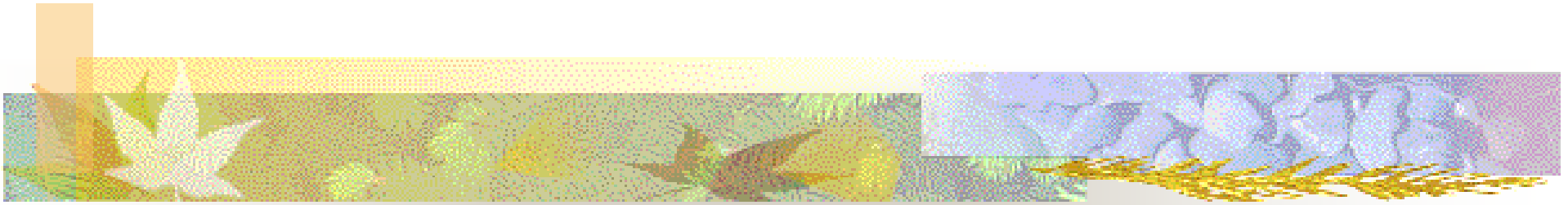
北海道大学
HOKKAIDO UNIVERSITY

# コンピュータシステム
# （アーキテクチャ　第４回）



工学部 情報エレクトロニクス学科
大学院 情報科学研究院 情報理工学部門
堀山 貴史

# 前回（アーキテクチャ第3回）の内容

機械語命令と内部動作（2）

- 主記憶アドレス参照方式

アーキテクチャの基本知識（1）
（分類と概観・初期のメインフレーム）

- 価格・時代・用途による汎用計算機の分類：
  - メインフレーム、ミニコン・ワークステーション、パソコン
- 黎明期～初期のメインフレーム技術
  - 機械式から電子式へ
  - ハードワイアドとノイマン型計算機
  - バッチ処理とタイムシェアリング
  - IOプロセッサとバス構成
  - ファミリー思想

# 今回の内容

## 機械語命令と内部動作(3)

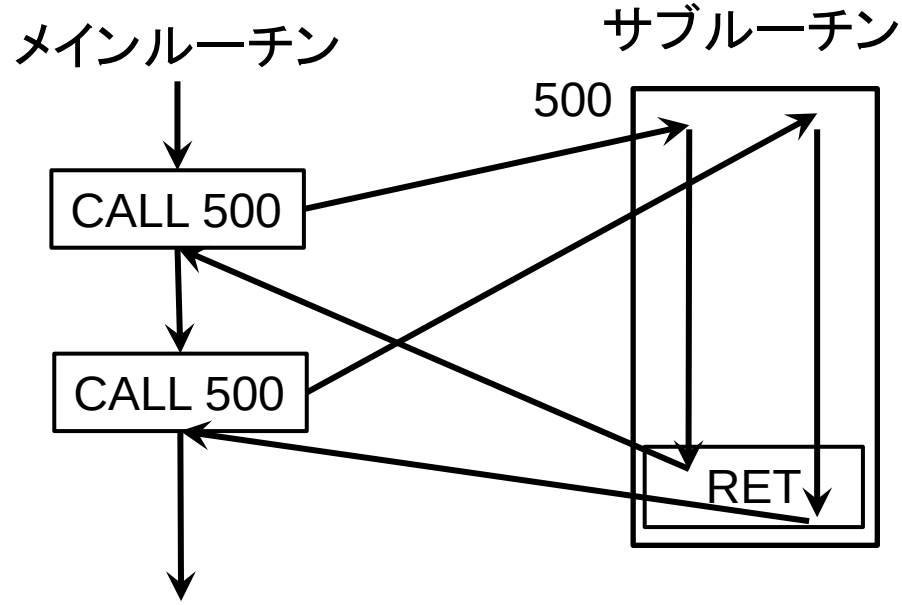
- サブルーチンコール、算術式とスタック

## アーキテクチャの基本知識(2)
(メインフレームの発展)

- マイクロプログラム
  - エミュレーション、ファームウェア
- メインフレーム互換機
- 仮想メモリ、仮想マシン
- ベクトル計算機、並列計算機
  - スーパーコンピュータ
  - パイプライン処理とマルチプロセッシング、並行と並列
- CISC と RISC

# サブルーチンコール(sub-routine call)

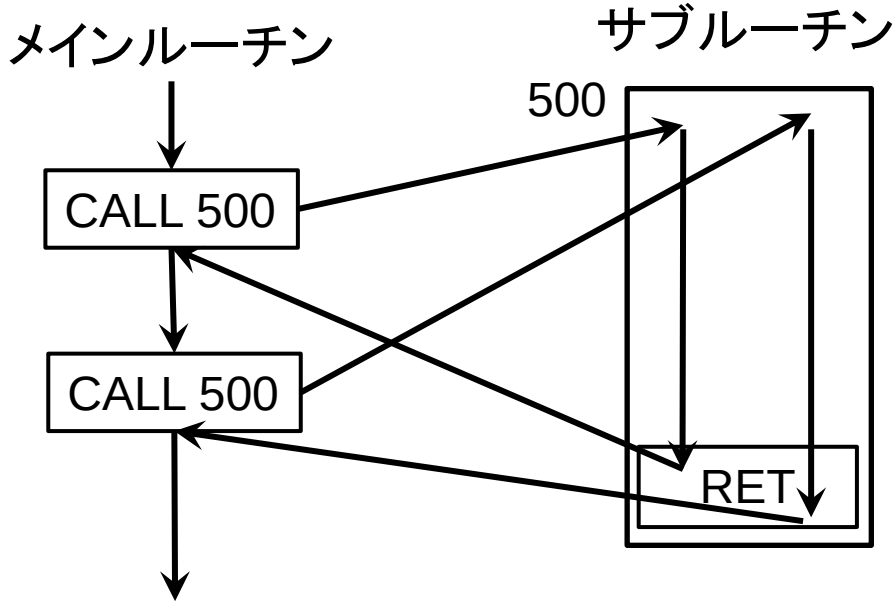
- プログラムの中に、同じ部分計算が何度も現れるときは、1か所にまとめたい
  (例えば、加算とシフトを組合せた乗算サブルーチンなど)
  - **全体の命令数を削減して、メモリ使用量を節約できる**
  - **部分計算を保存・再利用することでプログラム開発を効率化**




機械語だけでなく、プログラム一般のお話です

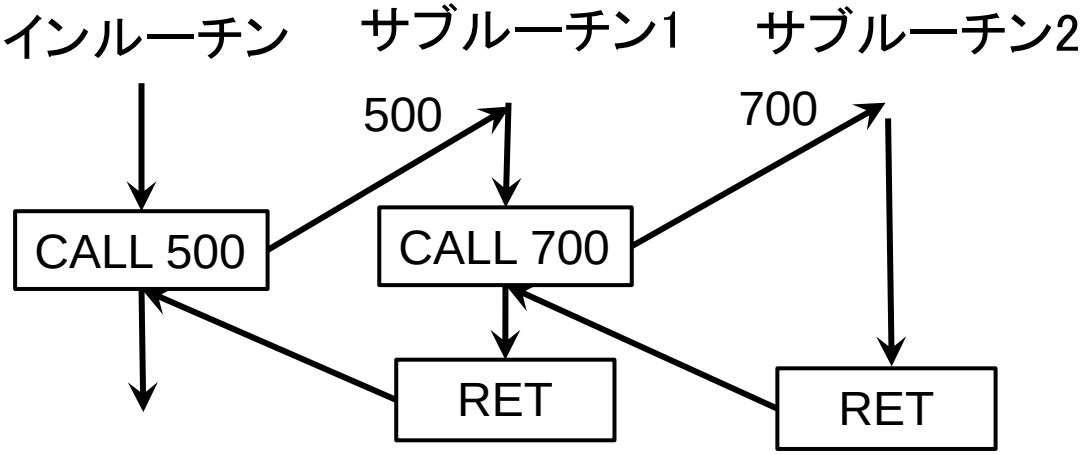
# サブルーチンコール(sub-routine call)

- プログラムの中に、同じ部分計算が何度も現れるときは、1か所にまとめたい
  (例えば、加算とシフトを組合せた乗算サブルーチンなど)
  - **全体の命令数を削減して、メモリ使用量を節約できる**
  - **部分計算を保存・再利用することでプログラム開発を効率化**
- プログラムカウンタに行先番地をセットすればジャンプ
- 戻るときのために呼出し元の番地を保存する必要がある。
- メインとサブの間でデータを受け渡すための記憶場所を決めておく必要がある



# 多重サブルーチンコール

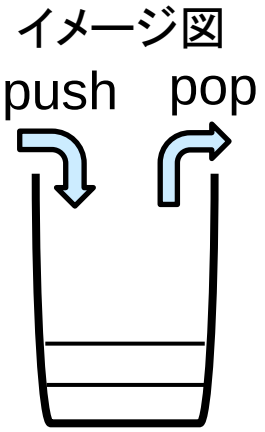
- サブルーチンから別のサブルーチンを呼び出すこともある
  - ２つ以上前の戻り先も憶えておく必要がある

