatore SPIM

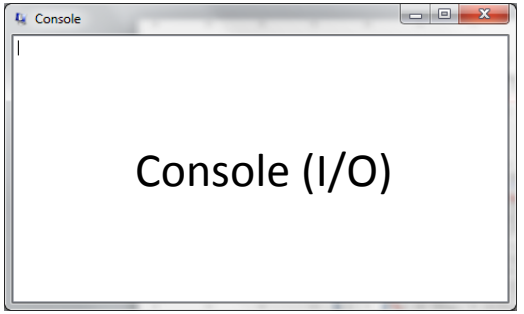
SPIM

- E’ un simulatore di una CPU che obbedisce alle convenzioni MIPS
- Perchè 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)

SPIM (user interface)

Memoria (Text e Data)

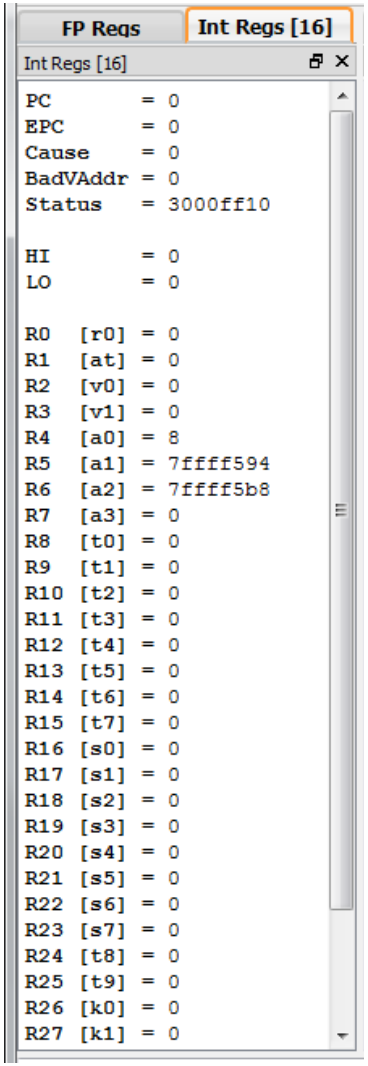




Banco Registri

Logs

SPIM (Registri)



- 32 registri a 32bit per operazioni su interi (**$0..$31**)
- 32 registri a 32 bit per operazioni in virgola mobile (**$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** vengono usati nella gestione delle eccezioni.

- NB **1 WORD = 32 bit**
- I registri general-purpose sono chiamati R0, R1, … R31, tra [] c'è il nome secondo la convenzione MIPS
- Il loro è sempre visibile durante l'esecuzione di un programma, può essere visualizzato secondo di verse codifiche (Register -> <binary|hex|decimal>)

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.

Spim (Memoria)

- Un programma Assembly è composto da due elementi distinti, che risiedono nella RAM del calcolatore:
  - Segmento testo (Tet:, a partire da indirizzo 0x00400000)

indirizzo codice Rappr. simbolica Linea nel source file

```
[00400000] 8fa40000  lw $4, 0($29)           ; 183: lw $a0 0($sp) # argc
```

  - Segmento dati (Data:, a partire da indirizzo 0x10000000)

```
User Stack [7ffff590]..[80000000]
[7ffff590]    00000008  7ffff696  7ffff686  7ffff67f    . . . . . .
```

indirizzo dati Rappr. stringa

- Il programma Assembly viene caricato da un file sorgente e rilocato a partire dall'indirizzo 0x00400024 in Text

- Dall'indirizzo 0x00400000 a 0x00400024 c'è un preambolo di inizializzazione, una chiamata all'istruzione che sta alla label **main** e infine, una volta concluso main, la syscall exit.

Richiamo di istruzioni aritmetiche (somma, sottrazione)

- add $s1, $s2, $s3 # $s1 = $s2 + $s3, oveflow detected
- sub $s1, $s2, $s3 # $s1 = $s2 - $s3, oveflow detected
- addi $s1, $s2, 13 # $s1 = $s2 + cost, oveflow detected
- addu $s1, $s2, $s3 # $s1 = $s2 + $s3, unsigned, oveflow undetected
- subu $s1, $s2, $s3 # $s1 = $s2 - $s3, unsigned, oveflow undetected
- addui $s1, $s2, 27 # $s1 = $s2 + cost, unsigned, oveflow undetected

Es. 1.1

- 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 in un file di testo*
- *(2) caricare il file in SPIM e osservare il segmento testo*
- *(3) lanciare l'esecuzione con* F5, cosa succede?
- *(4) osservare come variano i registri coinvolti nelle operazioni*
- *(5) ripetere mediante l'uso di un breakpoint, aggiornare il source file, re-inizializzare il simulatore e ricominciare da (2)*

