# コンピュータ アーキテクチャ I　第５回

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)
東京大学大学院情報理工学系研究科 創造情報学専攻

# 課題の解説

# 課題 4

- １０から０までを２づつ下りながら数える以下のループをアセンブリ言語で書け
  - 使用する命令セットは第２回の講義資料のものに準じる
- ヒント：
  - 減算には add の代わりに sub を使う
  - 初期値は li 命令で設定
  - 講義中の for 文の例のようにメモリ上に i を置くとややこしいので，レジスタの上で全て終わらした方が楽

```
1: for (i = 10; i >= 0; i-=2) {
2: }
```

# C 言語

- C 言語

```
1: for (i = 10; i >= 0; i-=2) {
2: }
```

- そのままだと考えづらいので，
  まず上記のループを下記の形に変換して考える

```
1:      i = 10;         // 初期化部分
2: LABEL:               // ループの先頭
3:      i = i - 2;      // カウンタの更新
4:      if (i >= 0)     // ループの継続判定
5:          goto LABEL; // LABEL に戻る
```

# アセンブリ言語

```
// i = 10;
// レジスタ A に 10 をいれる
li 10→A
// if (i >= 0) で使う 0 をセット
// B に 0 を読み込む
li 0→B
// 戻ってくるためにラベルを定義
LABEL:
// i = i - 2;
// A から 2 を引く
sub A,1→2
// if (i >= 0)
//    goto LABEL;
0x40C: b A>=B, LABEL // 条件がなりたっていたら LABEL に飛ぶ
```

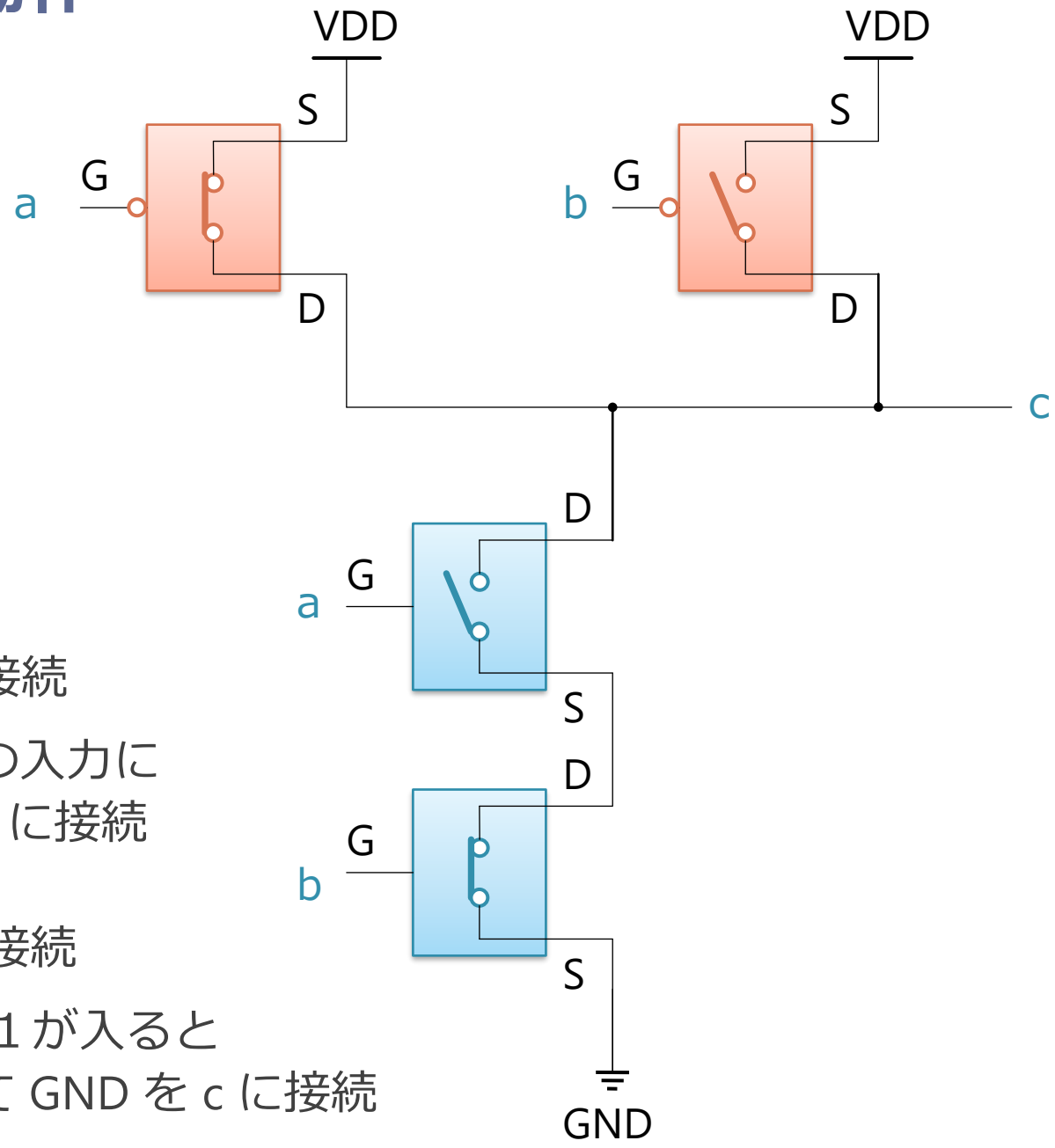
# 前回の振り返り

# 論理回路の作り方

- 以下のそれぞれの仕組みを使った回路を説明
  1. リレー
  2. CMOS
- これらの論理的な動作は実はほぼ同じに作れる
  - リレーの方が直感的にわかりやすいのでこちらから説明

# NAND ゲートの動作

| a | b | c |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




- 上側の P 型 2 つは並列接続
  - a と bどちらか片方の入力に 0 が入ると VDD を c に接続
- 下側の N 型 2 つは直列接続
  - a と b 双方の入力に 1 が入ると 2つとも ON になって GND を c に接続

# CMOS: Complementary Metal–Oxide–Semiconductor

- CMOS は NMOS と PMOS の 2 種類のトランジスタから成る
  - 電界で電荷（電子/正孔）を動かして ON/OFF する
- 基本的に N 型/P 型リレーとほぼ同じ動作
  - つまり全く同じように論理回路が組める
- 現代のコンピュータはこの CMOS を使って作られている

# リレーがあればプログラムが実行できる

- コンピュータがあれば，プログラムが実行できる
  1. コンピュータは機械語を解釈して実行できる
  2. 機械語はアセンブリ言語から変換できる
  3. アセンブリ言語は C 言語からコンパイルできる

- 途中に多数のステップがあるが，まとめると，
  - リレーがあれば，C 言語のプログラムを実行できる

# 一般化すると，

- **なんらかの入力に従って ON/OFF できるスイッチがあれば，それでプログラムが実行できるコンピュータが作れる**
  - 具体的な回路の組み方は，回路の種別ごとに違う事もある
  - NAND さえ出来ればこっちのもの

# 命令パイプライン

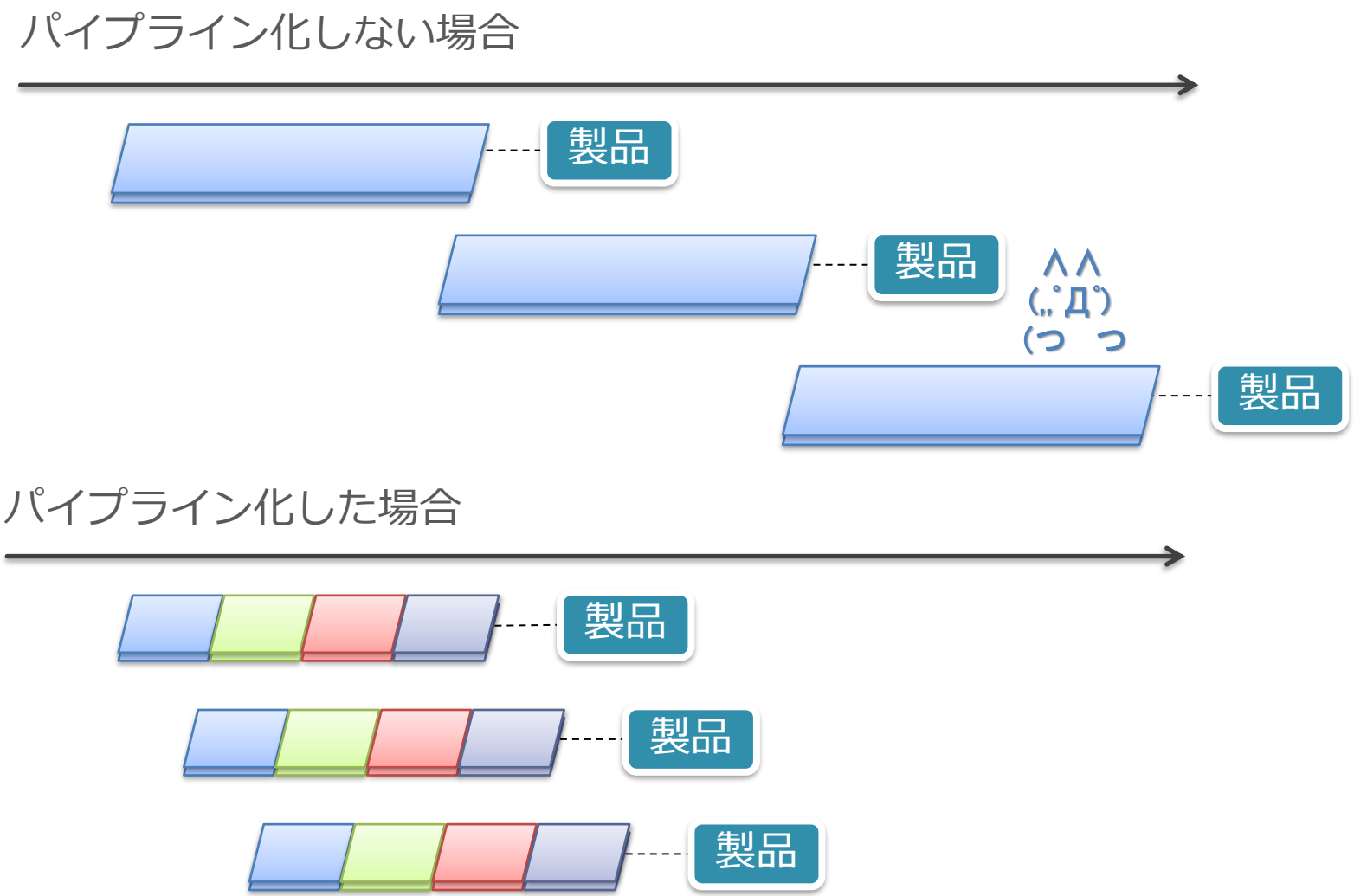
# 導入：工場のラインを考える


- ベルトコンベアのラインの上を製品が流れていく
  - ４人の人が，それぞれの工程の作業をおこなって完成
- 上のように1つしか製品をながさないと，
  - 各人は他の人が作業している間はヒマ

# 導入：工場のラインを考える


- 実際の工場：複数の製品を同時に流す
  - 各工程を並列して処理することによりスループットを向上
  - さっきの4倍の速度で製品ができあがっていく
- これが 命令パイプライン

# パイプライン化による性能向上



# 今日の内容：命令パイプライン

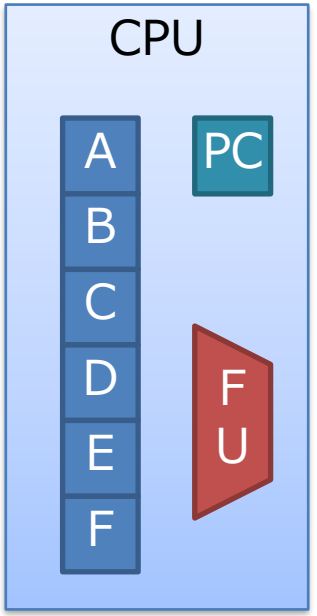
1. シングル・サイクル・プロセッサの動作
   - 全ての命令の処理が１サイクルで完結
     - これまでに説明していたプロセッサの動作と同じ
   - パイプライン化を前提とした，より詳細なものを使って復習
2. 上記のパイプライン化
   - 具体的にどうパイプライン化するか
3. パイプライン化の性能への影響

