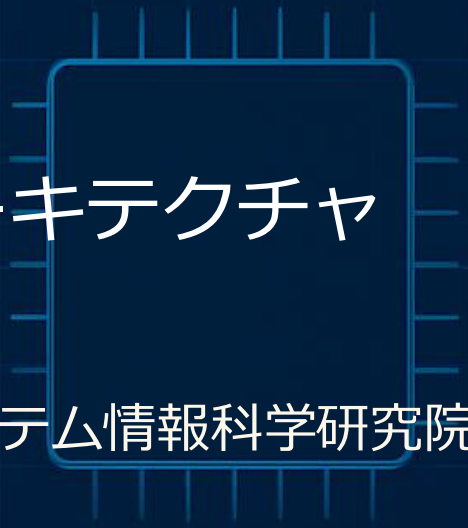
集中講義令和8年2月24日～26日

コンピュータシステム入門



Day 2: コンピュータアーキテクチャ

講師　陳　オリビア　（大学院システム情報科学研究院）

TA　GPT-5 Thinking (OPEN AI)

Gemini 2.5 pro (Google)

CONTENTS

今日のスケジュール

	時間帯	モジュール	タイプ
午前中	10:00 ~ 11:00	デジタル回路の基礎（２）	講義
	11:00 ~ 11:15	Coffee Break	
	11:15 ~ 12:30	コンピューターアーキテクチャ	講義
	12:30 ~ 13:30	Lunch Break	
午後	13:30 ~ 14:30	性能評価とチップ設計	講義
	14:30 ~ 15:00	Verilogで回路を記述してみよう	演習
	15:00 ~ 15:15	Coffee Break	
	15:15 ~ 17:00	Verilogで回路を記述してみよう（継続）	演習

CONTENTS

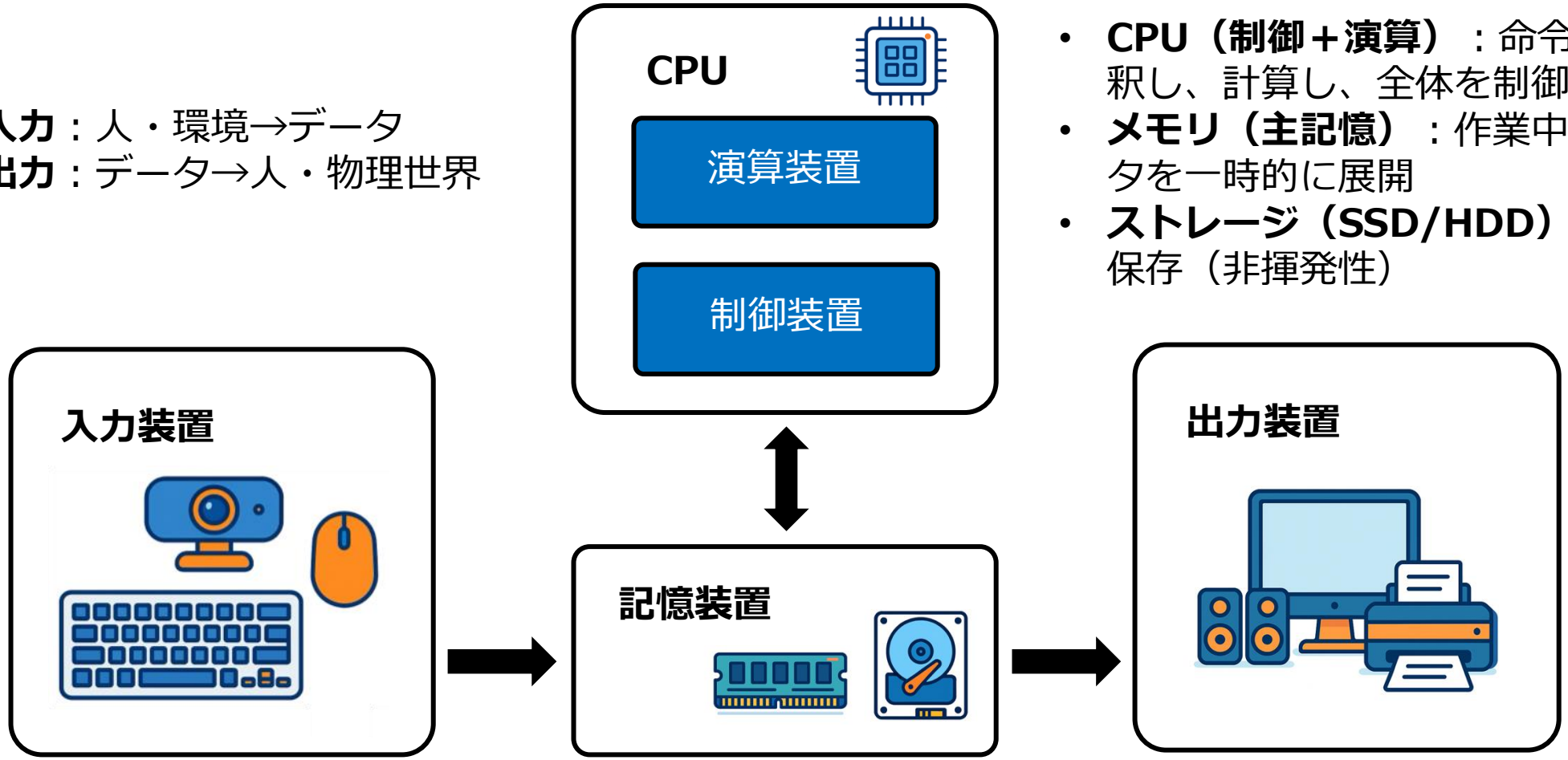
モジュール５：コンピュータアーキテクチャ

CONTENTS

モジュール５：コンピュータアーキテクチャ

ここまでの復習：コンピュータの５大装置

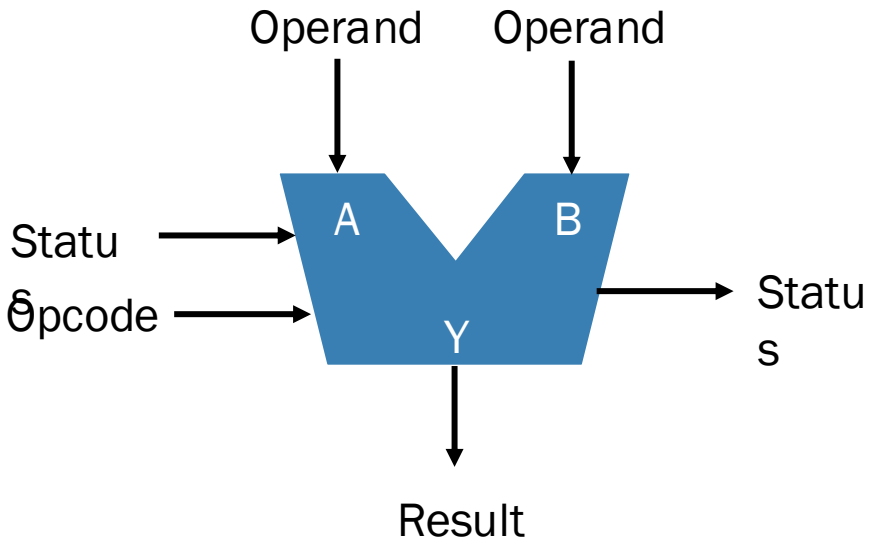
- **入力**：人・環境→データ
- **出力**：データ→人・物理世界



- **CPU（制御＋演算）**：命令を解釈し、計算し、全体を制御
- **メモリ（主記憶）**：作業中のデータを一時的に展開
- **ストレージ（SSD/HDD）**：長期保存（非揮発性）

