Pamięć podręczna - cache memory – zasada działania (odczyt)

Dostęp do pamięci:

- w pierwszej kolejności niezbędnych danych (instrukcji) poszukuje się w najbliższym bloku pamięci - o najkrótszym czasie dostępu (typowo – *cache* Level 1),

- jeżeli żądane elementy znajdują się w pamięci podręcznej (tzw. trafienie – ***hit***), przenoszone są bezpośrednio do miejsca docelowego (rejestry, jednostki wykonawcze) w CPU,

- jeżeli jednak (na przeszukiwanym poziomie w hierarchii pamięci) potrzebnych danych brakuje (chybienie – ***miss***), wykonywany jest dostęp do następnego poziomu pamięci (o większej pojemności, ale i dłuższym czasie dostępu).

- Z poziomu niższego (np. pamięci operacyjnej) do pamięci położonej „bliżej” CPU (np. podręcznej) pobierany jest **cały blok** danych – nie tylko zawierający żądany element (instrukcję, dane) ale również **jego najbliższe sąsiedztwo**: tzw. **linia** (*cache line* - pamięć *cache* podzielona jest na bloki-linie, najczęściej o długości 64 bajtów).

Oczywiście żądane dane (instrukcje) również trafiają do CPU…

Pamięć podręczna - cache memory – zasada działania c.d.

Użyteczność i wydajność pamięci podręcznej zależy od

lokalności przestrzennych i czasowych w konkretnym programie czyli:

- jeżeli CPU przetwarza kolejne (umieszczone „po sobie” w kolejnych komórkach pamięci RAM) instrukcje i kolejne dane (nie ma skoków) jeden dostęp do „wolnej” pamięci głównej (operacyjnej) powoduje załadowanie do *cache* od razu kilku elementów sąsiednich – które będą już pobierane z szybkiej pamięci podręcznej.

- jeżeli CPU wykonuje rozkazy w pętli – jest duże prawdopodobieństwo, że raz pobrane rozkazy z pamięci głównej będą w już *cache* podczas kolejnej iteracji.

Miarą wydajności pamięci podręcznej
(lub też miarą lokalności w kodzie) może być:

współczynnik trafień (hit_rate) = number_of_hits / all_memory_requests

lub **współczynnik chybień: (miss_rate)** = 1 – hit_rate

Pamięć podręczna

• ma dużo mniejszą pojemność od głównej pamięci operacyjnej*,

• może przechowywać dane pochodzące z wielu różnych w lokalizacji pamięci głównej.

Powstają problemy:

• Skąd procesor ma wiedzieć, czy **żądane** dane/instrukcje **są już** w pamięci *cache*?

• Jak je znaleźć? Adresy komórek pamięci głównej nie muszą odpowiadać* adresom komórek *cache*…

Potrzebny jest jakiś schemat/algorytm decydujący o (szybkim!) wyborze linii pamięci *cache*, w których ulokowane mają zostać dane pobrane z pamięci operacyjnej.

Najprostsze rozwiązanie:

numer linii cache jest wyznaczany na podstawie adresów danych w pamięci głównej.

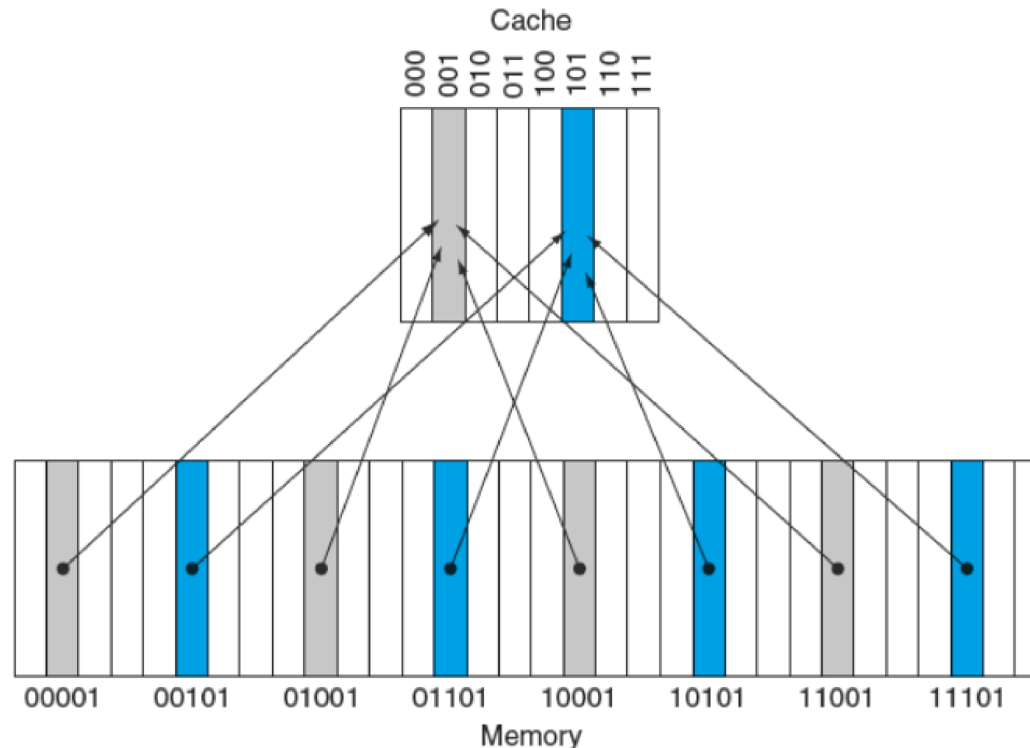
Direct mapped cache
mapowanie/odwzorowanie bezpośrednie - najprostsze

