Università
degli Studi
di Padova

DIPARTIMENTO
MATEMATICA

CPUSim

Laboratorio di Architettura degli Elaboratori

Corso di Laurea in Informatica A.A. 2020/2021

Prof. Nicolò Navarin
Dott. Davide Rigoni

30 Novembre 2020

CPUSim

- Scaricare il simulatore dal moodle del corso, oppure

  `http://www.cs.colby.edu/djskrien/CPUSim/CPUSim4.0.11.zip`
- Estrarre l'archivio (ricordatevi dove)
- Fare doppio click su CPUSim-4.0.11.jar
- Se non funziona, da console:

```
cd /vostro_path/CPUSim
```


```
java -cp .:richtextfx-0.6.10.jar:
    reactfx-2.0-M4.jar -jar CPUSim
    -4.0.11.jar
```

```
java -cp .;richtextfx-0.6.10.jar;
    reactfx-2.0-M4.jar -jar CPUSim
    -4.0.11.jar
```

- Aprire la CPU di esempio **Wombat1**:

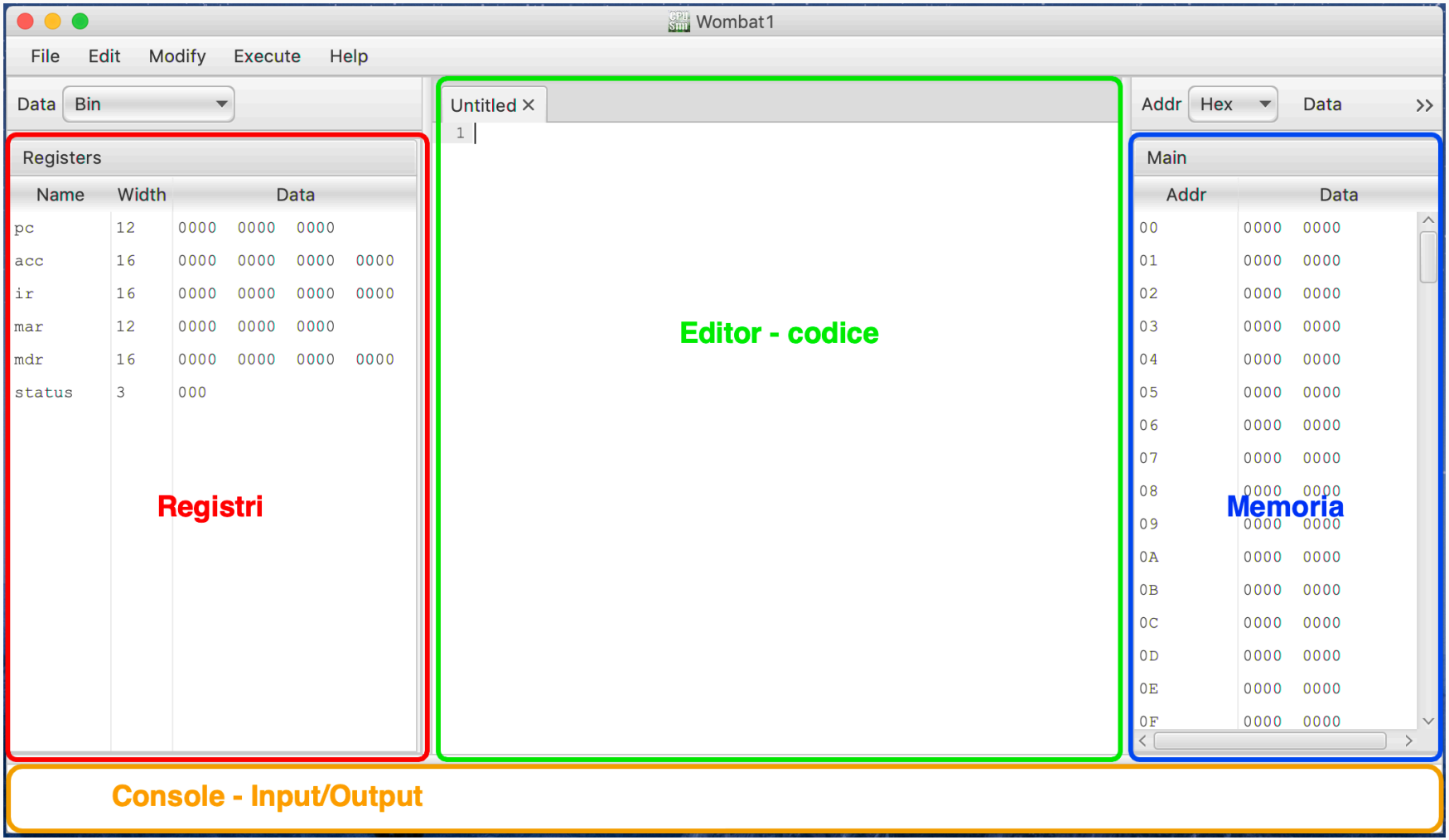
  File → *Open Machine* → *SampleAssignments/Wombat1.cpu*

