

01: LABORATORIO 01

**ISA & ASSEMBLY
REGISTER FILE
INTRODUZIONE A SPIM
OPERAZIONI ARITMETICHE**

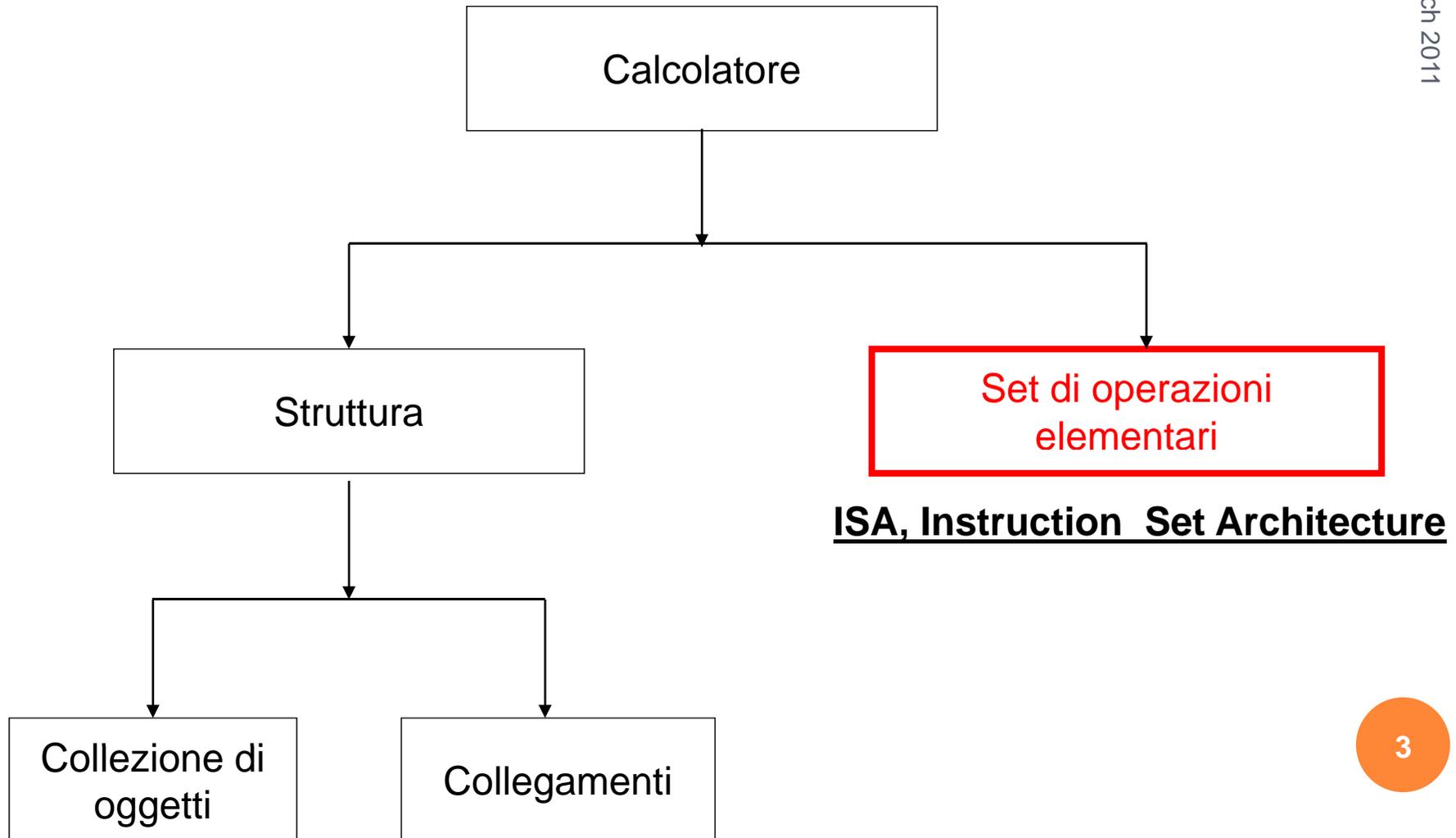
I. Frosio

SOMMARIO

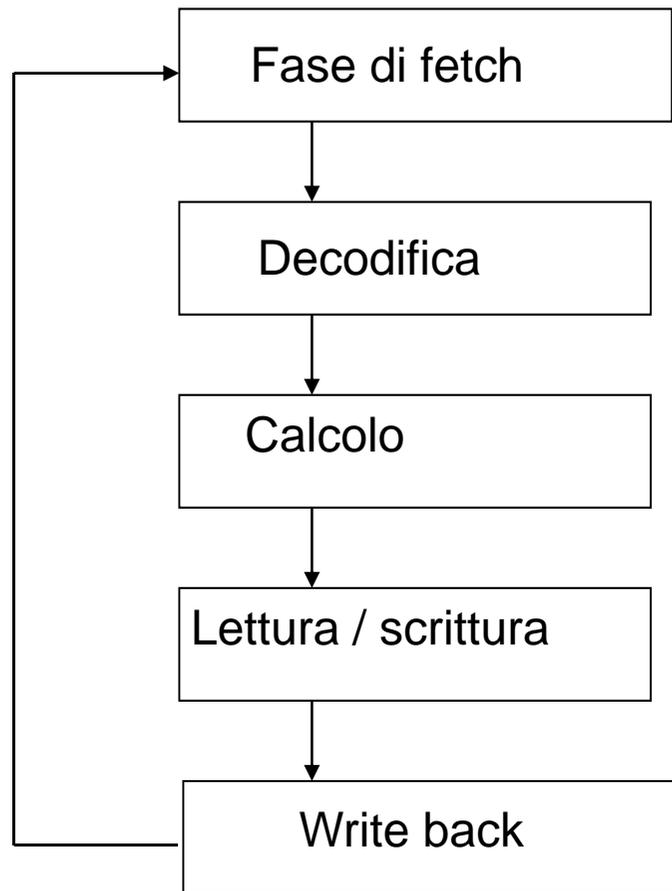
- ISA & Linguaggio macchina ←
- Assembly
- I registri
- Istruzioni aritmetiche
- Obiettivo
- PCSpim
- Operazioni aritmetiche

ISA & LINGUAGGIO MACCHINA: DESCRIZIONE DI UN ELABORATORE

18 March 2011



ISA & LINGUAGGIO MACCHINA: CARATTERISTICHE DI UN'ISA



Formato e codifica di un'istruzione
– tipi di formati e dimensione delle istruzioni.

Posizione degli operandi e risultato.
– quanti?
– dove? (memoria e/o registri)

Tipo e dimensione dei dati

Operazioni consentite

ISA & LINGUAGGIO MACCHINA: DEFINIZIONE DI UN'ISA

Definizione del funzionamento: insieme delle istruzioni (interfaccia verso i linguaggi ad alto livello).

- Tipologia di istruzioni.
- Meccanismo di funzionamento.

Definizione del formato: codifica delle istruzioni (interfaccia verso l'HW).

- Formato delle istruzioni.
- Suddivisione in gruppi omogenei dei bit che costituiscono l'istruzione.

ISA & LINGUAGGIO MACCHINA: TIPI DI ISTRUZIONE

- Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:
 - Istruzioni aritmetico-logiche;
 - Istruzioni di trasferimento da/verso la memoria (*load/store*);
 - Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
 - Istruzioni di trasferimento in ingresso/uscita (I/O).

