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*

Dataflow - Oversigt

- Et programeksempel
- Dataflow udførelse og maskine
- Programforløb og spekulativ udførelse
- Lagerbårne afhængigheder

Et gennemgående program-eksempel

Vi bruger en simpel lille stump kode til at illustrere pointerne

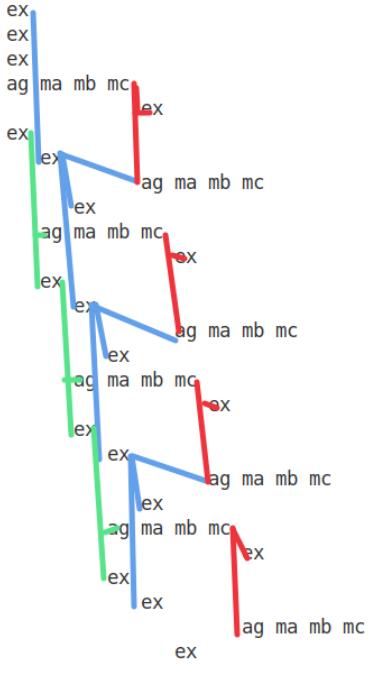
Min egen lille string copy funktion:

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

```
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:
```

Dataflow udførelse

```
ADDI a5, a0, 0                  R[15] <- ffffd4
ADD a0, a0, a1                  R[10] <- ffffde
BEQ a1, zero, 100bc         {_}
LBU a4, 0(a2)                   R[14] <- 62
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104ad
ADDI a5, a5, 1                  R[15] <- ffffd5
SB a4, -1(a5)                           62       ->  Mem[ffffd4]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 6c
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104ae
ADDI a5, a5, 1                  R[15] <- ffffd6
SB a4, -1(a5)                           6c       ->  Mem[ffffd5]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 61
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104af
ADDI a5, a5, 1                  R[15] <- ffffd7
SB a4, -1(a5)                           61       ->  Mem[ffffd6]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 62
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104b0
ADDI a5, a5, 1                  R[15] <- ffffd8
SB a4, -1(a5)                           62       ->  Mem[ffffd7]
BNE a5, a0, 100a0           {T}
```



blå = a5, grøn = a2, rød = a4.

Dataflow udførelse klarer her en iteration pr clock.

