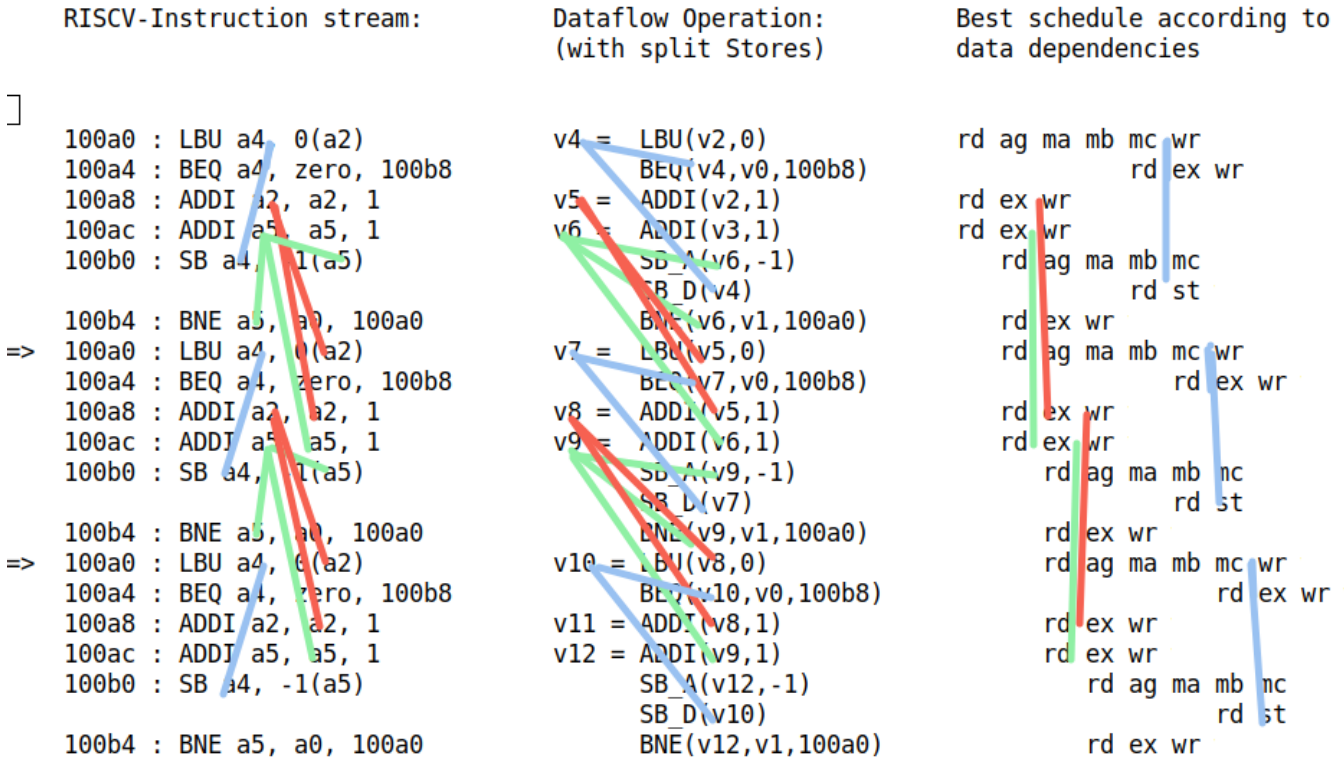
ILP - Instruction Level Parallelism

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.



Fra RISCV til Dataflow

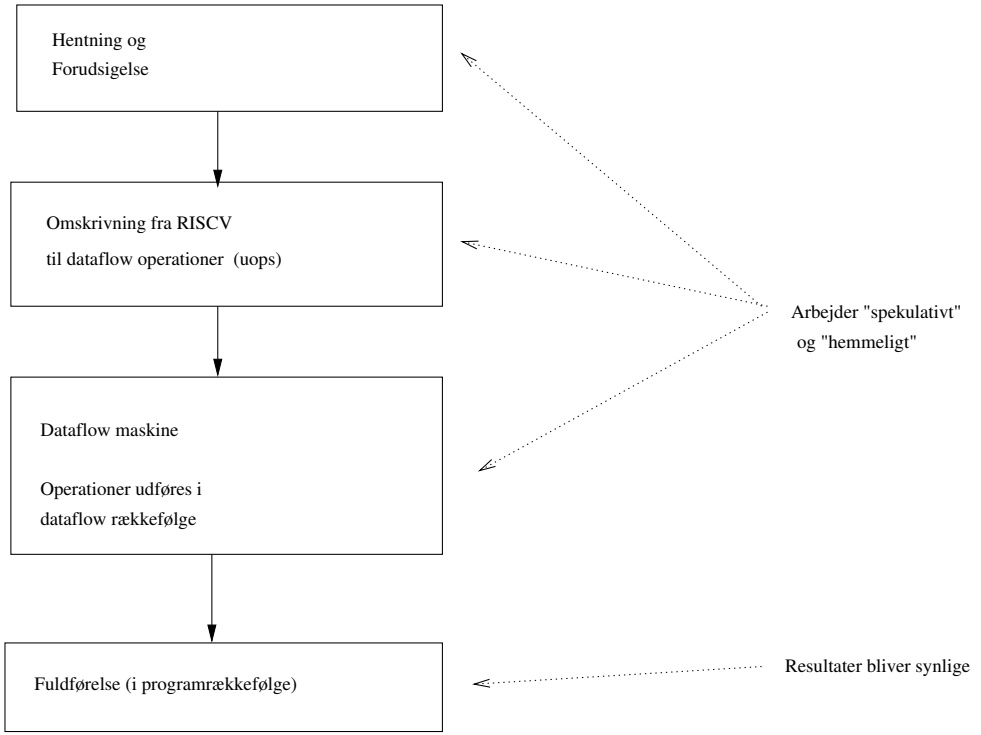
For at kunne bruge en dataflow maskine til at udføre programmer i RISCV maskinsprog (eller andet sekventielt maskinsprog) er det nødvendigt at oversætte RISCV instruktionerne.

Sådan en maskine er opbygget som 4 dele organiseret som en pipeline

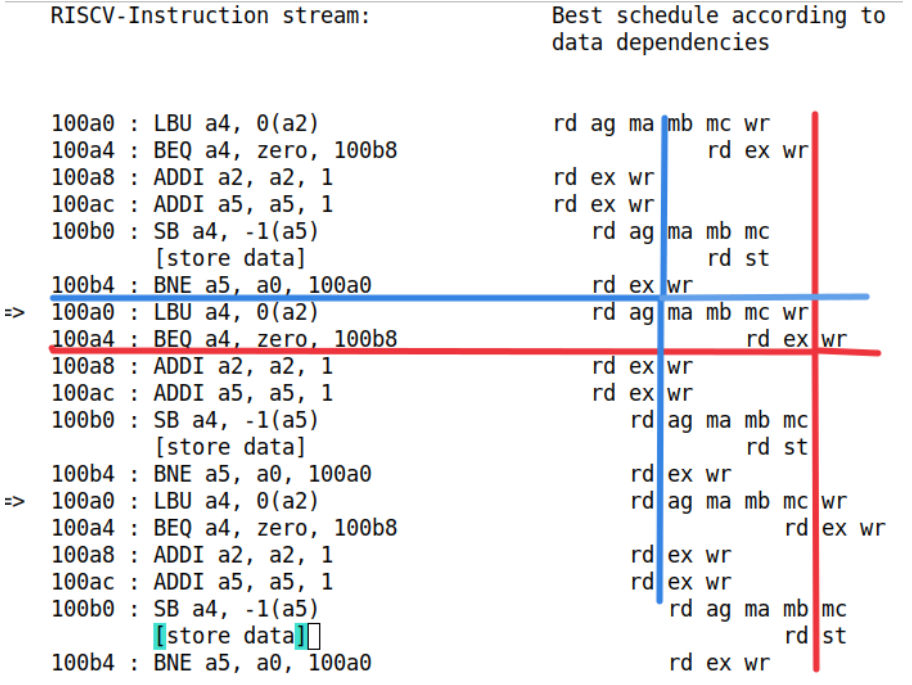
- Hentning og forudsigelse af programforløbet. Det sker i program rækkefølge.
- Omskrivning af RISCV instruktioner til de operationer som dataflow maskinen forstår. Ligeledes i program rækkefølge.
- Selve dataflow delen. Her udføres dataflow operationer i dataflow rækkefølge af flere små pipelines. De styres som beskrevet i tidligere forelæsning.
- Fuldførelse - her opsamles resultater produceret i dataflow rækkefølge og "præsenteres" i RISCV-program-rækkefølge

