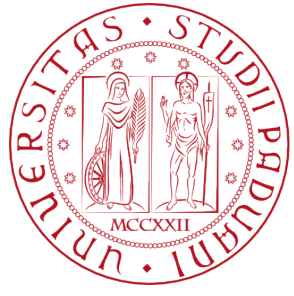
CPUSim-2
Laboratorio 16/1/2015.

Nicolò Navarin

e-mail: nnavarin@math.unipd.it



Università
degli Studi
di Padova

Intro

Nel laboratorio di oggi:

- impareremo come definire una nuova CPU nel simulatore
- definiremo dei registri ad uso generale
- vedremo come funziona l'indirizzamento a registro
  - capiremo perchè aumenta la complessità delle istruzioni
- vedremo l'indirizzamento indiretto

Prima di iniziare:

- `www.math.unipd.it/~sperduti/architettura1.html`
- scarichiamo *Wombat 2 (CPUSim)* e *Wombat 3 (CPUSim)* e salviamo i file nella cartella *SampleAssignments* di CPUSim.

Esercizio 4

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

Esercizio 4 - Soluzione

Listing 1: Esercizio 4

```
read                ; legge -> acc
store x             ; acc -> x
read                ; legge -> acc
store y             ; acc -> y
ciclo: jmpz fine    ;
load sum            ; sum -> acc
add x               ; acc + x -> acc
store sum           ; acc -> sum
load y              ;  y -> acc
subtract uno        ; acc - 1->acc
store y             ; acc -> y
jump ciclo
fine: load sum      ; somme parziali -> acc
write
stop
x: .data 2 0;
y: .data 2 0;
sum: .data 2 0;
uno: .data 2 1;
```

Nuova CPU

Limitazioni di Wombat1:

- un solo registro dati (*accumulatore*);
- scrivere programmi anche semplici è macchinoso e richiede molti accessi alla memoria;
- più registri dati $\rightarrow$ meno accessi alla memoria;
- programmi più intuitivi.

Creiamo una nuova cpu partendo da *Wombat*1:

- aprire la cpu di esempio *Wombat*1;
- *File*$\rightarrow$ *Save Machine As* Wombat2-test.cpu.

Possiamo modificare le specifiche attraverso il menu *Modify*.

Array di registri generici

Definiamo un array di registri.

