
第9回

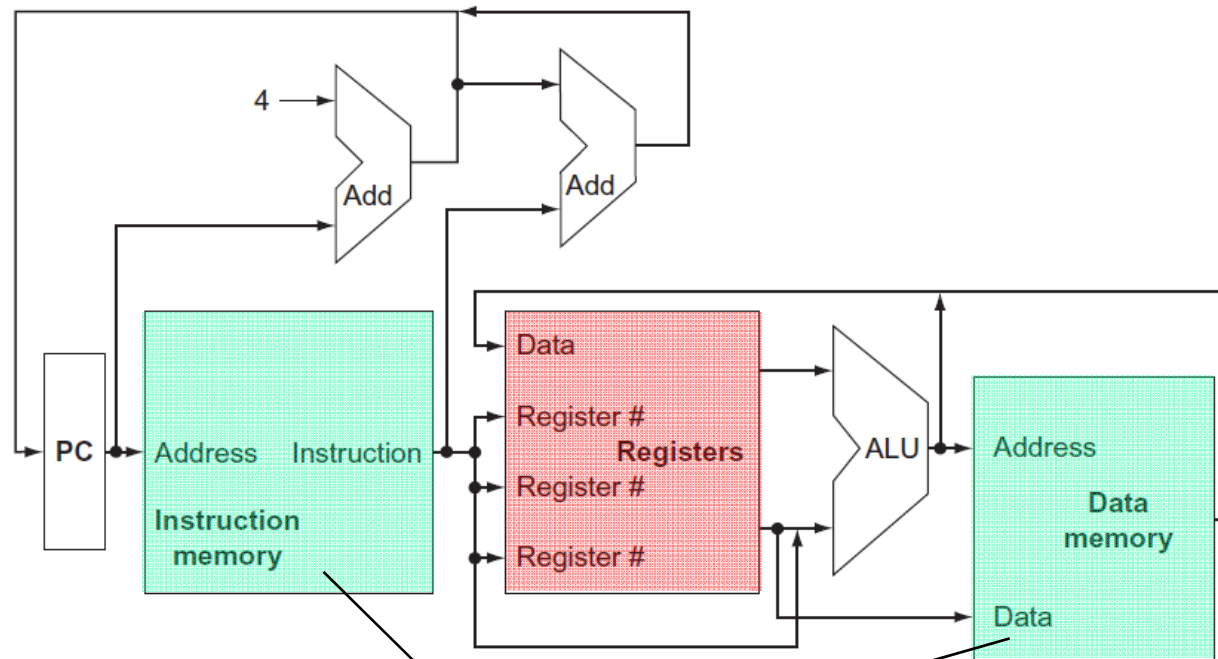
2019/01/15

メモリ

図の多くはPatterson, Hennessy: Computer organization and design 5th editionより引用

メモリについて

- 前回までプロセッサの内部構造を学んできた

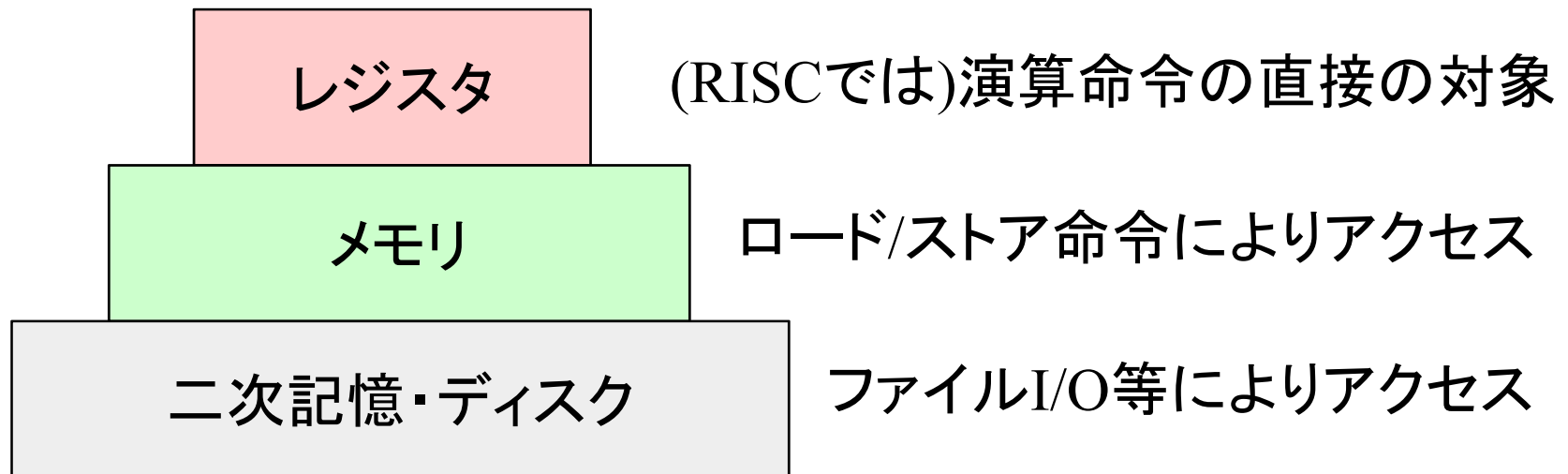


今回のトピック: **メモリ**

これまで、メモリは巨大な記憶装置とだけ考えてきた
⇔ レジスタは32bit × 32個しかない

計算機の記憶階層 (ユーザからの視点)

計算機内でデータを保持する場所は複数あり、利用方法はISAレベルで区別されている



ユーザから見たメモリの特徴 (1)

