Computer Arkitektur
Maskin-niveau programmering I

Forelæsning 4
Brian Nielsen

Credits to
Randy Bryant & Dave O'Hallaron (CMU)

Kursusgang 4-6: x86-64 Assembler

Intro til x86 Assembler: Adressering

- X86 Historik
- Assembly og objekt kode
  - gcc,as,gdb,objdump
- Instruktionsæt arkitektur
- Words (Q,L,W, B)
- Registre
- Addressering
  - Immediate,
  - Registre
  - Mem
- Aritmetiske operationer
- Logiske operationer

X86 Assembler: Kontrol- og data-strukturer

- Sammenligner
- Betingelsesflag
- Selektion
  - if-then-else
  - Betinget tildeling
- Iteration
  - While
  - Do-while
  - For
- Data-strukturer
  - Layout af Arrays i 1D og 2D
  - Indeksering
  - Structs
  - Alignment

X86 Assembler: Procedurekald

- Køretidsstak
- Push/Pop
- Kald og retur
- Parameteroverførsel
- Kalder/kaldte gemte registre
- Lokale variable
- Stack-frame
- Rekursion
- Buffer-overløb
  - Sikkerhedshuller og angreb
  - Kanarier
  - Addresserums randomisering
  - Non-executable stak beskyttelse

- **Hvordan ser maskinens grænseflade ud overfor programmøren?**
- **Hvad er en Instruktionssæt Arkitektur?**
- **Hvordan kodes høj-niveau (C) kontrol strukturer op?**
- **Hvordan opstår og forebygges sårbarheder ved bufferoverløb?**

Vigtige læringsmål for dagens kursusgang

PP3.1
PP3.5
PP3.7
PP3.10
PP3.11

- Grundlæggende forståelse og brug af x86-64 ISA-laget
  - Brug af Processor Registre
  - Adressering og adresseringsformer
  - Forstå og opskrive x86 assembler til beregning af simple udtryk

CH2

- Forstå forskellen imellem kilde-kode, objekt-kode og maskin kode
  - Bruge værktøjerne gcc, (assembler)og objdump til assemblering og dissassemblering

- Mindre intensivt
  - Historik

Intel x86 Udvikling: Milepæle

Navn	*Dato*	*Transistorer*	*MHz*
• 8086	1978	29K	5-10
• Første 16-bit Intel processor. Basis for IBM PC & DOS			
• 1MB adresserum			
• 80386	1985	275K	16-33
• Første 32 bit Intel processor, benævnt IA32			
• Tilføjet "flad adressering", kan køre Unix			
• Pentium 4E	2004	125M	2800-3800
• Første 64-bit Intel x86 processor, benævnt x86-64			
• Core 2	2006	291M	1060-3500
• Første multi-core Intel processor			
• Core i7	2008	731M	1700-3900
• 4 kerner			
• Core i9	2017	7000M	2600-4000
• 18 kerner			

Konkurrent: Advanced Micro Devices (AMD)

IBM PC AT (80286)

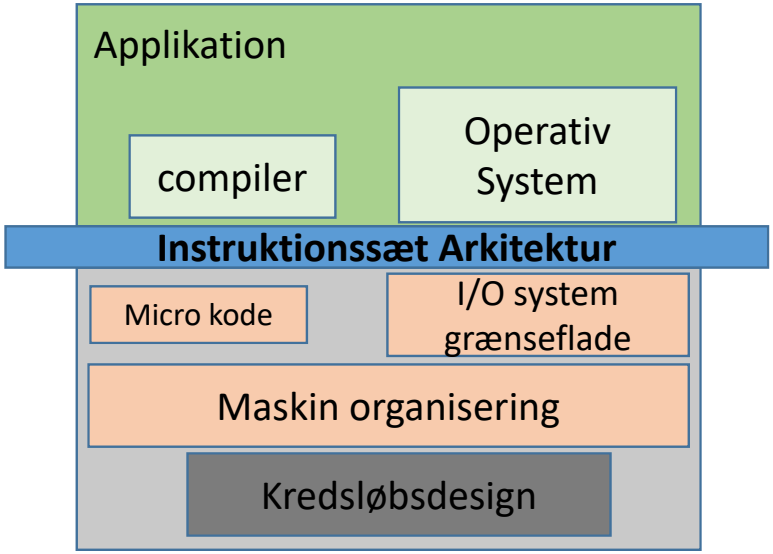
Tilføjede features

- Instruktioner til Multimedie / vektor processering
- Instruktioner til betingede operationer
- Matematik/Float
- Hukommelsesadministration (paging)
- Udvikling fra 16 -> 32 -> 64 bits
- Hyperthreading
- Flere/mange kerner
- Kryptering / virtualisering
- Masser af tricks til øgning af hastighed
- …

C, ASM, Maskinkode

Definitioner

- ISA-Arkitektur: (instruction set architecture) De dele af processorens design, som man skal forstå for at kunne læse/skrive assembler og maskinkode.
  - Processorens "API"
  - Instruktioner? Ordlængde? Registre? Indkodning?, mv.
- Micro-arkitektur: Implementeringen af arkitekturen.
  - Omfatter, bl. a. Udførelsen af instruktion, pipelining, "branch prediction", logiske bygge-blokke og deres sammenhæng
- Eksempler på ISA:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Ofte anvendt i mobil-telefoner
  - PowerPC, M68000, VLIW,…



Java VM, Common Language Runtime (CLR) har også en ISA!

