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

課題の解説

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

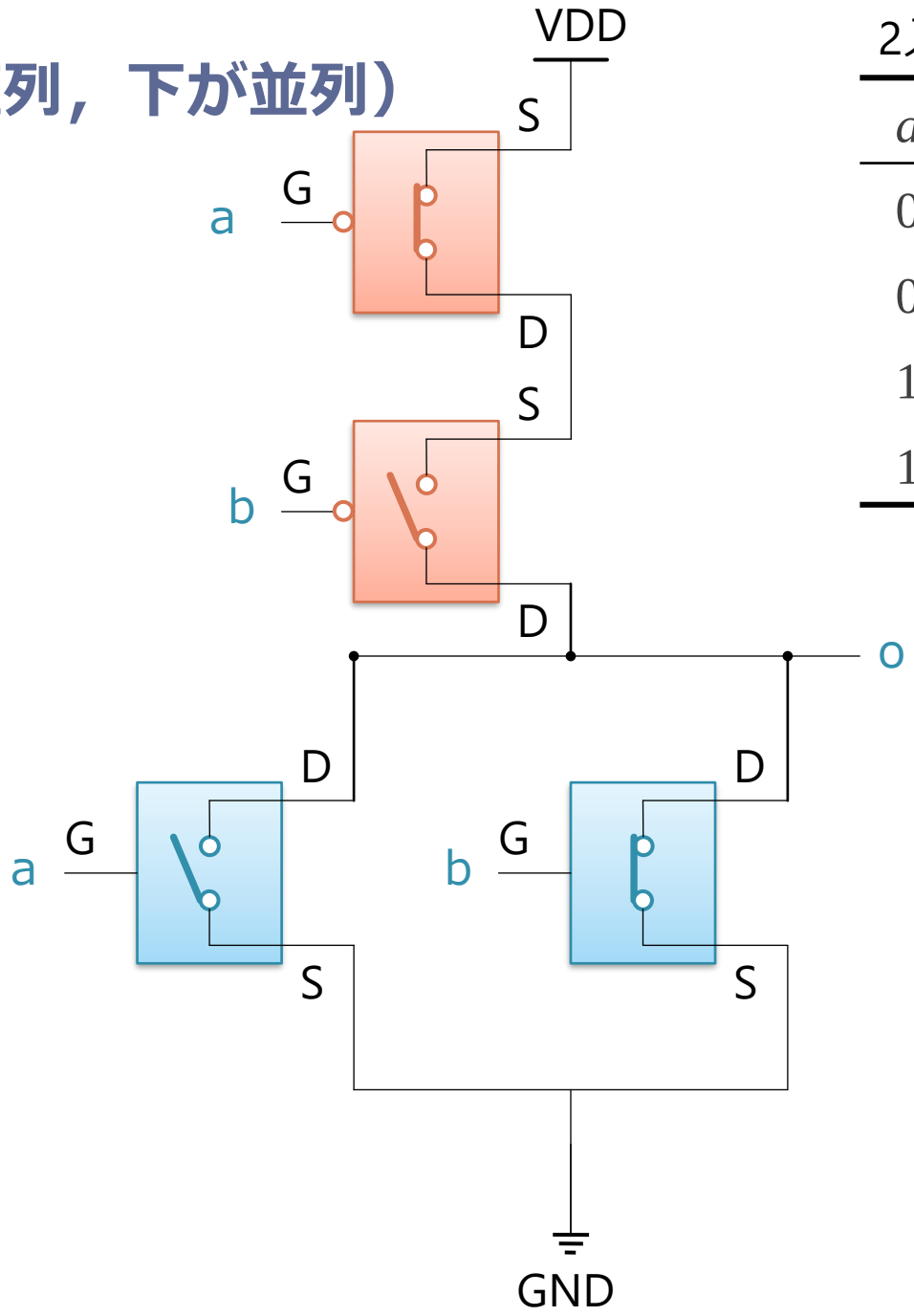
3入力 NOR の真理値表

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

OR の作り方

- NOR を作って NOT で反転する
  - NAND を作って NOT で反転して AND を作ってたのと同じ

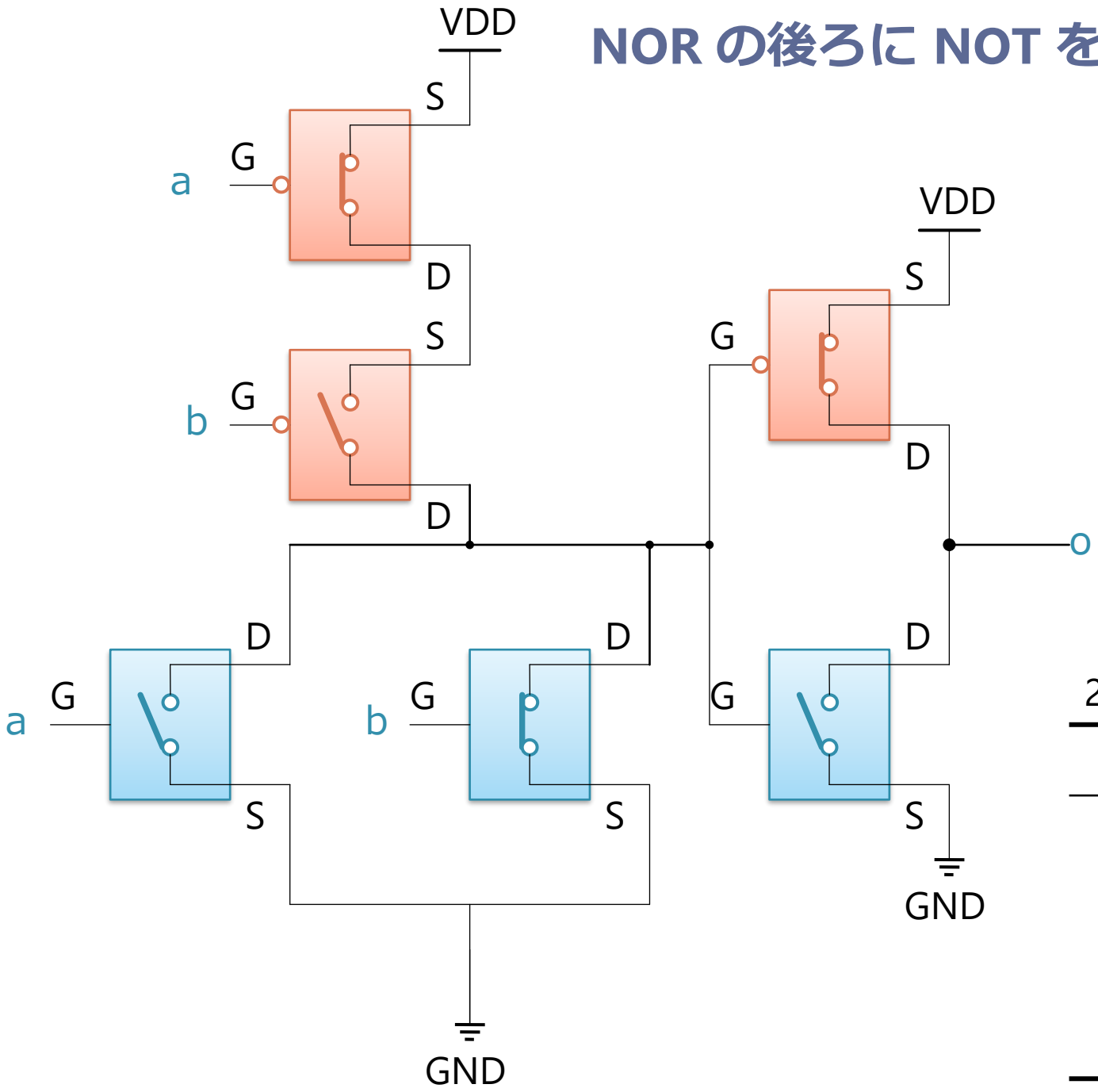
NOR
（上が直列，下が並列）



2入力 NOR の真理値表

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

NOR の後ろに NOT を接続



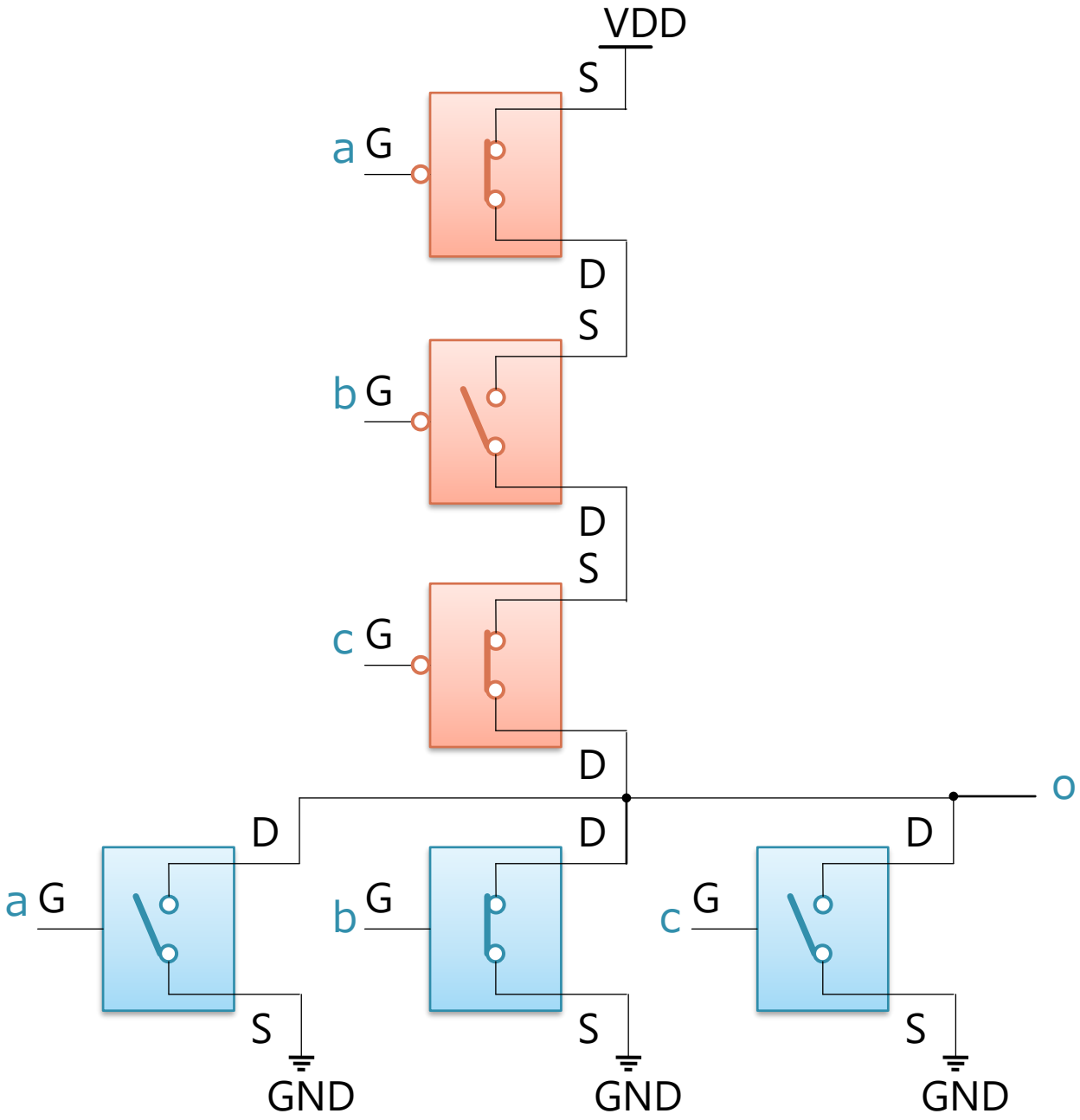
2入力 OR の真理値表

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

3入力NOR

3入力 NOR の真理値表

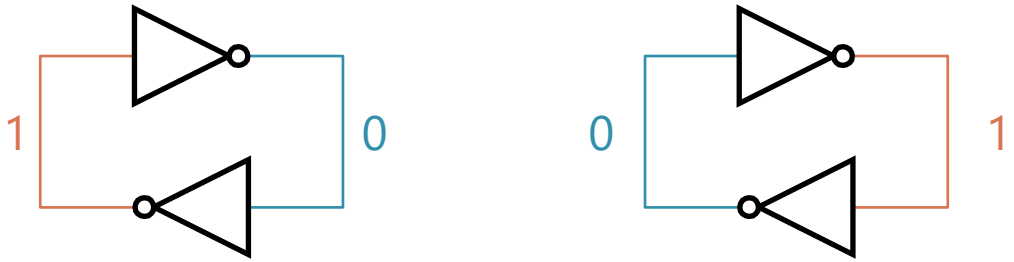
a	b	c	o
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



