RICHIAMI SUL MIPS

- UN PO' DI STORIA, CONCETTI BASE e DEFINIZIONI GENERALI
- L'ARCHITETTURA DI RIFERIMENTO: LA MACCHINA MIPS
- INSTRUCTION SET MIPS R2000
- ISTRUCTION SET SEMPLIFICATO: eMIPS
- ESEMPI DI PROGRAMMAZIONE ASSEMBLY DEL PROCESSORE eMIPS
- PASSAGGIO ASSEMBLY-LINGUAGGIO MACCHINA

UN PO' DI STORIA, CONCETTI BASE

e

DEFINIZIONI GENERALI

CPU : Unità centrale di elaborazione, (*Central Processing Unit*), anche chiamata, esclusivamente nella sua implementazione fisica, processore … Compito della CPU è quello di leggere le istruzioni e i dati dalla memoria ed eseguire le istruzioni; il risultato della esecuzione di una istruzione dipende dal dato su cui opera e dallo stato interno della CPU stessa, che tiene traccia delle passate operazioni. (Wikipedia).

Un processore è una macchina sequenziale che esegue un flusso di istruzioni residenti in memoria al fine di manipolare dei dati, a loro volta residenti in memoria. (Hennessy – Patterson)

L'Instruction Set Architecture (ISA) specifica l'insieme delle caratteristiche di un processore visibili al programmatore e al compilatore, rappresenta dunque l'interfaccia fra l'hardware e il software.

Suddividendo i calcolatori in tre classi: desktop PC, server e embedded computer, l'ISA è considerabile in larga misura uguale in tutti e tre i casi. Invece nel caso di soluzioni ISA fortemente specializzate, dove si ha a che fare per esempio con elaborazione numerica di segnali o immagini (tipicamente DSP), in cui si deve porre molta enfasi sull'elaborazione in tempo reale, è largamente diffusa la tendenza ad inserire delle istruzioni dedicate a particolari sequenze di operazioni nell'ISA dei microprocessori di tipo generale.

Le funzionalità del processore vengono definite nell' ISA in termini di:

- dati gestibili,
- registri destinati a contenere e manipolare i dati
- istruzioni che prelevano i dati dalla memoria
- istruzioni di manipolazione dei dati
- istruzioni di "salto"
- modalità di funzionamento speciali
- istruzioni speciali

CISC: supporto di istruzioni macchina equivalenti a operazioni anche di notevole complessità e di modalità complesse di indirizzamento.

Motivazioni:

- programma oggetto relativamente breve
  - minor occupazione di memoria,
  - minor numero di letture dalla memoria istruzioni;
- un'unità hardware dedicata esegue una singola operazione complessa in tempo molto minore di quello richiesto dall'esecuzione di una *sequenza di istruzioni semplici*
  - per queste operazioni si ottengono prestazioni più elevate.

I CISC erano largamente in uso intorno agli anni '70 fino ai primi anni '80 e venivano usati sia per i mainframe IBM che per i minicomputer (es. VAX) e per i microprocessori delle prime generazioni. Caso-limite: "macchine a esecuzione diretta" (per linguaggi ad alto livello).

Svantaggi:

- Complessità elevata delle istruzioni di macchina
  - unità di controllo necessariamente microprogrammata (più lenta della soluzione FSM);
- Grande complessità interna dell'unità di controllo (in particolare per la decodifica)
  - periodo di clock più lungo -> il ciclo di clock determina la sincronizzazione di *tutte* le operazioni della CPU, quindi questa risulterà rallentata!

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

Intel 80x86 (1978) – 10 semplici istruzioni costituiscono il 96% delle istruzioni eseguite su una media di 5 diversi benchmark (SPECint92).

Ottimizzazioni spinte di istruzioni "rare" non convengono! E' necessario rendere veloci le istruzioni che si usano più spesso e funzionanti tutte le altre.

The first major step on the road to a truly portable x86-based device is the "Menlow" platform that Intel touted at the most recent IDF. Menlow, which pairs Intel's 45nm mobile "Silverthorne" CPU with a special ultramobile chipset called "Poulsbo," will feature support for 802.11n and WiMAX, so that the whole Menlow package can deliver a full wireless Internet experience by running standard (x86) Linux and Windows browsers.

http://arstechnica.com/articles/paedia/cpu/Intels-x86-ISA-grows-down-today-laptops-tomorrow-the-iPhoneBlackberry.ars

RISC

Obiettivi:

- semplificare l'unità di controllo
  - ciclo di clock più breve –> esteso miglioramento delle prestazioni
- ridurre il numero di modalità di indirizzamento ammesse per i diversi tipi di istruzioni
  - istruzioni omogenee e di lunghezza fissata -> uno o un numero limitato di formati ammessi semplificano la decodifica e standardizzano la sequenza di operazioni hardware

MODALITA' DI INDIRIZZAMENTO

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4] ← Regs[R4] + Regs[R3]	When a value is in a register.
Immediate	Add R4,#3	Regs[R4] ← Regs[R4] + 3	For constants.
Displacement	Add R4,100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1 + R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	Regs[R1] ← Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer p, then mode yields $*p$.
Autoincrement	Add R1,(R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2] ← Regs[R2] + d	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.
Autodecrement	Add R1,-(R2)	Regs[R2] ← Regs[R2] − d Regs[R1] ← Regs[R1] + Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]	Used to index arrays. May be applied to any indexed addressing mode in some computers.

MIPS

Leggere la tabella:

(<-) : sta per un assegnamento

Mem[x] : contenuto della locazione di memoria x

Regs[y] : contenuto del registro y

Mem[Reg[z]] : contenuto della locazione di memoria il cui indirizzo è dato dal valore memorizzato nel registro z.

Avere modalità di indirizzamento complesse sicuramente può ridurre il numero delle istruzioni di cui si compone un programma, e conseguentemente la sua immagine in memoria. Allo stesso tempo possono richiedere un hardware di decodifica delle istruzioni dedicato e molto più complesso.

