

Hoofdstuk 4

Programmatuur



H4: Programmatuur

- Het assembleerprogramma
- Subroutines, linker, macro's
- Hogere vs. lagere programmeertalen
- De Java virtuele machine



H4: Programmatuur

- Het assembleerprogramma
 - Voorbeeld van een programma (zonder UCLL-bevelen)
 - Taak van het assembleerprogramma
 - Vertaalwerk (zonder symbolische adressen)
 - Vertaalwerk (met symbolische adressen)
 - Na vertaling: de uitvoering

Voorbeeld van een programma

- Vb.: Uit een gegeven lijst getallen alle niet-negatieve getallen optellen, de som omrekenen naar ASCII en opslaan vanaf adres **posext**; idem voor de negatieve getallen.
 - **Zonder 'UCLL-bevelen'!**

```
extern ExitProcess  
[section .data]  
tabel:    dd 45  
          dd 100  
          dd -15  
          dd 65  
          dd 75  
          dd -22  
          dd -18  
          dd -7  
          dd 20  
          dd 10  
  
wneg:     resd 1  
posext:   times 12 db (' ')  
negext:   times 12 db (' ')
```

[section .code]

start:

```
    sub ebx, ebx      ; EBX: pos. getallen  
    sub edx, edx      ; EDX: neg. getallen  
    sub edi, edi  
    mov ecx, 10
```

lus: mov eax, [tabel + edi]

```
        cmp eax, 0
```

```
        jl nega
```

```
        add ebx, eax
```

```
        jmp verder
```

nega: add edx, eax

verder: add edi, 4

```
        loop lus
```

```
    mov [wneg], edx
    mov eax, ebx
    std
    mov edi, posext + 11
    mov ebx, 10
lusp:   mov edx, 0
        idiv ebx
        add dl, 30h
        xchg al, dl
        stosb
        xchg al, dl
        cmp eax, 0
        jne lusp
```

```
    mov edi, negext + 11
    sub eax, eax
    sub eax, [wneg]
lusn:   mov edx, 0
        idiv ebx
        add dl, 30h
        xchg al, dl
        stosb
        xchg al, dl
        cmp eax, 0
        jne lusn
        mov al, '-'
        stosb
        call ExitProcess
```

Machine- en assemblerbevelen

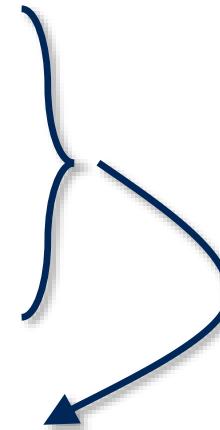
- In een programma staan
 - Machinebevelen
add, mov, sub, imul, ...
 - Assembler-bevelen
extern, section, ...
- **Assembler-bevelen** zijn aanwijzingen aan het vertaalprogramma. Het vertaalprogramma gebruikt deze aanwijzingen tijdens het vertalen. Synoniem: **DIRECTIEF**.

Uit de UCLL-bevelen

- **extern ExitProcess**
...
call ExitProcess
 - Terug naar het besturingssysteem. ExitProcess is een symbolisch adres dat elders gedefinieerd is.
- **[section .data]**
...
[section .code]
 - Hier begint het data-, het code-segment.
- **start:**
 - Hier begint de uitvoering

Taak van het vertaalprogramma

- Wij gebruiken
 - Mnemotechnische functiecodes
 - Symbolische adressen
 - Directe operanden
 - ...
- Processor verwacht machinebevelen



In de vertaalde versie moet alles staan wat nodig is om het bevel te kunnen uitvoeren.

Wat is alles?

- Functiecode (**wat** wil je doen)
- Operand (**met wat** wil je werken)
 - Directe operand?
 - Verplaatsing?
- Welk register van de C.V.E.
 - EAX, of EBX, of ... , of EDI, of ESI, ...

***Opmerking:** soms is de ganse vertaling
maar één byte, bvb. movsb*



de byteteller

[section .data]					
000	2D000000	tabel:	dd	45	
004	64000000		dd	100	
008	FFFFFFF		dd	-15	
00C	41000000		dd	65	
010	4B			75	
014	EA	De byteteller <ul style="list-style-type: none">• <i>per segment een andere teller</i>• <i>begint bij 0</i>• <i>waarde byteteller is aantal bytes dat al gebruikt is.</i>			
018	EB			-22	
01C	F9			-18	
020	14			-7	
024	0A000000		dd	20	
028	<res 00000004>	wneg:	resd	1	
02C	20<rept>	posext:	times 12 db (' ')		
038	20<rept>	negext:	times 12 db (' ')		



000	29DB	sub ebx,ebx
002	29D2	sub edx,edx
004	29FF	sub edi,edi
006	B90A000000	mov ecx,10
00B	8B87	lus:
011	3D00000000	mov eax,[tabel+edi]
016	7C ..	cmp eax,0
018	01C3	j1 r
01A	E9	add ebx,eax
01F	01C2	jmp verder
021	81C704000000	nega:
027	E2..	verder:
029	8915.....	add edx,eax
02F	89D8	add edi,4
031	FD	loop lus
...	...	mov [wneg],edx
		mov eax,ebx
		std
		...

Symbolische Adressen??

Definitie van symbolisch adres

- Vb. 1

...

nega: add edx, eax

...

- betekent: met nega bedoel ik het adres van de byte waar dit bevel zal beginnen.

- Vb. 2

...

wneg: resd 1

...

- betekent: met wneg bedoel ik het adres van de byte waar dit dubbelwoord zal beginnen.

Symbolische Adressen

- Naar een symbolisch adres wordt **verwezen**
 - De referentie kan eerst komen, d.w.z.: vóór de definitie
- Niet elk bevel kan bij de eerste lezing vertaald worden
 - Vertaling gebeurt in twee fasen

Vertalen in fasen

- **1° fase (1st pass):**
 - Elke lijn van de broncode lezen en bytes tellen
 - Aparte teller voor elk segment (gegevens, bevelen)
 - Bij definitie van symbolisch adres: info bijhouden
 - symbolisch adres, waarde v.d. byteteller opslaan in een tabel
 - = de symbolentabel maken
- **2° fase (2nd pass): alles vertalen.**



Fase 1: de symbolentabel

000 [section .data]

tabel: dd 45

dd 100

yteteller

028 wneg: resd 1

02C posext: times 12 db ('')

038 negext: times 12 db ('')

000 「section .code」

start:

sub ebx, ebx

3

mov eax, ...

3

Symbool	Waarde
tabel	00000000
wneg	00000028
posext	0000002C
negext	00000038
lus	0000000B
...	

Fase 1: de symbolentabel

000 [section .data]

000 tabel: dd 45

004 dd 100

Besluit: tijdens 1^e fase wordt de symbolentabel gemaakt. Voor elk symbolisch adres bevat de tabel:

- De verplaatsing t.o.v. de basis
- Het segment waar het gedefinieerd is
- Nog andere informatie

Symbool	Waarde
tabel	00000000
wneg	00000028
posext	0000002C
negext	00000038
lus	0000000B
...	

000 sub_eax, eax

002 ...

00B lus: mov eax, ...

011 ...

Fase 2: alles vertalen

- Vertaalprogramma leest weer iedere lijn
 - Alles vertalen, ook ieder symbolisch adres
- Anders voor s.a. uit code- of uit data-segment.
 - Constanten en veranderlijken: verplaatsing = (waarde v.d. byteteller)
 - Sprongbevelen: niet het adres v.h. bevel, maar “spring een aantal bytes verder”

Vertalen van sprongbevelen

017 7C ??

j1 nega

019 ...

