Oversigt over forelæsningerne i maskinarkitektur

1. En essentiel maskine bygget af nogle passende byggeklodser. Så simpelt som muligt - men ikke simplere. A2.

2. Et deep dive i hvordan klodserne bygges. Masser af detaljer der giver baggrund

3. Pipelining - hvorfor og hvordan? Performance! Mere performance! Hvor langt kan man gå? Hvordan? Mere realisme

4. Avanceret mikroarkitektur. Parallel udførsel af sekventiel kode. Hvordan?

Abstraktionsniveuaer

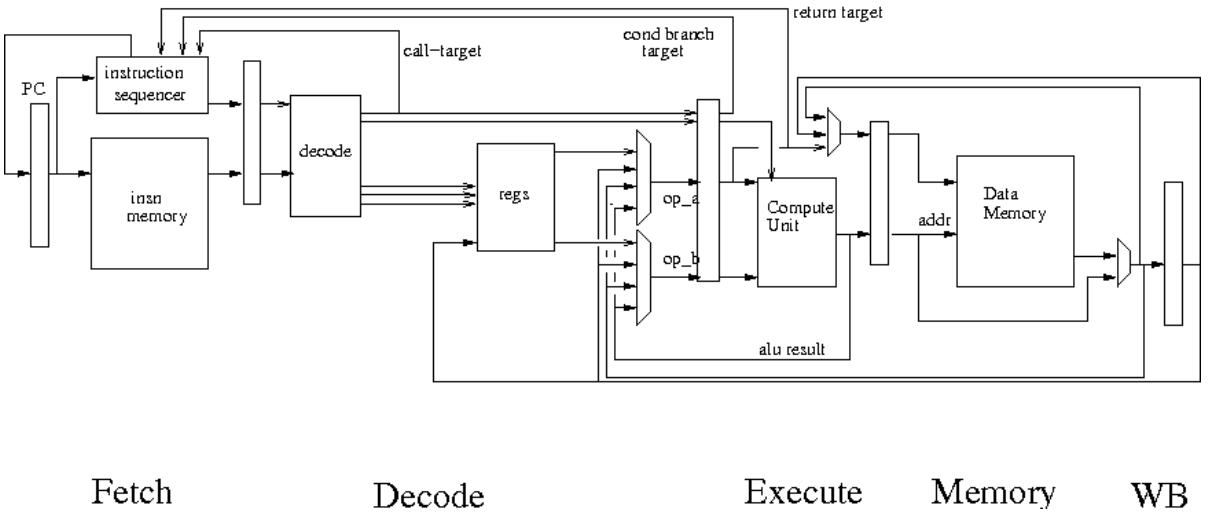
1. Højniveau-programmeringssprog: Erlang, OCaml osv
2. Maskinnære programmeringssprog: C
3. Assembler / Symbolsk Maskinsprog: x86, ARM, MIPS
4. Arkitektur (ISA): Maskinsprog - ordrer indkodet som tal
5. Mikroarkitektur: ting som lager, registre, regneenheder, afkodere og hvordan de forbindes så det bliver en maskine
6. Standard celler: Simple funktioner af få bit (1-4) med et eller to resultater. Lagring af data (flip-flops)
7. Transistorer
8. Fysik. Eller noget der ligner

Højtydende mikroarkitektur - Agenda

1. Recap: Udfordringen
2. Out-of-order execution
3. Implementation
4. Detaljer fra en virkelig maskine

Data afhængigheder i en simpel pipeline

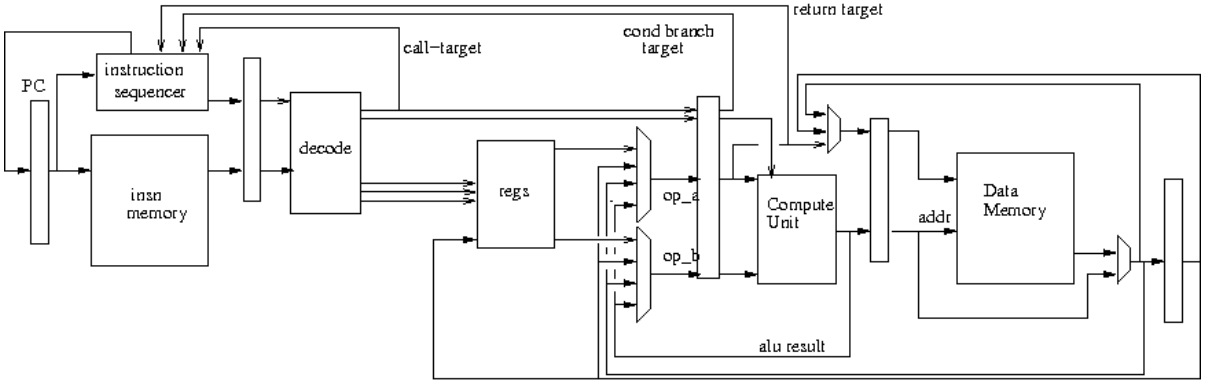
```
insn             FDXMW
movq %r11,%r14    FDXMW
addq %r14,%r17     FDXMW
insn                FDXMW
```



Læsning fra lageret

kaster skygge på 1 instruktion

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



Fetch Decode Execute Memory WB

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.

