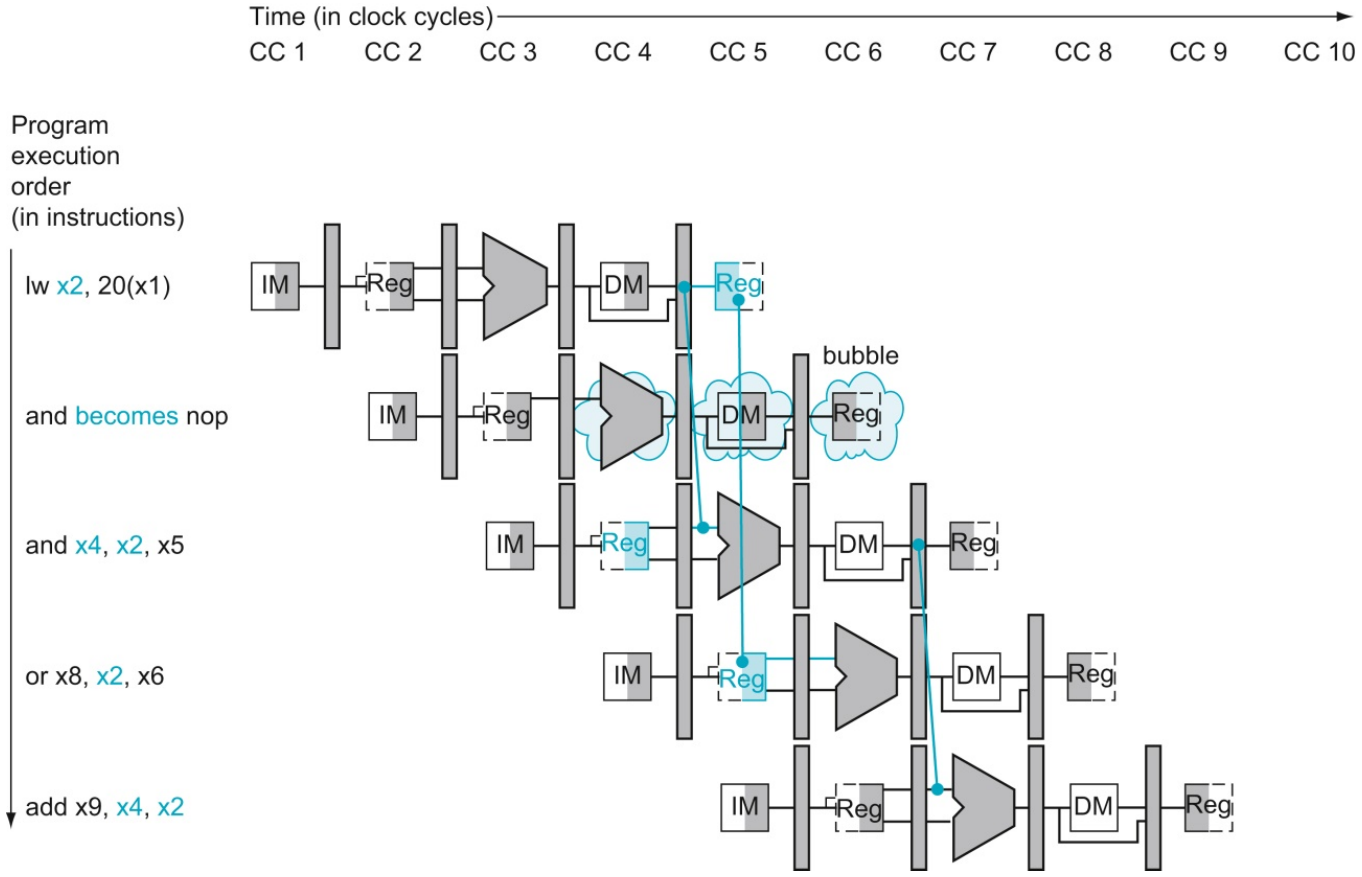
Højtydende mikroarkitektur - Agenda

1. Recap: Udfordringen
2. ILP. Instruction Level Parallelism
3. Out-of-order execution
4. Lidt detaljer fra en virkelig maskine
5. Lidt om afviklingsplot
6. Opsamling

