Udgangspunkt

I sidste forelæsning tog vi udgangspunkt i sekventiel udførelse af et program og udviklede mekanismer derfra som kunne lave mere arbejde i parallel (pipelining, superscalar)

Og ramte så alskens forhindringer i form af data afhængigheder og kontrol-afhængigheder

Nu vender vi bøtten på hovedet:

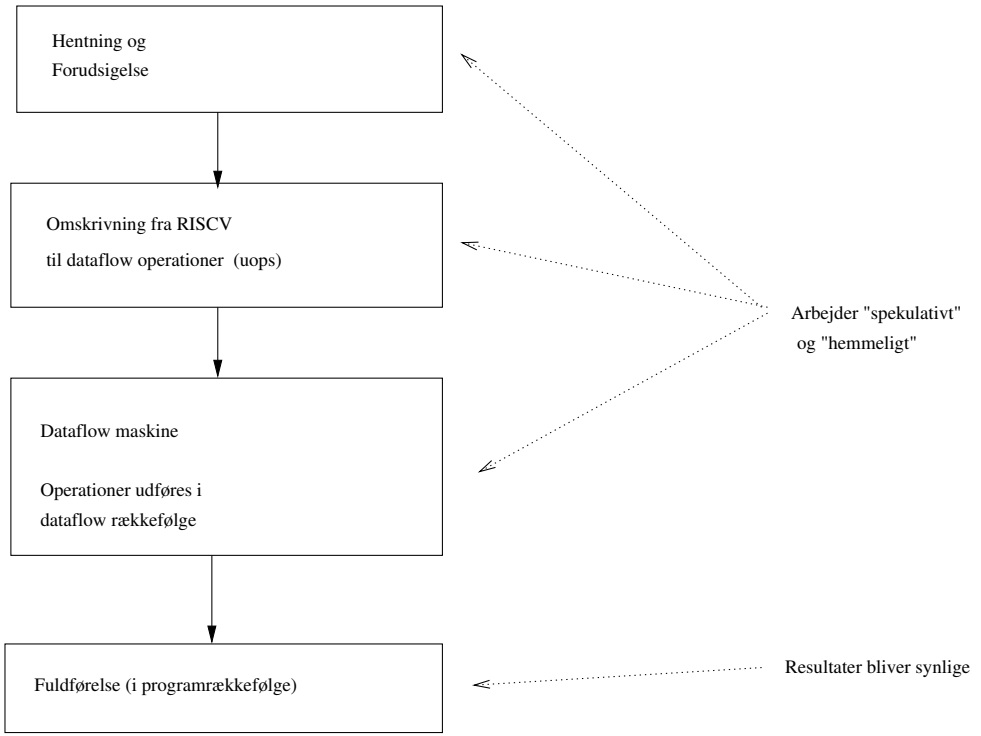
Vi tager udgangspunkt i en maskine der fra starten udfører ordrer så hurtigt som data afhængigheder tillader det .... og som (indtil videre) ignorerer kontrol- afhængigheder.

Vi udvikler en *dataflow-maskine*

Derpå tilpasser vi den til udførelse af et maskinsprog med sekventiel semantik.

Dataflow - Oversigt

Vi skal bruge en dataflow maskine som en del af en større plan!



Dataflow - Agenda

- Et programeksempel
- Dataflow udførelse og maskine
- Spekulativ udførelse
- Dataflow for operationer med variabel latenstid
- Parallel maskine vs Sekventiel semantik: Lagerbårne afhængigheder

Et gennemgående program-eksempel

Vi bruger en simpel lille stump kode til at illustrere pointerne

Min egen lille string copy funktion:

```
void stringcopy(char* to, int max, const char* from) {
  while (max && *from) {
    max--;
    *to++ = *from++;
  }
}
```

Den indre løkke i RISCV assembler:

```
loop :   LBU a4, 0(a2)
         BEQ a4, zero, loop_exit
         ADDI a2, a2, 1
         ADDI a5, a5, 1
         SB a4, -1(a5)
         BNE a5, a0, loop
loop_exit:
```

Fra RISCV til Dataflow (I)

RISCV-Instruction stream:	Dataflow Operation: (with split Stores)	Register nr to phys reg - mapping: (the '*' marks a change)
		a0 = v1, a2 = v2, a4 = ??, a5 = v3
100a0 : LBU a4, 0(a2)	v4 = LBU(v2,0)	a0 = v1, a2 = v2, *a4 = v4, a5 = v3
100a4 : BEQ a4, zero, 100b8	BEQ(v4,v0,100b8)	a0 = v1, a2 = v2, a4 = v4, a5 = v3
100a8 : ADDI a2, a2, 1	v5 = ADDI(v2,1)	a0 = v1, *a2 = v5, a4 = v4, a5 = v3
100ac : ADDI a5, a5, 1	v6 = ADDI(v3,1)	a0 = v1, a2 = v5, a4 = v4, *a5 = v6
100b0 : SB a4, -1(a5)	SB_A(v6,-1)	a0 = v1, a2 = v5, a4 = v4, a5 = v6
	SB_D(v4)	a0 = v1, a2 = v5, a4 = v4, a5 = v6
100b4 : BNE a5, a0, 100a0	BNE(v6,v1,100a0)	a0 = v1, a2 = v5, a4 = v4, a5 = v6
100a0 : LBU a4, 0(a2)	v7 = LBU(v5,0)	a0 = v1, a2 = v5, *a4 = v7, a5 = v6
100a4 : BEQ a4, zero, 100b8	BEQ(v7,v0,100b8)	a0 = v1, a2 = v5, a4 = v7, a5 = v6
100a8 : ADDI a2, a2, 1	v8 = ADDI(v5,1)	a0 = v1, *a2 = v8, a4 = v7, a5 = v6
100ac : ADDI a5, a5, 1	v9 = ADDI(v6,1)	a0 = v1, a2 = v8, a4 = v7, *a5 = v9
100b0 : SB a4, -1(a5)	SB_A(v9,-1)	a0 = v1, a2 = v8, a4 = v7, a5 = v9
	SB_D(v7)	a0 = v1, a2 = v8, a4 = v7, a5 = v9
100b4 : BNE a5, a0, 100a0	BNE(v9,v1,100a0)	a0 = v1, a2 = v8, a4 = v7, a5 = v9
100a0 : LBU a4, 0(a2)	v10 = LBU(v8,0)	a0 = v1, a2 = v8, *a4 = v10, a5 = v9
100a4 : BEQ a4, zero, 100b8	BEQ(v10,v0,100b8)	a0 = v1, a2 = v8, a4 = v10, a5 = v9
100a8 : ADDI a2, a2, 1	v11 = ADDI(v8,1)	a0 = v1, *a2 = v11, a4 = v10, a5 = v9
100ac : ADDI a5, a5, 1	v12 = ADDI(v9,1)	a0 = v1, a2 = v11, a4 = v10, *a5 = v12
100b0 : SB a4, -1(a5)	SB_A(v12,-1)	a0 = v1, a2 = v11, a4 = v10, a5 = v12
	SB_D(v10)	a0 = v1, a2 = v11, a4 = v10, a5 = v12
100b4 : BNE a5, a0, 100a0	BNE(v12,v1,100a0)	a0 = v1, a2 = v11, a4 = v10, a5 = v12

