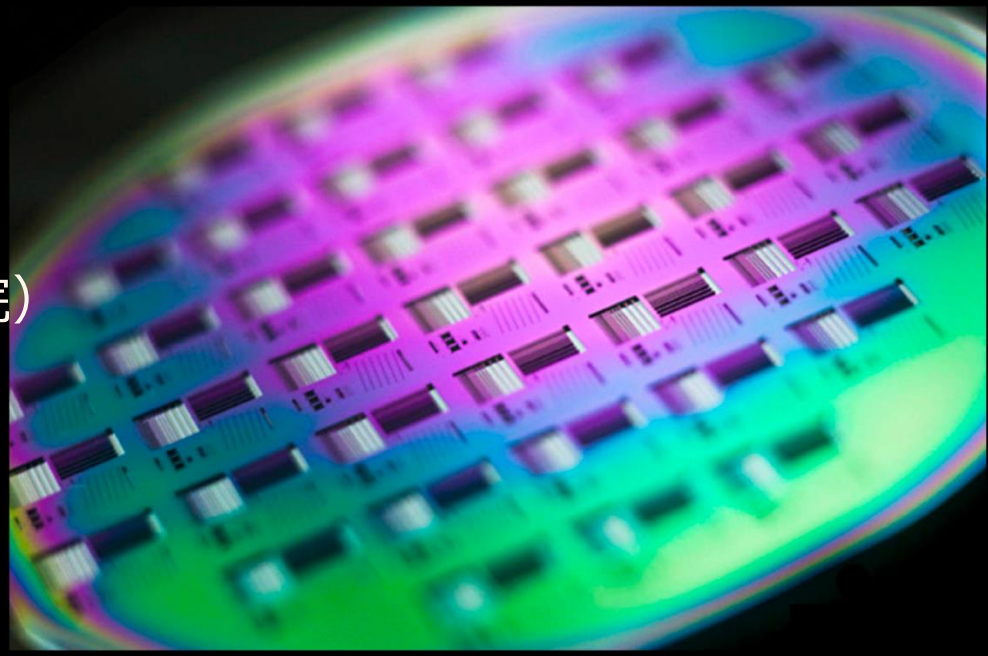
集中講義令和8年2月24日～26日

# コンピュータシステム入門

## Day 2: 性能評価とチップ設計

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

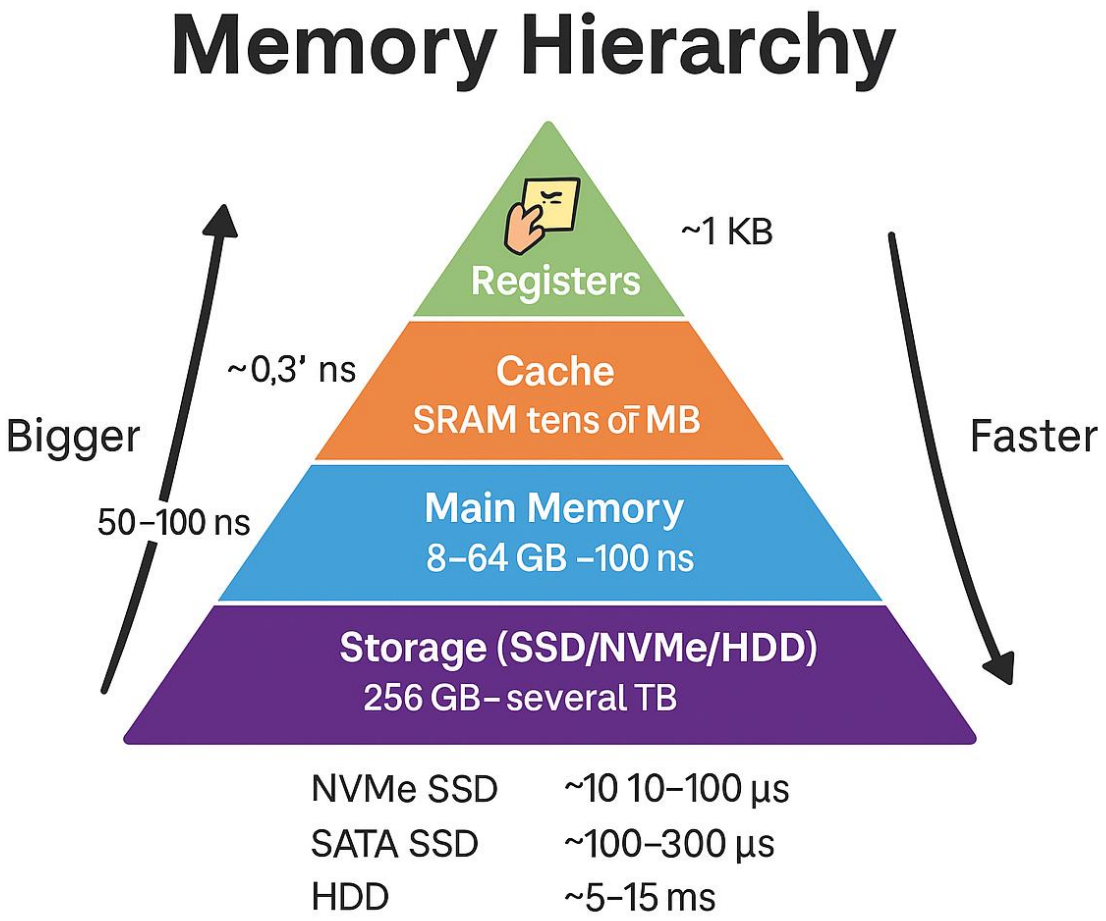
# 今日のスケジュール

| | 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

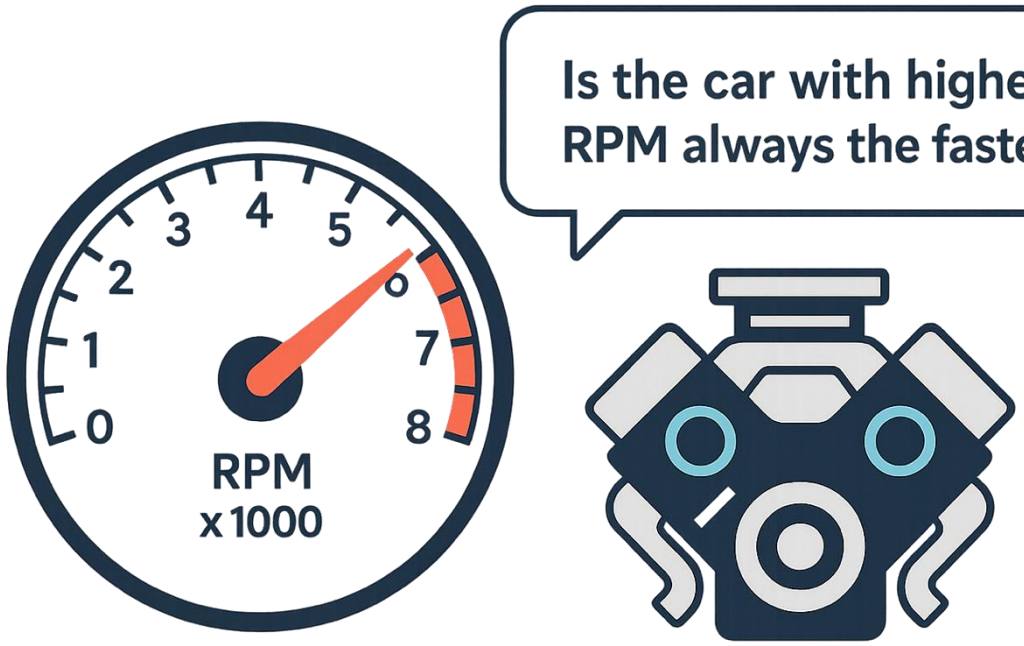
# モジュール６：性能評価とチップ設計

# 前回の復習：メモリの階層構造



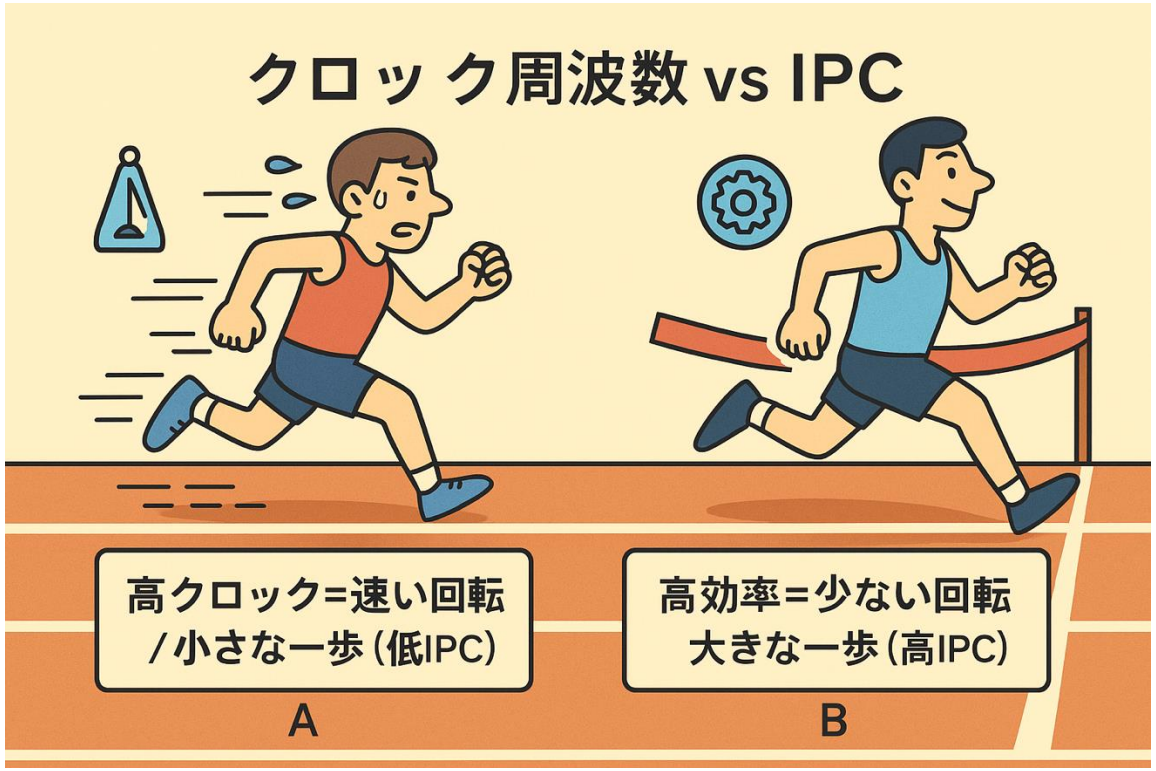

- CPUはキャッシュ等を使いボトルネックを緩和している。
- では、CPU自体の「速さ」は何で決まるのか？

# よくある誤解①：「速さ」の指標




- コンピュータの性能指標としてよく使われる「クロック周波数 (GHz)」
- これは、車のエンジン回転数(RPM)のようなもの
- しかし、回転数が高いだけで、その車が最速だとは限らない。なぜなら…

# よくある誤解②：「1回転あたりの仕事量」




- CPUも同じで、1クロック（1回転）でどれだけ多くの仕事ができるか（＝歩幅の大きさ）が重要
- この「賢さ」を示す指標が、後で学ぶ **IPC (Instructions Per Clock)**

# 「速さ」だけでは決まらない

クロック周波数

コア数

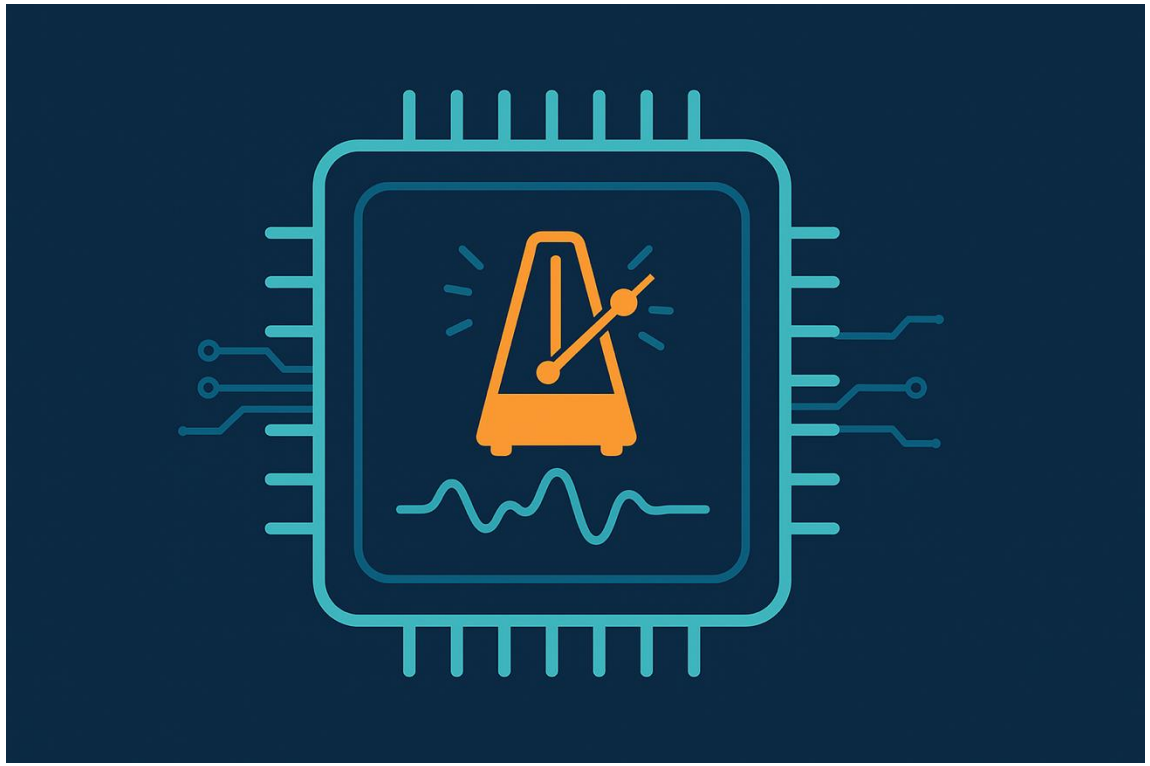
IPC(効率)

