Højtydende mikroarkitektur - Agenda

1. Recap: Udfordringen (done, i mandags)

2. ILP. Instruction Level Parallelism (done, i mandags)

3. Out-of-order execution

    - Forudsigelse (done, i mandags)
    - Fusionering af instruktioner (done, i mandags))
    - Omdøbning (done, i mandags)
    - Schedulering <---------- Vi er her!
    - Lagertilgang, Load og Store

4. Lidt detaljer fra en virkelig maskine

5. Opsamling

Hvor nåede vi til i mandags?

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

Out-of-order execution er baseret på 3 elementer:

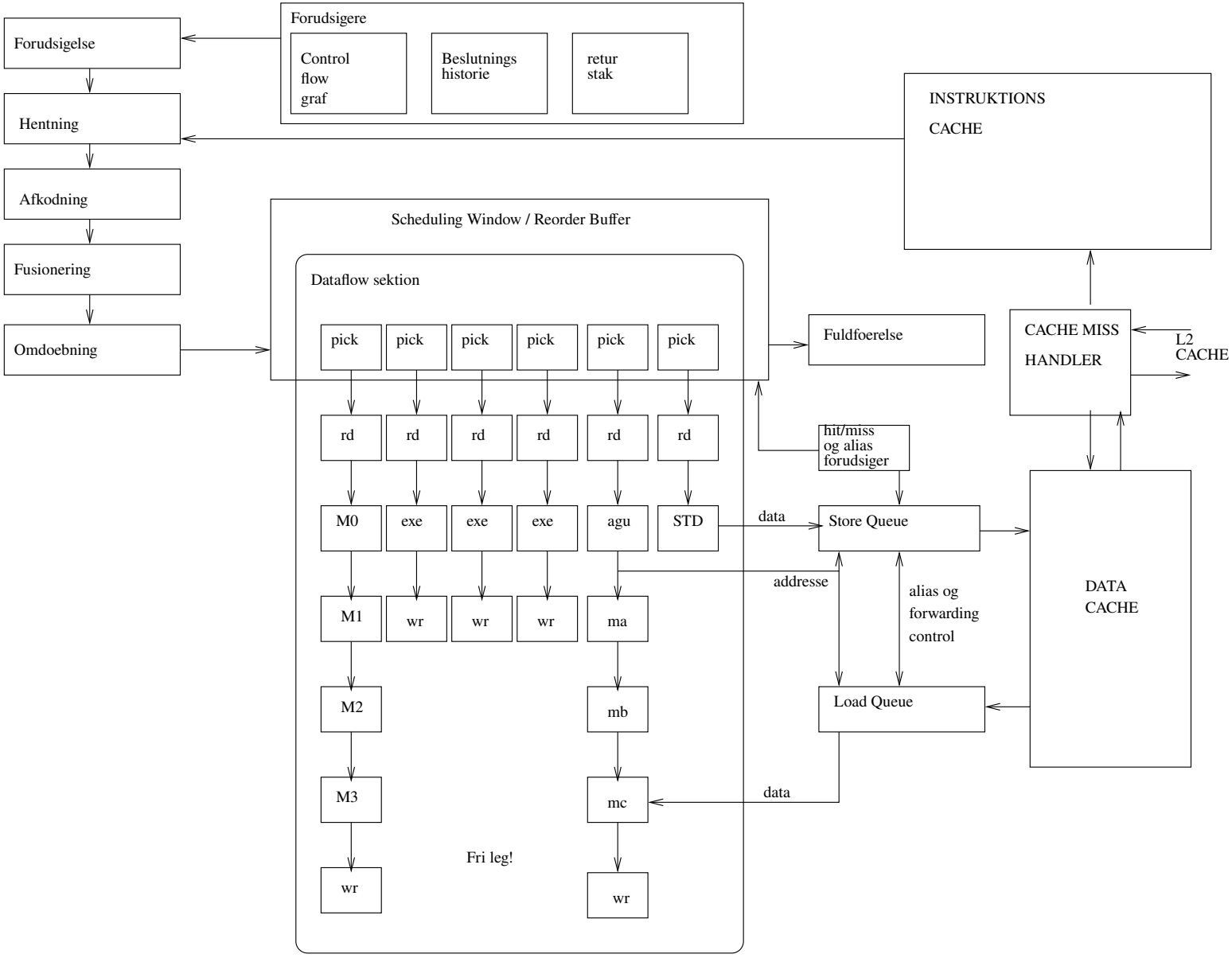
- Spekulativ udførelse
- Forudsigelse af programforløb (og mere…om lidt)
- Dataflow drevet udførelsesrækkefølge

Out-of-order maskiner består af en in-order del og en out-of-order del.

