Chương 5

Bộ nhớ đệm (Caches)

Nội dung

- Phân cấp bộ nhớ
  - Làm thế nào để tạo ra một bộ nhớ lớn và nhanh?
  - Liên kết SRAM, DRAM, và đĩa cứng
- Caching
  - Những bộ nhớ nhỏ lưu những dữ liệu quan trọng
  - Ví dụ
- Bộ nhớ cache làm việc như thế nào?
  - Các thẻ: Tags
  - Các khối: Blocks (lines)
- Thực thi
  - 3 loại cache: kết hợp toàn phần (Fully-associative), kết hợp theo tập hợp (set-associative), ánh xạ trực tiếp (direct-mapped)

- Hiệu năng

Đặt vấn đề

Disks are big, but super slow

SRAM is fast, but small

	Capacity	Latency	Throughput	Cost
Disk	3TB	8 ms	200 MB/s	$0.07/GB
Flash	256GB	85 µs	500 MB/s	$1.48/GB
DRAM	16GB	65 ns	10,240 MB/s	$12.50/GB
SRAM	8MB	13 ns	26,624 MB/s	$7,200/GB
SRAM	32kB	1.3 ns	47,104 MB/s	

- Cần bộ nhớ lớn và nhanh
  - Bộ nhớ lệnh lớn ISA : 232 memory address (4GB)
  - Yêu cầu nhanh vì 33% các lệnh là loads/stores và 100% các lệnh cần phải tải về thanh ghi lệnh

- Tồn tại bộ nhớ có thể có dung lượng lớn và truy nhập nhanh?

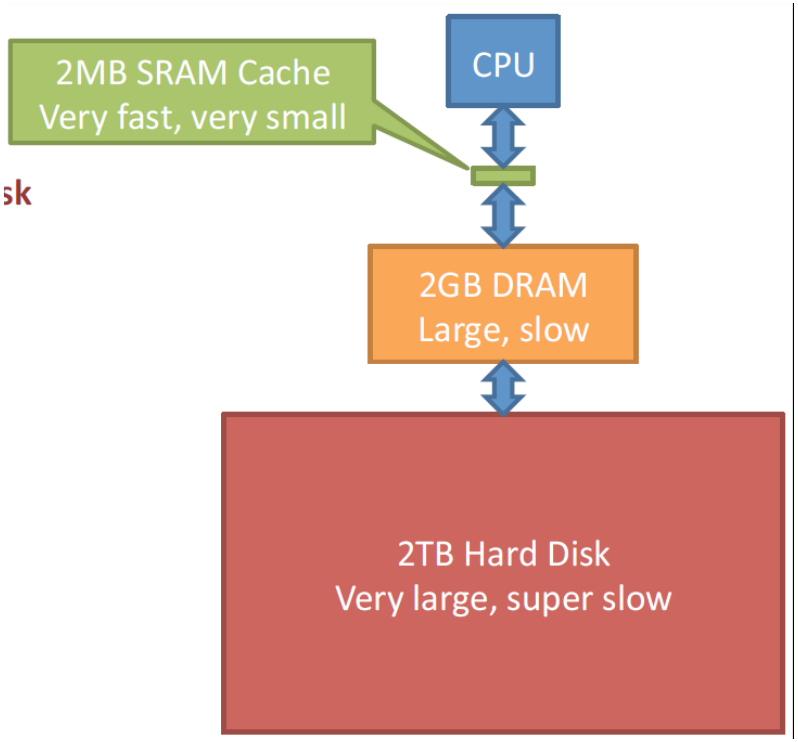
Bộ nhớ lớn và nhanh

- Các loại bộ nhớ đã có?
  - Hard disk: **Huge** (1000 GB) **Super slow** (1M cycles)
  - Flash: **Big** (100 GB) **Very slow** (1k cycles)
  - DRAM: **Medium** (10 GB) **Slow** (100 cycles)
  - SRAM: **Small** (10 MB) **Fast** (1-10 cycles)
- Cần bộ nhớ **nhanh và lớn**
  - Không thể sử dụng **SRAM** (too **small**)
  - Không thể sử dụng **DRAM** (too **slow** and **small**)
  - Không thể sử dụng **Flash/Hard disk** (way too **slow**)
- Có thể kết nối giữa chúng:
  - **Speed** từ (small) **SRAMs**
  - **Size** từ (big) **DRAM** và **Hard disk**

Xây dựng một phân cấp sử dụng công nghệ khác để tận dụng các ưu điểm của các bộ nhớ có sẵn.

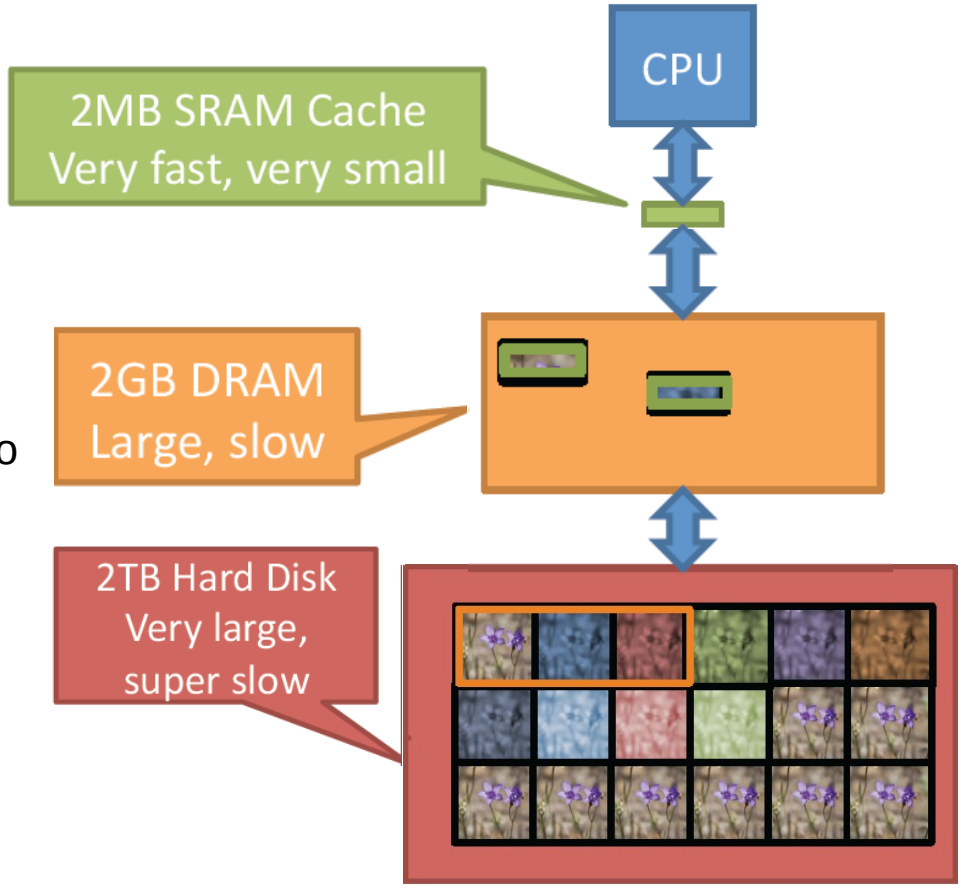
Phân cấp bộ nhớ

- **Phân loại:**
  - Dung lượng nhỏ và nhanh: SRAM
  - Chậm: DRAM
  - Đĩa cứng dung lượng lớn nhưng rất chậm
- **Viễn cảnh**:
  - Rất lớn
  - Rất nhanh (on average)
- **Mục tiêu?**
  - Lưu trữ thông tin quan trọng trong bộ nhớ nhanh.
  - Di chuyển những thông tin không quan trọng vào bộ nhớ chậm

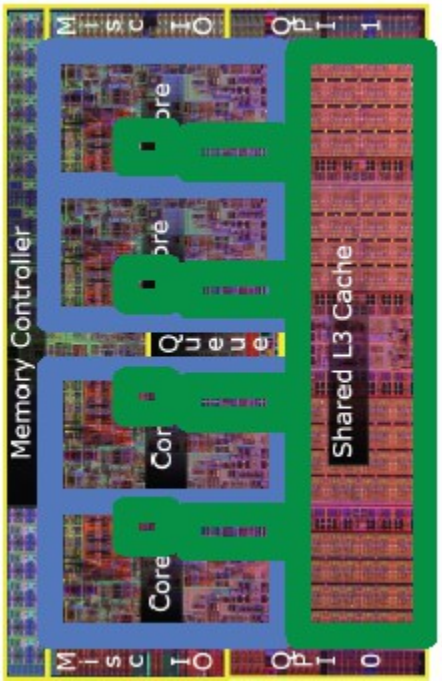


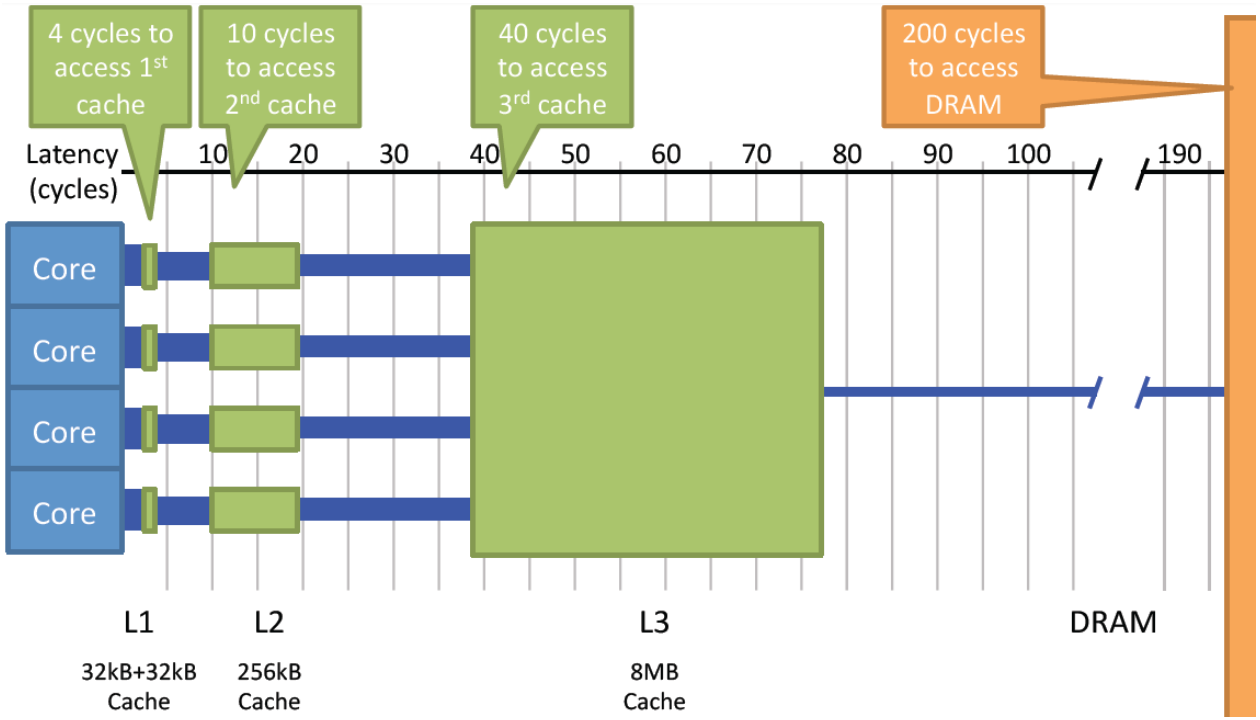
Ví dụ: sửa video

- Video dung lượng lớn (lớn hơn DRAM)
- **Lưu vào ổ cứng**
- **Tải phần cần chỉnh sửa vào DRAM**
- **CPU tải dữ liệu để xử lý vào cache.**
- Di chuyển dữ liệu mới vào DRAM và cache khi xử lý video
- **Chú ý:**
  - Lưu những dữ liệu quan trọng vào bộ nhớ nhanh
  - Di chuyển những dữ liệu không quan trọng vào bộ nhớ chậm

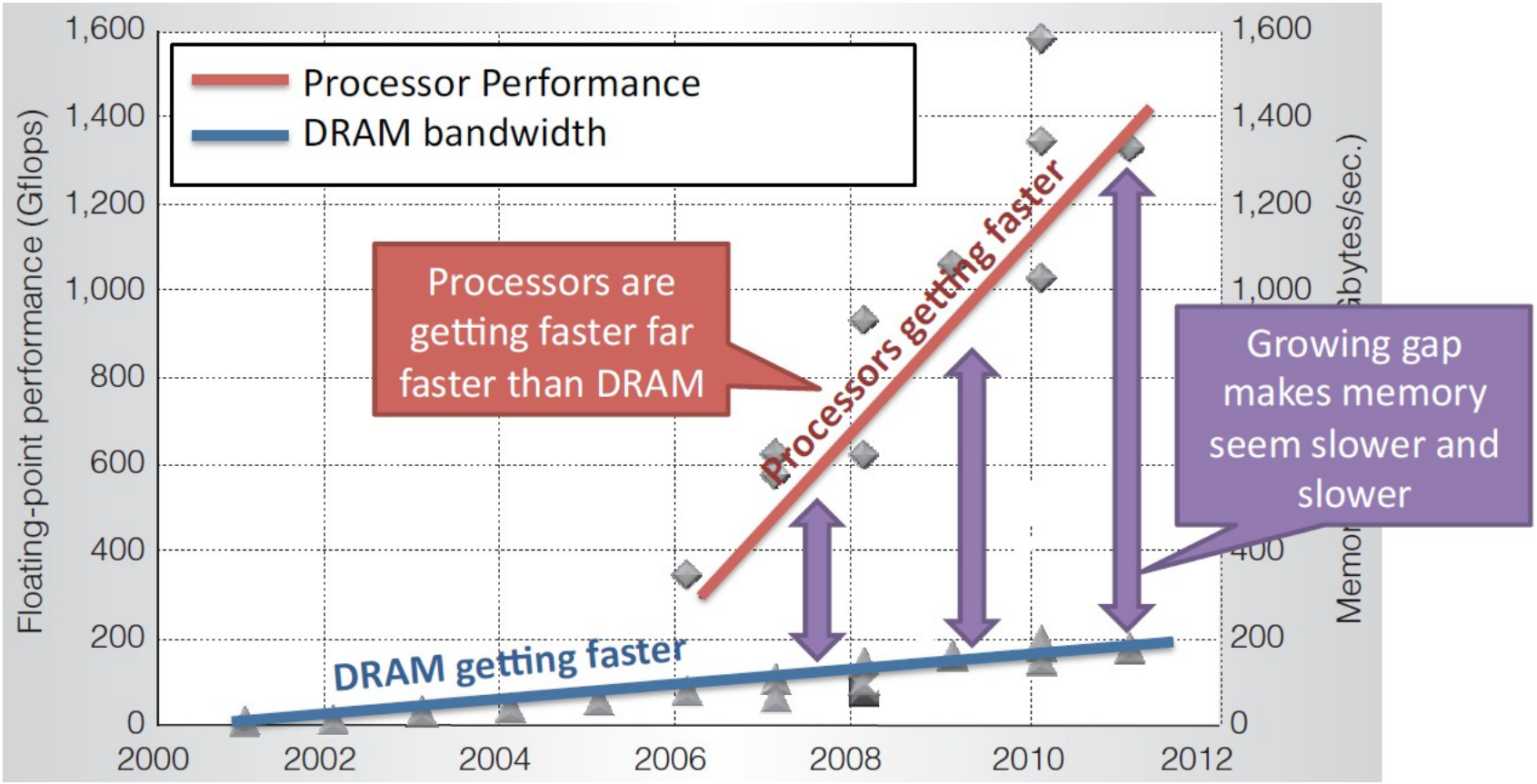


Phân cấp bộ nhớ ngày nay (Intel Nehalem)





So sánh sự phát triển công nghệ...



Làm thế nào để SRAMs có dung lượng lớn hơn, DRAM truy cập nhanh hơn?

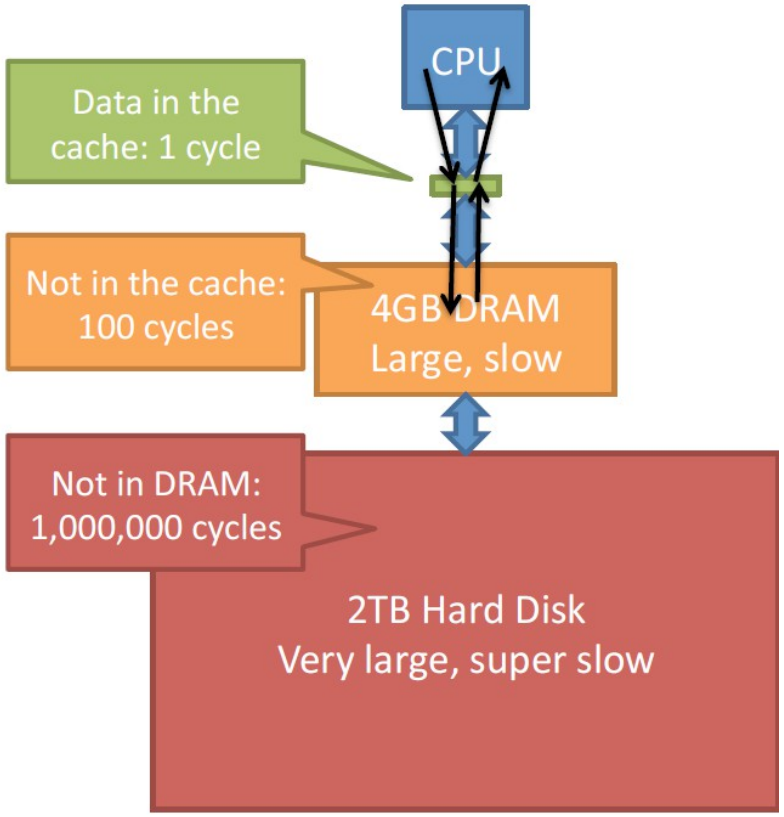
Các ý tưởng cơ bản về cache

• Đặt những dữ liệu quan trọng trong bộ nhớ nhỏ và nhanh **(cache).**

• Nếu truy cập (load/store) những dữ liệu quan trọng, cần thực hiện nhanh.

• Nếu truy cập (load/store) những dữ liệu khác, dịch chuyển dữ liệu vào trong cache.