- コンピュータの真の性能は、
  - 動作リズムの速さ（クロック周波数）
  - 同時に働ける人数（コア数）
  - 1動作あたりの効率（IPC）

# 性能を決める3つの要素

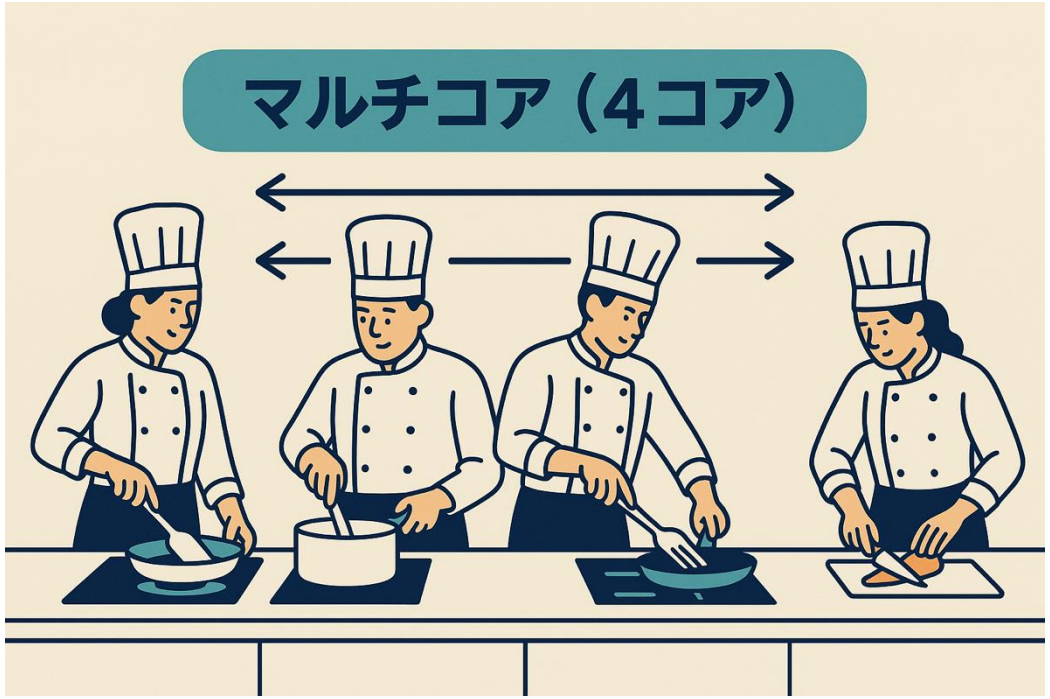
**性能 ＝ ①クロック周波数 × ②コア数 × ③IPC**

# 要素① クロック周波数



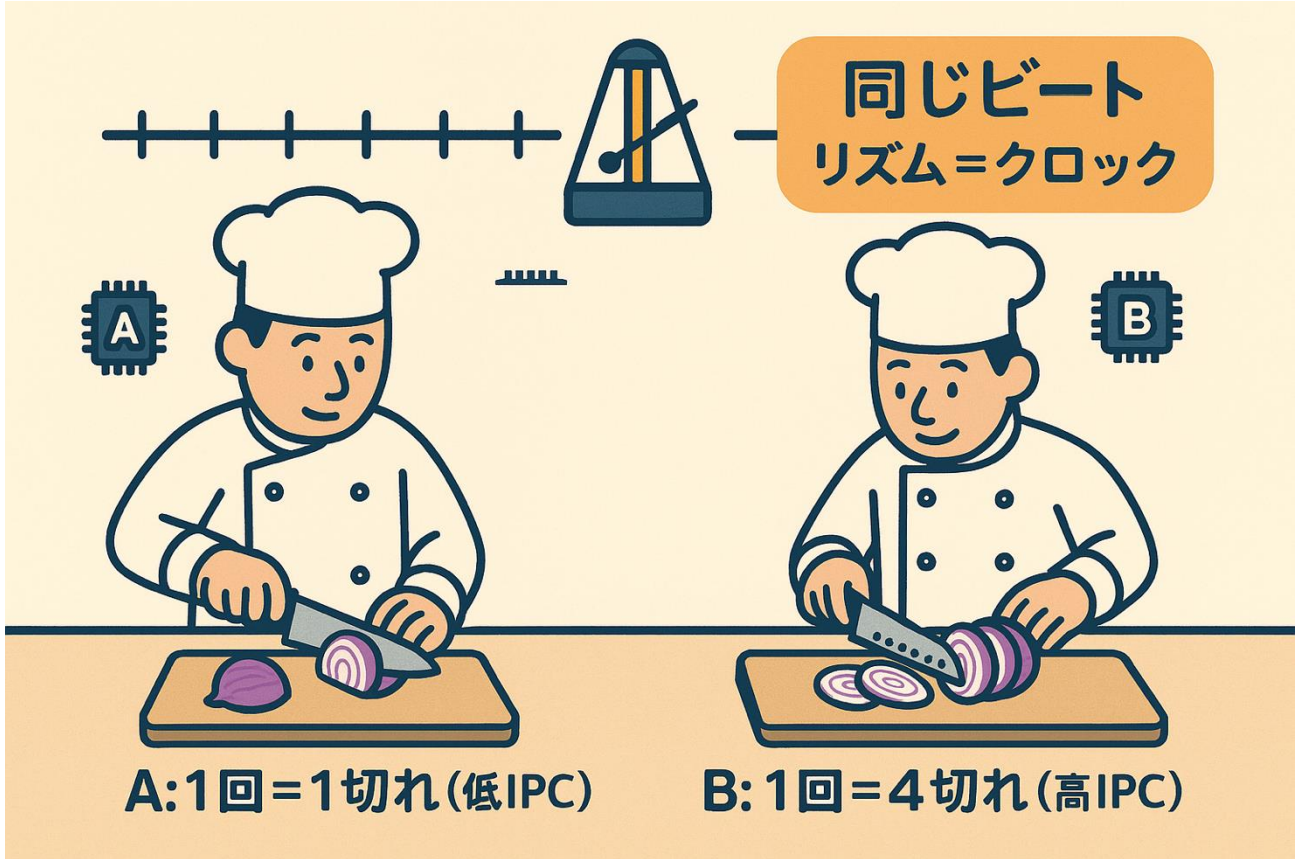
- クロック周波数：CPUの動作リズムの速さ
- 1GHz ＝ 1秒間に10億回

# 要素② コア数


- コア：CPUの「頭脳」そのもの
- 並列処理能力に関わる。

# 要素③ IPC (Instructions Per Clock)




- IPC：1クロックあたりに実行できる命令の数
- CPUの「賢さ」「効率の良さ」を示す

# 性能の方程式（まとめ）

**性能 ＝ ①クロック周波数 × ②コア数 × ③IPC**

AppleのMシリーズCPUが高性能なのは、この「賢さ(IPC)」が非常に高いから。

CONTENTS

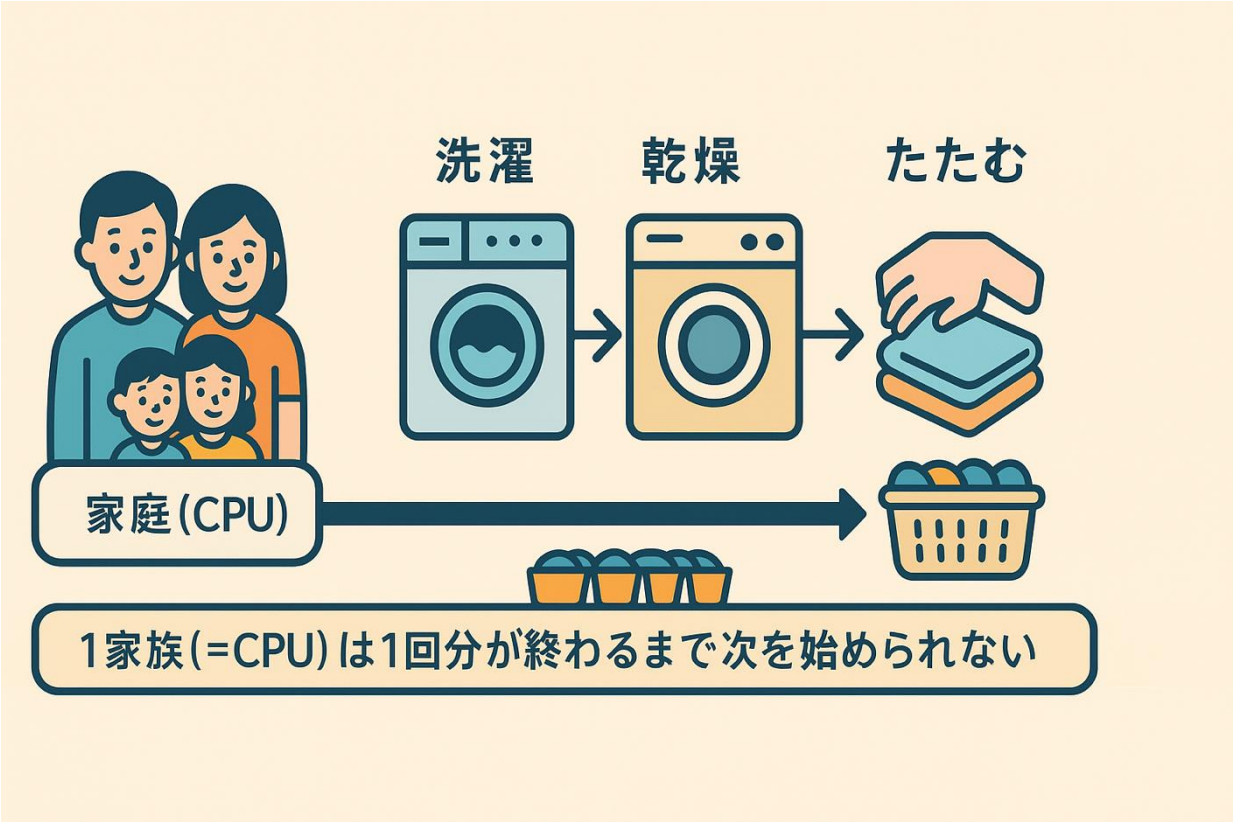
# モジュール６：性能評価とチップ設計

# IPCを高めるには？

CPUの「賢さ」の秘密は、命令をいかに効率よく、無駄なく処理するかの工夫にある。

その代表例が「パイプライン処理」

# もしパイプラインがなかったら…




- 1つの命令の全工程（フェッチ～格納）が終わるまで、次の命令を開始できない。
  CPUの多くの部分が「手待ち」状態になる。

# パイプライン処理の導入

## 同時進行（パイプライン）


- **パイプライン処理：命令をステージに分割し、並行処理する**
- ある命令が「デコード」段階にあるとき、次の命令は「フェッチ」段階に入る

# パイプライン処理の概念図

| CLK / Instr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |

# パイプライン処理の概念図

| Instr. \ CLK | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |

# パイプライン処理の概念図

| CLK / Instr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |

# パイプライン処理の概念図

| Instr. \ CLK | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | | | | |
| 5 | | | | | | | |

# パイプライン処理の概念図

| CLK / Instr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | IF | ID | EX | MEM |
| 5 | | | | | | | |

# パイプライン処理の概念図

| CLK / Instr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | IF | ID | EX | MEM |
| 5 | | | | | IF | ID | EX |

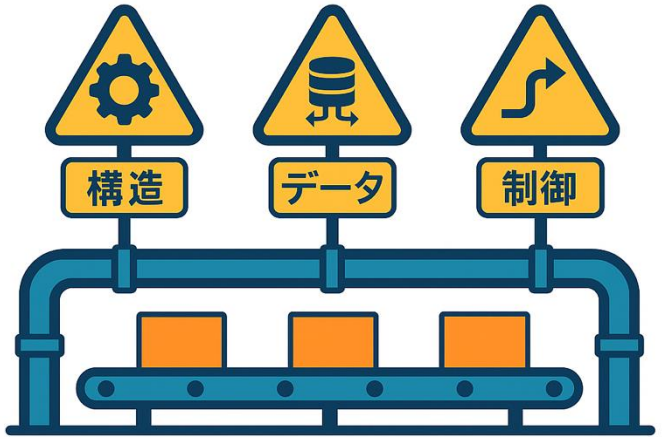
# パイプラインの課題：ハザード

流れが止まってしまう「ハザード（Hazard）」と呼ばれる3種類の問題が発生

- **構造ハザード (Structural Hazard):** 資源の取り合い
  - **原因:** ハードウェアの資源（メモリやALUなど）が一つしかないのに、複数の命令が同時に使おうとすると発生

- **データハザード (Data Hazard):** 計算結果の待ち合わせ
  - **原因:** 前の命令の計算結果を、次の命令がすぐに使いたい場合に発生

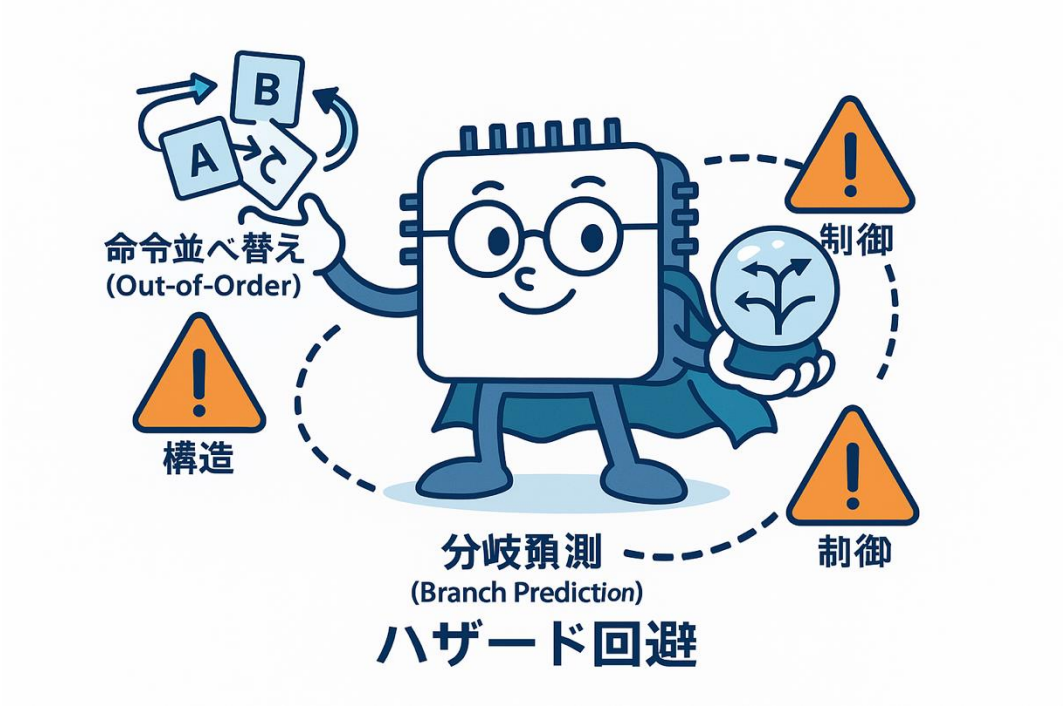
- **制御ハザード (Control Hazard):** 次の行き先が分からない
  - **原因:** if文のような条件分岐命令で、どちらの命令を実行すれば良いか、結果が分かるまで確定しない場合に発生



