

KYUSHU UNIVERSITY · SYS-104

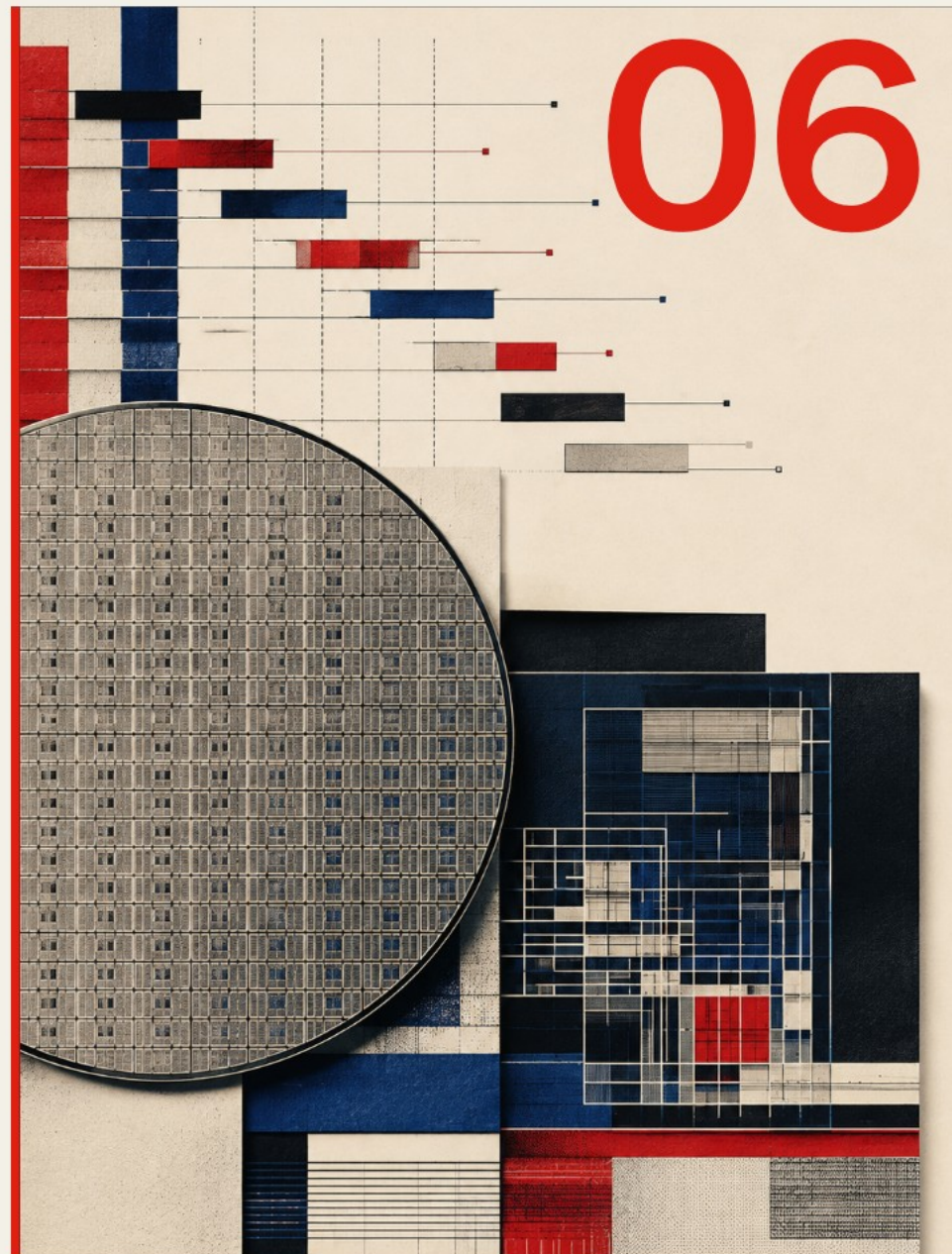
# 性能評価と チップ設計

速さを測り、回路を形にする

2026 年 9 月

Olivia Chen Toyama · C2LAB

LECTURE 6



# 同じ命令でも、速さは変わる

IMPLEMENTATION A

LOAD / ADD

同じ仕事を実行（数値は仮定例）

12.0 ms

=

同じ ISA  
同じ結果

IMPLEMENTATION B

LOAD / ADD

内部の作り方を改善（仮定例）

4.8 ms

# 写真処理の「速さ」を、どう測る？



**一枚の結果を、早く見たい**

開始してから終わるまでの時間を測る

**たくさんの写真を処理したい**

一分間に、何枚終わられるかを測る

**同じ「速い」でも、目的が違う**

まず仕事と、測る量を決める

GHz やコア数だけでなく、使う人が待つ時間と、完了する量を見る

# 今回の到達目標

01 / MEASURE

## 性能を測る

応答時間、スループット、CPU 時間の式を使い分ける

02 / IMPROVE

## 速くする

パイプラインとボトルネックから、改善の効く場所を見つける

03 / REALIZE

## 回路を形にする

HDL、EDA、製造、検証のつながりを説明する

目標 01 → 第1部 / 目標 02 → 第2・3部 / 目標 03 → 第4・5部

# 講義の流れ

目標 01 性能を測る

目標 02 速くする

目標 03 回路を形にする

PART 01

## 性能の 測り方

時間・処理量  
CPI・IPC

p.05-11

7 頁

PART 02

## 高速化の 工夫と限界

多核・パイプライン  
待ちの対策

p.12-27

16 頁

PART 03

## 実際の仕事の ボトルネック

局所性・I/O  
測定と改善

p.28-35

8 頁

PART 04

## 回路の 設計と検証

HDL・検証  
合成・配置配線

p.36-46

11 頁

PART 05

## 半導体チップの 製造

ウェハ加工  
実装・検査

p.47-62

16 頁

→ **第 1 部** 性能の測り方

導入 p.01-04 / まとめ・次回 p.63-64

# 応答時間とスループット

同じ画像フィルタを使う仮定例：一枚の処理に 0.8 秒、4 枚まで同時に処理できる

## RESPONSE TIME

### 一枚の結果を待つ時間



開始 0                      完了 0.8 秒                      1.6 秒

開始から完了まで

**0.8 秒**

## THROUGHPUT

### 時間あたりに終わる枚数

|      |      |
|------|------|
| 写真 A | 写真 E |
| 写真 B | 写真 F |
| 写真 C | 写真 G |
| 写真 D | 写真 H |

