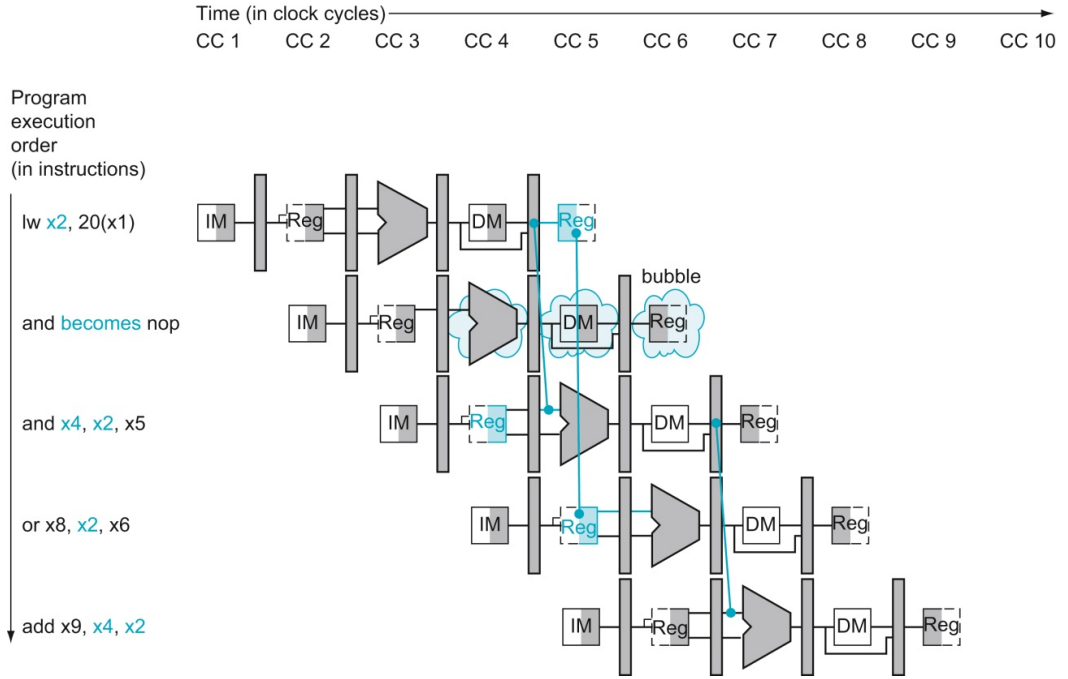
Højtydende mikroarkitektur - Agenda

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 Mm Wb
and  x4,x2,x5           Fe De De Ex Mm Wb
or   x8,x2,x6            De Fe De Ex Mm Wb
add  x9,x4,x2                Fe De Ex Mm 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 ko
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 ko
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?

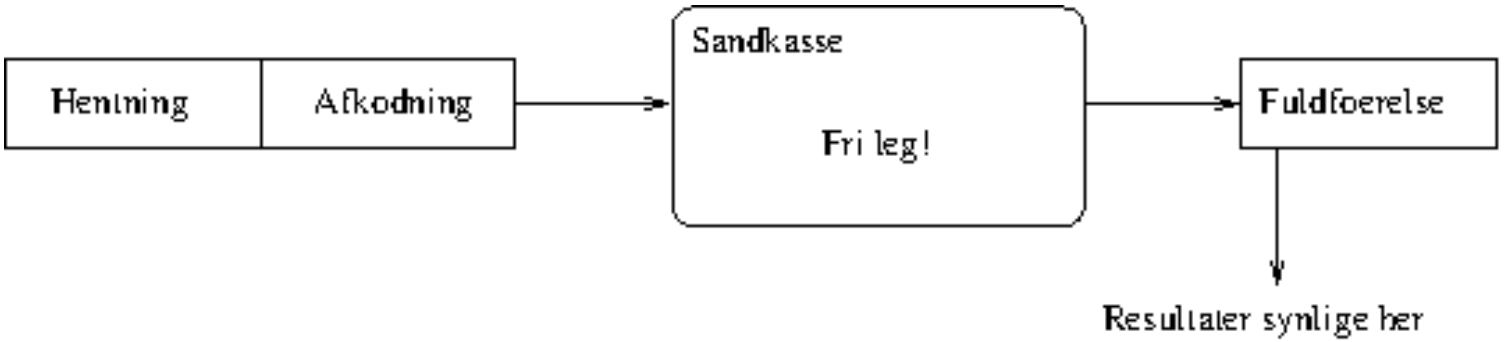
Hvad med denne efterfølgende instruktions eventuelle skrivning til registre? til lageret?

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:

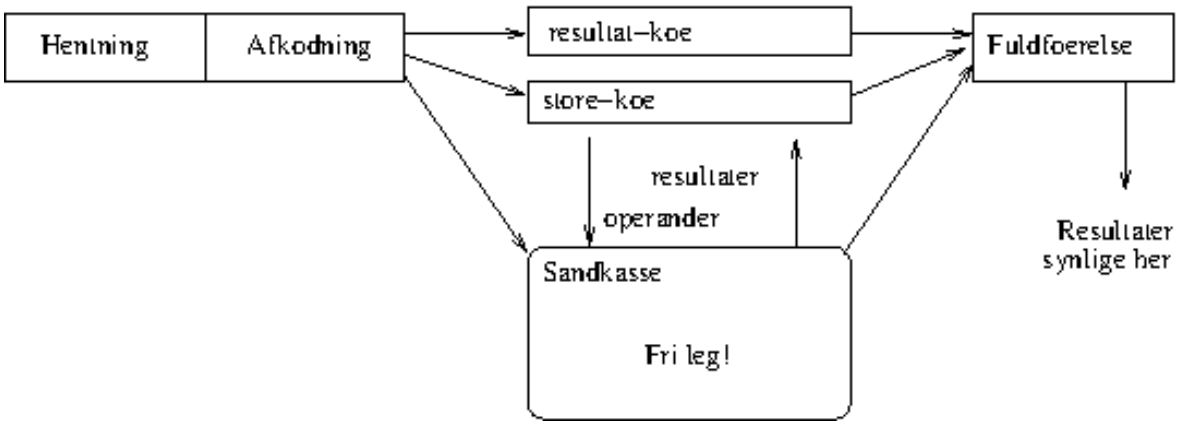
1. Udførelsesrækkefølge fastlægges ud fra afhængigheder mellem instruktioner, ikke deres sekventielle rækkefølge i programmet
2. Spekulativ udførelse: Instruktioner udføres aggressivt i en "sandkasse", alle resultater/side-effekter skjules for omverdenen. "What happens in Vegas stays in Vegas"
3. Forudsigelse 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



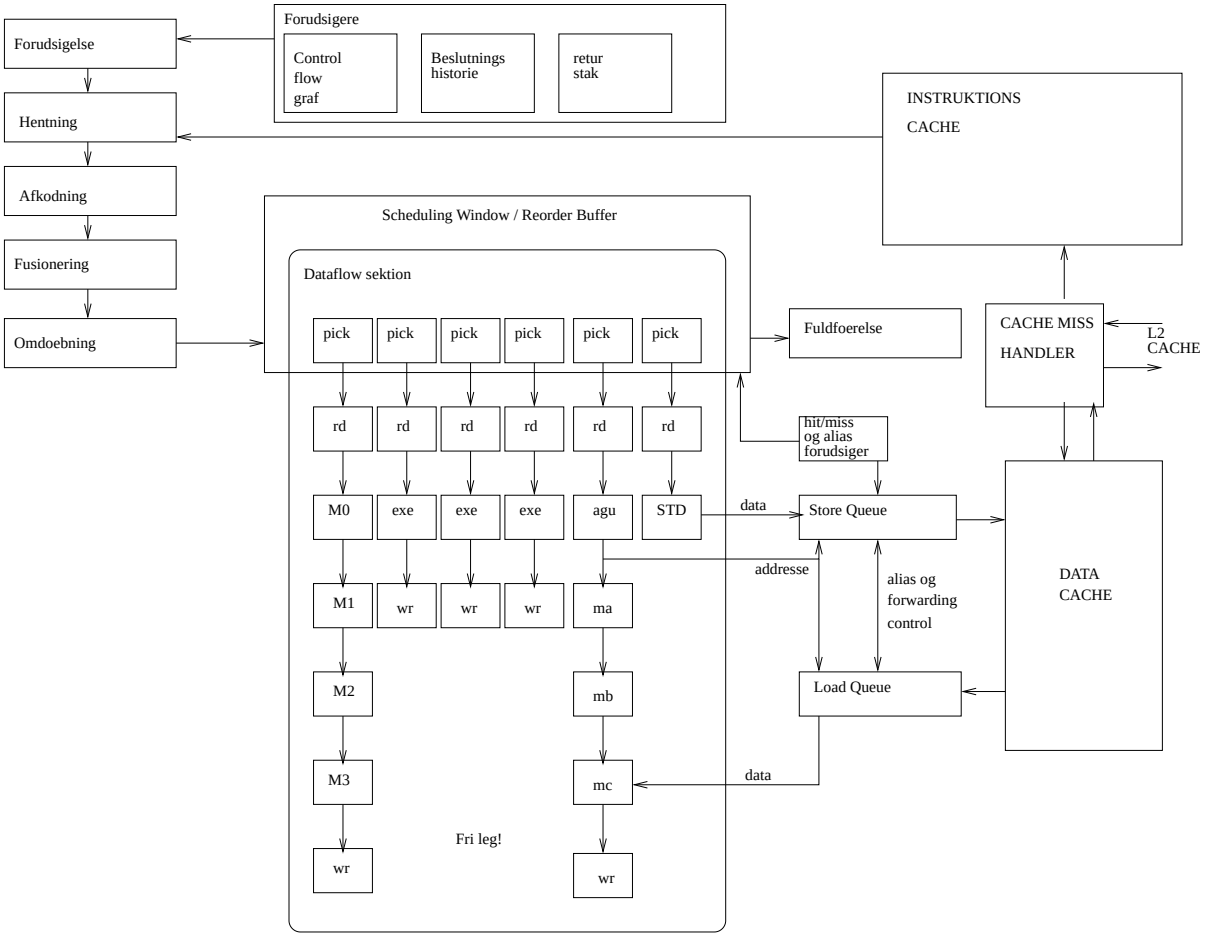
Når man så skal finde indholdet af et register, må man søge igennem resultat-køen og finde den rette skrivning til et givet register nummer (der kan være mange). På samme måde må læsning fra lageret søge gennem store-køen for at finde en eventuel skrivning til lageret.

