

INF105 - 06
Fra .py til CPU

David Grellscheid

Q&A



fra hello.py to CPU

print("Hello World")

Python-interpreter

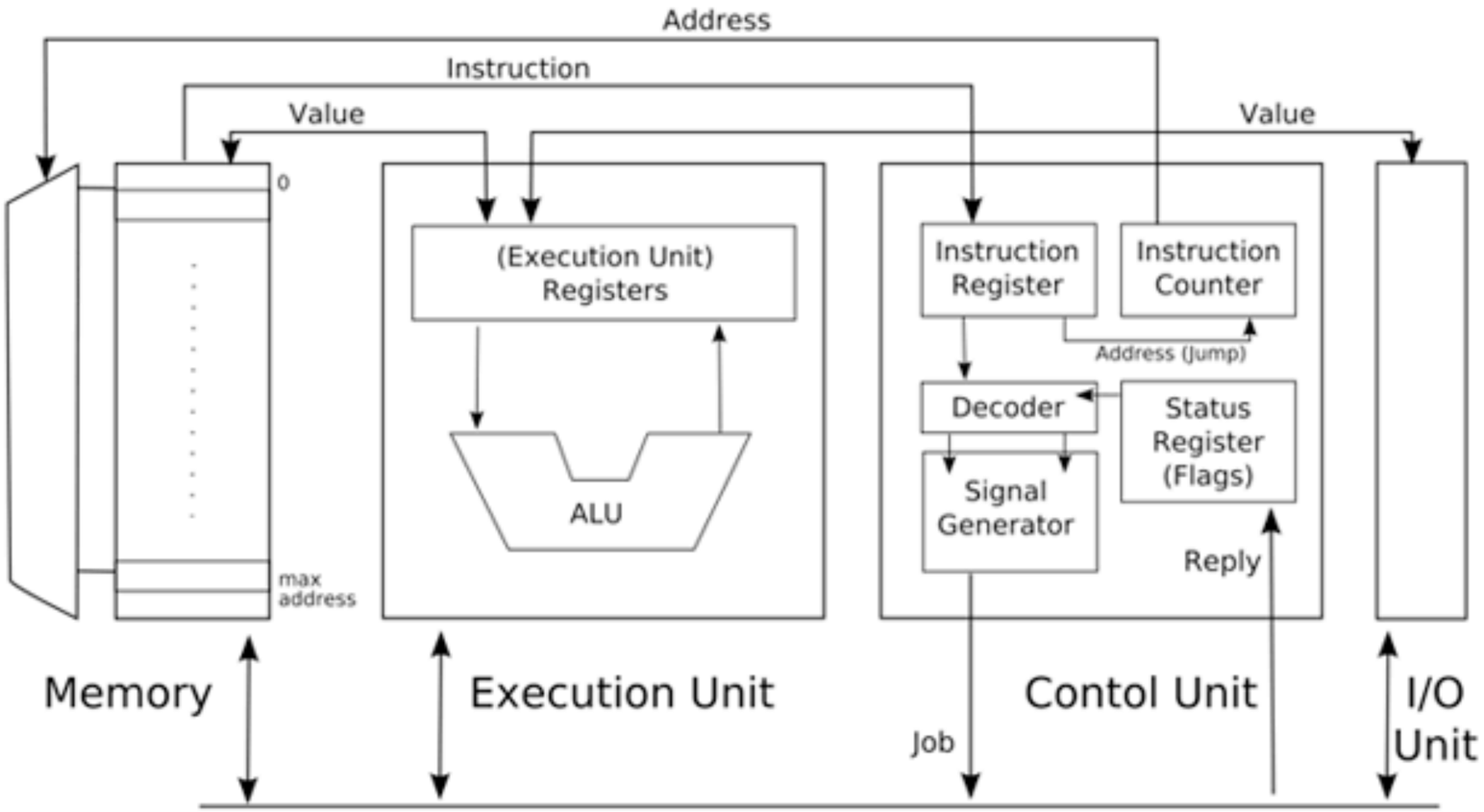
C (many high-level langs at this level)