CPUSim - overview

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

- Registri: "spazio di lavoro" della CPU, e 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 devono condividere lo stesso set di istruzioni. Le istruzioni sono in linguaggio **ASSEMBLY**;
- Microistruzioni: le unità di base usate per definire le istruzioni, le quali sono implementate a livello hardware e possono essere diverse in ogni modello di CPU.

CPUSim - finestra principale

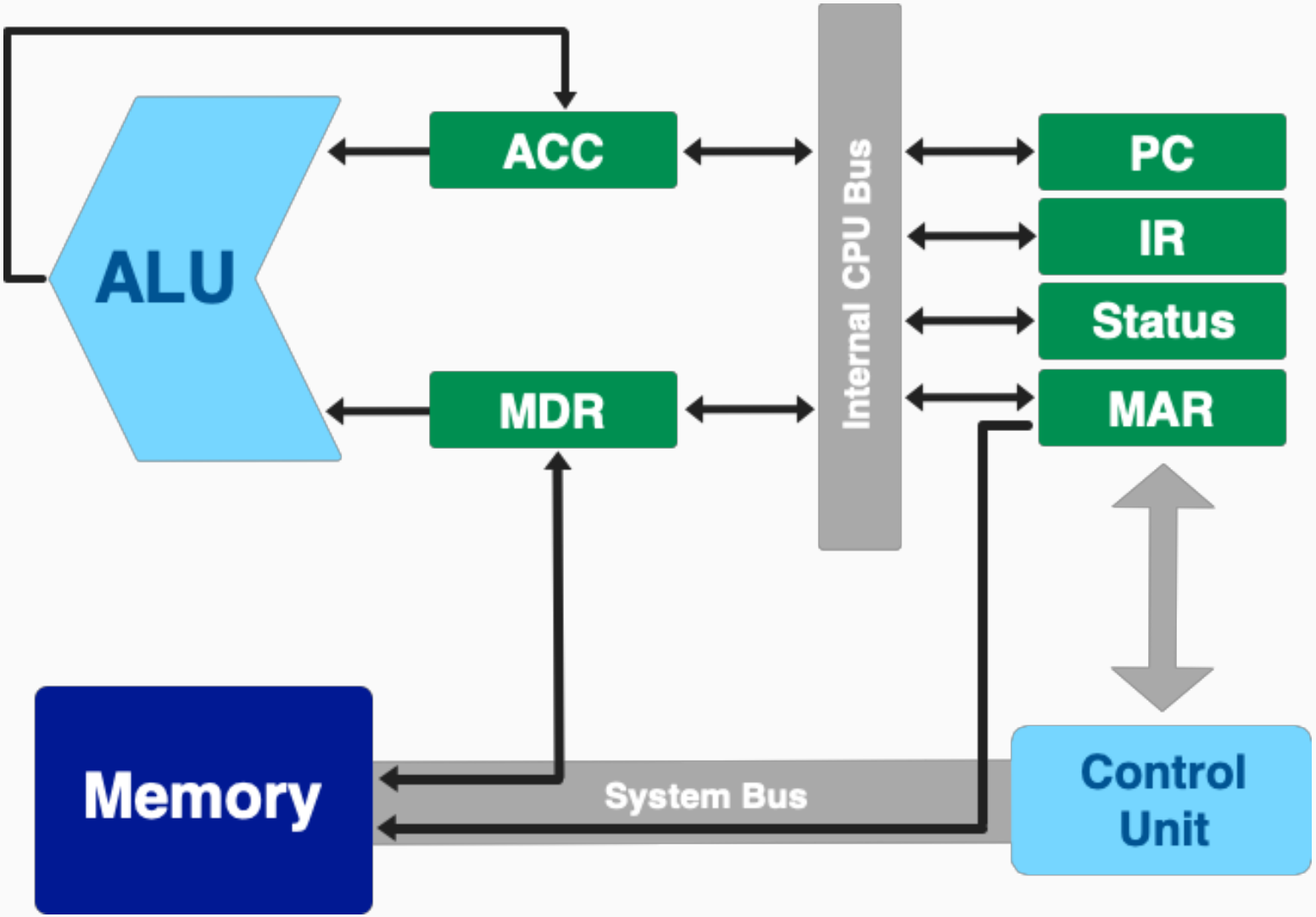


Wombat1 - registri

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 (in precedenza) 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: visualizzabili dal menu *Modify* $\rightarrow$ *Microinstructions* (e.g. Arithmetic, TransferRtoR), permettono di

- Trasferire dati tra registri;
- Trasferire da/a memoria centrale;
- Eseguire operazioni aritmetico-logiche.

