Föreläsningsanteckningar
5. Cacheminnen

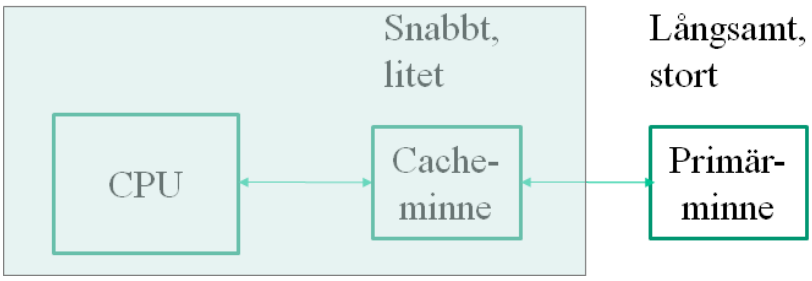
Olle Seger 2012
Anders Nilsson 2016

1 Inledning

Bakgrunden till att cacheminnen behövs för nästan alla datorer är enkel. Vi kan kallt räkna med att processorn är pipelinad. Typiskt gäller detta:

- Processorn behöver hämta en ny instruktion varje klockcykel. Ofta behövs också en operand samtidigt.
- Vår dator behöver ett minne på låt oss säga 1 GB.
- De här kraven går inte ihop. Processorn är minst 10 gånger snabbare (hungrigare) än ett minne av den storleken.

Ett cacheminne är ett litet och snabbt minne, som är placerat mellan CPU och primärminne: Grejen är att automatiskt försöka hålla de oftast använda instruktionerna/data i cacheminnet.



Låt oss göra N accesser och anta följande parametrar:

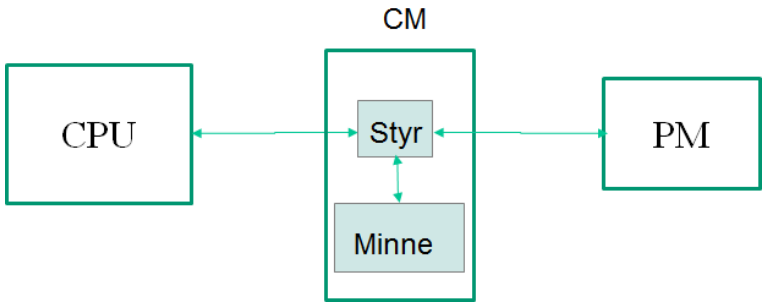
minne	accesstid	antal accesser
cache	t_{CM}	N_{CM}
primär	t_{PM}	$N_{PM} = N - N_{CM}$

Medelaccestiden ges av:

$$\begin{aligned} t_{med} &= \frac{N_{CM} \cdot t_{CM} + (N - N_{CM}) \cdot t_{PM}}{N} \\ &= h \cdot t_{CM} + (1 - h) \cdot t_{PM}, \end{aligned}$$

där $h = N_{CM}/N$ kallas träffkvot eller hit ratio. Av formeln framgår att om träffkvoten närmar sig 1, så blir medelaccestiden lika med cacheminnets accestid. En typisk träffvot är 0,95.

Ett cacheminne består av en styrenhet (cache controller) och ett snabbt minne:



När vi accessar minnet händer följande:

- När datorn startar är cacheminnet tomt .
- Varje minnesaccess ger till att börja med en cachemiss, kopiering PM->CM. Vanligt är att vid en cachemiss kopiera flera (t ex 4) instruktioner/data till CM. Kallas en cacheline.

Anledningen till att det går att få höga träffkvoter är nu följande:

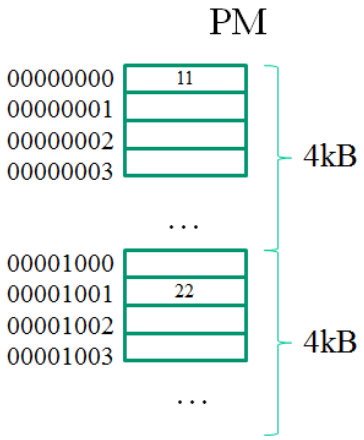
- Program innehåller loopar => sannolikheten är stor att samma instruktion/data accessas igen (temporal lokalitet).
- Program/data är sekvensiella => sannolikheten är stor att också nästa instruktion/data accessas (spatial lokalitet).