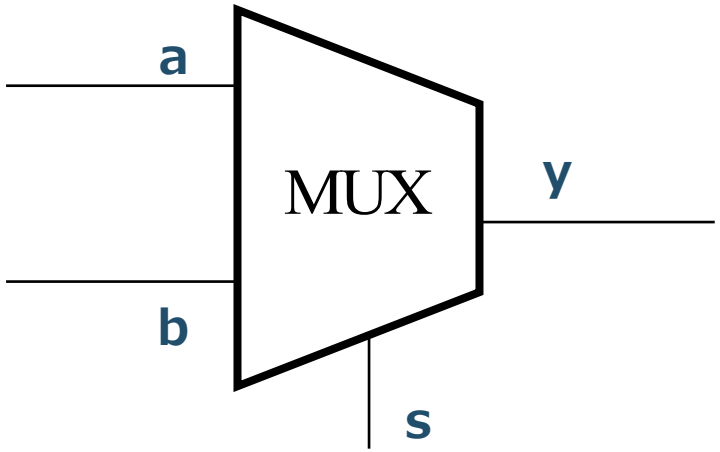
ここまでの復習：組み合わせ回路

計算センター：ALU



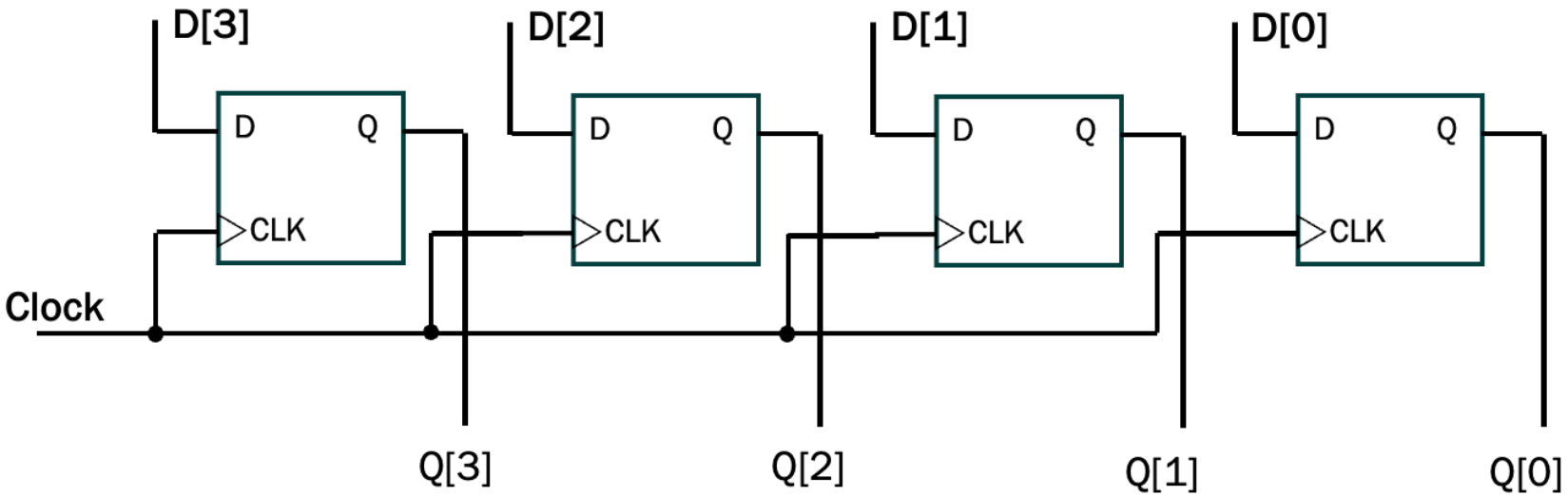
演算

計算の切り替えスイッチ：マルチプレクサ（MUX）



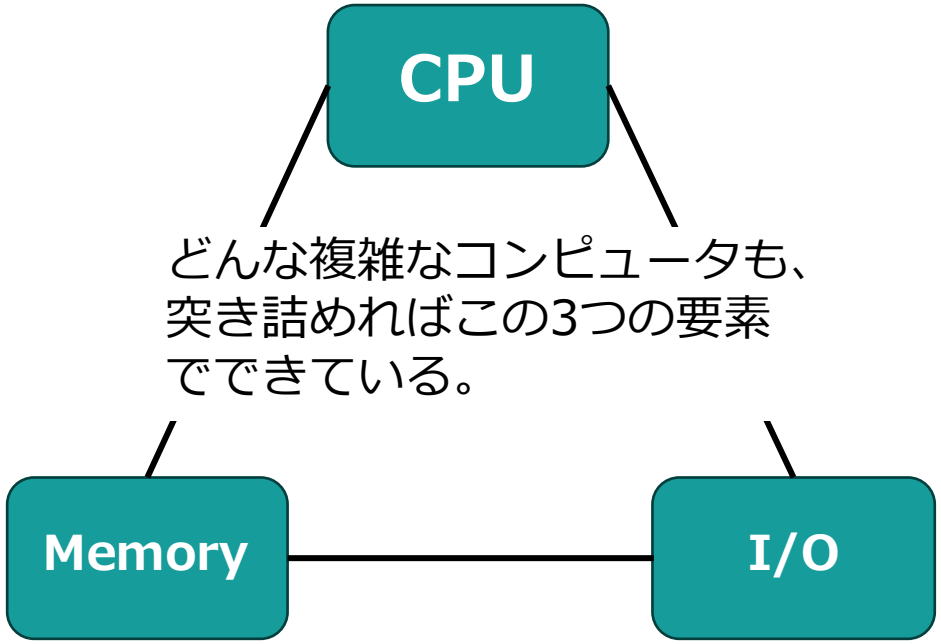
制御

ここまでの復習：順序回路

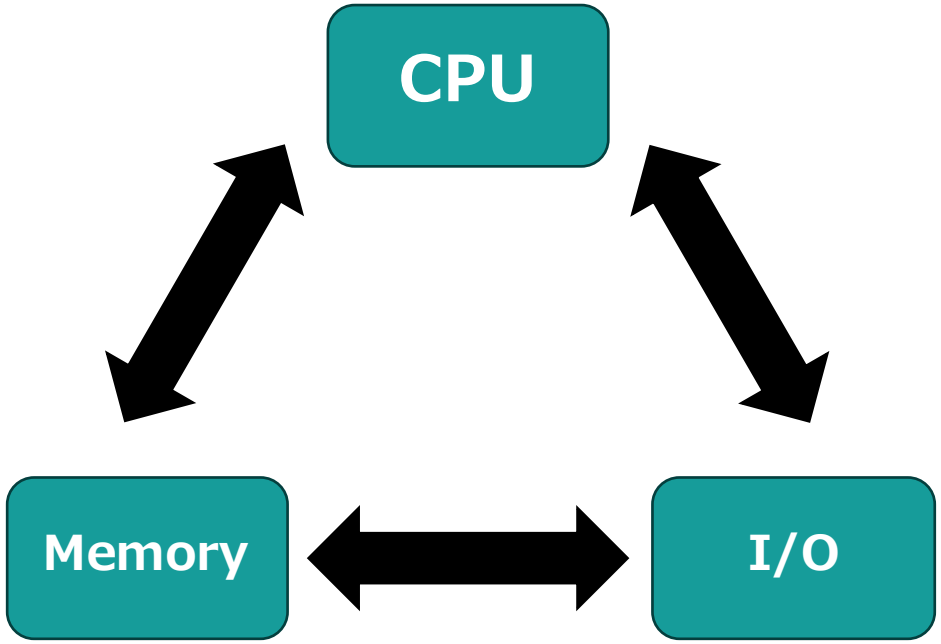


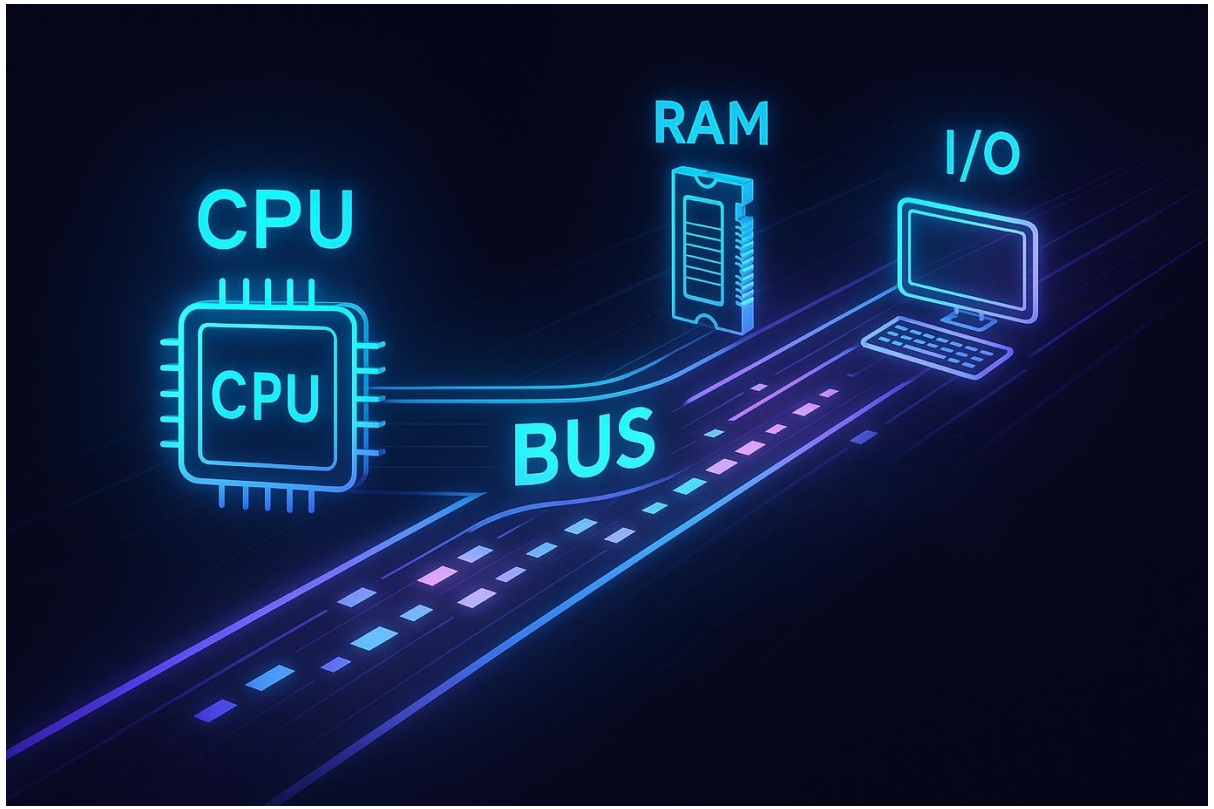
レジスタ：複数のビットをまとめて記憶する装置

コンピュータの三大要素



要素を繋ぐ道「バス」





バスの3つの役割

バスは、役割に応じて3種類に分かれている

アドレスバス

CPUがメモリの「住所」を指定するための道

データバス

CPUとメモリの間で、実際のデータ（中身）が通る道

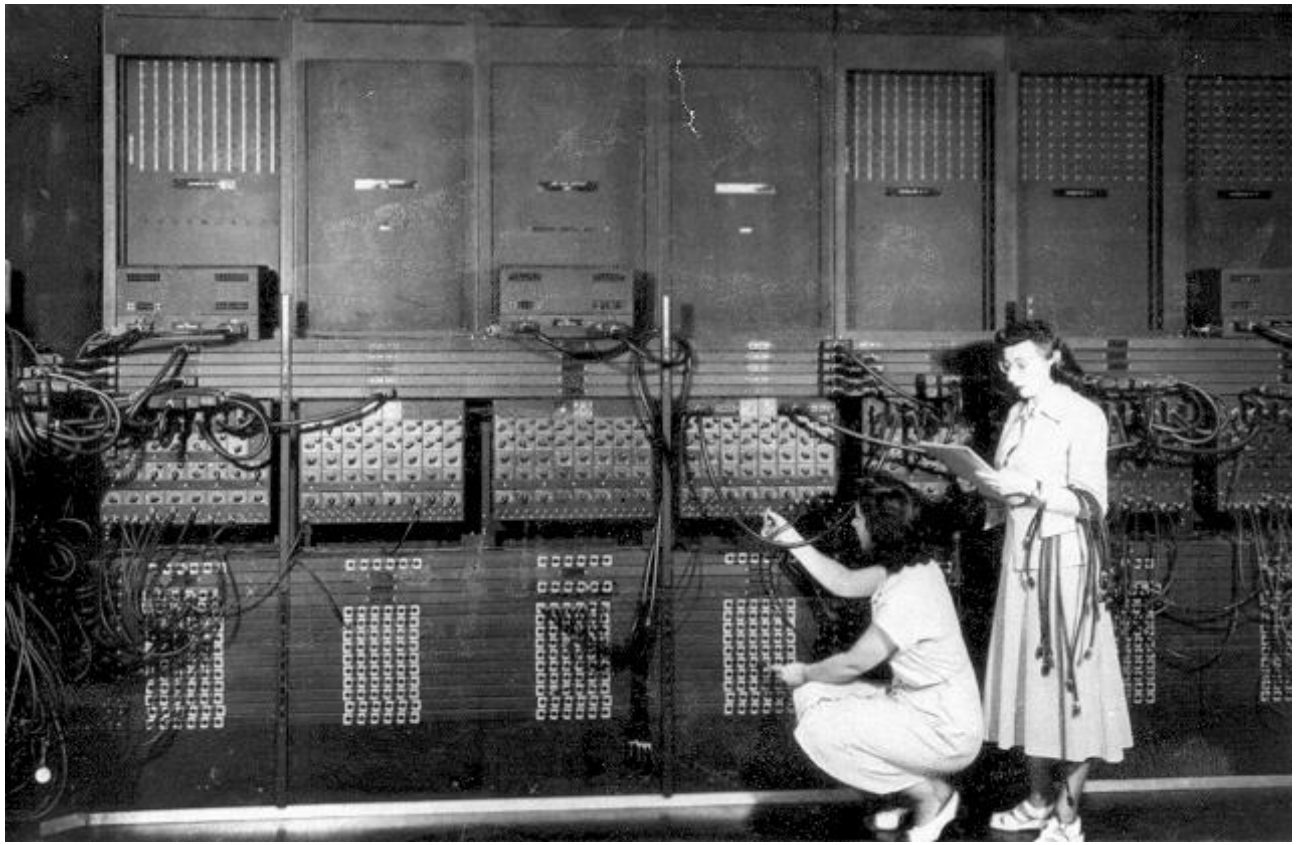
制御バス

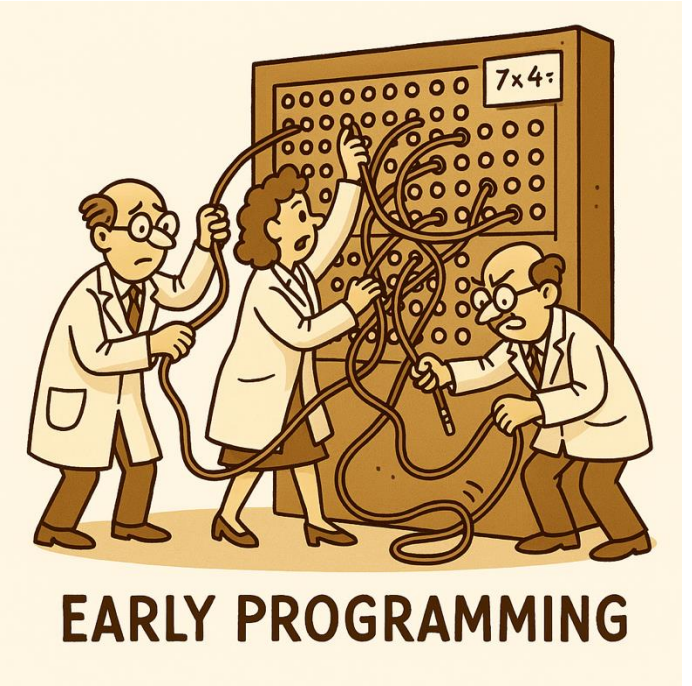
