TSEA28 Datorteknik Y (och U)

Föreläsning 15

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

- Andra processorstrukturer
- Intro lab 5
  - Princip
  - Exempel

Praktiska kommentarer

- Sorteringstävlingen deadline: Onsdag 14/5 kl 23:59.
  - Jag ska kunna ladda design, ändra värden i PM adress E0-FF, trycka på regs=0, nollställ klockcykler, och sedan Kont.
  - Sista maskinkodsinstruktionen ska vara ”HALT” från 1:a delen av labben.
  - Bidrag skickas via email till Kent.Palmkvist@liu.se
- Extra redovisningstillfälle lab 1-3
  - 10 Juni kl 9.00 i MUX1
- Extra redovisningstillfälle lab 4
  - 10 Juni kl 13.15 i MUX1

Lab 5, beskrivning av uppgifterna

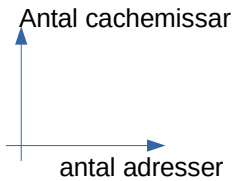
- Uppgift 0: Förberedelseuppgifter!
  - Se att man förstått principell funktion hos cache och bus
- Uppgift 1: Analysera busstrafik när instruktion hämtas
  - Vilka adresser läses? På vilket sätt?
  - När startar läsning, när får processorn reda på vilken nästa instruktion är?
  - Vad händer (på bussen) efter att processorn frågat efter instruktionen med innan svaret kommit?
  - Tips antal instruktioner som körts: Räkna från att subrutinanrop gjorts i main (räkna ej med instruktioner i main)

Lab 5, beskrivning av uppgifterna

- Uppgift 2: Ta reda på datacachens associativitet
  - Skriv litet program som läser en uppsättning data från minnet flera gånger (3 gånger).
    - Målet är att alla data ligger i minnet på adresser som ska placeras på samma cacheline i cache men med olika tag.
  - Titta på bussens läsningar
    - När data får plats i cache läses data bara en gång
    - När alla data inte får plats i cache kommer samma adress läsas flera gånger
  - Ta reda på när cachen inte klarar att lagra alla värden mellan läsningarna.
  - Se upp med storleken på cache (512KB, antal byte i en cacheline från förberedelseuppgift F.2).

Lab 5, tips bakgrund

- Uppgift 2: Ta reda på datacachens associativitet
  - Okänd associativitet => vet inte hur många parallella direktadresserade cachestrukturer
  - Börja med anta direktaddresserad cache (associativitet 1)
  - Beräkna steglängd för att få samma index på alla olika adresser
    - Räcker veta storleken på cache
    - Alternativ: räkna ut vilka bitar som används till respektive funktion
    - Möjliga problem vid fel steglängd
  - Fyll i tabell
    - Minst 2, troligen fler värden än så
  - Rita graf, identifiera associativitet



Praktiska kommentarer

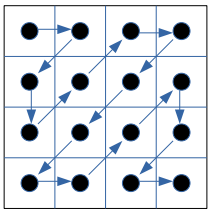
- Laboration 5 förberedelser kan göras på distans
  - Kräver möjlighet använda maskiner i grinden mha rdp-protokoll
  - Logga in via rdpklienter.edu.liu.se
- Andra kurser samt även arbete utanför schemalagd tid i labbet
  - Måste kontrollera att ingen schemalagd labb pågår
  - Måste bara försöka använda lediga maskiner
- Tillgång till labb efter 1:a labbtillfället (lab 5a) genomförts för alla
- Labb på plats och förberedelse mha rdp använder lab mux4

Variantera på datorarkitekturer

Exempel på vanliga beräkningar

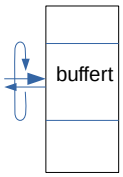
- JPEG/MPEG kodning
  - 8x8 cosinustransform
  - Kvantisering (viktning och avrundning)
  - Zig-zag utläsning (Ordna om data)
  - RLL-kodning (ange antal 0 följd av värde)
- MP3
  - Uppdelning av ljud i frekvensband (cosinustransform)
  - Diverse kodning (baserad på örats förmåga särskilja ljud)
  - Huffmankodning (olika antal bitar i symboler, få bitar för symboler som kommer ofta)

$$y_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right]$$



Digital signalbehandling

- MP3 och JPEG exempel på digital signalbehandling
- Typiska egenskaper
  - Ofta bestående av multiplikationer och additioner
    - Summa av produkter = FIR filter
  - Mycket loopar
  - Hantering av buffertar (t ex 16 senaste värden)
  - Cirkulär adressering (bas + offset mod size)
    - Flytta inte på data i buffert, flytta start och slutpunkt istället
  - Extrema krav på energieffektivitet
    - Mobila/batteridrivna applikationer vanliga
    - Energi per operation