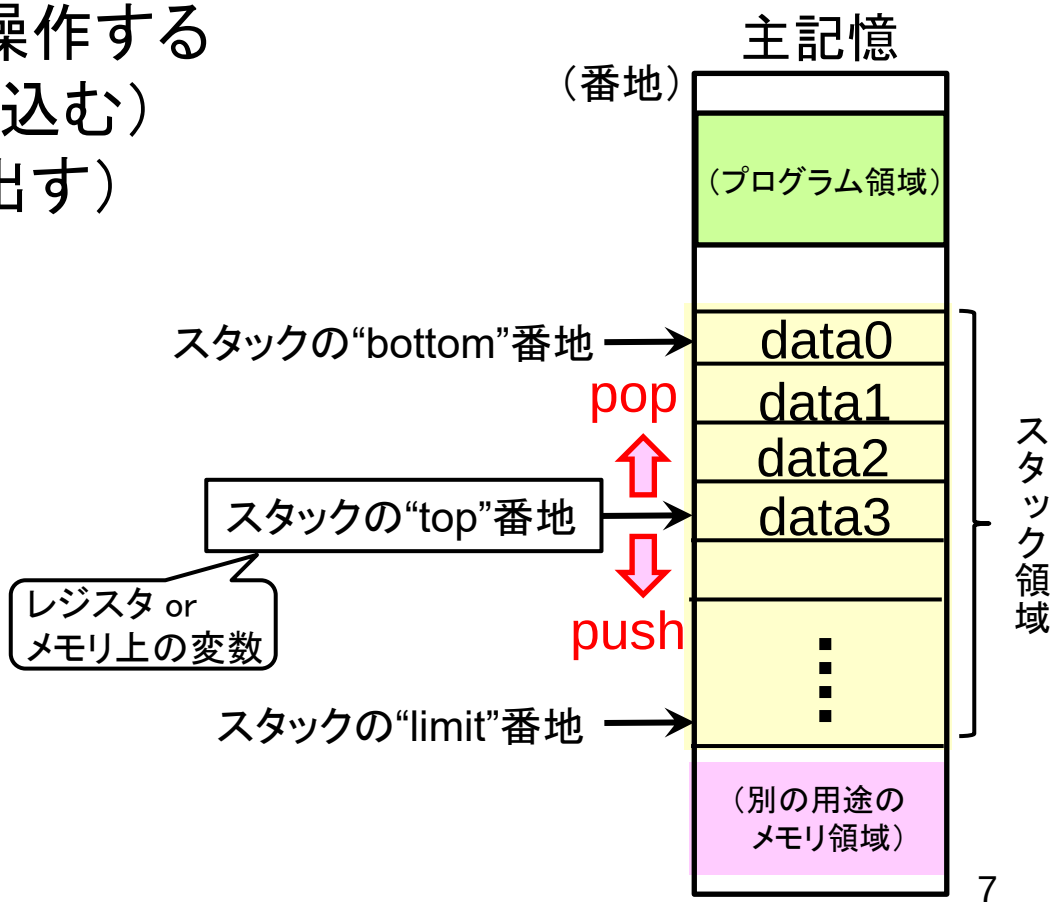



- 自分自身をサブルーチンで呼び出すこともある
（再帰呼び出し；recursive call）
  - （例）n ! = n × (n-1) ! を計算するサブルーチン
  - 計算途中のレジスタの内容も退避する必要がある

# スタック(stack)

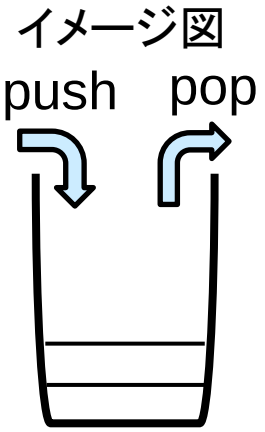
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

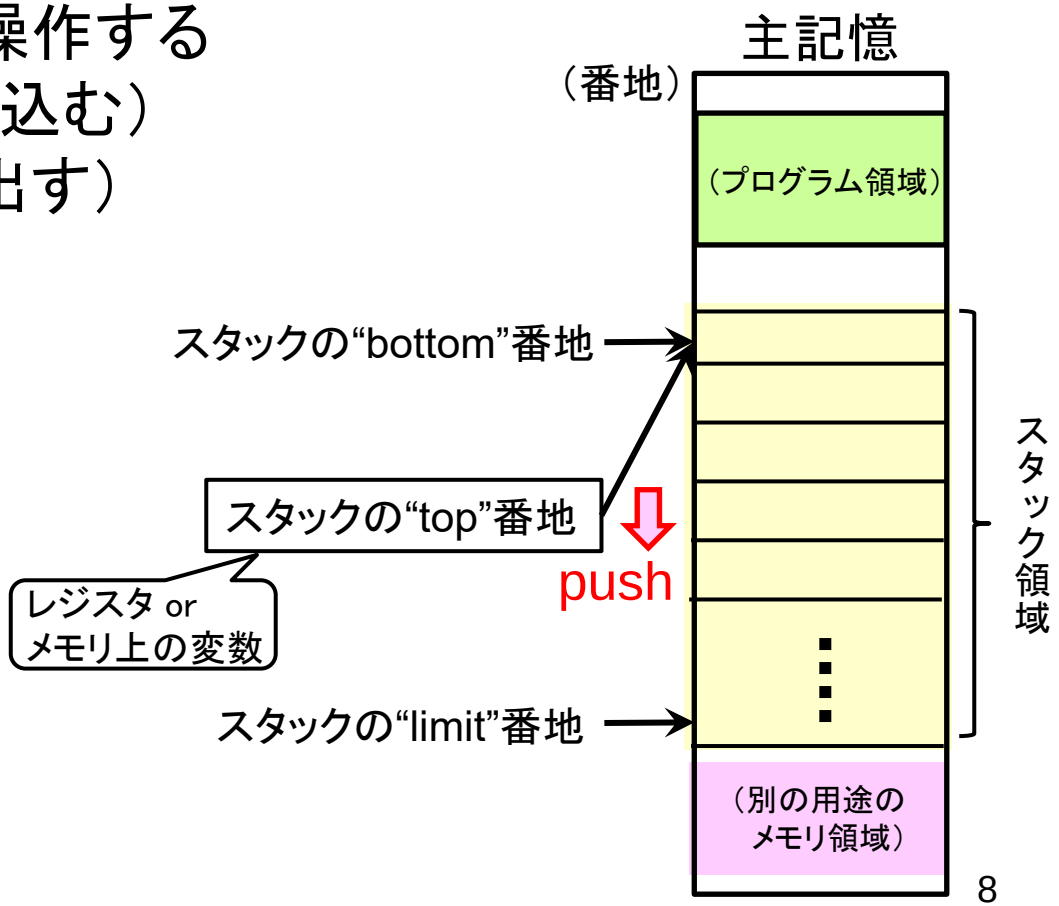


# スタック(stack)

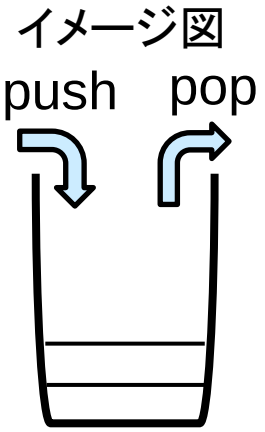
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

