計算機科学入門

Antoine Trouvé
アントワン　トルヴェ
九州大学大学院システム情報科学研究院・情報知能工学部門（助教）
2014 / 07 / 11

自己紹介

名	姓
Antoine アントワン	Trouvé トルヴェ

2006年
フランスのボルドー大学で修士取得
（「ENSEIRB」グランゼコール）

2011年
九州大学で博士取得

2007年～2014年
九州先端科学技術研究所（研究員）

現在
九州大学工学部（助教）
＠伊都キャンパス

組込みシステム（スマホ　等）

プロセッサー・アーキテクチャー

コンパイラー

研究分野

クラウド

膨大データ統計解析

スパコンピューター

コード最適化手法

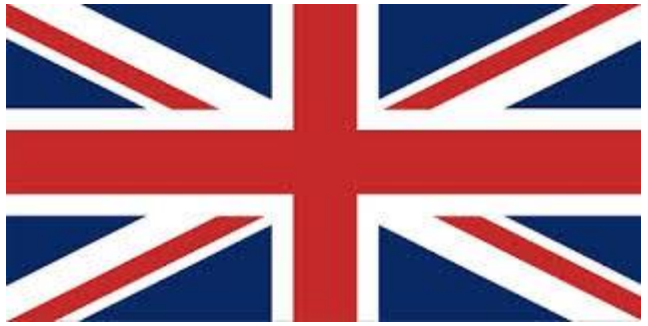
機械学習

再構成可能プロセッサー

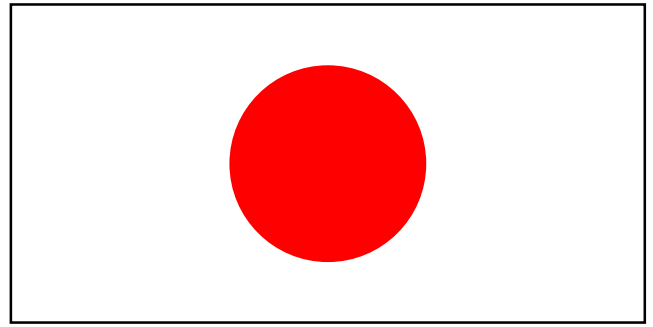
定量データ可視化

本日の概要

- スパコン入門
- プロセッサ入門
- プログラミング入門
- コンパイラと入門
- 最適化入門
- 並列プログラミング入門
- 切り抜き遊び



英語



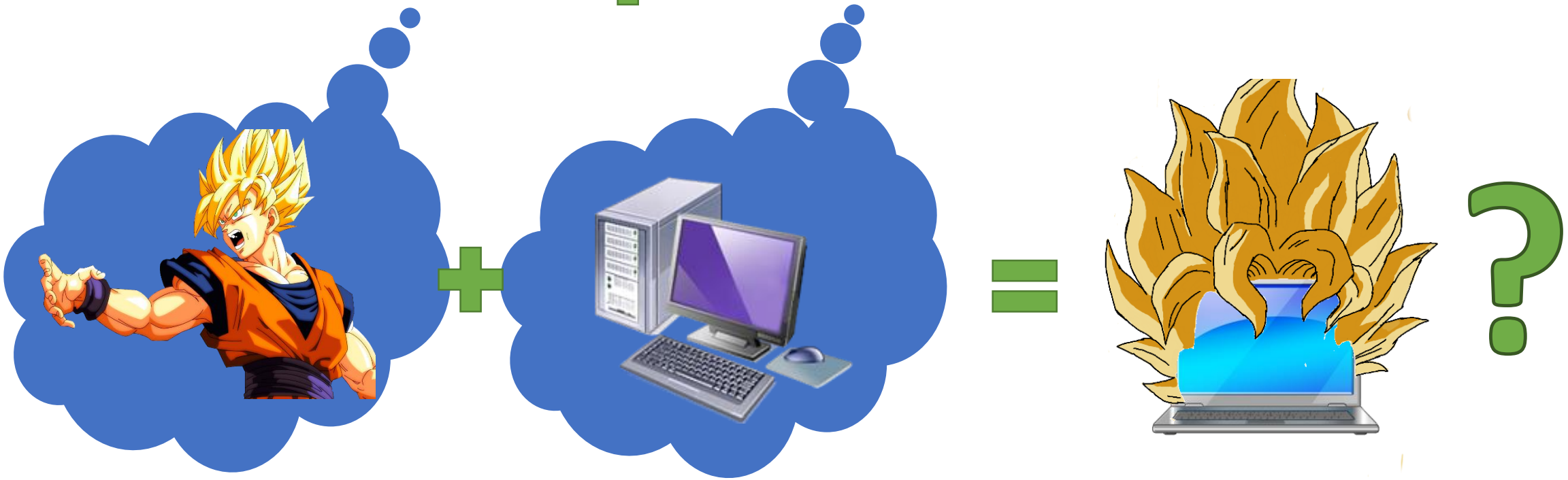
日本語

スパコンとは

What is a supercomputer ?

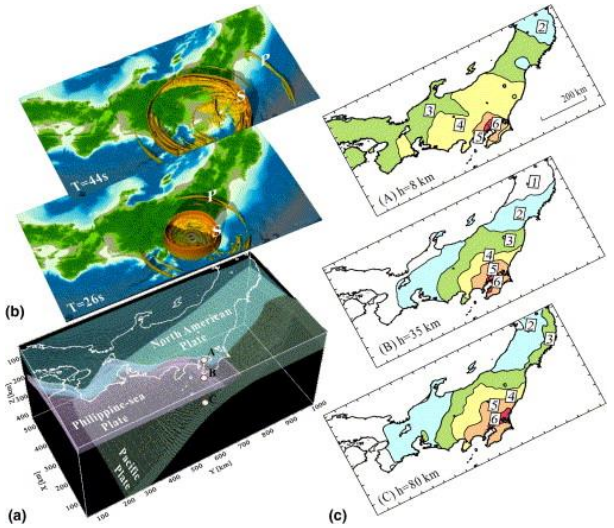
「スパコン」＝「スパー・コンピューター」の略

スパー ＋ コンピューター



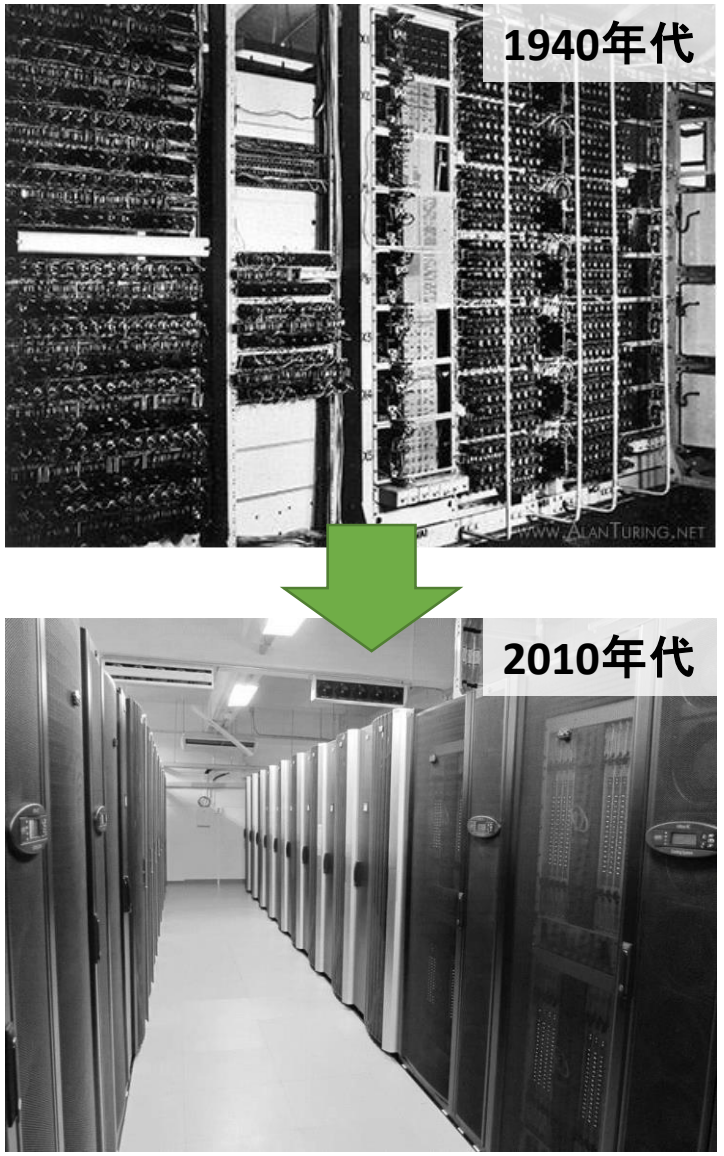
スパコンは科学者のためのツールです

- 第３科学を可能とする
  - 第１科学：理論
  - 第２科学： 実験
  - **第３科学：シミュレーション**
- シミュレーションとは？
  - 目的：自然をパソコン上に再現すること
  - 実験ができないときに、その代わりになるもの
- スパコンの応用事例
  - 分子作り（例：薬、ナノ分子）
  - 防災における津波シミュレーション
  - 宇宙シミュレーション
  - 原発設計
  - スマホ設計
- 体の中のビタミンのようにスパコンは日本の科学を支援してる
  - なくても生きていけますが、病気になる
  - あると自分をビックリするほどいろんなことを出来る
  - 直接影響を観測しにくい



スパコンは昔からあります

- スパコンは世界大戦代のパソコンに似ています
  - 写真（上）:ドイツのColossus（コロッサス）
  - 写真（下）: 東工大のTsubame（つばめ）
- しかし、初めてと言われているスパコンは英国のマンチェスター大学が１９６３年に設置
  - 名前 : Atlas（アトラス）
  - 整数計算性能 : 0.63MOPS（1マイクロセカンドで出来る整数計算）
  - 不動小数点計算性能 : 0.62MOPS（ 1マイクロセカンドで出来る不動小数点計算）



普通のパソコンと現代のスパコンの違いは？

- まず、外側から
  - 値段：スパコンは個人で買えない（数億円）
  - サイズ：スパコンは建物が必要（パソコンは机が必要）
  - ディスプレイ：スパコンはディスプレイがありません（ネットワークで接続し、利用）
  - インターフェース：スパコンはコマンドラインで操作
- あと、内部構造も違います
  - 沢山のパソコンが繋いでるというイメージ
  - 逐次計算が不得意で、並列計算は得意

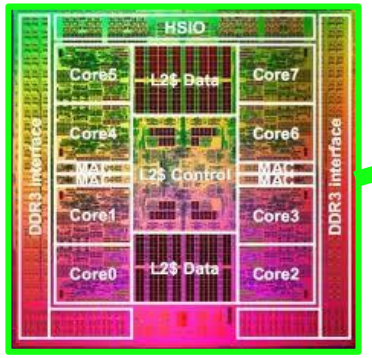
そうでもないスパコンもあり得ますが、なぜかこうなってしみました

後で改めて説明します

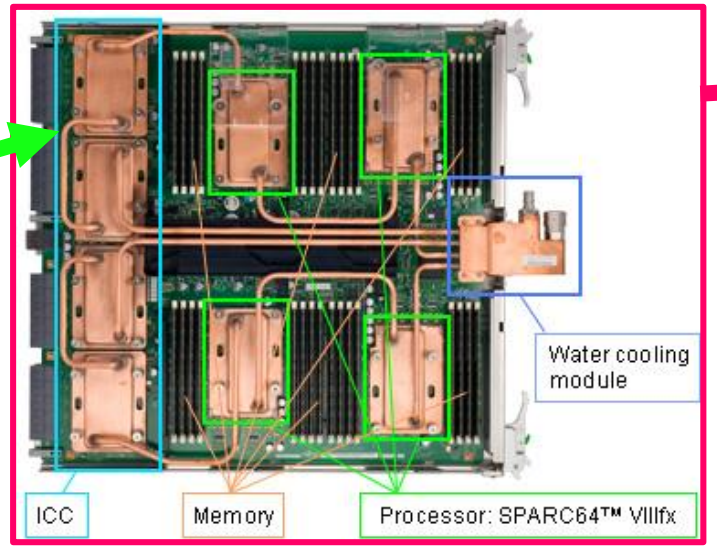
通常のパソコン

K Computer (Japan #1)

