Architetture e reti di calcolatori 0081809INGINF05

Indice del corso

1 Intro

- Intro -

2 Introduzione alla progettazione dei computer

Nella realizzazione di un calcolatore è necessario considerare due parti fondamentali (su cui vengono effettuate continue ricerche):

- tecnologia: in avanzamento veloce, sempre più veloci ed efficienti con costi ridotti
- idee: praticamente **immutate** dalle prima macchine realizzate. (le ricerche provano a trovare macchine basati su concezioni di idee innovative)

Il primo prototipo di macchina programmabile nasce nel 1837 da Charles Babbage (Macchina Analitica), contenente da tutte le funzioni principali (programmata con schede perforate). Effettuava calcoli in controllo automatico.
Nel **1936 Alan Turing** definisce matematicamente e risponde alle domande fondamentali:

- che cos'è un "calcolatore"?
- quali sono i problemi che può risolvere?

Il concetto di computer è legato al concetto di algoritmo risolutivo di un problema, si definiscono le caratteristiche principali.

Caratteristiche principali dell'algoritmo:

- la lista di azioni è finita (DEVONO ESSERE FINITE);
- ogni azione è comprensibile per la macchina senza ambiguità;
- ogni azione è eseguibile da parte della macchina;
- le azioni vengono svolte in sequenza a partire dalla prima;
- la lista può comprendere azioni, dette di salto, che cambiano l'ordine di esecuzione passando all'esecuzione di un'azione della lista diversa dalla successiva;
- la lista include una azione di arresto che ferma l'esecuzione da parte della macchina.

Altre nozioni:
Computazione algoritmica: esecuzione della lista di azioni dell'algoritmo da parte della macchina, in sequenza a partire dalla prima. Le azioni sono scandite da intervalli di tempo (istanti), parti possono essere ciclicamente ripetute (anche all'infinito).
Problema algoritmicamente risolubile: significa che tale problema può essere risolto da un algoritmo per un QUALSIASI insieme di dati (generici).

Nel 1936 Turing definisce la Macchina di Turing, capace di eseguire computazione algorima, di importanza storica perché definisce (con modelli matematici PRECISI):

- I principi alla base della computazione algoritmica
- i requisiti minimi necessari per costruire un qualunque dispositivo che sia capace di effettuare una computazione algoritmica

La macchina assume un numero FINITO di stati e può scrivere o leggere su un nastro un simbolo alfabetico, il nastro è infinito ma la sequenza di simboli è finita.
La macchina effettua transizioni di stato in istanti di tempo. Lo stato successivo dipende dallo stato

attuale e dal simbolo letto dal nastro. Si può anche scrivere sul nastro il risultato della computazione.

> E' matematicamente dimostrabile che è possibile codificare un qualsiasi algoritmo con I simboli alfabetici e definire una Macchina di Turing che esegua l'algoritmo codificato
>
> QUINDI
>
> La macchina di Turing può eseguire qualsiasi algoritmo

La macchina vendo solo un numero finito di stati ha un numero finito di azioni base, come i computer attuali.
La macchina di Turing è di difficile realizzazione nel 1945 John Von Neumann definisce il modello della macchina di Von Neumann contenente memoria (codifica dell'algoritmo in lettura e dati in r/w), e altri dispositivi di calcolo. Tale macchina è di più facile realizzazione contenendo però tutte le idee della macchina di Turing.

Teoria della computazione:

1. Teorema 1: la macchina di Turing e di Von Neuman hanno la stessa potenza computazionale, risolvono lo stesso sottoinsieme dei problemi, cioè l'insieme dei problemi algoritmicamente risolvibili
2. Teorema 2: esistono problemi che NON sono algoritmicamente risolvibili, non esistono algoritmi che restituiscono soluzione

2.1 Linguaggi

Importante negli algoritmi è che le azioni NON devono essere ambigue, per fare ciò si utilizzano i **linguaggi formali** dove le informazioni sono rappresentate senza ambiguità.
Un linguaggio formale utilizzato per la codifica di algoritmi è detto **Linguaggio di Programmazione.**

- ALGORITMO scritto in un Linguaggio di Programmazione è detto PROGRAMMA,
- una AZIONE scritta in un Linguaggio di Programmazione è detta ISTRUZIONE.

Le macchine possono comprendere solo il linguaggio macchina, direttamente connesso alla struttura fisica del PC, ogni computer ha il suo.
I linguaggio macchina è molto diverso dal linguaggio dell'uomo detto linguaggio naturale.
I Linguaggi ad Alto livello sono più facili da utilizzare dall'uomo ma non comprensibili alle macchine, per tradurli è necessario utilizzare il **COMPILATORE**.

2.2 Hardware

Componenti FISICHE del computer. Gestisce informazioni esterne (input), le elabora (ALU), memorizza, fornisce elaborazioni esterne (output), controlla l'esecuzione nel tempo delle funzioni (unità do controllo CU).
La CPU è la fusione di unità aritmetico logica (ALU) e CU. La comunicazione tra le parti avviene tramite BUS (Binary Unit System). Le esecuzioni delle operazioni sono scandite nel tempo dal CLOCK. Durante l'esecuzione degli algoritmi si passa di stato nel tempo, il clock scandisce tali passaggi.

2.3 Software

Insieme dei programmi (algoritmi) residenti in memoria. Distinti in Software Utente e Software di Sistema. I software utenti risolvono problemi specifici. Il software di sistema è diviso in:

- Sistema operativo
  - gestisce l'hardware assegnando i programmi da eseguire
  - interagisce con l'utente
- Compilatore: traduce da linguaggi ad alto livello in equivalenti a linguaggio macchina

3 La Gerarchia dei Linguaggi di programmazione e i Registri

3.1 Linguaggio macchina

Una macchina sa eseguire SOLO il linguaggio macchina (numero finito di azioni), un computer viene progettato a partire dal linguaggio macchina, ovvero la sua architettura.
Principalmente esistono due approcci di progettazione:

- Architetture CISC (Complex Instruction Set Computer), istruzioni complesse. Una istruzione effettua varie azioni, più lente, hardware complesso, programmi più piccoli.
- Architetture RISC (Reduced Instruction Set Computer), [ristrette più che ridotte rispetto a CISC], istruzioni semplici e veloci, hardware più semplice, programmi più grossi

Tutti i linguaggi macchina sono sequenze binarie, bit (binary digit). La lunghezza delle istruzioni influenza l'hardware. Nel MIPS sono TUTTE lunghe 32 bit, assieme ai dati (32 bit).
La distinzione tra dati e istruzioni nasce da come il computer usa e preleva le sequenze.
Un programma in un linguaggio macchina è utilizzabile SOLO sui computer che eseguono tale linguagio.

3.2 Linguaggio Assembly

Il linguaggio Assembly (linguaggio formale) permette di NON codificare un programma in binario ma con simboli alfanumerici inglesi e numeri in decimale. L'assebler è un programma che traduce i simboli in linguaggio macchina. Ad una istruzione assmbly corrisponde una istruzione macchina. Ogni computer ha il suo Assembly, la progettazione dell'architettura parte dal linguaggio assembly e connessione macchina, ha gli stessi svantaggi del linguaggio macchina, è più facile da leggere perché separa l'istruzione e i dati ma è comunque complesso gestire un programma in Assembly.

3.3 Linguaggi ad Alto Livello

I linguaggi ad alto livello sono linguaggi formali che sono più facili da usare perché hanno regole più semplici e più vicine all'uomo. Sono tipicamente più brevi e sono indipendenti dal linguaggio macchina, sono stati creati i compilatori per tradurre i linguaggi da alto livello a macchina (in molti casi generano prima il codice assembly che viene tradotto dall'assembler). Attualmente l'assembly viene usato per applicazioni particolari di ottimizzazione (soprattutto su sistemi embedded).

3.4 Registri

Il registro è un dispositivo di memorizzazione per memorizzare istruzioni e dati e per elaborarli.
Il registro è l'unità MINIMA di memorizzazione, possono esserci:

- registri isolati: collegati a dispositivi con compiti particolari. Vengono usati automaticamente per compiti SPECIFICI. Ogni registro ha un nome in base al suo compito es Program Counter (PC)
- blocchi di registri: più registri in sequenza, ogni registro ha un indirizzo per essere individuato in modo UNIVOCO, se non lo si ha non si può accedere alla informazione. Di fissata lunghezza e di numero finito.

I registri sono numerati da 0, se si hanno N registri si hanno gli indirizzi da 0 a N-1.
L'accesso a un registro può essere fatto in lettura (SENZA distruggere il contenuto) e scrittura (SOSTITUZIONE).
La memoria principale contiene un blocco di registri separato da quelli del processore collegato da BUS, molto più grande ma lenta, usato per memorizzare dati e programmi.
I registri del processore sono pochi ma veloci ed utilizzati per eseguire istruzioni.

3.5 Registri del processore MIPS

La lunghezza dei registri del MIPS è 32bit, i registri sono a sua volta 32, anche se alcuni di essi sono rieservati e non direttamente utilizzabili. Nell'Assembly MIPS per indicare **l'indirizzo di un registro del processore si usa il "$"** seguito da due caratteri alfanumerici per distinguere i registri in base allo scopo.
I registri per i dati sono denominati da $t0 a $t9 e $s0 a $s9. Altri registri sono riservati per chiamata di procedura. L'indirizzo $zero contiene la costante 0 (tutti i bit a 0), NON modificabile.

