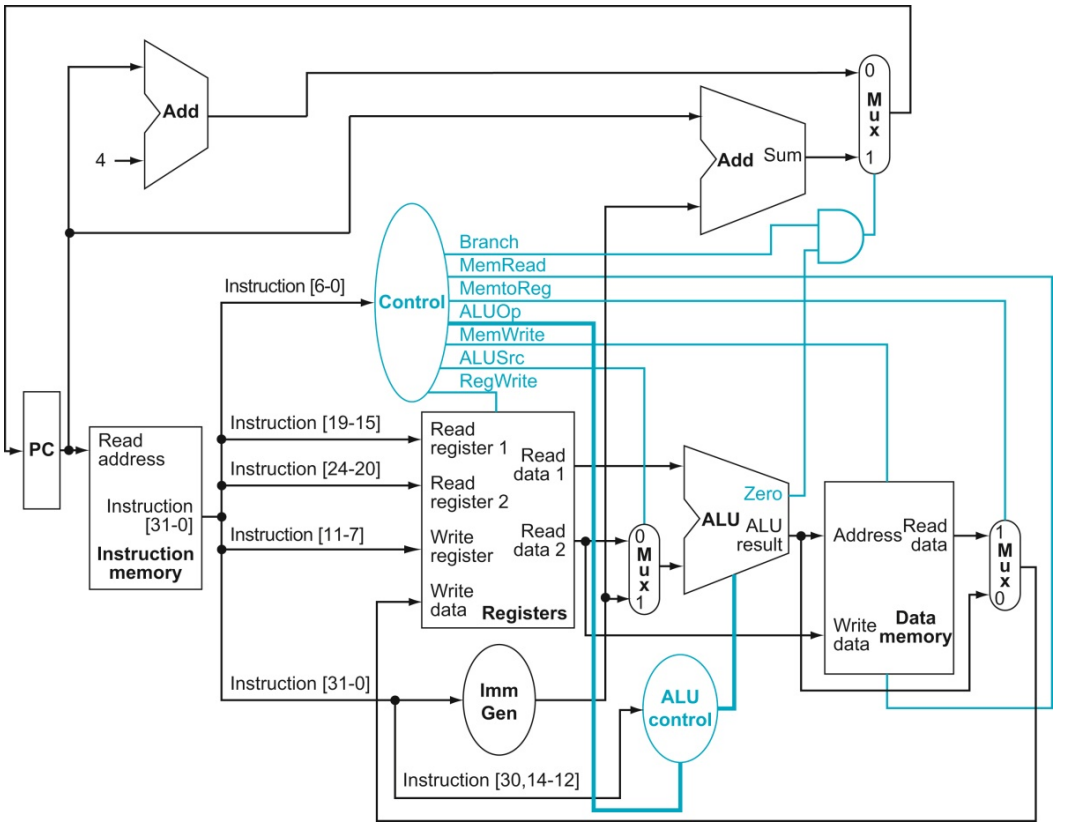
Pipelining - Agenda

1. Hvad er pipelining?
2. Hvilke problemer skal man løse når man laver en pipeline
3. Hvilke teknikker tager man til hjælp
4. Ambitiøs? Flere instruktioner per clock?
5. Realisme - CODs pipeline er forældet

Single cycle mikroarkitektur



Den længste signalvej bestemmer clock-frekvensen. Alle instruktioner bliver så langsomme som den langsomste.

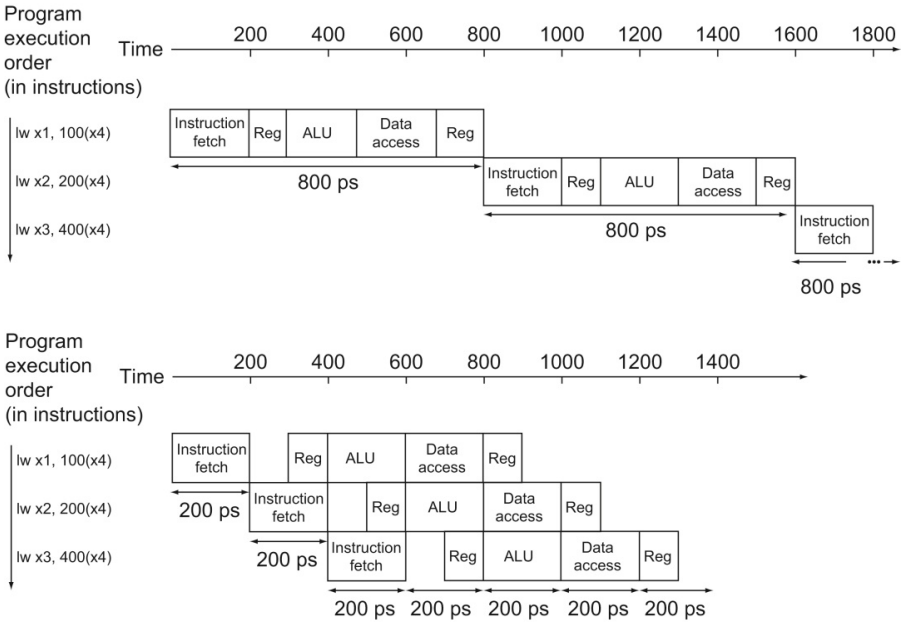
Hvad er pipelining

Pipelining går ud på at arrangere udførelsen af instruktioner som et samlebånd. Den klassiske pipeline som man ofte ser i lærebøgerne har 5 trin:

- 'Fe' for "fetch" - hentning af instruktion
- 'De' for "decode" - afkodning
- 'Ex' for "execute" - udførelse (ALU)
- 'Me' for "memory" - læsning fra lageret
- 'Wb' for "writeback" - opdatering af registre

Ovenstående elementer findes også i en single-cycle mikroarkitektur. Hvis vi for eksemplets skyld antager at hvert trin i samlebåndet ovenfor ville tage 1ns, så ville en single-cycle mikroarkitektur køre med 200MHz.

Fra single-cycle til pipelined datavej



Vi pumper flere instruktioner igennem (stort set) den samme maskine....

Vi inddeler vores single-cycle datavej i flere "trin", så hvert trin kan indeholde en instruktion. Hvert trin skal helst tage cirka lige lang tid. Så siger man at pipelinen er "balanceret".

Pipelining

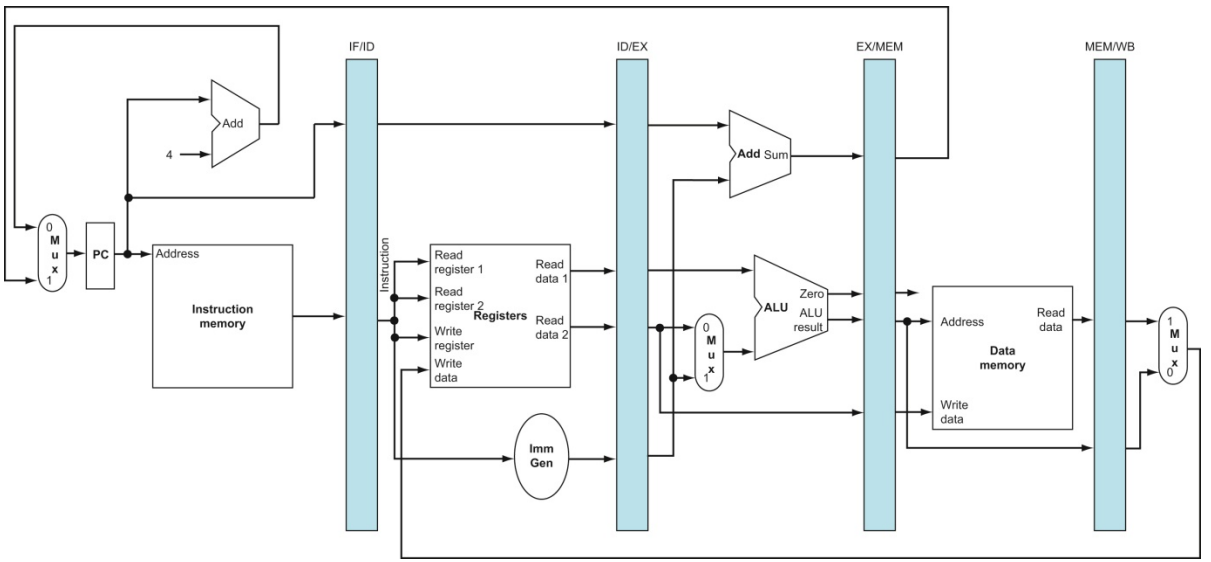
Pipelinens trin skal adskilles af registre - ellers går det galt - men så kan trinnene udføres samtidigt, men med forskellige instruktioner i hvert sit trin.

```
insn1  Fe De Ex Me Wb
insn2     Fe De Ex Me Wb
insn3        Fe De Ex Me Wb
insn4           Fe De Ex Me Wb
insn5              Fe De Ex Me Wb
```

Antag at hvert "trin" tager 0.2 nanosekund. Så kan samlebåndet bevæge sig med op mod 5 GHz. Udefra ser det ud som om de fleste instruktioner udføres på en enkelt clock, men i virkeligheden er den samlede udførelsestid for en instruktion (let) forøget.

En pipelined mikroarkitektur

Simpel fem-trins pipeline (uden bypass/forwarding)



Bemærk hvordan der er indskudt registre (såkaldte pipeline-registre) foran afkoderen, ALUen, data-cachen og til sidst, før skrivning til registrene.

Structural hazards

Det bliver ret besværligt, hvis samme hardware stump skal bruges fra flere forskellige trin i pipelinen. Så skal man koordinere brugen. Det kaldes en "structural hazard".