ISA & LINGUAGGIO MACCHINA: ISTRUZIONI DI UN'ISA

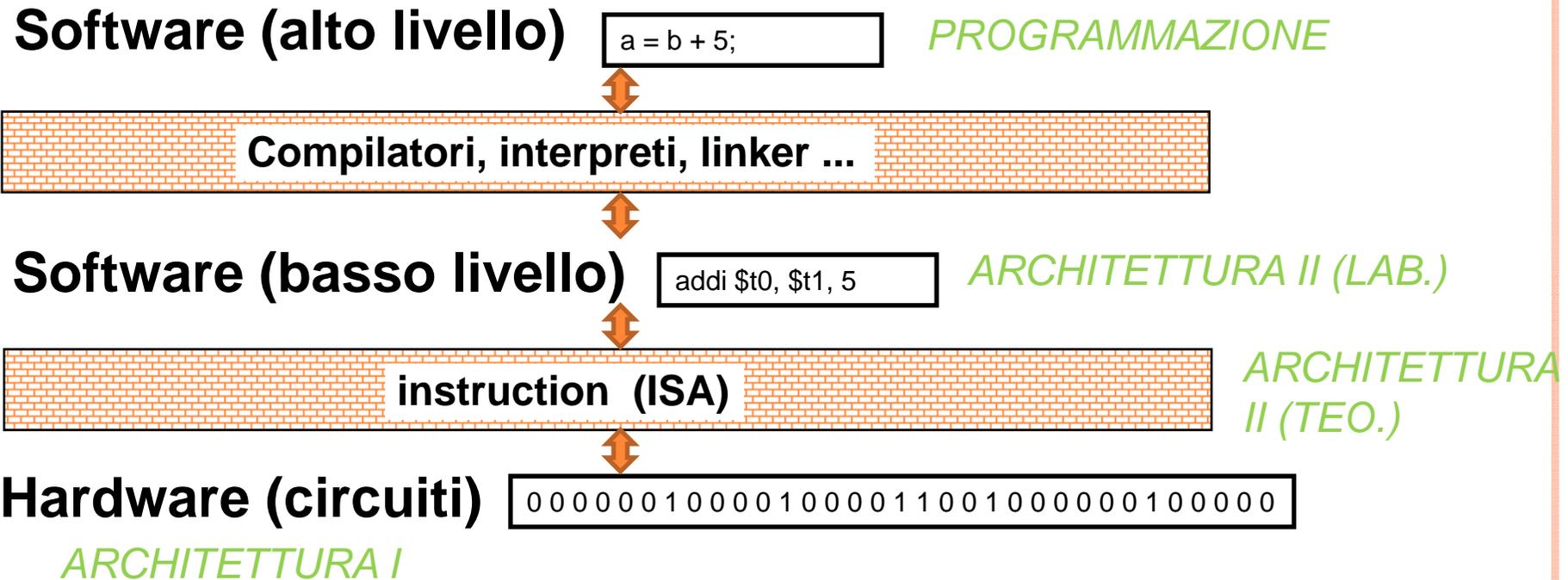
Devono contenere tutte le informazioni necessarie ad eseguire il ciclo di esecuzione dell'istruzione: registri, comandi,

Ogni architettura di processore ha il suo linguaggio macchina

- Architettura dell'insieme delle istruzioni elementari messe a disposizione dalla macchina (in linguaggio macchina).
 - **ISA (Instruction Set Architecture)**
- Due processori con lo stesso linguaggio macchina hanno la stessa architettura delle istruzioni anche se le implementazioni hardware possono essere diverse.
- Consente al SW di accedere direttamente all'hardware di un calcolatore.

L'architettura delle istruzioni, specifica come vengono costruite le istruzioni in modo tale che siano comprensibili alla macchina (in linguaggio macchina).

ISA & LINGUAGGIO MACCHINA: CIRCUITI, ISA, PROGRAMMAZIONE AD ALTO LIVELLO



Quale è più facile modificare?

SOMMARIO

- ISA & Linguaggio macchina
- Assembly ←
- I registri
- Istruzioni aritmetiche
- Obiettivo
- PCSpim
- Operazioni aritmetiche 2

ASSEMBLY: LE ISTRUZIONI IN LINGUAGGIO MACCHINA

- Linguaggio di programmazione direttamente comprensibile dalla macchina
 - Le parole di memoria sono interpretate come *istruzioni*
 - Vocabolario è *l'insieme delle istruzioni (instruction set)*

**Programma in
linguaggio ad alto
livello (C)**

```
a = a + c  
b = b + a  
var = m [a]
```

**Programma in
linguaggio
macchina**

```
011100010101010  
000110101000111  
000010000010000  
001000100010000
```

ISTRUZIONI DELL'ISA & ASSEMBLY

- Linguaggio ad alto livello (C)

```
for (i=0; i<100; i++){  
    a[i]=false;  
    if ((i%3)==0)  
        a[i] = true;  
    printf("i = %d.\n",i);  
}
```

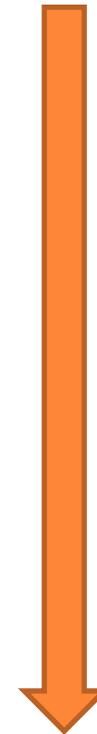
- Assembly

```
Init:  
add $t0, $t0, $s3      # $t0 = $t0 + $s3  
sw $a0, 32($sp)       # save in memory  
J Init                 # Jump to init
```

- Linguaggio macchina

```
00001010100000110111011101110110  
1000010101001010101001010010100  
01010101010101010101010101011000
```

Comprensibile per un
umano



Comprensibile per un
calcolatore

ISTRUZIONI DELL'ISA & ASSEMBLY

- Le istruzioni Assembly sono una rappresentazione simbolica del linguaggio macchina comprensibile dall'HW.
- Le forme di controllo del flusso sono limitate rispetto ai linguaggi ad alto livello.
- E' a tutti gli effetti un linguaggio di programmazione che fornisce la visibilità diretta sull'HW.
- Il codice può essere utilizzato solo su macchine che condividono l'ISA.

