

Computer Arkitektur

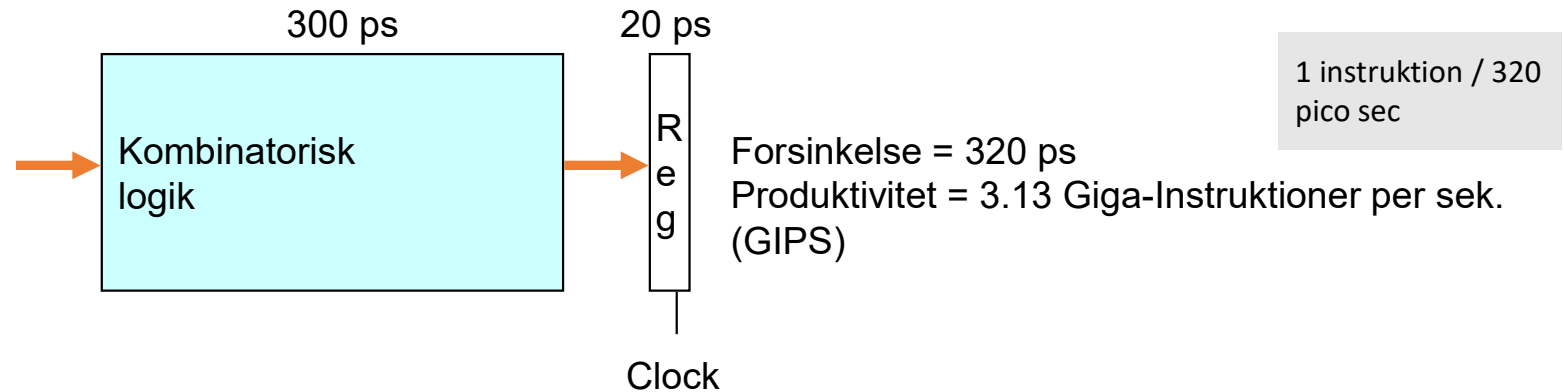
Instruktionsniveau-parallelisme og compiler optimeringer: Hvad compileren kan gøre og ikke kan gøre

Forelæsning 7
Brian Nielsen

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

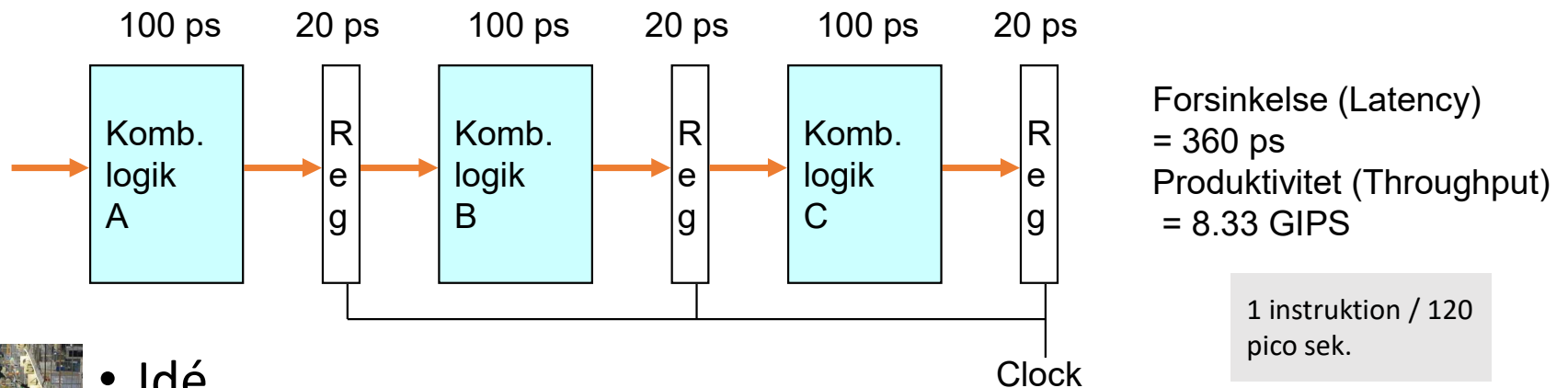
Pipelining Principper

Eksempel fra beregning i digital kredsløb



- Beregning i éet stort kombinatorisk kredsløb
 - Beregning kræver ialt 300 pico-sekunder
 - Plus 20 pico-sek. til at gemme resultatet i et register
 - Varigheden (perioden) på clock cyclus skal være mindst 320 ps
 - Frekvens= perioder/sek =
 $= 1/320\text{ps} = 1/320 \cdot 10^{-12} = 1000/320 \text{ GHz} = 3.13 \text{ GHz}$

3-trins Pipelined Udgave



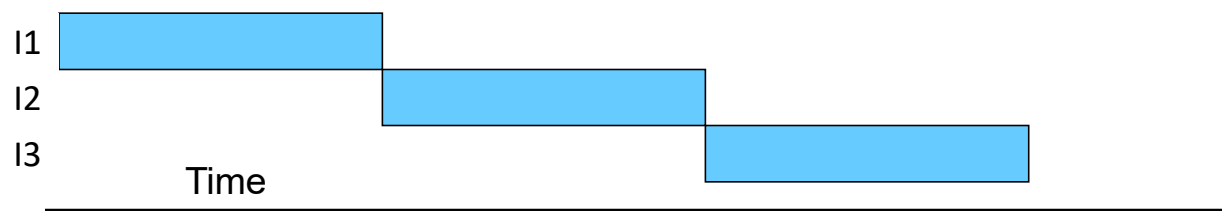
• Idé

- Del komplekst kombinatorisk logik ned i 3 blokke af hver (max) 100 ps
- Hver blok skal holdes adskilt af registre, så signaler fra forskellige instruktioner ikke blandes sammen
- Kan begynde ny instruktion når foregående har passeret igennem trin A.
 - Kan begynde ny opgave/instruktion hver 120 ps
- Samlede forsinkelse øges
 - 360 ps fra start til slut



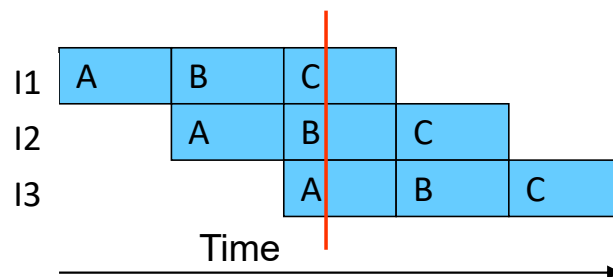
Pipeline Diagram

- Sekventielt



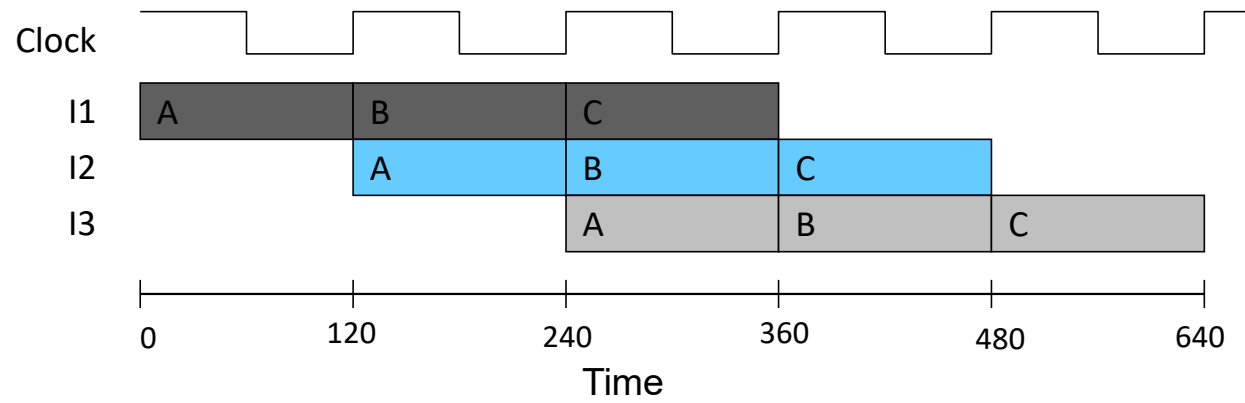
- Afvent start af ny operation til den foregående er afsluttet

- 3-trins Pipeline: Trin A,B,C

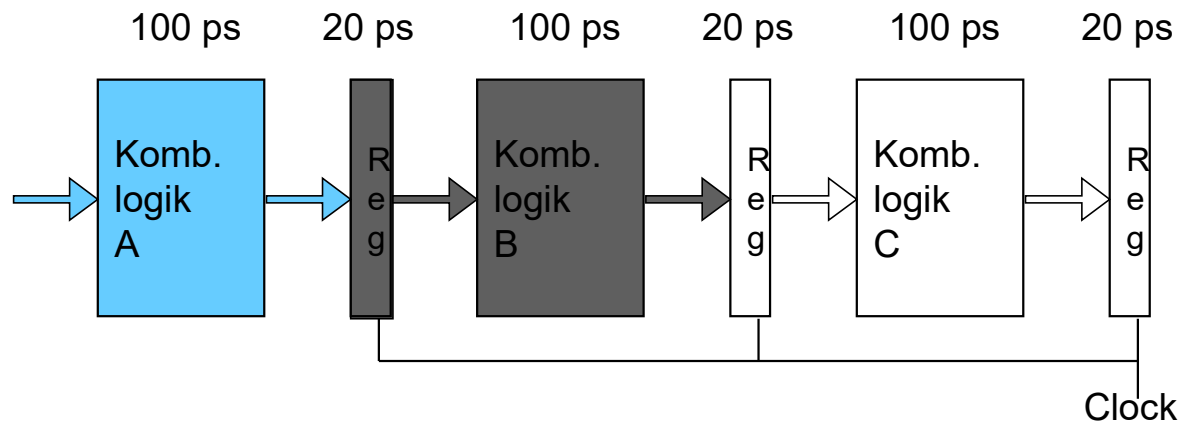
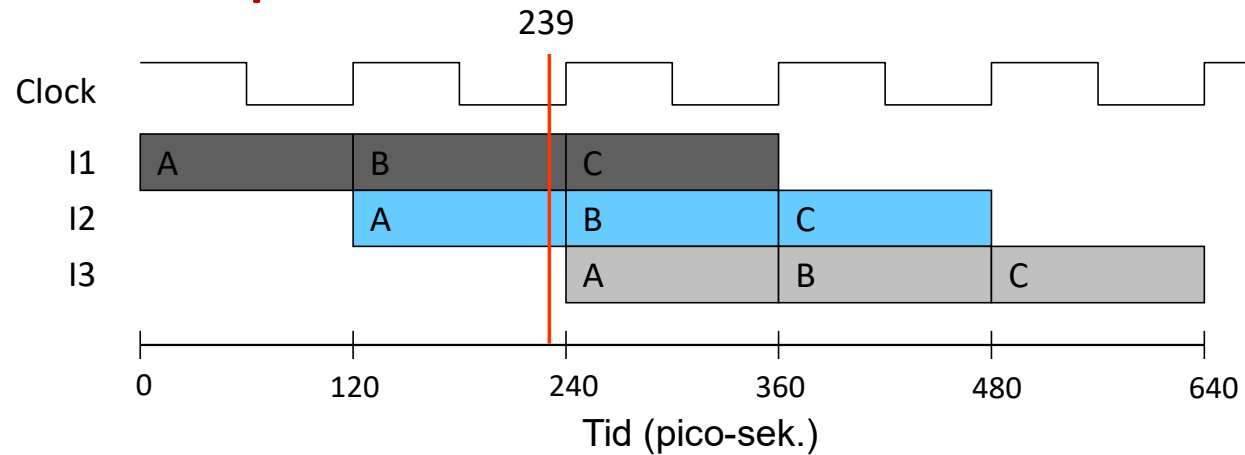


- Op til 3 operationer igang samtidigt; kun én operation i gang i hvert trin.

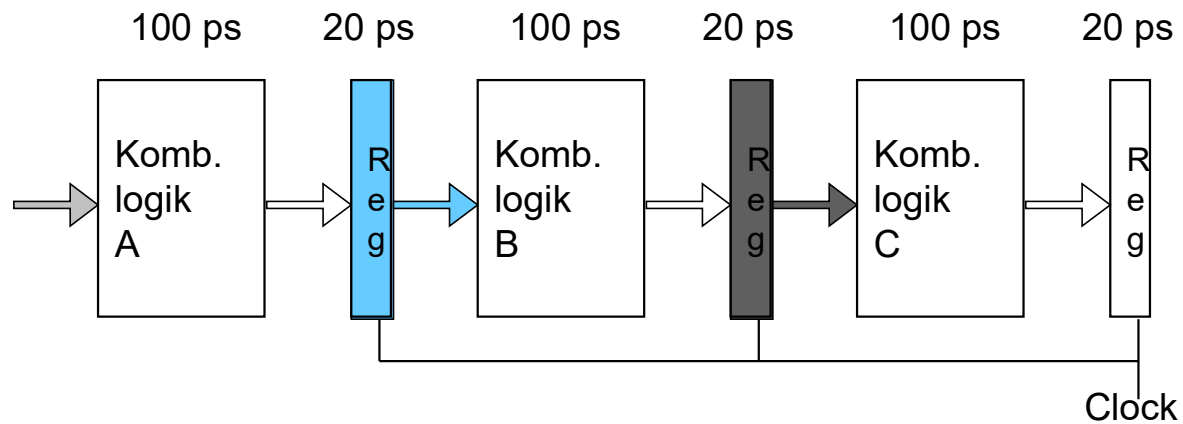
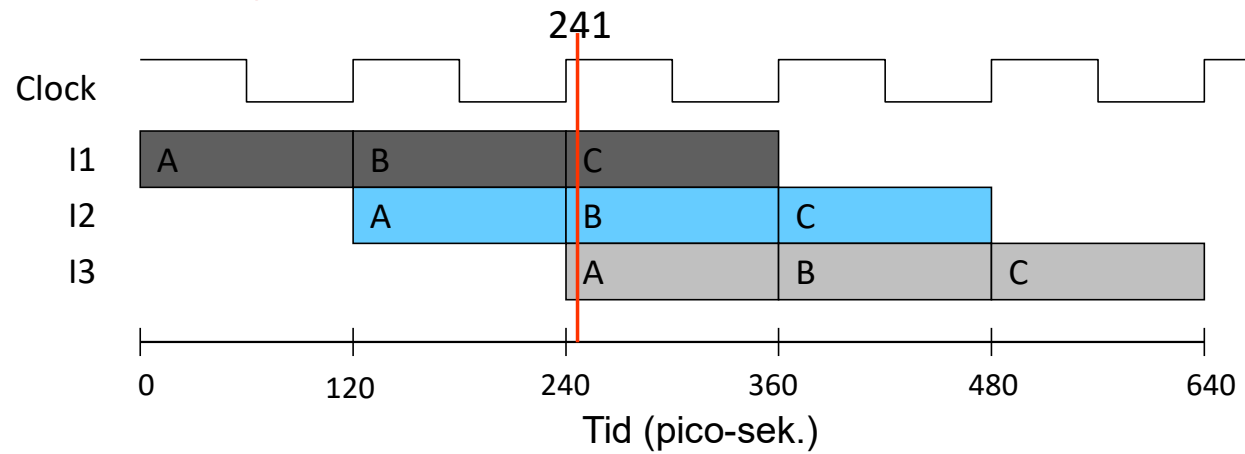
Udførelse i Pipeline