88 128 **Processors**



プロセッサ

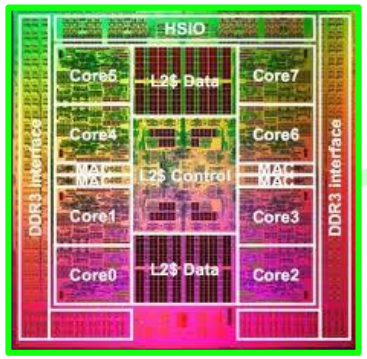


ボード毎に4プロセッサ

クロゼット毎に12ボード

クロゼット間の通信

864クロゼット

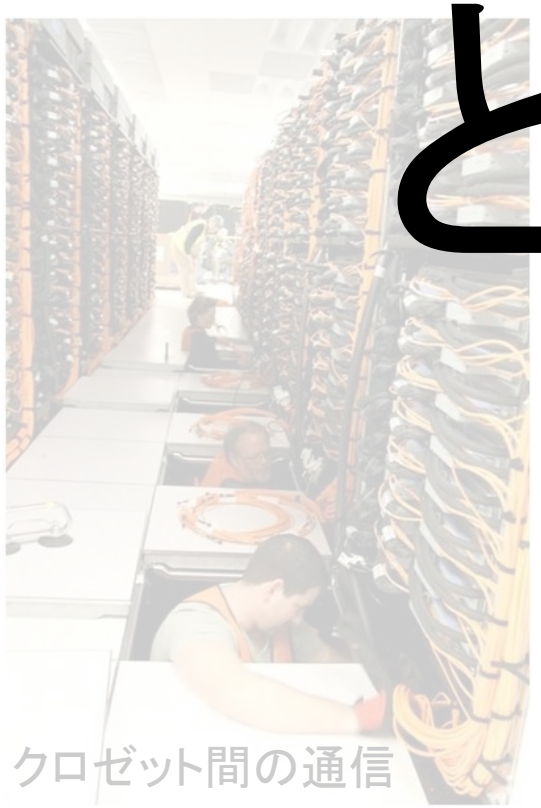


プロセッサ

ボード毎にプロセッサ

クロゼット毎に12ボード

とは？



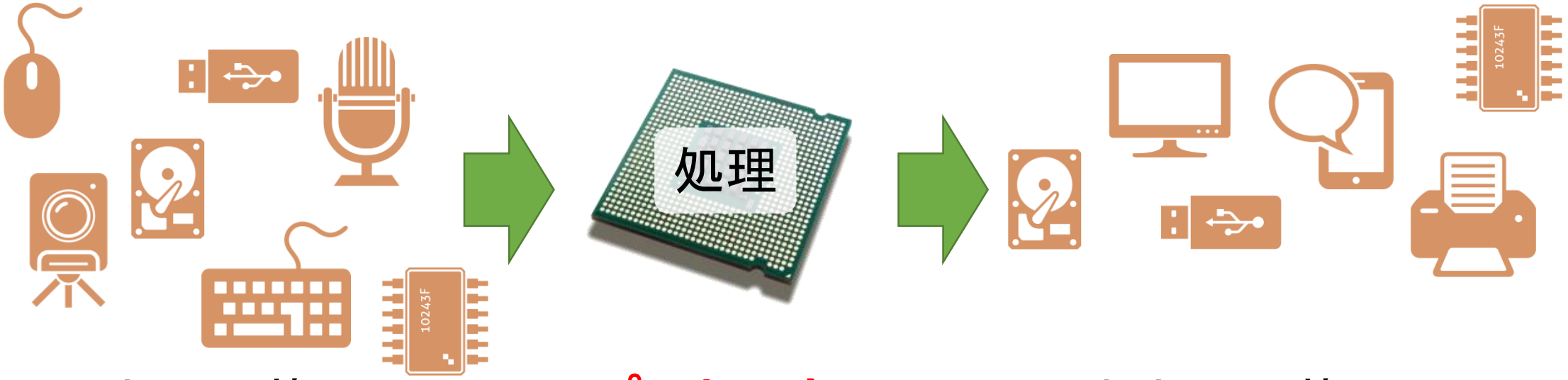
クロゼット間の通信

864クロゼット

プロセッサとは？

What is a processor

パソコンにおいて、プロセッサは処理する部品です



入力周辺装置　　プロセッサ　　出力周辺装置

諸々な身近なデバイスに搭載されている

+諸々な「スマートデバイス」　プリンタ、車　・・・

注意：ある程度複雑さがないとプロセッサよりも「マイクロコントロラ」という

プロセッサは脳みそではない

プロセッサはオートマタです：
- 新しい理論を作ることはできません（以前に決められた理論に従って動作する機械です）
- 製造時ハードは決まっています

≠

脳みそは
- 入力を多く収集し、理論展開を作ります。
- 一生内部構成は変わります：神経細胞も細胞間の接続も


頑固な人でも！

得意な事：定量解析（算数、記憶）

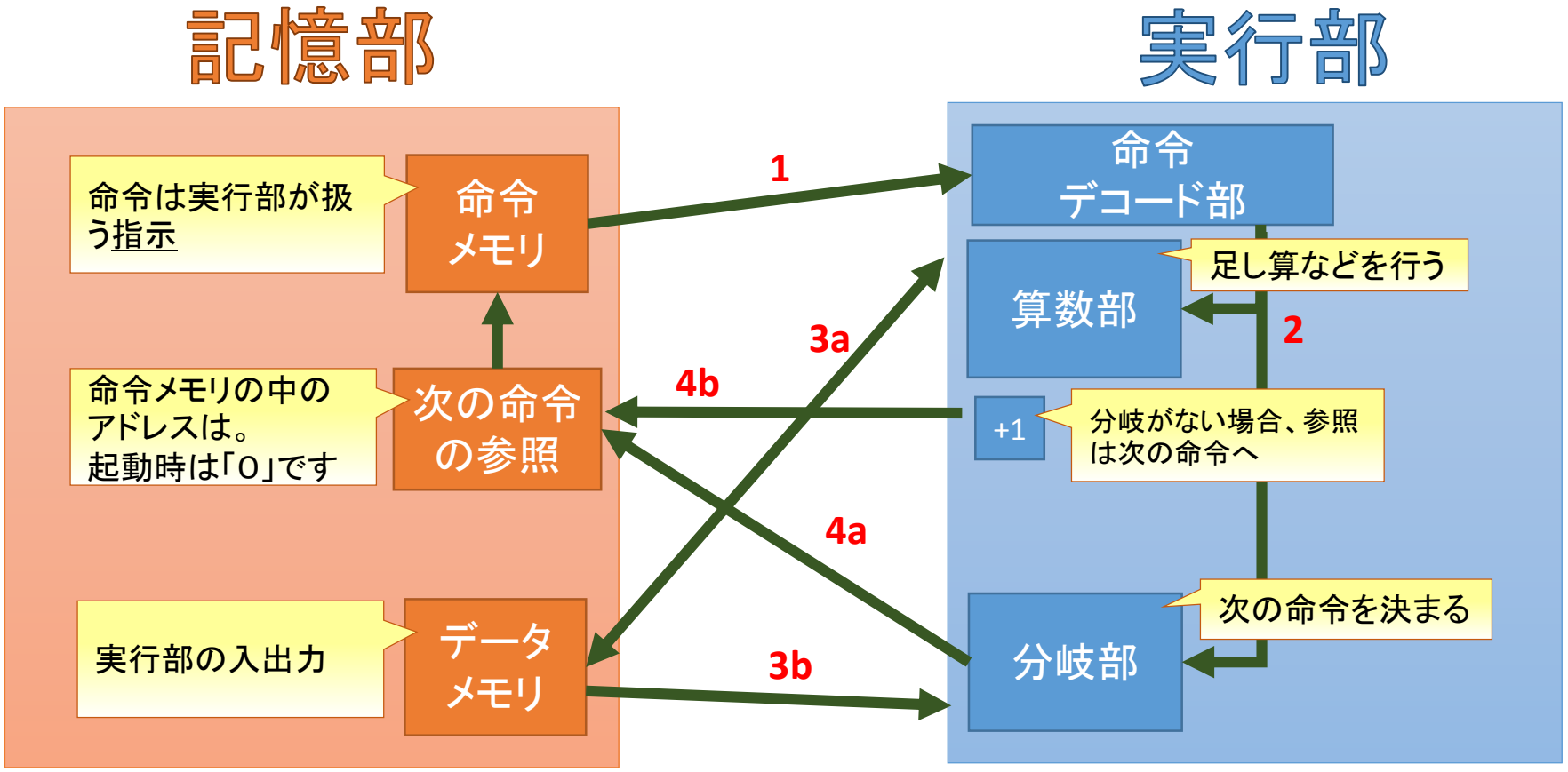
得意な事：質的解析（パターン認識）

しかし、プロセッサはどんなアルゴリズムをでも実行は可能です！

（1932年：チューリング氏が「チューリングマシン」を定義し、証明したこと）

プロセッサ上に脳みそを真似るためにの問題点：
- **アルゴリズム化：人間である研究者は脳みそを真似るアルゴリズムを考える必要がある**
- **実行効率の向上：現代の脳みそシミュレーションはすごく沢山の計算資源が必要です（脳みそよりも効率が100万回低い）**

プロセッサの内部構成（イメージ）



1. 命令読み出す
2. 命令をデコードし、敵する実行ユニットに回す
3a. 算数命令の場合：データを読み出し、計算し、結果を書き込む
3b. 分岐命令の場合：データを読み出す
4a. 分岐命令の場合：次の命令の参照を出力
4b. 算数命令の場合：次の命令は逐次順で決める（アドレスを+1）

命令、命令セット、プログラム

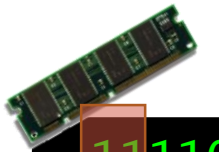
- 命令は実行部への指示
  - 算数：演算の種類と入出力の参照
  - 分岐：理論の種類
  - 設計時で決められます
- 命令セットはプロセッサが分かる言語
  - 単語：命令
  - 分布もあります
- プログラムは命令のシーケンス
  - プロセッサは1個ずつ実行する
  - 命令メモリで保存（バイナリで表現）

ちなみに、何でバイナリを使う？（2を低とした数値）
- 現代のピロセッサはバイナリ理論ゲートで作ってあるます
- バイナリを使ってる理由は設計しやすいから（1/0：on /off）

命令セットの定義

命令	10形式
上	00
下	01
左	10
右	11

命令ワードの幅：2列



命令メモリ

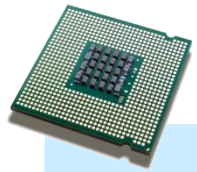
11110010010111

0 2 4 6 81012

次の命令の参照（単位：列）

プログラム：
右、右、上、左、下、下右

カエルの例



Processor (one core)

プロセッサの種類

命令セットや性能によってプロセッサは向き不向きがあります

汎用プロセッサ（別名：I/Oプロセッサ）：
- どんな計算でも同じような性能でできます
- 特にメモリは効率よく入出できます
- 例：主なインテル社やARMのプロセッサ

SIMDプロセッサ
- SIMD：Single Instruction Multiple Data
- 配列や行列計算は得意
- メモリの利用は諸々な制限あり

原因は性能よく配列を読み出せるために設計時の妥協

グラフィックスプロセッサ
- 3次元幾何学計算は得意
- その他の計算は遅い（特に分岐）

他にもあります：暗号化向け、デジタル信号処理向け　など

現代プロセッサは複数の種類に有する

命令セットの混合化：プロセッサは複数の命令セットの種類を導入
→　現代のインテルプロセッサは：汎用・SIMD・暗号

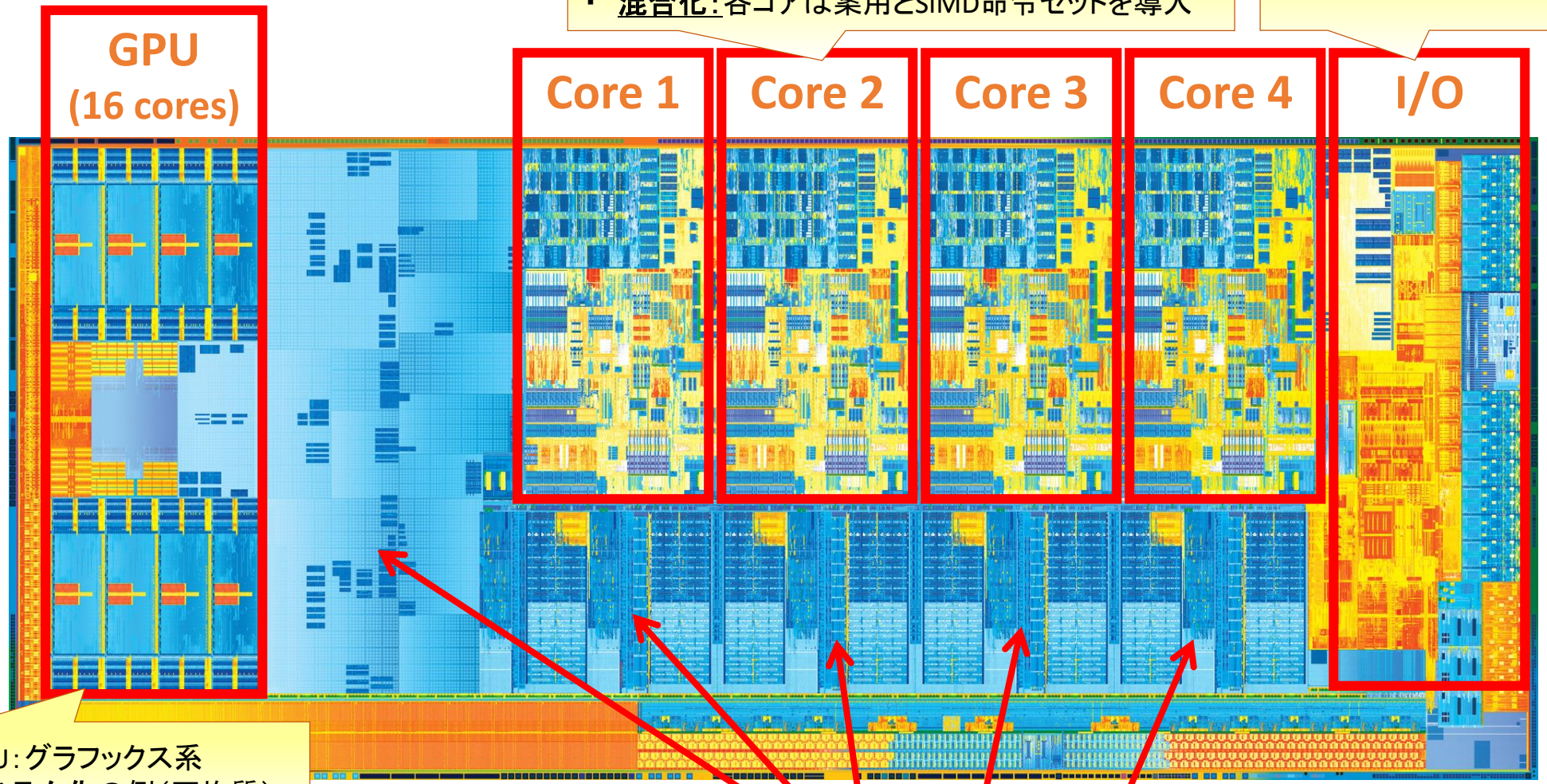
システム化：同じ基盤に複数なプロセッサを搭載
→　現代のインテルプロセッサは：グラフィックス系も搭載

実例:インテル社の「Ivy Bridge」

初期世代のプロセッサは1コアしかなかったので、「コア」というコンセプトがなかったが、現代は最近多数になっており、「コア」という言葉が出てきました

- コアはプロセッサの機能を持っている部分
- **システム化**:4コア:案用プロセッサを4個搭載
- **混合化**:各コアは案用とSIMD命令セットを導入

- 「I/O」は入出という意味
- メモリや周辺装置との接続



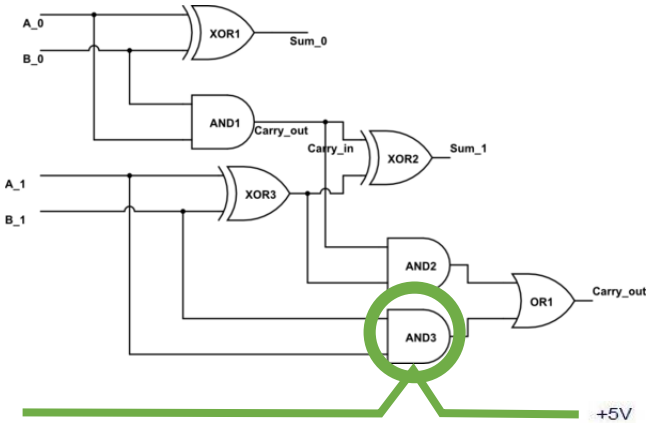
- GPU:グラフックス系
- システム化の例(不均質)

メモリ(キャッシュ)

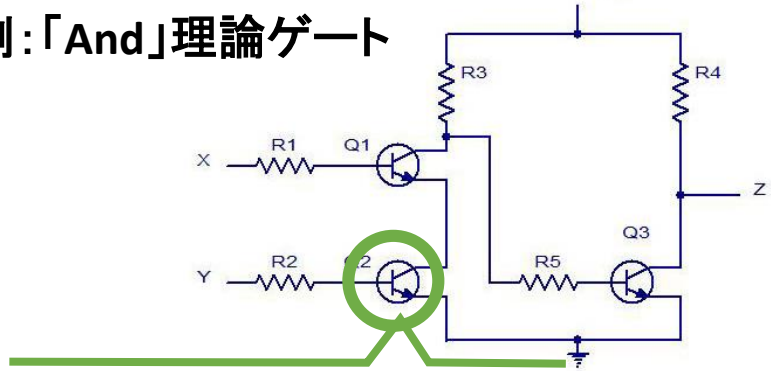
プロセッサの設計

- 物理的にプロセッサは理論ゲートで作られています
  - 理論ゲートはブール演算をできる小さな回路
  - ボール演算：ADD、OR、XOR　等
  - バイナリ理論ゲートを組み合わせることによって、どんな計算でもできます（どの低でも）
- 理論ゲートは主にトランジスタで作られています
  - トランジスタ数でプロセッサの複雑さや消費電力を分かります（大体）
  - 主にシリコンで作ってあります（半導体）