- Vertaalprogramma weet hoe de sprong uitgevoerd wordt:
 - haalcyclus
 - uitvoeringscyclus
- Als de sprong moet uitgevoerd worden, hoe moet de bevelenteller aangepast w.?
 - In bevelenteller zal 0000019 staan
 - nega = 00000020 (symbolentabel)
 - j1 nega wordt 7C 07 omdat 0000019 + 0000007 = 00000020

Bevelenteller vs byteteller

- Tijdens vertalen: bvtellen.
- Hoe verandert de bevelesteller?
 - beginnend op de bevellijn.
 - tijdens vertalen, de lengte bijtellen.

- Tijdens de uitvoering: bevelenteller aanpassen.
- Hoe verandert de bevelesteller?
 - beginnend op de bevellijn.
 - tijdens uitvoeren, de lengte bijtellen.

Vertalen van sprongbevelen

00C 8B87 ... lus: mov eax, [tabel + edi]

...

028 E2 ?? loop lus

02A ...

- Als de sprong moet uitgevoerd worden, hoe moet de bevelentellerangepast worden?
 - Hoeveel zal in bevelenteller staan? 02A
 - lus = ? (symbolentabel) 00C
 - Hoeveel is 00C – 02A?
 - $12d - 42d = -30d = -(1Eh) = E2$
- **loop lus** wordt **E2 E2**

Uitvoering van een programma

- Vertaalde versie staat op de schijf, nl.:

```
0000002D 00000064 FFFFFFFF1 00000041 0000004B
FFFFFEA FFFFFFE9 FFFFFFF9 00000014 0000000A ???????
2020202020202020202020202020202020202020202020
29 DB 29 D2 29 FF B9 0A 00 00 00 FC 8B 87 00 00 00 00 3D
00 00 00 00 7C 07 01 C3 E9 02 00 00 00 01 C2 81 C7 04 00
00 00 E2 E2 89 15 28 00 00 00 89 D8 FD BF 37 00 00 00 BB
0A 00 00 BA 00 00 00 00 F7 FB 80 CA 30 86 C2 AA 86 C2 3D
00 00 00 75 EA BF 43 00
00 00 BA 00 00 00 F7 FB 80 CA 30 86
C2 AA 86 C2 ...
```

Uitvoering van een programma

- Als het programma gestart wordt, gaat het besturingssysteem
 1. Het programma kopiëren naar het werkgeheugen
 2. De basis (van data- en codesegment) instellen
 3. Springen naar het begin van het programma

Uitvoering van een programma

- Een programma wordt zo vertaald dat het **om het even waar** in het werkgeheugen kan staan.
 - In een programma: geen adressen maar verplaatsingen.
- Andere plaats:
 - de basis heeft een andere waarde
 - de verplaatsingen blijven dezelfde



H4: Programmatuur

- Het assembleerprogramma
- **Subroutines, linker, macro's**
- Hogere vs. lagere programmeertalen
- De Java virtuele machine



H4: Programmatuur

- Subroutine's, linker, macro's
 - Waarom?
 - EIP-register, stapel
 - Call-, ret-bevel
 - Subroutines: nevenwerkingen
 - Subroutines: parameters
 - Externe subroutines
 - Systeemroutines
 - Macro's

Waarom subprogramma's?

- Grote programma's verdelen in **deelprogramma's** die samenwerken
 - Verdeel en heers
 - Meerdere programmeurs
 - Correcties
 - Herbruikbaar
- 2 soorten: **macro's** en **subroutines**

Register EIP

- **Algemeen:**
 - CPU heeft het adres van het volgend-uit-te-voeren bevel nodig
 - adres = basis + verplaatsing
- **Intel-processor:**
verplaatsing van volgend-uit-te-voeren bevel
= inhoud bevelenteller
= inhoud **EIP**

(Waar staat de basis?)

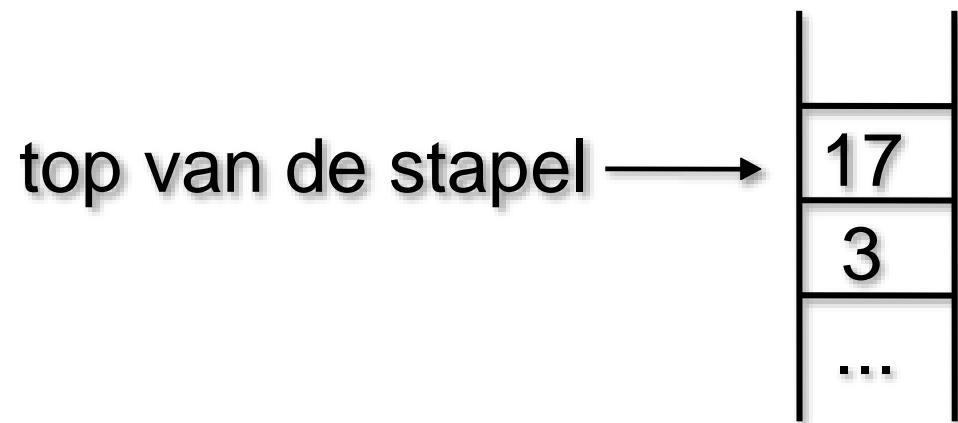
Stapelbevelen

- **Stapel:** datastructuur in het werkgeheugen
 - Data toevoegen met **push**
 - Data afhalen met **pop**

```
mov eax, 3  
push eax
```

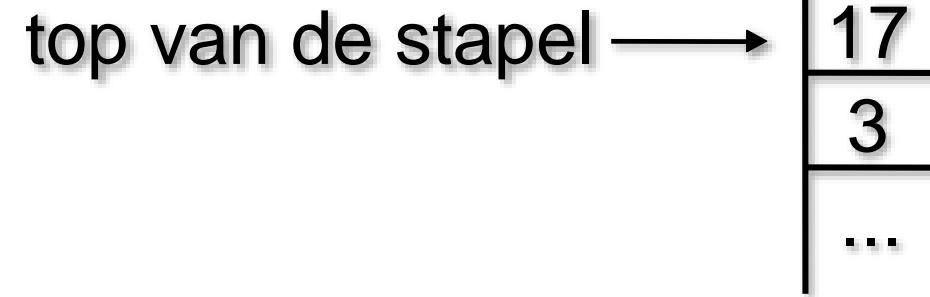


```
mov ebx, 17  
push ebx
```

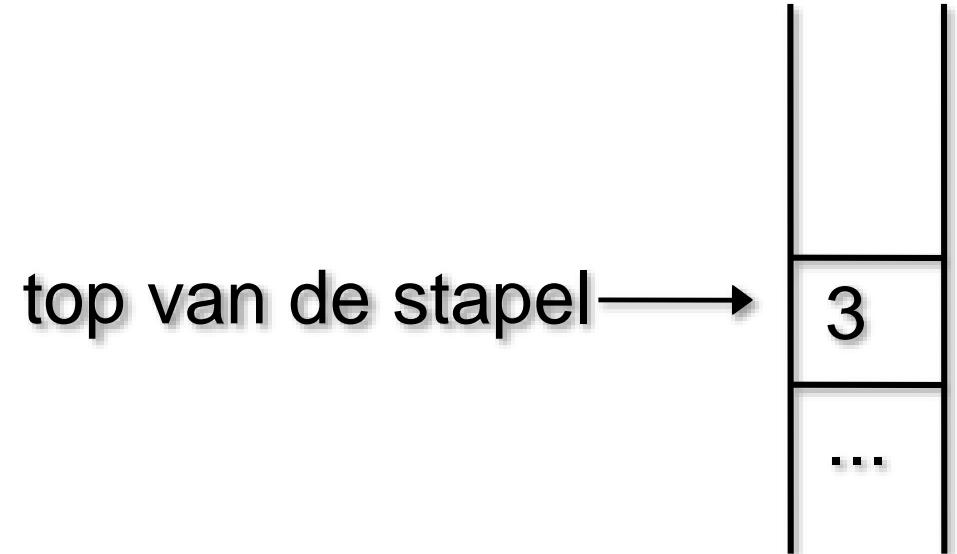


Stapelbevelen

pop ebx



EBX = 17, en:



De stapel

- **Stapel** is deel van het werkgeheugen
 - Apart segment
 - Aangeven met stack pointer = (stapelwijzer)
 - Intel-processor: bewaard in **ESP**
 - ESP = verplaatsing van de "bovenste" byte van de stapel

- Wat kan men op de stapel plaatsen?
 1. de inhoud van een register: `push ecx`
 2. een dubbelwoord: `push dword [getal]`
 3. een verplaatsing: `push dword getal`
- Wat is het verschil tussen (2) en (3)?

