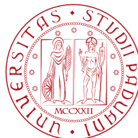
CPUSim - 2

Laboratorio 14/12/2016

Tommaso Padoan

e-mail: padoan@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 perché aumenta la complessità delle istruzioni.

Prima di iniziare:

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

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

Università degli Studi di Padova

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
```

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 Wombat1:

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

Possiamo modificare le specifiche attraverso il menu *Modify*.

Array di registri generici

Definiamo un array di registri:

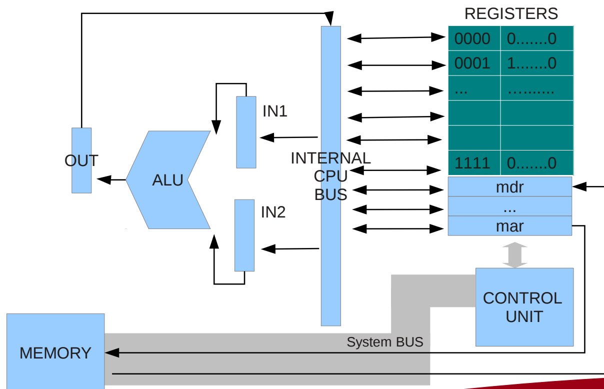
- *Modify* → *Hardware modules* → *RegisterArray*
- length=16 (numero di registri), width=16 (registri dati)
- *View* → *Register array R*
- i registri nell'array sono riferiti attraverso il loro indice (non più accumulatore, ma registro 0, registro 1,. . . , registro 15)
- abbiamo creato un array con 16 registri: 4 bit per l'indirizzamento (indirizzamento registro!).

Non serve più l'*accumulatore*!

- per ora non lo rimuoviamo perché 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 e 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 (dell'array)
- 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**
  4 bit 4 bit 4 bit 4 bit
- lunghezza istruzione: 16 bit.

Microistruzioni - ALU

Ridefiniamo la microistruzione di addizione in modo che utilizzi i nuovi registri dedicati (IN1+IN2→OUT).

Definiamo microistruzioni per il trasferimento:

- da una posizione nell'array di registri agli input della ALU (*transferAtoR*):
  - *ROP1* → *IN1* dal registro di posizione indicata dai bit da 4 a 7 dell'IR a IN1
  - *ROP2* → *IN2* dal registro di posizione indicata dai bit da 8 a 11 dell'IR a IN2
- dall'output della ALU ad un registro nell'array (*transferRtoA*):
  - *OUT* → *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: load

L'istruzione *addR* occupa 16 bit, come tutte le istruzioni di 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 *register*
- **op (4 bit) + addr (12 bit) + register (4 bit) = 20 bit**, ma le istruzioni in Wombat1 sono di lunghezza fissa di 16 bit!

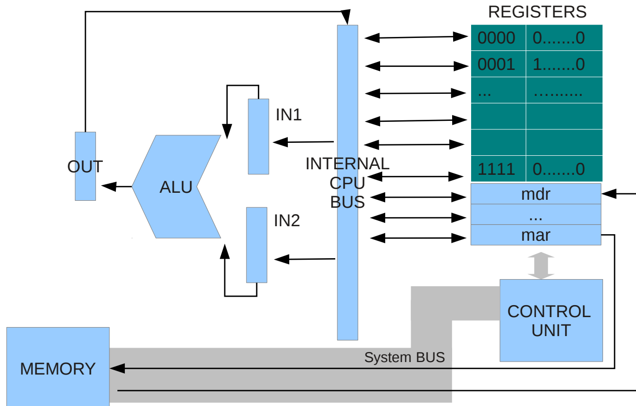
Lunghezza istruzioni

Come fare per poter usare anche istruzioni più grandi?

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.

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



Wombat 2.1 - Instruction set

INSTRUCTION	MEANING	DESCRIPTION
readR reg1	input → reg1	Input from keyboard in reg1
writeR reg1	reg1 → output	Write value of reg1
multiplyR reg1 reg2 reg3	reg1 * reg2 → reg3	Multiply contents of two registers
divideR reg1 reg2 reg3	reg1 / reg2 → reg3	Divide contents of two registers
subtractR reg1 reg2 reg3	reg1 - reg2 → reg3	Subtract contents of two registers
addR reg1 reg2 reg3	reg1 + reg2 → reg3	Add contents of two registers
loadR reg1 addr	Mem[addr] → reg1	Load word from memory in reg1
storeR reg1 addr	reg1 → 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 1

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 1 - Soluzione

Università degli Studi di Padova

Esercizio 1

```
        readR 0             ; input -> R[0]
        jmpnR 0 neg         ; se R[0]<0, salta a neg
fine:   writeR 0            ; output R[0]
        stop                ; termina esecuzione
neg:    loadR 1 -uno        ; M[-uno] -> R[1]
        multiplyR 0 1 0     ; R[0] * R[1] -> R[0]
        jump fine           ; salta a fine

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

Esercizio 2

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

Esercizio 2 - Soluzione

Esercizio 2

```
        readR 0              ; input -> R[0]
        readR 1              ; input -> R[1]
        loadR 2 zero         ; M[zero] -> R[2]
        loadR 3 uno          ; M[uno] -> R[3]
ciclo:  jmpzR 1 fine         ; se R[1]=0, va a fine
        addR 2 0 2           ; R[2] + R[0] -> R[2]
        subtractR 1 3 1      ; R[1] - R[3] -> R[1]
        jump ciclo           ; salta a ciclo
fine:   writeR 2             ; output R[2]
        stop                 ; termina esecuzione

zero:  .data 2 0 ; il valore 0
uno:   .data 2 1 ; il valore 1
```