5 主記憶

♣ メモリーに関する基礎知識 (覚える)

♣ 「キャッシュ」と「仮想記憶」の概念を知る

5.1 基礎知識

(1) ROM と RAM

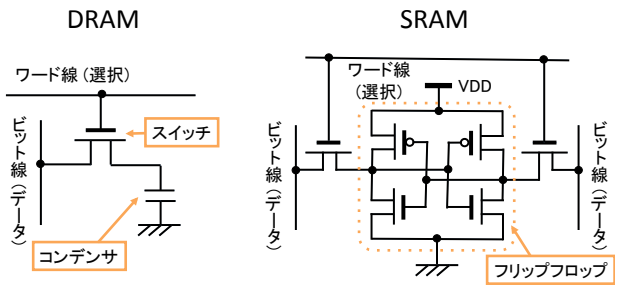
- ROM (read-only memory; 読み出し専用 メモリー)
  - CPU からデータの 読み出し はできるが, 書き込み はできない
  - データはあらかじめ書き込まれている
  - 電源切断 → データは消失 しない
  - データを 書き換え 可能なものとそうでないものがある
    - マスク ROM … 製造時に 回路 として作成 ⇒ 書き換え不可
    - EPROM (erasable programmable ROM) … 紫外線や高電圧により消去/書き込み
- RAM (random access memory)
  - CPU からデータの読み出し/書き込みが可能
  - 電源切断 → データは消失 する (揮発 性)
- コンピュータの主記憶は, 通常 ROM と RAM の組合せで構成
  - PC, EWS, メインフレーム等
    - 起動 時に実行される初期化プログラムを ROM に格納
    - OS や応用プログラムは HDD/SSD 等から RAM にロードして実行
  - 組込みシステムやゲーム機
    - OS や応用プログラムも ROM に置かれることが多い

(2) RAM を実現する半導体メモリー

- DRAM (dynamic RAM)
  - コンデンサ に 電荷 を蓄える/蓄えないにより 1 ビットを記憶
  - 1 セルをコンデンサ 1 個とトランジスタ 1 個で構成
  - リフレッシュ が必要
    - コンデンサに蓄えられた電荷は徐々に 消失 するので,
      消失前に値を 読み出し て 再度書き込む

    - ＊ 数十 ms 毎に行う必要がある
    - ＊ そのための 回路 が必要 (CPU がリフレッシュを行うわけではない)
- SRAM (static RAM)
  - フリップフロップ回路により 1 ビットを記憶
  - 1 セルを 4〜6 個程度のトランジスタにより構成
  - DRAM より集積度は 低い が, 高 速で, リフレッシュ 不要
- SRAM と DRAM の比較

	DRAM	SRAM
アクセス速度	遅い	速い
集積度	高い	低い
コスト	安い	高い
リフレッシュ	必要	不要



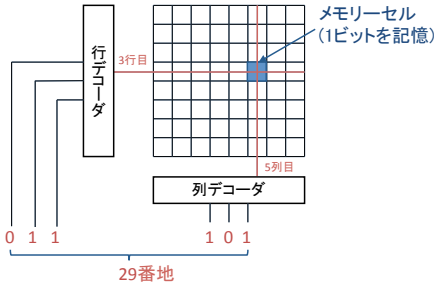
(3) メモリの構造

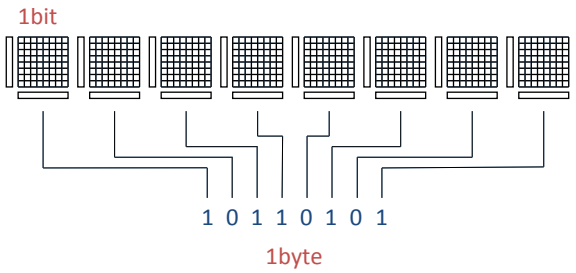
- メモリセルは通常 2 次元に配列する
  - メモリーセル (1 ビットを記憶する素子) の指定 ⇒ 行 と 列 を指定

