コンピュータ アーキテクチャ Ⅰ　第６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

- 気が早いかもしれませんが、期末テストの日程は7/31(月)ですか？

質問や感想など

- iPadやスマホで資料を開いた時、スライドが見切れないようにする方法はありますか？
  - Adobe Acrobat や 365 のビューアを使えば問題無いように思う

質問や感想など

- 論理回路が苦手です…。途中から何をやっているか分からなくなってきます
  - どこから分からなかったか，書いてもらえると

質問や感想など

- 今回の授業ではD-FFの動作のあたりが難しいように感じました。
- D-FFの説明に出てくるマルチプレクサとインバータがよく分からなくなってしまったのですが、NOTゲートがインバータで、マルチプレクサの選択によりNOTゲートが実現されているということでしょうか。

質問や感想など

- リレーは前後の電位によってONかOFFか変わるので描く時はスイッチの開いている閉じているは適当という認識でよいのでしょうか。何かオーソドックスな表記の仕方があったりしますか。
  - わかれば，別に適当で良いです

質問や感想など

- ゲートを構成する際に直列回路を並列回路のどこに接続するかで悩んだのですが、接続する位置によって実際の処理などに違いが出ることはあるのでしょうか。

質問や感想など

- マリオメーカーの例で、ANDゲートが簡単に作れることが理解できました。（マリオメーカーのANDゲートはそんなに実用的だはないですね、、、）

質問や感想など

- ステージの長さが一部分だけ長くなってしまった場合はどのように改善されるのでしょうか？

質問や感想など

- 命令パイプラインを実際に使って作業時間を短縮している具体例などはありますか？

質問や感想など

- for文のアセンブリ言語を活かして書こうと思いましたが、場合分けがLABEL1.2でいいのか分かりません。あと、メモリの振り分けもよく分かりません。

質問や感想など

- （パイプライン化について）直感では一続きの処理が始まってから終わるまでにかかる時間が短くなりそうだと感じていたため、短くならずむしろ長くなるかもしれないということを知り、驚きました。説明で理解できたものの、直感とは異なるなと思いました。

質問や感想など

- パイプライン化に限界があるとのことでしたが、実際何段くらいパイプライン化できるのでしょうか。
- パイプラインの段数は、無限にはできないという話がありましたが、実際の場面ではどのように決められているのか気になりました。
- パイプライン化の限界があることが分かりましたが、具体的に何段階くらいが限界なんですか？

質問や感想など

- ステージ間のDFF回路同士はどのように繋がっていないといけないのでしょうか、誤差は大きくならないのでしょうか。

質問や感想など

- ファストフードの店も、商品を作るときにパイプライン全体の一番遅い人に合わせて動きます。

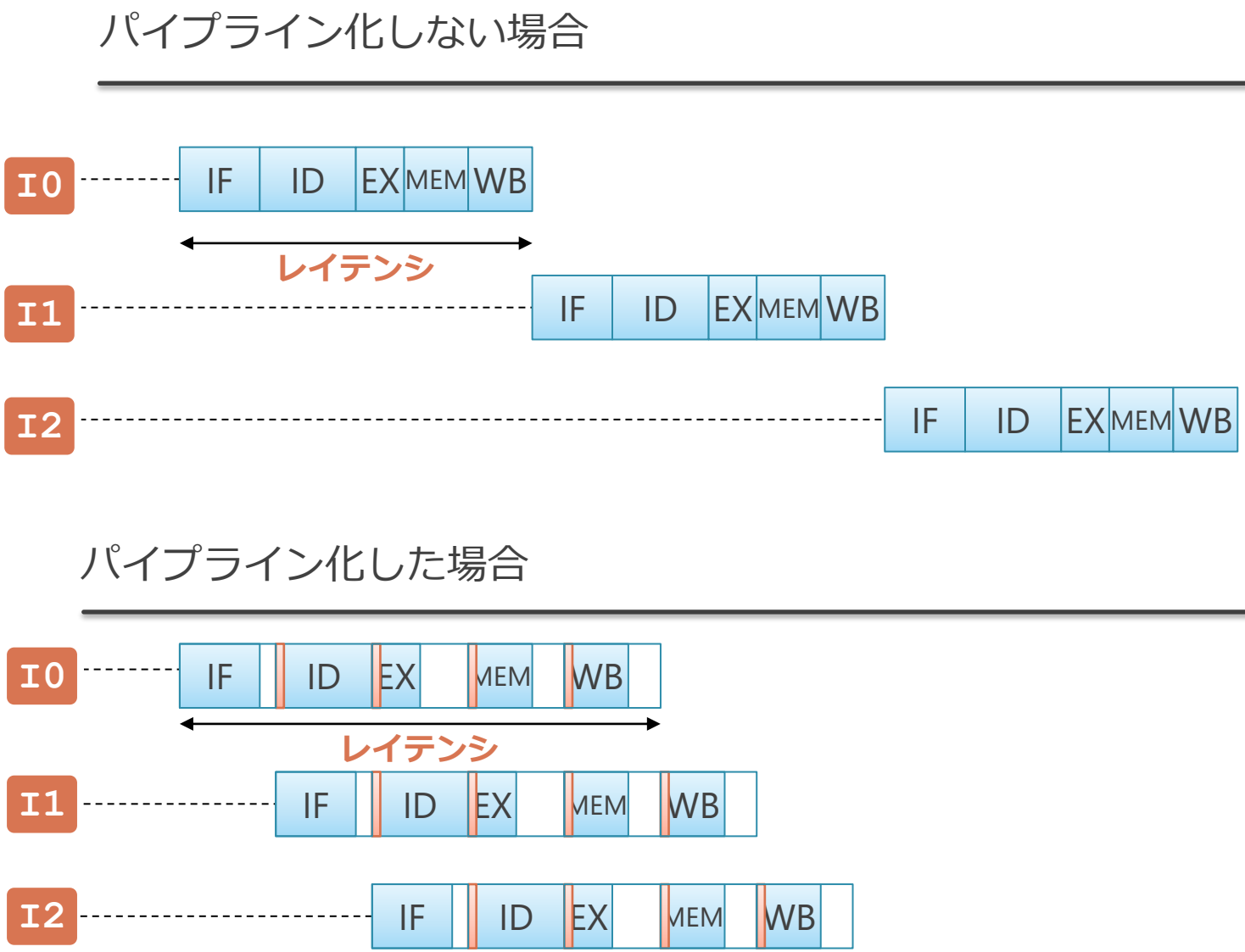
質問や感想など

- 課題5.1について、分岐命令bを使って飛んだところの1行だけ命令を実行してくれるのか、それともC言語の繰り返しのようにそれ以下の命令を全て実行するのかどちらですか。
  - 分岐命令とは「PC を書き換える命令」

質問や感想など

- パイプライン化の効果のレイテンシとして、原理的には短くならない理由がわかりませんでした。フリップフロップが入るとしても、全工程を同時に行なっているから、少しは短くなると思いました。（スライドの66では短くなっているように見えました。）

「レイテンシ」と書いている部分 = １つの命令の処理開始から終わりまで は，パイプライン化により伸びている



質問や感想など

- CPUはキーボード操作など割り込み処理していますが、その処理のためにクロック周波数の異なるD-ff回路を複数搭載しているのですか？

質問や感想など

- ラッチが開くまでの時間が、長くかかっているステージに合わせられるということは、極端な話、レイテンシがどんなに短くてもどこかのステージに極度に時間のかかる命令だとパイプライン化する前よりも時間がかかってしまうということでしょうか。

質問や感想など

- 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とありましたが、具体的に何をしているのかは我々にはわからないのでしょうか

質問や感想など

- 今ちょうどデスクトップPCを買おうと思っているのでCPUの話やクロックの話は選ぶ参考になります。
- 回路の話とプログラムの話はそれぞれなんとなくわかるのですがなかなか回路をプログラミングするというイメージがつきません

質問や感想など

- 一つの命令のみが極端に時間がかかる場合、その長さ分でラッチで区切ると１サイクルがすごく長くなってしまうと思うのですが、そういう場合はどのようにすれば良いのでしょうか？

  - その命令が出現するのが稀な場合は，その命令が出てくる時だけパイプライン全体を止めてその命令の結果が出るのを待ちます
  - 除算とかはそうなっていることが多い

質問や感想など

- 質問：ステージをどこで切るかによって効率化が図られているということでしたが、その切り方は事前に決められているのでしょうか？それとも実際の作業も進み方に合わせて逐次調整されているのでしょうか？

質問や感想など

- D-FFの動作について、ループが形成されると値を記憶することができるとおっしゃっていたと思うのですが、どのように値を記憶しているのかわからなかったの教えていただきたいです。

課題の解説

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 5.1

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

```
    li 0x100→A    // i のアドレス 0x100
    ld (A)→B      // i の値をメモリから読む
    li 2→C        // j の値を即値で設定
    b B>C LABEL1  // B>C なら LABEL1 に飛ぶ
    sub B-1→B     // else 節の i = i - 1
    j LABEL2      // LABEL2 に飛ぶ
LABEL1:           //
    add B+1→B     // then 節の i = i + 1
LABEL2:
    st B→(A)      // i をメモリに書き戻す
```

