Controletransferinstructies

= Instructies die verandering brengen in de zuiver sequentiële uitvoering van de instructies. Ze doen dit door de instructiewijzer een andere waarde te geven. Ze kunnen met andere woorden worden gezien als mov-instructies naar **eip**.

Sprongen

1/ Onvoorwaardelijke sprongen

```
jmp   adres
```

reg[eip] = adres

2/ Voorwaardelijke sprongen

instructie	sprong
jz	jump if zero
jc	jump if carry
jo	jump if overflow
jp	jump if parity
js	jump if sign
jnz	jump if not zero
jnc	jump if not carry
jno	jump if not overflow
jnp	jump if not parity
jns	jump if not sign

instructie		sprong	
jg	jnle	jump if greater	2-complement
jge	jnl	jump if greater or equal	2-complement
jl	jnge	jump if less	2-complement
jle	jng	jump if less or equal	2-complement
je		jump if equal	
ja	jnbe	jump if above	binair
jae	jnb	jump if above or equal	binair
jb	jnae	jump if below	binair
jbe	jna	jump if below or equal	binair

Springt slechts indien aan een gestelde voorwaarde is voldaan. (bijvoorbeeld een **if** instructie)
We kunnen verloop van een programma voorstellen door een **controleverloopgraaf** met doorvalpaden al of niet. Een controleverloopgraaf is een aaneenschakeling van een aantal basisblokken.
De linkse spronginstructies testen de bits in het EFLAGS register, terwijl de rechtse een vergelijking maakt tussen twee getallen. De vergelijking gebeurt, zoals eerder gezegd, door de twee getallen van elkaar af te trekken zodat de ALU de toestandbits zet.

3/ Berekende sprongen

Een **statische sprong** springt naar een vast adres, terwijl een **berekende sprong** springt naar een adres welke gegeven wordt door een register/geheugeninhoud. (voorbeeld van gebruik bij **switch**)

```
mov ebx,100
jmp  ebx
```

reg[eip] = reg[ebx]

Switch 1
Geneste if-lussen! Als het aantal gevallen echter groot is, dan neemt de gemiddelde tijd nodig om een geval te vinden lineair toe met het aantal gevallen in de switch instructie. Een oplossing is dan om in een binaire boom te zoeken → logaritmisch.

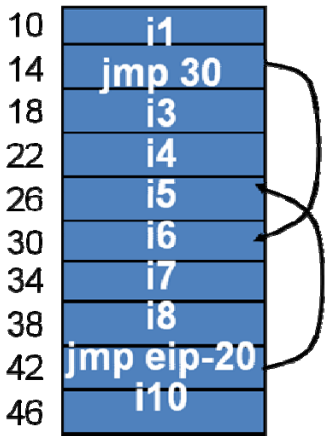
Switch 2
Berekende sprong in combinatie met een adrestabel.
Bvb mov eax, [tabel+ebx*4]
Eender welk geval kan nu in een constante tijd worden berekend. De extra kost bestaat uit de plaats nodig om de adressen in op te slaan, de controle op de geldige index, en de kost om het adres uit de tabel te halen.

```
tabel:  .long   $22
        .long   $25
        .long   $23
        .long   $24
```

4/ Absoluut versus relatief adres

- Absoluut
  - spring naar adres n
- Relatief
  - spring n bytes verder/terug

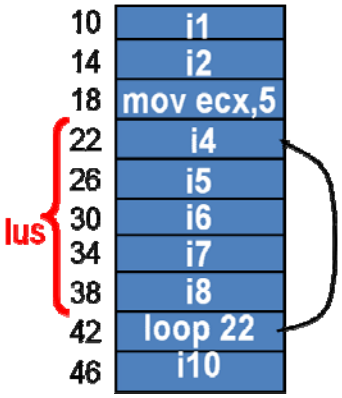
reg[eip] = reg[eip] + n



Relatieve sprongadressen zijn beter, omdat de sprong zo onafhankelijk wordt van wijzigingen elders in de code. (=**positie-onafhankelijke code**). Als men echter instructies invoegt tussen spronginstructie en doeladres, moet men de sprongafstand corrigeren!

