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

質問や感想など

- 論理回路について難しくてあまりわからなかった
- 訳分からなくなってきたので復習しようと思います。
- 課題があまりわかりませんでした解説をお願いします！！

質問や感想など

- 真空管でもコンピュータが作れるという話がありましたが、現在真空管はどのようなものに使われているのでしょうか？それとも、もう実用的なものには使われていないのでしょうか？

質問や感想など

- コンピュータのCPUなどには、CMOSが大量に乗っているということで合っているでしょうか。

質問や感想など

- MOSは工学系の話ですか？

質問や感想など

- スイッチのON・OFFで1と0を表していることや、論理回路の作り方など、これまでバラバラに学んできたことが繋がったように感じて面白かったです。

質問や感想など

- PCにはメモリ内の命令の番地が保持されていて一つの命令が実行されるたび更新されると認識しているのですが、資料の例によって命令の番地が1ずつ増えるものと4ずつ増えるものがあり、そのどちらについてもジャンプ命令がないためPCの増え方が1ずつまたは4ずつで固定されていると辻褄が合わないように思いました。PCの更新の具体的な挙動を教えていただきたいです。

質問や感想など

- NOTゲートとNANDゲートがP型とN型で作れれば、ANDやORも作れるとのことでしたが、NANDゲートにNOTゲートをつけてANDを作るのは、話だけ聞くと二度手間な気がしました。NOTとNANDを使わずにANDを作ることはできないのでしょうか。

質問や感想など

- 参考書籍の中で「抽象化」という言葉が出てきました。コンピュータを層として分割して設計ができるような概念を作ること、例えば電子回路に対する二進数や高水準言語に対するアセンブリ言語はコンピュータにおける抽象化と言える。このような認識であっていますか。

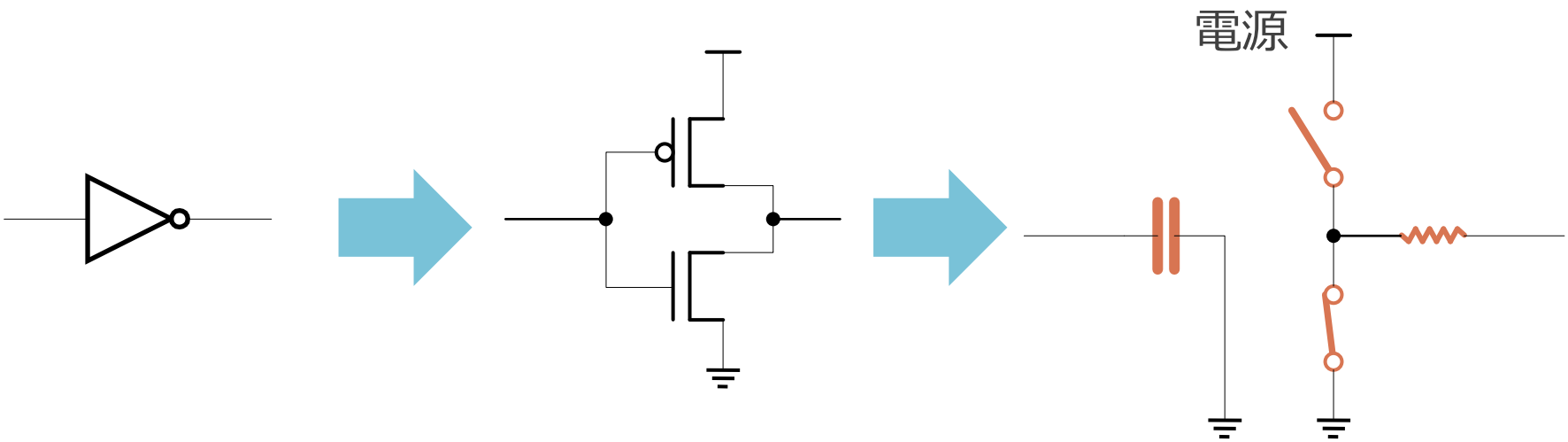
質問や感想など

- CMOS以外のリレーや真空管、レッドストーンの良さはそれぞれにありますか？
- CMOSの欠点は何か気になりました。（値段が高いとかかな、、と思ったのですが、目立った欠点がなさそうだったので気になりました。）

質問や感想など

- 遅延のところがあまりわかりませんでした。ゲートがそもそも何かわからないです。資料の図のオレンジ色の方のコンデンサの役目がないように思ったのですが何のためにあるのですか。
  - 「ゲート」は AND や OR のような論理関数を実現した回路のこと
- 等価回路で抵抗が必要な理由がわかりませんでした。

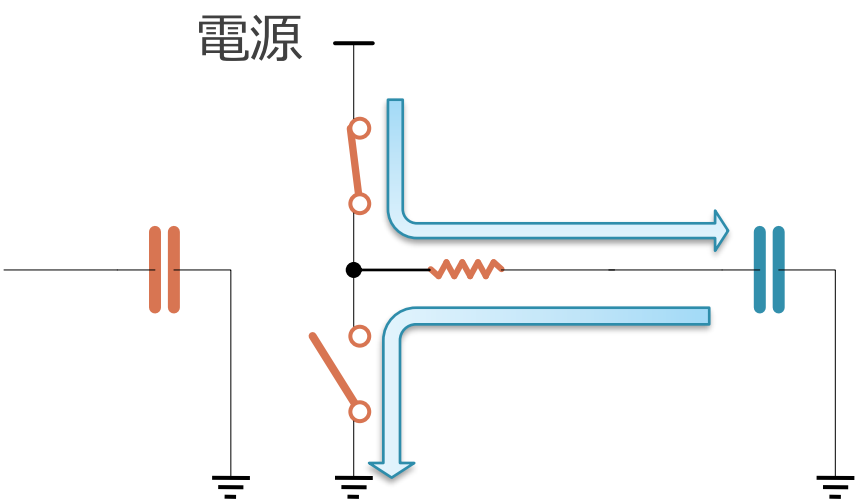
CMOS ゲートの等価回路



- 抵抗 & コンデンサと，連動したスイッチによって表せる
  - コンデンサに充電：下のスイッチがON
  - コンデンサを放電：上のスイッチがON

- コンデンサ：
  - 電荷を溜めることが出来るもの
  - 電子を引き寄せてチャネルを作る = 充電している

消費エネルギー



- 消費エネルギーは，主にコンデンサへの充放電で消費される
  - 消費エネルギーは電圧の二乗に比例：$E = CV^2$
  - 電荷 $Q = CV$ が，電圧 V の分だけ電源から GND へ移動するから
    - 忘れた人は高校の物理の教科書を読もう
- 他にリーク電流と呼ばれるものによる消費もある
  - スイッチが一部ガバガバなので，常時多少漏れてる

質問や感想など

- 周波数が高いと電圧が上がる（逆も然り）という現象がなぜ起こるのかの理解が難しかったです。この高速と低速という話は携帯の月に使用できるギガ数に到達してしまったら低速になってしまうといったような高速・低速の話と同じなのか気になりました。

質問や感想など

- 前回の感想を入力し忘れたのですが、期限過ぎると入力もできなくなってしまうのを変えることは可能でしょうか。

質問や感想など

- 高校生の頃などは、電卓がどうなっているのか想像もつきませんでした。でも、こんな簡単な（きっと簡単ではないけど）論理回路だけでできてしまうんだなと感動しました。

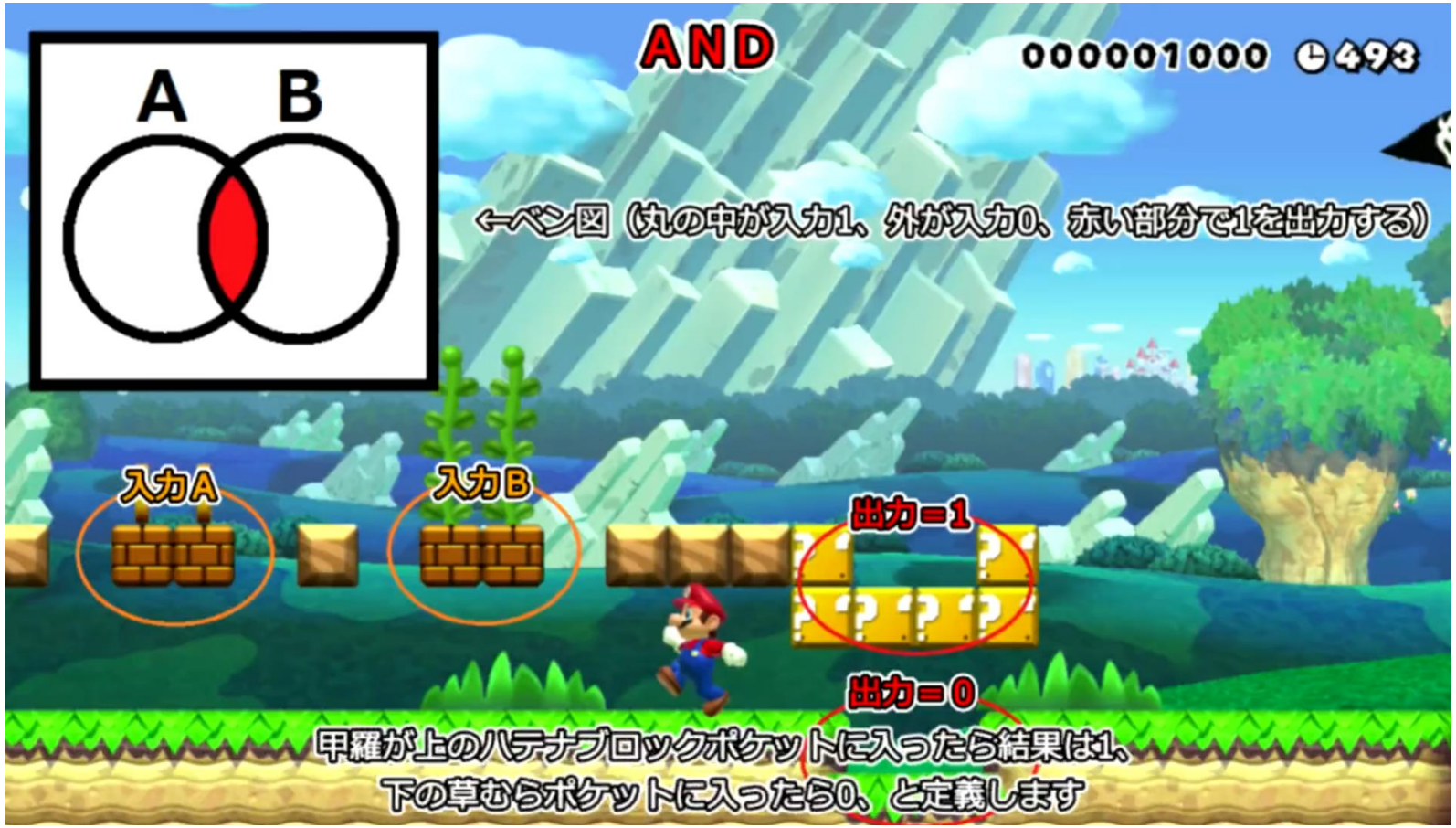
質問や感想など

- 高校で物理を学習していてなんの役に立っているのかわかっていなかったけど去年学習した論理回路にも使われていたのかということを学べて驚くだった。遅延の違いとは処理の速さのことですか？

質問や感想など

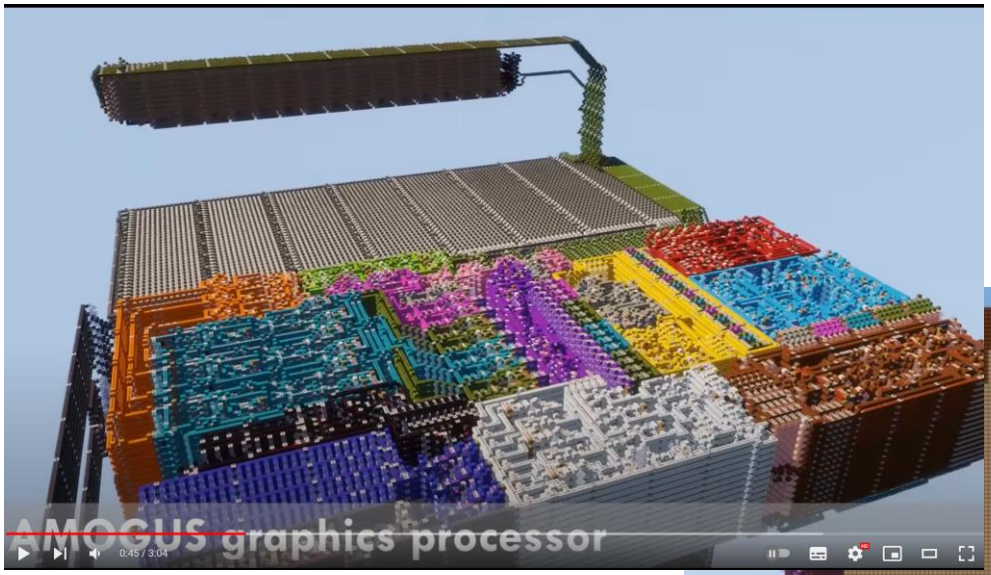
- 有効と無効が選べてつなぐことができる場合にゲーム内でも計算機を作れるのなら、他のゲームでも作れそうだなと思いました。
- さらに、マリオのコースを自由に作れるゲーム「スーパーマリオメーカー」でAND,OR,NOTの論理回路を作り、計算機を作成した人もいるようです。

マリオメーカーの場合



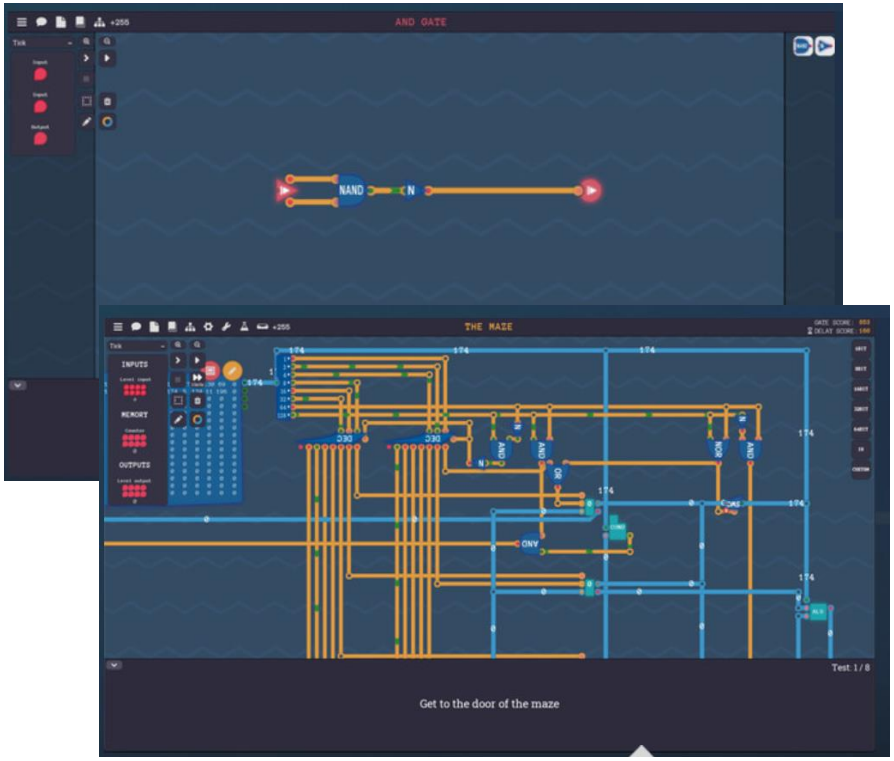
- https://www.youtube.com/watch?v=bpgiUUFY73g より

