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

感想とか質問とか

- 浮動小数点は正規化処理が重いとのことでしたが、IEEE 754以外に精度などを妥協するような実数表現は考えられていたりするのでしょうか？
- また、分岐予測ではその後の計算を全て捨てているように見えましたが、マルチスレッディングなどによる分岐と無関係なデータも捨ててしまうのでしょうか？

感想とか質問とか

- 本題ではないですが、D-JOLTはバイオハザードネタですか…？
- 学部の頃にハザードがなぜ起こるのかぐらいは知っていましたが、その対策とそれがどのくらい遅延を起こすのかは理解していませんでした。対応関係を追って復習します。
- 現状、最も利用されているデータ・ハザードの解消方法はマルチスレッディングなのでしょうか。

感想とか質問とか

- マイクロ命令に分解するためのオーバーヘッドはどの程度になりますか？
- 前回の内容と比べると大分わかりやすかったです
- IF ID EX WBあたりの省略された単語がわからずorすぐに思い出せず困っています。

感想とか質問とか

- この辺の知識をもう一度整理して詰め直したいのですが何かおすすめの書籍ありますでしょうか？？"
  - 「パターソン&ヘネシー」
    - 「ヘネシー&パターソン」もあるが，こっちのが高度
  - 「コンピュータ・システム 」（ステマ）
    - https://www.amazon.co.jp/dp/B07RK18MFT

感想とか質問とか

- データハザードのマルチスレッディングが遅延スロットの上位互換のように思えました。
- マルチスレッディングの欠点にあまりぴんときませんでした。
- どれも欠点があるのですが、その中でもフォワーディングを選ぶ最大のメリットはなんなのでしょうか？

感想とか質問とか

- IntelのCPUには実は不具合が多いという話からの疑問ですが、"バグ"という表現は主にソフトウェアの不具合を指し、"エラッタ"はCPUの不具合などを指していることが多い気がします。"エラッタ"という言葉の明確な定義はあるのでしょうか？また、なぜCPUの不具合では"エラッタ"という言葉が使われるのでしょうか？

- 用語などが統一されないことに、企業の強気を感じました笑

感想とか質問とか

- パイプラインの可視化ツール面白いです．自分も使って見てみたいのですがどこかで公開していますか？
  - Konata 本体
    - https://github.com/shioyadan/Konata
  - ロボ太先生の記事
    - アーキテクチャシミュレータGem5を使ってみる
      - https://qiita.com/kaityo256/items/00fc50221d86ce3ff2ea
    - アーキテクチャシミュレータGem5を使ってみる　その2「OoOを実感する」
      - https://qiita.com/kaityo256/items/a0b79c7601c2cd10c2eb

感想とか質問とか

- Windows Updateでインテルのマイクロ命令のアップデートも細々と行われていたりする，という話がありましたが，AMDのCPUを用いてPCを動かしている場合はどうなるのでしょうか
- 制御ハザードに関して、予測した分岐結果を複数実行して正しいと判明したものを採用するみたいな方法は、実際つかわれないのでしょうか？（分岐命令が多いこと、予測がほぼ当たることが問題？）
- if文の分岐先の計算量を減らすと制御ハザードによるスループットの低下の影響を少し抑えられるのかなと思いました

感想とか質問とか

- 自分で本で読んで理解していたつもりだったが、講義を聞くことで間違った認識をしていた箇所があったことに気がつけてよかった。

- 前回，「深層学習で徐々に早くなる…」と質問をしたものですが，ある処理をLループするとき，（例えば）1ループ目の処理時間よりも10ループ目の処理時間が短くなるのはなぜか，という意図で質問をしました．（どこで齟齬が生まれているのかがわかっていません…）

感想とか質問とか

- 今から(趣味で)CPUを1から作るなら、パイプライン化のことも見越して設計したほうがいいのでしょうか
- アセンブリ書いたりしていてメモリ間movとか中身どうなってんだろうと思っていたが、思ったより普通だとわかった
- 今は主流でない技術も突然脚光を浴びたりするのでCISCとかクロック同期しないやつとか何かのタイミングで復活してより高性能なものが作られたりしたら面白いなと思った

感想とか質問とか

- 以前分岐予測"ペナルティ"という語を初めて目にした時、"ペナルティ"というだけあって後から与えられるもののように感じてしばらく意味が分かりませんでした
- IntelのCPUですらかなりバグが出るという話がありましたが、修正されたバグには具体的にどのようなものがあるのでしょうか？

感想とか質問とか

- 分岐予測の詳細を聞いてからの方が適切な質問かもしれませんが、実際のCPUでどのぐらい分岐予測が失敗するのかが知りたいです。

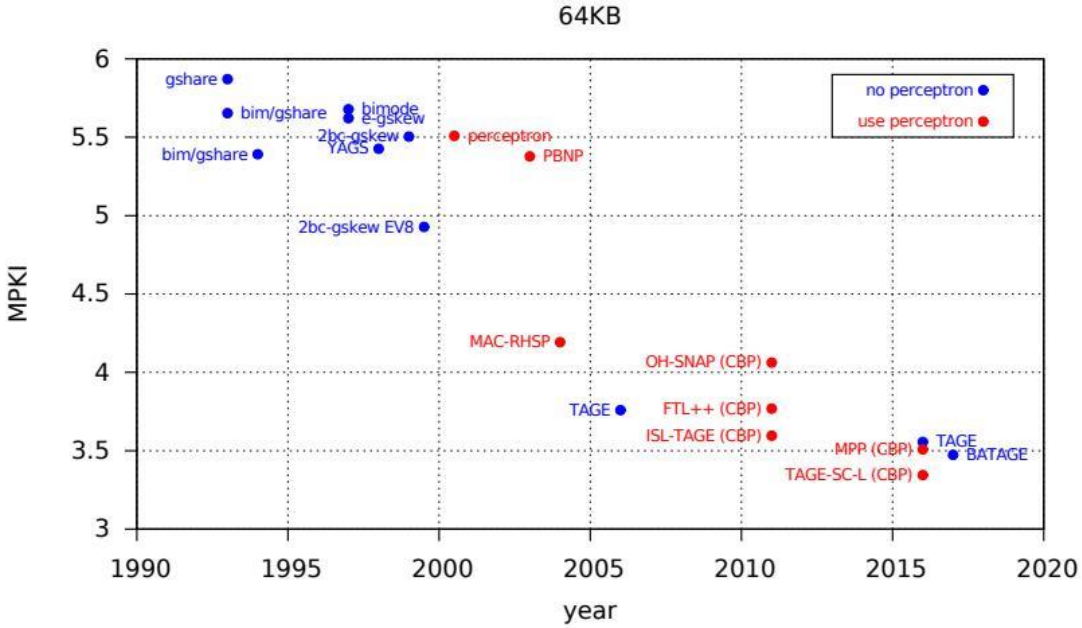


Figure 1: Average number of mispredictions per 1000 instructions (MPKI) for various conditional branch predictors on the CBP 2016 traces for 8KB, 32KB and 64KB storage budgets (see Appendix A).