For at få god ydeevne skal man hurtigt kunne hente instruktioner og fylde dataflow-delen af maskinen med dem. Nu om dage gerne 4-8 instruktioner per clock-periode.

Fra RISCV til Dataflow



Spekulativ udførelse?



Instruktioner under den røde streg må og skal udføres *spekulativt* relativ til hoppet på adresse 100a4. Hvis vi i stedet afventede hoppets afgørelse, ville vi tabe muligheden for alt for meget parallelt arbejde.

Dataflow vs sekventielt maskinsprog

En helt simpel dataflow maskine ved ikke noget om evt rækkefølge af instruktioner. For at kunne bruge den til at udføre et sekventielt program er vi nød til at tilføje nogle mekanismer der gør.

- I sidste forelæsning berørte vi kort behovet for spekulativ udførelse. Og dermed indirekte behovet for at kunne annullere instruktioner som er hentet og anbragt i dataflow maskinen, men som så viser sig ikke at skulle udføres.
- Vi tilføjede en STORE-kø, som holder spekulative skrivninger til lageret indtil de ikke er spekulative længere. Spekulative skrivninger må ikke nå til lageret.
- Det, at lagerreferencer har en rækkefølge (har sekventiel semantik) fører til at vi må sammenligne laggerreferencers adresse for at afgøre om der er en lagerbåren afhængighed (LOAD instruktioner skal se tidligere STORE instruktioner). Så vi gjorde STORE-køen søgbar for LOAD instruktionerne. Vi har også splittet STORE i to operationer der kunne udføres uafhængigt af hinanden for at kunne beregne adresser så tidligt som muligt.

Omskrivning af instruktioner

Når en bunke RISCV instruktioner ankommer fra instruktionscachen bruges de til at generere tilsvarende dataflow-operationer. I tilfældet RISCV er det ret nemt, for mere komplicerede maskinsprog (x86) er det omfattende.

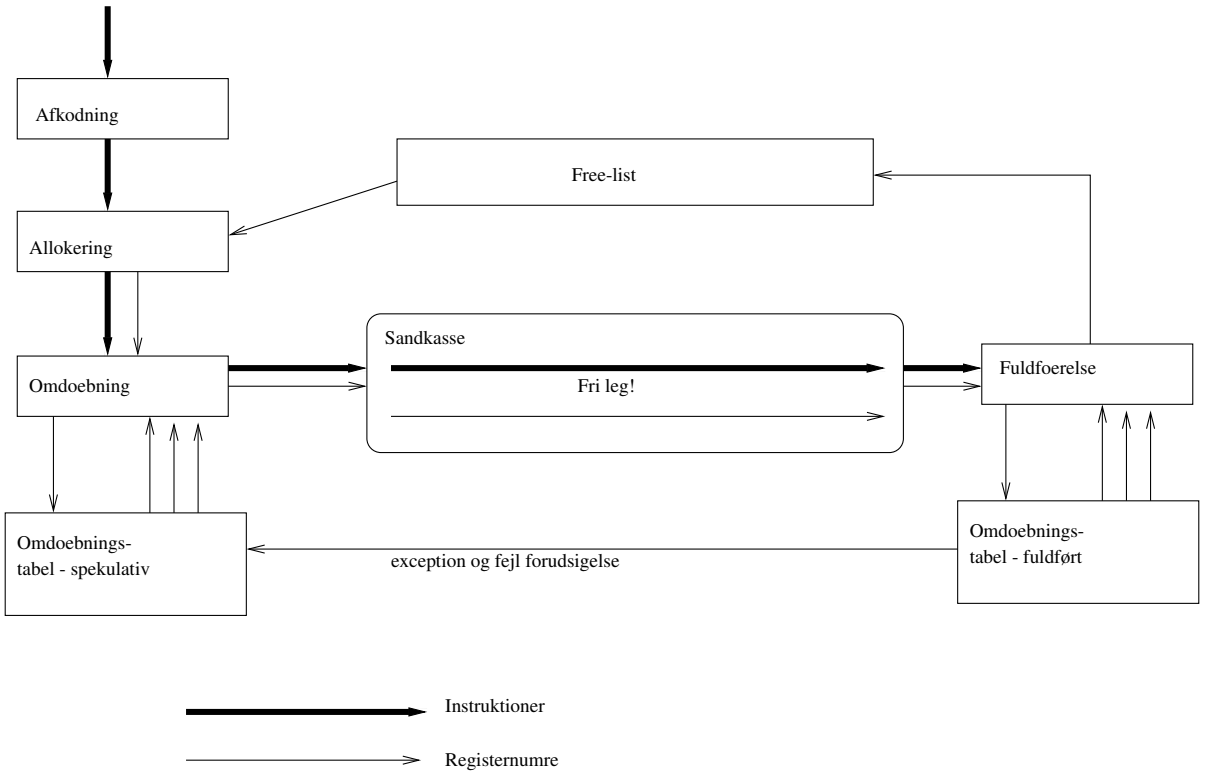
I vores eksempel-maskine her på CompSys fylder processen 5 pipeline trin. I virkelige maskiner kan det være såvel færre som flere. Vi bruger:

```
De  Decode, Afkodning.
    Tillige fastlægges afhængigheder mellem instruktionerne
Fu  Fusion.
    Hvisse instruktions-sekvenser slås sammen til en enkelt operation
Al  Allocate.
    Her allokeres resourcer og fysiske registre
Re  Rename. Omdøbning.
    Dataflow operationer får fysiske register numre i stedet for RISCV registre
Qu  Queue.
    De genererede operationer indsættes i dataflow-maskinen.
```