każdemu blokowi w pamięci głównej przydzielane jest **tylko jedno,** stałe miejsce (jedna linia) w pamięci *cache*:

numer_linii_cache = adres_bloku_w_RAM modulo liczba_linii_w_cache

przykład: mapowanie 32 bloków pamięci operacyjnej w 8 liniową pamięć *cache* (czyli pamięć podręczna jest cztery razy mniejsza od głównej - RAM)

Zakładamy, że rozmiar bloku odpowiada długości jednej linii!



Pamięć podręczna

- Wynik operacji ($x \bmod 2^n$) zawiera się w przedziale od 0 do 2^n-1 co odpowiada układowi *n* najmłodszych bitów liczby *x* - więc nie trzeba nic liczyć.

- W jednej linii *cache* mogą się znaleźć dane z jednej z wielu potencjalnych lokalizacji w pamięci głównej (tu: z czterech, np. wszystkie bloki z końcówką adresu 001 mogą znaleźć się tylko w linii 1, a z końcówką 101- tylko w linii 5).

- Takie rozwiązanie samo w sobie jest niejednoznaczne – potrzebna jest dodatkowa informacja, jasno określająca z którego miejsca w pamięci operacyjnej (jednego z wielu możliwych) pochodzą dane aktualnie znajdujące się w linii *cache.*

- Dodatkowo, potrzebna jest informacja czy dana linia *cache* zawiera poprawne dane/instrukcje aktualnie wykonywanego programu, a nie np. losowe śmieci (po włączeniu zasilania) lub pozostałości po uprzednio zakończonym programie.

Pamięć podręczna

Aby rozwiązać powyższe problemy pamięć *cache* jest zorganizowana jako struktura danych z dodatkowymi „polami”:

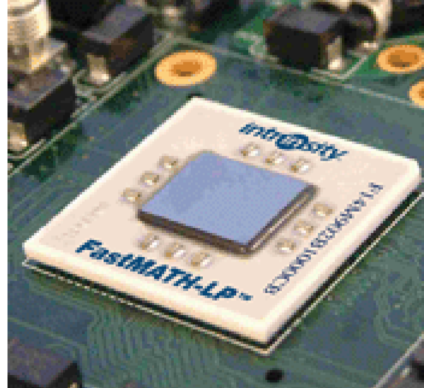
tag – zawiera starszą część oryginalnego adresu (z pamięci operacyjnej). Pozwala to zweryfikować czy linia zawiera faktycznie dane z pożądanej lokalizacji w pamięci głównej (a nie innej, ale mapowanej w tej samej linii),

valid_bit – określa, czy linia zawiera poprawne, aktualne dane, czy losowe „śmieci”,

reference_bit – określający czy do danej linii w ostatnim czasie był dostęp R/W,

dirty_bit – określa, czy zawartość linii była modyfikowana, czy nie.

Cache memory – przykład - pamięć *cache* o pojemności 16kB: zorganizowana jest jako 256 linii o długości 64 bajtów (512 bitów)



MIPS CPU

Tag | Index | Block offset

Address (showing bit positions)

31 ··· 14 13 ··· 6 5 ··· 2 1 0

18 | 8 | 4 | Byte offset

Hit

Tag

Index

Block offset

Data

18 bits

512 bits

V | Tag | Data

256 entries

18

32

32

32

komparator

Mux

AND

32

Cache memory – przykład – cd

procesor MIPS operuje 32 bitowymi adresami i 32 bitowymi słowami
(wszystkie dane i instrukcje mają po 4 bajty, adresy są podzielne przez 4)

chcąc odczytać dane/instrukcje spod adresu „*x*” w pamięci głównej, adres ten dzielony jest na trzy części:

- najmłodsze 6 bitów – niewykorzystane
(ew. bity 2-5 pozwalają potem wybrać 1 z 16 cztero bajtowych elementów w linijce),

- kolejne 8 bitów – wybiera/adresuje 1 z 256 linii w pamięci *cache,*

- najstarsze 18 bitów adresu „*x*” komparator porównuje ze starszą częścią oryginalnego adresu przechowywanego w polu Tag wybranej linii,