DSP (Digital Signal Processor)

- Speciell familj av processor har skapats
  - 12, 16 eller 24 bitars dataord (inte bara 8, 16, 32, eller 64 bitar)
  - Multipla minnesbankar och bussar
    - Läs argument från minnen parallellt
  - Speciella adresseringsmoder
    - buffer, zigzag-ordning, bitreverserad ordning
  - Ingen cache (oftast)
    - Måste alltid veta hur lång tid operationer tar (realtidskrav)
  - Speciella instruktioner
    - Multiplicera och addera, loopräknare som inte kräver separata subtrahera och hoppa instruktioner
  - Låg effektförbrukning
- En hel del finesser har även introducerats i vanliga processorer

DSP finesser adderade till vanliga processorer

- Ofta appliceras samma operation på många data
- Exempel: Summera element i två vektorer

  $c_i = a_i + b_i$
- Med 32-bitars och 64-bitars processorer blir hantering av data (16 eller 24 bitar) ineffektivt
  - Bättre att läsa och hantera flera data parallellt
  - Exempel: 4 stycken 16-bitars data i ett 64-bitars ord
    - Addition måste spärra carry från 16-bitar dataord till nästa
- Känt som multimedia extensions
  - Intel: MMX, ARM: NEON, PowerPC: 3Dnow

SIMD (Single Instruction Multiple Data)

- Multimedia-expansion är exempel på SIMD
  - Flera data hanteras parallelt av en instruktion
    - Speciella processorer
  - Vektorbaserad beräkning
    - Ofta korta vektorer i vanliga processorer (2-8 värden)
- Läs och skriv parallella data (vektoraddition)
  - Jämför lab1, checkcode subrutinen
- Exempel principiell funktion hos instruktion:

  Avkoda instruktion

  Läs 8 data

  Läs ytterligare 8 data

  Addera data två och två parallellt

  Skriv 8 data
- Mycket snabbare än att läsa och skriva ett data i taget
  - Instruktionsavkodning och loophantering tillkommer

Exempel på SIMD instruktion (Intel MMX)

- Addition av konstant C till 8-bitars värden i en 24-elements vektor (Intel MMX)

```
        MOVQ MM1,C          ; 8 kopior av konstant i MM1
        MOV  CX,3           ; loopräknare (24=3*8)
        MOV  ESI,0          ; index in i vektor
Next:   MOVQ MM0,x[ESI]     ; ladda 8 byte
        PADDB    MM0,MM1    ; Addera byte för byte
        MOVQ x[ESI],MM0     ; Spara resultat
        ADD  ESI,8          ; öka index med 8
        LOOP Next           ; loophantering med cx
        EMMES               ; klar med MMX operationer
```

Hantering av overflow i SIMD

- Data från en beräkning får inte spilla över i beräkningen bredvid
  - Carry får inte skickas vidare
  - Svårt hantera flera carry parallellt om inte inbyggt stöd i processorstrukturen
- Vanlig (ej SIMD/multimedia) overflow (wraparound)
  - 0xff + 1 = 0x00, 0x00 - 1 = 0xFF
- SIMD/multimedia hantering
  - 0xff + 1 = 0xff
  - Mättnatslogik används (använd närmaste tillåtna värde)
    - Max = 0xF....F, Min = 0x0...0 för positiva heltal
    - Max = 0x7F..F, Min = 0x80...0 för tvåkomplement