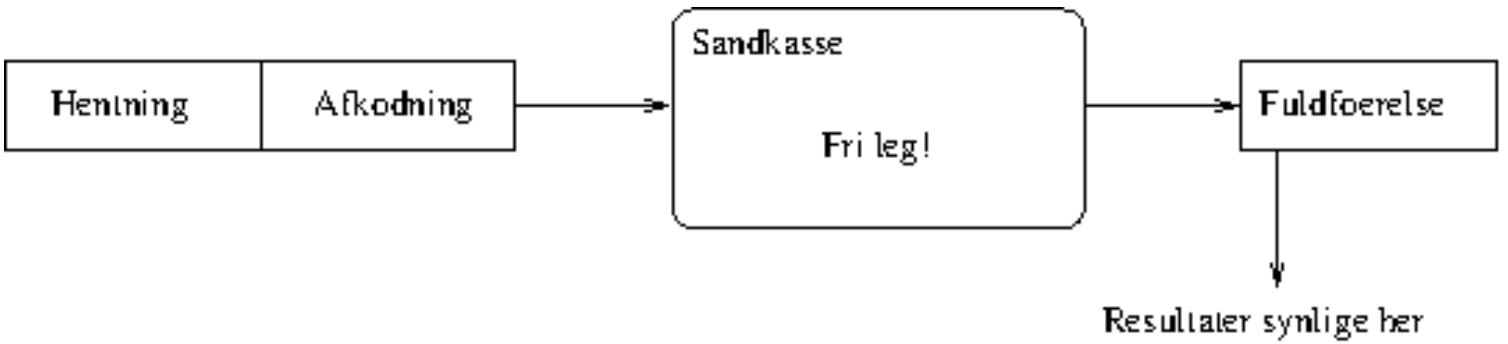
Så hvad gør vi så?

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"
3. Forudsigelse af programforløb gør det muligt at "fylde sandkassen" hurtigt

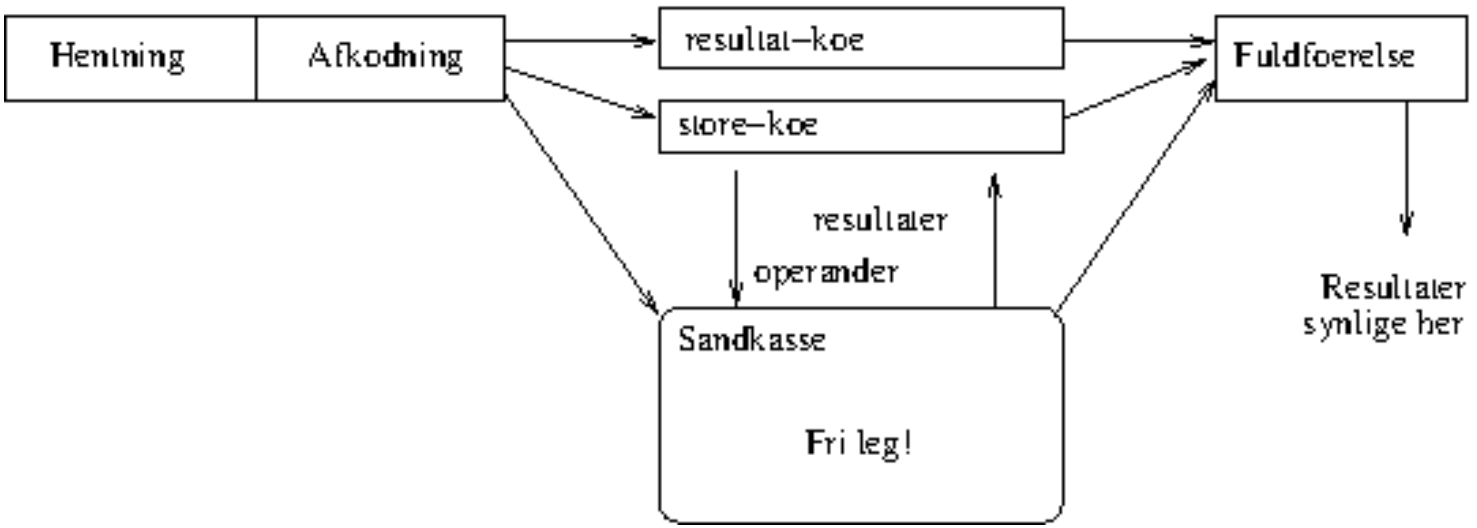
Hvorfor skal vi have en "sandkasse" ?



