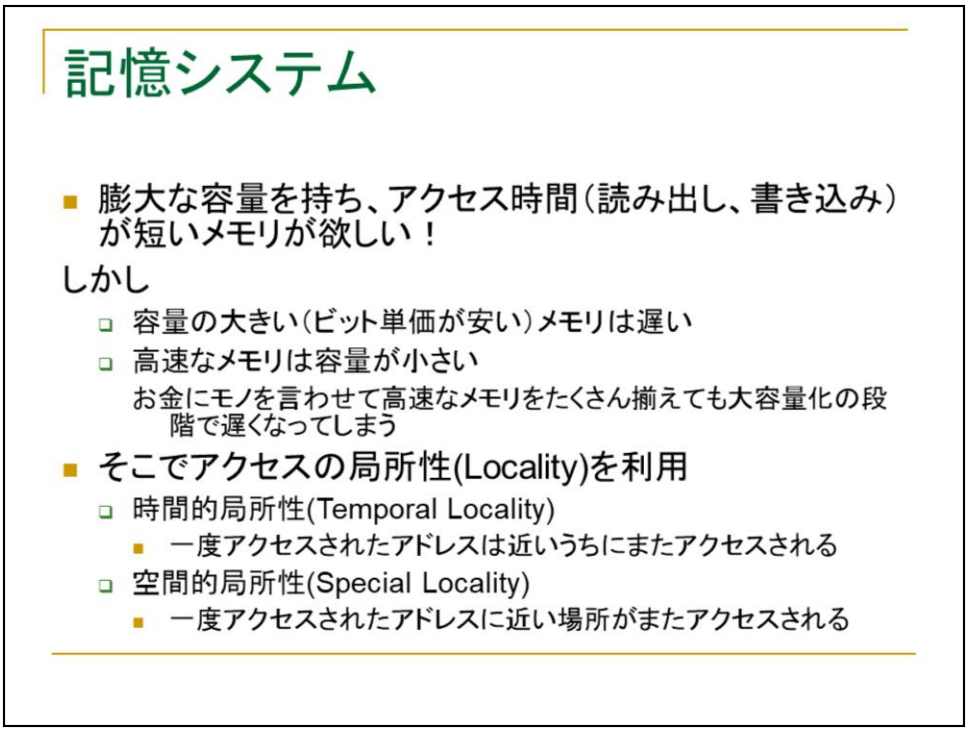
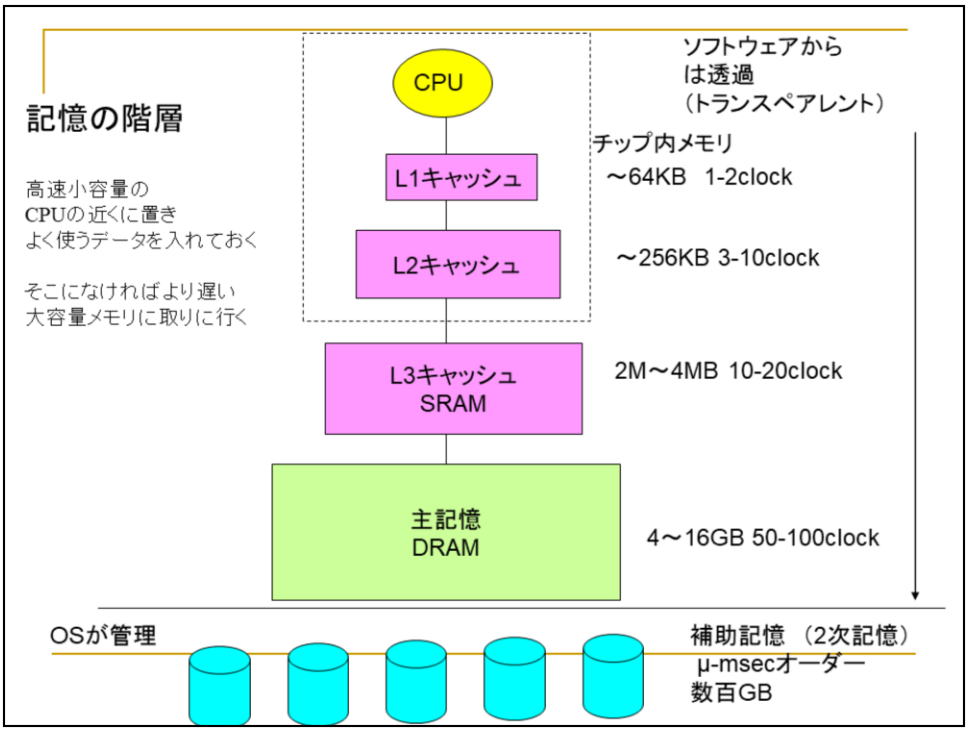
記憶の階層とキャッシュ

天野英晴

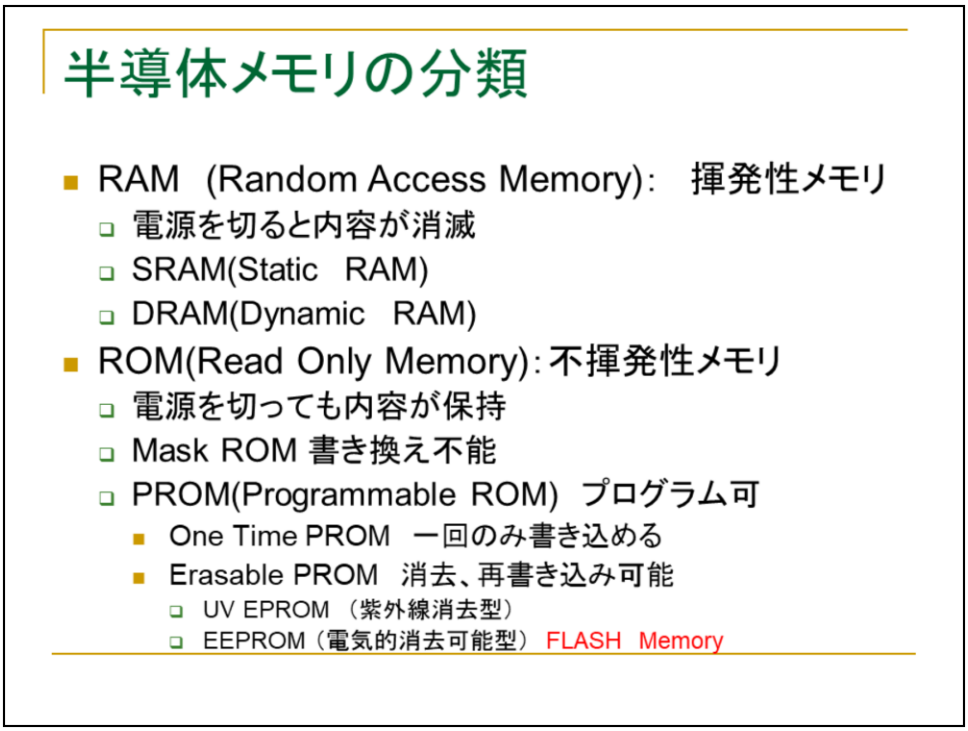
今まではCPUの設計に注力して来ました。これからしばらくコンピュータの記憶システムについて紹介します。



コンピュータの記憶システムに必要なことは容量とアクセスの高速性です。簡単に言えば、膨大な記憶容量を持っていて、瞬時に読み書きできるメモリが理想のメモリです。しかし、残念なことに容量の大きい、すなわちビット単価の安いメモリは遅く、高速なメモリほど容量が小さいです。ではお金が無限にあるとして、高速なメモリを山ほどそろえれば、大容量で高速なメモリができるのでしょうか？これは実際にはうまく行きません。高速なメモリをたくさんならべて一つのメモリにする段階で遅くなってしまうのです。それよりもっと良い方法は、アクセスの局所性（Locality)を用いる方法です。アクセス局所性には時間的局所性（Temporal Locality）と、空間的局所性（Special Locality)があります。時間的局所性は一度アクセスされたアドレスは近いうちにまたアクセスされるというもので、空間的局所性は一度アクセスされたアドレスに近い場所が再びアクセスされるというものです。両者は関係を持ちつつ微妙に違っています。



時間的、空間的な局所性がある場合、頻繁にアクセスされるデータは近くのアドレスに固まっているはずです。これを利用して、高速小容量のメモリであるほど、CPUの近くに置き、低速大容量のメモリであるほど、CPUの遠くに置き、階層構造を作ります。良く使うデータ（命令も）を入れておく高速小容量のメモリのことをキャッシュと呼びます。キャッシュ上にあれば、CPUは高速にデータを読み書きできます。キャッシュ上になければ、次のレベルのメモリに取りに行きます。最もCPUに近いキャッシュはCPUと同じチップ内に置かれており、L1キャッシュと呼びます。CPUはできれば1クロックで、ダメでも数クロックでL1キャッシュにアクセスします。これにはずれたら、次のレベルであるL2キャッシュに取りに行きます。最近のCPUはL2,L3までチップ内に入れておく場合が多いです。これにはずれると次のレベルはオンボードキャッシュです。これはボード上のSRAM（Static RAM)を使います。これにはずれると、主記憶にデータを取りに行きます。主記憶にはDRAM(Dynamic RAM)が使われます。ここまでは、ソフトウェアには見えない（トランスペアレント）構造になっています。主記憶中にデータがない場合、補助記憶に取りに行きますが、これはOSが管理するのが普通です。補助記憶は伝統的にディスクが使われていますが、最近はNAND型のフラッシュメモリも増えています。



メモリシステムを理解するためには、使われるメモリの性質を理解することが必須です。しかし、これはこの授業の範囲ではなく、電子回路基礎で紹介しています。しかし履修していない人も居ると思いますし、要点だけ復習しておきましょう。

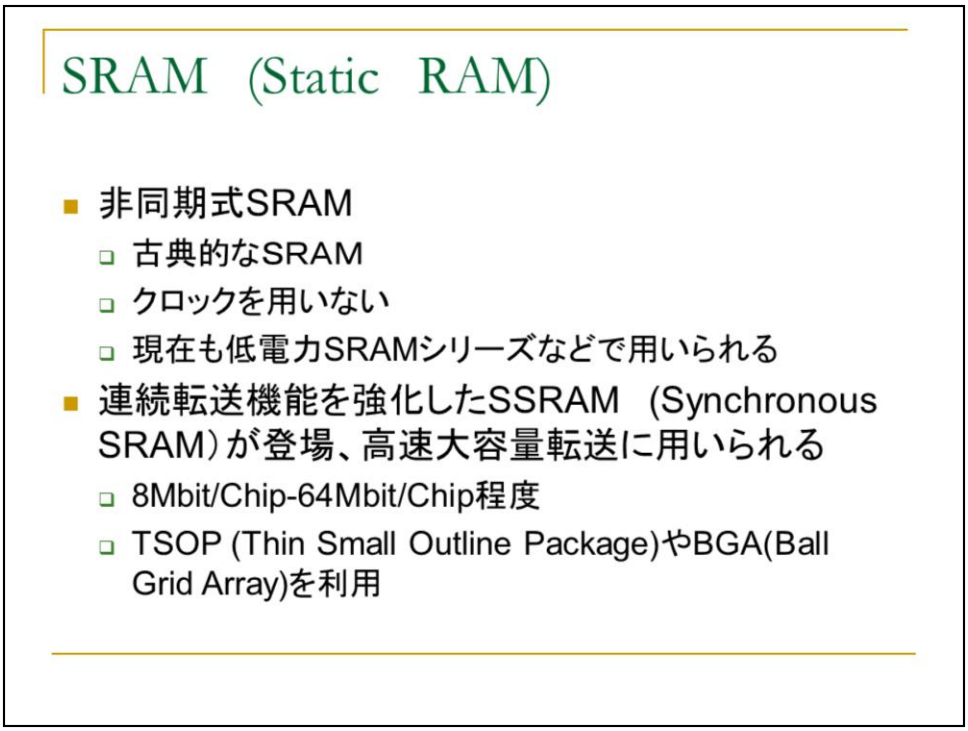
半導体メモリは、RAMとROMに分類されます。RAMとROMは本来の意味とはかなり違った使い方をされています。RAMはRandom Access Memoryの略で、アドレスに関わらずアクセスの方法と時間が同じものを指します。ROMはRead Only Memoryの略で読み出し専用メモリの意味です。しかし、最近ではRAMは揮発性メモリ、つまり電源を切るとデータが消えてしまうメモリ、ROMは不揮発性メモリ、すなわち電源を切ってもデータが消えないメモリの意味に使われます。