    ⇒ 通常, アドレスの 上位 を行アドレス, 下位 を列アドレスとする

    (10 進数での例え) 2563 番地 ⇒ 25 行 63 列

    (2 進数) b10101111 番地 ⇒ b1010 行 b1111 列





- バイトや語の構成
  - 上の図は 1 ビット分

    これを 8 個並べれば, 1バイト データを読み書きするメモリーができる

(4) アドレスマルチプレクシング (address multiplexing)

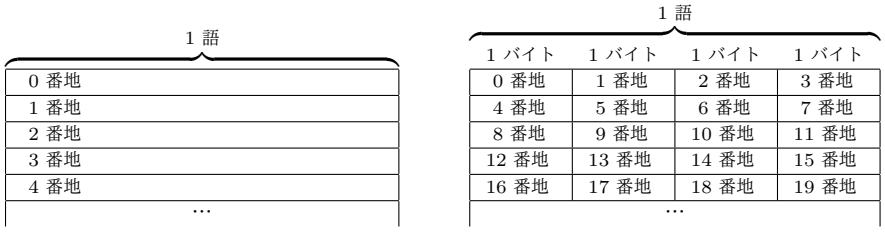
- アドレスを 複数回 に分けて供給し, アドレス信号線 の本数を減らす方法
  - 2 回 (行 アドレスと 列 アドレス) に分けることが多い

(例) 4GB (32 ビットでアドレッシング) のメモリー

⇒ まず行アドレス (16 ビット) を入力し, 次に列アドレス (16 ビット) を入力

(5) ワードアドレスとバイトアドレス

- ワードアドレス
  - メモリーの番地がコンピュータの 1 語 (word) 単位で付けられる
  - 特殊なコンピュータ (DSP [1]や数値計算専用のコンピュータ) で用いられる



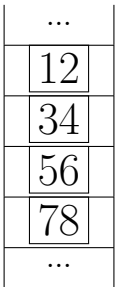
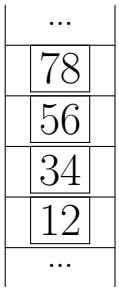
ワードアドレスのメモリー　　バイトアドレスのメモリー

- バイトアドレス (byte addressed, byte addressable)
  - メモリーの番地が 1 バイト単位で付けられている
  - 汎用 のコンピュータ (MIPS や x86 等) で用いられる[2]

☆ 以下では, バイトアドレスのコンピュータを扱う

(6) バイトオーダー/エンディアン (byte order / endianness)

複数バイト (で一かたまり) のデータを, バイト単位で主記憶やファイルに格納する際のバイトの 順序

	ビッグエンディアン (big endian)	リトルエンディアン (little endian)
(例) 4 バイトのデータ x 12345678 を 100~103 番地に格納する場合の配置	 ”big end comes first” と覚える	 ”little end comes first” と覚える
コンピュータ	SPARC, Java VM 等	x86 , Z80 等
	(切り替え可能; bi endian と言う)	MIPS , ARM, PowerPC
ファイルフォーマット	JPEG, Photoshop 等	BMP, GIF 等

[1]Digital Signal Processor; ディジタル信号処理専用プロセッサ

[2]汎用のコンピュータでは文字データを扱うので, 1 バイト単位でデータにアクセスできることが必須となる.

☆ 通常のプログラミングでは特にエンディアンを意識する必要はないが,

バイナリファイル (2進 表現でデータを保存したファイル) の読み書き[3]や,

ネットワーク を介したデータ送受信では注意が必要.

例題 5.1 ビッグエンディアンの MIPS において, `$30=1024`, `$8=x12345678` のときに, `sw $8,0($30)` を実行すると, 主記憶の 1024 ∼ 1027 番地はそれぞれどんな値になるか. また, これに続けて `lw $8,0($30)` を実行すると, `$8` の値はいくらになるか.

