

KYUSHU UNIVERSITY · SYS-104

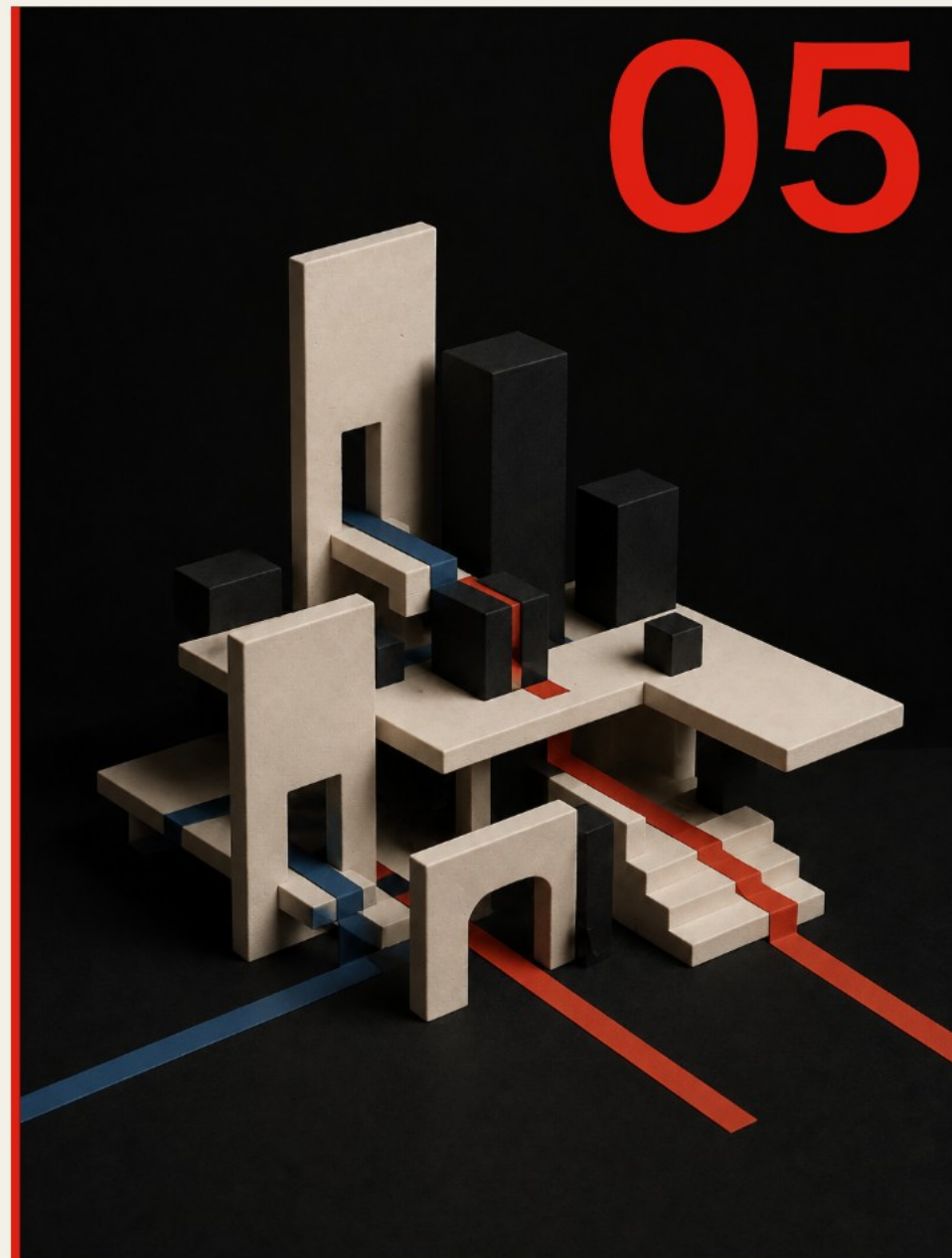
コンピュータ アーキテクチャ

命令が動く仕組み

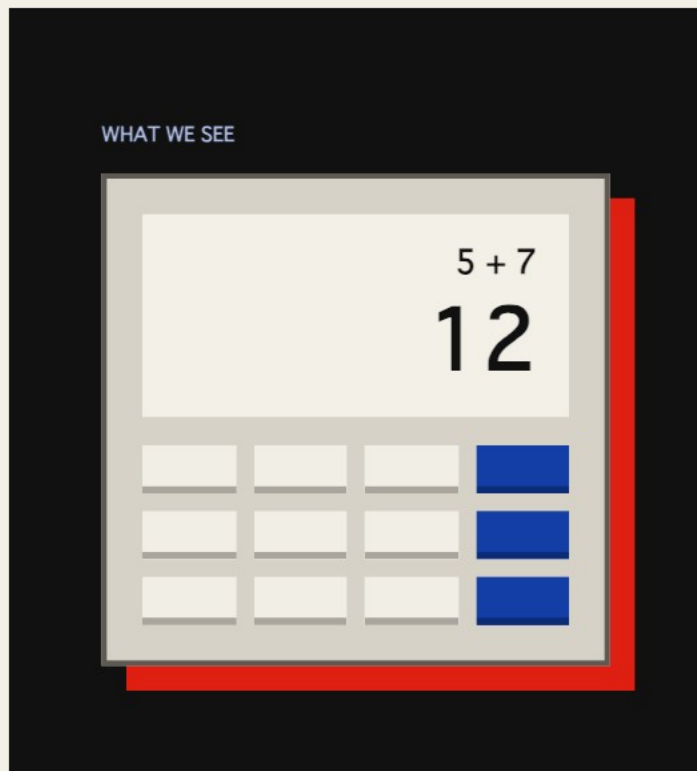
2026 年 9 月

Olivia Chen Toyama · C2LAB

LECTURE 5



記憶した命令を、どう使う？



WHAT THE MACHINE MUST DO

命令とデータは、メモリに置かれている

命令を読む → 値を使う → 結果を残す

命令は、部品をどう動かし
5 + 7 を 12 にするのか

注文が、作業の内容を決める



注文を読む・指示する

何を作るかに合わせて、作業を選ぶ

材料・道具を使って作業する

指示に従って、材料を加工する

CPU でも、命令の内容に応じて、使う回路と値の流れを決める。

今回の流れ

5 + 7 を計算するには、何が必要か。

まず、処理・記憶・入出力の分担をつかむ。

01 / SYSTEM

CPU・メモリ・I/O

02 / PROGRAM

プログラム内蔵

03 / JOURNEY

命令の実行

04 / CONTRACT

命令の共通ルール

05 / CACHE

メモリ階層

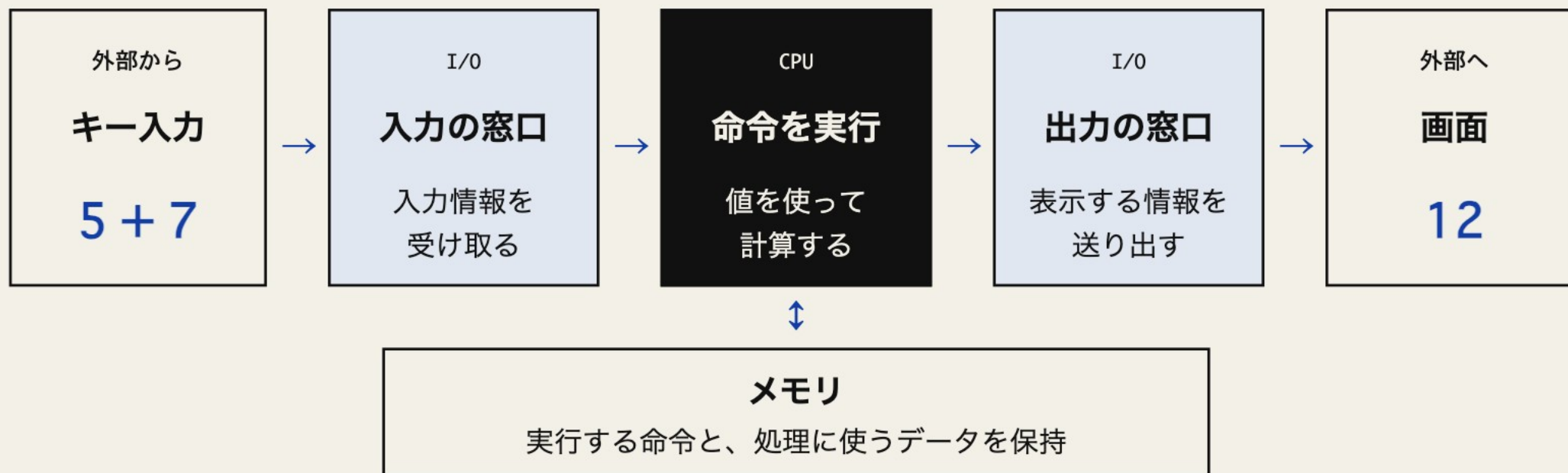
CPU・メモリ・I/O



処理する・保持する・外部とつなぐ 三つの役割を情報の流れで結ぶ

I/O：入力から、結果の表示へ

入力した情報を受け取り、計算した結果を外へ渡す。

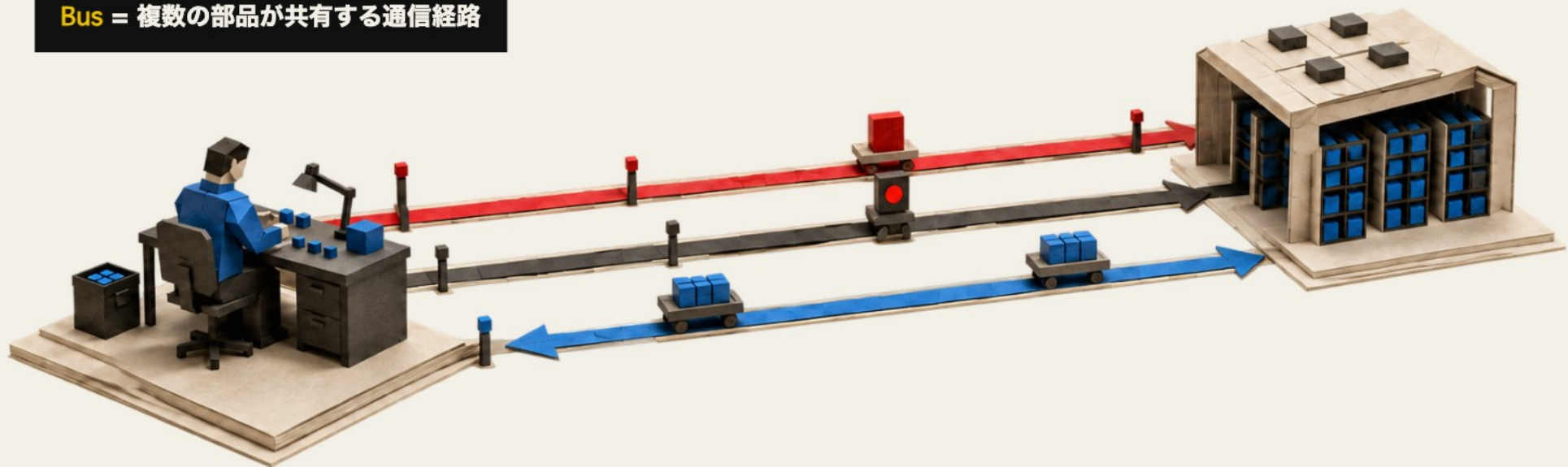


I/O は、CPU の計算と外部の機器をつなぐ窓口。

計算機アプリ全体の流れを単純化した図。一つの ADD 命令が、入力から表示まですべてを行うわけではない。

バス：情報を運ぶ共用道路

Bus = 複数の部品が共有する通信経路



ADDRESS
行き先

CONTROL
READ / WRITE

DATA
運ぶ内容

バスが運ぶ三つの情報

バス 部品間で情報をやり取りする通信路



三つの役割 場所・動作・内容を区別する。一本ずつの配線を表す図ではない

要求を送り、値を受け取る

0x1000 にある値を読む

1 読み出しを要求

2 データが戻る



要求 場所と動作を伝えた。値は、まだ CPU に届いていない

今回の流れ

部品をつなぐ経路はできた。

今度は、仕事の手順をどこに置くかを考える。

01 / SYSTEM

CPU・メモリ・I/O

02 / PROGRAM

