コンピュータ アーキテクチャ I　第7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 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

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
  （rd←rs1,rs2 として考える）
  0000000 00011 00010 000 00001 0110011

  - (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
    0**1**00000 00011 00010 000 00001 0110011
    0**1**00 0000 0011 0001 0000 0000 1011 0011
    0x403100b3

  - (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
    0000000 00**100** 000**11** 000 000**10** 0110011
    0000 0000 0**100** 000**1 1**000 000**1 0**011 0011
    0x00418133

  - (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
    0000000 **1**00**00** 00010 000 00001 0**0**10011
    0000 000**1 0**00**00** 0001 0000 0000 10**0**1 0011
    0x01010093

前回の振り返り

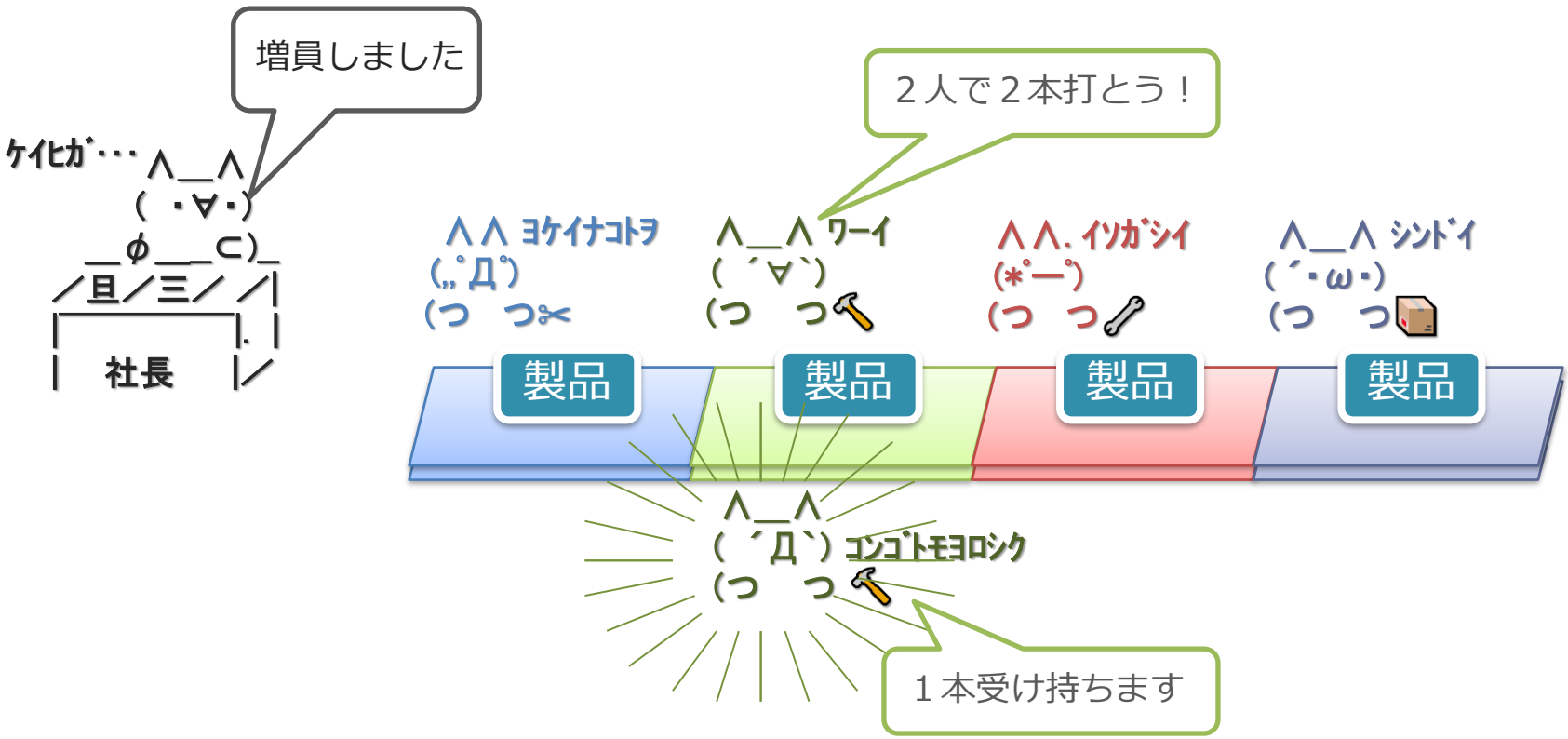
ハザード（hazard）

- 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」

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

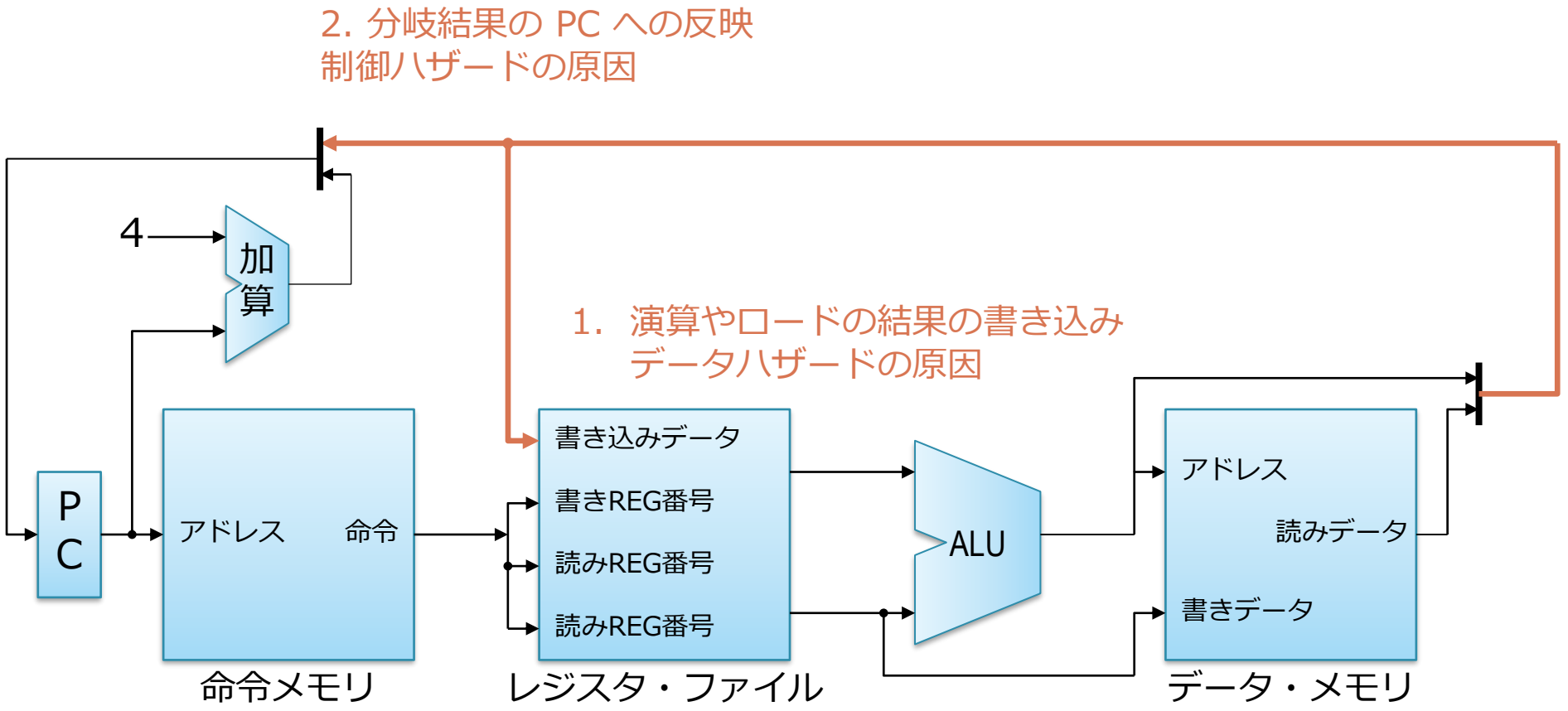
2. 非構造ハザード：バックエッジに由来
   a. データ・ハザード
   b. 制御ハザード

