計算機科学概論 — これまでのまとめ Apr. 26, 2013 奥乃 博

(*http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/13/IntroCS/*)

1 計算システムの構成

- 計算機の 5 つの古典的な構成要素：

  制御 (*control*), **データパス** (*datapath*), **メモリ** (*memory*), **入力** (*input*), **出力** (*output*)

  (最初の 2 つは**プロセッサ** (*processor*)」

- ハードウェアとソフトウェアは**階層構造**をとる.
- 重要なインターフェース: **命令セットアーキテクチャ**(*Instruction Set Architecture*)

2 性能評価の項目

- **応答時間** (*response time*)

  — 入力に対して出力が得られる時間

- **実行時間** (*execution time*)

  — タスクの開始から終了までの時間

- **スループット** (*throughput*)

  — 時間内に行なった総仕事量

> 問 1 次の変更は上記のどの項目を改善するか.
>
> 1. 計算機中のプロセッサを高速のものと交換
> 2. システムにプロセッサを増設.

性能 (*performance*) は実行時間の**逆数**. **性能比** n は

$$\mathbf{n} = \frac{\textbf{性能}_\mathbf{X}}{\textbf{性能}_\mathbf{Y}} = \frac{\textbf{実行時間}_\mathbf{Y}}{\textbf{実行時間}_\mathbf{X}}$$

> 問 2 あるプログラムが計算機 A では 10 秒, 計算機 B では 15 秒かかる. A は B よりどれだけ早いか.

3 時間による性能測定

プログラムの実行時間 (execution time) とは:

- **実時間** (*wall-clock time, response time, elapsed time*)

  — プログラムが終了するまでの時間

- **CPU 実行時間, CPU 時間**

  — **ユーザ** CPU 時間と**システム** CPU 時間に分かれる.

UNIX システムでの応答:

```
90.0u 12.9s 2:39 65%
```

システム CPU 時間, ユーザ CPU 時間, 実時間, 実時間の比率

クロック間隔 (*clock period*) とは

- **クロックサイクル** (*clock cycle*) e.g., 10ns
- **クロックレート** (*clock rate*) e.g., 100MHz

[注意] マシンサイクルとクロックサイクルは同じことが多いが, 必ずしも一致しているわけではない.

4 性能を決める基本要素

記号	性能の基本要素	単位
T	プログラム実行時間	秒
PC	プログラムの実行命令数	命令
CPI	平均命令実行サイクル数	サイクル/命令
MC	マシンサイクル	秒/サイクル

$$T = PC \times CPI \times MC$$

性能の各基本要素の値に影響を与える要因:

- PC: 命令セットアーキテクチャ, コンパイラ, アルゴリズム
- CPI: 論理方式, 命令セットアーキテクチャ
- MC: デバイス, 論理方式

一般に, PC と CPI とは相反する.
論理方式の例: キャッシュ, スーパスカラ方式, out-of-order 実行方式,

> 問 3 あるプログラムは計算機 A (100MHz) では 10 秒かかる. これを 6 秒で走らせる計算機 B を設計したい. ただし, B では A よりもクロックサイクルが 1.2 倍かかる. B のクロックレートは最低いくらか.

> 問 4 同一の命令集合アーキテクチャの計算機が 2 種類ある. A のクロックサイクルは 10ns, あるプログラムの CPI は 2.0. B のクロックサイクルは 20ns, 同じプログラムの CPI は 1.2 である. どちらがどれだけ早いか.

命令をクラス分けする．このとき, C_i をクラス i の命令の個数, CPI_i をそのクラスに対する平均 CPI とすると, 全命令に対する CPI は

$$\mathbf{CPI} = (\sum_{i=1}^{n}(\mathrm{CPI}_i \times \mathrm{C}_i))/(\sum_{i=n}^{n}\mathrm{C}_i)$$

> 問 5 命令クラス A, B, C の各 CPI を 1, 2, 3 とする．このとき, コード列 1 と 2 とではどちらが早いか．各々の CPI を求めて, 速度を議論せよ．

コード列	C_A	C_B	C_C
1	2	1	2
2	4	1	1

5 良く使用される性能尺度

5.1 MIPS 値 *million instructions per second*

$$\text{MIPS 値} = \frac{\text{命令数}}{\text{実行時間} \times 10^6} = \frac{\text{クロックレート}}{\text{CPI} \times 10^6}$$

$$\text{実行時間} = \frac{\text{命令数}}{\text{MIPS 値} \times 10^6}$$

この MIPS 値を**ネイティブ** (*native*) **MIPS 値**と呼ぶ．

> 問 6 問 5 と同じ命令クラスで, 同一プログラムに対して 2 種類のコンパイラが以下のようなコードを生成した．計算機のクロックレートを 100MHz とすると, 両者の性能を MIPS 値と実行時間の点から比較せよ．

