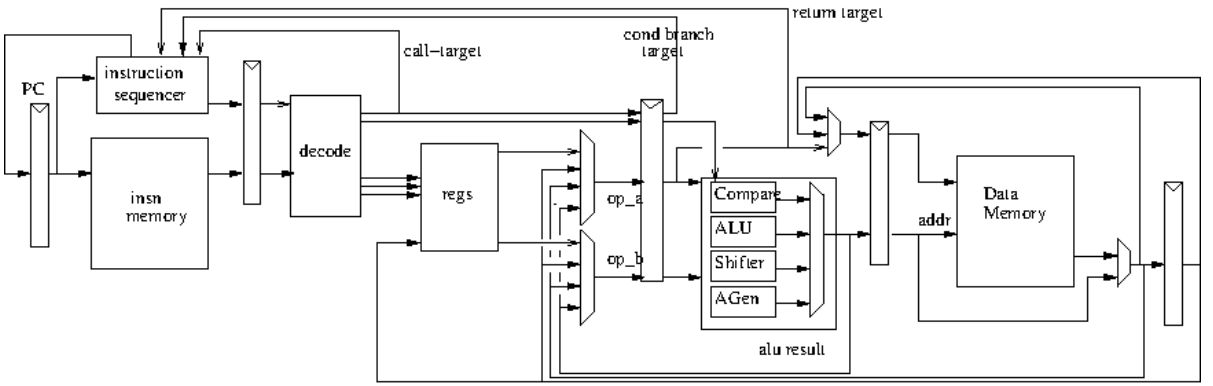
Højtydende mikroarkitektur - Agenda

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

Data afhængigheder i en simpel pipeline

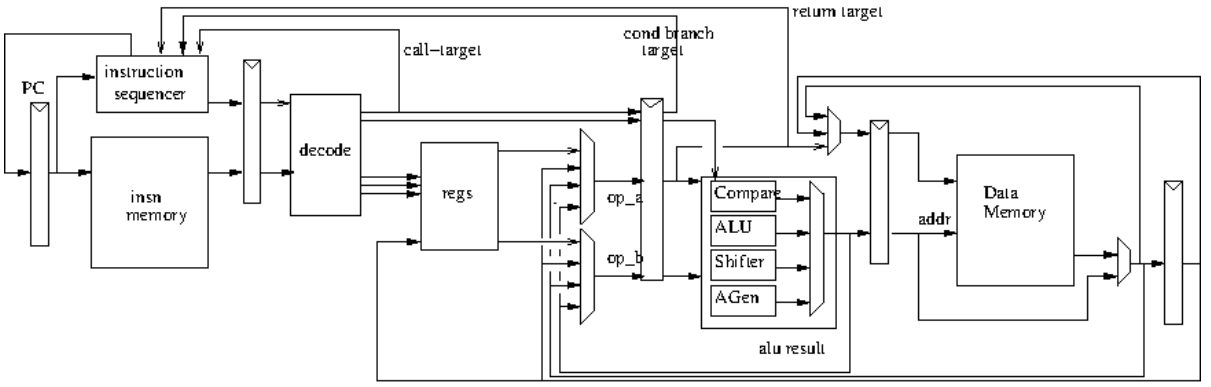
```
insn             FDXMW
movq %r11,%r14    FDXMW
addq %r14,%r17     FDXMW
insn                FDXMW
```



Læsning fra lageret

kaster skygge på 1 instruktion

```
insn                FDXMW
movq 8(%r11),%r14    FDXMW
addq %r14,%r17        FDDXMW
insn                   FFDXMW
```



Læsning fra 3-cycle cache

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

(prikker '.' for ekstra pipeline trin i lager-hierarkiet):

```
insn                F..DXM..W
movq 8(%r11),%r14    F..DXM..W
addq %r14,%r17        F..DDDDXM..W   <-- forsinkes nu 3 cykler
insn                   FF....DXM..W
```

Movq fra lageret kaster nu en skygge på 3 instruktioner.

Hop og 3-cycle cache

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

Her det betingede hop fra før:

```
insn                        F..DXM..W
cbge %r12,%r13,target        F..DXM..W     <-- vi kan afgøre hop i 'X'
<shaddow>                     F..D         <-- og slå de her instruktioner ihjel
<shaddow+1>                    F..
<shaddow+2>                     F.          <-- for ikke at tale om dem her
<shaddow+3>                      F
target: insn                      F..DXM..W  <-- og starte hentning
```

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 næsten halv hastighed!

Det bliver værre

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

```
insn                F..DXM..W
movq 8(%r11),%r14   F..DXM..W
addq %r14,%r17       F..DDDDXM..W   <-- forsinkes 3 cykler
insn                 F.....DXM..W
```

Forsinkelsen er den samme som i den simplere pipeline (3 cykler), men da der ellers kan pumpes dobbelt så mange instruktioner igennem er "skyggen" efter movq-instruktionen dobbelt så lang, 6 instruktioner.

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.

```
                  0123456     -- Bemærkning
movq (r10),r11    FDXMW       --
addq $100,r11     FDDDXW      -- må vente
movq r9,(r14)      FDXM       -- ingen afhængighed, så hvorfor vente?  <---- BEMÆRK!
addq $1,r10        FFDXXW     --
```

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

Men instruktioner er også afhængige af tidligere hop, kald og retur.

Et betinget hop kan forsinke efterfølgende instruktioner.

```
                0123456               -- Bemærkning
movq (r10),r11  F--DXM--W             --
beq r11,r7,L1   F--D----X             -- hop afgøres sent
movq r9,r12             F--DXW        -- 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!

```
                0123456               -- Bemærkning
movq (r10),r11  F--DXM--W             --
beq r11,r7,L1   F--D----X             -- hop afgøres sent og er forudsagt korrekt
movq r9,r12      F--DXMW              -- 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?

