



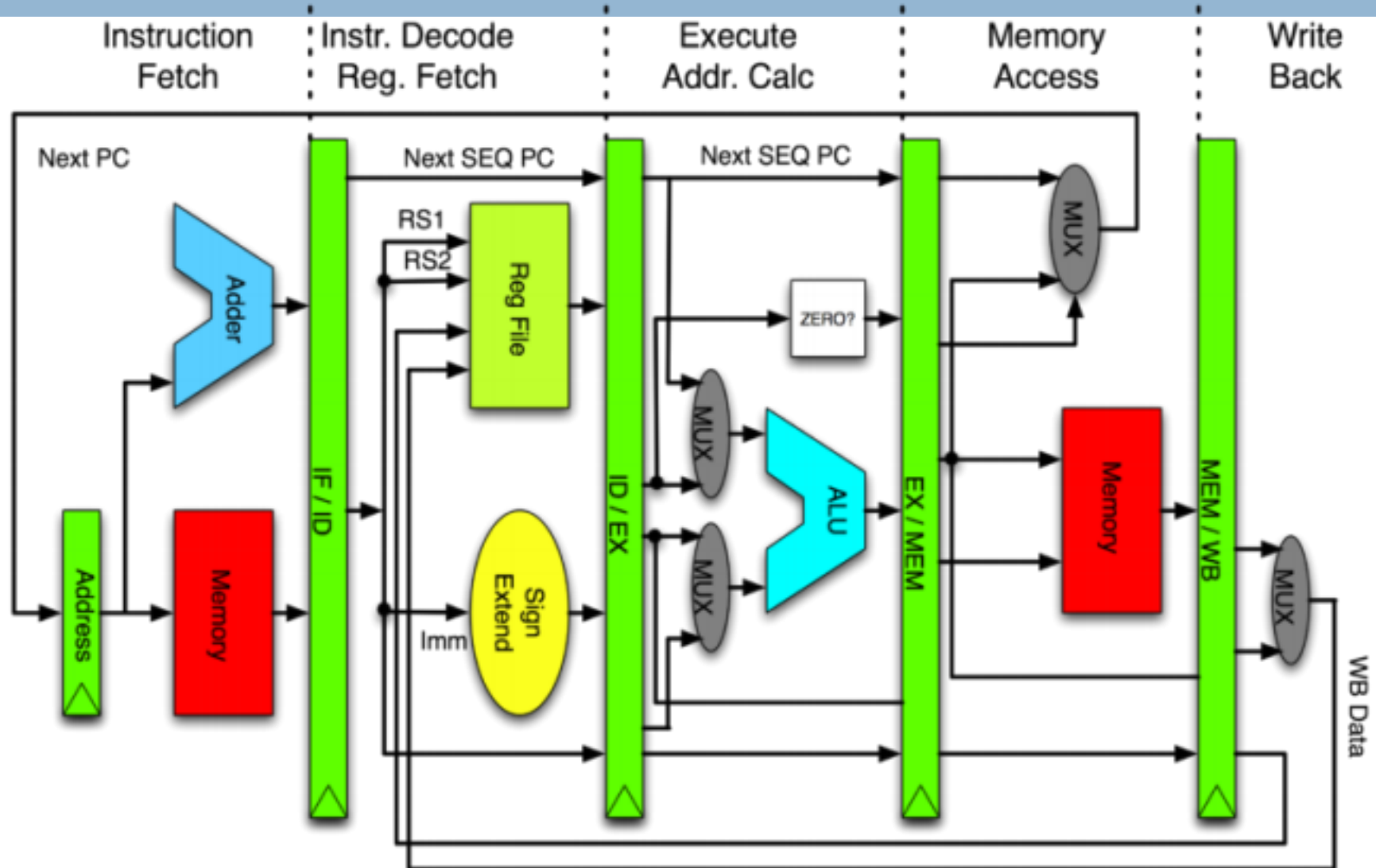
Introduzione a MIPS64 (I parte)