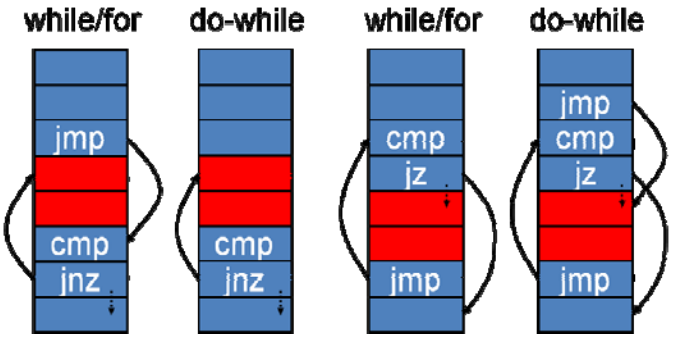
Lussen

1/ Loop
In de praktijk wordt de instructie **loop adres** niet meer gebruikt. Deze instructie decrementeert ecx en springt naar het opgegeven adres indien ecx != 0. Het gebruik ervan onderstelt dat ecx vooraf goed werd ingesteld.



2/ Voorwaardelijke sprong = geprogrammeerde lus
Men kan zelf de loopinstructie nabootsen door twee commando's:
sub ecx, 1
jnz xx

3/ Lusimplementaties



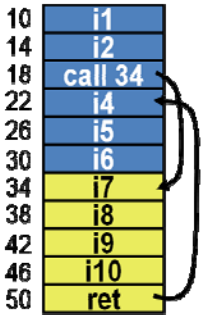
Indien het aantal iteraties vast is, en tijdens compileren gekend, dan spreekt men van een **manifeste lus**.

Procedureoproep & terugkeer

1/ Functie oproep

```
call adres
    reg[esp] = reg[esp]-4
    mem[reg[esp]] = reg[eip]
    reg[eip] = adres

ret
    reg[eip] = mem[reg[esp]];
    reg[esp] = reg[esp] + 4
```



Call en ret slaan/halen het returnadres op/van de stack op/af!

In plaats van het returnadres op te slaan op de stack, kan het ook opgeslagen worden in een register. Dit register wordt dan een **linkregister** genoemd. Dit is efficiënter, maar het werkt eigenlijk enkel maar goed in **bladroutines** (functies die geen andere functies meer oproepen).

```
call r,adres        bal  r, adres
    reg[r] = reg[eip]
    reg[eip] = adres

jmp r
    reg[eip] = reg[r]
```

Voorbeeld: vijfvoud. Zie slides 24-32 voor de code

2/ Parameterdoorgave via de stapel

```
vijfvoud:
    mov eax,[esp+4]
    cmp eax,0
    jg    positief
    xor   eax, eax
    ret
positief:
    mov ebx, 5
    imul ebx
    ret

main:
    push 6
    call  vijfvoud
    add  esp,4
    mov g, eax
```

De add esp,4 zorgt ervoor dat de plaats die ingenomen werd op de stapel door "6" weer wordt vrijgegeven. In sommige progammeertalen laat men echter de argumenten (hier de waarde 6) opruimen door de functie zelf door de **ret** instructie. Dit is echter alleen mogelijk in talen waarbij de functie het aantal argumenten kent.

```
vijfvoud:
    mov eax,[esp+4]
    cmp eax,0
    jg    positief
    xor   eax, eax
    ret   4
positief:
    mov ebx, 5
    imul ebx
    ret   4

main:
    push 6
    call  vijfvoud
    add  esp,4
    mov g, eax
```

3/ Lokale veranderlijken (in subroutines)

Worden meestal op de stapel bijgehouden door esp eentje naar boven te verschuiven en dan de temporele waarde op die plaats op te slaan... Bvb **mov [esp],eax**

4/ Uitgewerkt voorbeeld: maximum van 4 getallen

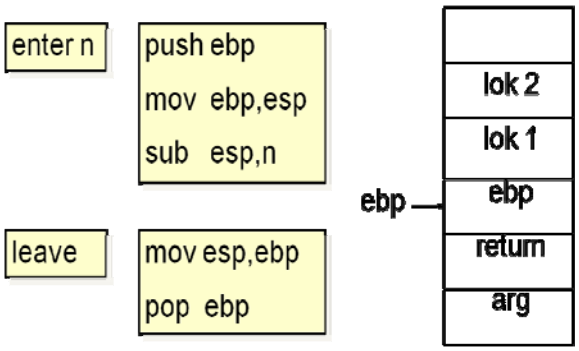
Hier wordt gewerkt met **stack frame pointers** ebp. Het is de bedoeling dat ebp nu als stack pointer wordt gebruikt voor een nieuwe functie. Dit wordt goedgezet door de volgende instructies:

push ebp
mov ebp,esp

Een stack frame breek je (natuurlijk) terug af door

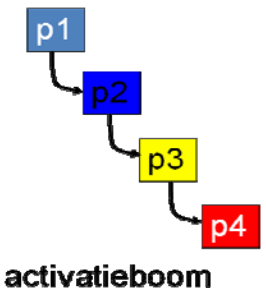
mov esp,ebp
pop ebp.