Dataflow operationerne:

- Ligner til forveksling tilsvarende RISCV instruktioner -- undtagen STORE
- STORE instruktioner er splittet i TO dataflow operationer, en for data og en for adresse-håndtering (her bliver SB til SB_A og SB_D).
- Denne opslitning gør det muligt at lave adresseberegninger tidligere. (Vigtigt for ydeevne - forklaring følger senere)

Fra RISCV til Dataflow (II)

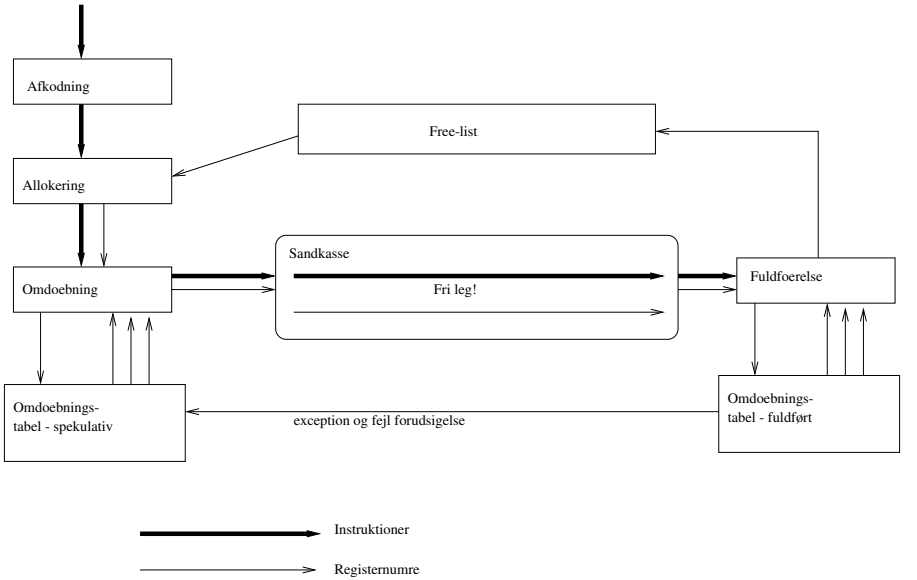
```
RISCV-Instruction stream:          Dataflow Operation:          Register nr to phys reg - mapping:
                                   (with split Stores)          (the '*' marks a change)

                                                                a0 = v1,  a2 = v2,   a4 = ??,   a5 = v3
100a0 : LBU a4, 0(a2)              v4 =  LBU(v2,0)              a0 = v1,  a2 = v2,  *a4 = v4,   a5 = v3
100a4 : BEQ a4, zero, 100b8              BEQ(v4,v0,100b8)       a0 = v1,  a2 = v2,   a4 = v4,   a5 = v3
100a8 : ADDI a2, a2, 1             v5 =  ADDI(v2,1)             a0 = v1, *a2 = v5,   a4 = v4,   a5 = v3
100ac : ADDI a5, a5, 1             v6 =  ADDI(v3,1)             a0 = v1,  a2 = v5,   a4 = v4,  *a5 = v6
100b0 : SB a4, -1(a5)                    SB_A(v6,-1)            a0 = v1,  a2 = v5,   a4 = v4,   a5 = v6
                                         SB_D(v4)               a0 = v1,  a2 = v5,   a4 = v4,   a5 = v6
100b4 : BNE a5, a0, 100a0                BNE(v6,v1,100a0)       a0 = v1,  a2 = v5,   a4 = v4,   a5 = v6
100a0 : LBU a4, 0(a2)              v7 =  LBU(v5,0)              a0 = v1,  a2 = v5,  *a4 = v7,   a5 = v6
100a4 : BEQ a4, zero, 100b8              BEQ(v7,v0,100b8)       a0 = v1,  a2 = v5,   a4 = v7,   a5 = v6
100a8 : ADDI a2, a2, 1             v8 =  ADDI(v5,1)             a0 = v1, *a2 = v8,   a4 = v7,   a5 = v6
100ac : ADDI a5, a5, 1             v9 =  ADDI(v6,1)             a0 = v1,  a2 = v8,   a4 = v7,  *a5 = v9
100b0 : SB a4, -1(a5)                    SB_A(v9,-1)            a0 = v1,  a2 = v8,   a4 = v7,   a5 = v9
                                         SB_D(v7)               a0 = v1,  a2 = v8,   a4 = v7,   a5 = v9
100b4 : BNE a5, a0, 100a0                BNE(v9,v1,100a0)       a0 = v1,  a2 = v8,   a4 = v7,   a5 = v9
100a0 : LBU a4, 0(a2)              v10 = LBU(v8,0)              a0 = v1,  a2 = v8,  *a4 = v10,  a5 = v9
100a4 : BEQ a4, zero, 100b8              BEQ(v10,v0,100b8)      a0 = v1,  a2 = v8,   a4 = v10,  a5 = v9
100a8 : ADDI a2, a2, 1             v11 = ADDI(v8,1)             a0 = v1, *a2 = v11,  a4 = v10,  a5 = v9
100ac : ADDI a5, a5, 1             v12 = ADDI(v9,1)             a0 = v1,  a2 = v11,  a4 = v10, *a5 = v12
100b0 : SB a4, -1(a5)                    SB_A(v12,-1)           a0 = v1,  a2 = v11,  a4 = v10,  a5 = v12
                                         SB_D(v10)              a0 = v1,  a2 = v11,  a4 = v10,  a5 = v12
100b4 : BNE a5, a0, 100a0                BNE(v12,v1,100a0)      a0 = v1,  a2 = v11,  a4 = v10,  a5 = v12
```