ハザード

# ハザードへの対策

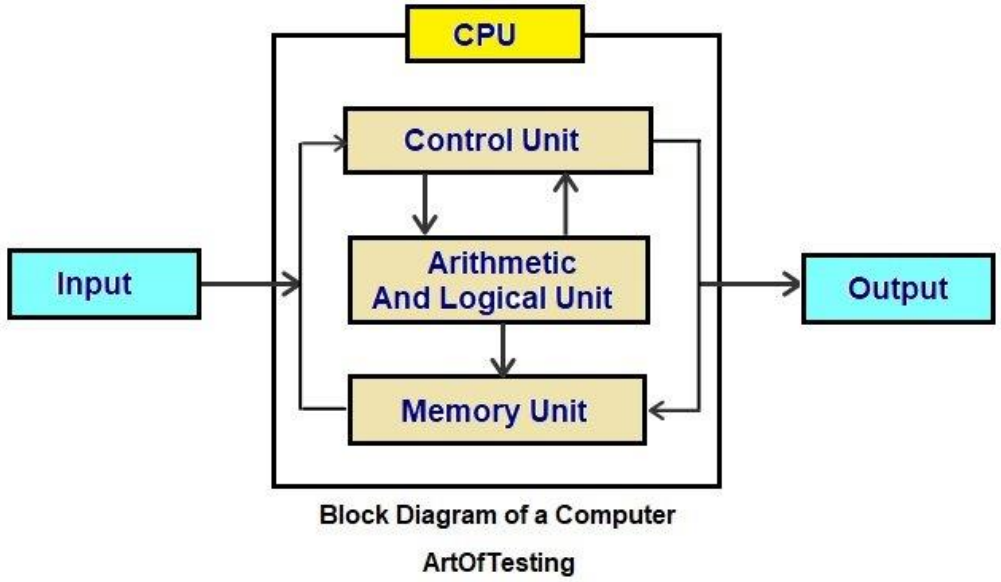
- 現代のCPUは、これらのハザードを解決するための様々な賢い技術を持っている
- **例:** 命令の実行順序を入れ替えたり（アウト・オブ・オーダー実行）、分岐の結果を予測したり（分岐予測）することで、パイプラインの停止を最小限に抑えている



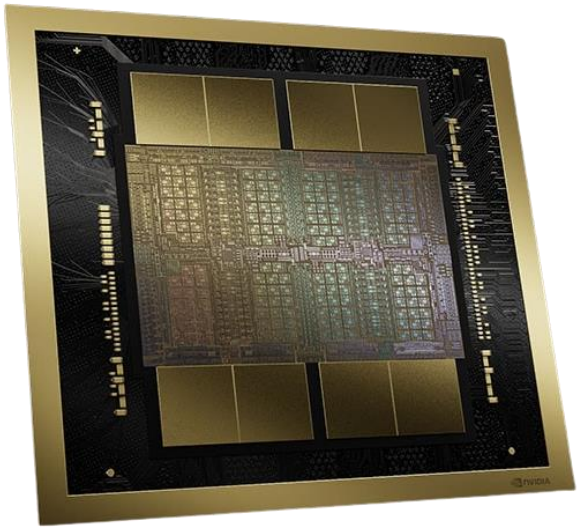
CONTENTS

# モジュール６：性能評価とチップ設計

# 設計から製造へ



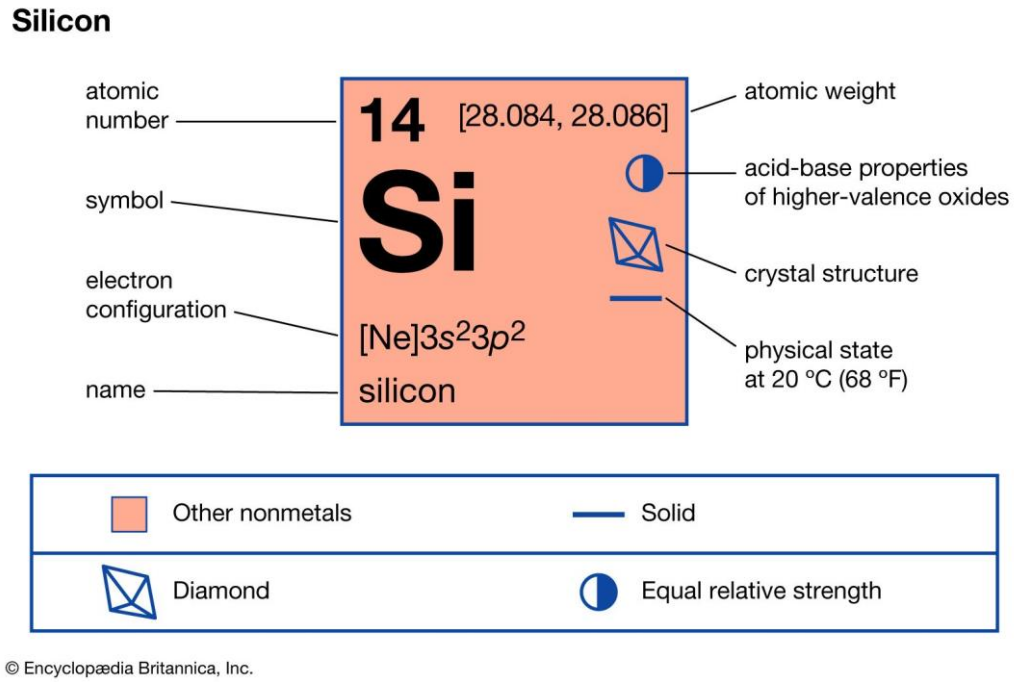

?



抽象的な設計図は、どうやってこの小さな物理的なチップになるのか？

# コンピュータは“石”から生まれた

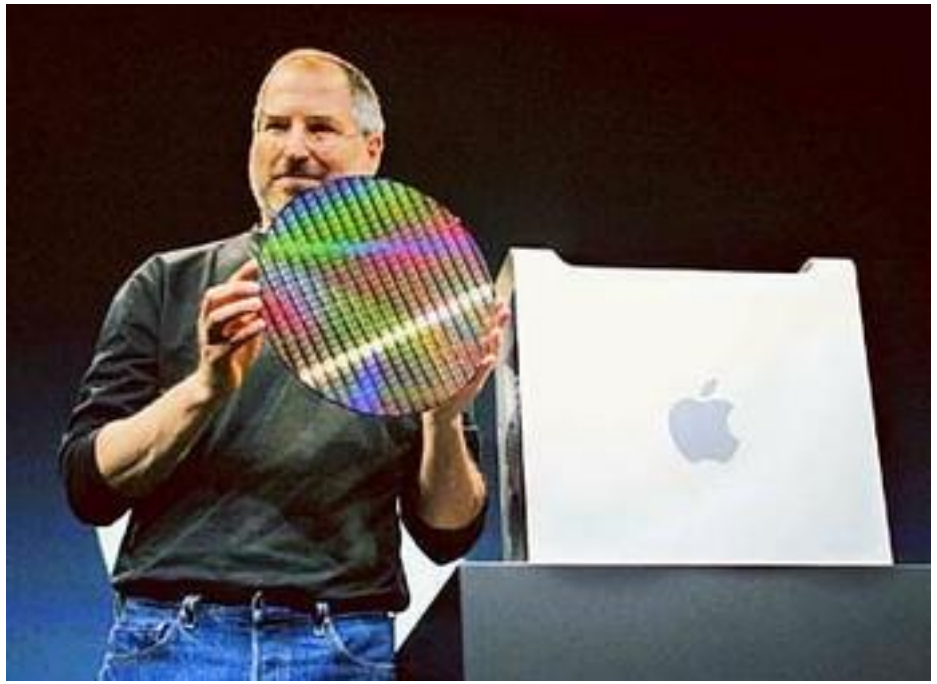
CPUの主原料は**ケイ素（シリコン）**




シリコンは石英（$SiO_2$）から精製されます。※シリコン（Si）とシリコーン（シリコン樹脂）は別物。

# シリコンインゴットとウェハー

https://www.mmtc.co.jp/en/products/silicon-s.html



Steve Jobs with a IBM PowerPC wafer (2003)

超高純度のシリコンの塊（インゴット）を作り、薄くスライスする。

# 回路の印刷技術「フォトリソグラフィ」

**抽象的な設計図は、どうやってこの小さな物理的なチップになるのか？**

- 光でパターン（回路）を焼き付ける技術
- Tシャツの版画に似ている



フォトレジスト＝感光材、マスク（先端はレチクル）、
露光＝紫外線（DUV/EUV）

# フォトリソグラフィの工程

1. レジスト（感光材）をスピンで薄く塗る（ソフトベーク）

2. マスクを重ねて位置合わせし、紫外線で露光する

3. 現像液で不要なレジストを溶かし、模様を出す

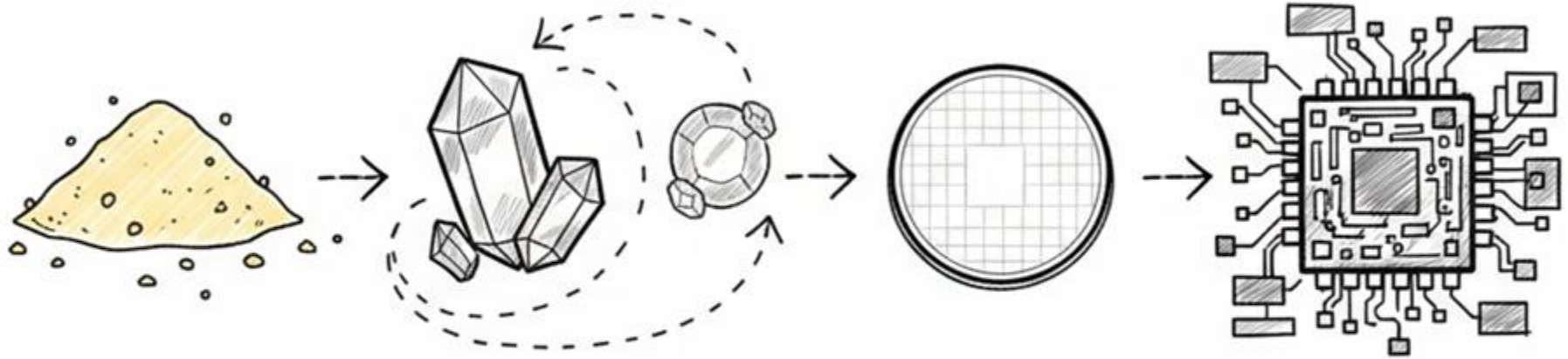
4. 薬品やプラズマで材料を削る（エッチング）。レジストを剥離し、次の層へ

この工程を何百回も繰り返し、立体的な構造を作り上げる。

# ウェハーからチップへ

ウェハーを切り分け（ダイシング）→電気的テスト→良品をパッケージに封入して完成

# 砂からスーパー頭脳へ：マイクロチップの作り方

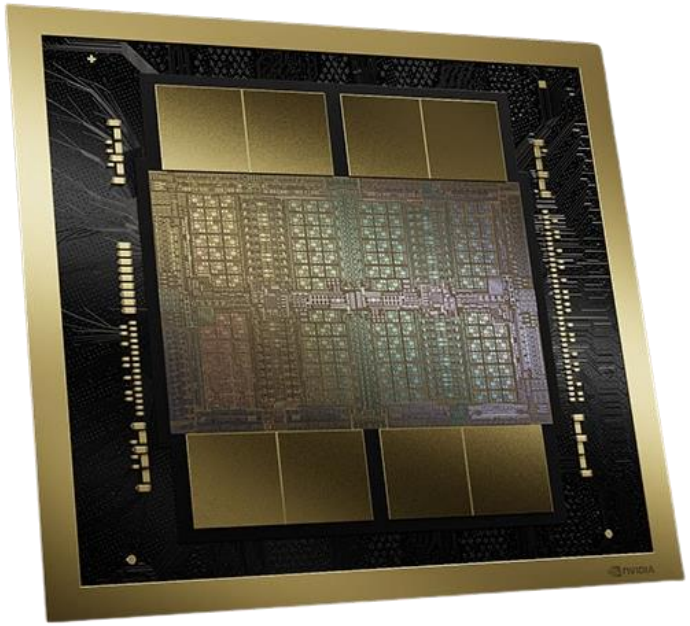


CONTENTS

# モジュール６：性能評価とチップ設計

# 複雑な回路をどう設計する？

チップは、論理ゲートをただ寄せ集めただけで出来上がるものではありません。



NVIDIA Blackwell GPU

**2080億個トランジスタ ≈520億個論理ゲート**

エンジニアが 1 秒に 1 個の論理ゲートを手作業で配置したとしても、すべてを置き終えるまでには丸々 1600 年かかる。

# ハードウェア記述言語 (HDL)

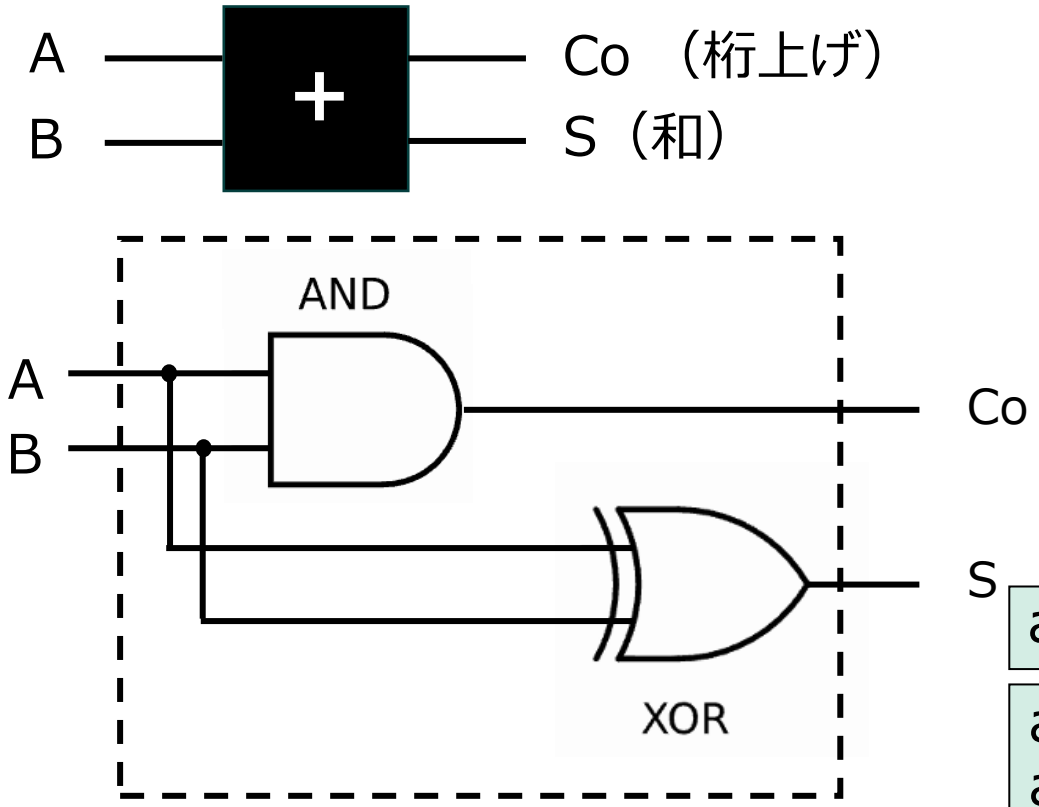
回路設計者も、ハードウェア記述言語 (HDL) という「言語」で回路の動作を記述する

**HDL (Hardware Description Language):**

- 回路の**構造**（どの部品がどう繋がっているか）や**動作**（どんな入力に対してどう振る舞うか）を、テキストベースの「言語」で記述するためのもの。
- これにより、人間はトランジスタレベルの詳細から解放され、より抽象的なレベルで設計に集中できる。
- **主なHDL:** Verilog HDL と VHDL の2つが業界標準として広く使われている。

