パタヘネ第五章

Arch kota

目次

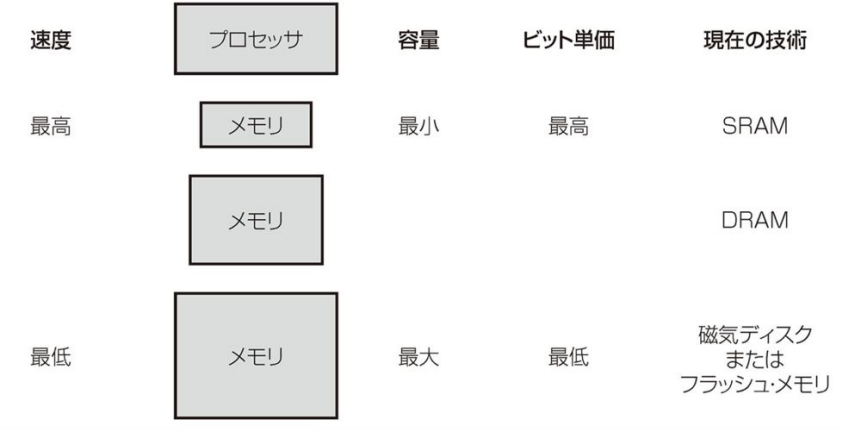
- 記憶の階層化
- 記憶容量の種類
- キャッシュの基礎
- キャッシュの性能向上
- 仮想記憶
- 誤り訂正/誤り検出

記憶の階層化とは？（図書館の例）

- レポートを書くために図書館に行き、レポートの内容に適した本を見つけて机の横に積む
- 最も必要であろう情報を近くに置いておき、それ以外は本棚に止まる
  - これが階層化
- 必要であろう情報の性質 = **局所性**
  - 時間的局所性：ある情報が参照された時、それはまもなく再び参照される確率が高い
  - 空間的局所性：ある情報が参照されたとき、その近くにある情報もまた参照される可能性が高い

コンピュータにおける記憶の階層化

- 記憶容量においてやり取りされるデータの最小単位を**ブロック**という
  - ブロックには複数ワードが入る
  - ブロック単位でキャッシュに乗ったり乗らなかったりする=**空間的局所性**
  - あるブロックへのアクセスが発生したら上位レベルのメモリにデータコピー=**時間的局所性**



記憶容量の種類

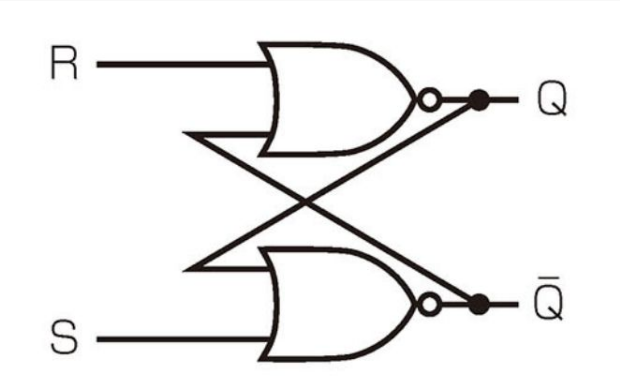
- メモリへのアクセス時間が短いものほど上位レベルのメモリに利用される
  - プロセッサの高速化に適応するため（ノイマンボトルネック&アムダールの法則）
  - 一般に主記憶として用いられるDRAMと二次記憶として用いられるフラッシュメモリ、磁気ディスクの間には大きなアクセス時間の差がある

メモリ・テクノロジ	代表的なアクセス時間	2020 年時点の 1 ギビバイト当たりのコスト
SRAM	0.5 ～ 2.5ns	500 ～ 1,000 ドル
DRAM	50 ～ 70ns	3 ～ 6 ドル
フラッシュ半導体メモリ	5,000 ～ 50,000ns	0.06 ～ 0.12 ドル
磁気ディスク	5,000,000 ～ 20,000,000ns	0.01 ～ 0.02 ドル

クロック入力を持たない記憶装置

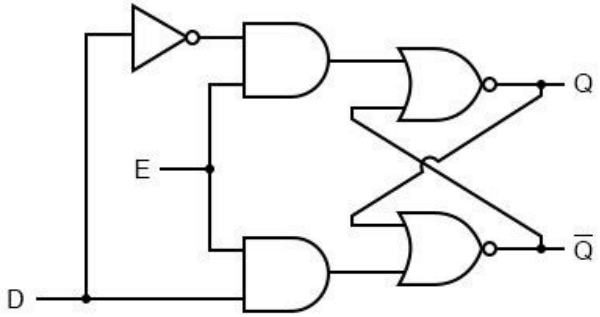
SRラッチ（SR Latch）

- 一対のNORもしくはNANDゲートから構成される
- SとR（セット、リセット）へのアサートがなければQとQの前の値を保持する。Sが1ならばQは1, Rが1ならばQは0となる
- S/Rがアサートされていれば入力の変化に応じていつでも状態が変更する



Dラッチ

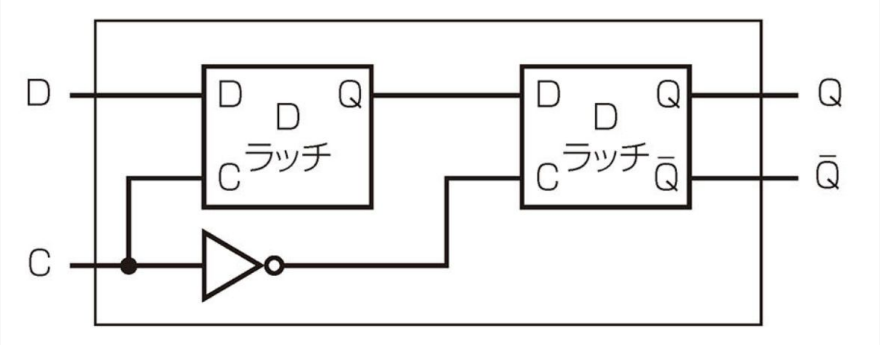
- 入力はデータ値とクロック信号
- クロック信号が0の時、SRラッチの値が保持される
- クロック信号が1の時、データ値の変化につれて出力値が変わる