Dataflow operationerne:

- Bruger et unikt nummer til at identificere hver afhængighed (her v1-v12).
- Og et unikt register til at holde hver værdi. Disse registre benævnes "fysiske registre".
- Til højre ses sammenhængen mellem register på maskinsprogsniveau og fysisk register, for hver instruktion.

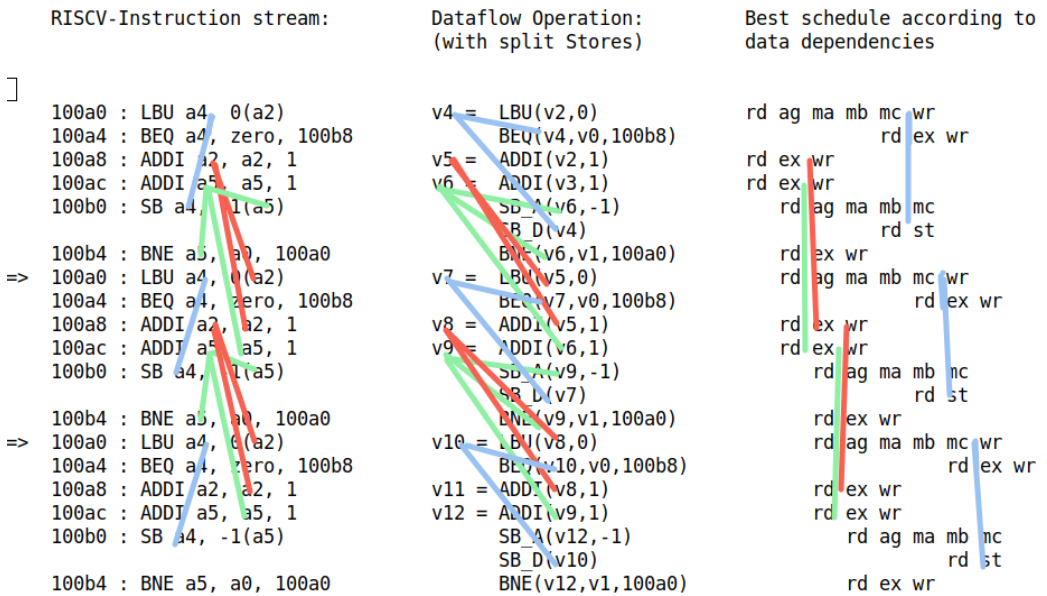
Flow af fysiske register numre



- Hver instruktion får "omdøbt" register numre for kilderegistre
- Hver instruktion får ny unik fysisk register for resultatregister
- Fyiske registre allokeres fra friliste
- Tidligere fysisk register for resultatregister frigives ved fuldførelse
- De grumme detaljer følger ved næste forelæsning

Dataflow udførelse

Dataafhængighederne i vort lille programeksempel tillader en hel iteration pr clock for vores kode-eksempel. (Så det er maximum for en uendelig avanceret maskine!)



dataafhængigheder: blå = a4, grøn = a5, rød = a2.

ex=execute, ag=address generate, ma-mc=memory access, rd=read operands, wr=write result

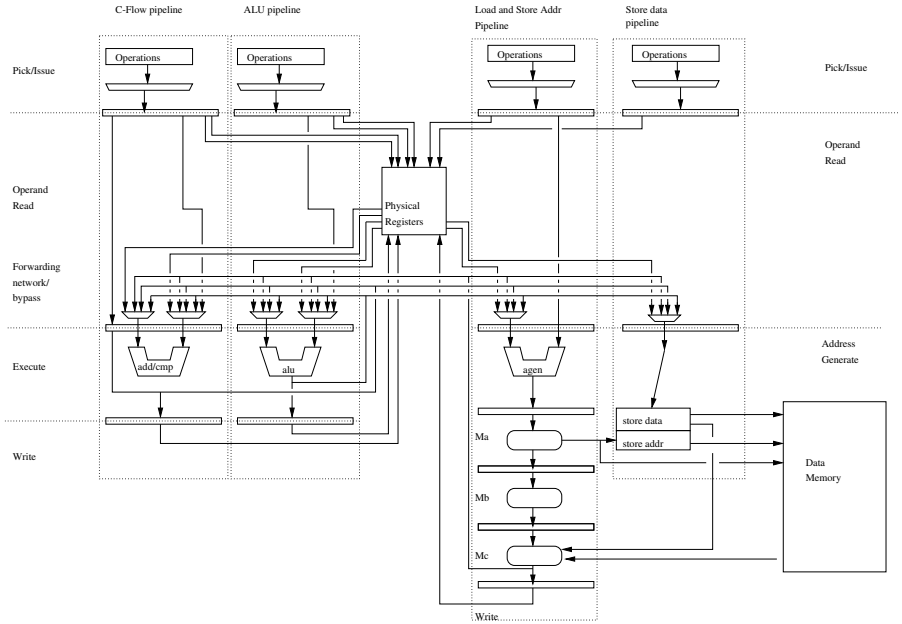
Dataflow udførelse (II)