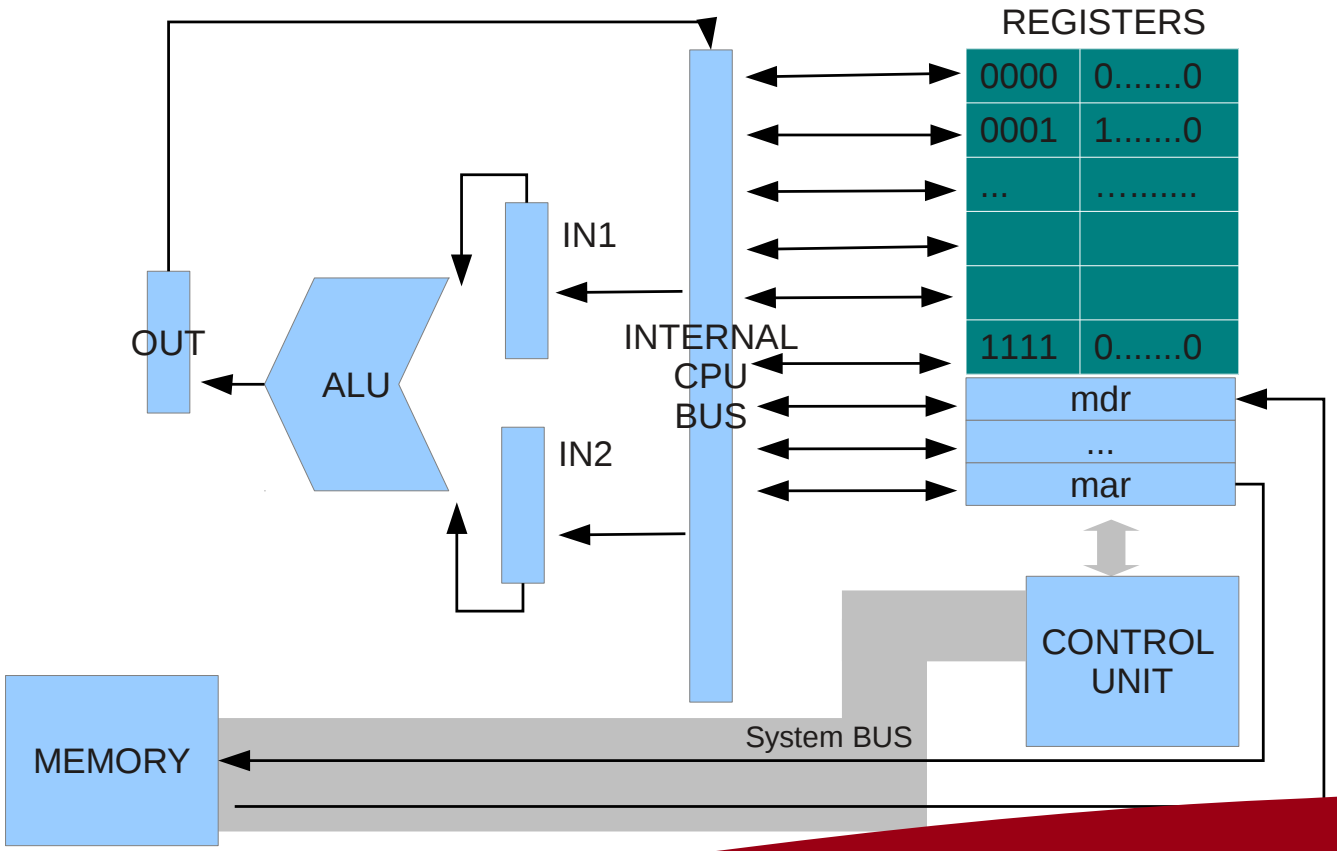
- *Modify* $\rightarrow$ *Hardware Modules* $\rightarrow$ *RegisterArray*.
- Lenght=16, size=16 (vogliamo registri dati).
- *View* $\rightarrow$ *Register array R*.
- I registri nell'array sono riferiti attraverso il loro indice (non più accumulatore, ma registro 1, registro 2..., registro 16).
- Abbiamo creato un array con 16 registri: 4 bit per l'indirizzamento (indirizzamento registro!).

Non serve più l'*accumulatore*!

- Per ora non lo rimuoviamo perchè dovremmo ridefinire tutte le operazioni che lo utilizzano, ma ragioniamo come se non ci fosse.

Che registri userà la ALU ora che non c'è più ACC?

- Una soluzione: registri dedicati.
- Definiamo i registri IN1 IN2 OUT, di input e output per la ALU (registri dati)?



Istruzioni

Ora dobbiamo definire le istruzioni per la nuova architettura: come usare questi nuovi registri?
Proviamo a definire una nuova operazione:

- ADD tra registri.
- siccome ora abbiamo a disposizione vari registri dati, sarà necessario specificare
  - primo registro di input
  - secondo registro di input
  - registro di output
- **op + reg input1 + reg input2 + reg output**
  4bit 4 bit 4 bit 4bit
- lunghezza istruzione: 16 bit

Microistruzioni - ALU

- Ridefiniamo la microistruzione di addizione in modo che utilizzi questi nuovi registri (IN1+IN2=OUT).

Definiamo microistruzioni per il trasferimento.

- Da una posizione nell'array di registri agli operandi della ALU(transferAtoR):
  - $ROP1 \rightarrow IN1$ dal registro di posizione indicata dai bit da 4 a 7 dell'IR a IN1;
  - $ROP2 \rightarrow IN2$ dal registro di posizione indicata dai bit da 8 a 11 dell'IR a IN2.
- Dall'output della ALU ai registri (transferRtoA):
  - $OUT \rightarrow ROP3$ da OUT al registro di posizione indicata dai bit da 12 a 15 dell'IR;
- Definiamo l'istruzione *addR reg reg reg* (*reg* è un campo dato unsigned di 4 bit)

