Hogeschool van Amsterdam

System and Middleware Programming

C en het compileer proces

LES 2 – compileer opties & stack gebruik.

Docenten :
Frans Schippers (f.h.schippers@hva.nl)
Emeri Koenen (e.p.koenen@hva.nl)

Cyber security – jaar 1 blok 4

Lesopbouw [C – deel]

Hogeschool van Amsterdam

De leerdoelen van dit onderdeel v/d cursus zijn :

1. **De student herkent en begrijpt :**
   a) De wijze waarop programma's kunnen worden uitgevoerd
   b) Het proces van compileren [van source-code naar executable-code]
   c) Structuur van een executable bestand [incl. principe van virtueel geheugen]
   d) Functie aanroep [principe en gebruik van de stack]

2. **De student is in staat :**
   a) Eenvoudige Assembly instructies 'uit te voeren'.
   b) Algemene structuren in disassembled code te duiden
   c) Nut van analyse tools te beschrijven en de resultaten correct te interpreteren

We zullen ons in deze cursus beperken tot programma's en tools op het Linux platform.
(voor Windows platform zijn alle tools en details anders , maar de principes blijven gelijk !)

Lesopbouw [C – deel]

Hogeschool van Amsterdam

Dit deel van de cursus bestaat uit 3 lessen :

1. Typen programmeer talen
   Introductie : compileren & linken , ELF
   gereedschap : gcc (gnu c-compiler), file, hexeditor
2. Compileer proces
   compileer en link stappen
   virtueel geheugen, proces start, details van de ELF file structuur
   Functies en stack gebruik
   Assembly code (introductie)
   gereedschap : readelf, enkele utilities, gdb (gnu debugger)
3. Assembly code (vervolg)
   opcodes, condities en loops [programma structuren → ISA structuren]
   registers en adressering
   gereedschap: GHIDRA

Bestanden : van source naar exe

Hogeschool van Amsterdam

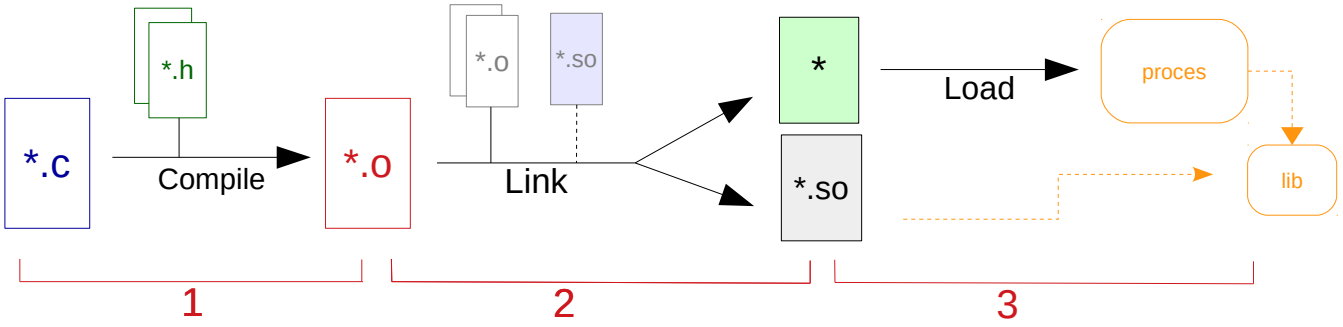
Een aantal verschillende bestandstypen spelen een rol bij het maken van een executable bestand.

- *.c source file. Bevat de code en includes van header files.
- *.h header files. Bevat definities van functies. [in 'bijbehorende' libraries]
- *.o object files. Bevat de gecompileerde instructies.
- *.so shared object. Dit is een library met gecompileerde code van 'aanroepbare' functies.
  Deze kunnen niet als proces gestart worden, maar wel geladen worden in memory
  (door het OS).
- * executables. Deze hebben (in principe) geen extensie.
  Deze kunnen als proces gestart worden ! (hebben een startpunt = 'main' functie)
  Hebben hun functionaliteit in de vorm van compiled instructies.
  Ze kunnen 'eigen' functies aanroepen en library functies via een 'stub'.

