

# 先進計算機構成論 06

---

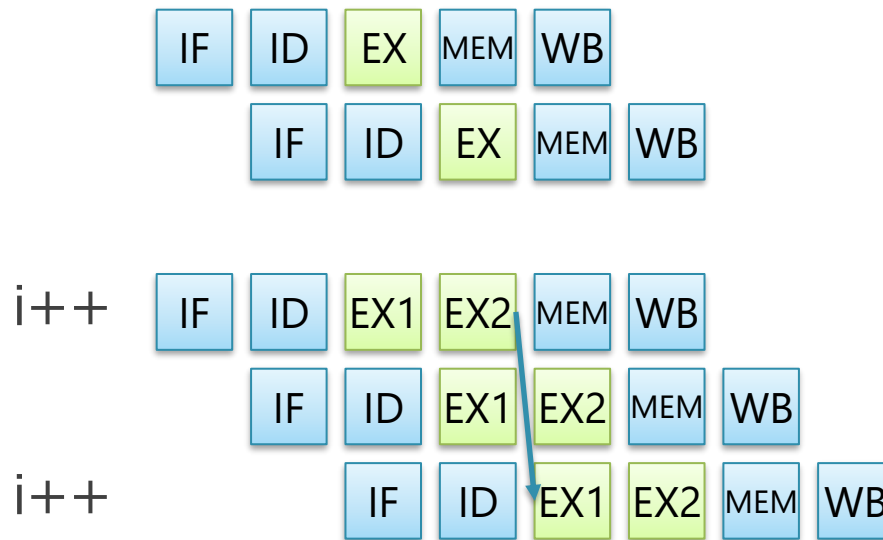
東京大学大学院 情報理工学系研究科 創造情報学専攻

塩谷 亮太

shioya@ci.i.u-tokyo.ac.jp

- ALUの演算は、パイプラインで途中で分断してしまうと性能が落ちるという話があったと思うのですが、その理由がよくわからなかったです。

# 演算器のパイプライン化による性能低下



- 下側は ALU での演算（EX ステージ）を 2 ステージにパイプライン化
  - ◇ 1 命令目の結果は，EX2 が終わるまで使えない
  - ◇ 2 命令目には，1 命令目に依存した命令がおけない  
= 「i++」 みたいな処理は，2 サイクルに 1 回しかできない

# 質問と感想

- アドレスからデータを1クロックで取り出す仕組みが以前の勉強であまりわからなかったので興味がある.
- ◇ 次回以降にメモリの話があります
- また, メモリアドレスは64ビット長で指定できるが, ハードディスクのアドレス指定方法はどうしているのでしょうか.

- マイクロ命令への分解というのは、デコードを複数回処理することで構造ハザードを防ぐという認識でよろしいでしょうか。
- ◇ 構造ハザードを起こさない命令列に変換している

- ハードウェアに記述されいているような周波数はどれだけの精度のものなのでしょうか。

- パイプラインが製品によって様々な配分がされているということがわかりましたが, 結局こういったものが性能のよいものなのかがわかりませんでした. 用途ごとに求められる性能は違うから一概には言えないということでしょうか.
- ◇ トレードオフがあるので, 目標とする性能やハードウェアの大きさにあわせて決める
  - パイプラインが深いほどクロックは上げられるが, 段数が深くなるほど分岐予測などのペナルティが増える
  - パイプラインの本数が多いほど性能が上がるが, ハードウェアが大きくなる

# 質問と感想

- 分岐予測がどれくらい当たるものなのか気になりました。
- 分岐予測は過去の実行結果を参照し同一の予測をするとのことだったと思うのですが、これほど単純な仕組みで高精度の予測が可能なのか気になりました。"
- I have a question related to branch prediction, but maybe it has been mentioned during the class due to my poor Japanese. How good can branch prediction expected to be?



# The prediction accuracy of state-of-the-art predictors

Cited from *Pierre Michaud, "An alternative TAGE-like conditional branch predictor", TACO 2018*

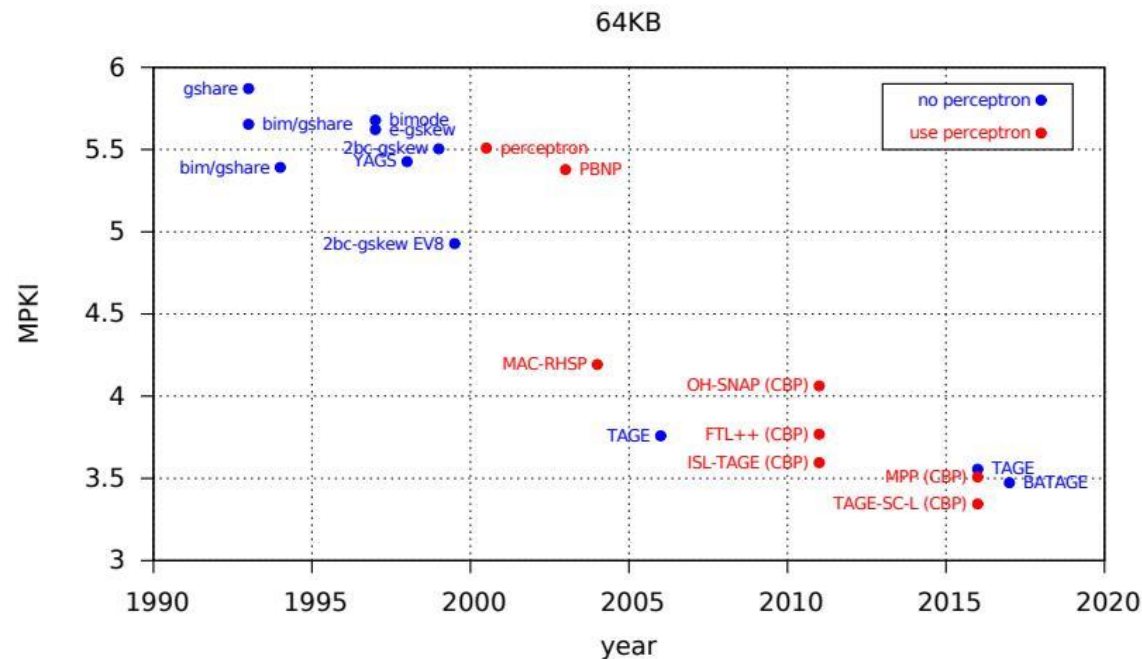


Figure 1: Average number of mispredictions per 1000 instructions (MPKI) for various conditional branch predictors on the CBP 2016 traces for 8KB, 32KB and 64KB storage budgets (see Appendix A).

- ◇ 1000 instructions usually include around 200~250 branch instructions.
- ◇  $MPKI=3.5$  means that the prediction accuracy is  $1 - 3.5/200 \doteq 98\%$ .

- 分岐予測はペナルティが非常に致命的になってしまうという欠点があると仰っていましたが、実際には平均するとどれくらいの時間ロスがあり、それを込みで導入する利点があるのかを定量的に知りたい。
- ◇ ペナルティは時間的ロスはおよそパイプラインの長さのサイクル数になる
- ◇ 導入しない場合は単にその時間を待つだけなので、なんらかの予測を入れた方が一方的にお得
  - 方向の予測なら、完全にでたらめな予測でも50%は当たる

- ごめんなさい！結構おいて行かれています！  
そもそもはパイプライン化を進めるにあたって問題があるから今やっていることを考えているという感じですか？  
もうちょっと全体的な構造に対する現在やっていることの位置がわかったらうれしいです！
- ◇ パイプライン化すると各種のハザードと呼ばれる問題が起きる
  - 構造ハザード
  - 非構造ハザード
- ◇ 前回前々回は、上記ハザードのそれぞれへの対策を述べていた
- ◇ 非構造ハザードのうち制御ハザードの対策方法が分岐予測

- mipsの遅延スロットは分岐命令の後に置かれるものだと思っていましたが、(ロード)データハザードのときにも使われていたことを初めて知りました
- 遅延分岐とはどのようなものでしょうか

- 基本的には短いアセンブリの方が速いと思いますが、時々gccなどのコンパイラが最適化であえて余計な命令の入った長いアセンブリを出力することがあり、それが余計な命令を加えない時よりも速かったことがあります。これはパイプラインを考慮した結果なのかと思いました。
- ◇ 分岐の飛び先が、キャッシュのブロック境界に合っていた方が効率が良いとかはある

- パイプライン化の限界のところで消費電力と熱の話がありましたが、放熱能力の改善というのはどのくらいあるものなののでしょうか？(10年単位で何倍くらいの変化なののでしょうか)

- IntelやAMDのCPUではhyperthreading機能（1 core = 2 threads）が一般的ですがこれはハードウェアのthreadということですか？一つのcoreに三つ以上のthreadを組めるのはあまりみられないのはなぜですか？
  - ◇ 1スレッドを2スレッドにするのは10%以内ぐらいの回路量オーバーヘッドで作れるが、4スレッドにするとオーバーヘッドが無視できなくなってくる
  - ◇ キャッシュを共有することになるので、4スレッドも走らせるとミス率が高くなりすぎる

- 分岐予測でBTBを引くという仕組みは、一度読んだ命令を再度実施しないと意味がない気がするが、ifではなくwhileやforなどのループに特化した機能なのだろうか？
- ◇ ループなどの再実行を前提にしているのはその通り
- ◇ if も含むプログラムのほぼ全てはループの中にある



- 学部の方岐予測の授業では、静的・動的方岐予測しか扱わなかった  
ので、そもそも方岐命令かどうかを判別する機構が必要だというのは  
初耳でした。
- また、方岐予測のためのBTBの構造は、ダイレクトマップキャッ  
シュに似ているように感じました。

# 質問と感想

- 分岐予測の概要については良く理解できたのですが、最後のBTBのところで、なぜBTBで予測出来るのか(特にtagがどうやって作られるのか)が授業中に理解出来なかったため、説明していただけると幸いです。

◇ // index が命令アドレスの下8ビットの場合の BTB (256要素)

```
struct {tag, target} btb[256];  
index = PC & 0xff;    // 命令アドレスの下8ビットを取り除く  
tag    = PC >> 8;     // 命令アドレスの残りを取り出す  
  
// btb のエントリには, index が同じものしか来ない  
// → index と tag が同じなら, それは格納時の PC が等しい  
if (btb[index].tag == tag)  
    target = btb[index].target
```

- I wonder why the branch prediction related information is not computed during compile time.
- ◇ Today I will introduce "static" branch prediction.