# シングル・サイクル・プロセッサの動作

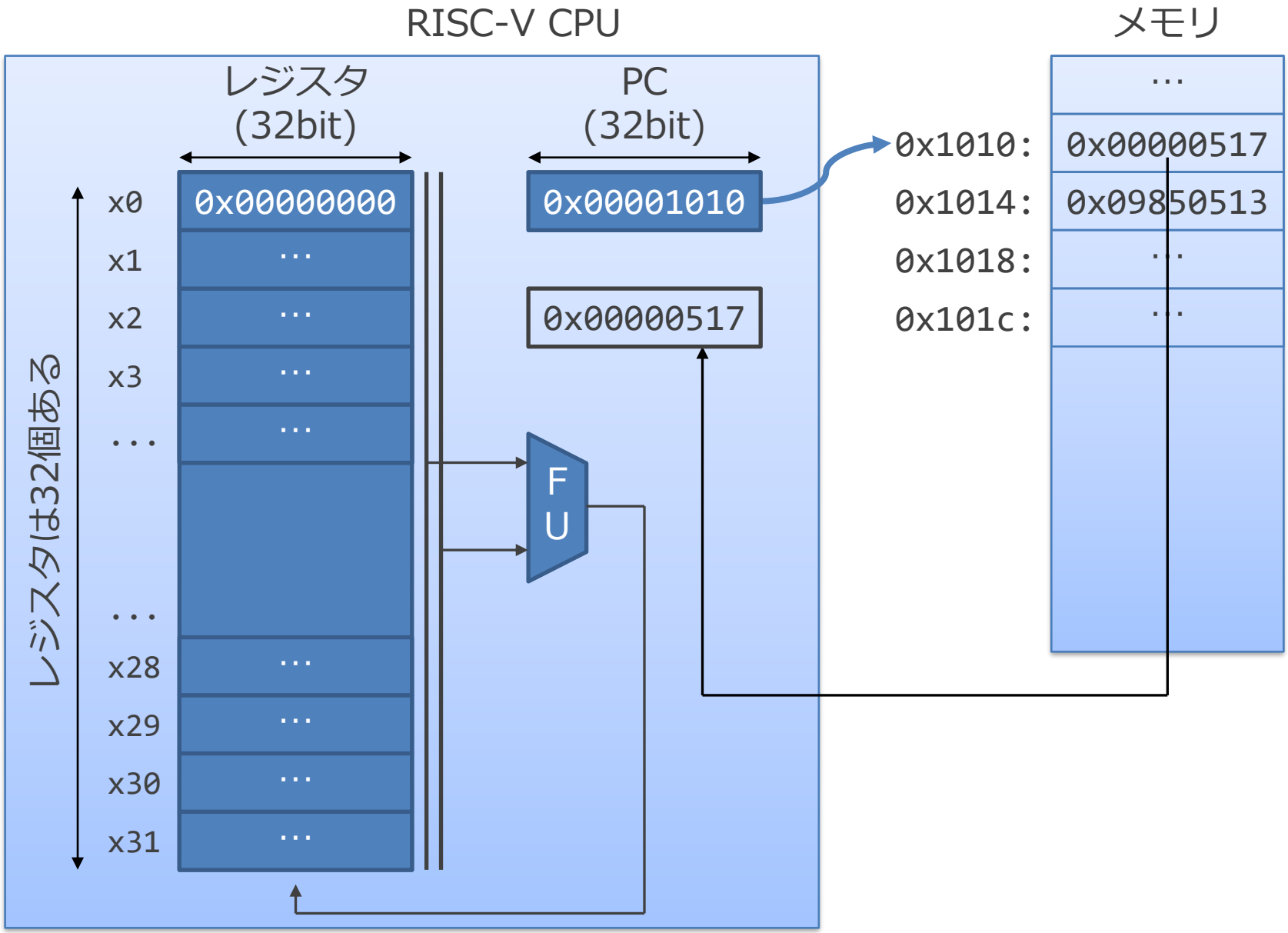
# もくじ

# これまでに説明した CPU のイメージ

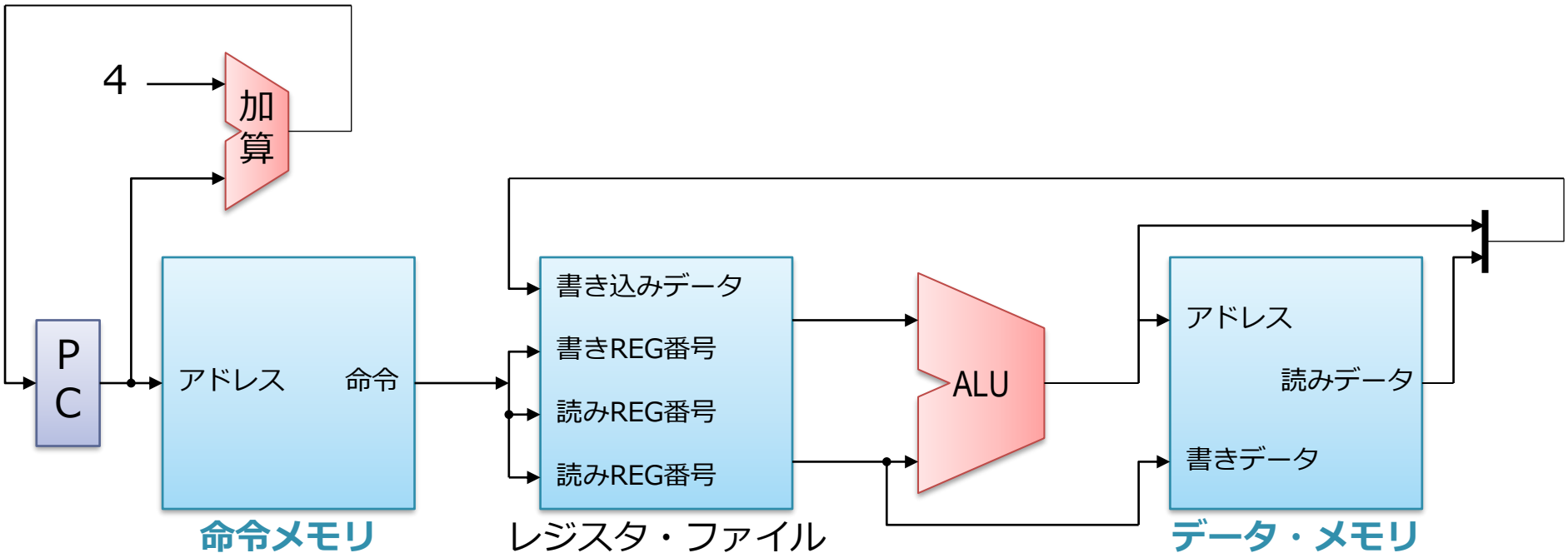
- コンピュータの心臓部
  - メモリから命令を読み出し，計算する
- 構成要素：
  - 演算器（FU: Functional Unit）
    - 加算器や AND 演算器など
    - 指示された種類の演算を行う
  - レジスタ・ファイル（右図では A,B,C...）
    - メモリと同様にデータを記憶する
      - 位置を指定して読み書きする
    - CPU の演算は，このレジスタ上でのみ行う
  - PC（Program Counter）
    - 現在見ている命令のアドレスを記憶している場所



# これまでに説明した RISC-V (32bit) のイメージ



# ベースとなるシングル・サイクル・プロセッサ

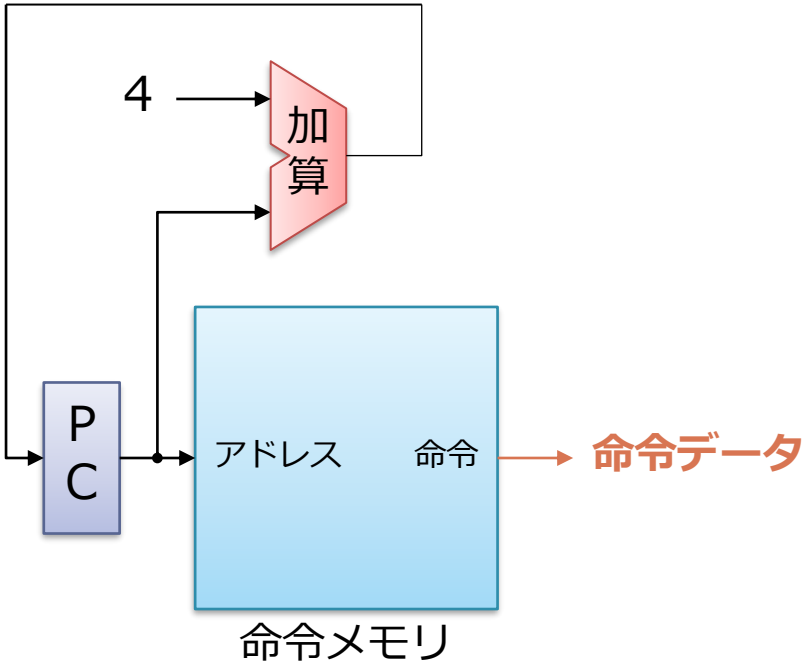



- 以前説明したものとの違い：
  - メモリが命令メモリとデータメモリに別れている
  - 算術 & 論理演算，ロード，ストアのみを実行可能
    - 分岐とジャンプは，簡単のために今は考えない

# 1命令の実行フェーズ

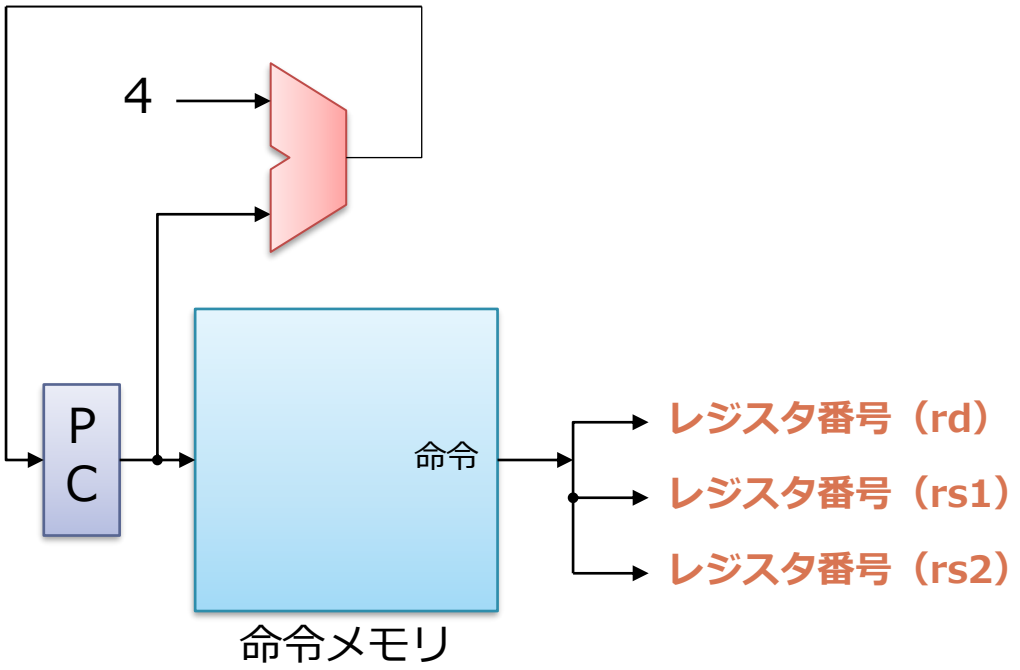
- 実行フェーズ
  1. フェッチ
  2. デコード
  3. レジスタ読み出し
  4. 実行
  5. レジスタ書き戻し
- RISC-V の加算命令を実行する流れをざっとみる

# 命令フェッチ

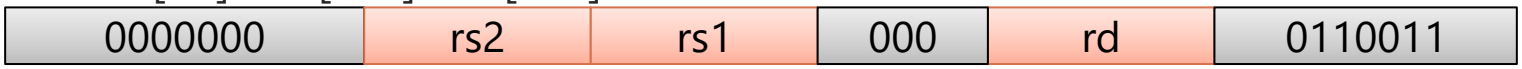



- 命令メモリから命令を読み出す
  - 命令メモリを順に読んでいくため，PC は毎サイクル加算される
  - 足している４は，RSIC-V では命令の幅が４バイトだから
  - 基本的に，この部分はどの命令でも変わらない

# 命令デコード

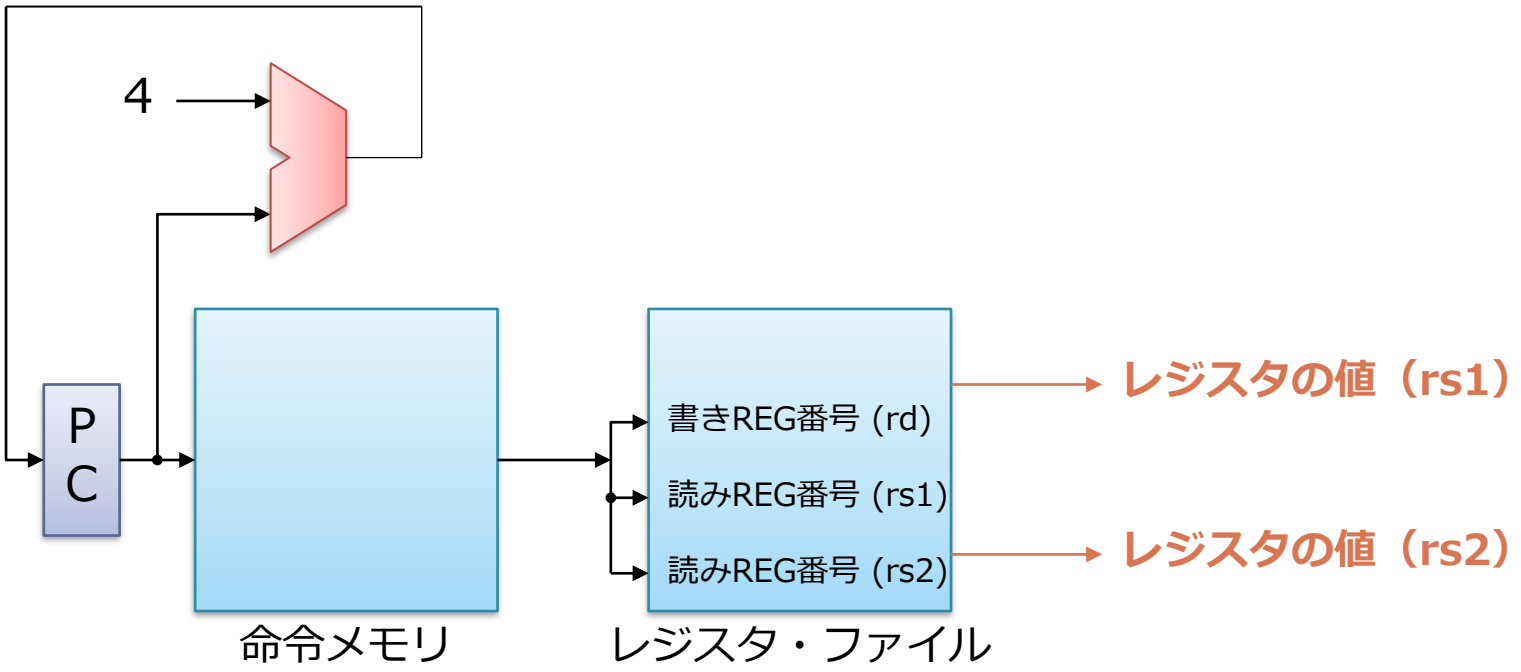



ADD : x[rd] ← x[rs1] + x[rs2]