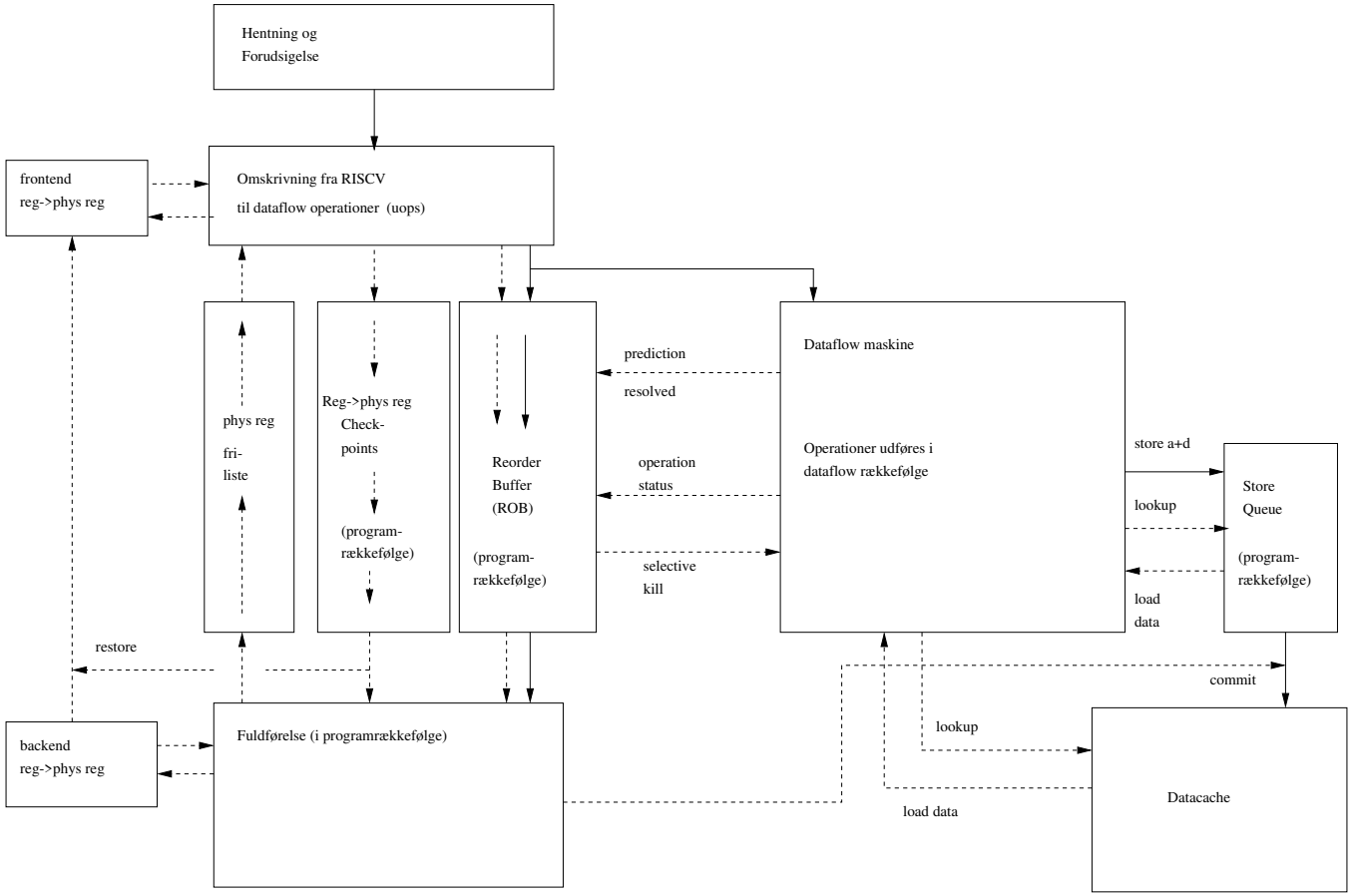
For at forstå hvordan denne del af maskinen virker må vi starte med at få et overblik over hvordan vi får en parallel maskine til at have sekventiel semantik

Omskrivning af instruktioner (II)

Registeromdøbning sikrer at alle data-afhængigheder i en programsekvens kan identificeres og addresseres som fysiske registre. Registeromdøbning er centralt i omskrivning fra ISA til dataflow representation:



Mikroarkitektur



(Hav denne figur synlig samtidigt med de næste slides)

Instruktions-flow

(Se diagram på tidligere slide samtidigt)

Vi opretholder program-rækkefølgen af instruktioner vha en ROB - "re-order buffer". Den kan opfattes som en lang liste af instrukioner. Nye instruktioner indsættes i den ene ende samtidig med at de sendes til dataflow maskinen. De ældste instruktioner udtages af den anden ende i programrækkefølge når de er udført af dataflow maskinen.

Dataflow-delen signalerer til ROB'en når en operation er fuldført. Det sker selvfølgelig out-of-order. Instruktioner kan fejle (e.g. page-fault, division med nul) og hvis det sker, signaleres det også til ROB'en.

Control-flow instruktioner forudsiges tidligere i pipelinen, men forudsigelserne verificeres i dataflow-delen (I "ex" trinnet af C-flow pipelinen) og fejl-forudsigelser signaleres til ROB'en.

Den sidste del af maskinen, kaldet "Fuldførelse", tager instruktioner ud af ROB'en i programrækkefølge. Når en STORE instruktion (der ikke har fejlet) udtages, signaleres til STORE-køen og den ældste skrivning i køen udføres, dvs datacachen opdateres.

Instruktions-flow (II)

Lad os antage at vi kan indsætte dataflow-operationer fra 4 instruktioner/ clock og fuldføre 4 instruktioner pr clock. Det vil give følgende afvikling af vores programeksempel:

```
     RISCV-Instruction stream:

     100a0 : LBU a4, 0(a2)
     100a4 : BEQ a4, zero, 100b8
     100a8 : ADDI a2, a2, 1
     100ac : ADDI a5, a5, 1
     100b0 : SB a4, -1(a5)
             [store data]
     100b4 : BNE a5, a0, 100a0
=>   100a0 : LBU a4, 0(a2)
     100a4 : BEQ a4, zero, 100b8
     100a8 : ADDI a2, a2, 1
     100ac : ADDI a5, a5, 1
     100b0 : SB a4, -1(a5)
             [store data]
     100b4 : BNE a5, a0, 100a0
=>   100a0 : LBU a4, 0(a2)
     100a4 : BEQ a4, zero, 100b8
     100a8 : ADDI a2, a2, 1
     100ac : ADDI a5, a5, 1
     100b0 : SB a4, -1(a5)
             [store data]
     100b4 : BNE a5, a0, 100a0

Qu = Enqueue instruction/uop
Q* = Enqueue store-data uop
```

```
Schedule on small realistic machine

Qu pk rd ag ma mb mc wr Ca Cb
Qu -- -- -- -- pk rd ex wr Ca Cb
Qu pk rd ex wr             Ca Cb
Qu -- pk rd ex wr          Ca Cb
   Qu -- pk rd ag ma mb mc Ca Cb
   Q* -- -- -- pk rd st    C*
   Qu -- pk rd ex wr             Ca Cb
   Qu -- -- pk rd ag ma mb mc wr Ca Cb
   Qu -- -- -- -- -- -- pk rd ex wr Ca Cb
      Qu pk rd ex wr                Ca Cb
      Qu -- pk rd ex wr             Ca Cb
      Qu -- -- pk rd ag ma mb mc    Ca Cb
      Q* -- -- -- -- -- pk rd st    C*
      Qu -- -- pk rd ex wr             Ca Cb
         Qu -- -- pk rd ag ma mb mc wr Ca Cb
         Qu -- -- -- -- -- -- pk rd ex wr Ca Cb
         Qu -- pk rd ex wr                Ca Cb
         Qu -- -- pk rd ex wr             Ca Cb
            Qu -- -- pk rd ag ma mb mc    Ca Cb
            Q* -- -- -- -- -- -- pk rd st C*
            Qu -- -- pk rd ex wr             Ca Cb

 Ca = Commit/Retire, stage A
 Cb = Commit/Retire, stage B
 C* = Commit/Retire store-data uop
```

Registeromdøbning (I)

(Se diagram på tidligere slide samtidigt)

Værdier i dataflow maskinen har hver en unik ID, også kaldet det fysiske registernummer.