コード生成	C_A	C_B	C_C
コンパイラ 1	5	1	1
コンパイラ 2	10	1	1

MIPS 値の落し穴:

- ピーク MIPS 値にダマサれない．

  命令ミックスのとりかたによって MIPS 値が変化．問 6 の計算機のピーク MIPS 値は 100 MIPS．そのとき, CPI = 1.0．すなわち, クラス A の命令のみ．

- **相対 MIPS 値**は, 特定のプログラムと入力に対してのみ有効．

  80 年代の標準計算機 DEC VAX11/780 — 1 MIPS

5.2 MFLOPS 値 *million floating-point operations per second*

$$\text{MFLOPS 値} = \frac{\text{プログラム中の浮動小数点演算回数}}{\text{実行時間} \times 10^6}$$

- MIPS 値の様に命令数ではなく, 演算回数を使用

  $\Longrightarrow$ 計算機に依存しないと主張．

- 浮動小数点演算回数は計算機によって変わる．

  Cray-2 (除算なし) vs. M 68882 (除算あり)

- 整数, 浮動小数点, 単長, 倍長の組合せにより変化．

  正規化 (*normalized*) MFLOPS 値 —

  演算によって重みづけ (プログラムにより変化)

6 ベンチマークプログラム

性能評価のために**ワークロード** (*workload*) を形成するのに使用するプログラムを**ベンチマーク** (*benchmark*) と呼ぶ．実際の応用プログラムがもっともよいベンチマークであることが経験的に分かっている．

例: SPEC ベンチマーク群 (時代とともに変わる)

総合評価法: 実行時間の算術平均, 重みつき平均 (ワークロードの係数を掛ける), . . .

7 性能評価での注意事項

1. ある部分の性能を改善できたからといって, その比率だけ全体的な性能が向上するわけではない．

   アムダールの法則 (*Amdahl's law*)

   $$\text{速度向上} = \frac{\text{改善後の性能}}{\text{改善前の性能}} = \frac{\text{改善前の実行時間}}{\text{改善後の実行時間}}$$

   初期のアムダールの法則は

   $$\text{改善後の実行時間} = \Bigl(\frac{\text{改善された実行時間}}{\text{改善率}} + \text{改善とは無関係の実行時間}\Bigr)$$

   アムダールの法則の補則: 共通部分を高速化せよ．

2. ハードウェアに依存しない尺度を使用すれば (e.g., 速度の尺度としてコードサイズを使用), 性能が予測できるか．×

表 1: n 進法

2 進数 (binary)	8 進数 (octal)	10 進数 (decilam)	16 進数 (hexadecimal)	2 進数 (binary)	8 進数 (octal)	10 進数 (decilam)	16 進数 (hexadecimal)
0000	00	00	0	1000	10	08	8
0001	01	01	1	1001	11	09	9
0010	02	02	2	1010	12	10	A
0011	03	03	3	1011	13	11	B
0100	04	04	4	1100	14	12	C
0101	05	05	5	1101	15	13	D
0110	06	06	6	1110	16	14	E
0111	07	07	7	1111	17	15	F

3. 性能評価をするときには, 3 つの基本要素 (**マシンサイクル**, **CPI**, **命令数**) を使用すること.

4. 性能比較するのに, ピーク性能を使用するな.

5. 合成型ベンチマークで性能予測をするな.

   Whetstone (Algol → Fortran, 科学技術計算), Dhrystone (Ada → C, システムプログラム)

6. 性能価格比 (cost/performance) の設計には, 高性能を狙った設計と低価格を狙った性能 を両極端とするさまざまな価値観があることに注意.

8 計算機の構成要素の接続

経路 (*Path*) — データや制御のための信号の伝達

1. 専用接続 (point-to-point connection) : 2 つの装置間の通信専用. 例: メモリーバス (CPU と MMU 間), 入出力バス (CPU と IOU 間)

2. バス接続 (bus connection, multi-drop connection) : 共用. 単一バス接続, or 複数バス接続.

9 主記憶装置の構成

- 主記憶装置 — プログラムやデータを記憶するための番地のつけられた記憶セル (語) の集合.

- **ビット** (*bit*, b) — データの最小単位, $2^{16} = 65,536 = 64K$, $2^{32} = 4,294,967,296 = 4G$

- **最上位ビット** (*Most Significant Bit*, MSB),

  最下位ビット (*Least Significant Bit*, LSB) — 1 語の左端・右端のビット

- **語長** — 記憶セルの大きさ, **バイト** (*byte*, B) — 1 文字を格納するための単位, 通常は 8 ビット

- **パリティビット** (*parity bit*) — 信頼性向上のために付加されたパリティチェック用ビット.

