Università
degli Studi
di Padova

DIPARTIMENTO
MATEMATICA

# CPUSim

## Laboratorio di Architettura degli Elaboratori

Corso di Laurea in Informatica A.A. 2019/2020

---

Nicolo' Navarin
Davide Rigoni
nnavarin@math.unipd.it

14 dicembre 2020

# CPUSim

- Scaricare il simulatore dal moodle del corso (`CPUSim4.0.11.zip`);
- Estrarre l'archivio (ricordatevi dove);
- Avviare il simulatore:
  - doppio click;
  - da console, posizionandosi nella cartella del simulatore, eseguire il comando (**Nota**: tutto sulla stessa riga):
    - macOS/Linux

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

    - Windows

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

# Nel laboratorio di oggi

- Impareremo come definire/modificare una CPU nel simulatore;
- Definiremo dei registri ad uso generale;
- Vedremo come funziona l'indirizzamento a registro;
- Capiremo perché aumenta la complessità delle istruzioni.

# Prima di iniziare

- Accedere al moodle del corso;
- Scaricare *Wombat2*;
- Salvare il file nella cartella *SampleAssignments* di CPUSim (`CPUSim4.0.11/SampleAssignments`).

# Definizione di una nuova CPU

Limitazioni di **Wombat1**:

- Un solo registro dati (*accumulatore*);
- Scrivere programmi anche semplici è “macchinoso” e richiede molti accessi alla memoria.

# Definizione di una nuova CPU

Limitazioni di **Wombat1**:

- Un solo registro dati (*accumulatore*);
- Scrivere programmi anche semplici è "macchinoso" e richiede molti accessi alla memoria.

Possibile soluzione e benefici:

- **Più registri** dati $\Rightarrow$ meno accessi alla memoria;
- Programmi più **intuitivi**.

# Definizione di una nuova CPU

Limitazioni di **Wombat1**:

- Un solo registro dati (*accumulatore*);
- Scrivere programmi anche semplici è "macchinoso" e richiede molti accessi alla memoria.

Possibile soluzione e benefici:

- **Più registri** dati $\Rightarrow$ meno accessi alla memoria;
- Programmi più **intuitivi**.

**Creiamo** una nuova **CPU** partendo da Wombat1:

- Aprire la CPU di esempio Wombat1;
- *File* $\rightarrow$ *Save Machine As* $\rightarrow$ *Wombat2-test.cpu*.

Possiamo modificare le specifiche attraverso il menu *Modify*.

# Definizione di un array di registri generici

- *Modify* → *Hardware modules* → *RegisterArray*;
- `length=16` (numero di registri), `width=16` (registri dati);
- I registri nell'array sono **riferiti attraverso il loro indice** (registro 0, registro 1,. . . , registro 15);
- 16 registri: 4 bit per l'indirizzamento (indirizzamento registro).

Non serve più l'*accumulatore*!

Momentaneamente non rimuoviamo ACC perché ridefiniremo "tutte" le operazioni che lo utilizzano, ma ragioniamo come se non ci fosse.

# ALU - definizione di nuovi registri

Che registri userà la ALU ora che non esiste più ACC?

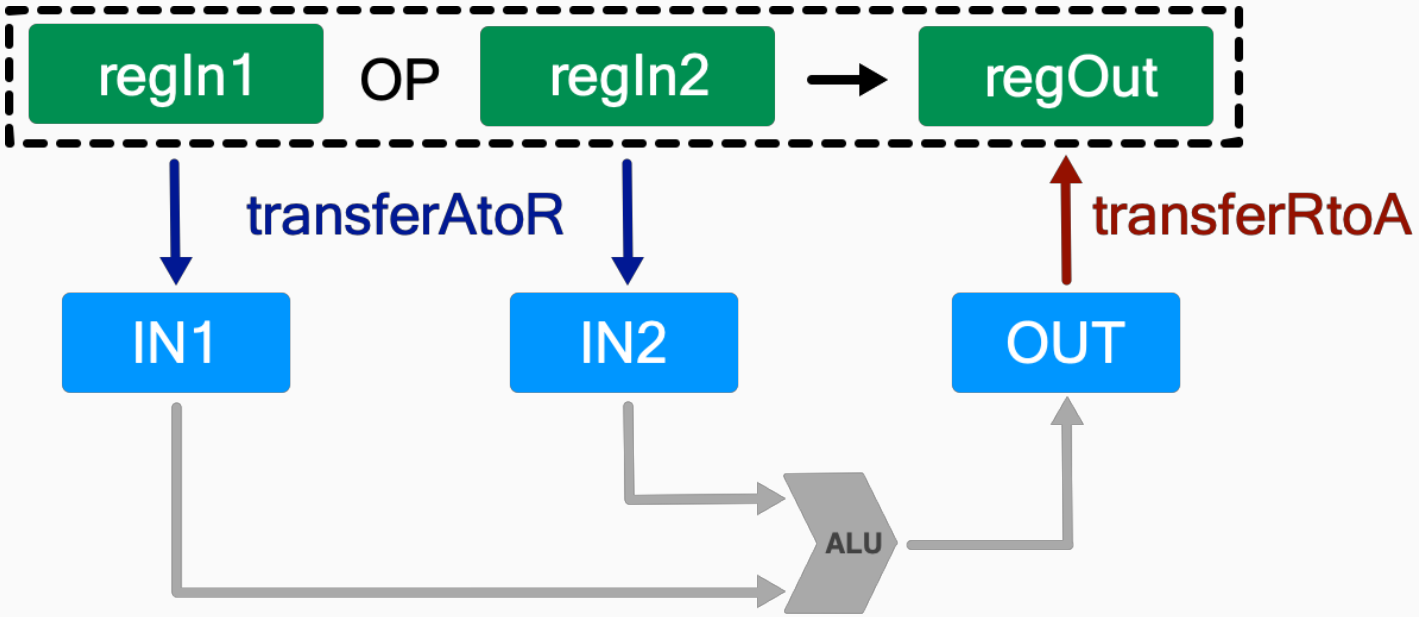
- Possibile soluzione: **registri dedicati**;
- Definiamo i registri IN1, IN2 e OUT, di input e output per la ALU (registri dati)
  - *Modify* $\rightarrow$ *Hardware modules* $\rightarrow$ *Register*;
  - Fissiamo la lunghezza dei registri a 16 bit.