E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

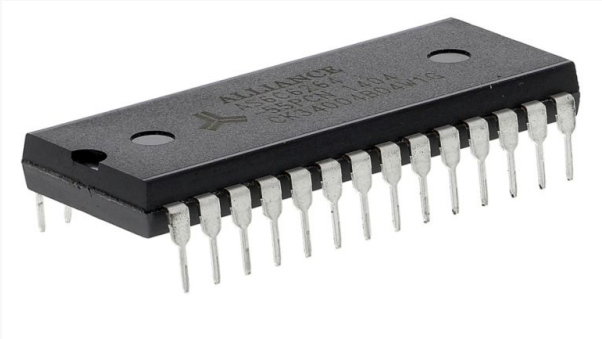
Dフリップフロップ

- 下図はクロックの立ち下がりをトリガーとする
- クロック入力がアサートされると最初のラッチが開いてDが取り込まれる
- クロック入力Cが0になると2段目のラッチが開いてQが出力される



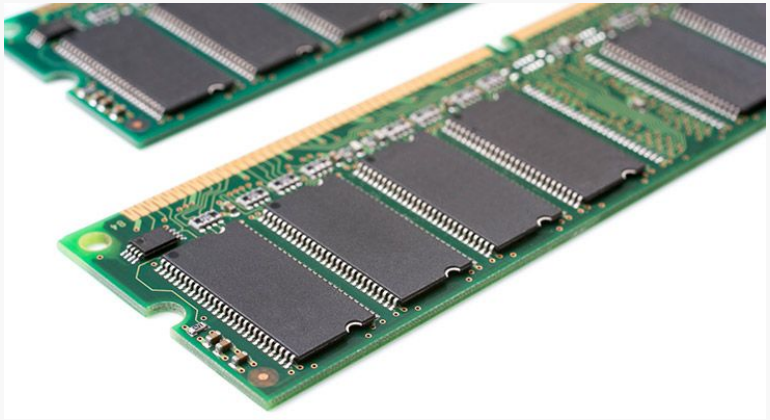
SRAM

- 配列上の記憶構造をした集積回路
- フリップフロップ回路を利用しているためリフレッシュ動作は必要ない
- 1ビットの値の保持に6-8個のトランジスタを用いる
- 主にキャッシュ用に使われ、現代ではほとんどがプロセッサチップ上に統合されている



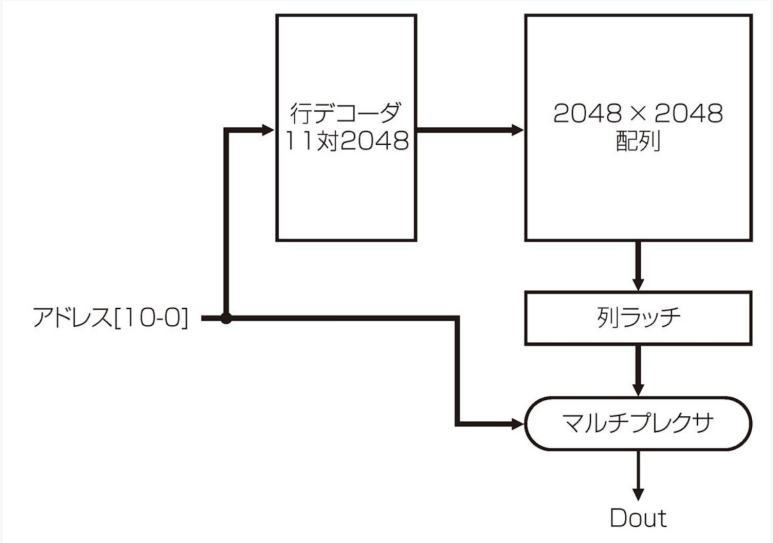
DRAM

- DRAMにおいては1ビットの値を保持するために必要なトランジスタは一つ
  - キャパシタが蓄える電荷にアクセスするために単一のトランジスタを用いる
  - SRAMと比較してビットあたりの密度が高くなり価格が安くなる
- 値は電荷なのでリフレッシュが必要（ダイナミックと呼ばれる理由）



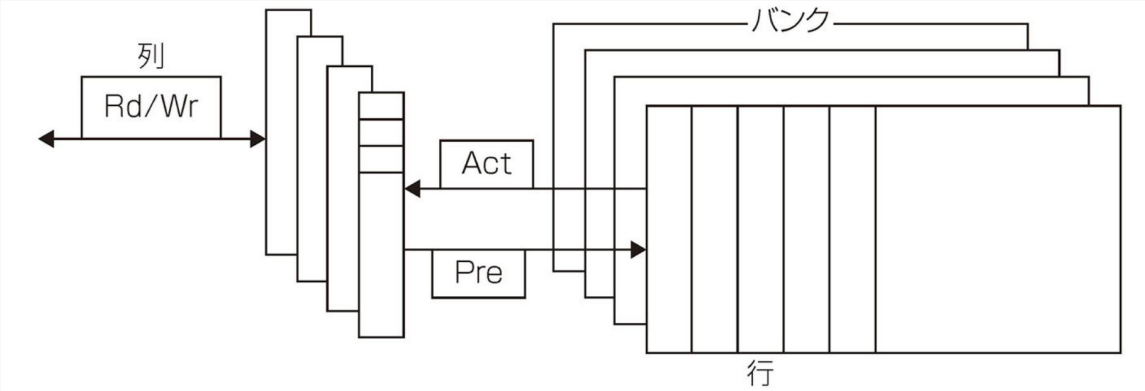
DRAMへのアクセス

- 行アクセス
  - DRAMの配列から１行選択し、それに対応するラインをアサートする
  - その行内容が全て列ラッチに格納される
- 列アクセス
  - 列ラッチから対象の列を選択する
- これがDRAMのアクセス時間の遅さの原因