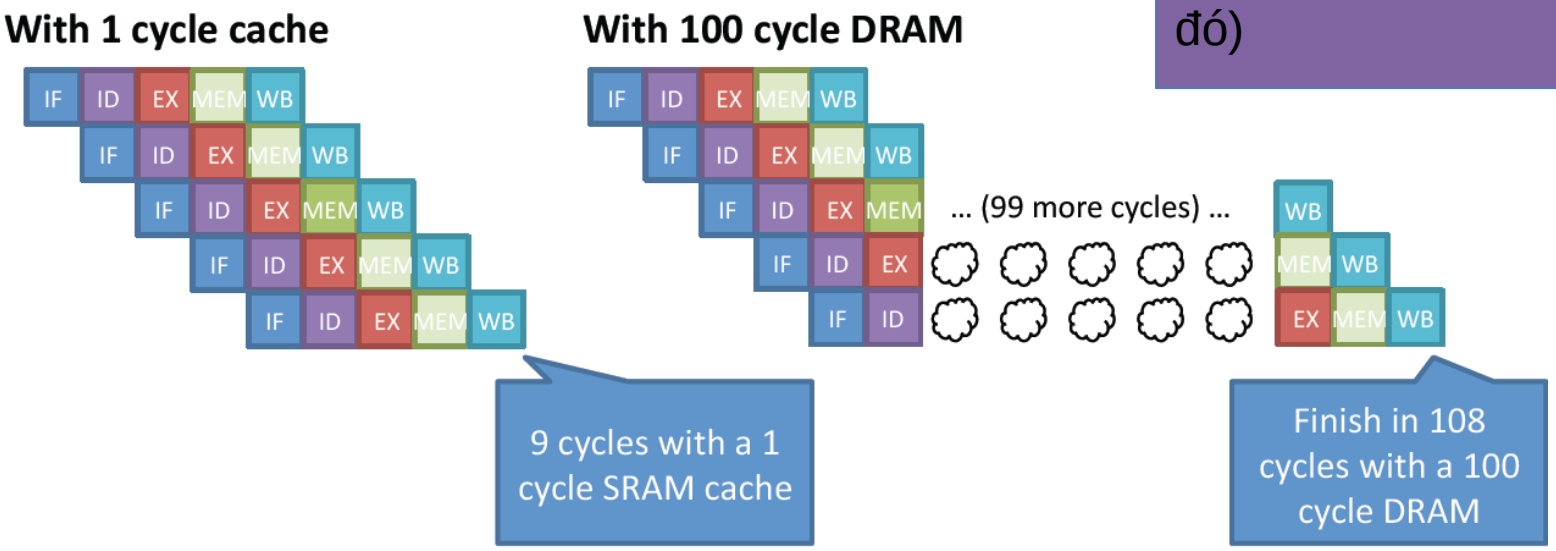
• Nếu đặt chính xác dữ liệu cần dùng vào cache, khi đó hầu hết các truy cập sẽ tìm ra dữ liệu hữu ích trong cache và trở nên nhanh hơn.



Hiệu năng của caches

- Truy nhập dữ liệu trong DRAM hết 100 chu kỳ
- Truy nhập dữ liệu trong Cache (SRAM) hết 1 chu kỳ
- Tỷ lệ lệnh load/stores là 33%.

Bỏ qua việc tải lệnh (Cần thêm bộ nhớ thứ hai sau đó)



Tính toán hiệu năng cache

- **Sử dụng DRAM:**
  - Truy nhập dữ liệu trong DRAM hết 100 chu kỳ
  - (33% tải/lưu dữ liệu)*100 chu kỳ = 33 chu kỳ truy cập bộ nhớ/ lệnh.

- **Sử dụng một SRAM cache hoàn hảo (dữ liệu trong cache 100% thời gian):**
  - (33% tải/lưu dữ liệu)*1 chu kỳ = 0.33 chu kỳ truy cập bộ nhớ / lệnh.

- **Sử dụng SRAM cache thực tế hơn (dữ liệu ở trong 90% thời gian):**
  - (33% tải/lưu dữ liệu)*(1 chu kỳ 90% thời gian + 100 chu kỳ 10% thời gian)
    =0.33*(1*0.9+100*0.1) =0.33*(1.9) = 0.67 chu kỳ truy cập bộ nhớ / lệnh

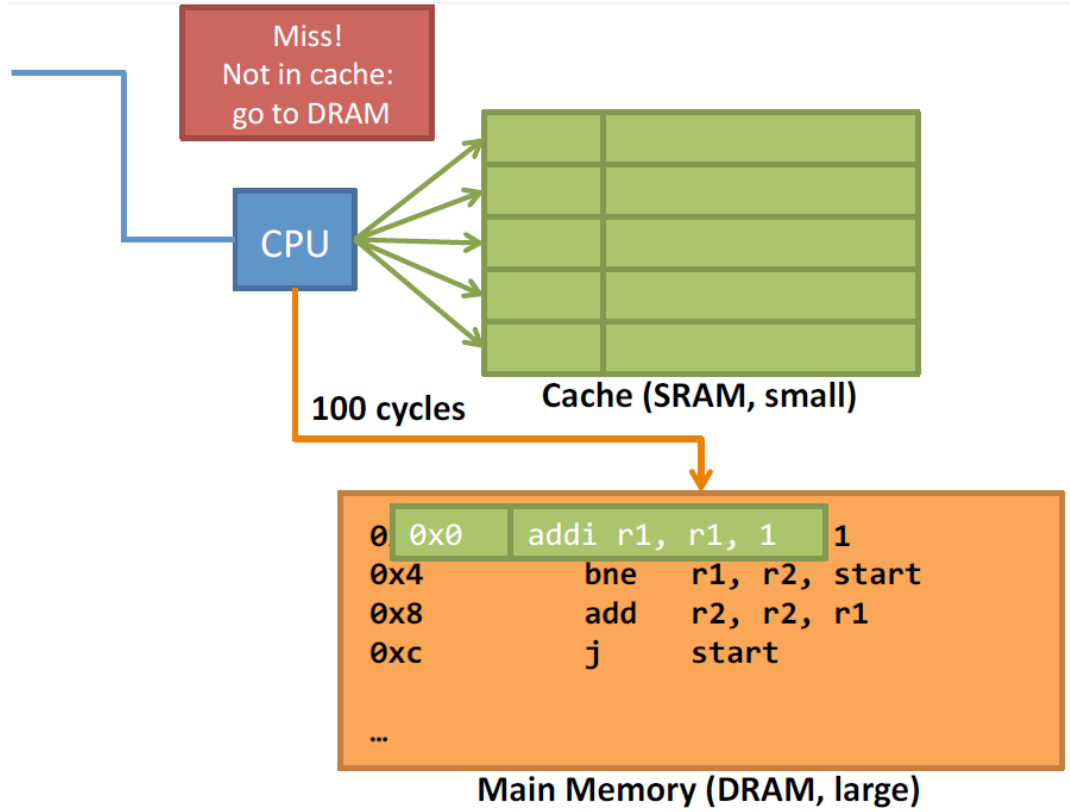
Ví dụ: bộ nhớ đệm

```
0x0 start:  addi  r1, r1, 1
0x4         bne   r1, r2, start
0x8         add   r2, r2, r1
0xc         j     start

…
```

Main Memory (DRAM, large)

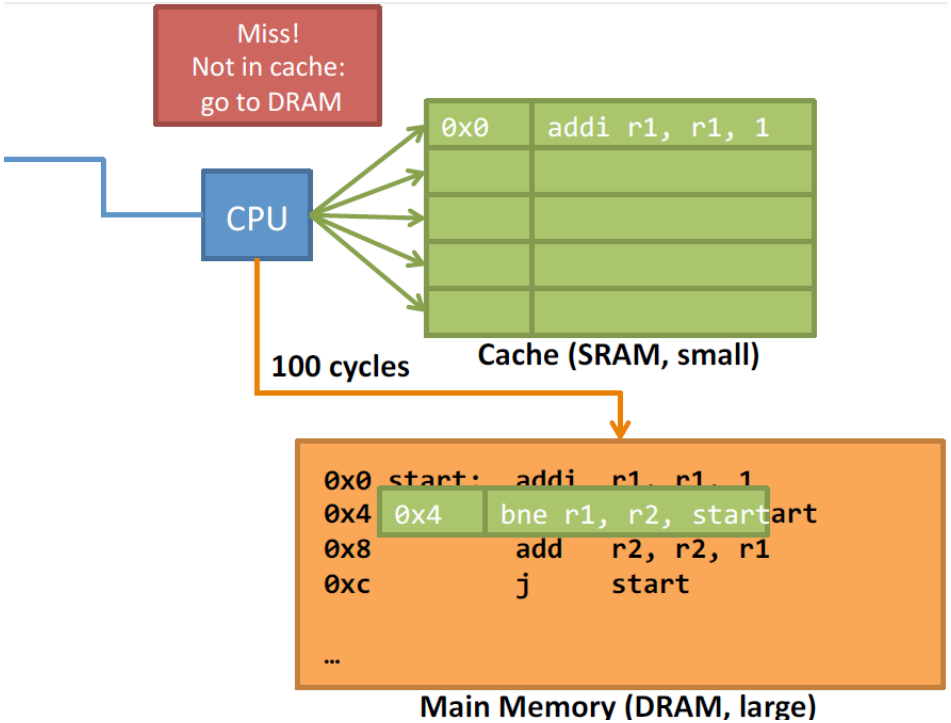
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2



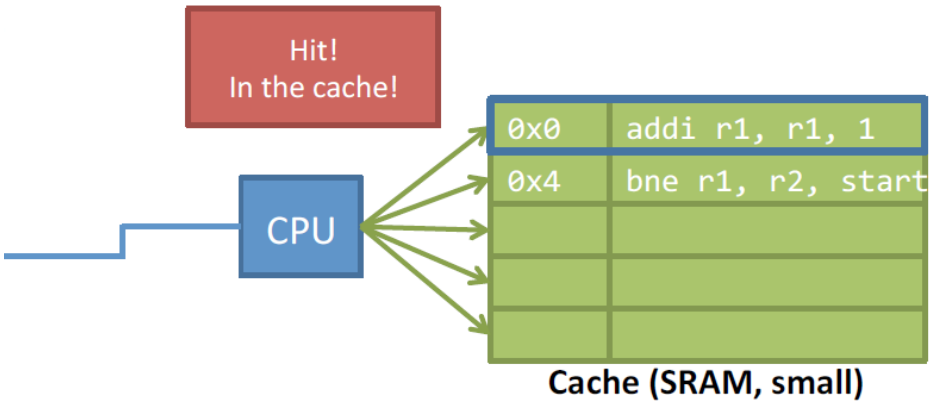
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2



Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2

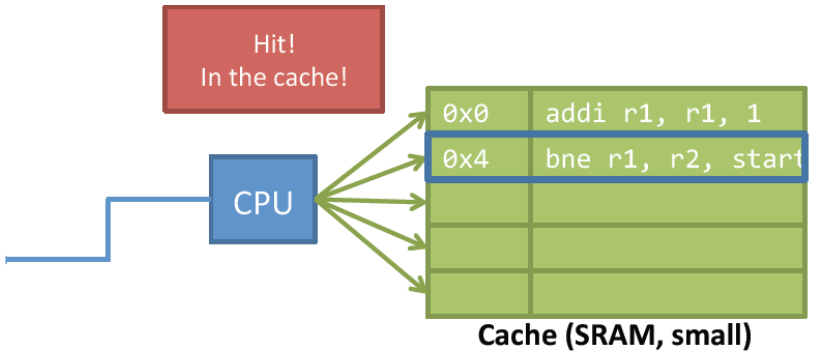



```
0x0 start:  addi  r1, r1, 1
0x4         bne   r1, r2, start
0x8         add   r2, r2, r1
0xc         j     start

…
```

Main Memory (DRAM, large)

Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2

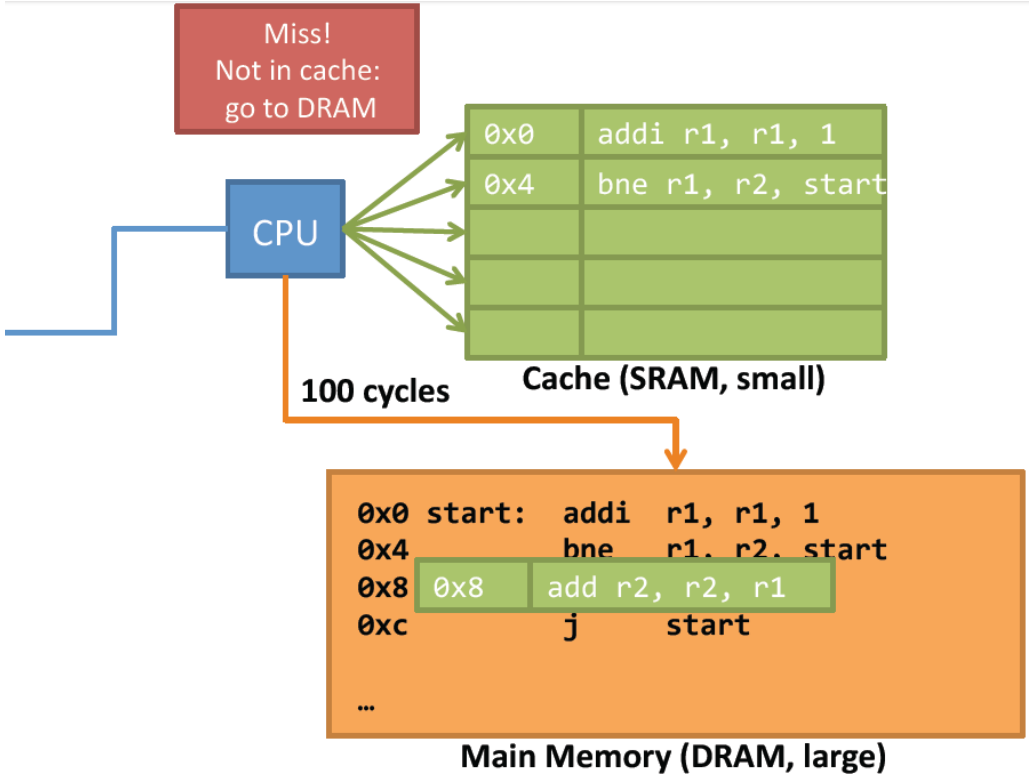



```
0x0 start:  addi  r1, r1, 1
0x4         bne   r1, r2, start
0x8         add   r2, r2, r1
0xc         j     start

…
```

Main Memory (DRAM, large)

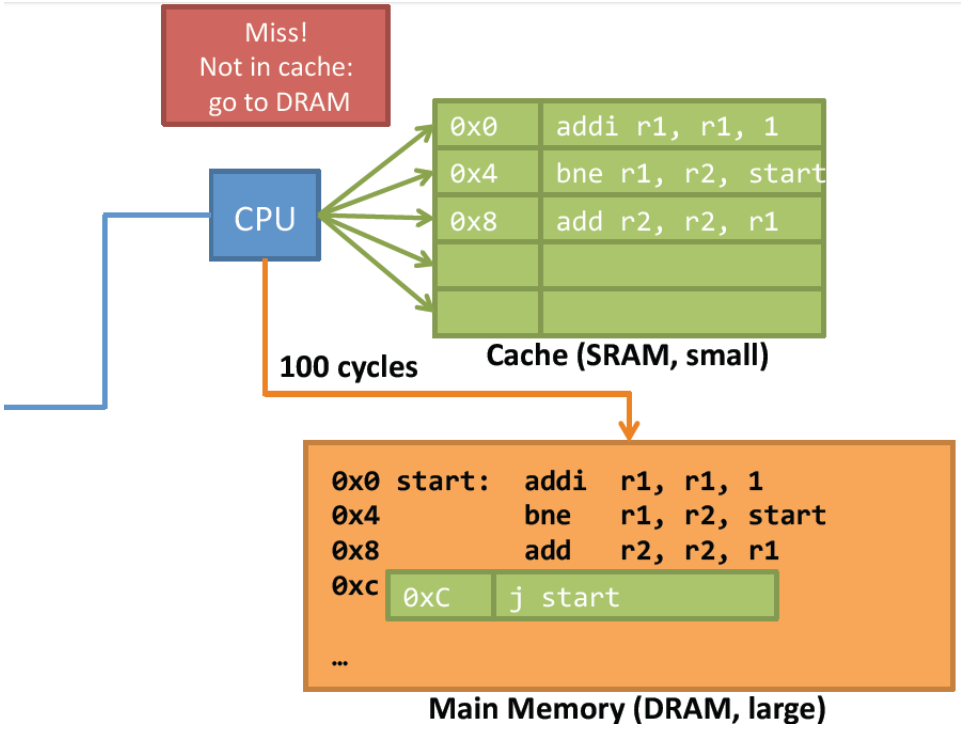
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4



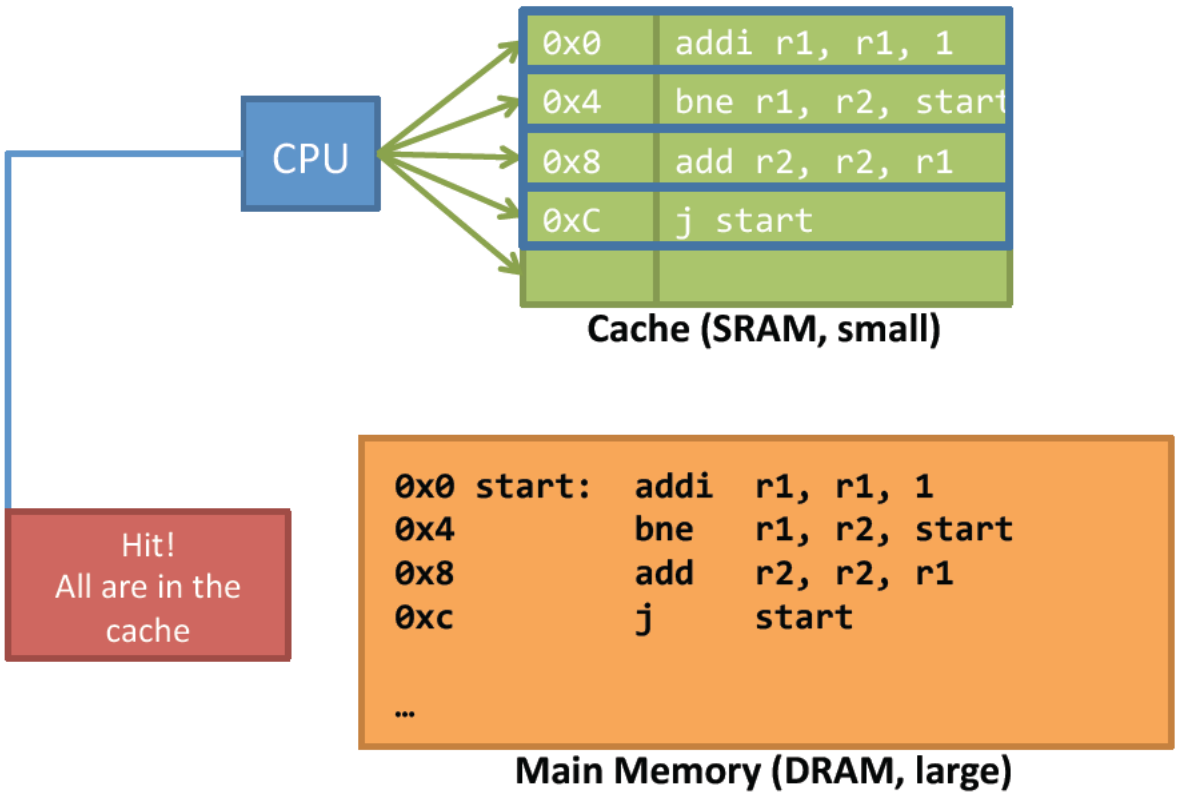
Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4
305	0xC	FETCH	2	4
…				
405	0xC	j	2	4



Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4
305	0xC	FETCH	2	4
…				
405	0xC	j	2	4
406	0x0	addi	3	4
407	0x4	bne	3	4
408	0x0	addi	4	4
409	0x4	bne	4	4
410	0x8	add	4	8
411	0xC	j	4	8
412	0x0	addi	5	8

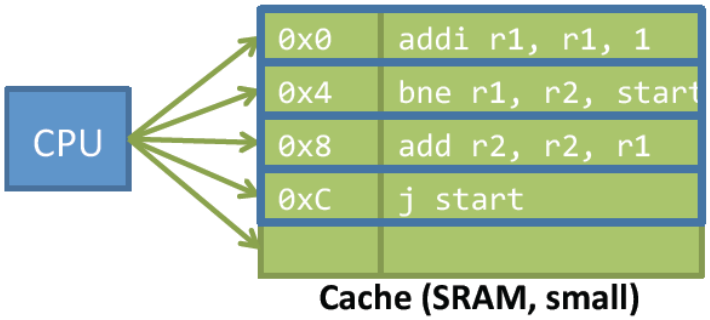


Example: caching instructions

Cycle	Addr	Inst	r1	r2
0	0x0	FETCH	0	2
…				
100	0x0	addi	1	2
101	0x4	FETCH	1	2
…				
201	0x4	bne	1	2
202	0x0	addi	2	2
203	0x4	bne	2	2
204	0x8	FETCH	2	2
…				
304	0x8	add	2	4
305	0xC	FETCH	2	4
…				
405	0xC	j	2	4
406	0x0	addi	3	4
407	0x4	bne	3	4
408	0x0	addi	4	4
409	0x4	bne	4	4
410	0x8	add	4	8
411	0xC	j	4	8
412	0x0	addi	5	8