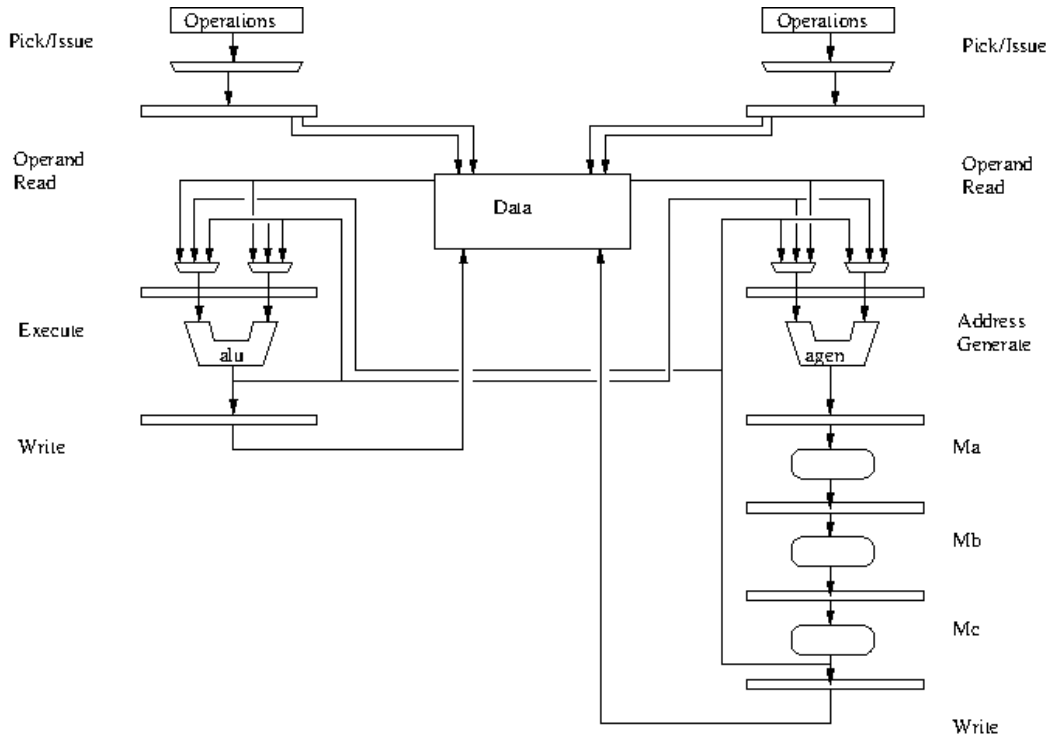
Dataflow udførelse II

Vi kan se vi behøver lagerplads til data. Så vi tilføjer læsning og skrivning til diagrammet:

```
ADDI a5, a0, 0              R[15] <- ffffd4                     rd ex wr
ADD a0, a0, a1              R[10] <- ffffde                     rd ex wr
BEQ a1, zero, 100bc     {_}                                     rd ex
LBU a4, 0(a2)               R[14] <- 62                         rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                           rd ex
ADDI a2, a2, 1              R[12] <- 104ad                      rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd5                        rd ex wr
SB a4, -1(a5)                       62       ->  Mem[ffffd4]             rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                           rd ex
LBU a4, 0(a2)               R[14] <- 6c                         rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                           rd ex
ADDI a2, a2, 1              R[12] <- 104ae                      rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd6                        rd ex wr
SB a4, -1(a5)                       6c       ->  Mem[ffffd5]             rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                           rd ex
LBU a4, 0(a2)               R[14] <- 61                         rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                           rd ex
ADDI a2, a2, 1              R[12] <- 104af                      rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd7                        rd ex wr
SB a4, -1(a5)                       61       ->  Mem[ffffd6]             rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                           rd ex wr
LBU a4, 0(a2)               R[14] <- 62                         rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                           rd ex
ADDI a2, a2, 1              R[12] <- 104b0                      rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd8                        rd ex wr
SB a4, -1(a5)                       62       ->  Mem[ffffd7]             rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                           rd ex wr
```

En dataflow maskine

Vi kalder maskinens instruktioner for "operationer"



Vi ser: To pipelines (en til aritmetik, en til lagertilgang), et dedikeret lager til værdier og en mekanisme til at udvælge hvilken operation der skal udføres i hver. Udvælgelsen har sit eget pipeline trin i toppen

Pipeline trin - opsamling

Vi har nu følgende trin/aktiviteter i vores maskine

```
rd = read (register/operand)
ex = execute (aritmetisk operation, hop/kald/retur)
wr = skriv (register/resultat)
ag = generer adresse
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")
```

Dataflow udførelse

Med det ekstra pipeline trine til udvælgelse af operationer ser afviklingsplottet sådan her ud:

```
ADDI a5, a0, 0              R[15] <- ffffd4                    pk rd ex wr
ADD a0, a0, a1              R[10] <- ffffde                    pk rd ex wr
BEQ a1, zero, 100bc     {_}                                    pk rd ex
LBU a4, 0(a2)               R[14] <- 62                        pk rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                       pk rd ex
ADDI a2, a2, 1              R[12] <- 104ad                     pk rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd5                       pk rd ex wr
SB a4, -1(a5)                       62       ->  Mem[ffffd4]             pk rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                          pk rd ex
LBU a4, 0(a2)               R[14] <- 6c                        pk rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                          pk rd ex
ADDI a2, a2, 1              R[12] <- 104ae                     pk rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd6                       pk rd ex wr
SB a4, -1(a5)                       6c       ->  Mem[ffffd5]             pk rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                             pk rd ex
LBU a4, 0(a2)               R[14] <- 61                        pk rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                             pk rd ex
ADDI a2, a2, 1              R[12] <- 104af                     pk rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd7                       pk rd ex wr
SB a4, -1(a5)                       61       ->  Mem[ffffd6]             pk rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                             pk rd ex wr
LBU a4, 0(a2)               R[14] <- 62                        pk rd ag ma mb mc wr
BEQ a4, zero, 100b8     {_}                                             pk rd ex
ADDI a2, a2, 1              R[12] <- 104b0                     pk rd ex wr
ADDI a5, a5, 1              R[15] <- ffffd8                       pk rd ex wr
SB a4, -1(a5)                       62       ->  Mem[ffffd7]             pk rd ag ma mb mc wr
BNE a5, a0, 100a0       {T}                                             pk rd ex wr
```

Dataflow "operationer"

Hvilke data skal indkodes i operationerne for at vi kan udvælge dem i dataflow rækkefølge?

