

## CALCOLATORI ELETTRONICI

**Effetto n+1:** effetto per il quale si è cercato sempre di aumentare le capacità dei calcolatori sfruttandole a pieno e generando in questo modo tabelle di corrispondenza sempre più articolate (con istruzioni arrivate ad essere di anche 54 byte); per questo effetto ogni volta che si raggiungono le n istruzioni e si rende necessario inserire la  $n+1$  esima si vanno a penalizzare le performance di tutte le precedenti:

e.g. se ho 128 istruzioni da codificare necessito di 7 bit ma, qualora intendessi implementare la 129esima si renderebbe necessario un decoder a 8 bit per una sola istruzione, con conseguente perdita di performance e di ottimizzazione anche per le precedenti 128;

Si è passati dallo Z-80 di decine di anni fa con pagine di istruzioni solo per la somma ai MIPS con soltanto tre istruzioni per la somma

La perdita dal punto di vista prestazionale è su due fronti principalmente, ovvero nelle fasi di fetch e di decodifica:

- **FETCH**: la fase di fetch è minata dall'effetto n+1 in quanto esso rende le istruzioni sempre più lunghe e formate da sempre più bit/byte, con conseguente lavoro maggiore per il trasferimento dell'istruzione dalla memoria alla CPU e viceversa (vedi esempio 128 -> 129 istruzioni)
- **DECODIFICA**: la decodifica è peggiorata dall'effetto n+1 in quanto un decoder necessita di n linee di ingresso per coprire *almeno*  $2^n$  istruzioni. Ciò implica che, come nell'esempio precedente, aumentando di un'unità il # di istruzioni c'è il rischio di trovarsi con un decodificatore effettivamente complesso dal punto di vista strutturale e circuitale

Per questo motivo Patterson ha proposto di minimizzare le istruzioni decodificabili da un calcolatore (alcune talvolta non venivano nemmeno considerate dai compilatori) a favore di un'efficienza massima nelle poche necessarie istruzioni che vengono eseguite più di frequente.

Si tende pertanto a prediligere le performance del poco necessario piuttosto che la ricchezza di funzioni inutilizzate, che rallenterebbero inutilmente le performance del calcolatore

Si passa pertanto dall'architettura **CISC** all'architettura **RISC**:

- **CISC**: (*Complex Instruction Set Computer*): indica un'architettura per microprocessori formata da un set di istruzioni contenente operazioni complesse (lettura/scrittura memoria, modifica, salvataggio...) eseguite in una singola istruzione

Prediligono, come filosofia di base, una riduzione del divario tra linguaggio macchina e linguaggio di alto livello, con un set ricco di istruzioni, a sfavore della complessità e della ricchezza di queste ultime, con conseguente perdita di prestazioni rispetto al Risc in quanto la dimensione di ogni istruzione penalizza le performance

- **RISC** (*Reduced Instruction Set Computer*): indica un'architettura introdotta da Patterson che ha come scopo la riduzione al minimo indispensabile delle istruzioni di un calcolatore dal punto di vista del linguaggio assemblativo, con tuttavia la ricerca dell'ottimizzazione di quei pochi comandi fondamentali come scopo. Meno comandi usati, meno occupazione inutile di bit per il singolo comando e velocizzazione delle due fasi di fetch e di decode (vedi esempio n+1)

Lo svantaggio delle RISC è il fatto che il programmatore deve adeguarsi ai pochi comandi essenziali che sono presenti, a favore tuttavia di un minor spreco di risorse, di un'eliminazione di comandi inutili e di una maggior efficienza esecutiva

La cpu percepisce soltanto 0 e 1 dal mondo esterno, pertanto il significato di una sequenza di questi ultimi dipende dalla situazione in cui mi trovo e dal calcolo che sto svolgendo; essendo infatti una macchina sincrona sa esattamente cosa succede in ogni istante all'interno del calcolatore e si aspetta un determinato tipo di dato piuttosto che un altro dipendentemente dal contesto intorno ad essa.

Il processore utilizzato, **MIPS**, è un processore a 32 bit che da quindi la possibilità di

rappresentare  $2^{32}$  configurazioni differenti. Tuttavia è necessario notare che possono presentarsi situazioni di errore all'interno dei calcoli, in v.a. o in C2:

e.g. 1111 +	tale risultato sarebbe esatto in un'ottica di calcolo in C2 (-1 + +1), ma errato in un'ottica di calcolo in v.a. in quanto senza tener conto del riporto otterrei 0 sommando due positivi.
0001 =	
-----	
(1)000	
0111 +	tale risultato sarebbe errato calcolando in C2, in quanto dai positivi finisco nei negativi ( <b>Overflow</b> ) ma sarebbe perfettamente corretto calcolando in v.a.
0001=	
-----	
1000	

Per la prevenzione dell'Overflow solitamente si utilizzano i registri dei flag:

Essi sono registri che segnalano l'eventuale presenza di riporto o overflow alla Alu, permettendo un'esecuzione corretta dei calcoli e dei risultati, chiamando un salto qualora si presentasse una delle due situazioni precedenti (salto a sottoprogramma che terrebbe conto della situazione aggiustando il risultato)

Add ...  
JP OW ....  
....

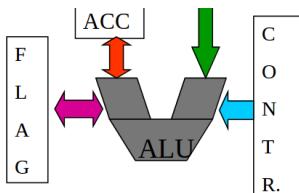
E' più lenta di quella dei MIPS qualora non si verificassero condizioni di OF o riporto poiché c'è il tempo di informazione e interrogazione dei flag da considerare

Il **MIPS** tuttavia non presenta tale registro, gestendo in maniera differente l'eventuale presentarsi di una delle due situazioni precedenti; esso parte dall'idea che non ci siano situazioni di overflow o di riporto, eseguendo le istruzioni senza JP:

Add ...  
....

E' infatti la stessa CPU a gestire l'eventuale esecuzione o meno di un programma con chiamata a sottoprogramma nel caso di un overflow (*chiamata interna*); In assenza di interrupt è più efficiente, ma qualora si verificasse l'overflow avrei una chiamata a sottoprogramma più lenta della subroutine fissata dell'altro tipo di gestione.

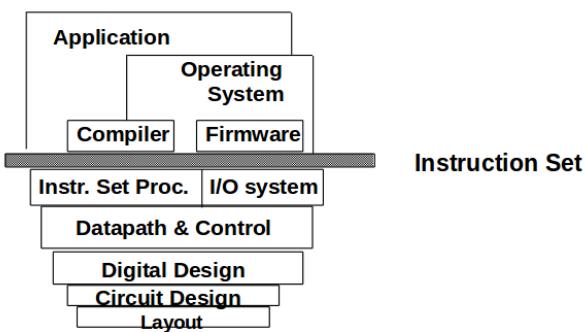
Qual è più efficiente delle due? Come sempre, *dipende dallo scopo*. E' necessario eseguire un calcolo probabilistico sulle possibilità di overflow in relazione all'utilizzo che deve essere fatto del processore (nel caso dei MIPS si è valutato che  $2^{32}-1$  rappresentazioni fossero un # sufficiente a non rendere necessario il controllo dei flag dell'overflow, in quanto si verifica poco di frequentemente in relazione all'utilizzo, con conseguente miglioramento prestazionale anche a fronte delle poche volte che si presenta)



A differenza di quanto avviene nelle CPU convenzionali all'interno dei MIPS **è la ALU che comunica direttamente con il controllore informandolo dell'eventuale overflow.**

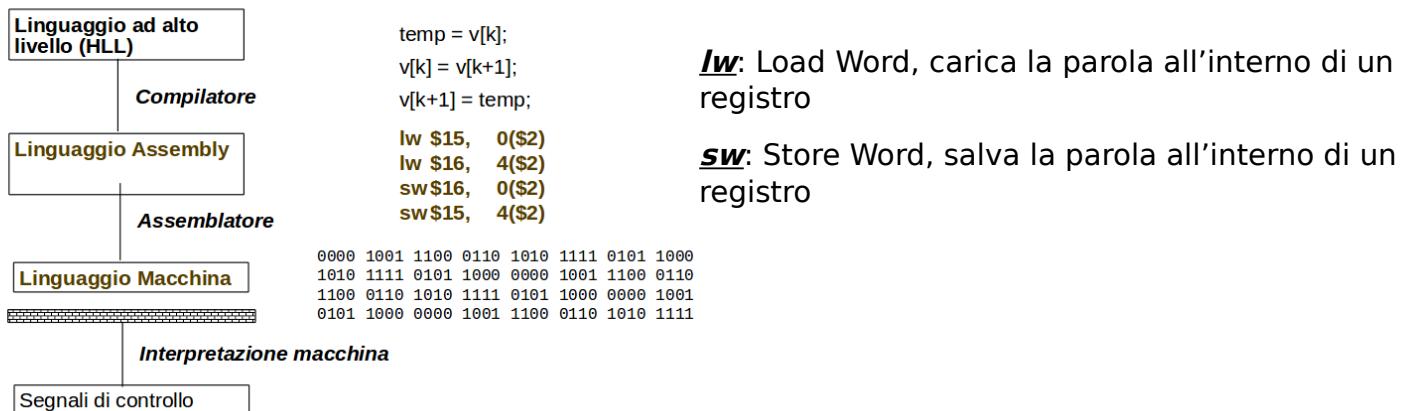
E' necessario implementare un modo per inserire all'interno dell'istruzione tutte le informazioni necessarie: benchè infatti trattandosi di un processore RISC non abbia grandi modalità di indirizzamento, essendo gli indirizzi di 32 bit, i registri 32 (ovvero altri 5 bit, 2<sup>5</sup> = 32), le istruzioni altri 32 bit etc etc e dovendo contenere l'istruzione sia codice operativo che operando o suo indirizzo, si rende necessaria una tecnica particolare affinché sia possibile inserire nell'istruzione tutti i dati necessari. (\*)

**ASTRAZIONE:** è la filosofia per cui chi si occupa della progettazione di un sistema complesso non è necessario che conosca alla perfezione tutto il sistema; esso infatti viene suddiviso in più livelli astratti, in cui ogni progettista ha competenze riguardo a uno o pochi di essi, non preoccupandosi di ciò che accade dal punto di vista progettuale a livelli più elevati o più bassi.



La differenza tra i linguaggi di alto livello e basso livello è soprattutto nella complessità del codice e nell'efficienza del linguaggio (più è vicino al linguaggio macchina più l'esecuzione è rapida). Quando però scriviamo un programma ad alto livello esso viene convertito da un interprete o un compilatore in linguaggio di basso livello e successivamente in linguaggio macchina.

Così se ad esempio volessi spostare due elementi di un vettore invertendoli, ad alto livello sarebbe questione di poche righe di codice, in linguaggio assemblativo sarebbe un po' più complesso ed articolato e in linguaggio macchina sarebbe una sequenza di 0/1.



**ASSEMBLATORE:** gestisce le tabelle di corrispondenza tra comandi in linguaggio assemblativo e l'equivalente istruzione in linguaggio macchina. Ha il grosso svantaggio di dipendere dall'architettura del processore di riferimento (non essendo per questo portabile) e di avere una bassa leggibilità per la sua lontananza dal linguaggio naturale.

Per sopperire a queste problematiche sono nati i *compilatori/interpreti* che hanno il grosso vantaggio di gestire linguaggi indipendenti dalla macchina (cambia l'interprete, non il sorgente del codice) e di essere molto più leggibili per la maggior vicinanza con il linguaggio naturale.

Hanno tuttavia degli svantaggi:

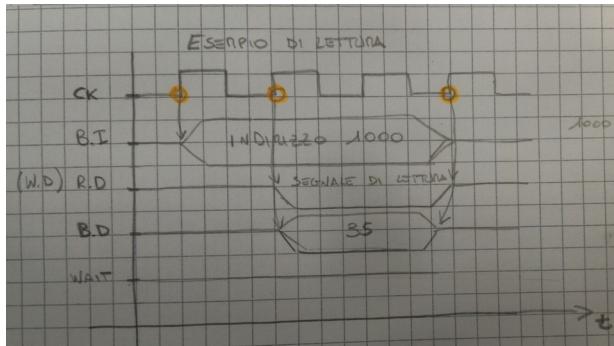
- occupano molta più memoria di un programma assmblato, in quanto è necessaria tutta l'interfaccia di traduzione in linguaggio macchina (più laboriosa dei linguaggi assemblativi)
- prestazionalmente è meno efficiente di un linguaggio assemblativo, proprio per la sua maggior distanza dal linguaggio macchina.

La scelta di una tipologia di linguaggio piuttosto che un'altra dipende principalmente dall'applicazione del programma che devo andare a fare → se necessito di una rapida risposta a discapito della portabilità e leggibilità, e.g. ABS, allora l'assemblativo sarà preferibile.