2 Olika typer av cacheminnen

Vi går nu direkt på några olika sätta att bygga cacheminnen. Som ett exempel tänker vi oss att primärminnet är 4 GB och att databussen är 32 bitar bred. Minnet är alltså organiserat 1G x 4B. För att adressera primärminnet behövs 32 adressledningar.

Vi vill bygga ett cacheminne med storleken 4 kB. Cachelinens storlek ska vara 4 ord = 16 byte.

När man resonerar runt cacheminnen, kan det hjälpa att tänka på primärminnet, som indelat i block lika med cacheminnets storlek. Vi tänker ett steg vidare och visar också cachelines:



Vi slår också fast att en byte kan finnas på vilken adress som helst i minnet. Ett ord (=4 bytes) kan bara finnas på en adress, som är en multipel av 4. På samma sätt gäller att en cacheline bor på en multiplel av 16 i adressrymden.

2.1 Associativt minne

Nedanstående bild visar en associativ cache. Vi har valt storleken 4kB, alltså består den av 256 identiska rader bestående av en komparator, ett tagregister och cachelinen (16 B). Efter att det första blocket i PM har besökts, finns detta i cachen (naturligvis är hela cachelinarna fyllda):

Observera att vi för varje cacheline, måste spara varifrån i minnet den kommer. Det räcker med att spara cachelinens nummer. I minnet får det plats $2^{32}/2^4 = 2^{28}$ cachelines, alltså de 28 mest signifikanta bitarna. I bilden visas de 7 mest signifikanta hex-siffrorna.

Vid sökning i cacheminnet jämförs adressens 28 mest signifikanta bitar parallellt med 256 komparatorer med de sparade taggarna. Vid likhet någonstans har vi en cacheträff.

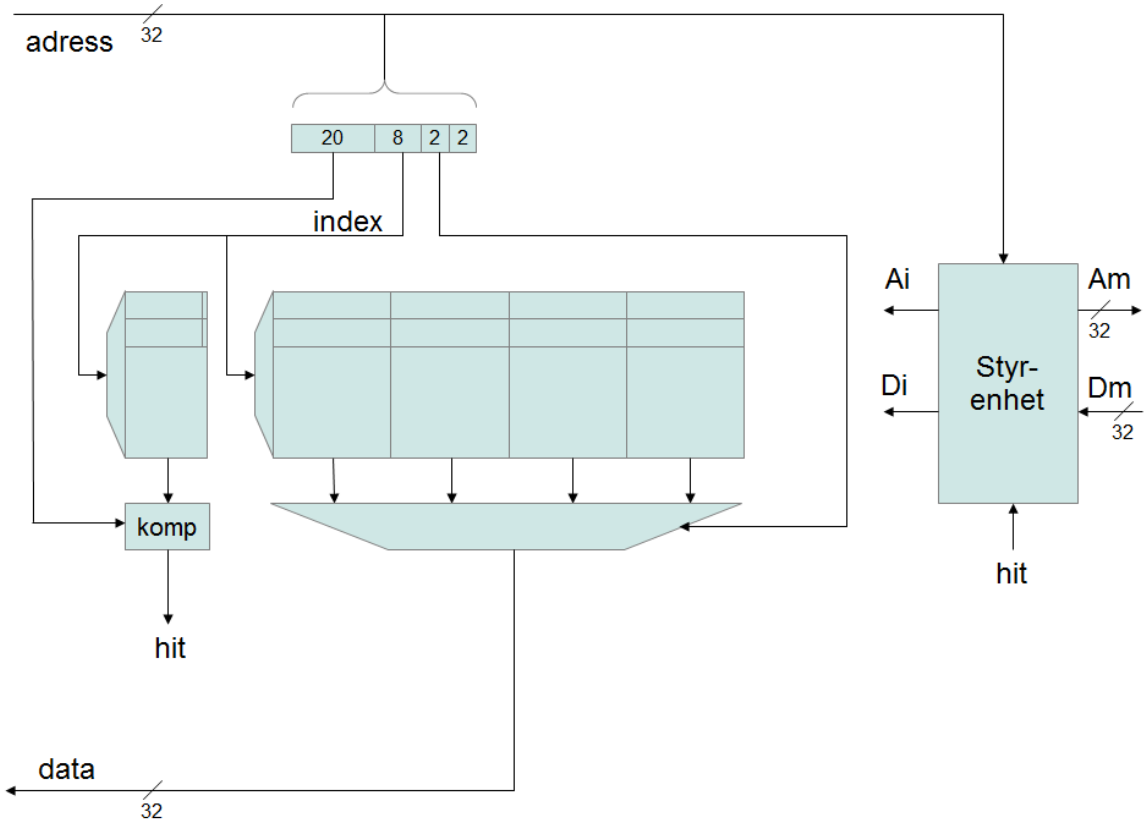
Observera att till varje cacheline hör en kontrollbit: valid. Annars kan vi inte skilja på skräp och riktiga data. Vid reset (eller cachetömning) nollställs alla validbitar.