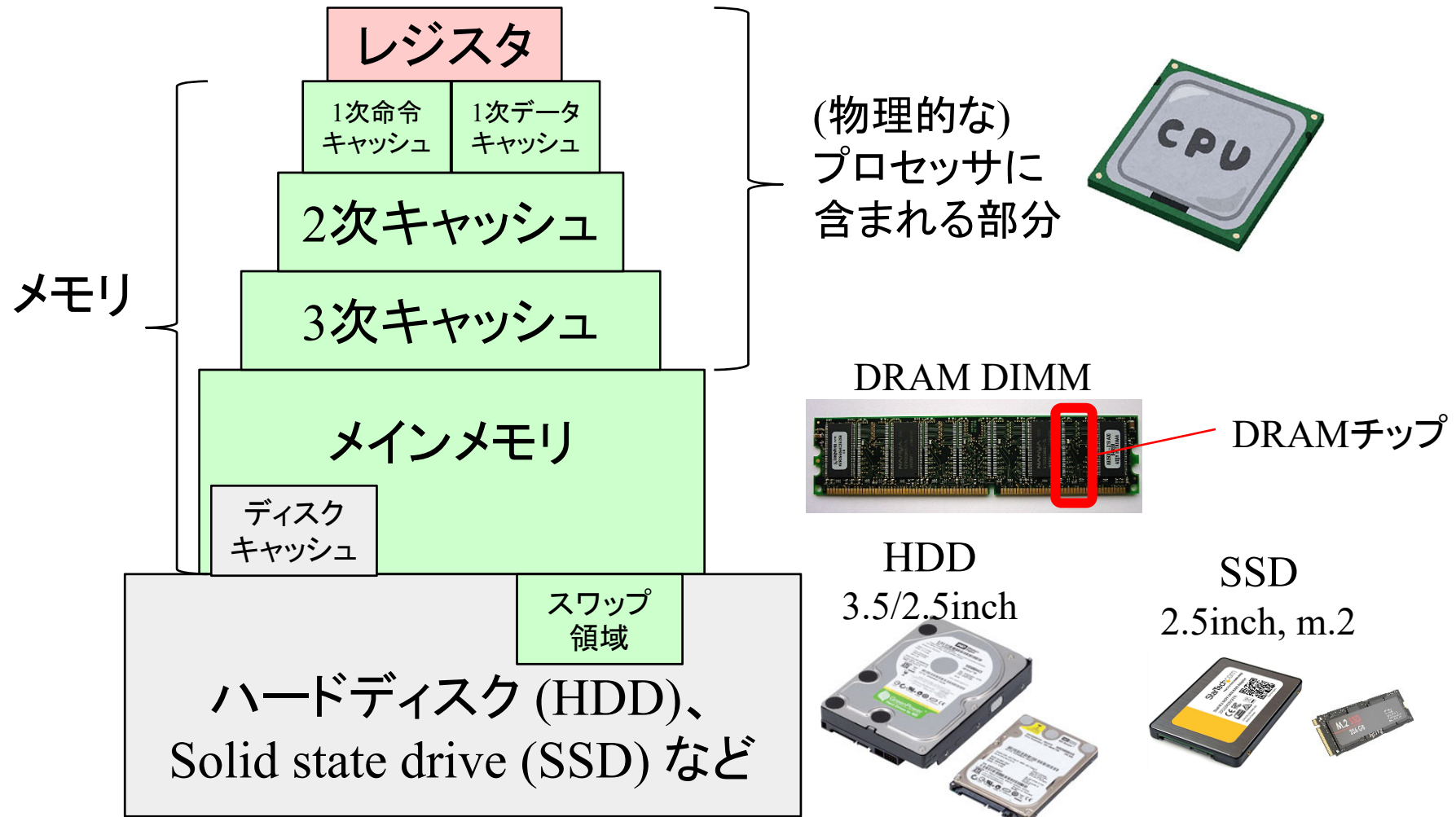
メモリは、ユーザからは、以下のような動作をするユニットとして見える

- アドレス($0 \sim 2^{32}-1$ のいずれか)を指定して、データにアクセスする
 - 64bitプロセッサならアドレスは $0 \sim 2^{64}-1$
- 1つのアドレスには**1Byte (=8bit)**のデータが格納
- 連続した1, 2, 4Byte単位...でアクセス
 - MIPSの場合、
 - lw/swは4Byte
 - 例: \$2が10000のとき、lw \$1, 0(\$2)を行うと、10000~10003の4つの連続アドレスから、4Byte=32bitのデータを読み込む
 - lhu/shは2Byte
 - lbu/sbは1Byte
 - 64bitプロセッサの場合、8Byteアクセス命令あり

ユーザから見たメモリの特徴 (2)

- どんなアドレスにもアクセスできるわけではない。有効でないアドレスをアクセスすると、例外(ハードウェア的なエラーコードのようなもの)が起き、通常はプログラムが停止する
 - 有効な命令も、データ(変数やmallocの結果など)も、無いアドレス
 - システム的には、メモリマップされていないアドレス
 - 4で割り切れないアドレスに対してlw/sw
- 容量は、 2^{32} Byte (=4GB)あるとは限らない
 - 2^{64} Byte (=16EB)の容量を持つ計算機は存在しない
- 物理アドレス/仮想アドレスの違いあり。後述
- 電源を落とすとデータは消える(揮発性)。ディスク(不揮発性)と違う


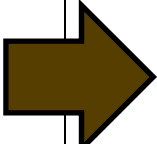

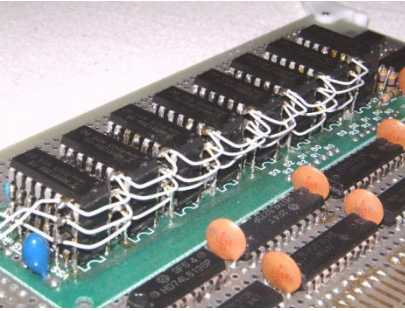
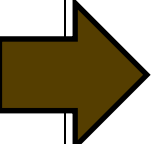

計算機の実際の記憶階層 (典型的なもの)



- **キャッシュ (cache):** メモリのデータのうち、一部の大事なデータを格納する箇所
 - 3次キャッシュ (Level 3 cache)を、L3\$と略記すること
 - 同様に、L2\$, L1I\$, L1D\$

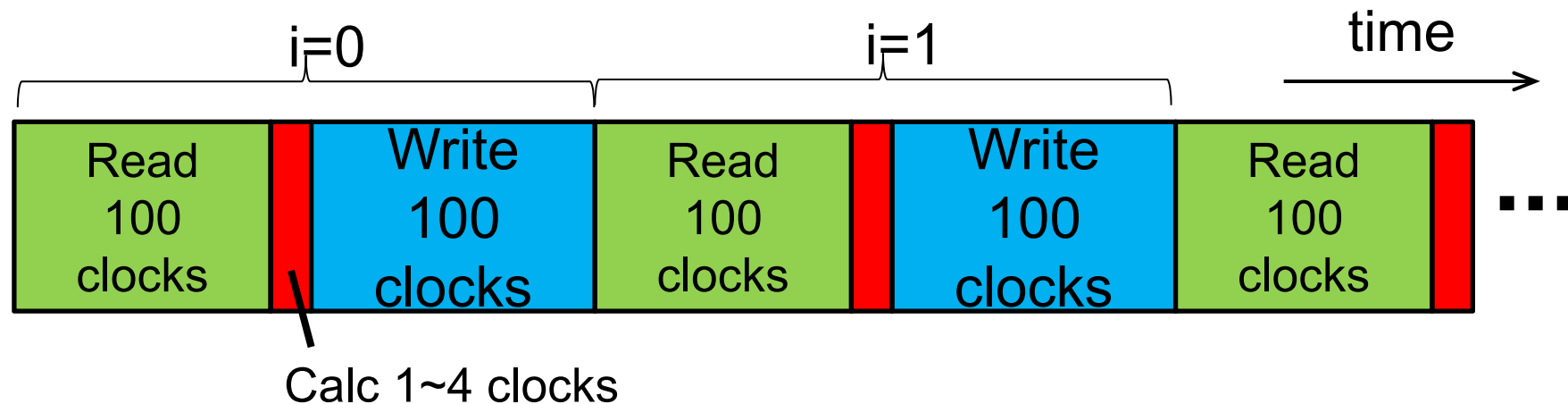
なぜ現代の計算機では単純なメモリ構造ではいけないか

- 一枚岩のフラットなメモリの構造では、現在必要とされる
 - 高速なアクセス
 - 大容量を両立できないため → **キャッシュの必要性**

	Around 1980		Present
CPU	2MHz → 1clock = 500ns 	x1000 	2GHz → 1clock = 0.5ns 
Memory (DRAM)	Access time = 2000ns(?) 	x40(?) 	Access time = 50ns or more  DRAMアクセスは 100クロック以上

もしも、あらゆるメモリアクセスに100clockかかる と？

```
for (i = 0; i < n; i++) { A[i] = A[i]*2.0; }
```