Het proces : compileer – link – execute

Hogeschool van Amsterdam

Het maken en runnen van een executable file kent meerdere stappen :




```
*.c   source code
*.h   header
*.o   object file
*.so  shared object(dll)
*     executable
```

1. Het c-programma wordt **gecompileerd** naar een object file. [*.c / *.h → *.o]
2. Een of meer object files worden **gelinkt** tot een excutable file (of een library)
3. Een excutable file kan worden **geladen** in een proces

Het proces : compileer – link – execute

Hogeschool van Amsterdam

Het ELF format wordt gebruikt voor zowel

object files [voor linking E**L**F]
als
executable files [voor excutie **E**LF]

Headers :

- de secties (via section headers) zijn nodig om te kunnen **linken**.
- de segmenten (via program headers) zijn nodig om te kunnen **laden**.

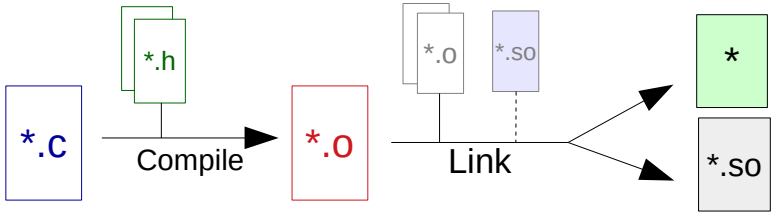
Preproces – Compile – Link

Hogeschool van Amsterdam

3 stappen

0. *Preproces : headers worden ingekopieerd*

1. Compile : c-code wordt vertaald naar object code
   met assembly-code als 'tussenstap'.



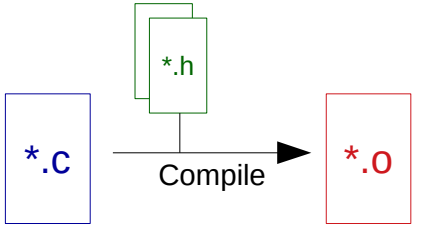
Informatie over globale variabelen / functies*, worden in de **secties** / **sectie headers** vastgelegd.
(deze info is nodig bij het linken) *Het resultaat is een 'object file' [in **ELF** format]*

2. Link : een of meer object files worden samengesteld.
   Nu wordt een definitieve 'layout' van het virtuele geheugen gemaakt.
   Nu is alle informatie beschikbaar. *Het resultaat is een 'executable file' of library [in **ELF** format]*

* *onder anderen de namen (!)*

0. Preprocessing

Hogeschool van Amsterdam



Includes wordt ingekopieerd
alle functies en constanten, gedefinieerd in de header, zijn nu 'onderdeel' van het programma.

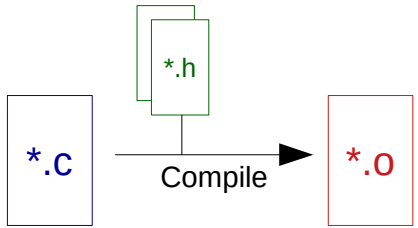
Commando

- gcc -E stopt na het preprocess stap. [geeft output in 'speciaal' formaat.]
- cpp is de preprocessor.

1. Compile [hier gebeurt een heleboel !]

Hogeschool van Amsterdam

De compiler maakt een **lay-out van het virtuele geheugen**.



1. elke **globale** variabele **en** elke **functie** krijgt een 'vaste' locatie
2. elke **functie** krijgt extra instructies voor gebruik v/d stack *
3. de **instructies** worden vertaald naar binaire instructies (te begrijpen m.b.v. assembler instructies)
4. (tussen) **resultaten** worden in registers geplaatst
5. alle **programma structuren** (**if**, **for**, **while**, etc) krijgen jump instructies,
   condities in deze programma structuren worden cmp of test instructies

Elk item in deze lijst wordt later in detail besproken !

* zie document : stack.pdf

2. Link algemeen

Hogeschool van Amsterdam

Stappen in het link proces

- Alle object files worden samengevoegd.
- Alle '**stubs**' moeten worden ingevuld.
  - Functies uit een andere object file zijn nu 'bekend'.
  - functies uit bibliotheken :
    - Bij 'statisch linken' wordt de functie uit de bibliotheek gekopieerd.
    - Bij 'dynamisch linken' wordt extra instructie toegevoegd om op run time de juiste bibliotheek (DLL) te kunnen vinden.
- Nu kan de volledige 'plattegrond' van het virtuele geheugen worden gemaakt en alle adressen (offsets) van variabelen, jumps en functies worden berekend.
  Deze info wordt in de **program headers** / **segmenten** vastgeled.

