TSEA28 Datorteknik Y (och U)

Föreläsning 15

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

- Fortsättning cache
  - Mer cacheexempel
- Bussar
  - Enkla delade bussar, PCI
  - Snabbare crossbar, PCI-express
  - ARM AXI-buss
- DMA
- Intro lab 5
  - Princip
  - Exempel

Praktiska kommentarer

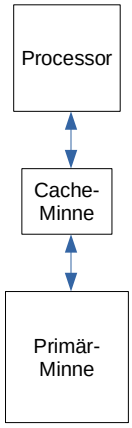
- Sorteringstävlingen deadline: Onsdag 8/5 kl 23:59.
  - Jag ska kunna ladda design, ändra värden i PM adress E0-FF, trycka på regs=0, nollställ klockcykler, och sedan Kont.
  - Sista maskinkodsinstruktionen ska vara ”HALT” från 1:a delen av labben.
  - Bidrag skickas via email till Kent.Palmkvist@liu.se

Praktiska kommentarer

- Laboration 5 förberedelser kan delvis 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

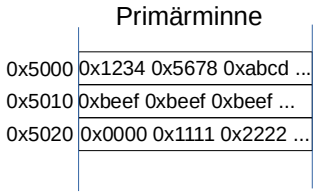
Cacheminnet

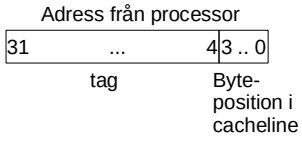
- Litet snabbt minne
  - All kommunikation till primärminnet går via cacheminnet
  - Innehåller kopior av minnesblock runt senaste läsningar och skrivningar
- Läsning
  - Returnera data om det redan finns i cache
    - Snabb läsning (1 till 2 klockcykler)
  - Om data saknas så hämtas det från primärminne
    - Långsam läsning (10-tal till 100-tal klockcykler)
- Skrivning
  - Spara kopia i cacheminnet

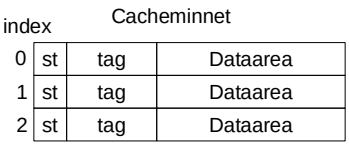


Cacheminnets struktur

- Multipla cachelinjer (cachelines)
  - Innehåller kopia av block från primärminnet
- Varje cachelinje i cacheminnet består av
  - Dataarea (kopia av primärminnet)
    - Vanlig storlek 16 eller 32 bytes (128 eller 256 bit per cacheline)
  - Märkarea (tag) anger adressen till kopians orginal i primärminnet
  - Statusbitar (st). Giltigt innehåll i dataarea, ändrad data etc.

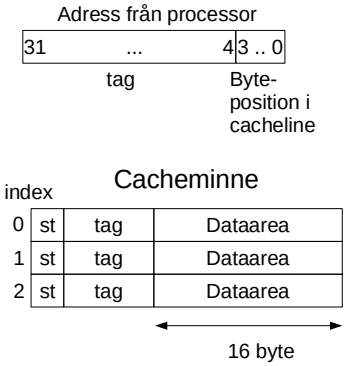






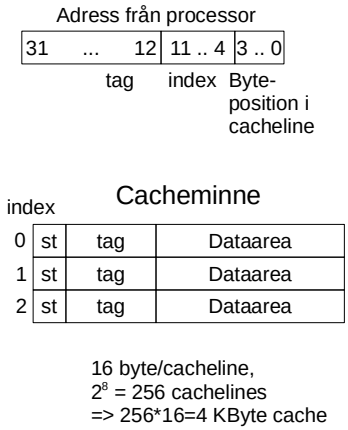
Fullt associativ cachestruktur

- Fördel
  - Enkel att använda (och förstå?)
  - Alla cachelines kan placeras var som helst i cacheminnet
    - Använder hela cacheminnet
- Nackdel
  - Dyr
    - Varje cacheline innehåller jämförelsehårdvara för att jämföra tag och adressens tag-del
    - Tag är lång (antal bitar i adress – 4 i exemplet ovan)
  - Långsam
    - Resultat från alla jämförelser ska samlas ihop innan beslut om cache-hit eller cachemiss



Direktaddresserad cache

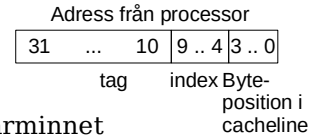
- Varje adress i primärminnet kan bara placeras på en plats i cache
  - Cache < Minne => flera olika minnesadresser placeras på samma plats i cache
- Del av adressen används för att bestämma index till cache
- Resten adressen sparas i tag för att ange vilken minnesline som finns i cache
  - Endast en jämförelse per minnesaccess: Addressens högsta bitar jämfört med lagrad tag
  - Enklare implementera, kan använda vanliga minnen
    - 1 jämförare för hela cache istället för 1 jämförare för varje cacheline