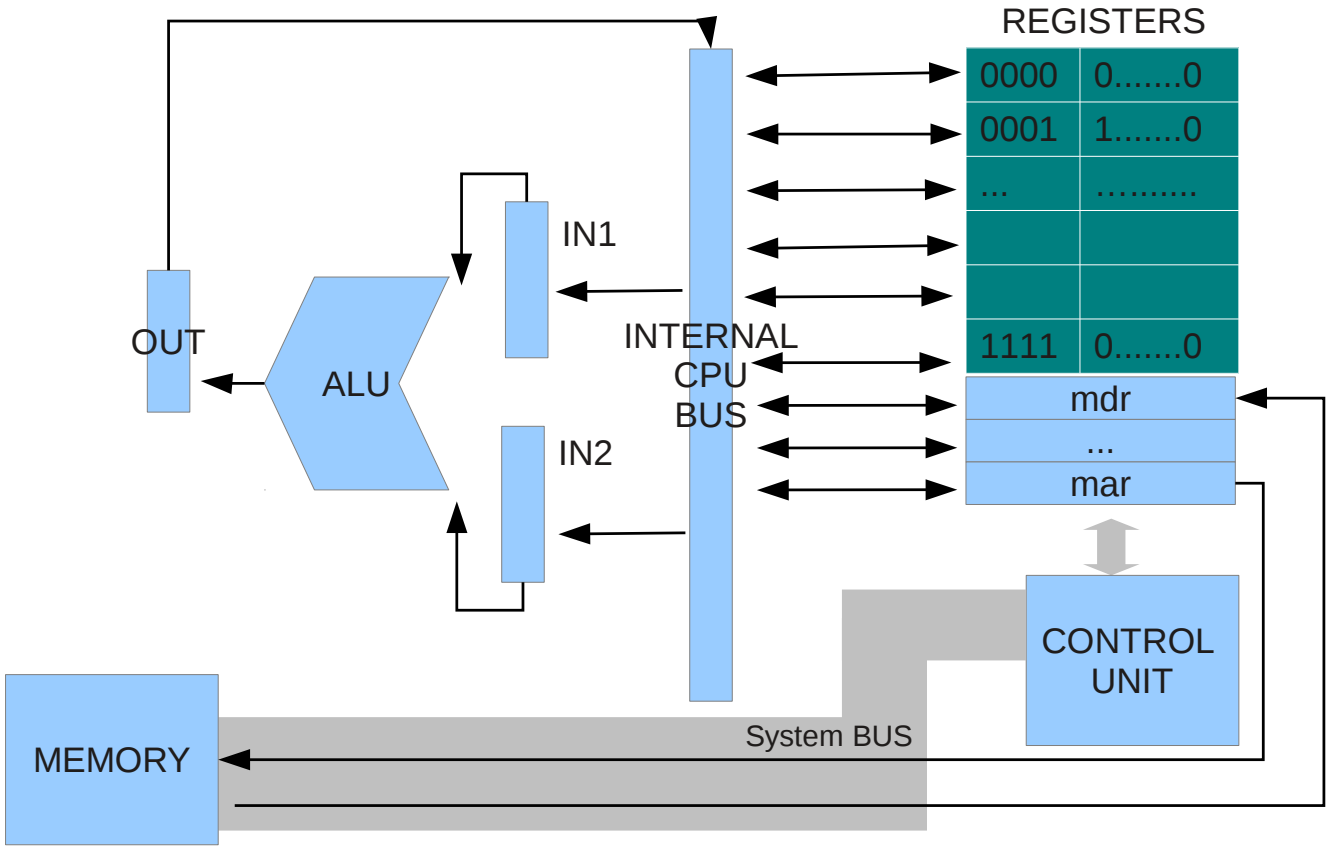
Altre istruzioni: LoadR

- L'istruzione *addR* occupa 16 bit, come tutte le istruzioni Wombat1.
- Proviamo a definire la *loadR*
  - Semantica: carica il contenuto di una cella di memoria in un registro.
  - Quale cella di memoria? serve un campo dati *address*.
  - Quale registro? serve un campo dati *reg*.
  - *op (4bit) + address (12bit) + registro (4bit)* = 20 bit, ma le istruzioni Wombat1 sono di lunghezza fissa di 16 bit!

  Possibili soluzioni:
  - Lunghezza variabile: viene specificato il numero di operandi.
  - Formato ibrido: ogni istruzione ha una dimensione diversa predeterminata.
    - La CPU fa il fetch dei primi 16 bit e decodifica l'istruzione;
    - nell'implementazione dell'istruzione stessa, viene eseguito il fetch della parte mancante.