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

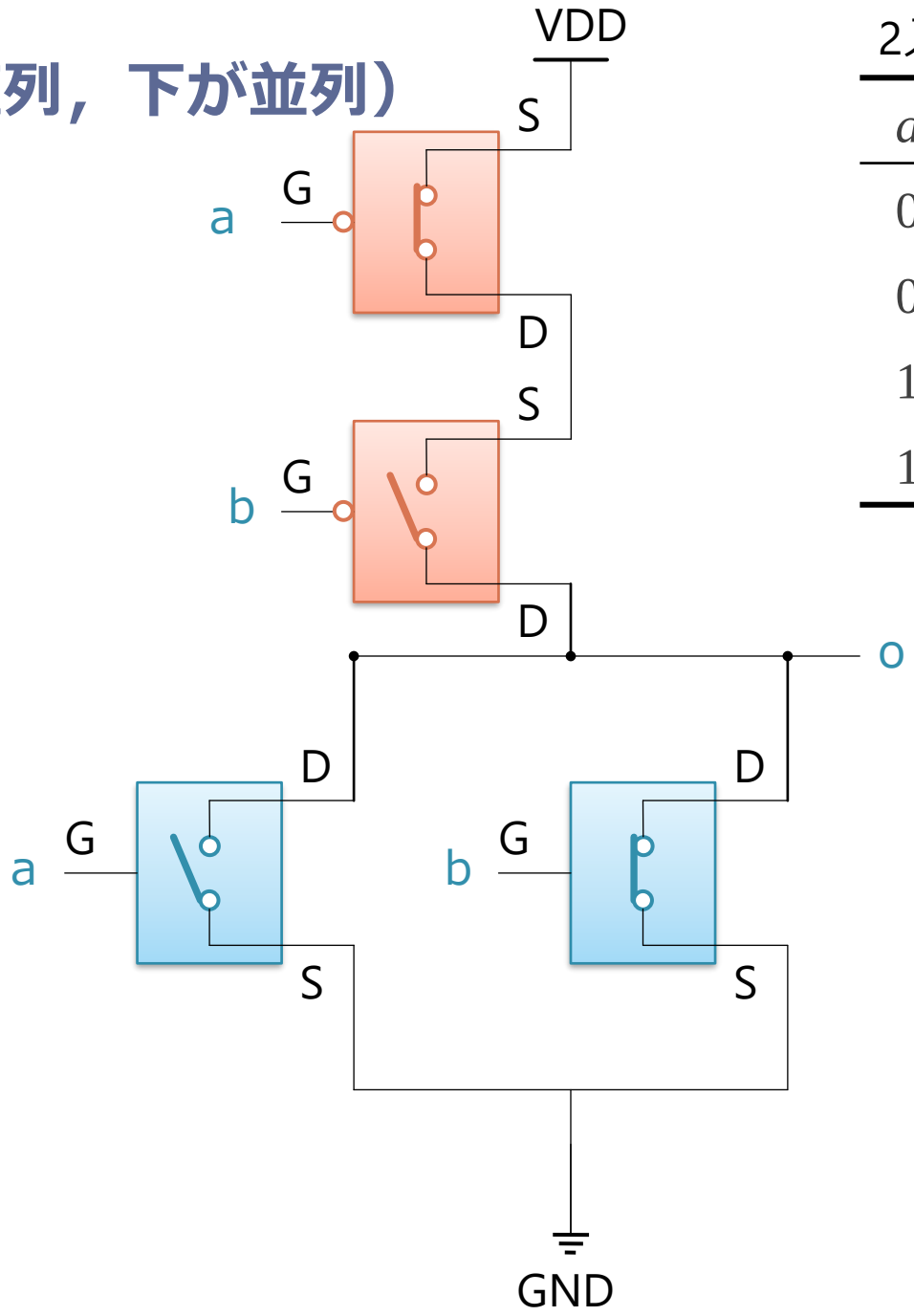
2入力 NOR の真理値表

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

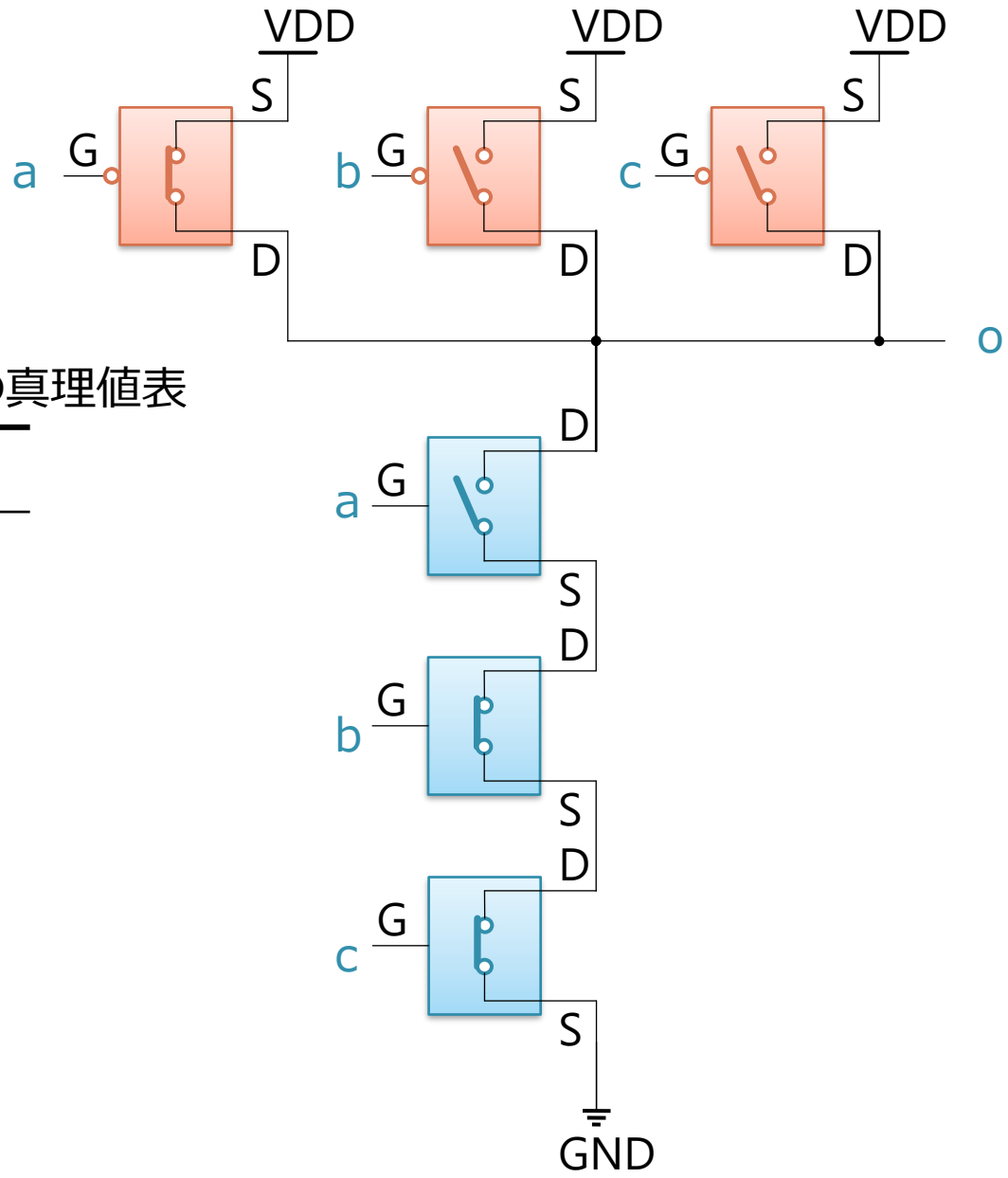
NOR
（上が直列，下が並列）



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



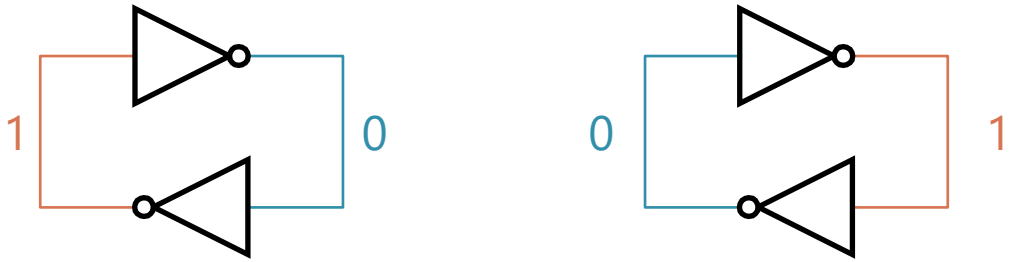
3入力 NAND の真理値表

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

前回の振り返り

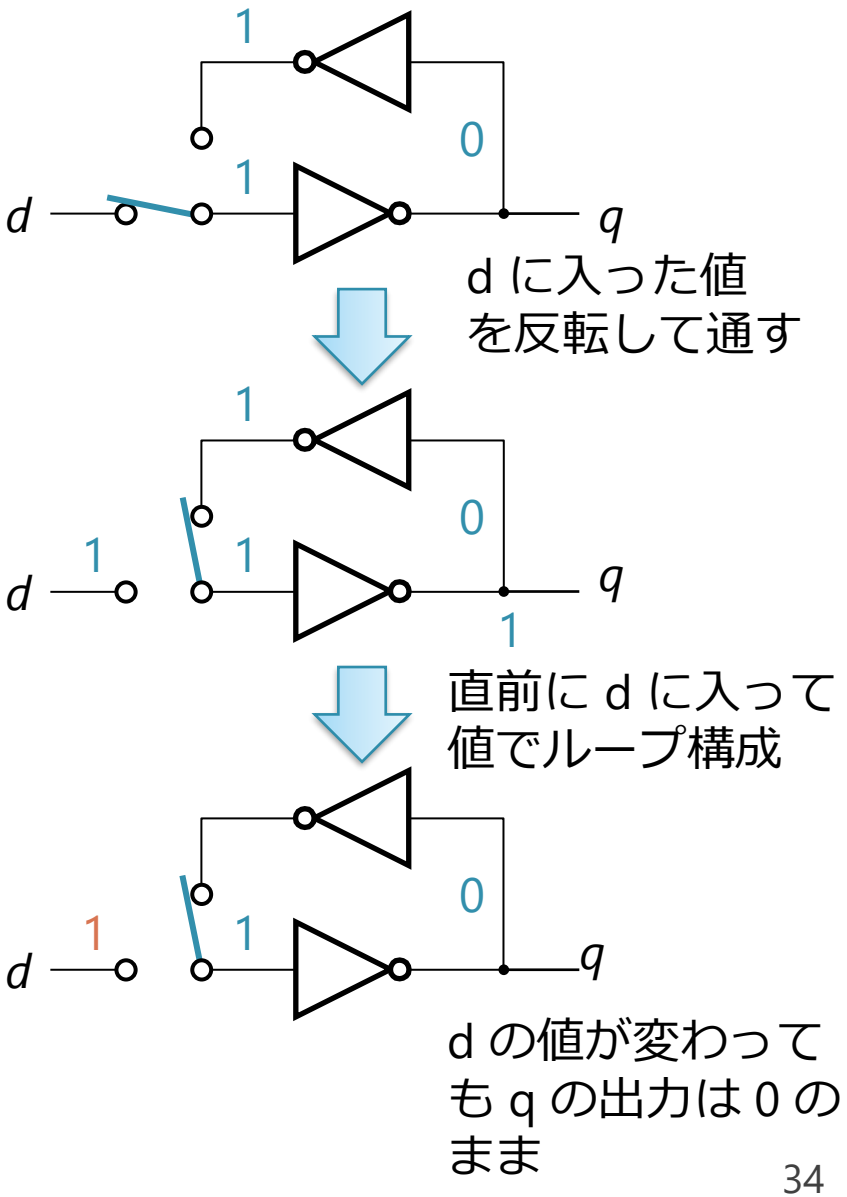
記憶素子の原理

- 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



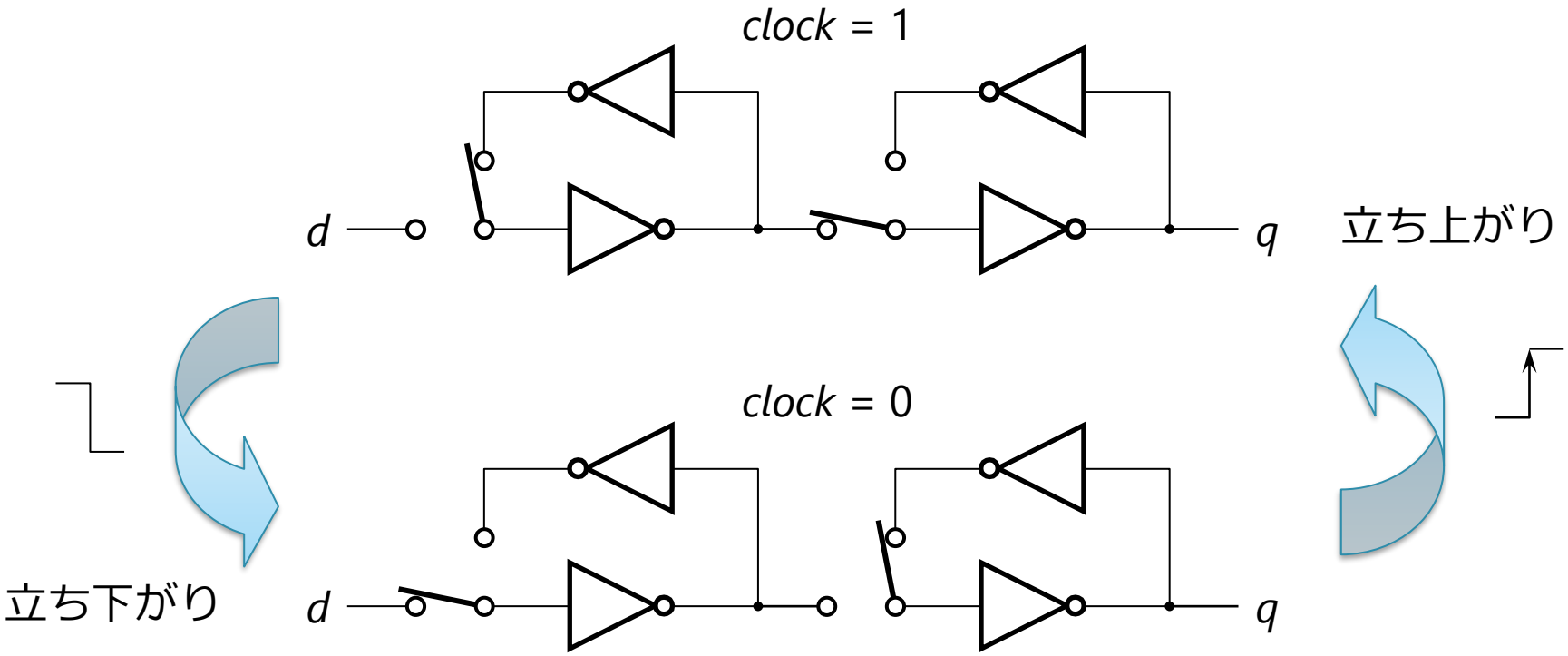
D ラッチの回路

- 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

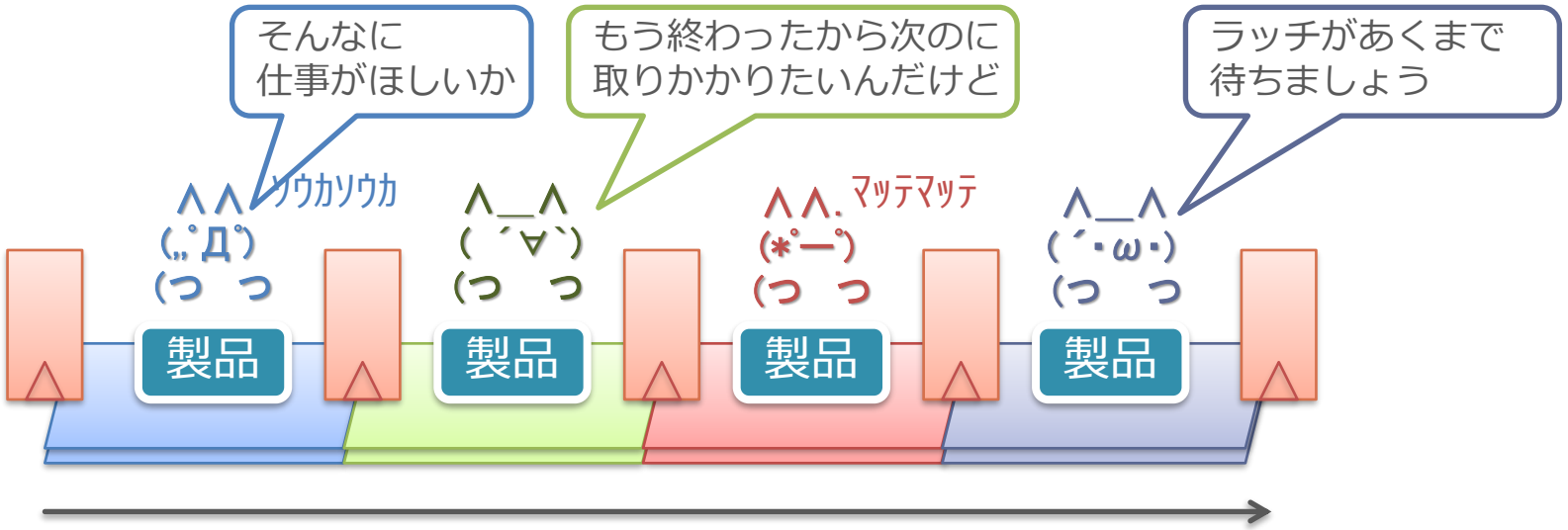
- 構造：D ラッチを2つ繋げたもの
  - ◇ D ラッチ１つだと，半周期は d に入った値が反転して素通しするので使いにくい
  - ◇ ２つ直列に繋げて素通しの期間をなくす



工場のラインを考える（再）

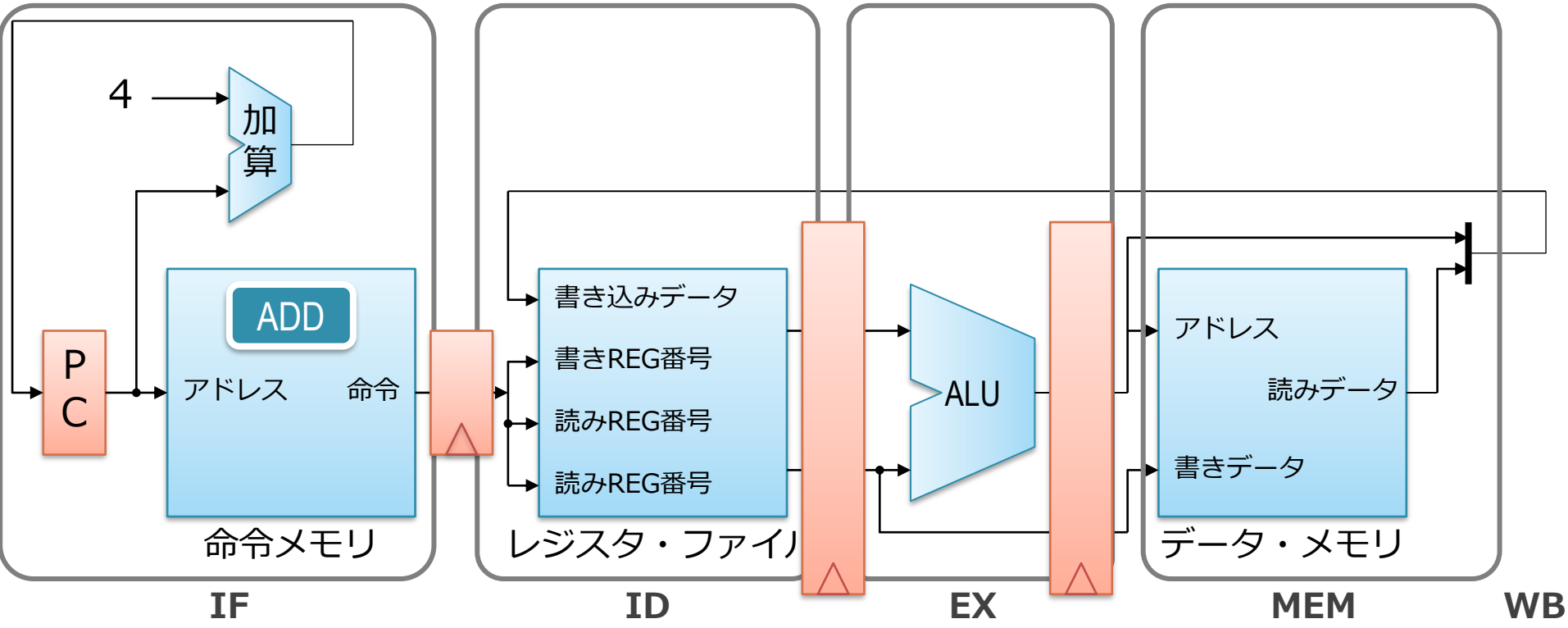
- 実際の工場：複数の製品を同時に流す
  - 各工程を並列して処理することによりスループットを向上
- これが 命令パイプライン

パイプライン・ラッチのイメージ



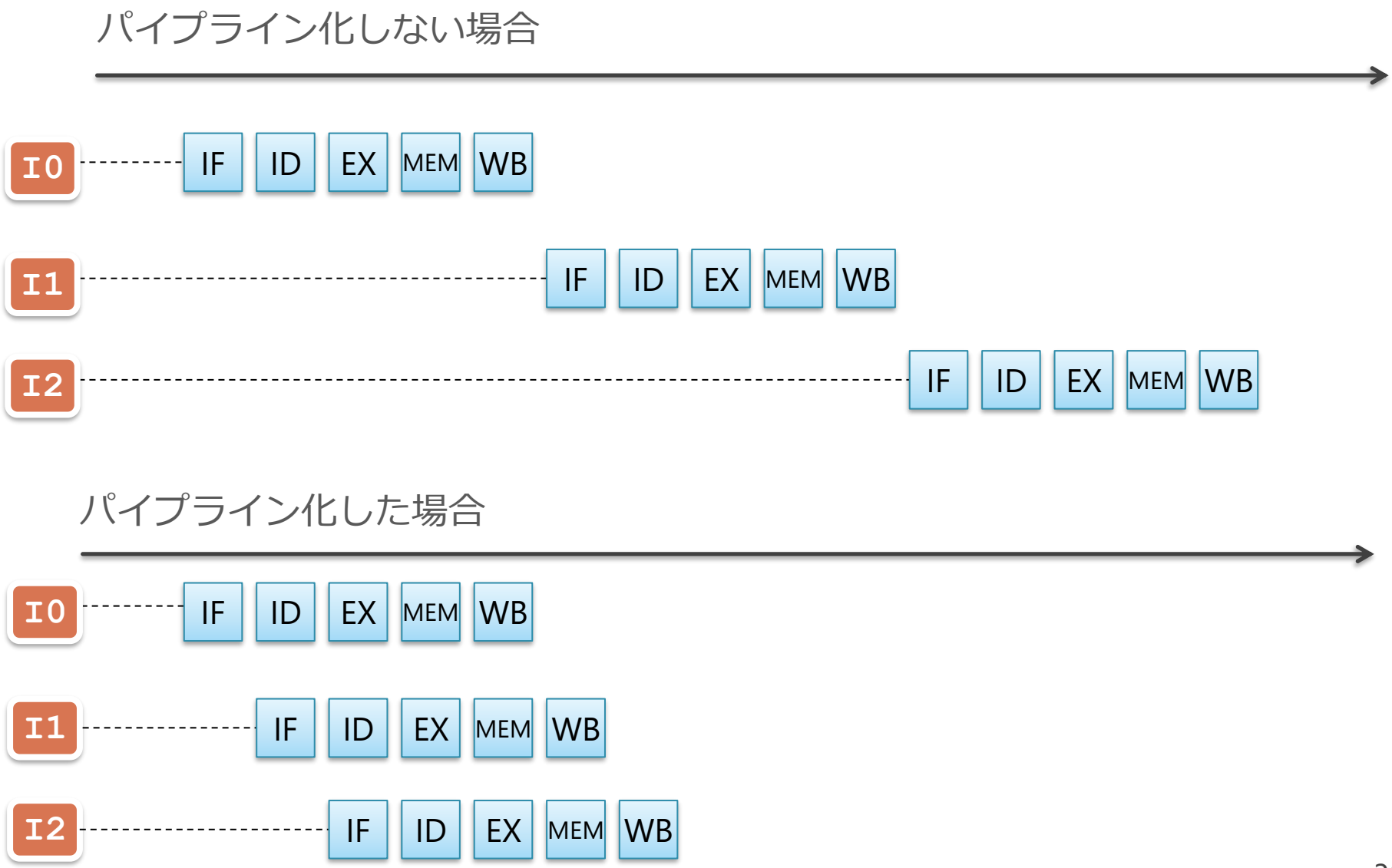
- 各人の作業が終わっても，ラッチが開くまでは次の人に製品を送れない
  - 複数ステージ間で信号が混じるのを防ぐ
  - 指定された時間までラッチでドアを開かなくするイメージ？