主記憶装置の構成

1. 番地信号線 (address lines) — 番地を指定 (本数: 番地のビット数以下, 多重化, multiplex)

2. データ信号線 (data lines) — 書き込むデータを送り, 読み出すデータを受ける (本数: データのビット数以下)

3. 読み出し/書き込み制御線 (read/write control line) — 読み出しか書き出しかを指定 (1 本)

4. 選択信号線 (select signal line, chip select line) — 読み出しや書き込みのタイミングを決定 (1 本).

10 数の表現

10.1 整数 (integer), 自然数

1. **符号+絶対値表現** (*signed magnitude representation*) — 符号に 1 ビット割り当てる.

2. **1 の補数表現** (*one's complement representation*) — 絶対値を 2 進数で表現し, 負の数であれば 0 と 1 を反転.

3. **2 の補数表現** (*two's complement representation*) — 1 の補数表現に 1 を加える.

4. **n 増し表現** (*n-biased representation*) — 正のみ

[**練習問題**] 上記の整数表現法を比較せよ.

表 2: 負の数の表現法

10 進数	符号+絶対値表現	1 の補数表現	2 の補数表現	7 増し表現	10 進数	符号+絶対値表現	1 の補数表現	2 の補数表現	7 増し表現
8	—	—	—	1111	-8	—	—	1000	—
7	0111	0111	0111	1110	-7	1111	1000	1001	0000
6	0110	0110	0110	1101	-6	1110	1001	1010	0001
5	0101	0101	0101	1100	-5	1101	1010	1011	0010
4	0100	0100	0100	1011	-4	1100	1011	1100	0011
3	0011	0011	0011	1010	-3	1011	1100	1101	0100
2	0010	0010	0010	1001	-2	1010	1101	1110	0101
1	0001	0001	0001	1000	-1	1001	1110	1111	0110
0	0000	0000	0000	0111	0	1000	1111	0000	0111

10.2 実数 (real number) — IEEE, URR

$(-1)^s \times m \times b^e$ 　ただし, m: 仮数 (mantissa, significand), b: 基数 (base), e: 指数 (exponent).

1. **固定小数点表示** (*fixed point number*) — 整数もこの表現法の 1 種
2. **浮動小数点表示** (*floating point number*)

   仮数の MSB が 1 である浮動小数点表現を**正規化** (normalized) 浮動小数点表現と言う. (下線部は 2 進数)

   $15 = \underline{1.111} \times 2^{\underline{11}}$, $-129 = -\underline{1.0000001} \times 2^{\underline{111}}$, $0.125 = \underline{1.0} \times 2^{\underline{-11}}$, $0.015625 = \underline{1.0} \times 2^{\underline{-110}}$

浮動小数点表現の標準規格: IEEE 754 floating-point standard

32 ビット表現

1 ビット	8 ビット	23 ビット
符号 s	指数 e	仮数 m

は $(-1)^s \times (1+m) \times 2^{(e-127)}$ を表現.

$$\begin{cases} e = 255, m \neq 0 : & \text{非数} (Nan) \\ e = 255, m = 0 : & (-1)^s \times \infty \\ 0 < e < 255 : & (-1)^s \times 2^{e-127} \times \underline{1.m} \\ e = 0, m \neq 0 : & (-1)^s \times 2^{-126} \times \underline{0.m} \\ e = 0, m = 0 : & (-1)^s \times 0 \end{cases}$$

64 ビット表現

1 ビット	11 ビット	52 ビット
符号	指数	仮数

は $(-1)^s \times (1+m) \times 2^{(e-1023)}$ を表現.

10.3 IEEE 754 での例外処理

- **桁あふれ** (*overflow*) : 演算結果が決められたビット数で表現できず.
- **アンダーフロー** (*underflow*) : 値の絶対値が小さすぎて表現できず.

問 1

1. 43267 を 2 進数, 7 進数で表現せよ.
2. 0.05078125 を 2 進数で表現せよ.
3. 43267 を浮動小数点 (2 進数) で表せ.
4. -0.75 を浮動小数点 (2 進数) で表せ.

問 2 次の IEEE 754 浮動小数点数は 10 進数でいくらか.

1. 01000011001101100000000000000000
2. 10000000110100000000000000000000

11 情報交換用符号

- **ASCII 符号** (American Standard Code for Information Interchange) — 7 ビット符号
- **EBCDIC 符号** (Extended Binary Coded Decimal Interchange Code) — 8 ビット符号
- **JIS8 ビット符号** — ASCII 符号 + 半角かな文字
- **JIS 漢字符号** — 2 バイト使用 JIS X 0208 ('83)

  ASCII 符号とは Shift-In/Shift-Out の escape sequence (ESC + 2 バイト) により区別.
- **EUC-JP 符号** (Extended Unix Code) — 8 ビット. MSB セットで 2 バイト符号. ISO-2022. (G0: ASCII, G1: JIS X 0208-1983, JIS X 208-1990).

