先進計算機構成論 05

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

質問と回答とか

- ARMのレジスターの値を一気にメモリに書き込むのコマンドを紹介されましたが、具体的にどの仕組みでしょうか？dual-portメモリを使っても同時に書き込むアドレスが２つですので、書き込みが複数のクロックに実行しているでしょうか？

ARM（32ビット）の Load/Store Multiple (LDM/STM) 命令

ビットマスクで指定した最大16個のレジスタへのロードストアを行う
関数呼び出し/復帰の際の，レジスタの保存や復帰でよく使われる

A3.12 Load and Store Multiple instructions

Load Multiple instructions load a subset, or possibly all, of the general-purpose registers from memory.

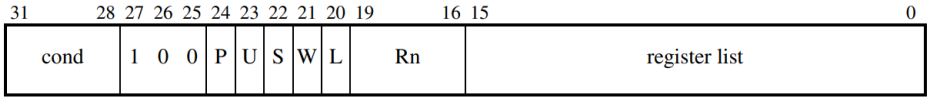
Store Multiple instructions store a subset, or possibly all, of the general-purpose registers to memory.

Load and Store Multiple instructions have a single instruction format:

```
LDM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
STM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
```

where:

```
<addressing_mode> = IA | IB | DA | DB | FD | FA | ED | EA
```



register list The list of <registers> has one bit for each general-purpose register. Bit 0 is for R0, and bit 15 is for R15 (the PC).

The register syntax list is an opening bracket, followed by a comma-separated list of registers, followed by a closing bracket. A sequence of consecutive registers can be specified by separating the first and last registers in the range with a minus sign.

P, U, and W bits These distinguish between the different types of addressing mode (see *Addressing Mode 4 - Load and Store Multiple* on page A5-41).

S bit For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR after all the registers have been loaded. For all STMs, and LDMs that do not load the PC, it indicates that when the processor is in a privileged mode, the User mode banked registers are transferred and not the registers of the current mode.

L bit This distinguishes between a Load (L==1) and a Store (L==0) instruction.

Rn This specifies the base register used by the addressing mode.

A3.12.1 Examples

```
STMFD    R13!, {R0 - R12, LR}
LDMFD    R13!, {R0 - R12, PC}
LDMIA    R0, {R5 - R8}
STMDA    R1!, {R2, R5, R7 - R9, R11}
```

- ARM Architecture Reference Manual より

質問と回答とか

- AArch64 が出たとき、 32-bit のとはかなり違って驚かされた覚えがあるんですが、この変化による貢献は実際どれほどあるものなのでしょうか。

質問と回答とか

- RISC-Vの命令幅が4バイト=32ビットとのことですが，近年の命令セットは基本64ビットなのかと思っていました．Windowsの32ビット，64ビットはまた別の話でしょうか.

質問と回答とか

- CISCの複雑な命令を分解したマイクロ命令は、RISC-Vの命令(ニーモニック)と同じような感じなのでしょうか。"

質問と回答とか

- ハザードの解決におけるマイクロ命令などは気合な気がしました。マイクロ命令への変換は種類としてどのくらいの量が必要になるのでしょうか。

- 解析結果をまとめているサイトも
  https://uops.info/table.html
  下の例は，x86-64 の CMOVBE 命令の Alder Lake 大きいコアでの分解の説明

CMOVBE (R64, M64)

Summary:	"Conditional Move"
Reference:	https://www.felixcloutier.com/x86/CMOVcc.html
Extension:	BASE
Category:	CMOV
ISA-Set:	CMOV
CPL:	3
iform:	CMOVBE_GPRv_MEMv
iclass:	CMOVBE
ASM:	CMOVBE

Operands

- Operand 1 (w): Register (RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15)
- Operand 2 (r): Memory
- Operand 3 (r, suppressed): Flags (CF: r, ZF: r)

Available performance data

- Alder Lake-P
- Alder Lake-E
- Rocket Lake
- Tiger Lake

Alder Lake-P

- Measurements
  - Latencies
    - Latency operand 1 → 1: 1
    - Latency operand 2 → 1 (address, base register): 6
    - Latency operand 2 → 1 (address, index register): 6
    - Latency operand 2 → 1 (memory): ≤6
    - Latency operand 3 → 1: 2
  - Throughput
    - Computed from the port usage: 1.00
    - Measured (loop): 1.02 (if an indexed addressing mode is used: 1.00)
    - Measured (unrolled): 1.00
  - Number of µops
    - Executed: 3
    - Retire slots: 2 (if an indexed addressing mode is used: 3)
    - Decoded (MITE): 2
    - Microcode Sequencer (MS): 0
    - Requires the complex decoder (4 other instructions can be decoded with simple decoders in the same cycle)
  - Port usage: 2*p06+1*p23A

質問と回答とか

- マイクロ命令にすると構造ハザードを考えなくて済むという利点を挙げていましたが、そう設計したのだから当たり前では？と少し混乱してしまいました。

  - ◇ 他の方法でも構造ハザードは解決できますが，
    他の方法と比べて，プロセッサ全体への工夫が必要なく
    デコード部分での分解のみ考えれば良いので楽という意図でした.

質問と回答とか

- ステージの切り方で非同期処理が紹介されていましたが、ステージ間の間隔をそろえるために具体的にどのような方法をとっていたのでしょうか。
- パイプラインステージの分割は自動で最適化できるのでしょうか。人間ががんばるしかないのでしょうか。

質問と回答とか

- 少しわからなくなってしまったのですが，マイクロ命令書き換え可能というのはデコード回路を書き換えているのでしょうか？FPGAみたいなのがあるということでしょうか？(根本的に何かをミスリードしているような気がします...)

  - 元の命令 → マイクロ命令 のハッシュ表のようなデータ（テーブル）を持っており，それが書き換えられるということです
    どの命令がデコード回路で分解され，どの命令が表を使うかも切り替えられるようになっていると思います

質問と回答とか

- また、マイクロ命令への変換について授業中によく理解できなかったので、説明していただけると幸いです。
- マイクロ命令の分解の難しさがいまいち理解できませんでした。

質問と回答とか

- デコードにかかる時間はどのくらい？

質問と回答とか

- インテルのコアにバグが多いとのことでしたが、Intelの品質の低さは何に起因しているのでしょうか？

  - インテルのコアにたくさんバグがあるのはそうだが，他社のコアも同様．本当に致命的なバグはそこまで多くない．正しく作る事はとても難しいという事が言いたかったです．

質問と回答とか

- マイクロ命令にデコードするというのはひたすら場合分けして考えるとのことでしたが、それ（複雑な命令の組み合わせ）を人間が考えているからエラッタが多く発生しているのでしょうか。
- 人間が完全に場合分けして記述してあげなくても、ある程度のルールを記述して状況に応じてコンピュータが適切に資源を分配する、というのは実現不可能なのでしょうか。

質問と回答とか

- 昔IntelのCPUで浮動小数点除算に使うテーブルが間違っていて、若干ずれた答えが出てくるというバグが大問題になった記憶があります。Intelが出荷した全CPUを無償交換するということになりましたが、結局希望者はそんなに多くなかったらしいですね。
  その経験もあってバグっていても後でパッチを当てられるような構造にしたのでしょうかね？

質問と回答とか

- 確かにCISCだと構造ハザードが置きやすそうなので、マイクロコードに置き換えるというのも腑に落ちました(じゃあ最初からRISCにすれば？というのはナンセンスなんでしょうか...?)
- CPU の動作を後から書き換えられる仕組みは頭がいいなと思うと同時に、細かい命令に分割するくらいなら最初からそういう命令のアーキテクチャだったらいいのにと思います。（互換性.....）
  - ◇ 最近は一周まわって CISC を分解するのが良いみたいなこともあります
    （メモリ上では表現がコンパクトだから等

質問と回答とか

- マイクロ命令の分解に結構なエネルギーが割かれていることを知り、構造ハザードが具体的に及ぼす影響が感じられました。それと同時に、上のレイヤーではハザードを意識しなくてもハザードを解消しながら命令を並べてくれるのはありがたいな、とも思いました。

質問と回答とか

- パイプラインの各ステージをより細分化して「EX1」「EX2」のようになっているものをどこかで見た記憶があるのですが、実際のCPUではこのような細分化は行われているのでしょうか
- 実際のプログラムでどれくらいパイプライン化で効率よく実行できるのかが気になりました。

質問と回答とか

- クロック周波数が頭打ちになる理由の一つには、パイプラインの「実行」ステージでの命令実行が間に合わなくなるという面もあるのだろうか？"

質問と回答とか

- 機械学習のライブラリーなどでpipelineという機能が実装されている場合がありますが，その中身は講義の中のpipelineと構造が似ているのか気になりました.
  - ◇ 考え方は同じだと思います

質問と回答とか

- （聞き逃したかもしれませんが）「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」ということでしたが、具体的にはどのようなステージ・どのような命令で遅延が大きくなる傾向があるのでしょうか。

  - 基本的には加算などの演算ステージと，後半で扱う out-of-order 実行のためのスケジューリングステージです.
    これらはパイプライン化が性能低下に直結するためです.

質問と回答とか

- 1サークルになった相互接続的な回路に命令の実行（IF,ID,EX,MEM,WB）に、回路の途中で始められますか。

質問と回答とか

- メモリの確保などの最適化をするときは具体的にどのようなプロセスを踏んでやっているのでしょうか.
  - ◇ 「メモリの確保などの最適化」がさすものがわからないので，具体例を書いてみて欲しい

質問と回答とか

- 曖昧な質問で申し訳ありませんが、広義でのコンピュータというもの（量子コンピュータ、さらにその先）は今後どういったものになっていくのか、先生の意見をお聞きしたいです。

前回の内容

1. 命令パイプライン
2. 構造ハザード：ハード資源の不足に起因
3. 構造ハザードの解決方法
    1. ハードウェアの増強
    2. 時分割処理
    3. マイクロ命令への変換

今日の内容

1. 非構造ハザード
2. 命令パイプラインと性能
3. 分岐予測（前編）

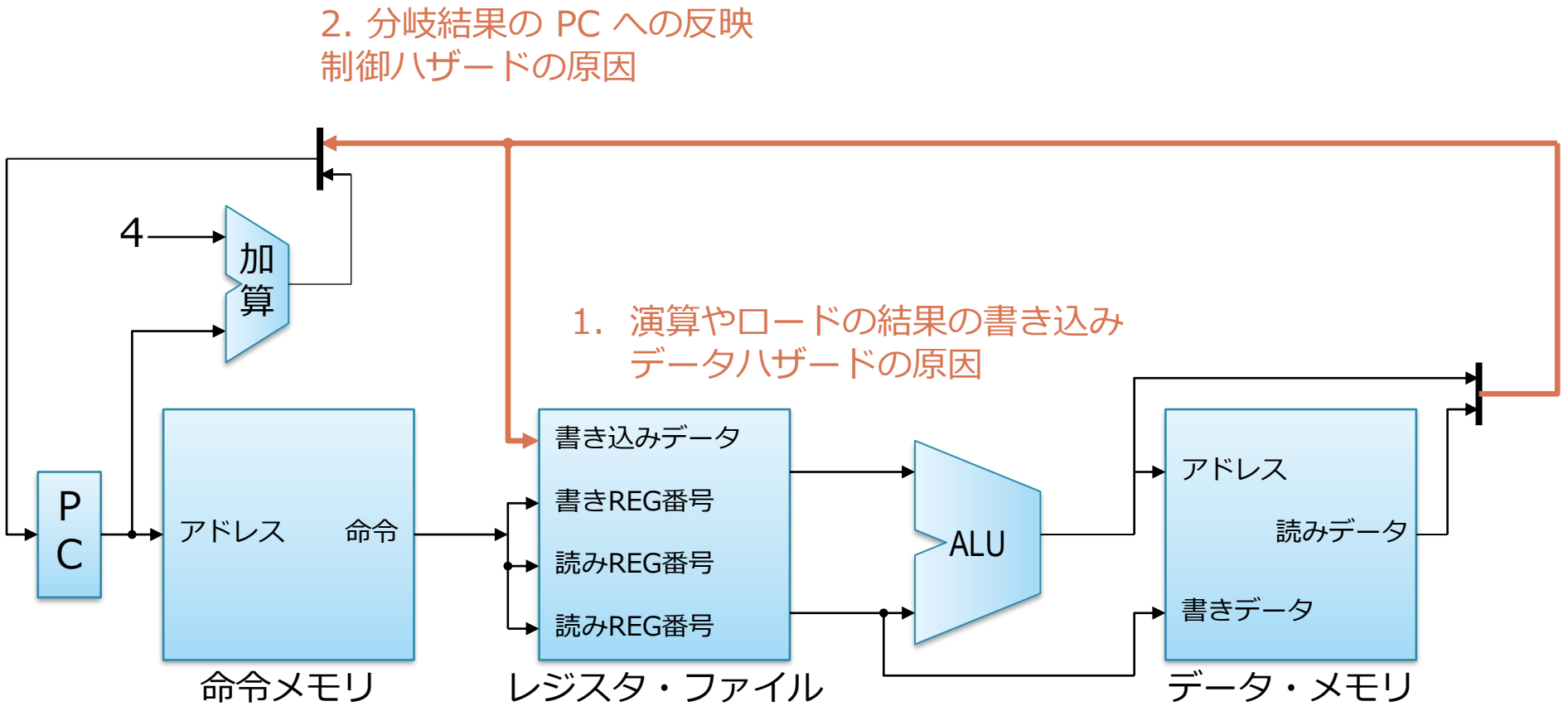
非構造ハザード

非構造ハザード

1. 構造ハザード：ハード資源の不足に起因（前回講義）

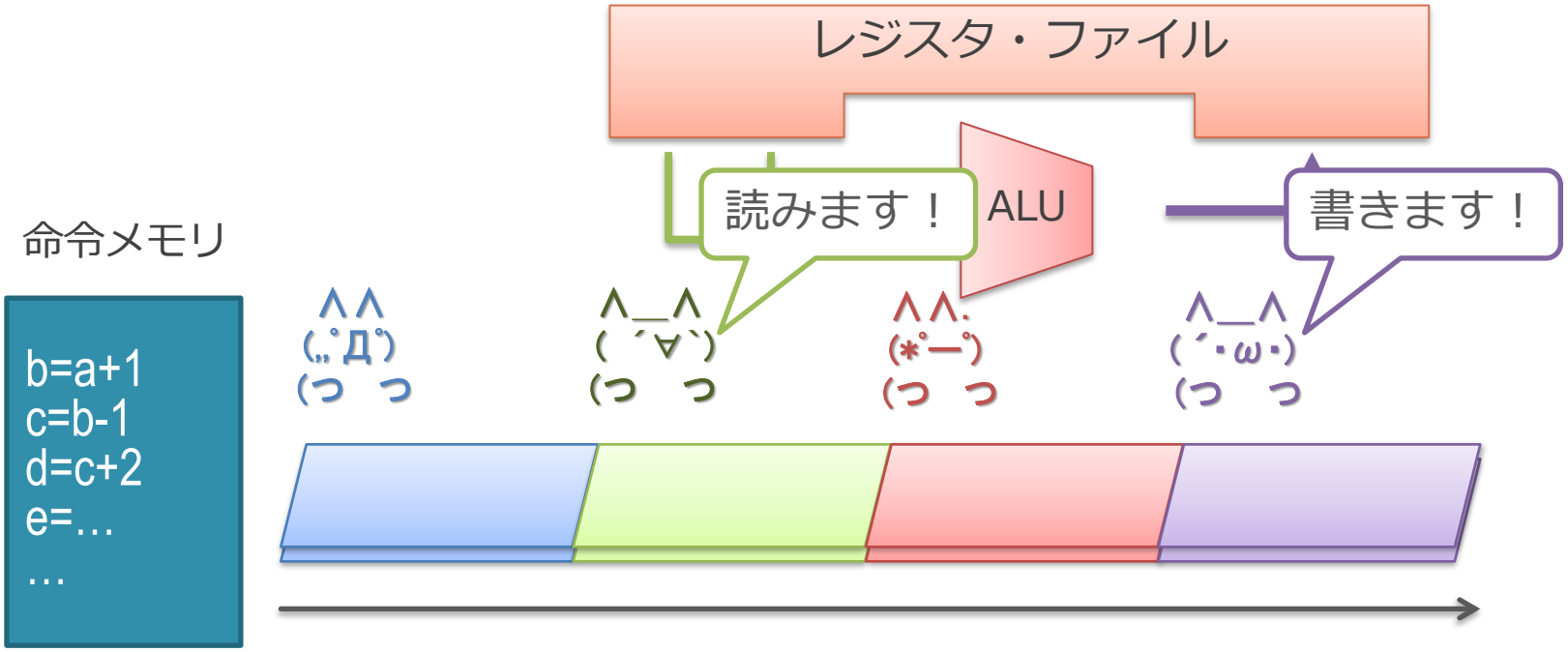
2. **非構造ハザード：バックエッジに由来**
   a. データ・ハザード
   b. 制御ハザード

バックエッジとは：逆方向（右から左）にいく信号



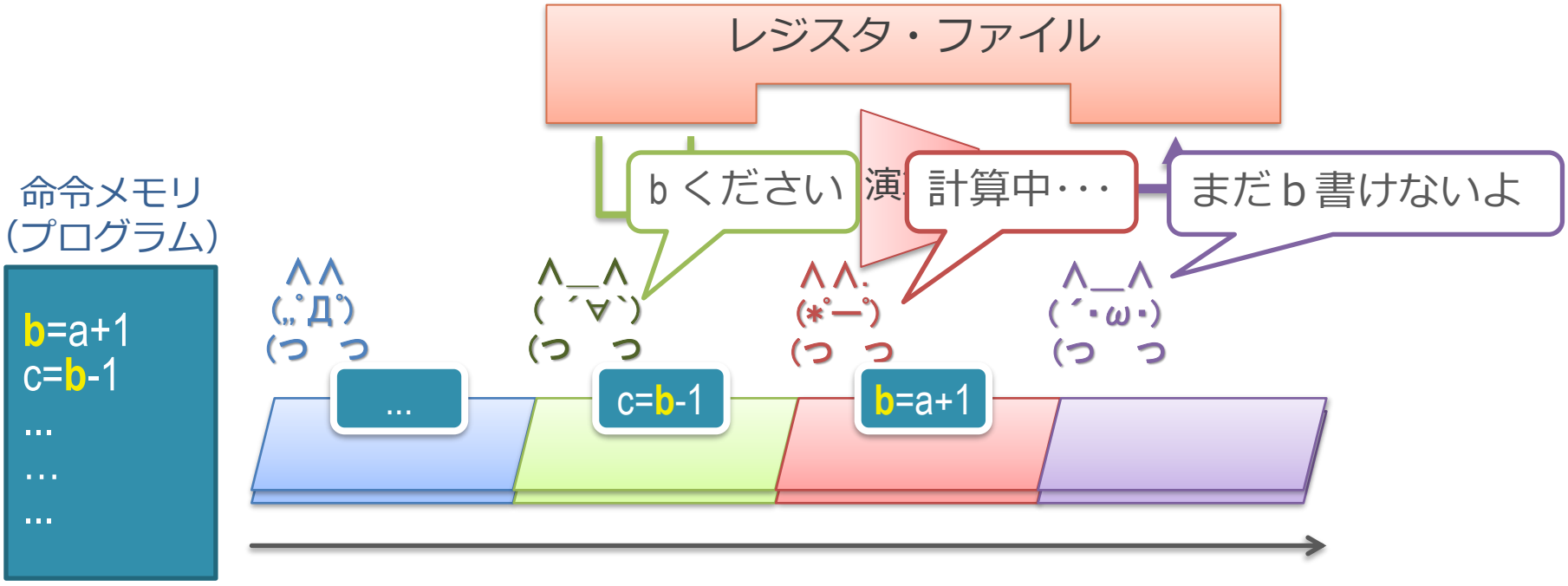
- バックエッジがあるため，命令を単純に流せない場合がある
  - ◇ 工場のラインのように，一方向に流せない