プログラム内蔵

03 / JOURNEY

命令の実行

04 / CONTRACT

命令の共通ルール

05 / CACHE

メモリ階層

ENIAC のプログラミング



初期 ENIAC のプログラミング

手順を与える方法

**ケーブルをつなぎ、
スイッチを設定する。**

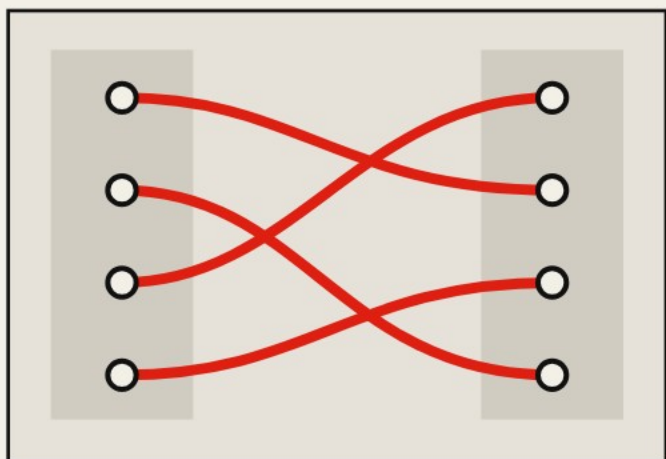
どの部品を、どの順番で動かすか。
その手順を、人が接続で指定していた。

計算の前に、機械の設定が必要だった。

計算は速い、変更は遅い

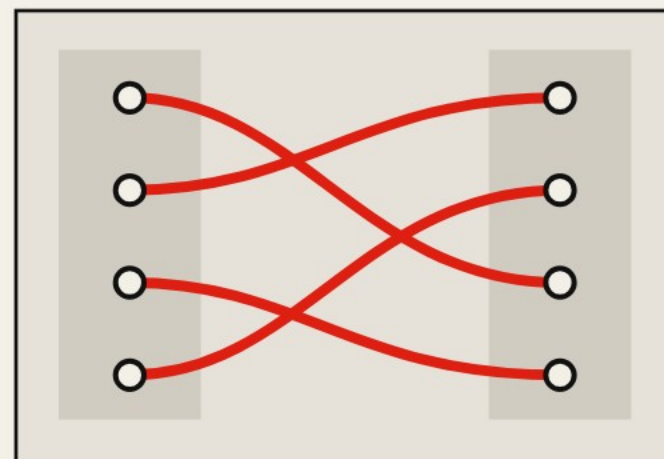
別の仕事をさせるには、接続を設定し直す。

仕事 A の接続



人がつなぎ直す

仕事 B の接続

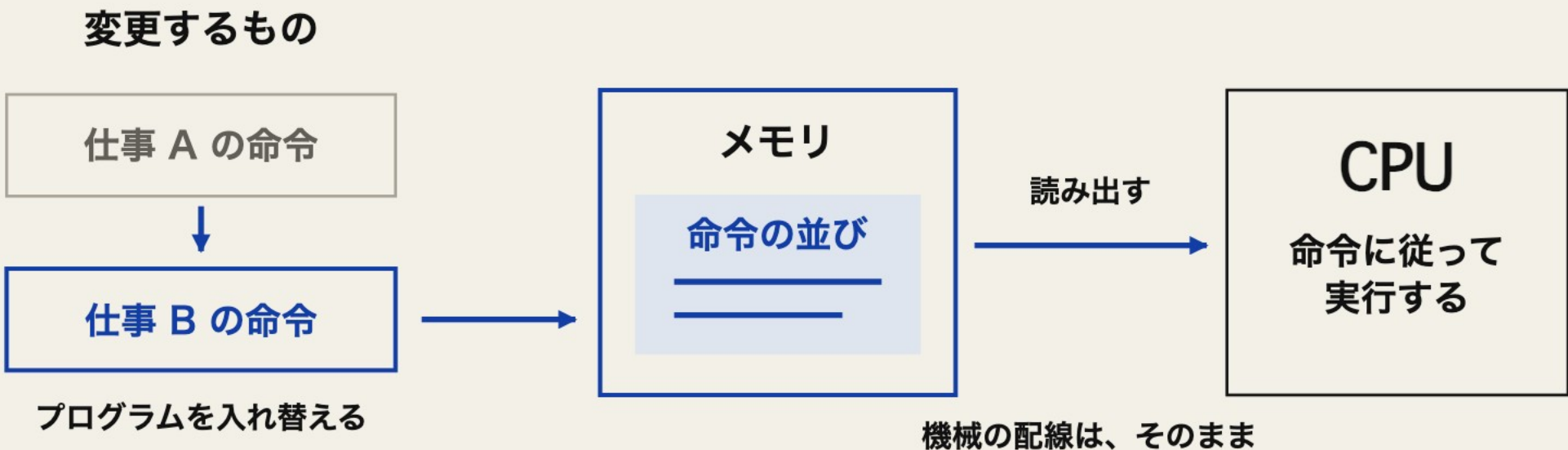


計算は機械が行う。手順の変更には、人の作業が要る。

接続変更を表す模式図。実機の配線図ではない。

命令そのものを保存する

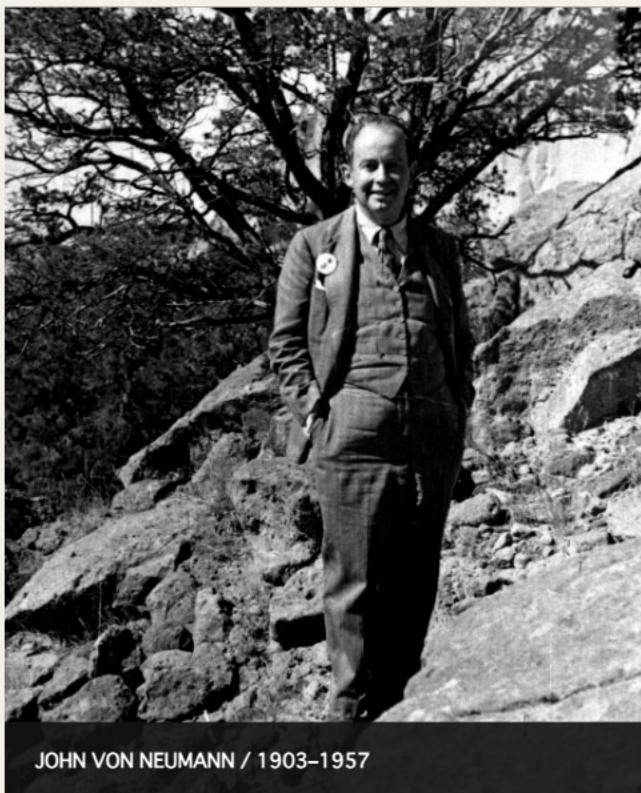
命令も、データのようにメモリへ保存できないか？



手順を変える = 配線の変更から、命令の入れ替えへ。

プログラム内蔵方式：命令をメモリに保存し、読み出して実行する。

フォン・ノイマンとプログラム内蔵方式



MATHEMATICIAN + COMPUTER PIONEER

命令を、データと同じように メモリに保存する

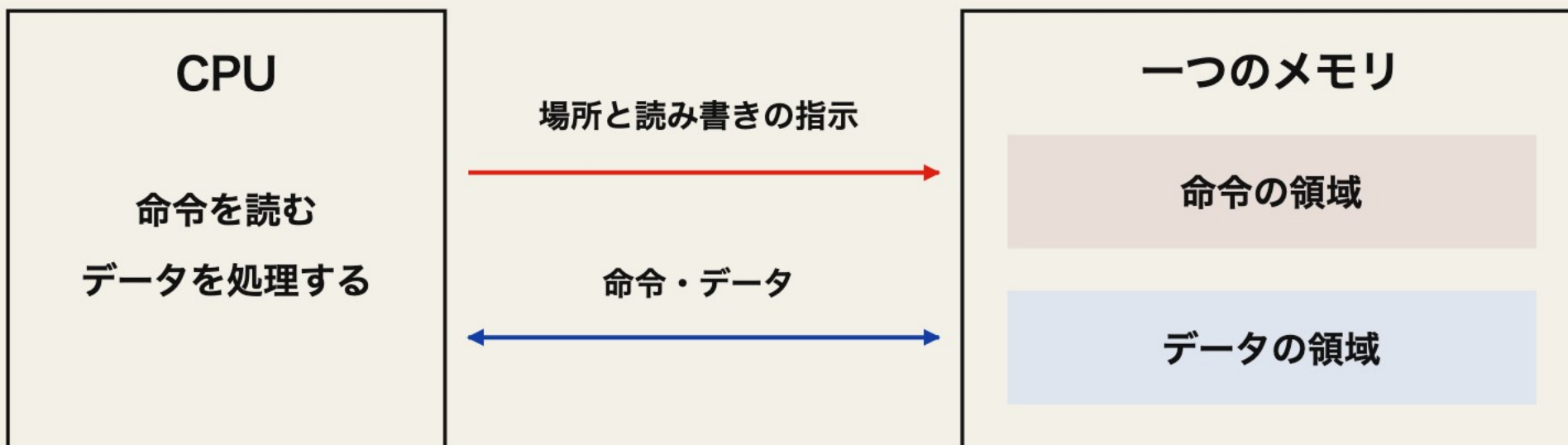
1945 年の EDVAC 報告で、この構成を体系的に記述し、広く伝えた。
Eckert、Mauchly らとの議論から生まれた構想である。

配線を組み替える



命令を書き替える

命令とデータを、一つのメモリに置く



命令もデータも、同じアドレス空間の中に置く。

ノイマン型の基本モデル：保存した命令を CPU が読み、計算して状態を更新する。

命令もデータもビット列

メモリに保存されるもの	ビット列	使うときの役割
命令	0001 0110 0101 0000	実行する操作などを表す
数値	0000 0000 0010 1010	数値 42 を表す

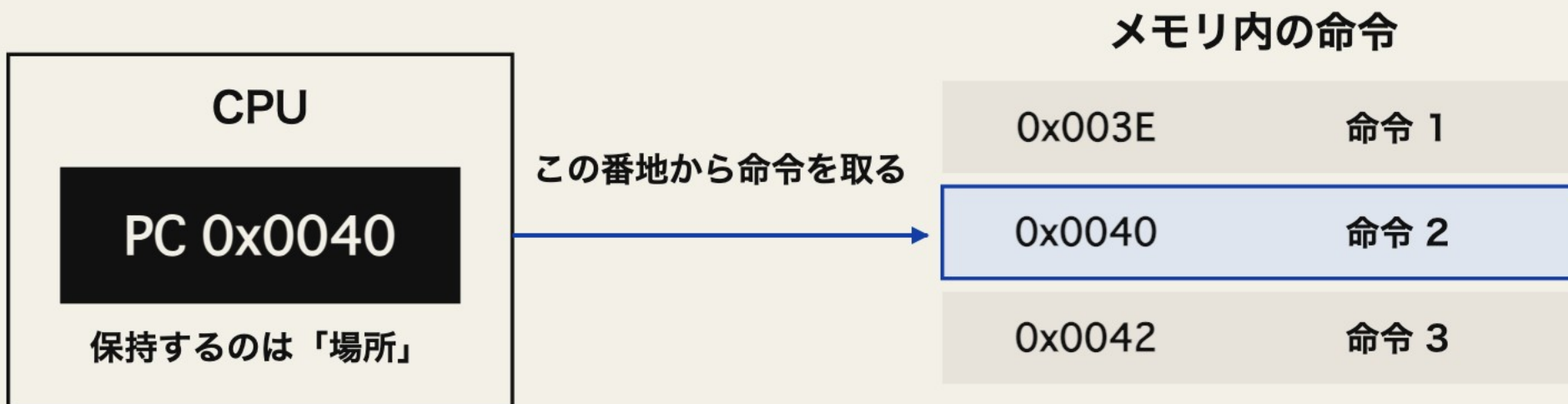
どちらも 0 と 1 で保存できる。読み出したビットを、決められた規則に従って使う。

命令をどのビット列で表すかは、この後で説明する。

PC：次に読む命令の場所

PC = Program Counter (プログラムカウンタ)