# ALU - microistruzioni

- Ridefiniamo la microistruzione di addizione in modo che utilizzi i nuovi registri dedicati (IN1 + IN2 $\rightarrow$ OUT);

# ALU - microistruzioni

- Ridefiniamo la microistruzione di addizione in modo che utilizzi i nuovi registri dedicati (IN1 + IN2 $\rightarrow$ OUT);
- Definiamo le microistruzioni per il **trasferimento**:



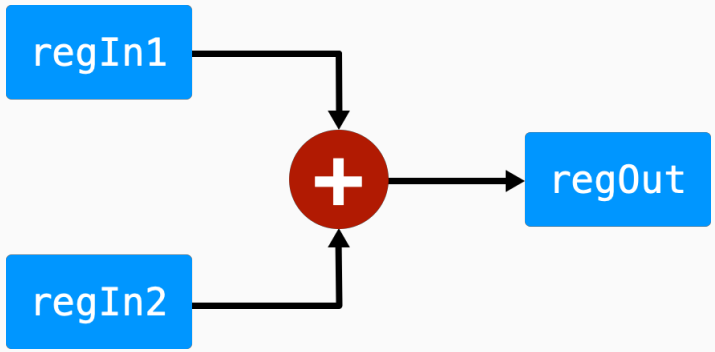

**Nota**: l'istruzione si trova in IR!

# Istruzioni

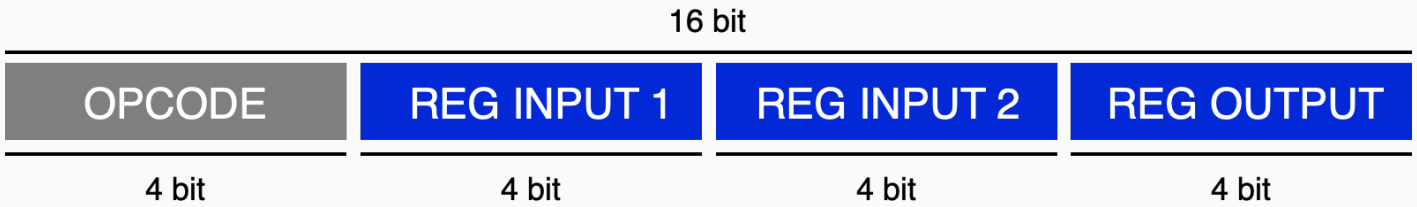
Come usare questi nuovi registri?