前回の振り返り

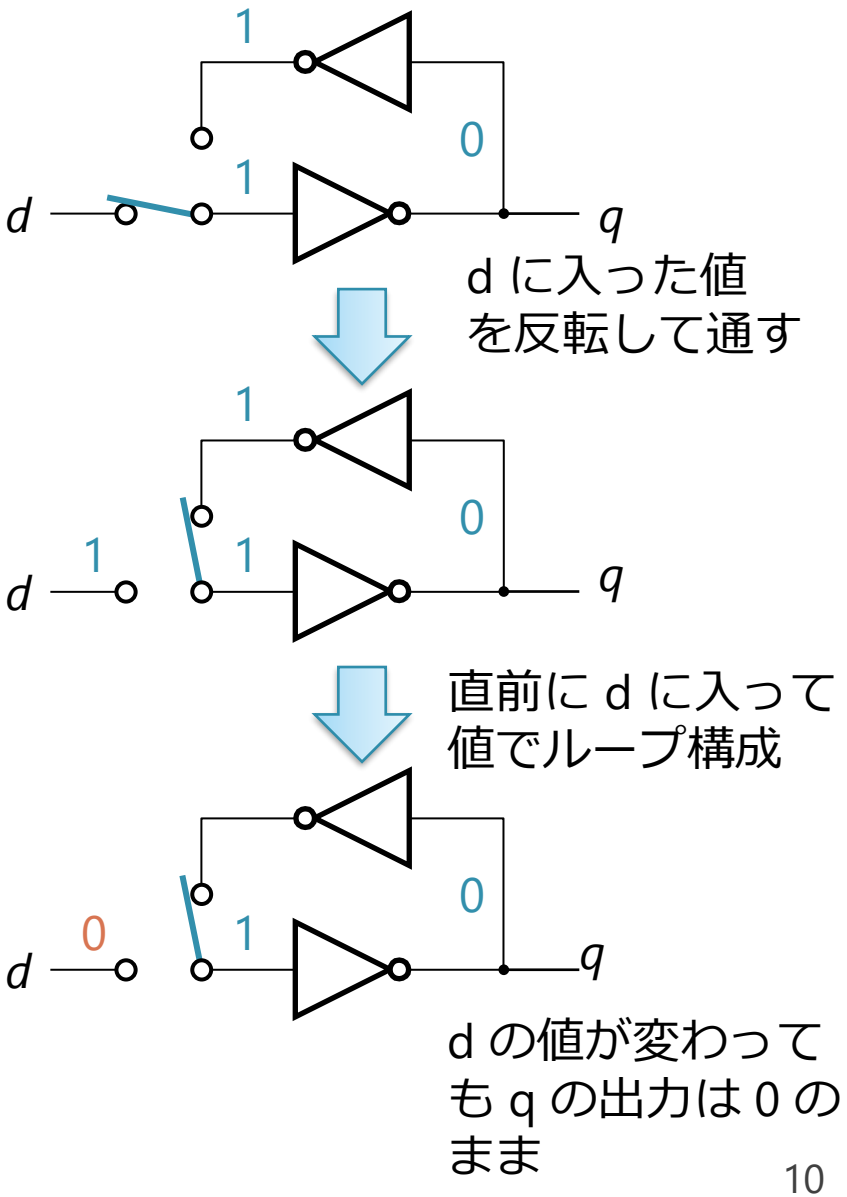
記憶素子の原理

- 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



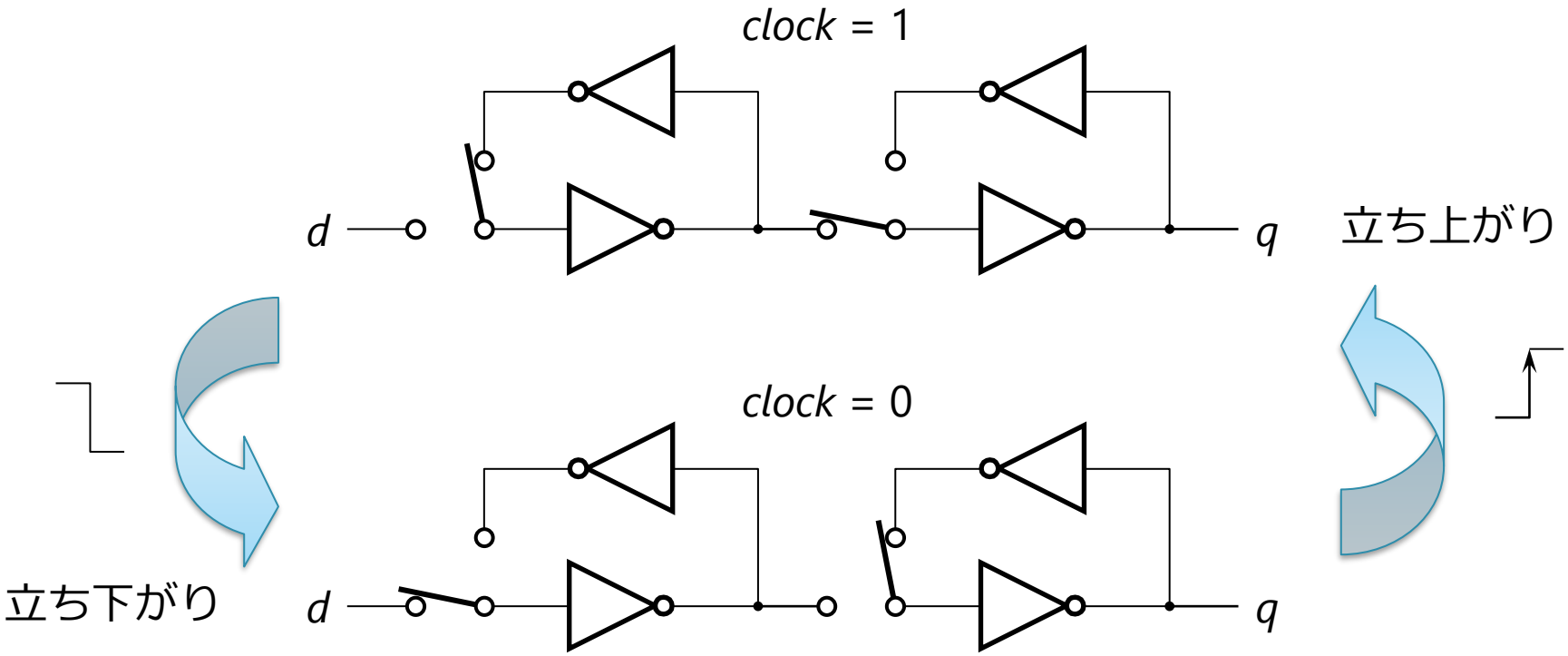
D ラッチの回路

- 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

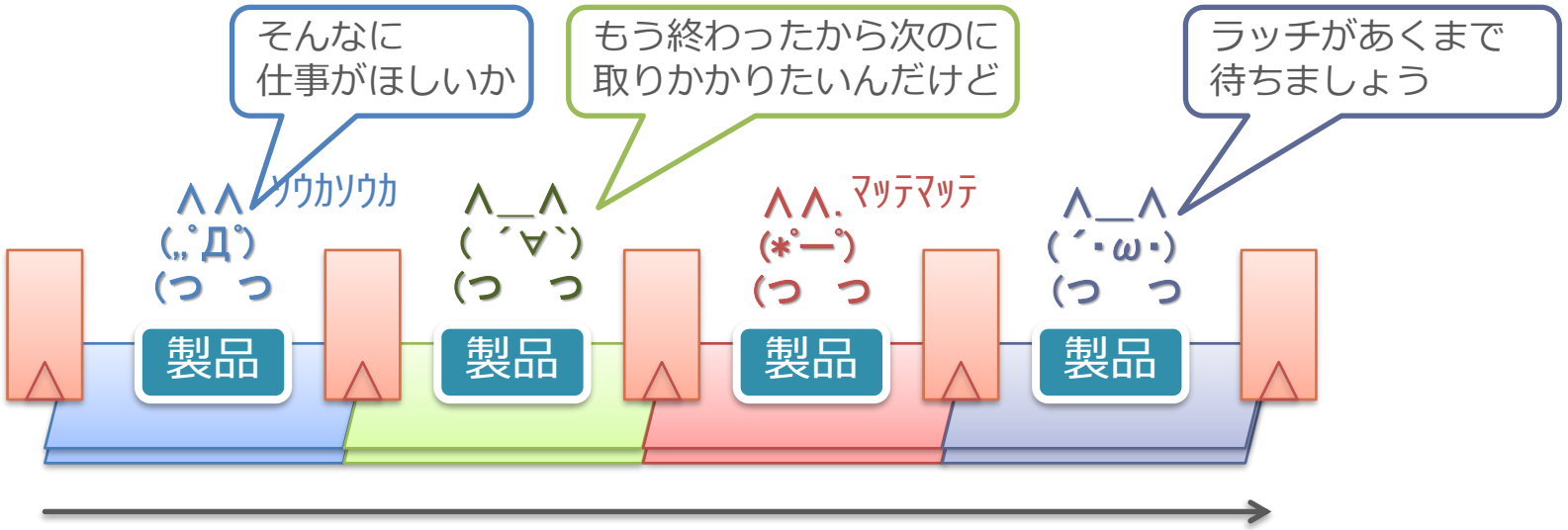
- 構造：D ラッチを2つ繋げたもの
  - ◇ D ラッチ１つだと，半周期は d に入った値が反転して素通しするので使いにくい
  - ◇ ２つ直列に繋げて素通しの期間をなくす



工場のラインを考える（再）

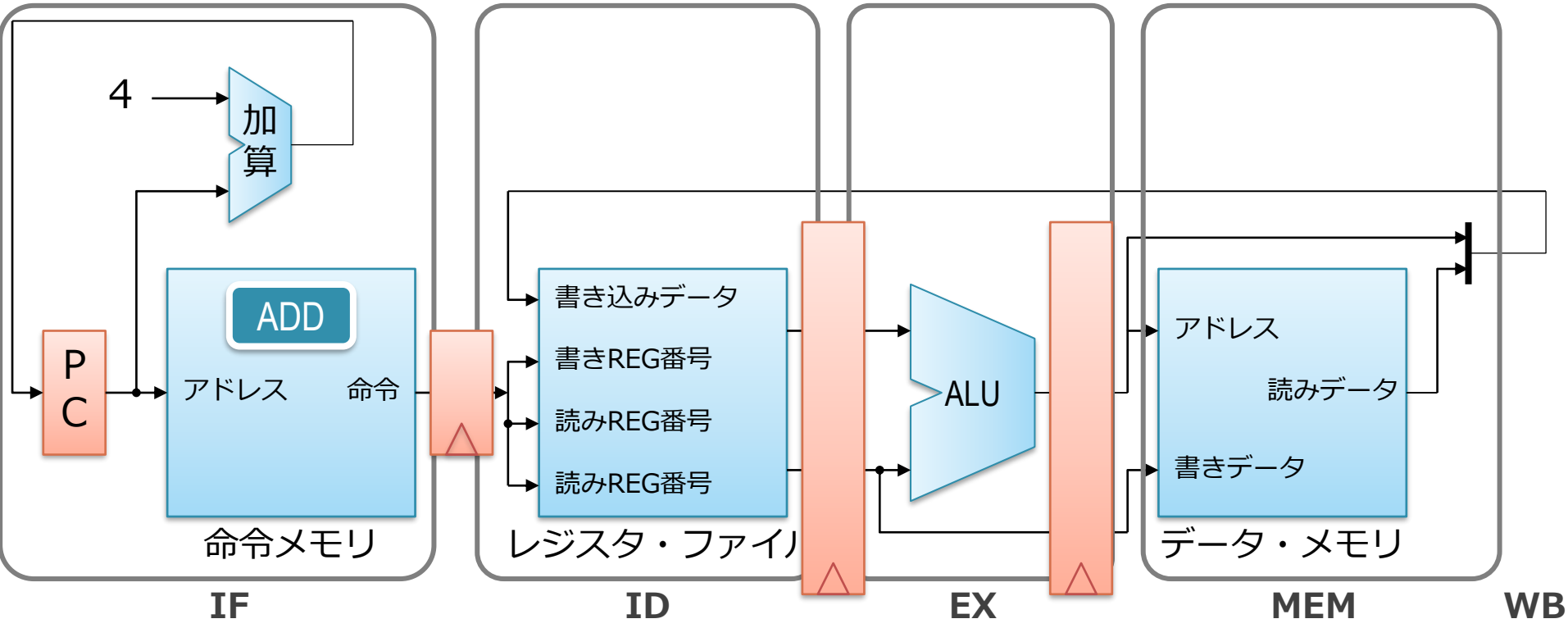
- 実際の工場：複数の製品を同時に流す
  - 各工程を並列して処理することによりスループットを向上
- これが 命令パイプライン

パイプライン・ラッチのイメージ



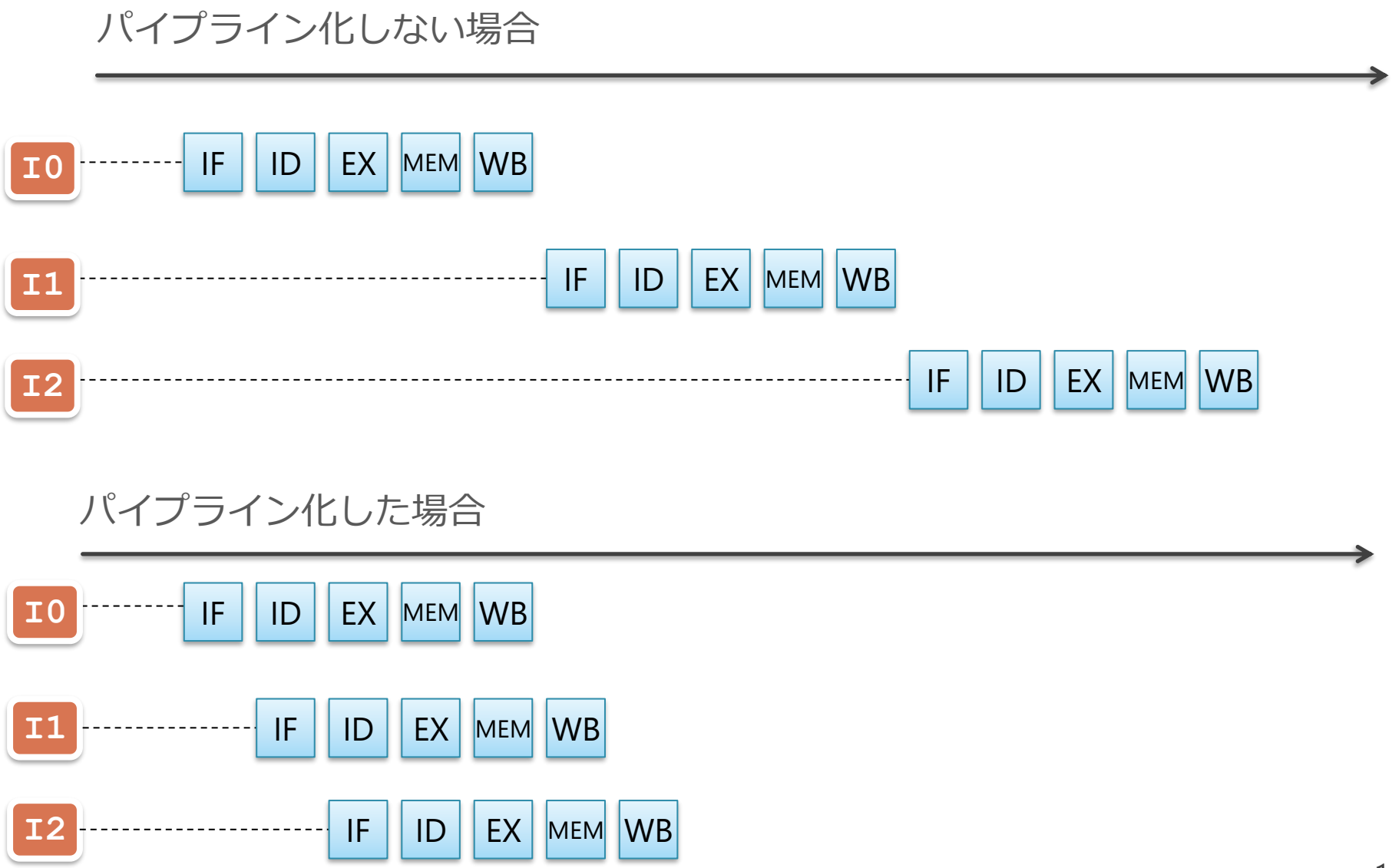
- 各人の作業が終わっても，ラッチが開くまでは次の人に製品を送れない
  - 複数ステージ間で信号が混じるのを防ぐ
  - 指定された時間までラッチでドアを開かなくするイメージ？

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：
    「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

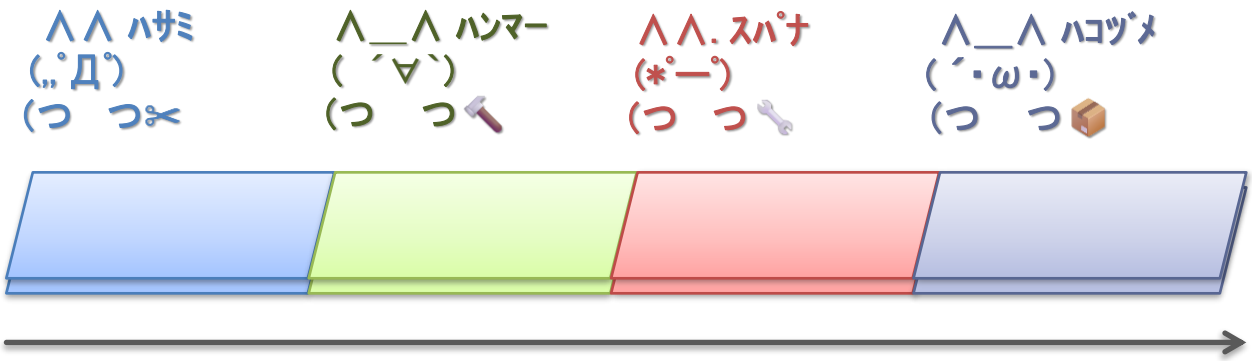
もくじ

構造ハザード

構造ハザード

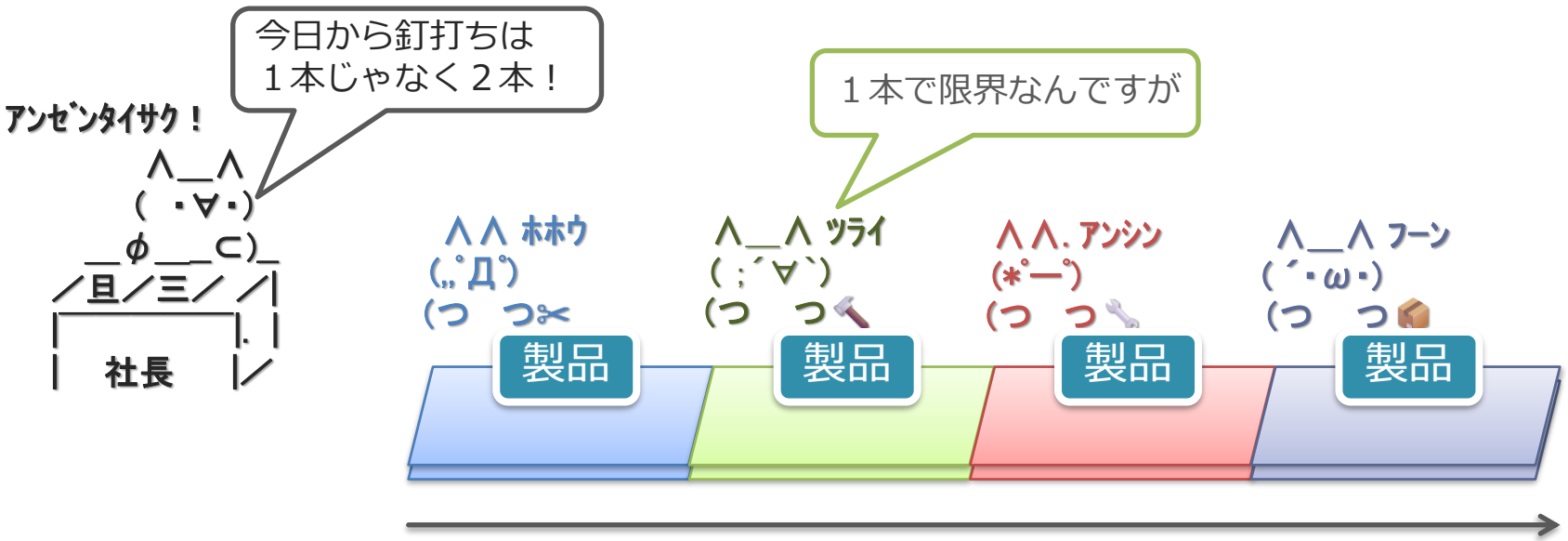
- ハード資源の不足により，パイプラインがうまく動作しないこと
- 工場のラインの例を使って説明

