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

Lezione 12:

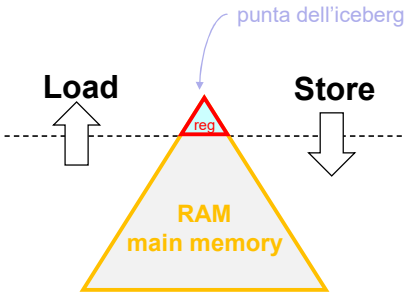
CPU e linguaggio MIPS

Parte D: load and store (RAM)

Marco Tarini

87

Ripasso: scambi di memoria fra registri e Main Memory



- I dati che i nostri programmi devono elaborare sono molti numerosi di più di quelli che registri possono mantenere
- Il grosso dei dati è memorizzato in Main Memory (RAM)
  - puro storage di dati, senza capacità di elaborazione diretta
- Scambio di dati avviene attraverso operazioni di **LOAD** e **STORE**
  - Copiano *un word alla volta*
  - Il linguaggio macchina MIPS supporta questi comandi (stiamo per vedere come)
  - Il linguaggio assembly MIPS prevede le istruzioni corrispondenti (vediamo con quale sintassi)

88

Istruzione MIPS per load da memoria

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

Load Word	14	5	0
OPCODE	*RS*	*RT*	*IMMEDIATE (16 bits)*

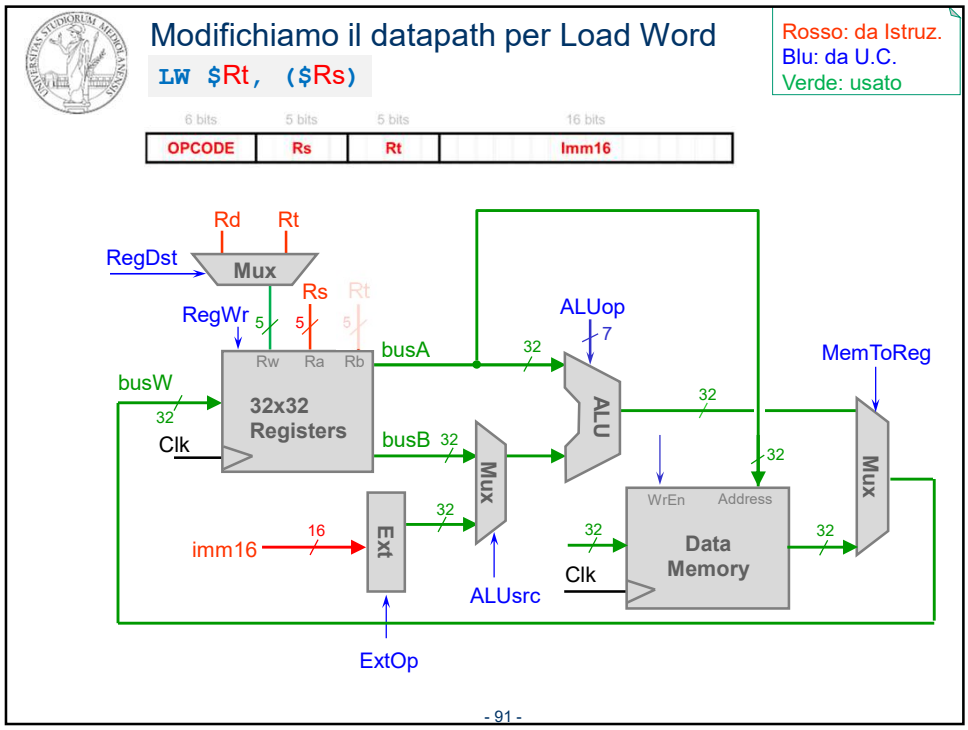
Tradotta in assembly MIPS: `LW $5, ($14)`

A parole: *« Accedi la memoria RAM in lettura all'indirizzo contenuto nel Registro 14, e memorizza la parola letta nel Registro 5 »*

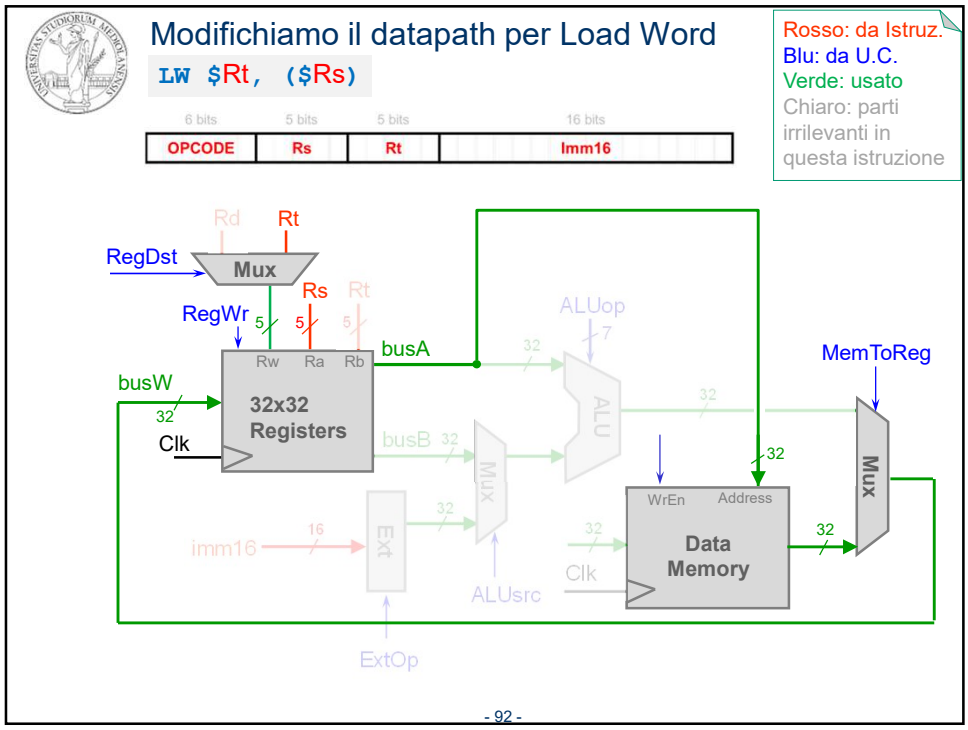
Istruzione MIPS per load da memoria: note

