Simpel pipeline

Pipeline faser og ressourcer

Siden slutningen af 70'erne hvor de første pipelinede arkitekturer blev introduceret, har en instruktion gennemgået flere faser når den afvikles. Nogle faser er generiske; nogle afhænger af instruktionen.

Faserne gennemløbes i rækkefølge bestemt af instruktionstype og mikroarkitektur.

Betragt for eksempel afviklingen på en simpel pipeline, typisk for de første RISC maskiner konstrueret i 80'erne. Her er der fem faser:

- `F` : Fetch, indlæsning af instruktionen fra hukommelse,
- `D` : Decode, afkodning af instruktionen og udlæsning fra registerfil,
- `X` : eXecute, udførsel af aritmetisk/logisk operation, samt beregning af mulig adresse,
- `M` : Memory, læsning fra eller skrivning til hukommelsen,
- `W` : Writeback, tilbageskrivning til registerfilen.

Her ses nogle begrænsninger for en 5-trins pipeline:

- Alle instruktioner gennemløber: `FDXMW`
- Tilgængelige ressourcer: `F:1`, `D:1`, `X:1`, `M:1`, `W:1`

Ovenstående skal læses som: alle instruktioner passerer gennem samtlige fem trin ordnet som beskrevet; der findes en ressource for hvert trin, altså der kan kun være en instruktion i hver trin.

Bemærk at det er en voldsom forenkling at udtrykke begrænsningen for instruktionshentning i et antal af instruktioner. Især hvis instruktionen kommer til at ligge over to cache linier. For en maskine med instruktioner af forskellig længde er bindingen mere korrekt udtrykt som et antal bytes. Først i forbindelse med afkodning er det klart, hvor en instruktion begynder og slutter. Den lille detalje vil vi se bort fra.

Eksempel: Simpel pipeline mikroarkitektur

I en simpel pipeline vil afviklingsplottet for et mindre program være følgende:

```
                012345678
movq (r10),r11  FDXMW
mulq r10,r12     FDXMW
addq $100,r13     FDXMW
```

```
movq r14,(r10)      FDXMW
subq $1,r10          FDXMW
```

Her ses at første instruktion bliver indhentet i første clock periode. I anden clock periode vil anden instruktion blive indhentet samtidig med at første instruktion bliver afkodet, osv.

Det er vigtigt at vi overholde begrænsningerne. For at tjekke det ser vi at: * første begrænsning bliver overholdt, da alle linier i plottet indeholder alle fem trin, og * anden begrænsning bliver overholdt da hver søjle (clock periode) kun indeholder hvert trin en gang.

Hvis vi prøver at udregne ydeevnen for programmet, kan vi se at det samlet bruger 9 clock perioder: dette svare til antallet af instruktioner (5) + antallet af trin minus 1. Vi kan derefter udregne `CPI = 9/5 = 1,8`. Dette er dog misvisende fordi det ville have været muligt for efterfølgende instruktioner at starte tidligere. Vi skal i stedet tælle clock perioder fra start til slut af 'X' fasen. Hvis vi gør det, får vi en CPI på 1. Det er det sammen som for vores enkelt-cyklus maskine, men da hver fase udfører mindre arbejde, kan den pipelinede arkitektur nå en væsentlig højere clock-frekvens. Sandsynligvis omkring 4 gange højere.

Instruktioners latenstid

På en pipelined arkitektur bruger instruktioner en eller flere clock-perioder til at producere et resultat. Det kaldes instruktionens latenstid. Latenstiden er den tid der går fra instruktionen modtager/fremfinder sin sidste indgående operand og til en efterfølgende instruktion som afhænger af resultatet kan begynde sin beregning.

Man planlægger normalt en mikroarkitektur således at de grundlæggende aritmetisk og logiske instruktioner har en latenstid på en enkelt clock periode.

Andre instruktioner kan så få længere latenstid, fordi de udfører et mere kompliceret stykke arbejde. For eksempel er multiplikation mere kompliceret end addition og har derfor en latenstid på 3-4 clock perioder.

Tilgang til lageret er også mere kompliceret og tager længere tid end en enkelt clock periode. I den pipeline vi præsenterede ovenfor har instruktioner der læser fra lageret en latenstid på 2 clock-perioder, for eksempel. Det er i tilfælde af cache-hit. Situationen omkring et cache-miss er væsentlig mere kompliceret og vil ikke blive beskrevet her.