非効率的すぎる

数GHzのCPUでも、10MFlopsを越えられない

このような問題の解決に向けて、
キャッシュをはじめ様々な技術が提案されてきた

キャッシュの役割

- キャッシュは、メインメモリより小さく、速い記憶装置
 - キャッシュの中でも、1次キャッシュが最も速く、小さい
 - 「最近」アクセスされたメモリの内容を覚えておく
 - 3次キャッシュ、2次キャッシュ、1次キャッシュに複製しておく
 - 次回に同じアドレスがアクセスされたら、キャッシュからデータをプロセッサに供給
→ **速度向上！**
- 一般に、一回のメモリアクセスは以下に分類される
- 欲しかったデータがキャッシュに存在した → **キャッシュヒット**
 - キャッシュに存在せず、メインメモリをアクセスする必要があった → **キャッシュミス**
 - より厳密には、1次はミスしたが2次でヒットした...などが起こる
 - キャッシュの容量は小さい → いつかはデータを破棄する必要

なぜキャッシュはうまくいく(と思われている)か？

多くのソフトウェアが、以下のような性質を持つため

- 一度アクセスされたアドレスは、近い将来またアクセスされる可能性が高い
(**時間的局所性**)
- 一度アクセスされたアドレスの近傍が、近い将来アクセスされる可能性が高い
(**空間的局所性**)

例:

```
for (i = 0; i < 100; i++) C[i] = A[i] + B[i];
```

→ **A[10]のあとA[11]がアクセスされ、その後A[12]が...**

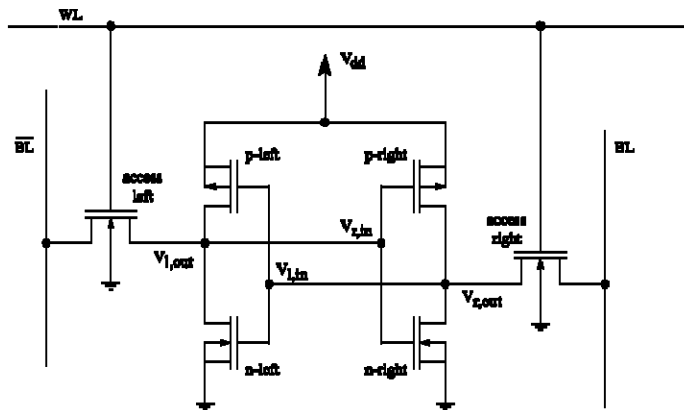
関数呼び出しのスタックフレーム

関数fがf1を、次にf2を呼び出すとき、f1とf2のスタックフレームは同じ/近い領域が使いまわされる

キャッシュとメインメモリのハードウェア

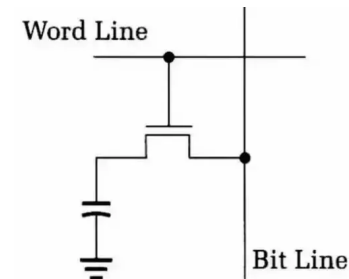
- **SRAM (Static Random Access Memory)**

- 主にキャッシュに使われる
- Flip-Flopに近い回路により1bitを記憶
- トランジスタでできている



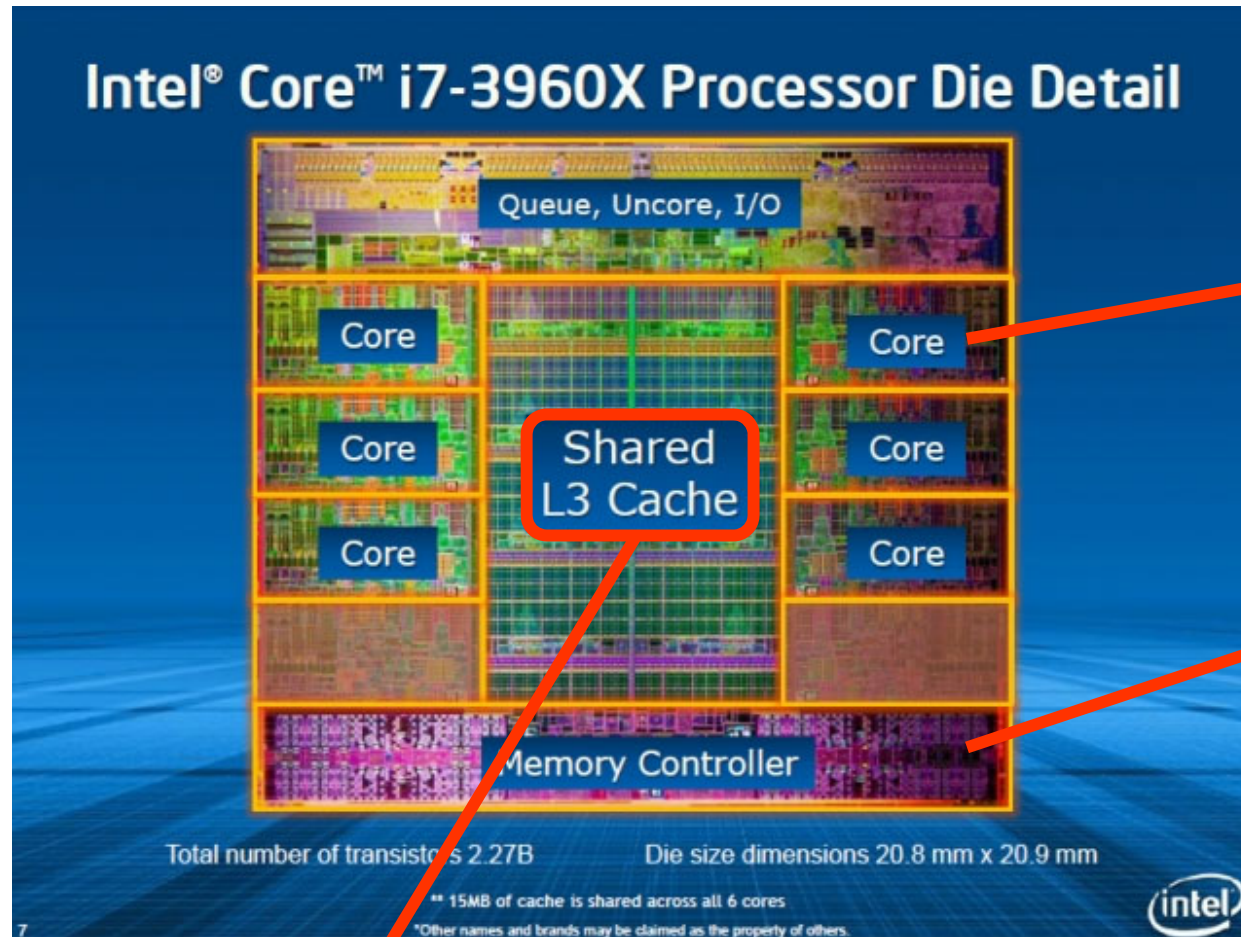
- **DRAM (Dynamic Random Access Memory)**

- 主にメインメモリに使われる
- キャパシタ(capacitor)の電荷で1bitを表現
- 電荷の消失を防ぐため、約100msおきに"refresh"という動作が必要



SRAMのほうが高速だが、1bitの面積は大きく大容量不向き → キャッシュへ
DRAMは大容量可能だが、プロセッサと同じチップ搭載が困難 → メインメモリへ

プロセッサ内に含まれたキャッシュ



L1, L2キャッシュは
各コアに含まれる

メモリコントローラ:
外部のメインメモリ
とやりとりする機構

近年L3キャッシュは
数～数十MB

ITmediaより引用

実際のプロセッサのキャッシュ/メモリ性能の例

Intel Haswell

Intel i7-4770 (Haswell), 3.4 GHz (Turbo Boost off), 22 nm

L1 Data cache = 32 KB, 64 B/line, 8-WAY.

L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.