Push en Pop

- **push**: iets op de stapel plaatsen.
 - eerst ESP verminderen (4, 2, ?)
 - dan kopiëren (registerinhoud, ...)
- **pop**: omgekeerd.
- **ESP** = verplaatsing van de bovenste byte van de stapel.

Wat als men nog niets op de stapel gezet heeft? Er is ruimte gereserveerd voor de stapel (bvb. 1000 bytes). ESP is dan initieel de verplaatsing van de 1^e byte voorbij deze ruimte.



ADRES INHOUD

172	??
176	??
180	??
184	??
188	??
192	??
196	??
200	??

→ push dword 1234
push dword 99
pop eax
push -17

ESP:

204



ADRES INHOUD

172	??
176	??
180	??
184	??
188	??
192	??
196	??
200	1234

push dword 1234
push dword 99
pop eax
push -17

ESP:

200



ADRES INHOUD

172	??
176	??
180	??
184	??
188	??
192	??
196	99
200	1234

push dword 1234
push dword 99
pop eax
push -17

ESP:

196



ADRES INHOUD

172	??
176	??
180	??
184	??
188	??
192	??
196	99
200	1234

push dword 1234
push dword 99
pop eax
push -17

ESP:

200



ADRES INHOUD

172	??
176	??
180	??
184	??
188	??
192	??
196	-17
200	1234

push dword 1234
push dword 99
pop eax
push -17

ESP:

196

Stapel: slimme toepassing

- Snelste manier om een registerinhoud even te bewaren
 - push eax**
 - ...
 - pop eax**
- Stapel
 - push**: iets op de stapel plaatsen
 - pop**: iets van de stapel halen
 - Geen adres vereist.
 - Volgorde!

Vroeger:

hulpd: resd 1

...

mov [hulpd], eax

...

mov eax,[hulpd]



H4: Programmatuur

- Subroutine's, linker, macro's
 - Waarom?
 - EIP-register, stapel
 - Call-, ret-bevel
 - Subroutines: nevenwerkingen
 - Subroutines: parameters
 - Externe subroutines
 - Systeemroutines
 - Macro's



```
inv [haha]
mov eax, [a]
imul dword [a]
mov ebx, eax
mov eax, [b]
imul dword [b]
add ebx, eax
mov [x], eax
...
uit [x]
mov eax, [a]
imul dword [a]
mov ebx, eax
mov eax, [b]
imul dword [b]
add ebx, eax
add esi, edi
...
```

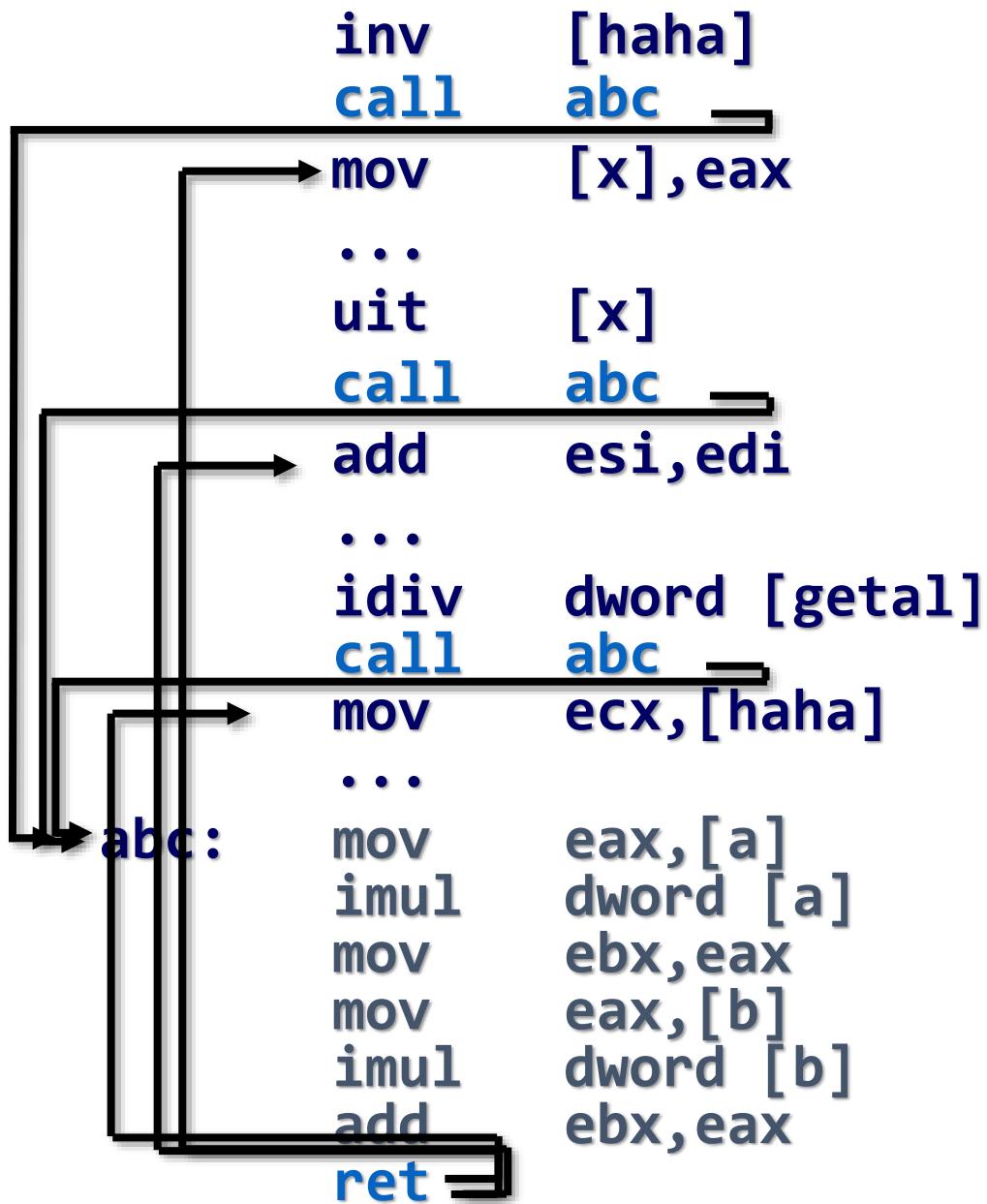
```
...
idiv dword [getal]
mov eax, [a]
imul dword [a]
mov ebx, eax
mov eax, [b]
imul dword [b]
add ebx, eax
mov ecx, [haha]
...
...
```

Wat is er mis met
dit programma?



```
inv [haha]
mov eax, [a]
imul dword [a]
mov ebx, eax
mov eax, [b]
imul dword [b]
add ebx, eax
mov [x], eax
...
uit [x]
mov eax, [a]
imul dword [a]
mov ebx, eax
mov eax, [b]
imul dword [b]
add ebx, eax
add esi, edi
```

```
...
idiv dword [getal]
mov eax, [a]
imul dword [a]
mov ebx, eax
mov eax, [b]
imul dword [b]
add ebx, eax
mov ecx, [haha]
...
```



- **Ret** = return = keer terug
 - Naar daar waar je vandaan komt.
- **Call**: meer dan jump
 - onthoud waar je bent
 - spring naar ...
 - **Terugkeeradres** = inhoud EIP = (adres van volgend bevel) wordt bewaard op de stapel.