4 Le Istruzioni Aritmetiche dell'Assembly MIPS

4.1 Sintassi dell'assembly MIPS

Le istruzioni sono dette comandi eseguiti ciecamente, seguono regole precise (linguaggio formale).
La macchina non guarda la semantica (significato), quello lo deve conoscere il programmatore.
L'assembly MIPS usa simboli inglesi e numeri decimali, ogni istruzione è su una riga. La sequenza alfanumerica iniziale è il codice operativo (istruzione), gli altri simboli sono i dati e/o registri.
I commenti iniziano per '#' (da lì in poi l'assembler ignora la riga e va alla successiva)

4.2 Addizione: Istruzione add

Per una medesima operazione sono presenti più istruzioni (es se tra registri o in memoria), l'addizione può essere *add* (registri) oppure *addi* (immediata). L'add somma interi con segno.

```
add $t0, $s1, $s2
$t0 destinazione
$s1 - $s2 addendi
```

Solo $t0 viene modificato, gli addendi restano invariati. I dati degli addendi devono essere già caricati.

E’ possibile impostare un registro sia come destinazione che come addendo (o anche come destinazione ed entrambi gli addendi)

4.3 Addizione: istruzione addi

Identica ad *add*, ma un addendo è dato come valore immediato invece che tramite registro.

```
addi $t1, $s0, 350
$t1 destinazione
$s0 addendo
350 (numero con segno immediato)
```

Come l’istruzione precedente viene modificato solo il registro destinazione, il registro contenente l’addendo resta invariato. Un addendo può esere usato anche come destinazione.

4.4 Sottrazione: istruzione sub

Effettua una sottrazione tra operandi interi con segno, analogo ad *add*.

```
sub $t0, $s1, $s2
$t0 risultato, destinazione
$s1 minuendo (l’ordine del registro è importante, non vale la prop. commutativa)
$s2 sottraendo (l’ordine del registro è importante, non vale la prop. Commutativa)
```

Non esiste in Assembly una sottrazione con operando costante, si può utilizzare l’add con l’opposto della costante (ricordare che l’idea è avere un set di istruzioni semplificato).

Presenti esempi importanti al termine della lezione

5 Notazione posizionale pesata

5.1 Sequenze binarie

Abbiamo due tipi di informazioni da registrare: i dati e le istruzioni. Entrambe sono salvate tramite segnali elettrici (alti e bassi) riconducibili a due simboli dell’alfabeto {0 (basso),1 (alto)}. I due simboli sono associati ad intervalli con valori di soglia distinti tra massimi e minimi.
Usando solo due simboli le informazioni salvate occupano molto più spazio ma si evita incertezze che possono verificarsi all’aumentare del numero di livelli utilizzati (es oscillazioni).

Sequenza di 1 cifra binaria è detta bit (binary digit);
Sequenza di 8 cifre binarie è detta byte (cioè 8 bit);
Sequenza di 32 cifre binarie è detta parola (word, 32 bit, 4 byte).

Sono I blocchi di memoria MINIMI a cui la macchina riesce ad accedere.
I dati sono elaborati salvando i dati tramite apposite notazioni (con regole identiche tra le varie architetture es naturali, reali, razionali/irrazionali). Dato che in un computer tutto è finito è possibile rappresentare solo sequenze di lunghezze finite, es i valori interi sono compresi tra massimo e minimo e quelli frazionari hanno una certa precisione massima che in molti casi porterà ad un errore di approssimazione.

5.2 Notazione posizionale pesata: regole di rappresentazione

La notazione pesata in base B (generica) rappresenta interi positivi, il pedice indica la base B.

Regole:

1. sequenza finita di cifre nell'alfabeto {0, 1, 2, 3, …, B-1} (nella vita quotidiana è implicito B=10)
2. ogni cifra è moltiplicata per un peso diverso associato alla posizione che occupa nella sequenza (es 589 = 5*10^2 + 8 * 10^1 + 9*10^0; ricordare che B^0 = 1 per definizione)
3. i pesi sono tutte le potenze della base B a partire da B0 in ordine crescente da destra verso sinistra
4. il valore del numero è dato dalla somma di tutte le cifre moltiplicate per i relativi pesi.

A destra abbiamo la cifra più significativa, a sinistra la meno significativa. Una cifra identica di simboli riportata in basi diverse corrisponde a valori diversi (decimale, binaria etc.), se formata da solo 0 è 0 se formata da tutti 0 con la cifra meno significativa a 1 è 1.

5.3 Metodi di conversione di base

Conversione tramite metodo aritmetico (da base B=X a base B=10)

132_5 = 1 * 5^2 + 3 * 5^1 + 2 * 5^0 = 1*25 + 3*5 + 2*1 = 42

Il numero per cui moltiplicare i valori varia con la base in questo caso si passa da B=5 a B=10.

Metodo delle divisioni successive: (da base B=10 a base B=X)

1. si effettua una prima divisione del numero per la nuova base B
2. si ripete sul quoziente ottenuto la divisione per B fino a quando il quoziente è maggiore o uguale a B.

La sequenza delle cifre della rappresentazione nella nuova base B si costruisce a partire dalla cifra più significativa considerando l'ultimo quoziente minore di B e tutti i resti procedendo dall'ultimo fino al primo.

Da 90_{10} a base 8:

90/8 = 11 resto 2

11/8 = 1 resto 3

1<8 – Mi fermo.

Costruisco il numero usando i resti con la cifra pià significativa come ultima divisione: 132_8

5.4 Rappresentazione in base 2

Le conversioni da/in base 2 seguono gli stessi passi già descritti sopra, ovviamente esistono solo i valori 0 ed 1. E' presente la proprietà per cui se la cifra MENO significativa è 0 il numero è pari, se 1 il numero è dispari (i contributi dopo la prima cifra sono moltiplicati per la base 2 e quindi sarà un numero sicuramente pari, solo la cifra meno significativa è dispari e la relativa somma con gli attributi pari darà un numero dispari).

Proprietà combinatorie fondamentali:

Fissato una lunghezza N di sequenze binarie {0,1} si ha:

- I valori vanno da 0 ad 2^N-1

- Le combinazioni diverse sono 2^N

5.5 Contenuto ed indirizzi dei registri nell'Architettura MIPS

Il processore MIPS fissa la lunghezza dei valori dei registri del processore come sequenze di 32 bit. Di conseguenza è possibile avere 2^N combinazioni diverse e in notazione posizionale pesata valori da 0 a 2^N-1 circa 4 miliardi (ricordare che tutti 0 è una combinazione, per questo si perde un valore).
Se una operazione porta un valore maggiore si verifica un errore di overflow (traboccamento), che porta ad errori nella esecuzione del programma, il processore ha sistemi di controllo per tale errore.
Il numero di registri del processore è 32, di conseguenza per indicarli sono necessari indirizzi a 5 bit (2^5 = 32 COMBINAZIONI, non valori)

6 Notazione in complemento a 2

6.1 Addizione di interi positivi

L'addizione in qualsiasi base si effettua sommando i valori sulle singole posizioni della sequenza calcolando la somma degli operandi ed il riporto (la parte più significativa).
Regola dell'addizione in notazione posizionale pesata in qualsiasi base:

1. si pone uguale a 0 il riporto sulla posizione della cifra meno significativa (più a destra);
2. procedendo da destra verso sinistra, su ogni posizione si esegue la somma delle relative tre cifre degli operandi e del riporto;
3. la cifra meno significativa della somma ottenuta fornisce la cifra del risultato relativo alla posizione, e quella più significativa fornisce il riporto sulla posizione successiva;
4. nella posizione più significativa (più a sinistra) la cifra calcolata come riporto sulla posizione successiva non è utilizzata nella rappresentazione del risultato.

In base due gli operandi sono 3 (i due addendi ed il resto) quindi 2^3 combinazioni:

- Tutte e 3 a 0 = 0 (00)
- Una cifra 1 le altre 0 = 1 (01)
- Due cifre 1 e l'altra 0 = 2 (10)
- Tutte a 1 = 3 (11)

6.2 Notazione in modulo e segno

E' la notazione con cui gestiamo i numeri negativi, abbiamo il segno separato dal valore (modulo).
In binario può essere usato il bit più significativo per salvare l'informazione relativa al segno (bit di segno), usando lo 0 per i numeri positivi e il valore 1 per i negativi.
Se i segni sono uguali il risultato è la somma, se i segni sono discordi il risultato è la differenza tra il numero con il modulo maggiore e quello con modulo minore; il segno è dato dal numero maggiore.
Per evitare tali controlli e semplificare l'hardware dell'ALU si utilizza la notazione in complemento alla base B (notazione in complemento a due nei computer), per velocizzare e diminuire la complessità.

6.3 Notazione in complemento a 2: regole di rappresentazione

Si basa su sequenze di lunghezza fissa. Si usano le stesse regole di rappresentazione della notazione

posizionale pesata ma la cifra più significativa ha un peso NEGATIVO mentre gli altri hanno peso positivo come già visto precedentemente.
Il valor negativo varia in base alla lunghezza della sequenza e dimezza il numero di valori rappresentabili nella sequenza di bit presenti. Non si ha modulo e segno separati ma congiunti nel calcolo del valore.
111101 = 1x(−25) + 1x24 + 1x23 + 1x22 + 0x21 + 1x20 = −32 + 16 + 8 + 4 + 0 + 1 = −32 + 29 = −3

Proprietà del bit più significativo: il bit più significativo determina il segno: se 0 positivo, se 1 negativo

6.4 Addizione di interi in complemento a 2

Oltre ad eliminare il problema di analizzare i segni ed i moduli per stabilire come effettuare l'addizione si ha:

Proprietà dell'addizione:
La somma di due numeri con segno si effettua SEMPRE sommando le sequenze binarie che li rappresenta, senza dovere valutare modulo e segno

6.5 Intervallo dei valori rappresentabili in complemento a 2

Usando sempre lunghezze finite si hanno valori compresi tra massimo e minimo

Sequenze di lunghezza N, rappresentazione in complemento a 2
INTERVALLO DI RAPPRESENTAZIONE = [-2^{N-1} , $2^{N-1} - 1$]
Minimo negativo -2^{N-1}
Massimo positivo $2^{N-1} - 1$

Dato che il minimo è sul bit più significativo SICURAMENTE è più grande dei numeri positivi.