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 registeret (der kan være mange). På samme måde må læsning fra lageret konsultere 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. De centrale elementer i sandkassen er:

1. Registeromdøbning - giv hvert resultat en unik ID
2. Planlægning af udførelsesrækkefølge, "scheduling"
3. Styring af forwarding
4. Læsning fra lageret

Registeromdøbning

Det er nødvendigt at kunne identificere hvert eneste resultat på en simpel og hurtig måde. Derfor giver vi hvert resultat en unik ID. Denne ID tildeles når en instruktion lukkes ind i sandkassen.

Alle referencer til registre erstattes med den unikke ID for den værdi der sidst er skrevet til det pågældende register - også før værdien faktisk er tilgængelig. Inden i sandkassen er alle referencer til register (også CCR) erstattet med unikke resultat numre.

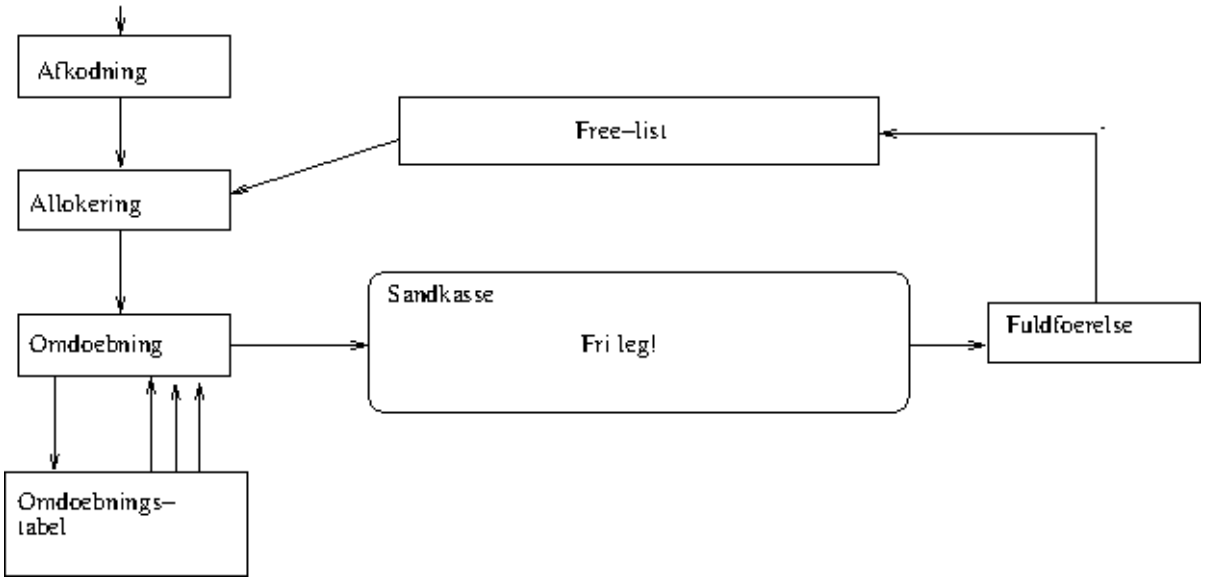
Denne proces kaldes register-omdøbning

```
movq %r8,%r9            movq p101 -> p110
addq $18,%r9            addq $18,p110 -> p111
andq $9000,%r9          andq $9000,p111 -> p112
```

Hver instruktion får et nyt resultat-id (pnnn)

Sådan arbejder registeromdøberen

1. Resultat-ID'er allokeres fra en fri-liste.
2. Registre der skal læses slås op i en omdøbnings-tabel. Tabellen indeholder for hvert register den seneste resultat-ID. Det *tidligere* Resultat-ID for destinations-registeret følges med instruktionen rundt i sandkassen
3. Tabellen opdateres med de nye resultat-ID'er
4. Når instruktionen engang er færdiggjort med succes og alle tidligere instruktioner også er færdiggjort med succes, fuldfører vi instruktionen (også kaldet "commit") ved at tilføje det *tidligere* resultat-ID til fri-listen



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.

