

►CAPITOLO I◄

INTRODUZIONE

Lezione del 26/09/07.

CISC vs. RISC

I processori possono seguire due logiche distinte:

La logica **CISC** → calcolatori a set di istruzioni complesse. Ci sono tante istruzioni diverse per compiere una sola operazione (ad esempio nello Z80 ci sono più di 10 diverse istruzioni di `add`).

Vantaggi: programmi più compatti.

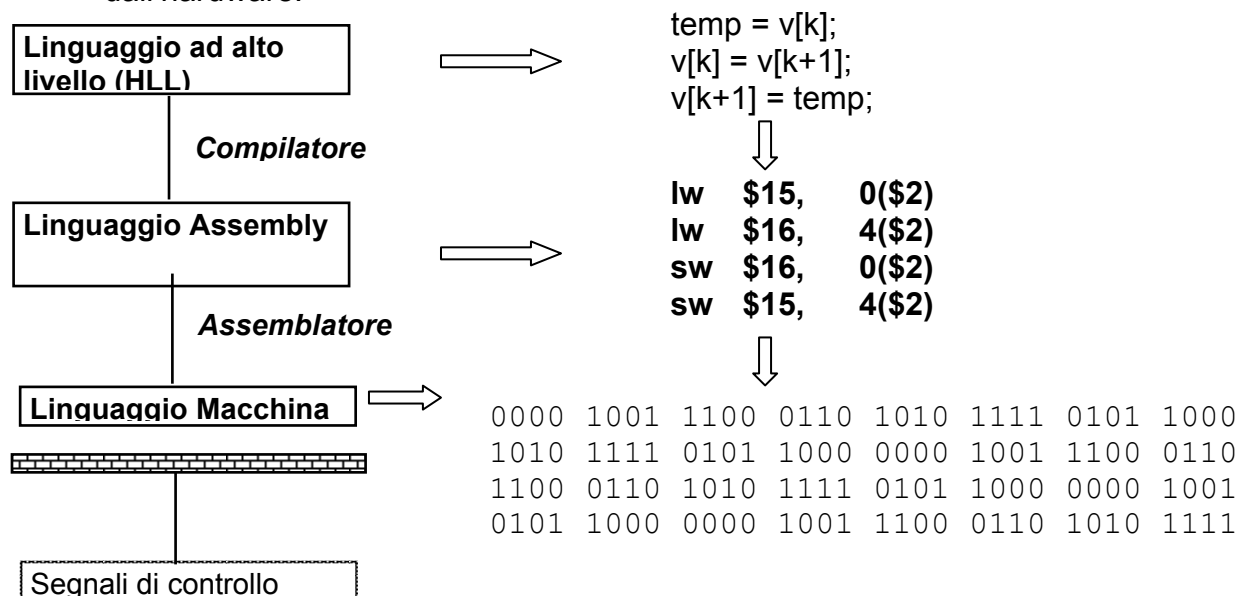
Svantaggi: **effetto $n+1$** => un processore ha n istruzioni; ne aggiungo una, cioè progetto un supporto hardware in più per eseguire questa istruzione aggiuntiva. Purtroppo, però le n istruzioni iniziali del mio calcolatore vengono penalizzate dall'aggiunta di questa istruzione, soprattutto nella fase di *fetch* e di *decodifica* dell'istruzione:

- ▷ nella fase di *fetch*, l'aggiunta di un'istruzione implica conseguenze in quanto se ho poche istruzioni, queste si compongono di pochi bit e quindi trasferirli dalla memoria è più semplice che non se ho molti bit, cosa che accade se ho tante istruzioni;
- ▷ *decodificare* un'istruzione, infatti, significa capire, date n linee in ingresso, quali delle 2^n condizioni è quella che realmente mi interessa; aggiungere delle istruzioni implica quindi un numero maggiore di possibilità di scelta e quindi un aumento del tempo per reperire quella di mio interesse.

La logica **RISC** → calcolatore a set di istruzioni ridotto. Una sola istruzione per compiere una sola operazione (nel MIPS una sola istruzione di `add`)

Vantaggio: rapidità nell'esecuzione di una istruzione.

Svantaggio: il programmatore si deve adeguare alle poche istruzioni rese disponibili dall'hardware.



I calcolatori elettronici comprendono solo 2 segnali: on e off (vero o falso, acceso o spento...) quindi l'alfabeto macchina consta di 2 simboli: 0 e 1. Il linguaggio macchina è un insieme di numeri in base 2 dove ogni lettera è una cifra binaria detta bit. I calcolatori funzionano grazie alle istruzioni che il programmatore gli fa eseguire: le istruzioni sono semplicemente una serie di bit comprensibili al calcolatore. Poiché comunicare con il

calcolatore attraverso numeri binari è molto complicato per gli esseri umani, sono stati inventati dei programmi in grado di tradurre una notazione simbolica, comprensibile per l'uomo, in codice binario. Il primo di questi programmi aveva il nome di **assemblatore** e traduceva la versione simbolica delle istruzioni, detta *assembly language*, della corrispondente forma binaria, cioè in linguaggio macchina. Sono stati poi inventati i linguaggi ad alto livello, linguaggi cioè molto più simili alla logica del programmatore che a quella della macchina: tali linguaggi vengono tradotti da un compilatore in un linguaggio di tipo assemblatore, il quale a sua volta lo traduce in linguaggio macchina.

Commenti alle istruzioni:

lw \$15, 0(\$2)

Istruzione che carica dalla memoria ciò che trova all'indirizzo contenuto in \$2 nel registro \$15. Per eseguire questa istruzione è dunque necessario:

- l'indirizzo di memoria al quale trovare l'informazione di mio interesse → per identificare l'indirizzo di memoria di mio interesse servono 32 bit, in quanto il MIPS ha 32 bit sul bus degli indirizzi.
- la destinazione, cioè il registro nel quale porre l'informazione che arriva dalla memoria → il MIPS ha a disposizione 32 registri. Servono 5 bit per identificare il registro di mio interesse.

ISTRUZIONE

6 bit	5 bit	32 bit
Codice operativo	Destinazione = registro	Indirizzo di memoria.

6+5+32=43 bit! Sono troppi, poiché il MIPS ha il bus dati da 32 bit!

Si può notare che il dispendio maggiore di bit è utilizzato nell'identificare l'indirizzo di memoria da cui prelevare il dato. La soluzione è la seguente: nell'istruzione non specifico l'indirizzo di memoria a cui devo andare, ma un registro in cui è presente l'indirizzo di memoria a cui devo andare, precedentemente caricato in tale registro.

6+5+5=16 bit!

I 16 bit mancanti dei 32 trasferiti, vengono utilizzati come offset, cioè come spostamento, rispetto all'indirizzo di memoria a cui punta il registro. Lo spostamento possibile è di $2^{15} < S < 2^{15}-1$, poiché i numeri sono in complemento a 2.

6 bit	5 bit	5 bit	16 bit
Codice operativo	Destinazione = registro	Registro con l'indirizzo.	Spostamento

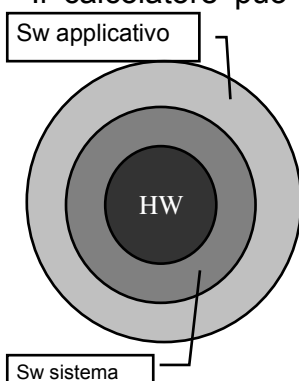
sw \$16, 0(\$2)

Istruzione che scrive il contenuto del registro \$16 all'indirizzo di memoria contenuto in \$2.

6 bit	5 bit	5 bit	16 bit
Codice operativo	Sorgente = registro	Registro con l'indirizzo.	Spostamento

Struttura del software.

Il calcolatore può eseguire solo istruzioni di basso livello. Passare da una complessa applicazione descritta in un linguaggio ad alto livello più vicino a quello umano fino ad arrivare a semplici istruzioni comprese dalla macchina, coinvolge diversi strati di software che interpretano e traducono le operazioni di alto livello in operazioni semplici per il calcolatore. Questi strati di software sono organizzati in maniera gerarchica:



- nel cerchio più esterno compare l'**applicazione software**, cioè programmi utente o mirati all'utente (editors, spreadsheet);

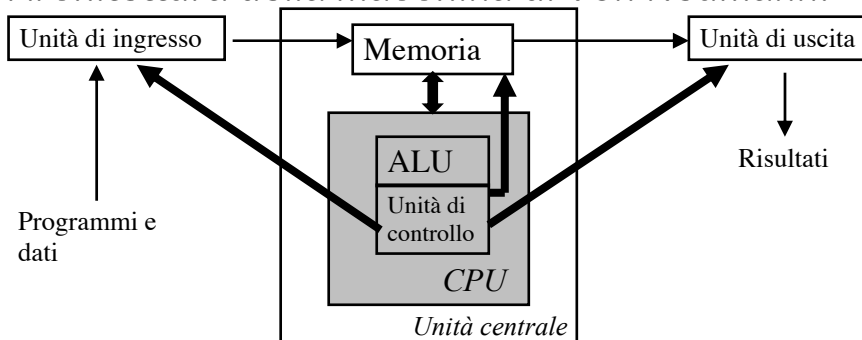
nel cerchio intermedio compaiono delle componenti software, dette **software di sistema**, cioè l'insieme di programmi che forniscono servizi (Sistema Operativo, compilatori). Esistono molti tipi di software di sistema, ma i due essenziali sono:

- il sistema operativo → opera come interfaccia tra il programma utente e l'hardware, provvedendo a funzioni di supervisione, di gestione delle operazioni base di I/O, di gestione di spazio sui dispositivi di immagazzinamento dati e memoria, di condivisione del calcolatore tra più applicazioni...
 - il compilatore → traduce un programma scritto in un linguaggio ad alto livello, C o Java, in istruzioni che l'hardware può eseguire.
- nella componente più interna è presente l'**hardware**.

I linguaggi HLL permettono

- progettazione in linguaggio simile a quello naturale
- una maggior concisione rispetto al linguaggio macchina
- indipendenza dal calcolatore
- riutilizzo routine frequentemente impiegate ⇒ librerie di subroutine

Architettura della macchina di Von Neumann.



L'architettura della maggior parte dei calcolatori elettronici è organizzata secondo il modello della Macchina di Von Neumann.

La Macchina di Von Neumann è costituita da tre elementi funzionali fondamentali:

- ☆ **Unità di input**, tramite la quale i dati e i programmi vengono inseriti nel calcolatore per essere elaborati.
- ☆ **unità centrale di elaborazione**, composta da:
 - ⊗ memoria centrale;
 - ⊗ CPU (che a sua volta comprende ALU e unità di controllo). È la parte attiva di un calcolatore, quella che esegue fedelmente le istruzioni di un programma, sommando numeri, eseguendo confronti su di essi, segnalando ai dispositivi di I/O di attivarsi...
 - ⊗ I bus di comunicazione.
- ☆ **Unità di output**, necessaria affinché i dati elaborati possano essere restituiti all'operatore.

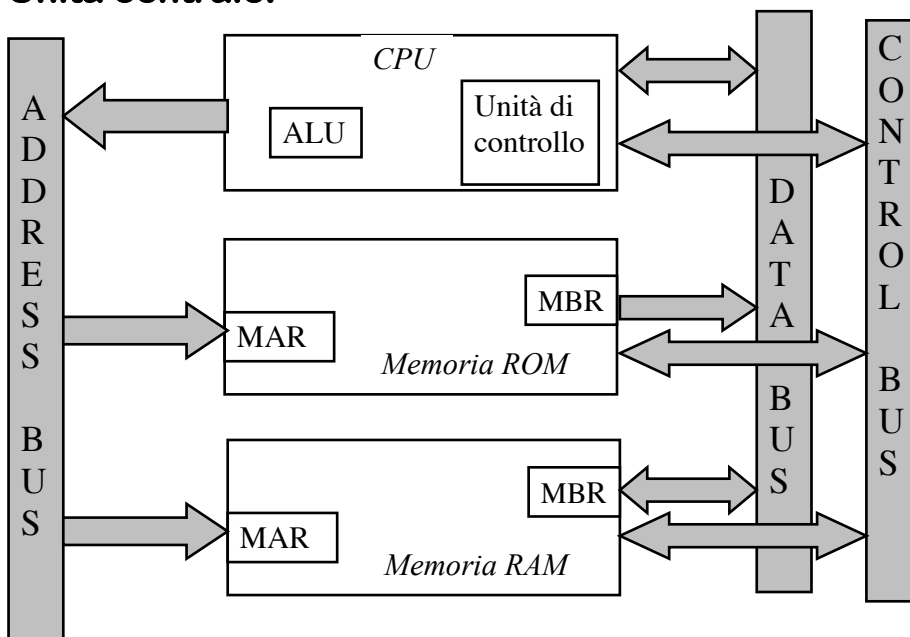
In un computer la circuiteria che esegue le operazioni sui dati è isolata nella CPU che è composta da un'unità aritmetico-logica, che racchiude i circuiti che eseguono l'elaborazione dei dati vera e propria e l'unità di controllo, che contiene i circuiti necessari per coordinare le attività della macchina. Inoltre, per la memorizzazione temporanea delle informazioni, la CPU contiene dei registri. I registri generici fungono da luoghi di memorizzazione temporanea per i dati che sono elaborati dalla CPU: conservano i dati in ingresso alla circuiteria della ALU e forniscono uno spazio di memorizzazione per i risultati ottenuti. Per eseguire un'operazione sui dati registrati nella memoria principale, l'unità di controllo deve prima trasferirli nei registri generici, informare la ALU della nuova collocazione, attivare la circuiteria appropriata e comunicarle il registro che deve ricevere i

risultati. Per trasferire i dati, l'unità centrale e la memoria principale sono collegate da un insieme di fili detti bus. Tramite i bus la CPU è in grado di estrarre, leggere, posizionare o scrivere i dati in memoria principale, specificando l'indirizzo della relativa cella di memoria che si vuole raggiungere o in cui si vogliono depositare i dati. Sui bus transitano dati, indirizzi, segnali di controllo.

Un *programma eseguibile* dalla macchina di Von Neumann consiste in una lista di istruzioni registrate in memoria centrale, che devono essere eseguite una alla volta secondo l'ordine specificato nel programma fino a quando non si incontra un'istruzione di controllo, la quale può alterare il flusso sequenziale stabilendo il numero d'ordine della successiva istruzione da eseguire.

Lezione del 8/10/07

Unità centrale.



Nei circuiti elettronici digitale sono 3 le condizioni di funzionamento:

- forzato a 0;
- forzato a 1;
- condizione di non interferenza, cioè di alta impedenza che esclude la linea dal circuito. Questa è determinata dal bus control, dal segnale di read o di write.

Bus control.

- Linea di **read** (rd) e di **write** (wr) comandate dalla CPU, che controlla istante per istante che riceve e chi pilota le linee;
- Il calcolatore è una macchina sincrona, legata cioè ad una temporizzazione: è necessario quindi un segnale di temporizzazione sulle linee del bus control: segnale di **ck**.
- Linea di **interrupt** (int): interrompe l'esecuzione di un programma per dare priorità a eventi più importanti avvenuti eccezionalmente.
- Legata alla linea di interrupt è la linea di **accettazione di interrupt** (int ack), cioè un segnale che viene inviato da chi ha ricevuto il comando di interrupt, per accettarlo o rifiutarlo. È un segnale di conferma di accoglimento della richiesta dell'interrupt.
- La linea di **wait**.

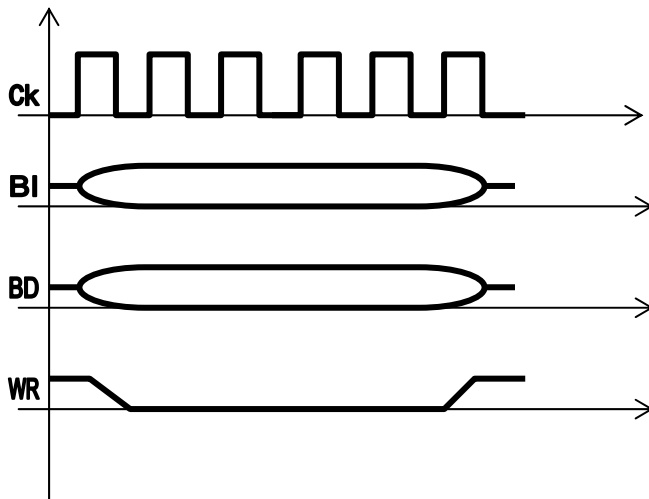


Diagramma temporale che rappresenta le componenti attivate dalla CPU per effettuare una scrittura in memoria. Vengono attivati:

- › Il bus indirizzi, nel quale è specificata l'indirizzo della cella di memoria di mio interesse;
- › Il bus dei dati, nel quale è presente il dato da salvare in memoria;
- › il segnale di write che rimane attivo finché la scrittura è terminata.

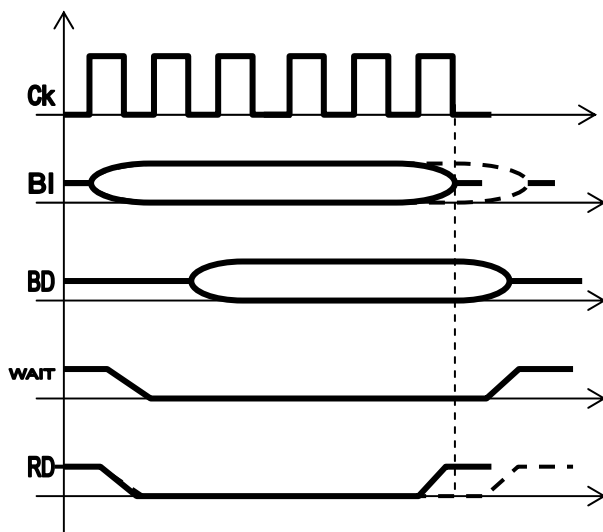


Diagramma temporale che rappresenta le componenti attivate dalla CPU per effettuare una lettura dalla memoria. Vengono attivati:

- › Il bus indirizzi, nel quale è specificata l'indirizzo della cella di memoria in cui si trova il dato mio interesse;
- › Il bus dei dati, nel quale è presente il dato che la CPU ha richiesto. Tuttavia ora il bus dati non può essere attivato contemporaneamente al bus indirizzi, in quanto è necessario del tempo per reperire l'informazione in memoria. Il dato non è subito reperibile sul bus, poiché prima bisogna raggiungere la casella in cui è presente il dato e poi inviarlo sul bus.

Tuttavia la CPU è molto più veloce della memoria: il periodo di clock della CPU può essere ad esempio 2 GHz, mentre la memoria ha velocità dell'ordine dei MHz! C'è il rischio che la CPU legga il dato dalla memoria, ancora prima che questo sia pronto, con la conseguente lettura di bit casuali. Per questo è necessaria la linea del bus dei controlli di **wait**: il segnale di wait viene inviato dalla memoria alla CPU per comunicare che il dato richiesto non è ancora pronto. La CPU fa permanere il segnale di read finché permane il segnale di wait. Lo svantaggio nell'adottare questa situazione consiste nel fatto che la CPU viene rallentata, poiché è costretta ad adeguarsi ai tempi di ricerca della memoria. Il ciclo si conclude quando il segnale di wait si interrompe e quindi anche il segnale di read cessa.

Le dimensioni del:

BUS INDIRIZZI → influenza la massima dimensione di memoria raggiungibile, lo spazio di memorizzazione: ho 2^m celle di memoria, se m è il numero dei bit del bus.

BUS CONTROLLI → dipendono dal dispositivo e dalle peculiarità del sistema.

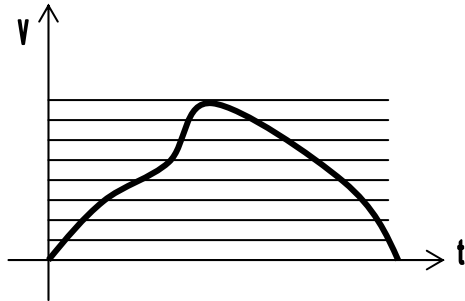
BUS DATI → influenza il numero di dati che il processore è in grado di elaborare parallelamente. I microprocessori attuali hanno bus dati a 8, 16, 32, 64 bit.

Lezione del 10/10/07

Esempio:

Variazione nella precisione di misura di 1 Kg. in un sistema di pesatura basato su microprocessori con diversa dimensione del bus dati

Il peso è una grandezza analogica: devo trasformarlo in una grandezza digitale con un Analog to Digital Converter.



Divido l'intervallo di misurazione in tanti intervallini: il primo assume valore identicamente uguale a 0, l'ultimo uguale a 1. Il numero di intervallino influisce pesantemente sulla precisione di rappresentazione.

Se il numero di bit del bus dati ha 4 bit, posso rappresentare 2^4 informazioni diverse; 2^4 è cioè il numero di intervallino in cui ho diviso l'intervallo di misurazione. La massima differenza tra un intervallino e , cioè la grandezza dell'intervallino, è 6.25%.

<i>Numero di bit bus dati</i>	4	8	16
<i>Dati rappresentabili</i>	$2^4=16$	$2^8= 256$	$2^{16}= 65.536$
<i>Precisione relativa</i>	6.25%	~3.9 ‰	~0.015‰
<i>Precisione max.</i>	62.5 gr	~3.9 gr	~0.015 gr

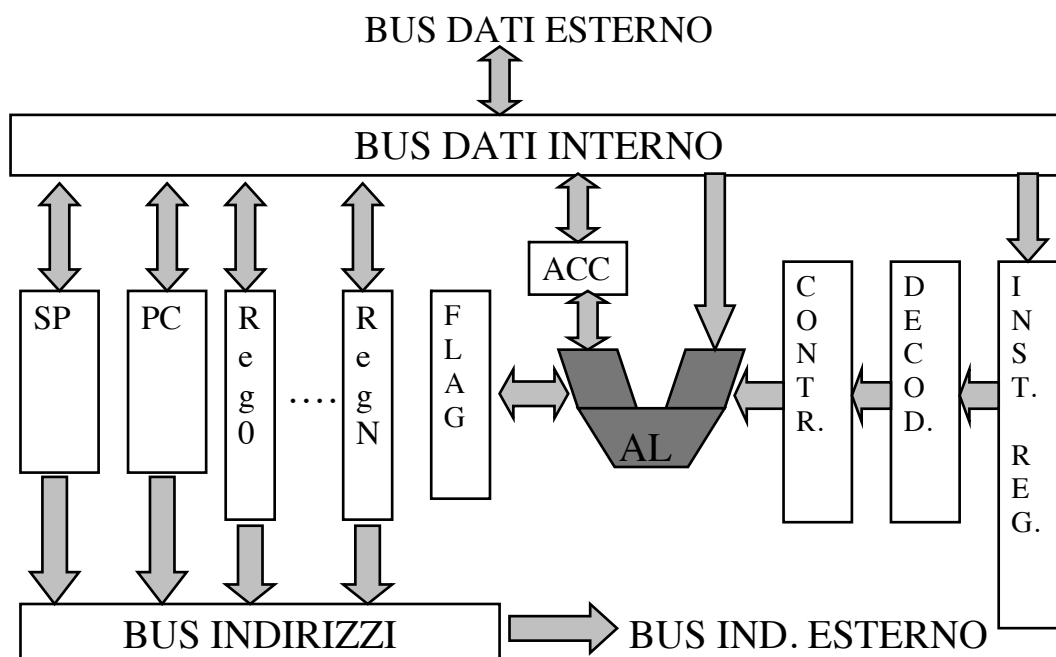
Dispositivi single chip: racchiudono all'interno di un unico circuito integrato, memorie, bus, CPU.

Vantaggio: occupazione di uno spazio minimo.

Svantaggio: non è espandibile.

DSP→ Digital Signal Processor. Sono CPU particolari per elaborare dati con particolare rapidità.

La CPU.



CPU → Central Processing Unit = processore. È il vero “cervello” di qualsiasi computer. È costituito da uno o più chip, circuiti integrati che sono in grado di eseguire istruzioni di programma, leggere e scrivere nella memoria. La CPU include tutti i circuiti logici che permettono di prelevare in memoria una istruzione con gli operandi, decodificarla, eseguirla e trasferire qualsiasi dato verso la memoria o altri dispositivi.

È formata da:

Bus dati interno → permette alla Cpu di scambiare dati tra le varie parti che la compongono e i dispositivi esterni (memorie). \longleftrightarrow Entrano ed escono i dati.

Bus indirizzi interno → permette alla Cpu di identificare gli indirizzi dei dati e delle istruzioni di cui necessita per eseguire le operazioni richieste e per immagazzinare in dispositivi esterni i risultati ottenuti. \longrightarrow Solo uscente: la CPU richiede tramite il bus indirizzi interno e poi quello esterno, alle memorie, gli indirizzi in cui sono contenuti i dati e le istruzioni di cui necessita. Questi entrano nella CPU attraverso il bus dati.

REGISTRI INTERNI → sono un insieme di memorie ad altissima velocità della capacità di pochi byte.