Data afhængigheder i en simpel pipeline

To notationer for det samme:

```
                        1  2  3  4  5  6  7  8  9
lw   x2,20(x1)          Fe De Ex Me Wb
and  x4,x2,x5              Fe De De Ex Me Wb
or   x8,x2,x6                 De Fe De Ex Me Wb
add  x9,x4,x2                    Fe De Ex Me Wb
```



Realisme: Læsning fra 3-cycle cache

Langsommere opslag i lageret påvirker både hentning af instruktioner og hentning af data.

```
                      0  1  2  3  4  5  6  7  8
lw   x11,0(x10)       Fa Fb Fc De Ex Ma Mb Mc Wb
addi x11,x11,100         Fa Fb Fc De De De De Ex Ma Mb Mc Wb
sw   x11,0(x14)             Fa Fb Fc Fc Fc Fc De Ex Ma Mb Mc Wb
addi x10,x10,1                  Fa Fb Fb Fb Fb Fc De Ex Ma Mb Mc Wb
```

Vi kan markere "stall" tydeligere:

```
                      0  1  2  3  4  5  6  7  8
lw   x11,0(x10)       Fa Fb Fc De Ex Ma Mb Mc Wb
addi x11,x11,100         Fa Fb Fc >> >> >> De Ex Ma Mb Mc Wb
sw   x11,0(x14)             Fa Fb >> >> >> Fc De Ex Ma Mb Mc Wb
addi x10,x10,1                  Fa >> >> >> Fb Fc De Ex Ma Mb Mc Wb
```

Læsning fra lageret kaster en skygge på 3 instruktioner.

Hop og 3-cycle cache

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

Her det betingede hop fra sidste forelæsning

```
insn                    Fa Fb Fc De Ex Ma Mb Mc Wb
ble x12,x13,target         Fa Fb Fc De Ex Ma Mb Mc Wb     <-- vi kan afgøre hop i 'Ex'
<shaddow>                     Fa Fb Fc De                 <-- og slå de her instruktioner ihjel
<shaddow+1>                      Fa Fb Fc
<shaddow+2>                         Fa Fb                <-- for ikke at tale om dem her
<shaddow+3>                            Fa
target: insn                              Fa Fb Fc De Ex Ma Mb Mc Wb    <-- og starte hentning hrt
```

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

Sammenfatning

Pipelining er muligvis nemt - men at få max ydeevne er svært.

Overordnet set bliver ydeevnen begrænset af sammenkoblingen af:

1. Længere pipelines som passer til moderne CMOS-teknologi.
2. Sekventiel semantik
3. Data- og kontrol-afhængigheder

De længere pipelines leder nemt til flere stalls fremfor bedre ydeevne.

Så hvad gør vi så?

ILP - Instruction Level Parallelism

Hvis man betragter en sekvens af instruktioner vil man opdage at den oftest indeholder instruktioner, som ikke er afhængige af resultater fra de umiddelbart foregående. Disse instruktioner kunne i princippet udføres tidligere, samtidigt med instruktioner de ikke afhang af. (her et plot for en to-vejs superskalar med realistisk cache tilgang)

```
                0  1  2  3  4  5  6  7  8  9  10 11 12 13
lw   x10,4(x16)  Fa Fb Fc De Ex Ma Mb Mc Wb
addi x10,128     Fa Fb Fc >> >> >> >> De Ex Ma Mb Mc Wb     // må vente
lw   x11,8(x16)     Fa Fb Fc >> >> >> De Ex Ma Mb Mc Wb     // ingen afhængighed!
addi x11,-128       Fa Fb Fc >> >> >> >> >> >> >> De Ex Ma ...
```

Instruktion nummer 3 er uafhængig af de forudgående instruktioner.

Denne egenskab ved programmer kaldes ILP, Instruction Level Parallelism.

Hvis man forestiller sig at muligheden for parallel udførelse KUN var begrænset af data-afhængigheder, så indeholder programmer typisk meget ILP.

begrænsningen fra kontrolafhængigheder

Instruktioner afhænger ikke kun af data. De er også afhængige af tidligere hop, kald og retur.