For hver instruktion er sammenlignes de indgående resultat-ID'er som instruktionen skal bruge, med resultat-ID'er for de resultater der snart vil blive produceret.

Når en instruktion således har "observeret" alle de resultat-ID'er som den har brug for, bliver den markeret som "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 resultat-ID vil i en senere clock-cyklus blive ført tilbage til planlæggeren for få startet afhængige instruktioner.

Der findes både maskiner med flere små planlægningskredsløb som hver maximalt starter en instruktion per clock, og maskiner med færre store planlægninskredsløb som kan starte flere instruktioner per clock.

Styring af forwarding

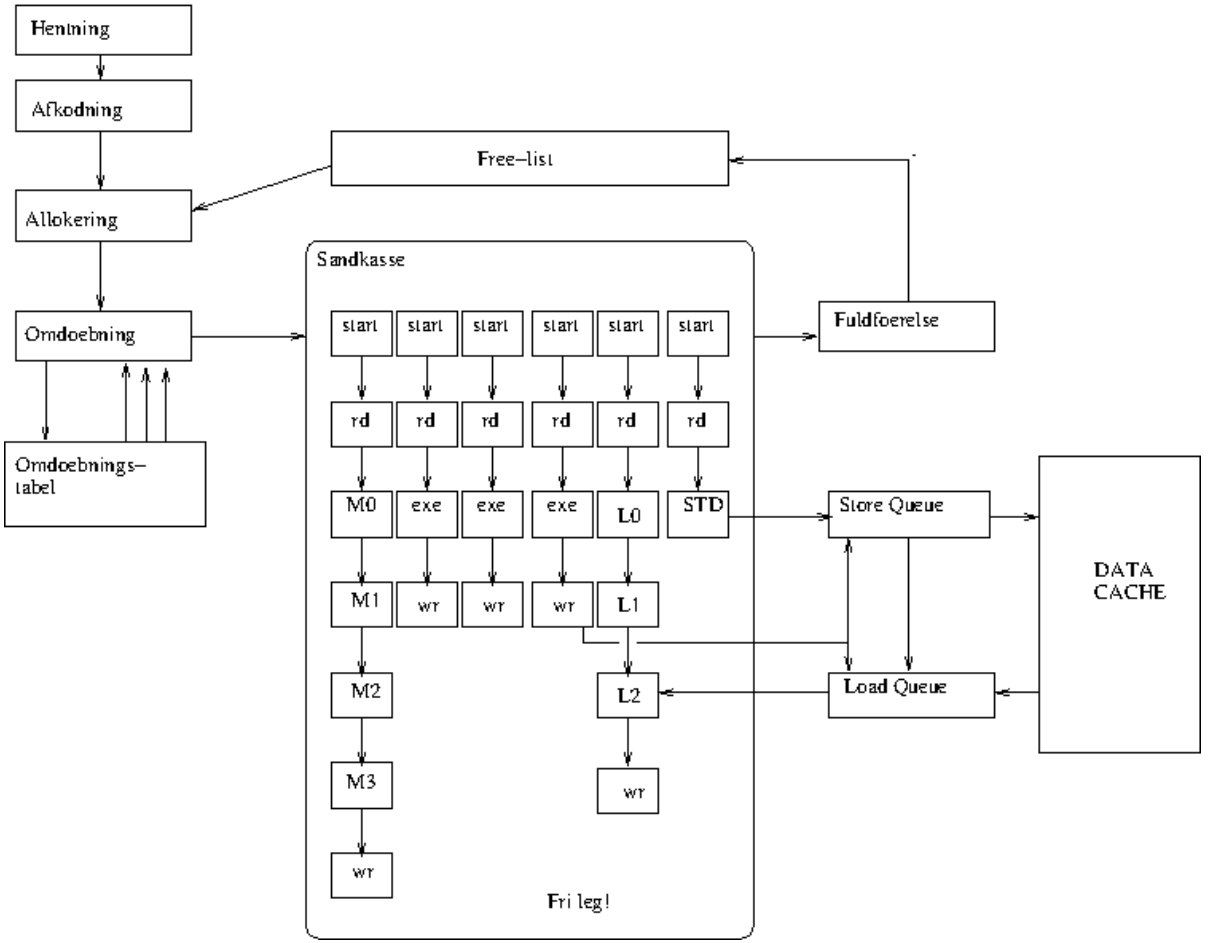
Forwarding fra en instruktion til en afhængig instruktion sker præcis som i almindelige pipelines. Der kan være flere regne-enheder der skal forbindes, men konceptuelt er det faktisk simplere at forwarde i en out-of-order maskine end i en superskalar.