Qualora dovessi aver meno necessità di prestazioni al millisecondo ma dovessi preferire la portabilità del linguaggio allora un alto livello sarebbe migliore.

Nelle istruzioni precedentemente analizzate cosa cambia tra lettura e scrittura, nelle due fasi?

- > il codice operativo è differente
- > per l'*address bus* non c'è alcuna differenza, prescindendo da r o w
- > il *control bus* cambia la modalità (r/w) a seconda del comando
- > il *data bus* cambia la direzione di comando a seconda di r/w
- > Il tempo di attivazione del *data bus* varia a seconda di quale delle due operazioni io stia



svolgendo. In scrittura infatti il dato è già pronto e viene mandato subito attraverso il databus all'indirizzo interessato. In lettura invece è necessario prima trovare la cella, interellarla e solo allora si riceve il dato sul bus.

Il fatto che la memoria sia dal punto di vista della velocità molti ordini di grandezza inferiore rispetto alla CPU spiega perché *vi siano così tanti registri affiancati alla CPU*.

Nel MIPS ad esempio ho 32 registri adiacenti, che essendo interni risultano essere più performanti della lettura diretta in memoria al di fuori della CPU. Questa gestione dei registri per la velocizzazione dei processi implica tuttavia che **non sia possibile inserire i registri direttamente nell'istruzione**

Se infatti il # massimo di bit per tutta l'istruzione è 32, non posso usarli tutti solo per l'indirizzo interessato; la soluzione è l'utilizzo dell'**indirizzamento indiretto a registro**, così facendo inserisco all'interno dei registri adiacenti l'indirizzo a cui voglio recarmi, utilizzando soltanto 5 bit ( $2^5 = 32$ ) per identificare i registri invece di usarli tutti e 32 per l'indirizzo effettivo

### **ANALISI ISTRUZIONI**

**Iw \$15, 0(\$2)**

**Iw \$16, 4(\$2)**

**sw \$15, 0(\$2)**

**sw \$16, 4(\$2)**

\* Iw ed sw sono i nomi delle istruzioni da eseguire (LoadWord, StoreWord) → 6 bit

\* \$15 e \$16 sono i destinatari dei trasferimenti nelle letture e l'origine del dato nelle scritture → 5 bit

\* (\$2) tra parentesi indica il *puntatore al registro* in cui è indicato l'indirizzo effettivo in memoria dell'indirizzamento (\$0-\$31 con multipli di 4) → 5 bit

\* qualora volessi eseguire un indirizzamento diretto in memoria sarei costretto a spezzare l'istruzione in due passaggi.

Ho utilizzato 16 dei 32 bit, come utilizzo gli altri 16?

Avendo a che fare con un numero comunque limitato di registri rispetto alle dimensioni della memoria devo ottimizzare al massimo il loro utilizzo; per questo quando ho a che fare con dati vettoriali o matriciali non utilizzo un puntatore per ogni casella di memoria, ma uso sempre lo stesso registro a cui aggiungo/tolgo un **offset** che mi dica di quanto spostarmi dal primo dato per ottenere il secondo che mi interessa. Essi sono gli 0 o 4 che compaiono prima del registro interessato che punta alla memoria.

Le caselle di memoria sono solitamente di 4 byte e l'offset è rappresentato in C2 per due motivi:

- ° la somma e la differenza si effettuano entrambe come somma algebrica di numero
- ° posso rappresentare sia positivi che negativi

996	1000	1004	1008	1012	1016	1020	1024	1028
-----	------	------	------	------	------	------	------	------

In cui l'offset indica lo spostamento, ad esempio, dalla cella 1000 alla 1004, utilizzando un solo puntatore a registro.

(obs: qualora superassi l'intervallo di validità della mia rappresentazione è l'assemblatore che mi comunica in fase di traduzione l'eventuale errore commesso)

N.B. fetch ed execute sono indipendenti anche nell'esecuzione, pertanto non ho sovrapposizione o conflitto nella gestione del dato rispetto all'istruzione: prima viene tradotta questa e poi prelevato il primo.

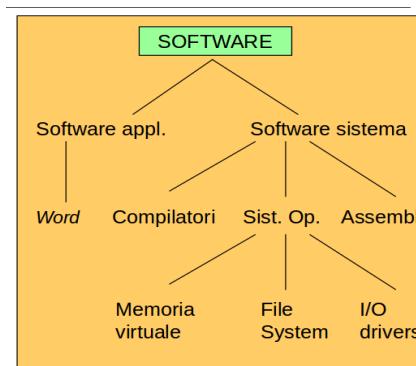
## STRUTTURA DEL SW

i linguaggi HLL permettono una progettazione con un linguaggio estremamente leggibile e ~ a quello naturale con indipendenza dalla macchina di lavoro e dalla sua architettura, oltre a una maggior concisione rispetto al linguaggio macchina. E' tuttavia meno efficiente

**Sistema operativo:** programma separato che supervisiona l'utilizzo della macchina da parte dei programmi utente, rendendo a quest'ultimo l'uso del calcolatore completamente trasparente.

**Sw di sistema:** insieme dei programmi che forniscono servizi (Sistema operativo, compilatori, assemblatori...)

**Sw applicativo:** programmi utente o mirati all'utente (editors, spreadsheet...)



**Architettura del set di istruzioni:** è un'interfaccia tra hw e sw di basso livello che standardizza il formato delle istruzioni e i paterni di bit a livello di linguaggio macchina. Ha il grosso vantaggio delle implementazioni differenti della stessa macchina, di contro però talvolta non permette di sfruttare a pieno o avvalersi delle nuove tecnologie.

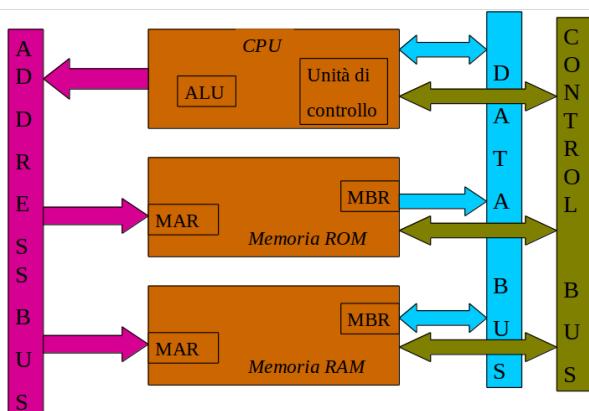
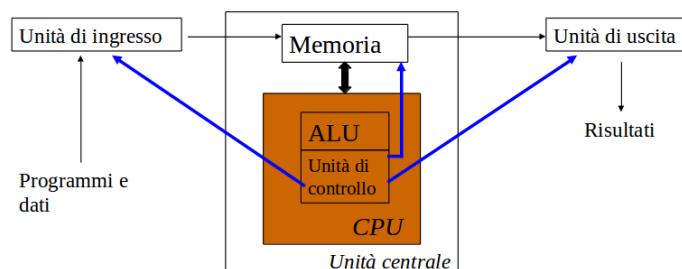
Standardizza le istruzioni e fa da tramite tra l'hw vero e proprio e il codice di basso livello;

## MACCHINA DI VON NEUMANN

E' un'architettura concettuale che prescinde dalla tecnologia e vale ancora oggi per molti calcolatori. Rappresenta la schematizzazione di un calcolatore.

Esso è una *macchina sincrona* e i bit hanno un determinato significato dipendentemente da cosa la CPU si

aspetta e dal contesto in cui mi trovo (per questo motivo la modifica di un solo bit nelle istruzioni può portare a notevoli errori nei risultati, dato che ogni secondo vengono eseguite mld di istruzioni)



## UNITA' CENTRALE:

E' formata da varie parti che compongono la base del calcolatore dal punto di vista hw: le componenti principali sono la CPU, con la ALU (che effettivamente svolge le operazioni di calcolo del calcolatore) e le memorie:

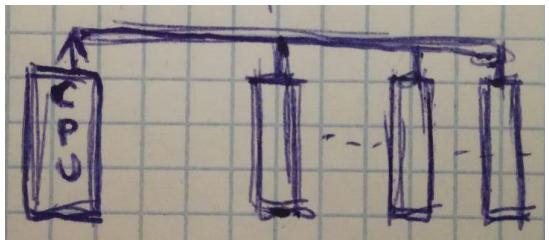
- RAM
- ROM

Per gestire le informazioni entrano in azione i bus, e sono di tre tipologie:

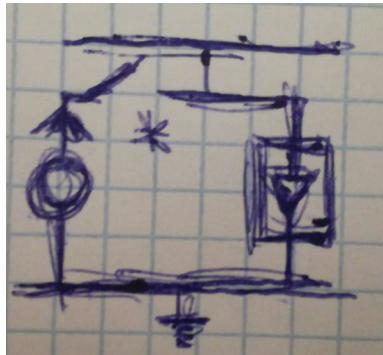
> Bus degli indirizzi: è il bus responsabile di chiamare la locazione in memoria a cui la CPU punta. Tutti gli indirizzi ricevono la chiamata ma soltanto l'interessato risponde alla CPU.

Il # di linee dipende dalla finalità del calcolatore ( $n$  linee  $\rightarrow 2^n$  indirizzi);

*obs:* non ho conflitto di comunicazione in quanto è sempre la CPU a fare da master e la memoria da slave (a meno del caso di DMA); tuttavia il fatto che i dispositivi NON siano ideali implica che vi sia caduta di potenziale, con conseguente rischio di errata trasmissione dell'informazione (in pilotaggio devo assegnare V minima sufficiente alla comunicazione senza errori)



> Bus dei dati: è di fatto il bus che ha il compito di trasferire fisicamente i dati. Può essere



utilizzato sia dalla CPU che dalle periferiche in r&w. Pertanto è necessario che si evitino conflitti di istruzioni sulle linee (*conflitto di tensione*, se due elementi vogliono scrivere in contemporanea, con conseguente scarica sul cavo e rischio di fulminare il collegamento a causa della tensione molto elevata)

Per evitare che questo si verifichi devo introdurre *un'impedenza* molto elevata tramite un interruttore che stacchi fisicamente il generatore quando l'elemento è in lettura e chiuda il c.to quando è in scrittura (**può essere chiuso solo un c.to per volta**).

Sono i cosiddetti stadi tristate: - piloto 1 – piloto 0 – c.to staccato

in ricezione

Il comando su quale interruttore chiudere spetta alla CPU e dal bus dei controlli con linee di read e write (in cui uno solo è in write e n in read)

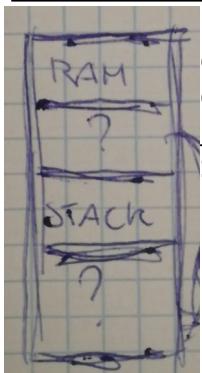
> Bus dei controlli: è il bus che effettivamente controlla le operazioni interne alla CPU, attraverso dei segnali di controllo delle azioni che si svolgono. Esempi di segnali sul bus degli indirizzi sono il segnale di reset, il segnale di interrupt (gestito attraverso delle priorità), il segnale di richiesta/concessione DMA ... ..

? : Se il tempo concesso alle memorie non è sufficientemente lungo?

La linea pilotata trasmette dati inaffidabili ed incompleti, devo pertanto prevedere una situazione del genere  $\rightarrow$  linea di wait: la CPU capisce che non deve continuare con le operazioni ma "aspettare" il completamento della trasmissione delle memorie.

Un'altra linea presente sul bus dei controlli per i dispositivi I/O è la *memory request*: tramite questa linea si definisce chi colloquia con la CPU

#### MAPPA DI MEMORIA:



è l'associazione tra gli indirizzi raggiungibili e i dispositivi connessi ai bus (NB: esistono indirizzi a cui non ho interlocutori; il numero di indirizzi utilizzati dipende dallo scopo del calcolatore che sto progettando, ma se non mi servono possono rimanere inutilizzati)

Se ad esempio miservo 10Mb non ha senso utilizzare un TB di memoria né indirizzarne tanta, anche se magari i data bus della CPU lo consentirebbero

Cosa accadrebbe se tentassi di accedere ad una porzione mappata come vuota, che pertanto non ha un corrispettivo indirizzo in memoria?

E' necessario fare il distinguo delle due fasi:

> Scrittura: la CPU invia l'input di scrittura all'indirizzo deciso e, secondo lei, la scrittura va a buon fine, con il dato che in realtà viene perso poiché non esistendo l'indirizzo viene inviato a vuoto; l'assemblatore inoltre non si accorge dell'errore in quanto è un problema in

esecuzione, non a livello di sintassi o semantica del codice.

> **Lettura**: durante la lettura la CPU invia la richiesta all'indirizzo interessato, con la risposta che è nulla essendo il suddetto inesistente. La lettura avviene sui dati residui sul databus, con errore di lettura dei dati e, conseguentemente, del programma.