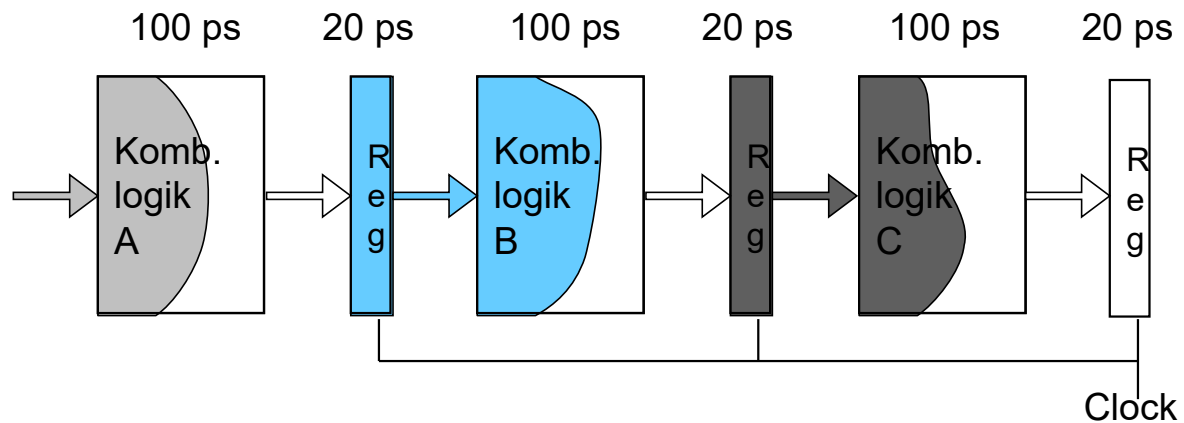
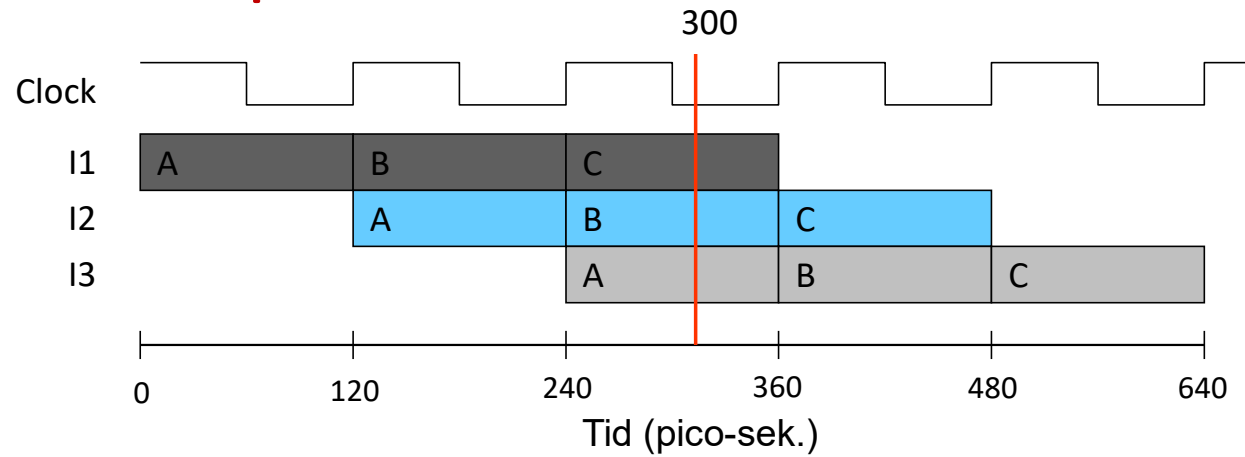
Udførelse i Pipeline 1



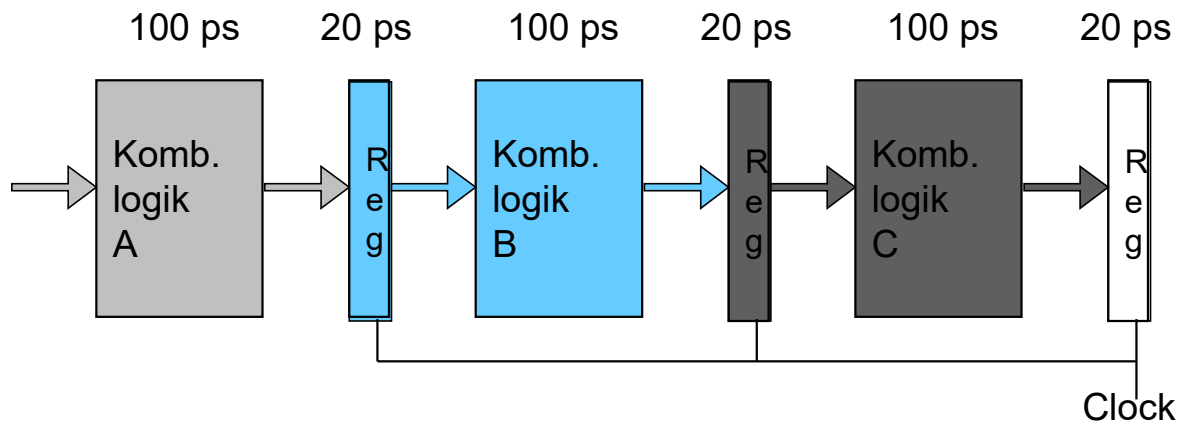
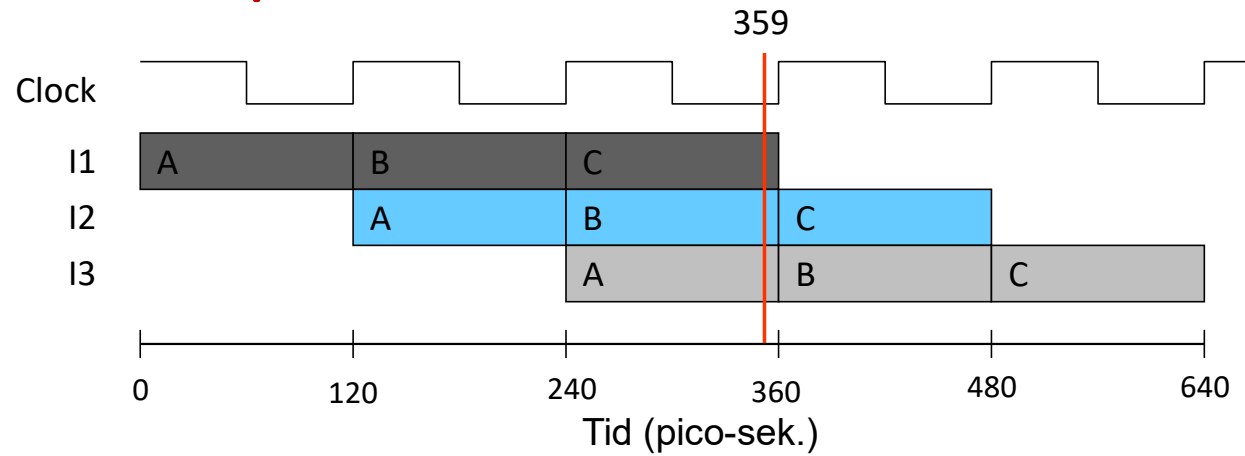
Udførelse i Pipeline 2



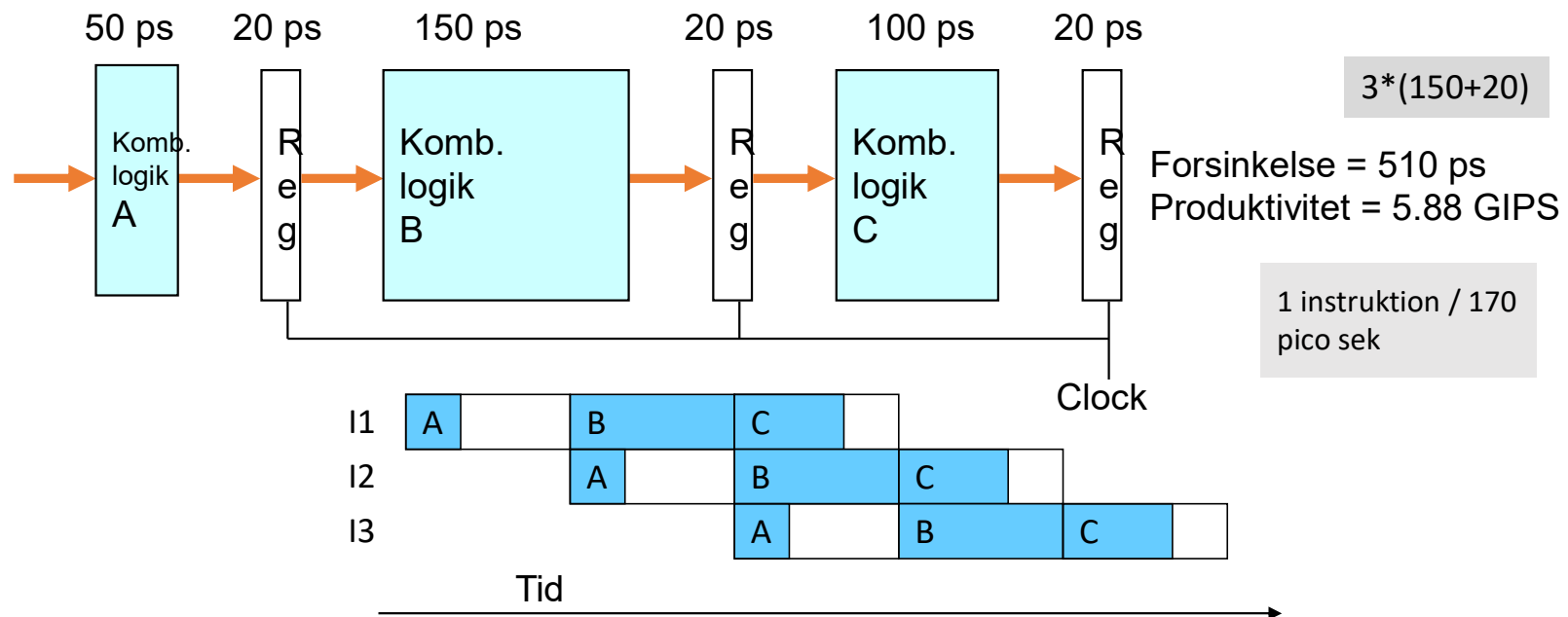
Udførelse i Pipeline 3



Udførelse i Pipeline 4

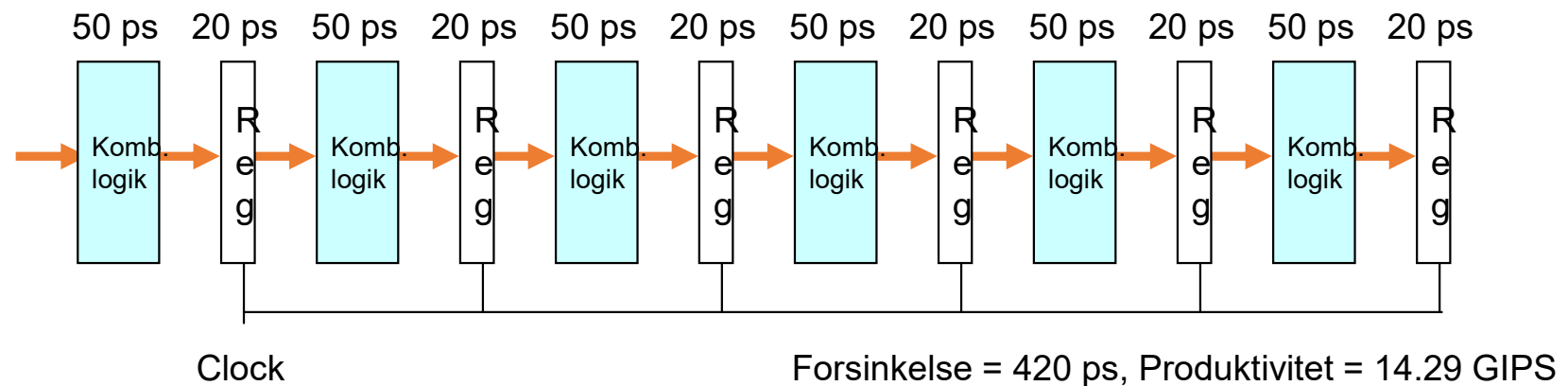


Problem 1: Forskellige Trin-størrelser



- Nogle trin kan være mere beregningskrævende end andre, dermed tage længere tid
- Produktivitet bestemmes af langsomste trin (flaskehals)
- Andre trin stages ligger stille ("idle") i stor del af tiden
- Det er udfordrende at opdele systemet i balancerede trin

Problem2: Register Overhead



- Des flere trin, des mere betyder tiden til indlæsning af registre
- Dybde af pipeline=antal trin
- Andel af en clock cyklus, der forbruges på indlæsning af register:
 - 1-trins pipeline: 6.25%
 - 3-trins pipeline: 16.67%
 - 6-trins pipeline: 28.57%
- Moderne processorer opnår høj hastighed ved brug af meget dyb pipeline
 - Haswell: 14

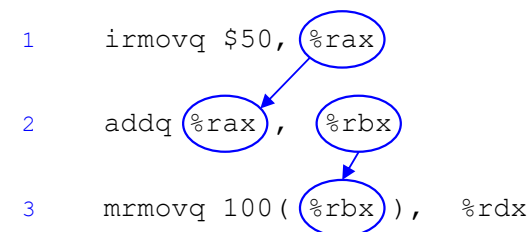
<https://softwareengineering.stackexchange.com/questions/210818/how-long-is-a-typical-modern-microprocessor-pipeline>

Problem: 3+4

Data-afhængigheder

- Resultat fra en instruktion bruges som operand til en anden
 - Read-after-write (RAW) afhængighed

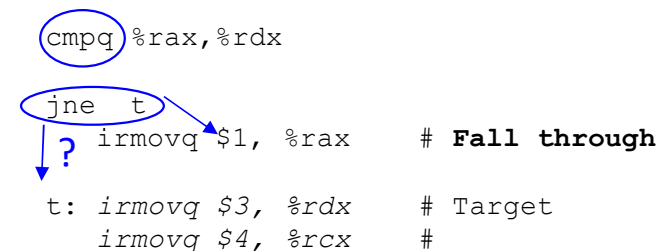
```
1  irmovq $50, %rax
2  addq %rax, %rbx
3  mrmovq 100(%rbx), %rdx
```