- Hvad skal gøres? ADD/SUB/XOR/LD osv
- Indgående data ID
- Produceret data ID

Produceret data lagres i et lageområde der kan addresseres via ID.

Hvert enkelt data/værdi skal have en unik ID. Ellers går det galt.

Vor pick/issue logik skal så udvælge en instruktion til hver pipeline pr cycle

Dataflow issue logic

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 maskinen kan bestyre. Bit nr ID i vektoren er sat, hvis værdien identificeret med 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 værdi-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.

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

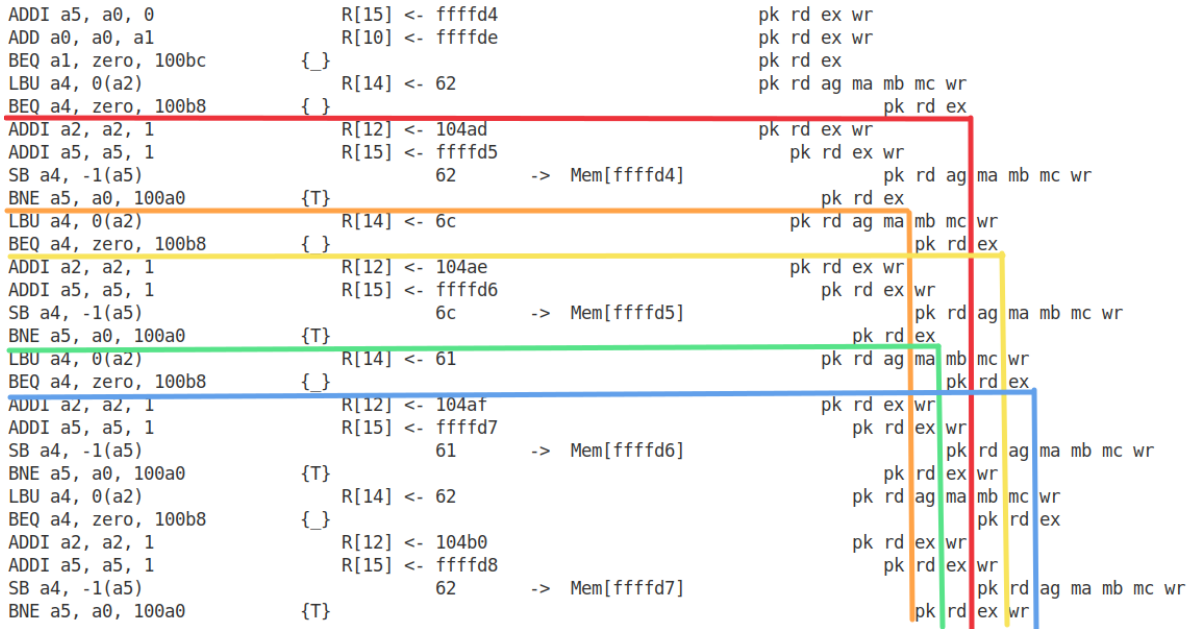
```
Paratvektor: v0,v1,v2,v3,v4,v5,v6
Kø (aritmetik):
  v3 = ADD(v0,v1)   fuldført      -- -- -- -- pk rd ex wr
  v4 = ADD(v3,v1)   startet       -- -- -- -- -- pk rd ex
  v5 = OR(v0,v2)    fuldført      pk rd ex wr -- -- -- --
  v6 = SUB(v0,v5)   fuldført      -- pk rd ex wr -- -- --
Kø (load)
  v1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

Programforløb og skrivning til lager



- spekulativ udførelse -- ABSOLUT nødvendigt
- dataflow og spekulativ udførelse - senere!
- skrivning til lageret og spekulativ udførelse - en udfordring!

Lagerbårne afhængigheder

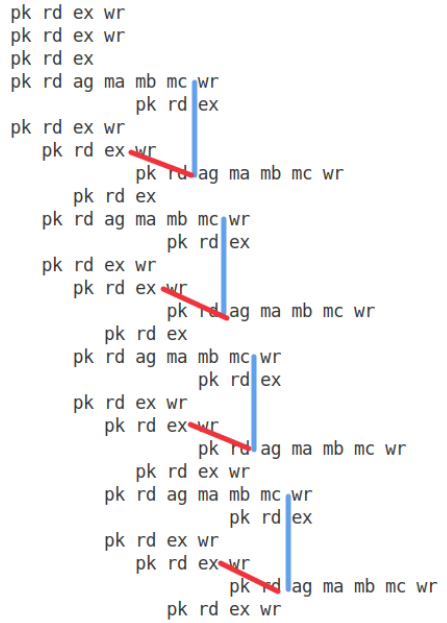
Betænk forskellen mellem

- strcpy(A, 1000, B)
- strcpy(A+1, 1000, A)

En LOAD skal se effekten af tidligere STORE operationer

Bestemmelse af lagerbårne afhængigheder