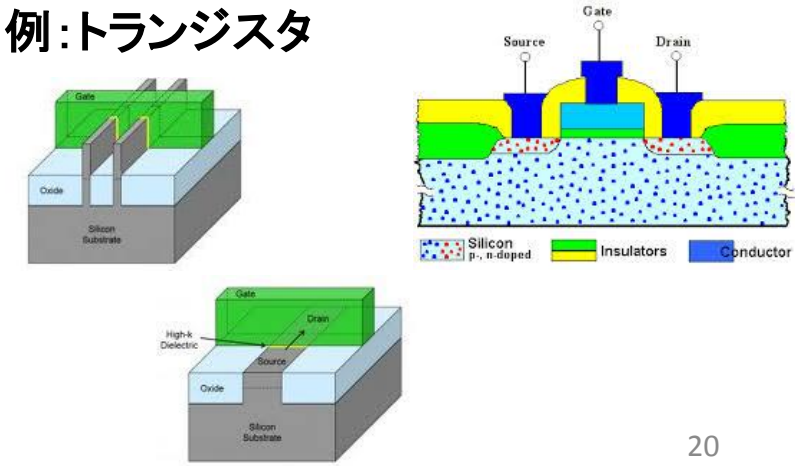
例：2 bit 足し算



例：「And」理論ゲート



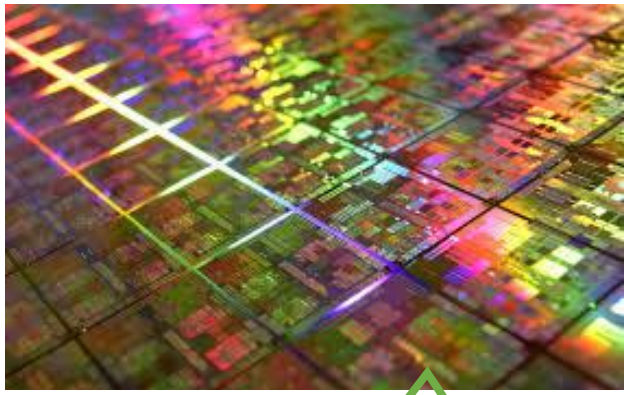
例：トランジスタ

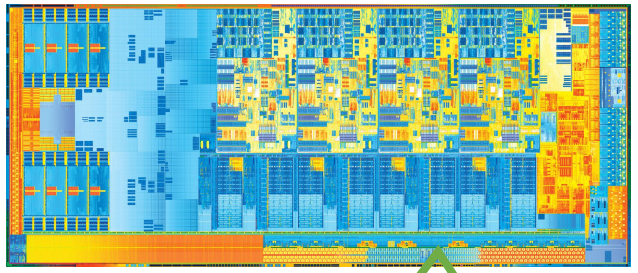


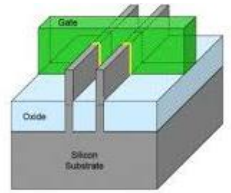
プロセッサの製造

- 製造小型化について
  - トランジスタを小さく作るとプロセッサの複雑さを向上できます
  - が、小型化によってトランジスタの特徴が変わります（発熱、消費電力、信頼性　など）
  - 各小型化ステップ毎にトランジスタの構成および材料を変える必要があります
  - 最新製造技術の製造密度は14ナノです（Si原子は5ナノ）
- ピロセッサ製造所について
  - 現代プロセッサの製造ができる会社は本の一握り
    - 各小型化ステップ毎にさらに減ります！
  - 製造所への投資はリスク（＝コスト）が高い
    - 例：インテルの14ナノ製造所：**約5千億円**（50億ドル）
  - 最先端製造所を持っている会社：インテル、TSMC、Global Foundry（旧：AMD）、ST Microelectronics
    - TSMCはスマホ向けの部本をほぼ製造しています（客：アップル、サムスン　など）

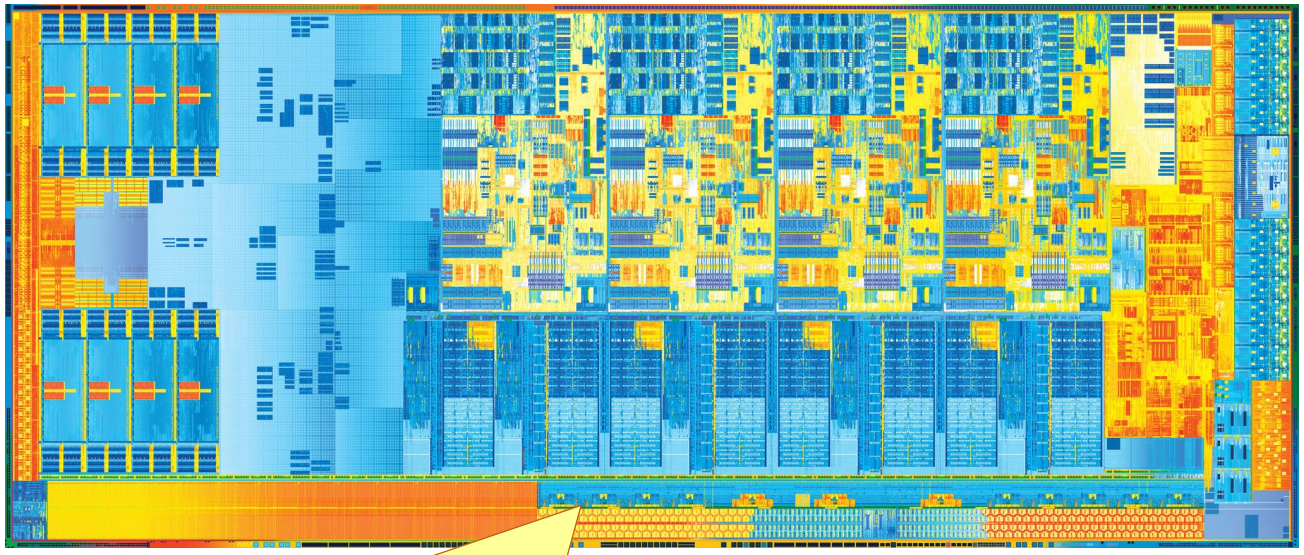
・　物理的に5ナノよりも小さくできない！
・　近いうちに新しい技術が必要になる！

