Università degli Studi di Milano «La Statale»
Dipartimento di Informatica

Lezione 12:

CPU e linguaggio MIPS

Parte B: le istruzioni di tipo «R»

Marco Tarini

19

Piano delle lezione: verso un'ISA e la relativa CPU

Continuiamo la nostra progettazione di vari aspetti, di pari passo:

- Un **Istruction Set**: un insieme di possibili istruzioni (in **linguaggio macchina**) che possiamo usare per esprimere i nostri **programmi** ("**binari**")
  - **programma** (*binario*) = sequenza di istruzioni (*in linguaggio macchina*)
- Per ogni istruzione, una **sintassi** (come si scrive, in linguaggio macchina, come serie di 0 e 1) e la relativa **semantica** (che cosa fa)
- *Più a basso livello*: la progettazione di una **architettura** HW, cioè di una **CPU** in grado di intepretare le istruzioni dell IS
  - il ciclo **fetch and execute** di un programma scritto in quel linguaggio macchina
- *Più ad alto livello*: l'equivalente in linguaggio **assembly** per esprimere le stesse istruzioni in modo più comprensibile per un essere umano
  - Che l'**assembler** tradurrà in Linguaggio macchina
  - Vedremo anche note sulla programmazione in questo linguaggio assembly

Come annunciato, per tutti i nostri esempi ci riferiremo ad un ISA esistente: il **MIPS.**

Oggi vediamo (quasi) tutte le istruzioni "**di tipo R**": operazioni (logiche o matemematiche) fra registri

20

Esempio di una istruzione MIPS in linguaggio macchina

0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0

Operazione fra registri	6	11	31	---	SUM
OPCODE	*RS*	*RT*	*RD*	*---*	*FUNCTION*

Tradotta in assembly MIPS: `ADD $31, $6, $11`

A parole: *« Somma il Registro 11 e il Registro 6 e memorizza il risultato nel Registro 31 »*

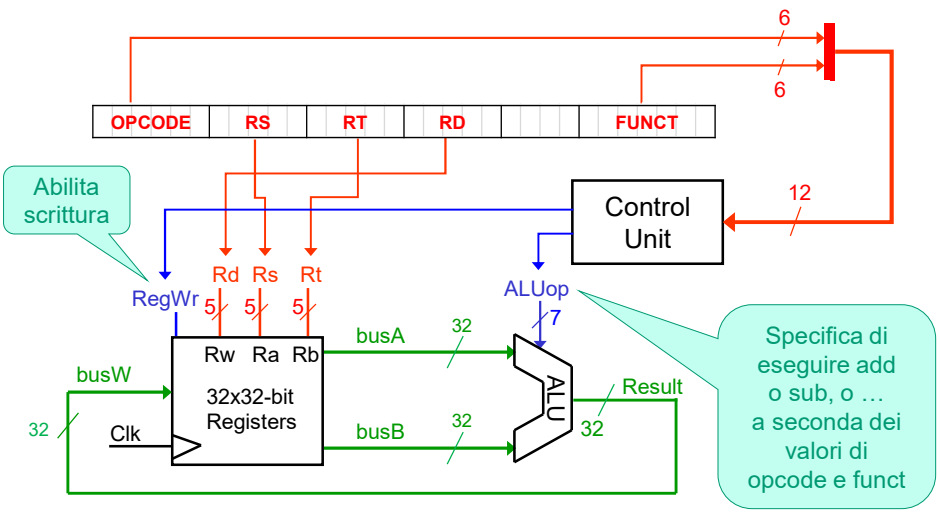
21

Sintassi di linguaggio macchina: le istruzioni di tipo R

- Hanno il campo Opcode (i 6 bit più significativi – cioè quelli a sinistra) a tutti 0 (000000)
- Il resto dell'istruzione (che in MIPS è sempre di 32 bit) è divisa in vari campi, come mostrato
- In particolare, i 6 bit meno significativi (quelli più a destra) identificano quale sia l'operazione logica o matematica richiesta