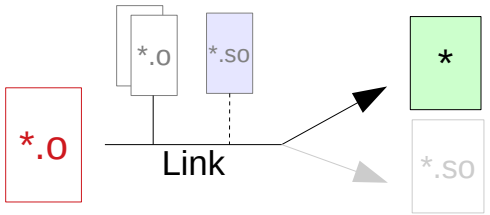
Het resultaat is een binair bestand [in **ELF** format].

3. Link – resultaat

Hogeschool van Amsterdam

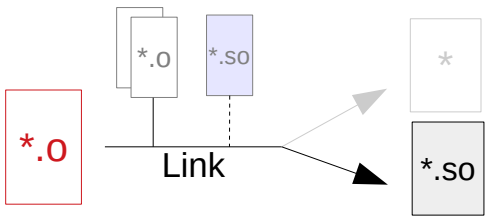
Het resultaat is :

- een executable bestand
  een normaal programma
  heeft een entry punt.



- een shared object / DLL
  een library
  heeft **geen** entry punt

  alle functies/variabelen kunnen worden gebruikt
  om statisch mee te linken
  of
  om (tijdens het runnen) dynamisch te gebruiken. [mbv stubs]



Merk op : ook een executable kan als library gebruikt worden

3. Link – symbols

Hogeschool van Amsterdam

Symbols

Om, in run-time, een functie in een library te kunnen gebruiken moet het aanroepende programma de library en de naam van de functie kennen.
De library* moet weten op welk adres de code van de functie staat, en moet dit m.b.v. de naam kunnen vinden.

Hiervoor wordt een **symbol table** gebruikt.
simpelweg een lijst van namen met offsets binnen het code (= .text) segment.

Strip

Deze tabel kan worden leeggemaakt d.m.v. een **strip** commando.
Dan kunnen de functies niet meer gevonden worden door andere programma's. [zie : man strip]

Ook de malware – analist kent nu de namen niet meer !

Debug symbols

Ten behoeve van het debug proces kan door de compiler extra informatie worden vastgelegd in een executable/library.
Hierbij wordt vastgelegd bij welke source code regel een machine instructie behoort.

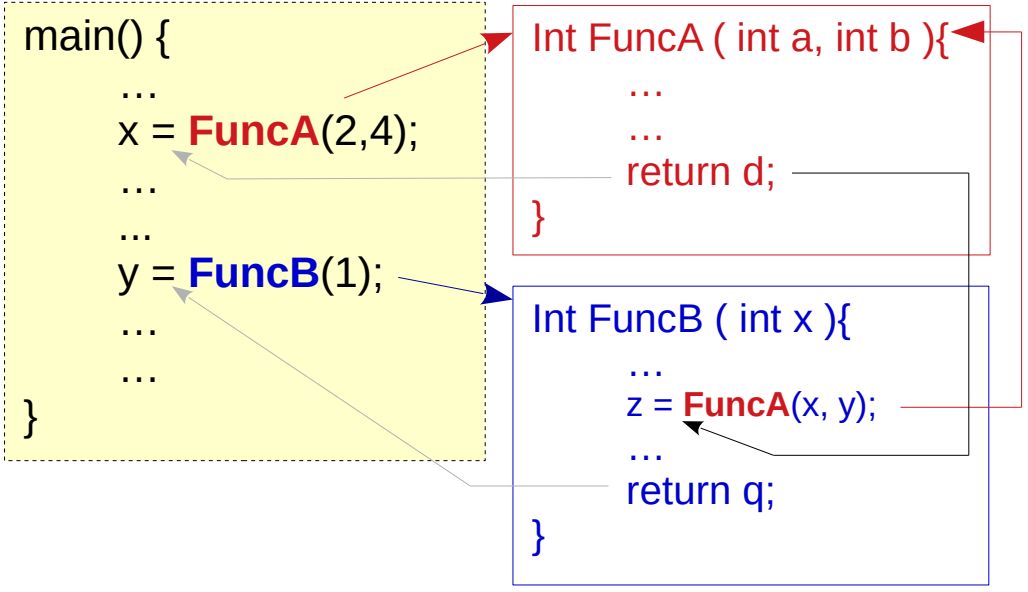
Met behulp van debug informatie kan de hele source code (incl. alle namen) worden gereconstrueerd !

* dit geldt natuurlijk ook voor executables !