Assembly

Machine code instructions

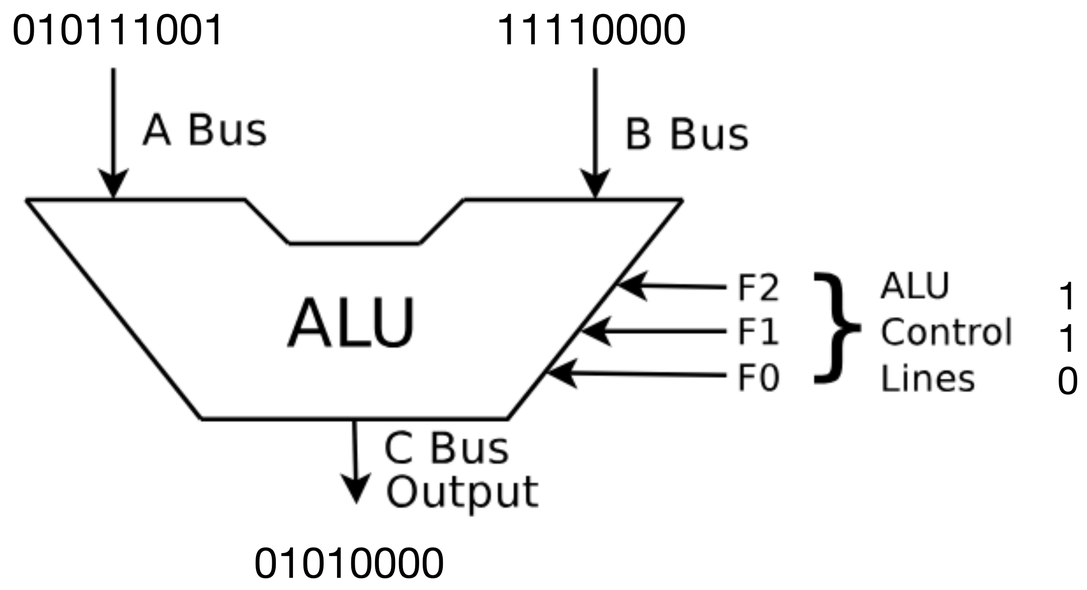
CPU

Von-Neumann arkitektur



Både instrukser og data lagres sammen i minne.
De fleste moderne CPU har denne strukturen

Arithmetic Logic Unit



F2	F1	F0	Output
0	0	0	A
0	0	1	B
0	1	0	A+1
0	1	1	B+1
1	0	0	A+B
1	0	1	A-B
1	1	0	A AND B
1	1	1	A OR B

8-bit ALU
(oftest nå 64-bit)

https://archive.org/download/computer-organization-and-design-fourth-edition-the-hardware-software-interface-/Step-by-Step%20Design%20and%20Simulation%20of%20a%20Simple%20CPU.2445296.pdf