工場のライン



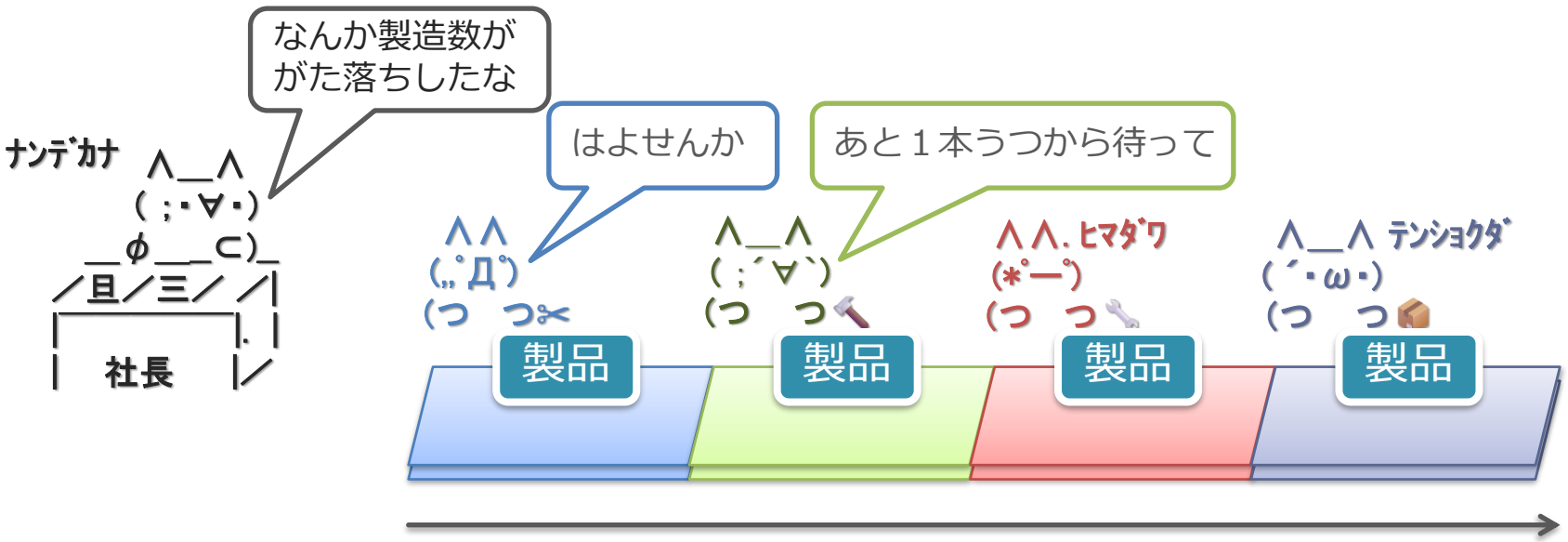
- 各ステージの担当者は製品が流れるまでの間に一定の仕事ができる
  - はさみで紙を１枚切る
  - ハンマーで釘を１本打つ
  - スパナでねじを１個しめる
  - 箱に製品を１つ入れる

釘の本数の変更



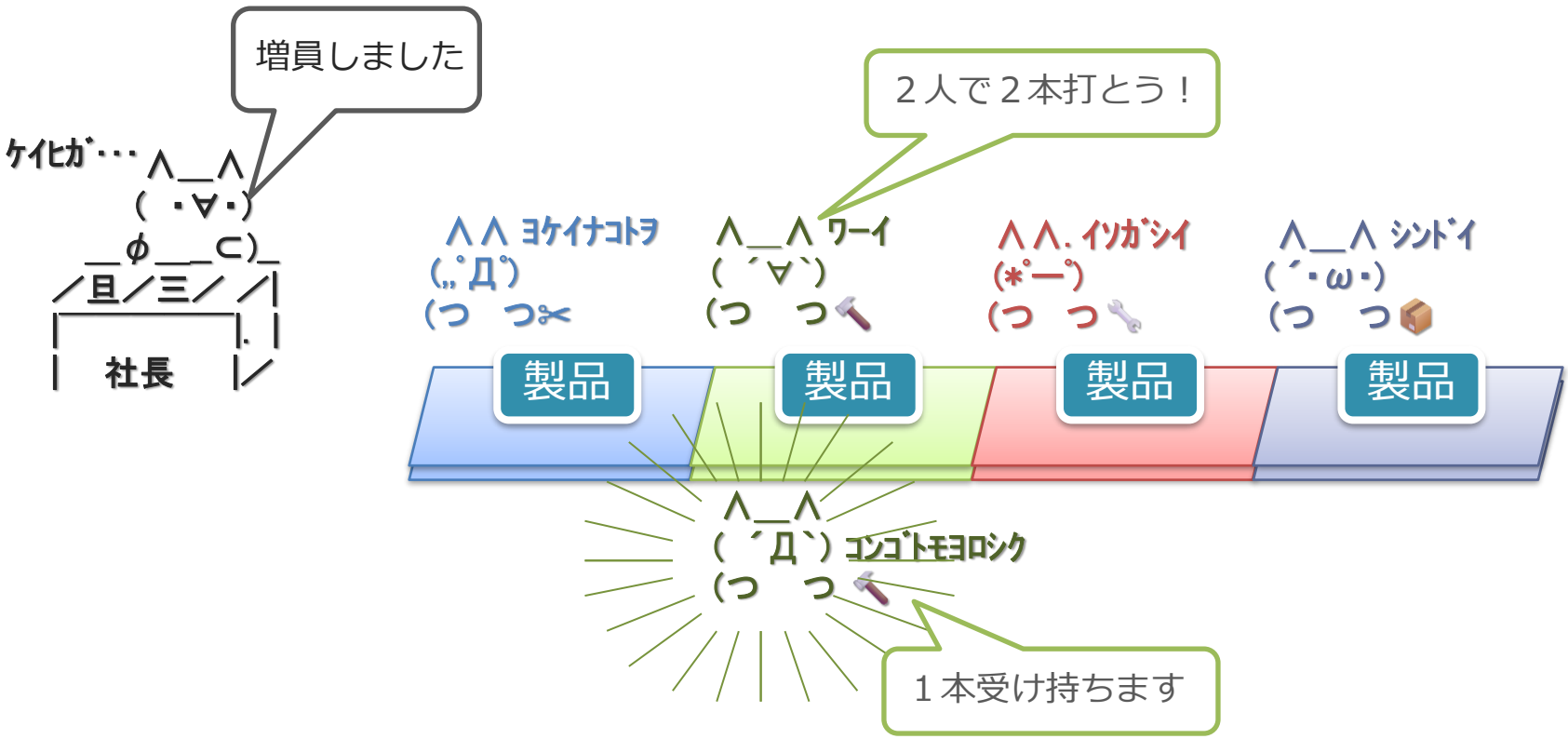
- 製品の安全対策のために釘の本数が１本から２本に
  - しかし，ハンマーを持っている釘打ち担当は単位時間に１本しか打てない

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

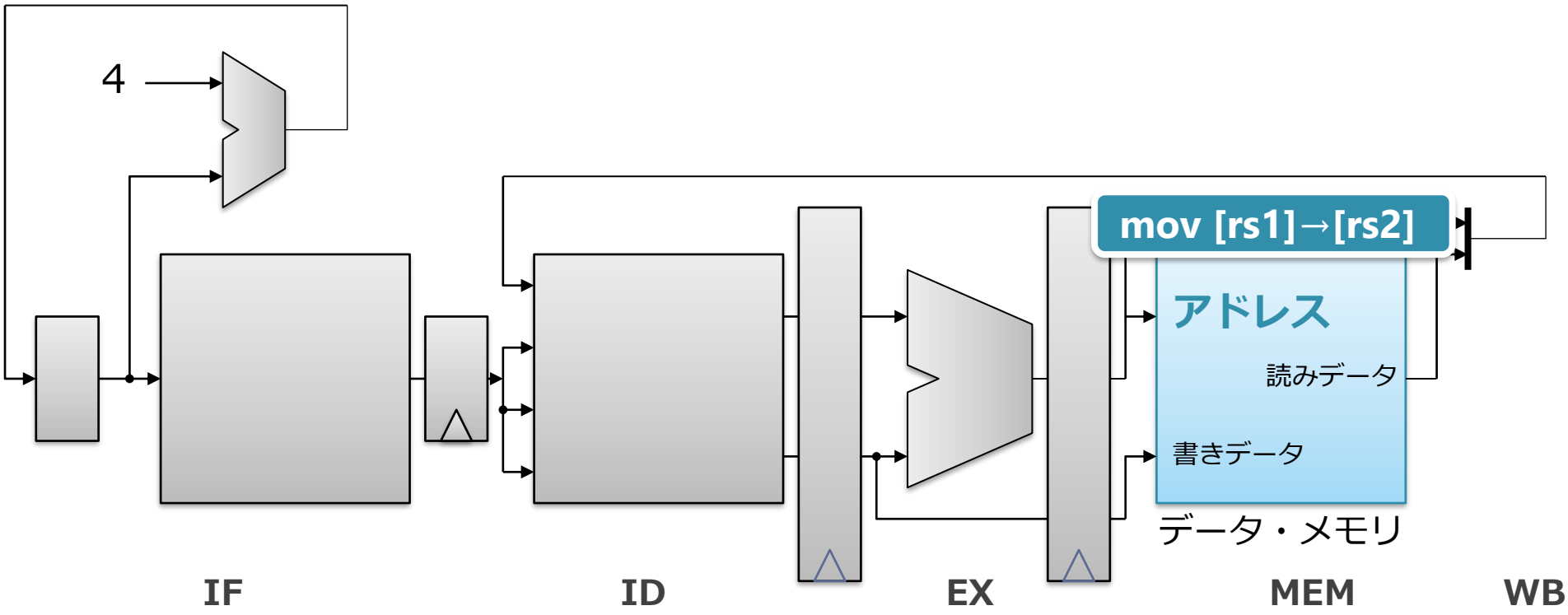
コンピュータにおける構造ハザード

- 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

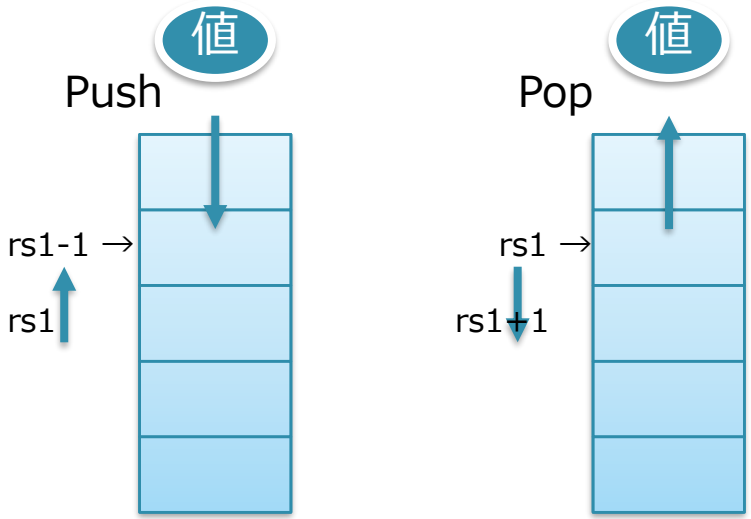
mov [rs1]→[rs2] // [rs1]→[rs2] へのコピー



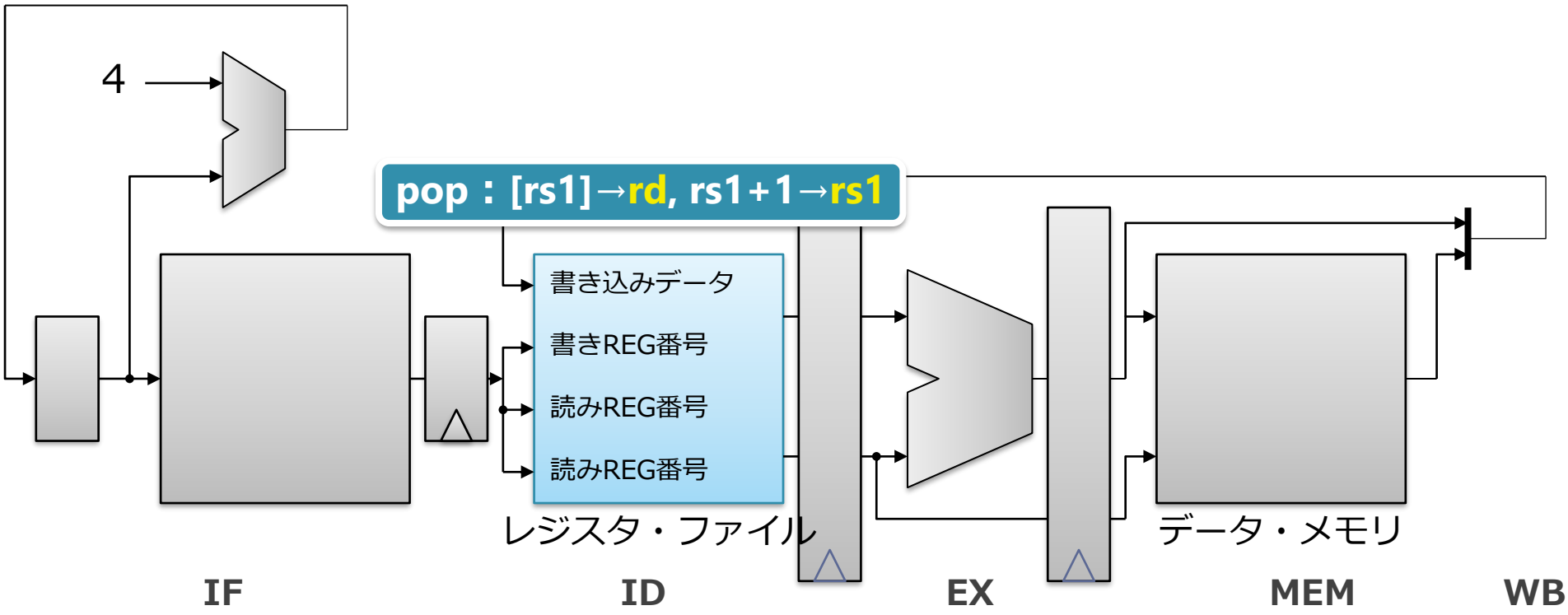
- ある１つのサイクルにメモリを「読んで」「 書く」必要がある
  - しかし，データ・メモリのアドレスの口は1つしかない
  - MEM ステージでデータ・メモリの読みと書きが同時にできない