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

バックエッジとは：逆方向（右から左）にいく信号

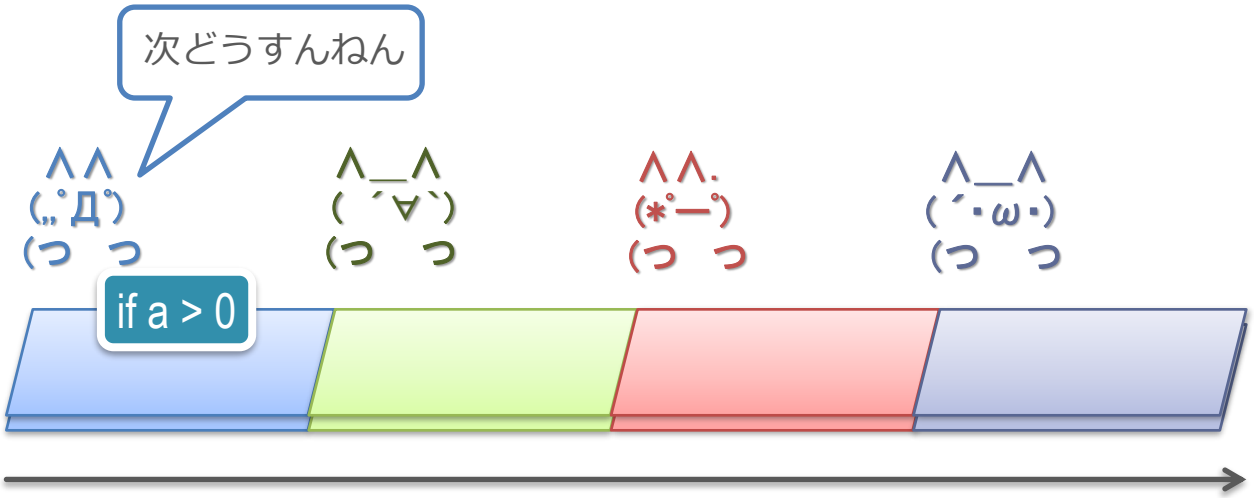


- バックエッジがあるため，命令を単純に流せない場合がある
  - 工場のラインのように，一方向に流せない

分岐命令の処理と制御ハザード

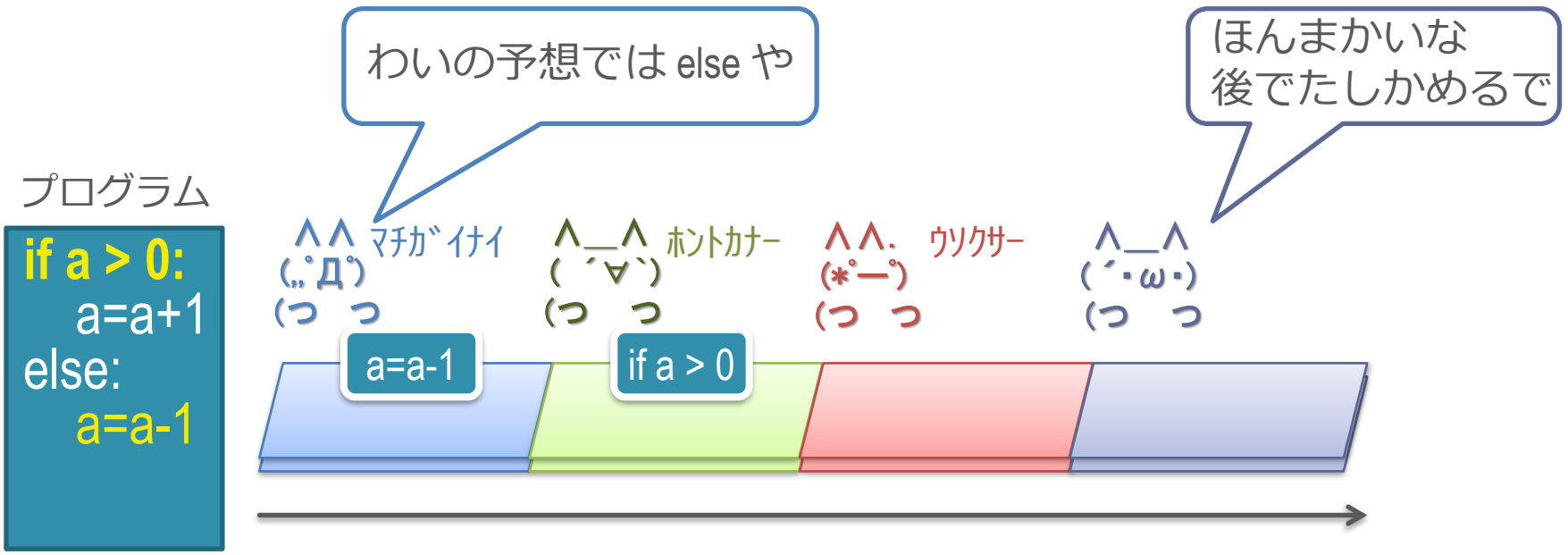
命令メモリ

```
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

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

命令の並列実行

今日の内容

1. 命令の並列実行の基本
2. データ依存
3. 静的命令スケジューリング
4. 動的命令スケジューリング

命令の並列実行

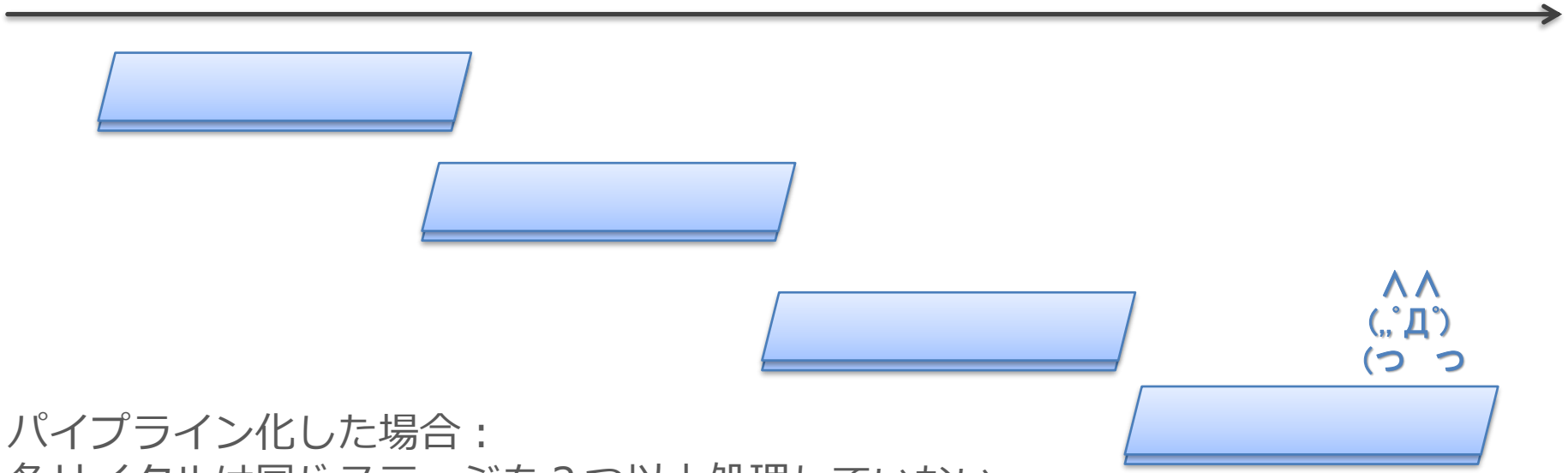
1. スカラ・プロセッサ：
   - １クロック・サイクルあたり１命令を処理
2. スーパスカラ・プロセッサ
   - １クロック・サイクルあたり２命令以上を処理

スカラ・プロセッサ

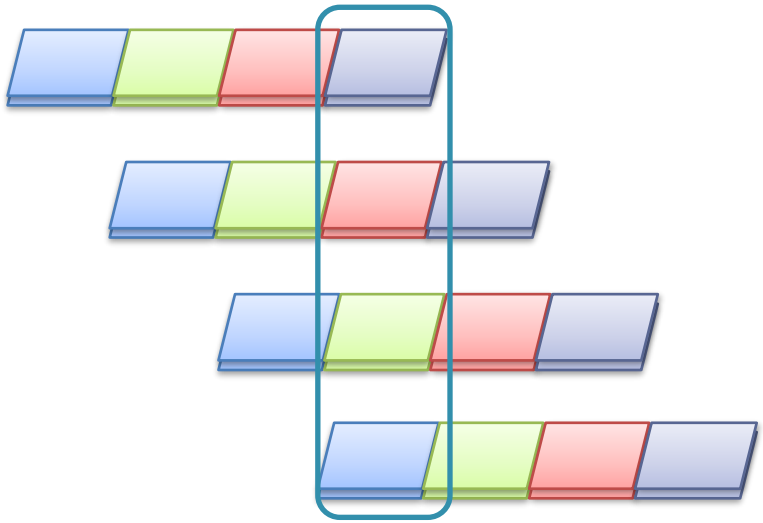
- １クロック・サイクルあたりに単一の命令を実行するプロセッサ
- パイプライン化：
  - １つの命令に関わる処理を分割して毎サイクル並列に実行
  - パイプライン化すると「単一の命令を実行」にならない？
    - １クロック・サイクルあたりでみると，単一の命令を処理
    - １クロック・サイクルに同じステージを複数は処理しない

パイプライン化

パイプライン化しない場合：

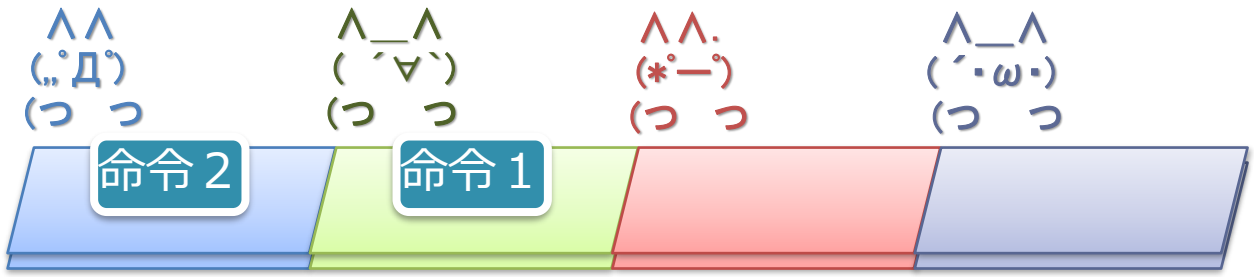


パイプライン化した場合：
各サイクルは同じステージを 2 つ以上処理していない



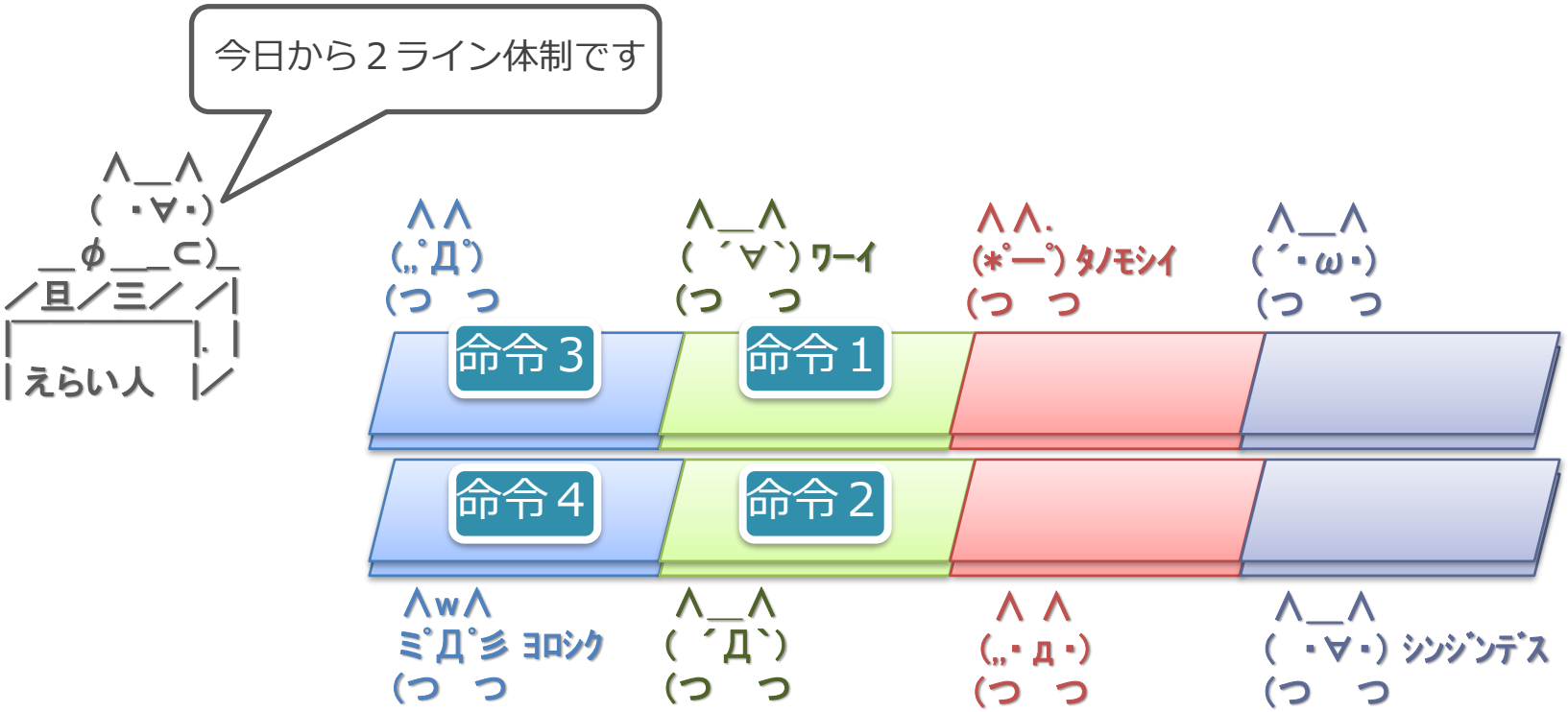
パイプライン化による性能向上の限界（復習）

- パイプライン化による性能向上には限界がある
  1. 回路的な理由による周波数向上の限界
     - D-FF の遅延
     - 電力供給と熱（冷却）の壁
  2. アーキテクチャ的な理由による実効性能の限界
     - ハザードによる実効性能の低下



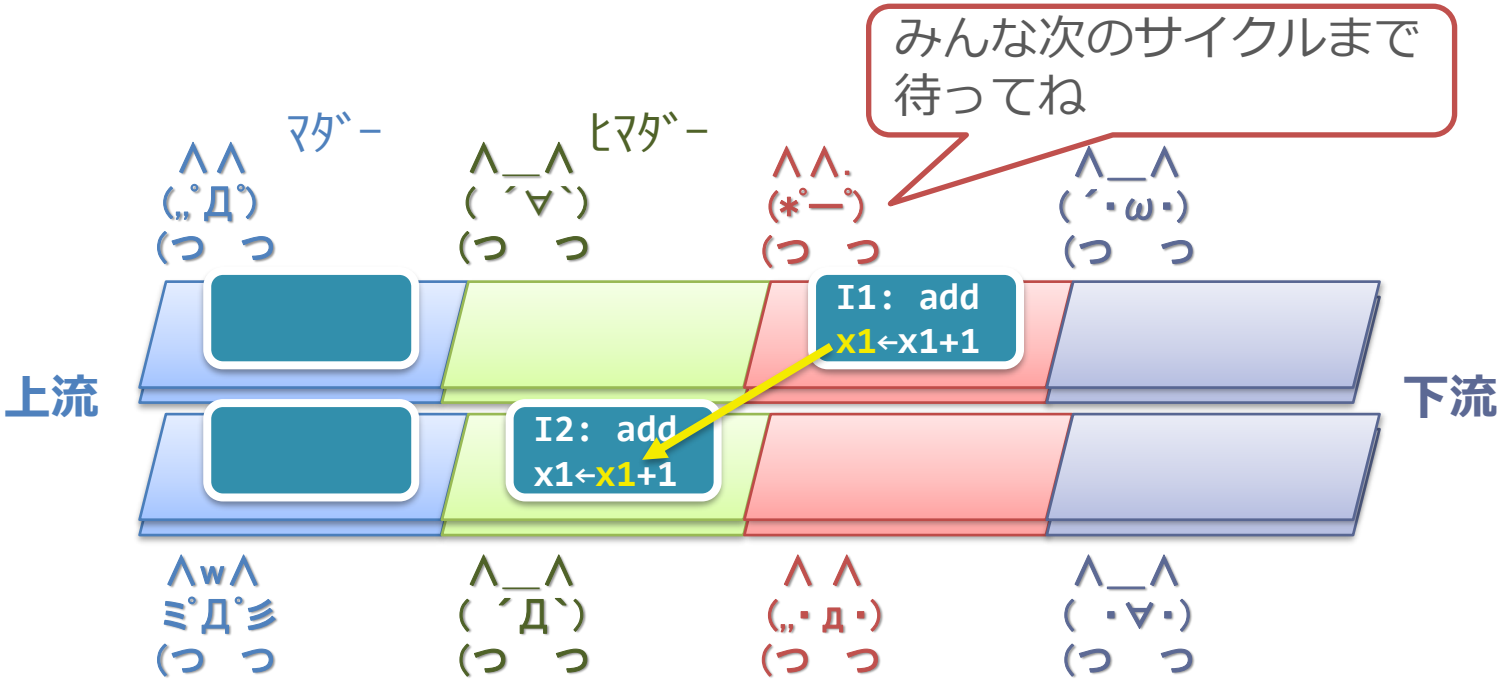
スーパスカラ・プロセッサ（superscalar processor）

- スーパスカラ・プロセッサ
  - パイプラインや関連する演算器などを複数並べる
  - 複数の命令を並行して処理して性能を向上

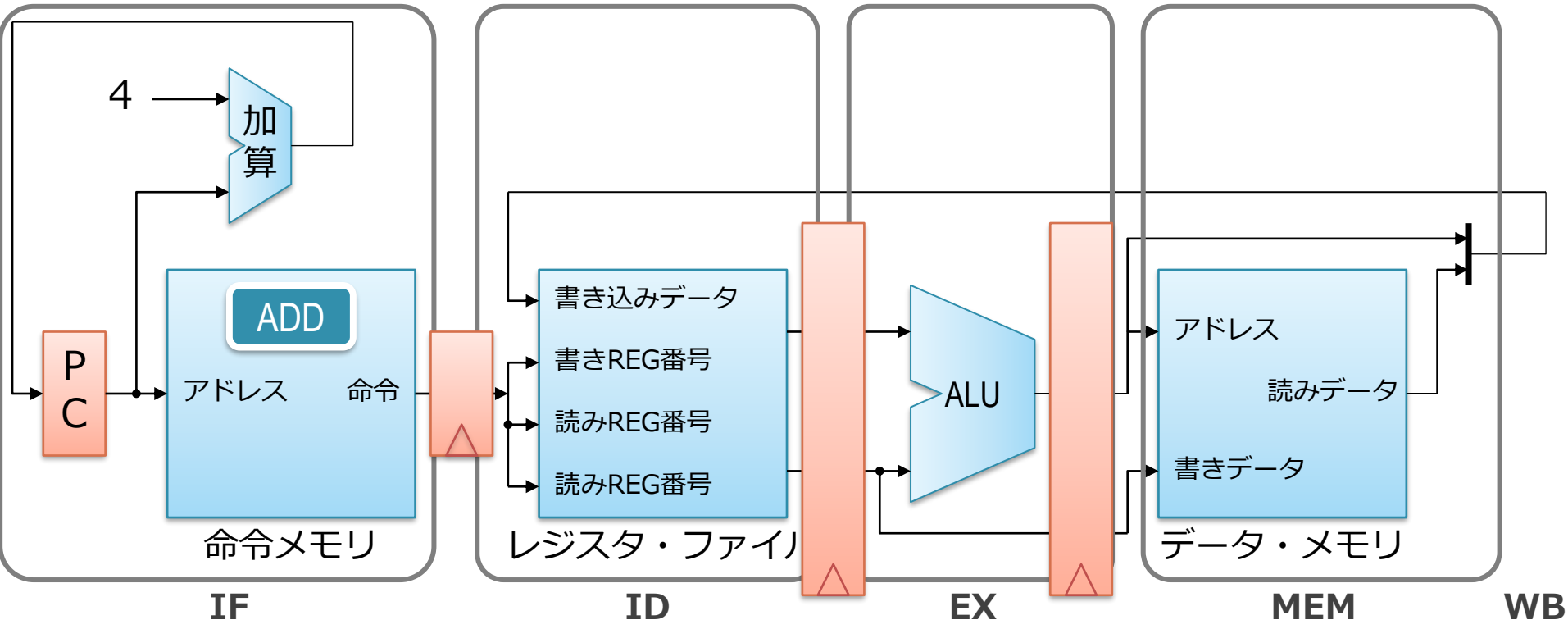


単純なスーパスカラ・プロセッサの動作

- 同時にフェッチしてきた命令間に依存がない場合は並列に実行
  - もし依存がある場合は，後続の命令全てを待たせて処理
    - パイプラインの上流側を全てストールさせる
  - これはプログラムの意味を保つため
    - 下の図だと `I1` と `I2` を並列に計算したらおかしくなる

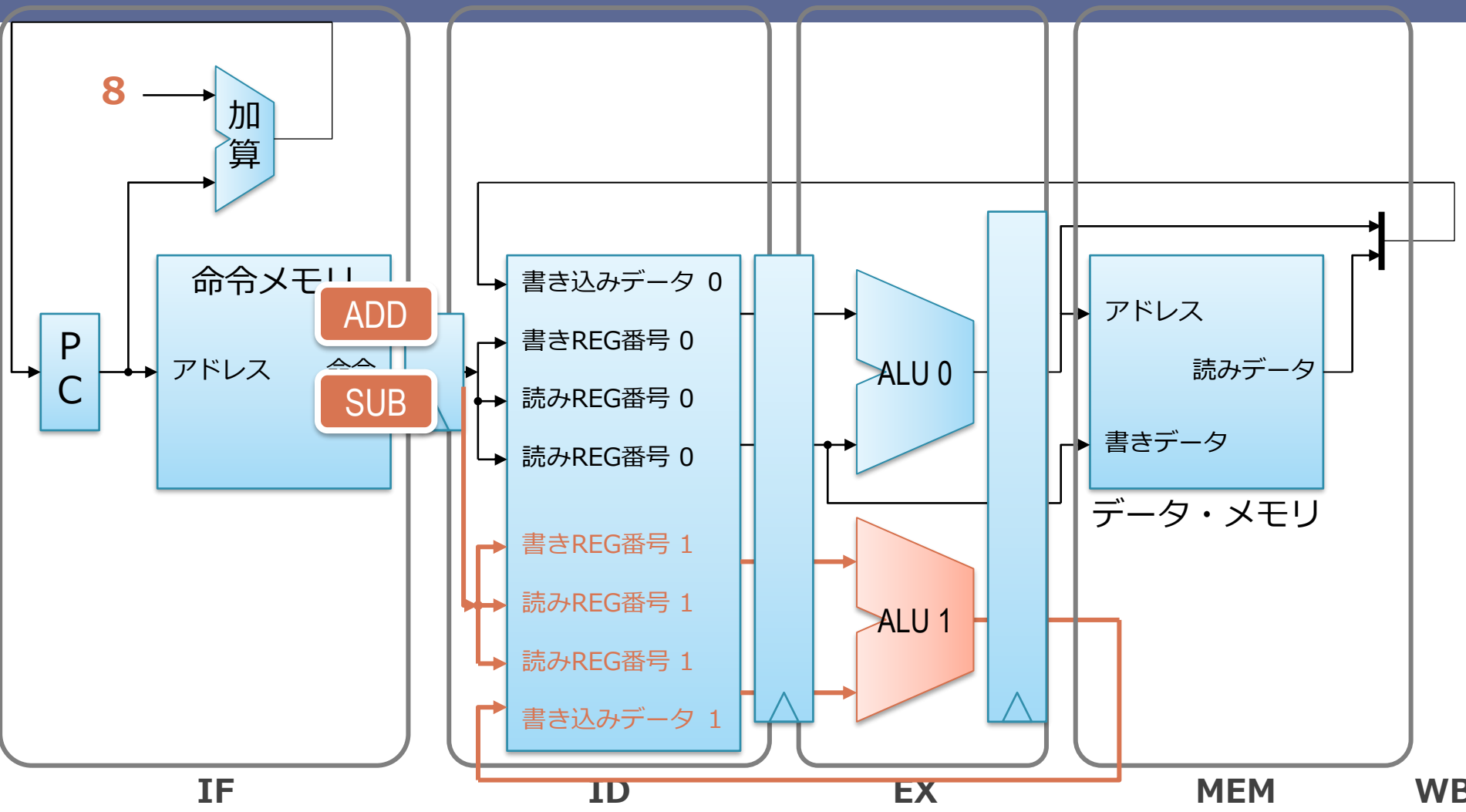


パイプライン化されたスカラ・プロセッサのブロック図



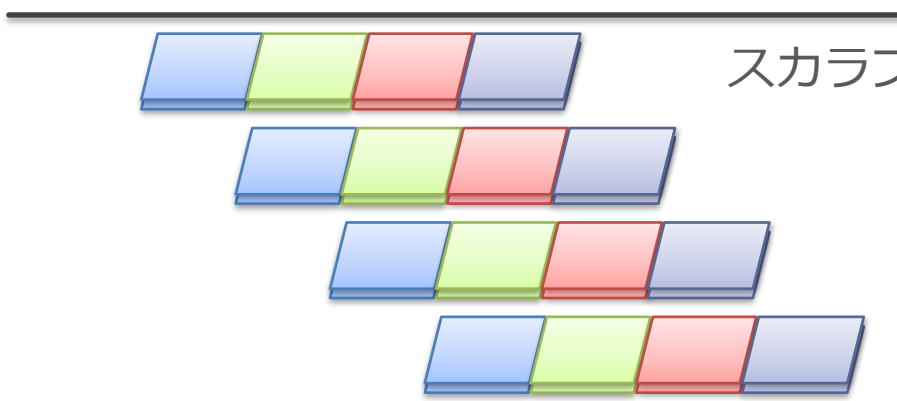
- 各ステージの間に，D-FF（オレンジの四角）をいれる
  - WB の書き込みについては，レジスタ・ファイル自体がクロックに同期して書き込みが行われるので D-FF は不要