Functies : probleem → oplossing

Hogeschool van Amsterdam

Bekijk het volgende schematische programma :



Als de functie klaar is dan :

- moet het programma verder gaan met de 1e instructie na de aanroep !
  Maar waar is dat ??
- De returnwaarde moet beschikbaar komen op de plaats waar de functie is aangeroepen !
  Maar waar is dat ??
- De functie moet parameter mee (kunnen) krijgen !
  Hoe doen we dat ??

We kunnen deze zaken niet vastleggen bij de functie zelf. Het is immers steeds anders !!

Oplossing :

Administreer deze zaken bij elke aanroep van een functie !

Gebruik hiervoor een stack.

Stack & Functies

Hogeschool van Amsterdam

Functies en de **stack** zijn sterk met elkaar verbonden.

Functie :

1. aanroep op alle plaatsen mogelijk, ook herhaald / genest
   een nieuwe aanroep is mogelijk , voordat de vorige functie returned !!
2. parameters zijn steeds anders, lokale variabelen hebben andere waarden
3. vervolg na afloop , afhankelijk van plaats van aanroep

Stack :

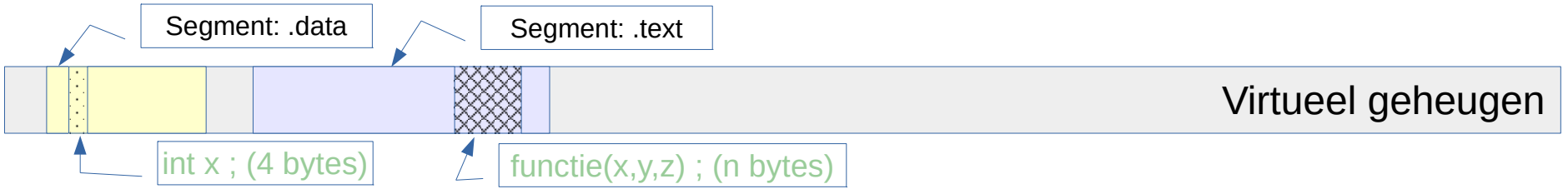
1. mogelijkheid van 'stapeling'
2. ruimte voor opslag van tijdelijke variabelen (parameters)
3. herstel oorspronkelijke situatie.

Voor een uitgebreide beschrijving van stack-gebruik en functie-aanroepen, wordt verwezen naar het document ***stack.pdf***

Virtueel geheugen – 1

Hogeschool van Amsterdam

*elke **segment** krijgt 'eigen' info en plaats in het geheugen.*



Variabelen :

- Globale variabelen krijgen een plaats in het **.data** segment.
  (of .rodata voor read-only data)

Functies :

- de instructies van een functie worden in het **.text** segment geplaatst.
  (het segment heet .text en bevat code !)

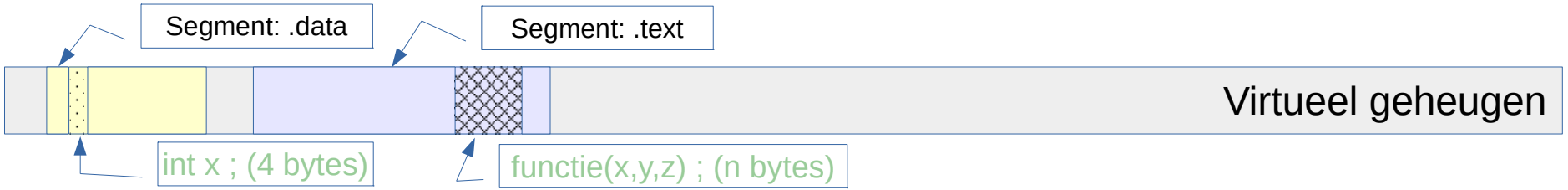
Elk segment krijgt :
- permissies [RWE]
- informatie voor het laad-proces.

Voor het correct starten van het proces.

Virtueel geheugen – 2

Hogeschool van Amsterdam

*elke **globale** variabele en elke **functie** krijgt een 'vaste' locatie.*



De compiler :

1. verzamelt alle onderdelen voor de elk segment
2. bepaalt voor elk deel de afstand tot het begin van het segment (de offset !)
3. geeft elk segment in plaats in het virtueel geheugen
4. herberekent voor elke variabele en functie de afstand tot het begin van het virt. Geheugen.
   Dit geeft het adres van de variabele/functie.

Virtueel geheugen – 3

Hogeschool van Amsterdam

Hoe wordt een variabele of functie gevonden in virtueel geheugen ?