MBR → Memory Buffer Register: vi vengono temporaneamente collocati i dati provenienti dalla memoria a seguito di lettura, o da ricopiare in memoria in caso di scrittura.

MAR → Memory Address Register: contiene l'indirizzo della locazione di memoria nella quale si trova il dato da leggere o da scrivere che transiterà nel MBR.

Registri generici → sono destinati a contenere dati, risultati intermedi o indirizzi di memoria durante le elaborazioni. \longleftrightarrow ENTRANTE E USCENTE VERSO IL BUS DATI : i dati intermedi possono essere memorizzati e riutilizzati dalla ALU.

\longrightarrow USCENTE VERSO IL BUS INDIRIZZI: i dati contenuti possono essere memorizzati in qualche dispositivo esterno all'indirizzo fornito dal registro.

Program Counter –PC– → è un registro particolare in cui viene memorizzato l'indirizzo di memoria al quale il processore trova la prossima istruzione da caricare ed eseguire. Ogni programma, infatti, viene memorizzato come una sequenza ordinata di istruzioni, ciascuna delle quali possiede un proprio indirizzo. Un incremento del PC comporta, al ciclo successivo, il caricamento dell'istruzione immediatamente successiva a quella eseguita. È un registro che si auto-aggiorna. Tuttavia alcune istruzioni di jump, modificano il contenuto del PC in modo che una locazione diversa da quella immediatamente successiva diviene la prossima istruzione da eseguire. Perciò, se l'istruzione di *jump*, il PC salta all'indirizzo di memoria indicato nell'istruzione di *jump*.

\longleftrightarrow Verso il bus dati interno USCENTE: richiede informazioni alle memorie esterne se un'operazione va rifatta n° volte. ENTRANTE: carica la successiva istruzione che il processore deve eseguire.

\longrightarrow USCENTE VERSO IL BUS INDIRIZZI: punta all'indirizzo a cui trovare l'istruzione successiva.

Stack pointer → è un registro della CPU che contiene l'ultimo indirizzo dell'ultima istruzione che la CPU ha temporaneamente abbandonato per eseguirne un'altra. Lo stack pointer punta alla sommità dello stack, cioè ad un'area di memoria gestita in logica LIFO, *last in first out*, cioè l'ultimo valore introdotto è il primo ad uscire. È grazie allo stack pointer che possiamo gestire situazioni di chiamate a sottoprogrammi, senza perdere l'operazione interrotta.

\longrightarrow USCENTE VERSO IL BUS INDIRIZZI: ha bisogno di un circuito di incremento e decremento per funzionare.

IR → Instruction Register: vi viene memorizzata l'istruzione da eseguire, letta dalla memoria. NB: solo il codice operativo!

ALU (Arithmetic and Logic Unit) → è la parte che compie effettivamente i calcoli aritmetici e logici utilizzando i valori depositati nei registri durante la fase di caricamento delle

istruzioni (fetch). La ALU ha 2 ingressi dal bus dati interno: uno passa dall'accumulatore, uno diretto. \longleftrightarrow DAL BUS DATI ALL'ACCUMULATORE: nell'Acc. viene caricato uno degli operandi che servono per eseguire l'istruzione. Alla fine dell'esecuzione dell'istruzione, il risultato, salvato nell'accumulatore, attraverso il bus dati interno viene caricato nei registri interni, oppure, attraverso il bus dati interno e poi esterno, nelle memorie esterne. \longleftrightarrow DALL'ACCUMULATORE ALLA ALU: l'operando salvato nell'accumulatore viene prelevato dalla ALU per eseguire l'istruzione e il risultato, definitivo o provvisorio, viene ricaricato nell'accumulatore.

\implies DIRETTO ENTRANTE DAL BUS DATI INTERNO: carica nella ALU l'altro operando che serve per eseguire il calcolo.

La ALU ha alcuni registri interni:

Registro dei FLAG \rightarrow interagisce esclusivamente con la ALU. È un registro speciale che fornisce informazioni riguardo all'ultima operazione aritmetico-logica eseguita. È costituito dall'insieme di più flag, cioè segnalatori che indicano il verificarsi di condizioni particolari, quali: l'overflow, il carry, zero, negative.

\longleftrightarrow CON LA ALU: Il risultato viene mandato dalla ALU al FLAG: questo ci dice se c'è un errore. Gli errori possono essere:

CARRY \rightarrow riporto o prestito: superamento della capacità di rappresentazione dei numeri in valore assoluto.

OVERFLOW \rightarrow superamento della capacità di rappresentazione per i numeri in complemento a 2 (sommando due positivi si ottiene un negativo.)

Accumulatore \rightarrow dove è memorizzato uno degli operandi coinvolti nell'operazione aritmetica o logica e dove rimane memorizzato il risultato di tale operazione;

UNITA' DI CONTROLLO \rightarrow È la parte della CPU che gestisce la successione delle operazioni da svolgere, sincronizzando le attività di tutti gli altri elementi. A tale scopo preleva dalla memoria centrale una alla volta le istruzioni del programma, che vengono caricate nell'IR, le decodifica (tramite il DECODER di istruzioni) e le esegue inviando gli opportuni segnali di controllo agli organi della CPU che ne attuano l'esecuzione.

Instruction Register –IR– \rightarrow vi viene, attraverso il bus dati interno, inserita l'istruzione che deve essere eseguita. Nota bene: solo il codice operativo!!!

Decoder \rightarrow decodifica il codice operativo che arriva dall'IR: fa capire al processore l'istruzione che deve essere eseguita.

Controller \rightarrow è la parte della CPU che gestisce la successione delle operazioni da svolgere e le attività di tutti gli altri elementi, non attivando le parti non necessarie per l'esecuzione dell'istruzione (così rende il processo più veloce).

CPU “standard” vs. MIPS.

CPU standard

```
L1:  ADD A, Reg1
      JP ovfw, Err
      ...
      ...
      ...
      ...
Err:  ...
```

MIPS

```
add $t1, $t2, $t3
...
...
...
```

\rightarrow Nel MIPS **non c'è il flag di overflow**. Quando scrivo `add $t1, $t2, $t3` ignoro il problema che possa generarsi un overflow, scrivendo quindi di seguito altre istruzioni. Se effettivamente non si verifica overflow, posso continuare ad eseguire le operazioni successive senza alcun problema, in quanto non c'è nessun errore. Quando invece si verifica overflow la ALU genera una sorta di interrupt interno che fa saltare in un

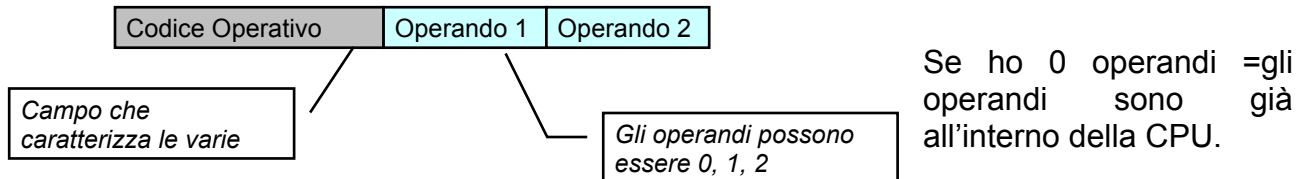
sottoprogramma ben definito che gestisce il problema dell'overflow con istruzioni correttive.

- Vantaggio: quando non si verifica overflow è più vantaggiosa la soluzione del MIPS, in quanto ho delle istruzioni in meno.
 - Svantaggio: ad una prima approssimazione sembrerebbe più svantaggiosa la soluzione del MIPS in caso l'overflow si verificasse, poiché devo interrompere il main program per passare il controllo ad una subroutine, impiegando così più tempo. Tuttavia la probabilità che si verifichi un problema di overflow, dipende dalla dimensione del bus dati: poiché nel MIPS il bus dati è a 32 bit, quindi di grandi dimensioni, ho scarsa probabilità che si verifichi overflow. Nei pochi casi in cui si verifica genero l'eccezione, altrimenti l'esecuzione del programma continua dopo l'operazione di `add`, senza bisogno di aggiungere ulteriori istruzioni che correggano possibili overflow.
 - Poiché avviene di frequente nei programmi di aver bisogno di far confronti con il numero 0, il MIPS prevede che un registro, detto **\$zero**, interno alla CPU assuma sempre valore 0. Questo registro non è modificabile dal programmatore. Il vantaggio che se ne ricava è una maggior velocità nell'effettuare tale confronto, in quanto l'operando 0 è interno alla CPU e non in memoria. Lo svantaggio è legato alla perdita di un registro per la programmazione.
 - Gestione di sottoprogrammi:
 - 1^soluzione: prevedo un registro "PC old" in cui, in caso di chiamata a sottoprogramma, salvo l'indirizzo di rientro al main program. Questa soluzione è funzionale solo nei casi in cui non ci sono chiamate nidificate a sottoprogramma. Sia per, esempio, M il programma principale: in M compare una chiamata al sottoprogramma A; in "PC old" salvo l'indirizzo di rientro verso M, cioè l'indirizzo dell'istruzione in M, successiva a quella che ha effettuato la chiamata a sottoprogramma. Nel sottoprogramma A compare una chiamata al sottoprogramma B; in "PC old" salvo l'indirizzo di rientro verso A, cioè l'indirizzo dell'istruzione in A, successiva a quella che ha effettuato la chiamata a sottoprogramma. Così facendo ho perso l'indirizzo di rientro da A verso M!
 - 2^soluzione: prevedo un'area di memoria gestita con logica LIFO, last in, first out, in cui inserisco gli indirizzi delle varie chiamate a sottoprogramma. Quando un sottoprogramma viene chiamato, l'indirizzo di rientro, cioè quello dell'istruzione successiva a quella che ha effettuato la chiamata, viene memorizzato nello stack, a cui punta lo stack pointer; quando un sottoprogramma termina l'esecuzione, viene letto dalla pila l'ultimo indirizzo caricato che non è stato ancora prelevato e viene trasferito nel Program Counter. La gestione LIFO dello stack permette di nidificare le chiamate ai sottoprogrammi. Tale soluzione è, però, carente dal punto di vista delle prestazioni, in quanto, ogni lettura di memoria richiede molto tempo.
- Il MIPS prevede entrambe le possibilità: Hennessy e Patterson, hanno effettuato degli studi su alcuni dei programmi in uso più importanti e hanno notato che il caso più frequente è quello di una sola chiamata a sottoprogramma e non di chiamate nidificate. All'interno della CPU è presente un registro `$ra`, in cui, quando viene chiamato un sottoprogramma con l'istruzione `jal` (*jump and link*), viene salvato l'indirizzo di rientro al programma chiamante. Poiché tale possibilità non prevede nidificazione, il programmatore, qualora ci fossero chiamate nidificate a sottoprogramma, deve provvedere a salvare gli indirizzi di rientro dei vari sottoprogrammi nello stack.
- Nel MIPS tutte le operazioni matematiche impongono di utilizzare i registri. Gli operandi devono quindi essere preventivamente caricati in opportuni registri. Per questo nel MIPS non c'è l'accumulatore.

Formato di un'istruzione.

Le istruzioni sono codificate da stringhe di bit. Una volta caricata nell'IR, un'istruzione deve essere decodificata ed eseguita. Il decodificatore ha il compito, ricevute n linee in ingresso di capire quale fra le 2^n combinazioni diverse è quella di mio interesse, cioè di attivare in uscita la sola linea corrispondente all'operazione identificata dal codice operativo arrivato in ingresso. A tal scopo l'unità di controllo deve conoscere:

1. **CODICE OPERATIVO** → è il campo che caratterizza le varie istruzioni.
2. **SORGENTE** → dati su cui operare.
3. **DESTINAZIONE** → dove porre il risultato e, se sorgente e destinazione sono in memoria le MODALITA' DI INDIRIZZAMENTO.



Ciclo di esecuzione di un'istruzione.

L'unità di controllo svolge il suo lavoro ripetendo continuamente un algoritmo che la guida attraverso un processo a 3 fasi detto ciclo macchina. Le tre fasi sono:

Fetch → prelevamento dell'istruzione: l'unità di controllo richiede che la memoria principale fornisca l'istruzione memorizzata all'indirizzo indicato dal Program Counter. L'unità di controllo pone l'istruzione ricevuta dalla memoria nel suo registro delle istruzioni e poi incrementa il PC, in modo che esso contenga l'indirizzo della fase successiva.

1. (PC) → MAR
2. ((MAR)) → MBR (PC) + 1 → PC
3. (MBR) → IR.

I passi 1, 2, 3 permettono di caricare in IR (instruction register) il codice operativo (OP Code) dell'istruzione corrente. Passi analoghi permettono di caricare in opportuni registri della CPU gli operandi presenti nell'istruzione. In tal caso, nel passo 3 la destinazione del dato proveniente dalla memoria non è più IR, ma opportuni registri.

Nella fase di fetch sono coinvolti: il PC, l'unità di controllo, i registri (o l'istruzione register).

Decode → decodifica: l'unità di controllo decodifica l'istruzione, cioè scompone il campo operando nei suoi componenti sulla base del codice operativo dell'istruzione.

Sono coinvolti: il decoder.

Execute → l'unità di controllo esegue l'istruzione attivando la circuiteria adeguata a svolgere il compito richiesto. Per esempio se l'istruzione è un'operazione aritmetiche l'unità di controllo attiva la ALU e i registri necessari.

Coinvolti: la ALU, i registri.

Una volta che l'istruzione è stata eseguita, l'unità di controllo inizia un nuovo ciclo con la fase di reperimento.

Esempio 1: Somma tra il contenuto del registro R2 e il contenuto dell'accumulatore. Il risultato va nell'accumulatore.

FORMATO	codice operativo
FETCH	come in precedenza
ESECUZIONE	(R2)+(ACC)→ACC

Esempio 2: somma tra il contenuto della cella di memoria il cui indirizzo è specificato nell'istruzione ed il contenuto dell'accumulatore; il risultato va nell'accumulatore

- FORMATO: codice operativo+operando
- FETCH:
 - 1) (PC)→MAR
 - 2) ((MAR)) →MBR; (PC)+1 →PC
 - 3) (MBR) →IR
 - 4) (PC)→MAR
 - 5) ((MAR)) →MBR; (PC)+1 →PC
 - 6) (MBR) →Rn
- EXECUTE:
 - 1) (Rn) →MAR
 - 2) ((MAR)) →MBR
 - 3) (MBR) →Rn
 - 4) (Rn)+(ACC) →ACC

Esempio 3: saltare all'istruzione che è memorizzata nella cella il cui indirizzo è specificato all'interno dell'istruzione corrente:

- FORMATO: codice operativo+operando
- FETCH:
 - 1) (PC)→MAR
 - 2) ((MAR)) →MBR; (PC)+1 →PC
 - 3) (MBR) →IR
 - 4) (PC)→MAR
 - 5) ((MAR)) →MBR; (PC)+1 →PC
 - 6) (MBR) →Rn
- EXECUTE:
 - (Rn) →PC

►CAPITOLO 2◄

LE ISTRUZIONI

LEZIONE DEL 29/10/07.

CASE/SWITCH

```
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;
}

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

```

    slt $t3, $s5, $zero # k < 0?
    bne $t3, $zero, Exit
    slt $t3, $s5, $t2 # k > 3?
    beq $t3, $zero, Exit
    add $t1, $s5, $s5
    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:

```

Nel linguaggio ad alto livello confronto K con 0, poi con 1, poi con 2, poi con 3...cioè mi chiedo: k=0? Se no mi chiedo K=1? Se no mi chiedo K=2?...alcune condizioni di K sono particolarmente veloci: se K=1, l'esecuzione è veloce... se ho 100 casi e K=100 devo fare 100 test prima di trovare il caso giusto!!! Questa soluzione è particolarmente svantaggiosa per i tempi di esecuzione.

Soluzione:

```

slt $t3, $s5, $zero # k < 0?
bne $t3, $zero, Exit
slt $t3, $s5, $t2 # k > 3?
beq $t3, $zero, Exit

```

Dato l'esempio precedente queste 4 istruzioni controllano se $0 < k < 4$. se questo non è verificato il programma esce e non esegue nessuna delle operazioni richieste.

La soluzione al problema della lentezza di effettuare i test reiterati è la seguente:

si costruisce in memoria una tabella, detta **tabella degli indirizzi di salto**, in cui vengono collocati gli indirizzi di inizio delle varie routine, poiché scrivere il nome di un'etichetta in realtà significa scrivere l'indirizzo che l'assemblatore nella tavola dei simboli ha associato a quella etichetta. L'indirizzo di partenza di questa tabella è contenuta in un registro (\$t4).

Se $K=0 \rightarrow$ devo raggiungere la routine L0 \Rightarrow devo raggiungere la casella in memoria identificata da \$t4.

Se $k=1 \rightarrow$ devo raggiungere la routine L1 \Rightarrow devo spostarmi di 1 word nella tabella degli indirizzi di salto: sommo il valore $4=k*4$ all'indirizzo \$t4.

L0
L1
L2
L3

Se $k=2 \rightarrow$ devo raggiungere la routine L2 \Rightarrow devo spostarmi di 2 word nella tabella degli indirizzi di salto: sommo il valore $8=k*4$ all'indirizzo \$t4...

Guardando il codice:

L'indirizzamento a byte del processore mi impone di moltiplicare il valore di k per 4:

se $K=0 \rightarrow 0$

se $k=1 \rightarrow 4$

se $k=2 \rightarrow 8$

se $k=3 \rightarrow 12$.

add \$t1, \$s5, \$s5

add \$t1, \$t1, \$t1 # $\$t1=4*k$

Ora sommo il valore di K all'indirizzo di partenza della tabella degli indirizzi di salto. Così ottengo l'indirizzo della routine a cui voglio saltare.

add \$t1, \$t1, \$t4

Leggo l'indirizzo della routine a cui voglio andare e lo carico nel Program Counter.

lw \$t0, 0(\$t1)

jr \$t0 #vai a indir. letto

Vantaggi:

- ☆ Il tempo per raggiungere ogni singola routine non varia se varia il numero dei casi da verificare. Ha un tempo di esecuzione piccolo e indipendente dal numero di casi.
- ☆ Il numero di istruzioni non varia al variare del numero dei casi da verificare.

Svantaggi:

- ☆ All'aumentare del numero di casi da verificare aumenta la dimensione della tabella degli indirizzi di salto.

GESTIONE DELLE SUBROUTINE.

Una procedura o subroutine è uno strumento che i programmatori C e Java utilizzano per strutturare i programmi, per renderli più comprensibili e per permetterne il riutilizzo del codice. Le subroutine acquisiscono risorse, svolgono determinate istruzioni e tornano al punto di partenza, restituendo risultati. Durante l'esecuzione di una procedura, il programma esegue e seguenti passi:

1. mettere i parametri in un luogo accessibile alla subroutine;
2. trasferire il controllo alla subroutine;
3. acquisire le risorse necessarie per memorizzare dei dati;
4. eseguire il compito richiesto;
5. mettere il risultato in un luogo accessibile al programma chiamante;
6. restituire il controllo al punto di origine;

I registri sono il luogo che permette l'accesso ai dati più rapido. Il software MIPS usa le seguenti convenzioni per allocare i suoi 32 registri nelle chiamate a sottoprogrammi:

\$a0-\$a3 \rightarrow quattro **registri argomento** per il passaggio dei parametri;

\$v0, \$v1 \rightarrow due **registri valore** per la restituzione dei valori;

\$ra \rightarrow un **registro di ritorno** per tornare al punto di origine (all'istruzione successiva quella che ha effettuato la chiamata).

L'istruzione **jal** (jump and link) salta a un indirizzo e contemporaneamente salva l'indirizzo dell'istruzione successiva a quella di salto nel registro \$ra, detto indirizzo di ritorno. Tale istruzione, quindi, salva il valore PC+4 (il Program Counter è un registro che contiene l'indirizzo dell'istruzione del programma correntemente in esecuzione) nel registro \$ra, per creare il collegamento all'indirizzo dell'istruzione successiva a quella di

chiamata, così da predisporre il ritorno al main. L'indirizzo di ritorno è necessario, poiché la stessa subroutine può essere chiamata da diversi punti del programma.

Per effettuare il ritorno dalla subroutine i calcolatori MIPS utilizzano l'istruzione `jr` (salta tramite registro), che implica un salto incondizionato all'indirizzo specificato in un registro.

La gestione delle subroutine avviene in questo modo:

- › il **programma chiamante** mette i valori dei parametri da passare alla subroutine nei registri **\$a0-\$a3** e utilizza l'istruzione `jal x` per saltare al sottoprogramma X.
- › Il **programma chiamato** (cioè X) esegue le operazioni richieste, memorizza i risultati nei registri **\$v0** e **\$v1** e restituisce il controllo al chiamante con l'istruzione `jr $ra`.

Si supponga che un compilatore abbia bisogno, all'interno di una procedura, di un numero maggiore di registri rispetto ai quattro per i parametri e ai due per i valori da restituire.

Inoltre, per svolgere qualsiasi compito, il programma chiamato necessita dei registri: poiché il numero di registri del MIPS non è elevato, accade che il programma chiamato utilizzi dei registri che già il programma chiamante stava utilizzando: non è possibile che i valori che questi registri contenevano prima della chiamata a sottoprogramma vadano perduti! Qualunque registro utilizzato dal programma chiamante deve, quindi, essere riportato al valore che conteneva prima della chiamata. In questa situazione è necessario **riversare i registri in memoria**. La struttura ideale per riversare i registri è lo **stack**, una coda del tipo last-in-first-out (cioè l'ultimo a entrare è il primo a uscire). Lo stack ha bisogno del puntatore all'indirizzo del dato introdotto più di recente, per indicare dove la procedura può memorizzare i registri da riversare e dove può recuperare i vecchi valori dei registri. Questo **stack pointer** è aggiornato ogni volta che si inserisce o si estrae il valore di un registro. Esistono termini generali per indicare il trasferimento dati da e verso lo stack:

push => memorizzazione di un dato nello stack;

pop => estrazione di un dato dall' stack.

Il software MIPS alloca un altro registro appositamente per lo stack: `$sp`, lo stack pointer denota l'indirizzo dell'elemento dello stack allocato più di recente, il quale mostra dove i registri devono essere salvati o dove si trovano i vecchi valori dei registri salvati. Lo stack cresce a partire da indirizzi di memoria alti verso indirizzi di memoria bassi. Questa convenzione indica che quando vengono inseriti dei dati nello stack, il valore dello stack pointer diminuisce; al contrario quando sono estratti dati dallo stack, aumenta il valore dello stack pointer riducendo la dimensione dello stack.

USO DELLO STACK

```
int proc (int g, int h, int i, int j)
{
    int f;
    f=(g+h) - (i+j) ;
    return f;
}
```

Per convenzione:

\$t0-\$t9 temporanei da non salvare

\$s0-\$s7 da conservare

si potevano risparmiare 2

push/pop

I parametri g,h, i,j corrispondono ai registri argomento `$a0`, `$a1`, `$a2` e `$a3` e f corrisponde a `$s0`.

```
proc:addi $sp, $sp, -12 # 3 push
      sw  $t1, 8($sp)
      sw  $t0, 4($sp)
      sw  $s0, 0($sp)
      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

```

Il programma inizia con l'etichetta della procedura proc. Il passo successivo consiste nel salvare tutti i registri usati dalla procedura. È necessario salvare i contenuti dei registri \$s0, \$t0, \$t1. Si crea così lo spazio per tre parole all'interno dello stack dove vengono memorizzati i vecchi valori:

```
addi $sp, $sp, -12.
```

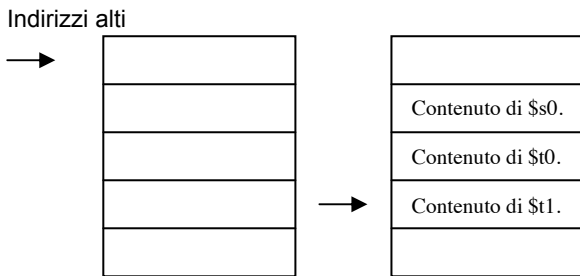
Con questa istruzione si aggiorna lo stack pointer, per fare posto a 3 parole.

```

sw  $t1, 8($sp)
sw  $t0, 4($sp)
sw  $s0, 0($sp)

```

Con queste 3 istruzioni il contenuto dei 3 registri viene salvato in memoria.



Le tre istruzioni successive corrispondono al calcolo che la procedura deve svolgere.

```

add $t0, $a0, $a1 # calc. f
add $t1, $a2, $a3
sub $s0, $t0, $t1

```

Per restituire il valore di f occorre poi copiarlo in un registro di ritorno, cioè in un registro v.

```
add $v0, $s0, $zero # $v0=f
```

Prima del ritorno al programma chiamante i vecchi valori vengono ripristinati all'interno dei registri, estraendoli dallo stack (pop):

```

lw  $s0, 0($sp) # 3 pop
lw  $t0, 4($sp)
lw  $t1, 8($sp)

```

Lo stack pointer viene dunque aggiornato per eliminare i 3 elementi:

```
addi $sp, $sp, 12
```

la procedura termina con un'istruzione di salto tramite registro che utilizza l'indirizzo di ritorno.