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

パイプライン化による性能（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（ハザード）による実効性能の限界**

ハザード

ハザード（hazard）

- パイプラインがうまく動作しないこと
  - パタヘネ（教科書）の定義だと，
    「パイプラインにおいて次のサイクルに次の命令を実行できないこと」

- ハザードの種類：
  1. 構造ハザード
  2. 非構造ハザード
     1. データ・ハザード
     2. 制御ハザード

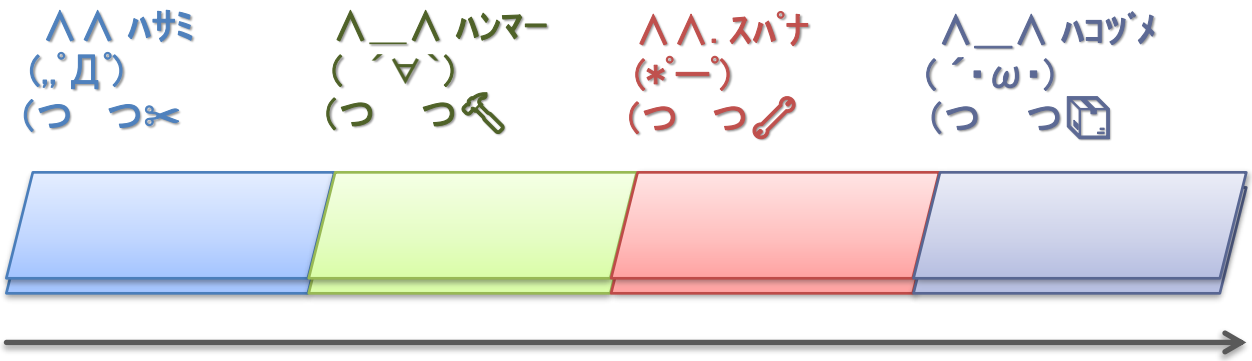
もくじ

構造ハザード

構造ハザード

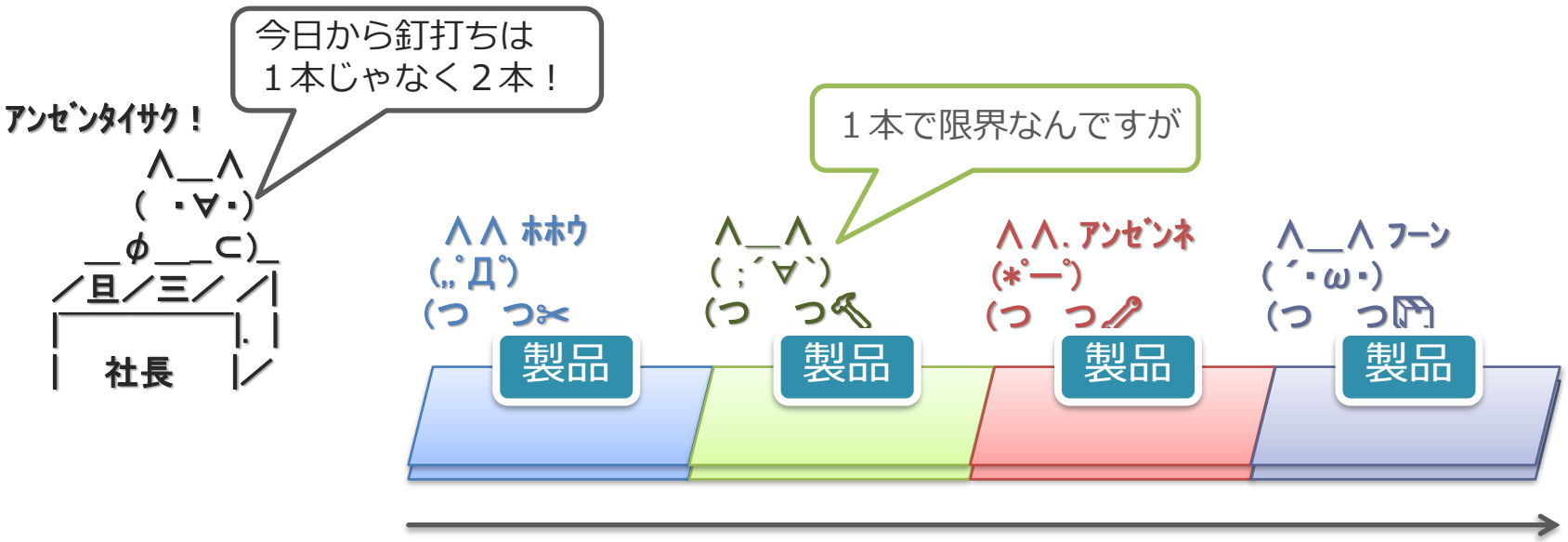
- ハード資源の不足により，パイプラインがうまく動作しないこと
- 工場のラインの例を使って説明

工場のライン



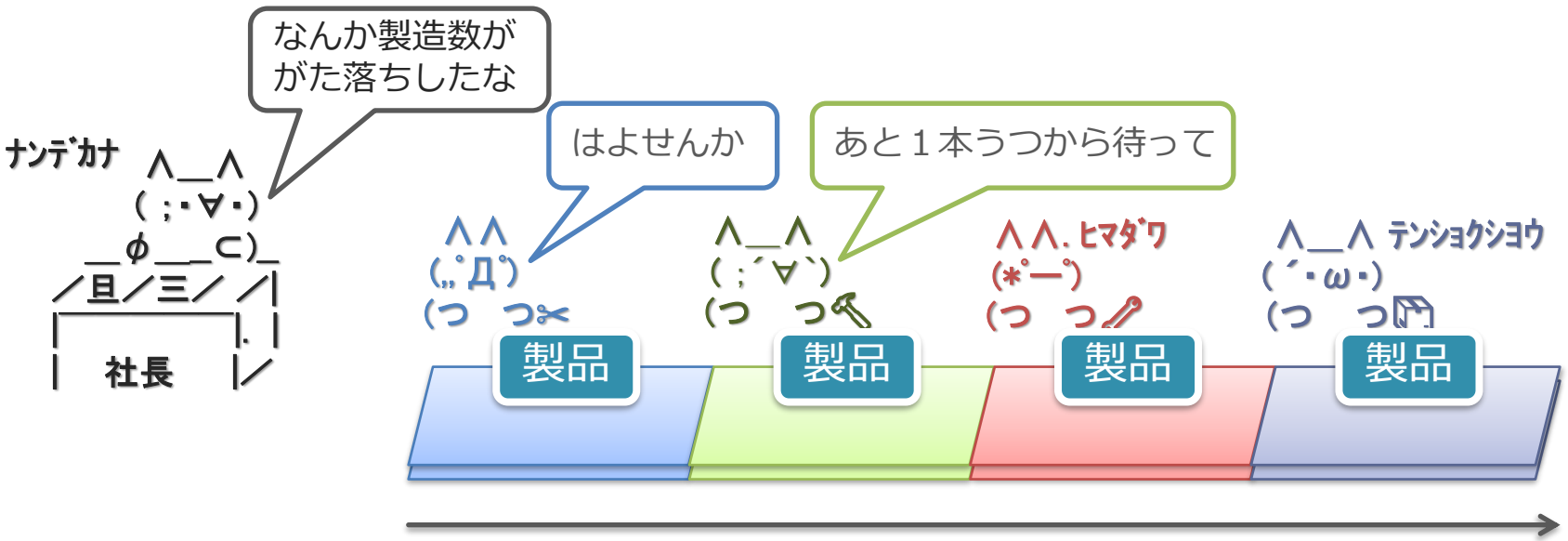
- 各ステージの担当者は製品が流れるまでの間に一定の仕事ができる
  - はさみで紙を１枚切る
  - ハンマーで釘を１本打つ
  - スパナでねじを１個しめる
  - 箱に製品を１つ入れる

釘の本数の変更



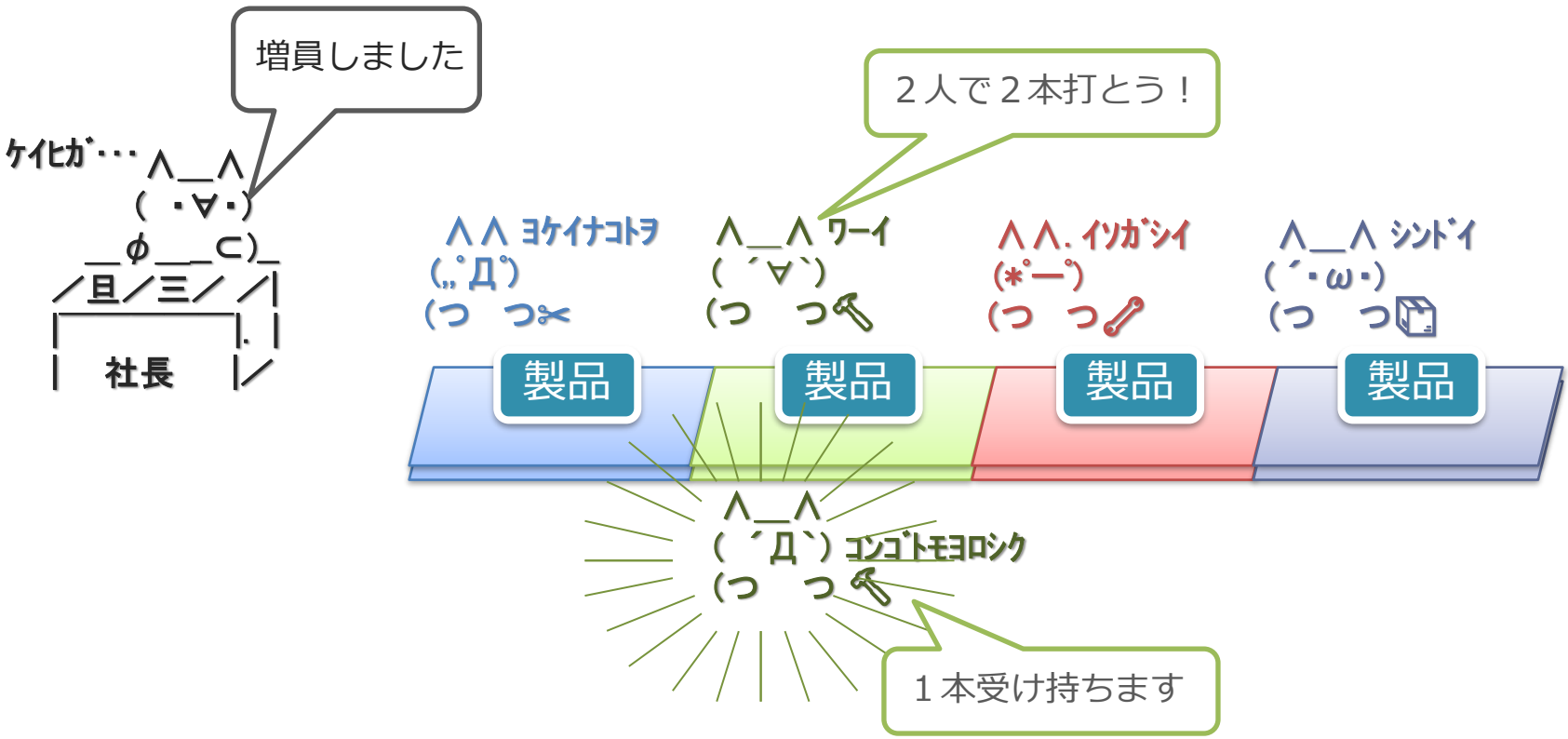
- 製品の安全対策のために釘の本数が１本から２本に
  - しかし，ハンマーを持っている釘打ち担当は単位時間に１本しか打てない

釘打ちの人のところで構造ハザード



- ２人目 (;´∀`) の人は単位時間に１本しか打てない
  - ラインは一番遅い人に合わせて動く（前回の講義より）
  - そこで２本打つまでライン全体が止まる
  - 全体の速度がそこで決まってしまう

増員による構造ハザードの解消



- １人増やして緑のステージで単位時間に２本の釘が打てるように
  - これがハード資源の追加による構造ハザードの解消
  - ただし，追加しただけ経費がかかる
    - （ハードだとその分複雑になって電力を食う

注意

- あくまで資源が足りない（=人員を増やせば解決する）場合についてのみ，構造ハザードという
- 「釘打ちの本数が追加」は構造ハザード
  - 本数が増えてもそれぞれ並列に打てる
  - ステージの人員を増やせば解決できる
- 「釘打ちの後に釘の頭のヤスリ磨きを追加」は構造ハザードではない
  - 釘打ちとヤスリがけは作業に依存関係がある
  - ステージの人員を増やしても速くはできない
  - この場合はパイプライン段数を増やせば解決できる

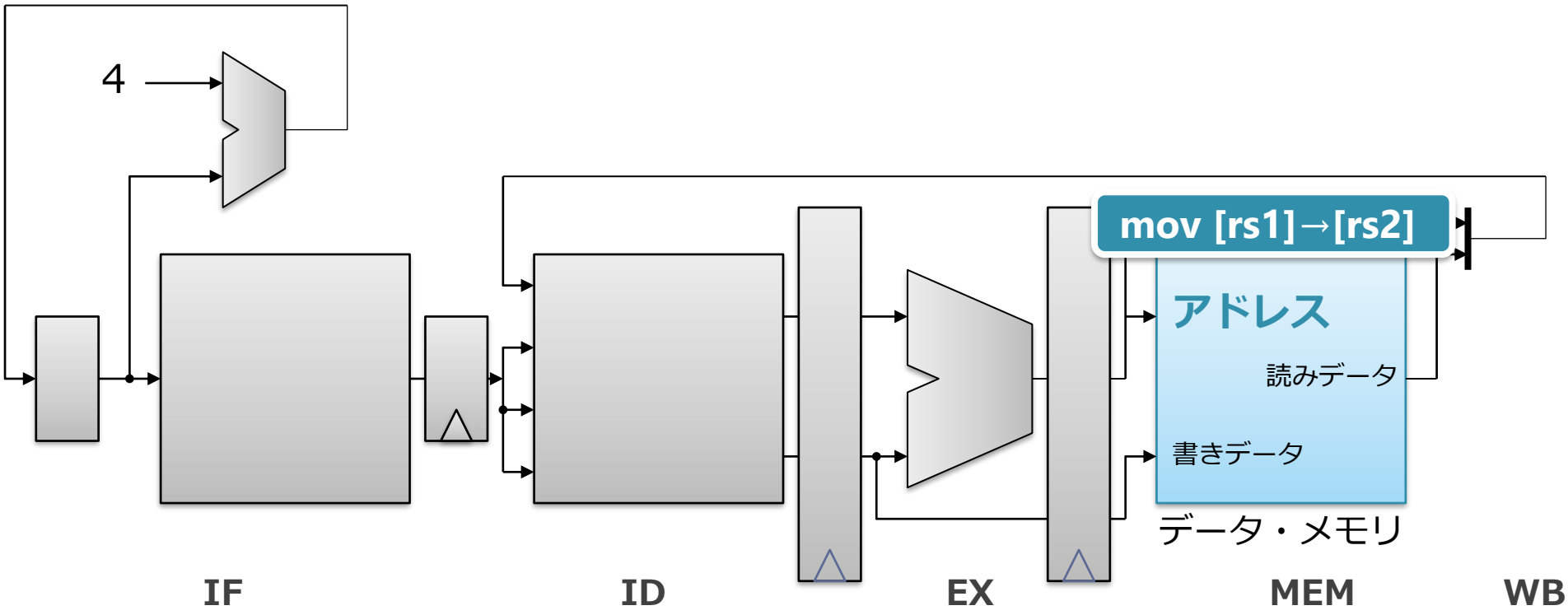
コンピュータにおける構造ハザード

- ハード資源：
  - 演算器（FU），レジスタ・ファイル，メモリ など
- 構造ハザード：
  - ハード資源の不足により，パイプラインがうまく動作しないこと
  - いくつかの例を使った説明，解消方法について解説