RAMの容量

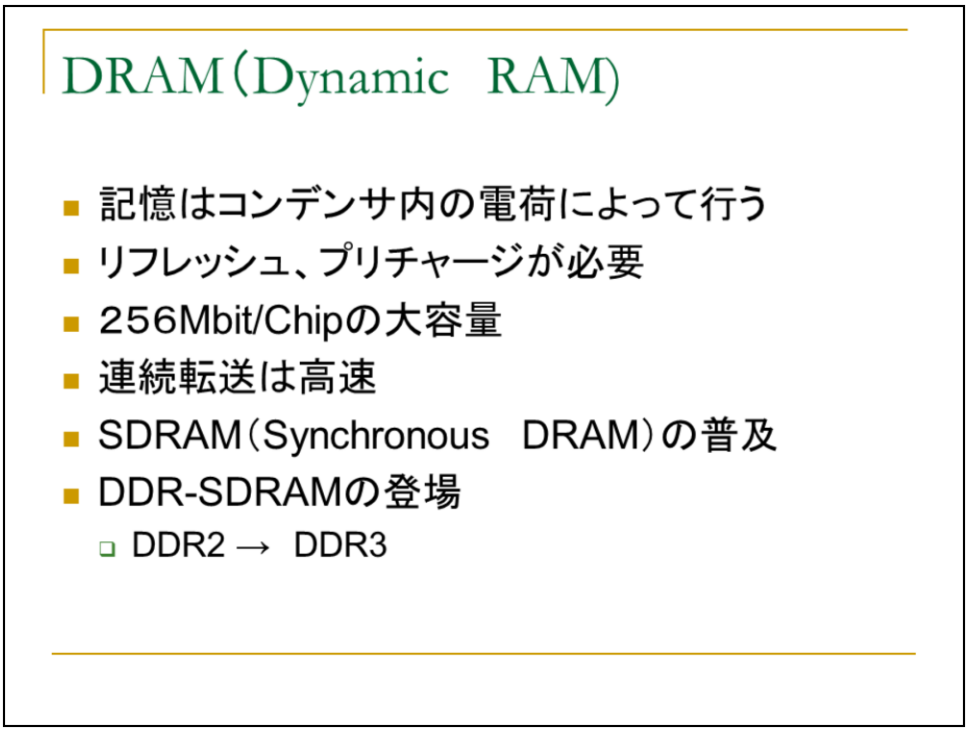
- 深さ×幅
- 右の表に幅を掛ければ全体の容量が出る
- 省略した言い方でも十分（端数を覚えている人は少ない）

アドレス本数	容量	省略した言い方
8	256	256
10	1024	1K
12	4096	4K
16	65536	64K
18	262144	256K
20	1048576	1M
24	16777216	16M
28	26835456	256M
30	1073741824	1G
32	4204067296	4G

RAMの容量は、アドレスの本数をn、一つのアドレスに保持できるデータの幅をwとすると2のn乗×wになります。wはたいてい1、2、4，8，16など2のk乗になるので、全体は2の階乗になります。メモリの容量は膨大なので、皆さんはこれに慣れる必要があります。2の10乗が1K、2の20乗が1M、2の30乗が1Gというのだけは、覚え易いのでぜひ覚えてください。



ではまずSRAMすなわちStatic RAMを紹介しましょう。古典的なSRAMはクロックを持たない非同期式で、現在でも低電力用に使われています。一方、コンピュータのキャッシュメモリ（計算機構成で紹介します）など、高速読み書きが必要な用途には連続転送機能を強化した同期式SRAM（SSRAM)が用いられます。チップ当たり8Mbitから64Mbit程度までを格納することができ、基板の表面に高密度実装するため、TSOPやBGAなどのパッケージに入っています。

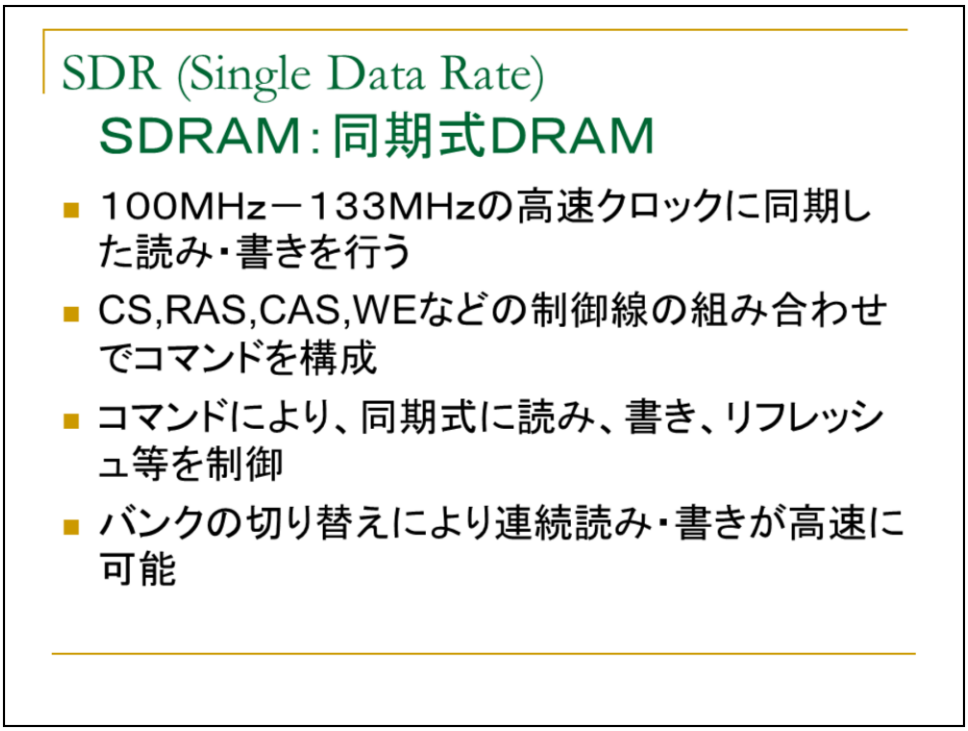


では、次にDRAMすなわちDynamic RAMを紹介します。ラッチの状態で記憶を行うSRAMに対してDRAMは半導体内部のコンデンサ内に電荷が蓄えられているかどうかによって情報を記憶します。コンデンサの中の電荷を扱うため、一定の間隔で充電をしなおすリフレッシュ、比較用コンデンサを充電するプリチャージなどが必要で、使い難いです。その代わりチップ当たりの容量はSRAMのほぼ4倍あり、大容量の記憶が可能です。最近は同期型DRAMの普及により、連続転送は高速に行うことができるようになりました。

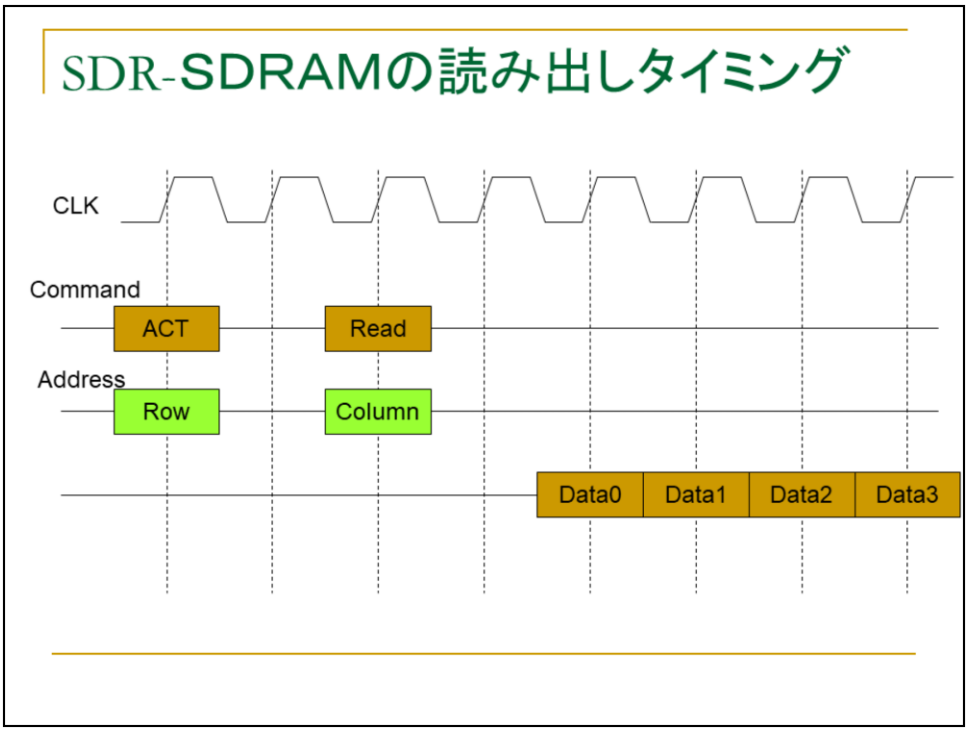
DDR-SDRAMカードの例

- 下は1GBでやや小さい。今は4GB－8GBのカードが良く使われる

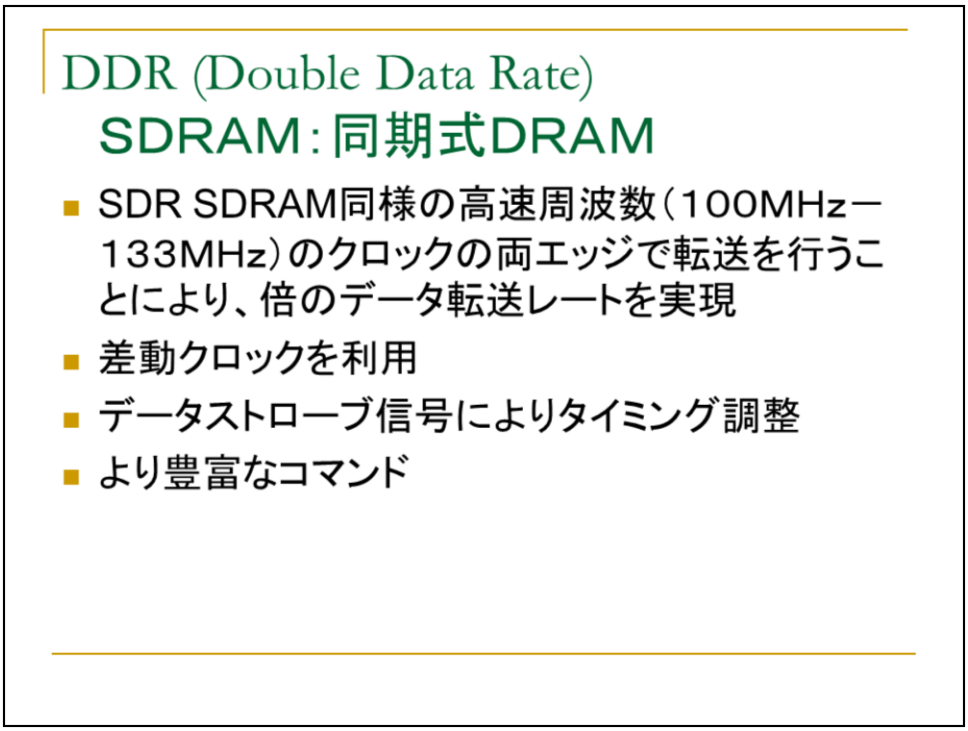
すなわち、DRAMはカードの形で売られます。この図は1Gバイトのカードの一例です。最近は1つのカード上に4Gバイトから8Gバイトの容量が搭載されています。