CARATTERIZZAZIONE DI UN ISA IN BASE AGLI OPERANDI

Due caratteristiche fondamentali stabilite da un instruction set sono:

- stabilire il numero massimo di operandi ammessi per ogni istruzione (ovviamente aritmetico/logica)
  - macchine a tre operandi;
  - macchine a due operandi;
  - macchine a un operando;
  - macchine a zero operandi;
- stabilire quanti di questi operandi possono essere direttamente indirizzabili dalla memoria
  - macchine memoria-memoria
  - macchine registro-memoria
  - macchine registro-registro

Macchine a tre operandi: e.g. add Ris, op1, op2

- Vantaggio: programmazione semplice.
- Problema: il numero di bit per codificare gli indirizzi degli operandi e dei risultati è limitato se si vuole che l'istruzione sia lunga al massimo una word.
  - Utilizzare registri interni alla CPU;
  - Utilizzare soluzioni ibride (e.g. uno in memoria e due nei registri);
  - Utilizzare istruzioni di lunghezza variabile (anche di più parole).

Macchine a due operandi: e.g. add Ris, op2 (il primo indirizzo inizialmente contiene un operando, alla fine conterrà il risultato).

- Vantaggio: più bit a disposizione per gli indirizzi; in genere, uno è in memoria (op2).
- Problema: occorre salvare in memoria il primo operando, se si vuole riutilizzarlo in futuro (alla fine dell'operazione, non è più disponibile nel registro in cui si trovava).

Macchine a un operando: e.g. add op (si fornisce esplicitamente l'indirizzo di un solo operando in quanto il secondo è inizialmente contenuto in un registro "privilegiato" della CPU, detto accumulatore, che al termine conterrà il risultato).

- Vantaggio: molti bit a disposizione per l'indirizzo.
- Problema: occorre predisporre inizialmente un operando nell'accumulatore, e trasferire poi il risultato dall'accumulatore alla sua destinazione finale.

Molto usato in CPU vecchie e con parola di memoria corte.

Macchine a zero operandi: sono macchine che fanno riferimento allo stack. Inizialmente gli operandi si trovano (in ordine opportuno) in cima allo stack, da cui vengono prelevati e al termine delle operazioni vi viene depositato il risultato.

Di fatto, soluzione non adottata nelle CPU attuali – il sovraccarico di gestione dello stack rovina le prestazioni.

Macchine Memoria-Memoria: nelle istruzioni viene indicato direttamente l'indirizzo di memoria dell'operando.

- Vantaggio: non vengono sprecati registri temporanei per dati magari non riutilizzati successivamente.
- Svantaggi: le variazioni nella dimensione delle istruzioni possono essere notevoli e soprattutto gli accessi in memoria rallentano l'esecuzione delle istruzioni.

Macchine Registro-Memoria: è una soluzione ibrida in cui parte degli operandi che verranno utilizzati è residente in memoria parte invece nei registri interni della CPU.

- Vantaggi: non vengono sprecati registri temporanei per dati magari non riutilizzati successivamente e parzialmente viene evitato il collo di bottiglia dell'accesso in memoria.
- Svantaggi: gli operandi non sono equivalenti e dover codificare il numero di un registro e un indirizzo di memoria in ogni istruzione potrebbe limitare il numero di registri utilizzabili (limiti sul numero di bit a disposizione); in più a seconda di dove si trovano gli operandi il numero di colpi di clock per eseguire le istruzioni può essere molto variabile.

Macchine Registro-Registro: tutti gli operandi vengono caricati nei registri interni della CPU per essere poi utilizzati per le operazioni aritmetico-logiche. Sono dette anche architetture load-store.

- Vantaggi: è possibile avere un numero limitato di formati per le istruzioni e soprattutto di lunghezza fissata, oltre alla logica di decodifica anche il modello di generazione del codice risulterà semplificato e la variabilità nel numero di colpi di clock necessari a eseguire le istruzioni viene limitata notevolmente.
- Svantaggio: l'immagine del programma in memoria è più grande perché ci saranno più o meno tante istruzioni di load quanti sono gli operandi (a meno di non riutilizzare quelli già caricati).

Particolarmente adatte alle architetture con un elevato numero di registri a disposizione.

RIASSUMENDO…

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

E' utile definire prima le classi di istruzioni che una ISA deve essere in grado di gestire:

- istruzioni di spostamento dei dati:

  spostamento dei dati dalla memoria ai vari registri interni di un processore (al minimo se non si parla di registri GPR si avranno sicuramente PC e ACC), spostamento fra i registri e più raramente fra le locazioni di memoria;

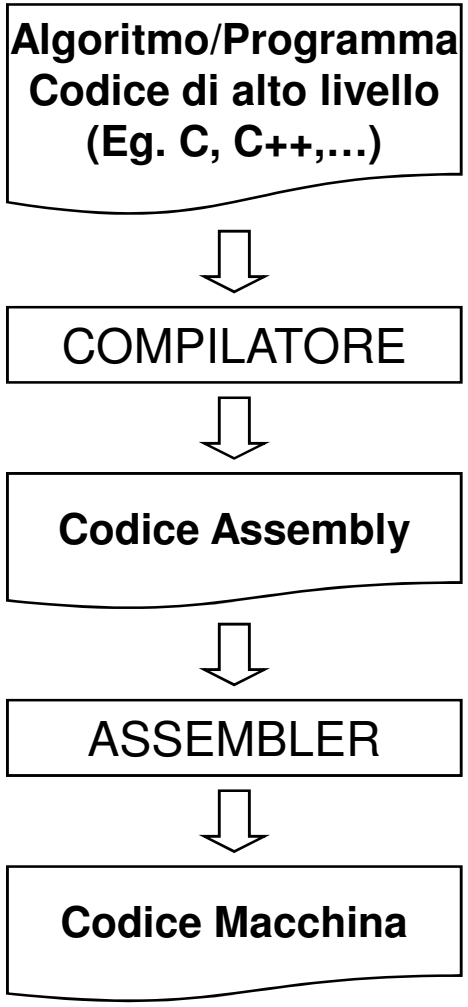
- istruzioni di manipolazione dati:

  prelevano i dati dalla loro locazione (memoria o registro) e li utilizzano come sorgenti per operazioni aritmetico/logiche. Alla fine dell'esecuzione tipicamente il risultato viene salvato nella locazione di destinazione (memoria o registro);

- istruzioni di modifica del flusso di esecuzione (salto):

  modificano il contenuto del PC per variare il flusso di esecuzione del programma (di base sequenziale).