「今から読み込みます」「書き込みます」といった制御信号が通る道

CONTENTS

モジュール５：コンピュータアーキテクチャ

初期コンピュータの悩み



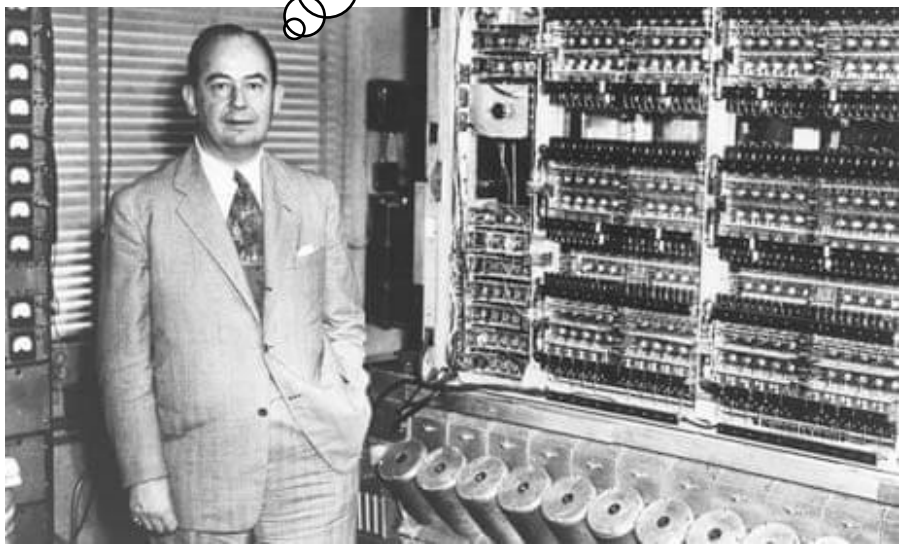


- **ENIACの「再プログラミング」**
- 計算内容を変えるたびに、数日かけて物理的に配線を繋ぎ変える必要があった。

天才のひらめき

「プログラム（命令）も、データ（数値）も、同じ『数字』じゃないか？」

Program = Data



- ジョン・フォン・ノイマン (1903-1957)

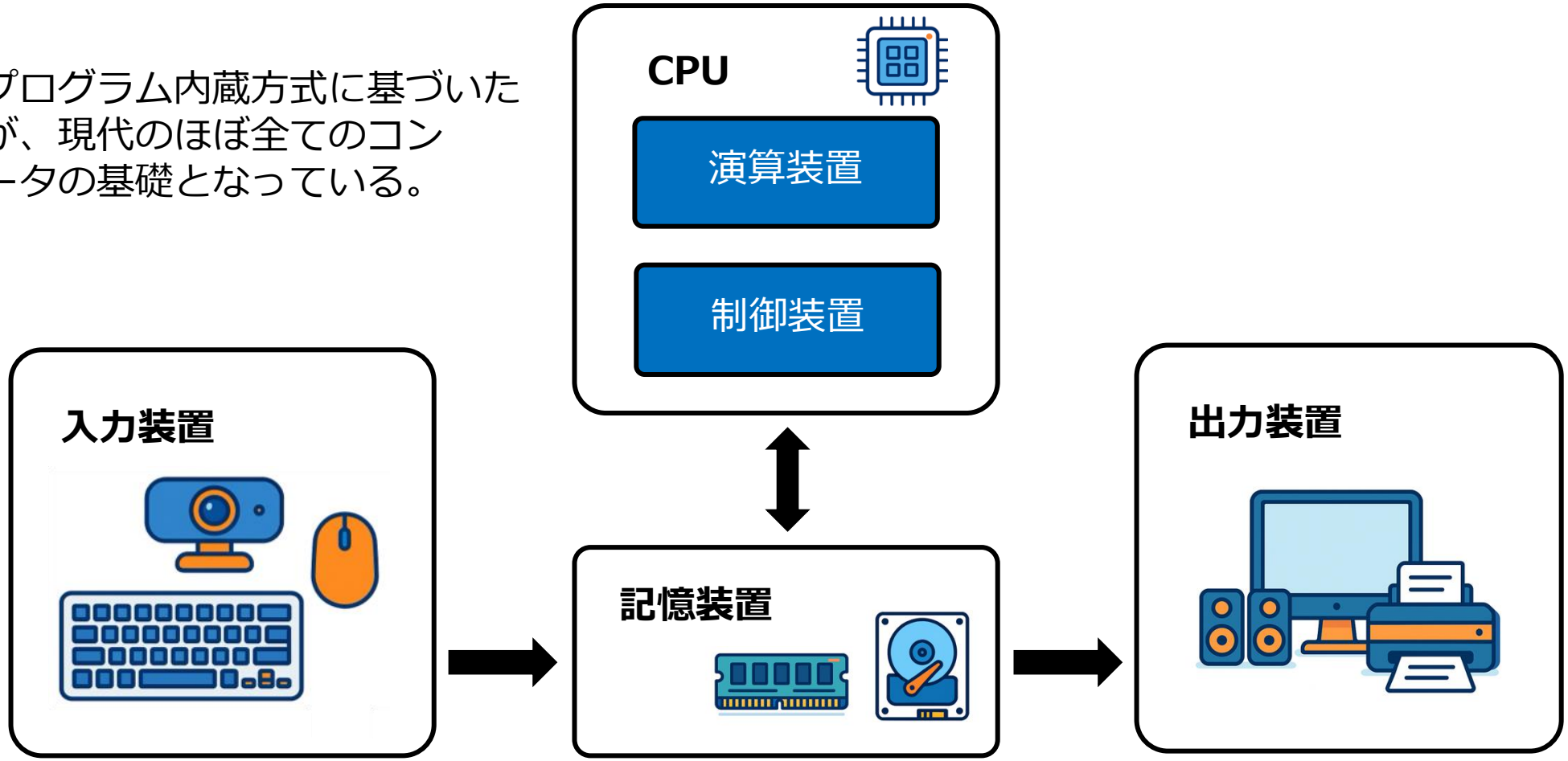
プログラム内蔵方式
(ストアドプログラム方式)

ストアドプログラム方式

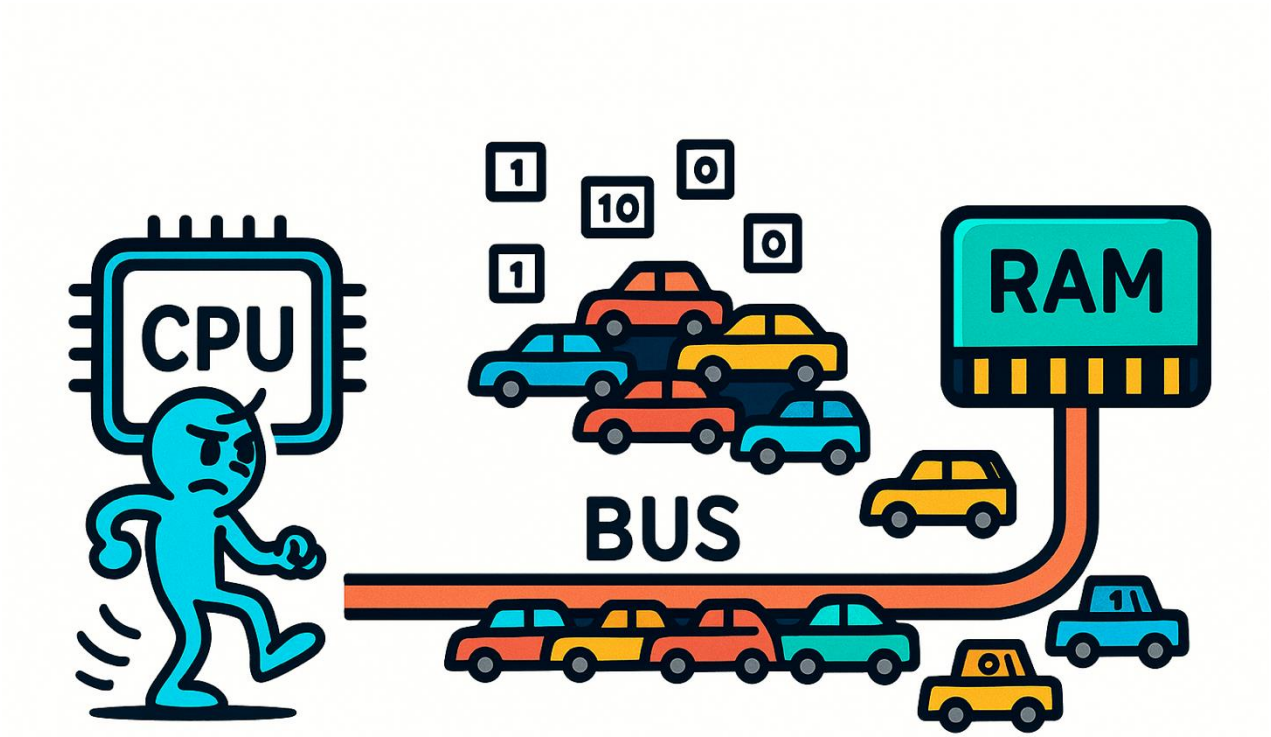
- **プログラム内蔵方式（ストアードプログラム方式）**
- プログラムとデータを、同じメモリに保存する。
- 配線を変更せず、ソフトウェアを書き換えるだけで、コンピュータが全く違う仕事をできるようになった！