Da RISCV instruktioner og dataflow operationer er så tæt på hinanden vil vi herfra tillade os blot at vise RISCV instruktioner som om de var dataflow operationer. Dog med en ekstra linie til at beskrive udførelsen af store-data operationen. Vi viser heller ikke brugen af fysiske registre i stedet for logiske (programmørsynlige)

```
   RISCV-Instruction stream:              Best schedule according to
                                          data dependencies


   100a0 : LBU a4, 0(a2)                  rd ag ma mb mc wr
   100a4 : BEQ a4, zero, 100b8                       rd ex wr
   100a8 : ADDI a2, a2, 1                 rd ex wr
   100ac : ADDI a5, a5, 1                 rd ex wr
   100b0 : SB a4, -1(a5)                     rd ag ma mb mc
           [store data]                                rd st
   100b4 : BNE a5, a0, 100a0                 rd ex wr
=> 100a0 : LBU a4, 0(a2)                     rd ag ma mb mc wr
   100a4 : BEQ a4, zero, 100b8                          rd ex wr
   100a8 : ADDI a2, a2, 1                    rd ex wr
   100ac : ADDI a5, a5, 1                    rd ex wr
   100b0 : SB a4, -1(a5)                        rd ag ma mb mc
           [store data]                                   rd st
   100b4 : BNE a5, a0, 100a0                    rd ex wr
=> 100a0 : LBU a4, 0(a2)                        rd ag ma mb mc wr
   100a4 : BEQ a4, zero, 100b8                             rd ex wr
   100a8 : ADDI a2, a2, 1                       rd ex wr
   100ac : ADDI a5, a5, 1                       rd ex wr
   100b0 : SB a4, -1(a5)                           rd ag ma mb mc
           [store data]                                      rd st
   100b4 : BNE a5, a0, 100a0                       rd ex wr
```

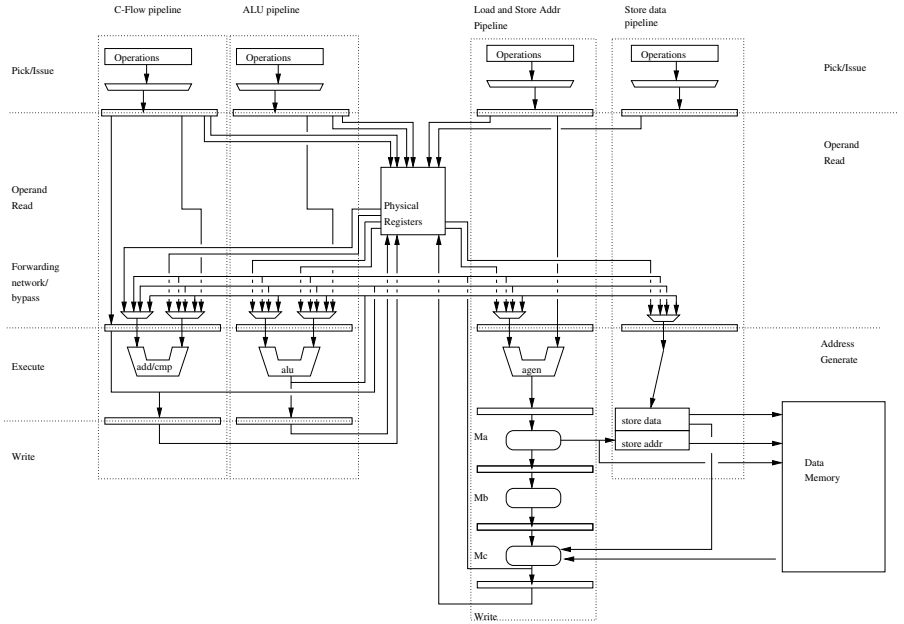
En lille dataflow maskine



Fire pipelines (fra venstre mod højre):

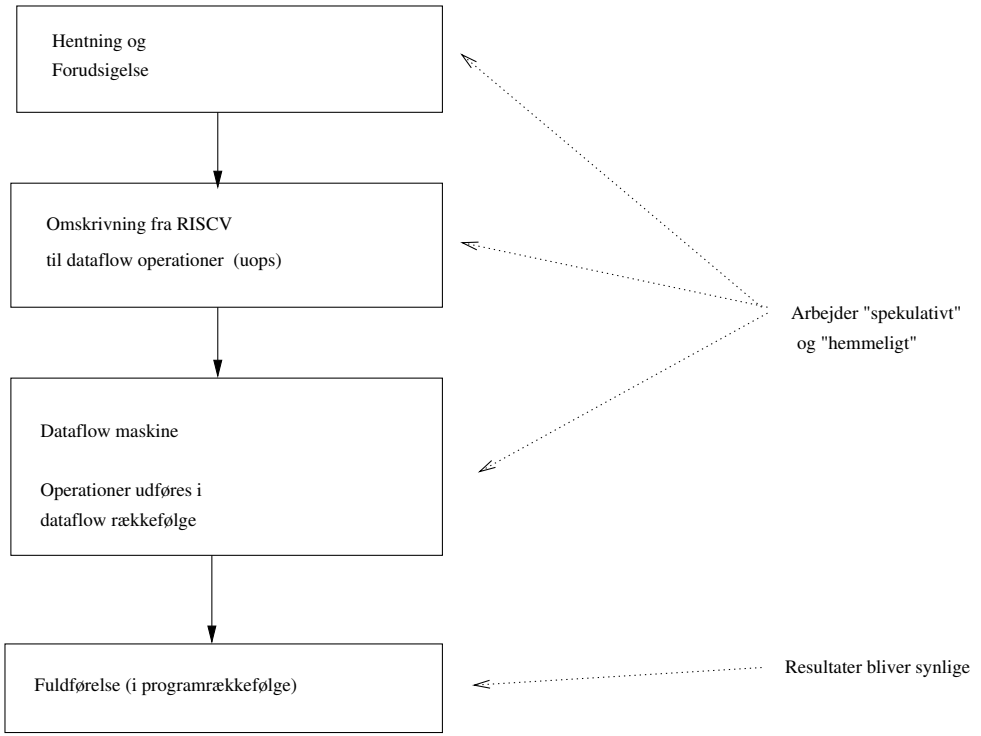
- Control-flow
- Aritmetik
- Load og Store-adresse
- Store data