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
- 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



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 listen 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 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

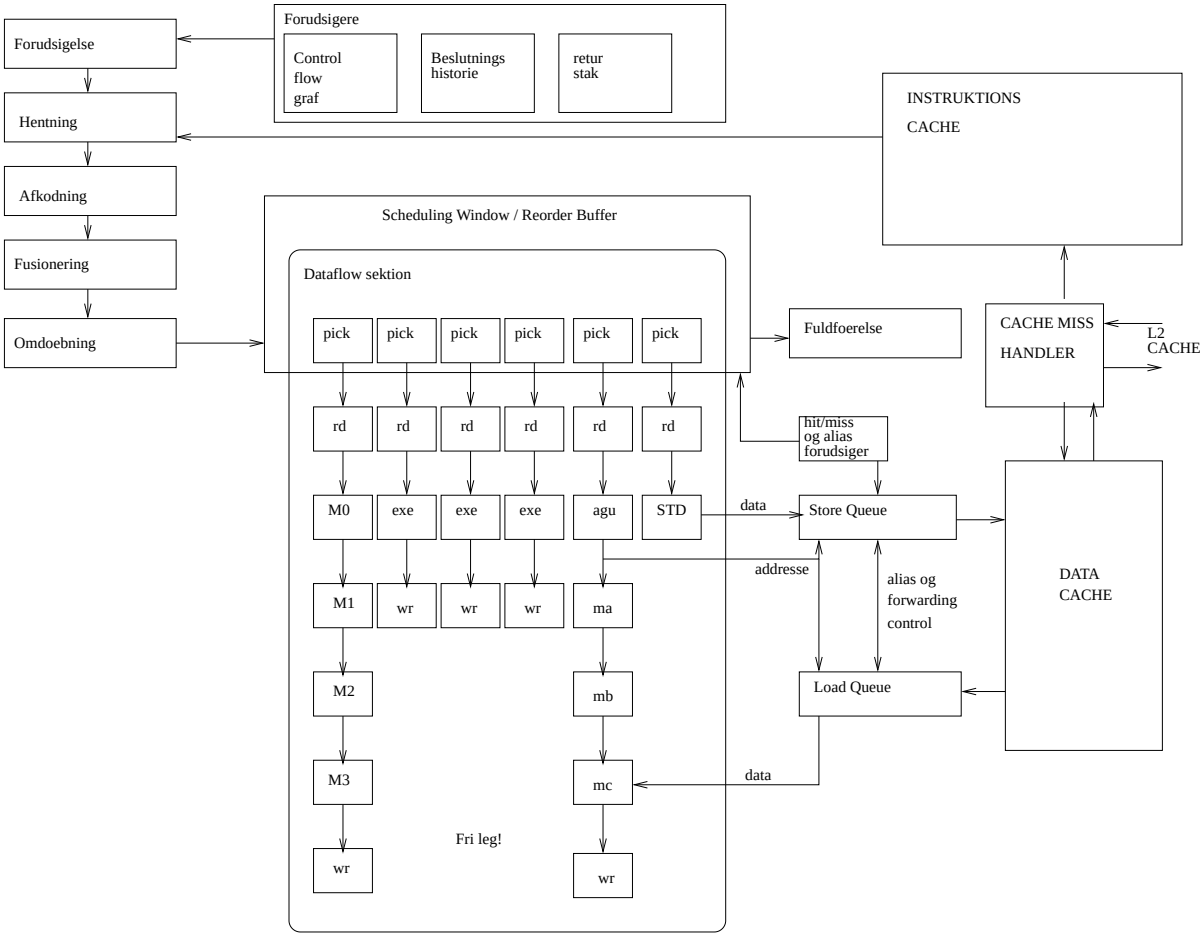
Registeromdøbningen skal også kunne håndtere fejlforudsigelser af hop, kald eller retur. En simpel implementation kunne bruge mekanismen til håndtering af exceptions, men det indebærer at vente til den fejlforudsagte instruktion bliver den ældste. Fejlforudsagte hop er meget hyppigere end exceptions,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

OOO - Mikroarkitektur - overview



Planlægning af udførelsesrækkefølge

Et særligt planlægnings-kredsløb er ansvarligt for at fastlægge hvornår instruktioner skal udføres. Planlægnings-kredsløbet kan bedst opfattes som en form for "aktiv" hukommelse hvori instruktionerne afventer deres operander.

Udover denne hukommelse har kredsløbet en "parat-vektor". Det er en bit-vektor med en bit for hvert fysisk register. Bitten er sat, hvis det fysiske register har modtaget et resultat.

Hver instruktion tjekker *hele tiden* de bit i parat-vektoren, som svarer til de input-værdier de afhænger af.

Når en instruktion således "observerer" alle dens værdier er parat, bliver den ligeledes "parat". Flere instruktioner kan blive parat samtidigt.

Et prioriteringskredsløb udvælger så den instruktion som kan få lov at starte blandt de instruktioner der er parate. Nu er instruktionen "udtaget". Dens resultatnummer vil i en senere clock-cyklus blive blive brugt til at sætte en bit i parak-vektoren

Lad os prøve med et eksempel:

Planlægning

Lad os forestille os at de fire instruktioner fra tidligere netop er passeret gennem registeromdøbning og når til planlægsningstrinnet.

Parat: p0,p1,p2,p3

ADD p0,p1 -> p4 [parat]
SUB p2,p4 -> p5 [ikke parat]
ADD p0,p2 -> p6 [parat]
SUB p6,p3 -> p7 [ikke parat]

Planlæggeren kan vælge mellem den første og den tredje instruktion. Den vælger den første og i næste cycle sættes bit for p4 i parat-vektoren.

Planlægning (II)

I næste cycle er p4 med i parat-vektorn:

Parat: p0,p1,p2,p3,p4

ADD p0,p1 -> p4 [startet]
SUB p2,p4 -> p5 [parat]
ADD p0,p2 -> p6 [parat]
SUB p6,p3 -> p7 [ikke parat]

Så nu bliver instruktion nummer to også parat. Planlæggeren kan så vælge mellem den anden og den tredje instruktion. Den vælger den anden og i næste cycle sættes bit for p5 i parat-vektoren.