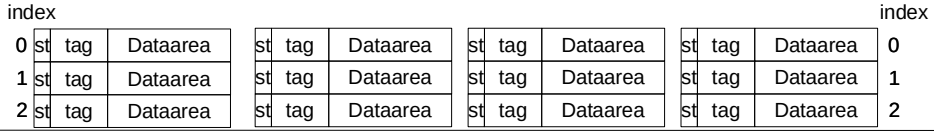


16 byte/cacheline,
2^8 = 256 cachelines
=> 256*16=4 KByte cache

Gruppassociativ cache – en kompromiss

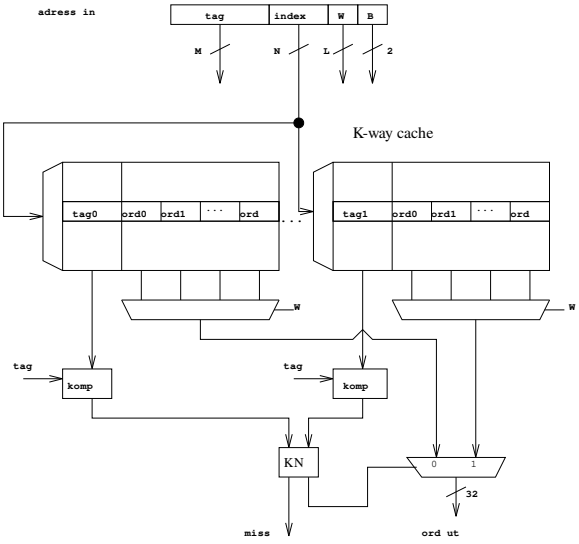
- Låt varje adress i primärminnet kunna placeras på flera ställen i cache
- Exempel
  - I en fyrvägs gruppassociativ cache kan en viss adress i primärminnet placeras i en av fyra cachelines (kallat olika vägar)
  - I princip 4 direktadresserade cache placerade bredvid varandra, kräver 4 jämförelser per åtkomst i cache
- I exemplet ovan skulle tag öka i längd jämfört med direktadresserad cache
  - 22 bitar till tag, 6 bitar för index till cacheline, 4 bitar för byteposition i cacheline





Gruppassociativ cache – implementation

- Från lab-manualen
- Byteadress W+B bitar
  - Levererar 32-bitars ord
    - B-bitarna används inte
- Två SRAM inritade som lagrar tag och dataarea (+ dirty/valid flaggor)
- Komp testar om tag lika med utläst tagX
- KN är en logisk funktion
  - avgör träff/miss
  - vilket minne som gav träff



Fler exempel på cacheeffekter

- Beräkna C = A · B
  - Matris A lagrad i primärminnet
    - Normal eller transponerad (rad först eller kolumn först)
  - Vektorer B och C lagrad i primärminnet

$$A \cdot B = \begin{pmatrix} a_{00} & \cdots & a_{0(n-1)} \\ \vdots & \ddots & \vdots \\ a_{(n-1)0} & \cdots & a_{(n-1)(n-1)} \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} a_{00} \cdot b_0 + \ldots + a_{0(n-1)} \cdot b_{n-1} \\ \vdots \\ a_{(n-1)1} * b_1 + \ldots a_{(n-1)(n-1)} * b_{n-1} \end{pmatrix}$$

- Matrisstorlekar
  - 2000x2000, 2048x2048, 2092x2092
    - n=2000, 2048 respektive 2092

Matrismultiplikation

- Kod för matrismultiplikation (kolumnvis lagrad matris)

```
for(i=0; i < MATRIXSIZE; i = i + 1) {
    float result = 0;                    // Deklarera variabeln result
    for(j=0; j < MATRIXSIZE; j++) {
        result = result + a[j][i]*b[j];
    }
    c[i] = result;
}
```

- Resultat på min förra laptop (i7-4702MQ@2.2GHz)
  - Kod finns på föreläsningssidan (mult.c)
  - N = 2000 tid: 13.7 ms
  - N = 2048 tid: 40.7 ms
  - N = 2096 tid: 14.1 ms
- Tid större för 2048 trots att 2096 utför fler beräkningar!

i7-1165G7@2.8GHz

N = 2000 tid 8.3 ms
N = 2048 tid 21 ms
N = 2096 tid 9.8 ms

N = 4000 tid 36 ms
N = 4096 tid 127 ms
N = 4192 tid 41 ms

Varför är 2048 långsammare än 2056?

- i7-4702MQ cachestruktur (L2)
  - 256KB 8-vägs gruppassociativ, 64 byte cacheline => 6 bitar väljer byte
  - 256*1024/8=32768 byte/väg
  - 32768/64=512 rader => 9 bitar index
- Accessmönster för 2048:
  - 2048 = 0x800, 4 byte per element => 0x2000 mellan varje adress
    - $0x2000 = 0|010\ 0000\ 00|00\ 0000_2$
  - 0x0000,0x2000,0x4000,0x600,0x8000,0xA000,…
    - Samma index efter ett tag (256 KB 8-vägs => 256*1024/8=32768 byte/väg => 0x8000 => samma index efter 4 läsningar)
    - Alla vägar fyllda efter 8*4=32 läsningar => får inte plats med nästa läsning
    - Kastar bort cacheline data innan data använts (varje cacheline får plats med 64/4=16 värden => skulle vilja få data ligga kvar under 16 yttervarv).

Varför är 2048 långsammare än 2096?

- Fallet 2096 (14.1 ms) hamnar inte rader på samma index så fort
  - 2096 = 0x830, 4 byte per värde => 0x20C0
    - Adress $0x20c0 = 0|010\ 0000\ 11|00\ 0000_2$
  - 0x0000, 0x20C0, 0x4180, 0x6240, 0x8300, ...
    - Olika index under 256 läsningar, 257:e läsning => 0x101*0x20C0 = 0x20E0C0
      - Adress $0x20e0c0 = 0010\ 0000\ 1|110\ 0000\ 11|00\ 0000_2$
    - 257:e läsning placeras i en annan väg (finns 8 vägar) => får fortfarande plats
- Hittar nästan alla inlästa data i cache efter 1:a inre varvet (alla i-värden) => nästa varv hittar data i cache (upp till 15 varv).
  - Kan hitta inläst data för nästa varv i loop!
  - Data på adress 0x20C0 (a10) redan inläst (64 byte cacheline) etc.

