

# **Implementatie Programmeertalen**

# Moderne Programmeertalen

Turing machine niet praktisch voor echte problemen

## Eisen aan programmeertaal

- effectief en efficient
  - een programmeur moet zonder al te veel moeite kunnen programmeren: algoritmen makkelijk uit te drukken
  - de taal moet efficient door een moderne computer verwerkt kunnen worden
  - we bekijken een subset van Python en bekijken hoe die wordt afgebeeld op een moderne processor architectuur

# Essential Python: voorbeeld faculteit

```
1 def fac(n):  
2     res = 1  
3     k = 1  
4     while k <= n:  
5         res = res * k  
6         k = k + 1  
7     return res  
8  
9 print(fac(5))
```

# Analyse

`fac` is een functie

`k` en `res` zijn variabelen

`k = 1, res = 1` zijn toekenningen

`while k <= n:` is een conditioneel statement

`res = res * k` is het berekenen van een  
expressie en een toekenning

met `return res` beëindig je de functie en geef  
je het resultaat terug

# Moderne Microprocessor

Voert instructies uit

- simpele operaties die (in principe) sequentieel worden uitgevoerd
- instructies staan in geheugen
- de instructie die wordt uitgevoerd staat op positie pc (program counter)
- na iedere instructie wordt pc met 1 verhoogd
- instructies kunnen waarden in geheugen gebruiken en wijzigen

# Geheugen

Is fysiek aaneengesloten: groot array

Maar wordt logisch opgedeeld in delen:

- programma gedeelte: waar instructies staan
- stack gedeelte: voor lokale variabelen  
functies, functie administratie en voor  
berekeningen
- heap: voor de opslag van data zoals lijsten,  
etc
- evt: constant pool, zoals strings en andere  
constante waarden uit het programma

# Processoren

- de processor hardware is een interpreter voor de instructies
- de instructies bepalen wat een processor kan
- processoren met dezelfde instructie set zijn compatibel (i86 familie Intel) maar ook de AMD processoren
- kunnen wel hardwarematig heel verschillend zijn (sneller is duurder)

# Virtuele Machines

- virtuele processor met virtuele instructies
- als programma (interpreter) gerealiseerd op andere processor
- meestal eenvoudiger dan echte processoren
- instructies virtuele machine heten bytecode
- voorbeelden:
  - Java Virtuele Machine JVM
  - Python virtuele machine
  - WebAssembly voor snelle executie in webbrowsers
  - JMVM: speciaal voor functionele talen
- Voor dit vak gebruiken we een vereenvoudigde versie van de JVM



# Java Virtuele Machine JVM

- Java of Python programma vertaald naar byte code instructies
- Byte code instructies uitgevoerd door JVM
- JVM is processor architectuur
  - kan in principe in hardware worden uitgevoerd
  - meestal als virtuele machine gerealiseerd (interpreter)
  - ingebouwde Just-In-Time compiler vertaald door naar native instructies van processor waarop JVM draait
  - hierdoor toch hoge performance

# JVM

JVM heeft een stack architectuur

- operaties werken op stack
- variabelen staan op stack
- argumenten functies e.d gaan ook via stack

Vertaald programma (instructies)

- staat in geheugen JVM
- worden na elkaar uitgevoerd
- programma kan springen

# Stack

Vergelijkbaar met PushDown automaat

- bovenaan toevoegen en verwijderen (push en pop)
- rekenkundige bewerkingen op bovenste (1 of 2) elementen
- opslag van lokale variabelen