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

```
                0123456                -- Bemærkning
movq (r10),r11  F--DXM--W              --
beq r11,r7,L1   F--D----X              -- hop afgøres sent og er forudsagt korrekt
movq r9,r12      F--DXMW               -- 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?

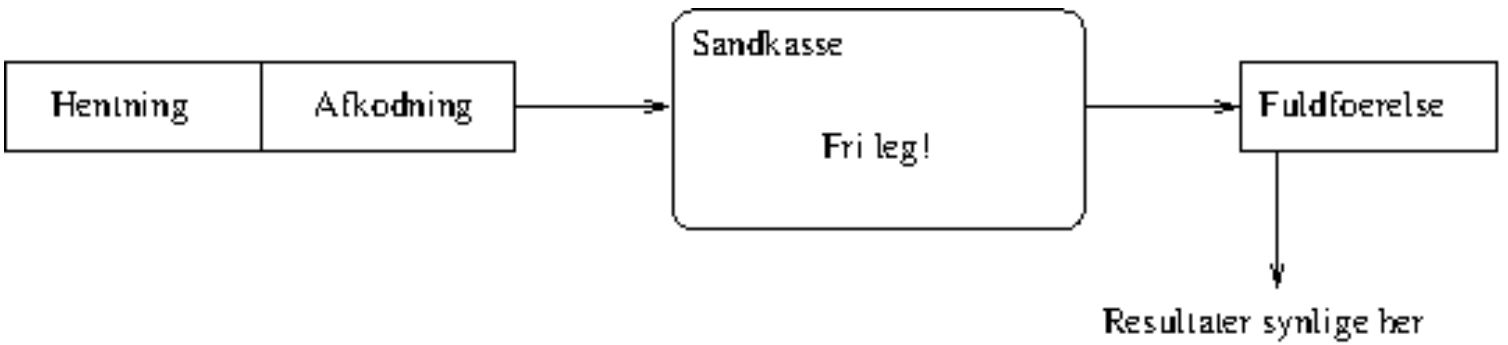
Alle side-effekter må holdes hemmelige, indtil vi faktisk ved om de skal med i udførelsen eller ej!

OOO - Overview

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

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

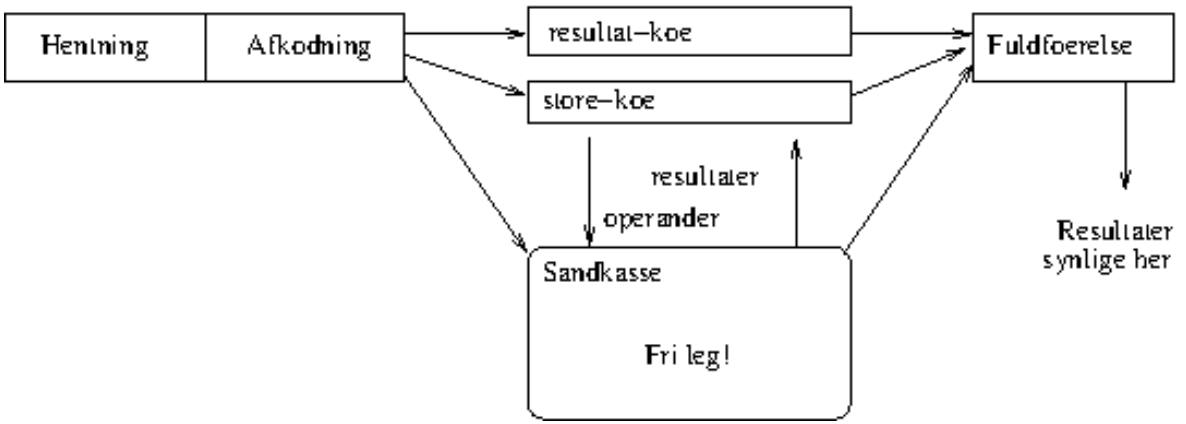
Hvorfor har vi brug for en "sandkasse" ?



Implementation - en sandkasse

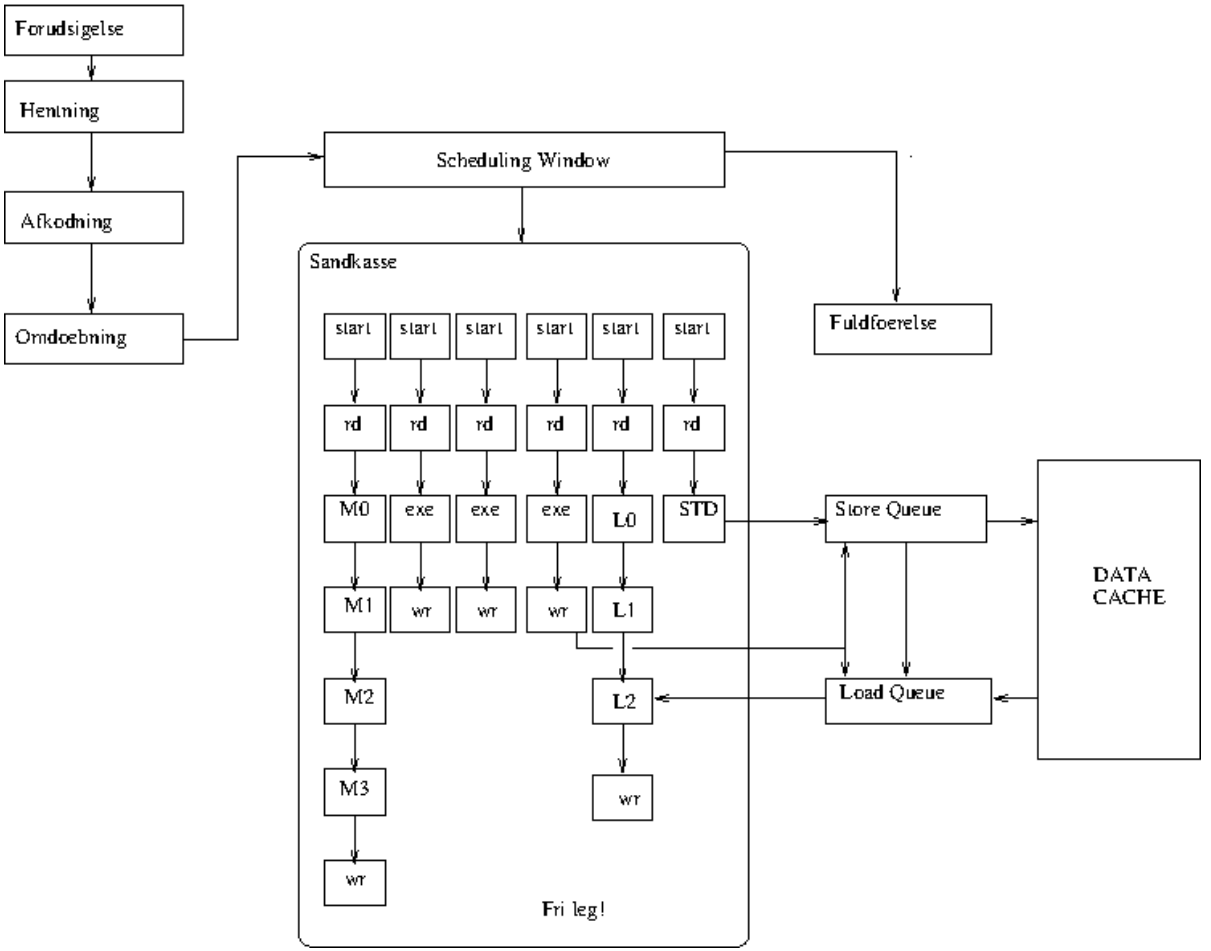
For at kunne lave en sandkasse skal vi isolere effekten af instruktioner indtil vi ved at de skal udføres. Man kunne lagre resultater i to køer:

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.

OOO - Mikroarkitektur - overview



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 3 essentielle komponenter:

- Planlægning af udførelses-rækkefølge (Scheduling)
- Unik ID til enhver instruktion og måske især dens resultat (Renaming)
- Afskærmning fra omverdenen (Skrive kø, søgbar)

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 (vX) registre og 8 fysiske registre (pX).

Vi har fortegnelse over resultater fra længst fuldførte instruktioner i form af et fysisk registernummer for hvert logisk registernummer