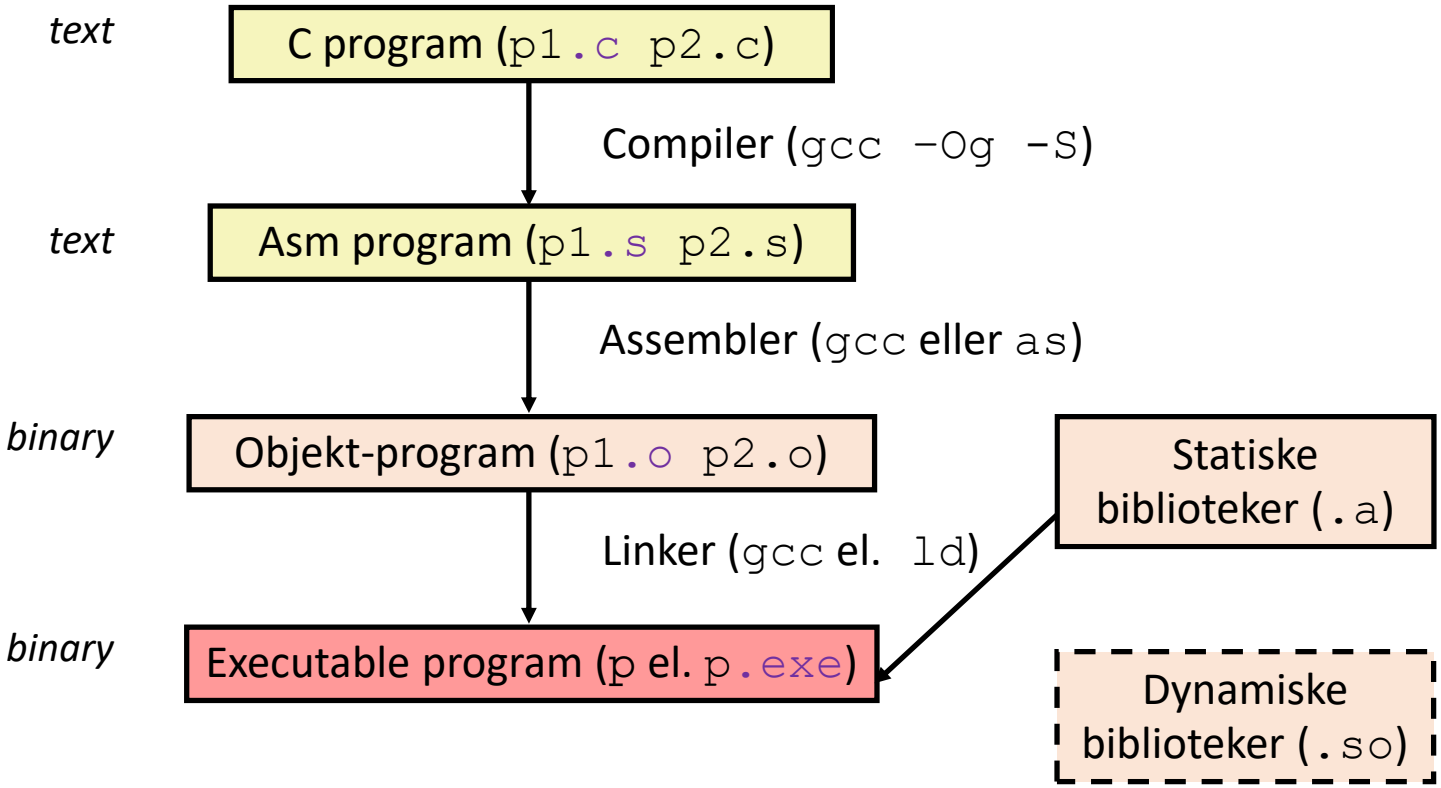
https://en.wikipedia.org/wiki/List_of_instruction_sets

Oversættelse af et C-program

- Kildetekst i filerne **`p1.c p2.c`**
- Oversættes med kommandoen: **`gcc –Og p1.c p2.c -o p`**
  - Debug venlig optimeringer (**`-Og -ggdb`**)
  - Gem binære resultat i filen **`p`**

Assembler kode: Tekstuel repræsentation af maskinkode.

Maskinkode: binær repræsentation af programmet som det lagres/udføres af processoren



Oversættelse af et C-program: Eksempel

C kode (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

x86-64 Assembler (sum.s)

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Objekt-kode (sum.o)

```
0x0000005
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0x00
  0x00
  0x00
  0x00
  0x48
  0x89
  0x03
  0x5b
  0xc3
```

sum.exe (a.out)

```
0x0400595:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0x48
  0x89
  0x03
  0x5b
  0xc3
```

Kan genereres (fx i den virtuelle maskine) med kommandoen

`gcc –Og –S sum.c`

Producerer assembler filen `sum.s`

OBS! Resultatet afhænger af compiler og compilerversion, og options.

- Totalt 14 bytes for sumstore
- Hver instruktion fylder 1, 3, or 5 bytes
- Starter på addresse `0x0400595`

Demo

- Se videoen:
- https://www.moodle.aau.dk/mod/page/view.php?id=1009972
- Fra minut 28-40.

Objekt kode

Kode for `sumstore`

```
0x0000005
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0x00
  0x00
  0x00
  0x00
  0x48
  0x89
  0x03
  0x5b
  0xc3
```

```
0x0400595:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0x48
  0x89
  0x03
  0x5b
  0xc3
```

- Totalt 14 bytes
- Hver instruktion fylder 1, 3, or 5 bytes
- Starter på adresse `0x0400595`

- Assembler
  - Oversætter `.s` til `.o`
  - Binær indkodning af hver instruktion
  - Næsten fuldstændig repræsentation af et eksekvérbart program
  - Mangler linking af referencer mellem kode i forskellige filer
- Linker
  - Kombinerer filerne til een.
  - Kobler referencer mellem filer
  - Kombinerer med statiske biblioteker
    - Fx., kode for **`malloc`**, **`printf`**
  - Mange biblioteker linkes *dynamisk*
    - Foretages når programmet startes

Disassemblering af objekt-kode

Disassembleret

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

- Disassembler

  `objdump -d sum`

  - Nyttigt værktøj til at undersøge objekt-kode
  - Analyserer bit-mønstret af en serie instruktioner
  - Udskriver tilsvarende assembler kode
  - Kan køres på en `a.out` (executable) eller `.o` fil

Alternativ disassemblering m. GDB

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- I gdb debugger værktøjet: **`gdb sum`**
  - **`>disassemble sumstore`**
    - Disassemble procedure
  - **`>x/14xb sumstore`**
    - Examine the 14 bytes starting at `sumstore`

What kan/må disassembleres?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

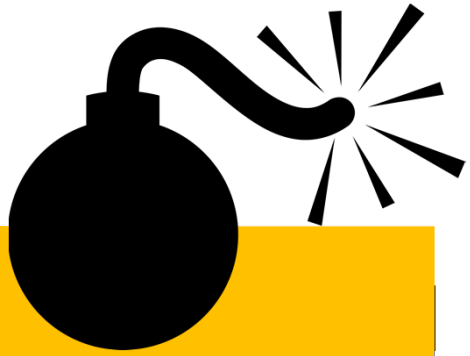
Reverse engineering forbudt i
Microsoft End User License Agreement

Også Java byte-kode / .NET kan også disasembleres

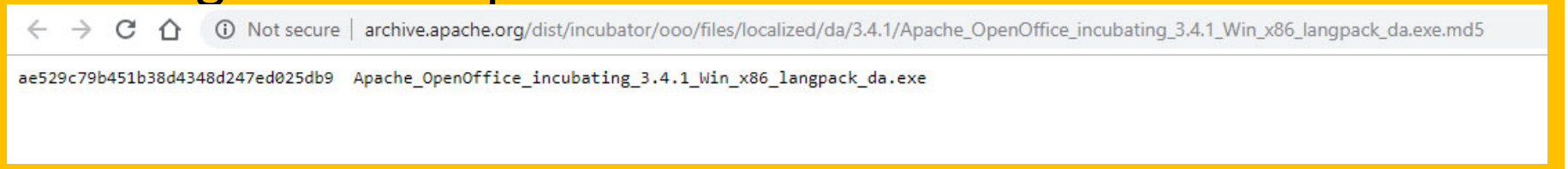
- ”disassembly attacks”
- ”Obfuscator”tool

- Alt som kan fortolkes som eksekverbart kode
- Disassembler undersøger bytes og rekonstruerer assembler kode.