Transponerad Matrismultiplikation

- Kod för matrismultiplikation

```
for(i=0; i < MATRIXSIZE; i = i + 1) {
    float result = 0;                       // Deklarera variabeln result
    for(j=0; j < MATRIXSIZE; j++) {
        result = result + a[i][j]*b[j];     // Ändrat a index (förra hade a[j][i])
    }
    c[i] = result;
}
```

- Resultat på min förra laptop (i7-4702MQ@2.2GHz)
  - N = 2000 tid: 11.5 ms
  - N = 2048 tid: 12.0 ms
  - N = 2096 tid: 12.6 ms

Fuskade lite, gjorde inte transponering

- Lägg till transponering innan multiplikation

```
for(i=0; i < MATRIXSIZE; i = i + 1) {
    float tmp = 0;                  // Deklarera variabeln tmp
    for(j=i; j < MATRIXSIZE; j++) {
        tmp = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = tmp;
    }
}
```

- Resultat på min förra laptop (i7-4702MQ@2.2GHz)
  - N = 2000 tid: 9.3 ms
  - N = 2048 tid: 39.6 ms
  - N = 2096 tid: 10.2 ms
  - Inte så mycket vinst denna gång jämfört med ej transponerad

Bättre transponering

- Transponera små block

```
for (i = 0; i < MATRIXSIZE; i += BLOCKSIZE) {
    float tmp = 0;
    for (j = 0; j < MATRIXSIZE; j += BLOCKSIZE) {
        // transpose the block beginning at [i,j]
        for (k = i; k < i + BLOCKSIZE; ++k) {
            for (l = j; l < j + BLOCKSIZE; ++l) {
                tmp = a[k][l];
                a[k][l] = a[l][k];
                a[l][k] = tmp;
            }
        }
    }
}
```

- 2000: 19.2 ms när BLOCKSIZE=8
- 2048: 29.9 ms när BLOCKSIZE=8
- 2096: 21.2 ms när BLOCKSIZE=8

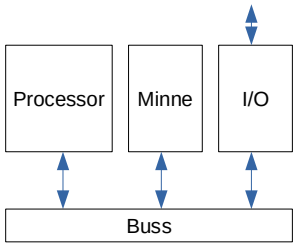
Mer avancerade egenskaper hos cache

- Cache förhämtning (prefetch)
  - Cache ”gissar” vad som ska hända härnäst
    - Instruktioner läses oftast i sekvens
  - Cache läser instruktioner i förväg
  - Är exempel på spekulativt beteende
- Förhämtning har för och nackdelar
  - Kan ge snabbare läsning
    - Cacheträff trots att det är första gången en adress i cacheline efterfrågas
  - Kan långsammare läsning
    - Buss/minne upptaget med förhämtning => kan inte läsa någon annan adress samtidigt
- Hjälper inte om programmet gör mycket hopp
  - Cache vet inte vad instruktionerna betyder

Bussar

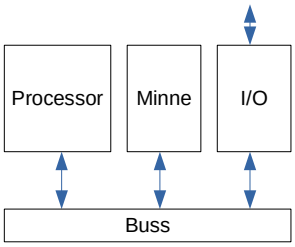
Bussar

- Kopplar ihop olika komponenter
  - Samma princip som bussar inuti processorn
  - Delad resurs, något enhet måste vara den som styr
- Koppla ihop processor med övriga enheter
  - Minne (av olika typer: ROM, RAM etc.)
  - I/O enheter (Serieport, Tangentbord, Hårddiskkontroller, Relä/sensorer etc.)
  - Tillåter utbyggbar design med t ex instickskort
- Bussens exakta definition varierar
  - Varje fabrikant försöker ofta behålla samma interface för alla kretsar i samma familj
    - MC68000
    - ARM – AXI
    - IBM PC - ISA



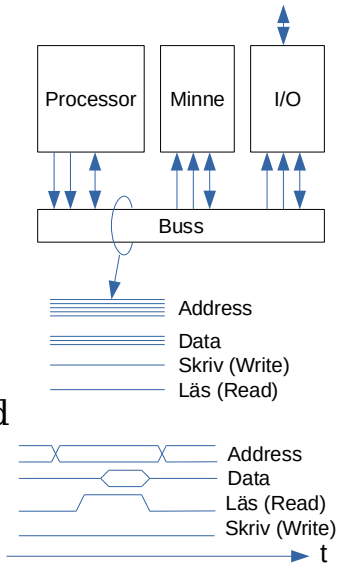
Bussar

- Informationsflödet på en buss
  - Address att läsa från/skriva till
  - Data till/från enhet
  - Styrsignaler (lässignal, skrivsignal)
- Oftast processorn som styr kommunikationen över bussen
  - Läsning/skrivning till adresser i minnet/IO-enheter



Enkla bussar

- Enklast: kopiera interna bussen mellan processor och programminnet
  - Addressbuss (address att läsa/skriva)
  - Databuss (värdet att läsa/skriva)
  - Styrsignaler
- Varje enhet avkodar Address och styrsignaler
  - Addresserad enhet får skriva data på databussen
- Processor förväntar sig svar inom fördefinierad tid
  - Ingen kontroll om address är korrekt/tillåten
  - Maximal svarstid bestäms ofta av klockfrekvens
  - Fungerar bra mot statiska RAM och ROM

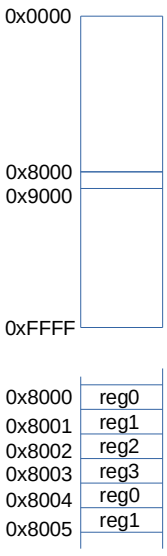


De första versionerna (ca 1970-1980)

- Varje processorfamilj hade sin egen buss-standard
- Mål var att kunna använda så få pinnar och ledningar som möjligt
  - Fler pinnar och ledningar => högre pris
  - Billigare kontaktdon för expansionskort
- Minnen (oftast SRAM och ROM) ungefär lika snabba som processorn
  - 1 läsning/skrivning till minne per klockcykel
- Flera minnen anslöts till samma buss (ROM+SRAM)
  - Addressavkodning mha högsta bitarna i minnesaddressen
- Dubbelriktade Databussar
  - Data för läsning och skrivning på samma pinnar