2.2 Direktmappad cache

Associativt minne är ett dyrt men bra sätt att bygga cache på. 4kB-cache innehåller alltså 256 komparatorer. Fördelen är att cachelinar kan placeras var som helst i cachen. Hur det går till hoppar vi över.

Nedanstående bild visar en direktmappad cache på 4kB. Den innehåller bara en komparator. För att åstadkomma detta kan inte cachelinarna placeras fritt

längre! Vi tar samma exempel som förra gången. Början på det övre blocket har besökts.



Det bör påpekas att ovanstående är bara en metod att bygga en cache med endast en komparator. Hur har det gått till?

Vi bestämmer oss för detta: en cacheline från minnet ska placeras på samma relativa position i cacheminnet som i blocket! Denna position (rad) brukar kallas index. Vi gör en formel av detta. (A = adress, CL = cachelinens storlek, CM = cacheminnets storlek)

$$\begin{aligned} index &= (A \mod CM)/CL \\ &= (A \mod 2^{12})/2^4 \end{aligned}$$

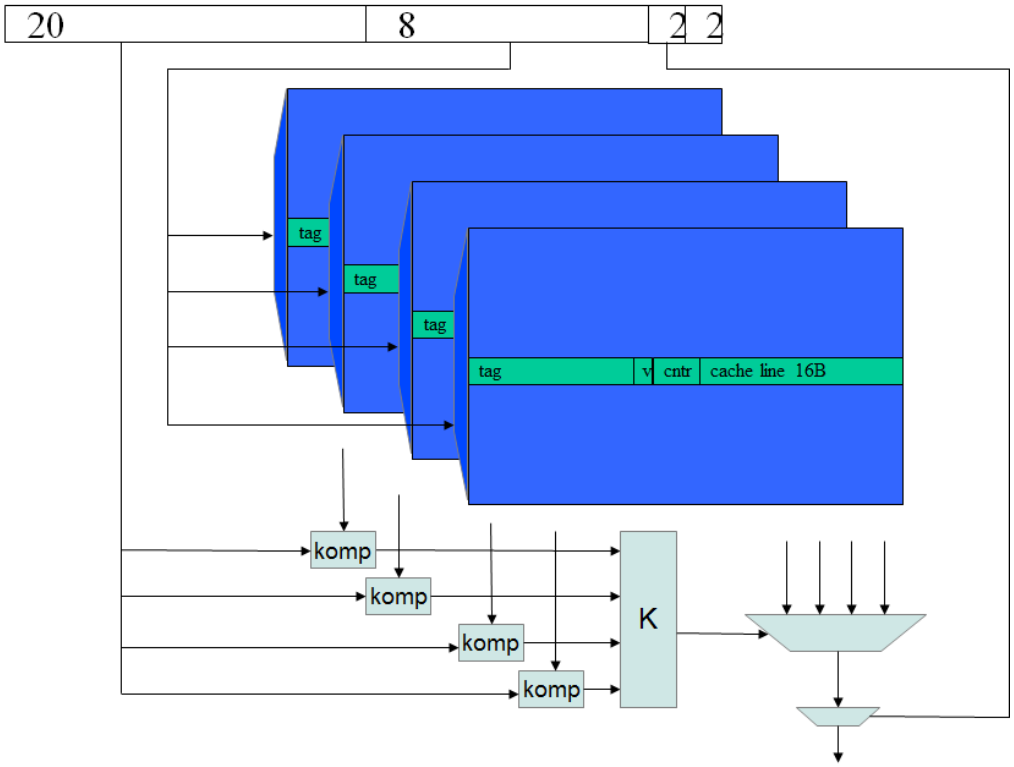
och taggen är helt enkelt blocknumret och ges av

