Hoe rekend een computer cpu optellen, aftreken, vermenigvuldigen, delen, machtverheffing, vierkandwortel

Een **CPU rekent alles uiteindelijk met hele simpele bouwstenen**: **bits** (0 en 1), **logische schakelingen** en **klokpulsen**. Hieronder leg ik per bewerking uit **hoe dat intern gebeurt**, stap voor stap, zonder te technisch te worden.

1️⃣ Optellen

Binaire optelling

Een CPU telt op in **binair** (0 en 1):

Voorbeeld:

```
 1011   (11)
+ 0101   (5)
-------
 10000  (16)
```

Half-adder & Full-adder

- **Half-adder**: telt 2 bits op
- **Full-adder**: telt 2 bits + een carry (overdracht) op

Deze zijn opgebouwd uit **logische poorten**:

- AND
- OR
- XOR

➡️ Een 64-bit CPU heeft **64 full-adders achter elkaar**.

2️⃣ Aftrekken

Aftrekken = **optellen met negatieve getallen**

Two's complement

CPU's gebruiken **two's complement**:

1. Keer bits om
2. Tel 1 erbij op

Voorbeeld:

5 – 3 = 2

5 = 00000101

-3 = 11111101 (two's complement)

Dan:

5 + (-3) = 2

➡️ De **zelfde optelhardware** wordt gebruikt!

3️⃣ Vermenigvuldigen

Eenvoudig idee

Vermenigvuldigen = **herhaald optellen**:

6 × 4 = 6 + 6 + 6 + 6

In hardware

CPU gebruikt:

- **shift (schuiven)**
- **optellen**

Voorbeeld:

13 × 3 = (13 schift 1 keer) + 13

Moderne CPU's hebben een **multiplier unit** die dit extreem snel parallel doet.

4️⃣ Delen

Delen = **herhaald aftrekken + verschuiven**

Voorbeeld:

20 ÷ 4

20 - 4 - 4 - 4 - 4 - 4 = 0

In hardware

- Vergelijken
- Aftrekken
- Bits schuiven

➡️ Dit kost meer klokcycli dan optellen of vermenigvuldigen.

5️⃣ Machtverheffing (bijv. x^n)

Integer machten

$2^3 = 2 \times 2 \times 2$

CPU doet:

- herhaald vermenigvuldigen
- of **exponentiation by squaring** (sneller)

Voorbeeld:

$x^8 = ((x^2)^2)^2$

➡️ Wordt meestal door **software** geregeld, niet speciale hardware.

6️⃣ Vierkantswortel

Geen simpele hardware-truc

Vierkantswortel gebeurt via **benaderingen**.

Veelgebruikte methodes:

- Newton-Raphson methode
- Digit-by-digit algoritme

Voorbeeld Newton:

$x_{n+1} = (x_n + a/x_n) / 2$

CPU herhaalt:

- delen
- optellen

Moderne CPU's hebben soms een **sqrt-instructie** in de FPU (floating-point unit).

🔢 Floating-point (kommagetallen)

Kommagetallen gebruiken:

- **IEEE 754 standaard**

- aparte **FPU**

Bijvoorbeeld:

3.14 = teken + exponent + mantisse

➡️ Berekeningen zijn complexer dan bij gehele getallen.

🧠 Samenvatting

Bewerking	**Hoe CPU het doet**
Optellen	Logische poorten
Aftrekken	Two's complement
Vermenigvuldigen	Shift + optellen
Delen	Shift + aftrekken
Macht	Herhaald vermenigvuldigen
Wortel	Iteratieve benadering

🧠 Hoe een CPU rekent – VISUEEL

🔹 Basis: bits & logische poorten

Alles is **0 of 1** (uit = aan).

0 = geen spanning

1 = spanning

Logische poorten:

AND-poort

A ─┐

  AND ──▶ output

B ─┘

XOR-poort (belangrijk voor optellen)

0 XOR 0 = 0

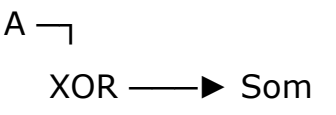
0 XOR 1 = 1

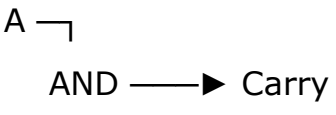
1 XOR 0 = 1

1 XOR 1 = 0

1️⃣ Optellen (Full Adder)

Eén bit optellen





Full Adder (met carry)



Carry = (A AND B) OR (Carry AND (A XOR B))

8-bit optelling (bijv. 13 + 7)

```
  00001101
+ 00000111
  ----------
  00010100
```