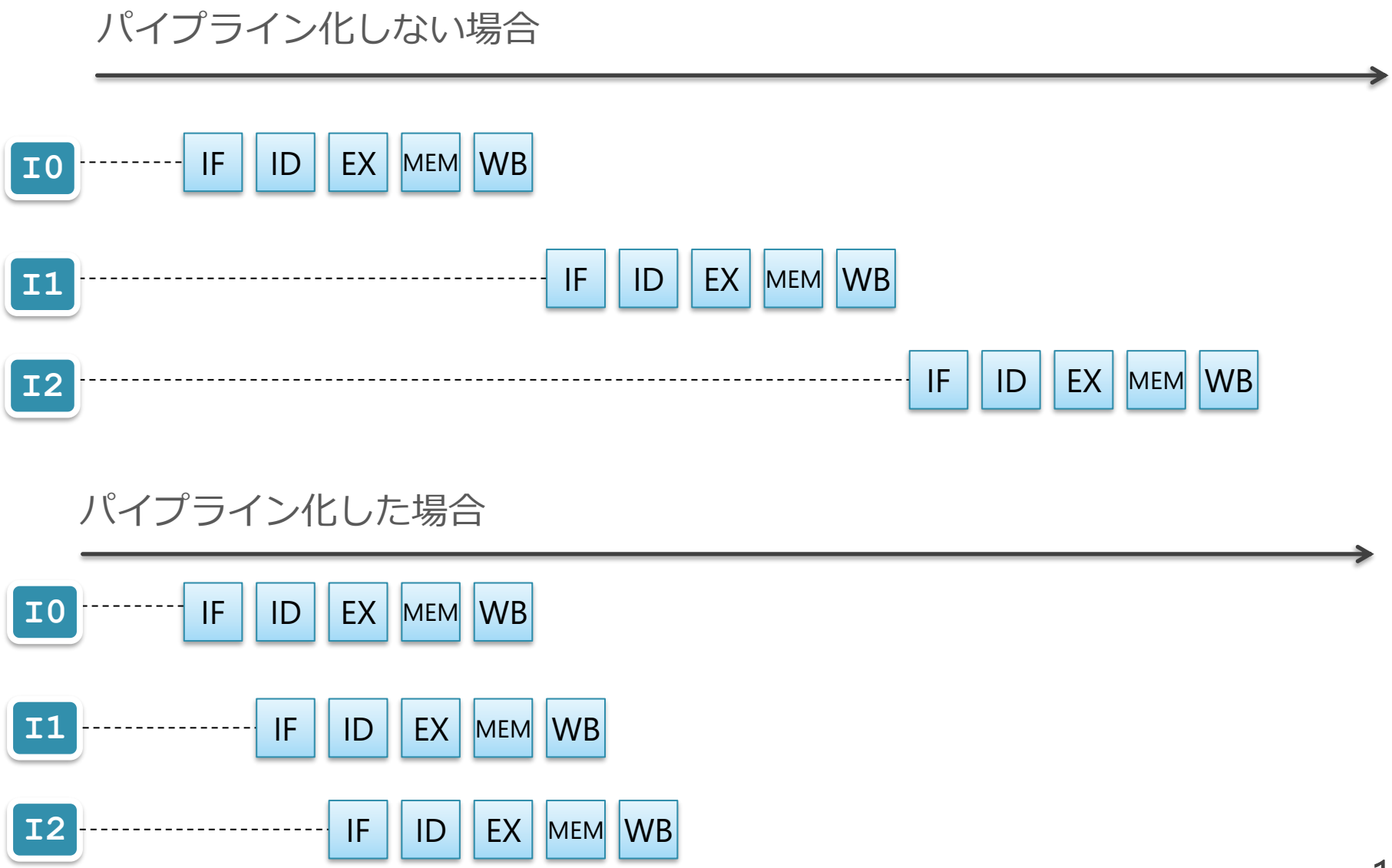
前回の内容

1. 命令パイプラインの詳細
2. ハザードとその解決方法

今日の内容

1. 命令パイプラインと性能
2. 分岐予測（前編）

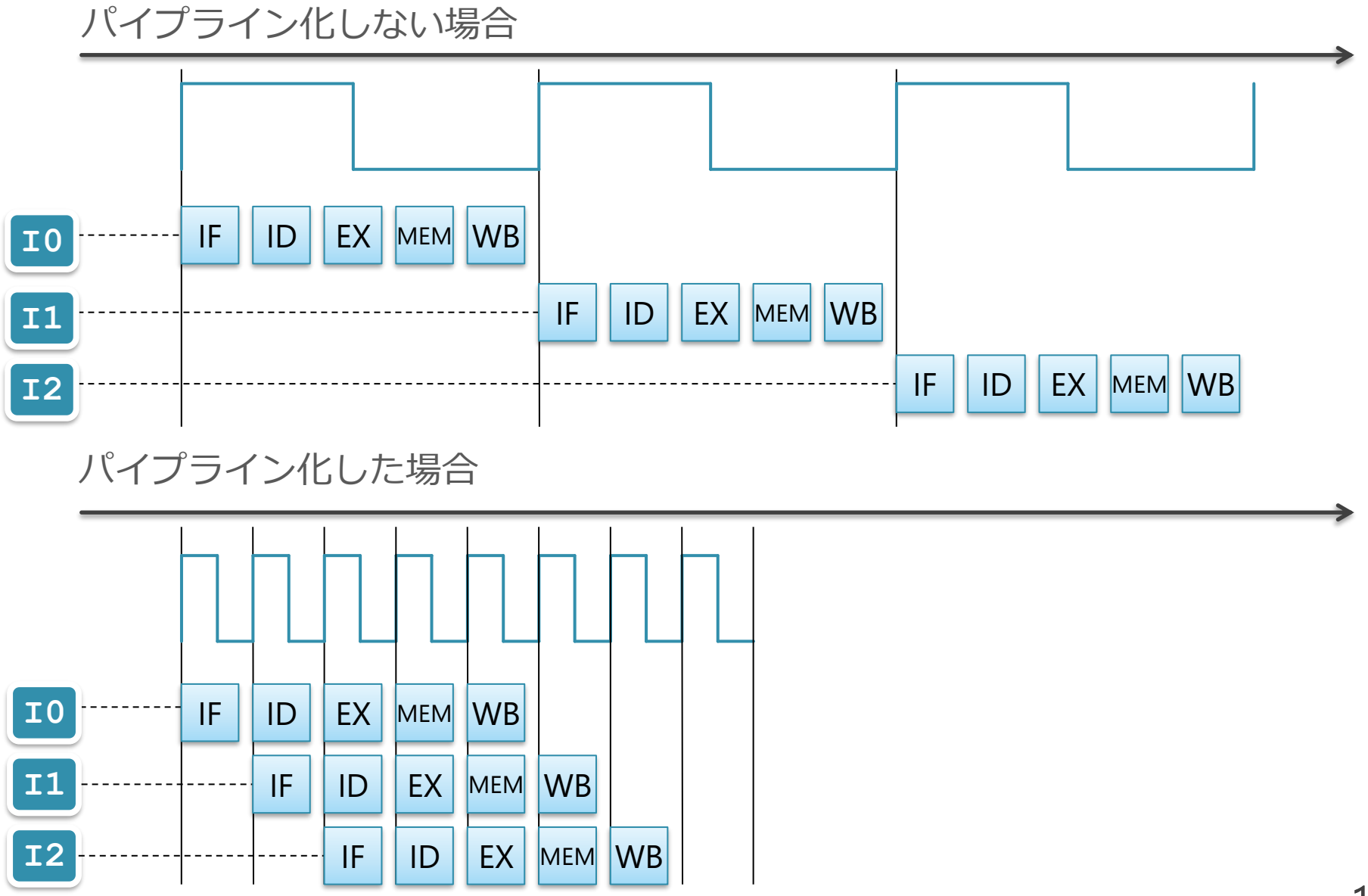
パイプライン化によるスループット向上



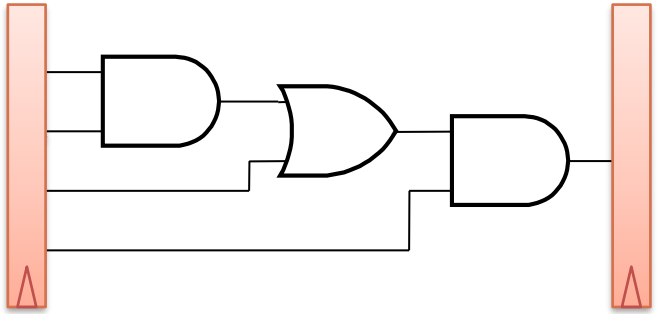
パイプライン化の意味

- 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

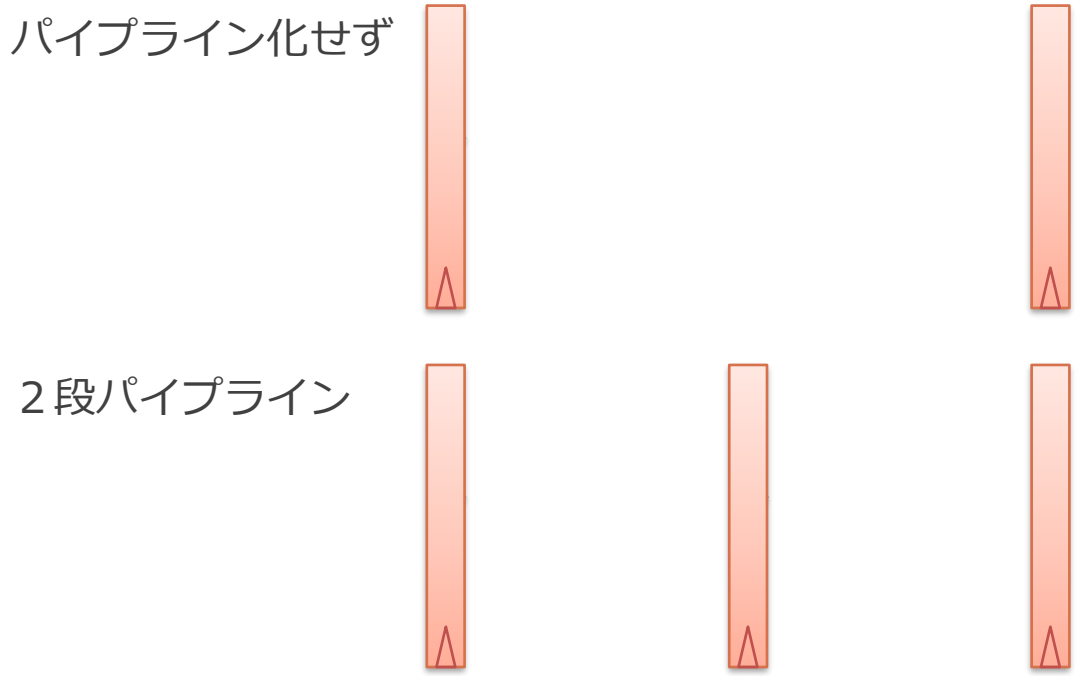


ステージ内の信号の伝播を考える



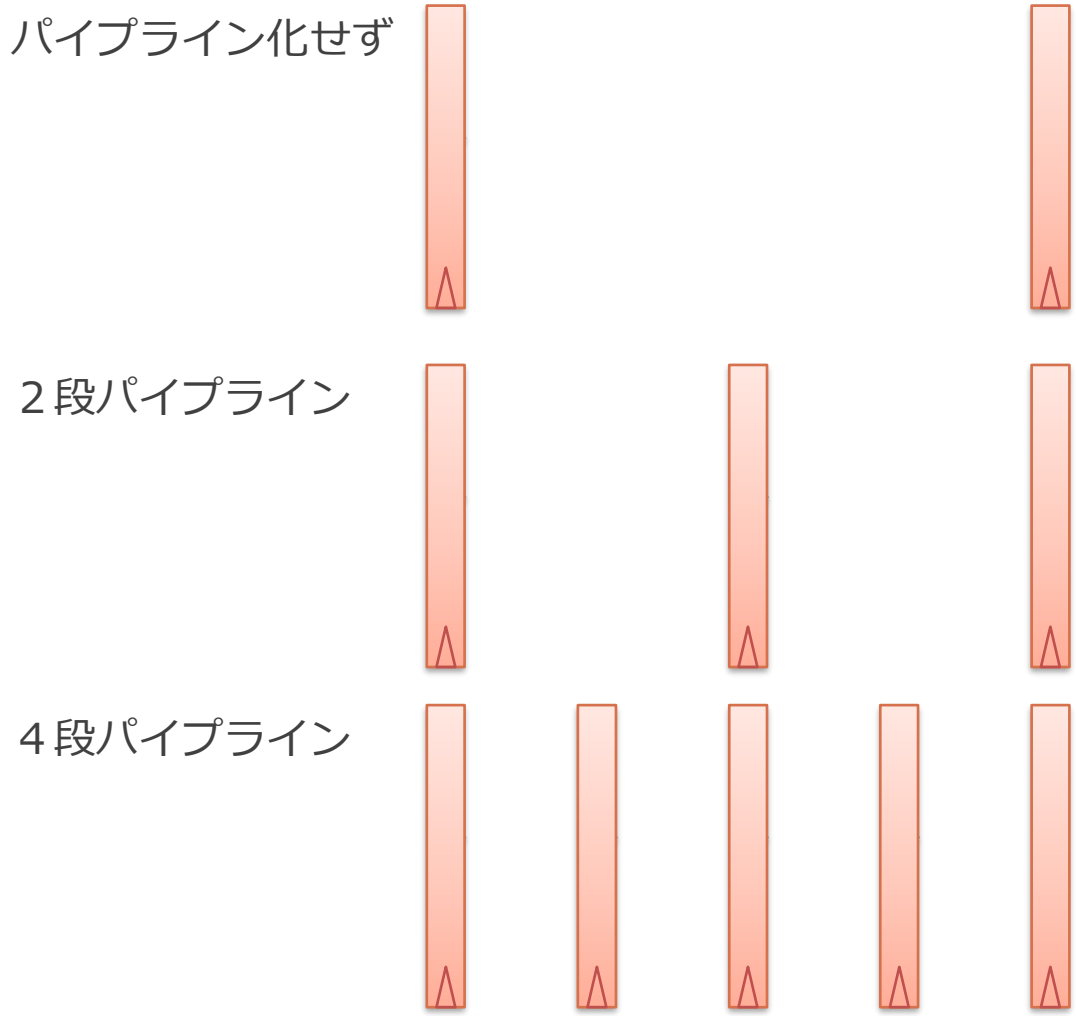
- パイプライン：ステージ：
  - ◇ 複数のパイプライン・ラッチで挟まれてた，
  - ◇ 組み合わせ回路（ゲート）
- 矢印の動き：
  - ◇ クロック開始時に，左のラッチからでた信号が
  - ◇ 組み合わせ回路を通って，伝播していく様子

