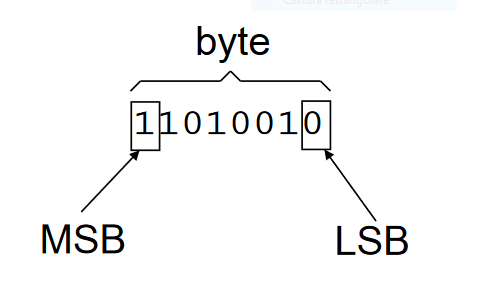
ARCHITETTURA DEGLI ELABORATORI

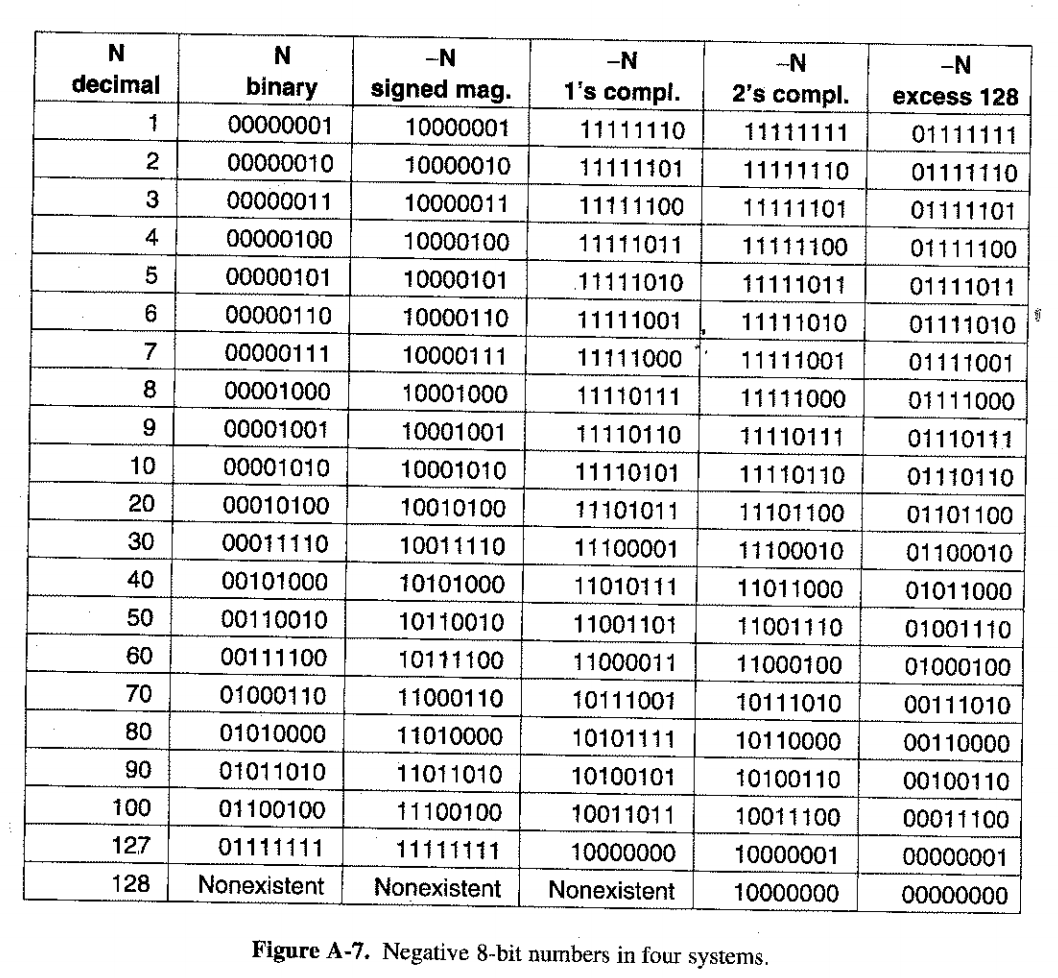
# Rappresentazione dell’informazione

Introduzione

* Una rappresentazione è un modo per descrivere un’entità
* Proprietà principali di un sistema numerico posizionale:
  + Ogni numero intero può essere rappresentato (rango illimitato)
  + A ogni numero intero corrisponde un solo insieme ordinato di cifre (rappresentazione unica)
* **Byte**: una sequenza di otto bit consecutivi
  + **Most Significant Bit (MSB):** il bit più a sinistra
  + **Least Significant Bit (LSB):** il bit più a destra
* La rappresentabilità dei valori è legata al numero di cifre disponibili
* Il numero di cifre impiegate nella rappresentazione di valori numerici è limitato
* Si ha **overflow** quando si è nell’impossibilità di rappresentare il risultato di una operazione con il numero di cifre a disposizione

Numeri Interi

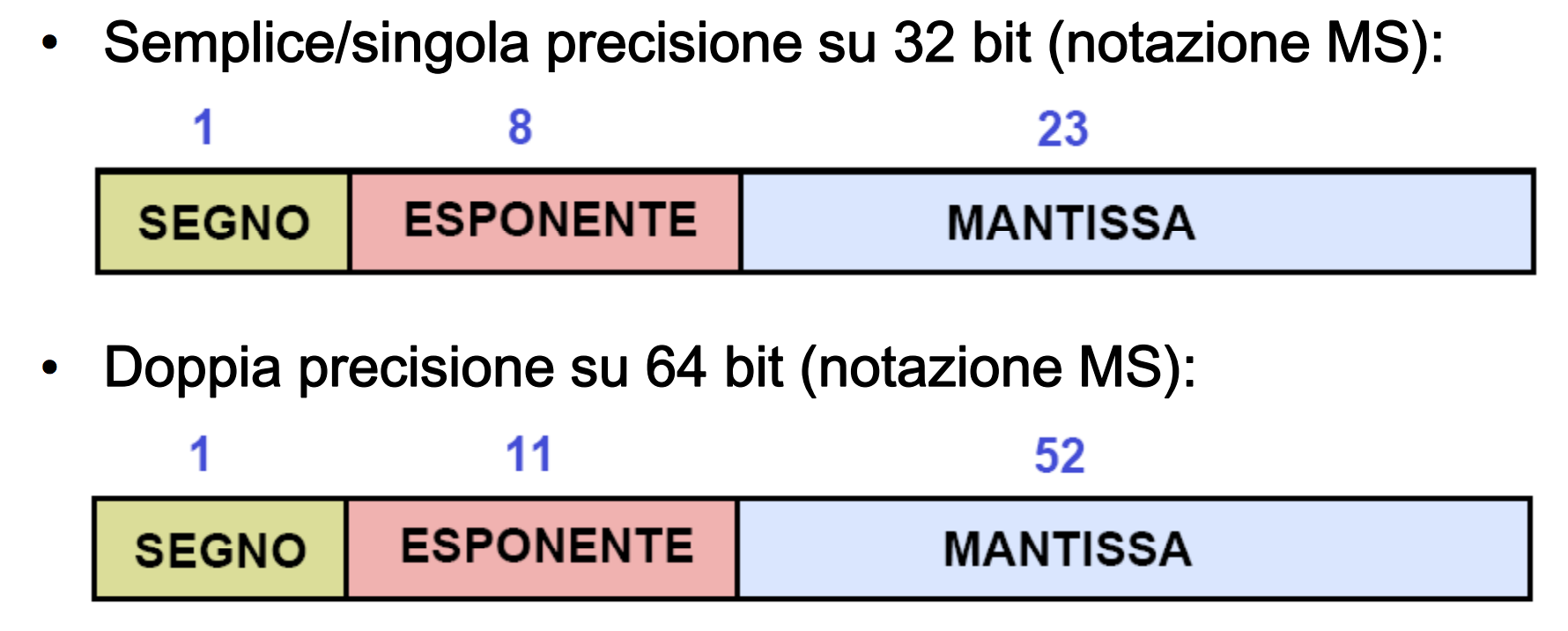
* **MODULO E SEGNO**
  + Aggiungo a sinistra del numero binario un bit per il segno
    - 0 🡪 POSITIVO
    - 1 🡪 NEGATIVO
* **COMPLEMENTO A 1**
  + Nego tutte le cifre
* **COMPLEMENTO A 2**
  + Complemento a 1 🡪 Aggiungo +1
* **ECCESSO 128**
  + Al numero da convertire aggiungo 128, poi trasformo in binario



**NB: In complemento a 1, il riporto più a sinistra viene sommato alla fine, mentre in complemento a 2 viene SCARTATO**

Numeri razionali

* Con un numero finito di cifre è solo possibile rappresentare un **numero razionale che approssima con un certo errore il numero reale dato**
* Vengono usate due notazioni:
  + **Notazione in virgola fissa** 
    - Dedica parte delle cifre alla parte intera e le altre alla parte frazionaria parte frazionaria: **± XXX .YY**
    - Il numero di bit della parte frazionaria e quello della parte intera sono **costanti**
    - Per le operazioni aritmetiche bisogna **incolonnare virgola sotto virgola** gli addendi
    - I **numeri negativi** possono essere rappresentati in MS o CA2
  + **Notazione in virgola mobile**
    - Dedica alcune cifre a rappresentare un esponente della base che indica l’ordine di grandezza del numero rappresentato
    - La posizione della virgola non è fissa, ma varia per avere una rappresentazione in notazione scientifica in cui:
      * Un’unica cifra a sinistra della virgola
      * Una parte frazionaria una parte frazionaria
      * Un esponente al quale si deve elevare la base del numero



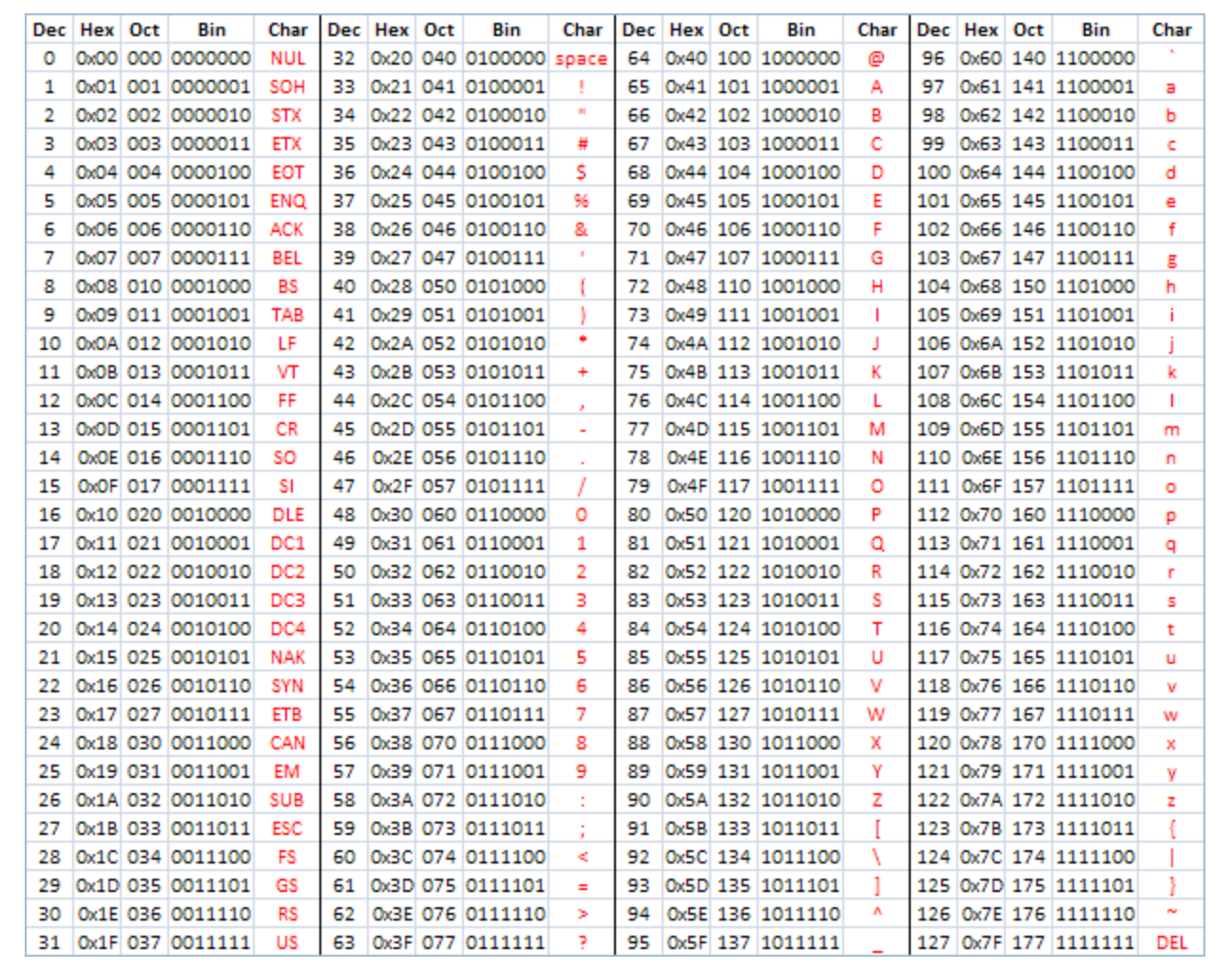
* + - L’esponente può assumere valori negativi (quindi i numeri in virgola mobile possono essere rappresentati in MS e CA2)
    - **ESPONENTE IN ECCESSO 127**