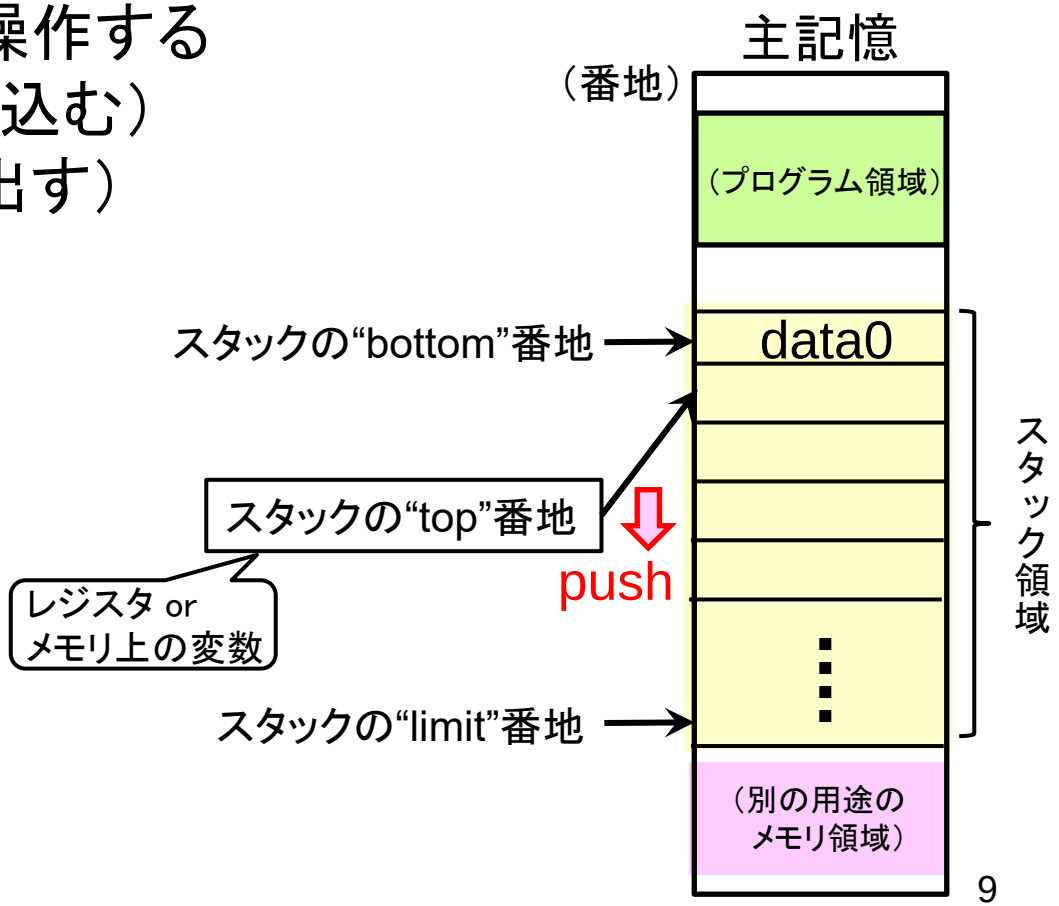


# スタック(stack)

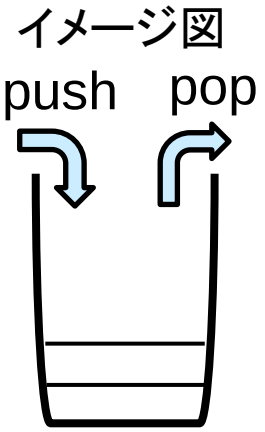
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

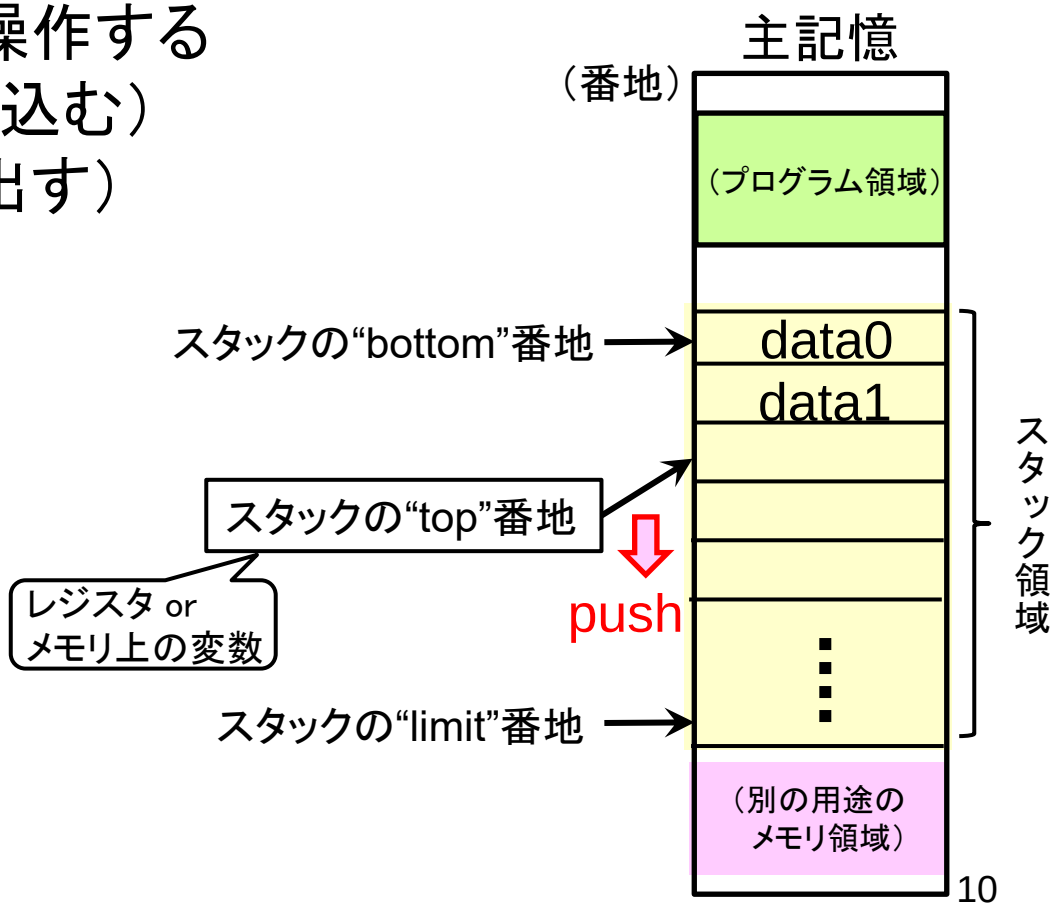


# スタック(stack)

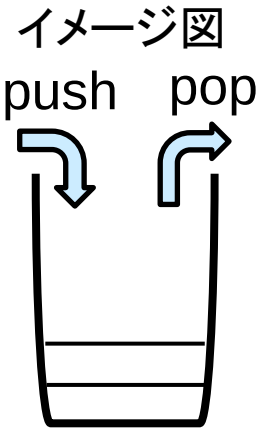
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

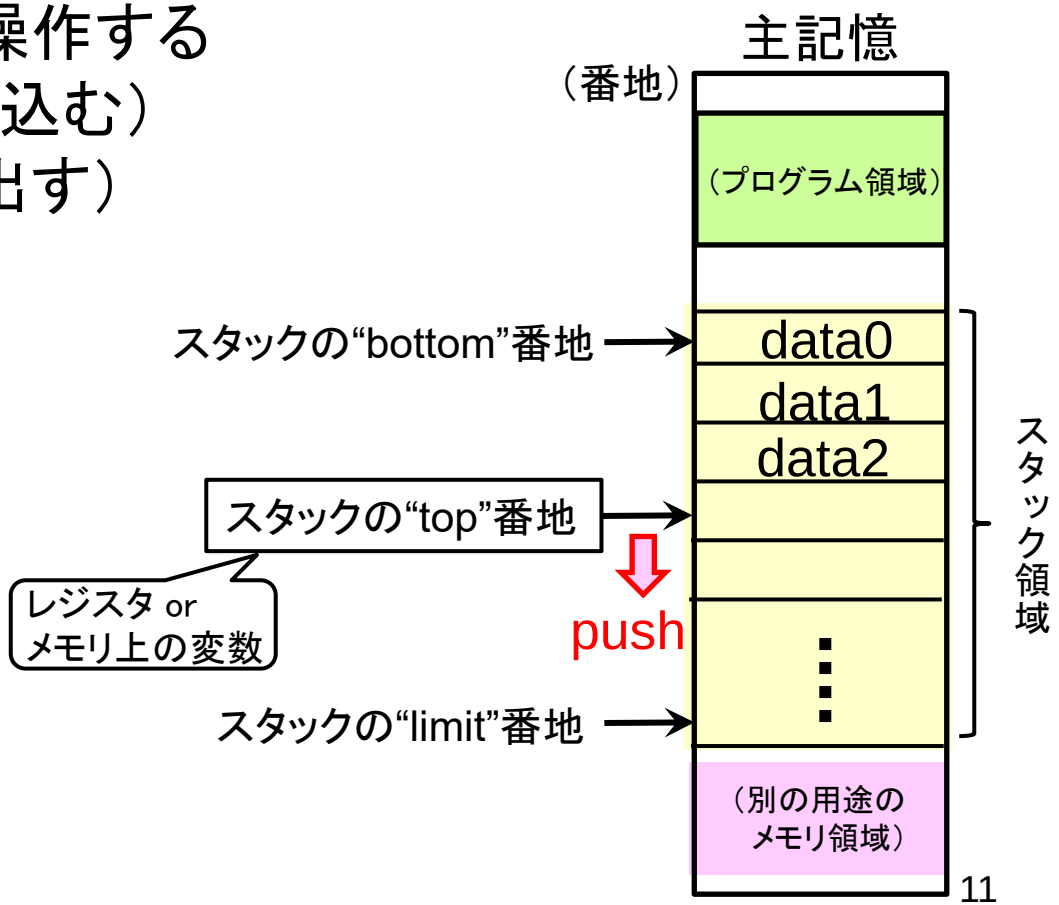


# スタック(stack)

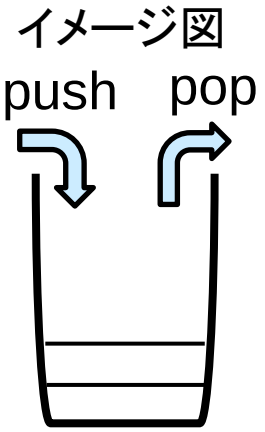
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