L2 cache = 256 KB, 64 B/line, 8-WAY

L3 cache = 8 MB, 64 B/line

L1 Data Cache Latency = 4 cycles for simple access via pointer

L1 Data Cache Latency = 5 cycles for access with complex address calculation

L2 Cache Latency = 12 cycles

L3 Cache Latency = 36 cycles

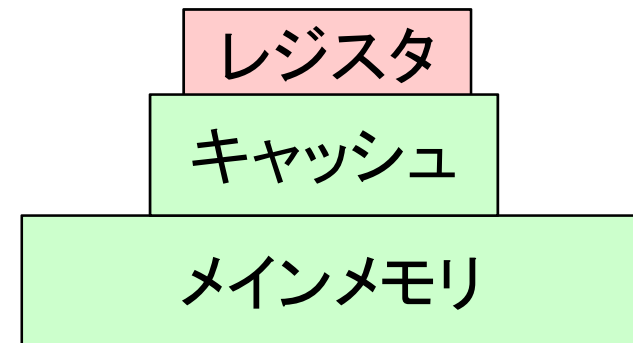
RAM Latency = 36 cycles + 57 ns

<https://www.7-cpu.com/cpu/Haswell.html>

※ メインメモリ容量は、PCでは自由。数～数十GB

これから考えるキャッシュ

- 1層だけ。命令とデータの区別もしない
- 容量: 16KB
- キャッシュブロック (cache block, または cache lineとも)のサイズ: 16B
 - つまり、 $16\text{KB}/16\text{B} = 1024$ 個のブロックからなるキャッシュ