0                      0.8 秒                      1.6 秒

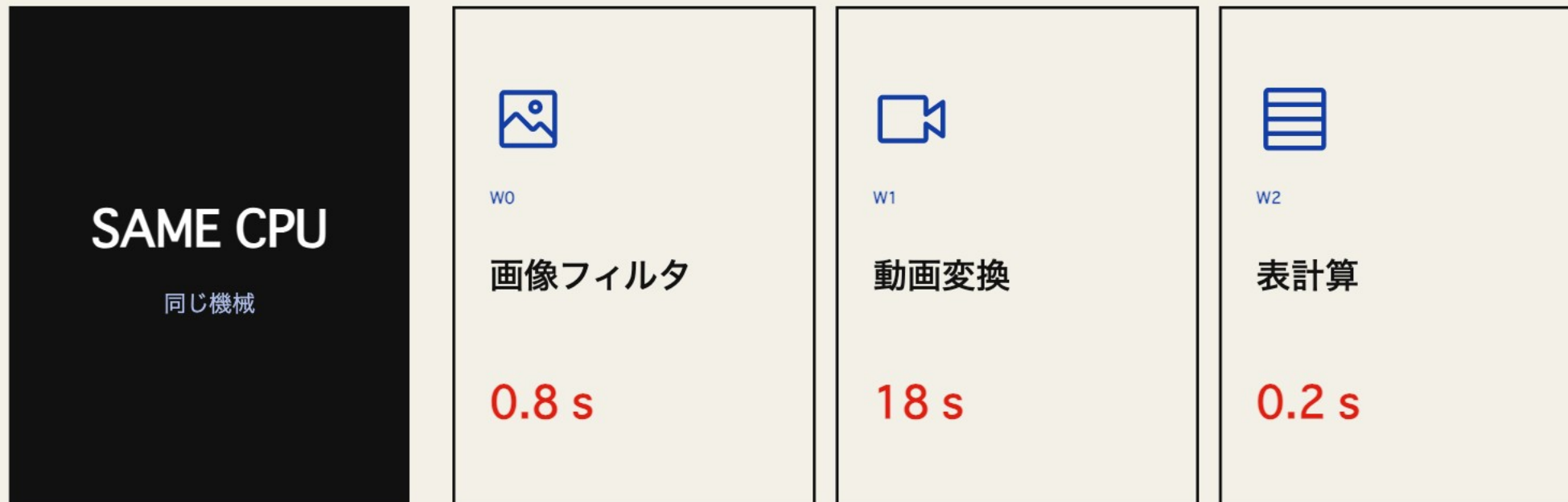
8 枚 ÷ 1.6 秒

**5 枚 / 秒**

たくさん同時に終わられても、**一枚が終わる時間は別に測る**

独立した写真を処理するモデル。待ち行列・起動時間・資源の取り合いは省略。

# 仕事が変われば、速さも変わる

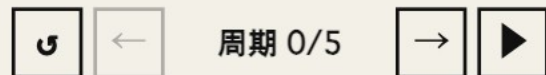


数値は仮定例 比較するときは、処理内容・入力・条件をそろえる

# 一周期で、一命令が終わる？

前回の五段階を、今回は各段階一周期と仮定して数える

**LOAD：データを読み、レジスタに残す**



経過した周期

0



完了した命令

0

開始前：これから一命令を進める

クロックは **進めるタイミング** 命令の完了を数える合図ではない

**では、M の段階でデータが届かなかったら？ →**

ここからは簡略モデル：一命令ずつ、各段階を一周期で実行する 実際の CPU が必ず五周期で動くという意味ではない

# データを待つ間も、時刻は進む

同じ CPU ・ 同じクロック ・ 同じ四命令

① LOAD → ② ADD → ③ ADD → ④ ADD

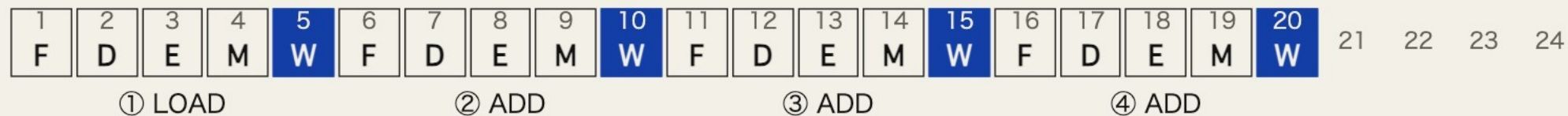
🔄 ⏪ 周期 24/24 ⏩ ▶

🔧 CPU ↔ キャッシュ → 主記憶

近くになれば、データが届くまで待つ

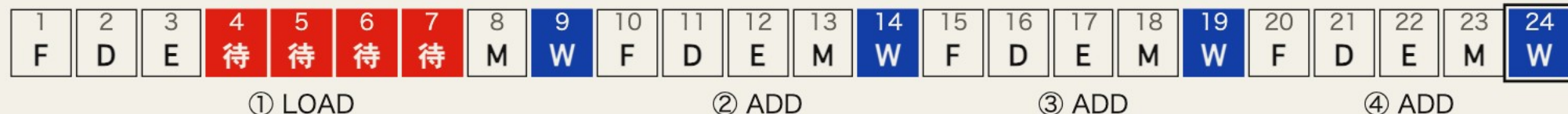
キャッシュにある

4 / 4 命令完了 全部で 20 周期



主記憶から届くのを待つ

4 / 4 命令完了 全部で 24 周期



待ち始めを見る →

**追加の四周期は、命令を増やした時間ではなく待ち時間**

このモデルでは、LOAD が終わるまで後続の ADD も進まない

仮定例：命令を重ねずに実行し、キャッシュにない場合だけ四周期の待ちを追加 ADD の M はデータにアクセスせず通過

# 命令の進み方を測る：CPI と IPC

前頁で数えた結果 完了数はどちらも **4 命令**

待ちなし **20 周期** / 待ちあり **24 周期**

## CPI Cycles Per Instruction

一命令あたり、平均何周期？

総周期数 ÷ 完了命令数

待ちなし  $20 \div 4 = 5$

待ちあり  $24 \div 4 = 6$

待ち時間も含めると

**一命令あたりの平均負担が増える**

CPU 時間の計算に使う

## IPC Instructions Per Cycle

一周期あたり、平均何命令？

完了命令数 ÷ 総周期数

待ちなし  $4 \div 20 = 0.20$

待ちあり  $4 \div 24 \approx 0.167$

同じ時間に完了できる

**命令のペースが落ちる**

命令がどれだけ進んだかを調べる

同じ実行を、割り算の向きを変えて見ている  **$IPC = 1 \div CPI$**

# 周期数から、かかった時間へ

同じ四命令の例を、実際の時間に直す

命令数だけでなく、待ちを含む周期数が必要

仮に 1 GHz

一秒に十億周期 → 一周期は 1 ns

$$\text{完了命令数} \times \frac{\text{一命令あたりの平均周期数}}{\text{CPI}} \div \frac{\text{一秒あたりの周期数}}{\text{周波数}} = \text{CPU 時間}$$