- 各ステージの処理が早く終わっても，次のクロックまでは D-FF で信号の伝搬は止まる

単純な 2-way スーパスカラ・プロセッサの例



- フェッチ，レジスタ・アクセス，ALU を 2 命令分に拡張（赤線）
- この例では，データ・メモリは 1 つのまま（並列実行に制限がある）

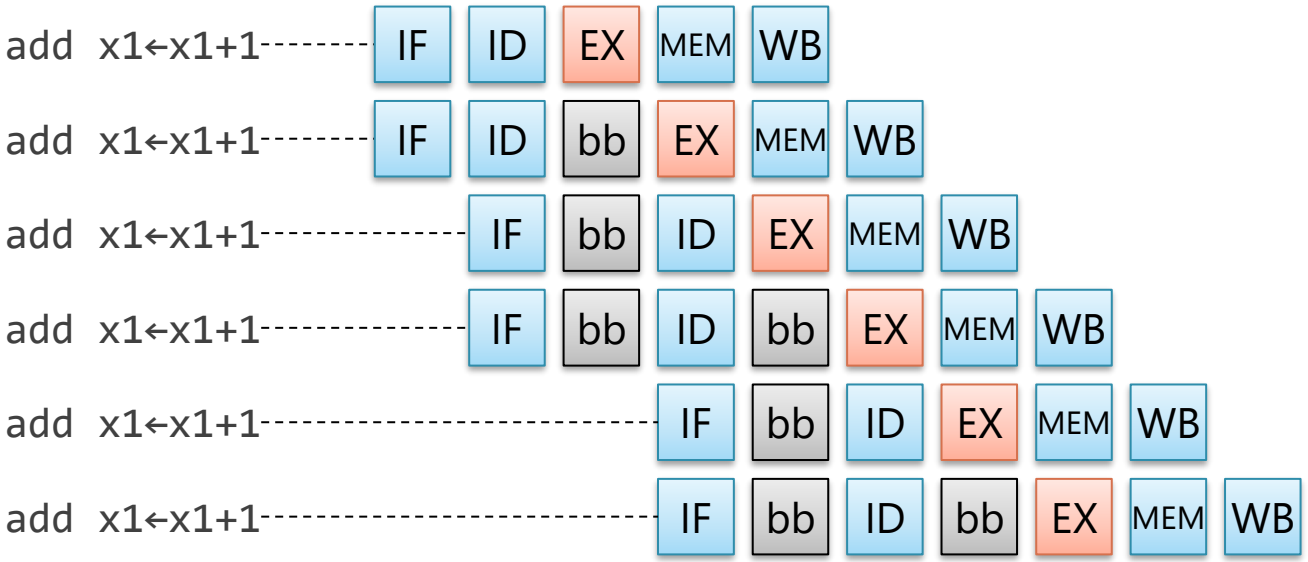
理想的な場合のスーパスカラ（2-way）による性能向上 =単位時間あたりの命令処理数が倍増



スーパスカラによる並列実行の制約

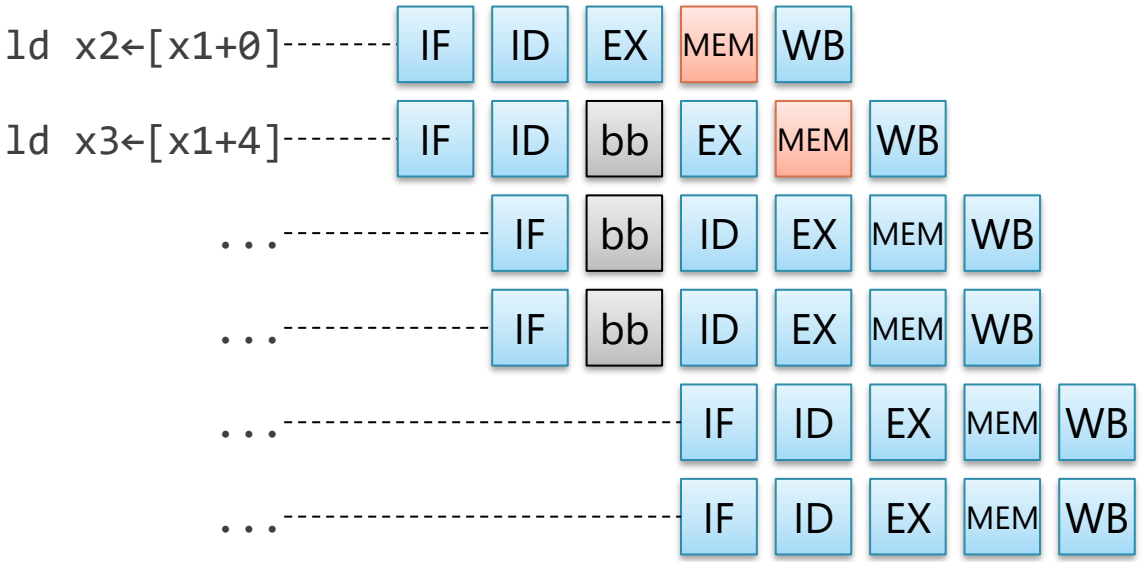
- さまざまな制約があり，実際の性能向上は N 倍にはならない
  - 2-way なら実際には数割ぐらいの性能向上
- 制約の例：
  1. 同時にフェッチされた命令間に依存がある場合
  2. 構造ハザードが起きる場合

1. 同時にフェッチされた命令間に依存がある場合



- 最悪の場合：上記のように全ての命令間に連続に依存があるとき
  - 演算が逐次的に行われるようにバブルが入る
  - この場合はスカラ・プロセッサから全く性能があがらない

2. 構造ハザードが起きる場合



- 例：先ほどのブロック図のように，データ・メモリが１つしかない場合
  - ロード命令は１サイクルに１つしか実行できない
  - 上記のように，ロードが連続するとバブルが入る

- 回路規模が大きい & 使用頻度が低い演算器はパイプライン間で共有されることが多い = 複数同時に来ると止まる
  - 乗算器，除算器，超越関数の演算器など

単純なスーパスカラによる並列実行のまとめ

- これまでに説明したような単純なスーパスカラではあまり大きな性能向上が期待できない
  - 2-way なら数割ぐらいの向上
- 同時実行幅（way 数）を増やしていっても,
  何かの制約ですぐ止まる
  - n 命令のうち１つでもひっかかってたらダメ

同時実行幅を増やしていっても，何かの制約ですぐ止まる

- どうする？
  - 1. 構造ハザード
    → 必要な回路を増やせば解決する
    （当然にコストが増える
  - 2. データ依存
    → **命令スケジューリング**

今日の内容

1. 命令の並列実行
2. **データ依存**
3. 静的命令スケジューリング
4. 動的命令スケジューリング

命令間の依存関係

- 命令のスケジューリング
  - プログラムの意味を変えずに，命令の実行順を並び変えること
  - これによって並列に実行できる命令を増やす
- プログラムの意味が変わらない = 依存関係をくずさない
  - 以降のスライドでは，スケジューリングの背景として命令間の依存関係を整理しておく

命令間の依存関係

1. 制御依存
2. データ依存
    1. 真の依存
    2. 偽の依存

制御依存

- 分岐とその後ろにある命令間の依存
  - 分岐命令の後ろにある命令は，分岐先がわかるまで実行不能
  - 分岐先が確定するまでどこを実行すれば良いか不明なため
- これは分岐予測による投機実行により，効果的に解決できる（前回の講義）

データ依存

1. 真の依存
   - フロー依存：　　RAW（read after write）
2. 偽の依存
   - 逆依存：　　WAR（write after read）
   - 出力依存：　　WAW（write after write）

真の依存：フロー依存 RAW（read after write）

- 文字通り，同じレジスタを「書いた後に読む」際の依存
  - 「真の依存」，「フロー依存」，「RAW」は
    呼び方が違うだけでおなじものを指している
  - 一般に「データの依存関係」と言われたら思い浮かべるもの

- 真の依存の例：I1 が終わらないと I2 は実行できない