22

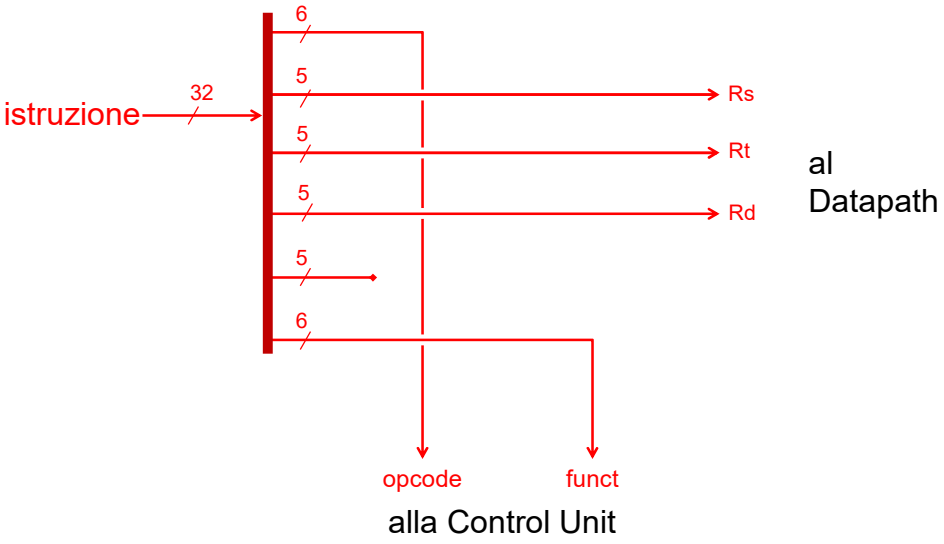
Datapath per operazioni fra registri



- 23 -

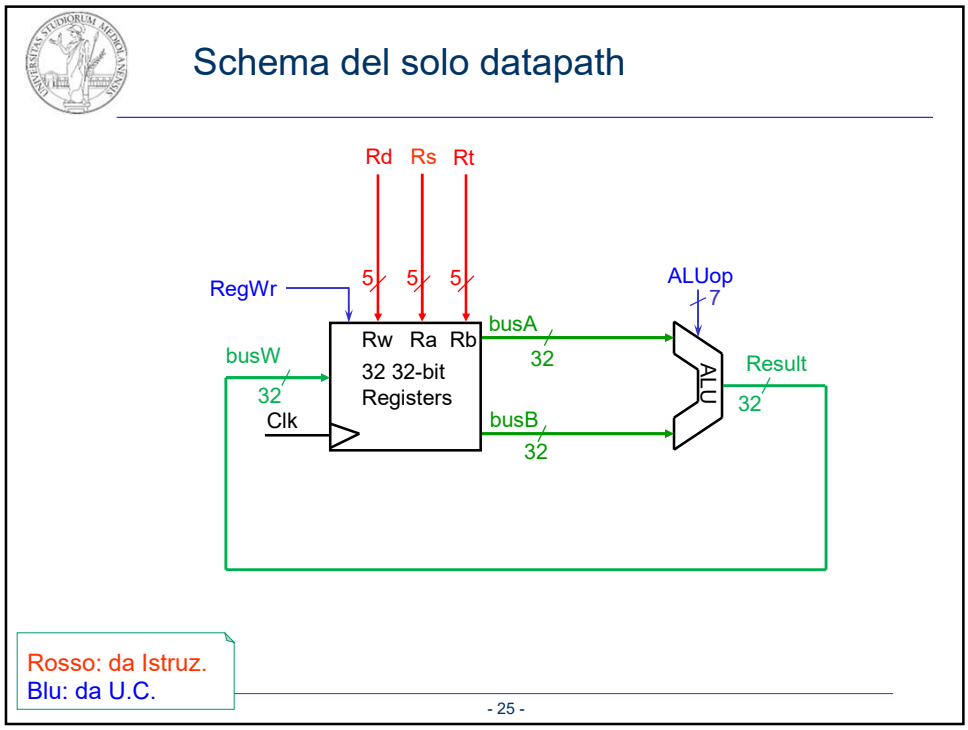
23

Suddividere l'istruzione MIPS in campi: (per le istruzioni di tipo "R")