Kontrol afhængigheder

- Hvilken instruktion skal udføres næst afhænger af resultat af en tidligere

```
    cmpq %rax,%rdx
    jne t
    ? irmovq $1, %rax    # Fall through
    t: irmovq $3, %rdx    # Target
       irmovq $4, %rcx    #
```



En 5 trins model for Y86

I SEQ

- Trin beregnes i sekvens
- Én instruktion behandles ad gangen
- I én cyklus

I PIPE: 5 trin

- Fetch+newPC
- Decode
- Execute
- Memory
- Writeback

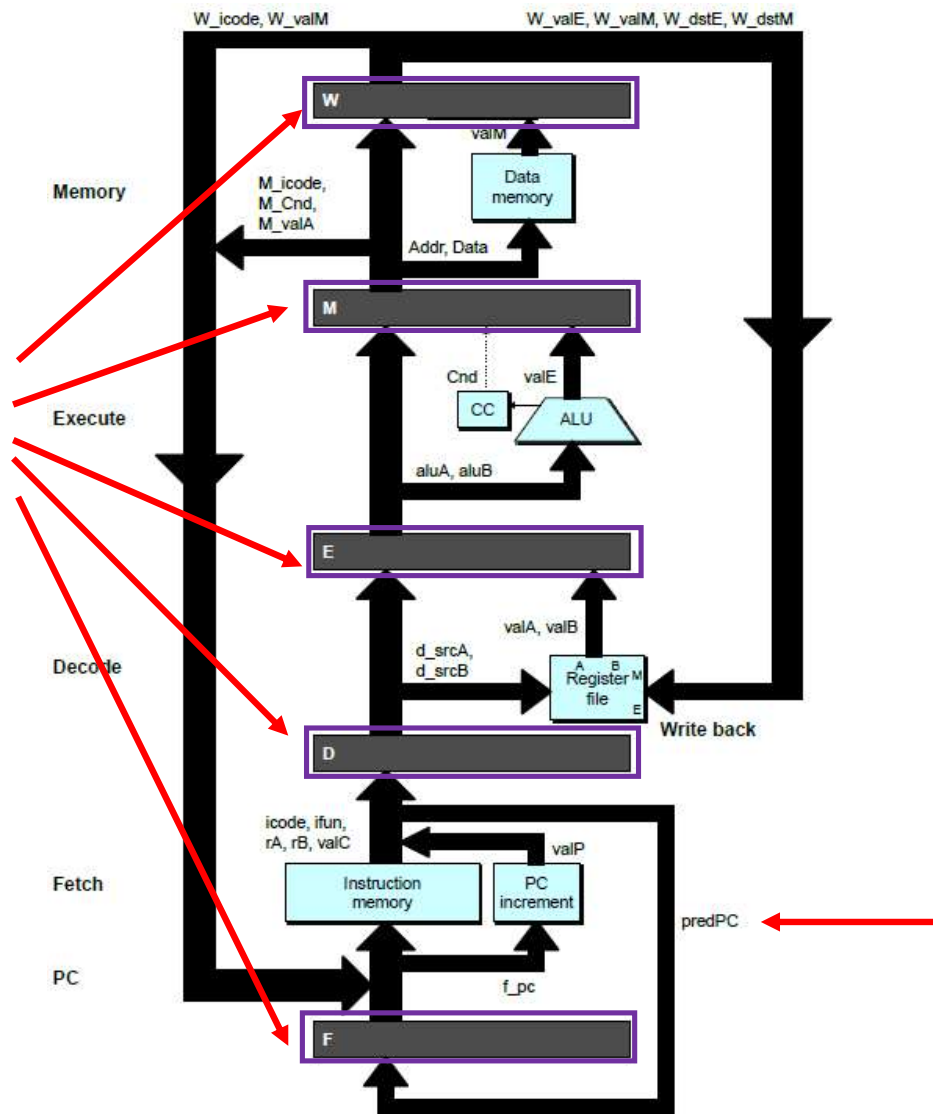
NB:

- Data fra hukk. først udlæst efter M-trin (4.)
- Cond. Codes kendes først efter E-trin (3.)

	OPq rA, rB // $OP \in \{\text{add, sub, and, xor}\}$	
Fetch	$\text{icode:ifun} \leftarrow M_1[PC]$ $\text{rA:rB} \leftarrow M_1[PC+1]$ $\text{valP} \leftarrow PC+2$	Læs instruktions byte Læs register byte Beregn næste PC
Decode	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$	Læs operand A Læs operand B
Execute	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC	Udfør ALU operation Sæt condition code register
Memory		
Write back	$R[rB] \leftarrow \text{valE}$	Skriv resultat tilbage
PC update	$PC \leftarrow \text{valP}$	Opdater PC

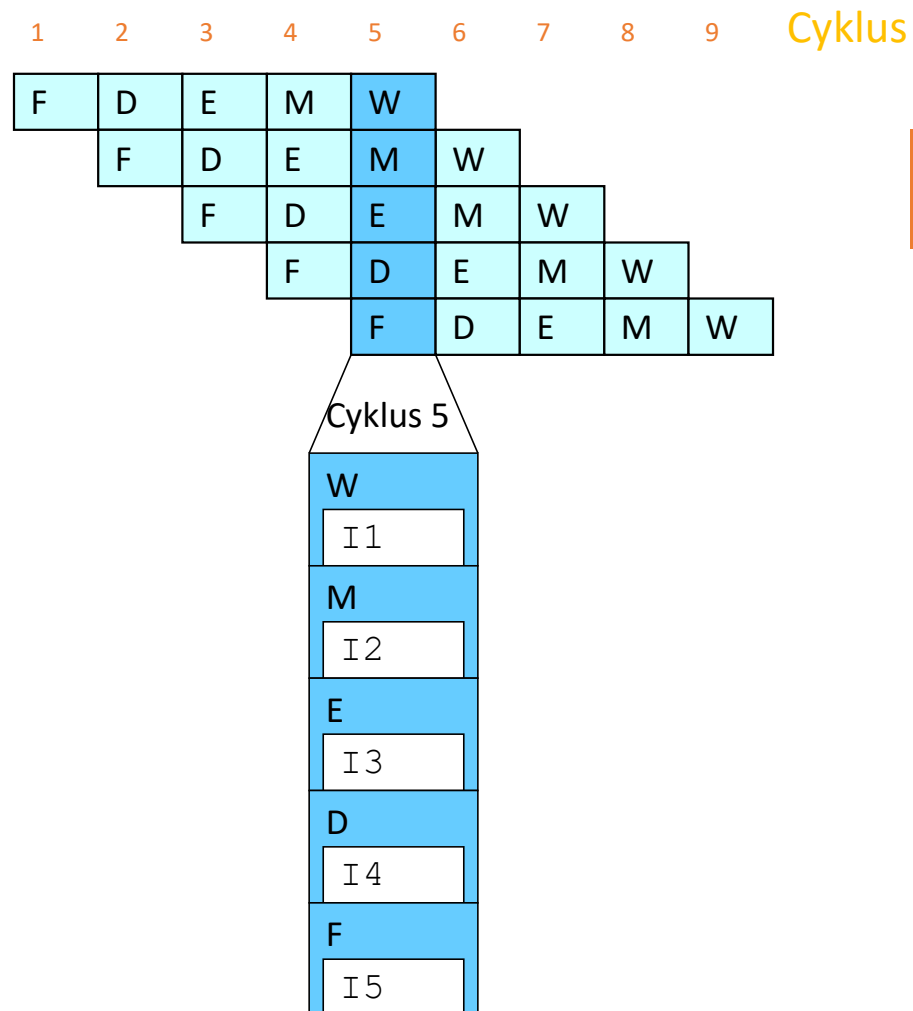
Opdeling i Pipeline: Indsæt Hardware Registre

- Pipeline registre adskiller trinene
- Pipeline register X gemmer inputs som trin X skal bruge (typisk mellemresultat fra foregående trin)
 - **F-register**: gemmer **spået (predicted)** værdi for PC
 - **D-register**: gemmer værdier fra seneste indlæste instruktion (fetched), som skal dekodes.
 - **E-register**: gemmer værdier fra seneste dekode instruktion (som skal udføres)
 - **M-register**: gemmer resultat af seneste instruktion i execute trin (herunder condition codes).
 - **W-register**: gemmer beregnede/ indlæste værdier som skal skrive tilbage i register bank
- Bemærk: **PC update** nu del af Fetch
 - Aktivt i starten af clock cyklus



Pipeline Demonstration

```
irmovq    $1,%rax    #I1
irmovq    $2,%rcx    #I2
irmovq    $3,%rdx    #I3
irmovq    $4,%rbx    #I4
halt      #I5
```



Pipeline
Diagram!

cart@ubuntu:~/sim/pipe\$./psim -g ../y86-code/basic.yo

Data-afhængigheder: 2 Nop's

Kun 2 "NOP"s

```
# demo-h2.js
```

```
0x000: irmovq $10,%rdx
```

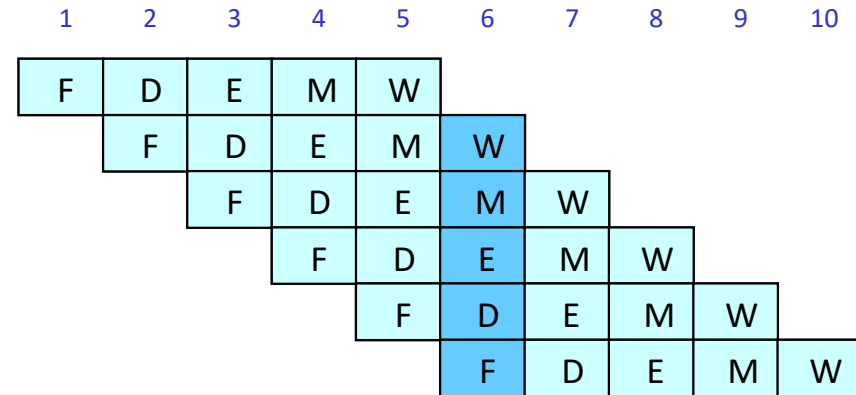
```
0x00a: irmovq $3,%rax
```

```
0x014: nop
```

```
0x015: nop
```

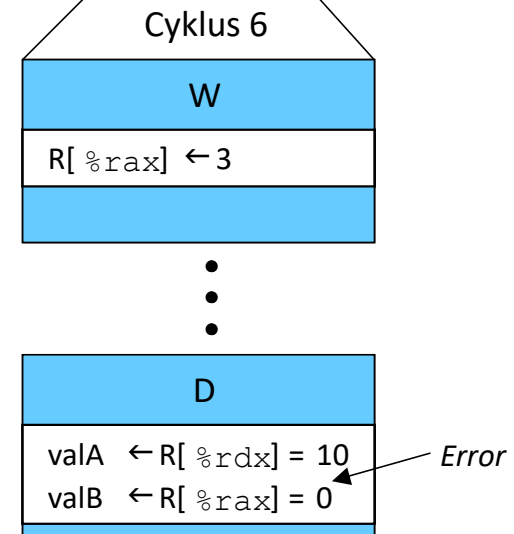
```
0x016: addq %rdx,%rax
```

```
0x018: halt
```



movq\$3 er i write-back, men skrives først når klokken slår

I samme cyklus er addq i decode trin: læser tidligere værdi af %rax register



Stalling for Data Dependencies

```
# demo-h2.js
```

```
0x000: irmovq $10,%rdx
```

```
0x00a: irmovq $3,%rax
```

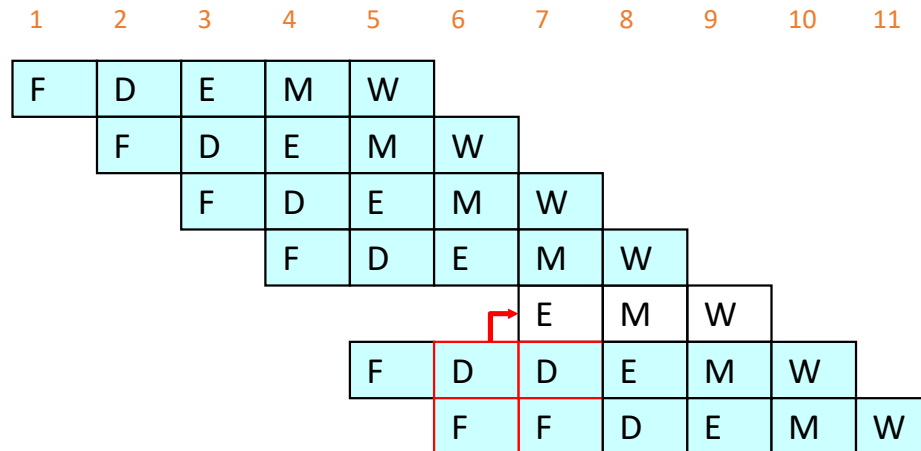
```
0x014: nop
```

```
0x015: nop
```

bubble

```
0x016: addq %rdx,%rax
```

```
0x018: halt
```



Princip:

- Hvis en instruktion følger "for tæt" efter en anden, så tilbageholder vi den
- Hold instruktionen tilbage (her i decode) og forsøg i næste cyklus: **STALLING**
- Kontrol logik indskyder dynamisk **no-op** (her i execute trin): "**luft-boble**":
- **Stalling kan minimeres med data-forwarding i pipelinen, men load-use konflikt koster stadig en cykle.**

```
mrmovq (%rbx),%rax
addq    %rbx,%rax
```

Pipelines og betingede hops

```
0x000:    xorq %rax,%rax
0x002:    jne t           # Not taken
0x00b:    irmovq $1, %rax # Fall through
0x015:    nop
0x016:    nop
0x017:    nop
0x018:    halt
0x019:  t:  irmovq $3, %rdx # Target (Should not execute)
0x023:    irmovq $4, %rcx # Should not execute
0x02d:    irmovq $5, %rdx # Should not execute
```

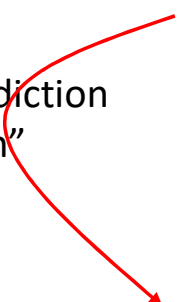
- En pipelinet processor skal gerne starte en ny instruktion pr clock-cyklus.
- Men ved en **betinget forgrening** kender den ikke hvilken gren skal udføres før forgrening passerer exe trinet hvor CC sættes!
 - STALL??
 - Vær optimistisk og "gæt" og udfør den valgte gren.
 - Gæt på **"jump-taken"**: næste PC sættes til valC (destination fra instruktion)
 - Ryd op ved fejl gæt "Instruktioner i pipelinen skal annulleres.
- EN lignende (men lidt værre) situation findes ved RET; her kendes retur adresse først efter mem-trinet (indlæsning fra stakken)!
 - STALL