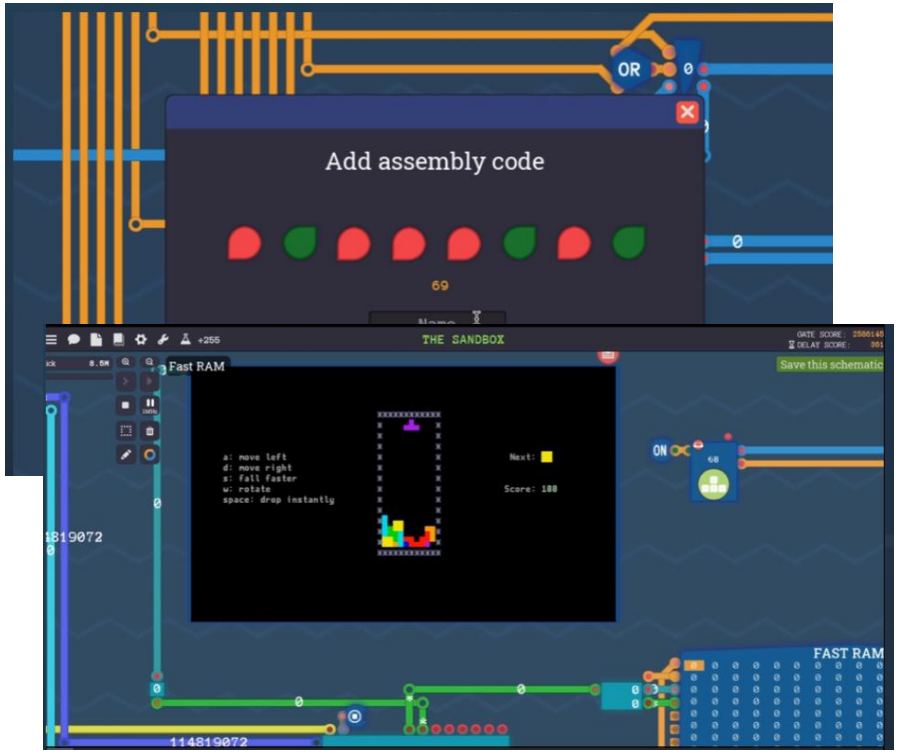
マインクラフトの中でマインクラフト



- 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

課題の解説

課題 4

- 第１回の講義資料を参考に，１０から０までを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: }
```

C 言語

- C 言語

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

```
1:      i = 10;
        0x400: li 10    → A    // レジスタ A に 10 をいれる
        0x404: li 0x0f4 → B    // B に 0x0f4 (i の番地) をいれる
        0x408: st A → (B)      // A を (B) にかきこむ（= i を更新）
2: LABEL:
3:      i = i - 1;
        0x40C: li 0x0f4 → B    // B に 0x0f4 (i の番地) をいれる
        0x410: ld (B)   → A    // (B) を A に読み込む（= i 読み込む）
        0x414: sub A,1  → A    // A から 1 を引く
        0x418: st A → (B)      // A を (B) にかきこむ（= i を更新）
4:      if (i >= 0)
5:         goto LABEL;
        0x41c: li 0 → B         // B に 0 を読み込む
        0x420: b A >= B, 0x40C // 条件がなりたっていたら LABEL に
```

アセンブリ言語（メモリを使わない別解）