```
jr  $ra
```

In questo esempio sono stati utilizzati dei registri temporanei ed è stata fatta l'ipotesi che il loro valore originale dovesse essere salvato e ripristinato. Per evitare di salvare e ripristinare registri il cui valore non viene mai utilizzato, il software MIPS suddivide 18 dei registri in due gruppi:

\$t0-\$t9 → registri temporanei che non sono preservati in caso di chiamata a sottoprogramma: il programmatore è libero di utilizzarli in sottoprogrammi senza dover salvare e ripristinare il valore che avevano prima della chiamata.

\$s0-\$s7 → registri che devono essere preservati in caso di chiamata a sottoprogramma, cioè registri che, se utilizzati, devono essere salvati e ripristinati dal programma chiamato. Questa convenzione riduce la necessità di salvare registri in memoria.

Nell'esempio precedente, dato che il programma chiamante non si aspetta che i registri \$t0 e \$t1 siano preservati durante la chiamata a sottoprogramma si possono eliminare dal codice 2 istruzioni di push e 2 di pop.

Registri preservati	Registri non preservati
Registri: \$s0-\$s9 Stack pointer: \$sp Registro di ritorno: \$ra.	Registri temporanei: \$t0-\$t9. Registri argomento: \$a0-\$a3. Registri valore: \$v0-\$v1.

Lezione del 31/10/2007

PROCEDURE ANNI DATE.

Si supponga per esempio che il programma principale chiami la procedura A con un parametro uguale a 3, mettendo il valore 3 nel registri \$a0 e usando l'istruzione `jal A`. L'indirizzo di ritorno da A al main è dunque contenuto in \$ra. Si supponga poi che la procedura A chiami la procedura B con un `jal B` passandole il valore 7 posto in \$a0. si verificano 2 problemi:

- dato che A non ha ancora finito il suo lavoro, si verifica un conflitto nell'uso del registro \$a0;
- si verifica anche un conflitto nell'utilizzo di \$ra: l'istruzione di `jal B` provvede a salvare l'indirizzo di ritorno della procedura B verso la procedura a in \$ra, eliminando l'indirizzo di ritorno della procedura A verso il main.

Una soluzione consiste nel salvare nello stack tutti i registri che devono essere preservati.

- Il programma chiamante memorizza nello stack qualsiasi registro argomento (\$a0-\$a3) o registri temporaneo (\$t0-\$t9) di cui avrà bisogno dopo la chiamata;
- Il programma chiamato invece salva nello stack il registro di ritorno \$ra e gli altri registri che utilizza (\$s0-\$s7).

Esempio: programma che calcola il fattoriale di un numero n.

```
int fattoriale (int n)
{
    if (n<1) return (1);
    else return (n*fact(n-1));
}
```

Il parametro n corrisponde al registro argomento \$a0.

Il programma innanzitutto salva nello stack due registri: l'indirizzo di ritorno e \$a0. Si aggiorna lo stack per fare posto a 2 elementi e si salvano l'indirizzo di ritorno e il parametro n.

```
Fattoriale:    addi $sp, $sp, -8
               sw   $ra, 4($sp)
               sw   $a0, 0($sp)
```

La prima volta che la procedura fattoriale viene chiamata, l'istruzione `sw` salva un indirizzo nel programma che l'ha chiamata. Le due istruzioni successive verificano se n è minore di 1, saltando a L1 se $n \geq 1$


```

    slti $t0, $a0, 1
    beq $t0, $zero, L1

```

Se $n < 1$, fattoriale restituisce il valore 1 mettendolo in un registro valore. Quindi ripristina dallo stack i due valori salvati e salta all'indirizzo di ritorno.

```

    addi $v0, $zero, 1
    addi $sp, $sp, 8
    jr $ra

```

prima dell'aggiornamento della stack pointer si sarebbero dovuti ripristinare \$a0 e \$ra, ma dato che quando n è minore di 1 non cambiano, è possibile non farlo.

Se n non è minore di 1 si salta a L1: il parametro n viene decrementato di 1 e viene nuovamente chiamata la procedura Fattoriale passandole tale valore.

```

L1:  addi $sa, $sa, -1
     jal Fattoriale

```

l'istruzione successiva è quella a cui la procedura Fattoriale ritornerà; il vecchio indirizzo di ritorno e il vecchio parametro sono ripristinati, oltre ad eseguire l'aggiornamento dello stack pointer:

```

    lw  $a0, 0($sp)      # ind. = L1+8
    lw  $ra, 4($sp)
    addi $sp, $sp, 8

```

Successivamente nel valore \$v0 viene memorizzato il prodotto del valore corrente per il vecchio parametro \$a0; qui si suppone l'esistenza di un'operazione di moltiplicazione, analizzata più avanti. Infine la procedura fattoriale salta nuovamente all'indirizzo di ritorno.

```

    mul $v0, $a0, $v0
    jr  $ra

```

Gestione dello stack

Al 1° richiamo salva nello stack:

- 1) l'indirizzo di ritorno che è nella zona del chiamante
(nome attribuito JALM + 4);
- 2) il valore di \$a0 = n .

Al 2° richiamo salva nello stack:

- 1) l'indirizzo della procedura Fattoriale (indicato da L1+8);
- 2) il valore di \$a0 = $n-1$.

Al 3° richiamo salva nello stack L1+8 e \$a0 = $n-2$.

.....

Al n -mo richiamo salva nello stack L1+8 e \$a0 = 0.

Esempi di esecuzione al variare di n :

$n=0$

$\$ra = JALM+4$
$\$a0 = n = 0$

$n=1$

$\$ra = JALM+4$	} 1^a esecuzione
$\$a0 = n = 1$	
$\$ra = L1+8$	} 2^a esecuzione
$\$a0 = n-1 = 0$	

Alla prima iterazione salta a L1;
\$a0=0;
\$ra=L1+8.

Alla seconda iterazione non salta a L1: ritorna a L1+8, dove \$a0=1; ra=JALM+4;
\$v0*1=\$v0 e ritorna al main.

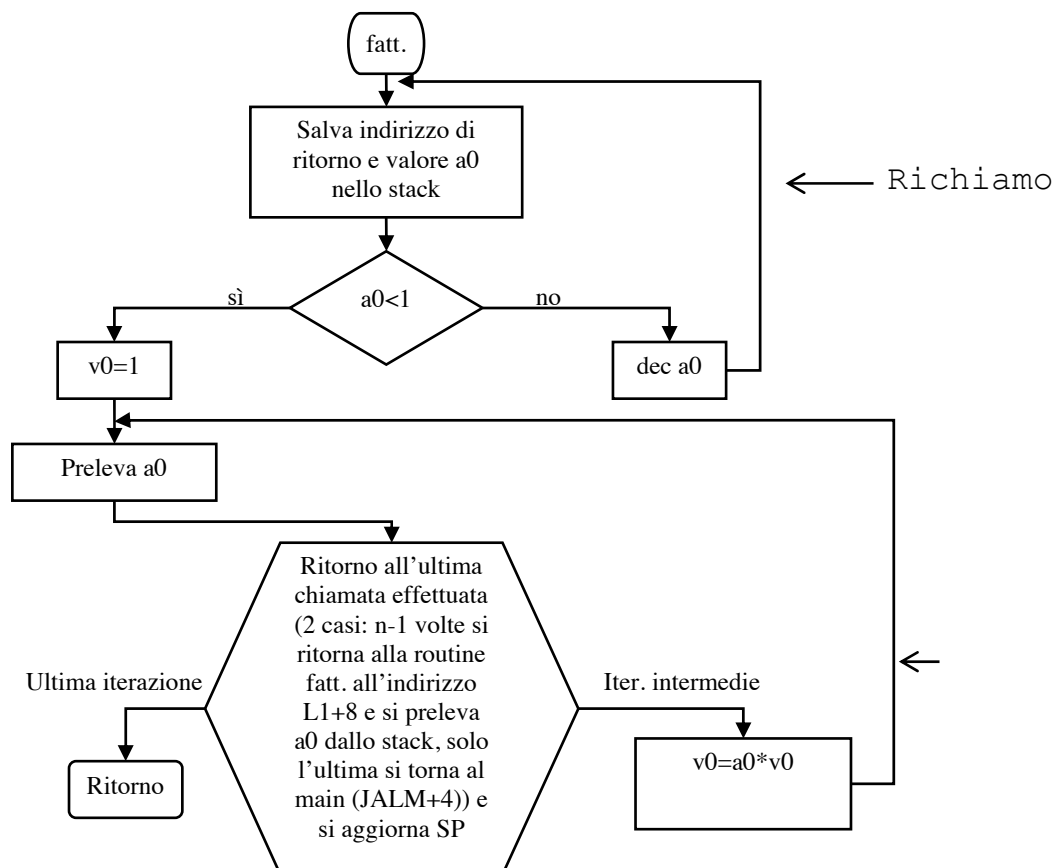
n=2

\$ra = JALM+4	1^a esecuzione
\$a0 = n = 2	
\$ra = L1+8	2^a esecuzione
\$a0 = n-1 = 1	
\$ra = L1+8	3^a esecuzione
\$a0 = n-2 = 0	

Alla 1^iterazione salta a L1; a0 diventa 1; ra=L1+8;

Alla 2^iterazione salta a L1; a0 diventa 0; ra=l1+8;

Alla 3^ iterazione: non salta a L1, quindi v0=1 e torna a L1+8, a0=1; ra=L1+8; v0*1=v0;
torna a L1+8, a0=2, ra=JALM+4, v0=1*a0=2 e torna al main program.



```

Fattoriale:      addi $sp, $sp, -8
                  sw  $ra, 4($sp)
                  sw  $a0, 0($sp)
                  slti $t0, $a0, 1
                  beq $t0, $zero, L1
                  addi $v0, $zero, 1
                  addi $sp, $sp, 8
                  jr  $ra
L1:              addi $sa, $sa, -1
                  jal Fattoriale
                  lw  $a0, 0($sp)      # ind. = L1+8
                  lw  $ra, 4($sp)
                  addi $sp, $sp, 8
                  mul $v0, $a0, $v0
                  jr  $ra

```

Commenti ai programmi per il calcolo del fattoriale:

Limite massimo di cui possiamo calcolare il fattoriale?

Il numero massimo di cui possiamo calcolare il fattoriale è limitato dal valore stesso che il fattoriale assume, in quanto questo risultato deve essere contenuto in un registro: non deve superare i 32 bit!!!

Cosa accade se il numero che passiamo al sottoprogramma è maggiore del numero massimo di cui vogliamo calcolare il fattoriale, cioè cosa accade se $n! > 2^{32} - 1$? Il risultato che viene restituito sono i soli 32 bit meno significativi del $n!$ calcolato.

Nel fare la moltiplicazione, il processore utilizza 2 registri: Hi, in cui mette i 32 bit più alti, più significativi del numero ($n!$) e Lo, in cui mette i 32 bit meno significativi (RICORDA: moltiplicare 2 numero da 32 bit = numero da 64 bit!).

Come si modifica il programma per controllare a tempo di esecuzione se ho superato il numero massimo di cui posso calcolare il fattoriale? Se il registro Hi contiene un numero diverso da 0, significa che il risultato del fattoriale supera i 32 bit a disposizione.

Lezione del 5/11/07

GESTI ONE CARATTERI

La maggior parte dei calcolatori utilizza 8 bit per rappresentare i caratteri e la codifica ASCII è quella più utilizzata. L'elaborazione di testi è talmente diffusa che il MIPS offre istruzioni apposite per trasferire i byte: **load byte** (**lb**) prende un byte dalla memoria, mettendolo negli 8 bit meno significativi di un registro (quelli a destra), mentre **store byte** (**sb**) prende il bit corrispondente agli 8 bit meno significativi di un registro e lo mette in memoria.

Esempio: programma che copia la stringa y nella stringa x, usando il byte Null come carattere di fine stringa.

In realtà ci sono 3 modi per capire se una stringa è finita:

1. il primo valore contenuto nella stringa di caratteri ne codifica la lunghezza;
2. c'è una variabile ulteriore, oltre alla stringa, che è la lunghezza della stringa stessa;
3. le stringhe sono terminate dal carattere Null (ATTENZIONE: null è diverso da 0, ma null in codice ASCII è rappresentato da 000, mentre 0=048).

```

void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != 0) /* copia e test byte */
        i = i + 1;
}

```

Si supponga che gli indirizzi di base dei vettori siano: x=\$a0; y=\$a1; mentre i=\$s0.

```

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 vai a L1
           lw   $s0, 0($sp)           # ripristina $s0 dallo stack
           addi $sp, $sp, 4
           jr   $ra                   # ritorno

```

La procedura aggiorna lo stack pointer e poi salva il registro \$s0 nello stack:

```

addi $sp, $sp, -4
sw   $s0, 0($sp)

```

per inizializzare i=0, l'istruzione successiva pone \$s0 a 0, sommando 0 a 0 e mettendo la somma in \$s0.

```

add  $s0, $zero, $zero

```

Inizia il ciclo. L'indirizzo y[i] è creato sommando i a y[] e ponendo il risultato in \$t1:

```

add  $t1, $a1, $s0

```

Non è necessario moltiplicare i per 4, dato che y è un vettore di byte e non di word: non è necessario rispettare il vincolo di allineamento! Per caricare il carattere presente in y[i] si usa l'istruzione load byte che mette il carattere in \$t2:

```

lb   $t2, 0($t1)

```

con un calcolo analogo mettiamo l'indirizzo di x[i] in \$t3 e il carattere che si trova in \$t2 viene scritto nella locazione di memoria così individuata:

```

add  $t3, $a0, $s0
sb   $t2, 0($t3)

```

incrementiamo i e controlliamo se il carattere di fine stringa è Null, cioè se il carattere è 0:

```

addi $s0, $s0, 1
bne  $t2, $zero, L1

```

se la stringa è terminata, vengono ripristinati i valori di \$s0 e dello stack pointer e si esce dalla procedura.

```

lw   $s0, 0($sp)
addi $sp, $sp, 4
jr   $ra

```

Nota: se invece del registro \$s0, veniva usato un registro t, ad esempio \$t4, non erano necessarie le operazioni di push e pop.

OPERANDI IMMEDIATI.

Molto spesso accade che si necessiti, all'interno di un programma di piccole costanti:

```
addi $sp, $sp, 4
```

Possibili soluzioni:

- ☆ mettere in memoria le costanti tipiche e caricarle da questa → soluzione troppo lenta!
- ☆ Creare registri preservati per le costanti, come il \$zero → nel MIPS ho a disposizione solo 32 registri: sono troppo pochi per memorizzare dati, costanti, valori...!
- ☆ Modifichiamo le istruzioni di `add`, `slt`, `and`, `or`...e le facciamo diventare `addi` = add with immediate...

PRINCIPIO DI PROGETTO 3: *rendi il caso più comune il più veloce possibile.*

Le operazioni su costante avvengono molto di frequente e includendo le costanti all'interno delle operazioni aritmetiche queste ultime risultano molto più veloci rispetto a quando le costanti sono caricate dalla memoria.

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

Dimensioni delle costanti:

```
addi $29, $29, 4
```

Il formato utilizzato è il formato `I`: ho a disposizione al massimo 16 bit. Se la costante è lunga al massimo 16 bit, quando la ALU fa la somma, somma un numero di 32 bit a uno di 16: i 16 bit mancanti?

Faccio l'operazione dell'**estensione del segno**: poiché per la somma la ALU lavora con numeri in complemento a 2, quando voglio trasformare un numero da 16 bit a 32 bit, copio ripetutamente nei 16 bit mancanti il bit di segno

- ☆ Posso settarli a **0** se la **costante è positiva**;
- ☆ Se la costante è negativa? un numero in complemento a 2 negativo ha il bit più significativo uguale a 1: se settassi anche questi 16 bit a 0, sommerei un numero positivo (esempio: `addi $29, $29, -1`. il numero `-1=1111111111111111`. Se aggiungo 16 zeri: `-1=0000000000000000 1111111111111111`. Questo numero è 65536, e non -1!!!). Se il **numero è negativo** aggiungo 16 bit identicamente uguali a **1**.

L'estensione del segno è usata anche nelle istruzioni di `load` e di `store`, di `bne` e di `beq`, di `slti`.

`Andi` e `ori`, invece non estendono il segno, ma riempiono i 16 bit mancanti di 0: non sono numeri in complemento a due, ma sequenze di bit.

Gestione delle grandi costanti.

Che cosa accade se ho bisogno di costanti grandi? Se la costante è più grande di 16 bit? Sebbene, infatti, le costanti siano molto spesso piccole e trovino spazio all'interno del campo di 16 bit a loro assegnato, qualche volta sono più grandi. L'insieme di istruzioni MIPS include l'istruzione *load upper immediate*, `lui`, per caricare i 16 bit più significativi di una costante, nella parte alta di un registro, consentendo a un'istruzione successiva di specificare la parte bassa della costante. Questa istruzione successiva è `ori`: nei 16 bit più alti è identicamente uguale a zero: infatti $0+x=x$, mentre nei 16 bit più bassi è uguale alla parte bassa della costante da caricare.

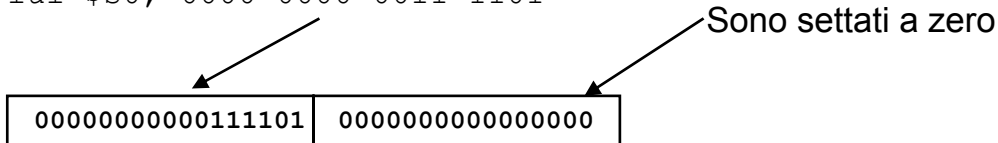
Esempio: devo caricare la costante seguente nel registro \$s0:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

```
lui $s0, 0000 0000 0011 1101
```

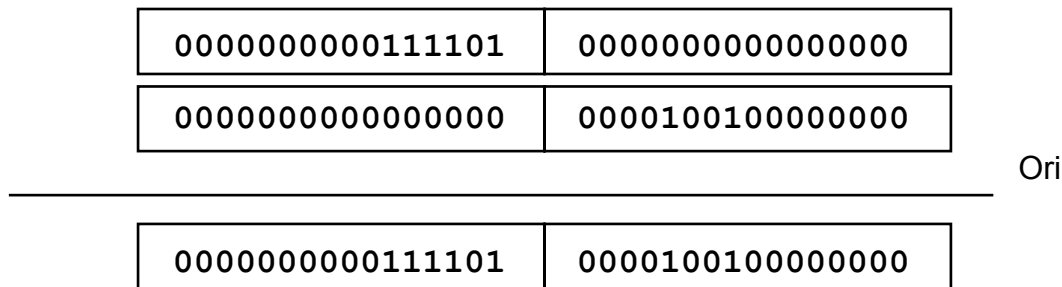
a questo punto il registro \$s0 contiene:

```
lui $s0, 0000 0000 0011 1101
```



Ora dobbiamo sommare i 16 bit meno significativi, il cui valore è 0000 1001 0000 0000

```
Ori $s0, $s0, 0000 1001 0000 0000
```



Il valore finale nel registro \$s0 è il valore desiderato.

In realtà , invece di scrivere le due istruzioni di `lui` e `ori`, l'assemblatore accetta la pseudoistruzione `li` (*load immediate*).

Ad esempio: 3 casi possibili:

`li $t0, 0x1124ABF0` = $\begin{cases} \text{lui } \$t0, 0x1124 \\ \text{ori } \$t0, \$t0, 0xABF0 \end{cases}$

`li $t0, 25.000` = $\begin{cases} \text{----- perché 25.000 è rappresentabile con 16 bit} \\ \text{ori } \$t0, \$t0, 25.000 \end{cases}$

`li $t0, 65.536` = $\begin{cases} \text{lui } \$t0, 1 & \text{perché } 65.536 = 0x0001\ 0000 \\ \text{-----} \end{cases}$

Può capitare che all'interno del programma ho bisogno di caricare un indirizzo di memoria, che è lungo 32 bit. Lo posso fare con l'istruzione `la` (*load address*). `msg` sia il nome dell'etichetta a cui l'assemblatore associa l'indirizzo di memoria.