- また、関係ないですが先生のアスキーアートが非常に愉快で面白い  
と思いながらいつも見てます。





# 前回のおさらい

1. 命令パイプラインと性能
2. 分岐予測（前編）
  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（前回はこちらまで）
  3. 分岐方向予測
    1. 静的予測
    2. 動的予測

# 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）

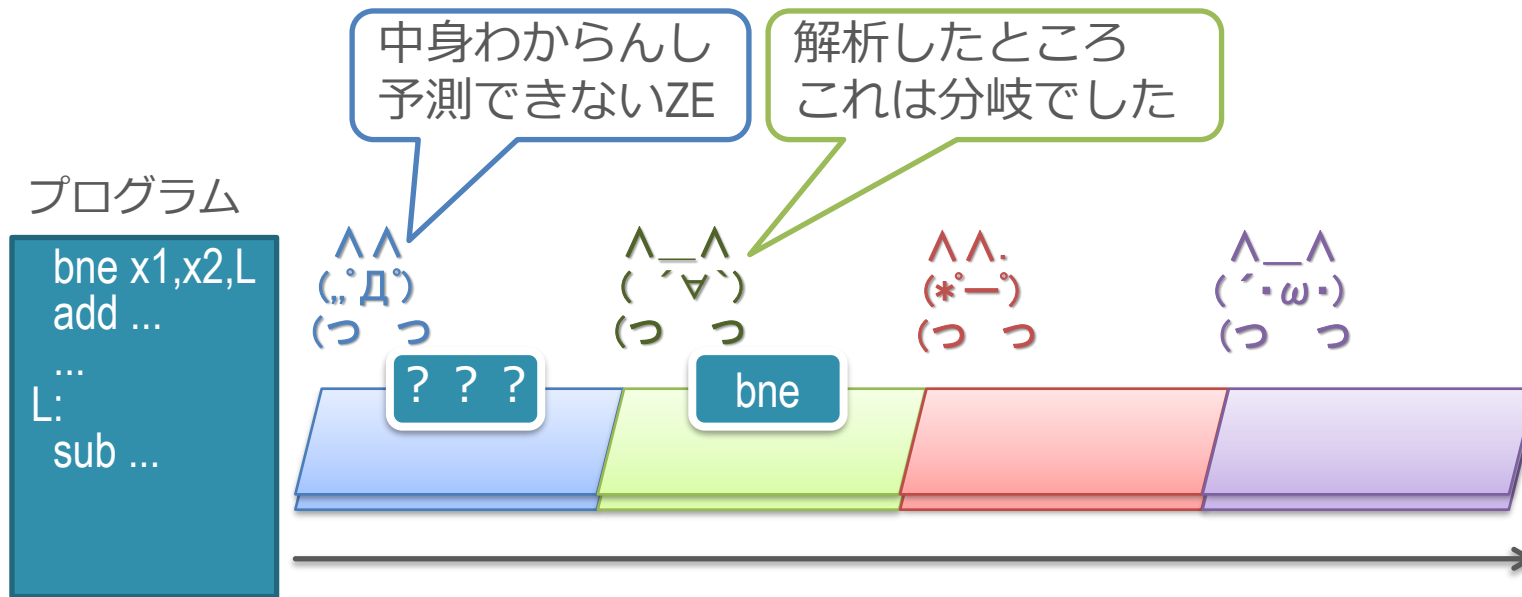
- 分岐先 アドレス or ターゲット
  - ◇ 分岐が成立した際の飛び先のアドレスのこと
- 前方分岐：
  - ◇ 分岐先ターゲットが分岐自身のアドレスよりも大きい分岐のこと
  - ◇ プログラムの進行方向に対して前方に飛ぶことから
- 後方分岐：
  - ◇ 分岐先ターゲットが分岐自身のアドレスよりも小さい分岐のこと
  - ◇ 後方に飛ぶ = ループを作る