- jeśli starsze części obu adresów są identyczne (czyli dane w cache pochodzą z właściwego miejsca) oraz pamięć nie zawiera przypadkowych śmieci (valid_bit: V=1) mamy trafienie (Hit) i dane mogą zostać pobrane przez procesor z *cache.*

Przykład użycia 4-liniowego ***direct mapped cache*** (na podstawie Computer Organisation and Design)

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

Przykład użycia 4-liniowego ***direct mapped cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

stan początkowy – przed uruchomieniem programu pamięć podręczna nie zawiera żadnych potrzebnych danych

0	1	2	3
invalid	invalid	invalid	invalid

Przykład użycia 4-liniowego ***direct mapped cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

odczyt spod adresu:	cache:				
	0	1	2	3	
0 miss	**mem(0)**	invalid	invalid	invalid	dane z adr 0 ładowane są do linii 0

Przykład użycia 4-liniowego ***direct mapped cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

odczyt spod adresu:	cache: 0	1	2	3	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	**mem(8)**	invalid	invalid	invalid	odwzorowanie bezpośrednie wymusza ładowanie danych z adr 8 również do linii 0. Poprzednie zostały nadpisane!

Przykład użycia 4-liniowego ***direct mapped cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

odczyt spod adresu:	cache: 0	1	2	3	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	**mem(8)**	invalid	invalid	invalid	
0 miss	**mem(0)**	invalid	invalid	invalid	ponowne pobranie danych z adr 0 chociaż przed chwilą były!

Przykład użycia 4-liniowego ***direct mapped cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

odczyt spod adresu:	cache:				
	0	1	2	3	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	**mem(8)**	invalid	invalid	invalid	
0 miss	**mem(0)**	invalid	invalid	invalid	
6 miss	mem(0)	invalid	**mem(6)**	invalid	dane z adr 6 zapisywane są w linii 2, linia 0 nadal przechowuje swoją zawartość

Przykład użycia 4-liniowego ***direct mapped cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8

adres w pamięci głównej	numer linii cache
0	0 mod 4 = 0
6	6 mod 4 = 2
8	8 mod 4 = 0

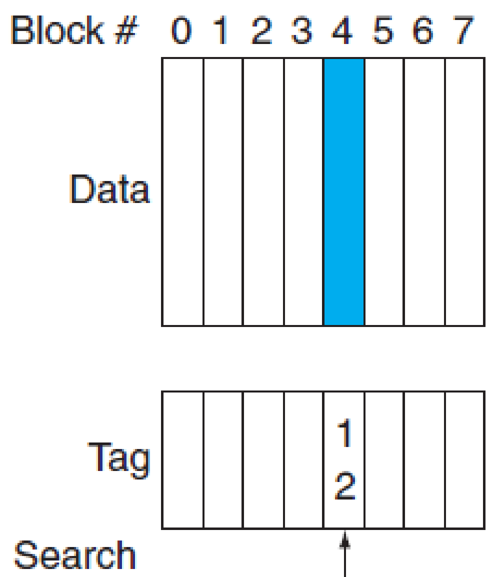
odczyt spod adresu:	cache: 0	1	2	3	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	**mem(8)**	invalid	invalid	invalid	
0 miss	**mem(0)**	invalid	invalid	invalid	
6 miss	mem(0)	invalid	**mem(6)**	invalid	
8 miss	**mem(8)**	invalid	mem(6)	invalid	znowu potrzebne to, co już było przd chwilą ładowane

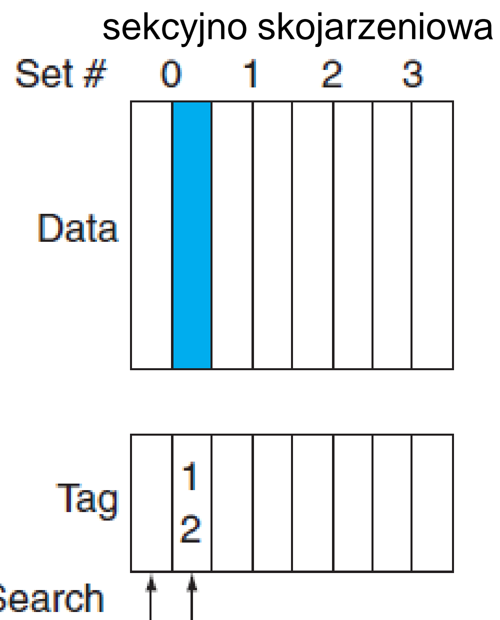
słaby algorytm odwzorowania skutecznie pogarsza efektywność pamięci cache typu direct mapped (tutaj cache w ogóle się nie przydał, a skomplikował tylko układ)

Cache memory – lepsze sposoby mapowania

różne organizacje tej samej, ośmioliniowej pamięci *cache*

two-way/four sets
dwudrożna/czterosekcyjna





12 mod **8** = 4

12 mod **4** = 0
(adr mod liczba sekcji)