DAL LINGUAGGIO DI ALTO LIVELLO AL LINGUAGGIO MACCHINA



L'insieme delle istruzioni macchina è il *programma oggetto* (composto da istruzioni che fanno parte dell'instruction set) e viene prodotto partendo da un *programma sorgente* scritto tipicamente in un linguaggio di alto livello.

Il codice macchina prodotto viene poi caricato nella memoria istruzioni (modello di Harvard) o nella parte di memoria centrale dedicata alle istruzioni (modello di Von Neumann).

Allo scopo di eseguire un algoritmo il processore esegue sequenzialmente il flusso di istruzioni caricate nella memoria istruzioni e afferenti al proprio instruction set.

T0: le istruzioni che codificano il programma da eseguire risiedono nella memoria istruzioni

Per poter essere eseguita ciascuna istruzione deve essere caricata nel registro interno del processore dedicato (IR-INSTRUCTION REGISTRER), questa fase viene definita FETCH.Il fetch delle istruzioni avviene in maniera del tutto sequenziale rispetto alle istruzioni salvate in memoria che vengono quindi prelevate in ordine, salvo poi la presenza di particolari istruzioni di salto (condizionato o meno) che possono alterare la sequenzialità.

L'informazione relativa all'istruzione successiva da caricare viene mantenuta in un registro interno della CPU dedicato, denominato PROGRAM COUNTER (PC). Infatti contestualmente al caricamento dell'istruzione viene anche aggiornato il PC.

Tipicamente l'incremento del PC è automatico (di una quantità tale che possa puntare l'istruzione successiva).

Seguono le fasi di decodifica, esecuzione dell'istruzione e salvataggio degli eventuali risultati.

L'ARCHITETTURA DI RIFERIMENTO: LA MACCHINA MIPS

La prima versione dell'ISA MIPS è una delle prime architetture di tipo RISC ed è stata sviluppata da John L. Hennessy (inizio anni '80). L'ISA si è quindi evoluta in innumerevoli versioni ciascuna delle quali è stata implementata da diverse microarchitetture. Per questo si parla di famiglia di processori MIPS.

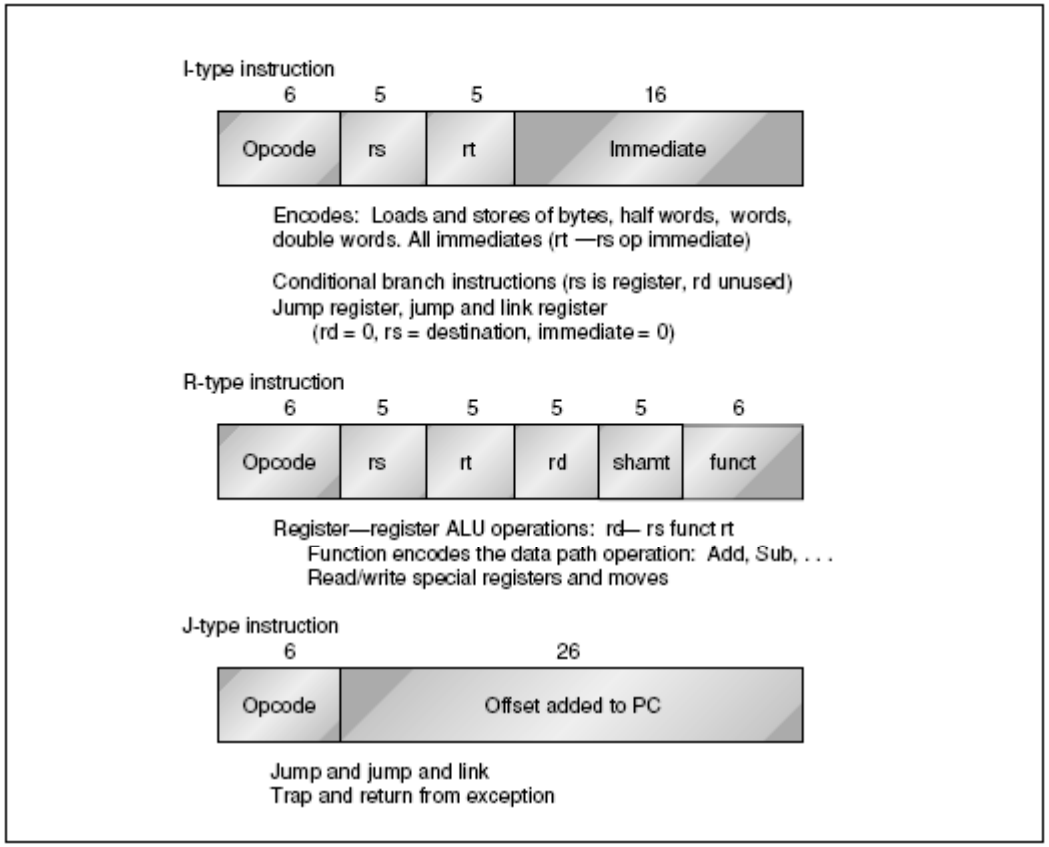
Le più evolute versioni di processori MIPS sono attualmente integrate in dispositivi elettronici embedded di largo utilizzo.

- Consolle per l'intrattenimento :
  e.g. la PS2 (mentre la PS3 utilizza il processore CELL) utilizza come main processor il MIPS R5900 a 64 bit
- Stampanti Laser
- Decoder TV Satellitare
- Router
- Navigatori satellitari per automobili
- Macchine fotografiche digitali

- L’architettura dei processori della famiglia MIPS è di tipo load-store (macchine registro-registro): le istruzioni ALU hanno due operandi in registri sorgente (unica eccezione: un operando è un immediato) e il risultato viene scritto nel registro destinazione
- Solo le istruzioni load e store hanno accesso alla memoria dati.
- Le istruzioni sono lunghe 4 byte.
- Indirizzamento in memoria: fa riferimento al byte (indirizzi consecutivi differiscono per 4 byte).
- Istruzioni di controllo (jump e branch): destinazione indicata mediante indirizzamento relativo, o rispetto al Program Counter o rispetto a un registro base.