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



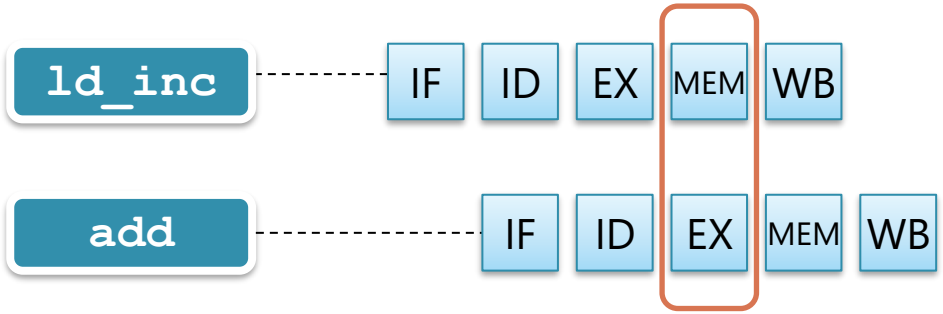
- WB ステージでレジスタに rd と rs1 の2つを書き込む必要がある
  - 2 つのレジスタが 1 つの命令により更新されている
  - レジスタ・ファイルへの書き込みは，同時に 2 つはできない

構造ハザードの例３

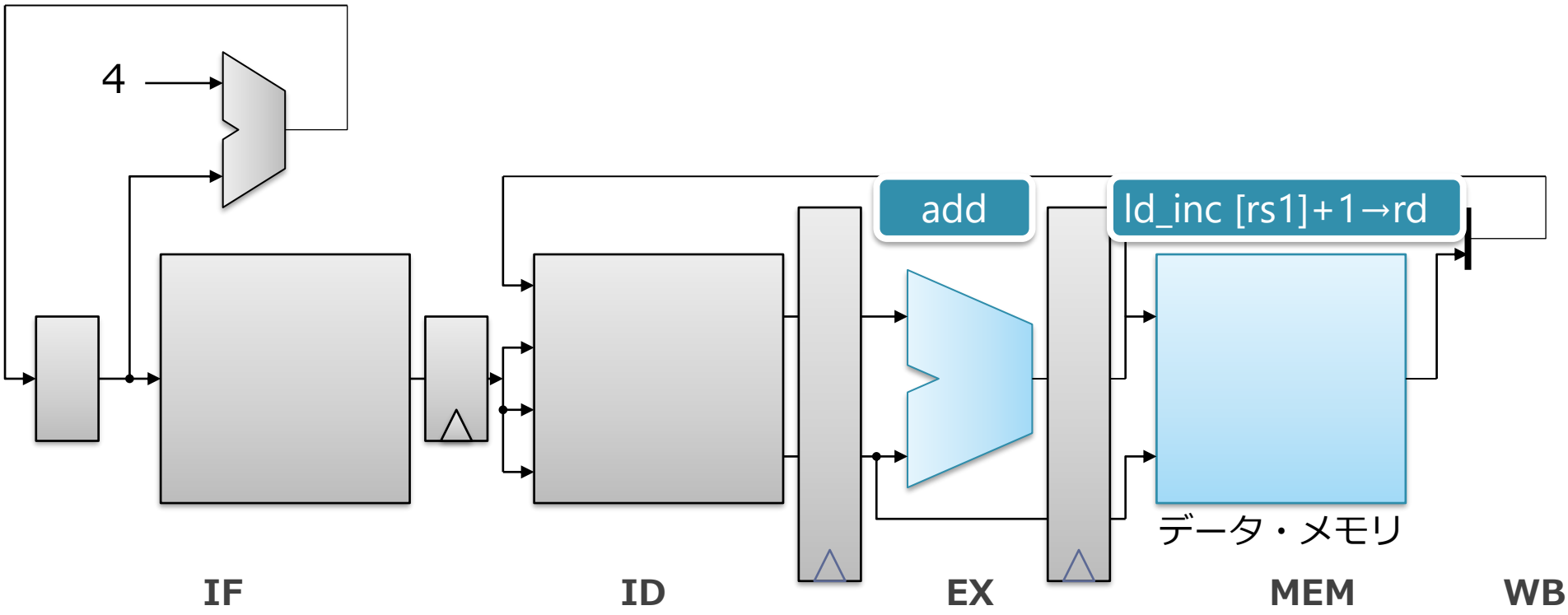
- 使用資源の異なるステージ間のぶつかりでも起きる
  - これまでの例は，同じステージ内で資源が足りない例
- **`ld_inc [rs1]+1→rd`** のような命令があったとする
  1. rs1 の指すアドレスからメモリを読む
  2. 読んだ値にさらに+１してから rd に書く
- 一見，資源は足りているようだが･･･
  - レジスタの読み書きは１つずつしかない
  - メモリも１カ所を読むだけ
  - 加算も１回行うだけ

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

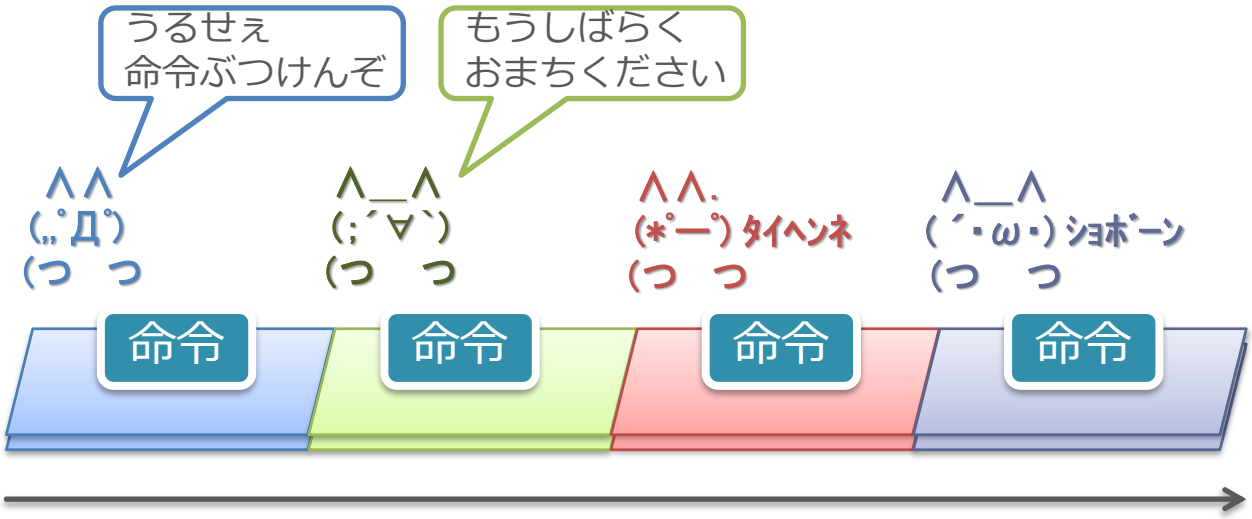
構造ハザードの解決方法

- 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

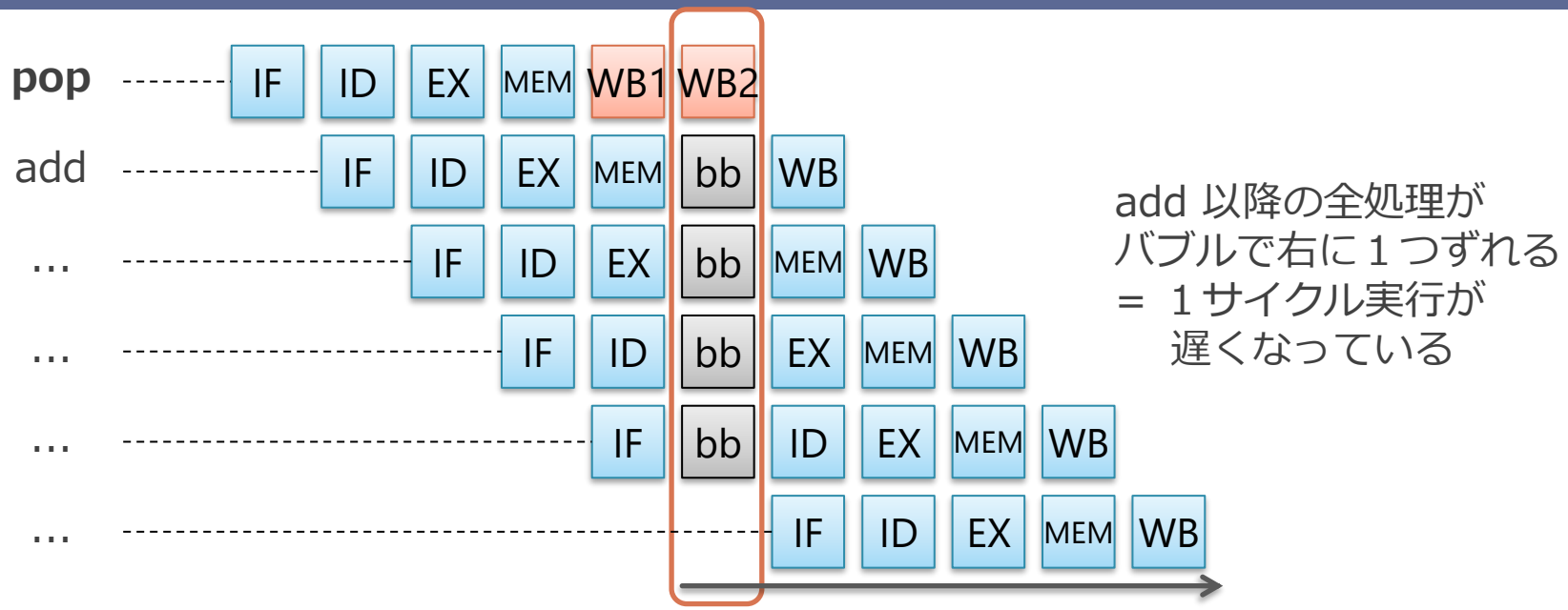


- 上流を止めないと破綻する
  - (;´∀`) が複数サイクルをかけて仕事をしている場合，命令はそこにとどまり続ける
  - その間は上流をとめないと命令をおく場所がないし，依存関係がまもられない
- (*ﾟ―ﾟ) より下流は流れていっても，この場合は問題ない

パイプラインを止めること

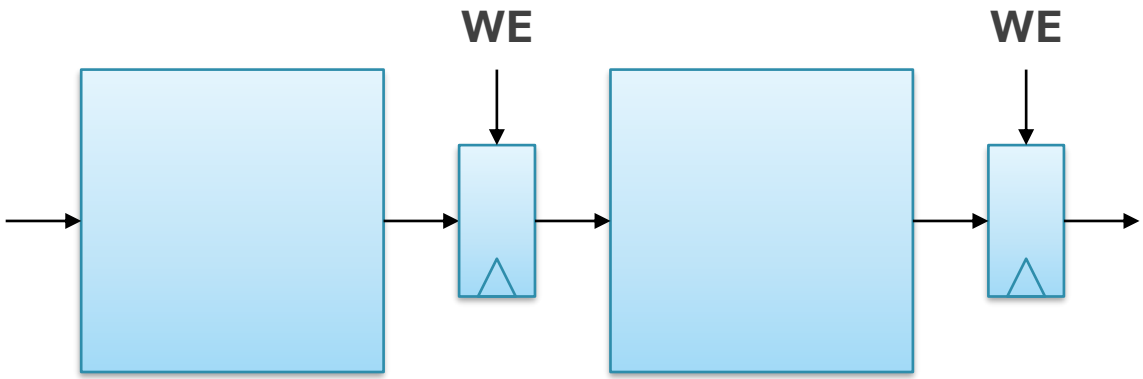
- パイプラインを止めるとことを「ストール」や「インターロック」という
  - 本や人によって，意味や使い方が微妙に統一されていない
  - この講義では，以降はストールで統一

ストールの動作



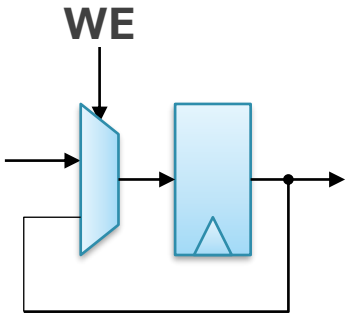
- pop： 1 つレジスタに書いたあと，次のサイクルで書き込む
  - WB1 と WB2 の 2 サイクルで書き込む
  - WB2 の間は上流を全て止める
- パイプライン・チャート上では上記のようになる
  - 止める原因の命令の下が全部右にずれる
  - ずれた部分の空き（bb）を「バブル」とよぶ

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

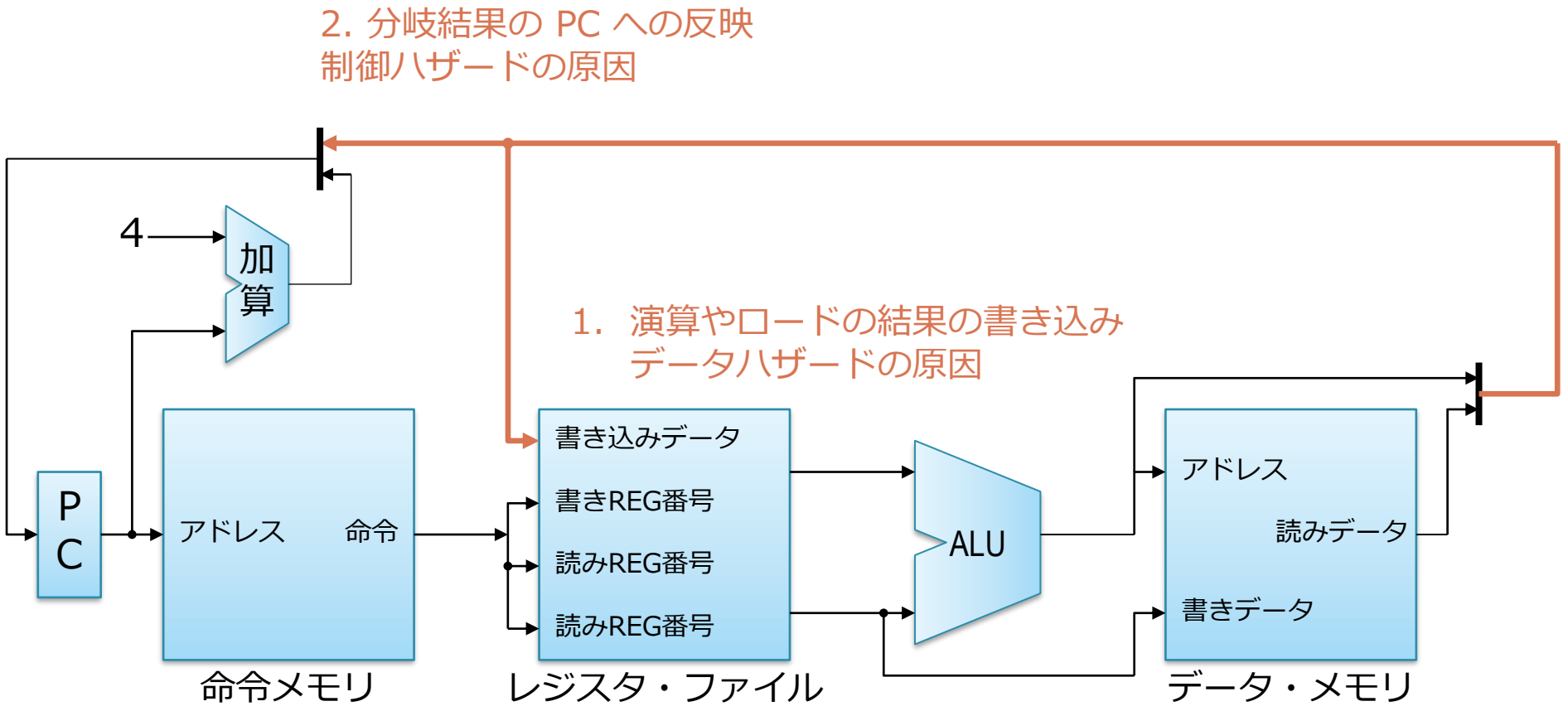


- たとえば D-FF とマルチプレクサで作れる
  - WE が 1 の時は，左からきた入力を選んで書き込む
  - WE が 0 の時は，その時の自分自身の出力を選んで書き込む