Jämförelser och villkorliga hopp i SIMD

- Olika data jämförs samtidigt
  - Flera olika resultat parallellt, men det går inte ha olika instruktionssekvenser parallellt
  - Måste lösa villkorliga hopp utan att ändra instruktionssekvens
  - En lösning: Sätt värde till bara 1:or eller bara 0:or som resultat på jämförelse
- Exempel: Sätt 8-bitars pixlar med värde < 50 till 0
  - Antag MM0 = 0x5050505050505050

```
PCMPGTb MM0, MM1      ; jämför 8 pixlar i MM1
PAND    MM1, MM0 ; Sätt pixlar = 0 om pixel < 50
```

Pixlar i MM1:	20 8F C2 30 34 40 F1 A3	
Tröskelvärde i MM0:	50 50 50 50 50 50 50 50	
Jämförelseresultat i MM0:	00 FF FF 00 00 00 FF FF	(efter PCMPGTb)
Slutresultat i MM1:	00 8F C2 00 00 00 F1 A3	(efter PAND)

Skalärprodukt som SIMD

- Orginalkod: skalärprodukt på två vektorer med 40 element var
- SIMD-lösning med 64-bitars register

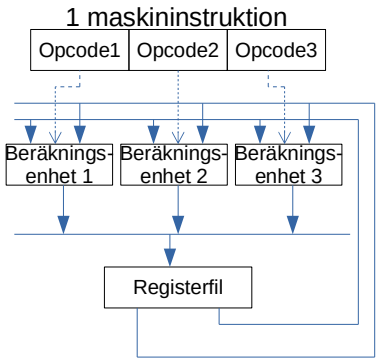
```
dotprod:  mov       r0,#40     ; r0 är loopräknaren
          mov       MM2,#0     ; MM2 är ackumulator
loop:
          ldrl      MM0,[r4],#4    ; Läs 4 ord parallellt
          ldrl      MM1,[r4],#4
          pmaddwd   MM0,MM1        ; multiplicera 16-bitarstal parvis, addera ihop
parvis
          paddd     MM0,MM2        ; addera produkt av två första paren
          psrq      MM0,32         ; flytta 32-bitars produkt till lägre 32 bitar i MM0
          paddd     MM0,MM2        ; addera två sista produkterna till ackumulator
          subs      r0,r0,#4       ; Räkna ner loopräknaren
          bne       loop
          nop                      ; <-- Delayslot för hoppet

          bx        lr
```


Flynns taxanomi

- Single Instruction Single data (SISD)
  - Den ”vanliga processorn”, 1 sekvens av data påverkas av 1 sekvens av instruktioner
- Single Instruction Multiple data (SIMD)
  - Specialstruktur för att applicera samma instruktion till många datasekvenser samtidigt
- Multiple Instruction Multiple Data (MIMD)
  - Flera ”vanliga” processorer sammansatta i en dator
  - Det vanliga fallet numera (t ex 4 kärnor i en PC, 8 kärnor i en mobiltelefon etc.)
- MISD (multiple instruction, single data) finns det?
  - Feltoleranta system (olika algoritmer på samma indata + majoritetsbeslut)

Ökning av antal operationer per klockcykel

- Pipelining tillåter maximalt 1 instruktion per klockcykel
- Alternativ: Ange flera operationer i samma maskininstruktion
  - Används av en del DSP-processorer
  - Varje maskininstruktion har separata fält motsvarande flera vanliga instruktioner
  - Kan skapas genom att kombinera två eller fler instruktioner från en vanlig processor
  - Brukar innebära flera beräkningsenheter (ALU, multiplikator, shiftenhet) som kan användas parallellt



VLIW (Very Long Instruction Words) processor

- Kombination av många opcoder skapar långa maskinkodsinstruktioner
  - 256 bitar inte ovanligt
  - Liknar väldigt mycket mikrokodsinstruktioner
- Olika beräkningsenheter kan ha olika egenskaper
  - Ett par ALU, någon multiplikator, någon skiftenhet
- Kan inte alltid packa ihop en sekvens så att alla enheter används samtidigt
  - Kräver addition av NOP (liknar stall i en pipeline)

VLIW exempel

- Antag 2 ALU samt en multiplikator
- Exempel 1: sekvensiell kod

```
ADD R1,R2,R3    ; R1 = R2+R3
ADD R4,R5,R6    ; R4 = R5+R6
MULT R7,R8,R9   ; R7 = R8 * R9
```

