# Avanceret pipeline, anonyme faser

Det er lidt træls, hvis man skal redegøre separat for hver enkelt fase en instruktion gennemløber i en moderne mikroarkitektur. Det skyldes at moderne mikroarkitekturer afvikler instruktioner i mange forskellige faser og disse faser kan tage forskellig længde.

## Længere pipelines

I moderne CMOS er det ikke realistisk at lave et cache-opslag på en enkelt cyklus. Typisk bruges tre cykler, som er fuldt pipelinet. Oftest er det heller ikke muligt at fuldt afkode en instruktion på en enkelt cyklus.

For eksempel, i den simple pipeline ventede (stall) vi på at en load instruktion var helt færdig med `M` fasen før den næste instruktion gik i gang, med denne fase. I en moderne mikroarkitekturer kan vi have flere instruktioner, som er i gang med at læse eller skrive.

Vi kunne løse dette ved at navngive hver enkelt af de ekstra faser der kræves og opskrive regler for hver af dem. Det bliver dog hurtigt meget kompliceret og bliver umuligt at overskue hvis vi pludselig skal holde styr på 3 forskellige navne for dele indhentning, 2 for afkodning, 3 for cache adgang osv. Det varer ikke længe før vi løber tør for bogstaver.

I stedet vælger vi en simplere notation. Når vi opskriver faserne for en instruktionsklasse kan vi

- tilføje de ekstra **anonyme** faser som er påkrævet med `-` og
- angive hvor mange instruktioner der kan befinde sig i "mellemrummet" mellem to navngivne faser.

Vi kan betragte tidligere beskrivelser af begrænsninger som specialtilfælde, hvor *ingen* instruktioner må være i anonyme faser, dvs: `F-D: 0`, `D-X: 0`, `X-M: 0`, `M-W: 0`. Her forstås således at "`F-D: 0`" at der ingen instruktioner må være, som har gennemført fase F, men ikke påbegyndt D. Med andre ord: Fase `D` skal følge direkte efter fase `F` i afviklingsplottet

## Eksempel 1: Avanceret pipeline

Lad os nu definerer en mere avanceret (og realistisk) mikroarkitektur. Lad os først fastsætte begrænsningerne på vores instruktioner

| | Instruktion | Faser | Dataafhængigheder |
|---|---|---|---|
| Aritmetik | `op a b` | `F--D-XW` | `depend(X,a), depend(X,b), produce(X,b)` |
| Læsning | `movq (a),b` | `F--D-XM--W` | `depend(X,a), produce(M+2,b)` |
| Skrivning | `movq b,(a)` | `F--D-XM` | `depend(X,a), depend(M,b)` |

Dataafhængighederne er de samme som tidligere, men den undtagelse at aritmetik instruktionerne, nu ikke har en `M` fase og derfor producerer deres resultat til i fase `X` til fase `W`. Det ses af faserne. Her har vi indsat to anonyme faser `-` efter `F` og `M` for at definerer cache adgang tager i alt 3 clock perioder. På samme måde kan vi se at afkodningen i `D` nu tager 2 clock perioder. Vi kan stadig lave stalls in disse anonyme faser, så det er muligt at der er flere end antallet mellem to givne faser. Ved læsning definerer vi at resultatet til `b` er klar 2 skridt efter

Vi kan nu sætte de udvidede specifikationer for faserne som

- Tilgængelige ressourcer: `F:2`, `D:2`, `X:2`, `M:1`, `W:2`
- Antal instruktioner under beregning: `F-D: 4`, `D-X: 2`, `M-W: 2`
- `inorder(F,D,X,M,W)`

Vi har det samme antal ressourcer som i vores superskalar arkitektur, men kan nu have 4 ekstra instruktioner i de anonyme faser mellem `F` og `D`; altså i gang med at blive indhentet. Der er 2 ekstra mellem `D` og `X`, samt 2 ekstra mellem `M` og `W`. Det er implicit at der ikke kan være nogen mellem `X` og `M` da det her kun tager en clock periode at beregne en værdi. Vi sikre stadig at alle faser afvikles in-order.