```
ADDI a5, a0, 0                  R[15] <- ffffd4
ADD a0, a0, a1                  R[10] <- ffffde
BEQ a1, zero, 100bc         {_}
LBU a4, 0(a2)                   R[14] <- 62
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104ad
ADDI a5, a5, 1                  R[15] <- ffffd5
SB a4, -1(a5)                           62       ->  Mem[ffffd4]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 6c
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104ae
ADDI a5, a5, 1                  R[15] <- ffffd6
SB a4, -1(a5)                           6c       ->  Mem[ffffd5]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 61
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104af
ADDI a5, a5, 1                  R[15] <- ffffd7
SB a4, -1(a5)                           61       ->  Mem[ffffd6]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 62
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104b0
ADDI a5, a5, 1                  R[15] <- ffffd8
SB a4, -1(a5)                           62       ->  Mem[ffffd7]
BNE a5, a0, 100a0           {T}
```

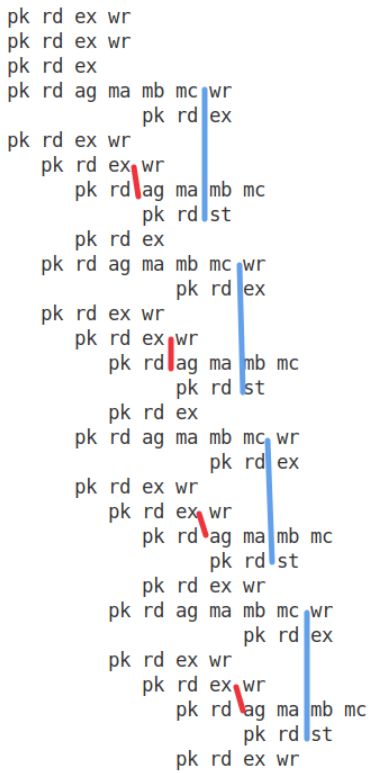


Vi vil gerne kunne fastlægge afhængigheder så tidligt som muligt. Det er svært, hvis STORE instruktioners adresseberegning skal vente på det data, der skal "stores"

Opsplitning af STORE

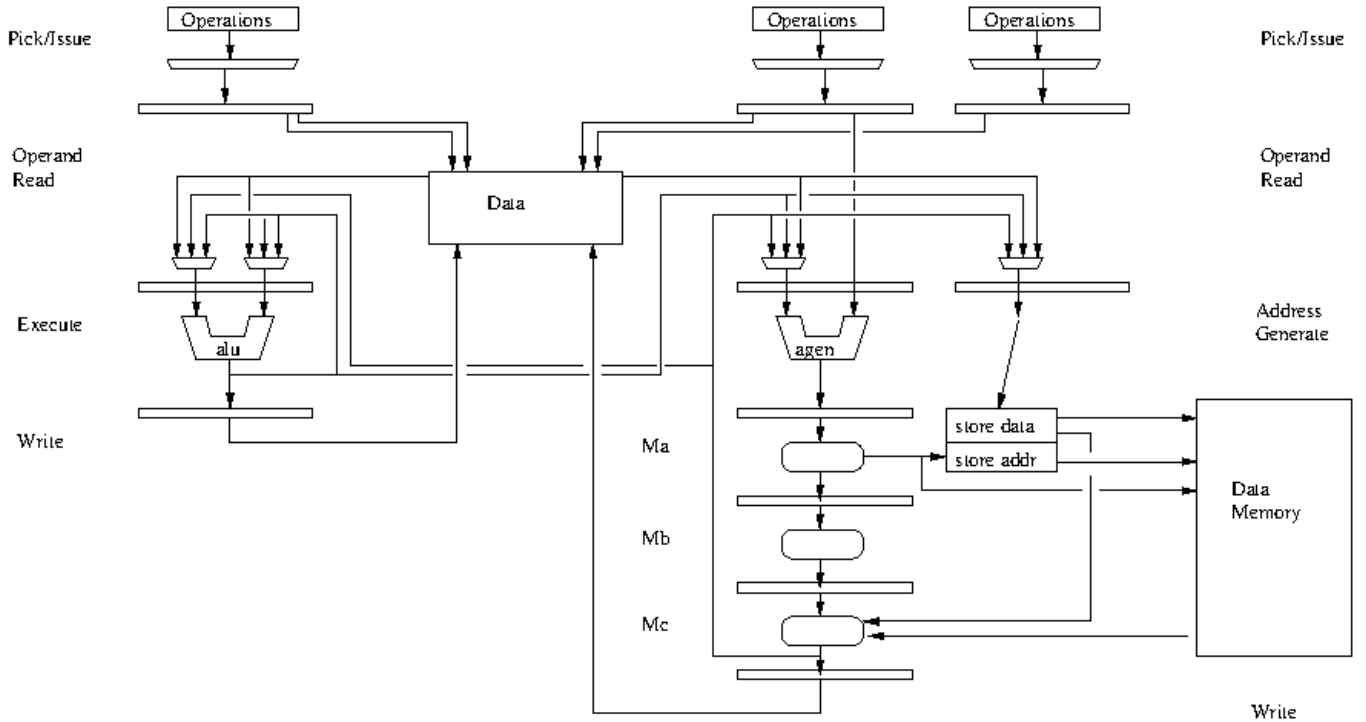
Vi splitter STORE operationer i to: "store address" og "store data"