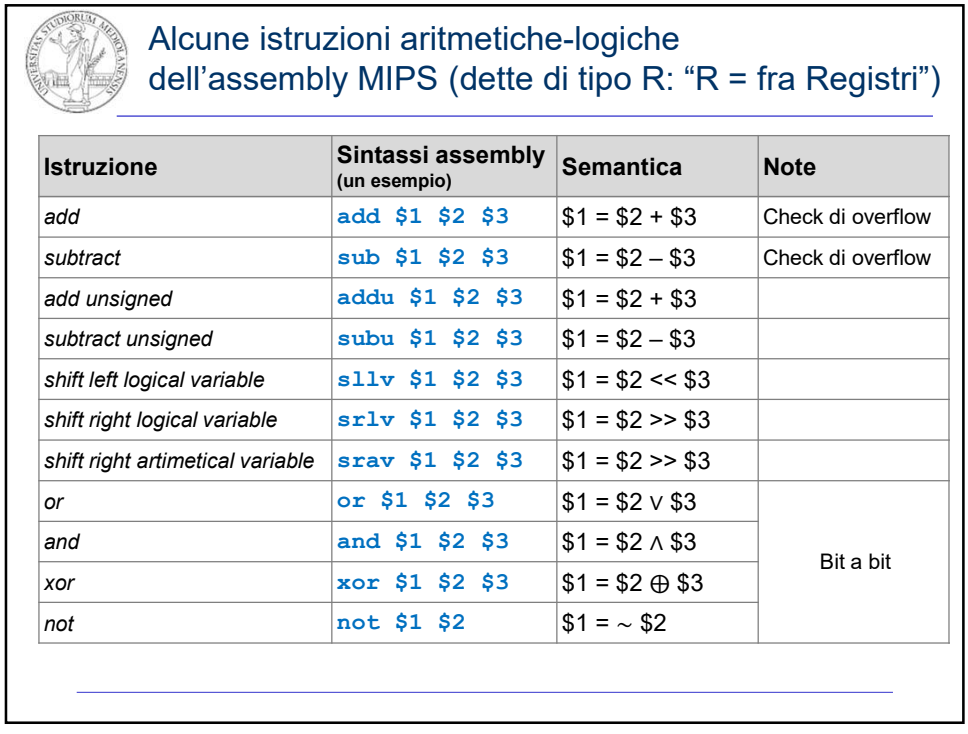
- 24 -

24



Schema del solo datapath

25



Alcune istruzioni aritmetiche-logiche dell'assembly MIPS (dette di tipo R: "R = fra Registri")

Istruzione	**Sintassi assembly** (un esempio)	**Semantica**	**Note**
add	`add $1 $2 $3`	$1 = $2 + $3	Check di overflow
subtract	`sub $1 $2 $3`	$1 = $2 – $3	Check di overflow
add unsigned	`addu $1 $2 $3`	$1 = $2 + $3	
subtract unsigned	`subu $1 $2 $3`	$1 = $2 – $3	
shift left logical variable	`sllv $1 $2 $3`	$1 = $2 << $3	
shift right logical variable	`srlv $1 $2 $3`	$1 = $2 >> $3	
shift right artimetical variable	`srav $1 $2 $3`	$1 = $2 >> $3	
or	`or $1 $2 $3`	$1 = $2 ∨ $3	Bit a bit
and	`and $1 $2 $3`	$1 = $2 ∧ $3	
xor	`xor $1 $2 $3`	$1 = $2 ⊕ $3	
not	`not $1 $2`	$1 = ~ $2	

26

Sintassi dell'assembly: schema di tutte le istruzioni di tipo R

codice «mnemonico» dell'op.

destination

operands

```
ADD $31, $6, $11
```

le virgole sono opzionali

27

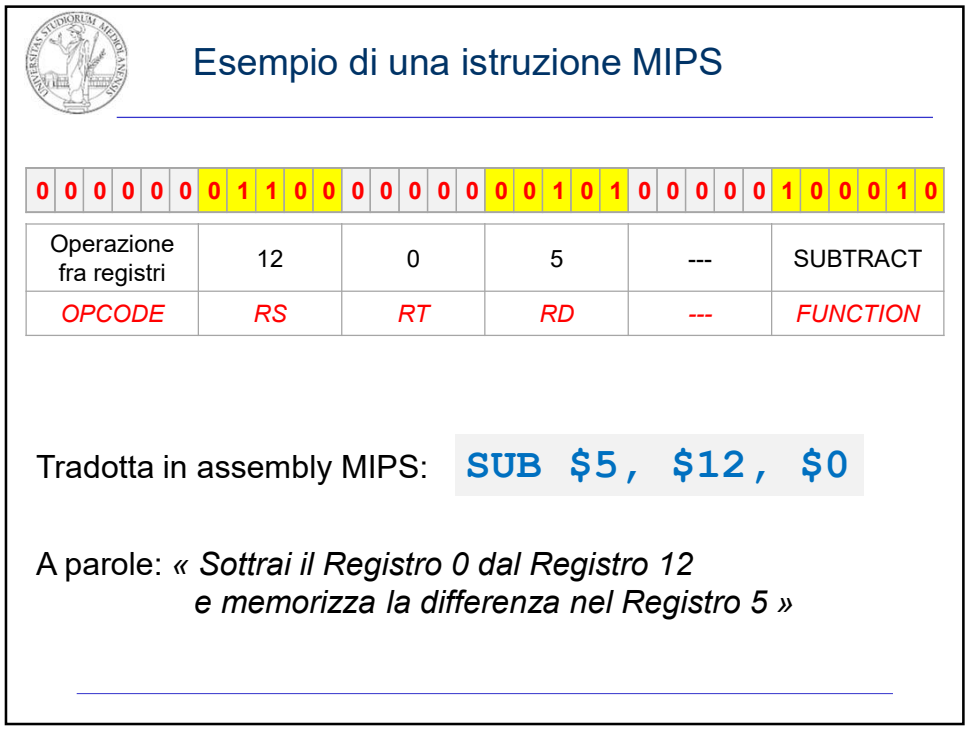
Esempio di una istruzione MIPS **in esadecimale**