HDLは回路を記述する言語、プログラミング言語ではない

# Verilog HDLの例




**半加算器　Half Adder**

```
/* 半加算器のmoduleの定義 */
module half_add(a, b, co, s);
   /* input port */
   input a, b;
   /* output port */
   output co, s;

   /* assign文 */
   assign co = a & b;
   assign s =  a ^ b;
endmodule
```

assign文で使用できる論理演算子

| | |
|---|---|
| a \| b　aとbのOR | a ~\| b　aとbのNOR |
| a & b　aとbのAND | a ~& b　aとbのNAND |
| a ^ b　aとbのXOR | a ~^ b　aとbのXNOR |
| ~a　aのNOT | |

# 電子自動化設計EDA

## 仕様設計

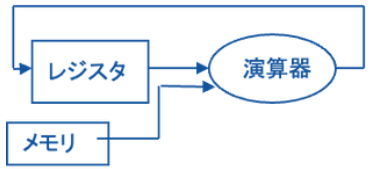
命令セット，レジスタセット
データ、アドレス、制御信号
パイプライン、割込み、キャッシュ

LD GR1, 100, GR7

ADD GR1, 200, GR7

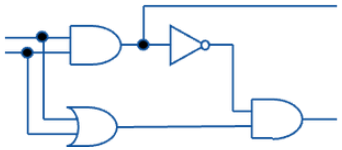
## 機能設計

組み合せ回路と
レジスタ



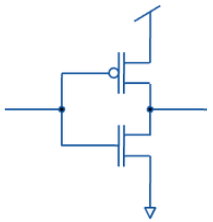

## 論理設計

AND, OR, NOT, FF



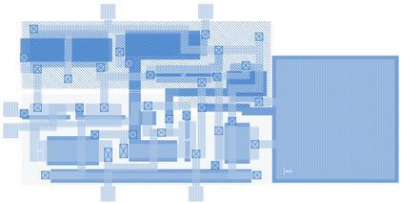
## 回路設計

トランジスタ



## レイアウト設計

チップに配置、配線

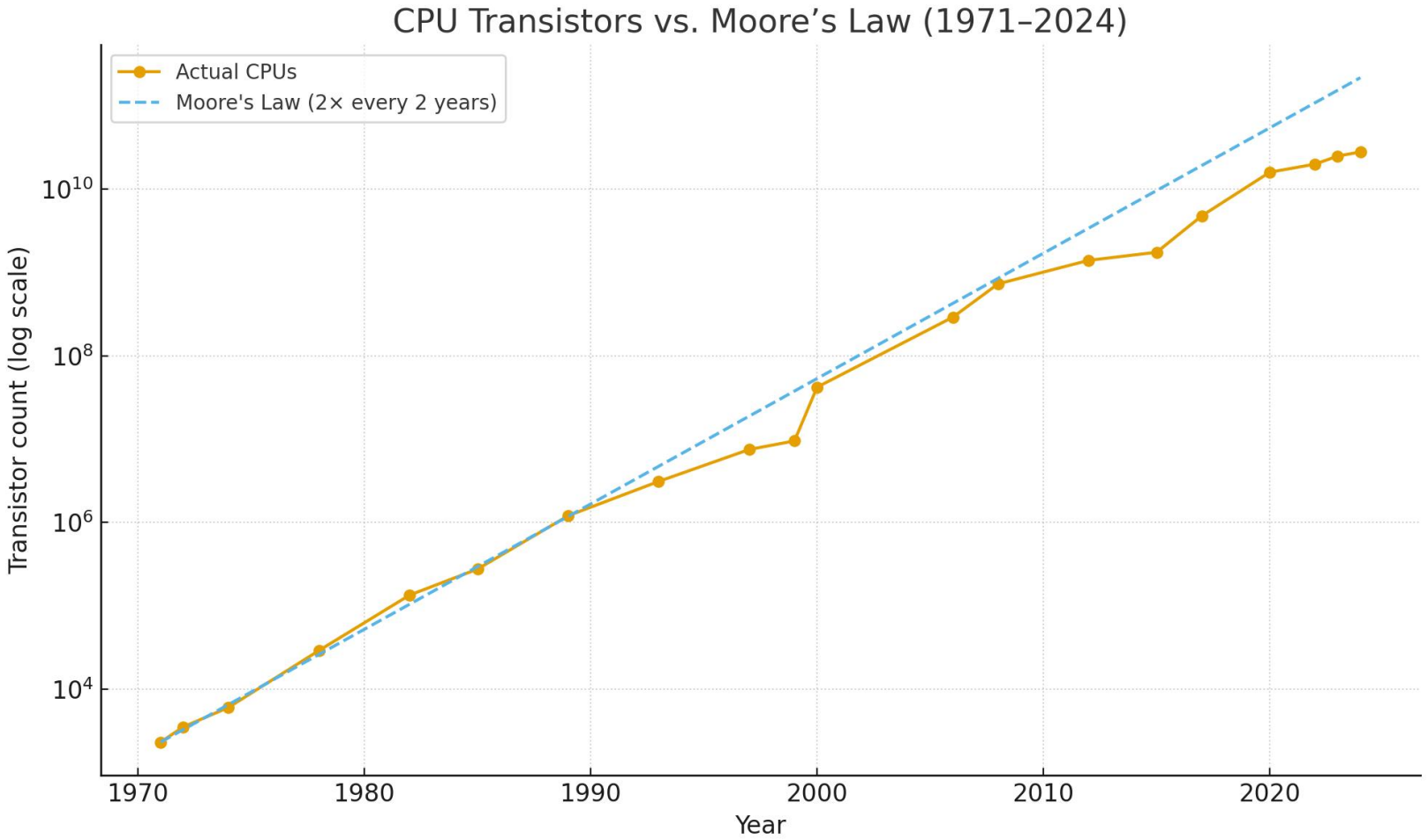


Design Entry → RTL → Netlist → Schematic → GDSII

Automatically generated during each step

Electronic Design Automation (EDA) Flow

# ムーアの法則




半導体チップのトランジスタ数は、約2年で2倍になる。この驚異的な進化が、IT社会を支えてきた。

# 微細化の限界と未来

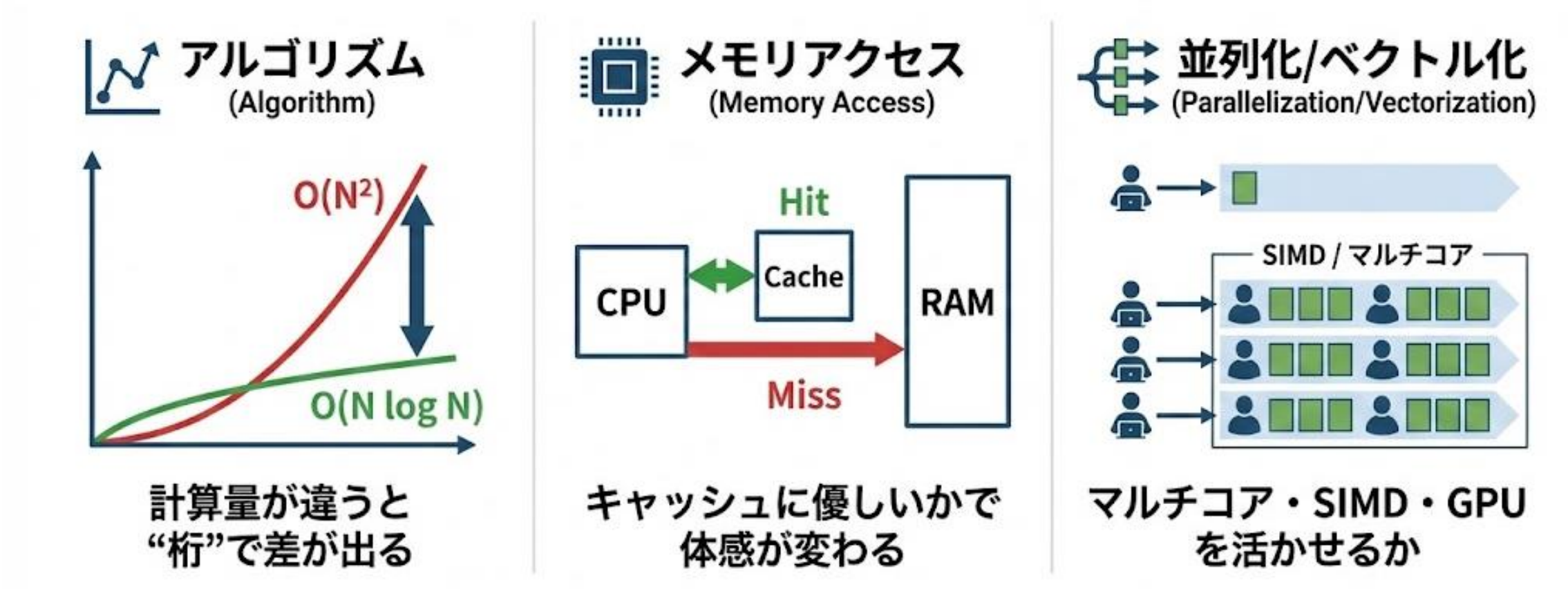
- **短チャネル効果・ばらつきの増大・配線RC遅延・発熱密度・露光の解像限界（EUV, レジストのゆらぎ）** など、限界要因は複合的
- トランジスタが原子レベルに近づくと、電子がバリア（ゲート絶縁膜）を確率的にすり抜ける「量子トンネル効果」が増え、漏れ電流が増大
- これ以上の性能向上には、3D積層・チップレット・新材料・新アーキテクチャなど、新しいアプローチが必要

# コンピュータは「層」でできている（抽象化の階層）

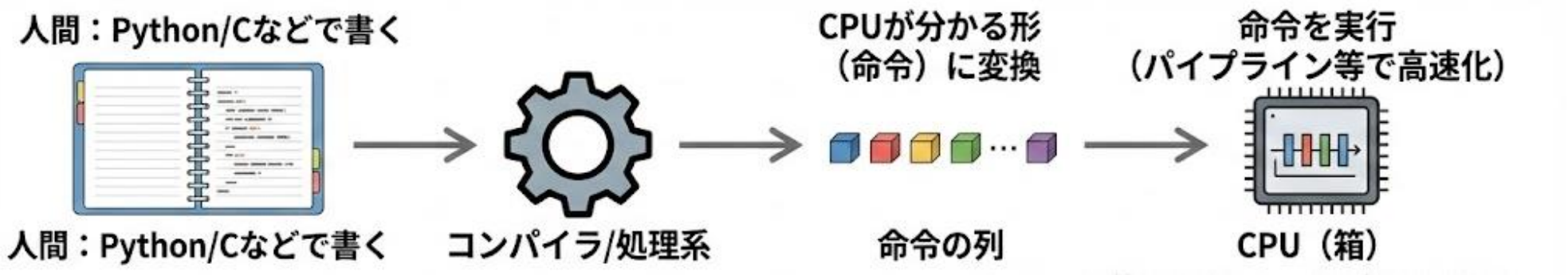
- 私たちは普段、**アプリ**を操作している
- でも実際には、下に行くほど「物理」に近い層が積み重なっている
- 今日（Day2）は下側（回路・設計・製造）を見た
- 次回（Day3）は、**OS**という「層」がどうやってハードを管理するかを見る