Eksempel på fejlgæt

demo-j.js

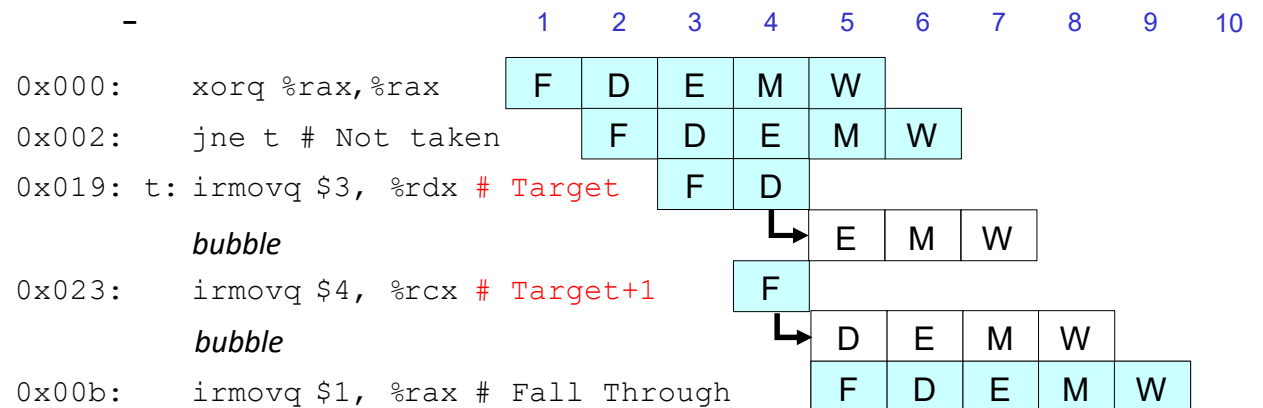
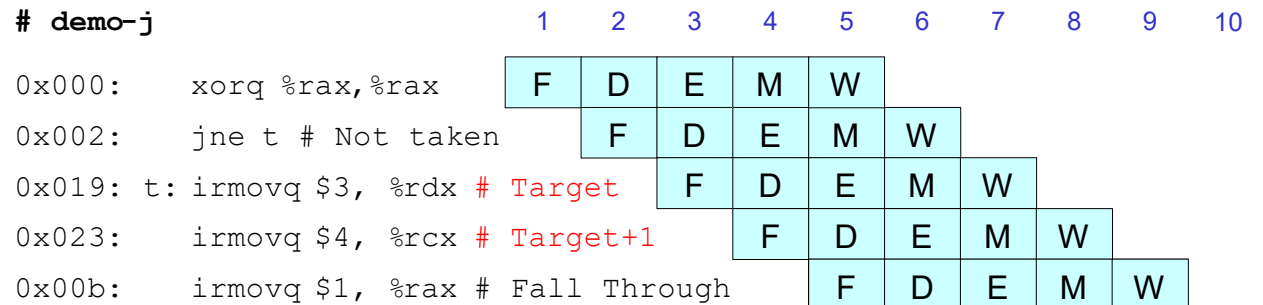
```
0x000:    xorq %rax,%rax
0x002:    jne  t           # Not taken
0x00b:    irmovq $1, %rax  # Fall through
0x015:    nop
0x016:    nop
0x017:    nop
0x018:    halt
0x019: t:  irmovq $3, %rdx  # Target (Should not execute)
0x023:    irmovq $4, %rcx  # Should not execute
0x02d:    irmovq $5, %rdx  # Should not execute
```

Branch-prediction
spår: "taken"



- Skulle kun udføre de første 8 instruktioner

Pipeline diagram for fejlgættet forgrening



- “Hop Taget” gættet fejlagtigt udfører to instruktioner fra hop-adressen
- Detekteret i cyclus 4: JNE i EXE
- Anullér fejlgættede instruktioner inden de får en effekt, der bliver synlige for programmøren (inden write-back).
- Skal moses. “Instruction squashing”
- Overskriv med “bobler” i decode og exec trin

Yde-evne

Metrikker for yde-evne

- Clock rate/frekvens
 - Måles i Giga- Hertz
 - Bestemmes af opdeling i trin og kredsløbsdesign
 - Hold arbejds-mængden pr trin lille
- Raten instruktioner udføres med
 - CPI: cycles per instruction
 - Gennemsnits-beregning, hvor mange clock cycles kræver hver instruktion?
 - Bestemmes af pipeline-design og (typiske) programmers work-load
 - Fx, Hvor ofte fejlgættes forgreninger?

CPI for PIPE

- CPI må ligge tæt på 1.0
 - Henter ny instruktion hver clock cyclus
 - Afslutter instruktion næsten hver cyclus
 - Dog har hver enkelt instruktion en forsinkelse på 5 trin (dermed 5 cyclers)
- CPI er > 1.0
 - Da den sommetider skal stalle eller annullere instruktioner
- Beregning af CPI
 - C clock cyclers brugt
 - I instruktioner er færdigbehandlede
 - B bobler indskudt ($C = I + B$)
$$\text{CPI} = C/I = (I+B)/I = 1.0 + B/I$$
 - Faktoren B/I repræsenterer den gennemsnitlige straf forårsaget af bobler

CPI for PIPE (Fortsat)

Analyse/benchmarking af typiske programmer

$$B/I = LP + MP + RP$$

Typiske Værdier

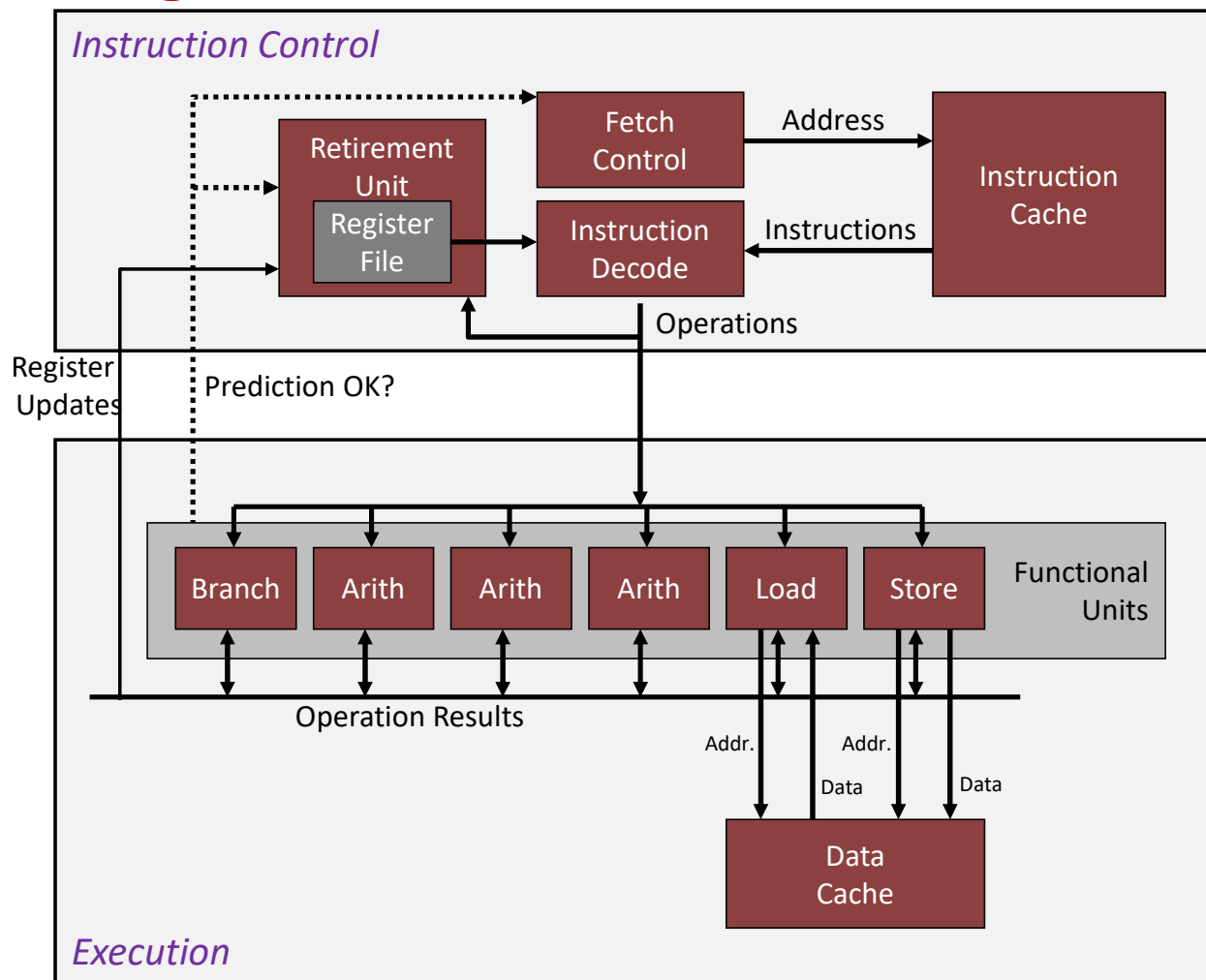
- LP: Straf fra load/use hazard stalling
 - Andel af instruktioner som er loads 0.25
 - Andel af heraf, der kræver stall 0.20
 - Antal bobler, der indskydes pr gang 1 $\Rightarrow LP = 0.25 * 0.20 * 1 = 0.05$
- MP: Straf fra fejlgættede jumps
 - Andel af instruktioner som er betingede jumps 0.20
 - Andel heraf som er fejlgættet 0.40
 - Antal bobler, der indskydes pr gang 2 $\Rightarrow MP = 0.20 * 0.40 * 2 = 0.16$
- RP: Straf fra `ret` instruktioner
 - Andel af instruktioner som er returns 0.02
 - Antal bobler, der indskydes pr gang 3 $\Rightarrow RP = 0.02 * 3 = 0.06$
- Samlet straf $0.05 + 0.16 + 0.06 = 0.27$
 $\Rightarrow CPI = 1.27$ (Ikke så ringe endda!)

- Dyreste bidrag: branch-mis-prediction
- CMOVE laver ikke kontrol forgrening: ingen omkostning ved branch mis-prediction, godt for performance!!

Moderne processorer

Moderne CPU Design

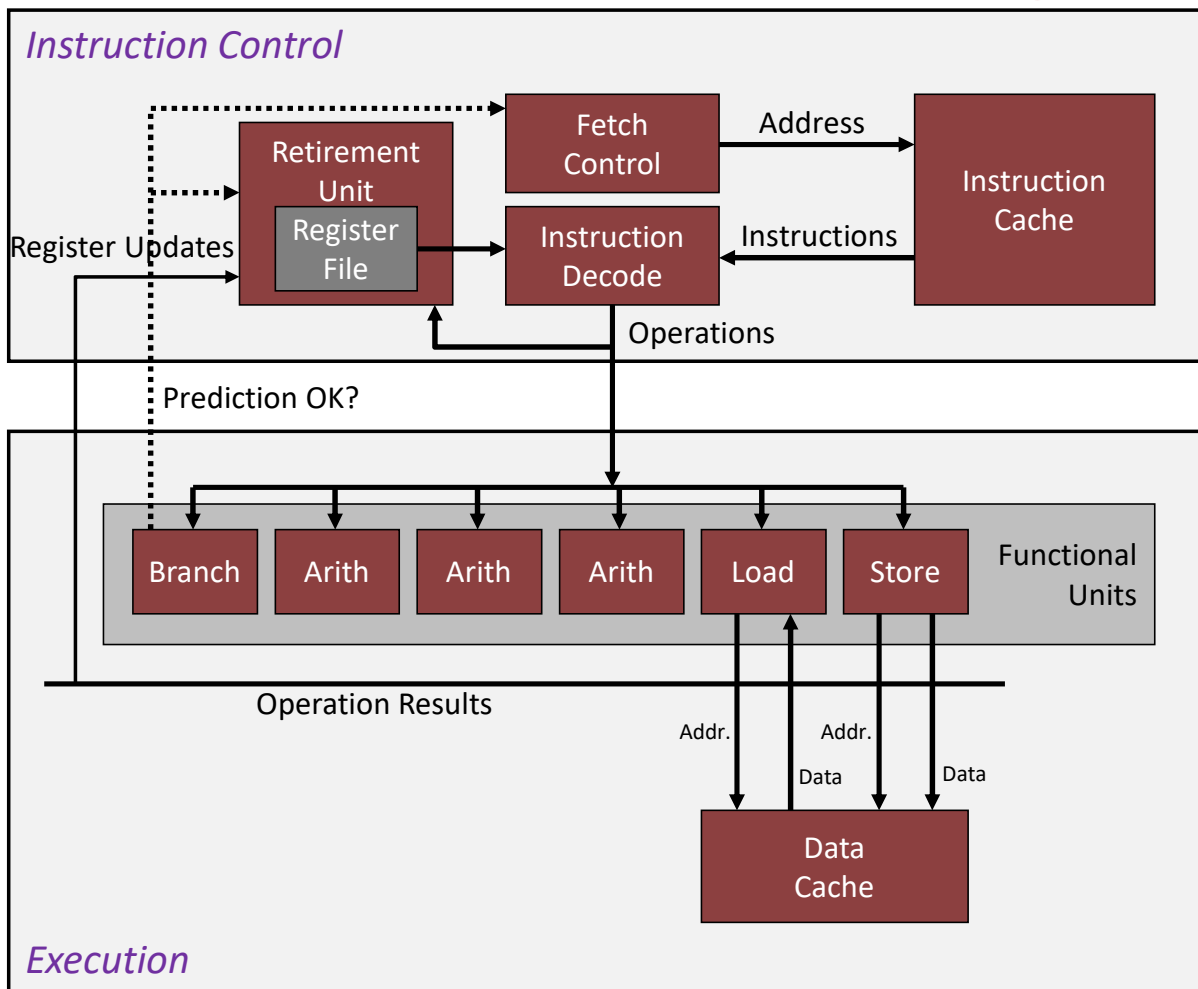
- Henter indkodede instruktioner fra hukommelsen
- Oversætter Instruktioner til μ Operationer
- Multiple "funktions-enheder" opererer uafhængigt, parallelt
- Spekulativ udførelse: Avanceret dynamisk branch prediction
- Konverterer register referencer til mærkater ("Tags") der binder dest. til src af efterfølgende instruktion
- Skriver kun registre når instruktionen er helt gennemført "retired": **Kontrol logik i Retirement enheden sikrer at adfærden svarer til sekventiel program udførelse**
- Operationer udføres så snart operanderne er klar (og en ledig enhed findes) - ikke nødvendigvis i samme rækkefølge som i programmet!
 - Generalisering af videresendelse



Superscalare Processorer

- **Definition:** En superscalar processor kan udstede og afvikle *flere instruktioner i éen cyclus*. Instruktionerne hentes fra en sekventiel strøm af instruktioner, og schedulers dynamisk.
 - **Out-of-order udførelse:** Processor udfører programmets instruktioner i anden rækkefølge end den programmøren har bestemt (dog med ISA-semantik) for at udnytte ledige funktionelle enheder
 - **Spekulativ udførelse:** Processor udfører instruktioner, inden den ved om de skal udføres (Fx. Gætter hvilken forgrening der skal tages. Branch-prediction)
- **Gevinst:** Superscalar processor udnytter den *instruktions niveau parallelisme* som de fleste programmer har, uden programmøren skal røre en finger!
- De fleste moderne CPUs er superscalare.
- Intel: siden Pentium generationen(1993)