Położenie bloku danych o adresie 12

blok z pamięci głównej może być zapisany w jednej z „kilku” (>1) możliwych lokalizacji (linii) (tutaj jednej z dwóch)

dowolny blok pamięci głównej może być umieszczony w dowolnej linii cache

Cache memory

Ponieważ, w przeciwieństwie do **direct mapped cache**, w organizacjach **set associative** i **fully associative** możliwe jest więcej niż jedno potencjalne umiejscowienie danych w *cache* - potrzebny jest algorytm wyboru linii, do której trafią kolejne odczytane elementy:

• random – losowy…,

• kolejka FIFO (First In - First Out),

• LFU (Least Frequently Used),

• **LRU** (Least Recently Used) – w pierwszej kolejności nadpisana nowymi danymi będzie linia od najdłuższego czasu nieużywana,

(skoro przez pewien czas nie było do niej dostępu – CPU pewnie zajmuje się już inną częścią programu i innymi danymi – „lokalności” zostały utracone).

Jedynym z najczęściej stosowanych mechanizmów wyboru linii jest jeden z wariantów LRU, np.

approximated LRU:

- co pewien czas sterownik pamięci kasuje reference_bits wszystkich linii,

- jeśli do danej linii wykonywany jest dostęp - reference_bit ustawiany jest na „1”,

- gdy podczas ładowania kolejnych danych trzeba ponownie wybrać którąś linię do zapisu, w pierwszej kolejności wybierane będą linie z reference_bit=0

  (czyli takie, które nie były używane od ostatniego kasowania bitów).

Przykład użycia 4-liniowego ***two-way set associative cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8
(jak w poprzednim przykładzie)

adres w pamięci głównej	numer sekcji wymiana linii **LRU**
0	0 mod 2 = 0
6	6 mod 2 = 0
8	8 mod 2 = 0

odczyt spod adresu:	cache: dwie sekcje po dwie linie			
	0	0	1	1
	invalid	invalid	invalid	invalid

Przykład użycia 4-liniowego ***two-way set associative cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8
(jak w poprzednim przykładzie)

adres w pamięci głównej	numer sekcji wymiana linii **LRU**
0	0 mod 2 = 0
6	6 mod 2 = 0
8	8 mod 2 = 0

odczyt spod adresu:	cache: dwie sekcje po dwie linie				
	0	0	1	1	
0 miss	**mem(0)**	invalid	invalid	invalid	mamy do wyboru dwie puste linie pierwszy wpis np. do tej o niższym numerze

Przykład użycia 4-liniowego ***two-way set associative cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8
(jak w poprzednim przykładzie)

adres w pamięci głównej	numer sekcji wymiana linii **LRU**
0	0 mod 2 = 0
6	6 mod 2 = 0
8	8 mod 2 = 0

odczyt spod adresu:	cache: dwie sekcje po dwie linie 0	0	1	1	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	mem(0)	**mem(8)**	invalid	invalid	adres 8 również mapowany do sekcji 0, ale mamy wybór: najdłużej nieużywana jest druga linia w tej sekcji. Nic się nie nadpisuje

Przykład użycia 4-liniowego ***two-way set associative cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8
(jak w poprzednim przykładzie)

adres w pamięci głównej	numer sekcji wymiana linii **LRU**
0	0 mod 2 = 0
6	6 mod 2 = 0
8	8 mod 2 = 0

odczyt spod adresu:	cache: dwie sekcje po dwie linie				
	0	0	1	1	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	mem(0)	**mem(8)**	invalid	invalid	
0 hit	mem(0)	mem(8)	invalid	invalid	dzięki temu, przy ponownym dostępie do tego samego elementu dane są obecne i mogą zostać pobrane z szybkiej pamięci cache

Przykład użycia 4-liniowego ***two-way set associative cache***

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8
(jak w poprzednim przykładzie)

adres w pamięci głównej	numer sekcji wymiana linii **LRU**
0	0 mod 2 = 0
6	6 mod 2 = 0
8	8 mod 2 = 0

odczyt spod adresu:	cache: dwie sekcje po dwie linie			
	0	0	1	1
0 miss	**mem(0)**	invalid	invalid	invalid
8 miss	mem(0)	**mem(8)**	invalid	invalid
0 hit	mem(0)	mem(8)	invalid	invalid
6 miss	mem(0)	**mem(6)**	invalid	invalid

Podczas wprowadzania kolejnych danych wybieramy linię drugą - wg LRU (ostatni dostęp był do pierwszej)

Przykład użycia 4-liniowego *two-way set associative cache*

CPU procesor wykonuje serię odczytów kolejno spod adresów: 0, 8, 0, 6, 8
(jak w poprzednim przykładzie)

adres w pamięci głównej	numer sekcji wymiana linii **LRU**
0	0 mod 2 = 0
6	6 mod 2 = 0
8	8 mod 2 = 0

odczyt spod adresu:	cache: dwie sekcje po dwie linie				
	0	0	1	1	
0 miss	**mem(0)**	invalid	invalid	invalid	
8 miss	mem(0)	**mem(8)**	invalid	invalid	
0 hit	mem(0)	mem(8)	invalid	invalid	
6 miss	mem(0)	**mem(6)**	invalid	invalid	
8 miss	**mem(8)**	mem(6)	invalid	invalid	niestety, pamięć jest mała, a sekcji i linii niewiele. Ponowny zapis 8 (poprzednia została nadpisana)