0	0	C	B	F	8	2	0
0 0 0 0	0 0 0 0	1 1 0 0	1 0 1 1	1 1 1 1	1 0 0 0	0 0 1 0	0 0 0 0

Operazione fra registri	6	11	31	---	SUM
OPCODE	*RS*	*RT*	*RD*	*---*	*FUNCTION*

```
ADD $31, $6, $11
```

29



Esempio di una istruzione MIPS

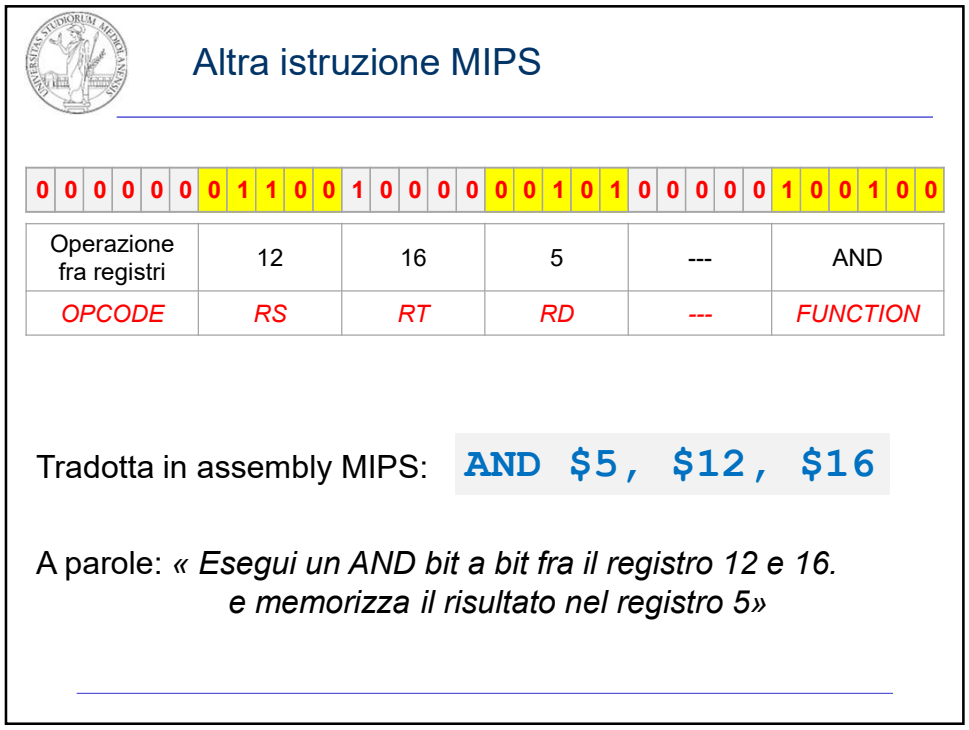
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0

Operazione fra registri	12	0	5	---	SUBTRACT
OPCODE	*RS*	*RT*	*RD*	*---*	*FUNCTION*

Tradotta in assembly MIPS: `SUB $5, $12, $0`

A parole: *« Sottrai il Registro 0 dal Registro 12 e memorizza la differenza nel Registro 5 »*

31



Altra istruzione MIPS

0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0

Operazione fra registri	12	16	5	---	AND
OPCODE	*RS*	*RT*	*RD*	*---*	*FUNCTION*

Tradotta in assembly MIPS: `AND $5, $12, $16`

A parole: *« Esegui un AND bit a bit fra il registro 12 e 16. e memorizza il risultato nel registro 5»*