- **ISO-2022-JP 符号** — MIME 文字セット.

  `ESC ( B` ⇒ ASCII,

  `ESC ( J` ⇒ JIS X 0201-1976 (ローマ字),

  `ESC $ @` ⇒ JIS X 0208-1978 (7 ビット × 2),

  `ESC $ B` ⇒ JIS X 0208-1983 (7 ビット × 2).
- **シフト JIS 漢字符号** — 2 バイト (MS-DOS 用)
- **UNICODE** — 2 バイト (Windows NT, ISO10646)

11.1 バイト順序 (Byte Order) .... 要注意

Big-Endian バイト番号

0	1	2	3

例: IBM 360, HP, SUN, PC/RT, MC, News, SGI

Little-Endian バイト番号

3	2	1	0

例: Vax, DECstation, 80x86

ビットパターンには固有の意味はない. それに対する命令によってその意味が決まる. 計算機中の数と実世界の数との違いは, [ ].

「**プログラム内蔵式** (*stored-program*) **の概念**」とは

計算機は 2 つの原則で構成されている: (1) 命令は数として表現. (2) プログラムは数としてメモリに格納されアクセスされる.

12 命令の形式 (instructuction format)

命令セットの設計でのポイント

- **実行時間** — ある仕事をするための命令数, 各命令の実行時間, 主記憶装置へのアクセス回数
- **記憶装置に利用効率** — 各命令のビット数, ある仕事をするための命令数
- **ハードウェアのコスト**— 中央処理装置のハードウェアの素子数, 規則構造の割合

命令 (instruction) = 命令コード (instruction code) + オペランド (operands)

命令形式の分類 — オペランドの種類で分類する

3 番地命令形式 (3-addr instruction format) — RISC

命令コード	番地指定オペランド 1	番地指定オペランド 2	番地指定オペランド 3

2 番地命令形式 (2-addr instruction format)

メモリメモリ間命令形式 (Memory to Memory instruction format, MM 形式)

1 番地命令形式 (1-addr instruction format)

アキュムレータ (Accumulator) を使用.

0 番地命令形式 (0-addr instruction format)

汎用レジスタ向き命令形式 (general register-oriented instruction format)

命令コード	レジスタ指定	番地指定オペランド

レジスタメモリ間命令形式 (Register to Memory instruction format, RM 形式)

命令コード	レジスタ指定 1	レジスタ指定 2

レジスタレジスタ間命令形式 (Register to Register instruction format, RR 形式)

以下の例題：$\mathbf{A} \longleftarrow (\mathbf{B}) \times (\mathbf{C}) - (\mathbf{D}) \times (\mathbf{E})$

左側が命令セット, 右側が例題のコーディング

12.1 3 番地命令形式

`ADD A B C`	$(A)+(B) \to C$	$\mathbf{A} \longleftarrow (\mathbf{B}) \times (\mathbf{C}) - (\mathbf{D}) \times (\mathbf{E})$	
`SUB A B C`	$(A)-(B) \to C$	`MUL B C T1`	$(B) \times (C) \to T1$
`MUL A B C`	$(A)\times(B) \to C$	`MUL D E T2`	$(D) \times (E) \to T2$
`DIV A B C`	$(A)\div(B) \to C$	`SUB T1 T2 A`	$(T1) \times (T2) \to A$

12.2 2 番地命令形式

`ADD A B`	$(A)+(B) \to A$	`MOV A B`	$(B) \to A$
`SUB A B`	$(A)-(B) \to A$	`MUL A C`	$(A) \times (C) \to A$
`MUL A B`	$(A)\times(B) \to A$	`MOV T D`	$(D) \to T$
`DIV A B`	$(A)\div(B) \to A$	`MUL T E`	$(T) \times (E) \to T$
`MOV A B`	$(B) \to A$	`SUB A T`	$(A) - (T) \to A$

12.3 1 番地命令形式

		`LOAD D`	$(D) \to Acc$
`LOAD A`	$(A) \to Acc$	`MUL E`	$(Acc) \times (E) \to Acc$
`STORE A`	$(Acc) \to A$	`STORE A`	$(Acc) \to A$
`ADD A`	$(Acc)+(A) \to Acc$	`LOAD B`	$(B) \to Acc$
`SUB A`	$(Acc)-(A) \to Acc$	`MUL C`	$(Acc) \times (C) \to Acc$
`MUL A`	$(Acc)\times(A) \to Acc$	`SUB A`	$(Acc) \times (A) \to Acc$
`DIV A`	$(Acc)\div(A) \to Acc$	`STORE A`	$(Acc) \to A$