Det skyldes at resultaterne i en out-of-order maskine er identificeret med et unikt resultat-ID. Til sammenligning er resultater i de simplere maskiner identificeret med register numre, som kan være i brug flere steder i pipelinen samtidig. Det kræver en efterfølgende prioritering sådan at en modtager altid får den rette "udgave" af et register.

I en out-of-order maskine kan styresignalerne til forwarding multiplexerne sættes af planlægningskredsløbet og så følge instruktionen ned igennem pipelinen.

Det er aldrig nødvendigt at "bremse" en pipeline inden i sandkassen. Instruktioner starter først når man ved at de kan gennemføres.

Overblik over mikroarkitekturen



Læsning fra lageret

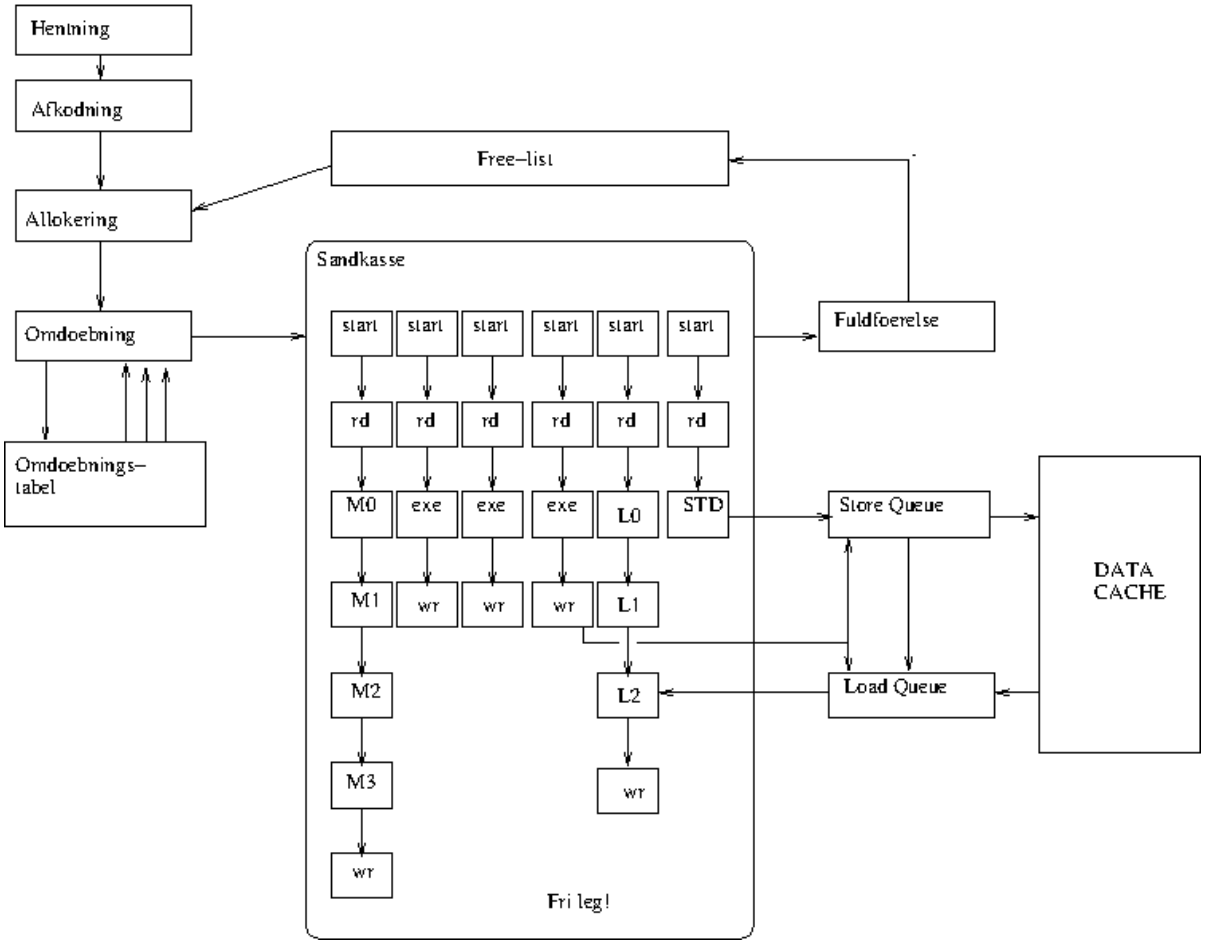
Skrivninger til lageret skal blive i sandkassen. Derfor har vi tilføjet en store-kø. Det har betydninger for læsninger fra lageret:

- 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, og nogle bytes der skal læses fra cachen
- Der kan være en ældre skrivning, hvor adressen endnu ikke er beregnet. Hvad gør man så?