```
1:      i = 10;
        0x400: li 10    → A     // レジスタ A に 10 をいれる
4:      if (i >= 0) で使う 0
        0x404: li 0 → B         // B に 0 を読み込む
2: LABEL:
3:      i = i – 1;
        0x408: sub A,1  → A    // A から 1 を引く
4:      if (i >= 0)
5:         goto LABEL;
        0x40C: b A >= B, 0x408 // 条件がなりたっていたら LABEL に
```

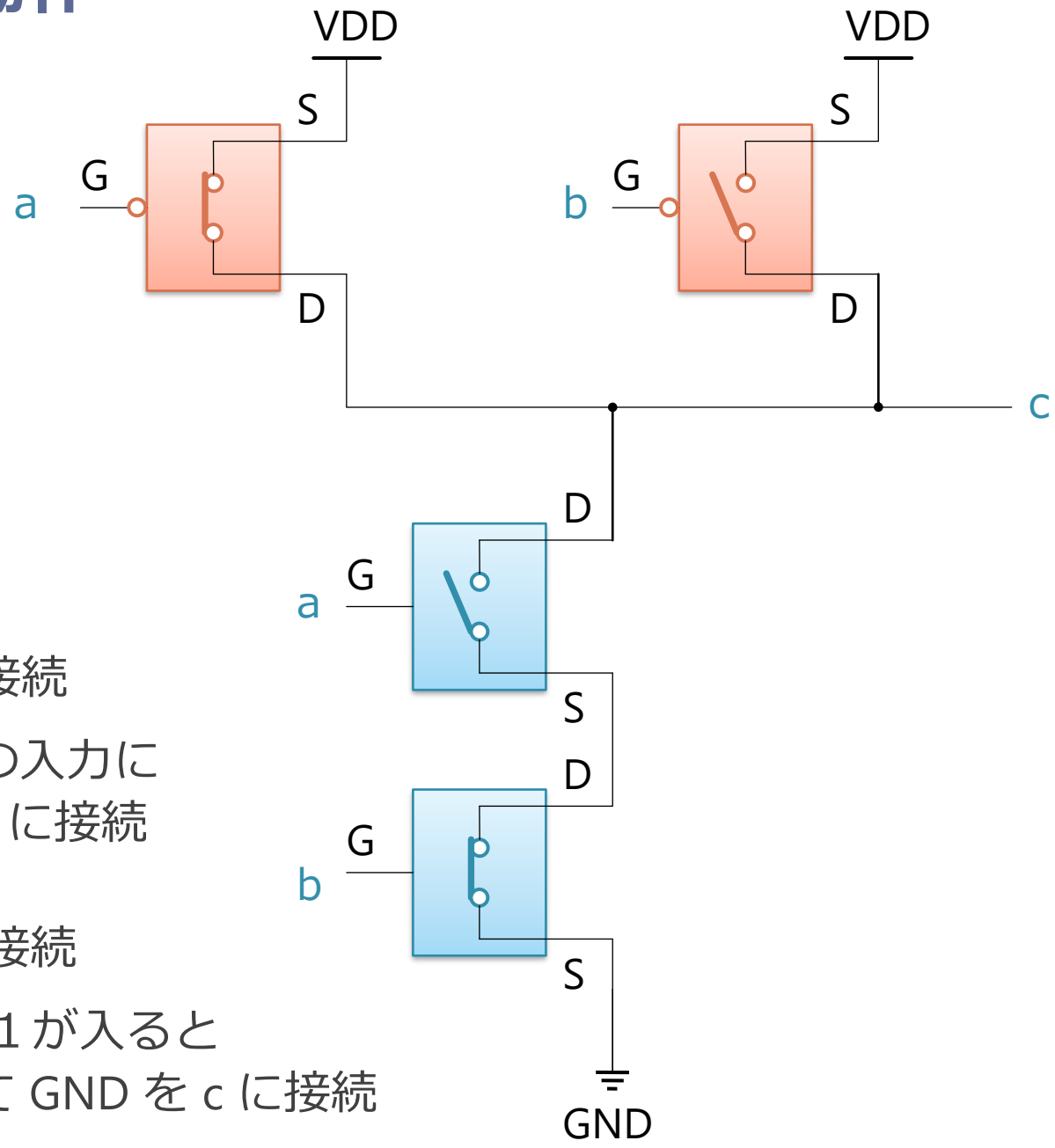
前回の振り返り

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

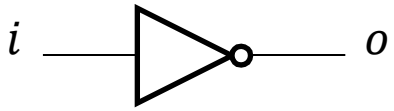


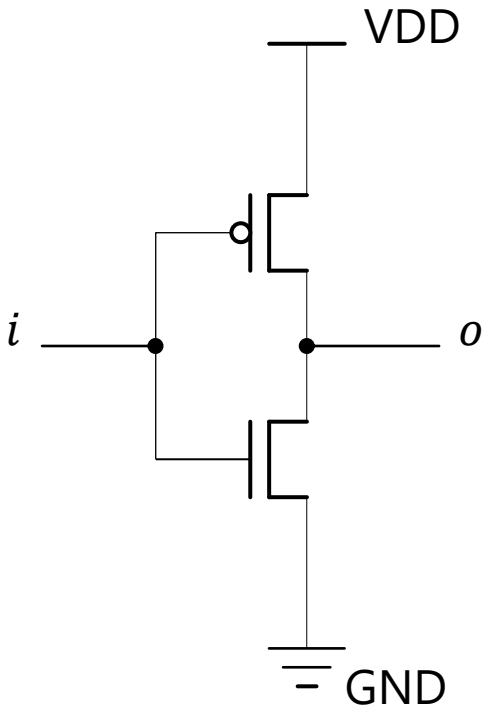
- 上側の P 型 2 つは並列接続
  - a と bどちらか片方の入力に 0 が入ると VDD を c に接続
- 下側の N 型 2 つは直列接続
  - a と b 双方の入力に 1 が入ると 2つとも ON になって GND を c に接続

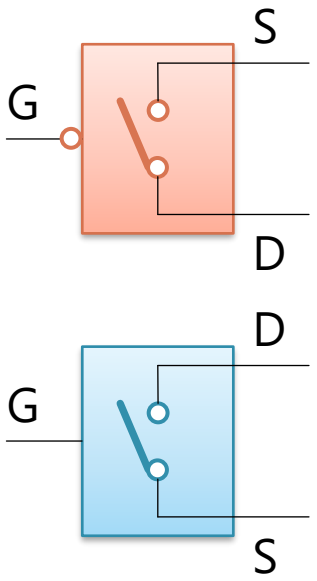
CMOS: Complementary Metal–Oxide–Semiconductor

- 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

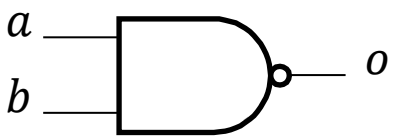
NOT ゲートの例





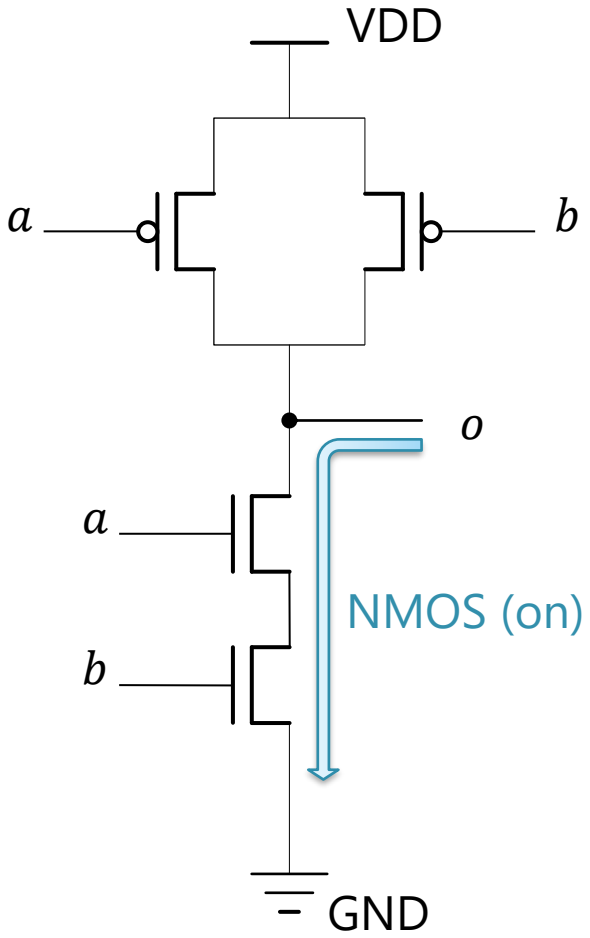


NAND の例



$o = (a \cdot b)'$

a	b	o
0	0	1
0	1	1
1	0	1
1	1	0



- a と b 共に 1 （高電位）の場合
  - PMOS（上側）は，off
  - NMOS（下側）は，双方 on = GND と繋がる
- 出力が 0（低電位に）

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

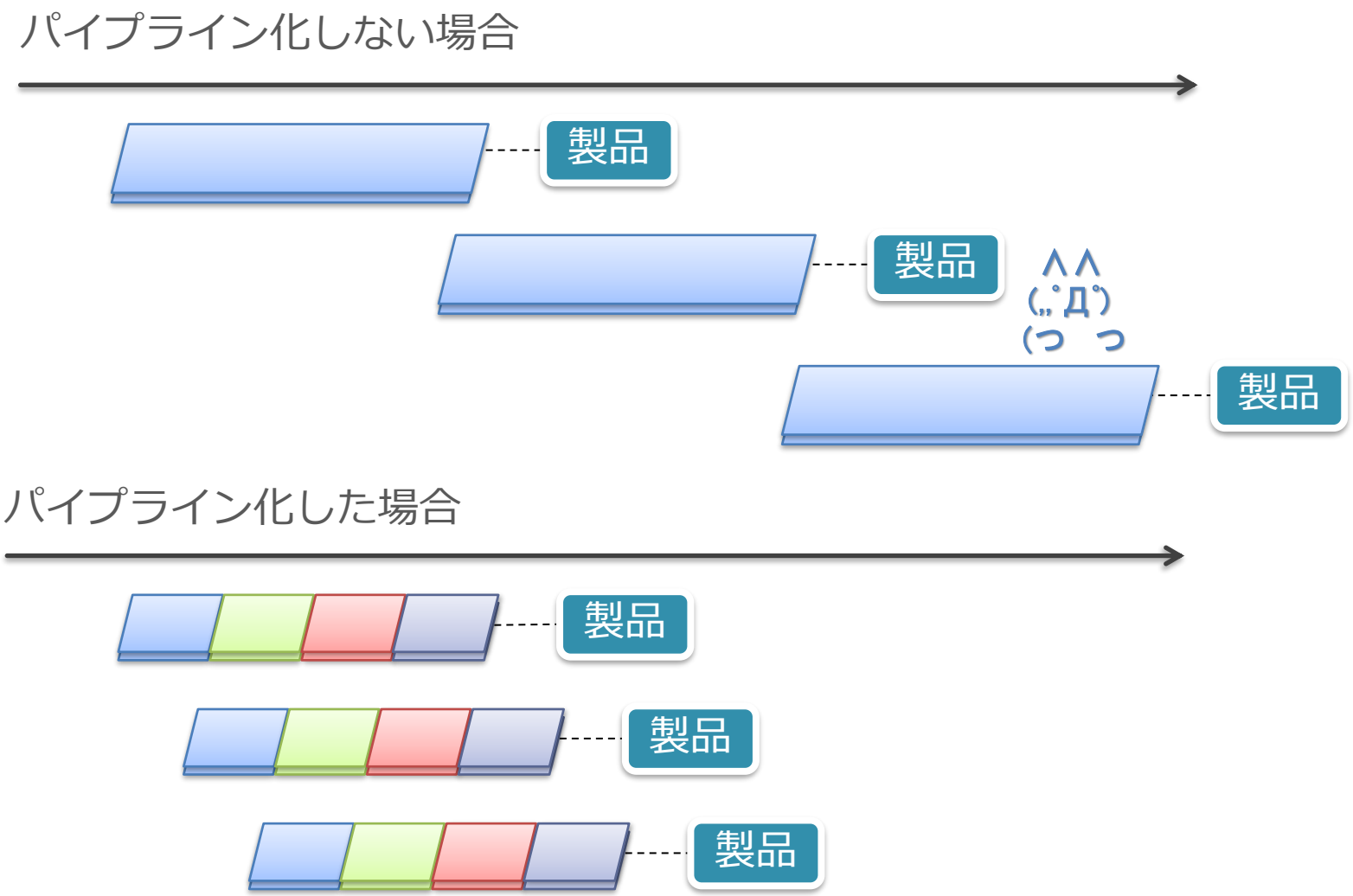
導入：工場のラインを考える

- 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

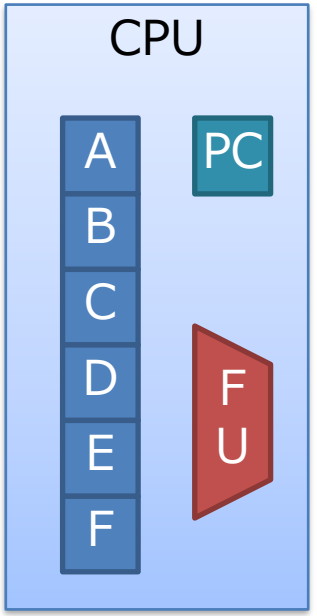
1. シングル・サイクル・プロセッサの動作
   - 全ての命令の処理が１サイクルで完結
     - これまでに説明していたプロセッサの動作と同じ
