

TSEA28 Datorteknik Y (och U)

Föreläsning 13

Kent Palmkvist, ISY



Dagens föreläsning

- Strukturella konflikter
- SIMD
- VLIW
- Superskalära processorer
- Minnen

Praktiska kommentarer

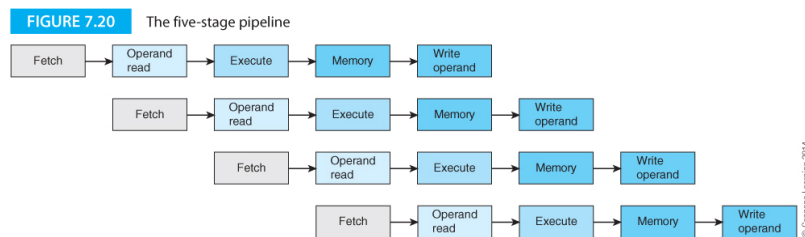
- Sorteringsuppgiften: Glöm inte testa mer än given sekvens i labbdokumentet!
 - Flera lika värden?
 - Minsta möjliga värdet (0x8000)
 - Största möjliga värdet (0x7fff)
 - Bara positiva respektive bara negativa värden?
- Sorteringstävlingen deadline: Onsdag 8/5 kl 23:59.
 - Jag ska kunna ladda design, 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 (eller använda SEQ-fältet).

Mål för datorarkitektur

- Utför så många instruktioner (arbete) per tidsenhet som möjligt
 - Med begränsning av effektförbrukning och pris
- Orsak till ökande beräkningskrav (applikationsområden)
 - Matematiska beräkningar (väderprognoser, lösning av ekvationssystem, simulering av konstruktioner, CAD)
 - Multimedia (ljudkodning, bildkomprimering)
 - Kryptering (krypterad överföring, signering, bitcoin, block-chain)
 - Kommunikation (modulering, felrättande koder)
 - Autonoma system (maskininlärning, identifiering)

Typer av konflikter i en pipelinad processor

- Tillåter flera instruktion exekverade parallellt
 - Starta nästa innan föregående utförd färdigt
- Startar fortfarande bara en instruktion per klockcykel
 - Sekvensiell instruktionsexekvering
 - Arbetar parallellt med olika delar av flera instruktioner samtidigt
- Får problem med diverse konflikter
 - Tvingas stanna exekvering av efterföljande instruktioner i vissa fall (STALL)

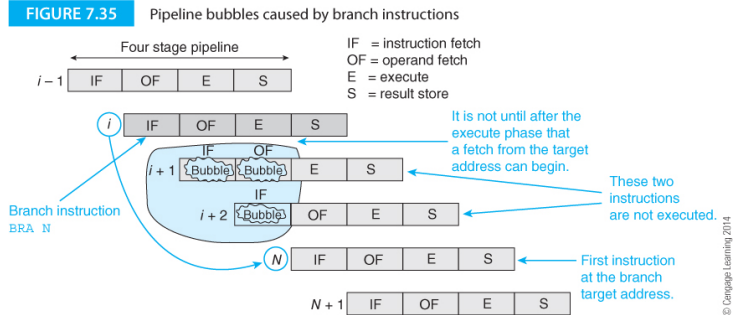


Typer av konflikter i en pipelinad processor

- Datakonflikt (data inte tillgänglig)
 - Data inte tillgängligt från annan tidigare instruktion
 - ADD R3,R4,R5 ; R3 = R4+R5
 - ADD R2,R3,R4 ; R3 inte tillgänglig direkt (R2=R3+R4)
- Styrkonflikt (nästa instruktion hämtas fel)
 - Icke-linjär instruktionssekvens
 - BRA newplace ; nästa instruktion blir inte
 - other: MOVE R2,#23 ; instruktion på nästa adress
- Strukturell konflikt (upptagen hårdvara)
 - Beräkningsenhet/resurs upptagen av annan instruktion
 - ADD R0,R1,R2 ; addition både av operand och
 - LDR R0,[R0,#4] ; adressberäkning