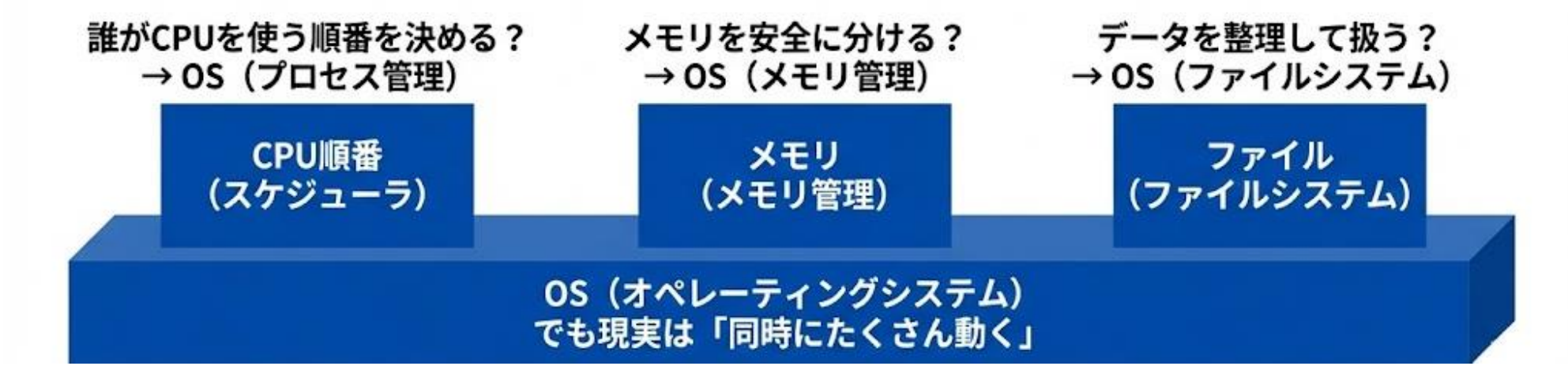
# 同じCPUでも速さが変わる：ソフトウェアの影響

- **性能 = ハードだけで決まらない（ソフトの“使い方”で変わる）**



# アプリは最終的に「命令」になってCPUで動く



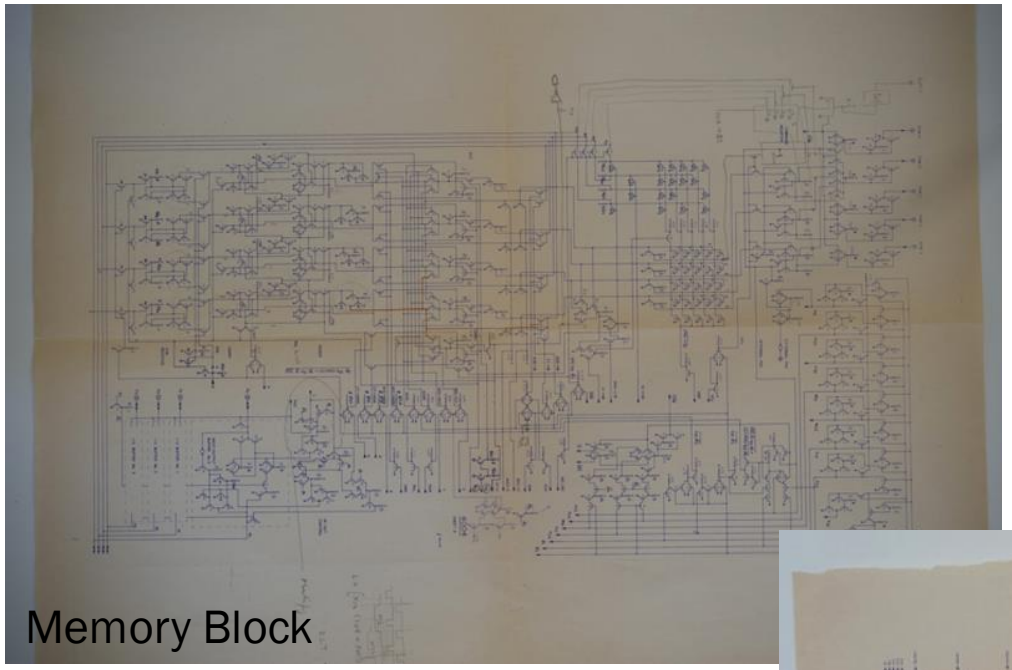



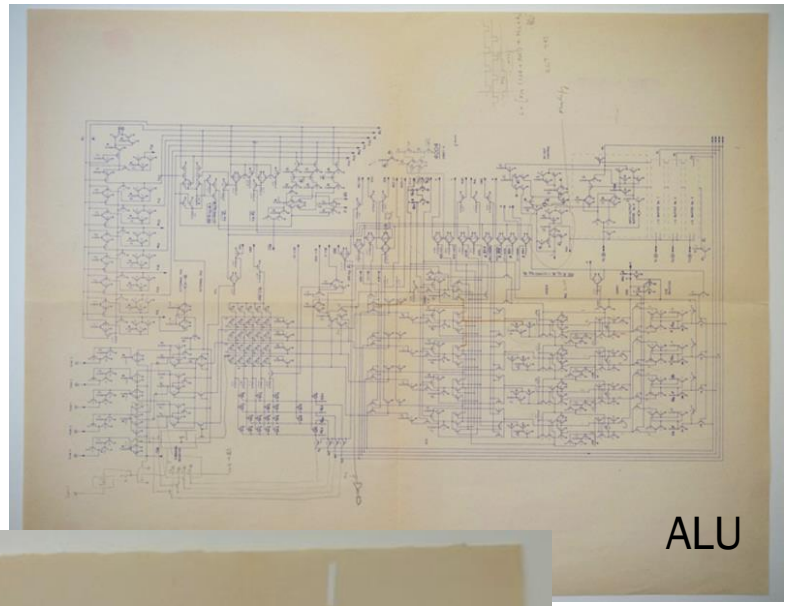
# 次回予告

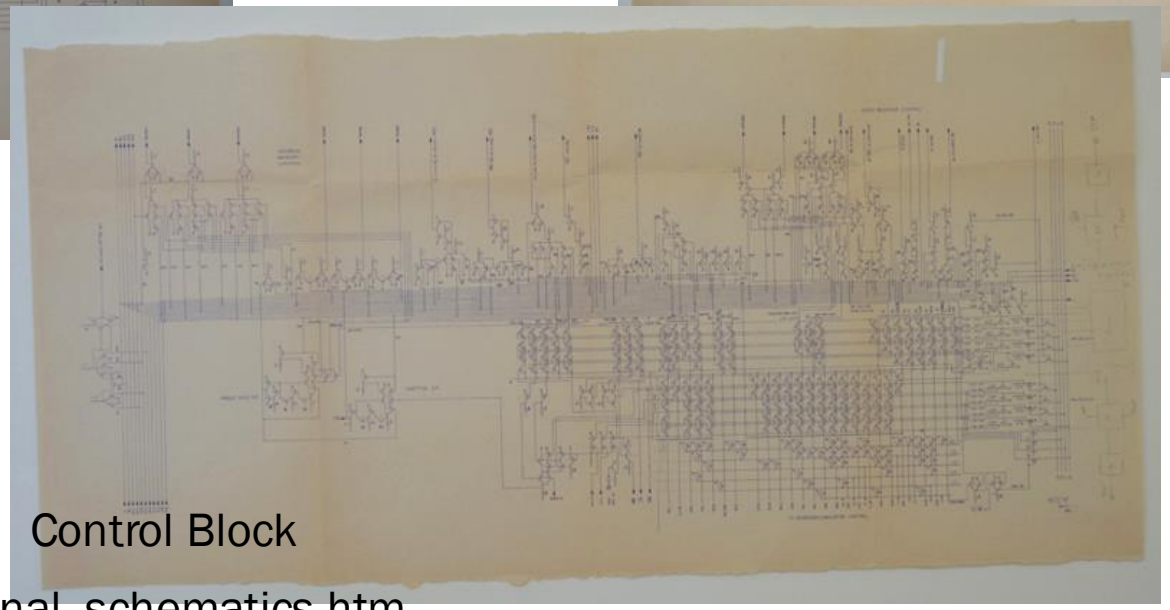
## 次回：オペレーティングシステム（OS）

今日学んだ設計図の評価指標と記述方法を踏まえ、
次回はそれらを実際に動作させるための基盤技術で
あるオペレーティングシステム（OS）に移る。

# Intel 4004の設計図

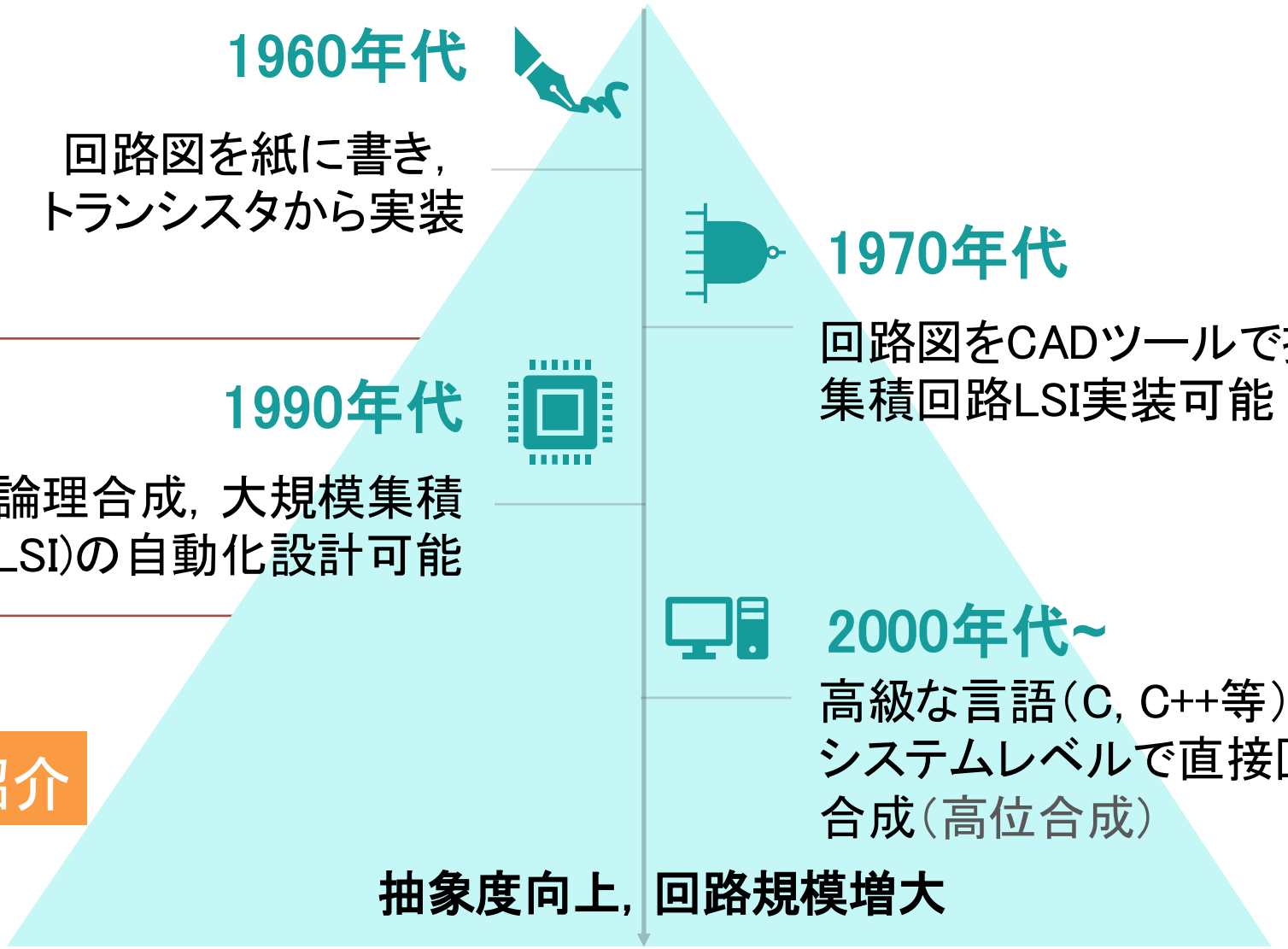

Memory Block


ALU

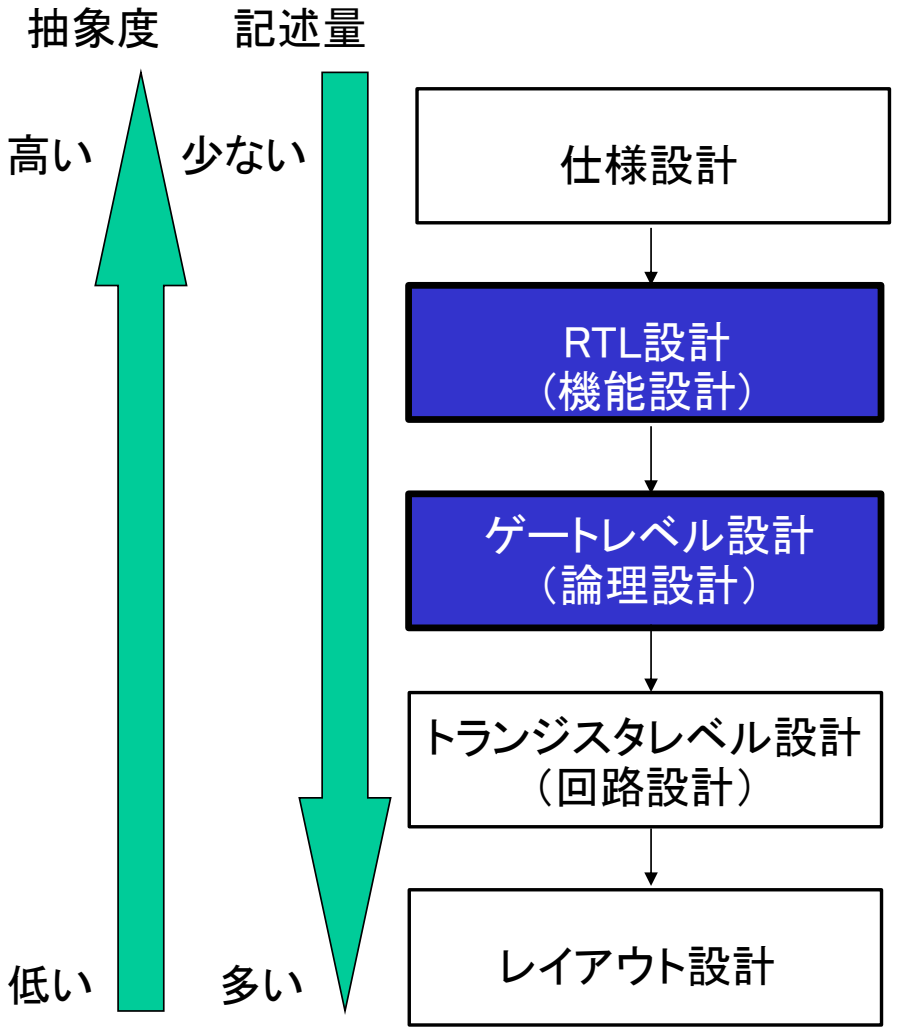

Control Block

http://www.intel4004.com/4004_original_schematics.htm

# 設計方法の推移


1960年代
回路図を紙に書き，
トランシスタから実装
1970年代
回路図をCADツールで描く，
集積回路LSI実装可能
1990年代
HDLより論理合成，大規模集積
回路(VLSI)の自動化設計可能
2000年代~
高級な言語（C，C++等）より、
システムレベルで直接回路を
合成（高位合成）
ここを紹介
抽象度向上，回路規模増大

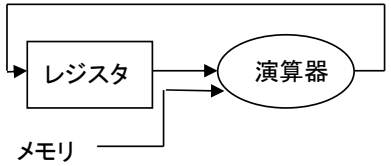
# VLSI 設計段階と表現



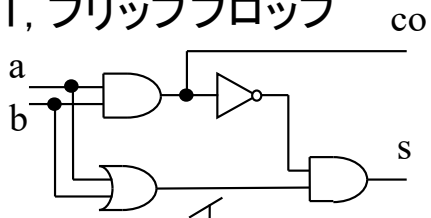

命令セット，レジスタセット
データ、アドレス、制御信号
パイプライン、割込み、キャッシュ

```
LD   GR1, 100, GR7
ADD GR1, 200, GR7
```

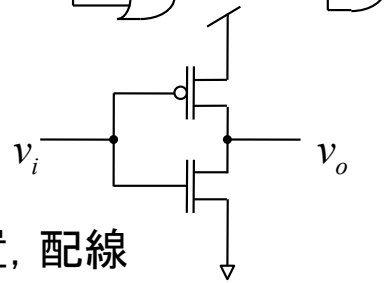
組合せ回路とレジスタ




AND, OR, NOT, フリップフロップ




トランジスタ




チップに配置，配線

HDL:目的とする回路の機能を人間のわかりやすいようにテキストで記述して，その記述から自動的に回路記述を生成する。

# HDL設計の利点

- 回路記述の簡単化
  - 例）加算器: z = a + b
- シミュレーション早期開始が可能
  - RTL設計のままシミュレーションが可能
    （RTL; Register Transfer Level）
- ゲートレベル設計の自動化
  - 論理合成ツールが論理回路をFPGAなどに自動的に合成
    （FPGA; Field Programmable Gate Array）

**HDLの種類**

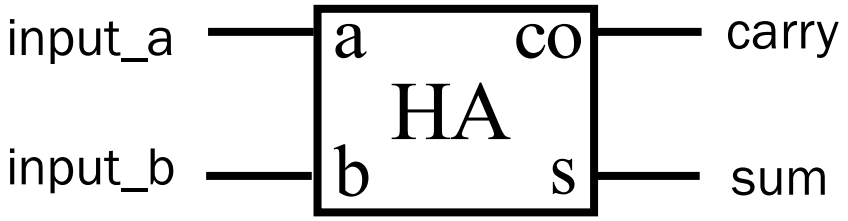
- Verilog-HDL：Cadence社の論理シミュレータ用言語から派生
- VHDL：回路仕様を書くことを目的に、標準化