フォン・ノイマン型アーキテクチャ

このプログラム内蔵方式に基づいた設計が、現代のほぼ全てのコンピュータの基礎となっている。



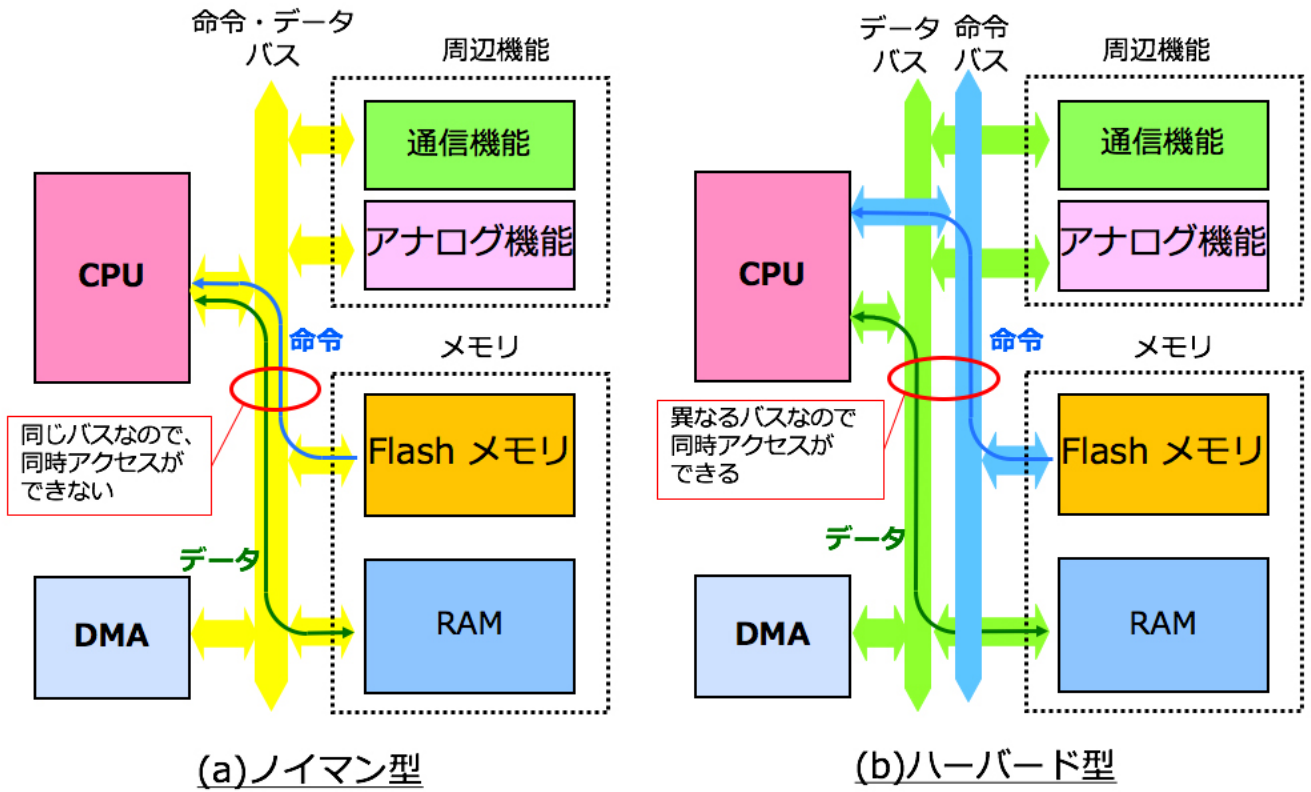
フォン・ノイマン・ボトルネック



- **弱点：フォン・ノイマン・ボトルネック**
- CPUの処理速度に比べ、メモリとのデータのやり取り（バス）が遅い

発展：ハーバードアーキテクチャ

https://edn.itmedia.co.jp/edn/articles/1703/21/news021.html#l_tt170321_QA36_01.jpg

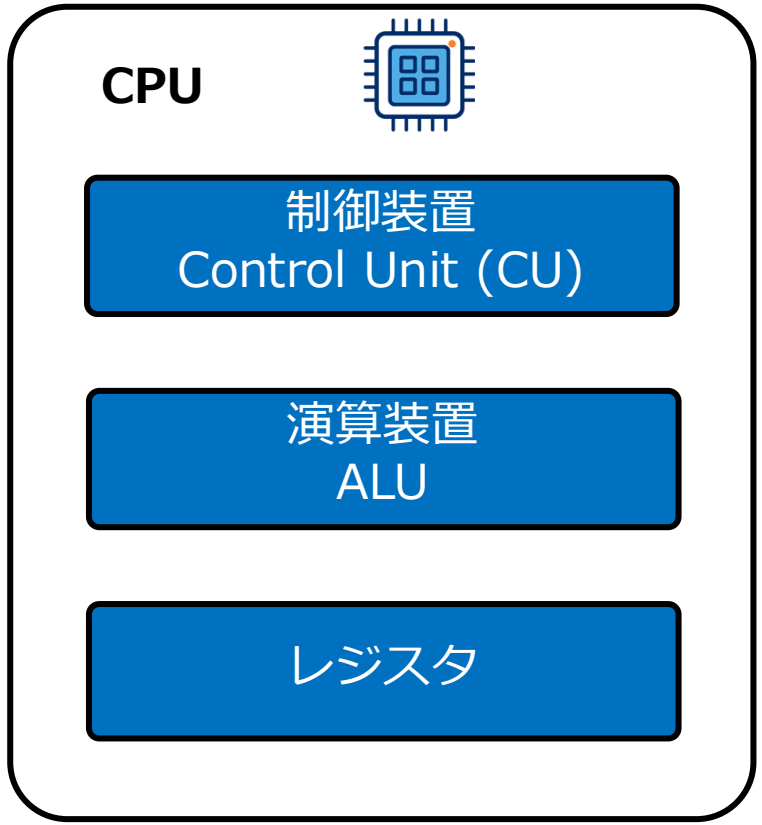


- 命令とデータを物理的に別のバスでやり取りする方式
- ボトルネックを緩和できるため、一部の高性能プロセッサで採用されている

CONTENTS

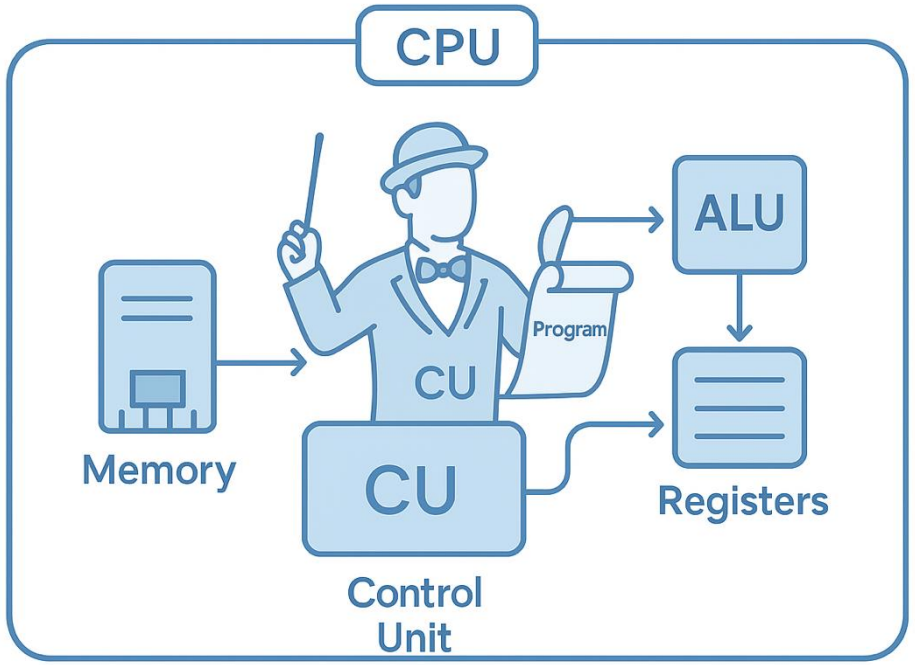
モジュール５：コンピュータアーキテクチャ

CPUの内部を覗く



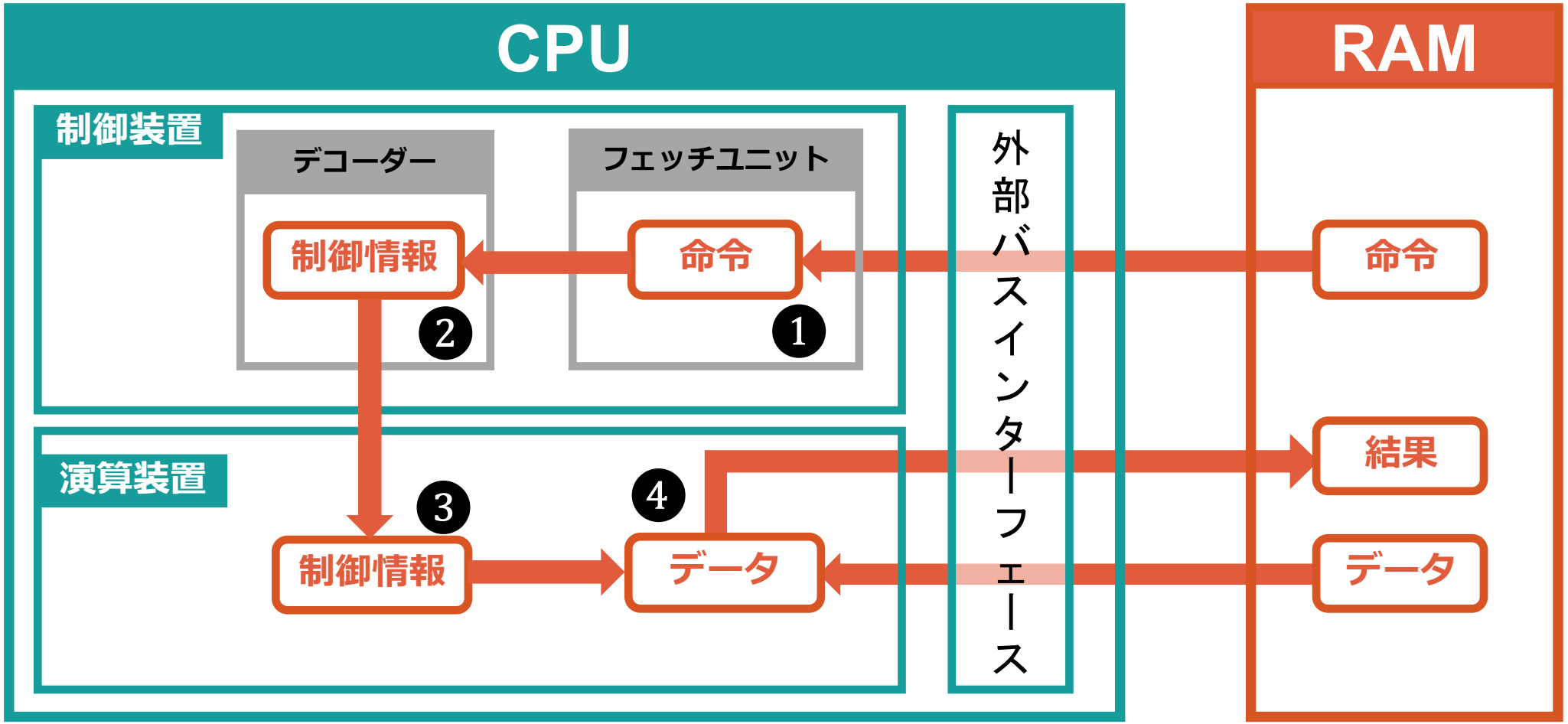
- **制御装置 (CU):** 命令を解読し、各装置に指示を出す「司令塔」
- **演算装置 (ALU):** 実際に計算や比較を行う「実行部隊」
- **レジスタ:** CPU内部の超高速な「一時記憶場所」