I mandags mødte vi (ikke muren, men) in-order delen af maskinen

- Forudsigelse og instruktionshentning
- Fusionering/Oversættelse af visse instruktioner til andre
- Register-omdøbning
- Fuldførelse/Commit

000 - Mikroarkitektur - overview



Out-of-order delen

I out-of-order delen finder vi følgende aktiviteter, organiseret i forskellige pipelines:

- pk: pick, udvælg instruktion
- rd: read, læs instruktionens input-registre, modtag forwardede værdier sidst i cycle.
- ex: execute, for aritmetik og hop
- ag: address generate, for load/store
- ma,mb,mc: cache opslag for load, cache check for store
- st: store data i store-kø
- wr/wb: write result to physical register

Den vigtigste opgave (i dag) er at få arrangeret udførelsen af instruktioner således at de passerer gennem de relevant aktiviteter i en rækkefølge der giver god ydeevne. Det sker i det pipeline trin der hedder "pk" for "pick instruction" Det er samtidigt det første trin i de små pipelines i maskinens dataflow sektion.

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 - Cycle 1

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

```
Parat: p0,p1,p2,p3                  Afvikling

LW imm(p0) -> p4 [parat]            -- pk
SUB p2,p4  -> p5 [ikke parat]       -- --
ADD p0,p2  -> p6 [parat]            -- --
SUB p6,p3  -> p7 [ikke parat]       -- --
```

Planlæggeren kan her kun vælge mellem første og tredje instruktion. Den vælger den første. En LW er fire cycles om at få sit resultat, så først fire cycles senere vil bitten for p4 være sat i paratvektoren.

Planlægning - Cycle 2

I denne cyklus er paratvektoren uændret

```
Parat: p0,p1,p2,p3                  Afvikling

LW imm(p0) -> p4 [startet]          -- pk rd
SUB p2,p4  -> p5 [ikke parat]       -- -- --
ADD p0,p2  -> p6 [parat]            -- -- pk
SUB p6,p3  -> p7 [ikke parat]       -- -- --
```

Planlæggeren kan her kun vælge den tredje instruktion. En ADD er en cycle om at beregne sit resultat, så allerede næste cycle sættes bit for p6 i parat-vektoren.

Planlægning - Cycle 3

I denne cyklus er p6 med i paratvektoren

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

LW imm(p0) -> p4 [startet]          -- pk rd ag
SUB p2,p4  -> p5 [ikke parat]       -- -- -- --
ADD p0,p2  -> p6 [startet]          -- -- pk rd
SUB p6,p3  -> p7 [parat]            -- -- -- pk
```

Planlæggeren kan her kun vælge den fjerde instruktion. En SUB er en cycle om at beregne sit resultat, så allerede næste cycle sættes bit for p7 i parat-vektoren.

Planlægning - Cycle 4

I denne cyklus er p7 med i paratvektoren

```
Parat: p0,p1,p2,p3,p6,p7            Afvikling

LW imm(p0) -> p4 [startet]          -- pk rd ag ma
SUB p2,p4  -> p5 [ikke parat]       -- -- -- -- --
ADD p0,p2  -> p6 [startet]          -- -- pk rd ex
SUB p6,p3  -> p7 [startet]          -- -- -- pk rd
```

Planlæggeren kan ikke vælge flere instruktioner i denne cyklus

Planlægning - Cycle 4

I denne cyklus er p4 med i startvektoren (det er 4. cycle, remember)

```
Parat: p0,p1,p2,p3,p4,p6,p7          Afvikling

LW imm(p0) -> p4 [startet]           -- pk rd ag ma mb
SUB p2,p4  -> p5 [parat]             -- -- -- -- -- pk
ADD p0,p2  -> p6 [startet]           -- -- pk rd ex wb
SUB p6,p3  -> p7 [startet]           -- -- -- pk rd ex
```

Planlæggeren kan kun vælge den anden instruktion. Det er en SUB der er en cycle om at beregne sit resultat, så allerede næste cycle tilføjes p5 til paratvektoren.

Planlægning - Cycle 5

I denne cyklus er p5 med i startvektoren

```
Parat: p0,p1,p2,p3,p4,p5,p6,p7      Afvikling

LW imm(p0) -> p4 [startet]          -- pk rd ag ma mb mc
SUB p2,p4  -> p5 [startet]          -- -- -- -- -- pk rd
ADD p0,p2  -> p6 [startet]          -- -- pk rd ex wb --
SUB p6,p3  -> p7 [startet]          -- -- -- pk rd ex wb
```

Nu er der ikke flere instruktioner der kan udvælges. Resten af forløbet følger bare de respektive pipelines.

Planlægning - efterfølgende cycles