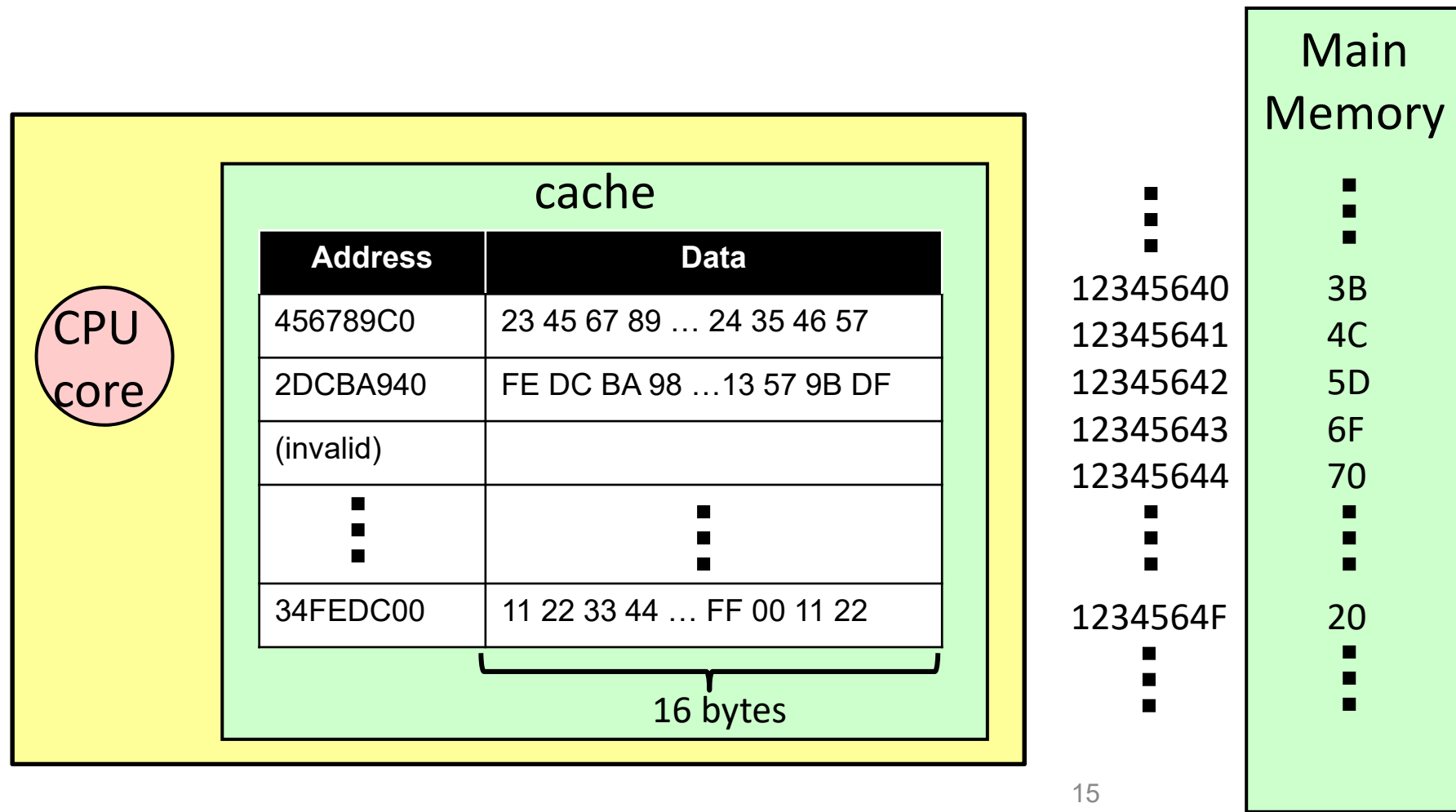


キャッシュブロックとは？

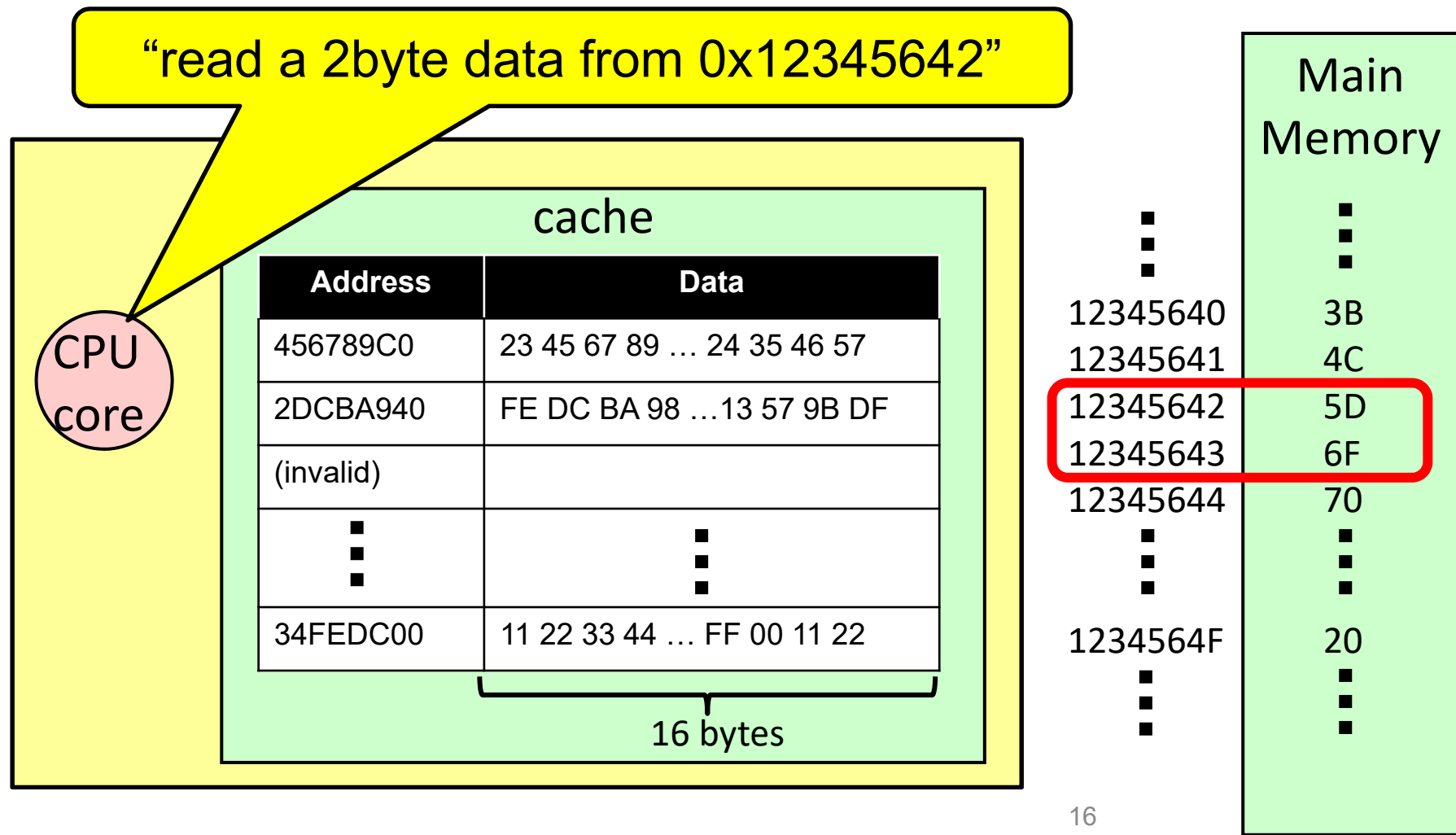
- 機械語レベルのメモリアクセス単位は1B, 2B, 4B...だが、メインメモリ ⇔ キャッシュ間のやりとりは、**固定長の単位(=キャッシュブロック)**ごとで行う
 - 今回の例では16B単位。Intel CPUでは64B単位
- 例: ab004 (16進)のアドレスへアクセスがあった → 実は、ab000～ab00f (16進)の16バイトがキャッシュに複製される。
- なぜこんなことを？
 - (1) ハードウェアを簡単にするため
 - (2) 空間的局所性に対応するため

**A[0], A[1], A[2], A[3].... (各要素4B)を順にアクセスするとき、
キャッシュミスは4回に1回だけ → キャッシュミス率25%, ヒット率75%**

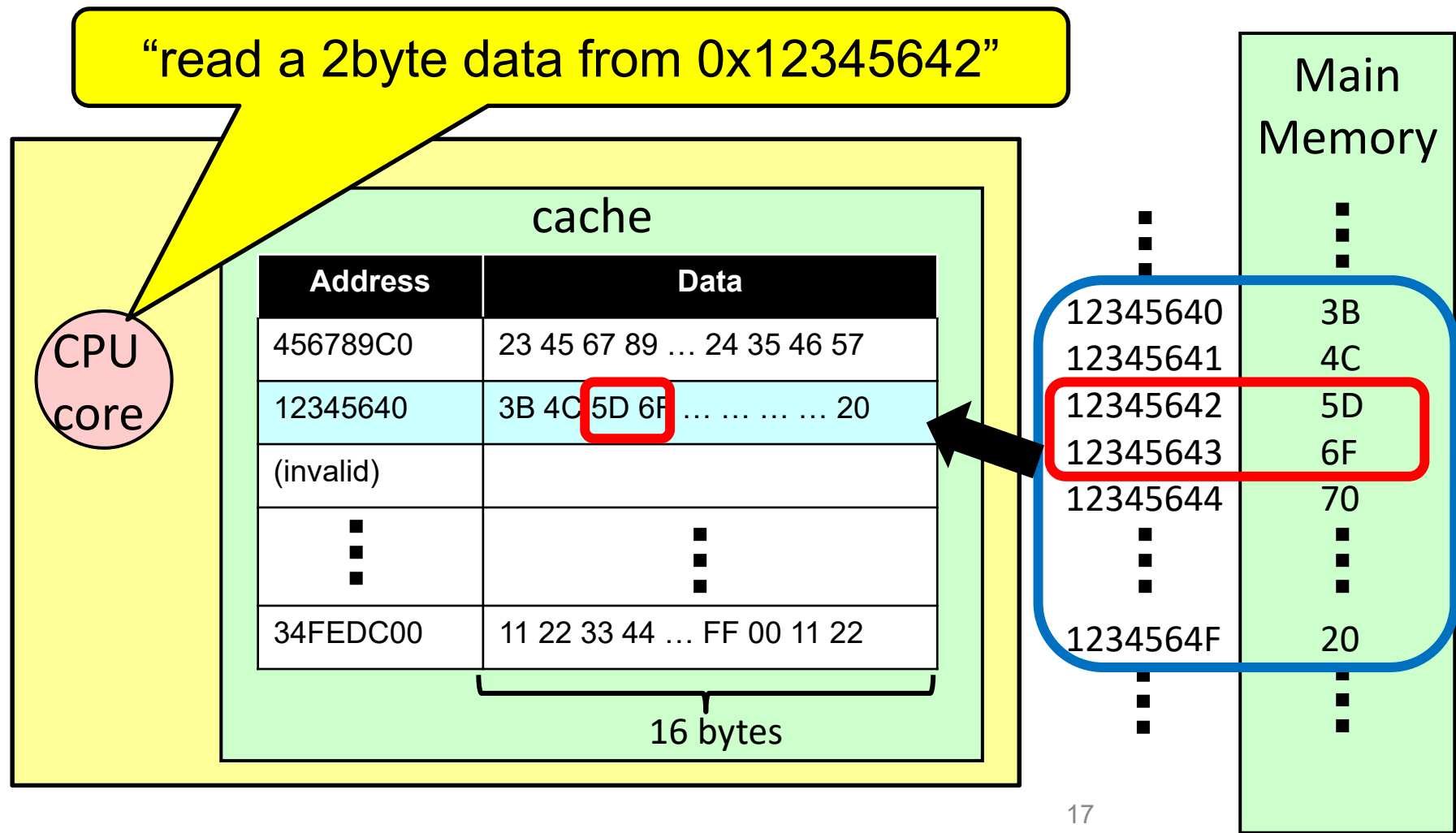
キャッシュのイメージ



プロセッサが読み込み命令を行うとき



プロセッサが読み込み命令を行うとき



キャッシュの実現方法の選択肢

- メインメモリのアドレスとキャッシュ内容をどう対応させるか？
 - ダイレクトマップ (direct mapped)方式
 - セットアソシアティブ (set associative)方式
 - フルアソシアティブ (full associative)方式
 - キャッシュがあふれたときの対応も異なる
- 書き込み命令 (swなど)のときどうするか？
 - ライトスルー (write through)
 - ライトバック (write back)