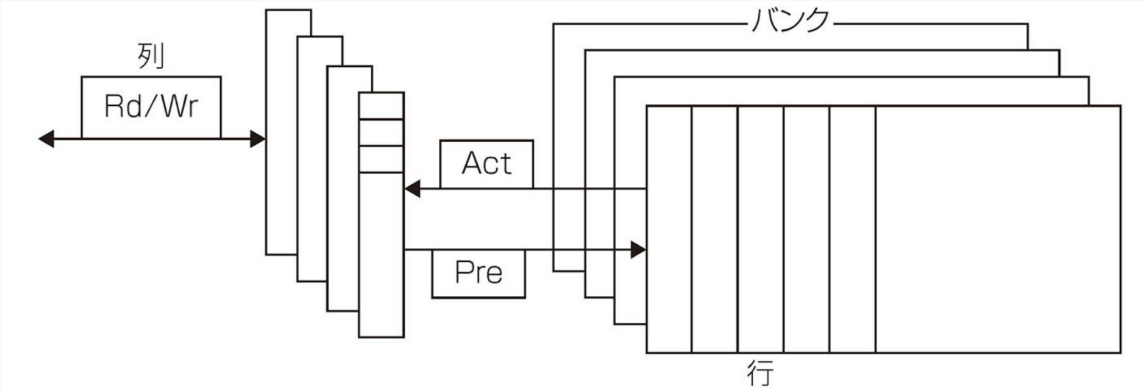
読出の後に列ラッチを書き戻すことでリフレッシュできる

DRAMへのアクセス



- 現代のDRAMは一連の行から構成される複数のバンクで構成される
- 先述のように行アクセスによって列ラッチに行が転送される
  - これはバッファの役割も果たす: バッファ内のデータはSRAMのような働きをする
- SDRAM(Synchronous DRAM)ではクロックが追加された
  - クロックの立ち上がりと立ち下がりエッジの両方でデータ転送を行う機能をDDRという

DRAMへのアクセス



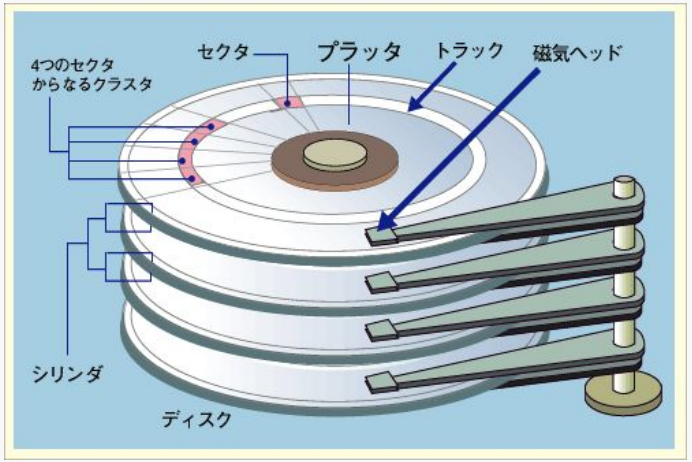
- DDRを用いた大量のデータ転送に適応するためにバンクを複数持つ
- **アドレス・インターリービング**
  - 個々のバンクにバッファを備えることで、アクセス時間を多少犠牲にしてバンド幅を増やす
  - バンクが4つあれば1回のメモリアクセスで4つのバンクを順にアクセスする

フラッシュメモリ

- フラッシュメモリは書き込みを繰り返すとビットが劣化する
  - フローティングゲートなる場所に電荷を蓄えるが、その際に電子は基板とゲートの間にある酸化絶縁膜を突き破って移動する。これを繰り返すと膜が劣化して絶縁性が低下する
- ビットの劣化に対応するために**ウェア・レベリング**が用いられる
  - 何度も書き込まれたブロックへの書き込みを別のブロックへ切り替える
  - ウェア・レベリングを行うことで実質的な歩留まりの改善を行うこともできる

ディスクメモリ

- 磁気ハードディスクは数枚のプラッタからなる
  - 毎分5400-15000回転
- ディスクの一つの同心円をトラックという
  - 盤面あたり数万本のトラック
  - トラックはさらにセクターに分割される



- フラッシュメモリのような書き込みによる劣化はない
- ディスクコントローラーはキャッシュを持つ
  - ヘッドを移動させる過程で通過するセクターの情報が格納される

キャッシュ

キャッシュの役割

- 主記憶のうちの一部のブロックをプロセッサ内で持つ
  - 命令/データへのアクセスが速くなりプロセッサの高速化に適応できる
- 主記憶のうちどのブロックをどのように保持し、どのように書き込むのか

ダイレクトマップ方式

- メモリのロケーションからキャッシュ中のロケーションを一意に求める
  - ブロックアドレス % キャッシュ中のブロック数
- キャッシュ中のブロック数が2の冪乗であれば、ブロックアドレスの下位数ビットがキャッシュ中のインデックスになり都合が良い
- インデックスとして使われなかった上位ビットをタグとして付加する
  - キャッシュ中の一つのインデックスに複数メモリアドレスの内容が入りうるため

ダイレクトマップ

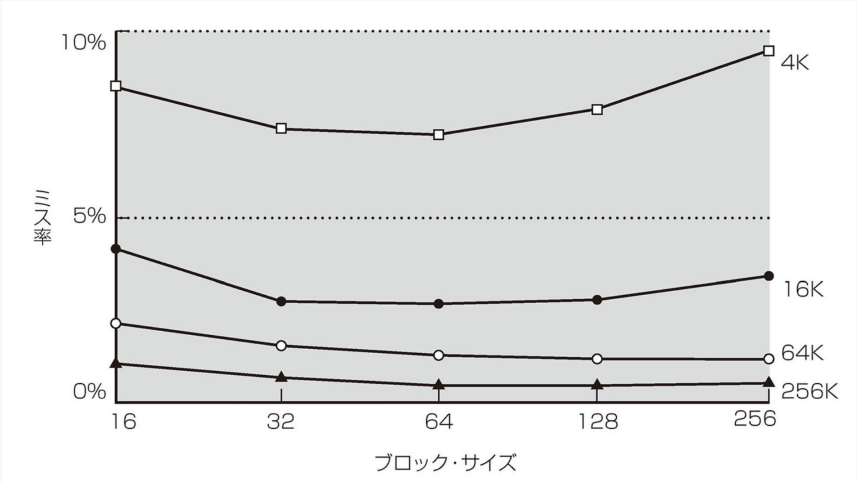
単純化した例：8ブロックからなるキャッシュ

- メモリアドレス00001のキャッシュインデックスは001, タグは00
- メモリアドレス11001のキャッシュインデックスは001, タグは11
    - MIPSにおいては、実際は最下位2ビットはワード内のバイトオフセット指定に使われる