Wombat 2.1

- Carichiamo la CPU *Wombat2.1.cpu*
- Vediamo l'implementazione della *loadR*.



Wombat 2.1 - Instruction set

INSTRUCTION	MEANING	DESCRIPTION
readR reg1	Read n $\rightarrow$ reg1	Input from keyboard in reg1
writeR reg1	reg1 $\rightarrow$ output	Write value of reg1
multiplyR reg1 reg2 reg3	reg1 * reg2 $\rightarrow$ reg3	Multiply contents of two registers
divideR reg1 reg2 reg3	reg1 / reg2 $\rightarrow$ reg3	Divide contents of two registers
subtractR reg1 reg2 reg3	reg1 - reg2 $\rightarrow$ reg3	Subtract contents of two registers
addR reg1 reg2 reg3	reg1 + reg2 $\rightarrow$ reg3	Add contents of two registers
loadR reg1 addr	Mem[addr] $\rightarrow$ reg1	Load word from memory in reg1
storeR reg1 addr	reg1 $\rightarrow$ Mem[addr]	Store word in memory from reg1
jmpzR reg1 addr	If $reg1 = 0$ jump to addr	Conditional jump (reg1=0)
jmpnR reg1 addr	If $reg1 < 0$ jump to addr	Conditional jump ($reg1 < 0$)
jump addr	jump to addr	
stop	stop execution	

Esercizio 2 - Wombat 2.1

Scrivere un programma ASSEMBLY per la CPU Wombat2.1 che legge un intero in ingresso e ritorna 0 se l'intero è pari, -1 altrimenti (usare divisione intera e moltiplicazione).

Esercizio 2 - Soluzione

Listing 2: Esercizio 2

```
readR 1             ; n -> Reg[1]
loadR 2 due         ; 2 -> Reg[2]
divideR 1 2 3       ; Reg[1]/Reg[2] -> Reg[3]
multiplyR 3 2 4     ; Reg[3]*Reg[2] -> Reg[4]
subtractR 4 1 5     ; Reg[4] - Reg[1] -> Reg[5]
writeR 5            ; visualizza Reg[5]
stop                ; termina il programma
due: .data 2 2      ; variabile due
```

Esercizio 3

Scrivere un programma ASSEMBLY per la CPU Wombat2.1 che calcola e stampa il valore assoluto di un intero ricevuto in input. Ancora molto simile a Wombat1.

Esercizio 3 - Soluzione

Listing 3: Esercizio 3

```
readR  1                ;input -> Reg[1]
jmpnR 1 negativo        ;se Reg[1] <0 vai a negativo
fine: writeR 1          ;Reg[1] -> output
stop                    ;stop
negativo:loadR 0 zero   ;Reg[0] = 0
subtractR 0 1 1         ;Reg[0] - Reg[1] -> Reg[1]
jump fine               ;va alla fine
zero: .data 2 0         ;2 byte dove mettere zero
```

Esercizio 4

Scrivere un programma ASSEMBLY per la CPU Wombat2.1 che calcola il prodotto di due interi usando somme.