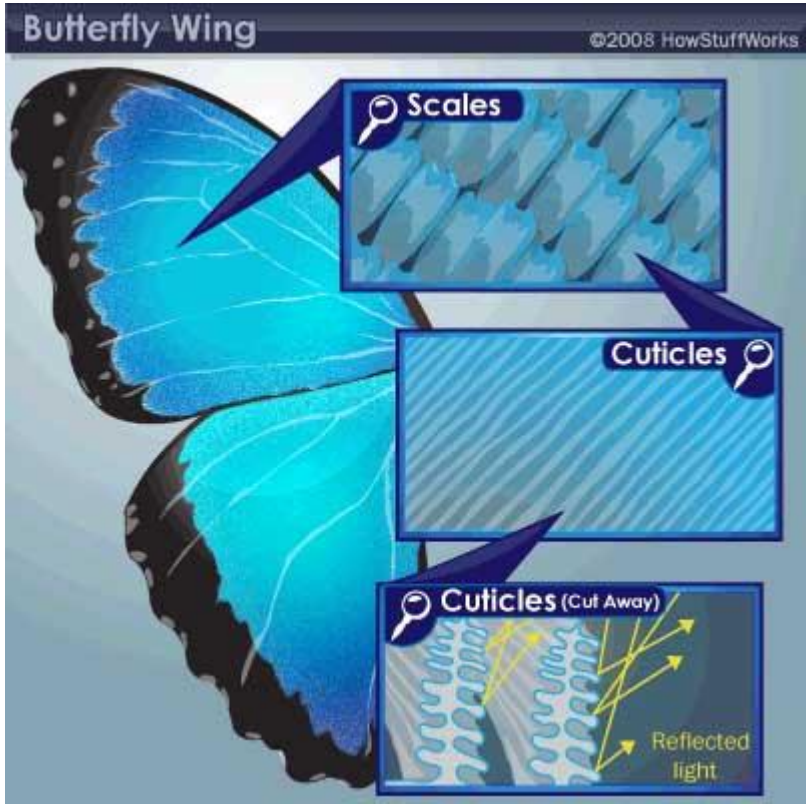






（ちなみに）プロセッサーの写真の色について



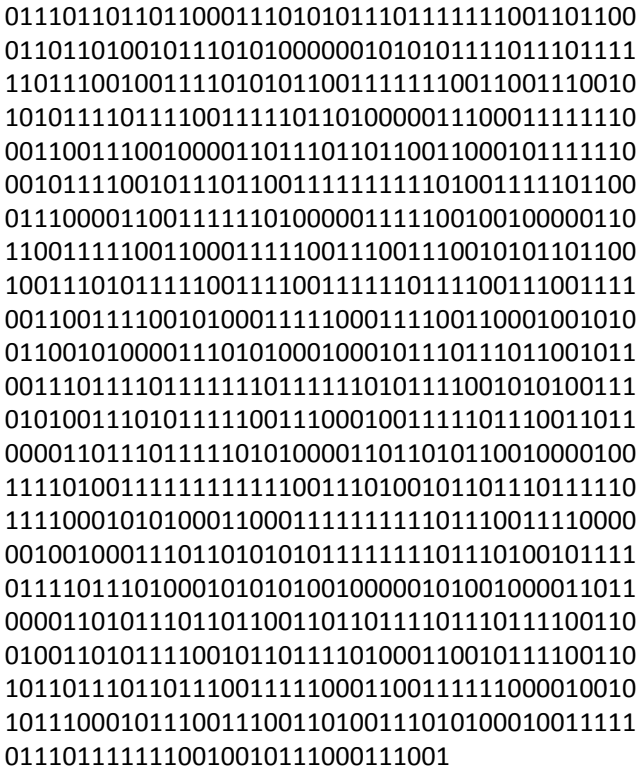


ナノスケールな凸凹により、光のは回折し、
色んな色が映っています！
蝶の翼と同じ現象！

プログラミング言語について

About programming languages

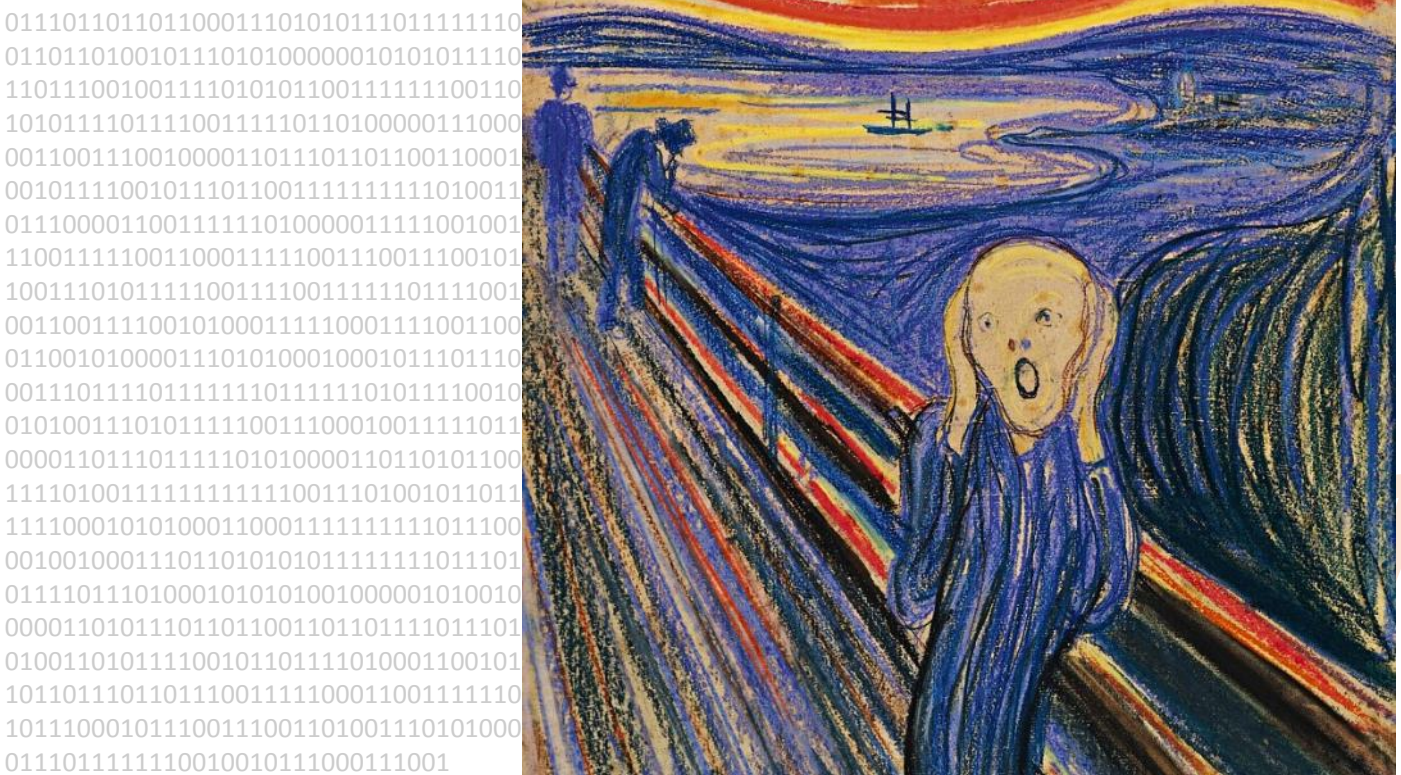
問題：プロセッサは「1」と「0」しか通じません



このプログラムは何を表していますか？

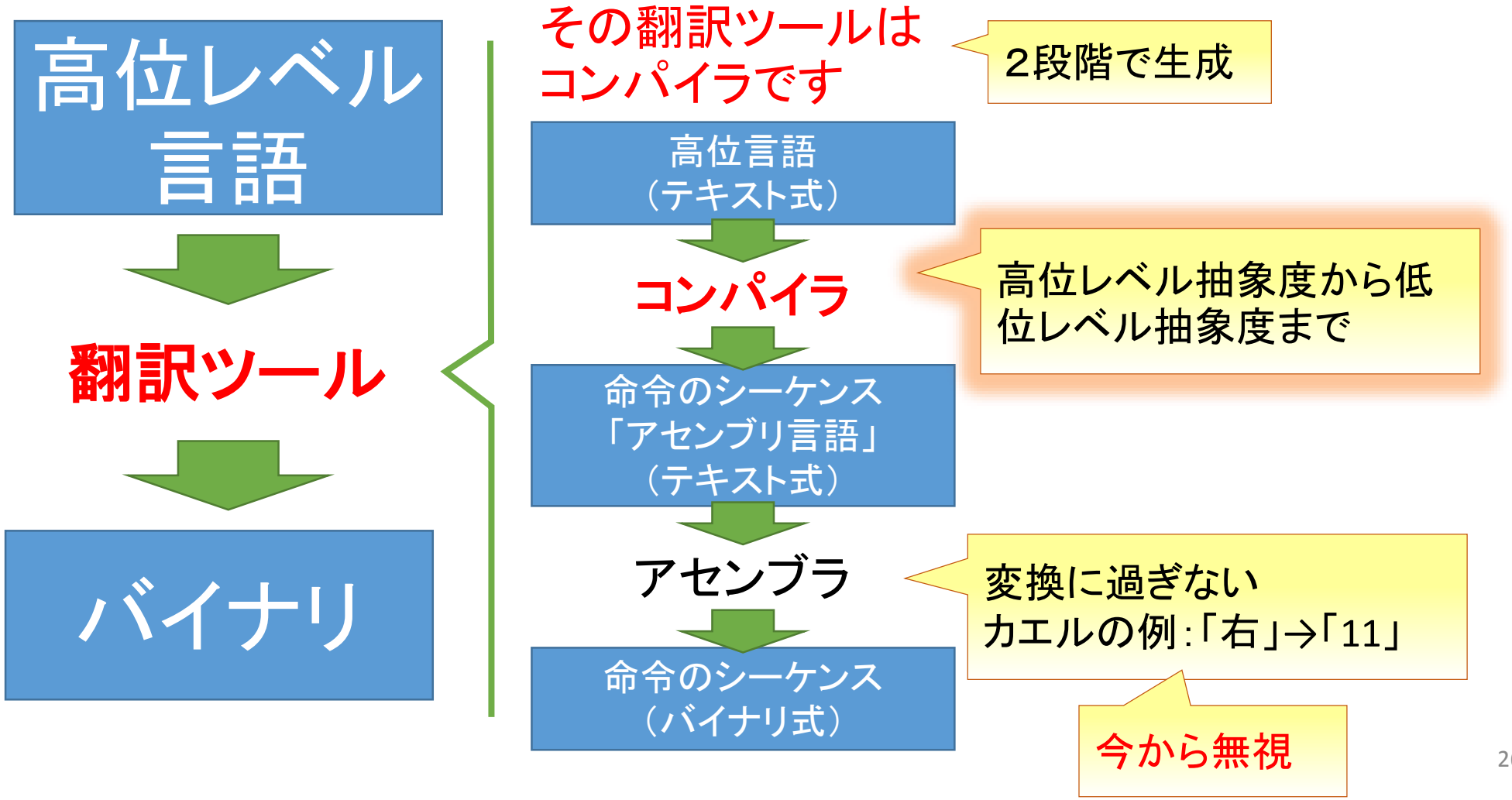
誰か？

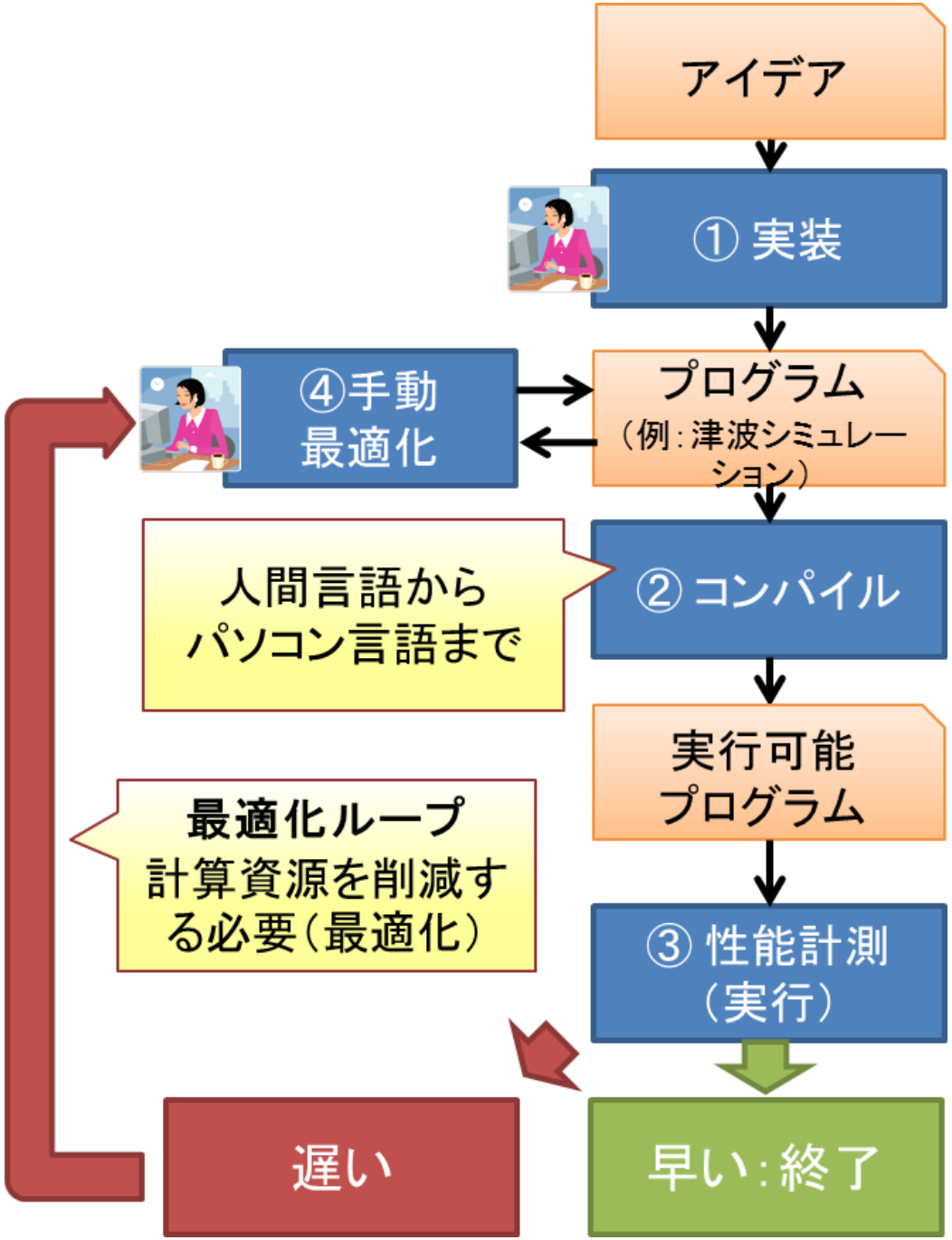
問題：プロセッサはデータを通じません



Der Schrei（さけび）より、**エドヴァルド・ムンク（1863-1944）**

解決策:プログラムは高位レベル言語で書いて、通訳ツールを利用





開発フロー

- アイデアから始まります（アルゴリズム設計を含む）
- ループです：対象を達成するまで磨く（性能、機能　など）

今から「高位レベル言語」は「プログラミング言語」と言います

高位レベル言語を紹介

プログラミング言語(1/2)
第1・2世代

1940's: machine code (first generation of prog. Lang.)

- Programming using binary code directly
- Example of the frog: 11110010010111

But binary has low productivity

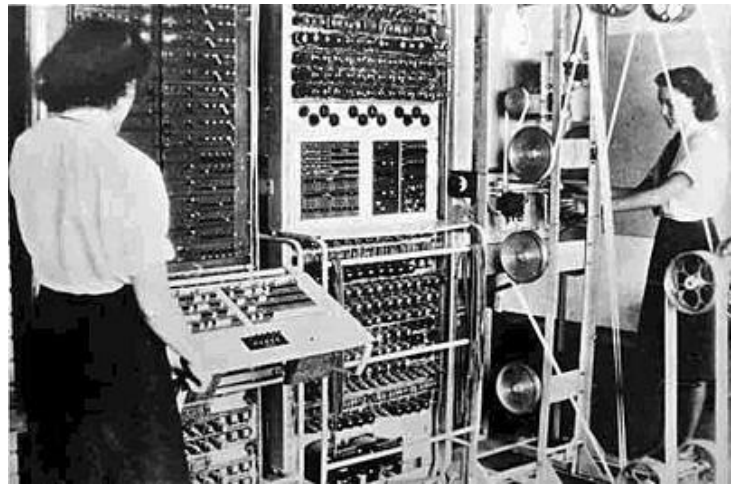
- Too complex for human being: error prone
- Very hard to write large programs

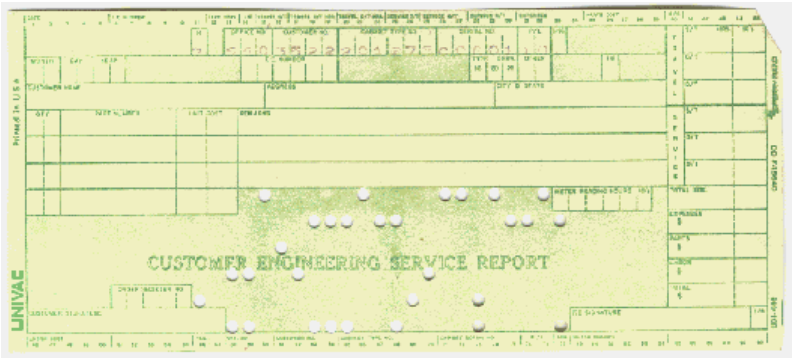
1950's: assembly language (second generation of prog. lang.)

- Instead of writing "1" and "0", people write "add" or "sub"
- Example of the frog: 右;右;上;左;下;下;右

Productivity is better than binary, but it could be better

- Quick fix: people use "macro assembly instructions": instead of writing 右;右 we can write 2回右
- No real "high level language" yet





プログラミング言語の歴史 (2/2)
現代的な言語へ

End of 1950: Apparition of first programming languages (third generation of prog. lang.)

- Fortran: scientific calculations
- Cobol: data processing
- Lisp: functional language

1969-1973: C language

- Created in Bell laboratories (USA) to implement the first UNIX OS
- The most used language right now
- Meant for system programming, but used for everything now (unfortunately)

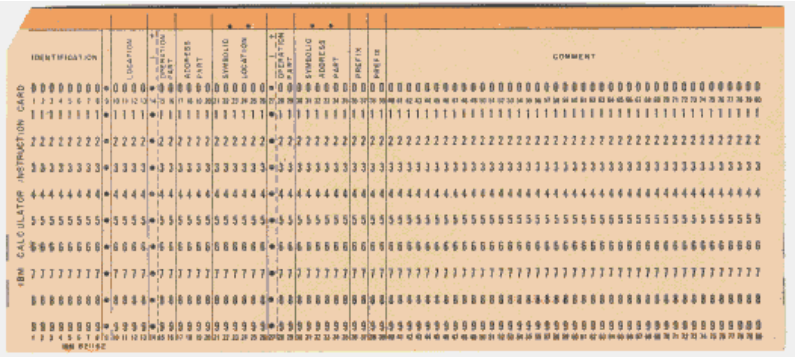
1983: C++ language (object-oriented language)

- Extension of C to support object-oriented programming
- Widely popular now

1996: Java (virtual machines and just-in-time compilation)

- Resembles C++, but abstracts memory allocations
- Originality: the Java compiler compiles in bytecode, not machine code

～70年代まで：パンチカード
穴をあけて、指示を表す



現在：テキストファイル：文字で

```
ggplot(cgs.size, aes(x=nb.wids, y=nb.cgs)) + geom_point() + geom_line(data=data.frame(nb.wids=cgs.size[nb.wids<200,]$nb.wid

##############################################################################
### The CGS hit, that is, the ratio of test TC which have a good scenario in the training CGS list
##############################################################################

### Percentage of test TC which have at least a good scenario in the train CGS
nb.shots <- 10
percent.step <- 0.1
good.thresholds <- c(0.05,0.1, 0.25)
train.sizes.folds <- 2:10
# cgs.ratios <- data.table(train.size=rep(0,length(train.sizes.percent)*nb.shots*length(good.thresholds)), ratio=0, good.th
last.index <- 0
for(good.threshold in good.thresholds) {
    for(cgs.fold in train.sizes.folds) {
        train.size.tmp.percent <- 1-1/cgs.fold
        train.size.tmp <- ceiling(length(all.wids) * train.size.tmp.percent)
        for(shot in 1:nb.shots) {
            train.wids.tmp.ids <- sample(1:length(all.wids), train.size.tmp)
            train.wids.tmp  <- all.wids[train.wids.tmp.ids ]
            test.wids.tmp   <- all.wids[-train.wids.tmp.ids]

            train.data.tmp  <- all.data[train.wids.tmp]
            test.data.tmp   <- all.data[test.wids.tmp ]
            train.data.good.tmp <- train.data.tmp[d0>good.threshold,]
            test.data.good.tmp  <- test.data.tmp[d0>good.threshold,]

            train.cgs.tmp <- cgs(train.data.good.tmp)

            ratio <- length(unique(test.data.good.tmp$wid[test.data.good.tmp$sc_id %in% train.cgs.tmp])) / length(test.wids

            last.index <- last.index+1
            index <- last.index

            print(index)

            cgs.ratios[index]$train.size <- train.size.tmp.percent
            cgs.ratios[index]$ratio <- ratio
            cgs.ratios[index]$good.threshold <- good.threshold
            cgs.ratios[index]$fold <- cgs.fold
        }
    }
}

cgs.ratios$fold <- 1/(1-cgs.ratios$train.size)

ggplot(cgs.ratios, aes(x=fold, y=ratio, color=factor(good.threshold))) + geom_point() + ylim(c(0,1)) + scale_x_continuous(l

ggplot(cgs.ratios, aes(x=fold, y=ratio, color=factor(good.threshold))) + scale_x_continuous(limits=c(2,10), breaks=2:10) +
```

プログラミング言語とアセンブリ：実例

「C」という言語

```
for(i=0; i<max; i++){
  t = a;
  a = b;
  b = t + a;
  i++;
}
```

命令セット：x86-64

```
LBB0_2:
  movl %eax, %esi
  movl %edi, %eax
  addl %esi, %eax
  addl $2, %edx
  cmpl %ecx, %edx
  movl %esi, %edi
  jl LBB0_2
```

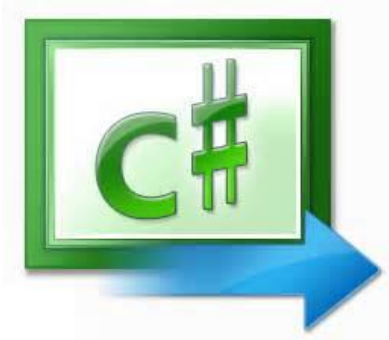
現代のインテルプロセッサ

命令セット：ARMv7

```
LBB0_1:
  mov r1, r0
  add r2, r2, #2
  add r0, r1, r3
  mov r3, r1
  cmp r2, r12
  blt LBB0_1
```