L'unico modo di accorgersi di un errore del genere è utilizzando un ambiente di collaudo (*i.e.* IDE) che abbia la mappatura di memoria di riferimento integrata e segnali eventuali errori di indirizzamento in lettura o scrittura

**Databus**: capacità di elaborazione del processore (quanti bit possono essere elaborati in parallelo)

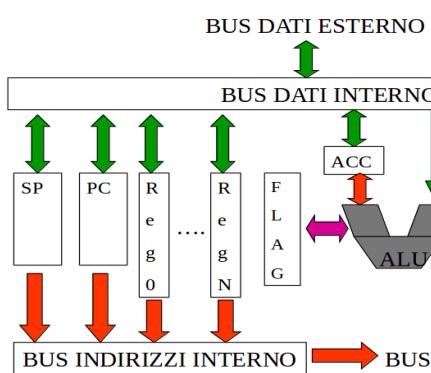
**Address bus**: indica la capacità di memorizzazione del processore ( $2^n$  celle di memoria, dove n è il # di bit del bus)

**Capacità di indirizzamento**: indica il numero di celle differenti a cui posso accedere (e.g.  $2^{10}$  byte → 1KB)

NB: la capacità di elaborazione del processore indica quanti bit possono essere elaborati *contemporaneamente*, non la precisione effettiva che poco raggiungere nel calcolo; potrei aver bisogno di una precisione per cui servono 16 bit, con a disposizione soltanto 8 bit → per fare ciò devo semplicemente dividere il comando in due mandate da 8 bit ciascuno, analizzando prima la parte di minor peso e poi la parte di maggior peso. L'utilizzo di processori più o meno capaci da questo punto di vista dipende dalle esigenze di progetto: se si necessitano tempi di risposta quasi immediati si preferirà un processore più capace (la perdita maggiore del dividere in più mandate un'istruzione sta in qualche microsecondo d'esecuzione.); se si necessita di occupare meno spazio a discapito della piccolissima perdita si preferirà un processore a minor capacità elaborativa, dividendo in più momenti le istruzioni (avere il doppio di *satabus* implica un aumento considerevole dello spazio occupato, sia in termini di silicio che in termini di processore ultimato)

Single Chip: uniscono in un unico c.to integrato più di uno dei blocchi costituenti un microcalcolatore, o eventualmente tutti

## ARCHITETTURA CPU:



nel MIPS non è presente il flag di overflow perché la situazione viene gestita in maniera differente, con la ALU che comunica con il blocco di controllo che gestisce le eccezioni ed eventuali riporti

Nel mips inoltre *non è presente* l'accumulatore, a causa di un'equivalenza dei 32 registri interni

*Obs*: non posso avere istruzione del tipo ADD A, (1000) poiché non ho abbastanza bit in quanto sono 32 i totali dell'istruzione; dovrò utilizzare un'istruzione nella forma ADD, \$R1, \$R2 in cui faccio riferimento ai registri interni

Nemmeno come operatore può essere inserito il valore, a meno di casi particolari, poiché avrei un utilizzo non ottimale dei 32 bit che ho a disposizione, che talvolta potrebbero anche non bastare

Nei MIPS i registri forniscono:

- *operandi*
- *indirizzi*
- *incamerano i risultati*

La fase di lettura nel MIPS (durante la fase di fetch) ha soltanto un'operazione di lettura, che contiene tutte le informazioni necessarie al processore

**Stack Pointer:** gestisce le chiamate a sottoprogramma partendo dallo stack, salvando il program counter prima del salto nello stack

All'interno del MIPS questa non è la soluzione principale, prediligendo l'utilizzo di *un solo registro per il pc*, detto **ReturnAddress** (reg #31)

Il fatto di avere solo un registro per le chiamate implica che per eseguire sotochiamate annidate sia necessario gestire manualmente lo spostamento e il salvataggio del PC nello stack e il richiamo con lo sp; essendo tuttavia casi remoti si è preferito implementare una soluzione notevolmente più rapida (RA è un registro interno, a differenza dello stack)

**JAL (JumpAndLink):** è il comando incaricato di salvare il PC all'interno del RA e saltare alla sotochiamata; crea il collegamento a cui è necessario tornare

**JR (JumpToRegister):** è il comando che consente il salto all'indirizzo del registro indicato

e.g. JR \$ra dove \$ indica che il valore successivo sarà un registro

Qualora io debba effettuare una chiamata nidificata con JL andrei a perdere il PC, a meno di salvarlo *manualmente* prima di eseguire il salto, modificando SP, liberando spazio nello stack e indirizzandolo manualmente in memoria

e.g:

```
ADDI $sp $sp-4 //libero lo spazio nello stack, puntando alla pagina libera  
SW $ra, 0($sp) //scrivo il valore del return address in stack appena liberato  
JAL sub 2 //salto alla subroutine 2, annidata in quella attuale sub1
```

nota: soltanto JAL è l'istruzione di salto al sottoprogramma, pertanto nelle sub intermedie è necessario che venga scritta l'istruzione duale a quella precedente, con la modifica della pila in lettura del valore successivo, affinché venga scritto in RA e con la quale si possa tornare al main

```
LW $ra, 0($sp) //comando tipico in cui carico il valore del ra nello sp
```

Non posso indirizzare allo SP con l'offset dell'istruzione: a livello di efficacia cambierebbe poco, ma avrei il grosso problema di non salvare il valore dello SP che mi serve per il ritorno

Nei MIPS inoltre, come già detto, non esiste il flag di *overflow/riporto* pertanto lavorando con le somme e le rappresentazioni numeriche devo controllare manualmente che la rappresentazione del numero ottenuto sia corretta e coerente

Idem con la moltiplicazione (e.g.  $2^{16} \cdot 2^{16} = 2^{32}$ , che con 32 bit di rappresentazione è (1)00.....000 di cui leggo soltanto gli 0. Per avere il # corretto dovrei avere 33 bit)

**Reg0:** nel MIPS uno dei registri, il \$Reg0 appunto, è di sola lettura e contiene il valore 0 in quanto capita spesso di dover confronti o portare a zero qualche variabile, e averlo in un registro interno porta ad un notevole miglioramento prestazionale rispetto alla memoria o l'averlo come variabile in istruzione dopo il c. operativo (invece di avere op.code+operando ho soltanto codice operativo, con l'istruzione che non deve uscire dalla memoria)

La fase di fetch rimane la stessa studiata a fondamenti, ovvero:

```
> (PC) → MAR  
> ((MAR)) → MBR; (PC) + 4 → (PC)  
> (MBR) → IR
```

Il *decodificatore* è un elemento che in ingresso riceve n linee e, dipendentemente dal numero d'ordine attivato viene riconosciuta l'istruzione da eseguire

Proprio le istruzioni e la loro lunghezza è stata l'ottimizzazione maggiore su cui hanno puntato gli sviluppatori del MIPS: esse infatti sono limitate, così come le modalità di indirizzamento (*non posso, a meno di costanti ridotte, avere dopo il codice operativo indirizzi o costanti*, essendo l'intera istruzione al massimo formata da 32 bit)

add A,1000 non può esistere, devo utilizzare necessariamente i registri (e qualora l'elemento sia in memoria sarebbero necessarie due fasi: una di prelievo e una per l'esecuzione dell'istruzione)

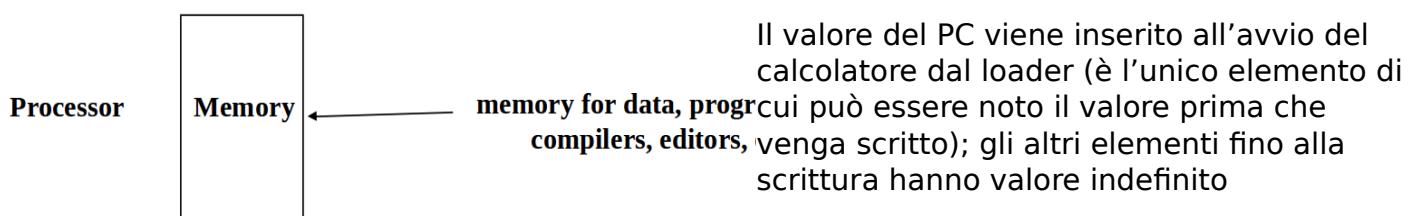
## ***Salti:***

essendo l'istruzione limitata a 32 bit totali, come faccio ad eseguire un salto in memoria indicando sia il codice operativo che il # d'ordine della cella a cui saltare?

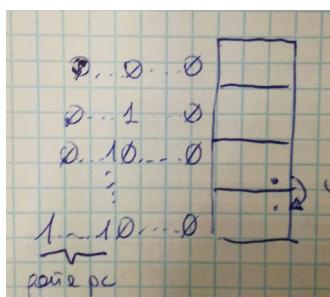
Non avendo i MIPS spazio sufficiente per inserire l'indirizzo (32 di istruzione totale ma anche 32 di indirizzo d'arrivo) è necessario utilizzare un metodo che aggiri il problema:

*principio di località degli accessi:* sfrutta il principio per cui se accedo ad una determinata area di memoria, quasi sicuramente nei momenti successivi avrò bisogno di un dato nell'intorno di quella particolare area. Così facendo posso, con 6 bit di op.code e 26 restanti ovviare al problema → nell'indirizzo le cifre più significative rimangono invariate (vengono ereditate dal pc precedente)

obs: è la stessa logica con cui, nella paginazione della memoria, vengono gestite le pagine e gli spostamenti al loro interno: l'indirizzo delle pagine viene tradotto da fisica a virtuale, l'offset all'interno della pagina rimane invariato



?=come faccio se devo spostarmi tra un macrogruppo di celle e quello adiacente, benché l'offset sia minimo (ovvero a modificare il macrogruppo definito dal PC)



Utilizzo un'istruzione specifica, ovvero:

**jr \$** in cui quello che compare in un registro è l'effettivo punto di memoria a cui devo saltare

OBS: al togliere dell'alimentazione la ROM rimane esattamente com'è mentre la RAM no; pertanto il simulatore porta a 0 i valori di ROM ma nella realtà essi contengono all'accensione valori non determinabili (a differenza del PC); non essendoci una posizione predefinita in memoria nemmeno lo SP è indicizzato ad un valore definito

## **MIPS**

Il MIPS e il linguaggio assemblativo in generale sono molto più restrittivi di qualunque linguaggio HL (e.g. nella somma sono obbligato ad utilizzare 3 registri, non posso utilizzare direttamente un addendo o un indirizzo di memoria). Questo con l'obiettivo di massimizzare le performance e minimizzare i costi. La semplicità infatti favorisce la regolarità, con l'idea che più piccola è la circuiteria (devo avvicinarmi o utilizzare solo il minimo indispensabile) più veloci sono le esecuzioni dei programmi (*i.e. se invece di 32 registri ne avessi 17 non cambiarebbe nulla, perchè i bit di indirizzamento sarebbero comunque 5; similmente per averne 33 tanto vale averne 64, con un aumento circuitale tuttavia notevole e una penalizzazione per le istruzioni più utilizzate*)

- somma:            **add \$0, \$1, \$2**

dove \$0 è l'indicatore del registro di arrivo, \$1 ed \$2 sono i registri da cui prelevare gli addendi

N.B. è compito del programmatore controllare che la rappresentazione della somma sia coerente ed effettivamente corretta, *non ci sono flag*

- somma iterata:    **add \$t0, \$s1, \$s2**  
                          **add \$t1, \$s3, \$s4**  
                          **add \$s0, \$t0, \$t1**

che dal p.to di vista esecutivo corrisponde a fare  $A = B + C + D + E$  calcolando la somma di  $B + C$  e  $D + E$  e poi sommando tra loro i risultati

*obs:* i registri \$tn sono i registri temporanei di immagazzinamento dati

avrei potuto fare anche la somma iterata sempre sullo stesso indirizzo destinazione, con la differenza che non avrei mantenuto il dato in tale registro, sovrascrivendolo, per eventuali necessità successive

- differenza:        **sub \$0, \$1, \$2**

con la stessa logica di indirizzi della somma, ovvero il primo è l'indirizzo destinazione e i due successivi sono gli indirizzi di prelevamento del dato

N.B. → gli operandi devono essere registri, a causa del limite a 32 bit delle istruzioni; qualora tuttavia volessi gestire un # di variabili maggiore di quello immagazzinabile nei registri (che di fatto sono limitati, più qualcuno dedicato non scrivibile) allora potrei comunque ricorrere alla memoria, con un registro che invece che contenere il dato conterrà l'indirizzo di memoria a cui prelevare il dato

### Organizzazione della memoria:

La memoria deve essere immaginata come un array monodimensionale diviso in macroelementi

155542	155543	155544	155545	15554	....	....	....
--------	--------	--------	--------	-------	------	------	------

Nelle memorie non si indirizzano mai i singoli bit, sarebbe controproducente dal punto di vista prestazionale e estremamente poco pratico. Per questo si utilizza l'indicizzazione degli elementi della memoria di 8 bit in 8 bit → sono indicizzati i byte

Perchè 8 bit e non, ad esempio, 32? perché 32 sono utili soltanto in determinati contesti in cui è richiesta una precisione estremamente elevata, ma tendenzialmente nella maggior parte dei casi 32 bit risultano sovabbondanti e 8 sono più che sufficienti (e muoversi all'interno degli elementi è difficile, pertanto meglio tenere configurazioni più ridotte per semplicità di utilizzo) → ad esempio il codice ASCII o i gradi di forni non industriali necessitano di precisione ridotta e sono perfettamente sufficienti 8 bit di utilizzo

### Come gestisco informazioni più grandi dimensionalmente?

(ovvero istruzioni, indirizzi etc, che ad esempio sono formati da 32 bit, ovvero 4 bytes?)