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

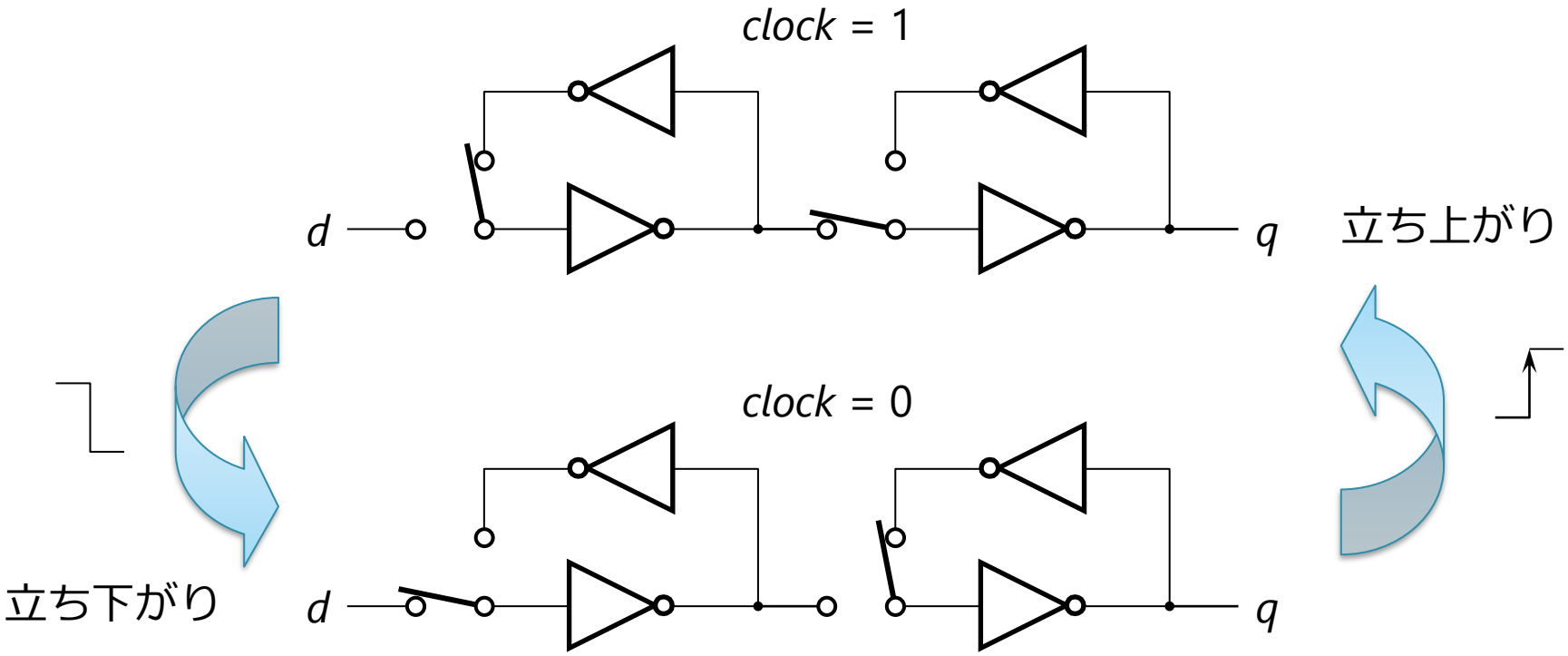
- パイプライン段数を増やしていけば，どこまでも速くなるのか？
  - ◇ ならない
- 理由：
  1. 回路的な理由による周波数向上の限界
  2. アーキテクチャ的な理由による実効性能の限界