En dataflow maskine (II)



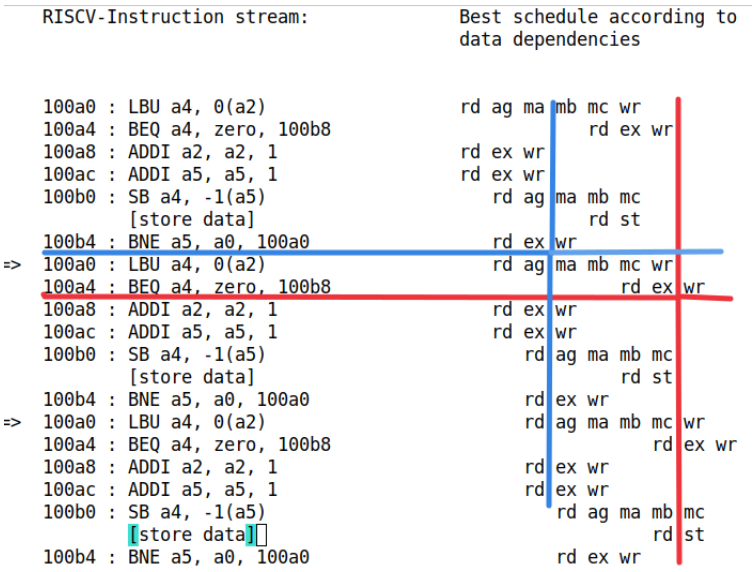
- I midten et lager til værdier - de fysiske registre
- Et forwarding netværk/bypass - vi har fuld forwarding
- En store-kø... her gemmes skrivninger til lageret indtil de må gennemføres (forklaring følger)
- Øverst i hver pipeline: et kredsløb til at udvælge en instruktion til udførelse (aka pick/issue)

Store-kø - hvorfor?



Vor dataflow-maskine arbejder "spekulativt" ... effekten af dens arbejde skal først gøres synlig, når vi ved at det er OK. Derfor må skrivninger til lageret placeres i en kø, indtil fuldførelse.

Store-kø - hvorfor? (II)



- Spekulativ udførelse -- ABSOLUT nødvendigt
- Skrivninger til lageret skal "holdes tilbage" til de ikke er spekulative længere.
- Læsninger skal se tidligere skrivninger - selvom de er holdt tilbage!
- Vi har brug for en søgbar kø til udestående skrivninger. STORE-køen.

Pipeline trin - opsamling

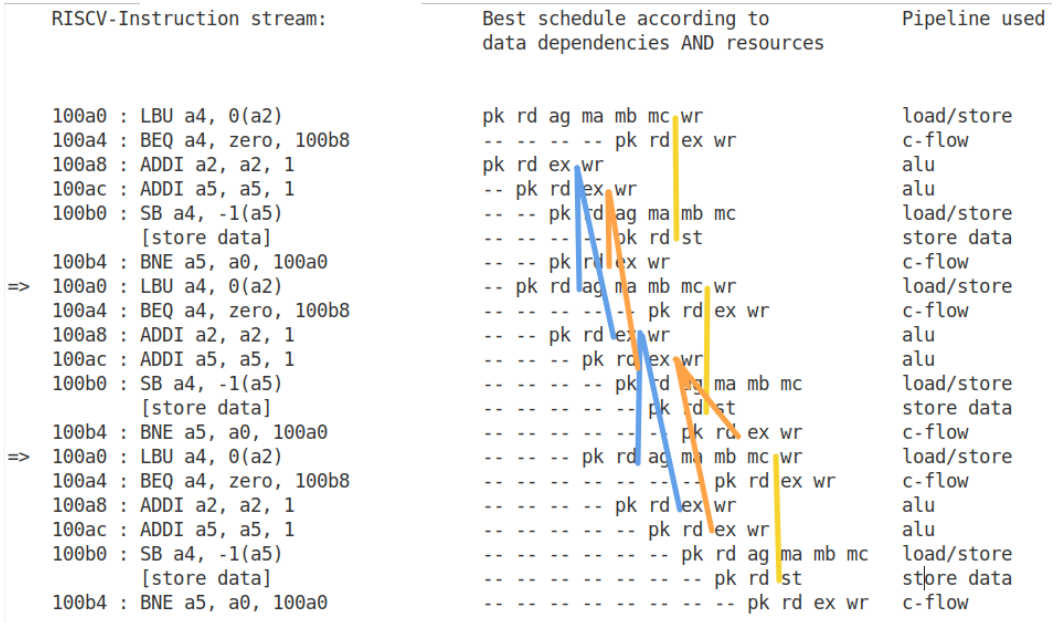
Vores lille dataflow-maskine med 4 pipelines har følgende trin/aktiviteter

```
rd = read (register/operand)
ex = execute (aritmetisk operation, hop/kald/retur)
wr = skriv (register/resultat)
ag = generer adresse (load/store)
st = skriv store data til kø
ma = trin 1 af data lager tilgang
mb = trin 2 af data lager tilgang
mc = trin 3 af data lager tilgang
pk = pick/udvælg operation (engelsk: "issue")
-- = operation afventer at blive valgt til udførelse
```

Virkelige maskiner følger samme princip men har væsentlig flere pipelines.