キャッシュにある

同じ 1 GHz

4 命令 × 5 周期 / 命令

20 周期

20 周期 × 1 ns / 周期

20 ns

処理 20 ns

主記憶を待つ

同じ 1 GHz

4 命令 × 6 周期 / 命令

24 周期

24 周期 × 1 ns / 周期

24 ns

処理 20 ns

待ち 4

周波数が同じでも、待ちが増えると CPI が上がり、時間も延びる

では、周波数を上げ続けることはできる？ →

数値は前頁と同じ簡略モデルで、実機の性能値ではない 1 ns = 十億分の一秒 アプリ全体の I/O 待ちとは区別する

# 高速化の工夫と限界

目標 01 性能を測る

目標 02 速くする

目標 03 回路を形にする

## PART 01

### 性能の 測り方

時間・処理量  
CPI・IPC

p.05-11

7 頁

## PART 02

### 高速化の 工夫と限界

多核・パイプライン  
待ちの対策

p.12-27

16 頁

## PART 03

### 実際の仕事の ボトルネック

局所性・I/O  
測定と改善

p.28-35

8 頁

## PART 04

### 回路の 設計と検証

HDL・検証  
合成・配置配線

p.36-46

11 頁

## PART 05

### 半導体チップの 製造

ウェハ加工  
実装・検査

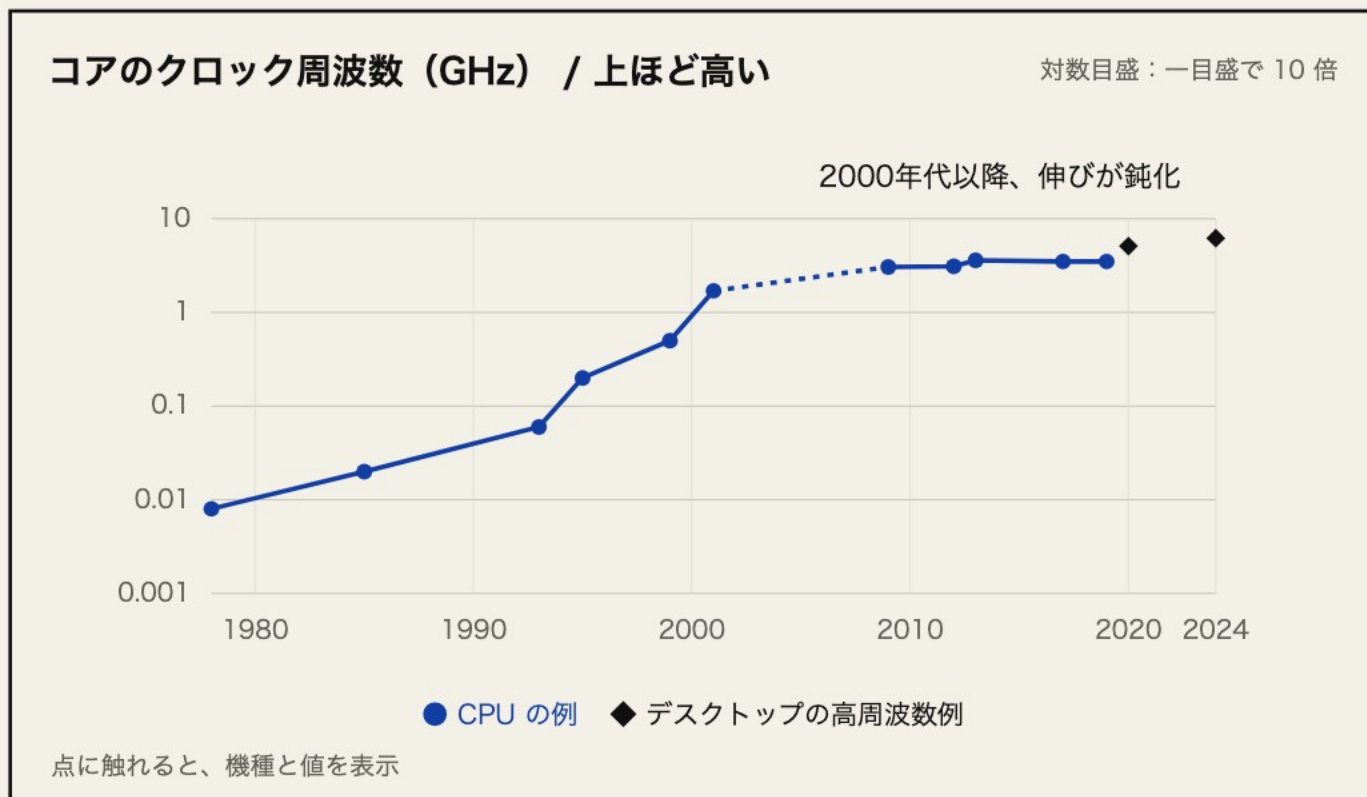
p.47-62

16 頁

→ **第 2 部** 高速化の工夫と限界

導入 p.01-04 / まとめ・次回 p.63-64

# 周波数は上げ続けられる？



## 回路の制約

**結果が安定する  
時間が必要**

論理ゲートと配線に遅延  
早すぎる合図では途中の値を取る

## 電力・熱の制約

**周波数 ↑  
消費電力・発熱 ↑**

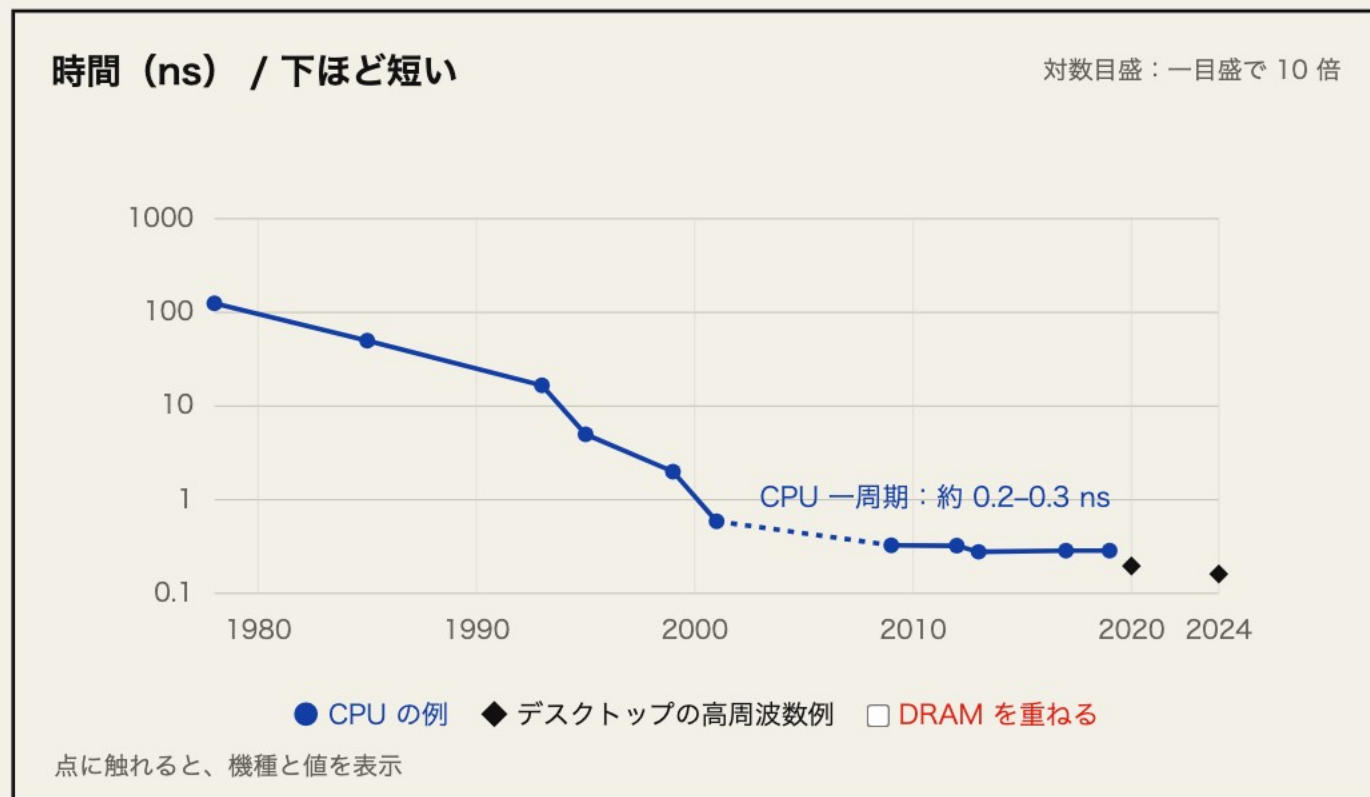
電圧も上げると負担が増える  
冷却と電源には限りがある

代表機種の公称値・最大値を掲載 線は掲載点の案内で、年平均ではない コア数の合計でも、単一コアの処理性能でもない

◆ は別系統の製品例 2024年の 6.2 GHz は条件付き最大ターボで、常時・全コアの保証値ではない

出典：[B. Gregg, USENIX LISA 2021, pp.5-6, 32](#) / [Intel, 2024](#) 掲載データの範囲：CPU 1978-2024 / DRAM 2000-2020

# キャッシュでも残る待ちを、数える



第5回 → 今回

## キャッシュで 待ちを減らした

それでも残る待ちを、  
今日は周期数で数える。

仮定例：計算 2 s + 待ち 8 s

1 倍速

2 倍速

計算ゼロ

10 秒 待ちの 8 秒は残る

CPU は掲載機種の周波数を  $1 \div \text{GHz}$  で ns に換算 DRAM はチップ内部の待ち時間の例で、主記憶アクセス全体の時間ではない  
DRAM の年・仕様は出典の世代例に従う（普及年ではない） 実際には計算と待ちが重なる場合もある

出典：[B. Gregg, USENIX LISA 2021, pp.5-6, 32](#) / [Intel, 2024](#) 掲載データの範囲：CPU 1978-2024 / DRAM 2000-2020

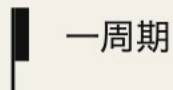
# 現在の DDR5 にも、待ち時間は残る

CPU の一周期に対して、RAM の読み出しにはどれくらい待つ？

CPU の一周期

**約 0.175 ns**

5.7 GHz のとき



RAM の CAS 待ち時間

**16.25 ns**

DDR5-6400 / CL52 の例



同じ時間軸で比べると

**約 93 周期分**

CPU が速くても、データの到着を待つ時間がある

**CAS は読み出し待ちの一部分**

メモリ全体のアクセス時間や、実際に CPU が止まる時間と同じではない

例：Intel Core Ultra 9 285K 最大 5.7 GHz / Kingston KVR64A52BS8-32。動作条件で値は変わる。

# 周波数以外の工夫

周波数には限界があり、データ待ちも残る

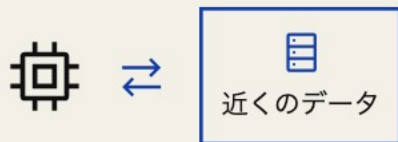
そこで、周波数以外の工夫を考える

前回の復習

Lecture 5

## 待ちを減らす

### キャッシュ



よく使うデータを CPU の近くに置き、メモリまで取りに行く回数を減らす

前回の「机の引き出し」を思い出す

このあと

次のページ →

## 仕事を分担する

### マルチコア



独立した仕事を複数のコアに分け、同時に処理する

まず、写真を分担するとどう変わるか

このあとの話題

処理を重ねる

## 段階を重ねる

### パイプライン

命令 A

|   |   |   |   |   |
|---|---|---|---|---|
| F | D | E | M | W |
|---|---|---|---|---|

命令 B

|  |   |   |   |   |   |
|--|---|---|---|---|---|
|  | F | D | E | M | W |
|--|---|---|---|---|---|

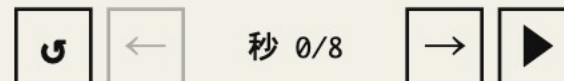
一つのコアの中で、複数の命令の異なる段階を同時に進める

Lec5 の五段階を、重ねて使う

多コア化だけでメモリ待ちは消えない コアが増えるほど、メモリへのアクセスが混み合う場合もある

# 複数のコアで、写真を分担する

 8 枚の写真を、同じフィルタで変換



**コア** = 命令を実行する CPU 内の計算単位    **マルチコア** = 複数のコアで分担

**1 コア** 一枚ずつ、順番に 0 / 8 枚 完了

☞ 1  1 処理中  2 待機  3 待機  4 待機  5 待機  6 待機  7 待機  8 待機

8 枚すべての完了まで 8 秒

**4 コア** 別の写真を、同時に 0 / 8 枚 完了

|     |                                                                                            |                                                                                           |
|-----|--------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| ☞ 1 |  1 処理中  |  5 待機  |
| ☞ 2 |  2 処理中  |  6 待機  |
| ☞ 3 |  3 処理中  |  7 待機  |
| ☞ 4 |  4 処理中 |  8 待機 |

8 枚すべての完了まで 2 秒

**一枚の処理は、どちらも 1 秒**

独立した写真を同時に処理するので、全体は早く終わる

仮定例：各写真 1 秒、同じ速さのコア、並列処理に対応したプログラム    メモリ待ち・分担の手間は省略

**どんな仕事でも 4 倍速、ではない**

前の結果が必要な部分は分担できない → 次の例へ

# アムダールの法則



次は、準備を含む別の仕事で考える  
 仮定例：準備 3 秒 + 画像変換 7 秒

Gene M. Amdahl · 1967

同じ仕事を、何コアで？

1 2 4 8 ∞

順番に行う準備

分担できる画像変換



1 コアの時 10 秒



1 コアのまま 10 秒

変わらない 3 秒 + 分担する  $7 \div 1$  秒 = 10 秒

AMDAHL'S LAW

全体の速さは  
 分担できない部分に制限される

高速化率  
 もとの時間 ÷ 新しい時間 **1.00 倍**

この例:  $10 \div (3 + 7 \div N)$

N はコア数  
 無限に増やしても  $10 \div 3 \approx 3.33$  倍

コアを買い足す前に  
 どこまで分担できるかを調べる

「30% しかない準備」が、最後には全体の上限になる

仮定例：全体 10 秒のうち直列 30%・並列 70% 仕事量は固定、コアは同じ速さ、通信・分担の手間は省略

# 一つのコアの中で、もっと進める？

多コア化では、分けられる仕事を複数のコアへ渡した  
次は、一つのコアの中の使い方に注目する

ここまで / コアの間で分担

担当するコアを増やす



独立した仕事を、別々のコアで同時に処理

ここから / 一つのコアの内部

空いている段階を使う

一つのコア



命令 A を計算中に、次の命令を取れないか？

コアを増やすだけでなく、**違う命令の段階を重ねる**

第 2 部の続き：パイプライン →

重ならない実行との比較図    パイプラインもアムダールの法則を回避するものではなく、改善できる部分に効果が限られる

# 段階を分けて、別の仕事を重ねる



## 厨房なら

一皿目を加熱している間に、次の皿の材料を準備できる

## CPU なら

一つ目を計算している間に、次の命令を読み取れる

## パイプライン

段階ごとに担当を分け、違う命令を重ねて進める

IF 取得 → ID 解読 → EX 計算 → MEM メモリ操作 → WB 書き戻し

# 一つずつ実行する場合と比べる



完了 **0/4**    最初の命令 **5 周期**    完了間隔 **5 周期**

**I0 がすべて終わってから I1 を始める**

独立した命令・各段階 1 周期の理想モデル / MEM でメモリ操作が必要なのは LOAD・STORE

# 遅い段階が、全体の刻みを決める

各段階 1 ns

EX だけ 2 ns

同じクロックで全段階を区切るモデル



一周期の長さ 1 ns    理想の完了間隔 1 ns    一命令の通過時間 5 ns

周期数だけでなく、一周期の長さも CPU 時間を変える

CPU 時間 = 命令数 × CPI × 一周期の時間 / 最も遅い段階に合わせるモデル (レジスタ遅延は省略)

# 重ねると、どこで待つ？

結果待ち

部品の競合

行き先待ち



STEP 0/2



データハザード

## LOAD の結果を次の ADD が使う

時間 → 空欄は、この命令が進んでいない周期

LOAD → R3 ... R3 → ADD

LOAD は R3 に入れる値をメモリから読む

LOAD R3

1

2

3

4

5

6

7

8

9

IF

ID

EX

ADD R4,R3

IF

ID

## ハザード：待機や転送などで対処しないと、正しく進めない

転送経路（forwarding）があるモデル／なければ、さらに待つ場合がある

# ハザードの三つの原因

**ハザード**：命令を重ねたとき、そのままでは正しく進められない状況



**結果がまだない**

**データハザード**

前の ADD が作る値を、次の ADD が使う

どう対処する？

**結果を直接渡す／必要な間だけ待つ**

フォワーディング・ストール



**部品を取り合う**

**構造ハザード**

命令の取得とデータの読み出しが、同じ入口を使う

どう対処する？

**時間をずらす／入口や部品を分ける**

例：命令用とデータ用の入口を分離



**行き先が決まらない**

**制御ハザード**

条件分岐の結果が出る前に、次の命令を取りたい

どう対処する？

**行き先を予測／外れたら取り消して再開**

分岐予測 (branch prediction)

まず正しさを守る そのうえで、**待ち時間ややり直しを減らす**

ストール = 進行を一時的に止めること 部品の競合は構成によって異なる 次は結果の転送、その次は分岐予測を見る

# 保存を待たずに、結果を渡す

保存を待つ

結果を直接渡す



周期 0/6



ADD → ADD : フォワーディング

初期値 R1 = 0

A: R1 ← 2 + 3

B: R2 ← R1 + 4

保存を待たずに、計算済みの 5 を次の演算器へ直接渡す

A の結果

B の計算

?