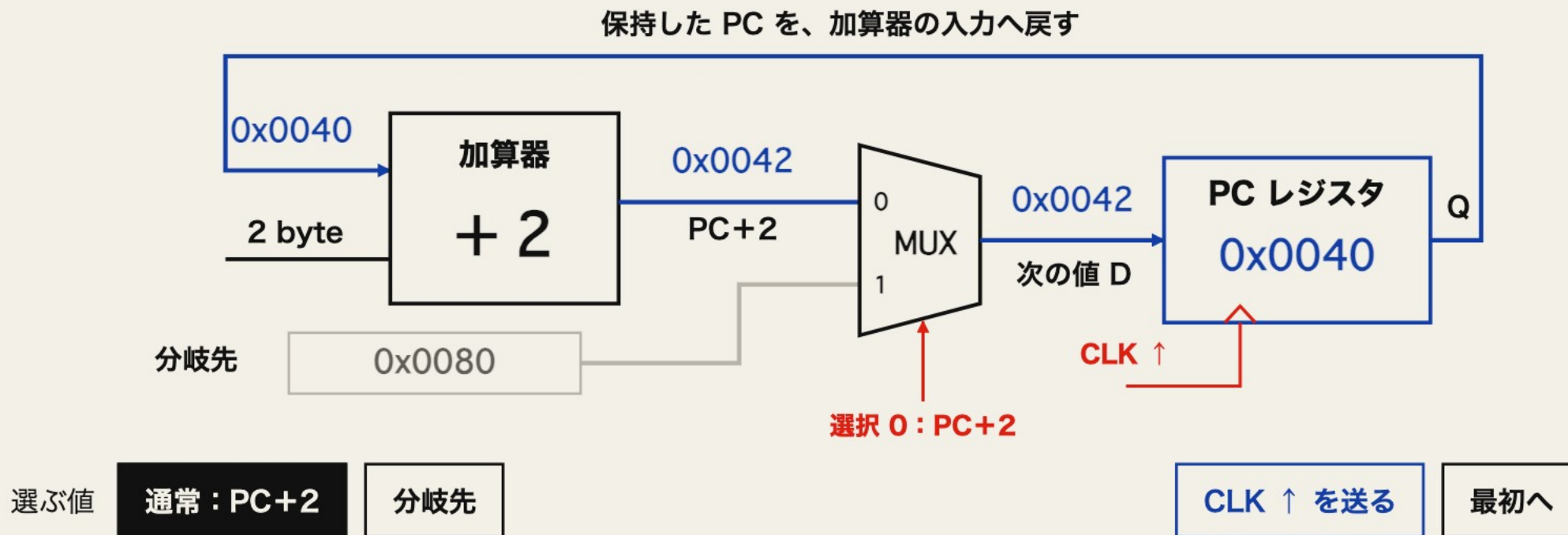
次に読み出す命令のアドレスを保持する、CPU 内のレジスタ。



前回のカウンタ（レジスタ+加算器）を応用し、命令のアドレスを進める。

PC を変えると、実行順序が変わる

前回の「レジスタ+加算器」に、次の値を選ぶ **MUX** を加える。



選択を変えても、CLK ↑ までは PC の値を保持する。

命令は 16 bit=2 byte、アドレスは byte 単位。分岐先の計算・分岐するかの判定は省略。

今回の流れ

メモリに命令を置き、PC で選べるようになった。

一つの ADD と LOAD を、CPU の中で追ってみる。

01 / SYSTEM

CPU・メモリ・I/O

02 / PROGRAM

プログラム内蔵

03 / JOURNEY

命令の実行

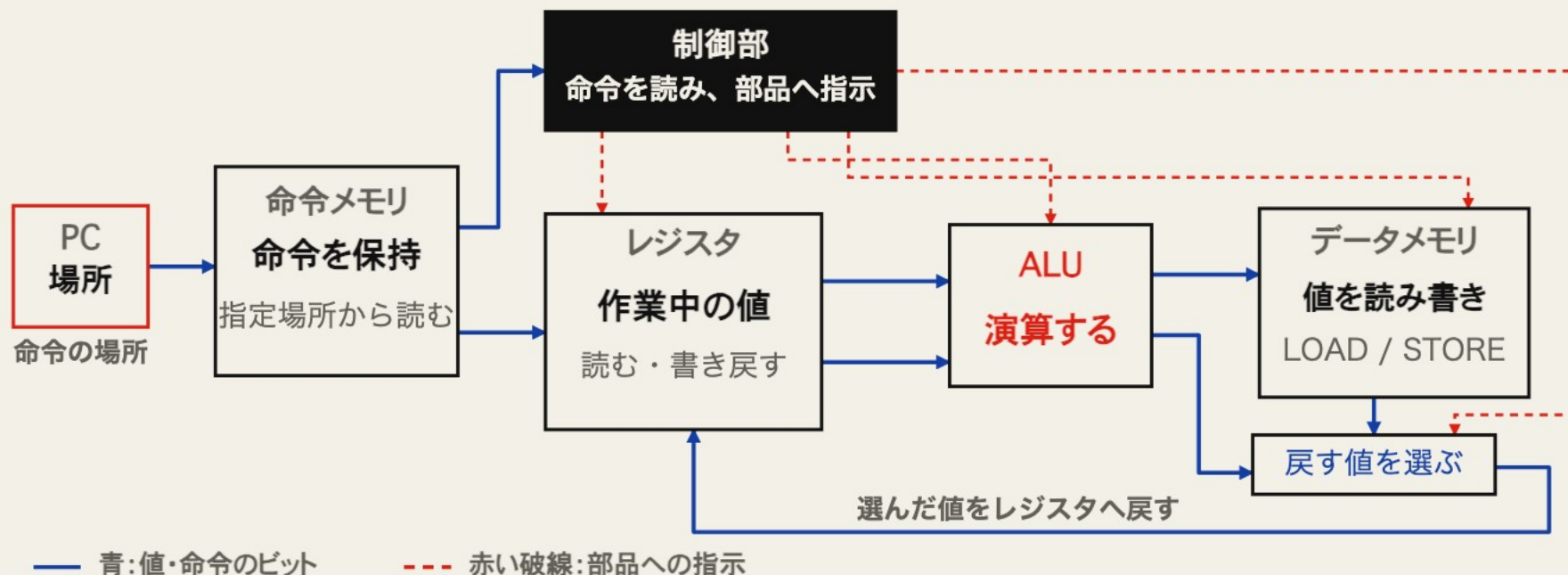
04 / CONTRACT

命令の共通ルール

05 / CACHE

メモリ階層

CPU 内部は「指示」と「値の流れ」



CONTROL

制御部：部品に動作を指示する

命令を解釈し、演算・読み書き・値の選択を制御する。

DATAPATH

データパス：値を運び、処理する

前回のレジスタ、前々回の ALU が、この経路を構成する。

この回では、制御信号を作る回路とタイミングの詳細は省略する。命令に応じて何を動かすかに注目する。

ADD R3, R1, R2 の意味

$$R3 \leftarrow R1 + R2$$

R1 と R2 の値を足し、結果を R3 に保存する。

読み出す値

R1 = 5

読み出す値

R2 = 7

書き込み先

R3 ← 12

R1・R2・R3 は、CPU 内のレジスタの名前。先頭の R3 が結果の保存先。

授業用の架空モデル レジスタ R0-R7 / 値・命令は 16 bit / メモリは byte 単位のアドレス

ADD を同じ図で最後まで追う

ADD R3, R1, R2 **R3** ← **R1** + **R2**

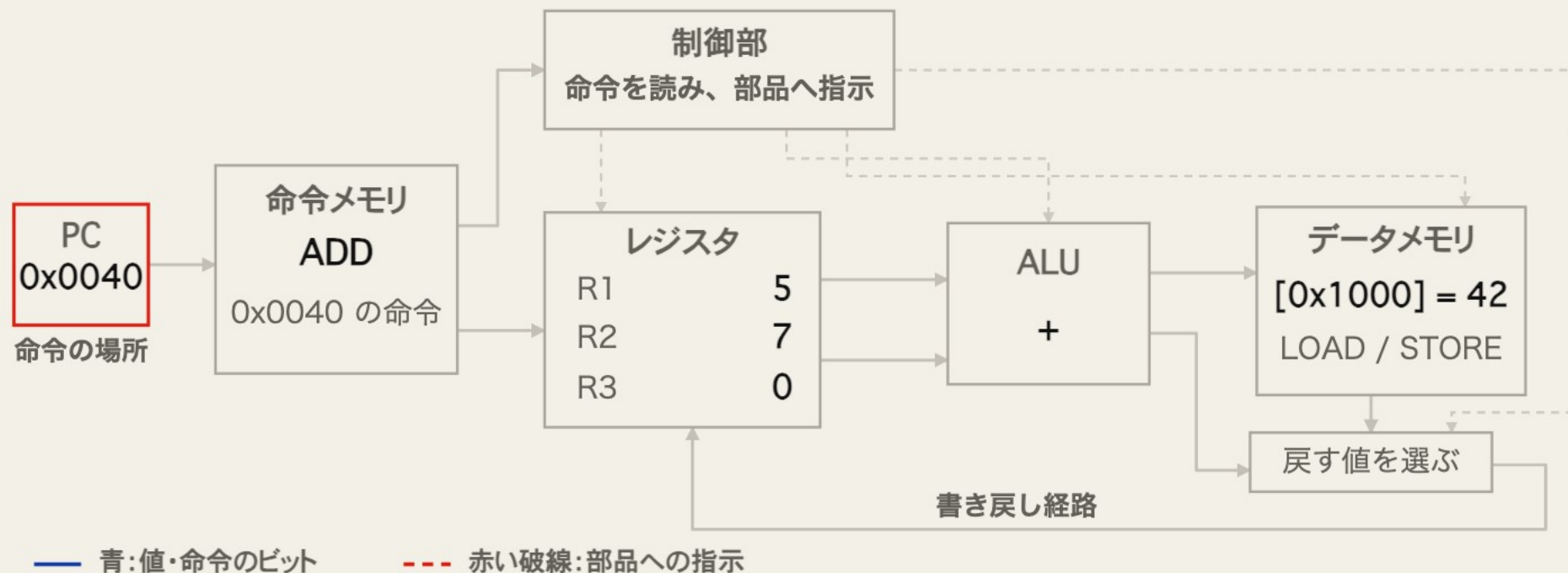
1 命令を取る

2 解釈・値を読む

3 ALU で計算

4 必要ならメモリ

5 書き戻す



準備 R1 = 5、R2 = 7、R3 = 0。命令はメモリの 0x0040 番地にある。