Et betinget hop kan forsinke efterfølgende instruktioner.

```
lw   x11,0(x10)   Fa Fb Fc De Ex Ma Mb Mc Wb
beq  x11,x7,L1    Fa Fb Fc >> >> >> >> De Ex Ma Mb Mc Wb
addi x5,x6,8                                Fa Fb Fc De Ex Ma Mb Mc Wb
```

Her er den sidste instruktion meget forsinket, fordi hoppet blev afgjort sent. Med hop forudsigelse kan man (ofte) få hentet de rigtige instruktioner tidligere

Men så er problemet hvad man kan tillade sig, når et hop forudsiges korrekt!

```
lw   x11,0(x10)   Fa Fb Fc De Ex Ma Mb Mc Wb
beq  x11,x7,L1    Fa Fb Fc >> >> >> >> De Ex Ma Mb Mc Wb    // hop afgøres sent, men forudsagt korrekt
addi x5,x6,8         Fa Fb Fc >> >> >> De Ex Ma Mb Mc Wb    // hvorfor vente?
```

Hvis første instruktion har et miss i datacachen, kan der blive udført mange instruktioner, før det efterfølgende hop afgøres. Hvad kan man tillade sig at udføre af instruktioner efter hoppet?

Konsekvens: Spekulativ udførelse

Hvis vi vil have max ydeevne er vi nødt til at bygge en maskine der tillader instruktioner at blive udført *før* man afgør hop, som de samme instruktioner har en kontrol-afhængighed på

Det kaldes spekulativ udførelse

```
lw   x11,0(x10)   Fa Fb Fc De Ex Ma Mb Mc Wb
beq  x11,x7,L1    Fa Fb Fc >> >> >> >> De Ex Ma Mb Mc Wb    // hop afgøres sent, men forudsagt korrekt
addi x5,x6,8         Fa Fb Fc >> >> >> De Ex Ma Mb Mc Wb    // hvorfor vente?
```

Men hvad nu hvis hoppet ovenfor blev forudsagt forkert? Den efterfølgende instruktion er allerede udført på det tidspunkt hvor vi afgør hoppet?

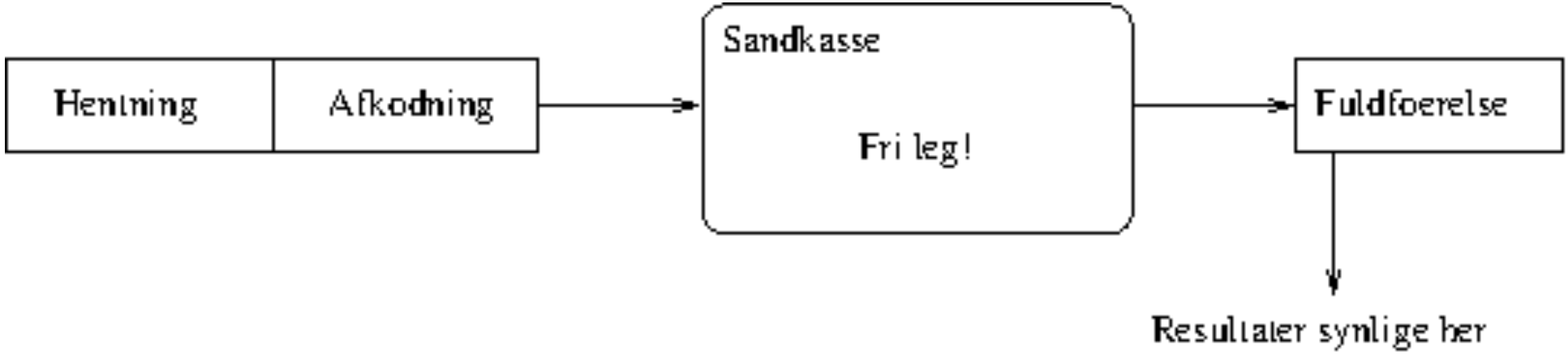
Hvad med denne efterfølgende instruktions eventuelle skrivning til registre? til lageret?

Alle side-effekter må holdes hemmelige, indtil vi faktisk ved om de skal med i udførelsen eller ej!

OOO - Overview

Out-of-order execution, eller "dynamisk udførelse" beror på tre principper:

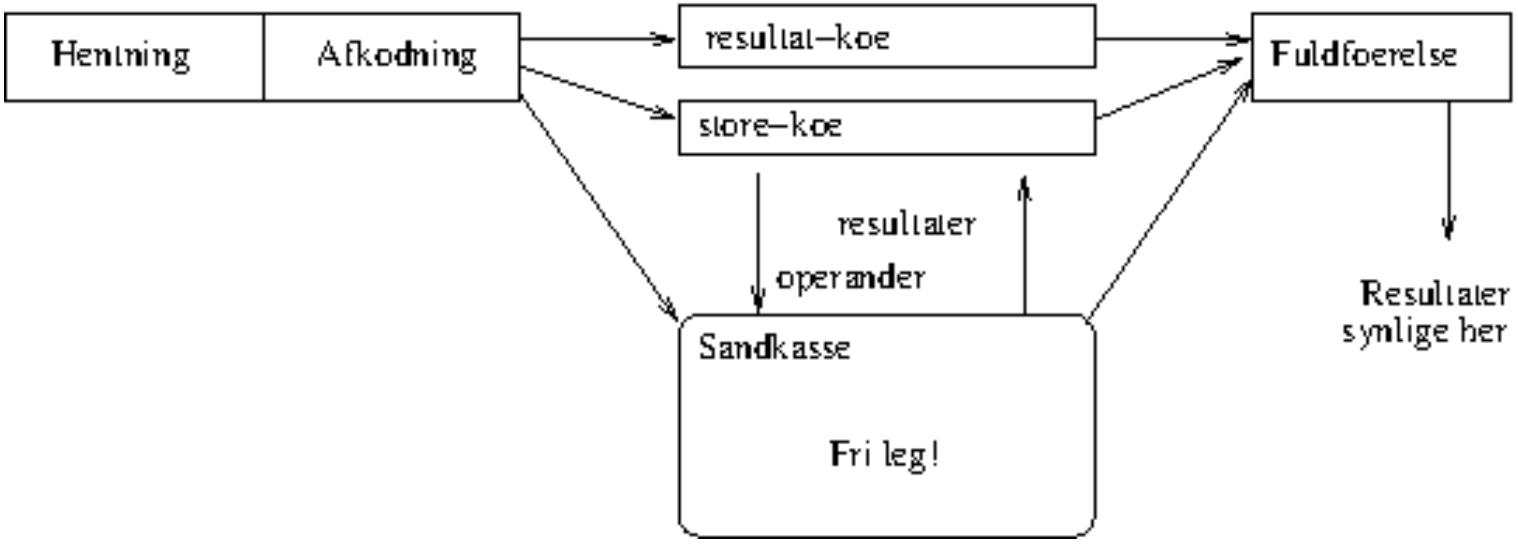
1. Udførelsesrækkefølge fastlægges ud fra afhængigheder mellem instruktioner, ikke deres sekventielle rækkefølge i programmet
2. Spekulativ udførelse: Instruktioner udføres aggressivt i en "sandkasse", alle resultater/side-effekter skjules for omverdenen. "What happens in Vegas stays in Vegas"
3. Forudsigelse af programforløb gør det muligt at "fylde sandkassen" før vi kender programforløbet.



Implementation - en sandkasse

For at kunne lave en sandkasse skal vi isolere effekten af instruktioner indtil vi ved at de skal udføres. Man kunne lagre resultater i to køer:

1. Resultat-kø - udestående skrivninger til registre
2. Store-kø - udestående skrivninger til lageret



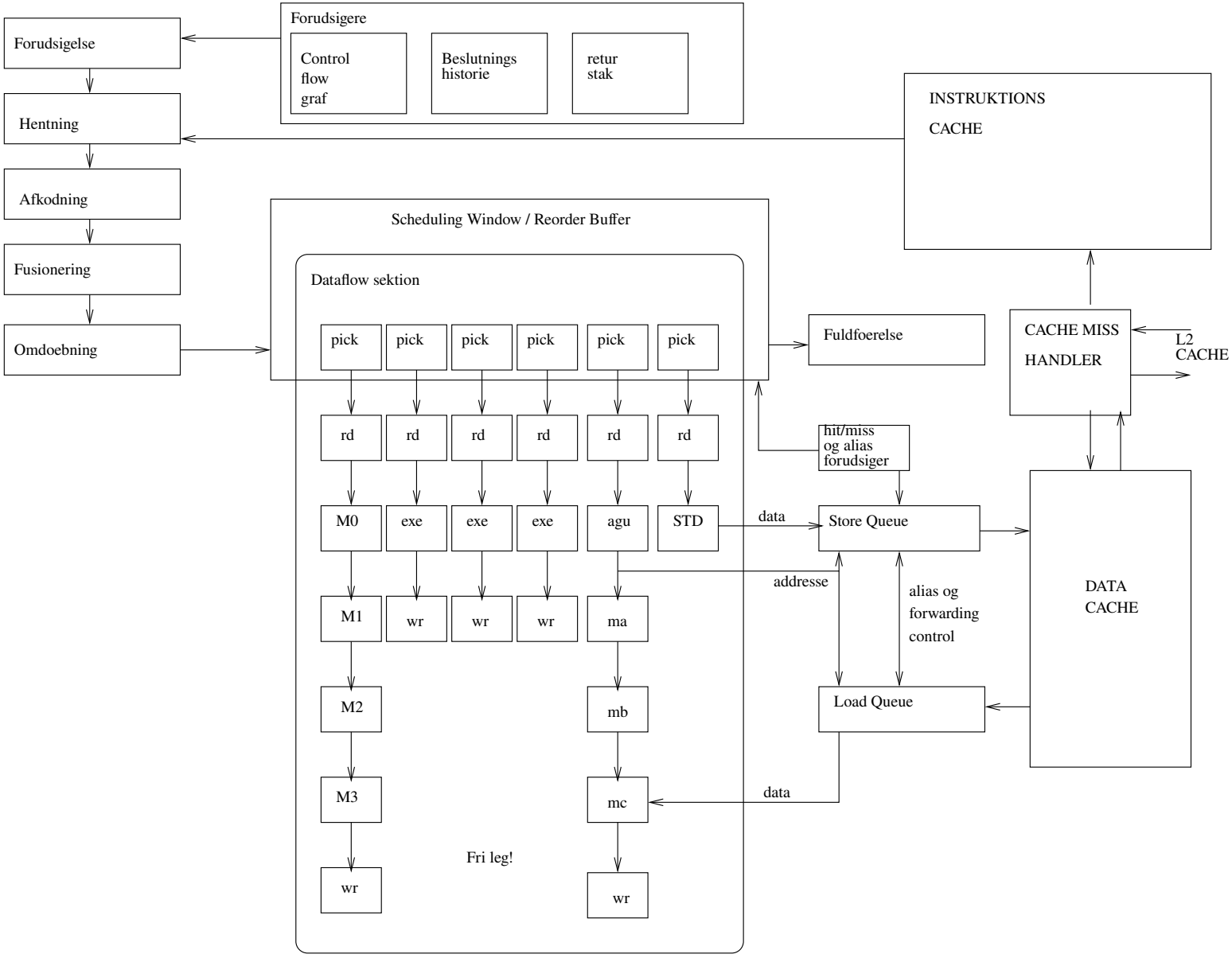
Når man så skal finde indholdet af et register, må man søge igennem resultat-køen og finde den rette skrivning til et givet register nummer (der kan være mange). På samme måde må læsning fra lageret søge gennem store-køen for at finde en eventuel skrivning til lageret.

Centrale elementer af sandkassen

Sandkassen er der hvor vi udfører instruktionerne. Pænt afskærmet fra resten af verden. Om nødvendigt kan vi gå amok og regne forkert, bare vi korrigerer i tide. Der er 4 essentielle komponenter:

- Unik ID til enhver instruktion og måske især dens resultat (Renaming)
  - Ellers har vi ikke en chance for at kunne holde styr på udførelsen
- Planlægning af udførelses-rækkefølge (Scheduling)
  - Vi planlægger så udførelse sker i data-flow rækkefølge
  - Fint ord for at vi vælger de instruktioner først, hvis input er beregnet
- Afskærmning fra omverdenen (Skrive kø, søgbar)
  - Så vi kan "spekulere" i fred - på basis af forudsigelser der kan være forkerte.
- Fuldførelse - når resultater bliver synlige for omverden (Commit, Completion, Retirement)

OOO - Mikroarkitektur - overview



Forudsigelse af programforløb

Programforløbet forudsiges på basis af 3 lagerblokke allerforrest i pipelinen.

- BTB/CFG (branch target buffer) - lille cache der associerer PC med
  - Næste PC hvis et betinget hop tages
  - Næste PC hvis et betinget hop ikke tages
  - PC-efter-kodeblok / Størrelse af kodeblok
  - Er der et betinget hop?
  - Er der kald?
  - Er der retur?
- RAS (return address stack)
- Forudsiger for betinget hop (beslutningshistorien)

Parallelt med at PC en sendes til instruktionscache for instruktionshentning, bruges den også til opslag i BTB og hop forudsiger. Hvis BTB'en siger at koden indeholder et betinget hop, så afgør hopforudsigeren om det skal tages eller ej. Hvis et hop ikke tages, så siger BTB'en om der er kald eller retur. Ved kald skubbes PC-efter-kodeblok på RAS, ved return tages næste PC fra RAS.