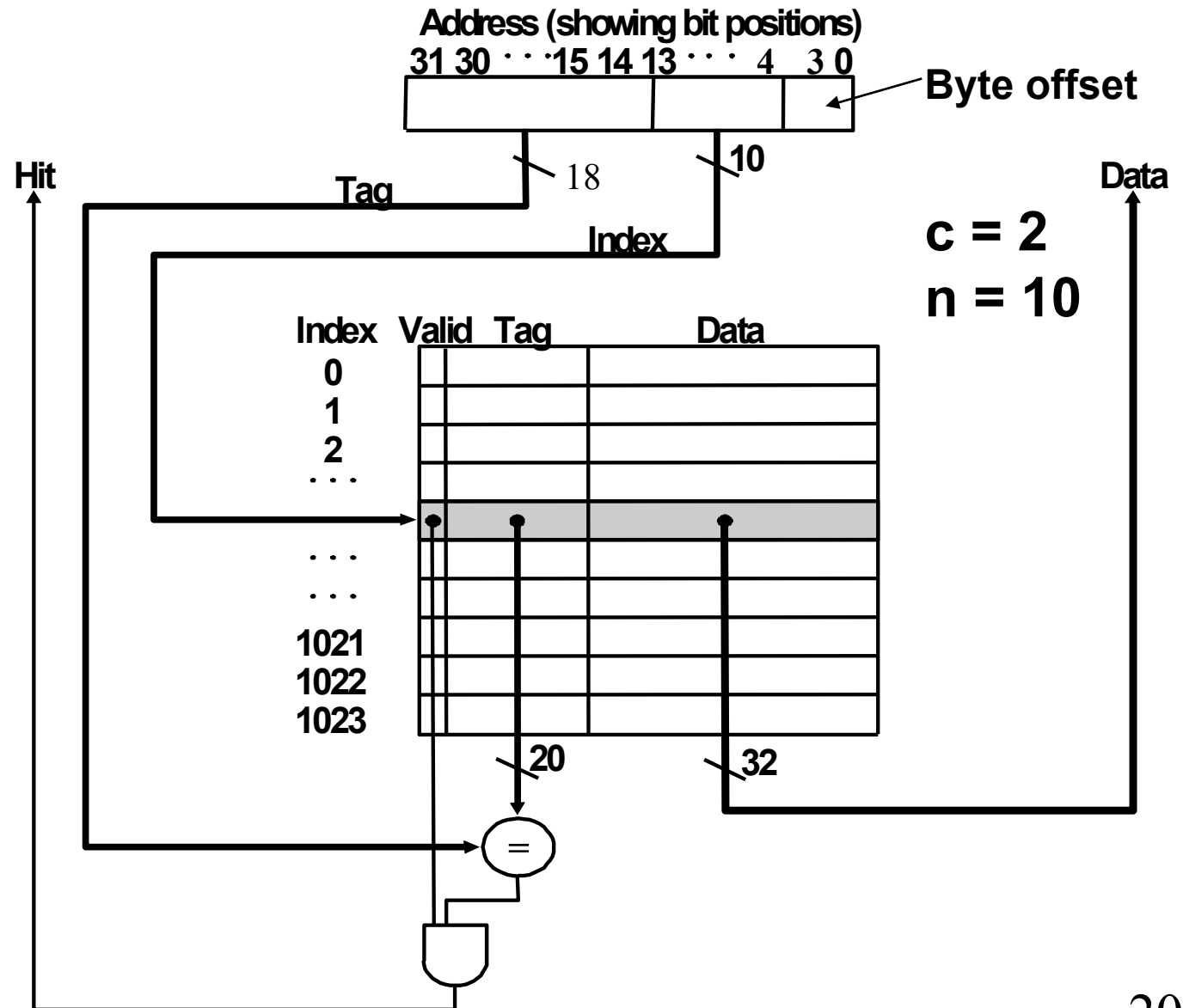
最も単純なキャッシュ: Direct Mapped キャッシュ

- キャッシュブロックサイズが 2^c , ブロック数 2^n とする
 - ここでは $c = 4$ (16B), $n = 10 \rightarrow$ キャッシュは 16KB
- 32bitアドレスを以下のように解釈



- 「Index部分」が、キャッシュ内のどのブロックに相当するか決定
- アクセス時には、tagを見てキャッシュヒットかミスか判断

Direct Mapped キャッシュの実装



ダイレクトマップの長所と短所

- 長所: ハードウェアが(他より)単純、高速
- 短所: 運の悪いときのキャッシュミスが増えてしまう
 - ブロックが重複して追い出されるデータ (victim) の選択アルゴリズムが悪い

例:

```
for (i = 0; i < 100; i++) C[i] = A[i] + B[i];
```

もし配列A, B, Cのアドレスの差が $16B \times 1024$ の倍数だったら?

A[16]がキャッシュに載ったあと、B[16]のアクセスにより追い出されてしまう

→ せっかく空間的局所性があるのに、毎回キャッシュミスしてしまう

理想的にはたとえば、全ブロックの中から一番最近使われていない (least recently used/LRU) データを選んで捨てる...などがよい

→ これは、フルアソシアティブ方式と呼ばれるが、ハードウェアで実装が困難・遅い!

- 中間の方法である、セットアソシアティブ方式がよく使われる

セットアソシアティブ方式の考え方

tag	index	あるアドレス
-----	-------	--------

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

ダイレクト:
収まるべきキャッシュ
ラインが一意に決まる
→ 運が悪いと衝突多い

セットアソシアティブ:
収まるべきキャッシュ
ラインがway数だけある
→ 衝突を減らせる
同じセットの中では、
LRUでvictimを決める

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

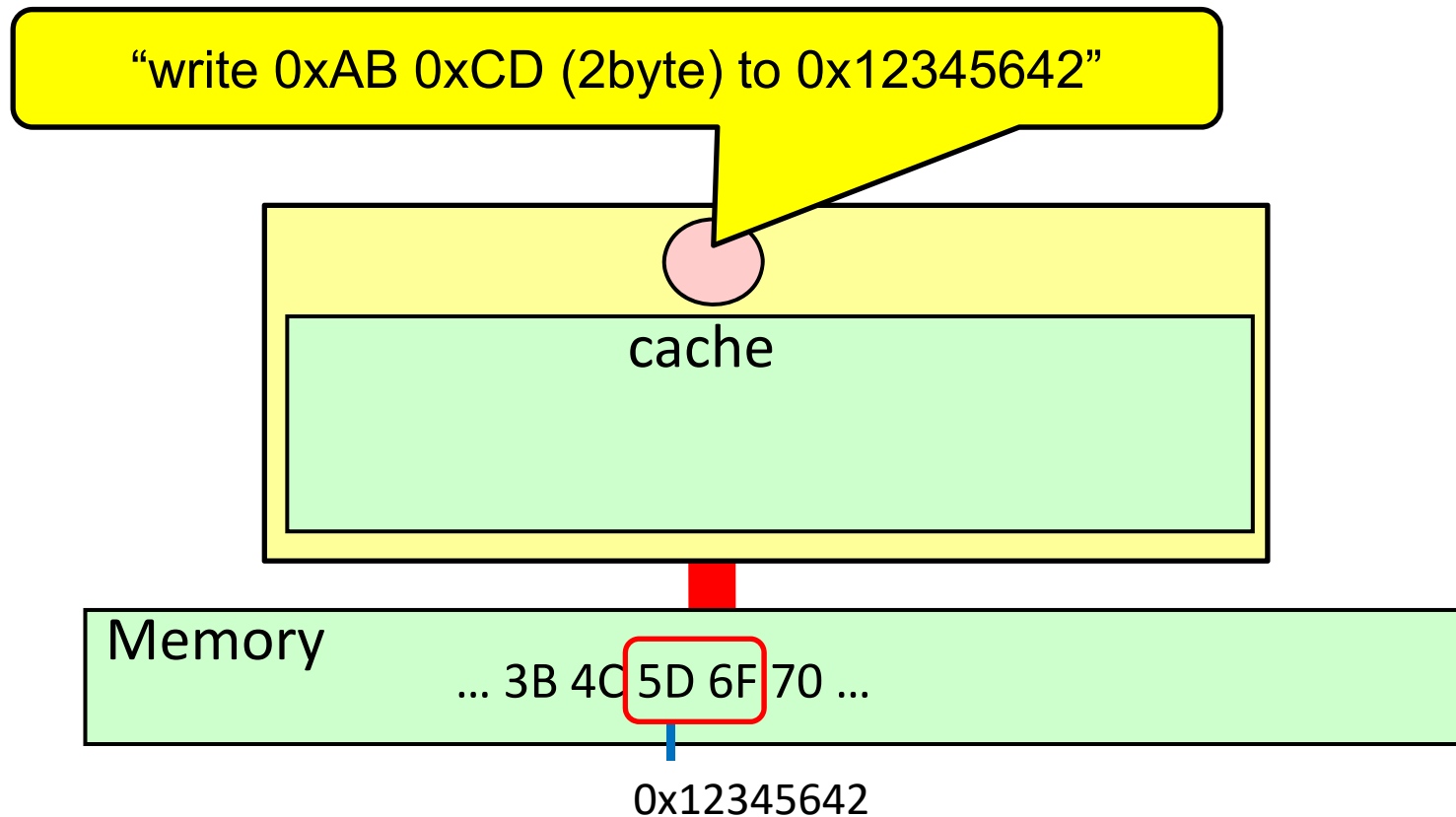
Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