Lokale variabele	`RBX + offset`	Offset t.o.v. base pointer : in stack frame !
Globale variabele	`RIP + offset`	Offset t.o.v. de instructie pointer.
functie	`adres`	Direct adres. immers de functie en de plaats van aanroep zitten beide in hetzelfde segment (**.text**)

Assembly – intro 1

Hogeschool van Amsterdam

De compiler vertaalt de broncode naar machine instructies.

Dit gaat in 2 stappen :

1. Code instructies naar assembly instructies.
2. Assembly instructies naar machine instructies.

Terug vertalen is slechts gedeeltelijk mogelijk.

1. Van machine instructies terug naar assembly is **mogelijk**
   (de 'vertaling' is 1 op 1)

2. Van assembly naar bron-code is **niet** (goed) **mogelijk**.
   Informatie is verloren gegaan.
   Namen van variabelen (en functies) zijn niet meer bekend.
   programma structuren zijn vervangen door jump-structuren.

We zullen ons dus moeten behelpen met assembly instructies als we alleen de gecompileerde vorm van een programma hebben.

Zoals het geval is bij analyse van malware !!

Assembly – intro 2

Hogeschool van Amsterdam

Assembly 'werkt' direct op de hardware.

Er wordt gebruik gemaakt van :

- vaste waarden
- registers
- adressen en offsets.

Instructie

- Beschreven met opcodes en operanden
  (~ parameters)
- Voeren zeer elementaire bewerkingen uit.
  [verplaats , ophogen , optellen, etc …]

Notaties
voor assembly – instructies bestaan twee verschillende manieren van noteren :

- **INTEL notatie**
  de standaard van veel ASM compilers (MASM, NASM, TASM), en op het intel platform.
- **AT&T notatie**
  standaard op het linux/unix platform.
  de GAS (gnu assembler) ondersteund ook Intel notatie.

Assembly – notatie

Hogeschool van Amsterdam

	AT&T	Intel
Parameter volgorde	Source – destination movl $1, %eax	Destination – source mov eax, 5
Parameter type	Extra suffix (q,l,w,b) geeft de lengte van de operant	Lengte bepaald door register, anders met 'BYTE', 'DWORD', etc.
prefix	$ voor 'immediate value' % voor registers	Geen prefix, compiler bepaalt type
Effectief adres	DISP(base,index,scale) movl mem(%ebx, %eax, 4), %edx	Als een rekenkundige expressie : mov edx, [ebx + eax*4 + mem]

In de vervolg slides wordt de Intel notatie gebruikt. (is het meest simpel …)

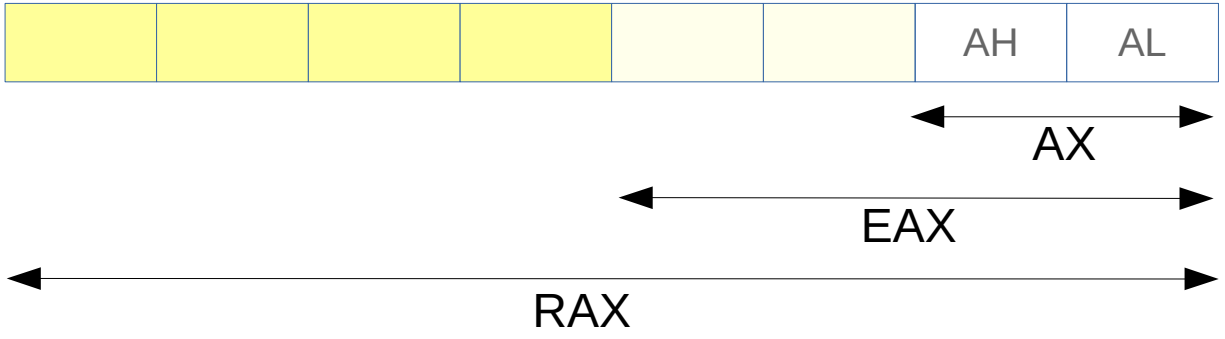
Assembly – registers

Hogeschool van Amsterdam

(tussen) **resultaten** worden in registers geplaatst

Registers zijn (zeer snelle) geheugen plaatsen, direct op de CPU.

In 1970: 2 bytes, tegenwoordig 8 bytes (64 bit !)