主にスマホやタブレットに搭載

プログラミング言語が沢山あります！

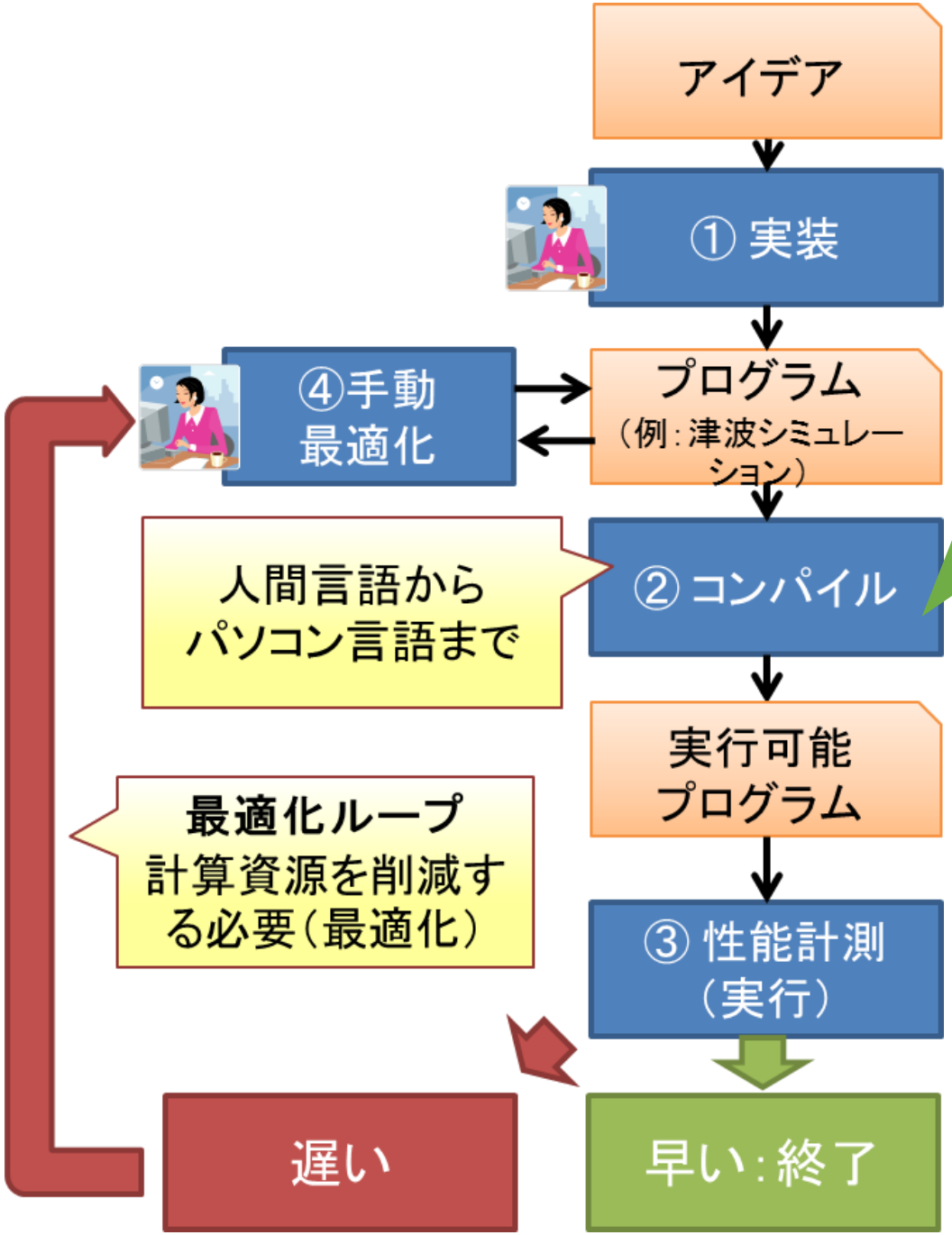


なぜこんなに沢山？

- 世代：昔い言語がそのまま使われていて、現代的な言語も出てきます
  - 例：「COBOL」は昔いの銀行のシステムの実装のために使われていて、信頼性上の心配により、書き直したくない
- 応用対象：元号によって向き不向きがあります
  - 例１：C言語はシステム・プログラムを実相しやすい（ハードに近い）
  - 例２：マトラブは数学アルゴリズムを設計しやすい
  - 例３：オブジェクト指向言語は大きいプロヘクトに向いています
- 対象ハードウェア
  - 例：通常の「C」はGPUで利用できませんので（想定しているメモリ・モデルは異なる）、CUDAが存在している
- 開発者の好み
  - 例：「Phython」と「Ruby」は機能が近いが、両方が生き残っています

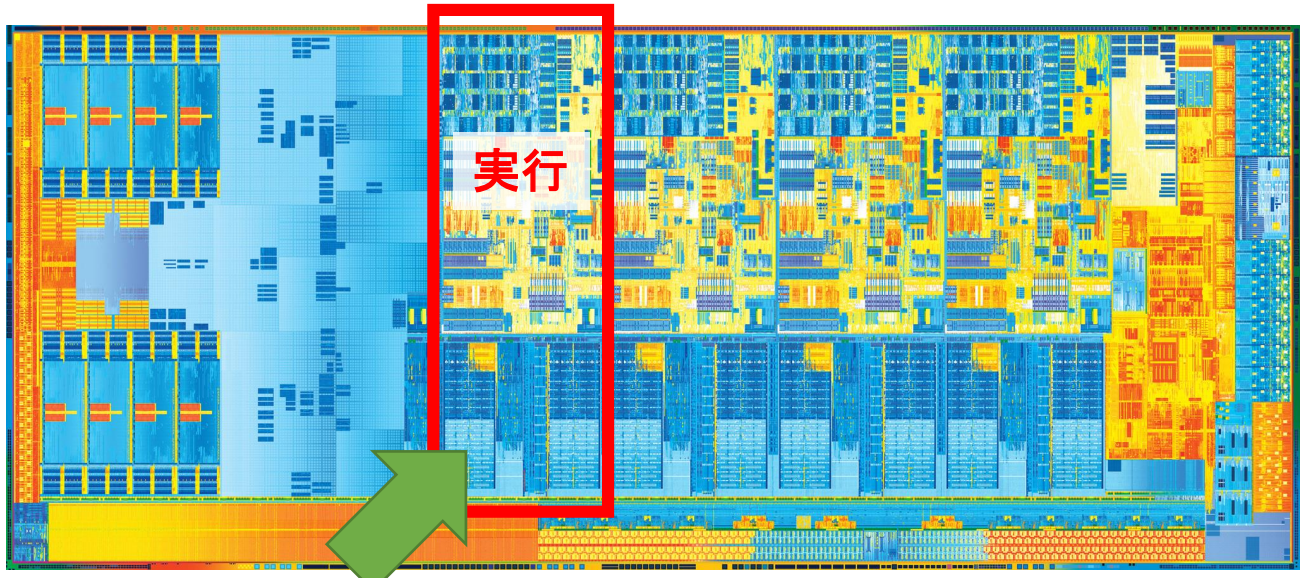
最適化コンパイラについて

About optimizing compilers


アイデア
① 実装
プログラム
（例：津波シミュレーション）
④手動
最適化
② コンパイル
人間言語から
パソコン言語まで
実行可能
プログラム
最適化ループ
計算資源を削減する必要（最適化）
③ 性能計測
（実行）
遅い
早い：終了

現在地

プログラムの旅：開発者の脳からプロセッサの命令メモリまで



プログラム言語

コンパイラ

11110010
010111

アセンブラ

```
right
right
top
left
bottom
bottom
right
```

アセンブリ言語

- 原則、プログラムは1コア上で実行します
- マルチコア実行も出来ますが、後で紹介します

問題：コンパイラは通訳ツールですが、

質のいい通訳が必要！

Émile Zola est un homme éminemment sociable, multipliant les amitiés de tous ordres et tous milieux, tout en refusant les mondanités. Passionné par ses semblables, il privilégie cependant les amitiés artistiques et littéraires, et fuit les politiques. Dès 1868 et grâce à ses travaux journalistiques, il se lie avec les frères Goncourt, Edmond et Jules. Puis en 1871, il rencontre Gustave Flaubert. Celui-ci, à l'occasion de réunions dominicales, l'introduit auprès d'Alphonse Daudet et Ivan Tourgueniev. Toute sa vie, Zola gardera la nostalgie de ce ≪ petit groupe ≫ dans lequel de ≪ trois à six, on entreprenait un galop à travers tous les sujets, où la littérature revenait chaque fois, le livre ou la pièce du moment, les questions générales, les théories les plus risquées28. ≫.

Google

世俗を否定しながら、エミール·ゾラは、すべての種類の友情とすべての背景を掛ける極めて社交的人間である。仲間に魅了されたが、彼は芸術的、文学的な友情を好み、およびポリシーを逃げる。1868と彼のジャーナリズムの仕事のおかげで、彼はゴンクール、エドモンドとジュール·兄弟を助けた。その後1871年に、彼はギュスターヴ·フローベールに会った。後者は、日曜日の集会の際は、彼にアルフォンス·ドーデやツルゲーネフを導入しました。彼の人生を通して、ゾラは「三から六、文学はたびに返されたすべての被験者、全体のギャロップが一瞬の本や作品、一般的な問題我々は約束している「小グループ」の懐かしさを保つ理論最もrisquées28。」。

フランス語、
エミール・ゾラ（1802-1985）
ウィキペディアによる伝記

日本語版：グーグルトランスレータより

「質のいいプログラム」とは？

別な言い方：計算資源を
無駄にしない

性能および
効率がいい

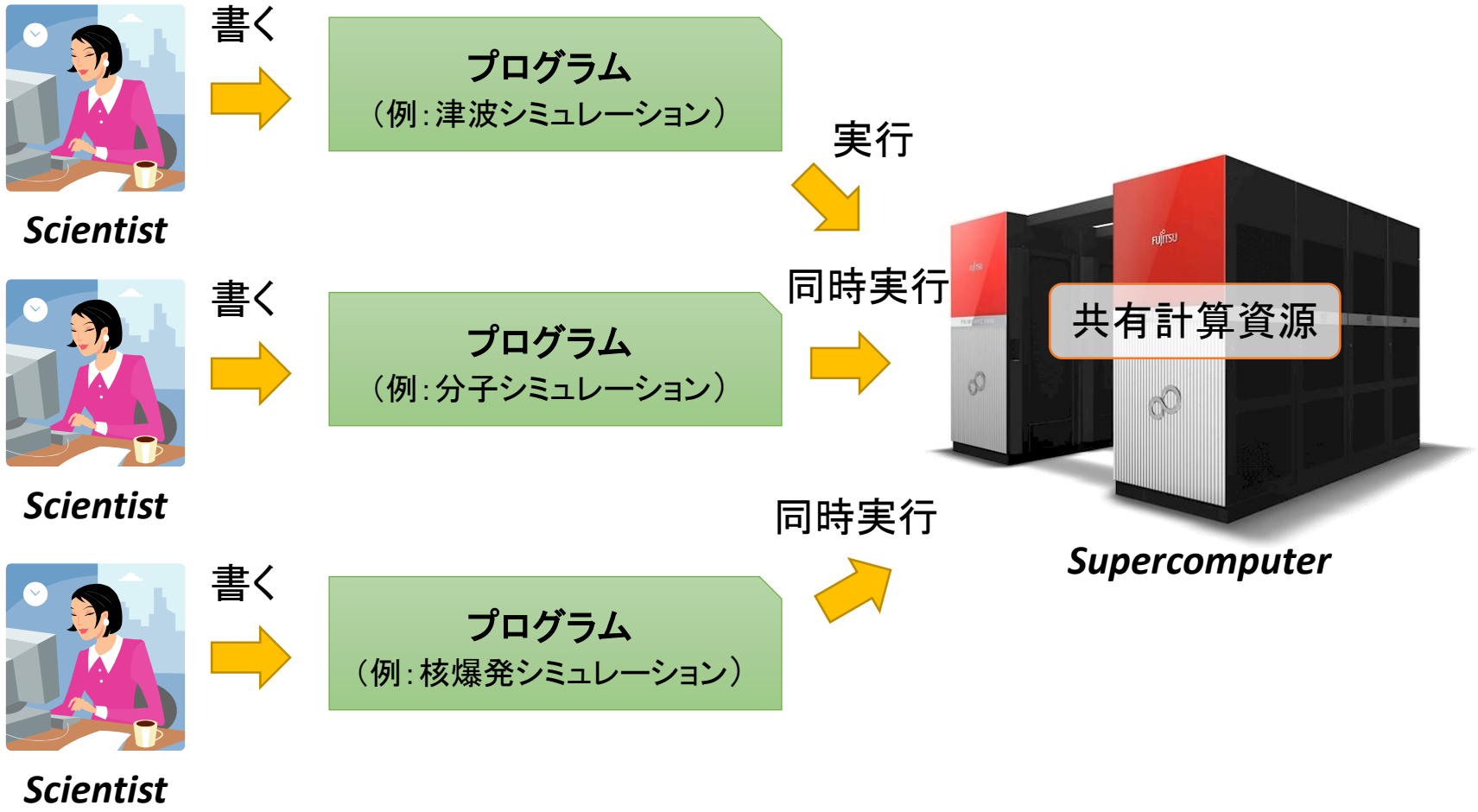
＋

出力は
一致します

無駄な計算をしない

最適な表現を使う

「計算資源」について
スパコンは共有計算資源！



「計算資源」について

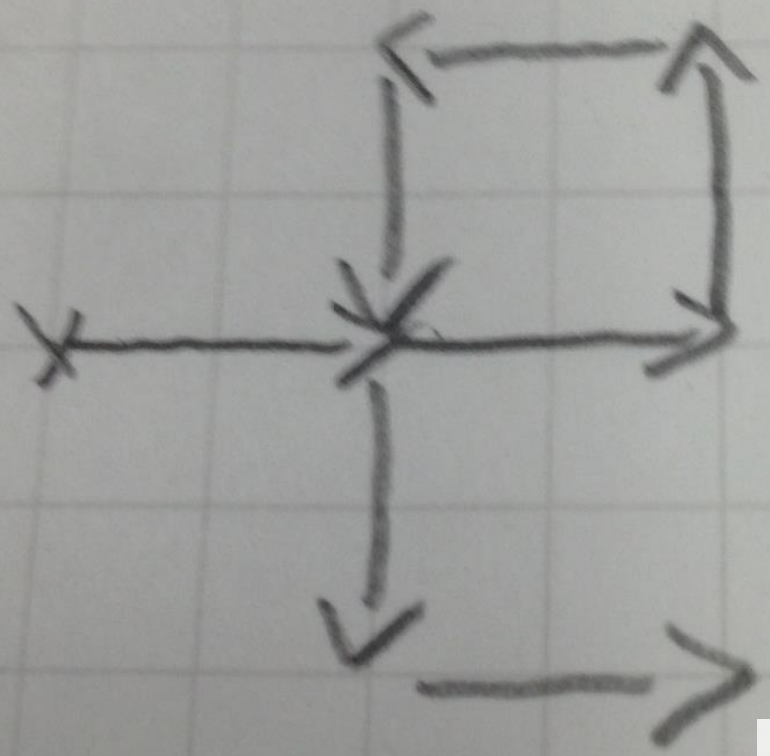
- 計算資源とは
  - 1人の科学者のプログラムが使っている計算能力
- その資源を無駄に使うと
  - 同じスパコンでできる科学が減る
  - プログラムの実行時間が増える
  - 計算するための消費電も増える
  - 駐車スペースとの比較：無駄に使うと駐車できる車が減る（と他のドライバがイライラする）
- その資源はどうやって決まるのか？
  - スパコンの構成：変えられない
  - 科学者が書いたプログラムの質：最適化コンパイラを使って変えられます！
- 問題点
  - 科学者はITの専門家ではないので、質の悪いものを書いてしまう
  - 結果：現在のスパコン利用効率は低い！無駄に使ってます！
- 解決策：コンパイラは自動的に最適化を行う