- In **linguaggio macchina**: l'indirizzo di memoria al quale accedere (di 32 bit) non può essere contenuto nell'istruzione (lunga 32 bit in tutto)
  - Invece, la Load Word specifica un registro che *contiene* questo indirizzo
- In **assembly**: il 1° argomento è il registro che sovrascriviamo (come al solito)
- In **assembly**: la sintassi richiede di incapsulare 2° argomento, cioè il registro che contiene l'indirizzo di RAM, fra parentesi tonde
  - Questo è il modo di questo linguaggio assembly per aiutarci a ricordare che ha luogo una «**indirezione**»
  - Cioè che, nell'esempio, scriviamo in $5 non *il word contenuto del registro $14* ma *il word in RAM che ha come indirizzo il word contenuto nel registro $14*
- Cioè: *direttamente* il valore di $14 ma *indirettamente* il valore di RAM a quell'indirizzo.
  - Valore che possiamo denotare con **RAM[$14]** , usando la notazione che hai visto nel corso di programmazione per gli "array" (o "vettori")

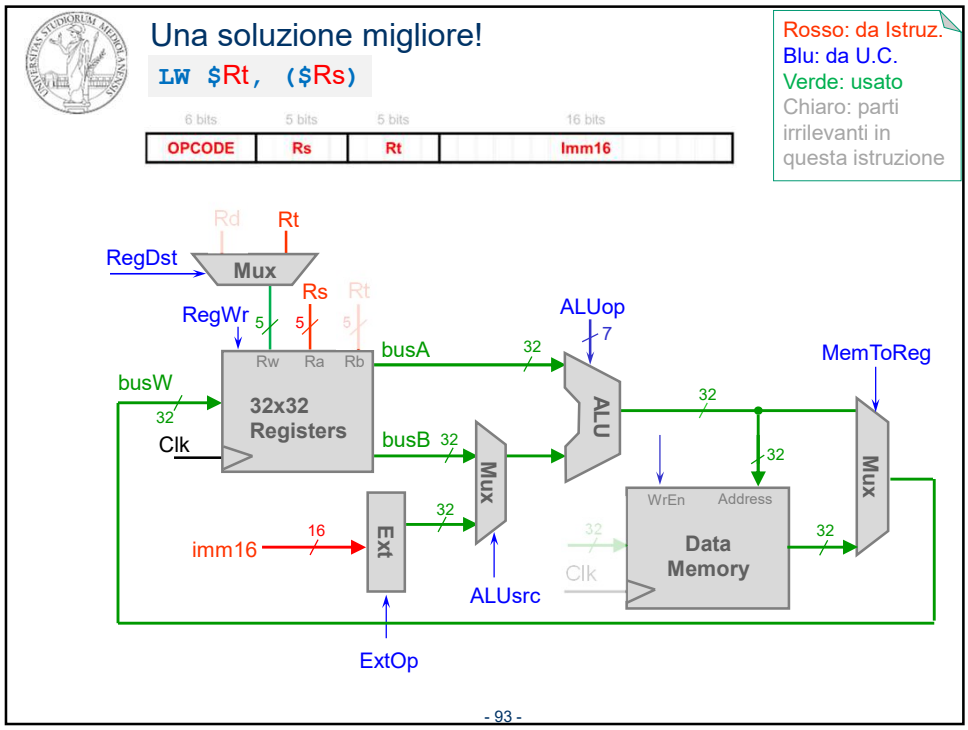
Nota il termine


Modifichiamo il datapath per Load Word
LW $Rt, ($Rs)
Rosso: da Istruz.
Blu: da U.C.
Verde: usato
6 bits
5 bits
5 bits
16 bits
OPCODE
Rs
Rt
Imm16
Rd
Rt
RegDst
Mux
Rs
Rt
RegWr
5
5
5
busA
32
ALUop
7
MemToReg
busW
32
Rw
Ra
Rb
32x32
Registers
Clk
busB
32
Mux
ALU
32
32
Mux
WrEn
Address
imm16
16
Ext
32
32
Data
Memory
32
Clk
ALUsrc
ExtOp
- 91 -