```
la $t0, msg
```

=carica in `t0` l'indirizzo associato all'etichetta `msg`. L'indirizzo di memoria identificato con `msg`, può essere uno dei 3 casi sopra descritti: l'assemblatore capisce in quale caso siamo e si comporta di conseguenza.

Lezione del 7/11/07.

LI NGUAGGI O ASSEMBLATI VO CONTRO LI NGUAGGI O MACCHI NA:

- il linguaggio assemblativo offre la possibilità di utilizzare una rappresentazione simbolica; questo risulta più semplice rispetto a scrivere sequenze di numeri. Tuttavia, c'è una sintassi da rispettare: ad esempio nelle istruzioni aritmetiche il primo registro è la destinazione;

- è importante ricordare che al di sotto del linguaggio assemblativo c'è il linguaggio macchina;
- il linguaggio assemblativo consente di scrivere pseudoistruzioni, significative per il programmatore, che successivamente l'assemblatore traduce in sequenze di istruzioni interne al MIPS.
- Tuttavia quando è necessario considerare le prestazioni del processore (tempo di esecuzione, spazio di memoria occupato), bisogna tenere conto delle istruzioni reali, non delle pseudoistruzioni.

RI ASSUNTO 1:

panorami ca sul MIPS:

- le istruzioni del MIPS sono tutte lunghe 32 bit: questo deriva dalla logica RISC, cioè dei calcolatori a set di istruzioni ridotte. Ciò poiché si vuole perseguire l'obiettivo, non tanto della semplicità dell'hardware, quanto dell'efficienza, intesa come tempi di risposta quanto più brevi possibili e minima occupazione di memoria.
- Esistono solo 3 formati di istruzioni diversi: I, R, J. In tutti e 3 il primo campo rappresenta il codice operativo, in modo da poter capire subito di quale istruzione e quindi di quale formato si tratta.
- Grazie anche al compilatore riusciamo di ottenere buone prestazioni.

op	rs	rt	rd	shamt	funct	R
op	rs	rt	16 bit address			I
op	26 bit address					J

MODALITÀ DI INDIRIZZAMENTO.

Indirizzamento nei salti.

- bne \$t4,\$t5,Label → la prossima istruzione è a Label se \$t4 è diverso da \$t5
 beq \$t4,\$t5,Label → la prossima istruzione è a Label se \$t4 è uguale a \$t5
 j Label → la prossima istruzione è a Label.

Il modo di indirizzamento più semplice è quello dell'istruzione **jump**:

- utilizza il formato J, in cui 6 bit specificano il codice operativo e i restanti bit sono il campo indirizzo. Tuttavia 26 bit non bastano ad identificare un indirizzo:
 - in realtà è come se fossero 28, in quanto questi indirizzi puntano a word e perciò i 2 bit meno significativi sono identicamente 0. *se un binario finisce con 00 è multiplo di 4*
 - Mancano ancora 4 bit per avere un indirizzo intero: per il principio di località degli accessi, la distribuzione degli indirizzi generati durante l'esecuzione di un programma non è casuale: esiste un'elevata probabilità che a partire da un indirizzo ne venga poi generato uno simile, cioè a pochi accessi in memoria. Per questo i 4 bit più significativi rimangono invariati rispetto a quelli del PC all'atto dell'esecuzione di questa istruzione.
 - Così facendo è come se dividessimo la memoria in 16 blocchi, poiché sono 4 i bit che rimangono invariati. Ognuno dei blocchi è costituito da 2^{26} word.
- L'istruzione **j** ha il vincolo di rimanere all'interno del blocco.
- Le istruzioni di jump e di jump-and-link chiamano procedure che possono essere anche molto distanti dal PC corrente. Utilizzano, per questo, la modalità di indirizzamento pseudodirreto.

Istruzioni di salto condizionato, bne, beq:

- utilizza il **formato I**, in quanto le istruzioni di salto condizionato devono specificare 2 operandi oltre all'indirizzo di salto, rispetto all'istruzione di jump.

Queste istruzioni hanno a disposizione solo 16 bit per l'indirizzo di salto. Se gli indirizzi del programma dovessero trovar posto in questo campo di 16 bit, ne seguirebbe che nessun programma può avere dimensioni superiori a 2^{16} . Una soluzione a tale problema, consiste nella specificare un registro il cui valore deve essere sommato all'indirizzo del salto. L'istruzione di salto condizionato dovrebbe quindi effettuare il seguente calcolo:

$$\text{Program Counter} = \text{Registro} + \text{Indirizzo di salto.}$$

La somma consentirebbe al programma di raggiungere una dimensione pari a 2^{32} , tuttavia quale registro si può utilizzare? La risposta deriva dal contesto in cui i salti condizionati vengono utilizzati: essi si trovano tipicamente nei cicli e nei costrutti di tipo *if*, quindi hanno la tendenza ad eseguire salti a istruzioni vicine (inferiore a 16 istruzioni). Dal momento che il PC contiene l'indirizzo dell'istruzione corrente, si può saltare fino a una distanza di $\pm 2^{15}$ istruzioni rispetto a quella in esecuzione, se si usa il C come registro da sommare all'indirizzo di salto. Quasi tutti i cicli e i costrutti *if* hanno dimensione inferiore alle 2^{16} word, quindi il PC è la scelta ideale.

- Nei 16 bit a disposizione collochiamo quindi lo spostamento che vogliamo effettuare rispetto al PC.

Esempio: le istruzioni di *lw* e *beq* (o *bnq*) condividono lo stesso formato: differenze?

lw \$t0, 12 (\$t1)

- › punto di partenza: \$t1
- › lo spostamento, 12, deve essere necessariamente un multiplo di 4 per rispettare il vincolo di allineamento. Se \$t1=1000 vado all'indirizzo 1012.
- › Intervallo di spostamento è: $-2^{15} < N < 2^{15} - 1$ byte o $-2^{13} < N < 2^{13} - 1$ word.

beq \$t0, \$t1, 12

- › punto di partenza: PC
- › lo spostamento 12, non deve essere necessariamente multiplo di 4: nel campo da 16 bit per l'istruzione *beq* specifichiamo di quante istruzioni avanti o indietro rispetto al PC mi voglio muovere, non di quanti byte! Quindi effettivamente non mi sposto di 12, ma di 12×4 , per rispettare il vincolo di allineamento! Se PC=1000 vado all'indirizzo 1048.
- › Intervallo di spostamento è: $-2^{15} < N < 2^{15} - 1$ word o $-2^{17} < N < 2^{17} - 1$ byte.

Modalità di indirizzamento del MIPS:

modalità di indirizzamento = modo in cui l'istruzione riesce a trovare le informazioni di cui ha bisogno.

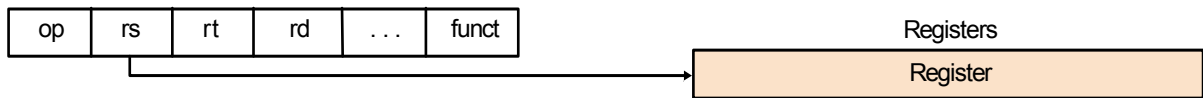
Indirizzamento immediato → in cui l'operando è una costante specificata nell'istruzione. Utilizza il formato I: nei 16 bit riservati all'indirizzo troviamo l'operando di nostro interesse.

1. Immediate addressing

op	rs	rt	Immediate
----	----	----	-----------

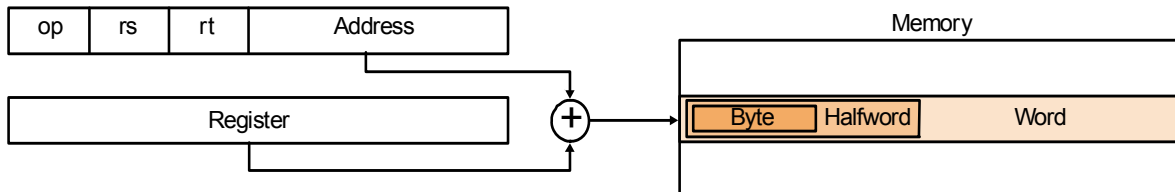
Indirizzamento tramite registro → in cui l'operando è un registro (add, sub).

2. Register addressing



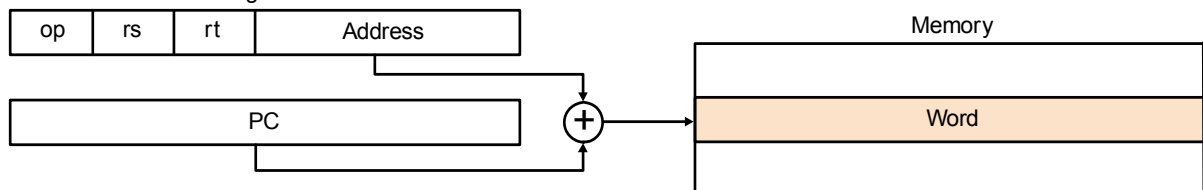
Indirizzamento tramite base → in cui l'operando è in una locazione di memoria individuata dalla somma del contenuto di un registro e di una costante specificata nell'istruzione (le istruzioni di load e store).

3. Base addressing



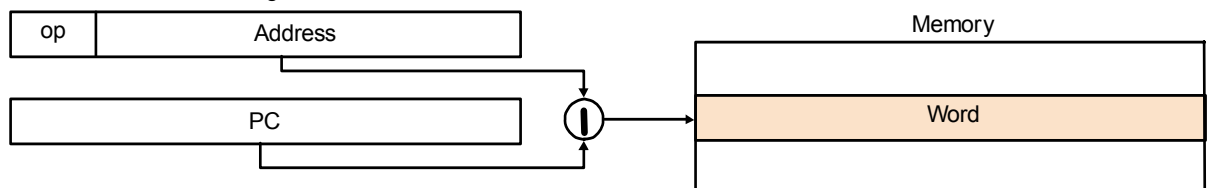
Indirizzamento relativo al Program Counter → in cui l'indirizzamento è la somma del contenuto del Program Counter e della costante nell'istruzione (istruzioni beq, bne).

4. PC-relative addressing



Indirizzamento pseudodiretto → in cui l'indirizzo è ottenuto concatenando i 26 bit dell'istruzione con i 4 bit più significativi del PC. (istruzioni di jump)

5. Pseudodirect addressing



addi → utilizza un indirizzamento a registro per identificare destinazione e primo operando; indirizzamento immediato per identificare la costante.

Esempio:

```
bne $t0, $t1, dopo
```

```
...a
```

```
...b
```

```
...c
```

```
dopo:...
```

se dopo è troppo lontano?

Soluzione:

```

    beq $t0, $t1, poi
    j  dopo
poi: ...a
    ...b
    ...c

```

dopo:...

questa è in realtà un'operazione che fa l'assemblatore se si accorge che `dopo` è troppo lontano. E se anche `j dopo` è troppo lontano?

```

la $t2, dopo (se dopo è un indirizzo di word)
jr $t2

```

PSEUDOISTRUZIONI .

Le pseudoistruzioni sono versioni modificate delle istruzioni vere, trattate dall'assemblatore. Queste istruzioni non devono essere implementate in hardware, ma la loro presenza nel linguaggio assembler semplifica le fasi di traduzione e di programmazione. Le pseudoistruzioni consentono all'assembler MIPS di avere un insieme di istruzioni più ricco di quello implementato in hardware; l'unico costo rappresentato dal registro `$at` riservato all'assemblatore.

Esempi:

Pseudo istruzione: `move $t0, $t1` **# \$t0 = \$t1**

Istruzione vera: `add $t0, $zero, $t1`

Pseudo istruzione: `blt $s1, $s2, Label`

Istruzioni vere: `slt $at, $s1, $s2`
 `bne $at, $zero, Label`

Altri esempi:

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

RIASSUNTO:

MIPS operands		
Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero,	Fast locations for data. In MIPS, data must be in registers to perform
	\$a0-\$a3, \$v0-\$v1, \$gp,	arithmetic. MIPS register \$zero always equals 0. Register \$at is
	\$fp, \$sp, \$ra, \$at	reserved for the assembler to handle large constants.
2 ³⁰ memory words	Memory[0],	Accessed only by data transfer instructions. MIPS uses byte addresses, so
	Memory[4], ...,	sequential words differ by 4. Memory holds data structures, such as arrays,
	Memory[4294967292]	and spilled registers, such as those saved on procedure calls.

32 registri → accesso rapido ai dati. Nel MIPS i dati devono essere presenti nei registri per eseguire operazioni aritmetiche. Il registro \$zero ha sempre valore 0. il registro \$at è riservato all'assemblatore.

230 parole di memoria → accesso solo tramite le istruzioni di trasferimento dati. Il MIPS usa l'indirizzamento a byte, quindi gli indirizzi delle parole consecutive differiscono di 4 unità. La memoria contiene le strutture dati come vettori, registri riversati e i registri salvati durante la chiamata a sottoprogramma.

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$s1, \$s2, 25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1, \$s2, \$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set less than immediate	slti \$s1, \$s2, 100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = PC + 4$; go to 10000	For procedure call

Lezione del 12/11/07

Vettori e puntatori.

Vediamo 2 esempi di procedure che azzerano tutti gli elementi di un vettore.

Procedura *azzer* che utilizza un vettore.

```
azzer (int vett[], int dim)
{
    int i;
    for (i=0; i<dim; i++)
        vett[i] = 0;
}
```

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

```
azzer:      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
```

Innanzitutto inizializziamo l'indice *i* a 0.

```
move $t0, $zero
```

Per porre *vett[i]=0* per prima cosa occorre trovare il suo indirizzo: a questo scopo si moltiplica *i* per 4 per ottenere l'indirizzo in byte:

```
add $t1, $t0, $t0
add $t1, $t1, $t1
```

Dal momento che l'indirizzo di partenza del vettore è in un registro, si deve sommare tale indirizzo all'indice per ottenere *vett[i]*:

```
add $t2, $a0, $t1
```

In \$t2 ora troviamo l'indirizzo di *vett[i]*. Ora si può memorizzare 0 nella locazione trovata:

```
sw $zero, 0($t2)
```

Questa istruzione è alla fine del corpo del ciclo *for*, quindi il passo successivo è incrementare l'indice *i*:

```
addi $t0, $t0, 1
```

Il test del ciclo verifica se *i* è minore della dimensione del vettore:

```
slt $t3, $t0, $a1
bne $t3, $zero, L1
```

Se *i* è minore della dimensione del vettore, salta a L1, altrimenti si ritorna al programma chiamante.

Procedura *azzera* che utilizza i puntatori.

```

azz2 (int *vett, int dim)
{   int *p;
    for (p=&vett[0]; p<&vett[dim]; p++)
        *p = 0;
}

```

L'indirizzo di una variabile è indicato con `&`, mentre l'oggetto puntato da un puntatore è indicato da `*`. Dalle dichiarazioni si vede che sia `vett` sia `p` sono puntatori a interi. La prima parte del ciclo `for` assegna l'indirizzo del primo elemento di `vett` al puntatore `p`. La seconda parte effettua un controllo per vedere se il puntatore ha superato l'ultimo elemento di `vett`. Incrementare di uno il puntatore nell'ultima parte del ciclo `for` significa spostarlo all'elemento successivo di dimensioni pari a quelle dichiarate: dato che `p` è un puntatore a interi, il compilatore genera delle istruzioni MIPS che incrementano `p` di 4 unità. L'assegnazione all'interno del ciclo pone a 0 l'oggetto puntato da `p`.

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

```

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

```

il codice inizia con l'assegnazione dell'indirizzo del primo elemento del vettore al puntatore `p`:

```
move $t0, $a0
```

calcoliamo poi l'indirizzo dell'ultimo elemento del vettore, moltiplicando la dimensione per 4 al fine riottenere il valore in byte; poniamo in `$t2` l'indirizzo dell'ultimo elemento del vettore:

```

add  $t1, $a1, $a1
add  $t1, $t1, $t1
add  $t2, $a0, $t1

```

La parte successiva è relativa al codice del corpo del ciclo `for`, che semplicemente memorizza 0 nella posizione puntata da `p`:

```
sw $zero, 0($t0)
```

L'istruzione successiva deve aggiornare `p` per far sì che punti alla parola successiva. In C incrementare un puntatore di 1 significa farlo puntare all'oggetto successivo in sequenza: dato che `p` è un puntatore a interi, ciascuno dei quali occupa 4 byte, bisogna incrementare `p` di 4 unità:

```
addi $t0, $t0, 4
```

La fase successiva consiste nel test di fine ciclo, che controlla che `p` sia minore dell'ultimo elemento del vettore:

```

slt  $t3, $t0, $t2
bne  $t3, $zero, L2

```

Se $p <$ dell'ultimo elemento del vettore salta a L2, altrimenti torna al main.

Confrontando le due versioni notiamo che:

- › la prima versione deve avere la moltiplicazione e la somma necessariamente all'interno del ciclo, dato che i viene incrementato e quindi l'indirizzo deve essere ricalcolato a partire dal nuovo valore, mentre la versione con i puntatori incrementa direttamente il puntatore p ;
- › la versione con i puntatori riduce da 7 a 4 il numero di istruzioni eseguite per ogni ciclo, ed è quindi più veloce

Architetture alternative

fornire al processore istruzioni più potenti:

persegue l'obiettivo di ridurre il numero di istruzioni per eseguire varie funzioni. Tuttavia nel fare ciò ci sono dei vantaggi, ma anche degli svantaggi: (RISC vs CISC)

vantaggi:

- › linguaggio più compatto e ampio: posso utilizzare una sola istruzione per compiere che svolge determinati compiti invece che costruirmi passo per passo piccole istruzioni che possano alla fine svolgere lo stesso compito;

svantaggi:

- › ci sono tante istruzioni che assolvono tante funzioni diverse: è più difficile la scelta e la memorizzazione per il programmatore;
- › il tempo di ciclo per ogni istruzione è maggiore: ho bisogno di più periodi di clock per portare a termine una singola istruzione. Non bisogna dimenticare che avere tante istruzioni diverse porta all'effetto $n+1$: aggiungere varie istruzioni in più rallenta in realtà, l'esecuzione di tutte le istruzioni, anche quelle fondamentali, in quanto l'hardware ha molta scelta e quindi impiega più tempo ad arrivare all'istruzione giusta. (RISC vs CISC)

Riassunto: i principi di progetto.

Semplicità e regolarità sono strettamente correlate → la ricerca della regolarità è alla base di molte delle caratteristiche dell'insieme delle istruzioni MIPS: stessa dimensione per tutte le istruzioni, sempre 3 operandi di tipo registro nelle istruzioni aritmetiche, campi registro sempre nelle stesse posizioni all'interno di tutti i formati di istruzioni...

Minori sono le dimensioni, maggiore è la velocità → la velocità di esecuzione è il motivo per cui il MIPS ha 32 registri anziché molti di più.

Rendere il caso veloce il più frequente → esempi di questo tipo che riguardano il MIPS includono il modo di indirizzamento relativo al PC per i salti condizionati e l'indirizzamento immediato per gli operandi costanti ...

Un buon progetto richiede buoni compromessi → esempi relativi al MIPS è la scelta di 3 formati di istruzioni anziché 1, il fatto di avere registri dedicati (\$ra, \$zero), il compromesso che permette di specificare indirizzi e costanti grandi pur mantenendo la stessa lunghezza per tutte le istruzioni...

CAPITOLO 3

-L'ARITMETICA DEI CALCOLATORI-

Lezione del 14/11/07.

I bit sono solo bit e non hanno un significato intrinseco. Le convenzioni definiscono le relazioni tra bit e numeri.

I numeri che dobbiamo gestire sono di dimensioni finite, cioè hanno un intervallo di rappresentazione finito: ciò porta al problema dell'**overflow**, cioè del *superamento della capacità di rappresentazione per i numeri in complemento a 2*.

Ci sono diversi modi per rappresentare numeri con il segno:

Modulo e segno	Complemento a 1	Complemento a 2
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

Osservazioni:

- In tutte e 3 le configurazioni, il bit più significativo, cioè quello più a sinistra, è un indicatore del segno del numero: se è 0 il numero è positivo, se è 1 il numero è negativo.
- Nelle configurazioni Modulo e Segno, Complemento a 1, ho due rappresentazioni dello 0, ciò comporta il bilanciamento tra il numero di configurazioni per i numeri negativi e i numeri positivi.
- Nella configurazione Complemento a 2, c'è una sola rappresentazione dello 0: ciò comporta uno sbilanciamento tra numeri positivi e numeri negativi: i numeri negativi vanno da 1 a 3, mentre i numeri negativi da -1 a -4.

➔ Problema: $A+B$ se $A=0$ \Rightarrow posso sempre farlo, non ho problemi!
 $A-B$ se $A=0$ e $B=-4$ $\Rightarrow A+B=4$ non è rappresentabile in Ca2: overflow!!!

-----dal passato-----

Come rappresentare i numeri negativi.

Rappresentazione in Modulo e Segno (anche detta Binario Naturale)

Si utilizza un bit per rappresentare il segno del numero considerato

0 \rightarrow + (numero positivo)

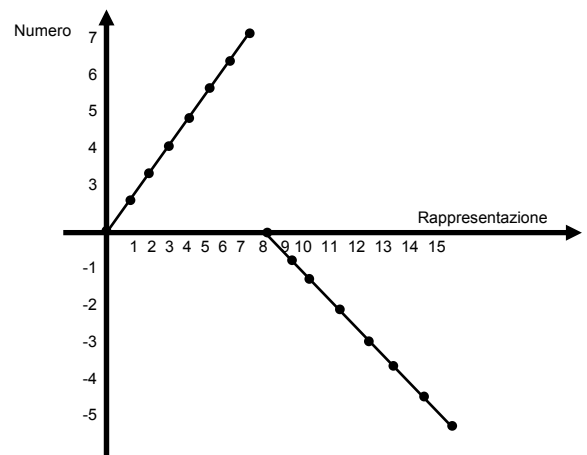
1 \rightarrow - (numero negativo)

Se consideriamo un byte, rimangono ora 7 bit per il modulo del numero: i numeri rappresentabili sono perciò $\pm[0-127]$

+/-	b6	b5	b4	b3	b2	b1	b0
-----	----	----	----	----	----	----	----

Ho 2 problemi:

1. il doppio 0;
2. andamento del grafico prima crescente e poi decrescente: è difficile stabilire quale fra due numeri è maggiore e quale è minore.



Significato in modulo e segno	Rappresentazione binaria	Significato in valore assoluto
+7	0 111	7
+6	0 110	6
+5	0 101	5
+4	0 100	4
+3	0 011	3
+2	0 010	2
+1	0 001	1
+0	0 000	0
-0	1 000	8
-1	1 001	9
-2	1 010	10
-3	1 011	11
-4	1 100	12
-5	1 101	13
-6	1 110	14
-7	1 111	15

Addizione e sottrazione sono le operazioni di cui si deve disporre per poter realizzare qualsiasi operazione aritmetica più complessa. Si supponga che il calcolatore abbia una "Unità Aritmetica" che realizzi indipendentemente le due operazioni. Di fronte ad una somma algebrica, il calcolatore dovrebbe:

- confrontare i due segni.
- se uguali, attivare il circuito di addizione.
- se diversi, identificare il maggiore (in valore assoluto) ed attivare il circuito di sottrazione.
- completare il risultato con il segno corretto.

I passi indicati non sono eseguibili contemporaneamente perché ognuno dipende dai precedenti. In pratica per effettuare somma e sottrazione si ricorre ad un unico circuito, utilizzando un metodo che permette di evitare le operazioni di confronto.

Complemento alla base.

Nella rappresentazione in complemento alla base con n cifre le b^n combinazioni rappresentano numeri positivi e negativi.

In particolare:

- Le combinazioni da $0 < p < b^n/2 - 1$ rappresentano i numeri positivi, rispettando la usuale rappresentazione posizionale;
- Le combinazioni da $b^n/2 < n < b^n - 1$ rappresentano i numeri negativi, con la seguente definizione:

dato un numero positivo X , il suo corrispondente negativo è dato da: $b^n - X$

$b=2, n=5$

Positivi da 0 a 01111

Negativi da 10000 a 11111

$X = 01011$; trovo $-X$

$$\begin{array}{r} 100000 - \\ 01011 = \\ \hline 10101 \end{array}$$

$b=2, n=7$

Positivi da 0 a 0111111

Negativi da 1000000 a 1111111

$X = 0011000$; trovo $-X$

$$\begin{array}{r} 10000000 - \\ 0011000 = \\ \hline 1101000 \end{array}$$

Regola pratica: partendo dal bit meno significativo, si riportano invariati tutti i bit fino al primo bit a 1 compreso; si complementano i rimanenti bit ($0 \rightarrow 1$, $1 \rightarrow 0$). Oppure: complemento ogni singolo bit e aggiungo 1, in quanto il complemento alla base è uguale al complemento alla base meno 1, +1.

Complemento alla base -1.

Nella rappresentazione in complemento alla base -1 con n cifre le b^n combinazioni rappresentano numeri positivi e negativi.

In particolare:

- Le combinazioni da $0 \leq p < b^n/2 - 1$ rappresentano i numeri positivi, rispettando la usuale rappresentazione posizionale;
- Le combinazioni da $b^n/2 \leq n < b^n - 1$ rappresentano i numeri negativi, con la seguente definizione:
dato un numero positivo X , il suo corrispondente negativo è dato da: $(b^n - 1) - X$

$X=36, b=10, n=2$

in complemento alla base -1 è: $99 - 36 = 63$

Si ottiene complementando a 9 ogni singola cifra

$X=01011, b=2, n=5$

$-X$ in complemento alla base -1 è: $(2^5 - 1) - X = (10000 - 1) - X \Rightarrow 1111 - 01011 = 10100$

Regola pratica: si ottiene complementando ogni singolo bit ($0 \rightarrow 1, 1 \rightarrow 0$)

Quindi per la rappresentazione in complemento:

- Rappresentazione in complemento a 2: i numeri positivi sono rappresentati dal loro modulo e hanno il bit più significativo (segno) posto a 0. I numeri negativi sono rappresentati dalla quantità che manca al numero positivo per arrivare alla base elevata al numero di cifre utilizzate, segno compreso. Pertanto i numeri negativi hanno il bit del segno sempre a 1.
- Metà delle configurazioni sono perciò riservate ai numeri positivi e metà ai numeri negativi.
- Discorsi analoghi possono essere fatti per basi diverse da 2: in base 10 un numero è negativo se la prima cifra è ≥ 5 , in base 8 se ≥ 4 , in base 16 se ≥ 8 .

Per la rappresentazione in complemento alla base:

Con n bit a disposizione:

- Il numero minimo rappresentabile è -2^{n-1}
- Il numero massimo rappresentabile è $2^{n-1} - 1$.
- -1 è rappresentato da tutti 1 qualunque sia il numero di bit considerato.
- Il numero può essere interpretato considerando il bit più significativo con segno negativo.

Utilità del complemento alla base:

Con la tecnica del complemento si può utilizzare un solo circuito per effettuare sia l'addizione, sia la sottrazione.

Operiamo in base 10 e vogliamo calcolare $A - B$. Si supponga di conoscere il risultato dell'operazione $10 - B$ (complemento a 10 di B). Allora:

$$A - B = A + (10 - B) \text{ a condizione che si trascuri il riporto}$$

Analogo discorso con k cifre purché si disponga del risultato dell'operazione $10^k - B$ (complemento a 10^k). Si ricordi sempre di fissare il numero di cifre.

Se operiamo in base 2, con k cifre:

$$A - B = A + (2^k - B) \text{ a condizione che si trascuri il riporto.}$$

Se si utilizza la tecnica del complemento alla base -1 occorre sommare il riporto al risultato finale.

esempi:

Complemento a 2

$$\begin{array}{r} 19 + (-17) \\ 010011 \\ \underline{101111} \\ 1000010 \text{ (+2)} \end{array}$$

Complemento a 1

$$\begin{array}{r} 19 + (-17) \\ 010011 \\ \underline{101110} \\ 000001 \\ \underline{1} \\ 000010 \text{ (+2)} \end{array}$$

Complemento a 2

$$\begin{array}{r} (-17) + (-2) \\ 101111 \\ \underline{111110} \\ 1101101 \text{ (-19)} \end{array}$$

Complemento a 1

$$\begin{array}{r} (-17) + (-2) \\ 101110 \\ \underline{111101} \\ 1101011 \\ \underline{1} \\ 101100 \text{ (-19)} \end{array}$$

Problemi:

somma di due numeri positivi con 6 cifre binarie.

$$\begin{array}{r} 19 + 17 \\ 010011 + \\ \underline{010001} = \\ 100100 \text{ (-28)} \end{array}$$

Si sono sommati due numeri positivi e si è ottenuto un numero negativo. Il risultato corretto (+36) non è rappresentabile in complemento a 2 con 6 bit (massimo numero rappresentabile = +31). Il fenomeno si chiama traboccamento o **overflow**.

Complemento a 2

$$\begin{array}{r} (-19) + (-17) \\ 101101 + \\ \underline{101111} = \\ 1011100 \text{ (28)} \end{array}$$

Si sono sommati due numeri negativi e si è ottenuto un numero positivo. Il risultato corretto (-36) non è rappresentabile in complemento a 2 con 6 bit (minimo numero rappresentabile = -32 in C.a 2 e -31 in C. a 1). Il fenomeno si chiama traboccamento o **overflow**.

$$\begin{array}{r} 01111000 + \\ \underline{01101001} = \\ 11100001 \end{array}$$

Complemento a 2

$$\begin{array}{r} + 120 + \\ + 105 = \\ - 31 \end{array}$$

Overflow=1 (si)

Valore Assoluto

$$\begin{array}{r} 120 + \\ 105 = \\ 225 \end{array}$$

Riporto=0 (no)