ASSEMBLY: LINGUAGGIO C, SOMMA DEI PRIMI CENTO NUMERI NATURALI (AL QUADRATO)

```
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);  
}
```

ASSEMBLY: LINGUAGGIO ASSEMBLY, SOMMA DEI PRIMI CENTO NUMERI NATURALI (AL QUADRATO)

```
.text
.align 2
.globl main
main:
    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)
      lw $t8, 24($sp)
      mult $t4, $t6, $t6
      addu $t9, $t8, $t4
      addu $t9, $t8, $t7
      sw $t9, 24($sp)
      addu $t7, $t6, 1
      sw $t7, 28($sp)
      bne $t5, 100, loop
      .....
```

ASSEMBLY: COME LINGUAGGIO DI PROGRAMMAZIONE

- Principali *svantaggi* della programmazione in linguaggio assembly:
 - Mancanza di portabilità dei programmi su macchine diverse
 - Maggiore lunghezza, difficoltà di comprensione, facilità d'errore rispetto ai programmi scritti in un linguaggio ad alto livello

- Principali *vantaggi* della programmazione in linguaggio assembly:
 - Ottimizzazione delle prestazioni.
 - Massimo sfruttamento delle potenzialità dell'hardware sottostante.

- Le strutture di controllo hanno forme limitate
- Non esistono tipi di dati all'infuori di interi, virgola mobile e caratteri.
- La gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito dal programmatore.

ASSEMBLY: COME LINGUAGGIO DI PROGRAMMAZIONE

- Alcune applicazioni richiedono un approccio *ibrido* nel quale le parti più critiche del programma sono scritte in assembly (per massimizzare le prestazioni) e le altre parti sono scritte in un linguaggio ad alto livello (le prestazioni dipendono dalle capacità di ottimizzazione del compilatore).

Esempio: Sistemi embedded o dedicati

Sistemi “automatici” di traduzione da linguaggio ad alto livello (linguaggio C) ad assembly e codice binario ed implementazione circuitale (e.g. sistemi di sviluppo per FPGA).

SOMMARIO

- ISA & Linguaggio macchina
- Assembly
- I registri ←
- Istruzioni aritmetiche
- Obiettivo
- PCSpim
- Operazioni aritmetiche 2

MIPS

- Il MIPS è una CPU **RISC** ad architettura **load-store** che sfrutta massicciamente il **pipelining**.
- Set di istruzioni ridotto (Reduced Instruction Set Cpu), limitato ad istruzioni che sfruttano pienamente la struttura a fasi successive (*pipeline*);
 - Es. il caricamento di una costante in un registro è ottenuto con una sequenza equivalente di operazioni aritmetiche.
- Per analoghi motivi le operazioni da e verso la memoria sono limitate al solo caricamento e salvataggio dei registri (*load-store*).
 - Non è possibile sommare direttamente un dato contenuto in memoria con un registro: occorre prima spostare il dato in memoria in un registro della CPU.

REGISTRI: INTRODUZIONE

- I registri sono associati (temporaneamente) alle variabili di un programma dal compilatore.
- Un processore possiede un numero limitato di registri: ad esempio il processore MIPS possiede **32 registri composti da 32-bit (word), register file.**
- I registri possono essere direttamente indirizzati mediante il loro numero progressivo (0, ..., 31) preceduto da \$: ad es. **\$0, \$1, ..., \$31**
- Per convenzione di utilizzo, sono stati introdotti nomi simbolici significativi. Sono preceduti da \$, ad esempio:
\$s0, \$s1, ..., \$s7 (\$s8) Per indicare variabili in C
\$t0, \$t1, ... \$t9 Per indicare variabili temporanee