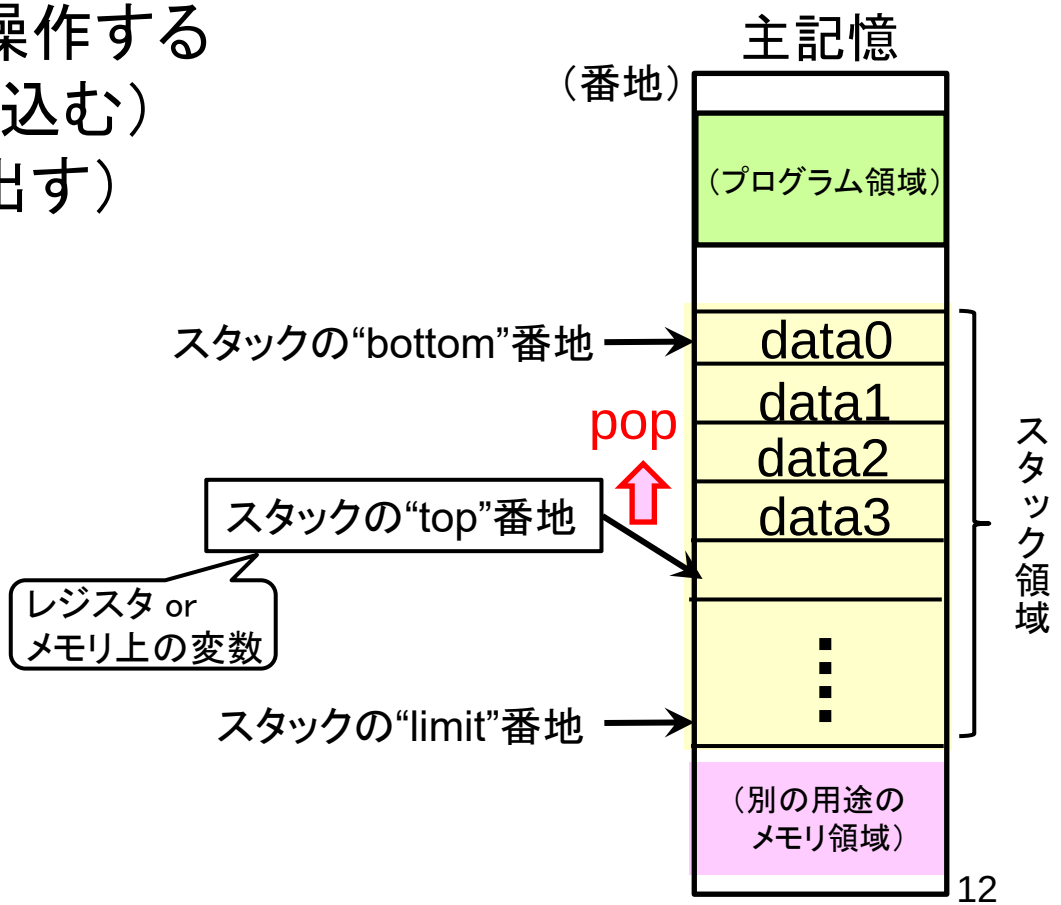


# スタック(stack)

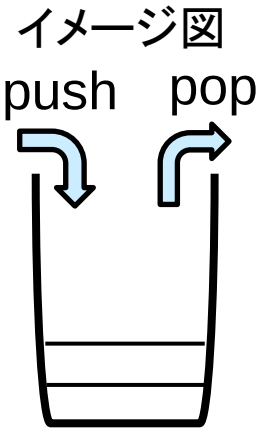
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

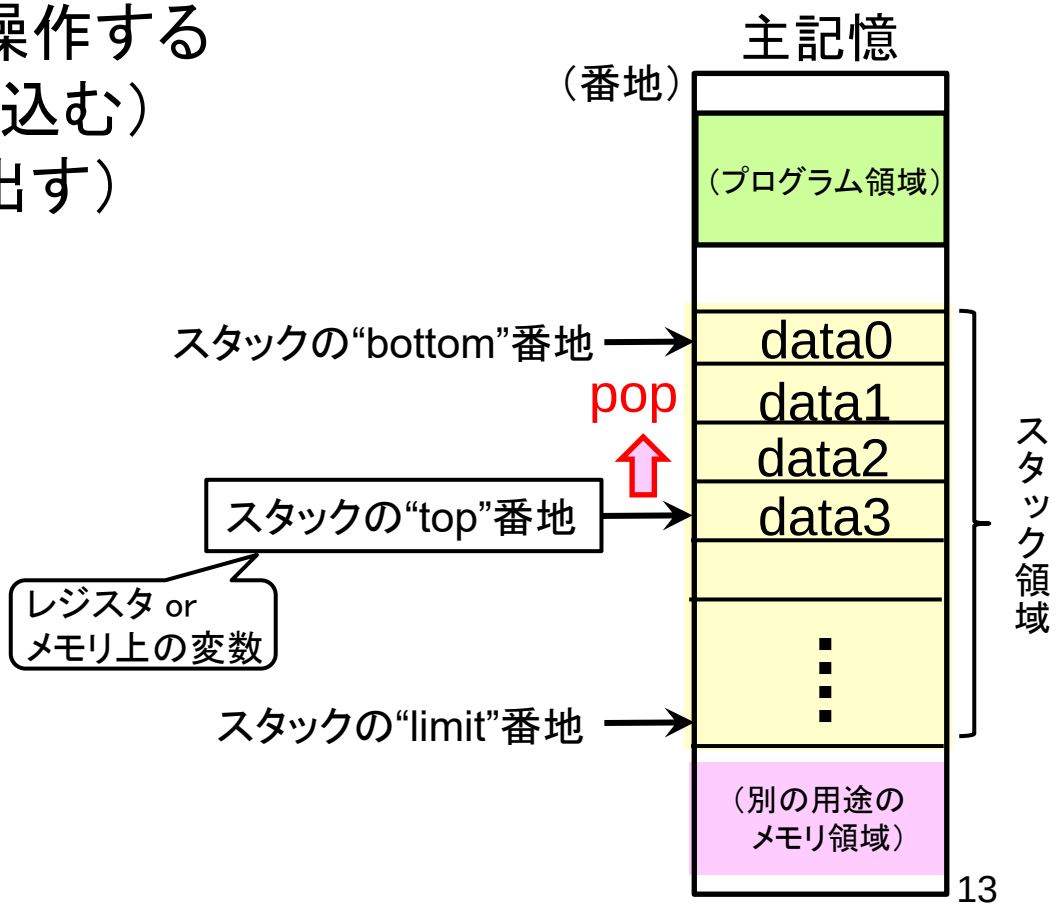


# スタック(stack)

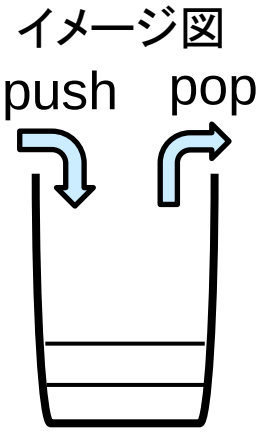
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)