構造ハザードの例１：メモリ間 mov

- 例１：仮に mov [rs1]→[rs2] のような命令があったとする
  - rs1 で指定されるアドレスのメモリの値を読んで，
  - rs2 で指定されるアドレスのメモリに書き込む
- 実際に，x86 にはこのような命令がある

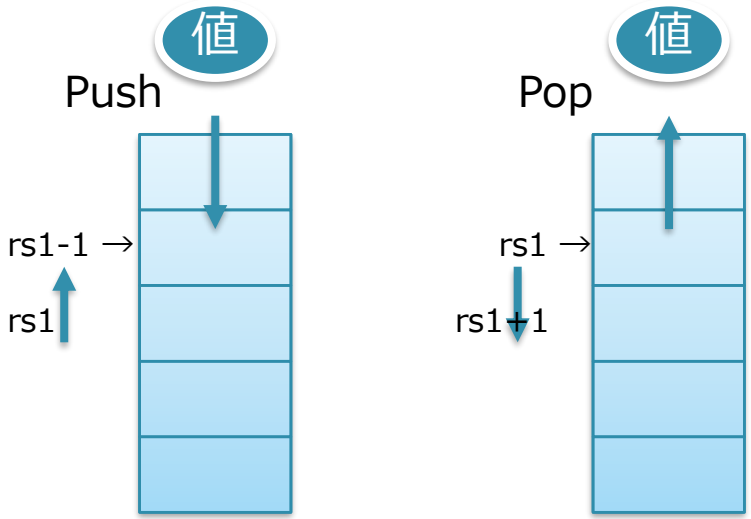
mov [rs1]→[rs2] // [rs1]→[rs2] へのコピー



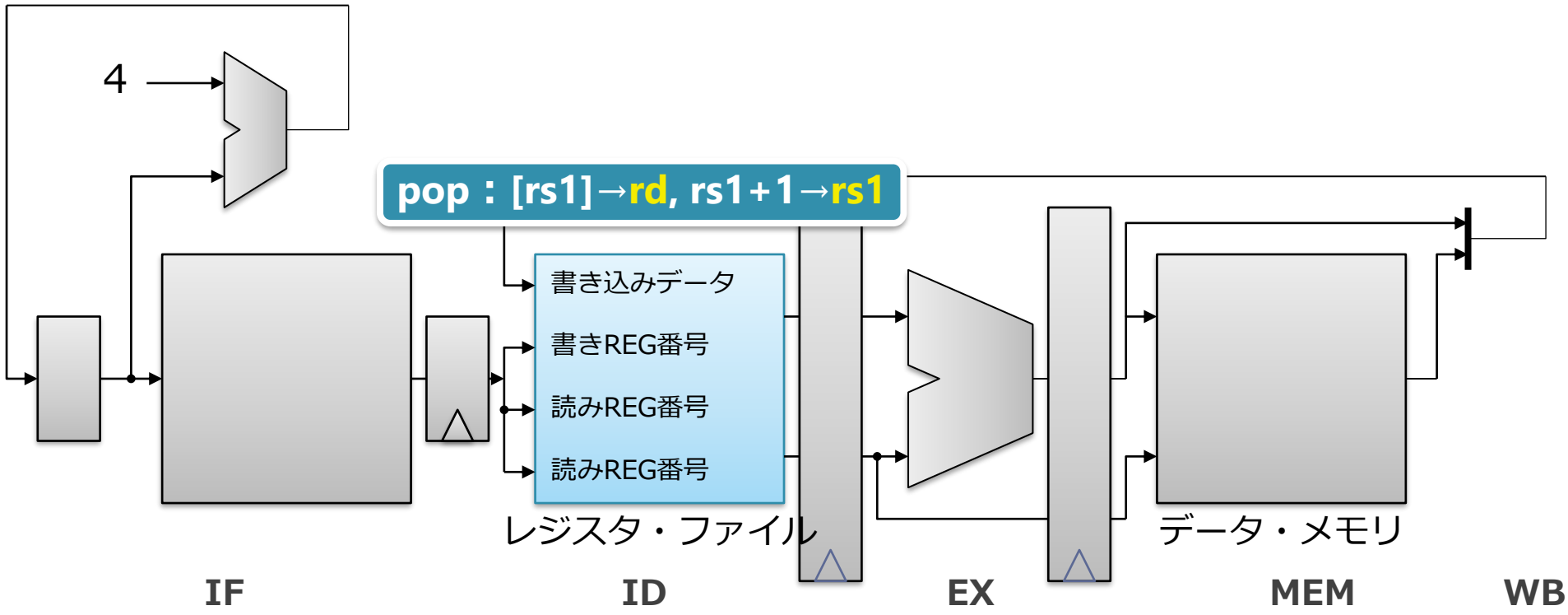
- ある１つのサイクルにメモリを「読んで」「 書く」必要がある
  - しかし，データ・メモリのアドレスの口は1つしかない
  - MEM ステージでデータ・メモリの読みと書きが同時にできない

構造ハザードの例２：push/pop

- x86 や ARM ではスタック操作のための push/pop 命令がある
  - push：rs1-1→rd,　r2→[rd]
    1. スタック・ポインタ（が入ってるレジスタ）をデクリメントし，
    2. それをアドレスにしてメモリに値を書き込む
  - pop：[rs1]→rd, rs1+1→rs1
    1. スタック・ポインタをアドレスにして値を読む
    2. スタック・ポインタをインクリメント



pop：[rs1]→rd, rs1+1→rs1



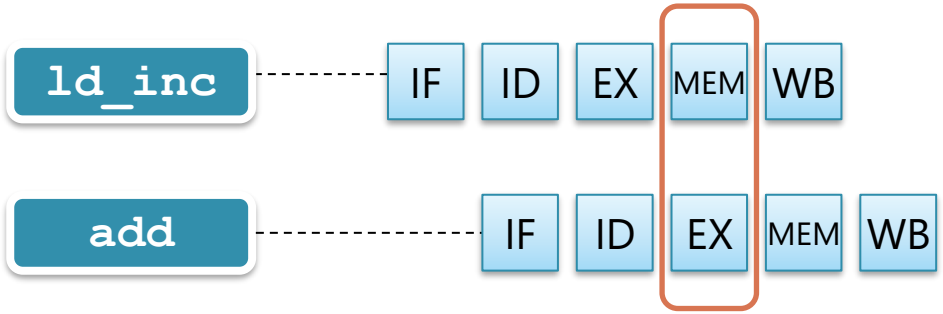
- WB ステージでレジスタに rd と rs1 の2つを書き込む必要がある
  - 2 つのレジスタが 1 つの命令により更新されている
  - レジスタ・ファイルへの書き込みは，同時に 2 つはできない

構造ハザードの例３

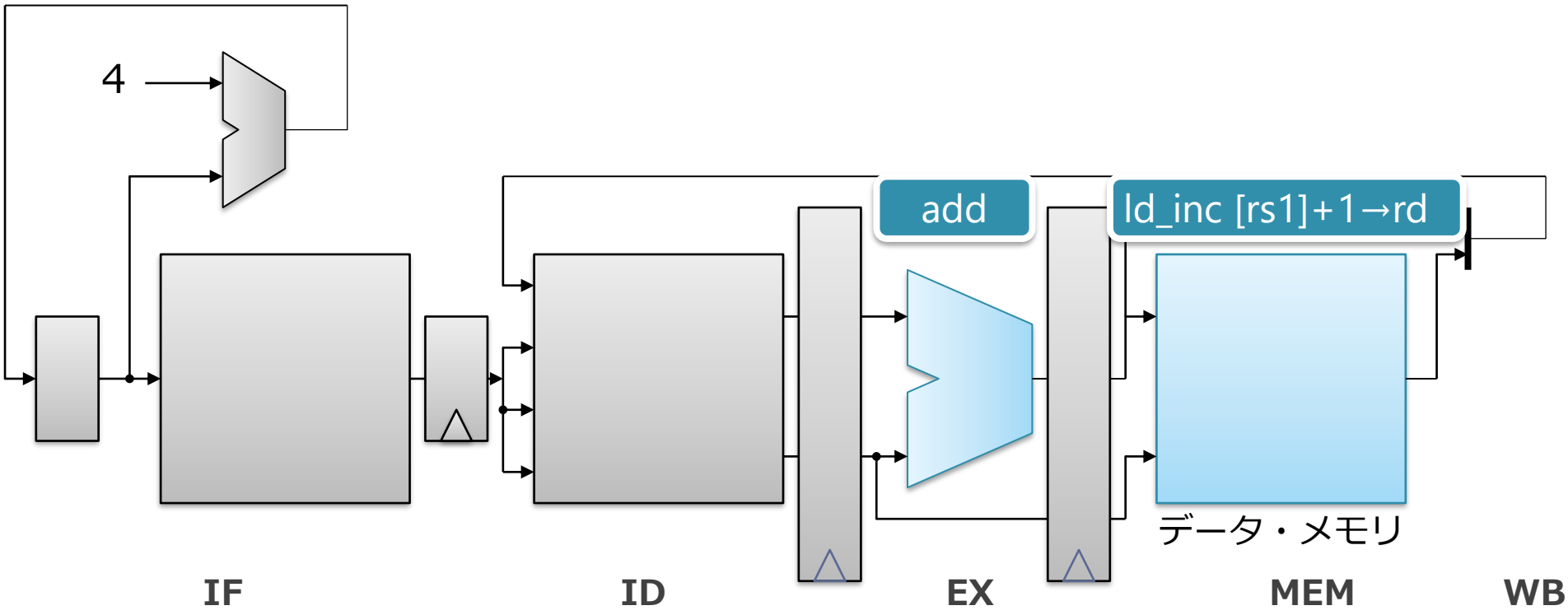
- 使用資源の異なるステージ間のぶつかりでも起きる
  - これまでの例は，同じステージ内で資源が足りない例
- **`ld_inc [rs1]+1→rd`** のような命令があったとする
  1. rs1 の指すアドレスからメモリを読む
  2. 読んだ値にさらに+１してから rd に書く
- 一見，資源は足りているようだが･･･
  - レジスタの読み書きは１つずつしかない
  - メモリも１カ所を読むだけ
  - 加算も１回行うだけ

構造ハザードの例３

- ld_inc [rs1]+1→rd と add が連続した場合：
  - ld_inc で，MEM ステージから読んだ値を加算しようとしても，
  - そのサイクルは後続の add が演算器を使っているので使用できない



ld_inc [rs1]+1→rd と add が連続した場合



- EX ステージ以外では，演算器にはアクセスできない
  - 他の命令が使っている可能性がある

構造ハザードの解決方法

- 解決方法
  1. ハードウェアの増強
  2. 時分割処理

解決方法１：ハードウェアの増強

- ハードウェアを増強する
  - mov [rs1]→[rs2]
    - 複数箇所のメモリを同時に読み書きできるように
  - pop
    - レジスタに２つ同時に書き込めるように
  - ld_inc [rs1]+1→rd
    - MEM ステージに専用の加算器を追加

解決方法１：ハードウェアの増強

- 利点：オーバーヘッドをいとわなければ，基本これで解決
- 欠点：回路規模が増える

1. 機能の増強量に比例した回路が必要
   - なにも考えないで対応していくと，ものすごい数の回路になる
   - 例：ARM は全１６レジスタを一気にメモリに書ける命令がある
2. 機能の増強量に対して，線形より大きなオーダーで回路規模が増える場合もある
   - 加算器などなら，増やした数の分だけ線形に回路が増える
   - メモリやレジスタは，同時に読み書きできる数の２乗で回路が大きくなる性質がある

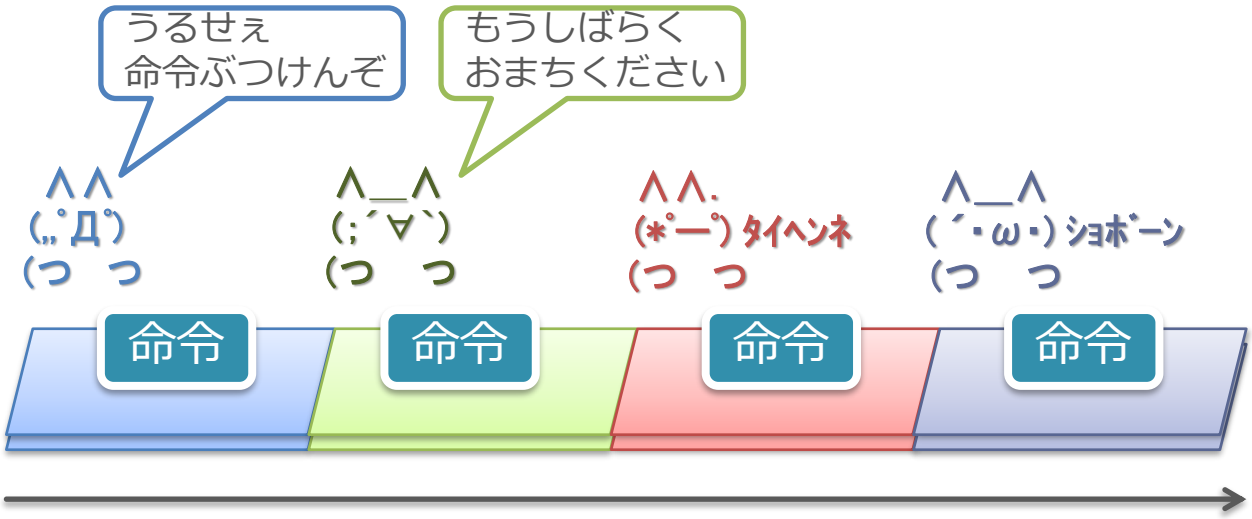
構造ハザードの解決方法

- 解決方法
  1. ハードウェアの増強
  2. **時分割処理**

解決方法２：時分割で処理

- 構造ハザードの原因：
  - ハードウェア（の機能）が足りない
- パイプラインを止めて，複数のサイクルをかけて処理する
  - mov [rs1]→[rs2]
    - メモリを読んだあと，次のサイクルで書きこむ
  - pop
    - １つレジスタに書いたあと，次のサイクルで書き込む
  - ld_inc [rs1]+1→rd
    - ld_inc が MEM で値を読んだら，次のサイクルで +1

なぜパイプラインを止めるのか

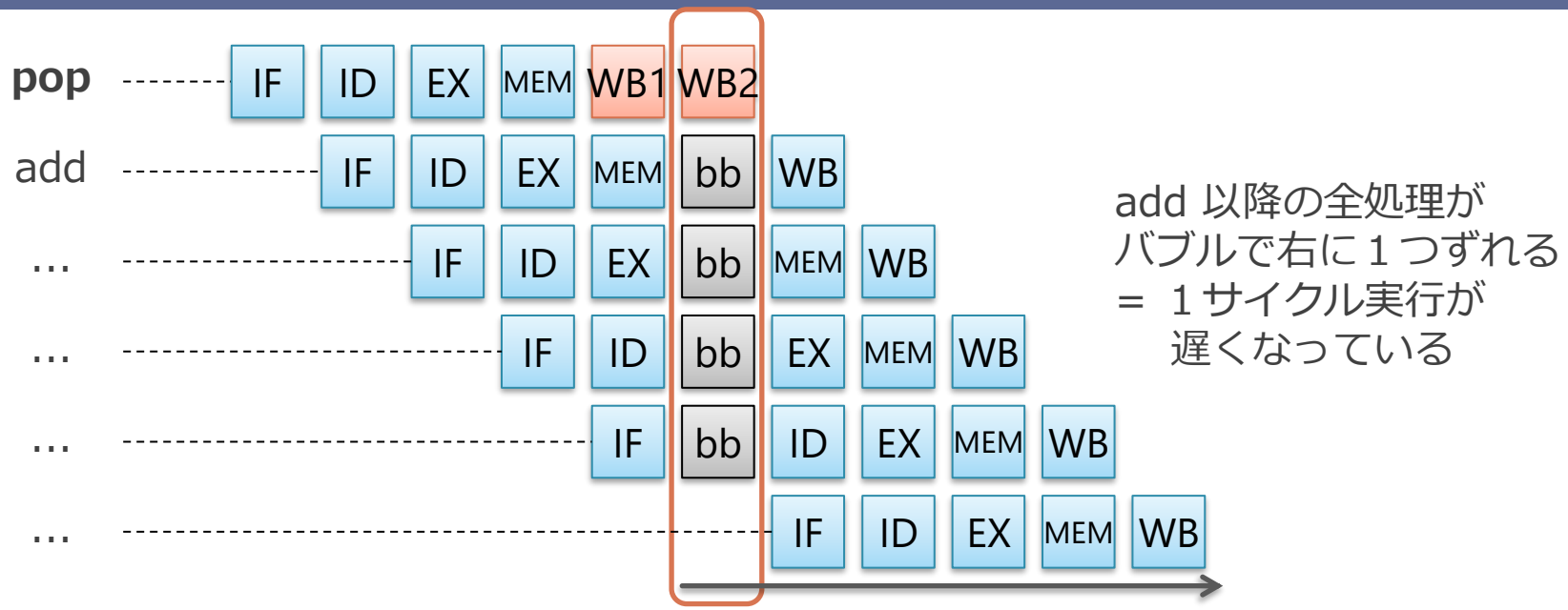


- 上流を止めないと破綻する
  - (;´∀`) が複数サイクルをかけて仕事をしている場合，命令はそこにとどまり続ける
  - その間は上流をとめないと命令をおく場所がないし，依存関係がまもられない
- (*ﾟ―ﾟ) より下流は流れていっても，この場合は問題ない