Rezultat lepszy niż w direct mapped – 1 trafienie

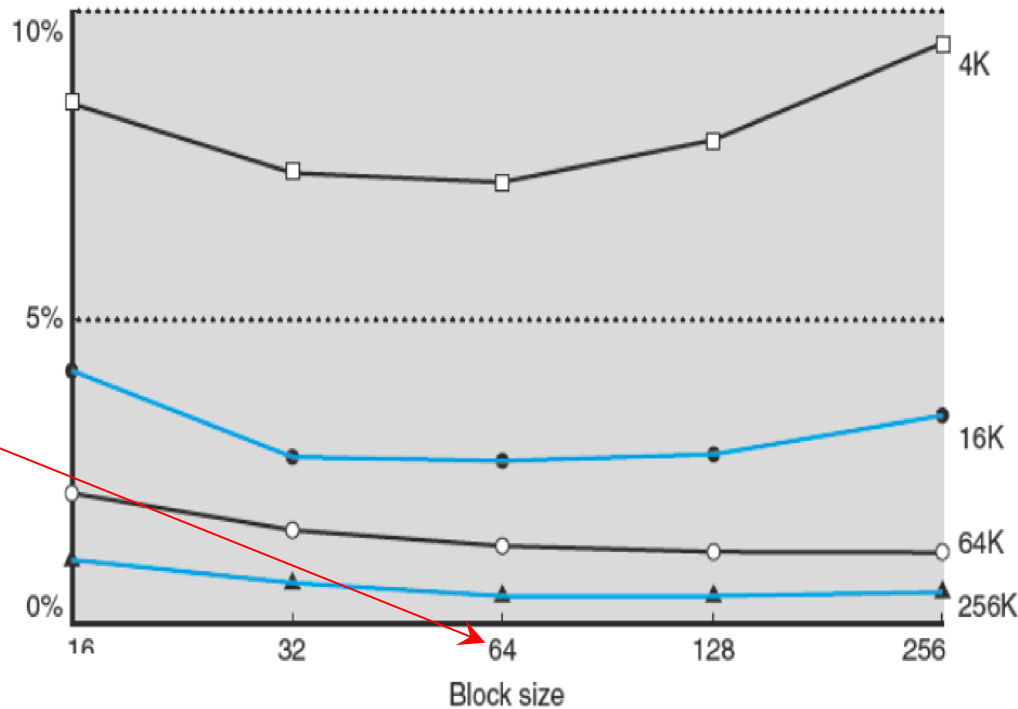
Przykład użycia 4-liniowy ***fully associative cache***

CPU procesor wykonuje tą samą serię odczytów: 0, 8, 0, 6, 8

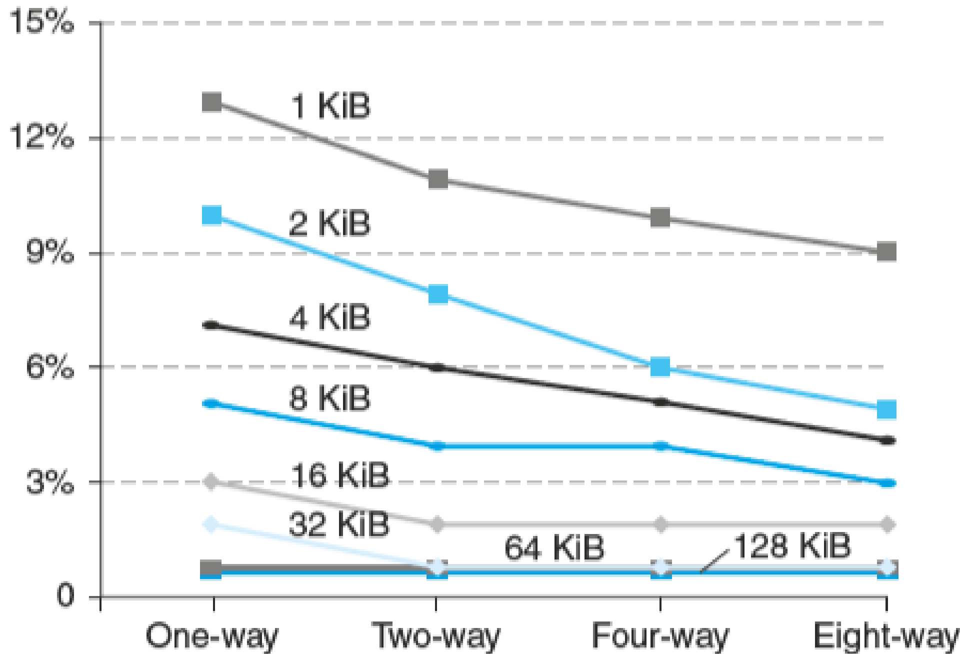
adres w pamięci głównej	jedna sekcja z czterema liniami: cztery możliwości wyboru niezależnie od adresu wymiana linii **LRU**			
	0	1	2	3
0 miss	**mem(0)**	invalid	invalid	invalid
8 miss	mem(0)	**mem(8)**	invalid	invalid
0 hit	mem(0)	mem(8)	invalid	invalid
6 miss	mem(0)	mem(8)	**mem(6)**	invalid
8 hit	mem(0)	mem(8)	mem(6)	invalid