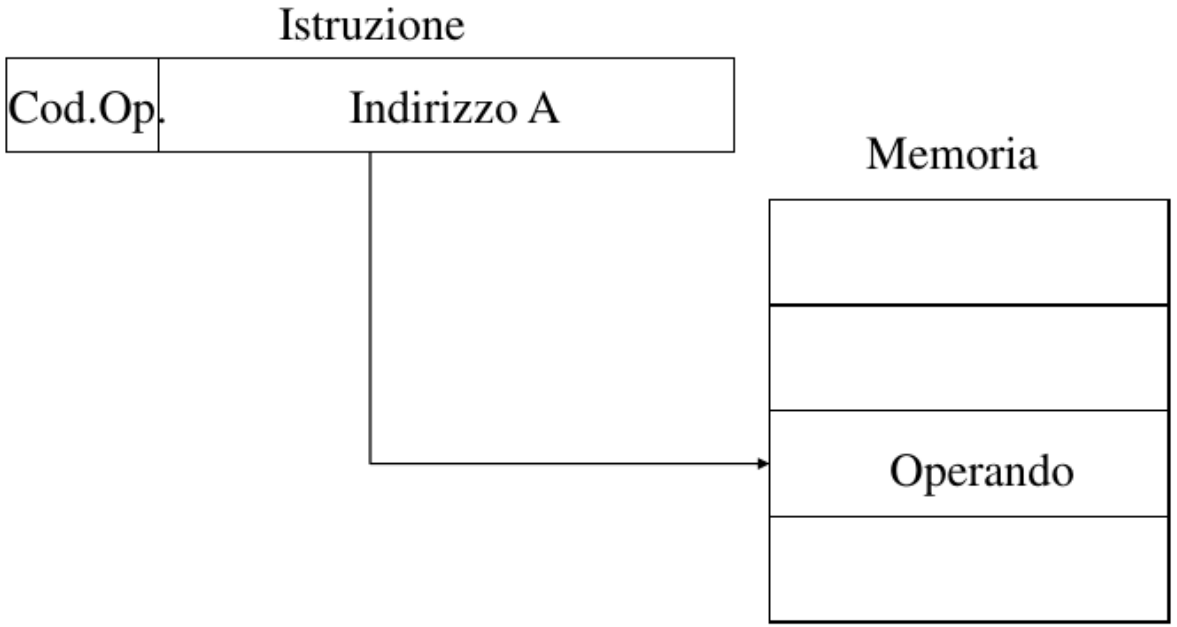
Esercizio 4 - Soluzione

Listing 4: Esercizio 4

```
readR 0                 ; moltiplicando -> Reg[0]
readR 2                 ; moltiplicatore -> Reg[2]
loadR 1 uno             ; uno -> Reg[1]
ciclo: jmpzR 2 fine     ; vai a fine se Reg[2]=0
addR 0 3 3              ; Reg[0] + Reg[3] -> Reg[3]
subtractR 2 1 2         ; Reg[2] - Reg[1] -> Reg[2]
jump ciclo              ; vai a ciclo
fine: writeR 3          ; output Reg[2]
stop
uno: .data 2 1;
```

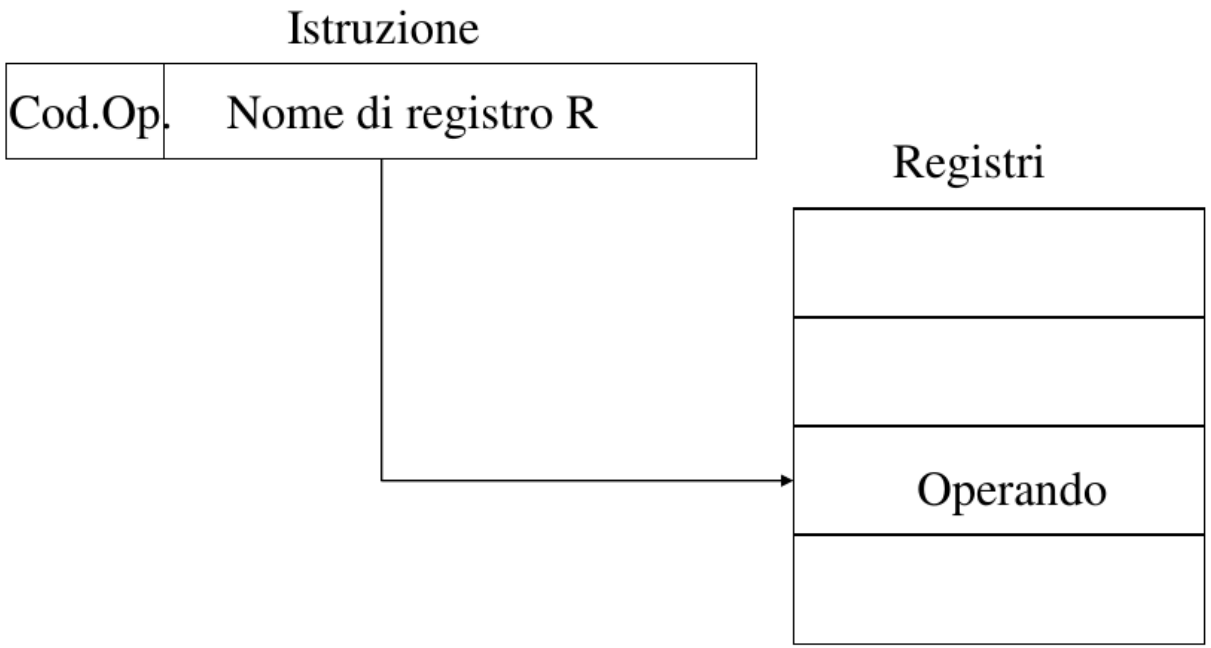
Modi di indirizzamento visti in lab - 1

- Diretto (campo indirizzo = indirizzo dell'operando)
  - CPU Wombat1
  - e.g. ADD X (acc + Mem[X]− >acc)
  - 1 accesso alla memoria all'indirizzo X