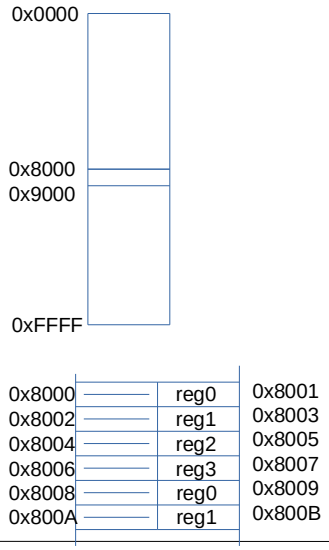
Bussar, addressavkodning

- Varje enhet avkodar Address och styrsignaler
  - Dela upp addressområdet mha MSB bitar i address
- Använd MSB bitar i address för att dela upp adressarea
  - Så får bitar som möjigt används
    - Snabbare och billigare
- Exempel: Enklare 8-bitars I/O enhet med få adresser (t ex enklare GPIO)
  - Placera in en I/O enhet med 4 register i addressrymd med 16 bitars adress med start på adress 0x8000
  - Använd 4 bitar för att dela upp adressrymd
    - Varje område består av 4096 adresser 2^(16-4)
    - Varje register finns på 1024 olika adresser



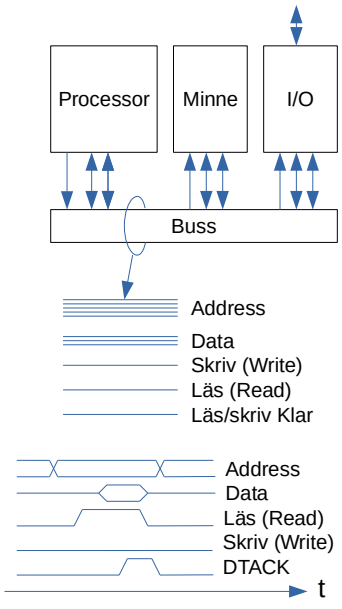
Databussbredd vs adress

- Om databuss bredare än register kan inte alla databussbitar anslutas
  - Rita addressmappen med ordbredd
    - Exempel 16 bit: Udda byteadresser till höger, jämna till vänster
- Exempel: enhet kan bara leverera 8 bitar åt gången, buss förväntar 16 bitar
  - Ignorera övriga databitar
  - Ser ut som registren utspridda (t ex endast placerade på udda adresser)
  - D0-D7 ansluts till I/O-enhet, D8-D15 ansluts inte alls
  - Minnesmapp ger tomt innehåll på jämna adresser
    - Läsning ger skräp, skrivning påverkar inget



Bussar, mer avancerad timing

- Svarstid kan variera mellan olika minnen och/eller I/O enheter
  - Lösning: Varje enhet får indikera att data finns tillgängligt
  - I Darma (lab-systemet) fås avbrott om läsning i minne som inte svarar (som inte finns)
- Ytterligare kontrollsignaler behövs
  - MC68000: DTACK anger data mottagits/sänts
  - Mikroprogrammet i processorn måste hantera långsamma minnesaccesser
  - En form av handskakning



Nackdelar med enkla bussar

- Endast en överföring per gång
  - Andra aktiviteter tvingas vänta
- Fler anslutningar => långsammare buss
  - Elektriskt svårare att ändra nivåer på ledningarna
- All kommunikation styrd av processorn
  - Processorn måste hantera all dataflytt
- Viss kommunikation skulle kunna hanteras separat
  - Skapande av skärminnehåll
  - Flytta data mellan minnesarea och I/O enhet
    - Ex: ljudkort, nätverksdata, hårddiskdata

Hantering av I/O

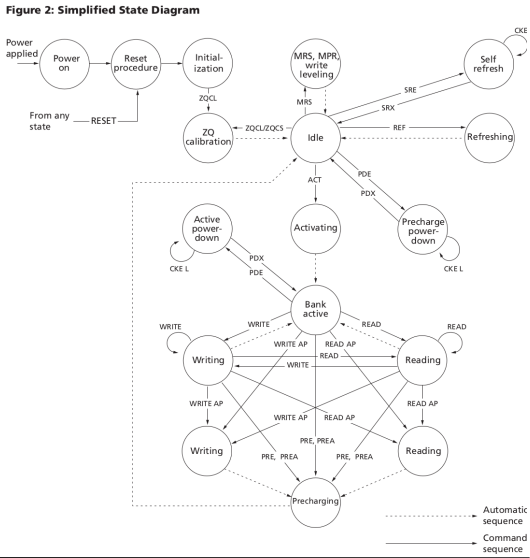
- Programmerad I/O (lab1 på Darma)
  - Processor väntar aktivt (busy wait) på nytt data
  - Enkel att programmera, väldigt ineffektiv
- Avbrottsstyrd I/O (lab3 på Darma)
  - Avbrott indikerar nytt data
  - Effektivare, men ganska ineffektivt om mycket data
  - Måste fortfarande hämta och avkoda instruktioner för flytt av data
- Direkt Memory Access (DMA)
  - Låt I/O-kontrollern själv skriva direkt i minnet
  - I/O->minne istället för I/O->CPU->minne

DMA – Bustillgång

- I/O-enhet fungerar som egen processor
  - Kan flytta data i hastighet definierad av buss istället för processor
- Kräver att processor kan överlåta kontroller över bussen till annan enhet
  - Arbitrering av buss
    - Avgör vem som kontrollerar bussen om flera DMA-enheter vill kontrollera bussen samtidigt
- Processor bestämmer fortfarande vad som ska hända
  - Konfigurera DMA med startadress, längd, generering av avbrott när överföring klar etc.