REGISTRI: USO E CONVENZIONI

	Nome	Numero	Utilizzo
→	\$zero	0	costante zero
	\$at	1	riservato per l'assemblatore
	\$v0-\$v1	2-3	valori di ritorno di una procedura
	\$a0-\$a3	4-7	argomenti di una procedura
→	\$t0-\$t7	8-15	registri temporanei (non salvati)
→	\$s0-\$s7	16-23	registri salvati
→	\$t8-\$t9	24-25	registri temporanei (non salvati)
	\$k0-\$k1	26-27	gestione delle eccezioni
	\$gp	28	puntatore alla global area (dati)
	\$sp	29	stack pointer
	\$s8	30	registro salvato (fp)
	\$ra	31	indirizzo di ritorno

REGISTRI (MIPS)

- 32 registri general-purpose a 32bit per operazioni su interi (**\$0..\$31**)
- 32 registri general-purpose per operazioni in virgola mobile a 32bit (**\$FP0..\$FP31**)
 - Per le operazioni in doppia precisione si utilizzano coppie di registri contigui (**\$FP0, \$FP2, \$FP4, ...**)
- registri speciali, sempre a 32bit, usati per compiti particolari (esempi:
 - il **Program Counter (PC)** contiene l'indirizzo dell'istruzione da eseguire
 - **HI** e **LO** usati nella moltiplicazione e nella divisione
 - **Status** contiene i flag di stato della CPU.
- NB 1 **WORD** = 32 bit

