Eksamen 2012  
3. semester  
I3MPS - Microprocessor systemer  
Kasper Nissen

**Spørgsmål 1**

**Cache**

**Spørgsmål**

* Hvad er et memory hierarki?
* Hvad er en cache- og hvad bruger vi den til?
* Hvordan er en cache opbygget?
* Hvilke begrænsninger har en cache og hvordan kan vi minimere disse?
* Hvordan kan man som programmør optimere brugen af cache?

**Pensum:** Text: ARM System Developer’s Guide, Sloss/Symes/Wright: 12-12.32 (Cache) + 14-14.5 (MMU) +<http://people.redhat.com/drepper/cpumemory.pdf> ch 6 -> side 50 linie 7 +<http://tldp.org/LDP/tlk/mm/memory.html> 3.1-3.6

**Exercise 10**

**Hvad er en Cache? (undgå langsom forbindelse til hukommelse)**

* Meget lille og hurtig hukommelse
* En bro mellem hurtig og langsomme enheder -> f.eks. L1 -> bro mellem processor og hukommelse

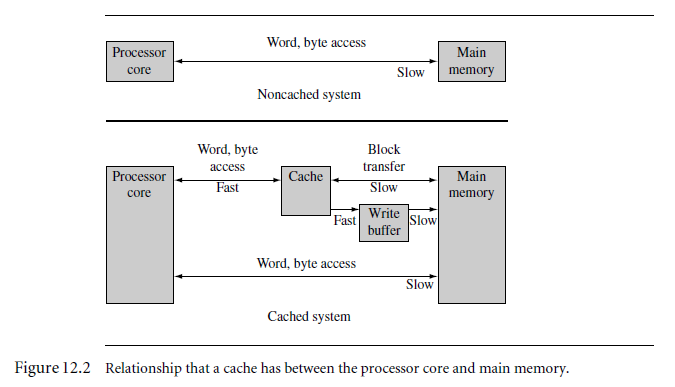
Cache er et **lille datalager mellem CPU’ens kerne og det vi kalder main memory**. Processoren bruger cachememory i stedet for main memory så snart det er muligt, for at kunne optimere systemets performance. **Målet med cachen er at reducere hukommelses flaskehalsproblemer** som ophober sig i kernen p**ga. af for langsom hukommelse.**

Ofte bliver der sammen med cache brugt en **write buffer**, som er en meget **lille first-in-first-out(FIFO)** hukommelse, som også er placet mellem processorens kerne og main memory**. Formålet med write bufferen er at frigøre processorens kerne** og main memory fra de langsomme skrive tider som er kendetegnet for main memory.

Ordet cache er fransk og betyder “**a concealed place for storage**”. Når vi ser på embeddede systemer som ARM (devkit8000 f.eks.) så er denne definition meget præcis. Det kommer til udtryk i og med at cache hukommelsen **og write bufferen er designet til at være ”usynlige” for software(kode**). Det vil også sige at kode skrevet til et system uden cache, ikke nødvendigvis behøver at blive omskrevet når det afvikles på en processor med cache.

Virker pga. typisk **små loops af data lokalt.**

**Forskel på et system med cache og et uden**



Som det ses på figuren så viser det øverste system et system uden cache, her bliver main memory tilgået direkte fra processorens kerne. Systemet nedenunder viser et system med cache. Cache memory er meget hurtigere end main memory, og vil derfor respondere meget hurtig til f.eks. data requests fra kernen. Cachens forhold til main memory består af overførsler at små blokke af data mellem den langsomme main memory og den meget hurtigere cache hukommelse. **Disse blokke kaldes også for cache lines**. Som fortalt tidligere så agere write bufferen som en midlertidig buffer som frigiver tilgængelig plads i cachens hukommelse. Cachen overfører altså en cache line til write bufferen ved høj hastighed hvorefter write bufferen sender det videre til main memory ved en langsom hastighed.

**Er der ”bivirkninger” på et system med cache?**

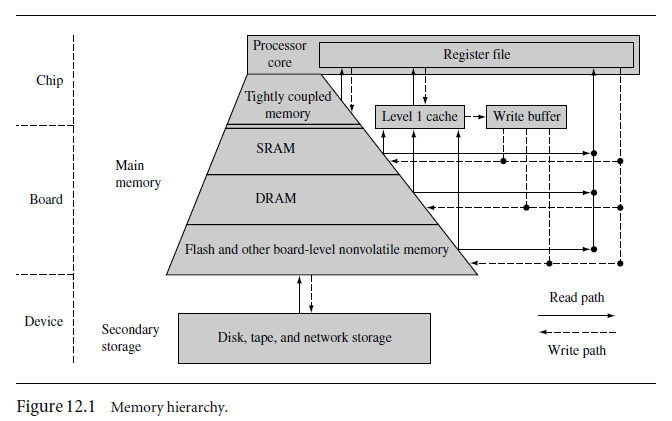
**Ja,** et af de største problemer med brugen af et system med cache er, at det er utroligt svært at vide hvor hurtigt et program afvikles. Et programs afviklingstid vil også varier fra gang til gang, da vi ikke kender til hvad er er ligger i cachen på det givne tidspunkt.