En latenstid på mere end en cyklus for en instruktion kan både opstå fordi selve eksekverings-delen er splittet i flere faser (pipelined eksekvering) eller fordi eksekverings-delen foregår iterativt så instruktionen bliver i samme fase i flere clock-perioder. I denne note antager vi at eksekvering altid er fuldt pipelinet. I virkelige maskiner gælder det for de fleste instruktioner med undtagelse af division.

Data afhængigheder og forwarding

Mere signifikant end latenstiden er data afhængigheder. Det har ikke været et problem i vores tidligere eksempler (ikke en tilfældighed), men kan hurtigt blive det for normale programmer.

Overvej følgende program:

```
movq (r10),r11
addq $100,r11
movq r11,(r10)
addq $8,r10
subq $1,r12
```

Her bliver register `r11` opdateret i instruktionen lige før det bliver læst; endda to gange. F.eks. indlæser første instruktion en værdi til `r11`, som anden instruktion straks lægger noget til; men også fra anden til tredje instruktion. Vi kan lave data-flow graph, som beskrevet i CSapp, som vil tydeliggøre de data afhængigheder, som eksisterer i programmet. Instruktionsnummeret er indsat efter navnet.

```
    r10  r11   r12
      | \ |     |
      | movq1   |
      |   |     |
      | addq2   |
      | / |     |
    movq3 |     |
      |   |     |
    addq4 |    subq5
      |   |     |
    r10  r11   r12
```

Hvis vi laver et simpelt afviklingsplot som før, vil vi få følgende:

```
movq (r10),r11  FDXMW
addq $100,r11    FDXMW
movq r11,(r10)    FDXMW
addq $8,r10        FDXMW
subq $1,r12         FDXMW
(OVENSTÅENDE VIRKER IKKE)
```

Ud over den tydelige tekst, som indikerer det, kan vi overbevise os selv om at ovenstående ikke virker. Vi har en data afhængighed mellem læsningen fra lageret og første addition og vi ved fra den tidligere uformelle beskrivelse at læsning fra hukommelse sker i `M` fasen. Men i plottet laver vi additionen i `X` samtidig med `M`; altså før vi har værdien til rådighed.

For at undgå dette er vi nødt til at tilføje afhængighederne til vores instruktioner. Det kan vi skrive på følgende måde:

- Aritmetik `op  a b`: `depend(X,a), depend(X,b), produce(X,b)`
- Læsning `movq (a),b`: `depend(X,a), produce(M,b)`
- Skrivning `movq b,(a)`: `depend(X,a), depend(M,b)`

Her står at aritmetiske instruktioner er afhængige af, at værdierne for både `a` og `b` er klar til fase `X`, samt at de producerer deres resultat til register `b` i fase `X`, så resultatet kan bruges fra starten af den efterfølgende fase. Læsning fra hukommelsen kræver at adressen der skal læses fra register `a` er klar til fase `X` (husk at vi har beregningen af adressen i `X` fasen, selvom læsningen først foregår i `M` fasen), mens resultatet af læsningen til register `b` er klar i fase `M` altså til fase `W`. Ved skrivning til hukommelsen skal adressen i register `a` være klar til fase `X`, mens værdien først skal være klar til fase `M`. Skrivning til hukommelsen har ikke noget resultat.

Vær opmærksom på at ovenstående implementerer en arkitektur med fuld forwarding. Altså at alle værdier kan bruges umiddelbart i næste clock periode i alle efterfølgende instruktioner; dvs. før de reelt set er skrevet tilbage til registerfilen. Hvis vi i stedet ville have en maskine uden forwarding, ville alle værdier bliver produceret til fase `W`, hvor vi reelt skriver værdien tilbage.

Eksempel: Data afhængigheder

Lad os nu definere det korrekte afviklingspot for eksemplet. Først, lad os dog opsummere alt vi har defineret for maskinen:

- Tilgængelige ressourcer: `F:1`, `D:1`, `X:1`, `M:1`, `W:1`

	Instruktion	Faser	Dataafhængigheder
Aritmetik	`op  a b`	`FDXMW`	`depend(X,a), depend(X,b), produce(X,b)`
Læsning	`movq (a),b`	`FDXMW`	`depend(X,a), produce(M,b)`
Skrivning	`movq b,(a)`	`FDXMW`	`depend(X,a), depend(M,b)`

```
                012345678901    -- Beskrivelse
movq (r10),r11  FDXMW           -- produce(M,r11)
addq $100,r11    FDDXMW         -- depend(X,r11), produce(X,r11), stall i D
movq r11,(r10)    FFDXMW        -- Stall i F, depend(X,r11)
addq $8,r10         FDXMW       -- Forsinket start på F
subq $1,r12          FDXMW      --
```

Bemærk hvorledes instruktion nr. 2 bliver forsinket en clock periode i sin `D`-fase, fordi den afhænger af `r11` som bliver produceret af den forudgående instruktion der har en latenstid på 2 clock-perioder.