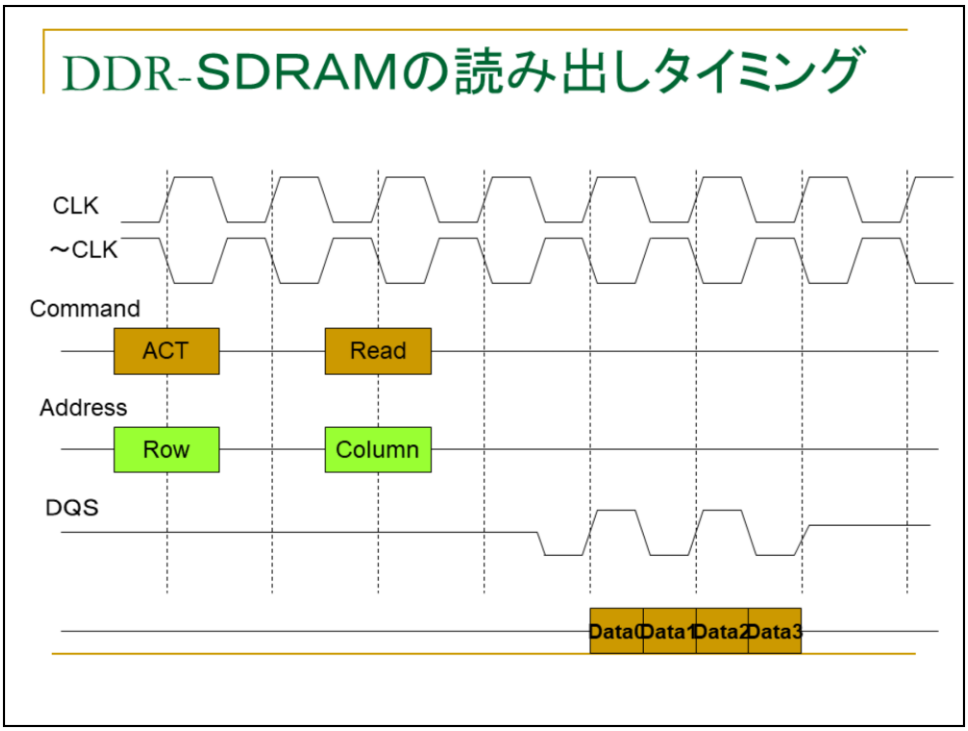
DRAMはコンピュータの主記憶として使われますので、キャッシュとの間で高速なブロック転送能力が必要とされます。DRAMは一行読んでくるのは時間が掛かりますが、読んできた行内で連続してデータを転送するならば高速に行うことができます。チップ内部に何個か独立したブロックを設けておけば、とぎれなく連続データを供給することができます。このためには、転送用のクロックを設けてこれに同期して転送するのが適しています。そこで、クロックに同期して転送を行う同期式DRAMというのが表れました。同期式DRAMでは、今までのDRAMのCS,RAS,CAS,WEなどの制御端子はセットとしてコマンドとして与えるようにしました。これが同期式DRAM、Synchronous DRAM(SDRAM)です。



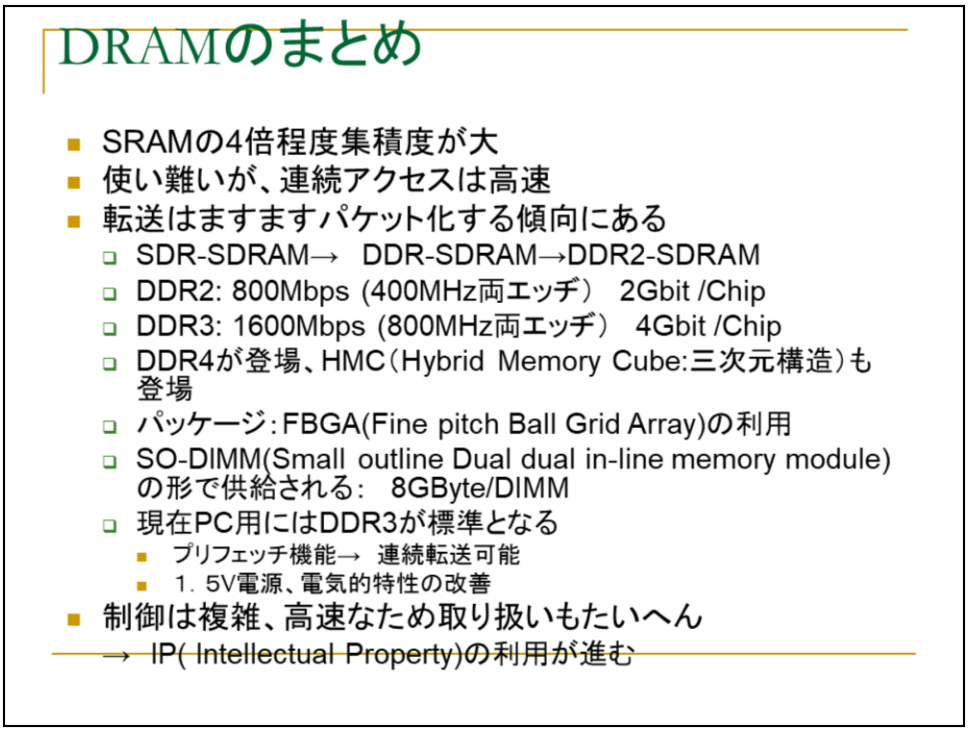
初期のSDRAMのタイミングチャートを示します。クロックに同期してACT(Activation)コマンドを与えると同時に行アドレスを与えます。次にReadコマンドと共に列アドレスを与え、1クロック置いてデータが順番に読み出されます。最初のデータが1個読み出されるまでは時間が掛かりますが、一度データ転送が始まれば次々とデータを送ることができます。



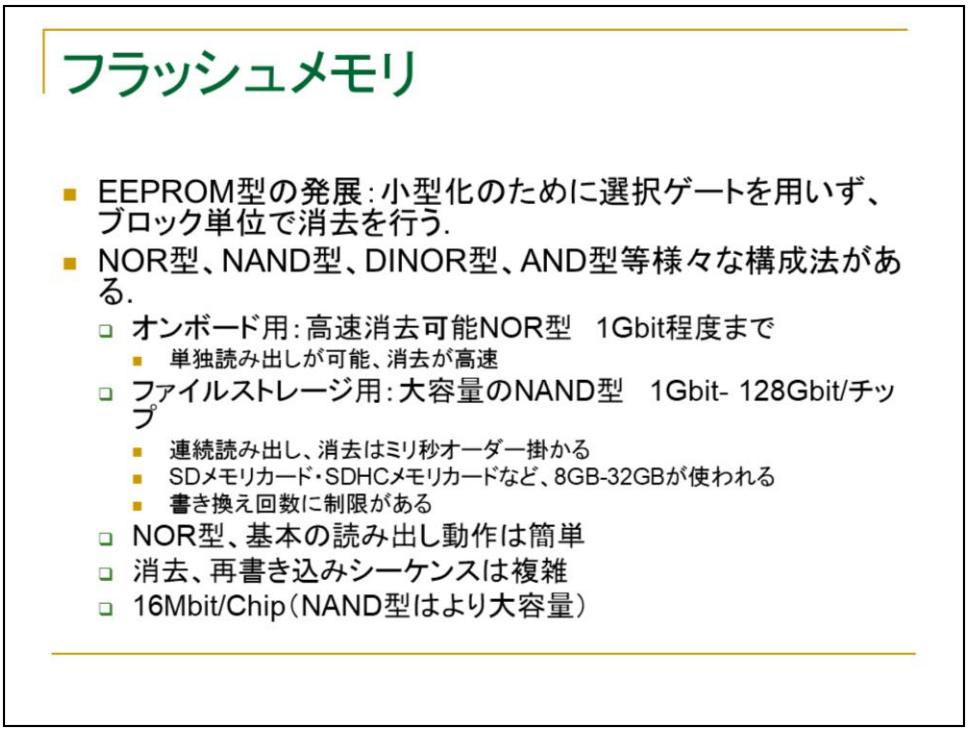
コンピュータのCPU（中央処理装置）の性能向上はとどまることを知らず、SDRAMの転送性能もすぐに足りなくなりました。このため、クロックの立ち上がり、立下りの両方のエッジを使って倍の転送レートを実現する方法が登場しました。これは、Double Data Rate (DDR)　SDRAMと呼びます。



DDR-SDRAMは、コマンドとアドレスの与え方はSDRとほぼ同じ（コマンドは種類が増えています）ですが、クロックとクロックの反転の両方を与えます（差動クロックと呼びます）。クロック周波数も上がり、転送能力は大幅に向上しました。



では、DRAMをまとめましょう。クロックの両エッジを使ったDDR－SDRAMは、よりクロック周波数を高めて動作電圧を下げたDDR2に置き換わり、さらにDDR-3が現在もっとも良く使われます。これは800MHzの両エッジでデータの転送を行います。さらにこの上の版であるDDR-4が登場しています。一方、DRAMチップを三次元的に積層したHMC（Hybrid Memory Cube)も登場し、今後どのような方式がメジャーになるか目が離せないところです。この辺は、様々なコンピュータに搭載できなければならないので、「標準化」が重要です。基本的な動作原理はDDR2-4は同じなのですが、電気的な仕様や動作周波数が違ってくるのです。ちなみに、このような高速のDRAMに接続を行う制御回路を作るのは大変で、ここには以前紹介したIPを使います。DRAMに代わる新しい記憶素子としてFeRAMやMRAMなどが開発されていますが、まだ広く使われるには至っていません。今後しばらくはDRAMの重要性は落ちることはないようです。