Il valore -1 è rappresentato da una serie di tutti bit a valore 1, indipendentemente dalla lunghezza

Se l'operazione porta ad avere un risultato troppo grande non rappresentabile si ha errore di overflow.

7 Proprietà della Notazione in complemento a 2

7.1 Regola per la rappresentazione dell'opposto di un numero

La notazione in complemento a due permette di effettuare addizioni senza considerare il modulo e il segno degli operandi e gode di ulteriori proprietà.

Proprietà per ottenere l'opposto del numero

- Complementare la sequenza del numero bit a bit (scambiare gli 0 con 1 e viceversa)
- Sommare alla sequenza ottenuta il valore 1

Per verificare la correttezza è possibile sommare ad un numero il suo opposto e verificare che la relativa somma sia 0 (ignorando il riporto finale).

7.2 Sottrazione come somma con l'opposto

Utilizzando la notazione in complemento a due è possibile effettuare la sottrazione come somma di un numero per l'opposto dell'altro (sottrazione come somma con l'opposto)

$$A+B=A+(-B)$$

In questo modo si utilizza lo stesso hardware (circuito) per fare l'addizione con piccole modifiche.
Nei linguaggi ad alto livello si usa la dichiarazione del tipo per indicare come salvare ed usare i dati in una variabile (se con o senza segno, es *int* del C/C++).

7.3 Regola per riconoscere l'Overflow

La notazione in complemento a due definisce la lunghezza della sequenza sulla quale effettuare i calcoli (compresi tra massimo e minimo). L'intervallo di una lunghezza a N bit è $[-2^{N-1}, 2^{N-1}-1]$.
Se il calcolo di un numero esce da tale intervallo si ha un errore di overflow (traboccamento) e la sequenza nel registro NON è corretta. Per rappresentare tale numero sarebbe necessario utilizzare più bit di quelli effettivamente disponibili.
È molto importante riconoscere tale tipo di errore altrimenti il programma porterebbe ad avere risultati errati, esistono due casi:

- Numeri con segni diversi: impossibile andare in overflow
- Numeri con segno uguale: possibile overflow

Si ha la seguente regola:

Regola per riconoscere l'overflow
Indicando i due resti con:

- C_{N-1} (riporto su bit più significativo della sequenza)
- C_N (riporto non usato, riporto della cifra successiva della lunghezza massima della sequenza)

Se i riporti sono uguali il risultato è corretto, se i due riporti sono DIVERSI si ha l'errore di overflow:

$$C_N \neq C_{N-1}$$

Tale regola riconosce anche il caso in cui si effettua l'opposto del minimo numero negativo -2^{N-1} che non è rappresentabile sulla stessa sequenza di bit come valore positivo (vedere l'intervallo massimo).

7.4 Regola per l'estensione del segno

Il valore di un numero scritto in complemento a 2 è legato al peso negativo in base alla cifra più significativa, l'estensione del segno serve per mantenere il valore invariato allungando il numero di bit con cui la cifra è rappresentata. Molto usata nelle istruzioni di trasferimento dati dove numeri da 16bit sono convertiti a 32bit da dare all'ALU (lezioni successive)

Regola per estensione del segno
Aggiungendo a sinistra della sequenza un numero qualsiasi di bit tutti uguali al bit di segno il valore rappresentato dalla sequenzaresta invariato (se positivo 0, se negativo 1).

Gli standard di rappresentazione dei numeri sono stati standardizzati nelle varie famiglie di processori per facilitare la portabilità dei programmi. Il MIPS mediante queste convenzioni riesce ad avere un hardware semplificato che utilizza lo stesso circuito per effettuare più operazioni

8 Formato di Tipo R per add e sub e Indirizzamento tramite registro

8.1 Formati e modalità di indirizzamento

Le istruzioni binarie che costituiscono il linguaggio macchina devono essere strutturate in modo che siano presenti:

- tutte le informazioni necessarie ad eseguire l'istruzione
- indicare dove reperire i dati su cui eseguire tale operazione.

Ogni istruzione a 32 bit (nel MIPS sono tutte fisse a 32bit non è così nei CISC) è completa e non necessita di altre informazioni aggiuntive per essere eseguita, è una caratteristica fondamentale di progettazione dell'architettura.
Tutte le operazioni hanno due nozioni fondamentali alla base di tutti i linguaggi macchina, sono necessarie entrambe e sono alla base per la definizione dell'architettura del calcolatore:

- Formato della istruzione
- Modalità di indirizzamento

Ogni istruzione è suddivisa in campi (sotto sequenze) in cui sono rappresentate le informazioni in modo univoco. La divisione è prettamente logica e non fisica e i vari campi occupano sempre la stessa posizione ed hanno la stessa lunghezza in base alla tipologia di formato a cui appartengono.

Il **formato** di una istruzione in linguaggio macchina è dato dalla struttura dei campi (sotto sequenza) nella sequenza dei bit e ne specifica, per ogni campo, la lunghezza e la posizione.

La modalità di indirizzamento invece indica come reperire i dati della istruzione usando i campi specificati (cioè come usare le informazioni dei campi per ottenere le informazioni).
Tutti i registri e le locazioni di memoria hanno un indirizzo, se non si conosce tale indirizzo non è possibile accedere alla informazione in nessun modo.

La MODALITÀ DI INDIRIZZAMENTO
di un'istruzione in Linguaggio Macchina stabilisce le regole per determinare gli indirizzi dei registri coinvolti nell'esecuzione, mediante il contenuto dei campi stabiliti dal Formato dell'istruzione.

Il **formato** è la specificazione del contenuto in campi della istruzione (lunghezza e posizione)
La **modalità** di indirizzamento è usata per calcolare gli indirizzi, a cui accedere per il recupero i dati

8.1.1 Formati e modalità di indirizzamento nelle istruzioni del MIPS

Tutte le istruzioni nel MIPS sono lunghe 32 bit, le varie istruzioni appartengono ad uno dei tre formati.

Formato Tipo R (Register)	Gli operandi e la destinazione dell'operazione sono in registri interni al processore
Formato Tipo I (Immediate)	Un campo è specificato da un numero usato come operando incluso direttamente nella istruzione. Usati per calcolare indirizzi di memoria, come salti o come operandi
Formato tipo J (Jump)	Un campo dell'istruzione contiene un numero intero con segno per calcolare l'indirizzo di memoria a cui saltare.

I tipi sono specifici del MIPS ma molte regole sono comuni a tutte le piattaforme.

I tre tipi di formato usano uno dei seguenti tipi di indirizzamento (varia in base alla operazione):

- Indirizzamento tramite registro (Tipo R).
- Indirizzamento immediato (Tipo I).
- Indirizzamento tramite Base e Offset (Tipo I).
- Indirizzamento relativo al Program Counter (Tipo I).
- Indirizzamento pseudodiretto (Tipo J).

Per ragioni di sicurezza non è possibile accedere in **modifica** alle locazioni di memoria che conservano la lista delle istruzioni dei programmi da eseguire.

8.2 Il formato di Tipo R

Nelle istruzioni di tipo R tutti gli operandi sono contenuti nei registri del processore. Usate da tutte le istruzioni aritmetico logiche.
Per identificare uno dei 32 registri è necessario utilizzare 5 bit (2^5=32 combinazioni).
Il formato R divide i 32 bit dell'istruzione in 6 campi:

op (Operation Code) [6 bit]	rs (primo operando) [5bit]	rt (secondo operando) [5bit]	rd (register destination) [5 bit]	shamt (shift) [5bit]	funct (funzione) [6 bit]

- OP: codice operativo, presente in tutte le istruzioni. Indica l'operazione e il formato dell'istruzione. Per tutte le istruzioni R "op" è una sequenza di 6 bit a valore 0 dato che la operazione è specificata nel campo "funct".
- rs: primo operando, è un registro del processore, si indica l'indirizzo mediante 5 bit.
- rt: secondo operando, è un registro del processore, si indica l'indirizzo mediante 5 bit.
- rd: registro di destinazione, il contenuto precedente verrà sovrascritto dal risultato, si indica l'indirizzo mediante 5 bit.
- Shamt: usato solo nelle istruzioni di shift, indica lo scorrimento di bit da applicare al registro. Dato che non è possibile non specificare un parametro è impostato a 0 per tutte le istruzioni che non lo utilizzano attivamente
- funct: operazione da eseguire nella istruzione, campo presente solo nelle istruzioni R.

Il campo "op", nei formati diversi da R, specifica l'azione associata all'istruzione. Presente in tutti i linguaggi macchina. Questo campo viene usato anche per capire oltre al tipo anche il formato della istruzione. I nomi rs e rd ricordano i nomi dei registri utilizzati per elaborare i dati (spiegato dopo).

8.3 L'Indirizzamento tramite registro

Il formato R è associato solo alle istruzioni di tipo R
I 5 bit rs,rt,rd sono gli indirizzi del registro del processore su cui eseguire le operazioni dell'istruzione
Tendo conto delle regole sintattiche è possibile tradurre l'istruzione Assembly in istruzione Macchina, quindi convertendo i vari campi in sequenze binarie.

8.4 Formato di Tipo R per add e sub

Hanno formato di Tipo R, quindi tramite **registro**. Hanno le stesse regole sintattiche, cambia solo il campo “funct” (ricordare che “op code” nelle istruzioni R è composto da 6 bit a 0).

Funzione	Campo funct in decimale	Campo funct in binario
Add	32	100000
Sub	34	100010

I registri del processore essendo 32 possono essere identificati con indirizzi a 5 bit (2^5=32)

A differenza della istruzione assembly, nella sequenza binaria l’indirizzo di destinazione è l’ultimo campo dei registri, “rd”. Gli operandi precedono l’indirizzo di destinazione.
Il campo “shamt“ anche se non è propriamente usato deve essere impostato come una sequenza di 5 bit a 0.