Tabella ASCII

* Codice su 7 bit
* Usato per la codifica dei caratteri
* Assegna un codice univoco a un carattere

ASCII Estesa

* Codice su 8 bit



UNICODE

* Inizialmente rappresentato come una codifica su 16 bit, ma poi esteso a 24 e 32 bit a 24 e 32 bit
* Unicode è in continua evoluzione e continua ad aggiungere più caratteri

## Reti logiche

Introduzione

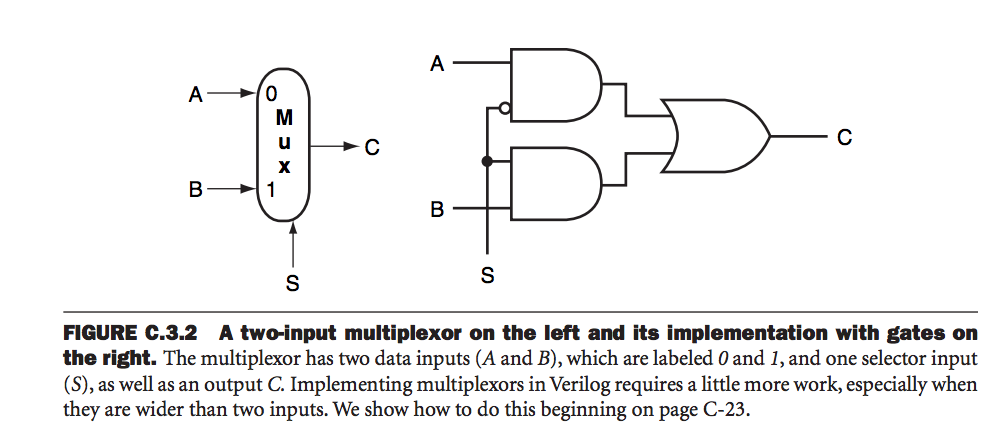
* **RETI COMBINATORIE**
  + Combinano gli ingressi
* **RETI SEQUENZIALI**
  + Combinano sia gli **INGRESSI PRECEDENTI** sia quelli **ATTUALI**

## Reti combinatorie

Decoder

* Inputs 
  11 
  12 
  Outl 
  Out7 
  a. A 3-bit decoder 
  Outputs 
  10 out7 oute outs out4 out3 Out2 Outl outo 
  b. The truth table for a 3-bit decoder Componente con n input e 2^n output

Multiplexer

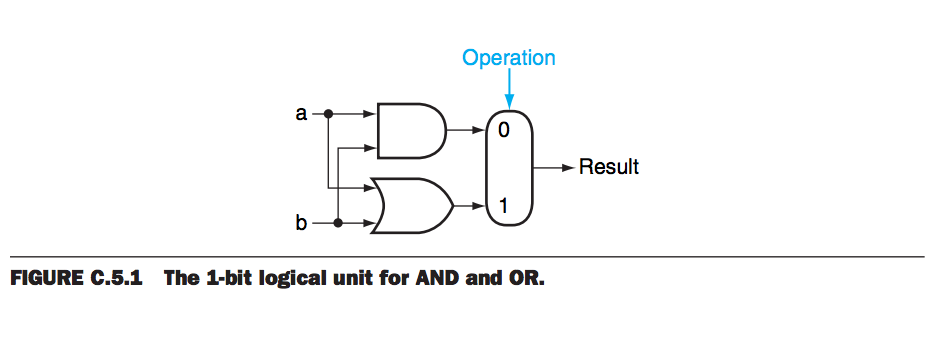
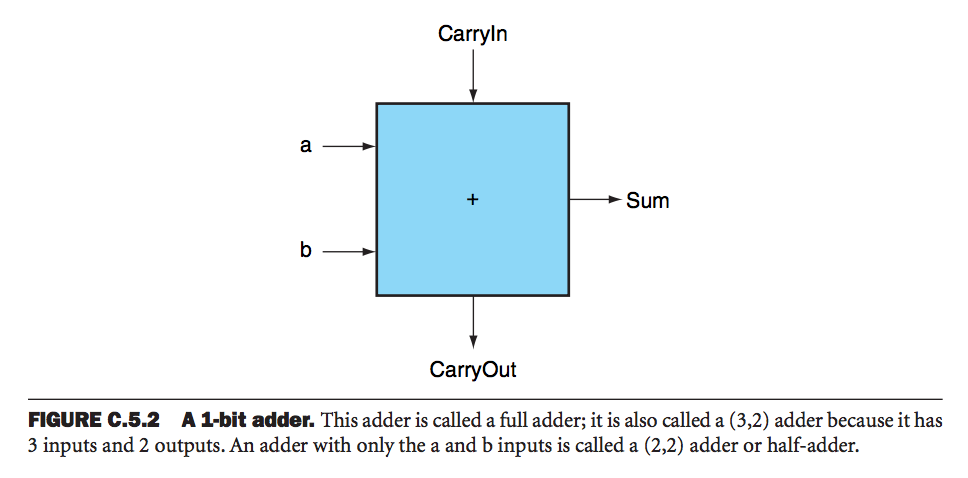


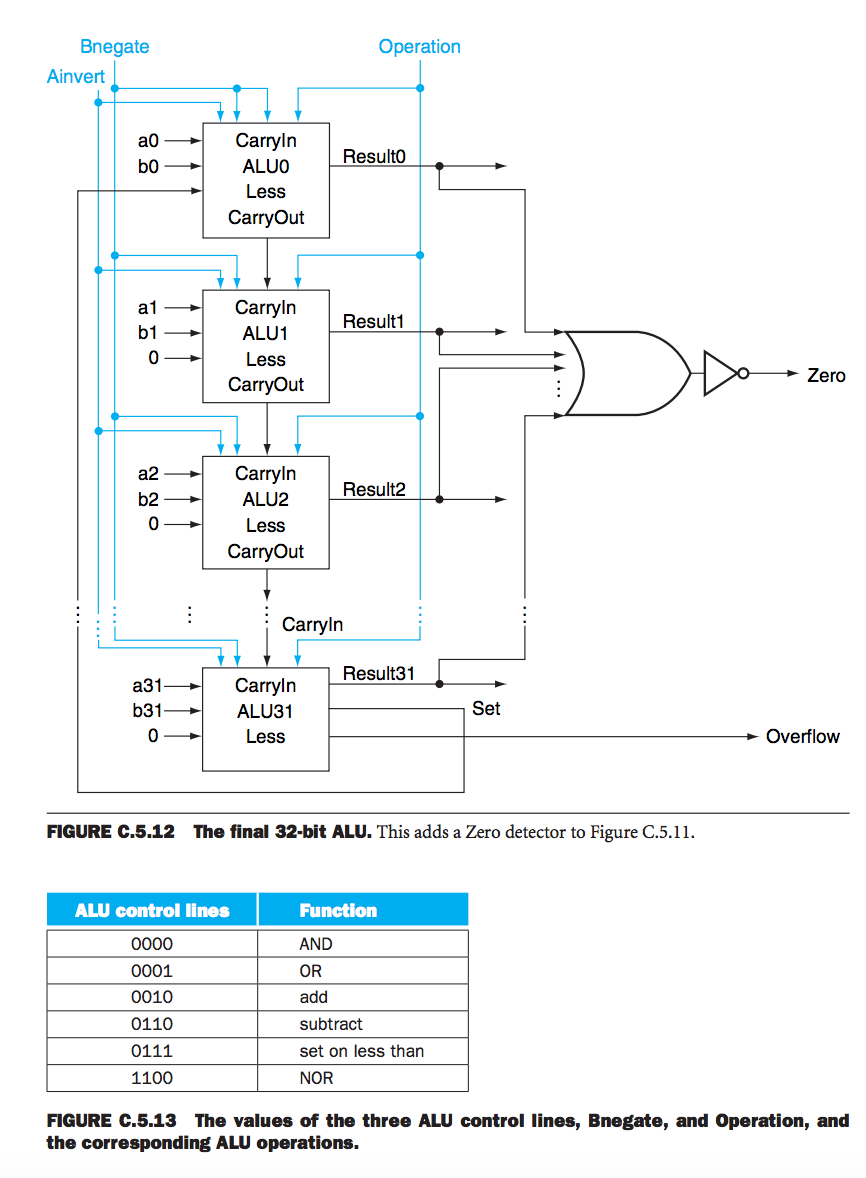
* Componente con **n input, 2^m selettori** e **1 output**

ROM

* Read-Only Memory
* “Memory” è fuorviante
* Ha **2^n indirizzi (altezza)** e **2^n output di larghezza variabile**
* Ne esistono diversi tipi
  + PROM 🡪 Programmable
  + EPROM 🡪 Erasable-Programmable
  + EEPROM 🡪 Electric-Erasable-Programmable

ALU (1-bit)

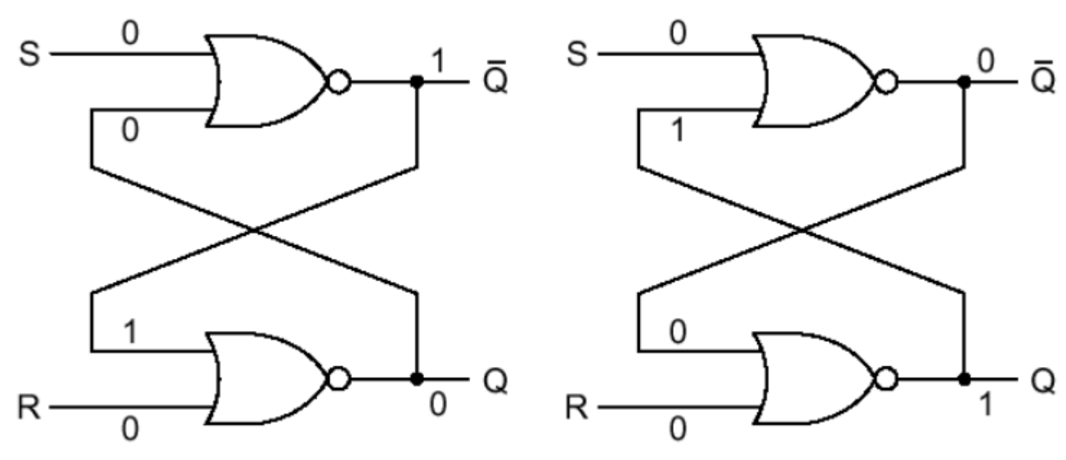
* Gestisce dati a 32 bit
* Modalità didattica 🡪 32 ALU a 1 bit
* **PARTE LOGICA:**
* **PARTE ARITMETICA:**
* **SCHEMA COMPLETO:**



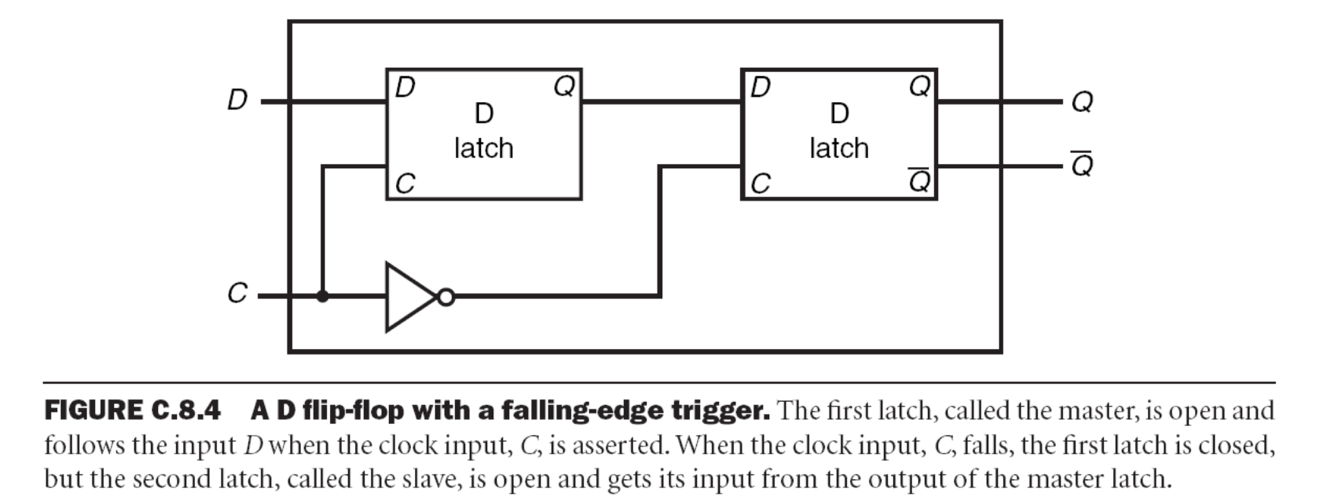
## Reti sequenziali

* Per realizzare circuiti sequenziali è necessario un elemento di memoria per memorizzare lo stato