   - パイプライン化を前提とした，より詳細なものを使って復習
2. 上記のパイプライン化
   - 具体的にどうパイプライン化するか
3. パイプライン化の性能への影響

シングル・サイクル・プロセッサの動作

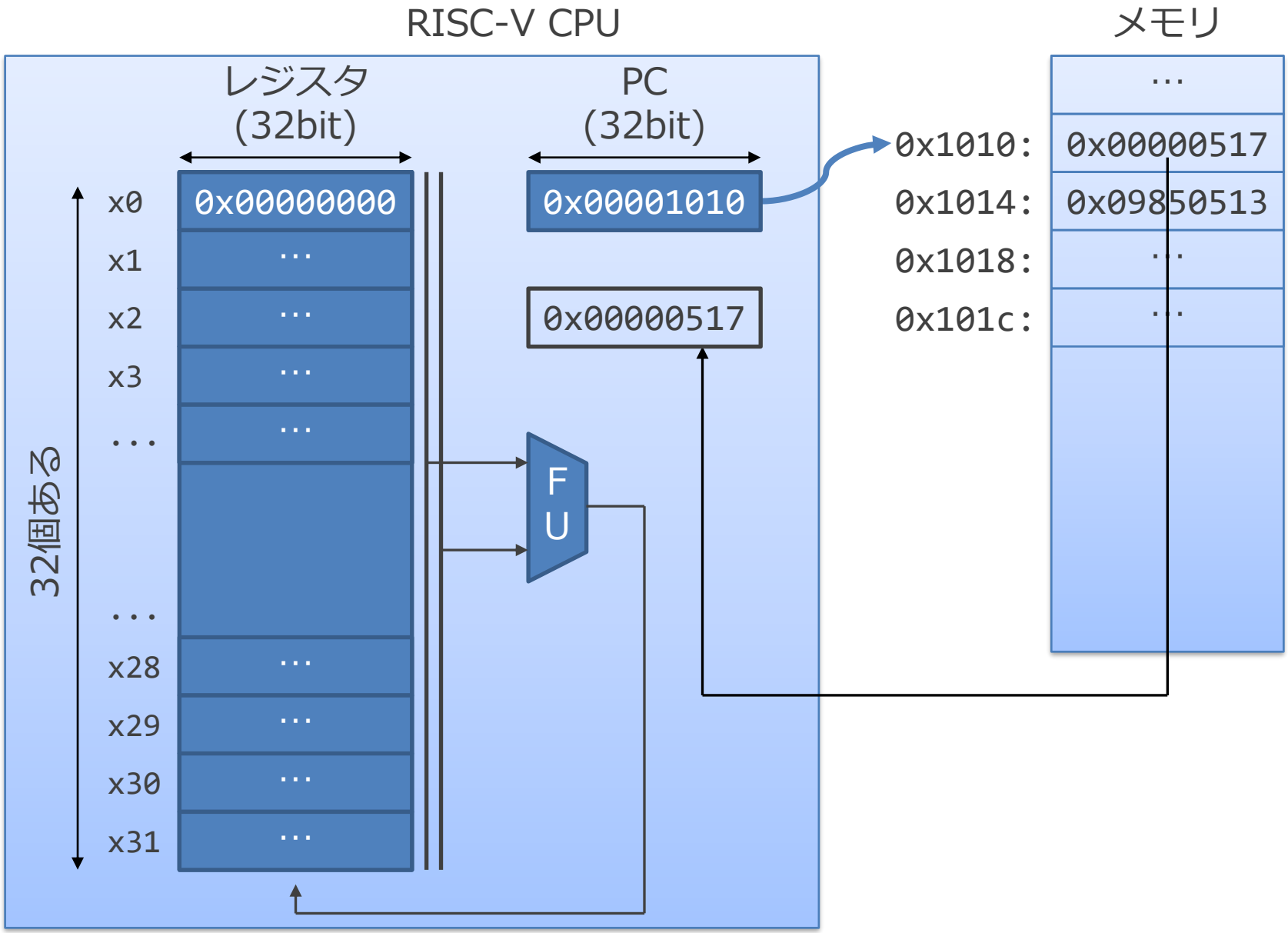
もくじ

これまでに説明した CPU のイメージ

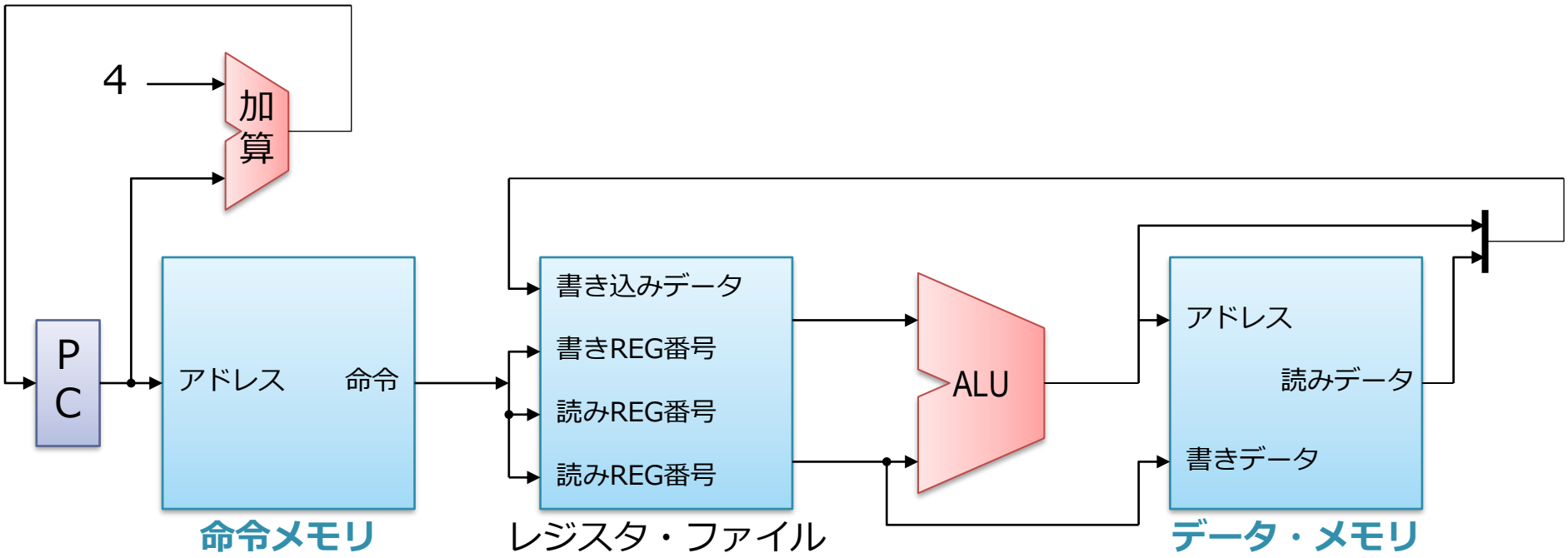
- コンピュータの心臓部
  - メモリから命令を読み出し，計算する
- 構成要素：
  - 演算器（FU: Functional Unit）
    - 加算器や AND 演算器など
    - 指示された種類の演算を行う
  - レジスタ・ファイル（右図では A,B,C...）
    - メモリと同様にデータを記憶する
      - 位置を指定して読み書きする
    - CPU の演算は，このレジスタ上でのみ行う
  - PC（Program Counter）
    - 現在見ている命令のアドレスを記憶している場所



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

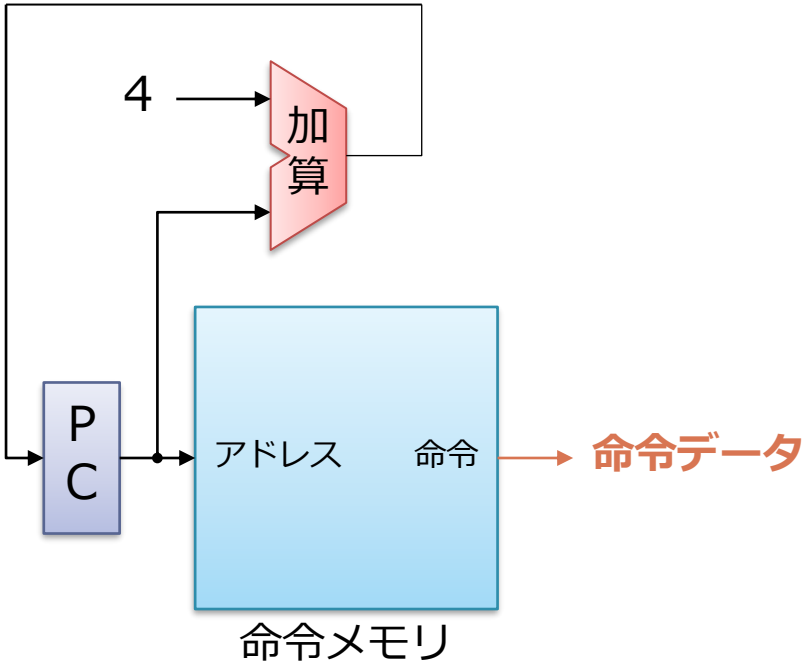


- 以前説明したものとの違い：
  - メモリが命令メモリとデータメモリに別れている
  - 算術 & 論理演算，ロード，ストアのみを実行可能
    - 分岐とジャンプは，簡単のために今は考えない

1命令の実行フェーズ

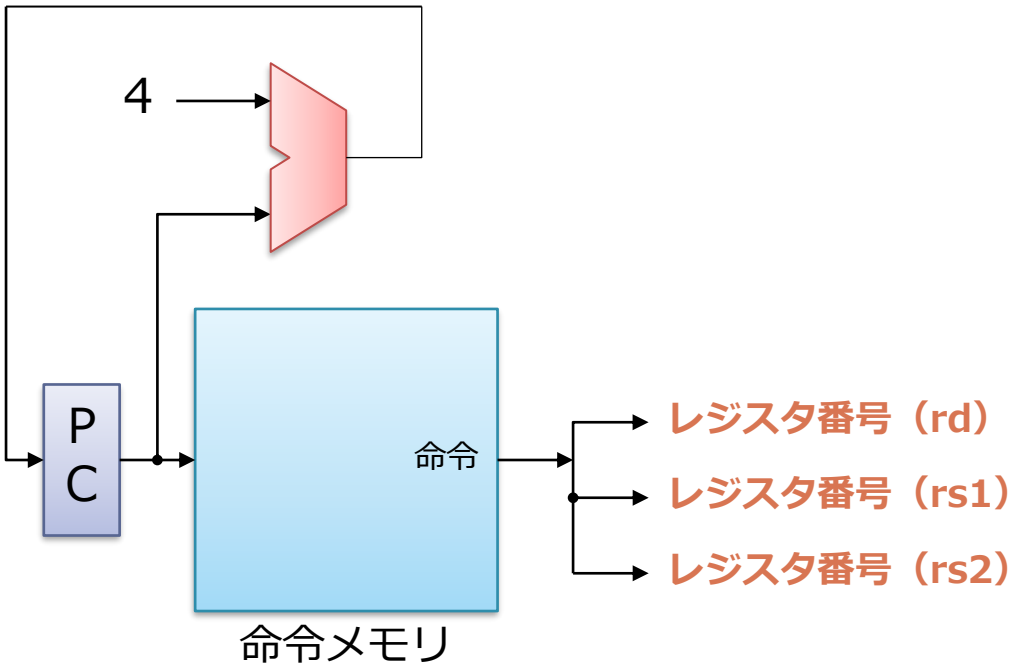
- 実行フェーズ
  1. フェッチ
  2. デコード
  3. レジスタ読み出し
  4. 実行
  5. レジスタ書き戻し
- RISC-V の加算命令を実行する流れをざっとみる

命令フェッチ

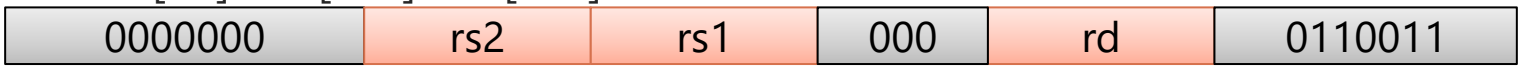


- 命令メモリから命令を読み出す
  - 命令メモリを順に読んでいくため，PC は毎サイクル加算される
  - 足している４は，RSIC-V では命令の幅が４バイトだから
  - 基本的に，この部分はどの命令でも変わらない

命令デコード

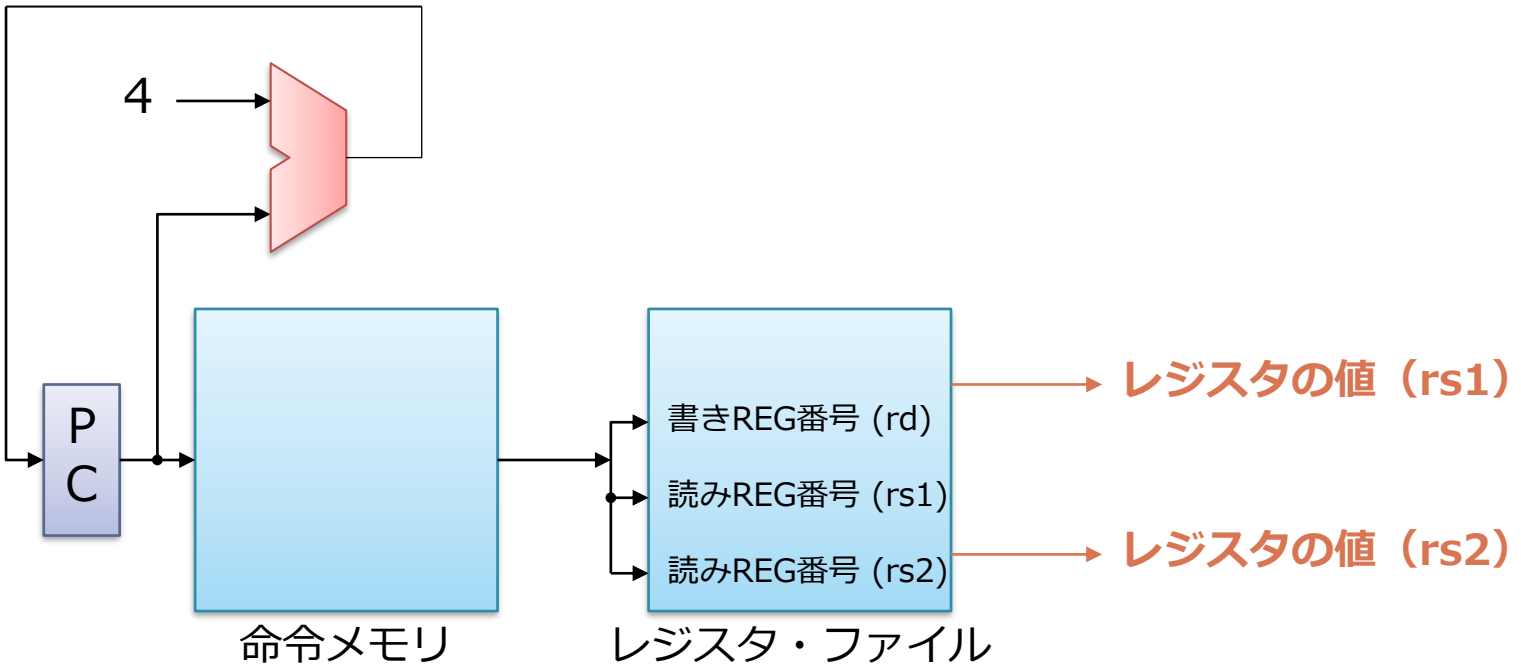


ADD : x[rd] ← x[rs1] + x[rs2]



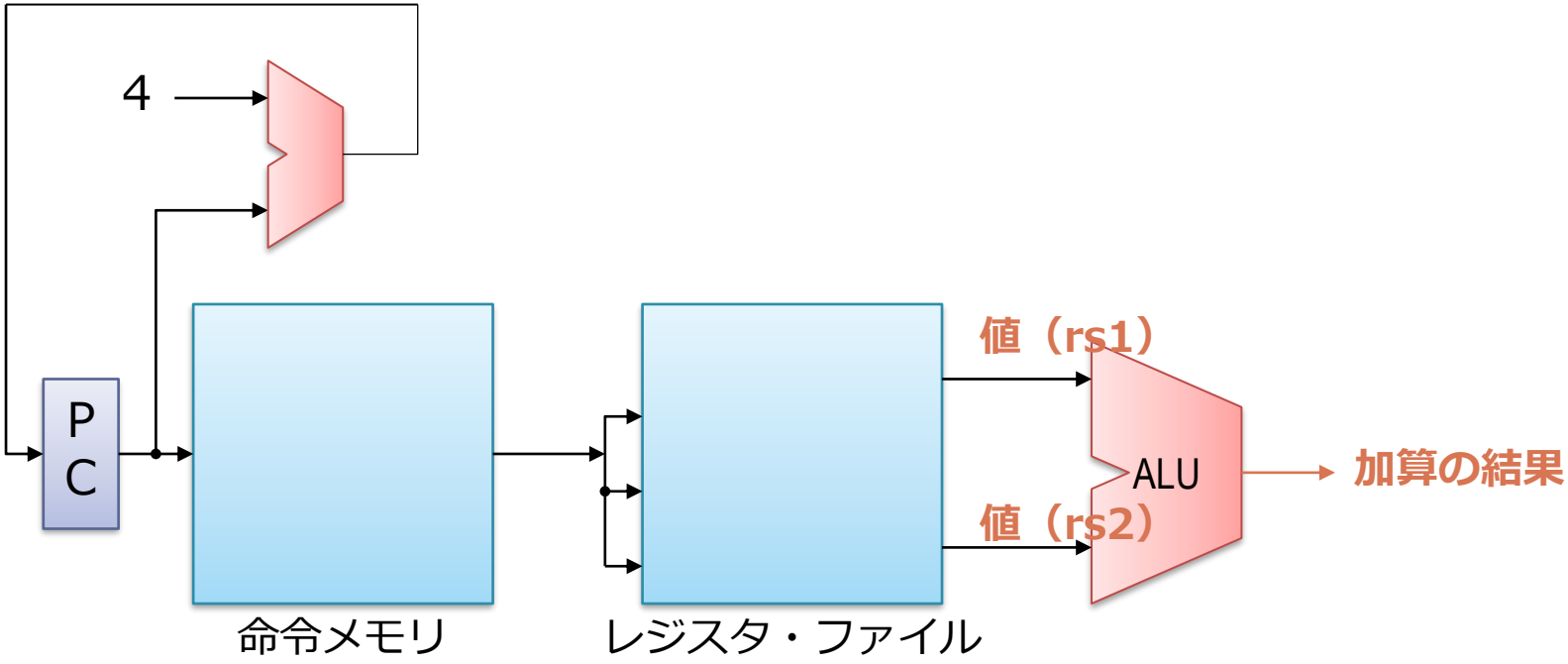
- 取り出した命令からレジスタ番号を表す部分のビットを取り出す
  - ソース（rs1, rs2）とディスティネーション（rd）

レジスタ読み出し



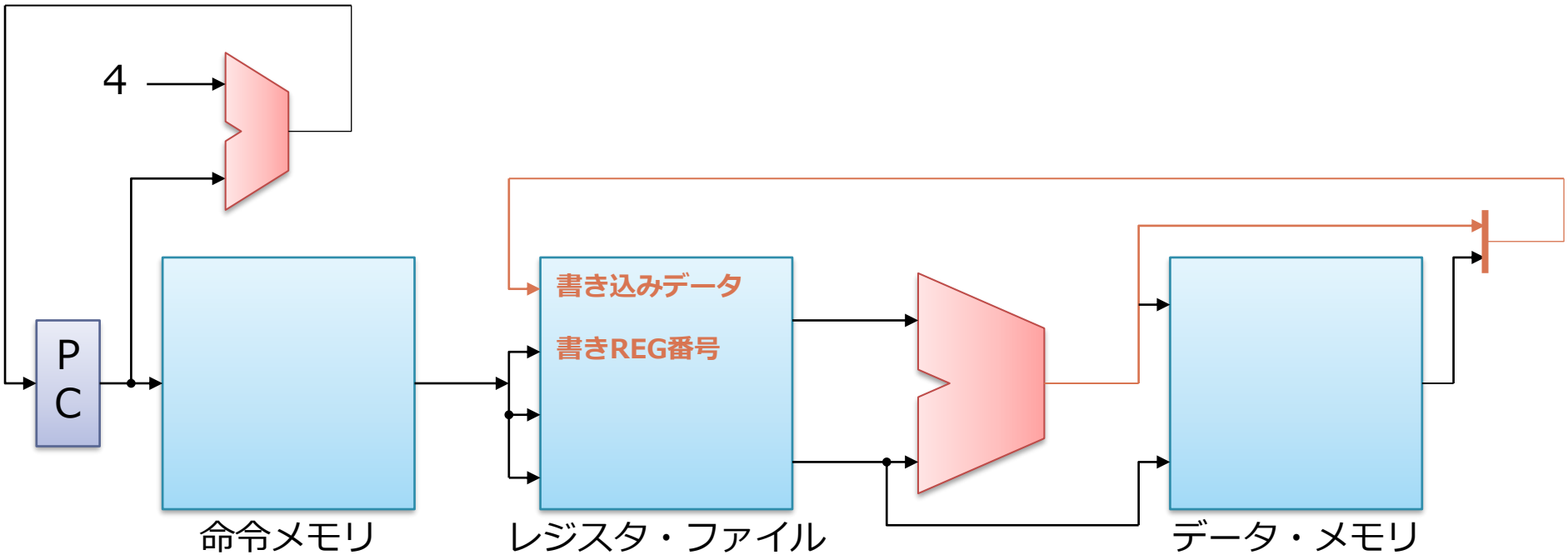
- デコードで得られたレジスタ番号を使って RF にアクセス
  - ソース・オペランドの値を読み出す

実行