MIPS64: caratteristiche



- Nel 1981 *John L. Hennessy* della Stanford University avviò un gruppo di ricerca sulle architetture RISC
- **RISC (vs CISC):** Istruzioni complesse non implementate direttamente
- **Pipelining:** possibile grazie a istruzioni semplici che eseguono in un ciclo di clock.
- **Load/Store:** ogni dato va caricato dalla memoria con un'istruzione prima di essere manipolato

MIPS Pipelining



MIPS 64 Instruction Set Architecture

■ **Concetti architetturali:**

- Semplicità del load/store IS
- Semplicità nella decodifica (istruzioni a lunghezza fissa da 32 bit)

■ **Caratteristiche**

- Compatibilità con MIPS32 ISA
- 32x64-bit General Purpose Registers
- 32x64-bit Floating Point Registers
- La lunghezza di una word è di 64 bit.
- Supporta variabili a 8, 16, 32 e 64 bit
- Indirizzamento della memory a byte, Little Endian

Registri

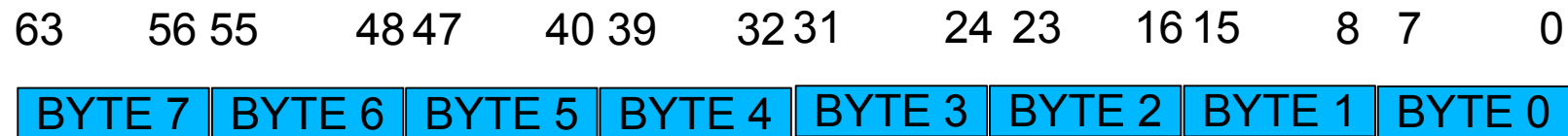
- L'ISA del MIPS64 contiene 32 (R0-R31) general-purpose registers da 64 bit
- I registri R1-R31 sono dei reali GP registers
- R0 contiene sempre il valore 0 e non può essere modificato
- R31 è utilizzato per conservare l'indirizzo di ritorno per le istruzioni JAL e JALR

Note sull'uso dei registri

- I registri sono "omogenei", possiamo utilizzarli per allocare i valori scegliendoli come ci piace (eccetto $R0$, che vale sempre 0).
- In realtà, i compilatori adottano delle consuetudini per scegliere i registri: es. $R29$ come stack-pointer, $R31$ per l'indirizzo di ritorno dalle procedure, etc
- In quanto "compilatori-umani", cercare di fare attenzione e cercare di adottare uno stile: Es:
 - $R2-R7$: indici (es. Gli i, j, k, n del C/Java)
 - $R8-R15$: valori temporanei (temp, pippo, etc..)
 - $R16-R27$: deposito valori salvati (somma=)

Registri

- I bit dei registri sono numerati come 63-0, da destra a sinistra.
- L'ordinamento dei byte è fatto in modo simile



- Un registro può essere caricato con
 - × un byte (8-bit)
 - × un halfword (16-bit)
 - × una word a (32-bit)
 - × Una double word (64-bit)

Registri speciali

- **PC**, Program Counter, contiene l'indirizzo dell'istruzione da leggere dalla memoria (32 bit)
- HI e LO, due registri interni alla CPU a cui si accede mediante le istruzioni MFLO and MFHI

Struttura programmi assembly

EduMIPS64

- **Direttive:** Danno indicazioni sull'interpretazione del testo. Non corrispondono ad alcuna operazione dal punto di vista della semantica del programma assembly
- **Direttiva `.data`** Il codice generato dopo questa direttiva viene allocato nel segmento dei dati.
- **Direttiva `.code`** Il codice generato dopo questa direttiva viene allocato nel segmento testo.
- **Etichette:** associate ad un indirizzo di memoria
- **Commenti:** iniziano con ;

Struttura programmi assembly

EduMIPS64



```
; This is a comment
```

```
.data
```

```
Label: .word 15 ;This is a comment
```

```
.code
```

```
daddi r1,r0,0
```

In questo esempio:

Direttive, etichette, commenti, istruzioni

Tool di Simulazione: EduMIPS64

- Sviluppato da alcuni studenti di Laboratorio di Calcolatori (anno 2006) presso la Facoltà di Ingegneria di Catania
- Multi-piattaforma (Java based)

<http://www.edumips.org>

Consiglio: scaricare al più presto e se possibile eseguire direttamente in aula gli esempi proposti