- E’ prevista la presenza di un moltiplicare integrato per dati a 32 bit con risultato a 64 bit.
- I registri interni visibili del MIPS sono:
  - 32 registri general purpose a 32 bit, il registro zero contiene 0 e non può essere modificato,
  - due registri a 32 bit, Hi e Lo, per memorizzare rispettivamente i 32 bit più significativi e i 32 bit meno significativi del risultato di moltiplicazione
  - un registro Program Counter a 32 bit
  - un registro Interrupt Address Register
  - registri per elaborazione Floating Point

- Istruzioni di tipo L: load e store, istruzioni che operano su immediati, salti condizionati (con indirizzamento indiretto da registro)
- Istruzioni di tipo R: istruzioni ALU registro-registro



- Istruzioni di tipo J: salti incondizionati, chiamate a subroutine (indirizzamento relativo al PC).

INSTRUCTION SET

MIPS R2000

ISTRUZIONI DI ACCESSO ALLA MEMORIA (8)

Nome	Sintassi	Descrizione
Load word	lw rt, imm(rs)	Regs[rt]<={mem[addr],mem[addr+1],mem[addr+2],mem[addr+3]}
Load half word	lh rt, imm(rs)	Regs[rt]<={{16{mem[addr]$_{31}$}},mem[addr],mem[addr+1]}
Load byte	lb rt, imm(rs)	Regs[rt]<={{24{mem[addr]$_{31}$}},mem[addr]}
Load half word unsigned	lhu rt, imm(rs)	Regs[rt]<={{16{1'b0}},mem[addr]}
Load byte unsigned	lbu rt, imm(rs)	Regs[rt]<={{24{1'b0}}mem[addr]}
Store word	sw rt, imm(rs)	Mem[addr]<=Regs[rt]
Store half word	sh rt, imm(rs)	Mem[addr]<=Regs[rt]$_{[15:0]}$
Store byte	sb rt, imm(rs)	Mem[addr]<=Regs[rt]$_{[7:0]}$

5 load e 3 store

L'indirizzo di accesso (addr) è sempre la somma dell'immediato e del contenuto del registro puntato da rs: addr:=imm+Reg[rs]

ISTRUZIONI ARITMETICO-LOGICHE CON INDIRIZZAMENTO IMMEDIATO (10)

Nome	Sintassi	Descrizione
Addition Immediate (with overflow)	addi rt, rs, imm	Regs[rt]<=Regs[rs]+imm
Addition Immediate (without overflow)	addiu rt, rs, imm	Regs[rt]<=Regs[rs]+imm
And Immediate	andi rt, rs, imm	Regs[rt]<=Regs[rs] and imm
Or Immediate	ori rt, rs, imm	Regs[rt]<=Regs[rs] or imm
Xor Immediate	xori rt, rs, imm	Regs[rt]<=Regs[rs] xor imm
Sub Immediate(with overflow)	sub rt, rs, imm	Regs[rt]<=Regs[rs]-imm
Sub Immediatte(without overflow)	subu rt, rs, imm	Regs[rt]<=Regs[rs]-imm
Shift left logical	sll rd, rt, shamt	Regs[rd]<=Regs[rs]<<shamt
Shift right aritmetic	sra rd, rt, shamt	Regs[rd]=(Regs[rt]>>shamt)|({32{Regs[rs]$_{[31]}$}}<<(32-shamt))
Shift right logical	srl rd, rt, shamt	Regs[rd]<=Regs[rs]>>shamt

ISTRUZIONI ARITMETICO-LOGICHE CON INDIRIZZAMENTO REGISTER (19)

Nome	Sintassi	Descrizione
Absolute value	abs rt, rs	Regs[rt]=abs(Regs[rs])
Addition (with overflow)	add rd, rs, rt	Regs[rd]=Regs[rs]+Regs[rt]
Addition (without overflow)	addu rd, rs, rt	Regs[rd]=Regs[rs]+Regs[rt]
And	and rd, rs, rt	Regs[rd]=Regs[rs] and Regs[rt]
Multiply	mult rs, rt	Regs[hi]=(Regs[rs]*Regs[rt])$_{[63:32]}$ Regs[lo]=(Regs[rs]*Regs[rt])$_{[31:0]}$
Unsigned Multiply	multu rs, rt	Come mult senza tener conto del segno degli operandi
Negate (with overflow)	neg rd, rs	Regs[rd]=-Regs[rs]
Negate (without overflow)	negu rd, rs	Regs[rd]=-Regs[rs]
Nor	nor rd, rs, rt	Regs[rd]=Regs[rs] nor Regs[rt]
Not	not rt, rs	Regs[rd]=~Regs[rs]
Or	or rd, rs, rt	Regs[rd]=Regs[rs] or Regs[rt]
Shift left logical variable	sllv rd, rs, rt	Regs[rd]=Regs[rs]<<Regs[rt]
Shift right logical variable	srlv rd, rs, rt	Regs[rd]=Regs[rs]>>Regs[rt]
Shift right aritmetic variable	srav rd, rs, rt	Regs[rd]=(Regs[rt]>>Regs[rs]) | ({32{Regs[rs]$_{[31]}$}}<<(32-Regs[rs]))
Rotate left	rol rd, rs, rt	Regs[rd]=Regs[rs]+Regs[rt]
Rotate right	ror rd, rs, rt	Regs[rd]=Regs[rs]+Regs[rt]
Subtract(with overflow)	sub rd, rs, rt	Regs[rd]=Regs[rs]-Regs[rt]
Subtract(without overflow)	subu rd, rs, rt	Regs[rd]=Regs[rs]-Regs[rt]
Exclusive or	xor rd, rs, rt	Regs[rd]=Regs[rs] xor Regs[rt]

ISTRUZIONI DI COMPARAZIONE (12)