データ・ハザード



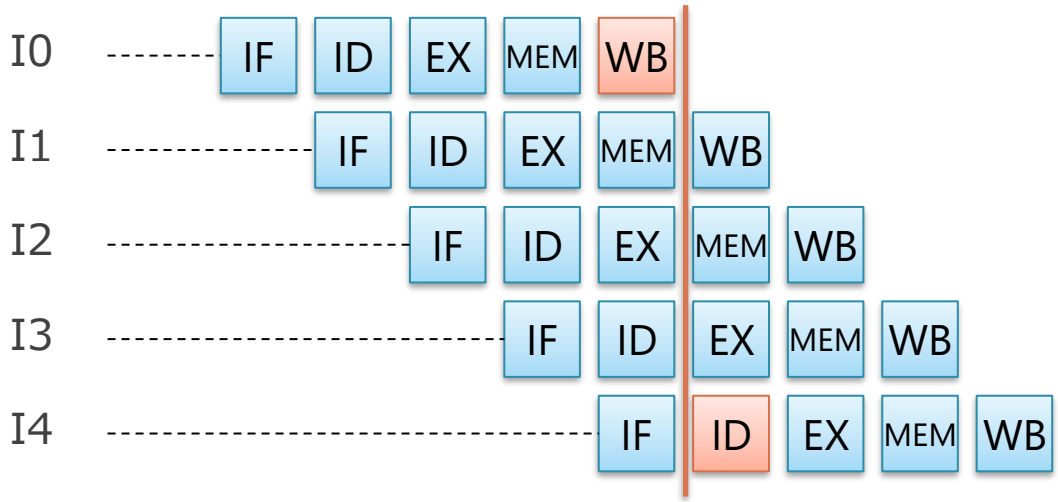
- レジスタ・ファイルへのアクセス
  - ◇ 演算の入力は(´∀`) の人がレジスタ・ファイルから読み出す
  - ◇ 演算の結果は(´･ω･) の人がレジスタ・ファイルに書き込む

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

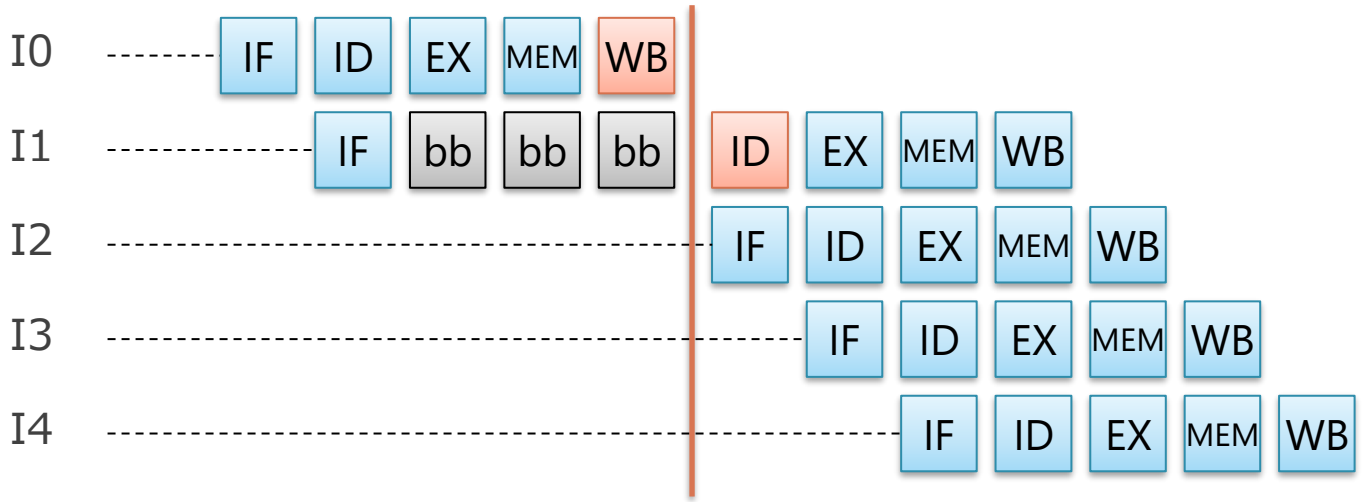


- I0 の WB が終わるまで，その結果はレジスタに書き込まれない
  - ◇ I4 までは，その値がレジスタから得られない
  - ◇ ID ステージでレジスタを読むため

データ・ハザードの解消方法

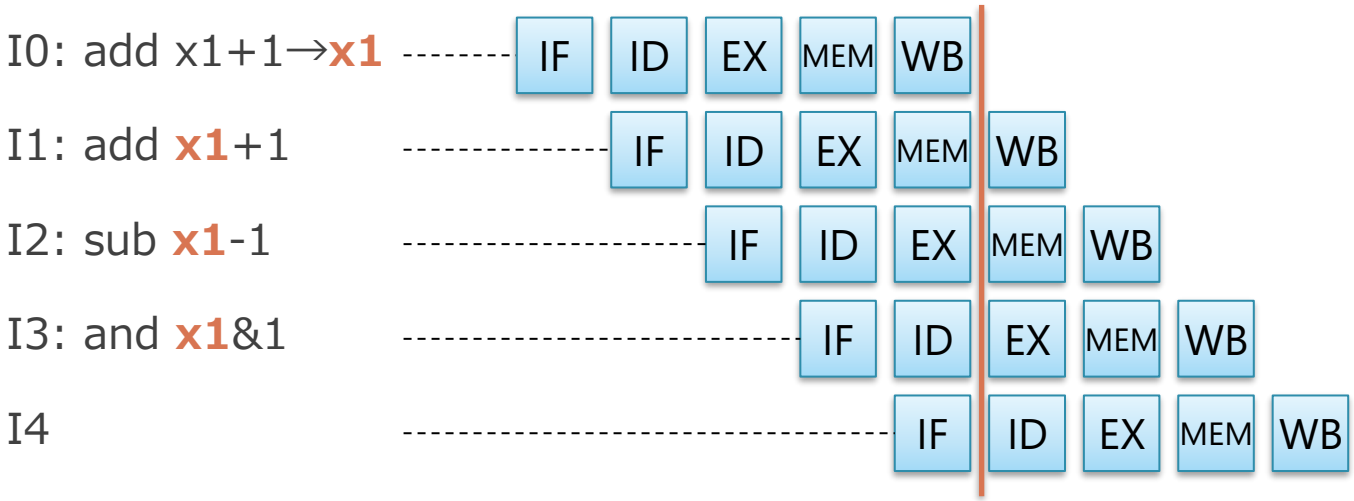
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. フォワーディング
  4. マルチスレッディング

1．ストールさせる



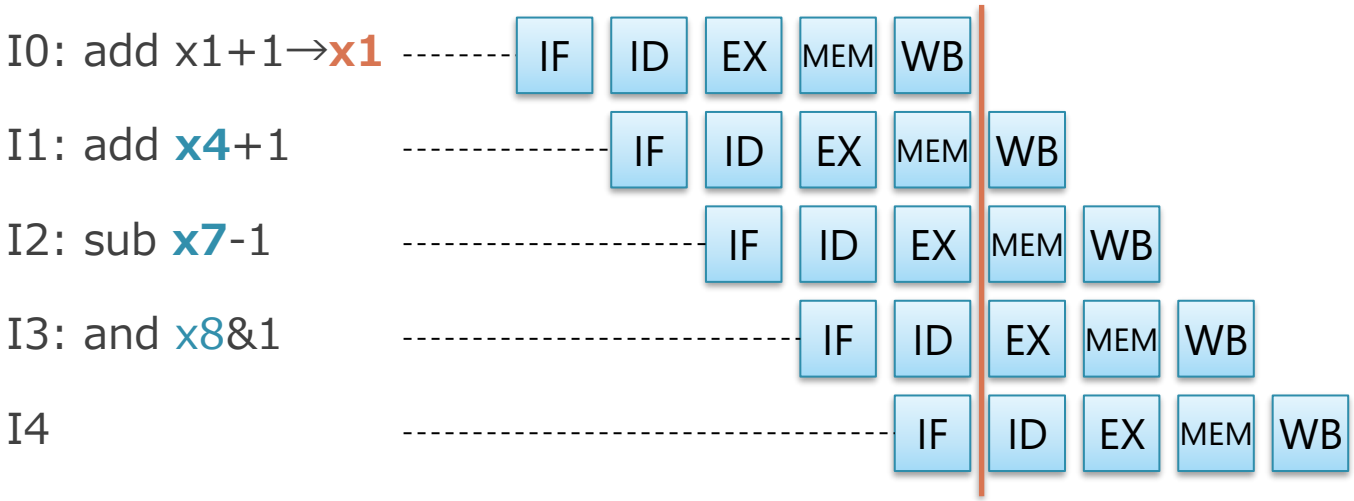
- I0 の WB が終わるまで，後続の命令を遅らせる
  - ◇ I1 の ID が，I0 の WB の右にくるまでストール
  - ◇ I1 は I0 の結果を使える
- 欠点：プログラムの実行がとても遅くなる

2. 遅延スロット（なにもしない）



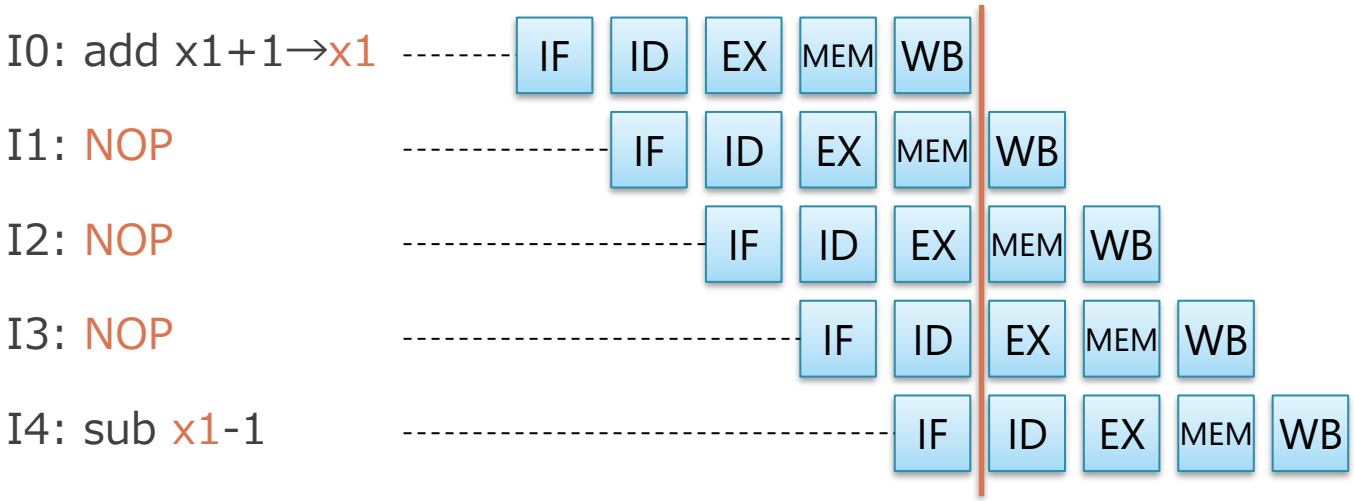
- 特になにも対策せず，
  「ある命令の結果は，数命令先まで見えない」と言う仕様にする
  - ◇ 上の例だと I1，I2，I3 は，I0 の結果は見えない
  - ◇ I1，I2，I3 には，I0 で add する前の値が見え続ける

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
  - ◇ この部分を「遅延スロット」と呼ぶ
    - □ この場合，遅延スロットが 3 命令分ある
  - ◇ 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



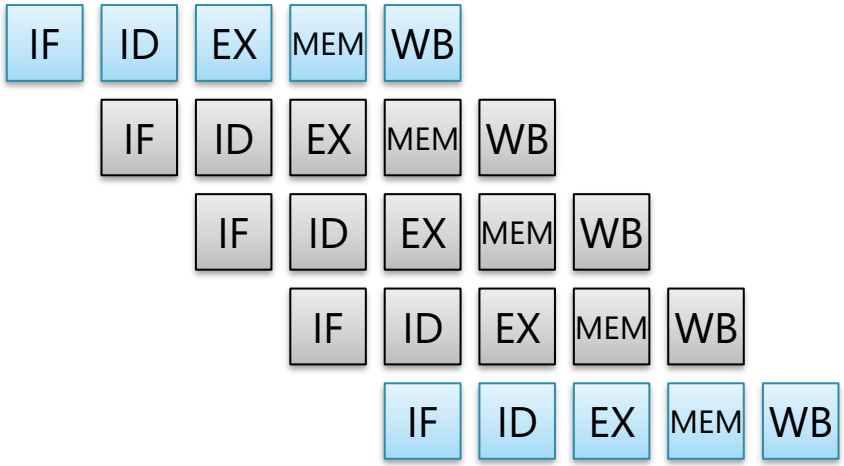
- もしそのような命令がない場合，
  - ◇ NOP（No Operation）と呼ぶ何もしない命令をいれる
  - ◇ これもコンパイル時にいれておくる必要がある
- 上の例は，x1 に 1 を足した結果を使う以外の処理がなかった場合

遅延スロットの利点

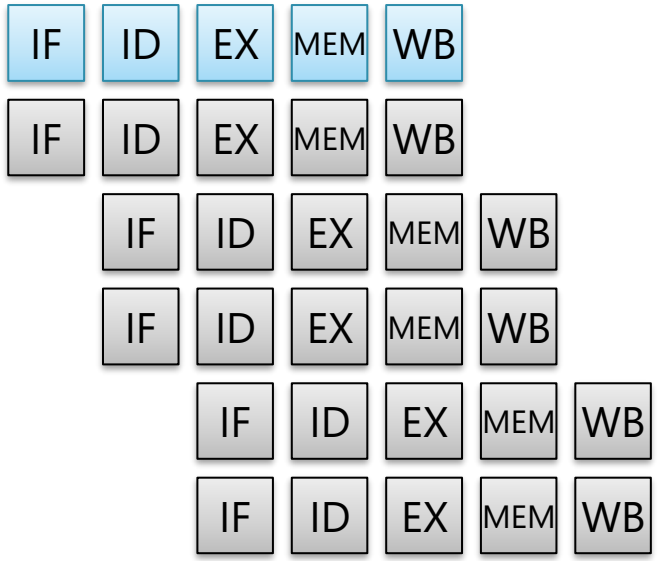
- 利点：
  - ◇ なにもしないので，ハードは最も単純
  - ◇ 並列にできる命令があれば，性能も下がらない

遅延スロットの欠点

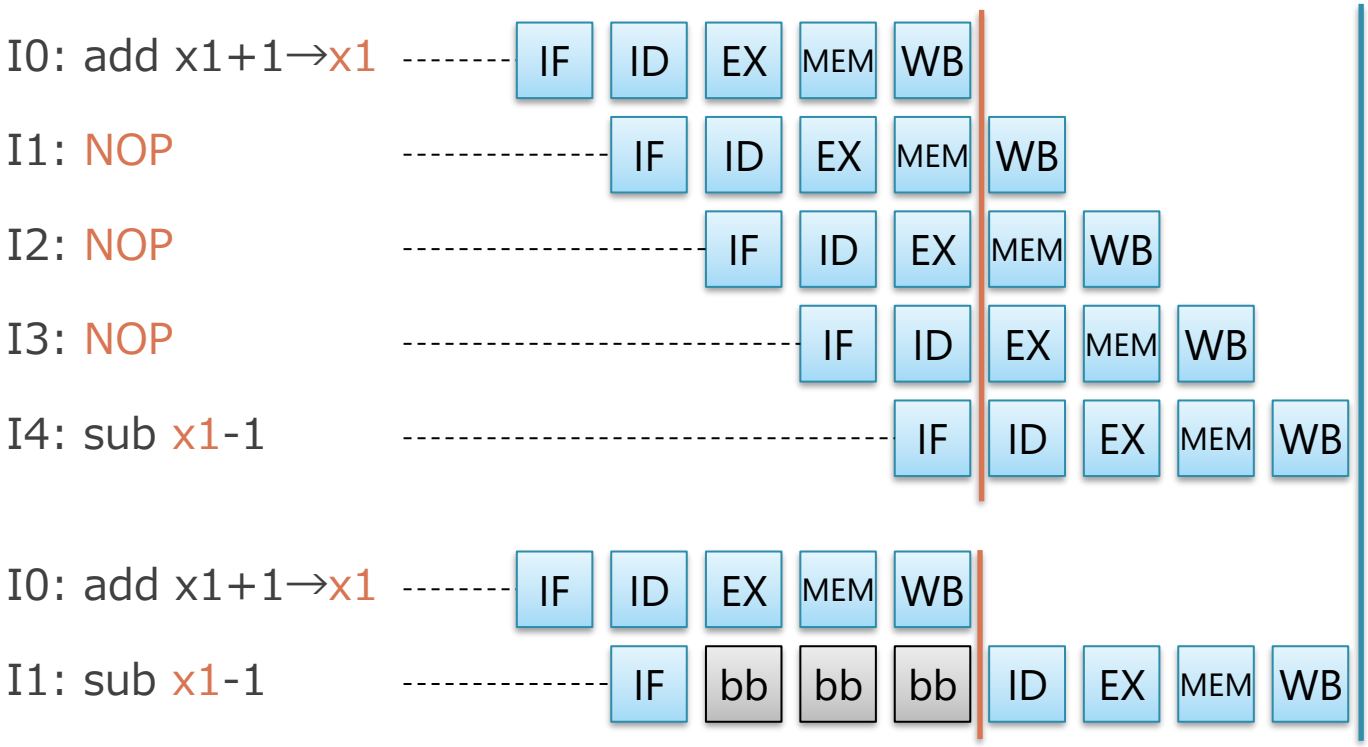
- 欠点：「仕様」なので，一度決めると変えられない
  - ◇ 後からパイプラインの段数や構造を変えると互換性がなくなる
    - □ クロックをあげるために，段数を増やせない
  - ◇ 複数の命令を同時処理しようとしたときにも互換性がなくなる
  - ◇ MIPS では遅延スロットが 1 命令分，仕様として存在
    - □ 互換性のためにこれを忠実に再現するため後年は逆に複雑化