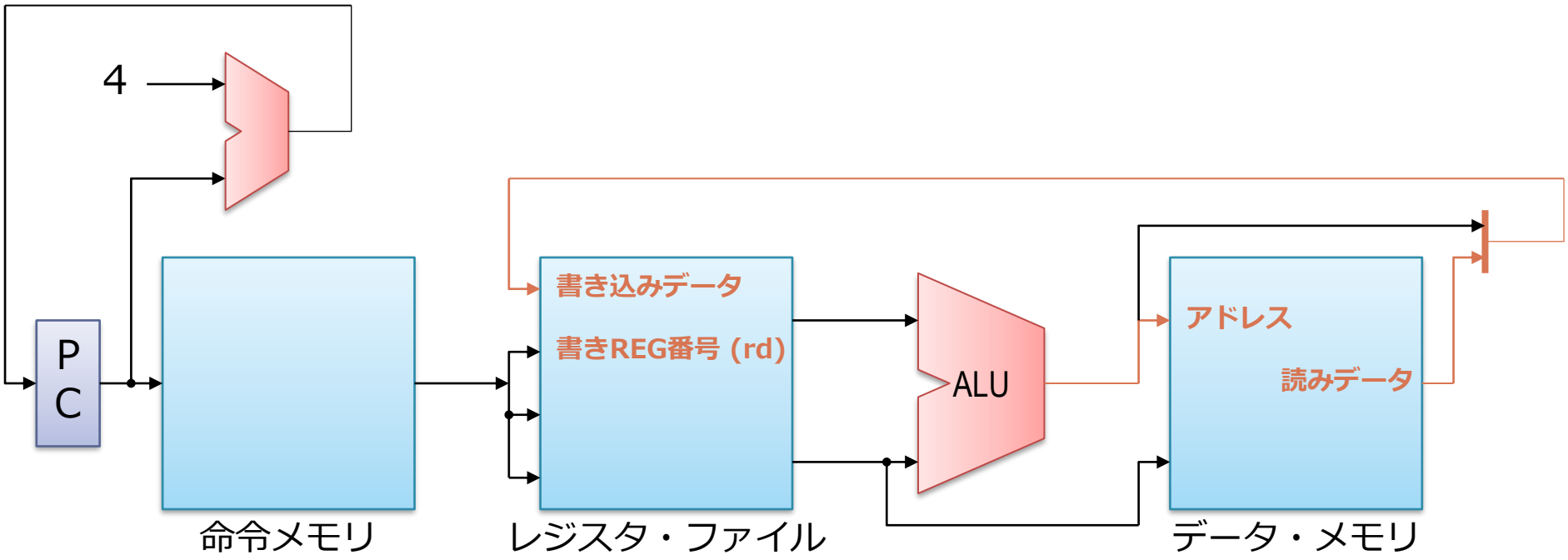
- RF から読みだした２つの値を加算

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

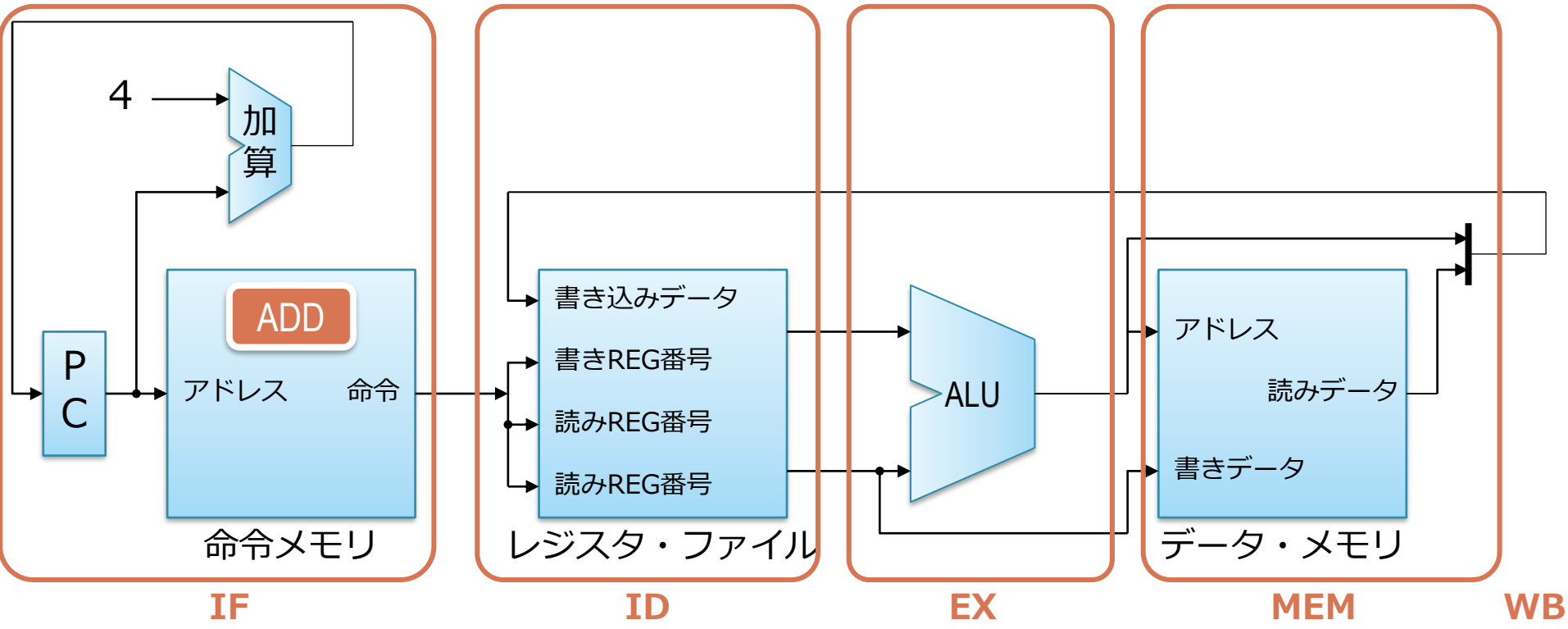


LW : x[rd] ← (x[rs1] + immediate)

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

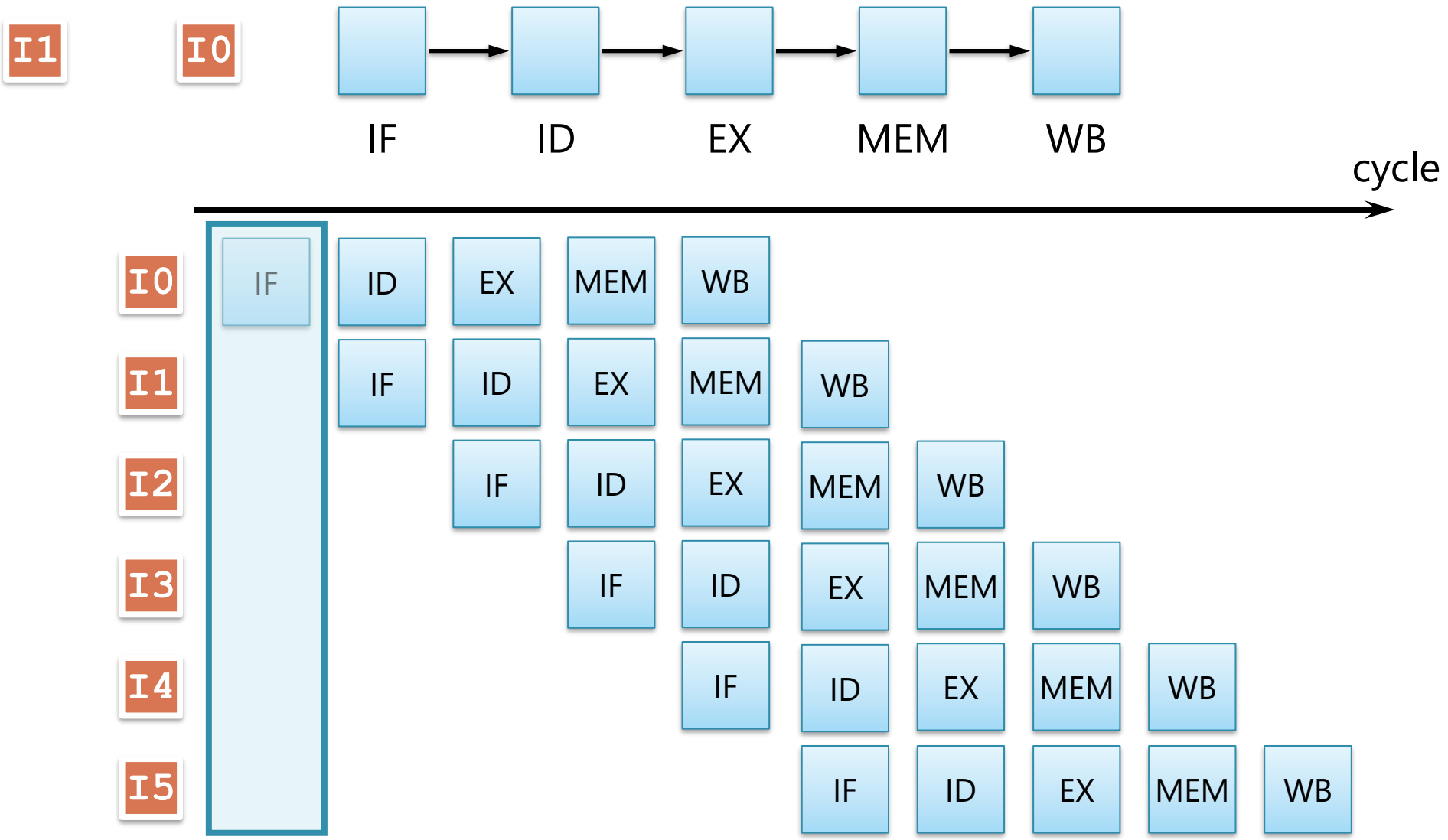
パイプライン化

- 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

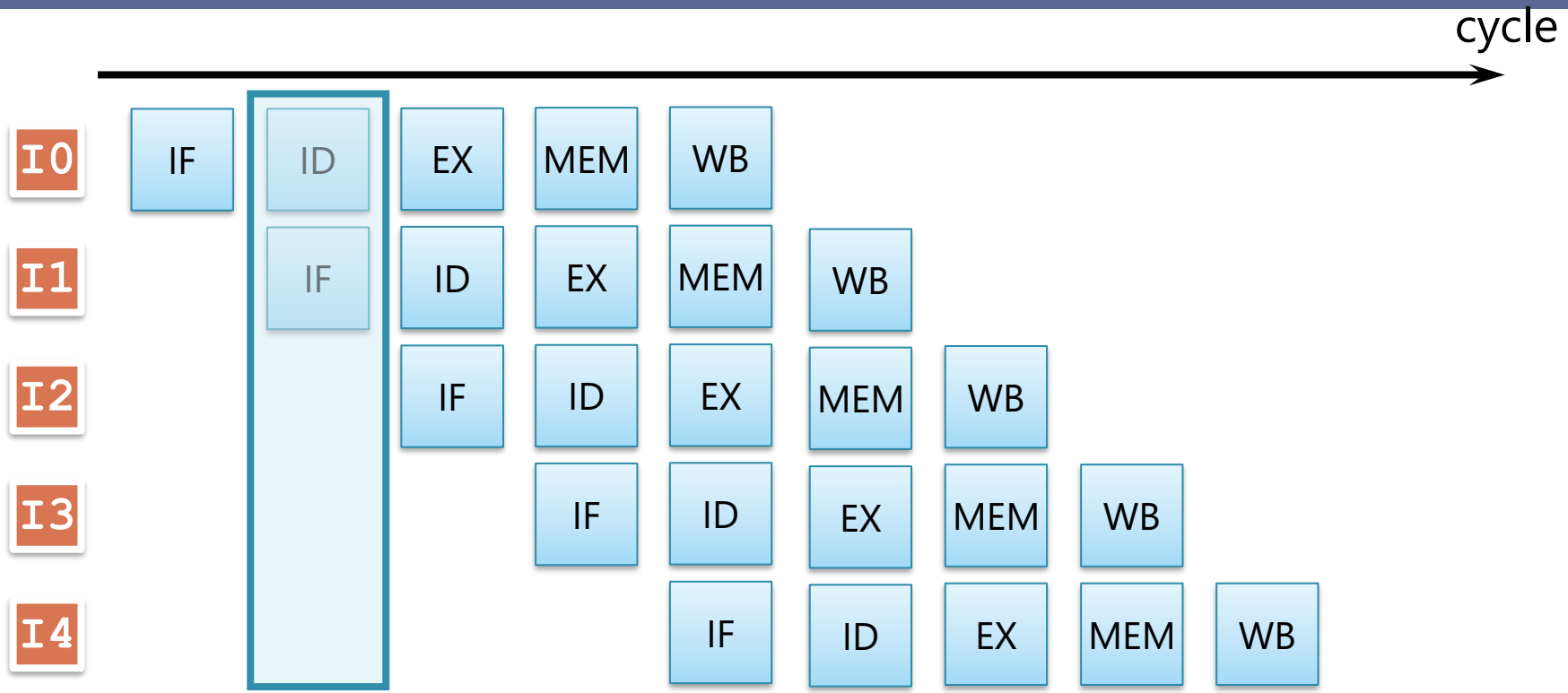
工場のラインを考える（再）

- 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

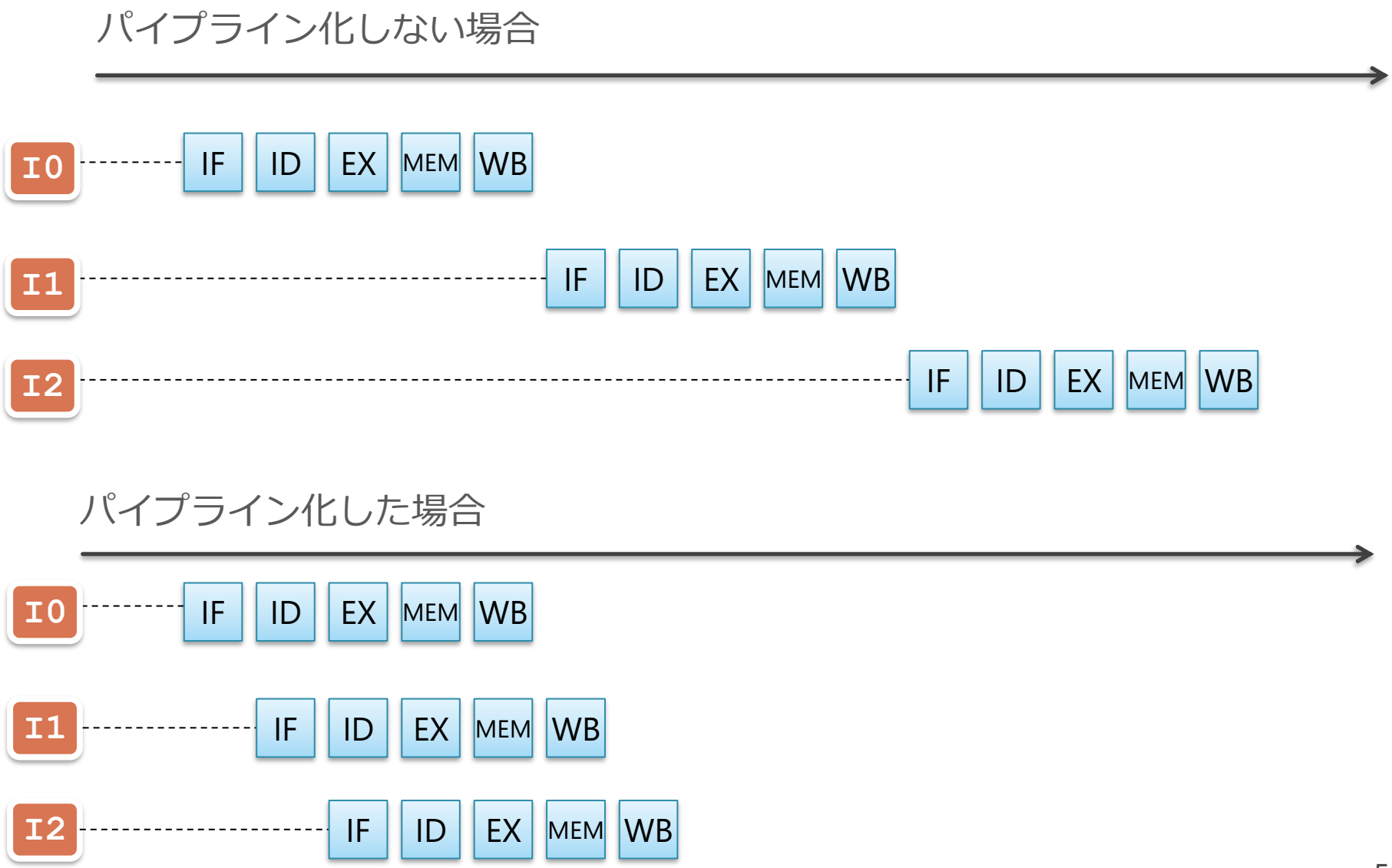


パイプライン・チャートの見方
ここから先で多用されるので重要

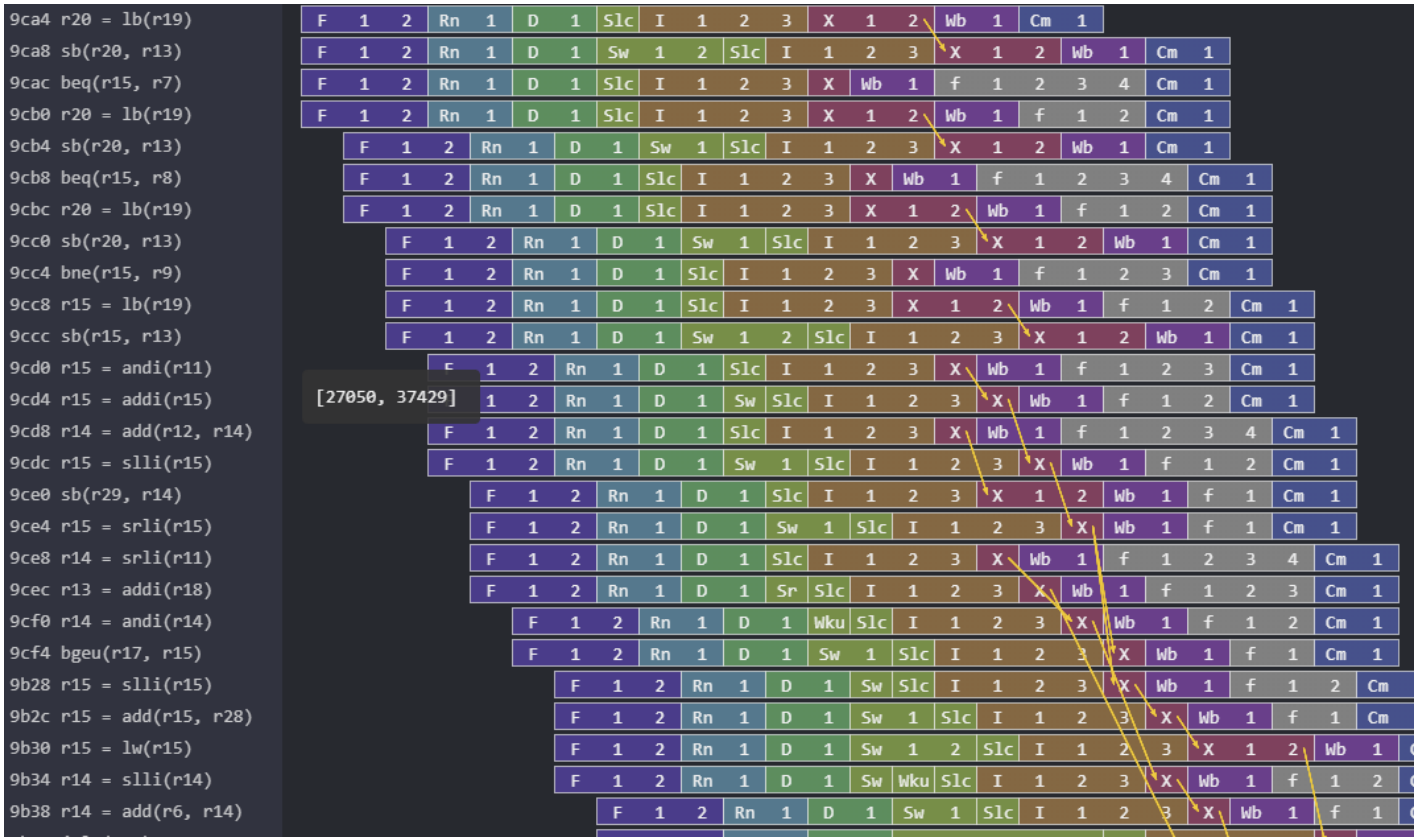


- 左から右にむかって時間は進む
- 上から下にむかって命令が実行順に置かれる
- 各ステージを表す四角は左側にある命令がその時そこにいることを示す
  - 上記では２サイクル目に，I0 が ID に，I1 が IF で処理されている

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

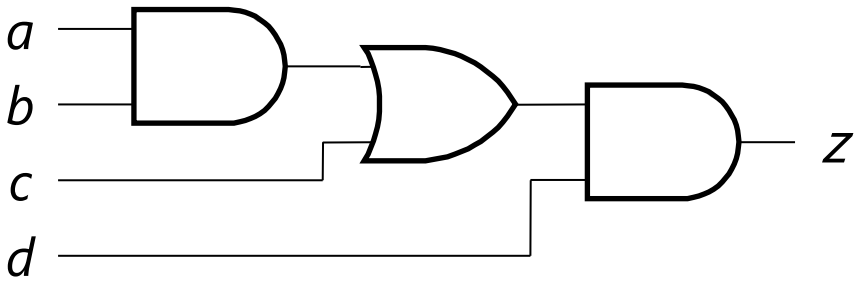


- 塩谷が開発している RISC-V CPU（RSD）の実行を可視化したもの
  - https://github.com/rsd-devel/rsd
  - out-of-order 実行をしているので，途中からプログラム順とは異なるタイミングで実行が進んでいる

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

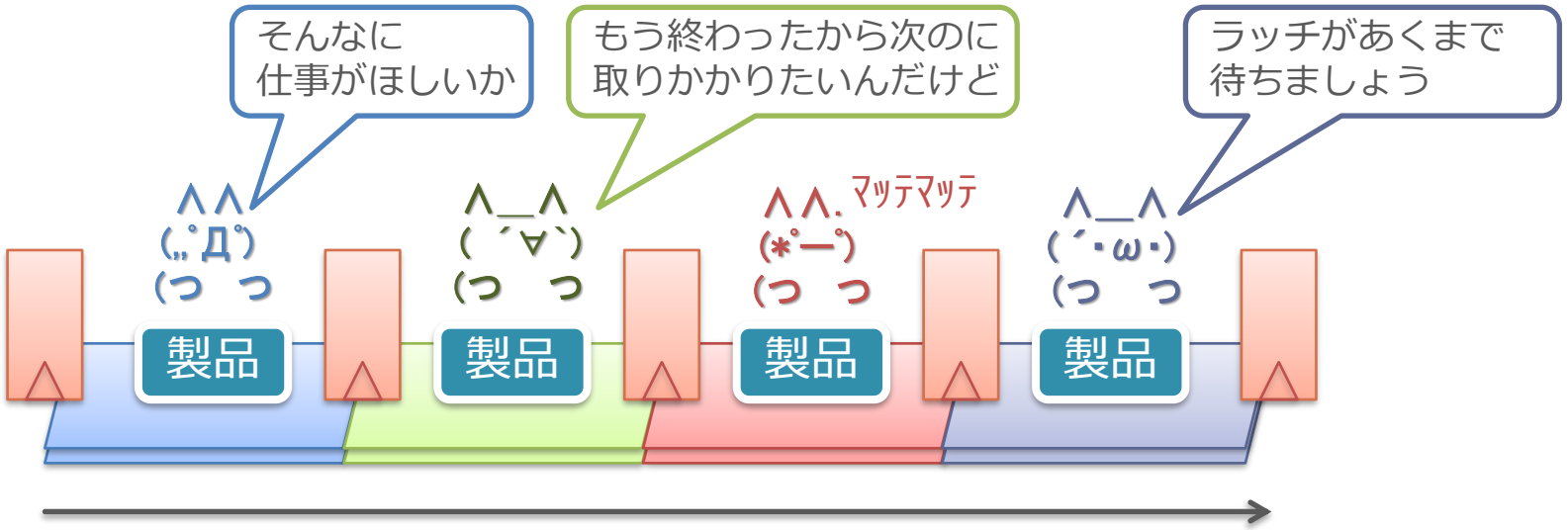
1. 各ステージを完全に同じ長さにすることは凄く難しい
   - 同じ長さ=同じ遅延=全く同じ段数の組み合わせ回路
2. 長いステージであっても信号は絶えず変化する可能性がある
   - 短いパスから順に出力に反映される
   - たとえば下の回路で *a,b,c,d* が全て変化したとすると，まず *d* の変化が *z* に反映し，次に *d* が･･･



パイプライン化（オーバーラップ）の実現方法

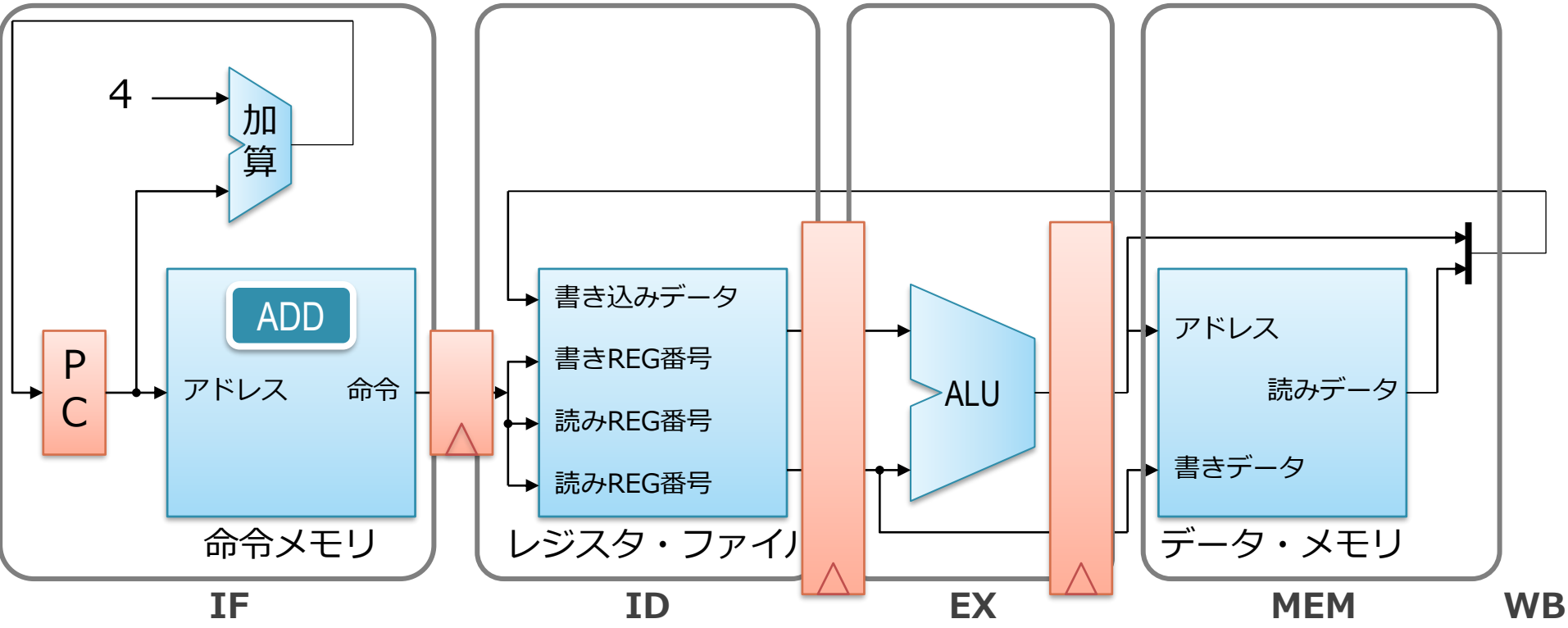
- シングルサイクル・プロセッサにフリップ・フロップをいれる