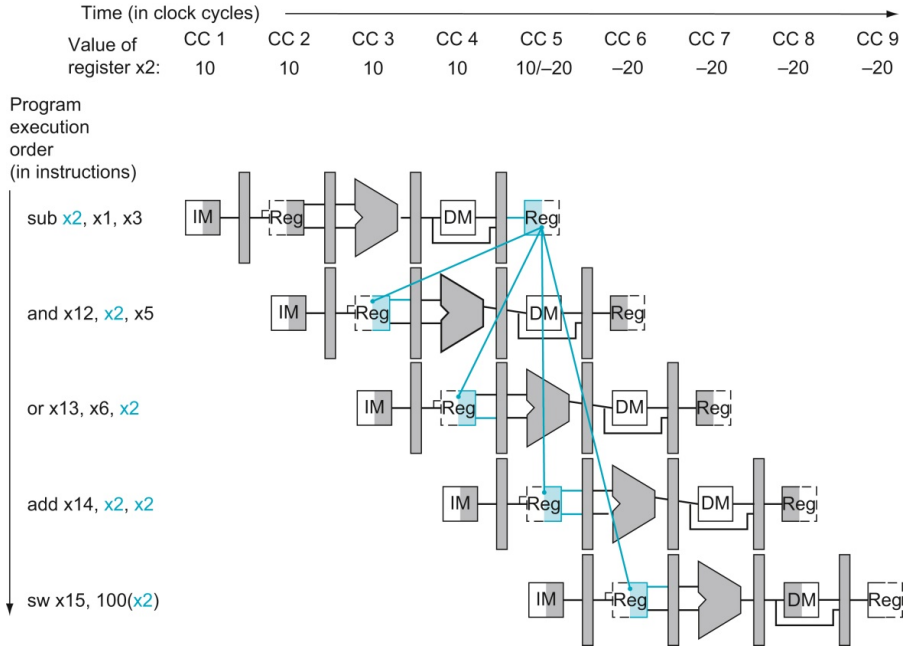
Moderne maskiner har separat instruktions- og datacache blandt andet for at undgå at skulle koordinere tilgang til en enkelt meget hyppigt brugt lager-klods.

RISC-V er designet til at lette pipelinede implementationer

- Instruktioner har samme længde
- Hver instruktion bruger kun en resource (e.g. adder, datacache) en gang

Data hazards

Vi vil tilstræbe at hver instruktion kan bruge resultatet af den umiddelbart forudgående instruktion. Det er vi vant til fra vore single-cycle maskine. Det er bare ikke så nemt:



Men ups - værdien af X2 (produceret af første instruktion) er ikke blevet skrevet til registeret på det tidspunkt, hvor den efterfølgende instruktion skal læse fra registeret. Det kaldes en "data hazard".

Løsning af data hazards ved at "stalle"

En triviel løsning er at bremse pipelinen for de senere instruktioner indtil det er sikkert at fortsætte. Det kaldes et "stall".

```
insn            Fe De Ex Me Wb
sub x2,x1,x3       Fe De Ex Me Wb
and x12,x2,x5         Fe De De De Ex Me Wb
insn                     Fe Fe Fe De Ex Me Wb
```

Men det koster gevaldigt meget på ydeevnen

Data afhængigheder - løsning i software

Simpelt! Compileren må indsætte instruktioner der er uafhængige!

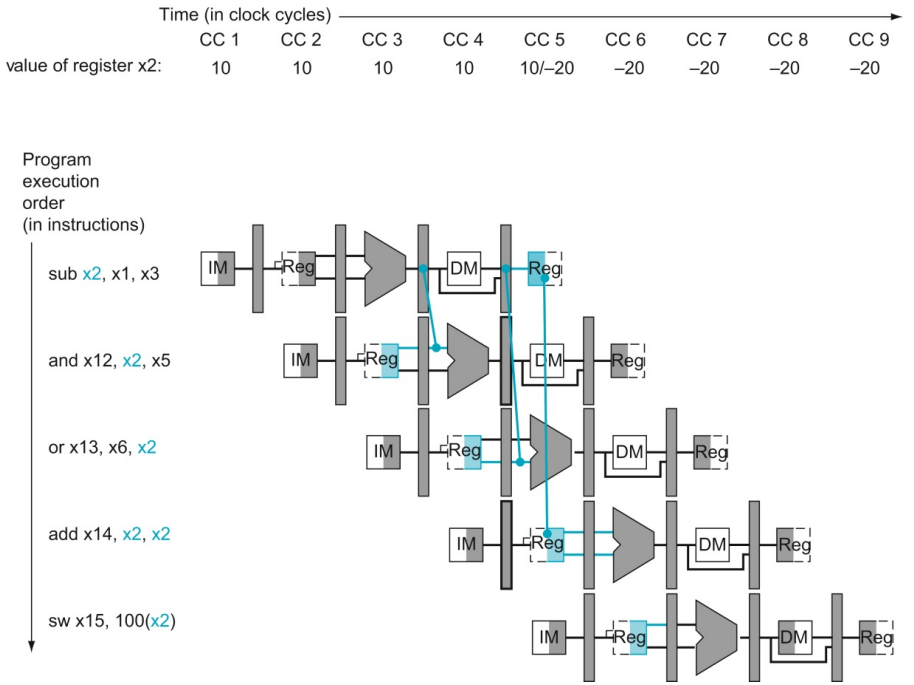
```
insn            Fe De Ex Me Wb
sub x2,x1,x3       Fe De Ex Me Wb
uafh insn             Fe De Ex Me Wb
uafh insn                Fe De Ex Me Wb
and x12,x2,x5               Fe De Ex Me Wb
insn                           Fe De Ex Me Wb
```

De to instruktioner der skal være uafhængige siges at befinde sig i et "delay slot" eller "shaddow" af den foregående instruktion.