CPUはどうやってプログラムを実行するのか？



- CPUの仕事は、メモリに書かれたプログラム（命令のリスト）を、1行ずつ正確に、そして超高速に実行すること。
- そのために、CPUは時計の針のように正確な「命令サイクル」と呼ばれる手順を繰り返している。

CPUの仕事サイクル（命令サイクル）



CPUの仕事サイクル（命令サイクル）

❶命令フェッチ：Instruction Fetch (IF)

❷命令解読：Instruction Decode (ID)

❸命令実行：Execute (EX)

❹結果格納：Write Back (WB)

❶命令フェッチ：Instruction Fetch (IF)

制御装置(CU)

プログラムカウンタ(PC)

次に読むべきページ

❶命令フェッチ：Instruction Fetch (IF)

「しおり」が指すページをレシピ本から取り寄せる

⇩

次の命令を読み出す

取り寄せたレシピの1行を、目の前の「レシピ立て」に置く

⇩

読み出した命令を命令レジスタにコピー

❷命令解読：Instruction Decode (ID)

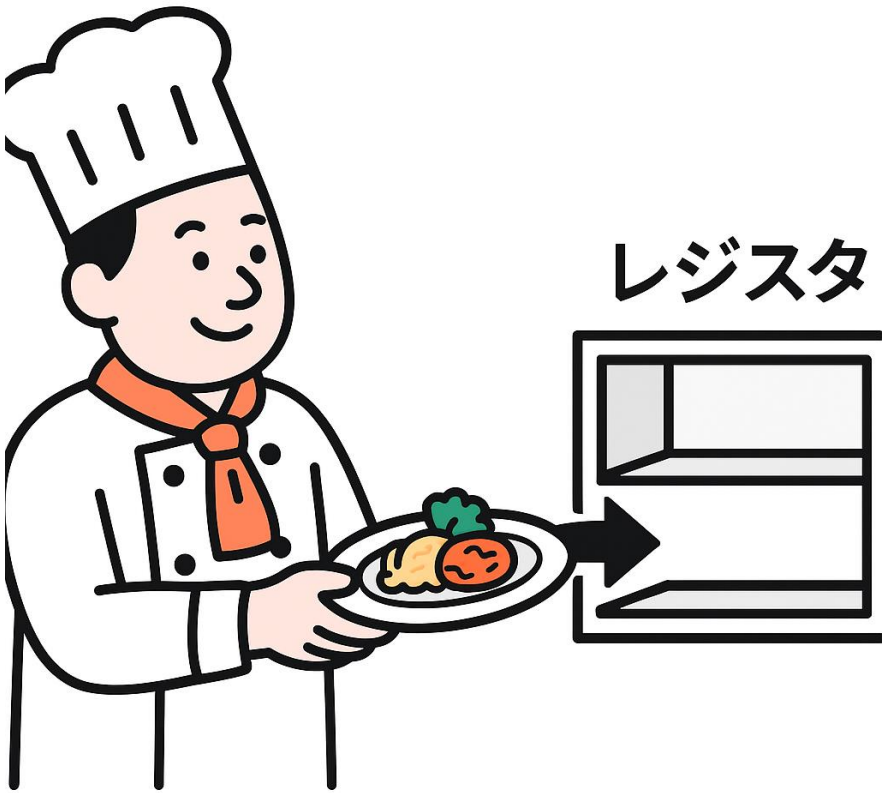
読み取れた命令：ADD R1 R2

- 3つの部品に分解される：
  - ADD（足せ）＝オペコード
  - R1（食材トレー1）＝宛先/出力
  - R2（食材トレー2）＝入力

命令レジスタの内容を解読し、それがどんな種類の命令で、
どのデータを使うのかを判断する。

❸命令実行：Execute (EX)

❹結果格納：Write Back (WB)



よく使われる命令の例

命令	ニモニック
即値ロード	LDI IMM
レジスタ間転送	MOV GRa, GRb
加算	ADD GRa, GRb
減算	SUB GRa, GRb
ロード	LD GRa, GRb
ストア	ST GRa, GRb
無条件ジャンプ	JMP ADRS

メモリへアクセス

CPUの仕事サイクル（命令サイクル）

❶命令フェッチ：Instruction Fetch (IF)

❷命令解読：Instruction Decode (ID)

❸命令実行：Execute (EX)

❹メモリアクセス：Memory Access (MEM)

❺結果格納：Write Back (WB)

しかし、命令の正体とは？

ADD R1, R2  **0001110100100001**

- 私たちは「ADD」と書いていますが、CPUが見ているのはただの「0」と「1」の羅列
- CPUはどうやって0001が「足し算」だと知るのでしょうか？
- そこには、ハードウェアとソフトウェアの間の「約束事」が必要

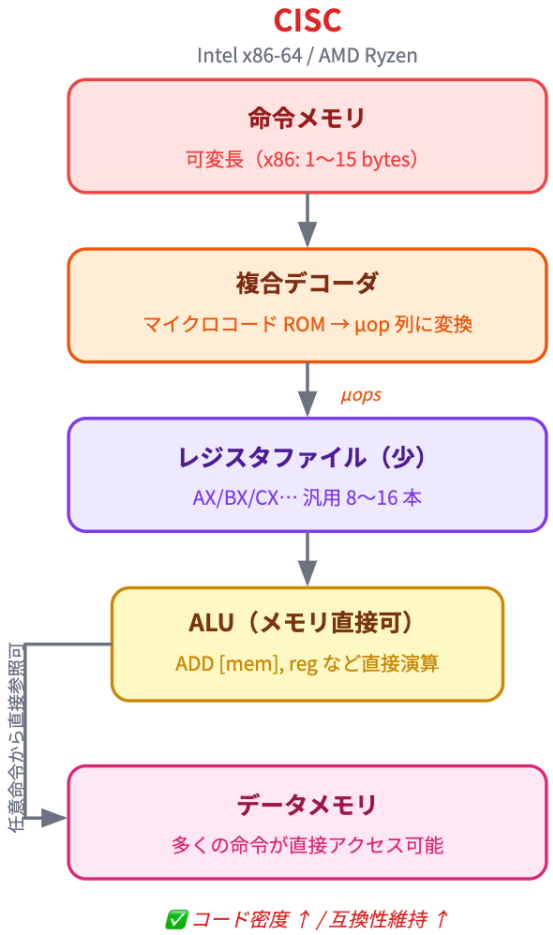
命令セットアーキテクチャ (ISA)

- **ISA：CPUが理解できる命令（言葉）の体系**
- 例：「ADD（足せ）」「LOAD（読め）」「STORE（書け）」
- この「辞書」の設計思想によって、CPUは大きく2種類に分かれる

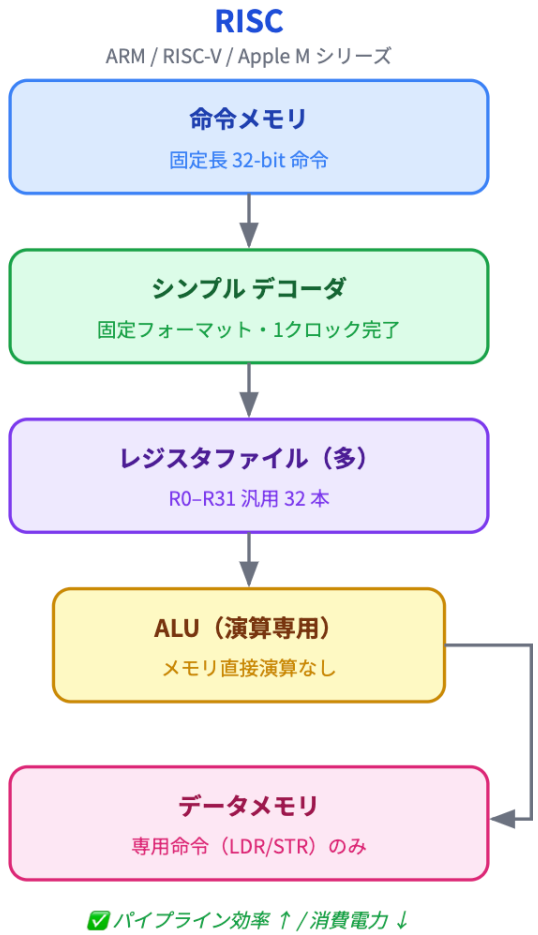
CSIC vs RISC

CISC: Complex Instruction Set Computer (複雑命令セットコンピュータ)

誕生: 1970年代から主流
思想: ハードウェア側で複雑な命令を用意し、ソフトウェア開発を楽にする
特徴: 命令の長さや実行時間が可変で、回路が複雑になりがち



RISC: Reduced Instruction Set Computer (縮小命令セットコンピュータ)



誕生: 1980年代に登場
思想: 単純な命令に絞り、ハードウェアをシンプルにして高速化する
特徴: 命令長が固定で、パイプライン処理などで高速化しやすい、省電力

現在の主流アーキテクチャ

- **CISC (主にx86アーキテクチャ):**
  - **代表例:** Intel (Core iシリーズ), AMD (Ryzenシリーズ)
  - **主な用途:** デスクトップPC、ノートPC、サーバー
  - **強み：**ソフト資産が巨大（Windows/多くの業務ソフト）、高性能コア

- **RISC (主にARMアーキテクチャ):**
  - **代表例:** Apple (Mシリーズ), Qualcomm (Snapdragon), NVIDIA
  - **主な用途:** スマートフォン、タブレット、組み込みシステム、最近ではPCやサーバーにも拡大
  - **強み：**省電力設計、ライセンス方式で多社がSoCを作れる（Apple/Qualcomm等

RISC-V（RISC系ISA、オープン）：誰でも使える命令セット

- 強み：オープン標準、拡張が柔軟（用途別に命令を足せる）
- 現状：組み込み・研究・教育で増加、PC級は発展途上

RISC-V®

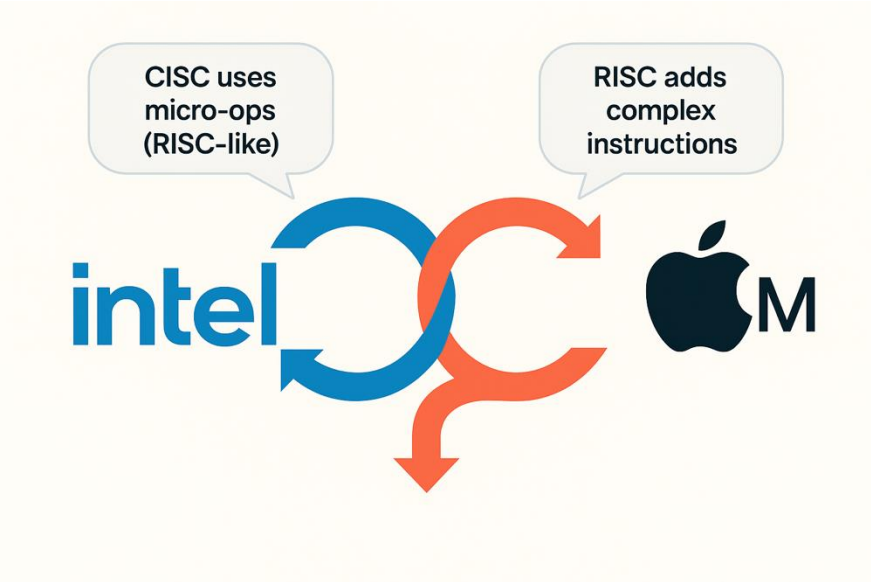
どちらが優れている？

境界は曖昧に：

- 現代のCISC (x86)プロセッサは、内部で複雑な命令を単純なRISC風の命令（マイクロオペレーション）に**変換**してから実行している。
- 一方、RISCプロセッサも、よく使う処理を高速化するための専用命令を追加している。