`sw` *Rt*`,`*Imm*`(`*Rs*`)` …`Mem[` *Rs* + *sx*(*Imm*) `,4] =` *Rt*

`lw` *Rt*`,`*Imm*`(`*Rs*`)` …*Rt*`= Mem[` *Rs* + *sx*(*Imm*) `,4]`

(答)

番地	1024	1025	1026	1027
値	x 12	x 34	x 56	x 78

`$8 = x` 12345678

例題 5.2 x86 (リトルエンディアン) において, `ebp=1024`, `eax=x12345678` のときに, `movl %eax,(%ebp)` を実行すると, 主記憶の 1024 ∼ 1027 番地はそれぞれどんな値になるか. また, これに続けて `movl (%ebp),%eax` を実行すると, `eax` の値はいくらになるか. ただし, `movl` 命令の動作は次の通り.

`movl %`*Rs*`,(%`*Rb*`)` …`Mem[`*Rb*`, 4] =` *Rs* (ストア)

`movl (%`*Rb*`),%`*Rd* …*Rd*`= Mem[`*Rb*`, 4]` (ロード)

(答)

番地	1024	1025	1026	1027
値	x 78	x 56	x 34	x 12

`eax = x` 12345678

例題 5.3 `x` と `y` はいずれも 16 ビット (2 バイト) の符号無し整数型の変数とする. ビッグエンディアンのコンピュータで, `x=1` のとき, 変数 `x` の値をファイルにバイナリ形式で書き込み, それをリトルエンディアンのコンピュータで変数 `y` に読み込むと, `y` の値はいくらになっているか.

(答) `y =` 256 となる.

ファイルには 10 進数で 1 が書き込まれる. これを 16 進数で表すと 00 01 である.

リトルエンディアンのコンピュータでこのデータを読み込むと, バイトの順序が入れ替わるので, 読み込むデータは, 16 進数で 01 00 となる. これは 10 進数では 256 である.

(7) メモリーインタリーブ (memory interleaving)

メモリーを複数個の単位 (バンク (bank)) に分割 ⇒ 複数データへのアクセスを高速化

(例) 下記の 4 バンク構成で, 連続する 4 バイトデータは, 1 バイトデータと同じ速度でアクセス可能

バンク 0	バンク 1	バンク 2	バンク 3
0 番地	1 番地	2 番地	3 番地
4 番地	5 番地	6 番地	7 番地
8 番地	9 番地	10 番地	11 番地
12 番地	13 番地	14 番地	15 番地
···	···	···	···

☆ 後述のキャッシュを用いる場合には, 主記憶との間のブロック (32B∼256B) の転送の高速化に有効

[3] C 言語標準関数の `printf` や `scanf` は, 文字列に変換したデータを入出力するので問題は生じないが, データの 2 進数表現のままで入出力する `write` や `read` を用いる場合には, エンディアンに気をつけなければならない.

(8) メモリー整列 (memory alignment)

- メモリー整列制約
  - 複数バイトのデータの **開始番地** がある整数の **倍数** でなければならないという制約

    (例) MIPS では 4 バイト (32 ビット) の整数データの先頭番地は **4** の倍数でなければならない

    (例) MIPS では 8 バイト (64 ビット) の浮動小数点データの先頭番地は **8** の倍数でなければならない

    ☆ 2^k バイトのデータの開始アドレスが **2^k** の倍数でなければならないことが多い

    ちなみに, アドレスが 2^k の倍数のとき, アドレスの下位 k ビットは **全て 0**
  - データの開始番地が n の倍数のとき, このデータは n バイト **境界に整列** されていると言う
- 整列の必要性
  - 整列されていないデータのアクセスは **遅い** /ハードウェアが **複雑**

    (例) 1 語 4 バイトのコンピュータでは, メモリーインタリーブで 1 語を高速にアクセスする

100 番地	101 番地	102 番地	103 番地
104 番地	105 番地	106 番地	107 番地

    * 100 番地や 104 番地から始まる 4 バイトデータのロード → 1 回のアクセス
    * 103 番地から始まる 4 バイトデータのロード → 2 回のアクセス + シフト + 論理演算

    1) | 100 番地 | 101 番地 | 102 番地 | 103 番地 | 100 番地から始まる 4 バイトをロード

    2) | 103 番地 | | | | 24 ビット (3 バイト) 左シフト

    3) | 103 番地 | | | | 104 番地から始まる 4 バイトをロード
       | 104 番地 | 105 番地 | 106 番地 | 107 番地 |