  - このためのフリップ・フロップをパイプライン・ラッチという
    - ラッチ（latch）は「ドアや門などの掛け金」の意味
  - パイプライン・ラッチで区切られた部分がステージとなる
- １サイクルの間はパイプライン・ラッチで信号がとまる
  - 複数ステージ間で信号が混じるのを防ぐ
  - 指定された時間までラッチでドアを開かなくするイメージ？

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



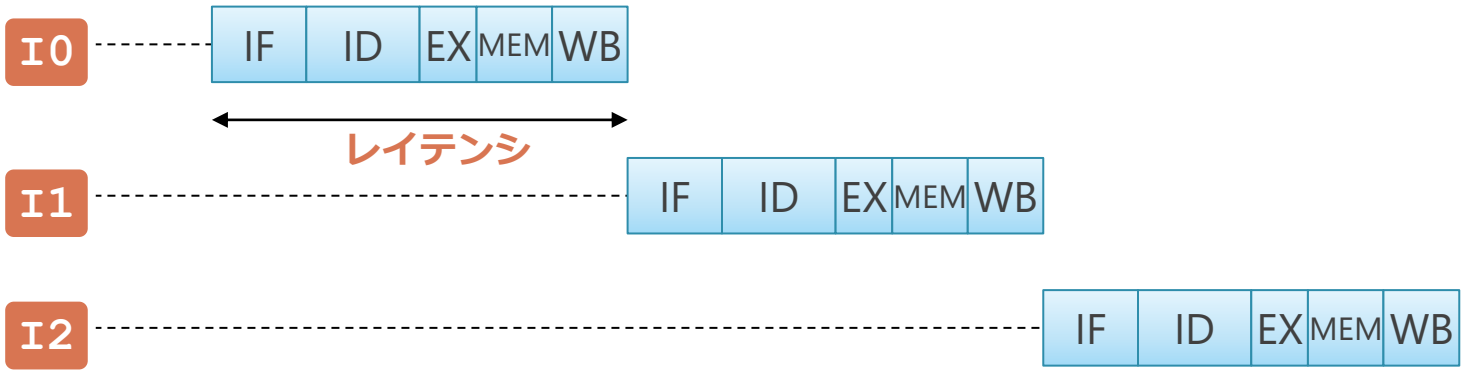
- 各ステージの間に，D-FF（オレンジの四角）をいれる
  - WB の書き込みについては，レジスタ・ファイル自体がクロックに同期して書き込みが行われるので D-FF は不要
- 各ステージの処理が早く終わっても，次のクロックまでは D-FF で信号の伝搬を止める

パイプライン化の効果

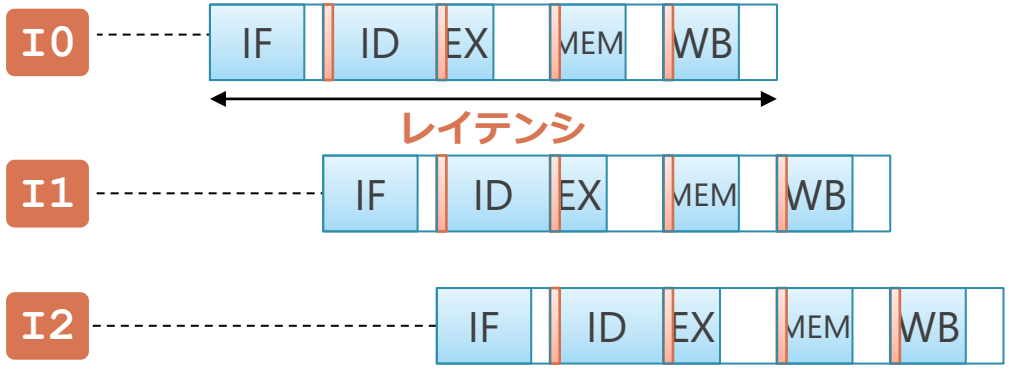
- レイテンシ (latency)：　　短くならない（か，やや延びる）
  - 一続きの処理が始まってから終わるまでにかかる時間
    - この場合，1命令の始まりから終わりまでの処理時間
  - 一番長い処理に合わせて動かすので伸びる
    - 原理的に短くならない
      （ステージ間にフリップフロップが入る分は絶対のびる）
- スループット (throughput)：　ステージ数倍だけ上がる
  - 単位時間当たりの処理量
  - この場合，単位時間あたりに実行される命令数

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

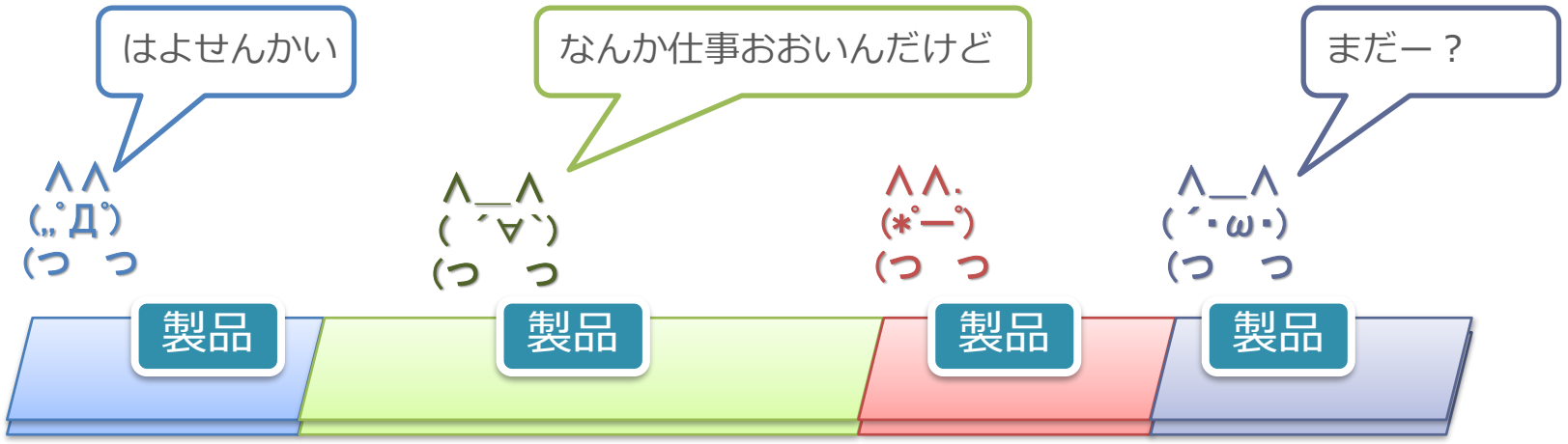


パイプライン化した場合



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

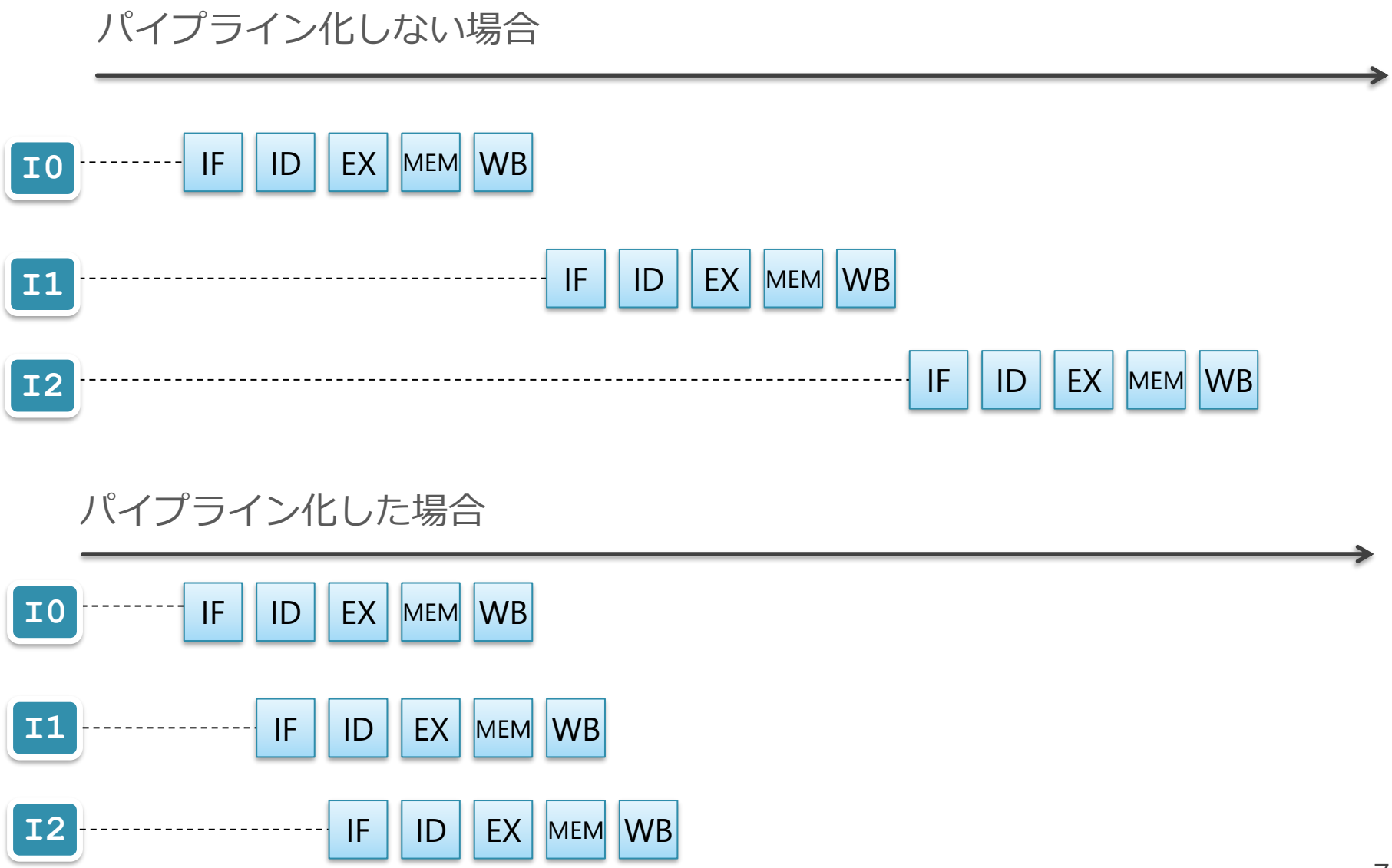
余談：非同期回路やウェーブ・パイプライン

- 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

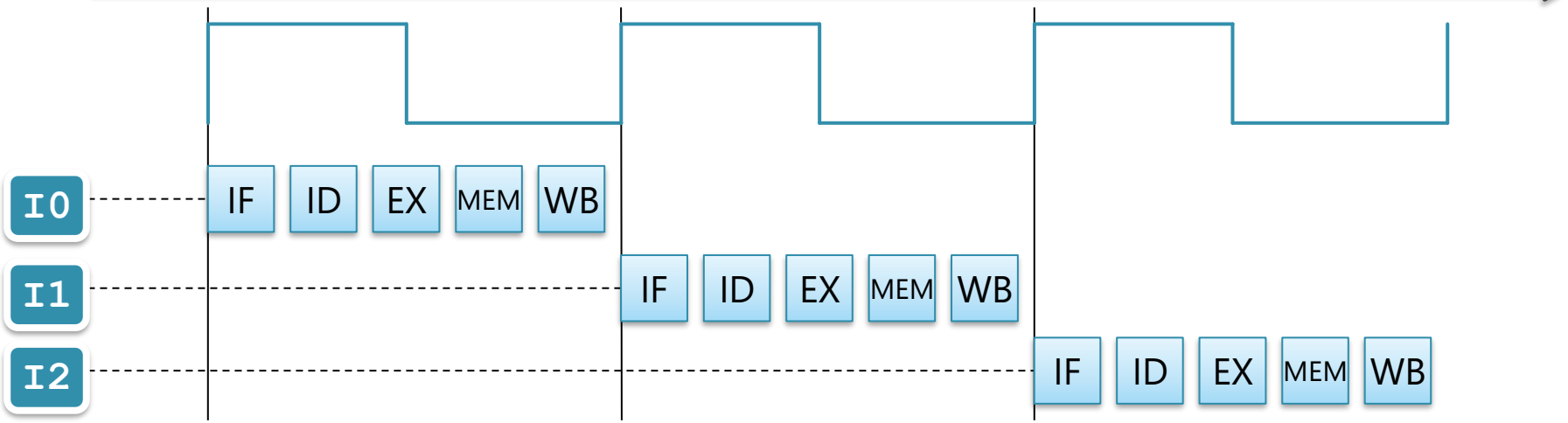


パイプライン化の意味

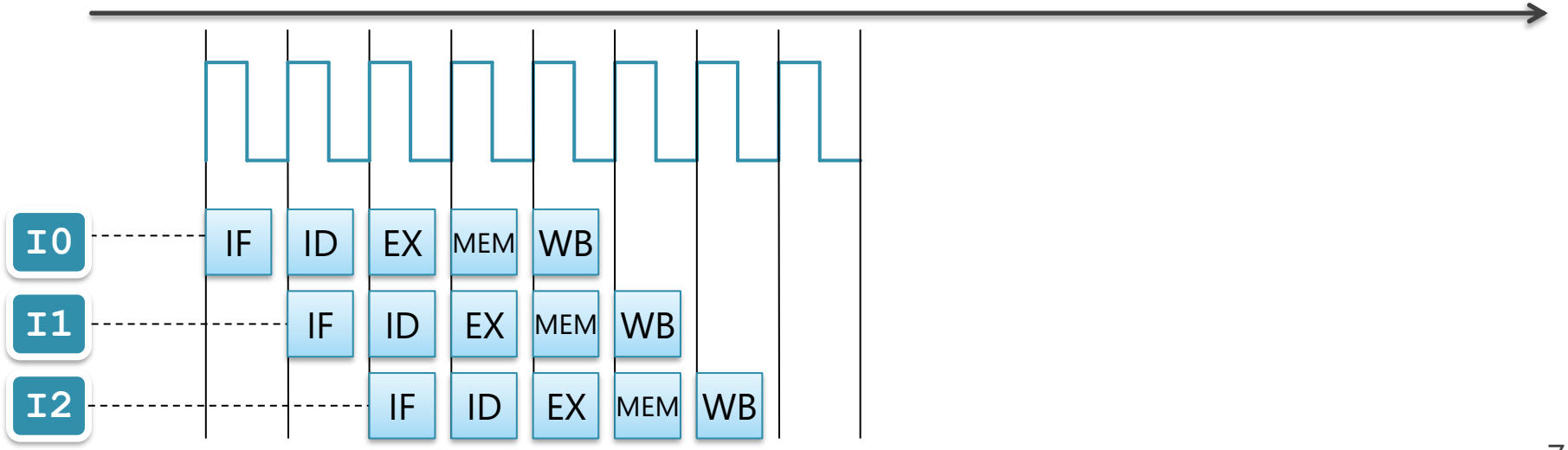
- パイプライン化の効果：
  - スループットの向上
  - = 単位時間あたりに処理できる命令の数の増加
  - = 動作クロック周波数の向上
- これらは同じ事を言い換えてるだけ

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

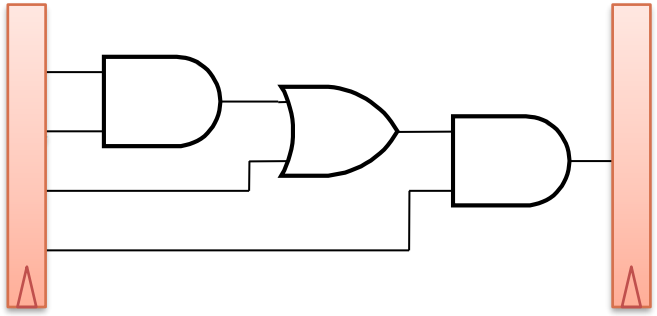
パイプライン化しない場合