REGISTRI (MIPS) - CONVENZIONI

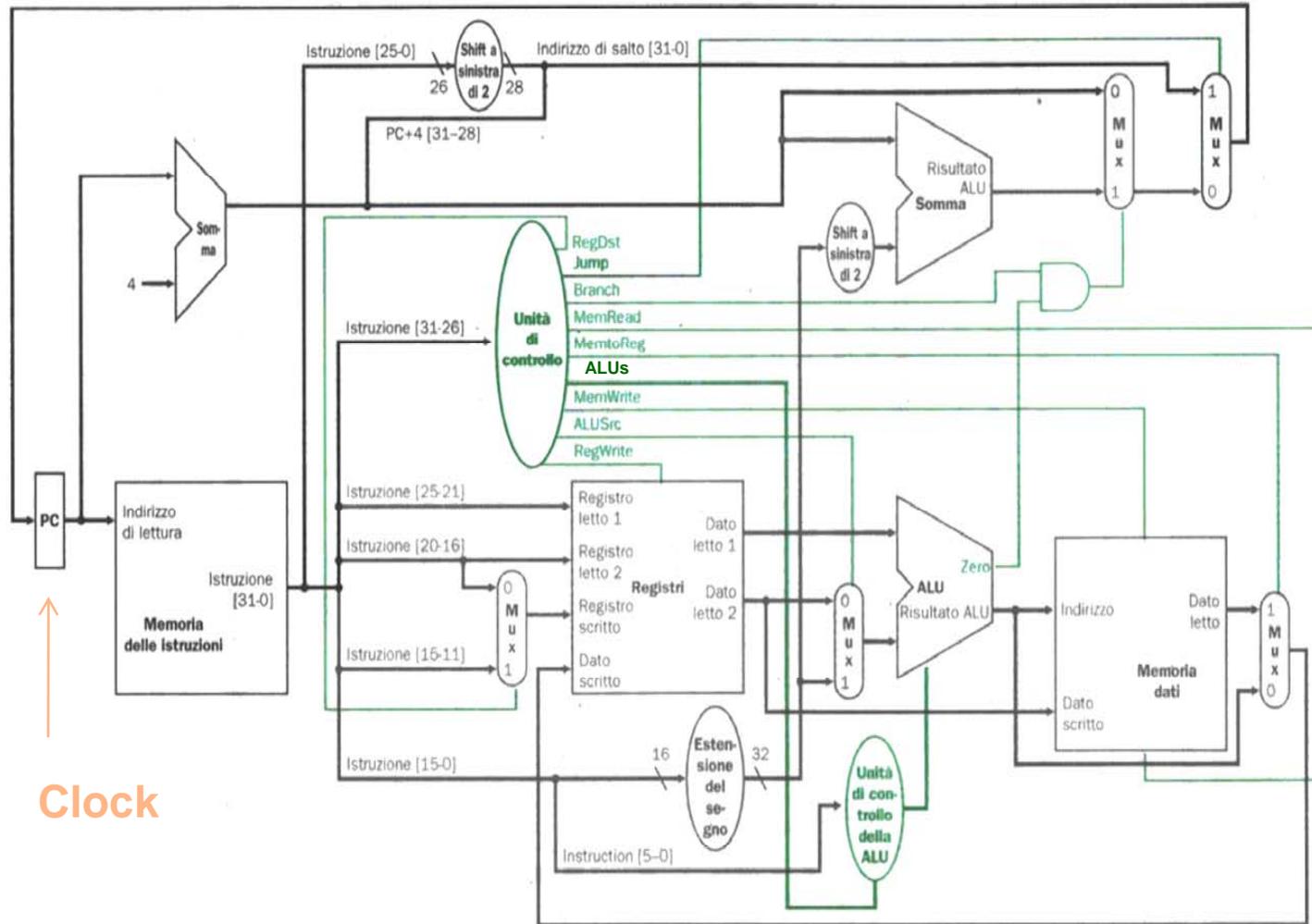
18 March 2011

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 **\$0 (\$zero)** settato costantemente al valore **0**

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

CPU + UC A SINGOLO CICLO



18 March 2011

SOMMARIO

- ISA & Linguaggio macchina
- Assembly
- I registri
- Istruzioni aritmetiche ←
- Obiettivo
- PCSpim
- Operazioni aritmetiche 2

OPERAZIONI ARITMETICHE

- In MIPS, un'istruzione aritmetico-logica prevede in genere 3 operandi:
 - Due registri contenenti i valori da elaborare (*registri sorgente*)
 - Un registro per il risultato (*registro destinazione*)
- L'ordine degli operandi è fisso:
 - <risultato, operando 1, operando 2>
- L'istruzione Assembly viene composta con il codice operativo dell'operazione e i tre campi relativi agli operandi:
 - <codice operazione> <risultato, operando 1, operando 2>

OPCODE DEST, SORG1, SORG2

NB: Le operazioni vengono eseguite esclusivamente su dati presenti nella CPU, non su dati residenti nella memoria!

OPERAZIONI ARITMETICHE (ADD)

- C vs Assembly:

- $R = A + B;$

- `add $s0, $s1, $s2`

Commento

`# $s0 ← $s1 + $s2`

- Il compilatore effettua la traduzione da linguaggio ad alto livello ad assembly, assegnando automaticamente i registri alle variabili.

OPERAZIONI ARITMETICHE (ADD IMMEDIATE)

- Un'operazione molto frequente è l'addizione di costanti;
- Per tale motivo l'ISA del MIPS-32 prevede un'istruzione particolare per effettuare tale operazione:
 - `addi $t0, $t1, 50` # $\$t0 \leftarrow \$t1 + 50$