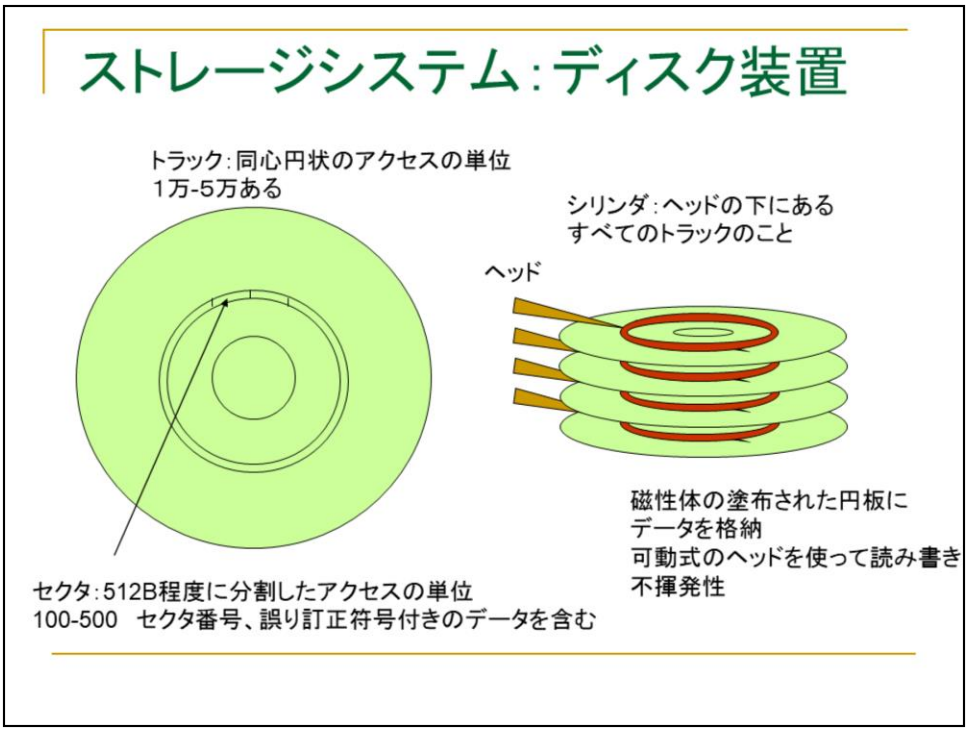


電気的消去可能なメモリの中に、小型化を行うため、選択ゲートを用いずブロック単位の消去を行うのがフラッシュメモリです。フラッシュメモリには現在には様々な方法があり、大きくNOR型、NAND型に分かれています。NOR型は高速消去可能で、単独データの読み出しが可能で、消去も高速です。読み出しはほとんどSRAMと同じように使うことができます。これはボード上に搭載して電源を切っても消えないデータ（FPGAの構成情報など、来週やります）を保存しておくために使います。一方で、NAND型は、連続読み出しになり、消去はミリ秒近く掛かります。しかし容量は大きく、SDメモリカード、SDHCメモリカードなど大量のデータを蓄えておくときに使います。みなさんが最も良く使うのはこのメモリだと思います。

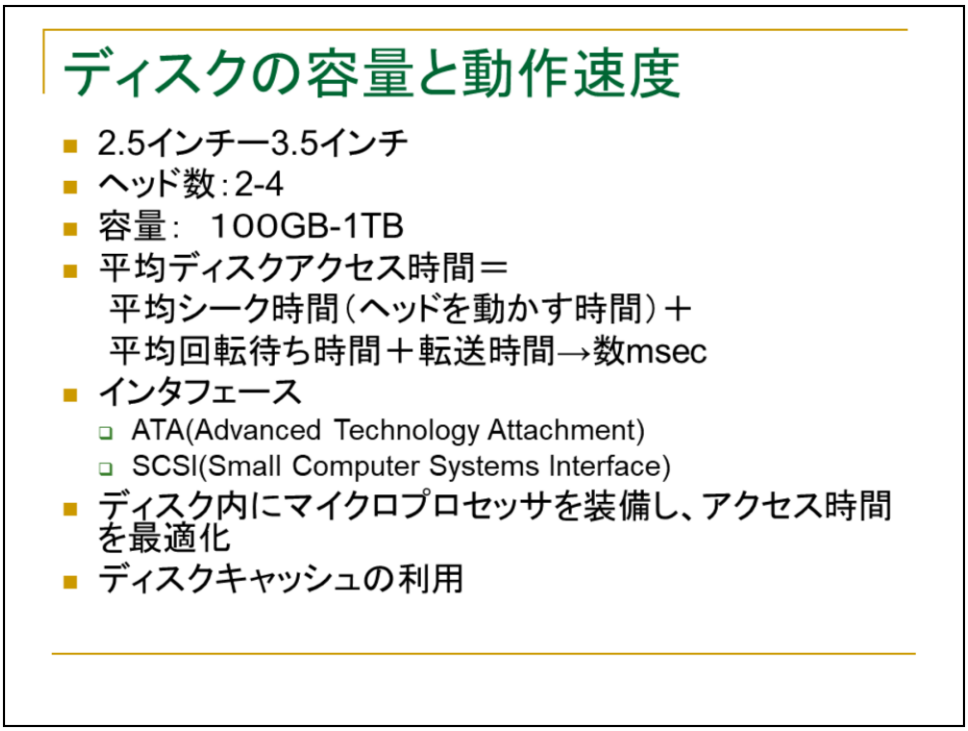
SSD(Solid State Drive)

- フラッシュメモリを利用した記憶システム（ストレージ）
- 大きいのは6.4TBの容量を持つ
- インタフェース
  - SATA(Serial ATA)
    - コスパに優れる
  - SAS(Serial Attached SCSI)
    - 性能、信頼性が高い
  - PCIe (PCI express)
    - PC用の汎用バス
  - NVMe(Non-Voratile Memory express)
    - PCIeを通じてSSDを繋ぐためのインタフェース

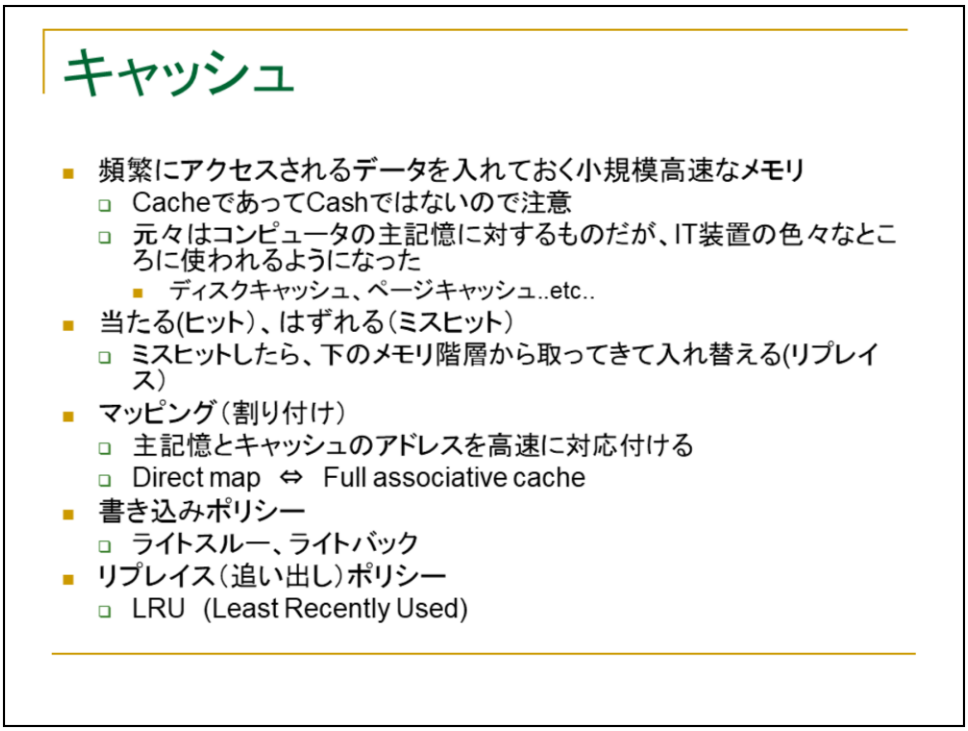
NAND型のフラッシュメモリのうちの大容量なものを使って作ったコンピュータ用の記憶装置をSSDと呼び次のページのHDD（Hard Disk Drive)と区別します。SSDはHDDに比べて小型でカード状になっており、高速です。PCやサーバーに接続して補助記憶として使います。HDDに従来から用いられたATAやSCSIのバスに接続するものと、PCIeに直接接続するもの、拡張用のバスNVMeを使うものがあります。



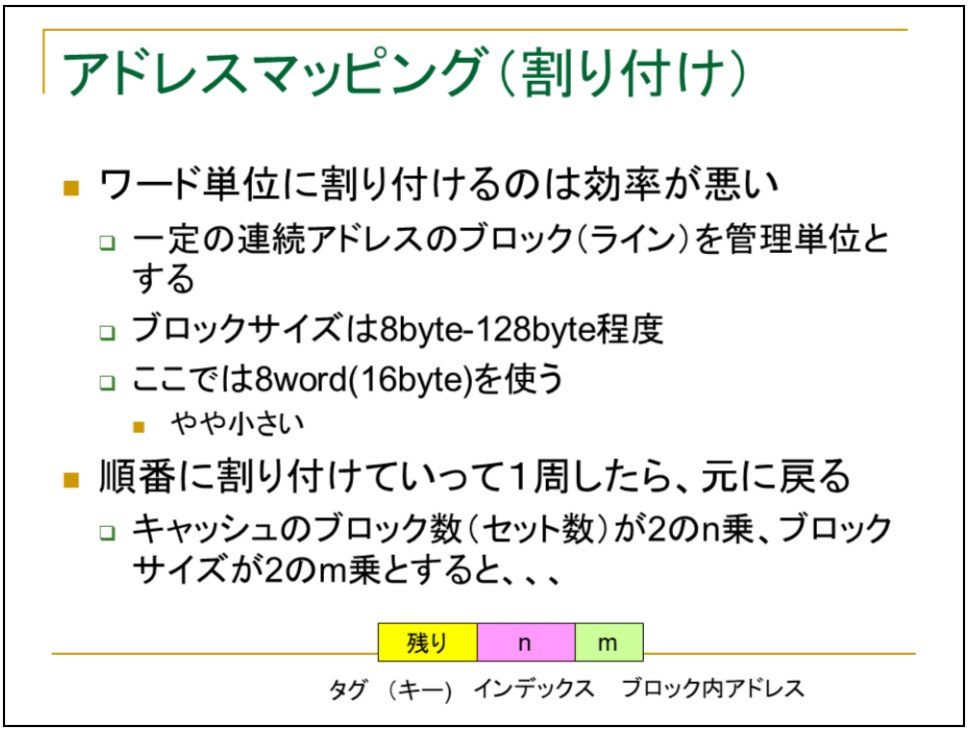
最近はSSDに押されてきているとはいえ、まだまだビッグデータ記憶の主力として活躍しているのがディスク装置です。ディスクは、磁気を塗布した円板で、表面上に磁気の形でデータを蓄えます。すなわち、ディスクは、電気的記憶媒体ではなく、磁気的記憶媒体です。データは一万から5万ある円周上に蓄えられ、この一周をトラックと呼びます。（陸上競技と同じです。）トラックは、100～500個程度の、512Bに分割したアクセスの単位に分割されます。これをセクタと呼びます。各セクタにはセクタ番号と誤り訂正符号付きのデータが含まれて居ます。ディスクは磁気的な記憶なので、読み書きするためには、磁気、電気変換を行う必要があります。これを行うのがヘッドです。ヘッドを伸び縮みさせ、円板を回転させることにより、ヘッドを読み書きしようとするデータの上に持ってきて、データを読んだり書いたりします。容量を大きくし、性能を上げるため、複数の円板を同軸上に回して複数のヘッダでアクセスする方法が一般的です。一時期にヘッドの下にある全てのトラックをまとめてシリンダと呼びます。100GB-1TBという大量のデータを記憶できますが、読み書きの時間は遅く、部分的に壊れ易い問題点があります。