Nome	Sintassi	Descrizione
Set on less than	slt rd, rs, rt	If(Regs[rs]<Regs[rt]) Regs[rd]=1 else Regs[rd]=0
Set on less than unsigned	sltu rd, rs, rt	Come slt senza tenere conto del segno degli operandi
Set on less than immediate	slti rt, rs, imm	If(Regs[rs]<imm) Regs[rd]=1 else Regs[rd]=0
Set on less than immediate unsigned	sltiu rt, rs, imm	Come slti senza tenere conto del segno degli operandi
Set on equal	seq rd, rs, rt	If(Regs[rs]=Regs[rt]) Regs[rd]=1 else Regs[rd]=0
Set on greater than or equal	sge rd, rs, rt	If(Regs[rs]>=Regs[rt]) Regs[rd]=1 else Regs[rd]=0
Set on greater than or equal unsigned	sgeu rd, rs, rt	Come sge senza tenere conto del segno degli operandi
Set on greater than	sgt rd, rs, rt	If(Regs[rs]>Regs[rt]) Regs[rd]=1 else Regs[rd]=0
Set on greater than unsigned	sgtu rd, rs, rt	Come sgt senza tenere conto del segno degli operandi
Set on less than or equal	sle rd, rs, rt	If(Regs[rs]<=Regs[rt]) Regs[rd]=1 else Regs[rd]=0
Set on less than or equal unsigned	sleu rd, rs, rt	Come sle senza tenere conto del segno degli operandi
Set on not equal	sne rd, rs, rt	If(Regs[rs]!=Regs[rt]) Regs[rd]=1 else Regs[rd]=0

2 con indirizzamento immediato

10 con indirizzamento register

ISTRUZIONI DI SALTO CONDIZIONATO (17)

Nome	Sintassi	Descrizione
Branch on equal	beq rs, rt, LABEL	Salta se Regs[rs]=Regs[rt]
Branch on greater than equal zero	bgez rs, LABEL	Salta se Regs[rs]>=0
Branch on greater than equal zero and link	bgezal rs, LABEL	Salta e Regs[31]<=PC+4 se Regs[rs]>=0
Branch on greater than zero	bgtz rs, LABEL	Salta se Regs[rs]>0
Branch on less than equal zero	blez rs, LABEL	Salta se Regs[rs]<=0
Branch on less than and link	bltzal rs, LABEL	Salta e Regs[31]<=PC+4 se Regs[rs]<0
Branch on less than zero	bltz rs, LABEL	Salta se Regs[rs]<0
Branch on not equal	bne rs, rt, LABEL	Salta se Regs[rs]!=Regs[rt]
Branch on greater than equal	bge rs, rt, LABEL	Salta se Regs[rs]>=Regs[rt]
Branch on greater than equal unsigned	bgeu rs, rt, LABEL	Come bge senza tenere conto del segno degli operandi
Branch on greater than	bgt rs, rt, LABEL	Salta se Regs[rs]>Regs[rt]
Branch on greater than unsigned	bgtu rs, rt, LABEL	Come bgt senza tenere conto del segno degli operandi
Branch on less than equal	ble rs, rt, LABEL	Salta se Regs[rs]<=Regs[rt]
Branch on less than equal unsigned	bleu rs, rt, LABEL	Come ble senza tenere conto del segno degli operandi
Branch on less than	blt rs, rt, LABEL	Salta se Regs[rs]<Regs[rt]
Branch on less than unsigned	bltu rs, rt, LABEL	Come blt senza tenere conto del segno degli operandi
Branch on not equal zero	bnez rs, rt, LABEL	Salta se Regs[rs]!=0

ISTRUZIONI DI SALTO INCONDIZIONATO (4)

Nome	Sintassi	Descrizione
Jump	j target	PC={{(PC+4)$_{[31:28]}$},{target},{2'b00}}
Jump and link	jal target	Regs[31]<=PC+4 e PC={{(PC+4)$_{[31:28]}$},{target},{2'b00}}
Jump register	jr rs	PC=Regs[rs]
Jump and link register	jalr rs, rd	Regs[rd]<=PC+4 e PC=Regs[rs]

ISTRUZIONI DI SPOSTAMENTO DATI FRA REGISTRI (4)

Nome	Sintassi	Descrizione
Move from hi	mfhi rd	Regs[rd]=Regs[hi]
Move from lo	mflo rd	Regs[rd]=Regs[lo]
Move to hi	mthi rs	Regs[hi]=Regs[rs]
Move to lo	mtlo rs	Regs[lo]=Regs[rs]

ISTRUZIONI DI MANIPOLAZIONI COSTANTI A 32 BIT (1)

Nome	Sintassi	Descrizione
Load upper immediate	lui rt, imm	Regs[rt]<= {imm,{16'h0 } }

ISTRUZIONI SPECIALI (2)

Nome	Sintassi	Descrizione
No operation	nop	-
Return from exeption	rfe	PC<=IAR e int_enable=1

PSEUDO-ISTRUZIONI (2)

Nome	Sintassi	Descrizione	Implementazione
move	move Rs, Rd	Regs[Rd]=Regs[Rs]	add Rd, Rs, $0
load 32 bit immediate	li Rd, imm_{32}	Regs[Rd] = imm_{32}	lui Rd, imm[31:16] ori Rd, Rd, imm[15:0]

INSTRUCTION SET SEMPLIFICATO:

eMIPS

Inizieremo la trattazione analizzando un sottoinsieme minimo delle istruzioni del ISA MIPS R2000, che chiameremo eMIPS

Non è implementata la moltiplicazione, non vi sono quindi istruzioni di scambio fra registri interni.

Istruzioni presenti:

- Istruzioni di lettura e scrittura dalla memoria (2 su 8)
- Istruzioni aritmetico-logiche (5 su 40)
- Istruzioni di scambio di dati tra registri interni (0 su 4)
- Istruzioni di modifica del flusso di esecuzione
  - Salto incondizionato (0 su 4)
  - Salto condizionato (1 su 17)

- load word (**lw**): carica su un registro interno un valore dalla memoria
- store word (**sw**): scrive in memoria il valore contenuto in un registro
- add (**add**): somma del valore di due registri
- subtract (**sub**): sottrazione tra il valore di due registri
- and (**and**): and logico bit a bit del contenuto di due registri
- or (**or**): or logico bit a bit del contenuto di due registri
- set on less than (**slt**): scrive 1 sul registro destinazione se il primo operando è minore del secondo, zero altrimenti.
- branch on equal (**beq**): salto condizionato all'uguaglianza dei valori contenuti nei 2 registri confrontati