Vi vedligeholder 2 omdøbnings-tabeller (RAT, register alias table) der for hver RISCV register angiver det tilsvarende fysiske registernummer. Vi har en omdøbnings-tabel i forenden af pipelinen og en anden i bagenden. Tabellen i forenden afspejler maskinens tilstand når nye instruktioner omskrives og tilføjes til ROB'en. Tabellen i bagenden afspejler maskinens tilstand når de ældste instruktioner skal fuldføres.

Under omskrivning fra RISCV til dataflow operationer oversættes hvert RISCV kilde-register reference til det tilsvarende fysiske registernummer ved opslag i tabellen. Til hvert destinations-register allokeres et ny fysiske register fra fri-listen og tabellen opdateres med en ny binding fra RISCV-register til fysisk register. Processen kaldes "registeromdøbning" (eng: "register renaming")

De fysiske registernumre ledsager instruktionen på resten af dens vej gennem maskinen.

Registeromdøbning (II)

Instruktionen får også det logiske destinationsregister(nummer) med, samt det "gamle" fysiske register nummer der tidligere har været bundet til destinationsregisteret.

Det logiske registernummer bruges sammem med det fysiske registernummer til at opdatere omdøbnings-tabellen ved fuldførelse.

Det "gamle" fysiske registernummer tilføjes til frilisten når instruktionen fuldføres.

Registeromdøbning (III)

Vi bruger to ekstra pipeline trin til registeromdøbning. Et trin til at allokere registre og øvrige resourcer (plads i skrive-kø, ROB, checkpoint kø) og et trin til at generere de færdige dataflow-operationer med omdøbte registre.

```
    RISCV-Instruction stream:              Schedule on small realistic machine


    100a0 : LBU a4, 0(a2)                  Al Rn Qu pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8            Al Rn Qu -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                 Al Rn Qu pk rd ex wr             Ca Cb
    100ac : ADDI a5, a5, 1                 Al Rn Qu -- pk rd ex wr          Ca Cb
    100b0 : SB a4, -1(a5)                     Al Rn Qu -- pk rd ag ma mb mc Ca Cb
            [store data]                            Q* -- -- -- pk rd st    C*
    100b4 : BNE a5, a0, 100a0                 Al Rn Qu -- pk rd ex wr             Ca Cb
=>  100a0 : LBU a4, 0(a2)                     Al Rn Qu -- -- pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8               Al Rn Qu -- -- -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                       Al Rn Qu pk rd ex wr                Ca Cb
    100ac : ADDI a5, a5, 1                       Al Rn Qu -- pk rd ex wr             Ca Cb
    100b0 : SB a4, -1(a5)                        Al Rn Qu -- -- pk rd ag ma mb mc    Ca Cb
            [store data]                               Q* -- -- -- -- -- pk rd st    C*
    100b4 : BNE a5, a0, 100a0                    Al Rn Qu -- -- pk rd ex wr             Ca Cb
=>  100a0 : LBU a4, 0(a2)                           Al Rn Qu -- -- pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8                     Al Rn Qu -- -- -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                          Al Rn Qu -- pk rd ex wr                Ca Cb
    100ac : ADDI a5, a5, 1                          Al Rn Qu -- -- pk rd ex wr             Ca Cb
    100b0 : SB a4, -1(a5)                              Al Rn Qu -- -- pk rd ag ma mb mc    Ca Cb
            [store data]                                     Q* -- -- -- -- -- -- pk rd st C*
    100b4 : BNE a5, a0, 100a0                          Al Rn Qu -- -- pk rd ex wr             Ca Cb

Qu = Enqueue instruction/uop               Ca = Commit/Retire, stage A
Q* = Enqueue store-data uop                Cb = Commit/Retire, stage B
Al = Allocate regs+resources               C* = Commit/Retire store-data uop
Rn = Rename
```

Allokering og Omdøbning

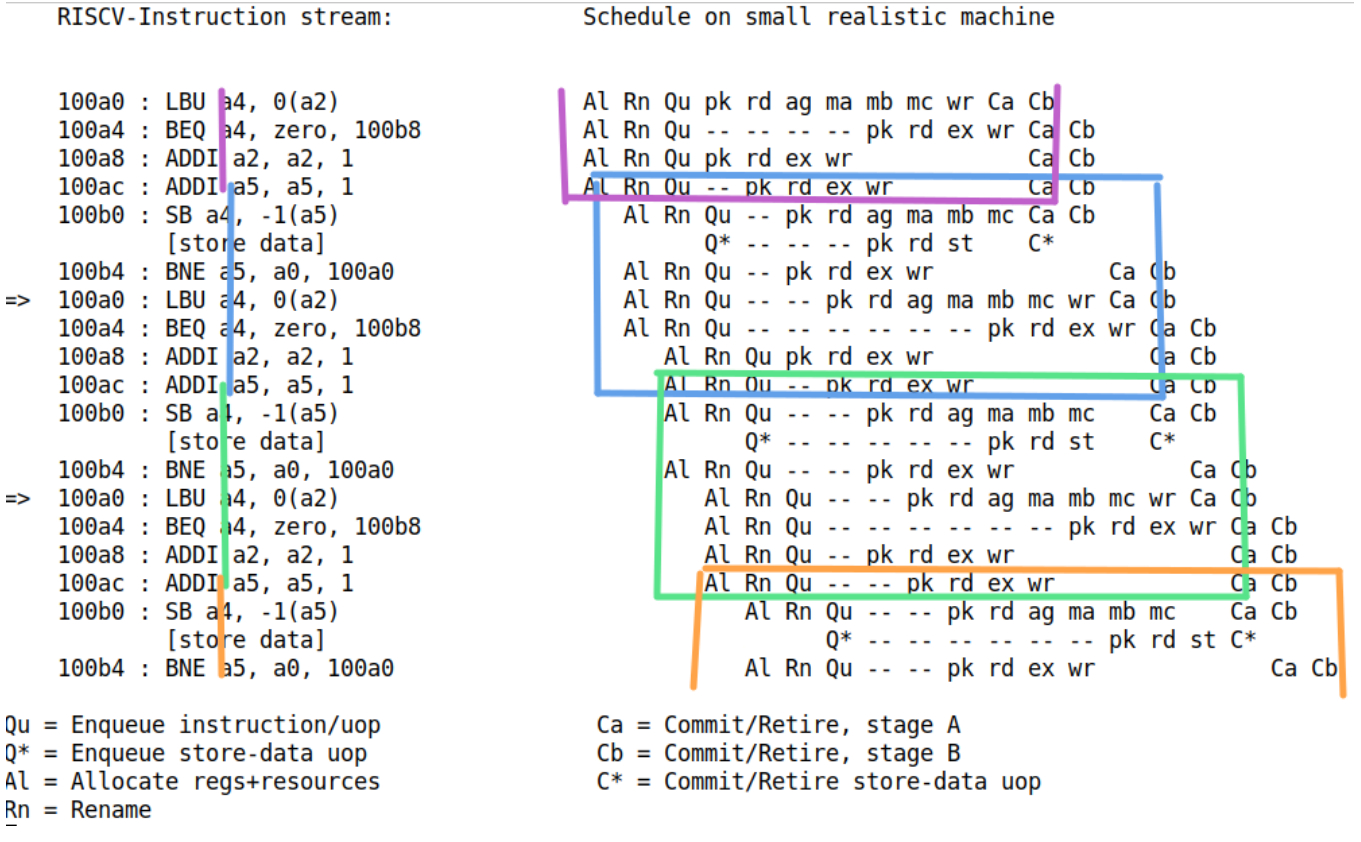
Efter fusionering ved man hvor mange dataflow operationer man producerer, hvor mange værdier operationerne vil producere og hvilke pipelines i dataflow maskinen der skal bruges.

