

CPUSim

Laboratorio 13/1/2015.

Nicolò Navarin

e-mail: nnavarin@math.unipd.it



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Scaricare il simulatore:

- `www.math.unipd.it/~sperduti/architettura1.html`
- Scorrere fino a sezione *Simulatori*, aprire la pagina *CPUSim*
- Scorrere fino a sezione *Downloads* e salvare (click destro, Save link as) *download a 5 MB zip file containing version 3.9.0*
- Estrarre l'archivio (ricordatevi dove!)

Avviare il simulatore:

- spostarsi nella directory del simulatore (e.g. `cd Architettura/CPUSim3.9.0`)
- aprire il file di istruzioni (`less InstallationInstructions.txt`)
- eseguire il comando specificato (`java -cp CPUSim3.9.jar:jhall.jar:CPUSimHelp3.9.jar cpusim.Main`)

Aprire la CPU di esempio Wombat1:

- `File` → `Open Machine` → `SampleAssignments/Wombat1.cpu`
- finestra `Registers` mostra i registri della CPU in simulazione e il loro contenuto
- finestra `RAM` mostra le celle di memoria ed il loro contenuto

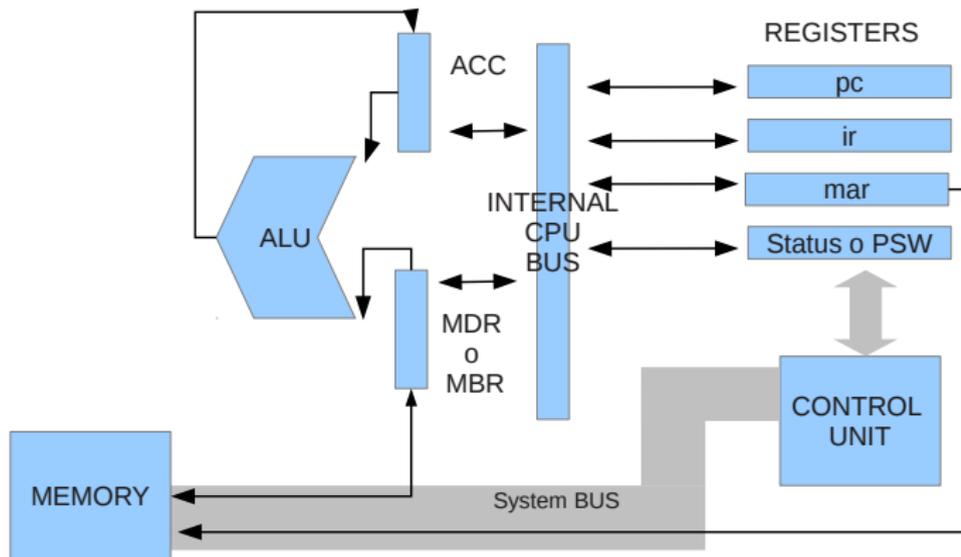
CPUSim permette di definire una CPU da simulare, composta da:

- Registri
 - “spazio di lavoro” della CPU
 - livello più alto della gerarchia di memoria.
- Istruzioni macchina
 - definiscono l'architettura
 - un programma scritto in linguaggio macchina può essere portato tra CPU diverse (ma che condividono lo stesso set di istruzioni).
- Microistruzioni
 - Le unità di base usate per definire le istruzioni
 - implementate a livello hardware (organizzazione)
 - possono essere diverse in ogni modello di CPU.
- Istruzioni in linguaggio ASSEMBLY
 - molto vicino al linguaggio macchina
 - codici mnemonici invece che codice binario per indirizzi di dati e istruzioni.