考えるべき問い：

- なぜ今、AppleはMacのCPUをIntel(CISC)から自社設計のARM(RISC)に切り替えたのか？
- これは、現代において「電力効率」と「性能」のバランスが、過去の「互換性」よりも重要になってきたことを示唆しているのかもしれない。

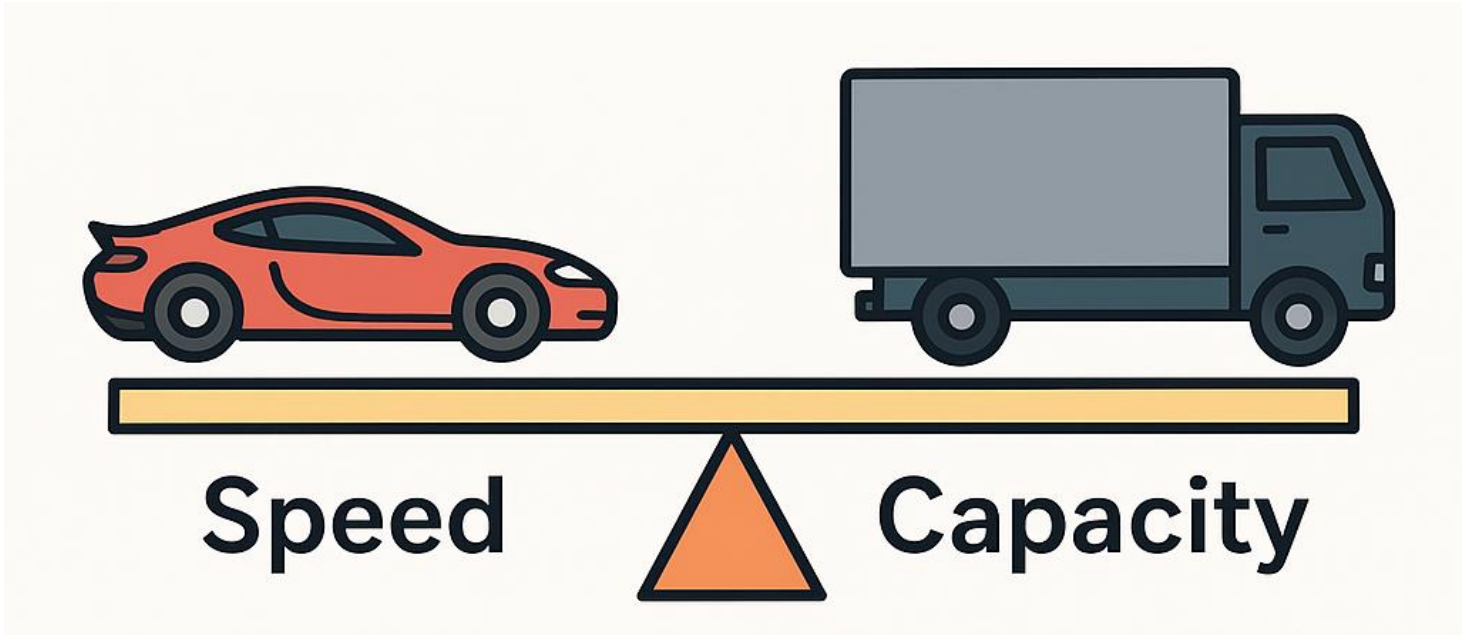


結論： もはや単純な二項対立ではなく、「どんな問題を解決したいか」という目的に合わせた設計思想の選択が重要。

CONTENTS

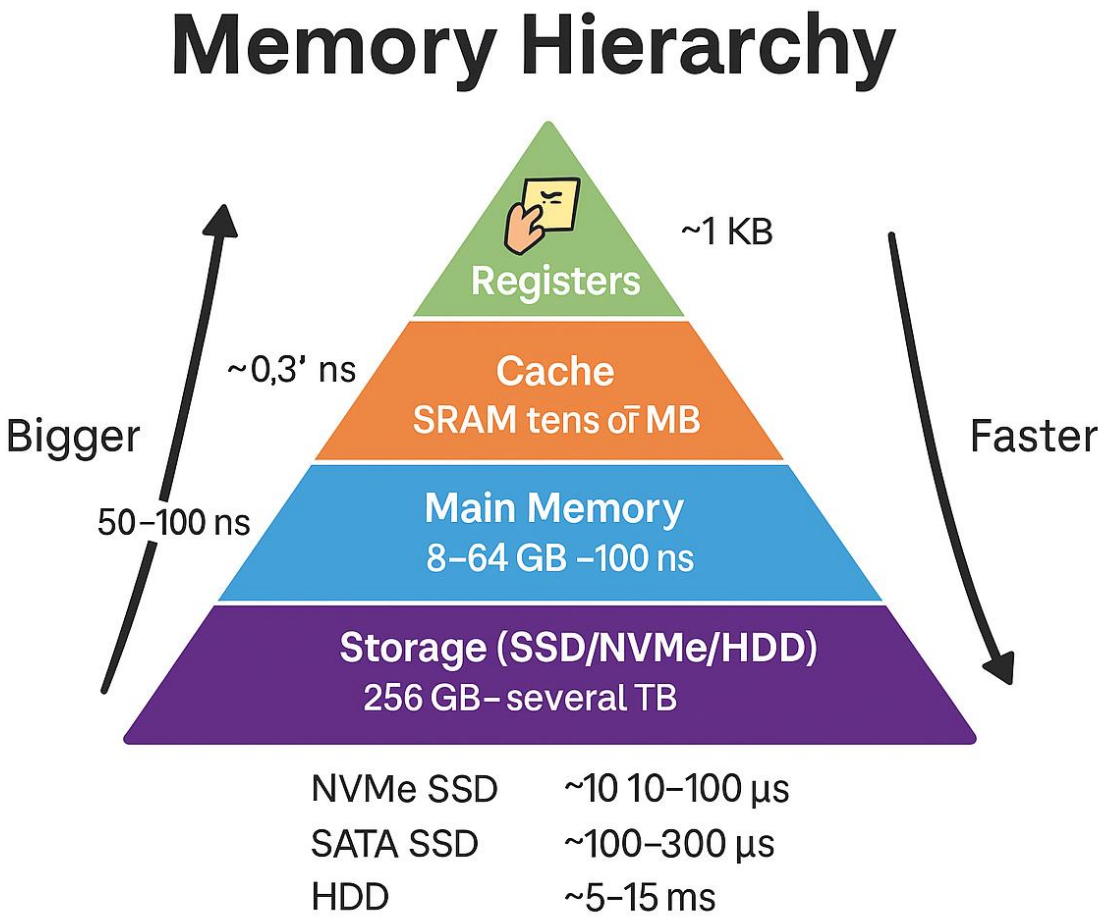
モジュール５：コンピュータアーキテクチャ

なぜメモリは複数種類あるのか？



「速さ」と「容量（安さ）」はトレードオフの関係にある。

記憶の階層構造

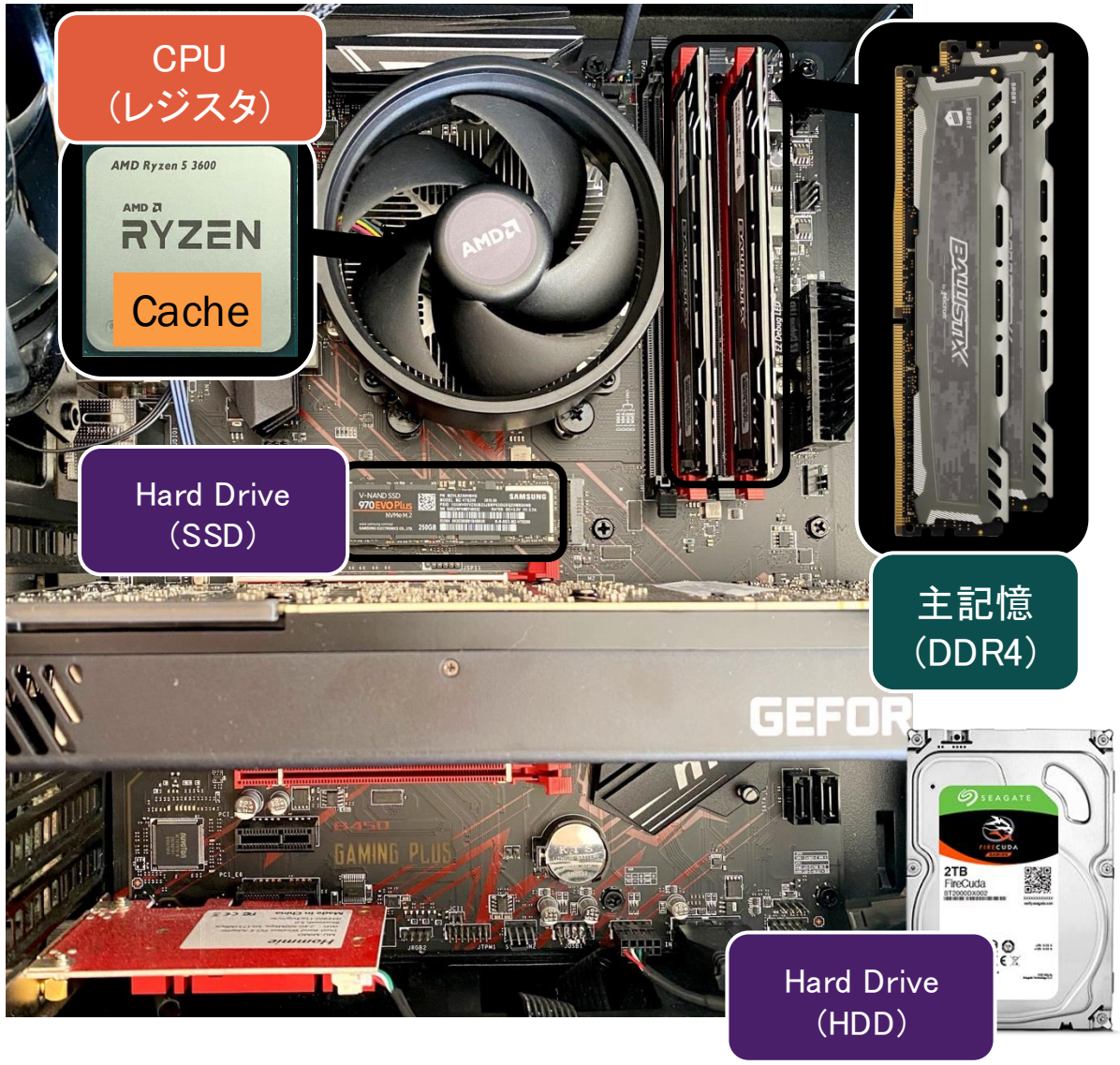


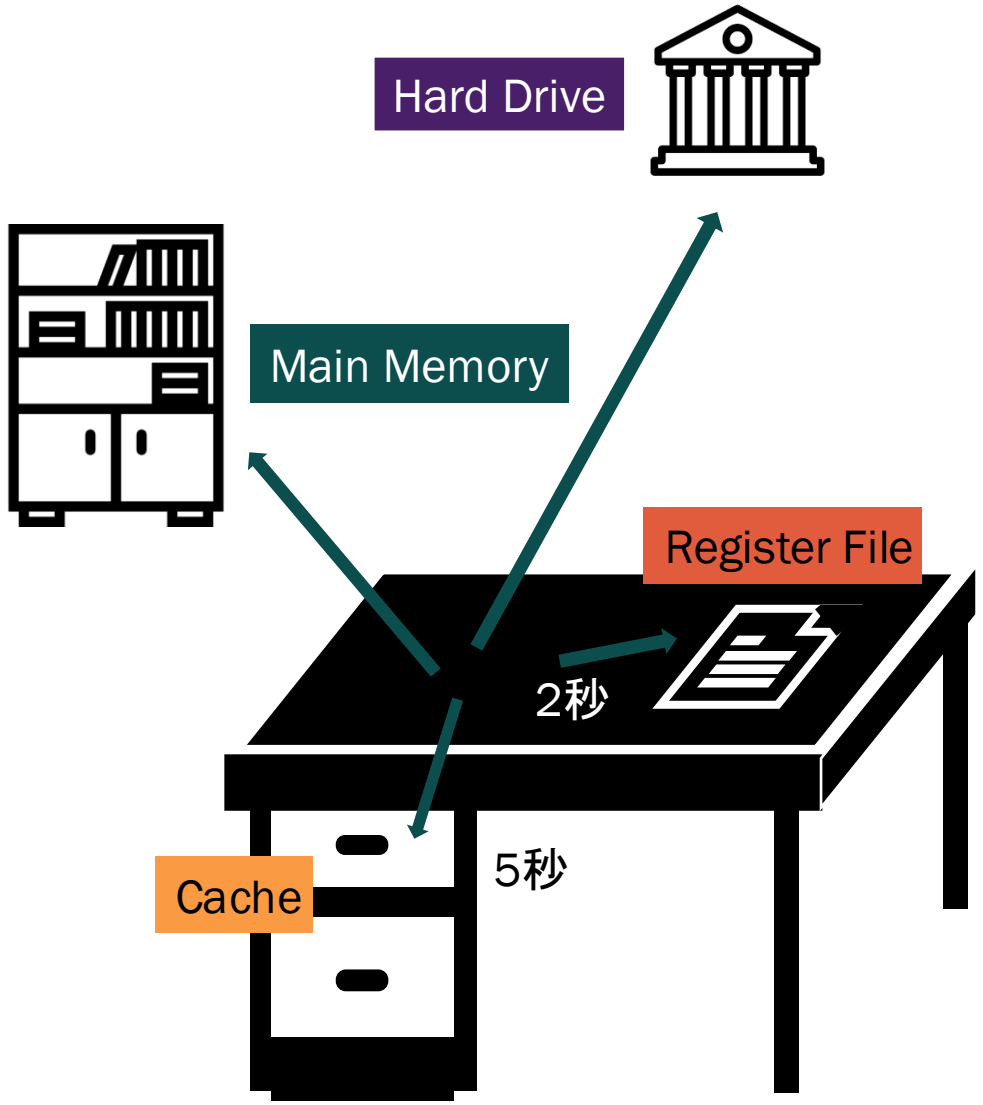
レジスタ　CPU内部、最速・容量極小

キャッシュメモリ　CPU内部（SRAM）、レジスタより遅いがDRAMより速い

メインメモリ　CPU外、揮発性

ストレージ　非揮発性、容量最大・最も遅い


CPU
（レジスタ）
AMD Ryzen 5 3600
AMD
RYZEN
Cache
Hard Drive
（SSD）
主記憶
（DDR4）
Hard Drive
（HDD）


Hard Drive
Main Memory
Register File
2秒
5秒
Cache

局所性の法則 (Principle of Locality)

経験からの結論：

コンピュータプログラムは時間と空間の局所性を示す．

空間的局所性
教科書の50ページを見たら、次は51ページを見る可能性が高い