    4) | 103 番地 | | | | 8 ビット (1 バイト) 論理右シフト
       | | 104 番地 | 105 番地 | 106 番地 |

    5) | 103 番地 | 104 番地 | 105 番地 | 106 番地 | 2 つのデータの論理和

  ☆ **x86** は, 整列されていないデータにアクセスする (上記の処理を行う) ハードウェアを備えている

  ⇒ メモリー整列制約はない

  `movl 103,%eax` (eax=M[103]) のような命令も実行可能

  ただし, 整列されていないデータのアクセスは **遅い** ため, **コンパイラ** が整列する

  ☆ **MIPS** は, 整列されていないデータにアクセスするためのハードウェアを持っていない

  ⇒ メモリー整列制約がある

  `lw $8,103($0)` ($8=M[103]) はメモリー整列制約違反 (**アドレスエラー** が発生)

(9) アドレス空間

- **論理** アドレス空間 (**logical** address space)

  命令 で指定できるアドレスの範囲 (あるいはその大きさ)

(例) MIPS: 32 ビットで番地指定

→ 論理アドレス空間は $\boxed{0}$ ~ $\boxed{2^{32}-1}$ (その大きさは $\boxed{2^{32}}$ B = $\boxed{4\text{GB}}$)

(例) 24 ビットでアドレス指定するコンピュータの論理アドレス空間の大きさは $\boxed{2^{24}}$ B = $\boxed{16\text{MB}}$

- $\boxed{\text{物理}}$ アドレス空間 ($\boxed{\text{physical}}$ address space)

  実際に $\boxed{\text{使用可能}}$ なアドレスの範囲 (あるいはその大きさ)

  (例) 論理アドレス空間が 4GB でも, 搭載されているメモリーが 2GB なら, 物理アドレス空間は $\boxed{2\text{GB}}$

10bit で 1K

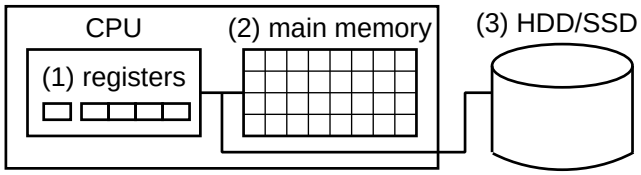
$2^{10} = 1024 \fallingdotseq 1000$ なので, $2^{10} = \boxed{1\text{K}}$, $2^{20} = \boxed{1\text{M}}$, $2^{30} = \boxed{1\text{G}}$ と覚えておくと便利.

例えば, $2^{32}\text{B} = 2^2 \times 2^{30}\text{B} = \boxed{4\text{GB}}$ と直ちにわかる.

5.2 記憶階層

コンピュータの記憶素子/装置とその特性

☆ 高速なものほど $\boxed{\text{小}}$ 容量で $\boxed{\text{高}}$ 価



	速度※1	容量※1	不揮発性※2	主な用途
(1) レジスタ (CPU 内部)	超高速 ($\boxed{1\text{ clock}}$)	小 (数 ~ 数十個)	×	一時記憶
(2) 主記憶	高速 ($\boxed{10}$ ~ $\boxed{100\text{ns}}$)	中 (~ $\boxed{\text{数 GB}}$)	×	実行中のプログラム 計算中のデータ
(3) 外部記憶 (HDD/SSD)	低速 ($\boxed{\text{数 ms}}$)	大 (~ $\boxed{\text{数 TB}}$)	○	インストールされたプログラム 「保存」されたデータ

※1 括弧内は標準的な PC の数値　※2 不揮発とは, 電源を切っても記憶が消失しないこと

- 記憶階層 (memory $\boxed{\text{hierarchy}}$)

  記憶装置の組合せにより $\boxed{\text{速度}}$ / $\boxed{\text{容量}}$ / $\boxed{\text{価格}}$ のバランスがとれた記憶を構成する