Planlægning (III)

Man kan godt bygge sådan nogle kredsløb, så de kan udvælge og starte flere instruktioner samtidigt. F.eks:

Parat: p0,p1,p2,p3

ADD p0,p1 -> p4 [parat]
SUB p2,p4 -> p5 [ikke parat]
ADD p0,p2 -> p6 [parat]
SUB p6,p3 -> p7 [ikke parat]

Og planlæggeren vælger så både den første og den tredje.

Parat: p0,p1,p2,p3,p4,p6

ADD p0,p1 -> p4 [startet]
SUB p2,p4 -> p5 [parat]
ADD p0,p2 -> p6 [startet]
SUB p6,p3 -> p7 [parat]

I den efterfølgende cycle udvælges så både anden og fjerde instruktion.

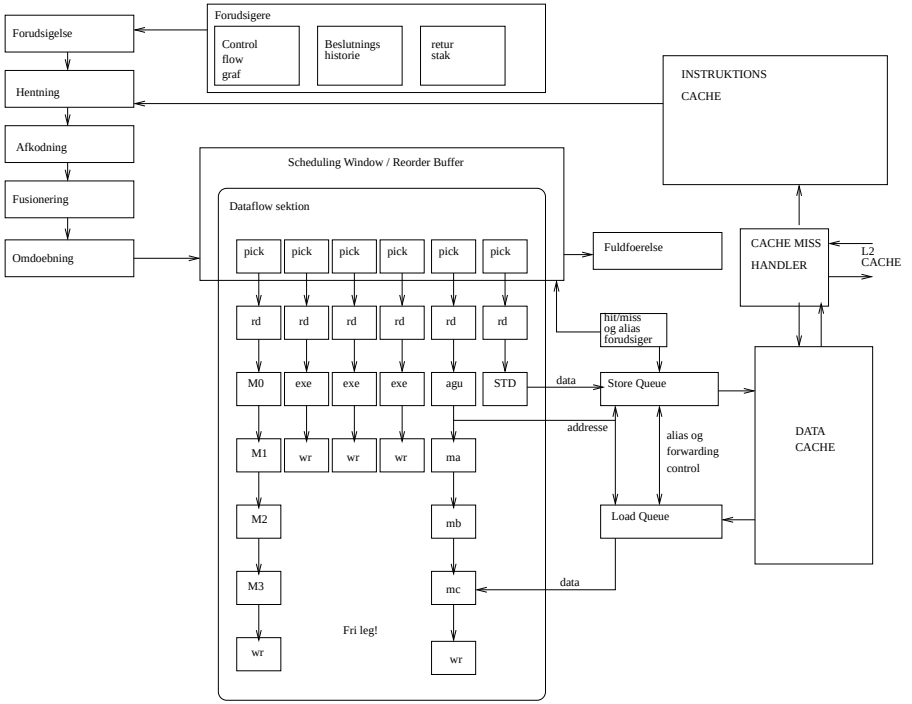
Opsamling fra mandag

Vi kiggede på vanskelighederne med at få max ydeevne ud af pipelines, især når de er mere realistiske og derfor længere end COD's 5-trins pipeline.

Vi begyndte gennemgang af out-of-order execution:

- Overordnet struktur - forende, bagende, dataflow-del
- Spekulativ udførelse og forudsigelse af program-forløb
- Registeromdøbning
- Dynamisk schedulering / Dataflowbaseret udførelse

OOO - Mikroarkitektur - overview




```
add  x12,x7,x3      Fa Fb Fc De Fu Al Rn Qu pk rd ex wb Ca Cb
mul  x11,x12,x9     Fa Fb Fc De Fu Al Rn Qu -- pk rd m0 m1 m2 m3 wb Ca Cb
addi x11,x11,400    Fa Fb Fc De Fu Al Rn Qu -- -- -- -- -- pk rd ex wb Ca Cb
addi x2,x12,32      Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wb -- -- -- -- Ca Cb
```

Læsning fra lageret (Load instruktioner)

Ved cache-hit har vi følgende forløb:

L1 hit: Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc wb Ca Cb

Ved cache-miss stall'er vi IKKE pipelinen. Vi udskyder bare "wb":

L2 hit: Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc -- -- -- -- -- -- -- -- -- wb Ca Cb
L1 hit: Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc wb -- -- -- -- -- -- -- -- Ca Cb

Andre load instruktioner kan udføres samtidigt med at den tidligere load instruktion venter på at få bragt data ind fra L2.

Mange cache-forbiere kan være under behandling samtidigt!

Skrivning til lageret (Store instruktioner)