  - Kan realiseras som en VLIW instruktion

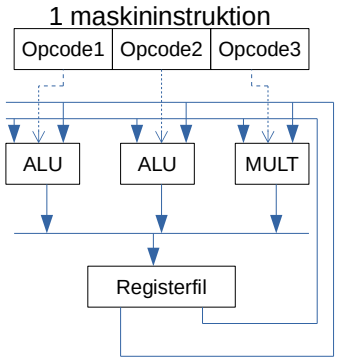
```
ADD R1,R2,R3 | ADD R4,R5,R6 | MULT R7,R8,R9
```

- Exempel 2: Sekvensiell kod

```
ADD R1,R2,R3
MULT R4,R5,R6
MULT R7,R8,R9
```

  - Kan inte realiseras i en instruktion (saknar 2 mult)

```
ADD R1,R2,R3 | NOP | MULT R4,R5,R6
NOP          | NOP | MULT R7,R8,R9
```



Exempel på VLIW

- Skalärprodukt på två vektorer med 40 element var

$$\sum_{i=1}^{40} x_i \cdot y_i = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_{40} \cdot y_{40}$$

- Antag VLIW där varje instruktion kan utföra två operationer
  - Databeroenden kan fortfarande inte lösas
  - Behöver skriva om kod som tidigare (för att hantera datakonflikt)

```
dotprod: mov r0,#40          |  mov r1,#0
loop:
         ldrshr2,[r4],#2     |  nop            ; En minnesport => en läsning

         ldrshr3,[r5],#2     |  nop            ; r3 kan inte användas direkt
         mul  r2,r2,r3       |  nop            ; Databeroenden, bara en instr.
         add  r1,r1,r3       |  subs  r0,r0,#1 ; Räkna ner loopräknaren
         bne  loop           |  nop            ; om bra branch prediction

         bx   lr
```

Exempel på VLIW, forts

- Utgå från version med två parallella loopar
- Bättre än förra
  - Fortfarande beroende av omskrivning av programmeraren

```
dotprod:  mov r0,#40     |     mov r1,#0       ; r1 är ackumulator
loop:
          ldrshr2,[r4],#2 |     nop             ; 2 iterationer per loop, forfarande bara 1 minne
          ldrshr3,[r5],#2 |     nop
          ldrshr6,[r4],#2 |     nop
          ldrshr7,[r5],#2 |     mul  r2,r3
          mul  r6,r6,r7   |     add  r1,r1,r3
          subsr0,r0,#2    |     nop             ; Räkna ner loopräknaren med TVÅ!
          bne  loop       |     add  r1,r6      ; <-- Delayslot för hoppet

          bx   lr
```


VLIW egenskaper

- Enkel att avkoda instruktioner
  - Dela upp långa instruktioner i delfunktioner och kör parallellt
- Kompilering/programmering komplicerat
  - Kräver full förståelse för arkitekturen
  - Kräver full kontroll över databeroenden
  - Svårt kunna utnyttja alla beräkningsenheter parallellt
  - Förändras arkitekturn måste program kompileras om