```
I1: add x1←x2+1

I2: add x3←x1+1
```

偽の依存１：逆依存 WAR（write after read）

- 同じレジスタを「読んだ後に書く」
  - 真の依存（書いた後に読む）と方向が逆
- 逆依存の例：

```
I1: add x2←x1+1

I2: add x1←x3+1
```

- I1 と I2 の間には真の依存は存在しない
  - 「←」の右の入力部分だけみると順番を入れ替えても問題ない
  - しかし，もしスケジュールして I2 を先にやると x1 が破壊されてしまう

逆依存がある場合にスケジューリングをすると結果が壊れる

- 逆依存がある命令列：

```
// 初期状態：x1=1, x2=0, x3=3
I1: add x2←x1+1
I2: add x1←x3+1
// 終了状態：x1=4, x2=2, x3=3
```

- I1 と I2 の実行順を入れ替えた場合：

```
// 初期状態：x1=1, x2=0, x3=3
I2: add x1←x3+1
I1: add x2←x1+1
// 終了状態：x1=4, x2=5, x3=3
```

偽の依存２：出力依存
WAW（write after write）

- 同じレジスタを「書いた後に書く」
- 出力依存の例：

```
I1: add x1←x2+1
          ↓
I2: add x1←x3+1
```

- 逆依存と同様に，I1 と I2 の間には真の依存は存在しない
  - 「←」の右辺にある入力部分だけをみると，順番を入れ替えても問題なさそう？
  - しかし，スケジュールして I2 を先にやると I1 により x1 が破壊されてしまう

出力依存がある場合にスケジューリングをすると結果が壊れる

- 逆依存がある命令列：

```
// 初期状態：x1=1, x2=0, x3=3
I1: add x1←x2+1
I2: add x1←x3+1
// 終了状態：x1=4, x2=2, x3=3
```

- I1 と I2 の実行順を入れ替えた場合：

```
// 初期状態：x1=1, x2=0, x3=3
I2: add x1←x3+1
I1: add x1←x2+1
// 終了状態：x1=1, x2=2, x3=3
```

真の依存と偽の依存

- 真の依存は原理的に取り除きようがない
- 偽の依存（逆依存と出力依存）はレジスタを使い回すことによって発生する
  - いろいろ取り除きようがある

偽の依存の解消の例

- たとえばレジスタがたくさんあれば，他のレジスタを使うようプログラムを書き換えて偽の依存を取り除ける
  - 逆依存（x4 を使うよう書き換え）

```
I1: add x2←x1+1          I1: add x2←x1+1
I2: add x1←x3+1    ⇒     I2: add x4←x3+1
```

  - 出力依存（x4 を使うよう書き換え）

```
I1: add x1←x2+1          I1: add x1←x2+1
I2: add x1←x3+1    ⇒     I2: add x4←x3+1
```

- ただし，使えるレジスタの数には限りがある
  - 上記は「たまたま」x4 は使っていなかった場合の例
  - 一般に記憶回路の容量と速度はトレードオフがある
    - 講義第２回「なぜレジスタとメモリがあるのか？」

今日の内容

1. 命令の並列実行
2. データ依存
3. **静的命令スケジューリング**
4. 動的命令スケジューリング

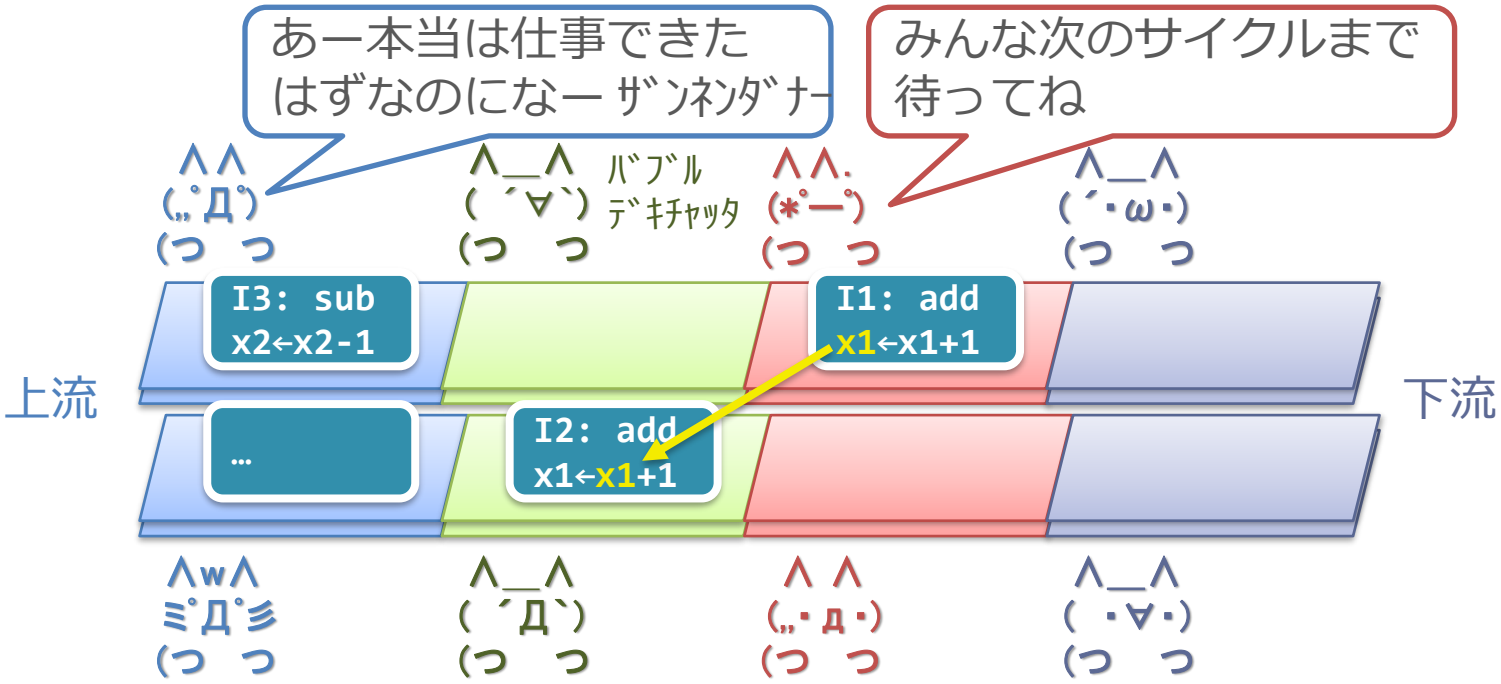
静的命令スケジューリング

- 静的命令スケジューリング
  - 並列実行できるようにプログラム内の命令を並びかえておく方法
  - 通常はコンパイラが行う
- 静的 vs. 動的
  - 静的：
    - 事前にプログラム内の命令を並び替えておく
    - = CPU からみると実行順は変化しない
  - 動的：
    - CPU が実行時に並び替える

単純なスーパスカラでの実行の例

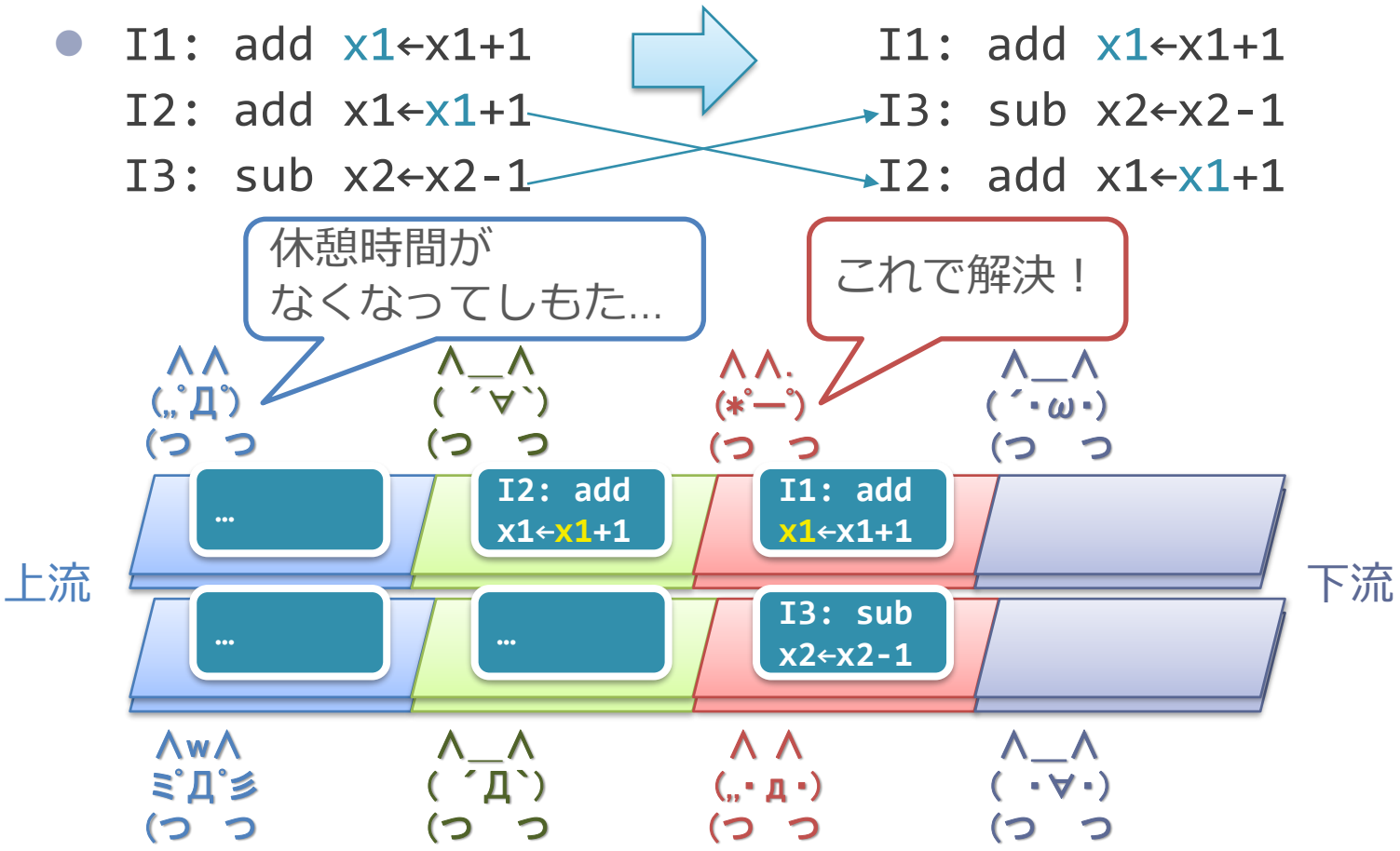
- 下記のコードでは I1 と I2 には真の依存があるが，I3 は無関係
  - しかし，上流が全部とまるので，I3 も実行できない

  - I1: add x1←x1+1
    I2: add x1←x1+1
    I3: sub x2←x2-1

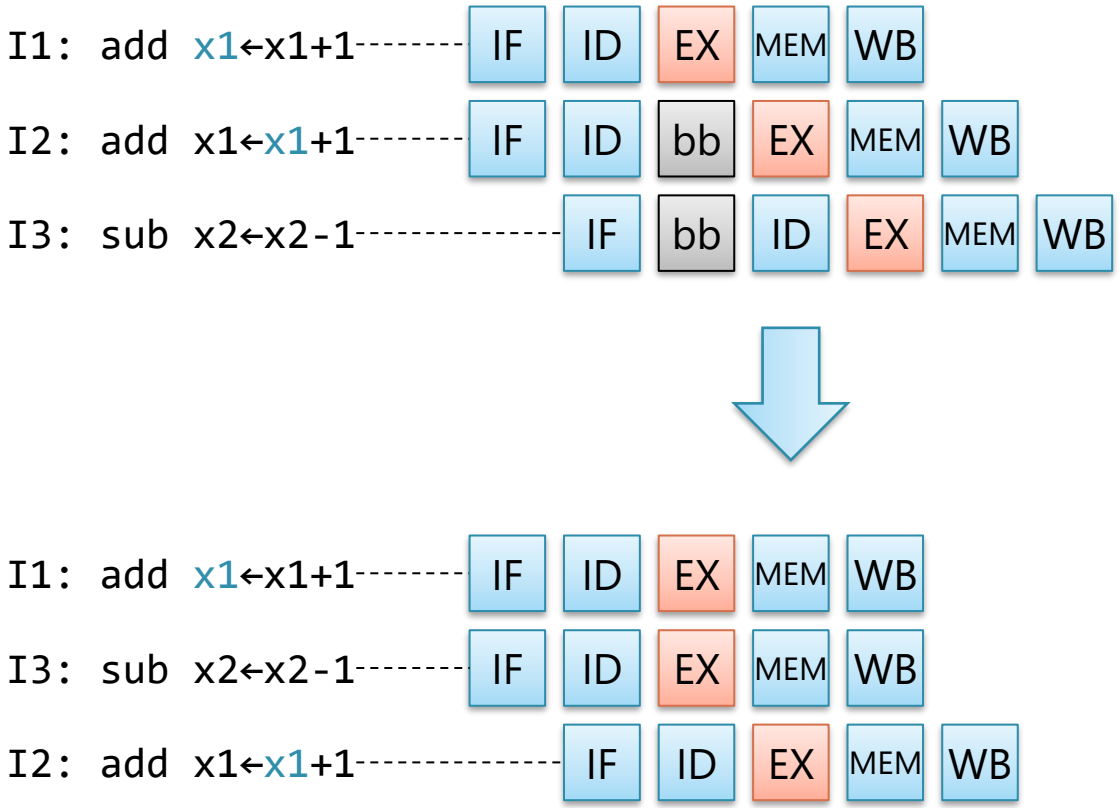


静的スケジューリングによる解決

- I2 と I3 を入れ替えておけば，パイプラインはとまらない
  - I1 と I3 が同時にパイプラインに投入される



静的スケジューリングによる解決



- I2 と I3 の順序を入れ替えたので，バブルが消えた

VLIW：Very Long Instruction Word

- 静的スケジューリングを前提としたCPUのアーキテクチャ
- 通常の命令を複数にまとめたものを１つの命令とする
  - 命令セットの仕様として，１つのVLIW命令内では依存関係を持てないようにする
  - ＝フェッチした後は必ずそれらは並列実行できる