回路的な理由

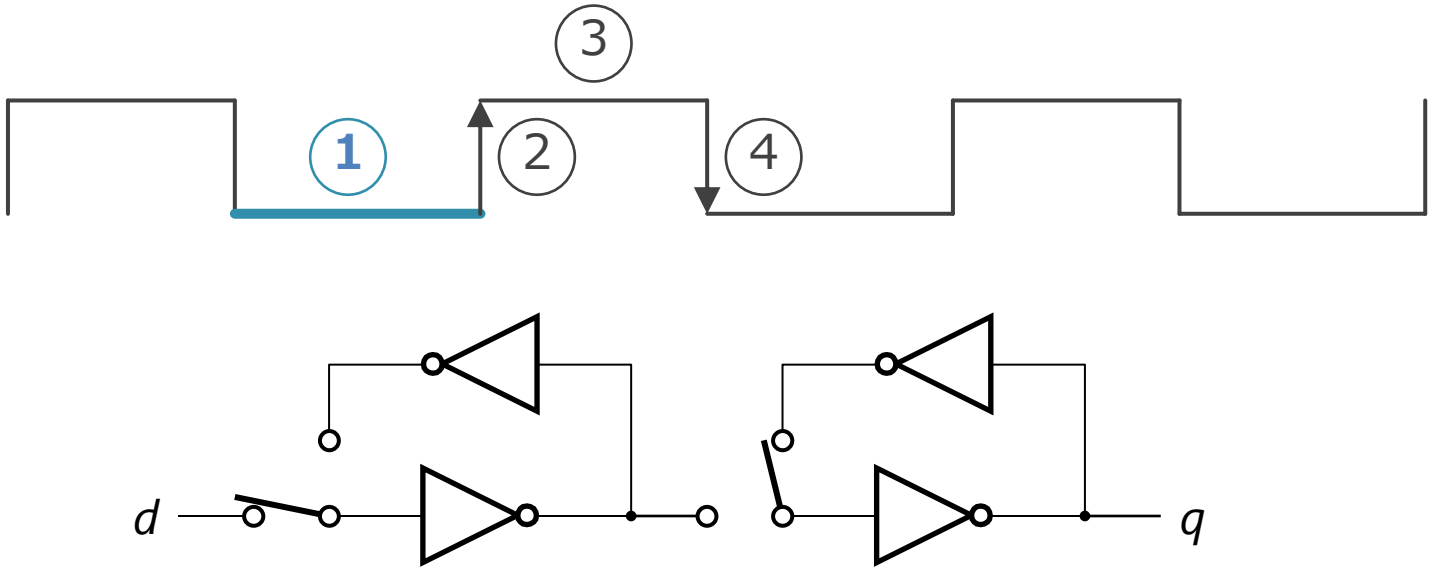
- 理由：
  1. D-FF 自体の遅延のため
  2. 消費電力と熱のため

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

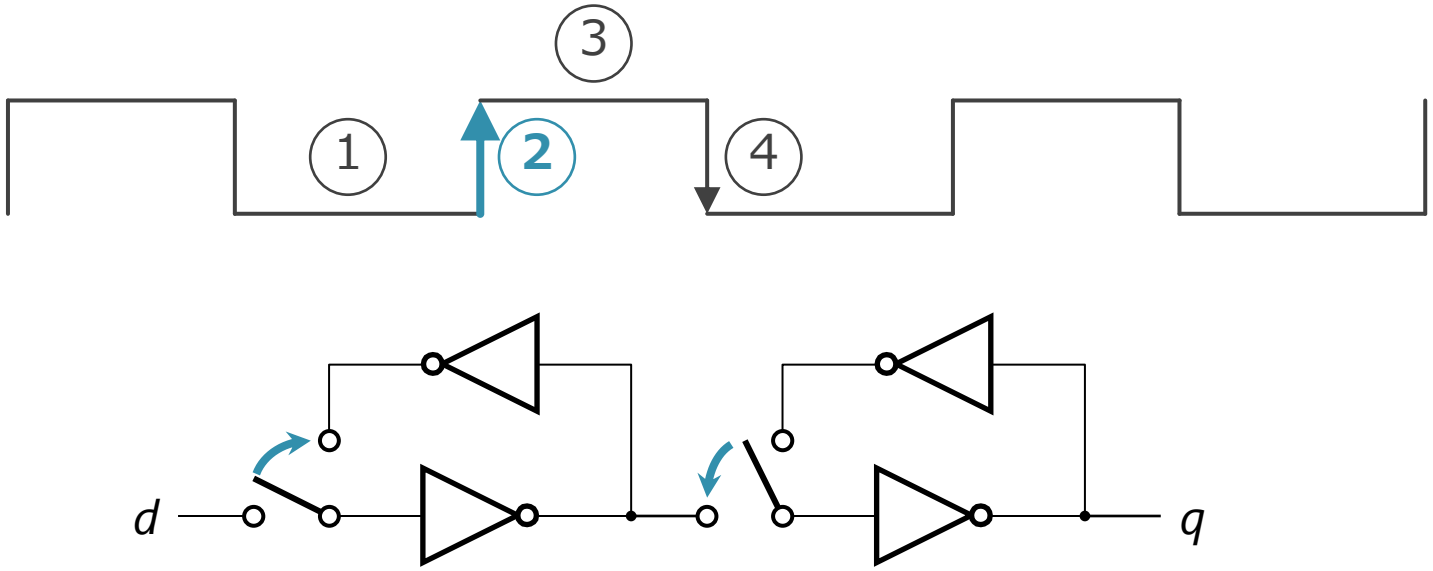


D-FF の動作 ① クロック信号が Low にあるとき



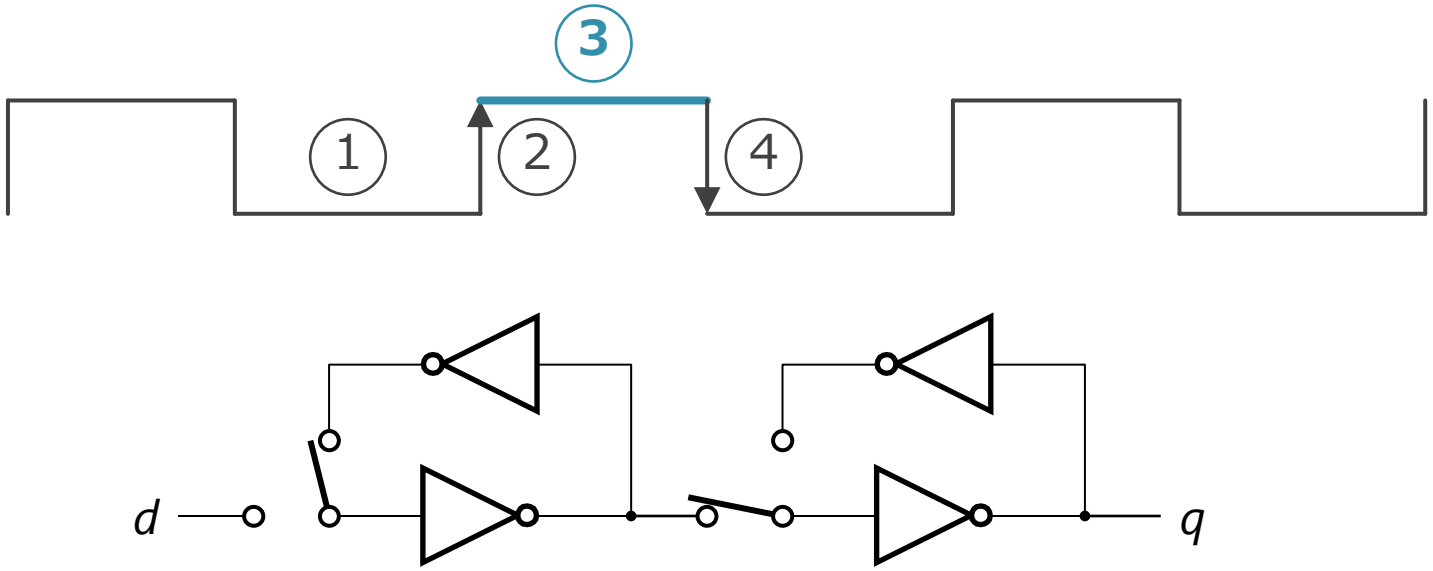
- 左側のループ：
  - ◇ d の入力の変化に応じて，インバータの状態が随時切り替わる
  - ◇ 右側のループとは遮断されている
- 右側のループ：
  - ◇ ループのインバータの状態（=記憶）が q に出力され続ける

D-FF の動作 ② クロック信号の立ち上がり



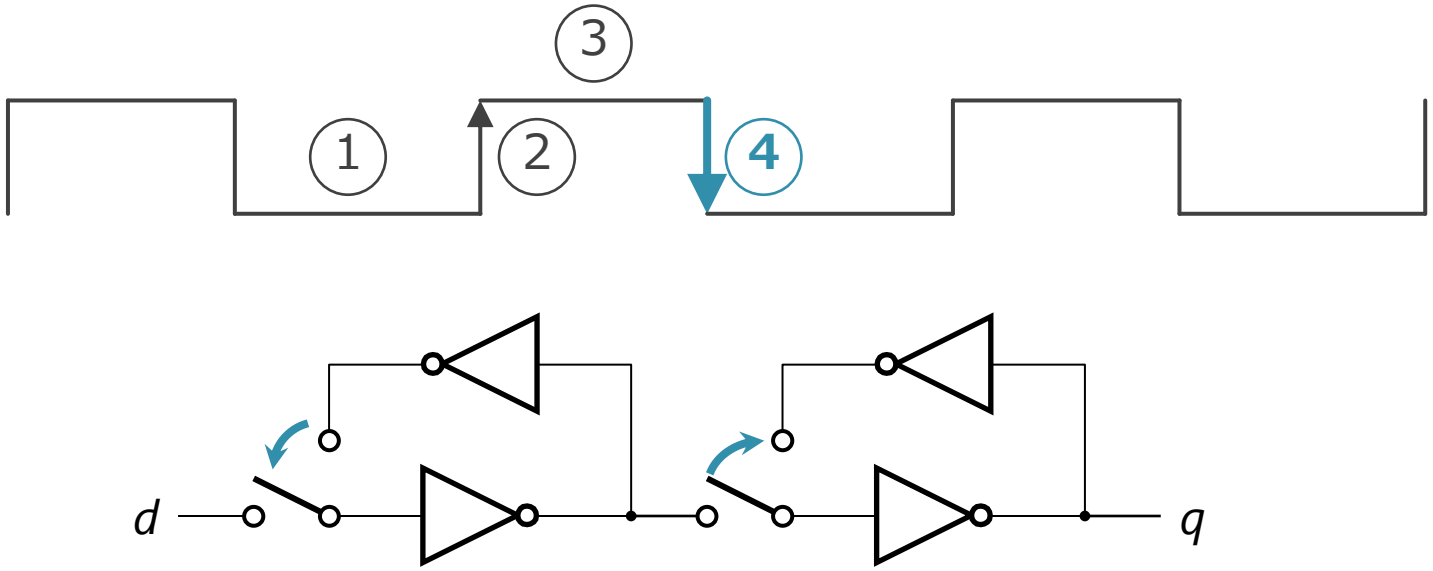
- 左側のループ：
  - ◇ *d* と遮断され，ループが形成される
  - ◇ 直前まで *d* に入力されていた信号が記憶される
- 右側のループ：
  - ◇ 左側のループと繋がり，ループが解除される

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ステージ内の遅延：インバータ換算で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