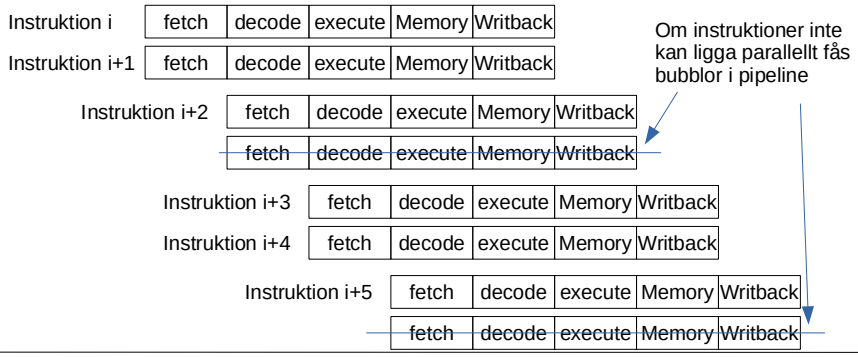
Superskalär processor

Alternativ till VLIW: Superskalär

- Öka antal instruktioner till flera per klockcykel
  - Läs många fler byte per läsning i minnet på en gång (bredare bussar)
- Om m parallella pipeline används kan maximalt m gånger snabbare exekvering fås
  - M-vägs superskalär processor
  - Kan få < 1 klockcykel per instruktion
    - dvs IPC (Instruction per Clockcycle) > 1
- Måste kunna utföra flera beräkningar parallellt
  - Fler ALU och motsvarande behövs
- Måste utföra samma beräkningar som en icke-superskalär processor
  - Databeroenden måste uppfyllas

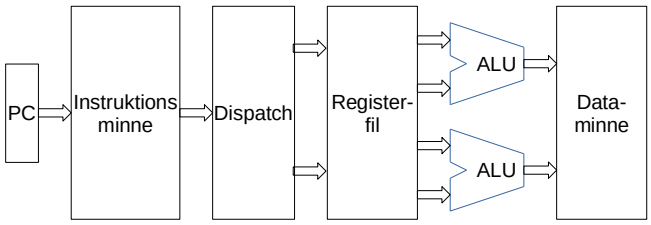
Superskalär RISC processor, funktion

- 2-vägs superskalär struktur
- Hämta två instruktioner varje klockcykel
  - Inte alltid möjligt exekvera båda parallellt



Superskalär processor, arkitektur

- Instruktionsminne skickar flera instruktioner per läsning
- Dispatch fördelar instruktioner till beräkningsenheter
  - Komplicerad enhet
- Registerfil med fler läs och skrivportar
- Flera exekveringsenheter
- Minne med flera vägar



Superskalär process, problem

- Ännu mer potentiella problem med konflikter
  - Datakonflikt går inte lösa med forwarding
- Kostnad för bubblor ökar
- Svårt hitta parallella instruktioner
  - Databeroenden begränsar hur många instruktioner som kan utföras parallellt
  - Var börjar nästa instruktion? Om olika längd måste nästa hittas snabbt
- Måste behålla samma beteende som en rent sekvensiell maskin
  - Samma utresultat
  - Samma ordning på minnesskrivning/läsning?

Exempel på superskalär exekvering

- Orginalkod: skalärprodukt på två vektorer med 40 element var

$$\sum_{i=1}^{40} x_i \cdot y_i = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_{40} \cdot y_{40}$$

- Antag 2-vägs superskalär processor (1 minne, 2 ALU, ingen delayslot)

```
dotprod:  mov r0,#40     ; r0 är loopräknaren  \__  samtidigt
          mov r1,#0      ; r1 är ackumulator   /
loop:
          ldrshr2,[r4],#2 ; Kan ej köras samtidigt med nästa pga 1 minne
          ldrshr3,[r5],#2 ;   -   '   '    -
          mul  r3,r3,r2   ; Databeroende, kan inte köra samtidigt med add.l
          add  r1,r1,r3   ; Kan köra samtidigt med sub
          sub  r0,r0,#1   ; Kan köras samtidigt med 1a ldrsh om hopp döljs av
          bne  loop       ; branch prediction

          bx   lr
```

Exempel konflikthantering (2-vägs superskalär processor, 1 minnesport)

- Orginalkod: skalärprodukt på två vektorer med 40 element var
- Orginalkod och utrullad kod (2 varv) $\sum_{i=1}^{40} x_i \cdot y_i = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_{40} \cdot y_{40}$

```
dotprod:    mov  r0,#40
            mov  r1,#0
loop:
          { ldrsh r2,[r4],#2
          { ldrsh r3,[r5],#2
          { mul   r3,r3,r2
          ( add   r1,r1,r3
          ( sub   r0,r0,#1
            bne   loop
            bx    lr
```

```
dotprod:    mov  r0,#40,     ; kan köras samtidigt med mov r1,#0
            mov  r1,#0
loop:
          { ldrsh r2,[r4],#2 ; 1 minne, kan inte köras samtidigt med nästa
          { ldrsh r3,[r5],#2 ; -   '   '   -
          { ldrsh r6,[r4],#2 ; -   '   '   -
          ( ldrsh r7,[r5],#2 ; kan köras samtidigt med mul
          ( mul   r3,r3,r2
          ( mul   r7,r7,r6   ; kan köras samtidig med add
          ( add   r1,r1,r3
          ( add   r1,r1,r7   ;
          ( subs  r0,r0,#2   ; Räkna ner loopräknaren med TVÅ!
            bne   loop
            bx    lr
```