**本講義で学習するHDL:**

Verilog HDL

# Half Adder（半加算器）

| a | b | co | s |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |




**入力 a, b を加算**

**加算結果を s に出力**

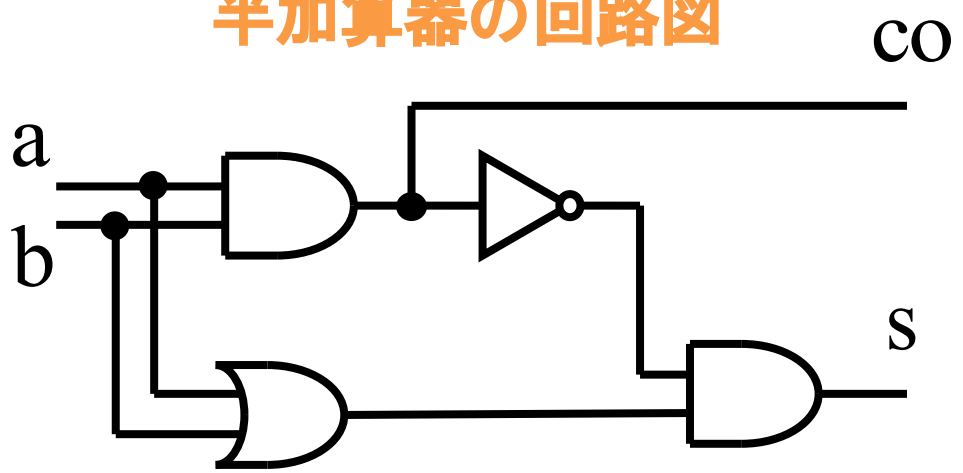
**桁上げ（キャリー）を co に出力**

## 半加算器の論理式

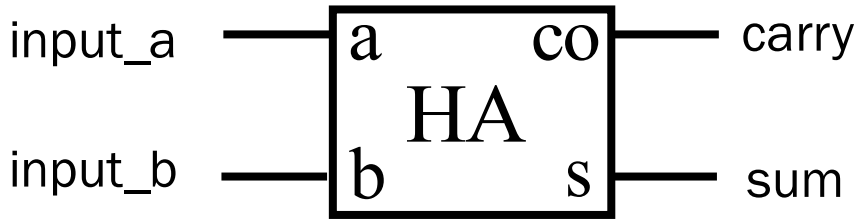
$$s = \overline{(a \cdot b)} \cdot (a + b)$$

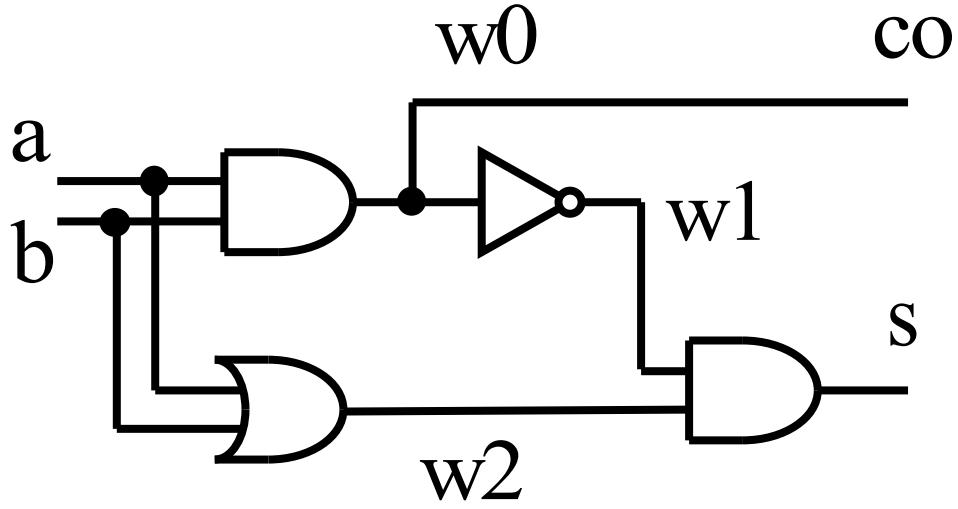
$$co = a \cdot b$$

## 半加算器の回路図



# Half Adder（半加算器）




```
/* 半加算器のmoduleの定義 */
module half_add(a, b, co, s);
    /* input port */
    input a, b;
    /* output port */
    output co, s;
    /* wire */
    wire w0, w1, w2;

    /* assign文 */
    assign w0 = a & b;
    assign w1 = ~w0;
    assign w2 = a | b;
    assign s = w1 & w2;
    assign co = w0;
endmodule
```

assign文で使用できる論理演算子

| | |
|---|---|
| a \| b | aとbのOR |
| a & b | aとbのAND |
| a ^ b | aとbのXOR |
| ~a | aのNOT |

| | |
|---|---|
| a ~\| b | aとbのNOR |
| a ~& b | aとbのNAND |
| a ~^ b | aとbのXNOR |

$$s = \overline{(a \cdot b)} \cdot (a + b)$$

$$co = a \cdot b$$

# moduleの定義

moduleの定義

任意の名前を指定

moduleのportを全て列挙

```
module モジュール名（ポート名，ポート名，...）；

  input ポート名，ポート名，... ；

  output ポート名，ポート名，... ；

  wire  ワイヤ名，ワイヤ名，... ；

  assign ワイヤ名 = 式 ；
  assign 出力ポート名 = 式 ；

endmodule
```

input portを全て列挙

output portを全て列挙

module内部のwireを全て列挙

右辺の式の信号を左辺のwireまたはoutput portに接続

moduleの定義の終了

モジュール名、ポート名、ワイヤ名として以下は使用できない

数字から始まる文字列、文法で定義済みの文字列

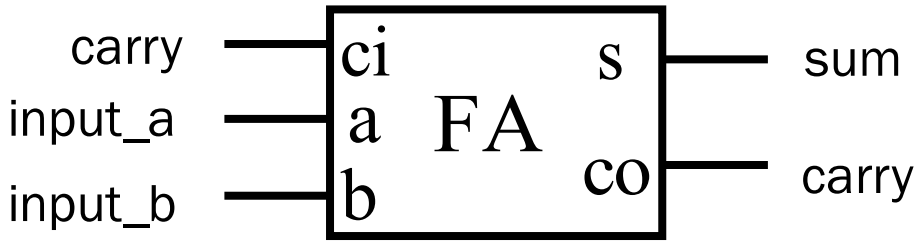
演算記号などを含む文字列

# Full Adder（全加算器）

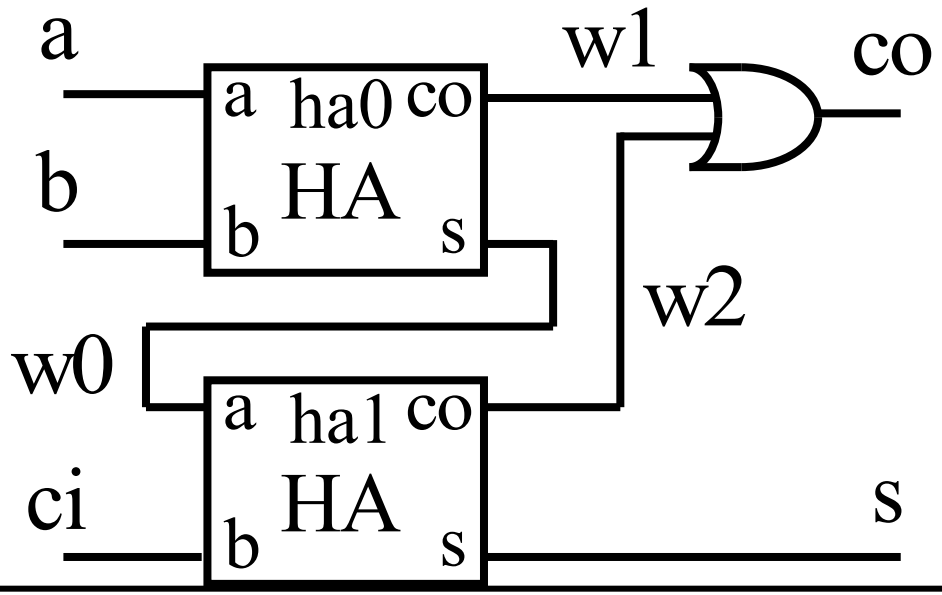
| a | b | ci | co | s |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

co　ci

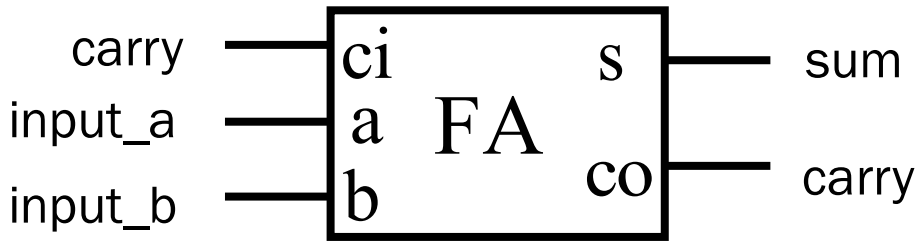
```
      1  1
a  ...0  0  1...
b +...0  1  1...
s ------------
     ...1  0  0...
```

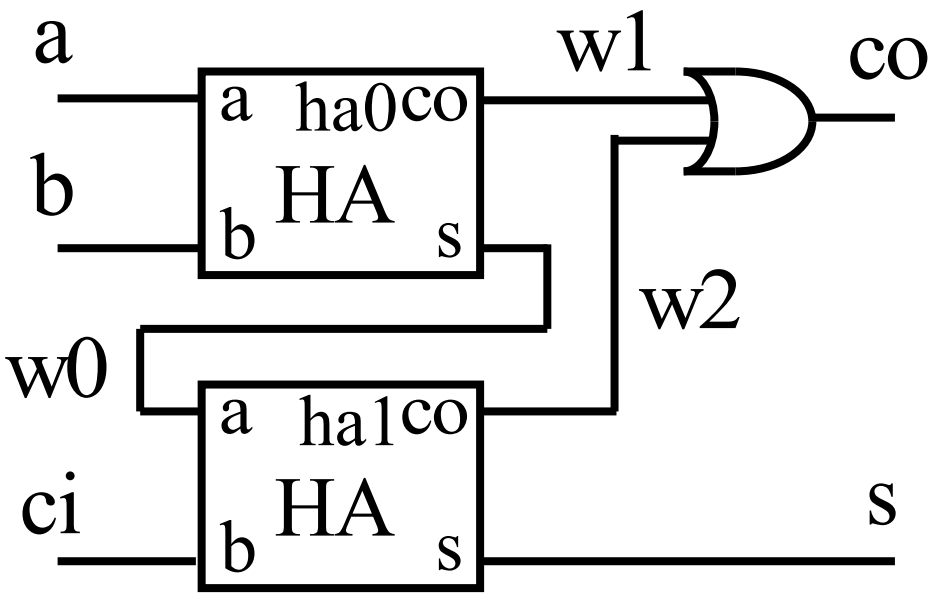



２つのHAとORゲート：
入力a, bを加算して
下位からのキャリーciを加算
どちらかの加算で桁上げしたら
上位へのキャリーcoを出力



# Full Adder（全加算器）




```
/* 全加算器のmoduleの定義 */
module full_add(a, b, ci, co, s);
    /* input port */
    input a, b, ci;
    /* output port */
    output co, s;
    /* wire */
    wire w0, w1, w2;

    /* instance文 */
    half_add ha0(.a(a),   .b(b),
                 .co(w1), .s(w0));
                          w0を介して接
      half_add ha1(.a(w0),続.b(ci),
                 .co(w2), .s(s));
    /* assign文 */
    assign co = w1 | w2;
endmodule
```

# instance文（下位moduleの使用）

```
module モジュール名 （ポート名，ポート名，...）；

  input ポート名，ポート名，... ；
  output ポート名，ポート名，... ；
  wire  ワイヤ名，ワイヤ名，... ；

  assign ワイヤ名 = 式 ；
  assign 出力ポート名 = 式 ；

  モジュール名 インスタンス名( .ポート名(  ワイヤ名  )，... ) ；

endmodule
```