- Der allokeres pladser i schedulerings-kredsløbene i de relevante pipelines til de nye dataflow operationer.
- Efter fusionering ved man hvor mange V-ID'er der er brug for til nye resultater. De allokeres fra fri-listen.
- Der allokeres plads i ROB'en til instruktionerne.
- Om nødvendigt allokeres plads i checkpoint-køen til et nyt checkpoint

Hvis (dele af) allokeringen ikke kan gennemføres må forenden af pipelinen stalle. Til slut genereres de færdige dataflow operationer. Afhængigheder mellem dem er udtrykt i form af fysiske register numre.

Register renaming og fuldførelse

Fysisk register for et destinations register kan frigives når vi fuldfører næste senere instruktion, der skriver til samme register, som illustreret her:



Hvert rektangel markerer *en* binding fra register a5 til et fysisk register

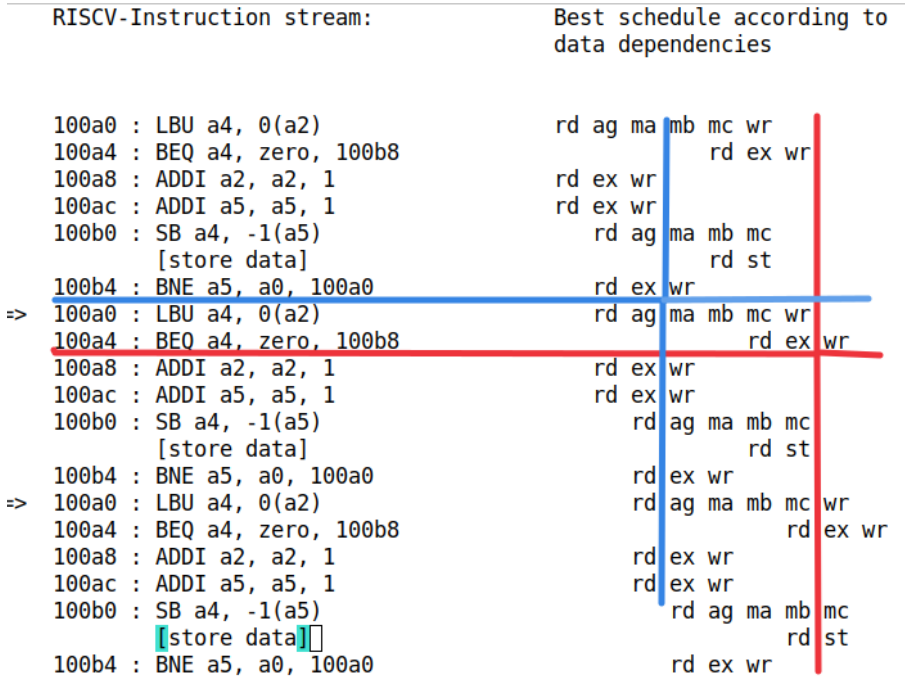
Sekventiel semantik

Hvad vil det sige i praksis:

- Man kan "stoppe" maskinen ved en given instruktion. Vi kalder denne instruktion for SGI ("sidste gyldige instruktion")
- Tidligere instruktioner i programforløbet skal da fremstå fuldt udførte.
- Senere instruktioner i programforløbet må ikke have ændret maskinens tilstand.

Ved maskinens "tilstand" forstås her, den tilstand som en senere programstump kan "se" ved at læse registre eller lager, eller den tilstand af lageret som en programstump på en anden processor kan "se" via delt lager.

Sekventiel semantik (II)



- Til venstre, øverst: Er udført, skulle også være udført.
- Til højre, øverst: Er ikke udført, skal udføres
- Til venstre, nederst: Er udført, skukke IKKE være det.
- Til højre, nederst: Er ikke udført, skal ikke udføres

Sekventiel semantik (III)

Vi har flere opgaver

- Vi skal kunne stoppe instruktioner/dataflow operationer der er senere i udførelsesrækkefølgen end SGIen
- Vi skal lade instruktioner der er tidligere end SGIen i udførelsesrækkefølgen fortsætte
- Vi skal kunne fjerne/skjule ændringer i registre eller lager fra instruktioner der er senere i udførelsesrækkefølgen end SGIen, selvom de tilsvarende dataflow operationer er fuldført.
- Vi skal stadig kunne "se" effekten af logisk tidligere instruktioner, selvom deres operationer ikke er udført, endsige påbegyndt.

Selektiv annullering.

Mål: At stoppe/fjerne al aktivitet der følger grænse-instruktionen.

Mekanisme:

- Hver instruktion/dataflow operation har en "valid" bit med sig. Kun hvis denne er sat arbejdes der på instruktionen.
- Hver instruktion/dataflow operation tildeles et nummer der angiver dens plads i program-rækkefølgen. Det gøres i "Al" trinnet.
- Nummeret følger instruktioner/operationer overalt i maskinen.
- Ved annullering rundsendes nummeret tilhørende SGIen.
- Hver cycle sammenligner et dedikeret kredsløb hver instruktions nummer med en evt annulleringsordre.
- Instruktioner med et nummer højere end SGIen får nulstillet deres "valid"-bit.
- Det gælder også for instruktionerne i ROB'en. Herefter er SGIen den yngste instruktion i ROB'en

Annullering af skrivning til lageret

Skrivninger til lageret havner i STORE-køen. Hvert element i køen får tilføjet nummeret på den skrivende instruktion og elementet kan annulleres på samme måde som instruktioner og dataflow operationer kan annulleres.

pyha - den var nem

Annullering af skrivning til registre

Vi har brug for at retablere omdøbningstabellerne der kobler registernumre til fysiske registre og for at frigive fysiske registre der er allokeret til instruktioner efter SGIen. I princippet er al nødvendig information vedhæftet instruktionerne i ROBen. Man kan udtage instruktionerne i rækkefølge og opdatere omdøbningstabellen i bagenden af pipelinen indtil man udtager SGIen.

I stedet bruges en kombination af checkpointing og patching. Med mellemrum tages et checkpoint af hele tabellen i forenden af pipelinen. Det tilføjes til en checkpoint kø og følger når instruktioner bevæger sig gennem ROB'en. Hver instruktion i ROB'en bærer nummeret på det sidst sete checkpoint med sig.

Vi kan nu retablere en en vilkårlig tidligere afbildning således:

- Vi retablerer omdøbningstabellen fra det checkpoint der er knyttet til SGIen.
- Det kan svare til en instruktion før SGIen. I så fald "patches" tabellen sekventielt ud fra instruktionerne mellem checkpoint og SGI.
- Nyere checkpoints frigives

Denne mekanisme kan retablere tilstanden efter en vilkårlig instruktion.