```
fuldført:  v0: p0, v1: p1, v2: p2, v3: p3

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

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

```
insn        resultat            -> omskrevet instruktion
ADDQ v0,v1  p4
```

Registeromdøbning (II)

Instruktionens logiske registre skal nu omdøbes. For hver register gennemsøger man listen af tidligere instruktioner efter opdateringer til registeret. Der kan være flere, og så vælges den seneste opdatering. Den har associeret et result-nummer/ et fysisk register nummer. Det er resultatet af omdøbningen.

Hvis der ikke er nogen matchende tidligere skrivninger, så finder man det fysiske registernummer i fortegnelsen over længst fuldførte instruktioner.

```
fuldført:  v0: p0, v1: p1, v2: p2, v3: p3

insn        resultat            -> omskrevet instruktion
ADDQ v0,v1  p4                     ADDQ p0,p1 -> p4
```

I dette tilfælde var der ingen tidligere instruktioner.

Registeromdøbning (III)

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

```
fuldført:  v0: p0, v1: p1, v2: p2, v3: p3

insn        resultat            -> omskrevet instruktion
ADDQ v0,v1  p4                     ADDQ p0,p1 -> p4
SUBQ v2,v1  p5                     SUBQ p2,p4 -> p5
ADDQ v0,v2  p6                     ADDQ p0,p2 -> p6
SUBQ v1,v3  p7                     SUBQ p6,p3 -> p7
```

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

Registeromdøbning (IV)

Når en instruktion har produceret en værdi og er blevet den ældste i listen, så kan den fuldføres ("commit", "retire"). Det gøres på følgende måde:

```
[fuldføres ADDQ v0,v1  -> ADDQ p0,p1 -> p4]

fuldført:  v0: p0, v1: p4, v2: p2, v3: p3

insn        resultat            -> omskrevet instruktion
SUBQ v2,v1  p5                     SUBQ p2,p4 -> p5
ADDQ v0,v2  p6                     ADDQ p0,p2 -> p6
SUBQ v1,v3  p7                     SUBQ p6,p3 -> p7
-           p1
```

Listen "skiftes" opad, så den ældste instruktion forsvinder. Dens resultatnummer og dens logiske destinationsregister bruges til at opdatere fortegnelsen over fuldførte instruktioner. Finesse: det forældende fysiske registernummer for destinations-registeret puttes nederst i køen til senere brug