- キャッシュのブロックが有効かどうかは各インデックスに**有効ビット**を付加して判断する
    - プロセッサが立ち上がったばかりの場合など、ブロックが有効でない場合がある

ブロックサイズは大きい方が良い？

- ブロックサイズが大きければ空間的局所性を利用してキャッシュミスが減る
- 大きくしすぎると。。。
  - キャッシュ容量に対して収まるブロックの数が減り、かえってミス率が上がる
  - ブロック転送の時間が増大し、ミス時のペナルティが大きくなる



キャッシュミスの取り扱い

- インオーダー方式のプロセッサであればキャッシュミスが発生するとコンピュータ全体がストールする
  - メモリアクセスが発生し、キャッシュにデータがコピーされるまでレジスタの内容を凍結
- 現代のアウトオブオーダー方式のプロセッサではその間他の命令を実行できる
  - ただ実行速度には大きな影響がある

ライトスルー（Write Through）

- 書き込み時に必ずキャッシュと主記憶の両方に書き込む
- 毎度メモリアクセスが発生すると性能が大きく下がる->**ライトバッファ**
  - 主記憶に対する書き込み待ちデータを蓄えるバッファ
  - プロセッサと主記憶の間に置かれ、プロセッサはキャッシュとライトバッファに書き込むだけで次の処理に移ることが出来る

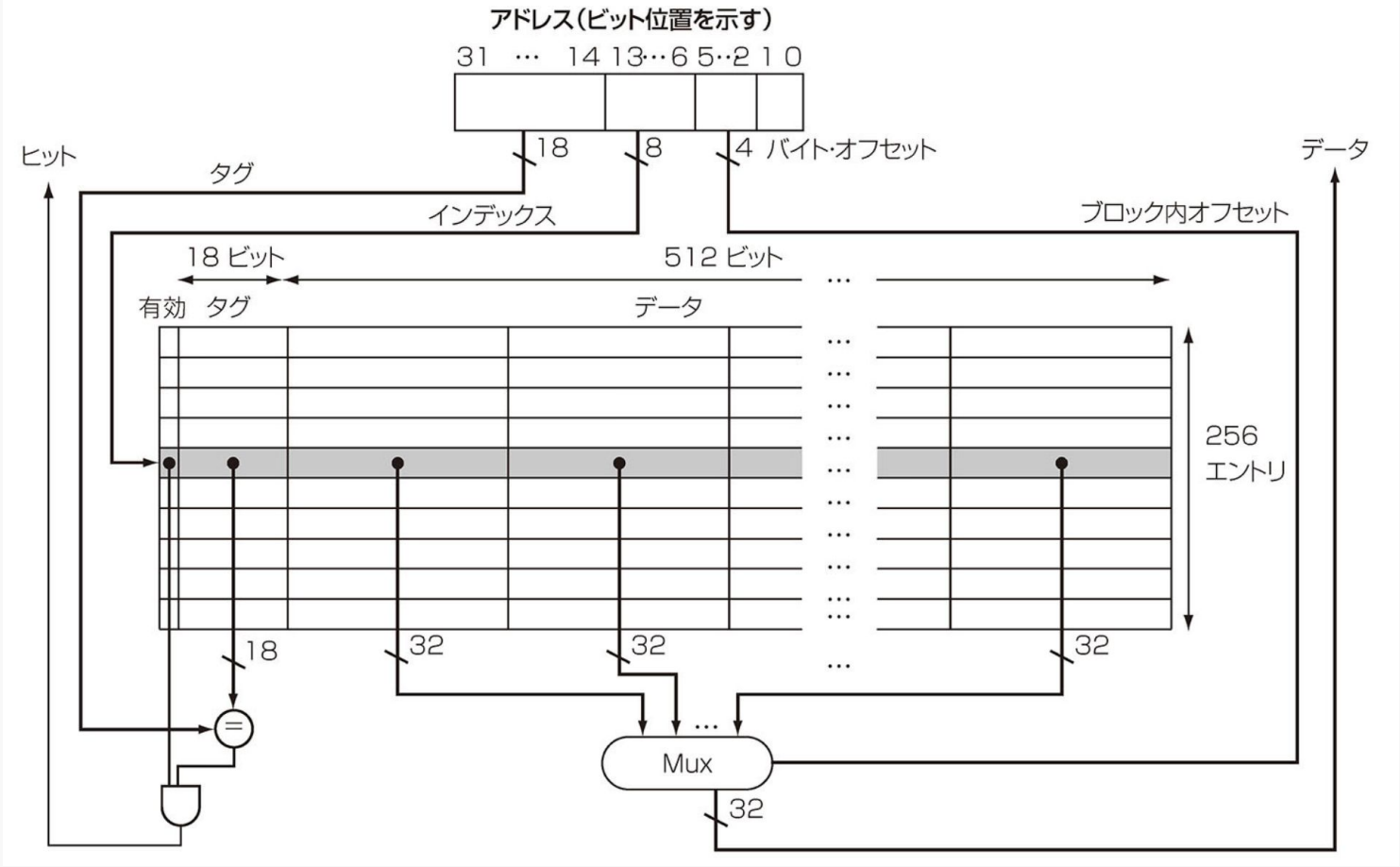
ライトバック（Write Back）

- 新しい値はキャッシュにのみ書き込まれる
- そのブロックが置き換え対象になってはじめて主記憶にも書き込まれる
- プロセッサからの書き込み要求が多い場合に有効
- ブロックを置き換える際にメモリアクセスを待つ時間をなくすため、**ライトバックバッファ**が用いられる
  - 原理はライトバッファと同じ
  - 置き換え対象のブロックをバッファに退避させ、新しいブロックをキャッシュに書く
  - バッファがその後主記憶への書き込みを行う