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

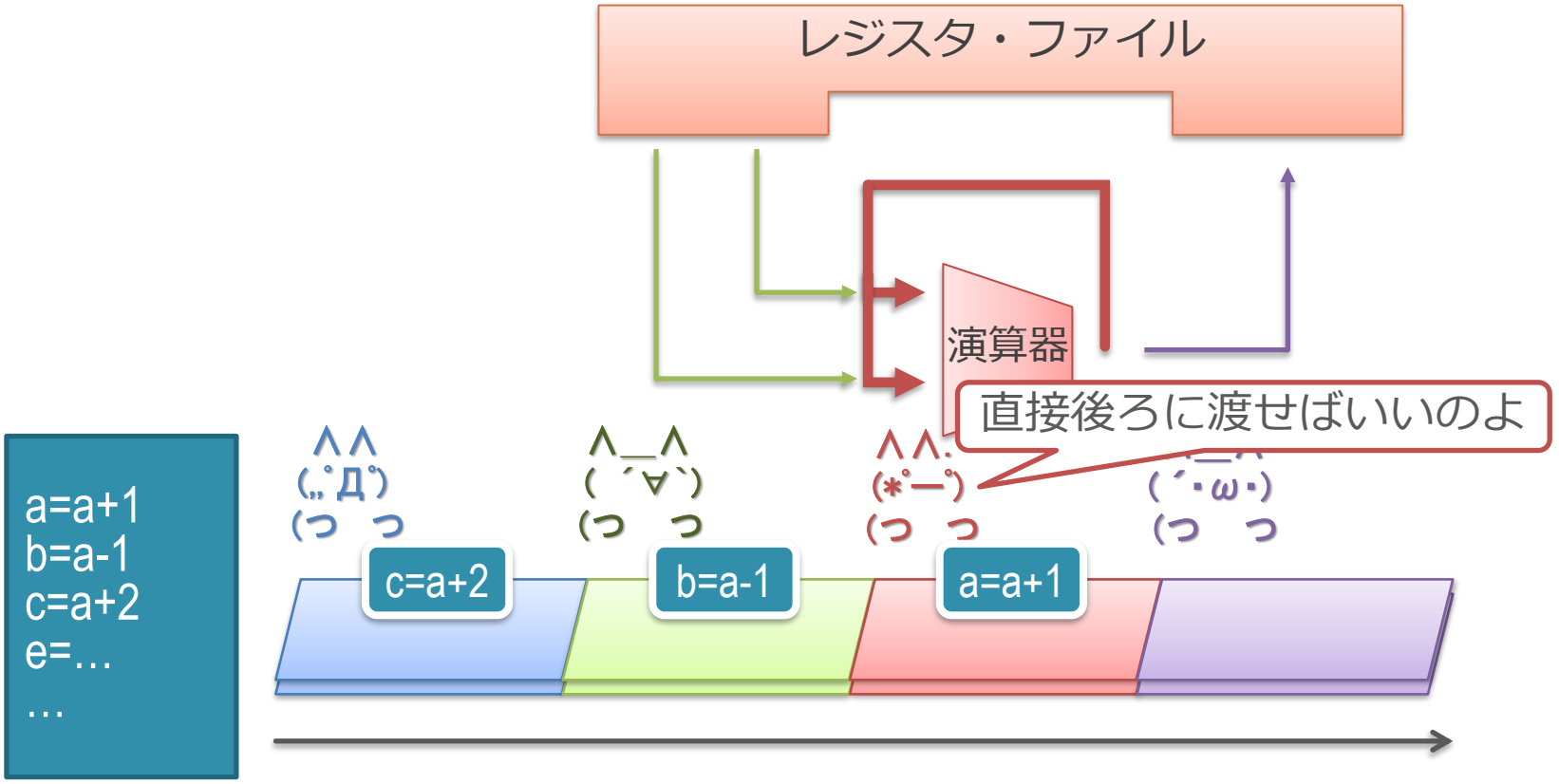


- 欠点２：並列してできる命令が常にあるとは限らない
  - ◇ NOP を入れるしかなくなる
  - ◇ 実質ストールしてバブルを入れるのと同じになってしまう

データ・ハザードの解消方法

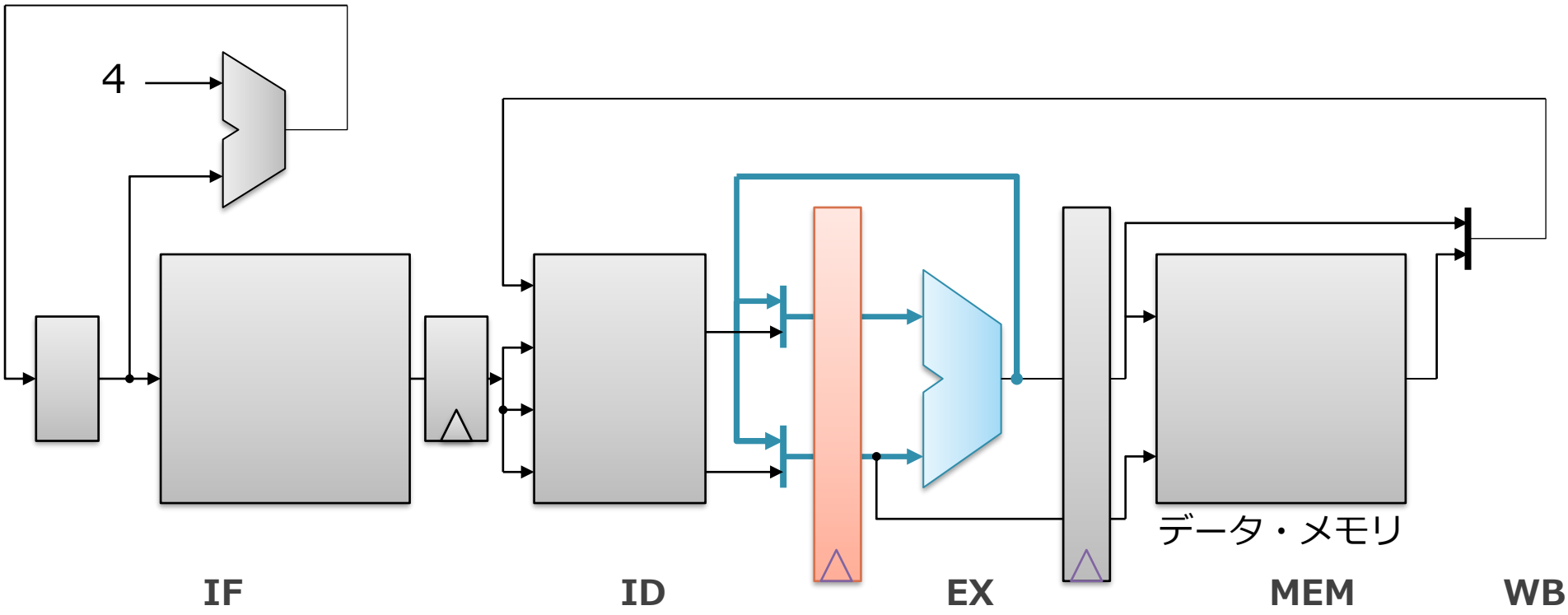
- 解消方法
  1. ストールさせる
  2. 遅延スロット（なにもしない）
  3. **フォワーディング**
  4. マルチスレッディング

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，次のサイクルにも結果を使えるようレジスタに書くと同時に手元に結果をおいておく

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

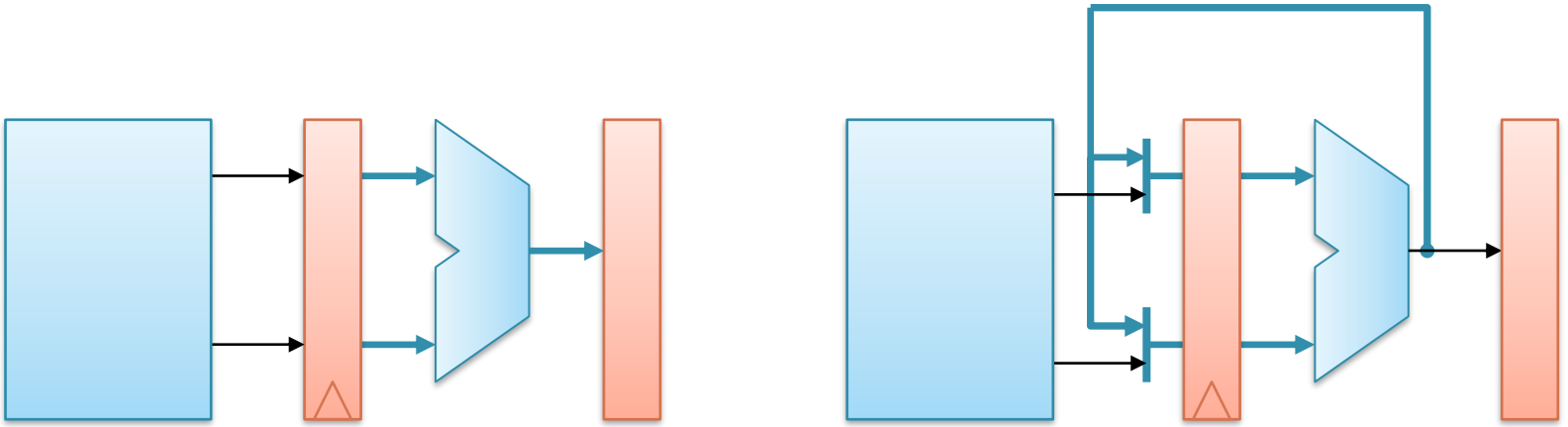
フォワーディングの利点

- 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

フォワーディングの問題

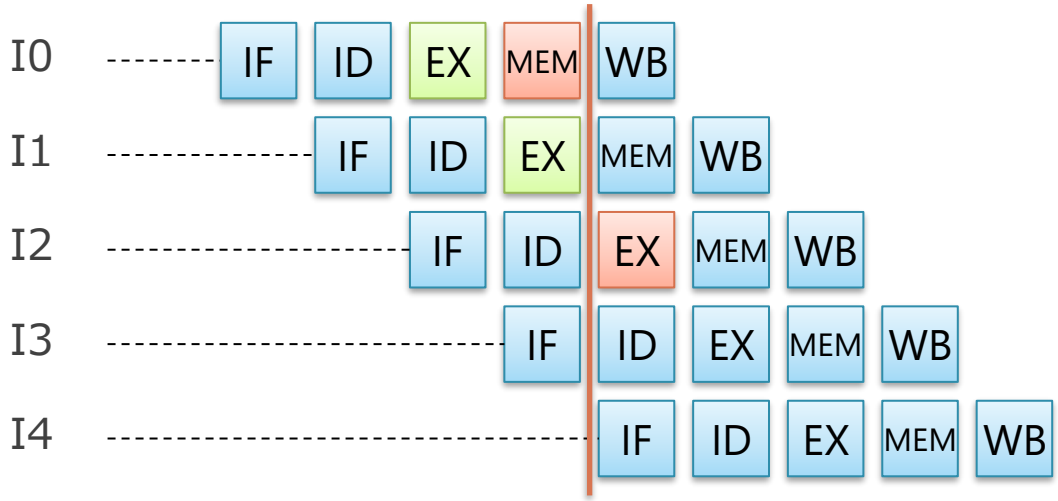
- 演算器とフォワーディングを含むステージは１サイクルで処理する必要がある
  - 分割すると，バブルを入れてるのと同じになってしまう
  - できればここはパイプライン化したくない
    - （FP 演算等の複雑なものは，やむなくパイプライン化している
- CPU 全体のクリティカル・パスになりやすい
  - １サイクルに多くの回路を詰め込む必要があるため
  - クロック周波数がここで決まることが多い

フォワーディングの問題



- フォワーディングは，この演算器の部分の遅延を増やしてしまう
  - ◇ 太線で描かれた，演算器の入力から始まるループが該当
  - ◇ クロック周波数の低下につながる

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

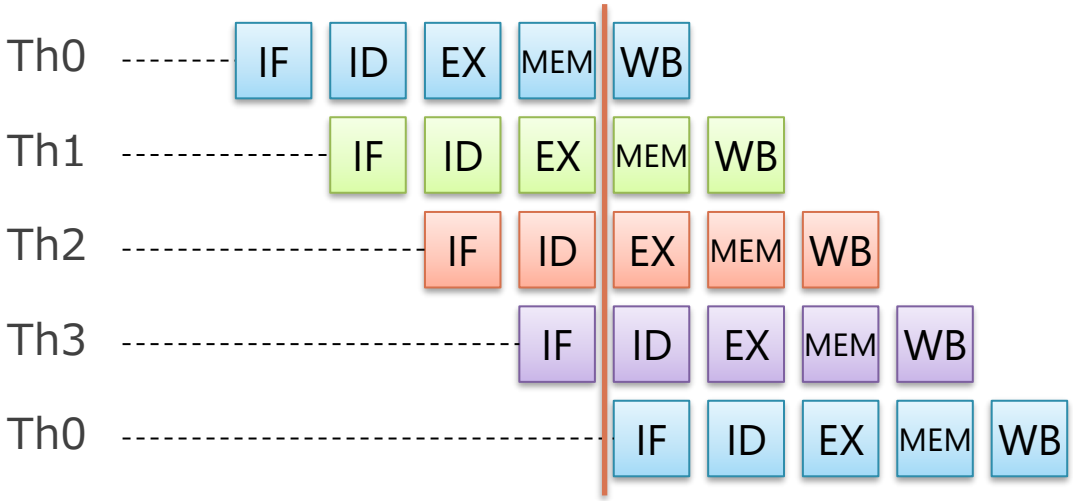
データ・ハザードの解消方法

- 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
  4. **マルチスレッディング**

マルチスレッディング

- 広義のマルチスレッド：
  - コンテキスト（PC やレジスタ）を複数持つこと
- ソフトウェアにおけるマルチスレッド
  - pthread とか
  - 複数のコンテキストが並列して動作
- ハードウェアにおけるマルチスレッド
  - ひとつの CPU 内に複数のコンテキストを複数持つ
  - 次ページの方法は「細粒度マルチスレッディング」と呼ぶ
    - ハードウェアのマルチスレッドは，他にもいろいろある
    - 「粗粒度 MT」「同時 MT」など

マルチスレッディング



- Th0 から Th3 までの4つのスレッドの命令を順に実行
  - ◇ 各スレッドは独立しているので，お互いの結果を読むことはない
- Th0 に戻ってくる頃には，前回の Th0 の結果が書き込まれている

マルチスレッディングの利点と欠点

- 利点：
  - ◇ 他の方法のような問題がおきない
    - □ 理想的にはバブルも発生せず，クロックも落ちない
  - ◇ 演算器をパイプライン化しても性能に影響がない
    - □ 他のスレッドを実行して時間をつぶしていればよい
- 欠点：
  1. 動かすスレッドがない場合は，止めておくしかない
     - □ GPU 等ではスレッドが大量にあるので，問題とならない
     - □ GPU ではループの各周がスレッドになっている
  2. スレッド数分のレジスタを持つ必要があるのでハードが大きい

データ・ハザードのまとめ

- 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
  4. **マルチスレッディング**

制御ハザード

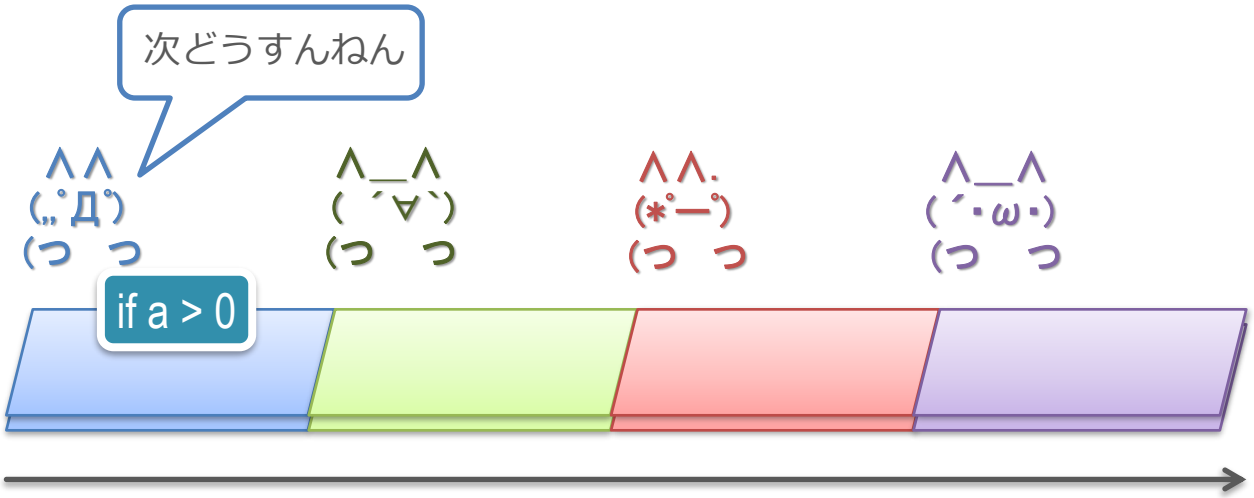
1. 構造ハザード

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）
  3. マルチスレッディング

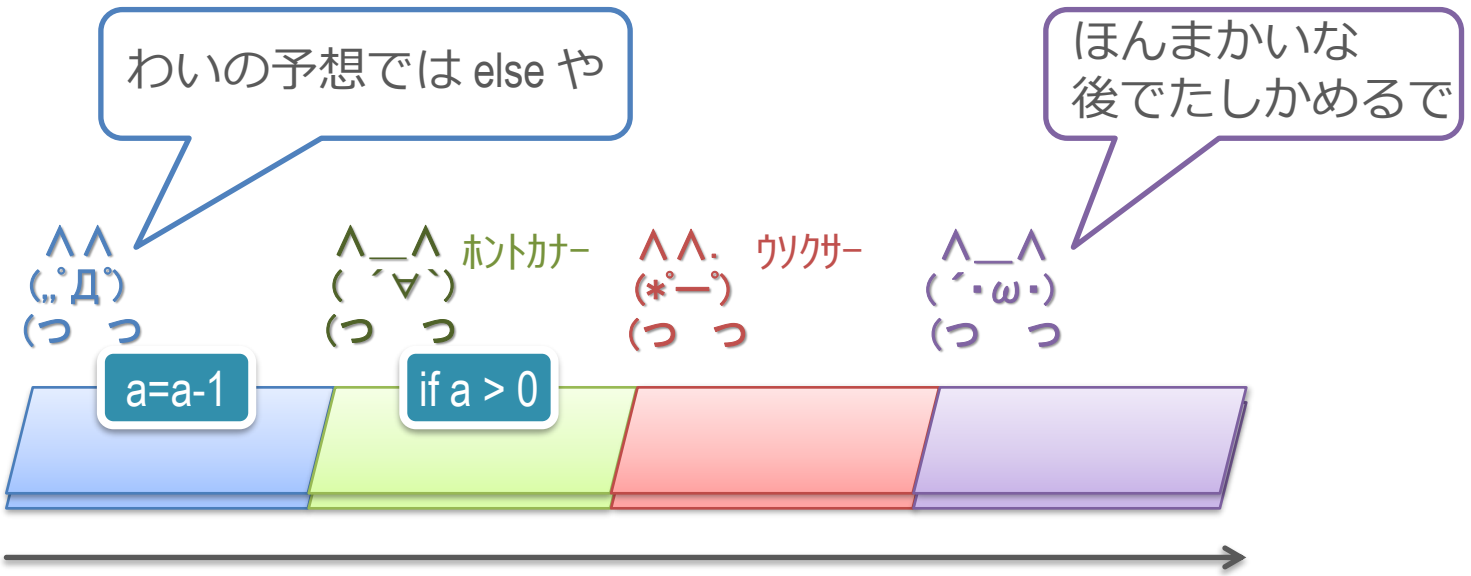
  ◇ 上記３までは，基本的にデータ・ハザードと同様にして適用できる
    - □ ただしフォワーディングは制御ハザードでは意味的に無理