# 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、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待ちは、バブルができる

## 2. 分岐先ターゲットの予測の必要性

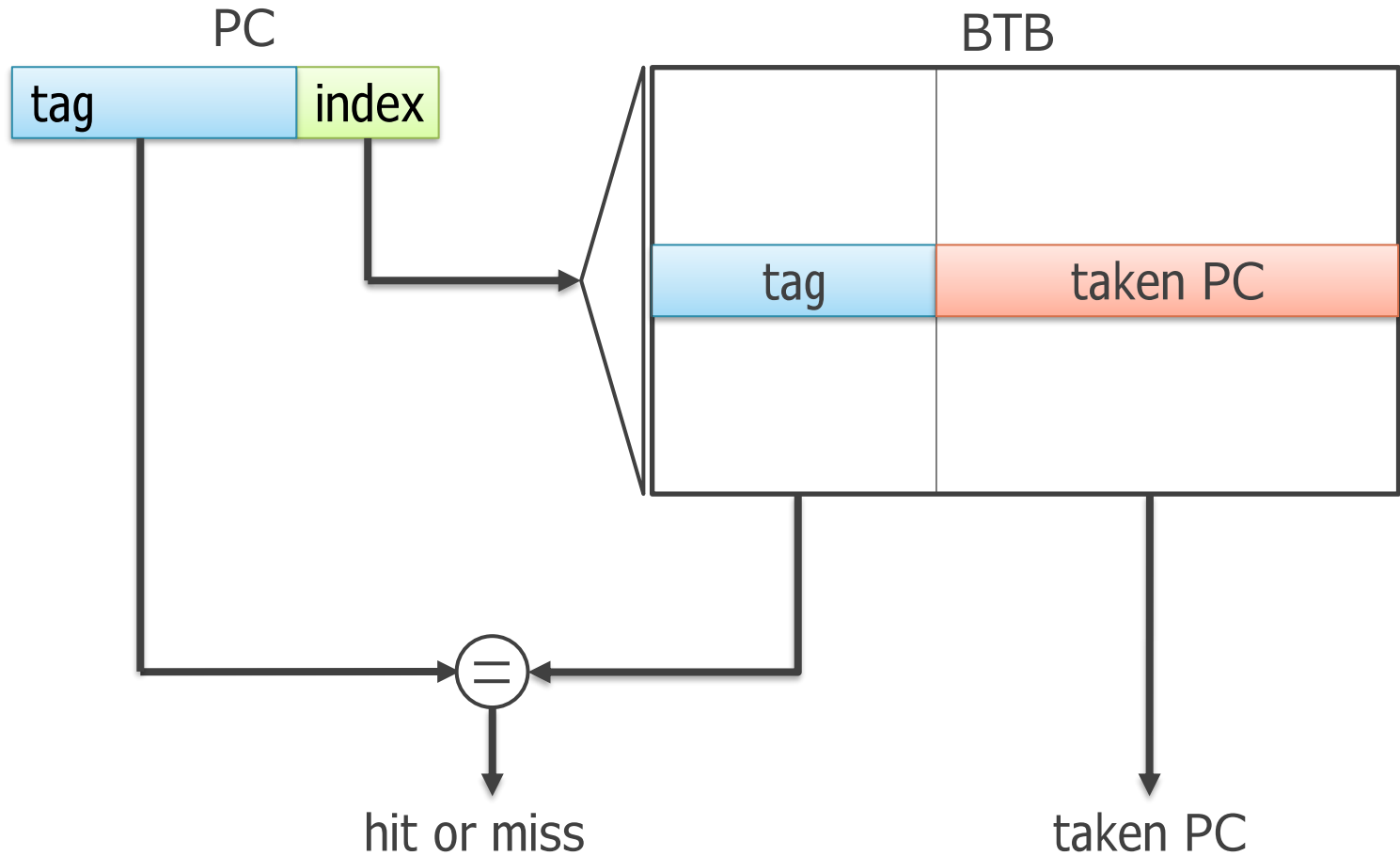


- メモリから命令が取れるまでは，分岐成立時の飛び先の場所もわからない
  - ◇ いくつ先 or いくつ前に飛ぶのか？

# 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

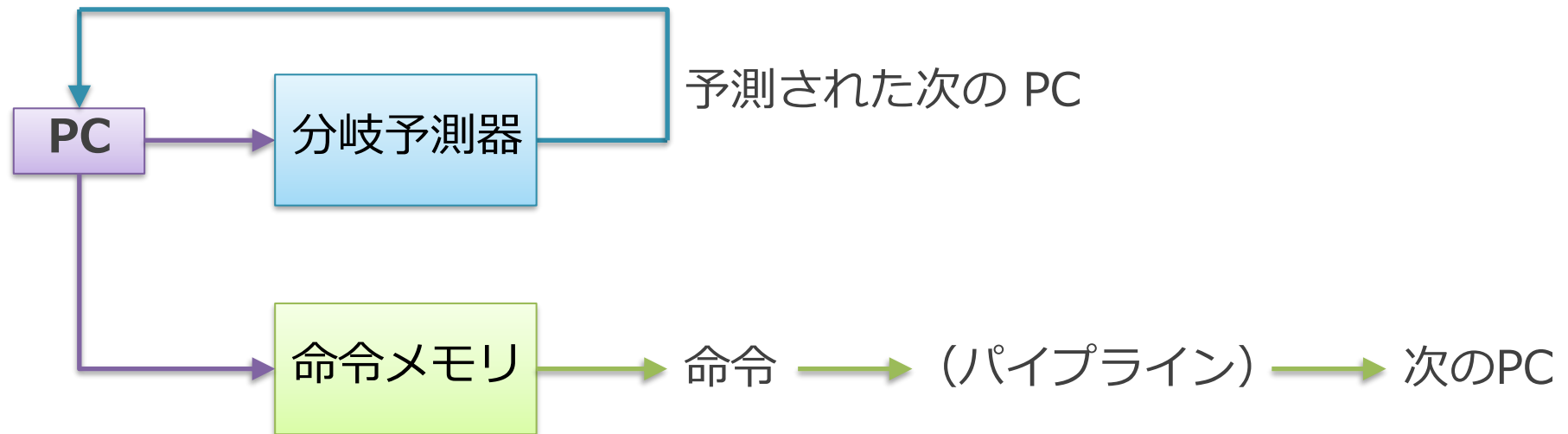
# 分岐予測の補足

1. 分岐予測器の全体構造
2. パイプラインとしての動作

# 分岐予測器の全体構造

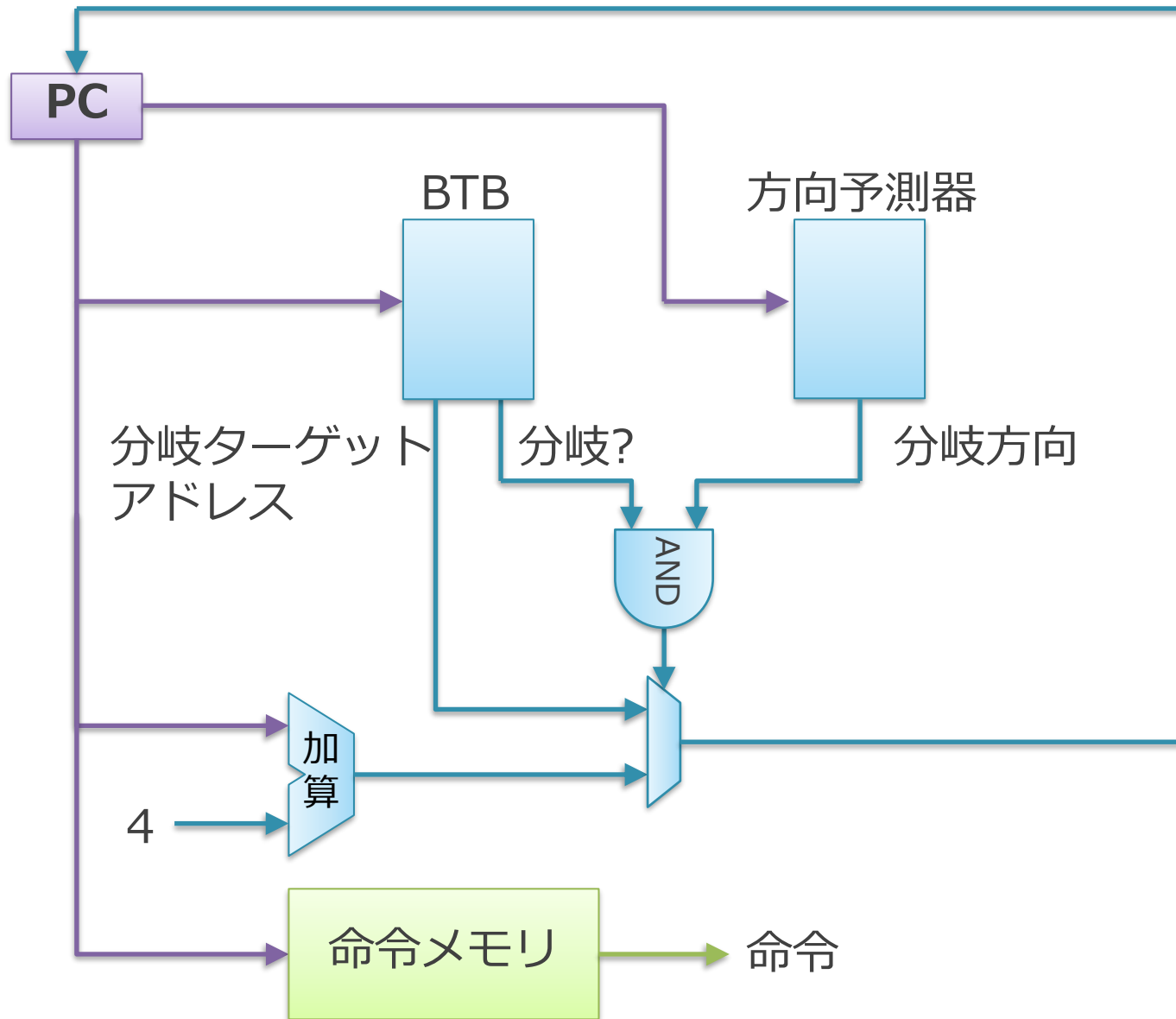
## ■ 分岐予測器全体の仕事：

- ◇ 現在の PC から次の PC を予測する
- ◇ 読み出された命令が実行されて次の PC が確定するのを待たずに次の PC を予測で決定する



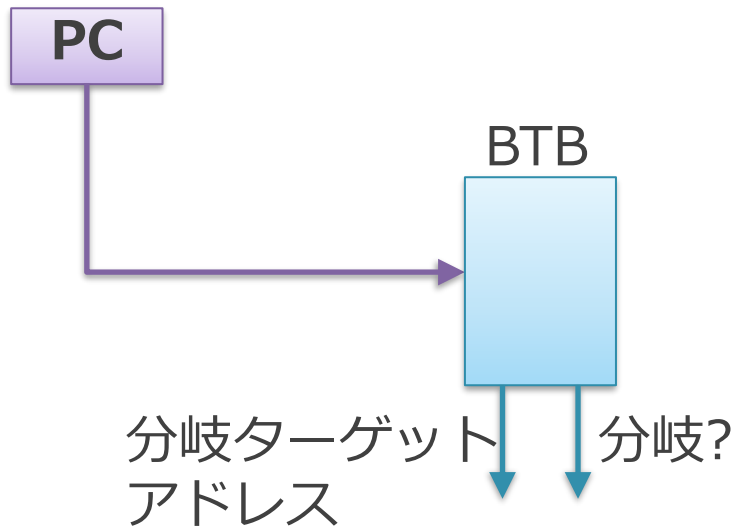


# 分岐予測器の全体構造



# 分岐予測器の動作（１）

## BTB による分岐ターゲットと分岐かどうかの予測

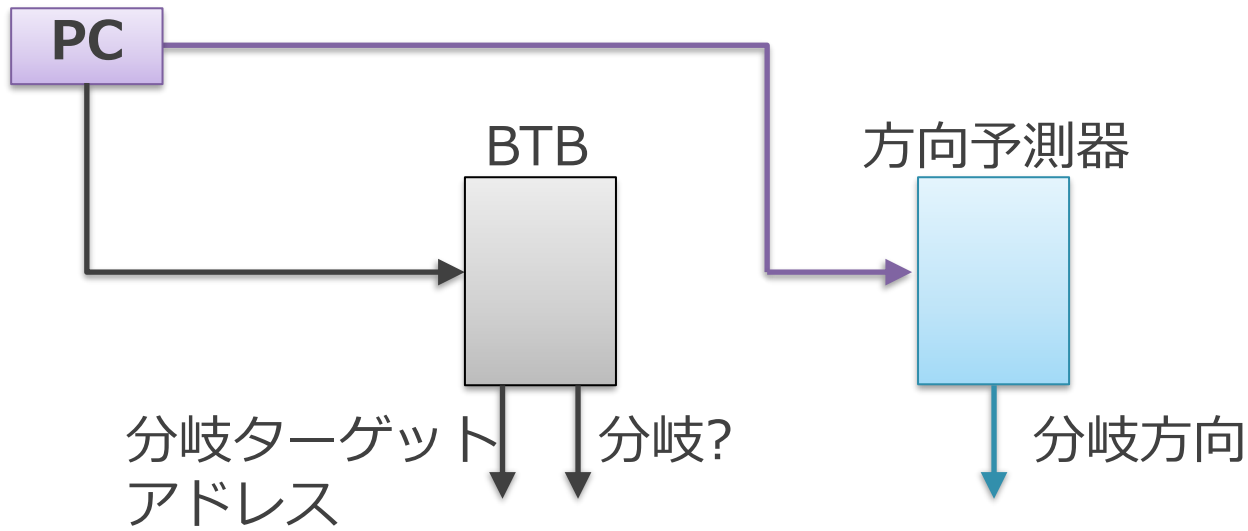


### ■ BTB により以下を予測

- ◇ 分岐の飛び先（ターゲット・アドレス）
- ◇ 分岐かどうか

# 分岐予測器の動作（２）

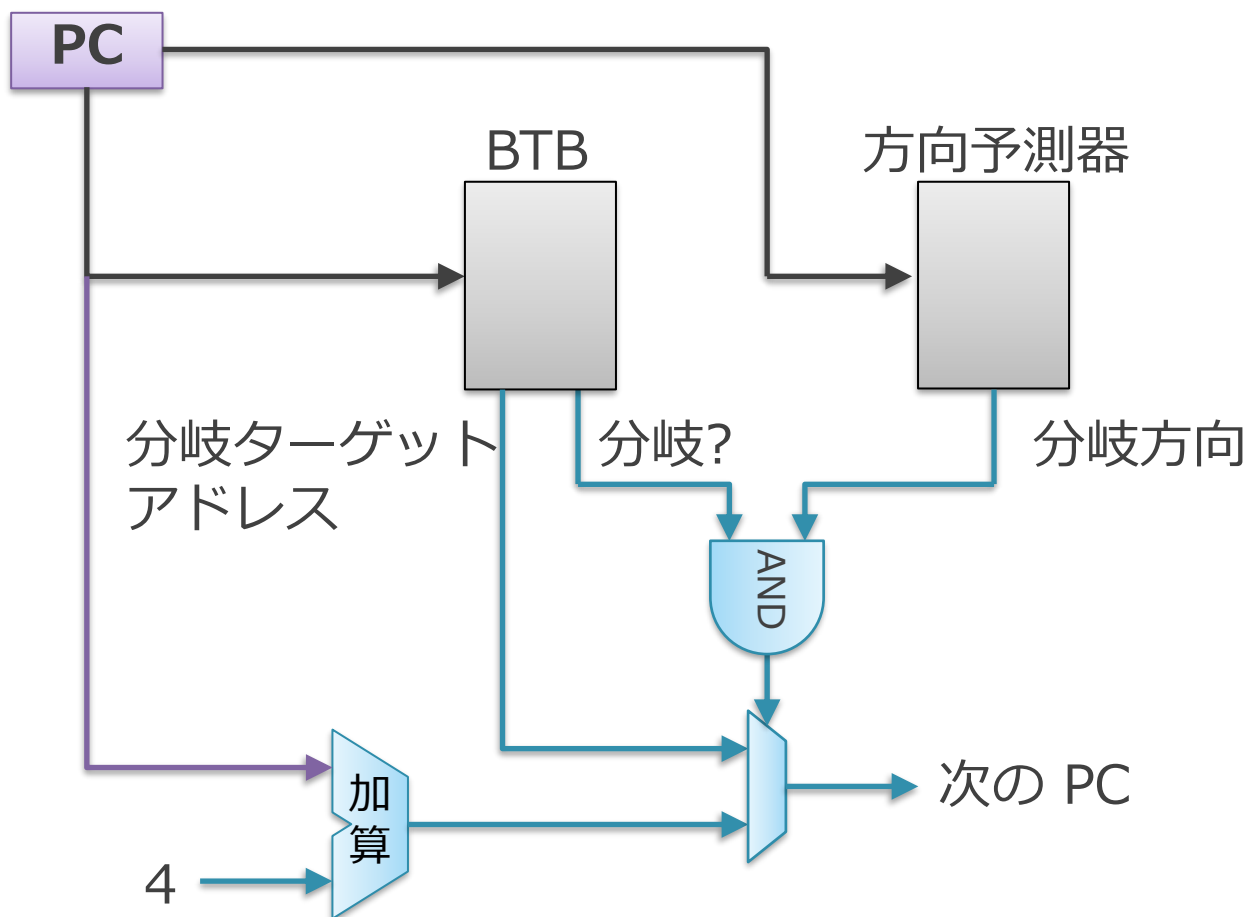
## 方向予測器による分岐方向の予測



- 方向予測器により，分岐の方向を予測