Sezione .data

- Nella sezione .data possono essere memorizzati dati

Type	Directive	Byte necessari
Byte	.byte	1
Half word	.word16	2
Word	.word32	4
Double Word	.word or .word64	8

Es

.byte 2

.word16 524

.word 32586

Allineamento automatico

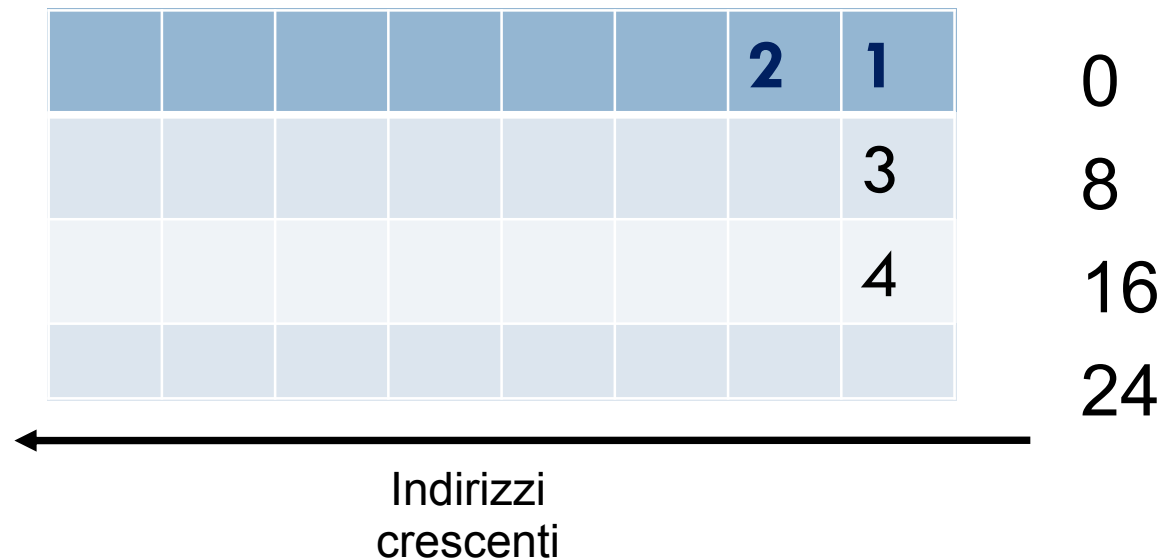
- Attenzione all'allineamento a 64 bit:

.data

.byte 1, 2

.byte 3

.byte 4



- Esercizio:** provare a digitare sezioni .data e a prevedere cosa compare nella finestra "Data"

Altre direttive

- `.space n`: lascia `n` bytes di spazio libero nella sezione `.data`
- `.asciiz "stringa"`: codifica la sequenza di caratteri in memoria e pone un byte nullo alla fine (terminatore C-like).
- NB: esiste anche una versione `.ascii`, senza `'z'` finale, ad indicare l'assenza del terminatore.

Sezione .code

- Contiene le istruzioni MIPS64 da eseguire. Tre tipologie di parametri:
 - **Registri:** 32 registri di 64 bit, indicati da R0 . . R31.
NB: R0 è cortocircuitato a 0.
Esempio: `dadd R1, R2, R0`
 - **Valori immediati:** espressi numericamente in decimale o esadecimale.
Esempio: `daddi R1, R1, 4`
- NB:** L'eventuale etichetta che precede un'istruzione è utilizzabile come un immediato che ha come valore l'indirizzo di memoria a cui si trova l'istruzione.

Sezione .code

- **Indirizzi:** sono utilizzati dalle istruzioni load/store e sono specificati nel formato:

`offset (Rbase)`

dove `Rbase` è un registro che contiene il valore *base* e `offset` è un valore immediato che contiene lo scostamento da sommare.

Ad esempio, se `R1` vale 1024

`lw R2, 8 (R1)`

caricherà la word che inizia all'indirizzo di memoria $8+1024$.