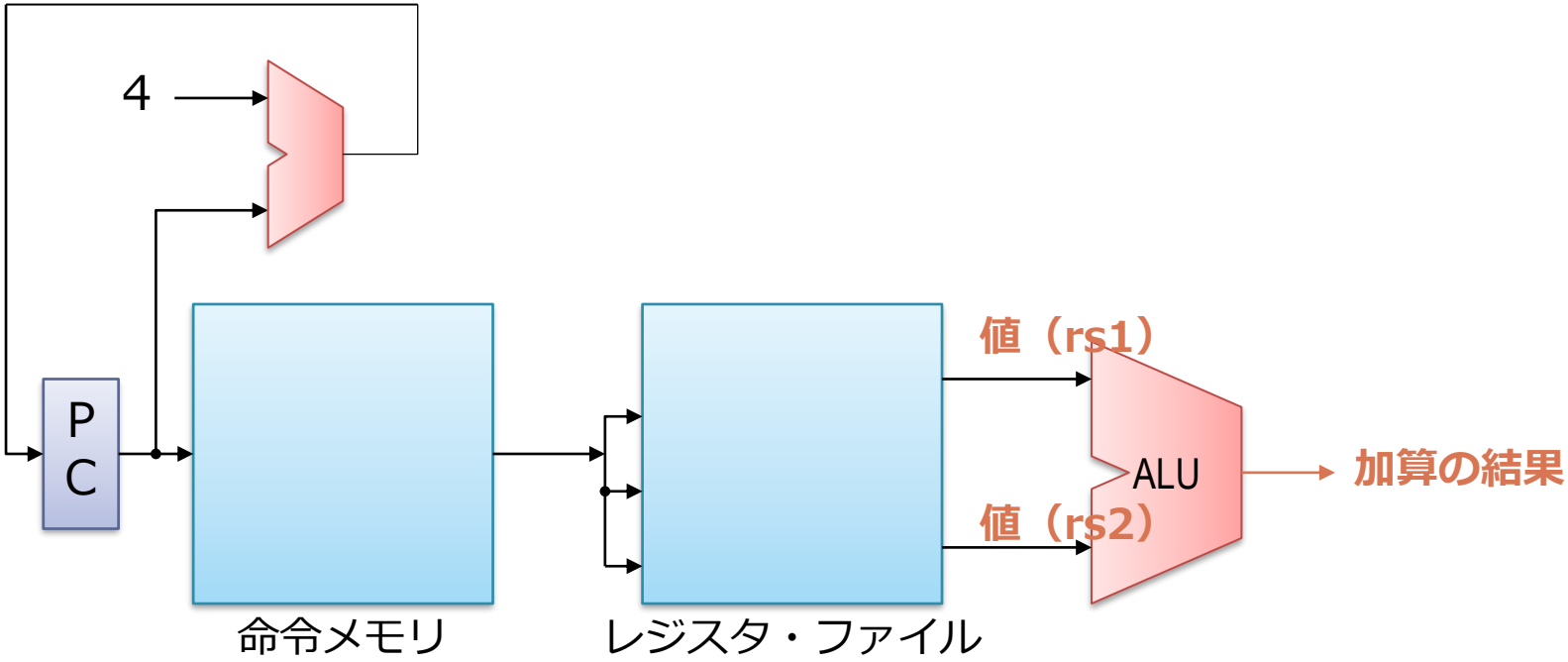

- 取り出した命令からレジスタ番号を表す部分のビットを取り出す
  - ソース（rs1, rs2）とディスティネーション（rd）

# レジスタ読み出し



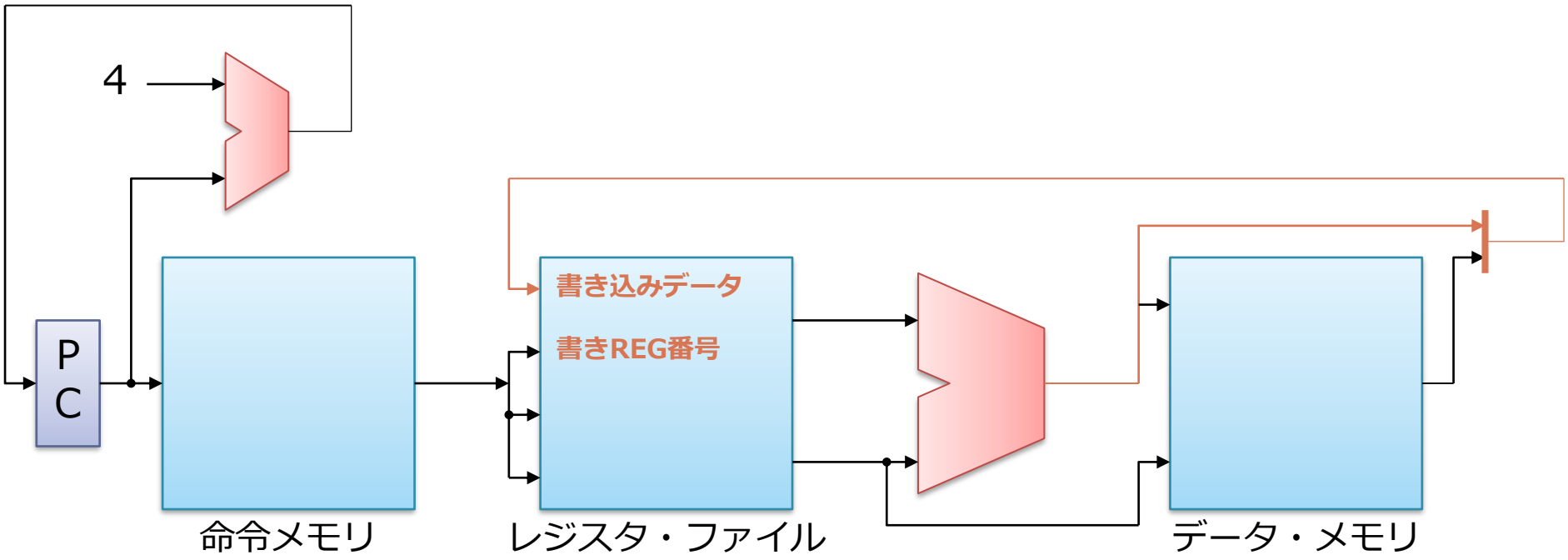

- デコードで得られたレジスタ番号を使って RF にアクセス
  - ソース・オペランドの値を読み出す

# 実行



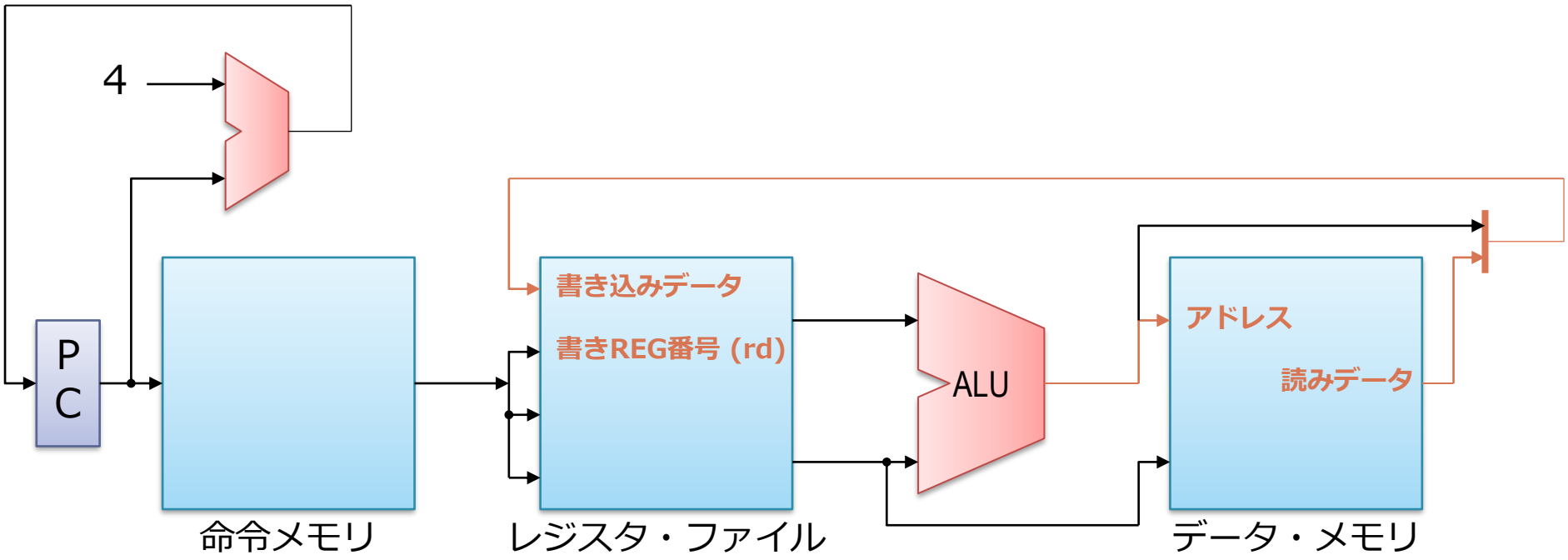

- RF から読みだした２つの値を加算

# レジスタ書き戻し




- 加算の結果をレジスタ・ファイルに書き戻す
  - データ・メモリには用がないので何もしない

# ロードの場合：メモリ・アクセスが加わる

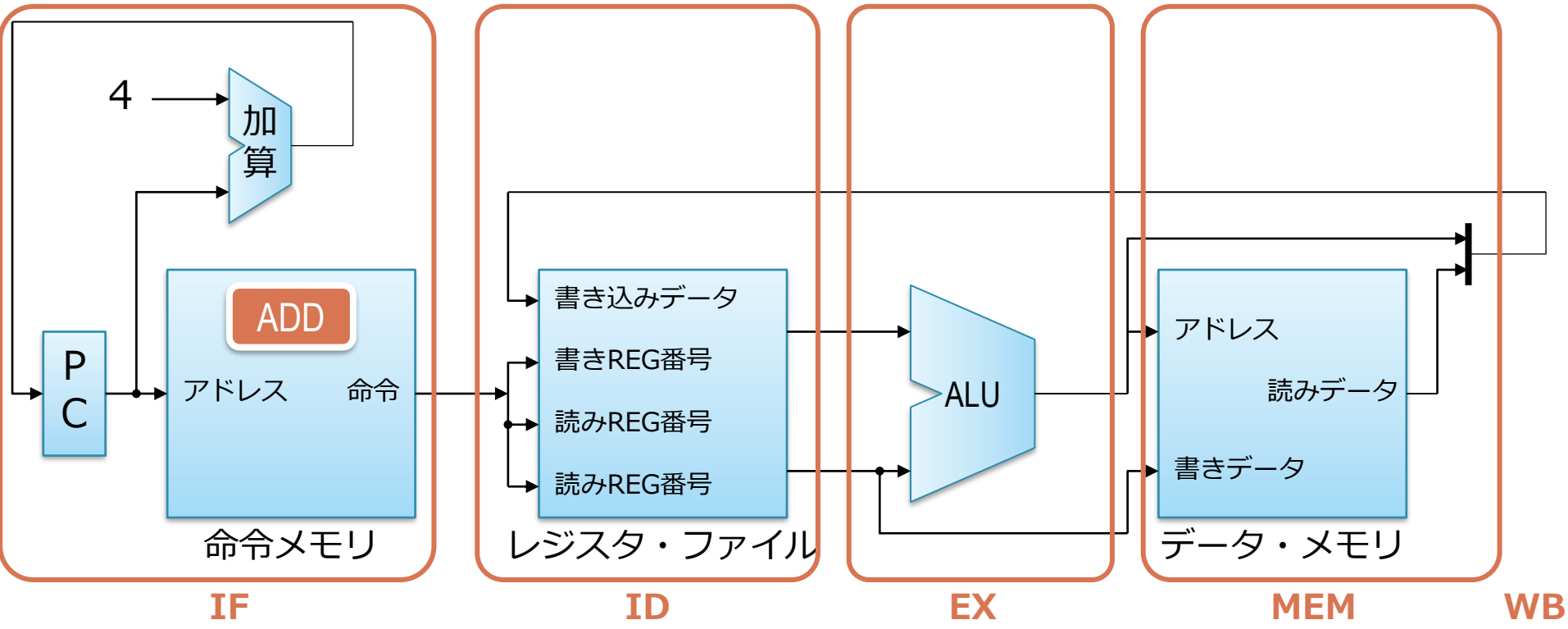



LW : x[rd] ← (x[rs1] + immediate)

| immediate[11:0] | rs1 | 010 | rd | 0000011 |
|---|---|---|---|---|

- 加算命令との違い：
  - アドレスの計算（x[rs1] + immediate）を ALU でやる
  - 得られたアドレスでデータ・メモリにアクセス

# 各処理は基本的には左から右に流れる




- 特定のユニットで仕事をしている間，他の部分は遊んでいる
- パイプライン化
  - これをもとに，導入で話したように処理をオーバーラップさせる

# パイプライン化

# もくじ

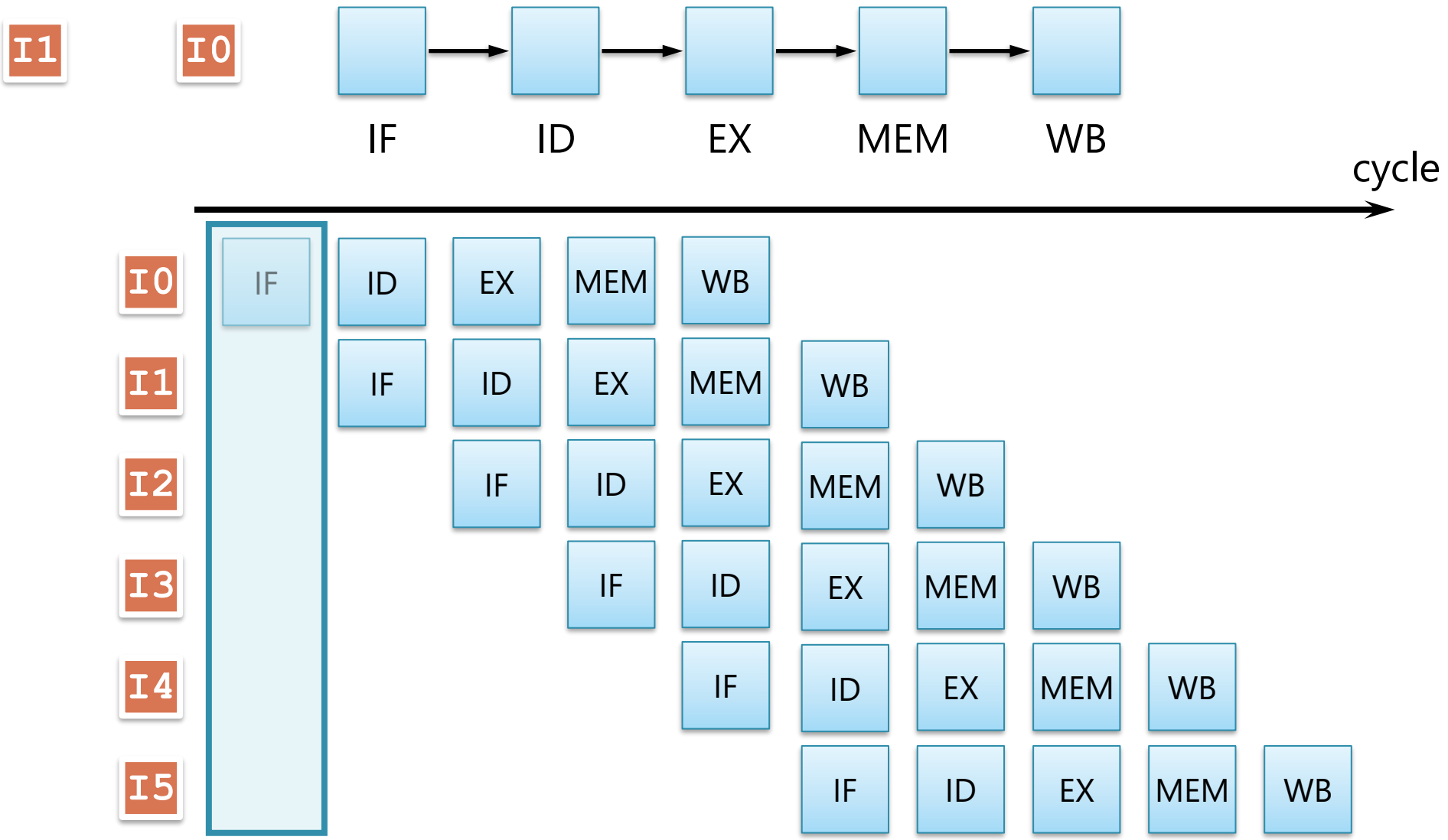
# パイプライン化

- 回路のまとまりをオーバラップさせる単位にする
  - この単位をステージと呼ぶ
- ステージ
  1. IF：　命令フェッチ
  2. ID：　デコードとレジスタ読み出し
  3. EX：　実行
  4. MEM：メモリ・アクセス
  5. WB：　レジスタ書き込み