Er bestaan echter instructies die hetzelfde doen:



De "n" zal een aantal plaatsen voorzien voor lokale veranderlijken. Leave zal de esp naar de vorige stack frame pointer doen wijzen.

De stack frame pointers zullen in feiten alle stack frames aan elkaar linken (=**dynamische links**). De bijhorende stack wordt ook wel nog **call stack** genoemd.
Een **activatieboom** geeft een grafische weergave van hoe de verschillende functies elkaar hebben aangeroepen.



activatieboom

5/ Oproepconventies
Tussen oproepende code en subroutine code moeten duidelijke afspraken bestaan over de interface van beide stukken code: **oproepconventies**, welke bepalen:

- hoe de argumenten worden doorgegeven: via de stapel (in welke volgorde) of via registers (welke registers?)
- welk register is stapelwijzer?
- of registers worden bewaard door oproeper of oproepeling
- hoe het terugkeeradres wordt doorgegeven (stapel of register)

6/ Optimalisatie
Kopiepropagatie
Wanneer een waarde wordt berekend in register r1, om dan later gekopieerd te worden in r2, kan ze evengoed onmiddellijk in r2 worden berekend.

Dode waarden
Waarden die niets doen

Idempotente code
Een instructie die een resultaat berekent welke reeds bestond. Bijvoorbeeld push ebp en pop ebp, terwijl er met ebp niets wordt gedaan.

Sprong naar doorvalpad
Soms kan na optimalisatie worden gesprongen naar een doorvalpad. Dit kan zonder meer weggelaten worden.

Functiesubstitutie of inlining
Het vervangen van een functieoproep door de functiedefinitie zelf. Dus in plaats van **call functie**, zet je op deze plaats gewoon de instructies die deze functie bevat.

Onderbrekingen

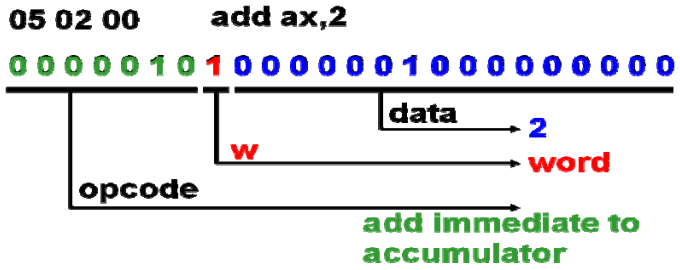
Onderbrekingen zijn eigenlijk een soort subroutines. Ze kunnen worden aangeroepen door de programmeur, maar ook door externe onderbrekingen door randapparatuur of CVE om een bepaalde situatie op te lossen. Het doeladres wordt via een **geheugenindirectie** verkregen. Aan elke onderbreking wordt een nummer toegekend, en het doeladres wordt gevonden in de **vectortabel** waarbij het nummer als index wordt gebruikt.

Stel dat tijdens een instructie een onderbreking wordt gegenereerd (int 3). Dit kan b.v. een onderbreking zijn die verband houdt met de instructie (divide by zero) of extern is (onderbreking om een I/O signaal, zoals muisbeweging, te behandelen). Als gevolg daarvan wordt gesprongen naar de subroutine waarnaar cel 3 in de vectortabel wijst. Na afloop van de onderbreking keert de uitvoering terug naar de volgende instructie en wordt de normale uitvoering voortgezet. Je kan ook expliciet het nummer van de onderbreking oproepen.
Als je nu zelf wil bepalen wat er moet gebeuren bij een welbepaalde onderbreking, zal je de vectortabel moeten overschrijven en het adres horende bij de onderbreking (stel int 2) te laten verwijzen naar de door ons gekozen subroutine in het geheugen.

Systeemoperaties zijn instructies die te maken hebben met de controle van de machine zelf en dus zeer machine specifiek. Ze bepalen o.a. instructies voor **manipulatie van de processortoestand**: onderbrekingen aan/uit, bswap,...

Instructiecodering

De instructiecodering is het finaal vertalen van de assemblercode naar binaire code. Dit gebeurt normaal door een assemblerprogramma.