非構造ハザード

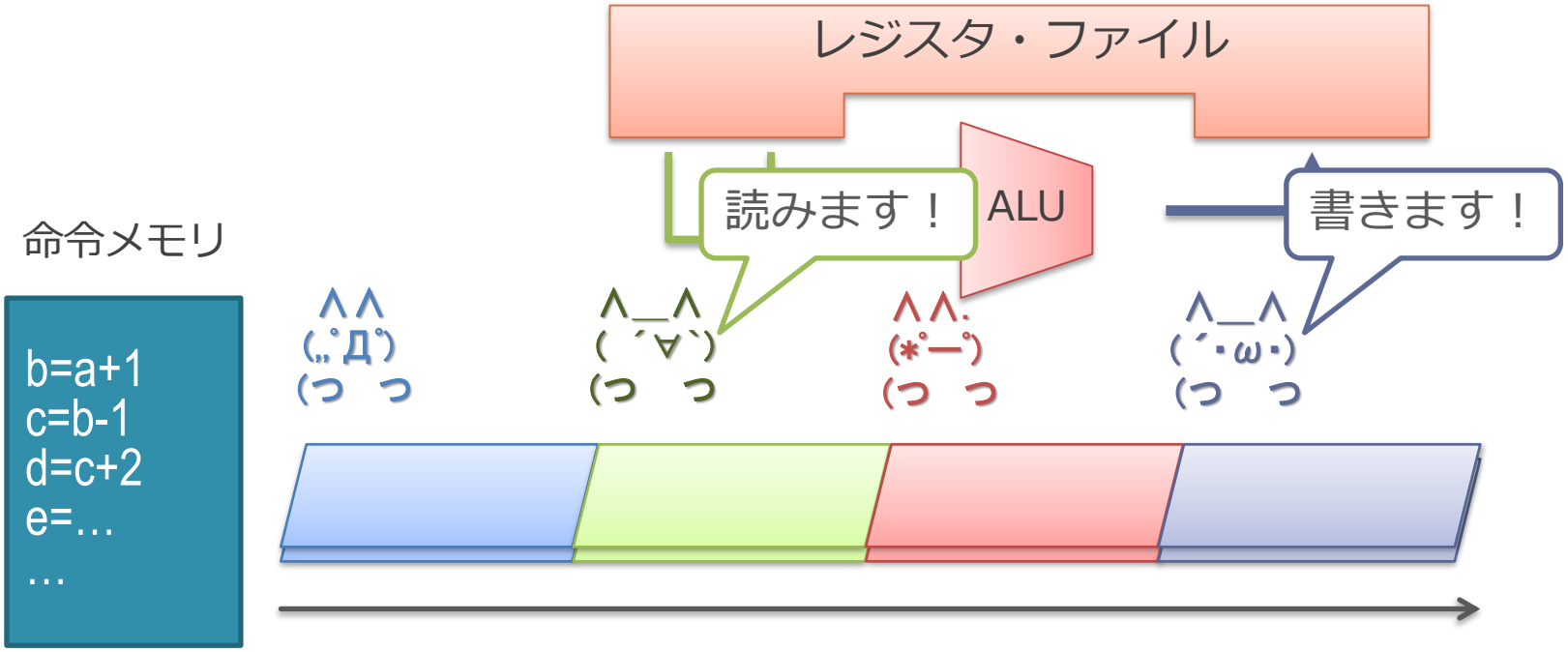
もくじ

バックエッジとは：逆方向（右から左）にいく信号



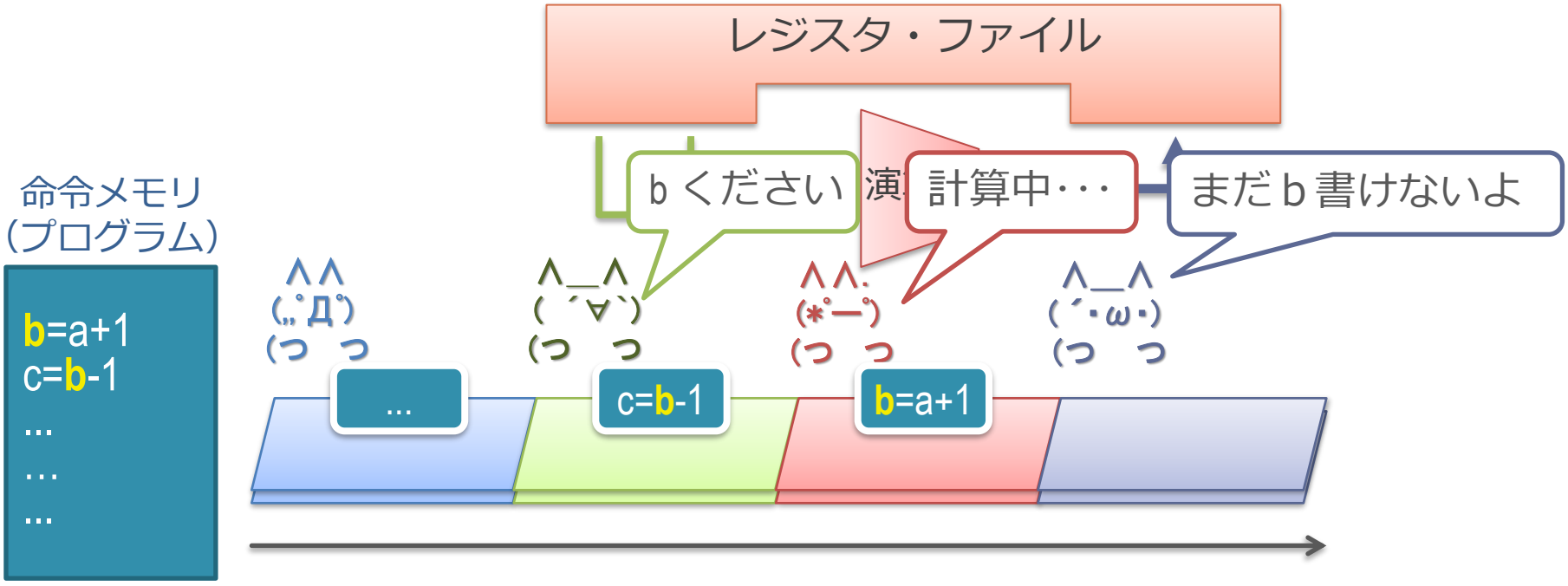
- バックエッジがあるため，命令を単純に流せない場合がある
  - 工場のラインのように，一方向に流せない

データ・ハザード



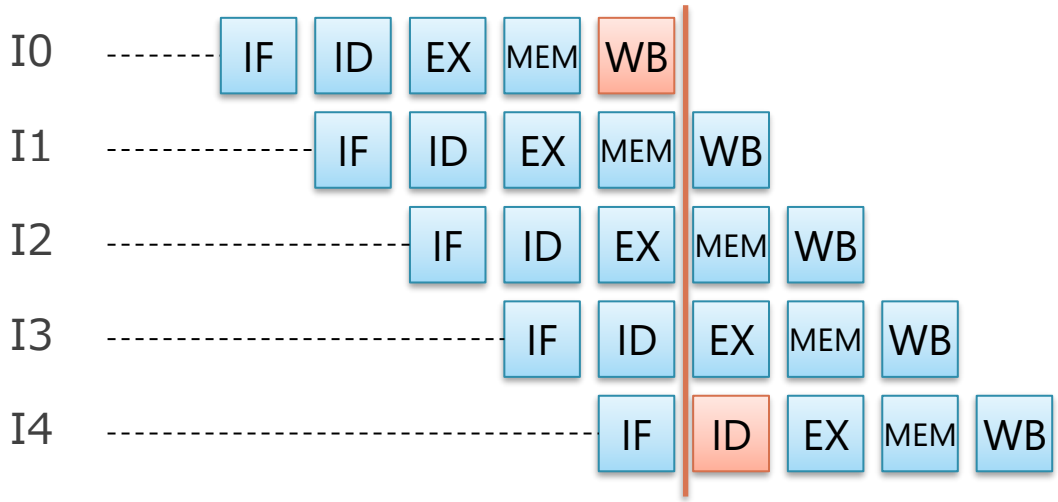
- レジスタ・ファイルへのアクセス
  - 演算の入力は(´∀`) の人がレジスタ・ファイルから読み出す
  - 演算の結果は(´･ω･) の人がレジスタ・ファイルに書き込む

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

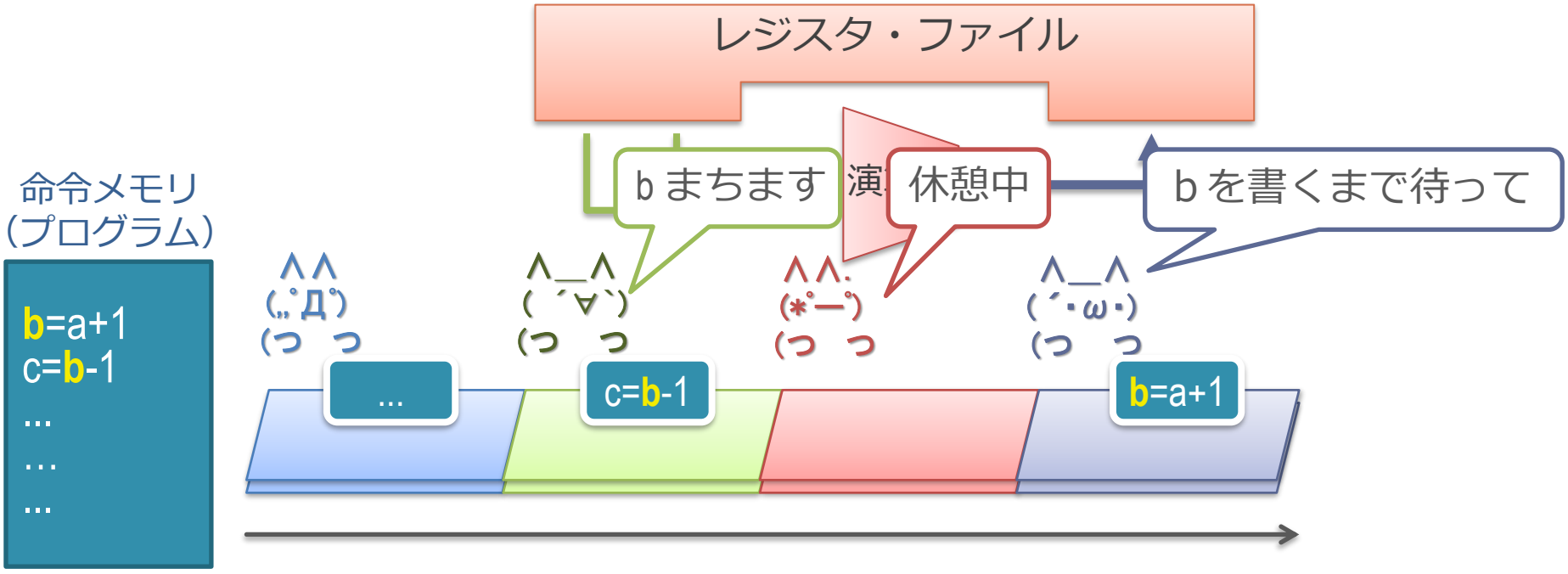


- I0 の WB が終わるまで，その結果はレジスタに書き込まれない
  - I4 までは，その値がレジスタから得られない
  - ID ステージでレジスタを読むため

データ・ハザードの解消方法

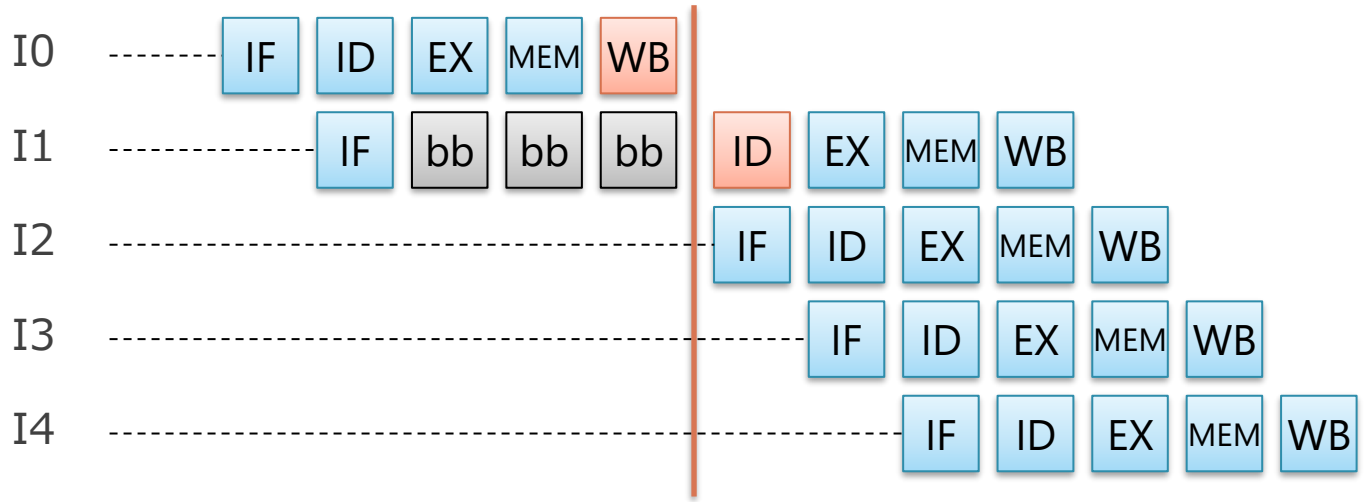
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. フォワーディング

1．ストール



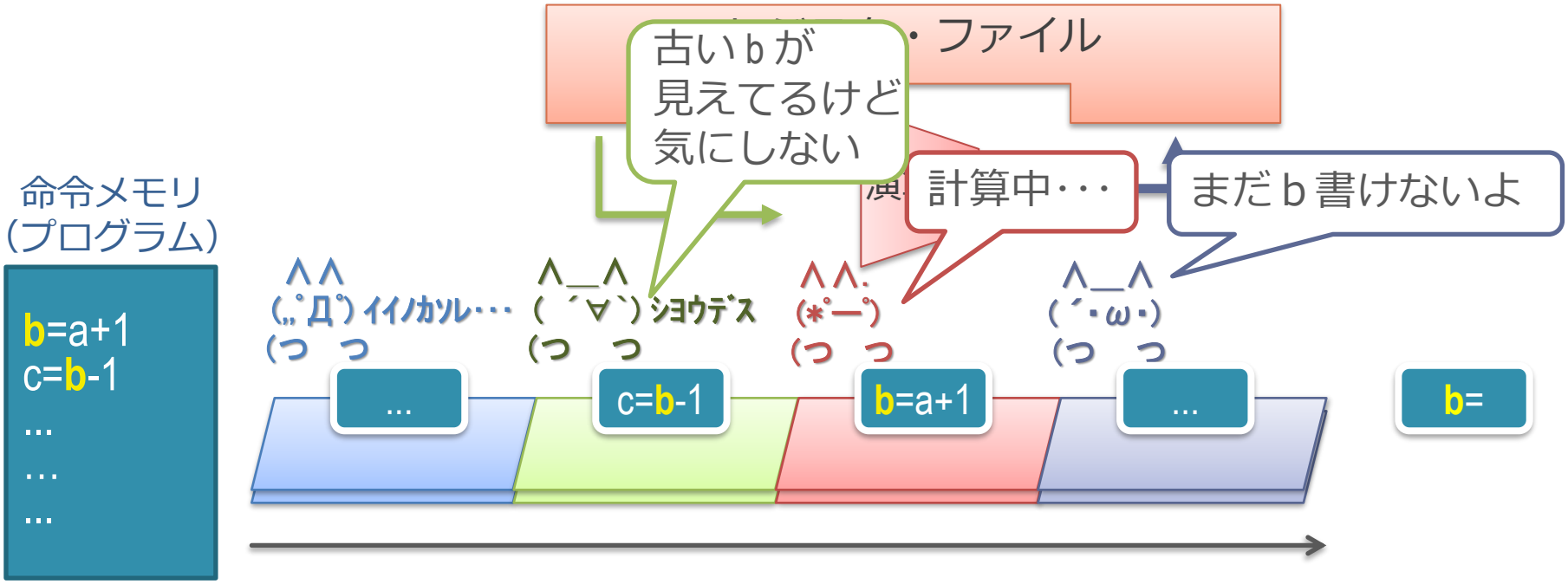
- 直前の命令の結果を使う命令が現れた場合：
  - (´･ω･) の人が計算結果をレジスタに書くまで，
    (´∀`) の人と上流のラインを止める
  - 間にバブル（何もしないステージ）が入る