キャッシュミスの取り扱い

- 書き込み先がキャッシュに存在しなかった場合、、
  - Write allocate: 主記憶に書き込んだのちキャッシュにも書き込む
  - No write allocate: 主記憶にのみ書き込む
- 書き込むデータが再び使われる可能性があるかどうかによる
  - OSが主記憶中のページをゼロクリアする場合などはNo write allocateで問題ない

例：Intrinsity FastMATHのデータキャッシュ



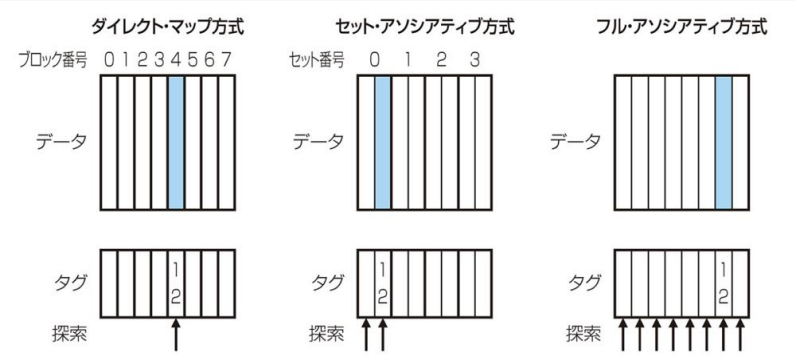
キャッシュの性能向上

工夫1: マッピングの工夫

- これまでダイレクトマップ方式でのマッピングでキャッシュを検討してきた
- ダイレクトマップはブロックの配置場所が一意に決まる一方、場合によってはあるロケーションで頻繁に置き換えが起こり、あるロケーションではずっと何も書き込まれないといった非効率な事態が起きる

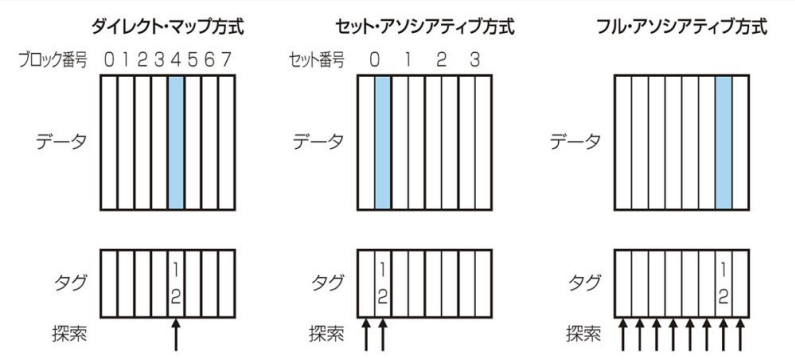
フルアソシアティブ方式（フル連想ともいう）

- ブロックをキャッシュのどこにおいても構わない
  - つまりタグが存在せず、キャッシュエントリの余る限りブロックを加えていく
- キャッシュ容量に対して無駄が少ない一方、ブロックを見つけるためにエントリを走査する必要がある
  - なのでブロック数が少ないキャッシュにおいて適用できる



セットアソシアティブ方式（セット連想ともいう）

- あるブロックを配置可能な場所がn箇所存在する
  - これをn-way セットアソシアティブと呼ぶ
- n個のブロックを配置可能な**セット**の連なりから構成される
- ダイレクトマップとフルアソシアティブの中間的な構成
  - すべてセットアソシアティブ方式と考えることもできる



アソシアティブ方式における置き換えブロックの選択

LRU法（Least Recently Used）

- セット内のブロックのうち参照されたのが最も古いブロックを置き換え
- 2-wayセットであれば1ビットで確実判断できる
  - セット内の要素数が増えるにつれて正確にLeast Recentを判断するのは難しくなる
  - 参照されたら参照ビットを立てて、置き換えの際に参照ビットが立っていないものを置き換えるという手法もある
- 他にもランダムにブロックを置き換えるランダム法も存在する

工夫2: マルチレベルキャッシュ

- いわゆるL1, L2など、キャッシュにおけるレベルわけ
- マルチレベルキャッシュにおいてL1キャッシュはクロックサイクルの短縮に専念できる
  - 通常単一レベルキャッシュと比べてL1キャッシュは容量が小さい
  - ミスペナルティの軽減はL2キャッシュが受け持つ