CPU semplice: 2 registri dati (utilizzabili dal “programmatore”)

Registri (*Modify* → *Hardware Modules*):

- **PC** (program counter) l'indirizzo della locazione di memoria contenente la successiva istruzione da eseguire
- **ACC** (accumulator) contiene i risultati della ALU
- **IR** (instruction register) contiene l'istruzione da eseguire, quella cioè puntata dal PC
- **MAR** (memory address register) contiene l'indirizzo della locazione di memoria che viene acceduta
- **MDR** (memory data register) contiene temporaneamente tutti i dati e le istruzioni che dalla memoria devono essere elaborati nel processore
- **Status** (registro di stato) memorizza una serie di bit indicativi dello stato corrente del processore (halt, overflow, underflow, ecc)

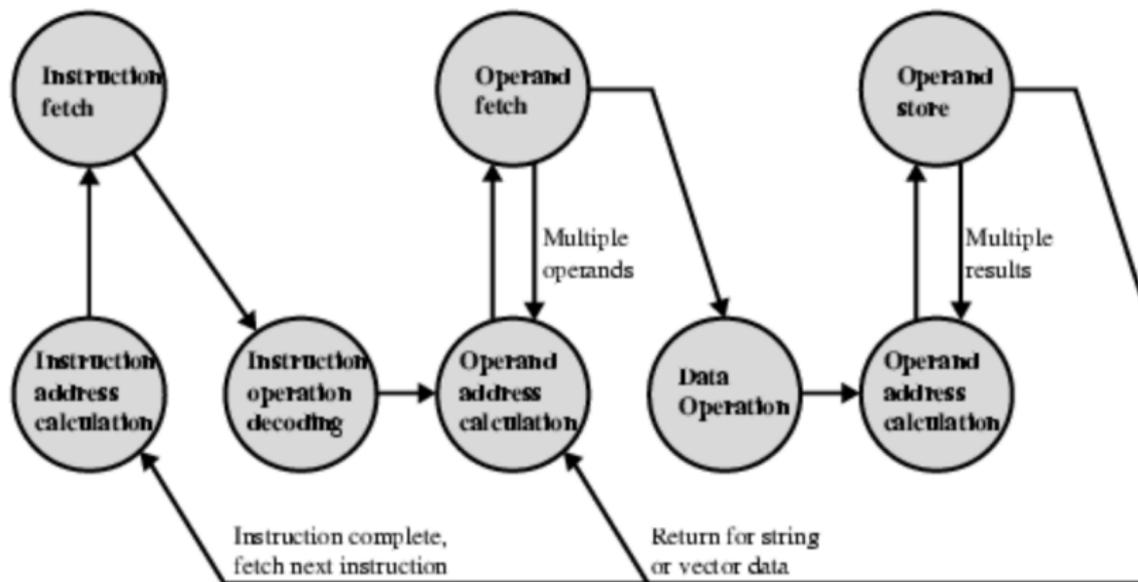


Microistruzioni:

- trasferimento dati tra registri
- trasferimento da/a memoria centrale
- operazioni aritmetico-logiche
- visualizzabili dal menu *Modify* → *Microinstructions* (e.g. Arithmetic, TransferRtoR)
- proviamo a definire una microistruzione che esegue l'AND bit a bit tra ACC e MDR, di tipo *Logical* che si chiamerà *accANDmdr- >acc?*

Istruzioni: ognuna è composta da una determinata sequenza di microistruzioni.

Wombat1 - Ciclo di esecuzione



Esecuzioni ripetute di cicli macchina. Ogni ciclo è composto da:

- Fetch sequence:
 - una sequenza di microistruzioni che carica la prossima istruzione da eseguire nell'IR e la decodifica
 - la sequenza è la stessa per ogni ciclo di esecuzione
 - visualizzabile dal menu Modify → Fetch sequence
- Execute sequence:
 - la sequenza di microistruzioni associate all'istruzione appena decodificata
 - varia da ciclo a ciclo

L'esecuzione termina quando viene settato a 1 un bit di *halt*.

Istruzioni:

- Dimensione fissata a 16 bit
- 4 bit per il codice operatore
- 12 bit per l'indirizzo (Quindi quanta RAM al massimo?)
- visualizzabili dal menu Modify → Machine instructions
- il pulsante *Edit fields* permette di modificare i campi dato.

Istruzioni ASSEMBLY:

- **[etichetta:] operatore operandi [;commento]**
e.g. `ADD x ; Mem[x]+acc → acc`
- **etichetta: .data nByte valore [;commento]**
 - Pseudo-istruzioni, per definire i dati
 - e.g. `X: .data 2 0 ; x è una locazione di memoria da 2 byte
inizializzata a 0`
- e.g. load/store, jump, add ...
- proviamo a definire un'istruzione che esegue l'and bit a bit tra ACC e una cella di memoria?