100 cycles per instruction: **in DRAM**

1 cycle per instruction: **all in cache**



Cache (SRAM, small)

```
0x0 start:  addi  r1, r1, 1
0x4         bne   r1, r2, start
0x8         add   r2, r2, r1
0xc         j     start

…
```

Main Memory (DRAM, large)

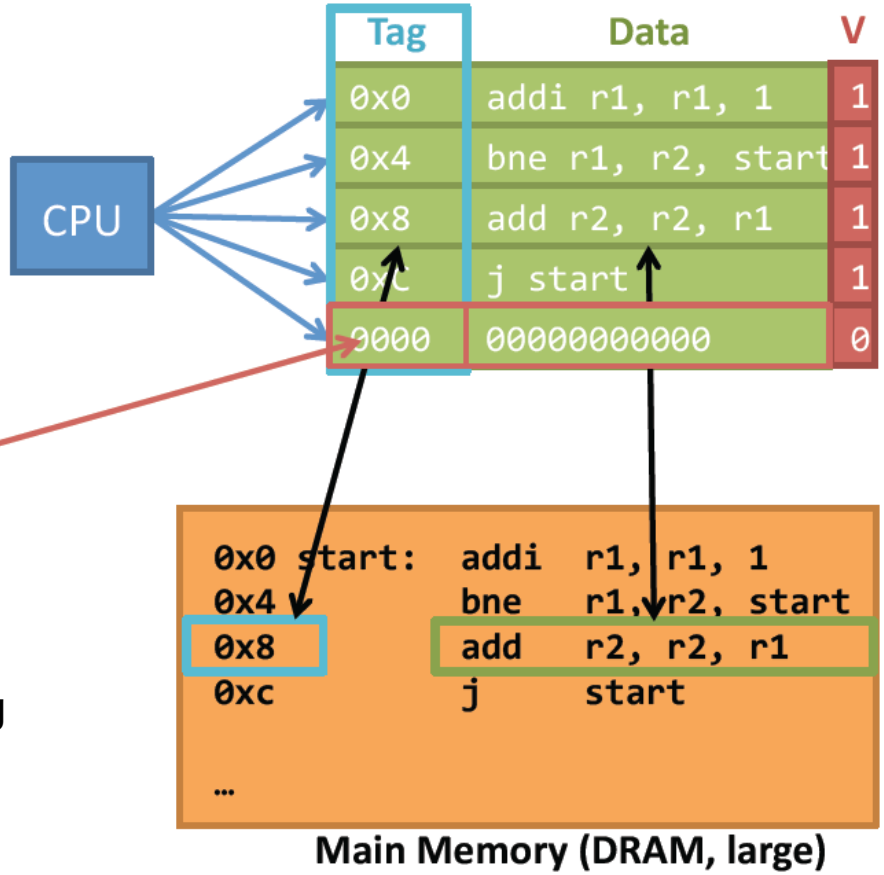
Cache làm việc như thế nào?

- Trả lời các câu hỏi sau:

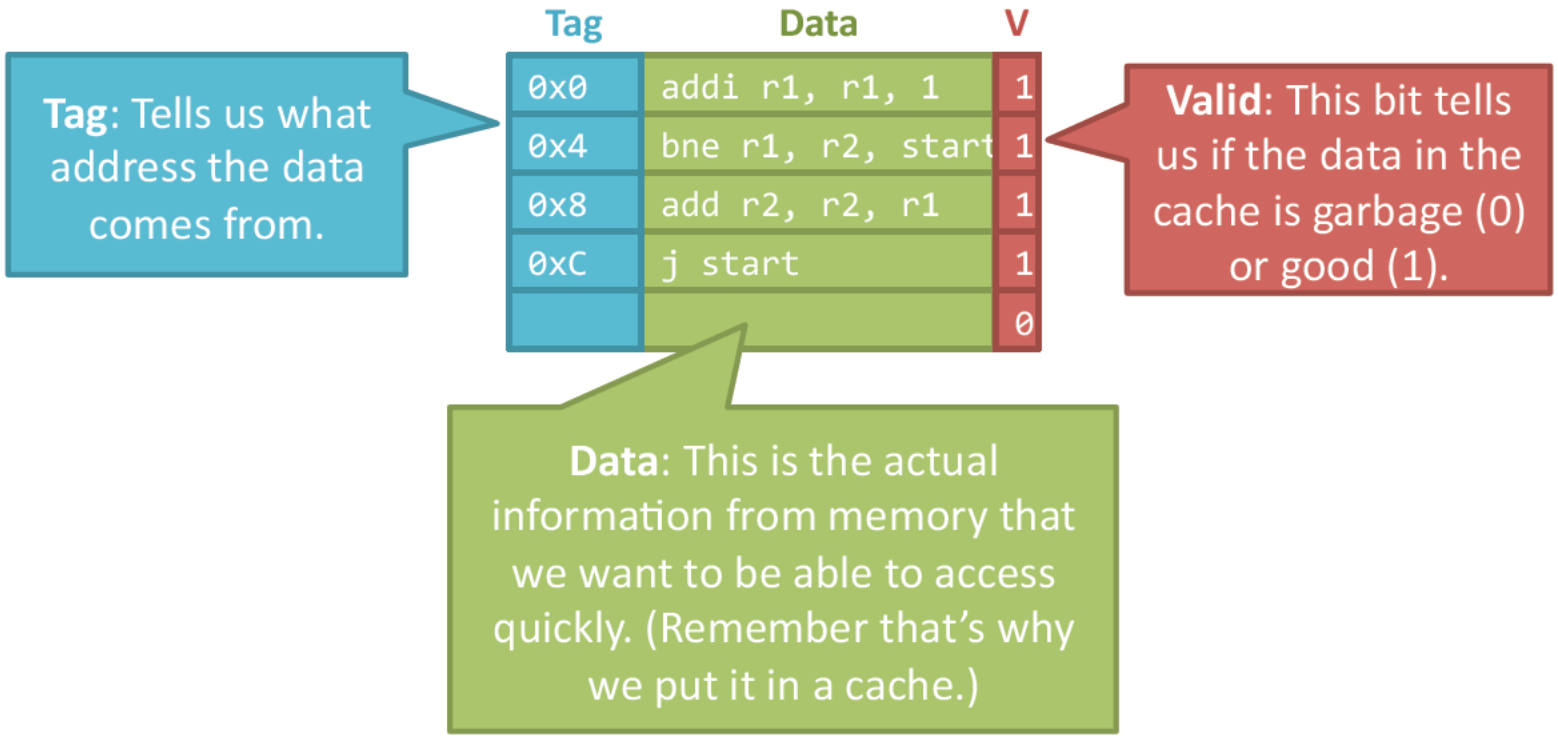
1. Sắp xếp khối trong cache như thế nào?
2. Làm thế nào để biết sự hiện diện của một khối trong cache – tìm kiếm và nhận diện cache.
3. Khối nào được thay thế trong trường hợp tìm kiếm dữ liệu trong cache thất bại.
4. Ghi vào bộ nhớ - ghi dữ liệu vào bộ nhớ như thế nào -> chiến thuật ghi.

Cache làm việc như thế nào?

- **Bộ nhớ kết hợp:**
  - Lưu trữ một thẻ (**tag) để chỉ ra định vị bộ nhớ trong cache.**
  - Các thẻ (**tag) là địa chỉ hoặc một phần địa chỉ của dữ liệu được định vị trong cache.**
  - Phần còn lại của cache là lưu trữ dữ liệu
- **Làm thế nào để biết dữ liệu ở trạng thái sẵn sàng?**
  - Cache bắt đầu tại thẻ có giá trị bằng 0
  - **Làm thế nào biết có hay không lệnh ở vị trí địa chỉ 0?**
- Bit đánh dấu - **Valid/Invalid bit**
  - Thêm bit 1 khi dữ liệu là hợp lệ và bit 0 nếu không hợp lệ.
  - Nếu thẻ ánh xạ đến có bit đánh dấu bằng 0 (invalid (0)), bỏ qua dữ liệu



Cấu tạo cache

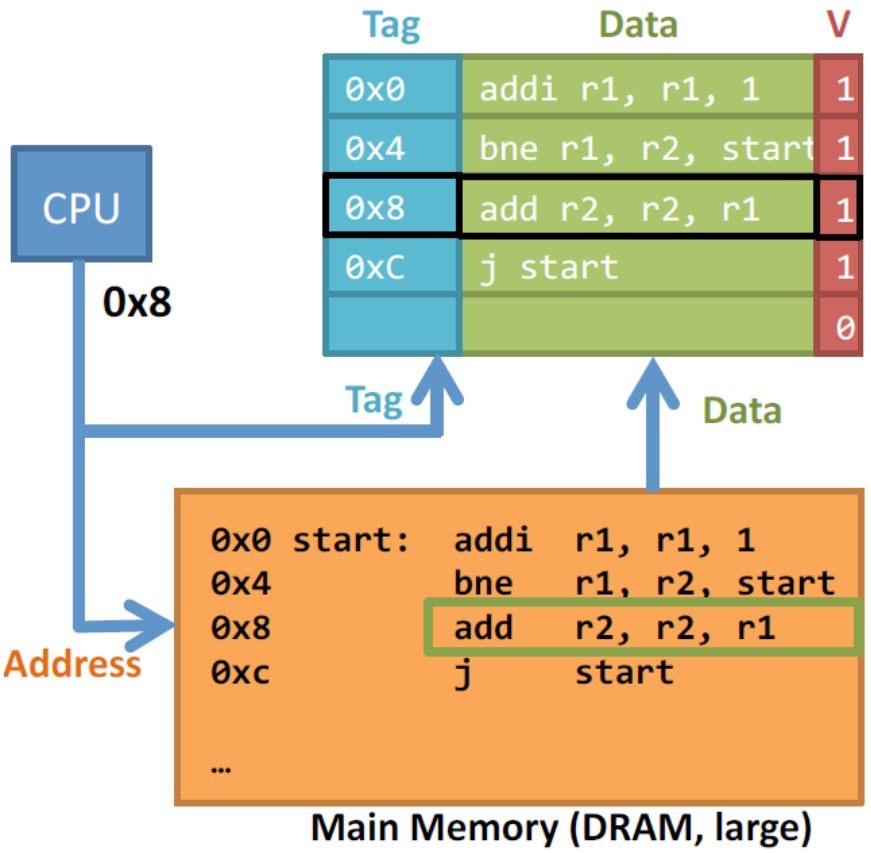


Lưu trữ dữ liệu trong cache

- Tải dữ liệu từ **Bộ nhớ**
- Lưu thẻ (**tag)** (0x8)
- Lưu dữ liệu (**data)** (add r2, r2, r1)
- Đặt giá trị **valid bit** (1)

Q: Khi nào các bit đánh dấu bị xóa?
1/ Không bao giờ
2/ Khi xóa bộ nhớ cache
3/ Sau khi tải về

A: 2
Cache bị xóa khi chuyển đổi giữa các chương trình hoặc để máy tính ở chế độ chờ. Cần xóa cache bởi vì dữ liệu sẽ không còn hợp lệ.

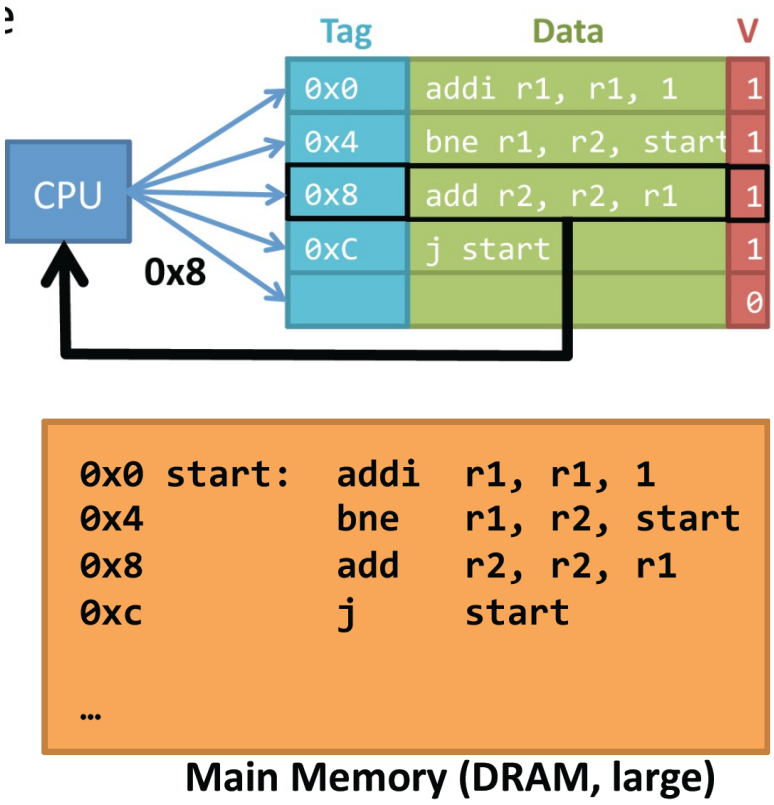


Truy cập dữ liệu từ cache

- Kiểm tra thẻ **tag** (0x8) trong cache
- Kiểm tra nếu bit hợp lệ bằng 1
- Đọc dữ liệu từ cache.

Q: Sẽ làm gì nếu bit đánh dấu bằng 0
1/ Bỏ qua dữ liệu.
2/ Đặt lại bằng 1.
2/ Lấy dữ liệu từ DRAM.

A: Lấy dữ liệu từ DRAM
Nếu các bit hợp lệ không được đặt giá trị có nghĩa là dữ liệu trong cache không hợp lệ. Trong trường hợp này cần lấy dữ liệu từ DRAM đưa vào cache.

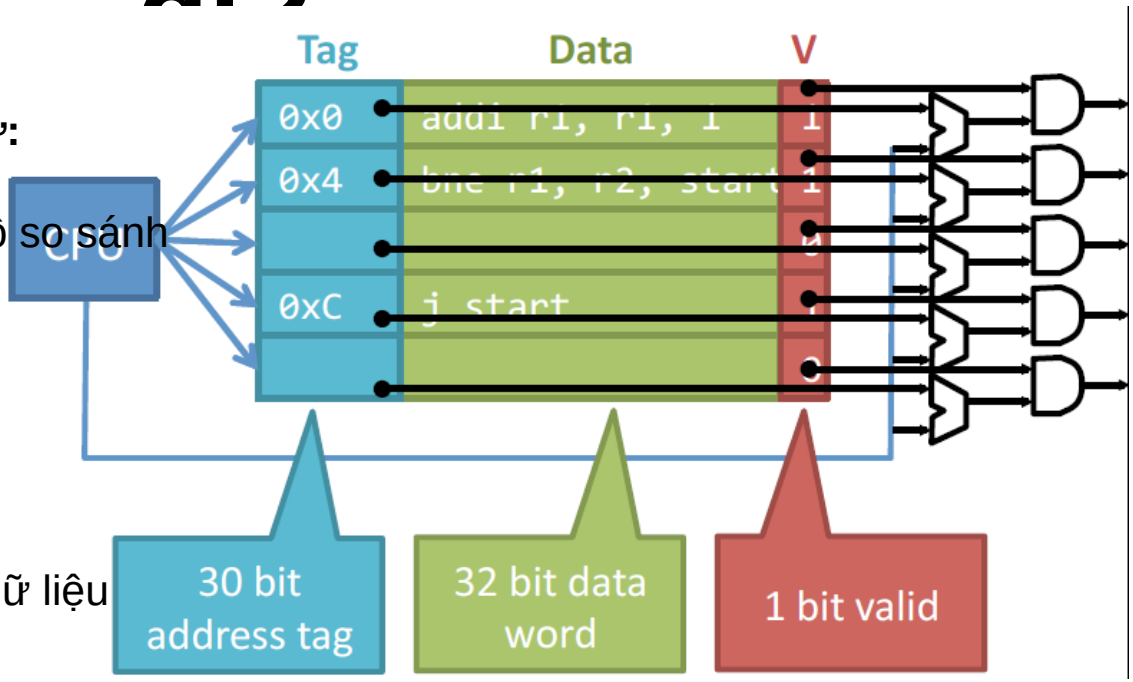


Cache blocks (lines)

Tăng hiệu năng bằng cách lưu trữ nhiều dữ liệu hơn ở mỗi tag

Hiệu năng sử dụng cache là gì?

- **Tìm kiếm một Cache n-phần tử:**
  - Cùng một vị trí → n chu kỳ
  - Cùng một thời điểm → n bộ so sánh
  - **Tốn kém!**
- Sử dụng bộ nhớ**:**
  - Data: 32 bits (one word)
  - Tag: 30 bits (one address)
  - Valid: 1bit
  - 63 bits cho mỗi phần tử
  - Chỉ có 32 sử dụng để lưu dữ liệu
  - **Rất không hiệu quả!**



Q: Bao nhiêu bit cần cho một địa chỉ thẻ (tag)?

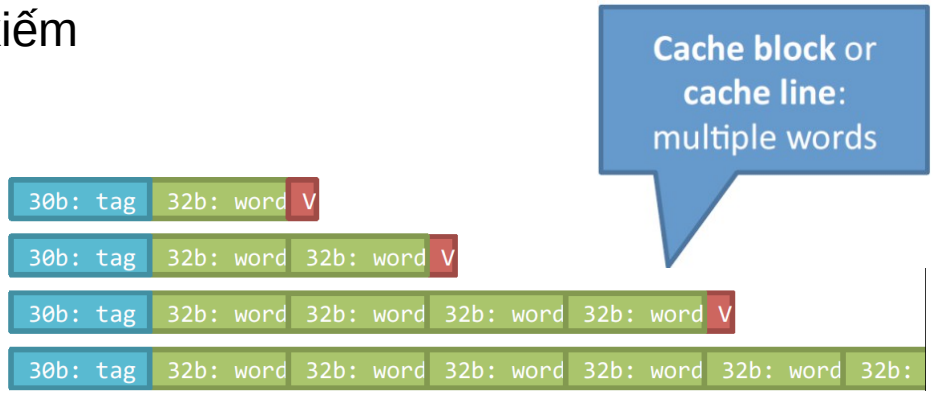
1. 4
2. 30
3. 32

A: 30
Địa chỉ thẻ xác định địa chỉ dữ liệu trong cache, cần bits để nhận diện địa chỉ từ.
(We would need 32 bits if the memory was not word-aligned.)