```

1 1 1 1 1 0 1 1 +
1 1 1 1 0 0 0 0 =
1 1 1 0 1 0 1 1

```

Complemento a 2:

- 5 +
- 16 =
- 21

Overflow=0

Valore assoluto:

251 +
240 =
235

Riporto=1

Somma di numeri in complemento a 2.

$0 \leq A \leq 2^{N-1} - 1$ $0 \leq B \leq 2^{N-1} - 1$	$-2^{N-1} \leq A < 0$ $0 \leq B \leq 2^{N-1} - 1$	$0 \leq A \leq 2^{N-1} - 1$ $-2^{N-1} \leq B < 0$	$-2^{N-1} \leq A < 0$ $-2^{N-1} \leq B < 0$
$0 \leq S \leq 2^N - 2$	$-2^{N-1} \leq S < 2^{N-1} - 1$	$-2^{N-1} \leq S < 2^{N-1} - 1$	$-2^N \leq S < 0$
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.

32 bit signed numbers:

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = + 1_{ten}

0000 0000 0000 0000 0000 0000 0000 0010_{two} = + 2_{ten}

...

0111 1111 1111 1111 1111 1111 1111 1110 = + 2,147,483,646

0111 1111 1111 1111 1111 1111 1111 1111 = + 2,147,483,647

1000 0000 0000 0000 0000 0000 0000 0000 = - 2,147,483,648

1000 0000 0000 0000 0000 0000 0000 0001 = - 2,147,483,647

1000 0000 0000 0000 0000 0000 0000 0010 = - 2,147,483,646...

...

1111 1111 1111 1111 1111 1111 1111 1101 = - 3

1111 1111 1111 1111 1111 1111 1111 1110 = - 2

1111 1111 1111 1111 1111 1111 1111 1111 = - 1

Siano:

\$t0=0000 0000 0000 0000 0000 0000 0000 0000 = 0

\$t1=1111 1111 1111 1111 1111 1111 1111 1111 = - 1

slt \$t2, \$t0, \$t1 Quanto vale \$t2?

- Se i numeri sono pensati in complemento a 2 \$t2 vale 0: \$t0=0 non è minore di \$t1=-1!
- Può capitare, tuttavia, che al programmatore interessi lavorare con degli indirizzi: siano ora \$t0 e \$t1 due registri che contengono indirizzi di memoria. In questo caso \$t0<\$t1, poiché i numeri sono pensati in valore assoluto, in quanto non ha senso prevedere degli indirizzi di memoria negativi!

È necessario utilizzare quindi 2 istruzioni diverse:

`slt` → lavora con numeri in complemento a 2.

Quindi `slt $t2, $t0, $t1`
`$t2=0.`

`sltu` → *set on less than unsigned*, lavora con numeri in valore assoluto.

Quindi `slt $t2, $t0, $t1`
`$t2=1.`

Siano:

`add $t2, $t0, $t1`

`$t0= 0 1 1 1 1 0 0 0 +`

`$t2= 0 1 1 0 1 0 0 1`

`0 1 1 1 1 0 0 0 +`

`0 1 1 0 1 0 0 1 =`

`1 1 1 0 0 0 0 1`

`$t2= 1 1 1 0 0 0 0 1`

- Se i numeri sono in complemento a 2 si genera l'eccezione di overflow, in quanto sommando due numeri positivi, il risultato è un numero negativo!
- Se i numeri sono in valore assoluto, per esempio se rappresentano i pesi di due oggetti, l'operazione dà un risultato corretto e quindi bisogna ignorare l'overflow.

Si pone ancora la necessità di 2 operazioni:

`add` → lavora con i numeri in complemento a 2 e causano eccezioni in caso di overflow.

`addu` → lavora con i numeri in valore assoluto, cioè senza segno e non causano eccezioni di overflow.

Estensione del segno.

È necessario quando si converte un numero binario rappresentato su n bit, in un numero rappresentato su più di n bit, ad esempio quando si deve convertire un numero da 16 bit in un numero da 32 bit. Si consideri, per esempio, il campo immediato delle istruzioni di *load*, *store*, *branch*, *add* e *set on less than*, che contiene un numero di 16 bit in complemento a 2, e che può quindi rappresentare numeri: $2^{15} < N < 2^{15} - 1$, cioè $-32.768 < N < 32.767$.

Per sommare il campo immediato ad un registro da 32 bit, l'elaboratore deve convertire il numero di 16 bit in un numero da 32 bit. La tecnica consiste nel prendere il bit di segno e nel replicarlo, in maniera da riempire i 16 bit mancanti più significativi. Ciò funziona, in quanto i numeri positivi in Ca2 possiedono effettivamente un numero infinito di 0 a sinistra e quelli negativi un numero infinito di 1 a destra. La combinazione di bit che rappresenta il numero effettivo infatti, tronca questi infiniti bit per poter rientrare nell'ampiezza della parola resa disponibile dall'hardware.

`0010` → `0000 0010`

`1010` → `1111 1010`

`lb $t0, 0($t1)`

= vai in memoria e carica il byte che si trova all'indirizzo di memoria contenuto in `$t1` senza spostamento, in `$t0`.

Eseguendo questo comando dalla memoria arrivano 8 bit, cioè i byte. Tuttavia `$t0` ha 32 byte a disposizione e i 24 bit che non arrivano dalla memoria non possono non essere pilotati. Questi 24 bit mancanti vanno gestiti in maniera corretta:

- Siano gli 8 bit che arrivano dalla memoria, la configurazione che rappresenta la lettera "a" in codice ASCII: i 24 bit mancanti possono assumere valore identicamente uguale a 0, in quanto nel codice ASCII non ci sono configurazioni di bit negative!

- Siano gli 8 bit che arrivano dalla memoria, la configurazione che rappresenta il valore -1 in complemento a 2: i 24 bit mancanti devono assumere il valore identicamente uguale a 1, cioè si deve estendere il segno, in quanto altrimenti verrebbe modificato il valore della costante che ho caricato dalla memoria!

Si rendono necessarie 2 istruzioni:

l_b → se si utilizzano numeri in Ca2: è quindi un'istruzione che estende il segno.

l_{b_u} → se si utilizzano numeri in valore assoluto (o codifiche di stringhe alfanumeriche): i 24 bit mancanti assumono identicamente valore 0.

►SECONDA PARTE◄

PROBLEMA DELL'OVERFLOW

Abbiamo bisogno di un indicatore che ci indichi in quale caso si verifica overflow. Poiché nel MIPS la dimensione della word è abbastanza elevata, non si verifica molto spesso il caso in cui la somma di due numeri dia l'overflow. Per questo non è vantaggioso chiedersi ogni volta se si è verificato overflow dopo una somma! Nel MIPS dopo aver eseguito una somma, non mi chiedo ogni volta se si verifica l'overflow, ma quando esso si verifica, si genera un'interruzione, cioè un evento esterno al processore. Al seguito di questo interrupt esterno il programma evolve verso una parte di codice predefinita in memoria di correzione con l'obbligo, naturalmente, di salvare l'indirizzo di rientro al programma.

Solitamente quando si salta ad una subroutine si salva l'indirizzo di rientro nel \$ra. Posso anche in questo caso salvare l'indirizzo di rientro al programma dopo che l'eccezione di overflow è stata gestita, nel \$ra? Non si può, in quanto non possiamo prevedere in che porzione di codice si genera l'overflow: se questo dovesse verificarsi all'interno di un sottoprogramma, salvando l'indirizzo di rientro al sottoprogramma dalla porzione di codice che gestisce l'overflow, sovrascrivo e quindi cancello, l'indirizzo di rientro al main, cioè l'indirizzo dell'istruzione successiva a quella che ha effettuato la chiamata al sottoprogramma. Per questo c'è una sorta di Programm Counter ulteriore che prende il nome di **Exception Programm Counter, EPC**, nel quale viene memorizzato l'indirizzo a cui dobbiamo tornare dopo che un'eccezione è stata gestita. Tale registro memorizza dunque l'indirizzo che ha provocato l'eccezione.

Per ritornare all'istruzione che ha generato l'eccezione posso scrivere: `jr $epc`? C'è un problema: come parametro dell'istruzione jr bisogna specificare un registro del MIPS: EPC non è uno dei 32 registri del MIPS! C'è un'istruzione apposita: **move from control 0, mfc0** seguita da uno dei 32 registri del MIPS e poi dal registro EPC. Tale istruzione copia il contenuto del registro EPC in uno dei registri general-purpose del MIPS, in maniera tale che per ritornare all'istruzione che ha causato l'eccezione, posso utilizzare un'istruzione di salto a registro.

Si pone un altro problema: non sono sicuro che t0 sia già in uso dal programma: per questo esistono 2 registri dedicati al sistema operativo, k1 e k2, che non vengono utilizzati dal programma.

```
mfc0 $k1, $epc
jr $k1
```

Il vantaggio che deriva da questa procedura è che non spreco tempo a chiedermi se l'overflow si verifica ogni volta che compio un'operazione che potrebbe generarlo. Quando in effetti l'overflow si verifica, generando l'eccezione, ho una maggiore perdita di tempo piuttosto che nel caso l'overflow venisse gestito all'interno del programma: tuttavia questo succede di rado, in quanto una word ha dimensioni abbastanza elevate.

Problema dell'overflow:

non si verifica mai overflow in una somma, addizionando un numero positivo ed uno negativo.

non si verifica mai overflow in una sottrazione, quando i segni sono gli stessi.

L'overflow può verificarsi quando:

- sommando 2 positivi ottengo un negativo;
- sommando 2 negativi ottengo in positivo;
- sottraendo un negativo da un positivo ottengo un negativo;
- sottraendo un positivo da un negativo ottengo un positivo;

Considerando le operazioni: $A+B$ e $A-B$:

- l'overflow si può verificare se $B=0$? No.

- l'overflow si può verificare se $A=0$? $0+B=B$ non pone problemi. $0-B=$ in Ca2 ho un negativo che non ha corrispondente positivo! Immaginiamo il caso in cui $B=-128$. $0-(-128)=128$ Overflow! Perché $+128$ in Ca2 non è rappresentabile!

Bisogna ricordare che non sempre vogliamo che l'overflow sia gestito e prima ancora rivelato: se lavoriamo con numero unsigned, sto considerando i numeri in valore assoluto (possono generare il problema del riporto)

`addi` =somma una costante. Se devo sommare come costante uno spostamento relativo all'indirizzo di memoria in cui mi trovo non ha senso considerare numero con segno. Se quindi mi trovo all'indirizzo 1111 e voglio sommare 1, non voglio che si generi l'eccezione di overflow → `addiu`. (anche questa estende il segno).

`sltiu` = confronta numeri senza segno (però anche questa estende il segno!).

OPERAZIONI LOGICHE.

Talvolta è necessario lavorare con singoli bit e non con numeri o costanti formati da 32 bit. Esistono per questo le operazioni di:

shifts:

shift left logical: `sll $t2, $s0, 8`.

In questo caso trasla a sinistra della quantità specificata nel campo di shift amount. L'istruzione di shift utilizza il formato R:

op rs rt rd shamt funct

000000	00000	10000	01010	01000	000000
--------	-------	-------	-------	--------------	--------

AND bit a bit: applico l'operatore AND al contenuto di 2 registri: `AND $t0, $t1, $t2`

`$t1= 00003C00` and

`$t2= 00000D00`

`$t0= 00000C00`

OR bit a bit: applico l'operatore OR al contenuto di 2 registri: `OR $t0, $t1, $t2`

`$t1= 00003C00` or

`$t2= 00000D00`

`$t0= 00003D00`

Si possono applicare se si ha interesse a svolgere operazioni booleane, ma anche se voglio mascherare una parte di bit di una word: se voglio mascherare dei bit tramite l'and lo faccio con lo 0, tramite l'or con l'1. L'exor serve come invertitore comandato.

Ricordiamo che le istruzioni logiche non estendono il segno, in quanto rappresentano solo sequenze di bit e non numeri con un significato intrinseco.

►APPENDICE A◄

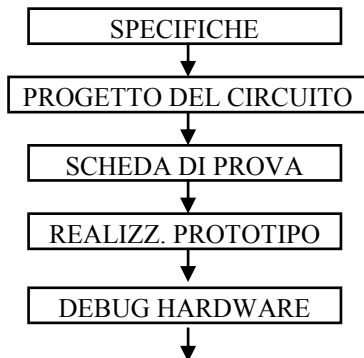
►SIMULATORE SPIM◄

Per risolvere un problema reale devo seguire 2 soluzioni parallelamente:

- la soluzione hardware;
- la soluzione software.

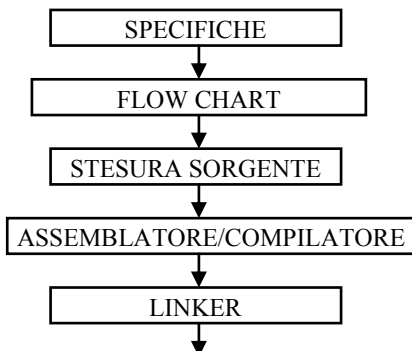
Una volta implementate entrambe le soluzioni devo integrare il sistema, cioè devo riunirle in modo efficace.

SVILUPPO HARDWARE.



Date le specifiche del problema devo progettare il circuito, costruire una scheda di prova e realizzare il prototipo. Per verificare che il sistema funzioni devo effettuare un debug dell'hardware. Questo ultimo passo di collaudo non può essere fatto solo e direttamente sulla scheda, a causa della complessità del progetto, del grande numero di componenti e di piste.

SVILUPPO SOFTWARE.



Date le specifiche del problema, decido che linguaggio di programmazione utilizzo. Ad esempio la necessità di programmare in assembler piuttosto che in un linguaggio ad alto livello si verifica quando la velocità di esecuzione o l'occupazione di memoria di un programma sono elementi critici. Dopodiché devo stendere il flow-chart stendere il programma sorgente. Dato il sorgente esso viene tradotto dall'assemblatore o dal compilatore. Tale operazione fornisce in uscita un file oggetto che contiene istruzioni macchina, dati e informazioni di supporto. In generale un file oggetto non può essere eseguito direttamente, perché

fa riferimento a procedure o dati contenuti in altri file. Un'**etichetta** è detta **esterna** o **globale** se è possibile fare riferimento all'oggetto etichettato anche a partire da altri file oltre a quello in cui è definita. Un'**etichetta** è **locale** se può essere usata solo all'interno del file in cui è definita. Dal momento che l'assemblatore elabora solo separatamente ed individualmente ciascuno dei file che compongono il programma, conosce solo gli indirizzi delle etichette locali. Per questo l'assemblatore dipende da un altro strumento, il linker, che riunisce un insieme di file oggetto e di librerie in un unico file eseguibile. Una volta ottenuto il file eseguibile devo simulare il suo funzionamento, cioè devo capire se svolge il proprio compito in modo adeguato.

Si pongono diversi problemi:

- innanzitutto, teoricamente, non posso testare il funzionamento dello sviluppo software finché l'hardware di supporto non è stato progettato e realizzato fisicamente;
- quando integro lo sviluppo hardware e quello software e controllo il funzionamento, se non funziona non riesco a capire il punto critico, in quanto ho troppe variabili in gioco.

I SIMULATORI: LO SPIM.

Per testare il funzionamento del software utilizzo un simulatore. Un simulatore è un ambiente totalmente software, non hardware (la componente hardware è solo quella del calcolatore in cui è ospitato), che replica il funzionamento della CPU.

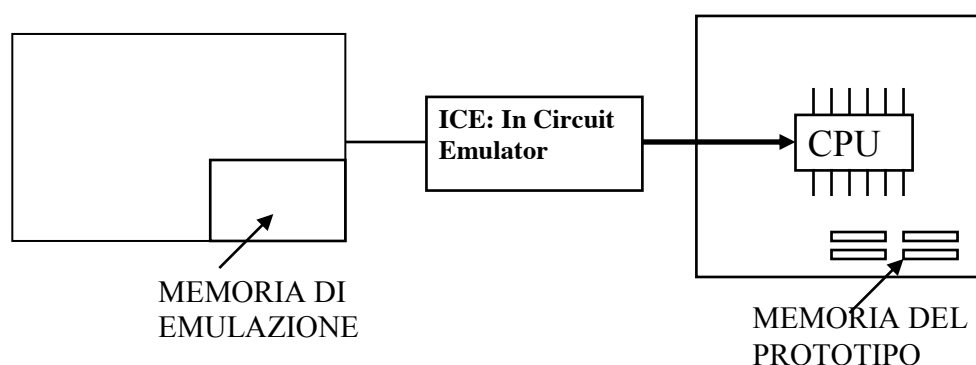
Il simulatore ci dà l'opportunità di vedere che cosa accade all'interno dei registri della CPU all'interno della memoria...svolgendo istruzione per istruzione. Il collaudo software viene fatto quindi indipendentemente da quello hardware.

In questo modo tuttavia, non riesco a collaudare i veri tempi di risposta e l'interazione con il mondo esterno (ad esempio la misurazione della temperatura...), perché non ho effettivamente una circuiteria che mi collega con il mondo esterno.

EMULAZIONE IN TEMPO REALE.

SISTEMA DI SVILUPPO

PROTOTIPO



Nel momento in cui integro il progetto software e quello hardware, anche se ho testato in parte le due componenti separatamente, può accadere che il sistema non funzioni comunque. Per risolvere questo problema ricorro all'utilizzo dell'emulatore.

Sia la parte a destra in arancione la scheda hardware con la CPU, le memorie...come dovrà essere alla fine. Se carico i programmi in quella scheda e il funzionamento non è corretto non riesco a capire dove si sia verificato l'errore. In questa scheda posso estrarre la CPU vera e inserire al suo posto un componente, detto **ICE, In Circuit Emulator**, che collego attraverso un'interfaccia comune ad un calcolatore per scopi generali. Facendo questo collegamento non altero nulla dal punto di vista elettronico e meccanico della mia scheda. In realtà è cambiato che riesco ad avere pieno controllo del software che in questo caso può interagire con il mondo esterno: riesco quindi a testare che cosa accade all'interno del mio sistema, inteso non più solo come CPU o come memoria, ma come sistema globale, hardware e software. Attraverso l'ICE faccio eseguire il programma, così come verrebbe eseguito se nella scheda ci fosse la CPU vera, riuscendo a vedere i risultati di una vera interazione con il mondo esterno.

La differenza tra il simulatore e l'emulatore è che il primo replica il 90% del mio sistema, mentre l'emulatore non replica, ma è il mio sistema, è la vera circuiteria. Con l'emulatore riesco a valutare i tempi effettivi di esecuzione e le risposte agli stimoli esterni al circuito.

Tuttavia talvolta non ho la possibilità di avere una CPU montata su uno zoccolo, cioè una CPU che può essere staccata dal circuito, in quanto gli zoccoli sono molto costosi e inoltre in alcune applicazioni in cui i circuiti sono sottoposti a forti accelerazioni, si corre il rischio che la CPU si stacchi dallo zoccolo. Quando non ho la possibilità di estrarre la CPU dal circuito non faccio più riferimento ad un dispositivo con una CPU interna, che prende il posto della CPU del circuito, poiché questa al suo interno ha una piccola porzione di silicio dedicata al collaudo. Collego il circuito ad un altro dispositivo che, connesso ad un calcolatore mi permette di collaudare il circuito, questa volta con la vera e propria CPU.

SPIM.

È un simulatore software che permette l'esecuzione di programmi scritti per i processori MIPS. Tramite l'utilizzo di questo strumento si possono individuare eventuali errori nel codice eseguibile. Quando si avvia SPIM si visualizza una schermata divisa in 5 parti:

- il pannello in alto è il **display dei registri**, che mostra il valore di tutti i registri, il PC, il EPC, cioè è la fotografia completa della mia CPU.

xspim											
PC	=	00000000	EPC	=	00000000	Cause	=	00000000	BadVaddr	=	00000000
Status	=	00000000	HI	=	00000000	LO	=	00000000			
General registers											
R0 (r0)	=	00000000	R8 (t0)	=	00000000	R16 (s0)	=	00000000	R24 (t8)	=	00000000
R1 (at)	=	00000000	R9 (t1)	=	00000000	R17 (s1)	=	00000000	R25 (s9)	=	00000000
R2 (v0)	=	00000000	R10 (t2)	=	00000000	R18 (s2)	=	00000000	R26 (k0)	=	00000000
R3 (v1)	=	00000000	R11 (t3)	=	00000000	R19 (s3)	=	00000000	R27 (k1)	=	00000000
R4 (a0)	=	00000000	R12 (t4)	=	00000000	R20 (s4)	=	00000000	R28 (gp)	=	00000000
R5 (a1)	=	00000000	R13 (t5)	=	00000000	R21 (s5)	=	00000000	R29 (sp)	=	00000000
R6 (a2)	=	00000000	R14 (t6)	=	00000000	R22 (s6)	=	00000000	R30 (s8)	=	00000000
R7 (a3)	=	00000000	R15 (t7)	=	00000000	R23 (s7)	=	00000000	R31 (ra)	=	00000000
Double floating-point registers											
FP0	=	0.000000	FP8	=	0.000000	FP16	=	0.000000	FP24	=	0.000000
FP2	=	0.000000	FP10	=	0.000000	FP18	=	0.000000	FP26	=	0.000000
FP4	=	0.000000	FP12	=	0.000000	FP20	=	0.000000	FP28	=	0.000000
FP6	=	0.000000	FP14	=	0.000000	FP22	=	0.000000	FP30	=	0.000000
Single floating-point registers											

- Il pannello sottostante detto **segmento di testo**, riporta le istruzioni del programma utente. All'estrema sinistra fra parentesi quadre, compare l'indirizzo esadecimale dell'istruzione. Il secondo numero è la codifica esadecimale dell'istruzione. Il terzo elemento è la descrizione mnemonica dell'istruzione. Tutto ciò che segue il ";" è l'effettiva riga di codice del programma utente che ha generato l'istruzione. Se dopo il ";" non c'è nulla, significa che l'istruzione è stata generata da SPIM come parte della traduzione di una pseudoistruzione

Text segments				
[0x00400000]	0x8fa40000	lw \$4, 0(\$29)	; 89: lw \$a0, 0(\$sp)	
[0x00400004]	0x27a50004	addiu \$5, \$29, 4	; 90: addiu \$a1, \$sp, 4	
[0x00400008]	0x24a60004	addiu \$6, \$5, 4	; 91: addiu \$a2, \$a1, 4	
[0x0040000c]	0x00041080	sll \$2, \$4, 2	; 92: sll \$v0, \$a0, 2	
[0x00400010]	0x00c23021	addu \$6, \$6, \$2	; 93: addu \$a2, \$a2, \$v0	
[0x00400014]	0x0c000000	jal 0x00000000 [main]	; 94: jal main	
[0x00400018]	0x3402000a	ori \$2, \$0, 10	; 95: li \$v0 10	
[0x0040001c]	0x0000000c	syscall	; 96: syscall	

- Il pannello successivo chiamato **segmento di dato e di stack** mostra i dati che si trovano nella memoria del programma e i dati nello stack del programma. Tra parentesi quadre è presente l'indirizzo di memoria, oltre sono rappresentati i dati che effettivamente sono memorizzate in memoria. Sono rappresentate 4 word per riga.

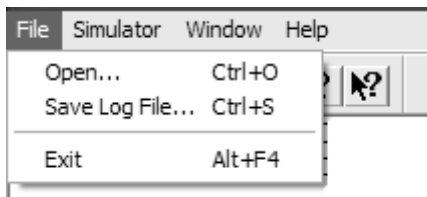
Data segments				
[0x10000000] ... [0x10010000]	0x00000000			
[0x10010004]	0x74706563	0x206e6f69	0x636f2000	0x726f6e67
[0x10010010]	0x72727563	0x61206465	0x6920646e	0x74707572
[0x10010020]	0x000a6465	0x495b2020	0x7265746e	0x6e67696c
[0x10010030]	0x0000205d	0x20200000	0x616e555b	0x6e69206e
[0x10010040]	0x61206465	0x65726464	0x69207373	0x00205d68
[0x10010050]	0x642f7473	0x20617461	0x63746566	0x64646120
[0x10010060]	0x555b2020	0x696c616e	0x64656e67	0x00205d65
[0x10010070]	0x73736572	0x206e6920	0x726f7473	

- Il pannello in basso è quello dei **messaggi di SPIM**, in cui compaiono le segnalazioni di errori.

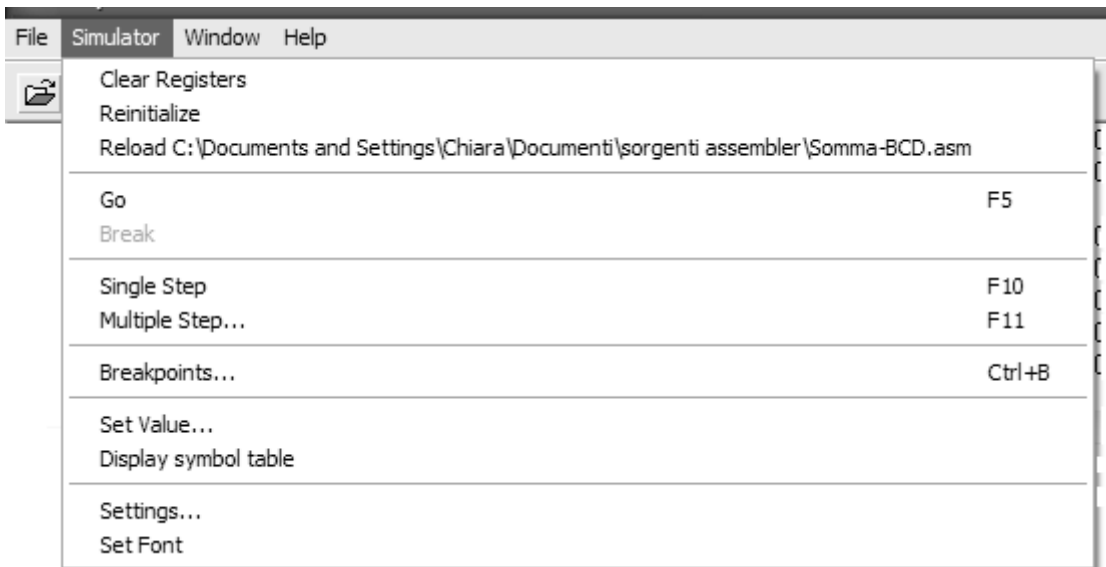
SPIM Version 5.9 of January 17, 1997
 Copyright (c) 1990– 1997 by James R. Larus (larus@cs.wisc.edu)
 All Rights Reserved.
 See the file README for a full copyright notice.

- La posizione dei pulsanti di controllo che consentono di introdurre i comandi di run, di load, di stop...varia a seconda dell'ambiente utilizzato.

Nella versione per Windows XP

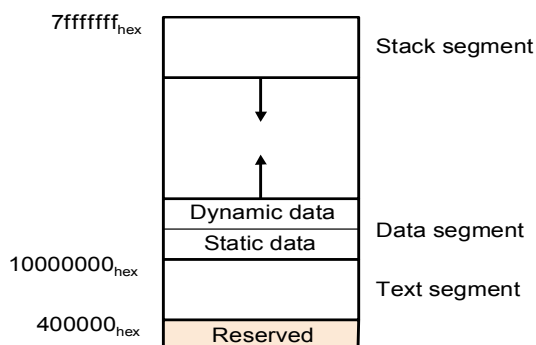


Serve per aprire i file eseguibili e per salvare.



Sono presenti una serie di comandi per il simulatore.

Lo SPIM non replica l'intera memoria del processore, che va dall'indirizzo con tutti 0 all'indirizzo con tutti 1, cioè 2^{32-1} celle di memoria. Lo SPIM fa riferimento ad una memoria più piccola e suddivisa in 3 segmenti:

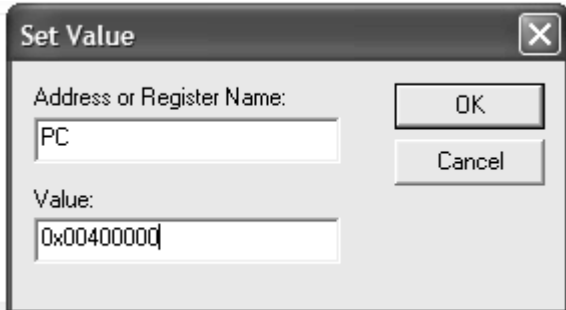


- **testo**: in cui vengono memorizzate le istruzioni di programma a partire da 0x400000.
- **dati** a partire da 0x10000000 (statici e dinamici).
- **stack**: a partire da 0x7fffffff (si espande in senso opposto all'area dati).

Vediamo come caricare ed eseguire un programma.

Con il tasto di Open dal menu File e caricare il programma di interesse. Se non ho commesso errori formali, cioè di stesura di codice, il programma viene caricato con successo.

Prima di far dare il comando di run devo controllare se il valore del PC corrisponde all'indirizzo della prima istruzione. Se questo non accade posso impostare il valore del PC con il comando "Set Value" nel menu "Simulator":



Tale comando può servire anche ad impostare i valori di qualsiasi altro registro e delle celle di memoria.

Dopo aver impostato il PC posso eseguire il programma.

La cosa migliore per verificare se il programma funziona è eseguirlo passo-passo. Per eseguire passo-passo un programma si richiama il comando "*single step*" (o il tasto f10): l'istruzione che deve essere eseguita è in reverse, inoltre il PC si è incrementato di 4 e può capitare che il valore di qualche registro si sia modificato. L'istruzione appena eseguita appare anche in basso nella finestra dei messaggi.

Un'alternativa migliore, soprattutto se il i cicli nel programma si ripetono un grande numero di volte, è usare dei **breakpoint**, cioè dei punti di arresto nei quali voglio verificare il funzionamento del programma. In un programma possono esserci più breakpoint: il programma si ferma al primo che trova. I breakpoint si impostano selezionando la voce "breakpoint" dal menu "Simulator" e inserendo nella finestra che appare l'indirizzo dell'istruzione alla quale si vuole che il programma si arresti. Per inserire effettivamente il breakpoint bisogna selezionare il pulsante add. Davanti all'indirizzo dell'istruzione selezionata come breakpoint compare un asterisco.

(se ho dei cicli, metto un breakpoint all'istruzione successiva a quella di fine ciclo, per verificare se le condizioni di fine ciclo sono giuste).

L'esecuzione passo-passo e l'introduzione di breakpoint fa sì che con ottime probabilità vengano scovati anche gli errori concettuali presenti nel programma.

PROGRAMMA 1

Programma per il test del funzionamento di un'area di memoria Ram: l'area è costituita da 64k byte a partire dall'indirizzo 0x10000000. La memoria viene completamente scritta e riletta effettuando un confronto fra i dati scritto e letto. In ogni cella si memorizza un dato uguale all'indirizzo della cella.

\$t0 usato come puntatore alla memoria e come dato da memorizzare/controllare.

\$t1 usato come indice di fine check

\$t2 usato per il dato riletto