1．ストールさせる



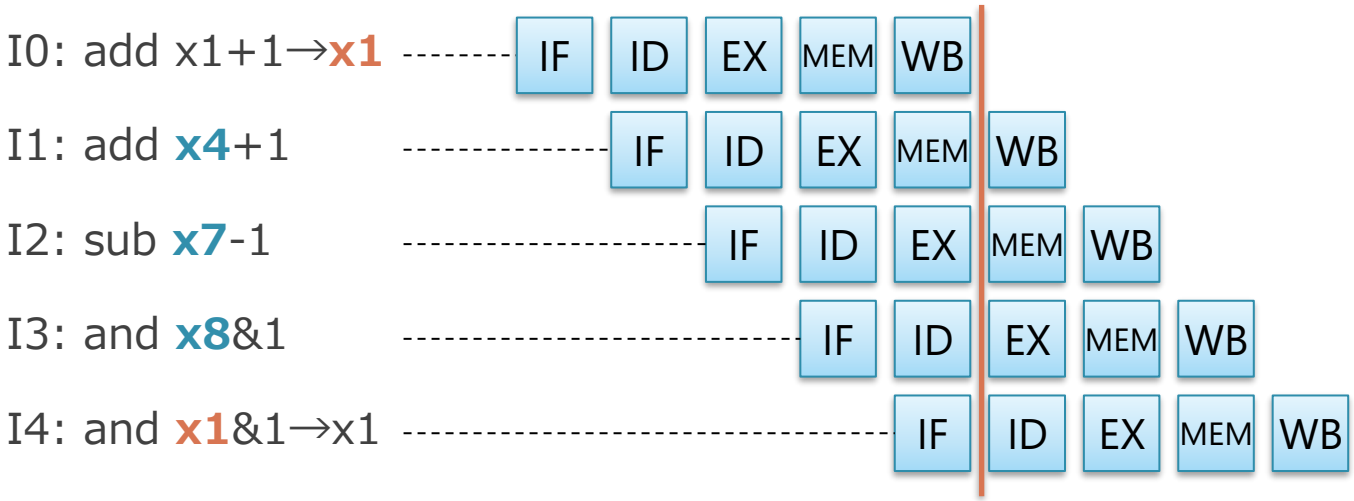
- I0 の WB が終わるまで，後続の命令を遅らせる
  - I1 の ID が，I0 の WB の右にくるまでストール
  - I1 は I0 の結果を使える
- 欠点：プログラムの実行がとても遅くなる

2. 遅延スロット（なにもしない）



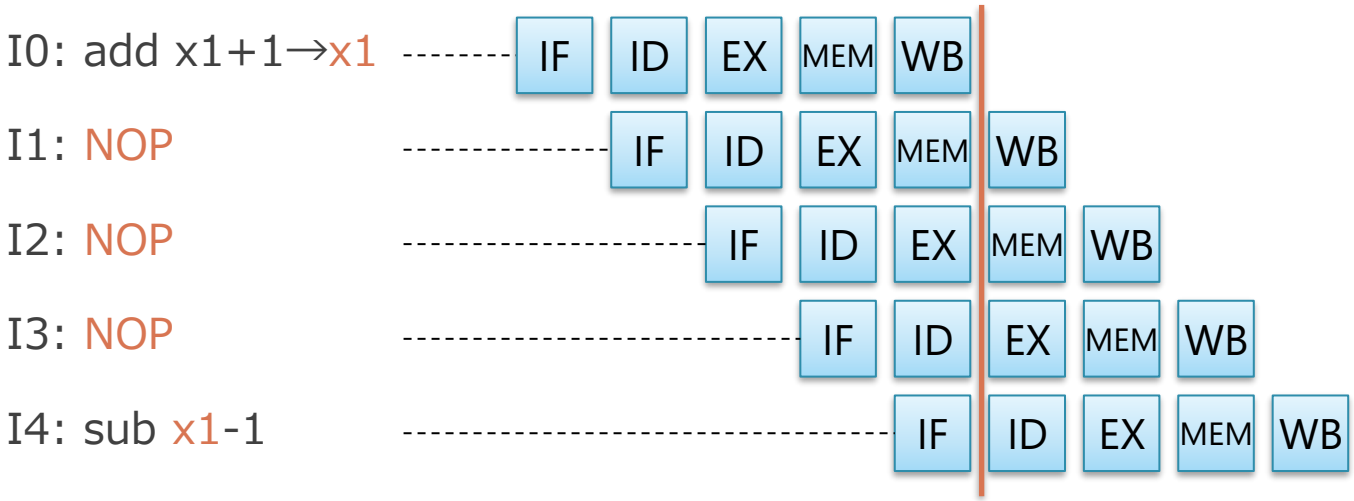
- 直前の命令の結果を使う命令が現れた場合：
  - （ ´∀`）の人は，その時レジスタ・ファイルにあるｂを気にせず読んでしまう
  - ２つ前の命令の結果のｂが読めてしまう
    - そう言う仕様ということにする

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



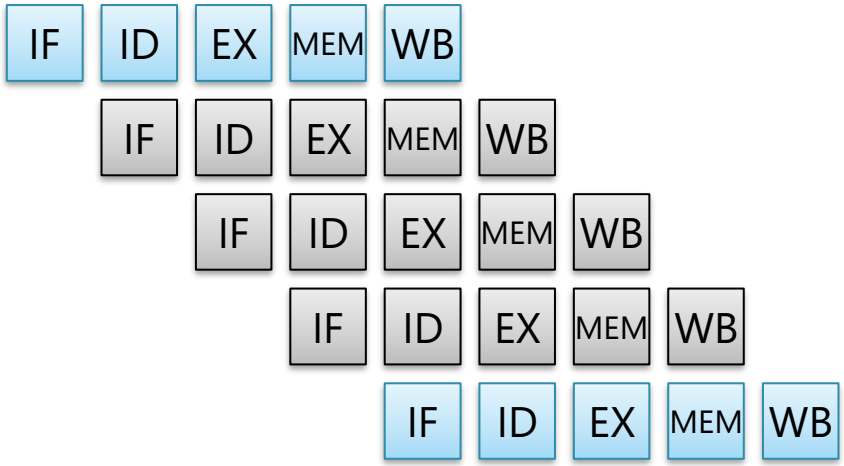
- もしそのような命令がない場合，
  - NOP（No Operation）と呼ぶ何もしない命令をいれる
  - これもコンパイル時にいれておくる必要がある
- 上の例は，x1 に 1 を足した結果を使う以外の処理がなかった場合

遅延スロットの利点

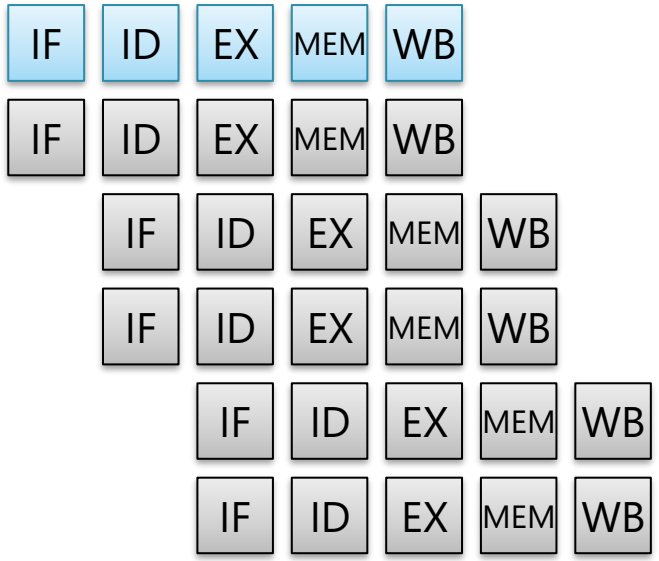
- 利点：
  - なにもしないので，ハードは最も単純
  - 並列にできる命令があれば，性能も下がらない

遅延スロットの欠点

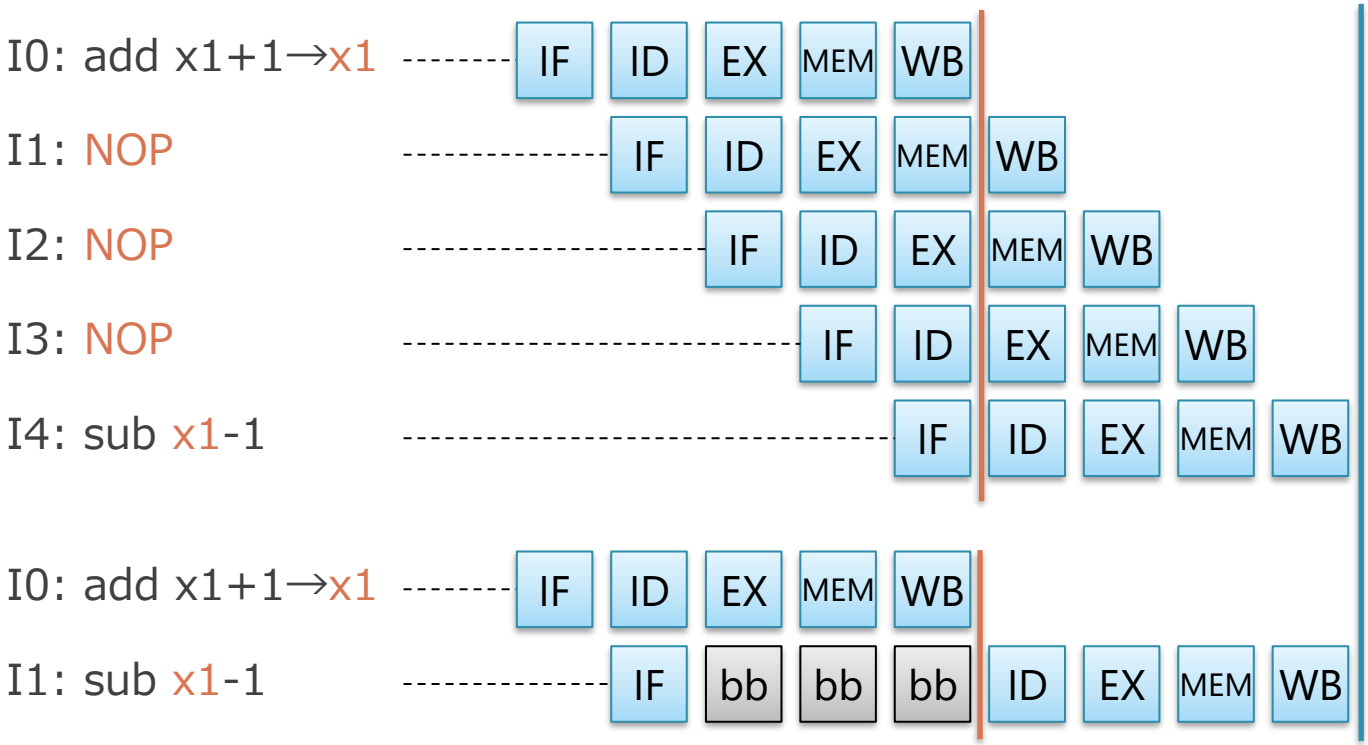
- 欠点：「仕様」なので，一度決めると変えられない
  - 後からパイプラインの段数や構造を変えると互換性がなくなる
    - クロックをあげるために，段数を増やせない
  - 複数の命令を同時処理しようとしたときにも互換性がなくなる
  - MIPS では遅延スロットが 1 命令分，仕様として存在
    - 互換性のためにこれを忠実に再現するため後年は逆に複雑化