パイプラインを止めること

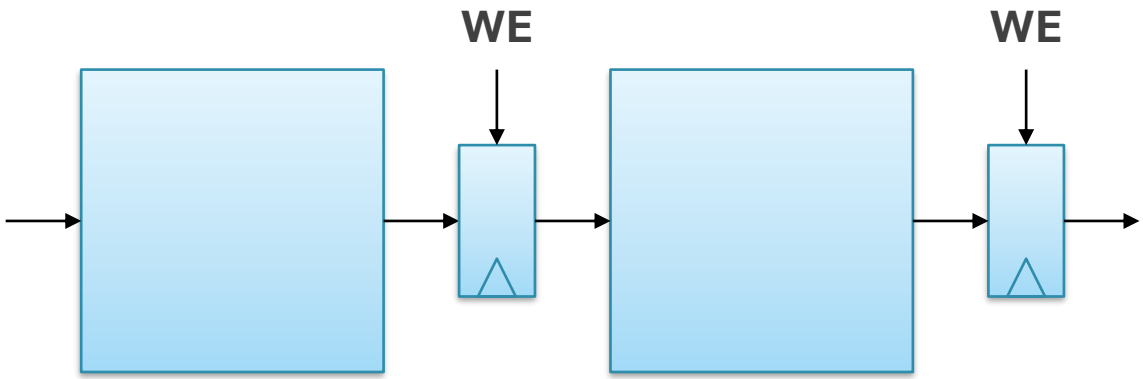
- パイプラインを止めるとことを「ストール」や「インターロック」という
  - 本や人によって，意味や使い方が微妙に統一されていない
  - この講義では，以降はストールで統一

ストールの動作



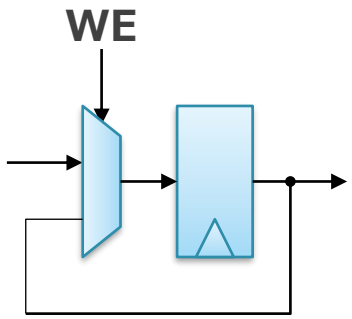
- pop： 1 つレジスタに書いたあと，次のサイクルで書き込む
  - WB1 と WB2 の 2 サイクルで書き込む
  - WB2 の間は上流を全て止める
- パイプライン・チャート上では上記のようになる
  - 止める原因の命令の下が全部右にずれる
  - ずれた部分の空き（bb）を「バブル」とよぶ

ストールの実現方法



- 回路的には，Write Enable（WE）つきの D-FF を使う
  - WE が 0 のサイクルは書き込みが行われない
  - ストールさせたい時は，そのステージの WE を 0 に

WE つき D-FF の実現方法

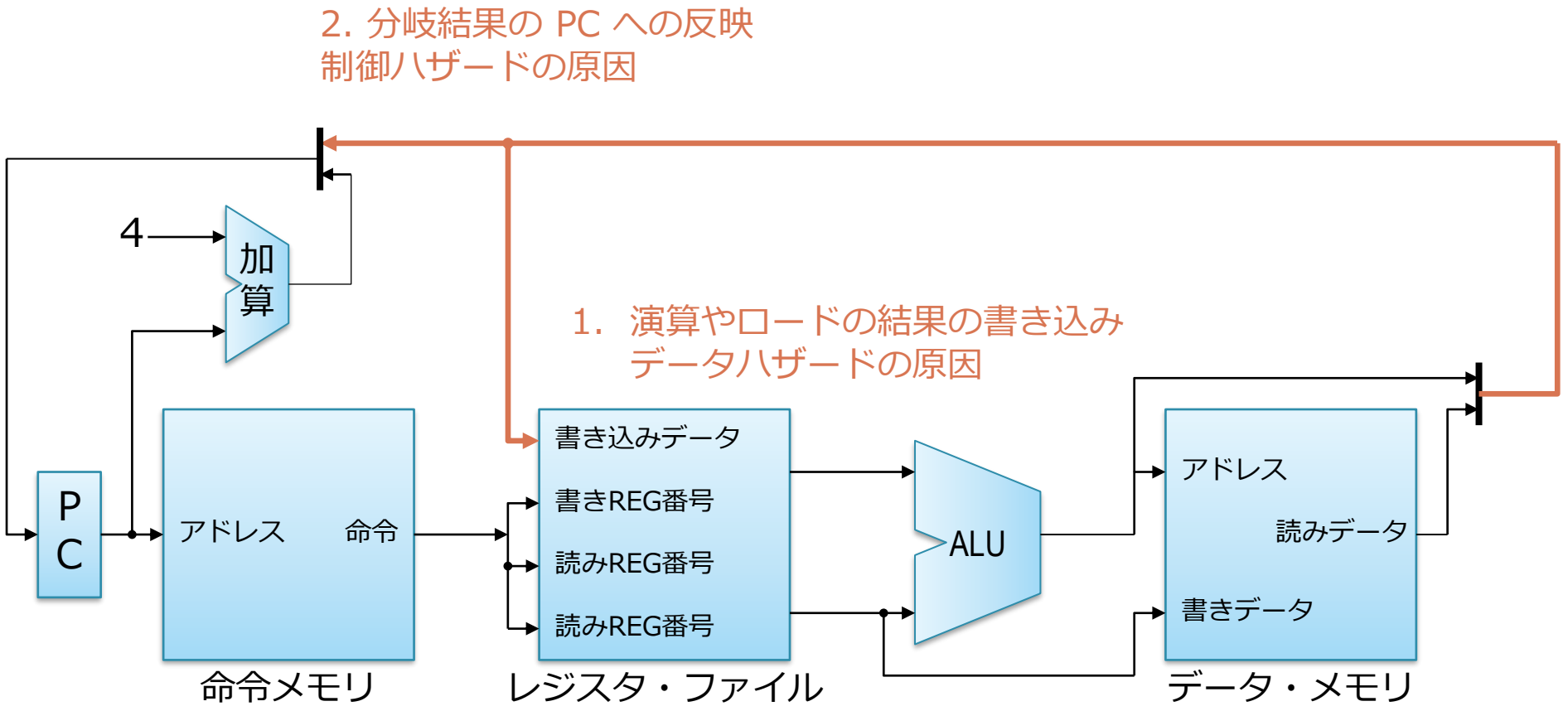


- たとえば D-FF とマルチプレクサで作れる
  - WE が 1 の時は，左からきた入力を選んで書き込む
  - WE が 0 の時は，その時の自分自身の出力を選んで書き込む

非構造ハザード

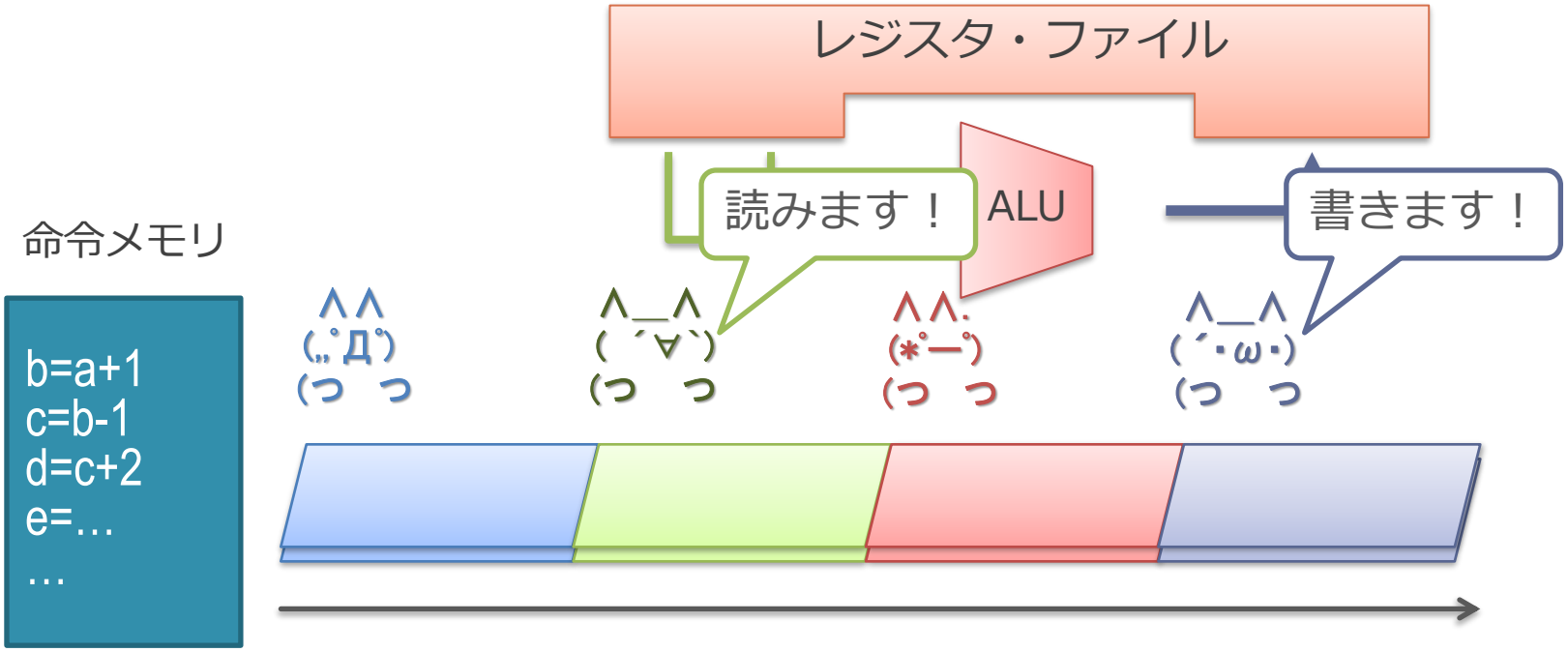
もくじ

バックエッジとは：逆方向（右から左）にいく信号



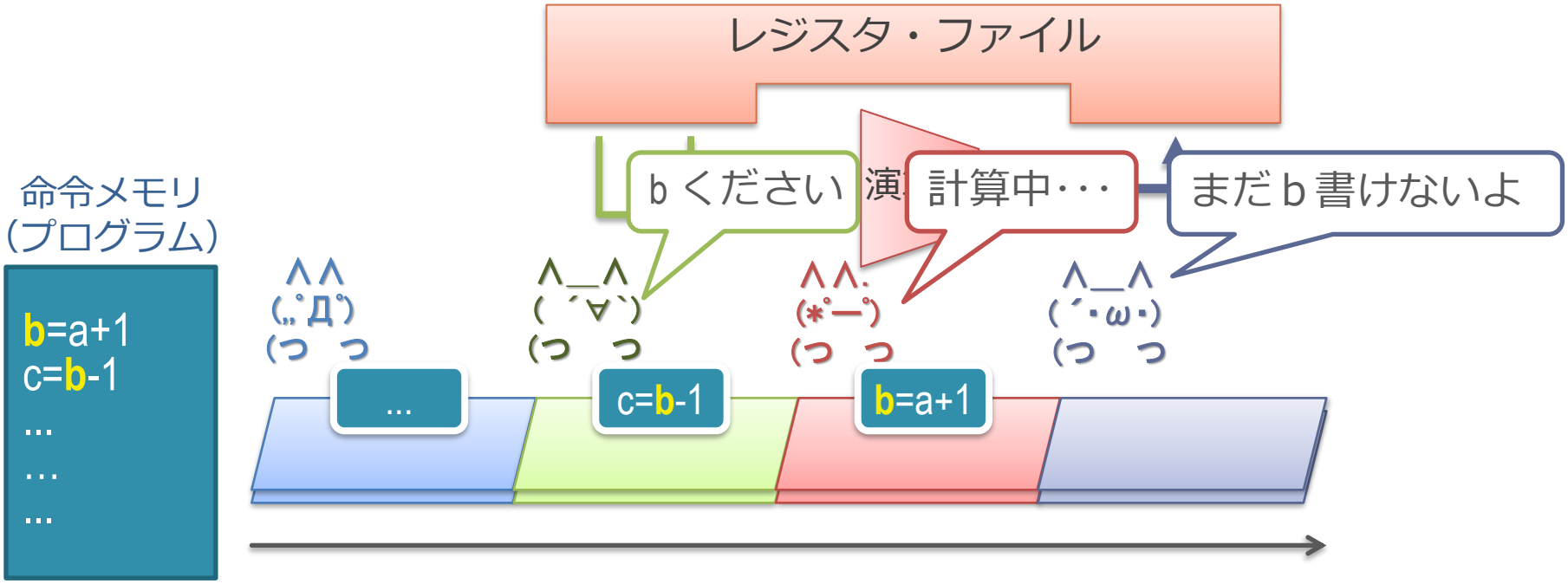
- バックエッジがあるため，命令を単純に流せない場合がある
  - 工場のラインのように，一方向に流せない

データ・ハザード



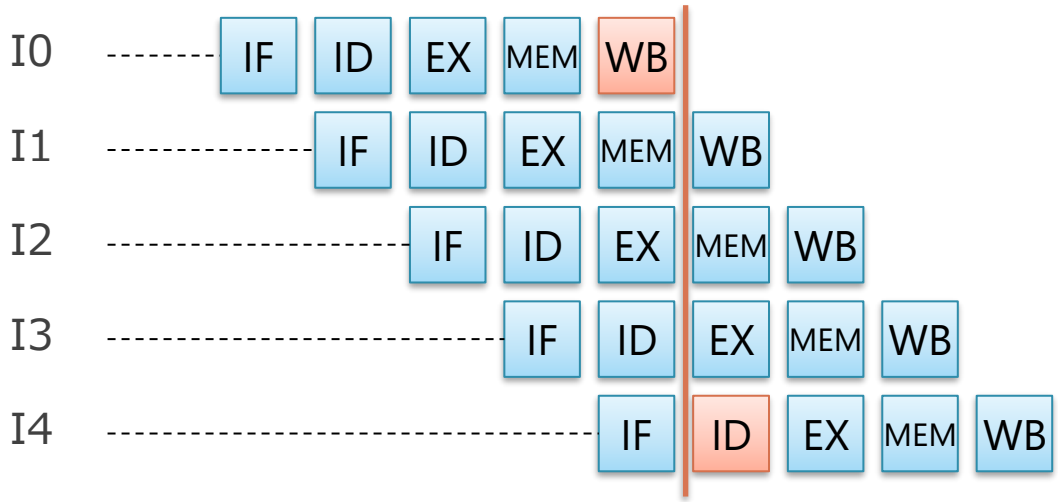
- レジスタ・ファイルへのアクセス
  - 演算の入力は(´∀`) の人がレジスタ・ファイルから読み出す
  - 演算の結果は(´･ω･) の人がレジスタ・ファイルに書き込む

データ・ハザード



- 直前の命令の結果を使う命令が現れた場合：
  - (´∀`) の人が **b**=a+1 の結果を読もうとしても,
  - (*ﾟ―ﾟ) の人がまだ計算中でレジスタ・ファイルに b が書けていない
  - (´・ω・) の人が計算結果をかけるのはさらに次のサイクル