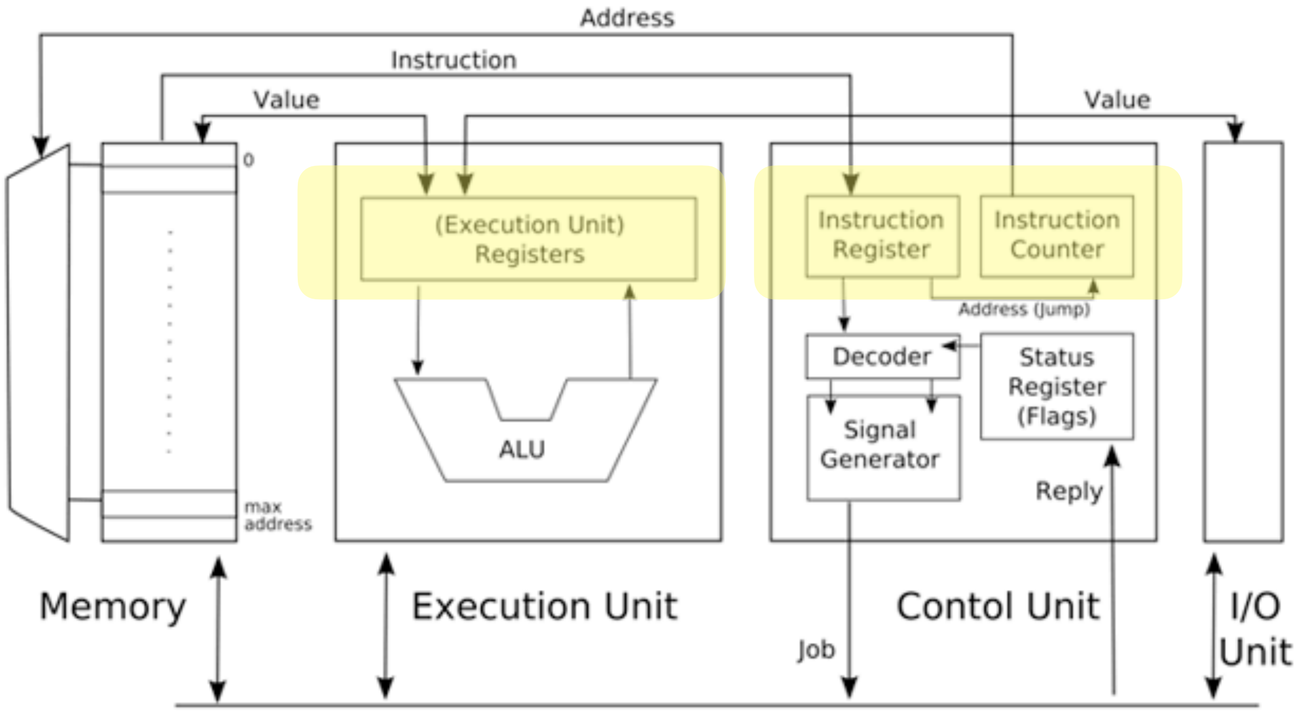
Arbeidssyklus

1. FETCH - last inn instruksen som IC peker på, øk IC

2. DECODE - tolke instruksen, beregn adresser som instr. trenger

3. EXECUTE - kjøre instr., f.eks
   move,
   add,
   compare,
   jump *(endrer IC)*

4. WRITE BACK
   - lagre resultat i registere



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

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

Demo

- https://github.com/m0ne/CPUSim

Demo

- https://github.com/m0ne/CPUSim
- binære opcodes er vanskelig å huske, istedenfor:
- 0xA0 -> MOV
  0x40 -> INC
  0x20 -> AND
  ...

- mer lesbart **assembly**-språk

Opcodes

			0x82...0x83/2 (186+)
ADD	Add	(1) `r/m += r/imm;` (2) `r += r/imm;`	0x00...0x05, 0x80/0...0x81/0, 0x82/0...0x83/0 (186+)
AND	Logical AND	(1) `r/m &= r/imm;` (2) `r &= r/imm;`	0x20...0x25, 0x80...0x81/4, 0x82...0x83/4 (186+)
CALL	Call procedure	`push eip; eip points to the instruction directly after the call`	0x9A, 0xE8, 0xFF/2, 0xFF/3
CBW	Convert byte to word	`AX = AL ; sign extended`	0x98
CLC	Clear carry flag	`CF = 0;`	0xF8
CLD	Clear direction flag	`DF = 0;`	0xFC
CLI	Clear interrupt flag	`IF = 0;`	0xFA
CMC	Complement carry flag	`CF = !CF;`	0xF5
CMP	Compare operands		0x38...0x3D, 0x80...0x81/7, 0x82...0x83/7 (186+)
	Compare bytes in memory.		

Spesifikt til hver CPU-arkitektur

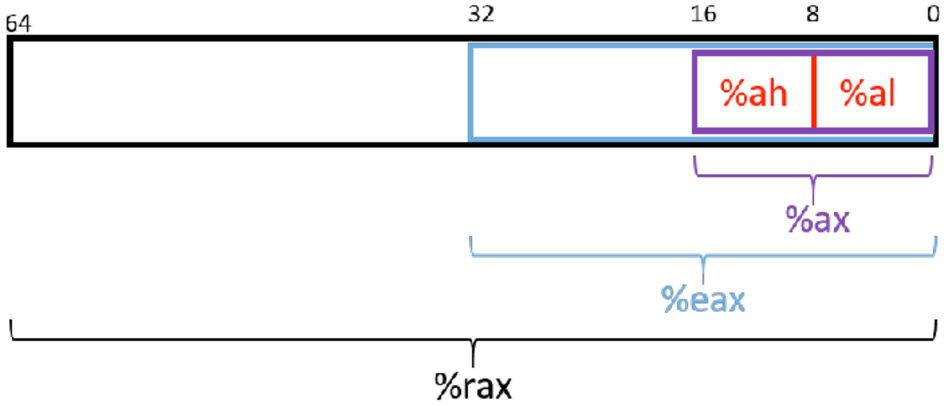
https://en.wikipedia.org/wiki/X86_instruction_listings