Modern CPU Design



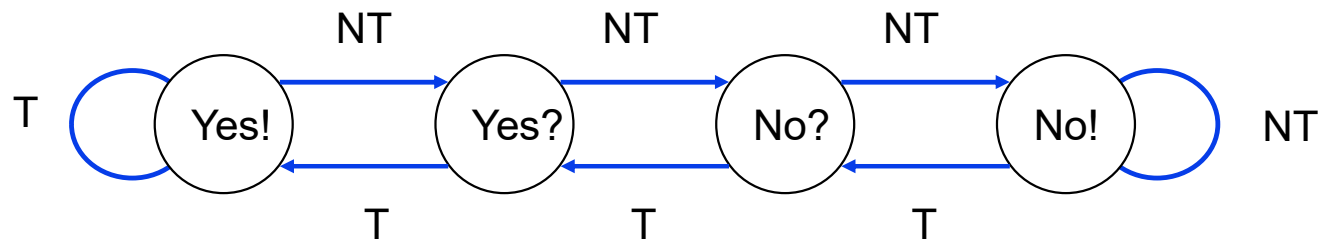
- FX Haswell corei7
- Super-scalar
- 8 parallel (pipelinede) enheder
 - 2 load med adresse beregning
 - 1 store med adresse beregning
 - 4 integer/long
 - 2 FP multiply
 - 1 FP add
 - 1 FP division
- Nogle instruktioner tager > 1 cykler, men kan være pipelinet

Komplekse instruktioner kan kræve flere enheder
`addq(%rax,%rdx,8), %rcx`
 => Load og add

<i>Instruktion</i>	<i>Latens (cycles)</i>	<i>Cycles/Issue</i>
Load/Store	4	1
Integer Multiply	3	1
Integer/Long Divide	3-30	3-30
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	3-15	3-15

Forbedret Branch Prediction

- Afgørende betydning for hastighed
 - Typisk 11–15 straf cyklers for fejlslæt
- Branch Target Buffer
 - 512 indgange (tabel med adresser som der hoppes til)
 - 4 bit historie (hvad var udfaldet de seneste besøg)
 - Adaptiv algoritme: Kan genkende gentagne mønstre, fx, skiftevis tag-hop, eller ikke-hop



- Tilstandsmaskine (for en given target adresse)
 - Hver gang "hop-tages", skift tilstand mod venstre
 - Når "tages ikke", skift tilstand mod højre
 - Processoren gætter på "hop tages" i tilstanden "Yes!" eller Yes?

Måling af yde-evne

- Clock rate/frekvens ?
- Raten instruktioner udføres med (CPI?)
- Benchmarks ?
 - Benchmarks via syntetiske workloads, fx
 - PassMark: stress tester via matematik tunge beregninger (compression, cryptering, fysik-sim)
 - 3DMark: 3D grafik til fx gaming.
 - PCMark 10: daglige “kontor” og produktivitets opgaver.
 - Single core / multi-core
 - Benchmarks via typiske rigtige arbejdsopgaver med applikationer (fx photoshop)

Compiler optimizer

Optimerende Compilere

- Oversæt programmet til maskinen, så den udnyttes effektivt
 - Register allokering
 - Valg af instruktioner og rækkefølge (scheduling)
 - Eliminering af "dead code"
 - Eliminering af øvrige mindre kilder til ineffektivitet
- Ændrer (normalt) ikke på asymptotisk kompleksitet
 - Overladt til programmøren at vælge den bedste algoritme
 - Forbedringer på "Store-O" er som regel mere vigtige end konstante faktorer
 - Men konstanterne gør en forskel: $O(1 \cdot n^2) \neq O(20 \cdot n^2)$
- Har svært ved at omgå "optimization blockers"
 - Mulige hukommelses aliaseringer (memory aliasing)
 - Mulige side-effekter af procedurer

Begrænsninger for optimerende Compilere

- Compiler må ikke ændre programmets adfærd:

gcc: -Og, -O1, -O2, -O3

- Det optimerede program skal gøre præcist de samme som det oprindelige program
- Forhindrer ofte compileren fra at anvende optimeringer, som kunne ændre programmets adfærd (i særlige/patologiske situationer, som måske/måske ikke vil optræde)
- Tiltænkt adfærd kan være indlysende for programmøren, kan blive uigennemskueligt pga-sprog/eller programmeringsstil
- Fx Aktuelt anvendte data område for en variabel, er mindre end dens type indikerer
- De fleste analyser udføres kun indenfor hver procedure (“intra-procedural”)
 - Analyse af programmet som ”et-hele” er normalt for beregningskrævende
 - Nyere GCC versioner laver “inter-procedural” analyser inden for hver fil.
 - Men ikke imellem kode i forskellige filer
- De fleste analyser baseres kun på *statisk* information
 - Compiler kan ikke forudse input, der gives mens programmet kører
- Når compileren er ”i tvivl” må den være konservativ

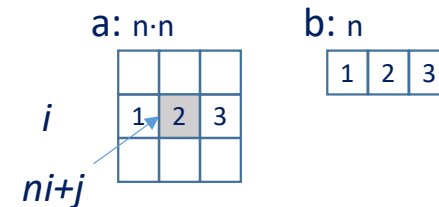
Just-in-time (JIT) compilering gør delvist dette

Almene nyttige optimeringer

- Optimeringer som du (eller compiler) bør gøre uanset processor type/compiler/sprog
- Kode flytning
 - Reducerer hyppigheden af den beregning
 - Hvis beregningen altid giver samme resultat
 - Flyt kode ud fra løkker

Skriver b-vektor til række i i a

```
void set_row(double *a, double *b,
             long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```



```
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```

Compiler-genereret Kode Flytning (-O1)

```
void set_row(double *a, double *b,
            long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
long j;
long ni = n*i;
double *rowp = a+ni;
for (j = 0; j < n; j++)
    *rowp++ = b[j];
```

Verificér om compiler
har optimeret som vi
har forventet?

```
set_row:
    testq    %rcx, %rcx          # Test n
    jle     .L1                  # If 0, goto done
    imulq    %rcx, %rdx          # ni = n*i
    leaq     (%rdi,%rdx,8), %rdx  # rowp = A + ni*8
    movl     $0, %eax            # j = 0
.L3:
    movsd    (%rsi,%rax,8), %xmm0 # t = b[j]
    movsd    %xmm0, (%rdx,%rax,8) # M[A+ni*8 + j*8] = t
    addq     $1, %rax            # j++
    cmpq     %rcx, %rax          # j:n
    jne     .L3                  # if !=, goto loop
.L1:
    rep ; ret                    # done:
```

NB: compiler vælger
ofte at bruge pointer
aritmetik til array
indeksering: kan
sometider give
besparelser: se
eksempel i
forelæsning 4

Reduktion af operator styrke

- Erstat dyre operationer med billigere
- **Shift**, **add** istedet for multiplikation og **division**

$16 * x \quad \rightarrow \quad x \ll 4$

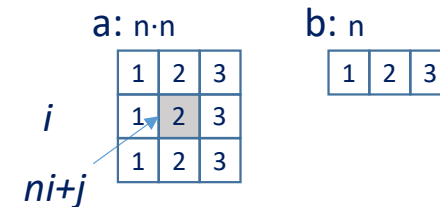
- Nyttegrad er maskin-afhængigt, afhænger af prisen for multiplikation og division
 - På Intel Nehalem, kræver heltals multiplikation 3 CPU cycles
- Genkend og undersøg sekvens af multiplikationer

Kopier b-vektor til alle rækker i a:
n er konstant for hver iteration: **mult**→**add**

```
for (i = 0; i < n; i++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```

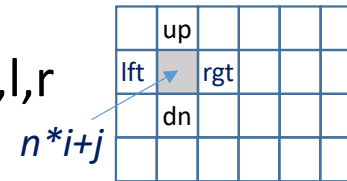


```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n;  
}
```



Genbrug resultat fra deludtryk

- GCC udfører dette med -O1
- Kode analyse viser at $n*i+j$ er fælles for u,d,l,r
- Fx Up: $(i-1)*n + j \rightarrow i*n - n + j$



```
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

3 multiplikationer: $i*n$, $(i-1)*n$, $(i+1)*n$

```
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplikation: $i*n$

Verificér om compiler har optimeret som vi har forventet.

```
leaq 1(%rsi), %rax # i+1
leaq -1(%rsi), %r8 # i-1
imulq %rcx, %rsi # i*n
imulq %rcx, %rax # (i+1)*n
imulq %rcx, %r8 # (i-1)*n
addq %rdx, %rsi # i*n+j
addq %rdx, %rax # (i+1)*n+j
addq %rdx, %r8 # (i-1)*n+j
```

```
imulq %rcx, %rsi # i*n
addq %rdx, %rsi # i*n+j
movq %rsi, %rax # i*n+j
subq %rcx, %rax # i*n+j-n
leaq(%rsi,%rcx), %rcx # i*n+j+n
```

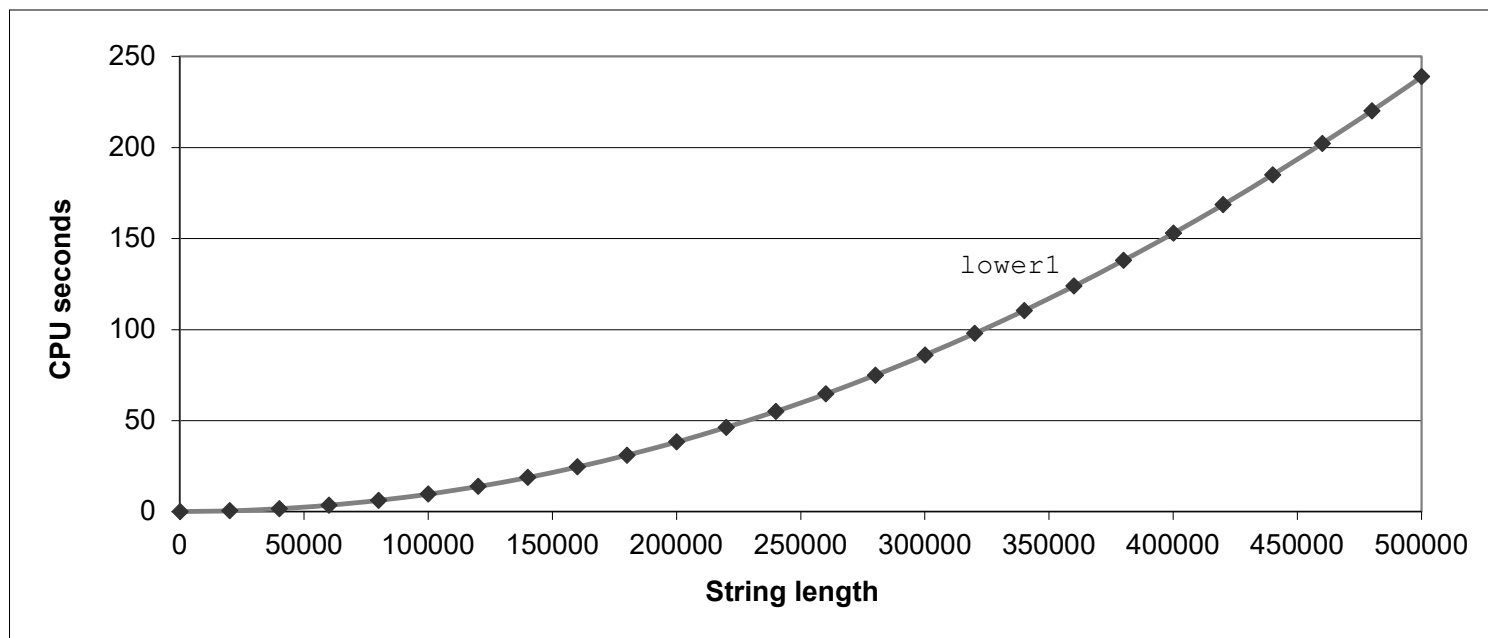
Optimeringsforhindring #1: Procedure kald

- Procedure til konvertering af streng til små bogstaver (Lower Case)

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Konverteringshastighed

- Køretid stiger kvadratisk $O(n^2)$



Analyse af hastighed

Goto version af "lower"

```
void lower(char *s)
{
    size_t i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

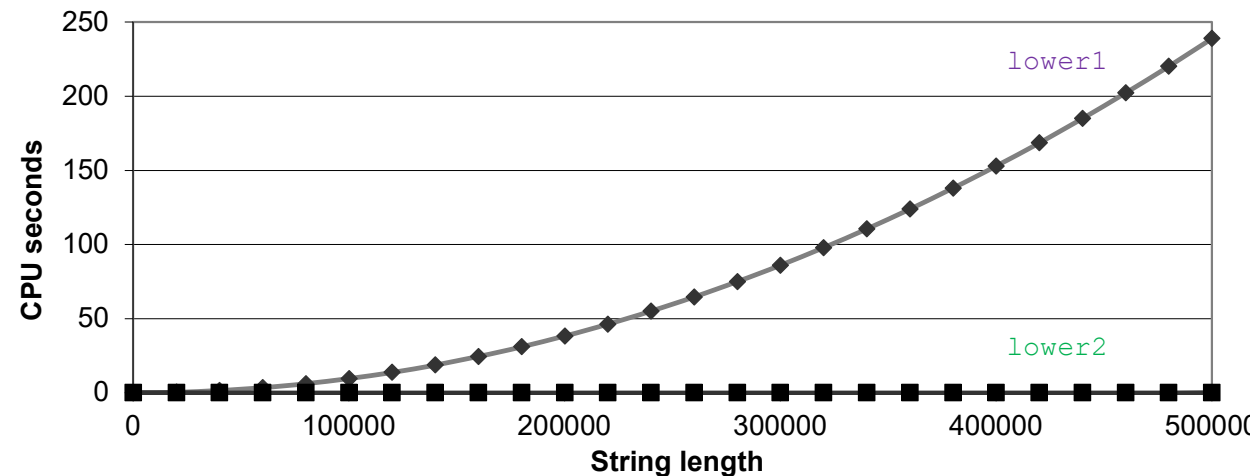
- Strlen hastighed: Vi søger strengen igennem fra start efter 0-byte terminering.
- Samlet hastighed for streng af længde N: $O(N^2)$
 - Strlen kaldt i hver iteration
 - N kald til strlen,
 - af hver N sammenligninger

NB: side-effekt

Skrivning til hukommelsen ændrer strengen;