Håndtering af fejlagtig forudsigelse

Den væsentligste brug af selektiv anullering er ved håndtering af forkert forudsagte control-flow instruktioner. Når en fejlforudsigelse detekteres udføres selektiv anullering med den fejl forudsagte instruktion som SGI. Samtidig omdirigeres instruktionshentning til den rette adresse.

Selectiv anullering er omhyggeligt designet så den kan gennemføres på det antal clock-cycles der er til rådighed før den omdirigerede instruktionsstrøm når frem til trinnet med register-omdøbning.

Fejlhåndtering (Exceptions)

Fejl (f.eks. page fault, unaligned access, division med nul) håndteres på lignende vis. Her udføres selektiv anullering med udgangspunkt i instruktionen *før* den fejlende instruktion. Hermed ser det ud som om den fejlende instruktion slet ikke er udført.

Maskinen skifter til "exception processing" - dvs adresse på fejlende instruktion gemmes i et særligt register, der skiftes til supervisor/kernel/ priviligeret mode og instruktions-hentning omdirigeres til den relevante exception rutine.

For nogle fejl kan operativsystemet vælge at fortsætte udførelsen ved at gentage forløbet inklusiv den tidligere fejlende instruktion -- nu under forhold, hvor den forhåbentlig ikke fejler. Eksempelvis ved først at bringe en side med data ind fra disk og ændre sidetabellerne tilsvarende.

Afkodning

Før vi kan gennemføre resource allokering og registeromdøbning må vi afkode RISCV instruktionerne.

Hver instruktion afkodes tilstrækkeligt til at vi senere kan bestemme hvilke resourcer den har brug.

Afhængigheder mellem instruktionerne indbyrdes bestemmes.

Instruktions-indkodningen har stor betydning for omkostning i både kredsløbsstørrelser og antallet af pipeline trin. For RISCV-er processen simpel, og især faciliteret af den meget regulære indkodning, som gør at kilde-registre og destinations-registre altid er placeret i de samme felter i instruktionsindkodningen.

For AMD64 er det meget kompliceret. Derfor ses det ofte at implementationer af AMD64 cacher afkodningsresultater eller hele "oversættelser" til et simplere RISC-lignende format.

Fusionering

Fusionering omskriver instruktionspar til interne instruktioner som kan udføres hurtigere. Oftest skal begge instruktioner i et par have samme destinations register. Eksempler:

```
lui x3,immA
addi x3,x3,immB         -->  li x3,immA+immB

sll x5,x4,imm
add x5,x5,x6            -->  shiftadd x5,x4,x6,cnst

add x5,x4,x3            -->  lw2 x5,imm(x4,x3)
lw x5,imm(x5)
```

Det er selvfølgelig kun muligt, hvis maskinens datavej understøtter de nye interne instruktioner. I ovenstående indgår f.eks. en ALU der ved addition tillige kan skifte det ene input mod venstre.

Særligt for AMD64

AMD64 fusionerer ofte CMP og Bcc (compare og betinget hop). Det gør RISCV ikke, da det allerede er en enkelt instruktion.

AMD64 fusionerer ofte MOV instruktioner med efterfølgende instruktioner, en process der benævnes "move elimination". Det er særlig vigtigt for AMD64, fordi instruktionerne kun kan angive to operander. Så hvis compileren har behov for at få udført A = B op C, hvor både B og C skal bruges senere, så må den generere

```
A = mov B
A = A op C
```

Ved move elimination vil maskinen så transformere denne sekvens "tilbage" til en RISCV-lignende intern repræsentation af A = B op C.

Afviklingsplot

Her ses afviklingen af vores eksempel instruktionsstrøm med afkodning og fusionering tilføjet.

```
    RISCV-Instruction stream:             Schedule on small realistic machine

    100a0 : LBU a4, 0(a2)                 De Fu Al Rn Qu pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8           De Fu Al Rn Qu -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                De Fu Al Rn Qu pk rd ex wr             Ca Cb
    100ac : ADDI a5, a5, 1                De Fu Al Rn Qu -- pk rd ex wr          Ca Cb
    100b0 : SB a4, -1(a5)                    De Fu Al Rn Qu -- pk rd ag ma mb mc Ca Cb
            [store data]                                Q* -- -- -- pk rd st    C*
    100b4 : BNE a5, a0, 100a0                De Fu Al Rn Qu -- pk rd ex wr             Ca Cb
=>  100a0 : LBU a4, 0(a2)                    De Fu Al Rn Qu -- -- pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8              De Fu Al Rn Qu -- -- -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                      De Fu Al Rn Qu pk rd ex wr                Ca Cb
    100ac : ADDI a5, a5, 1                      De Fu Al Rn Qu -- pk rd ex wr             Ca Cb
    100b0 : SB a4, -1(a5)                       De Fu Al Rn Qu -- -- pk rd ag ma mb mc    Ca Cb
            [store data]                                   Q* -- -- -- -- -- pk rd st    C*
    100b4 : BNE a5, a0, 100a0                   De Fu Al Rn Qu -- -- pk rd ex wr             Ca Cb
=>  100a0 : LBU a4, 0(a2)                          De Fu Al Rn Qu -- -- pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8                    De Fu Al Rn Qu -- -- -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                         De Fu Al Rn Qu -- pk rd ex wr                Ca Cb
    100ac : ADDI a5, a5, 1                         De Fu Al Rn Qu -- -- pk rd ex wr             Ca Cb
    100b0 : SB a4, -1(a5)                             De Fu Al Rn Qu -- -- pk rd ag ma mb mc    Ca Cb
            [store data]                                        Q* -- -- -- -- -- -- pk rd st C*
    100b4 : BNE a5, a0, 100a0                         De Fu Al Rn Qu -- -- pk rd ex wr             Ca Cb

Qu = Enqueue instruction/uop              Ca = Commit/Retire, stage A
Q* = Enqueue store-data uop               Cb = Commit/Retire, stage B
Al = Allocate regs+resources              C* = Commit/Retire store-data uop
Rn = Rename
De = Decode/determine dependencies
Fu = Fuse/Expand/Rewrite
```

Der er ikke meget at se, for eksemplet har ikke nogen instruktioner der kan fusioneres.

Overgang

Hermed slut på gennemgangen af hvordan vi kobler dataflow-maskinen til et maskinsprog med sekventiel semantik.

- Isolation / Skrive-kø / Reorder-buffer (ROB)
- Afbildning fra RISCV-registre til V-ID'er. Genbrug af V-ID'er
- Håndtering af fejlagtige forudsigelser og fejl
- Afkodning/omskrivning fra RISCV til dataflow operationer

Det er et ret generelt framework. Samme overordnede struktur selvom vi ændrer på hvilke "små" pipelines vi vil have internt i vores dataflow maskine

Kommentarer?

Vi mangler nu bare at få fyldt maskinen med de rette instruktioner. Hurtigt.

Forudsigelse af programforløb

Programforløbet forudsiges på basis af 3 lagerblokke allerforrest i pipelinen.

- BTB/CFG (branch target buffer) - lille cache der associerer PC med
  - Næste PC for et evt betinget hop
  - Næste PC hvis betinget hop ikke tages
  - PC-efter-kodeblok / Størrelse af kodeblok
  - Er der et betinget hop?
  - Er der kald?
  - Er der retur?
- RAS (return address stack)
- Forudsiger for betinget hop (beslutningshistorien)

Parallelt med at PC en sendes til instruktionscache for instruktionshentning, bruges den også til opslag i BTB og hop forudsiger. Hvis BTB'en siger at koden indeholder et betinget hop, så afgør hopforudsigeren om det skal tages eller ej. Hvis et hop ikke tages, så siger BTB'en om der er kald eller retur. Ved kald skubbes PC-efter-kodeblok på RAS, ved return tages næste PC fra RAS.