12.4 0 番地命令形式

PUSH	A $(SP)+1 \to SP; (A) \to (SP)$
POP	A $((SP)) \to A; (SP)-1 \to SP$
ADD	スタックトップ 2 つを pop し, 和を push
SUB	スタックトップ 2 つを pop し, 差を push
MUL	スタックトップ 2 つを pop し, 積を push
DIV	スタックトップ 2 つを pop し, 商を push

```
PUSH B
PUSH C
MUL
PUSH D
PUSH E
MUL
SUB
POP A
```

12.5 汎用レジスタ向き命令形式

LOAD r1 A	$(A) \to r1$	LOAD r0 B	$(B) \to r0$
STORE r1 A	$(r1) \to A$	LOAD r1 C	$(C) \to r1$
ADD r1 r2	$(r1)+(r2) \to r1$	MUL r0 r1	$(r0) \times (r1) \to r0$
SUB r1 r2	$(r1)-(r2) \to r1$	LOAD r1 D	$(D) \to r1$
MUL r1 r2	$(r1) \times (r2) \to r1$	LOAD r2 E	$(E) \to r2$
DIV r1 r2	$(r1) \div (r2) \to r1$	MUL r1 r2	$(r1) \times (r2) \to r1$
		SUB r0 r1	$(r0)-(r1) \to r0$
		STORE r0 A	$(r0) \to A$

LOAD r1 A	$(A) \to r1$	LOAD r0 B	$(B) \to r0$
STORE r1 A	$(r1) \to A$	MULM r0 C	$(r0) \times (C) \to r0$
ADDR r1 r2	$(r1)+(r2) \to r1$	LOAD r1 D	$(D) \to r1$
ADDM r1 A	$(r1)+(A) \to r1$	MULM r1 E	$(r1) \times (E) \to r1$
SUBR r1 r2	$(r1)-(r2) \to r1$	SUB r0 r1	$(r0)-(r1) \to r0$
SUBM r1 A	$(r1)-(A) \to r1$	STORE r0 A	$(r0) \to A$
MULR r1 r2	$(r1) \times (r2) \to r1$		
MULM r1 A	$(r1) \times (A) \to r1$		
DIVR r1 r2	$(r1) \div (r2) \to r1$		
DIVM r1 A	$(r1) \div (A) \to r1$		

12.6 (g) 命令セットの評価

評価のための仮定

- 主記憶の語数は 16M 語. 24 ビット番地つけ.
- 命令コードに 4 ビット必要.
- 汎用レジスタ計算機では汎用レジスタの数は 16 個. レジスタ指定に 4 ビット必要.

ベンチマークプログラム： $A \longleftarrow (B) \times (C) - (D) \times (E)$ による命令セットの評価

命令形式		3 番地	2 番地	1 番地	スタックマシン	汎用レジスタ (1)	汎用レジスタ (2)
命令ステップ数		3	5	7	8	8	6
メモリアクセス回数	命令	3	5	7	8	8	6
	データ	9	13	7	5	5	5
	合計	12	18	14	13	13	11
各命令のビット数		76	52	28	28 4	32 12	32 12
プログラムの合計ビット数		228	260	196	152	196	172

12.7 RISC と CISC プロセッサ

CISC (*Complex Instruction Set Computer*)
命令語の複雑さ: 16bit プロセッサ MC68000 114 種類, 32bit プロセッサ MC68020 139 種類

VLWI (*Very Long Word Instruction*)

RISC (*Reduced Instruction Set Computer*)
実現を容易にし, かつ, ソフトウェアの改良と相まって総合性能の向上を目指す.

1. 1 マシンサイクルで実行可能な単純な命令.
2. 制御系はハードワイアード, マイクロプログラミングが不要.
3. 命令語長が一定 (固定フォーマット命令).
4. レジスタの数が多い (高速命令, キャッシュ).
5. メモリアクセス命令を限定.
6. パイプライン制御方式+遅延分岐方式 (delayed jump).

12.8 制御構造

- **サブルーティン**値渡し (call by value), 番地渡し (call by address)

  パラメータの渡し方 — レジスタ, 固定番地, JSR 命令の次の番地に置く, スタック
- **再入可能性** 直列再使用可能 (serially reusable), 並列再使用可能 (parallelly reusable, reentrant)
- **再帰的呼び出し (recursive call)** 自分自身を直接・間接に呼び出す.
- **コルーティン (coroutine)** (1) 互いに呼び合う (2) 実行が前回中断したところから再開.

12.9 ブートストラッピング

靴紐を最初に通し，次により大きなものを通す手法.