Advarsel!



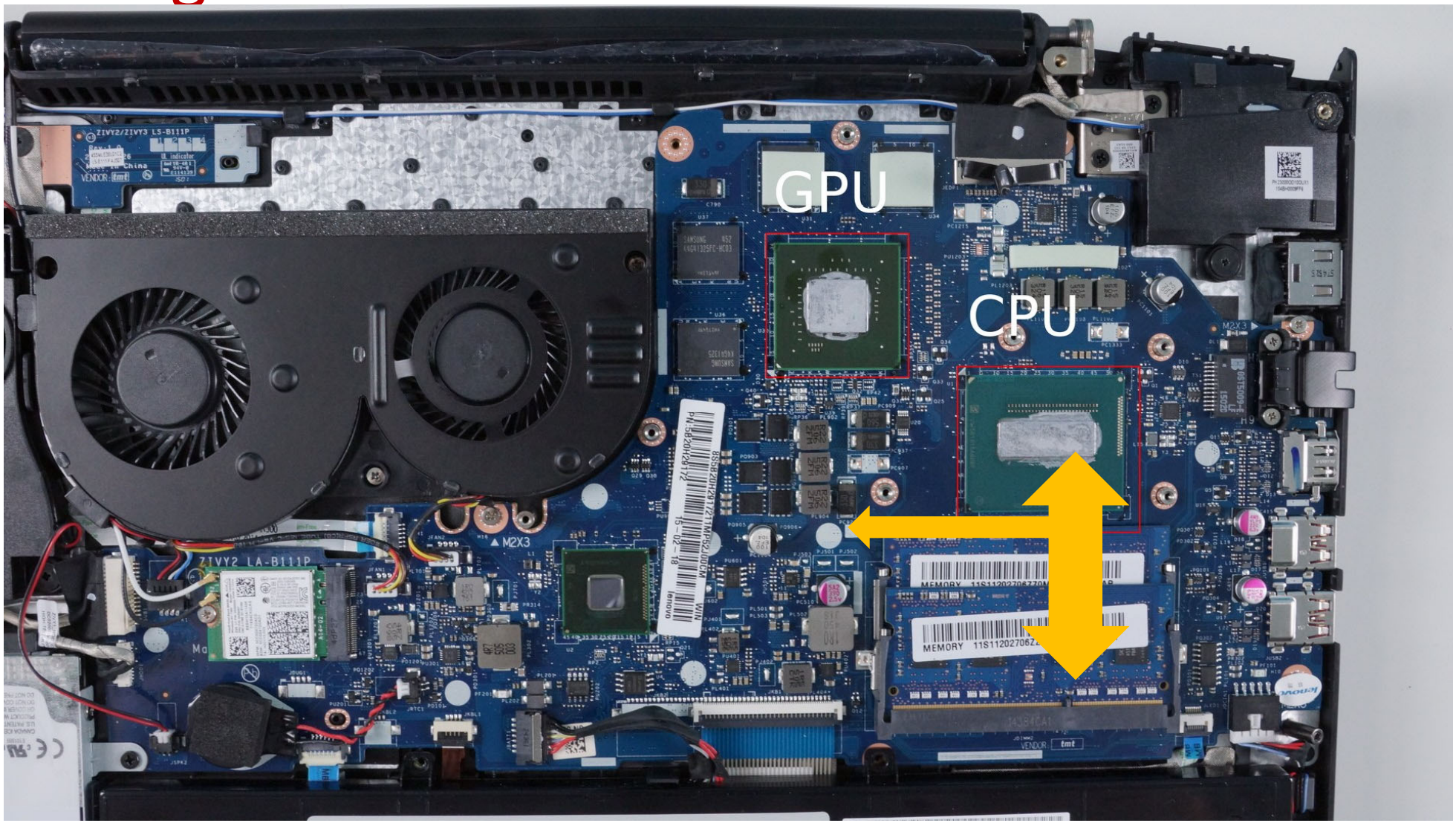
- Binære programmer er lette at modificere!
- Malware, Trojanske heste!
- Vær varsom med download af programmer (især executables) fra untrusted sites
- Check evt. filehash/MD5/checksum publiceret på trusted site (officiel udvikler) med de aktuelle værdier beregnet lokalt på filen fra et download mirror



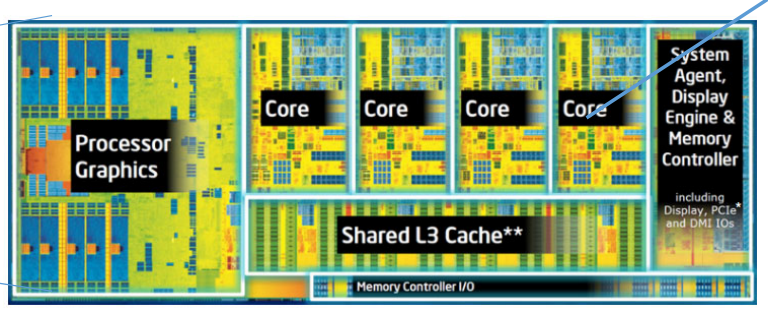
- Al
- Di
  assembler kode.