32

Un'altra istruzione MIPS simile

Operazione fra Registri	12	16	5	---	Shift a sinistra
OPCODE	*RS*	*RT*	*RD*	*---*	*FUNCTION*

Tradotta in assembly MIPS: `SLLV $5, $12, $16`

A parole: *« Fai uno shift a sinistra del Registro 12 di tante cifre quanti ne indica il Registro 16. e memorizza il risultato nel registro 5»*

33

Un'altra istruzione MIPS

Operazione fra Registri	12	16	5	---	Shift a destra
OPCODE	*RS*	*RT*	*RD*	*---*	*FUNCTION*

Tradotta in assembly MIPS: `SRLV $5, $12, $16`

A parole: *« Fai uno shift a destra del Registro 12 di tante cifre quanti ne indica il Registro 16. e memorizza il risultato nel registro 5»*

34

Istruzioni aritmetiche del'assembly MIPS note sull'uso (introduzione)

Visione a basso livello (HW):

- Abbiamo visto come una **CPU**
  (un circuito HW costituito dal **Datapath**, che include **ALU** e **banco di registri**, pilotato dalla **Unità di Controllo**)
  interpreta ed esegue le **istruzioni** di«**tipo R**», cioè operazioni fra registri

Visione ad alto livello (SW):

- Qui di seguito vedremo alcune note sul loro uso dal punto di vista dello **scrittore di programmi** in linguaggio **assembly MIPS**
- Che, di solito, non è un programmatore *umano*,
  ma un **compilatore** automatico che sta traducendo in assembly un linguaggio più ad alto livello come **C** o **GoLang**
- (ma che anche potesti essere tu, mentre svolgi un esercizio all'esame)

In comune e in mezzo ai due livelli c'è l'**instruction set** (linguaggio macchina)

- 35 -

35

Note sull'uso: operazioni **add** e **sub**, **addu** e **subu**

- L'unica differenza fra le versioni **signed** (default) e **unsigned** è se l'eventuale overflow debba generare un'**eccezione** o no
  - **Eccezione**: situazione particolare (gestita in primis dall'hardware) che causa l'interruzione del normale funzionamento del programma
- Nella sintassi dell'assembly mips:
  - **Unsigned** = ignora overflow
  - **Signed** = solleva eccezione se avviene overflow (in CP2)

No, non è intuitivo come mai MIPS adotti proprio questi termini.
Potevano chiamarsi «add with/without overflow check»

Come sappiamo (vedi lezione su CP2),
fra due numeri in complemento a 2,
l'overflow è determinato dal mismatch dei segni,
mentre fra due numeri senza segno, l'overflow è deteriminato dall'ultimo riporto. Il MIPS ingora questo tipo di overflow.