1. 一次ブートローダ (Initial Program Loader)
   - **BIOS (Basic Inut/Output System)**
   - **UEFI (Unifired Extensible Firmware Interface)**
   - **起動ドライブ**
     - MBR (Master Boot Record) 32bit 管理 (512byte/sector * 2^32 = 2.2TB
     - GPT (GUIDPartition Table) 64bit 管理
2. 二次ブートローダ

   GNU GRUB, Windows BOOTMGR, NTLDER

表 3: 覚えておくとよい経験法則

(1) **アムダールの法則:** バランスのとれたシステムでは, 1MIPS (*Million Instructions Per Second*) 毎に 1M バイトの主記憶と 1M ビット/秒の入出力データ転送率が必要である.

(2) **90/10 局所性の法則:** プログラム実行時間の 90%は, 全プログラムの 10%のコードで占められる.

(3) **DRAM 成長の法則:** DRAM の集積度は毎年 60%向上する. 3 年で 4 倍.

(4) **ディスク成長の法則:** ディスクの記録密度は毎年 25%向上する. 3 年で 2 倍.

(5) **アドレス消費の法則:** 平均的プログラムで必要なメモリ量は毎年 1.5 倍〜2 倍増加する. すなわち, アドレス・ビットは毎年 1/2〜1 ビット増加する.

(6) **90/50 分岐条件成立の法則:** 前に戻る分岐条件では 90%成立し, 先に進む分岐条件では 50%が成立する.

(7) **2 対 1 キャッシュの法則:** サイズが X のダイレクトマッピング方式のキャッシュのミス率は, サイズが X/2 の 2 ウェイセットアソシアティブ (2-way set associative) 方式のキャッシュのミス率とほぼ同じである.

13 記憶システム

高速大容量の主記憶装置を作りたい

技術装置	アクセス時間 '93 頃	アクセス時間 '08	$/MByte '93	$/GByte '08
ALU	数 10 ns	0.15 – 0.30ns	(演算速度)	
SRAM	8–35 ns	0.5 – 15ns	$100–400	$2K–5K
DRAM	90–120 ns	30 – 2000 ns	$25–50	$20 – 75
HDD	10–20 ms	5–20ms	$1–2	$0.2 – 2

- メモリ装置の**階層化** ⟹ **キャッシュ記憶・仮想記憶**
- レジスターファイル, SRAM, DRAM, SSD, HDD をうまく使いこなす.

局所性原理 (Principle of Locality)

- **時間的局所性** (temporal locality) — 参照されたら, すぐにもう一度参照される.
- **空間的局所性** (spatial locality) — 参照されたら, その近辺もすぐに参照される.

問題: 上記の 2 つの局所性の例を考えよ.

1. ブロックサイズとミスペナルティの関係
2. ブロックサイズとミスレートの関係

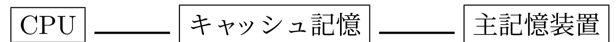
⟹ ブロックサイズと平均アクセス時間の関係
(ミスペナルティ — アクセス時間 + 転送時間)

共通の問題点

1. **Block placement** — ブロックを上位のどこに置けるのか
2. **Block identification** — 上位にブロックがあればどのように見つけるのか.
3. **Block replacement** — ミスがあったときにブロックをどこに置くのか.
4. **Write strategy** — 書き込みをどうするのか.

13.1 キャッシュ記憶 (cache memory)

13.1.1 キャッシュ記憶の基本原理



キャッシュ記憶の内容: (主記憶装置の番地, その番地の内容を保持するセル)

例: 主記憶装置の容量は 16M バイト, アクセス時間は 400 ナノ秒. キャッシュ記憶の容量は 64K バイト, アクセス時間は 40 ナノ秒.

キャッシュヒット率 (cache hit ratio) : 主記憶装置に対して, その番地の内容がキャッシュ記憶にある (ヒットする) 確率.
キャッシュミス: キャッシュ記憶になかったとき.

キャッシュ記憶アクセス速度が主記憶装置より 10 倍早く, キャッシュヒット率が 90%としよう. このとき主記憶装置の実効アクセス時間は

$((40 \times 0.9 + 400 \times 0.1) \div 400) \times 100 = 19$

より, 主記憶アクセス時間の 19%となる. 95%のキャッシュヒット率の場合には 14.5% となる.
[注意 1] キャッシュ記憶のサイズとその効果の関係は, プログラムによって変わる.
[注意 2] RISC プロセッサでの命令・データキャッシュ

13.1.2 キャッシュ記憶主記憶間のデータ交換

アクセスが書き込みで, キャッシュがヒットした場合:

1. **ストアスルー** (store through, write through) — キャッシュと主記憶と同時に書き込む.
2. **ストアイン** (store in, write in, write back) — キャッシュにしか書き込まない. キャッシュブロックの追い出しが起ったときに主記憶へ始めて書き出す. 書き出しビット (dirty bit) が必要.