Esempio:

```
add $t0, $s1, $s2
```

Diventa in decimale/binario

0 / 00000 (op code)	17 / 10001 ($s1)	18 / 10010 ($s2)	8 / 01000 ($0)	0 / 00000 (shamt)	32/ 100000 (funct)

Unendo il tutto si ottiene il codice macchina, nel caso si volesse fare la sottrazione basta cambiare il campo “funct”, gli ultimi 6 bit della sequenza.

```
00000010001100100100000000100000
```

Ci siamo comportati inversamente all’assemler. Noi sappiamo dare un significato alla operazione, la macchina esegue i comandi senza “ciecamente”, sa solo dove prendere i dati e come elaborarli. Vedere esercizi.

9 Le istruzioni Logiche e di Shift

9.1 Operatori logici AND, OR e di shift

Gli operatori logici sono funzioni booleani che accettano operandi booleani e restituiscono un valore booleano. Viene dato al valore 0 “falso” e al valore 1 “vero”. Ogni operatore logico è definito dalla relativa tabella di verità.

X	Y	AND		X	Y	OR
0	0	0		0	0	0
0	1	0		0	1	1
1	0	0		1	0	1
1	1	1		1	1	1

AND ha valore “vero” quando **entrambi** gli operatori hanno valore vero
OR ha valore “falso” quando **entrambi** gli operatori hanno valore falso
AND e OR sono operatori duali.

Tali operatori sono usati per costruire espressioni per rappresentare condizioni che modificano il flusso del programma, oppure possono essere applicati su una sequenza di bit, “operatori Logici bit a bit”.
L’AND è chiamato maschera perché nasconde il valore originale di una sequenza, lo 0 nasconde il bit originale della sequenza forzando il valore finale a 0, il valore 1 lascia immutato il valore originale.
L’OR (duale) serve a forzare dei valori di una sequenza ad 1, con il valore “vero” si forzano i valori originali della sequenza ad 1, con lo 0 si lasciano invariati

Operatore	AND		OR
Operando 1	10110110		10100110
Operando 2	**0000**1111		**1111**0000
Risultato	**0000**0110		**1111**0110

Lo shift (scorrimento) sposta i bit di una sequenza a destra o a sinistra lasciando invariato l’ordine immettendo bit a valore 0 nelle posizioni lasciate vuote e perdendo i bit che escono dalla sequenza.

Lo shift a sinistra viene usato per moltiplicare per potenze di 2 il valore (se in notazione pesata), lo shift a destra divide invece il valore rappresentato, perdendo la parte fratta (i bit persi devono essere 0, altrimenti cambia il valore del numero rappresentato). E’ analogo in decimale moltiplicando o dividendo per 10, in questo caso essendo la base 2 si divide o moltiplica per 2.
Tali operazioni di shift sono molto più rapide di operazioni aritmetiche effettuate dall’ALU.

9.2 Assembly MIPS: le istruzioni AND, OR, SLL, SRL

In tutti i linguaggi di programmazione le regole formali definiscono in maniera univoca le azioni che l’hardware deve svolgere. Nell’Assembly MIPS si usano le seguenti strutture base:

- Simboli inglesi e numeri in notazione decimale.
- Una singola istruzione a riga fatta da codice operativo (tipo istruzione e indirizzamento dei dati)
- Commenti, partono appena l’assembler trova il simbolo “#”, se si vogliono commentare più righe è necessario fare iniziare ogni riga con “#”.

Le istruzioni assembly AND/OR effettuano gli operatori logici bit a bit su due registri del processore scrivendo il risultato dell’operazione su un terzo registro indicato a livello di istruzione (usando gli indirizzi). AND (“&”) e OR (“|”) sono analoghi, cambia solo il codice di “funct”.
Sono istruzioni di tipo R con indirizzamento tramite registro, identico ad add/sub.

```
Assembly:
and $t0, $s1, $s2          # And bit a bit effettuato su $s1 e $s2, il risultato va in $t0, $s1 e $s2 sono immutati
C/C++:
a = b & c;                 //a=$t0 , b=$s1 , c=$s2
```

Le istruzioni di shift hanno codice operativo “sll” (shift left logical) o a destra “srl” (shift right logical), usano un solo operando (in un registro). I bit che escono dalla sequenza sono persi e le posizioni lasciate vuote sono valorizzate a 0.

```
Assembly:
sll $t1, $s0, 3          # shift a sinistra di tre posizioni di $s0, risultato in $t1. $s0 è immutato. 3 spostamenti.
C/C++:
a = b << 3               # a=$t1, b=$s0
```

Effettuando 3 spostamenti si moltiplica per 8 = 2^3. Lo shift a destra segue le medesime regole, dividendo. Molto più rapido lo shift rispetto a moltiplicare/dividere per 2 tramite ALU.

9.3 Assembly MIPS: le istruzioni AND, OR, SLL, SRL

Anche “ssl” e “srl” sono istruzioni di **formato Tipo R** (come “AND” e “OR”) ed usano l'indirizzamento tramite registro, come tutte le istruzioni aritmetico/logiche (il campo “op” è tutto a 0)

Funzione	Campo funct in decimale	Campo funct in binario
sll	0	000000
srl	2	000001

Il campo “rs” (il primo registro da indicare, subito dopo “op”, bit 6-10) non viene utilizzato, per questo motivo è riempito con una serie di 5 bit a 0 (cioè il registro $zero che è in lettura). In questo caso il campo “shamt” deve contenere il numero di bit da “shiftare”, è un numero positivo, e essendo composto da 5 bit può spostare praticamente tutti i bit, in quanto effettuando uno spostamento di 32 posizioni imposterà il registro con un numero contenente tutti i bit a 0 (sia per “sll” che per “srl”).
La traduzione in linguaggio macchina contiene lo spostamento del registro destinazione come ultimo anziché come primo registro della istruzione assembly:

```
ssl $t0, $s1, 7          # $t0 destinazione, $s1 registro su cui spostare I bit, 7 I bit da spostare. $s1 resta invariato
```

Op 0 / 000000	Rs 0 / 00000	Rt 17 / 10001 (s1)	Rd 8 / 01000 ($t0)	Shamt 7 / 00111 (sll)	000000

Formato Tipo R usano **solo** il tipo di indirizzamento tramite registro, tutti i campi sono popolati dall’indirizzo dei registri dei processori che hanno l’informazione.

Ricordare che Formato tipo R e indirizzamentro tramite registro sono cose diverse

10 Formato di Tipo I per addi e Indirizzamento immediato

10.1 Il Formato di Tipo I e le relative Modalità di Indirizzamento

Formato I (Immediate), quindi immediato. Un operando è un numero intero in notazione a 2

direttamente nella istruzione macchina, esplicitato nella istruzione stessa.
Il campo immediato viene utilizzato in vari tipi di indirizzamento nel formato I (rivedere).
Come in tutte le istruzioni Assembly MIPS i primi sei bit sono relativi al *Codice Operativo*, la macchina con questo campo distingue il formato e la modalità di indirizzamento.
Successivamente sono presenti due indirizzi relativi a registri del processore, i restanti **16 bit** (32-6-5-5 = 16) sono disponibili per l'operando immediato (in complemento a 2).
Tutti i calcoli relativi all'operando immediato sono eseguiti dall'ALU, ma essa lavora SOLO con dati a 32 bit (non 16) e di conseguenza prima di essere inviati a tale unità i dati sono estesi di segno per aumentarne la lunghezza a 32 bit. Il campo immediato contiene valori che vanno da un minimo di -2^{16-1} a un massimo di $2^{16-1}-1$ $[-2^{15}; 2^{15}-1]$.

Op - 6 bit	Rs – registro CPU 5 bit	Rt – registro CPU 5 bit	Intero immediato – 16 bit

La modalità di indirizzamento stabilisce regole per determinare gli indirizzi dei registri da utilizzare per eseguire l'operazione mediante il contenuto dei campi della istruzione

Di seguito le varie modalità di indirizzamento (stesso formato quindi ordine e lunghezza dei campi) delle istruzioni I:

- **Indirizzamento immediato**: numero immediato, fornisce un operando tramite registro CPU e il secondo è il numero in complemento a 2 presente nella istruzione (istr. aritmetico/logiche).
- **Indirizzamento tramite base e offset (spiazzamento)**: il numero immediato è utilizzato per leggere o scrivere un dato dalla memoria centrale (trasferimento dati)
- **Indirizzamento relativo al Contatore di Programma (Program Counter)**: numero immediato utilizzato per calcolare l'istruzione a cui saltare per proseguire con l'esecuzione, salto condizionato.

Tutte le istruzioni hanno il medesimo formato ma vengono utilizzate da funzioni molto diverse. Stesso formato, diversa istruzione, medesima codifica immediata.

10.2 L'Indirizzamento immediato

Nelle istruzioni Formato I un operando è il campo a 16 bit dell'istruzione, i due registri forniscono l'altro operando e il registro di destinazione.

10.3 Formato di Tipo I e Indirizzamento Immediato per istruzione addi

L'istruzione MIPS "addi" (add immediate) è realizzata con istruzioni di Tipo I e indirizzamento immediato.
Il codice operativo ("op") è il valore decimale 8 e fornisce alla CPU il formato e le informazioni di indirizzamento.