  4. **分岐予測による投機実行**

分岐予測

プログラム

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

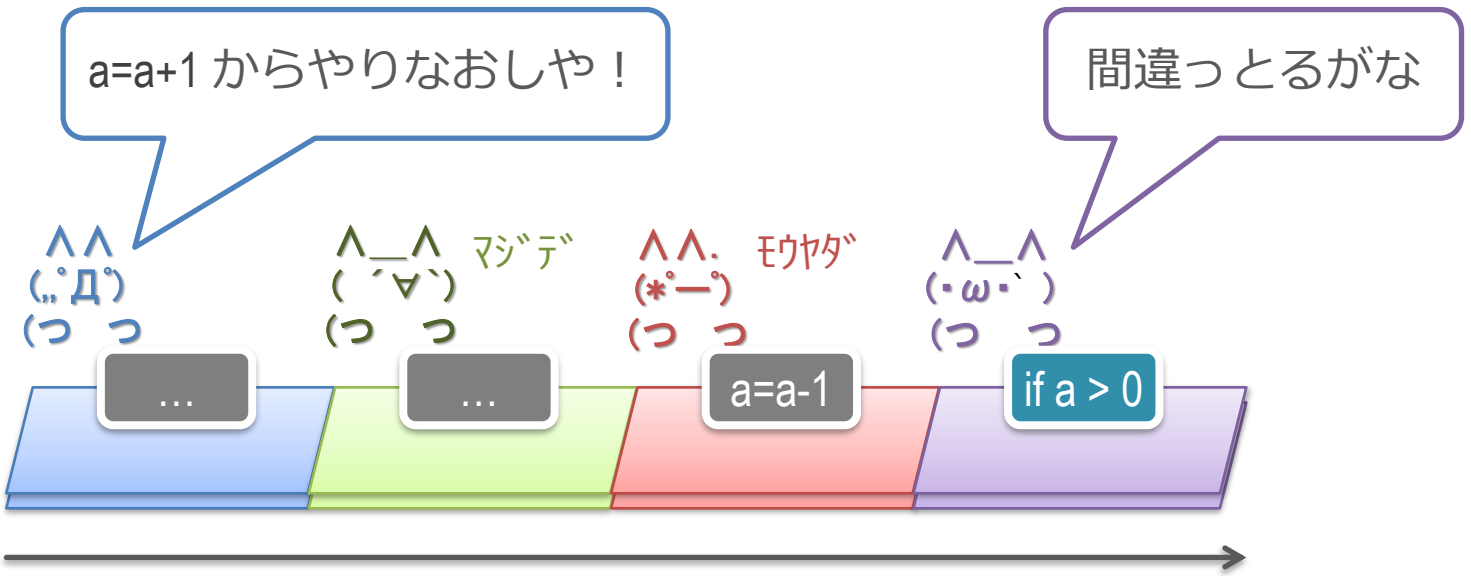


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

分岐予測ペナルティ

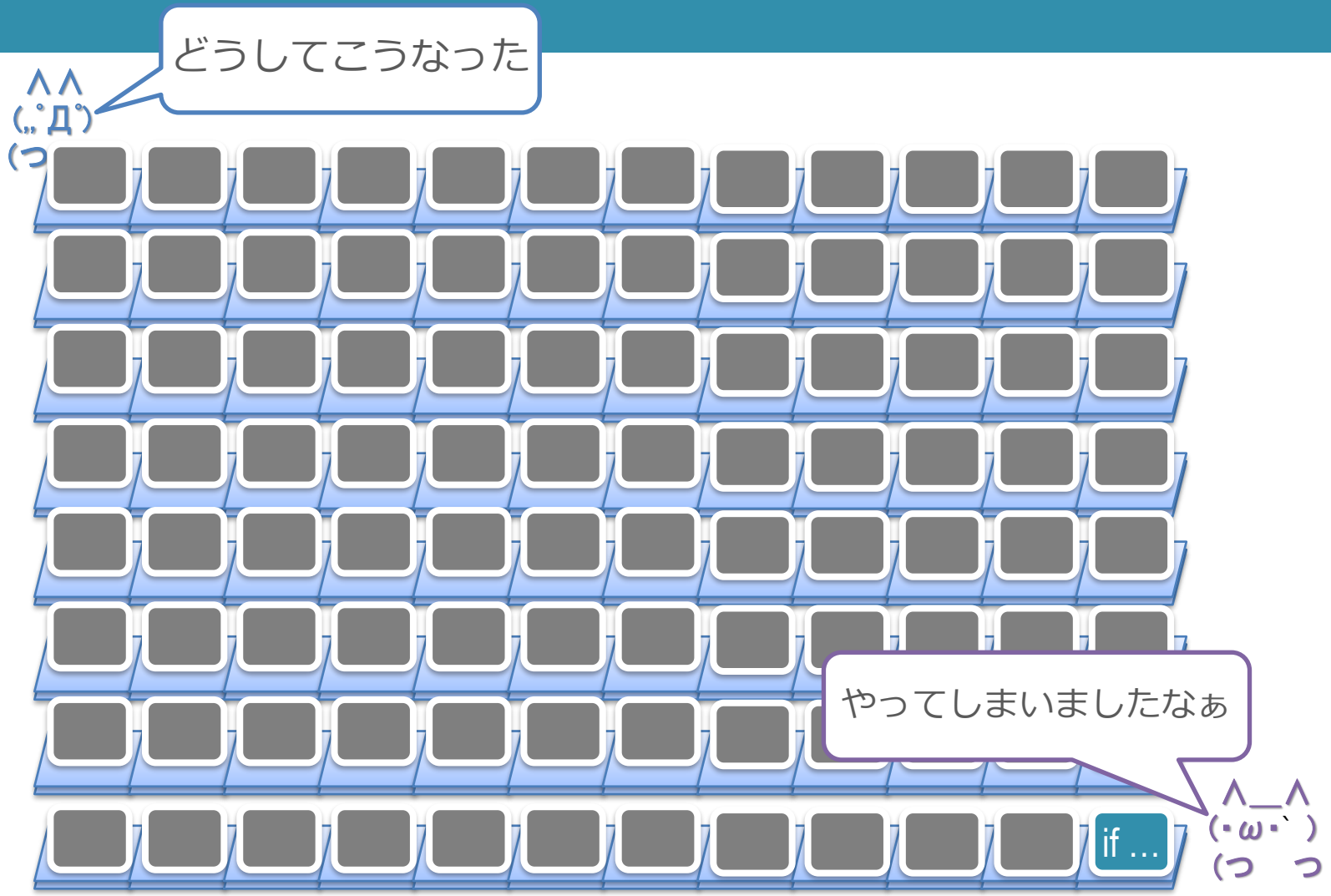
プログラム

```
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

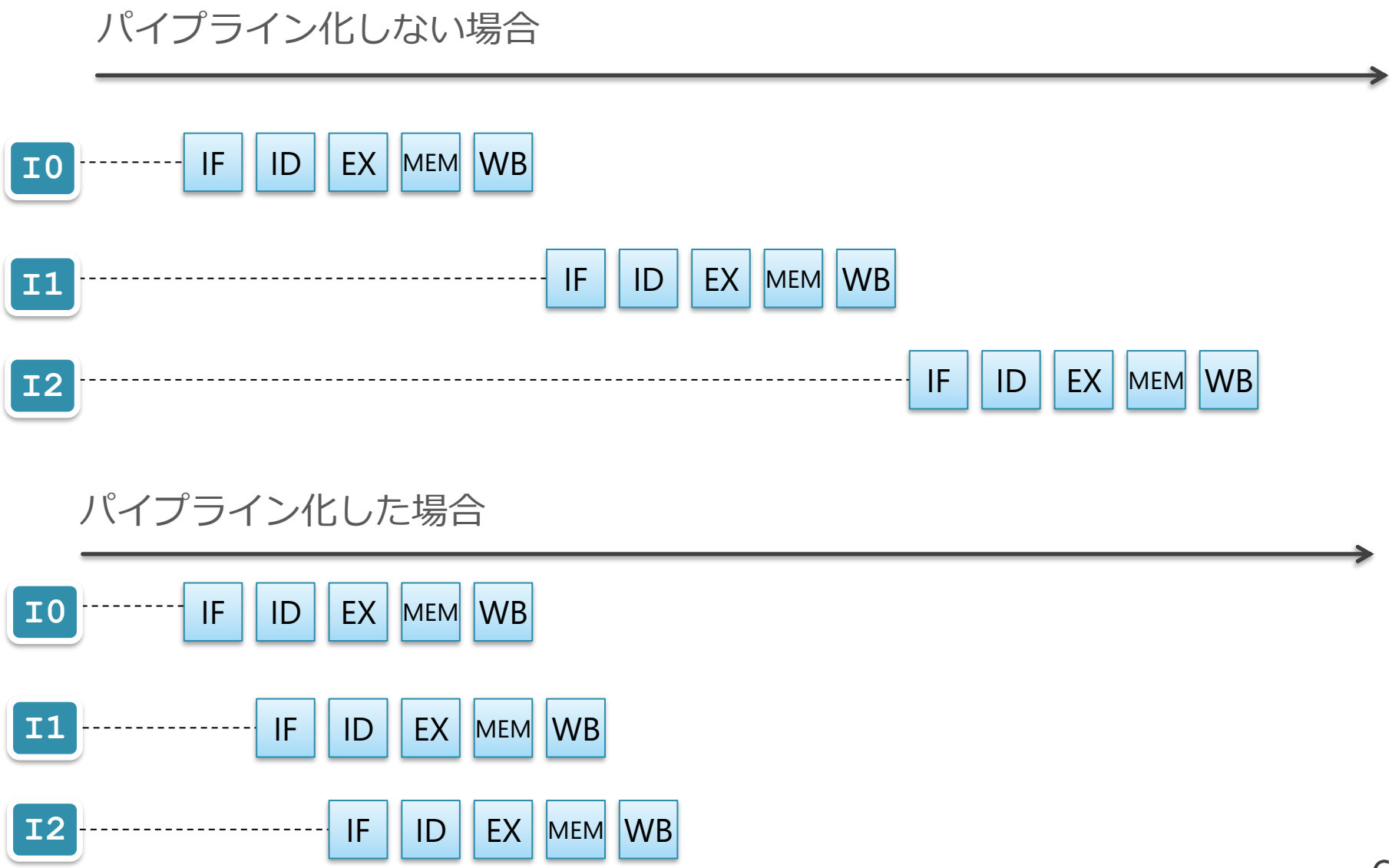


- 取り消しは最悪数十命令以上に
  - ◇ IBM POWER8 だと，8命令同時 × 10数段

命令パイプラインと性能

もくじ

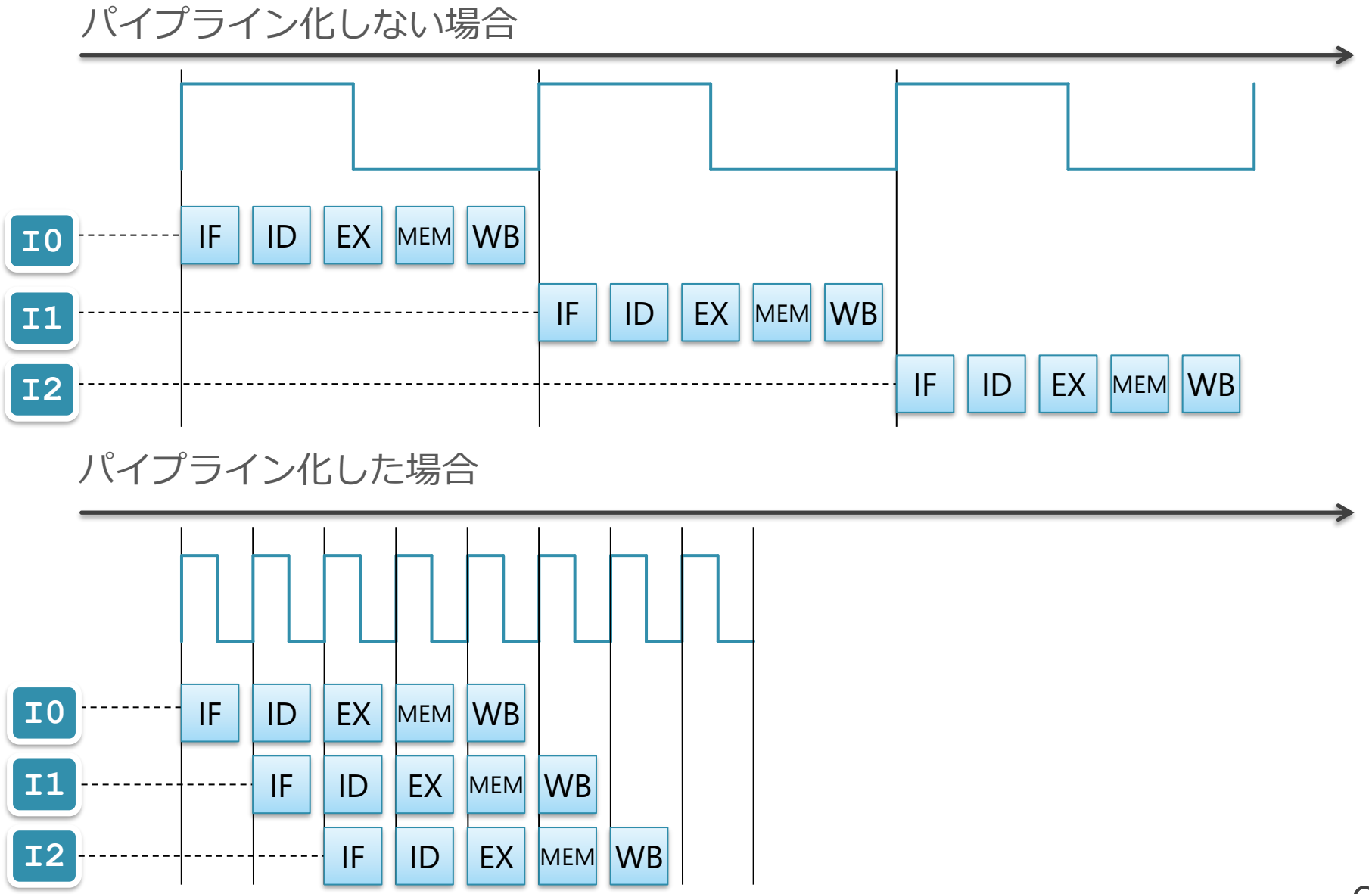
パイプライン化によるスループット向上



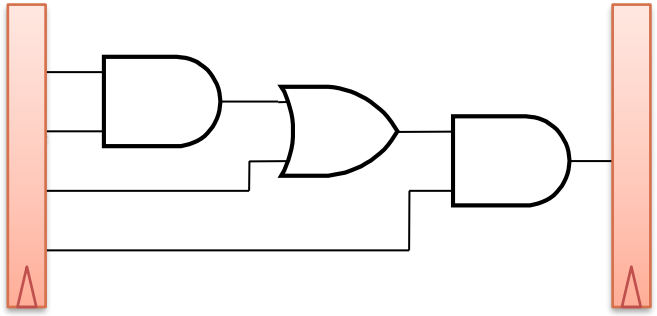
パイプライン化の意味

- 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

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

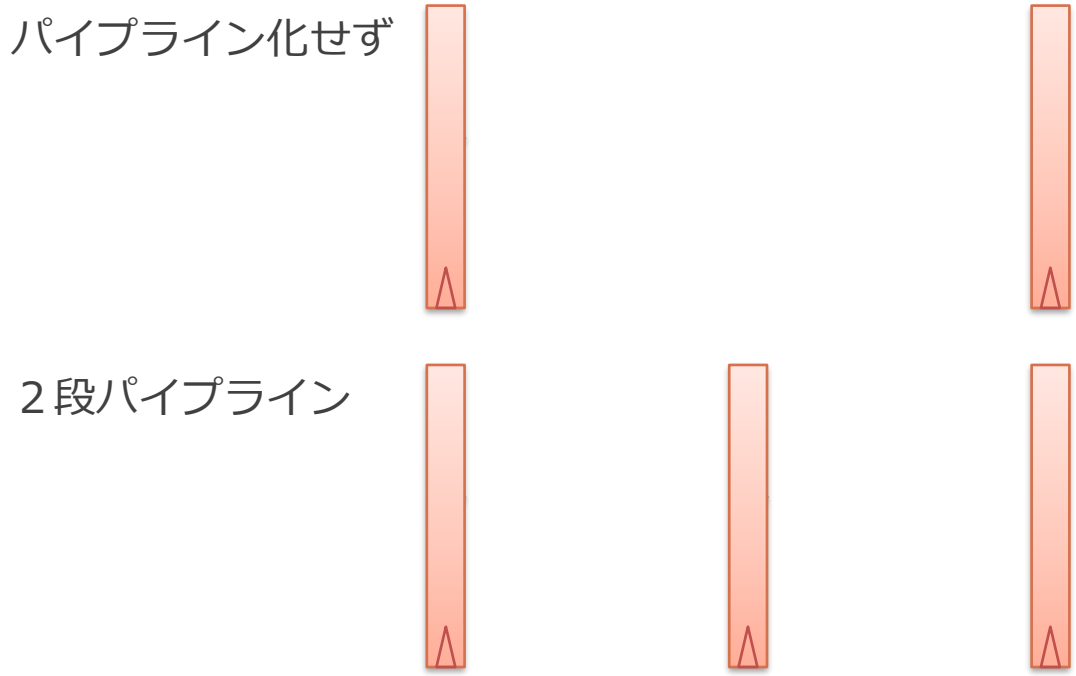


ステージ内の信号の伝播を考える



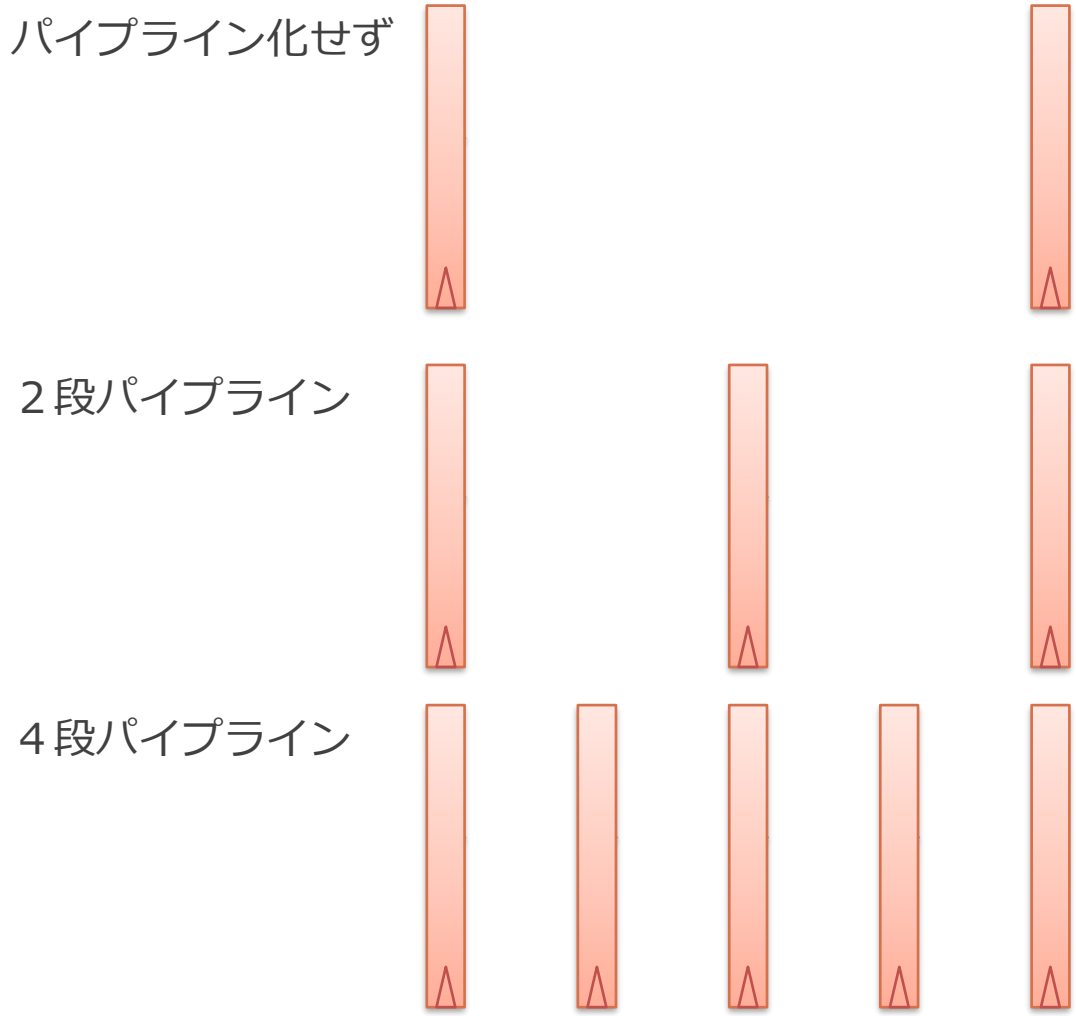
- パイプライン：ステージ：
  - ◇ 複数のパイプライン・ラッチで挟まれてた，
  - ◇ 組み合わせ回路（ゲート）
- 矢印の動き：
  - ◇ クロック開始時に，左のラッチからでた信号が
  - ◇ 組み合わせ回路を通って，伝播していく様子

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

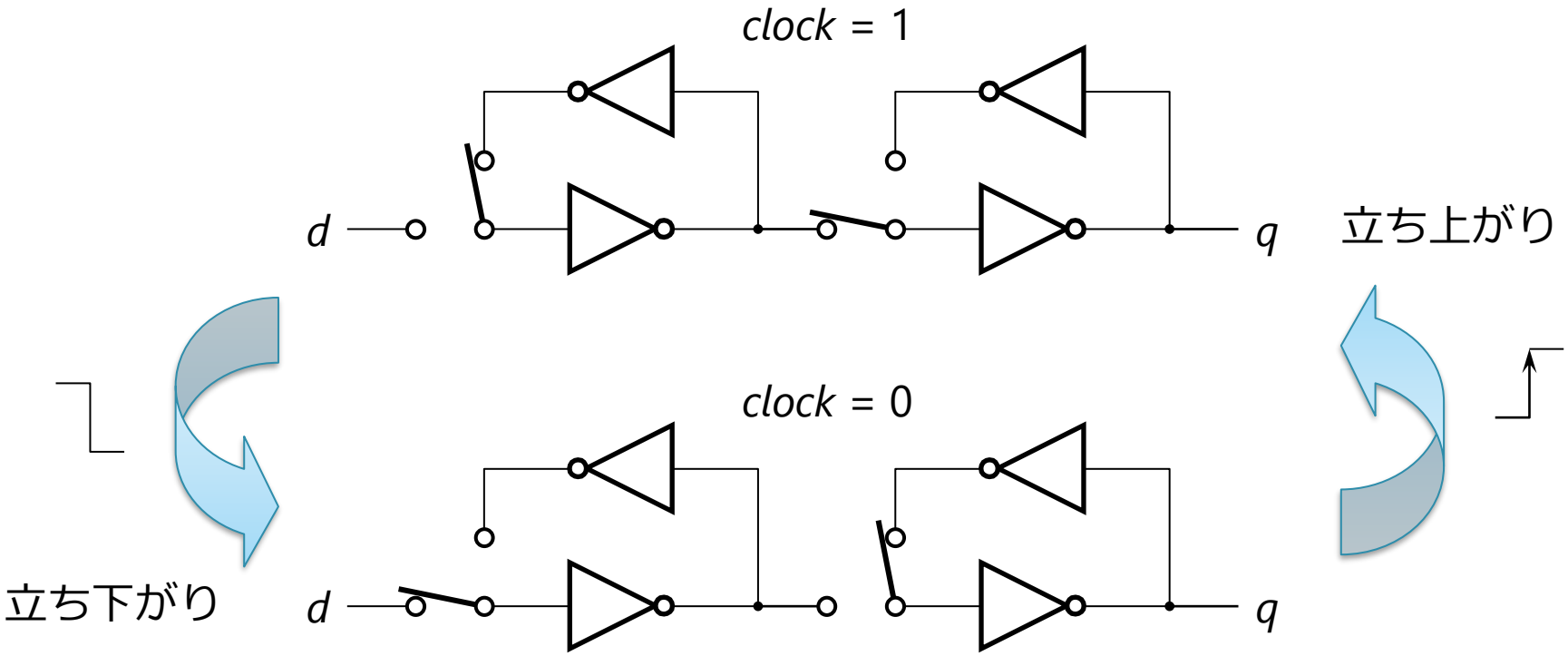
- パイプライン段数を増やしていけば，どこまでも速くなるのか？
  - ◇ ならない
- 理由：
  1. 回路的な理由による周波数向上の限界
  2. アーキテクチャ的な理由による実効性能の限界

回路的な理由