Tốn quá nhiều không gian cho thẻ

- **Hai phí tổn của caches:**
  - Tốn nhiều không gian cho thẻ (tags)
  - Tốn nhiều phần tử logic để tìm kiếm

- **Khắc phục vấn đề:**
  - 1 word / tag: 1/2 wasted
  - 2 words / tag: 1/3 wasted
  - 4 words / tag: 1/5 wasted
  - 8 words / tag: 1/9 wasted
  - 16 words / tag: 1/17 wasted (được sử dụng trong bộ xử lý ngày nay)

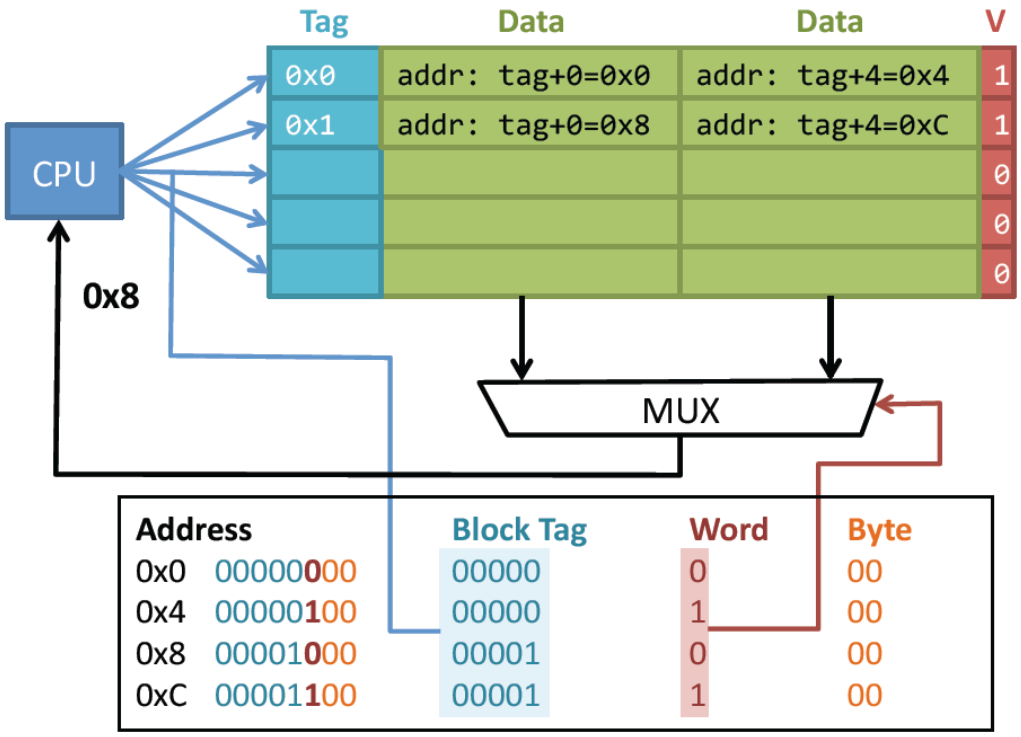


- **Phương pháp khắc phục?**
  - Các dữ liệu trong cache lớn hơn → ít thẻ hơn → hiệu năng lưu trữ tốt hơn

Truy nhập dữ liệu từ cache

- Tương tự như trước:
  - Kiểm tra nếu thẻ **tag** (0x8) ở trong cache
  - Kiểm tra nếu bit đánh dấu là 1
  - Đọc dữ liệu từ cache

- **Nhưng có nhiều từ hơn trong một khối cache:**
  - Sử dụng địa chỉ các bit để chọn từ đúng với bộ dồn kênh MUX
  - Tag chỉ so sánh các bit là như nhau trong toàn bộ khối



Đánh địa chỉ cho các cache có kích thước khối khác nhau

- **Địa chỉ:**
  - 2 bit cuối : xác định byte trong từ
    (không cần thiết bởi vì truy nhập bộ nhớ bằng các từ)
  - N bit tiếp theo: xác định từ trong khối
    (ví dụ, kích thước khối là 4 cần 2 bits để xác định từ)
  - 32-N-2 bit còn lại: tag
    (cần thiết để xác định địa chỉ bộ nhớ)

1 word cache block

TTTT TTTT TTTT TTTT
TTTT TTTT TTTT TTBB
30 tag, 2 byte

(1 bit to choose word)

2 word cache block

TTTT TTTT TTTT TTTT
TTTT TTTT TTTT TWBB
29 tag, 1 word, 2 byte

(2 word bits to choose from 4 words)

4 word cache block

TTTT TTTT TTTT TTTT
TTTT TTTT TTTT WWBB
28 tag, 2 word, 2 byte

Ví dụ: cache block size

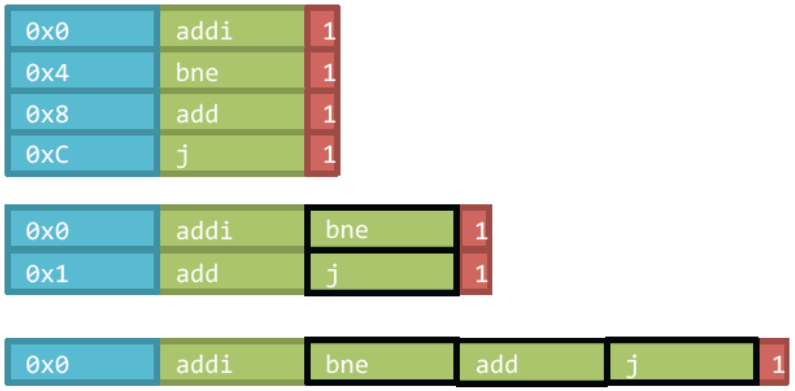
Addr	Inst	r1	r2	1w	2w	4w
0x0	addi	1	2	FETCH	FETCH	FETCH
0x4	bne	1	2	FETCH		
0x0	addi	2	2			
0x4	bne	2	2			
0x8	add	2	4	FETCH	FETCH	
0xC	j	2	4	FETCH		
0x0	addi	3	4			
0x4	bne	3	4			
0x0	addi	4	4			
0x4	bne	4	4			
0x8	add	4	8			
0xC	j	4	8			
0x0	addi	5	8			

Q: Tại sao kích thước block 4 từ là tốt ?

1. Ít tốn không gian hơn
2. Tải trước các dữ liệu cần dùng sau đó
3. Thẻ ít bit hơn.

A: 2

Bằng việc tải 3 từ tiếp theo cùng với từ cần dùng, tỷ lệ thành công cao hơn. Phương pháp này được gọi là không gian “cục bộ - địa phương”: Những dữ liệu gần với những dữ liệu vừa sử dụng là những dữ liệu hợp lệ nhất.



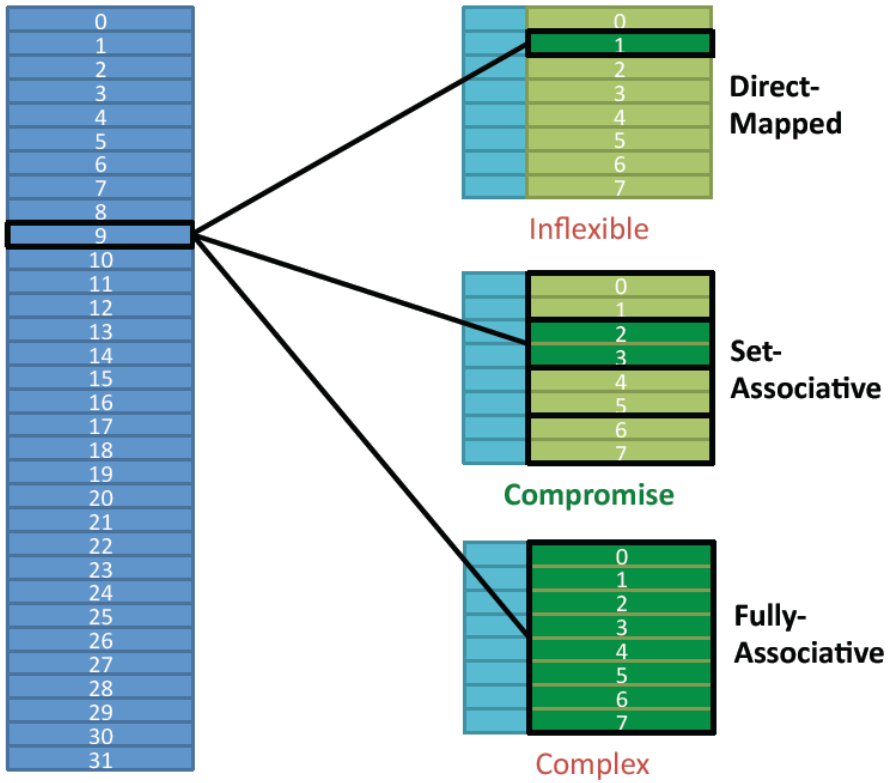

```
0x0 start:  addi  r1, r1, 1
0x4         bne   r1, r2, start
0x8         add   r2, r2, r1
0xc         j     start

…
```

Main Memory (DRAM, large)

Các khối cache lớn hơn
liên kết theo dòng/khối

- **Ưu điểm:**
  - **Ít tốn không gian cho các thẻ** (higher % data)
  - **Quy định cục bộ**:
    - N-1 words tiếp theo có sẽ được sử dụng sau đó
- **Nhược điểm:**
  - Nếu không sử dụng dữ liệu trong cache sẽ làm tốn không gian dữ liệu
  - Ví dụ:
    - for (int i=0; i<100; **i+=2**) {
      a[i]++;
      }
    - Tốn một nửa không gian
  - 64 bytes (8 words) là kích thước chuẩn
  - Intel luôn nạp 2 khối cache tại một thời điểm tương đương với 128 bytes hay 16 words

Các loại caches



Địa chỉ tương ứng một khối
(9 mod **8**) = 1

Tìm kiếm dễ dàng, không linh hoạt

Địa chỉ tương ứng với một tập .
(9 mod **4**) = Set 1

Tìm kiếm tốt hơn, linh hoạt hơn

Địa chỉ tương ứng với bất kỳ vị trí nào trong cache.

Khó trong việc tìm kiếm, rất linh hoạt

Bộ đệm liên kết toàn khối full – associative

• Tìm kiếm khối
– Cần **n bộ so sánh**
– Hoặc cần **n chu kỳ**

• Mục tiêu?
– Có thể đặt dữ liệu ở bất kỳ đâu trong cache: **flexibility**
– N blocks → có thể lưu dữ n mảng dữ liệu, mỗi vị trí đặt khối có thể chứa một trong một trong tất cả các khối trong bộ nhớ.

Q: Nhược điểm?

1. Ít lãng phí không gian
2. Chỉ cần một bộ so sánh
3. Chỉ một block được tìm kiếm

A: Chỉ một block được tìm kiếm
Tìm kiếm toàn bộ cache, với n bộ so sánh hoặc với một bộ sánh trong n chu kỳ

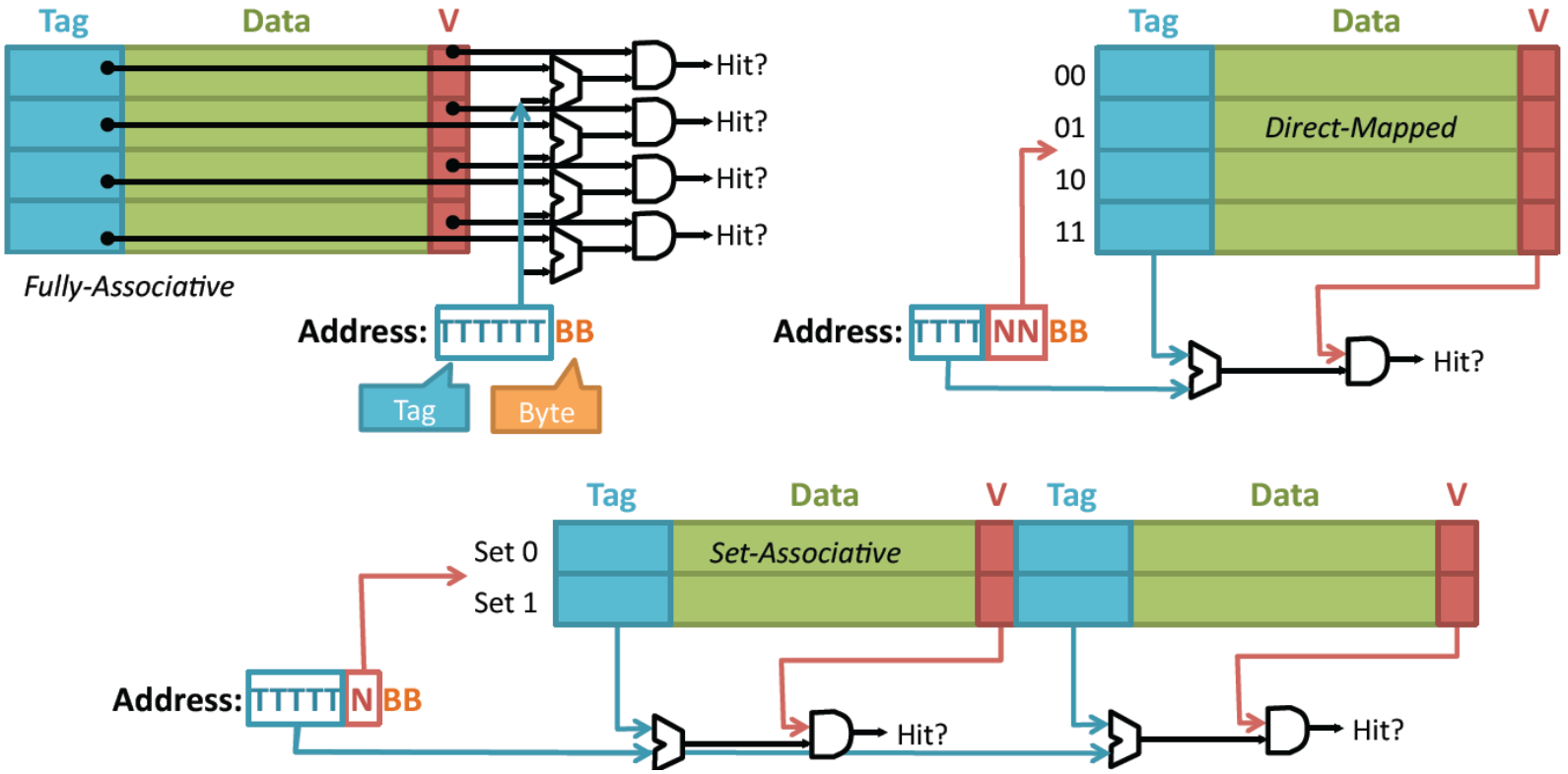
Ưu nhược điểm của các loại cache

- **Fully-associative**
  - Ưu điểm: **linh hoạt**, dữ liệu có thể đặt ở bất kỳ vị trí nào (no conflicts)
  - Nhược: Khó khăn trong việc tìm kiếm (slow/expensive)

- **Set-associative**
  - Ưu: **tương đối linh hoạt**: dữ liệu có thể đặt ở bất kỳ vị trí nào trong tập
  - Nhược: Chỉ cần tìm kiếm một vài vị trí trong tập (reasonable speed/complexity)

- **Direct-mapped**
  - Ưu: Tìm kiếm linh hoạt, chỉ cần tìm kiếm một vị trí (fast/simple)
  - Nhược: xảy ra xung đột dữ liệu (conflicts)

Thiết kế cache



Bộ nhớ đệm ánh xạ trực tiếp

tăng hiệu quả tìm kiếm

Vị trí khối trong bộ đệm ánh xạ trực tiếp

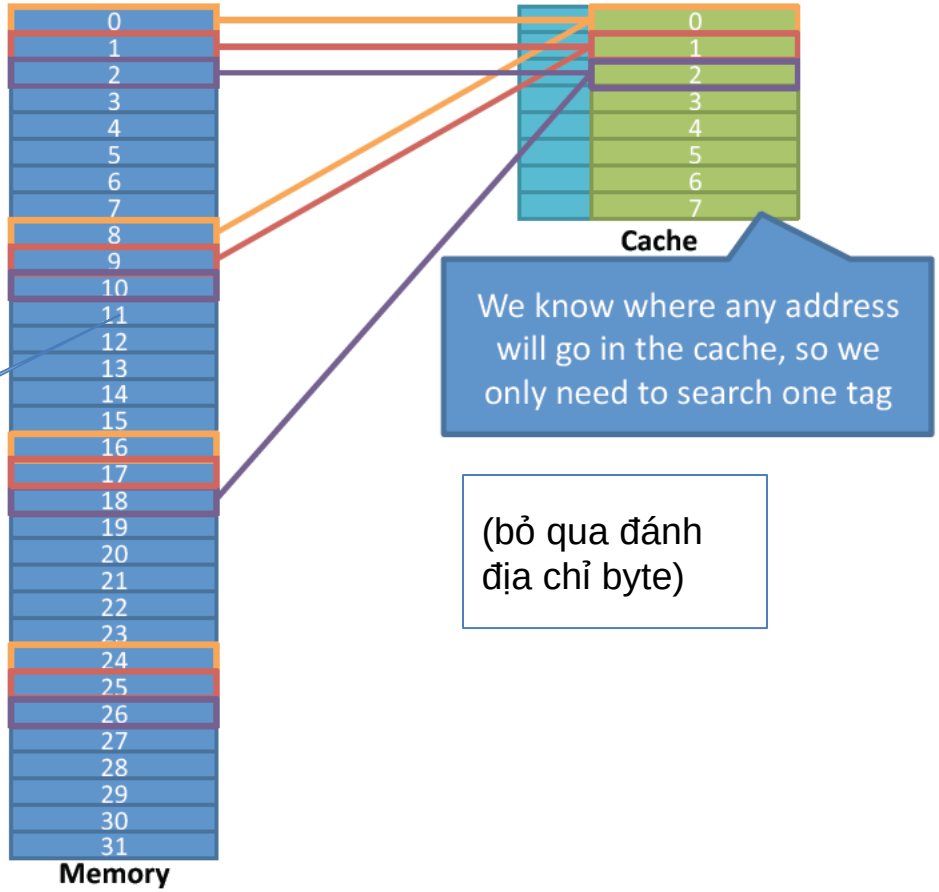
- Mỗi địa chỉ ánh xạ tới một block
- Đánh địa chỉ theo modulo (K = i mod n)

K : vị trí khối đặt trong cache
i : số thứ tự khối trong bộ nhớ trong
n : số khối của cache
Ví dụ:

- 0 → (0 mod 8) = 0
- 1 → (1 mod 8) = 1
- 2 → (2 mod 8) = 2
- 8 → (8 mod 8) = 0
- 9 → (9 mod 8) = 1
- 18 → (18 mod 8) = 2

- Chú ý 3 bits cuối:
  - 0 → **000000** → 0
  - 1 → **000001** → 1
  - 2 → **000010** → 2
  - 8 → **001000** → 0
  - 9 → **001001** → 1
  - 18 → **010010** → 2

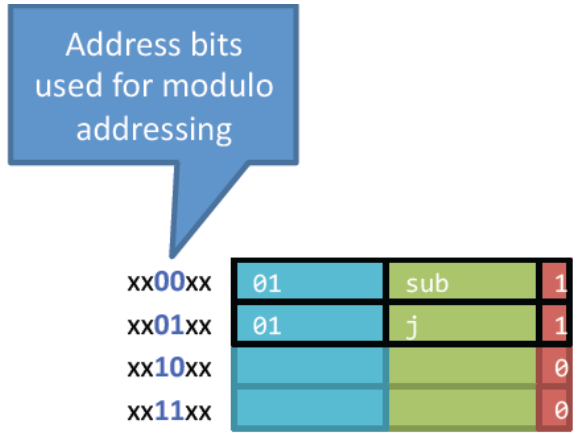
Tag bits

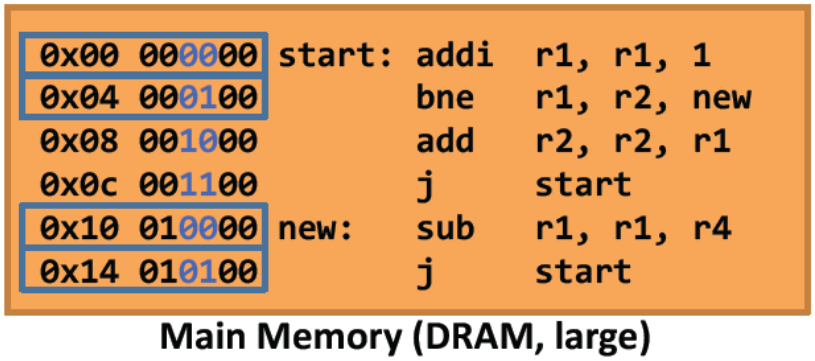


Ví dụ bộ đệm ánh xạ trực tiếp

• Ưu điểm: dễ dàng trong việc tìm kiếm
  – Sử dụng địa chỉ để tìm một vị trí (modulo addressing)
  – So sánh với tag

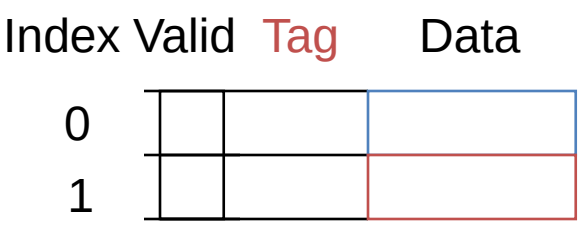
• Nhược điểm: kém linh hoạt (1 chu kỳ tìm kiếm 1 khối)
– Nếu 2 địa chỉ ánh xạ tới cùng một dòng khi đó chỉ có thể lưu trữ một khối dữ liệu
– **Xảy ra xung đột**
Nếu 2 địa chỉ khối trong bộ nhớ trong có cùng địa chỉ modulo, cả 2 không được lưu trữ cùng nhau trong cache, kể cả khi cache còn trống. (Inflexible → Conflicts)





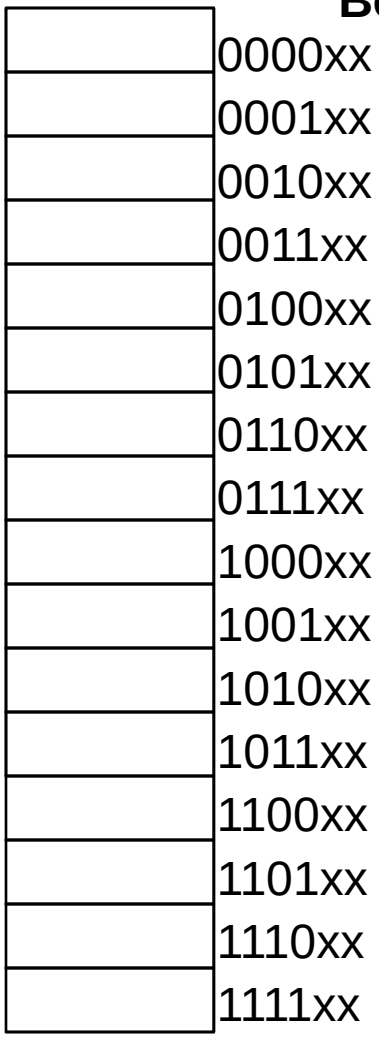
Bộ đệm ánh xạ trực tiếp đơn giản

Bộ đệm: 2 khối nhớ, mỗi khối có 2 từ



Q1: Có trong bộ đệm không?