Definiamo una nuova operazione: **ADD tra registri** (dell'array)

```
addR regIn1 regIn2 regOut
```




Formato istruzione `addR` (serve un nuovo campo *registro* di 4 bit):



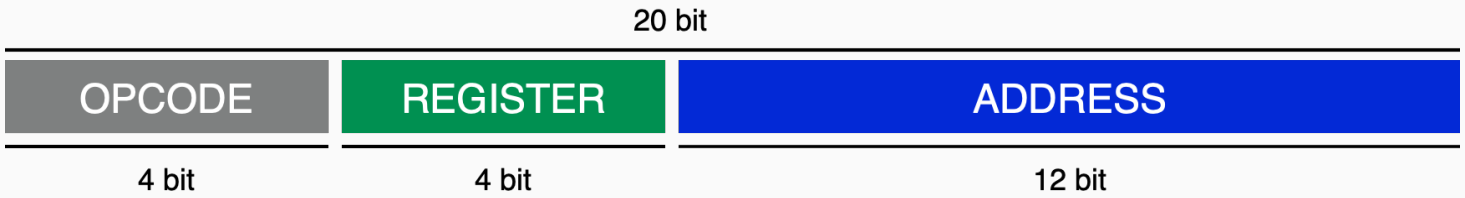
# Altre istruzioni - `load`

L'istruzione `addR` occupa 16 bit, come tutte le istruzioni di **Wombat1**.

Proviamo ad immaginare una possibile definizione di `loadR`:

- Semantica: carica il contenuto di una cella di memoria in un registro;
- **Quale registro?** $\Rightarrow$ serve un campo dati *register*;
- **Quale cella di memoria?** $\Rightarrow$ serve un campo dati *address*.

Esempio di possibile formato per l'istruzione `loadR`:




Ma le istruzioni in *Wombat1* hanno lunghezza fissata a 16 bit!

# Lunghezza istruzioni

Come fare per poter usare anche istruzioni più grandi?

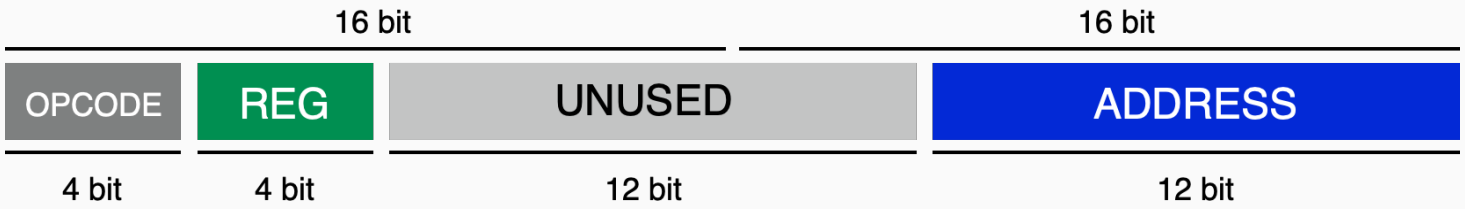
Possibili soluzioni:

- **Lunghezza variabile**:
  - ogni istruzione può richiedere un numero di bit diversi (a seconda degli operandi e di quanto è frequente)
  - maggiore code density, non ci sono bit inutilizzati
  - decodifica complessa
- **Formato ibrido o Mixed** - ogni istruzione ha una dimensione in un insieme limitato e predeterminato:
  - ARM: il processore può essere in modalità diverse, ognuna a lunghezza fissa
    - più decoder separati, ma semplici, e solo uno in funzione
  - Wombat2: la CPU fa il fetch dei primi 16 bit e decodifica l'istruzione (come in Wombat1);
    - nell'implementazione dell'istruzione stessa viene eseguito il fetch della parte mancante.