**Level 1 og 2 cache**

På den før viste figur, var der kun afbildet level1 cache, som beskrevet er den et high-speed, on-chip memory som midlertidig holder kode eller andet data fra et laver niveau i hierakiet.

Level 2 cache er placeret mellem L1 cache og noget langsommere hukommelse. Disse 2 typer er også kendt som **primary og secondary caches.**

**Hvad er et memory hiraki?**



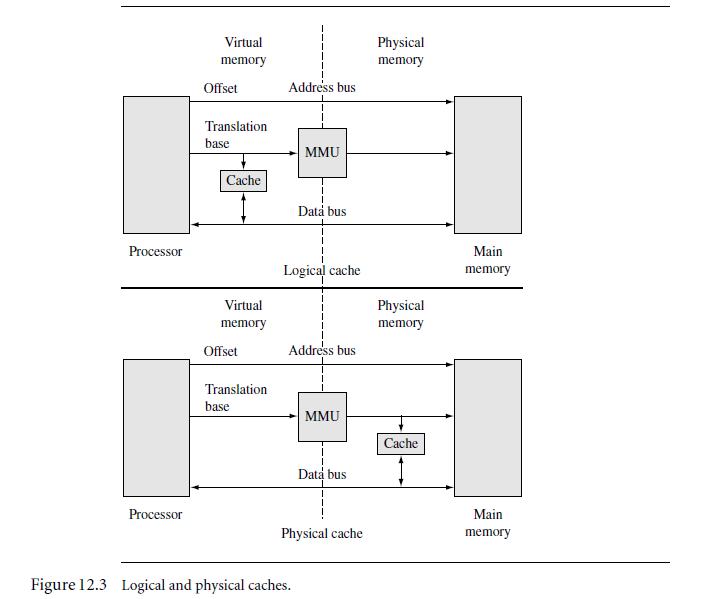
Et memory hierarki er en **oversigt over hvordan processoren kommunikere** med de **forskellige slags hukommelser.** Som det ses på figuren, så har vi jo processoren øverst. Processoren indeholder en **meget tæt koblet hukommelse, som er kendt under navnet register file**. Disse registrer den **hurtigste form for hukommelse** i et sådan system. Det **næste niveau** i hierakiet er det der kaldes **Tighly coupled memory(TCM**) og det er her vi finder **level 1 cache og write buffer**. Derunder har vi **så main memory**, som er komponenter som SRAM, DRAM og flash, som har det formål at holde programmer mens de kører på systemet. Det sidste led som ses på figuren er det der kaldes **secondary storage**, og det er her vi f.eks. **finder de langsomme enheder** som harddiske eller lign.

**En cache kan indsættes mellem hvert niveau(level)** i et sådan memory hierarki. En cache kan forbedre et systems performance. Man kan sige at cachens opgave er at tage noget information lagret i et lavere niveau i hierakiet og midlertidigt flytte det højere op.

**Opsamling: Sorteret i forhold til størrelse, pris og access time**

* Interne processpr registre -> Meget hurtig, meget dyr.
* Lille memory: Processor cache
* Main memory: RAM
* Device: Sekundær hukommelse.

Jo længere ned man kommer, jo større og langsommere bliver hukommelsen.

**Logical and physical cache**

En logical cache gemmer data i et virtuelt adresse lager og er lokaliseret mellem processoren og memory management unit(MMU). Processoren kan tilgå data fra logical cache direkte uden om MMU’en.

Hvorimod en physical cache gemmer hukommelse ved brug af fysiske adresser. En physical cache er lokaliseret mellem MMU og main memory. Dette medfører at hvis processoren vil tilgå hukommelsen, så skal MMU’en først oversætte den virtuelle adresse til en fysisk adresse før cache memory kan sende dataen til kernen.

**Hvordan er en cache opbygget?**

Cache arkitektur

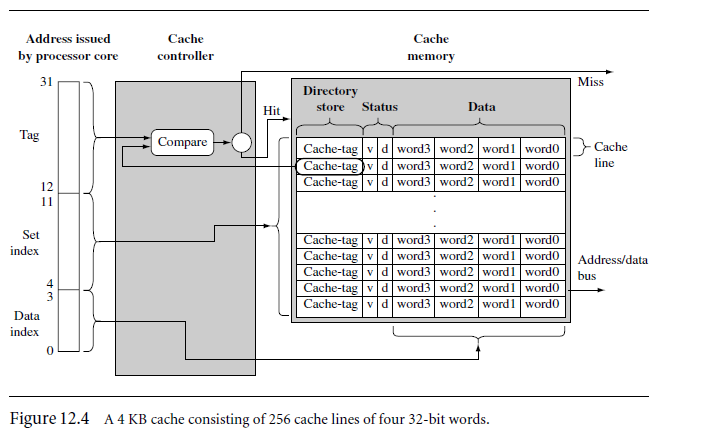
**ARM** bruger 2 forskellige bus arkitekturer i deres kerner, **Von Neumann og Harward**.