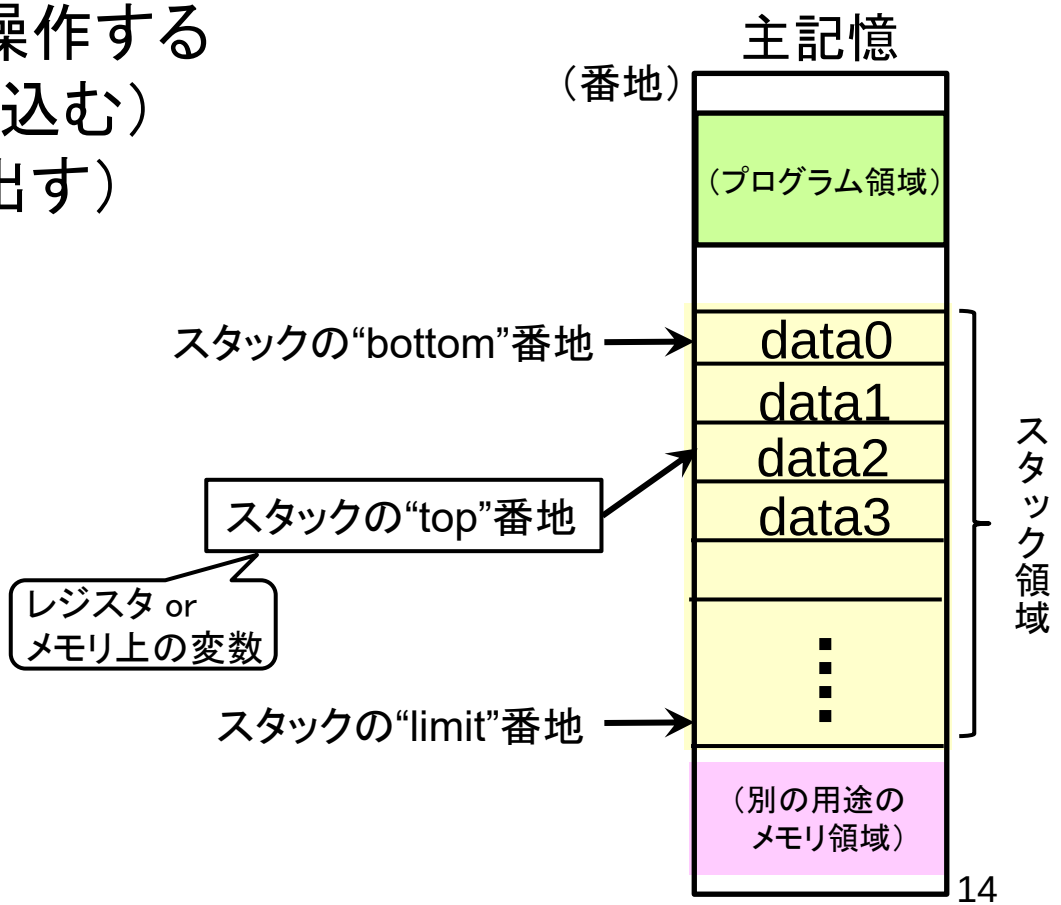


# スタック(stack)

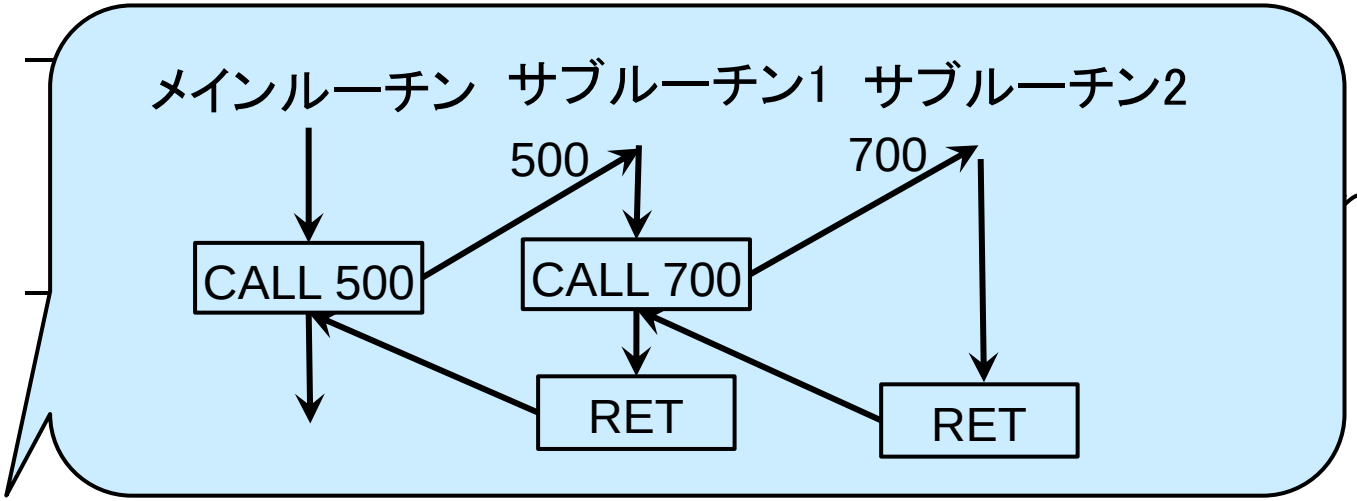
- 基本的なデータ構造の1つ
  - 後から入ったデータから順に取り出せる構造
    (First-In-Last-Out)
    (参考)First-In-First-Outの場合はキュー(queue)と呼ばれる
  - 2種類の命令だけを使って操作する
    Push命令(データを1つ書き込む)
    Pop命令(データを1つ読み出す)



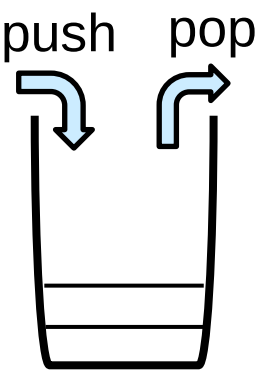
# スタック(stack)

- 基本的なデータ構造の1つ

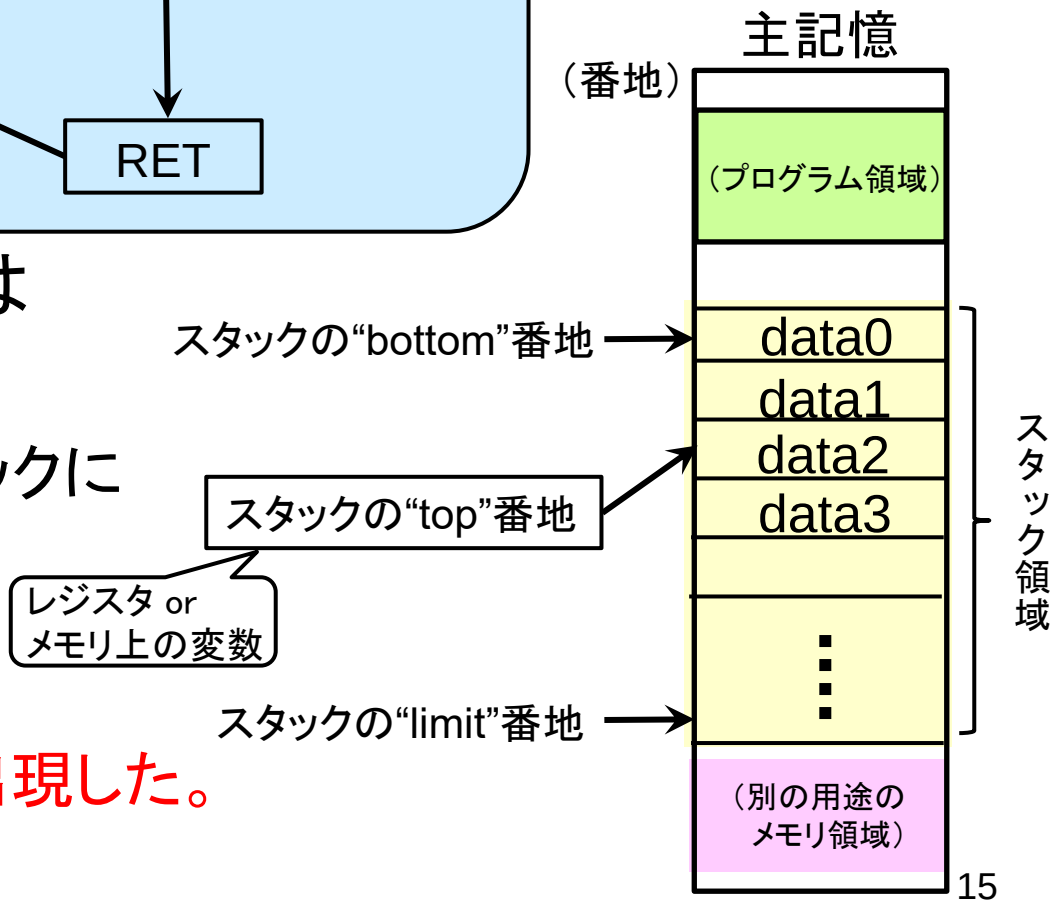



れる

イメージ図




- サブルーチンの戻り番地は
  スタックで管理できる。
  - 内部レジスタの状態もスタックに
    pushすれば退避できる。
    → サブルーチン用に
    スタックを直接操作する
    機械語命令を持つ機種も出現した。

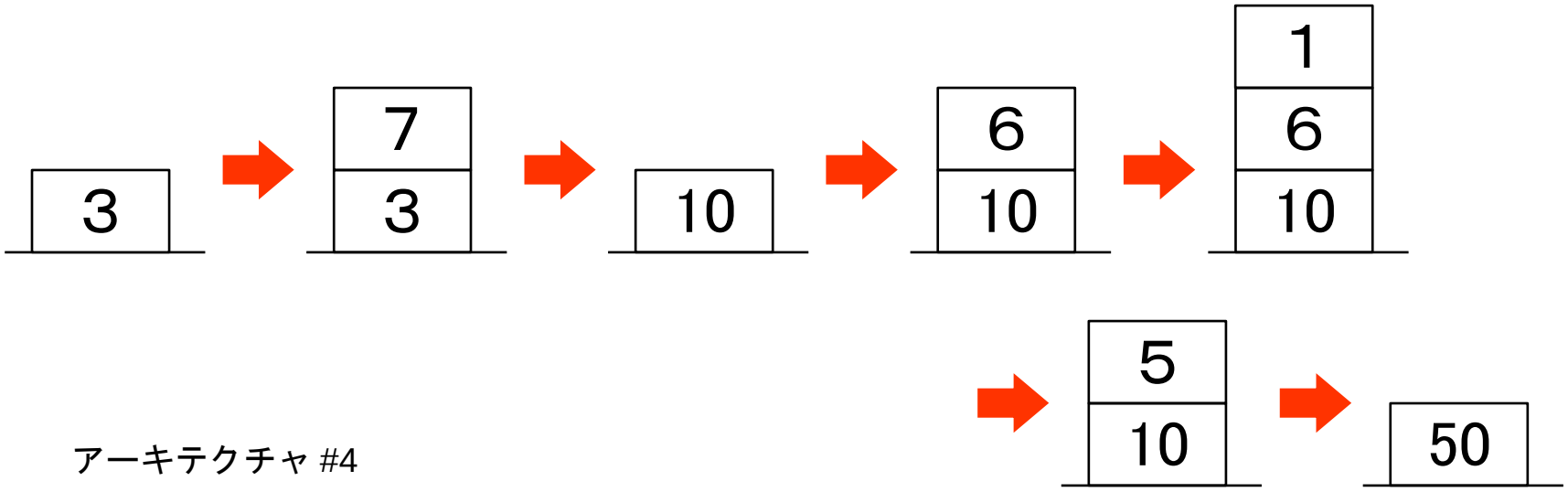


# 算術式とスタック

- 複数の演算を組合せた算術式も、スタックを使えば処理できる
  - （例）（３＋７）×（６－１）

３に７を足して(＋)、
６から１を引いた(－)ものを
かける(×)