Når først det her kredsløb er "trænet" på basis af programmets opførsel, så kan det levere en ny PC for hvert "cycle"

Indkodning af CFG, Eksempel

```
Branch Target Buffer      Instruction Cache

   entry   = 10074        10074:       addi    sp,sp,-16
   next[0] = 100b0        10078:       sw      ra,12(sp)
   next[1] = 10074        1007c:       sw      s0,8(sp)
   after   = 1009c        10080:       sw      s1,4(sp)
   predict, call          10084:       mv      s0,a0
                          10088:       li      a5,1
                          1008c:       bgeu    a5,a0,100b0 <fib+0x3c>
                          10090:       addi    a0,a0,-1
                          10094:       auipc   ra,0x0
                          10098:       jalr    -32(ra) # 10074 <fib>

   entry   = 1009c        1009c:       mv      s1,a0
   next[0] = xxx          100a0:       addi    a0,s0,-2
   next[1] = 10074        100a4:       auipc   ra,0x0
   after   = 100ac        100a8:       jalr    -48(ra) # 10074 <fib>
   no predict, call

   entry   = 100ac        100ac:       add     a0,s1,a0
   next[0] = xxx          100b0:       lw      ra,12(sp)
   next[1] = xxx          100b4:       lw      s0,8(sp)
   after = xxx            100b8:       lw      s1,4(sp)
   no predict, return     100bc:       addi    sp,sp,16
                          100c0:       ret
```

Dynamisk hop-forudsigelse (fra onsdag)

Dynamisk hop-forudsigelse opsamler data fra hoppenes historie og bruger 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/

Forudsigelse - Præcision vs Hastighed

Det kræver en hvis mængde lager til BTB og hopforudsiger for at få høj præcision. Desværre er store lager elementer langsommere end små, og høj præcision kan ikke fås i løbet af en enkelt clock cyklus.

Det er derfor normalt at man laver to forudsigelseskredsløb a la det tidligere beskrevne.

- Et der er optimeret til at kunne levere en forudsigelse pr clock. Det er så ikke så præcist.
- Et der er væsentligt større og optimeret til at være så præcist som budgettet tillader, men som er langsommere.

Det store kredsløb bruges så til at korrigere forudsigelserne fra det lille hurtige kredsløb.

Fusionering

Fusionering omskriver instruktionspar til interne instruktioner som kan udføres hurtigere. Oftest skal begge instruktioner i et par have samme destinations register. Eksempler:

```
lui x3,immA
addi x3,x3,immB         -->  li x3,immA+immB

asl x5,x4,imm
add x5,x5,x6            -->  shiftadd x5,x4,x6,cnst

add x5,x4,x3            -->  lw2 x5,imm(x4,x3)
lw x5,imm(x5)
```

Det er selvfølgelig kun muligt, hvis maskinens datavej understøtter de nye interne instruktioner. I ovenstående indgår f.eks. en adder der kan tage 3 input eller kan skifte det ene input mod venstre.

Intel IA64 fusionerer ofte CMP og Bcc (compare og betinget hop). Det gør RISCV ikke, da det allerede er en enkelt instruktion.

Unik ID til enhver instruktion - og resultat.

For at kunne "schedulere" instruktionerne (planlægge deres udførelse) er det nøvendigt at vi kan identificere dem (og alle operander) unikt. Vi skal bringe instruktionerne på en form: (A,B)->op->C, hvor A, B og C er unikke ID'er og 'op' angiver hvilken beregning der skal udføres.

Dette opnås ved *registeromdøbning*. Instruktionernes registre som vi kender dem fra assembler niveauet bliver erstattet med nye interne register-numre, der identificerer resultat og operander. Et meget større sæt registre er til rådighed, så der er plads til resultater fra alle de instruktioner der kan være under udførelse.

Vi skelner mellem logiske registre (dem programmøren kan se) og fysiske registre.

Instruktioner placeres i rækkefølge i en kø. Hver plads i køen har associeret et nyt fysisk registernummer, som bruges til at identificere netop den instruktion.

Registeromdøbning

Forestil dig vi kun har 4 logiske (Xn) registre og 8 fysiske registre (pn).