I processorer der benytter Von Neumann arkitekturen er der **en cache** der bruges til instruktioner og data. Denne type cache er kendt under navnet **unified cache.**

Derimod i processorer hvor **Harward arkitekturen** som benytter sig **at separate instruktions- og databus’** for at forbedre den overordnede perfomance. Dette kræver selvfølgelig også at der bruges **2 forskellige caches**; en intruktions cache(I-cache) og en data cache (D-cache). Denne type cache går også under navnet **split cache.**

Opbygning

En cache består **af 2 hoved elementer**; en **cache controller** og en **cache hukommelse**. Cache hukommelsen er et dedikeret hukommelses array, som indeholder cache lines.



**Cache controlleren** er en **hardware enhed** som kopiere kode **eller data fra main memory til cache automatisk**. Cachecontrolleren opfanger read og write requests, og måden et sådan **request bliver udført på er ved at inddele den i 3 dele:**

* Et tag-field
* Set index field
* Data field

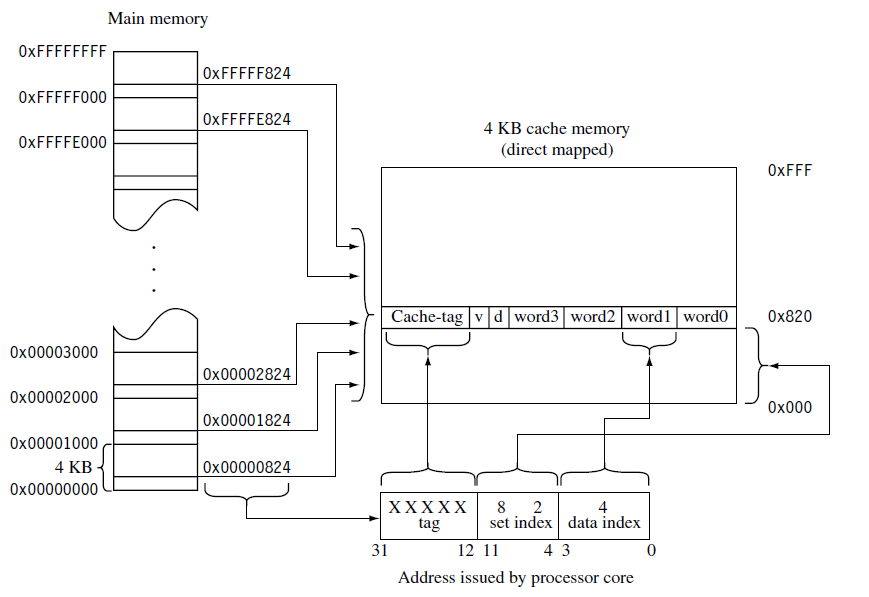
Først bruges set index delen af adressen til at lokalisere cache linien i cache hukommelse for at tjekke om den indeholder det forespurgte data. Controlleren tjekkere **valid bit**(valid bit er en markering af cache linien er aktiv, ellers kaldes den for **dirty**) for at se om cache line er aktiv og sammenligner cache-tag til det forespurgte flag. Hvis begge disse tjek er succesfulde, er der det der kaldes et **cache hit**. Hvis en af disse fejler, så er det et **cache miss**. Ved et cache-hit overføres indholdet af data-index til processeren og dette kan nu eksekveres. Ved cache-miss kopieres en cache linie fra main memory til cache memory og giver det til processeren, dette kaldes **cache line fill.**

**Write Policies**

* Write-back
  + Data bliver skrevet til cacher i stedet for memory
  + Data og cache er ikke enstemmig
  + Cacheline’s dirtybit er sat
  + Hvis værdien tilgåes igen, bliver den taget direkte fra cache.
  + Hvis cache linien bliver evicted skrives data til memory i stedet.
* Write-through
  + Data bliver skrevet til både cache og memory. Dvs cache opdatere hukommelsen.
  + Cache og memory er opdateret(synkroniseret)
  + Langsommere pga. dobbelt skrivning.