# 分岐予測器の動作 (3)

## 次の PC の予測



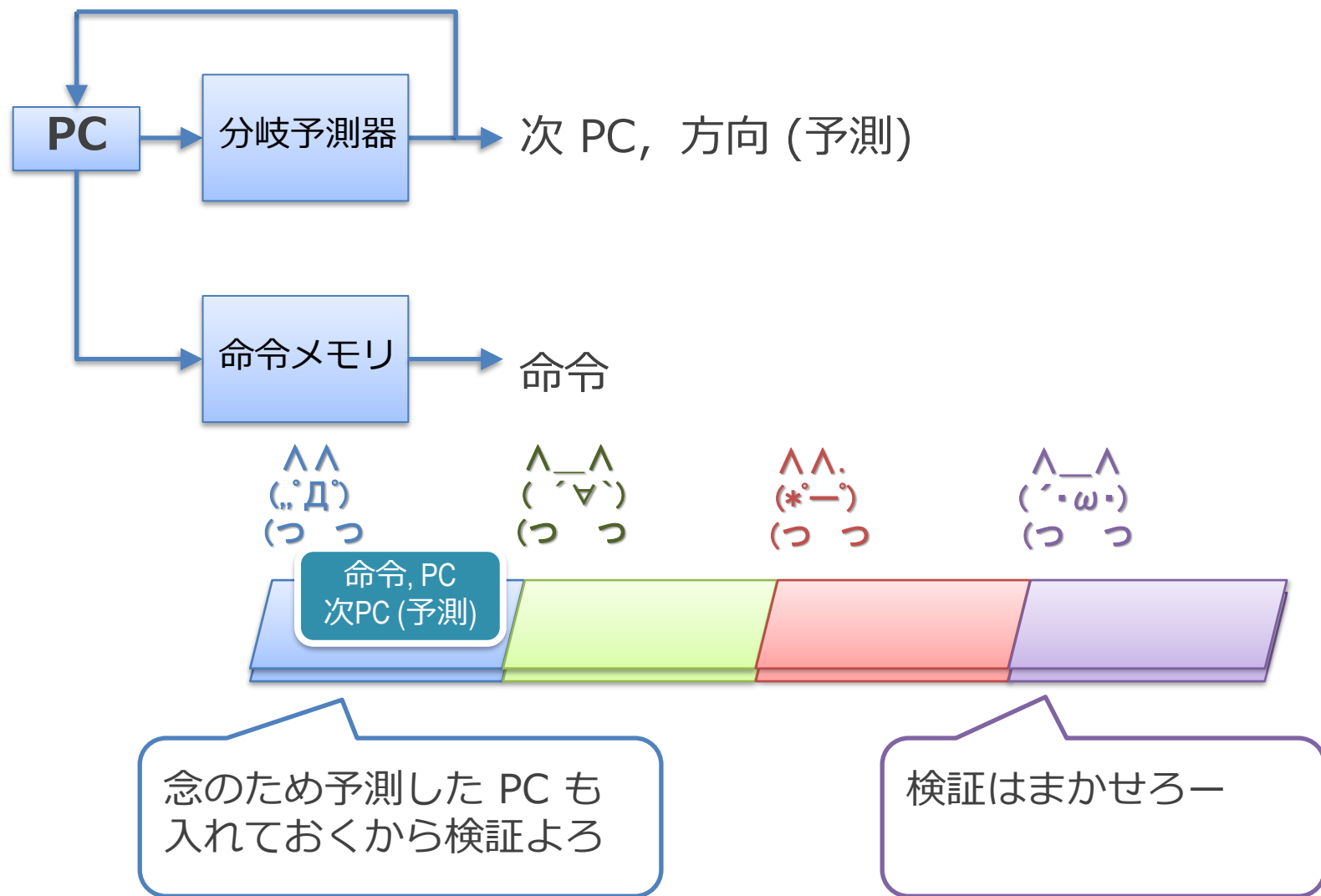
- 次の PC をマルチプレクサにより選択
  - ◇ 分岐命令かつ分岐が成立なら, ターゲット・アドレスを選択
  - ◇ そうでなければ  $PC + 4$  を選択

# 分岐予測の続き

1. 分岐予測器の全体構造
2. パイプラインとしての動作
3. 間接分岐予測

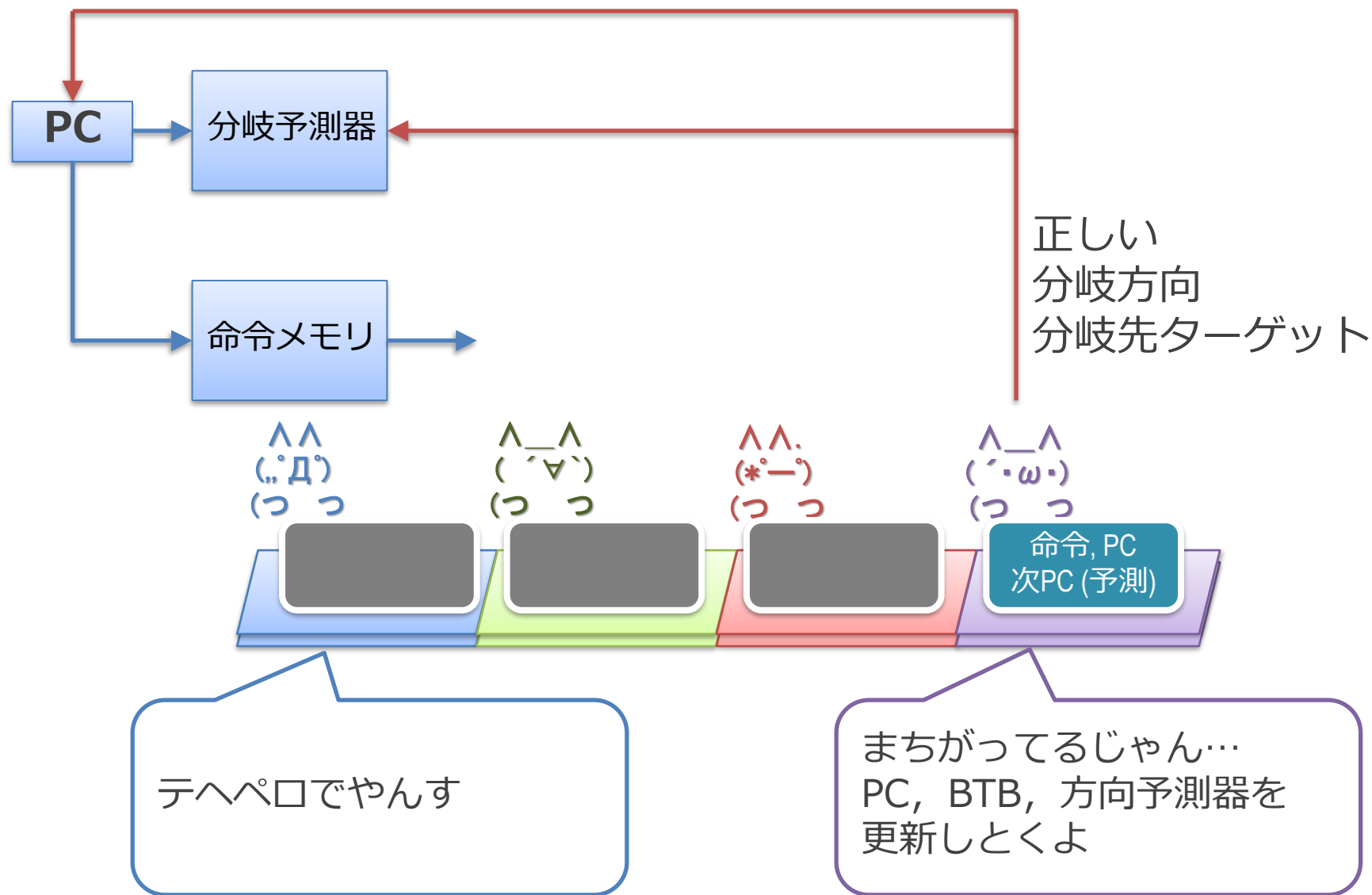
# パイプラインとしての動作 (1)

## 予測結果の PC や方向をパイプラインに流す



# パイプラインとしての動作 (2)

## 予測ミス判明時に予測器や PC を学習



# 今日の内容

1. 分岐予測（後編）
  1. 静的分岐予測
  2. 動的分岐予測



# 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

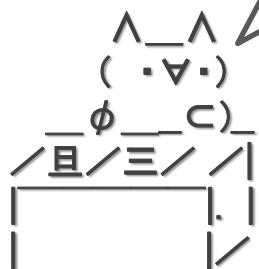
# 「後方分岐かどうか」 「成立/不成立のヒント」の予測

ヒントをうめておいたので  
これでヨシ！

プログラム

```
bne x1,x2,L  
add ...  
...  
L:  
sub ...
```

いやその、ここではまだ  
中身わからないんですけど...



$\Lambda \Lambda$   
( $\cdot \nabla$ )  
( $\cdot \nabla$ )

???

$\Lambda \Lambda$   
( $\cdot \nabla$ )  
( $\cdot \nabla$ )

$\Lambda \Lambda$   
( $\cdot \nabla$ )  
( $\cdot \nabla$ )

$\Lambda \Lambda$   
( $\cdot \nabla$ )  
( $\cdot \nabla$ )