Es. 1.2

- 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 valori iniziali 1, 2, 3 e 4

Es. 1.2 Soluzione e osservazioni

- 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 $t0 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 (HW 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`**

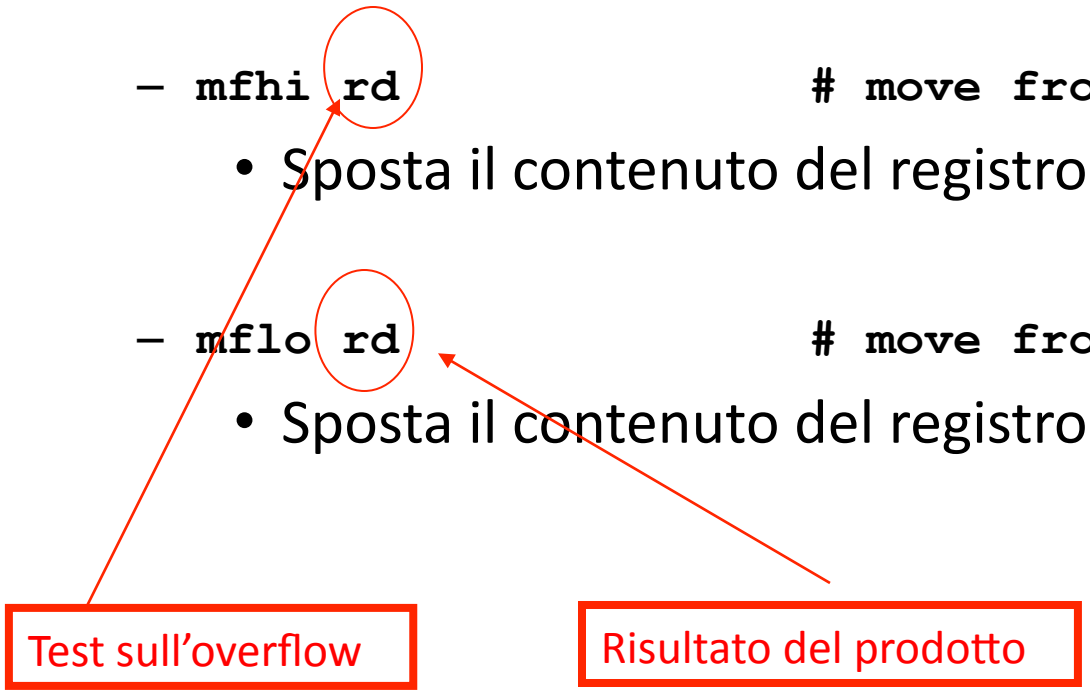
- Il registro destinazione è ***implicito.***

- 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: moltiplicazione

- mult $t0 $t1 # [Hi, Lo] = $t0 * $t1
- multu $t0 $t1 # [Hi, Lo] = $t0 * $t1

- Le operazioni per la moltiplicazione utilizzano due registri in ingresso (32 bit per registro);
- Il risultato della moltiplicazione (a 64 bit) viene posto in due registri dedicati, Hi (High order word) e Lo (Low order word).
- Se il registro Hi contiene un numero maggiore di 0, il risultato della moltiplicazione eccede i 32 bit e non potrà essere copiato in un registro (overflow).

Operazioni aritmetiche: moltiplicazione

- Il risultato della moltiplicazione viene prelevato dal registro Hi dal registro Lo utilizzando:

- mfhi $s5 # $s5 = hi, test overflow
- mflo $s4 # $s4 = lo, risultato (32 bit)

- E presente anche la versione unsigned della moltiplicazione (multu).

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 mflo.

Pseudo istruzioni

- Per facilitare il compito del programmatore, vengono introdotte delle pseudo-istruzioni.
- Ad una pseudo-istruzione, corrisponde una sequenza di (una o) più istruzioni dell'ISA.

- **move $t0, $t1 # $t0←$t1**
  corrisponde a: add $t0, $zero, $t1

- **mul $s0, $t1, $t2 # $t0 ← $t1 * $t2**
  corrisponde a: mult $t1, $t2;
  mflo $s0

- **div $s0, $t1, $t2 # $t0 ← $t1 / $t2**
  corrisponde a:div $t1, $t2
  mflo $t0

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

- Si implementi il codice Assembly che effettua 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 ossevino i valori di PC, ovvero le righe di memoria eseguite dal simulatore...)
- Att.ne! L'opzione bare machine deve essere disattiva per usare div a tre operatori.