最適化例1：
無駄な計算を削除する（カエルの例）

出発点から目的地まで行くのに、
無駄な動きがありますか？

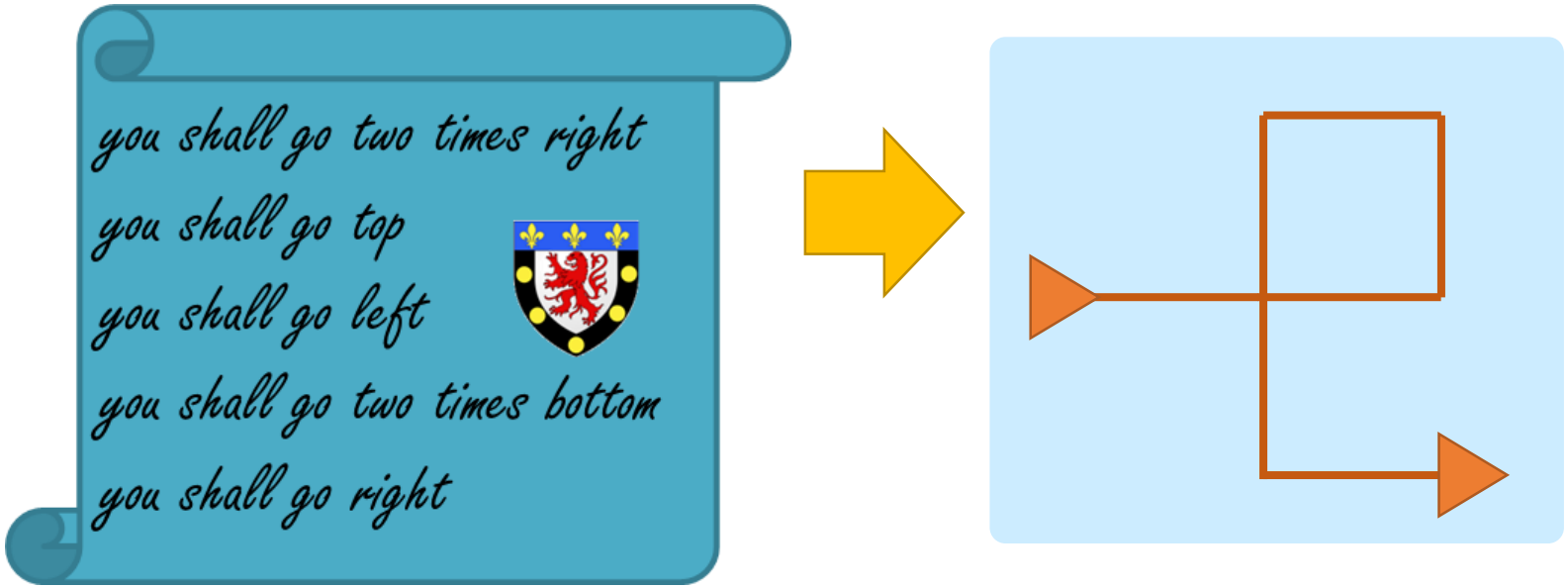
誰か？

It hard to answer from the text of the program:
people tend to use **graphical representation**



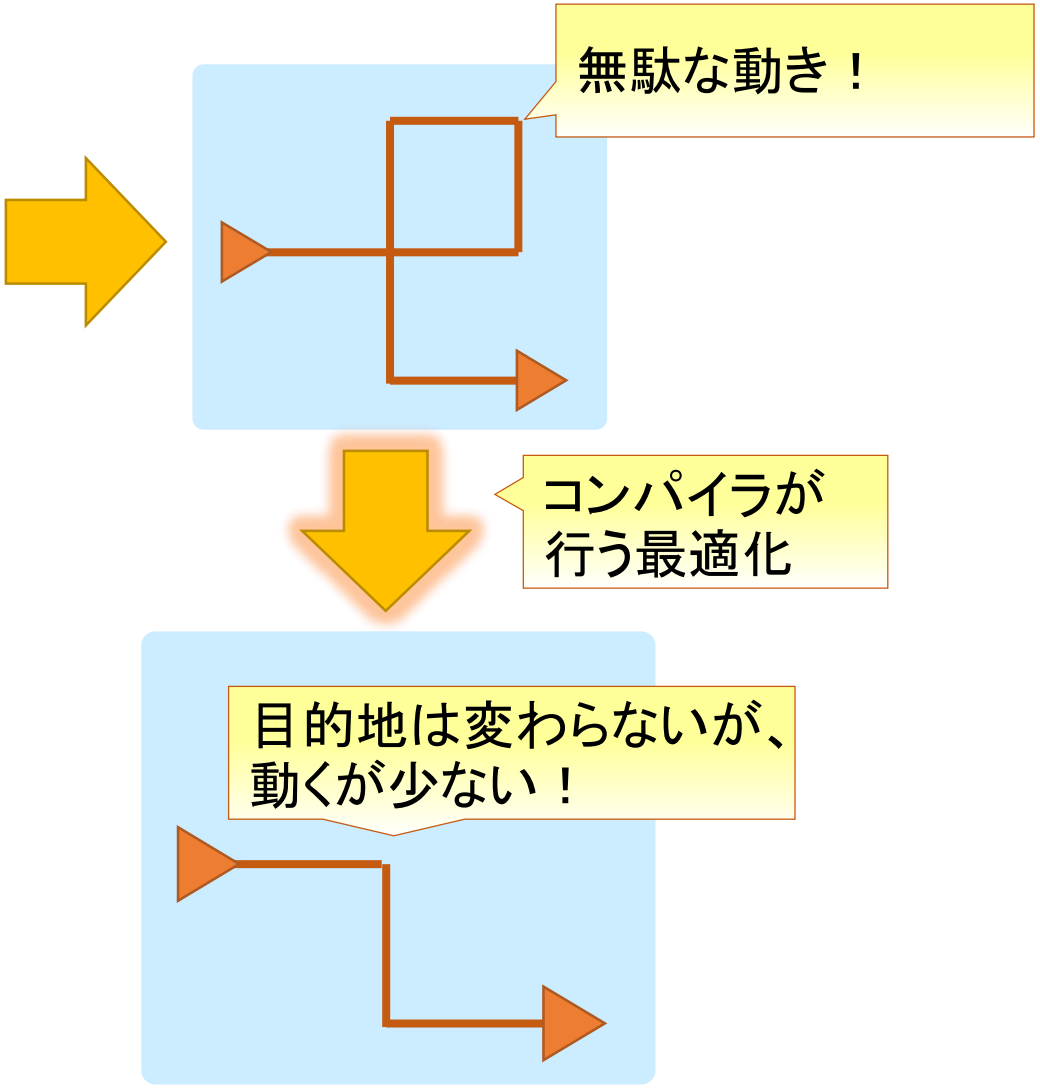
The compiler is the same !

中間言語について

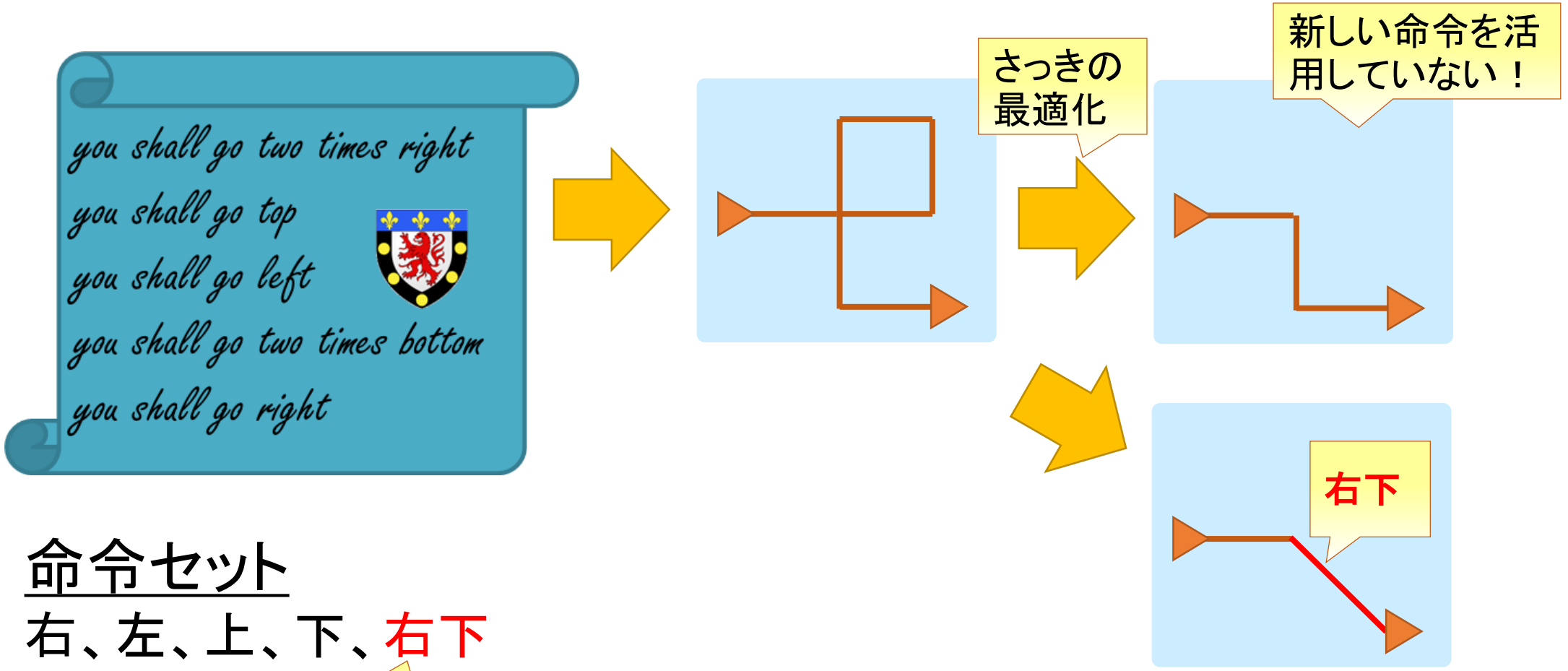


- The IR is the way the compiler represents program internally
- It expresses the important properties of the program for further analysis
- In particular, it eases **optimization**

最適化の例1：無駄な計算を削除



最適化の例2：最適な命令を使う



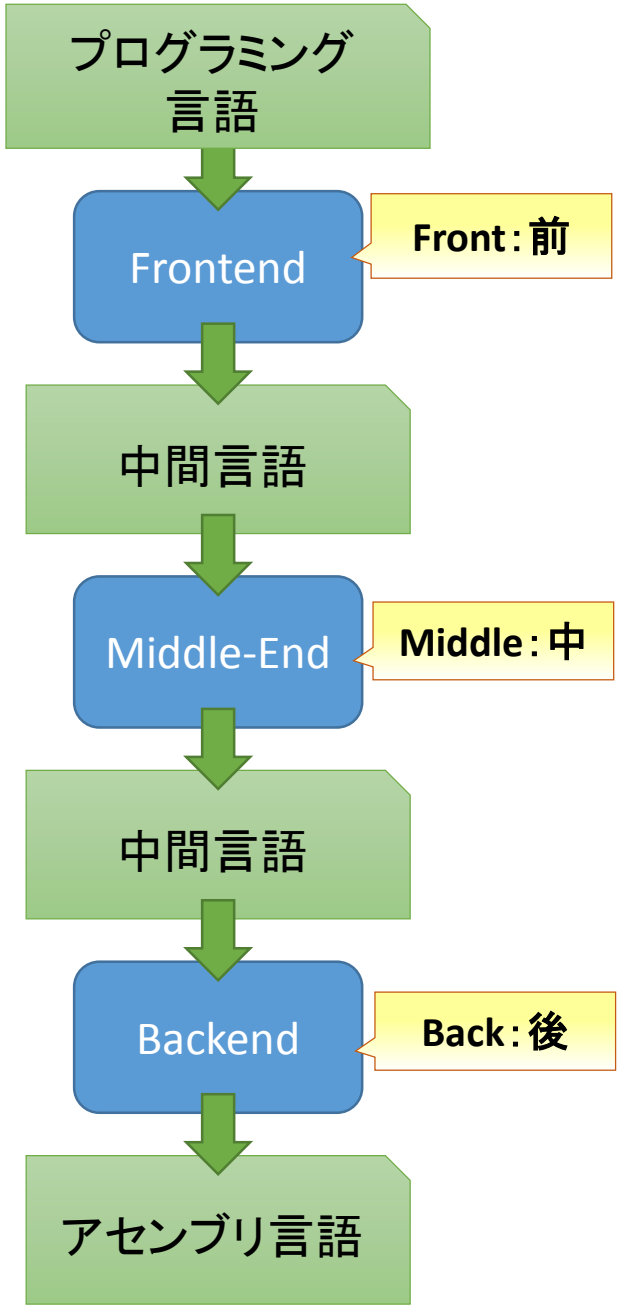
命令セット

右、左、上、下、右下

新しい命令！

Front / Middle / Back-end (1/2)

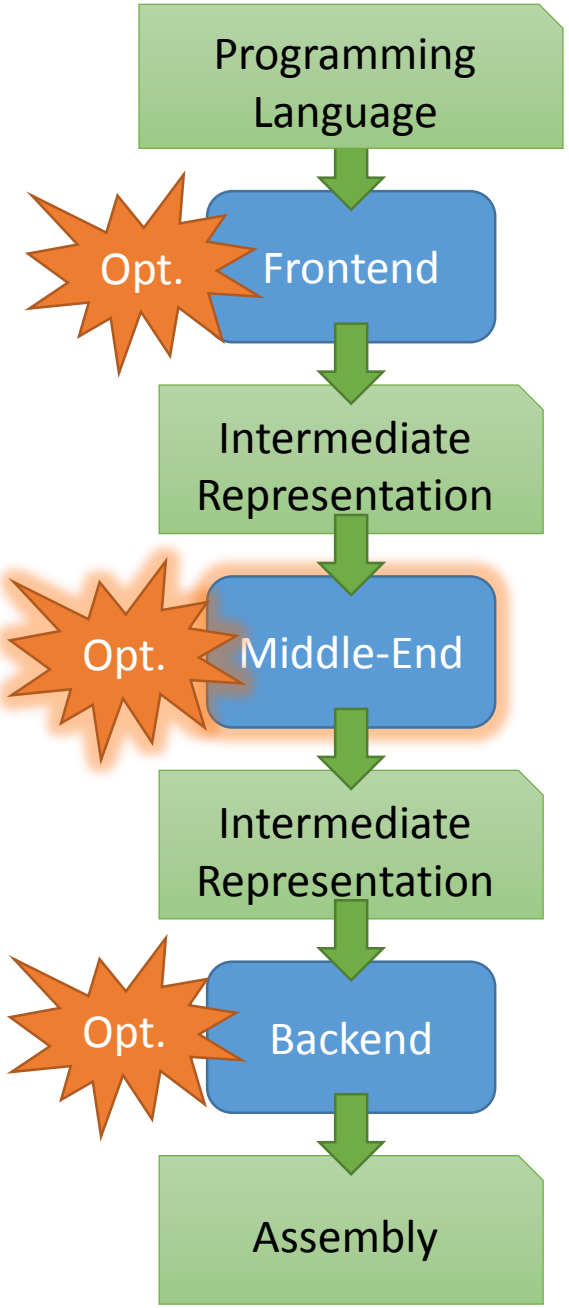
- **Frontend（フロントエンド）**
  - **Input:** Programming language
  - **Output:** Intermediate representation
  - **Key steps:** lexing, parsing
  - Often uses another IR inside for: the abstract syntax tree (AST)
- **Backend（バックエンド）**
  - **Input:** Intermediate representation
  - **Output:** Assembly
  - **Key steps:** instruction selection and register allocation
- **Middle-end（ミドルエンド）**
  - **Input:** Intermediate representation
  - **Output:** Intermediate representation
  - **Key steps:** many kinds of optimizations !
- **Intermediate representation (IR)（中間言語）**
  - Stored in memory, but can also be saved in files
  - Every compiler has its own IR (gcc, LLVM …)