DESCRIZIONE	ASSEMBLY	FUNZIONALITA'
Load word	lw Rt, imm(Rs)	REGS[Rt] = Mem(REGS[Rs]+imm)
Store word	sw Rt, imm(Rs)	Mem(REGS[Rs] + imm) = REGS[Rt]
Add	add Rd, Rs, Rt	REGS[Rd] = REGS[Rs] + REGS[Rt]
Subtract	sub Rd, Rs, Rt	REGS[Rd] = REGS[Rs] – REGS[Rt]
And	and Rd, Rs, Rt	REGS[Rd]= REGS[Rs] and REGS[Rt]
Or	or Rd, Rs, Rt	REGS[Rd] = REGS[Rs] or REGS[Rt]
Set on less than	slt Rd, Rs, Rt	if (REGS[Rs]<REGS[Rt]) REGS[Rd] = 1
Branch if equal	beq Rs, Rt, label1	if (REGS[Rs]==REGS[Rt]) PC = PC+4+imm

L'ISA MIPS completo gestisce dati di 6 tipi diversi: a 8, 16, 32 bit in rappresentazione naturale o complemento a 2. Il sistema di memoria deve poter gestire tutti i tipi di dato esistenti, fornendo i dati in uscita nella maniera appropriata, a seconda del valore assunto dai bit di controllo associati (opType). Essendo il dato minimo a 8 bit, la locazione di memoria elementare deve essere di 8 bit.

```
@100 01 // Most significant byte del dato 1
@101 21 // byte 2 del dato 1
@102 AB // byte 3 del dato 1
@103 00 // Least significant byte del dato 1
@104 43 // Most significant byte del dato 2
@105 01 // Least significant byte del dato 2
@106 C1 // Dato 3
```

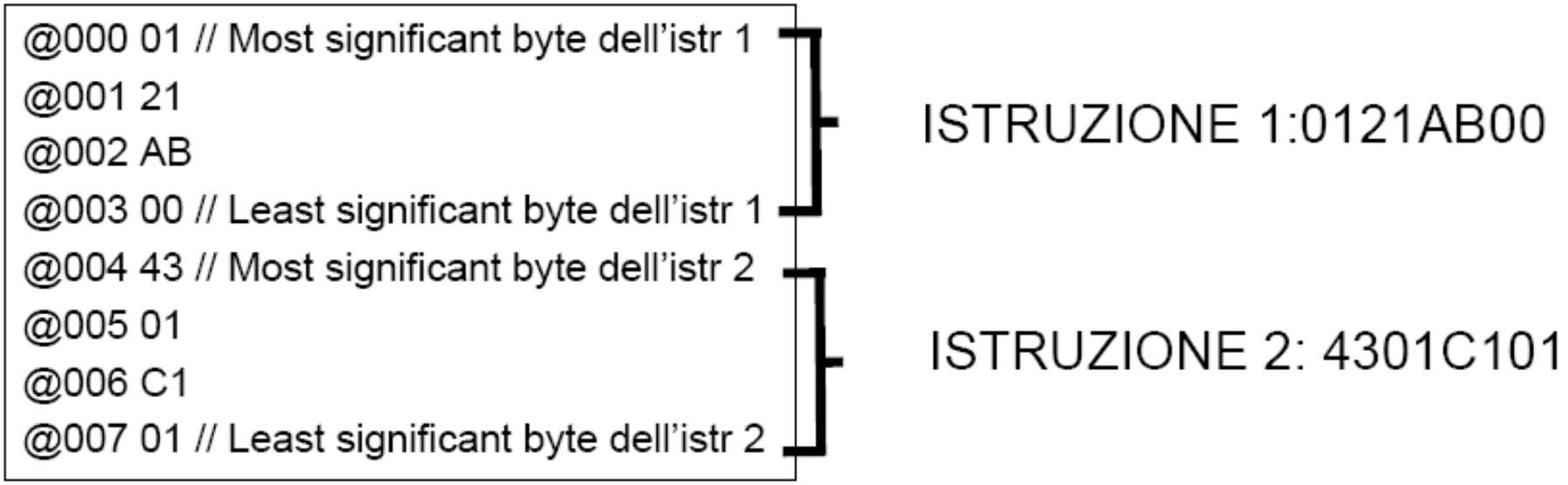
DATO 1, a 32 bit: 0121AB00

DATO 2, a 16 bit: 4301

DATO 3, a 8 bit: C1

```
always @(Addr)    // MODELLO VERILOG DELLA MEMORIA DATI
  case (opType)
   (word)      DataOut = {Data[Addr+0],Data[Addr+1],Data[Addr+2],Data[Addr+3]};
   (halfword) DataOut = {16'h0000,Data[Addr+0],Data[Addr+1]};
   (byte)       DataOut = {24'h000000, Data[Addr] };
endcase
```

Per non escludere la possibilità che al processore sia associato un sistema di memoria con memoria dati ed istruzioni non separate (memoria unica, modello di VonNeumann) è necessario che anche la memoria istruzioni sia indirizzabile un byte alla volta, nonostante le istruzioni siano sempre di 4 byte Ciò implica che l'aggiornamento del PC sia di 4 in 4.




```
always @(Addr) // MODELLO VERILOG DELLA MEMORIA INSTRUZIONI
 DataOut = {Data[Addr+0],Data[Addr+1],Data[Addr+2],Data[Addr+3]};
```

ESEMPI DI PROGRAMMAZIONE ASSEMBLY DEL PROCESSORE eMIPS

Specifica C:

int A,B,C; //dati a 32 bit

C=A+B;

Specifica aggiuntiva memoria dati (t=0):

@100 A

@104 B

Specifica aggiuntiva memoria dati (t=t_f):

@200 C

eMIPS Assembly

lw $1, 0x100($0)

lw $2, 0x104($0)

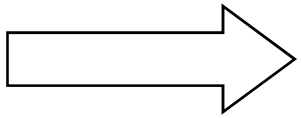
add $3, $1, $2

sw $3, 0x200($0)

L'esecuzione condizionale viene gestita sfruttando le istruzioni di salto condizionato (e non se si usa l'instruction set completo).

e.g.

if (cond) A;