Bộ nhớ chính: 16 từ



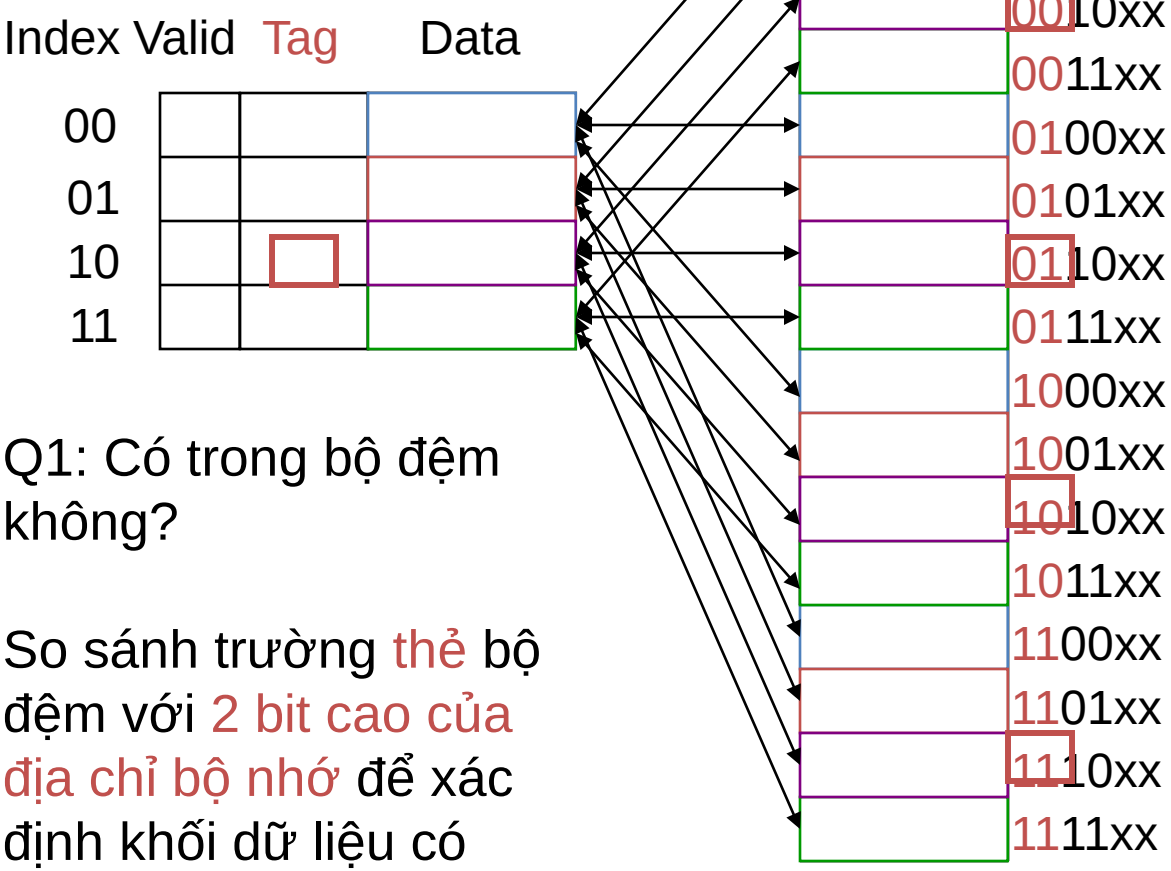
Các từ: 2 bít thấp dùng để xác định các byte trong từ (32b words)

Q2: Vị trí các từ trong bộ đệm?

(block address) modulo (# of blocks in the cache)

Bộ đệm ánh xạ trực tiếp đơn giản

Bộ đệm: 4 khối nhớ kích thước 1 từ



Các 1 từ: 2 bít thấp dùng để xác định các byte trong từ (32b words)

Q1: Có trong bộ đệm không?

So sánh trường thẻ bộ đệm với 2 bit cao của địa chỉ bộ nhớ để xác định khối dữ liệu có trong bộ đệm không?

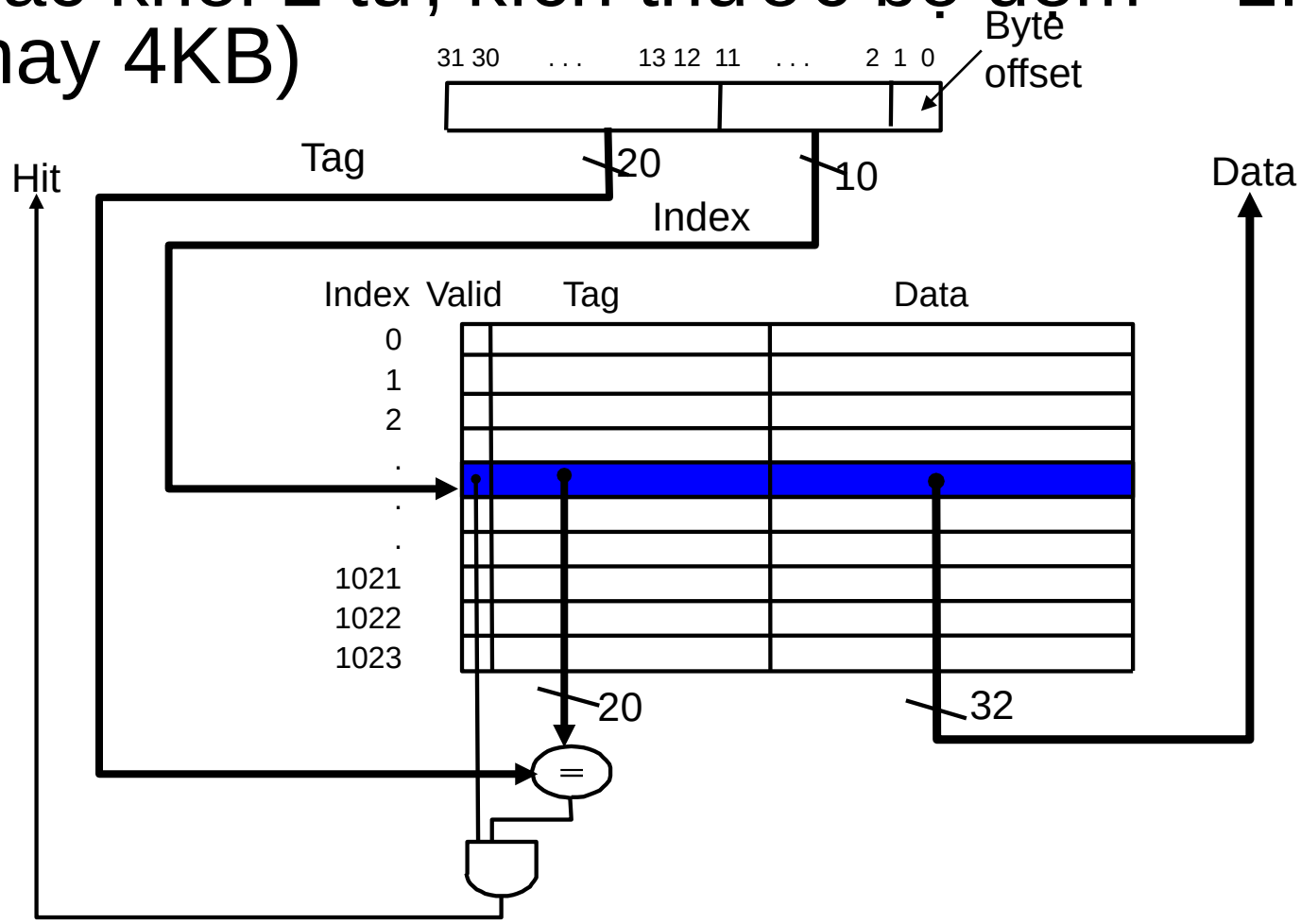
Q2: Vị trí các từ trong bộ đệm?

Dùng 2 bít thấp tiếp theo của địa chỉ –chỉ số – để xác định khối bộ đệm nào (i.e., chia lấy dư cho số khối trong bộ đệm)

(block address) modulo (# of blocks in the cache)

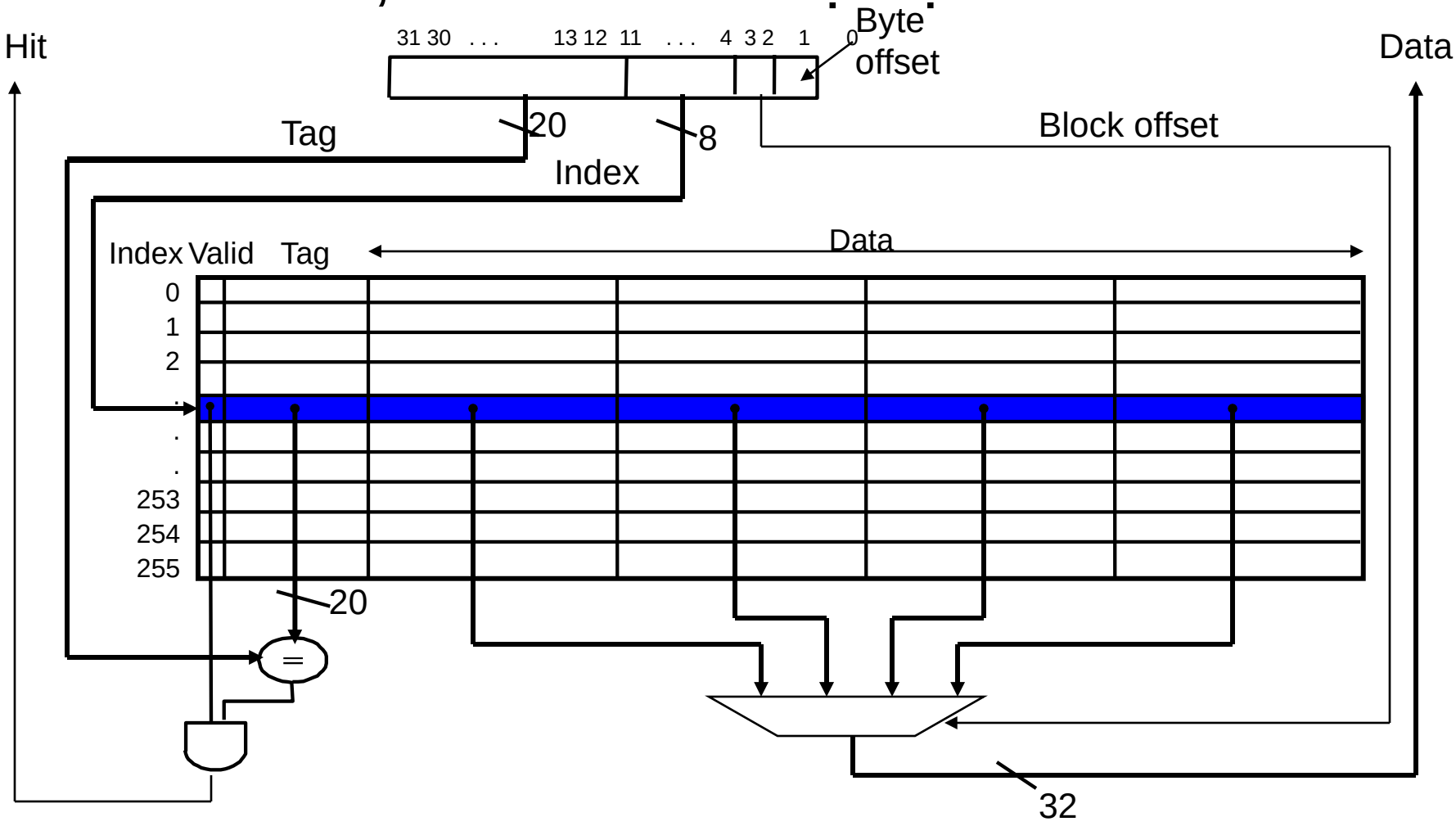
Bộ đệm ánh xạ trực tiếp MIPS

- Các khối 1 từ, kích thước bộ đệm = 1K từ (hay 4KB)



Bộ đệm ánh xạ trực tiếp khối nhiều từ

- Khối 4 từ, Kích thước bộ đệm = 1K words



Bộ nhớ đệm liên kết kiểu tập hợp: Set-associative caches

linh hoạt hơn

Có thể thỏa hiệp?

• Có thể kết hợp việc tìm kiếm dễ dàng trong bộ đệm ánh xạ trực tiếp (direct-mapped) và sự linh hoạt trong bộ đệm kết hợp toàn khối (fully-associative caches)?

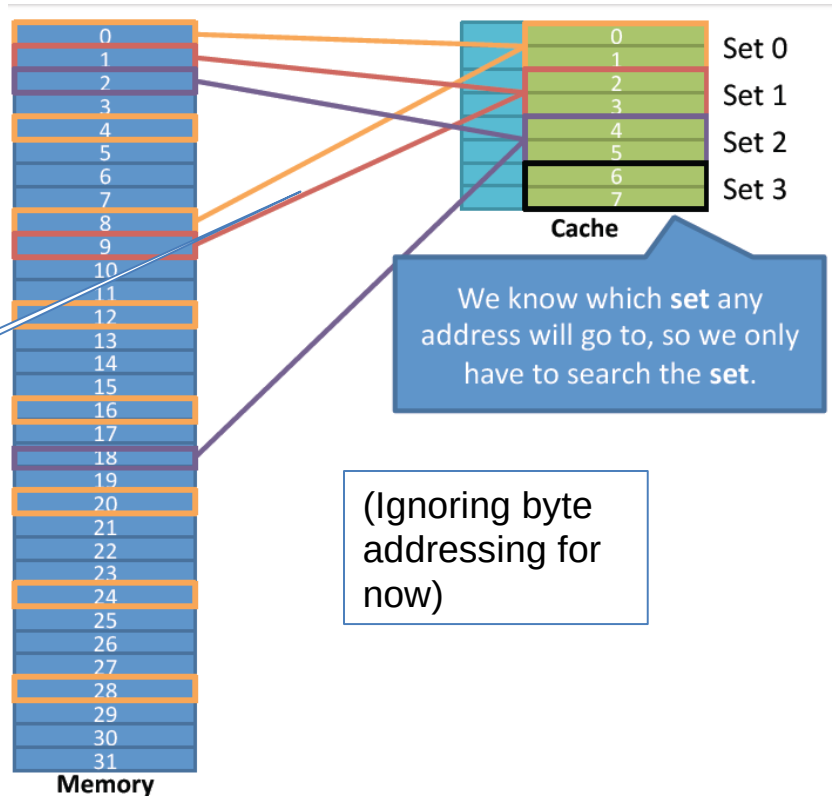
• Có**: Bộ đệm kết hợp theo tập hợp (set-associative)**
  – Mỗi khối nằm trong một tập (**1 set gồm nhiều khối** - có cùng địa chỉ modulo)
  – Mỗi tập có nhiều khối (tìm kiếm kết hợp)

• Dễ dàng hơn trong việc tìm kếm:
  – Xác định được khối dựa trên địa chỉ
  – Chỉ cần kiểm tra một số khối cache trong tập.
• Linh hoạt:
  – Có thể đặt khối ở bất kỳ đâu trong tập
  – Giảm số lần trượt bộ đệm.