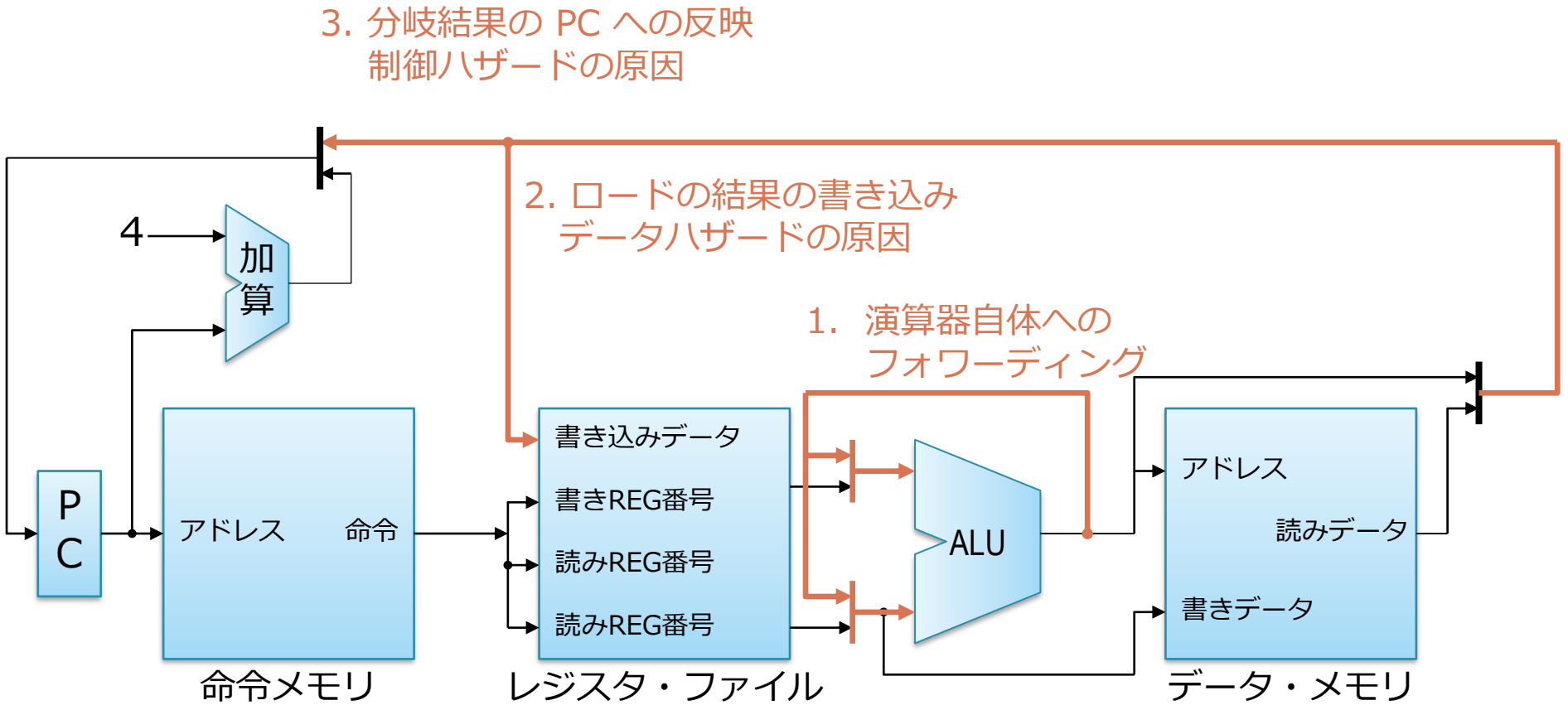
パイプライン化の限界

- 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

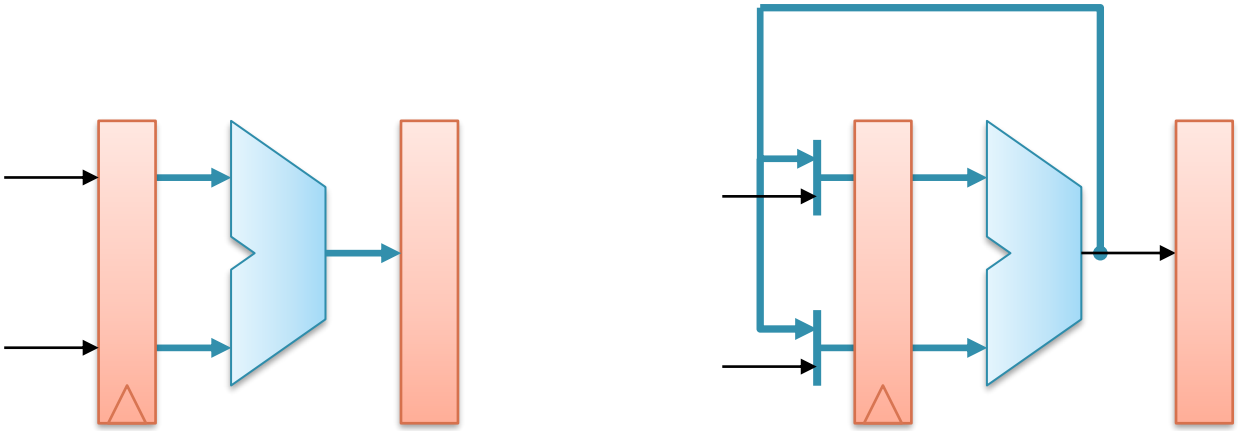


- バックエッジがあるため，命令を単純に流せない場合がある
  - ◇ 工場のラインのように，一方向に流せない

問題となるバックエッジ

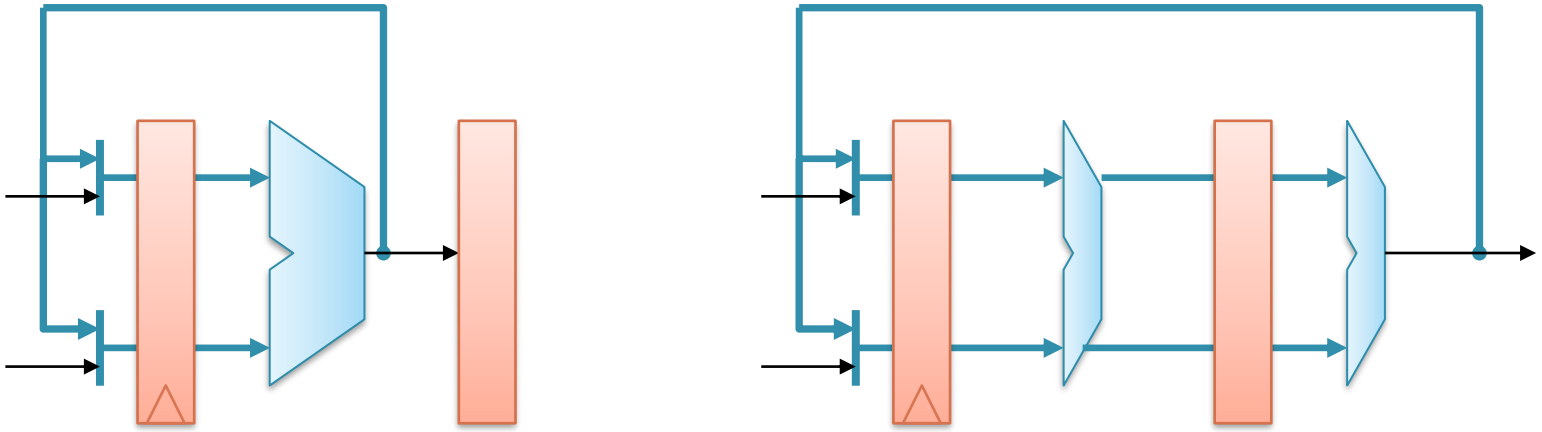
1. 演算器のフォワーディング
2. ロードによるデータ・メモリの読み出し
3. 分岐結果の PC への反映

1. 演算器のフォワーディング



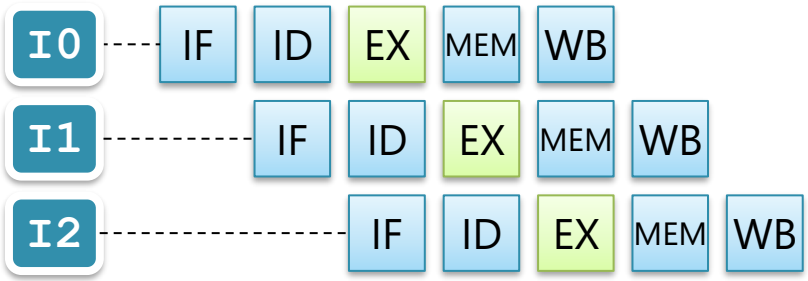
- ここは結構遅延が長い
  - ◇ クロック周波数の低下につながるのでパイプライン化したくなる

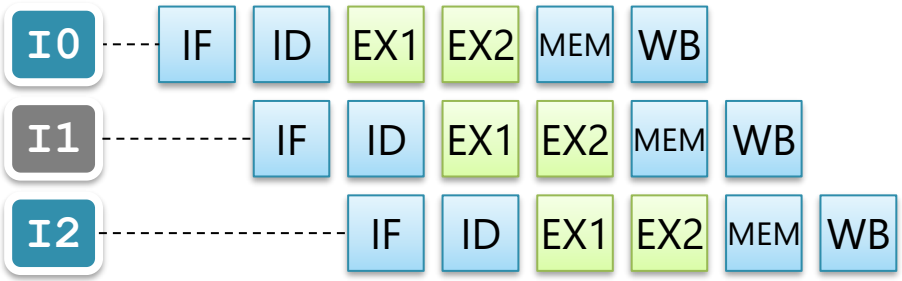
演算器のパイプライン化



- 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



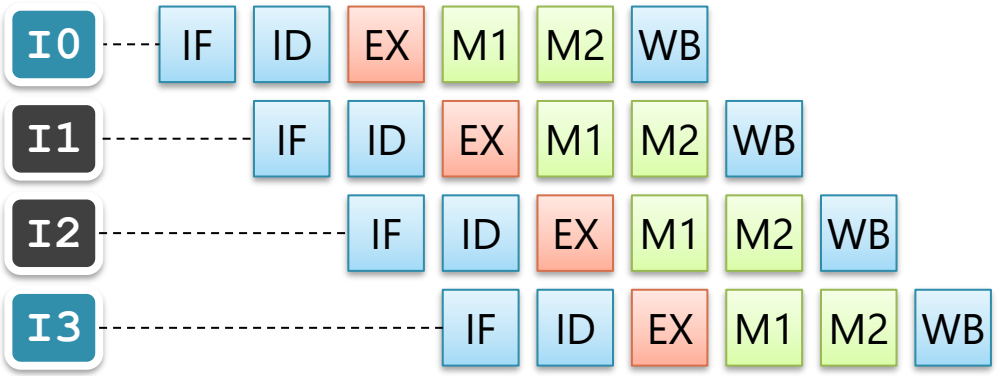


- 依存関係にある命令を連続して実行できなくなる
  - `I0` の EX2 が終わる前に，`I1` の EX1 が始まる
  - もし，他に実行するべき命令がおけなければ，遊ばせとくしかない
- 場合によっては性能が返って下がる
  - 周波数が上がったが，２サイクルに１回しか命令が実行できない
- 基本的な整数演算はパイプライン化せず１ステージを死守するのが普通
  - 乗除算や，浮動小数点演算はあきらめてパイプライン化