x86_64 registers

64-bit	32-bit	16-bit	8 high bits	8 low bits	Description
RAX	EAX	AX	AH	AL	Accumulator
RBX	EBX	BX	BH	BL	Base
RCX	ECX	CX	CH	CL	Counter
RDX	EDX	DX	DH	DL	Data
RSI	ESI	SI	N/A	SIL	Source
RDI	EDI	DI	N/A	DIL	Destination
RSP	ESP	SP	N/A	SPL	Stack Pointer
RBP	EBP	BP	N/A	BPL	Stack Base Pointer

64-bit	32-bit	16-bit	Description
RIP	EIP	IP	Instruction Pointer

flere til float-verdier:
xmm0
xmm1
...



https://diveintosystems.org/book/C7-x86_64/basics.html

arm registers

Most A64 instructions operate on registers. The architecture provides 31 general purpose registers. Each register can be used as a 64-bit X register **(X0..X30)**, or as a 32-bit W register **(W0..W30)**.

There is a separate set of 32 registers used for floating point and vector operations. These registers are 128-bit, but like the general-purpose registers, can be accessed in several ways. Bx is 8 bits, Hx is 16 bits and so on to Qx which is 128 bits.

The name you use for the register determines the size of the calculation. This example performs a 32-bit floating point addition: FADD S0, S1, S2

This example performs a 64-bit floating point addition: FADD D0, D1, D2

https://support.arm.com/documentation/102374/0103/Registers-in-AArch64---general-purpose-registers

Assembler x86

- Linux on x86_64
- (apk add binutils)
- as -o calc.o calc.S # assemble
- ld -o calc calc.o # link
- ./calc; echo $?
- objdump --disassemble calc

calc.S

```
.code64

.section .text

.global _start

_start:

  mov $8, %rax

  mov $7, %rbx

  add %rax, %rbx


  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

Verdien 8 i rax

Verdien 7 i rbx

Summe i rbx

Systemkall for å avslutte:

rax=60 betyr exit(), alltid bruker rdi som returverdien til programmet

mer om syscall i INF113

Assembler arm

- Linux on aarch64
- (apk add binutils)
- as -o calc.o calc.S # assemble
- ld -o calc calc.o # link
- ./calc; echo $?
- objdump --disassemble calc

calc.S

```
    .text

    .global   _start

_start:

    mov       w0, 8
    mov       w1, 7
    add       w0, w1, w0

    mov       w8, 93
    svc       0
```

Verdien 8 i w0

Verdien 7 i w1

Summe i w0

Systemkall for å avslutte:

w8=93 betyr exit(), alltid bruker w0 som returverdien til programmet

mer om syscall i INF113

```
calc:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:  48 c7 c0 08 00 00 00   mov    $0x8,%rax
  401007:  48 c7 c3 07 00 00 00   mov    $0x7,%rbx
  40100e:  48 01 c3               add    %rax,%rbx
  401011:  48 c7 c0 3c 00 00 00   mov    $0x3c,%rax
  401018:  48 89 df               mov    %rbx,%rdi
  40101b:  0f 05                  syscall
```

x86

aarch64

```
calc:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:  52800100   mov w0, #0x8                    // #8
  40007c:  528000e1   mov w1, #0x7                    // #7
  400080:  0b000020   add w0, w1, w0
  400084:  52800ba8   mov w8, #0x5d                   // #93
  400088:  d4000001   svc #0x0
```

Higher level language

- Assembler-instruksene er spesifikt til hver CPU-arkitektur
  (det som fysisk er implementert)
- C som et portabelt språk, som oversettes i assembly-kode av en "compiler"

```c
int mult(int a, int b) {
    return a*b;
}
```

x86_64

```
mult:
        movl    %edi, %eax
        imull   %esi, %eax
        ret
```

arm64

```
mult:
        mul     r0, r1, r0
        bx      lr
```

- godbolt.org Compiler explorer

Demo

- https://godbolt.org/
- enkle funksjoner
- funksjonskall
- compiler-optimering (-Os)

Higher level language

```c
int mult(int a, int b) {
    return a*b;
}
```

x86_64

```
mult:
        movl    %edi, %eax
        imull   %esi, %eax
        ret
```

arm64