Visueel in hardware:

[FA][FA][FA][FA][FA][FA][FA][FA]

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

bit0 → bit7 (carry schuift door)

➡️ Elke [FA] = Full Adder

2️⃣ Aftrekken (two's complement)

7 − 3

Stap 1: maak −3

3 = 00000011

NOT = 11111100

+1 = 11111101 ← -3

Stap 2: optellen

 00000111 (7)

+11111101 (-3)

 00000100 = 4

➡️ **Dezelfde opteller!**

3️⃣ Vermenigvuldigen (shift + add)

6 × 5

5 = 00000101

6 × 5 = 6 + 6 + 6 + 6 + 6

6 × (4 + 1)

Visueel:

00000110 (6)

Schift 0 keren = 00000110

Schift 2 keren = 00011000 (6 Schift 12 Schift 24)

+00011000 (24) (6x4) (6+6+6+6)

+00000110 (6)

 00011110 (30) (6x5) (6+6+6+6+6)

CPU gebruikt:

[Shift] → [Adder]

4️⃣ Delen (aftrekken + vergelijken)

20 ÷ 4

20 → 16 → 12 → 8 → 4 → 0

Visueel:

[Compare] → [Subtract] → [Shift]

Diagram:

20 ──┐

▼

[-4] → 16 → 12 → 8 → 4 → 0

➡️ Langzamer dan optellen!

5️⃣ Machtverheffing (bijv. 2^8)

Slimme methode:

2^8

$= (2^2)^2)^2$

Visueel:

2 → 4 → 16 → 256

Hardware:

[Multiplier] → [Register] → [Multiplier]

➡️ Meestal **software**, geen aparte hardware.

6️⃣ Vierkantswortel (benaderen)

√25

Start met gok:

x = 5

Formule:

x = (x + 25/x) / 2

Diagram:

[Guess]

↓

[Divide]

↓

[Add]

↓

[Divide by 2]

↓

[New Guess]

Herhaal tot stabiel.

🔢 Kommagetallen (FPU)

Floating-point getal:

+ 1.101 × 2^3

Opslag:

[Sign][Exponent][Mantissa]

CPU gebruikt aparte eenheid:

CPU

├─ ALU (integers)

└─ FPU (kommagetallen)

🧩 Grote samenvatting

Transistors

↓

Logische poorten

↓

Adders / Shifters

↓

ALU / FPU

↓

ALLE BEREKENINGEN

🧠 CPU-berekeningen in Assembly (Visueel & Stap-voor-stap)

🔹 Registers (denk aan snelle opslag)

RAX, RBX, RCX, RDX = getallen

1️⃣ Optellen

5 + 3

```
mov rax, 5      ; zet 5 in RAX
add rax, 3      ; RAX = RAX + 3
```

Visueel:

RAX: 5 ──add──▶ 8

2️⃣ Aftrekken

7 − 4

```
mov rax, 7
sub rax, 4
```

Intern:

4 → two's complement → + optellen

3️⃣ Vermenigvuldigen

6 × 5

```
mov rax, 6
mov rbx, 5
imul rbx        ; RAX = RAX * RBX
```

Resultaat:

RAX = 30

Visueel:

RAX ─┐

× ──▶ 30

RBX ─┘

4️⃣ Delen

20 ÷ 4

```
mov rax, 20
xor rdx, rdx    ; maak RDX = 0 (vereist)
mov rbx, 4
idiv rbx        ; RAX = quotiënt, RDX = rest
```

Resultaat:

RAX = 5

RDX = 0

5️⃣ Machtverheffing (x^3)

$2^3 = 8$

```
mov rax, 2
imul rax, 2     ; 2²
imul rax, 2     ; 2³
```

Sneller (conceptueel):

```
mov rax, 2
imul rax, rax   ; 2²
imul rax, rax   ; 2⁴
```

➡️ Geen aparte pow-instructie!

6️⃣ Vierkantswortel (integer, software)

√16

```
mov rcx, 1      ; gok
.loop:
mov rax, 16
```

```
xor rdx, rdx
idiv rcx        ; rax = 16 / gok
add rax, rcx
shr rax, 1      ; /2
cmp rax, rcx
je  .done
mov rcx, rax
jmp .loop
.done:
```

➡️ Iteratief, zoals eerder uitgelegd.