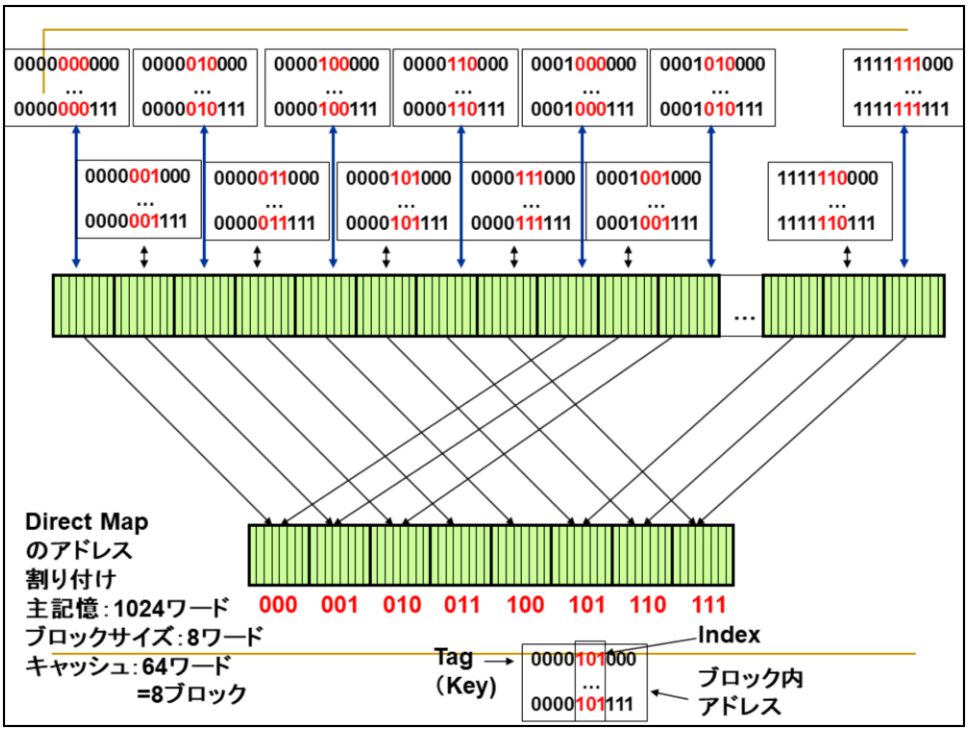
ディスク技術は古い歴史を持ちますが、まだまだ発達を止めていません。円板の直径は2.5から3.5インチで、ヘッド数は2から4が一般的です。容量も100GBから数TBに及びます。平均ディスクアクセス時間は、平均シーク時間（ヘッドを動かす時間）＋平均回転待ち時間＋転送時間となり、数ミリ秒くらいになります。ただし、最近のディスクはこの時間を減らすように、良く用いるデータを蓄えておくなど色々工夫しています。このためアクセス時間は時と場合に依るようになっています。ディスクをコンピュータに繋ぐには、ATAやSCSIなどと呼ばれるバスで繋ぎます。



ではメモリの基本がわかったところでキャッシュの話をしましょう。キャッシュとは頻繁にアクセスされるデータ(命令もデータの一種と考える)を入れておく小規模高速なメモリを指します。小銭のCashではなく、Cache(貴重なものを入れておく小物入れ)なのでご注意ください。この言葉はコンピュータの世界で大変有名になったので、IT機器の色々なところで使われるようになりました。ディスクキャッシュやページキャッシュとかがこの例です。キャッシュ上にデータが存在する場合は、ヒットと呼び、はずれるとミスヒット(ミス)と呼びます。ミスヒットしたら、下のメモリ階層から持ってきて入れ替えます。この処理をリプレイスと呼びます。キャッシュを理解するには三つのポイントがあります。一つはマッピングです。主記憶とキャッシュのアドレスを高速に対応付ける方法です。二つ目は書き込みポリシー、三つ目はリプレイスポリシーです。これを順に紹介しましょう。

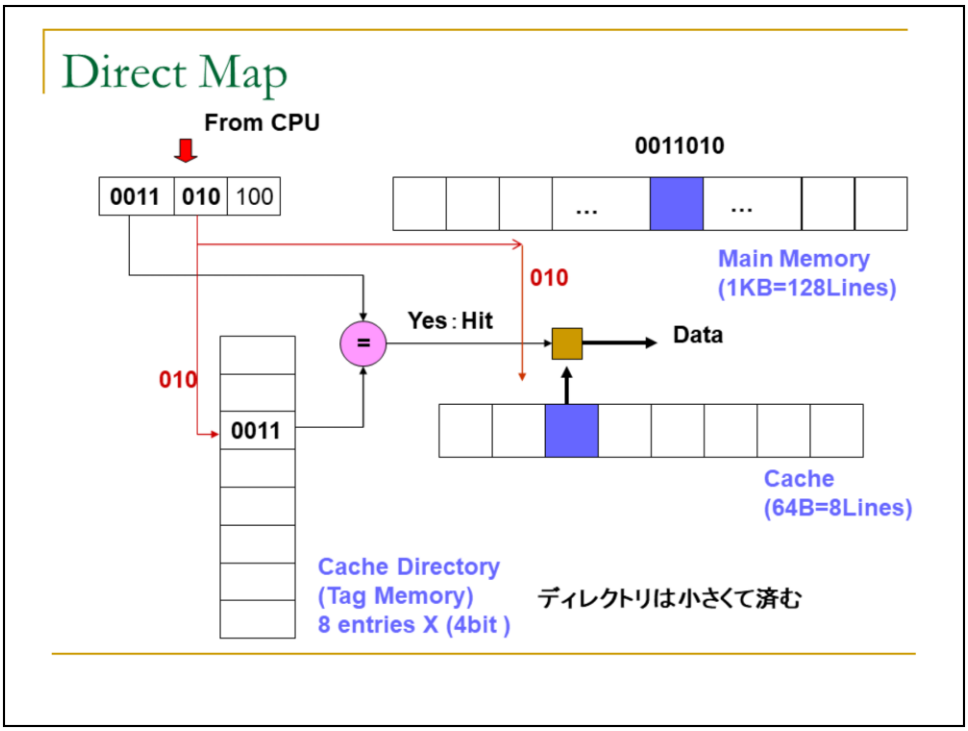


CPUからアドレスが出力されます。これを見て、キャッシュ上にあるのかないのか？あるならばどこにあるのか？を高速に判定して読み書きを行う必要があります。このため、メモリ上のデータをキャッシュ上のどこに置くか、その割付を行わなければならないです。まず、ワード単位に割り付けるのは効率が悪いので、一定の連続アドレスをブロック（ライン）として管理の単位とします。このブロックサイズは8バイトから128バイト程度です。ここでは8ワード（16バイト）を使います。ブロック単位で扱うことで、局所性の利点を生かして管理コストを下げることができます。では、このブロックをどのように割り付ければ良いでしょうか？基本的に非常な考え方を使います。順番にブロックを割り付けて行って一周したら元に戻ります。ちなみに、以後、メモリもキャッシュも全ては2のx乗の世界であると考えます。

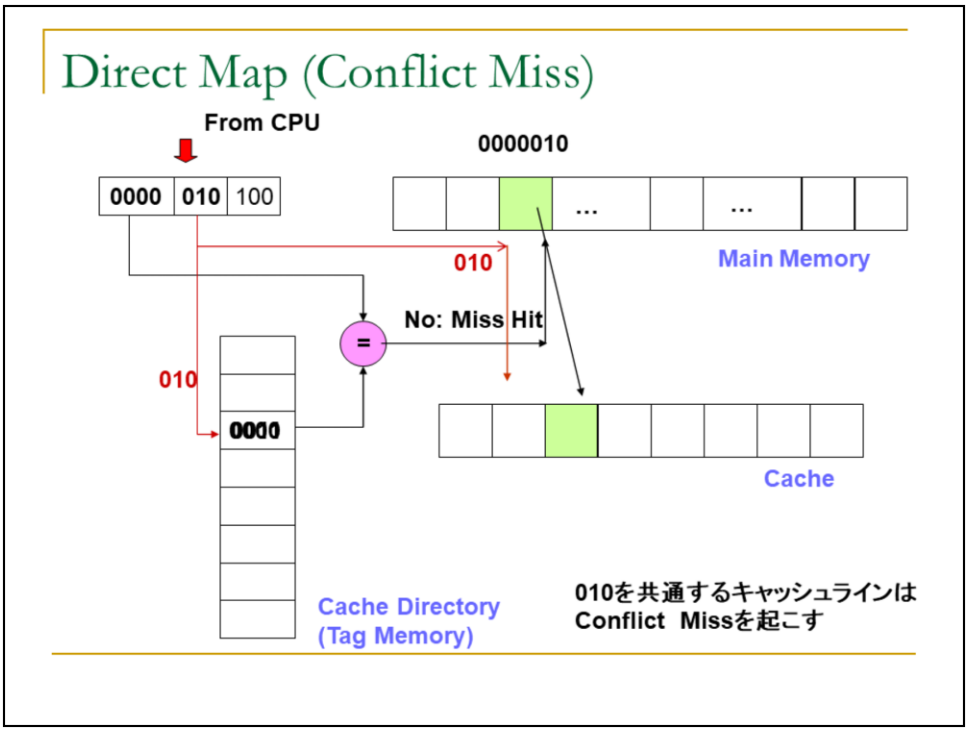


ではここで大変簡単な例を示しましょう。ブロックを8ワードとし、主記憶は1024ワード＝128ブロック、キャッシュは64ワード＝8ブロックと仮定します。キャッシュは0-7までブロックがあります。主記憶の0-127までのブロックを、キャッシュの0-7までのブロックに対して最初から順に割り付けます。8つ割り付けると、先頭に戻って再び割付を始めます。このようにすると、CPUのアドレスは、下から3ビット分はブロック内のどのワードを指定するかを示すアドレスになります。これをブロック内アドレスと呼び、キャッシュと主記憶とのマッピングとは関係なくなります。

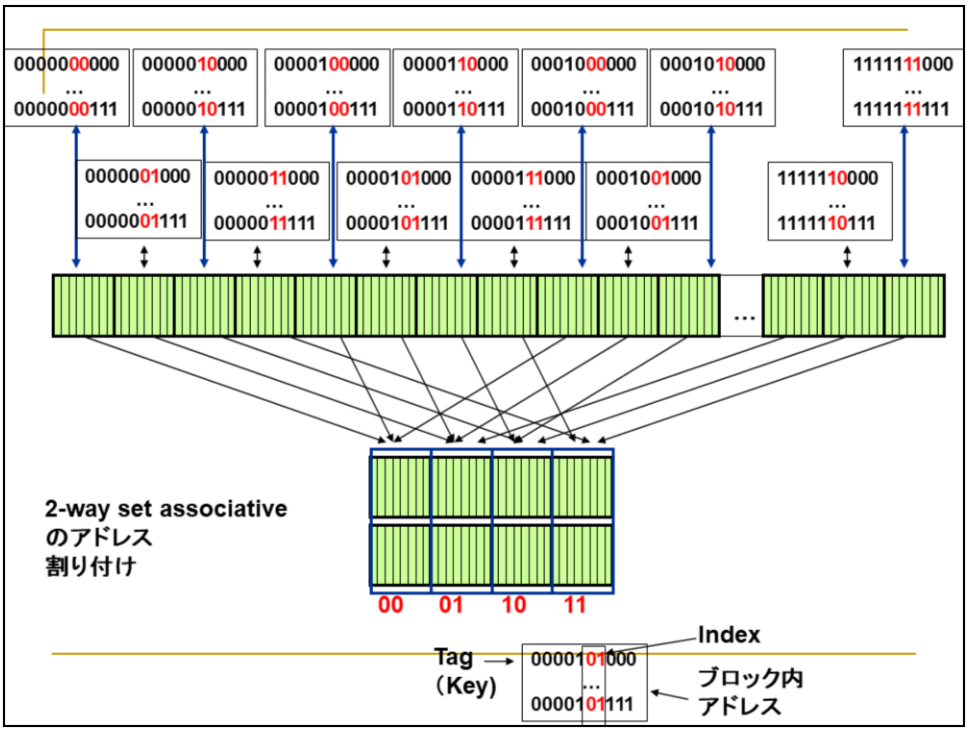
次の3ビットは、キャッシュの0-7までのどこの場所にブロックが割り当てられるかを示します。ここが101ならば5番目に割り付けられるのです。この部分のビットをインデックスと呼びます。残りの部分は、キャッシュに入ってしまったブロックを識別するのに使うことができるため、タグ（キー）と呼びます。