Classificazione delle Istruzioni

Le istruzioni possono essere classificate in base a due criteri:

- **Categoria:**

- aritmetico/logica (ALU),
- load/store,
- controllo di flusso,
- di sistema

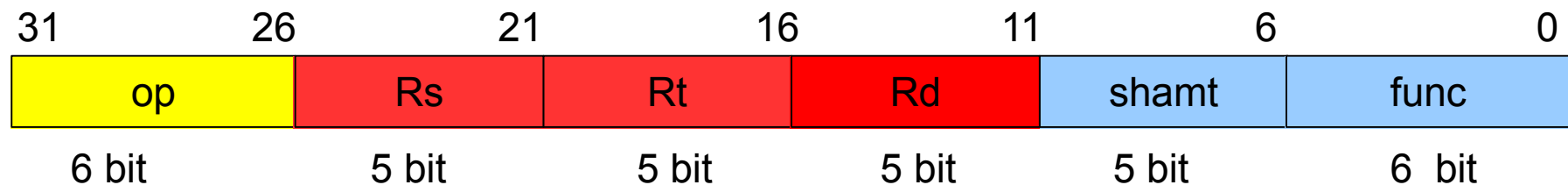
- **Formato:**

- **R-type:** tutti gli operandi sono registri
- **I-type:** uno degli argomenti è un immediato (16 bit)
- **J-type:** salti che non utilizzano registri come destinazioni

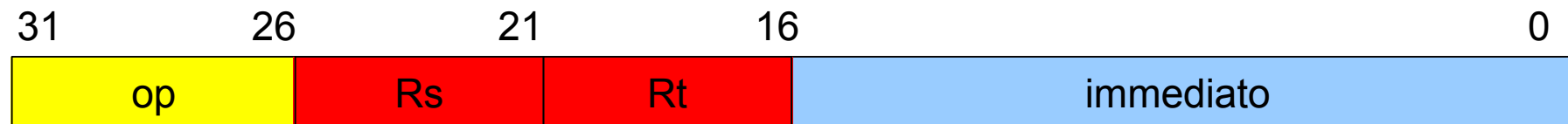
Formato delle Istruzioni

Tutte le istruzioni sono codificate in 32 bit

- R-type:



- I-type:



- J-type:



Istruzioni load/store: load double

LD Rt, offset(Rbase)

- Significato: carica 64 bit sul registro *Rt*, leggendo a partire dal byte all'indirizzo *offset+Rbase*, ossia:

$Rt = \text{mem}[\text{offset} + Rbase]$ dove:

- L'argomento *base* specifica un registro
- L'argomento *offset* è un immediato (16 bit)
- **NB:** pur essendo MIPS64 un'architettura a 64bit, si usa il termine "double" per indicare 8byte, come tradizione delle usuali macchine a 32 bit

Istruzioni Load/Store: load byte

LB Rt, offset(Rbase)

LBU Rt, offset(Rbase) (versione unsigned)

- Significato: carica il byte all'indirizzo offset+Rbase sul registro Rt
- **NB:** la **LB** estende il bit del segno. Es: se il byte vale -2 sarà codificato come 1111 1110, caricandolo su 64 bit DEVE diventare:

1111 1111 1111 1111 1111 1111 1110

NON

0000 0000 0000 0000 0000 1111 1110

Istruzioni load/store: LH

LH *Rt*, **offset**(**base**)

LHU *Rt*, **offset**(**base**) (versione unsigned)

- Significato: *carica mezza word (2 byte) a partire dall'indirizzo offset+base sul registro Rt*
- **NB:** analogamente a quanto visto con LB, la LH estende il bit del segno sui rimanenti 48 bit più alti a sinistra.

Istruzioni load/store: load word

LW *Rt*, *offset(base)*

LWU *Rt*, *offset(base)* (versione unsigned)

- Significato: carica il 32 bit (4 byte) a partire dall'indirizzo *offset+base* sul registro *Rt*
- **NB:** analogamente a quanto visto con **LB**, la **LH** estende il bit del segno sui rimanenti 32 bit più alti a sinistra.
- **NB:** l'architettura è a 64 bit, ma per tradizione rispetto a MIPS32 in questo caso caricare una word significa caricare 32 bit