- Input/output:
 - READ: legge un intero da input e lo mette in ACC
 - WRITE: scrive in output il contenuto di ACC
- Aritmetiche:
 - ADD X: somma il contenuto del registro ACC al contenuto della cella di memoria X e mette il risultato in ACC.
 - SUBTRACT X, MULTIPLY X, DIVIDE X (divisione intera)
- Trasferimento (da M a registri e viceversa):
 - STORE X: da registro ACC alla cella di memoria X
 - LOAD X: dalla cella di memoria X al registro ACC
- Salti:
 - JUMP X: salta all'istruzione con etichetta X
 - JMPN X: salta all'istruzione X se $ACC < 0$
 - JMPZ X: salta all'istruzione X se $ACC = 0$
- Stop:
 - STOP: segnala la fine del programma

Il primo programma in ASSEMBLY:

- File → Open text W1-0.a
- Start, Done e sum sono etichette
- le istruzioni sono evidenziate in blu
- solo alcune istruzioni hanno argomenti

Il programma deve essere:

- **Assemblato**: tradotto da linguaggio ASSEMBLY a linguaggio macchina (Execute → Assemble). Verrà effettuato un controllo di correttezza sintattica.
- **Caricato** in memoria per essere eseguito (Execute → Assemble&Load)
- **Eseguito** (Execute → Run)
 - il programma inizia l'esecuzione con l'istruzione il cui indirizzo si trova nel PC, inizialmente 0.
 - La macchina ripete cicli di esecuzione Fetch/Execute

- Reset: *Execute* → *Reset Everything*
- Ricaricare il programma: *Execute* → *Assemble&Load*
- Entrare in Debug mode: *Execute* → *Debug mode*.

In questa modalità:

- l'avanzamento dell'esecuzione è lasciato all'utente
- si può procedere una istruzione o microistruzione alla volta
- è possibile modificare a mano il contenuto di registri e RAM
- è possibile impostare breakpoint dalla finestra RAM

Scrivere un programma ASSEMBLY per la CPU Wombat1 che legge due numeri (usando la locazione di memoria x) e salva la somma nella locazione di memoria z

- aprire un nuovo file ASSEMBLY con *File* → *New text*
- salvare con *File* → *Save text as* (e.g. lab3_es1.a)

Listing 1: Esercizio 1

```
read ; primo intero -> acc
store x ; primo intero -> cella x
read ; secondo intero -> acc
add x ; M[x] + acc -> acc
store z ; acc -> cella z
write ; output acc
stop ; termina esecuzione
x: .data 2 0 ; 2 byte dove mettere x
z: .data 2 0 ; 2 byte dove mettere z
```

Scrivere un programma ASSEMBLY per la CPU Wombat1 che legge un intero in ingresso e ritorna 0 se l'intero è pari, -1 altrimenti (usare divisione intera e moltiplicazione).

Listing 2: Esercizio 2

```
read          ; n → acc
store x       ; acc → x
divide due    ; acc/2 → acc
multiply due  ; acc*2 → acc
subtract x    ; acc - x → acc
write        ; visualizza acc
stop         ; termina il programma

x: .data 2 0   ; variabile x
due: .data 2 2 ; variabile due
```

Scrivere un programma ASSEMBLY per la CPU Wombat1 che calcola e stampa il valore assoluto di un intero ricevuto in input.

Listing 3: Esercizio 3

```
read                ;input -> acc
jmpn negativo      ;se acc <0, salta a negativo
fine: write        ;acc -> output
stop               ;stop
negativo: store copia ;acc -> copia
load zero         ;zero -> acc
subtract copia    ;acc-copia -> acc
jump fine        ;va alla fine
zero: .data 2 0   ;2 byte dove mettere zero
copia: .data 2 0  ;2 byte dove mettere copia
```

Scrivere un programma ASSEMBLY per la CPU Wombat1 che calcola il prodotto di due interi usando somme.