```
ADDI a5, a0, 0                  R[15] <- ffffd4
ADD a0, a0, a1                  R[10] <- ffffde
BEQ a1, zero, 100bc         {_}
LBU a4, 0(a2)                   R[14] <- 62
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104ad
ADDI a5, a5, 1                  R[15] <- ffffd5
SB a4, -1(a5)                           62       ->  Mem[ffffd4]
  [store data]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 6c
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104ae
ADDI a5, a5, 1                  R[15] <- ffffd6
SB a4, -1(a5)                           6c       ->  Mem[ffffd5]
  [store data]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 61
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104af
ADDI a5, a5, 1                  R[15] <- ffffd7
SB a4, -1(a5)                           61       ->  Mem[ffffd6]
  [store data]
BNE a5, a0, 100a0           {T}
LBU a4, 0(a2)                   R[14] <- 62
BEQ a4, zero, 100b8         {_}
ADDI a2, a2, 1                  R[12] <- 104b0
ADDI a5, a5, 1                  R[15] <- ffffd8
SB a4, -1(a5)                           62       ->  Mem[ffffd7]
  [store data]
BNE a5, a0, 100a0           {T}
```



Nu foregår adresseberegning så tidligt som muligt!

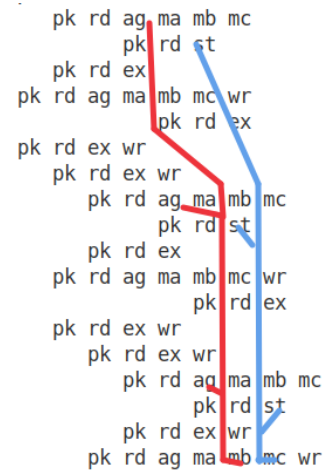
En bedre dataflow maskine



- Kø til spekulative skrivninger beskytter lageret mod for tidlig opdatering
- En LOAD kan se tidligere STOREs ved at søge i Køen og evt forwarde data
- Splittet STORE (ny dedikeret pipeline)

LOAD skal "se" tidligere STORE