 - I registri possono comparire sia come risultato che come operando, ad esempio:
 - `addi $t0, $t0, 1` # $\$t0++$

ISTRUZIONI ARITMETICO-LOGICHE: SEQUENZA

Il fatto che ogni istruzione aritmetica ha tre operandi sempre nella stessa posizione consente di semplificare l'hw, ma complica alcune cose...

Codice C: $Z = A - (B + C + D) \Rightarrow$
 $E = B + C + D; Z = A - E;$

Suppongo che le variabili siano contenute nei seguenti registri:

A -> \$s0 B -> \$s1 C -> \$s2 D -> \$s3 Z -> \$s5

Codice MIPS: `add $t0, $s1, $s2`
 `add $t1, $t0, $s3`
 `sub $s5, $s0, $t1`

ISTRUZIONI: ARITMETICO-LOGICHE, IMPLEMENTAZIONE ALTERNATIVA

- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma e sottrazione delle variabili A . . D nella variabile Z servono tre istruzioni :

Codice C: $Z = (A + B) - (C - D)$

codice MIPS:

```
add $t0, $s0, $s1
sub $t1, $s2, $s3
sub $s5, $t0, $t1
```

Quale implementazione è la migliore? Sceglierà il compilatore il quale cerca di massimizzare la parallelizzazione del codice.

ISTRUZIONI: ADD, VARIANTI

- `addi $s1, $s2, 100` `#add immediate`
 - Somma una costante: il valore del secondo operando è presente nell'istruzione come costante e sommata estesa in segno.
 $rt \leftarrow rs + \text{costante}$
- `addu $s0, $s1, $s2` `#add unsigned`
 - Evita overflow: la somma viene eseguita considerando gli addendi sempre positivi. Il bit più significativo è parte del numero e non è bit di segno.
- `addiu $s0, $s1, 100` `#add immediate unsigned`
 - Somma una costante ed evita overflow.