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

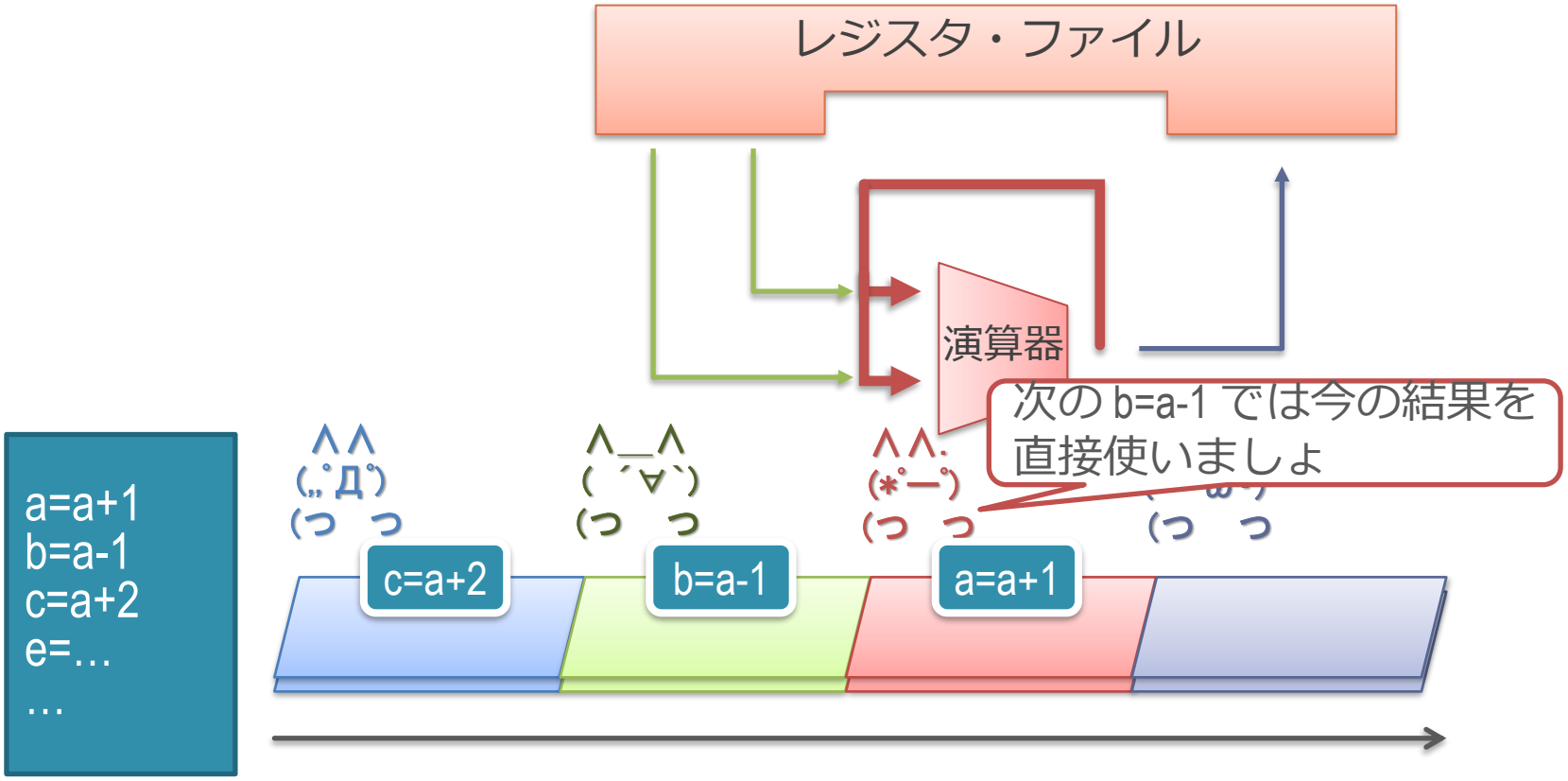


- 欠点２：並列してできる命令が常にあるとは限らない
  - NOP を入れるしかなくなる
  - 実質ストールしてバブルを入れるのと同じになってしまう

データ・ハザードの解消方法

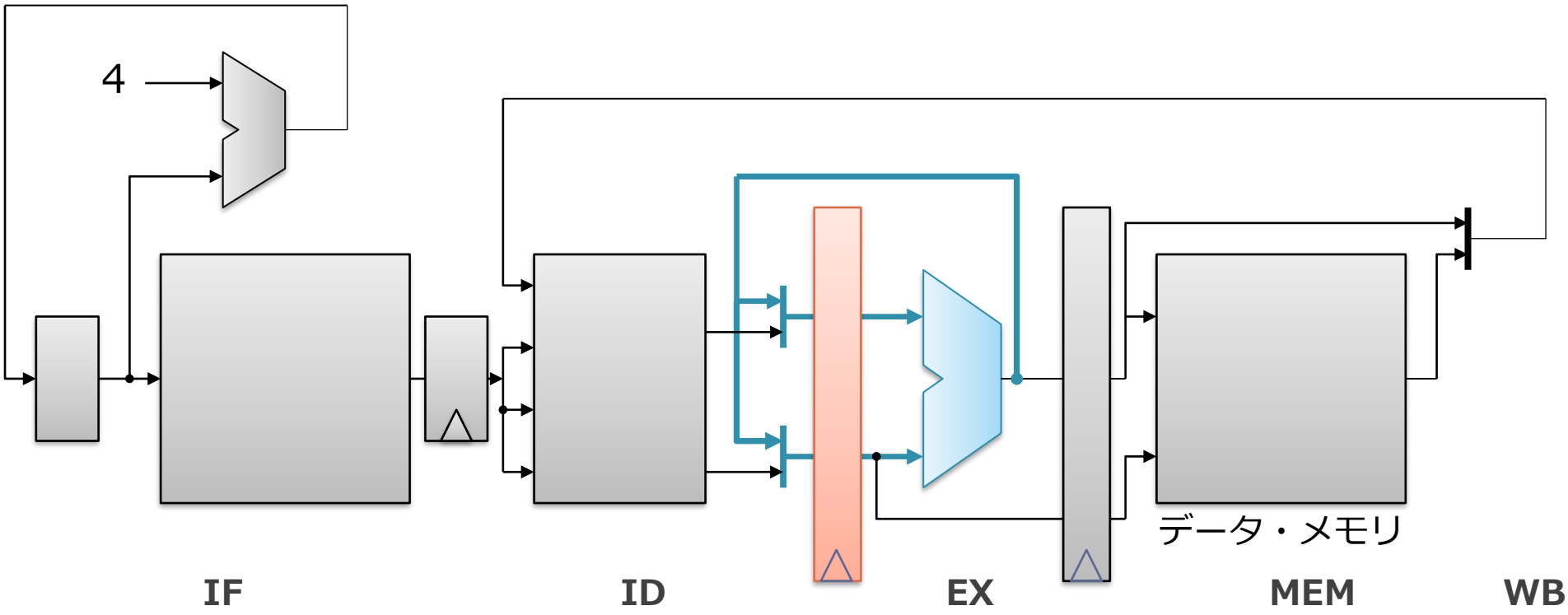
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. **フォワーディング**

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―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

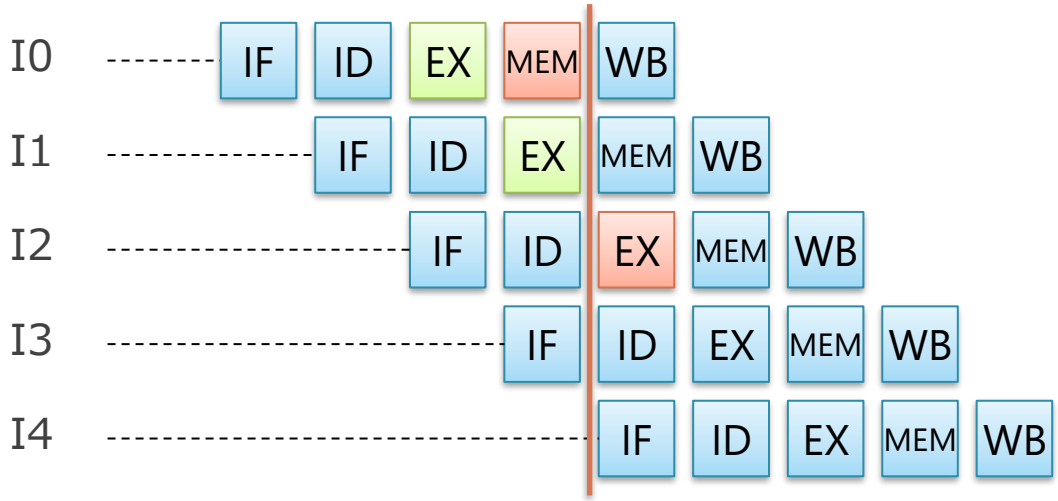


- 演算器の結果を，フィードバック
  - レジスタ・ファイルからの読み出し結果と選択して入力に

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

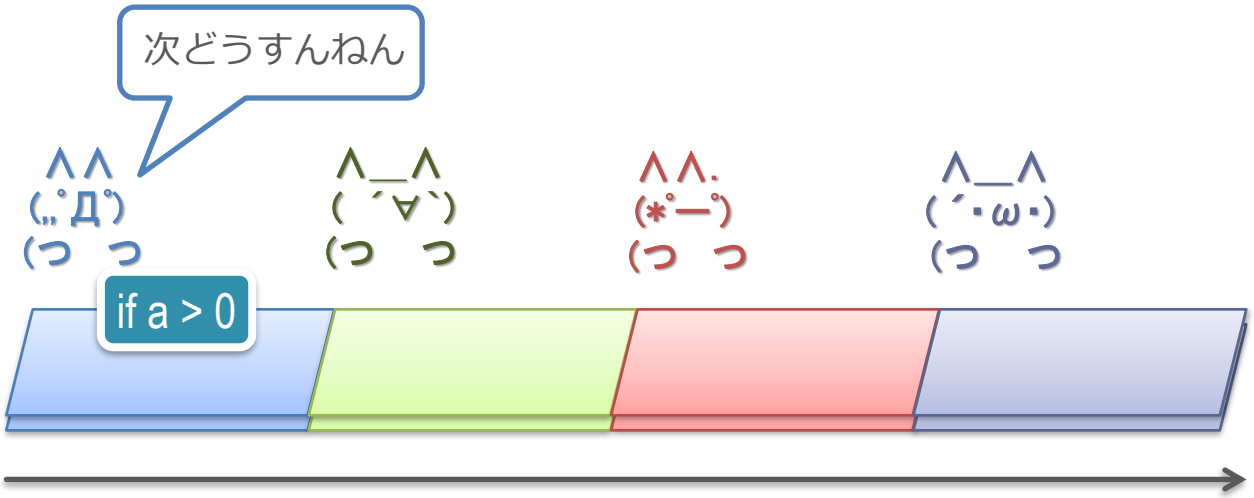
1. 構造ハザード

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

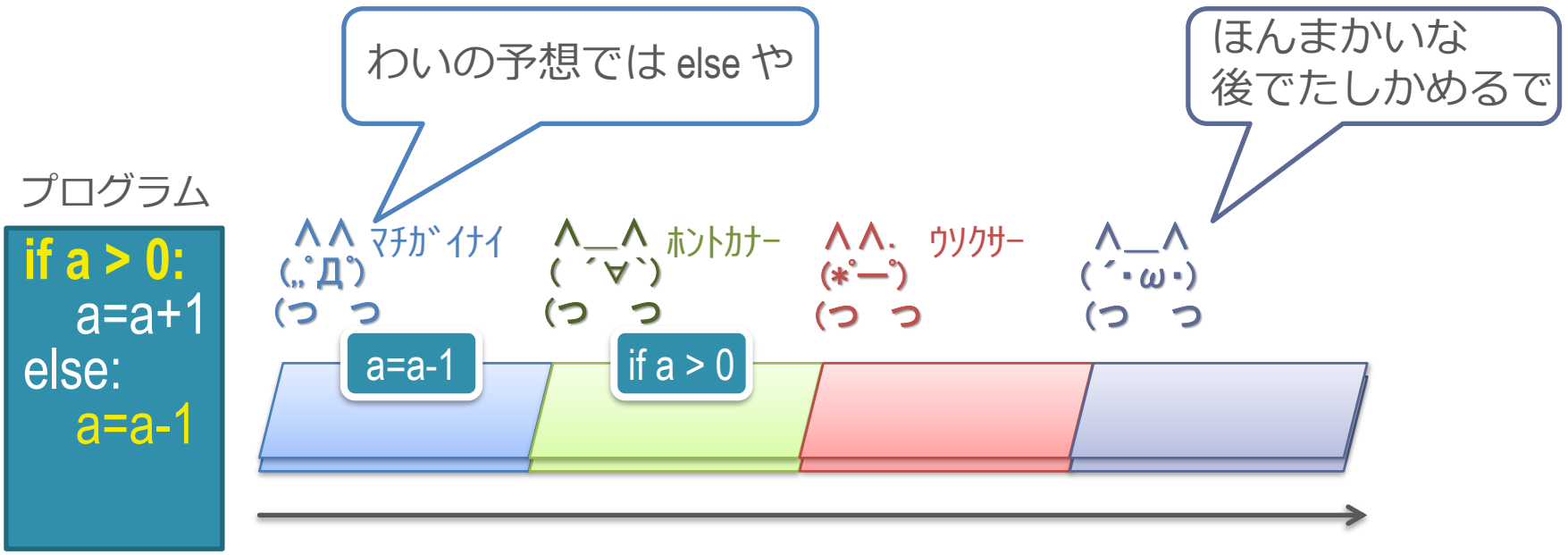
制御ハザードの解消方法

- 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

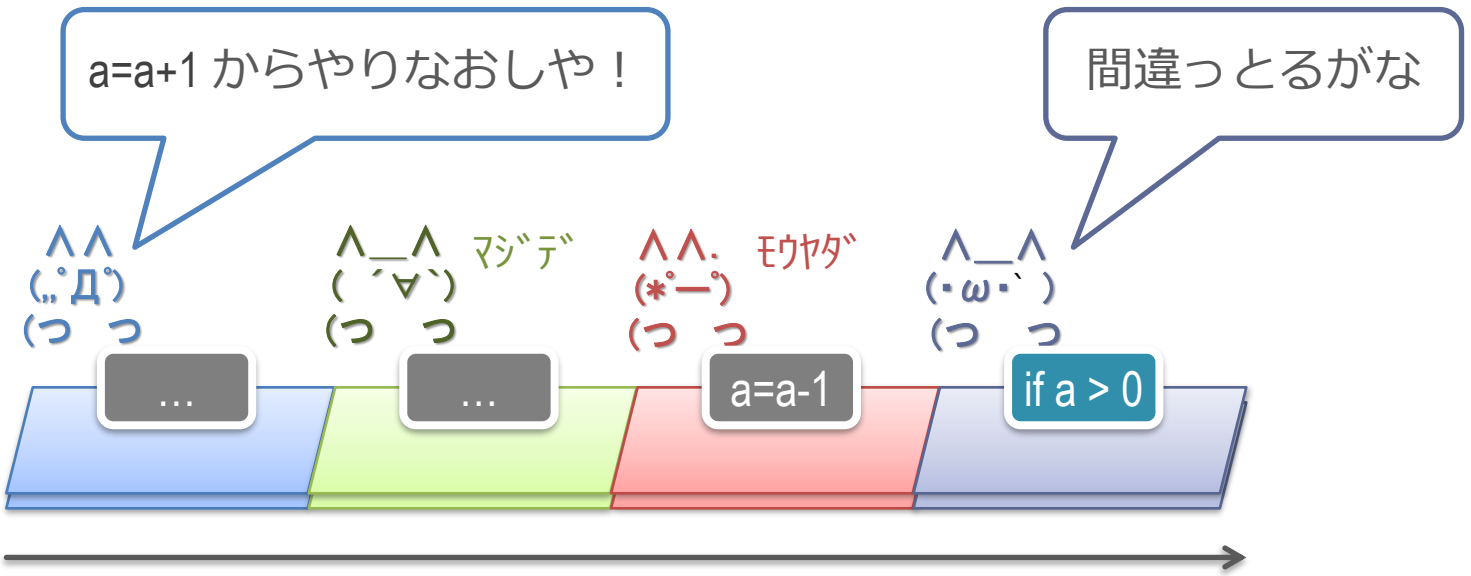


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

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

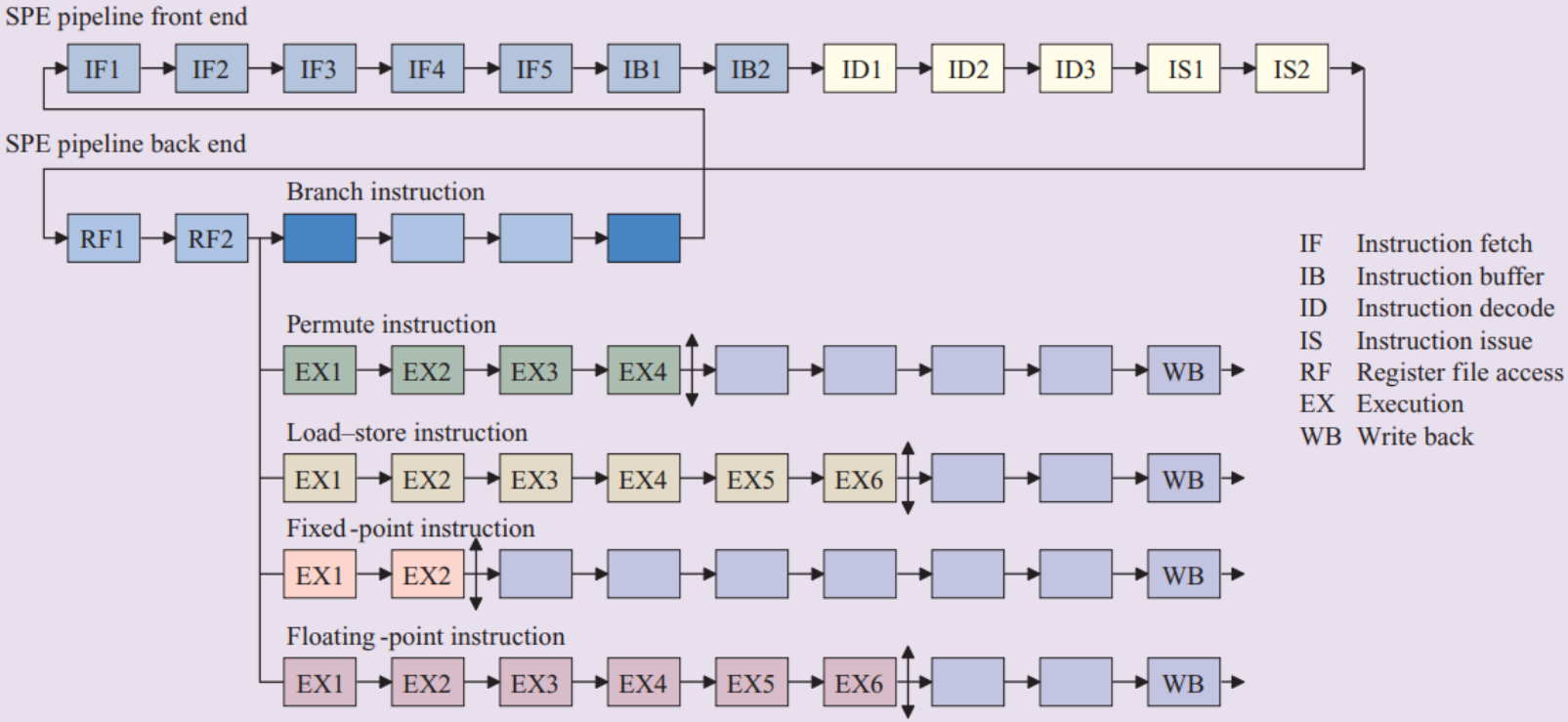
- バックエッジがないパイプライン
  - （回路的な限界にあたるまでは
  - パイプライン段数を増やせば増やすほど性能（周波数）が上がる
- バックエッジがあるパイプライン
  - パイプライン段数を増やすと，
    - 周波数そのものは上がる・・・が，
    - 場合によって，命令を処理できる実効的な速度が落ちる
    - 特に分岐予測ミス・ペナルティによる性能低下が大きい