```
Parat: p0,p1,p2,p3,p4,p5,p6,p7      Afvikling

LW imm(p0) -> p4 [startet]          -- pk rd ag ma mb mc wb --
SUB p2,p4  -> p5 [startet]          -- -- -- -- -- pk rd ex wb
ADD p0,p2  -> p6 [startet]          -- -- pk rd ex wb -- -- --
SUB p6,p3  -> p7 [startet]          -- -- -- pk rd ex wb -- --
```

Instruktionerne kan nu fuldføres (commit) og nye instruktioner kan indsættes i afviklingskredsløbet. Kredsløbet er selvfølgelig betydeligt større, så der samtidigt kan både indsættes, fuldføres(udtages) og scheduleres.

Planlægning - virker ikke altid!

I de tidligere slides byggede kredsløbet en rækkefølge på basis af antagelser om hvor længe instruktioner tager. Nemt for ADD, SUB osv.

Hvad med instruktioner hvor udførelsestiden kan variere. Hvad med Cache-miss? Vi antog uden videre at resultatet fra en load ville være klar 4 cycles senere.

```
                                  0  1  2  3  4  5  6  7  8
LW imm(p0) -> p4 [startet]        -- pk rd ag ma mb mc wb --
SUB p2,p4  -> p5 [startet]        -- -- -- -- -- pk rd ex wb
ADD p0,p2  -> p6 [startet]        -- -- pk rd ex wb -- -- --
SUB p6,p3  -> p7 [startet]        -- -- -- pk rd ex wb -- --
```

Det vides først sent i 'mc' trinnet om vi fik et hit eller miss. Det når vi til i cycle 6. Men vi udvalgte den afhængige instruktion til udførelse i cycle 5. Denne instruktion vil modtage en eller anden forkert værdi via forwarding sidst i cycle 6 og udføres i cycle 7 med forkert input.

Det er ikke en løsning at stalle de her pipelines! Hvad gør vi så?

Planlæging - fejlhåndtering

En ofte anvendt teknik består i at tage en lille backup af nogle af dataene fra planlægningskredsløbet, således at man kan "rulle det tilbage" til en tidligere tilstand. I det konkrete tilfælde kan vi i løbet af cycle 7 som er cyklen efter vi har detekteret cache-miss rulle planlægningskredsløbets tilstand tilbage til cycle 3 (markeret med !!):

```
                                  0  1  2  3  4  5  6  7
LW imm(p0) -> p4 [startet]        -- pk rd ag ma mb mc !!
SUB p2,p4  -> p5 [startet]        -- -- -- -- -- pk rd ex
ADD p0,p2  -> p6 [startet]        -- -- pk rd ex wb -- --
SUB p6,p3  -> p7 [startet]        -- -- -- pk rd ex wb --
```

Og cycle 8 bliver så:

```
Parat: p0,p1,p2,p3,p6,p7          Afvikling
                                  0  1  2  3  4  5  6  7  8
LW imm(p0) -> p4 [startet]        -- pk rd ag ma mb mc !! --
SUB p2,p4  -> p5 [ikke klar]      -- -- -- -- -- pk rd ex wb
ADD p0,p2  -> p6 [startet]        -- -- pk rd ex wb -- -- --
SUB p6,p3  -> p7 [startet]        -- -- -- pk rd ex wb -- --
```

Bemærk hvordan den afhængige instruktion bare fortsætter. Der kunne endda have været flere, det gør ikke noget. De vil blive genkørt

Planlægning - genstart

Når LOAD instruktionen flere cykler senere er klar til at levere sit resultat vil planlægningskredsløbet blive signaleret og p4 tilføjet til paratvektoren. Det ku ske i cycle 12:

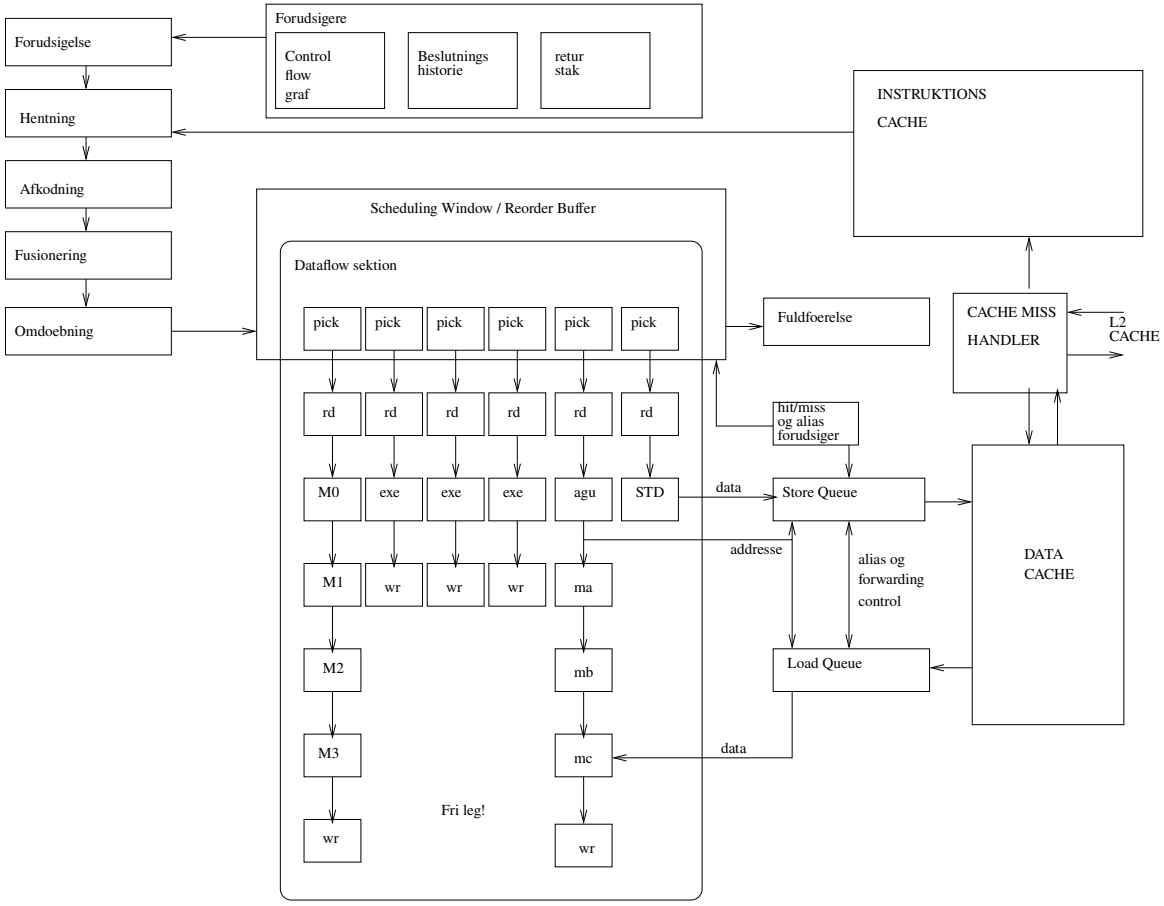
```
                                0  1  2  3  4  5  6  7  8  9  10 11 12
LW imm(p0) -> p4 [startet]      -- pk rd ag ma mb mc !! -- -- -- -- wb
SUB p2,p4  -> p5 [ikke klar]    -- -- -- -- -- pk rd ex wb -- -- -- --
ADD p0,p2  -> p6 [startet]      -- -- pk rd ex wb -- -- -- -- -- -- --
SUB p6,p3  -> p7 [startet]      -- -- -- pk rd ex wb -- -- -- -- -- --
```

Og så:

```
                                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
LW imm(p0) -> p4 [startet]      -- pk rd ag ma mb mc !! -- -- -- -- wb -- -- -- --
SUB p2,p4  -> p5 [startet]      -- -- -- -- -- pk rd ex wb -- -- -- -- pk rd ex wb
ADD p0,p2  -> p6 [startet]      -- -- pk rd ex wb -- -- -- -- -- -- -- -- -- -- --
SUB p6,p3  -> p7 [startet]      -- -- -- pk rd ex wb -- -- -- -- -- -- -- -- -- --
```

Vi vælger fra nu af at se bort fra forkert planlægning.

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" (som nævnt tidligere kan det have indebåret en fejl-planlægning og en genstart af planlægning. Det udelader vi i resten af præsentationen)

```
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 efter instruktionen fuldføres ("commit")
- Indtil da placeres skrivninger i en store-kø
- Vi viser ikke den endelige skrivning til lageret i vores afviklingsplot