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

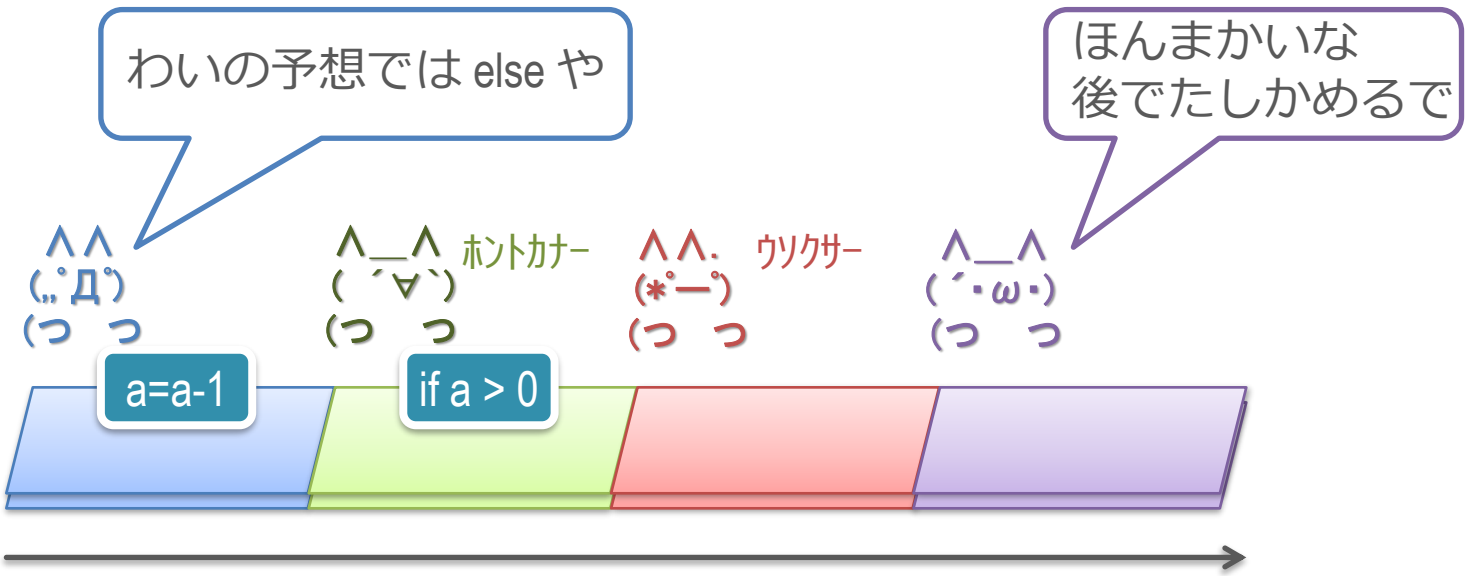
問題となるバックエッジ

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
```

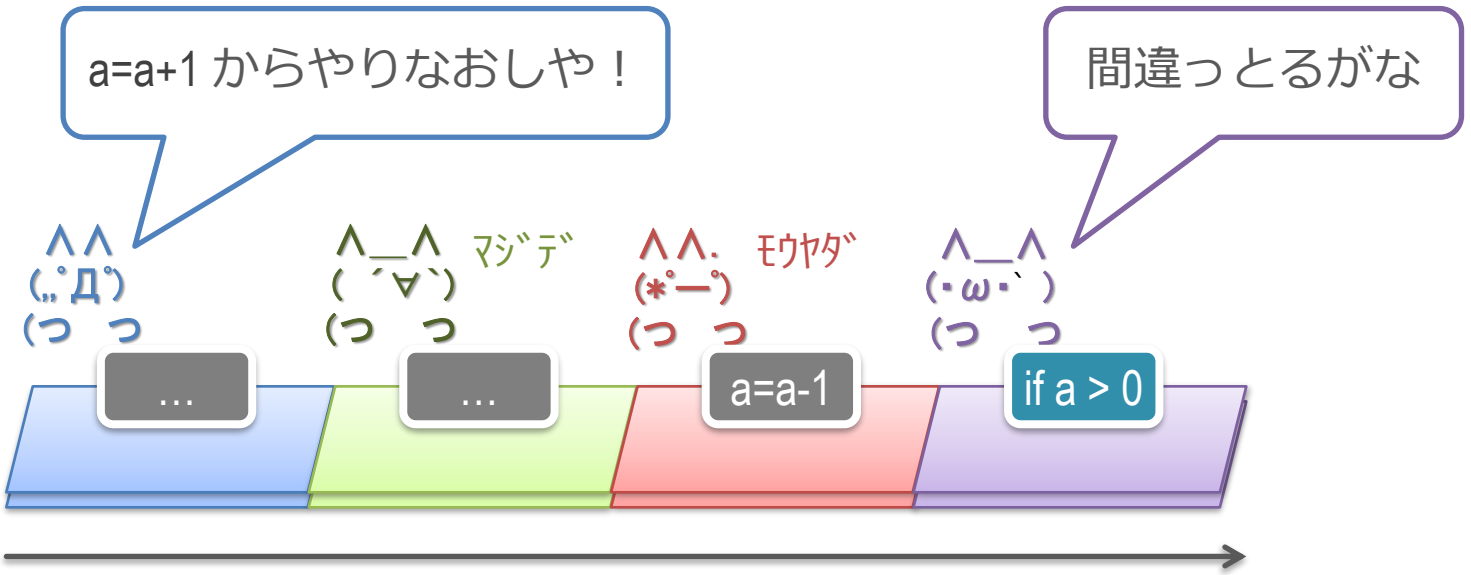


- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する

分岐予測ペナルティ

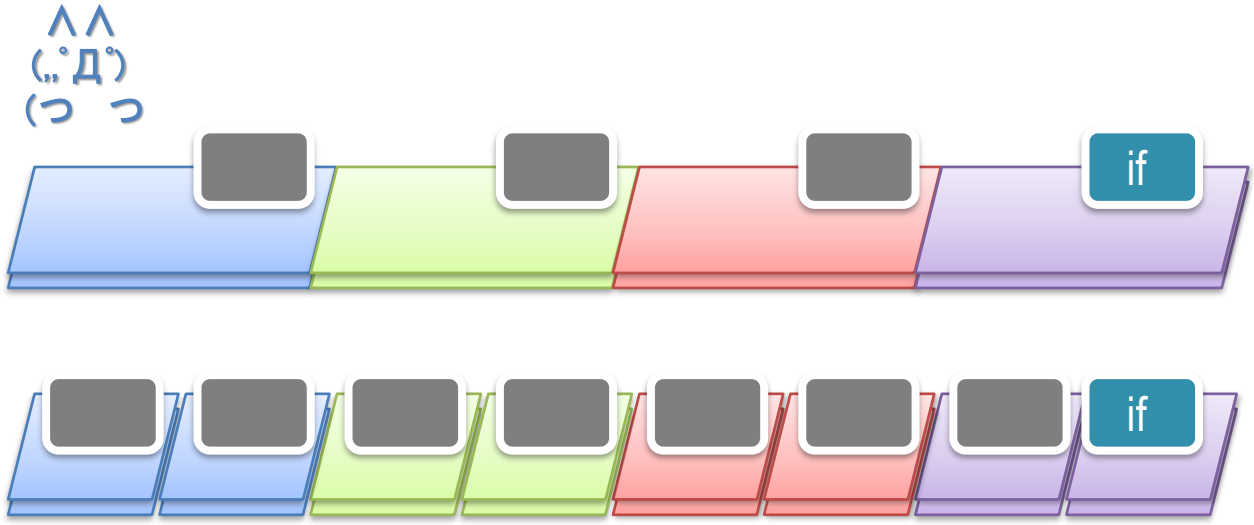
プログラム

```
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

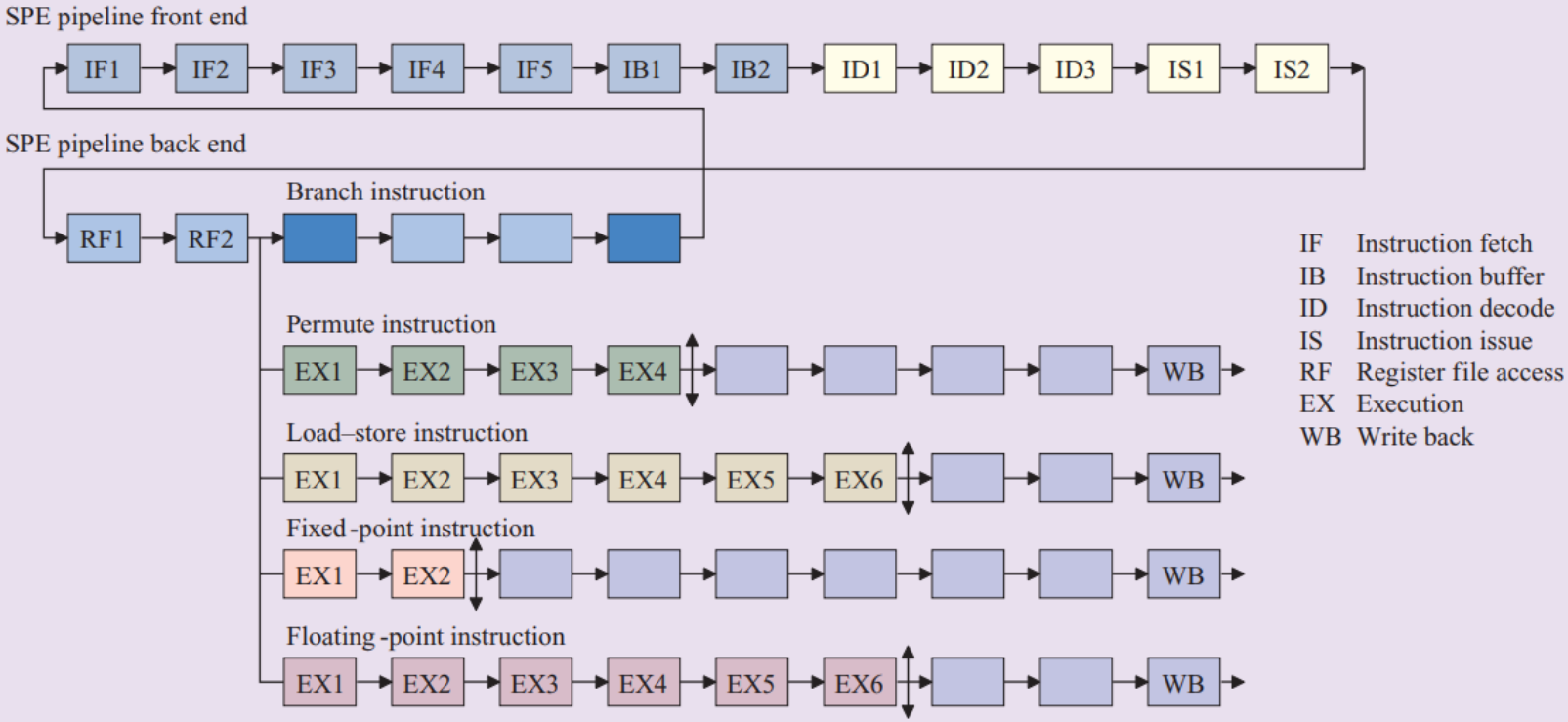
- 速度が上がらなくなる理由：
  - 回路的な理由による周波数向上の限界
    - D-FF の遅延
    - 電力と熱
  - アーキテクチャ的な理由による実効性能の限界
    - バックエッジによる実効性能の低下
    - （今日話した話題意外に，スーパスカラ固有の性能低下も

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

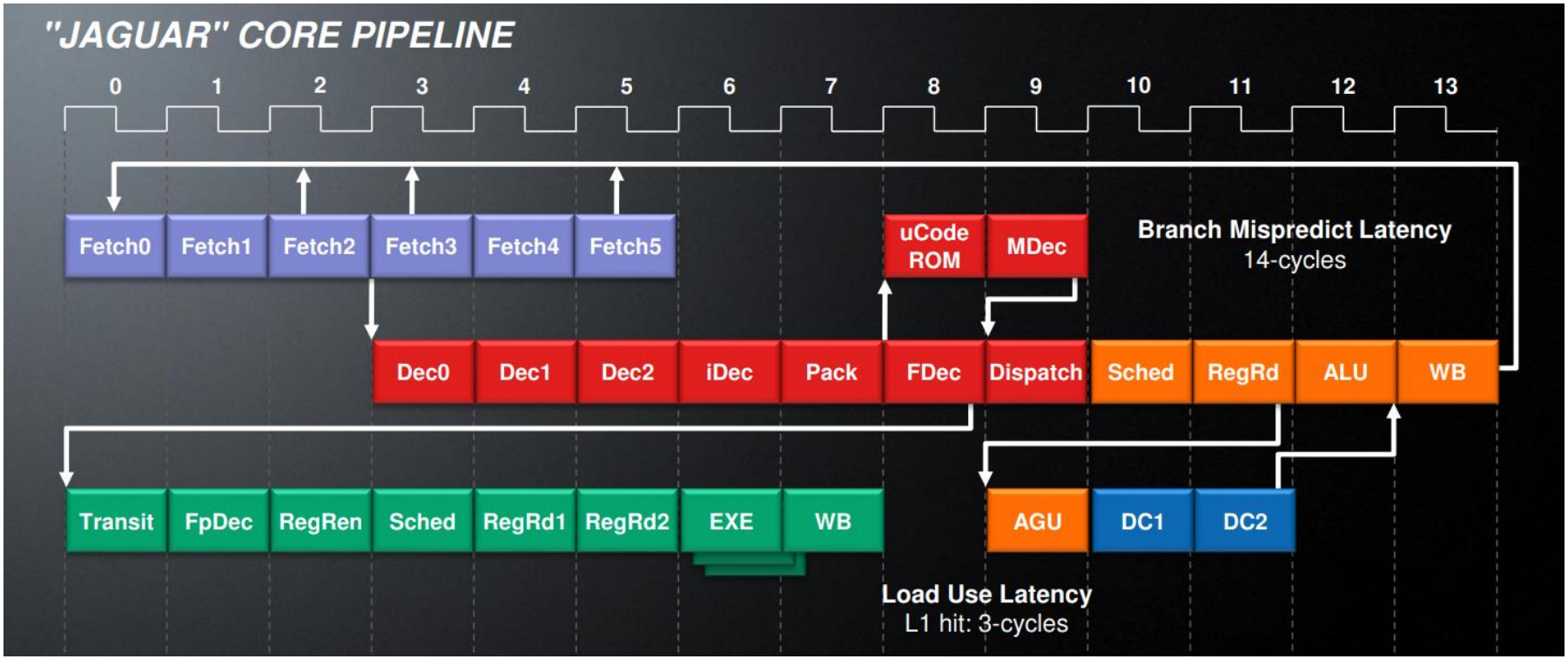
Sony/IBM/東芝 Cell (SPE)