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

I alle maskiner er det nødvendigt at kunne sætte load instruktionerne i en kø, hvor de kan afvente at de får lov til at tilgå cache eller kan opsamle de relevante data fra store-køen.

OOO - Mikroarkitektur - Fordi vi kan!



Bedre forudsigelse af programforløb

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/

Vi har glemt noget!

Håndtering af fejlagtig forudsigelse

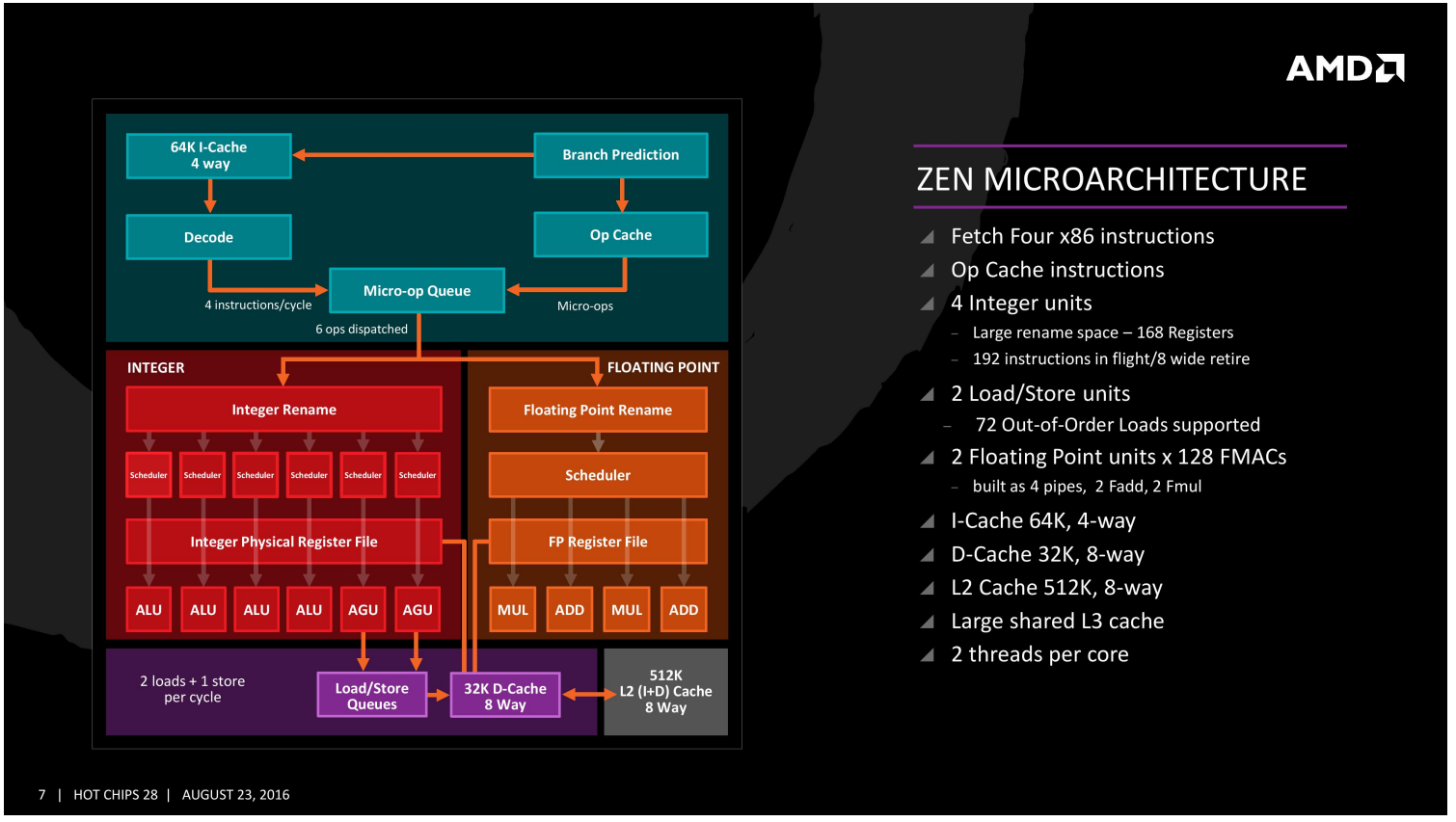
Når et hop afgøres og det viser sig at være forudsagt forkert, vil der i sandkassen være både instruktioner før og instruktioner efter hoppet. Kun de før hoppet skal fuldføres. Hvordan opnår man det?