命令／データメモリは同じメモリの用途別表示。命令取得後、PC は 0x0042 へ。

最初へ

前へ

次へ

LOAD は、メモリから値を取り込む

LOAD R3, [R1+0] **R3 ← メモリ[R1 + 0]**

ADD **LOAD**

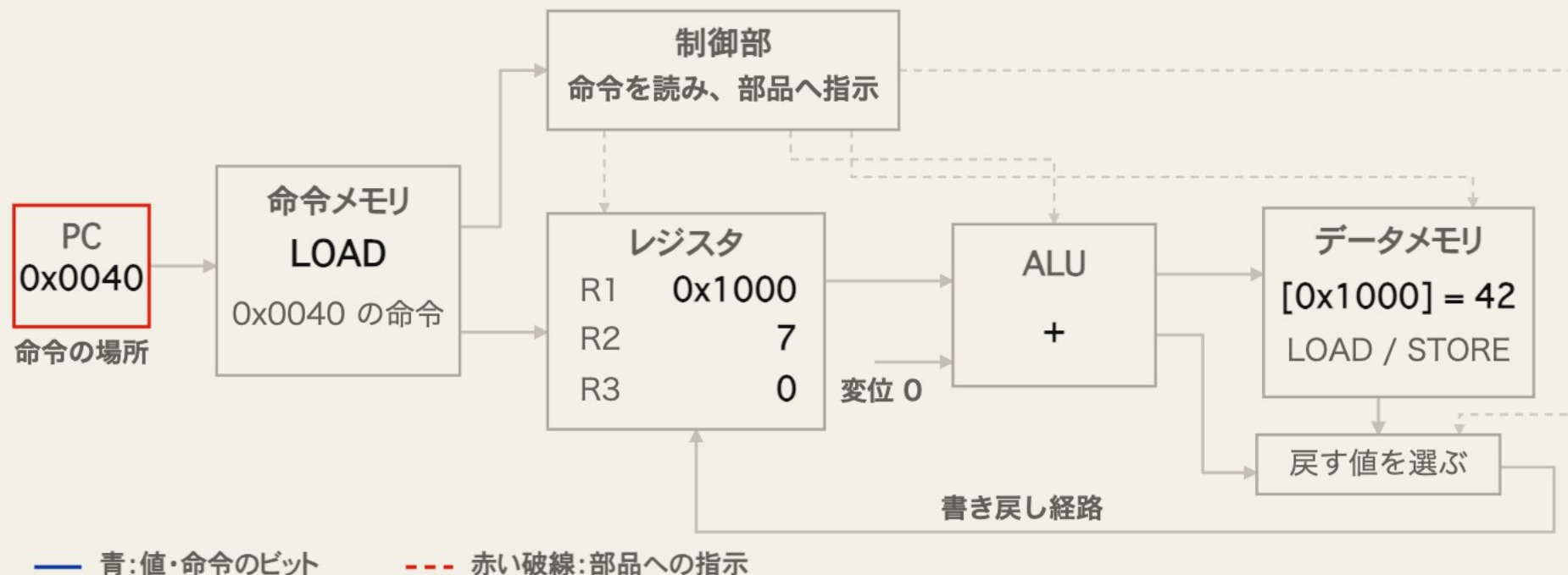
1 命令を取る

2 解釈・値を読む

3 ALU で計算

4 必要ならメモリ

5 書き戻す



準備 R1 はアドレス 0x1000。そこに保存された値を R3 に取り込む。

命令／データメモリは同じメモリの用途別表示。命令取得後、PC は 0x0042 へ。

最初へ

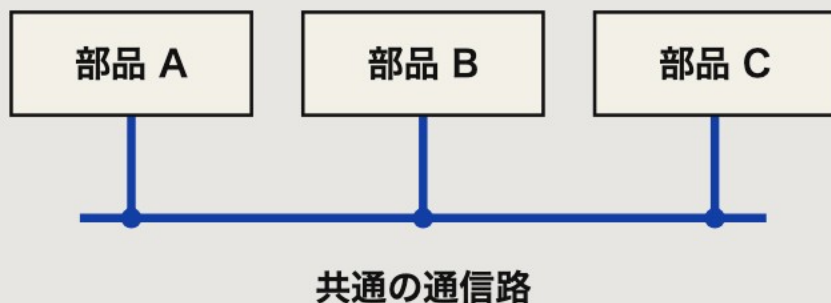
前へ

次へ

情報の役割と、接続の形を分ける

SHARED BUS

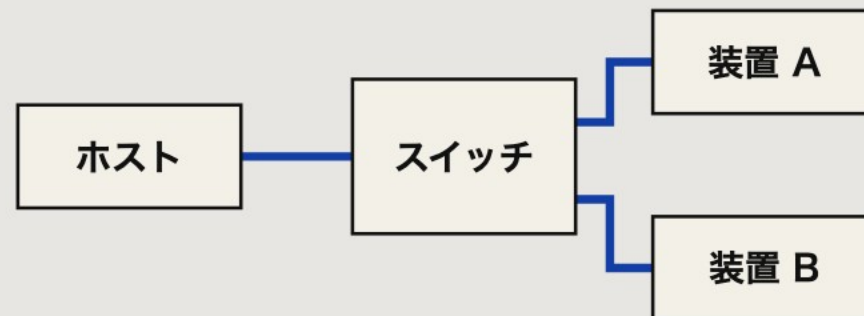
一本の経路を共有する



複数の部品が共通の線を使う
例：従来の並列 PCI バス

POINT-TO-POINT LINKS

相手ごとにリンクを結ぶ



各リンクは二つの接続先を結ぶ
例：PCI Express (PCIe)

三つの役割 ≠ 三組の専用配線

接続方式に応じて、要求やデータの運び方が変わる

命令実行を、五つの役割で整理する

いま追った仕事を、**五つの役割**として整理する。

IF / Fetch	ID / Decode	EX / Execute	MEM / Memory	WB / Writeback
命令を取る	解釈し、値を読む	ALU で計算	必要ならメモリ	結果を保存

ADD R3, R1, R2

ADD を取得	R1 と R2 を読む	$5 + 7 \rightarrow 12$	読み書きしない	$R3 \leftarrow 12$
---------	-------------	------------------------	---------	--------------------

LOAD R3, [R1+0]

LOAD を取得	R1 と変位を読む	$1000 + 0 \rightarrow 1000$	1000 番地 $\rightarrow 42$	$R3 \leftarrow 42$
----------	-----------	-----------------------------	--------------------------	--------------------