Forbedring af hastighed

```
void lower2(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```



- Flyt kaldet til `strlen` udenfor løkken
- Resultatet for `strlen` ændrer sig ikke fra en iteration til en anden
- Variant af kode-flytnings-optimering

Optimeringsforhindring #1: Procedure kald

- *Hvorfor kunne compileren ikke selv flytte kaldet af `strlen` ud af løkken?*
 - Procedure kunne have haft side effekter!
 - Ændrer global tilstand/variable for hvert kald
 - Proceduren kunne returnere forskellig resultat for et givet sæt argumenter
 - Afhænger af andre dele af den globale tilstand
 - Procedure `lower` kunne også påvirke `strlen`
- **Advarsel:**
 - Compiler behandler procedure kald som "black box"
 - Svage optimeringer omkring dem
- **Afhjælpes ved:**
 - Lav selv kode-flytning: du (bør) kender hvad procedurerne gør
 - Brug "inlining" af funktioner
 - Bruges af GCC på `-O1` for funktioner i samme fil

Compiler skulle kunne "bevise" at `s[i]` ikke ændres til 0

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Compiler skal kende til side-effekter af indmaden af den kaldte:

Hvis `strlen` så således ud var optimeringen ikke korrekt!

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Funktions inlining

- **In-lining:** Erstat kaldsstedet med kopi af funktionskroppen (manuelt eller af compiler)
 - Små funktioner giver anledning til relativt stort overhead via CALL og især RET
 - Giver også flere andre optimeringsmuligheder

```
static inline long MIN(long a,
long b) {
    if (a > b)
        return b;
    return a;
}

int main(){
    long x,y,z;
    initXYZ(&x,&y,&z);
    x=MIN(y,z);
    return x;
}
```

Uden Inlining (-Og)

```
main:
...
movq    8(%rsp), %rsi
movq    16(%rsp), %rdi
call    MIN
addq    $40, %rsp
ret
```

Med Inlining (-Og)

```
main:
...
movq    8(%rsp), %rdx    #y
movq    16(%rsp), %rax   #z
cmpq    %rax, %rdx      #y:z
jl      .L6
.L4:
addq    $40, %rsp       #cleanup stack
ret
.L6:
movq    %rdx, %rax      #ret y
jmp     .L4
```

Med Inlining og -O1

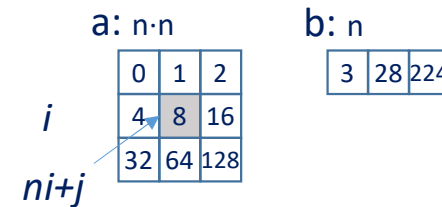
```
main:
...
movq    16(%rsp), %rax
cmpq    %rax, 8(%rsp)
cmovle  8(%rsp), %rax
addq    $40, %rsp
ret
```

Sidebemærkning: -O1 undgår også betinget forgrening og bruger CMOV (pipelining!)

- **Pris:** flere instruktioner, som dermed kræver mere plads
 - Kan være dårligt især for instruktions-CACHE
 - Brug kun ved små-funktioner
- Kan også fås i C vha. makroer (#define's), MEN makroer kan være alvorlig fejlkilde: læs fx.,
 - <https://stackoverflow.com/questions/1137575/inline-functions-vs-preprocessor-macros>

Indflydelse fra Hukommelsen

```
/* Sum rows is of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```



Verificér om compiler
har optimeret som vi
har forventet.

```
#%rdi=*a, %rsi=*b , %rax=i
# sum_rows1 inner loop
.L4:
    movsd    (%rsi,%rax,8), %xmm0    # FP load b[i]
    addsd    (%rdi), %xmm0           # FP add a[]
    movsd    %xmm0, (%rsi,%rax,8)    # FP store b[i]
    addq     $8, %rdi                #next a column
    cmpq     %rcx, %rdi              #final column?
    jne      .L4
```

- Koden opdaterer `b[i]` på hver iteration – dyr skrivning til hukk.!
- Kunne compiler ikke optimere dette væk, så den blot akkumulerer i et register ?

Aliasering

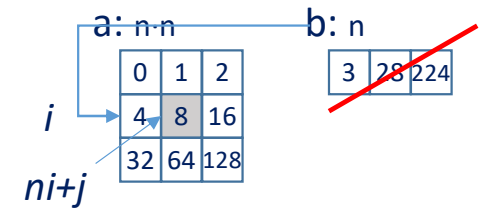
=Flere navne udpeger samme sted i hukommelsen

```
double A[9] =
{ 0, 1, 2,
  4, 8, 16},
  32, 64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

```
/* Sum rows is of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```



i=0 a[ni] b[i]

0	1	2	4	8	16	32	64	128
---	---	---	---	---	----	----	----	-----

0	1	2	3	8	16	32	64	128
---	---	---	---	---	----	----	----	-----

i=1 a[ni] b[i]

0	1	2	3	8	16	32	64	128
---	---	---	---	---	----	----	----	-----

B[1]=0 //A[4]=0

0	1	2	3	0	16	32	64	128
---	---	---	---	---	----	----	----	-----

B[1]+=A[3] //+=3

0	1	2	3	3	16	32	64	128
---	---	---	---	---	----	----	----	-----

B[1]+=A[4] //+=3

0	1	2	3	6	16	32	64	128
---	---	---	---	---	----	----	----	-----

B[1]+=A[5] //+=16

0	1	2	3	22	16	32	64	128
---	---	---	---	----	----	----	----	-----

i=2 b[i] a[ni]

0	1	2	3	22	16	32	64	128
---	---	---	---	----	----	----	----	-----

0	1	2	3	22	224	32	64	128
---	---	---	---	----	-----	----	----	-----

Fjernelse af Aliasing

- Compiler har ingen anelse om hvorvidt a,b kan risikere at være aliaseret => er konservativ
- Programmør ved at a, b aldrig aliaseres!
- Der er ikke behov for at skrive mellemresultat til hukommelsen:
- Gør det en vane at bruge lokal akkumulator variabel i løkker.
- Nemt for compiler at gennemskue at der er smart at lægge variabelen i et register
- Din måde at fortælle compileren at alisering ikke forekommer

Verificér om compiler har optimeret som vi har forventet.

```
/* Sum rows is of n X n matrix a
   and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
#%rdi=*a, %rsi=*b , %xmm0=val
.L10:
    addsd    (%rdi), %xmm0 # FP load + add
    addq     $8, %rdi
    cmpq     %rax, %rdi
    jne      .L10
```

Optimierung m. Instruktions-niveau parallelisme

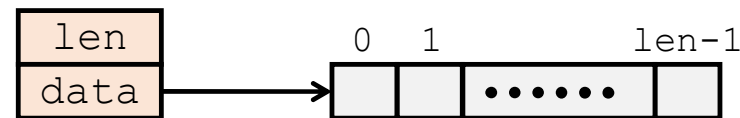
Udnyttelse af Instruktions niveau parallelisme

- I moderne processorer kan hardware udføre flere instruktioner parallelt
- Hastighed begrænset
 - data afhængigheder i programmet
 - Antal parallelle hardware regne-enheder i processoren
- Simple transformationer af programmet kan give drastisk hastighedsforbedring
 - Compilers kan tit ikke lave disse automatisk
 - Bl.a pga. float-point aritmetik ikke er associativ og distributiv

Benchmark Eksempel: Data Type for Vektorer

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```

```
typedef long data_t;
long data[4]={10,20,30,40};
vec myVec={4,data};
```



•Data Types

- Brug forskellige erklæringer af `data_t`
- `int`
- `long`
- `float`
- `double`

Index check

```
/* retrieve vector element
and store at val */
int get_vec_element
(vec *v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Ud-parameter

Benchmark Eksempel: Beregning

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```



Beregn sum eller produkt af vektor elementer

```
typedef long data_t;
long data[4]={10,20,30,40};
vec myVec={4,data};
Long res;
Combine_ADD(&myVec,&res);
//=0+V[0]+V[1]+ ... +V[n-1]=0+10+20+30+40=100
Combine_MUL(&myVec,&res);
//=1*V[0]*V[1]* ... *V[n-1]=1*10*20*30*40=240k
```

• Data Types

- Brug forskellige erklæringer af `data_t`
- `int`
- `long`
- `float`
- `double`

• Operationer

- Brug forskellige definitioner af `OP` og `IDENT`
- `+` og `0`
- `*` og `1`

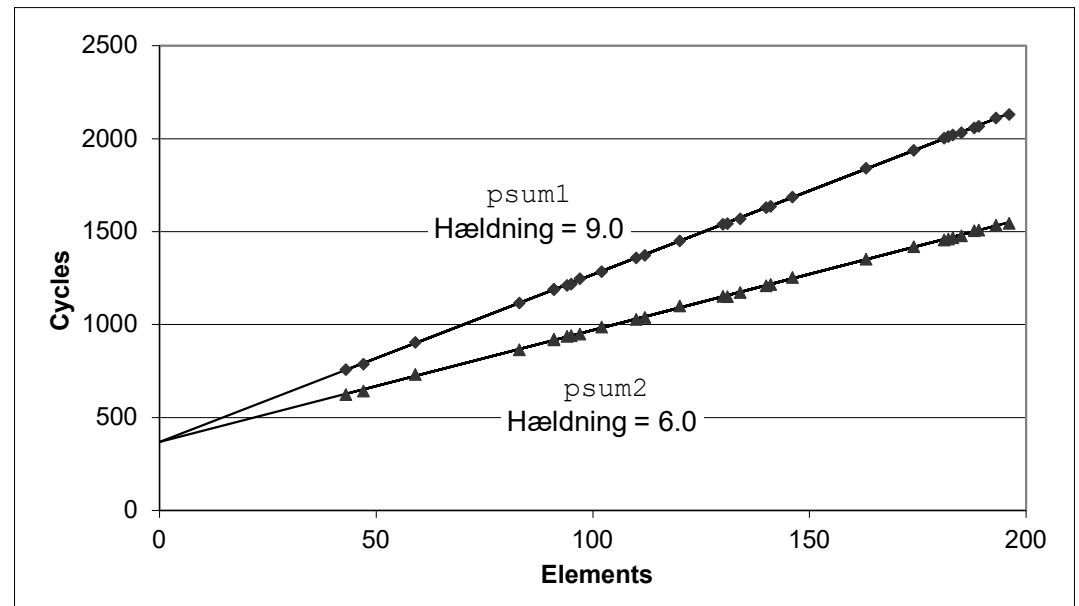
$x + \text{IDENT} = x$

$x * \text{IDENT} = x$

Gør det nemt at lave program varianter til sum/produkt af long/double

Performance Metrik: Clock Cycler pr. data Element (CPE)

- Bekvem måde at udtrykke hastigheden af et program, der opererer på data-elementer i vektorer, lister, mv.
- Længde = n
- I vektor-eksemplet gælder også:
CPE = cycles per OP
- $T(n) = CPE * n + \text{Overhead}$
 - CPE er hældningen på grafen
- Lille CPE er godt!



Benchmark Performance

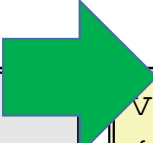
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Beregn sum eller produkt af
vektor elementer

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Combine1 uden optimering	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14

Målte CPE
værdier

Grundlæggende Optimeringer



```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

1. Flyt check af `vec_length` ud af løkken
 2. Undgå index-check i hver iteration: hånd inlinet `get_vec_elem`
 3. Akkumulér i lokal temporær (fjern potentiel aliasering iml. dst og v)
- HMM: vi har fjernet kald af `get_vec_elem`; har vi brudt indkapsling??

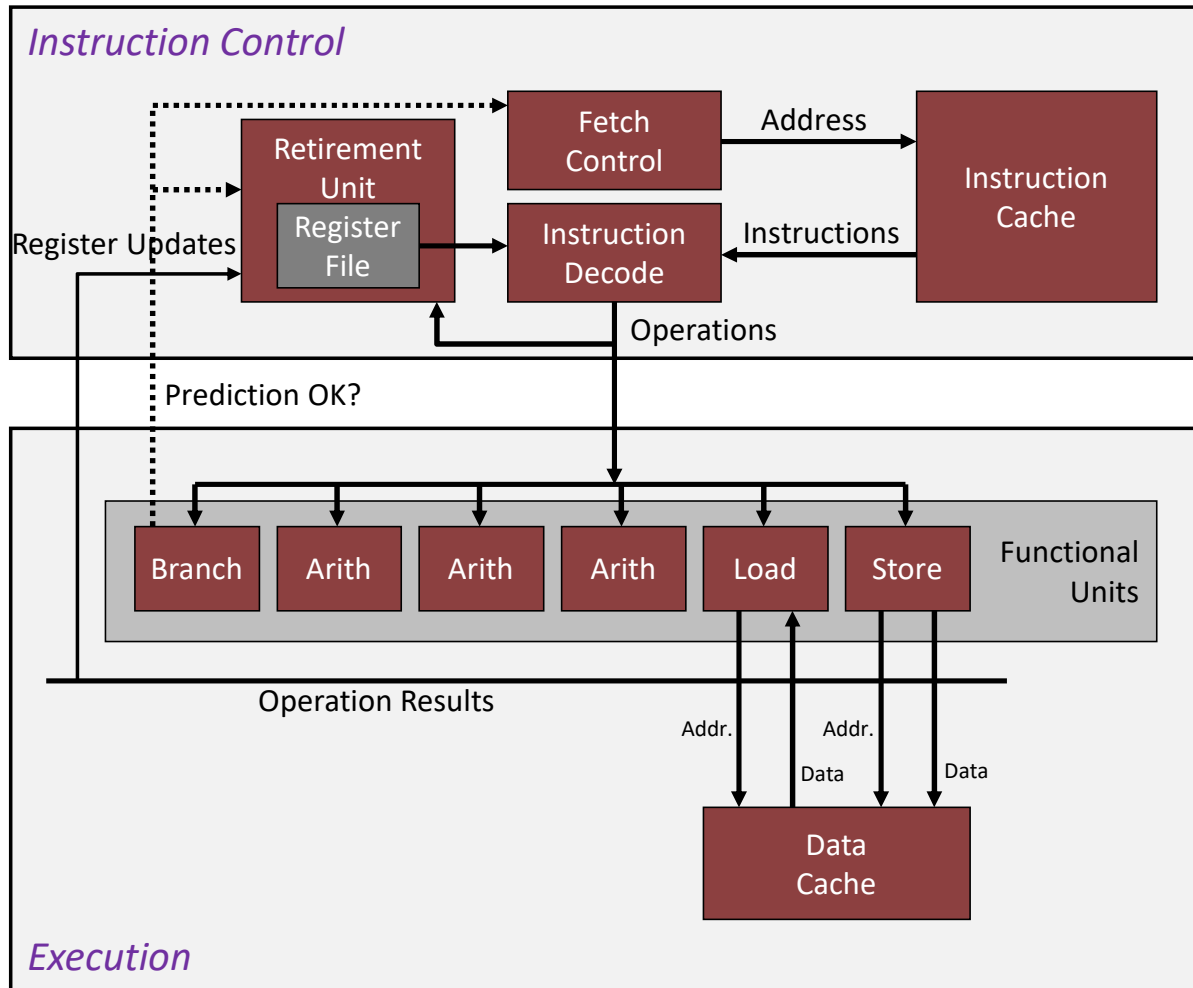
Effekt af Grundlæggende Optimeringer

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Combine1 -O1	10.12	10.12	10.17	11.14
Combine4 -O1	1.27	3.01	3.01	5.01

- Vores håndoptimering har elimineret åbenlyse kilder til overhead i løkken og optimerings-blokkere

Reminder: Moderne CPU Design



- FX Haswell corei7
- Super-scalar
- 8 parallel (pipelinede) enheder
 - 2 load med adresse beregning
 - 1 store med adresse beregning
 - 4 integer/long
 - 2 FP multiply
 - 1 FP add
 - 1 FP division
- Nogle instruktioner tager > 1 cykler, men kan være pipelinet

Komplekse instruktioner kan kræve flere enheder
`addq(%rax,%rdx,8), %rcx`
 => Load og add

<i>Instruktion</i>	<i>Latens (cycles)</i>	<i>Cycles/Issue</i>
Load/Store	4	1
Integer Multiply	3	1
Integer/Long Divide	3-30	3-30
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	3-15	3-15

x86-64 Oversættelse af Combine4

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latens grænse	1.00	3.00	3.00	5.00
Produktivitets grænse	0.50	1.00	1.00	0.50

Tilfældet: int add,
her er der noget
andet en latens, der
begrænser
hastigheden

- Forgrening?
- Anden ressource?