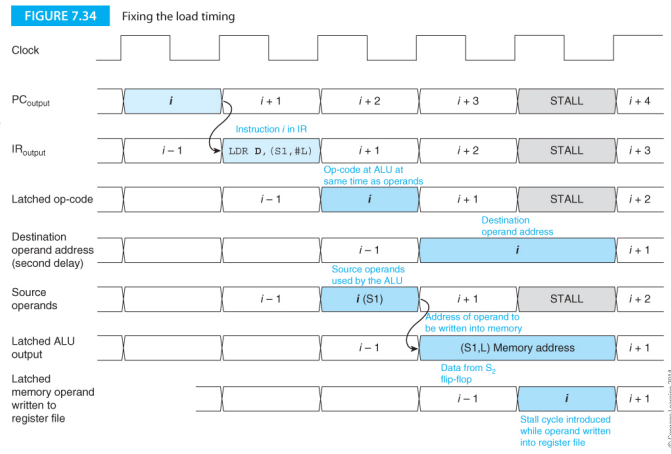
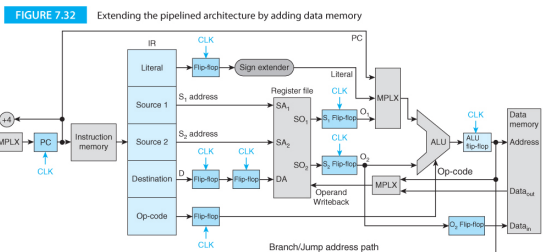
Styrkonflikt

- Nästa instruktion hämtas innan hopp avkodats
 - Execute påverkar programräknare (PC)
 - Måste låta bli utföra två instruktioner
 - Kasta bort hämtade instruktioner
- Gäller även subrutinanrop, avbrott etc.
- Villkorliga hopp som inte tas kostar inget extra



Strukturell konflikt exempel (minneläsning)

- Exempel: LDR D,(S1,#L) följd av ytterligare en likadan instruktion
 - Måste ge registerfil tid att spara värde från minne
- Samma hårdvara (minne) behöver ge access till flera instruktioner samtidigt



Hantering av strukturell konflikt

- Enklast? Mer hårdvara
 - Exempel: Ytterligare minnesport för samtidig läsning/skrivning
 - Mer hårdvara ökar kostnad
 - Avvägning mellan prestandavinst jämfört med kostnad
- För programmeraren
 - Undvik sekvenser som genererar strukturell konflikt
 - Specifika lösningar beroende på arkitektur
 - Ny arkitektur kräver omkompilering av kod

Exempel på hantering av pipelineproblem via programvara

- Skalärprodukt av två vektorer med 40 element var

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

- Antag enkel pipelined dator

- Delayslot och ingen forward
 - Hopp utför även instruktionen efter
 - Resultat från beräkningar kan inte läsas av instruktionen direkt efteråt

```

dotprod:  mov r0,#40      ; r4 och r5 pekar på vektorerna
           mov r1,#0      ; r0 är loopräknaren
           ; r1 är ackumulator
loop:     ldrshr2,[r4],#2  ; Läs ur 1:a vektor, öka sedan r4 med 2
           ldrshr3,[r5],#2 ; läs ur 2:a vektor
           mul  r3,r3,r2   ; beräkna M[r4]*M[r5], öka r4 och r5 med 2
           add  r1,r1,r3
           subsr0,r0,#1   ; räkna ned loopräknare
           bne  loop
           nop            ; ← Delayslot för hoppet

           bx   lr        ; återhopp från subrutin