$$\begin{aligned} tag &= A/CM \\ &= A/2^{12} \end{aligned}$$

Cachelinen som börjar på adressen $0x12345670$ ska alltså placeras på $index = 0x67$ med $tag = 0x12345$. Vi inser då att cachelinar på adresserna $0xzzzzzz670$ (z=godtycklig hex-siffra) får samma index. Detta är nackdelen med en-komparator-metoden.

2.3 Flervägscache

Den enklaste metoden att råda bot på detta är att helt enkelt duplicera cacheminnet:



Detta är en 4-vägs cache på 16 kB. Vid en cachemiss, så hämtas ett data och vi har nu ett val. Vilken av de 4 cachelinarna kommer att bli överskriven. Vi behöver en så kallad replacement algorithm. En vanlig sådan är LRU=least recently used. Den brukar beskrivas så här. Till varje cacheline hör en liten räknare (kanske 3 bitar). Denna används för att jämföra de 4 cachelinarna med samma index.

Detta är ett tänkbart scenario:

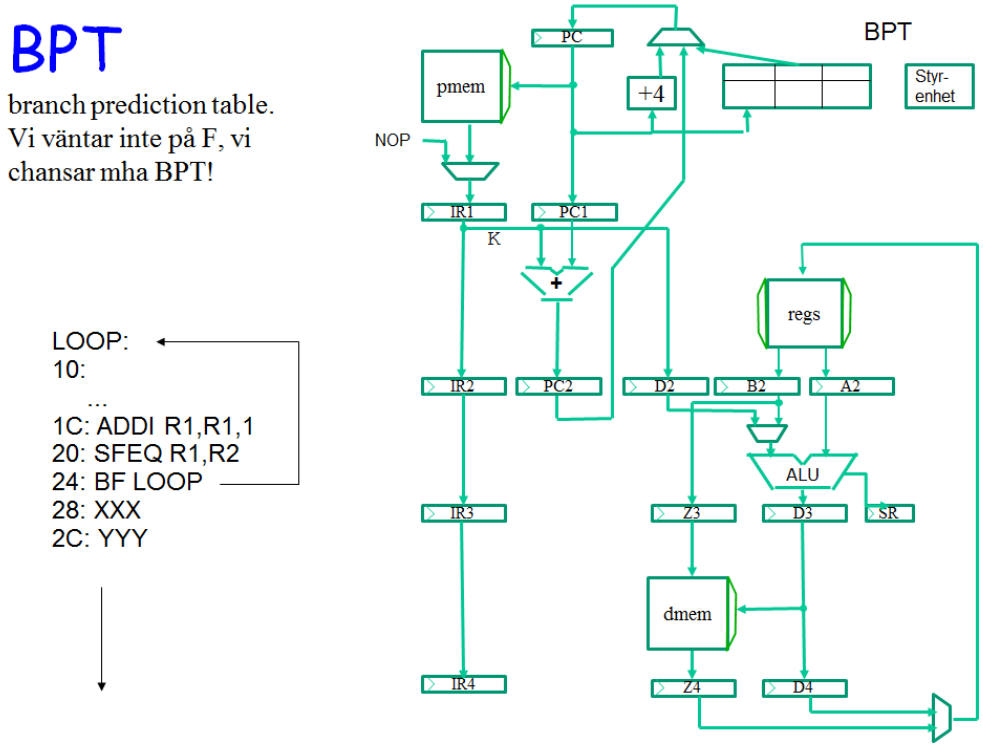
- cache hit: max->cntr

- cache miss: välj cacheline med minst cntr. Fyll på. max->cntr
- inget: räkna ner alla cntr (aging, dock inte på varje cp)

Ovanstående är den vanligaste cachetypen. Givet en viss cachestorlek t ex 16 kB, visar det sig vara bättre att organisera den som 4x4 kB istf 1x16 kB.

2.4 BPT = Branch Prediction Table

BPT avser att få bort den förtretliga NOP-en vid ett taget hopp. Betrakta vår dator med en installerad BPT:



En BPT, består av ett associativt minne och en styrenhet. I det associativa minnet lagras information om hoppinstruktioner i programmet:

- hoppets adress, var finns hoppet.
- hoppadressen, vart hoppar hoppet.

- hoppstatistik. Med exempelvis 2 bitar, kan vi hålla reda på om hoppet är: Strongly Taken (ST), Taken (T), Not Taken (NT) eller Strongly Not Taken (SNT). En styrenhet lyssnar på programflödet och uppdaterar bitarna.

Vi antar nu att vi har kört några varv i loopen, så att hoppet finns noterat i BPT. Ännu så länge har hoppet utförts varje gång, så hoppstatistiken säger ST. Vi ser nu att när PC har blivit 24 och hoppet ska hämtas ur programminnet så kan samtidigt PC få värdet 10. Detta är en chansning (prediktering). Någon klockcykel senare får vi reda på om chansningen var korrekt. Om så är fallet är det bara och köra på och vi har vunnit en klockcykel (den där NOP-en). Om inte måste en felaktigt inhämtad instruktion ersättas med NOP.

Vi tar en lite närmare titt på detta påstående med ett pipelinediagram. Vi inför ett nytt register SPC = sparad PC.

Vi kör detta program som en test. Vi antar att BPT falskeligen predikterar och skissar på vad som händer.

```
LOOP:
    10: ZZZ
    14: VVV
    18: WWW
    1C: ADDI R1,R1,1
    20: SFEQ R1,R2
    24: BF LOOP
    28: XXX
    2C: YYY
```

PC	IR1	IR2	IR3	IR4	Kommentar
24	SFEQ	ADDI	-	-	Använd BPT och 10->PC, PC+4 ->SPC
10	BF	SFEQ	ADDI	-	Hoppet redan verkställt.
14	ZZZ	BF	SFEQ	ADDI	Låt F=0. Fel chansat! SPC-> PC
28	NOP	NOP	BF	SFEQ	Vi får stopp på ZZZ,VVV. Sätt hoppstatistik = T.
2C	XXX	NOP	NOP	BF	Vi är på rätt spår