Mycket outnyttjad parallellism tillgänglig

- Orginalkod: 2 minnesaccess + 2 beräkningar per varv (+ 1 beräkning och 1 hopp)
- Antag 2-vägs superskalär!
  - Som bäst behövs bara 2 klockcykler per varv om loopstyrning kan undvikas
    - 2 minnesläsningar + 2 beräkningar

```
dotprod2t:  mov  r0,#40      ; kan köras samtidigt med mov r1,#0
            mov  r1,#0
loop2t:
          { ldrsh r2,[r4],#2 ; 1 minne, kan inte köras samtidigt med nästa
          { ldrsh r3,[r5],#2 ; -   '   '   -
          { ldrsh r6,[r4],#2 ; -   '   '   -
          ( ldrsh r7,[r5],#2 ; kan köras samtidigt med mul
          ( mul   r3,r3,r2
          ( mul   r7,r7,r6   ; kan köras samtidig med add
          ( add   r1,r1,r3
          ( add   r1,r1,r7   ;
          ( subs  r0,r0,#2   ; Räkna ner loopräknaren med TVÅ!
            bne   loop2t
            bx    lr
```

Bättre implementering: Fler varv

- Två parallella 2 varvs utrullning (antag det finns fler register)
- Flytta ordning, räkna inte ihop slutsumman direkt (ingen delayslot i detta exempel)
  - Många minnesläsningar i sekvens etc.

```
dotprod4t:  mov  r0,#40
            mov  r1,#0
            mov  r12,#0
loop4t:
            ldrsh r2,[r4],#2
            ldrsh r3,[r5],#2
            ldrsh r6,[r4],#2
            ldrsh r7,[r5],#2
            ldrsh r8,[r4],#2
            ldrsh r9,[r5],#2
            ldrsh r10,[r4],#2
            ldrsh r11,[r5],#2
            mul   r3,r3,r2
            mul   r7,r7,r6
            mul   r9,r9,r8
            mul   r11,r11,r10
            add   r1,r1,r3
            add   r12,r12,r7
            add   r1,r1,r9
            add   r12,r12,r11

            subs  r0,r0,#4
            bne   loop4t

            add   r1,r1,r12
            bx    lr
```


Bättre implementering: Fler varv, forts.

- Flyttat hälften av minnesläsningarna sist i loop
  - Måste då även sätta rätt värden innan loop startar

```
dotprod4t:  mov r0,#40
            mov r1,#0
            mov r12,#0
            ldrshr2,[r4],#2
            ldrshr3,[r5],#2
            ldrshr6,[r4],#2
            ldrshr7,[r5],#2
Loop4t:
            ldrshr8,[r4],#2
            ldrshr9,[r5],#2
            ldrshr10,[r4],#2
            ldrshr11,[r5],#2
            mul  r3,r3,r2
            mul  r7,r7,r6
            mul  r9,r9,r8
            mul  r11,r11,r10
            add  r1,r1,r3
            add  r12,r12,r7
            add  r1,r1,r9
            add  r12,r12,r11
            ldrshr2,[r4],#2
            ldrshr3,[r5],#2
            ldrshr6,[r4],#2
            ldrshr7,[r5],#2
            subsr0,r0,#4
            bne  loop4t
            add  r1,r1,r12
            bx   lr
```