```
SB a4, -1(a5)                          62       ->  Mem[ffffd4]
  [store data]
BNE a5, a0, 100a0          {T}
LBU a4, 0(a2)                 R[14] <- 6c
BEQ a4, zero, 100b8        {_}
ADDI a2, a2, 1                R[12] <- 104ae
ADDI a5, a5, 1                R[15] <- ffffd6
SB a4, -1(a5)                          6c       ->  Mem[ffffd5]
  [store data]
BNE a5, a0, 100a0          {T}
LBU a4, 0(a2)                 R[14] <- 61
BEQ a4, zero, 100b8        {_}
ADDI a2, a2, 1                R[12] <- 104af
ADDI a5, a5, 1                R[15] <- ffffd7
SB a4, -1(a5)                          61       ->  Mem[ffffd6]
  [store data]
BNE a5, a0, 100a0          {T}
LBU a4, 0(a2)                 R[14] <- 62
```



Hvis en tidligere STORE har skrevet sine data til store-køen kan en senere LOAD potentielt finde data der.

Vi går ikke her ind i, præcis hvad der sker i "ma", "mb" og "mc", men det ser vanskeligt ud.

Variabel latenstid

Årsager:

- Cache miss (SKAL vente)
- Alias med tidligere STORE hvor data kommer for sent (SKAL vente)
- Tidligere STORE med ukendt adresse (tager vi til sidst - den er grim!)

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

Variabel latenstid

Giver forkert valg af instruktioner:

```
ADDI a5, a0, 0                  R[15] <- ffffd4                         pk rd ex wr
ADD a0, a0, a1                  R[10] <- ffffde                         pk rd ex wr
BEQ a1, zero, 100bc         {_}                                         pk rd ex
LBU a4, 0(a2)                   R[14] <- 62                             pk rd ag ma mb mc wr
BEQ a4, zero, 100b8         {_}                                                     pk rd ex
ADDI a2, a2, 1                  R[12] <- 104ad                          pk rd ex wr
ADDI a5, a5, 1                  R[15] <- ffffd5                            pk rd ex wr
SB a4, -1(a5)                            62       ->  Mem[ffffd4]             pk rd ag ma mb mc
  [store data]                                                                      pk rd st
BNE a5, a0, 100a0           {T}                                               pk rd ex
LBU a4, 0(a2)                   R[14] <- 6c                                pk rd ag ma mb mc wr
BEQ a4, zero, 100b8         {_}                                                        pk rd ex
ADDI a2, a2, 1                  R[12] <- 104ae                             pk rd ex wr
ADDI a5, a5, 1                  R[15] <- ffffd6                               pk rd ex wr
SB a4, -1(a5)                            6c       ->  Mem[ffffd5]                pk rd ag ma mb mc
  [store data]                                                                         pk rd st
BNE a5, a0, 100a0           {T}                                                  pk rd ex
LBU a4, 0(a2)                   R[14] <- 61                                   pk rd ag ma mb mc wr
BEQ a4, zero, 100b8         {_}                                                           pk rd ex
ADDI a2, a2, 1                  R[12] <- 104af                                pk rd ex wr
ADDI a5, a5, 1                  R[15] <- ffffd7                                  pk rd ex wr
SB a4, -1(a5)                            61       ->  Mem[ffffd6]                   pk rd ag ma mb mc
  [store data]                                                                            pk rd st
BNE a5, a0, 100a0           {T}                                                     pk rd ex wr
LBU a4, 0(a2)                   R[14] <- 62                                      pk rd ag ma mb mc wr
BEQ a4, zero, 100b8         {_}                                                              pk rd ex
ADDI a2, a2, 1                  R[12] <- 104b0                                   pk rd ex wr
ADDI a5, a5, 1                  R[15] <- ffffd8                                     pk rd ex wr
SB a4, -1(a5)                            62       ->  Mem[ffffd7]                      pk rd ag ma mb mc
  [store data]                                                                               pk rd st
BNE a5, a0, 100a0           {T}                                                        pk rd ex wr
```

Fejlagtig schedulering

Vi kan undgå forkert schedulering, ved først at opdatere parat-vektoren når vi ved med sikkerhed hvornår en operation kan levere sit resultet. Det vil tilføje to clock perioder til den effektive latens-tid for ALLE LOAD operationer. Det er meget.

I almindelighed vælger man i stedet at gøre "pick" maskineriet lidt mere kompliceret, således at det kan "stilles tilbage" til en tidligere clock periode og genstarte derfra.

Vi kan udvide eksemplet fra tidligere, men antage at den første load instruktion giver et cache miss som detekteres i "mc"

Vi tilføjer tre nye aktiviteter til vores afviklingsplot