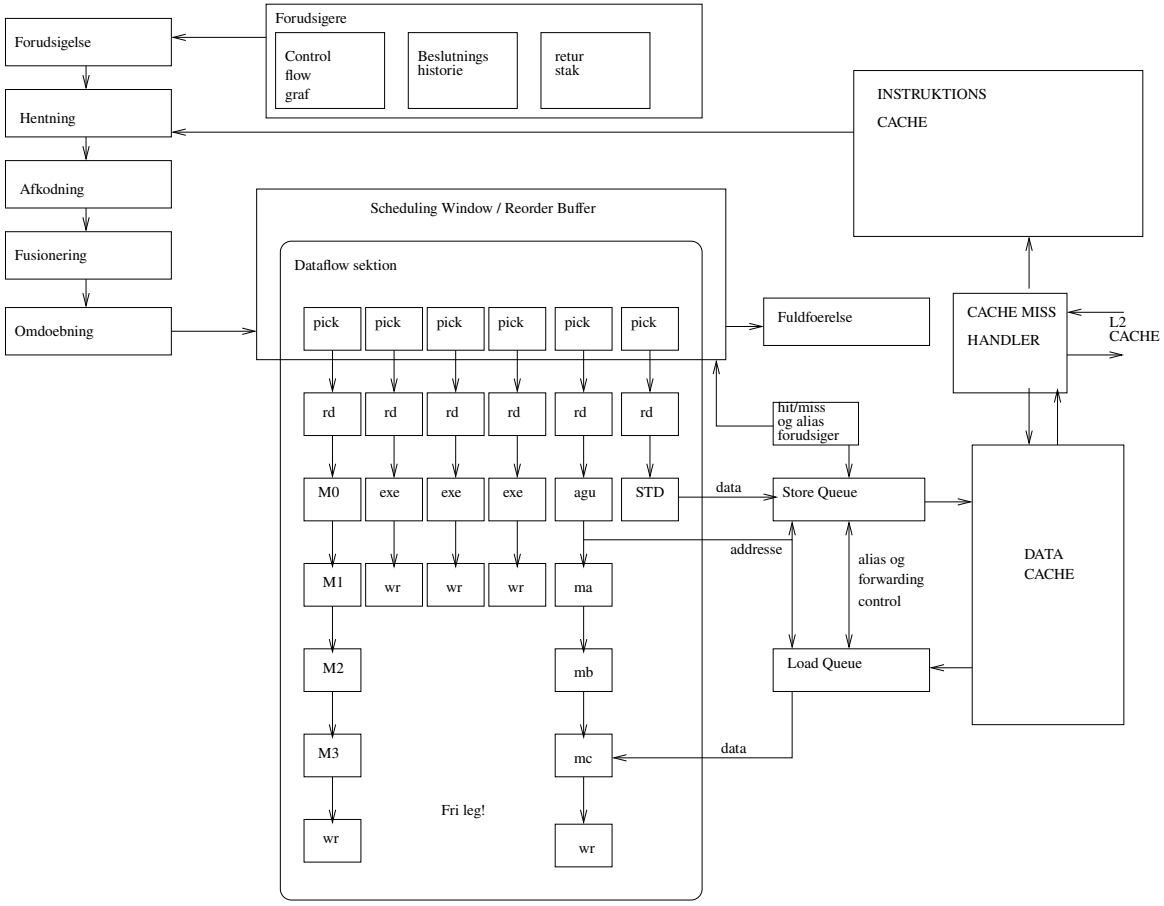
Hver skrivning i store-køen angiver (I programrækkefølge)

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

Addresse og Data elementerne opdateres separat af to forskellige "mikro" operationer: store-address og store-data. Enhver store instruktion opsplittes i disse to mikro operationer tidligere i pipelinen.

Selv om selve store køen indeholder skrivninger i programrækkefølge så opdateres info om disse skrivninger out-of-order.

Skrivning til lageret (II)



en STORE instruktion implementeres som to separate operationer

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

Store til load forwarding

Betragt følgende sekvens af instruktioner:

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

Hvor (i mikroarkitekturen) 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
add x12,x12,x4      Qu -- -- -- -- -- -- -- -- pk rd ex wr Ca Cb
sw  x12,40(x3)      Qu pk rd ag ma mb mc -- -- -- -- -- -- Ca Cb
-                   Q* -- -- -- -- -- -- -- -- -- pk rd st C*
lw  x12,40(x3)      Qu -- pk rd ag ma mb mc -- -- -- -- -- wb Ca Cb
addi x14,x12,40     Qu -- -- -- -- -- -- -- -- -- -- -- -- -- pk rd ex wb Ca Cb
```

Store data leveres sent fordi den første instruktion (add) udføres sent. Så den særlige mikro-operation for store-data kan først opdatere store køen i cycle 13.

Normalt ville load instruktionen have lavet "wb" i cycle 8, men på det tidspunkt er der ikke nogen data knyttet til den tidligere store instruktion.

I stedet kommer der en senere writeback i cycle 13, efter store-køen har modtaget data.

(og vi siger ikke noget om fejlagtig planlægning og genstart.....)

Tidligere store til ukendt adresse

Hvordan skal vi håndtere en tidligere store instruktion, der endnu ikke har fået beregnet sin addresse. Hvad skal efterfølgende load instruktioner gøre?

Muligheder:

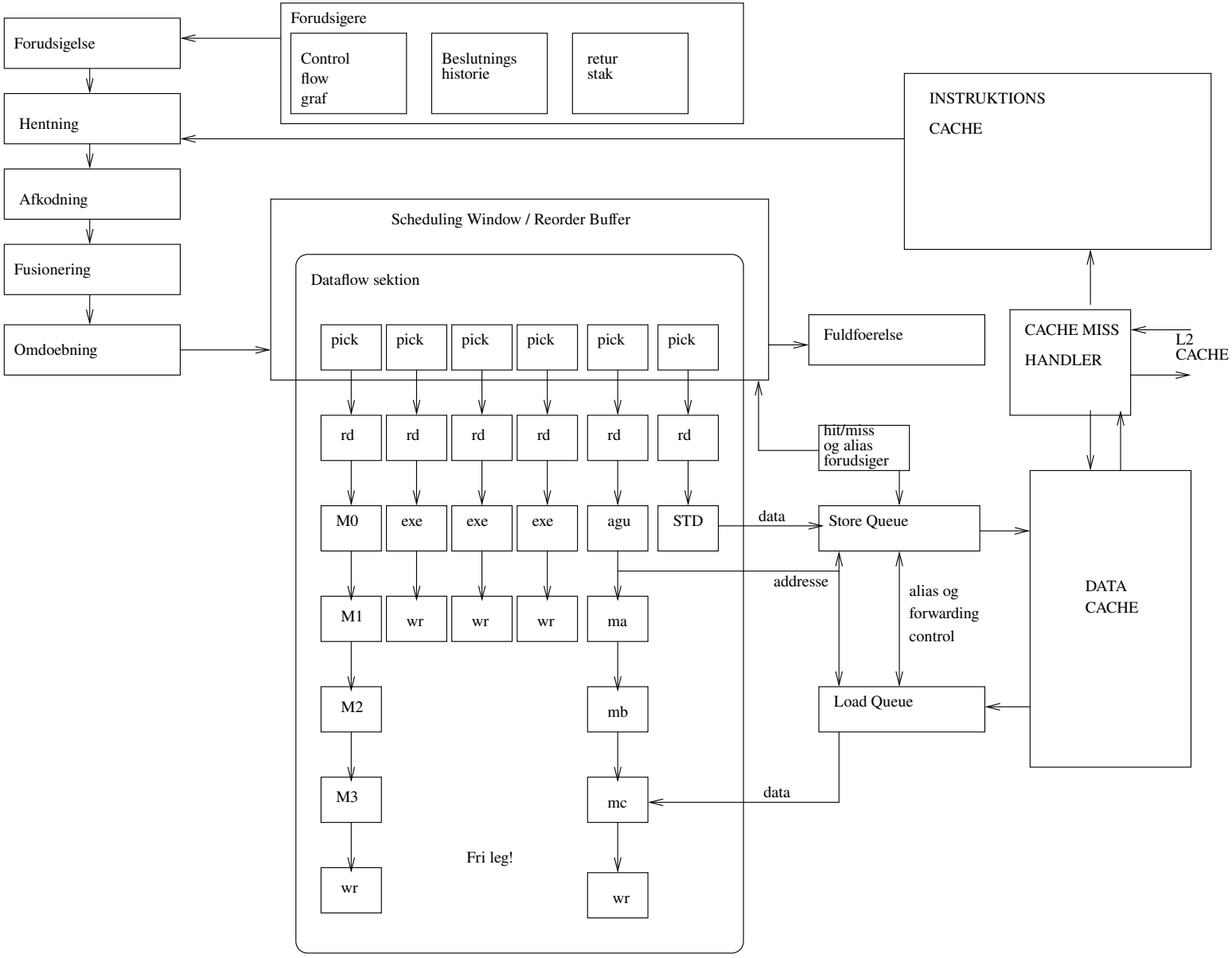
- Konservativt: load afventer store adresse.
- aggressivt: ignorer mulig afhængighed, opsaml load data fra cache og andre stores
- typisk: brug en forudsiger (alias-forudsiger) - valider forudsigelse senere

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

Vi antager at addressen er opdateret i 'mc', så en senere load kan foretage 'wb' en cycle senere.

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/

Parallel tilgang til lageret er meget vigtigt for ydeevnen

Det at kunne fortsætte tilgang til lageret på trods af et cache-miss er meget væsentligt. Det gør også at mange cache-miss kan detekteres tidligre så maskinen kan starte hentning af data der kan blive brug for senere så tidligt som muligt.

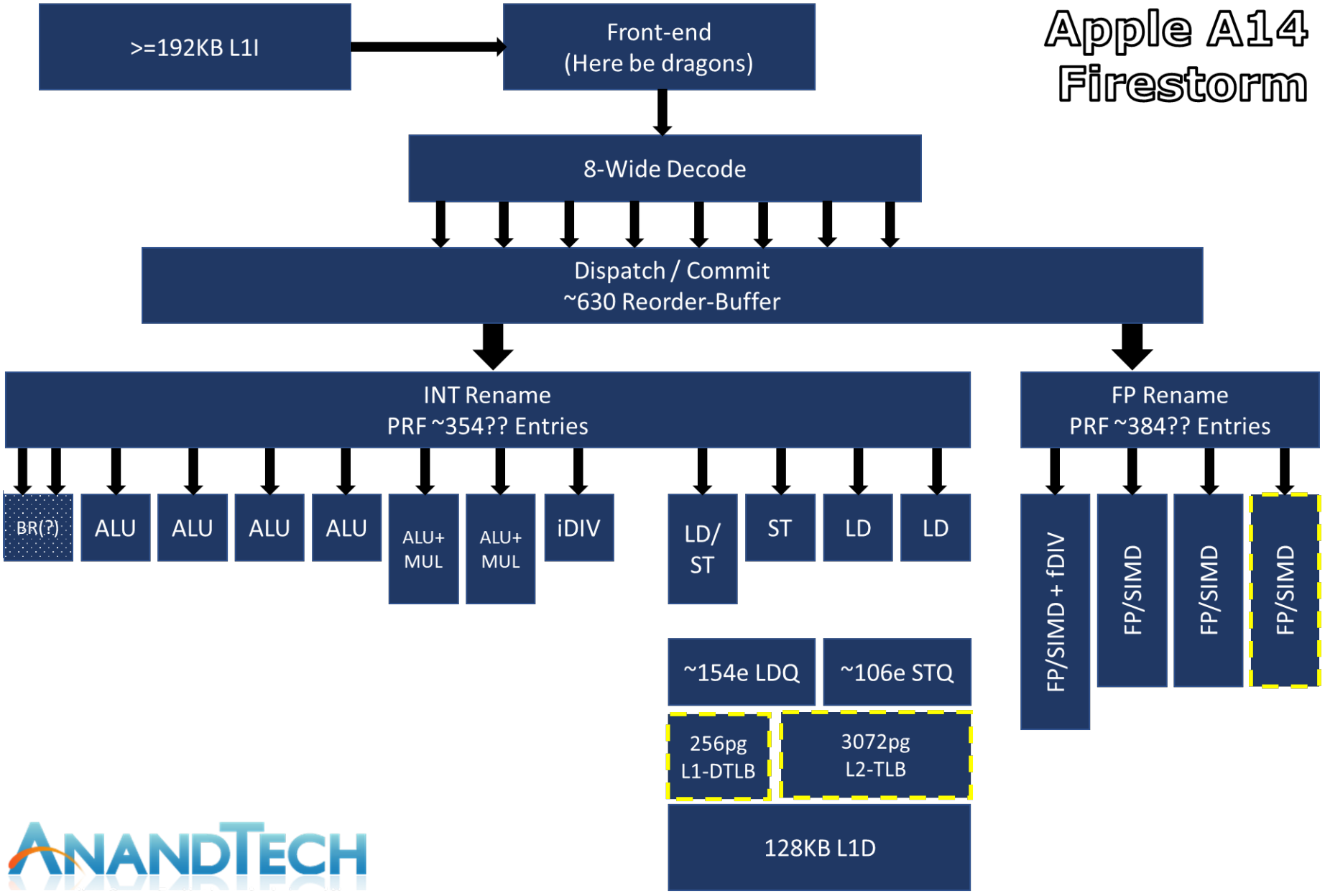
Og det fører til langt bedre udnyttelse af et stort lagerhierarki.