Det er sjældent muligt at finde nyttige instruktioner til at fylde i sådan et "delay slot". Så skal compileren indsætte "nop" instruktioner.

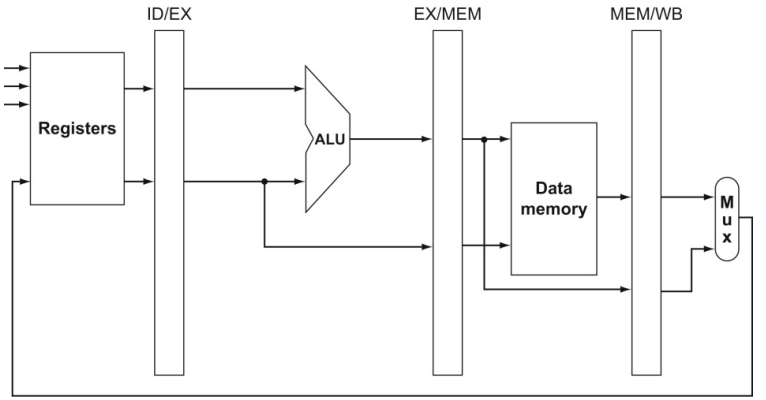
Data afhængigheder - løsning i hardware

Hardware løsning: Vi tilføjer dedikerede forbindelser fra resultat-siden af ALUen til der hvor værdierne skal bruges.

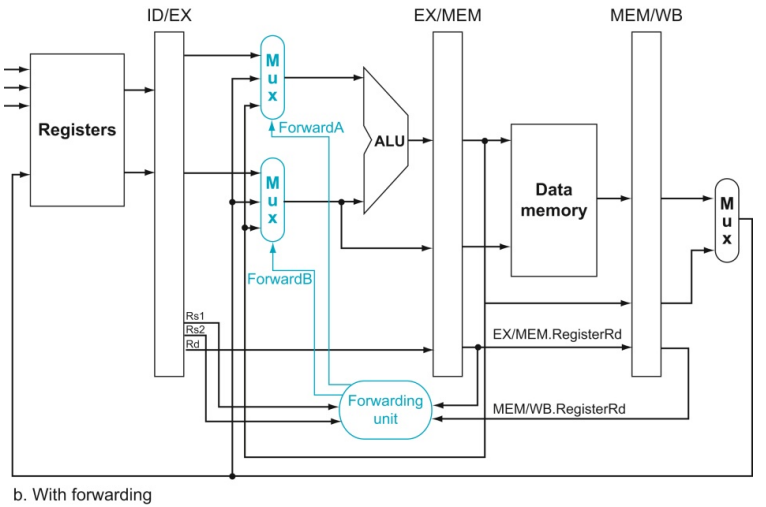


Det kaldes "bypassing" eller "forwarding". Det vil koste lidt på clockfrekvensen i forhold til hvis man ikke har bypassing. Men gevinsten er stor.

Tilføjelse af forwarding



a. No forwarding



b. With forwarding

Langsommere instruktioner

Det er ikke alle instruktioner der udføres i et enkelt trin - kun de aritmetisk/logiske (men IKKE multiplikation).

Læsning fra lageret har først et resultat efter 'Me' trinnet:

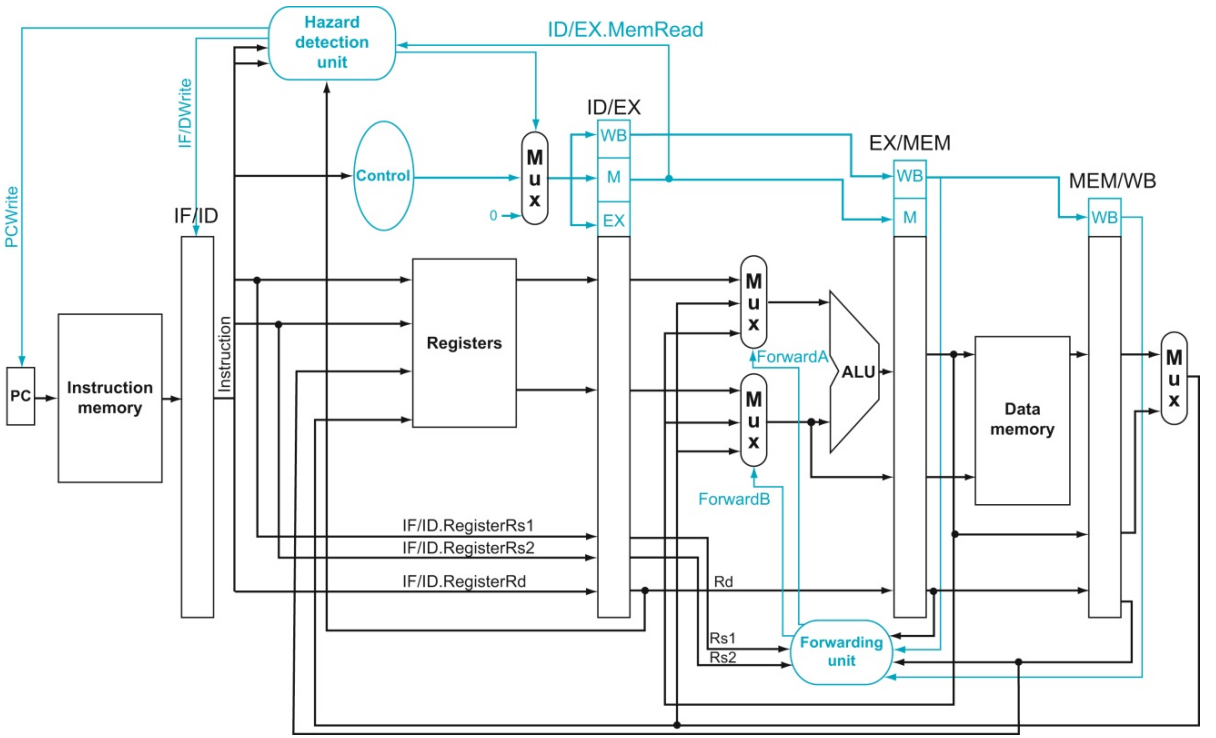
```
insn                Fe De Ex Me Wb
lw x12,40(x8)          Fe De Ex Me Wb
addi x14,x12,400          Fe De De Ex Me Wb
insn                         Fe Fe De Ex Me Wb
```

Her er der ikke noget at gøre. Selv med "forwarding" (eller "bypassing") må den afhængige instruktion forsinkes indtil data når frem.

Vi siger at load-instruktionen har en "skygge" på en instruktion. Hvis vi kan finde en uafhængig instruktion at placere i skyggen behøver vi ikke at bremse pipelinen.

Stall

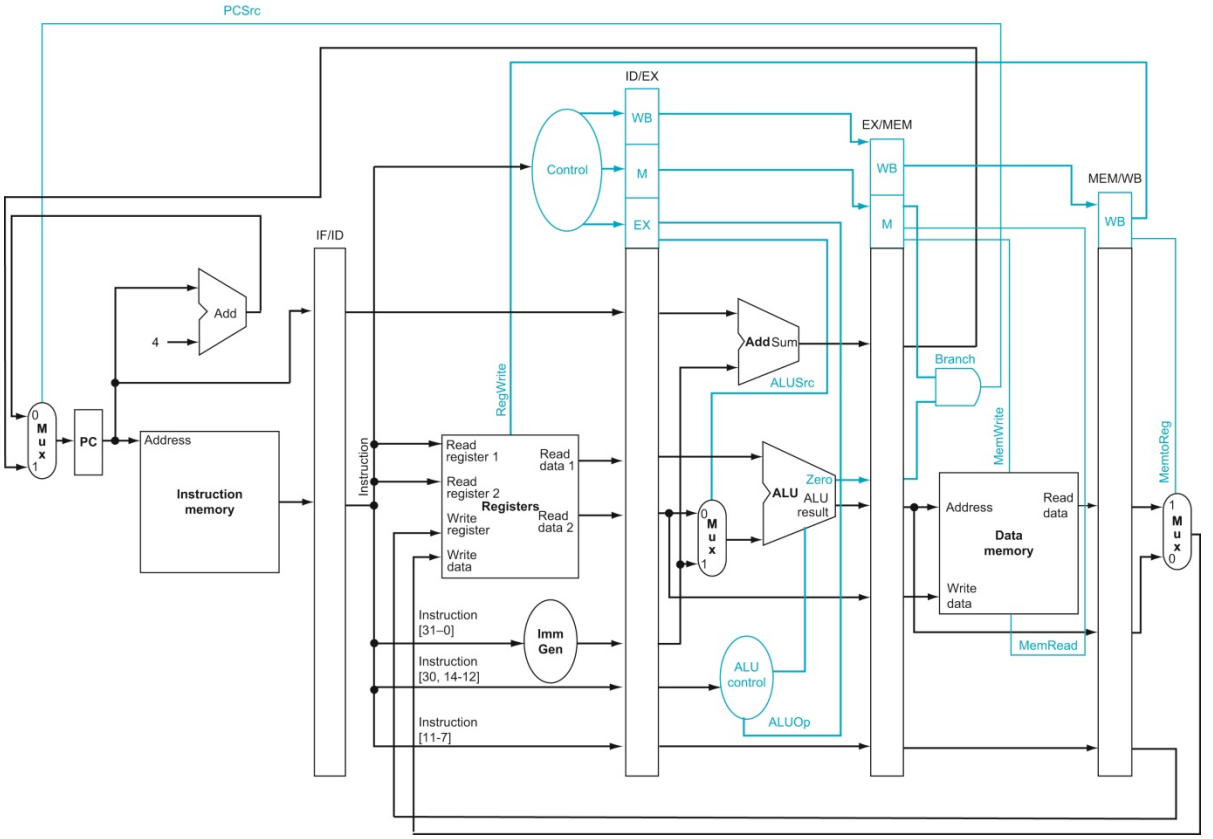
Hvordan bremser man flowet i en pipeline (staller) ?