Risk med flyttade minneläsningar

- 4 data läses extra (utanför vektorerna)!
  - Kan ge försök att läsa otillåtna minnesadresser (minnesskydd?)
- Löses enklast med extra specialavslutning av loop
  - Prolog före loop
    - Läs indata
  - Epilog efter loop
    - Avslutande skrivningar

```
dotprod4t: mov  r0,#40
           mov  r1,#0
           mov  r12,#0
           ldrsh r2,[r4],#2
           ldrsh r3,[r5],#2
           ldrsh r6,[r4],#2
           ldrsh r7,[r5],#2
Loop4t:
           ldrsh r8,[r4],#2
           ldrsh r9,[r5],#2
           ldrsh r10,[r4],#2
           ldrsh r11,[r5],#2
           mul   r3,r3,r2
           mul   r7,r7,r6
           mul   r9,r9,r8
           mul   r11,r11,r10
           add   r1,r1,r3
           add   r12,r12,r7
           add   r1,r1,r9
           add   r12,r12,r11
           ldrsh r2,[r4],#2
           ldrsh r3,[r5],#2
           ldrsh r6,[r4],#2
           ldrsh r7,[r5],#2
           subs r0,r0,#4
           bne   loop4t
           add   r1,r1,r12
           bx    lr
```


Nästan nere på 2 klockcykler per varv (9 klockcykler / 4 varv)

- Sprid ut beräkningarna bland läsinstruktionerna
  - Tillåter samtidigt beräkning och läsning utan databeroende
- Denna typ av omskrivning känd som software pipelining
- För en VLIW-processor skulle programmeraren behöva utföra denna omskrivning

```
dotprod4t: mov  r0,#40
           mov  r1,#0
           mov  r12,#0
           ldrsh r2,[r4],#2
           ldrsh r3,[r5],#2
           ldrsh r6,[r4],#2
           ldrsh r7,[r5],#2
Loop4t:
{          ldrsh r8,[r4],#2
{          mul   r3,r3,r2
{          ldrsh r9,[r5],#2
{          mul   r7,r7,r6
{          ldrsh r10,[r4],#2
{          add   r1,r1,r3
{          ldrsh r11,[r5],#2
           add   r12,r12,r7

{ ldrsh r2,[r4],#2
{ mul   r9,r9,r8
{ ldrsh r3,[r5],#2
{ mul   r11,r11,r10
{ ldrsh r6,[r4],#2
{ add   r1,r1,r9
{ ldrsh r7,[r5],#2
{ add   r12,r12,r11
{ subs r0,r0,#4
{ bne   loop4t

  add   r1,r1,r12
  bx    lr
```

Skillnad mellan VLIW och superskalär

- Superskalär tar själv reda på vilka instruktioner som kan köras parallellt
  - Varje instruktion är fortfarande en liten/enkel instruktion
  - Processorn försöker exekvera sekvensiell kod parallelt
  - Processorn räknar själv ut och hanterar data och kontrollberoende
  - Kompilator/programmerare kan förutsätta sekvensiellt exekverad kod
- VLIW har från början definierat vilka operationer som ska ske parallellt i varje klockcykel
  - Instruktionen är mycket större än för superskalär
  - Delinstruktioner kan inte automatiskt flyttas mellan klockcykler
  - Kompilator/programmerare måste garantera begränsningar hos arkitekturen
  - Svårt flytta kod mellan olika arkitekturer (även med samma instruktionsuppsättning)

Out-of-Order execution

Automatiserat sätt att ändra instruktions-ordning

- Processorn hämtar in många instruktioner och avkodar dom
- Bestäm sedan vilka som ska utföras beroende på om indata finns tillgängligt
  - Instruktioner kan då utföras i annan ordning
  - Måste garantera att slutresultat fortfarande blir samma
  - I exemplet: Läs flera varv ur orginalkod, välj att exekvera instruktioner som har indata tillgängligt i respektive klockcykel
- Känt som out-of-order execution
  - Kräver mycket hårdvara och bra branch prediction