（ アクセスされた項目に近い項目が間も無くアクセスされる確率が高い ）

時間的局所性
一度使った教科書のページは、すぐにもう一度見る可能性が高い
（ アクセスされた項目を二度アクセスする確率が高い ）

プログラムの動作には、「一度使ったものは、すぐまた使う」「近くにあるものは、すぐ使う」という性質がある。

局所性の法則 (Principle of Locality)

典型的なプログラムセグメント

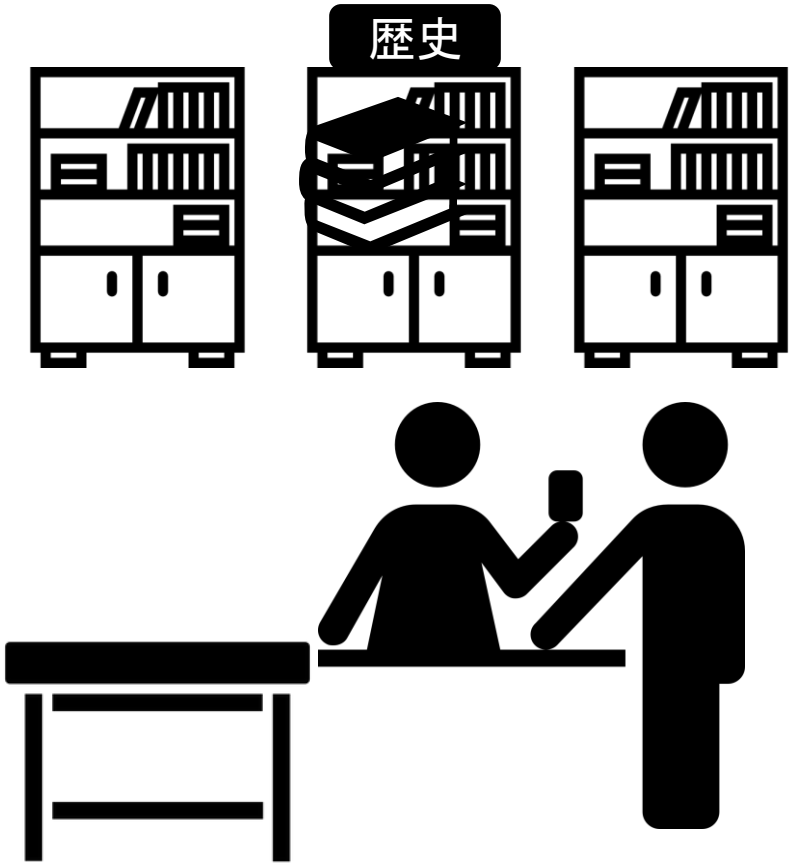
```
for (i=0; i<1000; i++)
{
    for (j=0; j<200; j++)
        {
            sum += a[i][j];
        }
}
```

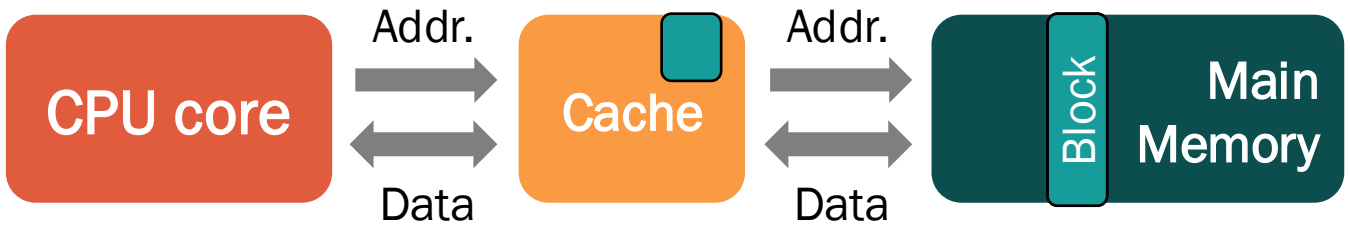
- **時間的局所性** (Temporal Locality):
  - アクセスされた項目（変数sum）を二度アクセスされる確率が高い.
- **空間的局所性** (Spatial Locality):
  - アクセスされた項目(a[0][0])に近い項目(a[0][1], a[0][2], …)が間も無くアクセスされる確率が高い.

Cacheは局所性の法則を活用し,
CPU性能を向上させる

キャッシュ (Cache) とは

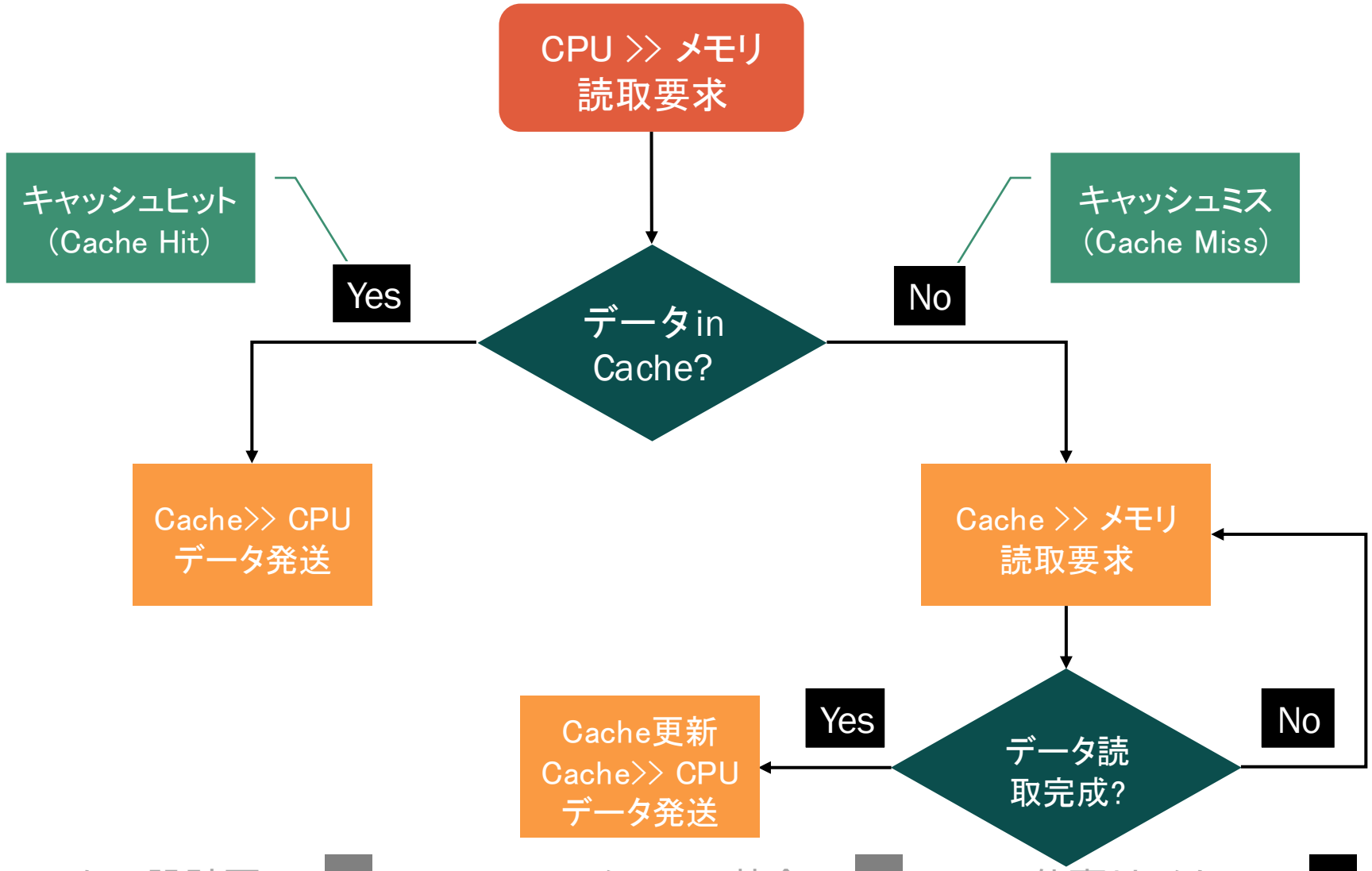
- キャッシュによる時間的局所性の活用
  - 参照するデータをメモリから取得する際，隣接するユニット内のデータを一緒に読み出す．
  - ブロックを単位に，メモリとのデータ交換
- キャッシュによる空間的局所性の活用
  - 最近頻繁にアクセスされるデータブロックを保存する．