Når først det her kredsløb er "trænet" på basis af programmets opførsel, så kan det levere en ny PC for hvert "cycle"

Indkodning af CFG, Eksempel

```
Branch Target Buffer

entry   = 10074
next[0] = 100b0
next[1] = 10074
after   = 1009c
predict, call




entry   = 1009c
next[0] = xxx
next[1] = 10074
after   = 100ac
no predict, call

entry   = 100ac
next[0] = xxx
next[1] = xxx
after = xxx
no predict, ret
```

```
entry = 100b0
next[0] = xxx
next[1] = xxx
after = xxx
no predict, ret
```

```
Instruction Cache

 10074: addi    sp,sp,-16
 10078: sw      ra,12(sp)
 1007c: sw      s0,8(sp)
 10080: sw      s1,4(sp)
 10084: mv      s0,a0
 10088: li      a5,1
 1008c: bgeu    a5,a0,100b0 <fib+0x3c>
 10090: addi    a0,a0,-1
 10094: auipc   ra,0x0
 10098: jalr    -32(ra) # 10074 <fib>

 1009c: mv      s1,a0
 100a0: addi    a0,s0,-2
 100a4: auipc   ra,0x0
 100a8: jalr    -48(ra) # 10074 <fib>

 100ac: add     a0,s1,a0

 100b0: lw      ra,12(sp)
 100b4: lw      s0,8(sp)
 100b8: lw      s1,4(sp)
 100bc: addi    sp,sp,16
 100c0: ret
```

Dynamisk hop-forudsigelse (fra onsdag)

Dynamisk hop-forudsigelse opsamler data fra hoppenes historie og bruger det til at forudsige den fremtidige opførsel.

Den simpleste udgave betragter hvert hop for sig. Man knytter en to-bit tæller til hver hop, i praksis ved at lave en række af tællere og bruge nogle bits fra PC'en til at vælge en tæller. Hver tæller opsumerer hoppets historie:

```
00 hop ikke taget (strongly not taken)
01 hop almindeligvis ikke taget (weakly not taken)
10 hop almindeligvis taget (weakly taken)
11 hop taget (strongly taken
```

Hver gang et hop afgøres opdateres den matchende tæller, enten i retning mod "hop ikke taget", eller mod "hop taget".

Dette kaldes "local" hop forudsigelse - fordi man betragter hvert betinget hop adskilt fra de andre.

Korrelerende hop-forudsigelse

Hop er ofte korrelerede med andre hop. Det kan man udnytte ved at opsamle hoppenes historie. En simpel fremgangsmåde er at indkode historien i et skifte-register. Når et hop tages skifter man '1' ind i skifteregisteret. Når et hop ikke tages skifter man '0' ind.

Som før har man en tabel af to-bit tællere der opdateres på samme måde som beskrevet for lokale forudsigere.

For at lave en forudsigelse laver man et "hash" af skifteregisteret og PC'en og bruger det til at slå op i tabellen med tællere. Bitvis XOR er en fin hash funktion i det her tilfælde.

Denne forudsiger kaldes "gshare" og kan ofte levere mere end 90% korrekte forudsigelser.

Der findes andre og betydeligt mere omfattende forudsigere der fungerer endnu bedre. Generelt skal man ikke tro at man kan forudsige sine egne hop bedre end maskinen kan.

Se f.eks. https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/

Forudsigelse - Præcision vs Hastighed

Det kræver en hvis mængde lager til BTB og hopforudsiger for at få høj præcision. Desværre er store lager elementer langsommere end små, og høj præcision kan ikke fås i løbet af en enkelt clock cyklus.

Det er derfor normalt at man laver to forudsigelseskredsløb a la det tidligere beskrevne.

- Et der er optimeret til at kunne levere en forudsigelse pr clock. Dette kredsløb giver ikke så præcise forudsigelser (~80-85%).
- Et der er væsentligt større og optimeret til at være så præcist som budgettet tillader (~95-98%), men som er langsommere.

Det store kredsløb bruges så til at korrigere forudsigelserne fra det lille hurtige kredsløb.

Instruktions-hentning

Det vil vi ikke gå i detaljer med. Blot to pointer:

Det er mest et spørgsmål om at have en meget bred vej fra instruktions-cache til næste trin (4-8 instruktioner = 16-32 bytes)

Det kan være nødvendigt for ydeevnen, at have to porte til instruktions-cachen, således at man kan hente instruktioner fra to fortløbende cache-blokke samtidigt. Ikke fordi man har brug for så mange instruktioner på en gang, men fordi man har brug for at hente instruktionssekvenser der går fra sidst i en cache-blok ind i begyndelsen af næste cache-blok.

Som tidligere nævnt i forbindelse med realistiske pipelines så antager vi at instruktionshentning gøres i tre pipeline trin, Fa, Fb og Fc uden at gå i detaljer med hvad der sker i de enkelte trin.

Komplet afviklingsdiagram

Her ses (endeligt) alle pipeline trin i en out-of-order maskine, der udfører vores programsekvens:

```
    RISCV-Instruction stream:             Schedule on small realistic machine


    100a0 : LBU a4, 0(a2)                 Fa Fb Fc De Fu Al Rn Qu pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8           Fa Fb Fc De Fu Al Rn Qu -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                Fa Fb Fc De Fu Al Rn Qu pk rd ex wr             Ca Cb
    100ac : ADDI a5, a5, 1                Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wr          Ca Cb
    100b0 : SB a4, -1(a5)                    Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc Ca Cb
            [store data]                                        Q* -- -- -- pk rd st    C*
    100b4 : BNE a5, a0, 100a0                Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wr             Ca Cb
=>  100a0 : LBU a4, 0(a2)                       Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8                 Fa Fb Fc De Fu Al Rn Qu -- -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                      Fa Fb Fc De Fu Al Rn Qu pk rd ex wr                Ca Cb
    100ac : ADDI a5, a5, 1                      Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wr             Ca Cb
    100b0 : SB a4, -1(a5)                          Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc    Ca Cb
            [store data]                                                Q* -- -- -- -- pk rd st    C*
    100b4 : BNE a5, a0, 100a0                      Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wr             Ca Cb
=>  100a0 : LBU a4, 0(a2)                             Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc wr Ca Cb
    100a4 : BEQ a4, zero, 100b8                       Fa Fb Fc De Fu Al Rn Qu -- -- -- -- -- pk rd ex wr Ca Cb
    100a8 : ADDI a2, a2, 1                            Fa Fb Fc De Fu Al Rn Qu pk rd ex wr                Ca Cb
    100ac : ADDI a5, a5, 1                            Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wr             Ca Cb
    100b0 : SB a4, -1(a5)                                Fa Fb Fc De Fu Al Rn Qu -- pk rd ag ma mb mc    Ca Cb
            [store data]                                                        Q* -- -- -- -- -- pk rd st C*
    100b4 : BNE a5, a0, 100a0                            Fa Fb Fc De Fu Al Rn Qu -- pk rd ex wr             Ca Cb

Qu = Enqueue instruction/uop              Ca = Commit/Retire, stage A
Q* = Enqueue store-data uop               Cb = Commit/Retire, stage B
Al = Allocate regs+resources              C* = Commit/Retire store-data uop
Rn = Rename                               Fa = Fetch, stage A
De = Decode/determine dependencies        Fb = Fetch, stage B
Fu = Fuse/Expand/Rewrite                  Fc = Fetch, stage C
```