Współczynnik chybień vs długość linii (dla typowych pojemnośći)

typowa długość linii *cache* - 64 bajty



współczynnik chybień vs skojarzenowość



One way=direct mapped

„wydłużenie" linii *cache*:

+ dla dużych pamięci *cache* powoduje spadek liczby chybień,

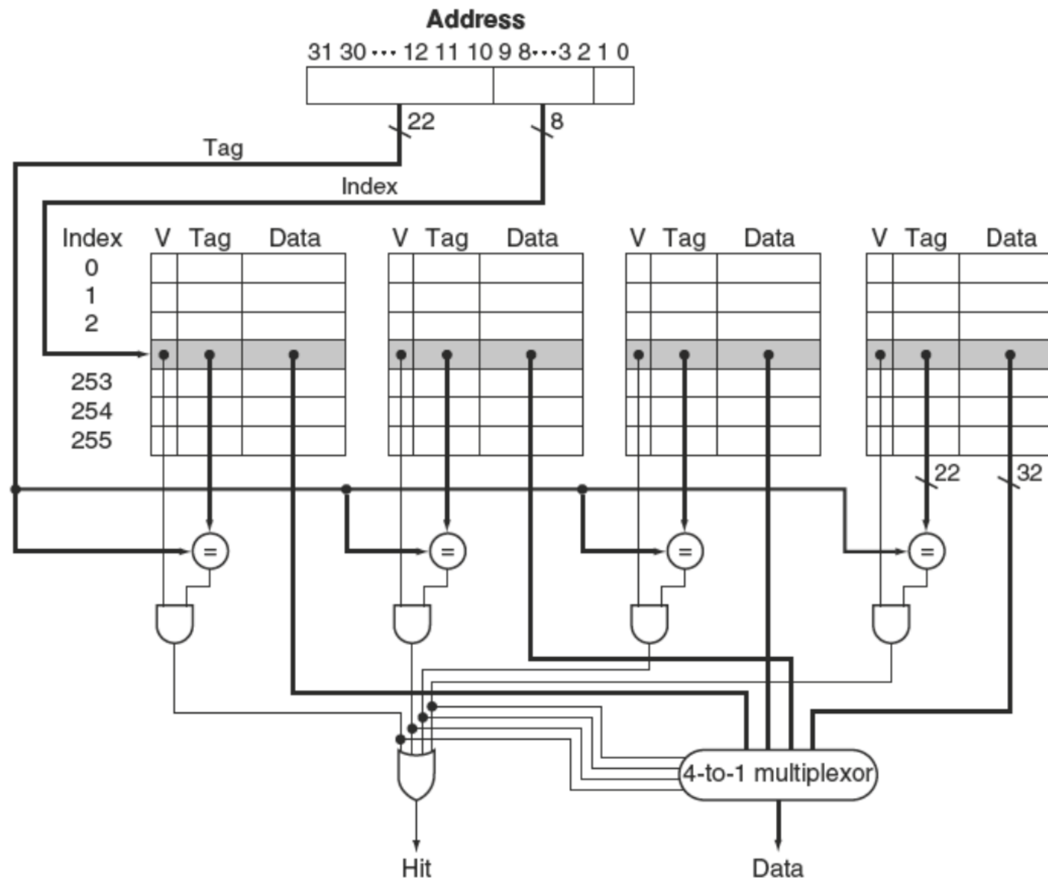
- w niewielkim *cache*, „długich linii" będzie mniej: ich zawartość będzie częściej wymieniana (mniej możliwości wyboru), a przechowywane informacje mogą nie zostać użyte w całości,

- ładowanie większych bloków z wolnej pamięci głównej może trwać dłużej…

Pełnej skojarzeniowości (***fully associative cache***) praktycznie nie stosuje się:

- wraz ze wzrostem skojarzeniowości i pojemności *cache* współczynnik chybień nie spada już tak gwałtownie jak np. przy zmianie z *direct mapped* na *two-way* (wykres na poprzednim slajdzie),

- wyższa skojarzeniowość to bardziej złożony (i wolniejszy) układ sterowania: (przykład: *four-way set associative*):



Zapis danych w pamięci podręcznej

Hierarchiczna budowa pamięci komputera powoduje, że te same dane (np. **zmienne** w programie) mogą być przechowywane w wielu kopiach, w pamięciach na różnym poziomie hierarchii – co skutecznie komplikuje mechanizm obsługi pamięci np. w komputerach wieloprocesorowych z pamięcią wspólną.
Pojawia się problem z zapisem danych po modyfikacji i potencjalną **utratą spójności** między kolejnymi kopiami tej samej zmiennej przechowywanych w różnych pamięciach.