Man undlader opdatering af de første trin i pipelinen og passerer en nul-operation ind i de efterfølgende trin som fortsat opdateres.

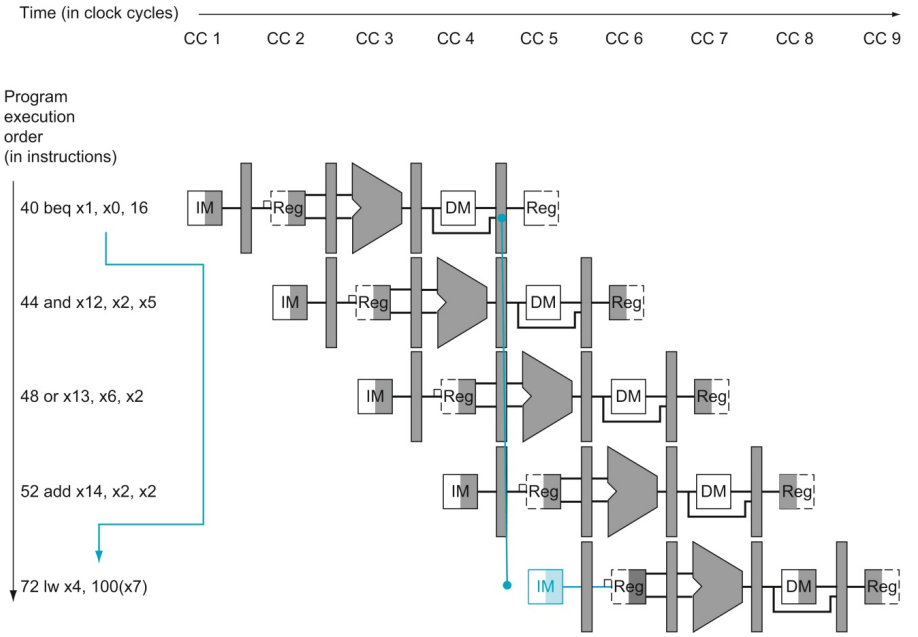
Betingede hop

Pipelining af betingede hop er en udfordring. Vi kan afkode og beregne adressen der eventuelt skal hoppes til i 'Ex'-trinnet (muligvis 'De'). Men hoppet afgøres først sidst i Ex, og derpå skal der bruges tid til at et styresignal kan styre den mux der udvælger den nye værdi til PC-registeret.



Betingede hop

Den ubehagelige konsekvens....



Her koster hoppet 4 clock cykler hvis det tages, 1 hvis det ikke gør.

Superscalar pipelining

Fint ord for mere end en instruktion per clock

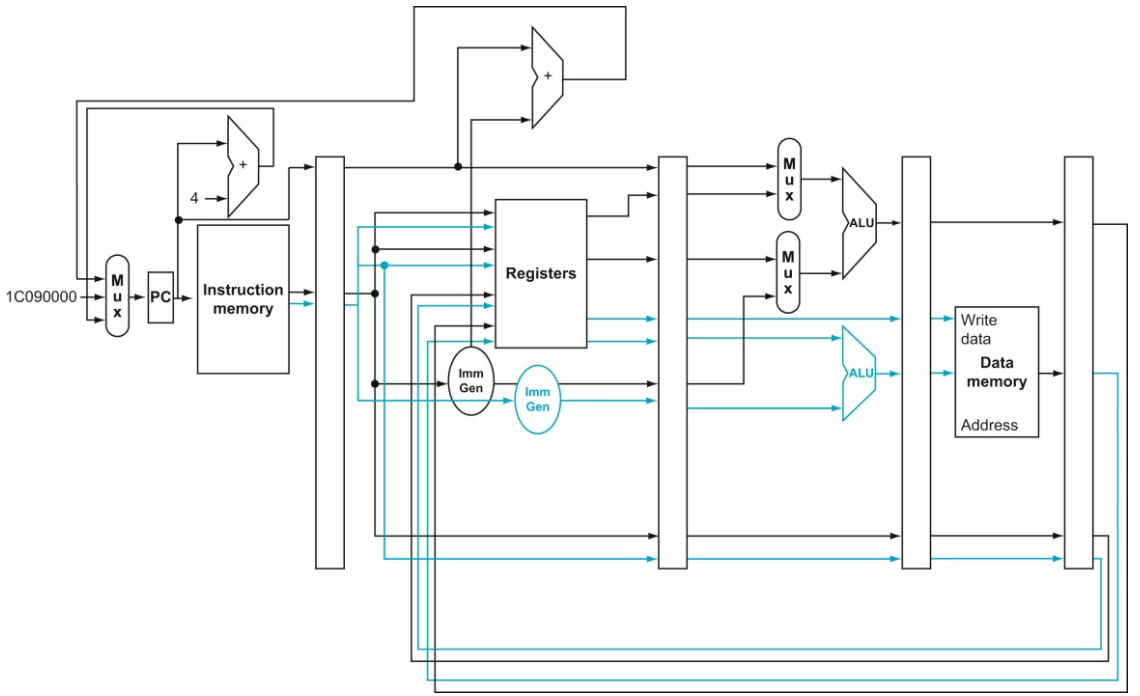
Det begynder med følgende observation: cache adgang koster meget hardware. Hvis vi har lavet hardware med mulighed for en adgang pr clock, så lad os udnytte det! Skaler resten af maskinen så der typisk er brug for cachen hver clock.