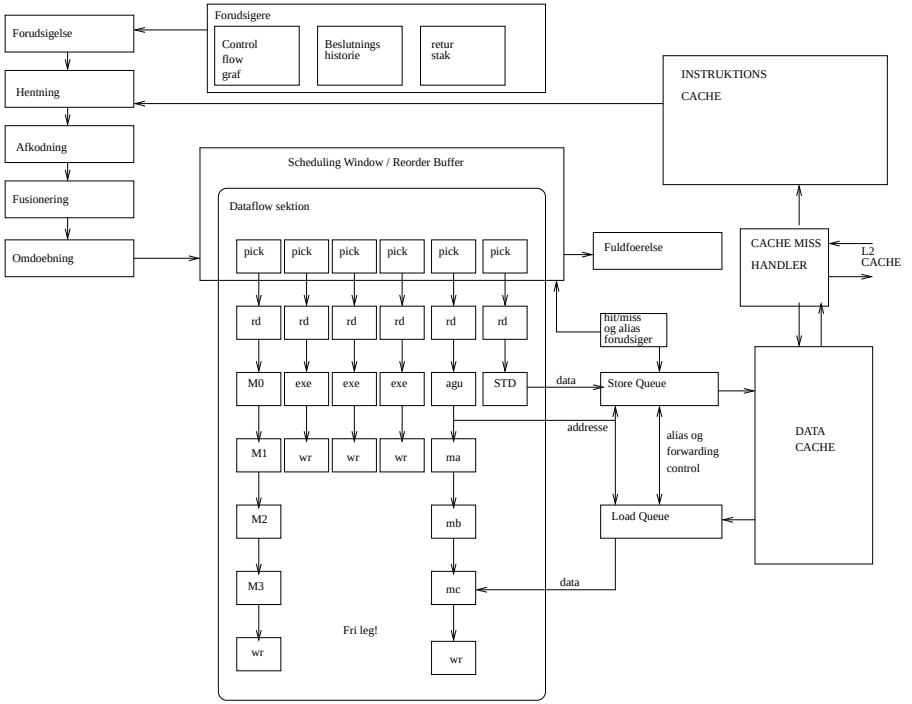
Vanskeligt

- Vi kan ikke lave spekulative skrivninger til lageret.
- Faktisk opdatering må vente til instruktionen fuldføres ("commit")
- Indtil da placeres skrivninger i en store-kø
- Vi viser ikke den endelige skrivning til lageret i vores afviklingsplot

Hver skrivning i store-køen angiver

- Adresse
- Gyldighed af adresse
- Data (word-aligned)
- Gyldighed af data - bitmaske, en bit pr byte

Skrivning til lageret (II)



STORE instruktioner implementeres som to separate operationer

```
lw  x12,36(x3)      Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc wb Ca Cb
sw  x12,40(x3)      Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc Ca Cb
-                             Qu -- -- -- -- pk rd st
```

Store til load forwarding

Betragt følgende sekvens af instruktioner:

```
sb  x7,(x4)
sb  x8,1(x4)
sb  x9,3(x4)
lw  x10,(x4)
```

Hvordan skal vi finde resultatet af load instruktionen til sidst!?

Store til load forwarding (II)

En load skal ikke kun søge i datacachen:

- Før (eller samtidigt med cache opslag) må man søge i store-køen efter skrivninger som overlapper med den læsning man vil lave.
- Der kan allerede være relevante data i store køen.
- Eller der kan være en markering i store køen af at der vil blive skrevet til den ønskede adresse senere
- Der kan være en partiel match: nogle bytes er i store køen, evt tilknyttet forskellige ventende store instruktioner, mens andre bytes skal læses fra cachen

De fleste out-of-order maskiner kan forwarde store data til en ventende load fra flere matchende stores.

Tidligere matchende store men data mangler

Håndteres som et cache miss - wb udskydes til data er blevet skrevet i store-køen

```
                  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
sw  x12,40(x3)      Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc -- -- -- Ca Cb
-                             Qu -- -- -- -- -- -- pk rd st
lw  x12,40(x3)      Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc -- -- wb Ca Cb
```

Normalt ville load instruktionen have lave "wb" i cycle 15, men på det tidspunkt er der ikke nogen data knyttet til den tidligere store instruktion.

I stedet kommer der en senere writeback i cycle 17, efter store-køen har modtaget data.

Tidligere store til ukendt adresse

Muligheder:

- Konservativt: load afventer store adresse.
- aggressivt: ignorer mulig afhængighed, opsaml load data fra cache og andre stores
- typisk: brug en forudsiger (alias-forudsiger) - valider forudsigelse senere

I vores afviklingsplot vælger vi den konservative mulighed. Vi forsinker blot "wb" for load instruktionen til efter alle tidligere store instruktioner har deres addresser i store-køen.

```
                  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
sw  x12,40(x3)      Fa Fb Fc De Fu Al Rn Qu -- -- -- pk rd ag ma mb mc Ca Cb
-                             Qu -- -- -- pk rd st
lw  x12,40(x3)      Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc -- -- -- wb Ca Cb
```

Forkert planlægning

Vi har snydt. Betragt igen en L1 hit, nu med en afhængig instruktion bagefter:

```
        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
L1 hit:   Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc wb Ca Cb
afh insn: Fa Fb Fc De Fu Al Rn Qu -- -- -- -- pk rd ex wb Ca Cb
```

Den afhængige instruktion udvælges i cyklus 12. Vi ved først i cyklus 13 (sidst i "mc")at vi har et cache miss.

Det her er hvad vi skriver i vores afviklingsplot:

```
L2 hit:   Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc -- -- -- -- -- -- -- -- -- wb Ca Cb
afh insn: Fa Fb Fc De Fu Al Rn Qu -- -- -- -- -- -- -- -- -- -- -- -- -- pk rd ex wb Ca Cb
```