Subroutines Nevenwerkingen

- Vb.: in een programma staat al :
rij: resd 10
g: resd 1
- Gevraagd: *schrijf een subroutine die het gemiddelde berekent en opslaat in geheugenadres g.*

Subroutines Nevenwerkingen

```
gem: mov eax, 0
     mov esi, 0
     mov ecx, 10
lus: add eax, [rij + esi]
     add esi, 4
     loop lus
     mov edx, 0
     mov ebx, 10
     idiv ebx
     mov [g], eax
     ret
```

Merk op: *de inhoud van de registers eax, ebx, ecx, edx en esi verandert!*

Subroutines Nevenwerkingen

- Wat kunnen we doen?
 1. Propere subroutine
 - alle gebruikte registers eerst wegbergen (op de stapel);
 - vóór de terugkeer herstellen.
 - Efficiënt?
 2. Oproepend programma **doet het zelf**
 - Efficiënt?
- Subroutine voorzien van goede commentaar.

Voorbeeld: a, b, c, d: adressen van dubbelwoorden.

Subroutine berekent: $\text{inh}(d) = \text{inh}(b)^2 - 4 * \text{inh}(a) * \text{inh}(c)$

```
discr:    push  eax
           push  ebx
           push  edx

           mov   eax, 4
           imul dword [a]
           imul dword [c]
           mov   ebx, eax
           mov   eax, [b]
           imul dword [b]
           sub   eax, ebx
           mov   [d], eax

           pop   edx
           pop   ebx
           pop   eax
           ret
```

Parameters

- We willen bvb.:
 - $\text{inh}(d) = \text{inh}(b)^2 - 4 * \text{inh}(a) * \text{inh}(c)$
 - $\text{inh}(dd) = \text{inh}(v)^2 - 4 * \text{inh}(u) * \text{inh}(w).$
- Voorbeeld 1:
 - Oproepend programma zet de **waarden op de stapel**
 - De subroutine werkt met de waarden die op de stapel staan
 - Waar plaatst de subroutine het resultaat? **On de stapel**

De subroutine verwijdert deze
waardes van de stapel

mma
reserveert op voorhand ruimte
voor het resultaat

Parameters

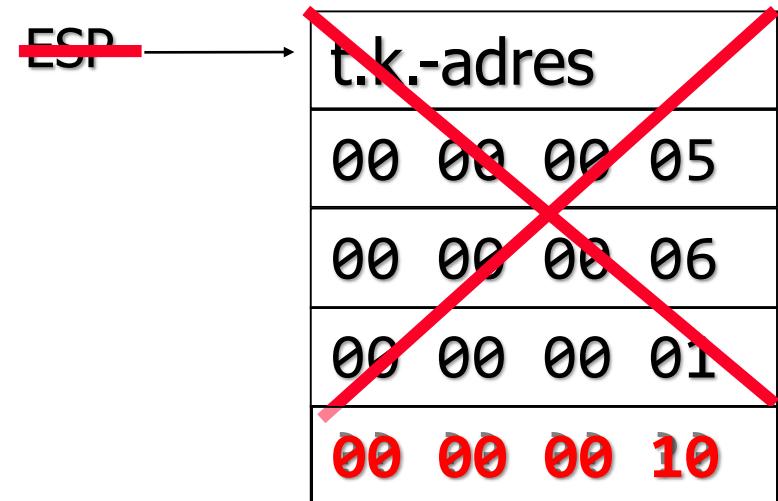
- Oproepend programma:

```
...
push dword [d]
push dword [a]
push dword [b]
push dword [c]
call descr
pop dword [d]
```

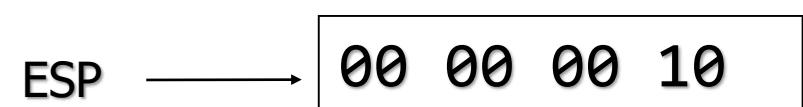
...

- De subroutine moet
 - Rekenen ($6^2 - 4 \times 1 \times 5$)
 - Resultaat op de stapel zetten
 - Andere parameters verwijderen

Begintoestand s.r.

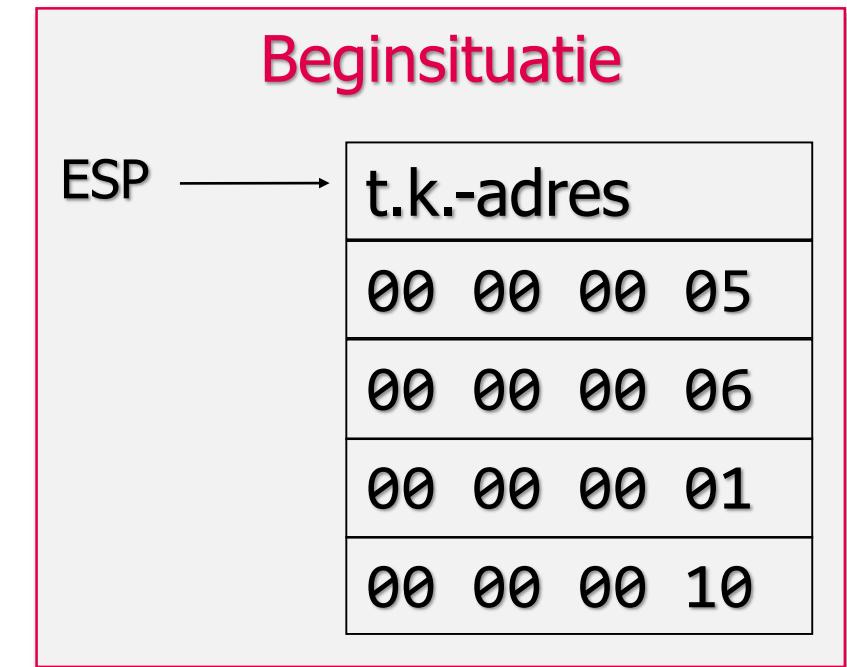
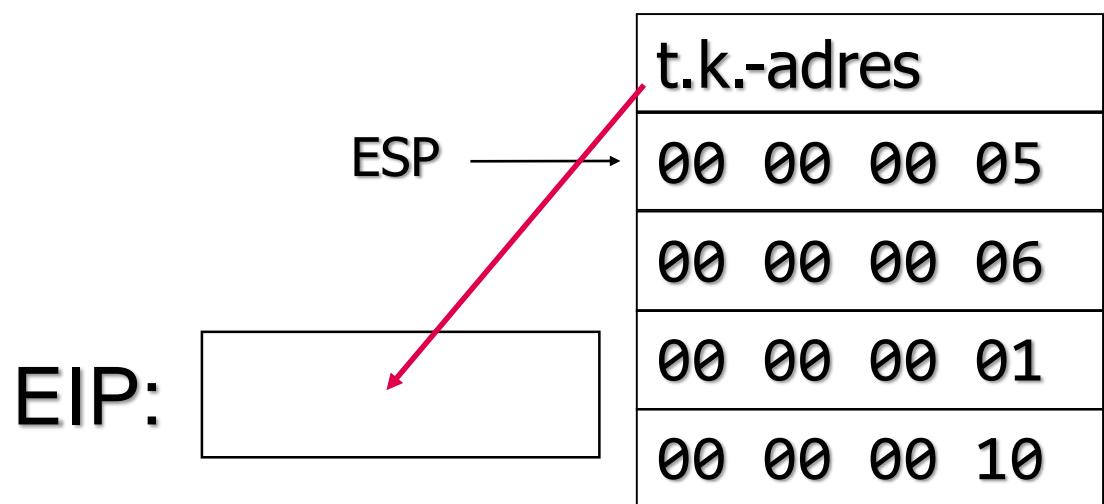