キャッシュがミスした場合:

1. ストアスルー — 主記憶にだけ書き込む.
2. ストアイン — 適当なキャッシュブロックの追い出し, その番地を含むブロックをキャッシュに読み込み, 変更する.

キャッシュ記憶の構造

キャッシュブロックアドレスアレイ	キャッシュブロックアレイ
キャッシュブロックの先頭番地 (主記憶装置での)	キャッシュブロック 2^n バイトの連続領域
下位ビットは省略 (4, 5, 6 ビット)	16, 32, 64 バイトのどれか

例: キャッシュブロックを 16 バイト, キャッシュ全体を 64K バイトとすると, キャッシュブロックアドレスアレイとキャッシュブロックアレイは各々 4K 個のエントリを持つ. また, 主記憶の番地空間が 2^{24} とすれば, キャッシュブロックアドレスは 20 ビット

キャッシュヒット/ミスのチェック:

- **Full associative mapping** — すべてのエントリと比較. 4K 個の比較器を使用.
- **Direct mapping** — 主記憶装置を 16 バイトのブロックに分割し, それを 4K で割り, その余りをエントリの位置とする. (各エントリには定まった 256 個のブロックが割当てられる).
- **Set associative mapping** — キャッシュを n グループ分割. 主記憶装置を 16 バイトのブロックに分割し, それを (4K/n) で割った余りをエントリとして, どれかのグループに割当てる.

$n = 4K \Longrightarrow$ full associative mapping
$n = 1 \Longrightarrow$ direct mapping

Set associative mapping 用追い出しアルゴリズム

1. **FIFO (First-In First-Out) 法** —
   最も古いものから追い出す.
2. **LRU (Least Recently Used) 法** —
   最近使われなくなったものを追い出す.

いずれの場合にもキャッシュブロックアドレスアレイの各エントリに追い出し情報が必要となる.

[宿題] 3 つのマッピング方式の長所・短所を比較せよ.

メモリインターリーブ (memory interleave)

— キャッシュと主記憶間での並列データ転送による高速化. 主記憶を分割し, 分割間で同時的に読み書きを行う.
大型計算機: 分割数=キャッシュブロックの大きさ.

キャッシュのミスの原因 — 3 つの C

1. **強制ミス (Compulsory misses)**: ブロックへの第 1 回目のアクセス. コールドスタートミス (cold start misses) ともいう.
2. **容量ミス (Capacity misses)**: プログラムに必要なすべてのブロックがキャッシュに入らない時. 捨てられたブロックに再度アクセス.
3. **競合ミス (Conflict misses)**: ブロック置換戦略が連想写像や直接写像の時.

設計変更	ミス率への影響	性能低下の可能性
サイズ ↗	容量ミス ↘	アクセス時間 ↗
連想性 ↗	競合ミスのミス率 ↘	アクセス時間 ↗
ブロックサイズ ↗	ブロックサイズの広い幅に対するミス率 ↘	ミスペナルティ ↗

13.2 セグメンテーションによる仮想記憶

仮想記憶 (virtual memory): 主記憶装置を見かけ上その物理的な大きさよりも大きく見せる方法.

13.2.1 セグメンテーションによる仮想記憶

論理番地 = セグメント番号 + セグメント内変位
セグメント表 = (存在ビット, サイズ, セグメントの先頭物理番地)

オンデマンドフェッチ (on-demand fetching) : 要求があってからスワップインが開始される.

多重プログラミング環境では, セグメントエラーが生じるとスワップインの間に別のプロセスを起動する. (**プロセススイッチ**が起る.)

多重仮想空間 — ジョブ毎にセグメント表を持つ.

記憶管理システム プログラム実行中に主記憶にないセグメントをアクセスするとセグメントエラーの割込が生じ, 呼ばれる.

メモリマップ表 (memory map table) : 主記憶の使用状況を管理

追い出しアルゴリズム (replacement algorithm)

1. 優先順位の低いプロセスのセグメントを追い出す
2. セグメントエラー以外の理由で待ち状態にあるプロセスのセグメントを追い出す
3. 上記以外, スワップインに必要な領域ができるまで待つ.

位置決めアルゴリズム (placement algorithm)

1. **First fit algorithm**: 主記憶の最も先頭にある空き領域に置く
2. **Cyclic first fit algorithm**: 前回スワップインを行った場所から最も近い空き領域に置く
3. **Best fit algorithm**: 最小の空き領域に置く

フラグメンテーション (memory fragmentation) : 主記憶内に小さ過ぎる未使用部分が多数出る現象. 外部フラグメンテーションともいう. 内部フラグメンテーションは, 仮想記憶でページ内で未使用部分が出る現象.
例: バローズ B5700, DEC PDP-11, Intel 80286/80386, Motorola 68030.

