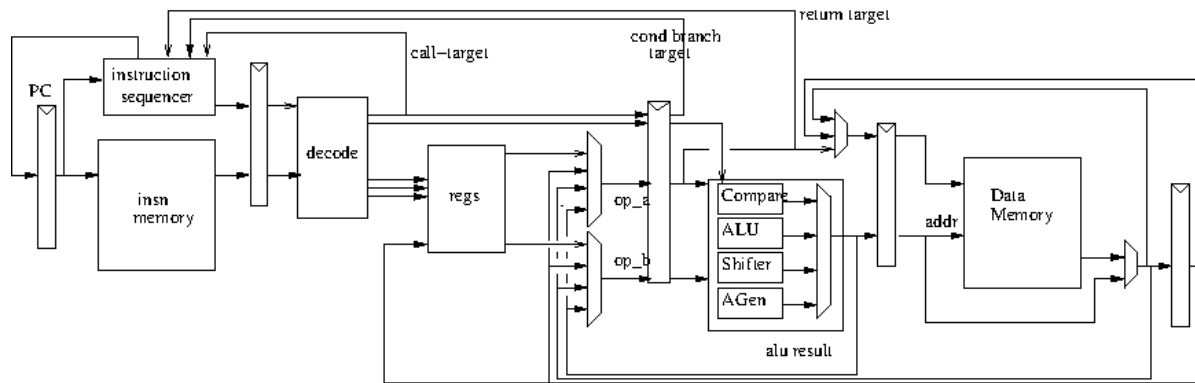


Højtydende mikroarkitektur - Agenda

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
```



Fetch

Decode

Execute

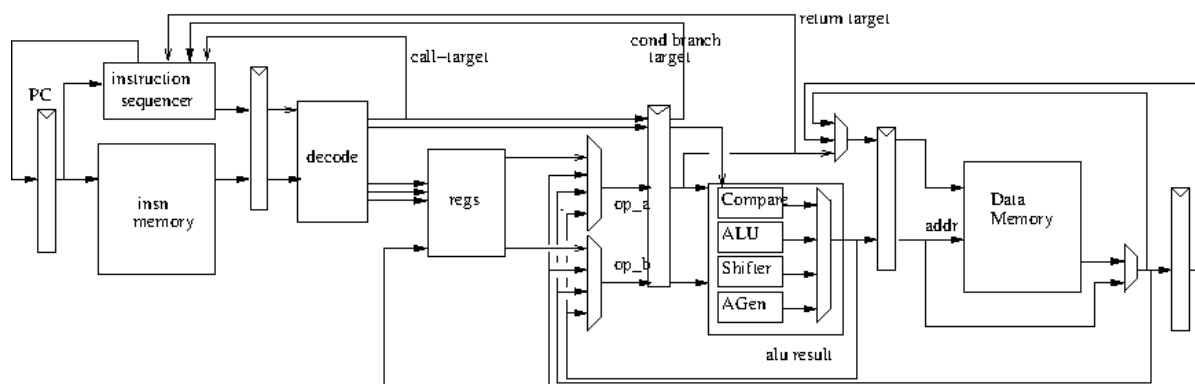
Memory

WB

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'
<shadow>      F..D          <-- og slå de her instruktioner ihjel
<shadow+1>    F..
<shadow+2>    F.           <-- for ikke at tale om dem her
<shadow+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 simple 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	FFDXW	--	

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?

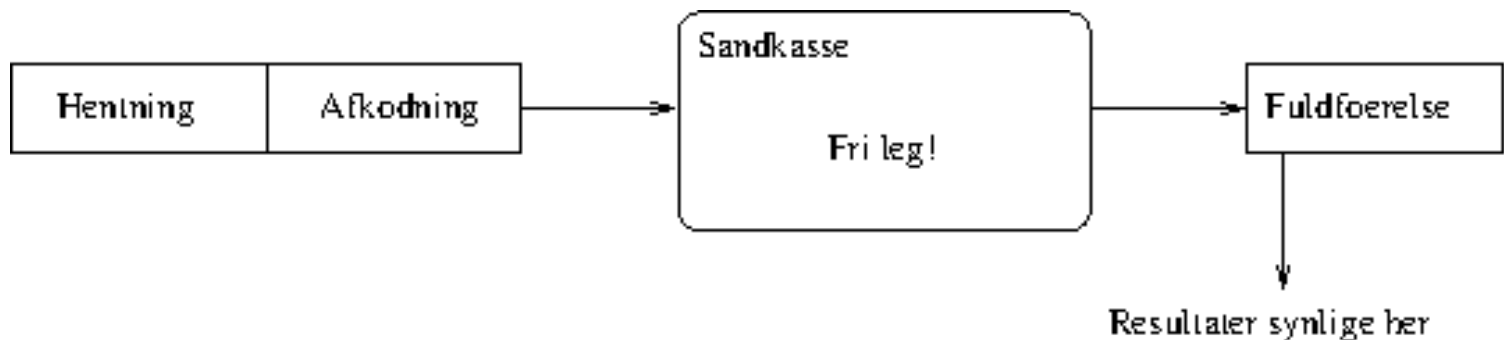
Hvis første instruktion har et miss i datacachen, kan der blive udført mange instruktioner, før det efterfølgende hop afgøres. Hvad kan man tillade sig?