Spekulativ udførelse - Annulering

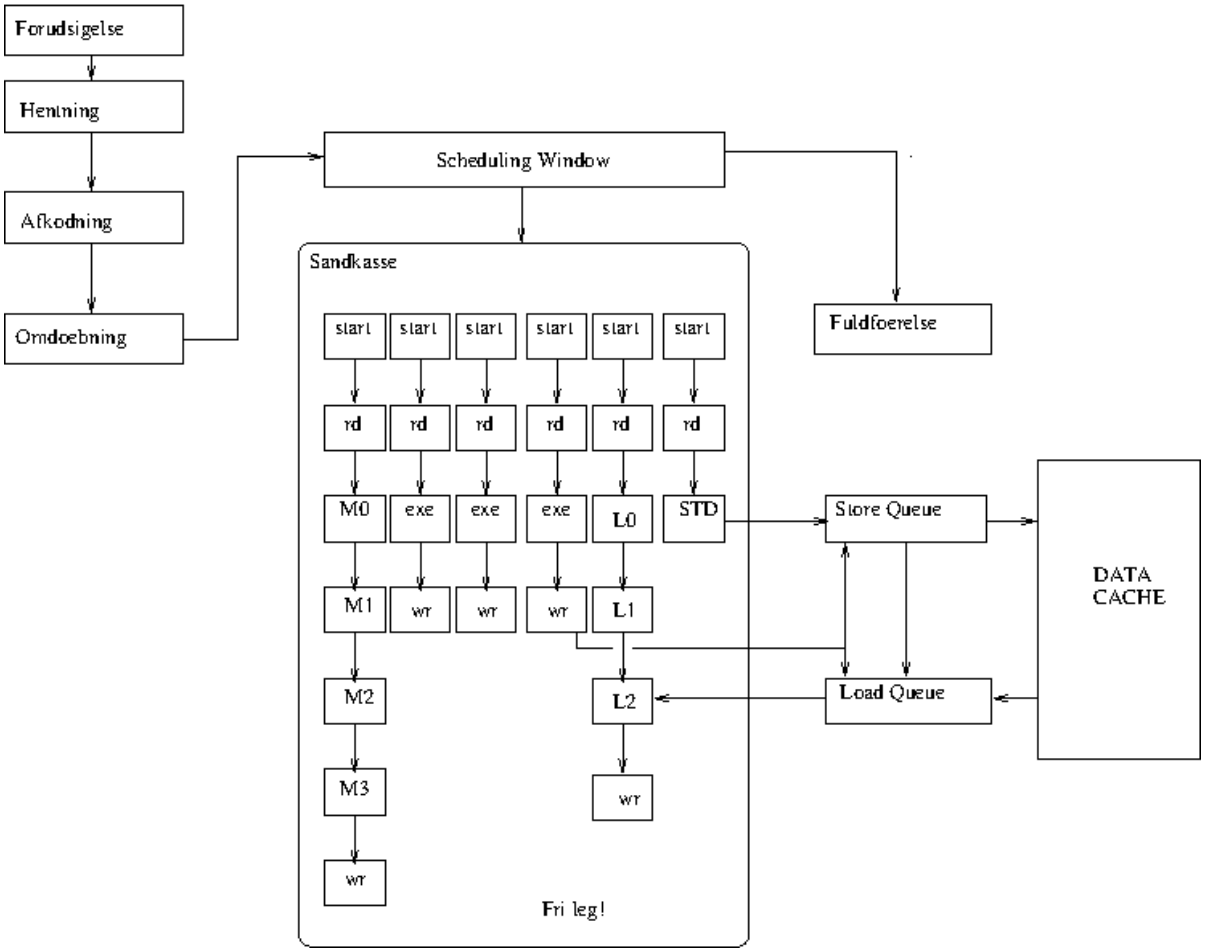
Registeromdøbning er omhyggeligt designet til at understøtte lynhurtig annulering: Hvis et hop er forudsagt forkert, så kan alle efterfølgende (spekulative) register opdateringer annuleres ved at de bliver fjernet fra køen.

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

Da alle instruktioner har deres eget unikke nummer (fysiske register), så kan man bruge en bitvektor til at sende information om hvilke instruktioner der er i live til alle afkroge af maskinen, uden risiko for forveksling.

På den måde kan vi stoppe al ugyldig aktivitet og rydde op i sandkassen på de få cykler der er til rådighed mens de nye instruktioner er på vej gennem hentning og afkodning, så alt er parat når de når til omdøbning.

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 "startet". 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

ADDQ p0,p1 -> p4 [parat]
SUBQ p2,p4 -> p5 [ikke parat]
ADDQ p0,p2 -> p6 [parat]
SUBQ 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

ADDQ p0,p1 -> p4 [startet]
SUBQ p2,p4 -> p5 [parat]
ADDQ p0,p2 -> p6 [parat]
SUBQ p6,p3 -> p7 [ikke parat]
```

Så nu bliver instruktion nummer to også parat.

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

ADDQ p0,p1 -> p4 [parat]
SUBQ p2,p4 -> p5 [ikke parat]
ADDQ p0,p2 -> p6 [parat]
SUBQ p6,p3 -> p7 [ikke parat]
```

