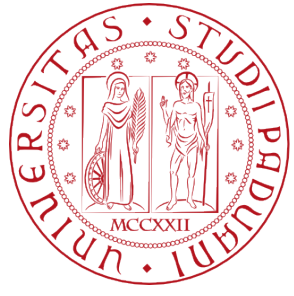
CPUSim

Laboratorio 30/11/2016

Tommaso Padoan

e-mail: padoan@math.unipd.it



Università
degli Studi
di Padova

CPUsim

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!).

CPUsim

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* $\rightarrow$ *Open Machine* $\rightarrow$ *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

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

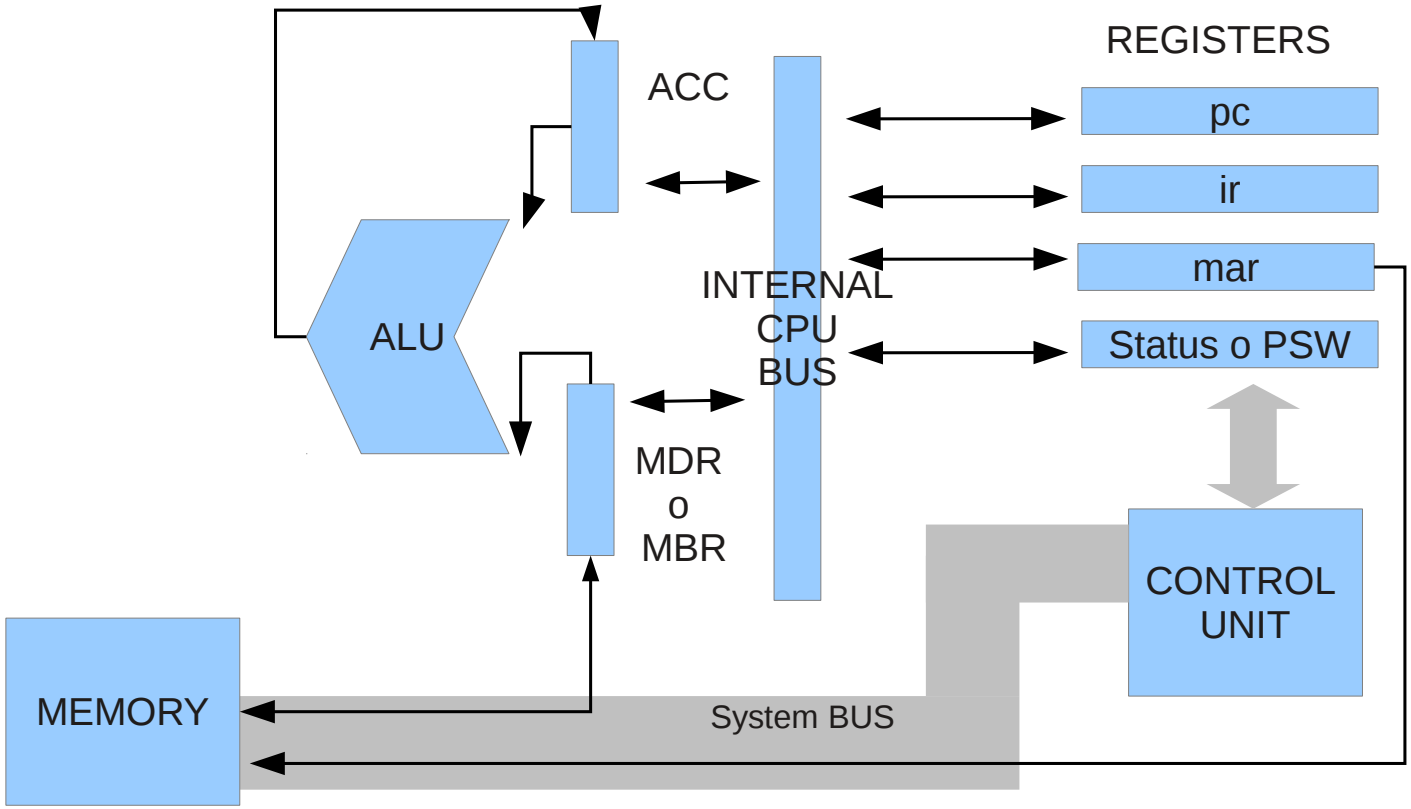
- Registri
  - "spazio di lavoro" della CPU
  - livello più alto della gerarchia di memoria.
- Istruzioni macchina
  - definiscono l'architettura
  - un programma scritto in linguaggio macchina può essere portato tra CPU diverse (ma che condividono lo stesso set di istruzioni).
- Microistruzioni
  - 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.