待機

R1 の保存値: 0

R1 への書き込みは第5周期

時間 →

F 取得 / D 解読 / E 計算 / M / W 保存

|      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|------|---|---|---|---|---|---|---|---|
| 命令 A | F | D | E | M | W |   |   |   |
| 命令 B |   | F | D | E | M | W |   |   |

A の E 終了 (第3周期) → B の E (第4周期)

二命令の完了まで 6 周期

直接渡せるのは **すでにできた結果** LOAD のデータ到着待ちなどは残る

五段階・各段階一周期のモデル 転送なしでは第5周期の前半に書き込み、後半に読み出すと仮定 実機の待ち周期数は構成で異なる

# 分岐先を予測して、先に進む

条件：R1 が 0 なら A、そうでなければ B

条件の判定中に「A へ進む」と予測する例



今回の実際の条件

**R1 = 0 (予測が当たる)**

R1 ≠ 0 (予測が外れる)

条件を調べる



予測して先へ



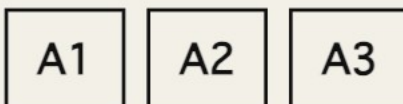
予測を確認



正しい結果を確定

行き先 A

予測した経路

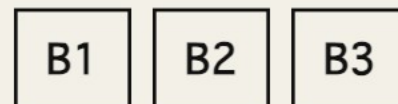


未着手

条件が分かる前に、こちらを選ぶと予測

行き先 B

もう一つの経路



未着手

R1 が 0 でないときに進む経路

## 条件の判定には時間がかかる

当たれば有効、外れればやり直しの時間が発生する

分岐予測は過去の分岐の傾向などを利用する これは動作順の模式図で、各操作が一周期という意味ではない

# 待っていない仕事を先に進める

## ① 値を読み込む

LOAD R1, [A] メモリからの到着待ち

## ② その値で計算する

ADD R2, R1, 1 ①の結果が必要

## ③ 別の値で計算する

ADD R5, R6, 1 ①②の結果は不要

## ③だけ先に計算できる

空いている演算器を使い、待ち時間に別の仕事を進める

## 見える結果は元の順序を守る

計算する順序と、結果を確定する順序を分ける

**アウトオブオーダー実行** 依存関係を壊さず、実行の順序を調整

# 実際の仕事で、ボトルネックを探す

目標 01 性能を測る

目標 02 速くする

目標 03 回路を形にする

PART 01

性能の  
測り方

時間・処理量  
CPI・IPC

p.05-11

7 頁

PART 02

高速化の  
工夫と限界

多核・パイプライン  
待ちの対策

p.12-27

16 頁

PART 03

実際の仕事の  
ボトルネック

局所性・I/O  
測定と改善

p.28-35

8 頁

PART 04

回路の  
設計と検証

HDL・検証  
合成・配置配線

p.36-46

11 頁

PART 05

半導体チップの  
製造

ウェハ加工  
実装・検査

p.47-62

16 頁

→ **第 3 部** 実際の仕事のボトルネック

導入 p.01-04 / まとめ・次回 p.63-64

# 同じ写真処理でも、待つ場所は違う



## 読み込む

SSD から写真のデータを持ってくる

## 計算する

RAM の画素を読み、CPU で色を変える

## 保存する

変更したデータを SSD に書き戻す

CPU の速さだけでは、この作業全体の速さは決まらない

# 局所性は、RAM を読む回数をどう変える？

隣の画素へ

離れた画素へ



読込 0/8



写真の色補正で、画素を読み出す

## RAM の 16 画素

|    |    |    |    |
|----|----|----|----|
| 0  | 1  | 2  | 3  |
| 4  | 5  | 6  | 7  |
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |

横一列が、まとめて運ぶ単位

## 小さなキャッシュ

— — — —

まだ空

同じ 4 画素を再利用できるかで  
RAM を読む回数が変わる

同じ 8 画素の要求順 0 1 2 3 4 5 6 7

HIT 0 / MISS 0

## 第5回の空間的局所性を、RAM を読む回数で確かめる

教材モデル：キャッシュは 1 ブロック、1 ブロックは 4 画素／実機のサイズ・速度とは異なる

# 条件ごとの処理は、先を予測しにくい



## 条件があるだけで遅いとは限らない

分岐を使う実装の例／コンパイラが別の命令へ変換する場合もある

### この画素は暗い？

暗い → 明るくする / それ以外 → そのまま

### 実行前に、行き先を予測

暗い・明るいの並びによって、予測の当たりやすさが変わる

### 予測が外れたら

不要な処理を取り消し、正しい行き先へ進む

# CPU を速くすると、何 ms 短くなる？

元の CPU

CPU の計算を 2 倍速に



STEP 0/3



写真を開く → 色補正 → 保存

SSD から写真を読み込む

250 ms

この仕事は、データの到着や保存完了を待つ

読み込み待ち 250

40

保存待ち 210

まず、どこに時間がかかっているか観察しよう

説明用の仮定値 / 重なりのない処理 / I/O = 入出力、SSD やネットワークとのやり取り

# 時間の内訳から、改善する場所を選ぶ

## CPU が長い

計算方法や並列化を調べる

## RAM 待ちが長い

読む順序とキャッシュの利用を調べる

## I/O 待ちが長い

データ量や保存先とのやり取りを調べる

**測る → 原因を考える → 変える → 同じ条件で測り直す**

数値は前頁と同じ仮定例／実際の測定では待ち時間とCPU時間を区別する

## 前の写真処理の例

I/O 待ち 460 ms (92%)

CPU の計算 40 ms (8%)

## まず調べる対象は？

CPU よりも、読み込み・保存の待ち時間

## プロファイリング

実行時間や待ちの内訳を測り、原因を絞る

# 同じ仕事で、改善を確かめる

写真処理を変更したあと、本当に速くなった？

枚数が違う比較

同じ写真で比較

変更前



100 枚

同じフィルタで変換

すべての処理が完了

50 秒



変更後



50 枚

同じフィルタで変換

すべての処理が完了

25 秒



仕事を半分にただけかもしれない

50 秒 → 25 秒でも、一枚あたりはどちらも 0.5 秒

入力 同じ写真・枚数・処理

数値は説明用の仮定例 変更した点以外の条件をそろえる

環境 同じ機器・設定

測定 複数回測り、中央値で比較

# 高速化を実現する回路

ここまでの高速化の工夫

実現するために必要な回路

## キャッシュ

データを近くに置く



## 記憶回路 + 検索・制御回路

必要なデータがあるか調べ、読み出す

## パイプライン

命令の処理を重ねる



## 段階の間のレジスタ + 制御回路

第4回の DFF を並べ、途中の結果を次の周期まで保持する

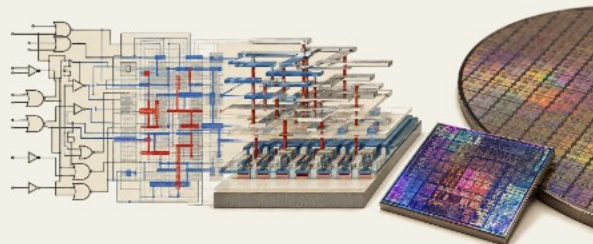
## マルチコア

独立した仕事を分担する



## 複数のコア + 接続する回路

それぞれが計算し、メモリとデータをやり取りする



次は、回路を作る側へ

**この仕組みを、どう記述して  
正しく動くと確かめる？**

**第4部：回路の設計と検証**

# 回路の設計と検証

目標 01 性能を測る

目標 02 速くする

目標 03 回路を形にする

## PART 01

### 性能の 測り方

時間・処理量  
CPI・IPC

p.05-11

7 頁

## PART 02

### 高速化の 工夫と限界

多核・パイプライン  
待ちの対策

p.12-27

16 頁

## PART 03

### 実際の仕事の ボトルネック

局所性・I/O  
測定と改善

p.28-35

8 頁

## PART 04

### 回路の 設計と検証

HDL・検証  
合成・配置配線

p.36-46

11 頁

## PART 05

### 半導体チップの 製造

ウェハ加工  
実装・検査

p.47-62

16 頁

→ **第 4 部** 回路の設計と検証

導入 p.01-04 / まとめ・次回 p.63-64

# 手作業の設計と、その限界



1970 年代初め：拡大した IC の模様を手で切り出す

写真・出典：[Computer History Museum](#)

回路が小さかった時代

## 部品と配線を 人が一つずつ決める

図を描き、接続を調べ  
製造に使う模様を作る

### 大規模化すると、手作業では限界

膨大な接続の確認と修正を  
人だけで管理しきれない

## コンピュータ自身に、設計を手伝わせる

# 設計をコンピュータに手伝わせる

**EDA** = 電子設計の自動化 Electronic Design Automation

設計・検証・変換を、コンピュータ上のツールで支える



手作業の設計

## 図と模様を描く

回路を描き、部品の位置や  
配線を人が決める

**1960-70 年代にも**

手作業での版図作成



コンピュータ支援設計 (CAD)