Dataflow udførelse

Sådan her ser afviklingsplottet ud for vor dataflow maskine. (max 1 load/store, max 1 aritmetisk op, max 1 control-flow op per cycle)



Hver iteration tager 2 cycles. Med 6 instruktioner per iteration får vi IPC = 3.

Dataflow issue logic

Hvordan vælger vi hvilke instruktioner vi skal udføre?

En mulig løsning er en (slags) "kø":

- Operationer indsættes i den ene ende
- Operationer står i indsættelsesrækkefølge
- Hver clock periode undersøger alle operationer (parallelt) om de er parat
- Et prioriteringskredsløb udvælger den ældste blandt de parate
- Den udvalgte operation tages ud af køen

Hvordan ved man, hvornår en operation er parat?

En operation er parat, når dens indgående operander er tilgængelige. Hver operation i "køen" må kontinuerligt overvåge, hvilke operander der produceres.

Hvornår er operander tilgængelige?

En mulig implementation kan bero på en "parat-vektor". Den har en bit for hver unik værdi / fysisk register ID. Bit nr ID i vektoren er sat, hvis det fysiske register med nr ID er tilgængelig. Hver ventende instruktion har en tilsvarende "afhængigheds-vektor" hvor 0 angiver en afhængighed. Når bitvist OR af parat-vektor og afhængighedsvektor er lutter 1, er alle operander tilgængelige og operationen er parat.

Når en operation er udvalgt til videre udførelse sætter man den bit i parat-vektoren der svarer til operationens udgående fysiske register ID

Hvis den valgte operation har en effektiv latenstid på 1, tilføjer man den nye ID til paratvektoren, så den er med næste clock-periode. Hvis operationen har en latenstid på 4 (for eksempel load med cache hit), skal man først tilføje den nye ID 3 perioder senere.

Det giver noget kompleksitet at hele denne udvælgelsesprocess skal ske et par maskincykler FØR ordrerne udføres. Dvs parat-vektoren opdateres i forventning om hvad der sker et par maskincykler senere.

Eksempel

Clock periode 1:

```
Paratvektor: v0,v2
Kø (aritmetik):
  v3 = ADD(v0,v1)   ikke parat    --
  v4 = ADD(v3,v1)   ikke parat    --
  v5 = OR(v0,v2)    parat         pk
  v6 = SUB(v0,v5)   ikke parat    --
Kø (load)
  v1 = LOAD(v0)     parat         pk
  v7 = LOAD(v6)     ikke parat    --
```

Eksempel

Clock periode 2:

```
Paratvektor: v0,v2,v5
Kø (aritmetik):
  v3 = ADD(v0,v1)   ikke parat    -- --
  v4 = ADD(v3,v1)   ikke parat    -- --
  v5 = OR(v0,v2)    startet       pk rd
  v6 = SUB(v0,v5)   parat         -- pk
Kø (load)
  v1 = LOAD(v0)     startet       pk rd
  v7 = LOAD(v6)     ikke parat    -- --
```

Eksempel

Clock periode 3:

```
Paratvektor: v0,v2,v5,v6
Kø (aritmetik):
  v3 = ADD(v0,v1)   ikke parat    -- -- --
  v4 = ADD(v3,v1)   ikke parat    -- -- --
  v5 = OR(v0,v2)    startet       pk rd ex
  v6 = SUB(v0,v5)   startet       -- pk rd
Kø (load)
  v1 = LOAD(v0)     startet       pk rd ag
  v7 = LOAD(v6)     parat         -- -- pk
```

Eksempel

Clock periode 4:

```
Paratvektor: v0,v2,v5,v6
Kø (aritmetik):
  v3 = ADD(v0,v1)   ikke parat    -- -- -- --
  v4 = ADD(v3,v1)   ikke parat    -- -- -- --
  v5 = OR(v0,v2)    fuldført      pk rd ex wr
  v6 = SUB(v0,v5)   startet       -- pk rd ex
Kø (load)
  v1 = LOAD(v0)     startet       pk rd ag ma
  v7 = LOAD(v6)     startet       -- -- pk rd
```

Eksempel

Clock periode 5:

```
Paratvektor: v0,v1,v2,v5,v6
Kø (aritmetik):
  v3 = ADD(v0,v1)   parat         -- -- -- -- pk
  v4 = ADD(v3,v1)   ikke parat    -- -- -- -- --
  v5 = OR(v0,v2)    fuldført      pk rd ex wr
  v6 = SUB(v0,v5)   fuldført      -- pk rd ex wr
Kø (load)
  v1 = LOAD(v0)     startet       pk rd ag ma mb
  v7 = LOAD(v6)     startet       -- -- pk rd ag
```

Bemærk hvordan tilføjelse til paratvektoren skal ske 2 clock perioder FØR den tilhørende værdi bliver produceret. Det er nødvendigt på grund af forsinkelsen fra instruktioner udvælges, til de skal modtage deres indgående operander.