```
li    $t0, 0x10000000 # N.B. e' una pseudoistruzione
lui   $t1, 0x1001
write: sw    $t0, 0($t0)
      addi $t0, $t0, 4
      bne  $t1, $t0, write
# fine scrittura
      lui  $t0, 0x1000
```

```

read:      lw    $t2, 0($t0)
           sub   $t3, $t0, $t2
           bne   $t3, $zero, error
           addi  $t0, $t0, 4
           bne   $t1, $t0, read

# lettura dati conclusa in modo OK
loop:      j      loop

# lettura dati KO: trovato un errore
error:     j      error

```

Modifiche e precisazioni:

```

sub   $t3, $t0, $t2
bne   $t3, $zero, error

```

potevo per risparmiare un'istruzione: `bne $t0, $t2, error`

PRINCIPALI DIRETTIVE PER L'ASSEMBLATORE MIPS.

L'assemblatore MIPS consente l'utilizzo di comandi, dette direttive, che non sono istruzioni del processore. Tali direttive cominciano tutte con il ".". Analizziamo le diverse direttive:

➤ **.ascii "stringa"**

Consente di caricare in memoria la stringa specificata successivamente. Se per esempio scrivo: `.ascii "lezione di calcolatori"`, l'assemblatore si preoccupa di collocare in memoria questa stringa codificata con i byte delle singole lettere. La memoria coinvolta in questa direttiva è la ROM. Infatti il messaggio che il programmatore salva in memoria non deve essere modificato dall'utente, inoltre nella memoria del mio dispositivo finale su cui voglio far comparire quella stringa non ho l'assemblatore, ma solo la porzione di codice già assemblata da eseguire che non può essere modificata.

➤ **.asciiz "stringa"**

Consente di caricare in memoria la stringa specificata successivamente inserendo alla fine il carattere Null. Avere il Null al termine della stringa è utile per verificare quando la stringa è terminata.

➤ **.byte b1, b2...bn**

Talvolta è conveniente specificare dei valori direttamente in memoria. In questo modo è più facile modificare tale valore, piuttosto che se fosse incorporato in un'istruzione che consente di trattare delle costanti (`addi...`). Tale direttiva memorizza gli n valori scritti di seguito in byte consecutivi della memoria.

➤ **.word w1, w2...wn**

Se voglio memorizzare costanti lunghe 32 bit in parole consecutive della memoria. Teoricamente si potrebbe presentare il problema dell'allineamento: non posso scrivere una costante a cavallo tra 2 parole. Tuttavia l'assemblatore si preoccupa di scrivere i dati all'inizio di una word, quindi allineati.

➤ **.data <ind>**

Specifica l'indirizzo di partenza dell'area che voglio contenga i dati. Se non è presente nessun indirizzo, gli elementi successivi sono memorizzati nel segmento di dato, quindi a partire dall'indirizzo 0x10010000. Non colloca nessun valore in memoria.

➤ **.text <ind>**

Specifica l'indirizzo di partenza dell'area che voglio contenga le istruzioni. Se non è presente nessun indirizzo, gli elementi successivi sono memorizzati nel segmento di testo, quindi all'indirizzo di default 0x0040000. Non colloca nessun valore in memoria.

➤ **.float f1,f2...fn**

Memorizza i numeri in virgola mobile in singola precisione, quindi su 32 bit, scritti di seguito in memoria.

➤ **.double d1, d2...dn**

Memorizza i numeri in virgola mobile in doppia precisione, quindi su 64 bit, scritti di seguito in memoria.

➤ **.space num**

Riserva uno spazio nella memoria dell'entità specificata di seguito. Fa riferimento alla memoria RAM, perché riservo uno spazio in cui durante l'evoluzione del programma scrivo dei risultati o dei dati.

Ho, per esempio, necessità di costruire una tabella di dati, per contenere per esempio dei risultati. Posso fare in questo modo:

```
ris:      .space 100
```

Così mi riservo un vettore di 100 spazi. I miei risultati all'indirizzo specificato da `ris`.

➤ **.global simb**

Dichiara l'etichetta `simb` globale, e che ad essa è possibile fare riferimento da altri file.

Nello sviluppo di un progetto complesso, è buona regola suddividere il programma in diverse porzioni di codice, ognuna delle quali si occupa di assolvere un solo compito in modo che sia più modificabile e correggibile. Una parte fa riferimento alle altre ricorrendo alle etichette globali. Un'**etichetta** è detta esterna o **globale** se è possibile fare riferimento all'oggetto etichettato anche a partire da altri file oltre a quello in cui è definita. Ad esempio:

```
main: ... ..      math:  ... ..  
      jal seno      seno:  ... ..
```

se non avessi definito l'etichetta `seno` globale, l'assemblatore dà errore, poiché non c'è nessuna etichetta nel main di nome `seno`. Definendo l'etichetta globale, l'assemblatore riconosce che quell'etichetta è presente in un'altra porzione di codice.

Sul mercato ci sono memorie con tagli predefiniti. Non troverò mai un dispositivo ROM reale che abbia esattamente il numero di caselle che mi interessa e se ne prendo una con più caselle vincolo il fatto che non posso collocare la RAM. Nella realtà non avrò mai ROM e RAM consecutive. In questo caso devo stabilire l'indirizzo da cui parte la RAM: lo faccio con la direttiva `.data`:

l'area della ROM comincia all'indirizzo: `.data 0x10000100`

```
msg:      .ascii "Lezione 1"  
cost:     .word 1, 2, 5, 19
```

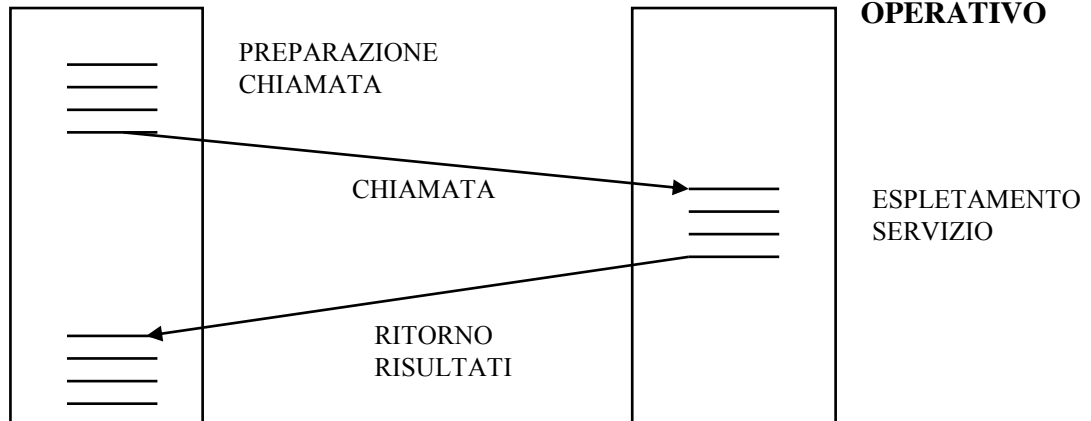
l'area della RAM comincia all'indirizzo: `.data 0x10011000`

```
ris:      .space 100
```

CHIAMATE DI SISTEMA: SYSCALL.

PROGRAMMA
UTENTE

SISTEMA
OPERATIVO



Uno dei difetti del simulatore è il fatto che non può interagire con il mondo esterno. Tuttavia, possiamo fare interagire in parte lo SPIM con l'esterno, in quanto esso è installato su un calcolatore, che quindi ha come input la tastiera e come output il video. Tramite la Syscall riusciamo a far interagire lo SPIM con il mondo esterno. Quindi un programma MIPS eseguito con SPIM può leggere i caratteri inseriti dall'utente e scrivere dei caratteri sulla finestra della console. Syscall non è un'istruzione MIPS, ma un comando dello SPIM!

SPIM fornisce tramite la System Call, servizi analoghi a quelli del sistema operativo. Syscall è vista come l'invocazione di una subroutine. Il passaggio dei parametri alla syscall avviene attraverso i registri \$a0-\$a3, se interi, o \$f12, se in virgola mobile; il codice della chiamata viene caricato nel registro \$v0. Le chiamate di sistema che forniscono un valore in uscita mettono il risultato nel registro \$v0, se interi, o \$f0, se in virgola mobile. Se ci sono più parametri, viene utilizzata la memoria.

Il seguente codice, ad esempio, vuole stampare sulla console "risposta=5".

```
.data                #segmento dati
str:
.asciiz "risposta ="

.text                #segmento testo
li $v0, 4            #codice della chiamata di sistema print_string
la $a0, str          #indirizzo della stringa da stampare
syscall              #stampa della stringa
li $v0, 1            #codice della chiamata di sistema print_int
li $a0, 5            #intero da stampare
syscall              #stampa dell'intero.
```

La chiamata di sistema:

`print_int` → stampa sulla console il numero intero che le viene passato come argomento nel registro \$a0.

`print_float` → stampa un numero in virgola mobile in singola precisione passato come argomento.

`print_double` → stampa un numero in virgola mobile in doppia precisione passato come argomento.

`print_string` → stampa la stringa passata come argomento nel registro \$a0 terminandola con il carattere Null. Nel registro sarà contenuto l'indirizzo di memoria in cui


la stringa comincia, in quanto non tutte le stringhe possono essere contenute in un registro.

`read_int` → leggono un'intera linea in ingresso fino al carattere a capo incluso. I caratteri che seguono il numero intero sono ignorati. Restituiscono i numeri letti in `$v0`.

`read_float` → leggono un'intera linea in ingresso fino al carattere a capo incluso. I caratteri che seguono il numero in virgola mobile sono ignorati. Restituiscono i numeri letti in `$v0`.

`read_double` → leggono un'intera linea in ingresso fino al carattere a capo incluso. I caratteri che seguono il numero in virgola mobile in doppia precisione sono ignorati. Restituiscono i numeri letti in `$v0`.

`read_string` → legge fino a n-1 caratteri scrivendoli in un buffer e terminando la stringa con il carattere Null.

	COMANDO	CODICE CHIAMATA	Input	Output
	Print_int	1	\$a0	
	Print_float	2	\$f1	
	Print_double	3	\$f1	
	Print_string	4	\$a0	
	Read_int	5		\$v0
	Read_float	6		\$f0
	Read_double	7		\$f0
	Read_string	8	\$a0=buf \$a1=lungh	

PROGRAMMA 2: SOMMA BCD

	BCD	Codice BCD → ogni cifra decimale viene codificata con 4 bit. È un codice
0	0000	pesato, poiché il valore di ogni cifra viene ottenuto eseguendo una somma
1	0001	pesata delle 4 cifre binarie che lo compongono. È usato in sistemi di
2	0010	visualizzazione (calcolatrici). Lascia 6 configurazioni inutilizzate delle 16 a
3	0011	disposizione con 4 bit.
4	0100	
5	0101	255 in BCD = 0010 0101 0101.
6	0110	NB: non si possono rappresentare i numeri negativi, in quanto rappresenta solo
7	0111	le cifre decimali!!!
8	1000	
9	1001	

Programma per la somma di 4 coppie di numeri memorizzati in codice ASCII. I risultati vengono memorizzati in una area che segue quella in cui ci sono i dati.

```
.data
dati:.byte    0x33, 0x36, 0x34, 0x38
      .byte    0x30, 0x39, 0x31, 0x33
# oppure avrei potuto scrivere:
# dati:  .ascii  "36480913"

.text
# Main program
# $s0 puntatore per il caricamento dati
# $s1 e $s2 addendi e poi risultato
# $s3 indirizzo fine area dati
# $t0 per l'esito di confronti
      li    $s0, 0x10010000    # = dati
      addi  $s3, $s0, 8
ciclo: lb    $a0, 0($s0)
      jal   check
      bne   $v0, $zero, error
      move  $s1, $a0 # pseudoistruzione
      andi  $s1, $s1, 0xf
# primo addendo in BCD
      lb    $a0, 1($s0)
      jal   check
      bne   $v0, $zero, error
      move  $s2, $a0 # pseudoistruzione
      andi  $s2, $s2, 0xf
# secondo addendo in BCD
      add   $s1, $s1, $s2
      slti  $t0, $s1, 0xa
      beq   $t0, $zero, aggBCD
# salta se bisogna aggiustare in BCD
# altrimenti converti in ASCII
ascii: move  $s2, $s1
      andi  $s1, $s1, 0xf
      ori   $s1, $s1, 0x30
```

```

        sb    $s1, 8($s0)
        srl   $s1, $s2, 4
        ori   $s1, $s1, 0x30
        sb    $s1, 9($s0)
        addi  $s0, $s0, 2
        bne   $s0, $s3, ciclo
# conclusione calcoli: tutto OK
fineOK:   j    fineOK

# aggiustamento BCD: + 6 per superare la
# i numeri non usati dal BCD
aggBCD:   addi $s1, $s1, 6
          j    ascii

# trovato un errore:
# un addendo non e' un numero
error:    j    error

# Subroutine di check se i carattere ASCII
# letto e' un numero
# $a0 contiene il dato da convertire
# $v0 = 0, se il dato e' un numero,
# $v0 = 1, altrimenti.
# Usa $t0 per il dato privo di parita'.
# Usa $t1 per l'esito dei confronti.
check:    add  $v0, $zero, $zero
          andi $t0, $a0, 0x7f
# eliminazione bit parita'
          slti $t1, $t0, 0x30
          bne  $t1, $zero, errore
          slti $t1, $t0, 0x3a
          beq  $t1, $zero, errore
          j    $ra
errore:    ori  $v0, $v0, 1
          j    $ra

```

PROGRAMMA 2 BIS: SOMMA BCD BIS.

Programma per la somma di 4 coppie di numeri memorizzati in codice ASCII. I risultati vengono memorizzati in una area che segue quella in cui ci sono i dati.

VERSIONE CON SYSCALL PER VISUALIZZAZIONI

```

.data
str1a:    .asciiz  "Il primo numero letto = "
str1b:    .asciiz  "Il secondo numero letto = "
str2:     .asciiz  "La somma vale = "
nl:       .asciiz  "\n"
fine:     .asciiz  "Fine calcoli!"
dati:     .byte    0x33, 0x36, 0x34, 0x38
          .byte    0x30, 0x39, 0x31, 0x33
# oppure avrei potuto scrivere:
# dati:     .ascii  "36480913"

```

```

    .text
# Main program
# $s0 puntatore per il caricamento dati
# $s1 e $s2 addendi e poi risultato
# $s3 indirizzo fine area dati
# $t0 per l'esito di confronti
    .globl    __start
    .globl    aggBCD
__start:  la    $s0, dati
          addi $s3, $s0, 8
ciclo:    lb    $a0, 0($s0)
          jal   check
          bne   $v0, $zero, error
          move  $s1, $a0 # pseudoistruzione
          andi  $s1, $s1, 0xf
# primo addendo in BCD e visualizza
          li    $v0, 4    #print string
          la    $a0, str1a
          syscall
          li    $v0, 1    #print int
          move  $a0, $s1
          syscall
          li    $v0, 4    #print string: a capo
          la    $a0, n1
          syscall
          lb    $a0, 1($s0)
          jal   check
          bne   $v0, $zero, error
          move  $s2, $a0 # pseudoistruzione
          andi  $s2, $s2, 0xf
# secondo addendo in BCD e visualizza
          li    $v0, 4    #print string
          la    $a0, str1b
          syscall
          li    $v0, 1    #print int
          move  $a0, $s2
          syscall
          li    $v0, 4    #print string: a capo
          la    $a0, n1
          syscall
          add   $s1, $s1, $s2
# visualizza il risultato
          li    $v0, 4    #print string
          la    $a0, str2
          syscall
          li    $v0, 1    #print int
          move  $a0, $s1
          syscall
          li    $v0, 4    #print string: a capo
          la    $a0, n1
          syscall

```

```

        slti $t0, $s1, 0xa
        beq  $t0, $zero, aggBCD
# salta se bisogna aggiustare in BCD
# altrimenti converti in ASCII
ascii:   move $s2, $s1
        andi $s1, $s1, 0xf
        ori  $s1, $s1, 0x30
        sb   $s1, 8($s0)
        srl  $s1, $s2, 4
        ori  $s1, $s1, 0x30
        sb   $s1, 9($s0)
        addi $s0, $s0, 2
        bne  $s0, $s3, ciclo
# conclusione calcoli: tutto OK
fineOK:  li   $v0, 4      #print string
        la   $a0, fine
        syscall
end:      j    end