３を push;　７を push;　2回 pop して加算結果を push;
６を push;　１を push;　2回 pop して減算結果を push;
2回 pop して乗算結果を push;　popして出力



# 算術式とスタック

- 複数の演算を組合せた算術式も、スタックを使えば処理できる
  - （例）（３＋７）×（６－１）

３に７を足して (＋)、
６から１を引いた (－) ものを
かける (×)

  3 を push;　7 を push;　2回 pop して加算結果を push;
  6 を push;　1 を push;　2回 pop して減算結果を push;
  2回 pop して乗算結果を push;  popして出力

- Accumulatorを持たず、スタック操作だけですべて計算するスタックマシンと呼ばれるアーキテクチャも提案された。
  - 機械語命令が簡潔になり数学的に美しいが、必ずしも性能が良くなるわけではない。
    （今でもJavaの中間コードなどで仮想的に使われている）

# 休 憩

- ここで、少し休憩しましょう。

- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

# 今回の内容

**機械語命令と内部動作（3）**

- サブルーチンコール、算術式とスタック

**アーキテクチャの基本知識（2）**
**（メインフレームの発展）**

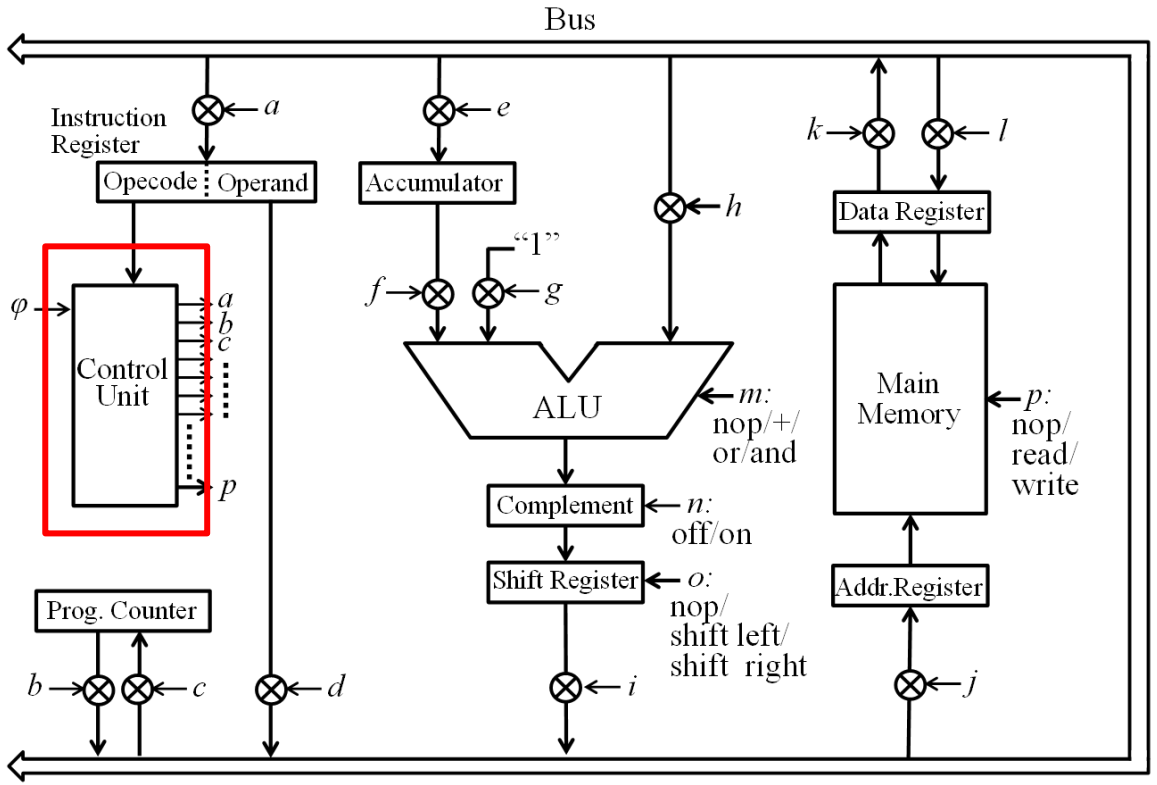
- マイクロプログラム
  - エミュレーション、ファームウェア
- メインフレーム互換機
- 仮想メモリ、仮想マシン
- ベクトル計算機、並列計算機
  - スーパーコンピュータ
  - パイプライン処理とマルチプロセッシング、並行と並列
- CISC と RISC

# マイクロプログラム方式

- 計算機の機械語命令は新機種が出るたびに進化を続けた。
  - 旧機種用のプログラムも動かないと困る。
  - さまざまな機種用の動作モードを用意するため、命令デコーダの電子回路が複雑化 → 複雑になり過ぎると動作速度が低下する
- 命令デコーダの内部を書き換え可能にしたい！