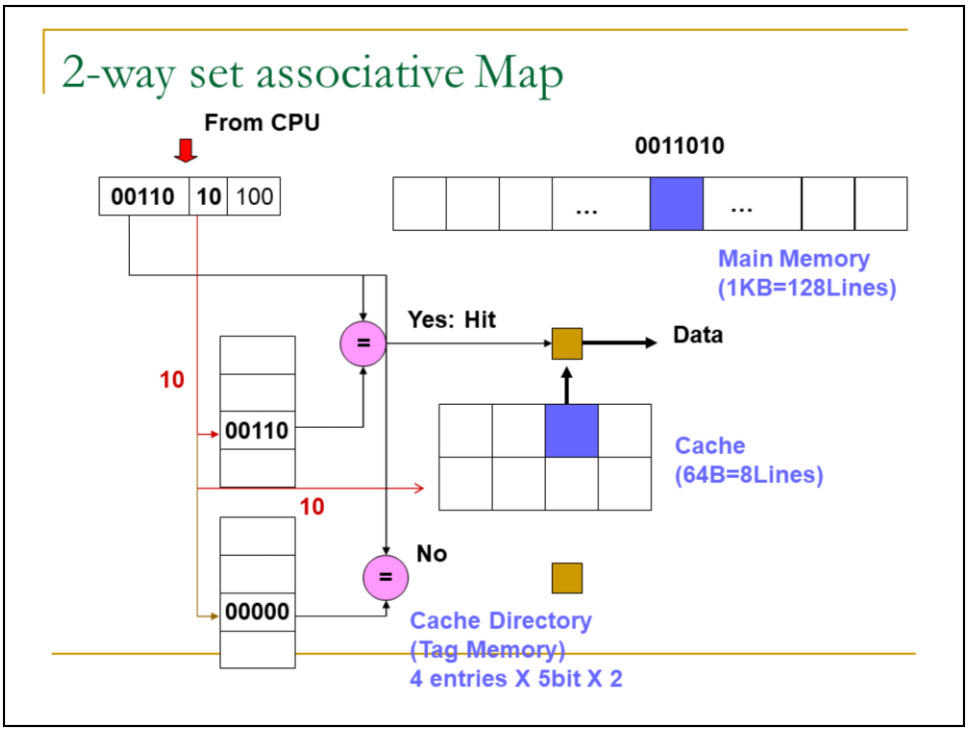
では、このブロックを管理するための機構を考えましょう。それぞれのキャッシュの場所に、どのブロックが入っているかは、タグで識別されます。キャッシュブロック数と同じ深さのメモリを用意し、対応する場所のタグをたくわえます。このメモリをタグメモリとかキャッシュディレクトリと呼びます。この図の例では、キャッシュの位置2=010に、主記憶のブロック0011010が蓄えられています。ブロックの蓄えられる位置は下の3ビットで示されますので、ブロック自体を識別するのはタグの部分であり、0011です。これがキャッシュディレクトリに蓄えられています。CPUからアドレスが出てくると、このうちインデックス部分の010でキャッシュディレクトリとキャッシュを同時にアクセスします。キャッシュディレクトリからは、この位置2に入っているブロック番号である0011が読み出されます。これをCPUのアドレスのタグ部と比較します。これが等しければキャッシュ上にCPUから出てきたアドレスに相当するブロックが存在することが分かります。ヒットしたのです。なので読み出したデータをCPUに渡してやります。この方法がキャッシュの基本でダイレクトマップ方式と呼びます。



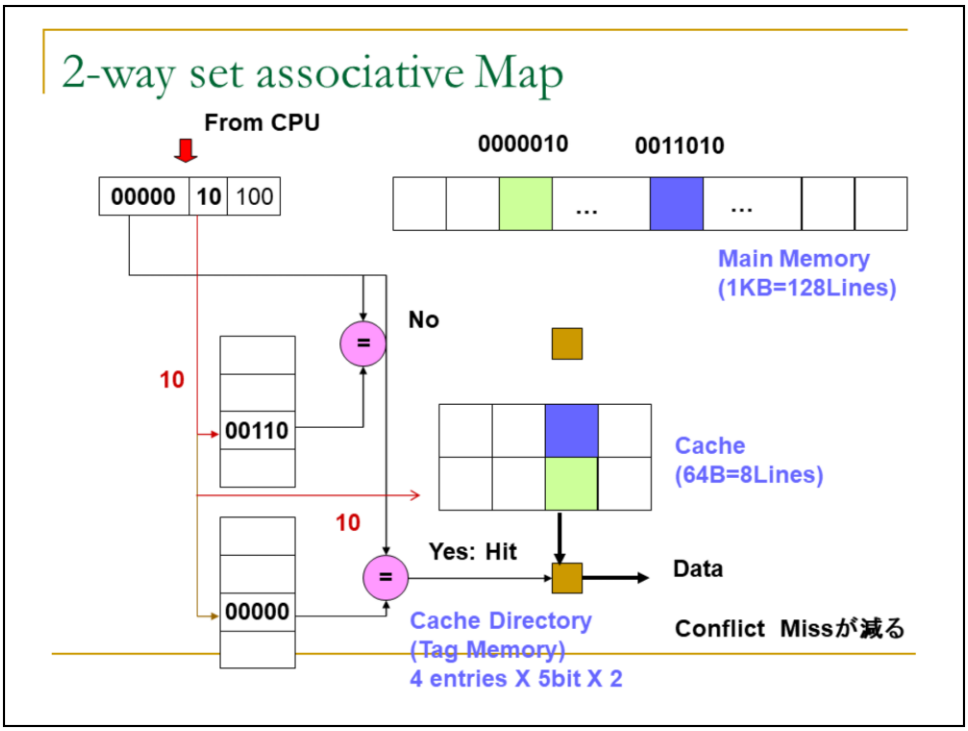
今度はCPUがブロック0000010をアクセスした場合を示します。スライドがPDFなので見難いかもしれませんが、前ページ同様、まず0011が読み出されます。ところがこれはCPUからのタグ0000と違っています。このため、ミスヒットと判定されます。この場合、主記憶からブロック0000010が読み出され、これがキャッシュの010の場所にコピーされます。この操作をリプレイスと呼びます。終了後、タグメモリの内容も0000に書き直します。それからキャッシュからデータを読み出してCPUに送ります。ダイレクトマップ方式では、主記憶のブロックが、キャッシュ上に置かれる位置が、インデックスによってただ一つに決まります。この場合、ブロック0011010と0000010は同じインデックス010を持つため、キャッシュ上で共存できません。キャッシュの容量に余裕があっても、インデックスが競合することで起こすミスのことを競合ミス（Conflict Miss)と呼びます。



インデックスの競合を緩和するため、キャッシュを二つに折りたたみ、ブロック二つを一つのセットと考えます。上の図では00、01, 10, 11の4つのセットを持ちます。また、上の図のそれぞれの行のことをウェイと呼びます。この場合2つウェイを持っていることから2ウェイセットアソシアティブキャッシュと呼びます。この場合、主記憶の各ブロックはセットに対して割り付けます。ダイレクトマップ同様に00から順に割り当てていって11まで来たら、最初に戻ります。すなわち、この場合インデックスはブロック内アドレスを除いた下位ビットになり、残りの5ビットがタグになります。



2ウェイセットアソシアティブキャッシュを実現するためにはキャッシュディレクトリを2組持ち、それぞれのウェイに対応させます。ダイレクトマップ同様に、インデックスでキャッシュディレクトリとキャッシュを同時に参照し、CPUからのアドレスとディレクトリから読み出したタグを比較します。この操作は二つのウェイに対して並列に行われますので、時間が増えるわけではありません。このうちヒットした場合について、ヒットした方から読み出したデータをCPUに送ります。



ここで、ダイレクトマップ同様、0000010のブロックに対してアクセスがあった場合はどうでしょうか？上の図に示すように両方のキャッシュブロックはキャッシュ上で共存することができます。両方のディレクトリを同時に検索して、CPUからのアドレスのタグ部と比較します。ヒットした方があれば、読み出されたデータがCPUに送られます。このことで、競合ミス（Conflict Miss)が減ります。2ウェイセットアソシアティブキャッシュは、二つのディレクトリを同時に参照するので、ダイレクトマップと比べて極端に遅延時間が増えるわけではありませんが、データを選ぶマルチプレクサ分は増えます。また、比較器などのコストも増加します。