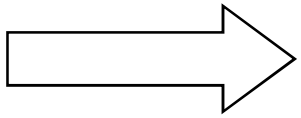
```
branch(cond==1), tagA
branch(1),end
tagA:
A
end:
```

---

e.g.

if (cond) A;

else B;



```
branch(cond==1), tagA
B
branch(1),end
tagA:
A
end:
```

Specifica C:
int A,B,C; //dati a 32 bit
If (A!=B) C=A+B;
else C=A-B;

Specifica aggiuntiva memoria dati (t=0):

@100 A

@104 B

Specifica aggiuntiva memoria dati (t=t_f):

@200 C

eMIPS Assembly

```
lw $1, 0x100($0)
lw $2, 0x104($0)
beq $1, $2, else_tag
if_tag:
add $3, $1, $2
beq $0, $0, store_tag
else_tag:
sub $3, $1, $2
store_tag:
sw $3, 0x200($0)
```

Specifica C:

```
integer acc; // dati a 32 bit
integer a[16];
acc = 0;
for( i=0; i<16;i++ )
acc=acc+a[i];
```

Specifica aggiuntiva memoria dati (t=0):

@100 a[0]
@104 a[1]
…
@13C a[15]

Specifica aggiuntiva memoria dati ($t=t_f$):

@200 acc

L'esecuzione ciclica di un loop viene gestita tramite l'utilizzo di un registro indice che deve essere:

- inizializzato in funzione del numero di iterazioni da eseguire;
- aggiornato ad ogni interazione;
- comparato con un registro di stop del conteggio alla fine di ogni iterazione, per decidere se il test di stop fallisce di reiterare il loop oppure di interromperlo.

L'accesso agli elementi di un vettore, con indice dipendente da quello del loop, può invece essere gestita tramite un registro detto "puntatore" che deve essere:

- inizializzato all'esterno del loop;
- utilizzato in ogni iterazione del loop per ottenere l'indirizzo della locazione di memoria in cui è memorizzato l'elemento del vettore a cui si deve accedere;
- aggiornato ad ogni iterazione.

In questo esempio si è deciso di utilizzare i registri come descritto di seguito:

- $1 acc
- $2 indice del loop, se nel registro acc viene caricato di default il primo elemento del vettore bisogna incrementare prima l'indice prima di fare il test
- $3 viene utilizzato come registro "puntatore" agli elementi del vettore
- $4 memorizza la costante da utilizzare per aggiornare il registro indice
- $5 memorizza la costante da utilizzare per aggiornare il registro puntatore
- $6 memorizza la costante utilizzata per il test di fine ciclo
- $7 buffer temporaneo

Specifica aggiuntiva memoria dati (t=0):

@100 a[0]

@104 a[1]

…

@13C a[15]

…

@400 0x1 // valore utilizzato per aggiornare l'indice

@404 0x4 // valore utilizzato per aggiornare il registro puntatore

@408 0xF // valore utilizzato per lo stop del conteggio

Specifica aggiuntiva memoria dati (t=t_f):

@200 acc

```
lw $1,0x100($0) ;inizializza ($1) col dato a[0]
add $2,$0,$0 ;inizializza ($2) a 0
add $3,$0,$0 ;inizializza ($3) a 0
lw $4,0x400($0) ;inizializza ($4) a 1
lw $5,0x404($0) ;inizializza ($5) a 4
lw $6,0x408($0) ;inizializza ($6) a F
loop:
add $2,$2,$4 ;aggiorna il registro indice sommando 1
add $3,$3,$5 ;aggiorna il registro puntatore sommando 4
lw $7, 0x100($3) ;carica a[i] su $7, (e.g. it1=> 100+REG[$3]=104:=a[1])
add $1,$1,$7 ;aggiorna l'accumulatore sommando il buffer
beq $2,$6,store ;fa il test del registro indice, se $2=$6 interrompe il ciclo
beq $0,$0, loop ;salta all'inizio del ciclo
store:
sw $1,0x200($0) ;scrive il risultato in memoria
```

La soluzione proposta non è l'unica possibile, esistono soluzioni assembly volte a "risparmiare" registri.

E' possibile, per esempio, utilizzare un solo registro che accorpi le funzionalità di registro indice per il test di fine ciclo e di registro puntatore.

L'inizializzazione e l'aggiornamento del registro di accesso ai dati, nonché il test di fine ciclo, vanno gestiti in funzione del fatto che il registro puntatore richiede l'accesso a dati di 32 bit.

In questa seconda versione del codice si è deciso di utilizzare i registri come descritto di seguito:

- $1 acc
- $2 indice del loop e "puntatore" all'elemento corrente del vettore
- $3 buffer temporaneo
- $4 memorizza la costante da utilizzare per aggiornare il registro indice/puntatore

Specifica aggiuntiva memoria dati (t=0):

@100 a[0]

@104 a[1]

…

@13C a[15]

…

@400 0x3C // valore utilizzato per inizializzare l'indice/puntatore

@404 0x4 // valore utilizzato per aggiornare l'indice/puntatore

Specifica aggiuntiva memoria dati (t=t_f):

@200 acc

```
lw $1,0x100($0) ;inizializza l'acc ($1) col dato a[0]
lw $2,0x400($0) ;inizializza ind/punt ($2) a 3C
lw $4,0x404($0) ;inizializza l'offset di aggiornamento ($4) a 4
loop:
lw $3, 0x100($2) ;carica a[i] su $3,(it1=> 100+REG[$2]=13C:=a[15])
sub $2,$2,$4 ;aggiorna ind/punt ($2) sottraendo 4 (it1 => 3C-4=38)
add $1,$1,$3 ;aggiorna l'accumulatore (it1 => a[0]+a[15])
beq $2,$0,store ;fa il test del registro indice, se $2=$0 interrompe il ciclo
beq $0,$0, loop ;salta all'inizio del ciclo
store:
sw $1,0x200($0);scrive il risultato in memoria
```

T0:

a @ 0x104

b @ 0x108

specifica aggiuntiva:

1 @ 0x10c

```
while (a < b) {
a = a + 1
b = b - 1
}
```

```
lw $1, 0x104($0)
lw $2, 0x108($0)
lw $4, 0x10C($0)

while_tag:
slt $3, $1, $2; #$3=1 se $1<$2 => a<b condizione per effettuare il while
beq $0,$3, fine
add $1,$1,$4
sub $2,$2,$4
beq $0,$0,while_tag
fine:
sw $1, 0x104($0)
sw $2, 0x108($0)
```

