

CPUSim

Laboratorio 14/11/2012.

Nicolò Navarin

e-mail: nnavarin@math.unipd.it



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

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 1: 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 2: 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.

Listing 3: 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;
```

Limitazioni di *Wombat1*:

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

Creiamo una nuova cpu partendo da *Wombat1*:

- aprire la cpu di esempio *Wombat1*;
- *File* → *Save Machine As Wombat2-test.cpu*.

Possiamo modificare le specifiche attraverso il menu *Modify*.

Definiamo un array di registri.

- *Modify* → *Hardware Modules* → *RegisterArray*.
- Length=16, size=16 (vogliamo registre dati).
- *View* → *Register array A*.
- 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.

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).
- Ridefiniamo la microistruzione di addizione in modo che utilizzi questi nuovi registri.

Definiamo microistruzioni per il trasferimento.

- Da una posizione nell'array di registri agli operandi della ALU:
 - dal registro di posizione indicata dai bit da 4 a 7 dell'IR a IN1;
 - dal registro di posizione indicata dai bit da 8 a 11 dell'IR a IN2.
- Dall'output della ALU ai registri (transferRtoA)
 - da OUT al registro di posizione indicata dai bit da 12 a 15 dell'IR;

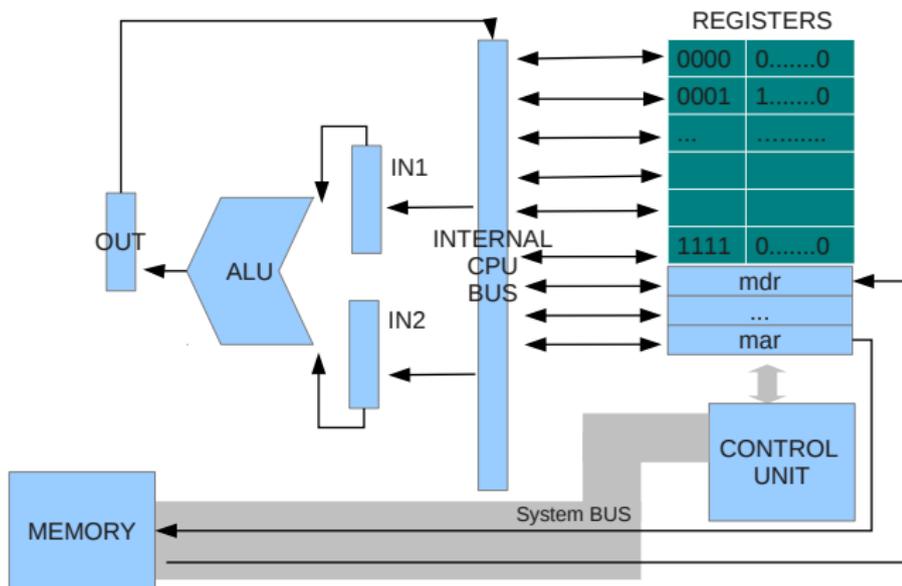
Ora dobbiamo definire le istruzioni per la nuova architettura come usare questi nuovi registri.

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
- Definiamo l'istruzione *addR reg reg reg* (*reg* è un campo dato unsigned di 4 bit)

- 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!
 - Lunghezza variabile: viene specificato il numero di operandi.
 - Formato ibrido: ogni istruzione può avere 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.

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



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



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).

Listing 4: 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
```

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.

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