■ 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

# 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

# 動的分岐予測

## ■ さまざまな予測手法を紹介

### 1. n ビット・カウンタ

1. 1ビット・カウンタ予測器
2. 2ビット・カウンタ予測器

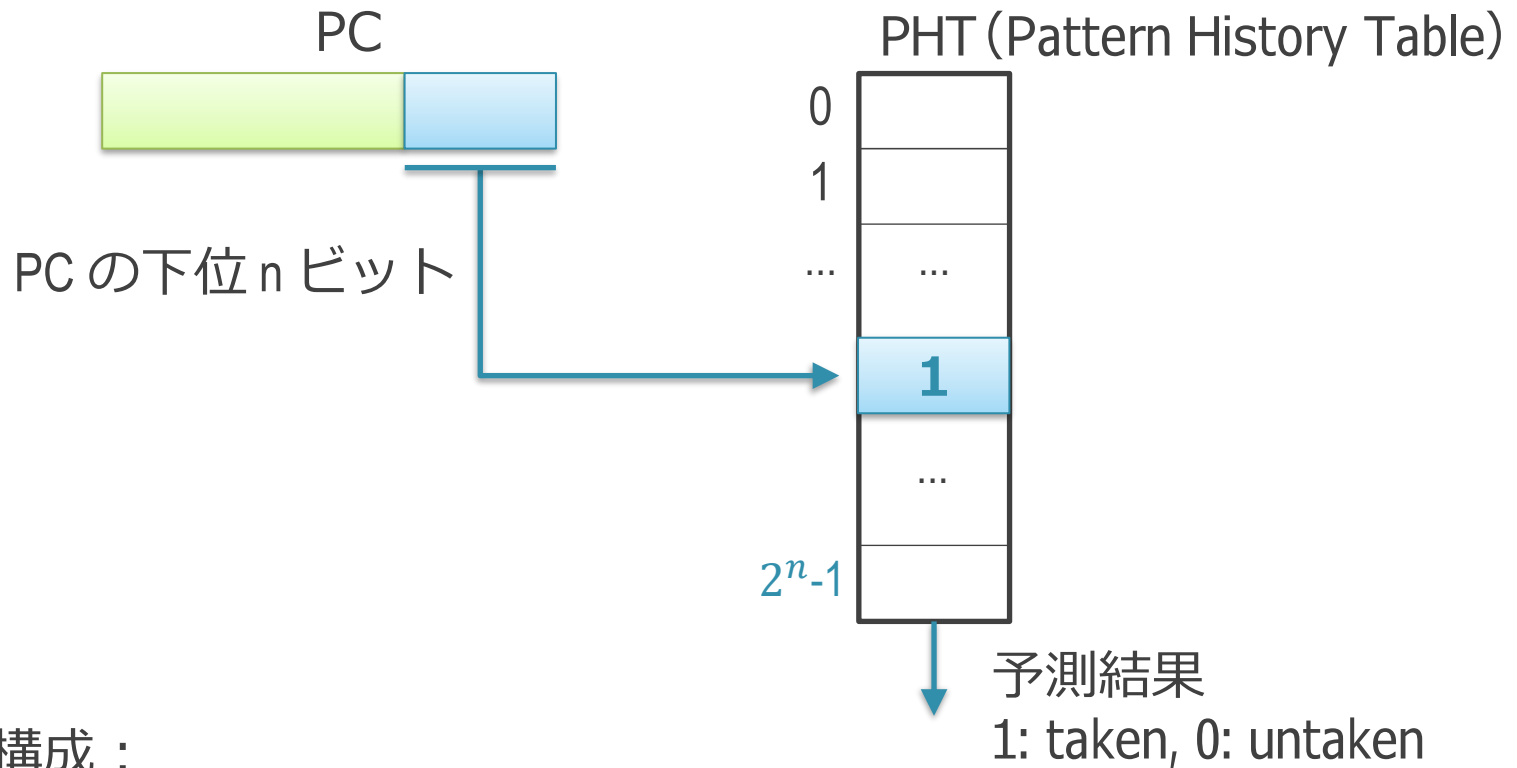
### 2. 履歴を用いたもの

1. ローカル履歴予測器
2. グローバル履歴予測器
3. より高度な予測器

## ■ 下に行くほど先進的

◇ それぞれ上にあるものを下敷きとしているので, 順に説明

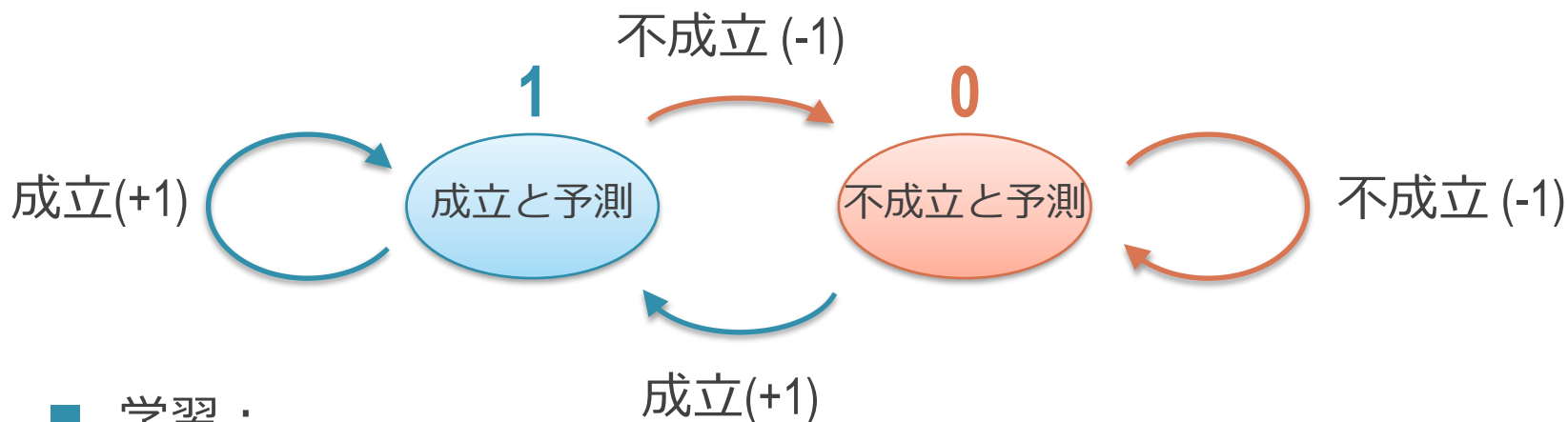
# 1ビット・カウンタ予測器



## ■ 構成 :

- ◇ PC の下位 n ビットをインデクスとしてアクセス
  - エントリ数は  $2^n - 1$  エントリ
- ◇ 各エントリは 1 ビットの飽和型カウンタ

# 1ビットの飽和型カウンタの状態遷移図



## ■ 学習 :

- ◇ 分岐が成立したら +1, 不成立なら -1
- ◇ 1 を超えたら1, 0を下回ったら0

## ■ 予測 :

- ◇ 1 なら成立, 0 なら不成立



# 1ビット・カウンタ予測器の意味

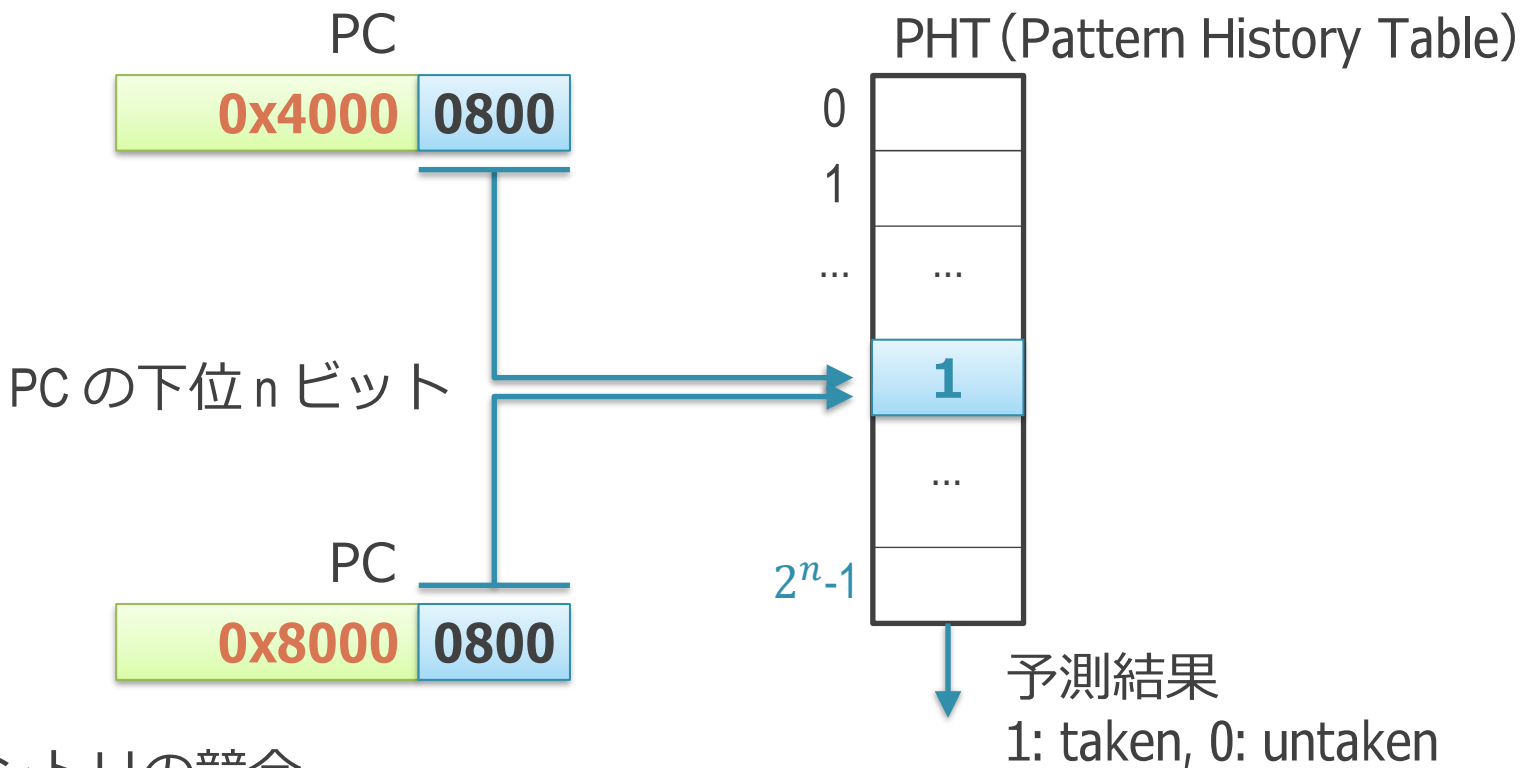
## ■ 動作：

- ◇ 静的分岐のPCごとに、前回の動的分岐の結果を再現
  - 分岐が成立 → カウンタが1に → 次回は成立と予測
  - 分岐が不成立 → カウンタが0に → 次回は不成立と予測

# 1ビット・カウンタ予測器の利点

- 静的な分岐命令の分岐方向に偏りがある場合に有効に働く
  - ◇ 例：ソースコード～行目の if は大体こっちに行く
- 静的分岐予測では対応不能な場合にも対応出来る
  - ◇ 偏りはあっても, コンパイル時には決定できない場合
  - ◇ 例：コマンドライン・オプションによる分岐
    - -hoge の時はこの分岐は常に不成立
    - -fuga の時は常に成立

# エントリの競合



## ■ エントリの競合

- ◇ 下位ビットがかぶると、異なるアドレスの分岐が同じエントリを使ってしまう
- ◇ 偏りが逆方向だと、予測精度を落とす
- ◇ エントリ数を増やす事で解消可能

# 分岐方向予測

1. 静的分岐予測
2. 動的分岐予測
  1. n ビット・カウンタ
    1. 1ビット・カウンタ予測器
    2. 2ビット・カウンタ予測器
  2. 履歴を用いたもの
    1. ローカル履歴予測器
    2. グローバル履歴予測器
    3. より高度な予測器

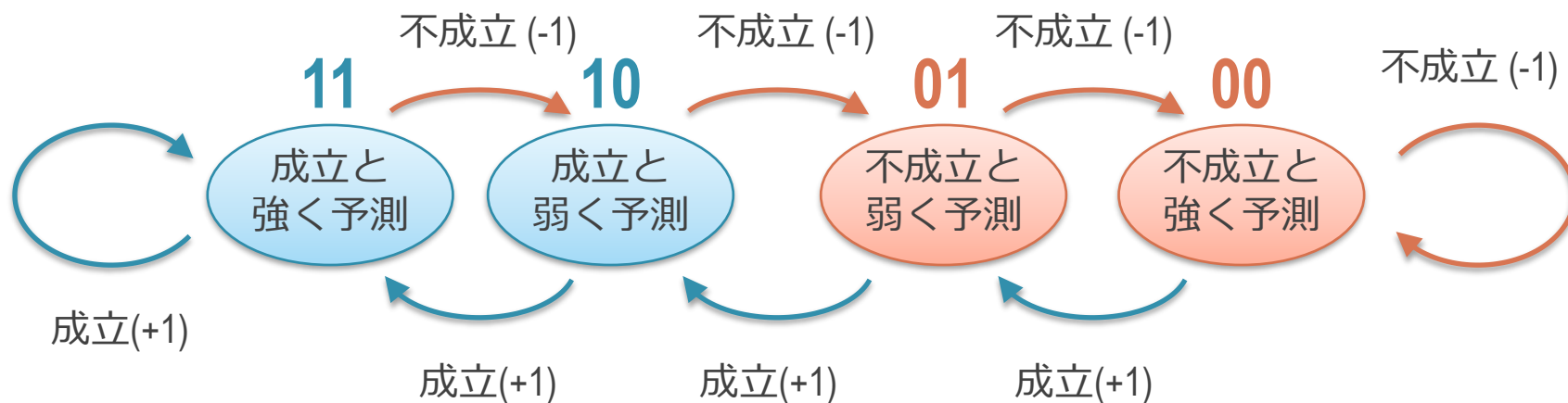
# 1ビット・カウンタ予測器の問題点：無駄な遷移

- 偏りがある場合に，無駄な状態遷移を起こす
  - ◇ たまに偏りと逆の方向に行った時に，戻ってくる時も必ず外す
  - ◇ 静的分岐予測で正しく偏りが拾えている場合，これは起きない
- 例： 成立：T，不成立：F とした場合，
  - ◇ カウンタ : 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1
  - ◇ 予測 : T T T T F T T T T T F T T T T
  - ◇ 実際の方向 : T T T F T T T T T F T T T T T
  - ◇ （不成立は2回だが，予測は4回はずしている