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**

Test sull'overflow

Risultato del prodotto



SOMMARIO

- ISA & Linguaggio macchina
- Assembly
- I registri
- Istruzioni aritmetiche
- Obiettivo 
- PCSpim
- Operazioni aritmetiche 2

OBIETTIVO

- Sia data la seguente riga di codice in linguaggio ad alto livello (es. C, Java):

$$A[100] = 5 + B[i] + C \quad (1),$$

dove A, B sono array mentre C, i sono variabili scalari.

- Sia dia la rappresentazione assembler (secondo le specifiche MIPS-32) delle istruzioni necessarie per eseguire (1).

OBIETTIVO: ISTRUZIONI DELL'ISA & ASSEMBLY

- Aritmetico – logiche (and, or, nor, ...)

[Prossime lezioni:

- *Trasferimento da / verso la memoria (load ,store)*
- *Salto condizionato / non condizionato (controllo di flusso del programma)*
- *Trasferimento in ingresso / uscita (I/O)]*

SOMMARIO

- ISA & Linguaggio macchina
- Assembly
- I registri
- Istruzioni aritmetiche
- Obiettivo
- PCSpim ←
- Operazioni aritmetiche 2

PCSPIM

- Simulatore (Windows, Linux e Mac in diverse versioni) di CPU MIPS.
- E' possibile caricare programmi in linguaggio assembly MIPS, tradurli in linguaggio macchina ed eseguirli:
 - in modalità passo-passo (viene eseguita una sola istruzione per volta – Tasto F10)
 - in modalità continua (il codice viene eseguito senza interruzioni – Tasto F5).
- E' possibile vedere prima, durante e dopo l'esecuzione lo stato dei registri ed alterarli se necessario.
- Sono disponibili un certo numero di funzioni predefinite, le **syscall** (es. per I/O).

PCSPIM

18 March 2011

The screenshot shows the PCSpim simulator interface. At the top, the status bar displays: PC = 00000000, EPC = 00000000, Cause = 00000000, BadVAddr = 00000000, Status = 3000ff10, HI = 00000000, LO = 00000000.

Area registri (Registers):

R0 (r0) = 00000000	R8 (t0) = 00000000	R16 (s0) = 00000000	R24 (t8) = 00000000
R1 (at) = 00000000	R9 (t1) = 00000000	R17 (s1) = 00000000	R25 (t9) = 00000000
R2 (v0) = 00000000	R10 (t2) = 00000000	R18 (s2) = 00000000	R26 (k0) = 00000000
R3 (v1) = 00000000	R11 (t3) = 00000000	R19 (s3) = 00000000	R27 (k1) = 00000000
R4 (a0) = 00000000	R12 (t4) = 00000000	R20 (s4) = 00000000	R28 (gp) = 10008000

Area codice (Assembly, LM) (Assembly code):

```
[0x00400000] 0x8fa40000 lw $4, 0($29) ; 183: lw $a0 0($sp) # argc
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 184: addiu $a1 $sp 4 # argv
[0x00400008] 0x24a60004 addiu $6, $5, 4 ; 185: addiu $a2 $a1 4 # envp
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 2
[0x00400014] 0x0c000000 jal 0x00000000 [main] ; 188: jal main
[0x00400018] 0x00000000 nop ; 189: nop
[0x0040001c] 0x3402000a ori $2, $0, 10 ; 191: li $v0 10
```

Area dati (data, stack, kernel) (Data, Stack, Kernel):

DATA
[0x10000000]...[0x10040000] 0x00000000

STACK
[0x7ffff58c] 0x00000000

KERNEL DATA
[0x90000000] 0x78452020 0x74706563 0x206e6f69 0x636f2000

Area log (Log):

```
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
DOS and Windows ports by David A. Carley.
Copyright 1997, Morgan Kaufmann Publishers.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
```

Area console (I/O) (Console):

The console window is currently empty.