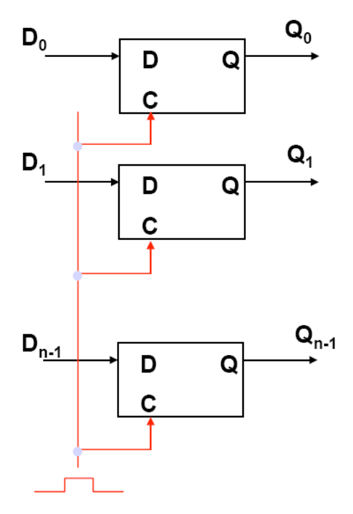
SR Latch

* S-R Latch è composto da 2 porte NOR concatenate
* Costituisce l’elemento base per costruire elementi di memoria più complessi
* Memorizza 1 bit e ricorda i valori di input precedenti
* La combinazione S=1 e R=1 non deve essere mai presentata al latch perché sarebbe Q = NOT Q = 0, ma il valore può essere arbitrario in quanto dipende dall’ordine del resetting

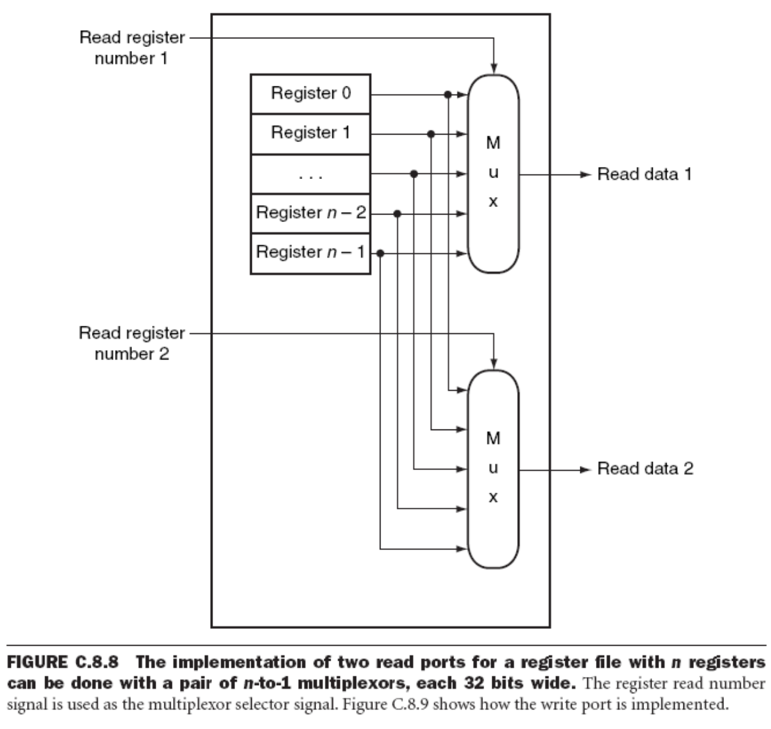
Flip Flop D

* Usa metodologia edge-triggered

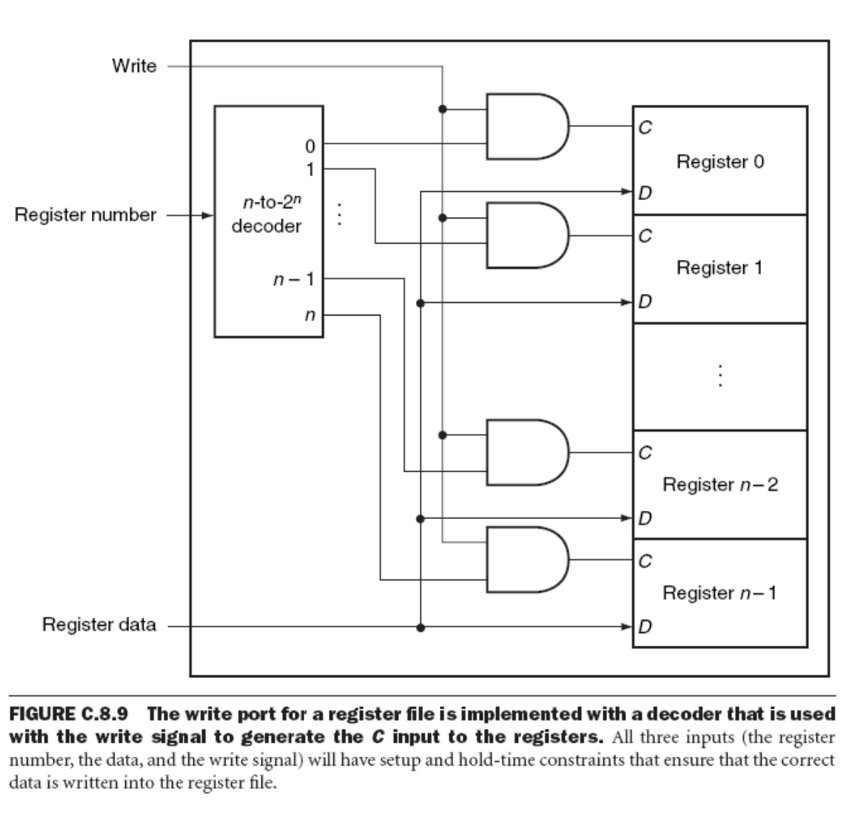
Register File

* Il **Datapath** (la parte operativa della CPU) contiene (oltre alla ALU) anche **registri** che **memorizzano** (all’interno della CPU) **gli operandi** delle istruzioni aritmetico/logiche
* Un registro è costituito da **n flip-flop** dove **n è il numero di bit di un registro**
  + Nel MIPS ogni registro è di 1 word = 4 byte = 32 bit
* I registri sono organizzati in un Register File
  + Il Register File del MIPS ha 32 registri (32 x 32 = 1024 flip-flop)
  + Il Register File permette la lettura di 2 registri e la scrittura di 1 registro
* Nel Datapath della CPU il clock non entra direttamente nei vari flip-flop; viene messo in AND con un segnale di controllo “Write”
* Il segnale Write (in AND con il clock) determina se, in corrispondenza del fronte di discesa del clock, il valore D debba (o meno) essere memorizzato nel registro

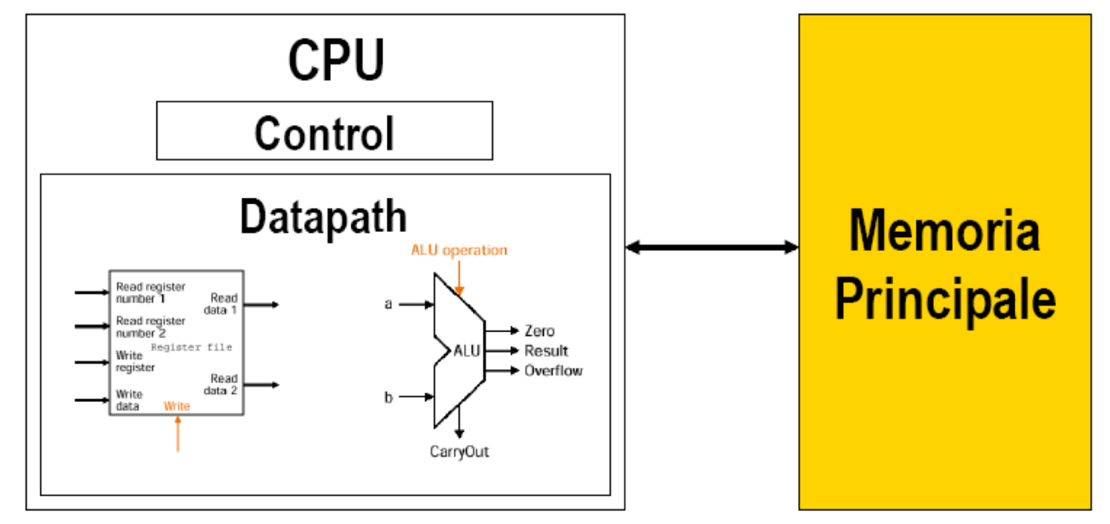
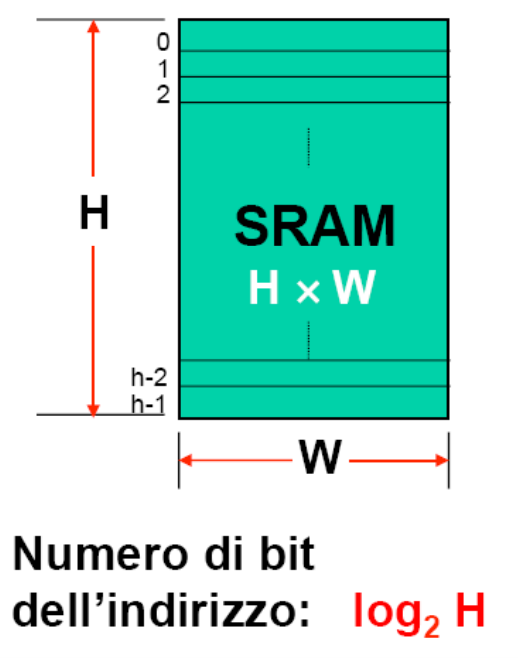
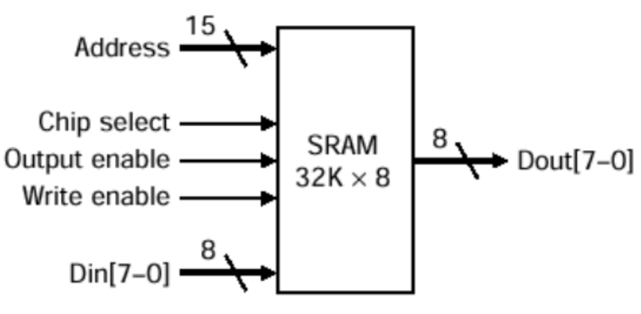
Lettura da Register file

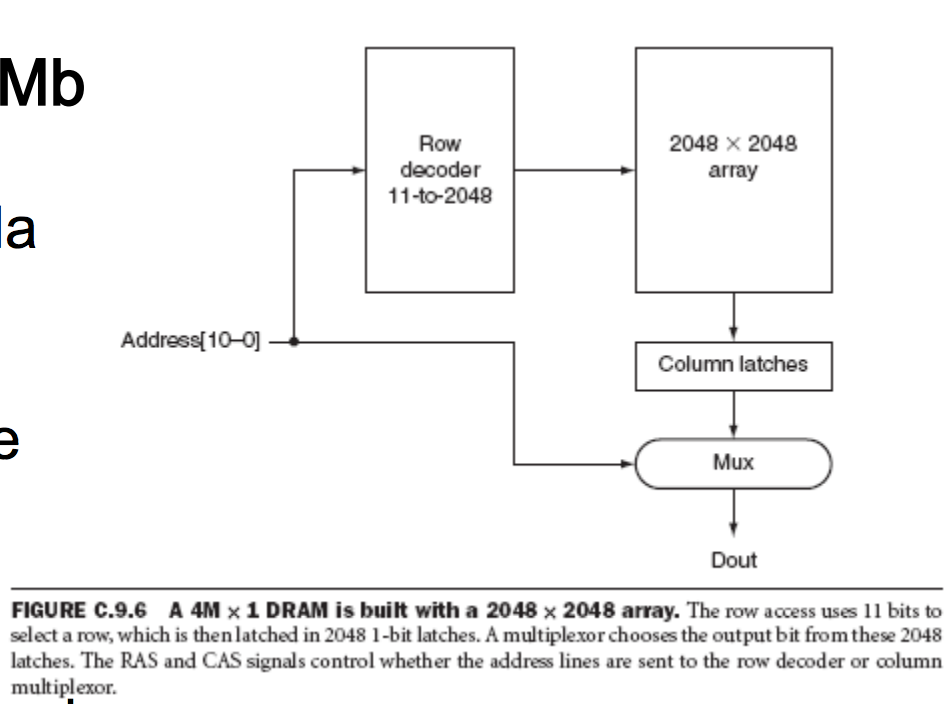
* Utilizza 2 segnali che indicano i registri da leggere (Read Reg1, Read Reg2)
* Utilizza 2 multiplexer: ognuno con 32 ingressi e un segnale di controllo da 5 bit (Read Reg1, Read Reg2)
  + Osservazione: Il register file fornisce sempre in output una coppia di registri; se i segnali di read non sono significativi tali registri vanno ignorati