## 2ビット・カウンタ予測器

- 1ビット・カウンタ予測器のカウンタを2ビットにする
  - ◇ 1回反対方向に行っても，次回も元の方角を予測することができる

## 2ビットの飽和型カウンタの状態遷移図



### ■ 学習 :

- ◇ 分岐が成立したら +1, 不成立なら -1
- ◇ 11(2進)を超えたら11, 0を下回ったら0

### ■ 予測 :

- ◇ 1なら成立, 0なら不成立

## 2ビット・カウンタ予測器の動作

- カウンタが 2 以上なら成立と予測，そうでなければ不成立と予測

- 例： 成立：T，不成立：F とした場合，

◇ カウンタ       : 11 11 11 10 11 11 11

（10進表記     : 3   3   3   2   3   3   3

◇ 予測           : T   T   T   T   T   T   T

◇ 実際の方向   : T   T   T   F   T   T   T

◇ 不成立は 1 回であり，予測ミスも 1 回



# 予測精度とカウンタの幅

- 一般に, 1 ビット・カウンタ予測器より性能が高いと言われる
  - ◇ 実際に Intel Pentium, MIPS R10000 などの CPU に搭載
- カウンタのビット数を 3 ビット以上にすることは通常ない
  - ◇ 特に性能が向上しないことが知られている
  - ◇ 場合によっては, 精度が落ちる
    - 分岐の傾向が変わった時に, 学習結果の反映が遅れると言われている

# 分岐方向予測

1. 静的分岐予測
2. 動的分岐予測
  1. n ビット・カウンタ
    1. 1ビット・カウンタ予測器
    2. 2ビット・カウンタ予測器
  2. 履歴を用いたもの
    1. ローカル履歴予測器
    2. グローバル履歴予測器
    3. TAGE 予測器

# n ビット・カウンタ予測器の問題

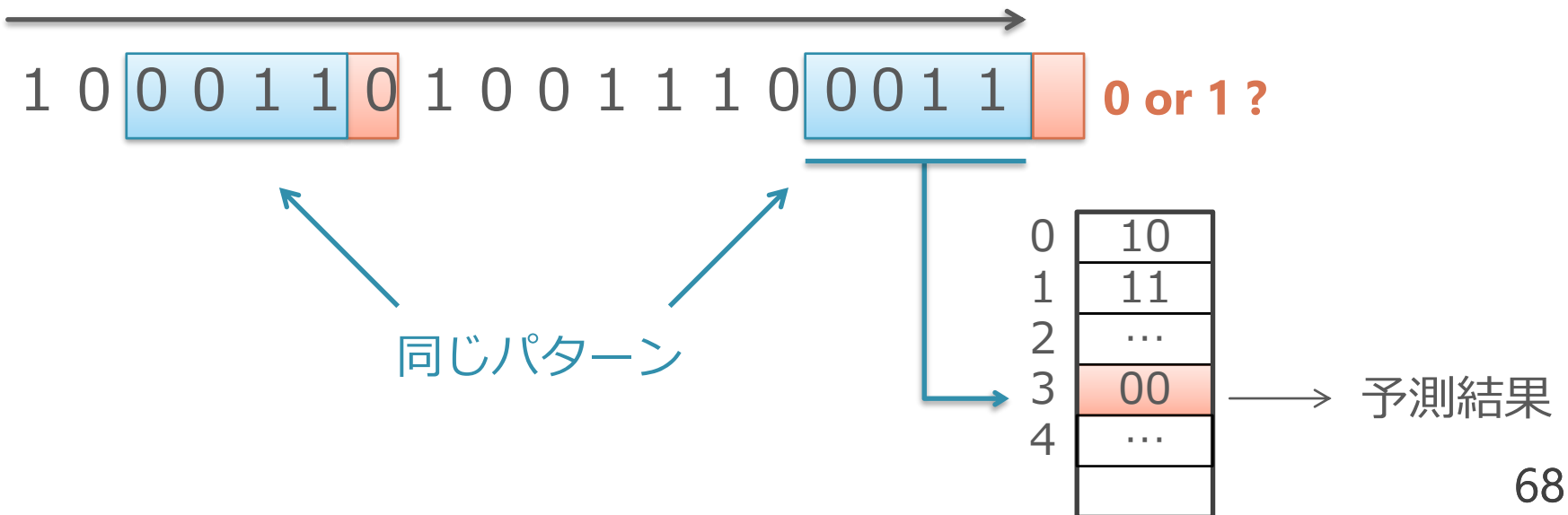
```
for (i = 0; i < 4; i++) {  
    ...  
}
```

- 動的に分岐の方向が頻繁に変わるものに対応できない
- 例：4回まわる for ループ
  - ◇ 4 回のうち 3 回は成立，1 回が不成立となる
  - ◇ TTTFTTTFTTTFTTT...
  - ◇ 2 ビット・カウンタ予測器の精度は  $3/4 = 75\%$  に
- モチベーション：
  - ◇ しかし明らかに規則性があるので，なんとか予測できないか

# 「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

履歴 成立：1 不成立：0



# 履歴とエントリの対応

PHT (2ビット・カウンタ)

直前の履歴	0000	00
	0001	11
	0010	11
	...	...
	1010	00
		...

- 直前の履歴ごとに、異なる PHT のエントリが割り当てられる
  - ◇ 0001 : カウンタが 11 なので、このパターンは次は 1
  - ◇ 1010 : カウンタが 00 なので、このパターンは次は 0

# PHT アクセス時のインデクスの生成は、履歴と PC を混ぜる

- 全てのアドレスの分岐でエントリが共有されてしまう
  - ◇ たとえば直前の履歴が同じ 1010 場合、  
PC が 0x4000 の分岐も 0x8044 の分岐も同じエントリを使ってしまう
    - (0x4000 とかは適当で、意味はない)
- 対策の例：PC と履歴をビット結合する
  - ◇ 0x4000 と 0xa (2進で1010=0xa) を結合 → 0x4000a をインデクスに
  - ◇ 0x8044 と 0xa (2進で1010=0xa) を結合 → 0x8044a をインデクスに

# 履歴を用いた予測器

- この履歴の保持方法/作り方の違いで、いくつかの方法がある

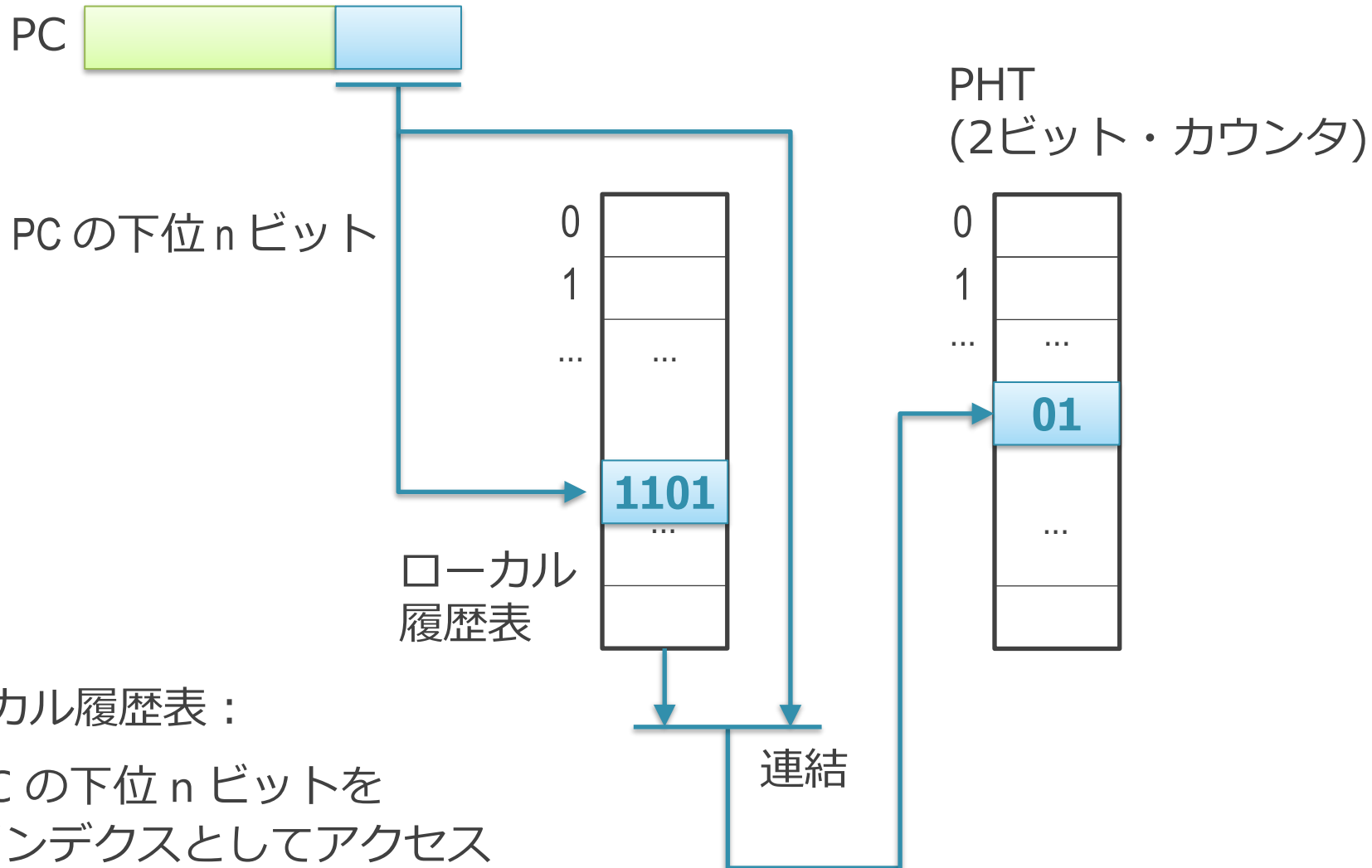
1. ローカル履歴予測器

- PC ごとに履歴を保持

2. グローバル履歴予測器

- PC を区別せず履歴を保持

# ローカル履歴予測器



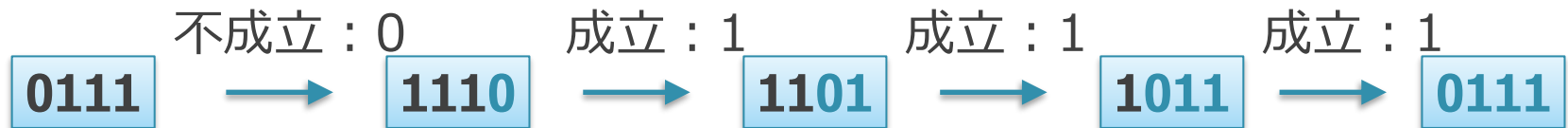
## ■ ローカル履歴表：

- ◇ PC の下位 n ビットをインデクスとしてアクセス
- ◇ 各エントリは複数ビットのシフト・レジスタ



# ローカル履歴表

- その PC の分岐の, 過去に分岐方向のパターンを表す
- 分岐が実行されるごとに,
  - ◇ 全体を左にシフトし
  - ◇ 右側から新しい結果を挿入
- 下の図の場合は 4 回に 1 回, 不成立 0 が挿入されている
  - ◇ 4 回だけ回る for ループのパターン