## 配置・配線を助ける

繰り返しの作業や  
接続の確認を機械に任せる

**1967 年**

CAD を用いた IC 設計の例



記述から回路へ

## 動作を書いて変換

言語で表した設計を検証し  
論理ゲートへ自動変換する

**1984 / 1987 年**

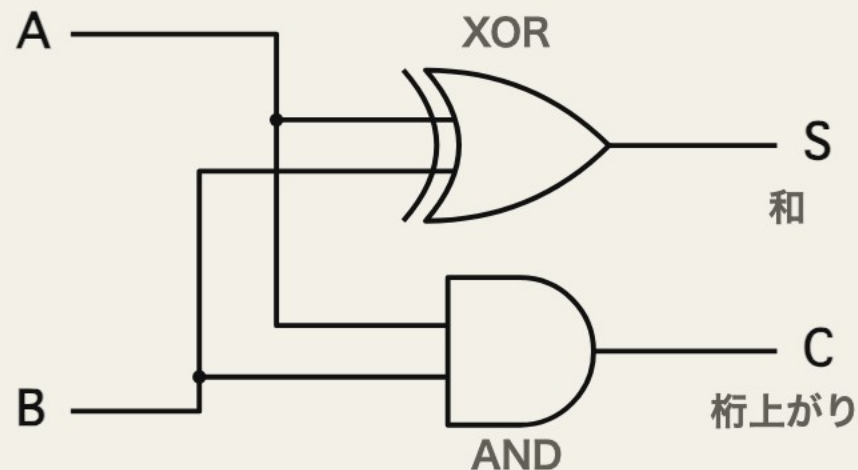
Verilog / 論理合成ツールの登場

「線を全部描く」から、「必要な動作を記述する」へ

代表的な出来事を抜粋 各方式は重なりながら発展 出典：CHM / Synopsys

# 回路の動作を、言語で表す

第 3 回の半加算器で、設計の流れを試す



同じ動作を、関係として表す

入力  $A \cdot B$  から  
出力  $S \cdot C$  を決める

$S$  (和) は、 $A$  と  $B$  が違うとき 1

$C$  (桁上がり) は、両方が 1 のとき 1

```
assign S = A ^ B;  
assign C = A & B;
```

**HDL**

ハードウェア記述言語

Hardware Description Language

回路の構造や動作を  
コンピュータが扱える形で表す言語

# 仕様から、製造用データへ



トップダウン：機能から、具体的な形へ

## HDL を使うのは、ここ

組合せ回路の関係や、クロックごとの状態更新を記述する

回路を表す設計データ

```
assign S = A ^ B;
```

```
assign C = A & B;
```

レジスタ間の処理や状態の移動を表す記述を RTL と呼ぶ

← 一直線で終わりではない  
各段階で確認し、必要なら前へ戻る

# HDL の種類と、記述の読み方

演習 2 で使う

## Verilog HDL

ここでは半加算器の記述を読む

## SystemVerilog

Verilog を拡張  
設計・検証の表現を充実

## VHDL

もう一つの代表的な  
ハードウェア記述言語

まず読むのは、この三つ

```
module half_adder (  
    input A, B,  
    output S, C  
);
```

名前と、入出力の端子

```
assign S = A ^ B;  
assign C = A & B;
```

入力と出力の関係  
^ XOR   & AND

```
endmodule
```

この回路の記述はここまで

書き方の詳しい練習は、演習 2 の前に扱う

# 二行は、同時に働く二つの回路

二行は、二つの回路の関係

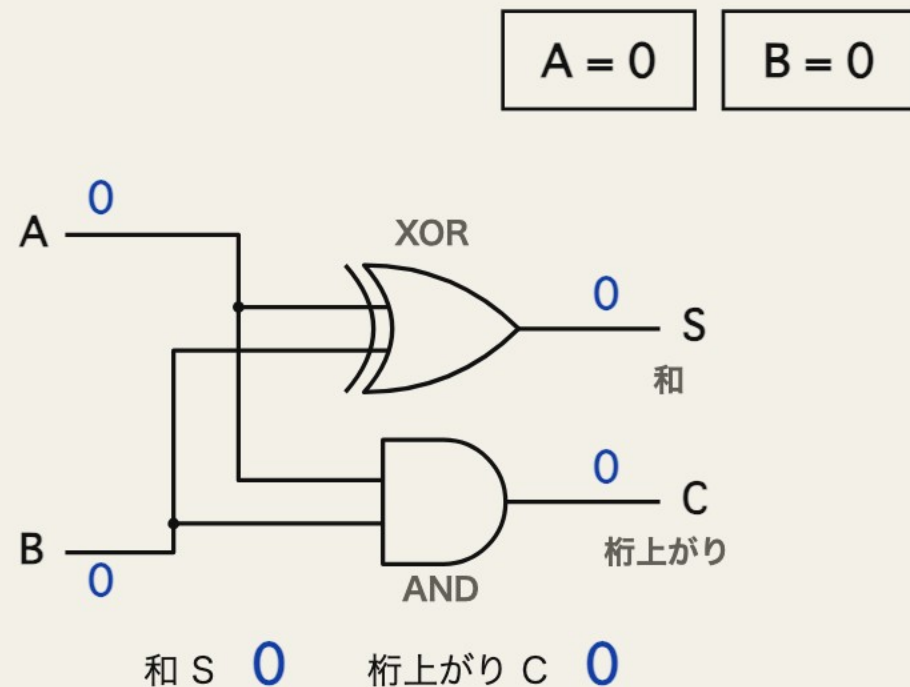
Verilog の記述

```
assign S = A ^ B;
```

```
assign C = A & B;
```

↑↓ 二行の順序を入れ替える

順序を変えても、同じ回路



## CPU が一行ずつ実行するプログラムではない

XOR と AND が並んで存在し、入力の変化にそれぞれ応答する

この例は二つの連続代入を表す 実回路では信号が伝わるまでに遅延がある

# 作る前に、動作を確かめる

半加算器の記述は、仕様どおりに動く？

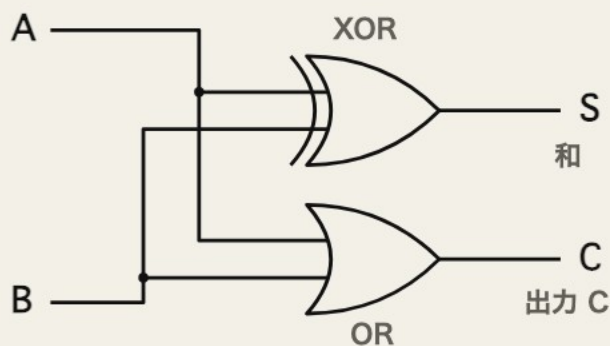
0 / 4 組 確認



`assign C = A | B;`     $S = A \oplus B$  は共通

C の式を AND へ修正

|          |          |          |          |
|----------|----------|----------|----------|
| A B = 00 | A B = 01 | A B = 10 | A B = 11 |
|----------|----------|----------|----------|



仕様とシミュレーション結果

| A B | 期待する S C | 得られた S C |
|-----|----------|----------|
| 0 0 | 0 0      | —        |
| 0 1 | 1 0      | —        |
| 1 0 | 1 0      | —        |
| 1 1 | 0 1      | —        |

期待値と実際の出力を照合する

シミュレーション：回路の動作をコンピュータ上で再現

テストベンチ：入力を与え、結果を確かめる検証用の環境

この半加算器は全 4 通りを確認できる    大きな回路では、ほかの検証方法も組み合わせる

# 論理合成で、ゲートの接続へ

同じ半加算器を、論理ゲートの接続へ

記述

論理の関係

ゲート接続

入力：合成可能な HDL

```
assign S = A ^ B;
```

```
assign C = A & B;
```

論理合成ツール

使える部品の情報

速度・面積などの設計条件

まだ、線の位置は決めていない

## S と C はどう決まる？

演算子と入出力の関係を  
ツールが解釈する



## 動作の意味を保ち、実装する回路へ変換する

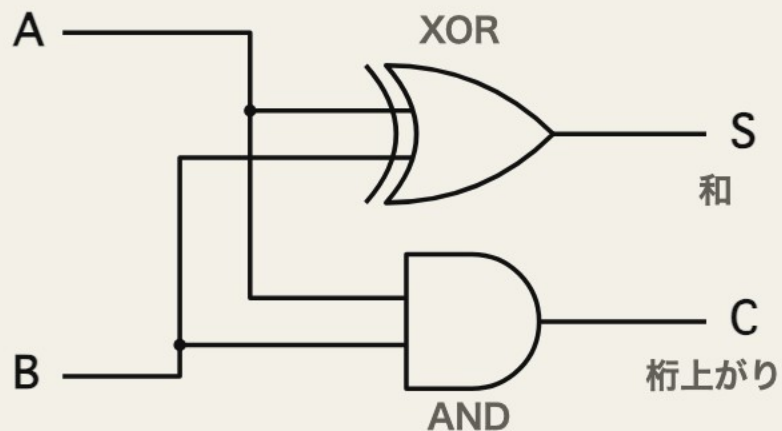
論理合成は「どの部品を、どうつなぐか」まで 実際の位置と配線は次の工程

ここでは XOR と AND で示す 実際の部品構成はライブラリや設計条件で変わる

# 部品を置き、配線をつなぐ

| 接続 | 配置 | 配線 |
|----|----|----|
|----|----|----|

同じ半加算器を、物理的な設計へ



論理合成の出力：ネットリスト

工程 1 / 3

## 接続だけが決まっている

XOR と AND に  $A \cdot B$  をつなぐ まだ部品の座標や線の通り道はない

✓ 論理的な接続を確認

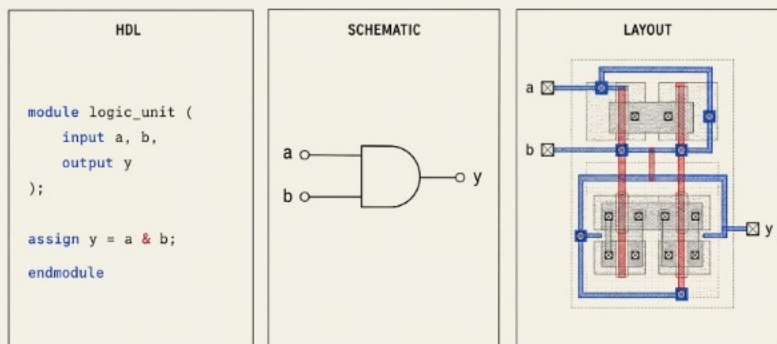
## 同じ動作でも、配置や配線で速さは変わる

条件を満たすまで設計を調整し、最後に層ごとの製造用データを出力する

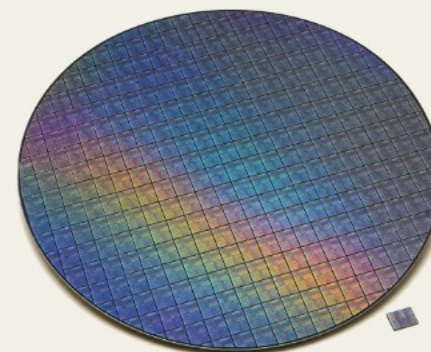
遅延は仮定値 (ns)。強調した S の出力配線以外の条件を固定した比較。実際は負荷や配線層も影響する。