次の命令へ → 同じ流れを繰り返す

MEM の読み書きは LOAD / STORE などで行う。

五段階は説明用の役割分けであり、すべての CPU が必ず 5 クロックで実行するという意味ではない。

今回の流れ

命令から結果までの動きが見えた。

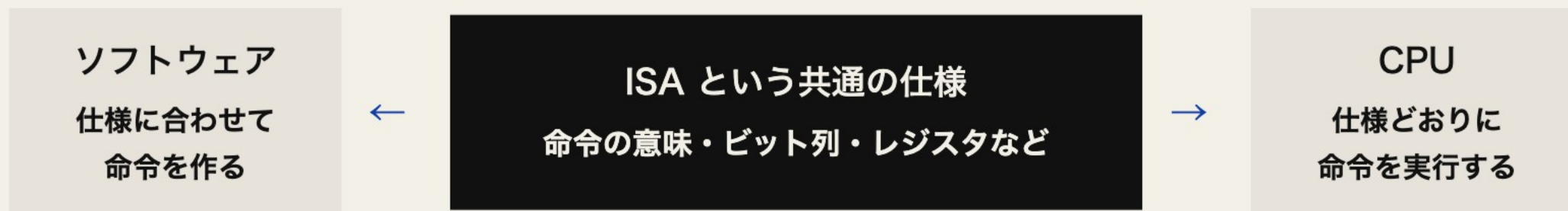
その意味とビット列を決める共通ルールを整理する。



ISA : ソフトと CPU をつなぐ約束

ISA Instruction Set Architecture
命令セットアーキテクチャ

CPU に何を命令でき、どんな結果になるかを定める仕様。



先ほどの授業用モデルでは

ADD R3, R1, R2 $\rightarrow$ $R3 \leftarrow R1 + R2$

ソフトも CPU も、「R1 と R2 を足して R3 に保存する」と解釈する。

この約束があるから、ソフトと CPU を別々に設計できる。

命令の三つのグループ

DATA TRANSFER

データを移す

LOAD / STORE

ARITHMETIC + LOGIC

計算する

ADD / SUB / AND

CONTROL FLOW

次の場所を変える

BRANCH / JUMP

アセンブリ：人が読める命令をビット列へ

ASSEMBLY / 人が読む表現 | 授業用の 16 bit ISA



ISA が決めること、実装が決めること

ISA は、命令の名前の一覧だけでなく、実行結果まで定める。

ISA が定める

ソフトウェアに見える動作

命令と符号化

何ができるか・どのビット列で表すか

レジスタとデータ

使えるレジスタ・整数などの幅や表現

メモリの指定

どのアドレスを、何 byte 読み書きするか

実行の流れ

次の PC・分岐・例外が起きたときの動作

内部の実装で決める

その動作を実現する回路の作り方

どう計算するか

ALU の数、処理の分け方、同時に進める仕事の数など。

同じ命令でも、速さは変わる

回路や動作周波数が違えば、性能や消費電力も変わる。

ADD の例

「どの値を足し、どこへ保存するか」は ISA。

「どんな加算回路で、何段階に分けて実行するか」は実装。

同じ仕事を、どんな命令に分けるか

同じ仕事：メモリの値に k を足し、同じ場所へ戻す。

仕事を分けて指定する

RISC の基本的な整数演算

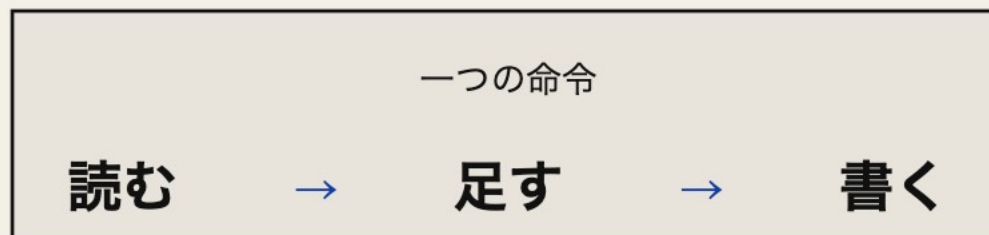


読み書きは LOAD / STORE。
演算はレジスタ同士で行う。

Reduced Instruction Set Computer

まとめた仕事も指定できる

CISC のメモリ演算の例



メモリ上の値を使う演算を、
一つの命令で指定できる場合がある。

Complex Instruction Set Computer

必要な仕事は共通。違うのは命令の区切り方。

命令数が少ないことだけでは、処理が速いとは決まらない。

同じメモリ更新を、どう命令にする？

同じ仕事 $\text{メモリの値} \leftarrow \text{メモリの値} + k$

32 bit の整数を更新する

RISC-V RV32I

3 命令

x10 : アドレス A / x6 : 加える値 k

```
lw x5, 0(x10)
```

読む → x5

```
add x5, x5, x6
```

足す → x5

```
sw x5, 0(x10)
```

書き戻す

転送と演算を、別の命令で指定する。

x86-64 Intel 記法

1 命令

rdi : アドレス A / eax : 加える値 k

```
add dword ptr [rdi], eax
```

[rdi] は、rdi が示すメモリ上の値。
dword ptr は、32 bit の読み書きを指定。

同じメモリ更新を、一つの命令で指定する。

必要な仕事は共通

メモリを読む



値を足す



メモリへ書く

この例の命令数は 3 対 1。実行時間の比ではない。共通のメモリ更新を比較し、フラグ等は省略。

RISC・CISC と、具体的な ISA の関係

RISC / CISC は設計の傾向。RISC-V・Arm・x86 は具体的な ISA の例。



RISC と RISC-V は別の名前

RISC は設計の傾向。

RISC-V は、その考え方を採る具体的な ISA。

RISC 同士でも命令は共通ではない

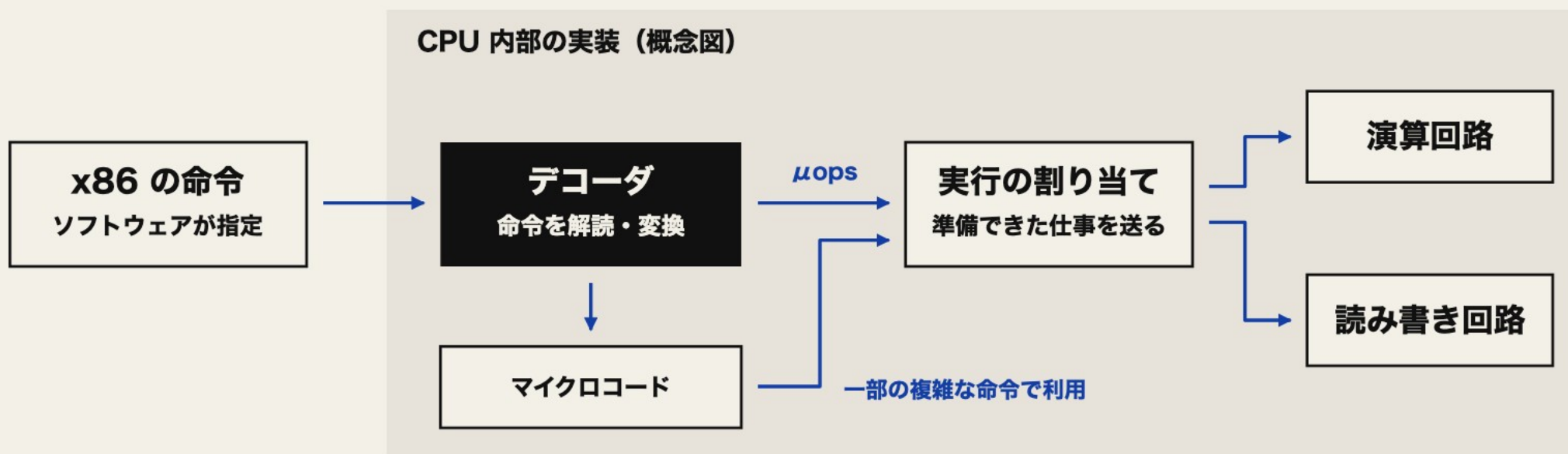
RISC-V と Arm A64 では符号化や規則が違う。

同じ機械語がそのまま動くとは限らない。

代表例で整理した図。Arm や x86 は複数の世代・命令セットを含む家族の名前でもある。

複雑な命令を、内部で小さな仕事へ

ソフトウェアに見える命令と、内部で実行する仕事は別の層にある。



μop（マイクロオペレーション）

CPU 内部で実行する小さな仕事。

一つの x86 命令が、1 個または複数になる。

命令の約束は変えない

内部の分け方は CPU によって異なる。

μop は RISC-V や Arm の命令ではない。

命令の長さにも、設計の違いがある

PC をどれだけ進めるかは、命令の長さにもよる。

RV32I / Arm A64

基本の命令は固定長



命令の区切りが一定。形式の規則性を保ちやすい。

x86

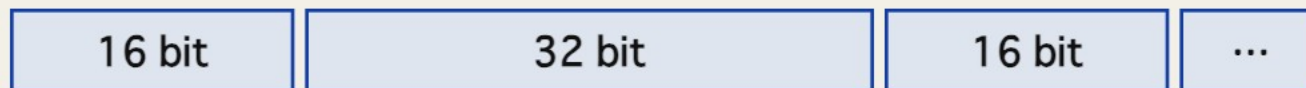
1~15 byte の可変長



命令やオペランドに応じて長さが変わる。B = byte。図は長さの例。

RISC-V + C 拡張

短い形式を追加



よく使う命令を短く表し、プログラムの大きさを減らす。

RISC でも、すべての命令が固定長とは限らない。

ここでの長さは「命令のビット列」。CPU が扱うデータの 32 / 64 bit とは別である。

同じ ISA でも、CPU の作り方は違う

同じ機械語のプログラム

同じ ISA ・ 必要な拡張命令に対応

用途の例 A / 携帯する計測器

電池と費用を重視する

必要な処理が間に合う範囲で、
小さな回路や省電力の実装を選ぶ。

用途の例 B / 大量のデータ解析

終わるまでの時間を重視する

処理を並行して進める実装も候補。
実際のデータで速さと電力を測る。

命令が約束する結果は同じ 処理にかかる時間・回路の大きさ・消費電力は変わる。

使うソフト・開発環境を確認し、性能・電力・費用を比べる。

アプリ全体の互換性には、ISA に加えて OS や呼び出し規則なども必要である。

今回の流れ

CPU が実現すべき結果は決まった。

必要な値を待つ時間を、どう減らせるだろうか。

01 / SYSTEM

CPU・メモリ・I/O

02 / PROGRAM

プログラム内蔵

03 / JOURNEY

命令の実行

04 / CONTRACT

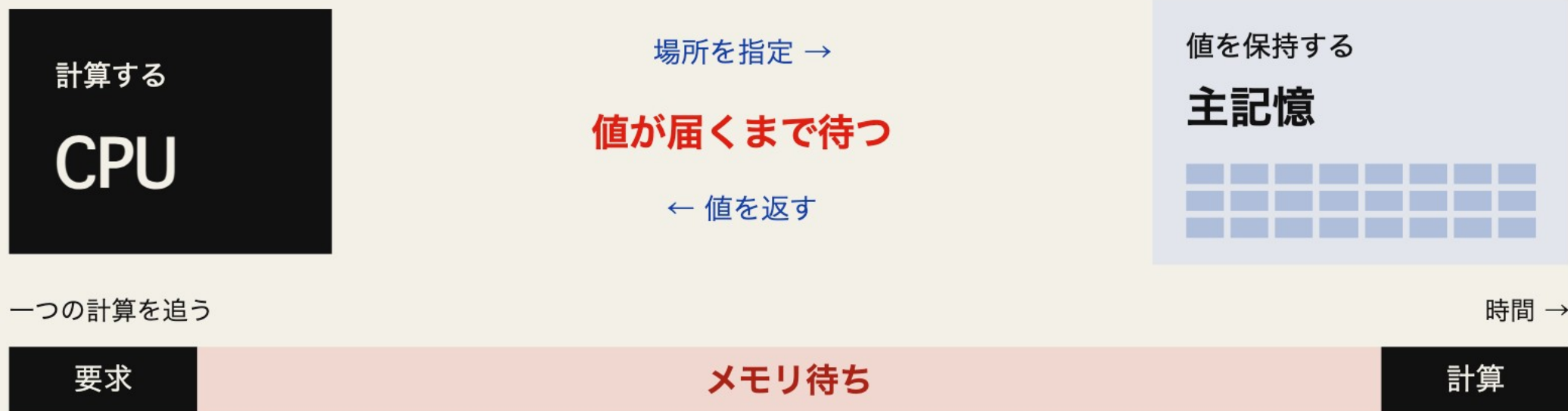
命令の共通ルール

05 / CACHE

メモリ階層

CPU が速くても、メモリを待つ

必要な値が届くまで、**その値を使う計算**は進められない。



CPU と主記憶の通信は、待ち時間と単位時間に運べる量で制約される。
ノイマン・ボトルネックとして考える問題。

次に考えること

次に使う場所に、何か傾向はないか？

すべてを速い記憶にすると、面積や費用が増える。

読み出す場所には、偏りがある

プログラムの読み出し先を追うと、**二つの偏り**が見えてくる。

同じ変数を繰り返し読む



時間的局所性

最近使った場所は、また使われやすい。

だから **最近の内容を残す**

配列の要素を順に読む



空間的局所性

使った場所の近くも、使われやすい。

だから **周囲もまとめて持つ**

この傾向を利用する

よく使いそうな一部を、近くにコピー。

キャッシュ

CACHE

アドレスは 16 進数、配列は 1 byte 要素の例。局所性はよく現れる傾向で、常に成立するとは限らない。

手元から遠くへ：記憶の階層



距離は取得時間の比喩。速度差には回路方式や容量も関わる。SSD / HDD は電源を切っても内容を保持する。

レジスタ・キャッシュ・主記憶は、役割が違う

どれも値を保持するが、**命令からの使い方**と**記憶回路**が違う。

記憶の種類	保持するもの・役割	どう指定するか	前回の回路とつなぐ
レジスタ	演算に使う値や結果 R1 = 5、R3 = 12	命令が名前で指定 ADD R3, R1, R2	レジスタ回路 CPU 内で少量を保持
キャッシュ	主記憶の一部のコピー アドレスと内容に対応づける	元のアドレスで照合 通常はハードウェアが管理	主に SRAM 高速だが、面積を使う
主記憶	実行中の命令とデータ キャッシュより大容量	アドレスで読み書き LOAD R3, [R1+0]	主に DRAM 高密度だが、応答に時間