# ローカル履歴予測器の動作例（１）

```
for (i = 0; i < 4; i++) {  
    ...  
}
```

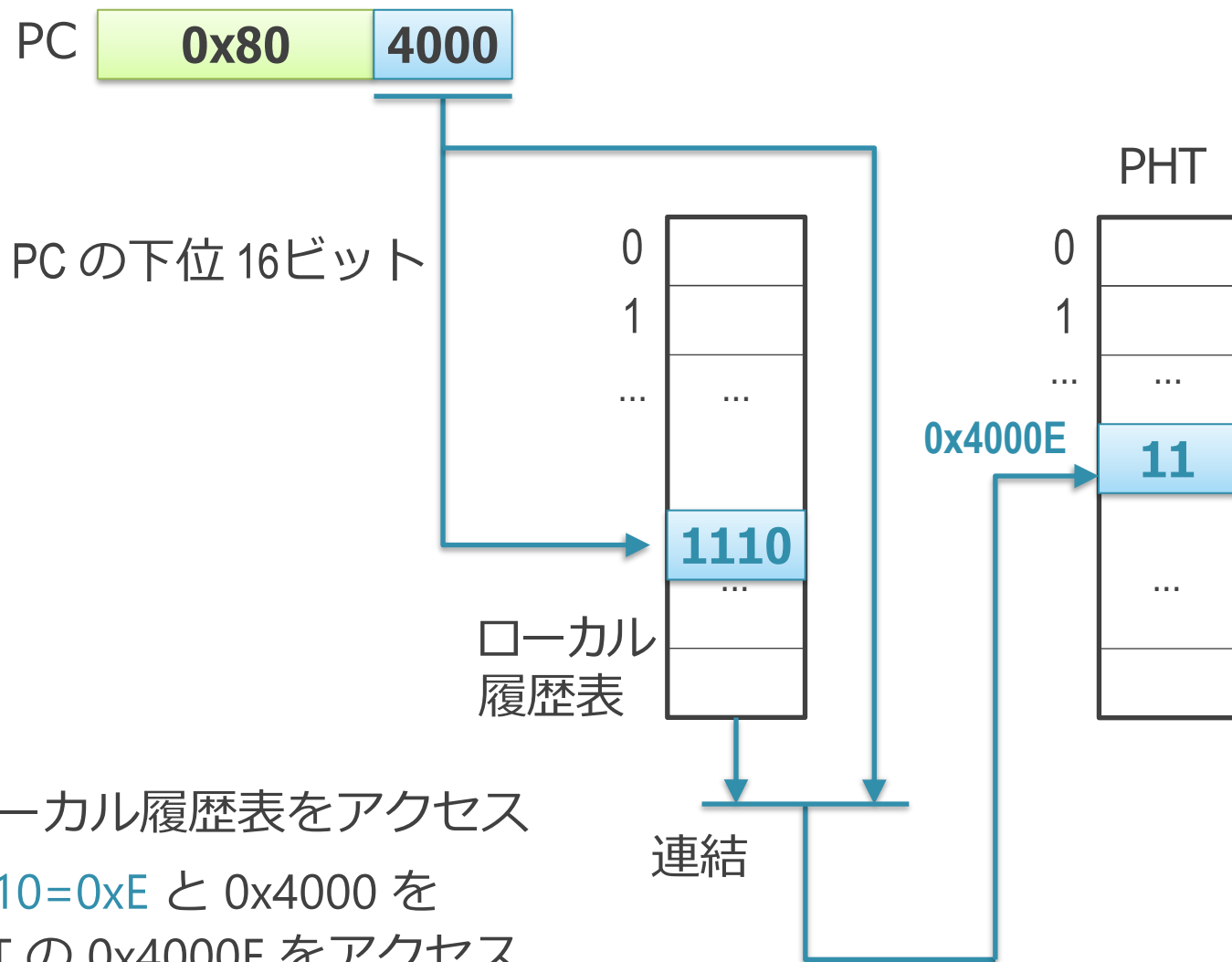
// 4回まわるループ

```
0x803ff8:      li    x1 ← 0           // x1 を 0 に初期化  
      L:  
0x803ffc:      add   x1 ← x1 + 1      // x1 をインクリメント  
0x804000:      bne   x1 != 4, L      // x1 が 4 でなければ L に飛ぶ
```

## ■ 例：4回まわるループ

- ◇ 4回のうち3回は成立，1回が不成立
- ◇ このループの後方分岐が 0x804000 にあったとする

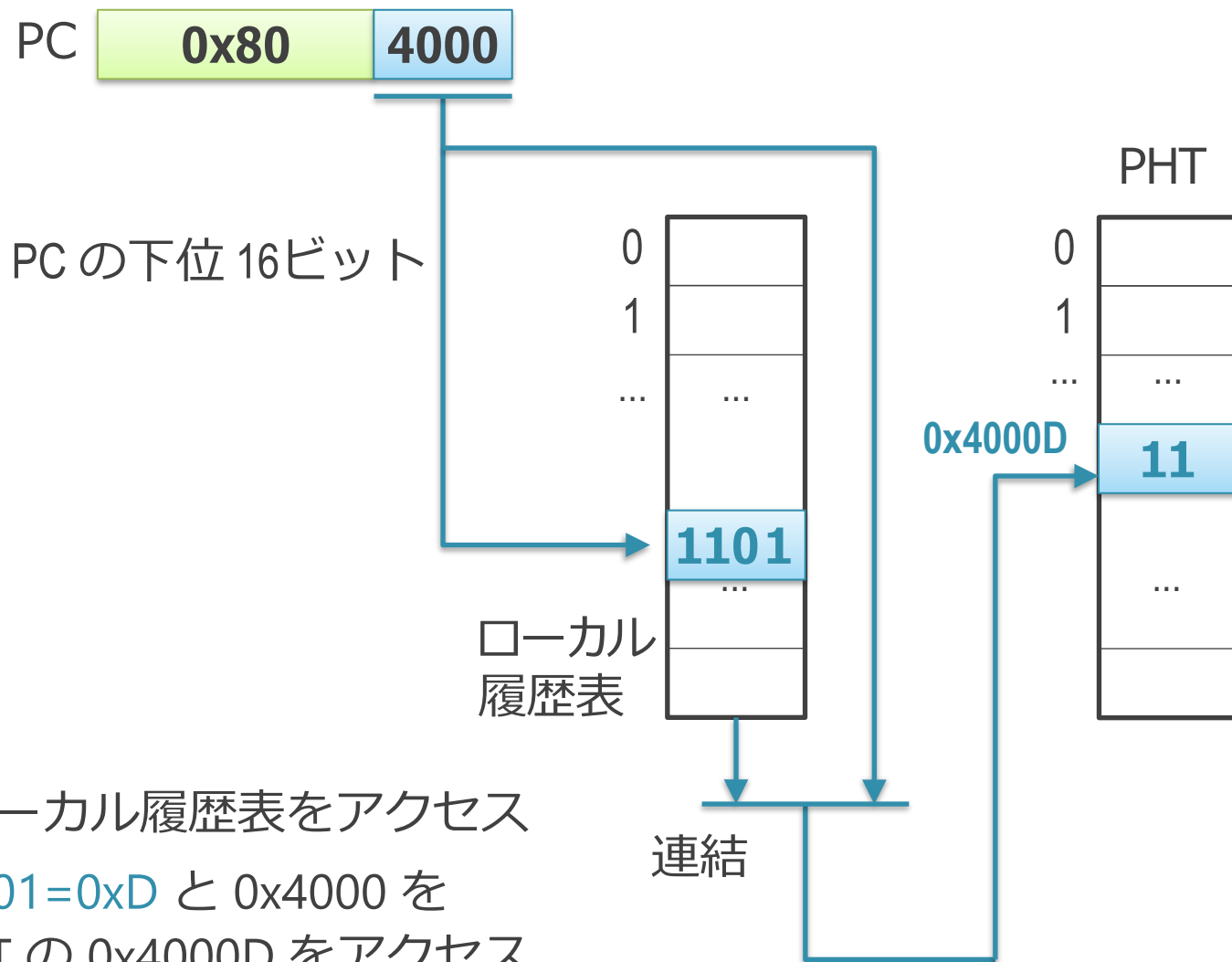
# ローカル履歴予測器の動作例（２）



## ■ 動作：

1. 0x4000 でローカル履歴表をアクセス
2. 得られた **1110=0xE** と 0x4000 を結合し, PHT の 0x4000E をアクセス
3. カウンタの中身が 11 なので成立と予測

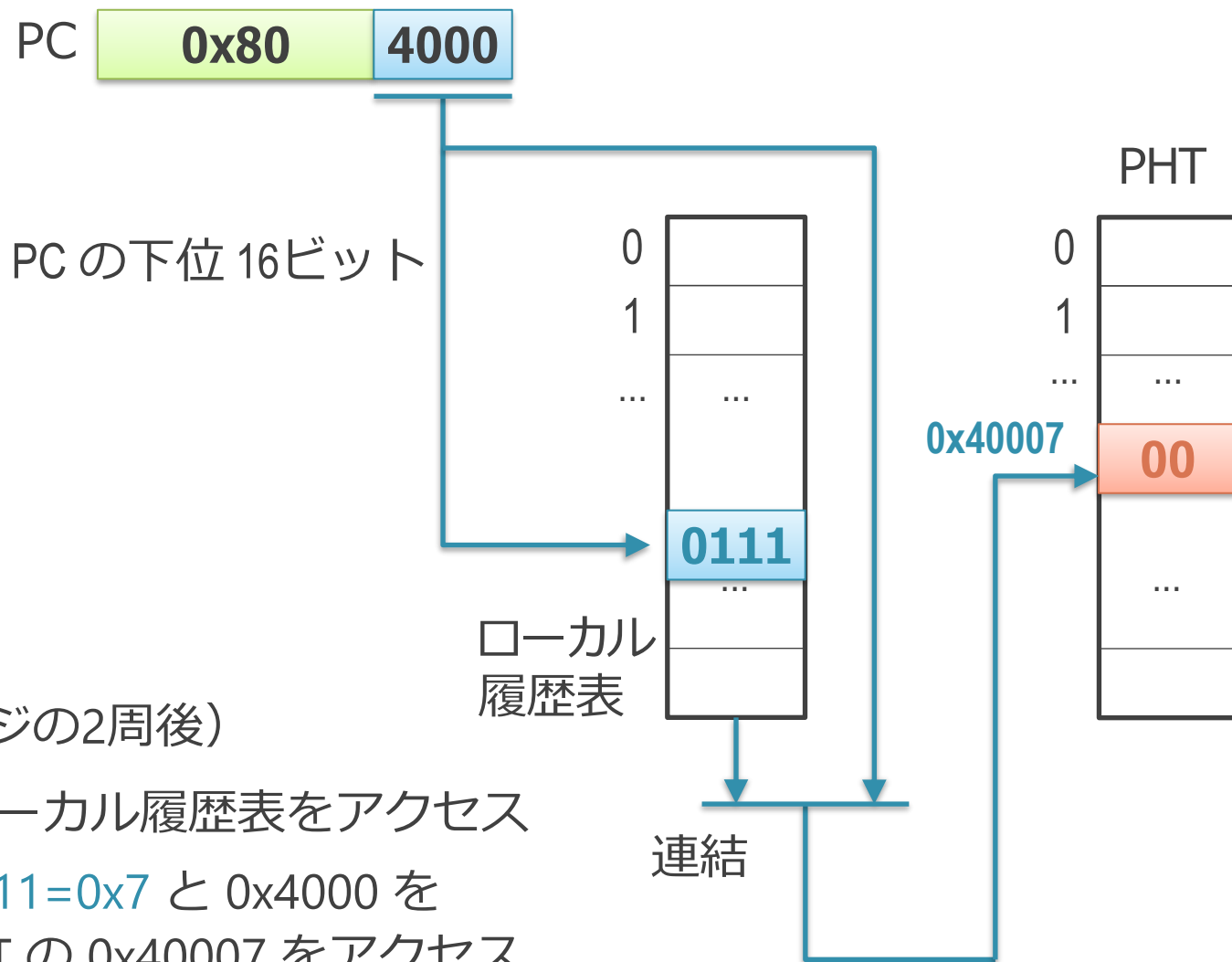
# ローカル履歴予測器の動作例（3）



## 動作：

1. 0x4000 でローカル履歴表をアクセス
2. 得られた 1101=0xD と 0x4000 を結合し, PHT の 0x4000D をアクセス
3. カウンタの中身が 11 なので成立と予測