I dati uniformi occupano caselle addiacenti (devono rispettare il **vincolo di allineamento**)

Pertanto non posso prendere una word con indirizzi diversi dai multipli di 4; non posso avere, inoltre, una situazione in cui un elemento inizia in una word e finisce nell'altra (**e.g.** un indirizzo occupa un'intera word, non può iniziare alla word x e continuare nella x+1)

(Non posso avere un' istruzione che si trova a cavallo di due words; essa violerebbe il vincolo di allineamento)

→ ogni word è 4 byte di dati

Con una gestione del genere ottengo:

>  $2^{32}$  bytes con indirizzi di byte da 0 a  $(2^{32}) - 1$

>  $2^{30}$  words con indirizzi di byte da 0 a  $(2^{32}) - 4$  con step di 4 in 4 (0...4...8...12...)

Qualora volessi leggere un byte dovrei avere un indirizzo che può essere un # qualunque, qualora invece volessi leggere una pagina *devo avere un indirizzo che è multiplo di 4*

Pertanto lw e sw devono produrre, come risultato dell'indirizzamento con i registri, un multiplo di 4 affinché si possa leggere una word

e.g. → lw \$t0, 0(\$t1) \$t1 deve contenere un indirizzo che deve essere multiplo di 4 (**e così l'offset del puntatore**)

L'assemblatore può accorgersi dell'errore? → NO, infatti non è un errore sintattico in quanto l'istruzione apparentemente è scritta correttamente (e l'assemblatore non sa cosa c'è nell'indirizzo puntato da \$1). E' compito del programmatore valutare la corretta logica dell'istruzione

Esso segnalerebbe invece l'errata indicazione dell'offset, in quanto risulterebbe proprio errata l'istruzione (da p.to di vista lessicale della stessa)

Cosa succede quando ho un'impostazione come quella sotto, dove ci sono due vettori e un elemento di linguaggio? Se il messaggio ha un # di elementi multiplo di 4 il vettore parte allineato all'inizio della Word, qualora non fosse così l'assemblatore in esecuzione lascerebbe delle caselle di byte libere affinché ogni blocco inizi allineato all'inizio della Word

Vettore
messaggio
vettore

*Obs:* le ultime due cifre di ogni word sono, necessariamente 00; essendo infatti multipli di 4 gli indirizzi è necessario che siano multipli di 100, pertanto le ultime due rimarranno sempre a 00  
Nel caso delle Halfwords invece sono multipli di 2, pertanto l'unico vincolo è sull'ultima cifra a 0

Halfwords: sono elementi che, come dice il nome, occupano 2 byte invece di 4, e sono utilizzate per muovere con comodità elementi di 16 bit, come possono essere le codifiche UNICODE:

In questo caso i registri devono puntare a elementi che indirizzano a celle con indirizzo multiplo di 2 e, necessariamente, l'offset delle istruzioni deve seguire la stessa logica  
lh \$t0, 2(\$t1)  
sh \$t0, ''

l'equivalente per il singolo byte è:

lb \$t0, n(\$t1) dove n può essere qualunque numero, essendo un indirizzo di byte e non di word o halfword

N.B è compito del programmatore fare in modo che non vi sia interferenza tra word, byte e halfword (e.g. se ho un vettore in cui ho allocato word a cui sovrascrivo una hw)

il fatto che i bit finali di una word siano determinati viene sfruttato per aumentare le possibilità del JUMP:

dovendo infatti un'istruzione occupare necessariamente un'intera word ne consegue che i bit meno significativi dell'indirizzo saranno 00. Pertanto essi vengono *sottintesi* estendendo la capacità di rappresentazione di un indirizzo da 26 a 28 → (in fase esecutiva l'indirizzo non occupa le postazioni 0-25 ma le postazioni 2-27), con gli ultimi bit a 0

### Modalità di rappresentazione:

ipotizzando ddi voler rappresentare un # del tipo:

| 12345678 |  
H            L

Che tipologie di memorizzazione posso utilizzare?

> *Big.end* → è la memorizzazione per cui all'inizio della word (byte indirizzato 00) c'è la cifra di peso maggiore (nel mio caso 12) [usata dal mips]

> *little.end* → è la memorizzazione inversa per cui all'indirizzo di byte 00 trovo le cifre meno significative (nel mio esempio 78).

*obs:* non è così fondamentale la tecnica di memorizzazione, l'unica cosa importante è rimanere coerenti all'interno di un programma o processore per evitare confusioni

### LETTURA:

In scrittura non ci sono problemi, in quanto sto scrivendo elementi al più grandi come la cella, che pertanto non daranno problemi di interpretazione o variazione di valore

Ma in lettura?

			DATO
--	--	--	------

Il dato in questo caso, nella word in memoria, occupa soltanto uno dei 4 byte che formano l'intera word; questo può generare errori in lettura se non dovesse essere corretto

Se infatti non modificassi minimamente il dato da leggere e prendessi per buono ciò che trovavo prima della lettura avrei una lettura completamente errata del dato, pertanto ho due possibilità (dipendentemente dal contesto):

→ *AND*: posso forzare tramite un AND a 0 tutti i bit che precedono il byte che mi interessa leggere (o la HalfWord); utilizzato per valori assoluti, codifiche alfanumeriche e complemento a 2 quando il numero è positivo

→ *ESTENSIONE DEL SEGNO*: utilizzato per il complemento a due per evitare errori in lettura; se infatti avessi ad esempio 0000 0011 e portassi a 1 ciò che precede avrei lettura errata da positivo a negativo; similmente se passassi da 1110 0010 portando a zero gli antecedenti porterei un # negativo ad essere positivo. L'*estensione del segno* è una tecnica che consiste nel portare i bit che precedono il byte di lettura alla stessa cifra della più significativa del suddetto, sia essa 0 o 1. Così facendo il segno viene mantenuto e la lettura effettuata è corretta.

Per usare una delle due modalità deve essere il programmatore a capire la rappresentazione utilizzata, e a ciascuna modalità corrisponde un comando:

```

lb $t0, 0($t1) carica ciò che trova in memoria forzando gli 0 precedenti al
byte che voglio leggere
lbu $t0, 0($t1) 'load byte unsigned' carica ciò che trova eseguendo
l'estensione del segno
lh $t0, 0($t1) carica ciò che trova nella halfword forzando gli zeri nel
mezzo byte più significativo vuoto
lhu $t0, 0($t1) carica ciò che trova nella halfword eseguendo l'estensione
del segno (per c2 al fine ddi evitare errori)

```

*obs:* \* non è necessaria nessuna specifica per i comandi lw poiché le word sono già della dimensione dei registri, pertanto non è necessario specificare nessuna modifica sul dato (non è infatti possibile avere alcuna interpretazione errata)

\* nello store, similmente, non è necessario specificare alcuna opzione poiché sto memorizzando un elemento che è al più grande come una word all'interno di una word, pertanto qualora non fosse abbastanza grande il dato da riempire la word in memoria avrei semplicemente le 3w rimanenti che rimangono com'erano, con l'aggiustamento correttivo che avviene in lettura

Spilling: tecnica per cui le variabili più utilizzate vengono tenute all'interno dei registri, mentre quelle usate meno di frequente vengono salvate in memoria ( l'accesso ai registri è infatti molto più immediato di quello in memoria)

esempio di scambio di due valori di un vettore in C e in assemblativo:

```
swap(int v[], int k);
{ int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Che in assemblativo diventa:

```
muli $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

Dove le prime due istruzioni servono a recarsi nella k-esima posizione all'interno del vettore; la moltiplicazione per 4 viene fatta poiché lo spostamento all'interno del vettore è di word in word (ogni elemento occupa 4 byte, ovvero una word);

Per ottenere lo stesso risultato avrei avuto anche altre possibilità:

- **sll \$s2, \$s5, 2**

Dove viene utilizzato il comando sll (ovvero Shift Logical Left) di shift logico, che per eseguire moltiplicazioni per multipli di 2 non fa altro che traslare di due posizioni gli elementi che compone il numero, ottenendo la moltiplicazione (come aggiungere/togliere 0 in moltiplicazioni e divisioni per 10 in decimale) → è il più efficiente dei tre metodi  
Il range dello spostamento, indicato con il due nel caso, è 1-31 ed è il numero di *posizioni* di cui mi sposto, ovvero lo spostamento di bit nella parola, non word)

- add \$s2, \$s5, \$s5 (k+k in \$s2)
add \$s2, \$s2, \$s2 (2k+2k = 4k)

Che è il meno efficace dei 3, prevede di sommare prima un registro a se stesso e successivamente ripetere l'operazione, per ottenere il displacement desiderato

#### TIPI DI RAPPRESENTAZIONI:

> formato **R**

R	op	rs	rt	rd	shamt	funct
---	----	----	----	----	-------	-------

E' il formato utilizzato nella forma delle istruzioni, dove op è il codice operativo, rs il registro sorgente, rt e rd i registri coinvolti, shamt è lo "shift amount", ovvero l'eventuale shift richiesta nell'istruzione quando si indica il registro, function è una sequenza di varianti che rappresentano le possibilità di un'istruzione (e.g. add e sub hanno lo stesso codice operativo, sono differenziate soltanto dalla variazione di funct)

> formato I

I	op	rs	rt	16 bit address
---	----	----	----	----------------

è il formato di quando ho un'istruzione che non contiene 3 registri ma, ad esempio, 2 + displacement. In questo caso i soli registri dedicati allo shift non basterebbero ad indirizzare in modo sufficiente lo spostamento e avrei uno spreco di bit dovuto al non utilizzo del terzo registro; in questo caso pertanto sfrutto i bit inutilizzati del terzo registro per l'indirizzamento; In questo caso i bit indicano lo spostamento da sommare al Pc per ottenere il nuovo indirizzo La ALU come decide cosa significa la sequenza di bit che gli arriva in quale configurazione si trova? → *dipende dal # del codice operativo, con un compromesso dal punto di vista della complessità organizzativa e gestionale del tutto*

NB: la CPU, a meno di toglierle l'alimentazione, fa sempre qualche azione e non può "non fare niente"; una strada per terminare un'esecuzione senza farle più compiere azioni potrebbe essere il richiamo ricorsivo di un registro → ciclo: j, ciclo

#### TIPI DI SALTO:

Esistono 2 tipologie di salto:

> **beq** → Branch if Equal

> **bne** → Branch if Not Equal

**bne \$t0, \$t1, Label**

**beq \$t0, \$t1, Label**

Dove *Label* indica il punto dove saltare, con la rappresentazione I che indica, tramite i 16 bit, dove saltare a partire dal PC; ? = quanto vale *Label* → Essendo che deve puntare ad un'istruzione e, pertanto, ad un inizio di word i 16 bit vengono utilizzati per indicare il displacement e non l'indirizzo, con salto di word in word (un displacement di 1 indica uno spostamento avanti di 4 byte, ovvero di un'intera istruzione)

bne \$s0, \$s1, Label

add \$s3, \$s0, \$s1

Label... ... ...

Lo spostamento dalla bne al Label è di 1 (un'istruzione) poiché devo tener conto dell'autoincremento del PC

> formato J

J	op	26 bit address
---	----	----------------

Quando devo gestire un salto ho soltanto l'istruzione di salto e l'indirizzo a cui saltare, pertanto per non sprecare spazio e aumentare la capacità di indirizzamento utilizzo tutti i 26 bit al di fuori dell'op-code per indirizzare la destinazione del salto; anche in questo caso doovendo saltare ad un'istruzione, i bit indicheranno word e formeranno multipli di 4; per questo motivo si sottintendono i 2 meno significativi (0 per essere 4X) e invece che sommare il risultato al PC i 26 bit formano il nuovo PC andando a sostituire i bit dal 2 al 27  
4 bit ereditati dal PC      26 di istruzione J      2 sottintesi a 0 per vincolo ordinamento

Così facendo la capacità di rappresentazione varia da  $2^{26} - 1$  a  $-2^{26}$ ; per raggiungere gli indirizzi non interni al range di rappresentazione è necessario utilizzare *l'indirizzamento a registro*

## SALTO:

j label

indica il salto incondizionato e la destinazione del salto come label; tramite i salti incondizionati posso costruire cicli come il for o terminare istruzioni componenti di un if:

- MIPS unconditional branch instructions:

j label

- Example:

```

if (i!=j)          beq $s4, $s5, Lab1
    h=i+j;
else             j Lab2
    h=i-j;
Lab1: sub $s3, $s4, $s5
Lab2: ...
  