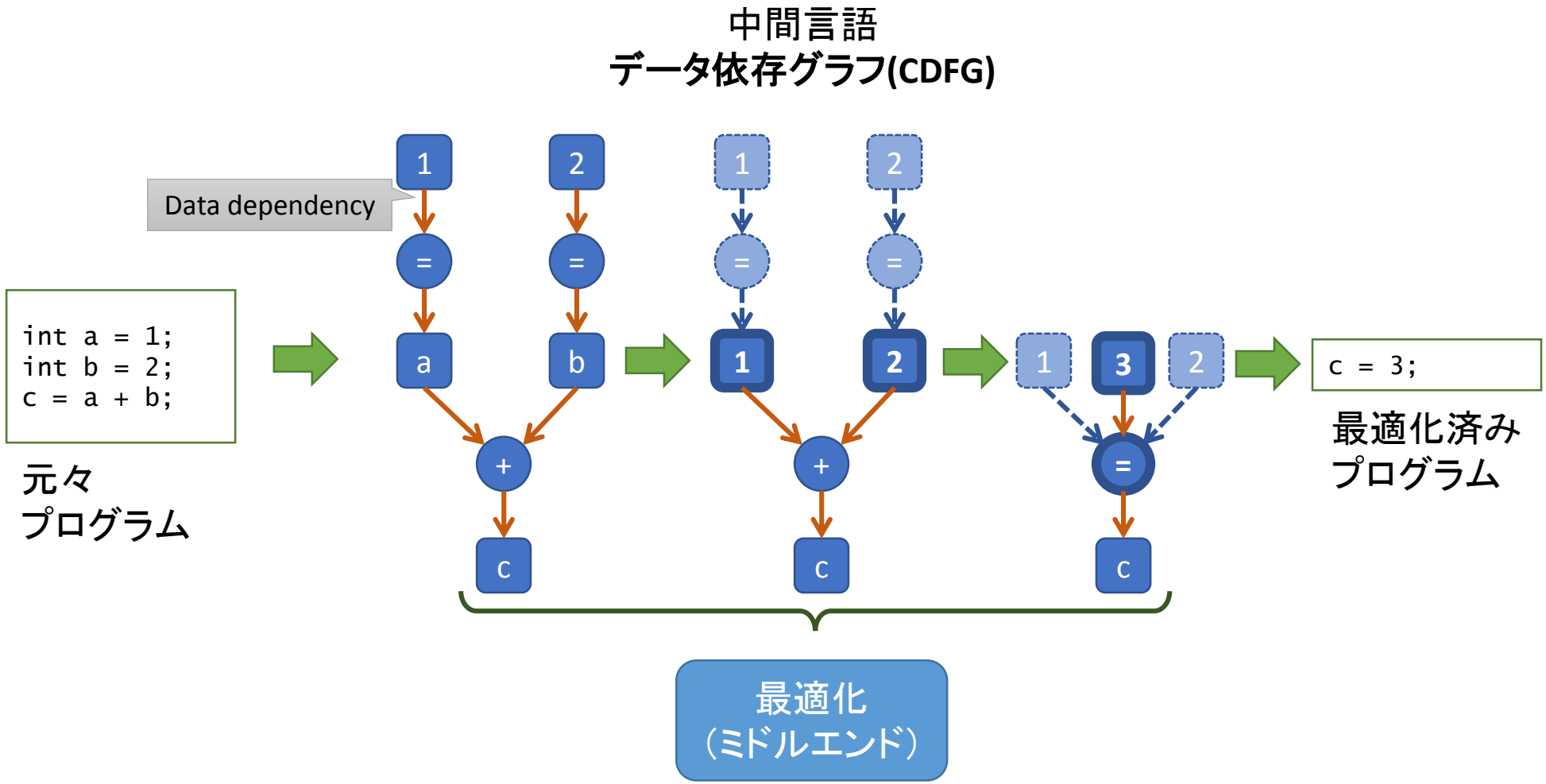
最適化はどこで行う？

- フロントエンド
  - 中間言語へ変換する時に通訳の質を高めるための技術を搭載
  - 高位レベル言語でしか分からないこともあり、フロントエンドでその機会を利用
- バックエンド
  - アセンブリ命令を選択する時に諸々な選択があり、最適な選択する必要がある
  - 例：先ほどの「右下」
- ミドルエンド
  - 主な最適化が行う

今回の着目

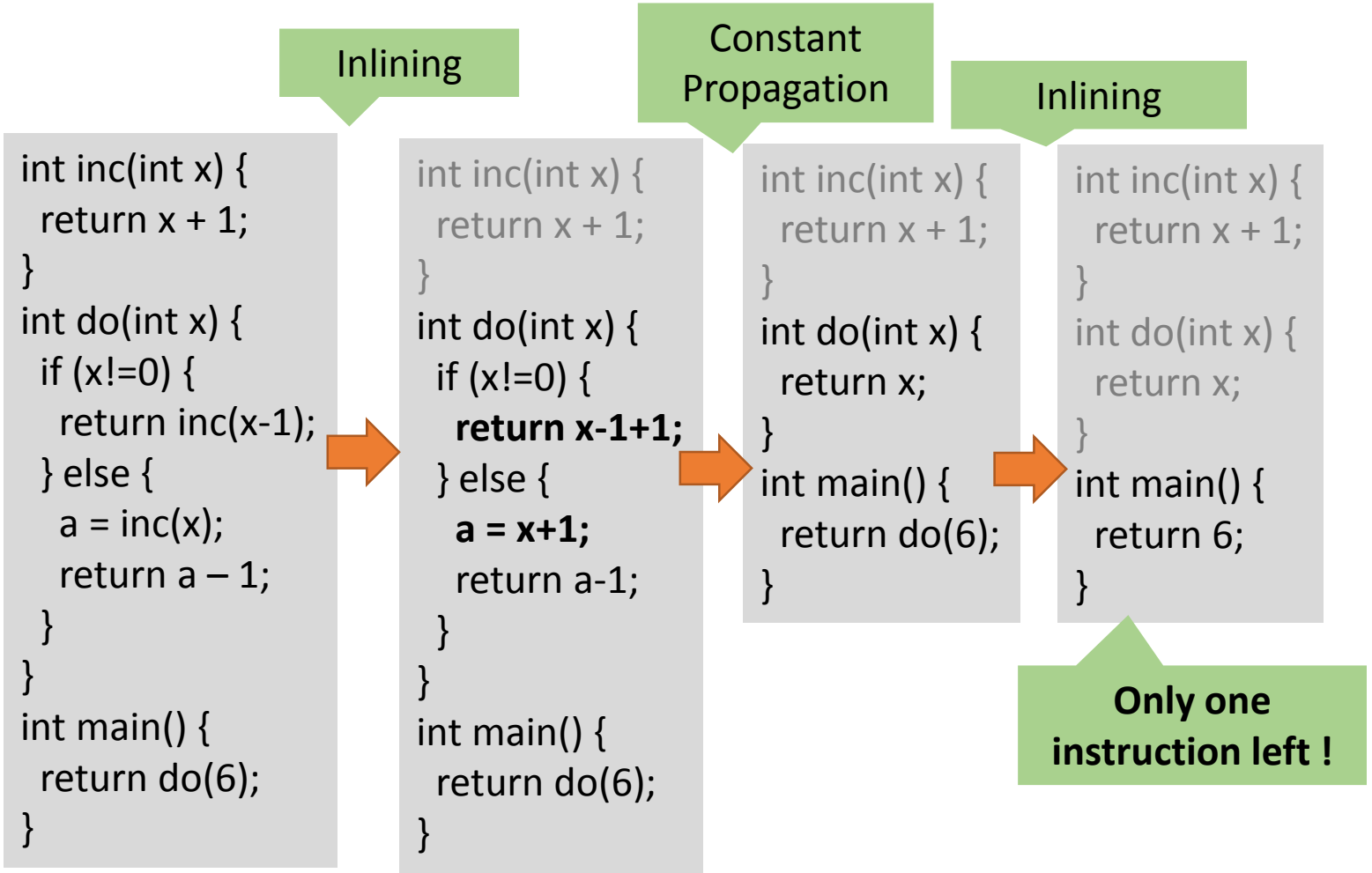


無駄な計算の削除：実例1
Constant Propagation



無駄な計算の削除：実例2

Constant Propagation ＋ Function Inlining



無駄な計算の削除：実例2

Constant Propagation ＋ Function Inlining

```
int inc(int x) {
  return x + 1;
}
int do(int x) {
  if (x!=0) {
    return inc(x-1);
  } else {
    a = inc(x);
    return a – 1;
  }
}
int main() {
  return do(6);
}
```

最適化の前

最適化の後

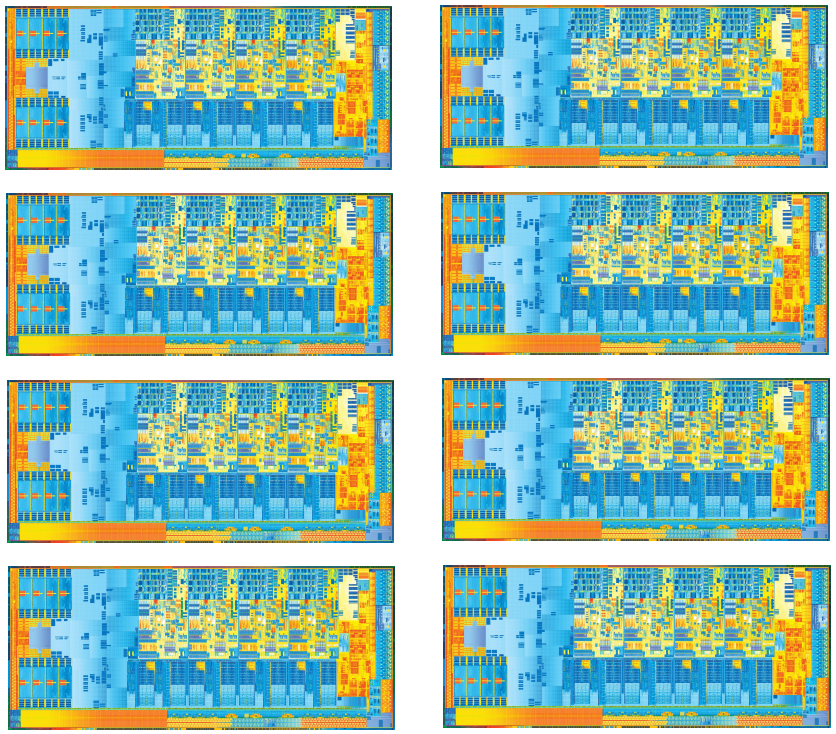
```
int main() {
  return 6;
}
```

スパコンにおける最適化
並列プログラミングの入門

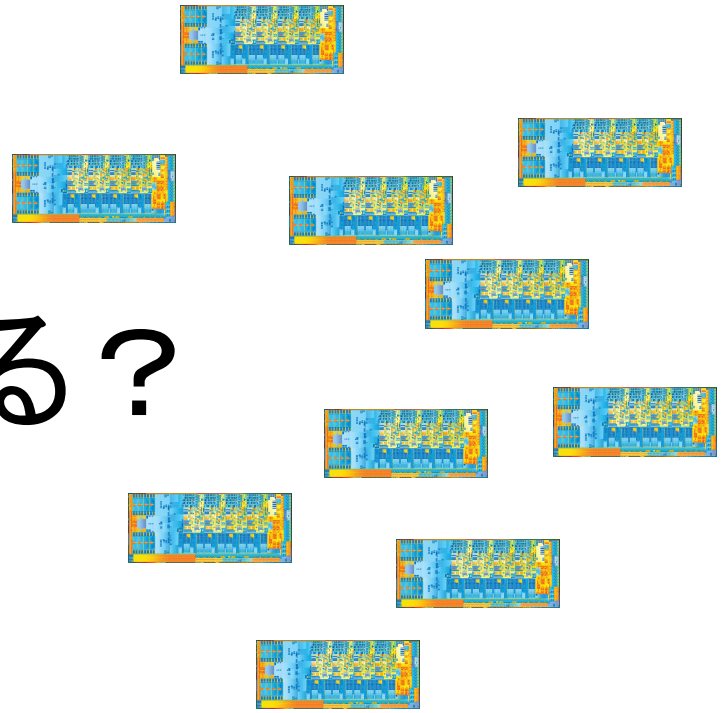
Optimizations for supercomputer

About parallel programing

スパコンは数多くのプロセッサを繋いでできたもの



Nプロセッサがあれば、



実際性能はN倍になる？

Nプロセッサがあれば、

実際性能はN倍になる？

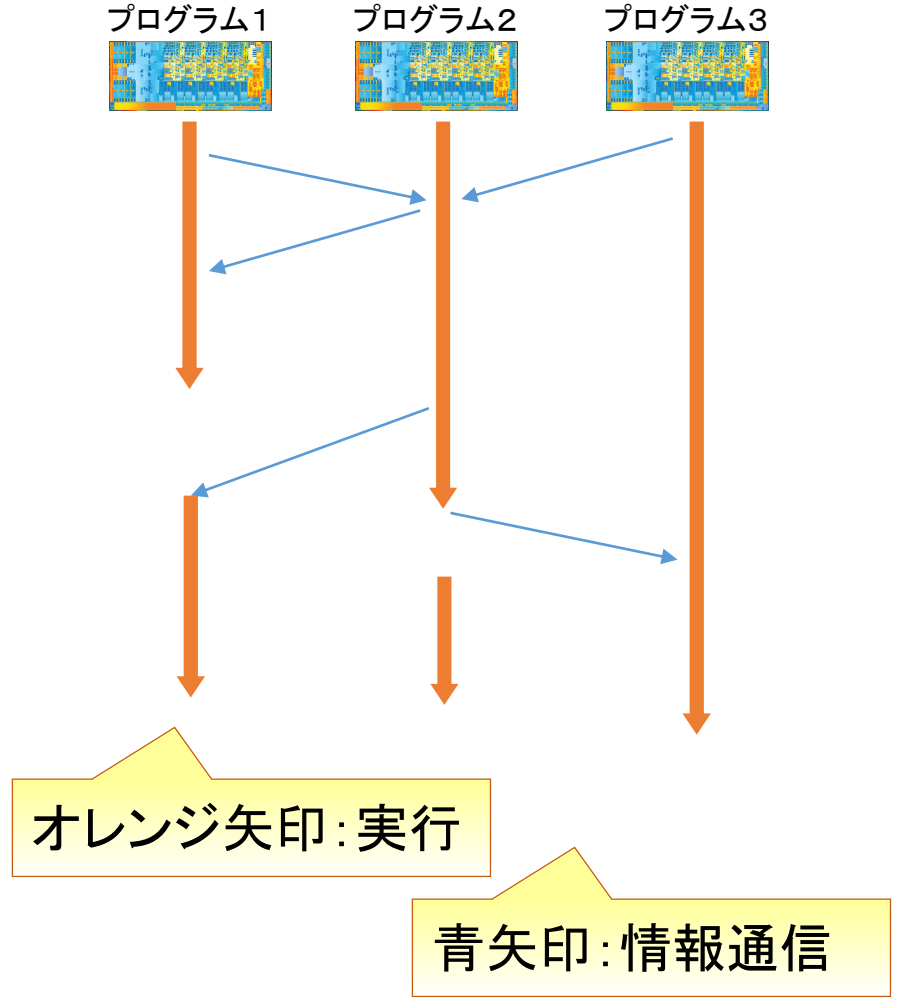
NO！

プロセッサ間の通信と協力は困難

並列プログラミングとは

- 複数のプロセッサ（あるいコア）でプログラムを実装すること
- 各プロセッサは異なる命令シーケンスを実行する
  - 同じのプログラムのコピー
  - 又は異なるプログラム
- 各プログラムは通信しています
  - 計算結果を伝える
  - 自分の状態を教える
- プログラム間は依存関係があります
  - 別のプログラムの計算結果がないと続けない
  - そのデータが遅かったら**待つ必要が出てくる**