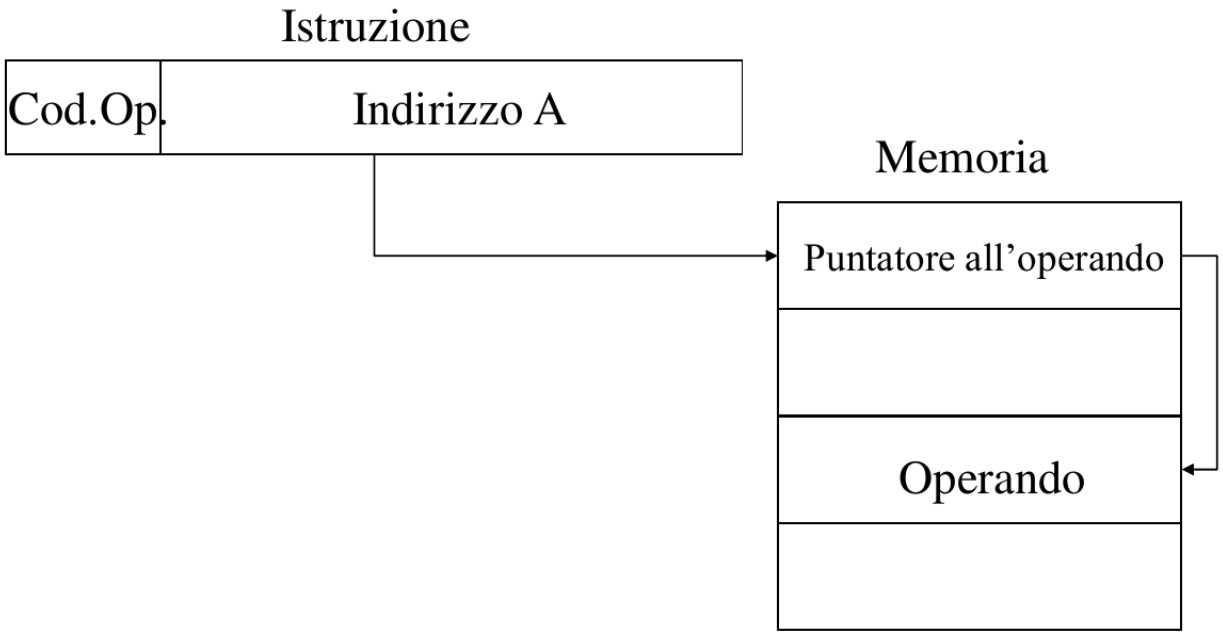
Modi di indirizzamento visti in lab - 2

- Registro (l'operando è in un registro specificato nel campo indirizzo)
  - CPU Wombat2