Algorytmy zapisu:

• **write-through** – po modyfikacji, dane zapisywane są zarówno w pamięci podręcznej, jak i w pamięci o niższym poziomie hierarchii – np. operacyjnej (głównej).
Dzięki temu kopie zmodyfikowanych danych na różnych poziomach hierarchii są zawsze spójne. Rozwiązanie jest proste, bezpieczne, ale wolne – przy każdym zapisie wymagany jest dostęp do wolnej pamięci głównej. Obecnie nie jest używane.

• **write-back** – po modyfikacji, **nowe wartości zapisywane są tylko w cache**.
Zawartość całej zmodyfikowanej* (dirty bit=1) linii *cache* przenoszona (wyładowywana) jest do pamięci na kolejnym poziomie dopiero wtedy, gdy taka linia musi zostać nadpisana nową zawartością. Rozwiązanie jest szybsze, obecnie powszechnie stosowane, wymagające jednak bardziej skomplikowanego sterowania. Występuje również chwilowa utrata spójności danych.

*linie niemodyfikowane (dirty bit=0) oczywiście nie muszą być zapisywane z powrotem do pamięci operacyjnej.

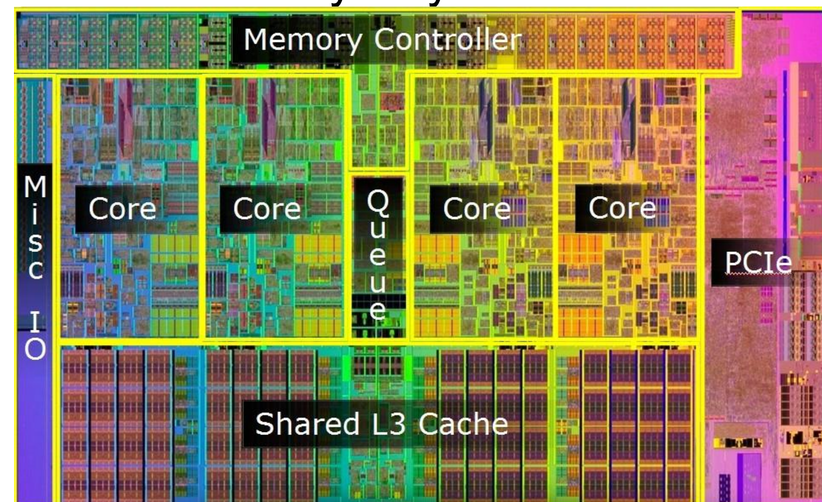
Wielopoziomowa pamięć cache

L1 – niewielka (np. 32kB), **zoptymalizowana pod kątem najkrótszego czasu dostępu z reguły rozdzielona jest na pamięć instrukcji i danych** (arch. Harwardzka) linie indeksowane adresami wirtualnymi,

L2 – większy rozmiar (np. 256kB), uzupełnia chybienia w L1, wyższa skojarzeniowość, trochę wolniejsza niż L1, linie indeksowane adresami fizycznymi.

Każdy rdzeń w procesorze (typu x86-64) posiada prywatne bloki L1 i L2*.

L3/LLC (Last Level Cache) - **współdzielona między rdzenie procesora** (pojemność liczona w MB), jeszcze większa skojarzeniowość (i wolniejsza) niż L2. W nowych CPU zgodnych z x86-64 podzielona jest na części (*slices*). Ich liczba odpowiada liczbie rdzeni, każda ma własny kontroler, połączenia z resztą układu są wykonane w topologii pierścienia. Linie indeksowane adresami fizycznymi. Dane z pamięci głównej rozlokowywane są po różnych częściach L3 na podstawie funkcji mieszającej** (*hash function*).



* W przypadku np. mikroarchitektury Intel Alder Lake, rdzenie „efficient" mogą współdzielić cache L2.

** C. Maurice, N. Le Scouarnec , Ch. Neumann, O. Heen, A. Francillon, Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters, RAID 2015:Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses, Vol. 9404, November, 2015, pp. 48–65.

Przykład: utrata spójności – koherentności pamięci

Komputer dwuprocesorowy (dwurdzeniowy) z pamięcią wspólną
- każdy procesor ma swoją, **oddzielną pamięć podręczną**,
- **pamięć operacyjna jest współdzielona między procesorami**,

- w pamięci głównej znajdują się dwie zmienne *x* i *y* – umieszczone blisko siebie, w jednym bloku (linii) wyrównanym do granicy długości linii *cache* (czyli adres bloku = wielokrotność długości linii).