```
mult:
        mul     r0, r1, r0
        bx      lr
```

Konsepter som f.eks "funksjon", "for-løkke", finnes ***ikke*** i assembly.
Compileren kan bare bruke "jump" og "compare" - instrukser.

Må avtales hvor funksjonsargumentene skal ligge: f.eks

System V Application Binary Interface
AMD64 Architecture Processor Supplement

2. If the class is INTEGER, the next available register of the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` is used[19].

Higher level language

```c
int mult(int a, int b) {
    return a*b;
}
```

x86_64

```
mult:
        movl    %edi, %eax
        imull   %esi, %eax
        ret
```

arm64

```
mult:
        mul     r0, r1, r0
        bx      lr
```

Konsepter som f.eks "funksjon", "for-løkke", finnes *ikke* i assembly.
Compileren kan bare bruke "jump" og "compare" - instrukser.



Må avtales hvor funksjo

stem V Application Binary Interface
MD64 Architecture Processor Supplement

2. If the class is INTEGER
   `%rcx`, `%r8` and `%r9` is use

, `%rdx`,

3. If the class is SSE, the
   in the order from `%xmm0` to `%xmm7`.

taken

C language

--- compiles to ----

```
gcc -s -o foo.S foo.c
```

Assembly language

---- assembles into -----

```
as -o foo.o  foo.S
```

machine code

----- linked to other machine code ----

```
ld -o foo  foo.o bar.o baz.o
```

gcc/clang will do all steps in one:

```
gcc -o foo foo.c bar.o baz.o
```

program binary

---- loaded into / runs on ----

```
./foo
```

CPU

(instruction pointer at first address, fetch, decode, execute, write, etc...)

Linker

```
.code64
.section .text
.global _start

_start:
  mov $8, %rax
  mov $7, %rbx

  add %rax, %rbx

  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

Linker

```
.code64
.section .text
.global _start

_start:
  mov $8, %rax
  mov $7, %rbx

  call _adder

  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

```
.code64
.section .text
.global _adder

_adder:
  add %rax, %rbx
  ret
```



Linker

```
.code64
.section .text
.global _start

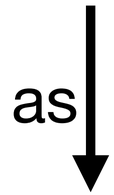
_start:
  mov $8, %rax
  mov $7, %rbx

  call _adder

  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

```
.code64
.section .text
.global _adder

_adder:
  add %rax, %rbx
  ret
```



ld:

```
calc.o
```



```
calc
```

```
$ ld calc.o -o calc
ld: calc.o: in function `_start':
(.text+0xf): undefined reference to `_adder'
```

Linker

```
.code64
.section .text
.global _start

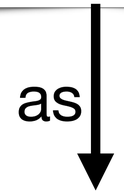
_start:
  mov $8, %rax
  mov $7, %rbx

  call _adder

  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

```
.code64
.section .text
.global _adder

_adder:
  add %rax, %rbx
  ret
```



calc.o

```
$ nm calc.o
                 U _adder
0000000000000000 T _start
```

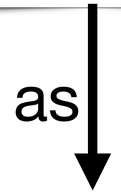
Linker

```
.code64
.section .text
.global _start

_start:
  mov $8, %rax
  mov $7, %rbx

  call _adder

  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

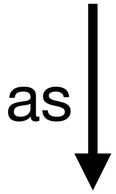


calc.o

```
$ nm calc.o
                 U _adder
0000000000000000 T _start
```

```
.code64
.section .text
.global _adder

_adder:
  add %rax, %rbx
  ret
```



mymath.o

```
$ nm mymath.o
0000000000000000 T _adder
```

Linker

```
.code64
.section .text
.global _start

_start:
  mov $8, %rax
  mov $7, %rbx

  call _adder

  mov $60, %rax
  mov %rbx, %rdi
  syscall
```

as

```
.code64
.section .text
.global _adder

_adder:
  add %rax, %rbx
  ret
```