Assembly

```
addi $t1, $s7, -13          # $t1 è la destinazione, $s7 è il primo operando, -13 è il secondo operando. $s7 resta immutato
```

C/C++

```
a = b + (-13)          // a = $t1, b = $s7
```

Come nelle istruzioni in formato R destinazione e operando a livello macchina sono invertiti, l'istruzione sopra diventa

Op - 8 / 001000	$s7 - 23 / 10111	$t1 - 9 / 01001	Immediato -13 / 1111111111110011

In linguaggio macchina è

```
00100010111010011111111111110011
```

11 Istruzioni lw e sw Assembly MIPS e Indirizzamento tramite Base e Offset

11.1 La memoria principale della architettura MIPS

Sia i dati che le istruzioni sono salvate in sequenze binarie, salvate in un dispositivo di memorizzazione (memoria centrale tipicamente).
Ogni sequenza binaria salvabile è costituita da una **locazione** detta **registro**. I registri hanno due modalità di accesso: in lettura dove la sequenza è prelevata senza modifiche al registro, in scrittura dove la sequenza viene sostituita perdendo la vecchia sequenza.
I blocchi di registri sono più registri posti sequenzialmente. La memoria principale è un blocco di registri diverso dai registri in CPU collegata tramite BUS, viene usato per salvare dati e programmi.
La velocità di accesso ai registri è molto diversa rispetto ai registri interni alla CPU.
Ogni locazione ha una lunghezza definita ed è associata ad un un indirizzo, se non abbiamo l'indirizzo non è possibile accedere al dato, è come non averlo. Gli indirizzi sono tanti quante le locazioni.
Gli indirizzi partono da 0 fino ad arrivare ad N-1, ogni cella ha un suo indirizzo numerico.
Ricordare:

- 1 cifra è detta bit, binary digit.
- 8 cifre binarie sono dette Byte, sinteticamente B
- 32 cifre binarie sono dette parola, word. 4 Byte. 32 bit.

> I processori MIPS hanno registri nella memoria principale di lunghezza fissata di 1 byte, quindi ogni registro memorizza 8bit. Di conseguenza 8 bit è la quantità minima di informazione che può essere letta e scritta.
> Essendo i registri di 32 bit (word) per rappresentare un registro/istruzione/dato sono utilizzati 4 locazioni di memoria consecutive.

Dato che la lunghezza massima di un numero dipende dalla lunghezza dei bit della sequenza e i registri CPU sono limitati a 32 bit si ha che gli indirizzi massimi indicizzabili sul MIPS sono 2^{32} (scelta progettuale della architettura).
Le misure di grandezza della memoria, essendo indirizzate con indirizzi binari, devono essere potenze del 2.

> Essendo ogni istruzione/dato lungo 32 bit, nel processore MIPS, si ha a disposizione circa 1 Giga di parole (1 miliardo).

11.2 Istruzione Assembly MIPS Load Word

La memoria principale è un blocco di locazioni simile ad i registri in CPU, ma di dimensioni maggiori e molto più lenta. I registri in CPU sono limitati, ma necessari ad alcuni tipi di operazioni. Le istruzioni di trasferimento leggono o scrivono su/dalla memoria.
Negli applicativi di grosse dimensioni le variabili sono portati in memoria principale per liberare i registri in CPU e lette al momento dell'utilizzo. I compilatori cercano di usare al massimo i registri disponibili.
Le istruzioni di lettura/scrittura dalla memoria richiedono a minimo l'utilizzo di una singola locazione di memoria, quindi un Byte (8bit). Il numero di locazioni aumenta in base alla lunghezza delle strutture, come le 4 necessarie per la word (uso più comune), utilizzata per i numeri.
Solitamente le celle di memoria utilizzate sono consecutive, aumentando la velocità di accesso.

Istruzione load word (in assembly "lw")

Indirizzamentro tramite Base ed Offset

Legge la memoria e scrive in un registro della CPU, la parte di memoria letta resta immutata. In questo caso si accede a 4 locazioni consecutive

```
lw $t1, 80($s0)          #$t1 è il registro CPU di destinazione, 80 è l'offset, $s0 è il registro CPU con indirizzo Base
```

Sintattica:

- "lw" - è il codice operativo ("op" o "opcode")
- "$t1 " - registro di destinazione, verrà scritto il dato letto dalla memoria centrale
- "80" - **numero intero di offset (spiazzamento)**. Indica di quante posizioni è necessario spostarsi dal registro di base
- "$s0" - **Registro Base** in cui è salvato come l'indirizzo in memoria centrale.

L'indirizzo da leggere in memoria centrale è ricavato dalla **somma tra indirizzo di base e l'offset**.

11.3 Indirizzamento tramite Base e Offset per lw

L'indirizzamento tramite Base e Offset usato da load word e store word calcola l'indirizzo a 32 bit della memoria centrale tramite:

Indirizzo di Memoria = contenuto del Registro Base + Offset

Nelle istruzioni "lw" e "sw" l'offset è una costante a 16 bit inserita dentro l'istruzione. La base è un indirizzo specifico dato che lungo 32 bit e può puntare a qualsiasi locazione di memoria.
Anche se si devono leggere 4 locazioni basta fornire l'indirizzo al primo byte della parola, gli altri sono letti automaticamente concatenando le sequenze dei vari Byte.
L'indirizzo di base non viene modificato da "lw" e "sw".
Ci sono due possibili modi per creare il dato concatenando le sequenze delle varie locazioni, dal byte a sinistra riempiendo verso destra la sequenza con le locazioni successive o viceversa:

- Big-endian (big end) usano il byte indicato in memoria come la parte più significativa (a destra) procedendo al caricamento dei byte meno significativi leggendo le celle successive in memoria
- Little-endian (little end) usano i byte indicato in memoria come la parte meno significativa procedendo a caricare le parti più significative avanzando le celle in memoria.

Il MIPS è Big-endian, parte dalla cifra più significativa (vedere pagina 171)

11.4 Istruzione Assembly MIPS Store Word

Effettua l'inverso di load word, legge da un registro in CPU e lo porta in memoria. Il registro è immutato dopo l'operazione,i dati in memoria principale vengono sovrascritti.

```
sw $t1, 60($s0)          #$t1 è la sorgente dei dati, 60 è l'offset, $s0 è il registro base che punta alla locazione in RAM
```

Utilizza indirizzamento tramite base e offset anche store word "sw".

L'operazione di spostamento di una variabile da un registro alla Memoria è denominata register spilling (versamento dei registri), mentre l'operazione inversa di spostamento di una variabile dalla Memoria a un registro è detta register filling (riempimento dei registri)

11.5 Indirizzamento tramite Base e Offset per sw

Analogo alla store word, la memoria è scritta anziché letta. I registri restano immutati, cambia la RAM.

In linguaggi ad alto livello si usano i puntatori per accedere alla memoria centrale. L'assembly è molto complesso dato che permette l'accesso diretto al programmatore. Il Java nasconde totalmente l'accesso in memoria diretto al programmatore astraendo il concetto di puntatore (impliciti).

12 Formato di Tipo I per lw e sw e gestione del Tipo di dato Array

12.1 Formato di tipo I e indirizzamento tramite base e offset per lw e sw

Le istruzioni "lw" e "sw" sono istruzioni i Tipo I con **indirizzamento tramite base offset**.
Il codice operativo per lw è il decimale 35 (100011) ed indica alla macchina sia il tipo che l'indirizzamento. Nella traduzione il campo destinazione è l'ultimo indirizzo al contrario della istruzione Assembly (come nelle istruzioni R).

```
lw $t1, 7 ($s0)          #destinazione $t1, $s0 base, 7 offset
```

Op - 35 / 100011	Rs $s0 - 16 / 10000	Rt $t1 - 9 / 01001	Immediato 7 / 0000000000000111

Istruzione Store Word "sw"

Anche la store è di tipo I con indirizzamento tramite base e offset. Il codice operativo è il decimale 43 (101011).

Durante la "lw" e la "sw" il campo immediato viene esteso di segno, portandolo da 16 a 32 bit e inviato all'ALU per ottenere l'indirizzo di accesso completo della memoria principale. Basta indicare la prima locazione di memoria dove è salvato il dato, le altre saranno lette automaticamente (big-endian).

12.2 Gestione del Tipo di dato Array

I linguaggi ad alto livello presentano il tipo di dato Array, strutturato e complesso rispetto ad un dato primitivo. E' una lista di elementi in spazi contigui di memoria dove ogni elemento è accessibile tramite un indice (parte dall'elemento 0 fino a N-1 dove N è la lunghezza dell'array).
Un array è identificato da un nome e contiene elementi dello stesso dipo, la dimensione è il numero

massimo di elementi che può contenere.

```
C++ / Array
int A[10];                    // si crea un array chiamato “A” di elementi interi
A[0] - A[1] - A[2] - A[3] - A[4] -  A[5] -  A[6] -  A[7] -  A[8] -  A[9]                    NON esiste l’elemento A[10]
```

Se si prova ad accedere ad A[10] si otterrà un “Array out of bound” e si sta accedendo a locazioni non destinate agli elementi dell’array. Alcuni linguaggi (Java) controllano gli indici evitando tale errore.
Inizializzando un array si occupa uno spazio di memoria che dipende dal tipo degli elementi e dalla lunghezza dell’array (“dimensioneSingoloElemento” x “lunghezzaArray”).

In assembly la gestione di un array implica l’uso di “lw” e “sw”, si assume che il primo Byte del primo elemento è l’indirizzo base da dove parte l’Array e l’Offset è utilizzato per gestire l’indice.