Adressering og data-flytning

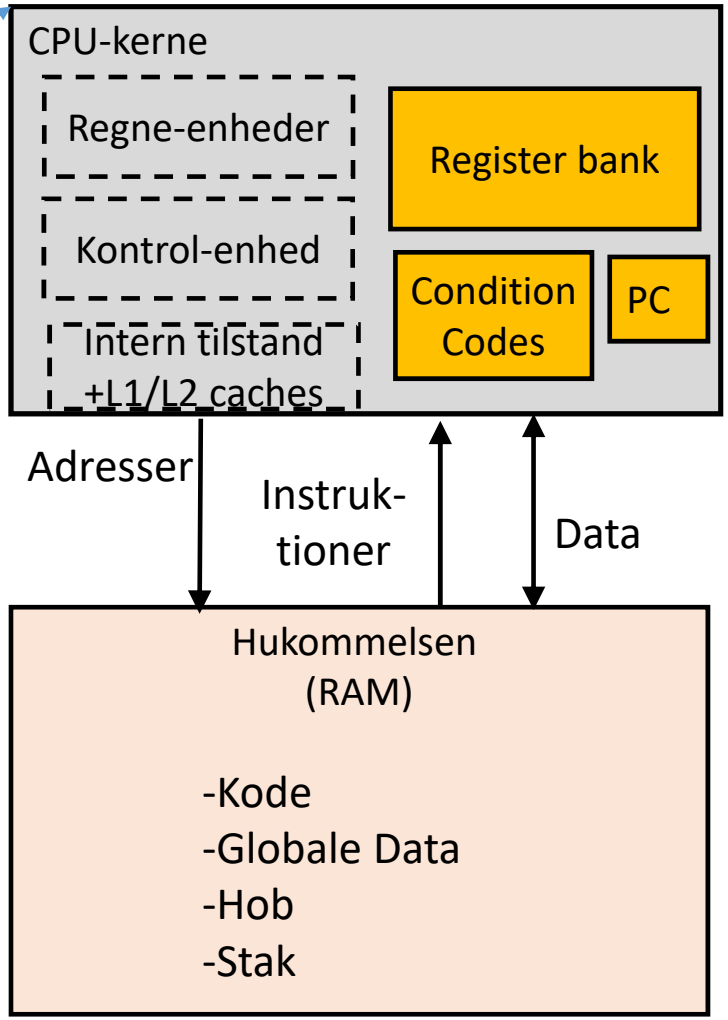
Virkeligheden



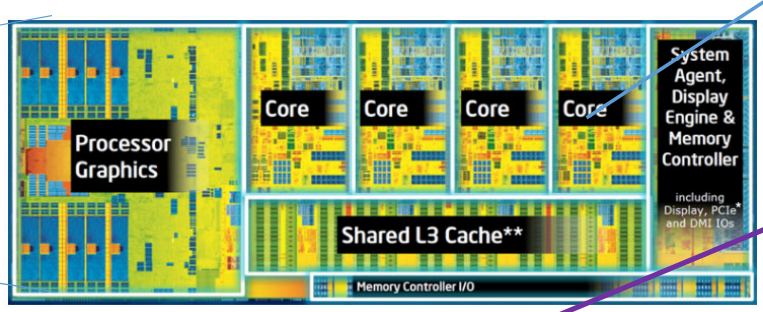
Simpel model



B
U
S



Simpel model



B
U
S

while(1) {
1. Send adresse til hukommelsen, som udbeder næste instuktion
2. Afvent data for instruktionen
3. Udfør instruktionen
4. Opdater registre med resultat,
5. Gem evt. resultat i hukk.
6. Beregn adresse på næste instruktion

}

CPU-kerne

Regne-enheder

Register bank

Kontrol-enhed

Condition Codes

PC

Intern tilstand +L1/L2 caches

Adresser

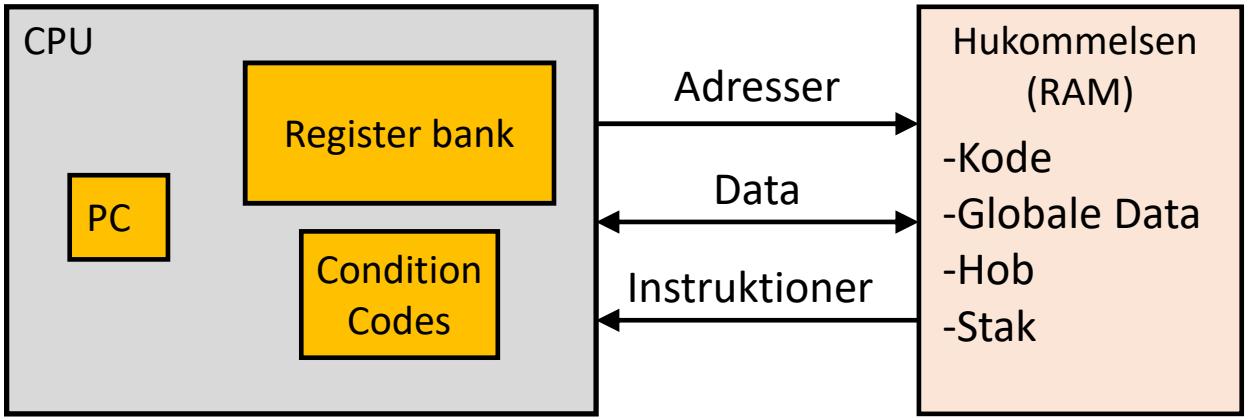
Instruk-tioner

Data

Hukommelsen (RAM)

-Kode
-Globale Data
-Hob
-Stak

Assembler model for maskinen



Programmør-synlig tilstand

- **PC: Program Tæller (program counter)**
  - Adresse på næste instruktion
  - Benævnt "RIP" (x86-64) (instruction pointer)
- **Register bank**
  - Hyppigt brugt program data
    - Operander regneoperationer tages tit herfra,
    - Resultat gemmes ofte der
- **Betingelsesflag (Condition codes)**
  - Antal boolske værdier (flag), der lagrer information om udfaldet af seneste aritmetiske eller logiske operation
  - Anvendes fx. til betinget forgrening

- **Hukommelsen**
  - Byte-adresérbar: $\mathbf{M_1[addr] \rightarrow b}$
  - Kode og bruger data
  - Stak til procedurekald

Assembler Karakteristika

Datatyper

- Heltallig data: 1, 2, 4, or 8 bytes
  - Data værdier (typeløst)
  - Adresser (typeløse pointers)
- Floating point data: 4, 8, or 10 bytes
- Sammensatte typer som arrays og structs
  - repræsenteres som sammen-hængende blok bytes i hukom-melsen
- Kode:
  - Blok af bytes som indkoder en sekvens af instruktioner

Operationer

- Overfører data
  - Indlæse data fra hukommelse til register
  - Gemme register data i hukommelse
- Udfører aritmetiske operationer på register eller data i hukommelsen
- Kontrol-mekanismer
  - Ubetinget spring (jumps a la “goto”)
  - Betingede forgrening (a la “if”)
  - Procedure kald/retur
  - Undtagelser/Afbrydelser (Exceptions/Interrupts)