Scrittura nel Register File

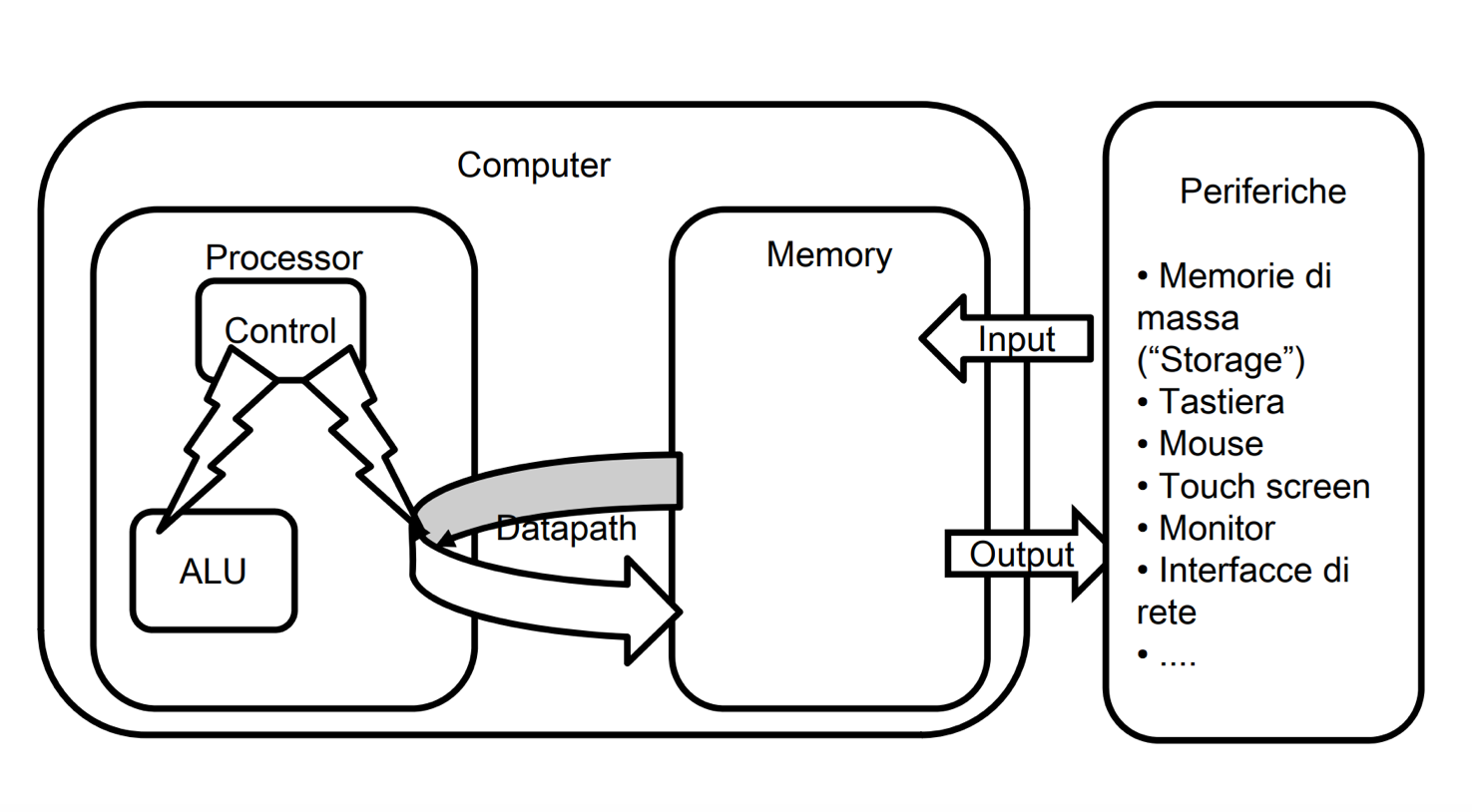
* Utilizza 3 segnali:
  + Il registro da scrivere (Register Number)
  + Il valore da scrivere (nel Register Data)
  + Il segnale di controllo Write che abilita la scrittura nel Register File
* Utilizza un decoder che decodifica il numero del registro da scrivere
* Il segnale Write (già in AND con il clock) è in AND con l’output del decoder
* Se Write non è affermato nessun valore sarà scritto nel registro
* Cosa succede se uno stesso registro del Register File viene acceduto in lettura e scrittura durante lo stesso ciclo di clock?
  + Sarà letto il valore memorizzato nel ciclo di clock precedente, mentre sarà scritto in nuovo valore indicato perché la scrittura sul registro avviene sul fronte di discesa del clock

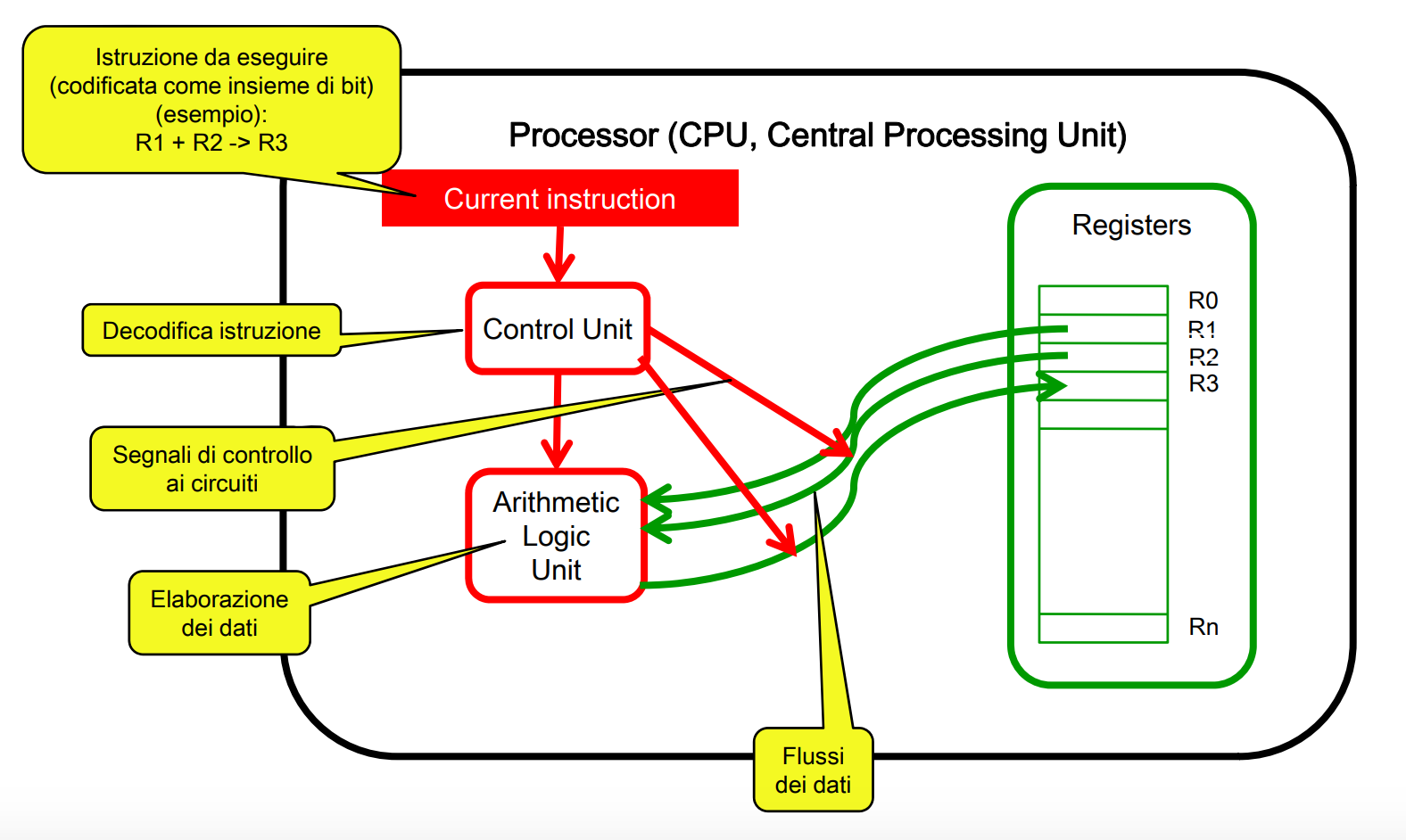
Memoria Centrale

* Per memorizzare i dati è necessaria una mole di memoria molto superiore
* Si ricorre all’utilizzo di **RAM**
  + **Meno veloce** dei registri
  + **Molto più capiente**
* **SRAM**: Static Ram
  + Usata per costruire memorie più veloci come la cache
  + Tempi di accesso: 0.5 – 2.5 ns
  + Realizzata come **matrice di latch**
    - Larghezza o ampiezza W (numero di latch per ogni cella)
    - Altezza H (numero di celle indirizzabili)
    - Singolo indirizzo usato sia per lettura che per scrittura
    - Non è possibile leggere e scrivere contemporaneamente (come per Register File)
  + ESEMPIO 1:
    - 32K x 8 (32 celle da 8 bit 🡪 256Kb)
    - 2^18 linee di indirizzo
    - 8 linee di output
  + **Peculiarità**:
    - **Non si possono usare multiplexer e decoder come per Register File**
    - Con un numero elevato di celle di memoria avremmo bisogno di decoder e multiplexer **enormi**
* **DRAM**: Dynamic Ram
  + Ogni bit è memorizzato tramite un condensatore (non usa latch)
  + Usata per memorie capienti come quella principale
  + Tempi di accesso: 50-70 ns
  + È necessario rinfrescare il contenuto della DRAM a intervalli di tempo prefissati
  + **MENO COSTOSA, PIÙ CAPIENTE, PIÙ LENTA DELLE SRAM**
  + ESEMPIO:
    - DRAM di 4M x 1 = 4Mb
    - Indirizzo totale su 22 bit
    - Indirizzo spezzato in 2 pezzi da 11 bit ciascuno
    - Parte alta e parte bassa dell’indirizzo considerate come indirizzo di riga e di colonna:
      * Indirizzo di riga ha effetto sul decoder
      * Indirizzo di colonna ha effetto sul multiplexer



## ISA (Instruction Set Architecture)

The “big picture”