7️⃣ Kommagetallen (FPU / SSE)

3.0 + 4.5

```
movsd xmm0, [three]
movsd xmm1, [four_point_five]
addsd xmm0, xmm1
Data:
three:          dq 3.0
four_point_five dq 4.5
```

Visueel:

XMM0 = 3.0

XMM1 = 4.5

XMM0 = 7.5

8️⃣ Alles samen (mini-programma)

(5 + 3) × 2

```
mov rax, 5
add rax, 3
imul rax, 2
```

Resultaat:

RAX = 16

🔌 Van Assembly → Hardware

Assembly instructie

↓

Micro-instructies

↓

ALU / FPU

↓

Transistors

Bijv.:

add rax, rbx

Wordt intern:

[Full Adder][Full Adder][Full Adder]...

🧩 Belangrijk om te onthouden

Assembly	Hardware
ADD	Full adders
SUB	Two’s complement
IMUL	Shift + add
IDIV	Aftrekken + vergelijken
SQRTSD	Iteratieve hardware
MOV	Register → register

🔁 Van C → Assembly → Hardware

🔹 Vooraf: wat doet een compiler?

Een compiler:

1. Analyseert je C-code
2. Zet die om naar **assembly**
3. Optimaliseert (sneller, minder instructies)
4. Zet assembly om naar **machinecode (bits)**

1️⃣ Optellen

C-code

```
int add(int a, int b) {
    return a + b;
}
```

Assembly (x86-64, sterk vereenvoudigd)

```
add:
    mov eax, edi    ; eax = a
    add eax, esi    ; eax = eax + b
    ret
```

Wat gebeurt hier?

a → register EDI

b → register ESI

EDI ──► EAX ──add──► resultaat

ESI ──────────────────▲

➡️ add gebruikt **exact dezelfde full adders** als eerder uitgelegd.

2️⃣ Aftrekken

C-code

```
int sub(int a, int b) {
    return a - b;
}
```

Assembly

```
sub:
    mov eax, edi
    sub eax, esi
    ret
```

➡️ Intern:

b → two's complement → optellen

3️⃣ Vermenigvuldigen

C-code

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

Assembly

```
mul:
    mov eax, edi
    imul eax, esi
    ret
```

Hardware

[Multiplier Unit]

↳ shift + add (parallel)

➡️ Eén C-operator → één CPU-instructie

4️⃣ Delen

C-code

```
int divi(int a, int b) {
    return a / b;
}
```

Assembly

```
divi:
```

```
    mov eax, edi
    cdq             ; maak EDX:EAX klaar
    idiv esi
    ret
```

Resultaat:

EAX = quotiënt

EDX = rest

➡️ Delen = **veel microstappen**, daarom traag.

5️⃣ Machtverheffing (geen operator!)

C-code

```
int power(int x) {
    return x * x * x;
}
```

Assembly

```
power:
    mov eax, edi
    imul eax, edi   ; x²
    imul eax, edi   ; x³
    ret
```

➡️ Compiler **rolt de macht uit**
➡️ Er bestaat **geen** pow-instructie voor integers

6️⃣ Slimme compiler-optimalisatie

C-code

```
int times8(int x) {
    return x * 8;
}
```

❌ **Slecht (maar logisch)**

imul eax, 8

✅ **Slim (compiler-optimalisatie)**

shl eax, 3   ; shift left 3 bits = ×8

➡️ **Shift is veel sneller dan vermenigvuldigen**

7️⃣ Vierkantswortel (double)

C-code

```
double root(double x) {
    return sqrt(x);
}
```

Assembly

```
sqrtsd xmm0, xmm0
ret
```

➡️ Gebruikt de **FPU**
➡️ Intern: iteratieve benadering in hardware

8️⃣ Volledig voorbeeld

C-code

```
int calc(int x) {
    return (x + 3) * 2;
}
```

Assembly (geoptimaliseerd)

```
calc:
    lea eax, [rdi + 3]  ; x + 3
    shl eax, 1          ; ×2
    ret
```

Wat zie je hier?

- ❌ Geen add
- ❌ Geen imul

- ✅ Slim adresrekenen (lea)
- ✅ Shift i.p.v. multiply

➡️ **Compilers zijn slim** 😄

🧠 Van C-regel tot transistor

C-code

↓

Abstract Syntax Tree

↓

Assembly

↓

Machinecode (bits)

↓

Micro-ops

↓

ALU / FPU

↓

Transistors schakelen

Bijvoorbeeld:

a + b

Wordt uiteindelijk:

64 full adders

+ carry propagation

+ klokpulsen

🔚 Samenvatting

Niveau	Voorbeeld
C	x * 8

Niveau	Voorbeeld
Assembly	shl eax, 3
Hardware	bits schuiven
Elektronica	transistors

Exacte compiler output (GCC/Clang -O0 vs -O2)

Ik gebruik **x86-64 Linux (System V ABI)** en **GCC/Clang-stijl AT&T/Intel gemengd**, maar ik leg alles uit zodat het leesbaar blijft.