De eerste 7 bits zijn de **opcode**, ofwel de opdracht die de instructie dient uit te voeren. Hier geeft ze aan dat er een constante moet worden opgeteld bij de accumulator. De 8ste bit geeft aan dat het om een woord (16 bit) gaat. De constante 2 die volgt in 2 byte (16 bit) is in little endian.

Het **databoek** bij een processor zal elke instructie in detail beschrijven. SUB zegt dat het om een subtraction gaat, 001010 is de opcode, en O D I T S Z A P C geeft aan welke vlaggen zullen worden gebruikt in het EFLAGS register.

Compilers, linkers en laders

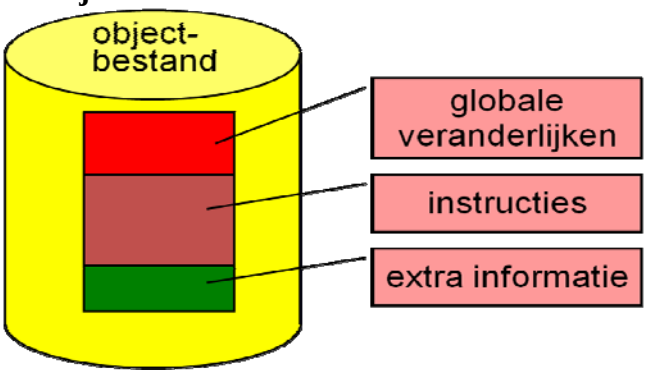
1/ Programmaontwikkeling
Het proces waarbij broncode wordt omgezet tot een uitvoerbaar bestand voor een bepaald platform (=programmaontwikkeling) bestaat uit een aantal stappen:

- **compiler** zet de broncode om in een **objectbestand**: vertaling naar assemblercode, en uiteindelijk naar binaire machinecode.
- men splitst een programma meestal op in meerdere broncodebestanden → meerdere objectbestanden. De **linker** zal alle objectbestanden samenbrengen en voegt er eventueel nog routines uit bibliotheken aan toe, om zo een uitvoerbaar bestand te maken. Een bibliotheek is een verzameling van gecompileerde routines om specifieke functionaliteit mogelijk te maken (I/O activiteiten, ...)

2/ Compiler
Het proces hierdoor uitgevoerd wordt opgesplitst in verschillende stappen:

- **lexicale analyse**: vrije tekst wordt geïnterpreteerd en omgezet tot elementaire **lexemen** van de taal. a, 12, then, while, (
- **syntactische analyse**: interpreteren van sequenties lexemen. if (a<b) then..
- **semantische analyse**: controle indien alle symboolnamen gedeclareerd zijn, en alle datatypes van alle operandi wordt gecontroleerd
- **optimalisatie**: programma optimaliseren
- **codegeneratiestap**: omzetting naar assembler
- **scheduling**: volgorde van instructies, berekeningen en geheugentoegangen veranderen om het resulterende programma nog sneller uit te voeren.

3/ Objectbestand



Bestaat uit 3 delen:

- **globale veranderlijken**
- **instructies**: binaire programmacode zelf
- **extra info**: info mbt het objectbestand nodig in de linker stap en tijdens uitvoering. (bvb naam, grootte, adres waar hoofdprog begint, ...)

4/ Linker

Verschillende objectbestanden samenvoegen tot een uitvoerbaar bestand. Het brengt dus allerlei info samen, en lost externe wijzigingen op. Met andere woorden, wanneer een bepaald objectbestand een externe routine of veranderlijke gebruikt, gaat de linker op zoek naar een uniek voorkomen van deze globale routine of veranderlijke in een lijst van andere objectbestanden. Ook wordt er nog aandacht besteedt aan **herloceerbare** adressen = adressen die nog geen finale waarde hebben gekregen en tijdens het linken nog moeten worden aangepast.

5/ Lader

Het uitvoerbare bestand wordt nu door de lader ingeladen in het geheugen. De verschillende geheugensegmenten worden opgeladen (veranderlijken,...) en bepaalde delen van het geheugen worden gereserveerd en registers worden klaargezet. De lader moet nu de finale adresaanpassingen doen. Welke adressen moeten worden veranderd, werd tijdens het compileren en linken gecreëerd en bewaard in het uitvoerbare bestand.
Het **grabbelgeheugen** is een deel van het geheugen welke beschikbaar is om dynamisch objecten te alloceren.