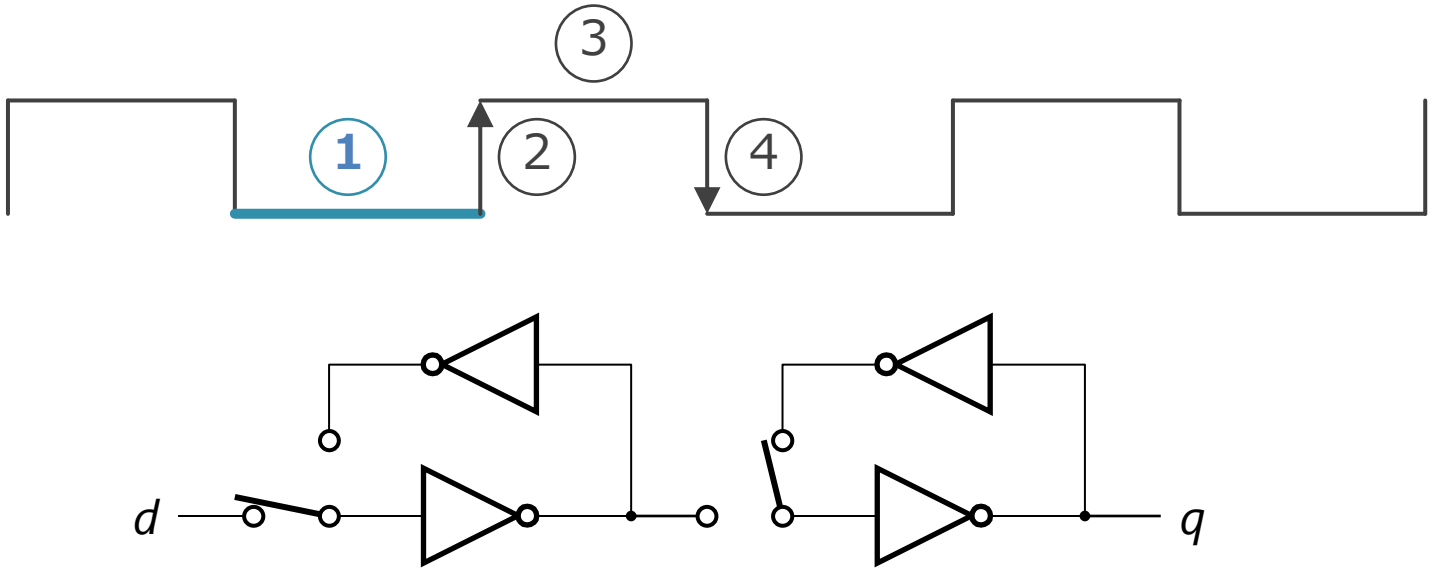
- 理由：
  1. D-FF 自体の遅延のため
  2. 消費電力と熱のため

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

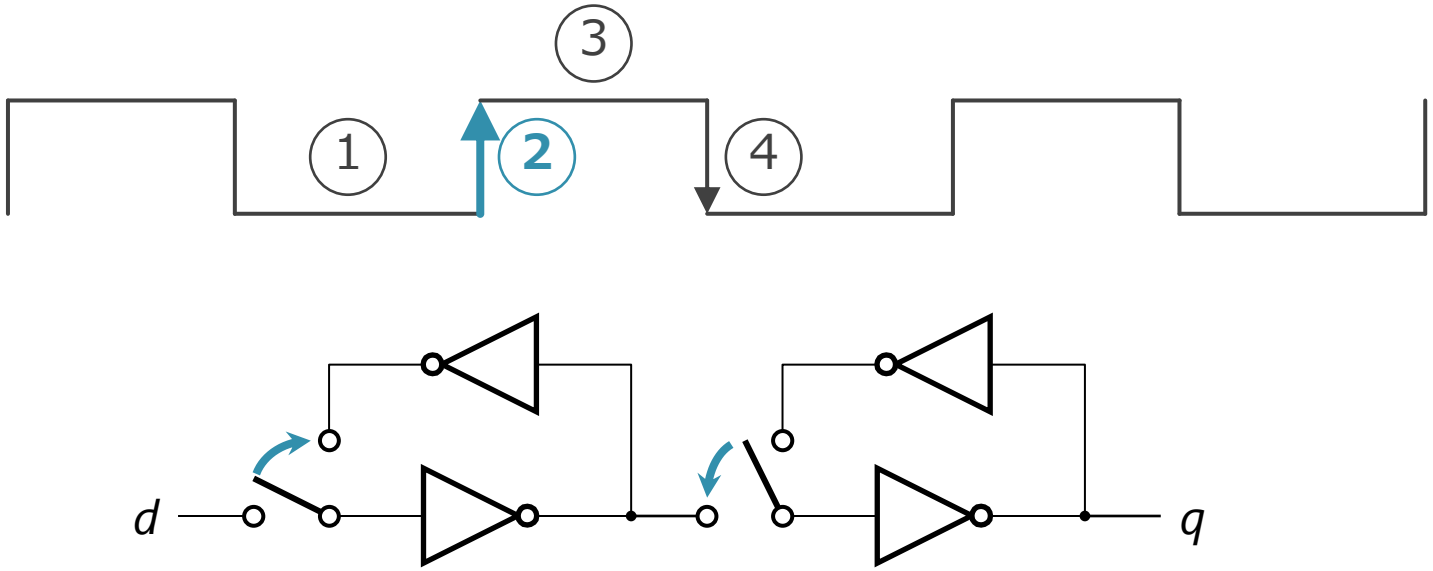


D-FF の動作 ① クロック信号が Low にあるとき



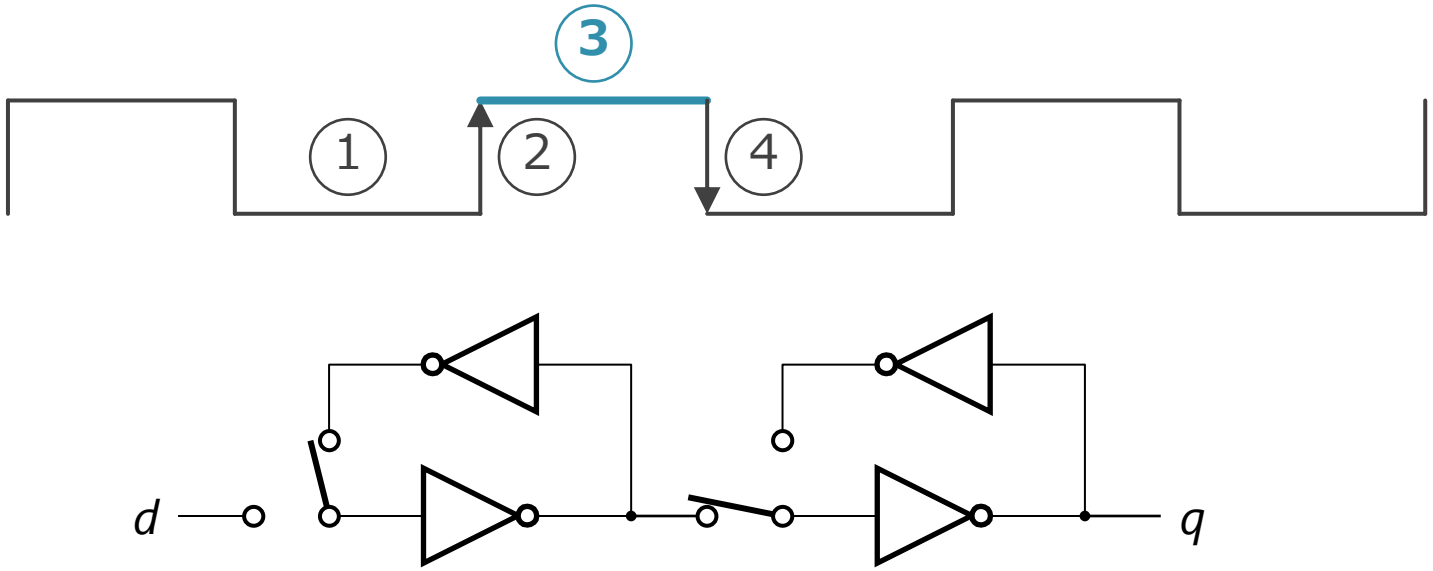
- 左側のループ：
  - ◇ d の入力の変化に応じて，インバータの状態が随時切り替わる
  - ◇ 右側のループとは遮断されている
- 右側のループ：
  - ◇ ループのインバータの状態（=記憶）が q に出力され続ける

D-FF の動作 ② クロック信号の立ち上がり



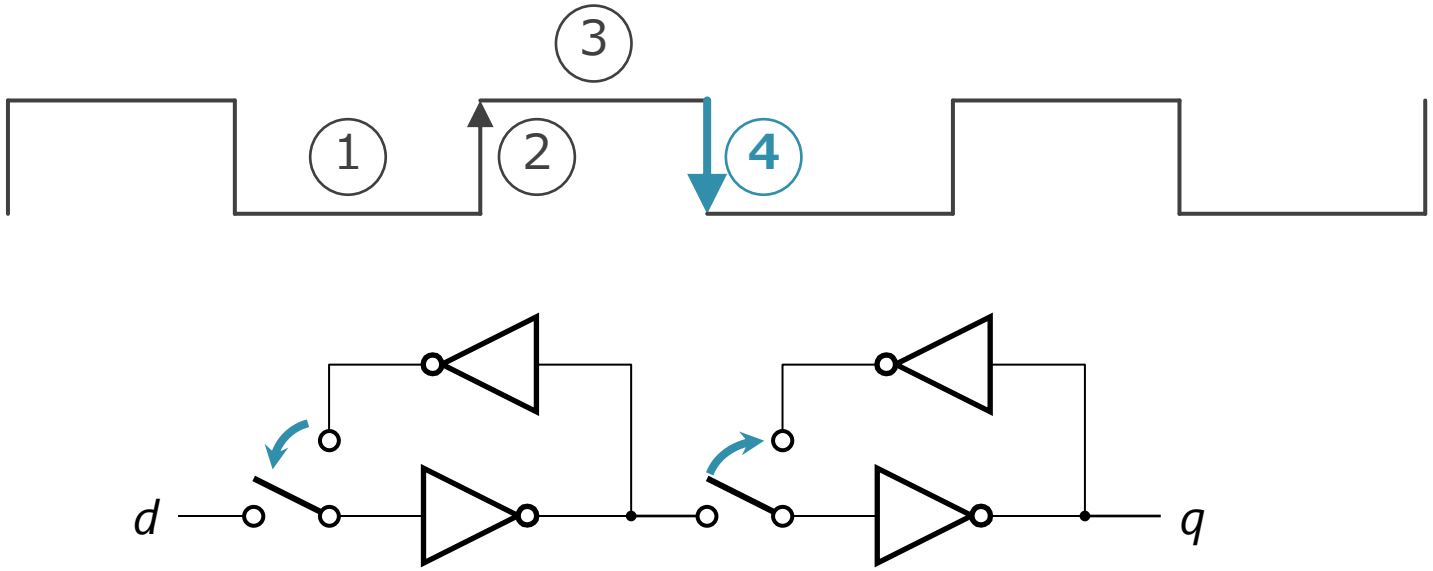
- 左側のループ：
  - ◇ *d* と遮断され，ループが形成される
  - ◇ 直前まで *d* に入力されていた信号が記憶される
- 右側のループ：
  - ◇ 左側のループと繋がり，ループが解除される

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

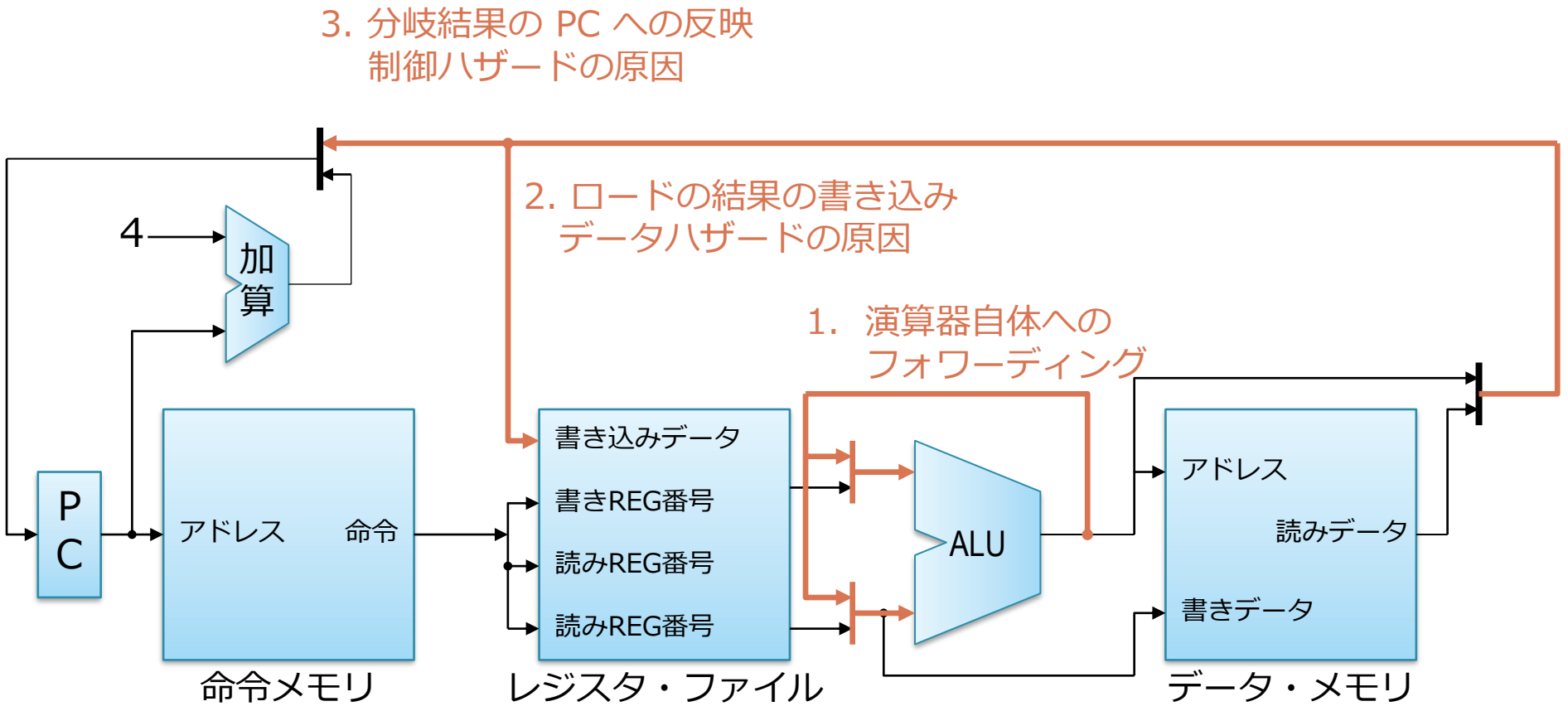
パイプライン化の限界

- 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
  2. **アーキテクチャ的な理由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

バックエッジ：逆方向（右から左）にいく信号

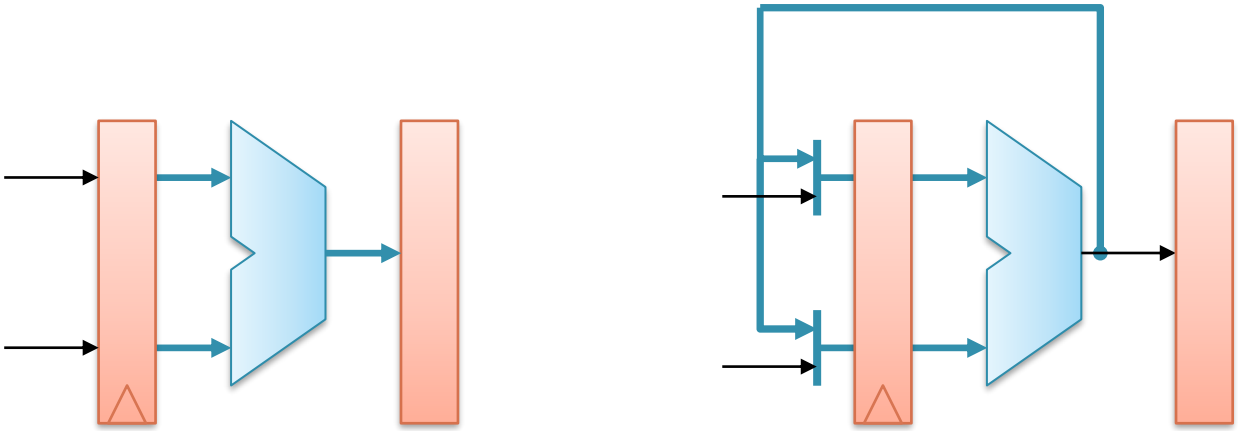


- バックエッジがあるため，命令を単純に流せない場合がある
  - ◇ 工場のラインのように，一方向に流せない

問題となるバックエッジ

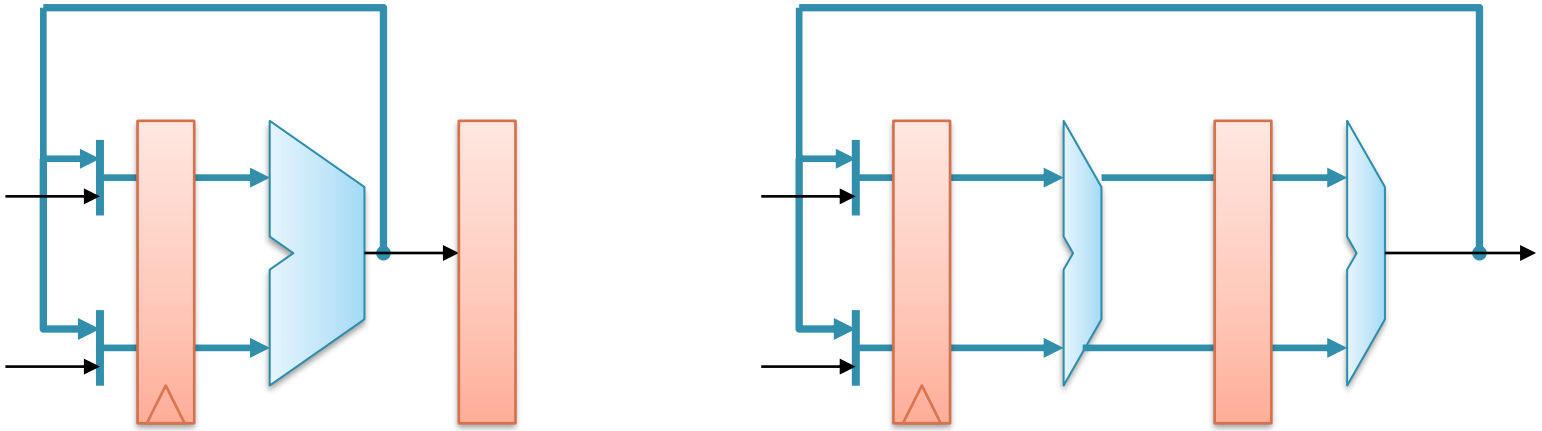
1. 演算器のフォワーディング
2. ロードによるデータ・メモリの読み出し
3. 分岐結果の PC への反映

1. 演算器のフォワーディング



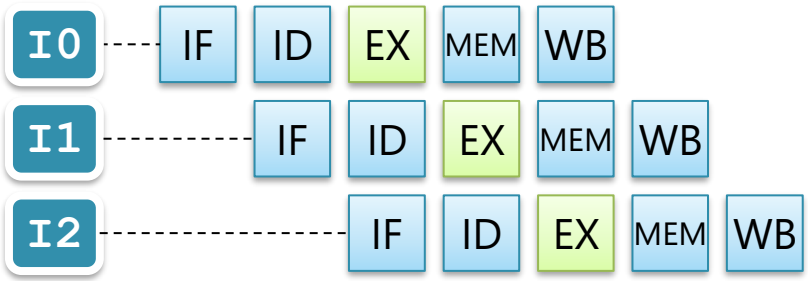
- ここは結構遅延が長い