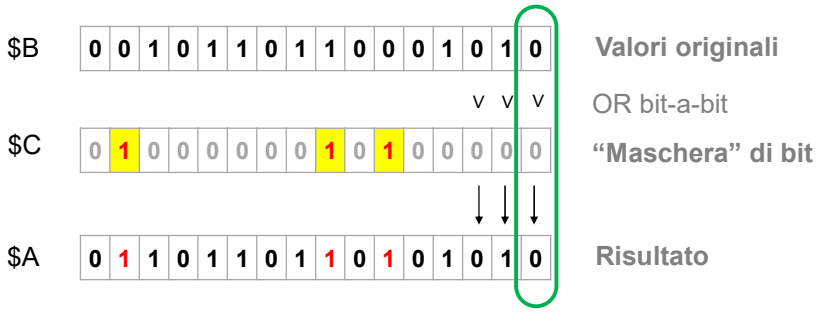
- 36 -

36

Note sull'uso: operazioni logiche bit a bit

- $A = $B ∨ $C
- OR: *metti ad uno questi bit!*

X + 0 = X
X + 1 = 1

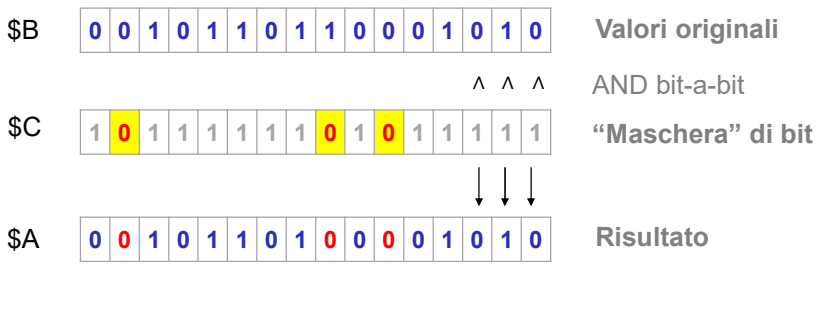


37

Note sull'uso: operazioni logiche bit a bit

- $A = $B ∧ $C
- AND: *metti a zero questi bit!*

X · 0 = 0
X · 1 = X

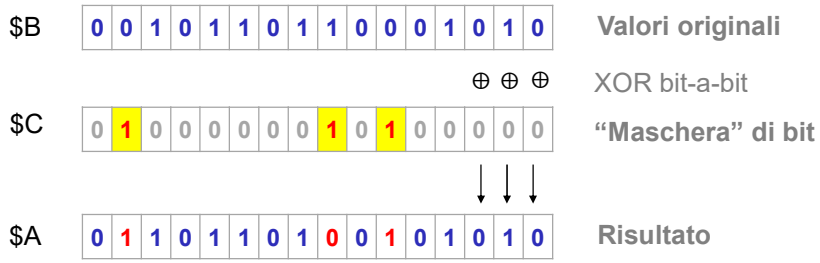


38

Note sull'uso: operazioni logiche bit a bit

- $A = $B $\oplus$ $C
- XOR: *flippa questi bit!*

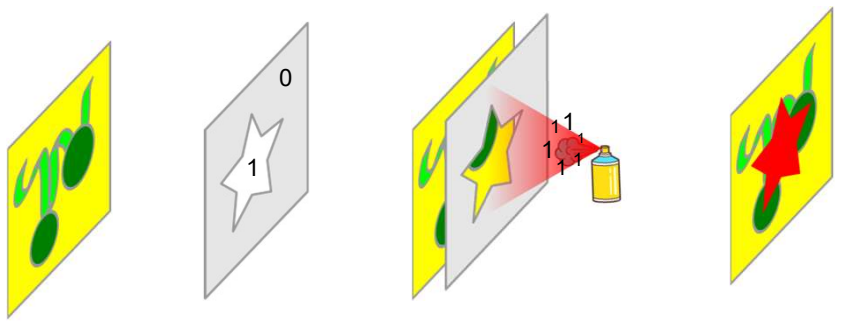
$X \oplus 0 = X$
$X \oplus 1 = \bar{X}$



39

Perché si chiama «maschera» di bit: è un po' come uno stencil in pittura

- Con l'OR:



40

Note sull'uso: operazioni logiche bit a bit

`NOT $13 $20`

- Cioè, $13 = NOT $20
- In generale: $A = NOT $B
- Il registro $A diventerà l'opposto, su ogni bit, del registro $B

Considerazioni sul numero di operandi

- Ad "alto" livello (linguaggio assembly) ("alto" solo per modo di dire):
  Questa istruzione (pur di tipo R) prevede un solo operando, non due come le altre!
- A basso livello (linguaggio macchina):
  No problem: *nei cinque bit che riportano il secondo operando l'assembler può scrivere qualsiasi cosa, per esempio 5 volte 0 (la codifica di $0)*
- A livello ancora più basso (architettura): No problem
  - Quando la UC (Unità di Controllo) chiede alla ALU il calcolo della funzione "bit-wise not", la ALU produrrà il suo output in funzione di uno solo dei due operandi (A), ignorando l'altro (B)
  - *Non conta quindi quale registro chiediamo al Banco di Registri di produrre nella sua seconda uscita out_B: va bene chiedere ad esempio il registro $0*

- 41 -

41

Note sull'uso: shift

- Sono dette operazioni di shift "**variabile**" perché il 2° operando è un registro (quindi un valore che varia, quindi una *variabile*)
  - Shift variabile:
    "shifta $5 di **$3** posti" (se **$3** vale 7, shifta $5 di 7)
  - Shift costante (non variabile):
    "shifta $5 di **7** posti"
- Shift a **sinistra** di *k* posti: moltiplicazione per 2^k
- Lo shift a **destra** si distingue fra
  - **Logico**:
    da sinistra compaiono 0 – semplice spostamento di bit
  - **Aritmetico**:
    da sinistra compare il MSB (0 o 1).
    Si ottiene sempre la divisone (intera, cioè per difetto) per 2^k, anche per i numeri negativi in CP2

  Si tratta di due funzioni distinte che possiamo richiedere alla ALU, ciascuna con il suo codice distinto.