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



AMD JAGUAR

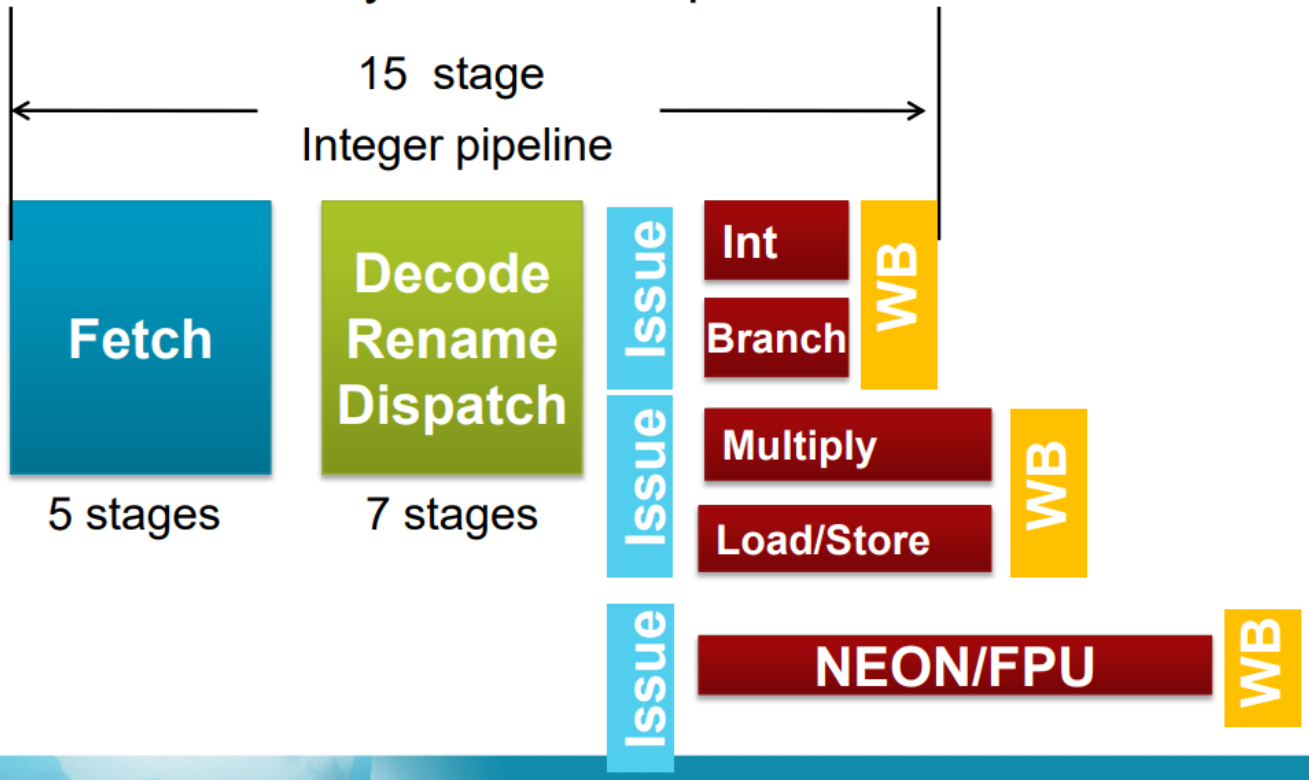
"JAGUAR" AMD's Next Generation Low Power x86 Core より



Cortex-A15 Pipeline Overview

15-Stage Integer Pipeline

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



今日の内容

1. 命令パイプラインと性能

2. **分岐予測**
   - ◇ **用語の定義からはじめる**

用語の定義（１）

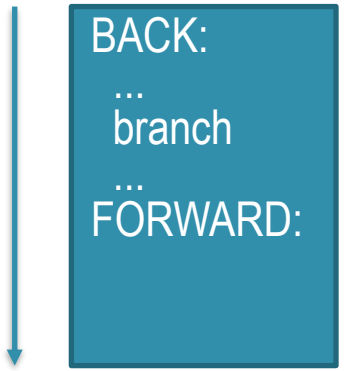
- 方向分岐
  - ◇ if 文のように，２方向に分岐する分岐命令
- 間接分岐
  - ◇ レジスタに格納されている値のアドレスに飛ぶ分岐命令
  - ◇ 任意の場所に飛ぶことができる

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）

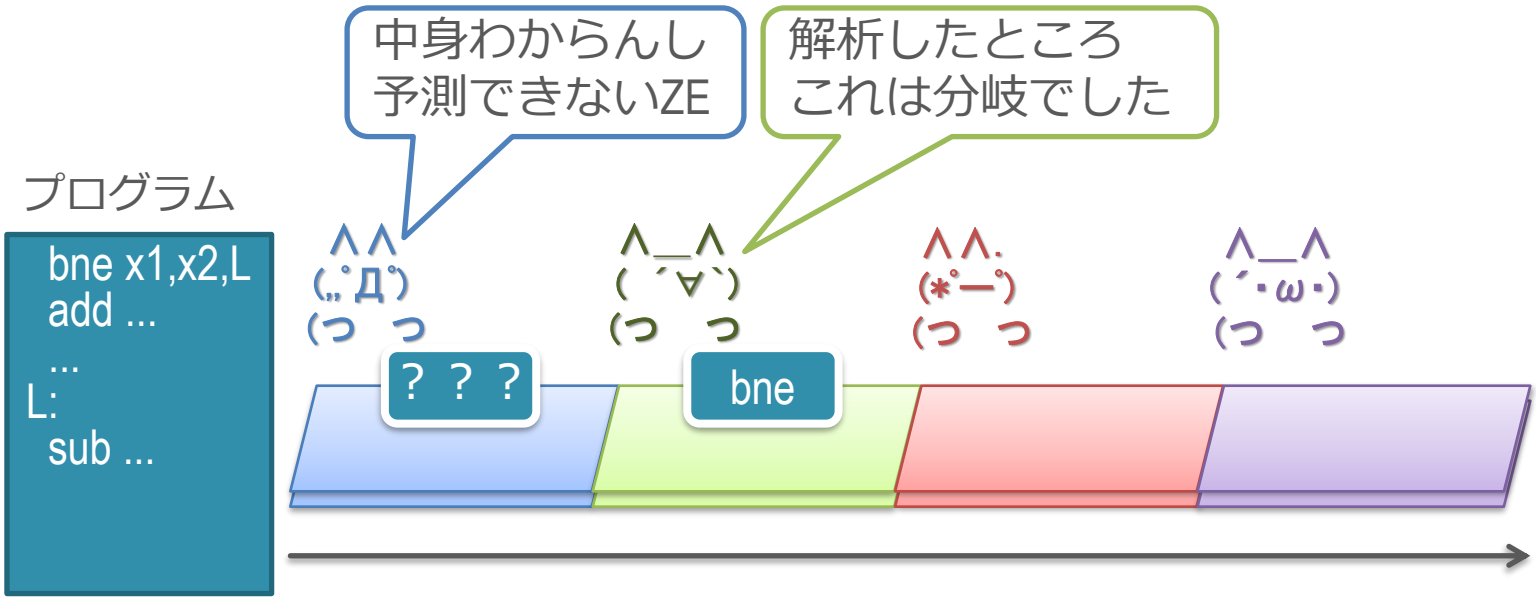
- 分岐先 アドレス or ターゲット
  - ◇ 分岐が成立した際の飛び先のアドレスのこと
- 前方分岐：
  - ◇ 分岐先ターゲットが分岐自身のアドレスよりも大きい分岐のこと
  - ◇ プログラムの進行方向に対して前方に飛ぶことから
- 後方分岐：
  - ◇ 分岐先ターゲットが分岐自身のアドレスよりも小さい分岐のこと
  - ◇ 後方に飛ぶ = ループを作る