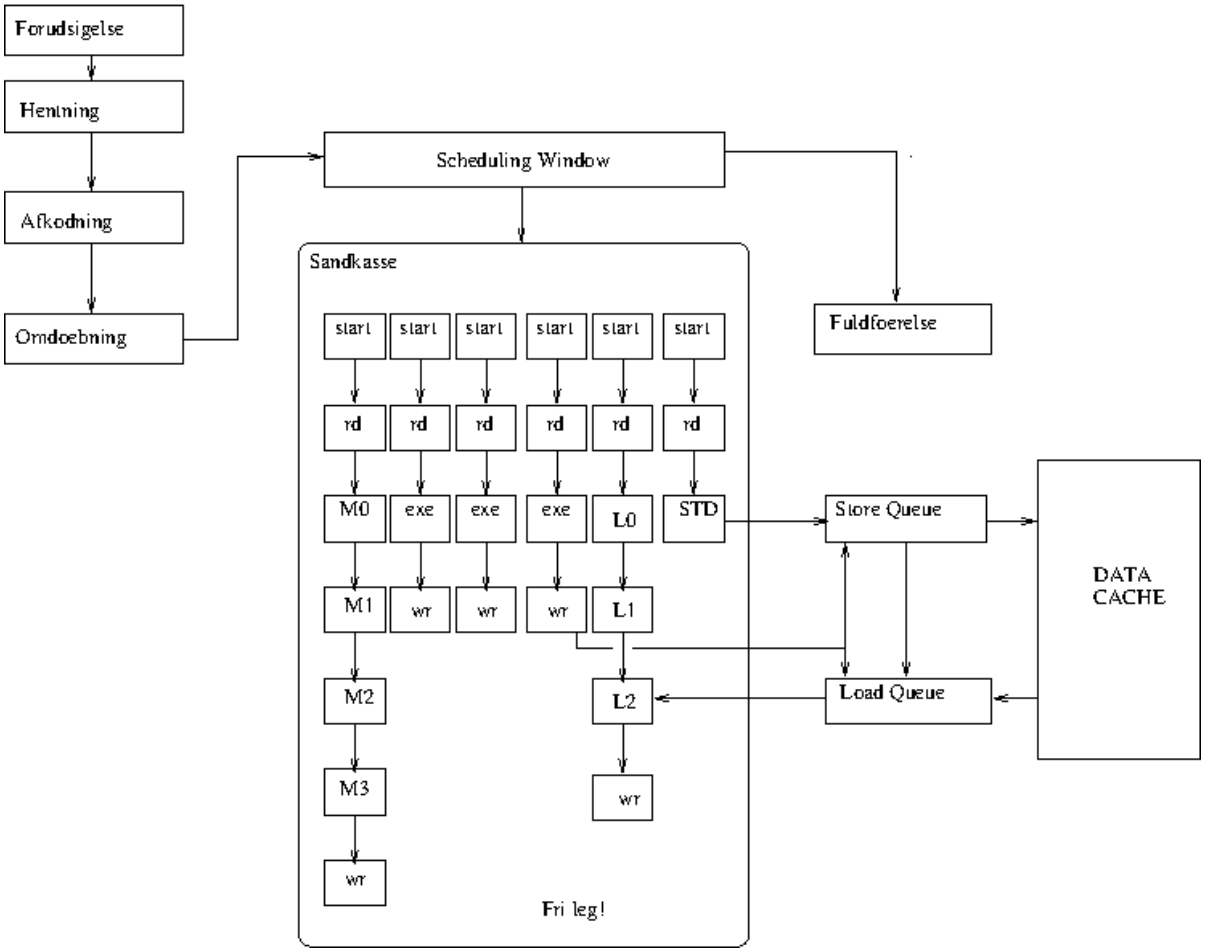
Og planlæggeren vælger så både den første og den tredje.