# Soorten instructie

## Rekenkundige

- operanden op de stack
- resultaat op de stack

## Branching (sprong)

- functies calls
- if, goto

## Geheugen

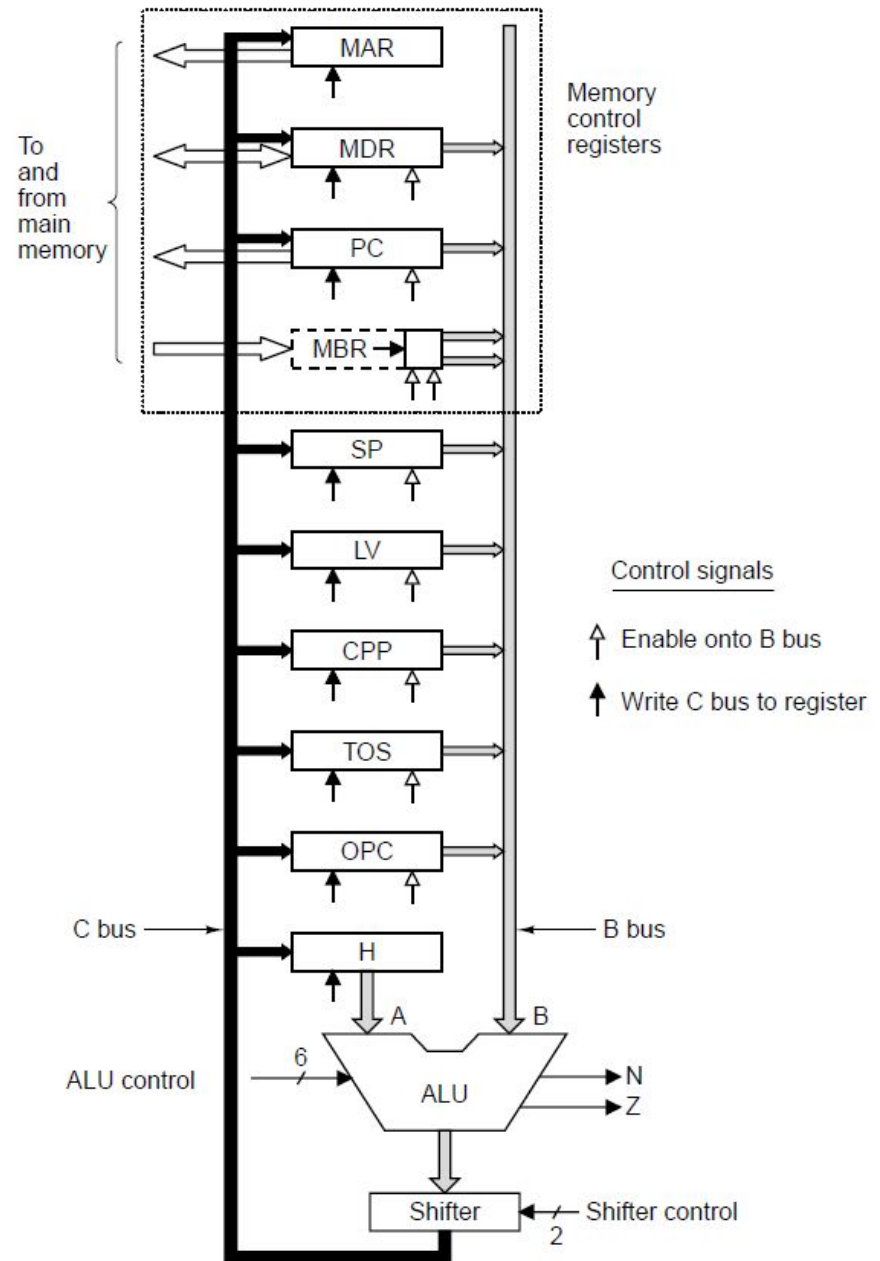
- laden (uit heap of stack naar stack)
- opslaan (op head of stack van waarde op stack)

# Instructions

Beschouwen nu subset van JVM instructieset (IJVM)

- geschikt voor simpele programma's zoals Essential Python
- alleen operaties op getallen

# Datapad IJVM



# Datapad

Schematisch overzicht van processor met datapaden en registers

ALU doet de operaties

Registers spelen een rol in administratie

1. SP: top van de stack (beschouw stack als array)
2. PC: adres volgende instructie (ook array)
3. LV: positie op stack van argumenten en lokale variabelen functie
4. TOS: kopie bovenste element stack
5. andere registers voor ons niet van belang (bv interactie met geheugen)

# Instructieset IJVM

<b>bipush const</b>	zet constante op stack
<b>goto offset</b>	zet pc op pc + offset
<b>iadd</b>	tel bovenste 2 elementen van stack op en zet resultaat op stack
<b>isub</b>	trek bovenste 2 elementen van stack af .....
<b>imult</b>	vermenigvuldig bovenste 2 elementen van stack ....
<b>idiv</b>	deel bovenste 2 elementen van stack ....
<b>land</b>	boolean and bovenste 2 elementen van stack ....
<b>lor</b>	boolean or bovenste 2 elementen van stack ....
<b>iinc vn const</b>	verhoog variabele met offset vn tov LV met const
<b>iload vn</b>	laad variable met offset vn tov LV op stack
<b>istore vn</b>	pop waarde van stack en dit in variabele met offset vn tov LV
<b>dup</b>	dupliceer top van stack
<b>pop</b>	pop waarde van stack
<b>swap</b>	verwissel bovenste 2 elementen stack
<b>iflt offset</b>	pop bovenste waarde stack, als $< 0$ ga verder bij pc + offset
<b>ifeq offset</b>	pop bovenste waarde stack, als $= 0$ ga verder bij pc + offset
<b>if_compeq offset</b>	pop bovenste 2 waarden stack, als gelijk ga verder bij pc + offset
<b>call nrargs nrlv npc</b>	uitleg apart
<b>ireturn</b>	uitleg apart