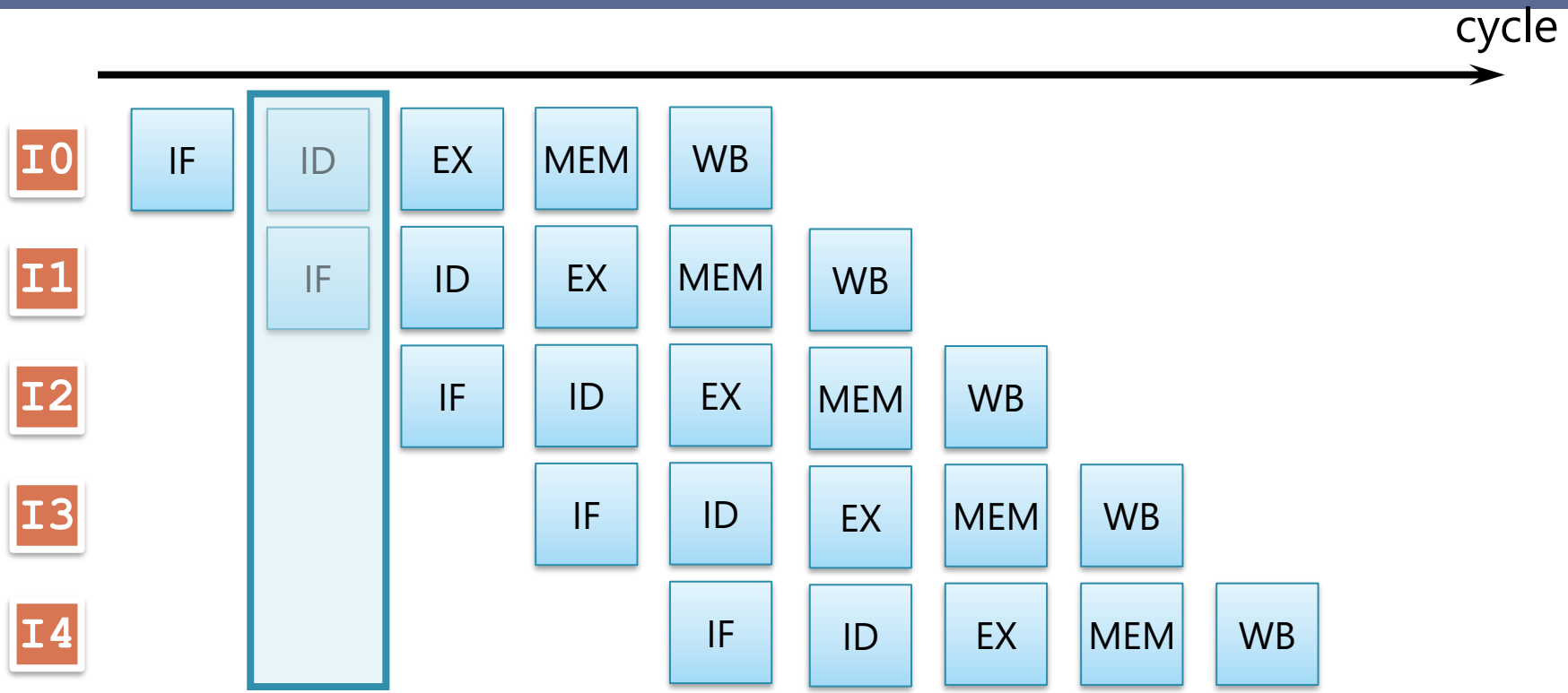
# 工場のラインを考える（再）


- 実際の工場：複数の製品を同時に流す
  - 各工程を並列して処理することによりスループットを向上
- これが 命令パイプライン

# 命令パイプラインの実行の様子

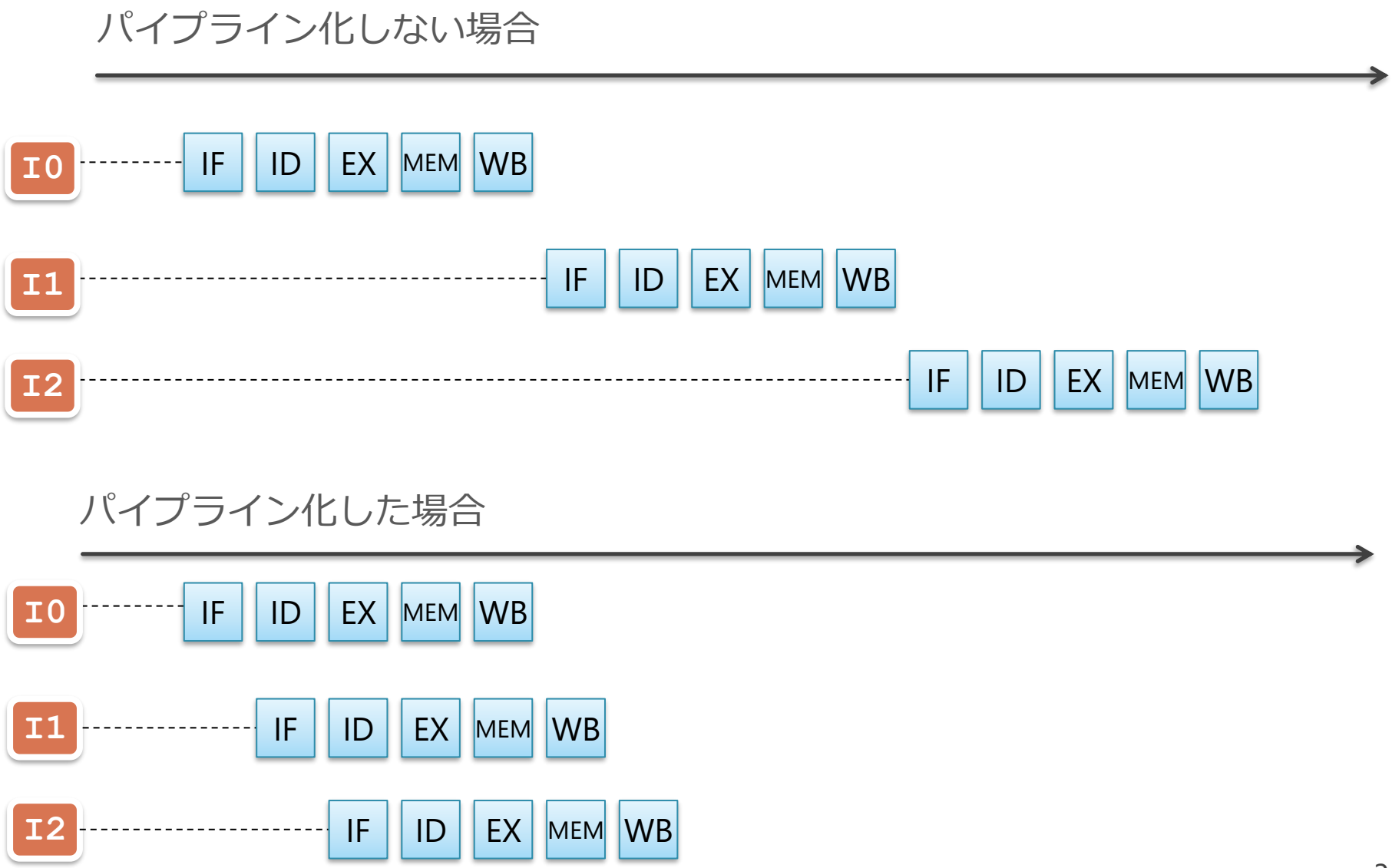


# パイプライン・チャートの見方
## ここから先で多用されるので重要

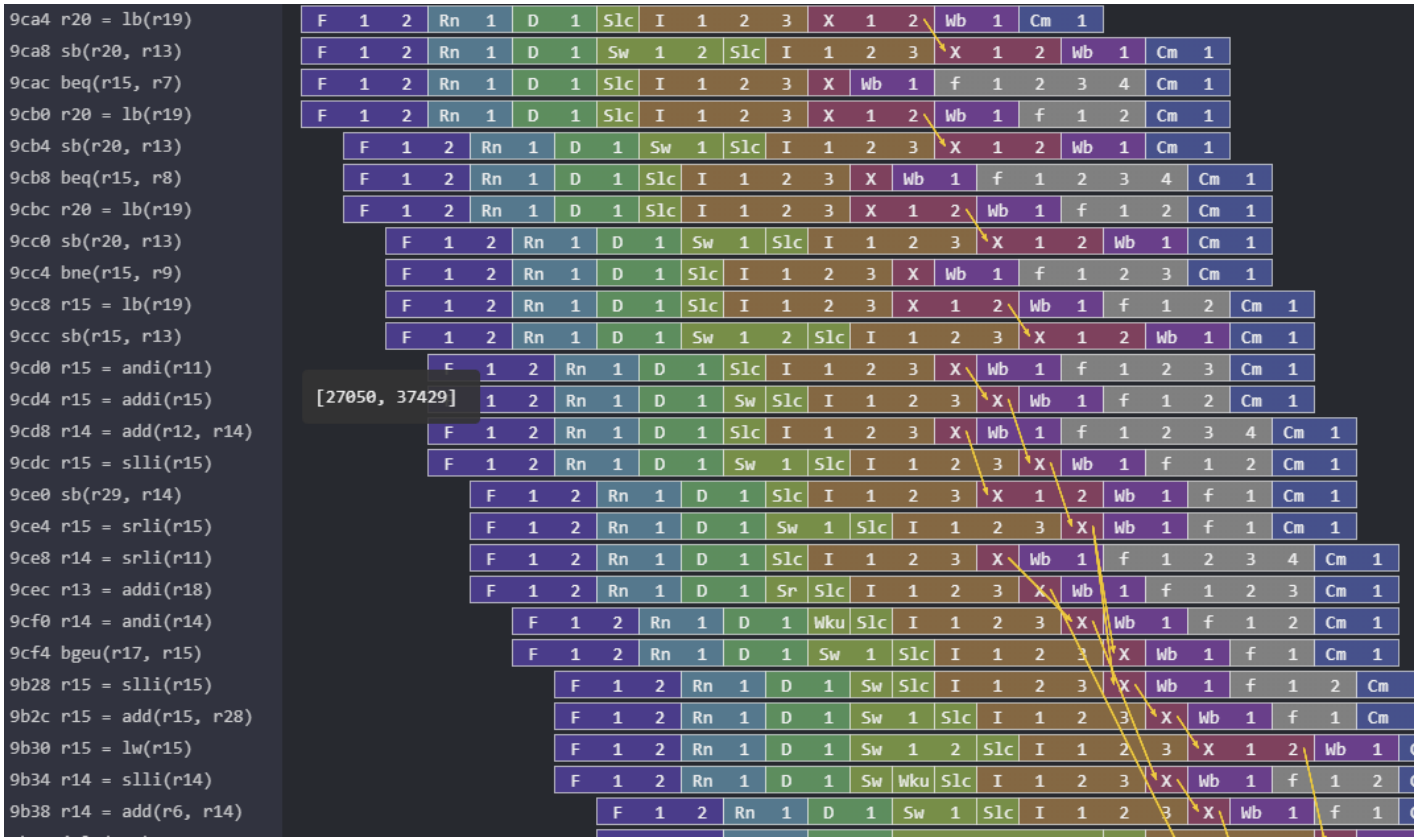



- 左から右にむかって時間は進む
- 上から下にむかって命令が実行順に置かれる
- 各ステージを表す四角は左側にある命令がその時そこにいることを示す
  - 上記では２サイクル目に，I0 が ID に，I1 が IF で処理されている

# パイプライン化による性能（スループット）向上



# 余談：実際の CPU を実行した場合のパイプライン



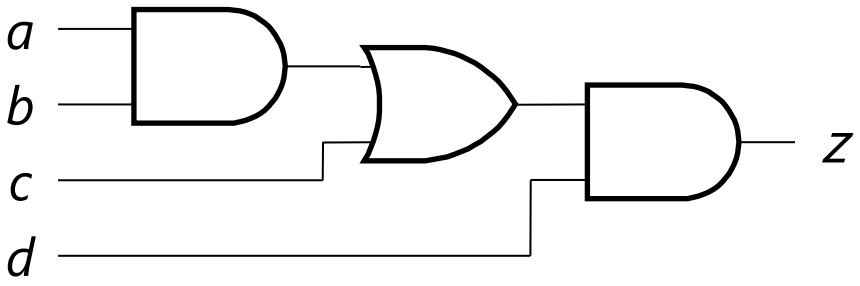

- 塩谷が開発している RISC-V CPU（RSD）の実行を可視化したもの
  - https://github.com/rsd-devel/rsd
  - out-of-order 実行をしているので，途中からプログラム順とは異なるタイミングで実行が進んでいる

# ステージを「どうやって」切るか

- シングル・サイクル・プロセッサの回路に適当に間隔をあけて命令を流せばよいというものもない

1. 各ステージを完全に同じ長さにすることは凄く難しい
   - 同じ長さ=同じ遅延=全く同じ段数の組み合わせ回路

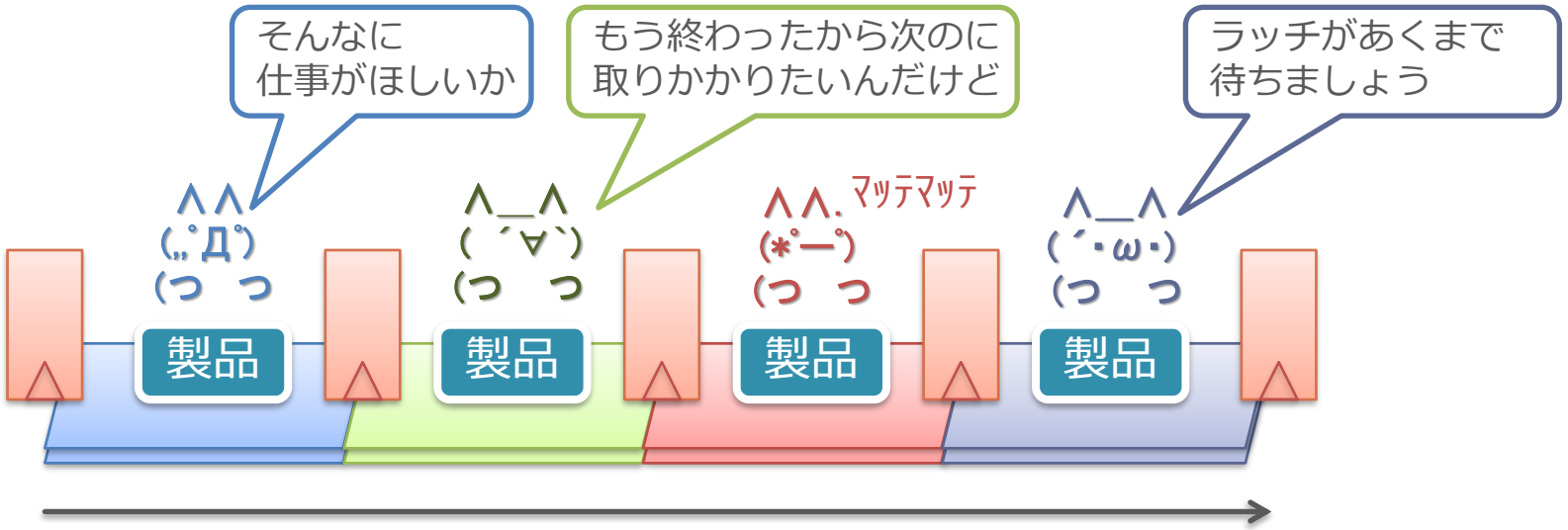
2. 長いステージであっても信号は絶えず変化する可能性がある
   - 短いパスから順に出力に反映される
   - たとえば下の回路で *a,b,c,d* が全て変化したとすると，まず *d* の変化が *z* に反映し，次に *d* が･･･



# パイプライン化（オーバーラップ）の実現方法

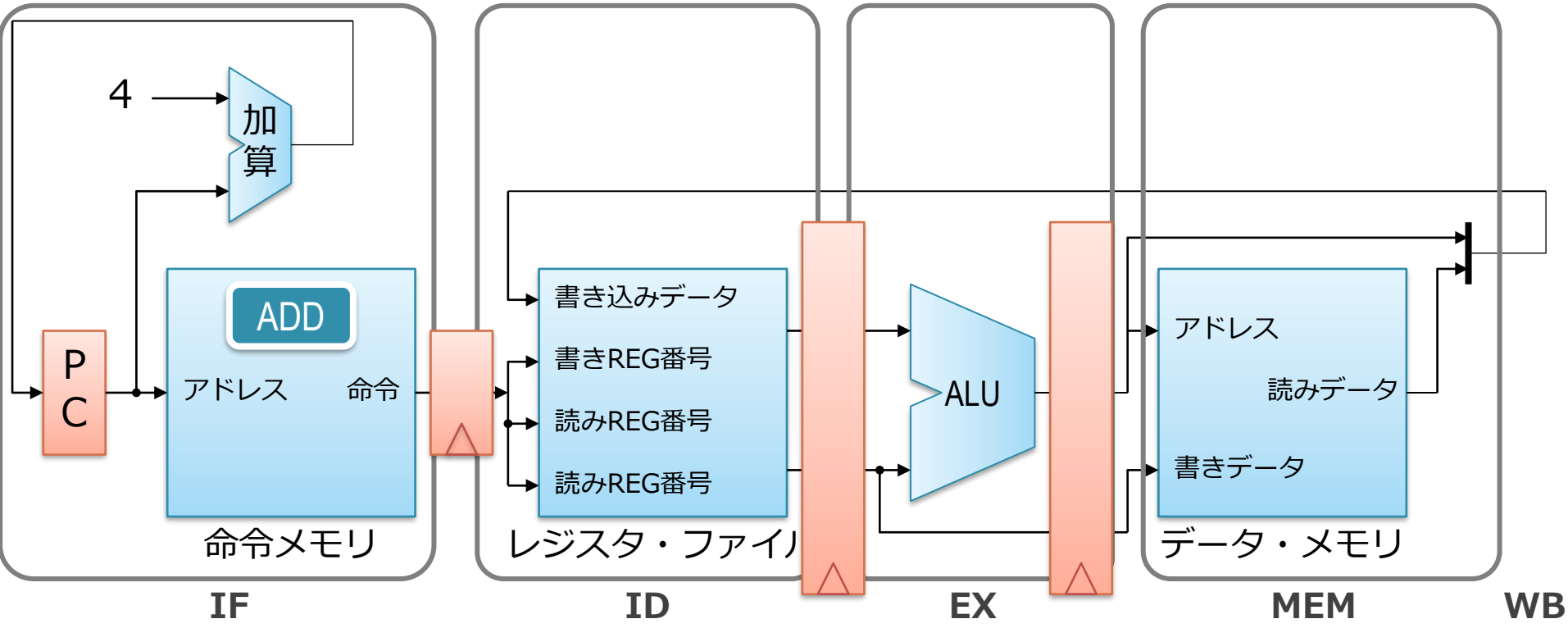
- シングルサイクル・プロセッサにフリップ・フロップをいれる
  - このためのフリップ・フロップをパイプライン・ラッチという
    - ラッチ（latch）は「ドアや門などの掛け金」の意味
  - パイプライン・ラッチで区切られた部分がステージとなる
- １サイクルの間はパイプライン・ラッチで信号がとまる
  - 複数ステージ間で信号が混じるのを防ぐ
  - 指定された時間までラッチでドアを開かなくするイメージ？

# パイプライン・ラッチのイメージ




- 各人の作業が終わっても，ラッチが開くまでは次の人に製品を送れない
  - 複数ステージ間で信号が混じるのを防ぐ
  - 指定された時間までラッチでドアを開かなくするイメージ？

# パイプライン化（オーバーラップ）の実現方法



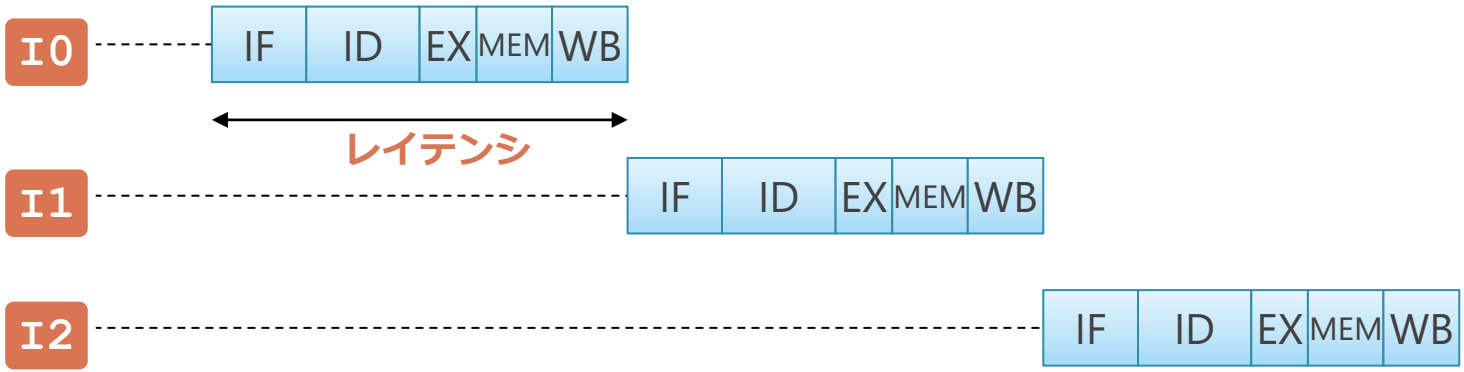

- 各ステージの間に，D-FF（オレンジの四角）をいれる
  - WB の書き込みについては，レジスタ・ファイル自体がクロックに同期して書き込みが行われるので D-FF は不要
- 各ステージの処理が早く終わっても，次のクロックまでは D-FF で信号の伝搬を止める