Ambition for en to-vejs superskalar:

```
insn 0      Fe De Ex Me Wb
insn 1      Fe De Ex Me Wb
insn 2         Fe De Ex Me Wb
insn 3         Fe De Ex Me Wb
insn 4            Fe De Ex Me Wb
insn 5            Fe De Ex Me Wb
```

2-vej superscalar

Et muligt design:



Kun en adgang til cache pr clock. To ALUer, men kun den ene kan håndtere immediates eller beregne forskydninger fra PC.

Structural Hazards

Vi har potentielt to instruktioner pr clock men kun en mulig tilgang til datacachen per clock. Så hvis de to instruktioner der hentes samtidigt begge skal tilgå lageret, så må vi stalle den ene:

```
lw  x3,40(x8)    Fe De Ex Me Wb
lw  x4,44(x8)    Fe De De Ex Me Wb
lw  x5,48(x8)       Fe De De Ex Me Wb
addi x8,x8,12          Fe De Ex Me Wb
```

Der er ofte også kun mulighed for at håndtere et hop/kald/retur per clock, men det er ikke nogen alvorlig begrænsning for en to-vejs superskalar.

Data hazards i en superskalar pipeline

I en superskalar pipeline er der langt flere mulige "stalls" på grund af data afhængigheder. Betragt flg afvikling på en 2-vejs superskalar pipeline:

```
lw x3,40(x8)    Fe De Ex Me Wb
add x3,x3,x7    Fe De De De Ex Wb
```

Forsinkelsen på en cyklus er den samme som i en simpel pipeline. Men betydningen for ydeevne er relativt større (1,5 instruktion i gennemsnit mod 1,0 for den simple pipeline).

```
lw x3,40(x8)    Fe De Ex Me Wb
uafh insn       Fe De Ex Me Wb
uafh insn          Fe De Ex Me Wb
add x3,x3,x7       Fe De De Ex Wb
```

eller

```
lw x3,40(x8)    Fe De Ex Me Wb
uafh insn       Fe De Ex Me Wb
add x3,x3,x7       Fe De De Ex Wb
insn               Fe De Ex Me Wb
```

Betingede hop i en superskalar pipeline

Betragt udførelsen af et betinget hop der tages i en to-vejs superskalar pipeline

```
insn                Fe De Ex Me Wb
beq x5,x6,target    Fe De Ex Me Wb
insn ej udført         Fe De Ex
insn ej udført         Fe De Ex
insn ej udført            Fe De
insn ej udført            Fe De
insn ej udført               Fe
insn ej udført               Fe
target insn                     Fe De Ex Me Wb
```

Et taget hop koster 6 tabte muligheder for at udføre instruktioner.

Nogen burde gøre noget.....

En mere realistisk pipeline

Det bliver værre....COD tegner et alt for rosenrødt billede.

I nyere CMOS teknologi er kommunikation relativt set blevet langsommere end beregning. Derfor er den simple 5-trins pipeline fra COD ikke i balance. Det tager *meget* længere tid at tilgå de primære caches end at lægge tal sammen.

Recap: Lageret er organiseret i et hierarki. Tættest på ALU og registre placeres L1-caches. Længere ude L2-caches. Og i mange tilfælge også L3 caches. Når man skal læse noget fra lageret kigger man først i L1, så L2, så L3 før man "går off-chip" og snakker med lagerblokke længere væk.

Men selvom L1-cachen er lille, så er den stadig for langsom til at kunne tilgås på en enkelt clock cyklus. Følgende er relativt almindeligt:

- L1: 16KB-64KB, 2-3 clock cykler
- L2: 256KB-1MB, 11-20 clock cykler
- L3: 1-4MB, 20-40 clock cykler
- Lager: GBytes, 100-200 clock cykler.

Lad os kigge på konsekvenserne af langsommere cache tilgang.

En pipeline med 3 cycle cache opslag

Langsommere opslag i lageret påvirker både hentning af instruktioner og hentning af data. Vi antager at cache tilgang kan pipelines i tre trin som vi kalder for "Fa","Fb" og "Fc" for instruktionshenting og "Ma", "Mb" og "Mc" for datatilgang.

```
insn               Fa Fb Fc De Ex Ma Mb Mc Wb
lw x12,40(x8)         Fa Fb Fc De Ex Ma Mb Mc Wb
addi x14,x12,400         Fa Fb Fc De De De De Ex Ma Mb Mc Wb
insn                        Fa Fb Fc Fc Fc Fc De Ex Ma Mb Mc Wb
```

Vi skal nu finde TRE uafhængige instruktioner mellem lw og den addi.