Eksempel på Maskin Instruktion

```
long t, long *dest;
…
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- C Code
  - Gemmer værdien af **`t`** i variable udpeget af **`dest`**
- Assembler
  - Flyt 8-byte værdi til hukommelsen
    - "Quad words" i x86-64 jargon
  - Operander:

    `t:` Register **`%rax`**

    `dest:` Register **`%rbx`**

    `*dest:` Hukommelsen (Memory) **`M[%rbx]`**
- Objekt-kode
  - 3-byte instruktion
  - Gemt i start adresse **`0x40059e`**

x86-64 Integer Registre, w=64

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

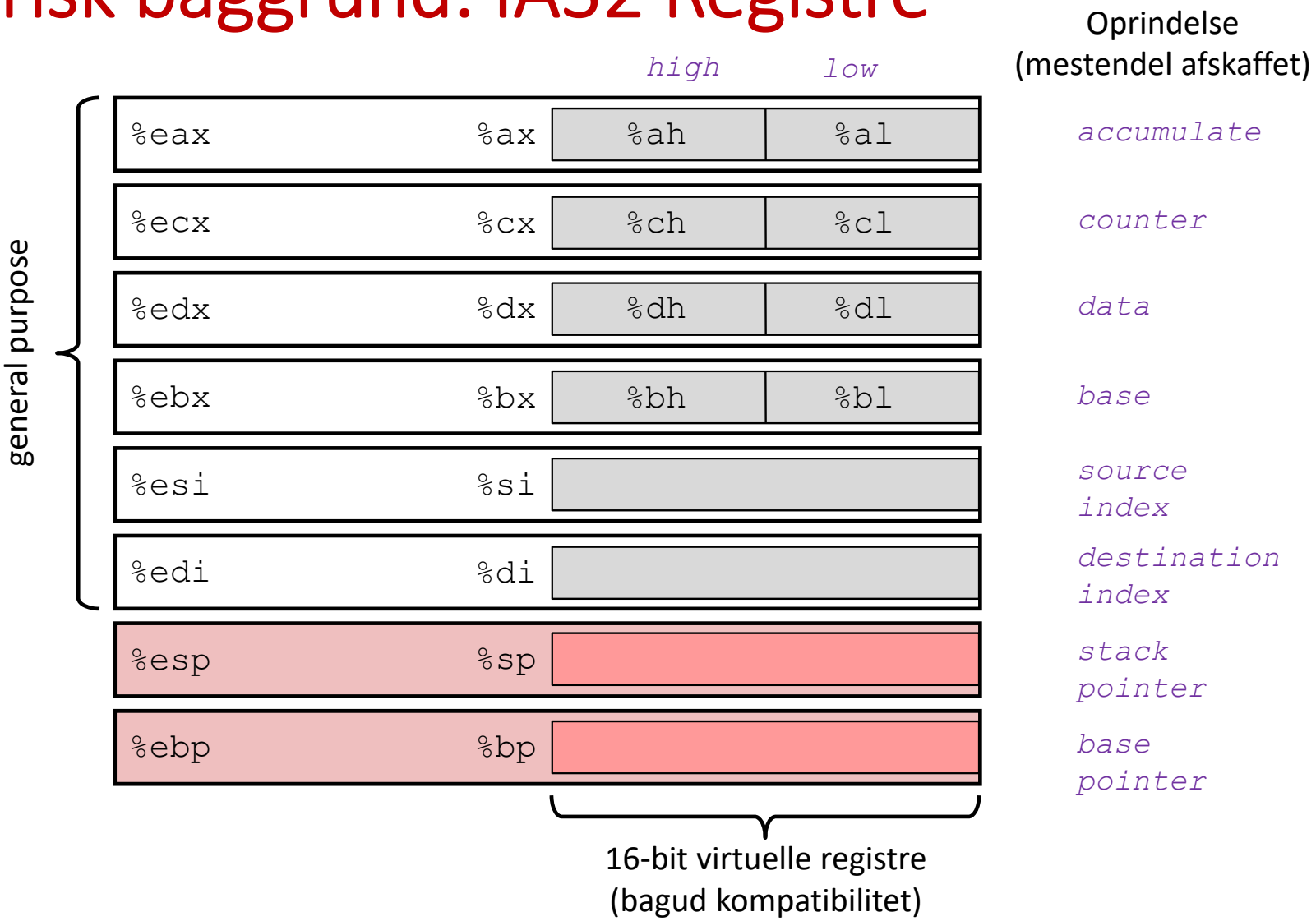
x86-64 Integer Registre

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- Vi kan referere til de 4 mindst betydende bytes (32 bits) ved ovenstående navne (virtuelle registre)

Historisk baggrund: IA32 Registre

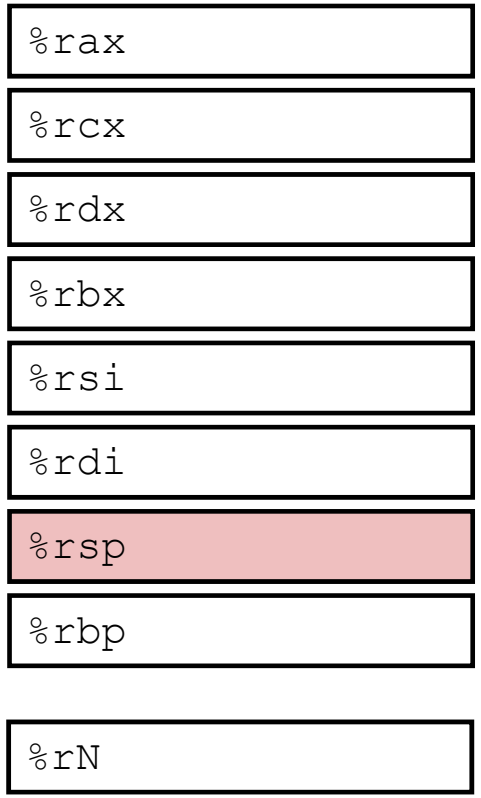


~~Moving~~ Kopiering af data

- Flytning af data

  movq ***Source, Dest*:**

- Operand Typer
  - ***Immediate (direkte):***
  - konstant heltallig data
    - Fx. **`$0x400, $-533`**
    - Som en C konstant, med foranstillet **`$`**
    - Kodet som 1, 2, or 4 bytes!!!
  - ***Register:*** Et af de 16 integer registre
    - Fx.: **`%rax, %r13`**
    - Men **`%rsp`** er reserveret
    - Andre kan have særlig brug i nogle specielle instruktioner
  - ***Memory:*** 8 følgende bytes i hukommelsen på start-adressen givet ved registeret
    - Fx.: **`(%rax)`**
    - Flere andre "address modes"

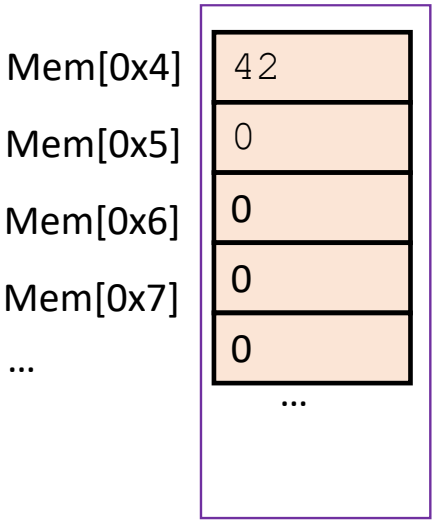


Eksempler

- Flytning af data

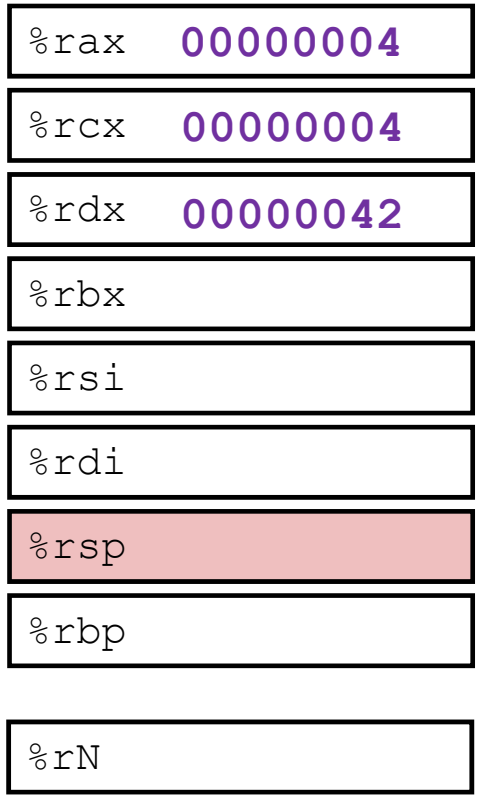
movq *Source, Dest*:

```
movq $0x4,%rax

movq %rax,%rcx

movq %(rax),%rdx
```



Quad=8 bytes

(little endian)



Mulig kombinationer af **`movq`** operander

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Vi kan ikke lave "memory-memory" overførsel i én instruktion

Direkte konstanter er kun 32 bits – brug dedikeret instruktion for at indlæse 64 bit værdier

x86-64 Integer Registers, w=64

Konventioner til funktions-kald (Linux)

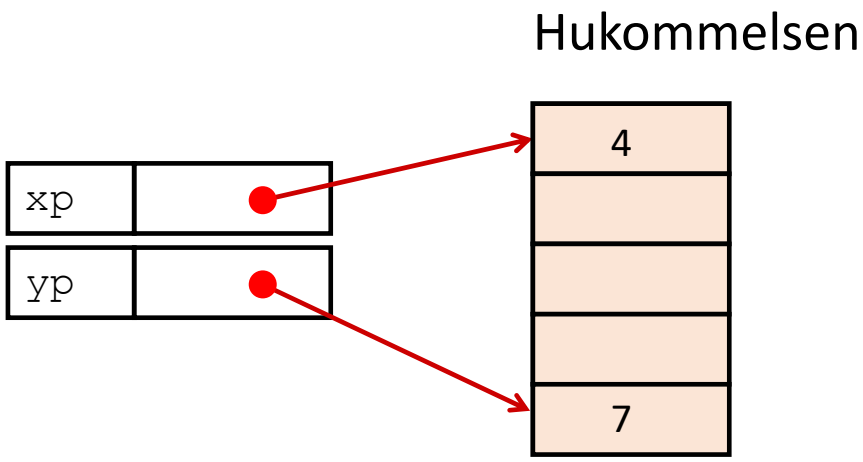
%rax *retur værdi*	%r8
%rbx	%r9
%rcx *argument 4*	%r10
%rdx *argument 3*	%r11
%rsi *argument 2*	%r12
%rdi *argument 1*	%r13
%rsp	%r14
%rbp	%r15

Argumenter (aktuelle værdier) overføres ofte i angivne registre

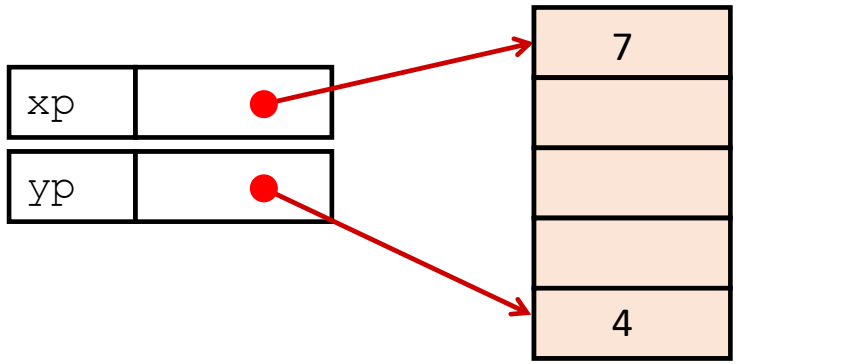
C: res=func(arg1,arg2,arg3) {…return v;}
ASM: "func(%rdi,%rsi,%rdx) {…return %rax;}"

Animering af **Swap**()

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

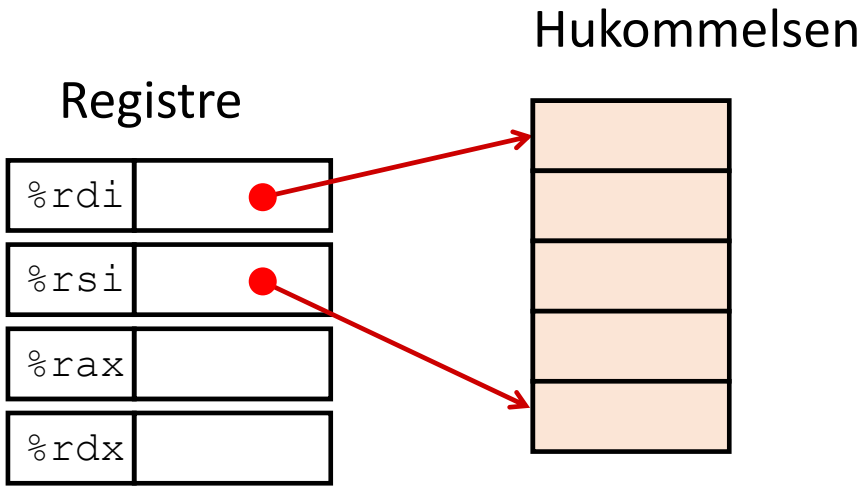


Ombytter indholdet af xp og yp
t0=4
t1=7
*xp=t1=4
*yp=t0=7



Animering af **Swap()**

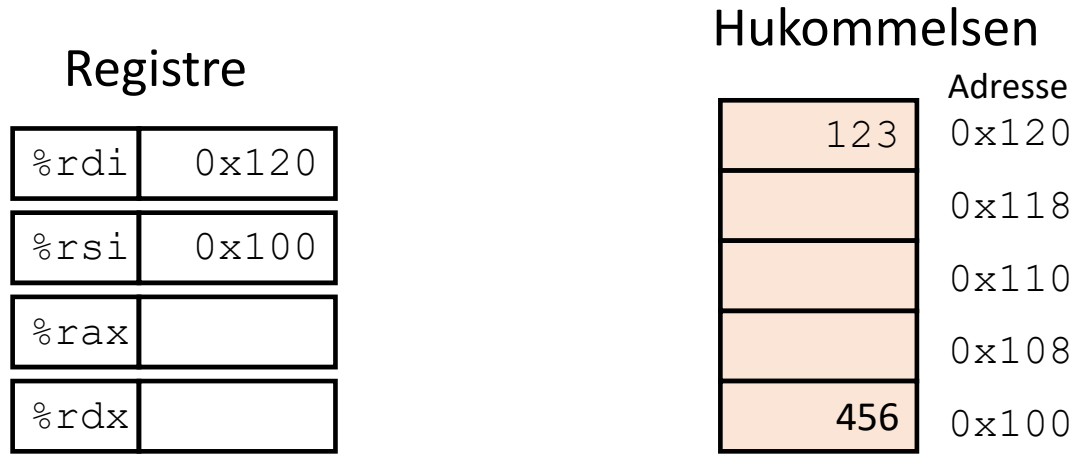