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

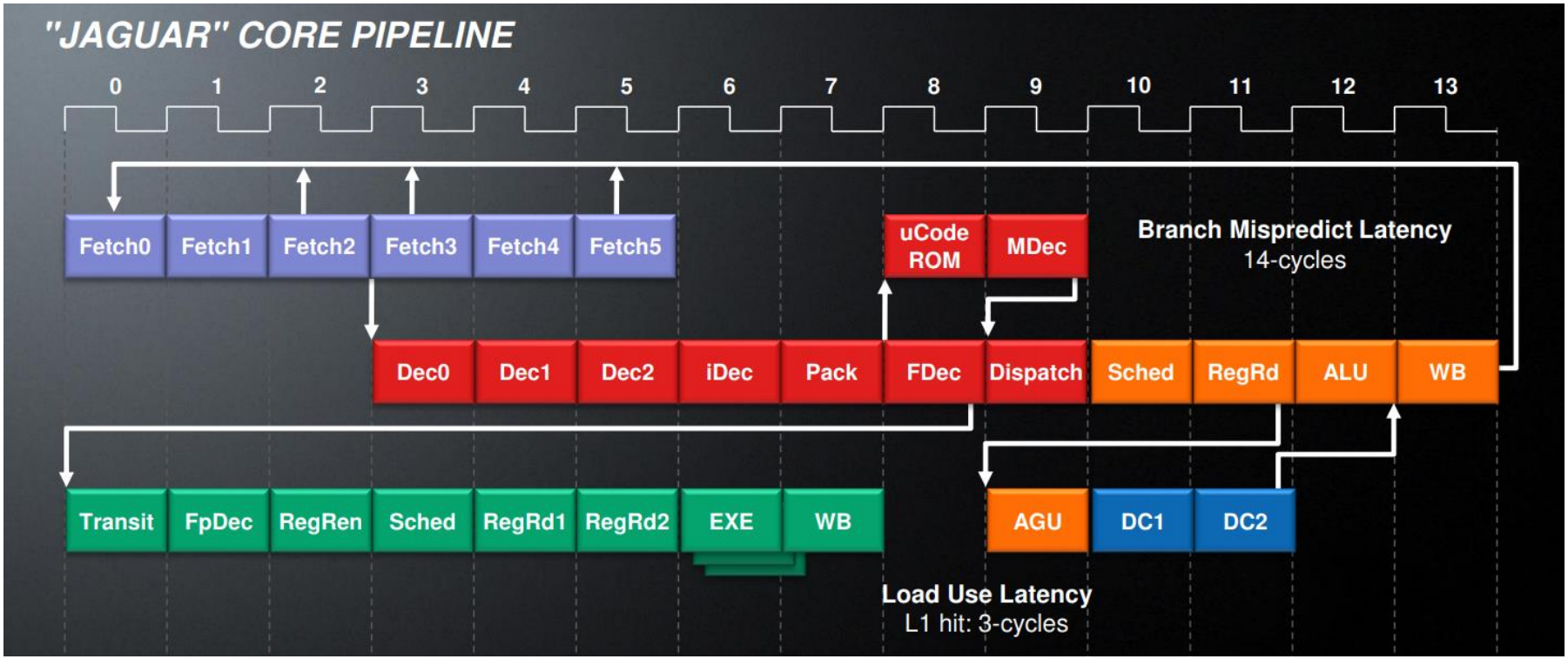
Sony/IBM/東芝 Cell (SPE)

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



AMD JAGUAR

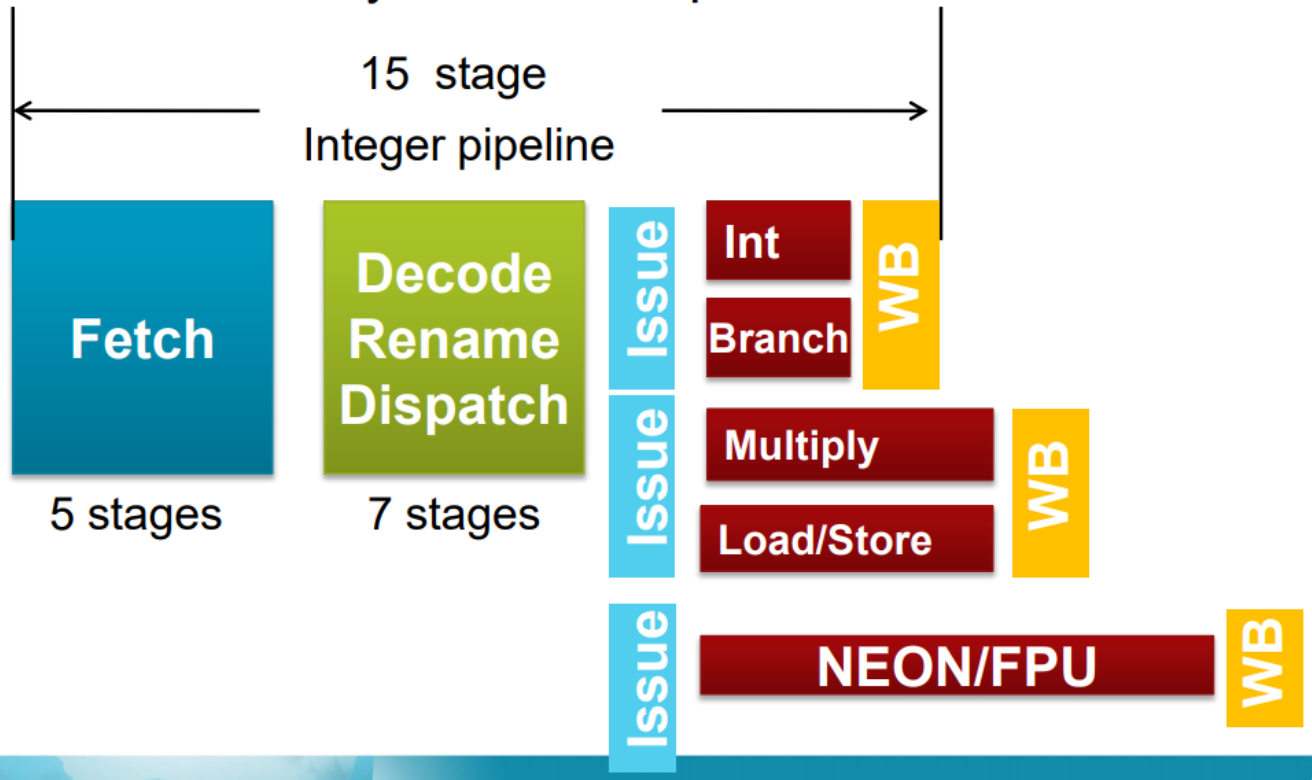
"JAGUAR" AMD's Next Generation Low Power x86 Core より



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

- RISC-V の「add x1←x2,x3」命令を２進数で表記すると以下の通りとなる
  0000000 00011 00010 000 00001 0110011
  （rd←rs1,rs2 として考える）
  - (1) 上記を sub x1←x2,x3 に書き換え，２進数と１６進数の双方で表記せよ
  - (2) 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
  - (3) 上記を addi x1←x2,16 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

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

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

提出方法

- 以下を提出：
  1. 課題 6 ：
     - 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
  - 次回講義開始まで

- 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
  - 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

質問や感想など

質問や感想など

- パイプライン化について、コンピュータが裏で様々な作業を平行して行っているのと似ているなと感じました。各工程の猫ちゃん、各々の性格が垣間見えて可愛かったです。
- パイプラインの動画がわかりやすかったです。アスキーアートの猫もかわいかったです！

質問や感想など

- NANDをどう変換したらORやNORになるかは分かったのですが回路をどう変えたら良いのか分かりませんでした……(課題)
  - 結構パターンは決まりきってるので，何個か作ってるうちにわかってきます

質問や感想など

- そしてやたらとパイプライン段階を増やせばいいというものでもないということを理解しました。

質問や感想など

- パプライン化はバッファリングと似たようなイメージなのかなと思いました。

質問や感想など

- 最初にベルトコンベアの例を用いて説明してくださったおかげで、その後何を考えていくのかとてもわかりやすく、難しくて全部は理解できていませんが、イメージだけは掴めました。今まで、勝手にコンピューターが効率的に動いているものだと思っていましたが、コンピューターが効率よく動くのは人間の設計によるものである、という認識を持つことができました。プロセッサに、ベルトコンベアの考え方を反映したように、実世界をコンピュータに取り入れることで、効率的なコンピュータを作る方法なのかなと思いました。

質問や感想など

- バッテリーは、スクリーンの明るさを保つ他に、中で様々な信号を出すために使われているのでしょうか？バッテリーが電流として使われる、ということでしょうか？

質問や感想など

- P型とN型がどちらも閉じていて、直列に直列につながっていた場合０と１どちらの値になるのですか？？
  - 大電流が流れて壊れるので，それが起きないように設計します

質問や感想など

- 資料の猫が可愛くてニコニコしてました。わかりやすくてよかったです。

質問や感想など

- 今回も難しかったです…。D-FF回路の動作など、授業中詳しく説明されているものは詳しい仕組みまでしっかり理解しておかなければいけないんでしょうか？
  - ものにもよりますが，D-FF の動作などはふわっとしもので大丈夫です.
  - 課題で出だしてる問題は解けるようになっておいてほしいです

質問や感想など

- パイプライン化すると、操作が終わっても次の操作に移れないこともあるという弊害があり、その弊害を少なくするために一つの操作だけ計算量が非常に多い（プログラムの計算量の大半を一つの操作でなしている）ということがないようにするといいとわかりました。パイプライン段階をどのくらいにすれば最短なのか計算する方法はあるのですか？

質問や感想など

- 授業内容とは関係ないことですが、4月授業分の補講、テストの日程がもし決まっていたら教えていただきたいです。
  - すいません，まだ決めかねてます

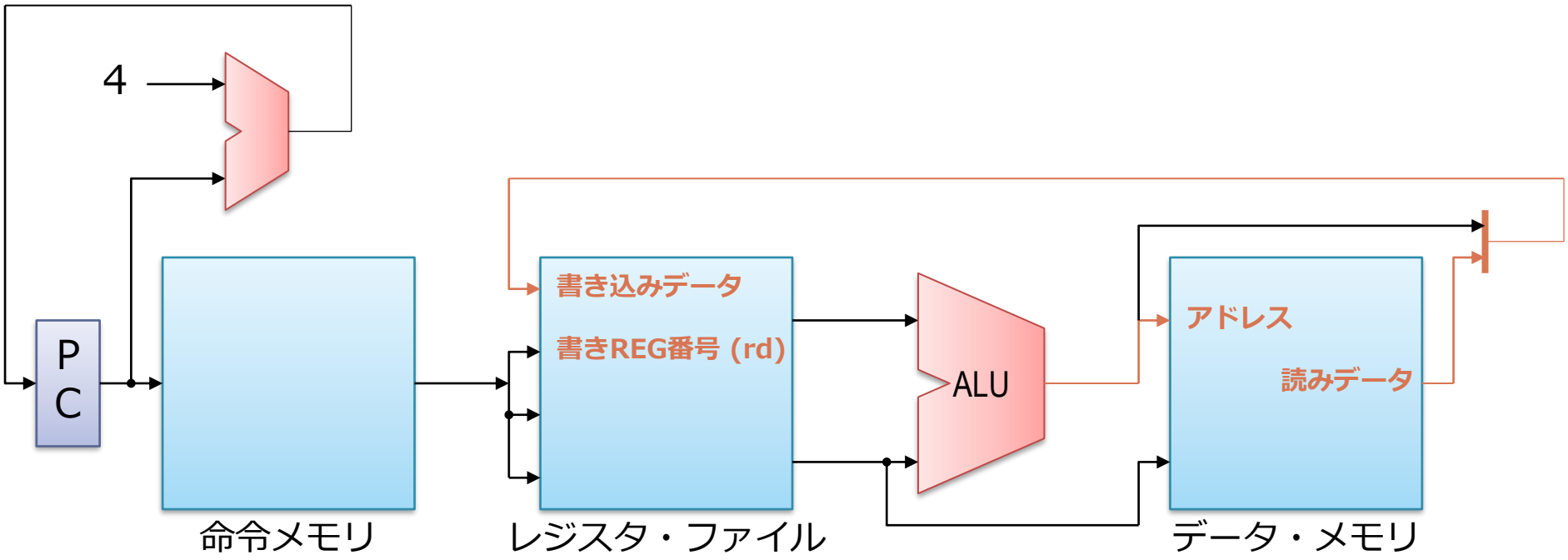
質問や感想など

- P型リレーに０の入力、N型リレーに１の入力を入れて直列に繋ぐとどうなりますか？何となく出力は０になりそうな気がします。
  - P と N の残りの端子に何をつなぐかによりますが，たとえばどう考えてるでしょうか？

質問や感想など

- 授業資料p28のシングル・サイクル・プロセッサの説明のところで質問です。命令メモリ→レジスタ・ファイル→ALUと進んでいった後にまた戻ってALUでアドレスの計算をしていて、この一回戻る工程がめんどくさく感じます。ここは省略できないのですか？
  - 「また戻って」のところが，どう戻ってるのかがよくわかりませんでした

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

質問や感想など

- パイプラインについて、処理を区切るときの工程の最小単位はありますか。

質問や感想など

- 実際に書いてみて、やっとN/Pリレーの図がつかめた気がします。
  電池とスイッチと豆電球を使ってつくってみたら楽しそうです。

質問や感想など

- 今回の授業でも、パイプラインや命令フェッチなど、難しい単語が多く出てきました。

質問や感想など

- IF,ID,EX,MEM,WB等出てきましたが、IFは命令フェッチの略ということですか。
  - そうです

質問や感想など

- 前回の課題で、LABELを2つ用意した（LABEL_UP,LABEL_END)のですが、同じコード内で複数ラベルを用いても良いですか。
  - はい OK です

質問や感想など

- Turing Completeが面白そうだったので、買ってみました。

質問や感想など

- あとP型・N型リレーの課題が難しかったが、パズルみたいで面白かった。

質問や感想など

- シングルサイクルプロセッサにフリップ・フロップを追加する方法が説明されていましたが、実際にどのようにしてこれを実装するのか、具体的な手順や考慮すべきポイントについて詳しく知りたいです。

- また、フリップ・フロップを追加する際に発生する遅延や、その遅延がクロック周波数に与える影響についても知りたいです。特に、どのような条件で遅延が問題になるのかについて具体例を交えて説明していただけると助かります。

質問や感想など

- 課題４はAが0の時も-2されるのでしょうか、、？
  - 講義資料内の例や口頭では説明していたのですが，ループ初回の大小比較まで for ループを再現するとちょっと複雑になるので，ここででは省いていました

質問や感想など

- パイプライン化すると、スループットが向上することは理解できたのですが、やっぱりパイプライン化する方がラッチを入れない時より効率が上がるのでしょうか。（ラッチを入れたことによる遅延もあるのかなと思いました。）