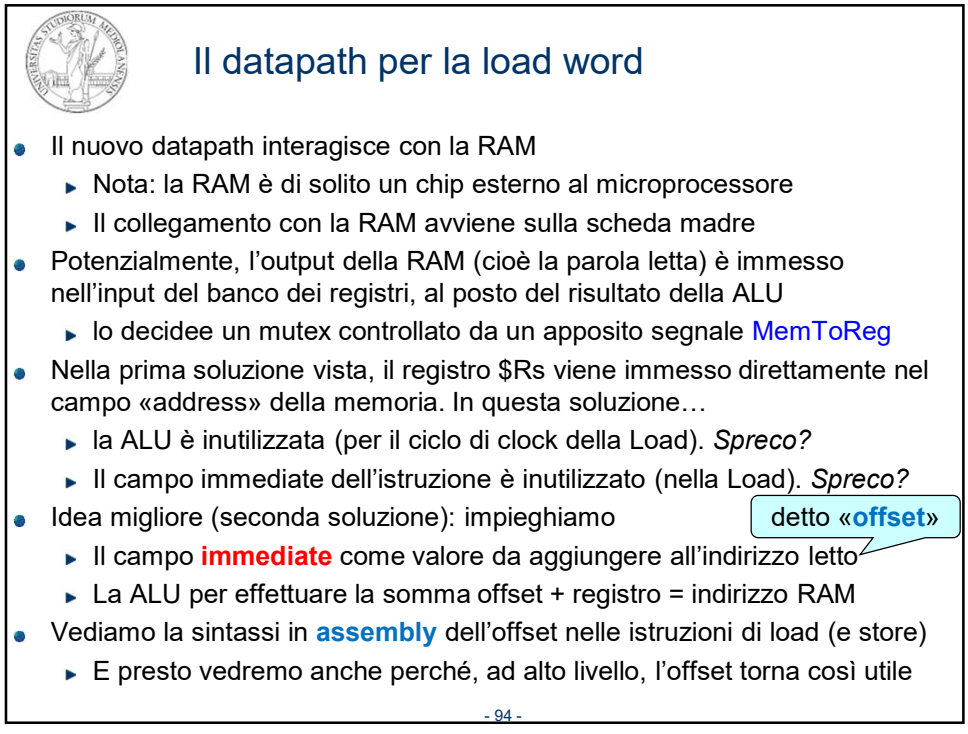
91


Modifichiamo il datapath per Load Word
LW $Rt, ($Rs)
Rosso: da Istruz.
Blu: da U.C.
Verde: usato
Chiaro: parti irrilevanti in questa istruzione
6 bits
5 bits
5 bits
16 bits
OPCODE
Rs
Rt
Imm16
Rd
Rt
RegDst
Mux
Rs
Rt
RegWr
5
5
5
busA
32
ALUop
7
MemToReg
busW
32
Rw
Ra
Rb
32x32
Registers
Clk
busB
32
Mux
ALU
32
32
Mux
WrEn
Address
imm16
16
Ext
32
32
Data
Memory
32
Clk
ALUsrc
ExtOp
- 92 -

92


Una soluzione migliore!
LW $Rt, ($Rs)
6 bits
5 bits
5 bits
16 bits
OPCODE
Rs
Rt
Imm16
Rosso: da Istruz.
Blu: da U.C.
Verde: usato
Chiaro: parti irrilevanti in questa istruzione
Rd
Rt
RegDst
Mux
Rs
Rt
RegWr
5
5
5
Rw
Ra
Rb
busA
32
ALUop
7
MemToReg
busW
32
32x32 Registers
Clk
busB
32
ALU
32
32
Mux
Mux
WrEn
Address
32
imm16
16
Ext
32
32
Data Memory
32
Clk
ALUsrc
ExtOp
- 93 -

93


Il datapath per la load word
Il nuovo datapath interagisce con la RAM
Nota: la RAM è di solito un chip esterno al microprocessore
Il collegamento con la RAM avviene sulla scheda madre
Potenzialmente, l'output della RAM (cioè la parola letta) è immesso nell'input del banco dei registri, al posto del risultato della ALU
Io decidee un mutex controllato da un apposito segnale MemToReg
Nella prima soluzione vista, il registro $Rs viene immesso direttamente nel campo «address» della memoria. In questa soluzione…
la ALU è inutilizzata (per il ciclo di clock della Load). Spreco?
Il campo immediate dell'istruzione è inutilizzato (nella Load). Spreco?
Idea migliore (seconda soluzione): impieghiamo
detto «offset»
Il campo immediate come valore da aggiungere all'indirizzo letto
La ALU per effettuare la somma offset + registro = indirizzo RAM
Vediamo la sintassi in assembly dell'offset nelle istruzioni di load (e store)
E presto vedremo anche perché, ad alto livello, l'offset torna così utile
- 94 -

94

Prossimo esempio di istruzione MIPS

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

Load Word	14	5	240
OPCODE	*RS*	*RT*	*IMMEDIATE (16 bits)*

Tradotta in assembly MIPS: `LW $5, 240($14)`

A parole: « *Somma 240 al Registro 14,*
accedi la memoria RAM all'indirizzo che hai ottenuto;
memorizza la parola letta nel Registro 5 »

- 95 -

95

Prossimo esempio di istruzione MIPS

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

Load Word	14	5	-1 *è in CP2!*
OPCODE	*RS*	*RT*	*IMMEDIATE (16 bits)*

Tradotta in assembly MIPS: `LW $5, -1($14)`

A parole: « *Togli 1 al Registro 14*
accedi la memoria RAM all'indirizzo che hai ottenuto;
memorizza la parola letta nel Registro 5 »

- 96 -

96

Prossimo esempio di istruzione MIPS: la store (copia da un registro alla RAM)