# 設計データを、実物のチップへ

ここまで：設計データ



次は：製造



動作・接続・配置を決める

模様を写し、材料を加工する

**完成したのは、チップではなく設計データ**

# 半導体チップの製造

目標 01 性能を測る

目標 02 速くする

目標 03 回路を形にする

## PART 01

### 性能の 測り方

時間・処理量  
CPI・IPC

p.05-11

7 頁

## PART 02

### 高速化の 工夫と限界

多核・パイプライン  
待ちの対策

p.12-27

16 頁

## PART 03

### 実際の仕事の ボトルネック

局所性・I/O  
測定と改善

p.28-35

8 頁

## PART 04

### 回路の 設計と検証

HDL・検証  
合成・配置配線

p.36-46

11 頁

## PART 05

### 半導体チップの 製造

ウェハ加工  
実装・検査

p.47-62

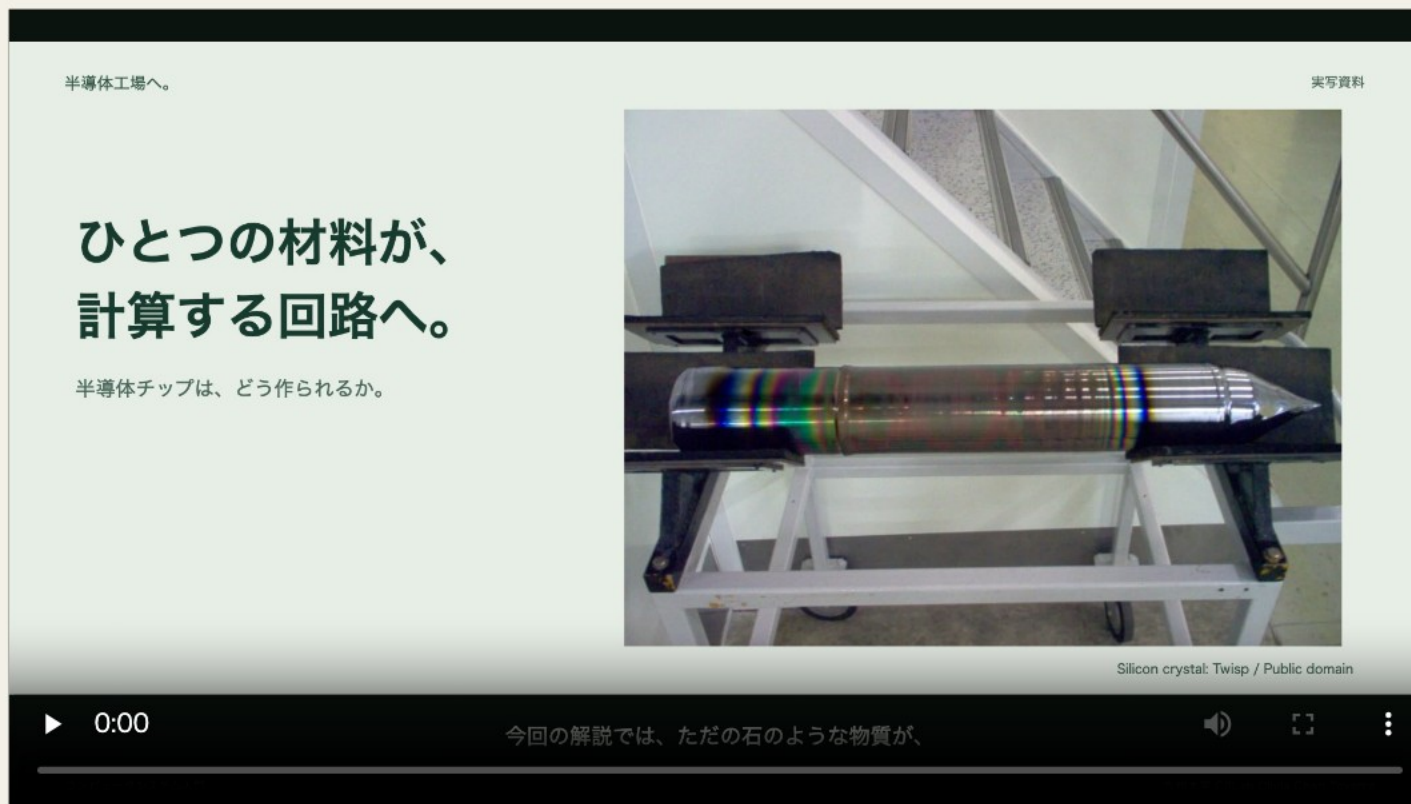
16 頁

→ **第 5 部** 半導体チップの製造

導入 p.01-04 / まとめ・次回 p.63-64

# 半導体製造の全体像

見る前の問い 加工する場所は？ 層の接続は？ 外部への接続は？



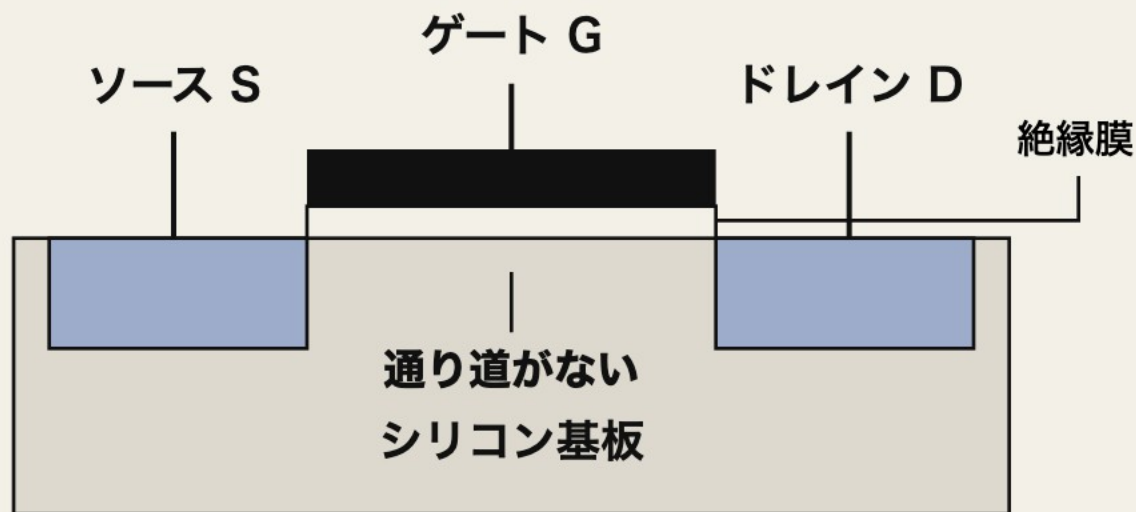
製造工程を映像で見る 9:15

[チャプター付き動画ページ](#) [🔗](#)