- 42 -

42

E le istruzioni…

- …per azzerare un registro?
- …per copiare un registro in un altro?
- …per invertire il segno di un registro?
- …per non far nulla?
  (che è detta «NO-OP» in alcuni IS, per **No-Op**eration.
  Cioè rimanere inattivi per un ciclo di clock.
  Ad esempio, eseguire una serie di NO-OP può essere utile per mettere in pausa l'elaboratore, nonostante il ciclo fetch-and-execute non si fermi mai.
  Questo modo di attendere è chiamato, a ragion veduta, «attesa attiva».)

Filosofia **RISC** in azione (Reduced Instruction Set Computer):
non c'è bisogno di includere queste istruzioni nella CPU
(nel datapath e nella control Unit, e/o nella ALU).
Il loro effetto è ottenibile usando le istruzioni che la CPU sa già fare.

(In che modo? C'è un esercizio su questo, in fondo)

Pseudo-istruzioni

- **Pseudo istruzione**: istruzione introdotta dal linguaggio assembly che viene tradotta (dall'assembler) in una o un paio (o più) di istruzioni «reali».
  - appartiene al livello di astrazione dell'Assembly, ma non esiste al livello del Linguaggio macchina
- Ad esempio, la pseudo-istruzione `MOVE`:

  `MOVE $3 $4` ("copia registro $4 nel registro $3")

  viene tradotta come l'istruzione reale

  `ADD $3 $4 $0`

  Funziona perché il registro $0, come abbiamo visto, contiene automaticamente sempre e solo il valore 0…0 cioè il numero 0.
  Anche se scrivessimo nel registro $0, il suo valore non cambierebbe.

Prodotti e divisioni in MIPS: in apparenza (le pseudo-istruzoni)

"Inventate" dall'assembly

Istruzione	Sintassi assembly (esempio)	Semantica	Note
multiply	`mul $1 $2 $3`	$1 = $2 x $3	
divide	`div $1 $2 $3`	$1 = $2 / $3	Divisione INTERA!
reminder	`rem $1 $2 $3`	$1 = $2 % $3	Resto

46

Prodotti e divisioni in MIPS: la realtà (le istruzioni "vere")

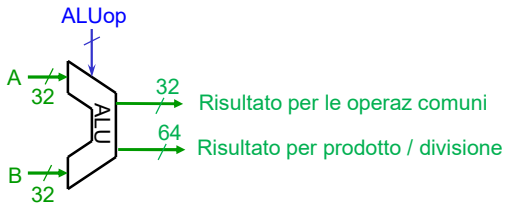
Supportate dall'ISA

Comando	Sintassi (esempio)	Semantica	Commenti
multiply	`mult $2,$3`	Hi | Lo = $2×$3	prodotto con segno: (risulato in 64 bit)
divide	`div $2,$3`	Lo = $2 / $3, Hi = $2 % $3	Lo = quoziente Hi = resto
move from Hi	`mfhi $1`	$1 = Hi	Copia Hi in un registro
move from Lo	`mflo $1`	$1 = Lo	Copia Lo in un registro

47

Prodotti e divisioni in MIPS: la realtà

- Moltiplicazioni e divisioni fanno eccezione al nostro schema
- Motivo: il risultato di un prodotto o una divisione di due parole di 32 bit non è una parola di 32 bit, ma di 64 bit (o DUE parole a 32 bit)
  - **Prodotto**: il risultato è una parola di 64 bit
  - **Divisione**: i risultati sono due parole: quoziente (32 bit)
    resto (32 bit)
- La nostra ALU in realtà ha due fasci di canali in uscita:



- 48 -

48

Prodotti e divisioni in MIPS: la realtà

Abbiamo già visto un registro «speciale»: il **PC**

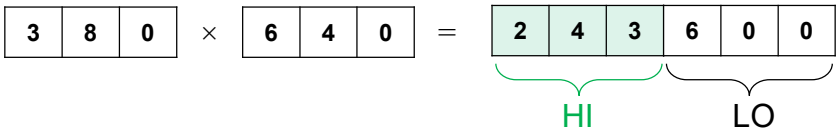
- I risultati di moltiplicazioni e divisioni sono memorizzati in due altri appositi registri «speciali» (non parte del banco dei registri), detti
  HI: i 32 bit più significativi, o il resto della divisione
  LO: i 32 bit meno significativi, o il quoziente della divisione
- Due apposite istruzioni («move from hi» e «move from lo») copiano questi registri in un registro «utente» a scelta
  - Registro «utente» = uno dei 32 del banco dei registri, da $0 a $31
- Non vediamo come modificare il Data Path per implementare tutte queste istruzioni (mult, div, mfhi, etc). Invece, per semplicità…
  - facciamo finta che MUL, DIV, REM, siano istruzioni «reali» come le altre (e come le altre, a due operandi e un risultato)
  - In realtà, sono PSEUDO istruzioni
  - Vedi esercizi in fondo, su come l'assembler le traduce
- Nella pratica: se il prodotto non è enorme (è minore di miliardi di unità) allora HI risulta essere tutti 0 e va bene usare solo LO per il risultato
  - In pratica, HI è l'«overflow del prodotto», ma è un overflow… di 32 bit