```

Databeroenden i koden

- r3 används direkt efter läsning i multiplikation och sedan i addition.
 - Ger stall i pipeline vid varje instruktion (mul samt add)
 - Varje varv: $1+1+2+2+1+1+1=9$ klockcykler
 - Totalt $1+1+40*9+1=363$ klockcykler
 - Försök använda delayslot
 - Måste hitta lämplig instruktion att flytta
- ```

dotprod: mov r0,#40 ; r4 och r5 pekar på vektorerna
 mov r1,#0 ; r0 är loopräknaren
 ; r1 är ackumulator
loop:
 ldrshr2,[r4],#2 ; Läs ur 1:a vektor, öka r4 med 2
 ldrshr3,[r5],#2 ; läs ur 2:a vektor, öka r5 med 2
 mul r3,r3,r2 ; produkt av vektorelement
 add r1,r1,r3
 subsr0,r0,#1 ; räkna ned loopräknare
 bne loop
 nop ; ← Delayslot för hoppet
bx lr ; återhopp från subrutin

```

## Utnyttja delayslot

- Subs r0,r0,#1 måste ligga före bne, kan inte flyttas
  - add r1,r1,r3 kan placeras i delayslot
    - Sparar 40 klockcykler. Dessutom minskar databeroende => ytterligare 40 sparade cykler
    - totalt 283 klockcykler
      - (ca 20% bättre)
  - Nästa förändring
    - Rulla upp loop (dvs utför flera beräkningar i varje varv i loop)
- ```

dotprod: mov r0,#40 ; r4 och r5 pekar på vektorerna
          mov r1,#0  ; r0 är loopräknaren
          ; r1 är ackumulator
loop:
  ldrshr2,[r4],#2 ; Läs ur 1:a vektor, öka sedan r4 med 2
  ldrshr3,[r5],#2 ; läs ur 2:a vektor, databeroende!
  mul r3,r3,r2    ; produkt av vektorelement
  subsr0,r0,#1   ; räkna ned loopräknare
  bne loop
  add r1,r1,r3    ; ← Delayslot för hoppet
bx lr          ; återhopp från subrutin

```

Utrullad loop, 2 mult per loop

- Minskar databeroende, tar bort STALL

- Sparar 40 klockcykler ytterligare
- 243 klockcykler

- Bättrat på prestanda för skalärprodukt med totalt $120/363=33\%$

```

dotprod:  mov r0,#40    ; r0 är loopräknaren
          mov r1,#0     ; r1 är ackumulator

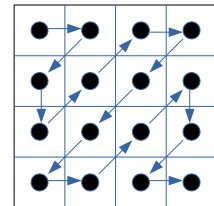
loop:     ldrshr2,[r4],#2 ; 2 iterationer per loop följer
          ldrshr3,[r5],#2 ; databeroenden vilket ger
          ldrshr6,[r4],#2 ; färre STALL
          ldrshr7,[r5],#2
          mul  r3,r3,r2
          mul  r6,r6,r7
          add  r1,r1,r3
          sub  r0,r0,#2  ; Räkna ner loopräknaren med TVÅ!
          bne  loop
          add  r1,r1,r6  ; <-- Delayslot för hoppet

          bx  lr
  
```

Exempel på andra vanliga beräkningar

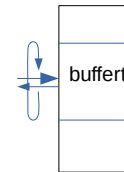
- JPEG/MPEG kodning
 - 8x8 cosinustransform
 - Kvantisering (viktning och avrundning)
 - Zig-zag utläsning (Ordna om data)
 - RLL-kodning (ange antal 0 följd av värde)
- MP3
 - Uppdelning av ljud i frekvensband (cosinustransform)
 - Diverse kodning (baserad på örats förmåga särskilja ljud)
 - Huffmankodning (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 processorer

- Speciell familj av processor har skapats
 - 12, 16 eller 24 bitars dataord (inte bara $2^{(8k)}$ 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 parallellt 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

- Originalkod: 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

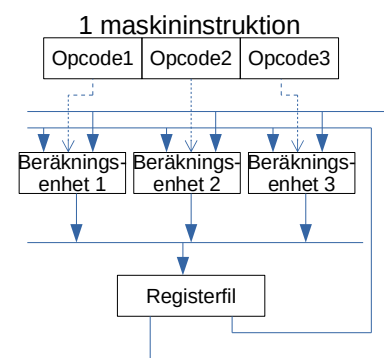
bx      lr
```

Flynnns 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 opcodeer 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
```