For Help, press F1 PC=0x00000000 EPC=0x00000000 Cause=0x00000000

PCSPIM

- Un programma Assembly è composto da due elementi distinti, che risiedono nella RAM del calcolatore:
 - Segmento dati [data:, da 0x10000000]
 - Segmento codice o testo [text:, da 0x00400024]
- L'area codice effettivamente definita da PCSpim parte da 0x00400000.
- Il programma Assembly viene caricato e rilocato a partire dalla locazione 0x00400024 poiché PCSpim, prima di lanciare effettivamente il programma, effettua alcune operazioni di servizio.
- Dalla locazione 0x00400000 alla 0x00400024, è presente un frammento di codice che prepara i registri per permettere al programma di accedere a dati definibili dall'utente a run-time (dati passati al programma al momento del lancio tramite una finestra di dialogo).
- Successivamente chiama l'indirizzo **main:** (0x00400024) con una chiamata a sub-routine per permettere un'uscita dall'esecuzione del programma "elegante".

SOMMARIO

- ISA & Linguaggio macchina
- Assembly
- I registri
- Istruzioni aritmetiche
- Obiettivo
- PCSpim
- Operazioni aritmetiche 2 ←

Es. 1.1

- Si scriva il codice Assembly che mette il valore 5 nel registro \$s1, il valore 7 nel registro \$s2 e la somma dei due nel registro \$s0. Si utilizzi PCSpim per verificare la correttezza del codice implementato e si osservi come varia il valore dei registri durante l'esecuzione del codice.
- Hint:
 - Il codice viene scritto utilizzando un editor di testo (es. notepad).
 - Il file viene aperto da PCSpim mediante il tasto Open...
 - E' necessario inserire la label main: all'inizio del codice implementato (solo in questo modo PCSpim colloca correttamente in memoria la porzione text).
 - Il codice viene eseguito passo passo utilizzando il tasto F10.

Es. 1.1 – SOLUZIONE & OSSERVAZIONI

```
main: addi $s1, $zero, 5    # $s1 = 5
      addi $s2, $zero, 7    # $s2 = 7
      add  $s0, $s1, $s2    # $s0 = $s1 + $s2
```

“Filosofia RISC” → L’assegnazione di un valore ad un registro avviene utilizzando addi (instruction set ridotto, circuiti efficienti ma programmazione complessa)

Es. 1.1 – SOLUZIONE & OSSERVAZIONI

- PCSpim esegue le righe di codice da 0x00400000 a 0x00400014 prima di eseguire il nostro codice.
- L'etichetta "main:" all'inizio del codice è indispensabile in quanto PCSpim mappa l'etichetta "main:" in posizione 0x00400024; in posizione 0x00400014, troviamo l'istruzione jal 0x00400024 (salto incondizionato a main:);
- Si noti che quanto viene eseguita questa operazione, il registro R31 (ra, return address) viene aggiornato con l'indirizzo di ritorno.
- Il registro PC viene costantemente aggiornato con l'indirizzo in memoria dell'istruzione corrente.

OPERAZIONI ARITMETICHE (SOMMA, SOTTRAZIONE)

- `add $s1, $s2, $s3` # $\$s1 = \$s2 + \$s3$, overflow detected
- `sub $s1, $s2, $s3` # $\$s1 = \$s2 - \$s3$, overflow detected
- `addi $s1, $s2, 13` # $\$s1 = \$s2 + \text{cost}$, overflow detected
- `addu $s1, $s2, $s3` # $\$s1 = \$s2 + \$s3$, unsigned, overflow undetected
- `subu $s1, $s2, $s3` # $\$s1 = \$s2 - \$s3$, unsigned, overflow undetected
- `addii $s1, $s2, 27` # $\$s1 = \$s2 + \text{cost}$, unsigned, overflow undetected

Es. 1.2

- Si traduca in Assembly la seguente riga di codice:
 $A = B + C - (D + E)$,
Supponendo che alle variabili A, ..., E vengano assegnati i registri \$s0, ..., \$s4.
- Si assegnino inizialmente i valori 1, 2, 3 e 4 alle variabili B, ..., E e si commenti il risultato ottenuto con PCSpim.

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 0xfffffc;
- prova:
 - $(1+2)-(3+4) = 3-7 = -4$
 - $0xfffffc = [1111,1111,1111,1111,1111,1111,1111,1100]_{C2} = \dots$
 - $\dots = -\{[0000,0000,0000,0000,0000,0000,0000,0011]+1\}_2 = -\{100\}_2 = -4$

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.

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 -> $\$s2 \bmod \$s3$ [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

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.