データ・ハザード

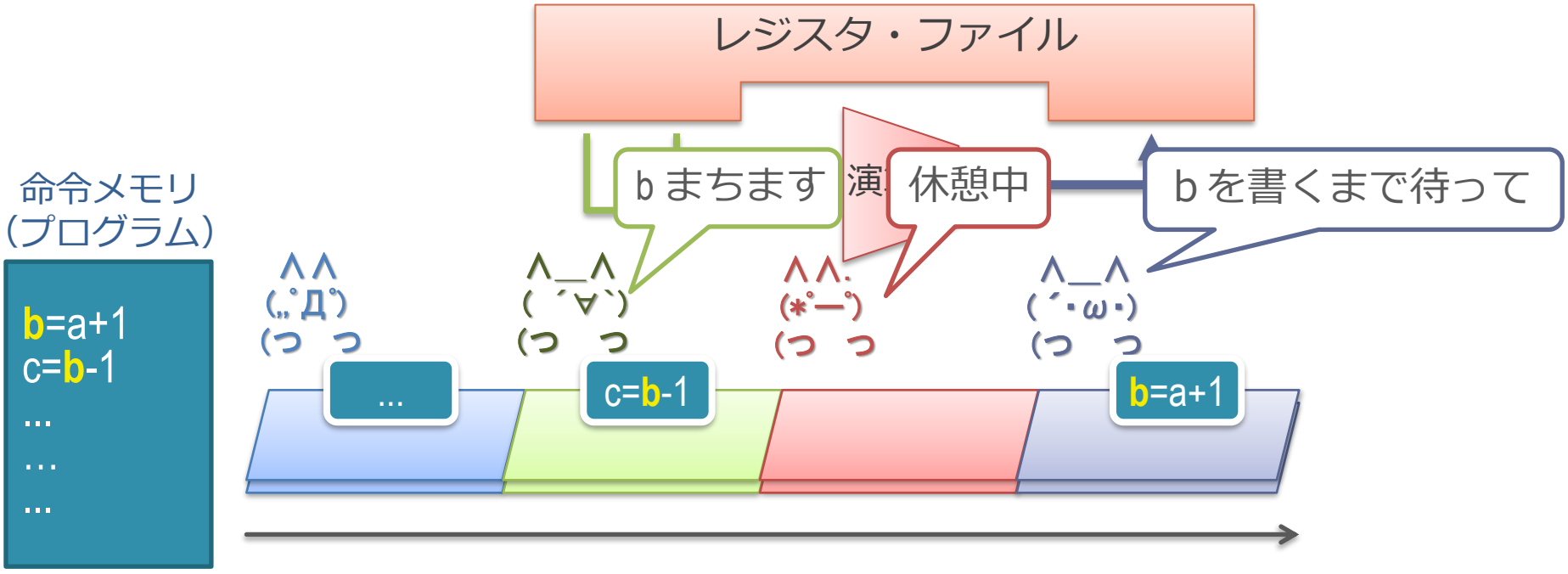


- I0 の WB が終わるまで，その結果はレジスタに書き込まれない
  - I4 までは，その値がレジスタから得られない
  - ID ステージでレジスタを読むため

データ・ハザードの解消方法

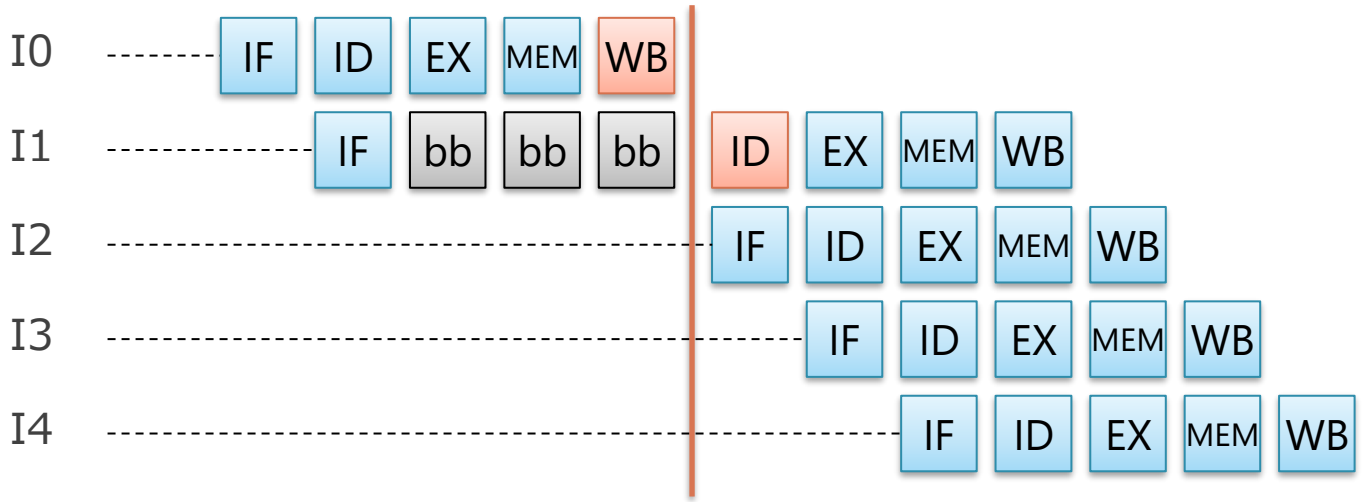
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. フォワーディング

1．ストール



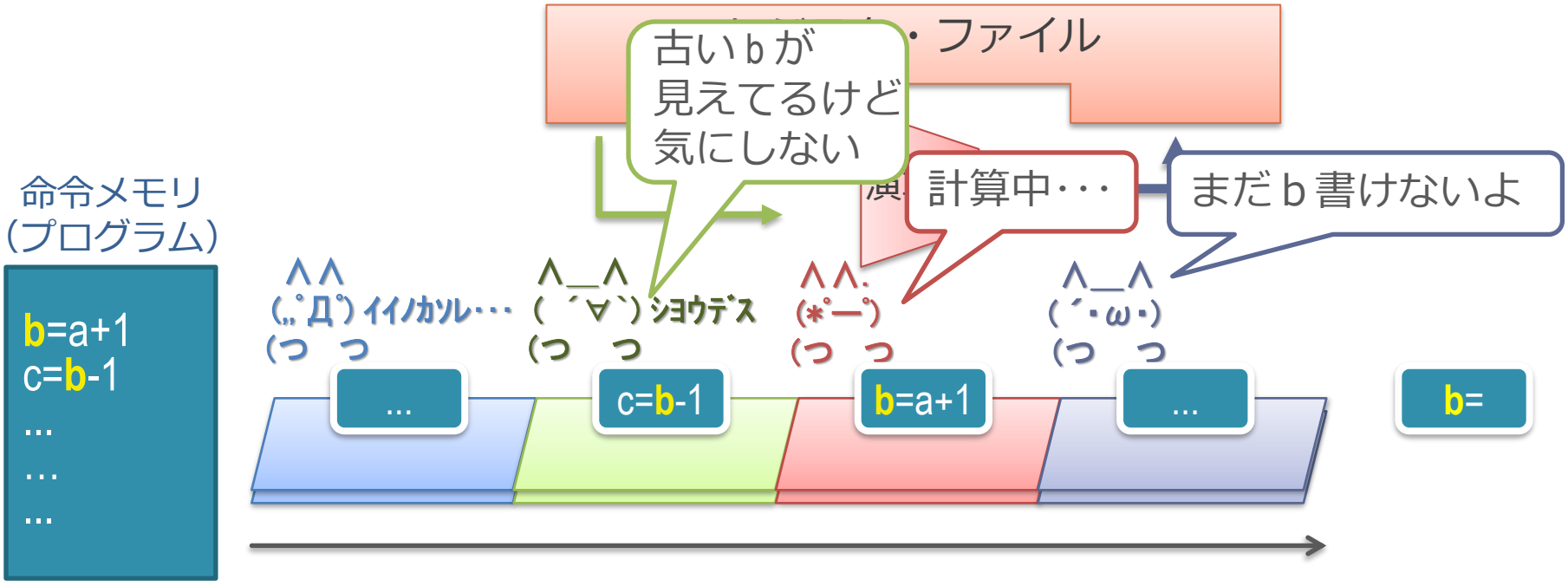
- 直前の命令の結果を使う命令が現れた場合：
  - (´･ω･) の人が計算結果をレジスタに書くまで，
    (´∀`) の人と上流のラインを止める
  - 間にバブル（何もしないステージ）が入る

1．ストールさせる



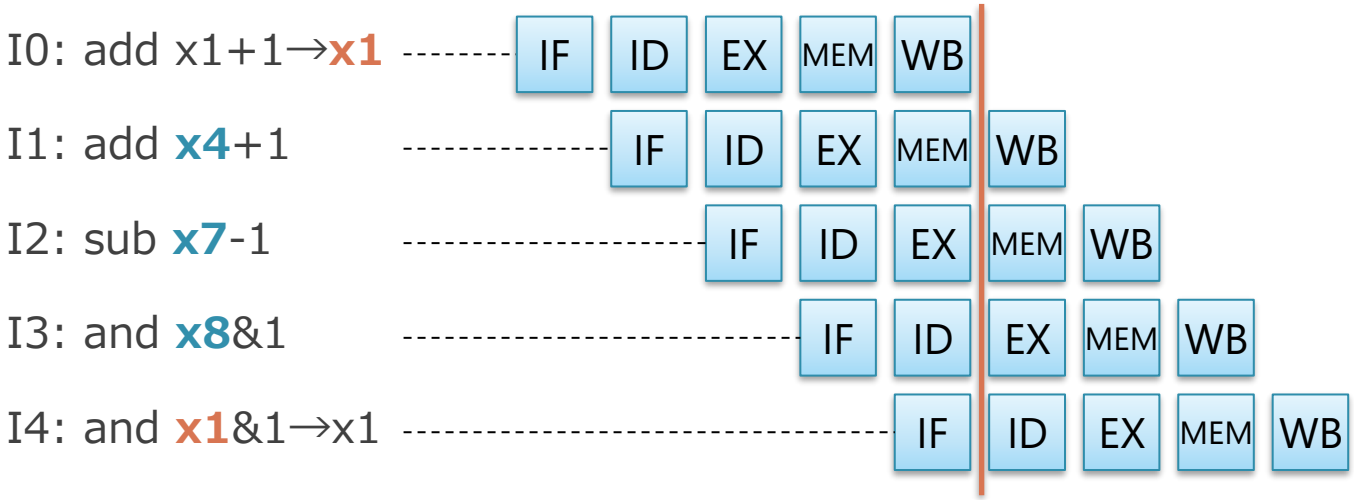
- I0 の WB が終わるまで，後続の命令を遅らせる
  - I1 の ID が，I0 の WB の右にくるまでストール
  - I1 は I0 の結果を使える
- 欠点：プログラムの実行がとても遅くなる

2. 遅延スロット（なにもしない）



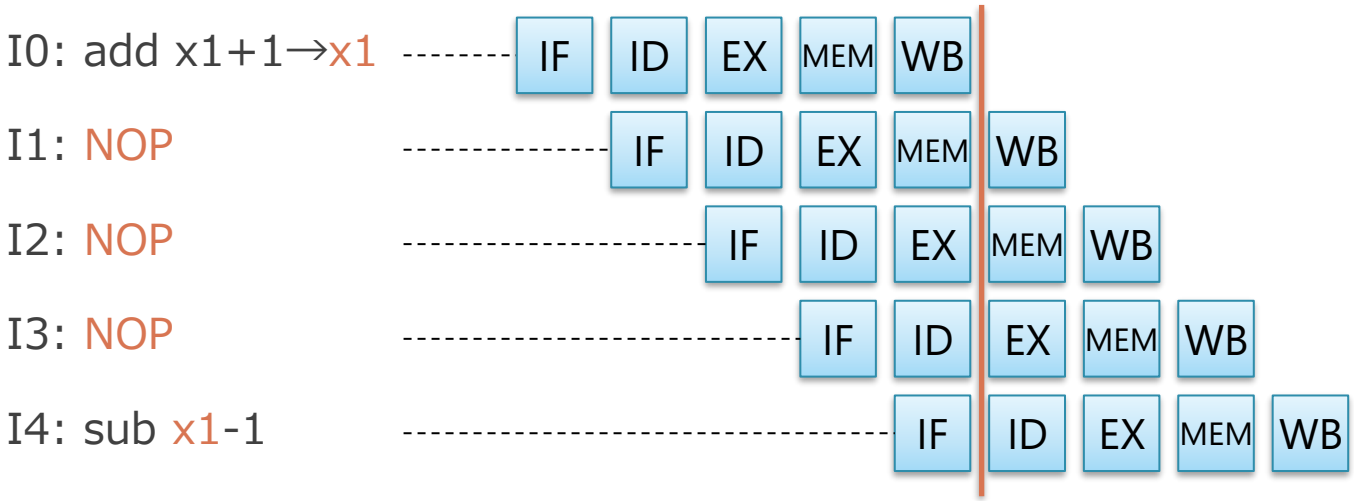
- 直前の命令の結果を使う命令が現れた場合：
  - (´∀`) の人は，その時レジスタ・ファイルにある b を気にせず読んでしまう
  - ２つ前の命令の結果の b が読めてしまう
    - そう言う仕様ということにする

2. 遅延スロット（なにもしない）



- ここに I0 の結果を使わない命令を入れれば，性能低下はない
  - この直前の命令の結果が見えない部分を「遅延スロット」と呼ぶ
    - この図では，遅延スロットが 3 命令分ある
    - それらは I1, I2, I3 は I0 の結果を使っていないので問題無い
  - 遅延スロットへの命令挿入はコンパイラががんばる
  - 人力でアセンブリ言語でがんばることもある

NOP の挿入



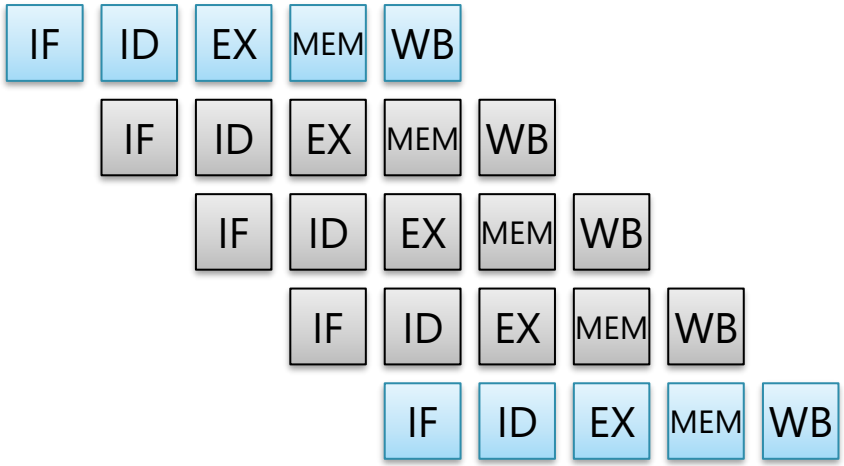
- もしそのような命令がない場合，
  - NOP（No Operation）と呼ぶ何もしない命令をいれる
  - これもコンパイル時にいれておくる必要がある
- 上の例は，x1 に 1 を足した結果を使う以外の処理がなかった場合

遅延スロットの利点

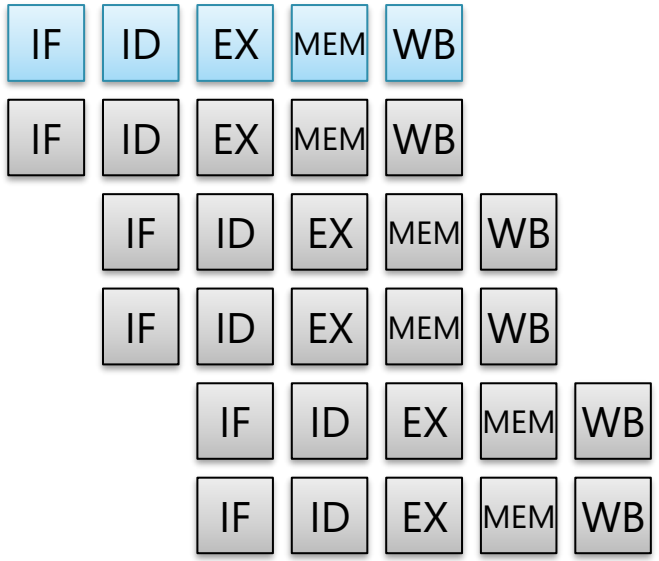
- 利点：
  - なにもしないので，ハードは最も単純
  - 並列にできる命令があれば，性能も下がらない