  - ◇ クロック周波数の低下につながるのでパイプライン化したくなる

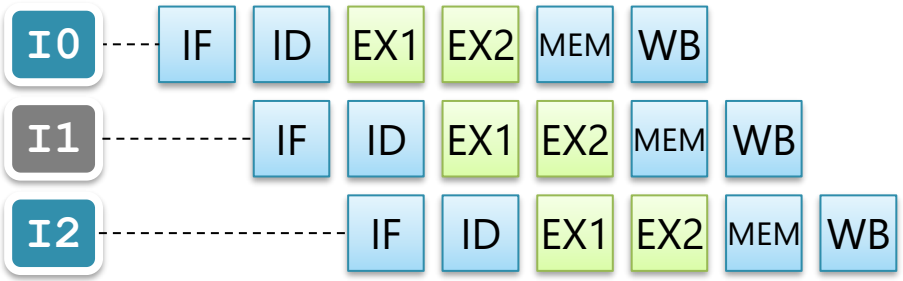
演算器のパイプライン化



- 演算器のパイプライン化
  - ◇ たとえば加算を，下位３２ビットと上位３２ビットを２ステージかけてやる
  - ◇ １ステージあたりの遅延は半分になる

演算器をパイプライン化した場合の問題



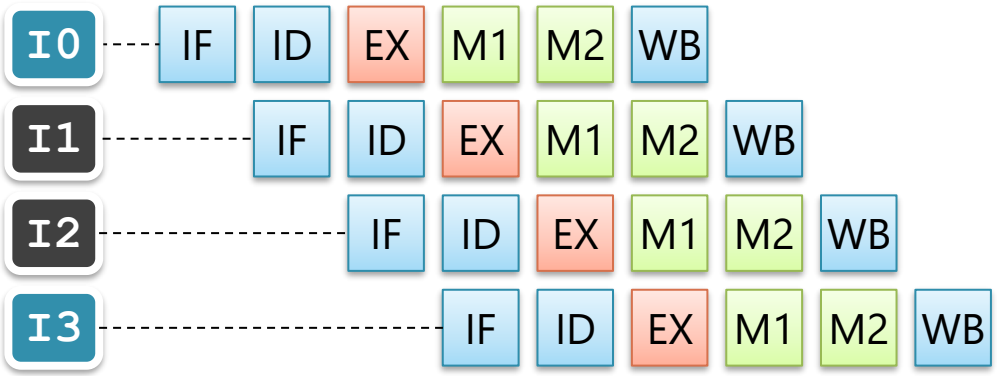


- 依存関係にある命令を連続して実行できなくなる
  - `I0` の EX2 が終わる前に，`I1` の EX1 が始まる
  - もし，他に実行するべき命令がおけなければ，遊ばせとくしかない
- 場合によっては性能が返って下がる
  - 周波数が上がったが，２サイクルに１回しか命令が実行できない
- 基本的な整数演算はパイプライン化せず１ステージを死守するのが普通
  - 乗除算や，浮動小数点演算はあきらめてパイプライン化

問題となるバックエッジ

1. 演算器のフォワーディング
2. **ロードによるデータ・メモリの読み出し**
3. 分岐結果の PC への反映

ロードによるデータ・メモリの読み出し



- 演算器の場合とほぼ同様
  - 上は，MEM が M1 と M2 にパイプライン化された場合
    - I1 と I2 は，I0 の結果を使えない
  - MEM ステージをパイプライン化すると，この部分が長くなる
- しかし，この部分をパイプライン化することはよくある
  - ロードは演算よりは出現頻度が低い
  - メモリ（キャッシュ）のレイテンシは演算器よりかなり長くなることが多いためしかたない

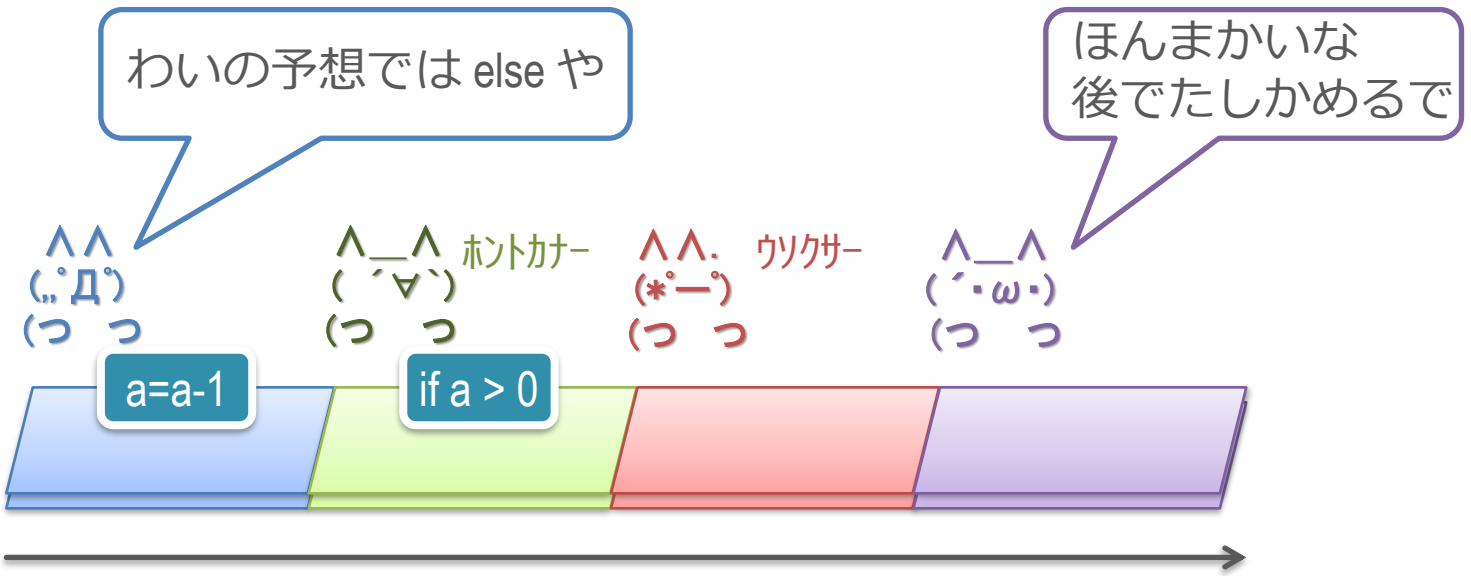
問題となるバックエッジ

1. 演算器のフォワーディング
2. ロードによるデータ・メモリの読み出し
3. **分岐結果の PC への反映**

分岐予測

プログラム

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

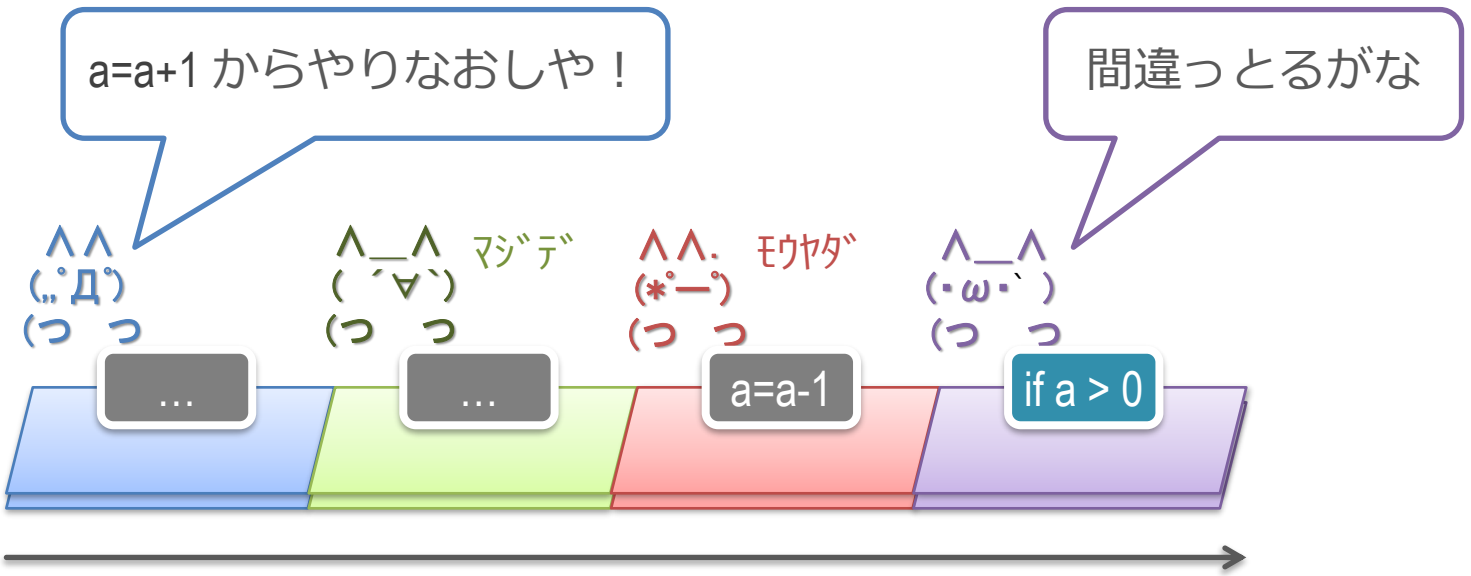


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

分岐予測ペナルティ

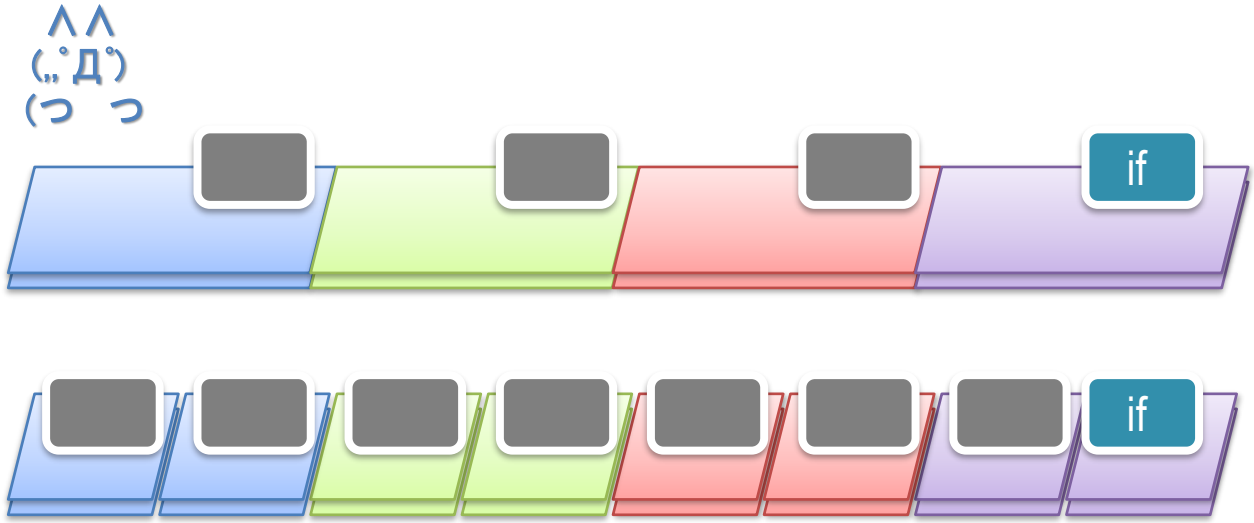
プログラム