**マイクロプログラム方式**

命令デコーダの内部信号線のON/OFFの手順をコード化してメモリに保存。これを読み出して制御するようにした。



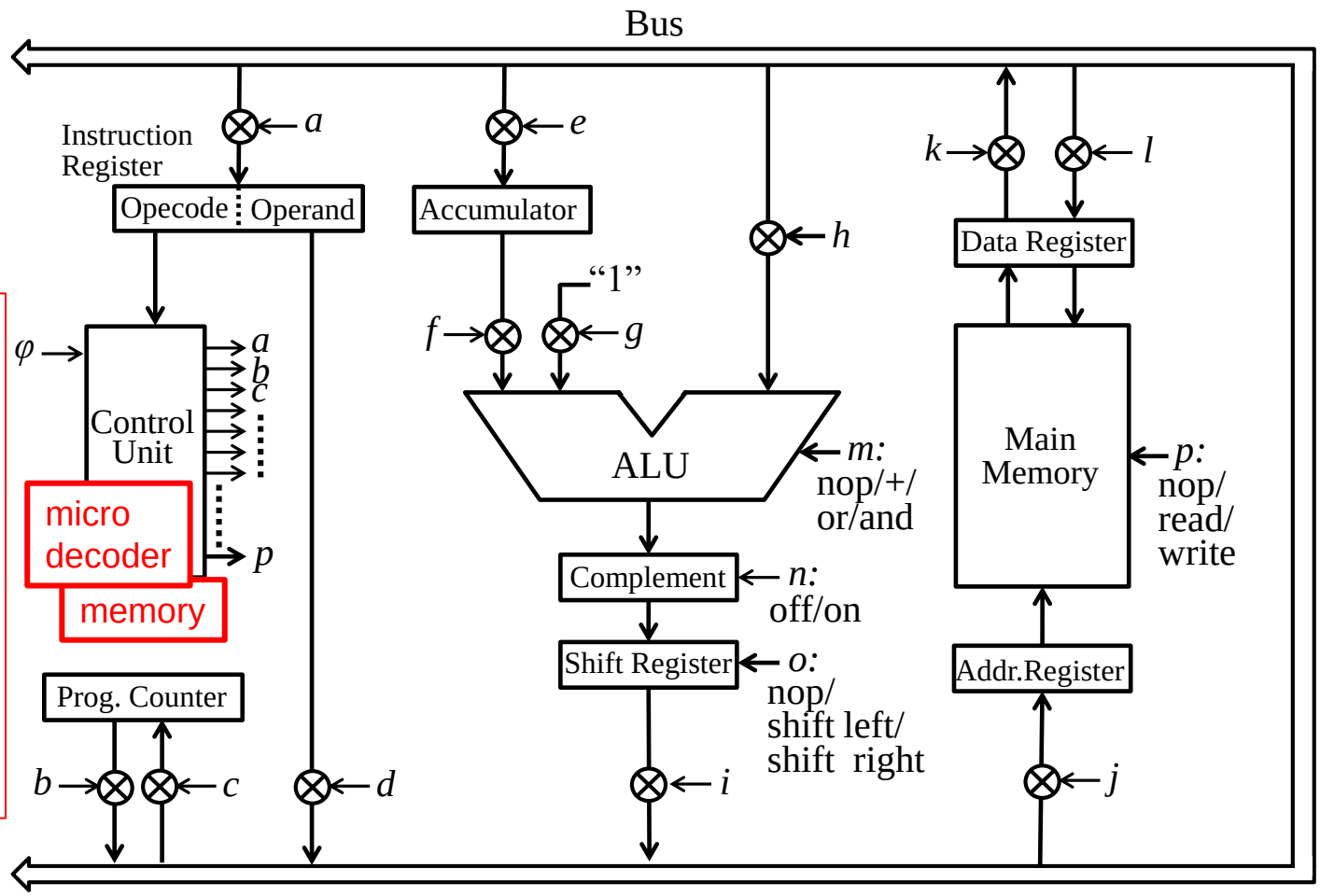
# 機械語命令とマイクロ命令

- 1つの機械語命令は、いくつかのマイクロ命令で構成されている。
- マイクロ命令を組合せれば、多様な機械語命令を簡単に作り出せる

Accの内容と200番地の内容を加算し、結果をAccに保持

A 200
clock3: *d*, *j*, *p*←"read"
clock4: *k*, *h*, *f*, *m*←"+"
clock5: *i*, *e*
clock6: *b*, *h*, *g*, *m*←"+"
clock7: *i*, *c*
（次のフェッチ動作へ）

第２回資料より



# マイクロプログラム方式による計算機設計

- ある程度の広い機械語命令をカバーできるように、マイクロ命令セットを決めておく
  - 各マイクロ命令を実行できるように、命令デコーダ内部の電子回路(マイクロ命令デコーダ)を作っておく
- 製造する機種の機械語命令セットを決める。
- 各機械語命令をマイクロ命令の組合せで記述する。(→ マイクロプログラミング)
  - 記述したマイクロプログラムを命令デコーダの内蔵メモリに書き込む

# マイクロプログラム方式による計算機設計

- ある程度の広い機械語命令をカバーできるように、マイクロ命令セットを決めておく
  - 各マイクロ命令を実行できるように、命令デコーダ内部の電子回路(マイクロ命令デコーダ)を作っておく
- 製造する機種の機械語命令セットを決める。
- 各機械語命令をマイクロ命令の組合せで記述する。(→ マイクロプログラミング)
  - 記述したマイクロプログラムを命令デコーダの内蔵メモリに書き込む
- 機械語命令セットを変更する必要のない計算機の場合は、読み出し専用メモリ(ROM; Read Only Memory)を内蔵させる
  - マイクロプログラム方式の命令デコーダを一度設計してしまえば、命令セットを変更しない従来の計算機も容易に設計できる。
    → 製造コストが下がる → 量産化、低価格化

# エミュレーションとシミュレーション

- 計算機が他の機種を模倣し、他機種用の機械語を実行することをエミュレーション(emulation)と呼ぶ。
  - シミュレーション(simulation)は物理現象などをモデル化して、計算機で模擬すること。エミュレーションとは意味が異なる。

# エミュレーションとシミュレーション

- 計算機が他の機種を模倣し、他機種用の機械語を実行することをエミュレーション(emulation)と呼ぶ。
  - シミュレーション(simulation)は物理現象などをモデル化して、計算機で模擬すること。エミュレーションとは意味が異なる。
- マイクロプログラムを書き換えて他機種に化けて実行
  **→ ハードウェアエミュレーション**
  - 高速に実行できるが、マイクロプログラムの書き換え作業が必要
  - 複数機種の機械語プログラムをタイムシェアリングで実行できない
- 機械語プログラムで他機種用のプログラムを仮想的に実行
  **→ ソフトウェアエミュレーション**
  - 低速だが、複数機種の機械語プログラムもタイムシェアリングで実行できる。

# マイクロプログラムとファームウェア

- マイクロプログラムのことをファームウェア(firmware)とも呼ぶ
  - ハードウェアを設定するためのプログラム
  - ハードウェアとソフトウェアの中間に位置する
  - マイクロプログラムでなくても、何らかのハードウェア設定をするためのプログラムをファームウェアと呼ぶことがある
- ファームウェアにバグがあると、計算機が壊れることがある
  - 例えば複数ゲートが同時にONになると、過電流が流れ燃え出す
- ファームウェア更新中に停電などで止まった場合も深刻な状態になることがある。

# メインフレーム互換機

- IBM System/360シリーズの登場(1964年～)
  - 完全な上位互換性のある機械語命令セット(ファミリー思想)
  - 4ビット10進数ではなく2進数。2の補数表現。浮動小数点にも対応
  - 商用OSもIBMが自社開発して搭載(OS/360)
  - IBMが汎用商用計算機の「巨人」に → 1強多弱の業界に
    (独占禁止法により米国司法省と係争も)
- メインフレーム全盛期(～1980年代)
  - アーキテクチャの枠組みが統一されたため、その上で多くの技術者による様々な技術が発展。**→ OS、仮想化技術、並列化技術、メモリアーキテクチャ、ストレージ技術(ハード/フロッピーディスク)**
  - 360シリーズから370シリーズへ、さらにその拡張
  - **IBM互換機メーカーや周辺機器メーカーが出現(コバンザメ商法)**
- 日本の計算機メーカーとの攻防(当時の通産省が政策的に育成)
  - IBM互換路線(日立、三菱、富士通など)　非互換路線(NECなど)
  - IBM産業スパイ事件(1982年)　日米通商摩擦の種に

# 休 憩

- ここで、少し休憩しましょう。

- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

# 今回の内容

**機械語命令と内部動作（3）**

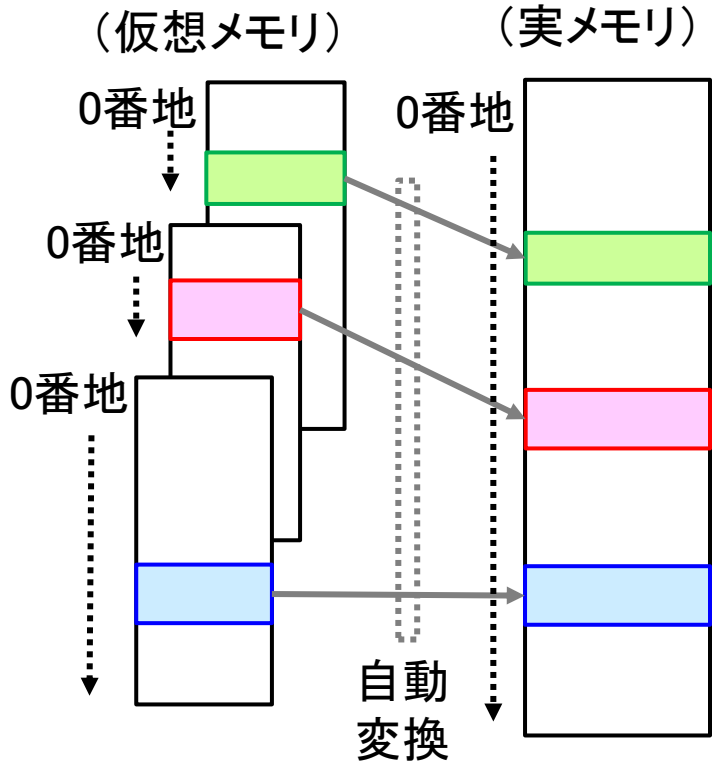
- サブルーチンコール、算術式とスタック

**アーキテクチャの基本知識（2）**
**（メインフレームの発展）**

- マイクロプログラム
  - エミュレーション、ファームウェア
- メインフレーム互換機
- 仮想メモリ、仮想マシン
- ベクトル計算機、並列計算機
  - スーパーコンピュータ
  - パイプライン処理とマルチプロセッシング、並行と並列
- CISCとRISC

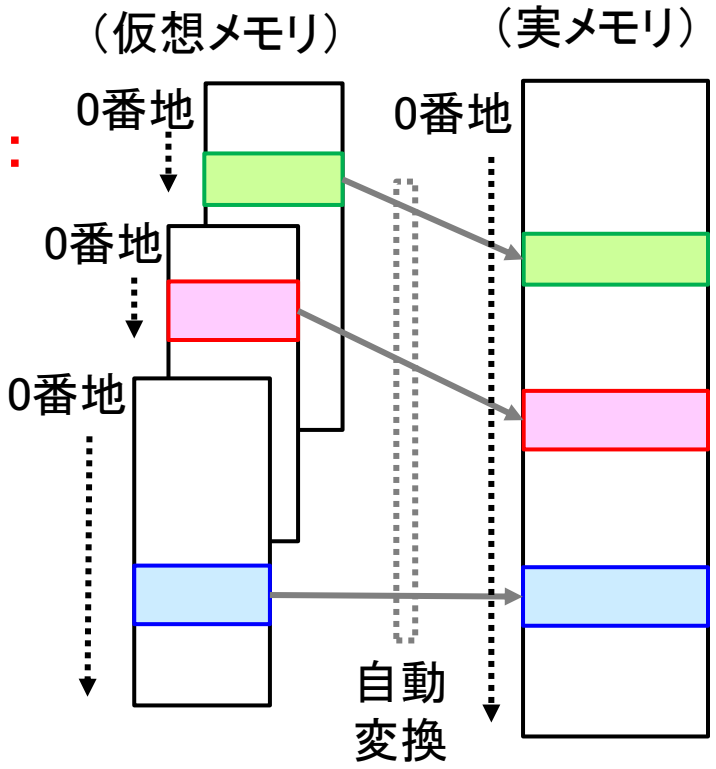
# 仮想メモリ技術

- タイムシェアリングなどで複数プログラムを（見かけ上）同時に実行する場合、主記憶サイズが足りなくなることがある
- 2次記憶からプログラムの一部のブロックだけを主記憶に持ってきて、必要に応じて入れ替えるという技法が開発された