Wombat1 - Registri

CPU semplice: 2 registri dati (utilizzabili dal "programmatore")
Registri (*Modify* $\rightarrow$ *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.).

Wombat1 - Architettura



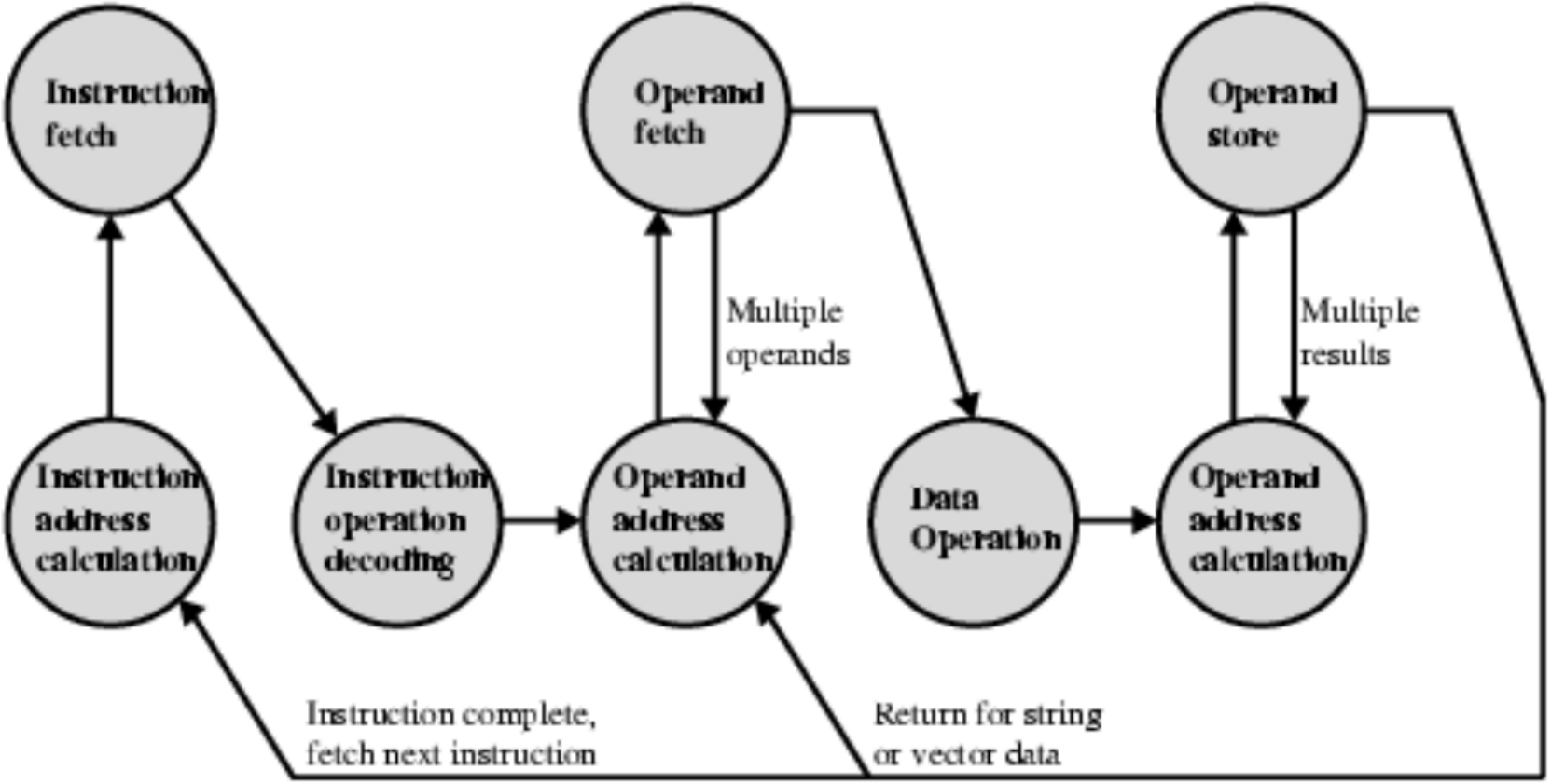
Wombat1 - Microistruzioni

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 un nuova 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



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* $\rightarrow$ *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*.

Wombat1 - Istruzioni

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?

Wombat1 - Instruction set

- 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 al contenuto del registro ACC il 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.

Esecuzione dei programmi in CPUSim

Il primo programma in ASSEMBLY:

- *File* → *Open text* → *SampleAssignments/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.

Controllare l'esecuzione: Debug mode

Prima fare:

- 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 breakpoints dalla finestra RAM.

Esercizio 1

Scrivere un programma ASSEMBLY per la CPU Wombat1 che legge un numero da input e scrive il suo successore $(n + 1)$ su output.

- Aprire un nuovo file ASSEMBLY con *File* $\rightarrow$ *New text*.
- Salvare con *File* $\rightarrow$ *Save text as* (e.g. Es1.a).

Esercizio 1 - Soluzione

Esercizio 1

```
read     ; input numero intero -> acc
add uno  ; acc + M[uno] -> acc
write    ; output acc
stop     ; termina esecuzione

uno: .data 2 1 ; il valore 1
```

Esercizio 2

Scrivere un programma ASSEMBLY per la CPU Wombat1 che legge due numeri (usando la locazione di memoria x) e salva la loro somma nella locazione di memoria y e la scrive su output.