パイプライン化した場合

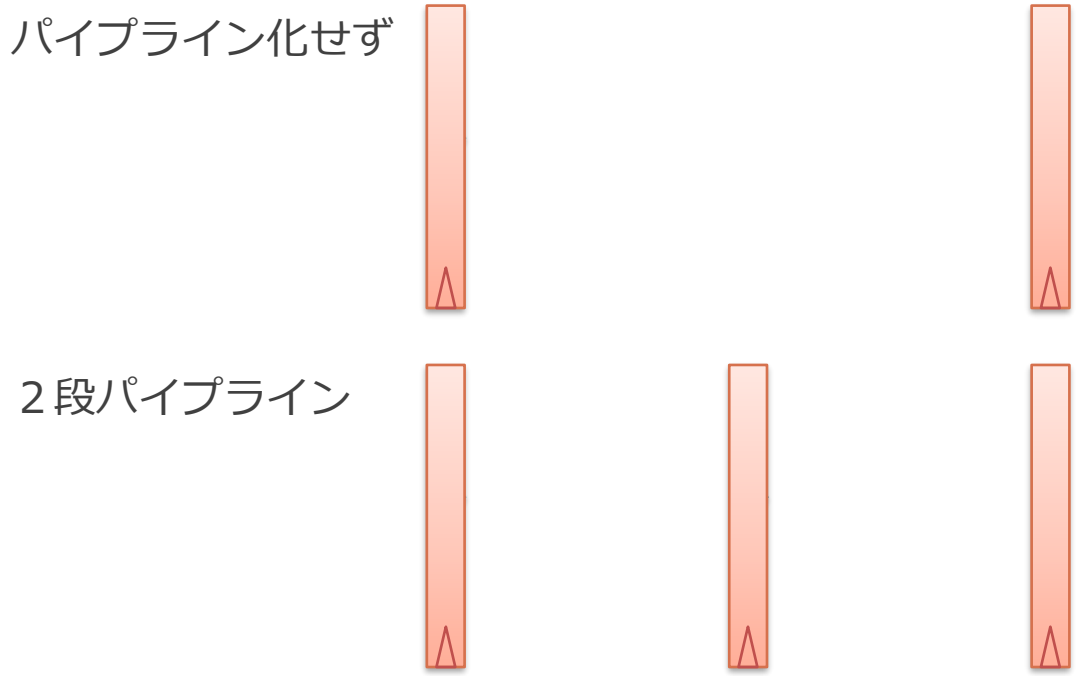


ステージ内の信号の伝播を考える



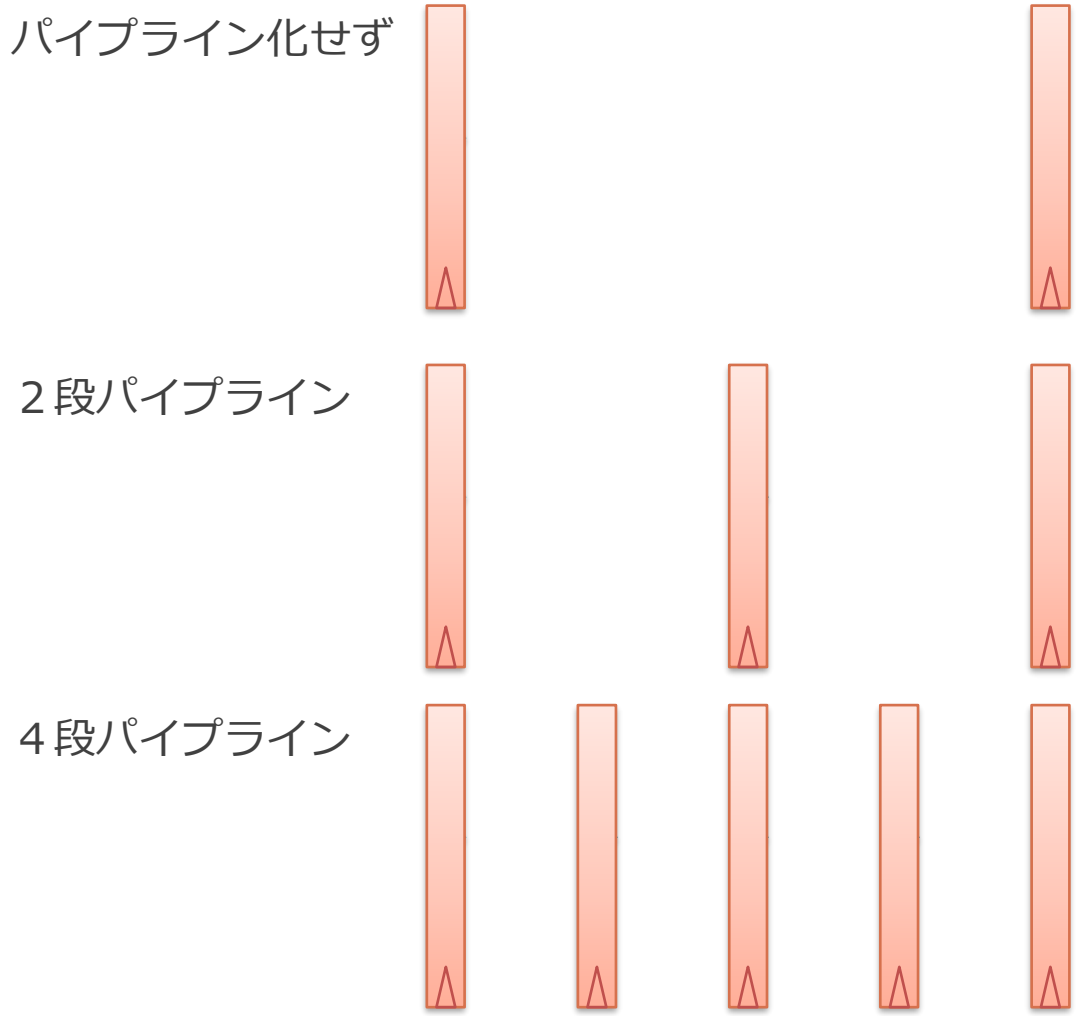
- パイプライン：ステージ：
  - 複数のパイプライン・ラッチで挟まれてた，
  - 組み合わせ回路（ゲート）
- 矢印の動き：
  - クロック開始時に，左のラッチからでた信号が
  - 組み合わせ回路を通って，伝播していく様子

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

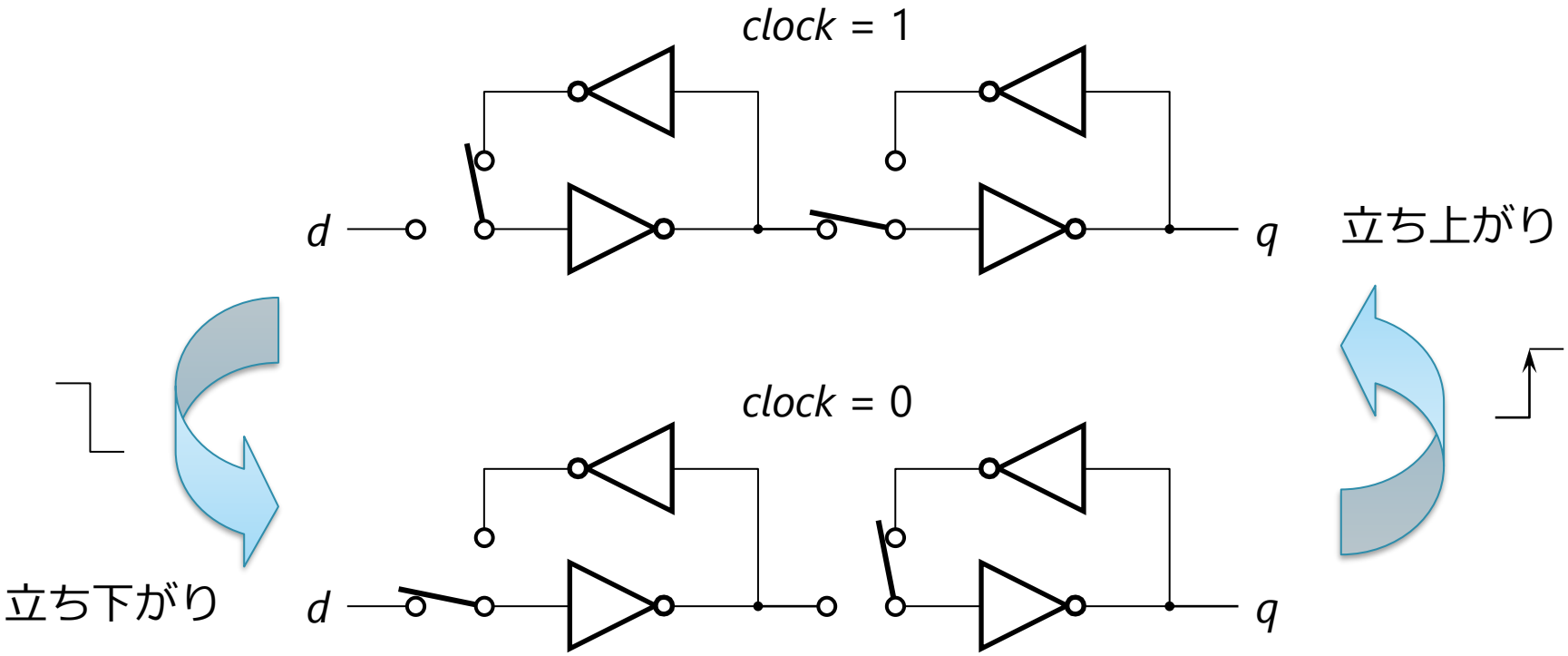
- パイプライン段数を増やしていけば，どこまでも速くなるのか？
  - ならない
- 理由：
  1. 回路的な理由による周波数向上の限界
  2. アーキテクチャ的な理由による実効性能の限界

回路的な理由

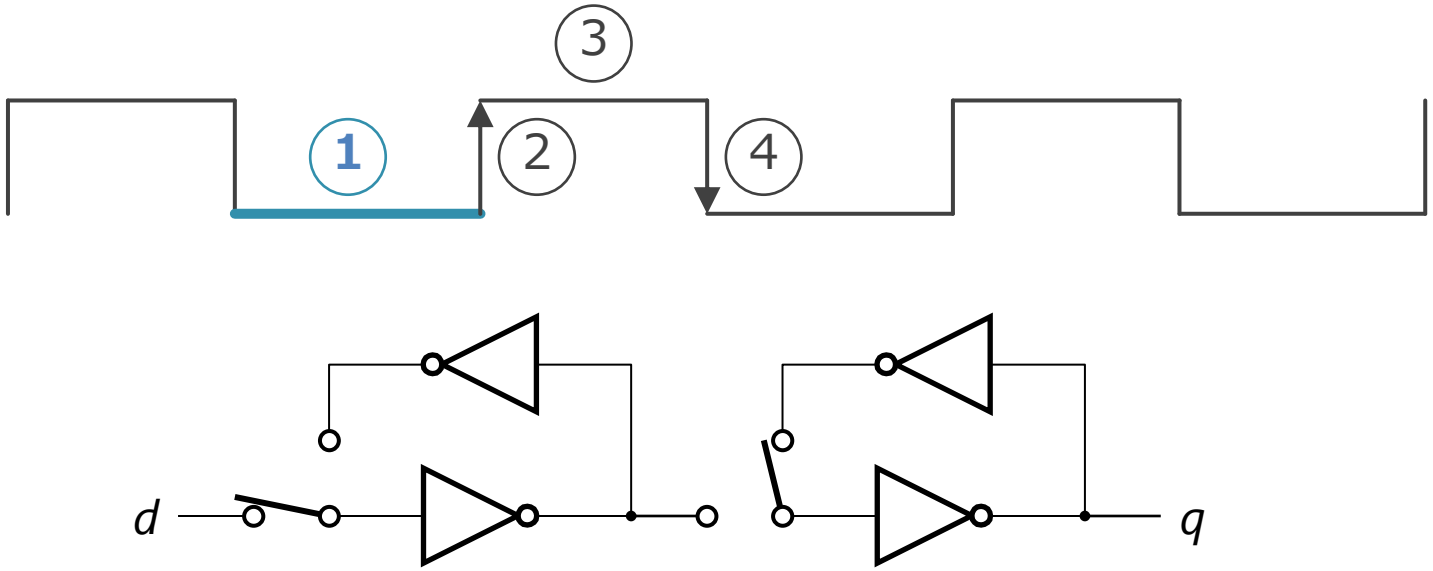
- 理由：
  1. D-FF 自体の遅延のため
  2. 消費電力と熱のため

D-FF の回路

- 構造：マルチプレクサが入った２つのインバータのループ
  - 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，*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

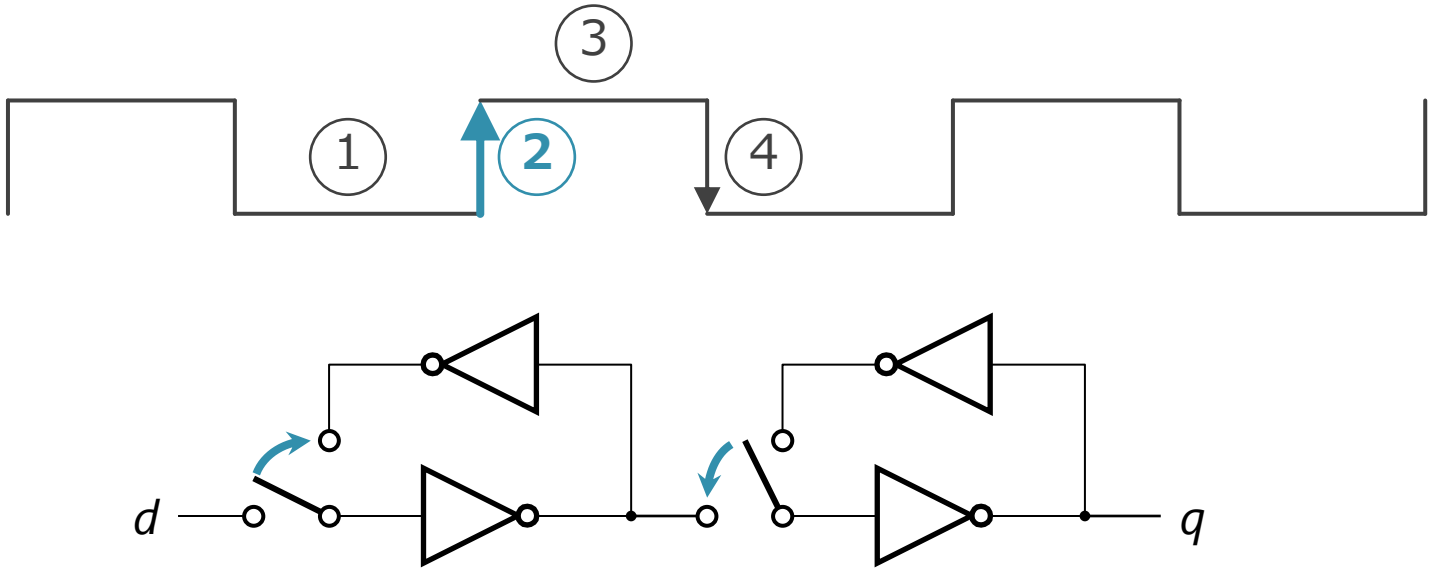


D-FF の動作 ① クロック信号が Low にあるとき



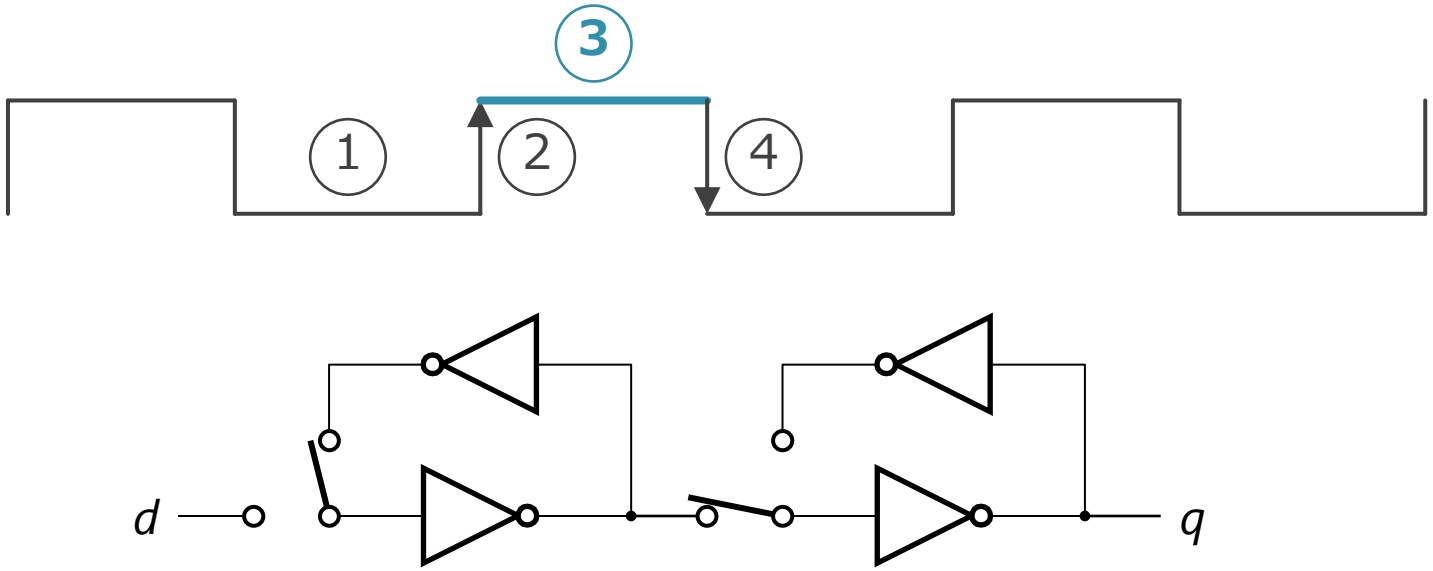
- 左側のループ：
  - ◇ d の入力の変化に応じて，インバータの状態が随時切り替わる
  - ◇ 右側のループとは遮断されている
- 右側のループ：
  - ◇ ループのインバータの状態（=記憶）が q に出力され続ける

D-FF の動作 ② クロック信号の立ち上がり



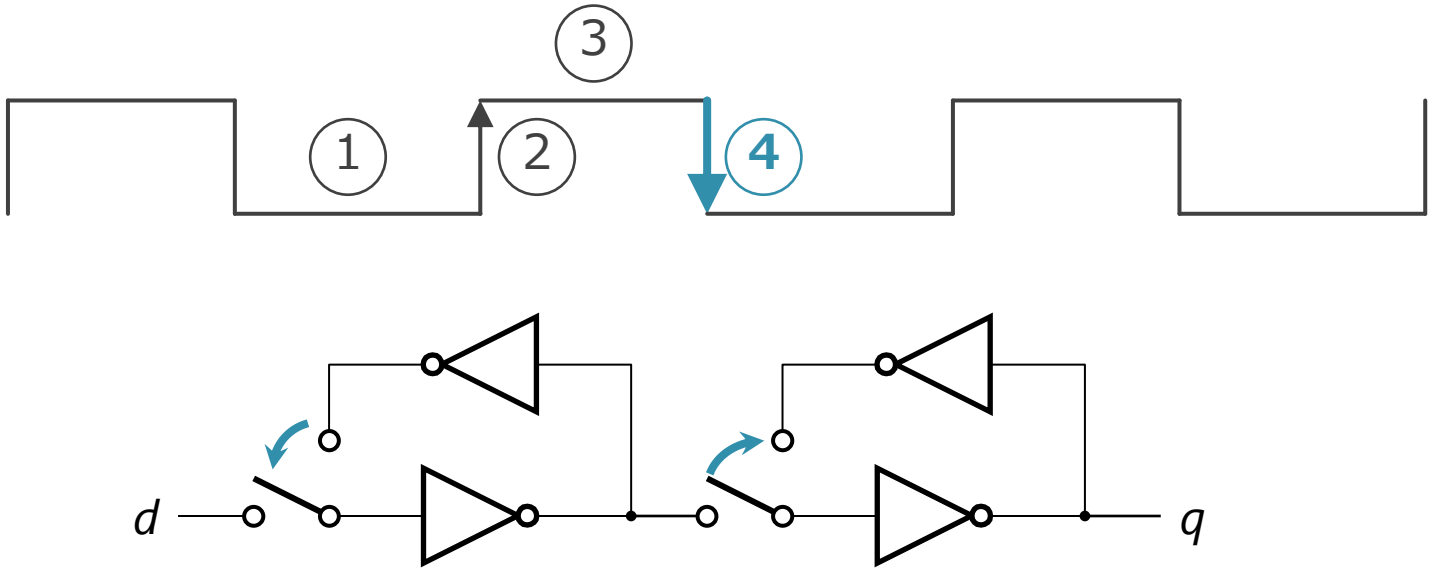
- 左側のループ：
  - ◇ *d* と遮断され，ループが形成される
  - ◇ 直前まで *d* に入力されていた信号が記憶される
- 右側のループ：
  - ◇ 左側のループと繋がり，ループが解除される

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 5.1

- 第 1 回の講義資料を参考に，以下の if 文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
  - 変数 i はメモリのアドレス 0x100 に割り当てられているものとせよ
  - 変数 i の初期値は任意（「...」）とせよ
  - 変数 j は任意のレジスタに割り当てて良い