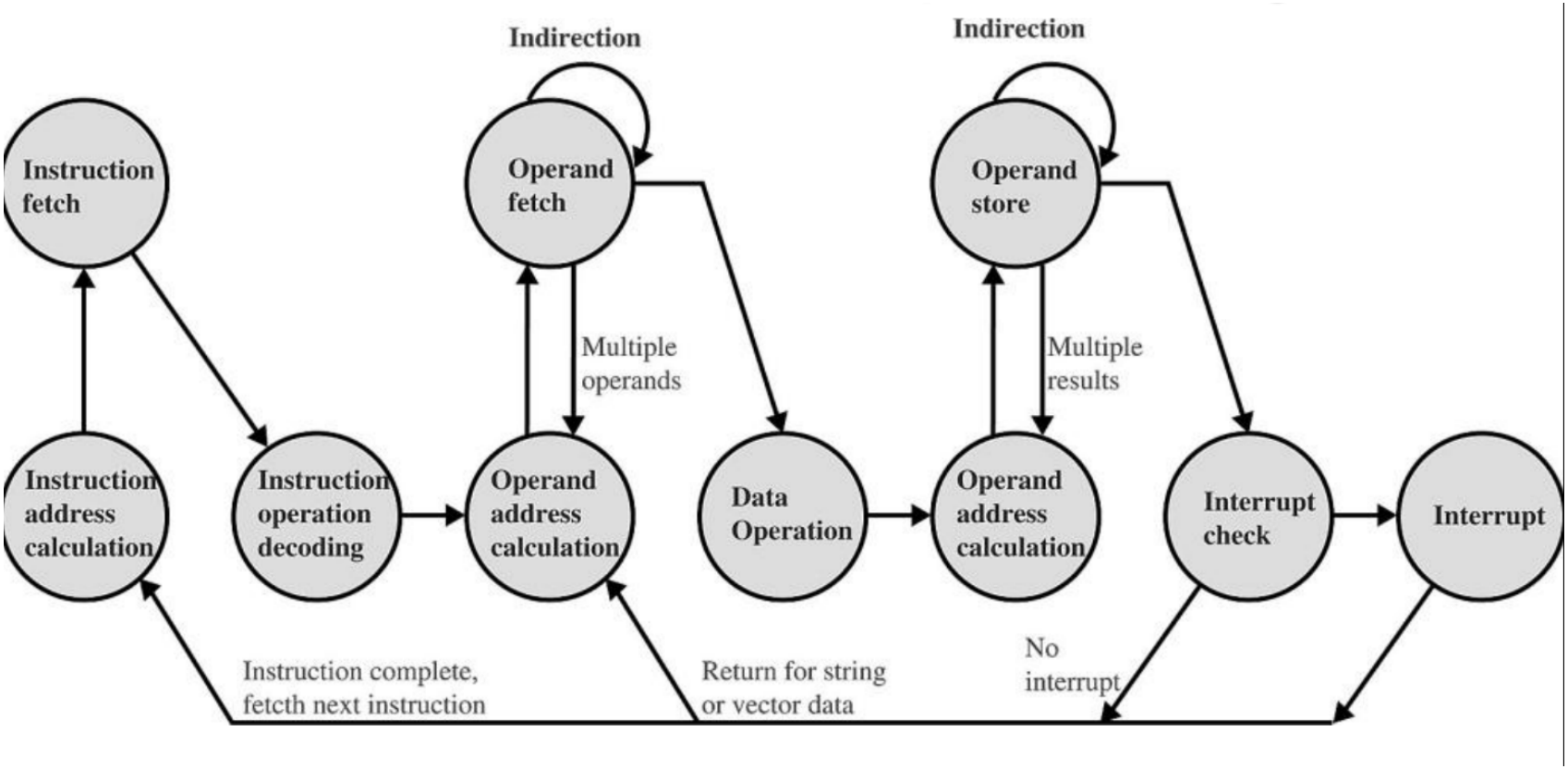


Ora vedremo...

- Indiretto (campo indirizzo= indirizzo di una cella di Mem, che contiene l'indirizzo dell'operando)
  - e.g.



Ind. indiretto - Ciclo di esecuzione



Indirizzamento indiretto: Wombat3

- Apriamo Wombat2.1.cpu
- Salviamo con un nuovo nome, e.g. Wombat3-test.cpu
- Se abbiamo altre istruzioni oltre quelle di default, le cancelliamo

Indirizzamento indiretto: Wombat3

- Definire nuove istruzioni che interpretino gli operandi:
  - *non* come indirizzi di memoria da caricare, ma
  - come indirizzi degli indirizzi di memoria da caricare.
  - sintassi: **op**(4 bit)+**reg**(4 bit)+**address** (del puntatore,12 bit)
  - Servirà una nuova microistruzione:
    - $mdr(4-15)->mar$;
    - copia i 12 bit meno significativi di mdr in mar.
  - Definiamo *LoadR* e *StoreR* con indirizzamento indiretto.