```
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

分岐予測ペナルティの大きさ



- パイプラインを深くすると，
  - ◇ = if が右に到達してミスが判明するまでのステージが増える
  - ◇ = 予測ミス時に取り消される命令数が大きくなる
    - □ 一瞬で全員を消せず，取り消す命令数に応じた時間がかかる
- 実時間が伸びているわけではないことに注意
  - ◇ if が右に到達するまでの実時間は変わってない
  - ◇ 矢印が伸びるアニメーションを思い出してほしい

パイプライン化の限界のまとめ

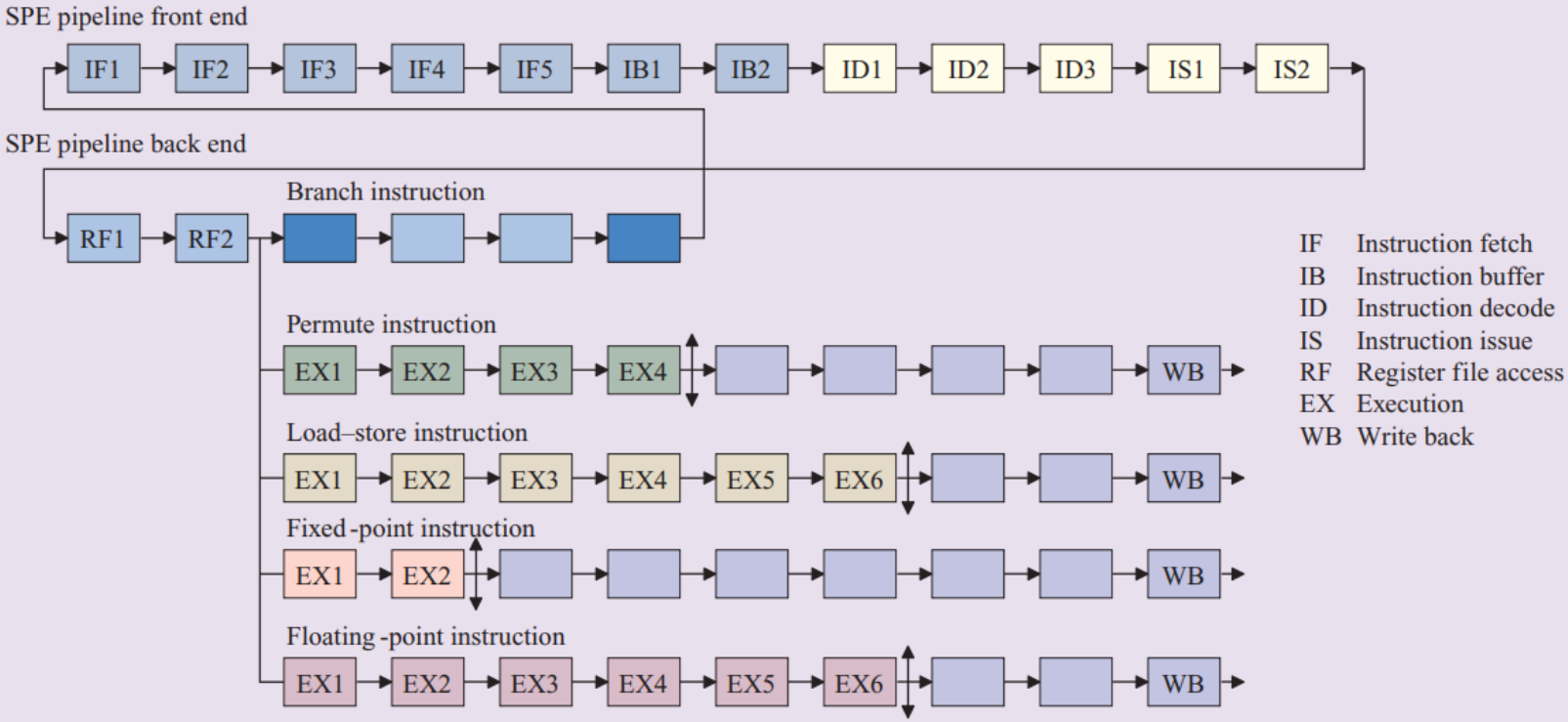
- 速度が上がらなくなる理由：
  - ◇ 回路的な理由による周波数向上の限界
    - □ D-FF の遅延
    - □ 電力と熱
  - ◇ アーキテクチャ的な理由による実効性能の限界
    - □ バックエッジによる実効性能の低下
    - □ （今日話した話題意外に，スーパスカラ固有の性能低下も

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

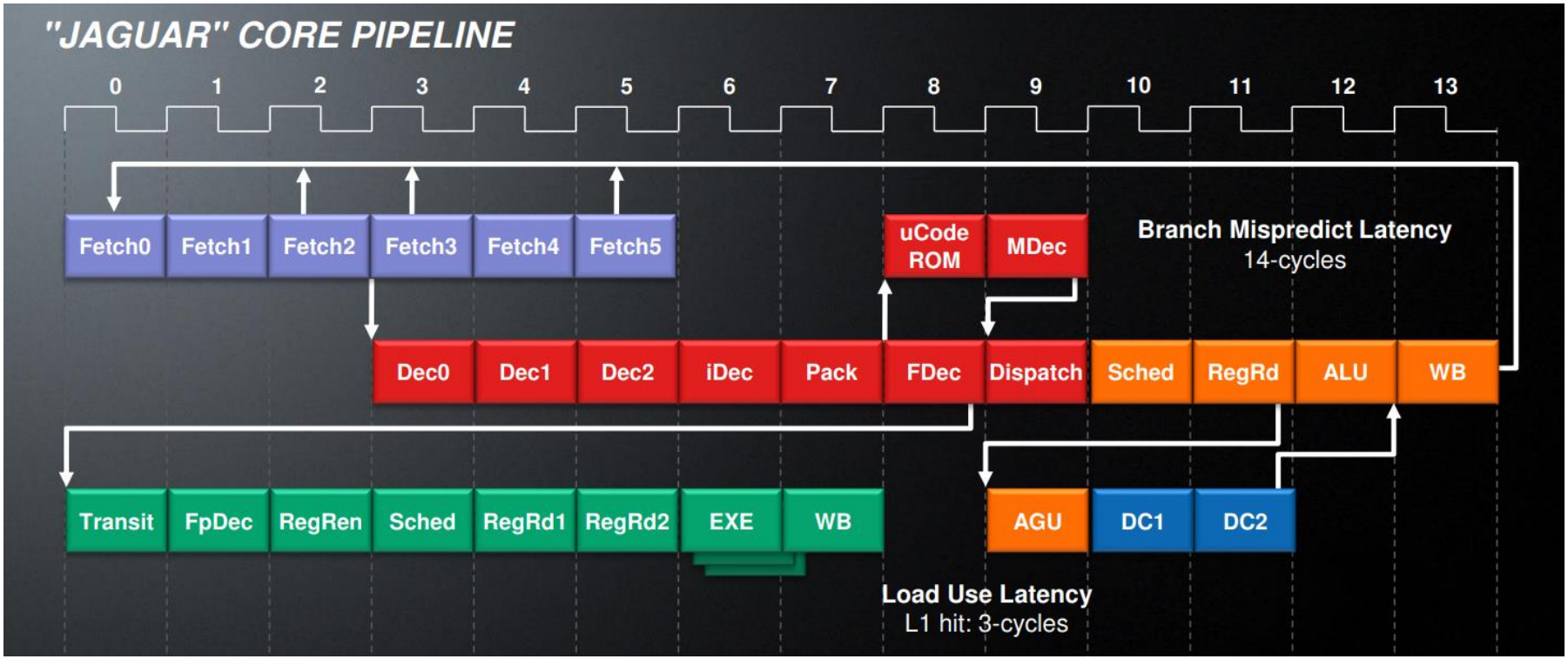
Sony/IBM/東芝 Cell (SPE)

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



AMD JAGUAR

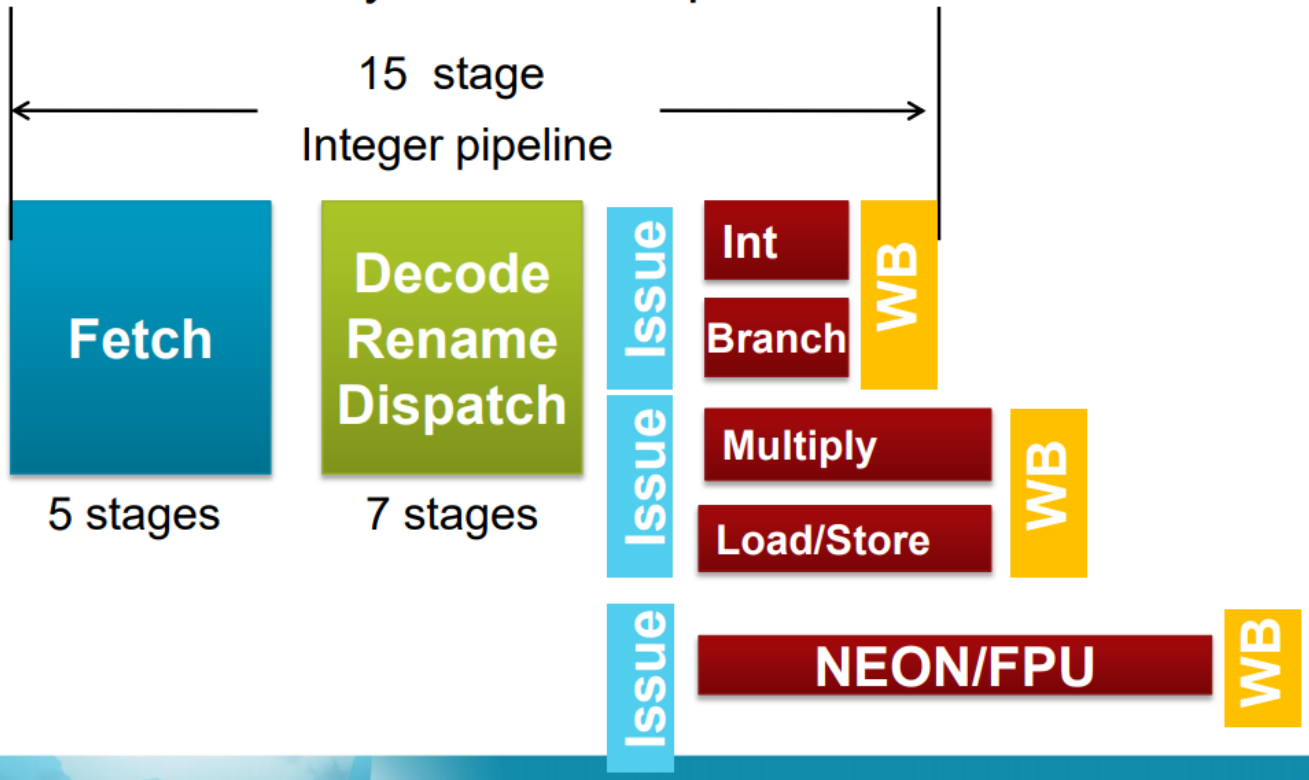
"JAGUAR" AMD's Next Generation Low Power x86 Core より



Cortex-A15 Pipeline Overview

15-Stage Integer Pipeline

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



分岐予測

もくじ

用語の定義（１）

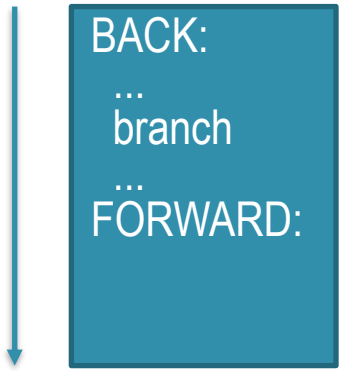
- 方向分岐
  - ◇ if 文のように，２方向に分岐する分岐命令
- 間接分岐
  - ◇ レジスタに格納されている値のアドレスに飛ぶ分岐命令
  - ◇ 任意の場所に飛ぶことができる

用語の定義（２）

- 分岐の成立/不成立
  - ◇ 条件が成立（taken）：　　指定されたアドレスへジャンプ
  - ◇ 条件が不成立（untaken）：次の命令（PC+４）に移る
- 例：bne x1, x2, TARGET
  - ◇ 成立：　x1 と x2 の値が異なった場合は，TARGET にジャンプ
  - ◇ 不成立：x1 と x2 の値が同じ場合は，次の PC に

用語の定義（３）

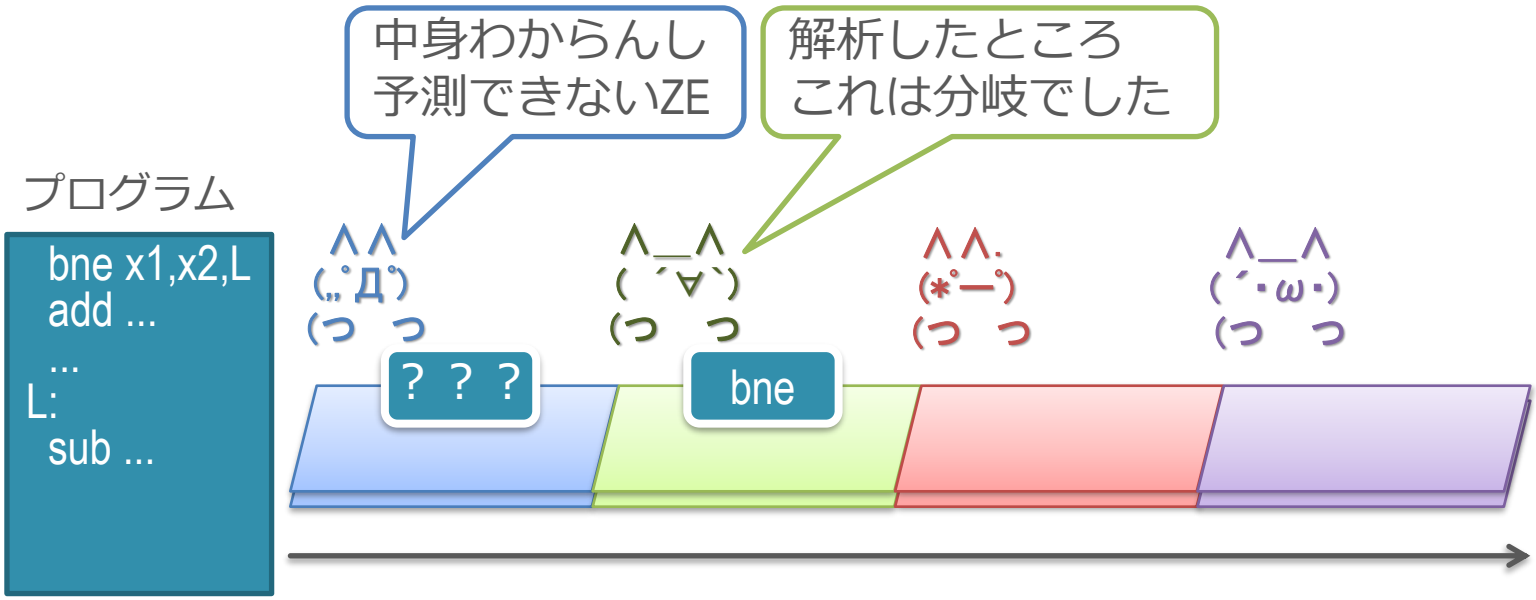
- 分岐先 アドレス or ターゲット
  - ◇ 分岐が成立した際の飛び先のアドレスのこと
- 前方分岐：
  - ◇ 分岐先ターゲットが分岐自身のアドレスよりも大きい分岐のこと
  - ◇ プログラムの進行方向に対して前方に飛ぶことから
- 後方分岐：
  - ◇ 分岐先ターゲットが分岐自身のアドレスよりも小さい分岐のこと
  - ◇ 後方に飛ぶ = ループを作る



分岐予測

- 分岐予測では，以下の３つを全て行う必要がある
  1. 分岐命令かどうか予測（分岐種別の予測）
  2. 分岐先ターゲット予測
  3. 分岐方向予測
- if-then-else の方向だけを予測していれば良いわけではない
- （今は方向分岐のみを扱い，間接分岐は考えない

1．分岐かどうか予測の必要性



- メモリから命令が取れるまでは，それが分岐かどうかはわからない
  - ◇ 命令フェッチは複数段にパイプライン化されていることが多い
  - ◇ 以降のターゲットや方向の予測をすべきかどうかが，わからない
- 一方パイプライン先頭では即座に次のアドレスを予測しないといけない
  - ◇ 分岐かどうかわかるまでまっていては，バブルができる