Prendendo come riferimento un Array di interi di 10 elementi, dove un intero è composto da 32 bit, quindi 4 locazioni di memoria consecutive possiamo calcolare l’indice A[k] riferendoci ad A[0] come **indirizzo base dell’Array** e aggiungendo un offset pari a “4 x K” per accedere alla prima locazione di memoria dell’elemento K. L’indirizzo base è **fondamentale** per la gestione dell’Array.
In linguaggi come il C sono presenti i puntatori, se si perde il puntatore viene perso anche l’array.
In Java gli array si gestiscono tramite riferimenti, le parentesi hanno una priorità maggiore rispetto agli altri operatori del linguaggio. Sono implementati anche controlli automatici per gli “out of bound”

13 Istruzioni di salto condizionato su uguaglianza e disuguaglianza

13.1 Istruzioni Assembly MIPS beq e bne

Utilizzate per cambiare il flusso di esecuzione di un programma, cambia la sequenza delle operazioni, può essere condizionato (se è verificata una condizione) o incondizionato (sempre eseguito).
Tutti i linguaggi prevedono una forma di istruzioni di salto tralasciando quindi parti del codice o ripetendone altre. I salti hanno la necessità di sapere a quale istruzione saltare, cioè l’istruzione che dovrà essere eseguita. Tramite le **etichette** è possibile identificare univocamente l’istruzione e riferirci all’indirizzo di memoria dove è salvata.
Il Program Counter (“PC”) vine automaticamente modificato impostando l’istruzione successiva nel caso l’istruzione non sia di salto, altrimenti viene impostato l’indirizzo della parola di memoria che ha l’istruzione desiderata (essendo delle word sono 4 locazioni di memoria non una, puntare alla istruzione successiva significa che il PC diventa PC+4).
Nel MIPS sono presenti due istruzioni per il salto condizionato che operano confrontando due registri del processore. “beq” (branch equal) salta se il contenuto è il medesimo, “bne” (branch not equal) salta se il contenuto è diverso.
L’istruzione di salto NON cambia la memoria o i registri, cambia solo il PC, se la condizione non è verificata si esegue l’istruzione successiva.

```
beq $t0, $t5, nome_etichetta          # beq codice operativo, $t0-$t5 registri da confrontare, istruzione a cui saltare
addi $s3, $s3,1
```

```
nome_etichetta: sw $s3, 0($s1)
```

L'operazione confronta i 32 bit dei due registri, se il valore è il medesimo si esegue un salto alla istruzione identificata da "nome_etichetta". L'istruzione "bne segue le medesime regole".
Le etichette vanno inserite solo per le istruzioni a cui si vuole saltare, nel caso sopra se la condizione è verificata si salta alla istruzione "sw" senza eseguire l'istruzione di somma.

13.2 Formato di Tipo I per beq e bne

Non esiste una trasposizione in linguaggio macchina della "Etichetta" usata in Assembly per sapere a quale operazione saltare, bisogna fornire l'indirizzo in memoria. L'indirizzo viene costruito utilizzando le informazioni disponibili nella istruzione di branch, tali istruzioni sono di **formato tipo I** e utilizzano l'**indirizzamento relativo al Program Counter**.
Il codice operativo per "beq" è 4 (000100), i due registri da confrontare sono in "rs" e "rt", le istruzioni da saltare sono scritti in notazione a 2 nel campo immediato. È possibile fornire un numero positivo o negativo di istruzioni da saltare rispetto alla istruzione "beq".
Nella traduzione in linguaggio macchina non è necessario invertire i due operandi.

```
Beq $t0, $t5, num_incremento
```

op - 4 / 00100	rs $t0 - 8 / 01000	rt $t5 – 13 / 01101	Immediato – num istr. da saltare

Il numero delle istruzioni da saltare è compilato dall'Assembler durante la traduzione, ricerca la etichetta e ne calcola il numero di istruzioni di distanza.

RICORDARE! Quando si esegue una istruzione il PC punta alla successiva, nell'esempio di codice sopra durante la valutazione di "beq" il registro PC punta alla istruzione "addi". L'assembler tiene conto di ciò sapendo che il PC punta già all'istruzione successiva.

L'istruzione "bne" ha codice operativo 5 (000101), con regole analoghe.

13.3 Indirizzamento relativo al Program Counter per beq e bne

Le istruzioni di salto condizionato calcolano l'indirizzo base al valore contenuto nel PC addizionando il valore contenuto nella parte immediata della istruzione. L'addizione è fatta dall'ALU dopo che il campo immediato è stato esteso a 32 bit. L'ampiezza del salto va da $-2^{16-}1$ fino a $2^{16-1}-1$ [-2^{15}; $2^{15}-1$].

Dato che ogni istruzione è lunga 4 Byte il valore contenuto nel campo immediato viene moltiplicato per 4 dopo essere stato esteso, viene fatto in automatico dall'hardware (shift a sinistra di 2 posizioni).

L'indirizzamento è simile a quello di base e offset, nel caso di salto l'offset è quello presente nel PC.
Indirizzo dell'istruzione a cui saltare = Program Counter + 4 x campo immediato esteso a 32 bit.

14 Istruzione di salto incondizionato

14.1 Istruzione Assembly MIPS j

L'istruzione di salto incondizionato è fatto con codice operativo "j" ("jump").
Non ha nessun operando ma solo l'etichetta (posta PRIMA della istruzione desiderata) a cui saltare in maniera incondizionata. Come nel caso precedente l'etichetta è un identificatore univoco di una istruzione, il salto NON modifica alcun valore nei registri in CPU eccetto quello del PC.

```
j nome_etichetta
```

Questo tipo di istruzione necessita istruzioni di controlli effettuate precedente dell'istruzione per non bloccare l'esecuzione del programma.

14.2 Formato di Tipo J per j

Le istruzioni sono lunghe 32 bit e salvate nella memoria principale. Quando si effettua un salto è necessario fornire la locazione della istruzione che vogliamo raggiungere.
L'istruzione "j" utilizza il **formato tipo j**, questo tipo di istruzione dedica il massimo spazio disponibile per identificare l'indirizzo a cui saltare, oltre i 6 bit del codice operativo è presente un altro campo da 26 bit per fornire parte dell'indirizzo.

op - 6 bit	Indirizzo di salto incompleto – 26 bit

Il codice operativo è dato dal valore decimale 2 (000010), l'altro indica 26 dei 32 bit della istruzione. I 6 bit mancanti dell'indirizzo vengono calcolati automaticamente dall'hardware a livello circuitale tramite l'**indirizzamento pseudodiretto**, nella istruzione viene fornita la parte essenziale dell'indirizzo.

14.3 Indirizzamento pseudodiretto per "j"

Indirizzamento pseudodiretto

I 26 bit dell'indirizzo indicano solo la parte principale, i 6 bit restanti non indicati sono ottenuti tramite:

- I quattro bit più significativi del Program Counter
- I 26 bit del campo indirizzo, indicato nella istruzione in formato "J".
- Due bit 00 nelle posizioni meno significative. Si usa la proprietà della notazione posizionale pesata, i multipli di 4 terminano con due 0 finali. La memoria principale ha blocchi di 8 bit, quindi ne servono 4 per ottenere l'istruzione.

I 4 bit più significativi sono quelli che cambiano meno durante l'esecuzione del programma. Fissando i 4 bit si ottiene che i 28 bit indicano 2^{28} = 256 Mega Byte (2^{8} x 2^{20}). In 256 MegaByte è possibile salvare 64 Mega di parole o di istruzioni.
Se il programma può essere memorizzato in tale spazio i primi 4 bit di indirizzamento non cambiano, è quindi possibile leggerli dal PC.
I primi 4 bit formano un segmento di memoria chiamato **pagina di memoria**.

15 Traduzione in Assembly MIPS della istruzione if-else e dei cicli for e while

15.1 Traduzione dell'istruzione if-else

Sono implementazione di controlli per evitare che pezzi di codice non facciano terminare l'esecuzione. La condizione "if-else" permette di eseguire due insiemi di istruzioni alternative in base ad una espressione logica. In assembly tale condizione deve essere ricreata tramite controlli e salti condizionati e non.

```
int test, a, b, f;
if ( test == 0 ) {          //salta a ELSE se test == 0 è falsa
        f = a ;
}
else {
        f = b ;
}
test = a□b ;
```

In Assembly:

```
test = $t5      a = $t1         b = $t2         f = $s0
bne $t5, $zero, ELSE   # salta a ELSE, se test == 0 è falsa (se $t5 è 0 si continua dalla ists. successiva), se test != 0 è vera
addi $s0, $t1, 0                # f = a
j END_IF                # esce da if-else
ELSE: addi $s0, $t2, 0  # f = b
END_IF: sub $t5, $t1, $t2 # test = a□b
```

Viene utilizzato l'operatore non uguale a zero per mantenere la stessa struttura e ordine delle istruzioni in C. Logicamente si salta se è diverso da zero, come nel nostro caso, altrimenti si prosegue con la esecuzione

> Importante l'istruzione "j END_IF" altrimenti dopo l'if sarebbe stato eseguito anche l'else, in questo modo dopo avere eseguito il blocco dell'if si riprende dopo il blocco istruzioni dell'else.

15.2 Traduzione del ciclo for

Si ripete una esecuzione fino a che la condizione risulta vera. Il for ripete un blocco di istruzioni per un numero fissato di volte. Solitamente una variabile di controllo conta il numero di ripetizioni, ogni esecuzione si incrementa la variabile. Ogni ciclo si verifica la condizione fino a che non risulta falsa. Se la condizione è falsa il ciclo termina senza eseguire nemmeno una volta le istruzioni, andando alla istruzione successiva. Il ciclo può portare ad una esecuzione infinita se la condizione è sempre vera.

```
int somma, k, Max ;
Max = 5 ;
for ( k = 0 ; k != Max ; k = k+1 ) {          //uscita dal ciclo for se k != Max è falsa
        somma = somma + k ;
}
Max = 0
```

```
- - In Assembly - -
somma = $s1    k = $t0          Max = $t5

addi $t5, $zero, 5              # Max = 5
add $t0, $zero, $zero           # k = 0 inizializzazione
FOR: beq $t0, $t5, END_FOR      # uscita dal ciclo for se k != Max è falsa, se k == Max è vera
add $s1, $s1, $t0                       # somma = somma + k
addi $t0, $t0, 1                        # k = k+1 incremento
j FOR                           # ripete il ciclo for
END_FOR: add $t5, $zero, $zero          # Max = 0
```

Come nella condizione “if-else” la condizione del salto è invertita per mantenere lo stesso ordine.
Da notare che la inizializzazione di “k” a 0 è fatta fuori dal ciclo.