Way数とミス率

- Way数が倍ならば割付可能な場所が倍になる
- しかし、割り付ける対象のブロック数も倍になる

→ 椅子とりゲームにたとえると

- ダイレクトマップ：椅子1個　ライバル16人
- 2ウェイ：椅子2個　ライバル32人

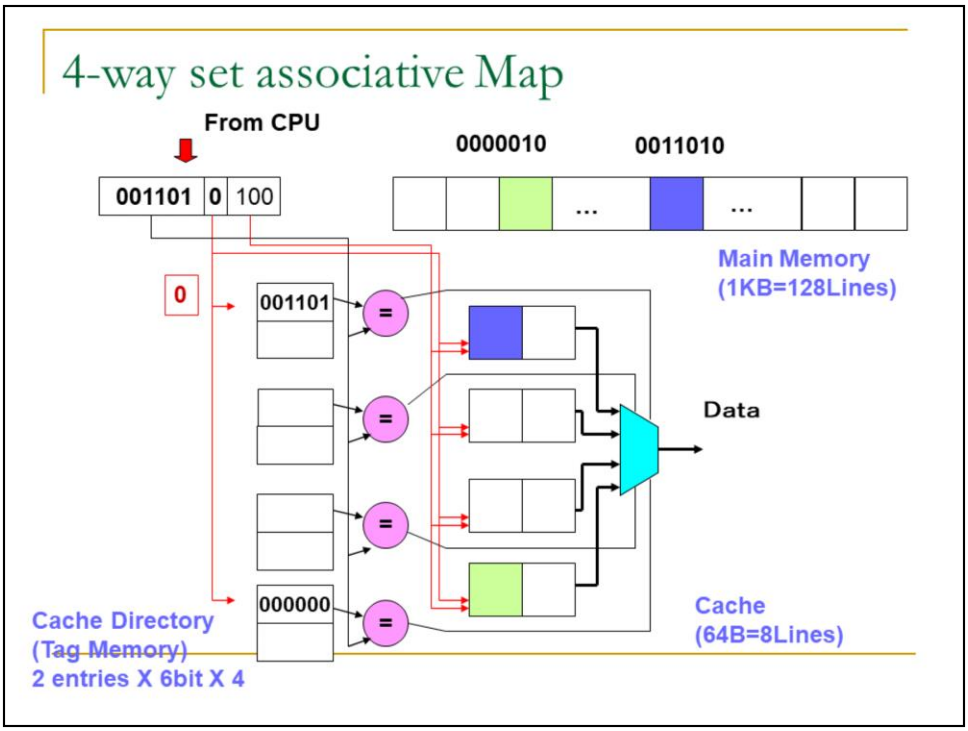
これではあまり状況が改善しないように思える

しかし局所性の原則からライバルは概ねやる気がない

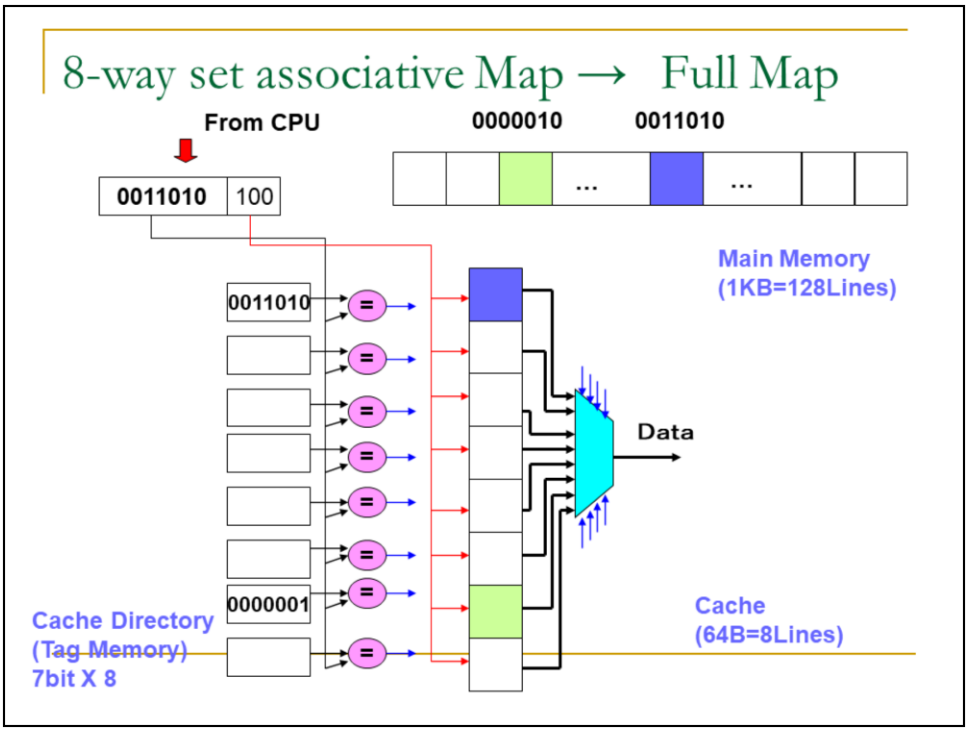
→たまに出てくるやる気のあるライバルと共存できるので2ウェイは効果的

倍のサイズのダイレクトマップと同じ程度の性能が出る

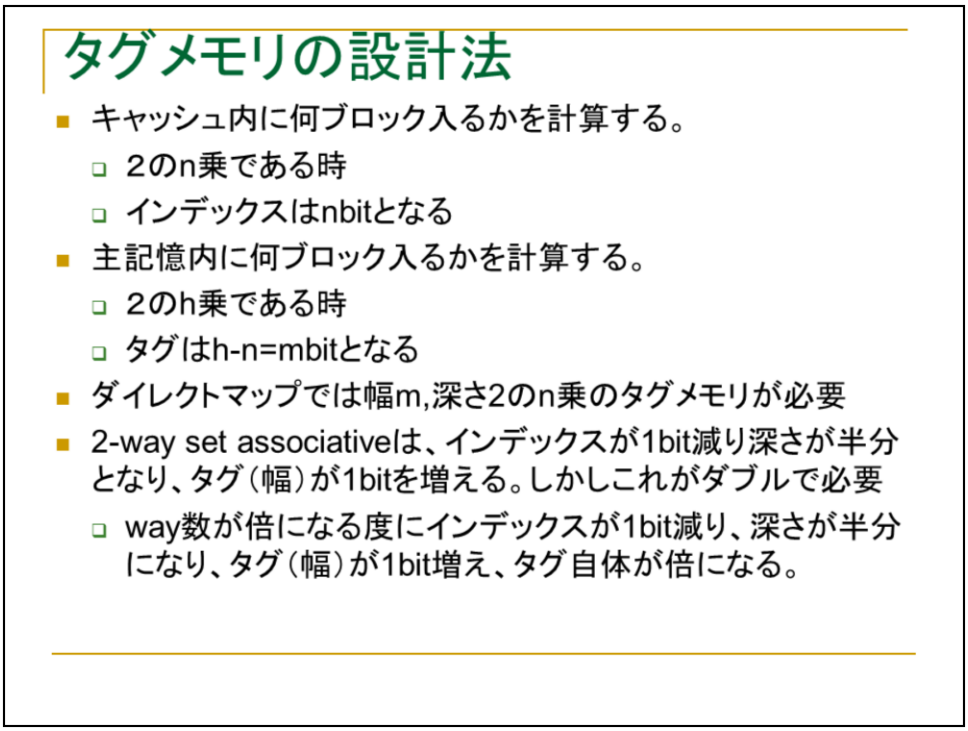
Way数が倍になると割付可能な場所が倍になります。一方で、割り付ける対象のブロック数も倍になります。これを椅子取りゲームにたとえると、前のページの例は、ダイレクトマップが1個の椅子を16人のライバルで取り合うのに対して、2ウェイは椅子の数は倍の2になるが、ライバルの数も32人になってしまいます。これだと状況があまり改善されないように見えます。しかし、ライバルはインデックス分アドレスの距離が離れた所に位置していますので、局所性の原則により、やる気がなく、ほぼ椅子は毎回確保可能な状態なのです。ここでたまたまやる気があるライバルが出てきた場合、椅子が1つだと毎回戦わなければならないですが、2つならば共存できます。これが2ウェイにする効果です。実際、後にグラフを示しますが、2ウェイセットアソシアティブは、2倍のサイズのダイレクトマップと同じ程度の性能が得られます。



この方針を進めてウェイ数をさらに増やすことも可能です。この図はセット数を2にして、ウェイ数を4にしたものです。セットは0か1のどちらかなので、これを1ビットのインデックスで選びます。キャッシュブロックは同じセット内の4つのウェイのどこにでも格納することができます。ディレクトリも4組持たせ、同時に参照して比較して等しいものがあれば、データをマルチプレクサにより選んでCPUに送ります。



ではさらにウェイ数を増やすとどうなるでしょうか？この場合、キャッシュが8ブロックしかないので、ウェイ数を8まで増やすとセットは1つになり、インデックスはなくなってしまいます。この機構では、主記憶のブロックはキャッシュのどの場所に入れることも可能です。セットが一つしかないキャッシュのことをフルマップキャッシュと呼びます。すなわち、ウェイが1のセットアソシアティブをダイレクトマップと呼び、セットが1つのセットアソシアティブをフルマップと呼ぶのです。一般的にはキャッシュの容量はもっと大きいので、フルマップはコストが大きすぎて現実的ではないです。



ここで、タグメモリの設計法をまとめましょう。ポイントはキャッシュに何ブロック入るか？ということです。今2のn乗個ブロック入るとすると、インデックスはn bitとなります。次に主記憶内にブロックが何ブロック入るかを求めます。2のh乗個ブロック入るとするとタグはh-n=mbitとなります。すなわちダイレクトマップでは幅m,深さが2のn乗のタグメモリが必要になることがわかります。2ウェイセットアソシアティブではインデックスが1ビット減り、深さが半分となり、幅は1ビット増えたディレクトリが2組必要になります。以降ウェイ数が倍になる度にインデックスが1ビット減り、深さは半分に、幅は1ビット増えたディレクトリが倍必要になります。

キャッシュのシミュレーション

- c1kai/directの下と、c1kai/2wayの下のそれぞれでtest.asmをシミュレーションする
- test.asm: 0番地と4番地の値を足し算、100番地と104番地の値を足し算。この値の差を求めて8番地の値及び108番地の値と足し算
  - 0: 0,1,2,3,4,5,6,7
  - 0x100: 0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47
  - よくわかんないプログラムだが$3が0xceになるはず
- 両者の動きを比較しよう

iverilogはmakeでOK

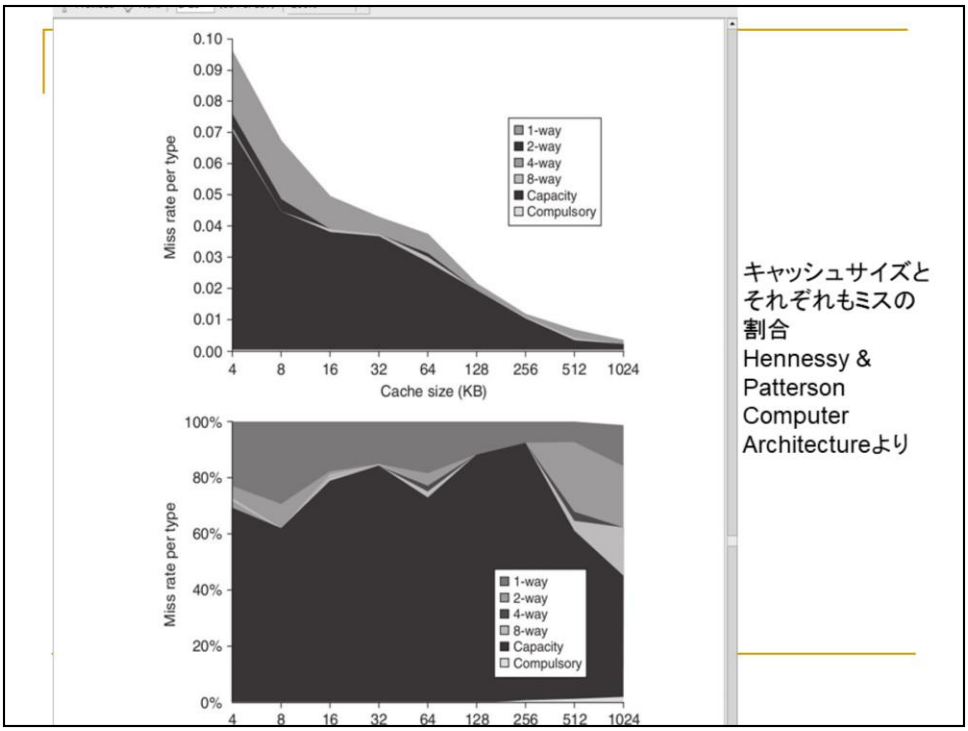
asm.plはmake testでOK

実行は./mipse

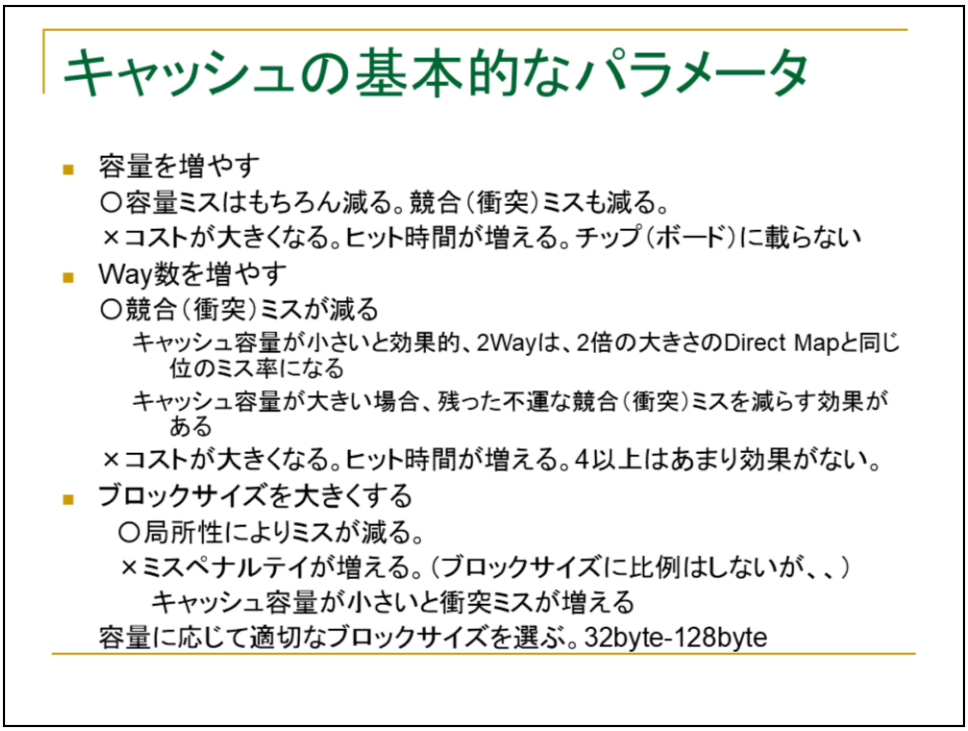
Clock Count, Wait Count, Hit, Missはそれぞれの回数