# Rekenkundige Instructies

Zet 6 en 7 op stack en tel ze op:

```
bipush 6
```

```
bipush 7
```

```
iadd
```

Na afloop staat 13 boven op de stack

# Rekenkundige expressies

3+6\*7

(3+6)\*7

wordt

wordt

bipush 3

bipush 3

bipust 6

bipust 6

bipush 7

iadd

imult

bipush 7

iadd

imult

Andersom

bipush 4

bipush 5

bipush 8

bipush 6

isub

imult

iadd

Probeer zelf

6 + 7 + 8

6 - 3 - 1

6 - (3 - 1)

17 - 3 \* (6 + 3 - 1)

# While en If

Test op bovenste waarde(n) op stack en spring (jump) naar pc + offset

While gaat op dezelfde manier

- mbv goto spring je terug naar if

Zie voorbeelden

# Functie

call, ireturn: om functie calls te doen

## Functie

- heeft argumenten
- heeft lokale variabelen
- geeft resultaat

## Basis Idee

- zet argumenten op de stack
- doe functie call
- resultaat staat daarna op top stack

# Functie call

Wat moet er gebeuren bij  
**call nrargs nrlv npc ?**

1. Argumenten staan er al
2. Maak extra ruimte op stack voor lokale variabelen (gebruik nrlv)
3. Zet oude pc en lv hier boven op
4. Update lv naar positie eerste argument
  - a. gebruik hiervoor nrargs en nrlv
5. Zet pc op eerste instructie functie