Masser af lagerblokke kan være i transit mellem caches og ind/ud fra lageret samtidigt (dusinvis pr Core - hundreder i alt i en server CPU).

Lidt provokerende kunne man vende det om: alt det her komplicerede out-of-order maskineri og bytten rundt på instruktioner osv, er der primært for at kunne generere cache-miss så hurtigt som muligt.

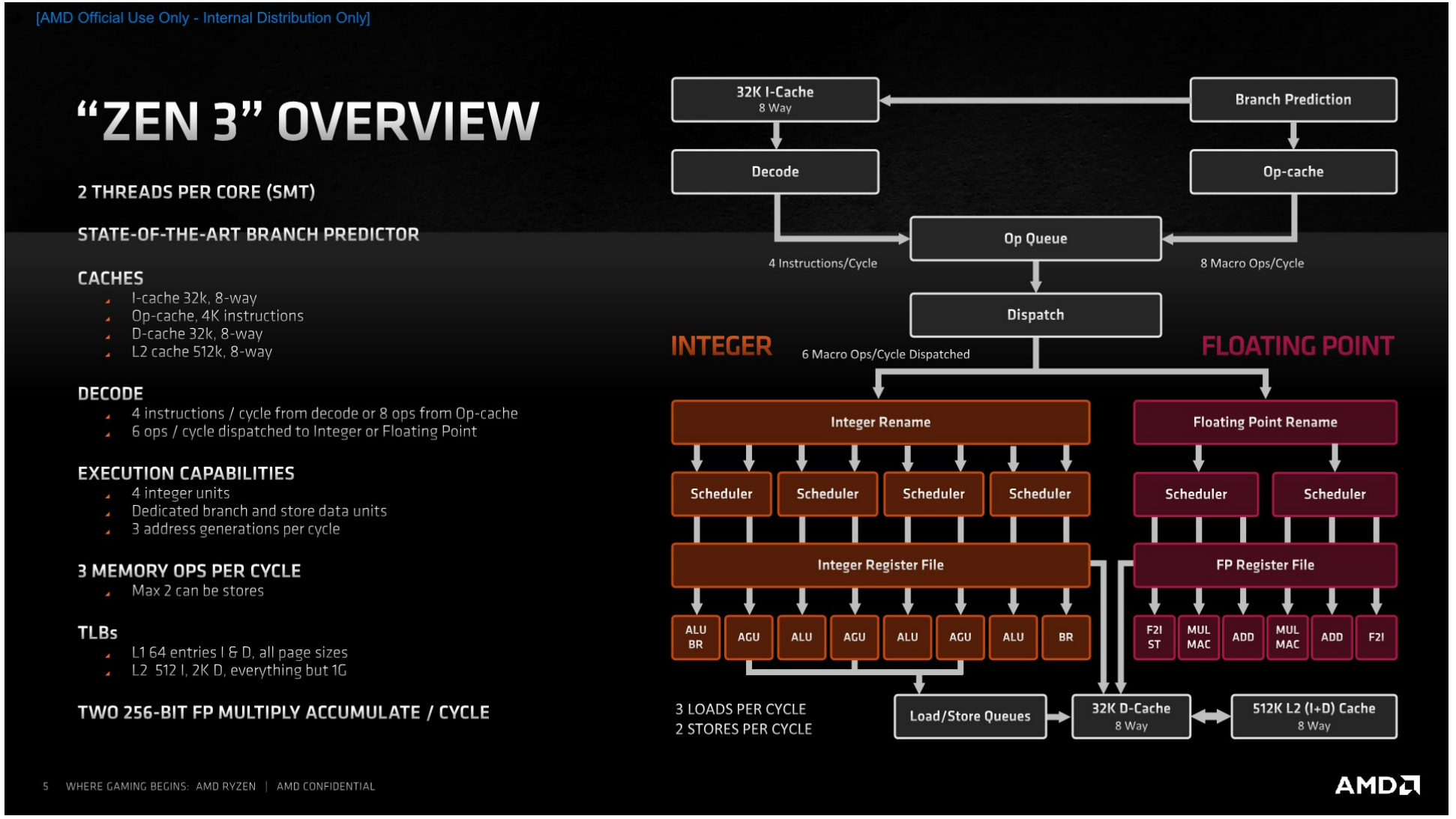
Og det er for at kunne generere de *rigtige* cache-miss at vi skal have god forudsigelse af programforløb.

En rigtig ARM (Apple M1) - rekonstrueret



ANANDTECH

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 eller RISCV? 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, dvs ved at give opgaven til Compileren. (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

(liiiige et indskud)

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. Derfor brugte man at have selve addresse-oversættelsen for kernen sat op, også i user mode.

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