```
1: i = ...;
2: j = 2;
3: if (i > j) {
4:     i = i + 1;
5: }
5: else {
6:     i = i - 1;
7: }
```

課題 5.2

- 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 NOR ゲート
  - 3入力 NAND ゲート

2入力 NOR の真理値表

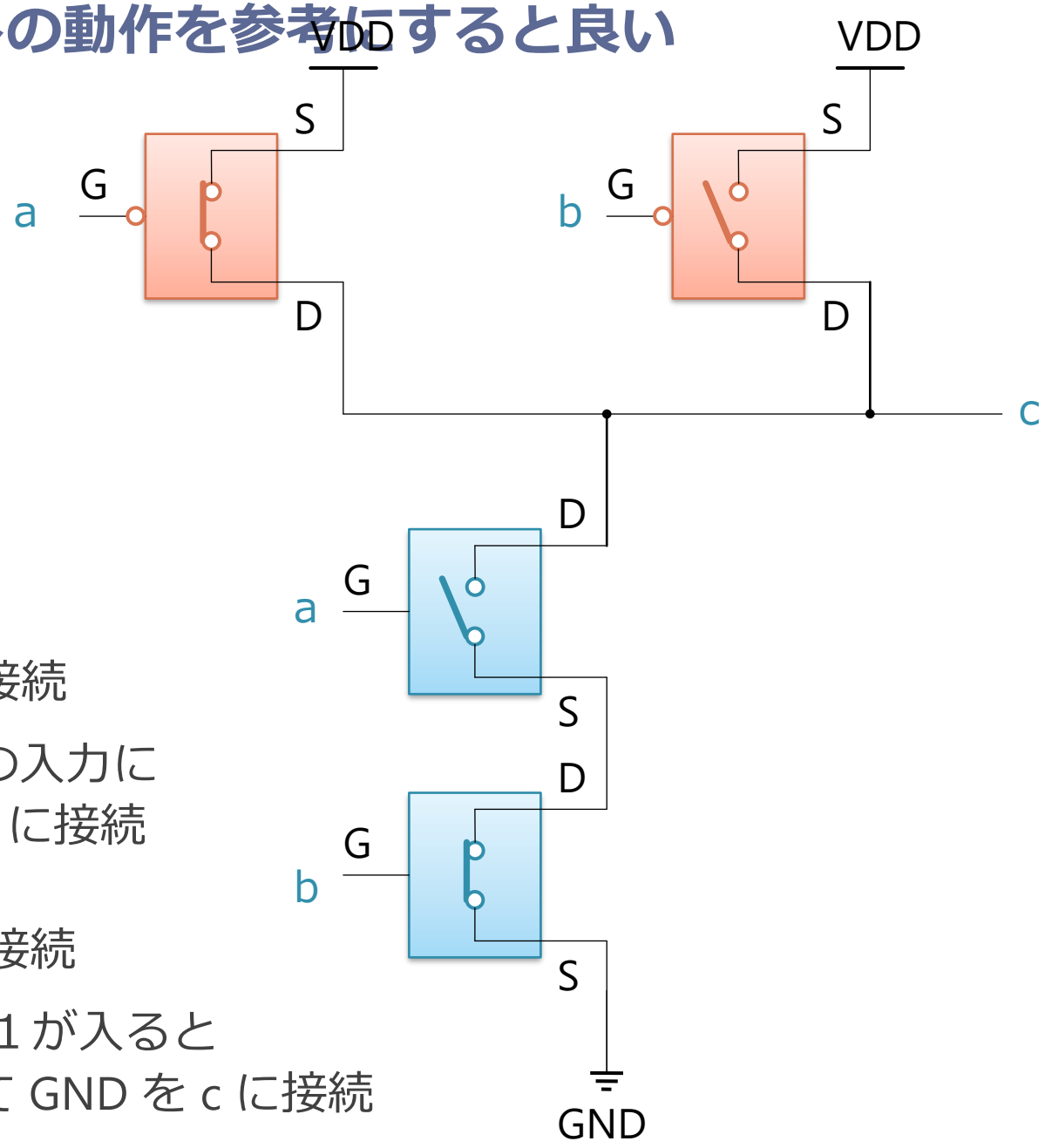
a	b	o
0	0	1
0	1	0
1	0	0
1	1	0

3入力 NAND の真理値表

a	b	c	o
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
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 5.1 と 5.2：
     - 提出は Moodle の「課題 5 」のところからお願いします
     - 課題 5.2 は画像（紙に書いてスマホで写真でもよし）
  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
  - Moodle に設定した締め切りまで（6/4 日曜日の 23:59 頃，要確認）

- 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