# パイプライン化の効果

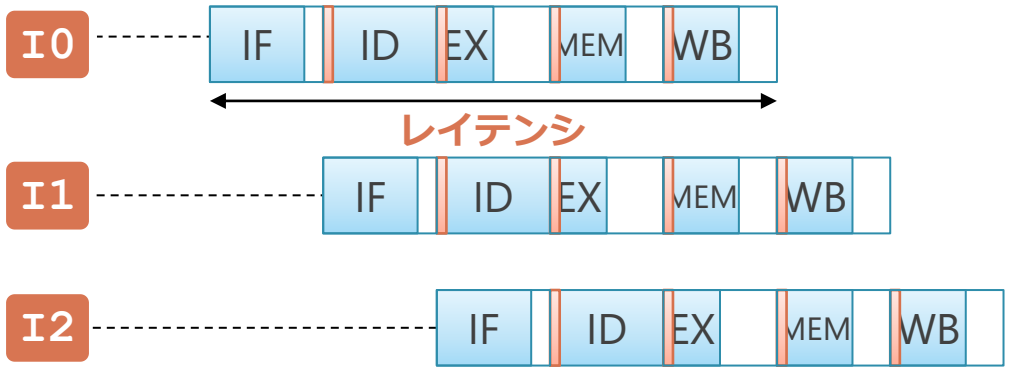
- レイテンシ (latency)：　短くならない（か，やや延びる）
  - 一続きの処理が始まってから終わるまでにかかる時間
    - この場合，1命令の始まりから終わりまでの処理時間
  - 一番長い処理に合わせて動かすので伸びる
    - 原理的に短くならない
      （ステージ間にフリップフロップが入る分は絶対のびる）
- スループット (throughput)：　ステージ数倍だけ上がる
  - 単位時間当たりの処理量
  - この場合，単位時間あたりに実行される命令数

# パイプライン化の効果
# レイテンシは伸びるが，単位時間あたりの処理命令数は増える

パイプライン化しない場合

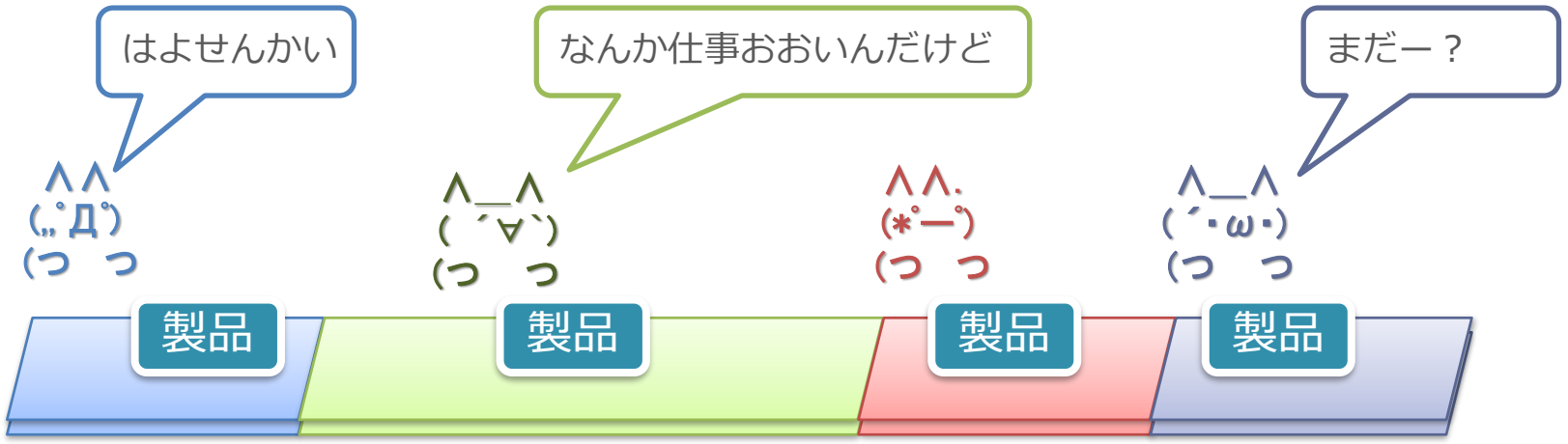



パイプライン化した場合



# ステージを「どこで」切るか

- 大きな回路のまとまりをステージにする
  - 回路のまとまりが大きい → 遅延も大きい
- この遅延の大きさが揃っていないと，綺麗にうごかない
  - パイプライン全体は，一番遅いステージの遅延にあわせて動く
  - 他の人が仕事が終わったからと言って，先に送れない
- 良くない例：緑の人だけ仕事が多いので，全体が動かせない



# ステージを「どこで」切るか

- ステージ
  1. IF：　命令フェッチ
  2. ID：　デコードとレジスタ読み出し
  3. EX：　実行
  4. MEM：メモリ・アクセス
  5. WB：　レジスタ書き込み

- 上記では，デコードとレジスタ読み出しが ID ステージにまとめられている
  - デコードにかかる遅延はほとんどない
  - 読み出した命令からオペランドを取り出すのは，単に信号線を繋ぐだけで良い

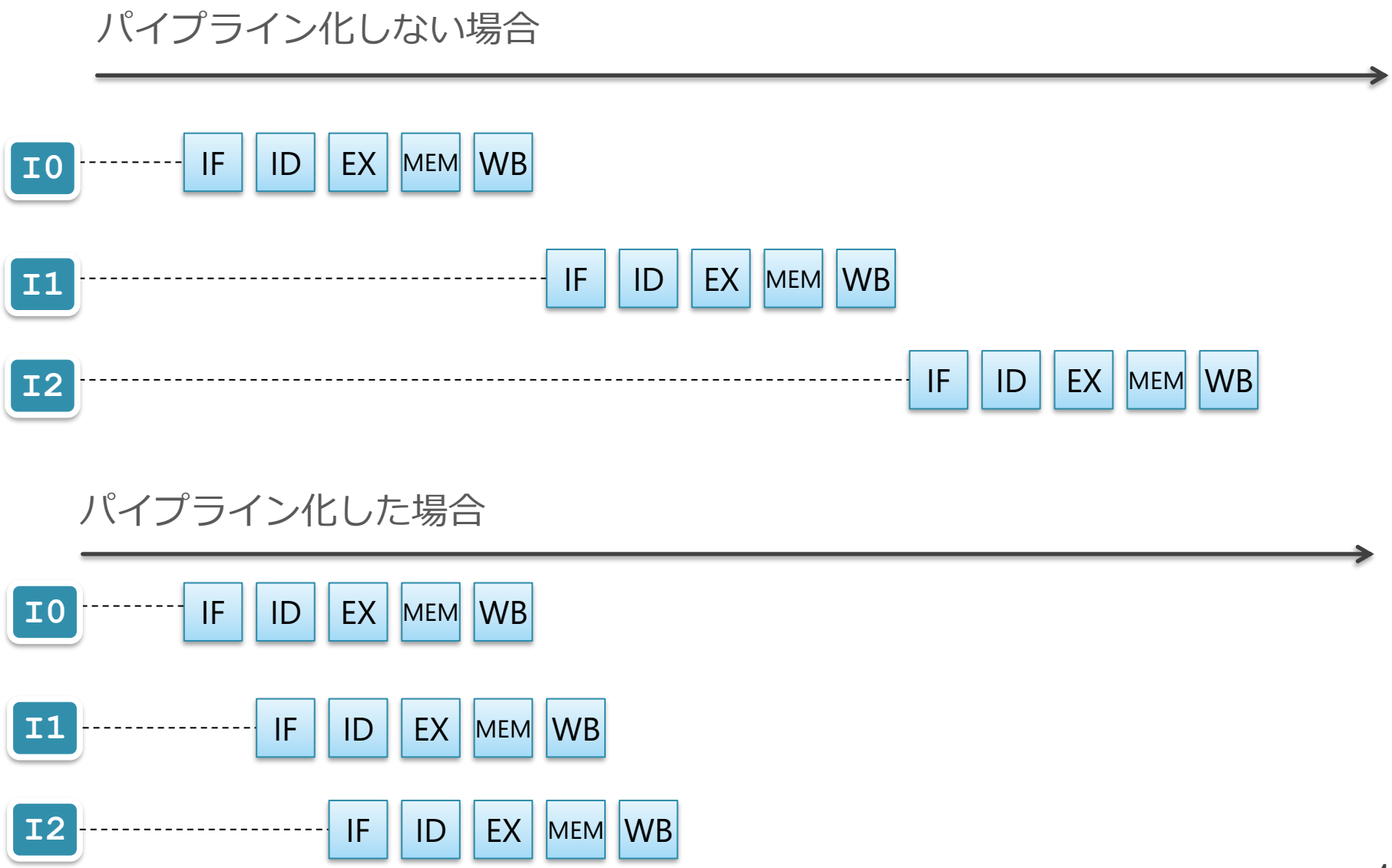
# 余談：非同期回路やウェーブ・パイプライン

- クロックによる同期化を使わずにパイプラインを作る方法もあるにはある
- やり方：
  1. 色々な方法でステージ間の遅延の大きさを気合いで揃える
  2. 一定間隔でデータを流す
- 設計 & 動作させることがすごく難しいので，主流ではない
  - 特に，高速動作がかなり難しい

# パイプライン化の性能への影響

# もくじ

# パイプライン化によるスループット向上

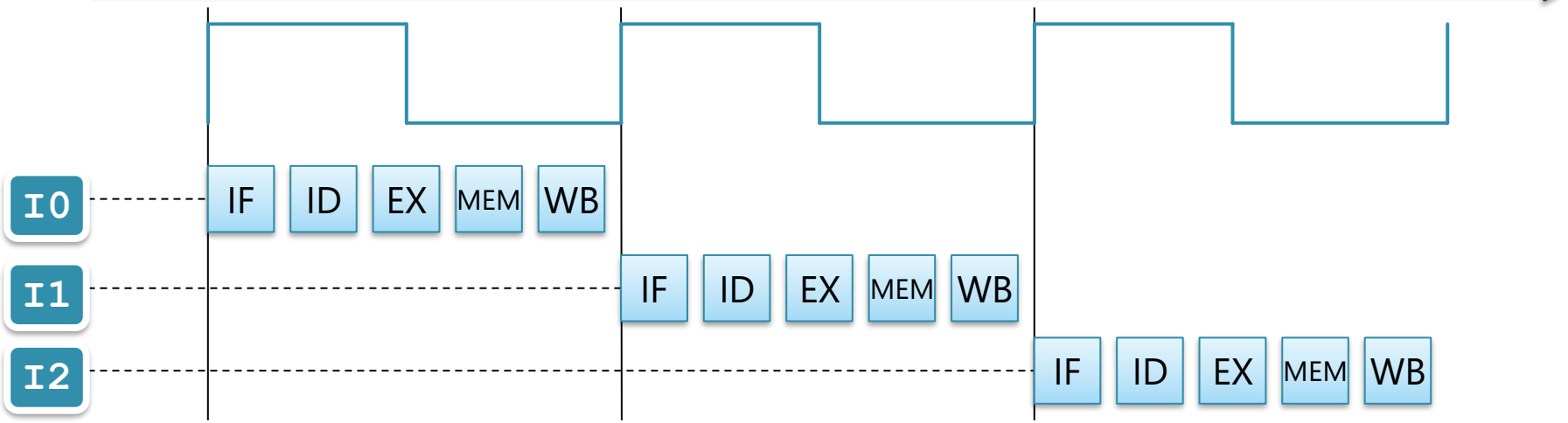


# パイプライン化の意味

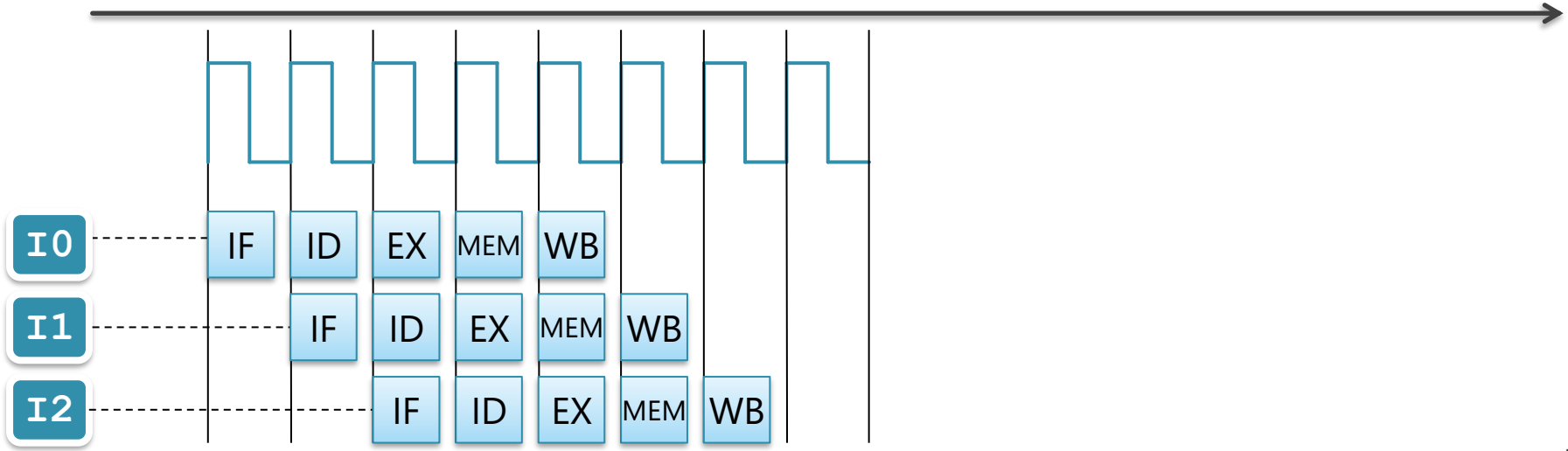
- パイプライン化の効果：
  - スループットの向上
  - = 単位時間あたりに処理できる命令の数の増加
  - = 動作クロック周波数の向上
- これらは同じ事を言い換えてるだけ