Bộ đệm kiểu tập hợp (Set-associative cache)

- Mỗi địa chỉ ánh xạ tới một khối
- Sử dụng địa chỉ modulo (K = i mod s)
  - K : vị trí khối đặt trong cache
  - i : số thứ tự khối trong bộ nhớ trong
  - s : số lượng tập hợp trong cache
  - – 0 → (0 mod 4) = anywhere in Set 0
  - – 1 → (1 mod 4) = anywhere in Set 1
  - – 2 → (2 mod 4) = anywhere in Set 2
  - – 8 → (8 mod 4) = anywhere in Set 0
  - – 9 → (9 mod 4) = anywhere in Set 1
  - – 18 → (18 mod 4) = anywhere in Set 2
- Xem xét 2 bits cuối:
  - – 0 → 000**00** → Set 0
  - – 1 → 000**01** → Set 1
  - – 2 → 000**10** → Set 2
  - – 8 → 001**00** → Set 0
  - – 9 → 001**01** → Set 1
  - – 18 → 010**10** → Set 2

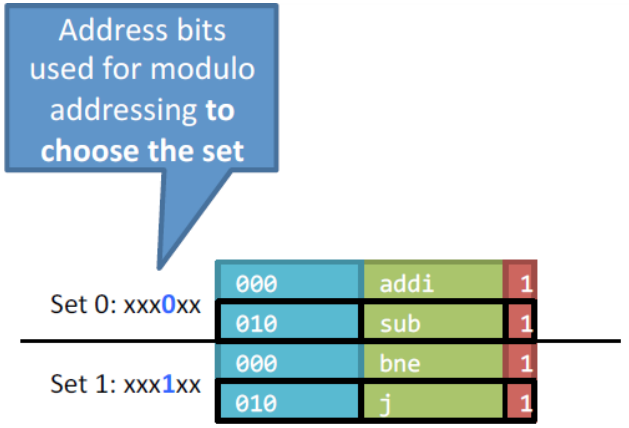
Tag bits



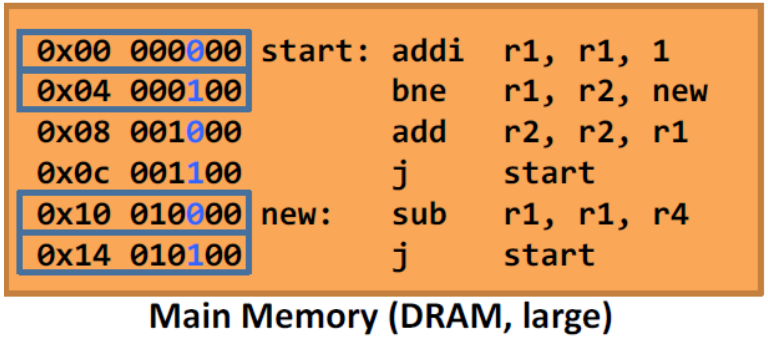
(Ignoring byte addressing for now)

Ví dụ: Set-associative

- **Ưu điểm: Dễ dàng trong việc tìm kiếm**
  - Sử dụng địa chỉ tìm kiếm các tập
  - So sánh tất cả các tag trong tập
- **Nhược điểm:**
  - Ít linh hoạt hơn kiểu fully-associative (can only go anywhere in the set)
  - Phức tạp hợp (kiểm tra nhiều tags hơn)

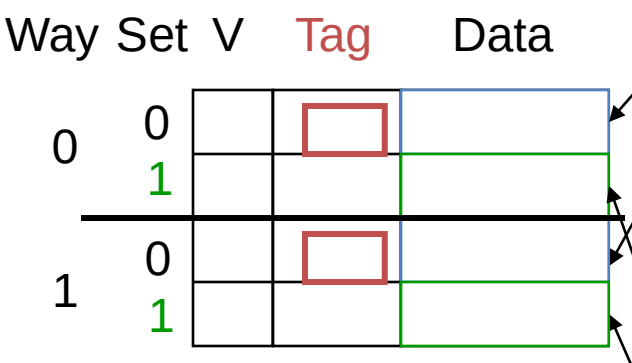


Nếu hai địa chỉ có cùng một địa chỉ modulo, có thể tìm chúng trong một tập. (More flexible → Fewer conflicts)



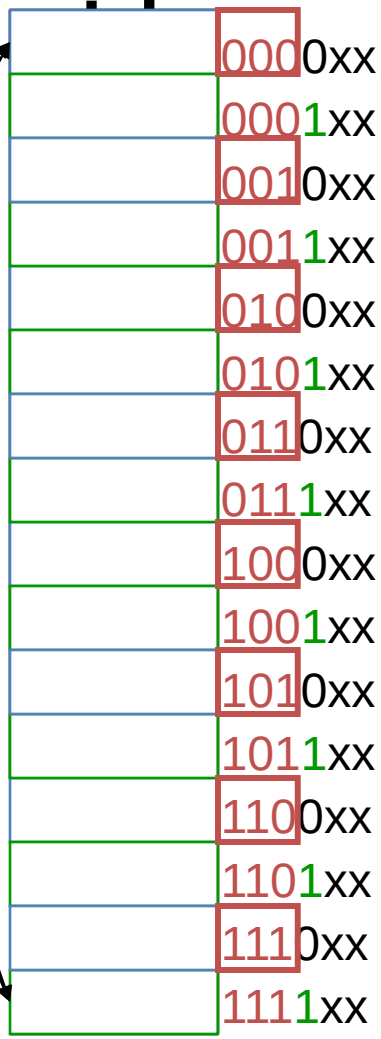
Bộ đệm kết hợp n đường

Bộ đệm: 4 khối, 2 tập



Bộ nhớ chính 16 khối 1 từ

Khối 1 từ, 2 bit thấp cuối dùng để xác định byte trong từ (từ 32b)



Q1: Có trong bộ đệm không?

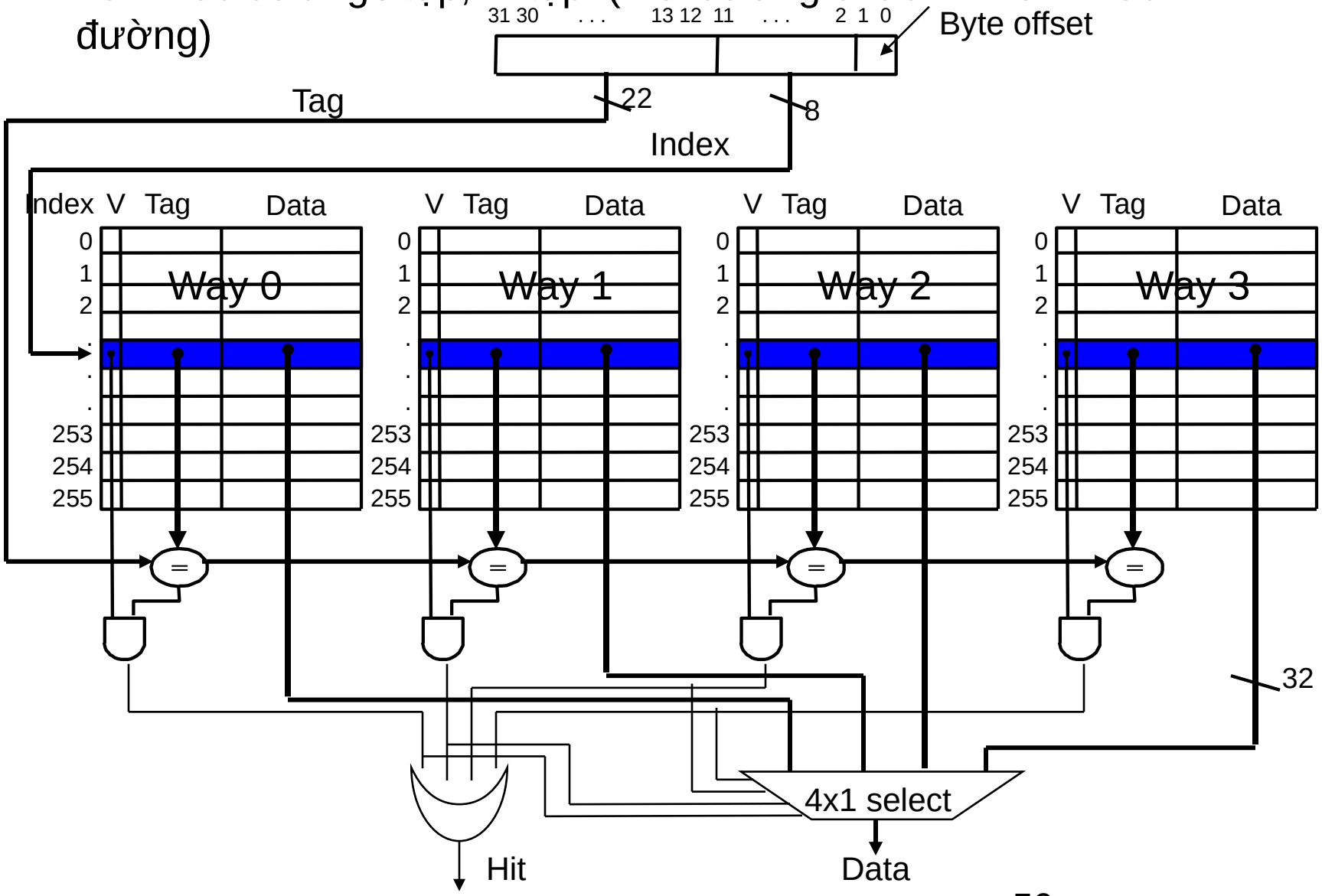
So sánh *tất cả* các thẻ trong tập với 3 bít địa chỉ cao

Q2: Vị trí từ trong bộ đệm?

Sử dụng bít thấp tiếp theo để xác định tập (i.e., chia lấy phần dư cho số tập trong bộ đệm)

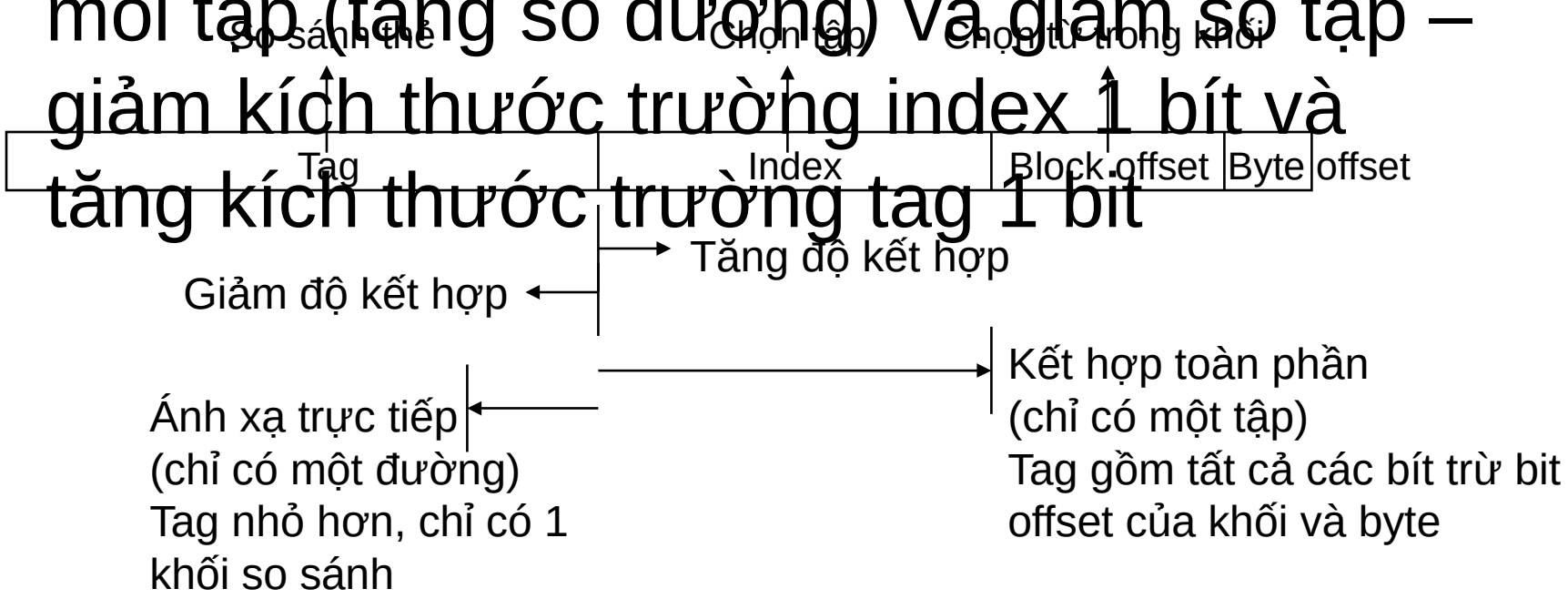
Bộ đệm kết hợp 4 tập

- 28 = 256 đường / tập, 4 tập (mỗi đường chứa 1 khối = 256 đường)



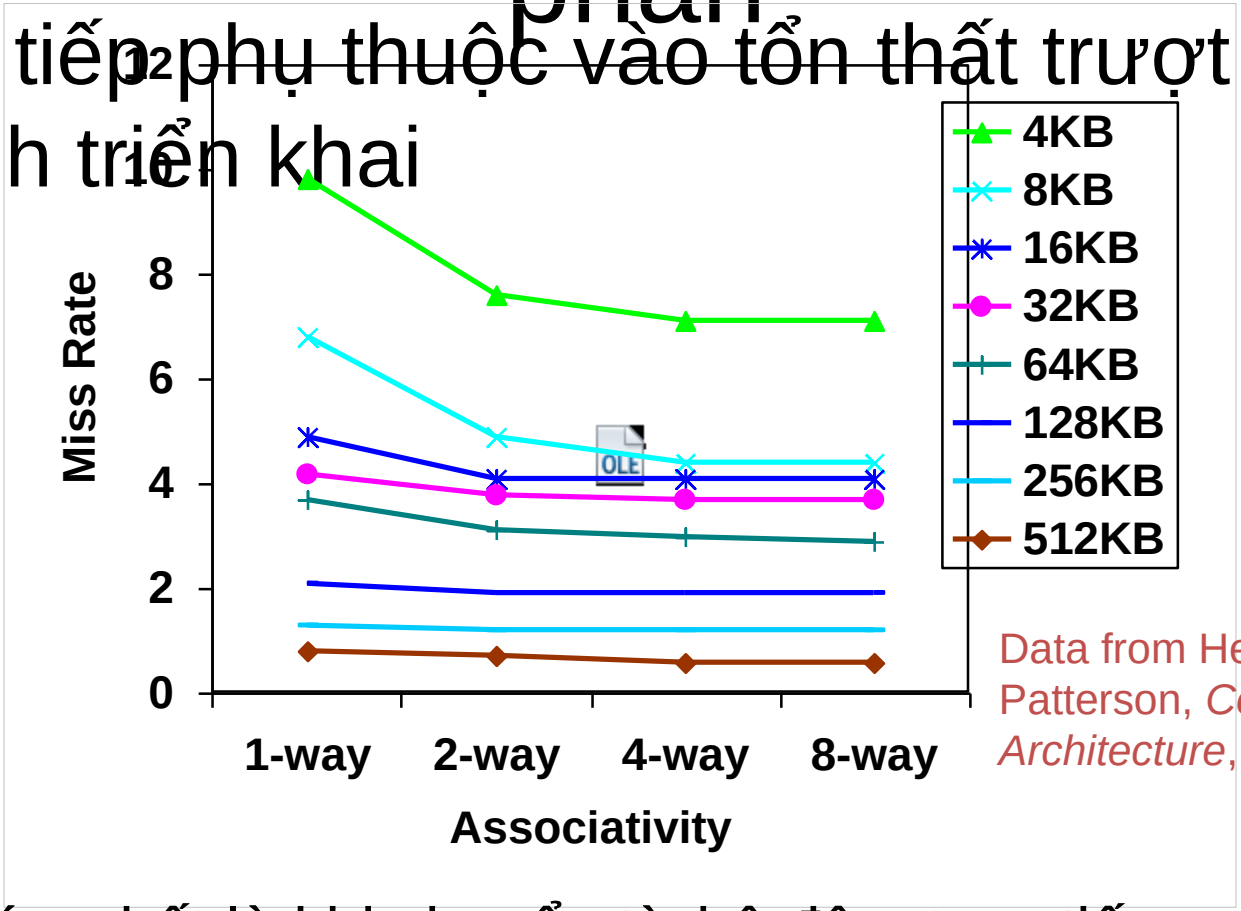
Bố trí bộ đệm kết hợp toàn phần / liên kết toàn phần

- Với kích thước bộ đệm cố định, tăng độ kết hợp theo hệ số 2 sẽ tăng số khối trong mỗi tập (tăng số đường) và giảm số tập – giảm kích thước trường index 1 bít và tăng kích thước trường tag 1 bit



Lợi ích của bộ đệm kết hợp toàn phần

- Lựa trọn giữa bộ đệm kết hợp và bộ đệm trực tiếp phụ thuộc vào tổn thất trượt và giá thành triển khai



Lợi ích lớn nhất là khi chuyển từ bộ đệm trực tiếp sang kết hợp 2 đường (tỉ lệ trượt giảm 20%+)

Kích thước các trường trong bộ đệm

- Số bit trong bộ đệm = số bit cho dữ liệu và bit cho các trường thẻ
  - Địa chỉ byte 32 bit
  - Bộ đệm ánh xạ trực tiếp 2n khối, *n* bits cho trường index
  - Kích thước khối là 2m từ (2m+2 bytes), *m* bits cho trường block offset xác định vị trí từ trong khối; 2 bits cho trường byte offset xác định vị trí byte trong từ
- Kích thước trường tag sẽ là?
- Tổng số bít trong bộ đệm ánh xạ trực tiếp

Bộ nhớ đệm ngày nay

- Intel Ivy Bridge (2012, 4-core x86)
  - L1 Data: 32kB 8-way set-associative 64-byte line
  - L1 Instruction: 32kB 8-way set-associative 64-byte line
  - L2 I&D: 256kB 8-way set-associative 64-byte line
  - L3 I&D: **8MB 16-way** set-associative 64-byte line

- Qualcomm Krait (2012, 4-core ARM)
  - L0 Data: **4kB Direct-mapped** 64-byte line
  - L0 Instruction: **4kB Direct-mapped** 64-byte line
  - L1Data: 16kB 4-way set-associative 64-byte line
  - L1 Instruction: 16kB 4-way set-associative 64-byte line
  - L2 I&D: **1MB 8-way** set-associative 64-byte line

Same block size across the whole hierarchy

"Filter" cache. Designed to save energy for small pieces of code

Higher associativity for larger caches.

Dung lượng bộ nhớ đệm

điều gì xảy ra nếu bộ nhớ đệm đầy?

Dung lượng caches

- Điều gì xảy ra khi cache đầy?

- **Chính xách thay thế**
  - Cần chọn một số dữ liệu trong caches để loại bỏ (**evict**)
  - Chọn những dữ liệu không sử dụng từ lâu nhất

- Dẫn đến:
  - **Direct-mapped: chỉ loại đi một khối** (bởi vì một khối tương ứng với một vị trí)
  - **Set/Fully-associative:**
    - Chọn ngẫu nhiên một khối để loại bỏ
    - Chọn khối nào không sử dụng lâu nhất (**least recently used** - LRU) để loại bỏ.

Chính sách thay thế

- Bốn chiến thuật chủ yếu chọn khối thay thế trong cache

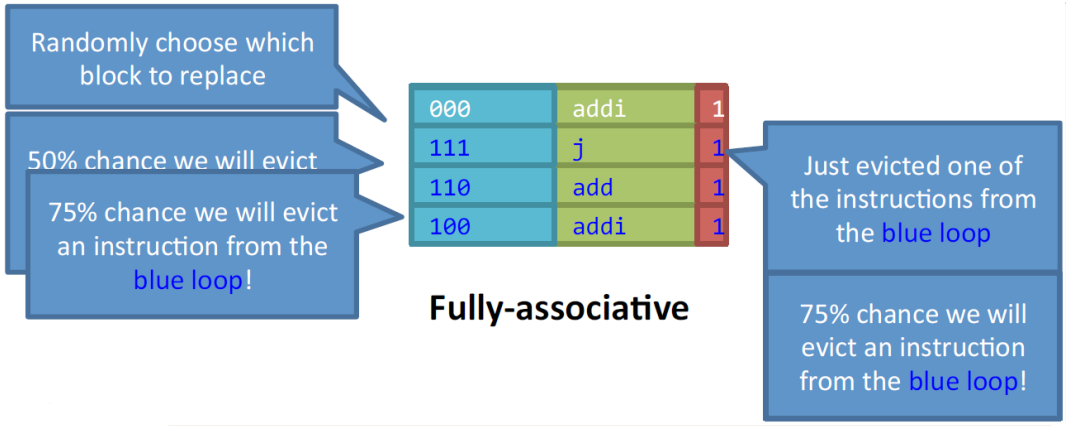
- Thay thế ngẫu nhiên
  – Để phân bố đồng đều việc thay thế, các khối cần thay thế trong cache được chọn ngẫu nhiên

- Khối xưa nhất (**Least Recently Used): hiệu quả trong các vị trí tạm thời**
  – Thay thế khối không được dùng từ lâu nhất

- Vào trước ra trước (FIFO)
  – Khối được đưa vào cache đầu tiên, nếu bị thay thế sẽ là khối bị thay thế trước nhất.