Na ret:



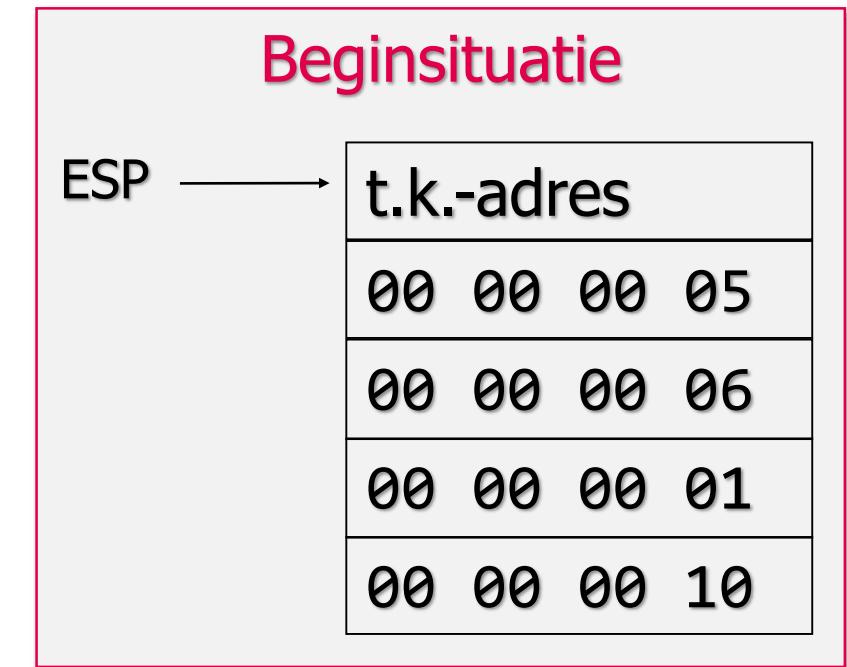
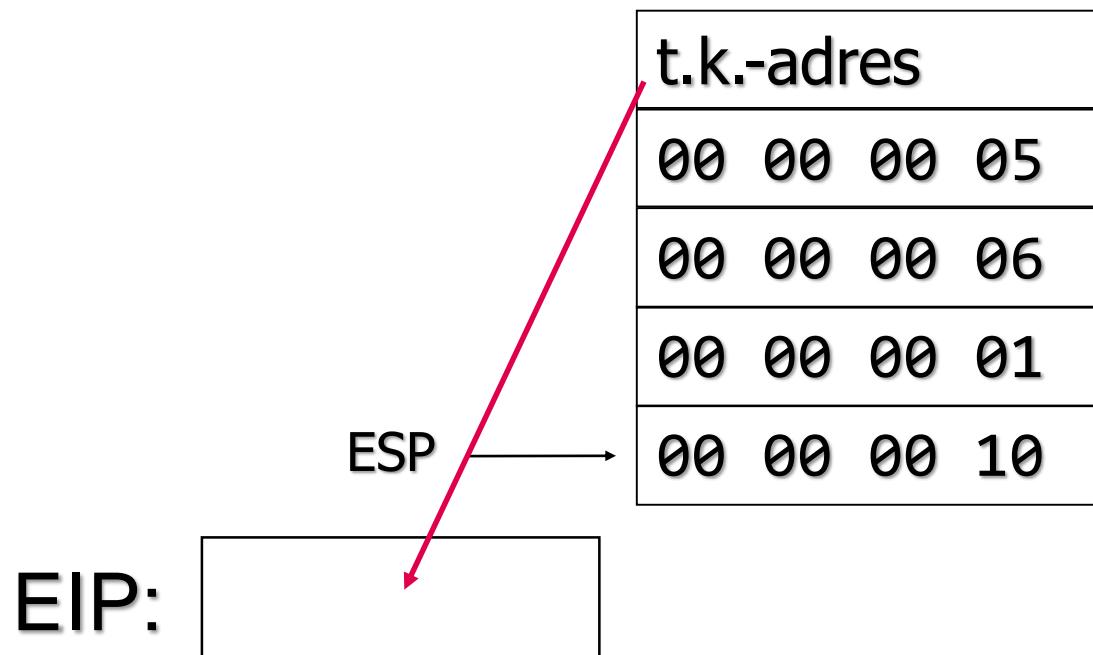
Parameters Verwijderen

- Na **ret**



Parameters Verwijderen

- Na **ret 12**

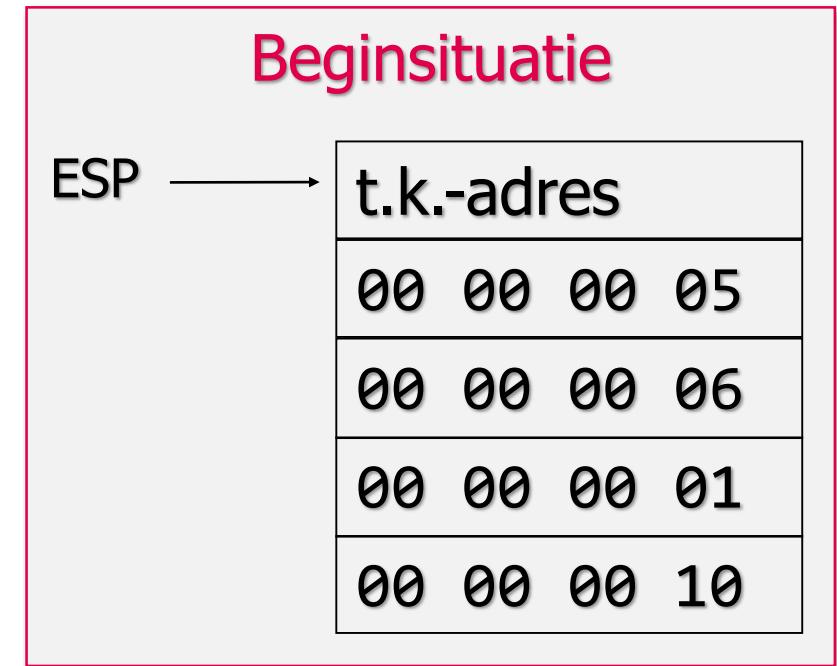


- **ret 12: zoals gewone ret én ESP + 12**
 - (gegevens gaan niet echt van de stapel af)

Parameters aanspreken

- Parameters van de stapel uitlezen kan via register ESP
 - ESP is een indexregister

; parameter ‘c’ (00000005) naar EAX
`mov eax, [esp + 4]`
- **Probleem:** ESP verandert doorheen de subroutine bij elke oproep naar pop of push



Parameters aanspreken

- **Oplossing:** kopie maken van ESP in ander (index-)register
 - register EBP dient hiervoor

```
push ebp  
mov ebp, esp
```

```
...  
; parameter 'c' (00000005) naar EAX  
mov eax, [ebp + 8]
```

```
...  
pop ebp
```

Beginsituatie	
ESP	→ t.k.-adres
	00 00 00 05
	00 00 00 06
	00 00 00 01
	00 00 00 10



Beginsituatie

ESP → t.k.-adres

00	00	00	05
00	00	00	06
00	00	00	01
??	??	??	??