**Direct-mapped cache – simplest form for cache**

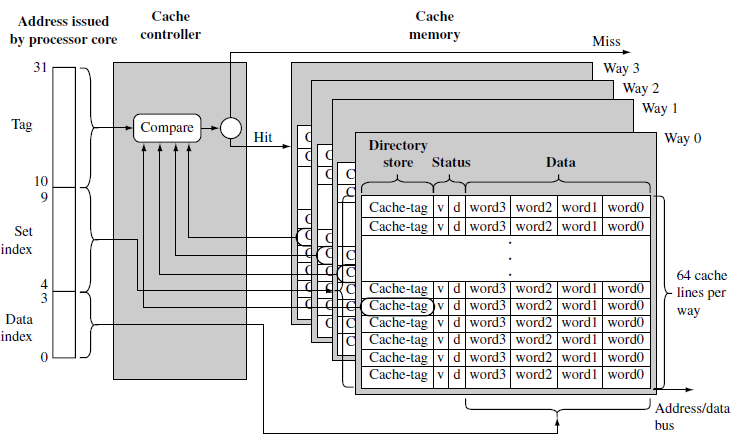
Hver adresse location I main memory mappes til en lokation I cachen. Main memory er langt større end cachen, og der vil derfor være mange adresser i main memory som mapper til samme adresse i cachen.



Figuren viser en direct mapped cache, hvor der principielt kunne være 1 million adresser i main memory som mappes til en adresse i cachen. Hvis der i cachen eksistere en **valid cache linie**, men som **repræsenteres af en anden adresse blok i main memory**, så bliver **cachen linie smidt ud** og erstattet af den cache linie med den matchende adresse. Dette **kaldes eviction**.

Problemer med direct-mapped cache: **Thrashing** – kamp om den samme cache hukommelse fra de mange fysiske hukommelsesadresser.

**Set Associative cache**



Minder meget om det tidligere, nu inddeles hukommelse blot i mindre lige store dele, som kaldes for **ways**. Eksemplet overfor er stadig en 4KB cache, så mængden af hukommelse er ikke ændret. Grunden til dette er at vi med dette design kan **reducere Thrashing**.

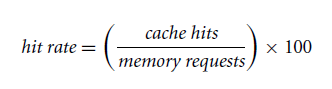
Nu bruges set-index til at adressere mere end en cache linie – den peget på en cache linie i hver way. Hvor vi før havde en stor way med 256 linier, har vi nu 4 med 64 linier i hver. Nu mappes **hver fysisk adresse til 4 forskellige steder i cache hukommelsen.**

**Fully association - CAM**

Med dette design, ledes tankerne overpå hvorfor man så ikke laver det sådan at enhver fysisk adresse kan mappes til en hver adresse i cachen, sådan at man på denne måde øger associationen og minimere Thrashingen. Denne type cache kaldes for fully association, men når associationen stiger så stiger kompleksiteten for hardwaren også.

**Måling af cache effektivitet**

Der bruges 2 termer til at måle en cache’s effektivitet**, cache hit rate, og cache miss rate.**



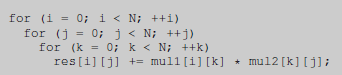
Cache miss udregnes på samme måde, blot med cache miss i tælleren i stedet.

**Hvordan kan man som programmør optimere brugen af cache?**

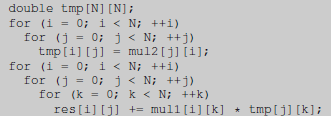
Som programmør vil det bedste sted at starte med at optimere ved være ved Level 1 cachen, da det er her vi ville kunne se den største optimering.

Vi lavede i øvelsen 2 programmer som bestod af nogle matrix beregninger et langsomt og et hurtigt program.

Langsomt:



Hurtigt:



Opgaven gik så ud på at compilere med forskellige compiler optimeringer O0, O1, O2, og se hvad det gjorde ved udførslen af vores program. Til at måle med blev brugt time, som måler tidsforbruget af eksekveringen.

Vi kom frem til følgende:

* O0 :  
  Reducerer compiler tiden, ellers bare default compilering
* O1 :  
  Prøver at reducerer code size og tiden der tager at kører programmet på. Det tager dog lang tid at compile. O1 tager loops med, som tager længere tid at kører end hvis man bare laver 400 linjer med ekserkvering, som O2 optimering gør
* O2 :  
  Her bliver alt optimeret, både compilertiden og tiden det tager at kører programmet på. Det eneste der ikke bliver optimeret er kodestørrelsen. Laver ikke loops da denne ikke går efter få kodestørrelsen ned - denne optimering folder loopet ud og skriver f.eks. 400 linjer med kode.

Som det ses så kører O1 optimeringen en smule hurtigere end O2!  
Dette kan skyldes at O2 optimeringen fylder cachen ud fordi den laver 400 linjer med kode.. Når cachen er fyldt ud skal den hente fra memory, og det kan være forklaringen på at O2 optimeringen tager en smule længere tid end O1.