Processore: Istruzione base sui registri

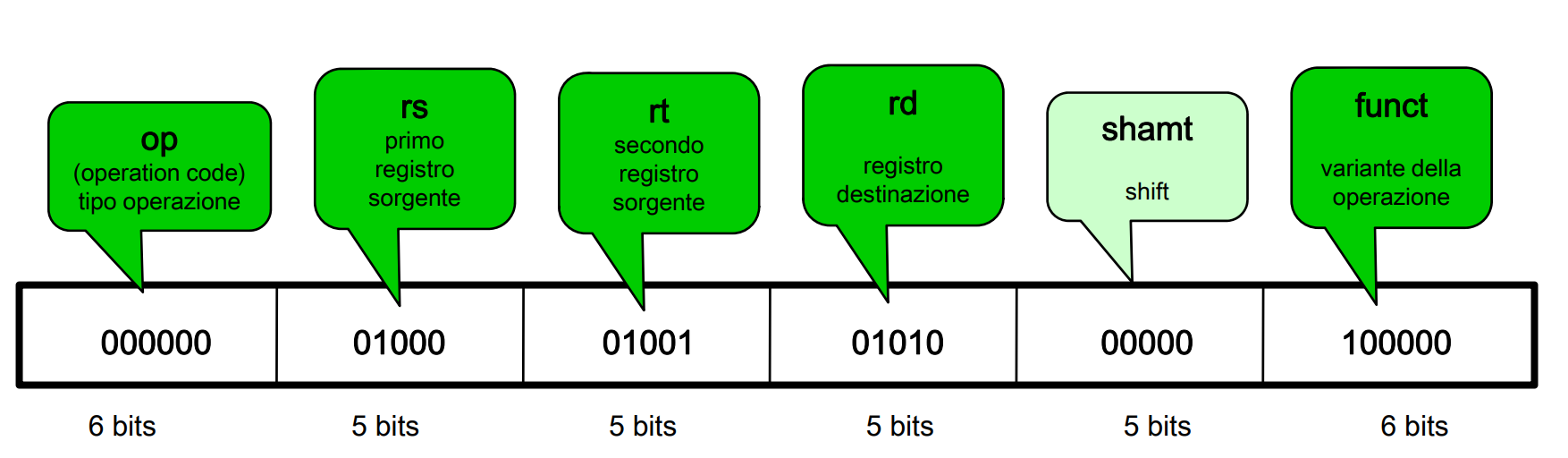
Tipologie progetto CPU

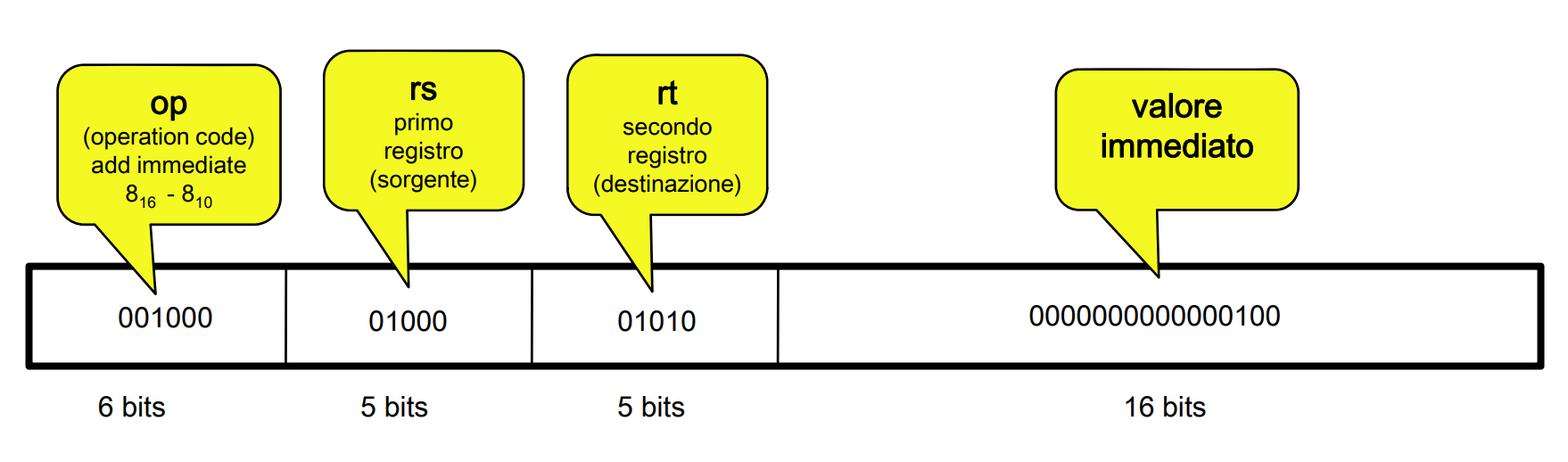
* **RISC 🡪 Reduced Instruction Set Computing** (ES. MIPS)
* **CISC 🡪 Complex Instruction Set Computing**

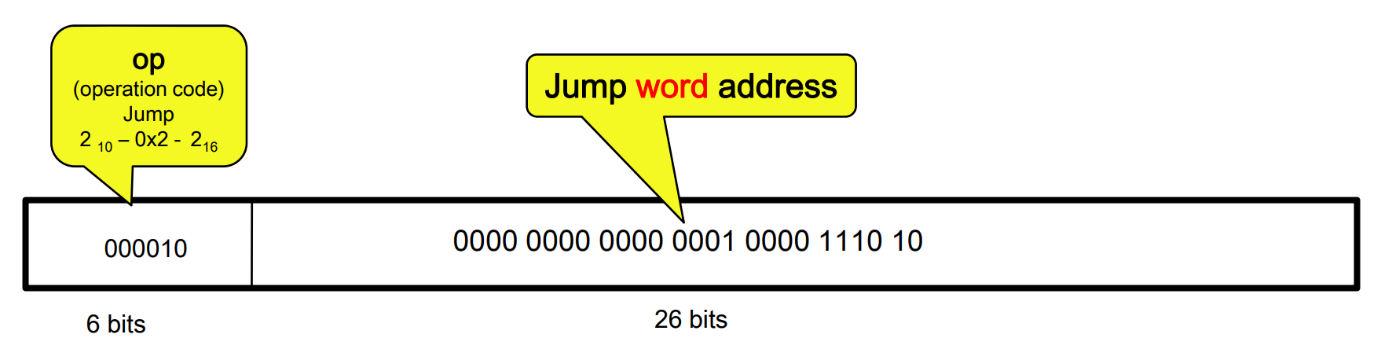
Architettura MIPS

* **RISC**
* **32 Registri di 32 bit**
* **Istruzioni di 32 bit (4 byte)**
  + Manipolazione dati SOLO sui registri
  + Trasferimento dati tra memoria e registri
  + Alterazione del flusso di controllo (Jumps)
* Problema architetturale
  + Rappresentazione di tutte le operazioni in 32 bit
* **TIPI DI ISTRUZIONE**
  + **R-TYPE**
  + **I-TYPE**
  + **J-TYPE**

Istruzioni R-TYPE

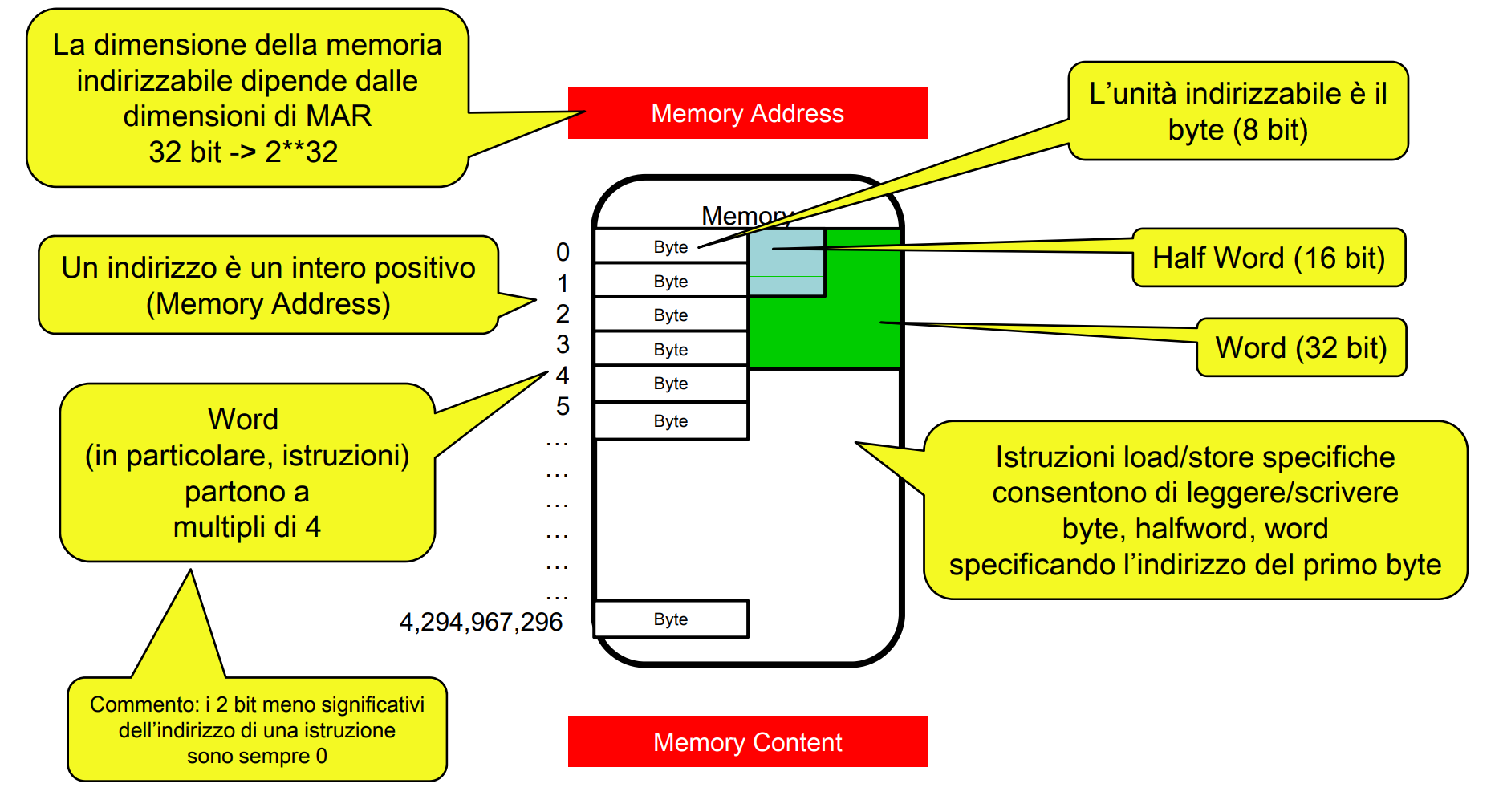


Istruzioni I-TYPE

Istruzioni J-TYPE

00 Sottintesi

Indirizzamento della memoria



Tipi di addressing

* Immediate 🡪 I-TYPE
* Register 🡪 R-TYPE
* Base 🡪 I-TYPE (Registro base + Offset)
* PC-Relative 🡪 I-TYPE (PC + Offset)
* Pseudodirect 🡪 J-TYPE

**NB**: L’OFFSET DELLE OPERAZIONI LW, SW ecc. CORRISPONDE AI BYTE

lw $t0, 56($s0)

add $t1, $s1, $t0

sw $t1, 68($s0)

(3 parole più avanti)

Branches

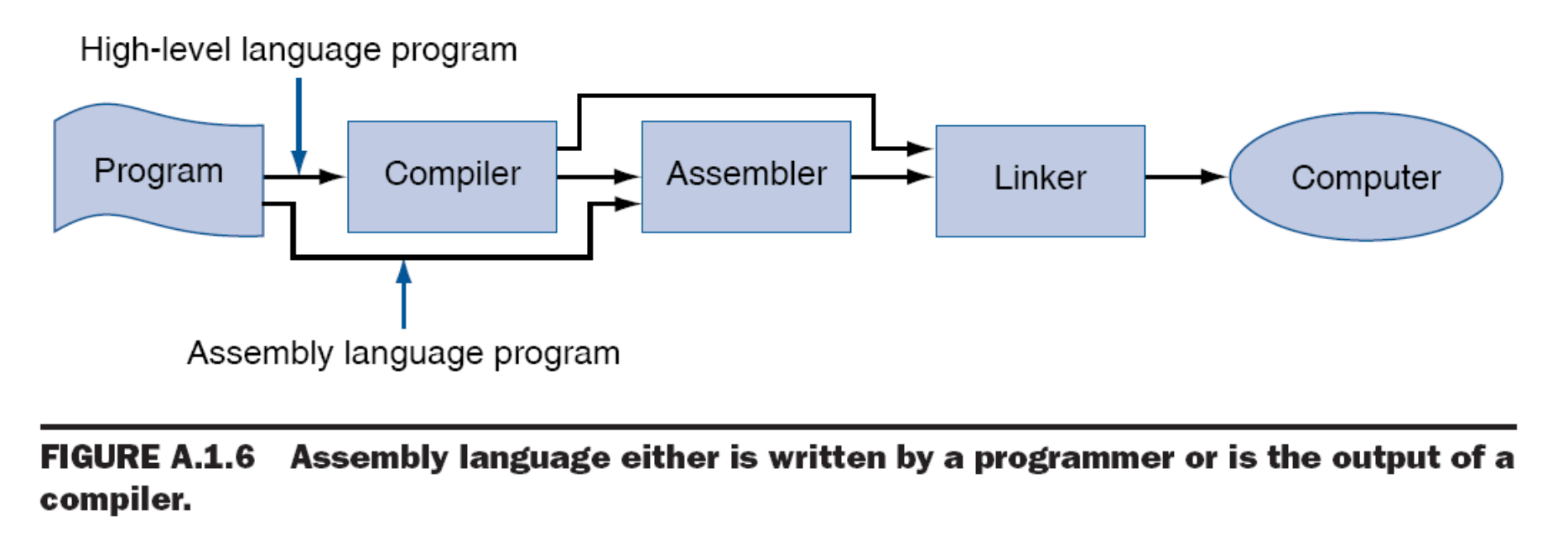
* Finestra 🡪 – 2^15 <= BA < 2^15
  + Viene usato il PC come “base” da cui eseguire l’offset
  + **NB**: Se CA è l’indirizzo dell’istruzione corrente, si salta a **CA + 4 + BranchAddr\*4**, dove BranchAddr corrisponde al numero di operazioni da saltare (ramo false del branch)