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

Store Word	14	5	0
OPCODE	*RS*	*RT*	*IMMEDIATE (16 bits)*

Tradotta in assembly MIPS: `SW $5, ($14)`

A parole: « *Memorizza il Registro 5 in memoria RAM all'indirizzo contenuto nel Registro 14* »

97

Nota: anche le store hanno l'offset

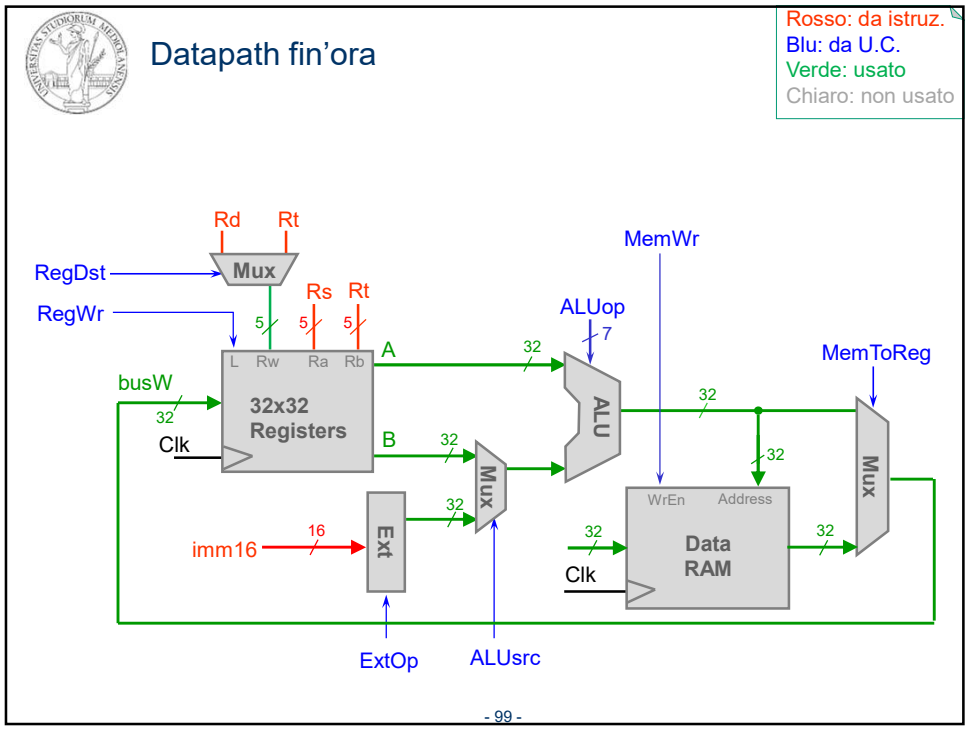
1	0	1	0	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Store Word	14	5	4 *in CP2*
OPCODE	*RS*	*RT*	*IMMEDIATE (16 bits)*

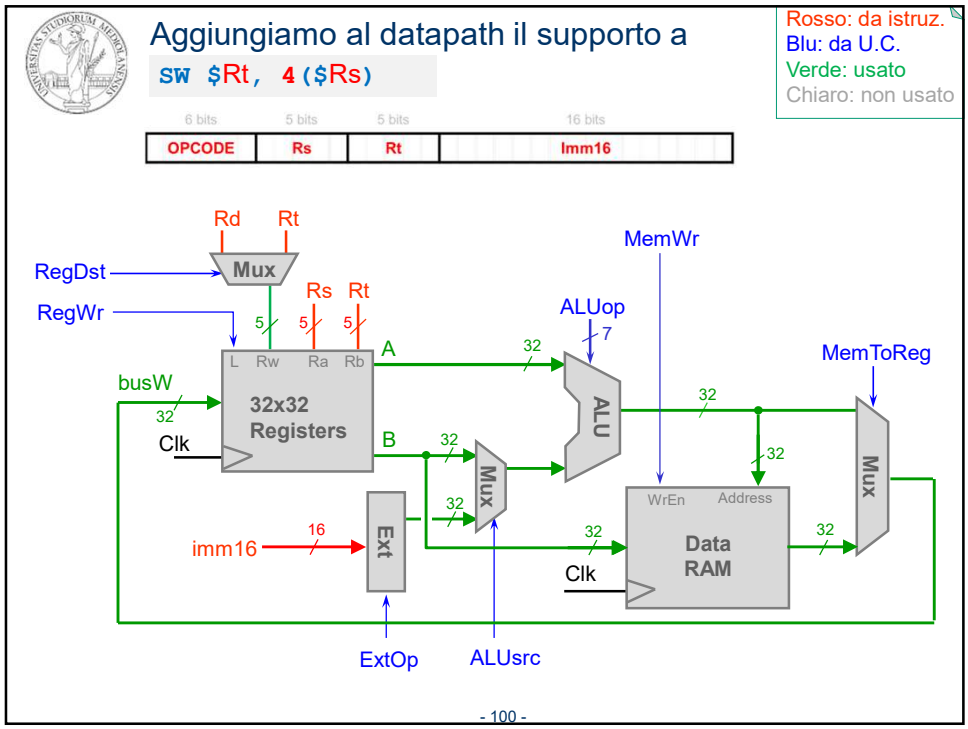
Tradotta in assembly MIPS: `SW $5, 4($14)`