```
I1: add x1←x1+1
I2: add x2←x2+1
I3: sub x3←x3-1
```

通常の命令セット

```
I1: add x1←x1+1
    add x2←x2+1
    sub x3←x3-1
```

VLIW ではこれで１命令
仕様としてこのグループの中に
依存関係があることを許さない

VLIW の利点と問題点

- 利点：
  - ハードウェアがすごく簡単
    - スーパスカラでは依存を検出して止める機構があった
    - VLIW では依存を検出して止める機構は不要
      - 仕様として 1 つの VLIW 命令内に依存は発生しない
- 問題点：
  1. 性能向上に限界がある
  2. 互換性がとりにくい

VLIW の問題１： 性能がいまいち出ない

- VLIW は静的スケジューリングに全面的に頼っている
  - あらかじめコンパイラが VLIW 命令を頑張って作る
  - しかし，コンパイラが出来る並び替えは結構自由度が低い

静的スケジューリングが難しい例 1

- 例1：分岐を乗り越えた並び替えは難しい
  - 3 行目のメモリ・アクセスを if 文の前に持ってくるのは困難
  - うかつにやるとメモリ・アクセス例外が起きて落ちる
    - NULL ポインタにアクセスしてしまう

```
1: i = i + 1
2: if (flag)
3:     a = *ptr; // flag が false の時は ptr は NULL
```

静的スケジューリングが難しい例２

- 例２：ポインタ参照の順番を入れ替えるのは難しい
  （不可能ではない）
  - ２ 行目と３行目のメモリ・アクセスを入れ替えることは困難
  - うかつにやると意味が変わる

```
1: func(int* a, int* b, int z){
2:      *a = z + 1;  // z+1 より先に c = *b を始めたい
3:      int c = *b;  // しかし a は b と同じ場所を
                     // 指している可能性がある
```

余談：C 言語などでのポインタ経由アクセス

- 以下では，a と b のために2回分のロード命令が生成される
  - メモリからレジスタに読み出した値は，普通は使い回す
  - しかし間にグローバル変数へのアクセスが入ると，
    一回 *ptr をロードしてレジスタに置いた値が使い回せない

- ```
  int g = 0;
  func(int* ptr){
      int a = (*ptr) + 1; // *ptr を読み出すロードが生成される
      g = 1; // ptr が g を指している可能性がある
      int b = (*ptr) - 1; // ここにもう一度ロードが入る
  ```
  ```