# シリコンで電流を制御する

ゲートの電圧で、二つの端子の間を**通す・止める**

OFF ON



nMOS の模式断面

## 止める

ゲート電圧が低いと、ソースとドレインを結ぶ通り道ができない

**S-D 間の電流を抑える**

この小さなスイッチを組み合わせ、**論理ゲートと記憶回路**を作る

基本動作を示す簡略図。形・寸法は実物と異なる。OFF でも微小な漏れ電流はある。

# 単結晶を育て、薄く切る



溶かす

単結晶を育てる

薄く切る

MATERIAL

材料

WAFER

ウェハ

LAYERS

層を作る

DIE

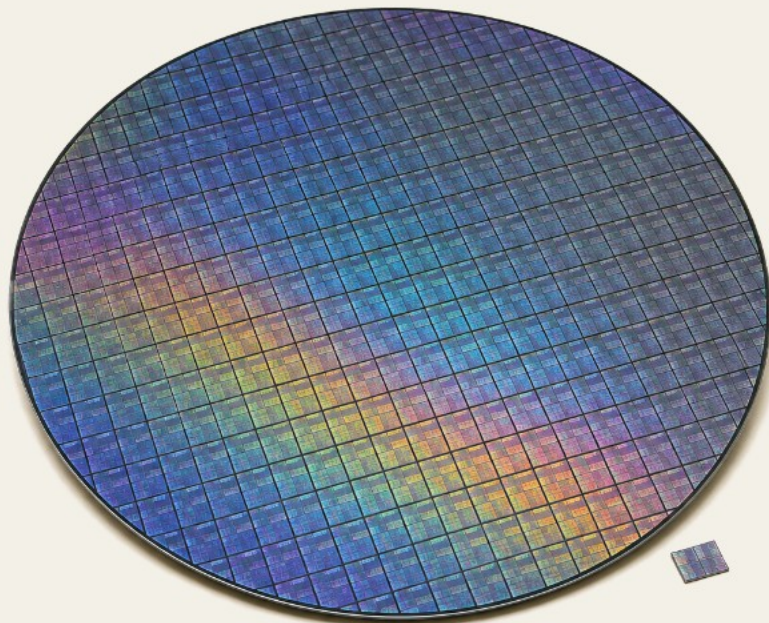
切り分ける

PACKAGE

外部につなぐ

結晶をそろえ、表面を磨く。微細な回路を作るための土台を用意する。

# 一枚のウェハに、同じ回路を並べる



WAFER / 一枚の円板

同じ回路を反復

DIE / 切り分ける単位

MATERIAL

材料

WAFER

ウェハ

LAYERS

層を作る

DIE

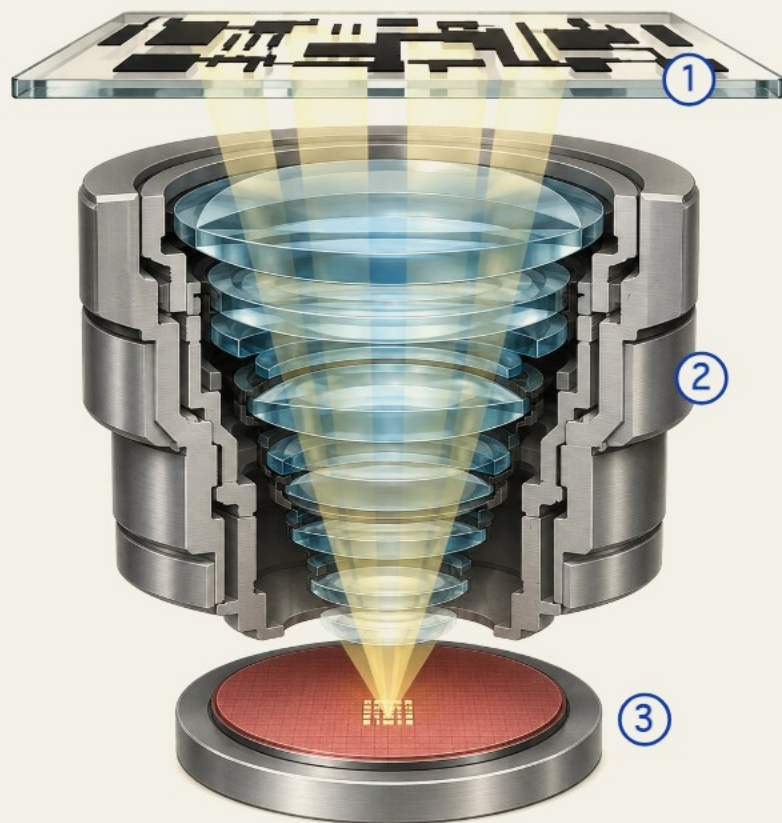
切り分ける

PACKAGE

外部につなぐ

同じ工程で多数のダイを加工し、一度に多くのチップを作る。

# 光で微細な模様を写す



透過型 DUV の原理を示す生成図（寸法・光路は模式的）

## 1 マスク

作りたい模様の原版

光を通す部分と遮る部分で模様を作る

## 2 縮小投影光学系

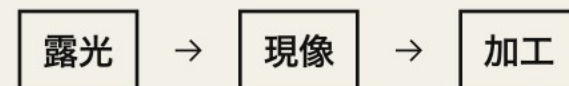
模様を小さく写す

レンズで感光材の表面にピントを合わせる

## 3 感光材（レジスト）

光に反応する薄い膜

現像すると、加工する場所を選ぶ模様が残る



光で直接削るのではなく

**加工する場所を指定する**

# 材料を加え、形と性質を変える

1 膜をつける

2 一部を削る

3 性質を変える

4 平らにする

加工前

まだ膜がない



下地

加工後

新しい薄膜



下地

工程の役割

**表面に材料を加える**

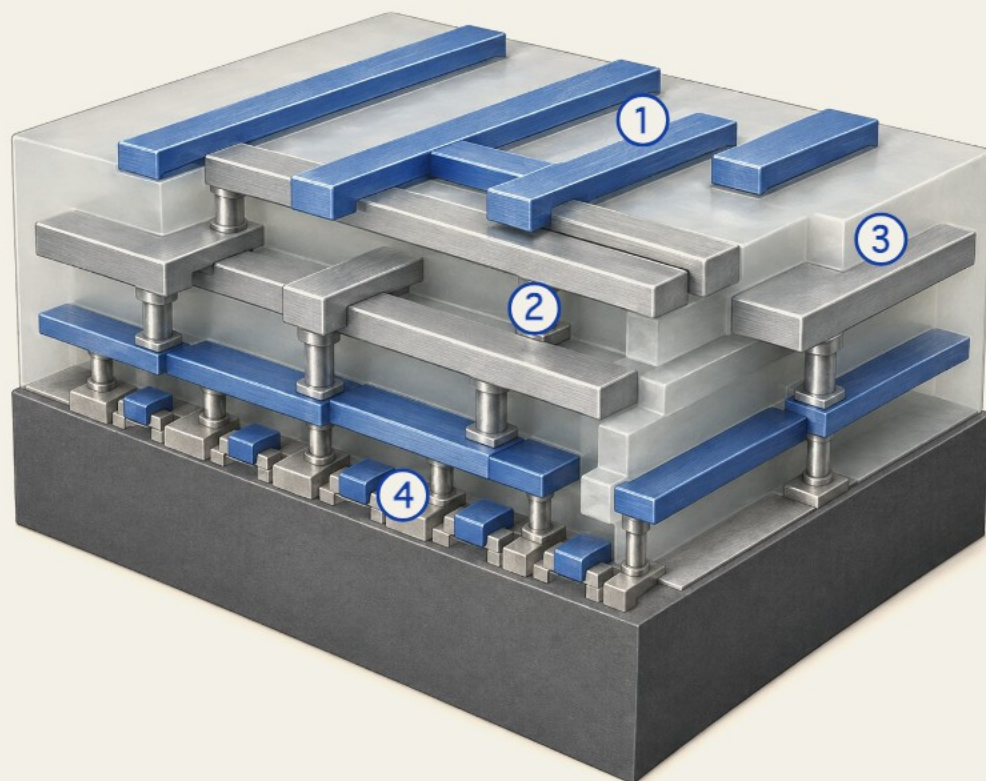
導体や絶縁体など、次の加工に必要な薄膜を作る

**薄膜が増える**

リソグラフィで**場所を選び**、材料の形や性質を変える工程を繰り返す

各操作の役割を比べる模式図。四つを一回ずつ行えば完成する、という工程順ではない。

# 配線を立体的につなぐ



多層配線の構造を示す生成図（寸法・材料色は模式的）

## 1 金属配線

複数の高さを使って、多くの部品を結ぶ

## 2 ビア（層間の接点）

つなぎたい場所にだけ垂直の接続を作る

## 3 絶縁材料

配線の間を埋め、電気が流れる場所を分ける

## 4 トランジスタ

シリコン表面の素子と、上の配線を接続する

### 各層の位置もそろえる



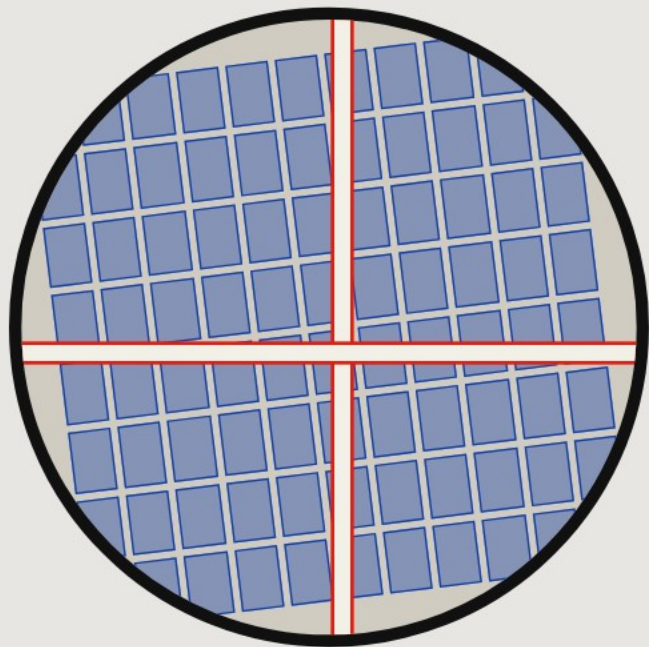
一致：つながる



ずれ：つながらない

模様を変え、位置を合わせて加工を反復

# ウェハをチップごとに切り分ける



MATERIAL  
材料

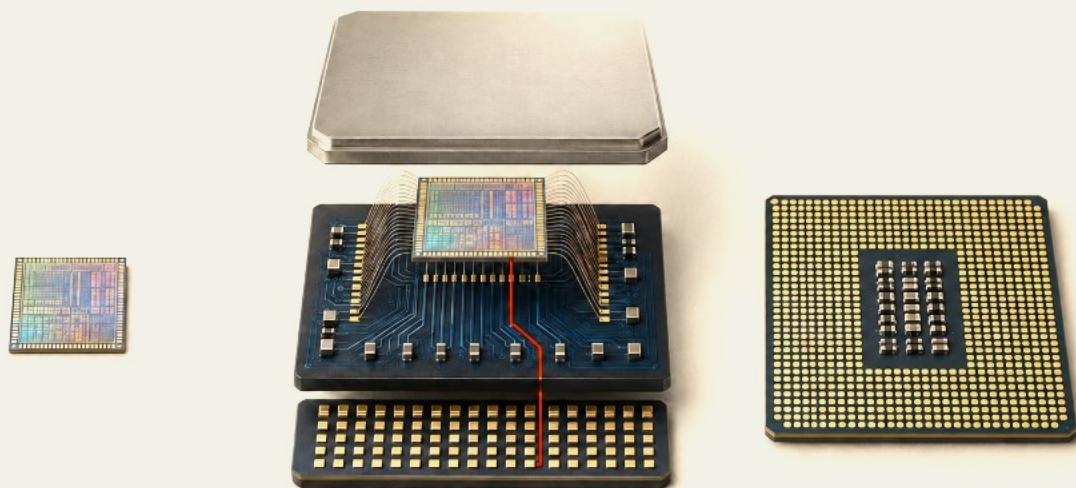
WAFER  
ウェハ

LAYERS  
層を作る

DIE  
切り分ける

PACKAGE  
外部につなぐ

# パッケージは外部との接点



裸のダイ

接続・保護・放熱

外部回路につなぐ接点

MATERIAL

材料

WAFER

ウェハ

LAYERS

層を作る

DIE

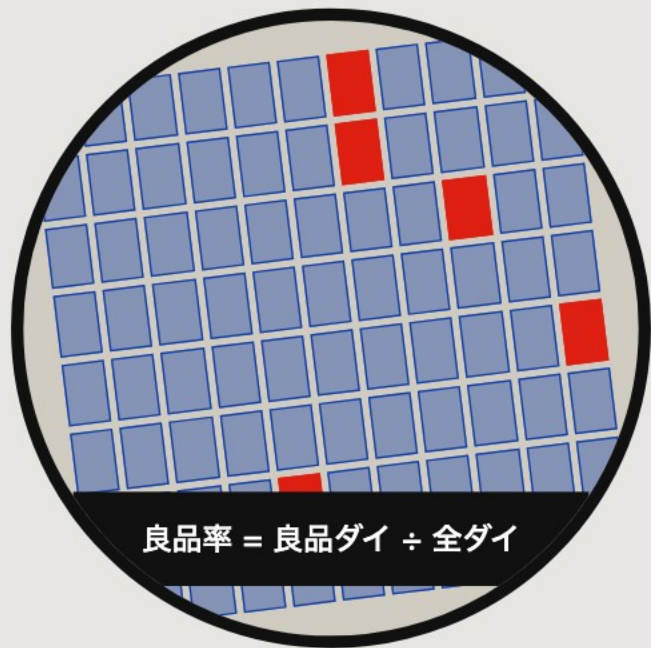
切り分ける

PACKAGE

外部につなぐ

小さなダイを、電力・信号・熱が出入りできる形にする。

# すべてのダイが良品とは限らない



MATERIAL  
材料

WAFER  
ウェハ

LAYERS  
層を作る

DIE  
切り分ける

PACKAGE  
外部につなぐ

# 半導体は、小さく集積してきた

1947

## トランジスタ

一つの小さな素子で  
電流を制御する

1958-59

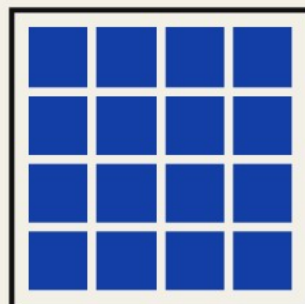
## 集積回路 (IC)