Istruzioni: composte da una determinata sequenza di microistruzioni.

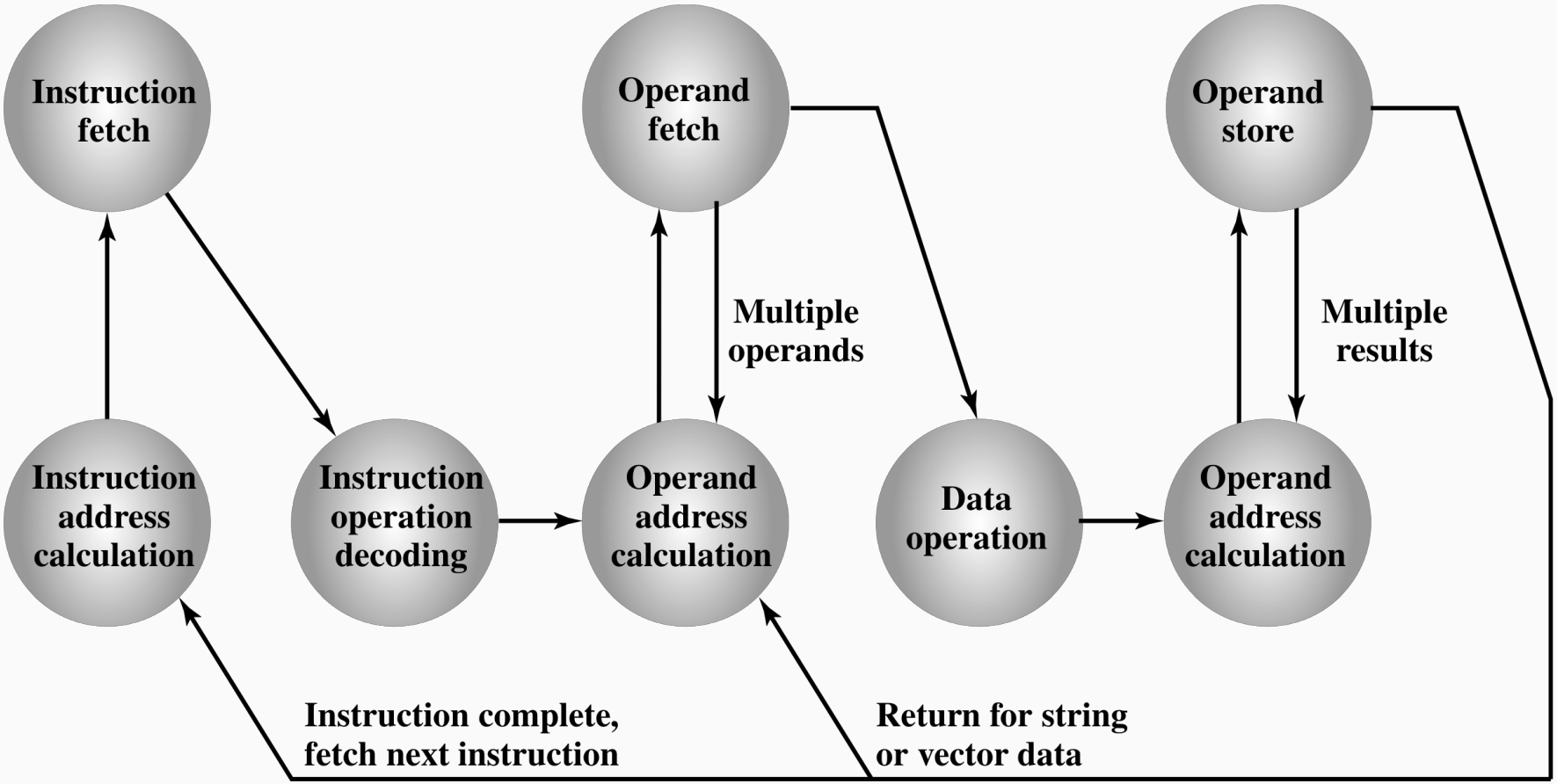
Wombat1 - microistruzione `accANDmdr->acc`