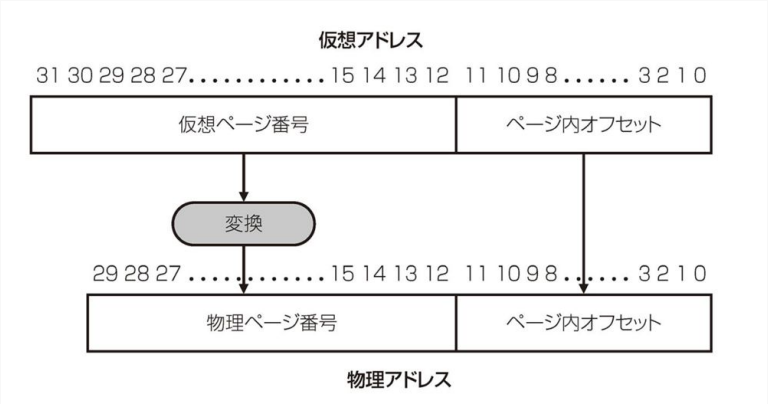
仮想記憶

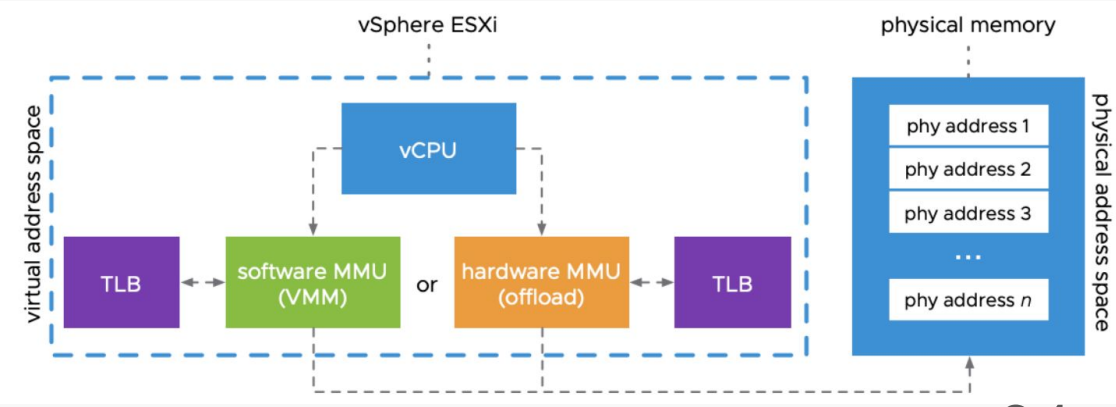
仮想記憶とは？

- ディスクなどの二次記憶を使って主記憶容量を拡張する手法
  - 元はタイムシェア型のコンピュータにおけるリソースの共有問題から考案された
  - 現代では特に仮想マシンにおけるリソースの共有に際して用いられる
- 仮想記憶は概念的にはキャッシュととても似ている
  - キャッシュにおけるブロック：仮想記憶における「ページ」
  - キャッシュにおけるミス：仮想記憶における「ページフォルト」

仮想記憶の仕組み

- **仮想アドレス**を用いて仮想的なメモリアドレスを定義する
- 仮想アドレスはハードウェアもしくはソフトウェアのMMUによって物理アドレスに変換され、実際にメモリアクセスが発生する





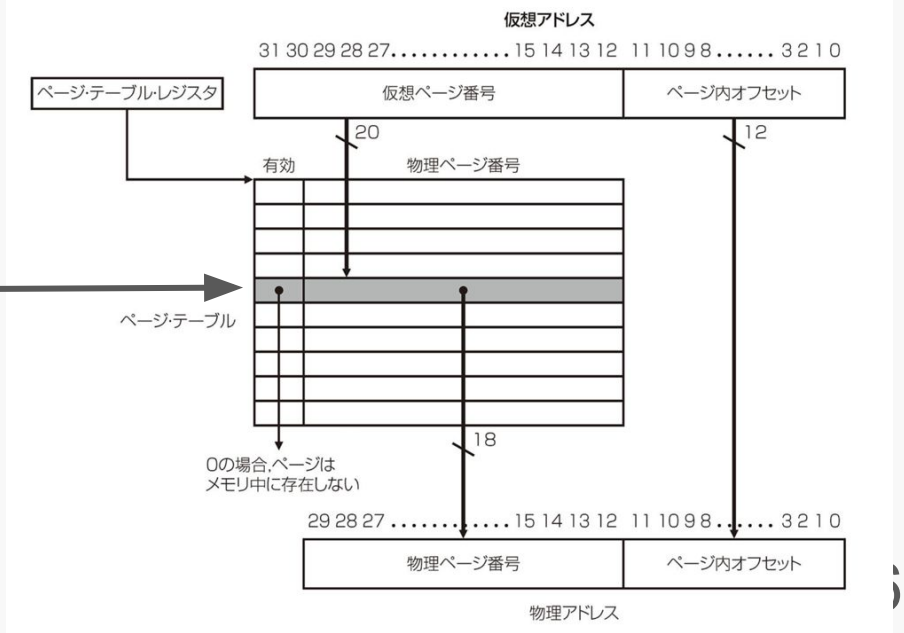
ページフォルト

- メモリアクセス時に主記憶にデータが存在しない場合
- ページフォルトが発生すると二次記憶にアクセスが発生する=激遅
  - 例外機構によってOSに制御が移り、二次記憶へのアクセスが発生
  - フラッシュメモリのアクセス時間はDRAMの100倍ほど
- 対策としてページサイズを大きくする方法がある（大きくしすぎるのはNG）
- 主記憶内にページをフルアソシアティブで配置する
  - 任意の場所に配置でき置き換えが減る
  - でも全数検索は遅いのでは。。。

ページテーブル

- 主記憶に位置する、仮想アドレス物理アドレスをインデックス付けした表
- **ページテーブルレジスタ**にページテーブルの始点が記されており、仮想アドレスがテーブルにおけるインデックスの役割を果たす
    - キャッシュではないのでタグは必要ない

有効ビットによって主記憶中にデータがあるかを示す



ページテーブル

- 主記憶のページテーブルの他に、二次記憶に対するページテーブルも持つ
  - 論理的には2つのテーブルは単一のテーブルとして考えられる
  - 実際にはディスク上に存在するページのアドレスは全て保持する必要があるので別のデータ構造に分けて保持する場合が多い