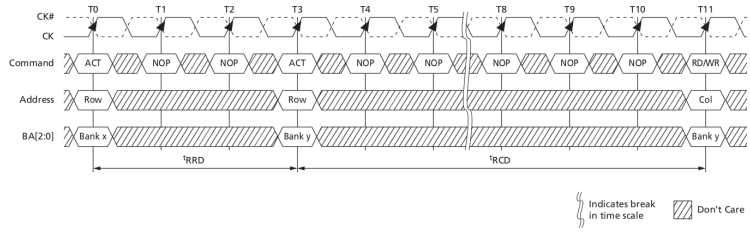
Repetition minnestiming för DRAM

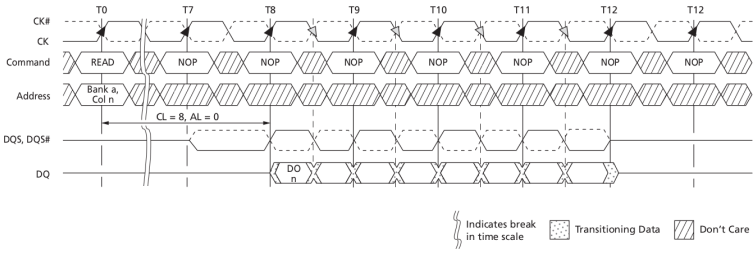
- Dynamiskt minne är mer komplicerat att kommunicera med
  - Varje minne har en kontroller som styr refresh och kommunikation
  - Ett protokoll med ca 25 kommandot
  - Ex: Datablad till Micron 256Mbit DDR3 minne är 211 sidor långt
- Mål med minnesdesign
  - Få pinnar, låg effektförbrukning
  - Hög genomströmningshastighet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



DRAM, läsning

- Aktivera först rätt bank av minne och välj rad
- Välj sedan kolumn och gör läsning/skrivning
  - Notera burst av data (flera i sekvens)
- Stor latens
  - Tid från rad skickad tills data mottagen
- Stor genomströmnings-hastighet när data väl kommer



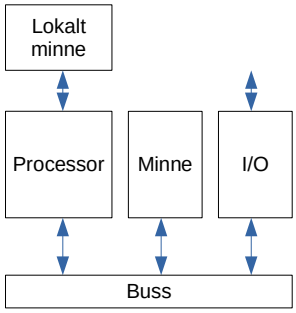


Egenskaper hos DRAM (DDR3 exempel)

- Lång tid start slumpmässig läsning/skrivning
  - 11+8 klockcykler
  - Klockfrekvens ca 500 Mhz => ca 2.5 M access/sekund
  - 2.5 GHz CPU => 1000 klockcykler i CPU per minnesaccess
- 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
- Matchar läsning/skrivning i cache
  - Hel cacheline kan hämtas eller skrivas i en överföring

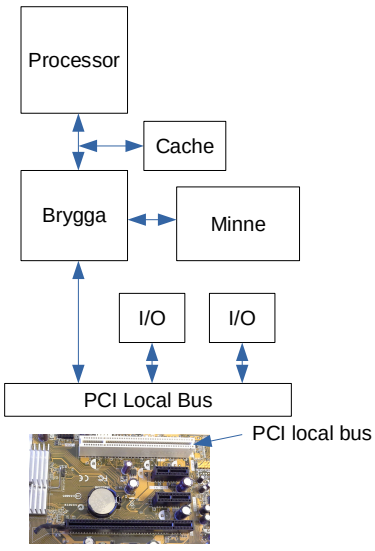
Alternativ till enkel buss

- Lägg till möjlighet att göra flera saker samtidigt
  - Lägg programminnet närmare processorn
  - Cache hjälper minska antal accesser till bussen
- Dra fördel av egenskaper hos dynamiska RAM, DMA överföringar samt cache
  - Ofta sekvensiell läsning/skrivning av flera adresser (burst)
  - Cacheline => 16 eller 32 adresser i sekvens
  - DMA => långa sekvenser, t ex disksektor 4096 byte
- Skicka inte ut processorns egna signaler
  - Sätt in en styrkrets emellan (kallad en brygga)



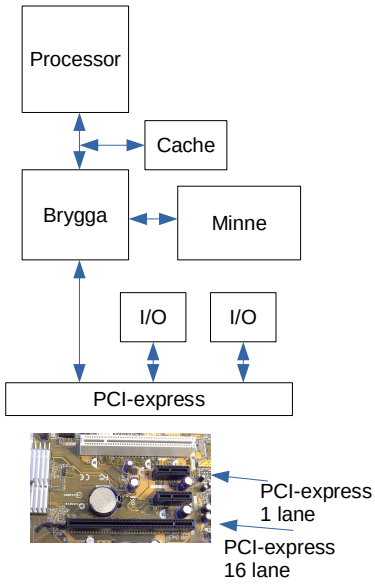
PCI bussen

- Används av fler processorfamiljer än x86 (laptop)
- Multiplexad parallellbuss med burst stöd
  - Data skickas parallellt på samma ledningar som address (kontrollsignaler anger vilket som skickas/tas emot)
  - Kan skicka sekvenser av data (burst)
  - 33 Mhz klockfrekvens
  - Data får buffras på vägen från avsändare till mottagare
  - Överföringar kan avbrytas och fortsättas senare
- Varje ansluten enhet kan identifieras
  - Unikt tillverkar ID och krets ID kan hämtas från varje anslutet expansionskort



PCI-express bussen

- Seriell punkt till punkt överföring
  - Tillåter högre överföringshastighet än parallell överföring
  - Full duplex (sända och ta emot data samtidigt)
  - Varje lane (två seriella anslutningar) kan hantera 250MB/s i varje riktning (ökar i senare versioner av standarden)
  - Flera lane kan kombineras (t ex 16 lane till grafikkort)
- Data paketeras på liknande sätt som i vanliga datornät
  - 12 till 16 byte header + data
  - All info (avbrott, data, adresser etc.) placeras i dataarean