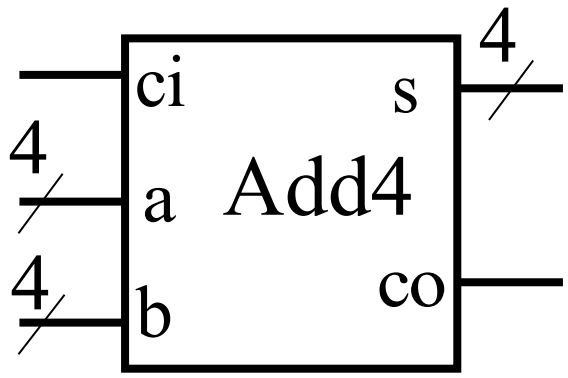
使用する下位moduleの
instanceの名前

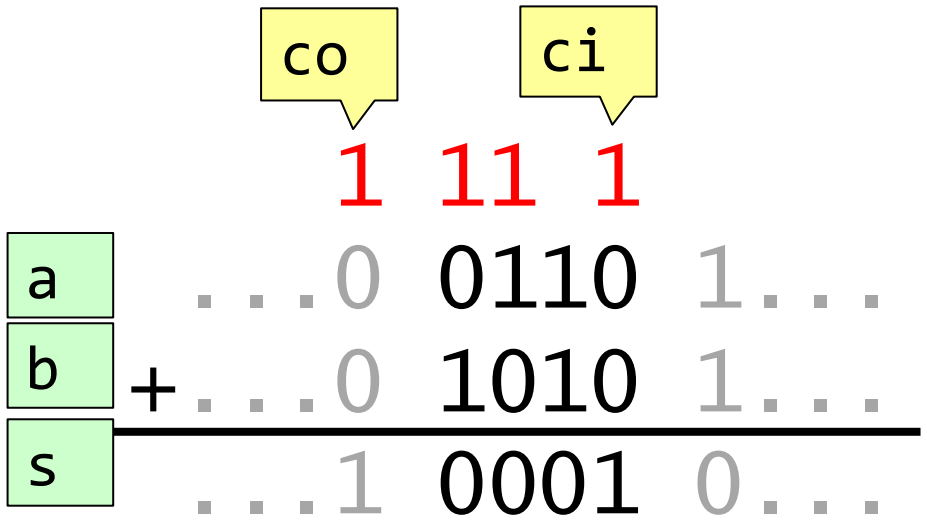
下位moduleで
宣言したportの名前

定義済みの
下位moduleの名前

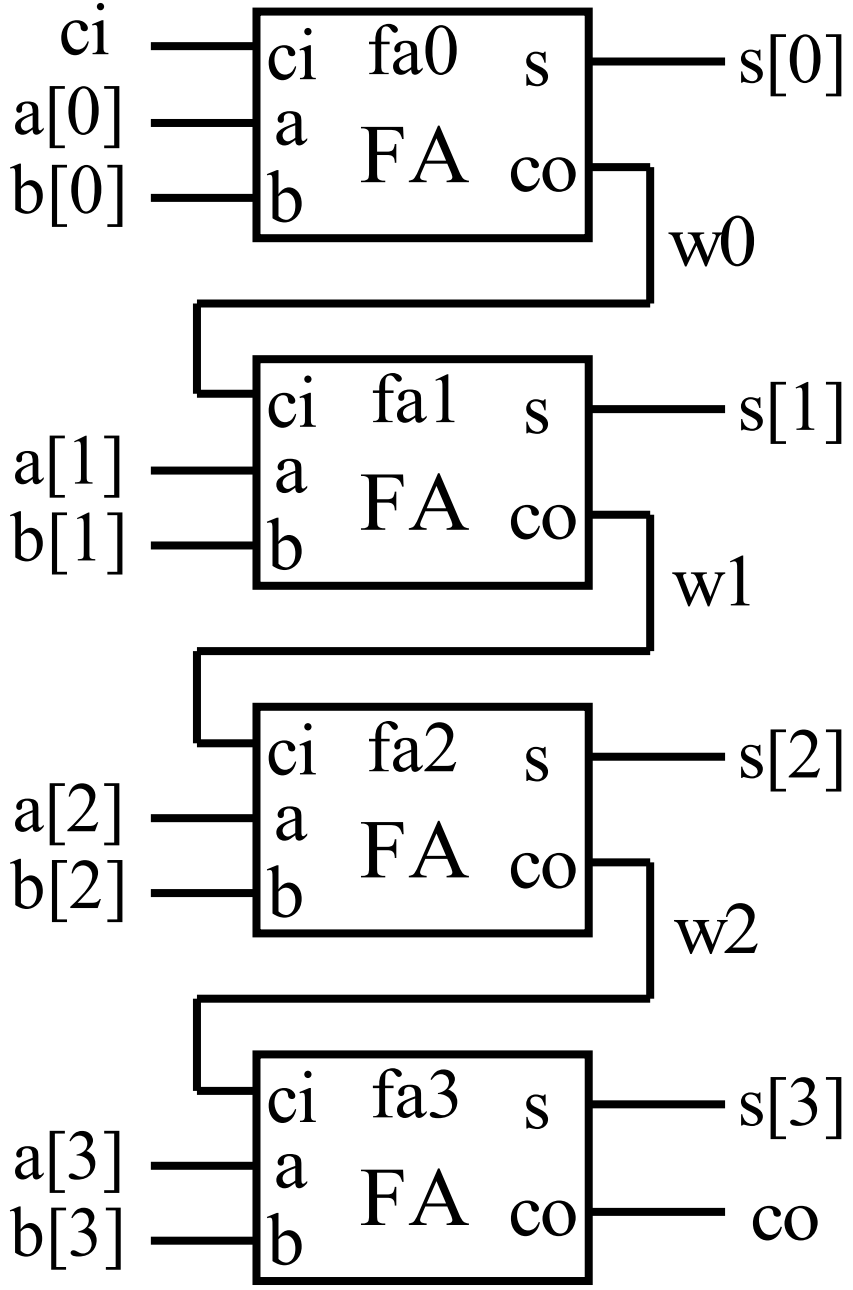
上位moduleで
宣言したwireの名前
またはportの名前

# 4bit Adder（加算器）

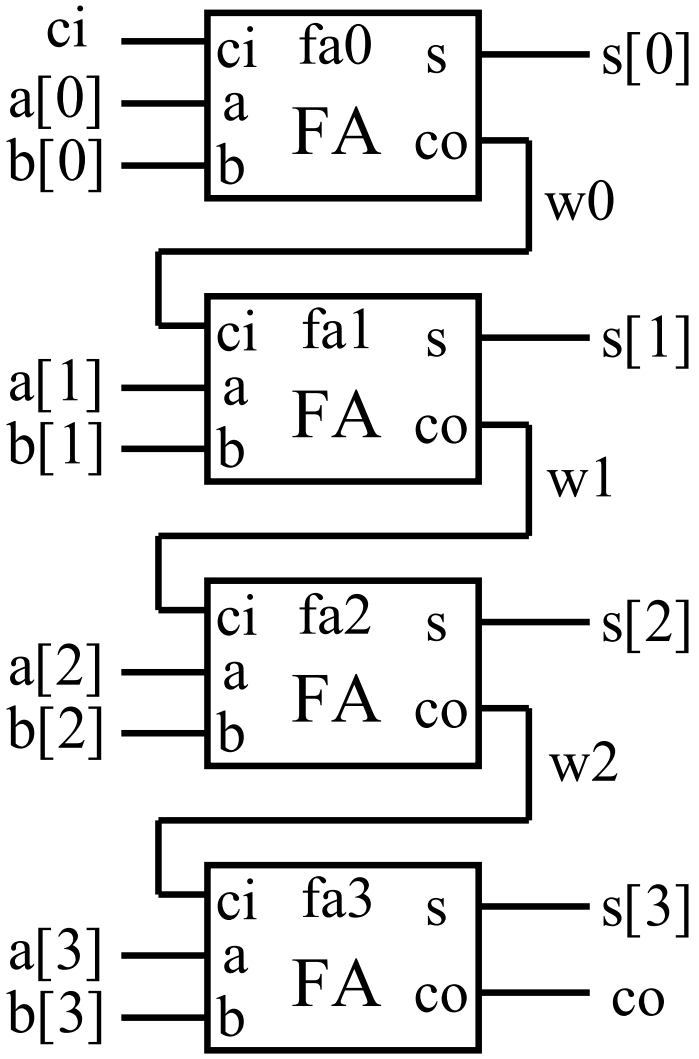



4つの全加算器：
下位桁のFAのキャリーcoと
上位桁のFAのキャリーciを接続



# 4bit Adder（加算器）

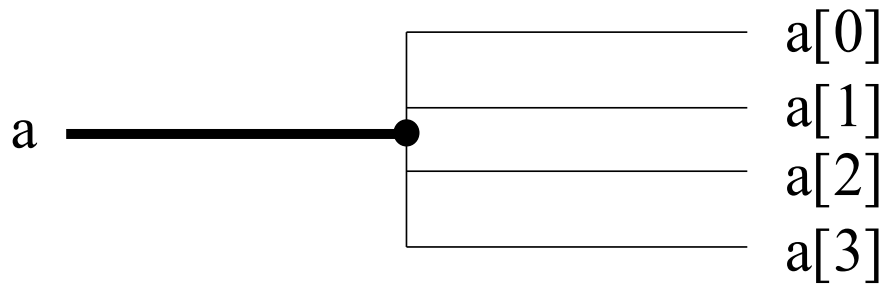



```
/* 4ビット加算器のmoduleの定義 */
module add4(a, b, ci, co, s);
    /* input port */
    input  [3:0] a, b;
    input        ci;
    /* output port */
    output       co;
    output [3:0] s;
    /* wire */
    wire w0, w1, w2;
    /* instance文 */
    full_add fa0(.a(a[0]), .b(b[0]),
        .ci(ci), .co(w0), .s(s[0]));

    full_add fa1(.a(a[1]), .b(b[1]),
        .ci(w0), .co(w1), .s(s[1]));

    full_add fa2(.a(a[2]), .b(b[2]),
        .ci(w1), .co(w2), .s(s[2]));

    full_add fa3(.a(a[3]), .b(b[3]),
        .ci(w2), .co(co), .s(s[3]));
endmodule
```

# 複数ビットの信号(1)

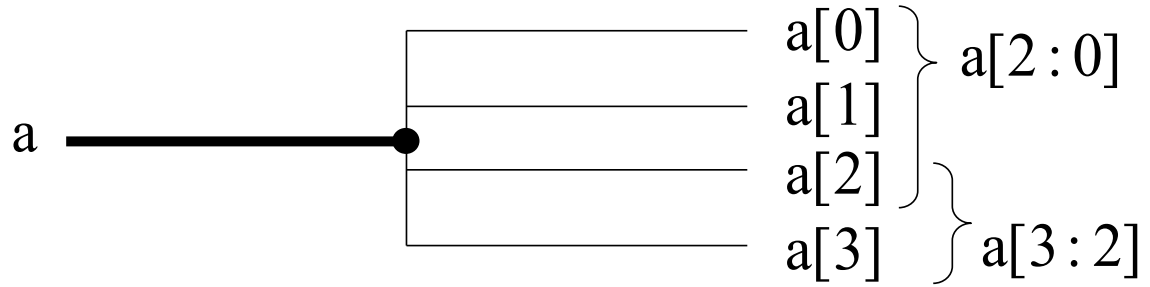
- 宣言方法　[MSB:LSB] ワイヤ名またはポート名
  - 4bitのwireを宣言する例: wire [3:0] a;

例)




- 信号の分離

例)



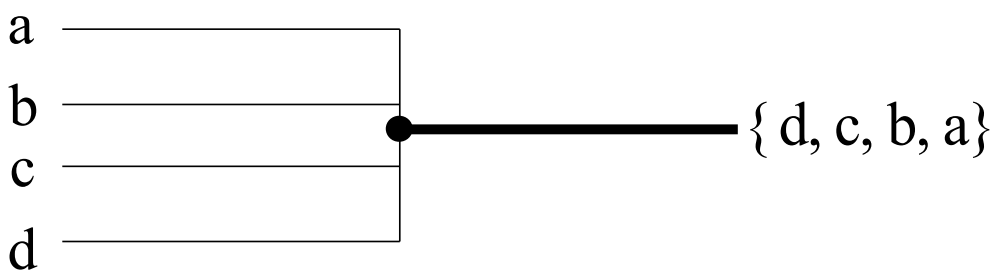

portやwireの宣言でビット幅を省略したときのビット幅は1

assign文やinstance文で省略した場合は宣言時のビット幅

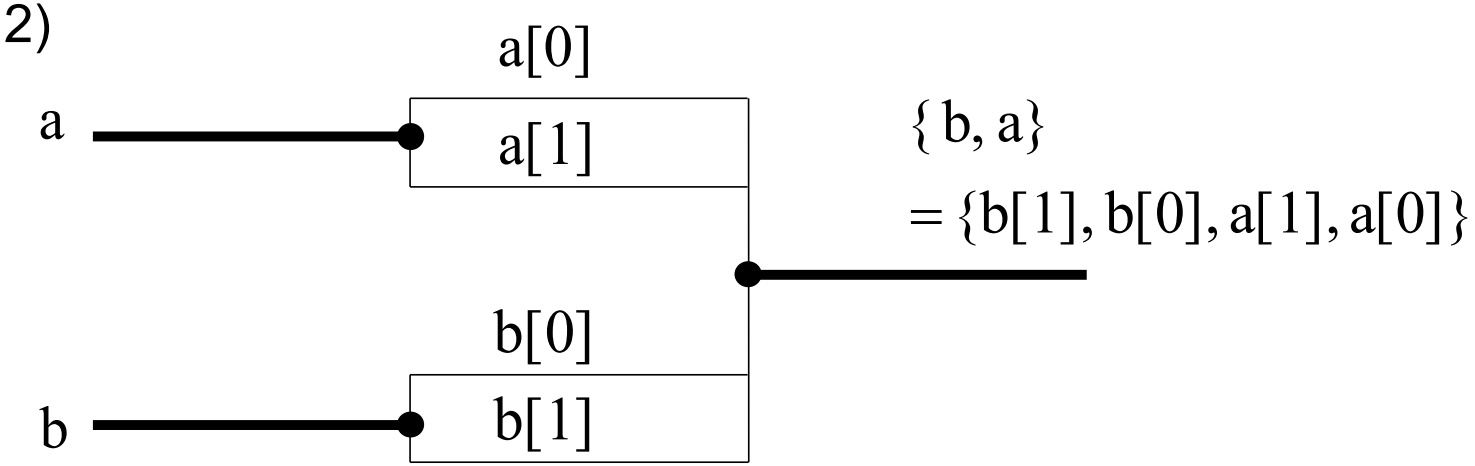
# 複数ビットの信号（2）

- 信号の連接

例1)

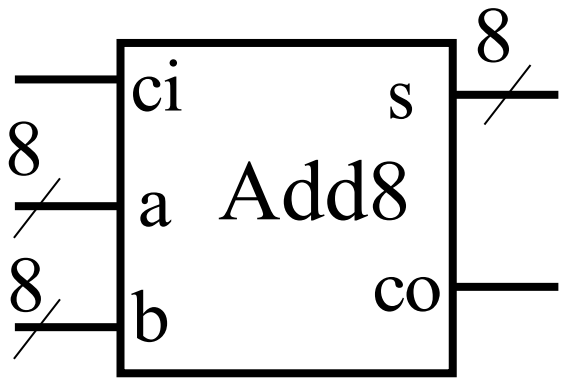


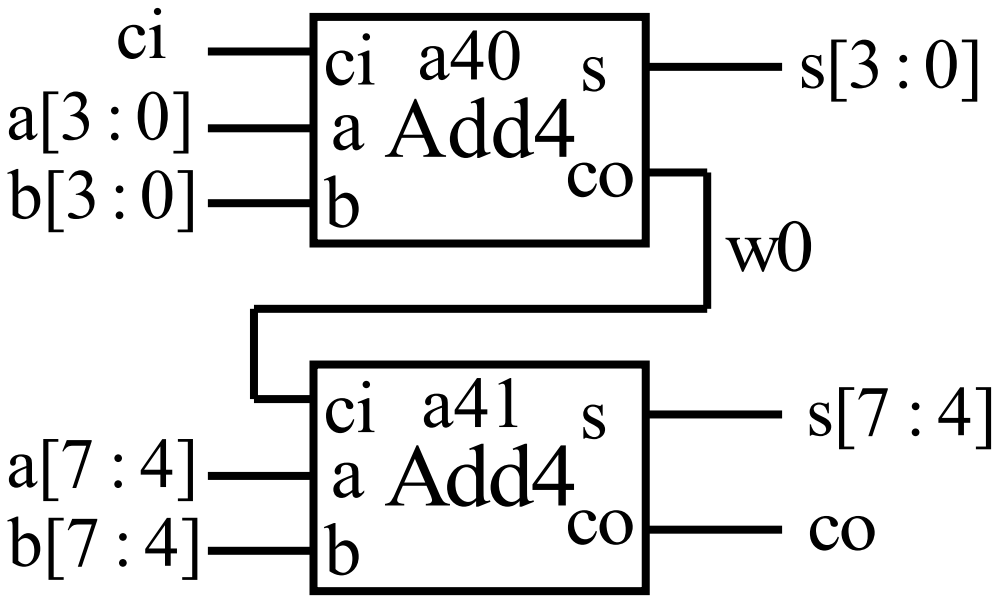

例2)