ESP = 01 A2 B3 C4
EBP = 00 00 CB A0

Na push ebp
Na mov ebp, esp

ESP → 00 00 CB A0

=EBP

00	00	CB	A0
t.k.-adres			
00	00	00	05
00	00	00	06
00	00	00	01
??	??	??	??

-4

ESP = 01 A2 B3 C0

EBP = 01 A2 B3 C0

Voorbeeld 1: uitwerking

- 1. Kopie van ESP maken**
- 2. Backup maken van registers**
- 3. Parameters verwijderen**
- 4. De berekening**

push ebp
mov ebp, esp

pop ebp

Voorbeeld 1: uitwerking

- | | |
|-------------------------------|--------------|
| 1. Kopie van ESP maken | push ebp |
| 2. Backup maken van registers | mov ebp, esp |
| 3. Parameters verwijderen | push eax |
| 4. De berekening | push ebx |
| | push edx |
| | pop edx |
| | pop ebx |
| | pop eax |
| | pop ebp |

Voorbeeld 1: uitwerking

- | | |
|-------------------------------|--|
| 1. Kopie van ESP maken | push ebp |
| 2. Backup maken van registers | mov ebp, esp |
| 3. Parameters verwijderen | push eax
push ebx
push edx |
| 4. De berekening | pop edx
pop ebx
pop eax
pop ebp
ret 12 |

Voorbeeld 1: uitwerking

- | | |
|-------------------------------|--------------|
| 1. Kopie van ESP maken | push ebp |
| 2. Backup maken van registers | mov ebp, esp |
| 3. Parameters verwijderen | push eax |
| 4. De berekening | push ebx |
| | push edx |

3

pop	edx
pop	ebx
pop	eax
pop	ebp
ret	12

Voorbeeld 1: de berekening

- Waar staan de parameters? **Op de stapel.**
 - In welke volgorde? **d, a, b, c.**
-
- Adres d = ebp + 20
 - Adres a = ebp + 16
 - Adres b = ebp + 12
 - Adres c = ebp + 8

Beginsituatie

EBP	→	00 00 CB A0
		t.k.-adres
c		00 00 00 05
b		00 00 00 06
a		00 00 00 01
d		?? ?? ?? ??



```
adres d = ebp + 20  
adres a = ebp + 16  
adres b = ebp + 12  
adres c = ebp + 8
```

...

```
mov eax, 4  
imul dword [ebp + 16]  
imul dword [ebp + 8]  
mov ebx, eax  
mov eax,[ebp + 12]  
imul dword [ebp + 12]  
sub eax, ebx  
mov [ebp + 20], eax
```

...

Voorbeeld 1: de aanroep

- Hoe doen we $\text{inh}(\text{dd}) = \text{inh}(\text{v})^2 - 4 * \text{inh}(\text{u}) * \text{inh}(\text{w})$?
- Zo:
 - push [dd]
 - push [u]
 - push [v]
 - push [w]
 - call descr
 - pop [dd]

ByRef vs. ByVal

- Vorige oplossing: **call by value**
 - Waardes van parameters worden op de stapel gezet
 - Probleem: Wat als er veel parameters zijn (bijv. een lijst van getallen)?
- Oplossing? **Call by reference.**
 - Het adres van parameters op de staple zetten

Voorbeeld 2

- "Schrijf een subroutine die het gemiddelde berekent van een lijst van getallen"
 - We hebben:

...

```
csys: resd 50      ; 50 uitslagen csys
gcsys: resd 1
bop: resd 48       ;48 uitslagen bop
gbop: resd 1
```

...

- We willen

```
inhoud[gcsys] = gemiddelde[0]
inhoud[gbop] = gemiddelde[1]
```

Onderstel: adres csys = 12 AB DD CC;
d.i. de uitslagen van csys staan op
adressen 12 AB DD CC, 12 AB DD D0, ...

ByRef Parameter

- Oproepend programma:

```
...
push dword [gcsys]
push dword 50
push dword csys
call gemidd
pop dword [gcsys]
```

...

- Verschil met het vorige: op de stapel staat het **beginadres** van de lijst van csys-getallen.
 - Dit heet: **call by reference**

Begintoestand s.r.

ESP →	t.k.-adres
	12 AB DD CC
	00 00 00 32
	?? ?? ?? ??

Voorbeeld 2: uitwerking

- 1. Kopie van ESP maken**
- 2. Backup maken van registers**
- 3. Parameters verwijderen**
- 4. De berekening**

push ebp
mov ebp, esp

pop ebp

Voorbeeld 2: uitwerking

1. Kopie van ESP maken
 2. Backup maken van registers
 3. Parameters verwijderen
 4. De berekening

```
push ebp  
mov ebp, esp
```

push eax
push ecx
push edx
push esi

pop	esi
pop	edx
pop	ecx
pop	eax
pop	ebp

Voorbeeld 2: uitwerking

1. Kopie van ESP maken
 2. Backup maken van registers
 3. Parameters verwijderen
 4. De berekening

```
push ebp  
mov ebp, esp  
  
push eax  
push ecx  
push edx  
push esi
```

pop	esi
pop	edx
pop	ecx
pop	eax
pop	ebp
ret	8

Voorbeeld 2: uitwerking

1. Kopie van ESP maken
2. Backup maken van registers
3. Parameters verwijderen
4. De berekening

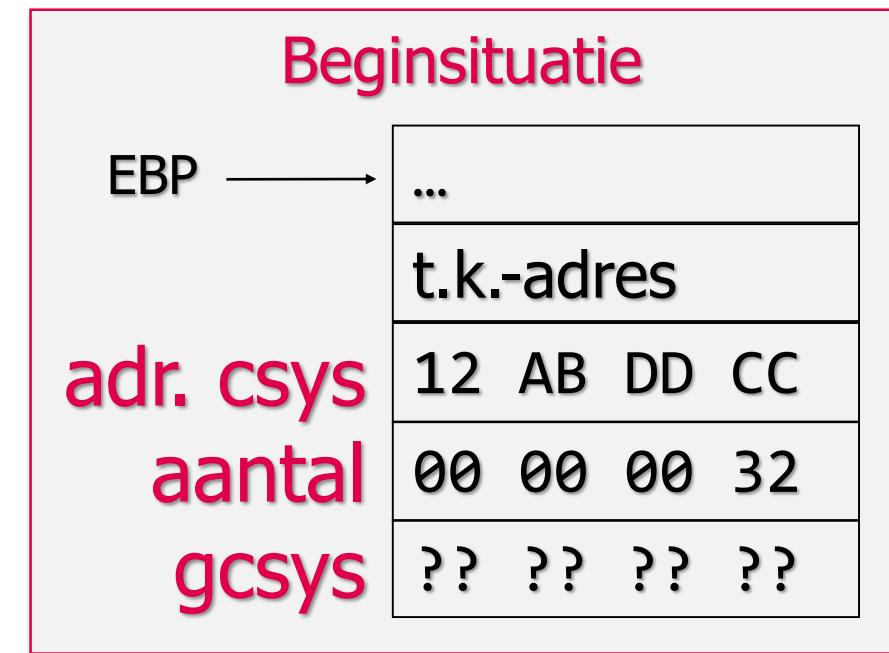
```
push ebp  
mov ebp, esp  
push eax  
push ecx  
push edx  
push esi
```

...

```
pop esi  
pop edx  
pop ecx  
pop eax  
pop ebp  
ret 8
```

Voorbeeld 2: de berekening

- Waar staan de parameters? **Op de stapel.**
 - In welke volgorde?
-
- Adres gcsys = ebp + 16
 - Adres aantal = ebp + 12
 - Adres (adres csys) = ebp + 8





```
Adres gcsys = ebp + 16  
Adres aantal = ebp + 12  
Adres (adres csys) = ebp + 8
```

...

mov ecx, [ebp + 12]

mov esi, [ebp + 8]

mov eax, 0

lus: add eax, [esi]

add esi, 4

loop lus

mov edx, 1

imul edx

idiv dword [ebp + 12]

mov [ebp + 16], eax

...