OOO - Opsamling

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

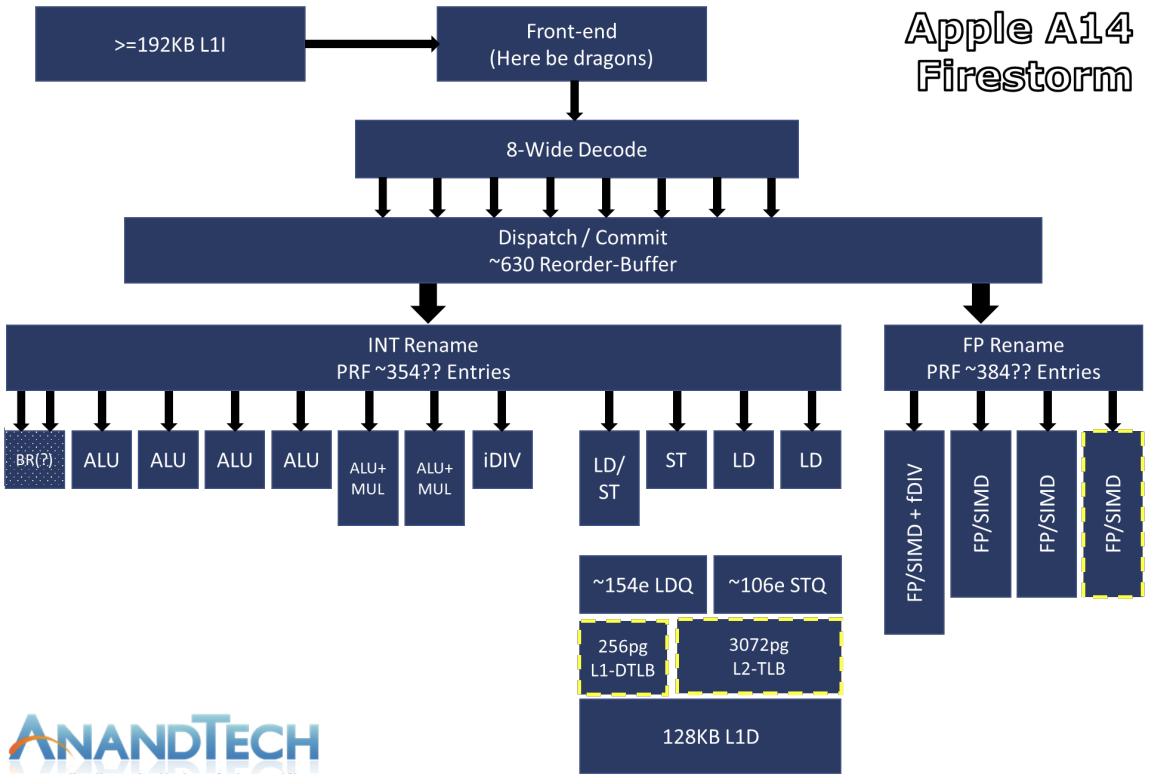
1. Udførelsesrækkefølge fastlægges ud fra afhængigheder mellem instruktioner, ikke deres sekventielle rækkefølge i programmet
2. Spekulativ udførelse: Instruktioner udføres aggressivt i en "sandkasse", alle resultater/side-effekter skjules for omverdenen. "What happens in Vegas stays in Vegas"
3. Forudsigelse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

Alle pipeline-trin forkortelser:

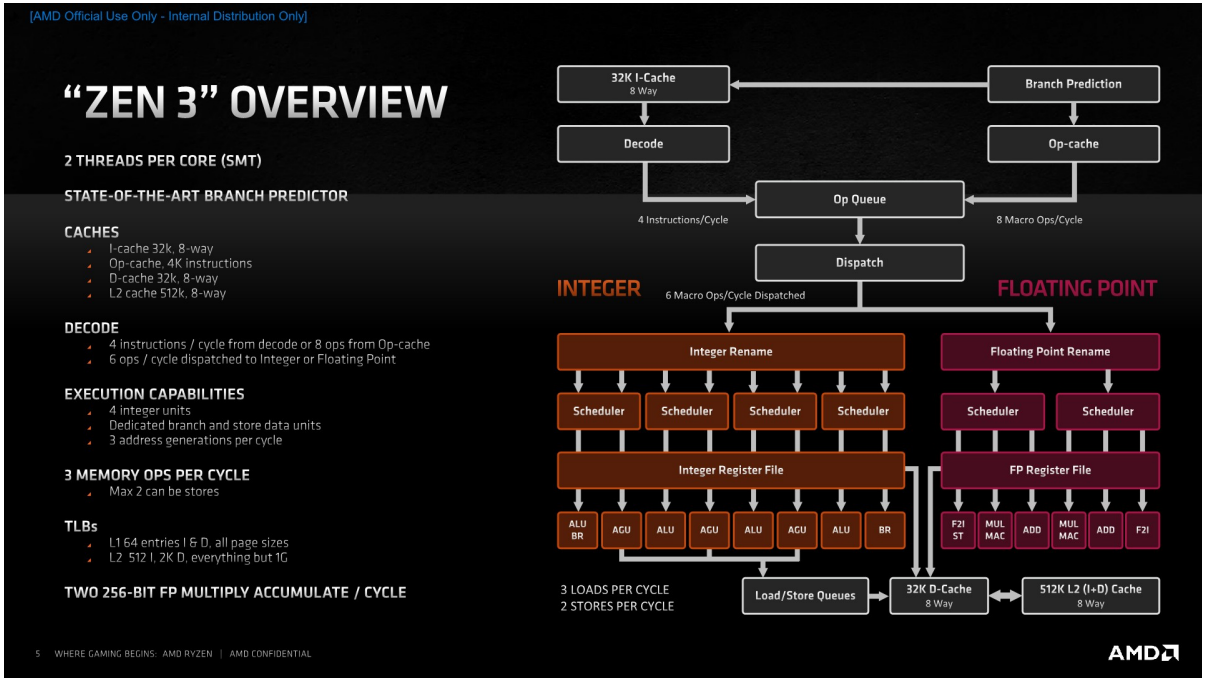
```
Inorder:                                  OutOfOrder/Dataflow
Fa  Fetch A                 pk  Pick (instruction)
Fb  Fetch B                 rd  Read (data)
Fc  Fetch C                 ex  Execute (arithmetic,control-flow)
De  Decode                  wb  Writeback (same as wr)
Fu  Fusion                  ag  Address Generate                    m0  Multiply 0
Al  Allocate                ma  Data cache A                        m1  Multiply 1
Re  Rename                  mb  Data cache B                        m2  Multiply 2
Qu  Queue                   mc  Data cache C                        m3  Multiply 3
Ca  Commit A                Q*  Queue store-data
Cb  Commit B                C*  Commit store-data

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 ag ma mb mc 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
```

En rigtig ARM (Apple M1) - rekonstrueret

Samme overordnede princip som gennemgået - men voldsomt flere resourcer!



En rigtig x86 (AMD Zen 3) - overblik



bemærk op-cache! her caches oversatte instruktioner så den tunge afkodning af AMD64 instruktioner kan minimeres.

Spørgsmål og Svar