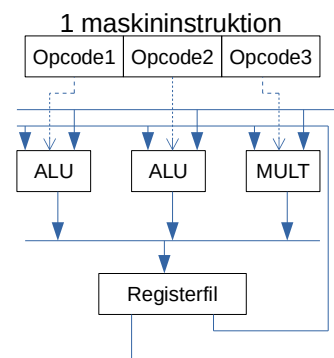
 - Kan realiseras som en VLIW instruktion


```
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

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

- Antag VLIW med två instruktioner

- 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

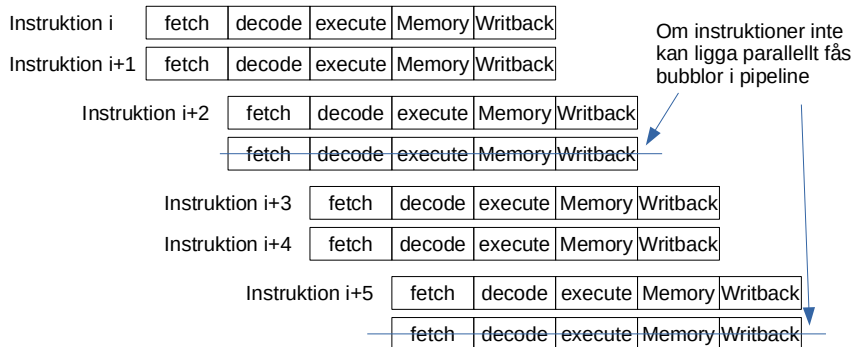
- Enkel att avkoda instruktioner
- Kompilering/programmering komplicerat
  - Kräver full förståelse för arkitekturen
  - Kräver full kontroll över databeroenden
  - Svårt kunna utnyttja alla beräkningsenheter parallellt
  - Förändras arkitekturn måste program kompileras om

## 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
- Måste kunna utföra flera beräkningar parallellt
  - Fler ALU och motsvarande behövs
- Måste utföra samma beräkningar som en icke-superskalär processor
  - Databeroenden måste uppfyllas

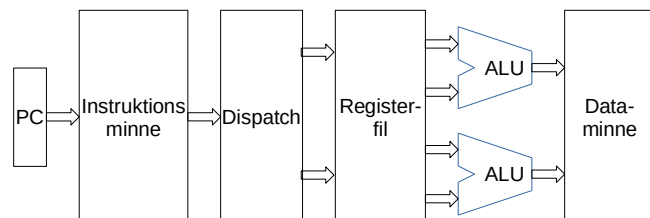
## Superskalär RISC processor, funktion

- 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

- Originalkod: skalärprodukt på två vektorer med 40 element var

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

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

```

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

```

## Superskalär exekvering, utrullad loop

- Skalärprodukt på två vektorer med 40 element var

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

Inte någon stor Upps snabbning (2 eller 3 tillfällen där 2 instruktioner utförs parallellt)

```
dotprod: mov r0,#40 ; kan köras samtidigt med mov r1,#0
 mov r1,#0
loop: ldrshr2,[r4],#2 ; 1 minne, kan inte köras samtidigt med nästa
 ldrshr3,[r5],#2 ; - ' ' -
 ldrshr6,[r4],#2 ; - ' ' -

 ldrshr7,[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 ;
 subsr0,r0,#2 ; Räkna ner loopräknaren med TVÅ!

 bne loop
 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 parallellt
  - 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 instruktionsuppställning)

# Minnen

## Problem med högre klockfrekvenser

- Alla delar i datorn är inte lika snabba
  - Register skickar fort ut värden på bussen/till beräkningselement
  - ALU går olika fort beroende på operation
    - Shift, AND, OR etc. går fort
    - Add/sub går långsammare
  - Multiplikation ännu långsammare
- Minnen är långsamma
  - Större minnen långsammare än mindre minnen (~ logaritmisk relation)
- Fysisk gräns: ljusets hastighet
  - 10 GHz motsvarar 100 ps/klockcykel = 3 cm (vakum!)
  - Långsammare på chip pga material
  - Ren propageringstid, inte medräknat tid för upp och urladdning av kapacitanser!