000 - 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.

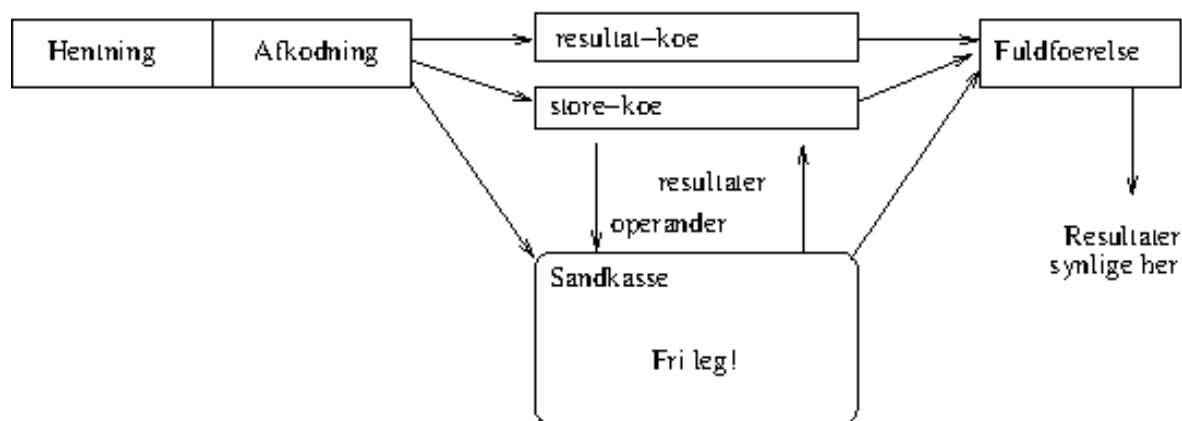
Hvorfor har vi brug for en "sandkasse" ?



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.

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 (Renaming)
- Afskærmning fra omverdenen (Skrive kø, søgbar)

Unik ID til enhver instruktion - og resultat.

For at kunne "schedulere" instruktionerne er det nødvendigt at vi kan identificere dem (og alle operander) unikt. Vi skal bringe instruktionerne på en form, hvor de $(A,B) \rightarrow \text{op} \rightarrow C$, 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.