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

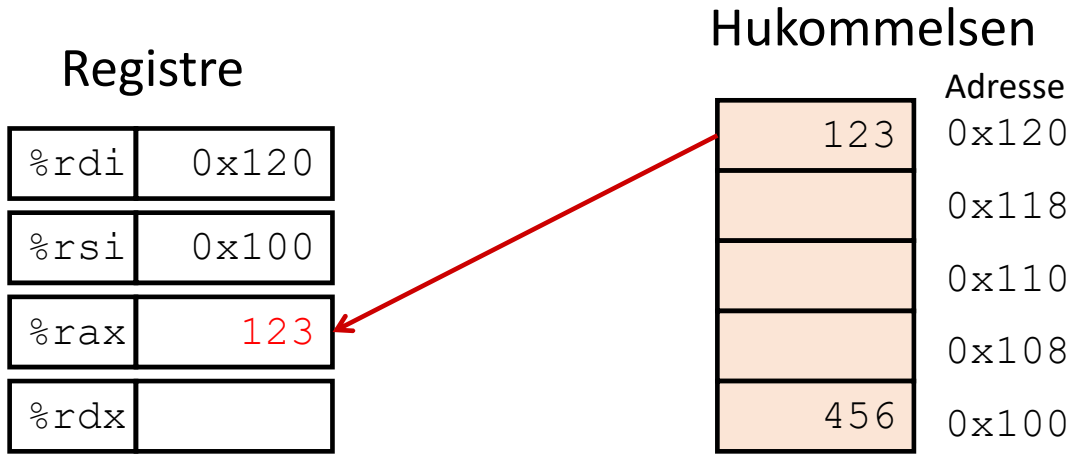
Register	Værdi
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

Animering af **`Swap`**()



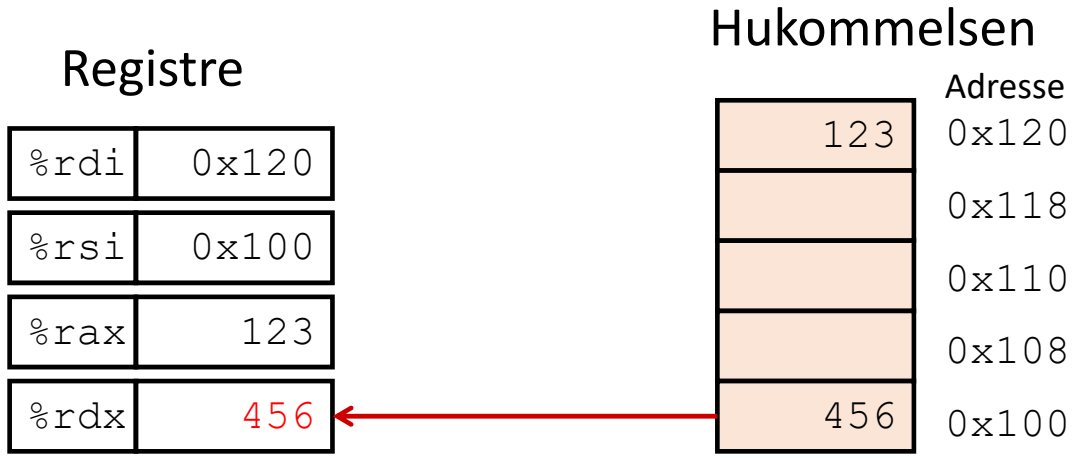

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Animering af **`Swap()`**



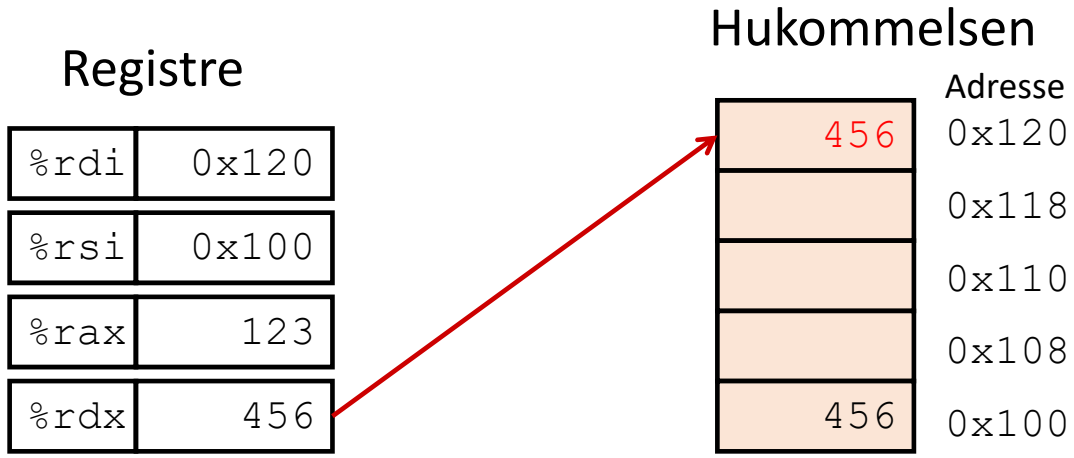

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Animering af **`Swap()`**



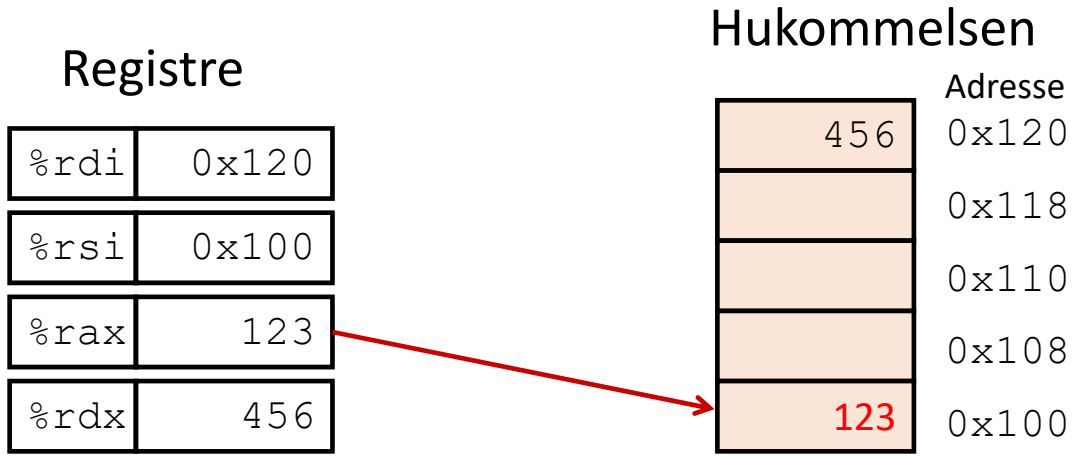

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Animering af **`Swap`**()




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Animering af **`Swap`**()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Simple adresseringsmåder

- Normalt
  (R) betyder: **Mem[Reg[R]]**
  - Register R angiver hukommelsesadresse
  - Aha! Pointer de-referering i C

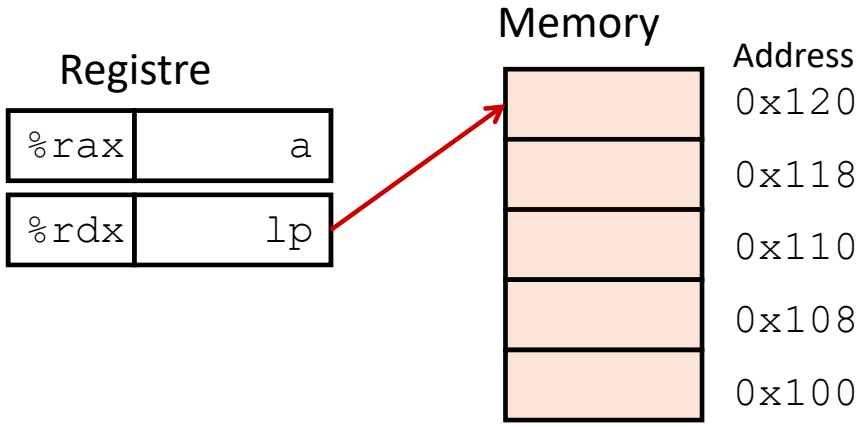
Vigtig notation fra bogen
- Mem modellers som array indekseret med addr
- Register bank modelleres som array indekseret med register id.

C