フルアソシアティブ: 衝突を最小にできると期待されるが、
実装が困難

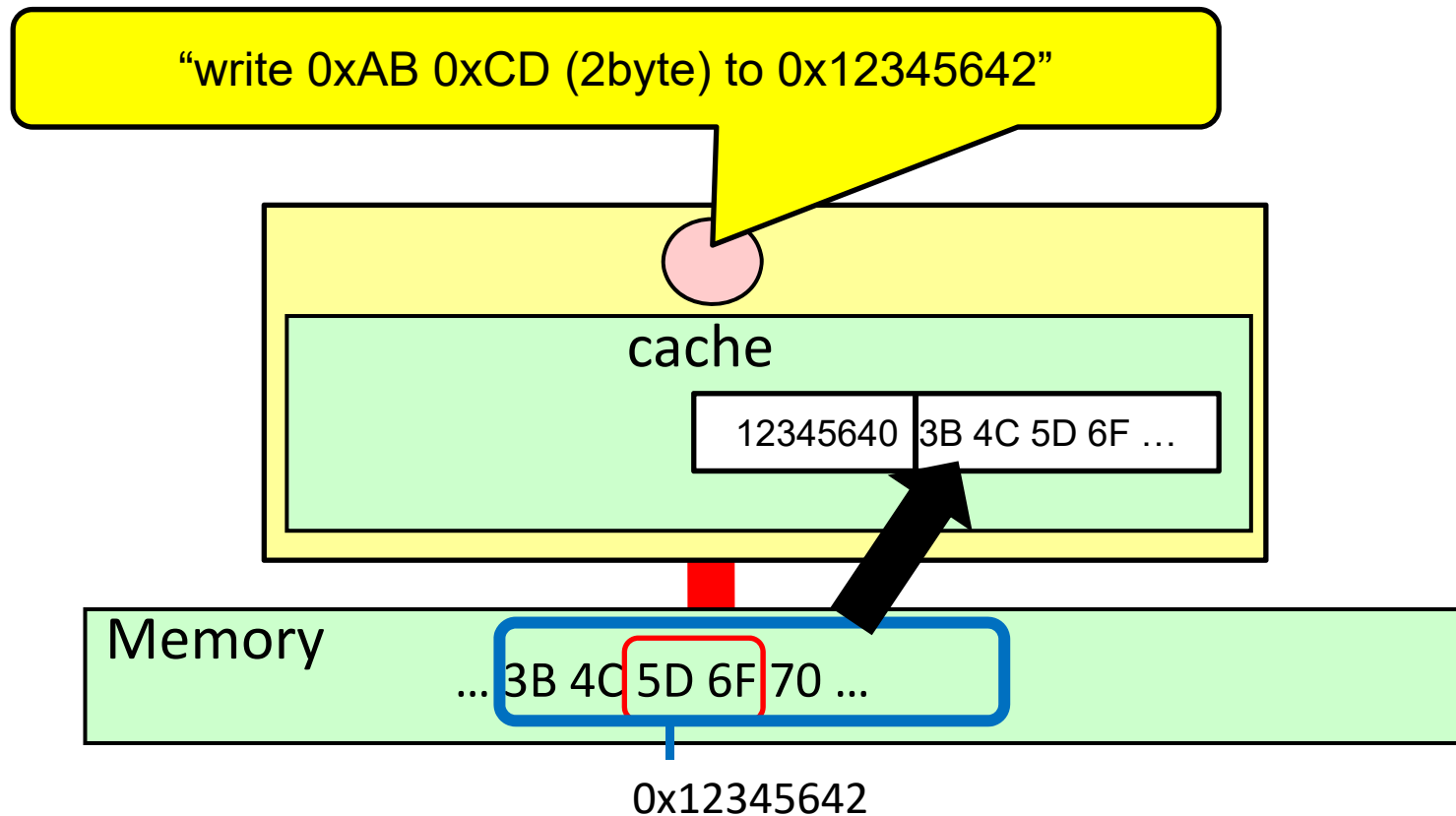
書き込み時のキャッシュ

- 書き込みアクセスは、読み込みより複雑となる
 - データを変更が起こる → 一貫性の問題



書き込み時のキャッシュ (続き)

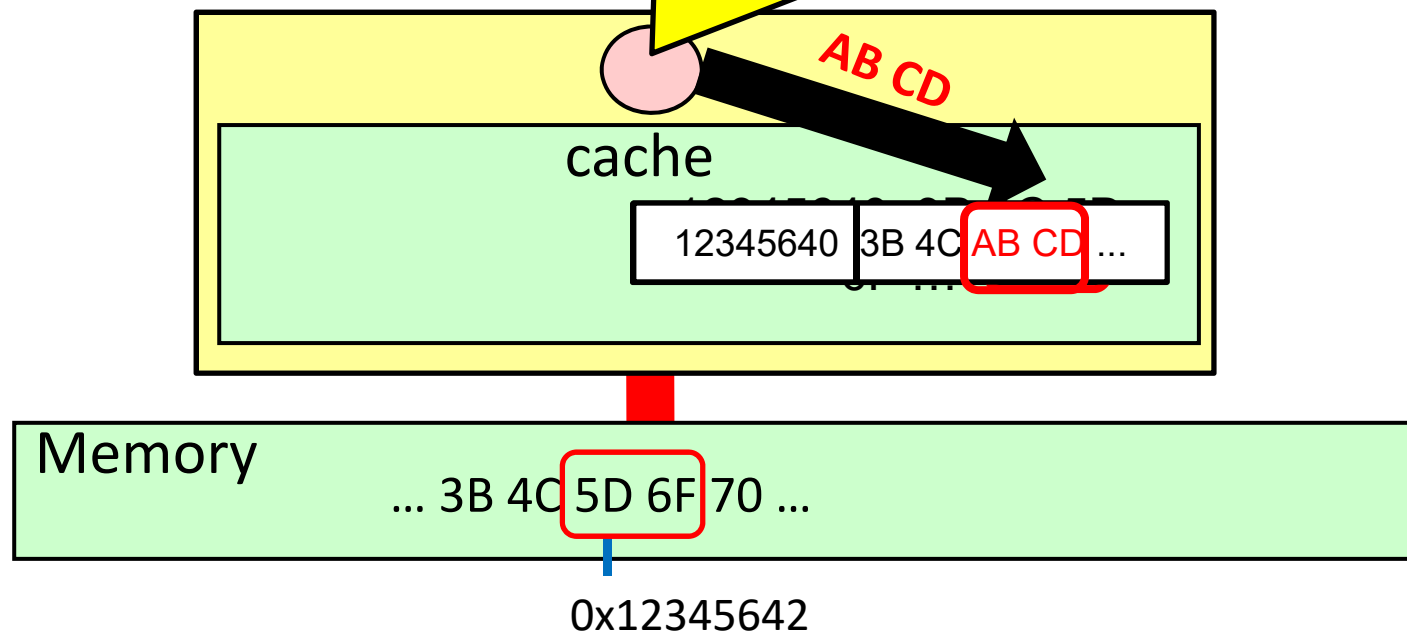
(ミス发生时) キャッシュブロック全体をメモリからキャッシュへコピー、つまり「読み込み」相当が起こる