```

Qualore l'indirizzo di destinazione del jump sia troppo lontano è l'assemblatore a segnalare l'errore e reindirizzare il jump all'indirizzo corretto

*esempio di programma che utilizza un ciclo:*

L1: g = g+A[i];

i = i+j;

if (i != j) foto L1

che in codice assemblativo diventa:

```

L1: add $t1, $s3, $s3      # $t1 = 2 * i
    add $t1, $t1, $t1      # $t1 = 4 * i
    add $t1, $t1, $s5      # $t1 = indirizzo di A[i]
    lw $t0, 0($t1) # $t0 = A[i]
    add $s1, $s1, $t0      # g = g + A[i]
    add $s3, $s3, $s4      # i = i + j
    bne $s3, $s2, L1# vai a L1 se i „ h
  
```

è un programma che preso un vettore somma gli elementi distanti j-posizioni fino al raggiungimento della massima soglia fissata h; all'inizio del programma ho un registro puntatore che punta all'inizio del vettore e due istruzioni che moltiplicano per quattro attraverso una doppia somma l'indicatore i, infatti -i è l'indicatore della posizione all'interno del vettore, ma ogni posizione è composta da 4 byte.

Nell'istruzione di add ad \$s5 ho lo spostamento dall'inizio del vettore alla posizione \$s5 + 4\*i  
*obs:* l'elaborazione di i potrei averla in 3 modi differenti: - moltiplicazione (non funzionale) - somme iterate - shift di due bit (= \*4)

**NB:** il registro che contiene i NON deve variare prima dell'incremento di j, infatti sia i che j sono indicatori della posizione all'interno del vettore e vanno moltiplicati dopo la somma, altrimenti avrei un incremento man mano che aumento j (j, 4j, 8j, 16j...); così facendo invece aggiorno man mano la posizione del vettore e, una volta aggiornata, la moltiplico \*4 per avere le word di spostamento

Blocchi base: sono blocchi che contengono un ciclo iniziato e finito, i primi ricercati dal compilatore per velocizzare l'esecuzione

*esempio 2 di programma con ciclo while:*

```

while (save [i] == k)
    i = i + j;
  
```

che in assemblativo diventa:

```

Loop: add $t1, $s3, $s3      # $t1 = 2 * i
      add $t1, $t1, $t1      # $t1 = 4 * i
      add $t1, $t1, $s6      # $t1 = indirizzo di save[i]
      lw $t0, 0($t1) # $t0 = save[i]
      bne $t0, $s5, Exit    # vai a Exit se save[i] „ k
      add $s3, $s3, $s4      # i = i + j
      j Loop                 # vai a Loop
  
```

Exit:

Anche in questo caso si nota come i puntatori all'interno dei vettori rimangano in formato

canonico per l'incremento e vengano moltiplicati soltanto prima dell'esecuzione del programma, in modo da avere incremento uniforme dal p.to di vista del passo

> **slt \$t0, \$s1, \$s2**

è l'istruzione di "Set if Lower Than", ovvero di confronto di due registri, nel caso \$s1 e \$s2, e di inserimento in \$t0 di un valore logico corrispondente al cfr, ovvero 1 se \$s1 < \$s2 0 se \$s1 > \$s2 (nel registro specifico vengono settati tutti i bit a 0 tranne il più significativo, dipendente dal risultato)

(In C sarebbe un implementazione del tipo if/else con condizioni di minoranza)

?= come faccio a controllare maggiore/minore su valori che non so in quale momento misurerò (e.g. temperatura forno) e in cui mi serve sapere esattamente quando diventano uguali? Rischierei di perdermi misura settando l'uguaglianza

Dovrei avere istruzioni di salto che dipendano dal cfr ma che coprano tutte le possibilità di confronto:

beq, bge, blt, bgt, ble, sge, slt, sgt, sle

Tuttavia all'interno del MIPS esiste soltanto slt come istruzione di confronto, pertanto per ovviare al problema sono state introdotte le **PSEUDOISTRUZIONI**:

Esse sono istruzioni richiamabili dal programmatore che, tuttavia, non hanno un vero riscontro nella tabella di corrispondenza, ma sono di fatto formate da più istruzioni in modo trasparente al programmatore (è l'assemblatore che al momento della chiamata di una pseudoistruzione andrà ad eseguire i vari steps che la compongono); ex:

**bge:** slt \$t0, \$s1, \$s2 # cfr tra registri e scrittura risultato logico  
beq \$t0, \$zero, Lab # cfr risultato logico con registro a 0 e salto a Lab

**blt:** slt \$t0, \$s1, \$s2  
bne \$t0, \$zero, Lab

Dal punto di vista del programmatore, in ambiente integrato, si avrà la differenza che nel caso di istruzioni reali implementate circuitalmente esse avranno la chiamata esadecimale corrispondente, nel caso di pseudoistruzioni (che di fatto non esistono dal punto di vista circuitale) si avrà l'elenco delle istruzioni che compongono le pseudo

NB: \$t0 se è in uso non deve essere sostituito, pertanto in realtà in pseudocodice non viene utilizzato dalla CPU, che usa invece il registro **\$at**, che può utilizzare soltanto lei e non il programmatore, in modo da evitare sovrascritture indesiderate ai registri usati

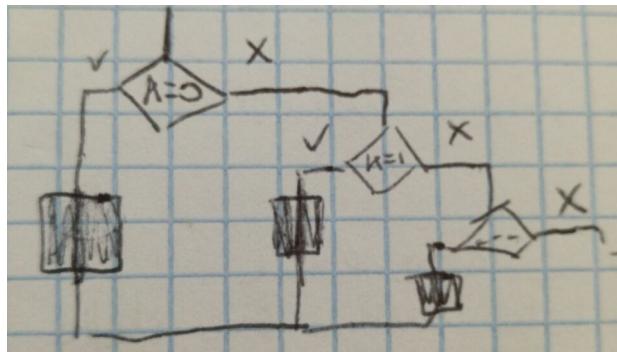
Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

← Reg. non preservati

→ Reg. preservati

## CASE- SWITCH:

In C è una struttura di controllo che ad una costante fa corrispondere l'esecuzione di un determinato comando di quelli possibili corrispondenti; come lo introduco in Assembly, senza utilizzare una struttura eccessivamente nidificata che non mi permette di avere accesso con lo stesso tempo al case1 o case1000 senza differenze?



Nel caso del case switch per avere tempi d'accesso uniformi a prescindere dal risultato del *case* è necessario trovare una routine che mi permetta di raggiungere ogni elemento di memoria con le istruzioni relative; ovvero una  $f(k)$  t.c. l'output sia  $(4*k) + \text{indirizzo tabella}$

Dove la tabella è una tab di corrispondenza tra le Ln label corrispondenti alle varie k e il loro indirizzo di memoria. Moltiplico per 4 il k poiché trattandosi di

istruzioni mi sposto di una word alla volta; così facendo ho tabella in cui ho memorizzato con l'ordine delle k possibilità l'indirizzo di ciascuna istruzione corrispondente. Quando ottengo una k dal controllo moltiplico k per 4 (per la word) e gli sommo il puntatore all'inizio della tabella; così facendo avrò che lo spostamento alla k-esima condizione sarà esattamente il displacement dalla cima della tabella, moltiplicando \*4 k e ottenendo tempi di raggiungimento unici a prescindere dal caso che si è verificato

(NB: se non avessi corrispondenza di sequenza tra i k e i risultati potrei semplicemente rendere i risultati non contemplati = exit, per far saltare direttamente il programma all'uscita del controllo qualora non sia stata realizzata una condizione verificabile)

L0
L1
L2
...
...
Ln

? = *come leggo la tabella*: con il `lw $t1($t0)`; Posso scrivere nel ProgramCounter → NO, non è uno dei 32 registri; per scrivere nel Pc è necessario utilizzare il jump

```

Switch (k) {
    case 0: f=i+j; break;
    case 1: f=g+h; break;
    case 2: f=g-h; break;
    case 3: f=i-j; break;
}
add $t1, $t1, $t1 # $t1=4*k
add $t1, $t1, $t4
lw $t0, 0($t1)
jr $t0 #vai a indir. letto
L0: add $s0, $s3, $s4 #k=0, f=i+j
j Exit
L1: add $s0, $s1, $s2 #k=1, f=g+h
j Exit
L2: sub $s0, $s1, $s2 #k=2, f=g-h
j Exit
L3: sub $s0, $s3, $s4 #k=3, f=i-j
Exit:

```

$f = \$s0, g = \$s1, h = \$s2, i = \$s3, j = \$s4, k = \$s5; \$t2 = 4; \$t4 = \text{indirizzo tabella etichette}$

`jr Label` con la quale salterei direttamente all'indirizzo di interesse, scrivendolo nel PC  
*NB: il valore nella tabella è il valore dell'indirizzo dell'istruzione, NON l'istruzione vera e propria (possono non seguire ordine logico i dati all'interno della stessa)*

Cosa cambierebbe con 1000 case invece che 5? Dal punto di vista della corrispondenza della tabella nulla, l'unica differenza sarebbe la maggiore occupazione di memoria della stessa; Esempio di case

switch scritto in linguaggio assemblativo; le prime 4 righe servono a fare controlli su K per andare direttamente all'exit qualora esso sia < 0 o >3; le righe successive invece servono a moltiplicare 4 per k per ottenere un displacement coerente con lo spostamento di w in w

### SOTTOPROGRAMMI:

Il comando permette di eseguire il salto è j1 ma è necessario, prima di eseguire il salto, salvare all'interno dei registri il contenuto di ciò che stavo facendo prima, per evitare di perdere dati.

I registri hanno funzione ben definita dipendentemente dal contesto e dalla tipologia a cui sono dedicati; ogni registro infatti svolge una funzione specifica al fine di semplificare la programmazione ed evitare sovrascritture:

Ad esempio se nel sottoprogramma utilizzo un registro devo assicurarmi che questo non venga utilizzato prima nel MAIN o che non avvengano sovrascritture che possano modificarne il contenuto senza controllo; per essere sicuro di ciò è necessario che prima dell'esecuzione del sottoprogramma io copi *in memoria, nello stack* i dati di tali registri e che, al termine della subroutine, venga eseguita la procedura inversa di prelevamento dallo stack e riscrittura dei registri

Dove si nota che, con le 3 sw/lw vengono copiati in memoria i valori dei registri (dopo aver lasciato spazio nello stack incrementandolo di 4\*n dove n è il # di registri da copiare) \*\*

```
int proc (int g, int h, int i, int j)
{   int f;
    f=(g+h)-(i+j);      proc: addi $sp, $sp, -12 # 3 push
    return f;           sw $t1, 8($sp)
}                     sw $t0, 4($sp)
g, h, i, j = $a0...$a3  sw $s0, 0($sp)
f = $s0              add $t0, $a0, $a1 # calc. f
                      add $t1, $a2, $a3
                      sub $s0, $t0, $t1
                      add $v0, $s0, $zero # $v0=f
lw $s0, 0($sp) # 3 pop
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12
jr $ra # ritorno
```

**Per convenzione:**  
**\$t0-\$t9 temporanei da non salvare**  
**\$s0-\$s7 da conservare**  
**si potevano risparmiare 2 push/pop**

\$t1
\$t0
\$s1
.....