Definire una microistruzione che esegue l'and tra ACC e MDR e mette il risultato in ACC:

- *Modify*$\rightarrow$ *Microinstructions*;
- Selezionare il tipo **Logical**;
- Premere **New** e dare un nome alla nuova istruzione (e.g., `acc&mdr->acc`);
- Definire la microistruzione:

type	AND
source1	ACC
source2	MDR
destination	ACC

Wombat1 - ciclo di esecuzione (1/2)



Wombat1 - ciclo di esecuzione (2/2)

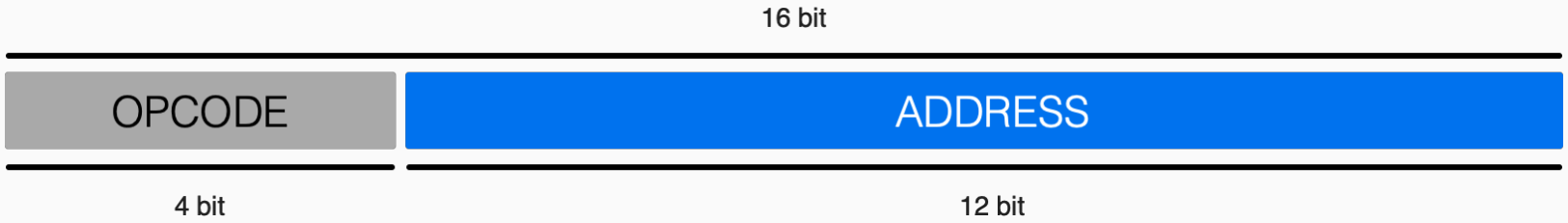
Sequenza di Fetch (*Modify* $\rightarrow$ *Fetch sequence*)

```
PC -> MAR
Main[MAR] -> MDR
MDR -> IR
inc2-PC          PC += 2 byte
decode-IR        decodifica l'istruzione
```

Sequenza di Execute

- La sequenza di microistruzioni associate all'istruzione appena decodificata;
- L'esecuzione termina quando viene posto a 1 il bit di `halt`.

Wombat1 - istruzioni



- Dimensione fissata a 16 bit;
- 4 bit per il codice operatore (OPCODE);
- 12 bit per l'indirizzo (quanta RAM al massimo?);
- Visualizzabili dal menu *Modify* $\rightarrow$ *Machine Instructions*. Il pulsante *Edit fields* permette di modificare i campi dato.

Wombat1 - istruzioni ASSEMBLY

- Formato di un'istruzione con operandi:

```
[etichetta:] operatore operandi [;commento]
```

e.g.,

```
ADD x ; Mem[x] + acc -> acc
```

- Esistono anche istruzioni senza operandi

  e.g., `write`, `read`, `stop`

- Pseudo-istruzione per definire i dati:

```
etichetta: .data nByte valore [;commento]
```

e.g.,

```
x: .data 2 0
```

`x` è una locazione di memoria da 2 byte inizializzata a 0.

Wombat1 - creazione di un'istruzione

Definiamo un'istruzione che esegue l'**and bit a bit** tra ACC e una cella di memoria.

- *Modify* $\rightarrow$ *Machine instructions*;
- Premere **New** e dare un nome alla nuova istruzione (e.g., `and`)
- Definire il formato [OPCODE|ADDRESS] con il Drag&Drop;
- Tab *Implementation*, sempre con Drag&Drop scrivere il seguente programma

```
ir(4-15)->mar
Mem[mar]->mdr
acc&mdr->acc
End
```

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.

Wombat1 - eseguire un programma

Carichiamo il nostro primo programma in ASSEMBLY:

File $\rightarrow$ *Open text* $\rightarrow$ *SampleAssignments/W1-0.a*

Il programma deve essere:

- Assemblato: tradotto da linguaggio ASSEMBLY a linguaggio macchina (*Execute* $\rightarrow$ *Assemble*). Verrà effettuato un controllo di correttezza sintattica.
- Caricato in memoria per essere eseguito (*Execute* $\rightarrow$ *Assemble & load*).
- Eseguito (*Execute* $\rightarrow$ *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.

Wombat1 - debug mode

Prima fare:

- Reset: *Execute* $\rightarrow$ *Reset everything*;
- Ricaricare il programma: *Execute* $\rightarrow$ *Assemble & load*;
- Entrare in Debug mode: *Execute* $\rightarrow$ *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

```
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

```
read    ; input primo intero -> acc
store x ; acc -> M[x]
read    ; input secondo intero -> acc
add x   ; acc + M[x] -> acc
store y ; acc -> M[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

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

-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

```
        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
```