左に記述した信号が上位ビット、右に記述した信号が下位ビットになる

# 8bit Adder（加算器）

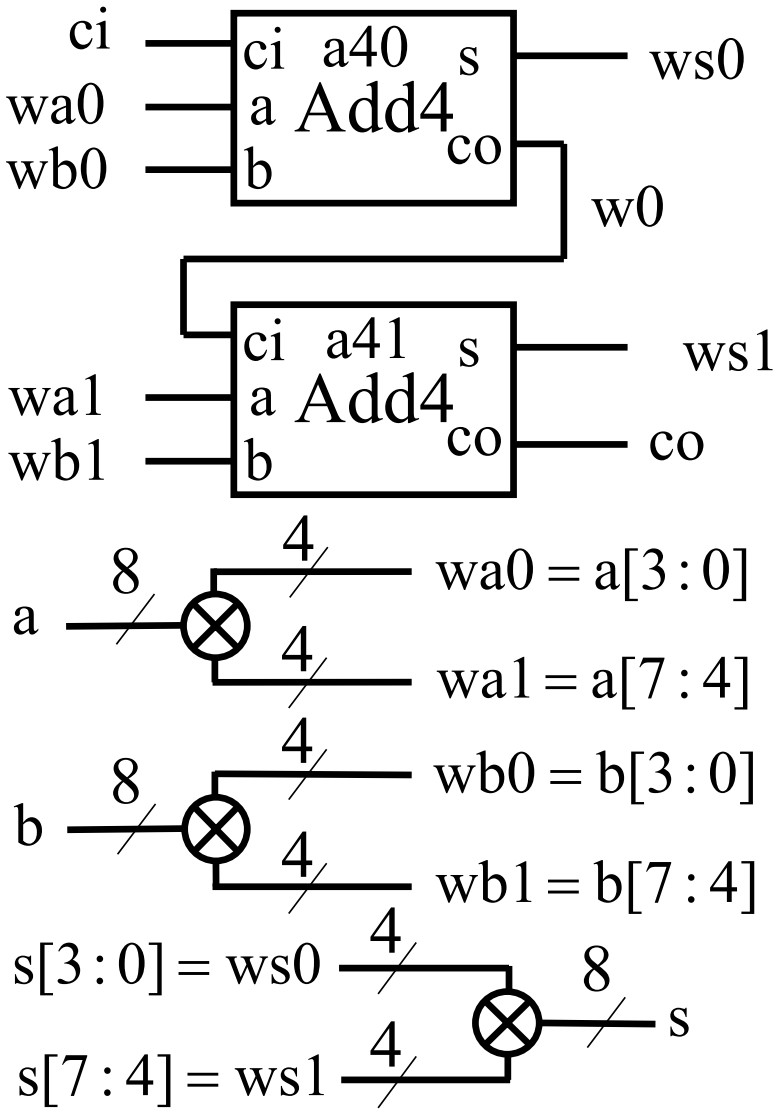



```
1  11  1
    0110  1010
+   1010  1100
-------------
    0001  0110
```

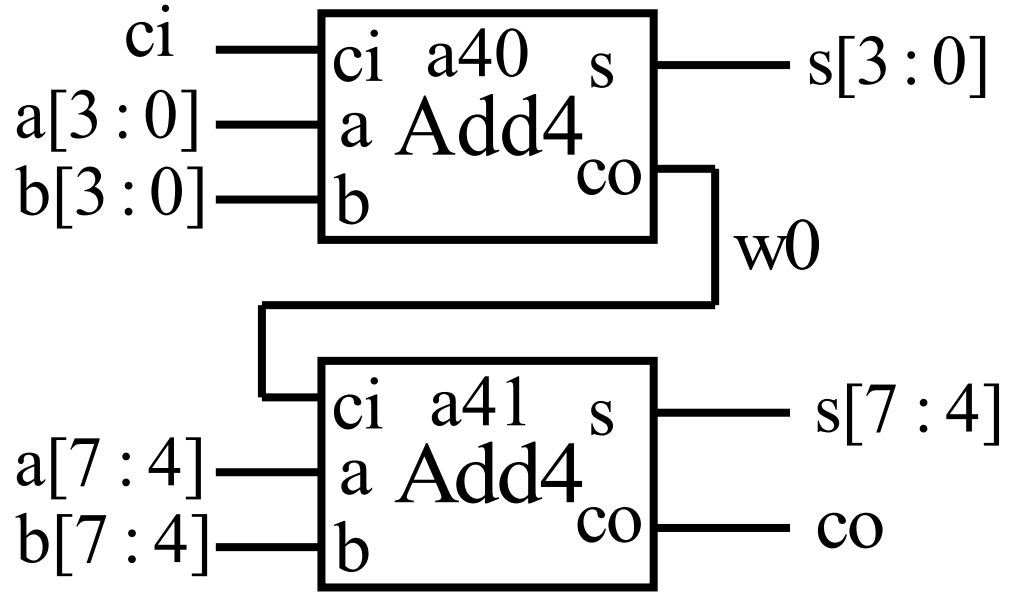
2つのAdd4：
下位桁のAdd4のキャリーcoと
上位桁のAdd4のキャリーciを接続

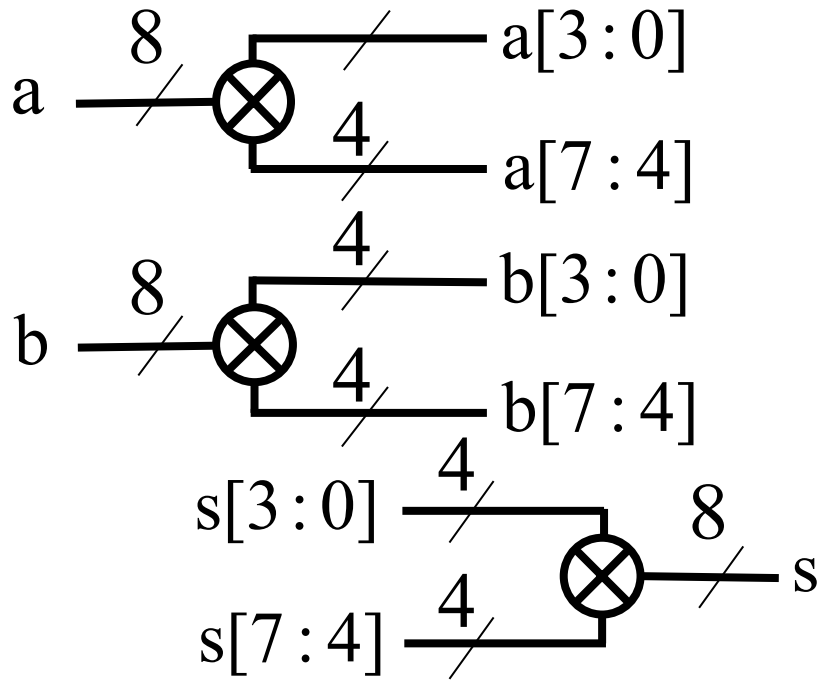
# 8bit Adder（加算器）




```
/* 8ビット加算器のmoduleの定義 */
module add8(a, b, ci, co, s);
    /* input port */
    input  [7:0] a, b;
    input        ci;
    /* output port */
    output       co;
    output [7:0] s;
    /* wire */
    wire w0;
    wire [3:0] wa0, wb0, ws0;
    wire [3:0] wb0, wb1, ws1;
    /* assign文、instance文 */
    assign wa0 = a[3:0];
    assign wa1 = a[7:4];
    assign wb0 = b[3:0];
    assign wb1 = b[7:4];
    add4 a40(.a(wa0), .b(wb0),
     .ci(ci), .co(w0), .s(ws0));
    add4 a41(.a(wa1), .b(wb1),
     .ci(w0), .co(co), .s(ws1));
    assign s = { ws1, ws0 };
endmodule
```

# 8bit Adder（別の記述）



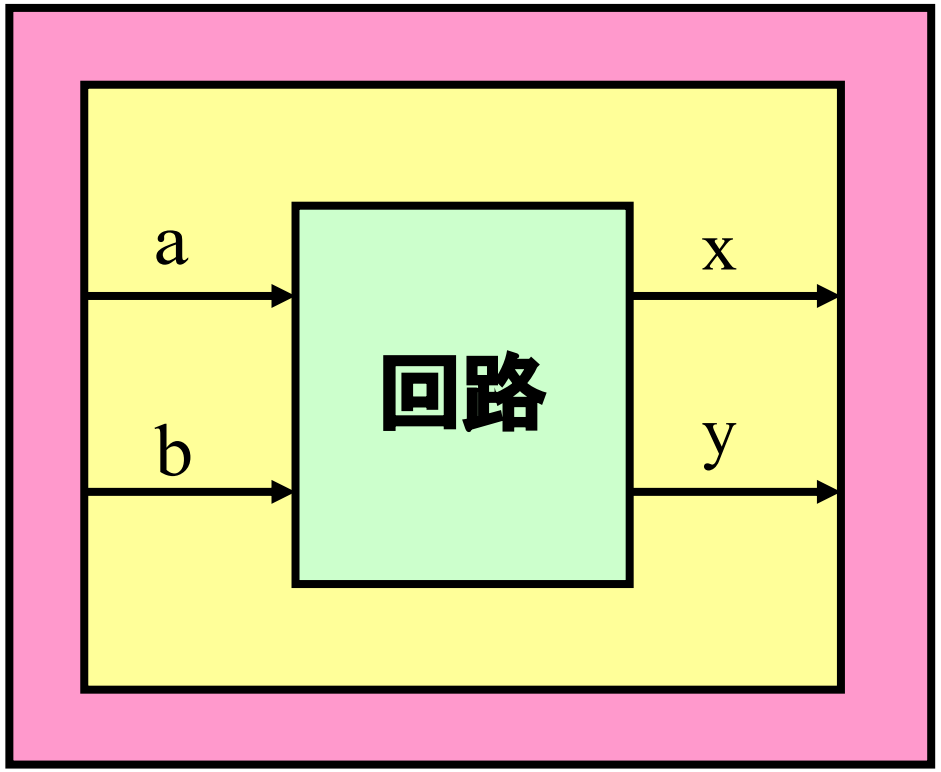

```
module add8(a, b, ci, co, s);
   input  [7:0] a, b;
   input        ci;
   output       co;
   output [7:0] s;
   wire w0;

   add4 a40(.a(a[3:0]), .b(b[3:0]), .ci(ci), .co(w0), .s(s[3:0]));
   add4 a41(.a(a[7:4]), .b(b[7:4]), .ci(w0), .co(co), .s(s[7:4]));
endmodule
```

# テストベンチ

**回路は入力が無ければ動かない！**

**例えば、電卓や時計は、ボタンを押してみないと**

**正しく動作するかどうかがわからない**

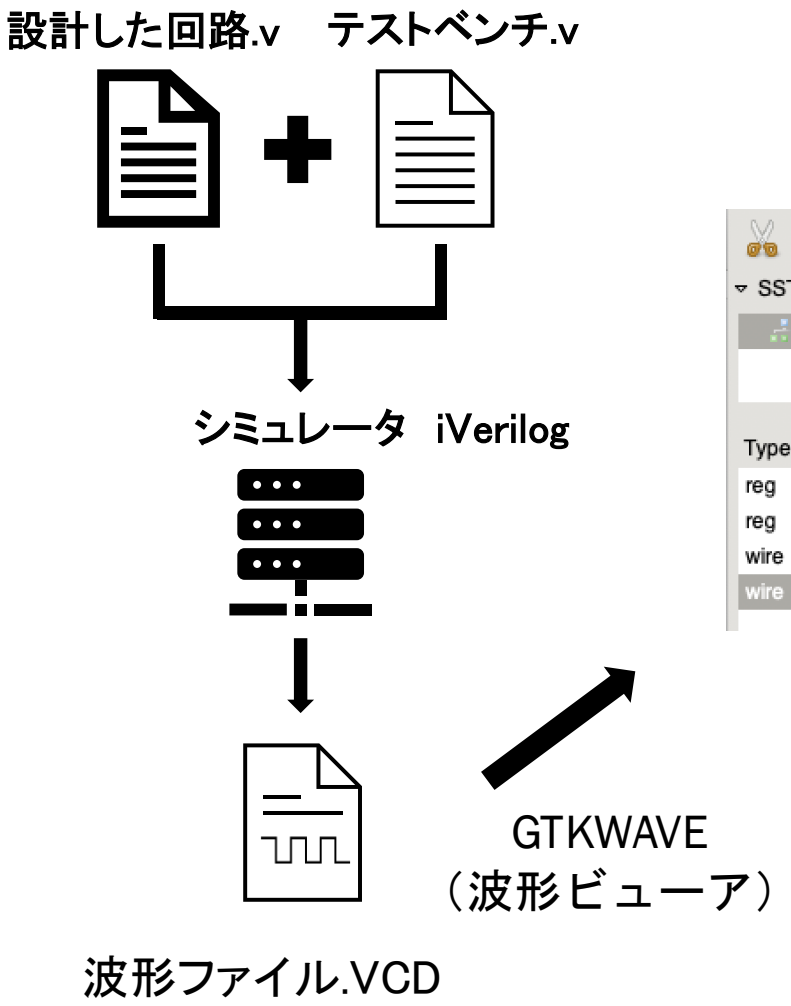



**回路に信号を入力して出力を観測**

| a | b | x | y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

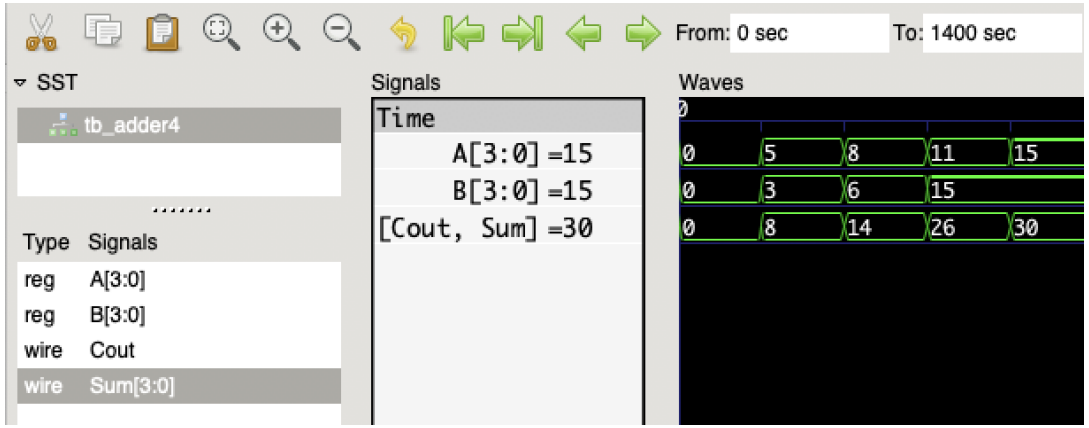
**テストベンチ**

# シミュレーション




## 設計した回路の機能検証

4-bit加算器シミュレーション波形例




- 動作レベルで設計機能検証可能
- 検証用テストベンチでもHDLで記述，検証効率向上

# (例) 半加算器のテストベンチ(1)

```
`timescale 1ns/1ns

module test_half_add();
    reg  a, b;
    wire co, s;

    half_add ha0(.a(a), .b(b), .co(co), .s(s));

    initial begin
        a = 1'b0; b = 1'b0;
        #10;
        a = 1'b0; b = 1'b1;
        #10;
        a = 1'b1; b = 1'b0;
        #10;
        a = 1'b1; b = 1'b1;
        #10;
        $finish();
    end

    ...

endmodule
```

単位時間と精度

テストベンチのmodule定義

テストするmoduleへの入力は reg
module**からの出力は**wire **で宣言**

テストするmoduleの接続

入力信号を定数で順に記述
**先頭の数値は信号のビット数**
b：2**進数**、d：10**進数**、h：16**進数**

**指定した時間が経過したら次の入力信号へ**

| a | b | co | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

# (例) 半加算器のテストベンチ(2)

```
`timescale 1ns/1ns
module test_half_add();

    ...

    initial begin
        $monitor($time,,"a=%b b=%b co=%b s=%b",
                 a, b, co, s);
    end


    initial begin
        $dumpfile("half_add.vcd");
        $dumpvars(0, test_half_add);
    end

endmodule
```

時間を表示

表示のフォーマット指定
%b：2進数、%d：10進数、%h：16進数

端末への表示

表示する信号を順に記述

波形表示用ファイル名の指定
拡張子はvcd

波形表示の開始時刻（通常は0）

テストベンチのmoduleの名前