Istruzioni store

`SD Rt, offset(base)`

`SW Rt, offset(base)`

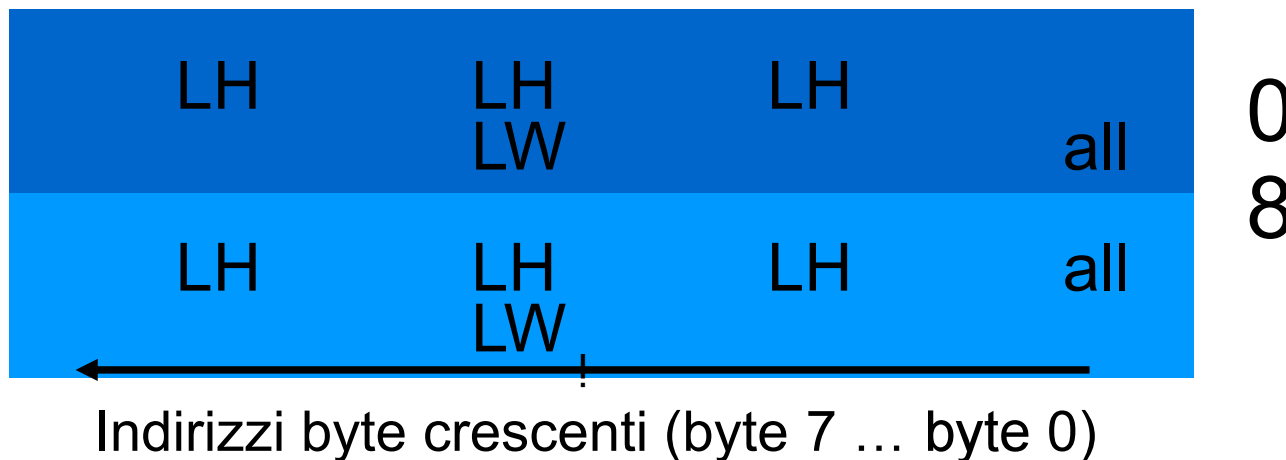
`SH Rt, offset(base)`

`SB Rt, offset(base)`

- Significato: *Memorizzano il contenuto di Rt, partendo da offset+base*
- NB: non ci sono le versioni unsigned, prendono solo i byte che servono e li scrivono
- Es: `SB R1, 0(R2)` prende solo un byte da R1 e lo memorizza ignorando il resto

Load/Store: allineamento

- Ogni accesso alla memoria deve partire ad un *indirizzo allineato rispetto alla dimensione del dato*. **Es.** LD (carica 8 byte) accede gli indirizzi 0, 8, 16, 24 etc.. LW (carica 4 byte) indirizzi 0,4,8,12 etc.. LH in 0,2,4,6,8 e LB ovunque!



Istruzioni ALU



Suddivisibili in quattro sotto-categorie:

- **Aritmetiche**
- **Comparazioni**
- **Logiche**
- **Shift**

ALU: somme tra registri

ADD Rd, Rs, Rt (somma a 32 bit)

DADD Rd, Rs, Rt (somma a 64 bit)

- Significato: $Rd = Rs + Rt$

Correlate:

- **ADDU, DADDU**: interpreta i valori nei registri come unsigned (in alcune implementazioni ignora overflow)

ALU: somme con immediato

ADDI Rd, Rs, Immediato (somma a 32 bit)

DADDI Rd, Rs, Immediato (somma a 64 bit)

- Significato: $Rd = Rs + Immediato$

Correlate:

- **ADDUI, DADDUI**: interpreta i valori come unsigned

□ Domanda:

Come si inizializza un registro Ri ?

□ **DADDI RI, R0, immediato**

ALU: sottrazioni

SUB Rd, Rs, Rt, DSUB Rd, Rs, Rt

- Significato: $Rd = Rs - Rt$
- Correlate:
 - **SUBU, DSUBU**: interpreta i valori dei registri come unsigned

Domanda:

*Perchè non c'è la **DSUBI** ??*

Perchè si usa direttamente la **DADDI** con immediato negativo

ALU: moltiplicazioni

DMULT Rs, Rt

Il risultato del prodotto viene scritto nella coppia di registri speciali Hi e LO