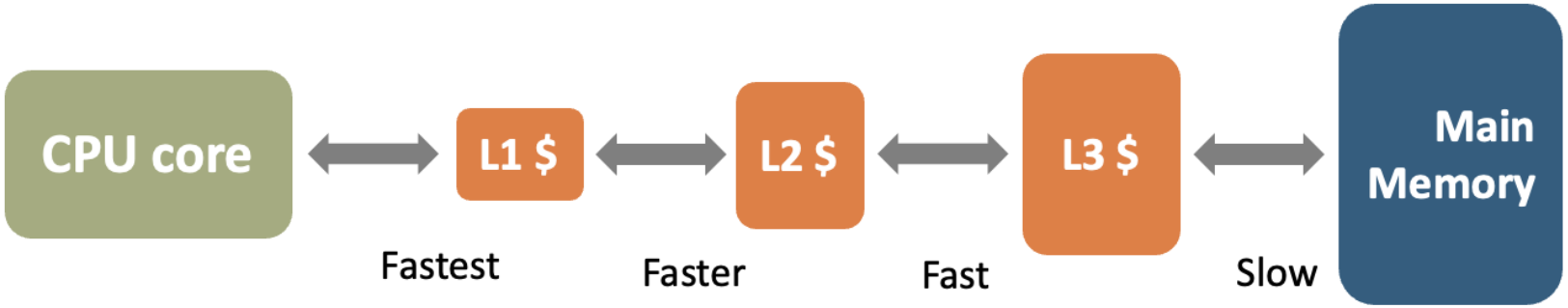


＊キャッシュ操作はハードウェア上で行うため，プログラム上には明白ではない

キャッシュへのアクセス流れ



マルチレベルキャッシュ

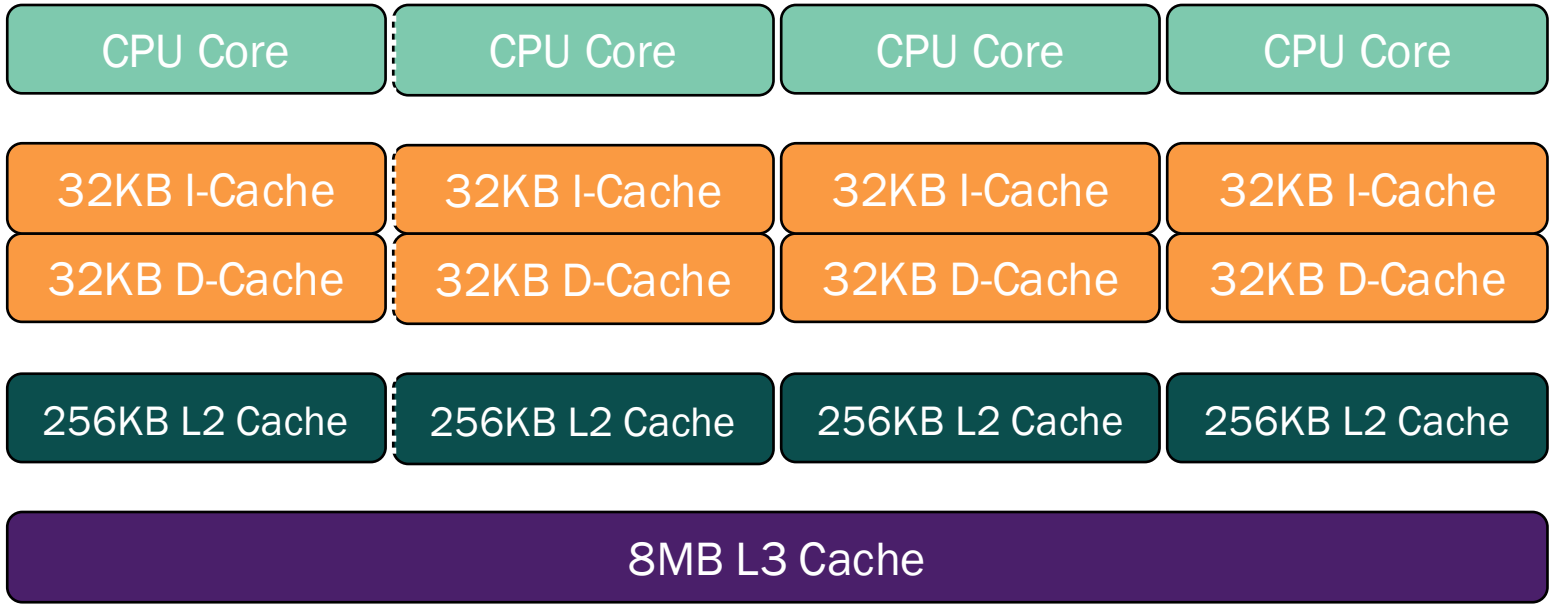


キャッシュ in Core i7

一次キャッシュ (L1 Cache)
- 分散キャッシュ
- 8-way associate
- Hit time: 4 clock cycles

二次キャッシュ (L2 Cache)
- 統合キャッシュ
- 8-way associate
- Hit time: 11 clock cycles

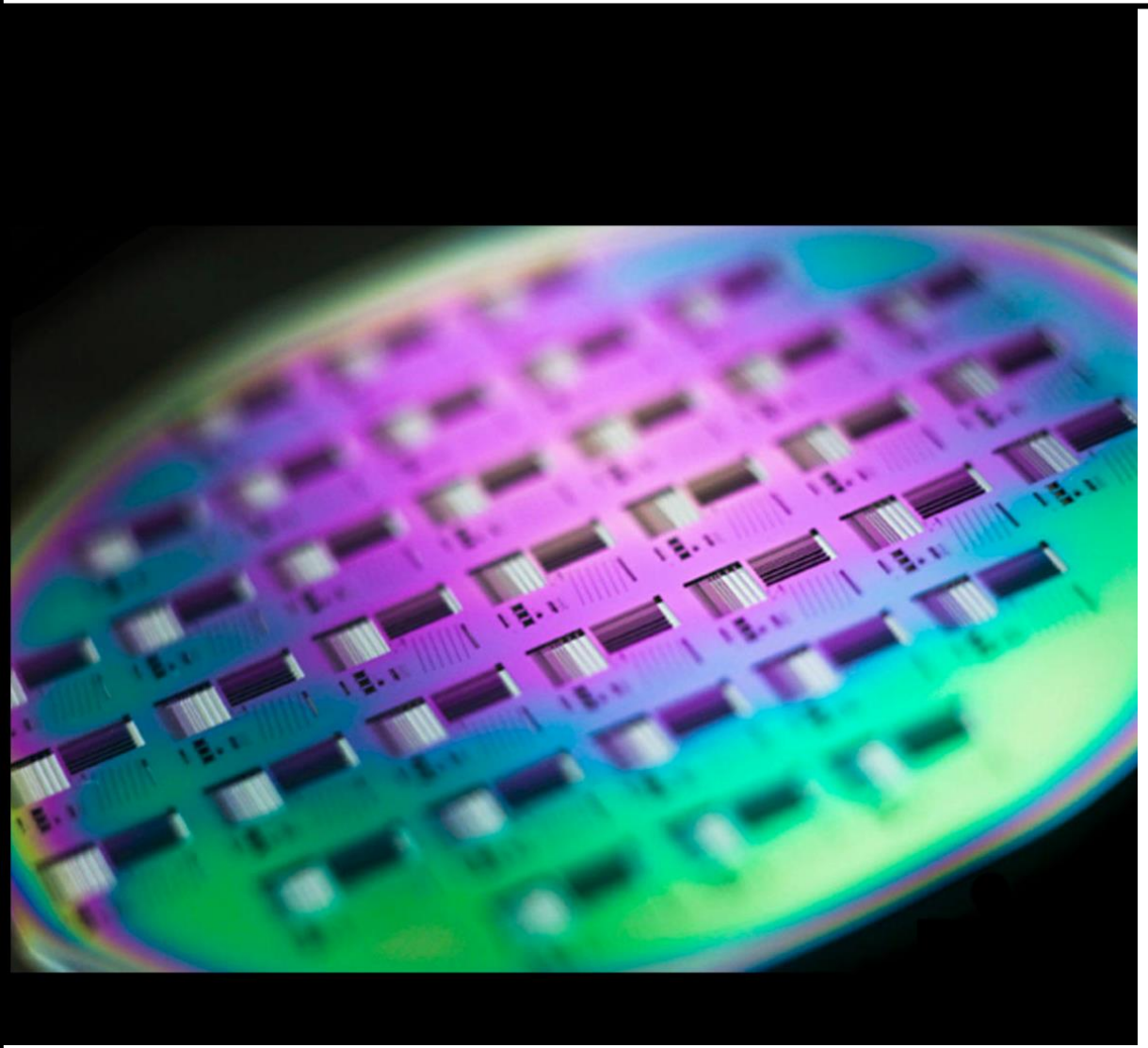
三次キャッシュ (L3 Cache)
- 統合キャッシュ
- 16-way associate
- マルチコア共有
- Hit time: 30~40 clock cycles



まとめ

- **フォン・ノイマン型アーキテクチャ** - プログラムとデータを同じメモリに置く革命。
- **CPUの仕事サイクル** - 「フェッチ→デコード→実行→格納」の4ステップ。
- **ISA (CISC vs RISC)** - CPUの「言語」の設計思想。
- **記憶の階層構造** - 速度とコストを両立させるための工夫。

次回予告



次回：性能評価とチップ設計

今日学んだ設計図の「良し悪し」はどうやって測るのか？そして、この複雑な設計図を、人間はどうやって記述するのか？