性能低下



各プログラムは単独で実行していますが、話もしています
（図は適当なイメージ）

並列プログラミングの課題
例1:リソース共有

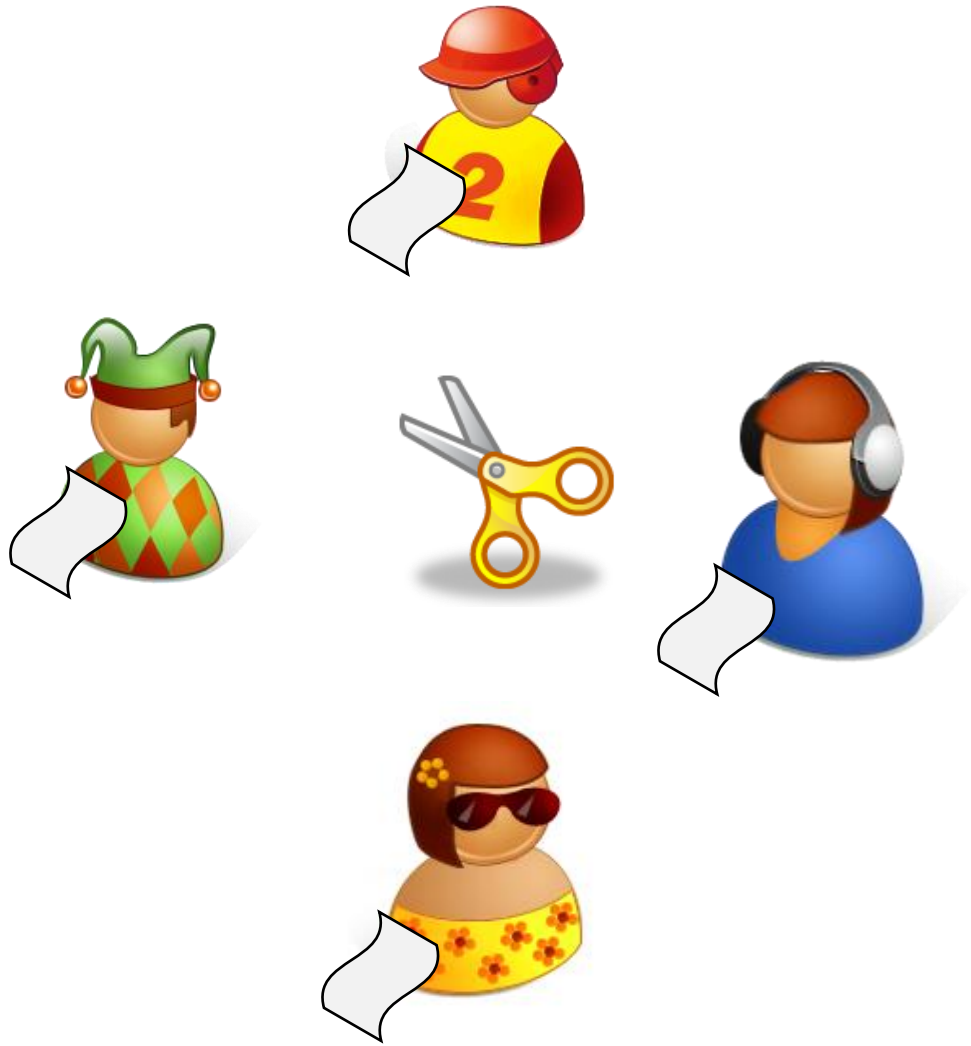
作業(計算):はさみで紙に4角を4つを切る

質問:どのぐらい早く出来るのか？

作業者の人数によります

はさみの数によります

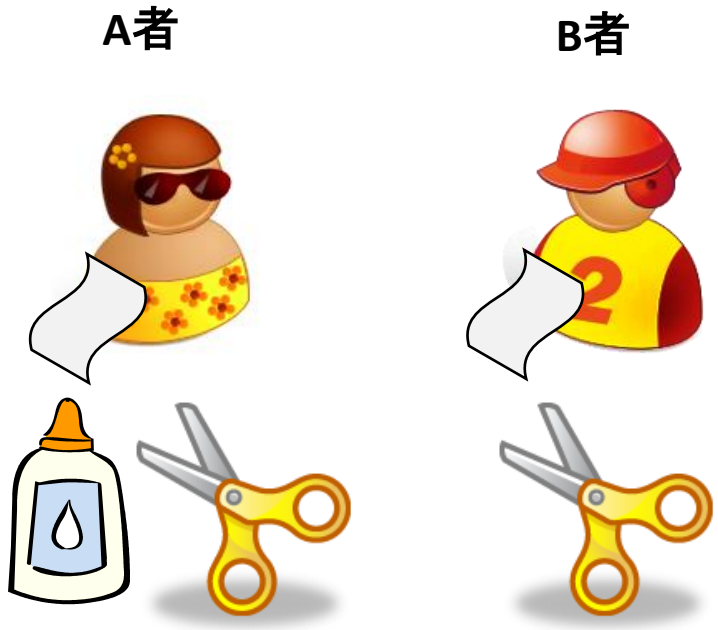
はさみは共有リソース
作業者にとってはネックです！



並列プログラミングの課題
例2：ロードインバランス

作業（計算）：

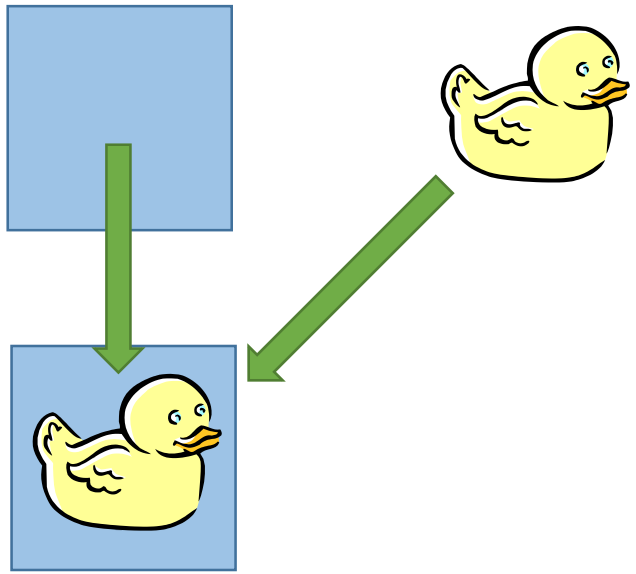
- A者ははさみで紙に4角を切って
- B者ははさみで紙に鴨を切る
- A者は四角に鴨を貼る（！）



質問：どのぐらい早く出来るのか？

A者は暇な時間が出てきます！

鴨を切るのは4角より大変ですので、時間かかります。
A者はB者が終わるまで待たないといけない：作業（ロード）バランスが悪い！



並列プログラミングの課題

複数のプログラムを協力して走らせるのは困難

- 独立したものは問題ない（「embarrassingly parallel」）
- 通信があると様々な問題が起こる

リソースの共有、ロードインバランス　など

性能低下

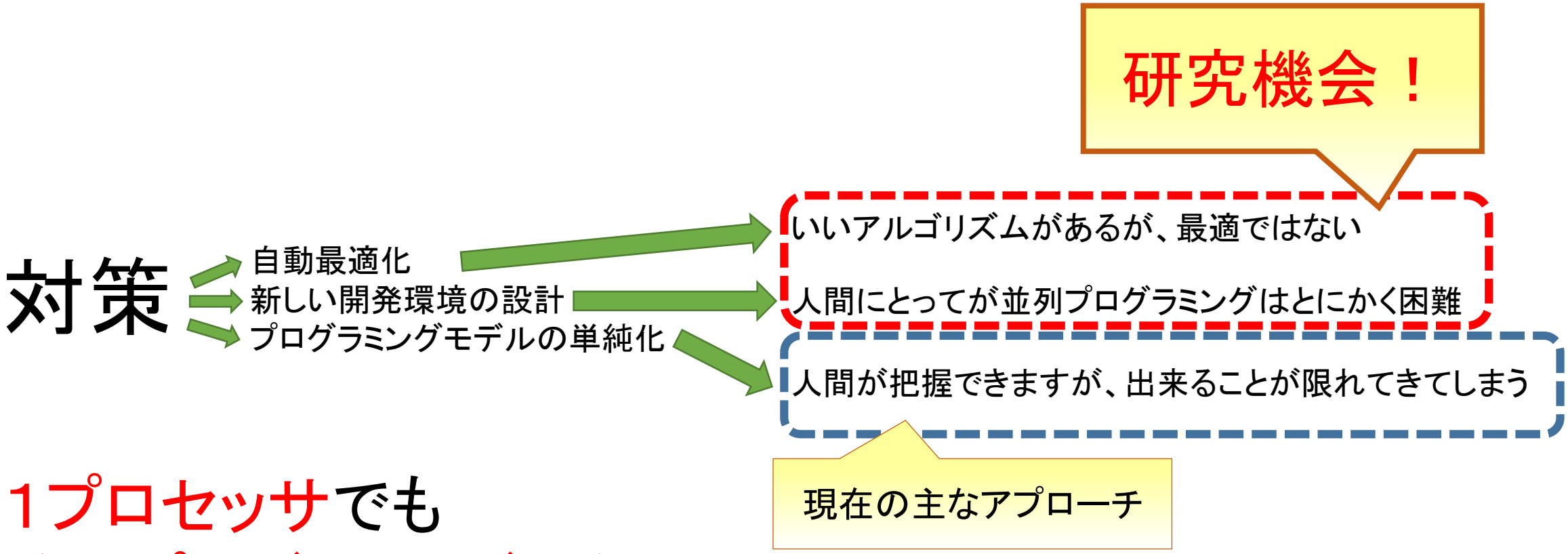
バッグ

結果は正しくない、実行が終わらない　など

対策

- 自動最適化 → いいアルゴリズムがあるが、最適ではない
- 新しい開発環境の設計 → 人間にとってが並列プログラミングはとにかく困難
- プログラミングモデルの単純化 → 人間が把握できますが、出来ることが限れてきてしまう

コンパイラ研究の課題



1プロセッサでも
並列プログラミングでも
敵している

最新開発環境の例：アップル社Xcode 6

可視化によって、開発者はプログラムの動作を把握出来ます！

僕の研究課題

コンパイラ最適化アルゴリズムの研究
（機械学習を利用）

開発者や人間がプログラムの動作や
データを把握するような技術の研究
（対話型可視化ツールの研究）

興味があれば「trouve@soc.ait.kyushu-u.ac.jp」へ

宿題

毎回忘れる人
が居る・・・

・ 回答メール・紙にちゃんと**名前**を書いてください！
・ メールで渡す送る人場合は件名に「計算機科学入門宿題」を書いてください！

宿題が3つありますが、
いずれかの宿題をしてください。
「 trouve@soc.ait.kyushu-u.ac.jp 」宛に送ってください。
締切：2014年7月14日

Homework 1
Supercomputer

The fastest supercomputer in Japan, the **K computer**, features 88128 processors and can execute at most 10^{16} floating point calculations per second.

In the following questions, we suppose that the processors cores will have the same performance and power consumption in the future.

1. The next milestone of supercomputer ("**exascale**") is expected to execute 10^{18} floating point operations per second. **How many processors** of the current machine would that require ?
2. In fact, we cannot raise the total number of processors, but we can raise the number of core per processor. The K computer features 8 cores per processor. **How many cores** would be required per processor in an exascale supercomputer ?
3. The K computer dissipates 10 MWatt of power. Please give an estimation of the **power consumption** an exascale supercomputer (hint: consider that all the power is dissipated by processor cores).
4. Please give your opinion on the **feasibility** of an exascale supercomputer with current technologies (hint: the power consumption of one household is about 5KW)
5. アンケート:プログラムを書いたことありますか？

Homework 2 Compilation

Program 1

```
a = b*2;
c = a*3;
```

Program 2

```
a = b*2;
c = a*3011;
```

We consider the assembly language with the following unique instruction:

- ***mult var val1 val2*** (corresponds to “var = val1 * val2”)
- ex: “*mult a b 6”* is the assembly for “a = b * 6”

1. Compile ***Program 1*** to this assembly

The assembly instruction **mult** corresponds to the following machine code:

0001<var><val1><val2>0000

Where <x> defined as follows:

- if x is a variable, <x> is the ASCII code of x (7 bits) prefixed by one bit set at “0”
  - ex: “d” becomes “01100100” (the ASCII code of “d” is “1100100”)
- if x is a number, <x> is the value of x coded in 7 bits, prefixed by one bit at “1”
  - ex: “42” becomes “101010” (“42” is “0101010” in binary)

2. Compile ***Program 1*** in machine code (hint: the code for the first instruction is “0001 0110 0001 0110 0010 0000 0010 0000”)
3. Explain why we can’t express ***Program 2*** in this machine code.
4. アンケート：プログラムを書いたことありますか？

Homework 3
Code Optimization

Program 1

```
a = b*2;
c = a*3;
```

1. Convert ***Program 1*** to IR (use the IR of slide 47 - graph)

<u>In the following, express all your answers in IR.</u>

2. How would you simplify ***Program 1*** code ? (hint: the optimized program would be one expression)

Some processors are faster to perform bit shift (<<) than multiplication (*). A bit shift is so that:

$$a * 2^n = a \ll n.$$

3. Replace the first multiplication of Program 1 with a bit shift

Let us consider the following optimization techniques

- ***s***: transforms multiplications by shifts when possible
- ***f***: replace two successive multiplications by a constant by one
  - ex: “u=v*21; w = u * 2” becomes “w = v * 42;”

4. Apply ***s*** and ***f*** to ***Program 1*** in the following order: ***sf*** then ***fs***. Are the results the same ?
5. We consider that a multiplications executes in 20 ns, and a shift in 10 ns. Please calculate the execution time of ***Program 1*** and the two programs you wrote in ***question 4***. Which one is the fastest ?
6. アンケート：プログラムを書いたことありますか？