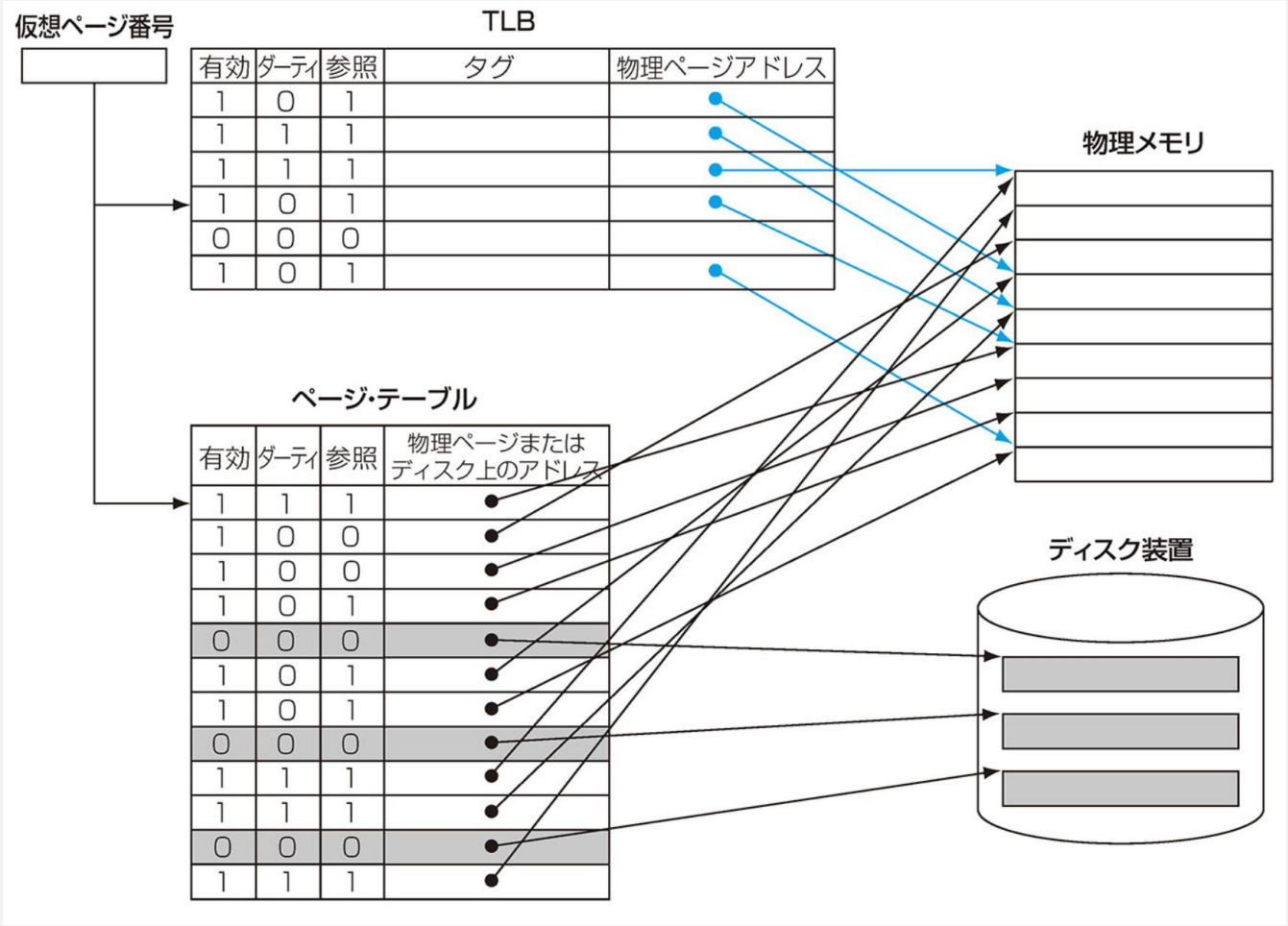
仮想記憶における書き込みと置き換え

- ライトバック方式を取らざるを得ない
- 置き換え時に無条件に書き戻すのではなく、**dirty bit**を参照する
  - ページに書き込んだ際にdirty bitを立てる
  - OSがそのページを置き換えるときにdirty bitが立っていれば二次記憶に書き込みを行う
- 置き換えについては厳密にLRU法を適用するのはコストが高い
  - メモリ参照のたびにデータ構造を更新するのはしんどい
  - ページがアクセスされるたびに参照ビットを立て、それを基に置き換え対象を決める
  - 参照ビットを用いない場合はソフトウェアのアルゴリズムに頼る

アドレス変換の高速化

TLB（Translation-Lookaside Buffer）

- ページテーブルのキャッシュ
- 通常のキャッシュと同じように仮想ページ番号の一部をタグとして保持
- ページテーブルの代わりにTLBにアクセスするので参照ビットやダーティビットも含める必要がある
- TLB→主記憶のページテーブル→二次記憶のページテーブルの順に参照して物理アドレスを得る