## Linguaggi

Caratteristiche

|  |  |  |
| --- | --- | --- |
| MACCHINA | ASSEMBLY | ALTO LIVELLO |
| * Direttamente comprensibile * Programmazione più lunga * Facile commettere errori | * Rappresentazione simbolica * Più comprensibile * Tradotto dall’assemblatore in linguaggio macchina | * Tradotti dal compilatore in linguaggio assembler * Simile al linguaggio corrente * Incremento di produttività * Portabilità |

## Catena programmativa



Compilatore

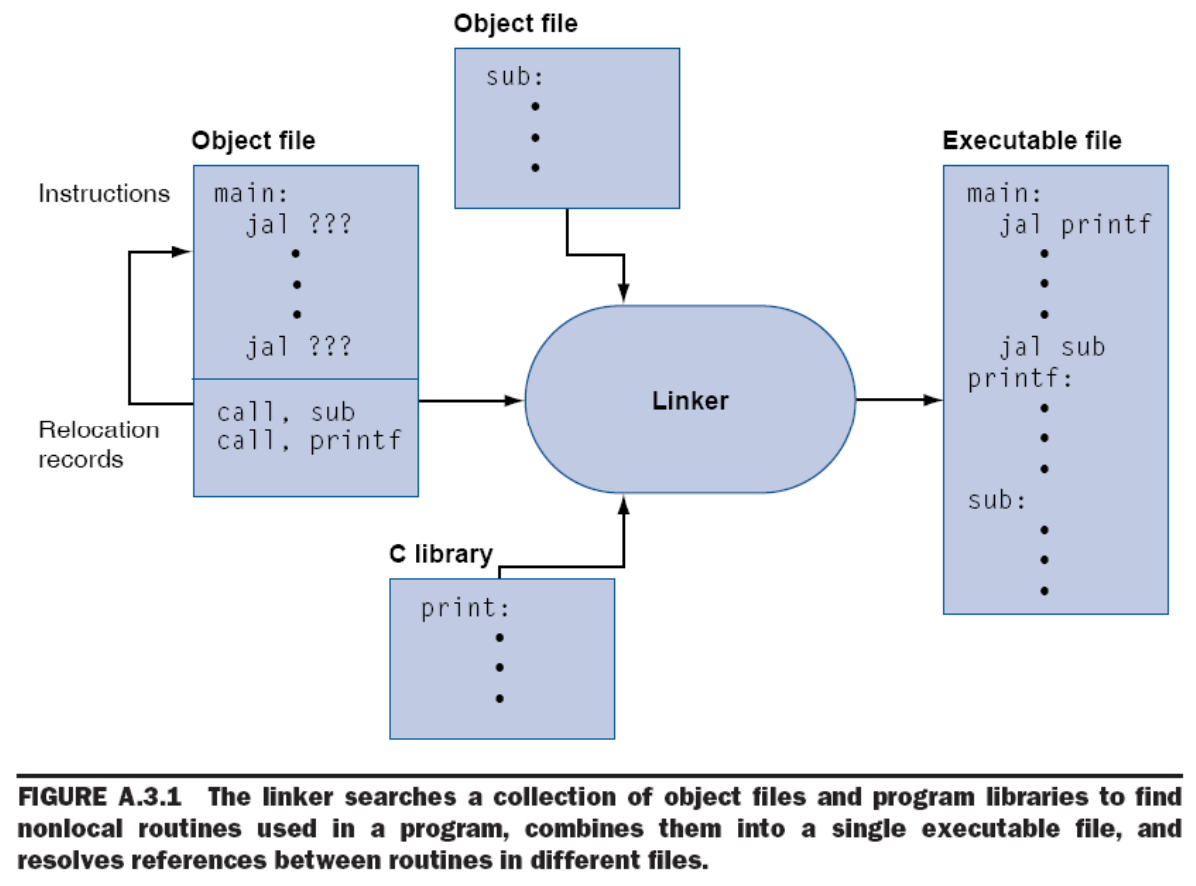
* Ha il compito di tradurre un programma scritto in linguaggio ad alto livello in linguaggio assembler
* Spesso con il termine compilazione si indica l’intero processo di traduzione da linguaggio ad alto livello a linguaggio macchina (essendo ad alto livello a linguaggio macchina (essendo l’assemblatore spesso integrato con il compilatore)

Assemblatore

* **Converte** un programma assembler (file sorgente) in linguaggio macchina (file oggetto)
* Gestisce le **etichette**
* Gestisce **pseudoistruzioni**
* Gestisce **numeri in base diverse**
* **Assemblaggio:**
  + Procedimento sequenziale di traduzione
  + Applicato modulo per modulo, costruendo la **tabella dei simboli**
  + Traduce i **codici simbolici in istruzioni**
  + Traduce i **riferimenti simbolici in indirizzi**
* **DEVE LEGGERE IL PROGRAMMA DUE VOLTE PER VIA DEI RIFERIMENTI DELLE ETICHETTE 🡪 TRADUTTORE A DUE PASSI**
* Ogni modulo assemblato di default parte da 0
* **TABELLA DEI SIMBOLI:**
  + **Primo passo 🡪** Contiene i riferimenti simbolici del modulo e i relativi indirizzi numerici
  + Per le **costanti** viene creata la coppia **<etichetta, valore>**
  + Per le etichette che definiscono **variabili** viene riservato lo spazio relativo creando la coppia **<etichetta, indirizzo>**
  + Per le etichette che costituiscono le istruzioni di jump l’assemblatore deve generare un riferimento all’indirizzo di jump
  + Global/Local 🡪 Scope di visibilità dei moduli
  + **NB:** Le etichette esterne **NON** vengono risolte dall’assembler

Linker

* Inserisce in memoria in modo simbolico il codice e i moduli dati
* Determini gli indirizzi dei dati e delle etichette che compaiono nelle istruzioni
* Corregge i riferimenti esterni e interni e **risolve i riferimenti in sospeso**
* **GENERA IL FILE ESEGUIBILE**

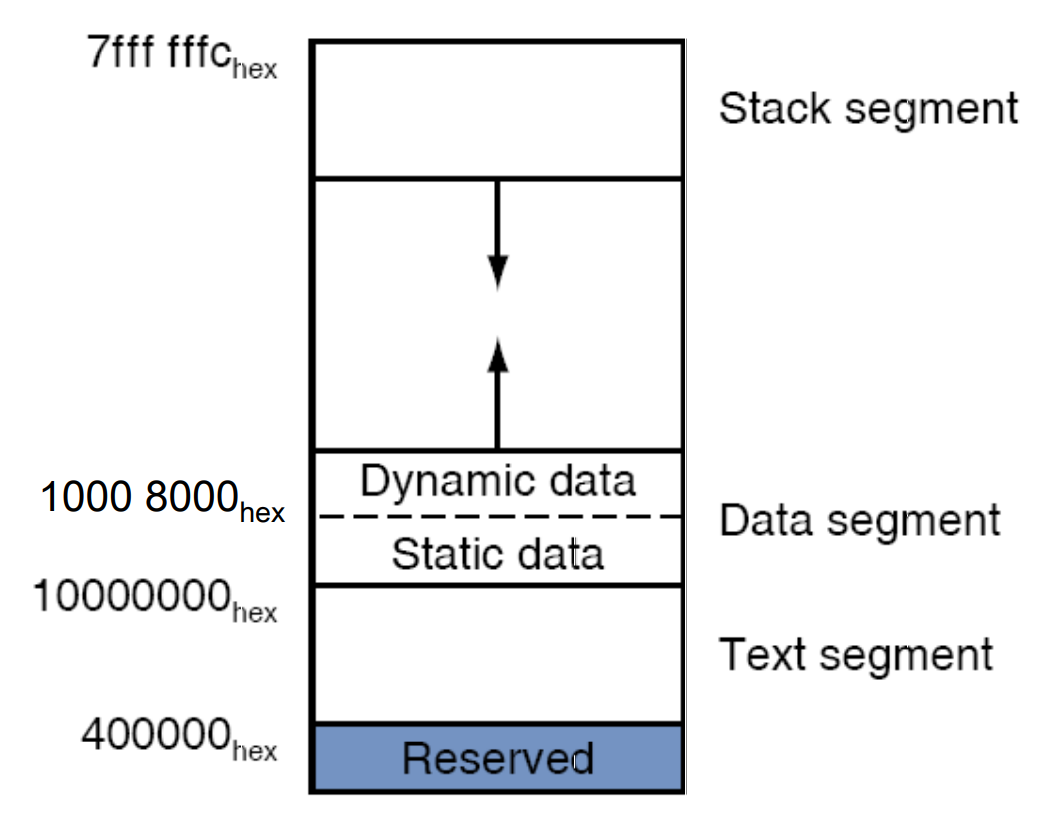


Loader

* **Lettura** dell’intestazione del file eseguibile per determinare la lunghezza di **Text Segment** e **Data Segment**
* **Crea** uno spazio di indirizzamento sufficiente a contenerli
* **Copia** delle istruzioni e dati dal file eseguibile all’interno della memoria
* **Copia** nello stack degli eventuali parametri passati al programma principale
* **Inizializzazione** dei registri e **impostazione** dello **stack pointer** affinché punti alla prima locazioni libera
* **Salto a una procedura di startup** la quale copia i parametri nei registri argomento e chiama la procedura principale del programma
* Quando la procedura principale restituisce il controllo, la procedura di startup termina il programma con una chiamata alla funzione di sistema **exit**

Direttive

* NON corrispondono a istruzioni macchina
* Sono indicazioni date all’assembler per consentirgli di:
  + **associare etichette simboliche a indirizzi**
  + **allocare spazio di memoria per le variabili**
  + **decidere in quali zone di memoria allocare istruzioni e dati**

****

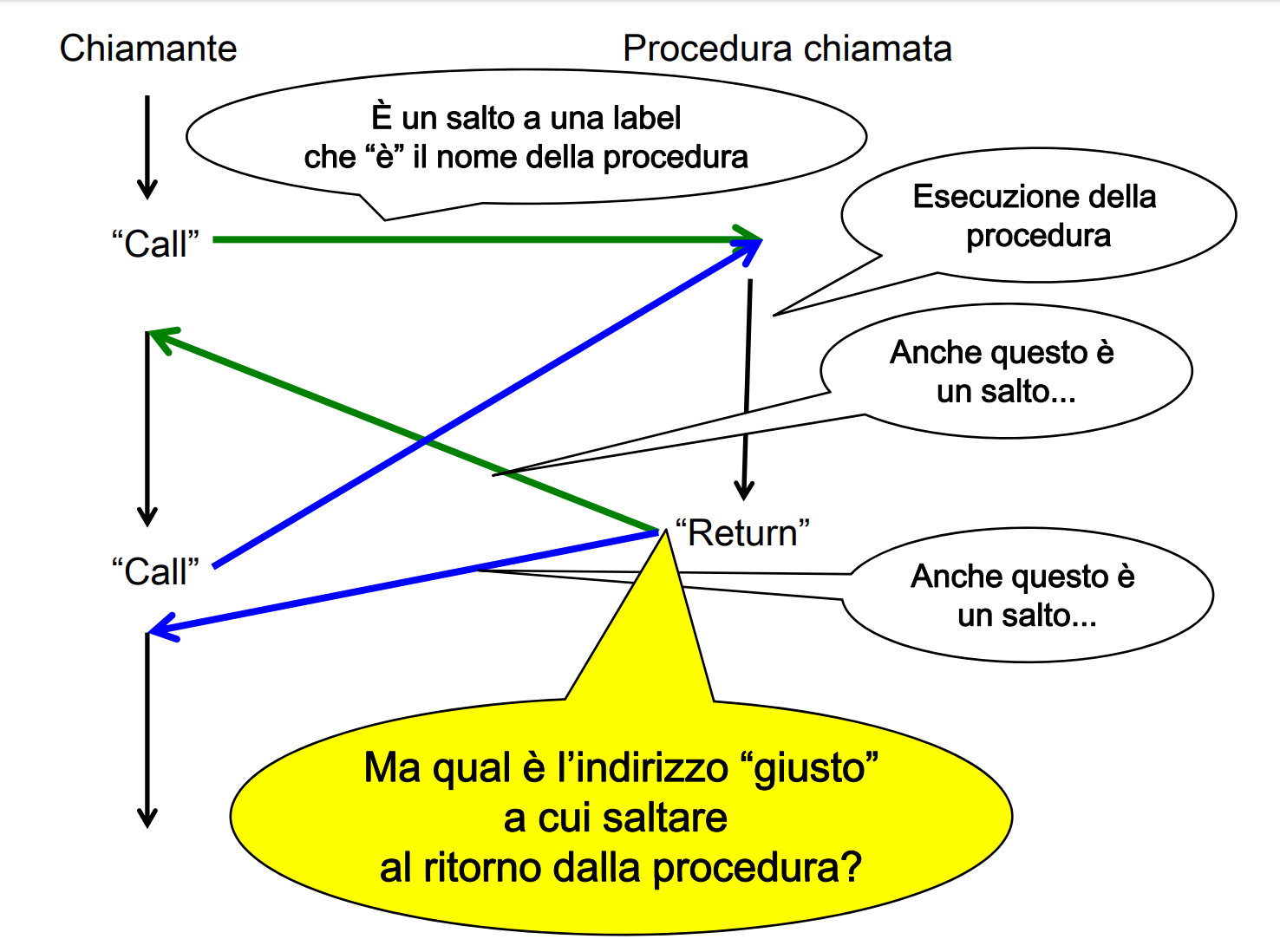
Pseudoistruzioni

* Non corrisponde a una sola istruzione macchina
* Tradotte dall’assembler in più istruzioni