# aggiustamento BCD: + 6 per superare la
# i numeri non usati dal BCD
aggBCD:  addi $s1, $s1, 6
        j    ascii

# trovato un errore:
# un addendo non e' un numero
error:   j    error

# Subroutine di check se il carattere ASCII
# letto e' un numero
# $a0 contiene il dato da convertire
# $v0 = 0, se il dato e' un numero,
# $v0 = 1, altrimenti.
# Usa $t0 per il dato privo di parita'.
# Usa $t1 per l'esito dei confronti.
check:   add  $v0, $zero, $zero
        andi $t0, $a0, 0x7f
# eliminazione bit parita'
        slti $t1, $t0, 0x30
        bne  $t1, $zero, errore
        slti $t1, $t0, 0x3a
        beq  $t1, $zero, errore
        j    $ra
errore:  ori  $v0, $v0, 1
        j    $ra

```

APPENDICE C COME E' FATTA UNA ALU?

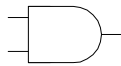
RIPASSO DI RETI LOGICHE.

Vorrei costruire un sommatore binario, ovvero un oggetto che, dati 2 ingressi ed un eventuale riporto, mi restituisca in uscita la somma di questi ed un riporto. Posso affermare che il tempo di esecuzione della somma è di 2Δ , essendo 2 gli addendi da sommare. Se facessi lo stesso ragionamento con n ingressi, dovrei dire che il tempo di esecuzione per la somma di n addendi è $n\Delta$.

Tuttavia c'è un teorema delle reti combinatorie che afferma che qualunque rete con in ingresso un qualunque numero di bit, può essere trasformata in una rete a 2 livelli che mi fornisce l'uscita in 2Δ , cioè nella somma dei tempi di passaggio attraverso ogni gate, cioè livello (ogni porta). Grazie infatti alle forme canoniche riesco ad ottenere una somma come un OR di tanti AND che lavorano in parallelo.

AND

A	B	O
0	0	0
0	1	0
1	0	0
1	1	1

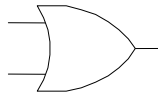


NAND

A	B	U
0	0	1
0	1	1
1	0	1
1	1	0

OR

A	B	O
0	0	0
0	1	1
1	0	1
1	1	1



NOR

A	B	U
0	0	1
0	1	0
1	0	0
1	1	0

EXOR

A	B	U
0	0	0
0	1	1
1	0	1
1	1	0

LEGGI DI DE MORGAN

$$\overline{x + y} = \bar{x} \cdot \bar{y}$$

$$\overline{x \cdot y} = \bar{x} + \bar{y}$$

Problema: consideriamo una funzione logica con 3 ingressi: A, B, C.

‣ L'uscita D è vera se è vero almeno 1 degli ingressi:

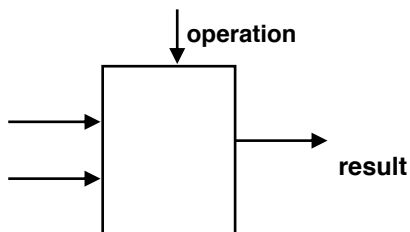
$$D = A + B + C$$

‣ L'uscita E è vera se 2 ingressi sono veri:

$E = \text{con la mappa di Karnaugh} = A \cdot BC + AB \cdot C + ABC \cdot$

- L'uscita F è vera se tutti e 3 gli ingressi sono veri:
 $F = ABC$.

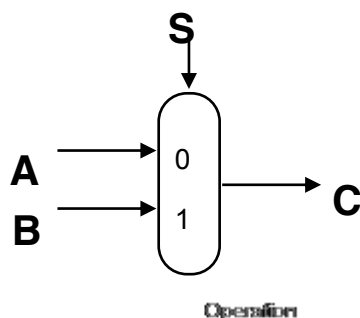
Vogliamo costruire 1 ALU: suppongo per ora di voler costruire 1 ALU ad 1 bit, che svolga le funzioni di AND, OR, somma, sottrazione.



In base al segnale che entra nella ALU, voglio, per ora svolgere l'AND e l'OR:

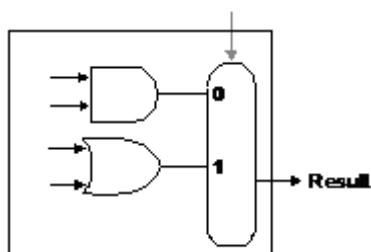
$R = AB + opB + opA$ è una rete a 2 livelli: OR di AND

Per facilitare la comprensione e la struttura della ALU introduciamo un **MULTIPLEXER**.



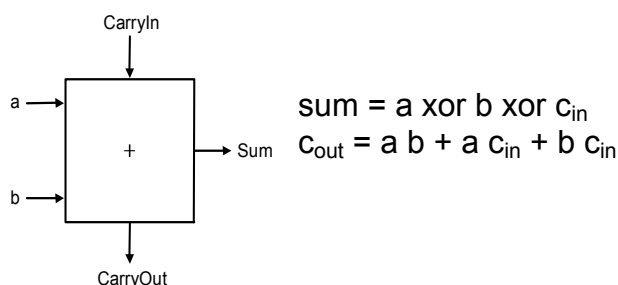
Un Multiplexer è un selettore di uscita, cioè un "interruttore" che permette di prendere, tra i due ingressi quella la cui codifica è uguale al segnale di comando. Si consideri un multiplexer a 2 vie, cioè a 3 ingressi, 2 di dato ed 1 di selezione. Il valore dell'ingresso di selezione determina quale degli ingressi viene usato per determinare il valore dell'uscita.

$$C = AS^* + BS$$



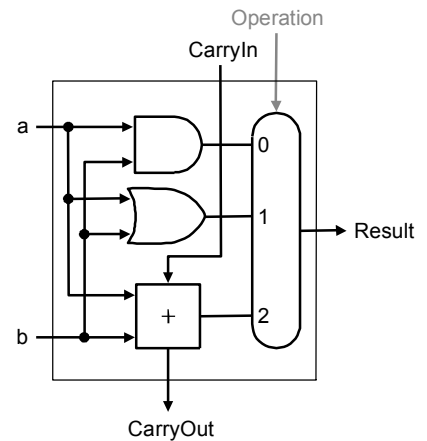
Per realizzare le funzioni di AND e OR, considero un singolo AND ed un singolo OR in ingresso ad un multiplexer: le uscite di queste due porte logiche rappresentano 2 degli ingressi al multiplexer. Il bit di controllo del multiplexer seleziona l'uscita Result: se tale bit è 0, esce l'operazione di AND, se tale bit è 1, esce l'operazione di OR:

Consideriamo la ALU ad 1 bit che faccia solo la somma:

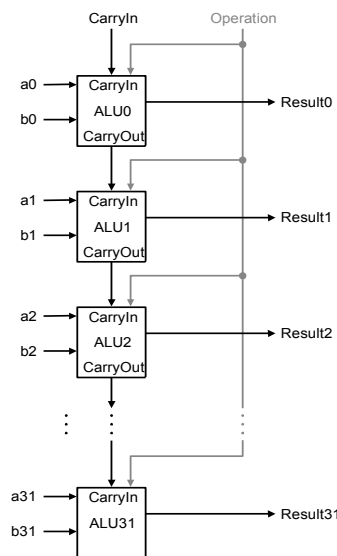


Se ora voglio creare una ALU ad 1 bit che svolga le operazioni di AND, OR e somma, devo aggiungere 1 bit di controllo al multiplexer, rispetto al caso in cui svolga solo AND ed OR. Consideriamo dunque un multiplexer a 4 ingressi ed 1 uscita, con 2 bit di controllo. Gli ingressi del multiplexer sono:

- l'uscita dell'AND;
- l'uscita dell'OR;
- l'uscita del sommatore.



L'uscita del multiplexer corrisponde all'uscita selezionata dal comando in ingresso. Quindi calcolo $A \text{ and } B$, $A \text{ or } B$, $A+B$: in uscita dalla ALU ho l'operazione selezionata dall'ingresso del multiplexer, che quindi funge da interruttore, chiudendosi solo per l'operazione di mio interesse. L'ingresso che comanda l'uscita del multiplexer, cioè l'*operation*, non è null'altro che il codice operativo dell'istruzione, cioè quella parte dell'istruzione che ordina l'operazione da svolgere.



Per creare una ALU a 32 bit pongo in cascata 32 ALU ad 1 bit. In questo caso abbiamo propagato i riporti. L'AND e l'OR si fanno in contemporanea a tutti i bit omologhi, la somma non può essere fatta contemporaneamente, ma devo tenere conto dei riporti.

SOMMATORI.

Ragioniamo per semplicità con una ALU a 16 bit. Sia Δ il tempo di passaggio attraverso 1 gate, cioè attraverso 1 porta logica.

RIPPLE CARRY ADDER

È il sommatore a propagazione di riporto. Ogni singolo sommatore impiega 2Δ per produrre ogni risultato e ogni CarryOut. Infatti:

$$\text{sum} = a \text{ xor } b \text{ xor } c_{in}$$

$$c_{out} = a b + a c_{in} + b c_{in}$$

Essi sono rappresentati come somme di prodotti, quindi devono passare attraverso 2 livelli di logica. Se 16 sono i bit che devo sommare, e ogni risultato viene prodotto in 2Δ , il tempo totale per avere la somma è di 32Δ .

SOMMATORE CON HARDWARE INFINITO.

Per il teorema delle reti combinatorie citato sopra, ogni rete combinatoria può essere semplificata ad una rete a 2 livelli che produce il risultato in 2Δ . Quindi vorrei costruire un sommatore che mi restituisce il risultato in 2Δ .

Innanzitutto non deve esserci logica in cascata, in quanto tale logica presuppone la propagazione dei riporti, ma una logica in parallelo.

Le espressioni dei riporti sono dati in questo modo:

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$

$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$

$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$

Data l'espressione di c_2 , sostituisco in essa l'espressione di c_1 :

$$\begin{aligned} c_2 &= b_1c_1 + a_1c_1 + a_1b_1 = b_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1b_1 = \\ &= b_1b_0c_0 + a_0b_1c_0 + a_0b_0b_1 + a_1b_0c_0 + a_1a_0c_0 + a_1a_0b_0 + a_1b_1 \end{aligned}$$

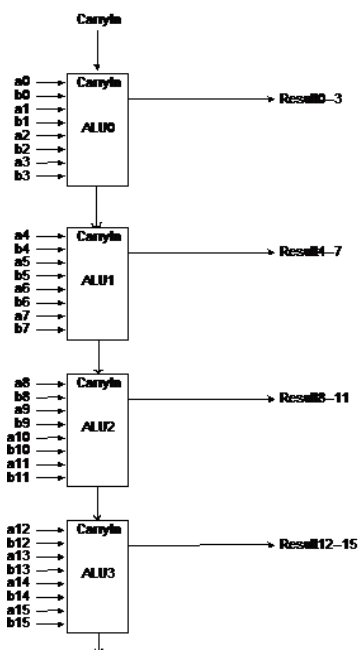
Da questa espressione deduco che ho riporto dai bit di peso 1 se:

- $a_1=1$ e $b_1=1 \rightarrow a_1b_1$
oppure:
- $a_1=1$ e $b_0=1$ e $a_0=1 \rightarrow a_1a_0b_0$
oppure:
- $a_1=1$ e $c_0=1$ e $a_0=1 \rightarrow a_1a_0c_0$
e così via.

Per calcolare c_2 ci impiego quindi 2Δ .

Ora se, data l'espressione di c_3 , sostituisco in essa l'espressione di c_2 , ottengo una somma di prodotto che dipende da a_2b_2 e dai bit di peso precedente. L'equazione diventa molto lunga, ma a livello di tempi di esecuzione ho sempre 2Δ .

Tuttavia tale soluzione non è fattibile, se pur in teoria ho un sommatore che restituisce la somma in 2Δ . Dal punto di vista teorico e concettuale, tale soluzione è fattibile ed è la migliore in assoluto, tuttavia, in pratica quando devo costruire c_{15} , ho un numero di dispositivi e di termini enorme, che occuperebbe troppo spazio!



Potresti considerare 4 sommatore infiniti in cascata: infatti poiché posso calcolare c_4 , in modo relativamente poco complesso, posso scegliere di adottare tale soluzione: costruisco un sommatore che calcola il risultato dei:

- bit di peso 0;
- bit di peso 1;
- bit di peso 2;
- bit di peso 3;

in questo modo ho un sommatore con hardware infinito, che impiega un tempo di 2Δ per restituire il risultato.

In cascata a questo sommatore ne pongo un altro che calcola il risultato dei:

- bit di peso 4;
- bit di peso 5;
- bit di peso 6;
- bit di peso 7;

in questo modo ho un altro sommatore con hardware infinito, che impiega un tempo di 2Δ per restituire il risultato.

Se voglio calcolare una somma su 16 bit, ho bisogno di 4 di questi dispositivi. Il tempo totale per calcolare il risultato risulta essere: $4 \cdot 2\Delta = 8\Delta$.

CARRY-LOOKAHEAD ADDER CON 1° LIVELLO DI ASTRAZIONE.

Cerco di interpretare in modo diverso la funzione che fornisce i riporti:

$$\begin{aligned} c_1 &= b_0c_0 + a_0c_0 + a_0b_0 & \rightarrow & c_1 = a_0b_0 + (a_0 + b_0)c_0 \\ c_2 &= b_1c_1 + a_1c_1 + a_1b_1 & \rightarrow & c_2 = a_1b_1 + (a_1 + b_1)c_1 \\ c_3 &= b_2c_2 + a_2c_2 + a_2b_2 & \rightarrow & c_3 = a_2b_2 + (b_2 + a_2)c_2 \end{aligned}$$

$$c_2 = a_1b_1 + (a_1 + b_1)c_1 = a_1b_1 + (a_1 + b_1)(a_0b_0 + (a_0 + b_0)c_0).$$

$$c_3 = a_2b_2 + (b_2 + a_2)c_2 = a_2b_2 + (b_2 + a_2)(a_1b_1 + (a_1 + b_1)(a_0b_0 + (a_0 + b_0)c_0))$$

$$c_1 = a_0b_0 + (b_0 + a_0)c_0$$

Questa espressione in realtà significa che, c'è riporto c_1 , se:

- $a_0b_0 \rightarrow b_0=1$ e $a_0=1$. chiamo questo termine di GENERAZIONE, g ; esso è 1 se entrambi gli ingressi sono a 1.
- $(b_0 + a_0)c_0 \rightarrow (b_0 + a_0)=1$ e $c_0=1$. chiamo questo termine PROPAGAZIONE, p ; esso è a 1 se almeno uno tra gli ingressi è a 1 e il riporto dello stadio precedente è a 1.

Questa espressione in realtà significa che, c'è riporto c_1 , se:

- Il termine di generazione è 1;
- Il termine di propagazione è 1.

Quindi:

$$c_1 = a_0b_0 + (b_0 + a_0)c_0 \rightarrow c_1 = g_0 + p_0c_0$$

Il riporto viene generato se i bit di peso omologo sono entrambi a 1, oppure se i bit di peso omologo propagano il riporto dei bit precedenti.

$$c_2 = a_1b_1 + (a_1 + b_1)c_1 = a_1b_1 + (a_1 + b_1)(a_0b_0 + (a_0 + b_0)c_0).$$

$$c_2 = g_1 + p_1c_1 \rightarrow c_2 = g_1 + p_1(g_0 + p_0c_0) = g_1 + p_1g_0 + p_1p_0c_0$$

$$c_2 = g_1 + p_1g_0 + p_1p_0c_0$$

Questa espressione in realtà significa che, c'è riporto c_2 , se:

- I bit di peso omologo lo generano, cioè sono entrambi a 1;
- Oppure se è stato generato dai bit di peso 0 ed è stato propagato dai bit di peso 1;
- Oppure se è arrivato un riporto c_0 dall'esterno e l'anno propagato i bit di peso 1 e di peso 0.

$$c_3 = a_2b_2 + (b_2 + a_2)c_2 = a_2b_2 + (b_2 + a_2)(a_1b_1 + (a_1 + b_1)(a_0b_0 + (a_0 + b_0)c_0))$$

$$c_3 = g_2 + p_2c_2 \rightarrow c_3 = g_2 + p_2(g_1 + p_1g_0 + p_1p_0c_0) =$$

$$c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$$

Questa espressione in realtà significa che, c'è riporto c_3 , se:

- I bit di peso omologo lo generano, cioè sono entrambi a 1;
- Oppure se è stato generato dai bit di peso 1 ed è stato propagato dai bit di peso 2;
- Oppure se è stato generato dai bit di peso 0 ed è stato propagato dai bit di peso 2 e dai bit di peso 1;
- Oppure se è arrivato un riporto c_0 dall'esterno e l'anno propagato i bit di peso 2, dai bit di peso 1 e di peso 0.

$$c_4 = g_3 + p_3c_3 \rightarrow c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

Questa espressione in realtà significa che, c'è riporto c_4 , se:

- I bit di peso omologo lo generano, cioè sono entrambi a 1;
- Oppure se è stato generato dai bit di peso 2 ed è stato propagato dai bit di peso 3;
- Oppure se è stato generato dai bit di peso 1 e i bit di peso 3 e di peso 2 lo propagano;

- Oppure se è stato generato dai bit di peso 0 ed è stato propagato dai bit di peso 3, dai bit di peso 2 e dai bit di peso 1;
- Oppure se è arrivato un riporto c_0 dall'esterno e l'anno propagato i bit di peso 3, dai bit di peso 2, dai bit di peso 1 e di peso 0.

Ragioniamo su quanto tempo ci impiega un circuito del genere a produrre il risultato.

Ogni riporto è dato da un'espressione del tipo somma di prodotti, quindi può essere svolto da una logica a 2 livelli: il tempo impiegato è quindi di 2Δ . Tuttavia devo ricordare che gli ingressi non sono g e p , ma a e b . I termini g e p si ottengono da a e b passando attraverso un ulteriore livello di logica: essi si ottengono in parallelo tramite operatori AND (generazione) e OR (propagazione), che quindi richiedono un tempo Δ . Il tempo totale per l'operazione risulta essere 3Δ .

Il tempo ottenuto è molto prossimo al caso ideale. Teoricamente in questo modo riuscirei a costruire un sommatore a 16 bit. Tuttavia è ancora complesso costruire i riporti dei bit di peso 14, 15 e l'hardware diventerebbe troppo ingombrante in termini di spazio.

Possiamo anche in questo caso perseguire la strada già battuta nel caso del sommatore con hardware infinito.

Posso realizzare un CLA a 16 bit, utilizzando 4CLA a 4bit in cascata. Il tempo totale che si richiede per fornire il risultato è di 9Δ :

CLA1 $\rightarrow 3\Delta$

CLA2 $\rightarrow 2\Delta$ in quanto i termini di generazione e propagazione dipendono solo dagli ingressi e quindi possono essere calcolati tutti insieme solo una volta, e quindi risultano disponibili immediatamente dopo Δ , che non deve essere sprecato ad ogni stadio, ma solo 1 volta. Il CLA2 deve aspettare il riporto dello stadio precedente e poi calcolare il risultato.

CLA3 $\rightarrow 2\Delta$ in quanto i termini di generazione e propagazione dipendono solo dagli ingressi e quindi possono essere calcolati tutti insieme solo una volta.

CLA4 $\rightarrow 2\Delta$ in quanto i termini di generazione e propagazione dipendono solo dagli ingressi e quindi possono essere calcolati tutti insieme solo una volta, per esempio al 1° stadio.

CARRY-LOOKAHEAD ADDER CON 2° LIVELLO DI ASTRAZIONE.

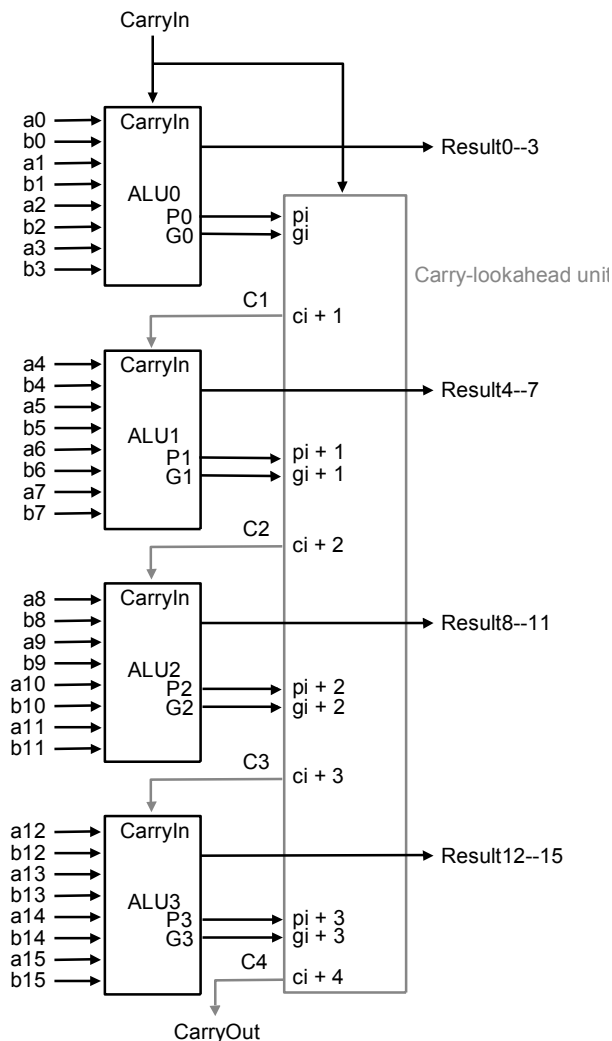
Realizziamo un "super" sommatore a 4 bit (CLA a 4 bit), in cui i riporti non siano in cascata, ma possano essere calcolati in parallelo.

I riporti in ingresso ai 4 CLA del sommatore a 16 bit sono simili ai riporti in uscita da ciascun bit del sommatore a 4 bit c_1, c_2, c_3, c_4

$$C_1 = G_0 + P_0 c_0$$

Il riporto del 1° blocco viene generato se:

- è lui stesso a generarlo, quindi a livello di bit di peso 0,1,2,3;
- oppure se esso propaga del blocco precedente.



$$C_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

Il riporto del 2° blocco viene generato se:

- è lui stesso a generarlo, quindi a livello di bit 4,5,6,7;
- oppure se è generato dal blocco precedente (bit di peso 0,1,2,3) ed esso lo propaga;
- oppure se c'è in ingresso un riporto c_0 che viene propagato dal blocco stesso e dal blocco precedente.

$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

Il riporto del 3° blocco viene generato se:

- è lui stesso a generarlo, quindi a livello di bit di peso 8,9,10,11;
- oppure se è generato dal blocco precedente (dai bit di peso 4,5,6,7) ed esso lo propaga;
- oppure se è generato dal 1° blocco (bit di peso 0,1,2,3) ed il 2° blocco ed il 3° blocco lo propagano;
- oppure se c'è in ingresso un riporto c_0 che viene propagato dal blocco stesso e dal blocco precedente e da quello precedente ancora.

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

Il riporto del 4° blocco viene generato se:

- è lui stesso a generarlo, quindi a livello di bit di peso 12, 13, 14, 15;
- oppure se è generato dal 3° blocco precedente (dai bit di peso 8,9,10,11) ed esso lo propaga;
- oppure se è generato dal 2° blocco (bit di peso 4,5,6,7) ed il blocco stesso e quelli precedenti lo propagano;
- oppure se c'è in ingresso un riporto c_0 che viene propagato dal blocco stesso e da quelli precedenti.

I "super" segnali "propaga" P_i sono dati da:

$$P_0 = p_3 p_2 p_1 p_0;$$

$$P_1 = p_7 p_6 p_5 p_4$$

$$P_2 = p_{11} p_{10} p_9 p_8;$$

$$P_3 = p_{15} p_{14} p_{13} p_{12}$$

I "super" segnali "genera" G_i sono dati da:

$$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

$$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$$

$$G_2 = g_{11} + p_{11} g_{10} + p_{11} p_{10} g_9 + p_{11} p_{10} p_9 g_8$$

$$G_3 = g_{15} + p_{15} g_{14} + p_{15} p_{14} g_{13} + p_{15} p_{14} p_{13} g_{12}$$

Se calcoliamo il tempo per fornire il risultato, tenendo conto del fatto che Δ è il tempo di risposta di AND e OR, esso risulta uguale a 5Δ . Infatti:

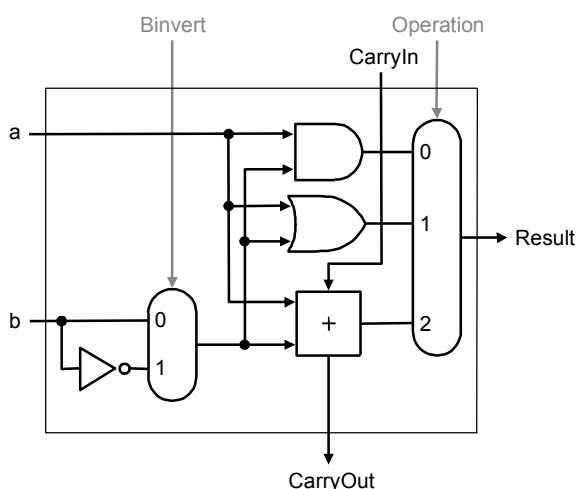
- impiego un tempo Δ per calcolare i p_i e g_i ;
- impiego 2Δ per produrre G_i , in quanto risulta essere una somma di prodotti, e Δ per calcolare P_i : considero dunque il tempo maggiore.
- 2Δ per calcolare i C_i a partire da G_i , P_i e c_0 , in quanto risulta essere una somma di prodotti.

Per 16 bit CLA è 6 volte più veloce di RCA.

CIRCUITO PER LA SOTTRAZIONE.

Se considero i numeri in complemento a 2, la sottrazione non è altro che la somma del numero complementato a cui devo aggiungere 1.

$$a - b = a + (-b) = a + (b_{\text{compl}} + 1).$$



Per fare la sottrazione utilizzo quindi un sommatore in questo modo:

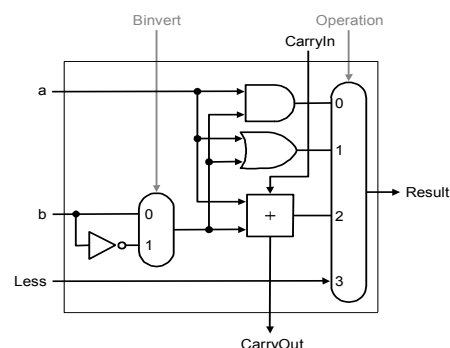
- un ingresso del sommatore rappresenta il numero da cui devo sottrarre: a ;
- l'altro ingresso del sommatore è l'uscita di un multiplexer i cui ingressi sono:
 - b ;
 - b negato;
 - il segnale di controllo che chiamo *Binvert*.

Se voglio dunque fare la somma $\text{Binvert}=0$ e all'ingresso del sommatore entrano a e b .

Se voglio fare la sottrazione $\text{Binvert}=1$ e all'ingresso del sommatore entrano a e $-b$. devo

ricordare che, poiché il numero è rappresentato in complemento a 2, per ottenere $-b$, devo complementare bit a bit e aggiungere 1: l'1 che devo aggiungere entra nel sommatore come CarryIn . Quindi poiché $\text{CarryIn}=1$ solo quando devo fare la sottrazione e $\text{Binvert}=1$ solo quando devo fare la sottrazione, in realtà posso utilizzare una linea sola: $\text{Binvert}=\text{CarryIn}$.

ISTRUZIONE DI SET ON LESS THEN.



È un'istruzione che fa un confronto tra due registri $\$t0$ e $\$t1$ del tipo $\$t0 > \$t1$. Essa sottintende un'operazione aritmetica: dopo aver svolto una differenza, restituisce una word in un registro in cui tutti i bit che vanno dal bit di peso 1 al bit di peso 31 sono identicamente uguali a 0 e il bit di peso 0 può assumere 2 valori distinti:

- 0 se si verifica la condizione di maggiore: $A > B$;
- 1 se si verifica la condizione di minore: $A < B$.

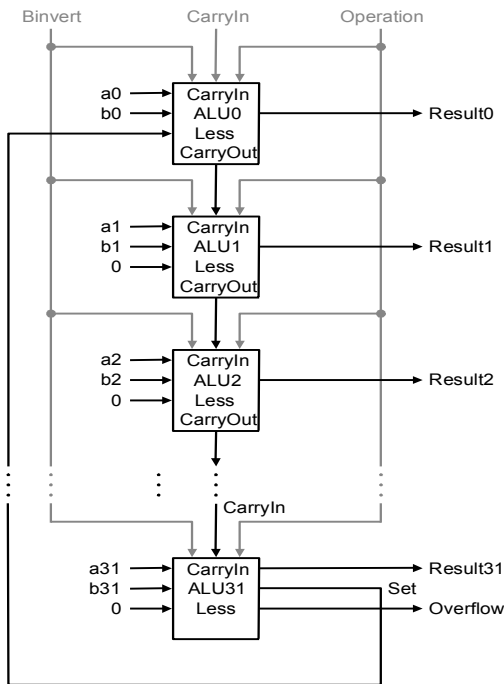
In realtà quindi non sono interessato al risultato della sottrazione, ma solo al segno! Quindi devo concentrarmi

sul bit di segno, cioè sul 31° bit, che è l'indicatore del segno del numero. Tale bit è:

- 0 se $A > B$;
- 1 se $A < B$.

Nella ALU devo quindi aggiungere una linea di less.

In una ALU a 32 bit, la linea di less dal bit 1 al bit 31 sarà identicamente uguale a 0.



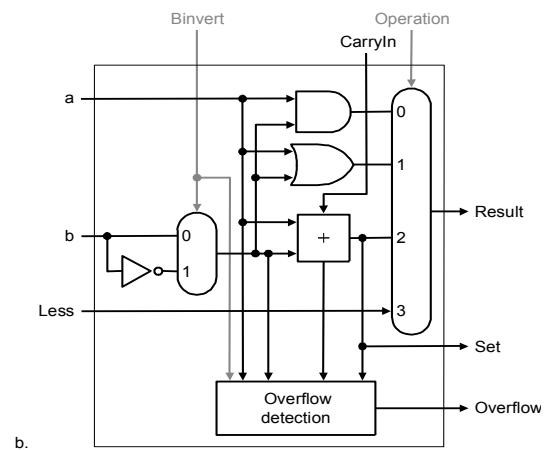
La linea di less del bit di peso 0, deve essere uguale al bit di segno: infatti se la il bit di segno è uguale a:

- 0 $\rightarrow A > B \rightarrow$ la linea di less deve valere 0;