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.

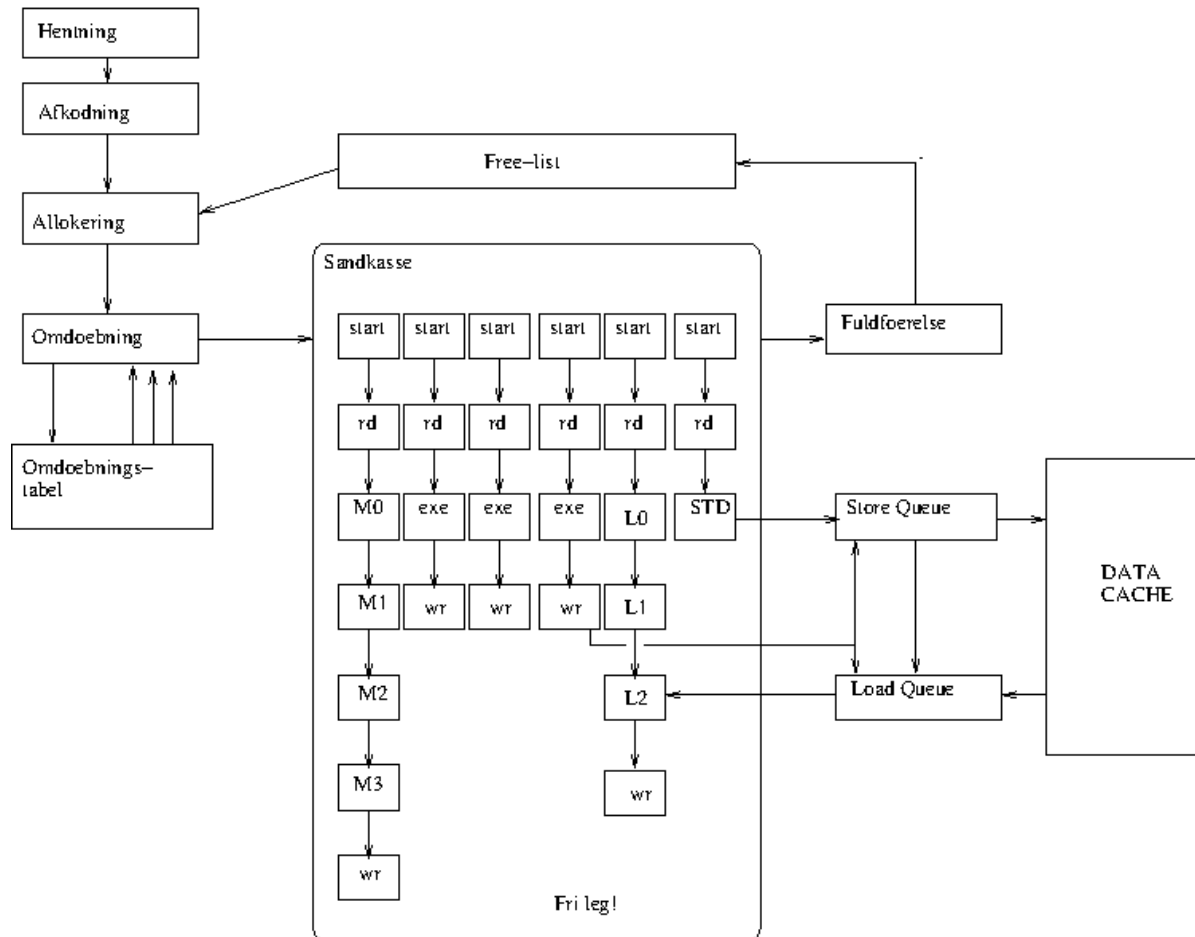
Tilgang til lageret

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

000 - 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.

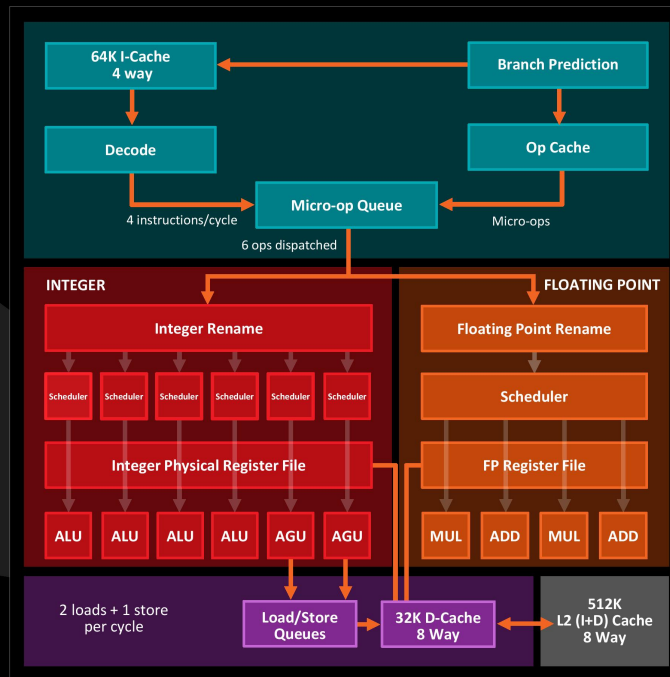
Derfor anvender man korrelerende forudsigere som finder mønstre i programmets historie og bruger disse mønstre til forudsigelse. Den gshare-forudsiger jeg introducerede i en tidligere forelæsning er ikke god nok til en stor maskine.