Esercizio 2 - Soluzione

Esercizio 2

```
read     ; input primo intero -> acc
store x  ; acc -> cella x
read     ; input secondo intero -> acc
add x    ; acc + M[x] -> acc
store y  ; acc -> cella y
write    ; output acc
stop     ; termina esecuzione

x: .data 2 0 ; 2 byte dove mettere x
y: .data 2 0 ; 2 byte dove mettere y
```

Esercizio 3

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

Esercizio 3 - Soluzione

Esercizio 3

```
        read            ; input -> acc
        jmpn neg        ; se acc<0, salta a neg
fine:   write           ; output acc
        stop            ; termina esecuzione
neg:    multiply -uno   ; acc * M[-uno] -> acc
        jump fine       ; salta a fine

-uno:   .data 2 -1 ; il valore -1
```

Esercizio 4

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

Esercizio 4 - Soluzione

Esercizio 4

```
        read            ; input primo fattore -> acc
        store x         ; acc -> cella x
        read            ; input secondo fattore -> acc
ciclo:  jmpz fine       ; se acc=0, salta a fine
        store y         ; acc -> cella y
        load sum        ; M[sum] -> acc
        add x           ; acc + M[x] -> acc
        store sum       ; acc -> cella sum
        load y          ; M[y] -> acc
        subtract uno    ; acc - M[uno] -> acc
        jump ciclo      ; salta a ciclo
fine:   load sum        ; M[sum] -> acc
        write           ; output acc
        stop            ; termina esecuzione
x:    .data 2 0 ; primo fattore
y:    .data 2 0 ; secondo fattore
sum:  .data 2 0 ; somma parziale
uno:  .data 2 1 ; il valore 1
```