書き込み時のキャッシュ (続き)

最後に、キャッシュ上のデータの一部(ここでは2B)を変更

“write 0xAB 0xCD (2byte) to 0x12345642”



この時点で、キャッシュとメインメモリ上のデータが
食い違っている？どう対応するか

書き込みポリシー

- ライトスルー (Write through)
 - 下層のデータをすぐに更新
 - **欠点**: 全書き込み命令がメインメモリを用いてしまうため、>100 clocks
- ライトバック (Write back) -- **こちらが主流**
 - 下層のデータを**後で更新**
 - 将来、**該当のキャッシュブロックが(victimとして選ばれて)追い出される**とき、はじめて下層を更新

キャッシュの存在により起こること

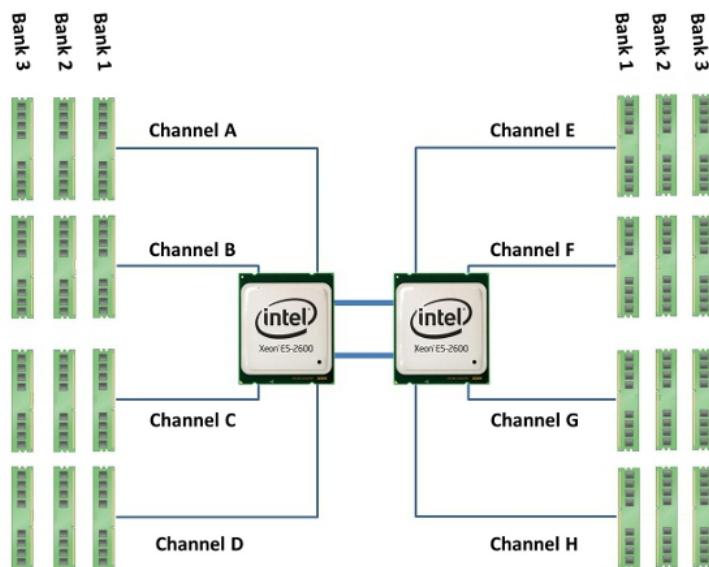
- メモリアクセスにかかる時間は一定でない
 - 単にlw \$2, 100(\$1)と書いてあっても
 - Cache hit時なら数clock
 - Cache miss時なら数百clock
 - マルチコアによるアクセス衝突があるともっと
- キャッシュブロックの存在により、連続アドレスを連続してアクセスするのが速い
 - 下記のプログラムには性能差が！
 - どちらも計算量は $O(mn)$ だが...

```
for (i=0; i<m;i++) {  
  for (j=0;j<n;j++) {  
    A[j][i] *= 2.0; }}  
A[j][i] *= 2.0; }}
```

```
for (j=0;j<n;j++) {  
  for (i=0; i<m;i++){  
    A[j][i] *= 2.0; }}
```

近年のメインメモリの技術

- メインメモリが(相対的に遅いので)キャッシュが重要
- しかし、メインメモリ/DRAMの技術向上をさぼっているわけではない
- 遅延の短縮はキャッシュにまかせ、大容量の方向へ
 - スループットもなるべく稼ぐ(キャッシュに勝てないが ☹)
- PC/サーバの場合はDRAMを載せたDIMMモジュールである
- プロセッサと、メインメモリの間の通信路 = メモリチャネル
 - DIMMの速度 × メモリチャネル数 = メインメモリのスループット



Cisco web pageより

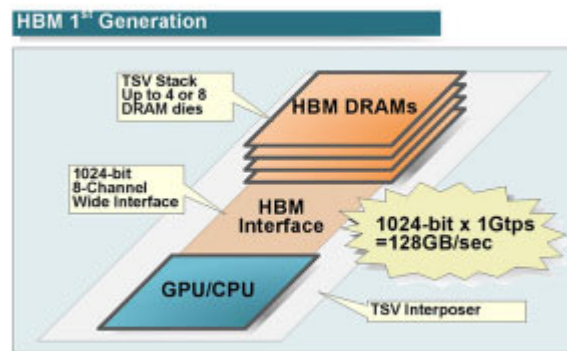
例: プロセッサあたり4メモリ
チャネル × 2プロセッサの図。

もしDDR4-2400 (19.2GB/s)で
あれば、総スループットは
 $19.2\text{GB/s} \times 4 \times 2 = 153.6\text{GB/s}$

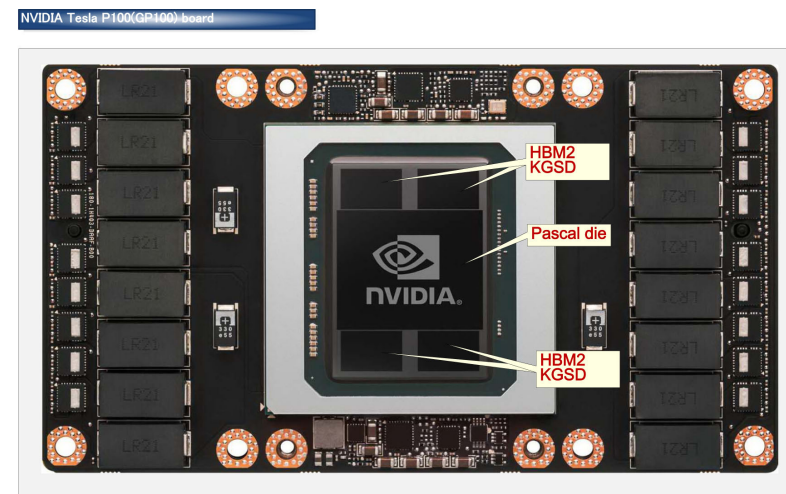
図ではチャネルあたり3枚DIMM
→ 容量は増えるが速度には寄与しない

近将来のメモリ技術

- ストレージの分野では、ハードディスクだけでなくフラッシュメモリが急速に普及
 - 例: SSD, USBメモリ (という名のストレージ)
- メモリの分野でも、新しい動き
 - DRAMの集積方法を変える
 - TSUBAME GPUでも用いられているHBM (High bandwidth memory)
... DRAMを三次元積層



Copyright (c) 2013 Hiroshige Goto All rights reserved.



Copyright (c) 2016 Hiroshige Goto All rights reserved.

- DRAM以外のメモリ、特に不揮発メモリ: MRAM, Intel 3D XPoint...
- メインメモリとストレージの区別が薄れる動きも