4 funk. enheder til int +
2 funk. enheder til load

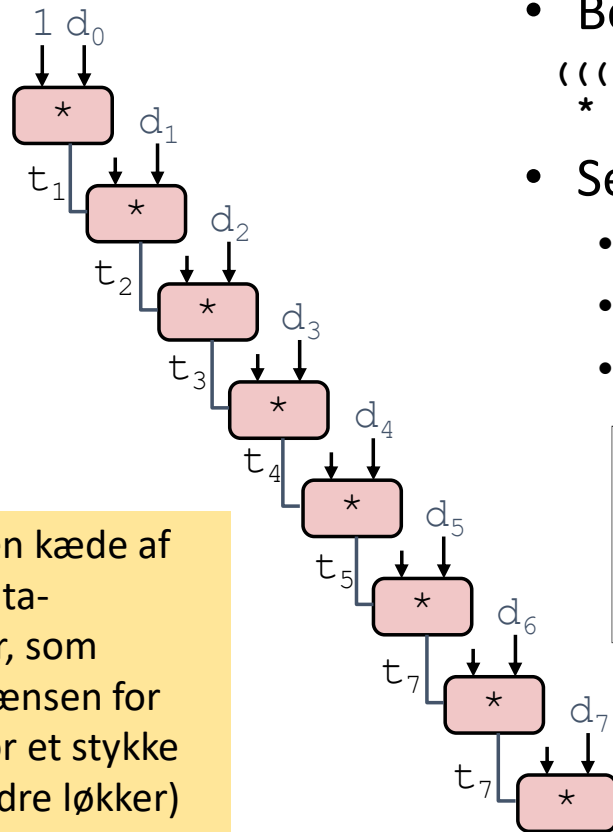
2 funk. enheder til FP *
2 funk. enheder til load

- Indre løkke (Tilfældet: Integer Multiply)

```
.L519:                                # Loop:
    imull    (%rax,%rdx,4), %ecx    # t = t * d[i] , t=%ecx
    addq     $1, %rdx              # i++
    cmpq     %rdx, %rbp            # Compare length:i
    jg       .L519                # If >, goto Loop
```

Data-afhængighed
imellem iterationer:
start af ny OP kan
ikke ske for tidligere
OP er helt færdig
(for fpmult: 5 cykler)

Combine4 = Serial beregning (OP = *)



Kritiske sti = den kæde af sekventielle data-afhængigheder, som bestemmer grænsen for hastigheden for et stykke kode (typisk indre løkker)

- Beregning (længde=8)

```
(((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])
```

- Sekventielle afhængigheder

- Hastighed: bestemmes af OP's latenstid
- imul kan ske **parallelt med** med add og cmp
- Imul har længste latens på 3 (add latens er 1)

```
.L519:                                # Loop:
    imull  (%rax,%rdx,4), %ecx        # t = t * d[i]
    addq   $1, %rdx                  # i++
    cmpq   %rdx, %rbp                # Compare length:i
    jg     .L519                     # If >, goto Loop
```

Udfoldning af løkker (Loop Unrolling): 2x1

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Udfører 2x så meget nyttigt arbejde per iteration
 - Reducerer overhead ved forgrening/indexering
 - Muliggør andre optimeringer

Effekt af udfoldning af løkker

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Udfold 2x1	1.01	3.01	3.01	5.01
Latens grænse	1.00	3.00	3.00	5.00
Produktivitets grænse	0.50	1.00	1.00	0.50

- Hjælper på integer add
 - Når latensgrænsen
- Andre forbedres ikke. *Hvofnudet?*
 - Stadig sekventielle afhængigheder:

```
x = (x OP d[i]) OP d[i+1];
```

Udfoldning af løkker plus Reassociering (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

$x = x \text{ OP } (d[i] \text{ OP } d[i+1]);$

Sammenlign med før

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

- Kan dette ændre resultatet for beregningen?
- Ja, for FP. *Hvofnudet?*

Effekt af Reassociering

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Udfold 2x1	1.01	3.01	3.01	5.01
Udfold 2x1a	1.01	1.51	1.51	2.51
Latens grænse	1.00	3.00	3.00	5.00
Produktivitets grænse	0.50	1.00	1.00	0.50

- Næsten 2x hastighedsøgning for Int *, FP +, FP *
 - Årsag: Bryder sekventiel afhængighed

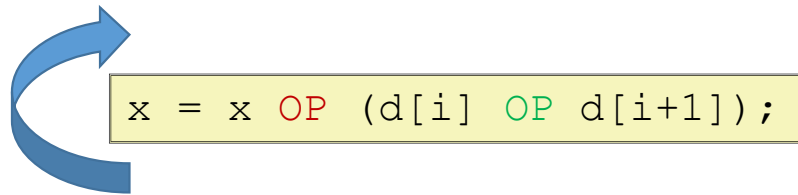
```
x = x OP (d[i] OP d[i+1]);
```

- Hvofnudet?

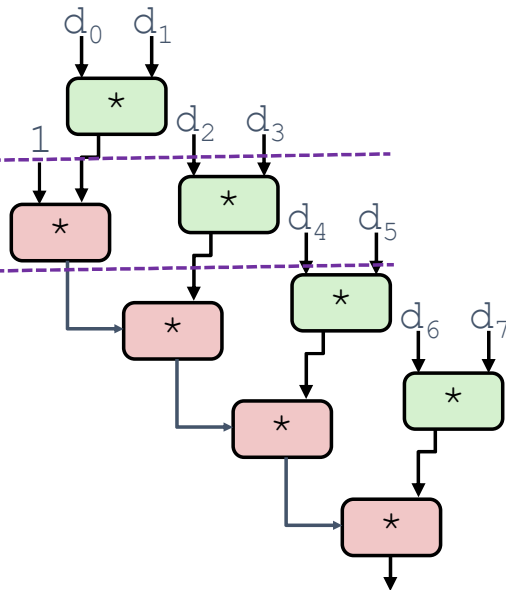
2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

Reassocieret Computation



Beregning på to
mult enheder parallelt!



- Hvad er ændret?

- OPs i den næste iteration kan startes tidligt (ingen afhængighed)
- Dermed parallelt med foregående OP

- Samlet hastighed

- N elementer, D cykler latens/operation
- $(N/2+1) \cdot D$ cycles:
CPE = D/2



Udfoldning af Løkker plus Separate Akumulatorer: 2x2

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

OP på x0 uafhængigt af
OP på x1 => parallelt

- Anderledes (mere general og normalt bedre) form for reassociering

Effekt af Separate Akkumulatorer

Metode	Integer(CPE)		Double FP(CPE)	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Udfold 2x1	1.01	3.01	3.01	5.01
Udfold 2x1a	1.01	1.51	1.51	2.51
Udfold 2x2	0.81	1.51	1.51	2.51
Latens grænse	1.00	3.00	3.00	5.00
Produktivitets grænse	0.50	1.00	1.00	0.50

- Bryder 1 CPE "barrieren" for Int+, stadig begrænset af to load enheder og loop overhead

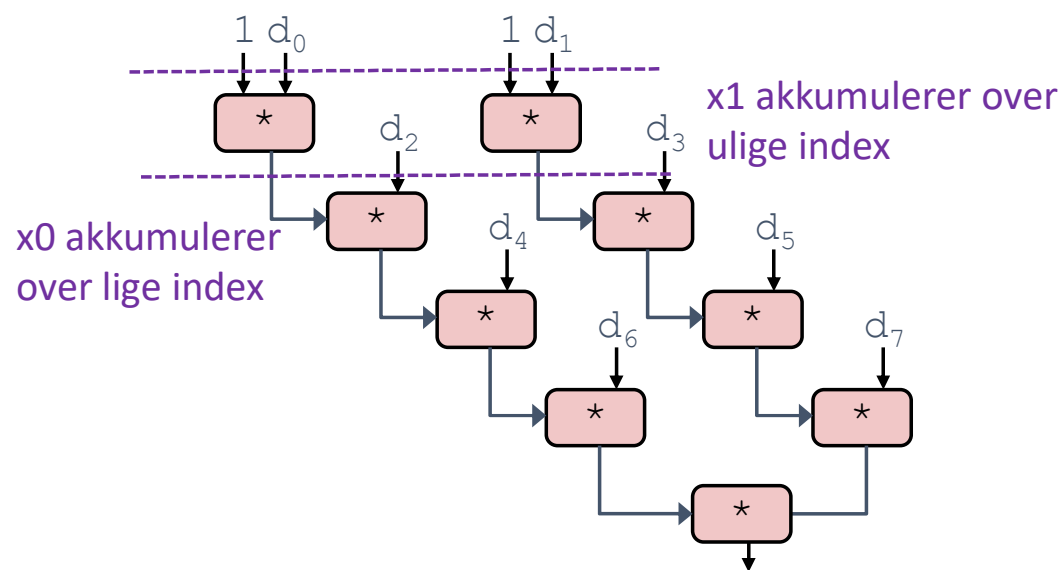
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- 2x hastighedsøgning (ifht udfold2x1) for Int *, FP +, FP *