# Minnesegenskaper

- Många olika egenskaper intressanta
- Processorns användning av minnesceller
  - Kan värde skrivas in i minnesceller (read-write resp. read-only)
  - Latens för läsning och skrivning
  - Genomströmningshastighet för läsning och skrivning
  - Försvinner värdena när strömmen slagits av (flyktiga)
- Tekniska/ekonomiska begränsningar
  - Pris per bit i minnet
  - Storlek på minnet
  - Speciell hantering för att behålla värdet (dynamiska)
  - Möjlighet skriva om innehåll
  - Effektförbrukning

# Minnesegenskaper – läsbara (av processorn)

- Enbart läsning (ROM, Read Only Memory)
  - Processorn kan bara läsa, inte skriva till minnet med en enkel instruktion
- Maskprogrammerade ROM
  - Programmeras vid tillverkning, billigt i stora volymer
- PROM: Programmable Read Only Memory
  - Skrivning kan göras 1 gång, oftast i speciell programmeringsutrustning
- EPROM: Erasable Programmable Read Only Memory
  - Minnet kan raderas i speciell utrustning (T ex UV) och programmeras om
- EEPROM: Electrically Erasable Programmable Read Only Memory
  - Minnet kan raderas cell för cell eller block elektriskt, eventuellt utan speciell utrustning
- FLASH: Speciell typ av EEPROM
  - Kan inte radera enstaka celler, utan stora block (kByte eller Mbyte) raderas per gång. Långsamma att skriva.
  - Tekniken i USB-stickor och SSD-diskar

## Minnesegenskaper – läs/skriv direkt

- Läs och skriv (RAM, Random Access Memory)
  - Processorn kan skriva och läsa till varje cell i minnet
- Beroende på teknik kan värdet i minnet försvinna om man inte läser värdet inom viss tid
  - SRAM: Statiskt RAM, värdet ligger kvar så länge strömmen är på
  - DRAM: Dynamiskt RAM, värdet försvinner om inte det läses och skrivs tillbaka inom viss tid (1-10 ms mellan varje läsning)
    - Bygger på lagring av laddning i en kondensator som sakta laddas ur
    - Jämför skriva i sanden på en strand med vågor. Texten blir till slut oläslig om den inte skrivs om med jämna mellanrum
  - SRAM mycket dyrare per minnescell och snabbare än DRAM

## Fler detaljer om minnestiming för DRAM

- Dynamiskt minne är mer komplicerat att kommunicera med
  - Varje minne har en kontroller som styr refresh och kommunikation
  - Ett protokoll med ca 25 kommandot
  - Ex: Datablad till Micron 256Mbit DDR3 minne är 211 sidor långt
- Mål med minnesdesign
  - Få pinnar, låg effektförbrukning
  - Hög genomströmningshastighet (pipelining)
- Alla persondatorer och telefoner använder DRAM

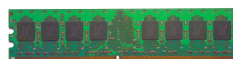
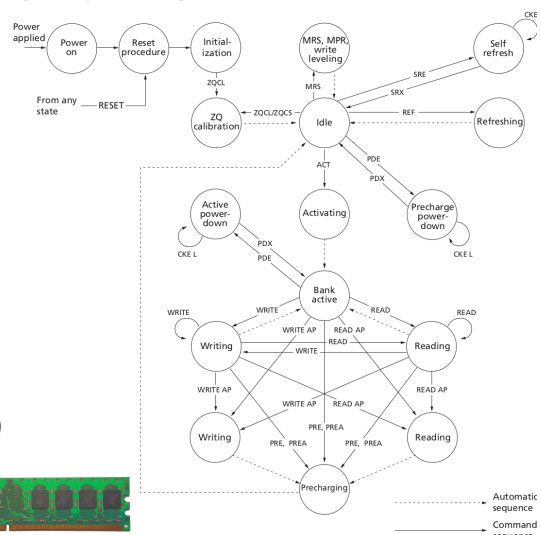
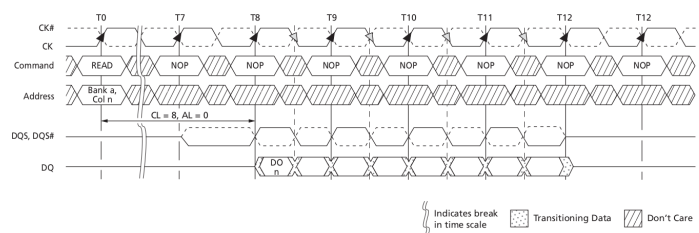
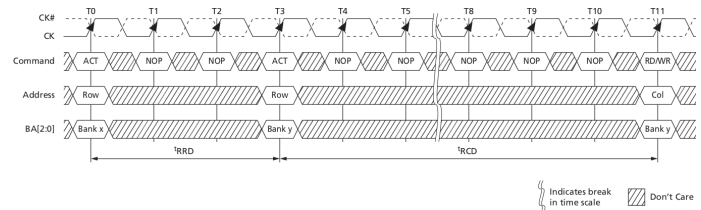