Fördelar med avancerade bussar

- Tillåter processorn arbeta (via t ex cache) utan att störas av andra transaktioner på bussen
  - T ex DMA överföring mellan I/O och primärminne
- Byte till snabbare/långsammare processor eller annan processorfamilj påverkar inte bussens funktion och timing
- Vissa typer (t ex PCI-express) kan tillåta byte av fysiskt gränssnitt utan att påverka hur bussen fungerar logiskt
  - Endast fysiska lagret påverkas vid byte av media
- Kan tillåta flera samtidiga transaktioner

Nackdelar med avancerade bussar

- Mer komplicerat lägga till enheter till bussen
  - Måste stödja protokoll och signaler
- Ökad fördröjning (latens) vid läsning/skrivning
- Exempel: Sätt bit 4 i I/O register X på address AdrX
  - Sekvens: Läs AdrX till internt register i processor
    Sätt bit 4 i internt register
    skriv tillbaks registervärde till AdrX
  - Med en avancerad buss tar detta lång tid (två minnesaccesser som inte kan placeras parallellt
- Alternativ lösning: Separata register för sätta 0 respektive 1 på port (2 olika adresser)
  - Skriv 0x10 till AdrX_1 sätter bit 4 till 1
    - Bitvis OR mellan värde och aktuellt register
  - Skriv 0x10 till AdrX_0 nollstället bit 4 till 0
    - Bitvis AND mellan inverterade värdet (dvs not värde) och aktuellt register

Alternativ buss: AXI (del av lab5!)

- ARM-processorer använder bl a denna buss
  - Advanced eXtensible Interface
  - I laboration 5 tittar ni närmare på version 3 av denna buss
  - Börjar bli vanlig med andra processorer också
- Vanligast på ett och samma chip
  - Ej för att kommunicera mellan olika expansionskort
  - På chip är det inga problem med många anslutningar/ledningar i bussen
- Uppbyggd av flera kanaler
  - Hantera adress och data med separata styr och kontrollsignaler
  - Var sin adressbuss för läsning respektive skrivning

Några begrepp

- Transaktion
  - En fullständig sekvens av överföringar för att genomföra en läsning eller skrivning
  - Läsning: Skicka adress
    Få tillbaks data från adressen
  - Skrivning: Skicka adress
    Skicka data
    Få tillbaks indikering om skrivning gick bra
- Kanal
  - En uppsättning signaler som skicka information i en riktning
    - T ex adress från processorn till minne eller I/O enhet

Finesser med AXI version 3

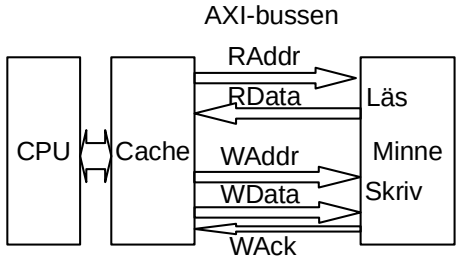
- Stöd för att specificera burst skrivning/läsning
  - Kan skriva/läsa flera ord per transaktion (överföring)
- Stöd för läsning av cachelines med ”critical word first”
  - Läs efterfrågat ord så fort som möjligt, dvs läs cacheline i fel ordning
  - Ex: Läs adress 0x80002C om varje cacheline innehåller 16 byte
    - Fyll på byte adress i ordning C,D,E,F,0,1,2,3,4,….9,A,B
- Pipelined
  - Kan begära nästa läsning innan data från föregående läsning kommit

Finesser med AXI version 3, forts.

- Stöd för Write Strobes
  - Möjliggör skrivning av enskild byte trots databuss bredare än 1 byte
  - Skriv inte alla bitar som skickas över databuss
  - Ex skriv byte adress 0x80003, databuss 16 bitar
    - Skicka byte på databuss, ange med writestrobe0 = 0 och writestrobe1 = 1 att bara andra byte på databuss ska skrivas
    - Vanligt 16 bitars skrivning har båda strobe = 1
- Kan tillåta returnerad data i fel ordning
  - Begär läsning adress 0x80000 och sedan från 0x90000
    - Kan få data från 0x90000 innan data från 0x80000
  - Kräver att sändare och mottagare stödjer finessen
  - Behöver någon form av identifiering av varje läsning/skrivning (ID)

Blockschema för AXI-bussen

- Var sin läs respektive skrivbuss
  - Address och data för läsning
  - Address, data och acknowledge för skriv
- Ytterligare styrsignaler i varje delbuss
  - Handskakning
  - ID-nummer
  - Burst-typ
  - Status



AXI3-bussen, vanliga signaler

- *ADDR: Adress
- *DATA: Data
- Handskakningssignaler
  - *VALID: Sändare har information
  - *READY: Mottagare redo att ta emot
  - Både *VALID och *READY måste vara 1 för att aktivera överföring
  - *LAST: Makerar sista ordet i transaktionen

AXI3-bussen, vanliga signaler, forts.

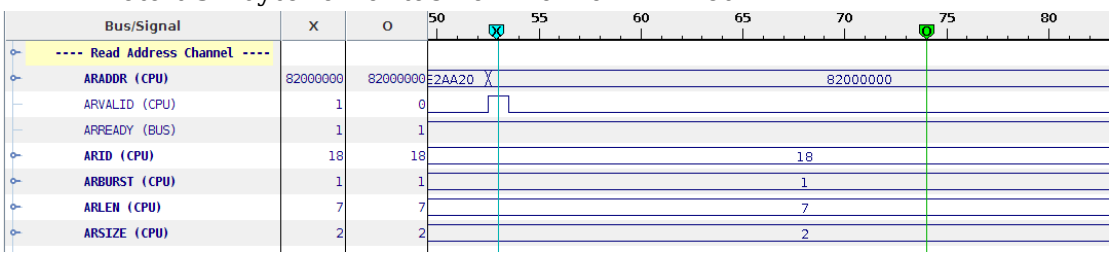
- *LEN och *SIZE: Transaktionens längd
  - Hur breda ord (SIZE), 1 motsvarar 2 byte
  - Antal ord (LEN)
  - Notera offset 1: 0 motsvarar 1 ord, 1 motsv. 2 ord etc.
- *BURST: Typ av burst
  - 00 : Specificerad adress används i hela transaktionen
  - 01: Adressen ökar hela tiden
  - 10: Cirkulär (används för att fylla en cacheline, med stöd för critical word first)
- *ID: Transaktioner med samma ID måste hanteras i den ordning de anländer