余談：C 言語などでのポインタ経由アクセス

- こういうときは *ptr をローカル変数に 1 回コピーしてからアクセスしたほうが速い
  - 以下のコードでは t と g は自明に別の変数

- ```
  int g = 0;
  func(int* ptr){
      int t = (*ptr); // *ptr を読み出すロードはここだけで発生
      int a = t + 1;
      g = 1;
      int b = t - 1;
  ```
  ```

VLIW の問題２：互換性がとりにくい

- 静的に CPU の挙動を仮定して命令をスケジュールする
  - = その仮定がくずすような変更（ハードの改良）ができない
- 要因：
  1. 並列実行幅が固定されている
  2. 実行タイミングを仮定してスケジュールされている
     （発展的なので，付録に

1. 並列実行幅が固定されている

- 仕様として「N 命令相当を 1 つの VLIW 命令 とする」としている
  - 性能を上げるために N を後から増やそうと思っても増やせない
  - 既存のコードが動かなくなってしまう

- たとえば N を 2 から 4 にすると互換性がとれない
  - I1: add x1←x1+1
    add x2←x2+1
    I2: sub x1←x1-1
    sub x2←x2-1
    ある VLIW バージョン 1

I1: add x1←x1+1
add x2←x2+1
sub x1←x1-1
sub x2←x2-1
ある VLIW バージョン 2
そのまま実行すると仕様違反

VLIW が有用な場所

- VLIW はここまでに述べたような理由により，現在主流ではない
- しかし，以下のような場面であれば有用
  1. 絶対性能よりも，ハードが小さいこと（電力）の要求が高い
  2. 動作させるソフトウェアが限られている場合
     - 組み込み分野では，特定の１つのソフトのみが動くような使い方をすることがある
       - 例：炊飯器の CPU（コントローラ）は，お米を炊く制御プログラムしか実行しない
     - こう言う場合は互換性が問題になりにくい
       - 任意のプログラムを実行する必要がないから

VLIW が有用な場所

- 典型的には，組み込み CPU が該当
  - 簡単なハードでそこそこの性能がだしたい時に有用
- CPU を作る学生実験で性能出したい場合なんかでも有望
  - 実装が簡単 & 課題となる少数のプログラムさえ速ければよい
  - 人力の静的スケジュールで最適化する
    - 専用のコンパイラを作るのはめっちゃ大変
    - 実験の課題だけ手で頑張って命令の順序を入れ替えるならなんとかなる

今日の内容

1. 命令の並列実行
2. データ依存
3. 静的命令スケジューリングと VLIW
4. **動的命令スケジューリング**

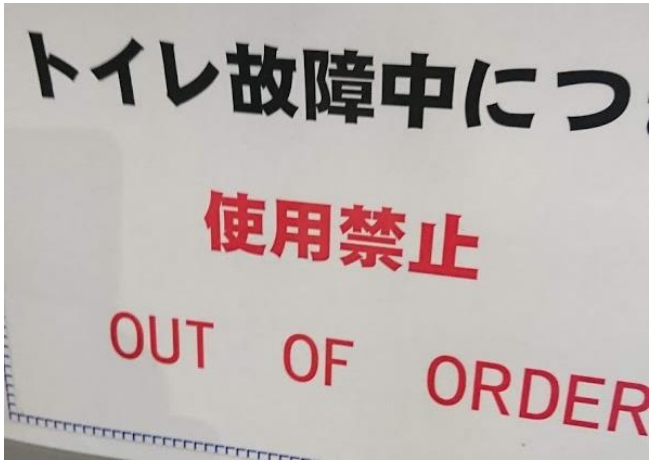
動的命令スケジューリング

- CPU により，うまく並列実行できるように命令を並びかえる方法
  - 静的：事前に並び替えておくので，CPU からみると変化しない
  - 動的：CPU が実行時に並び替える

言葉の定義

- 並び替えに関係する用語：
  - In-order：　　プログラムに書かれている順のこと
  - Out-of-order（OoO）：　上記とは違う順番のこと

- （一般には，公共性の高い機器が故障してることを言うらしい



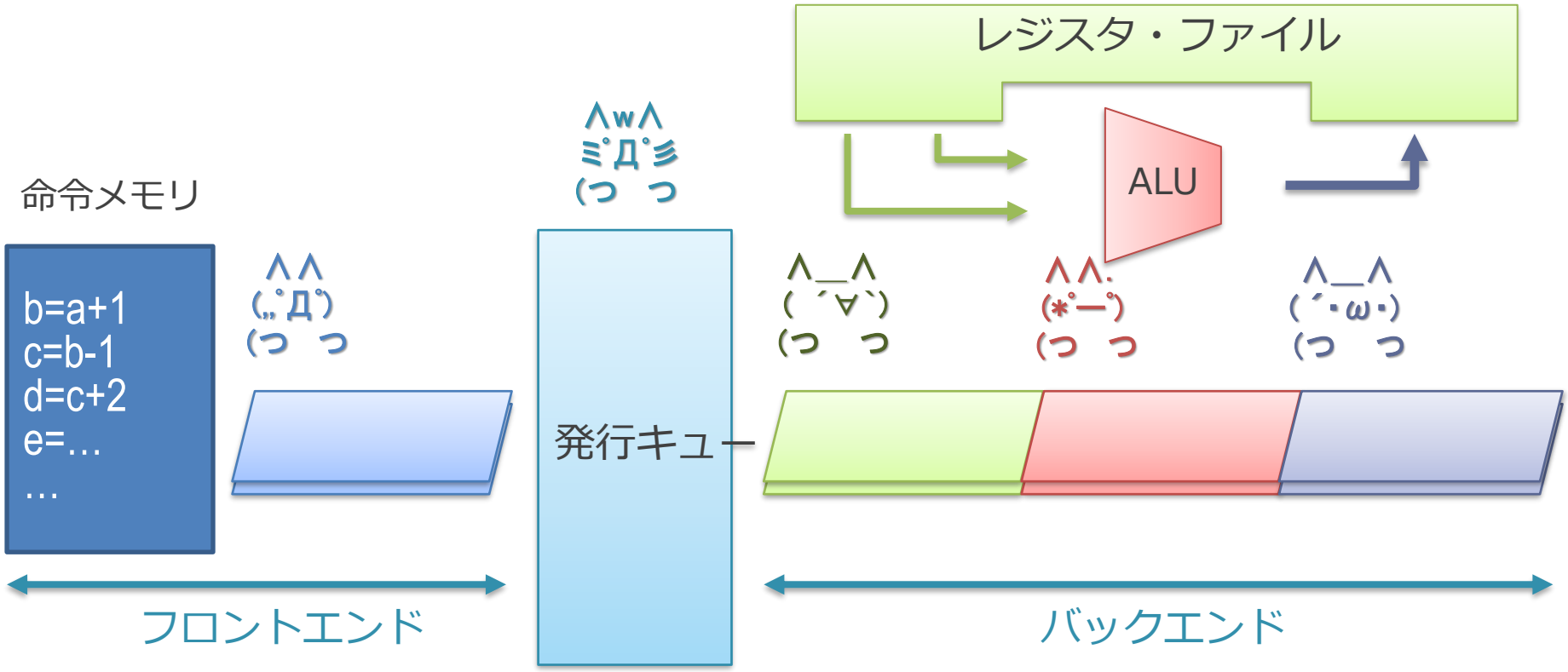
Out-of-order 実行

- Out-of-order 実行：
  - 動的に命令をスケジュールして実行すること
  - つまり「プログラム順 = in-order 」に実行しないこと
- Out-of-order スーパスカラ・プロセッサ
  - Out-of-order 実行を行うスーパスカラ・プロセッサのこと
  - 現在主流の高性能 CPU は，基本的にみなこのタイプ
    - PC やスマホ，サーバーは大体これ

Out-of-order 実行

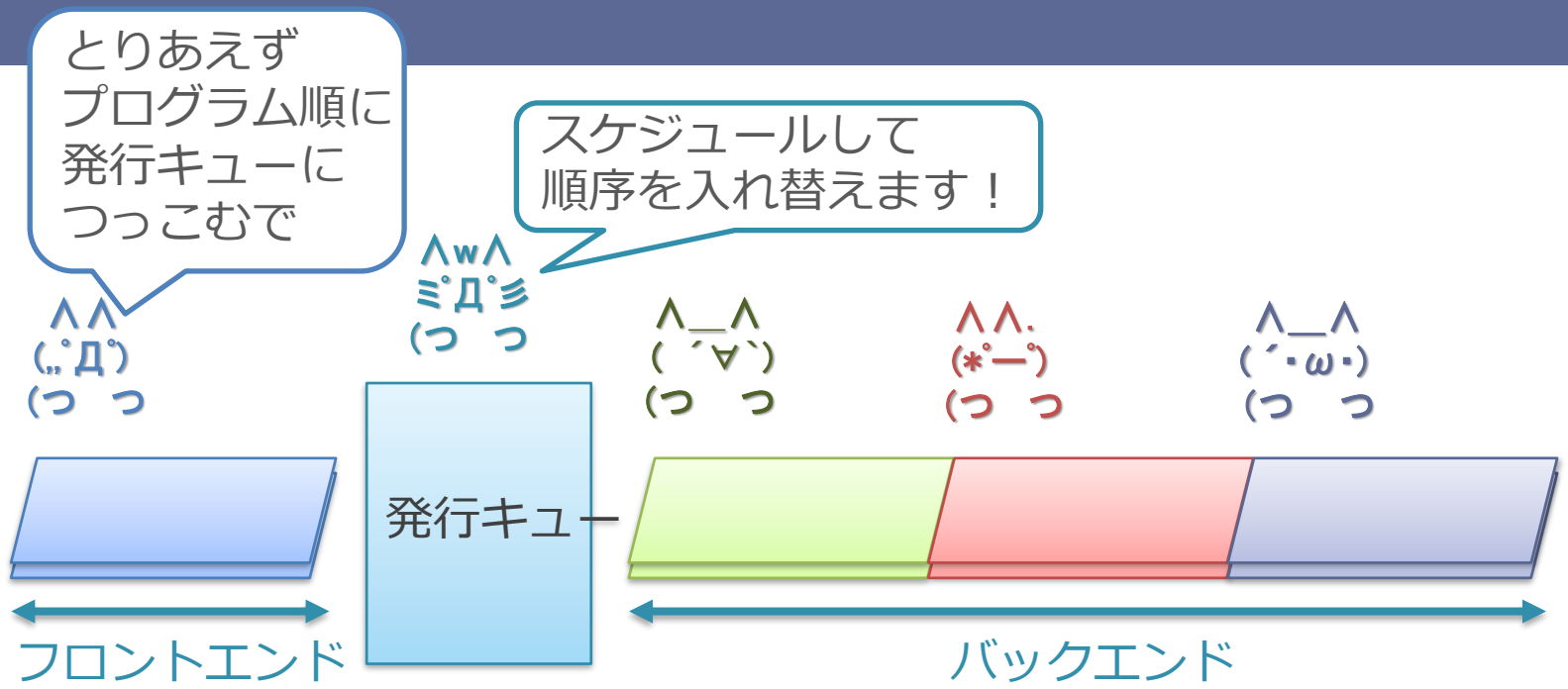
- スカラ or スーパスカラとは直行した概念
  - …ではあるが，普通は動的スケジューリングを行う CPU はスーパスカラ
- スカラで動的スケジューリングをやってもあまり意味がないから
  - Out-of-order 実行を行う機構をつける前に，まず in-order なままスーパスカラ化した方が良い
  - その方が，より少ない回路で性能があがる

Out-of-order スーパスカラ・プロセッサの構造



- 発行キューによって前後に分離された構造を持つ
  1. フロントエンド：　命令を供給
  2. 発行キュー：　命令の待ち合わせ
  3. バックエンド：　命令を実行

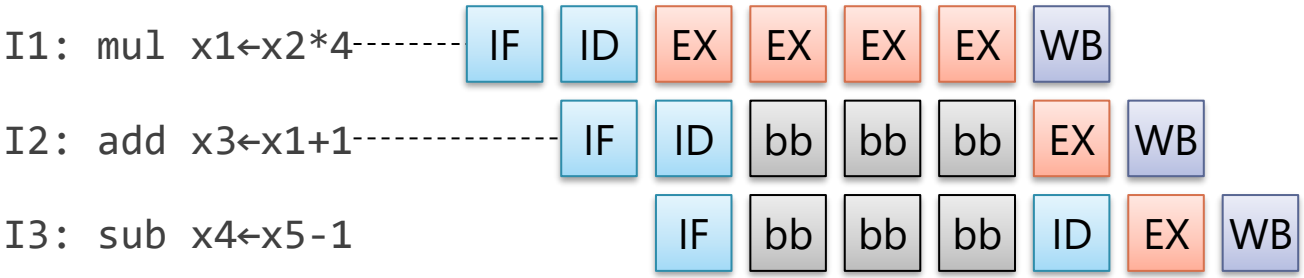
大ざっぱな動作



1. フロントエンドで命令をプログラムに順にフェッチ
2. 発行キューに投入
3. そのとき実行可能なものから順にバックエンドに命令を送信
4. レジスタを読んで演算器で実行し書き戻す

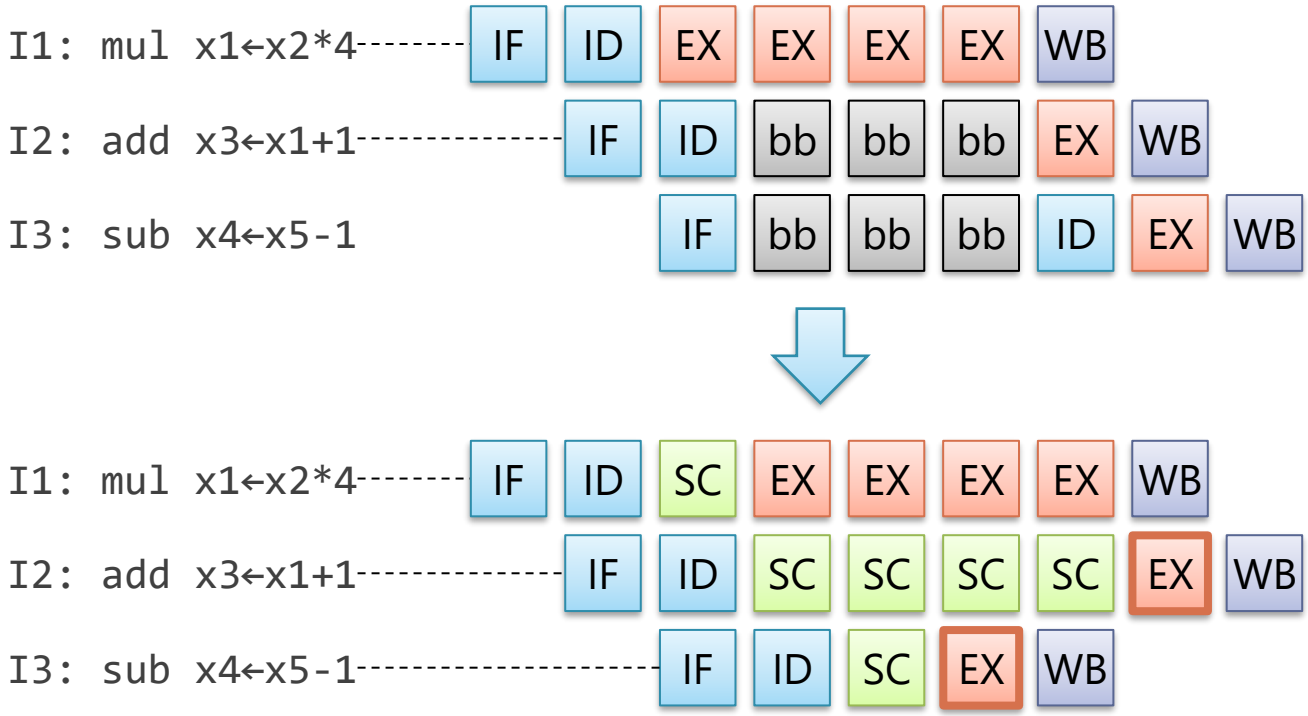
in-order 実行と out-of-order 実行の違い

- I1 の mul は４サイクルかかる
  - In-order 実行だと，I3 は I1 に依存していないが待たされる
  - プログラムに書かれた順に実行するため

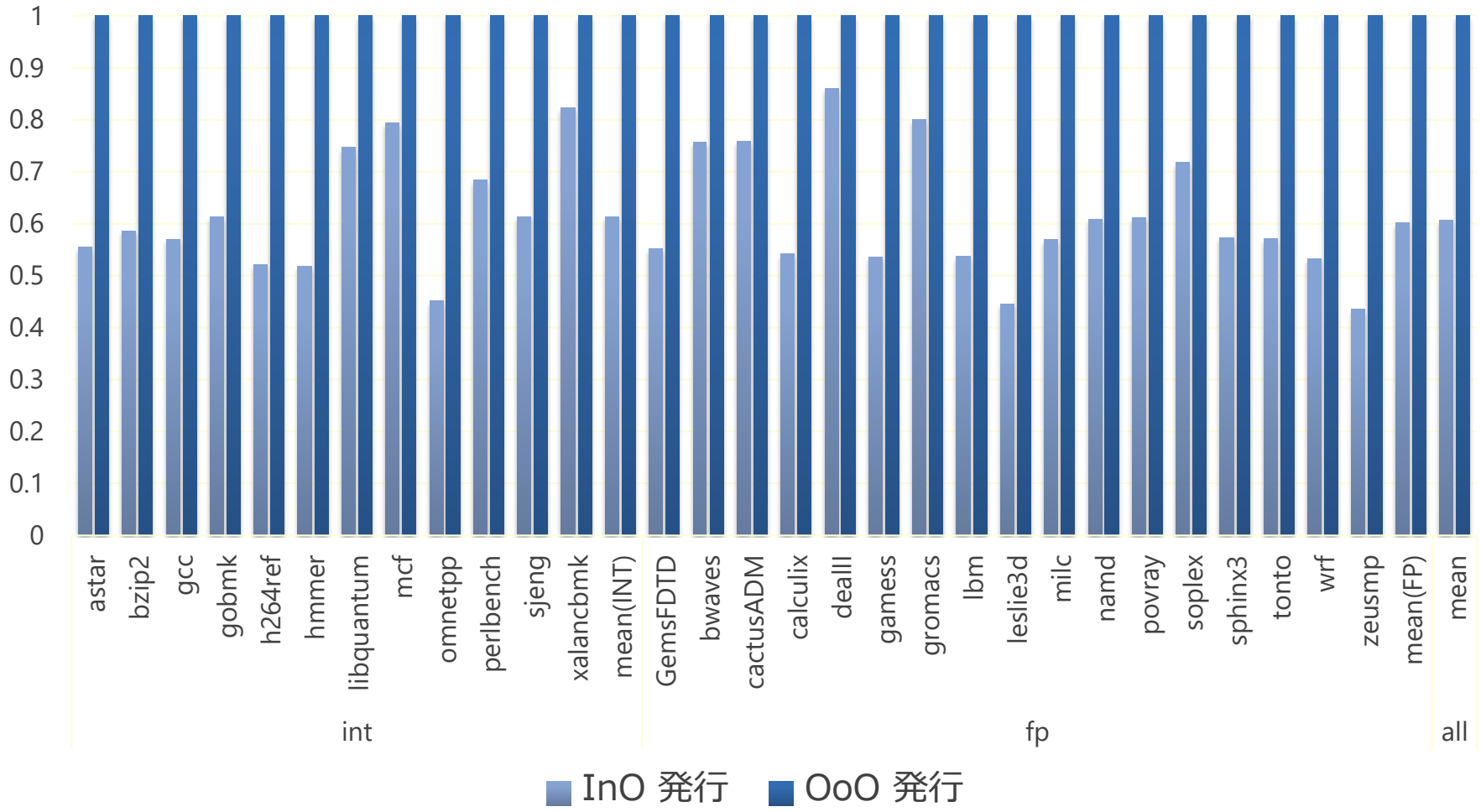


in-order 実行と out-of-order 実行の違い

- Out-of-order 実行だと，I3 が I2 を追い越して実行できる
  - 各命令はスケジュール（SC）に入って実行可能になるまで待つ
  - 発行キューは，その時実行可能なものから順に EX に命令を送る
    - x1 は計算中なので SC で待つ
    - x5 は既に結果が得られていたので，先に送信



In-order 実行と out-of-order 実行の性能
（SPEC CPU 2006 と呼ぶベンチマークをシミュレーションした結果より



- OoO 実行の CPU の性能で正規化
  - InO 実行の CPU の性能は，平均で OoO 実行の60%程度

余談：「スーパスカラ・プロセッサ」という言葉

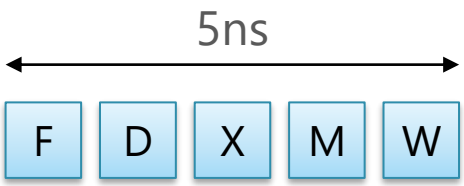
- 広義の「スーパスカラ・プロセッサ」
  - パイプラインや演算器を複数備え，複数の命令を同時に実行できるもの
- 単に「スーパスカラ・プロセッサ」と書いた場合：
  - 「out-of-order 実行を行うスーパスカラ・プロセッサ」の意味で使われることがある
  - 文脈による

まとめ

1. 命令の並列実行の基本
2. データ依存
3. 静的命令スケジューリング
4. 動的命令スケジューリング

課題 7

- 前提：
  - 各命令は以下の５つの処理を経て実行されるものとする
    - F：フェッチ，D：デコード，X：演算，
      M：メモリアクセス，W：書き戻し
  - これらの各処理には 1 nano second (ns) がかかるものとする
  - 演算しか行わずメモリアクセスを伴わない命令でも，必ず M を経るものとする
  - フォワーディングは常に行うものとする
  - 命令を実行するために「必要な時間」とは，最初の命令のフェッチ開始から最後の命令の書き戻しが終わるまでの時間とする
    - たとえば以下の１命令を実行するために「必要な時間」は 5ns



課題 7

- 前記の前提の下でスカラのシングル・サイクル・プロセッサを構成することを考える
- (1) その場合の最大動作周波数はいくつになるか述べよ
- (2) 以下の命令列を実行するのに必要な時間を計算せよ