1. Procesor CPU1 ładuje do rejestru wartość zmiennej *x*.
   Do pamięci *cache* CPU1 trafia nie tylko zmienna *x*, ale całe jej otoczenie (blok danych o długości jednej linii) – łącznie ze zmienną *y.*

2. Procesor CPU2 ładuje do rejestru wartość zmiennej *y.*
   Do pamięci *cache* CPU2 trafia nie tylko zmienna *y*, ale całe jej otoczenie (blok danych o długości jednej linii) – łącznie ze zmienną *x.*

Stan pamięci po tych operacjach:

| x y | - fragment pamięci operacyjnej

| x y | - linia pamięci *cache* CPU1

| x y | - linia pamięci *cache* CPU2

3. CPU1 modyfikuje zmienną *x* (np. w rejestrze) i zapisuje z powrotem w *cache*.
   (**algorytm write back!)**

4. CPU2 podobnie modyfikuje zmienną *y.*

- procesory pracują niezależnie, pobierając dane tylko z pamięci podręcznej,
- żaden z procesorów „nie wie co robi jego sąsiad",
- wartości zmiennych w pamięci operacyjnej pozostają **na razie** niezmienione.

Stan pamięci po operacjach 1-4:

	x	y		- fragment pamięci operacyjnej
|	xmod	y	| - linia pamięci *cache* CPU1
|	x	ymod	| - linia pamięci *cache* CPU2

W takiej sytuacji wartości zmiennych, jakie finalnie zostaną zapisane do pamięci głównej będą zależeć od tego, który procesor pierwszy zapisze (wyładuje) zawartość linii *cache* do pamięci operacyjnej!

Wyładowanie linii *cache* może nastąpić:

- jeśli dane w *cache* się „nie mieszczą" i zawartość linii musi być zastąpiona nową
- na żądanie – dedykowaną instrukcją, przy zmianie procesu itp.

Zapewnienie spójności pamięci *cache* **– cache coherency snooping protocol**

Stan pamięci po operacjach 1-2 (jak poprzednio):

| x y | - fragment pamięci operacyjnej
| x y | - linia pamięci *cache* CPU1
| x y | - linia pamięci *cache* CPU2

3. CPU1 modyfikuje zmienną *x* i zapisuje z powrotem w *cache* (alg. write back!).

4. Kontroler pamięci **śledzi (snoop)** stan linii *cache* i unieważnia wszystkie
inne kopie modyfikowanej linii w pamięciach podręcznych innych procesorów (tu CPU2).

Stan pamięci po operacjach 3 i 4:

| x y | - fragment pamięci operacyjnej
| xmod y | - linia pamięci *cache* CPU1
| invalid invalid | - unieważniona linia pamięci *cache* CPU2

5. CPU1 wyładowuje *cache* do pamięci głównej, a CPU2 pobiera dany blok ponownie:

| xmod y | - fragment pamięci operacyjnej
| xmod y | - linia pamięci *cache* CPU1
| xmod y | - linia pamięci *cache* CPU2

Zapewniona jest spójność – koherentność pamięci.

6. To samo się dzieje gdy CPU2 modyfikuje zmienną *y* i zapisuje z powrotem w *cache*.

7. Kontroler pamięci śledzi stan linii *cache* i unieważnia wszystkie inne kopie modyfikowanej linii w pamięciach podręcznych innych procesorów (tu CPU1).

Stan pamięci po operacjach 6 i 7:

xmod	y	- fragment pamięci operacyjnej
invalid	invalid	- linia pamięci *cache* CPU1
xmod	ymod	- linia pamięci *cache* CPU2

CPU2 wyładowuje *cache* do pamięci głównej, a CPU1 pobiera dany blok ponownie:

xmod	ymod	- fragment pamięci operacyjnej
xmod	ymod	- linia pamięci *cache* CPU1
xmod	ymod	- linia pamięci *cache* CPU2

Algorytm ten (jedna z wersji **snooping protocols**) – zgodnie z zasadą działania nazywa się „**write invalidate protocol”** (unieważnienie – skasowanie bitów „valid”).

- Zapewnienie spójności pamięci wymaga dodatkowych przeładowań linii *cache* - co obniża wydajność całego systemu.

- Osobną kwestią występującą tutaj jest tzw. ***false sharing***:

dwie niezależne zmienne (prywatne – każdego procesora wykonującego swój program) znalazły się we wspólnej linii *cache*. Każdy z procesorów operuje tylko na jednej zmiennej, ale za każdym razem wymieniana jest zawartość całej linii
– razem ze zmienną sąsiadującą.

- Zadanie odpowiedniego rozlokowania danych w przestrzeni adresowej spoczywa zarówno po stronie systemu operacyjnego, jak i programisty
(np. programy/procesy - oddzielne przestrzenie adresowe, wątki – osobne stosy itp.).