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

ADDQ p0,p1 -> p4 [startet]
SUBQ p2,p4 -> p5 [parat]
ADDQ p0,p2 -> p6 [startet]
SUBQ p6,p3 -> p7 [parat]
```

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

OOO - Mikroarkitektur - overview



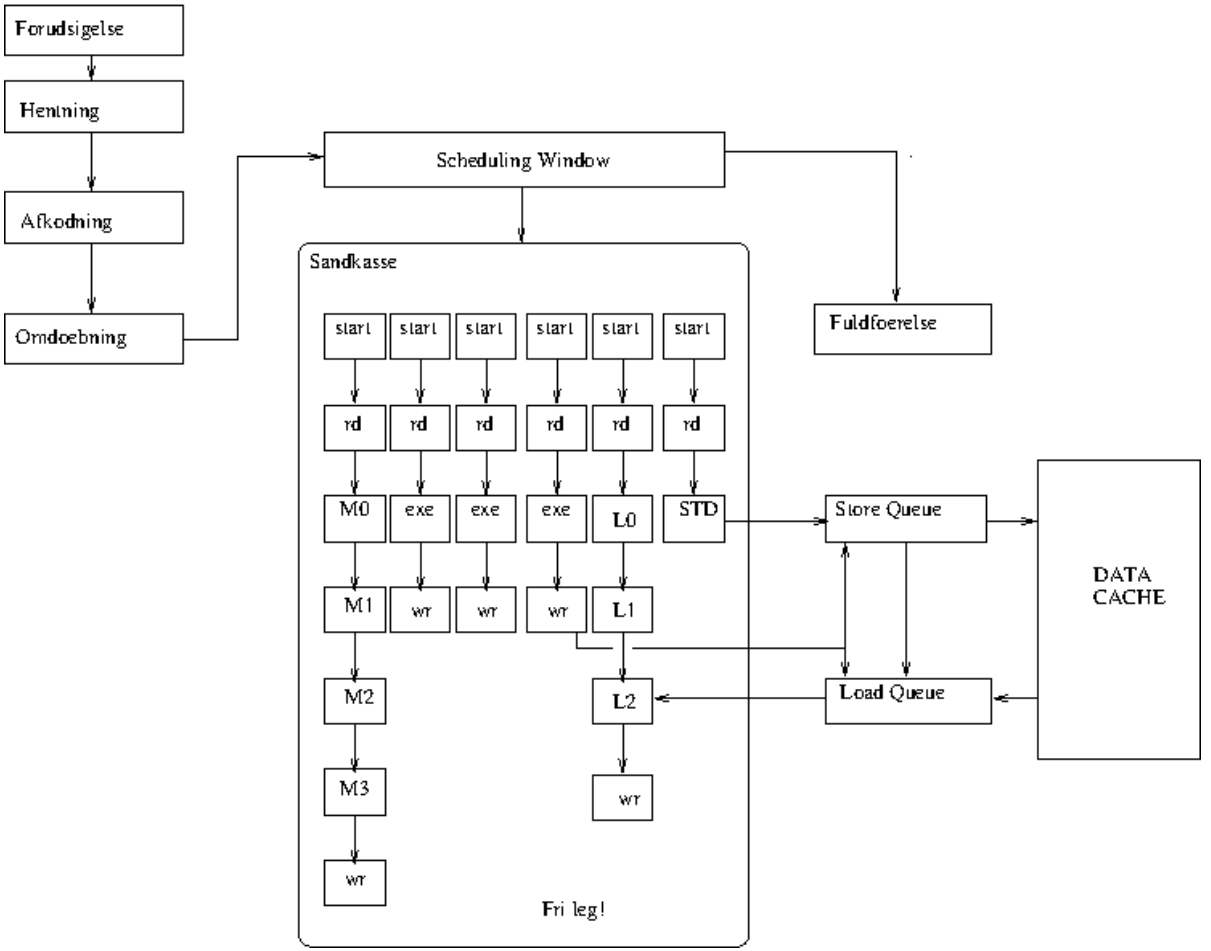
Tilgang til lageret

Skrivninger til lageret skal blive i sandkassen i et stykke tid. Derfor tilføjes en store-kø. Det har betydninger for læsninger fra lageret:

- 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, hvis der er fuldt match, men venter med at fuldføre loads der har partielt match indtil alle relevante store instruktioner er fuldført, har forladt store-køen og har opdateret cachen

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.

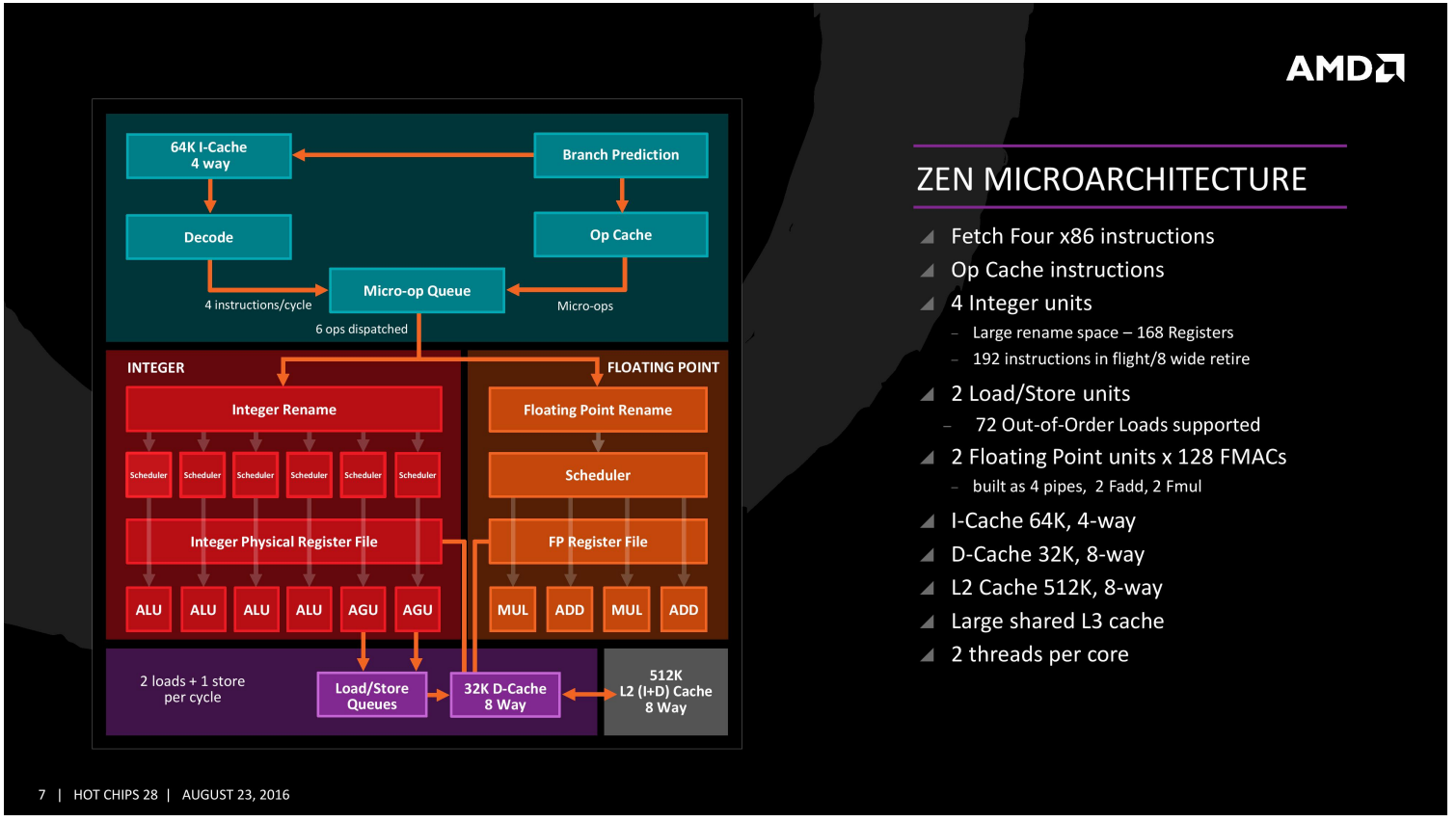