Se følgende eksempel på en kørsel af et program; læg mærke til at vi har to iterationer at en opdatering at et array:

```
                0123456789012345678    -- Vigtigste bemærkning
movq (r10),r11  F--D-XM--W             -- produce(M+2,r11)
addq $100,r11   F--D-----XW            -- depend(X,r11), produce(X,r11)
movq r11,(r10)   F--D----XM            -- depend(M,r11), depend(M,r11)
addq $8,r10      F--DDDDD-XW           -- produce(X,r10)
movq (r10),r11    F--DDDD--XM--W       -- depend(X,r10), produce(M+2,r11)
addq $100,r11     F------D-----XW      -- depend(X,r11), produce(X,r11)
movq r11,(r10)     F-----DD----XM      -- depend(X,r10), depend(M,r11)
addq $8,r10         F------DDDDD-XW    -- depend(X,r10), produce(X,r10)
```

Nu begynder der at ske en del.

1. instruktion forløber normalt, da vi har en "tom" pipeline; dog bør noteres at værdien fra læsning først er klar i fase `W`

2. instruktion kan indlæses og afkodes samtidig med den første; vi har to enheder. Dog er den afhængig af at `r11` er klar i `X`, så denne fase kan tidligst være samtidig med `W` fra første instruktion. Vi laver altså en stall i den anonyme afkodningsfase.
3. instruktion kan indlæses i anden fra anden periode og afkodning kan gå i gang som normalt. Vi har dog damme afhængighed som før of er nødt til at stall i `D`'s `-`.
4. instruktion kan begynde indlæsning og afkodning samtidig med den tidligere. Vi har dog allerede fyldt anden del af `D`, så vi staller denne i `D`; se at der allerede er to streger i søjlen over efter `D`. Afhængigheden på `r10` er endnu ikke noget problem, men vi noterer at den opdaterer `r10` i `W` fasen.
5. instruktion og vi starter anden iteration. Vi kan begynde indlæsningen som forventet, men er igen nødt til at stalle i `D` til vi har plads til at afkode. Vi har derefter en afhængighed på `r10` i `X` fasen, som vi så er nødt til at stalle en enkelt clock periode. Resten forløber som forventet, men indlæsning til `r11` er først klar til `W`.
6. instruktion er vi nu nødt til at stalle i indlæsningen, `F`, da der ikke er plads i afkodningen. Derefter er vi igen nødt til at stalle efter `D` for at vente på vores afhængighed på `r11`; `X` kan ikke ligge tidligere en samtidig med `W` fra før.
7. instruktion er igen en masse stall efter `F` og derefter `D`. Vi har først en afhængig på `r10` i `X` som kommer fra fjerde instruktion; vi har dog en in-order maskine og kan derfor ikke ligge adresseberegningen tidligere end `X` fra den forrige instruktion. Dette løser også afhængigheden på `r11` i `M`, da denne bliver produceret i forrige instruktions `W`.
8. instruktion forsinkes både i `F` og `D`, som tidligere. Ellers er der ikke noget at bemærke.

## Abstraktion, samlet indlæsning og afkodning

Anonyme faser kan også gøre det nemmere at se bort fra ting der ikke har interesse. For eksempel kan vi udelade afkodningstrinnet fra vores beskrivelse, da det altid følger direkte efter indhentning og har samme antal ressourcer. I stedet kan vi lave vores indlæsningstrin det længere og få samme afvikling:

| | Instruktion | Faser | Dataafhængigheder |
|---|---|---|---|
| Aritmetik | `op a b` | `F----XW` | `depend(X,a), depend(X,b), produce(X,b)` |
| Læsning | `movq (a),b` | `F----XM--W` | `depend(X,a), produce(M+2,b)` |
| Skrivning | `movq b,(a)` | `F----XM` | `depend(X,a), depend(M,b)` |

- Tilgængelige ressourcer: `F:2`, `X:2`, `M:1`, `W:2`
- Antal instruktioner under beregning: `F-X: 8`, `M-W: 2`

- `inorder(F,D,X,M,W)`

Dette giver den samme afvikling, blot er `D` ikke nævnt, men til gængæld kan det være mere overskueligt.

```
                0123456789012345678    -- Vigtigste bemærkning
movq (r10),r11  F----XM--W             -- produce(M+2,r11)
addq $100,r11   F--------XW            -- depend(X,r11), produce(X,r11)
movq r11,(r10)   F-------XM            -- depend(M,r11), depend(M,r11)
addq $8,r10      F--------XW           -- produce(X,r10)
movq (r10),r11    F--------XM--W       -- depend(X,r10), produce(M+2,r11)
addq $100,r11     F------------XW      -- depend(X,r11), produce(X,r11)
movq r11,(r10)     F-----------XM      -- depend(X,r10), depend(M,r11)
addq $8,r10        F------------XW     -- depend(X,r10), produce(X,r10)
```

Bemærk iøvrigt at selvom denne maskine kan håndtere 2 instruktioner per clk, så opnår den i ovenstående eksempel 4/9 IPC, dvs. mindre end ½ instruktion per clk.