では、シミュレーションをやってみて様子を見ましょう。

先ほどの式がどの程度正確かどうかは疑問の余地があるとはいえ、キャッシュの性能がミス率とミスペナルティによって決まることは間違いないです。すなわち、キャッシュの性能を上げるには、ミス率を減らすか、ミスペナルティを小さくすれば良いのです。まずミスについて検討しましょう。ミスは、容量ミス、競合ミス、初期化ミスの三つに分けて考えることができます。英語の頭文字をとって3つのCと呼びます。容量ミスは、キャッシュの絶対的な容量不足により生じるミス、競合（衝突）ミスは、インデックスが衝突することによって格納できなくなってしまう問題、最後の初期化（Compulsory:強制、必須という意味です）ミスは、スタート時、プロセス切り替え時など最初にキャッシュにブロックを持ってくるためのミスです。これは避けることができません。



このグラフは、キャッシュの原因を分類したもので、横軸にキャッシュ容量、縦軸にミス率を取っています。1-way(ダイレクトマップ)、2-way…とウェイ数が増えていくにつれ、競合ミスが減っていきます。ウェイ数を無限に増やしても減らすことができない部分が容量ミスになります。初期化ミスは下のほうに見える非常に細い筋です。下のグラフは上のグラフと同じデータですが、ミス率全体を100%と考えて、この中のミスの成分を示しています。



ミス率を減らすのに最も効果的な方法は容量を増やすことで、このことで容量ミス、競合ミスの両方が減ります。しかし、容量が増えるとコストが大きくなり、ヒット時間が増えます。さらに物理的にチップやボードに搭載できる量は制限されます。

次にWay数を増やすと競合(衝突)ミスが減ります。先の図を見ると、キャッシュ容量が小さいとき、2wayは2倍の容量のダイレクトマップとほとんど同じくらいのミス率になります。Way数を増やす効果は4，8と大きくするほど小さくなってしまい、4以上にしてもほとんど効果がなくなります。Way数を増やす効果はキャッシュ容量が小さいときに大きいですが、逆に容量が非常に大きい場合にも、不運な競合ミスを減らしてミス率を非常に小さくするために有効です。前のページの図をご覧下さい。Way数を増やすと比較器やマルチプレクサのコストが大きくなり、ヒット時の遅延が増えます。このため8より大きいものはほとんど使われません。

最後にブロックサイズを大きくする手があります。これについては次のページにグラフが載っています。

ブロックサイズと
ミスの割合
Hennessy &
Patterson
Computer
Architectureより

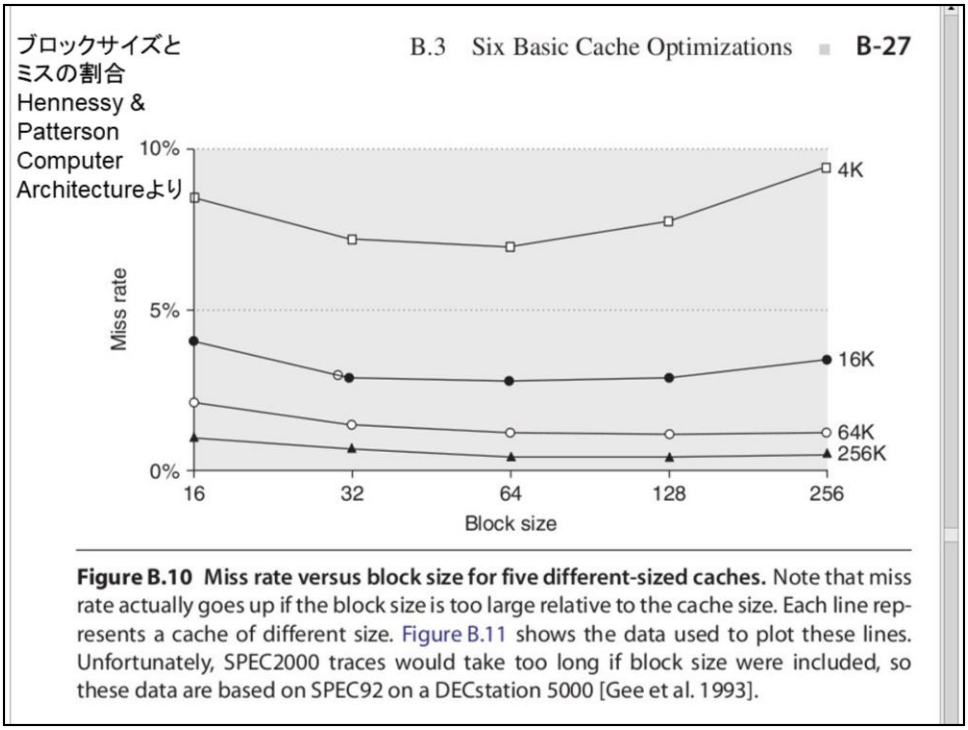
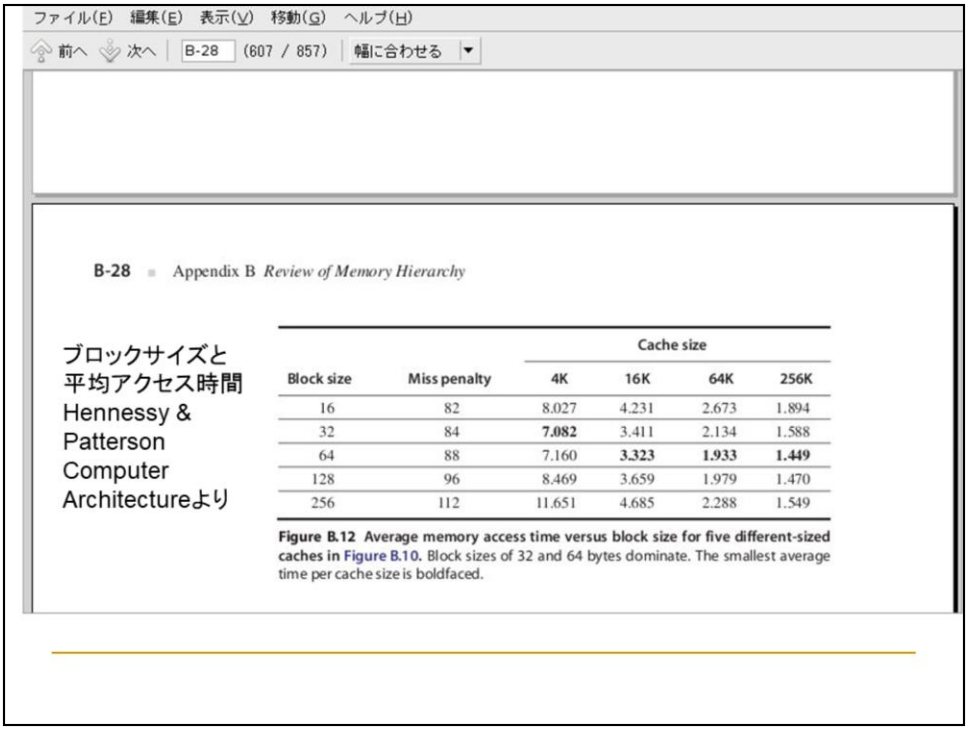


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 [Gee et al. 1993].

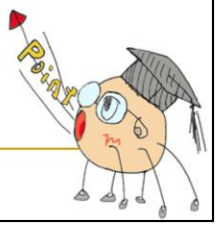
ブロックサイズを増やすと、一度に周辺のデータを取って来ることができるので、局所性の原則からミス率を減らすことができます。しかし、キャッシュ容量自体が小さいときにブロックサイズを大きくすると、インデックスが重なる可能性が増えるため、競合ミスが増えてしまいます。この図はサイズをパラメータに取っているので、一番小さい4Kでこの傾向がはっきり出ています。64K以上のサイズならばブロックサイズを増やしてもミス率は上がりません。とはいえ下がることもないです。



ブロックサイズを増やす問題点は、ミスペナルティが大きくなることです。大きいサイズのデータを動かすのでこれは当然です。しかしDRAMやバスの性質上、サイズに比例して増えるのではなく、増え方はずっとおだやかなものになります。この表はひとつの例であり実装でいろいろ変わりますが、ブロックサイズとミスペナルティ(クロック数)を示しています。キャッシュサイズのところに示してる数値はミス率とペナルティを掛けたものです。太字がもっとも小さい値です。これを見るともっとも小さくなるのは、ブロックサイズが32-64バイトであることがわかります。実際のキャッシュのブロックサイズもこの程度の値を取ります。

本日のまとめ

- 記憶の階層は、高速小容量のメモリをCPUの近くに、遅いが大容量のメモリを遠くに置く
  - 局所性の原則により、高速かつ大容量のメモリに見せかけることが可能
  - キャッシュ
    - 良く使うデータを入れておく高速小容量のメモリ
  - CPUから順にL1,L2はオンチップSRAM,L3はオンボードSRAM
  - 主記憶はDRAM、補助記憶はディスクかフラッシュメモリで作る
- キャッシュのマッピング
  - ダイレクトマップはindexによって入る場所が一箇所に決まる
  - n-wayはN箇所の中から選ぶことができる
  - フルマップはどこに置いても良い
  - キャッシュ構成の設計は28ページを参照
- キャッシュのミスの原因は3つのC
  - キャッシュ容量、ブロックサイズ、way数に依存



インフォ丸が教えてくれる今日のまとめです。

演習1

- 0x00番地からサイズ8の配列A[i]が、0x40番地から同じくサイズ8の配列B[i]が割り付けられている。
- enshu.asmは以下を計算するプログラムである

```
int i,dsum;
dsum =0;
for(i=0; i<8;i++)
  dsum += B[i]-A[i];
```

- これをダイレクトマップのキャッシュ（direct)で実行したときと2ウェイセットアソシアティブ（2way)で実行したときで、両者のミスの回数と、演算結果が出るまでのクロック数をシミュレーションして求めよ。（動かして出た数字を記録する）
- 計算の順番を変えることにより、directで実行した場合のミス率を減らせ（directは変更前、変更後の数値、2wayは変更前の数値を示し、変更したenshu.asmと共に提出せよ。

これはシミュレータを使ってenshu.asmを実行して様子を見る問題です。プログラムの意味とキャッシュの挙動を良く見てくださいませ。

演習2

- 64kワードの主記憶に対して4kワードのキャッシュを設ける
- ブロックサイズは16ワードとする
- ダイレクトマップ、2way set associative、4way set associativeキャッシュのタグメモリ構成をそれぞれ示せ
- ヒント：タグメモリの設計法のページを参照！

この問題は設計問題で、コンピュータを使う必要がないです。タグメモリの設計法のページを見ながらやってください。幅XXビット、深さYYがZZ個という表現で示してください。