Vi har fortegnelse over resultater fra længst fuldførte instruktioner i form af et fysisk registernummer for hvert logisk registernummer. Vi har også en tilsvarende fortegnelse for den fremtidige tilstand når nyeste instruktion engang vil nå enden af pipelinen.

```
fuldført:  x0: p0, x1: p1, x2: p2, x3: p3
fremtidig: x0: p0, x1: p1, x2: p2, x3: p3

insn        resultat            -> omskrevet instruktion
-           p4
-           p5
-           p6
-           p7
```

Når en ny instruktion ankommer, placeres den på første frie plads:

```
insn           resultat            -> omskrevet instruktion
ADD  x2,x1,x1  p4
```

Registeromdøbning (II)

Instruktionens logiske registre skal nu omdøbes. For hvert logisk register slår man det tilsvarende fysiske registernummer op i tabellen over den fremtidige tilstand

```
fuldført:  x0: p0, x1: p1, x2: p2, x3: p3
fremtidig: x0: p0, x1: p1, x2: p2, x3: p3

insn           resultat            -> omskrevet instruktion
ADD  x2,x1,x1  p4                     ADD p1,p1 -> p4
```

Derpå opdaterer man tabellen over den fremtidige tilstand

```
fremtidig: x0: p0, x1: p1, x2: p4, x3: p3
```

Registeromdøbning (III)

Vi kan fortsætte med flere instruktioner, indtil køen er fuld:

```
fuldført:  x0: p0, x1: p1, x2: p2, x3: p3
fremtidig: x0: p0, x1: p1, x2: p4, x3: p3

insn           resultat            -> omskrevet instruktion
ADD  x2,x1,x1  p4                     ADD p1,p1 -> p4
SUB  x2,x2,x3  p5                     SUB p4,p3 -> p5
ADD  x2,x1,x1  p6                     ADD p1,p1 -> p6
ADD  x3,x2,x0  p7                     ADD p6,p0 -> p7
```

Bemærk hvordan alle værdier herefter er identificeret med et unik resultat nummer efter omdøbningen.

```
fremtidig: x0: p0, x1: p1, x2: p6, x3: p7
```

Registeromdøbning (IV)