- 1 $\rightarrow A < B \rightarrow$ la linea di less deve valere 1;

Tuttavia il bit di segno non viene fuori come risultato sul bit 31, poiché come risultato esce 0, in quanto il multiplexer ha selezionato l'uscita della linea di less e non del sommatore. Perciò modifico la ALU ad 1 bit che effettua la sottrazione dei bit di peso 31, facendo uscire un segnale di set, che non è null'altro che il bit 31 che risulta dalla sottrazione, che mandiamo alla linea di less del bit di peso 0.

In questo caso sfrutto anche la 4[°] configurazione dei 2 bit in ingresso al multiplexer.

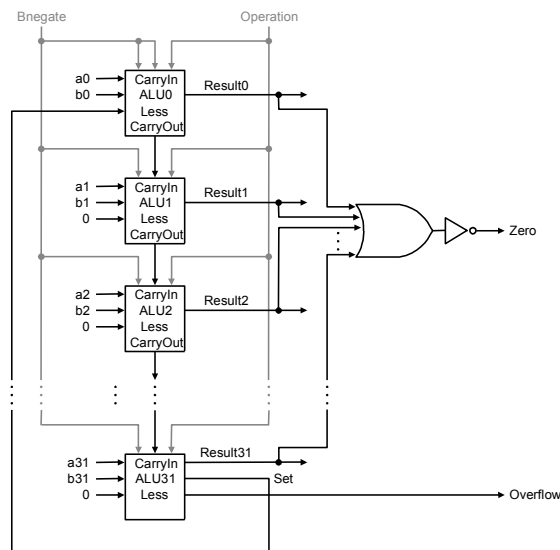
Come si vede dalla figura a fianco, che rappresenta la ALU dei bit di peso 31, il segnale di set è il risultato della sottrazione tra i due numeri, ma non è Result, in quanto il multiplexer ha selezionato la linea di less



OVERFLOW.

Poiché i numeri oggetto di somme e sottrazioni sono pensati essere in complemento a 2, si pone il problema di rivelare o meno la presenza dell'overflow, dopo aver compiuto le operazioni. L'overflow è rivelato dal circuito che tratta i bit di segno più alto, cioè il bit di segno. Il compito di rivelare la presenza di overflow spetta alla ALU dei bit di peso 31.

ISTRUZIONE DI BRANCH IF EQUAL.



Essa sottintende un confronto del tipo $A=B$. Per controllare tale condizione svolgiamo quindi un'operazione di differenza: se il risultato è identicamente uguale a 0, i due numeri sono uguali. Per testare se tutti e 32 i bit in uscita dalle ALU ad 1 bit sono uguali a zero si mandano tutti e 32 i bit del risultato in ingresso ad una porta NOR: se tale porta in uscita fornisce un 1, allora $A=B$, in quanto tutti e 32 i bit in ingresso sono a 0. Se l'uscita di tale NOR è 0, allora almeno 1 tra i 32 bit era diverso da 0, cioè $A \neq B$.

Le linee di controllo sono dunque formate da 3 bit:

- il bit più significativo rappresenta la linea di Binvert=CarrayIn;
- gli altri due bit rappresentano il controllo che seleziona l'uscita.

000 = and

001 = or

010 = add

110 = subtract

111 = slt

Bisogna notare che rimangono delle configurazioni non utilizzate.

CONCLUSIONE .

Possiamo costruire una ALU che supporti il set di istruzioni MIPS:

- l'idea chiave: usare un multiplexer per selezionare le uscite desiderate;
- possiamo svolgere l'operazione di differenza in modo efficiente usando il complemento a 2;
- possiamo replicare la ALU ad 1bit per costruire una ALU a 32 bit.
- Tutte le porte logiche lavorano sempre.
- La velocità di una porta è in qualche modo dipendente dal numero di ingressi alla porta stessa.
- La velocità del circuito dipende dal numero di porte collegate in serie: dobbiamo sempre contemplare il cammino critico, cioè quello che passa attraverso più livelli.
- La nostra principale preoccupazione è la comprensione.

▷CAPITOLO 3◁

▷NUMERI IN VIRGOLA MOBILE◁

Dobbiamo trovare un modo per rappresentare:

- ▷ numeri frazionari, non interi: 3.1416.
- ▷ numeri molto molto piccoli: 0,0000000001.
- ▷ numeri molto molto grandi: $3,78 \times 10^9$.

Per questi numeri si ricorre ad una notazione alternativa, detta notazione scientifica, che ha un'unica cifra a sinistra della virgola moltiplicato per una potenza della base. L'aritmetica dei calcolatori che supporta i numeri come quelli indicati sopra è detta virgola mobile, poiché rappresenta numeri in cui la virgola non è in posizione fissa.

La rappresentazione dei numeri in virgola mobile avviene nel seguente modo:

SEGNO ESPONENTE MANTISSA.

- Aumentando la dimensione della **mantissa** si migliora la sua **accuratezza**;
- Aumentando la dimensione dell'**esponente** si aumenta l'**intervallo dei numeri** rappresentabili.

Il processore MIPS appoggia lo standard IEEE 754 utilizza le seguenti convenzioni:

Singola precisione, su 32 bit:

- 1 bit per il segno;
- 8 bit per l'esponente;
- 23 bit per la mantissa.

I numeri rappresentabili vanno da: $-2^{127} < N < 2^{127}$.

Doppia precisione, su 64 bit:

- 1 bit per il segno;
- 11 bit per l'esponente;
- 52 bit per la mantissa.

Il motivo della rappresentazione dei numeri in segno, esponente e mantissa è da cercarsi nel fatto che, essendo il segno nel bit più significativo è immediato svolgere i confronti del tipo maggiore o minore, in caso di numeri di segno diverso. In caso di numeri con uguale segno, il posizionamento dell'esponente prima della mantissa fornisce una risposta immediata al confronto tra numeri, in quanto i numeri con esponente maggiore sono più grandi dei numeri con esponente minore. Se anche questo test non dovesse fornire una risposta esaustiva, allora si confrontano le mantisse.

Gli esponenti negativi rappresentano un problema per l'ordinamento. Se si usa il complemento a 2 o una qualunque altra notazione in cui gli esponenti negativi hanno un 1 nel bit più significativo del campo esponente, un esponente negativo apparirà come un numero grande. Per esempio:

$$1,0 \times 2^{-1} = 0 \ 11111111 \ 000000000000000000000000.$$

$$1,0 \times 2^{+1} = 0 \ 00000001 \ 000000000000000000000000.$$

La notazione migliore deve quindi rappresentare l'esponente più negativo con 0000...000 e l'esponente più positivo con 1111...111, in modo da avere un ordinamento crescente nei numeri. Per questo per l'esponente si utilizza la **notazione polarizzata**:

- ▷ la polarizzazione per la singola precisione è **127**;
- ▷ la polarizzazione per la doppia precisione è **1023**.

Quindi il numero in virgola mobile è dato da:

$$(-1)^{\text{segno}} \times (1 + \text{Mantissa}) \times 2^{(\text{esponente} - \text{polarizzazione})}$$

Inoltre, poiché in notazione normalizzata bisogna riportare il numero nella forma 0,10011... il numero appena a destra della virgola, in quanto sto utilizzando la base 2, è

per forza un 1. Posso sottintendere questo 1, un modo da avere 1 bit in più per la rappresentazione della mantissa e riportarmi quindi nella forma 1,...piuttosto che 0,...

Esempio 1:

conversione da decimale in virgola mobile: si converte il numero -0,75 in singola e doppia precisione.

In singola precisione:

In decimale: $0,75 = 3/4 = 3/2^2$

In binario: $-0,11 \times 2^0$.

L'esponente, tenendo conto della notazione polarizzata è: $-1+127=126= 0111\ 1110$

In notazione scientifica normalizzata: $-1,1 \times 2^{-1}$.

Il numero in virgola mobile in singola precisione è quindi:

1 0111 1111 1000 0000 0000 0000 0000 0000

La rappresentazione per un numero in singola precisione è:

$$(-1)^{\text{segno}} \times (1+\text{Mantissa}) \times 2^{(\text{esponente}-127)}$$

E quindi:

$$(-1)^1 \times (1+0,1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{(126-127)}$$

In doppia precisione:

In decimale: $0,75 = 3/4 = 3/2^2$

In binario: $-0,11 \times 2^0$.

L'esponente, tenendo conto della polarizzazione è: $-1+1023=1022= 0111\ 1110\ 100$

In notazione scientifica normalizzata: $-1,1 \times 2^{-1}$.

Il numero in virgola mobile in singola precisione è quindi:

1 0111 11110 100 1000 0000 0000 0000 0000 0000 0000...

La rappresentazione per un numero in singola precisione è:

$$(-1)^{\text{segno}} \times (1+\text{Mantissa}) \times 2^{(\text{esponente}-1023)}$$

E quindi:

$$(-1)^1 \times (1+0,1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{(1022-1023)}$$

Esempio2:

Conversione da binario a decimale: quale numero è rappresentato dalla seguente parola?

11000000101000000000000000000000

1 10000001 010000000000000000000000

Il bit di segno è 1 \rightarrow -

Il campo esponente contiene: 1000 0001 = 129. l'esponente è quindi: $129-127=2$

Il campo mantissa contiene: 010000000000000000000000 = $2^{-2}=0,25$. La mantissa è quindi: $1+0,25=1,25$.

Il numero è quindi $-1,25 \times 2^2 = -5$

La virgola mobile è quindi una rappresentazione di numeri che ci permette di poter rappresentare numeri molto grandi e molto piccoli, pagando in complessità e in precisione, in quanto abbiamo un numero di bit predefinito per rappresentare la mantissa di tali numeri. I numeri rappresentabili in virgola mobile vanno da 10^{-308} a 10^{308} .

Esistono tuttavia numeri molto grandi, anche se diversi da infinito, che non possono essere rappresentati. Di conseguenza anche nella rappresentazione in virgola mobile può accadere che si verifichi il problema dell'**overflow**: *esso si verifica quando il numero ha mantissa superiore alla massima mantissa rappresentabile e l'esponente è maggiore del massimo esponente rappresentabile.*

In virgola mobile si presenta anche il problema dell'**underflow**: *si verifica quando si vogliono rappresentare numeri inferiori al minimo numero positivo rappresentabile diverso da zero o superiori al massimo numero negativo rappresentabile.*

L'intervallo di variabilità dell'esponente: l'esponente in singola precisione ha a disposizione 8 bit: $2^8=255$. per l'esponente uso la notazione polarizzata con polarizzazione $2^{N-1}-1=127$. in questo modo l'esponente $-127 < E < +128$.

Se ho l'esponente: 1000 0100=132 → devo togliere la polarizzazione: $132-127=+5$.

Se ho l'esponente: 0111 1110=126 → devo togliere la polarizzazione: $126-127=-1$.

ISTRUZIONI MIPS PER LA VIRGOLA MOBILE.

Il processore MIPS mette a disposizione 32 registri per la virgola mobile, identificabili con **\$f** e il numero identificativo del registro, da 1 a 31. Essi richiedono quindi 5 bit per essere indirizzati.

Le istruzioni per la virgola mobile sono di due tipi:

.s → identificano numeri in singola precisione;

.d → identificano numeri in doppia precisione.

La sintassi per le istruzioni in virgola mobile è uguale a quella per gli interi:

add.s \$f0, \$f1, \$f2

\$f0=\$f2+\$f1

Per i numeri in doppia precisione ho bisogno di 64 bit, quindi di 2 registri: si utilizza la convenzione di utilizzare nelle istruzioni solo registri identificati con numeri pari: tali registri in realtà sottintendono anche l'utilizzo del registri successivo:

add.d \$f0, \$f2, \$f4

\$f0=\$f2+\$f4

\$f0 → \$f0 e \$f1.

\$f2 → \$f2 e \$f3.

\$f4 → \$f4 e \$f5.

Quindi in doppia precisione un registro corrisponde ad una coppia pari e dispari, identificato dal nome del registro pari.

ISTRUZIONI ARITMETICHE	
Singola precisione	Doppia precisione
add.s \$f1, \$f2, \$f3	add.d \$f0, \$f2, \$f4
sub.s \$f1, \$f2, \$f3	sub.d \$f0, \$f2, \$f4
mul.s \$f1, \$f2, \$f3	mul.d \$f0, \$f2, \$f4
div.s \$f1, \$f2, \$f3	div.d \$f0, \$f2, \$f4

Il MIPS mette quindi a disposizione 32+32 registri: tuttavia per indirizzare questi registri occorrono sempre 5 bit, in quando essi sono utilizzati in condizioni diverse e tali condizioni sono specificate nel codice operativo dell'istruzione. Se il codice operativo identifica un'istruzione che lavora con numeri interi, i registri utilizzati saranno i 32 dedicati agli interi, che necessitano di 5 bit per essere indirizzati. Se il codice operativo identifica un'istruzione che lavora con numeri in virgola mobile, i registri utilizzati saranno i 32 dedicati alla virgola mobile, che necessitano di 5 bit per essere indirizzati.

ISTRUZIONI DI TRASFERIMENTO DATI	
lwc1 \$f1,100(\$s2)	swc1 \$f1,100(\$s2)

\$f1=Memoria [\$s2+100]

Memoria [\$s2+100]= \$f1

In queste istruzioni il puntatore alla memoria è un registro dedicato agli interi, mentre il dato da scrivere/leggere è un numero in virgola mobile.

ISTRUZIONI COMPARE	
c.eq.s \$f1, \$f4	c.eq.d \$f2, \$f4
c.neq.s \$f1, \$f4	c.neq.d \$f2, \$f4
c.lt.s \$f1, \$f4	c.lt.d \$f2, \$f4
c.le.s \$f1, \$f4	c.le.d \$f2, \$f4
c.gt.s \$f1, \$f4	c.gt.d \$f2, \$f4
c.ge.s \$f1, \$f4	c.ge.d \$f2, \$f4

eq: equal;

neq: not equal;

lt: less than;

le: less or equal than;

gt: greater than;

ge: greater or equal than;

c.lt.s \$f1, \$f4 → se \$f1<\$f4, cond=1, altrimenti cond=0.

Nel fare questi confronti viene modificato un *flag* all'interno della CPU, in modo che le istruzioni di salto, saltino a seconda che il flag sia vero o falso.

ISTRUZIONI DI SALTO	
bclt 25 Va a PC+4+(25 x 4)	Salta se FP (floating point) è vero.
bclf 25 Va a PC+4+(25 x 4)	Salta se FP (floating point) è falso.

Un grosso problema per la rappresentazione dei numeri in virgola mobile è la precisione, in quanto ho un numero limitato di bit per rappresentare la mantissa e quindi devo troncare il numero. Per riuscire ad avere minimi errori di rappresentazione, in realtà il processore, usa per svolgere i calcoli interni 2 ulteriori bit, detti guardia e arrotondamento, che rappresentano le 2 cifre dopo quelle messe a disposizione dai bit riservati alla mantissa nel caso della singola e doppia precisione. In questo processore si può scegliere tra 4 diverse modalità di arrotondamento.

Immagino di dover eseguire la somma: $235+2,56$. per eseguire questa somma devo allineare i numeri:

$$\begin{array}{r} 235,00 + \\ \underline{2,56} = \\ 237,56 \end{array}$$

Se ho a disposizione solo 3 bit, la somma vale 237: ho dovuto troncare il risultato, commettendo un errore di 0,56.

CASI PARTICOLARI.

Ci sono casi in cui applicando la formula dei numeri in virgola mobile:

$$(-1)^{\text{segno}} \times (1 + \text{Mantissa}) \times 2^{(\text{esponente} - \text{polarizzazione})}$$

non riesco ad ottenere alcuni casi particolari, che tuttavia devono essere ben rappresentati e riconoscibili.

Per ottenere questi casi utilizzo delle configurazioni particolari di esponenti e mantissa.

Infatti in teoria:

- l'esponente dei numeri in singola precisione avrebbe variabilità da 0 a 255; polarizzando, quindi con la costante 127, l'intervallo di variabilità per l'esponente è $-127 < E < +128$.
- l'esponente dei numeri in doppia precisione avrebbe variabilità da 0 a 2047; polarizzando, quindi con la costante 1023, l'intervallo di variabilità per l'esponente è $-1023 < E < +1024$.

In particolare, però, affermo che:

- nel caso in cui l'esponente è compreso tra 1 e 254, cioè $-126 < E < +127$, applico la formula dei numeri in virgola mobile.
- Nel caso in cui l'esponente è rappresentato con tutti 0, cioè rappresenta 0, o con la polarizzazione -127 (-1023), e la mantissa è identicamente uguale a 0, non applico la formula, ma tale numero rappresenta lo 0.
- Nel caso in cui l'esponente è rappresentato con tutti 1, cioè rappresenta 255, o con la polarizzazione +128 (1024), e la mantissa è identicamente uguale a 0, non applico la formula, ma tale numero rappresenta $\pm\infty$.
- Nel caso in cui l'esponente è rappresentato con tutti 1, cioè rappresenta 255, o con la polarizzazione +128 (1024), e la mantissa è diversa 0, non applico la formula, ma tale numero rappresenta un NaN.

Singola Precisione		Doppia Precisione		Oggetto rappres.
Exp.	Mantissa	Exp.	Mantissa	
0	0	0	0	0
0	# ≠ 0	0	# ≠ 0	Sub-number
1÷254	∀ #	1÷2046	∀ #	#normalizzato
255	0	2047	0	$\pm\infty$
255	# ≠ 0	2047	# ≠ 0	NaN (Not a Number)

PROBLEMA 1: La rappresentazioni in virgola mobile secondo lo standard IEEE 754 impone che il numero sia normalizzato nella forma $1, \dots$ questo ci permette di poter contare su un bit aggiuntivo per la mantissa: sottintendendo il primo 1 è come se i bit della mantissa fossero 24 e non 23. Questo, tuttavia, porta immediatamente ad un problema: la **rappresentazione dello 0**. Tale numero, infatti, dovrebbe essere rappresentato nella mantissa con tutti 0: ma poiché normalizziamo il numero nella forma $1, \dots$ il numero che corrisponde alla mantissa identicamente uguale a 0, è applicando la formula, 1 e non 0, in quanto devo sempre aggiungere l'1 sottinteso!

SOLUZIONE: Nel caso in cui l'esponente è rappresentato con tutti 0, cioè rappresenta -127 , e la mantissa è identicamente uguale a 0, non applico la formula, ma tale numero rappresenta lo 0.

PROBLEMA 2: Può capitare, talvolta, che il risultato di una divisione sia $\pm\infty$, dividendo, per esempio, un numero reale con il numero 0. si pone il problema della **rappresentazione di "infinito"**.

SOLUZIONE: Nel caso in cui l'esponente è rappresentato con tutti 1, cioè rappresenta 255, o con la polarizzazione +128 (1024), e la mantissa è identicamente uguale a 0, non applico la formula, ma tale numero rappresenta $\pm\infty$.

PROBLEMA 3: Ci sono delle forme che matematicamente rappresentano le cosiddette "forme indefinite". Siamo nella situazione, in cui, per esempio dividiamo $0/0$, oppure in cui dobbiamo svolgere l'operazione di $+\infty-\infty$. Tali situazioni nell'aritmetica dei calcolatori vengono rappresentati con il simbolo NaN, cioè "Not a Number". Come si **rappresentano i NaN?**

SOLUZIONE: Nel caso in cui l'esponente è rappresentato con tutti 1, cioè rappresenta 255, o con la polarizzazione +128 (1024), e la mantissa è diversa 0, non applico la formula, ma tale numero rappresenta un NaN.

PROBLEMA 4:

Condizione perchè si verifichi **overflow**: quando superiamo il numero che ha massima mantissa ($=1+\text{tutti } 1=2$) e massimo esponente, cioè 2^{127} , in quanto l'esponente 128 è riservato per rappresentare $\pm\infty$, oppure NaN. Quindi il massimo numero a cui tende la rappresentazione è $2 \times 2^{127} = 2^{128}$. I numeri rappresentabili, affinché non si verifichi overflow sono $< 2^{128}$, per i positivi.

Condizione perchè si verifichi **underflow**: quando superiamo il numero che ha minima mantissa ($=1+\text{tutti } 0=1$) e minimo esponente, cioè 2^{-126} , in quanto l'esponente -127 è riservato per rappresentare 0. Quindi il minimo numero a cui tende la rappresentazione è $1 \times 2^{-126} = 2^{-126}$. Vorrei rappresentare numeri più piccoli di 2^{-126} . Di fatto, applicando la formula non posso ottenere numeri, nella mantissa minori di 1, in quanto il numero minimo è rappresentato da tutti 0, a cui però devo aggiungere l'1 sottinteso.

SOLUZIONE: posso fare riferimento a numeri, detti **sub-normali**, che si basano su una regola di rappresentazione diversa da quella che applica la formula. Quando ho l'esponente minimo, cioè 0, o con la polarizzazione -127 (-1023), e la mantissa è diversa da 0, non applico più la regola, cioè non sottintendo più l'1. Con questo tipo di convenzione, quindi, tenendo conto che il minimo numero rappresentabile nella mantissa è $0, \dots, 01$, cioè 2^{-23} , e che l'esponente è 2^{-126} , riesco a rappresentare numeri fino a $2^{-23} \times 2^{-126} = 2^{-149}$. In questo modo riesco a rendere graduale il problema dell'underflow, cioè riusciamo a rappresentare numeri minori di 2^{-126} , pagando però il fatto che l'implementazione è più complessa e una minore precisione, perché rappresentiamo 2^{-149} con 1 solo bit significativo. Infatti numeri compresi fra 2^{-126} e 2^{-149} , non useranno tutti i bit riservati dalla mantissa, poiché alcuni dovranno essere a 0.