```
sub x2←x3+x4
add x1←x2+x3
add x5←x6+x7
```

課題 7

- 前記の前提の下で，F,D,X,M,W の 5 ステージからなるパイプライン・プロセッサを構成することを考える
  ただし，in-order スカラ・プロセッサであるものとする

- (3) その場合の最大動作周波数はいくつになるか述べよ

- (4) 以下の命令列を実行するのに必要な時間を計算せよ
  sub x2←x3+x4
  add x1←x2+x3
  add x5←x6+x7

- (5) 以下の命令列を実行するのに必要な時間を計算せよ
  依存関係のために必要な場合のみパイプラインを適宜ストールして実行するものとせよ
  ld　x2←(x1)
  add x1←x2+x3
  ld　x2←(x3)
  add x5←x2+x7

課題 7

- (6) 以下の命令列を実行するのに必要な時間を計算せよ
  ここで mul は乗算命令であり X に 4 サイクルが必要である.
  依存関係のために必要な場合のみパイプラインを適宜ストールして実行するものとせよ

```
mul x2←x1+x4
add x1←x2+x3
add x5←x2+x7
```

- (7) 以下の命令列を実行するのに必要な時間を計算せよ
  ここで beq はオペランドが等しい時に分岐する分岐命令である
  プロセッサは分岐予測を行うものとし，beq が分岐予測ミスを起こして LABEL に飛んだあとにフラッシュされてやり直した場合を想定せよ

```
  li x1←1
  li x2←2
  beq x1==x2, LABEL
  add x1←x2+x3
LABEL:
  add x2←x3+x4
```

課題 7

- 各パイプライン・ステージを１文字としてテキスト・エディタ上や紙のノートに絵を描いて考えると良い
  - いちいち四角を描いたりすると，めんどくさい
- たとえばこんな：
  - ```
    F D X M W
      F D X M W
        F D X M W
    ```
    ```

提出方法

- 以下を提出：
  1. 課題 7：
     - 提出は Moodle の「課題 7」のところからお願いします
     - 紙に書いた場合は写真を撮ってアップロードしてください
  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

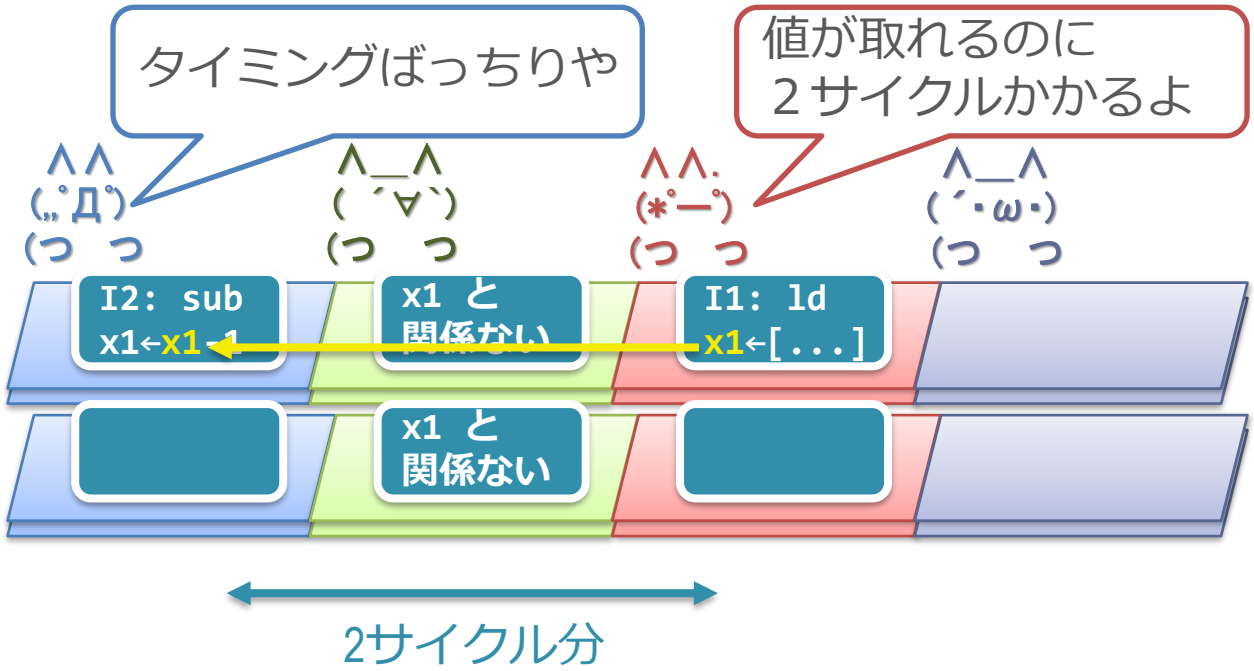
- 注意：
  - 課題の出来は，ある程度努力したあとがあれば良しです
    - 必ずしも正解していなくても良いです

付録

2.実行タイミングを仮定してスケジュールされている

- 複数サイクルかかる命令は，それに合わせてスケジュールされる
  - 「**M サイクル後**に結果が使用できる」前提でパイプラインが止まらないように事前に命令が並べてある

- 以下では，`I1: ld` の値が使えるタイミングで `I2` が実行できるよう両者を離してある



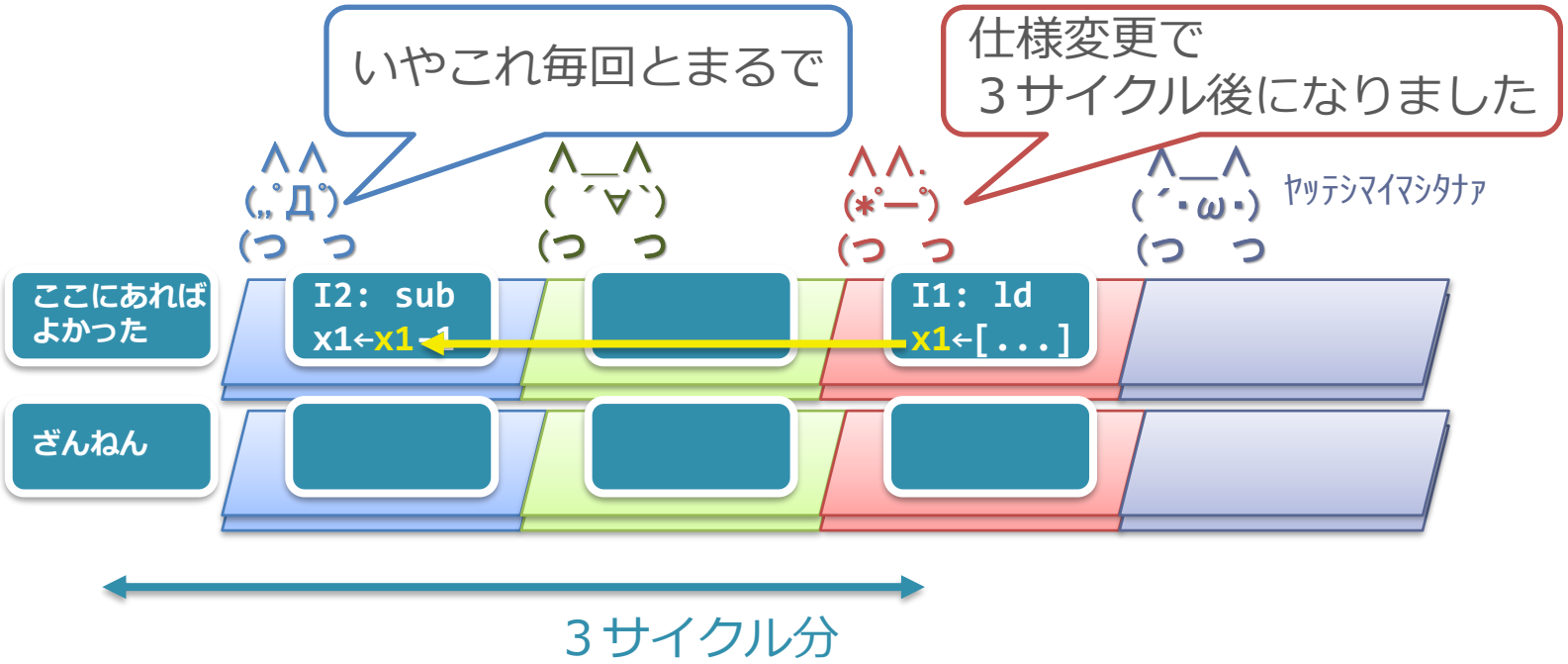
2. 実行タイミングを仮定してスケジュールされている

- M を変化させにくい
  1. 短くなる = 既存のコードは恩恵を受けられない
  2. 長くなる = 毎回バブルが発生して性能ががたおちする
     - 想定したタイミングに入力が揃わない

- たとえば，次世代の以下のような CPU 作る場合を考える：
  1. 乗算器を改良してレイテンシが短くなった
  2. キャッシュを倍にしてヒット率をあげたがレイテンシが多少伸びた

キャッシュのレイテンシが伸びた場合

- ld のレイテンシが 2 から 3 に変更となった場合
  - ２サイクルにジャストで合わせて I2 をスケジュールしておくと，毎回バブルが発生することに



実行タイミングを仮定してスケジュールされていることの他の問題

- そもそも実行時にレイテンシが動的に変化する場合は対応困難
  - キャッシュのヒットとミスが場合によってかわるようなロード
- コンパイラではあらかじめヒットかミスを仮定してスケジュール
  - プロファイラで事前に特性をとって，それに基づくことである程度緩和はできる