# 仮想メモリ技術

- タイムシェアリングなどで複数プログラムを（見かけ上）同時に実行する場合、主記憶サイズが足りなくなることがある
- 2次記憶からプログラムの一部のブロックだけを主記憶に持ってきて、必要に応じて入れ替えるという技法が開発された
  - 実行中のプログラムのメモリ番地は、実行するたびに異なる。
    → **仮想メモリ（virtual memory; 仮想記憶）**：プログラムを入れ替えるたびにメモリ番地を振り直していたらたいへんなので、物理アドレスではなく、プログラムごとに0番地から始まる論理アドレスでアクセスできるようにした。
  - 機械語のオペランドには論理アドレスを記入。命令デコーダ内部に物理アドレスへの変換テーブルを設置。



# 仮想メモリと仮想マシン

- 仮想メモリ技術は、ハードウェアで実現する場合とソフトウェア（OS)で実現する場合がある。
  - ハードウェアで実現した仮想メモリは、プログラム同士でアドレス空間が完全に分かれるので、一方のプログラムがバグなどで暴走しても、他のプログラムが影響を受けない。（→高信頼性）
  - 仮想メモリを使うと実メモリ容量より大きな空間を用意できる。（ただし広い空間にランダムアクセスすると著しく性能低下する）

# 仮想メモリと仮想マシン

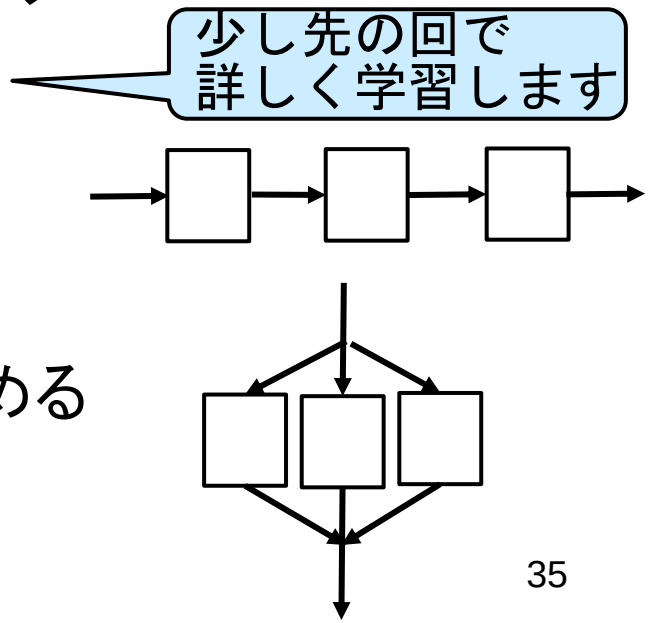
- 仮想メモリ技術は、ハードウェアで実現する場合とソフトウェア（OS)で実現する場合がある。
  - ハードウェアで実現した仮想メモリは、プログラム同士でアドレス空間が完全に分かれるので、一方のプログラムがバグなどで暴走しても、他のプログラムが影響を受けない。（→高信頼性）
  - 仮想メモリを使うと実メモリ容量より大きな空間を用意できる。（ただし広い空間にランダムアクセスすると著しく性能低下する）
- 仮想メモリをさらに発展させ、CPUや入出力まで仮想化したものを**仮想マシン（virtual machine; 仮想機械）**と呼ぶ。
  - プログラムごとに個別の計算機を使っているように見える。
  - プログラムごとに異なるOSを走らせることもできる。
  - 最近では個人用PCでも使える。クラウド型のサービスもある

# ベクトル計算機・並列計算機

- メインフレームの高性能化の競争が激化（1960年代～現在まで）
  - クロック周波数を上げる（→ 電子回路の高速化）
  - 問題を分割してそれぞれを同時に処理（→ 集積回路の大規模化）
- **その当時の最高性能を目指した計算機を「スーパーコンピュータ」(super computer)と呼ぶ**
  - **「これを超えたらスーパーコンピュータ」という明確な境界はない。**

# ベクトル計算機・並列計算機

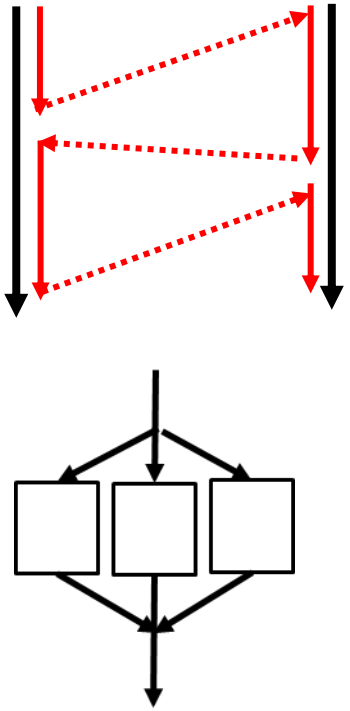
- メインフレームの高性能化の競争が激化（1960年代～現在まで）
  - クロック周波数を上げる（→ 電子回路の高速化）
  - 問題を分割してそれぞれを同時に処理（→ 集積回路の大規模化）
- **その当時の最高性能を目指した計算機を「スーパーコンピュータ」(super computer)と呼ぶ**
  - **「これを超えたらスーパーコンピュータ」という明確な境界はない。**
- 計算機高速化の方法は大きく分けて2通り
  - パイプライン型の処理(→ベクトル計算機)
    問題を直列に分解して流れ作業で高速化
    （例）ベルトコンベア、エスカレータ
  - 並列処理(マルチプロセッシング)：
    問題を並列に分解して計算して最後にまとめる
    （例）複数台並んだエレベータ



# 並行処理と並列処理

- 並行処理（concurrent processing）
  - 1個のプロセッサで、複数のプログラムを切り替えながら
    見かけ上同時に処理すること
    （例）１人の教員が複数の講義科目を同じ学期に担当する。

- 並列処理（multi-processing）
  - 複数のプロセッサで、1個のプログラムを
    手分けして同時に処理すること。
  - （例）1つの講義科目を3クラスに分け、
    複数の教員で分担する。



# CISC と RISC

- メインフレーム技術の主流は、ハードウェア構成の工夫と、それを活かす機械語命令の高機能化
  - 基本は accumulator と主記憶データとの様々な算術演算
  - １命令は1～3ワードの可変長。多様なアドレス方式
  - **CPUの電子回路が複雑化し、遅延が長くなりクロック高速化の障害に**

# CISC と RISC

- メインフレーム技術の主流は、ハードウェア構成の工夫と、それを活かす機械語命令の高機能化
  - 基本は accumulator と主記憶データとの様々な算術演算
  - １命令は1～3ワードの可変長。多様なアドレス方式
  - **CPUの電子回路が複雑化し、遅延が長くなりクロック高速化の障害に**
- メインフレームからマイクロプロセッサの時代へ（1990年代～）
  - 今までと逆に機械語命令を単純化して、電子回路を簡単にすれば、クロック周波数が上がって、全体として高速化できるのでは？
    **→ RISC（Reduced Instruction Set Computer）の提案**
    RISCの反対語は CISC (Complex Instruction Set Computer)
  - RISC: １命令は1ワード固定長。CPU内に高速なレジスタを多数配置。主記憶アクセスはロードとストアのみ。算術演算はレジスタ内で完結。
  - 同じ計算をするのに命令数が増えるが、コンパイラ技術でカバー。
  - 中小型機や組み込み用のCPUに採用され普及した。
  - 従来のCISC型もRISC的な考え方を採用して進化を続け、最近ではRISCとCISCの明確な境目はなくなっている。

# 今回のまとめ

機械語命令と内部動作(3)

- サブルーチンコール、算術式とスタック

アーキテクチャの基本知識(2)
(メインフレームの発展)

- マイクロプログラム
  - エミュレーション、ファームウェア
- メインフレーム互換機
- 仮想メモリ、仮想マシン
- ベクトル計算機、並列計算機
  - スーパーコンピュータ
  - パイプライン処理とマルチプロセッシング、並行と並列
- CISC と RISC