AXI3 signaler vid läsning, read adress channel

- Starta läsning genom att skicka önskad adress att läsa till bussen
- ARADDR[31:0]: Adress vi vill läsa ifrån
- ARVALID: Master vill börja en lästransaktion
- ARREADY: Slave är redo att ta emot en lästransaktion
- ARLEN/ARSIZE: Specificera antal ord att läsa
- ARBURST: Type av burst-läsning (cirkulär, linjär, fix)
- ARID: Master skickar identifikationsnummer
- (Plus några till)

AXI3 signaler vid läsning, read adress channel

- Klockcykelskala längst upp (100 MHz klockfrekvens)
- ARVALID hög under 1 klockcykel startar läsning på adress $82000000, vid tidpunkt 53.
  - ID = $18, SIZE=4 byte/ord (dvs klockcykel), LEN=8 ord, BURST=ökande adress
  - Totalt 32 byte förväntas komma från minnet

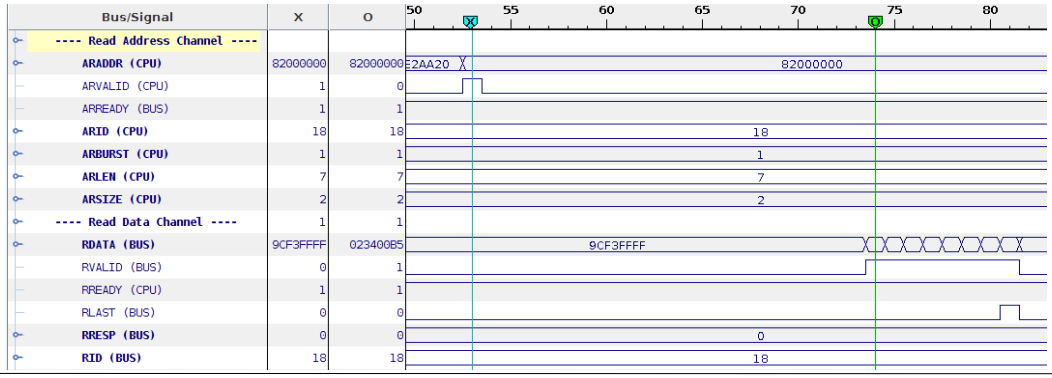


AXI3 signaler vid läsning, Read data channel

- Denna kanal innehåller data som lästs
- RDATA[31:0]: lästa ord från minnet
- RVALID: Slav vill börja skicka data
- RREADY: Master är redo att ta emot data från en lästransaktion
- RLAST: Indikerar när sista ordet skickas
- RRESP: Slav skickar information om status (genomfördes läsningen korrekt)
- RID: Slav skickar identifikationsnummer

AXI3 signaler vid läsning, read data channel

- 8 ord med 4 byte/ord, start tidpunkt 74 (RVALID=1 och RREADY=1)
- RRESP = 0 => allt ok. ID ska matcha adressens ID (RID = AID)
- RLAST indikerar slutet på dataöverföringen



AXI3 signaler vid skrivning, Write adress channel

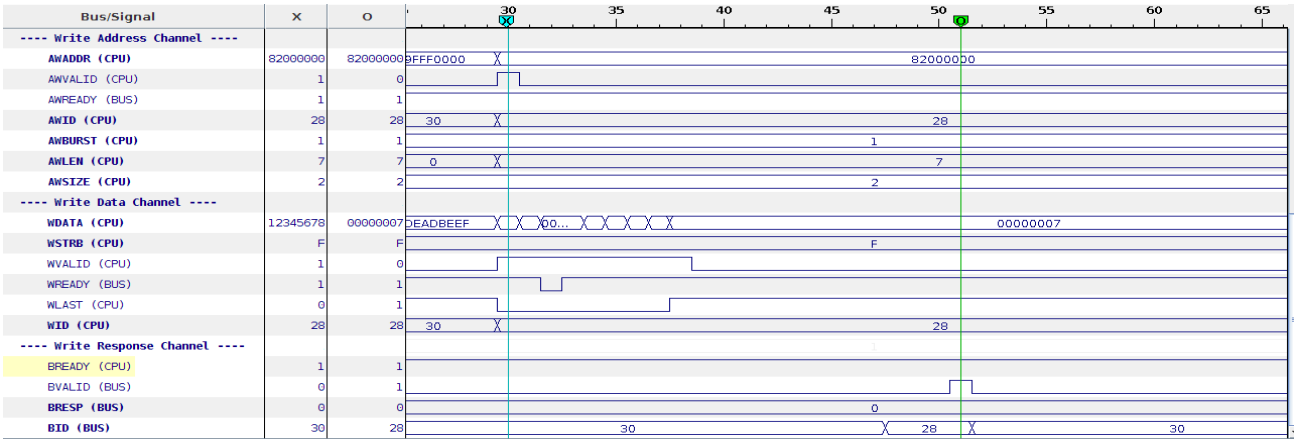
- Skickar adress som skall skrivas till bussen
- AWADDR[31:0]: Adress att skriva till
- AWVALID: Master vill börja en skrivtransaktion
- AWREADY: Slav är redo att ta emot en skrivtransaktion
- AWLEN och AWSIZE: Transaktionens längd och bredd
- AWBURST: Type av burst-skrivning (cirkulär, linjär, fix)
- AWID: Master skickar identifikationsnummer

AXI3 signaler vid skrivning, Write data channel

- Skickar datat som ska skrivas till bussen
- WData[31:0]: Data att skriva till minnet
- WSTRB[3:0]: Vilka byte i databussen innehåller data
- WVALID: Master lägger ut data på WData
- WREADY: Slav är redo att ta emot data
- WID: ID för denna överföring
- WLAST: Indikera sista ordet som ska skrivas

Exempel på skrivning till minne

- Notera pausen i burst vid 32:a klockcykeln (WREADY=0)



AXI3 signaler – Write response channel

- Skickar information från buss till processor om skrivningen gick bra
- BVALID: Slave har ett svar på en skrivning
- BREADY: Master redo att ta emot svar på en skrivning
- BID: Slave returnerar identifikationsnumret
- BRESP: Lyckades skrivningen

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?