```
long a;
long *lp;
*lp=a;
a=*lp;
```

ASM

```
movq rax, (%rdx)
movq (%rdx), %rax
```



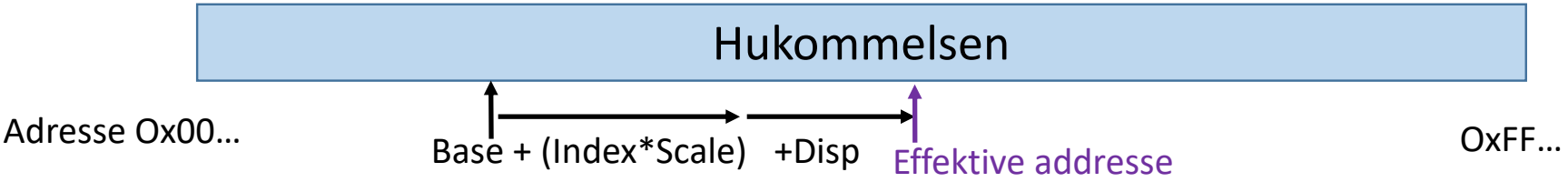
Fuldstændige adressingsmåde

- Det mest generelle format

  D(Rb,Ri,S) betyder: **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  `movq $512(%rsi,%rdi,8), %rax`

  - D: konstant forskydning ("displacement") 1, 2, or 4 bytes
  - Rb: Basis register: Ethvert af de 16 integer registre
  - Ri: Index register: Ethvert, undtagen **`%rsp`**
  - S: Skaleringsfaktor: 1, 2, 4, or 8



- Special Tilfælde

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]	*//disp=0, s=1*
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]	*//s=1*
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]	*//disp=0*

Eksempel på adresse beregning

%rdx	0xf000
%rcx	0x0100

Format: D(Rb,Ri,S)

Betydning: Reg[Rb]+S*Reg[Ri]+ D

Udtryk	Adresse beregning	Eff.Adresse
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Andre ordstørrelser på x86-64

C-erklæring	Intel Data type	Størrelse (bytes)	Assembler endelse	Ex. instruktion
char	Byte	1	b	`movb %cl, %al`
short	Word	2	w	`movw %cx, %ax`
int	Double Word	4	l	`movl %ecx, %eax`
long	Quad Word	8	q	`movq %rcx, %rax`
char *	Quad Word	8	q	`movq %rcx, %rax`

Bemærk:

- Ved `movb`, og `movw` ændres kun de angivne dele af 64 bit registret
- Undtagen `movl`, som sætter de 4 øverste bytes til 0: `movl $0xFFFF,%eax` => %rax=0x0000FFFF!
- Ved flytning af mindre data-type til større bør movzXX eller movzXX anvendes:
  - `movzbw Src, Dst` *# zero-extended byte to word*
  - `movsbl Src, Dst` *# sign-extended byte to double word*
  - ....

Aritmetiske og Logiske Instruktioner

Instruktion til adresse beregning

- `leaq` ***Src*, *Dst***
  - *"Load effective address"*
  - *Src* er "address mode" udtryk
  - Sætter *Dst* til adressen angivet ved udtrykket
- Anvendelser
  - Beregning af en adresse uden at tilgå hukommelsen a la **& operatoren i C**

```
long x[50];
long i=10;
long*p = &x[i];
```

```
# %rdx contains ptr to x,
# %rdi contains i
leaq (%rdx,%rdi,8), %rax
```

  - Beregning af aritmetiske udtryk af formen x + k*y
    - k = 1, 2, 4, or 8

```
long m12(long x)
{
  return x*12;
}
```

Compiler-genereret ASM:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2 =t*4
```

Udvalgte Aritmetiske Operationer

- To-operand instruktioner:

Format		***Betydning***	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	//***også kaldet shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	//***Aritmetisk højre***
`shrq`	*Src,Dest*	Dest = Dest >> Src	//***Logisk højre***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- Bemærk rækkefølge af argumenterne!
- Ingen skelnen mellem signed og unsigned int for add/mult (hvorfor ikke?)

Yderligere Arithmetiske Operationer

- 1-operand instruktioner

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

- Se lærerbogen for yderligere instruktioner

Eksempel på aritmetisk udtryk

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

Interessante Instruktioner

- **`leaq`**: adresse beregning
- **`salq`**: shift
- **`imulq`**: multiplikation
  - Kun brugt éen gang

Eksekvering af eksemplet

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Reg.	Indhold	leaq	addq	leaq	salq	leaq	imulq
%rdi	Arg x	x	x	x	x	x	x
%rsi	Arg y	y	y	y	y	y	y
%rdx	Arg z	z	z	3*y	3*y*2*2*2*2	48y	48y
%rax		x+y	x+y+z	x+y+z	x+y+z	x+y+z	(x+y+z) * (x+48y+4)
%rcx						x+48y+4	x+48y+4

Maskin programmering I: Resumé

- Historikken bag Intel processorer and arkitekturer
  - Design har udviklet sig henover mange år: mange særheder
- C, assembler, maskinkode
  - Processorens programørsynlige tilstand: program tæller, registre, ...
  - Compiler oversætter programsætninger, udtryk, procedure til sekvenser af lav-niveau maskin-instruktioner
- Grundlæggende Assembler: Registre, operander, move
  - x86-64 move instruktionen dækker mangeartede formater for data-flytning
- Aritmetik
  - C compileren “udtænker” hvilken kombination af instruktioner der er nødvendige for at udføre en given beregning