15.3 Traduzione del ciclo while

Il while cicla fino a che una condizione risulta vera, se tale condizione è falsa si salta il blocco di codice senza eseguirlo. A differenza del for non è indicato il numero di volte per cui ciclare, come nel caso precedente è possibile incorrere in un ciclo infinito se la condizione risulta sempre vera.

```
conta = 0 ;
while ( val != 0 ) {            //uscita dal ciclo while se val != 0 è falsa
        val = val >>1 ;         // la condizione divide per due, al massimo dopo 32 iterazioni si avranno
tutti bit a 0
        conta = conta + 1 ;
}
val = 7 ;

- - Assembly - -
val = $s1       conta = $t0
add $t0, $zero, $zero                   # conta = 0
WHILE: beq $s1, $zero, END_WHILE #uscita dal ciclo while, se val != 0 è falsa e si cicla, se val == 0 è vera
si esce
srl $s1, $s1, 1                         # val = val >>1
addi $t0, $t0, 1                                # conta = conta + 1
j WHILE                                 # ripete il ciclo while
END_WHILE : addi $s1, $zero, 7                  # val = 7
```

Equivalente al while, senza inizializzazione. La condizione di termine è invertita come nel “for”.

16 Gestione della chiamata di procedura con le istruzioni jal e jr

16.1 La chiamata di procedura

Le procedure sono programmi con compiti specifici, compilati separatamente e disponibili per essere chiamate dal programma principale. Riutilizzo del codice, paradigma procedurale+.
Le procedure sono funzioni se restituiscono valori, sono identificate da un nome e la lista di parametri. Il programma principale e la procedura comunicano attraverso la lista dei parametri, al termine la procedura può restituire valori e riprende l'esecuzione dal punto in cui è avvenuta la chiamata.
Essendo salvati in parti dell'eseguibile diverse il passaggio dal programma alla procedura avviene tramite un salto, quindi modificando il registro PC. Al termine si torna all'indirizzo successivo della chiamata nel programma principale (indirizzo di ritorno), è necessario anche considerare un sistema per il recupero degli eventuali risultati. Per fare tutto ciò esistono opportune istruzioni.

16.2 Le istruzioni Assembly MIPS jal e jr

Per la procedura sono necessari: parametri, valori di ritorno, registro di ritorno. Per fare tutto ciò sono riservati opportuni registri del processore:

- da $a0 a $a3 sono 4 registri predisposti per i parametri della procedura. In cui impostare i valori prima della chiamata a procedura.
- $v0 e $v1 sono registri per i due valori di ritorno, da impostare al termine della procedura.
- $ra indirizzo di ritorno a cui andare al termine della chiamata a procedurale

Per le chiamate a procedura in assmebly è disponibile l'istruzione di salto "jal" (Jump And Link). Tale istruzione imposta il program counter (Jump) e salva in $ra il registro di riorno (Link), cioè l'istruzione successiva alla chiamata a procedura. Il registro $ra consente di continuare l'esecuzione da dove era interrotta.
La procedura per terminare l'esecuzione e restituire il controllo alla procedura utilizza l'istruzione "jr" (Jump Register) che effettua il ritorno al programma chiamante. Il registro contiene già i 32 bit dell'indirizzo di memoria, non è necessario l'indirizzamento pseudodiretto. Vedere pag. 141.

```
-- Programma principale --
add $a0, $t0, $zero          #si imposta un parametro per la procedura
jal nome_procedura           #si invoca la procedura
-- Procedura --
add $v0, $s1, $zero          #valore di ritorno della procedura
jr $ra                       #si torna alla esecuzione del programma principale
```

16.3 Annidamento di chiamate di procedura e call stack

Le procedure possono chiamare altre procedure oppure se stesse (ricorsiva) durante l'esecuzione. Tali procedure sono quindi annidate fino ad arrivare alla procedura foglia che non chiama altre procedure. Quando si effettua una procedura la chiamante resta in attesa del termine della figlia, ovviamente non è possibile utilizzare i registri $a.

è necessario gestire uno stack (pila) contenente la lista dei vari parametri passati alle funzioni rispettando l'ordine di chiamata (LIFO). L'ultimo elemento inserito sarà il primo la cui procedra terminerà l'esecuzione e quindi anche il primo ad essere estratto.
Lo stack sono gestiti tramite "push" (inserito un nuovo elemento), e "pop" (si preleva l'ultimo elemento).
La **call stack** è una struttura che salva le informazioni delle chiamate di procedura attive (indirizzo di ritorno, parametri, etc..), cioè quelle non terminate. Le informazioni della singola esecuzione è chiamato "**frame**" della procedura. Ad ogni chiamata si genera un nuovo frame inserito in testa allo stack, al termine della procedura foglia la procedura cancellerà il relativo frame.
La call-stack è associato ad ogni programma ed occupa uno spazio limitato di memoria riservato all'avvio, se il programma effettua chiamate infinite si esaurirà lo spazio in stack generando il "call stack overflow". I linguaggi ad alto livello gestiscono la stack-call automaticamente, in assembly deve essere fatto dal programmatore in base anche dal sistema operativo, oppure effettuato dal traduttore.

16.4 Formato e Modalità di Indirizzamento di jal

Indirizzamento pseudodiretto.

16.5 Formato e Modalità di Indirizzamento di jr

Utilizzata per leggere il valore del registro e scriverlo nel PC, per riprendere l'esecuzione dalla istruzione successiva alla chiamata a procedura.
Il "Jump Register" utilizza il **Formato di tipo R** e **indirizzamento tramite registro**.
Il campo "op" è formato da 6 bit a 0, l'operazione in "funct" è data dal valore decimale 8 (001000) ed è necessario impostare valorizzato solo il primo campo registri "rs" con l'indirizzo del registro da cui leggere il valore dell'indirizzo, normalmente è "$ra" cioè l'indirizzo 31. Tutti gli altri campi sono valorizzati a 0.

17 Istruzioni di confronto e interi unsigned

17.1 Istruzione Assembly MIPS slt

Nei linguaggi ad alto livello esistono molti operandi per il controllo del flusso, la istruzione "slt" ("Set on Less Then") si va ad aggiungere alle "beq" e "bne". Permette di confrontare in base alla relazione d'ordine del minore due registri (imposta ad 1 se minore), utilizzando un registro aggiuntivo per memorizzarne il risultato. Non effettua automaticamente nessun salto, tale confronto è fatto dall'ALU.

```
slt $s0, $t1, $t2                # pone in $s0 vero (1) se $t1<$t2, altrimenti false (0) in $s0
```

I due registri confrontati restano immutati, viene modificato solo il registro destinazione.
SI EFFETTUA IL CONFRONTO SU VALORI IN COMPLEMENTO A 2, CIOÈ CON SEGNO.
È una istruzione formato tipo R con indirizzamento tramite registro, il campo "op operation code" è formato da 6 bit a 0, il campo "funct" è impostato dal valore decimale 42 (101010).
L'ordine dei registri degli operandi è il medesimo tra Assembly e codice macchina, quello di destinazione invece è invertito nell'ultimo indirizzo nel linguaggio macchina. Esempi pag. 263.

17.2 Tipo di dato unsigned

Per la gestione di alcuni dati (soprattutto puntatori) si ricorre a numeri senza segno (“unsigned”), sono rappresentati in notazione posizionale pesata invece che complemento a 2.

- Notazione in complemento a 2:[-2^{n-1} ; $2^{n-1}-1$]
- Notazione posizionale pesata:[0, 2^n-1]

Per N=8 si ha: in complemento a 2 [-128, 127]; in notazione pesata [0, 255].
In Assembly è possibile gestire i dati unsigned aggiungendo il suffisso “u” alla istruzione. Le istruzioni conservano il formato e l’indirizzamento delle controparti con segno.
In MIPS si utilizzano tutti e 32 i bit del registro durante l’esecuzione delle operazioni.

17.3 Istruzioni aritmetiche con operandi unsigned addu e subu

```
addu $s0, $t1, $t2
subu $s0, $t1, $t2
```

Il controllo dell’overflow è diverso in quanto NON vale il check sul valore degli ulltimi due riporti calcolati usata in notazione a complemento a 2

AND, OR, e tutte le operazioni logiche sono fatte senza segno, ma direttamente sui singoli bit.

17.4 Istruzione di confronto tra operandi unsigned sltu

```
sltu $s0, $t1, $t2
```

Cambia solo il campo “funct” della istruzione passando al valore 43 (101011). La notazione in complemento a 2 utilizza l’ultimo bit come valore negativo, di conseguenza saranno minori di tutti i valori che hanno il valore dell’ultimo bit a 0. Utilizzando “sltu” la cosa si inverte in quanto si passa alla notazione pesata (“unsigned”).

18 Operandi immediati e costanti a 32 bit

18.1 Istruzione di confronto con operando immediato slti

Controllo di esecuzione, controllo di un numero **costante** espresso direttamente in espressione.
Oltre al confronto tra registri esiste la versione di confronto immediato. Stesse regole di “addi”, l’operazione è “**slti**”, confronta un registro e un valore costante. **Formato I**, indirizzamento **immediato**.

```
slti $t1, $s7, -13          #$t1 è il risultato, true se $s7 < -13, false altrimenti
```

Il codice operativo è rappresentato dal valore 10 (001010). In linguaggio macchina il registro di destinazione è il secondo (inversione). Il campo immediato è di 16 bit in complemento a 2.
L’ALU effettua il controllo, di conseguenza il campo a 16 bit è esteso a 32.

18.2 Istruzioni logiche con operando immediato andi e ori

Sono operatori che lavorano bit a bit, NON su numeri, non esiste la controparte “unsigned”. In entrambi i casi i 16bit non specificati sono automaticamente completati con 0, non viene esteso il

segno. Sono Istruzioni con formato tipo I e indirizzamento immediato. “andi” ha codice operativo 12 e “ori” ha codice “13”.