- 2 つの重要な概念

  1. キャッシュ

     $\boxed{\text{CPU}}$ と $\boxed{\text{主記憶}}$ の間に高速な記憶装置を配置

     ⇒ 見かけ上, 主記憶へのアクセスを高速化

  2. 仮想記憶

     $\boxed{\text{主記憶}}$ と $\boxed{\text{補助記憶}}$ 装置を組み合わせる

     ⇒ 見かけ上, 実際よりも大きな主記憶を実現

5.2.1 キャッシュ

- キャッシュ (cache) [4]

  CPU と 主記憶の間に配置する, 小容量で高速の記憶装置

  * 主記憶に DRAM を, キャッシュに SRAM を用いることが多い

  CPU が主記憶にアクセスしたデータを保持し, 同じアクセスがあればそのデータを 再利用

  * 参照の 局所性 (一般的なプログラムは, 同じ番地のデータや命令に繰り返しアクセスするという性質) を利用[5]
  * プログラムからキャッシュの存在は見えない. すなわち, キャッシュの動作を観測したり制御する命令はない. (プログラムからキャッシュは transparent であるという) [6]

- 用語
  - キャッシュヒット (cache hit) … アクセスしたデータがキャッシュ内に見つかること
  - キャッシュミス (chache miss) … ~ 見つからないこと
  - ヒット率 (hit ratio) … 全メモリアクセスのうちキャッシュヒットした割合

    ☆ 標準的なキャッシュのヒット率は 80 %程度と言われる
  - キャッシュミスペナルティ (cache miss penalty)

    … キャッシュミス時に CPU が 待た される時間やクロック数
  - キャッシュコヒーレンス (cache coherence)

    キャッシュと主記憶のデータ 一貫性 (データの 食い違い がないこと)
  - 命令 キャッシュ (instruction cache) と データ キャッシュ (data cache)

    命令とデータに対して 別々 にキャッシュを設けることがある
  - 複数階層のキャッシュ

    CPU と主記憶の速度差が広がりつつあるため, 2 段階以上のキャッシュ構成を取る場合がある

    CPU に近い側からレベル 1 (L1) キャッシュ, レベル 2 (L2) キャッシュ, … と呼ぶ

例題 5.4 キャッシュヒット時のメモリーアクセスに必要なクロック数が 1, キャッシュミス率が 20%, キャッシュミスペナルティが 50 クロックのとき, メモリーアクセスに要する平均クロック数はいくらか.

(答) ヒット率が 80% でそのときは 1 クロック, ミス率が 20% でそのときは 51 クロックかかるので,

$\boxed{(0.80 \times 1 + 0.20 \times 51)} = 11$ クロック.

(別) ミスした時だけペナルティが加わるので, $\boxed{1 + (0.20 \times 50)} = 11$ クロック.

---

[4] cache は英語で「隠し場所」の意味.

[5] 音声や画像等マルチメディアデータの処理では必ずしもこの性質が成り立たたず, キャッシュの効果が得られないことがしばしば問題になる.

[6] 最近では, キャッシュの動作をプログラムから制御可能にして, メモリーアクセスの効率の向上を目指す手法がある.

5.2.2 仮想記憶

- 仮想記憶 (virtual memory)

  主記憶に入り切らないデータを 外部記憶 (HDD/SSD) に記憶

  → アクセスがあった時点で主記憶にロードする (主記憶が満杯なら, 他のデータを外部記憶に追い出す)

  * キャッシュと同様, 参照の局所 性を利用している
  * キャッシュと同様, 一般のプログラムから仮想記憶の存在は見えない.
  * 外部記憶へのアクセスは 入出力 であり, OS により制御される.

- 用語
  - ページフォルト (page fault) … アクセスしたデータが主記憶内に見つからないこと
- 仮想記憶の長所
  - 実際の 容量以上 の主記憶があるかのようにプログラムを実行できる
  - 物理アドレス空間では不連続な領域を, 連続 した領域として扱える
- 仮想記憶の短所
  - 遅い
  - 実装 (ハードウェアや OS) が 複雑

Nagisa ISHIURA