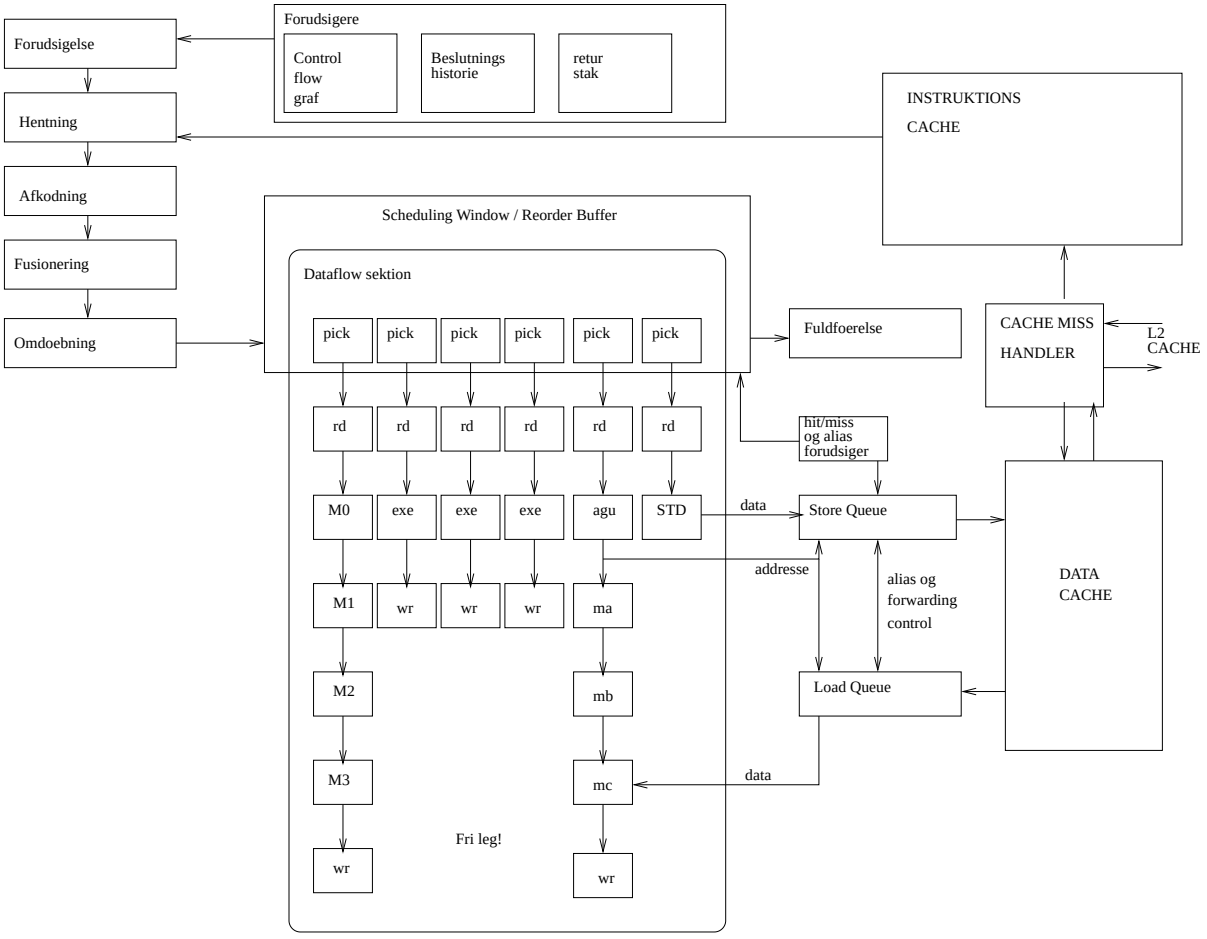
Det her er tættere på virkeligheden:

```
L2 hit:   Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc -- -- -- -- -- -- -- -- -- wb Ca Cb
afh insn: Fa Fb Fc De Fu Al Rn Qu -- -- -- -- pk rd !- -- -- -- -- -- -- pk rd ex wb Ca Cb
```

hvor "!" indikerer at instruktionen blev annulleret.

Vi vælger at se bort fra forkert planlægning.