H4: Programmatuur

- Subroutine's, linker, macro's
 - Waarom?
 - EIP-register, stapel
 - Call-, ret-bevel
 - Subroutines: nevenwerkingen
 - Subroutines: parameters
 - **Externe subroutines**
 - **Systeemroutines**
 - **Macro's**

Externe Subroutines

- Hoe kunnen we subroutines van het ene programma in het andere gebruiken?
 - Vb. in hulppr.asm
 - `global testsr`
 - ...
 - `testsr:` ...
 - ...
 - `ret`
 - Resultaat: ~~testsr: undefined symbol~~
- in hoofdpr.asm
 - `extern testsr`
 - ...
 - `call testsr`

Beide programma's worden apart vertaald; global en extern zijn directieven voor de linker.

Assembler en Linker

- De **assembler** converteert een .asm-bestand naar een objectbestand
 - Elk .asm-bestand kan bestaan uit een datasegment en een codesegment
- De **linker** plakt verschillende objectbestanden samen en genereert een executable.

Externe Subroutines

- Vb. in hulppr.asm

DATA

```
twee: dd 2
```

CODE

```
...
```

```
testsr:
```

```
...
```

```
ret
```

- in hoofdpr.asm

DATA

```
rij: resd 12
```

```
h: resb 'Hallo'
```

```
getal: resd 1
```

```
...
```

CODE

```
...
```

```
call testsr
```

```
uit [getal]
```

```
...
```

DATA

```
twee: dd 2
```

CODE

```
...
testsr:
...
ret
```

DATA

```
rij: resd 12
h: resb 'Hallo'
getal: resd 1
...
twee: dd 2
```

CODE

```
...
call testsr
uit [getal]
...
testsr:
...
ret
```

DATA

```
rij: resd 12
h: resb 'Hallo'
getal: resd 1
...
```

CODE

```
...
call testsr
uit [getal]
...
```

Systeemroutines

- Toegang tot randapparatuur (scherm, toetsenbord, bestand, ...) moet via besturingssysteem
 - Via subroutines
 - Maken deel uit van het besturingssysteem
 - Windows: o.a. kernel32.dll
- De linker linkt je programma met o.a. kernel32.dll

Systeemroutines oproepen

1. De routine als extern declareren
2. Eventueel de nodige parameters op de stapel zetten
3. call ...

```
extern ExitProcess
```

```
...
```

```
call ExitProcess
```

- Een **macro** is een afkorting van een aantal programmalijnen

```
%macro wissen 0 ; macro-prototype
    cld
    mov edi, outarea
    mov ecx, 70
    mov al, ' '
    rep stosb
%endmacro
```

Macro's oproepen

...

sub ebx, ebx
wissen

mov eax, haha

...

loop lus
wissen

mov ecx, 56

...

wissen

...

Het macro-blok wordt
tussengevoegd.



H4: Programmatuur

- Het assembleerprogramma
- Subroutines, linker, macro's
- Hogere vs. lagere programmeertalen
- De Java virtuele machine

Soorten programmeertalen

- **Inwendige machinetaal**
 - Hexadecimale (binaire) instructies
 - Geen symbolische adressen

```
...
29 05 14 01 00 00
F7 2D 14 01 00 00
75 BD
B8 23 00 00 00
...
```

Soorten programmeertalen

- **Uitwendige machinetaal**
 - Mnemotechnische functiecodes
 - Geen symbolische adressen

```
...  
29 05 14 01 00 00  
F7 2D 14 01 00 00  
75 BD  
B8 23 00 00 00  
...
```

```
...  
sub [114h], eax  
imul dword [114h]  
jnz FFFFFFFBDh  
mov eax, 35  
...
```

Soorten programmeertalen

- **Lagere Programmeertaal**
 - Mnemotechnische functiecodes
 - Wél symbolische adressen
 - 1 bevel = 1 processorinstructie
 - Wordt geassembleerd door een assembleerprogramma (E: assembler)
 - Ook: **assembleertaal**

```
...  
29 05 14 01 00 00  
F7 2D 14 01 00 00  
75 BD  
B8 23 00 00 00  
...
```

```
...  
sub [114h], eax  
imul dword [114h]  
jnz FFFFFFFBDh  
mov eax, 35  
...
```

```
...  
sub [a4], eax  
imul dword [a4]  
jnz opnieuw  
mov eax, 35  
...
```

Soorten programmeertalen

- **Hogere Programmeertaal**
 - 1 bevel = 1 of meerdere instructies
 - Wordt gecompileerd door een compilator (E: compiler)

```
...
a4 = a4 - val;
val = a4 * val;
if (a4 == 0) {
    val = 35;
...
}
```

```
...
29 05 14 01 00 00
F7 2D 14 01 00 00
75 BD
B8 23 00 00 00
...
```

```
...
sub [114h], eax
imul dword [114h]
jnz FFFFFFFBDh
mov eax, 35
...
```

```
...
sub [a4], eax
imul dword [a4]
jnz opnieuw
mov eax, 35
...
```

Soorten programmeertalen

- Vb. 1: in een hogere programmeertaal staat:

```
int[] temperatuur = new int [20];
```

...

```
temperatuur[5] = 17;
```

- Dit wordt door een compiler vertaald als:

```
mov eax, 5
imul dword [vier]      ; (vier: dd 4)
mov esi, eax
mov eax, 17
mov [temperatuur + esi], eax
```

Soorten programmeertalen

- Vb. 2: in een hogere programmeertaal staat:

```
x = a + b;
```

```
y = x * c;
```

- Dit wordt door een compiler vertaald als:

```
mov eax, [A]
```

```
add eax, [B]
```

```
mov [X], eax
```