# パイプライン化によるクロック周期の短縮
## クロックの立ち上がりごとに，１命令が処理

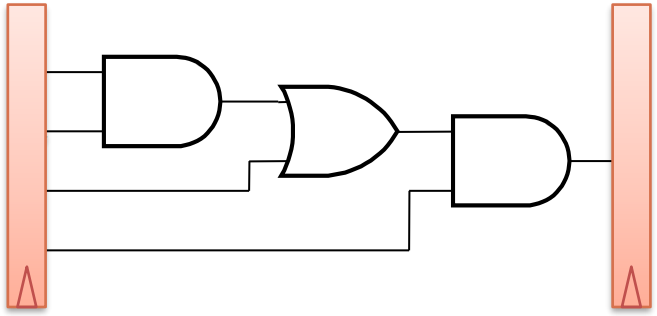
パイプライン化しない場合




パイプライン化した場合

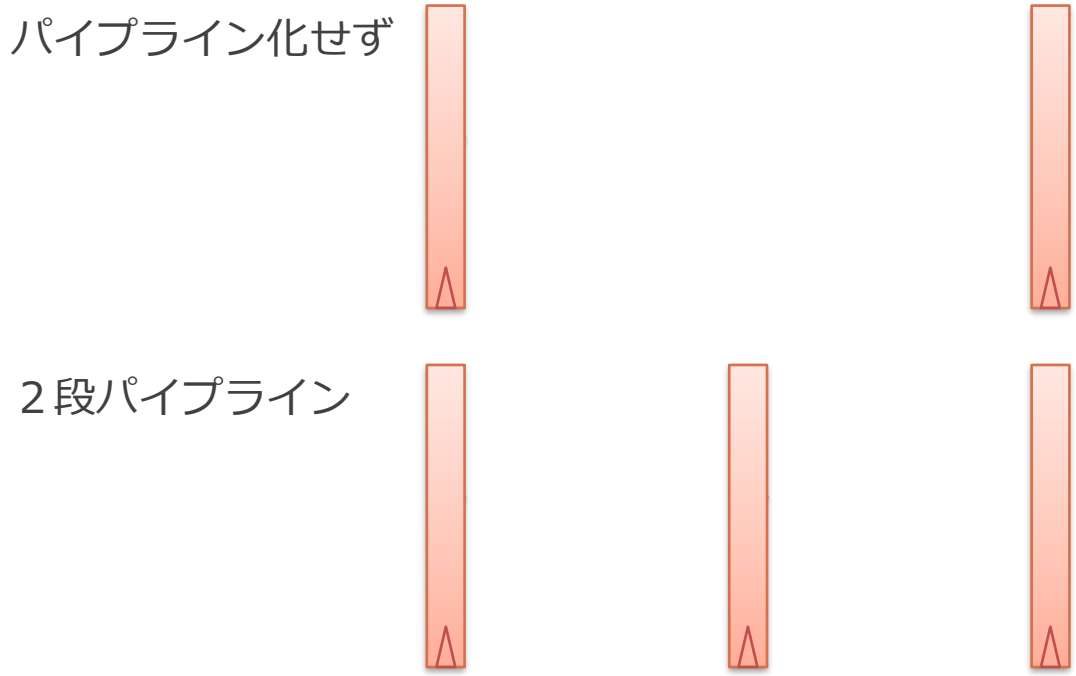


# ステージ内の信号の伝播を考える



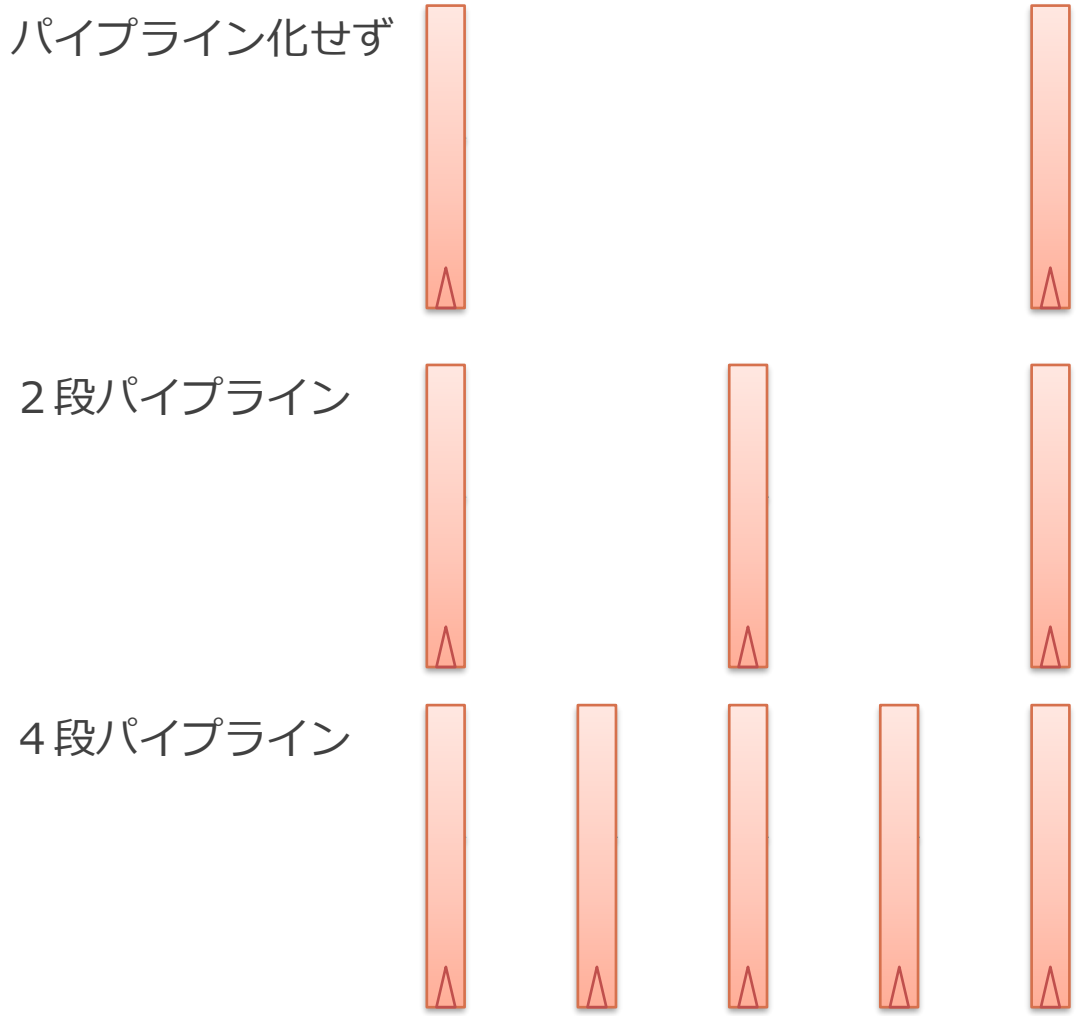
- パイプライン：ステージ：
  - 複数のパイプライン・ラッチで挟まれてた，
  - 組み合わせ回路（ゲート）
- 矢印の動き：
  - クロック開始時に，左のラッチからでた信号が
  - 組み合わせ回路を通って，伝播していく様子

# ２段にパイプライン化した場合




- クロック周波数は２倍に：
  - 各矢印の伸びる速度（信号が伝播する速度）自体は同じ
  - ２段パイプラインでは，ラッチから２回信号が出ている

# 4段にパイプライン化した場合




- クロックが４倍に
  - ４段パイプラインでは，ラッチから４回信号が出ている

# パイプライン化の限界

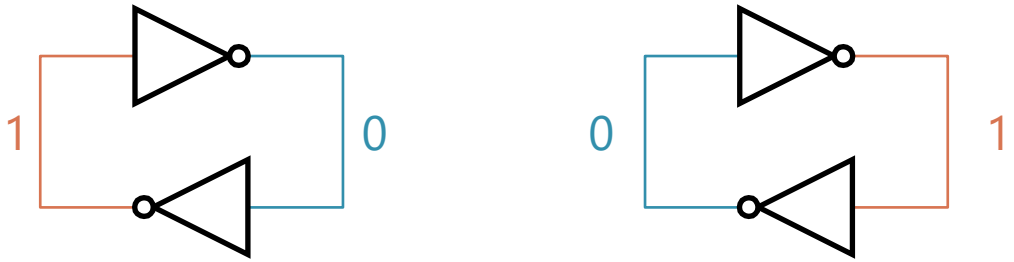

- パイプライン段数を増やしていけば，どこまでも速くなるのか？
  - ならない
- 理由：
  1. 回路的な理由による周波数向上の限界
  2. アーキテクチャ的な理由による実効性能の限界

# 回路的な理由

- 理由：
  1. D-FF 自体の遅延のため
  2. 消費電力と熱のため

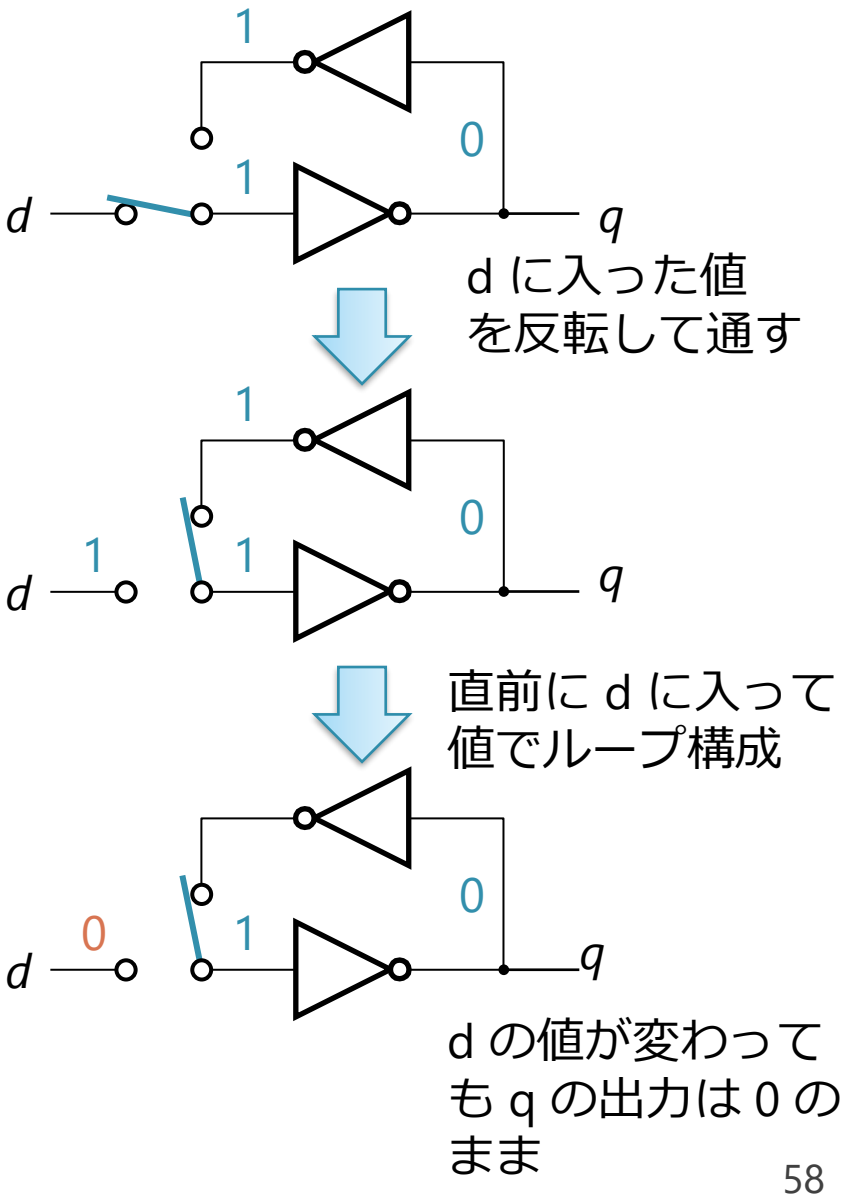
# 記憶素子の原理

- 記憶の実現方法：
  - ２つの NOT ゲート（インバータ）をループさせた回路により実現
  - 以下の２通りの安定状態がある
    - これのどっちになっているによって，1 bit の情報を記憶



# D ラッチの回路

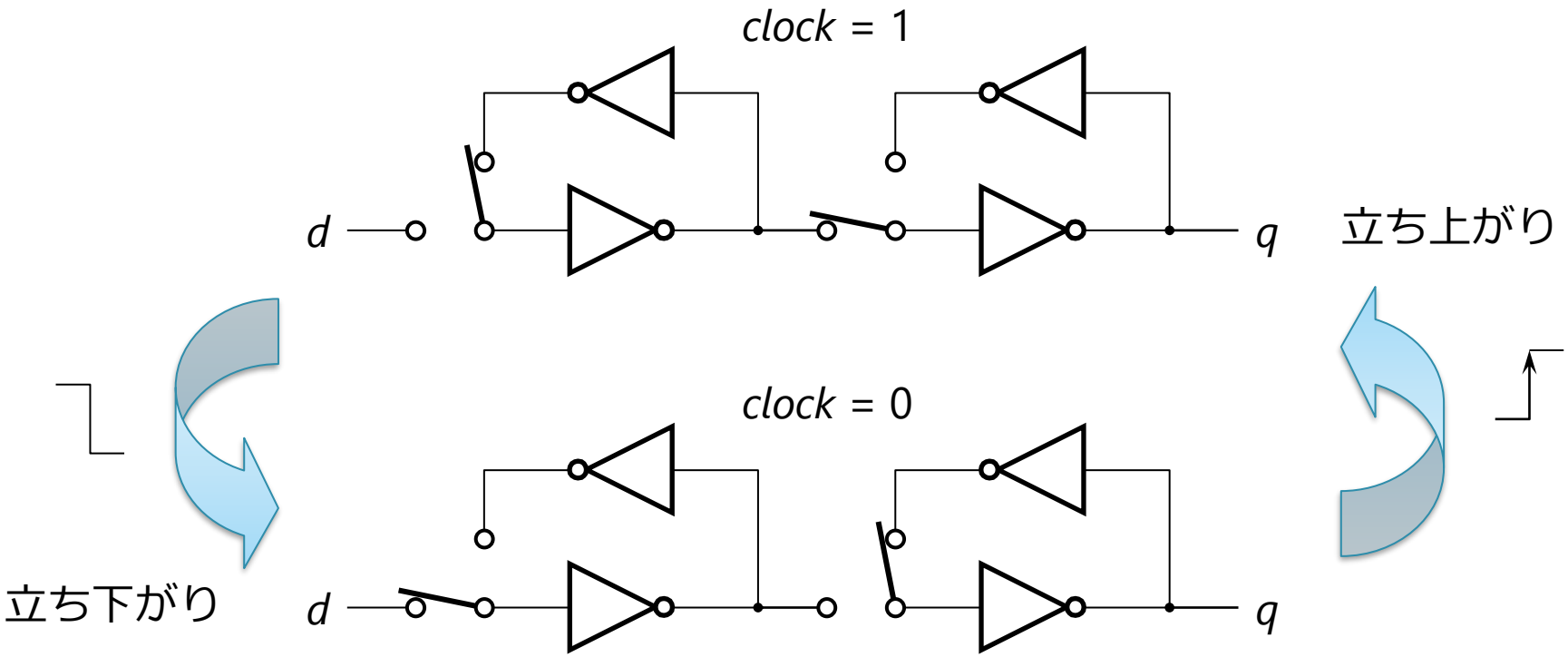
- 構造：マルチプレクサが入ったインバータのループ
  - ここではマルチプレクサを切り替えスイッチとして説明
  - クロックの立ち上がりのたびに，スイッチが切り替わる
  - *d* の値をループに取り入れ，取り入れた値が *q* から出力される



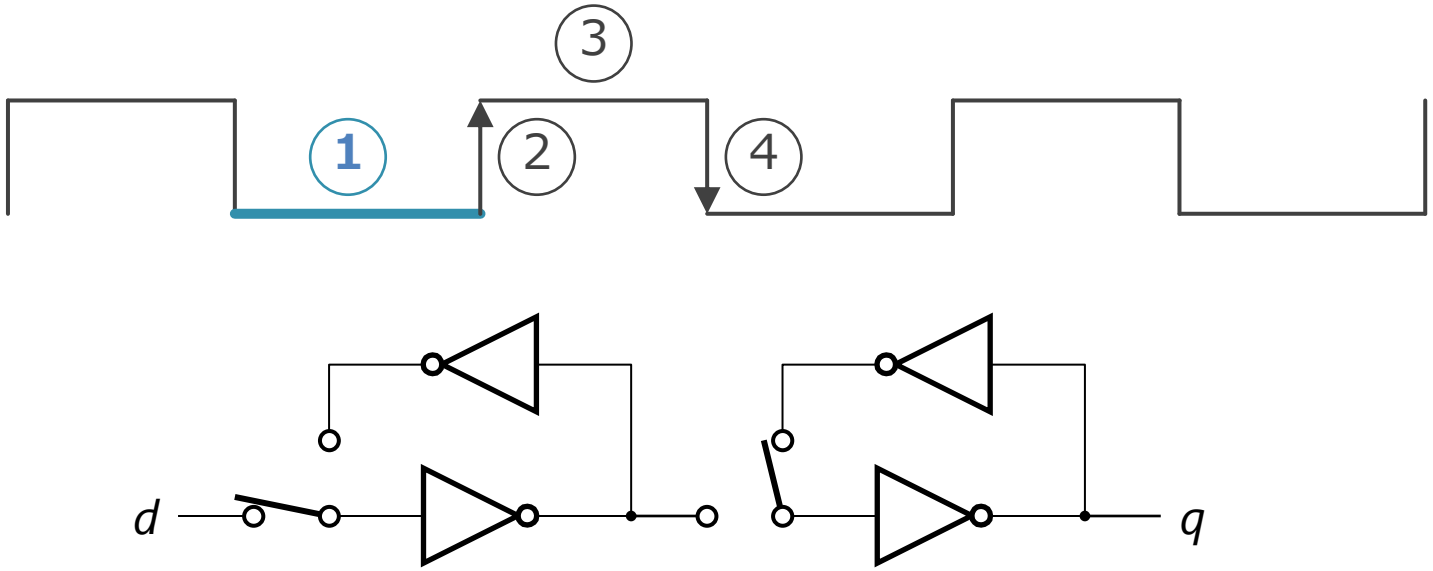
# D-FF の回路

- 構造：D ラッチを2つ繋げたもの
  - ◇ D ラッチ１つだと，半周期は d に入った値が反転して素通しするので使いにくい
  - ◇ ２つ直列に繋げて素通しの期間をなくす