Da risolvere…

T0:

a @ 0x104

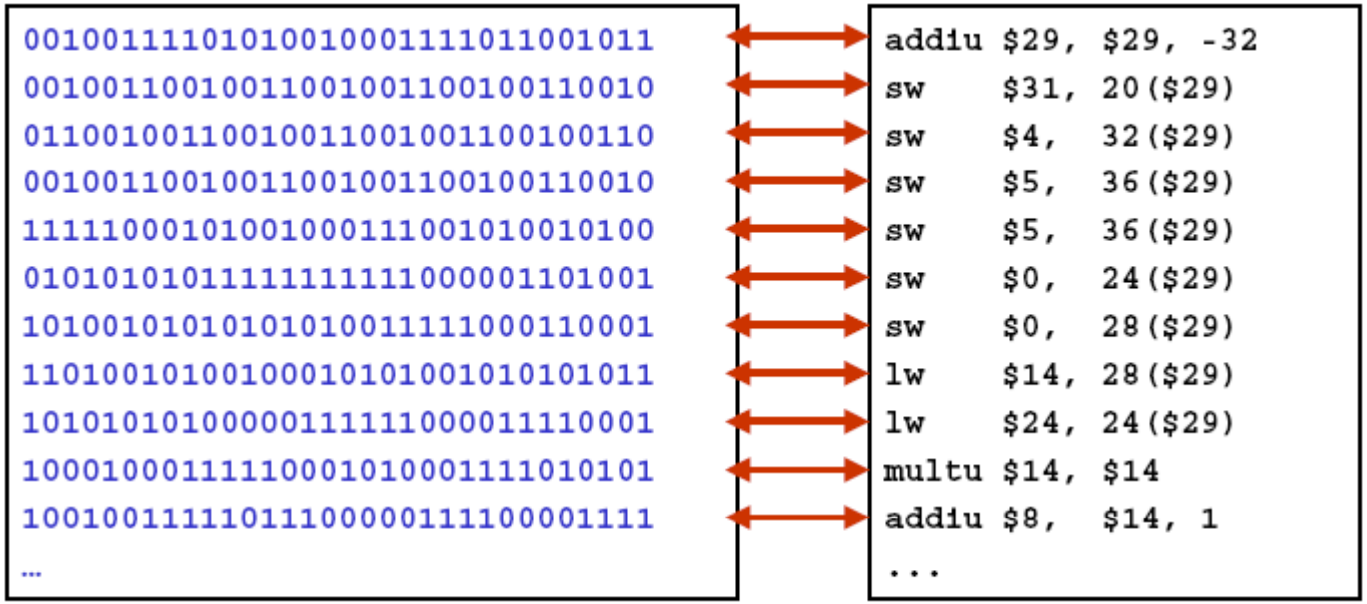
b @ 0x108

c @ 0x10c

Specifica aggiuntiva: ?

```
if (a < 0) {
    c = a;
    b = b+1
}
else {
    c =b;
    a = a-1;
}
```

PASSAGGIO
ASSEMBLY-LINGUAGGIO MACCHINA



L'assembly è la rappresentazione simbolica del linguaggio macchina, più comprensibile perché utilizza istruzioni invece di bit per rappresentare operazioni. Il linguaggio macchina invece è basato su numeri binari ed è utilizzato dai computer per memorizzare ed eseguire fisicamente i programmi.

ADD

Add Word

Format: ADD rd, rs, rt MIPS I

Purpose: To add 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ←GPR[rs] + GPR[rt]
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ←sign_extend(temp31..0)
endif
```

Exceptions:

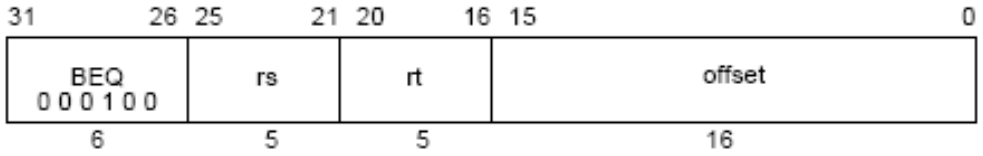
Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but, does not trap on overflow.

BEQ

Branch on Equal



Format: BEQ rs, rt, offset MIPS I

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:   tgt_offset ← sign_extend(offset || 0^2)
     condition ← (GPR[rs] = GPR[rt])
I+1 :if condition then
          PC ← PC + tgt_offset
     endif
```

Exceptions:

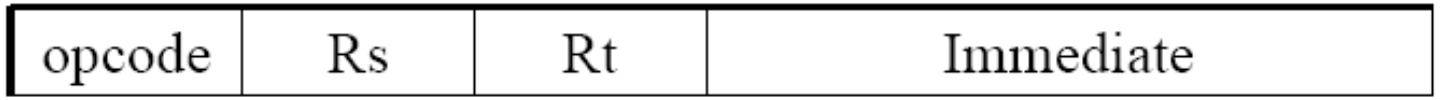
None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

lw $2, 0x104($0)

Istruzioni I-type: **lw Rt, Imm(Rs)**



ASM Op=6h'23 Rs=$0 Rt=$2 Imm=16h'104

BIN 100011 00000 00010 0000000100000100

BIN 1000 1100 0000 0010 0000 0001 0000 0100

HEX 8 c 0 2 0 1 0 4

HEX 8c020104

MATERIALE DI RIFERIMENTO

- CONCETTI BASE E DEFINIZIONI GENERALI:

  Patterson, Hennessy; *"Computer Architecture, A Quantitative Approach"*: capitolo 2 – Instruction Set Principles and Examples

- L'ARCHITETTURA DI RIFERIMENTO: LA MACCHINA MIPS

  Patterson, Hennessy; "Computer Architecture, A Quantitative Approach": capitolo 2 – Instruction Set Principles and Examples

  Patterson, Hennessy; "Computer Architecture, A Quantitative Approach": Appendice C – A Survey of RISC Architectures for Desktop, Server, and Embedded Computers