A parole: « *Memorizza il Registro 5 in memoria RAM all'indirizzo contenuto nel Registro 14… aumentato di 4 bytes* »

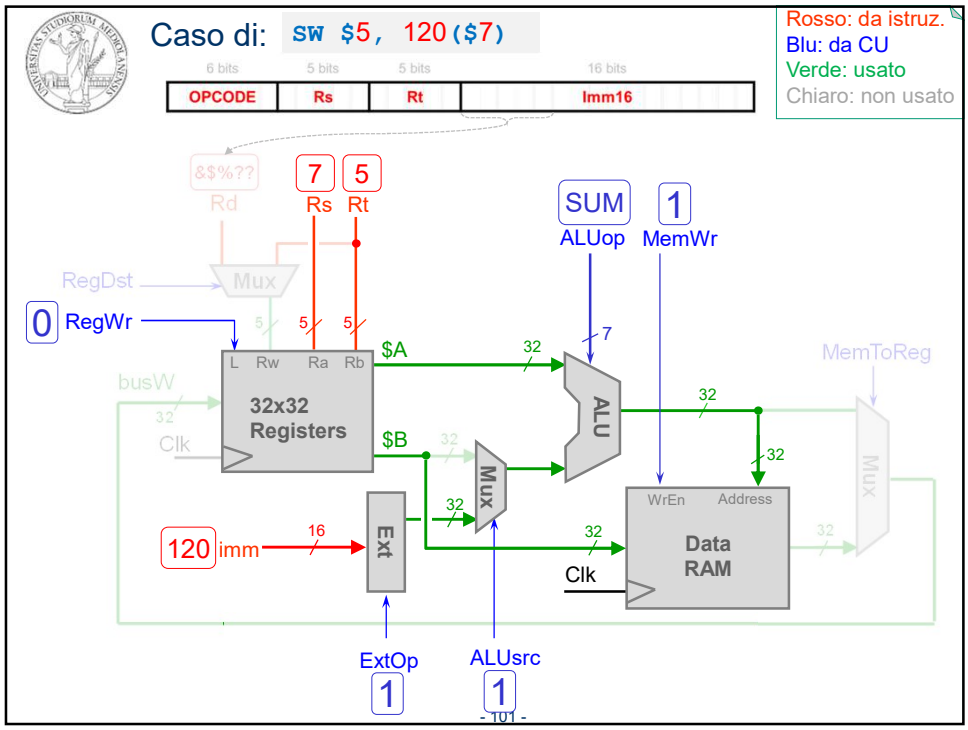
98


Datapath fin'ora
Rosso: da istruz.
Blu: da U.C.
Verde: usato
Chiaro: non usato
Rd
Rt
Mux
RegDst
RegWr
Rs
Rt
5
5
5
L Rw Ra Rb
32x32
Registers
busW
32
Clk
A
B
32
32
ALUop
7
ALU
MemWr
32
MemToReg
Mux
32
32
Mux
32
16
imm16
Ext
WrEn
Address
Data
RAM
32
32
Clk
ExtOp
ALUsrc
- 99 -

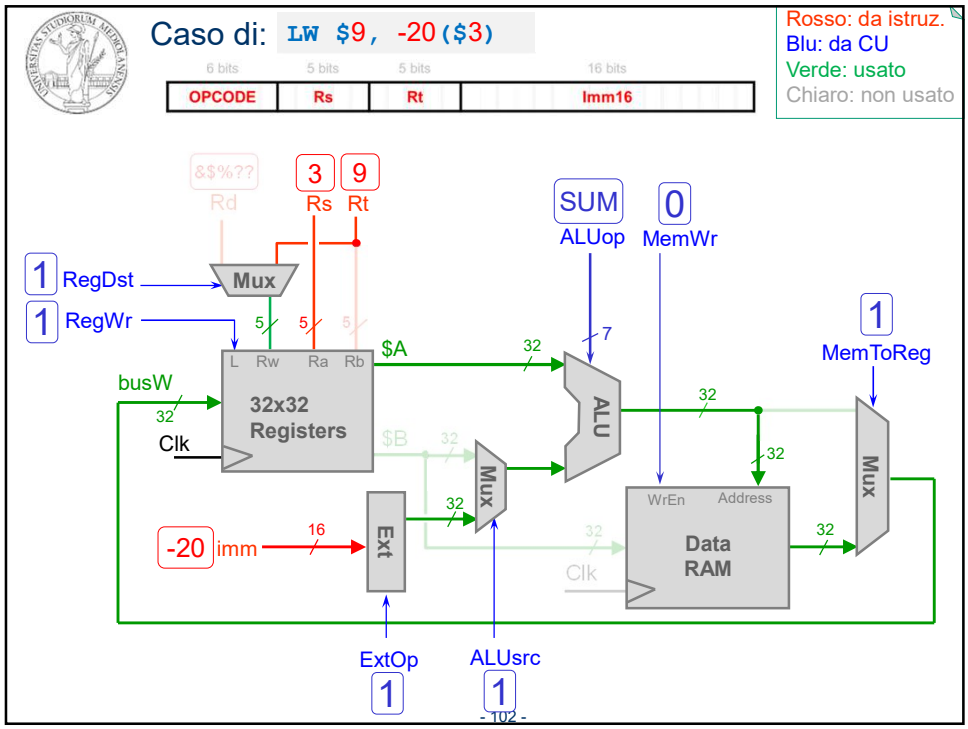
99


Aggiungiamo al datapath il supporto a
sw $Rt, 4($Rs)
6 bits
5 bits
5 bits
16 bits
OPCODE
Rs
Rt
Imm16
Rosso: da istruz.
Blu: da U.C.
Verde: usato
Chiaro: non usato
Rd
Rt
Mux
RegDst
RegWr
Rs
Rt
5
5
5
L Rw Ra Rb
32x32
Registers
busW
32
Clk
A
B
32
32
ALUop
7
ALU
MemWr
32
MemToReg
Mux
32
32
Mux
32
16
imm16
Ext
WrEn
Address
Data
RAM
32
32
Clk
ExtOp
ALUsrc
- 100 -