- クロックの立ち上がりのたびに，*d* の値がサンプリングされる
  - ◇ その値が次のサイクルの間 *q* から出力される

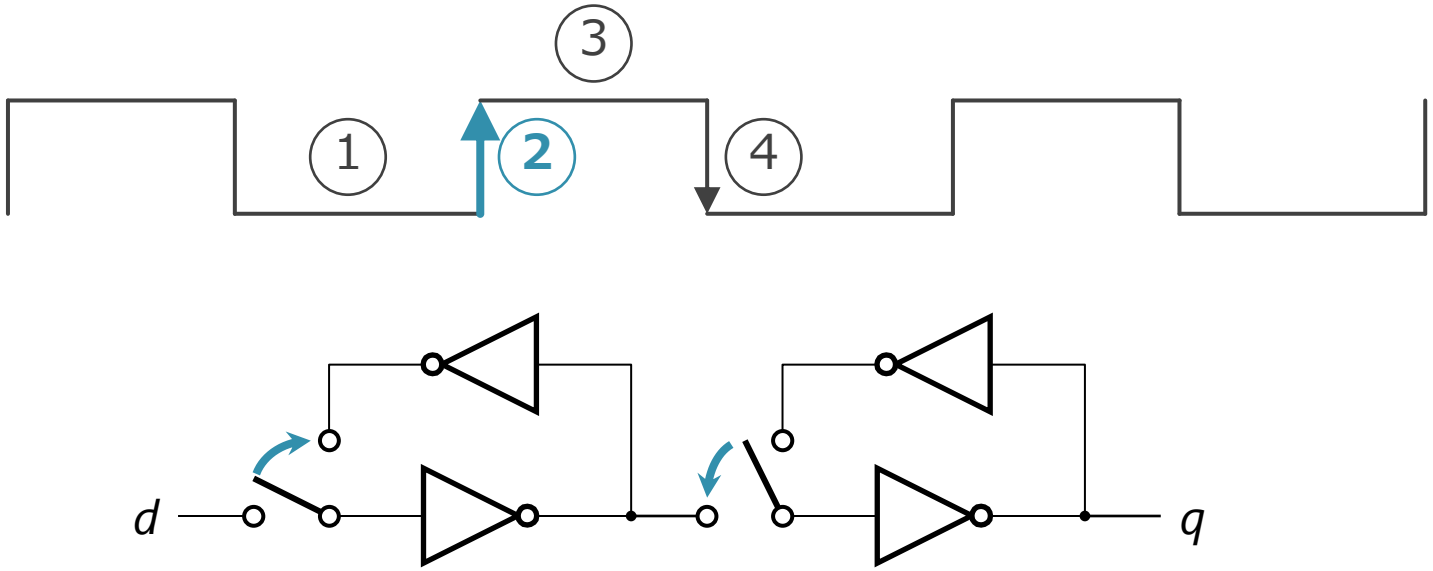


# D-FF の動作 ① クロック信号が Low にあるとき



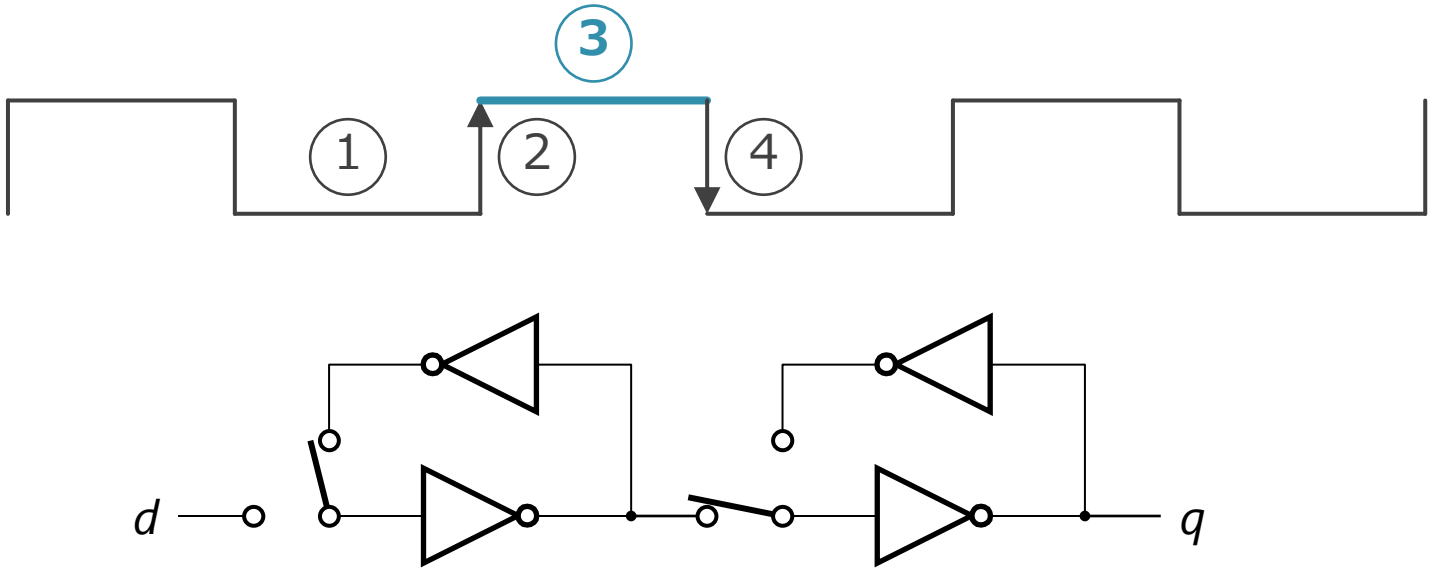

- 左側のループ：
  - ◇ $d$ の入力の変化に応じて，インバータの状態が随時切り替わる
  - ◇ 右側のループとは遮断されている
- 右側のループ：
  - ◇ ループのインバータの状態（=記憶）が $q$ に出力され続ける

# D-FF の動作 ② クロック信号の立ち上がり



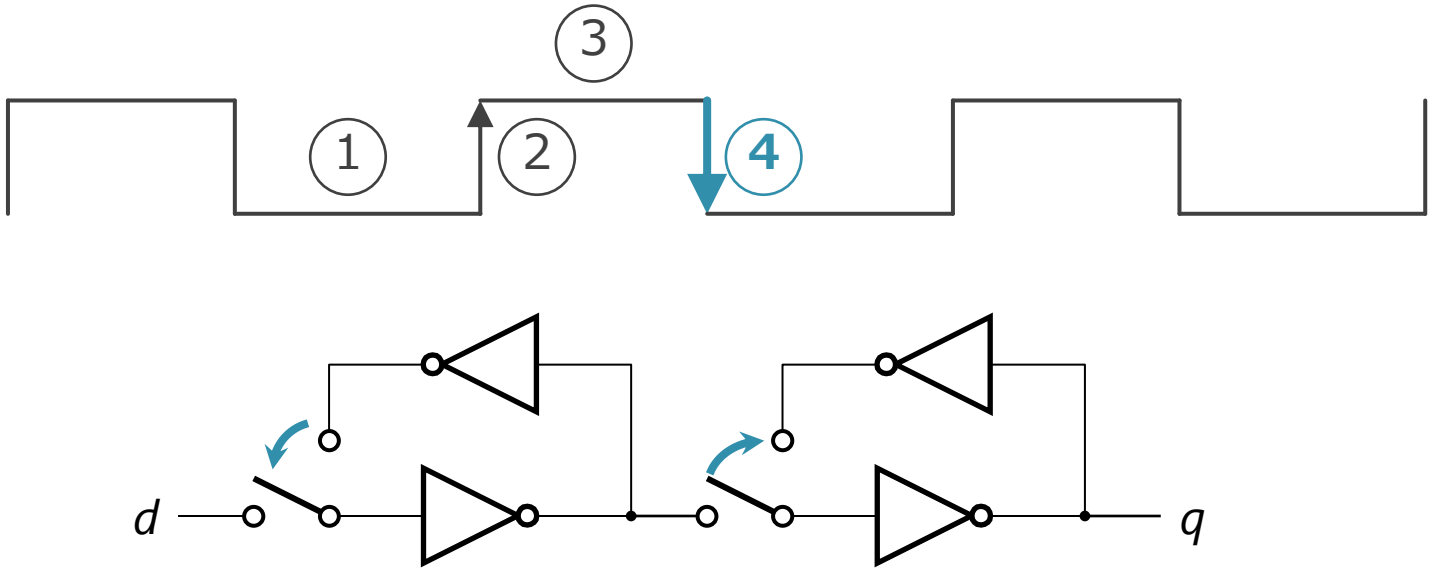

- 左側のループ：
  - ◇ *d* と遮断され，ループが形成される
  - ◇ 直前まで *d* に入力されていた信号が記憶される
- 右側のループ：
  - ◇ 左側のループと繋がり，ループが解除される

# D-FF の動作 ③ クロック信号が High




- 左側のループ：
  - ◇ クロックが立ち上がる直前の $d$ の内容を出し続ける
- 右側のループ：
  - ◇ 左側のループの出力を反転して $q$ に出力

# D-FF の動作 ④ クロック信号の立ち下がり




- 左側のループ：
  - ◇ ループが解除される
- 右側のループ：
  - ◇ 左側のループと遮断され，ループが形成される
  - ◇ それまで左側から入力された内容を出し続ける

# D-FF の遅延

- D-FF の遅延：これまでの４フェーズの動作の遅延
  - スイッチが切り替わるまでの遅延
  - スイッチが切り替わった後，
    インバータの入力に応じて出力が変化するまでの遅延
- クロック周波数を上げすぎると，これらの限界にぶつかる
  - １ステージ内の組み合わせ回路の遅延：
    インバータ換算で通常１０から２０段分ぐらい
  - なので，D-FF 自体の遅延は意外とバカにならない

# 理由２：消費電力と熱

- クロック周波数を上げる
  - → 単位時間あたりの回路全体の充放電の回数が増える
  - → 消費電力と，それによって発生する熱がそれだけ増える

1. 電力供給の限界
   - CPU のチップの端子から流し込める電流の限界
   - オームの法則：$V=IR$
     - 端子のピンの数で $R$ が決まる

2. 放熱の限界
   - 温度の上昇に，放熱が追いつかない

# まとめ

1. シングル・サイクル・プロセッサの動作
2. 上記のパイプライン化
3. パイプライン化の性能への影響

# 課題 5

- 第４回の講義資料を参考に，P型/N型リレーを使って以下を構成せよ
  - 2入力 OR ゲート
  - 3入力 NOR ゲート

2入力 OR の真理値表

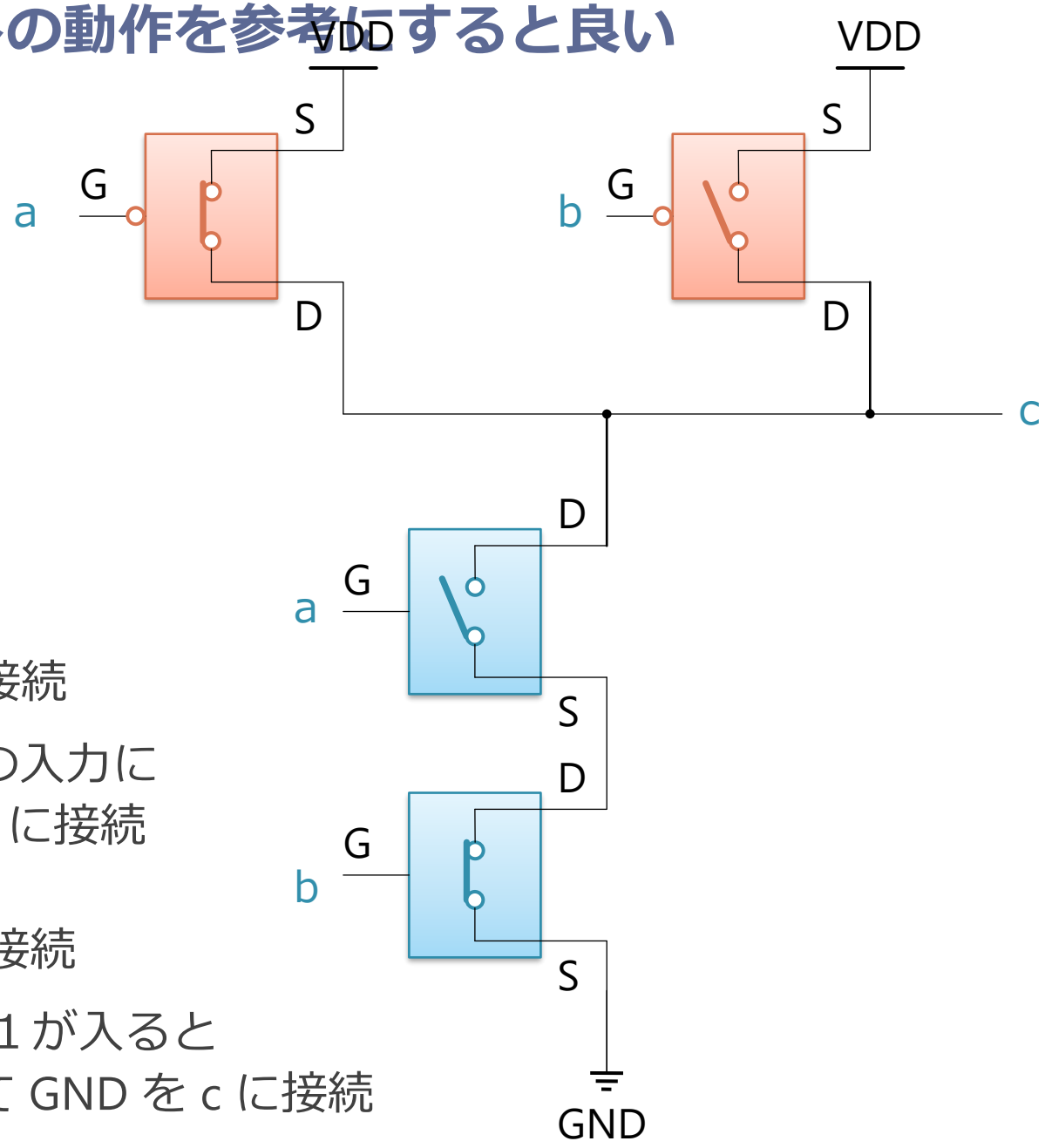
| $a$ | $b$ | $o$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

3入力 NOR の真理値表

| $a$ | $b$ | $c$ | $o$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# ヒント：
# 以下のNAND ゲートの動作を参考にすると良い

| a | b | c |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




- 上側の P 型 2 つは並列接続
  - a と bどちらか片方の入力に 0 が入ると VDD を c に接続
- 下側の N 型 2 つは直列接続
  - a と b 双方の入力に 1 が入ると 2つとも ON になって GND を c に接続

# 提出方法

- 以下を提出：
  1. 課題 5
     - 提出は Moodle の「課題 5 」のところからお願いします
     - 画像（紙に書いてスマホで写真でもよし）で提出してください
  2. 感想や質問：
     - 「感想や質問」のところに投稿してください
     - わからない場所がある場合，具体的に書いてもらえると良いです

- 提出締め切り
  - 来週の講義開始まで

- 注意：
  - 課題の出来は，ある程度努力したあとがあれば良しです
  - 必ずしも正解していなくても良いです

# 来週の講義について

- 6/11（火曜）は ZOOM によるリモートの講義の予定です
  - 出張で山梨にいる予定です
  - （本当は休講にしたいができない･･･
- URL 等は Moodle に後日掲載します

# 感想や質問

# 質問や感想など

- リレーのN型とP型のところで、なぜSとDが入れ替わっているのでしょうか。
- リレーで、SとDの違いが分からなかったのですが、Dが出力する方ということですかね？
  - CMOS では電源 or グラウンド側を source(S)，出力側を drain (D) と呼ぶため，それに合わせました

# 質問や感想など

- スライド68について、ハードウェアはどこに分類されるのですか

# リレーや真空管と比較した CMOS の利点 2

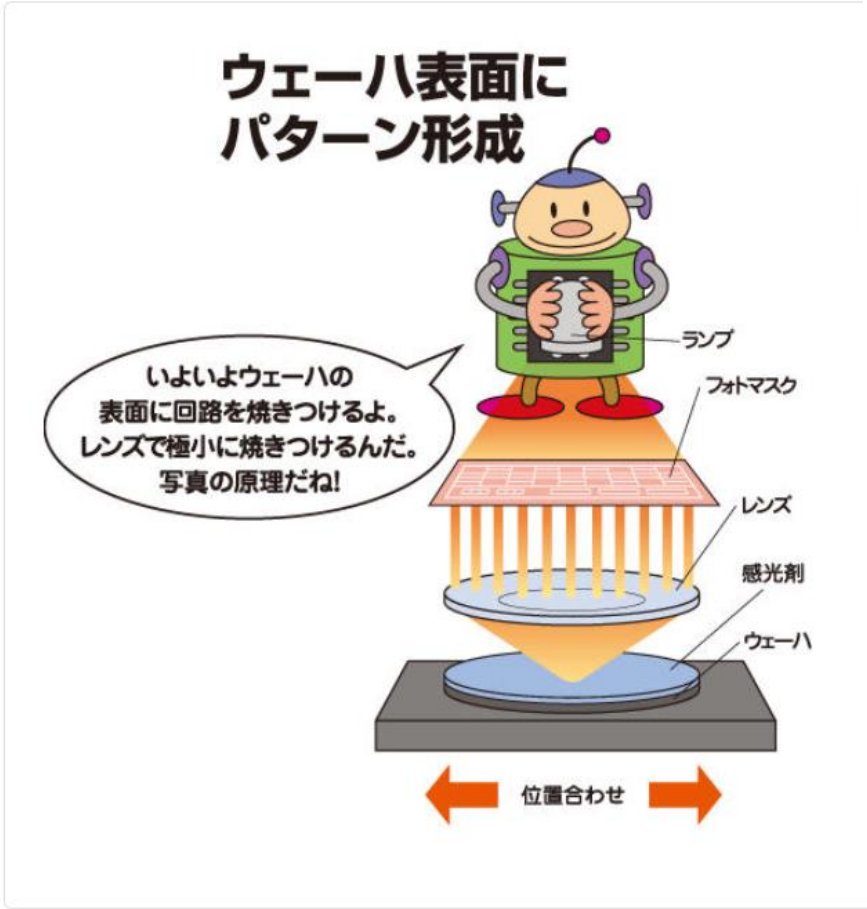
- 超省電力
  - リレーは物理的に動くので結構電気を食う
  - 真空管は常に暖め続けるのでめっちゃ電気を食う
- 壊れない：
  - リレーは物理的に動くので，かなり壊れる
  - 真空管はフィラメントがそのうち焼き切れるし，ガラスが割れる