as

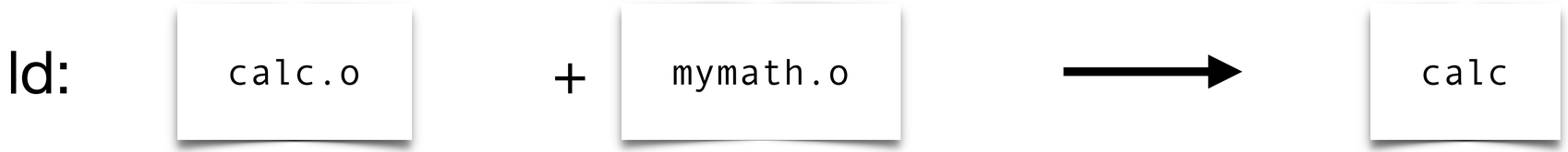
```
$ nm calc
000000000040101f T _adder
0000000000402000 T __bss_start
0000000000402000 T _edata
0000000000402000 T _end
0000000000401000 T _start
0000000000000000 T _adder
```

ld: calc.o + mymath.o → calc

```
$ nm calc.o
                 U _adder
0000000000000000 T _start
```

```
$ nm mymath.o
0000000000000000 T _adder
```

Linker tilpasser adressene




```
$ objdump -d calc.o

calc.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0: 48 c7 c0 08 00 00 00 mov    $0x8,%rax
   7: 48 c7 c3 07 00 00 00 mov    $0x7,%rbx
   e: e8 00 00 00 00       call   13 <_start+0x13>
  13: 48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
  1a: 48 89 df             mov    %rbx,%rdi
  1d: 0f 05                syscall
```

Linker tilpasser adressene

ld: calc.o + mymath.o → calc

```
$ objdump -d calc.o

calc.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:48 c7 c0 08 00 00 00 mov    $0x8,%rax
   7:48 c7 c3 07 00 00 00 mov    $0x7,%rbx
   e:e8 00 00 00 00       call   13 <_start+0x13>
  13:48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
  1a:48 89 df             mov    %rbx,%rdi
  1d:0f 05                syscall
```

```
$ objdump -r calc.o

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000f R_X86_64_PLT32    _adder-0x0000000000000004
```

Linker tilpasser adressene

ld: calc.o + mymath.o → calc

```
$ objdump -d calc.o

calc.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0: 48 c7 c0 08 00 00 00 mov    $0x8,%rax
   7: 48 c7 c3 07 00 00 00 mov    $0x7,%rbx
   e: e8 00 00 00 00       call   13 <_start+0x13>
  13: 48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
  1a: 48 89 df             mov    %rbx,%rdi
  1d: 0f 05                syscall
```

```
$ objdump -r calc.o

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000f R_X86_64_PLT32    _adder-0x0000000000000004
```

```
$ objdump -d calc

calc:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:  48 c7 c0 08 00 00 00 mov    $0x8,%rax
  401007:  48 c7 c3 07 00 00 00 mov    $0x7,%rbx
  40100e:  e8 0c 00 00 00       call   40101f <_adder>
  401013:  48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
  40101a:  48 89 df             mov    %rbx,%rdi
  40101d:  0f 05                syscall

000000000040101f <_adder>:
  40101f:  48 0f af d8          imul   %rax,%rbx
  401023:  c3                   ret
```

Linker tilpasser adressene

ld: calc.o + mymath.o → calc

```
$ objdump -d calc.o

calc.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0: 48 c7 c0 08 00 00 00 mov    $0x8,%rax
   7: 48 c7 c3 07 00 00 00 mov    $0x7,%rbx
   e: e8 00 00 00 00       call   13 <_start+0x13>
  13: 48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
  1a: 48 89 df             mov    %rbx,%rdi
  1d: 0f 05                syscall
```

```
$ objdump -d calc

calc:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000: 48 c7 c0 08 00 00 00 mov    $0x8,%rax
  401007: 48 c7 c3 07 00 00 00 mov    $0x7,%rbx
  40100e: e8 0c 00 00 00       call   40101f <_adder>
  401013: 48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
  40101a: 48 89 df             mov    %rbx,%rdi
  40101d: 0f 05                syscall

000000000040101f <_adder>:
  40101f: 48 0f af d8          imul   %rax,%rbx
  401023: c3                   ret
```

0x13 + 0x0c = 0x1f

old IC *new IC*

C language

--- compiles to ----

```
gcc -s -o foo.S foo.c
```

Assembly language

---- assembles into -----

```
as -o foo.o  foo.S
```

machine code

----- linked to other machine code ----

```
ld -o foo  foo.o bar.o baz.o
```

gcc/clang will do all steps in one:

```
gcc -o foo foo.c bar.o baz.o
```

program binary

---- loaded into / runs on ----

```
./foo
```

CPU

(instruction pointer at first address, fetch, decode, execute, write, etc...)