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/>

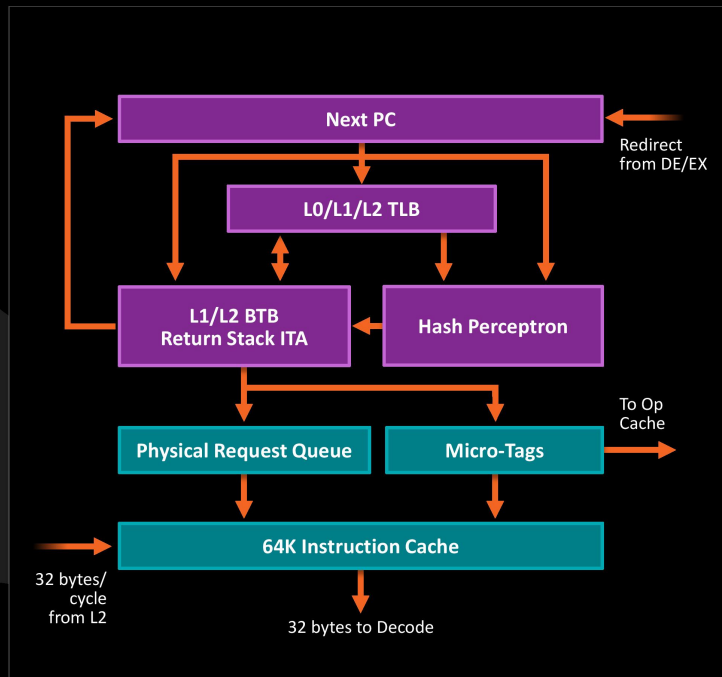
En rigtig x86 (AMD Ryzen) - overblik



ZEN MICROARCHITECTURE

- ▲ Fetch Four x86 instructions
- ▲ Op Cache instructions
- ▲ 4 Integer units
 - Large rename space – 168 Registers
 - 192 instructions in flight/8 wide retire
- ▲ 2 Load/Store units
 - 72 Out-of-Order Loads supported
- ▲ 2 Floating Point units x 128 FMACs
 - built as 4 pipes, 2 Fadd, 2 Fmul
- ▲ I-Cache 64K, 4-way
- ▲ D-Cache 32K, 8-way
- ▲ L2 Cache 512K, 8-way
- ▲ Large shared L3 cache
- ▲ 2 threads per core

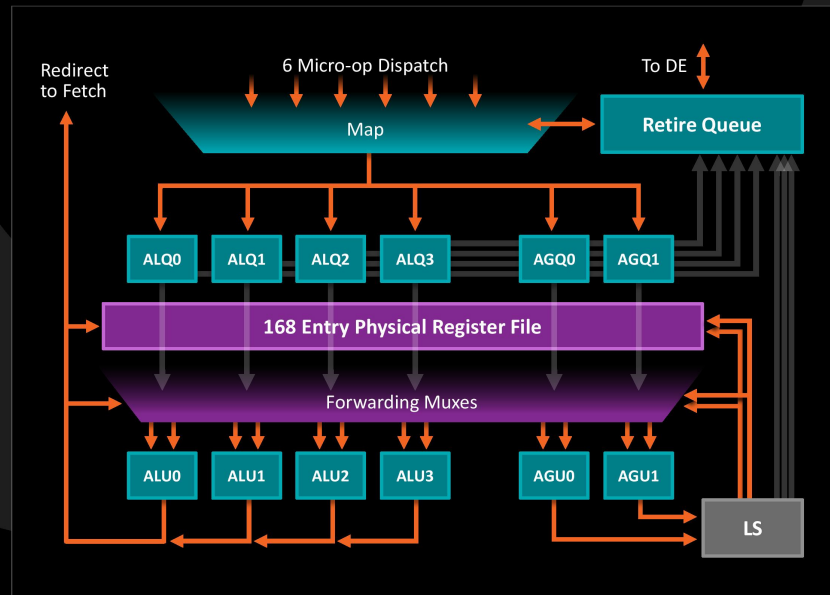
En rigtig x86 (AMD Ryzen) - hentning



FETCH

- ▲ Decoupled Branch Prediction
- ▲ TLB in the BP pipe
 - 8 entry L0 TLB, all page sizes
 - 64 entry L1 TLB, all page sizes
 - 512 entry L2 TLB, no 1G pages
- ▲ 2 branches per BTB entry
- ▲ Large L1 / L2 BTB
- ▲ 32 entry return stack
- ▲ Indirect Target Array (ITA)
- ▲ 64K, 4-way Instruction cache
- ▲ Micro-tags for IC & Op cache
- ▲ 32 byte fetch

En rigtig x86 (AMD Ryzen) - x86 sandkasse



EXECUTE

- ▲ 6x14 entry Scheduling Queues
- ▲ 168 entry Physical Register File
- ▲ 6 issue per cycle
 - 4 ALU's, 2 AGU's
- ▲ 192 entry Retire Queue
- ▲ Differential Checkpoints
- ▲ 2 Branches per cycle
- ▲ Move Elimination
- ▲ 8-Wide Retire

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? Det er grundlæggende den samme struktur der er under motorhjelman

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.

Opsamling - 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 in-order 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

Opsamling - Maskinarkitektur

$Køretid = \frac{Cycles}{Insn} \cdot \frac{Insn}{Prog} \cdot ClockPeriod$

- 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

Balance:

- Mindre clock periode ->
- større latenstid for nogle instruktioner ->
- lavere ILP

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.