100


Caso di: SW $5, 120($7)
Rosso: da istruz.
Blu: da CU
Verde: usato
Chiaro: non usato
6 bits
5 bits
5 bits
16 bits
OPCODE
Rs
Rt
Imm16
&$%??
Rd
7
Rs
5
Rt
SUM
ALUop
1
MemWr
RegDst
Mux
0
RegWr
5
5
5
L Rw Ra Rb
32x32
Registers
busW
32
Clk
$A
32
7
ALU
32
MemToReg
$B
32
32
Mux
32
WrEn
Address
Data
RAM
32
120
imm
16
Ext
32
Clk
Mux
ExtOp
1
ALUsrc
1
- 101 -

101


Caso di: LW $9, -20($3)
Rosso: da istruz.
Blu: da CU
Verde: usato
Chiaro: non usato
6 bits
5 bits
5 bits
16 bits
OPCODE
Rs
Rt
Imm16
&$%??
Rd
3
Rs
9
Rt
SUM
ALUop
0
MemWr
1
RegDst
Mux
1
RegWr
5
5
5
L Rw Ra Rb
32x32
Registers
busW
32
Clk
$A
32
7
ALU
32
1
MemToReg
$B
32
32
Mux
32
WrEn
Address
Data
RAM
32
-20
imm
16
Ext
32
Clk
Mux
ExtOp
1
ALUsrc
1
- 102 -

102

Accesso alla memoria RAM

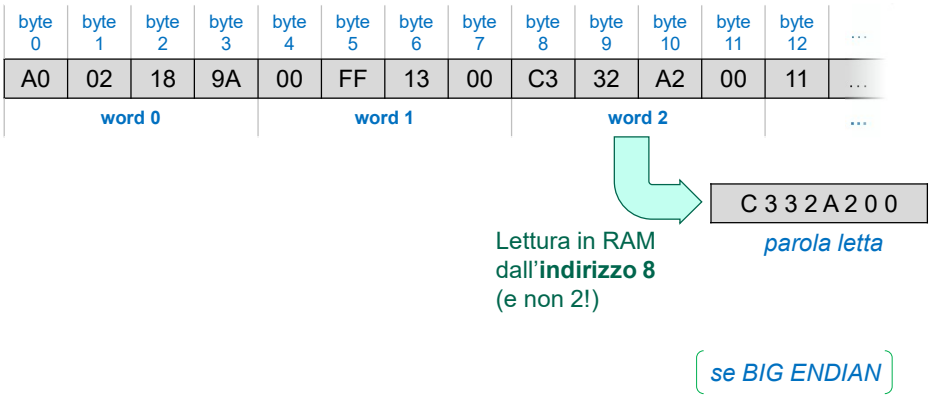
- Le modalità di accesso alla RAM sono una delle scelte dell'ISA
  - ISA diversi compiono scelte diverse
  - Diversi compromessi fra flessibilità, efficienza, facilità di realizzazione…
- Vediamo nei prossimi lucidi i dettagli dell'accesso in memoria del **MIPS-32**
- Eccone un sommario:
  - L'indirizzo è di 32 bit (spazio di indirizzamento: da 0 a 2^{32}-1)
  - Si accede sempre utilizzando l'indirizzo di un *byte*
  - Un accesso legge o scrive un **word** di **4 byte**
  - Gli accessi ai word devono essere «**allineati**» (vedi poi)
  - L'architettura è «**big-endian**»
  - (Vedremo nella prossima lezione: è anche possibile accedere ai singoli byte o a coppie di byte, invece che a parole di quattro byte)
  - Tutto quanto sopra si sia applica sia alle letture (load) sia alle scritture (store) in RAM

- 103 -

103

Accesso alla memoria RAM in MIPS

RAM:



- 105 -

105

Accesso alla memoria RAM in MIPS

RAM:

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7	byte 8	byte 9	byte 10	byte 11	byte 12	…
A0	02	18	9A	00	FF	13	00	C3	32	A2	00	11	…
word 0				word 1				word 2					…