OOO - Mikroarkitektur - overview



Forudsigelse af programforløb er meget vigtigt for ydeevnen

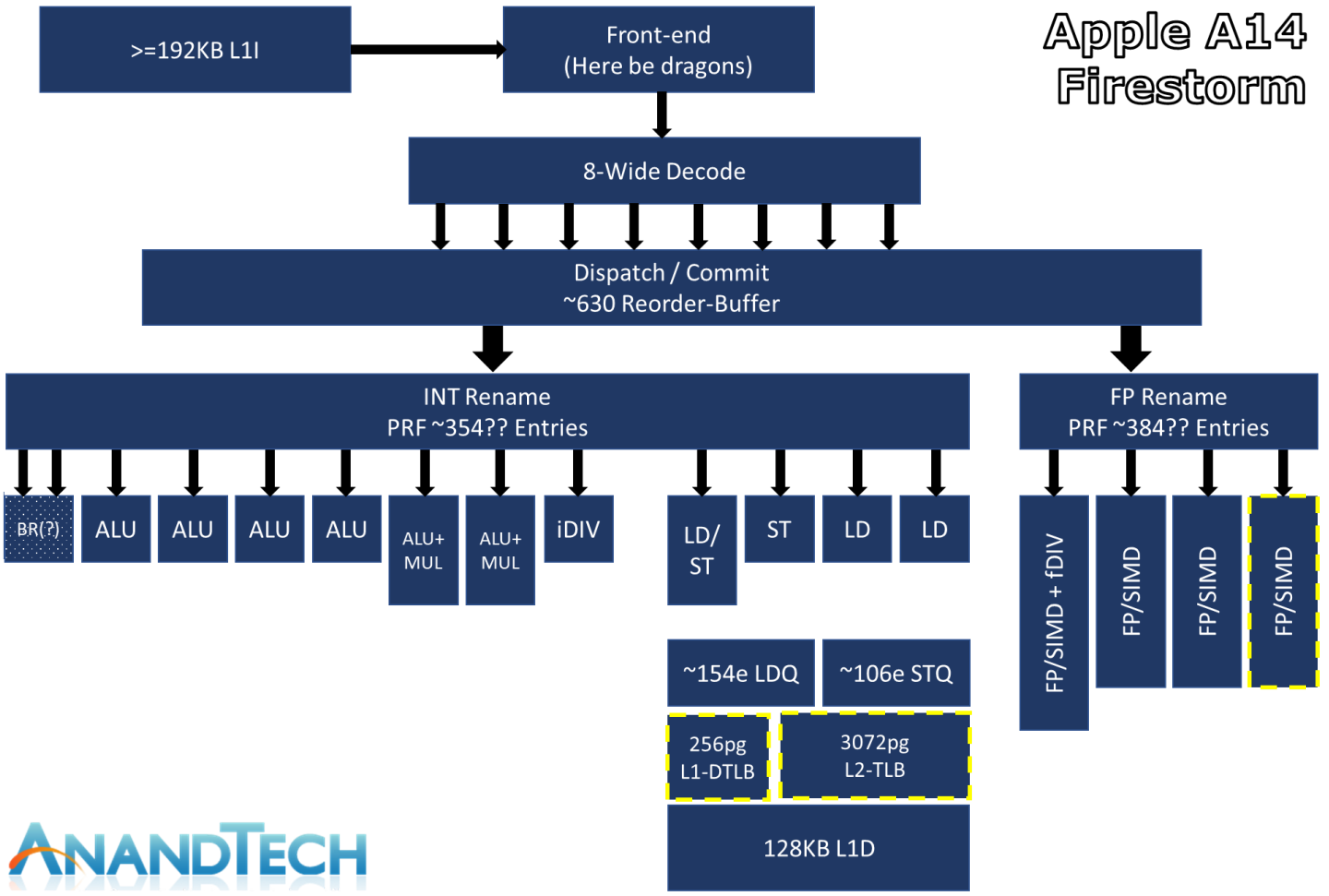
En out-of-order maskine kan arbejde på flere hundrede instruktioner ad gangen. Kvaliteten af forudsigelsen af programforløbet er absolut afgørende for at kunne hente så mange instruktioner hurtigt.

Derfor anvender man korrelerende forudsigere som finder mønstre i programmets historie og bruger disse mønstre til forudsigelse. Den gshare-forudsiger jeg introducerede i en tidligere forelæsning er ikke god nok til en stor maskine.

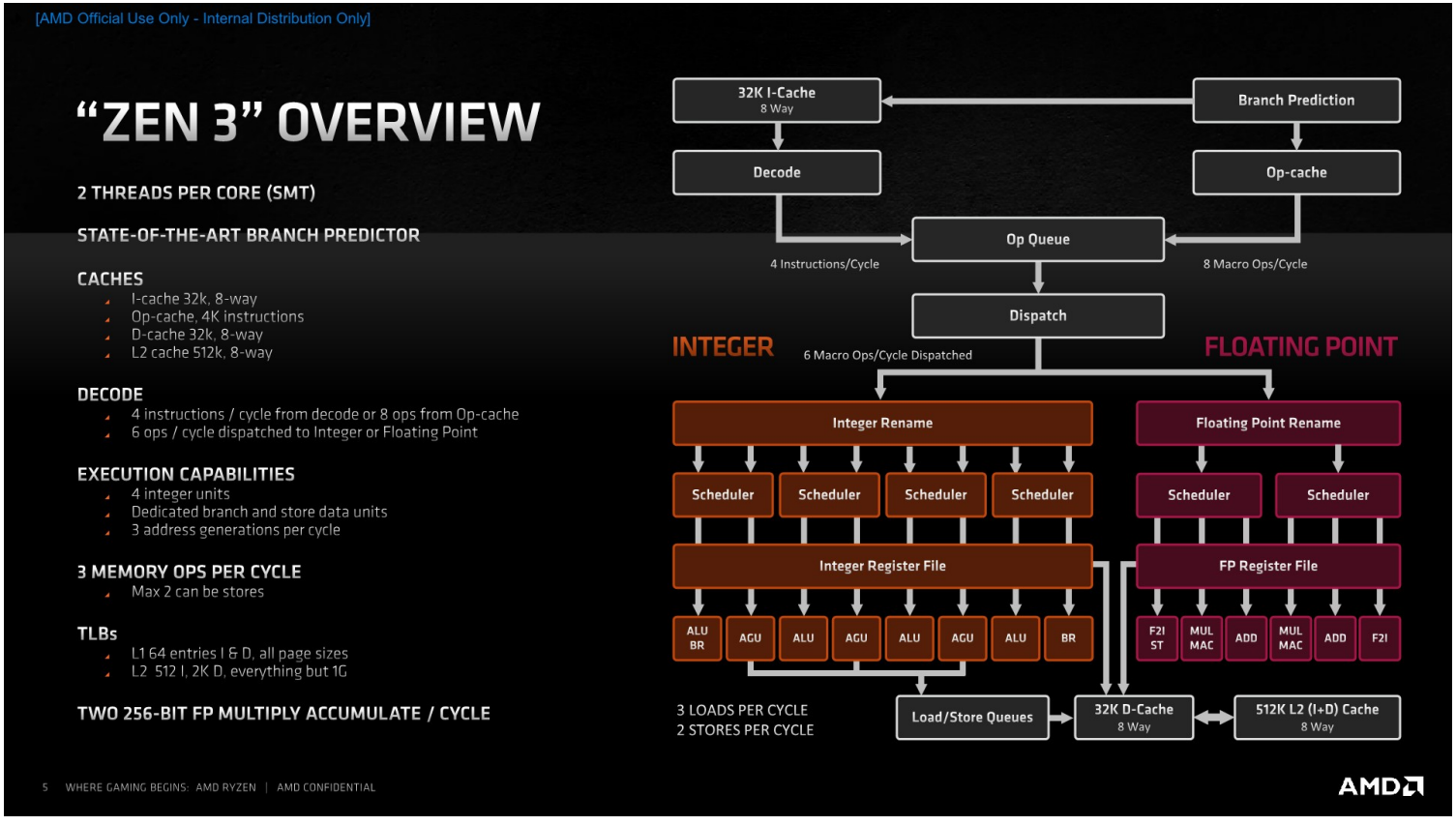
Der er foreslået forudsigere som er realistiske at bygge, og som kan levere flere hundrede instruktioner mellem hver fejl-forudsigelse for et repræsentativt udsnit af programmer.