セグメンテーションによる仮想記憶の利点・欠点

- セグメントは意味ある単位 $\Longrightarrow$ セグメントエラーの確率は小さい.
- セグメントを複数のプロセスで共有可 $\Longrightarrow$ プロセス間交信, 単純なプロセスでのデータ共有.
- 実行時にセグメントサイズが可変 $\Longrightarrow$ スタックのセグメント化に.
- 単一レベル記憶の実現: セグメントをファイルと一体化.

- 付加するハードウェアが複雑.
- サイズの大きいセグメント ⟹ 主記憶装置の利用効率が高まらない.
- セグメントの大きさが不揃い ⟹ 位置決めや追い出しアルゴリズムの実行時間が無視できない.
- 主記憶にはスワップインのために常時空き領域がいる.

13.3 ページングによる仮想記憶

ページング (paging) : 主記憶装置を**固定長**の領域 (ページ, page) に分割し, ページ単位でスワップイン・スワップアウトする. (空領域管理不要)
オンデマンドページング (on-demand paging): 要求が発生したときにスワップインを行う.
キャッシュ記憶との違い: ページエラー (page error, ページフォールト, page fault) が発生したときの遅延が 1000 倍 (キャッシュ記憶の場合には 10 倍)
ワーキングセットサイズ (working set size): ある一定数以下のページしかプログラムに割り当てないと, スワップインやスワップアウトが頻繁に生じる (**スラッシング** (thrashing)) ような大きさ.

13.3.1 ページングの機構

番地変換方式　論理番地 ⟶ 物理番地

- **直接写像法 (direct mapping)** 論理ページ分のエントリを持った表. 各エントリには, 主記憶にあるときだけ物理ページ番号が入る.
- **連想写像法 (associative mapping)** 物理ページ分のエントリを持った表. 各エントリには, そのページが含んでいる論理ページ番号が入る.

変換早見表 (TLB, translation-lookaside buffer) キャッシュの一種. 連想記憶ハードウェアの利用.

仮想記憶実現方式　ページ表を与える単位

- 単一仮想空間 (記憶) 方式 — ページ表がシステムに 1 つ. 仮想空間のサイズは一定.
- 多重仮想空間 (記憶) 方式 — ジョブ毎にページ表. ジョブに必要なアドレス空間と同じ仮想空間を持てる. ジョブの削除・生成に対して仮想空間が柔軟に対応可.

追い出しアルゴリズム　ページフォールト時の処理

- **LRU (Least Recently Used) 法**
  物理ページに参照カウンタ, シフトレジスタ
- **擬似 LRU (pseudo LRU) 法** 数ビットの参照カウンタ
- **ランダム (random) 法**

13.3.2 多重レベルページング

大容量論理空間への対応: ページを大ページと小ページに分割. 論理番地は
(大ページ番号, 小ページ番号, ページ内変位)

13.3.3 ページ化セグメンテーション

(セグメント番号, ページ番号, ページ内変位)
例: Multix/GE645, Intel 80386, Motorola MC68030,

13.4 まとめ

キャッシュ, TLB, 仮想記憶に共通の事項

1. **ブロックの置き場所** (*placement*) — 一箇所 (直接写像), 複数 (連想写像), 任意 (完全連想写像)
2. **ブロックを見つけ方** (*identification* —) インデックス付け (例, 直接写像 cache), 限定探索 (例, 連想写像 cache), 全探索 (例, 完全連想写像)
3. **ミス時に追いだしブロックの決定** (*replacement*) LRU (Least Recently Used), ランダム
4. **書き込みの伝播** (*write strategy*) — write through (store through), write back (copy back, store in)

13.4.1 オーバレイ (overlay) — 手動型

プログラムとデータを分割し (**セグメンテーション**), 磁気ディスクなどに取った連続領域 (**スワップ領域** (swap area)) に置く. セグメントの入れ替えを伴うロードを**スワップイン** (swap in), 追い出しを**スワップアウト** (swap out). データセグメントは書き込みがあった場合には, スワップ領域にセグメントを退避. プログラムセグメントは, 読み込みのみ.
メモリアロケーション (memory allocation) : セグメントの置き場所を決める.

リンケージセグメント (linkage segment) : セグメント間の参照を管理する常駐セグメント.
他プログラムセグメントの制御 — 間接番地指定で.
異なるデータセグメントのアクセス — インデックス修飾, ベースレジスタ修飾番地指定で.

オーバーレイ方式の利点・欠点

- スワップイン, スワップアウトの少ないメモリ使用効率の高いプログラムが書ける.
- セグメントのプリフェッチ, プリロードが可能.
- 大規模なプログラムに対して効率のよいメモリアロケーションが難しい.
- リンケージが複雑でデバッグしにくい.