```
** Operation aborteret
!! Operation afstår fra at levere resultat til planlagt tid
<> Operation kan nu (senere end planlagt) levere et resultat
```

Eksempel på "replay"

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
```

Vi udvælger her v3 = ADD(v0,v1) ud fra en antagelse om at LOAD instruktionen leverer sit resultat til tiden.

Eksempel på "replay"

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
```

I "mc" finder vor LOAD ud af, at den ikke kan levere resultat som forventet

Eksempel på "replay"

Clock periode 7:

```
Paratvektor: v0,v2,v3,v4,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   ikke parat    -- -- -- -- pk rd **
  v4 = ADD(v3,v1)   ikke 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 mb
```

Tidligere paratvektor retableres og operationer der er startet i clock periode 5 eller senere bliver annulleret

Eksempel på "replay"

Nogle clock perioder senere

```
Paratvektor: v0,v2,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   ikke parat    -- -- -- -- pk rd ** -- -- --
  v4 = ADD(v3,v1)   ikke 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)     fuldført      -- -- pk rd ag ma mb mc wb --
```

Nu kan vor LOAD instruktion levere sit resultat og opdaterer derfor paratvektoren

Eksempel på "replay"

Nogle clock perioder senere +1

```
Paratvektor: v0,v1,v2,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   parat         -- -- -- -- pk rd ** -- -- -- pk
  v4 = ADD(v3,v1)   ikke 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)     fuldført      pk rd ag ma mb mc !! -- -- <> wb
  v7 = LOAD(v6)     fuldført      -- -- pk rd ag ma mb mc wb -- --
```

Den første afhængige instruktion bliver udvalgt for anden gang

Eksempel på "replay"

Nogle clock perioder senere +2

```
Paratvektor: v0,v1,v2,v3,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   startet       -- -- -- -- pk rd ** -- -- -- pk rd
  v4 = ADD(v3,v1)   parat         -- -- -- -- -- pk ** -- -- -- -- pk
  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 !! -- -- <> wb --
  v7 = LOAD(v6)     fuldført      -- -- pk rd ag ma mb mc wb -- -- --
```

Den næste afhængige instruktion bliver udvalgt for anden gang

Eksempel på "replay"

Nogle clock perioder senere +3

```
Paratvektor: v0,v1,v2,v3,v4,v5,v6,v7
Kø (aritmetik):
  v3 = ADD(v0,v1)   startet       -- -- -- -- pk rd ** -- -- -- pk rd ex
  v4 = ADD(v3,v1)   startet       -- -- -- -- -- pk ** -- -- -- -- 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 !! -- -- <> wb -- --
  v7 = LOAD(v6)     fuldført      -- -- pk rd ag ma mb mc wb -- -- -- --
```

Og så kører møllen!

OBS: Vi stiller IKKE eksamensspørgsmål i "fejlagtig schedulering"

Spekulative afhængigheder

Specifik rettet mod situation hvor LOAD møder tidligere STORE med ukendt adresse

- Kun relevant hvis vi ønsker at udføre den LOAD på trods heraf
- Forudsig/antag at der ikke er nogen afhængigheder
- Her dur tidligere replay teknik ikke!!!
- Vi vender tilbage til dette problem, hvis der er tid og interesse

Opsamling

Og indholdet var:

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

Hvad tænker i om det her?

Spørgsmål?

Spørgsmål og Svar