- Vi vælger at afgøre hop i program rækkefølge
- Vi holder tal på hvor mange uafgjorte hop der er i sandkassen
- Vi tilføjer det tal til hver instruktion når den ankommer.
- Når et hop afgøres positivt "broadcaster" vi det til alle instruktioner, og hver af dem kan formindske deres tæller.
- Når et hop afgøres negativt (forudsagt forkert) "broadcaster" vi også det til alle instruktioner. Alle instruktioener med uafgjorte hop bliver "deaktiveret"
- Forskellige mekanismer frigør løbende resourcer for deaktiverede instruktioner
- Hentning, afkodning osv nulstilles og genstartes fra den rette adresse

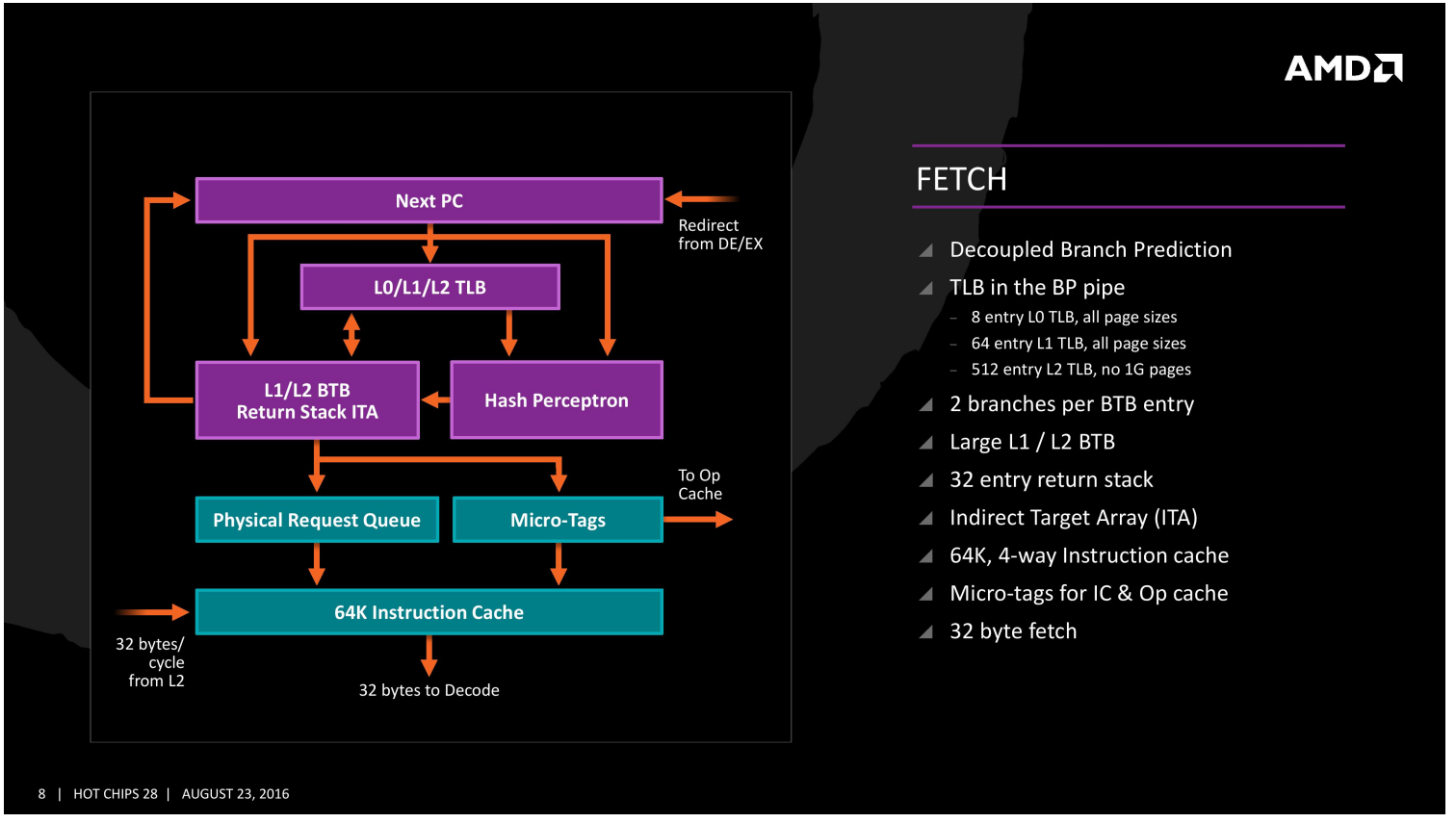
Hvad med omdøbningstabel og friliste?