🔧 Voorbeeldprogramma (C)

```
int calc(int x) {
    return (x + 3) * 2;
}
```

Compile-commando:

```
gcc -S calc.c -O0
gcc -S calc.c -O2
```

🟥 GCC met -O0 (geen optimalisatie)

Dit is **typisch wat je echt ziet**:

```
calc:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    add     eax, 3
    add     eax, eax
    pop     rbp
    ret
```

Wat gebeurt hier?

Stack frame (onnodig, maar duidelijk)

push rbp

mov rbp, rsp

Variabele x wordt op stack gezet

[rbp-4] = x

Rekenen

eax = x

eax = eax + 3

eax = eax * 2

➡️ **Elke C-stap blijft zichtbaar**
➡️ Makkelijk te debuggen
➡️ Traag en veel instructies

🟩 **GCC met -O2 (geoptimaliseerd)**

```
calc:
    lea     eax, [rdi+3]
    add     eax, eax
    ret
```

of soms:

```
calc:
    lea     eax, [rdi+rdi+6]
    ret
```

Wat zie je hier?

- ❌ Geen stack
- ❌ Geen lokale variabele
- ❌ Geen aparte add + mul
- ✅ Alles in registers
- ✅ Slimme instructiekeuze

lea truc

lea eax, [rdi+3] = x + 3

add eax, eax   = ×2

➡️ **Zelfde resultaat, ~3× minder werk**

⚖️ **Vergelijking -O0 vs -O2**

	-O0	-O2
Stack frame	Ja	Nee
Lokale variabelen	Op stack	Registers
Instructies	Veel	Weinig
Debugbaar	Ja	Moeilijk
Snelheid	Laag	Hoog

🔢 **Tweede voorbeeld: vermenigvuldiging**

C-code

```
int mul8(int x) {
    return x * 8;
}
```

GCC -O0

```
mul8:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    imul    eax, eax, 8
    pop     rbp
    ret
```

➡️ **Letterlijk * 8**

GCC -O2

```
mul8:
    lea     eax, [rdi*8]
    ret
of:
mul8:
    shl     eax, 3
    ret
```

➡️ **Shift i.p.v. vermenigvuldigen**

🔁 **Derde voorbeeld: functie-oproep**

C-code

```
int square(int x) {
    return x * x;
}
```

GCC -O0

```
square:
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    imul    eax, eax
    pop     rbp
    ret
```

GCC -O2

```
square:
```

```
    imul    eax, edi, edi
    ret
```

➡️ Eén instructie, klaar.

🧮 Floating-point voorbeeld

C-code

```
double add(double a, double b) {
    return a + b;
}
```

GCC -O0

```
add:
    push    rbp
    mov     rbp, rsp
    movsd   QWORD PTR [rbp-8], xmm0
    movsd   QWORD PTR [rbp-16], xmm1
    movsd   xmm0, QWORD PTR [rbp-8]
    addsd   xmm0, QWORD PTR [rbp-16]
    pop     rbp
    ret
```

➡️ Veel geheugenverkeer

GCC -O2

```
add:
    addsd   xmm0, xmm1
    ret
```

➡️ **FPU direct**

🧠 Wat leert dit je?

Compiler bij -O0:

- volgt C letterlijk
- bewaart alles in geheugen
- ideaal voor leren/debuggen

Compiler bij -O2:

- denkt als een CPU
- herschrijft wiskunde
- verwijdert alles wat niet nodig is
- gebruikt **ALU/FPU optimaal**

🪫 Van optimalisatie naar hardware

Bij -O2:

lea eax, [rdi+rdi+6]

Wordt in hardware:

Adder + Adder

↓

Full adders

↓

Transistors

De **vierkantswortel** is inderdaad het lastigste onderdeel, omdat het **fundamenteel anders** werkt dan optellen of vermenigvuldigen. Ik ga het **rustig, intuïtief en stap-voor-stap** uitleggen, met **visuele schema's**, en daarna pas iets technischer.

🧠 Hoe rekent een CPU de vierkantswortel?

❗ Belangrijk om eerst te weten

Een CPU:

- heeft **geen simpele "wortel-poort"**
- kan alleen:

- optellen
- aftrekken
- vermenigvuldigen
- delen
- bits schuiven

➡️ Een vierkantswortel wordt dus **benaderd**, niet "uitgerekend in één stap".