素子と接続を  
一つの半導体上に作る

1971

## マイクロプロセッサ

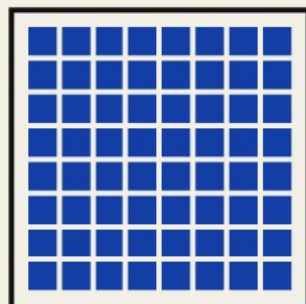
CPUの機能を  
一つのチップに集める



同じ面積に 16 個



一辺を 1/2  
面積は 1/4



同じ面積に 64 個

## 小さくして、より多く集める

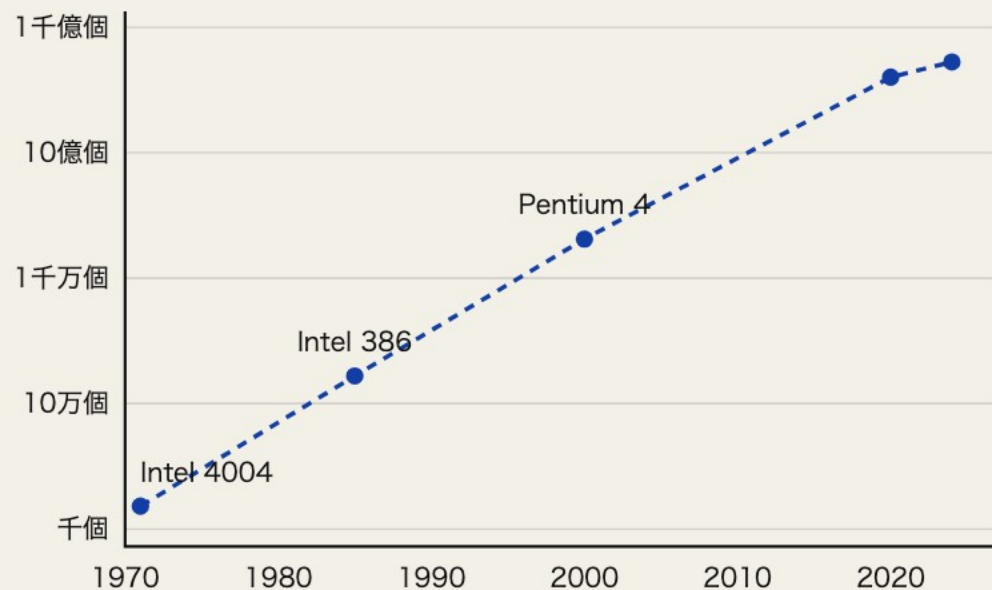
素子と配線の寸法・間隔を縮める  
露光技術、材料、構造を改良する

## 同じ面積に置ける回路が増える

四角は占有面積の模式例 実際の設計は配線なども必要

歴史：Computer History Museum / Intel 「3 nm」など現在の名称は技術世代を表し、すべての寸法が3 nmという意味ではない

# トランジスタ数の増加



縦軸は対数目盛：同じ高さの差が、同じ倍率を表す

1971 Intel 4004 約 2,300 個

1985 Intel 386 27.5 万個

2000 Pentium 4 4,200 万個

2020 Apple M1 160 億個

2024 Apple M4 280 億個

CPU単体とSoCを含む代表例  
チップ面積・機能は同一ではない

## ムーアの法則

集積する素子数が**約2年で倍増**するという経験的な見通し

1965年の予測を1975年に修正 物理法則でも、性能の倍増保証でもない

出典：Intel (4004 / 386 / Pentium 4)、Apple (M1 / M4) 点を結ぶ線は目安であり、密度や微細化だけの比較ではない

# 小さく、増やすだけでは足りない

数を増やせても、**全部を速く・安く・同時に動かせる**とは限らない



## 電流を止めにくい

### 小さくする限界

素子が短く、膜が薄くなると  
漏れ電流やばらつきが問題になる

電流を制御する  
形や材料を変える



## 熱を逃がしにくい

### 詰め込む・重ねる限界

多くの素子を動かすほど発熱  
立体化すると内部の放熱も難しい

電力・冷却を含めて  
配置と実装を考える



## 作ってつなぐのが難しい

### 配線と製造の限界

配線の遅延、加工の精度  
良品率と製造費が制約になる

巨大な一枚だけに頼らず  
小さなチップを組み合わせる

「微細化が止まった」ではなく、**微細化だけでは足りない**

# トランジスタの形を変える

狙い：小さくても電流を制御する

## ゲートで、通り道を囲む

平面型 → FinFET → GAA

ゲートが制御する面を増やす



平面型

FinFET

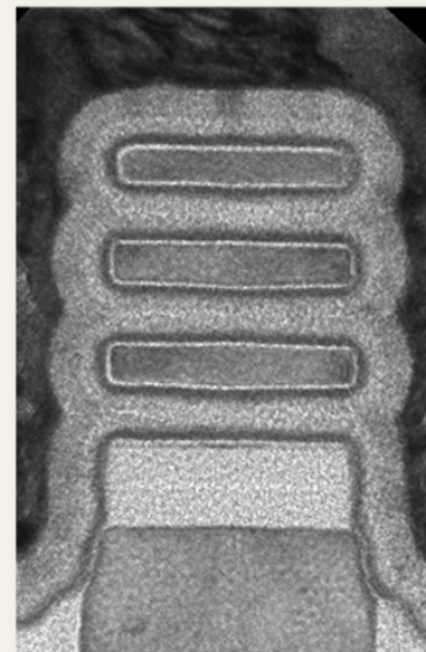
GAA

■ ゲート ■ 電流の通り道 制御する面の模式図

## GAA = Gate-All-Around

薄い通り道を全周から囲む構造

素子構造の改良は実用化と研究が並行して進む ここでの積層はトランジスタ内部であり、次ページのチップ積層とは異なる



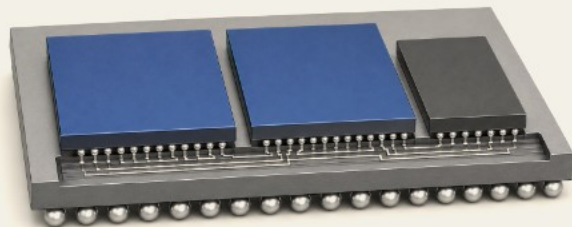
実際の断面を電子顕微鏡で見る  
横長の細い通り道が積層されている

[IBM Research / 2021年の研究発表](#)

# チップの組み合わせ方を変える

横に組み合わせる

チップレット / 2.5D実装



機能ごとに小さなチップを作り  
近い距離で接続する

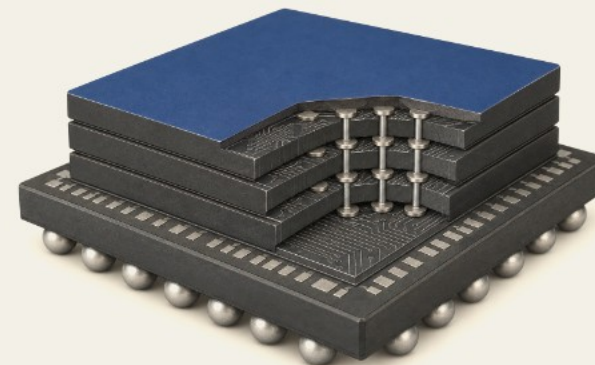
狙い：工程の使い分け・再利用・良品率

新しい課題もある：**放熱・接続の精度・検査・組立コスト**

生成模式図 実用化と研究が進む例：Intel EMIB / Foveros 3D積層は、一つのチップ内の多層配線とは別の技術

縦に重ねる

3D積層・パッケージ



複数のチップを積み重ね  
接続距離を短くする

狙い：高密度化・大きなデータ転送量

# 写真処理の速さを、回路までつなぐ

- |    |            |                                                        |
|----|------------|--------------------------------------------------------|
| 01 | 性能を測る      | 同じ写真・同じ処理で、時間と処理量を測る<br>第1部：CPI・IPC と CPU 時間           |
| 02 | 高速化の仕組みを知る | 多核・パイプラインで、仕事の進め方を変える<br>第2部：分担できない仕事や依存関係が限界になる       |
| 03 | 改善する場所を選ぶ  | RAM 待ちか、I/O 待ちか。内訳を見て対策を選ぶ<br>第3部：第5回の局所性を、読み出す回数で評価する |
| 04 | 回路を設計・検証する | 動作を記述し、正しさと遅延の条件を確かめる<br>第4部：第4回の DFF も、配置と配線で実物になる    |
| 05 | チップを実現する   | 設計を層に写し、加工・検査・接続を行う<br>第5部：配線・発熱・良品率など、実物の制約を考える       |

写真処理の速さを、**測る** → **工夫を知る** → **選ぶ** → **設計する** → **作る** で考える

# 次回：OSによる資源管理



同じパソコンで、複数の仕事が同時に進む

ハードウェアができた、その先

## 誰が、いつ CPUやメモリを使う？

ブラウザ、画像編集、ファイル同期  
同じ資源を取り合わずに使いたい

## OS オペレーティングシステム

資源を分け、使い方をそろえ  
プログラムどうしを保護する

---

このあとは演習 2 Verilogで記述 → 入力を変えて検証

[演習を開く](#) 