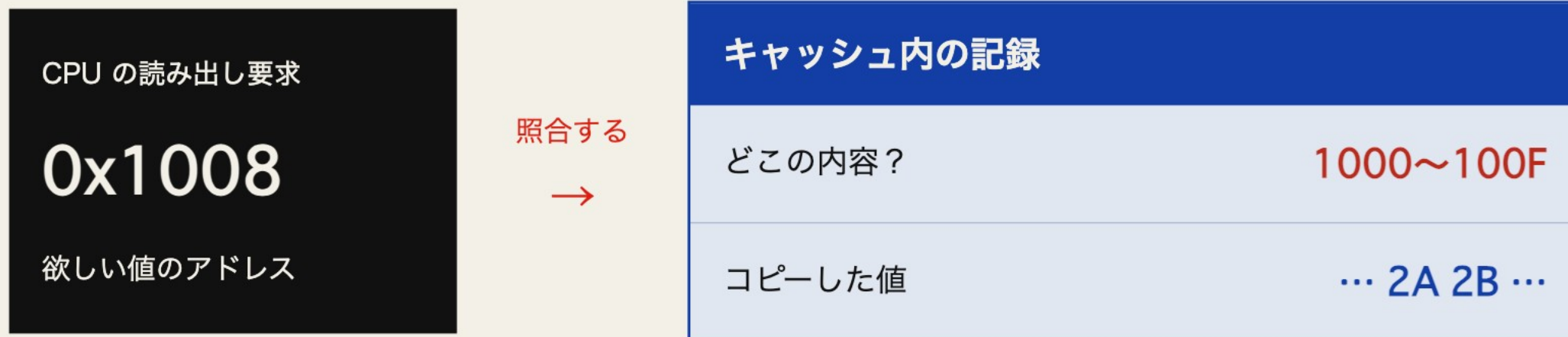
LOAD アドレスで値を読む → **R3 に保存する**

キャッシュはメモリの読み出しを速める。レジスタの代わりではない。

速さと容量の差には、距離だけでなく、回路方式・容量・照合や転送の仕組みも関わる。

キャッシュは「コピー」と「場所」を持つ

キャッシュには、**内容のコピー**と**元の場所を識別する情報**が必要。



HIT

コピーがある
キャッシュから返す

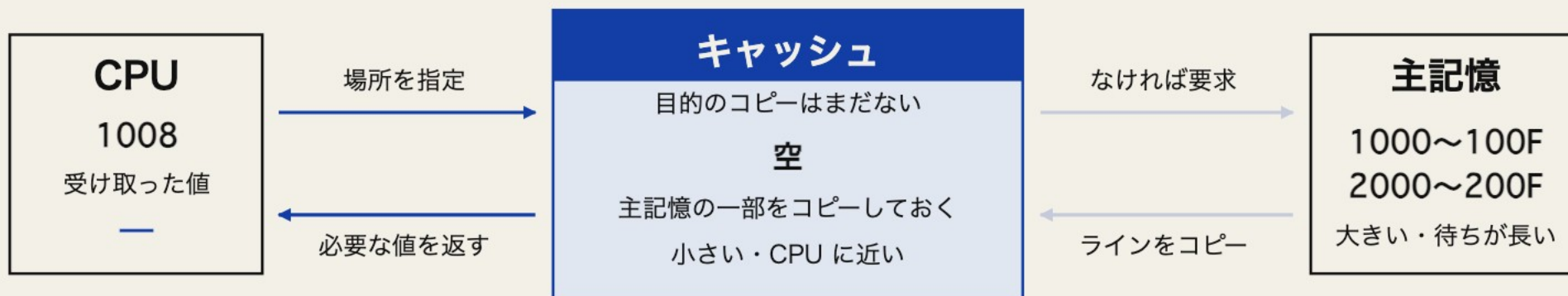
MISS

コピーがない
主記憶から取り込んで返す

CPU は元のアドレスを指定する。コピーの照合・取り込みは、通常ハードウェアが行う。

一度取り込むと、次は近くから返せる

1 byte の読み出しを追う

[最初へ](#)[自動再生](#)[前へ](#)[次へ](#)

コピーした 16 bytes

—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

準備

キャッシュは空。まず 0x1008 の 1 byte を読みたい。

0 準備

1 要求

2 MISS

3 取り込み

4 応答

5 次の要求

6 HIT

説明用：16 byte のラインを一つ保持。値・アドレスは 16 進数。読み出しだけを扱う。

キャッシュライン 16 bytes の例 / 数字は先頭からの位置



使ったラインを残す

隣の値も一緒に保持

容量には限りがある 空きがなければ、保持するラインを選んで置き換える。

次の読み出しは HIT? MISS?