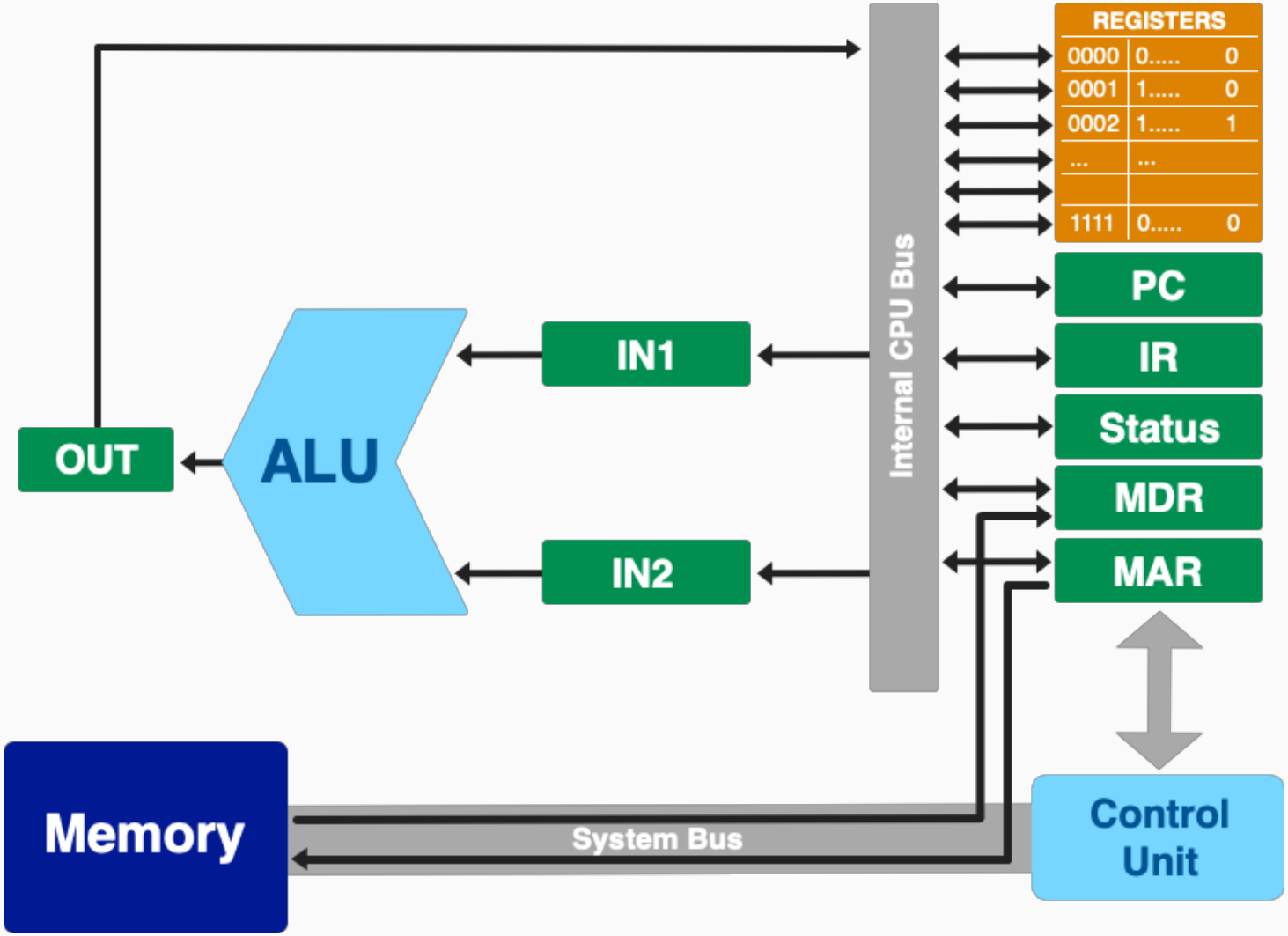
# Wombat 2

Ridefinire tutte le microistruzioni/istruzioni richiederebbe molto tempo!!

- Carichiamo la CPU **Wombat2.cpu**.
- Vediamo l'implementazione della **loadR**.



# Wombat2 - architettura



# Wombat2 - instruction set

| Istruzione | Significato | Descrizione |
|---|---|---|
| `readR R1` | input $\rightarrow$ R1 | Input from keyboard in R1 |
| `writeR R1` | R1 $\rightarrow$ output | Write value of R1 |
| `multiplyR R1 R2 R3` | R1 $\times$ R2 $\rightarrow$ R3 | Multiply contents of two registers |
| `divideR R1 R2 R3` | R1 / R2 $\rightarrow$ R3 | Divide contents of two registers |
| `subtractR R1 R2 R3` | R1 - R2 $\rightarrow$ R3 | Subtract contents of two registers |
| `addR R1 R2 R3` | R1 + R2 $\rightarrow$ R3 | Add contents of two registers |
| `loadR R1 addr` | Mem[addr] $\rightarrow$ R1 | Load word from memory in R1 |
| `storeR R1 addr` | R1 $\rightarrow$ Mem[addr] | Store word in memory from R1 |
| `jmpzR R1 addr` | If $R1 = 0$ jump to label | Conditional jump ($R1 = 0$) |
| `jmpnR R1 addr` | If $R1 < 0$ jump to label | Conditional jump ($R1 < 0$) |
| `jump addr` | jump to addr | |
| `stop` | stop execution | |

# Esercizio 1

Scrivere un programma ASSEMBLY per la CPU Wombat2 che calcola e stampa il valore assoluto di un intero ricevuto in input.

Ancora molto simile a Wombat1.

# Esercizio 2

Scrivere un programma ASSEMBLY per la CPU Wombat2 che calcola il prodotto di due interi ricevuti in input usando somme.

# Esercizio 3

Scrivere un programma ASSEMBLY per la CPU Wombat2 che calcola e restituisce il resto della divisione tra due interi ricevuti in input.

*Esempio: input 34 e 7, produce in output 6 (i.e., 34 mod 7 = 6)*

# Esercizio 4

Scrivere un programma ASSEMBLY per la CPU Wombat2 che esegue il fattoriale di un numero ricevuto in input.

# Esercizio 5

Scrivere un programma ASSEMBLY per la CPU Wombat2 che chiede di inserire da tastiera valori fintanto che non viene inserito un numero negativo. Infine stampa il massimo tra tutti i valori inseriti.