- 49 -

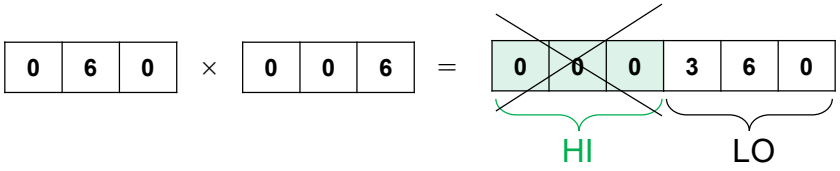
49

Concetto equivalente usando la base 10 (il concetto vale in tutte le basi)

- Il prodotto di due numeri interi di 3 cifre è un numero di (al massimo) 6 cifre



- Ma quando i due fattori che moltiplico sono abbastanza piccoli, (in particolare, quando loro prodotto è $<10^3 = 1000$), posso ignorare la parte HI e prendere solo la parte LO



- Nota: tutto questo riguarda solo i numeri INTERI. I numeri in virgola mobile non hanno affatto questo problema.

- 50 -

50

Esercizio: programmazione assembly

- Scrivi un programma assembly che calcoli in $20 il computo

  $20 = (($15 + $19) / ($15 - $19))^2

  - La divisione si intende intera.
  - No, il MIPS non prevede un'istruzione per calcolare il *quadrato* di un numero intero. E' RISC, no?

- Mettiamo che, in un dato momento, i registri da $0 a $8 contengano i valori da 0 a 8, e $9 contenga il valore 63.
  - Scrivi un breve programma che, dato un registro $18, scriva…
  - in $19 la sua divisione intera per 64,
  - in $20 il resto di questa divisione

  …ma **senza** poter usare le istruzioni div o mul / mult!

- 51 -

51

Esercizi: mettiti nei panni dell'**assembler**

In questi due esercizi, ti metterai nei panni dell'assembler.

- Come tradurresti le seguenti pseudo istruzioni in istruzioni normali?
  - `MUL $2 $6 $7`
  - `DIV $2 $3 $4`
  - `REM $3 $6 $7`
  - `ZERO $4` «azzera il registro $4»
  - `NEG $3 $5` «il registro $3 assume il valore del registro $5 cambiato di segno»
  - `ONE $3` «metti il registro $3 a tutti uno»
  - `NOOP` «no operation» (molte soluz possibili; bonus per eleganza: riesci farlo con una codifica bella «naturale»?)

  (quelle in *corsivo* non sono pseudo-istruzioni comunemente accettate)

- Traduci in linguaggio macchina, esprimendole come 4 bytes in esadecimale, le istruzioni `ADD $4 $4 $4`, e `SLRV $0 $0 $0`

52

Esercizi su shift e uso di maschere di bit

- **Immagina per semplicità che i registri siano di 8 bit soltanto:**
  - Ipotizza che il registro $6 memorizzi il valore -50 (in CP2) e che il registro $7 memorizzi il valore 2
  - Quale valore binario si ottiene nei registri $1,$2,$3 dopo le istruzioni
    ```
    SLLV $1 $6 $7
    SRLV $2 $6 $7
    SRAV $3 $6 $7
    ```

vedi lezione su rappresentazione testi e caratteri

- Verifica che il 5° bit da sinistra di un codice ASCII che rappresenta una lettera alfabetica vale 0 nelle maiuscole, e 1 nella sua versione minuscola
  - Ipotizza che, in un dato momento, i registri $1,$2,$3,.. fino a $8 contengano i valori 1, 2, 3 … fino a 8
  - Dato un codice ASCII (mauiscolo o minuscolo) nel registro $16, scrivi un programma assembly per:
    (a) renderlo minuscolo (a prescindere dal case originale)
    (b) renderlo maiuscolo (idem)
    (c) invertire il case (passare da minuscolo a maiuscolo, o viceversa)

53