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？

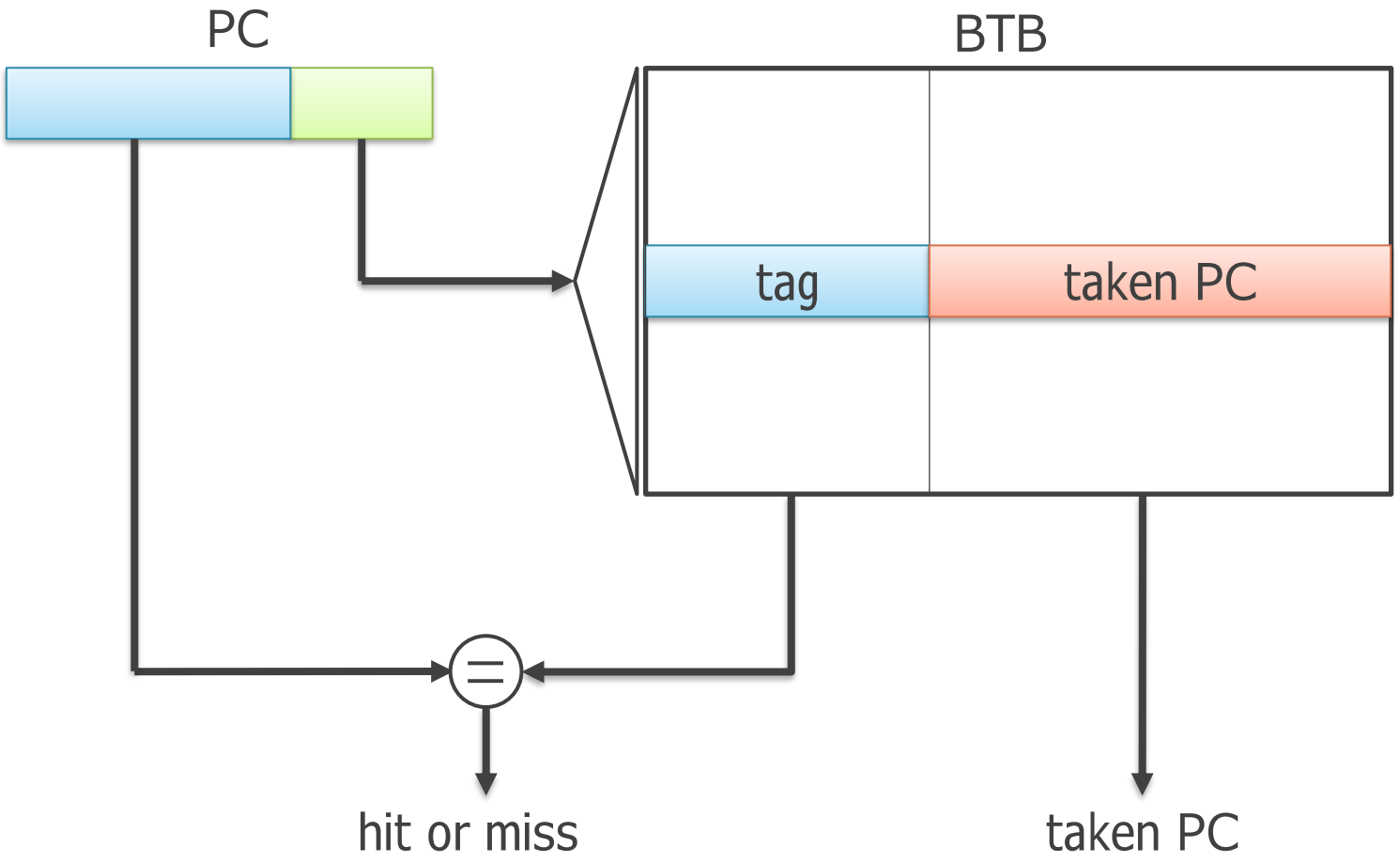
BTB（Branch Target Buffer）による予測

- 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

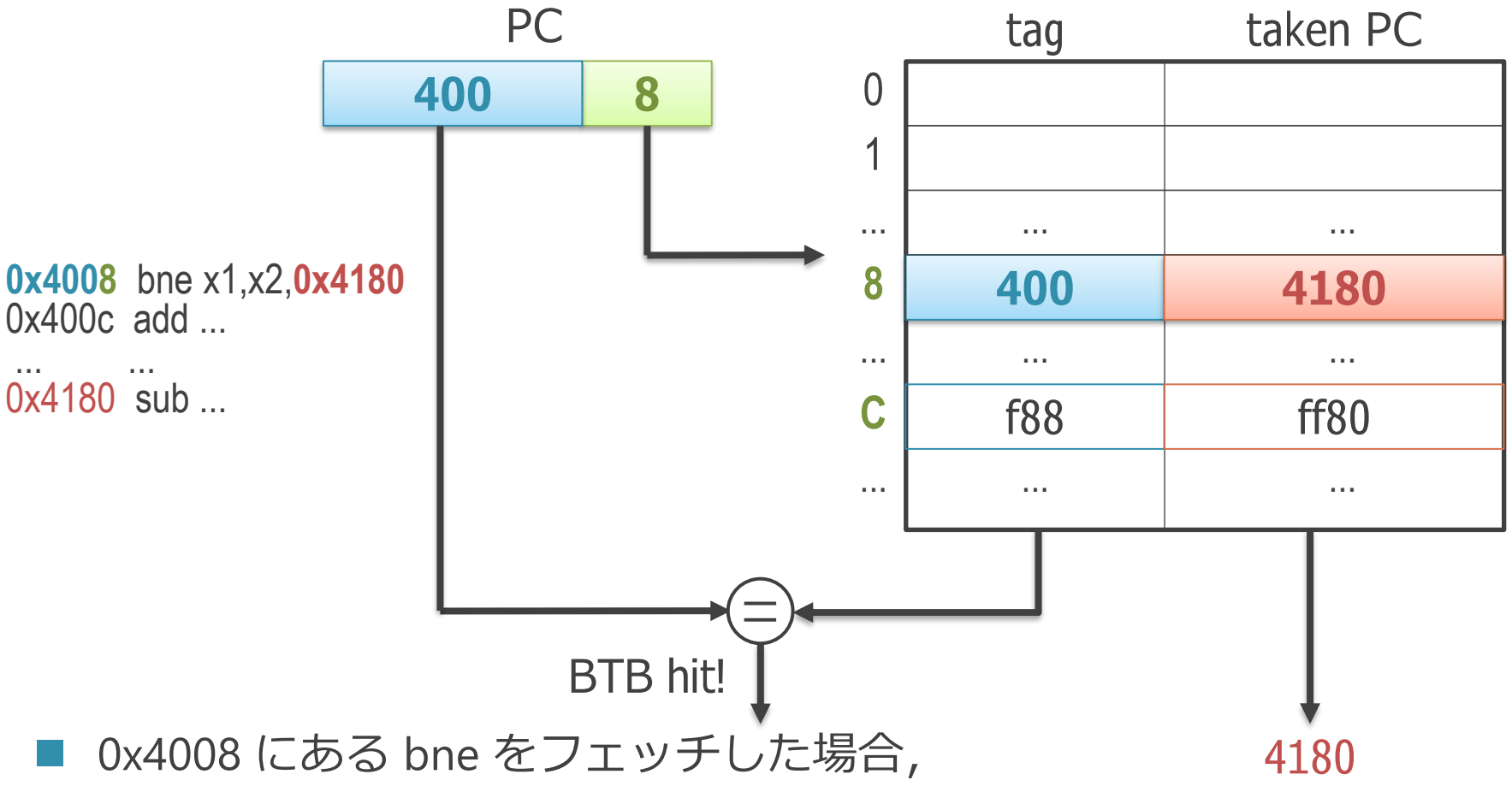
- 分岐命令の実行時に，この表にターゲットを登録しておく
  - ◇ 次回からは，表をひくとターゲットがとれる

BTB（Branch Target Buffer）による予測



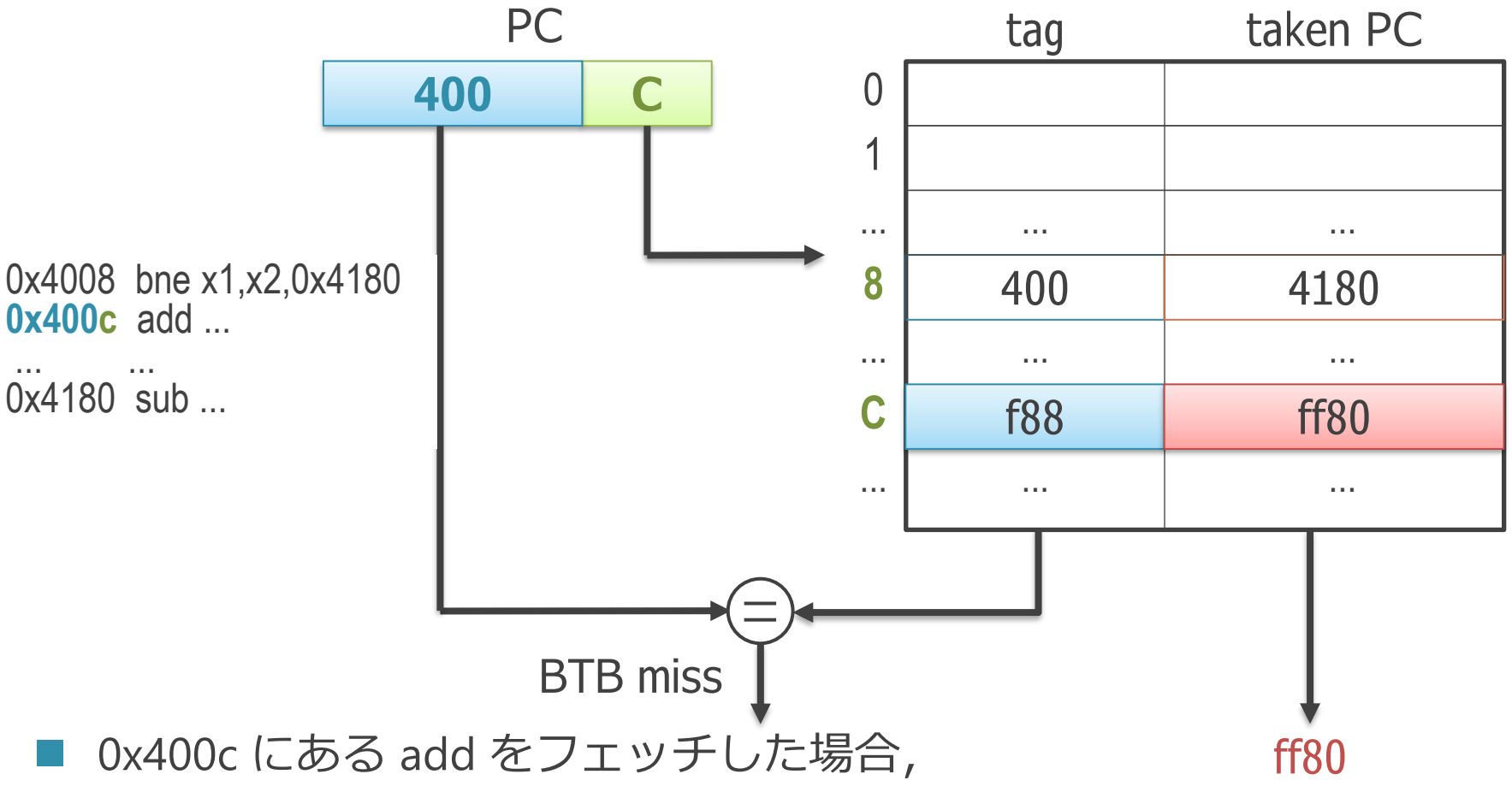
- 分岐かどうかと，分岐先ターゲットを同時に予測

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

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

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

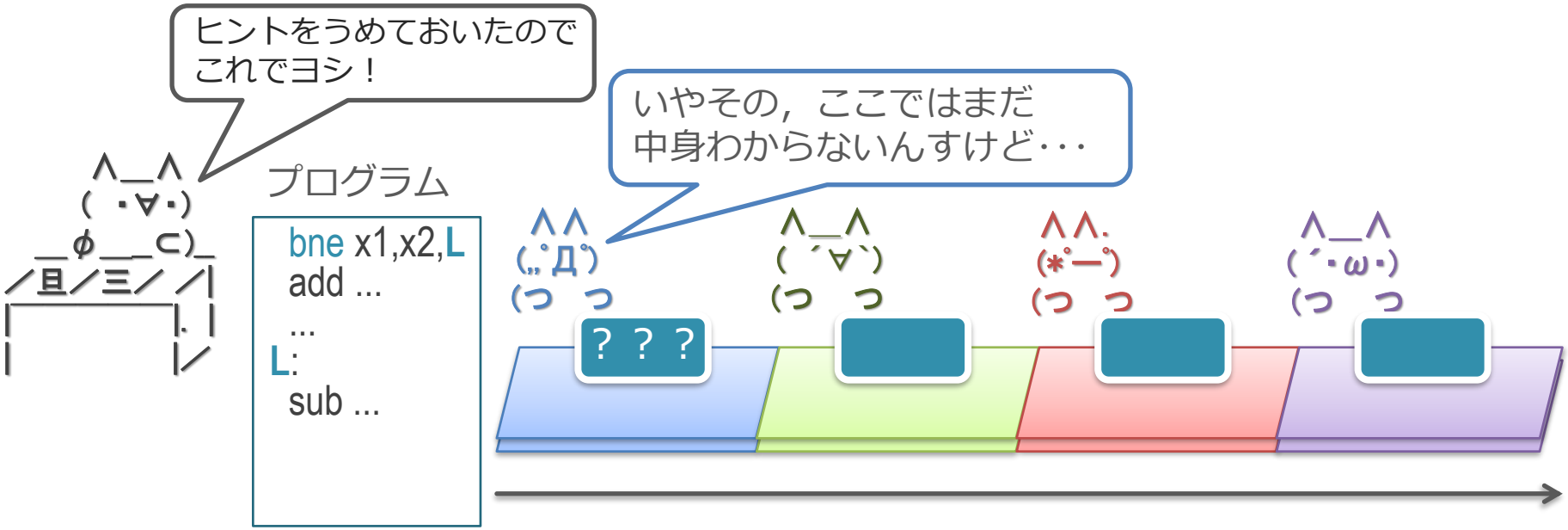
3. プロファイルによる予測

- 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
  - ◇ 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（なんか一言書いてね）
  - 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：branch
- 意見や内容へのリクエストもあったら書いてください