遅延スロットの欠点

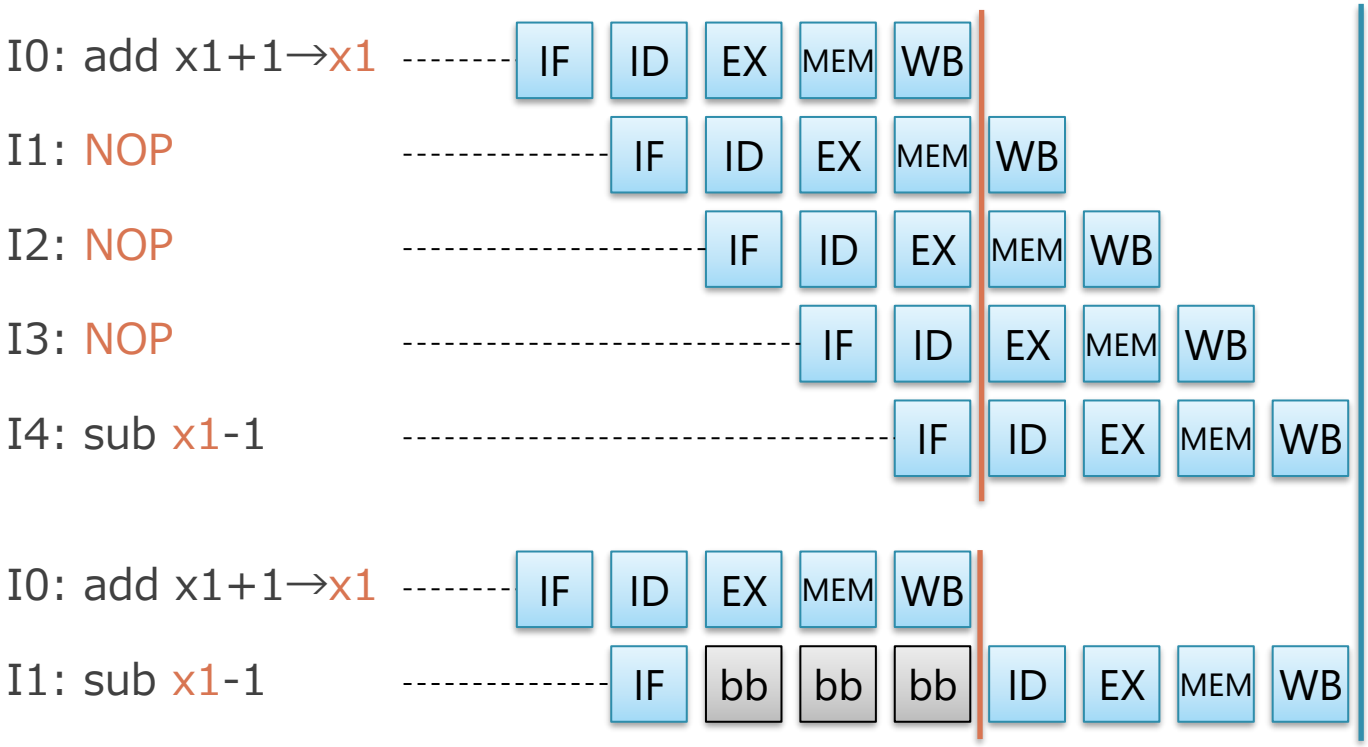
- 欠点：「仕様」なので，一度決めると変えられない
  - 後からパイプラインの段数や構造を変えると互換性がなくなる
    - クロックをあげるために，段数を増やせない
  - 複数の命令を同時処理しようとしたときにも互換性がなくなる
  - MIPS では遅延スロットが 1 命令分，仕様として存在
    - 互換性のためにこれを忠実に再現するため後年は逆に複雑化



2 命令同時処理すると，遅延スロットが増える



遅延スロットの欠点２

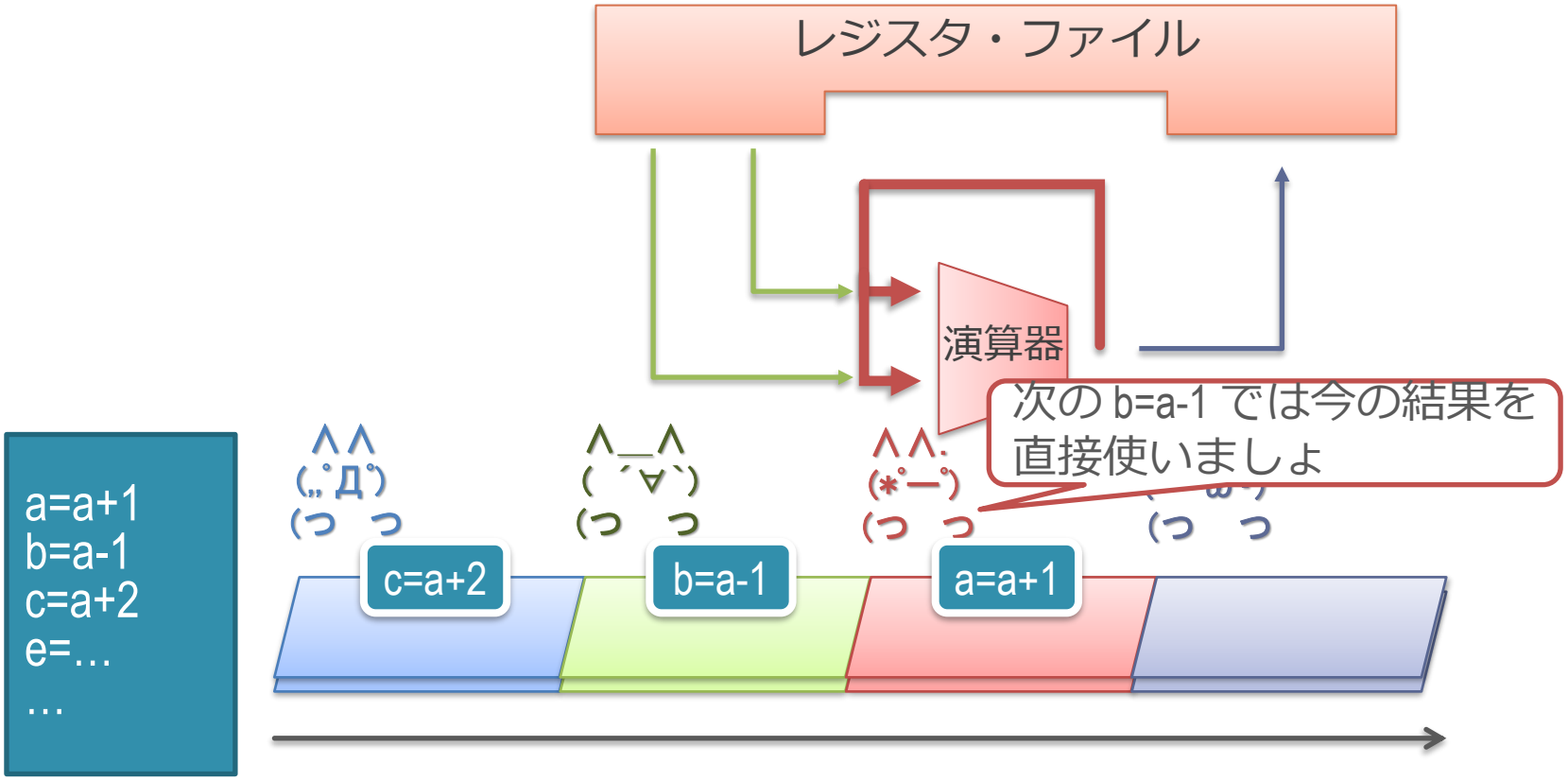


- 欠点２：並列してできる命令が常にあるとは限らない
  - NOP を入れるしかなくなる
  - 実質ストールしてバブルを入れるのと同じになってしまう

データ・ハザードの解消方法

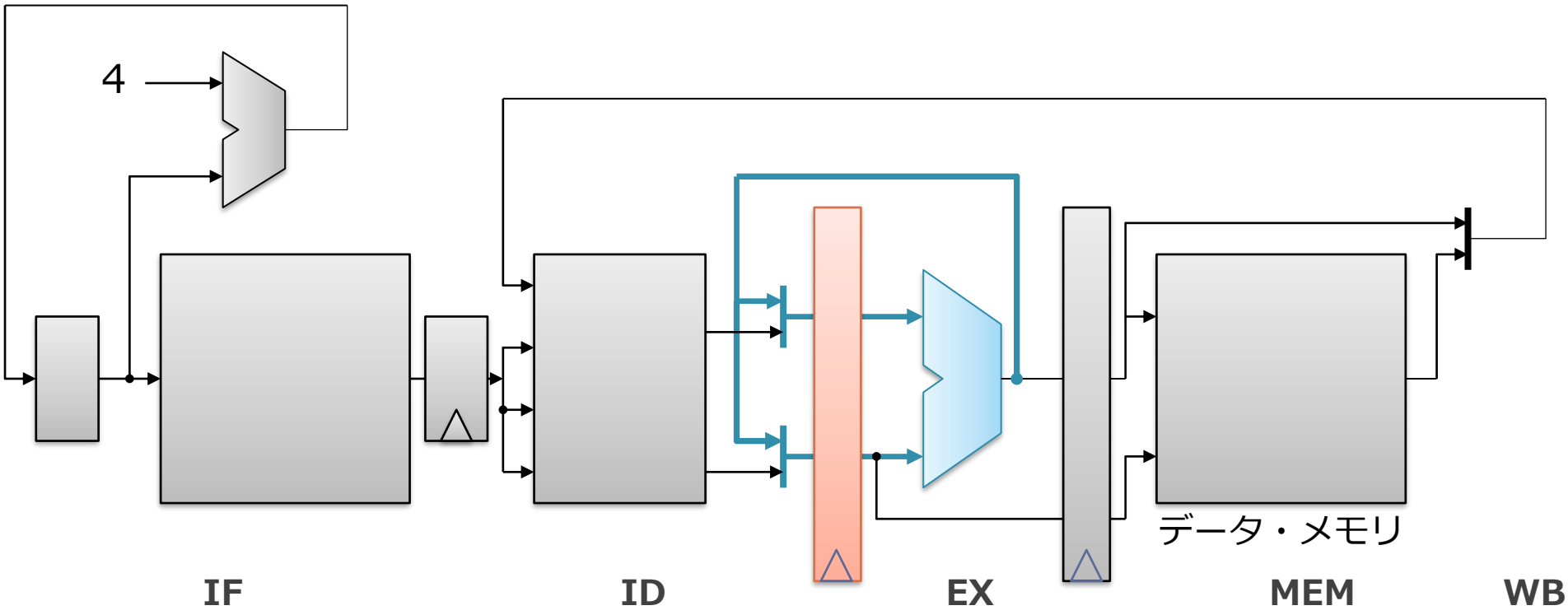
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. **フォワーディング**

フォワーディング



- フォワーディング（バイパスとも呼ぶ）
  - (*ﾟーﾟ) の人が，次のサイクルに自分の計算結果を即座に使えるよう，手元にも結果を置いて使う
  - (´∀`) がレジスタ・ファイルから読んできた値は必要に応じて捨てる

フォワーディングの回路

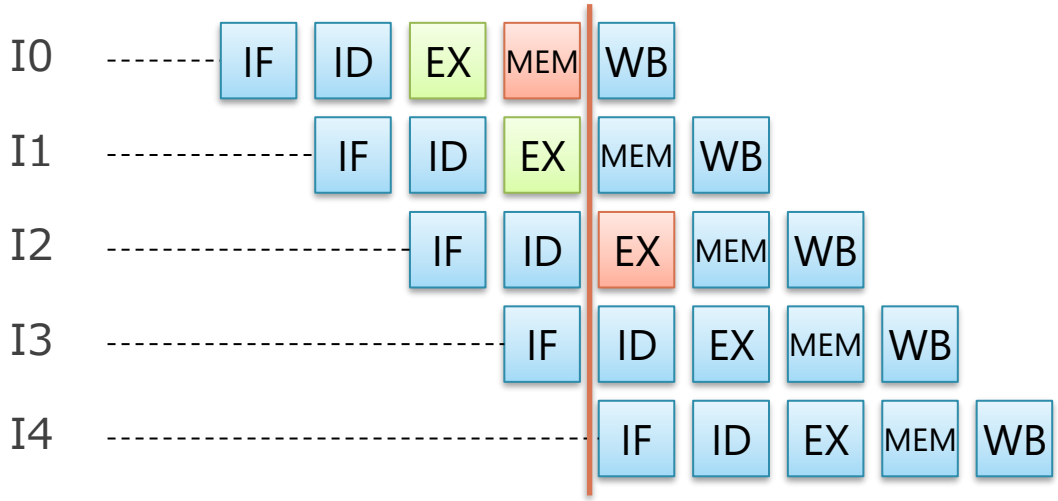


- 演算器の結果を，フィードバック
  - レジスタ・ファイルからの読み出し結果と選択して入力に

フォワーディングの利点

- 利点：
  - 依存関係がある命令が連続できてもパイプラインを動かし続けられる
  - バブルを発生させることがない

ロードについては，完全に解決はできない



- ロードではデータ・メモリを読むまでその値は取れない
  - 次の命令は，MEM より後に EX がこないといけない
    - I1 は，I0 のロード結果が見えない
  - この部分はストールや遅延スロットでなんとかすることがおおい

制御ハザード

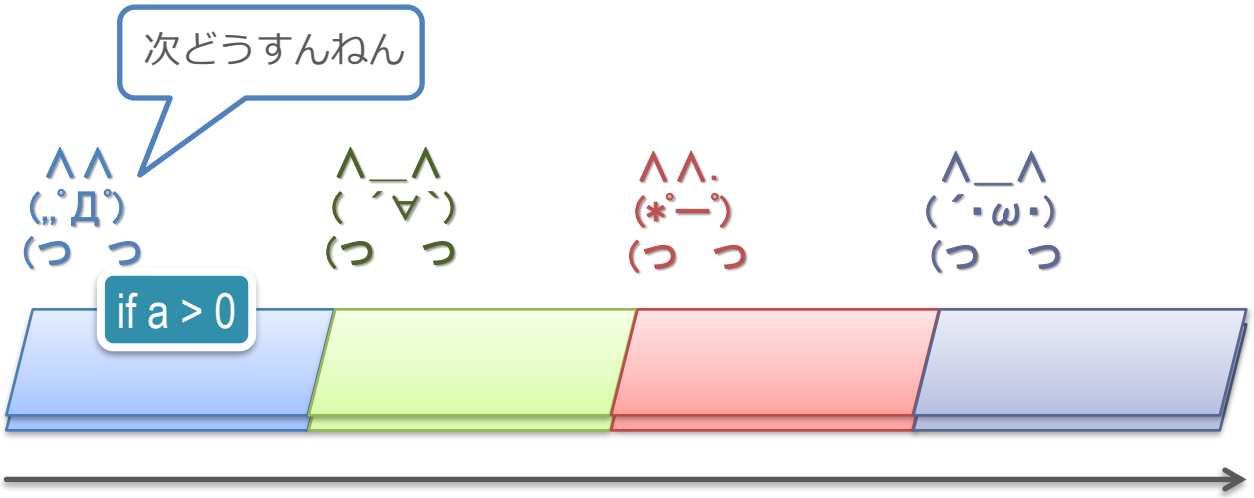
1. 構造ハザード

2. 非構造ハザード：バックエッジに由来
   a. データ・ハザード
   b. **制御ハザード**

分岐命令の処理と制御ハザード

命令メモリ

```
if a > 0:
    a=a+1
else:
    a=a-1
```



- 「if a > 0」の結果は最終段の (´･ω･) の人まで反映出来ない
  - 先頭は次に a=a+1 と a=a-1 のどちらを取り込めばいいのかわからない