Program eller Process?

```
#include <stdio.h>

int main() {
  for (int i=0; i<5; i++) {
    printf("Hello\n");
  }
}
```

Program:

sekvens av instrukser
lagret et sted

```
0000000100000f50 <_main>:
100000f50: 55                          pushq   %rbp
100000f51: 48 89 e5                    movq    %rsp, %rbp
100000f54: 48 83 ec 10                 subq    $16, %rsp
100000f58: c7 45 fc 00 00 00 00        movl    $0, -4(%rbp)
100000f5f: c7 45 f8 00 00 00 00        movl    $0, -8(%rbp)
100000f66: 83 7d f8 05                 cmpl    $5, -8(%rbp)
100000f6a: 0f 8d 1c 00 00 00           jge     0x100000f8c <_main+0x3c>
100000f70: 48 8d 3d 24 00 00 00        leaq    36(%rip), %rdi
100000f77: b0 00                       movb    $0, %al
100000f79: e8 17 00 00 00              callq   0x100000f95 <_printf+0x100000f95>
100000f7e: 8b 45 f8                    movl    -8(%rbp), %eax
100000f81: 83 c0 01                    addl    $1, %eax
100000f84: 89 45 f8                    movl    %eax, -8(%rbp)
100000f87: e9 da ff ff ff              jmp     0x100000f66 <_main+0x16>
100000f8c: 8b 45 fc                    movl    -4(%rbp), %eax
100000f8f: 48 83 c4 10                 addq    $16, %rsp
100000f93: 5d                          popq    %rbp
100000f94: c3                          retq
```

```
Hello
Hello
Hello
Hello
Hello
```

Process:

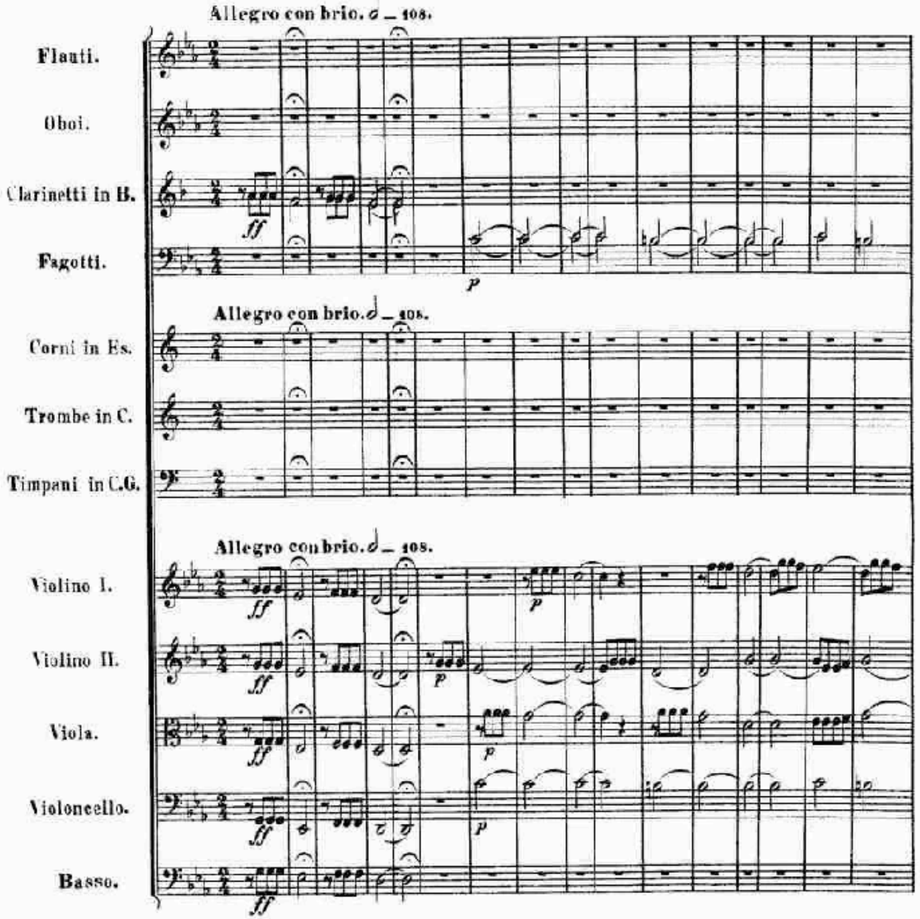
program under kjøring

Beethoven

Symphony No. 5

in C Minor

Op. 67



Program



https://en.wikipedia.org/wiki/File:MITO_Orchestra_Sinfonica_RAI.jpg

Process

demo

- ps
- top (q to quit)
- ls /proc

- kill

Python

- python-programmet er en *interpreter* som tolker python-språket
- skrevet i C; kompilert til hver CPU-arkitektur
  - objdump -d /usr/lib/libpython3.14.so.1.0 | less
  - strings /usr/lib/libpython3.14.so.1.0 | less
- (forenklet:) leser en linje, utfører instruksene, videre til neste...

A first interpreter...

```
#include <stdio.h>
#include <string.h>

int main() {
  for (;;) {
    int a, b, c;
    char op[2];

    printf("Add or sub two numbers: ");
    scanf("%d %1s%d",&a,op,&b);

    if (op[0] == '+') {
      c = a+b;
    }
    else if (op[0] == '-') {
      c = a-b;
    }
    else {
      a=b=c=0;
      strcpy(op,"?");
    }

    printf("\n%d %s %d = %d\n\n\n", a, op, b, c);
  }
}
```

Ikke ta dette på alvor, har masse problemer som må fikses!

fra hello.py to CPU

print("Hello World")

Python-interpreter

C (could be any of many high-level langs)

Assembly

Machine code instructions

CPU

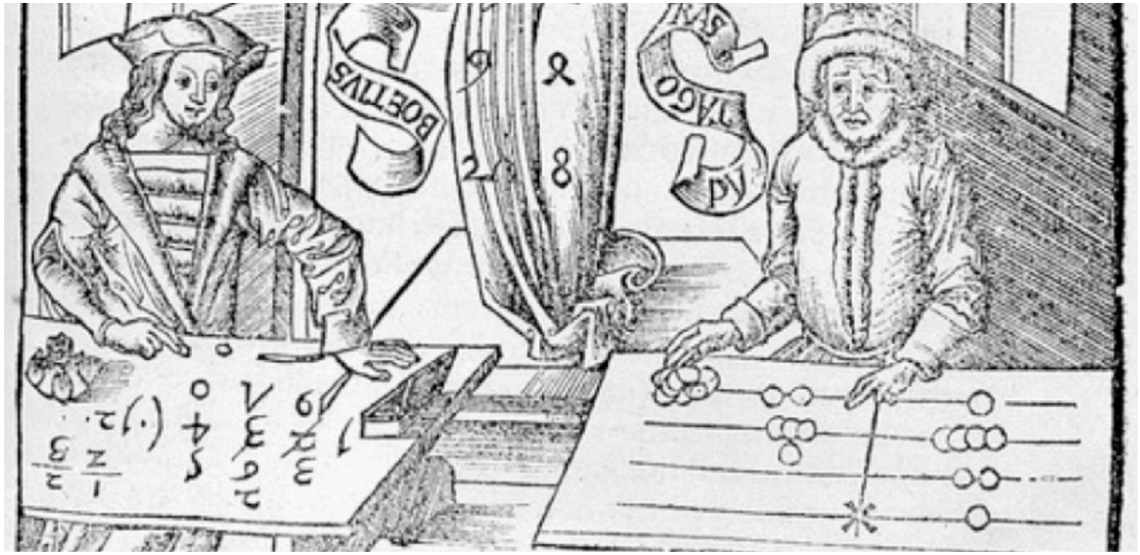
Tidsbruk

```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns
Main memory reference ...................... 100 ns
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Read 1 MB sequentially from SSD  ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

https://gist.github.com/hellerbarde/2843375

Tidsbruk

L1 cache reference	0.5 s
Branch mispredict	5 s
L2 cache reference	7 s
Main memory reference	100 s
SSD random read	1.7 days
Read 1 MB sequentially from memory	2.9 days
Read 1 MB sequentially from SSD	11.6 days
Disk seek	16.5 weeks
Read 1 MB sequentially from disk	7.8 months
Send packet CA->Netherlands->CA	4.8 years



https://gist.github.com/hellerbarde/2843375