2．分岐先ターゲットの予測の必要性

- メモリから命令が取れるまでは，分岐成立時の飛び先の場所もわからない
  - ◇ いくつ先 or いくつ前に飛ぶのか？

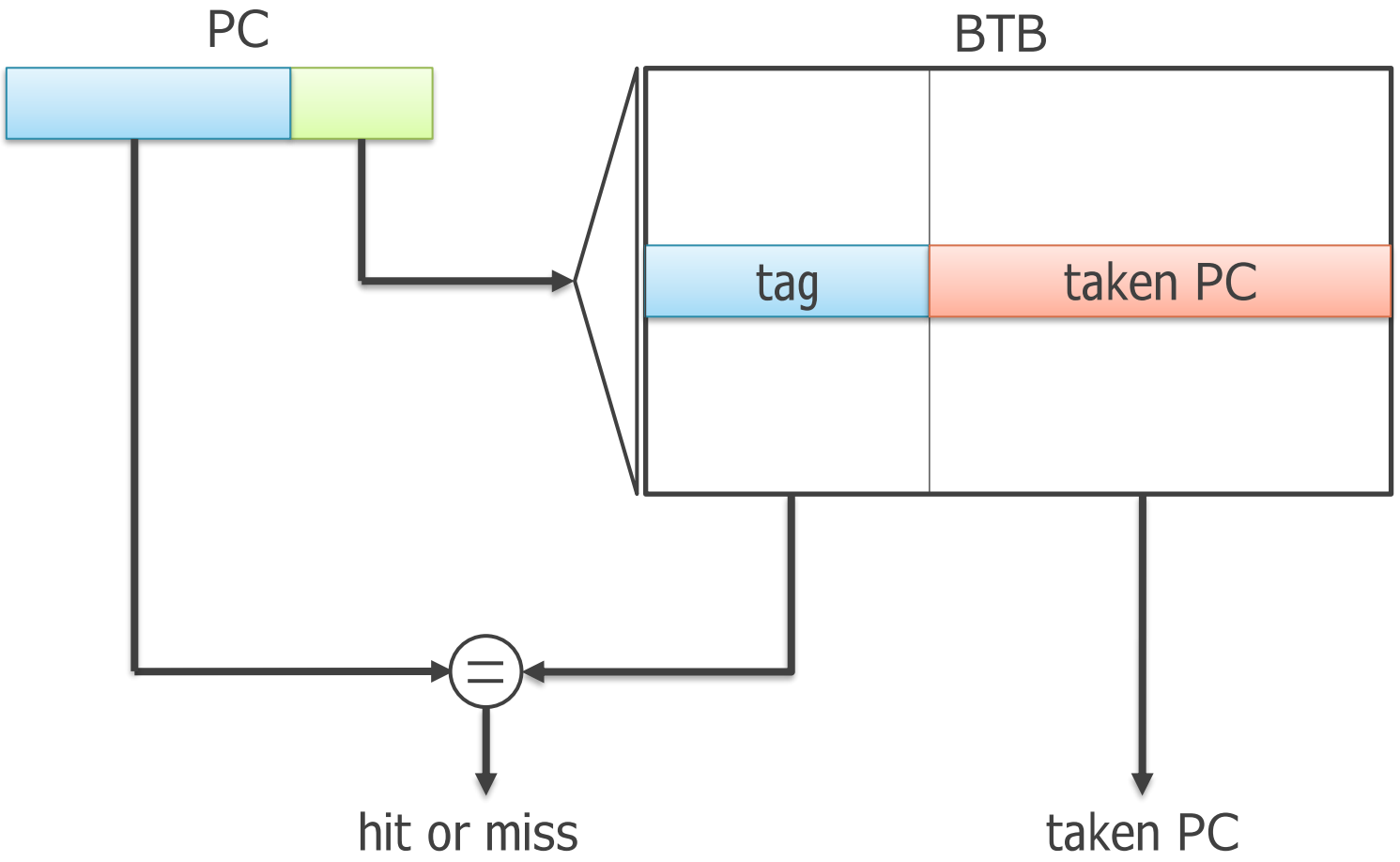
BTB（Branch Target Buffer）による予測

- BTB と呼ぶ表を使って以下を予測
  1. 分岐命令かどうか予測
  2. 分岐先ターゲット予測

- BTB
  - ◇ 入力：PC
  - ◇ 出力：
    - □ hit or miss
    - □ ターゲットのアドレス

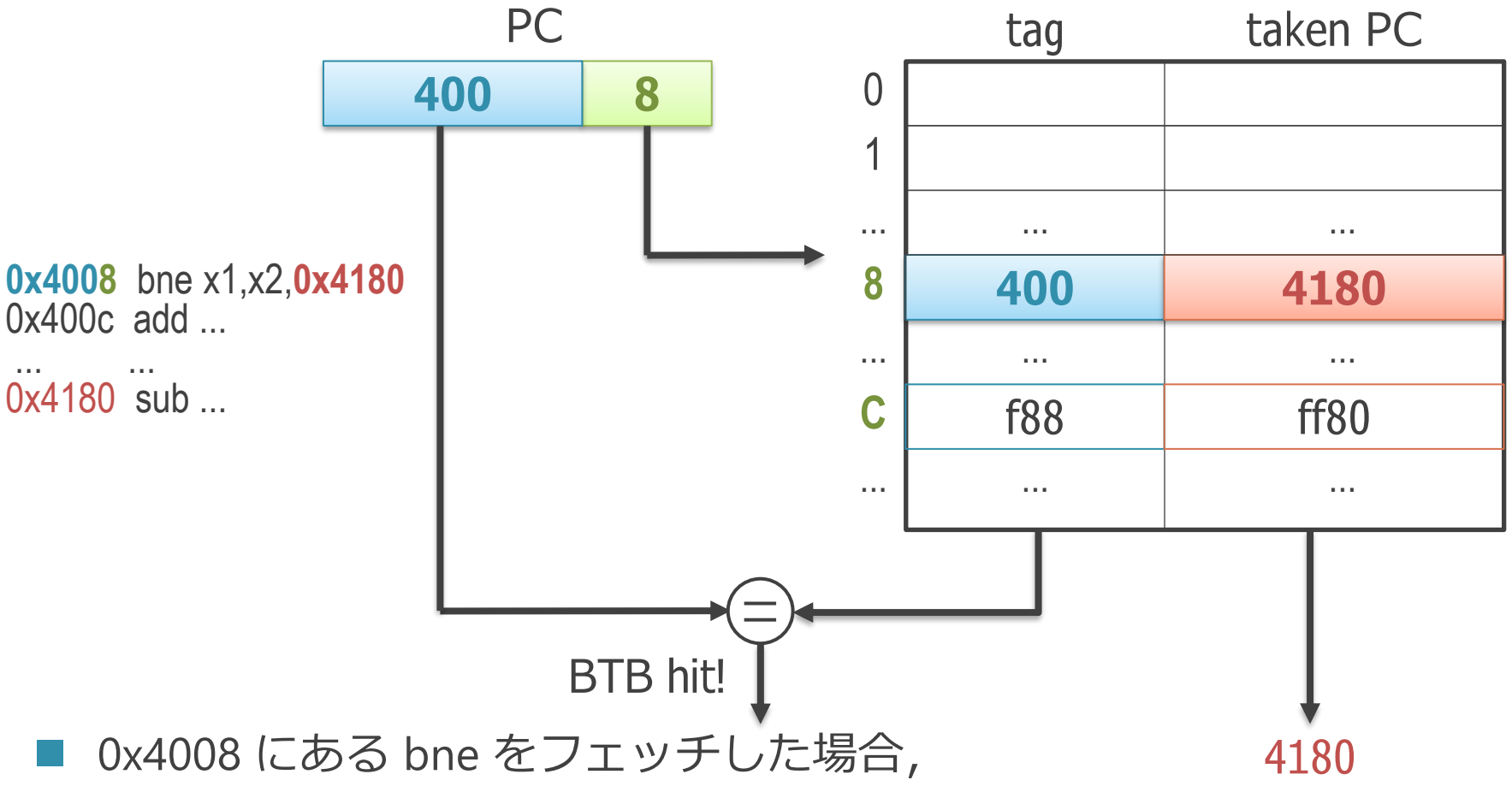
- 分岐命令の実行時に，この表にターゲットを登録しておく
  - ◇ 次回からは，表をひくとターゲットがとれる

BTB（Branch Target Buffer）による予測



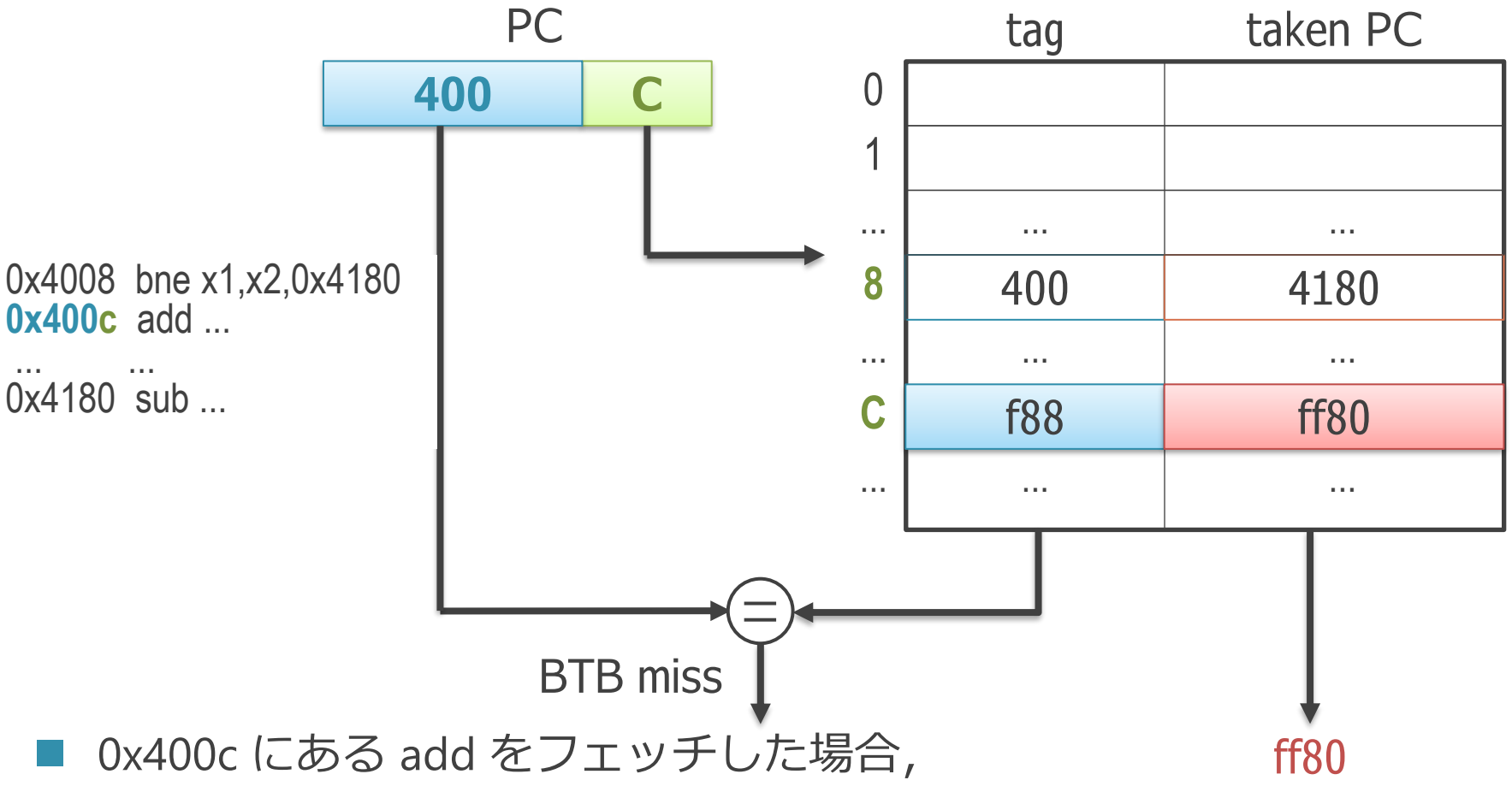
- 分岐かどうかと，分岐先ターゲットを同時に予測

BTB による予測（分岐命令の場合）



- 0x4008 にある bne をフェッチした場合，
  1. 0x4008 の下位の 8 を取り出し，BTB の 8 番エントリにアクセス
  2. 得られた tag と PC の上位の 0x400 が一致したのでヒット
  3. 0x4008 は分岐命令で，そのターゲットは 0x4180 と予測

BTB による予測（分岐以外の場合）



- 0x400c にある add をフェッチした場合，
  1. 0x400c の下位の c を取り出し，BTB の c 番エントリにアクセス
  2. 得られた tag と PC の上位が不一致なのでミス
  3. 0x400c は分岐命令ではないと予測

BTB の特徴

- 高速にアクセスする必要がある
  - ◇ １サイクル以内に処理が完結する必要がある
  - ◇ そうしないと，毎サイクル命令フェッチができない

1. エントリ数は比較的小さい
   - ◇ 最大でも数Kエントリ程度
   - ◇ 最近はそうでもなくなってきた（後述
2. ハッシュ表としては，かなり単純な構造を持つ
   - ◇ ハッシュ関数は，アドレスの一部を切り出してそのまま使う
   - ◇ 表の各エントリは固定長（古いものは上書きされる）

BTB の詳細

- 階層化された BTB を持つ場合もある
  - AMD Zen：L1+L2 BTB
  - ARM Cortex A72：64エントリL1 + 2Kエントリ L2 BTB
- L2 BTB アクセスには数サイクルかかる
  - L2 BTB により分岐が判明した場合，予測ミスからの回復が行われる

分岐かどうか&分岐先ターゲット予測のまとめ

- 分岐かどうか & 分岐先ターゲットを予測する必要がある
  - 命令をデコードするまでは，それらがわからない
- BTB を使った予測
  - BTB：機能的にはハッシュ表
    - 入力：予測対象の PC
    - 中身：分岐先ターゲット
  - フェッチ時は，まず BTB にアクセスして分岐かどうかとターゲットを予測

分岐予測

- 分岐予測
  1. 分岐命令かどうか予測
  2. 分岐先ターゲット予測
  3. **分岐方向予測**

分岐方向予測

- 以降，「分岐予測」と言った場合は「分岐方向予測」の意味に
- 以下の2つに大きく分けられる
  1. 静的分岐予測
  2. 動的分岐予測

静的分岐と動的分岐

```
// 10回まわるループ
i1:      li  x1 ← 0          // x1 を 0 に初期化
    L:
i2:      add x1 ← x1 + 1     // x1 をインクリメント
i3:      bne x1 != 10, L     // x1 が 10 でなければ L に飛ぶ
```

- 静的分岐：
  - ◇ プログラム内に書かれている分岐命令のこと
  - ◇ 上のコードでは，１つの静的分岐（i3）がある
- 動的分岐：
  - ◇ 実行中に現れる分岐命令のこと
  - ◇ 上のコードが実行された場合，i3 は 10 回実行される
  - ◇ = 10個の動的分岐がある
- 同様に，静的命令や動的命令という場合もある

分岐方向予測

- 以下の2つに大きく分けられる
  1. 静的分岐予測
     - 静的分岐に対する予測
     - プログラム開始時に予測結果は決まっており，実行中に予測結果は変化しない
  2. 動的分岐予測
     - 動的分岐に対する予測
     - プログラムの実行中に予測結果が変化する

分岐方向予測

- 分岐予測
  1. **静的分岐予測**
     1. 常に不成立と予測
     2. 前方分岐を不成立/後方分岐を成立と予測
     3. プロファイルによる予測
  2. 動的分岐予測

1. 常に不成立と予測

- 今の PC に対し，次の PC を常に読む
- あまり精度は良くない
  - ◇ 統計的に，大体 70% ぐらいの分岐命令は成立する
  - ◇ したがって，予測ヒット率は 30% ぐらい
- 最も単純で，予測のために特に追加のハードを必要としない
  - ◇ 古い CPU では実際にこれを搭載していたものも結構ある

2. 前方分岐を不成立/後方分岐を成立と予測

後方分岐

```
// 10回まわるループ
i1:      li  x1 ← 0          // x1 を 0 に初期化
    L:
i2:      add x1 ← x1 + 1     // x1 をインクリメント
i3:      bne x1 != 10, L     // x1 が 10 でなければ L に飛ぶ
```

- 統計的に，後方分岐は成立することが多い
  - ◇ ループを構成することが多く，繰り返し実行される
  - ◇ 典型的には 80% 以上が成立
- 前方分岐を不成立/後方分岐を成立
  - ◇ 前方分岐はコストを重視して，常に不成立と予測
  - ◇ 後方分岐は常に成立と予測

3. プロファイルによる予測

- 予測方法
  1. 分岐方向のプロファイルをとる
     - 事前にプログラムを実行して，静的分岐の方向の統計をとる
     - 「このアドレスの分岐命令は，大概成立 or 不成立」
  2. プロファイル結果に基づき，命令にヒントを埋め込む
     - 成立 or 不成立 の傾向を命令コードに埋め込んでおく
     - コンパイラにより行う
     - 命令セットのレベルで対応が必要
  3. CPU は命令内に埋め込まれたヒントに基づき予測

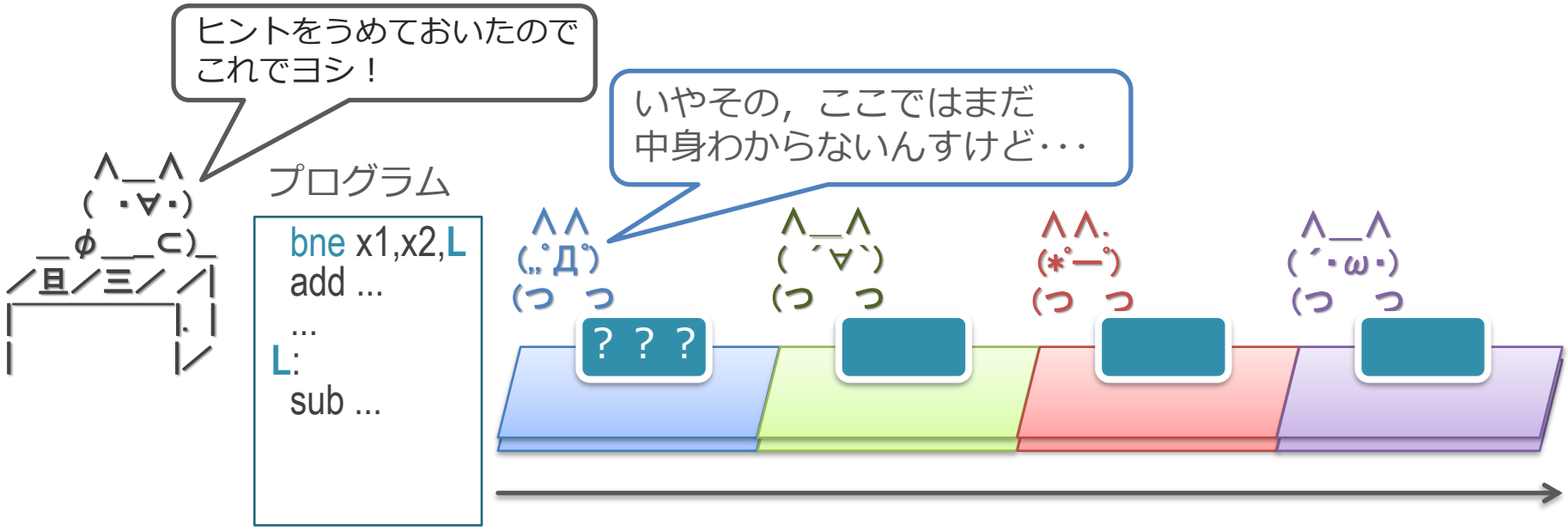
3. プロファイルによる予測

- そこそこの精度が出る
  - 静的分岐命令１つ１つの傾向が反映できる
    - 後方分岐だけど不成立が多い... とかに対応できる
  - 予測精度はだいたい 80% から 90% ぐらい

静的分岐予測の欠点

1. 分岐方向が毎回変わるようなものには本質的に対応できない
   - ◇ 例：同じ静的分岐で成立と不成立が交互に起きる
2. プロファイル時と挙動が異なる場合に対応出来ない
   - ◇ オプションや入力に応じてプログラムの挙動が大きく場合など
3. 意外とハードウェア・コストが安くない
   - ◇ 方向そのものの予測にはハードは必要がない
   - ◇ 成立すると予測する場合，BTB が別途いる
     - □ 分岐かどうか & 先ターゲット予測は必要
   - ◇ 「後方分岐かどうか」の予測や，
     「成立/不成立のヒント」の予測を行う必要がある

「後方分岐かどうか」「成立/不成立のヒント」の予測



- フェッチされた命令は，デコードするまでは以下がわからない
  1. 分岐命令かどうか？
  2. 分岐ターゲットはどこか？
- 同様に，
  - ◇ 「後方分岐かどうか」「成立/不成立のヒント」もわからない

別途ハードウェアが必要

- 「後方分岐かどうか」「成立/不成立のヒント」もわからない
  - ◇ 方向そのものを直接は予測しない
  - ◇ しかし，かわりに「後方分岐かどうか」等を予測する必要がある
- 別途それらを表に学習する？
  - ◇ 後述の動的分岐予測とあまりかわらない機構が必要

静的分岐予測のまとめ

- 静的な命令に対してあらかじめ予測
- 基本的に，今の CPU では使われていない
  - ◇ 予測精度の上限に限界がある
  - ◇ 意外とハードウェア・コストが安くない
- 次回は動的分岐予測

出欠と感想

- 本日の講義でよくわかったところ，わからなかったところ，質問，感想などを書いてください
  - ◇ LMS の出席を設定するので，そこにお願いします
  - ◇ パスワード:
- 意見や内容へのリクエストもあったら書いてください
- LMS の出席の締め切りは来週の講義開始までに設定してあります
  - ◇ 仕様上「遅刻」表示になりますが，特に減点等しません
  - ◇ 来週の講義開始までは感想や質問などを受け付けます