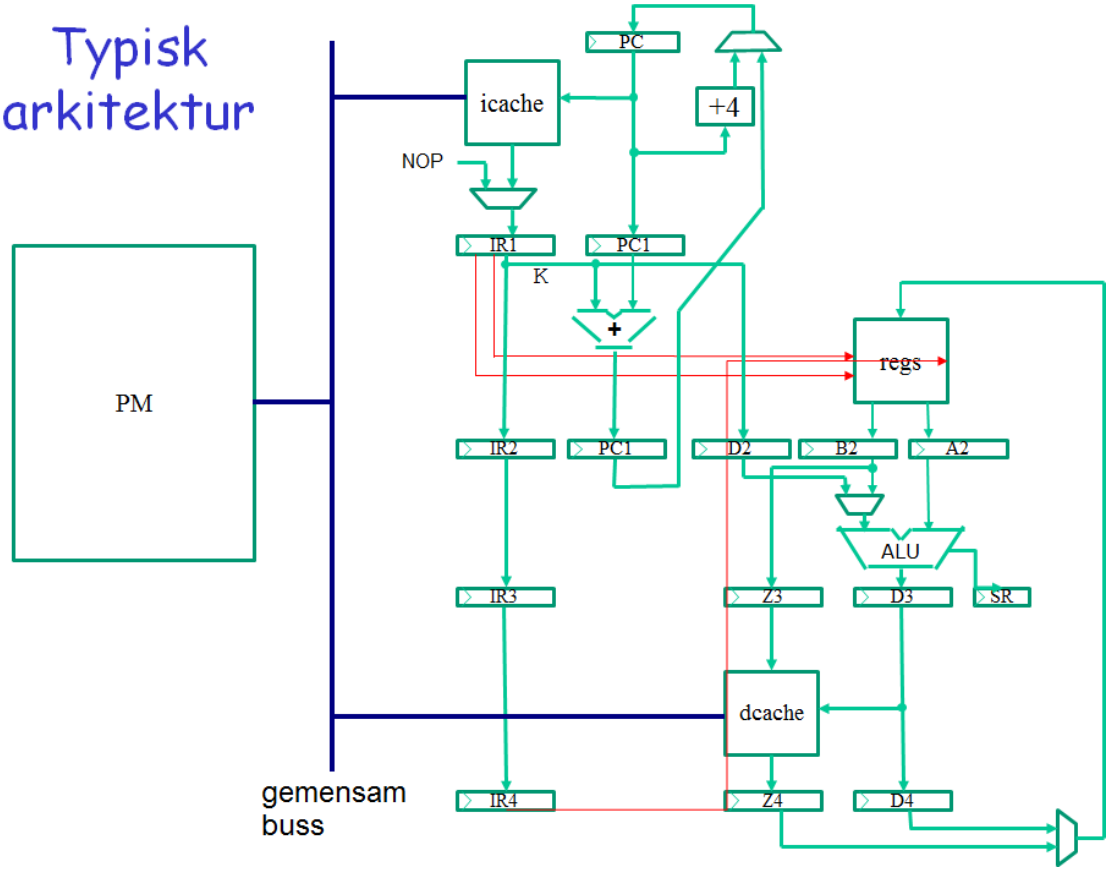
Vi konstaterar efter denna (långt ifrån fullständiga) undersökning:

- *Rätt prediktering*. Vinst av två klockcykler. Instruktionen `XXX` går inte in i pipelinen och någon NOP behövs inte alls.
- *Fel prediktering*. Förlust av två klockcykler.

Så länge rätt prediktioner överväger vinner vi på att koppla in en BPT.

2.5 Separata I- och D-cachar

Slutligen visar vi denna koppling, som är mycket vanlig. Instruktions- och data-minnena har bytts ut mot cacheminnen. Dessa båda cacheminnen kommunicerar över en gemensam buss mot primärminnet.



Vi konstaterar följande:

- Vid cachemiss i I-cache eller D-cache måste pipelinen stoppas.
- Båda cacharna använder en gemensam buss mot minnet. Samtidig miss i I-cache och D-cache, innebär påfyllning av en cache i taget. Pipelinen fryst hela tiden.

I D-cachen tillkommer en frågeställning: hur ska skrivning mot minnet hanteras. Två taktiker brukar nämnas i litteraturen:

- *write-thru*. Vid skrivning mot cacheminnet uppdateras även primärminnet.
- *write-back*. Uppdateringen av primärminnet sker endast när en cacheline kommer att skrivas över. En extra kontrollbit, dirty, håller reda på om cachelinen ändrats. Endast då måste en skrivning ske.

Referenser

[1] Josefsson M. (2008): *Föreläsningsunderlag TSEA49 Datorteknik D del 2*, LiU-tryck.