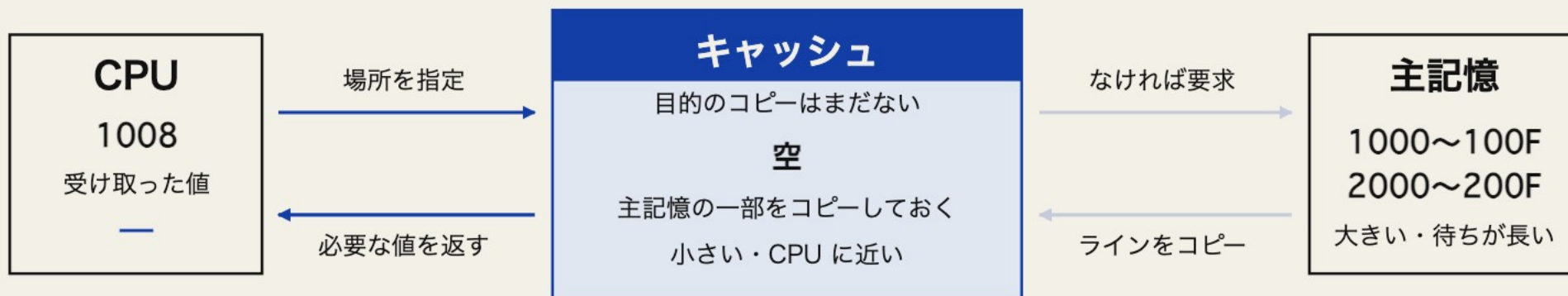
先に HIT / MISS を予想してから読む

空に戻す

同じ場所 1008

隣 1009

遠く 2000



コピーした 16 bytes

—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

予測する

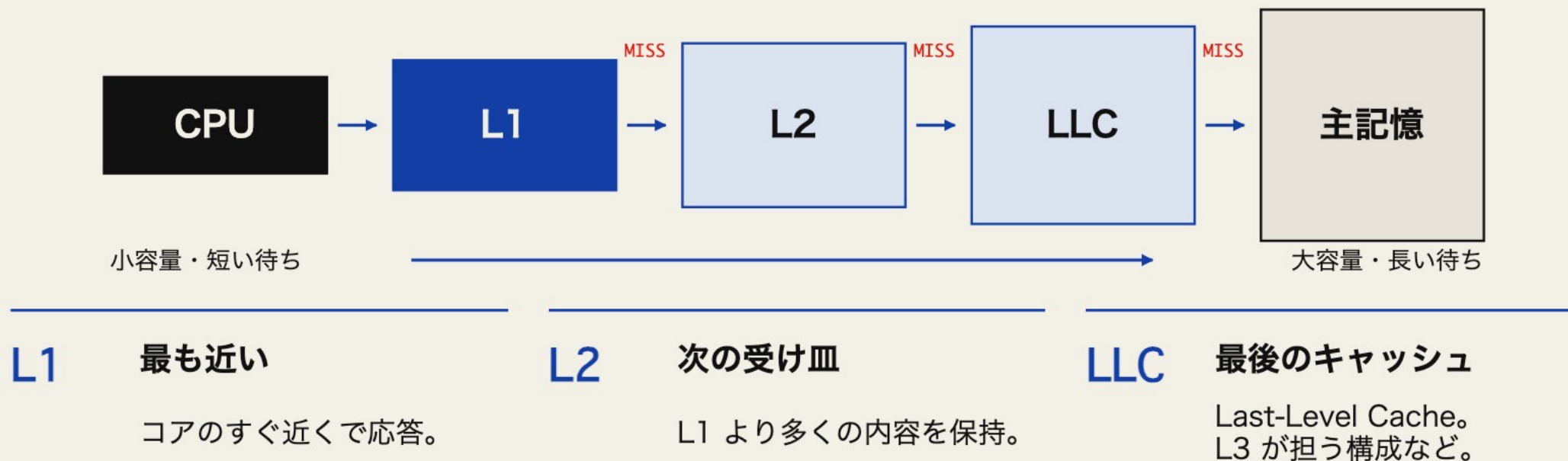
まず 0x1008 を読む。その後、同じ場所・隣・遠くを試す。

読み出しの履歴がここに残る

説明用：16 byte のラインを一つ保持。値・アドレスは 16 進数。読み出しだけを扱う。

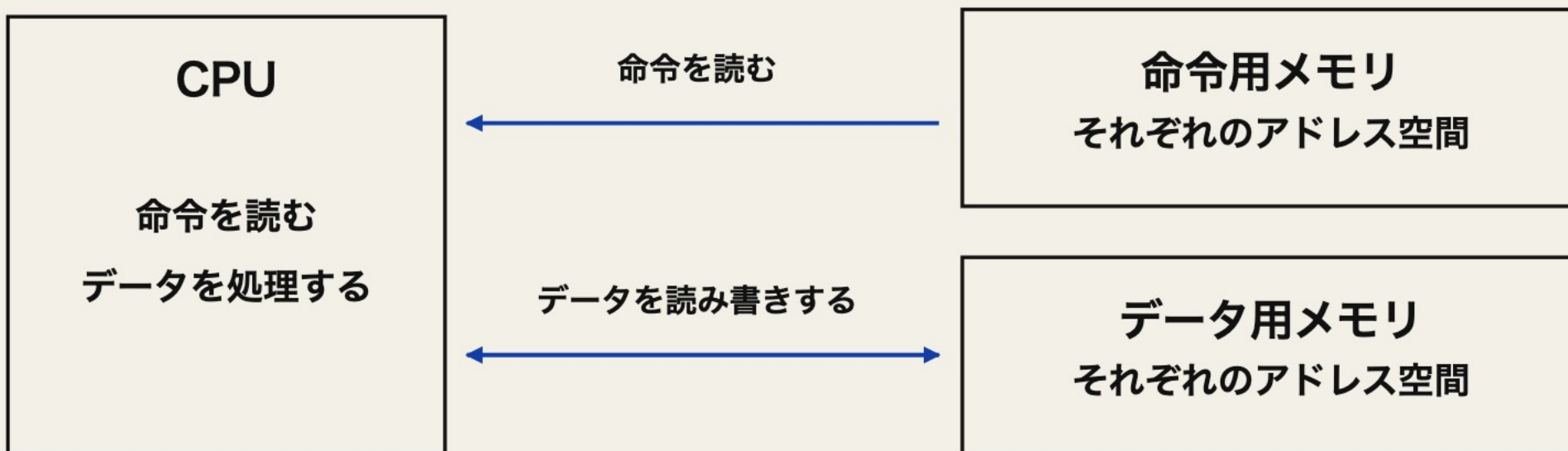
L1 がないときは、次の階層へ

近くに少量、遠くに大量。複数のキャッシュで速さと容量を両立する。



見つかった階層から返す 段数・容量・共有範囲は CPU によって異なる。

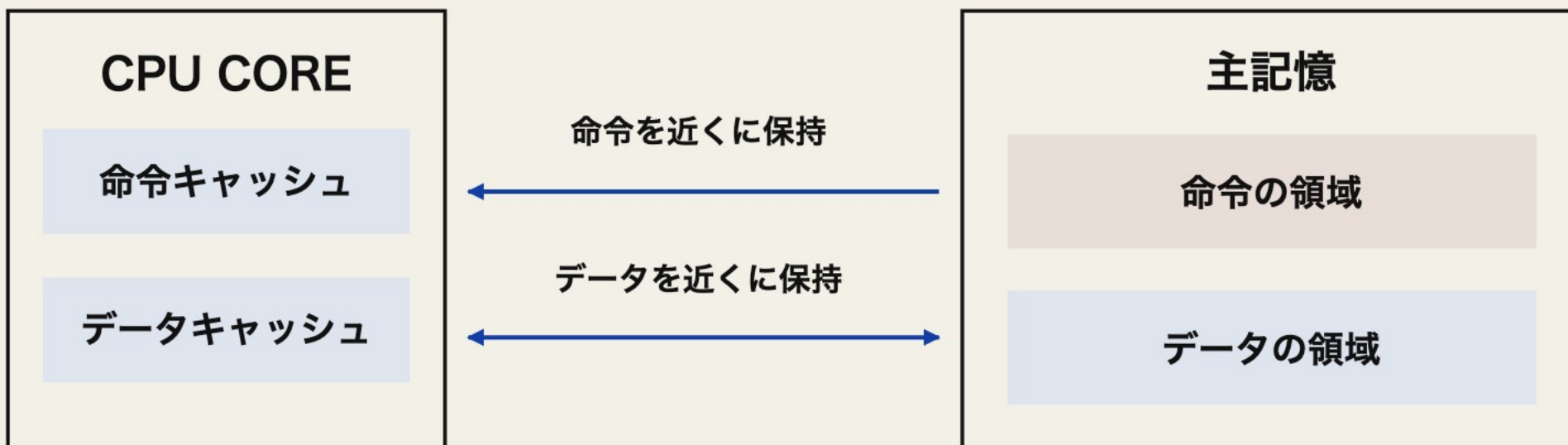
命令とデータに、別々の道を用意する



経路を分けると、命令とデータを並行して取得しやすい。

ハーバード型：命令とデータのメモリ・アドレス空間・経路を分ける。アドレスと制御信号の線は省略。

命令とデータは、近くで道を分ける



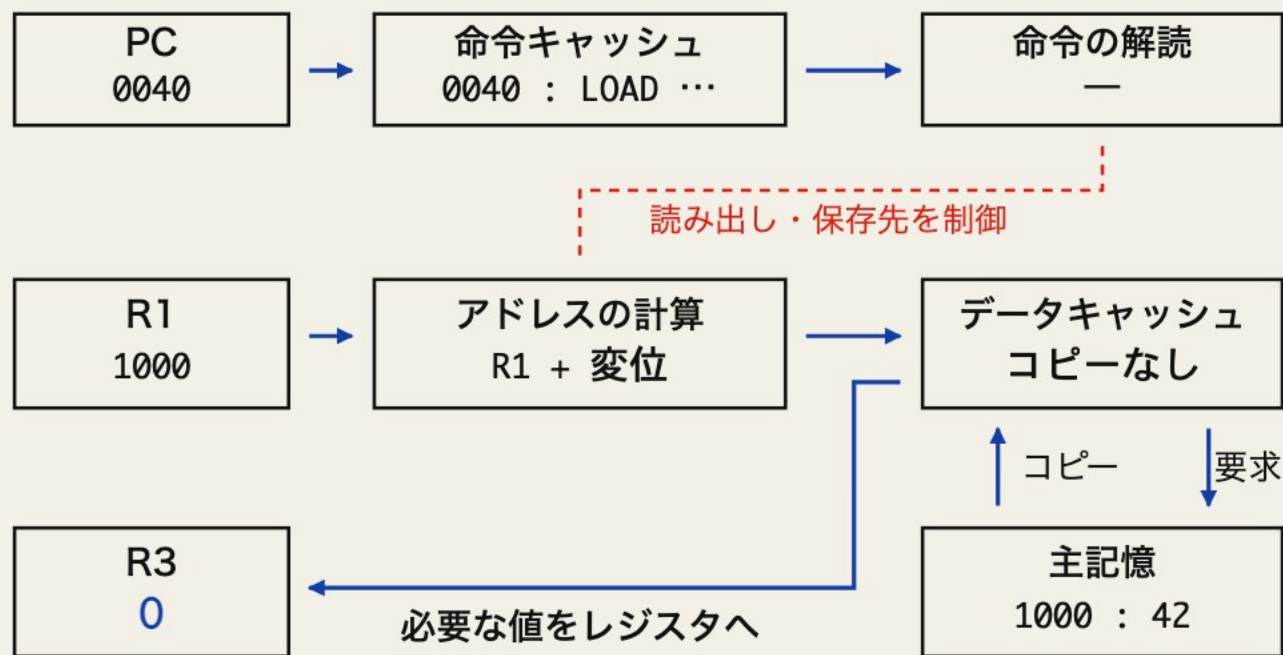
主記憶は共有し、CPU の近くでは命令とデータの道を分ける。

修正ハーバード型の構成例。命令の読み出しとデータの読み書きを、CPU の近くで分ける。

PC から R3 まで、一つにつなぐ

LOAD R3, [R1+0]

命令の場所：0040 / データの場所：1000

**0 準備**

1 命令を取る

2 意味を読む

3 場所を作る

4 データを探す

5 コピーを補う

6 結果を保存

7 次の命令へ

準備 PC は 0x0040。そこに `LOAD R3, [R1+0]` がある。

授業用モデル：命令・データ・レジスタは 16 bit。アドレスは 16 進数、値 42 は 10 進数。

[最初へ](#)[前へ](#)[次へ](#)

命令が、部品の状態を変える

冒頭の 5 + 7 に答える

ADD R3, R1, R2