VLIW の例：Intel Itanium

- インテルと HP で作った VLIW プロセッサ
  - 2000 年代前半ぐらいまでは x86 からこれに移行しようとしていた
  - EPIC アーキテクチャと言われる命令セットを持つ
- これまで述べたような VLIW の問題を緩和するような機構を色々投入
  - 命令セットの互換性をとりながら同時実行幅を増やす
  - 分岐を跨いだロードの移動をハードで支援

Intel Itanium の性能

- しかし，x86 よりも全然性能がでなかった
  1. 静的スケジューリングの限界
  2. レイテンシを仮定したコード
  3. クロックが上げられなかった
     1. 2. に関連して，キャッシュ・アクセスのステージ数を増やしてクロックを上げることができない
     2. VLIW の問題緩和の機構のせいで返って複雑化

Intel Itanium の末路

1. 当時 32 ビットから 64 ビットへの移行の要求が高まっていた
   - 主にメモリ使用量を増やすため
     - 32 ビットのアドレスで表せるのは 4GB まで
   - Itanium はこのための 64 ビット CPU でもあった

2. インテルは互換 CPU の製造開発を許したくなかった
   - しかし既に与えたライセンスは取り消せない
   - 64 ビット世代で内容を刷新して今度は独占を目指した

3. AMD が独自に x86-64 を策定
   - Itanium がさっぱり性能でないので，MS が見切りをつけて Windows の x86-64 対応を開始

4. 後追いでインテルも x86-64 の CPU を開発
   - Itanium は一応製造されているが， 2021 年に最終出荷で終了

質問とか感想

質問とか感想

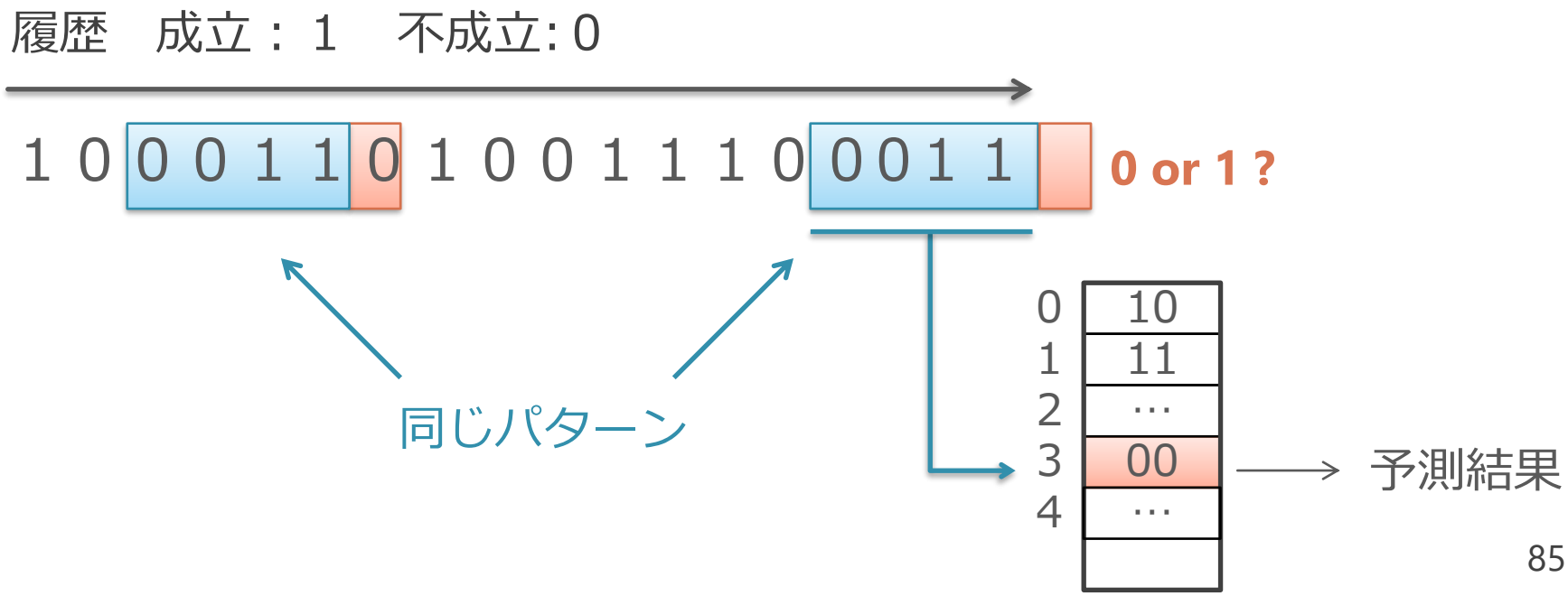
- immのところがいまいちわかっていません
- 課題6(3)についてですが、なぜ16だけそのままなんですか？
- 課題（３）のaddiのimmがよく分かりませんでした。rs1の２の補数を求めればよいということにして解いてみました。
- imm[11:0]の意味を教えてください。

質問とか感想

- 制御ハザードの解消方法の一つに分岐予測が挙げられていましたが、分岐予測は何をもとにして予測しているのか気になりました。
- 分岐予測の具体的なアルゴリズムにはどのような種類があり、それぞれの精度にはどのような違いがあるのでしょうか。また、これらのアルゴリズムが実際にどのように実装され、どのように動作するのかについても詳しく教えていただけると嬉しいです。
  - 命令アドレスごとに前回分岐した方向を憶えておいて，次もそっちに行くなどの方法を使います

「履歴（history）」を用いた予測器

- 基本的なアイデア：分岐方向の履歴をビット列で表す
  - 履歴のビット列をインデクスとしてテーブルにアクセス
    - テーブル自体は，２ビットカウンタ
  - 直前の履歴でテーブルをひく
    - 直前に同じパターンがくると，同じエントリにアクセス
    - 二進数で 0011 = 表の3番目のエントリ



質問とか感想

- 分岐予測のところでふと気になったのですが、ストールしている間命令を実行している猫が減るので動きが軽くなったりするのでしょうか。それともほとんど変化はないのでしょうか。

質問とか感想

- 一見分岐予測は無駄な行為だと思いましたが、何もせずに待っているよりはやり得なのだと納得しました。ただ、分岐予測をすることによって無駄にエネルギーを消費してしまうのではないかとも思いました。

質問とか感想

- 第5回の課題についてです。NORを作ってからNOTで反転するのではなく、初めからORを作ったのですが、それでも良いのでしょうか？

質問とか感想

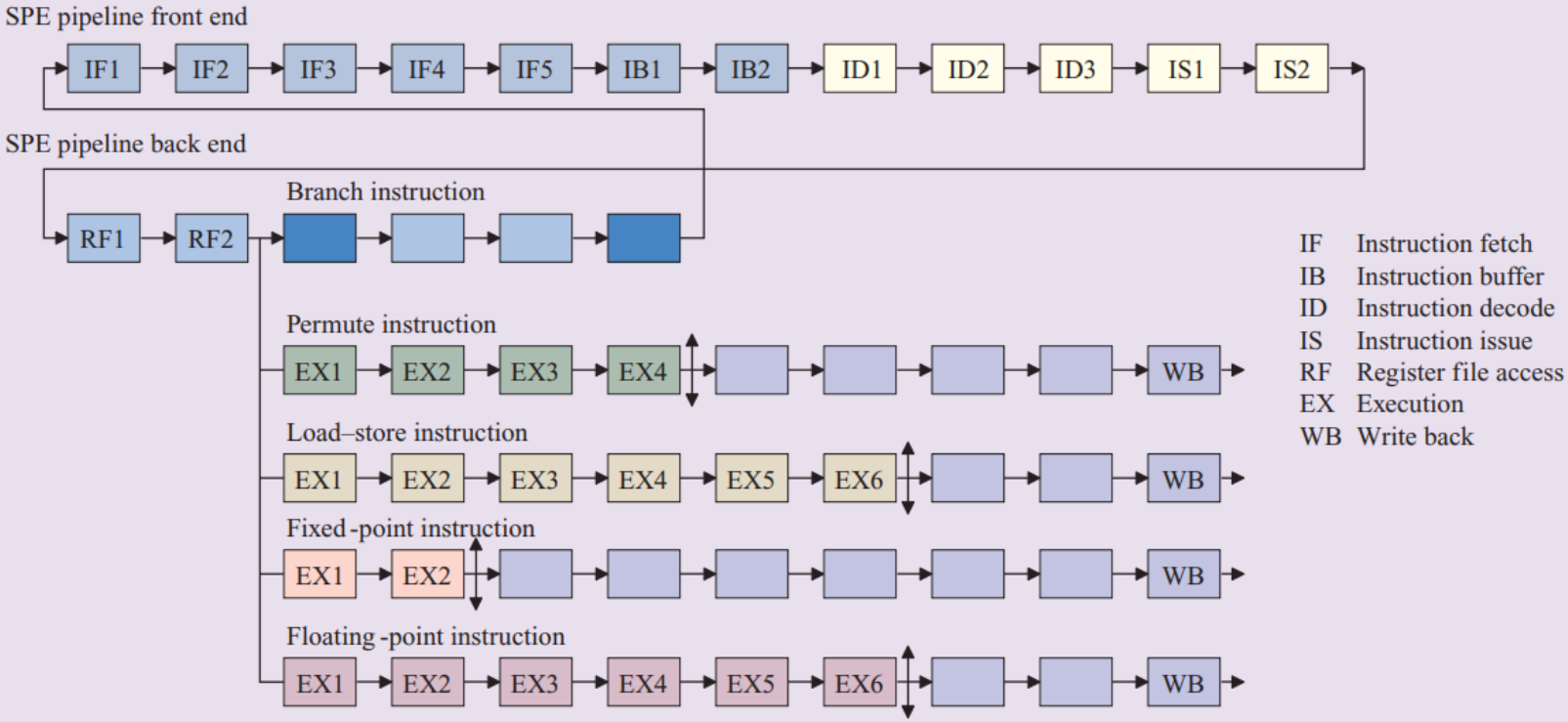
- 第3回目の内容なのですが、p.49のXORとORの違いがわかりません。計算している内容が同じに見えるのですが、違うのでしょうか？
  - すいません，誤植でした．訂正しました．

質問とか感想

- 74-76ページにかけての図の見方がいまいち分かりませんでした。

Sony/IBM/東芝 Cell (SPE)

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



質問とか感想

- 分岐予測のペナルティのロスが大きいときはストールか、遅延スロットで対応するというふうに対応を変えられるのでしょうか。

質問とか感想

- else ifで分岐がいくつもあったり予測の精度が低いときは、無駄になってしまう処理が多いと思うので、どのような仕組みで予測をしているのか気になりました

質問とか感想

- 私は聞きやすくてzoomの授業とても好きです。これから教室の授業をzoomでも配信してもらえたら嬉しいです。

質問とか感想

- 難しくて興味をなかなか持てないのですが、皆さんが感想でおっしゃているように、猫ちゃんが可愛いおかげで、興味を持てます。今回は特に、コンピューターが生きてるように思えて、身近に感じました。

質問とか感想

- シスプロでもスタック領域などの話を丁度習いました。

質問とか感想

- ハザード対策として当たりをつけてしまうという方法が予想外で驚きました。確かに、どうせ待ち時間がかかるのなら当たりが外れてもあまり問題がないし、当たっていれば処理が素早くできるのですね。

質問とか感想

- データ・ハザードの解消方法として、ストールは単純に止めるだけだったのに対し、遅延スロットは考え方が複雑でよくわからなくなってしまいました。古い値を読んでそういうことにすると本来の値と異なってしまうのをいつ修正するのかをもう一度説明していただきたいです。

質問とか感想

- 今回のスライドのp29やp42のような図を理解するのに時間がかかるので頑張ってわかるようになりたいです。

質問とか感想

- 構造ハザードと非構造ハザードについておおまかに分かった。マルチタスクが苦手なので、自分も頭の中で複数のことを同時に処理出来たらいいのになぁと思った。工場の従業員に愛着が湧いてきました。

質問とか感想

- 投機予測について、物質的な作業、例えば料理などにおいては廃棄のマイナスが大きく、実行するのは難しそうなので、CPU上だからできることなのかなと思いました。
  データハザードのように、直前の命令で更新された結果を使いたいが、その処理が間に合わなかった場合、更新前のものを読み取ってしまうことはあるのでしょうか。

質問とか感想

- 課題をやっていて再認識したこととして、今まで0と1の羅列に一体どのようにして意味を持たせているのかと疑問に思っていましたが、RISC-V の 基本整数命令の表を見てあらかじめ命令の型のようなものがあると分かり腑に落ちた感覚があります。

質問とか感想

- 大規模な高性能プロセッサの場合の分岐予測ペナルティの絶望感が凄まじくて一周回って笑ってしまいました。やってしまいましたなぁ……

質問とか感想

- 取り消しが続くと速度が落ちてしまうなら分岐予測後の命令をあまり繋げないことが速いプログラムを作る秘訣なのではないかと思った。

質問とか感想

- 今OCamlでもっと早いプログラムを書くために頑張っているのでこのように短縮できる技がOCamlにも使えればいいなと思った。

質問とか感想

- 一つ質問なのですが、試験日はいつになりそうでしょうか？バイト先の夏期講習の日程調査を提出しなくてはならず。。。教えていただきたいです