1️⃣ Het basisidee (mensentaal)

We zoeken een getal **x** zodat:

$x \times x \approx N$

Bijvoorbeeld:

$\sqrt{25} = 5$

Maar voor:

$\sqrt{10} \approx 3.162277...$

➡️ Er is **geen exact binair antwoord**, dus we moeten steeds **beter gokken**.

2️⃣ De belangrijkste methode: Newton–Raphson

Dit is wat moderne CPU's (in hardware!) gebruiken.

Formule:

$x_{n+1} = (x_n + N / x_n) / 2$

In woorden:

Neem een gok → deel → neem het gemiddelde → herhaal

3️⃣ Visueel voorbeeld: √25

Startgok

$x_0 = 10$

Iteratie 1

25 / 10 = 2.5

(10 + 2.5) / 2 = 6.25

Iteratie 2

25 / 6.25 = 4

(6.25 + 4) / 2 = 5.125

Iteratie 3

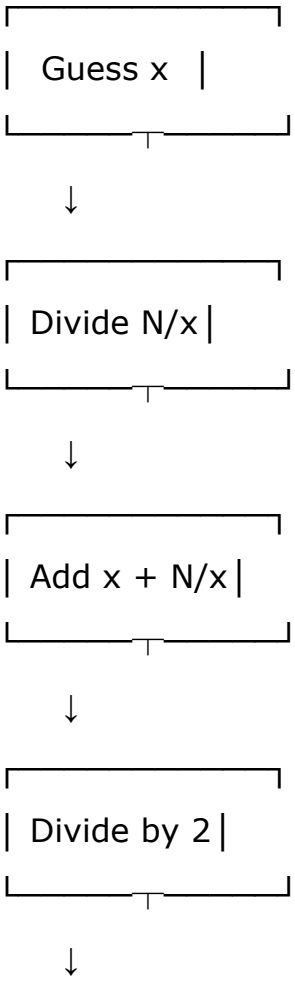
25 / 5.125 ≈ 4.88

(5.125 + 4.88) / 2 ≈ 5.002

➡️ **Convergeert razendsnel naar 5**

🔌 **Wat doet de CPU nu echt?**

In hardware ziet dit eruit als:



➡️ Dat is letterlijk wat er in de **FPU** gebeurt.

5️⃣ Maar hoe weet de CPU waar te beginnen?

Slimme truc: exponent-halvering

Bij **floating-point getallen** (IEEE-754):

N = mantissa × 2^exponent

Dan geldt:

√N = √mantissa × 2^(exponent/2)

➡️ De CPU:

1. halveert de exponent (bit-operatie!)
2. gebruikt een **kleine lookup table** voor een eerste gok

Dit maakt de start al **bijna goed**.

6️⃣ Hoeveel iteraties zijn nodig?

Type	Iteraties
float (32-bit)	2–3
double (64-bit)	3–5

➡️ Daarna is de fout **kleiner dan 1 bit**
➡️ Perfect volgens IEEE-754

7️⃣ Wat gebeurt bij sqrt() in C?

C-code

double y = sqrt(x);

Assembly

sqrtsd xmm0, xmm0

Intern (vereenvoudigd):

sqrtsd →

exponent halveren

lookup startwaarde

Newton-iteratie

afronden

➡️ **Alles in hardware**, geen loop in software.

8️⃣ Integer vierkantswortel (anders!)

Bij integers wordt vaak **bit-voor-bit** gewerkt.

Voorbeeld: √27

bit 4: 16 → te groot ❌

bit 3: 9 → past ✓

bit 2: 13 → te groot ❌

bit 1: 10 → past ✓

Resultaat:

√27 ≈ 5 (integer)

Hardware:

[Compare] → [Subtract] → [Shift]

➡️ Dit lijkt op **lange deling**.

9️⃣ Waarom is wortel "duur"?

Bewerking	**Cycli**
add	1
mul	3–5
div	10–20
sqrt	15–30

➡️ Wortel = meerdere delingen + afronden

🔚 Grote samenvatting (intuïtief)

Optellen → exact

Vermenigv → exact

Delen → exact (maar langzaam)

Wortel → slimme gok → steeds beter → klaar

Of nog simpeler:

🔍 **De CPU raadt, controleert, verbetert — supersnel**

Als afsluiter: mentaal beeld

Denk aan een **zoom-lens** 🔎:

- eerst vaag
- snel scherp
- dan perfect

Dat is precies hoe de CPU een vierkantswortel berekent.