# 質問や感想など

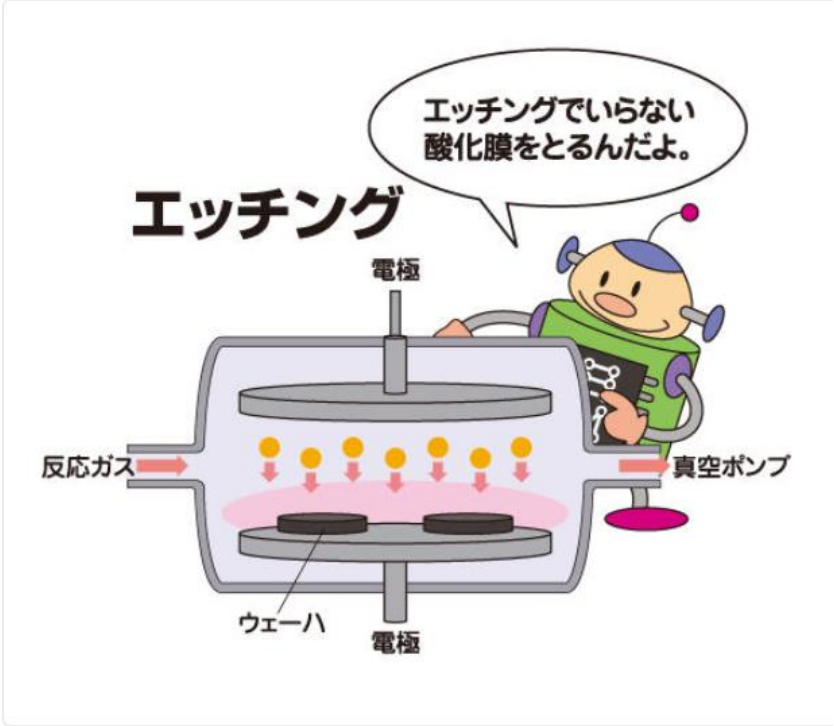
- ナノメートルのCMOSをどうやって作っているのか疑問に感じました。

# 日本半導体製造装置協会のサイトより
# https://www.seaj.or.jp/semi/proc/pre_process02.html

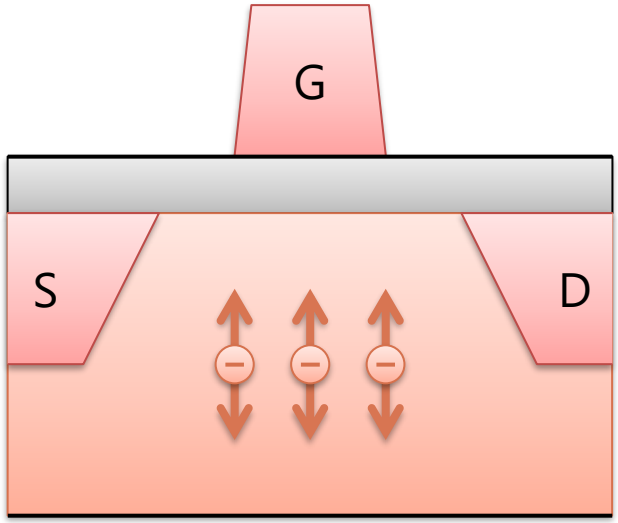
## ウェーハ表面にパターン形成




## エッチング



# 酸化・拡散・CVD・イオン注入

# 質問や感想など

- 今、0と1を物理的に表すには大きな装置を用いているイメージですが、実際にはとても小さい部品の中に何個も装置が入っているので驚きます。どうやって装置を小さくしているのか気になりました。昔のコンピュータが今とは比べ物にならないほど大きかったのは、この装置の大きさに関係しているのでしょうか？

# 質問や感想など

- emacsを使っている人が一生emacsを使うという話を聞いて驚愕しました。でも、情報科学科の教員方の様子を見る限り否定できないです。

# 質問や感想など

- CMOSはNMOSとPMOSでできていて、NMOSとPMOSでは低電位で通すか高電位で通すかとgateを何にした時にonになるかが違うので、電荷を動かすことでonoffを変えたりして扱っているというなんとなくの理解なのですがあっていますか？

# 質問や感想など

- また余談ですがこれまでのプログラミングの授業ではLABELを扱っていないため、使い方に慣れていないです。
- 課題のLABELは、いつものC言語にはないのでたまに混乱します。
  - C 言語における LABEL は，各行に行番号がついていて，それを読み替えたものだと考えても良いです
  - たとえば，LABEL: が１０行目にあった場合，goto LABEL は１０行目に飛ぶと言う感じ

# 質問や感想など

- コンピュータの内部構造を勉強する度に、単純なことの積み重ねで複雑な処理をこなしていてすごいなぁと感動します。あと、一年生の時に数理基礎論という授業で学習した論理回路が日の目を浴びていて(?)嬉しくなりました。

# 質問や感想など

- コンピュータの元々の部分に物理がたくさん使われていて興味深かったです。ON/OFFだけでこんなに高性能なことができると最初に思いついた人や、超効率的なCMOSの形式を編み出した人すごいなと感じました。

# 質問や感想など

- NANDの接続方法の回路を見た瞬間、複雑そうで脳が理解を拒否しようとした。考えついた人すごいなぁと思った。

# 質問や感想など

- リレーが普通の充電機を利用していて、CMOSが電荷を利用しているのでCMOSの方が移動距離が圧倒的に短くて速度が速いということですかね？

# 質問や感想など

- マイクラでコンピュータを作ってみたいのですが、設計図が載っている良いサイトなどはありますか？
  - 「レッドストーン回路」で検索すると色々出てくると思います

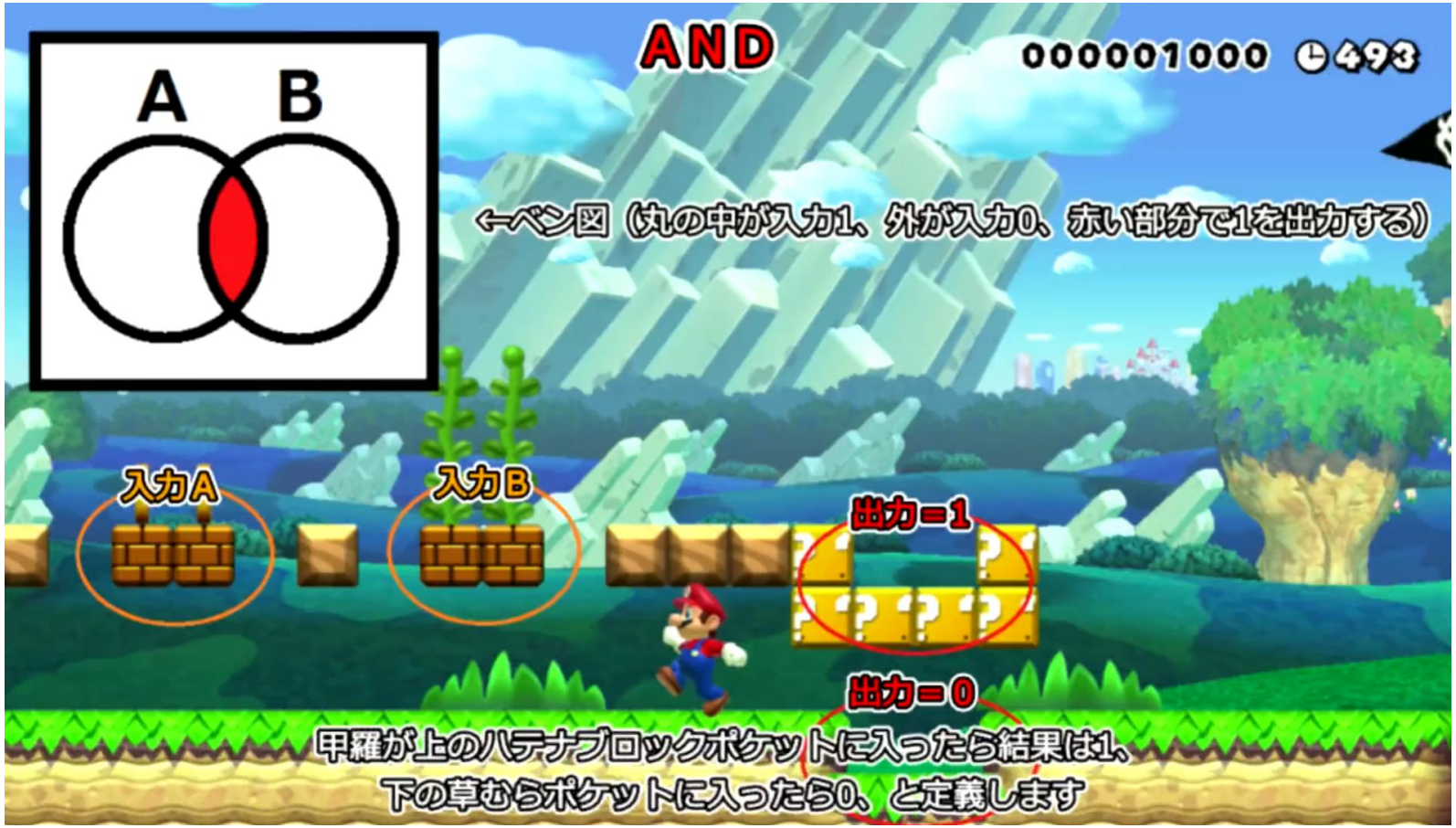
# 質問や感想など

- NANDとNOTを組み合わせてAND回路を作るのではなく、直接（？）AND回路をリレーで作ることはしないのか気になりました。レッドストーン回路でAND回路を作れるようなので作れないことはないと思うのですが、あえて直接AND回路を作ることはしないのでしょうか。

# 質問や感想など

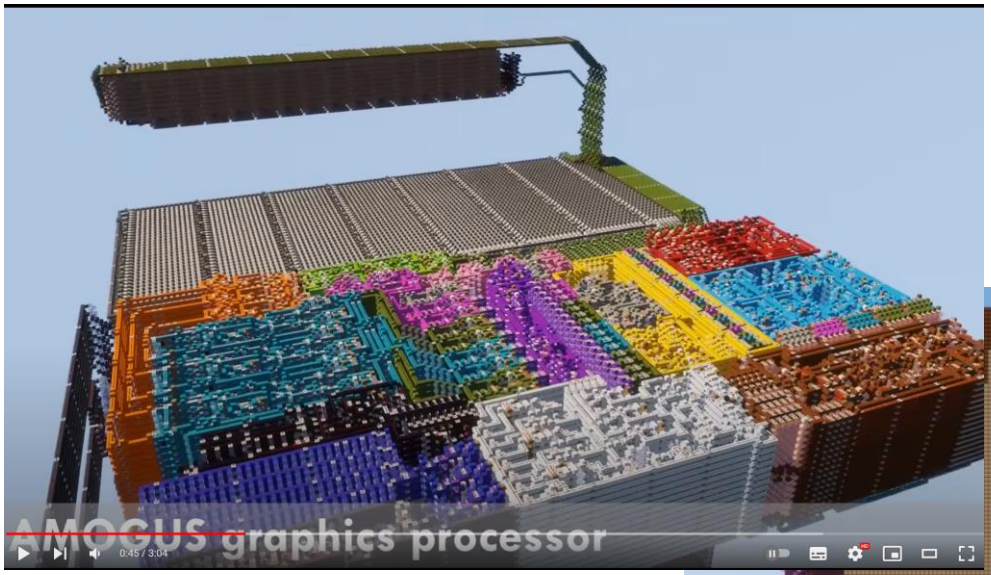
- 論理回路を組み合わせて計算機の処理が行われていることを改めて実感できた。また、マインクラフトで入力と出力を制御できることに気づいた人はすごいと思った。他のゲームでもコンピュータが作れるかもしれないと考えた。

# マリオメーカーの場合




- https://www.youtube.com/watch?v=bpgiUUFY73g より

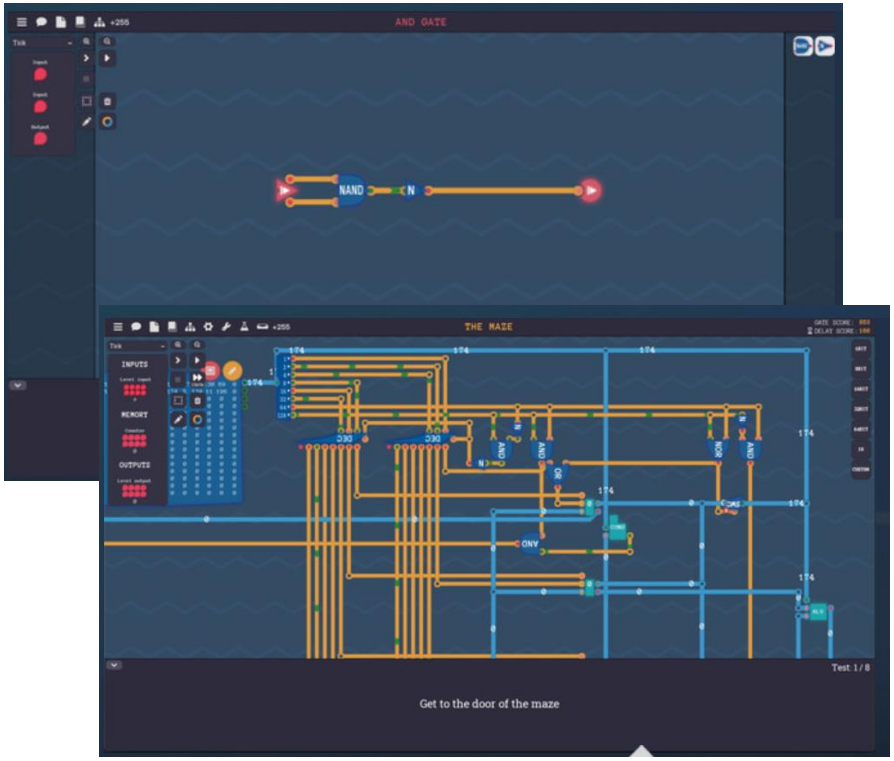
# マインクラフトの中でマインクラフト

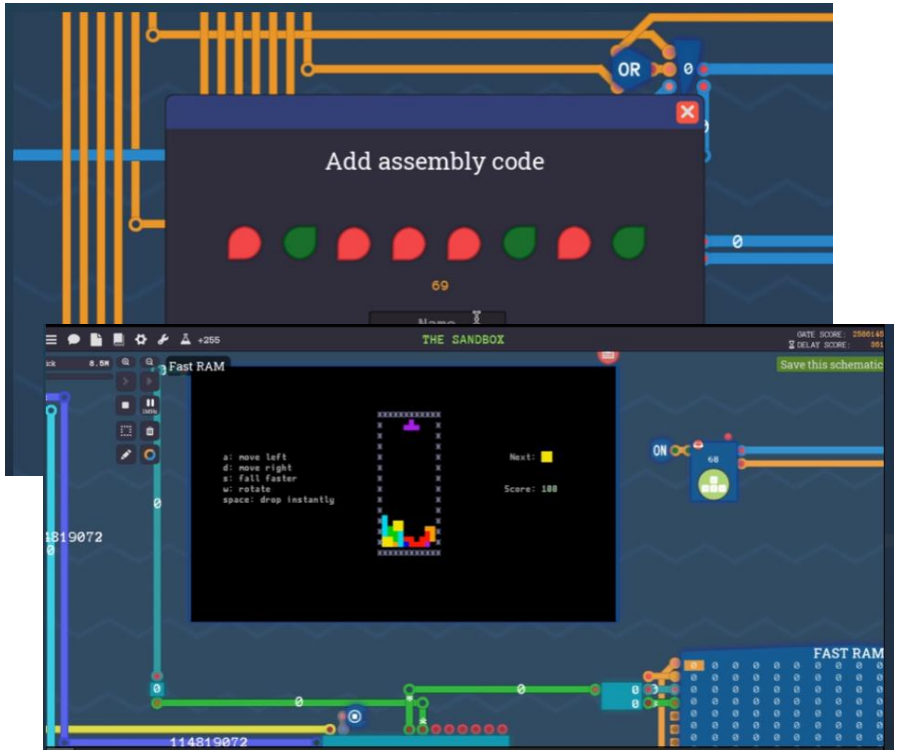




- https://www.youtube.com/watch?v=-BP7DhHTU-I より

# Turing Complete と言うゲーム
## NAND から初めてテトリスとかまでを作る




- https://store.steampowered.com/app/1444480/Turing_Complete/?l=japanese より