Separate Akkumulatorer

```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```



■ Hvad er ændret:

- To uafhængige operations-“strømme”
- Udføres parallelt

■ Samlet ydelse

- N elementer, D cykler latens/OP
- Burde være $(N/2+1)*D$ cykler:
CPE = D/2
- CPE matcher forudsigelsen!

Hvad nu?

Unfoldning og Akkumulering

- Idé
 - Vi kan udfolde til enhver grad L
 - Vi kan akkumulere K resultater parallelt
 - L skal være et produkt af K
 - " $L \times K$ unrolling"
- Begrænsninger
 - Aftagende udbytte
 - Vi kan ikke overstige produktivitets-begrænsningerne (throughput limitations) af eksekveringsenhedernes
 - Stort overhead ved korte vektorer
 - De afsluttende iterationer udføres sekventielt
 - Regneregler skal respekteres: Distributivitet / Associativitet

Udfoldning og Akkumulering: Double *

- Case
 - Intel Haswell, Double FP Multiplikation
 - Latensgrænse: 5.00 cyclers. Produktivitets-grænse 0.50 CPE
 - På ukendt processor: Forsøg sig frem med forskellige kombinationer

Akkumulatorer	FP *	Udfoldnings Grad L							
	K	1	2	3	4	6	8	10	12
	1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
	2		2.51		2.51		2.51		
	3			1.67					
	4				1.25		1.26		
	6					0.84			0.88
	8						0.63		
	10							0.51	
	12								0.52

Udfoldning og Akkumulering: Int +

- Case
 - Intel Haswell
 - Integer addition
 - Latens grænse: 1.00. Produktivitets-grænse : 0.5

Akkumulatorer	Int +	CPE ved Udfoldnings Grad L							
	K	1	2	3	4	6	8	10	12
	1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
	2		0.81		0.69		0.54		
	3			0.74					
	4				0.69		1.24		
	6					0.56			0.56
	8						0.54		
	10							0.54	
	12								0.56

Opnåelig yde-evne

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Bedste variant	0.54	1.01	1.01	0.52
Latens grænse	1.00	3.00	3.00	5.00
Produktivitets grænse	0.50	1.00	1.00	0.50

- Primært begrænset af produktiviteten af funktionelle enheder
- Sekundært begrænset af **Register Spilling**: ikke nok registre til at holde temporære variable (fx akkumulatorer) => Ekstra tid til hukommelsesadgang (stak)
- Op til 42X hastighedsforbedring ifht. oprindelige uoptimerede kode (Ca. 20 CPE)
- 20X ifht. -O1 (ca. 10 CPE)

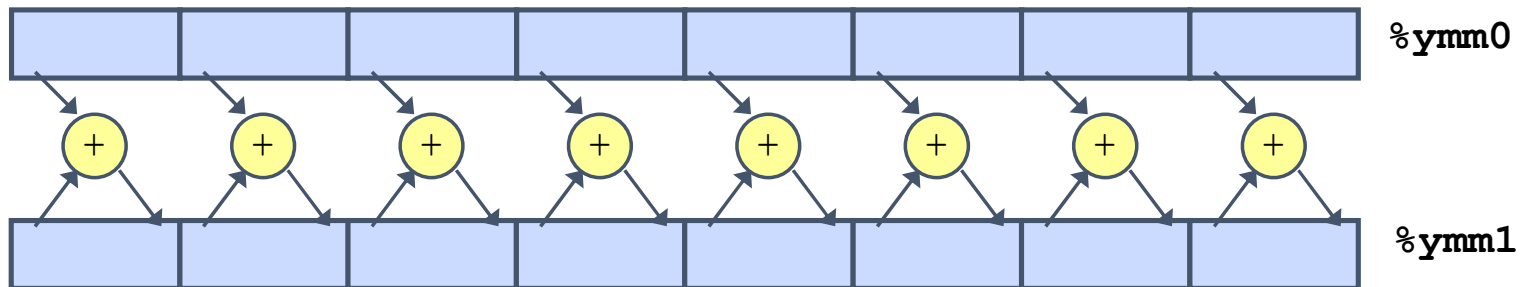
SIMD Operations

AVX= Advanced
Vector Xtension

- Variant af **S**ingle **I**nstruction **M**ultiple **D**ata parallelisme
- AVX2: 16 vektor registre af hver 32 bytes (opdateret udgave: “AVX-512” = 64 bytes)
- SIMD Operationer: 8 Int/Single Precision

`vaddsd %ymm0, %ymm1, %ymm1`

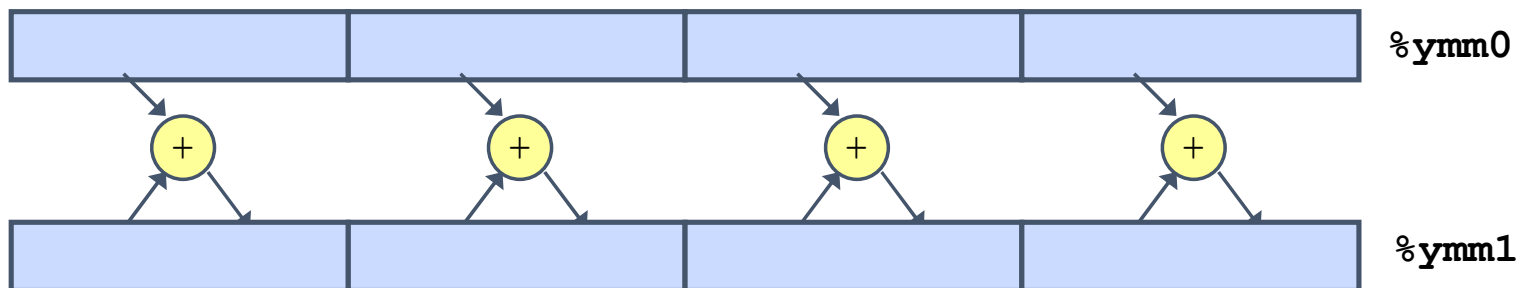
Parallelt



- SIMD Operationer: 4 Long/Double Precision

`vaddpd %ymm0, %ymm1, %ymm1`

Parallelt



Benchmark “Combine” vha. Vector Instruktioner

Metode	Integer (CPE)		Double FP (CPE)	
Operation	Add	Mult	Add	Mult
Bedste Skalare	0.54	1.01	1.01	0.52
Bedste Vektor	0.06	0.24	0.25	0.16
Latens grænse	0.50	3.00	3.00	5.00
Produktivitets grænse	0.50	1.00	1.00	0.50
Vektor Produktivitets grænse	0.06	0.12	0.25	0.12

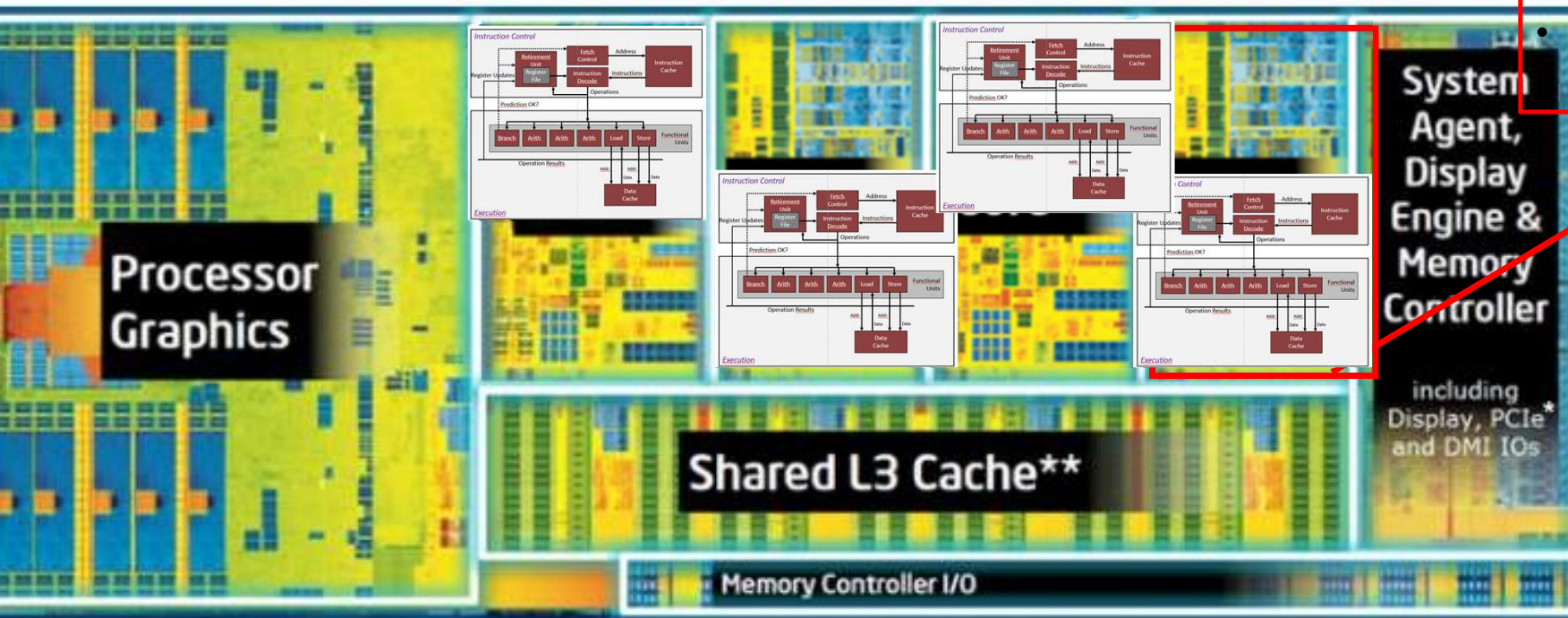
- Gør brug af AVX Instruktioner
 - Parallel operationer på multiple data elementer
 - Se Web Aside OPT:SIMD på CS:APP web.

Multi-core processing

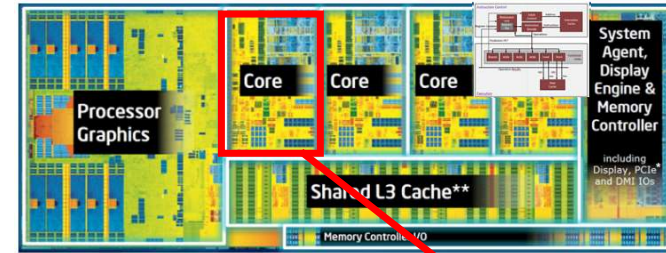
- Core-i7: op til 8 kerner (i9:18, Xeon 28, AMD EPYC: 32)

CPU chip indeholder flere processor kerner

- Egen register-bank
- Egen kontrol enhed
- Egne Funktionelle enheder
- Egen L1+L2 cache
- Andre ressourcer er delte: I/O, grafik, L3, RAM



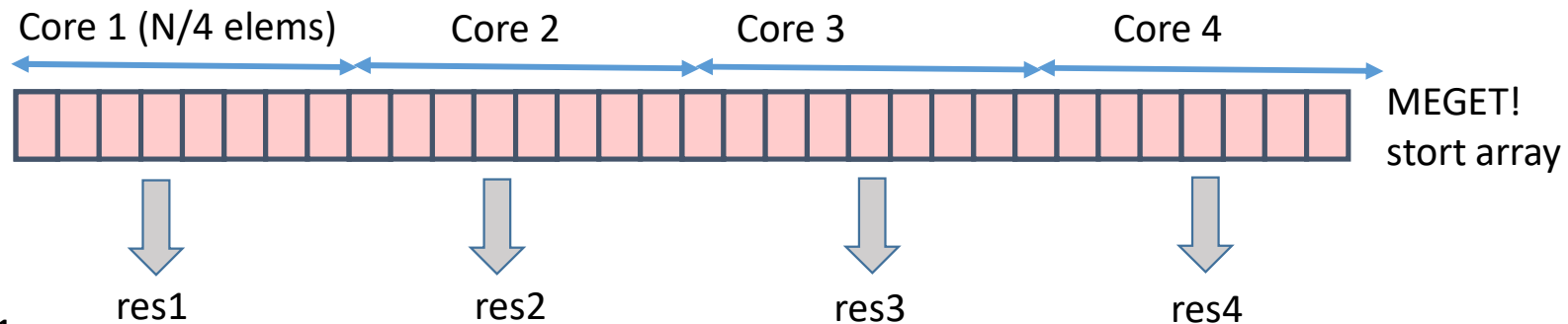
Multi-core processing



CPU chip indeholder flere processor kerner

- Egen register-bank
- Egen kontrol enhed
- Egne Funktionelle enheder
- Egen L1+L2 cache
- Andre ressourcer er delte: I/O, grafik, L3, RAM

- Core-i7: op til 8 kerner (i9:18, Xeon 28, AMD EPYC: 32)
- Eksplicit tråd-niveau parallelisme (thread-level parallelism)
 - `Thread_create(function ptr)`
 - `Thread_wait()`, `Thread_signal()`
 - Relativt store omkostninger ved administration af tråde: kræver relativt store beregningsopgaver ("grain size")
- Parallele Algoritmer
 - `combine_mp`: Pinlig parallel ("Embarassingly parallel")



Core 1:

Uddelegerer også opgaver, og beregner slut-resultat: $\text{res1 OP res2 OP res3 OP res4}$

En optimeringsmetode

Virkelighed om yde-evne (Performance)

- *Asymptotisk (tids-) kompleksitet*

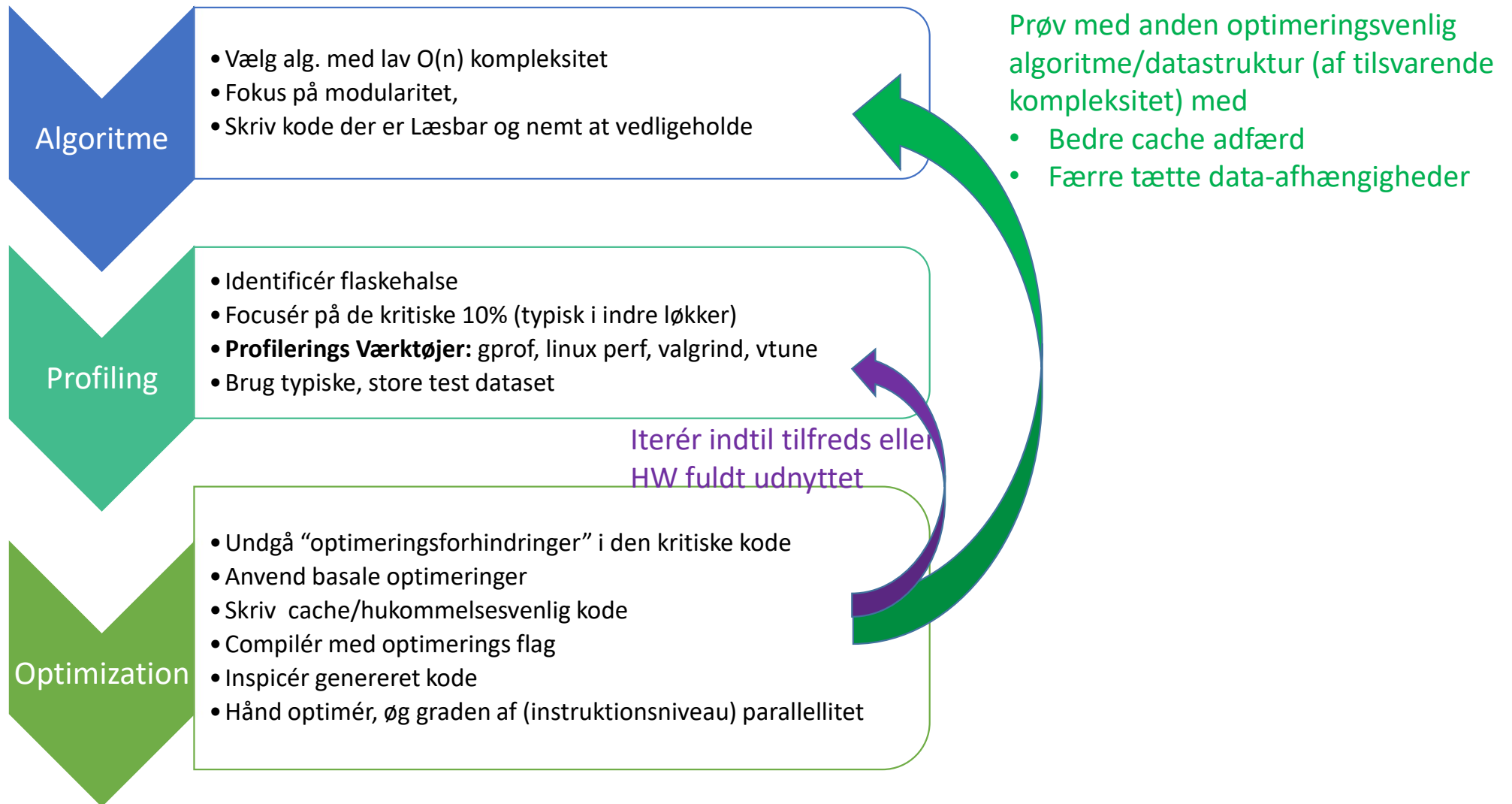
- Algoritme A , input størrelse n
- Antal beregningsskridt: $Tid(A(n))=10+2n+30n^2$
- A har kompleksitet **$O(n^2)$** , da n^2 dominerer konstanter og mindre led, når n bliver stort
- Kompleksitetsklasser: $O(1)$, $O(n)$, $O(n \log n)$, $O(n^k)$, $O(k^n)$, ..

- *Konstanterne gør en forskel!*

- Samme algoritme kan implementeres så *hastigheden varierer en faktor 10*
- Optimeringer skal ske på flere niveauer:
 - Valg af algoritme, data repræsentation, procedurer, og løkker

- *Pareto's 80/20 princip* for lovmæssig skævhed: 80% af udførelsestiden forbruges i 20% af koden (Variant: 90/10 reglen)
- Hvordan måler vi program hastighed og *identificerer flaskehalse*
- Hvordan kan vi forbedre hastighed?

Optimeringsmetode



Tids-måling

- Skal udføres nøje!
- Metrikker:
 - Clock-cycles vs. real-time vs. CPU-time vs. CPE
- Brug timer med høj opløsning (nano-sec) timer
- **Vær OBS på forstyrrelser:**
 - OS, netværk, virtual hukommelse, andre kørende applikationer
 - "CPU throttling" på bærbare (brug "high performance mode")
 - Indflydelse af måle-koden (ændret program!)
 - Brug aldrig debug printf (dyr I/O) i den kritiske kode
- Cache "warm-up" effekter
- **Statistik:**
 - Gennemsnit, median,
 - varians, min, max

```
long stats[ITER];
...
for(i=0;i<ITER;i++){
    start_time=get_time();
    //code to measure
    end_time=get_time();
    stats[i]=end-start;
}
```

```
//udlæs time stamp register
unsigned long lo, hi;
asm( "rdtsc": "=a"(lo), "=d"(hi));
return( lo | (hi << 32) );
//bad for OLD multi-core systems
```

```
//more general/portable
#include <time.h>
int clock_getres(...);
int clock_gettime(...);
```

Resumé

- Store-O kompleksitet vigtigt, men for samme algoritme kan aktuel køretid ofte optimeres en faktor >10 .
- Generelt nyttige Optimeringer
 - Kode flytning / forud beregning
 - Reducér operator styrke
 - Deling af fælles del-udtryk
 - Fjern unødvendige procedure kald
 - Brug conditional moves; undgå spaghetti-kode (dårlig branch prediction)
- Optimerings forhindringer
 - Procedure kald
 - Hukommelses aliasering
- Udnyt Instruktions-niveau parallelitet
 - Udfoldning af løkker og multiple akkumulatorer
- **Skriv hukommelses-”venlig” kode (næste gang)**
- **Brug profileringsværktøjer til identificere flaskehals: iteration**

Prøv jeres evner af på Challenge 8