Osservazioni

* I programmi sono immagazzinati in memoria insieme ai dati 🡪 **PROGRAMMA MEMORIZZATO**

## Flusso di controllo



Istruzione JAL

* **Salta a una procedura** indicata nell’istruzione e **crea un collegamento a dove deve ritornare per continuare** l’esecuzione del chiamante
* **Salva** nel registro **$ra** (registro 31) (“return address”) l’indirizzo a cui tornare dopo l’esecuzione della procedura (è l’indirizzo successivo a quello dell’istruzione jal, cioè l’indirizzo in cui si trova la jal + 4)
* Tale indirizzo si trova nel registro PC (Program Counter)

Istruzione JR

* **Salta all’indirizzo contenuto in un registro**
* **Caso speciale 🡪** jr $ra 🡪 Serve per effettuare il ritorno da una procedura all’indirizzo salvato precedentemente in un jal

Usi convenzionali dei registri

* $a0 - $a3 🡪 Argomenti procedure
* $v0 - $v1 🡪 Registri valore per la restituzione di risultati
* Le convenzioni permettono di scrivere procedure che:
  + Possono essere scritte senza bisogno di sapere come è fatto il programma che le chiama che le chiama
  + Possono essere chiamate senza bisogno di sapere come sono fatte “dentro”
* **NB: UN PARAMETRO PUÒ ESSERE UTILIZZATO SIA COME VALORE CHE COME INDIRIZZO**

Osservazioni

* **Una procedura è un modo per organizzare in modo comprensibile e riutilizzabile il codice**
* I 6 passi di una procedura:
  + **Setting** dei **parametri** in un luogo accessibile alla procedura
  + **Trasferire** il **controllo** alla procedura
  + **Acquisire risorse** per l’esecuzione della procedura
  + **Eseguire** il compito richiesto
  + **Mettere il risultato in un luogo accessibile al chiamante**
  + **Restituire il controllo al punto di partenza**
* Cosa succede se una procedura ne chiama un’altra?
  + Si perde il contenuto di $ra della prima chiamata?
* Procedure ricorsive?
  + -> uso dello stack
* Se una procedura usa registri, cosa succede del contenuto lasciato nei registri dal chiamante?
  + Convenzioni: registri $s e $t
* Dove stanno le variabili locali della procedura?
  + Stack frame

Syscall

* Analogo a una chiamata a procedura
* Convenzioni per le syscall:
  + Tabella a pag. A43 (Appendice A)
  + Impostare nel registro $v0 il codice della chiamata
  + Impostare i parametri nei registri $a0-$a3 (come da tabella)

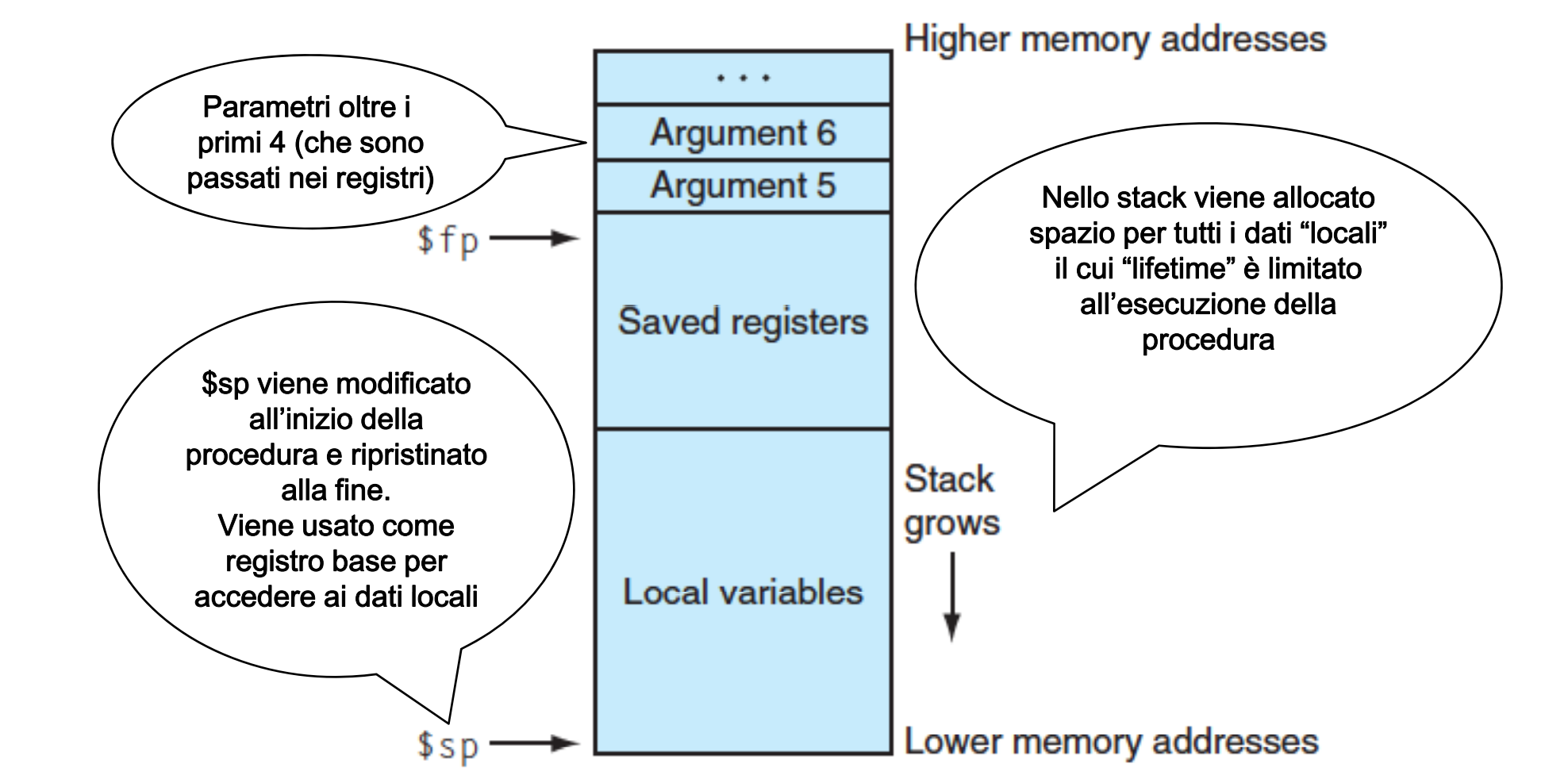
Registri salvati e temporanei

* REGISTRI SAVED ($s)
  + Il chiamante **ha il diritto di aspettarsi che i registri saved siano immutati dopo la procedura chiamata**
  + Se la procedura chiamata ne fa utilizzo **deve ripristinarne il loro valore originale**
  + Dove salvare il contenuto dei registri $s
    - Uso dello stack
* REGISTRI TEMPORANEI ($t)
  + Il chiamante **non si può aspettare di trovare immutati i contenuti dei registri $t dopo una chiamata a procedura**
  + I contenuti dei registri $t **devono essere salvati dal chiamante** prima della chiamata a procedura

Procedure innestate

* Procedure “foglia” e “non foglia”
  + Una procedura foglia NON chiama altre procedure
  + Una procedura non foglia chiama altre procedure
* Cosa succede se una procedura ne chiama un’altra?
  + Si perde il contenuto di $ra della prima chiamata?
  + Procedure ricorsive?
    - Bisogna che una procedura “non foglia” salvi il contenuto di $ra e lo ripristini prima del ritorno

Uso dello Stack



Cosa fa una procedura chiamata

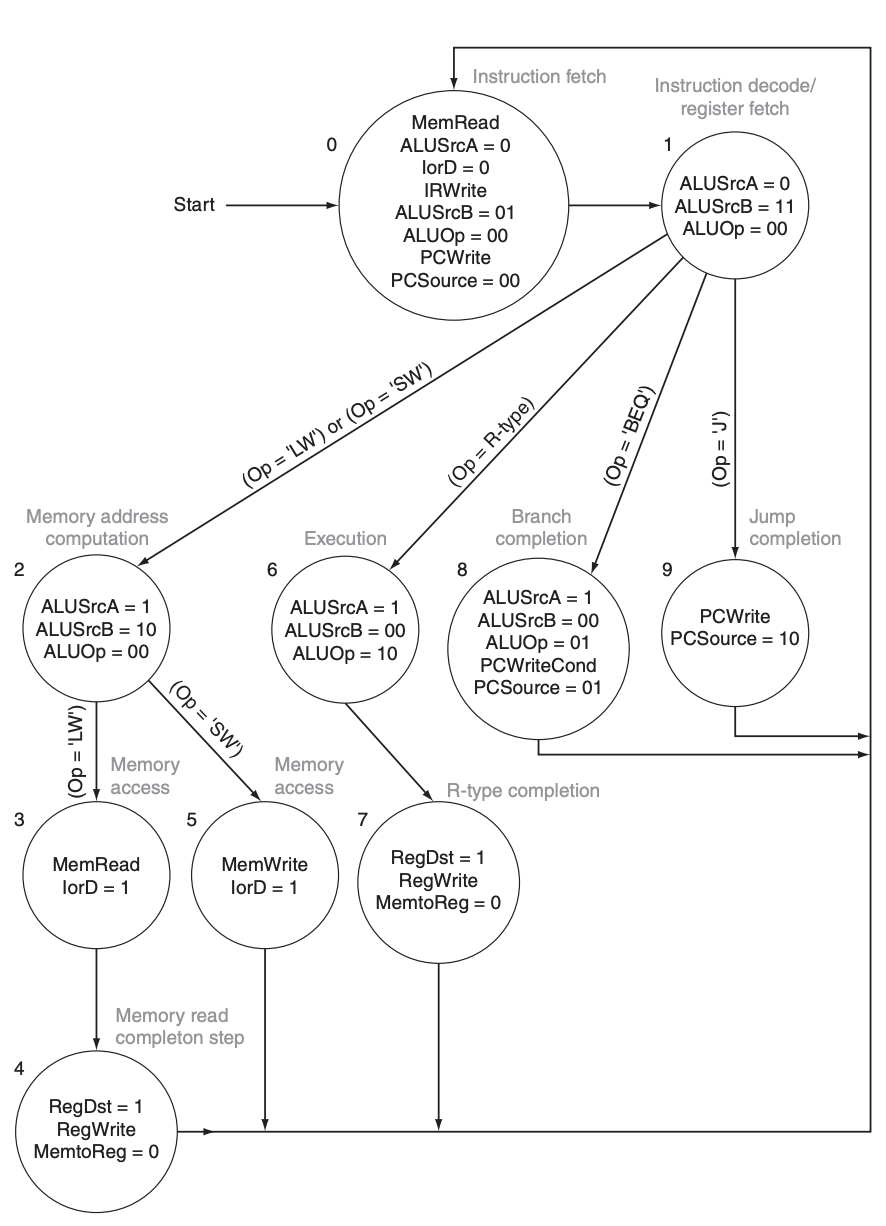
* Alloca spazio nello stack
* Decrementa $sp per lasciare in stack lo spazio necessario al salvataggio (1 word per ciascun registro da salvare) (ricordare che lo stack cresce “verso il basso” )
* Salva $ra
* Salva eventuali altri registri usando $sp come registro base
* Ripristina i registri
* Incrementa $sp per riportarlo alla situazione iniziale
* Jr $ra (ritorno dalla procedura)