Dette vil give en `CPI = 6/5 = 1,2`.

In-order udførsel af instruktioner

Men hov! Vi har lige fundet ud af at sidste instruktion ikke har dataafhængigheder til de øvrige, så hvorfor kan vi ikke spare en clock periode ved at lave:

```
                012345678901    -- Beskrivelse
movq (r10),r11  FDXMW           -- produce(M,r11)
addq $100,r11    FDDXMW         -- depend(X,r11), produce(X,r11), stall i D
movq r11,(r10)    FFDXMW        -- Stall i F, depend(X,r11)
addq $8,r10         FDXMW       -- Forsinket F
subq $1,r12          X          -- <-- kunne være smart (X er jo ledigt i
denne clock-periode), men hvordan?
```

Vi har måske lidt svært ved at se, hvordan en maskine overhovedet skulle kunne konstrueres således at ovenstående afviklingsrækkefølge kunne finde sted og en maskine er nødt til at læse instruktionerne i den rækkefølge, som er specificeret i vores program.

Vi tydeliggør denne begrænsning: Hver fase gennemføres i instruktions-rækkefølge.

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

Vi har overholdt dette i tidligere eksempler. Vi kan tjekke det ved at når vi læser hver søjle oppefra, skal vi se faserne bagfra.

I det her tilfælde kan vores oversætter forbedre situationen, ved at flytte sidste instruktion frem. Dermed kan vi opnå ovenstående udførsel:

```
                01234567890     -- Beskrivelse
movq (r10),r11  FDXMMW          -- produce(M,r11)
subq $1,r12      FDXXMW         --
addq $100,r11     FDDXMW        -- depend(X,r11), produce(X,r11), stall i D
movq r11,(r10)     FFDXMMW      -- Stall i F, depend(X,r11)
addq $8,r10          FDXXMW     -- Forsinket F
```

Vi kan altså nøjes med at benytte 5 clock perioder og får dermed en `CPI = 5/5 = 1,0`.

Kontrolafhængigheder

Vi modellerer effekten af hop, kald og retur ved at forsinke `F`. For betingede hop skelner vi mellem om hoppet tages eller ej.

Vi udtrykker effekten ved tildelinger til en ny tidsvariabel: NextF.time Og for enhver instruktion gælder altid at F.time >= NextF.time

```
call a,b:   FDXMW
ret  a:     FDXMW
cbcc a,b,x: FDXMW
```

Her er nogle mulige regler for kontrol-instruktioner i en simpel pipeline som beskrevet ovenfor

```
Instruktion  Taget  Effekt
Call         ja     produce(F+2,pc)
Ret          ja     produce(F+3,pc)
CBcc         nej    produce(F+1,pc)
             ja     produce(F+3,pc)
```

Husk at `pc` er vores specielle register til at pege på næste instruktion, så ovenstående peger på hvor hurtigt næste instruktion kan være klar.

Herunder ses to gennemløb af en indre løkke, hvor hop tages til sidst

```
                     012345678901234567
loop: movq (r10),r11   FDXMW                 -- produce(M,r11)
      addq $100,r11     FDDXMW               -- depend(X,r11), produce(X,r11)
      movq r11,(r10)     FFDXMW              -- depend(X,r10),
depend(M,r11), produce(M,r11)
      addq $8,r10          FDXMW             -- produce(X,r10)
      cbl  r10,r12,loop     FDXMW               -- produce(F+3, pc) (hop
taget)
loop: movq (r10),r11           FDXMW            -- produce(M,r11)
      addq $100,r11             FDDXMW          -- depend(X,r11),
produce(X,r11)
      movq r11,(r10)             FFDXMW         -- depend(A,r10),
depend(M,r11), produce(M,r11)
      addq $8,r10                  FDXMW        -- produce(X,r10)
      cbl  r10,r12,loop             FDXMW
```