0 0 A 2 3 2 C 3

parola letta

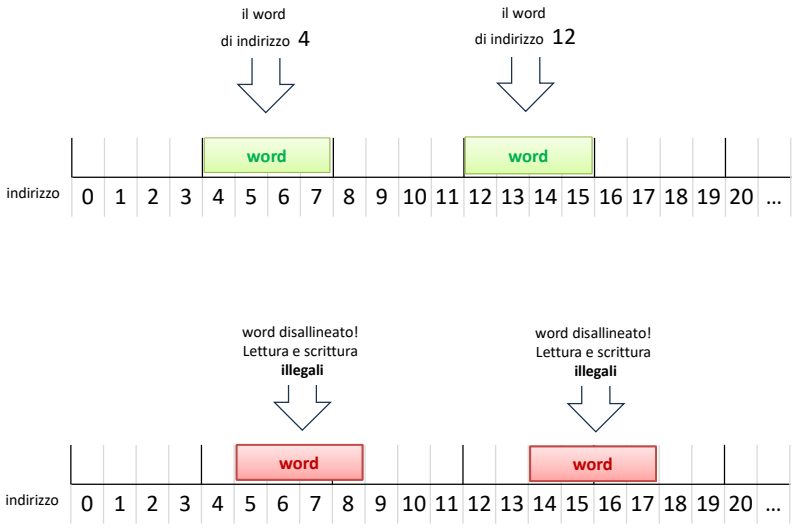
Lettura in RAM dall'**indirizzo 8** (e non 2!)

se LITTLE ENDIAN

Note sulla memoria RAM in MIPS

- In MIPS-32, **non** è consentito effettuare accessi in RAM a word "non allineati", cioè ad indirizzi (in bytes) non divisibili per 4
  - Il vincolo deriva da come è fatta la RAM al suo interno
- Tentare di accede ad un word non allineato solleva un'**eccezione** a tempo di esecuzione
  - Terminologia:
    eccezione = evenienza imprevista che può essere trattata (ad es) interrompendo il programma e riportando l'errore
  - Terminologia: a tempo di esecuzione = "**dinamica**"
- E' responsabilità del programmatore MIPS (umano o complilatore) effettuare solo comandi Load Word o Store Word con indirizzi che siano divisibili per 4
  - Domanda: come è fatto, **in binario**, un numero intero positivo divisibile per 4?
  - Risposta: termina con «00»

Allineamento dati: i word



108

Accesso allla memoria RAM in MIPS

- Se voglio accedere al word in memoria di indice N, devo quindi usare l'indirizzo (sempre in byte!): (N·4)
- Basta cioè aggiungere 2 zeri alla fine della codifica binaria di N

Un indirizzo di RAM
espresso come indice di un word (30 bits)

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

Lo stesso indirizzo
ma espresso come indice di bytes (32 bits)
(è quello che devo usare negli accessi in RAM)

- 109 -

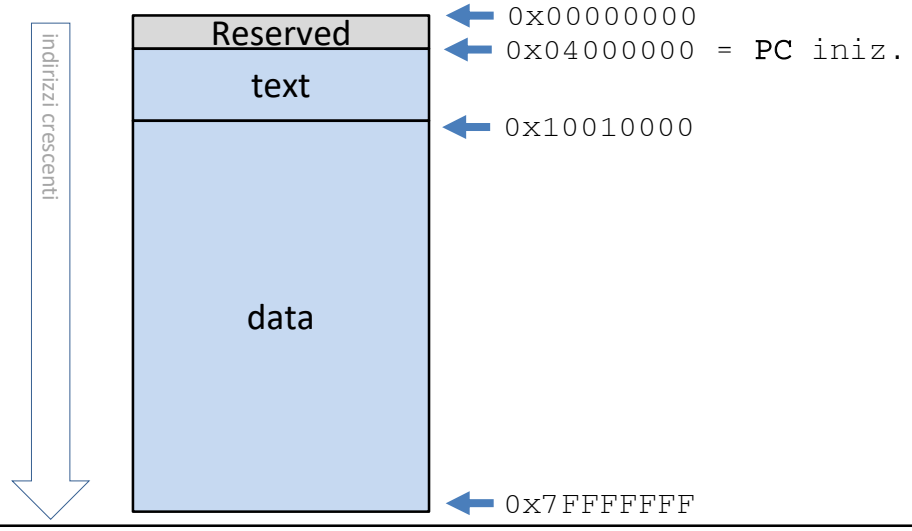
109

Programmi in assembly che salvano o leggono dati in RAM

- La memoria che accediamo in lettura e scrittura con load e store è quella che contiene i dati del nostro programma (non il codice)
- Scelta: quale locazione di RAM vogliamo usare per i dati?
- Per rispondere a questa e molte altre) durante la stesura di programmi, usiamo delle **convenzioni**
  - Seguire le convenzioni ha molta utilità: ad esempio, consente a nostri programmi scritti in assembly di interagire correttamente con altri programmi (librerie), e col sistema operativo, perché entrambi seguono le stesse convenzioni
  - Queste convenzioni appartengono al livello dell'assembly.
  - I livelli sottostanti (linguaggio macchina, architettura) non devono esserne consapevoli
- Un primo esempio di convenzione decide per noi dove, in memoria (cioè, a quali indirizzi), posizionare il programma (la sequenza di istruzioni), e dove i dati

110

Convenzione MIPS: partizione della memoria di aree



111

Convenzione MIPS: partizione della memoria di aree standard

- Programma (SW):
  - Dati
  - +
  - Istruzioni
- Entrambi risiedono in RAM («architettura di Von Neumann»)
- Convenzione MIPS: dividere la RAM in 2 segmenti, in posizioni fissate:
  - Segmento «**data**» (dati)
  - Segmento «**text**» (istruzioni)
  - C'è anche un terzo segmento «**riservato**» al Sistema Operativo (contiene i dati e istruzioni del SW «sistema operativo», e include ad esempio il codice che viene eseguiti per gestire le eccezioni, quando si verificano – non ce ne occupiamo)
- Il **Sistema Operativo**, per eseguire un programma, dopo aver caricato in memoria i suoi dati + istruzioni, posiziona il PC al valore della alla prima riga del segmento "text"
  - (di solito; o in generale, all'istruzione dove comincia il «main»)