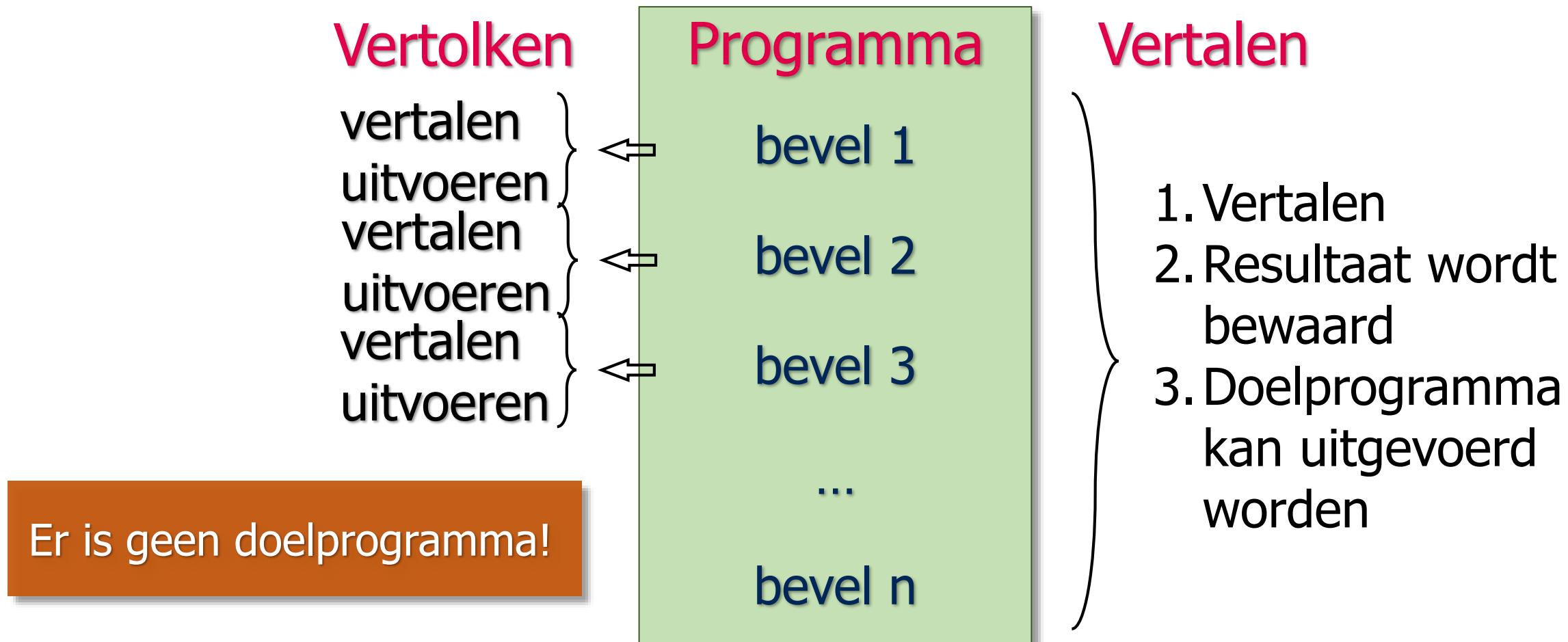
~~```
mov eax, [X]
```~~

```
imul dword [C]
```

```
mov [Y], eax
```

Een slimme compiler kan dit ook!

# Vertalen en Vertolken



# Vertalen en Vertolken

- Vertolken gebruikt **minder geheugen**
- Vertolker moet telkens **opnieuw vertalen**
  - Bij iedere uitvoering van het programma
  - Maar bvb. ook bij:  
doe 100 keer  
 $\{X = (A + B) / C;\}$   
Elk bevel van de lus wordt dus telkens opnieuw vertaald

# Microprogrammatie

- **Observatie:** Sommige instructies doen deels hetzelfde werk
  - **add, sub, ...:** ophalen van een dubbelwoord uit het werkgeheugen
  - **add, sub, cmp:** aanpassen van de vlaggen
  - **lodsb, stosb, movsb:** aanpassen van **esi** en/of **edi**

# Microprogrammatie

- Een bevel kan opgedeeld worden in meer elementaire bevelen: **micro-bevelen**

0305 2C000000  
(**add eax, [dw]**)

- Adres berekenen: basis + 0000002C
- Adres naar werkgeheugen sturen
- Lees-bevel naar werkgeheugen
- Inhoud eax kopiëren naar rekeneenheid
- Dubbelwoord naar rekeneenheid
- Optelbevel
- Resultaat naar eax

2B05 2C000000  
(**sub eax, [dw]**)

- Adres berekenen: basis + 0000002C
- Adres naar werkgeheugen sturen
- Lees-bevel naar werkgeheugen
- Inhoud eax kopiëren naar rekeneenheid
- Dubbelwoord naar rekeneenheid
- Aftrekbevel
- Resultaat naar eax

# Microprogrammatie

- Voor ieder bevel is er een lijstje met uit te voeren microbevelen: het **microprogramma**
  - Machinebevel uitvoeren = microprogramma uitvoeren
  - Microprogramma's bewaard in microgeheugen

# Microprogrammatie

- 1 bevel uit hogere programmeertaal wordt vervangen door meerdere machinebevelen
  - Gebeurt op voorhand door de compiler
- 1 machinebevel wordt vervangen door meerdere microbevelen
  - Gebeurt tijdens de uitvoering van het programma
  - de CPU **interpreteert** dus de machinebevelen

# RISC en CISC

- **CISC: Complex Instruction Set Computer**
  - Veel verschillende en krachtige machinebevelen
  - vb.: Intel x86 processoren
- **RISC: Reduced Instruction Set Computer :**
  - Beperkte set van machinebevelen
  - Bevelen zijn minder krachtig
  - Geen microbevelen
  - Meer registers, minder verbruik
  - “John Cocke of IBM Research originated the RISC concept in 1974 by proving that about 20% of the instructions in a computer did 80% of the work.”



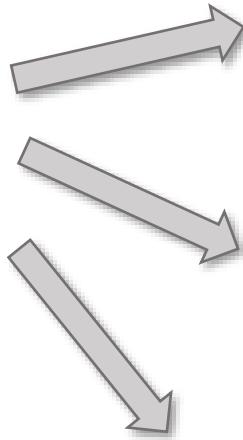
## H4: Programmatuur

- Het assembleerprogramma
- Subroutines, linker, macro's
- Hogere vs. lagere programmeertalen
- De Java virtuele machine

# Hogere Programmeertaal

```
int main() {
 exit();
}
```

C



```
start:
 call ExitProcess
```

x86/Win

```
_start:
 mov eax, 1
 int 80h
```

x86/Lin

```
_start:
 mov %r0, $0
 mov %r7, $1
 swi $0
```

ARM/Lin

# Hogere Programmeertaal

```
int main() {
 exit();
}
```

C

```
start:
 call ExitProcess
```

x86/Win

```
_start:
 mov eax, 1
 int 80h
```

x86/Lin

Compileruitvoer is afhankelijk van het type processor en het besturingssysteem.

```
_start:
 mov %r0, $0
 mov %r7, $1
 swi $0
```

ARM/Lin

# JVM: Java Virtuele Machine

- Java heeft een andere aanpak
- De **Java Virtuele Machine** is een virtuele processor
  - Bestaat niet echt
  - Heeft zijn eigen machinetaal: **bytecode**
- Java-code wordt gecompileerd naar bytecode
  - De bytecode van een Java-programma is hetzelfde voor elk apparaat dat het programma moet uitvoeren

# JVM: Java Virtuele Machine

- Wat is er nodig om een bytecode-programma uit te voeren op een bepaald apparaat? **Een vertolker.**
  - Elk bytecode-bevel omzetten naar machinebevelen van het apparaat
  - Deze machinebevelen uitvoeren
- Deze vertolkers (één per CPU/OS) worden ook Java Virtuele Machines genoemd

# De JVM-processor

- Bevelenteller, bevelregister
- Geen andere registers
- Gebruik van de stapel voor "alles", bvb.: optelling van 2 getallen:
  1. Zet 1<sup>e</sup> getal op de stapel
  2. Zet 2<sup>e</sup> getal op de stapel
  3. Tel op (hierdoor worden de 2 bovenste getallen van de stapel gehaald en opgeteld)
  4. Resultaat (de som) komt op de stapel

# Is bytecode efficient?

- JIT-compilatie: **Just In Time**
  - Na interpretatie wordt de vertaalde versie bewaard voor een volgende uitvoering
  - Efficiëntere uitvoering van o.a. lussen