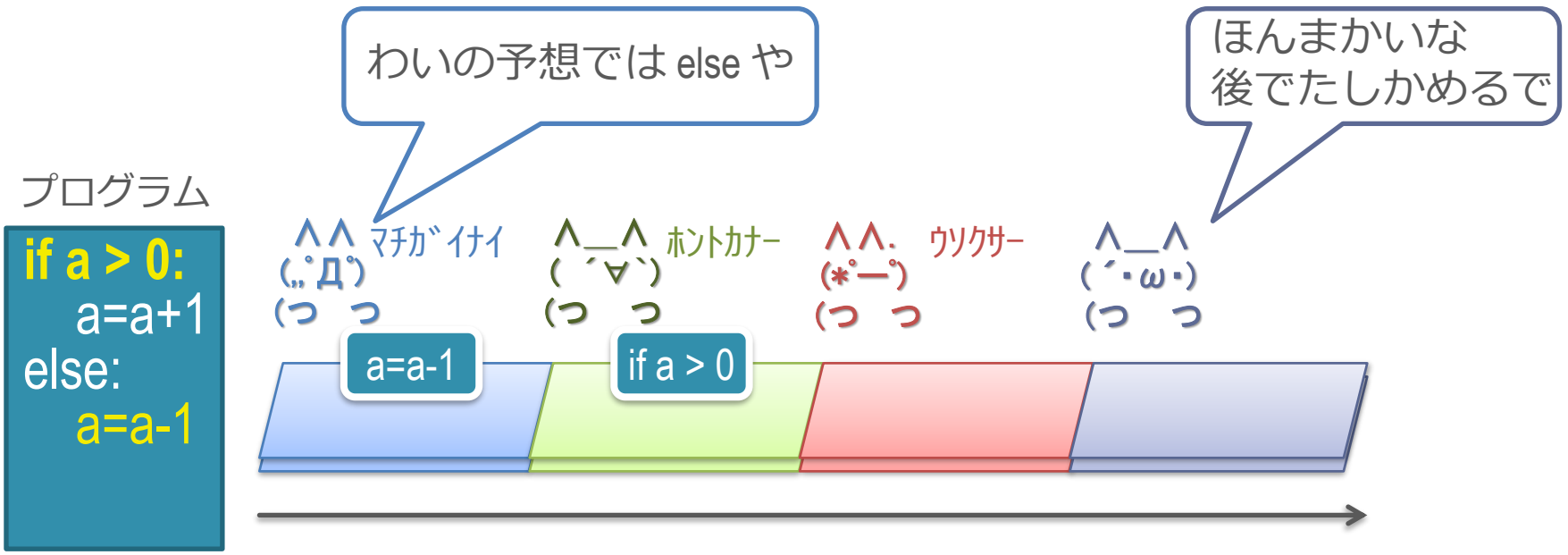
制御ハザードの解消方法

- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）

  - 上記は基本的にデータ・ハザードと同様にして適用できる
    - ただしフォワーディングは，制御ハザードでは意味的に無理

  3. **分岐予測による投機実行**

分岐予測

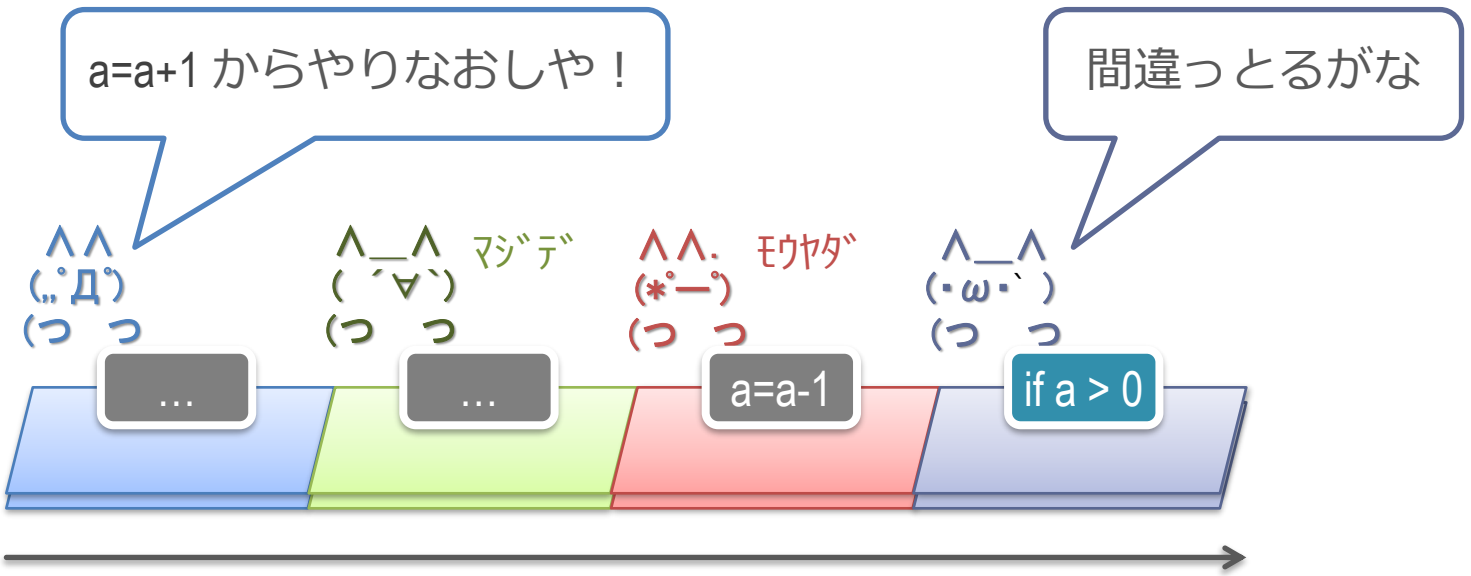


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

分岐予測ペナルティ

プログラム

```
if a > 0:
    a=a+1
else:
    a=a-1
```



- 予測が間違っていた場合，以降の処理を取り消してやり直す
- この図では，無駄になるのは3命令分

大規模な高性能プロセッサの場合

- 取り消しは最悪数十命令以上に
  - IBM POWER8 と言う CPU だと，8命令同時 × 10数段

パイプライン化の限界

- 速度が上がらなくなる理由：
  1. 回路的な理由による周波数向上の限界（前回の講義）
  2. **アーキテクチャ的な理由（ハザード）による実効性能の限界**

アーキテクチャ的な理由による実効性能の限界

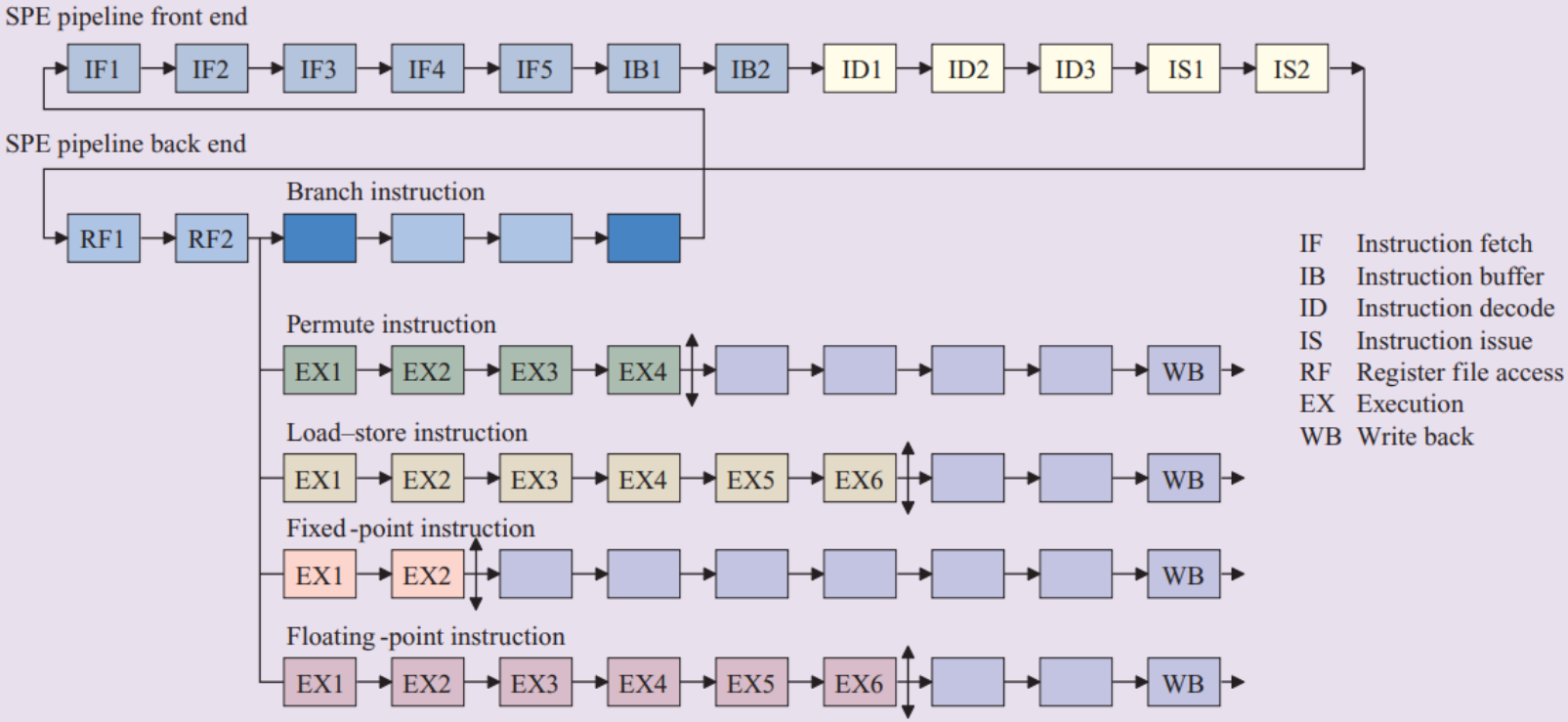
- バックエッジがないパイプライン
  - （回路的な限界にあたるまでは
  - パイプライン段数を増やせば増やすほど性能（周波数）が上がる
- バックエッジがあるパイプライン
  - パイプライン段数を増やすと，
    - 周波数そのものは上がる・・・が，
    - 場合によって，命令を処理できる実効的な速度が落ちる
    - 特に分岐予測ミス・ペナルティによる性能低下が大きい

余談：実際の CPU のパイプライン段数

- 現在は大体１５～２０段
- Intel Pentium4 （Prescott） ３１段
  - 2004年発売で 3.8 GHz
  - おそらく，歴史上最大の段数
    - 熱くなりすぎ & 性能が出ずで，この後ステージ数は減少
- AMD Zen： １９段
  - 2017年発売で 4.2GHz

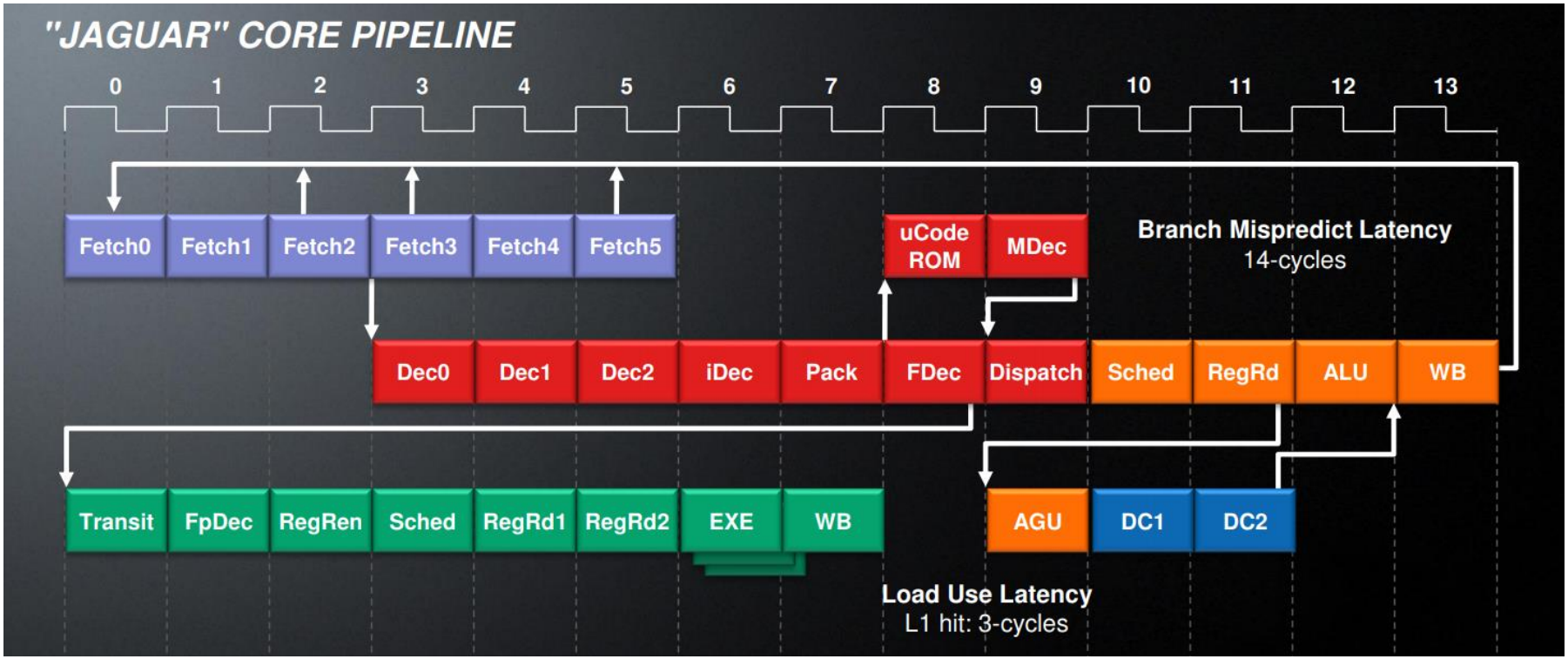
Sony/IBM/東芝 Cell (SPE)

Cell Broadband Engine Architecture and its first implementation—A performance view より



AMD JAGUAR

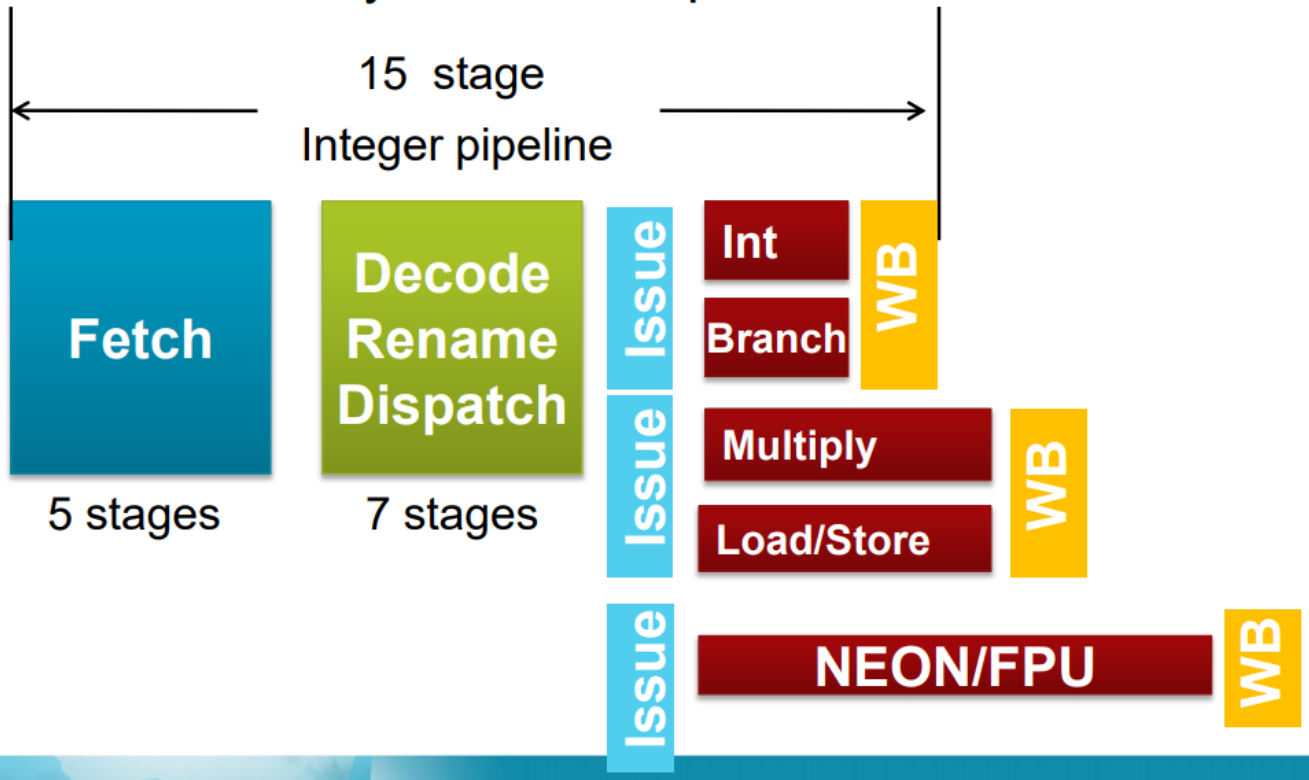
"JAGUAR" AMD's Next Generation Low Power x86 Core より



Cortex-A15 Pipeline Overview

15-Stage Integer Pipeline

- 4 extra cycles for multiply, load/store
- 2-10 extra cycles for complex media instructions



まとめ

1. 構造ハザード：ハード資源の不足に起因
    1. 構造ハザードとはなにか？
    2. その解決方法
2. 非構造ハザード：バックエッジに由来
    a. データ・ハザード
    b. 制御ハザード

課題 6.1

- (1) 0xf2a1 を２進数で表記せよ
- (2) 0111 1000 1111 0000 を１６進数で表記せよ
- RISC-V の「add x1←x1,x1」命令を２進数で表記すると以下の通りとなる
  0000000 00001 00001 000 00001 0110011
  - (3) 上記を sub x1←x1,x1 に書き換え，２進数と１６進数の双方で表記せよ
  - (4) 上記を add x2←x3,x4 に書き換え，２進数と１６進数の双方で表記せよ
- 第３回目の講義および次のページ仕様を参考にすると良い

103

RISC-V の 基本整数命令

- 概要
  - 加減算，論理演算，
    ロード・ストア，
    即値，分岐とジャンプなど
  - 各命令は 32bit 幅
- 前半の講義で４桁の数字で
  表していたことと同じことを
  32bit の中を細かく区切って
  やっている

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000	pred | succ	00000	000	00000	0001111	FENCE
0000	0000 | 0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

画像は下記より
The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2

課題 6.2

- 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:    if (i - 1 < 10) {
5:       i = i + 1;
6:    }
7: }
```

提出方法

- 以下を提出：
  1. 課題 6.1 と 6.2：
     - 提出は Moodle の「課題６」のところからお願いします
  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（6/11 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
  - 課題は成績の判定にかなり使われます
    - 仮に課題を一度も出さなかった場合，
      期末試験だけ受けてもまず通らないと思います