*obs:* addi serve a sommare al registro indicato una costante in C2, ed è usata per liberare lo stack

Dove viene utilizzato l'add su due registri differenti a \$zero per portare a \$S0 il valore di \$V0 avrei bisogno di un'istruzione più

immediata, tipo *move*; essa NON esiste come istruzione effettiva del processore ma può essere utilizzata → rientra nelle *pseudoistruzioni* utilizzate dal MIPS (ovvero istruzioni che non hanno un effettivo corrispettivo circuitale e una corrispondenza 1:1 istruzione/esecuzione ma che vengono tradotte dall'assemblatore nelle sottoistruzioni che le compongono; nel caso del move nella somma a \$zero di un registro)

E se in un sottoprogramma avessi avuto bisogno di un altro programma annidato? Avrei dovuto salvare anche il *main* in memoria poiché altrimenti esso sarebbe andato perso e non sarei più uscito dall'annidazione; per questo esiste una **convenzione sui registri**, con alcuni di essi che sono preservati e altri che invece non lo sono (e che quindi non richiedono di essere salvati in chiamata a sottoprocedura)

- > **\$t0 - \$t9** sono registri che NON vengono preservati in caso di chiamata a procedura
- > **\$s0 - \$s7** sono registri che vengono preservati in chiamata e che pertanto, se utilizzati, devono essere salvati in memoria e rispristinati al termine del chiamato
- > **\$a0 - \$a3** registri non preservati che fungono da input per gli argomenti delle funzioni (portano dati al sottoprogramma, e sono pertanto modificabili)
- > **\$v0 - \$v1** sono registri per il salvataggio del risultato, per comunicarlo dal sottoprogramma al main
- > **\$k0 - \$k1** sono registri non utilizzabili, vincolati all'assemblatore
- > **reg 1** registro non utilizzabile vincolato all'assemblatore per la gestione delle pseudoistruzioni (programmatore non può metterci mano)

\*\* in questo caso con la convenzione dei registri si sarebbero evitate due chiamate *push/pop* in quanto i due registri temporanei non avrebbero richiesto copiatura in memoria, a differenza del registro \$s0

Inoltre si rende superflua la presenza di \$s0 in quanto essendo modificabile potrei utilizzare direttamente \$v0 per lo storing del dato del sottoprogramma

### esempio programma con sottochiamate:

```
i = 0;  
while ((x[i] = y[i]) != 0) /* copia e test byte */  
i = i + 1;  
}  
strcpy: addi    $sp, $sp, -4  
        sw      $s0, 0($sp)          # salva $s0 nello stack  
        add    $s0, $zero, $zero      # i = 0  
L1: add   $t1, $a1, $s0          # ind. y[i] in $t1  
    lb     $t2, 0($t1)          # $t2 = y[i]  
    add   $t3, $a0, $s0          # ind. x[i] in $t3  
    sb     $t2, 0($t3)          # x[i] = y[i]  
    addi  $s0, $s0, 1           # i = i + 1  
    bne   $t2, $zero, L1         # se y[i] ≠ 0 vaia L1  
    lw     $s0, 0($sp)          # ripristina $s0 dallo s  
    addi  $sp, $sp, 4
```

#### *obs:*

- *\$zero = \$0* infatti chiamare il registro con il proprio nome o il proprio # d'ordine non fa alcuna differenza

- *errore*: c'è un errore concettuale per il quale nell'istruzione *lb* nonostante si leggano stringhe ASCII non si utilizza la LBU ma la LB (con rischio di errori in lettura o eliminazione bit parità per prolungamento segno)

- *non fa differenza*: in questo caso perché dopo la load byte è presente la store byte, che considerando soltanto i primi 8 bit *s'elimina comunque i possibili errori derivanti*

?= se nel sottoprogramma volessi utilizzare i registri *\$Sn* → devo salvarli all'ingresso nel sottoprogramma, affinché agli occhi del main rimangano invariati

?= se nel main volessi utilizzare *\$tn* → dovrei, quando entro in routine, salvarli come i valori *\$s*. Infatti a differenza dei registri *\$s* essi sono modificabili dal sottoprogramma e pertanto devo prima salvarli in memoria.

?= come faccio se ho informazioni da inserire nei registri *\$sa* che sono più del # dei registri o risultati nei registri *\$v* che sono più dei registri a disposizione? → NON posso utilizzare registri *\$s0* poiché essi sono in uso nel main (a meno di salvarli prima dell'utilizzao, complicando il programma)

E' necessario andare in memoria e utilizzare i registri come indirizzamenti;

### COSTANTI:

Come lavoro con le costanti, dato che capita spesso di doverle utilizzare come indici, come shift etc etc?

Ho più soluzioni:

> salvare le *costanti frequenti* in memoria e caricarle, ma avrei una perdita di velocità considerevole rispetto ai registri; tuttavia essi sarebbero liberi e non li occuperei con elementi di cui non ho sempre bisogno, essendo i registri limitati.

> posso utilizzare i registri per memorizzare le costanti in modo hard-wired, come avviene per *\$zero*; è tuttavia una strada non perseguitibile oltre un # limitato di registri in quanto essi sono limitati e dedicandone molti alle costanti andrei a perderne utilità e occuparne alcuni inutilmente

La soluzione più efficiente è quella di introdurre una convenzione per la quale io possa scrivere la costante direttamente all'interno dell'istruzione:

Essa viene introdotta con le istruzioni istruz -i

```
addi $29, $29, 4 // ex importante, è l'incremento dello SP  
slti $8, $18, 10 // utile nell'esempio del forno per settare limiti critici  
andi $29 $29, 6 // sono operatori che vengono utilizzati per la mascherature  
ori $29, $29, 4 // in cui per ovvi motivi NON estendo il segno (introdurrei
```

errori) (\*)

La cui sintassi sta per "cmd Immediate"; sacrifico il terzo registro per avere i 16 bit meno significativi dell'istruzione a disposizione della costante stessa. La variabilità, quindi, varia da  $2^{15}-1$  a  $-2^{15}$ ;

e.g. *mascheratura* (\*)

0110 0001                    nel caso specifico si passa dalla *a* alla *A* in ASCII  
                              → and

1101 1111

0100 0001

**?=** che valore assumono i bit più significativi nelle operazioni logiche? → NON posso estendere il segno nel caso di o.logiche, posso invece nel caso di somme/sottrazioni

**?=** ha senso che esista un'istruzione del tipo `subi $s0, $s0, 4`→ NO, al limite come pseudoistruzione. Infatti la differenza con costante, essendo questa in C2 e permettendo anche la rappresentazione dei negativi, implica che sia sufficiente fare una addi con -4 invece che una sottrazione con +4

?= come faccio a caricare in registro una costante di 32 bit, contando che nell'istruzione *al massimo* con la immediate posso storarne 16?

`lui $t0, 0x1234;` è il comando con cui Load Upper Immediate, carico la costante alle posizioni più pesanti dei 32 bit presenti nel registro. A questa istruzione faccio seguire una ora e ottengo esattamente il caricamento dei 32 bit all'interno del registro

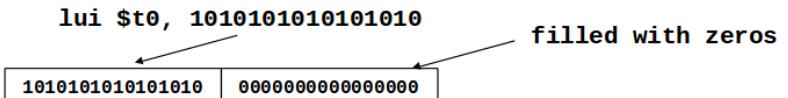
*obs:* NON posso fare addi \$t0, \$t0, 0x5678 poiché la cifra più elevata in peso potrebbe causare degli errori in caso di estensione del segno, con perdita di informazione anche nei bit più significativi

obs: **lui** è, a tutti gli effetti, un'espressione reale esistente; tuttavia sarebbe stata sostituibile da pseudocodice del tipo:

```
ori $t0, $zero, 0x1234  
sll $t0, $t0, 16
```

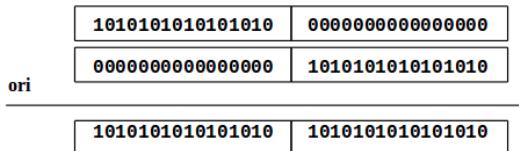
- We'd like to be able to load a 32 bit constant into a register
  - Must use two instructions, new "load upper immediate" instruction

Esempio di funzionamento  
della *lui*



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Esiste, per velocizzare la scrittura del codice e semplificarne la leggibilità, l'istruzione  
li \$t0, 0x12345678

Essa è una pseudoistruzione che viene eseguita dall'assemblatore come le due precedenti, *lui + ori*

*caso particolare:*

li \$t0, 65596 → in questo caso il numero da caricare in registro è esattamente  $2^{16}$ , ovvero 1 solo sul bit si posizione 17 pertanto l'assemblatore in questo caso non esegue le operazioni di prima ma semplicemente eseguirà un

```
lui $t0, 1
```

nota: quando sono sufficienti i bits più bassi la *l* viene interpretata semplicemente con un *ori* in \$t0, in quanto sono sufficienti quei bits

?= come creo la tabella nel *case switch* → con l'istruzione *la*

```
la $t0, label
```

con questa istruzione viene caricato l'indirizzo di Label direttamente in \$t0; così facendo posso creare la tabella, concatenando le istruzioni;

```
la $t0, I0  
sw $t0, ($t4)  
la $t0, I1  
sw $t0, 4($t4)
```

...

...

*li* e *la* caricano costanti e indirizzi in modo immediato (sono entrambe pseudoistruzioni formate da più istruzioni reali)

Le *pseudoistruzioni* semplificano l'utilizzo del processore: essendo un RISC non posso implementare un # eccessivo di istruzioni, ma posso avere istruzioni non implementate (che il compilatore provvederà a trasformare nella lista di istruzioni che effettivamente formano la pseudo)

e.g.

```
move $t0, $t1 → %fa la copia di $t1 in $t0, ma non esiste; in realtà è:  
          % add %t0, %t1, $zero
```

Un altro esempio è *bgt* o *blt*, che non esistono fisicamente e esistono soltanto come composizione delle uniche reali presenti: **slt**, **beq**, **bne**

? esiste un'istruzione del tipo *sw \$t0, 0x10010345* → NO, in quanto non posso indicare un \$t1 di 32 bit in quanto non bastano i bit dell'istruzione non bastano.

Potrebbe esistere come pseudoistruzione descritta da:

- *lui \$at, 0x1001*
- *ori \$at, \$at 0x0345*
- *sw \$t0, 0(\$at)*

NOTA: la *ori* è superflua infatti potrei utilizzare soltanto la parte alta dell'indirizzo e utilizzare la parte bassa come shift nella store word

```
sw $t0, 0344($at)
```

**NB:** la pseudoistruzione effettivamente esiste, tuttavia il numero che viene inserito come shift o all'interno della ORI DEVE essere un multipli di 4 per rispettare il VINCOLO DI ALLINEAMENTO DELLA MEMORIA



Il problema dell'utilizzare l'offset in B16 è che avrebbe dato problemi di conversione se avessi avuto un numero che iniziava con 1, per problema della rappresentazione in C2  
*Obs:* i tempi esecutivi dipendono dalle istruzioni vere, non dalle pseudoistruzioni

### **SALTI**

> *RELATIVI*: con beq e bne  
> *PSEUDOASSOLUTI*: con j

Con i 16/26 bit ho una capacità rappresentativa ddi  $2^{17-1}/2^{17}$  e  $2^{27-1}/2^{27}$  poiché non ho i due bit meno significativi, in quanto per rispettare il vincolo della memoria posso sottintendere i due bit meno significativi, essendo sempre a 0

? = cosa accade se nell'istruzione? Come viene calcolata la posizione di *poi*  
bne \$t0, \$t1, poi

...  
...  
...  
...  
poi...

Per calcolare la posizione di *poi* viene innanzitutto incrementato il PC + 4 per l'istruzione successiva; poi viene calcolato dall'assemblatore quanti byte di spazio separano il j dal *poi*.

Divide per 4 e calcola il numero di istruzioni (che nelle istruzioni di J sono il numero cche s trova a dx dell'istruzione); in esecuzione rimoltiplica per 4 per calcolare l'effettivo shift in byte

Qualora fosse eccessivo lo shift da effettuare e uscisse dai range massimi/minimi sarei obbligato a invertire l'istruzione e accorciare i salti;

beq \$t0, \$t1, dopo

j poi  
...  
...  
dopo  
...  
...  
...  
...

poi

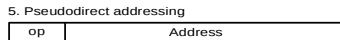
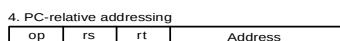
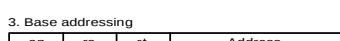
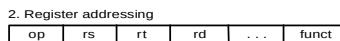
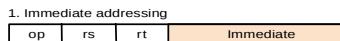
Dove inverto il senso dell'istruzione, saltando se non è verificata ma allungando il range di rappresentabilità e utilizzando l'istruzione j in catena

Una discreta capacità di accesso è garantita dal principio di località degli accessi; tuttavia può capitare di avere elementi all di fuori della rappresentabilità massima possibile  
Ho infatti al limite dei macroblocchi da 256MB, in quanto i 4 bit alti rimangono invariati; se tuttavia mi trovo oltre questo range di rappresentazione devo avere modo di spostarmi:

La modalità è:

la \$at, poi  
jr \$at

## **INDIRIZZAMENTI:**



> indirizzamento immediato:

nell'istruzione c'è il dato, utilizzata ad esempio dalle addi e ori

> a registro: nell'istruzione c'è il registro a cui si trova l'informazione, usata dalle sw/lw

> base addressing: nell'istruzione c'è il registro, il punto di partenza e spostamento da applicare al puntatore per avere locazione in memoria (contabilizza i byte in istruzioni, devo specificare puntatore all'indirizzo di partenza, NON modifica la base ma utilizza lo shift dall'istruzione)

> PC relative: come il precedente con la differenza che è relativo al displacement del PC, ragiona in istruzioni, da moltiplicare per 4 per avere in # di words (chiedere); non utilizza basd ma il PC come tale, e lo shifting è a word, pertanto deve essere moltiplicato per 4 prima di essere utilizzato

> Pseudodiretto: non posso specificare direttamente indirizzo per le limitazioni dovute all'architettura del MIPS, ma di fatto sto effettuando un salto direttamente alla locazione di memoria (di fatto 4 bit più significativi vengono ereditati dal PC mentre i due meno significativi sono 00 in quanto essendo word intere devono essere multipli di 4)

Per semplificare l'utilizzo del processore sono state implementate le pseudoistruzioni: sono istruzioni che circuitalmente non esistono ma che è possibile utilizzare, in quanto l'assemblatore le tradurrà nella sequenza di istruzioni reali che le compongono

- Versioni modificate delle istruzioni vere, trattate dall'assemblatore

- Esempi:

Pseudo istruzione: move \$t0, \$t1 # \$t0 = \$t1  
Istruzione vera: add \$t0, \$zero, \$t1

esempi di pseudoistruzioni e delle loro corrispettive istruzioni reali

Pseudo istruzione: blt \$s1, \$s2, Label  
Istruzioni vere: slt \$at, \$s1, \$s2  
bne \$at, \$zero, Label

NB: la durata effettiva dell'istruzione e l'occupazione in memoria della sequenza dipende dalle istruzioni reali che compongono la sequenza, NON dalle pseudo

- Altri esempi:

bgt, bge, ble; branch condizionati a locazioni distanti trasformati in un branch e una jump, li, etc.