Hi contiene i 64 bit più significativi del prodotto

LO contiene i 64 bit meno significativi del prodotto

- Correlate:
 - **DMULTU**: la versione unsigned
- **MFLO Rd**: copia il valore di LO nel registro Rd, in quanto le normali istruzioni non possono accedere a LO
- **MFHI Rd**, analoga a MFLO, per spostare il valore del registro speciale HI in Rd

ALU: divisioni

DDIV R_s , R_t

- Significato: esegue R_s/R_t e pone il quoziente in LO ed il resto in HI

ALU: comparazioni

SLT Rd, Rs, Rt

- Significato: Se $R_s < R_t$ allora $R_d = 1$ altrimenti 0.
- Correlate:
 - **SLTU**: interpreta i valori nei registri come unsigned

SLTI Rd, Rs, Immediato

- Significato: Se $R_s < \text{immediato}$ allora $R_d = 1$ altrimenti 0.
- **SLTUI**: interpreta i valori come unsigned

ALU: comparazioni

Domanda:

...e le altre condizioni ? ($>$, $\leq$, etc..)

Se si vuole verificare se un registro RA è maggiore di un immediato dobbiamo eseguire le seguenti istruzioni:

DADDI $RB, R0, immediato$

SLT Rd, RB, RA

Per verificare la condizione $RA \leq RB$, basta verificare che non sia vera la condizione $RB < RA$

ALU: istruzioni logiche

AND Rd, Rs, Rt

- Significato: *AND logico bit a bit tra Rs e Rt*
- Correlate:
 - **ANDI Rd, Rs, Immediato**
 - **OR, XOR** analoghe per le altre funzioni logiche

Domanda:

- *Come usarle per controllare se un numero è pari ?*

ALU: istruzioni di Shift

DSLL Rd, Rt, shamt

- Significato: $Rd = Rt \ll shamt$

NB: shamt (shift amount) è un valore immediato. Es:

DSLL R1, R2, 2

- Correlate:
 - Versione con i registri: **DSLLV Rd, Rs, Rt**
- **NB:** Dato che lo shift è verso sinistra, la parte destra è riempita con zeri, come una moltiplicazione per potenze di due

ALU: istruzioni di Shift

DSRL Rd, Rs, shamt

DSRLV Rd, Rs, Rt

- Shift verso destra, simmetriche rispetto alle DSLL.
- La parte sinistra viene riempita con zeri. Dal punto di vista logico è ok, ma dal punto di vista "aritmetico" non è più come dividere per 2

Es: $-2 = 1111 \dots 1111 1111 1111 1111 1111 1110$ diventa:
 $0111 \dots 1111 1111 1111 1111 1111 1111$

ossia $2^{31} - 1$, detto anche 2.147.483.647 !

ALU: shift aritmetico

DSRA Rd, Rs, shamt

DSRAV Rd, Rs, Rt

- **Estensione del segno:** Se il bit più a sinistra di Rs è zero, riempie a sinistra con altri zeri, altrimenti con 1
- Nel caso della slide precedente sul -2:

1111 1111 1111 1111 1111 1111 1110

diventa:

1111 1111 1111 1111 1111 1111 1111

ossia -1 (aritmeticamente meglio rispetto al caso precedente)

Inizializzazione di un registro con un valore maggiore di 16 bit

LUI Rt, immediate

- Carica i 16 bit dell'immediato nei 16 più alti della metà di 32 bit più bassa di Rt

Es: supponiamo Rt:

byte7 byte6 byte5 byte4 **byte3 byte2** byte1 byte0

allora dopo LUI Rt, 2049 (ossia 0000 1000 0000 0001)

0000 0000 0000 0000 1000 0000 0001 0000 0000 0000 0000

- NB: i byte 7..4 a sinistra subiscono l'estensione del segno (tutti 1 o zero a seconda del segno di Rt)

Inizializzazione di un registro con un valore maggiore di 16 bit

- Un immediato da rappresentare con più di 16 bit lo possiamo pensare nella forma $N = A * 2^{16} + B$
- L'inizializzazione può essere realizzata mediante una coppia di istruzioni

LUI RD, A

ADDI RD, RD, B

Istruzioni per il flow-control

- Salti condizionati:

BEQ Rs, Rt, offset

Salta all'indirizzo offset se $Rt = Rs$

- Correlate: ***BNE Rs, Rt, off*** (registri diversi anzichè uguali)

BEQZ Rs, offset

Salta all'indirizzo offset se $Rs = 0$

*Correlate: ***BNEZ*** (salta se $Rs \neq 0$)*

Istruzioni per il flow-control

- Salti incondizionati:
 - **JR Rs**: salta all'indirizzo contenuto in Rs (ossia copia Rs nel Program Counter)
 - **J target**: salta all'indirizzo specificato nell'immediato (es. Una etichetta)
 - **JALR Rs**: come JR, in più mette l'indirizzo dell'istruzione successiva (PC+8) dentro R31, in modo che si possa ritornare dal salto mediante una JR R31
 - **JAL target**: come JALR, ma usa l'immediato invece che il registro Rs

Istruzioni flow-control

- Che differenza c'è tra le due famiglie di salti ?
- Salti condizionati, solitamente *traduzione di strutture "if"*, salti relativamente vicini
- Salti incondizionati, *chiamata a procedure*, salti a regioni di codice anche molto distanti
- Le varie implementazioni di MIPS usano vari trucchetti per aumentare la raggiungibilità del codice (es. Salto relativo al PC)

Traduzione dell'istruzione if-else C (con salto se condizione falsa)

```
if (var==valore)
    istruzione_true;
else istruzione_false;
```

Associando alla variabile il registro r2

Può essere tradotto come:

```
                daddi    r8,r0, valore
istr_if :       bne r2, r8, istr_false        ;if (var!=valore)
                istruzione_true;
                j    successiva_if
istr_false:     istruzione_false;;
successiva_if:  ...
```

Traduzione dell'istruzione if-else C (con salto se condizione vera)

```
if (var==valore)
    istruzione_true;
else istruzione_false;
```

Associando alla variabile il registro r2

Può essere tradotto come:

```
                daddi  r8,r0, valore
istr_if :       beq r2, r8, istr_true      ;if (var==valore)
                istruzione_false;
                j  successiva_if
istr_true:     istruzione_true;
successiva_if: ....
```

Traduzione dell'istruzione if-else C (con salto se condizione vera)

```
if (var<valore)
    istruzione_true;
else istruzione_false;
```

Associando alla variabile il registro r2

Può essere tradotto come:

```
                slti  r8, r2, valore ;           r8=1 se r2< valore
istr_if :       bne  r8, r0, istr_true           ;if (var<valore)
                istruzione_false;
                j  successiva_if
istr_true:     istruzione_true;
successiva_if:  ....
```

Traduzione dell'istruzione if-else C (con salto se condizione falsa)

```
if (var>=valore)
    istruzione_true;
else istruzione_false;
```

Associando alla variabile il registro r2 e copiando il valore nel registro r3 il codice C può essere tradotto come:

```
        daddi r3, r0, valore
        slt  r8, r3, r2  ; r8=1 se r3< r2 ovvero se  var> valore
istr_if : beq r8, r0, istr_false      ;if (var>=valore)
        istruzione_true;
        j  successiva_if
istr_false: istruzione_false;
successiva_if: ....
```

Traduzione di un ciclo for C

```
for (i=0; i<5; i++)  
    istruzione_for;
```

Associando alla variabile i il registro r2

Può essere tradotto come:

```
Inizializza:    daddi r2, r0, 0           ;i=0  
for_loop:      slti r8, r2, 5           ; r8==1 se i<5  
               beq r8, r0, end_loop      ; se r8==0 i=5  
               istruzione_for  
               daddi r2, r2, 1           ;i++  
               j for_loop               ;  
end_loop:
```

Traduzione di un ciclo do-while C

```
i=0;
do {
    istruzione_while;
    i++;
}while (i!=10);
```

Associando alla variabile i il registro r2, può essere tradotto come:

```
Inizializza:  daddi r2, r0, 0           ;i=0
               daddi r8, r0, 10        ; r8=10
while_loop :  istruzione_while
               daddi r2, r2, 1          ;i++
               bne r2, r8, while_loop  ; se i!=10
end_loop:
```