Nuove Istruzioni

- **loadRInd reg addr**
  - *load registro indiretta*: addr è l'indirizzo dell'indirizzo della locazione di memoria dove caricare il contenuto di *reg*.
  - *Modify* $\rightarrow$ *MachineInstructions*
  - Duplichiamo la *loadR* e la modifichiamo:

Listing 5: loadInd

```
pc->mar
Main[mar]->mdr
mdr(4-15)->mar
Main[mar]->mdr
mdr(4-15)->mar
Main[mar]->mdr
mdr->ROP1
inc2-pc
End
```

- Definire **storeInd**

Wombat3 - Instruction Set

INSTRUCTION	MEANING	DESCRIPTION
readR reg1	Read n $\rightarrow$ reg1	Input from keyboard in reg1
writeR reg1	reg1 $\rightarrow$ output	Write value of reg1
multiplyR reg1 reg2 reg3	reg1 * reg2 $\rightarrow$ reg3	Multiply contents of two registers
divideR reg1 reg2 reg3	reg1 / reg2 $\rightarrow$ reg3	Divide contents of two registers
subtractR reg1 reg2 reg3	reg1 - reg2 $\rightarrow$ reg3	Subtract contents of two registers
addR reg1 reg2 reg3	reg1 + reg2 $\rightarrow$ reg3	Add contents of two registers
loadR reg1 addr	Mem[addr] $\rightarrow$ reg1	Load word from memory in reg1
storeR reg1 addr	reg1 $\rightarrow$ Mem[addr]	Store word in memory from reg1
jmpzR reg1 addr	If $reg1 = 0$ jump to addr	Conditional jump (reg1=0)
jmpnR reg1 addr	If $reg1 < 0$ jump to addr	Conditional jump ($reg1 < 0$)
jump addr	jump to addr	
stop	stop execution	
loadRInd reg1 addr	Mem[Mem[addr]] $\rightarrow$ reg1	Indirect load from memory in reg1
storeRInd reg1 addr	reg1 $\rightarrow$ Mem[Mem[addr]]	Indirect Store in memory from reg1

Esercizio 5

- Utilizzando istruzioni ad indirizzamento indiretto, il programma legge un valore X ed una sequenza di interi, li somma finché non legge un numero negativo, e salva la somma parziale in memoria all'indirizzo X. Il valore X non va conteggiato nella somma. Alla fine stampa la somma (senza includere l'ultimo numero).
- Nota: se definiamo la locazione per la somma ad un indirizzo occupato dal codice del programma, succedono cose strane! (Il codice viene sovrascritto).

Esercizio 5 - Soluzione

Listing 6: Esercizio 5

```
loadR 0 zero ;Mem[zero]->R[0]
readR 4 ; legge X->R[4]
storeR 4 somma; R[4]->Mem[somma]
storeRInd 0 somma ;R[0]-> Mem[Mem[somma]]
inizio: readR 1 ;legge n->R[1]
jmpnR 1 fine ;salta a Fine se R[1]<0.
addR 1 0 0 ;R[1]+R[0]->R[0]
storeRInd 0 somma ;R[0]->Mem[Mem[somma]]
jump inizio ;torna a inizio
fine: loadRInd 3 somma ;Mem[Mem[somma]]->R[3]
writeR 3 ;output R[3]
stop
zero: .data 2 0 ; zero
somma: .data 2 40 ;locazione (2 byte) per l'ind
                  ;dove
```

Esercizio 6

- Utilizzando istruzioni ad indirizzamento indiretto, il programma legge due interi, chiamiamoli *ind* e *cont*. Successivamente, scrive nell'indirizzo di memoria *ind* e nei *cont* successivi i valori $cont, cont - 1 \ldots 1$, rispettivamente.
- E.g. al termine dell'esecuzione del programma con $ind = 20$ e $cont = 4$, la situazione della memoria sarà la seguente.

0	0
2	0
...	...
20	4
22	3
24	2
26	1
...	...

Esercizio 6 - Soluzione

Università degli Studi di Padova

Listing 7: Esercizio 6

```
 readR 0 ;ind su cui scrivere->R[0]
storeR 0 ind ;R[0]->ind
loadR 1 uno;
loadR 2 due
readR 3; valore da scrivere->R[3]
ciclo: jmpzR 3 fine ;se R[3]=0 vai a fine
storeRInd 3 ind
subtractR 3 1 3
addR 0 2 0
storeR 0 ind
jump ciclo
fine: stop

due: .data 2 2
uno: .data 2 1;
ind: .data 2 0;locazione base
```