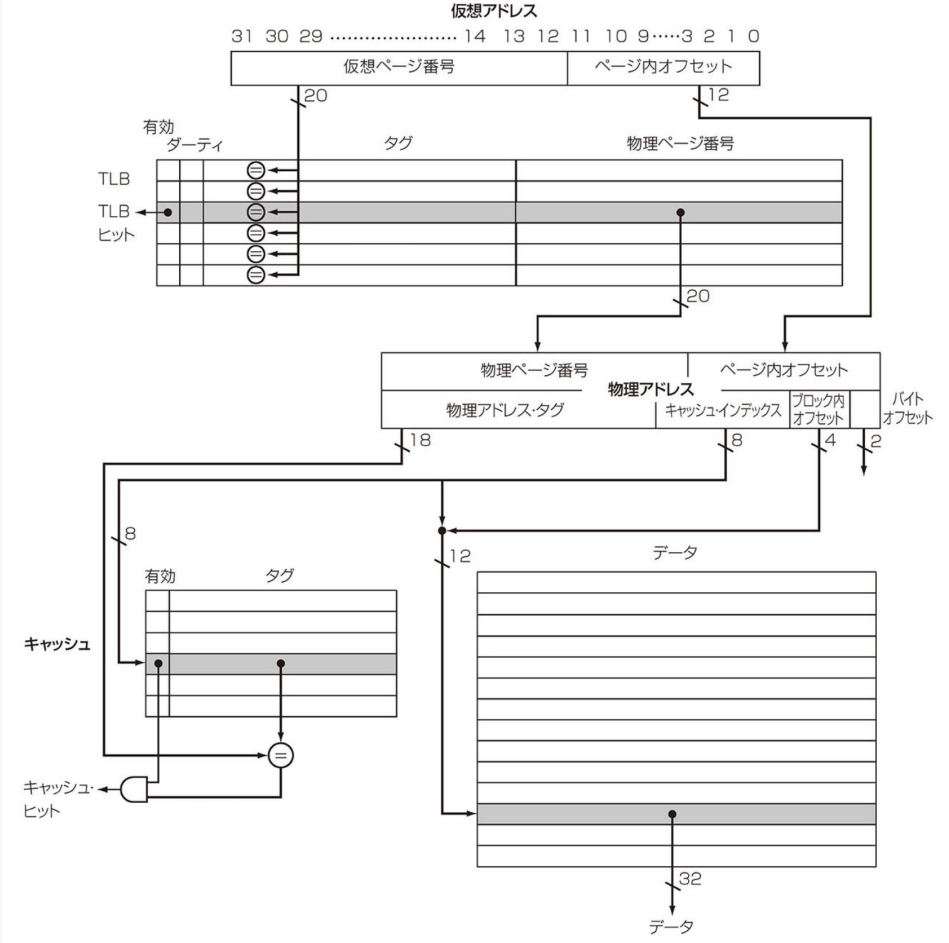

仮想ページ番号
TLB
有効
ダーティ
参照
タグ
物理ページアドレス
物理メモリ
ページ・テーブル
有効
ダーティ
参照
物理ページまたは
ディスク上のアドレス
ディスク装置

TLBのサイズ感

TLB 容量	16 ～ 512 エントリ
ブロック・サイズ	1 ～ 2 ページ·テーブル·エントリ（通常それぞれ 4 ～ 8 バイト）
ヒット時間	0.5 ～ 1 クロック・サイクル
ミス・ペナルティ	10 ～ 100 クロック・サイクル
ミス率	0.01%～ 1%

- ミス率を下げるためにフルアソシアティブ方式の小さなTLBを用いるケース
- TLBサイズを大きく取り、連想度の小さいセットアソシアティブ方式を採用するケース
    - 置き換えについてはランダムに選択するコンピュータが多い

Intrinsity FastMATHのTLBとキャッシュ構成



マルチプロセッサとキャッシュ

マルチプロセッサのキャッシュにおける一貫性の問題

1. CPU AがブロックXを読み出す -> AのキャッシュにXが入る
2. CPU BがブロックXを読み出す -> BのキャッシュにXが入る
3. CPU AがブロックXに書き込む -> Aのキャッシュは更新される

次にCPU BがブロックXを読み出す時、Bのキャッシュは古いままなので古いXの値を読み出してしまう

誤り訂正と誤り検出

ハミングの誤り検出

- あるビットパターンにおける1の数を数え、奇数だったら1、偶数だったら0を末尾に加える
  - この加えるビットをパリティビットという
  - パリティビットを付加された後のビット列の1の数は偶数になるはず
- 通信時やディスクへの書き込み時に1ビット誤りを検出できる
  - 厳密には「奇数ビットの誤りがあることを検出できる」だが、確率的には1ビットの誤り

ハミング誤り訂正コード（ECC）

- 誤りを検出するだけでなく誤りを訂正したい
- パリティビットをさらに増やしてこれを実現する
- ビットパターンの左から順に番号をふり、2の冪乗である全てのビットをパリティビットにし、それ以外をデータビットに使用する

ECC（10011010での例）

ビット位置		1	2	3	4	5	6	7	8	9	10	11	12
符号化されたデータ・ビット		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
パリティ・ビットの対象範囲	p1	×		×		×		×		×		×	
	p2		×	×			×	×			×	×	
	p4				×	×	×	×					×
	p8								×	×	×	×	×

- 8ビットの値10011010で考える
- この場合パリティビットは4ビット
  - p1は最下位ビットが1のデータビットをチェックする（0001, 0011, 0101, 0111, 1001…）
  - p2は最下位から2番目のビットが1のデータビットをチェックする（0010, 0011, 0110…）
  - p4は最下位から3番目のビットが1のデータビットをチェックする（0100, 0101, 0110…）
  - …

ECC（10011010での例）

ビット位置		1	2	3	4	5	6	7	8	9	10	11	12
符号化されたデータ・ビット		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
パリティ・ビットの対象範囲	p1	×		×		×		×		×		×	
	p2		×	×			×	×			×	×	
	p4				×	×	×	×					×
	p8								×	×	×	×	×

- それぞれのパリティビットの値を見てみる
  - p1はp1, d1, d2, d4, d5, d7をチェックするため0
  - p2はp2, d1, d3, d4, d6, d7をチェックするため1
  - p4はp4, d2, d3, d4, d8をチェックするため1
  - p8はp8, d5, d6, d7, d8をチェックするため0

ECC（10011010での例）

ビット位置		1	2	3	4	5	6	7	8	9	10	11	12
符号化されたデータ・ビット		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
パリティ・ビットの対象範囲	p1	×		×		×		×		×		×	
	p2		×	×			×	×			×	×	
	p4				×	×	×	×					×
	p8								×	×	×	×	×

- パリティビットを付加したビット列は011100101010

ECC（10011010での例）（最終的なビット列：011100101010）

ビット位置		1	2	3	4	5	6	7	8	9	10	11	12
符号化されたデータ・ビット		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
パリティ・ビットの対象範囲	p1	×		×		×		×		×		×	
	p2		×	×			×	×			×	×	
	p4				×	×	×	×					×
	p8								×	×	×	×	×

- ここでd6が反転してしまったとする -> 011100101**1**10
- d6をチェックしているのはp2, p8であり、p2, p8に不整合が生じることでd6に誤りがあることが判明する
- 誤りが判明したらビットを反転させれば良いので誤り訂正が可能になる

SEC/DED（Single Error Correction / Double Error Detection）

- 上述のパリティビットに加えて、パリティビットを加えた後のビット列に対するパリティビットをさらに末尾に加えることで2ビットの誤り検出ができる
- これらを組み合わせると1ビットの誤り訂正と2ビットの誤り検出が可能になる
- 今日のほとんどのサーバーのメモリではこのSEC/DEDが実装されている

以上

（キャッシュとマルチプロセッサの節はまた今度）