A – register

2 byte = 16 bits

4 byte = 32 bits

8 byte = 64 bits

Assembly – registers 2

Hogeschool van Amsterdam

(tussen) **resultaten** worden in registers geplaatst

De CPU kent meerdere registers. (64 bit mode)

General purpose registers:

8 'oude' registers : RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP

8 'nieuwe' registers : R8D t/m R15D

Segment registers :

CS (code segment), DS (data segment), SS (stack 'segment')
Extra segment pointers : ES, FS, GS.

Flag register :

EFLAGS register van 32 bits , elke bit een flag.

Instruction pointer register : RIP

Een apart register voor de instructie pointer

RAX – Accumulator for results
RBX – points to data in DS
RCX – counter for loops
RDX – I/O pointer
RSI – source pointer in DS
RDI – destination pointer in DS
RSP – Stack pointer in SS
RBP – pointer to stack-frame

Intel® 64 and IA-32 Architectures Software Developer's Manual
Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

Vol 1 § 3.4 Basic prog execution registers [blz 72]
Vol 1 § 3.4.3 EFLAGS register [blz 77 e.v.]

Assembly – machine instr.

Hogeschool van Amsterdam

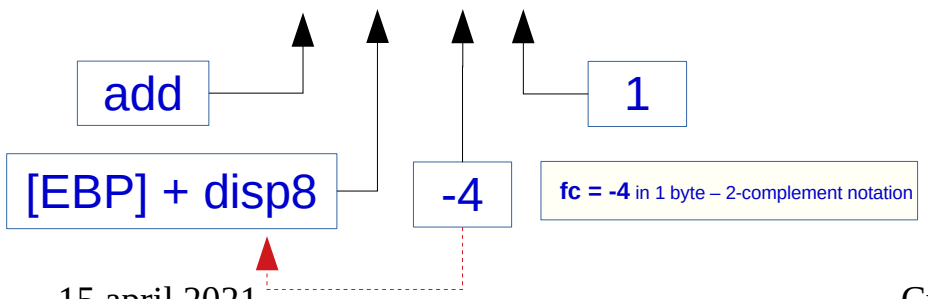
instructies in een exe zijn binaire code

Voorbeeld:

Stel lokale variabele a is van type int (4 bytes ; dubble word)

In source code, c-instructie voor 1 ophogen : `a++;`

83 45 fc 01 → add DWORD PTR [rbp-0x4], 0x1



Intel® 64 and IA-32 Architectures Software Developer's Manual
Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

Vol 2 § 2.1.5	Addr. mode encoding	[blz 506 e.v.]
Vol 2 § 3.2	Instruction ADD	[blz 601 e.v.]

Assembly – opcodes

Hogeschool van Amsterdam

Veel gebruikte assembler instructies :

Opcode	Betekenis	Opcode	Betekenis
MOV	Move van/naar registers/memory	**AND / OR / XOR / NOT**	Bitwise operaties
XCHG	Verwissel inhoud registers	**SHR / SHL**	Shift right / left
PUSH / POP	Stack functies	**JMP**	Jump (unconditional)
NEG	Verwissel getal van teken	**JE / JNE / J***	Jump if equal / not equal / etc.
ADD / SUB	Optellen / aftrekken	**LOOP / LOOPE**	Loop met RCX
MUL / DIV	Vermenigvuldigen / delen	**CALL / RET**	Call functie / return
INC / DEC	Verhogen / verlagen met 1	**NOP**	**Geen actie !!**
CMP	compare	**TEST**	test

Zie : stack.pdf

Intel® 64 and IA-32 Architectures
Software Developer's Manual
Combined Volumes: 1, 2A, 2B, 2C, 2D,
3A, 3B, 3C and 3D

Vol 2 chapter 3, 4, 5

Assembly – opcodes demo

Hogeschool van Amsterdam

Een reeks van instructies en het effect :

Instructie	***Toelichting***	**EAX**	**EDX**
mov edx, 13h	*Plaats de waarde 13h (=19 dec)*	?	13h
xor eax, eax	*eax wordt op nul gezet.*	0	13h
add eax, 2h	*tel 2 op bij eax*	2	13h
sub edx, eax	*Verlaag edx met de waarde van eax*	2	11h
mov eax, edx	*Copieer de waarde van edx naar eax*	11h	11h

Vaak zien we in een functie : de laatste acties kopieer een waarde naar eax.

conventie : **het eax register bevat de 'return-waarde' van een functie !**