Exempel på superskalär exekvering, avancerad version

- Samma exempel som Figure 8.6 – Figure 8.7 i boken

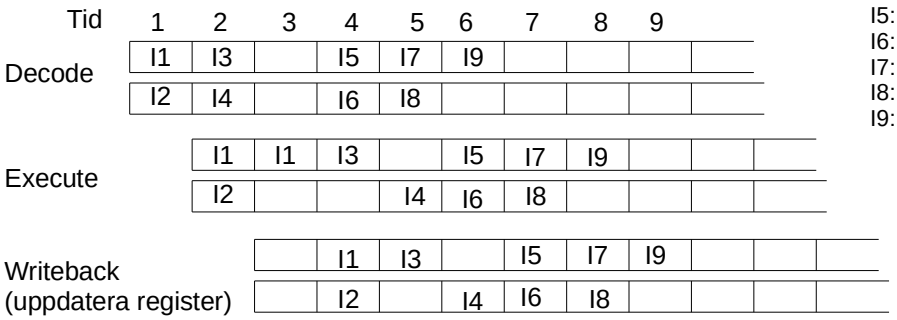
```
I1: ldr   r0,[r10]     ; r0 = minne(r10)
I2: add   r10,r10,#4   ; r10 = r10 + 4
I3: mul   r3,r2,r1     ; r3 = r1*r2
I4: mul   r4,r1,r0     ; r4 = r1*r0
I5: sub   r5,r4,r6     ; r5 = r4 – r6 , r4 skapad i I4:
I6: add   r7,r8,r2     ; r7 = r8 + r2
I7: mul   r0,r9,r5     ; r0 = r9*r5
I8: add   r1,r1,#4   ; r1 = r1 + 4
I9: add   r7,r7,#1   ; r7 = r7 + 1
```

- Antag 2-väg superskalär processor med 2 klockcyklers minnesläsning, 1 multiplikator.

Exempel på superskalär processor, forts.

- 2 klockcykler för minnesaccess, 1 mult.

```
I1:  ldr  r0,[r10]
I2:  add  r10,r10,#4
I3:  mul  r3,r2,r1
I4:  mul  r4,r1,r0
I5:  sub  r5,r4,r6
I6:  add  r7,r8,r2
I7:  mul  r0,r9,r5
I8:  add  r1,r1,#4
I9:  add  r7,r7,#1
```

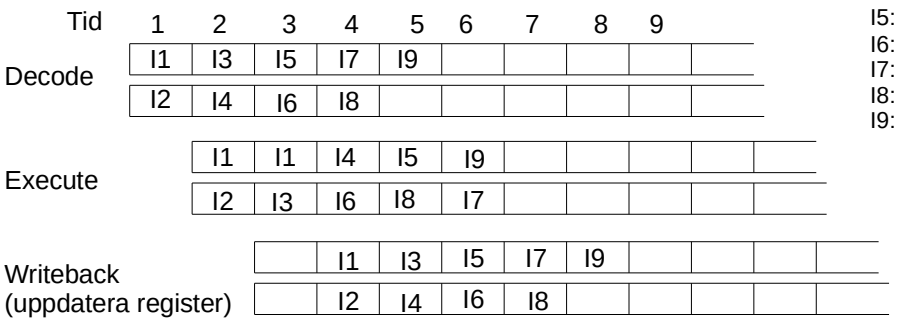


Samma ordning på registeruppdatering som enligt koden.
Begränsning: 2 cykel minnesläsning, 1 multiplikator

Exempel på superskalär processor, out-of-order exekvering

- 2 klockcykler för minnesaccess, 1 mult.

```
I1:  ldr  r0,[r10]
I2:  add  r10,r10,#4
I3:  mul  r3,r2,r1
I4:  mul  r4,r1,r0
I5:  sub  r5,r4,r6
I6:  add  r7,r8,r2
I7:  mul  r0,r9,r5
I8:  add  r1,r1,#4
I9:  add  r7,r7,#1
```



Begränsning: 2 cykel minnesläsning, 1 multiplikator
I6 utförs nu före I5, och I8 före I7. Dock samma ordning på registeruppdatering!

Problem och dellösningar för superskalär pipelined RISC

- Datakonflikt
  - Forwarding (kopiera ALU utdata till ALU indata)
- Styrkonflikt
  - Branch prediction (gissa nästa instruktionsadress)
- Strukturell konflikt
  - Dispatch delar ut instruktioner att utföra mellan de två ALU och minnesvägarna
- Kräver fortfarande anpassning av kod för att få snabbare exekvering
  - Ändring av antal och typ av beräkningsblock kräver annorlunda kod (instruktionsordning)
  - Löser inte alla prestandaproblem

LINKÖPINGS UNIVERSITET