- Khối có tần suất sử dụng ít nhất (LFU – Least Frequently Used): Khối trong cache được tham chiếu ít nhất

Thay thế ngẫu nhiên



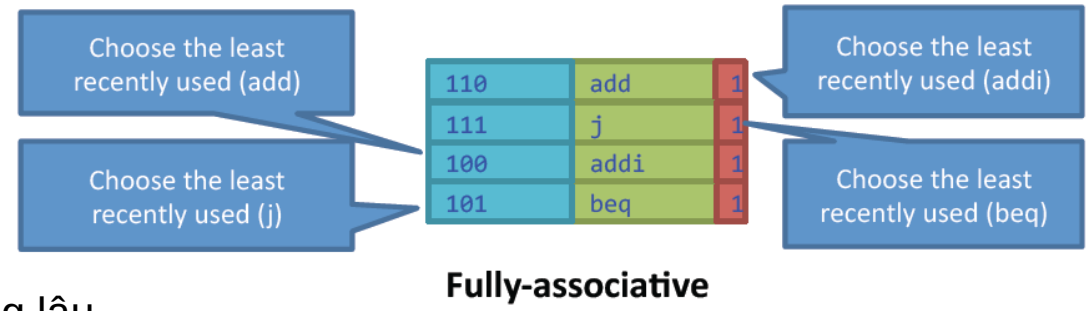
- Chọn ngẫu nhiên một khối để loại bỏ

- Ví dụ:
  - Black loop và blue loop
  - Khi chuyển đổi giữa hai vòng lặp dữ liệu luôn được sử dụng lại

- **Problem: Dữ liệu loại bỏ là dữ liệu cần sử dụng lại**

```
0x00 00000 start:  addi  r1, r1, 1
0x04 00100         beq   r1, r2, next
0x08 01000         add   r3, r3, r1
0x0c 01100         j     start
0x10 10000 next:   addi  r1, r1, 1
0x14 10100         beq   r1, r5, start
0x18 11000         add   r4, r4, r1
0x1c 11100         j next
...
```

Thay thế khối xưa nhất
Least recently used (LRU) replacement



- Loại bỏ khối không được sử dụng lâu nhất

- Ví dụ :
  - Black loop và blue loop
  - Khi chuyển đổi giữa các vòng lặp, cần tất cả các lệnh trong vòng lặp mầu xanh ở lại trong cache

```
0x00 00000 start:  addi  r1, r1, 1
0x04 00100         beq   r1, r2, next
0x08 01000         add   r3, r3, r1
0x0c 01100         j     start
0x10 10000 next:   addi  r1, r1, 1
0x14 10100         beq   r1, r5, start
0x18 11000         add   r4, r4, r1
0x1c 11100         j next
…
```

Ghi vào bộ nhớ đệm

Chiến thuật ghi?

Ghi vào bộ nhớ đệm

- Khi đọc lệnh hoặc dữ liệu: đặt vào cache
- Ghi như thế nào?

- **Write-through (ghi đồng thời)**
  - Thông tin được ghi đồng thời vào khối của cache và khối của bộ nhớ trong
  - Đơn giản, nhưng chậm (phải đợi ghi vào DRAM)

- **Write-back (ghi lại)**
  - Chỉ ghi dữ liệu vào cache
  - Nếu khối không có trong cache, cần nạp lại vào cache
  - Cần đánh dấu lại nếu dữ liệu trong cache được cập nhật
  - Khi một khối bị thay thế, khối này sẽ được ghi lại vào bộ nhớ trong.

Write-through

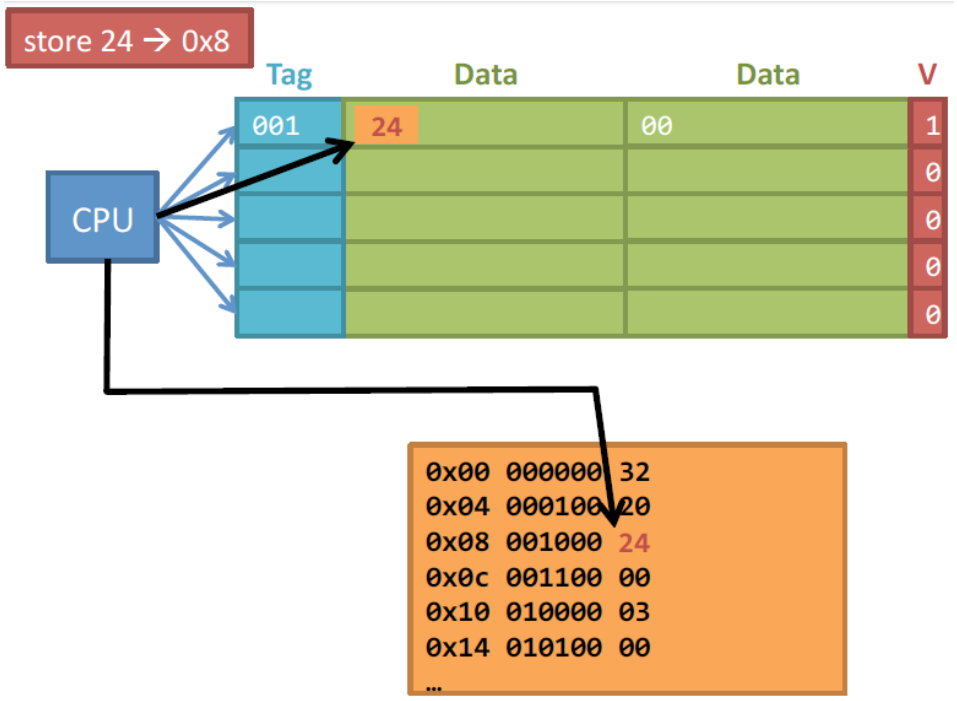
- Luôn ghi vào DRAM
- Nếu dữ liệu ở trong cache, khi đó ghi đồng thời vào cache.

Q: Ghi chậm hơn vì sao?

1. Phụ thuộc vào vị trí của dữ liệu trong cache.
2. Châm do ghi vào DRAM
3. Chậm do ghi vào DRAM và cache

A: Chậm do ghi vào DRAM

Ghi vào DRAM ở mọi thời điểm.



Write-through with allocate-on-write

• Luôn ghi vào DRAM

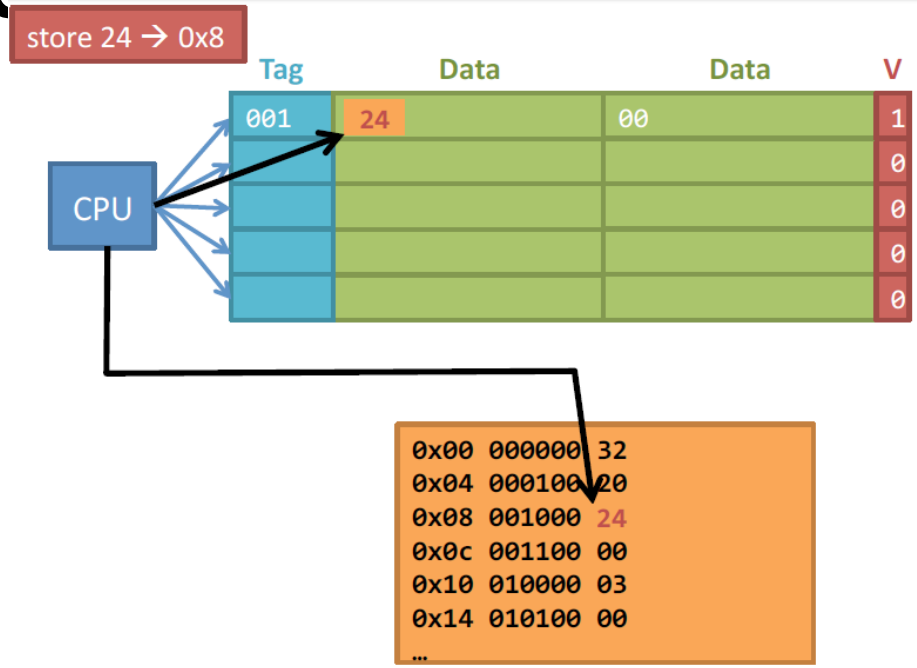
• Nếu dữ liệu trong cache, ghi đồng thời vào cache

• **Allocate-on-write**
– Nếu khối dữ liệu không có trong cache, tải khối về cache

Q: Tốt hơn tại sao?

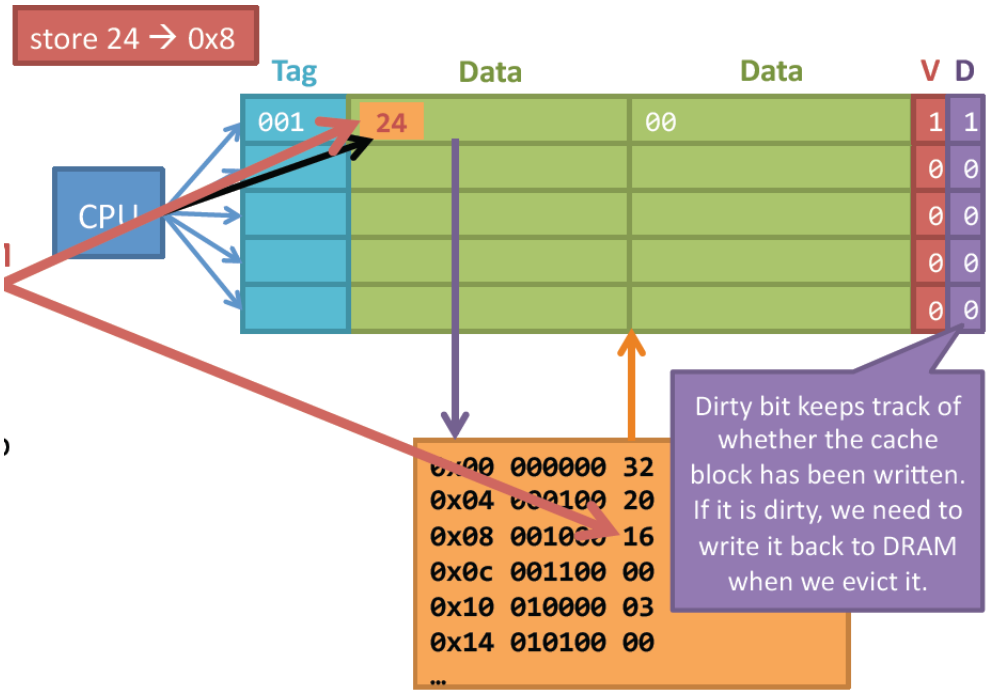
1. It isn't
2. Faster to read and write than just write
3. Subsequent reads will hit in the cache

A: Subsequent reads will hit in the cache
Thường hay truy cập dữ liệu và ghi lại, chỉ ghi một phần của khối như một từ hoặc một byte, do vậy tải vào cache sẽ nhanh hơn.



Write-back

- Luôn ghi vào cache
- Nếu khối dữ liệu không có trong cache, **tải vào cache**
- **Note: cache và DRAM là không như nhau!**
- Khi chúng ta loại bỏ một dòng cần phải biết khác với dữ liệu ghi từ DRAM như thế nào?



Ghi vào bộ nhớ đệm

• **Write-through đơn giản, nhưng chậm**
– Phải đợi ghi đồng thời với DRAM trong mọi lần ghi

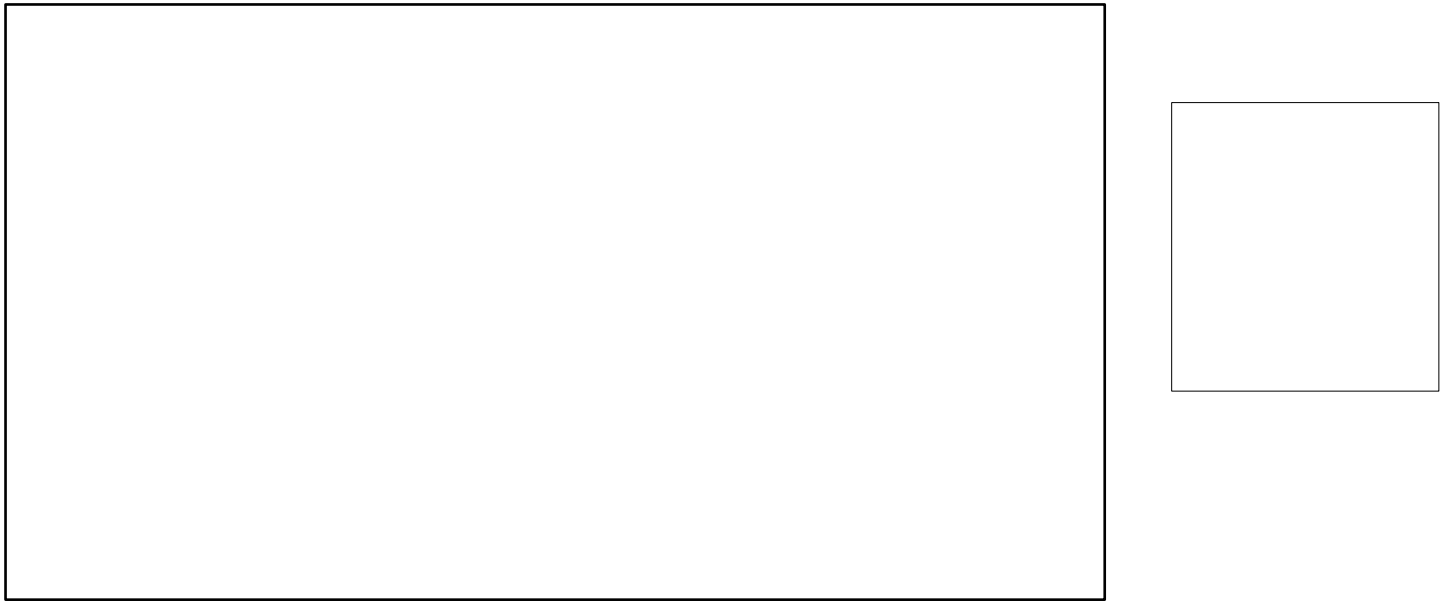
• Write-**back phức tạp hơn, nhưng nhanh**
– Phải đánh dấu dữ liệu ‘bẩn’ (dirty data) trong cache và ghi lại vào bộ nhớ trong nếu nó bị loại bỏ
– Ghi nhanh hơn

• **Allocate-on-write tải dữ liệu vào cache khi ghi**
– no allocate : chỉ tải dữ liệu vào cache khi đọc.

• Ghi vào cache ngày nay là kiểu write-back
– Intel’s new Xeon Phi có mức L2 write-through cache

Hiệu năng bộ nhớ đệm

Tỉ lệ trượt vs Kích thước khối vs Kích thước bộ đệm



- Tỉ lệ trượt tăng khi kích thước khối trở nên đáng kể so với kích thước bộ đệm vì với cũng kích thước bộ đệm số khối có thể lưu giữ giảm (tăng trượt do dung lượng)
- Tăng kích thước khối làm tổn thất trượt tăng

Xử lý trúng bộ đệm

- Đọc trúng (I$ và D$)
  - Đó là điều ta cần!

- Ghi trúng (chỉ với D$)
  - yêu cầu bộ đệm và bộ nhớ phải thống nhất (allocate)
    - luôn ghi dữ liệu vào cả khối bộ đệm và vào bộ nhớ ở mức kế tiếp (ghi xuyên - write-through)
    - ghi với tốc độ của bộ nhớ ở mức kế tiếp – chậm hơn! – sử dụng bộ đệm ghi (write buffer) và chỉ dừng khi bộ đệm ghi đầy

Xử lý trượt bộ đệm (Khối kích thước 1 từ)

- Đọc trượt (I$ và D$): mất thời gian *read_miss_penalty*
  - dừng đường ống, nạp khối từ bộ nhớ ở mức kế tiếp, đưa vào bộ đệm và gửi từ được yêu cầu tới bộ xử lý, tiếp tục đường ống
- Ghi trượt (D$) mất thời gian *write_miss_penalty và write_buffer_stalls*
  - Cấp phát và ghi – Đầu tiên đọc khối từ bộ nhớ và ghi từ vào khối

or

  - Không cấp phát và ghi– bỏ qua việc ghi vào bộ đệm; ghi từ vào bộ đệm ghi (tức là sẽ ghi

Đo hiệu năng bộ đệm

- Giả sử thời gian truy cập bộ nhớ khi trúng bộ đệm được bao gồm trong 1 chu kỳ thực hiện thông thường của CPU thì:

$$T_{cpu} = IC \times (CPI_{ideal} + MemStallC) \times T_c$$

CPIstall

- Số chu kỳ MemStallC là tổn thất trượt là tổng của read-stalls và write-stalls

  Read-stall cycles = reads/program × read miss rate × read miss penalty

  Write-stall cycles = (writes/program × write miss rate × write miss penalty) + write buffer stalls

- Với bộ đệm ghi xuyên, ta có công thức đơn giản

  Memory-stall cycles = accesses/program × miss rate × miss penalty

Ảnh hưởng của hiệu năng bộ đệm

- Tổn thất tương đối của bộ đệm sẽ tăng khi hiệu năng bộ xử lý tăng (tăng tốc độ đồng hồ và/hoặc giảm CPI)
  - Tốc độ bộ nhớ không được cải thiện nhanh như tốc độ bộ xử lý. Tổn thất trượt dùng để tính CPIstall được đo theo số chu kỳ bộ xử lý cần thiết để xử lý trượt
  - CPIideal càng thấp thì ảnh hưởng của dừng do trượt càng lớn
- Bộ xử lý với CPIideal = 2, tổn thất trượt là 100, 36% là lệnh load/store, tỉ lệ trượt bộ

Ảnh hưởng của hiệu năng bộ đệm

- Tổn thất tương đối của bộ đệm sẽ tăng khi hiệu năng bộ xử lý tăng (tăng tốc độ đồng hồ và/hoặc giảm CPI)
  - Tốc độ bộ nhớ không được cải thiện nhanh như tốc độ bộ xử lý. Tổn thất trượt dùng để tính CPIstall được đo theo số chu kỳ bộ xử lý cần thiết để xử lý trượt
  - CPIideal càng thấp thì ảnh hưởng của dừng do trượt càng lớn
- Bộ xử lý với CPIideal = 2, tổn thất trượt là 100, 36% là lệnh load/store, tỉ lệ trượt bộ nhớ I$ là 2% và bộ nhớ D$ là 4%

Nguyên nhân trượt bộ đệm

- Không tránh được:
  - Lần đầu truy cập khối
  - Giải pháp: tăng kích thước khối (làm tăng tổn thất trượt, khối rất lớn làm tăng tỉ lệ trượt)
- Dung lượng:
  - Bộ đệm không thể chứa toàn bộ các khối truy cập bởi chương trình
  - Giải pháp: tăng kích thước bộ đệm (có thể làm tăng thời gian truy cập)
- Xung đột:

Tỷ số trượt bộ đệm

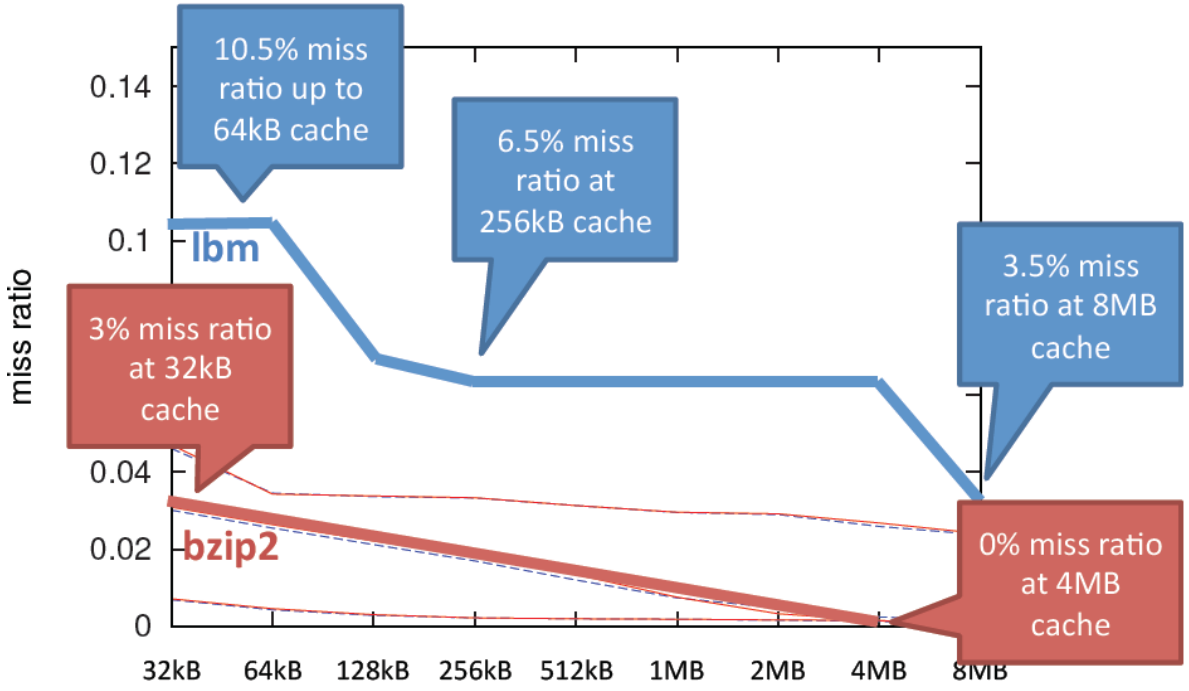
Miss ratio = % of cache misses = (# cache misses / # memory accesses)
Tỷ lệ trượt (%) = (số lần trượt/số lần truy nhập)

Q: Hiệu năng thay đổi thế nào khi tỷ số trượt bộ đệm giảm từ 10.5% đến 3.5%?