Derfor anvender man korrelerende forudsigere som finder mønstre i programmets historie og bruger disse mønstre til forudsigelse. Den gshare-forudsiger jeg introducerede i en tidligere forelæsning er ikke god nok til en stor maskine.

Der er foreslået forudsigere som er realistiske at bygge, og som kan levere 200-500 instruktioner mellem hver fejl-forudsigelse for et repræsentativt udsnit af programmer.

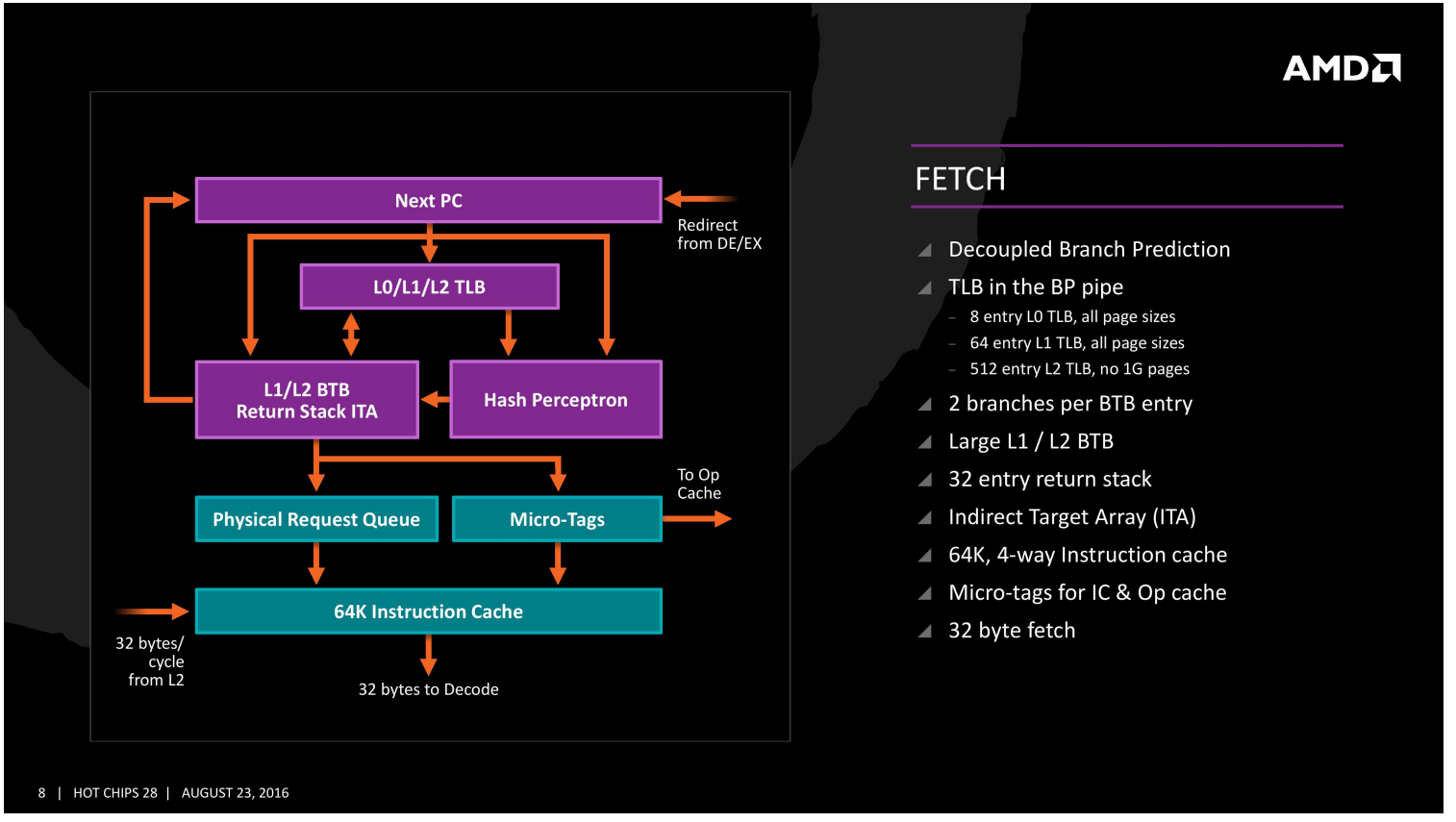
Vi ved ikke præcis hvilke forudsigere AMD og Intel bruger. De holder kortene tæt ind til kroppen.

Interesseret? https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/

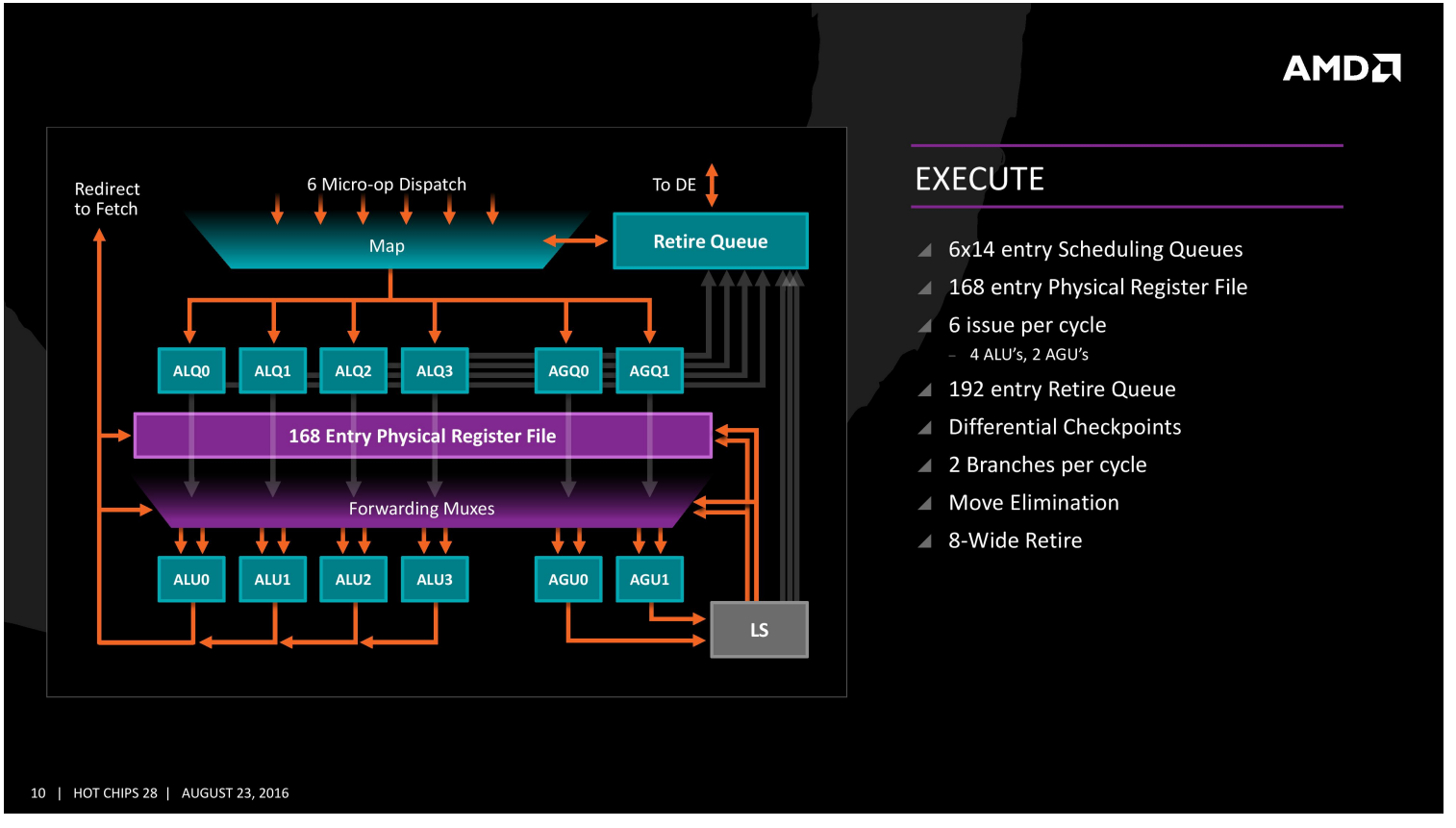
En rigtig x86 (AMD Ryzen) - overblik



En rigtig x86 (AMD Ryzen) - hentning



En rigtig x86 (AMD Ryzen) - x86 sandkasse



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

(Mis)brug af spekulativ udførelse

Betragt flg programstump