## Eksempel 2: Mere udrulning

Det ser ud til at vi nu bare kan indhente instruktioner, som vi lyster. Så når vi nu er så godt i gang, kan vi tage en udrulning mere.

```
                012345678901234567890123    -- Vigtigste bemærkning
movq (r10),r11  F----XM--W                  -- produce(M+2,r11)
addq $100,r11   F--------XW                 -- depend(X,r11), produce(X,r11)
movq r11,(r10)   F-------XM                 -- depend(M,r11), depend(M,r11)
addq $8,r10      F--------XW                -- produce(X,r10)
movq (r10),r11    F--------XM--W            -- depend(X,r10), produce(M+2,r11)
addq $100,r11     F------------XW           -- depend(X,r11), produce(X,r11)
movq r11,(r10)     F-----------XM           -- depend(X,r10), depend(M,r11)
addq $8,r10        F------------XW          -- depend(X,r10), produce(X,r10)
movq (r10),r11      F------------XM--W      -- depend(X,r10), produce(M+2,r11)
addq $100,r11       FFFFF------------XW     -- depend(X,r11), produce(X,r11)
movq r11,(r10)       FFFF------------XM     -- depend(X,r10), depend(M,r11)
addq $8,r10              F------------XW    -- depend(X,r10), produce(X,r10)
```

1. instruktion kan starte indhentning som normalt. Hvis vi tæller antallet af streger over dette `F` (hold tungen lige i munden) tæller vi 8, som er plads til. I den efterfølgende periode (nummer 5) fortsætter først instruktion til `X`, så denne instruktion kan fortsætte sin indlæsning. Så skal vi bare huske vores afhængighed på `r10`.
2. instruktion kan starte indhentning med den tidligere. Men nu er de anonyme faser fulde og vi bliver nødt til at blive i `F`. Afslutningen følger de tidligere iterationer.
3. Vi kan starte indlæsningen i periode 5, men er igen nødt til at stalle i `F`.
4. Nu er `F` også blevet fyldt og vi er nødt til forsinke selve indlæsningen. Det er først i periode 9 at anden instruktion komme ud af sin indlæsning og vi kan derfor starte denne.

Det man især kan tage med fra ovenstående eksempel, er hvor mange instruktioner, som er i gang med at blive indlæst og hvor lang tid denne del tager sammenlignet med selve beregningen.

## Køer

Ofte indgår køer af instruktioner i en mikroarkitektur. Ikke så tit i en simpel skalar pipeline, men ofte i superskalare pipelines. For eksempel er det ofte en fordel med en kø mellem instruktionshentning og afkodning. For maskiner med variabel-længde instruktioner kan det være nødvendigt fordi afkoderen har en begrænsning på hvor mange instruktioner den kan håndtere, mens instruktionshentningen har en begrænsning udtrykt i bytes eller cache-blokke. Dermed kan man undgå at indlæse samme instruktion flere gange. Dette mis-match afføder et behov for at have en eller flere bytes/instruktioner i kø mellem de to trin.

Vi vil modellerer en kø som et antal anonyme faser. For eksempel kan vi udtrykke en kø mellem F og D med plads til fire instruktioner som en ressource begrænsning: "F-D: 4"

## Cache-miss

Vi modellerer lager-hierarkiet ved at holde rede på indholdet af systemets cache(s) instruktion for instruktion. Cache-miss inkluderes i modellen alene derved at det påvirker latens-tiden for de instruktioner, der læser fra lageret. Dette er en voldsom forsimpling. Vi ser således bort fra en række typiske begrænsninger:

- I en helt simpel pipeline vil man måske fryse selve pipelinen indtil data er ankommet. Det vil have dårligere ydeevne, end blot at forøge latenstiden, som vi gør.
- Der kan være et begrænset antal overlappende læsninger i gang samtidigt.
- Der kan være et begrænset antal snavsede blokke som er smidt ud på et niveau af hierarkiet og venter i kø på at blive skrevet til det underliggende niveau.
- Der kan være ressource konflikter ved tilgang til en cache; f.eks. kan man måske ikke læse fra cachen, samtidigt med at data der hentes i respons på et tidligere miss ankommer og skrives til cachen.

Det ser vi alt sammen bort fra.

Der antages copy-back caching med LRU replacement.

## Kontrolafhængigheder

Se Kontrolafhængighed