112

Direttive Assembler

- Sono «indicazioni» nei nostri sorgenti in assembly, che «spiegano» all'Assembler su come trattare il codice assembly che le segue
- Sintassi: le direttive sono parole precedute da un punto
- (nota: non hanno corrispondente in linguaggio machcina)

- In MIPSweb tutte le direttive sono visibili sotto *help* → *directives*

113

Alcune direttive Assembler

`.data` : «caro assembler, di seguito ci sono dati (numerici, etc) da tradurre in binario e memorizzare nel segmento **data** della RAM (uno dopo l'altro)»

- In pratica: precede i dati del nostro software.
- I dati li possiamo specificare come literals, in base 10, 16 (0x…), con segno (in CP2), etc

`.text` : «caro assembler, di seguito ci sono istruzioni (o pseudo-istruzioni) da tradurre in linguaggio macchina e da memorizzare nel segmento **text** della RAM (una dopo l'altra)»

- In pratica: precede il codice del nostro software.

(ricorda: software = codice eseguibile + dati)

114

Info

- Le slides dopo questa anticipano alcuni dettagli finali sulle load e store che vedremo nella *prossima lezione*
- Sono inclusi in questo file solo per non frammentare l'esposizione delle slides in troppi file

- 115 -

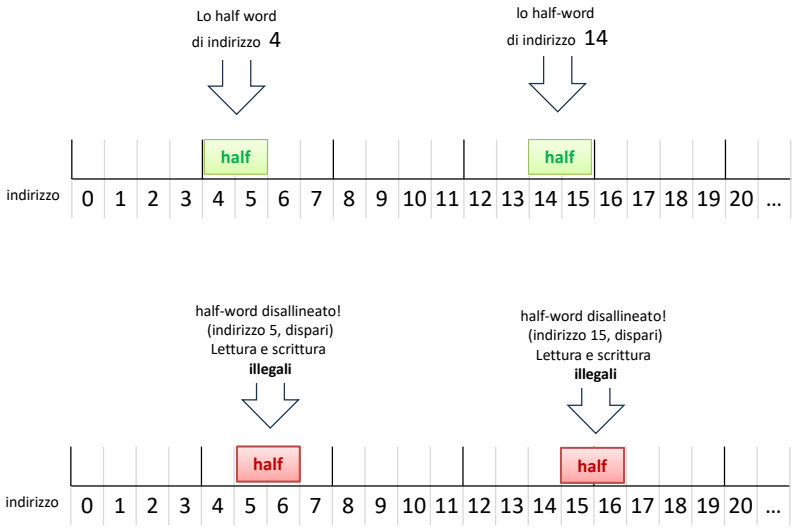
115

Accessi in memoria di singoli byte o di due byte (half-word)

- Fin'ora, abbiamo visto l'accesso in memoria di word (4 byte)
- Il MIPS-32 supporta anche l'accesso «a granularità più piccola»:
  - **Half** (cioè half-words): 2 byte (16 bit)
  - **Byte**: 8 bits.
- Si usano apposite istruzioni
  «**load half**» «**store half**» , «**load byte**» «**store byte**»
- Per semplicità, non vediamo il supporto HW di queste istruzioni
  - Necessarie nel **datapath**, e nel **banco di memoria** RAM
- Info pratiche per l'uso:
  - Le **store** sovrascrivono solo gli 1 o 2 byte richiesti della RAM, lasciando il resto della parola di RAM inalterata
  - Le **load** sovrascrivono sempre *tutto* un registro (di 4 byte)!
    Gli 1 o 2 byte letti dalla RAM sono estesi in segno.
    O, opzionalmente, con zero, se si usano le versione «**unsigned**» della «**load byte**» e «**load half**»
  - Per gli **half**, è richiesto l'allineamento (in load o store) a 2 byte

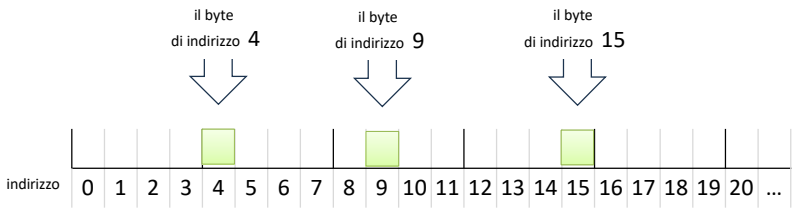
116

Allineamento dati: gli half-word



117

Allineamento dati: i byte (singoli)



qualsiasi accesso ad un byte è sempre «allineato»

118

Istruzioni di load e store in assembly MIPS: sommario

Istruzione	**Sintassi** (esempio)	**Semantica** (dell'esempio)	**Note**
store *word*	`sw $1 150($2)`	RAM[$2 + 150] ← $1	L'offset (in verde) è opzionale e vale 0 se non specificato
store *half*	`sh $1 150($2)`		
store *byte*	`sb $1 150($2)`		
load *word*	`lw $1 150($2)`	$1 ← RAM[$2 + 150]	
load *half*	`lh $1 150($2)`		
load *byte*	`lb $1 150($2)`		
load *half unsigned*	`lhu $1 150($2)`		
load *byte unsigned*	`lbu $1 150($2)`		

119