```
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 (betingelse der fejlforudsiges som "true", men evaluerer laaaangsomt til "false") {
   << følgende udføres spekulativt og annulleres derpå >>
   unsigned char probe = * (pointer der peger et forbudt sted hen);  // verboten!
   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 >>
```

Kapow! Der var sågar nogen der tænkte det var en del af et komplot!!

Se https://meltdownattack.com/

Opsamling - Performance!

Mikroarkitektur i Balance:

- Mindre clock periode -> generelt hurtigere afvikling, men....
- Mindre clock periode -> større latenstid for nogle instruktioner -> lavere ILP
- Mindre clock periode -> højere clk frekvens -> højere strømforbrug

Køretid = Cycles/Insn *Insn/Prog* ClockPeriode

- Insn/Prog: Compileren (programmørens) domæne
- ClockPeriod: CMOS Implementation, Længde af Signalveje
- Cycles/Insn:
  - Udnyttelse af reference lokalitet -> Caching.
  - Udnyttelse af ILP -> Superscalar. Out-of-order

Betydning for software (udviklere) ?

Betydning for software udviklere

Fokuser på tre (evt fire ting):

- Reference-lokalitet.
- Reference-lokalitet…
- Nævnte jeg reference-lokalitet?
- Vær sød ved din hopforudsiger :-)

Det sidste er rigtig vanskeligt. Her er et eksempel (beklager: reklame)
https://academy.realm.io/posts/how-we-beat-cpp-stl-binary-search/

Disclaimer: Jeg arbejder for Realm og var involveret i ovenstående arbejde.

Ofte er det fint kun at tage hensyn til de første tre.

Spørgsmål og Svar

Termer

Jeg har forsøgt at forsimple præsentationen ved at bruge nogle mere intuitive termer. Men det gør det vanskeligere at søge alternative præsentationer af emnet, så her er nogle oversættelser:

- Register-omdøbning: "Register renaming"
- Fri-liste: Ikke umiddelbart nogen term for det begreb - men det er en del af "sandkassen"
- Sandkasse: "Reorder buffer", "scheduling window"
- Planlægnings-kredsløb: "Scheduler", "Scheduling queue", "Reservation station"
- For Intels Itanium, bagvedliggende forskning: kombiner "EPIC" og "computer architecture"
- Wikipedias artikel https://en.wikipedia.org/wiki/Superscalar_processor er OK.
- Wikipedias artikel om Out-of-order er mere forvirrende.

De vigtige elementer i afviklingsplot

Følgende elementer er afgørende

- "Branch mispredict penalty". Den mindste afstand fra F til X.
- F-fasen er altid i programrækkefølge, og hvis et hop er forudsagt forkert, så sker den (for efterfølgende instruktioner) efter at hoppet afgøres.
- Krav om at bestemte faser skal være i programrækkefølge:
  - I inorder maskiner kan instruktionernes X-faser ikke overhale hinanden
  - I out-of-order maskiner skal hop stadig afgøres i rækkefølge (*)
  - I out-of-order maskiner skal adresser stadig beregnes i rækkefølge (*)
- Korrekt latenstid for udførelses faserne (fra X til W) (bemærk: afhænger af instruktionstypen, og af cache hit/miss for læsning fra lageret)
- Data-afhængigheder overholdes. X-fasen for en instruktion kan først begynde når de nødvendige operander er produceret.
- Resource-begrænsinger overholdes (f.eks. max 1 cache adgang pr cycle)

Andre elementer er ikke så vigtige. Der er mange forskellige aktiviteter i forenden af en out-of-order pipeline - om vi lige placerer D-fasen det rette sted betyder mindre.

(*) Der findes maskiner som ikke har disse krav. Men ikke på CompSys.