R1 = 5 / R2 = 7

R3:0 → 12

CPU が命令を読み、その規則どおりに値を使い、結果を残す。

01 / アドレス

命令の場所と
データの場所

PC 0040 → 命令

R1 1000 → データ

LOAD はまず命令を取り、
その指定先から値を読む。

02 / 値と速さ

経路が違っても
値は同じ

HIT キャッシュ → 42

MISS 主記憶から補充 → 42

同じ内容なら R3 の値は同じ。
変わるのは待ち時間。

03 / 命令の種類

ADD と LOAD で
値の出所が違う

ADD R1・R2 → ALU → R3

LOAD メモリ → 値 → R3

レジスタ同士の ADD は
データメモリを読まない。

命令の約束は同じ。キャッシュは取得経路と待ち時間を変える。アドレスは 16 進数。

演習：値の変化を予想してから進める

「足し算」を開き、アセンブル → 予想 → 「1ステップ」で確かめる。

足し算の例・3 番目の命令

ADD A, B

PC = 2 / A = 12 / B = 7

一命令を実行すると

→ **A = 19 PC = 3**

B とメモリは変わらない

授業のモデル

ADD R3, R1, R2

別のレジスタ R3 に結果を保存。

命令は 16 bit、通常 PC は 2 増える。

[CPU シミュレーターを開く ↗](#)

演習のモデル

ADD A, B

A を $A + B$ で更新。A~D は 8 bit。

PC は命令番号で、通常 1 増える。

演習の LOAD A, 12 は、数値 12 を直接 A に入れる命令。次の OUT A は、何を変えるだろうか？

同じ結果でも、速さが違うのはなぜ？

LEC 5 → LEC 6

同じ仕事。
同じ結果。

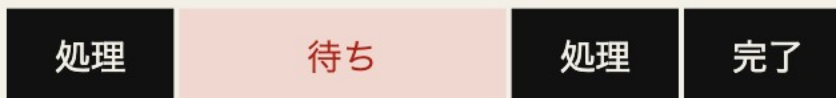
違うのは、
終わるまでの時間。

メモリ待ちが多い

時間 →



待ちを減らすと



時間の比率は説明用。実際の差は測定して確かめる。

01 / MEASURE

性能を測る

実行時間・処理量

02 / OVERLAP

仕事を重ねる

パイプライン・並列性

03 / BUILD

回路を形にする

設計・検証・チップ製造

NEXT / LECTURE 6

性能評価とチップ設計