- Vi retablerer den tilstand de havde da det fejl-forudsagte hop ankom.

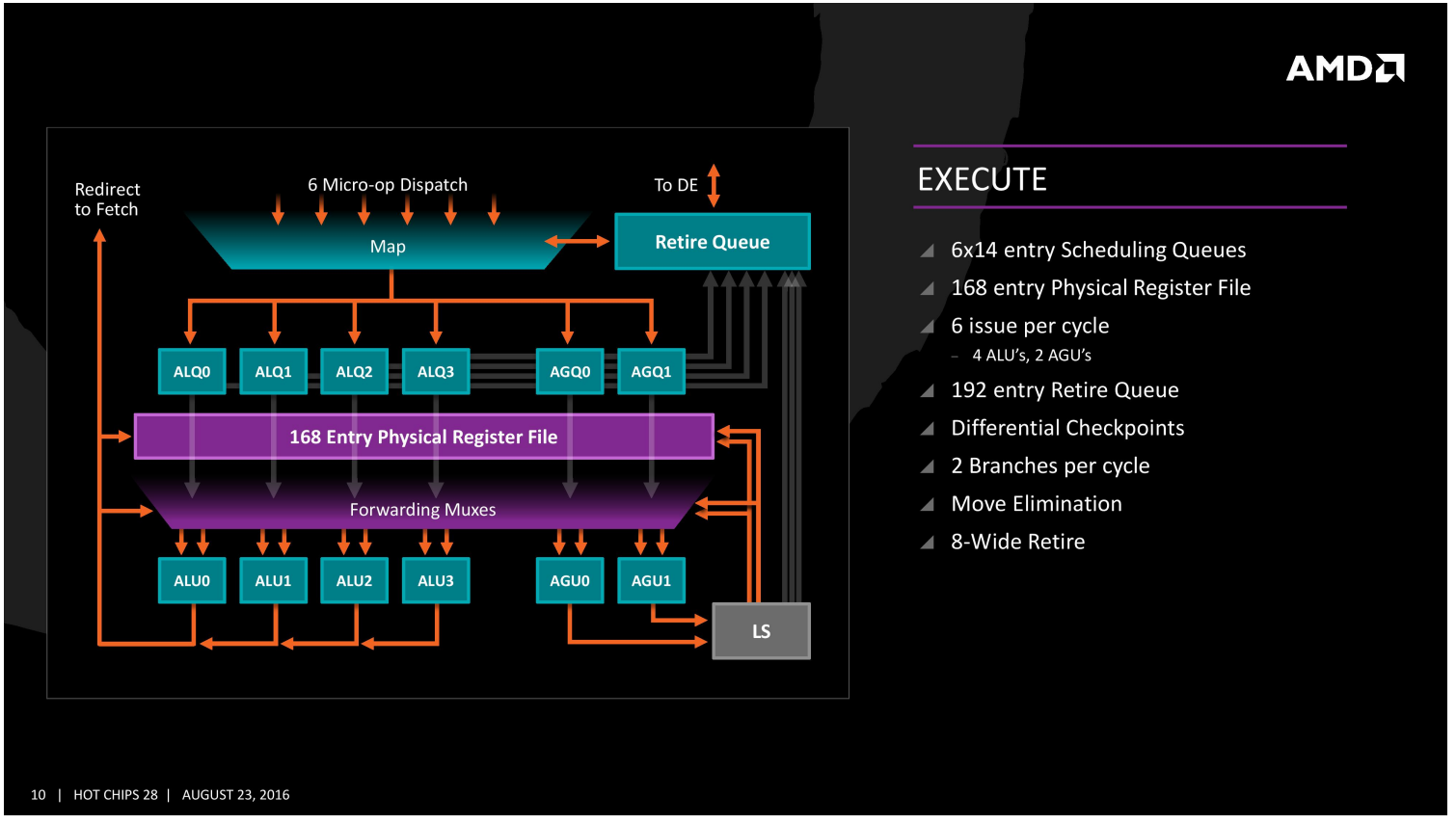
En rigtig x86 (AMD Ryzen) - overblik



En rigtig x86 (AMD Ryzen) - hentning



En rigtig x86 (AMD Ryzen) - x86 sandkasse



Opsamling - big picture

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? Det er grundlæggende den samme struktur 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.

Men ja, en Nokia 3310 kører længere på en opladning :-D

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.