## **UTILIZZO DI PUNTATORI:**

anche in linguaggio assemblativo, come in linguaggio di alto livello come può essere il C, vi sono ottimizzazioni di programmi tramite l'utilizzo di puntatori ad elementi rispetto all'utilizzo di elementi effettivi, in particolare per elementi sequenziali come vettori o char

Di seguito sono proposti due modi di eseguire lo stesso compito, ovvero l'azzeramento degli elementi di un vettore per l'inizializzazione, eseguiti in due modi differenti: a sx l'esecuzione più semplice, che non fa uso di puntatori alla memoria e di fatto utilizza la logica dell'elemento i-esimo. A dx l'esecuzione tramite puntatori alla memoria all'indirizzo di inizio dell'elemento per utilizzare poi la logica dello shift in memoria e velocizzare il tutto:

```

azz1 (int vett[], int dim)
{
    int i;
    for (i=0; i<dim; i++)
        vett[i] = 0;
}
azz1: move $t0, $zero      # i = 0
L1: add $t1, $t0, $t0      # 4 * i
    add $t1, $t1, $t1
    add $t2, $a0, $t1      # $t2 = indirizzo di vett[i]
    sw $zero, 0($t2)       # vett[i] = 0
    addi $t0, $t0, 1        # i = i + 1
    slt $t3, $t0, $a1      # i < dim ?
    bne $t3, $zero, L1     # se i < dim vai a L1
    jr $ra

```

Indirizzo vett = \$a0, dim = \$a1, i = \$t0

- azzero i
- moltiplico i\*4
- in t2 ho l'indice i-esimo elem.
- se i raggiunge "dim" vado a jr con salto a \$ra

```

azz2 (int *vett, int dim)
{
    int *p;                  // *p è l'oggetto puntato da p
                            // &vett è l'indirizzo di vett
    for (p=&vett[0]; p<&vett[dim]; p++)
        *p = 0;
}
azz2: move $t0, $a0          # p = indir vett[0]
      add $t1, $a1, $a1      # 4 * dim
      add $t1, $t1, $t1
      add $t2, $a0, $t1      # $t2 = indir di vett[dim]
L2:  sw $zero, 0($t0)        # mem puntata da p = 0
      addi $t0, $t0, 4        # p = p + 4
      slt $t3, $t0, $t2      # p < &vett[dim] ?
      bne $t3, $zero, L2     # se è vero vai a L2
      jr $ra

```

Indirizzo vett = \$a0, dim = \$a1, p = \$t0

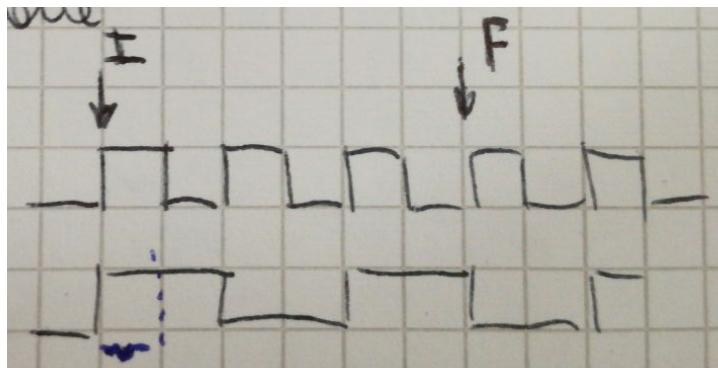
- vet = indirizzo di inizio
- in t0 porto ind. partenza
- moltiplico dim\*4
- in \$t2 trovo ind. Di fine vettore
- scrivo zero nell'indirizzo puntato da \$t0
- incremento puntatore di +4 e faccio cfr con indirizzo fine vettore

Nel primo NON avrei potuto fare direttamente i+4 poiché dim è la dimensione del vettore intesa come # di elementi, per fare un cfr coerente avrei dovuto moltiplicare anche essa per 4. Tra i due il più semplice e leggibile è il primo, mette dal punto di vista dell'efficienza è nettamente più rapido il secondo: esso infatti ha soltanto 4 istruzioni all'interno del ciclo al contrario del primo, che ne ha 7 (a fronte di più istruzioni preliminari, eseguite tuttavia una volta sola)

NB: il rapporto tuttavia NON è di 7/4 dal punto di vista della velocità di esecuzione del codice in quanto vi sono molti elementi che determinano l'effettiva velocità di esecuzione di un'istruzione

## CISC VS RISC

l'utilizzo di un processore RISC implicherebbe un numero di istruzioni molto maggiore, con conseguente aumento dei tempi esecutivi e dei CPI (Clock Per Instruction); infatti avendo un # molto maggiore di istruzioni implica un tempo maggiore anche a partire dalla decodifica delle stesse



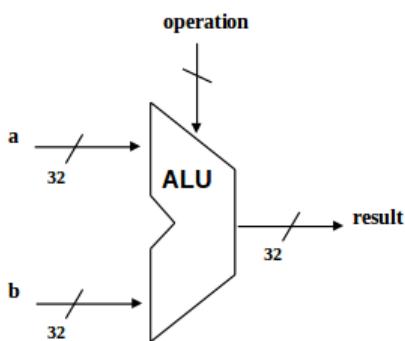
Il fatto di aumentare la lunghezza del periodo, diminuendo di fatto la frequenza è quella che sebbene un'istruzione possa così venire eseguita all'interno di un solo ciclo di clock, alcune componenti che rischieranno tempi minori si ritrovano a dover attendere prima di poter completare l'istruzione (e.g. se le parentesi blu indicano un'istruzione rapida essa sarà completata a metà del clock ma non potrà essere utilizzata fino alla fine dello stesso)

E' pertanto più conveniente concedere più periodi alla singola istruzione, come accade per le pseudoistruzioni, ad esempio, piuttosto che avere istruzioni con rischiuste molto lunghe di periodo e limitando fortemente le istruzioni più rapide (con il clock più veloce concedo il giusto tempo alle istruzioni rapide, e al limite più cicli alle istruzioni più lente)

## **PRINCIPI DI PROGETTO:**

- > semplicità favorisce regolarità (come nelle istruzioni matematiche e nel fatto di avere tutti gli elementi di dimensione fissata a 32 bit)
- > piccolo è più veloce (avere meno registri e meno istruzioni implica una maggior velocità nella ricerca all'interno degli stessi e nella decodifica)
- > buon progetto include buoni compromessi
- > rendere il caso più comune il più veloce possibile (e.g. addi, ori etc degli immediate)

## **IMPLEMENTAZIONI NUMERICHE:**



(nota: nell'IDE la visualizzazione decimale è SEMRPE in C2, pertanto è compito del programmatore vedere se essa sia corretta o se vada riadattata alla convenzione in uso nel registro a cui si fa riferimento; più semplice riferirsi alla B16)

I bit sono soltanto bit, non hanno significato intrinseco ma dipende dal contesto in cui si trovano (con una conseguente maggior complicatezza progettuale poiché limita il numero di rappresentazioni fattibili, rende la gestione di frazioni e virgole più articolata ... )

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

Per questo motivo ad esempio non esiste l'istruzione di `sub` poiché viene implementata dalla addi con un numero in C2 negativo come addendo

Posso lavorare in differenti modi con i numeri e le conversioni:

### **NUMERO SENZA SEGNO:**

La rappresentazione in v.a. del numero senza segno implica che con un byte io possa rappresentare  $2^8$  elementi, ovvero il range 0-255. Non è funzionale poiché arrivato a 255 + 1 torno a 000 con il numero di bit a disposizione

Inoltre non permette l'esecuzione di operazioni al di fuori dell'insieme N dei numeri naturali

### **MODULO E SEGNO ( Binario Naturale )**

Si utilizza un bit per la rappresentazione del segno ( $0 \rightarrow + ; 1 \rightarrow -$ )

Il problema di questa rappresentazione è duplice:

- > Perdo una rappresentazione per lo zero (che di fatto ha due rappresentazioni)
- > Con 8 bit posso rappresentare valori per 7 bit ovvero da 0 - 127
- > Un incremento binario corrisponde ad un aumento dei numeri positivi, ma un decrescimento dei negativi

(N.B il significato di una sequenza di bit non ha valore predeterminato, ma dipende dal contesto in cui mi trovo e dall'interpretazione che gli viene data)

Anche con questa notazione non risolvo il problema dell'unificazione dei passaggi per sottrazione e somma

Numero rappresentabili con questa notazione:  $-2^{N-1} + 1$  a  $2^{N-1} - 1$

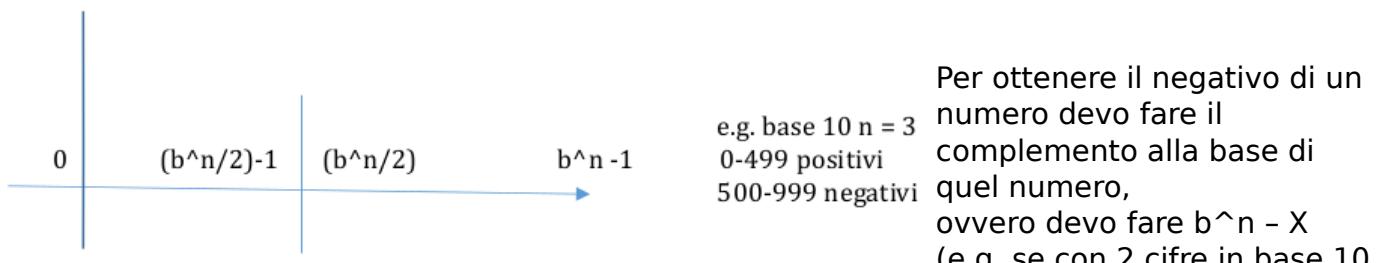
### **COMPLEMENTO ALLA BASE DI UN NUMERO**

N cifre, base B  $\rightarrow$  B N configurazioni

Posso usare la stessa notazione per rappresentare sia positivi che negativi:

- Da 0 a  $[B^N / 2] - 1$  rappresento i numeri positivi con la rapp. Posizionale
- Da  $[B^N / 2]$  fino a  $B^N - 1$  rappresento i negativi (non è bilanciata in quanto ho un numero negativo che non ha il corrispettivo positivo)

(e.g. con 4 bit rappresento i positivi da 0 a  $2^4 / 2 - 1$  ovvero da 0 a 7. Mentre da 8 a 15 rappresenterei i corrispettivi numeri negativi)



se volessi rappresentare -40 rappresenterei  $100 - 40 = 60$

All'atto pratico questo si traduce con il mantenere tutti i bit uguali partendo dai più significativi fino al primo 1, dopodiché complementare ogni numero ( 0 -> 1 ; 1 -> 0 )

Il -1 è rappresentato sempre da tutti uni

NB anche la sottrazione può generare OF, e.g  $0 - (-4) \rightarrow +4$  NON rappresentabile in c2 con 3 bit

Per complementare il numero devo sottrarre al numero  $2^N$  il numero positivo di cui voglio fare il negativo; o alternativamente complementare tutti i numeri fino all'ultimo 1 meno significativo incontrato

*Estremi altre basi:*

**16** → 0 / 0x7F positivi || 0x80 / 0xFE negativi

**8** → 0 / 37 positivi || 40 / 77 negativi

#### COMPLEMENTO ALLA BASE - 1 DI UN NUMERO

Anche in questo caso utilizzo metà intervallo per i numeri maggiori di 0 e l'altra metà per i numeri minori di zero. La differenza sta nel minimo numero negativo rappresentabile.