3 cycle cache opslag (II)

Ændringer i programrækkefølgen (kald, retur, hop) bliver dyrere

Her ses udførelsen af et betinget hop:

```
insn                Fa Fb Fc De Ex Ma Mb Mc Wb
beq x5,x6,target       Fa Fb Fc De Ex Ma Mb Mc Wb
insn ej udført            Fa Fb Fc De Ex
insn ej udført               Fa Fb Fc De
insn ej udført                  Fa Fb Fc
insn ej udført                     Fa Fb
insn ej udført                        Fa
target insn                              Fa Fb Fc De Ex Ma Mb Mc Wb
```

Prisen er nu 5 cykler for et taget hop, stadig en cyklus for et ikke taget hop.

Den gennemsnitlige afstand mellem hop er 6 instruktioner! Vi kører på halv hastighed af hvad vi gerne vil

Hvis i tvivl - GÆT!

Over tid har man udviklet forskellige teknikker til at klare hop, kald og retur i lange pipelines - her er tre af dem:

- "branch target caching" - en lille cache der associerer en del af adressen på et hop med de første instruktioner der hoppes til. Gør det muligt at have instruktionerne på target-addressen klar så snart man afkoder et hop.
- Retur forudsigelse - en lillebitte hardware stak (f.eks. 8 addresser) placeret tidligt i pipelinen så man kan få retur adressen uden at vente på at den læses fra registre, endsige forwardes fra en netop overstået læsning fra lageret.
- Hop forudsigelse - forudsigelse af om et betinget hop skal tages eller ej baseret på kontekst.

Jo dybere/bredere pipeline, jo vigtigere er disse teknikker

Vi nøjes med at kigge lidt nærmere på hop forudsigelse

Hop-forudsigelse

Ofte kan man forudsige om et hop skal tages eller ej. F.eks. vil hop der lukker en løkke (og hopper tilbage til begyndelsen af løkken) oftest skulle tages.

En simpel forudsigelse kunne være at hop der går bagud skal tages, men dem der hopper til en senere adresse skal ikke (denne strategi kaldes "btfnt" - backward taken, forward not taken). Lidt mere fleksibelt er det hvis en compiler kan markere sin forudsigelse så hardwaren kan se den.

Desværre er hop ret dynamiske, så statiske (altså compile-time) metoder har kun begrænset succes. Der skal dynamiske metoder til.

Dynamisk hop-forudsigelse

Dynamisk hop-forudsigelse opsamler data fra hoppenes historie og brugder det til at forudsige den fremtidige opførsel.

Den simpleste udgave betragter hvert hop for sig. Man knytter en to-bit tæller til hver hop, i praksis ved at lave en række af tællere og bruge nogle bits fra PC'en til at vælge en tæller. Hver tæller opsumerer hoppets historie:

```
00 hop ikke taget (strongly not taken)
01 hop almindeligvis ikke taget (weakly not taken)
10 hop almindeligvis taget (weakly taken)
11 hop taget (strongly taken
```

Hver gang et hop afgøres opdateres den matchende tæller, enten i retning mod "hop ikke taget", eller mod "hop taget".

Dette kaldes "local" hop forudsigelse - fordi man betragter hvert betinget hop adskilt fra de andre.

Korrelerende hop-forudsigelse

Hop er ofte korrelerede med andre hop. Det kan man udnytte ved at opsamle hoppenes historie. En simpel fremgangsmåde er at indkode historien i et skifte-register. Når et hop tages skifter man '1' ind i skifteregisteret. Når et hop ikke tages skifter man '0' ind.

Som før har man en tabel af to-bit tællere der opdateres på samme måde som beskrevet for lokale forudsigere.

For at lave en forudsigelse laver man et "hash" af skifteregisteret og PC'en og bruger det til at slå op i tabellen med tællere. Bitvis XOR er en fin hash funktion i det her tilfælde.

Denne forudsiger kaldes "gshare" og kan ofte levere mere end 90% korrekte forudsigelser.

Der findes andre og betydeligt mere omfattende forudsigere der fungerer endnu bedre. Generelt skal man ikke tro at man kan forudsige sine egne hop bedre end maskinen kan.

Se f.eks. https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/

Opsamling

Pyha - vi er nået langt fra den simple single-cycle mikroarkitektur

- Vi har introduceret en simpel pipeline - som kører med 5x højere clock frekvens - og forhåbentligt kan afvikle virkelige programmer ~4x hurtigere
- Vi har fundet ud af, at nej, det var for optimistisk. I moderne CMOS er lager relativt langsommere og CODs eksempel matcher ikke (men er en god introduktion)
- Vi har introduceret en pipeline der er bedre balanceret end CODs
- Vi har også introduceret de mere ambitiøse superskalare pipelines som er vidt udbredte, selv i batteridrevne enheder som smartphones.

Kan en 2-vejs superskalar køre dit program 2x hurtigere end den simplere skalare pipeline? Kan en 4-vejs mon køre det 4x hurtigere?

Det varierer voldsomt fra program til program hvor meget der vindes.

Spørgsmål og Svar