# Voorbeeld

```
def f(a, b):  
    c = a + b  
    return c
```

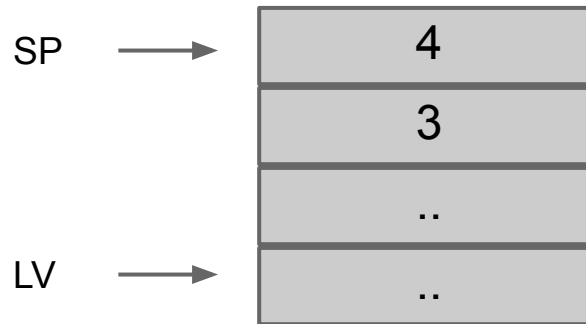
...

```
x = f (3,4);
```

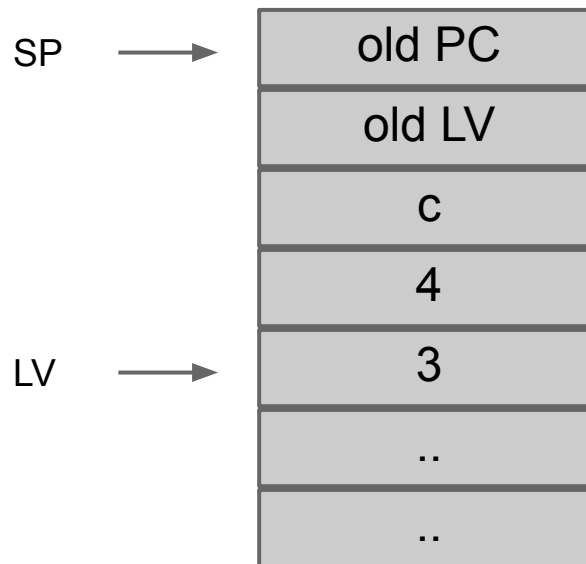
```
bipush 3
```

```
bipush 4
```

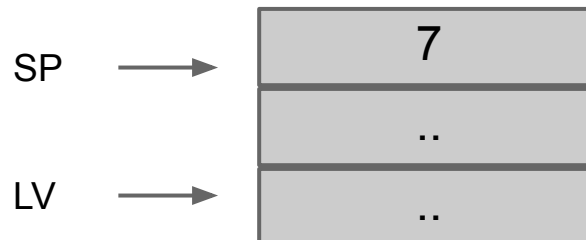
```
call 2 1 label
```



Voor



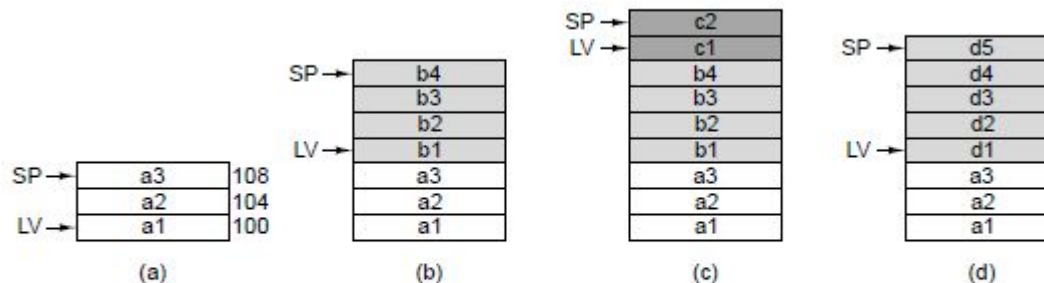
Tijdens



Na

# Lokale variabelen en stack

- Iedere functie call krijgt een eigen segment op de stack waarin argumenten en lokale variabelen staan opgeslagen (stack frame, daar staat ook nog wat administratie)
- Na return wordt dit weer opgeruimd.
- LV wijst steeds naar eerste argument huidige functie
- Functie calls kunnen genest en zelfs recursief zijn!



**Figure 4-8.** Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

# Noodzaak stack icm call en return

De eerste generaties computer en programmeertalen maakten geen gebruik van een stack voor opslag variabelen

- alle posities van lokale variabelen lagen van te voren vast
- hierdoor geen recursie mogelijk
- maar ook geen mogelijkheid tot stack overflow!



# Voorbeelden

We gebruiken labels voor goto, if\_, call

- geen offset berekeningen meer nodig
- is alleen voor leesbaarheid
- wordt vertaald naar adressen

## Voorbeeld

- aanroepcode
- minimaal 1 functie
- PRINT print top stack (doet ook pop)
- STOP stopt het programma

# Code voorbeelden

```
def telop(a,b):
```

```
    c = a + b
```

```
    return c
```

```
print(telop(3,4))
```

```
bipush 3
```

```
bipush 4
```

```
call 2 1 33
```

```
print
```

```
stop
```

```
33 iload 0
```

```
    iload 1
```

```
    iadd
```

```
    istore 2
```

```
    iload 2
```

```
    ireturn
```

# Code voorbeelden

```
def sumloop(n):
```

```
    s = 0
```

```
    while n != 0:
```

```
        s = s + n
```

```
        n = n - 1
```

```
    return s
```

```
sumloop(10)
```

```
bipush 10
```

```
call 1 1 1
```

```
print
```

```
stop
```

```
1 bipush 0
```

```
    istore 1
```

```
2 printstack
```

```
    iload 0
```

```
    ifeq 3
```

```
    iload 1
```

```
    iload 0
```

```
    iadd
```

```
    istore 1
```

```
    iload 0
```

```
    bipush 1
```

```
    isub
```

```
    istore 0
```

```
    goto 2
```

```
3 iload 1
```

```
    ireturn
```

# Code voorbeelden

```
sumrec(10)
```

```
def sumrec(n):  
    if n == 0:  
        return 0  
    else :  
        return (n + sumrec(n-1))
```

```
bipush 10  
call 1 0 1  
printstack  
print  
stop  
1 printstack  
  iload 0  
  ifeq 2  
  iload 0  
  dup  
  bipush 1  
  isub  
  call 1 0 1  
  iadd  
  printstack  
  ireturn  
2 bipush 0  
  ireturn
```

# Hanoi

```
bipush 3          iload 1
bipush 1          print
bipush 2          iload 2
bipush 3          print
call 4 0 33       iload 0
pop              bipush 1
stop             isub
33 iload 0         iload 3
ifeq 44          iload 2
iload 0          iload 1
bipush 1         call 4 0 33
isub            pop
iload 1         bipush 77
iload 3         ireturn
iload 2         44 bipush 66
call 4 0 33     ireturn
pop
```

```
def hanoi(n,a,b,c):
    if n > 0:
        hanoi(n-1,a,c,b)
        print(a,b)
        hanoi(n-1,c,b,a)

hanoi(5,1,2,3)
```

# Uitleg Simulator

7\*(8+4)

Kwadraat

Som recursief

Som loop

GGD

Fibonacci

Hanoi

```
1  bipush 7
2  bipush 8
```

Kies een voorbeeld of typ zelf een programma

Run

Load

Step

Reset

Clear

Stop

```
1  > Top of stack: 84
2  > Top of stack: 84
3
```

Run: voer het programma uit tot het eind

Load: (her)laad het programma voor step

Step: voer 1 instructie uit

Reset: ga in step weer terug naar instructie 1 en wis het uitvoerscherm

Clear: wis het uitvoerscherm

Stop: onderbreek een programma dat in een oneindige loop is geraakt

# Opgave

Maak eerst in Python een programma `deler(k,n)` dat test of  $n$  deelbaar is door  $k$

Doe dit door herhaaldelijk  $k$  op te tellen vanaf  $k$  tot dat je precies op  $n$  komt of er overheen gaat (geef 0 of 1 als resultaat).

Vertaal dit programma naar JVM code.