Cosa fa una procedura chiamante

* Impostare gli argomenti da passare alla procedura in $a0-$a3 (eventuali altri argomenti sono nella memoria o nello stack)
* Salvare eventualmente i registri $a0-$a3 e $t0-$t9 in quanto la procedura chiamata può usare liberamente questi registri
* Chiamare la procedura tramite l’istruzione jal nome\_procedura

## Datapath





Introduzione

* Le performance di una macchina sono determinate da 3 fattori
  + Numero di istruzioni da eseguire
  + Durata di un ciclo di clock
  + CPI (Clock Cycles per Instruction)

Implementazione

Tutte le operazioni eseguono questi due primi passaggi

1. FETCH 🡪
   1. Il PC punta all’indirizzo di memoria che contiene l’istruzione da eseguire e la preleva
   2. Scrive dentro l’IR il valore dell’istruzione prelevata dalla memoria
   3. Incrementa il suo valore di 4, mediante operazione con l’ALU e riscrive il puntatore a istruzione successiva dentro sé
2. DECODE (Calcolo del target address) 🡪
   1. L’ALU somma l’indirizzo base del PC al campo immediato/offset dell’istruzione (anche in caso di operazioni diverse da type-I) esteso di segno e shiftato di 2 verso sinistra
   2. Vengono letti i registri (IR[25:21] e IR[20:16]) e memorizzati dentro A e B
   3. Viene memorizzato il target address in ALUOut (anche se risultasse inutile utilizzarlo dopo)
3. EXECUTE
   1. R-TYPE
      1. 1° CICLO
         1. ALUOut <= A op B (sum, sub, and, or, slt)
      2. 2° CICLO
         1. IR[15:11] <= ALUOut
   2. JUMP
      1. PC <= IR[25:0] << 2
   3. BEQ
      1. ALUOut <= A - B
      2. If (Zero==true) PC <= valore del target address calcolato nel DECODE
   4. SW
      1. ALUOut <= A + IR[15:0] SOLO esteso di segno
      2. Mem[ALUOut] <= B
   5. LW
      1. ALUOut <= A + IR[15:0] SOLO esteso di segno
      2. MDR <= Mem[ALUOut]
      3. RegFile[IR[20:16]] <= MDR

Eccezioni

* ECCEZIONI 🡪 Sincrone all’esecuzione del programma (bad address, overflow)
* INTERRUPT 🡪 Asincrone
* TRAPS (SysCalls) 🡪 Programmatore

Il Mips32 prevede

* Registro Cause
* Gestore eccezioni 🡪 Software
* Registro EPC (Exception Program Counter) 🡪 Contiene l’istruzione da riprendere

Quando avviene un’eccezione:

1. Save
2. Cause identify
3. Salto al codice corrispondente la causa
4. Restore dei registri
5. Eret (torno all’istruzione scritta nell’EPC)

**NB**: Se quando sono nel gestore avviene un’altra eccezione non posso tornare all’indirizzo della seconda eccezione

* Metodo di **Vettorizzazione (gestione a livello Hardware)**

Causa + Indirizzo Base Vettore Eccezioni = Indirizzo del codice relativo all’eccezione

Mem[Indirizzo del codice relativo all’eccezione] 🡪 PC

## Input/Output

Periferiche Mappate in Memoria

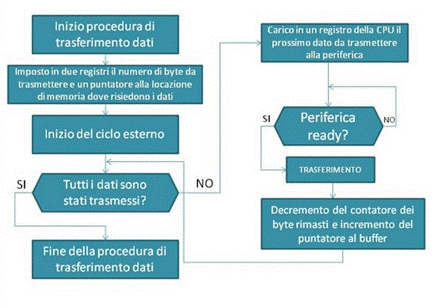
Ogni periferica viene riconosciuta dalla CPU come una locazione di memoria (lo spazio di indirizzamento della CPU comprende anche l’indirizzamento alle periferiche tramite BUS)

Ogni periferica possiede un’interfaccia di comunicazione con la CPU costituita da:

* Registro di stato
* Registro dati 🡪 I/O a seconda della periferica

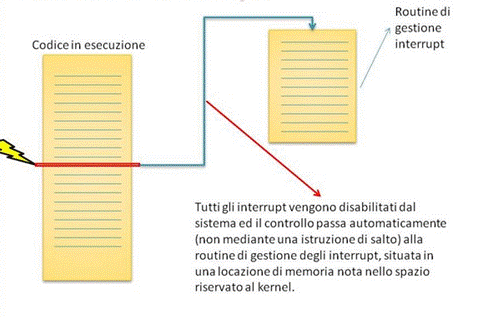
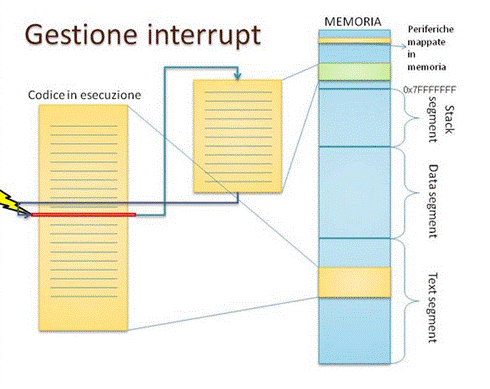
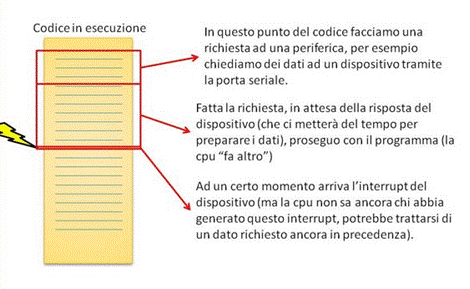
1. Controllo di Programma

* Tramite flusso di programma viene prelevato il bit ready dalla periferica mappata in memoria, confrontato nella CPU finché non risulta 1 (Busy Waiting)
* Quando il bit risulterà 1 viene eseguita una lw/sw sul registro dati della periferica
* LATENZA MINIMA 🡪 La CPU nota molto in fretta che la periferica è pronta (il tempo peggiore è il tempo di esecuzione di un ciclo di busy wait)
* ALTA BANDA PASSANTE 🡪 La CPU trasferisce subito il dato, di conseguenza la quantità di dati sarà molta in relazione al tempo

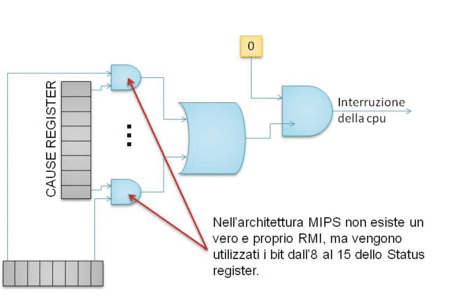


1. Gestione ad interrupt

* Linea di richiesta
* Quando una periferica genera un interrupt, la CPU deve eseguire una serie di istruzioni prestabilite, contenute a partire da una locazione di memoria nota a priori
  + SAVE
  + IDENTIFICAZIONE DELLA PERIFERICA INTERROMPENTE
  + GESTIONE DELLA PERIFERICA (TRASFERIMENTO DATO)
  + RESTORE
  + ERET (Ritorno dall’interrupt)



* Per identificare la periferica che ha generato l’interruzione, visto che la linea di richiesta è unica esistono delle strategie:
  + Si controllano le periferiche una ad una, capendo chi ha impostato il bit ready a 1
  + Vettorizzazione:
    - Base Vettore Interruzioni (Indirizzo di memoria noto) + Codice Periferica (ricevuto dalla linea di richiesta)
* Per non essere interrotti nuovamente durante la gestione di un interrupt esiste un meccanismo chiamato Mascheramento Globale (ogni bit mascheramento viene messo in and con i bit del cause reg)



* MINOR BANDA PASSANTE 🡪 Più istruzioni per identificare l’interrompente
* MAGGIORE LATENZA 🡪 La CPU si accorge dopo più tempo che la periferica ha mandato un segnale di interrupt

1. DMA (Direct Memory Access)

* Due registri aggiuntivi
  + L’indirizzo di memoria dal/al quale trasferire i dati
  + La quantità di dati da trasferire
* Fasi
  + Predisposizione registri
    - Stato
    - Dati
    - Puntatore alla memoria
    - Quantità dati da trasferire
  + Attivazione
    - La CPU mette a 1 un bit del registro stato della periferica
    - Da questo momento in poi la CPU può fare altro
  + Trasferimento elementare
    - Appena la periferica è pronta, accede alla memoria e trasferisce il contenuto del registro dati
    - Aggiorna il puntatore alla memoria e il contatore
    - Appena il contatore arriva a 0 viene generato un interrupt per notificare la CPU del trasferimento finito
* MASSIMA BANDA PASSANTE
* MINIMA LATENZA

## Gerarchie di memoria

Introduzione

Un programma, in un certo istante di tempo, accede soltanto a una porzione relativamente piccola del suo soltanto a una porzione relativamente piccola del suo spazio di indirizzamento

Due tipi di località:

* Temporale 🡪 Quando si fa riferimento ad un oggetto c’è la tendenza a fare riferimento allo stesso dopo poco tempo
* Spaziale 🡪 Quando si fa riferimento ad un elemento c’è la tendenza a fare riferimento ad elementi vicino allo stesso

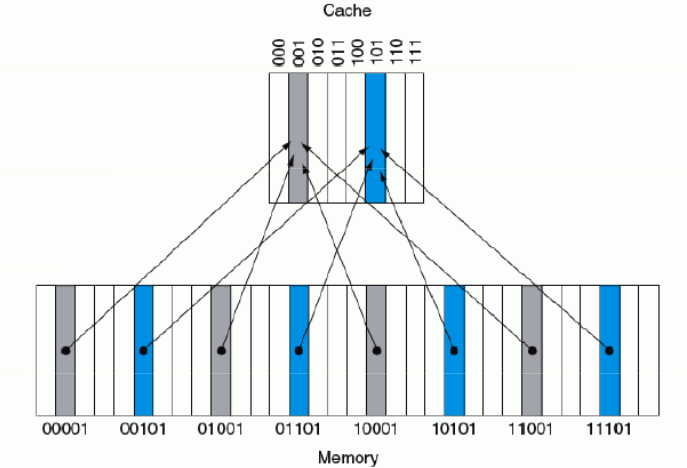
Definizioni

* Blocco 🡪 La più piccola quantità di informazione che può essere presente/assente in una gerarchia di memoria
* Hit 🡪 L’informazione richiesta dal processore si trova ai livelli più alti di gerarchia di memoria
* Miss 🡪 L’informazione richiesta dal processore NON si trova ai livelli più alti di gerarchia di memoria

Campi della cache

* Index
* Tag
* Valid Bit

Direct Mapped

A ciascun blocco della memoria corrisponde una specifica locazione nella cache

* Dimensione campo TAG 🡪 32 – (2 + m + r)
* Dimensione totale Cache 🡪 2^n\*(2^m\*32 + (32 – (n + m + 2)) + 1)

n = 2^n 🡪 Blocchi della cache

m = 2^m 🡪Word per ogni blocco della cache

*Da quanti bit è costituita una cache a mappatura diretta contenente 16KB di dati e avente blocchi da 4 parole, ipotizzando un indirizzo di memoria di 32 bit?*

* Capienza Cache => 16 KB => 16000 byte => 16000 / 4 => 4000 words => ca. 2^12 words totali
* Numero Blocchi => 4 words/blocco => 4000 words / 4 => 1000 words => 2^10 blocchi 🡪 m = 10
* Word Blocco => 4 words/blocco => 2^2 words/blocco 🡪 n = 2
* Bit/blocco => 4 words \* 32 bit => 128 bit/blocco
* CAPIENZA CACHE = 2^n \* (2^m \* 32 + (32 – (n + m + 2) + 1)) 🡪 2^10 \* (2^2 \* 32 + (32 – (2 + 10 + 2) + 1) = 2^10 \* 147 = 147 Kbit

Osservazioni

* Una cache con blocchi di dimensioni maggiori sfrutta maggiormente la località spaziale diminuendo la frequenza di miss
* **La frequenza di miss torna a salire se la dimensione dei blocchi diventa troppo grande rispetto alla dimensione della cache 🡪 Minor numero di blocchi, maggior competizione per occuparli 🡪 La cache si ritroverà ad essere sovrascritta più frequentemente, causando quindi miss più frequenti 🡪 Cresce inoltre il costo di una miss**