1. Slower
2. Stays the same
3. Faster

A: Faster

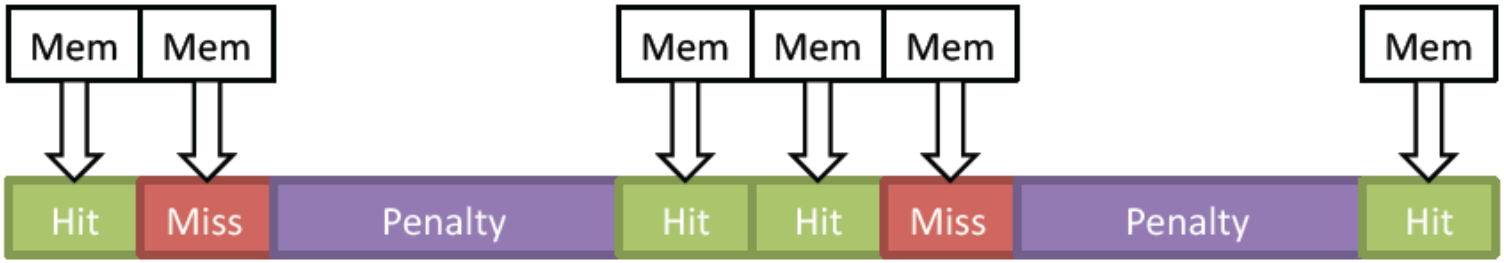
Các ứng dụng có thể chạy nhanh hơn, nhưng không thể biết là nhanh ơn bao nhiêu. Tỷ lệ Trúng cache lớn hơn rất nhiều so với trượt.



Average Memory Access Time (AMAT -Thời gian truy cập bộ nhớ trung bình)

• Số chu kỳ trung bình cho một truy cập bộ nhớ
= (hit time) + (miss %)*(miss time + miss penalty)

Miss Penalty là thời gian sau khi tìm kiếm trong cache.



Hit time = 1 Hit (1cycle)
Miss time = 1 Miss (1 cycle)
Miss penalty = 3 Penalty

% miss = 2/6 = 33%
AMAT = (1) + (33%)*(1 + 3) = 2.3 cycles per access

Ví dụ: AMAT

Machine 1

– 100 truy nhập DRAM

– 1 chu kỳ truy nhập cache (hit or miss)

• Tính **AMAT cho lbm?**

– cache có dung lượng 256kB ?

• 6% miss ratio

• (1) + (6%)*(1+100) = 7.06 cycles per memory access

– cache có dung lượng 8MB ?

• 3% miss ratio

• (1) + (3%)*(1+100) = 4.03 cycles per memory access

Machine 2

– 100 cycles to DRAM

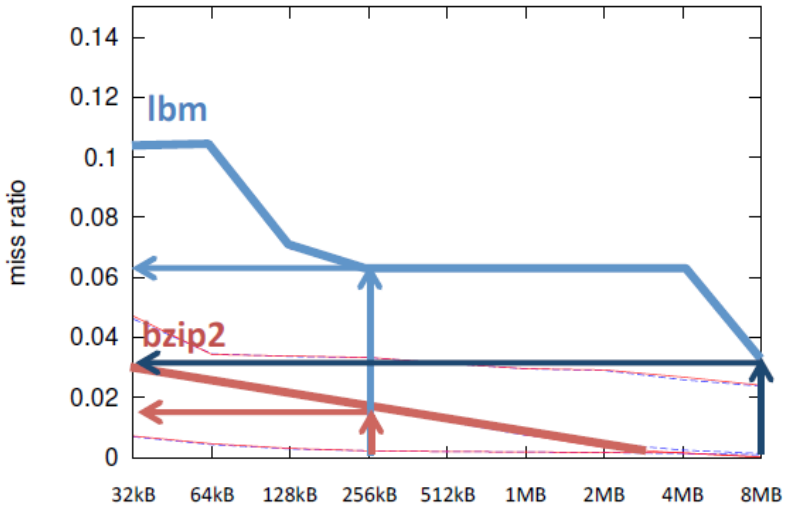
– 2 cycle cache access time (hit or miss)

• **Tính AMAT cho bzip2?**

– 256kB cache?

• 1.5% miss ratio

• (2) + (1.5%)*(2+100) = 3.53



Q: Tại sao bzip2's AMAT với 256kB cache gần giống với lbm's với 8MB?

1. Slower cache
2. Different miss ratios
3. Different applications

A: Slower cache

Cache mất 2 cycles cho Machine 2 và chỉ một chu kỳ cho Machine 1. Diều này làm ảnh hưởng đến AMAT.Ứng dụng khác nhau sẽ thấy 8MB cache có 3% miss ratio đối với lbm trong khi bzip2 có 1.5% miss ratio với cache 256kB

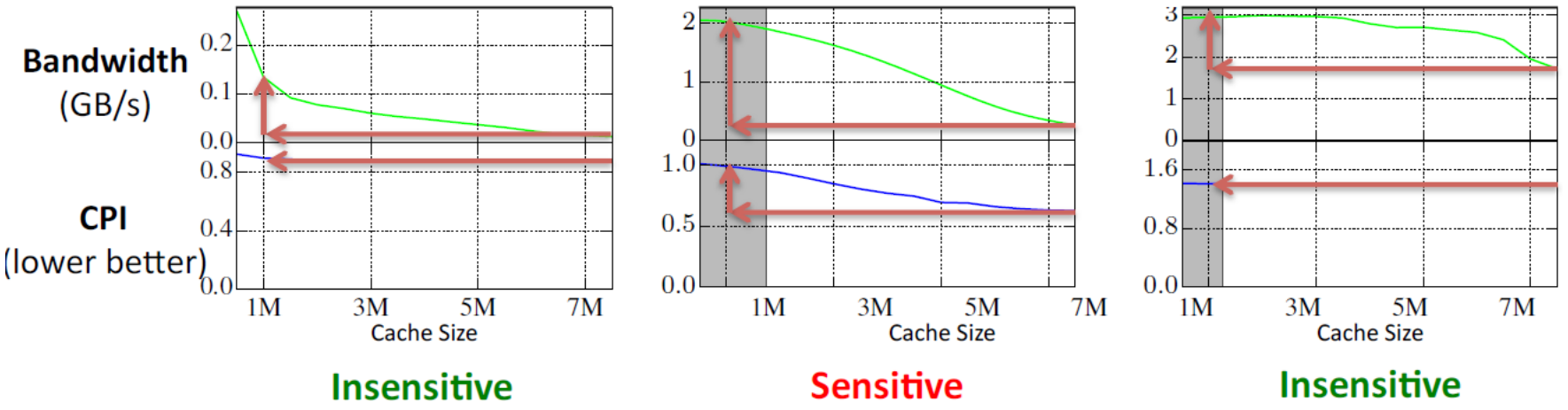
Performance impacts of memory access time

- Ảnh thưởng của cace đến hiệu năng như thế nào(CPI)?
  - 100 chu kỳ trừng phạt trượt bộ đệm
  - 3% tỷ lệ trượt
  - 1.33 lần truy nhập bộ nhớ trên một lệnh (1 for instruction 33% for data)
  - CPI = 1.0 khi thực thi thông thường (lý tưởng)

- Hiệu năng thay đổi thế nào?
  - CPI = CPIExecution + Memory stall cycles per instruction = 1.0 + 1.33*0.03*100 = 4.99
  - Phân cấp bộ nhớ này làm bộ xử lý chạy chậm hơn 5 lần!

- Caches là rất quan trọng! Caches là quan trọng nhất để tăng hiệu năng.

How much does cache matter?

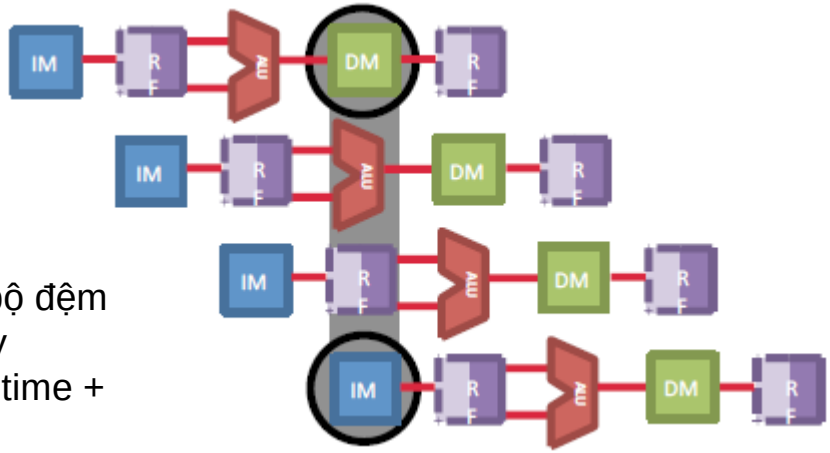


Insensitive **Sensitive** **Insensitive**

- When an application gets less cache it makes more accesses to memory. (Higher bandwidth)
- Some applications are more sensitive to trading off bandwidth and cache than others.

Phân tách lệnh và dữ liệu bộ nhớ đệm

- Cần nạp lệnh(**IF**) và dữ liệu (**MEM**) cùng một thời điểm

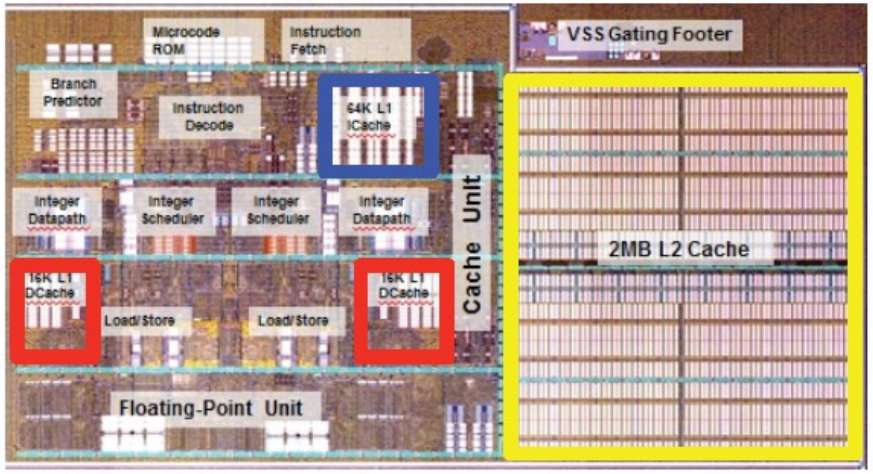


- Cần 2 loại cache:
  - **Instruction** cache (just instructions)
  - **Data** cache (just data)

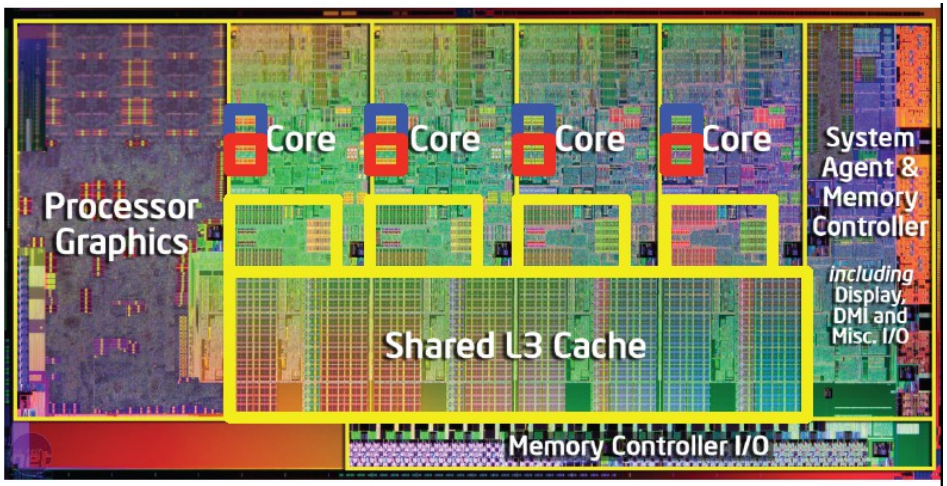
- AMAT khi phân chia cache:
  - AMAT: (% lệnh truy cập)*(hit time + (tỷ lệ trượt bộ đệm lệnh)*(miss time + miss penalty)) + (% dữ liệu truy cập)*(hit time + (tỷ lệ trượt bộ đệm dữ liệu)*(miss time + miss penalty))

- Ví dụ:
  - Thời gian truy nhập cache là 1 cycle (hit or miss) và thời gian truy nhập bộ nhớ 100 cycle
  - I-cache: 1% miss ratio, D-cache: 5% miss ratio
  - 33% là lệnh loads/stores → 25% truy cập dữ liệu / 75% truy cập lệnh
  - (75%)*(1 + (1%)*(1+100)) + (25%)*(1 + (5%)*(1+100)) = 1.5 + 1.5 = 3.0

AMD/Intel caches

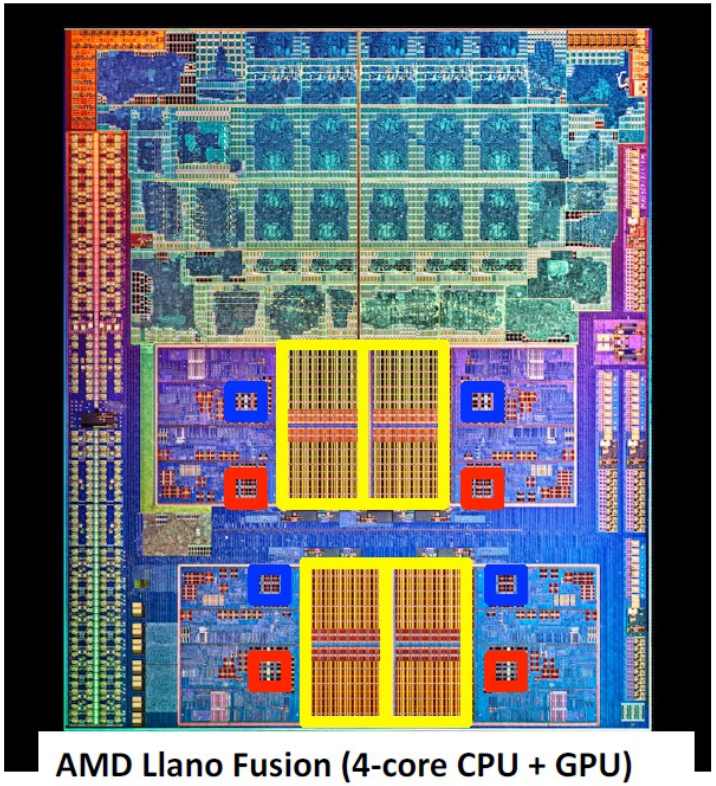


AMD Bulldozer dual-core: Shares one I-cache and two D-caches

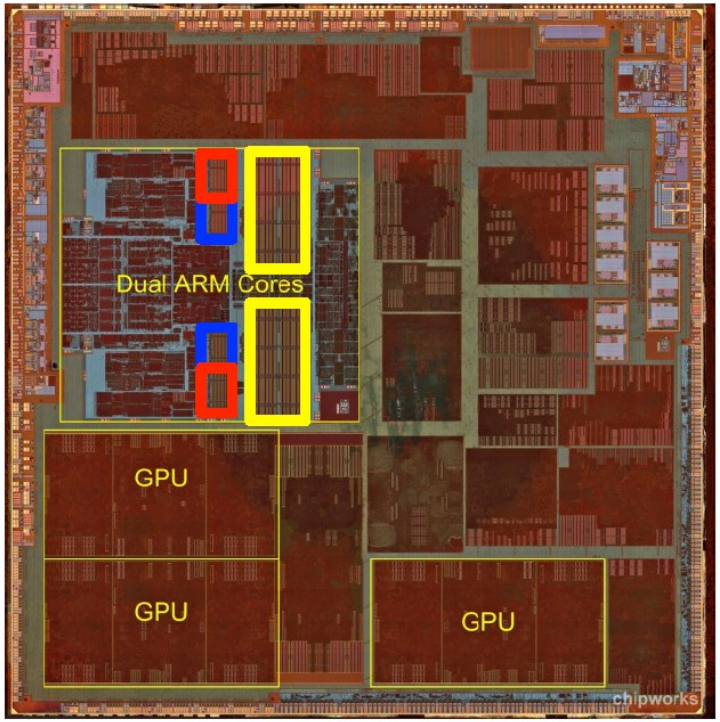


Intel SandyBridge 4-core

Heterogeneous processor caches



AMD Llano Fusion (4-core CPU + GPU)



Apple A6 (2-core CPU + GPU)

Memory hierarchy

- We want big and fast
  - Build a hierarchy where we keep the most important data in fast memory
  - Other data goes in slow memory
  - If we move the data correctly we provide the illusion of fast and big

- Registers 3 accesses/cycle 32--64

- Cache 1--10 cycles 8kB--256kB

- Cache 40 cycles 4--20MB
- DRAM 200 cycles 4--16GB
- Flash 1000+ cycles 64--512GB
- Hard Disk 1M+ cycles 2 --4TB

Summary: how to make the memory hierarchy work

- 3 different cache types
  - **Fully-associative**: Have to **search all blocks**, but very flexible
  - **Direct-mapped**: Only **one place for each block**, no flexibility
  - **Set-associative**: Only have to search **one set for each block**, flexible

- We can adjust the block (line) size to reduce the overhead of **tags**

- We figure out where data goes in a cache by looking at the **address**
  - Last **2 bits** are the **byte** in the word
  - Next **N bits** are the **word in the cache block**
  - **Remaining bits** are for the **tag**

- We have different write policies
  - **Write-through**: **slow**, simple
  - **Write-back**: **fast** (keeps the data just in the cache), more complex
- Performance effects are due to the average memory access time