Eksempel

Clock periode 6:

```
Paratvektor: v0,v1,v2,v3,v5,v6
Kø (aritmetik):
  v3 = ADD(v0,v1)   startet       -- -- -- -- pk rd
  v4 = ADD(v3,v1)   parat         -- -- -- -- -- pk
  v5 = OR(v0,v2)    fuldført      pk rd ex wr
  v6 = SUB(v0,v5)   fuldført      -- pk rd ex wr
Kø (load)
  v1 = LOAD(v0)     startet       pk rd ag ma mb mc
  v7 = LOAD(v6)     startet       -- -- pk rd ag ma
```

Eksempel

Clock periode 7:

```
Paratvektor: v0,v1,v2,v3,v4,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   startet       -- -- -- -- pk rd ex
  v4 = ADD(v3,v1)   startet       -- -- -- -- -- pk rd
  v5 = OR(v0,v2)    fuldført      pk rd ex wr
  v6 = SUB(v0,v5)   fuldført      -- pk rd ex wr
Kø (load)
  v1 = LOAD(v0)     fuldført      pk rd ag ma mb mc wr
  v7 = LOAD(v6)     startet       -- -- pk rd ag ma mb
```

Eksempel

Clock periode 8

```
Paratvektor: v0,v1,v2,v3,v4,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   fuldført      -- -- -- -- pk rd ex wr
  v4 = ADD(v3,v1)   startet       -- -- -- -- -- pk rd ex
  v5 = OR(v0,v2)    fuldført      pk rd ex wr
  v6 = SUB(v0,v5)   fuldført      -- pk rd ex wr
Kø (load)
  v1 = LOAD(v0)     fuldført      pk rd ag ma mb mc wr
  v7 = LOAD(v6)     startet       -- -- pk rd ag ma mb mc
```

Eksempel

Clock periode 9

```
Paratvektor: v0,v1,v2,v3,v4,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   fuldført      -- -- -- -- pk rd ex wr
  v4 = ADD(v3,v1)   fuldført      -- -- -- -- -- pk rd ex wr
  v5 = OR(v0,v2)    fuldført      pk rd ex wr
  v6 = SUB(v0,v5)   fuldført      -- pk rd ex wr
Kø (load)
  v1 = LOAD(v0)     fuldført      pk rd ag ma mb mc wr
  v7 = LOAD(v6)     fuldført      -- -- pk rd ag ma mb mc wr
```

Variabel latenstid

Nogle instruktioner har variabel latenstid. Heraf er især LOAD instruktioner vigtige. LOAD instruktioner har oftest en latenstid på 4 cycles, men følgende hændelser kan give længere latenstider:

- Cache miss
- TLB miss
- Alias med tidligere STORE hvor data kommer for sent
- Ekstern aktivitet kan forsinke adgang til datacachen
- Tidligere STORE med ukendt adresse (Den ser vi lige bort fra og vender tilbage til)

Udvælgelse af instruktioner sker på basis af *antagelser* om instruktioners latenstid. Når disse antagelser ikke holder vil vi komme til at køre afhængige operationer med forkerte input

```
LBU x5,0(x4)     -- -- pk ag ma mb mc wr
ADDI x6,x5,1     -- -- -- -- -- pk rd ex wr
```

Værdien knyttet til X5 skal forwardes i slutningen af "mc" til slutningen af "rd" for den afhængige operation, men opdatering af paratvektoren for X5 sker allerede i slutningen af "ma".

Latenstid og scheduling

Følgende strategier er mulige, når man ikke kender latenstiden af en instruktion i tide (hit/miss status kendt i mb eller mc):

- Optimistisk (det er den strategi vi har fulgt hidtil):

```
LBU x5,0(x4)     -- -- pk ag ma mb mc wr
ADDI x6,x5,1     -- -- -- -- -- pk rd ex wr
```

- Pessimistisk (scheduler først afhængig operation, når vi er sikker på latenstiden)

```
LBU x5,0(x4)     -- -- pk ag ma mb mc wr
ADDI x6,x5,1     -- -- -- -- -- -- -- pk rd ex wr
```

Den optimistiske strategi giver LOAD instruktioner en effektiv latenstid på 4 cycles. Den pessimistiske strategi kræver 6 cycles.

Latenstid og schedulering (II)

Den optimistiske strategi vil producere fejl, som skal korrigeres. Det kan man gøre ved at forbedre schedulerings-kredsløbet således at det kan genkøre de sidste tre maskincykler med de ændrede forudsætninger, når de detekteres.

Den pesimistiske strategi undgår fejl og behøver ikke noget maskineri til at korrigere afviklingen. Men en omkostning på 2 ekstra cycles ved hver lagerreference er i almindelighed ikke acceptabel.

Hvilken strategi skal man vælge? Pessimistisk eller optimistisk? Optimistisk er vanskeligst at implementere men giver bedst ydeevne.

Almindeligvis implementeres den optimistiske strategi eller en hybrid der skifter mellem optimistisk og pessimistisk på basis af en forudsiger.

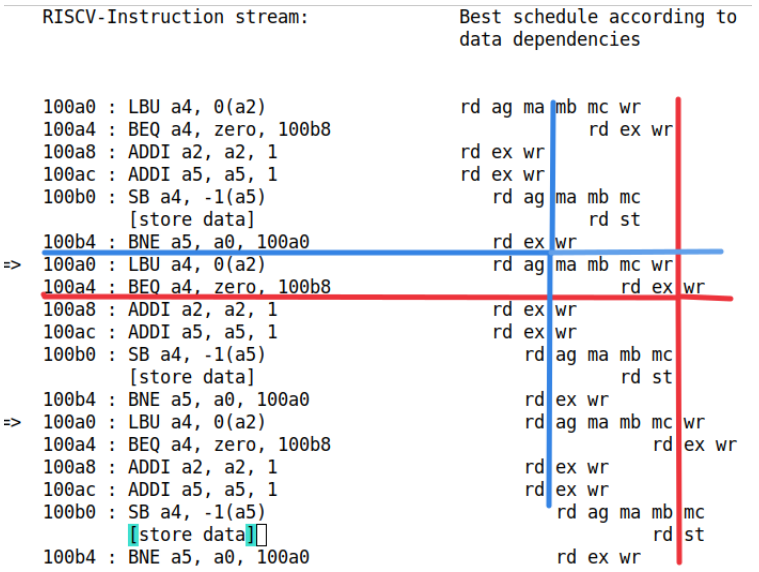
OBS: Vi stiller IKKE eksamensspørgsmål i "fejlagtig schedulering"

Tilpasning til sekventiel semantik

Vores dataflow maskine er en parallel maskine. Dens operationer har i udgangspunktet ingen rækkefølge. Vi vil nu tilpasse den, så den kan udføre instruktioner fra et maskinsprog med sekventiel semantik.