Lab 5, beskrivning av uppgifterna

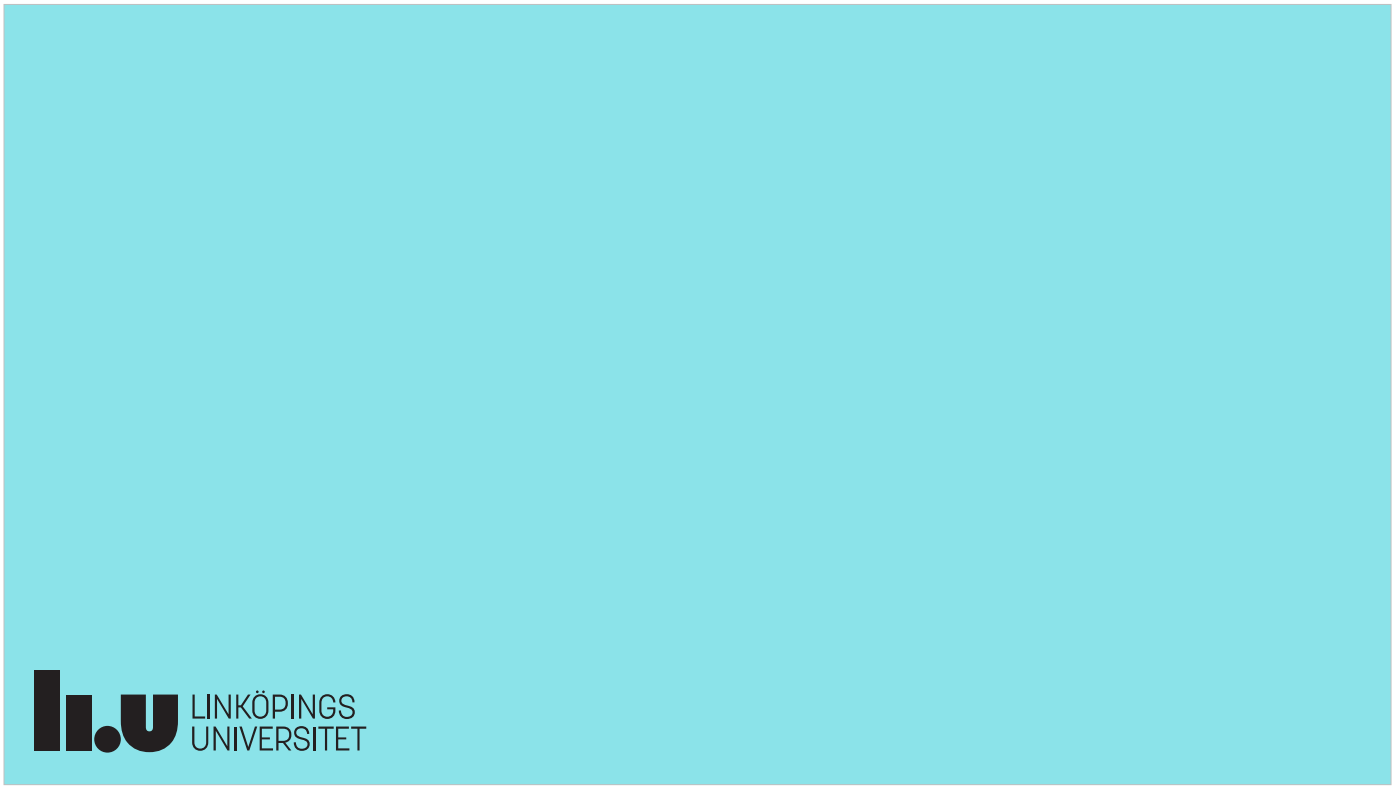
- Uppgift 2: Ta reda på datacachens associativitet
  - Skriv litet program som läser en uppsättning data från minnet flera gånger (3 gånger).
    - Målet är att alla data ligger i minnet på adresser som ska placeras på samma cacheline i cache men med olika tag.
  - Titta på bussens läsningar
    - När 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, beskrivning av uppgifterna

- Uppgift 3: Visa del av karta på skärm
  - Skriv om en kopieringsrutin så den går fortare
- Scenario: En ruta ur en karta ska kopieras in i ett bildminne
  - Rutan kan roteras och flyttas
    - Bildminnet fylls rad för rad med kopia av data från karta
    - Olika sekvenser på minnesadresser beroende på rotation och startposition
  - Läsningen från början är enkel och långsam
    - För långsam vid vinklar 90 och 270 grader
    - Ta reda på varför (varför fler cachemissar i ovanstående fall)
  - Mät med chipscope
    - Titta på adresser som läses ut (addresshopp när nästa rad läses)

Lab 5, beskrivning av uppgifterna

- Uppgift 3: Modifiera utläsningen
  - Läs delbilder istället (se avsnitt 8.5) (kolumner)
  - Mät prestanda för den modifierade koden
    - Rad motsvarar delbildens radlängd (kolumnbredd)
  - Glöm inte ta bort chipscope trigger funktion när prestanda hos modifierad kod testas


LINKÖPINGS
UNIVERSITET