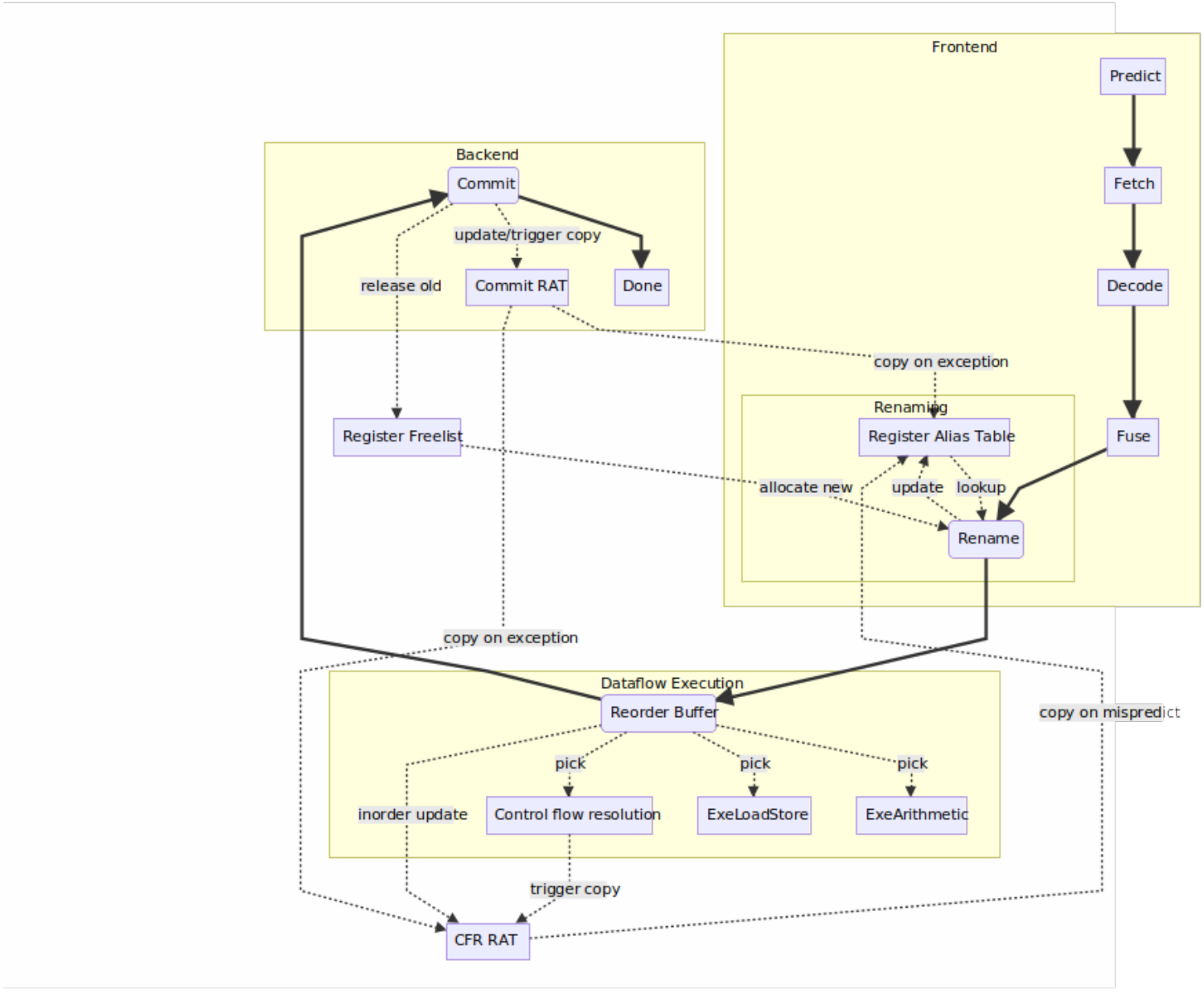
Når en instruktion har produceret en værdi og er blevet den ældste i listen, så kan den fuldføres ("commit", "retire"). Det gøres ved at opdatere tabellen over den fuldførte tilstand med et nyt fysisk register. Forinden læses det tidligere fysiske registernummer og tilføjes til køen til senere brug:

```
Fuldføres ADD x2,x1,x1 omdøbt til ADD p1,p1 -> p4

fuldført:  x0: p0, x1: p1, x2: p4, x3: p3
fremtidig: x0: p0, x1: p1, x2: p4, x3: p3

insn           resultat            -> omskrevet instruktion
SUB  x2,x2,x3  p5                     SUB p4,p3 -> p5
ADD  x2,x1,x1  p6                     ADD p1,p1 -> p6
ADD  x3,x2,x0  p7                     ADD p6,p0 -> p7
-              p2
```

Registeromdøbning - Flow



Registeromdøbning og præcise undtagelser (exceptions)

Registeromdøbning er designet til at understøtte præcise "exceptions". Hvis en instruktion "fejler", f.eks. tilgår en ulovlig virtuel addresse, så markeres den som fejlet. Når instruktionen er blevet den ældste og skal fuldføres ("commit") vil den trigge en exception.

På det tidspunkt er maskinen fuld af efterfølgende instruktioner i varierende stadier af udførelse. Disse instruktioner aborteres alle sammen.

Maskinen klargøres til håndtering af undtagelsen ved at alle de udestående register-opdateringer slettes fra køen og "fuldført" kopieres til "fremtidig".

Registeromdøbning og fejlforudsigelser

Registeromdøbningen skal også kunne håndtere fejlforudsigelser af hop, kald eller retur. En simpel implementation kunne bruge mekanismen til håndtering af exceptions, men det indebærer at vente til den fejlforudsagte instruktion bliver den ældste. Fejlforudsagte hop er meget hyppigere end exceptions, så vil vil gerne have en hurtigere mekanisme.

En mulig løsning kan være at knytte en "backup" af "fremtidig" tabellen til hver eneste forudsagte instruktion. Hvis instruktionen så viser sig at være fejl- forudsagt, retablerer man "fremtidig" fra backuppen.

Når et hop viser sig at være fejlforudsagt, så annulleres alle instruktioner der er yngre end hoppet. Det fremgår af vores kø hvilke instruktioner det er. Da alle instruktioner har deres eget unikke nummer (fysiske register), så kan man bruge en bitvektor til at sende information om hvilke instruktioner der skal anulleres til alle afkroge af maskinen, uden risiko for forveksling.

Nye instruktioner fra den korrigerede instruktionsstrøm kan umiddelbart derefter passere omdøbning - de vil se maskinens tilstand som den så ud umiddelbart efter det fejlagtigt forudsagte hop.