- Næste forelæsning: Instruktioner får en rækkefølge. Det vil blive muligt at stoppe udførelsen ved en instruktion, og alle tidligere instruktioner vil da være udført, og ingen efterfølgende instruktioner vil være udført.
- Denne forelæsning: Lager-referencer får en rækkefølge. En læsning fra lageret skal "se" en tidligere skrivning til samme addresse i lageret.

Spekulativ udførelse?



Her ses timing for hvornår vi kan afgøre om et hop skal tages eller ej

Instruktioner både før og efter er i varierende grad udført:

- Rød: Senere store er allerede udført - hvad hvis hoppet her er forudsagt forkert?
- Blå: Tidligere beq er ikke engang påbegyndt - hvad hvis DEN er forudsagt forkert?

Lagerbårne afhængigheder

Vi allokerer plads i STORE-køen i program-rækkefølge. De enkelte STORE operationer kan udføres i dataflow rækkefølge, men deres resultater (data og adresse) skal ligge i STORE-køen i program-rækkefølge.

En LOAD er nød til at gennemsøge "STORE-køen" for at "se" de tidligere skrivninger. Det gøres i parallel med læsning fra cache. Hvis der er en (eller flere) matchende skrivninger kan der så overføres data direkte fra disse til LOAD-instruktionen. Timing er det samme som et "hit" i primær cache.

Men hvad nu, hvis en LOAD ser en tidligere STORE, der endnu ikke har fået beregnet sin adresse?

- Skal man afvente at adressen beregnes? Hvilken omkostning har det at vente?
- Eller gøre antagelser om at den tidligere STORE nok er til en anden adresse?
- Hvis det sidste - hvad så, hvis man tager fejl?

Vent på STOREs adresseberegning

Dette er den nemmeste løsning at implementere. Man erstatter blot det komplicerede schedulerings-kredsløb for LOAD/STORE pipelinen med en ægte kø. Her ses vores programstump udsat for in-order adresseberegning.

```
    RISCV-Instruction stream:          Best schedule according to              Pipeline used
                                       data dependencies AND resources


    100a0 : LBU a4, 0(a2)              pk rd ag ma mb mc wr                    load/store
    100a4 : BEQ a4, zero, 100b8        -- -- -- -- pk rd ex wr                 c-flow
    100a8 : ADDI a2, a2, 1             pk rd ex wr                             alu
    100ac : ADDI a5, a5, 1             -- pk rd ex wr                          alu
    100b0 : SB a4, -1(a5)              -- -- pk rd ag ma mb mc                 load/store
            [store data]               -- -- -- -- pk rd st                    store data
    100b4 : BNE a5, a0, 100a0          -- -- pk rd ex wr                       c-flow
=>  100a0 : LBU a4, 0(a2)              -- -- -- pk rd ag ma mb mc wr           load/store
    100a4 : BEQ a4, zero, 100b8        -- -- -- -- -- -- -- pk rd ex wr        c-flow
    100a8 : ADDI a2, a2, 1             -- -- pk rd ex wr                       alu
    100ac : ADDI a5, a5, 1             -- -- -- pk rd ex wr                    alu
    100b0 : SB a4, -1(a5)              -- -- -- -- pk rd ag ma mb mc           load/store
            [store data]               -- -- -- -- -- -- -- -- pk rd st         store data
    100b4 : BNE a5, a0, 100a0          -- -- -- -- -- -- pk rd ex wr           c-flow
=>  100a0 : LBU a4, 0(a2)              -- -- -- -- -- -- pk rd ag ma mb mc wr  load/store
    100a4 : BEQ a4, zero, 100b8        -- -- -- -- -- -- -- -- -- pk rd ex wr  c-flow
    100a8 : ADDI a2, a2, 1             -- -- -- -- pk rd ex wr                 alu
    100ac : ADDI a5, a5, 1             -- -- -- -- -- pk rd ex wr              alu
    100b0 : SB a4, -1(a5)              -- -- -- -- -- -- pk rd ag ma mb mc     load/store
            [store data]               -- -- -- -- -- -- -- -- -- -- pk rd st  store data
    100b4 : BNE a5, a0, 100a0          -- -- -- -- -- -- -- -- pk rd ex wr     c-flow
```

For vores programeksempel ser vi en lille indledende forsinkelse, men derefter når vi -- måske en smule overraskende -- stadig 2 cycles per iteration (IPC=3).

Spekulativ håndtering af afhængigheder

For nyere og mere ambitiøse maskiner er det ikke godt nok.

Disse maskiner anvender alias-forudsigere. En alias-forudsiger styrer om individuelle LOAD instruktioner kun må startes efter tidligere STORE instruktioner, eller om de kan startes tidligere og spekulativt ignorere STORE instruktioner uden beregnet adresse.

Det er også nødvendigt at opbevare udførte LOAD instruktioner, således at en (logisk) tidligere STORE der får beregnet adressen senere, kan afgøre hvilke LOAD instruktioner, der har ignoreret en afhængighed.

Der kræves et væsentligt mere kompliceret maskineri til at rydde op når en overtrådt afhængighed detekteres.

Vi går ikke i yderligere detaljer :-)

Opsamling

Og indholdet var:

- Hvad er dataflow udførelse?
- Hvordan ser en dataflow mikroarkitektur ud?
- Hvordan udvælges operationer efter deres data afhængigheder?
- Hvordan håndterer vi dataflow med (visse) operationer med variabel latenstid
- Hvordan håndterer vi afhængigheder der går via lageret (STORE -> LOAD)

Hvad tænker i om det her?

Spørgsmål?

Spørgsmål og Svar