Vi ved ikke præcis hvilke forudsigere Apple, AMD og Intel bruger. De holder kortene tæt ind til kroppen. Dog ved vi at Zen-3 (fra AMD) bruger en TAGE forudsiger, og hvis du er interesseret i den, så kan du finde den her: https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/

En rigtig ARM (Apple M1) - rekonstrueret



En rigtig x86 (AMD Ryzen 3) - overblik



Opsamling - Out-of-order

Jagten på ydeevne har ledt os til nogle forbløffende komplicerede mikroarkitekturer. Deres design er i væsentlig grad *uafhængig* af det instruktions-sæt de skal udføre. ARM eller x86? Server eller feature-phone? Det er grundlæggende den samme mikroarkitektur der er under motorhjelmen.

Man skulle tro det kunne lade sig gøre at få samme ydeevne med et simplere design. Men den historiske udvikling er brolagt med fejlede forsøg på at opnå samme ydeevne uden brug af dynamisk planlægning af udførelsen. (Google: "itanic")

Det ser ud til at out-of-order maskinerne på en eller anden måde indtager et sweet-spot i computer arkitektur. De er ganske enerådende blandt de højest ydende maskiner. Selv i scenarier man opfatter som relativt sensitive overfor energiforbrug, såsom smartphones, anvender man out-of-order superskalare pipelines.

Kreativ brug af spekulativ udførelse

ind i mellem er der nogen der finder på en virkelig kreativ brug af teknologi. Sådan er det også med spekulativ udførelse. Man kunne jo spørge:

- Instruktioner som bliver fejlforudsagt og derfor bliver annulleret uden at modificere maskinens tilstand, de er jo usynlige. Eller er de?
- Kan man bruge instruktioner som bliver fejforudsagt og annulleret til noget?

De spørgsmål var der flere der stillede sig i løbet af 2017

Svaret åbnede for en hel ny runde af "side-channel attacks"

Paging og page protection (recap)

Vi har en beskyttelsesmekanisme som holder processer adskilt fra hinanden og fra operativsystemet.

- Adresser er virtuelle.
- Hver tilgang til cache indebærer oversættelse fra virtuel til fysisk adresse
- Oversættelse sker via en ATC/TLB (address translation cache/translation lookaside buffer)
- Oversættelse checker lovlighed
- Ulovligt forsøg på adgang sætter en fejl-status på instruktionen

Og særlig for spekulativ udførelse:

- Når en instruktion med fejl-status bliver den ældste og skal fuldføres (Commit)
- Så afbrydes normal udførelse, alle instruktioner smides ud
- Hvorpå processoren skifter til priviligeret "mode" og udfører kode til fejl-håndtering i operativsystemet

Så ulovlig læsning fra lageret kan *ikke* fuldføres og dermed ses udefra. Vel?

Layout af virtuelt addresserum

Maskiner har i mange år kunnet skelne mellem "user space" og "kernel space" adresser. Det har gjort det muligt at have begge dele i samme adresserum. For eksempel ku en 32-bit maskine have 2G afsat til kernen og 2G til en kørende proces. Processen kan ikke tilgå kernen fordi kernen er i kernel space og processen er i user-mode.

Når processen laver et systemkald skifter den til kernel-mode og kan tilgå kernel space. Det er praktisk hvis det sker hurtigt - for når man udfører et systemkald vil man gerne tilgå kernens datastrukturer hurtigt.

I mange år blev det anset som uproblematisk - standard practice. Alle vidste jo at oversættelsen fra virtuel til fysisk addresse indeholdt et check af adgangsrettigheder. Og hvis processen forsøgte at læse en adresse i kernel-space, så ville den trigge en page-fault. Og få et gok i nøden!

(Mis)brug af spekulativ udførelse

Betragt flg programstump

```
char meltdown(char* verboten) {
   typedef struct { /* noget der fylder en cacheblok */ } Block;
   Block side_channel[256];
   // kode der med garanti skubber 'side_channel' ud af cachen
   // kode der træner hopforudsigeren så den forudsiger nedenstående forkert
   if ( (* fejlforudsiges "true", men evaluerer laaaangsomt til "false" *) ) {
      // følgende udføres spekulativt og annulleres derpå
      unsigned char probe = *verboten;  // page fault!!!
      side_channel[probe] = 0;
   }
   // mål på tid det tager at tilgå alle elementer i side_channel
   // hvilket element er nu i cache - det der tog kortest tid at tilgå
   return fastest;
}
```

Kapow! Se https://meltdownattack.com/

Spørgsmål og Svar