Figure 2: Simplified State Diagram



## DRAM, läsning

- Aktivera först rätt bank (del) av minne och välj rad
- Välj sedan kolumn i raden och gör läsning eller skrivning
- Notera burst av data (flera adresser i sekvens)
  - Hög genomströmningshastighet då
- Många klockcykler mellan varje kommando!
  - Ganska stor latency



## Egenskaper hos DRAM (DDR3 exempel)

- Lång tid start slumpmässig läsning/skrivning
  - 11+8 klockcykler
  - Klockfrekvens ca 500 Mhz => ca 2.5 M access/sekund
    - Hopplöst långsamt!
    - 2.5 GHz CPU => 1000 klockcykler i CPU per minnesaccess
  - Pipelined minne => Kan påbörja fler accesser
- Möjlighet läsa burst av data
  - Kan läsa många data i sekvens (8 i DDR3 exemplet)
    - Utan att behöva vänta mellan varje data
  - Kan få hög genomströmningshastighet
- Aktuell DRAM generation DDR5 (fler men snabbare klockcykler)

# Minnesegenskaper – flyktigt/icke flyktigt

- Flyktigt minne, förlorar informationen vid spänningsbortfall
  - engelsk term: volatile
  - Måste initieras/fyllas när strömmen slås på (t ex SRAM, DRAM)
    - Slumpmässigt/pseudoslumpmässigt innehåll när ström slås på
- Icke-flyktiga minnen, behåller värde även om strömförsörjning tas bort
  - T ex: ROM, EPROM, FLASH
- Måste finnas icke-flyktigt minne i en dator
  - Annars vet maskinen inte vad som ska göras när strömmen slås på
    - BIOS, bootsekvens, etc.
  - Brukar vara av ROM typ
    - FLASH vanligt (tillåter uppdatering av bootsekvens etc.)

# Andra typer av minne

- Lagring av information som magnetisk egenskap
  - Hårddisk, magnetiska media (dock inte som program och dataminne)
  - Ferromagnetiska minnen (FE-RAM), icke-flyktigt utan behov att raderas i block
  - Skrivbara DVD (kombinerar optisk och magnetisk egenskap)
- Seriella minnen: lång väntan på 1:a värdet, sedan ganska snabbt nästa värde i samma sekvens
  - Hårddisk (ej SSD), floppy disk (roterande media)
  - Kassetband, tape (används fortfarande för backup)

# Sammanfattande minneshierarki

- Primärminne (data och programminne)
  - RAM
    - Läs och skrivbara, Flyktiga, snabba, ganska dyra
    - SRAM snabbast, dyrast
    - DRAM långsammare, mer komplicerad, billigare per bit
  - ROM
    - Endast läsning av processorn, icke-flyktiga, snabba, varierande pris
    - FLASH är snabba, ganska billiga, och omprogrammerbara
- Sekundärminnen
  - Hårddiskar (mekaniska, SSD), flash-minnen
  - Tape, NAS (hårddiskbaserat), molntjänster (hårddiskbaserat)