18.3 Costanti a 32 bit

Dato che i registri e le istruzioni sono entrambi di lunghezza prefissata a 32 bit si è limitati a costanti a 16 bit (lunghezza del campo immediato). Per gestire costanti maggiori di 16 bit si ricorre a tecniche di combinazione delle istruzioni disponibili, suddividendo il numero in due sotto sequenze da 16 bit ed utilizzando la funzione di shift:

```
Confronto con costante 65539 = 35536 + 3
ori $s0, $zero, 1          # pone il valore 1 nel registro $s0 (parte alta)
sll $s0, $s0, 16           # shift di $s0 di 16 posizioni a sinistra (shift della parte alta)
ori $s0, $s0, 3            # OR bit a bit tra il contenuto di $s0 e il valore 3 (parte bassa)
```

Siamo riusciti a scrivere una costante in $s0 senza passare dalla memoria centrale.

19 Codifica dei caratteri e Tipi di dato carattere e stringa

19.1 Codifica ASCII per i caratteri

I testi alfanumerici sono rappresentati in vari modi all’interno della macchina:
ASCII (American Standard Code for Information Interchange) del 1968 utilizza sequenze di 7bit. Ad ogni simbolo è associato un numero tra 0 e 127, i codici da 23 a 126 sono stampabili (95 codici), gli altri sono usati per formattare o come simboli di controllo. Importante la sequenza di 7 bit a 0 (“Null” oppure “\0”) usata nel C per indicare il carattere di fine stringa. I caratteri non stampabili sono rappresentati con una barra e una lettera/numero associato al carattere.
La codifica ASCII è molto limitata e per questo è stata creato “Extended ASCII” a 8 bit per carattere, include l’asci normale inserendo il bit più significativo con 0. Sono state aggiunti caratteri accentati e altri simboli.
Rimangono ancora fuori altri simboli non basati sull’alfabeto greco/latino come i caratteri cinesi o giapponesi.

19.2 Codifica Unicode per i caratteri

L’Unicode Consortium è il formato internazionale per gestire i testi in lingue diverse. Sono previste varie codifiche a 8-16-32 bit (UTF-8, UTF-16, UTF-32 UTF = Unicode Transformation Format).
La codifica ASCII è un sottoinsieme della codifica UNICODE.
Il più utilizzato è l’UTF-8 che utilizza gruppi di Byte in numero variabile, ciò consente di utilizzare il minor spazio possibile in base alla lingua su cui stiamo scrivendo.

19.3 Gestione dei tipi di dato carattere e stringa in Assembly MIPS

Per la gestione dei caratteri vengono utilizzati prevalentemente i caratteri “unsigned”, in questo modo nei registri NON viene effettuata l’estensione di segno che cambierebbe il carattere codificato.

```
Val = 'A'
ori $t0, $zero, 65          # si imposta in $t0 il valore 65 senza estensione di segno
```

Dato che in ASCII i caratteri crescono seguendo l'ordine di grandezza è possibile creare relazioni per gestire l'ordinamento tra caratteri. In aggiunta i caratteri minuscoli e maiuscoli differiscono di una costante pari a 32.
Le stringhe sono array di caratteri. Esistono istruzioni di lettura dalla memoria in base al tipo di codifica (8 – 16 – 32 bit). Vedere esercizi pag. 299.

20 Istruzioni Load e Store per Byte e half word, e Tipi di dato interi

20.1 ISTRUZIONI ASSEMBLY MISPS PER TRASFERIMENTO DI BYTE E HALF WORD

Sono istruzioni simili a load word e store word, ma anziché leggere 4 indirizzi contigui di memoria accedono a 2 oppure al singolo registro. Utilizzate anche per gestire i caratteri ASCII/UFT-16:

- suffisso "b" per il byte, 8 bit, singolo indirizzo di memoria
- suffisso "h" per la "half word", 16 bit, mezza parola

Le istruzioni disponibili sono le seguenti:

- lb (load Byte) - Esempio: "lb $t0, 50 ($s1)". Scrive negli 8 bit meno significativi del registro, effettua la estensione del segno
- lbu (load Byte unsigned) Esempio: lbu $t0, 50 ($s1). Scrive negli 8 bit meno significativi del registro, non effettua la estensione del segno, i bit rimanenti sono posti a 0
- lh (load half word) Esempio: lh $t0, 50 ($s1). Scrive nei 16 bit meno significativi del registro i dati letti dalla memoria. Effettua la estensione del segno.
- lhu (load half word unsigned) Esempio: lhu $t0, 50 ($s1). Scrive nei 16 bit meno significativi del registro i dati letti dalla memoria. Non effettua la estensione del segno, i bit rimanenti sono a 0.
- sb (store Byte) Esempio: sb $t0, 50 ($s1). Scrive nel registro gli 8 bit meno significativi.
- sh (store half word) Esempio: sh $t0, 50 ($s1). Scrive 16 bit meno significativi e li salva in due locazioni di memoria.

Le istruzioni di "store" effettuano una scrittura su registri grandi 1 Byte, essi sono molto più piccoli rispetto ai 32 bit di lunghezza dei registri. Per questo motivo non sono presenti le istruzioni "store unsigned" in quanto non si ha mai un dato più piccolo della locazione in memoria centrale.

Il MIPS è big-endian e la prima locazione di memoria del dato è quella con i bit più significativi.

20.2 Formato e Indirizzamento per trasferimento di Byte e half word

Utilizzano il Formato tipo I e indirizzamento Base-Offset.

20.3 Tipi di dato primitivi per gli interi

Le istruzioni MIPS a 8, 16 e 23 bit vengono utilizzate nei linguaggi ad alto livello per la gestione dei vari tipi di dato primitivo. Nei linguaggi ad alto livello è possibile effettuare conversioni mediante:

- widering conversion (conversione di ampliamento): aumenta il numero di bit della rappresentazione, tutti i valori originali sono rappresentabili
- narrowing conversion (conversione di restringimento): si restringe il numero di bit della rappresentazione, non tutti i valori originali sono rappresentabili.

21 Tabella dei registri del processore MIPS

Registri utilizzati per i dati:

- da $t0 a $t7 hanno l'indirizzo che parte da 8 ed arriva a 15
- da $s0 a $s7 hanno indirizzo che parte da 16 ed arriva a 23
- $t8 e $t9 hanno indirizzo 24 e 25

Indirizzo che punta alla istruzione successiva da eseguire:

- Program Counter (PC)

Gestione delle chiamate

- $sp – stack pointer, puntatore in memoria

22 Glossario

- ARCHITETTURA: insieme dei criteri di base sui quali è progettato e realizzato un computer
- Teoria della computazione: Studio dei problemi che possono essere risolti mediante computazione algoritmica
- PROGRAMMA : algoritmo scritto in un Linguaggio di Programmazione
- ISTRUZIONE: azione operazione scritta in un Linguaggio di Programmazione.
- Linguaggio di Programmazione: linguaggio formale utilizzato per la codifica di algoritmi
- BUS: Binary Unit System
- Locazione di memoria: Registro in memoria principale (NON IN CPU)
- Frame del Call Stack: blocco nel Call Stack relativo ad una chiamata.
- Assembly: linguaggio formale direttamente traducibile in linguaggio macchina.
- Assembler: traduttore da Assembly a linguaggio macchina.
- UTF: Unicode Transformation Format
- Pseudoistruzioni (Assembly): funzioni Assembly NON direttamente eseguibili dalla macchina ma realizzabili attraverso altre istruzioni (registro $at apposito per Assembler)
- Completezza funzionale: data una qualsiasi funzione booleana bastano AND, OR, NOT per costruire una espressione con la stessa tavola di verità della funzione data.
- Selettore di dati di tipo 2^N-1: Multiplexer

23 Link utili

- https://en.wikibooks.org/wiki/MIPS_Assembly/MIPS_Details
- https://en.wikibooks.org/wiki/MIPS_Assembly/Print_Version
- https://en.wikibooks.org/wiki/MIPS_Assembly/MIPS_Details – registri + assembly/codice macchina
- https://homepage.divms.uiowa.edu/~ghosh/6016.2.pdf – registri
- https://homes.di.unimi.it/pedersini/AER/AE2_15_L3.pdf – MIPS 32, vario
- https://www.andreaminini.org/elettronica/circuiti-logici/porte-logiche – porte logiche

24 Numeri ricorrenti

- 32 registri nel processore da 0 a 31 (5 bit)
- 32 bit di lunghezza per ogni registro
- 5 bit necessari per avere l'INDIRIZZO del registro (sono 32 registri quindi 2^5)
- 32 bit di lunghezza di qualsiasi istruzione
- 32 registri per i numeri FLOAT ($fx), oltre ai 32 registri già visti
- Ogni locazione di memoria è grande 1 Byte (8 bit), quindi per una istruzione o dato da mandare nel processore servono 4 locazioni consecutive
- 2^{32} registri massimi di memoria principale indirizzabile, 4Gbyte o 1Giga di parole
- Salto in notazione Unicode tra 'A' ed 'a' di 32 caratteri

25 Pagine importanti

- 135 spiegazione su formato molto specifica. IMP
- 313->316 – recap di formato e indirizzamento di varie istruzioni. Elenco completo.
- 219 + 222 pagine di memoria.
- 332 e seguenti, esempi applicativi su Array e Stringhe. **IMPORTANTE**. Lezione 22.
- 359 – spiegazione numeri con la virgola IEEE 754
- 397->405 – pagina delle principali proprietà dell'algebra booleana
- 354-545 – implementazione unità di controllo
- 550 - segnali di controllo dell'ALU
- 556 – tavole di verità per il funzionamento di CU e CU ALU
- 557 – CPU completa