Se infatti per la parte positiva non cambia assolutamente nulla, andando da 0 a  $[ b^n/2 ] - 1$ , con i numeri negativi vado da  $b^n/2$  a  $b^n - 1$  in cui però il massimo dei numeri rappresentabili non corrisponde più a -1 ma a 0-

Pertanto per rappresentare un numero negativo a partire a un positivo X la formula sarà :  
 $(b^n - 1) - X$

Invece che complementare un numero a  $B^n$  lo complemento a  $(B^n) - 1$ : pertanto ad esempio nel caso di  $B=10$  se volessi rappresentare -36 con 2 cifre lo complementerei a 99, ottenendo quindi  $-36 = 63$ ;

A livello pratico la grossa differenza con la notazione precedente è che invece che mantenere invariati i bit fino al primo uno a partire dalle cifre più significative, qui per ottenere il corrispettivo negativo e sufficiente invertire i valori di 0 e 1 di tutto il numero

(e.g. 0110101 = 53; se volessi rappresentare -53 basterebbe invertire e ottenere -> 1001010)

Pertanto riassumendo  $C_b = b^n - 1$

$$C(b-1) = (b^n - 1) - X$$

$$C_b - C(b-1) = 1$$

Se volessi sapere le cifre rappresentabili in ogni metà (positiva e negativa) di una base 16?

Avrei che il numero massimo positivo rappresentabile è  $(16^3)/2 - 1$ , ovvero  $2^7/2 - 1$ , ovvero  $2^6$ , che in esadecimale sarebbe 800.

Pertanto il massimo numero positivo rappresentabile in esadecimale sarà 7FF, il successivo, 800, rappresentera il corrispettivo negativo

Obs: ammettendo di essere in complemento vale la regola

$$A - B = A + (2^k - B) = A + (2^k - B - 1) + 1$$

Dove gli elementi all'interno delle parentesi sono esattamente i corrispettivi negativi del numero B. Ciò permette di eseguire, almeno in binario, una notevole semplificazione di quellache e la differenza in una serie di somme, senza i problemi di controllo di segno e valore

assoluto che si porta dietro la prima operazione

e.g. in C10 → voglio rappresentare -36 con 2 numeri →  $10^2 \rightarrow 100 \rightarrow 100 - 36 = 64$   
64 equivale a -36 rappresentato in C10

Problema sottrazione: per eseguire una sottrazione devo eseguire diversi passaggi, obbligatoriamente in sequenza e dipendenti da precedente.

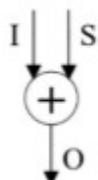
- > confrontare i segni
- > identificare il maggiore tra i due addendi
- > eseguire la somma algebrica tra i due
- > riaggiustare i segni a seconda del maggiore in va

Per ovviare a questo problema si è ricorso alle notazioni in complemento, che permettono di implementare su un unico circuito sia la somma che la differenza, senza dover passare dei suddetti steps;

Se infatti devo calcolare  $(A - B)$  essa equivale, nel caso del complemento, a  $A - (b^k - B)$  dove  $b$  è la base e  $b^k$  rappresenta il riporto da eliminare; introducendo anche un +/- 1 ottengo  $A - (b^k - B - 1) + 1$  che seppur notevolmente meno vantaggioso per un calcolo naturale a cui siamo abituati con la B10 porta a dei notevoli vantaggi del punto di vista circuitale:

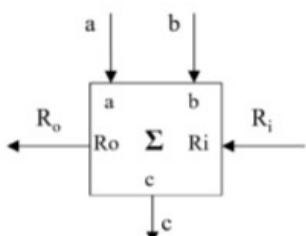
All'interno della parentesi infatti si trova esattamente la descrizione matematica del complemento a  $b-1$ , che nel caso di bit e base binaria si implementa semplicemente invertendo tutti i bit del v.a. del numero di cui voglio fare la differenza, ottendendo  $A+B^*+1$  dove  $A + B^*$  equivale ad  $A + (-B)$  in C1; aggiungendo 1 in ingresso passo dalla base in C1 alla base in C2;

Dal punto di vista circuitale questa selezione di invertenza o meno viene implementata con un exor, ovvero una somma a modulo 2 per la quale se  $S$  (selettore) = 0 in output ho esattamente l'input, se  $S = 1$  ho la negazione dell'input



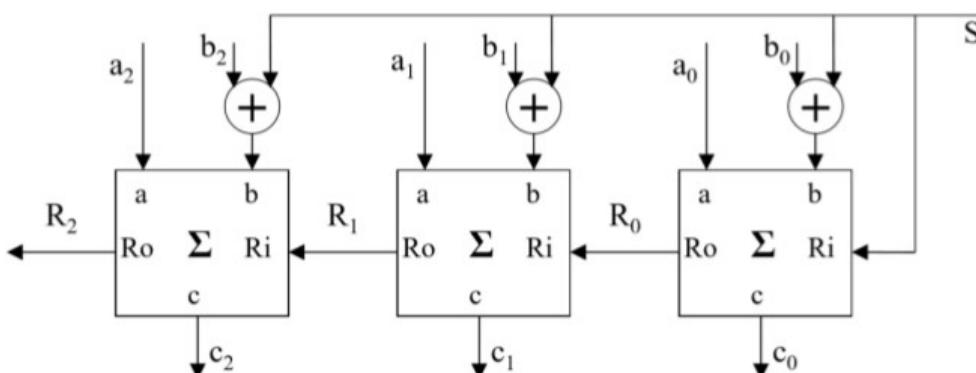
S	I	O
0	0	0
0	1	1
1	0	1
1	1	0

Questo c.to può essere implementato all'interno di un sommatore a 2 bit + riporto per decidere se effettuare una somma o una differenza (ovvero una somma con il complementato di B)



a	b	R <sub>i</sub>	c	R <sub>o</sub>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A sua volta un sommatore binario con riporto e complemento può essere combinato con altri per ottenere un sommatore binario completo in cui  $S$  decide se sia somma o differenza tramite complemento;



$S = 0 \rightarrow$  somma binaria v.a. con propagazione di riporto  
 $S = 1 \rightarrow$  somma con il complementato di B, ovvero  $A + B^* + 1 = A - B$

## OVERFLOW E RIPORTO:

010011 + sarebbe giusta tale rappresentazione? **DIPENDE** dalla convenzione con cui sto  
 010001 = lavorando; se infatti fossi in v.a. avrei un risultato giusto, se invece fossi in C2  
 ----- avrei una condizione di **overflow** (ovvero una condizione per la quale comando  
 100100 due numeri positivi esco dal range dei rappresentabili e sfocio nei negativi)  
 [sarebbe overflow anche se comando due negativi ottenessi un numero  
 positivo]

**NB:** l'overflow è diverso dal **riporto** che è la condizione che si genera quando attraverso la somma di due numeri in valore assoluto ottengo un numero non rappresentabile con il # di bit a disposizione;

11111011 + In questo caso si ha una condizione di riporto per cui comando due numeri in  
 11110000 = valore assoluto di 8 bit ottengo un numero che in v.a. richiederebbe 9 bit per  
 ----- essere rappresentato; pertanto mentre in C2 sarebbe un risultato esatto, in va  
 (1)11101011 esso non lo è in quanto rappresenta 235 invece di 491 (manca infatti il bit di peso 256)

Riassumendo le condizioni di overflow si generano nel caso di somme algebriche tra numeri in C2/C1 con segni differenti o con differenze A - B dove A=0 (se infatti B = max negativo in C2 avrei il negativo che non ha corrispettivo positivo come risultato, ottenendo così 0 invece del positivo corretto); dal punto di vista del riporto invece esso si verifica nel caso di somme di numeri in va che sommati portano oltre la massima rappresentabilità di bit disponibili

$0 \leq A \leq 2^{N-1} - 1$	$-2^{N-1} \leq A < 0$	$0 \leq A \leq 2^{N-1} - 1$	$-2^{N-1} \leq A < 0$
$0 \leq B \leq 2^{N-1} - 1$	$0 \leq B \leq 2^{N-1} - 1$	$-2^{N-1} \leq B < 0$	$-2^{N-1} \leq B < 0$
$0 \leq S \leq 2^N - 2$	$-2^{N-1} \leq S < 2^{N-1} - 1$	$-2^{N-1} \leq S < 2^{N-1} - 1$	$-2^N \leq S < 0$
Sommando due numeri positivi si ha overflow se si ottiene un numero negativo. S potrebbe non essere rappresentabile in N bit ma lo è sempre in N+1 bit.	Non ci sono mai problemi di overflow.	Non ci sono mai problemi di overflow.	Sommando due numeri negativi si ha overflow se si ottiene un numero positivo. S potrebbe non essere rappresentabile in N bit ma lo è sempre in N+1 bit.

## Problema:

ipotizzando di avere registri con i seguenti valori:

**\$t0** = 011.....1

**\$t1** = 100.....0

Come faccio ad eseguire un controllo tramite un SetIfLessThan ed essere sicuro di non avere errori?

**Slt \$t2, \$t0, \$t1;**

(se infatti la facessi senza prestare attenzione alla rappresentazione utilizzata nel suddetto caso avrei un negativo che risulterebbe maggiore di un positivo, con conseguente set a 0 di un flag che dovrebbe essere a 1)

Per questo motivo sono state implementate all'interno del MIPS due tipologie di istruzioni, anche per i confronti.

**slt \$t2, \$t0, \$t1**

permette il cfr tra i registri IN C2, pertanto mi permette di fare confronti tra numeri in C2 considerando correttamente i negativi;

```
sltu $t2, $t0, $t1
```

Mi permette di fare un confronto tra i due registri in termini di valori assoluti, ad esempio se sto confrontando l'indirizzo di due celle di memoria o due valori ASCII in cui i negativi son privi di significato devo utilizzare un cfr sui va

**NB:** l'errato utilizzo di una delle due istruzioni precedenti genera errori di set all'interno del registro di destinazione che sono difficilmente individuabili in quanto non sono errori sintattici di scrittura errata di un'istruzione, quanto più errori logici di confronto tra due numeri, NON segnalati dall'assemblatore

Anche nel caricamento nei registri è necessario prestare attenzione alla convenzione, pertanto esistono anche in questo caso due istruzioni che permettono di evitare errori di logica nei numeri rappresentati

**1b** → carica il byte ESTENDENDO il segno, pertanto mantenendo la rappresentazione in C2/C1  
**1bu** → carica il byte *unsigned* ovvero SENZA ESTENDERE il segno (mantenendo il v.a. utile per rappresentare e caricare valori ASCII o indirizzi di memoria)

### ? = come gestisco l'overflow ?

Una condizione di overflow nel mips, a differenza di altri processori, NON presenta i flag di controllo, ma data la difficoltà nel decifrarne gli errori presenta una casistica di utilizzo: se infatti si genera overflow automaticamente avvengono due operazioni, dopo che la ALU comunica l'incongruenza all'unità di controllo:

- > l'indirizzo del PC viene salvato
- > si genera un *jump* ad una sub di gestione dell'eccezione chiamata *exception handler*

Così facendo l'eccezione viene gestita da una sub che ne corregge i possibili errori; come ritorno dalla sub al main?

*Solitamente* avrei utilizzato

```
jr $ra
```

tuttavia non è possibile in quanto \$ra potrebbe essere in uso; per questo motivo l'indirizzo del PC viene salvato nel registro chiamato **\$epc** (Exception Program Counter)

Potrei pertanto utilizzare

```
jr $epc
```

NO, infatti non c'è corrispondenza nei 32 registri interni (rappresentabili con i 5 bit dell'istruzione); devo quindi usare un'istruzione particolare apposita in due passaggi:

- nfc0 \$k0, \$epc
- jr \$k0

dove il fatto di utilizzare i registri k0 deriva dal dover utilizzare registri che non siano propri dell'assemblatore né disponibili al programmatore, per evitare cancellazione di dati utili, mentre l'istruzione **mfco** è l'istruzione che permette il ritorno dell'indirizzo di \$epc (*Move From Coprocessor 0*)

Qualora io utilizzassi istruzioni del tipo addu, subu, sltu NON avrei overflow lavorando in v.a.

### ? e con il riporto ?

E' il programmatore a dover gestire questa tipologia di errore analizzando i risultati delle operazioni di addu, subu etc e utilizzando una SetonLessThan

Se infatti dovessi eseguire una somma tra valori assoluti generando riporto non ptrei utilizzare più dei 32 bit a disposizione, pertanto dovrei eseguire una slt tra il registro contenente il risultato e quello contenente uno dei due operandi (se è minore la somma dell'operando si è verificata una condizione di riporto)