# ローカル履歴予測器の動作例（3）



## ■ 動作：（前ページの2周後）

1. 0x4000 でローカル履歴表をアクセス
2. 得られた **0111=0x7** と 0x4000 を結合し, PHT の 0x40007 をアクセス
3. カウンタの中身が 00 なので不成立と予測

# ローカル履歴予測器のメリット

- 特定の PC の分岐方向にパターンがある場合, 有効に働く
- たとえば,
  - ◇ 成立と不成立を交互に繰り返す
  - ◇ 短い for ループ

# 履歴を用いた予測器

- この履歴の保持方法/作り方の違いで、いくつかの方法がある

1. ローカル履歴予測器

- PC ごとに履歴を保持

2. グローバル履歴予測器

- PC を区別せず履歴を保持

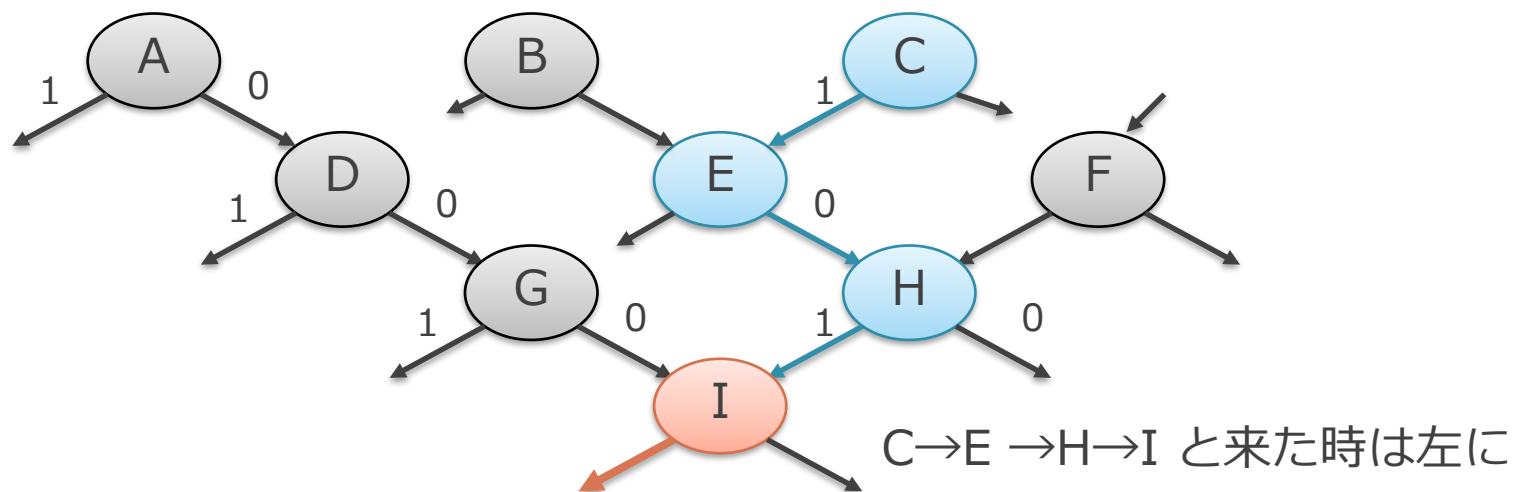
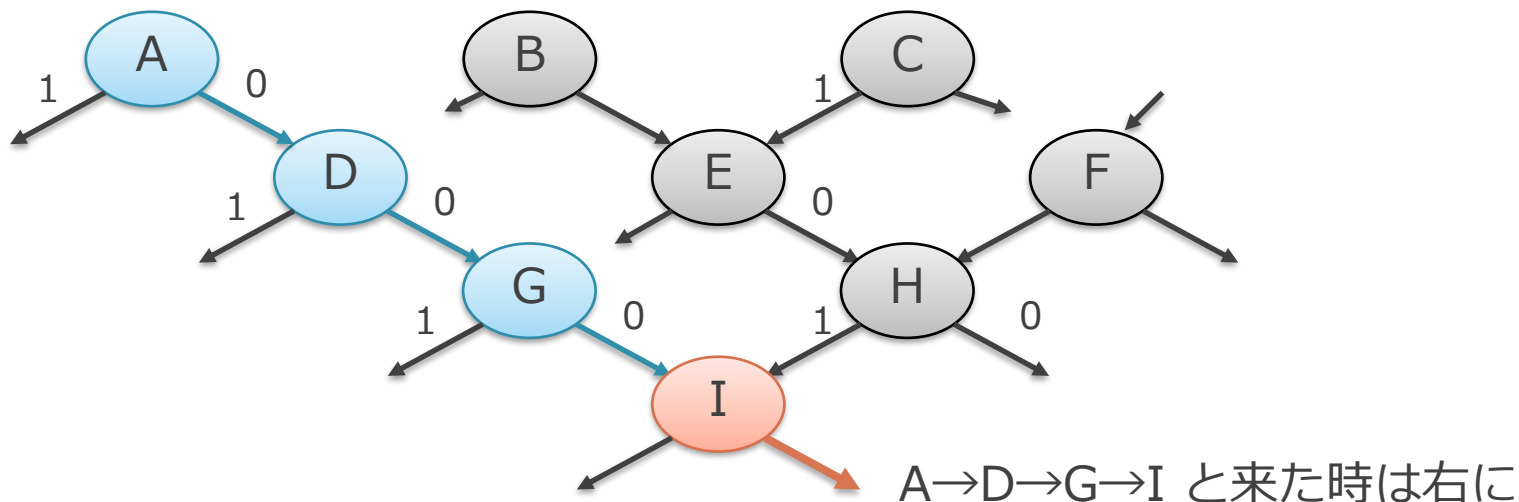
# グローバル履歴予測器のモチベーション

```
if (option == 1) {  
    ...  
}  
  
...  
if (option != 1) { // 上の if と必ず反対になる  
    ...  
}
```

- 複数の分岐間に相関があることが結構多い
  - ◇ ある分岐が成立したら、その後ろにある別の分岐は不成立... など
- こう言う相関を拾いたい



# グローバル履歴予測器のイメージ



- 「こういうパスを通ってきたときは、成立 or 不成立になりやすい」をうまく予測したい

# ローカル履歴予測器とグローバル履歴予測器

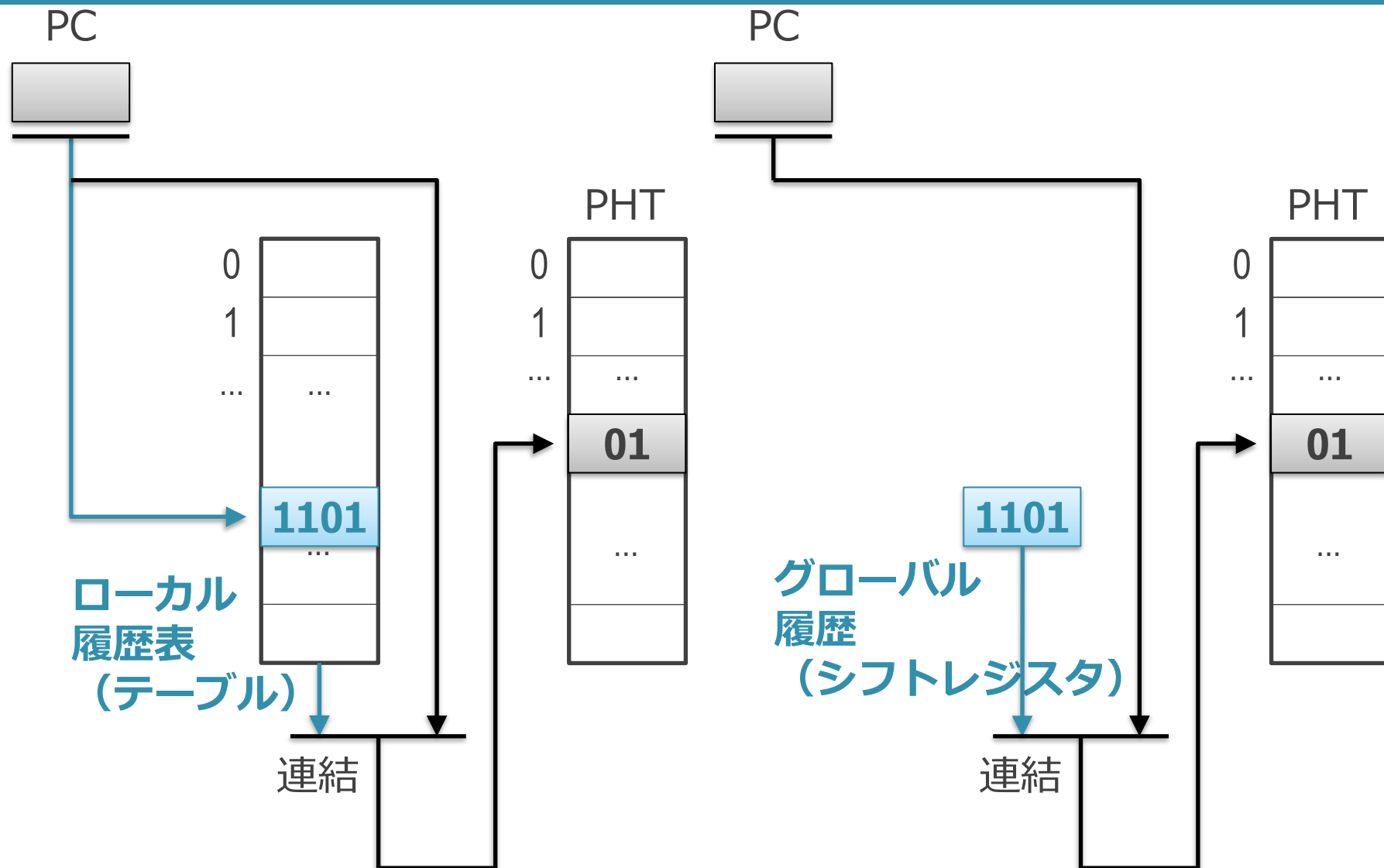
## ■ ローカル履歴予測器

- ◇ 各静的分岐のアドレスごと（ローカル）に，分岐方向の履歴を保持

## ■ グローバル履歴予測器

- ◇ 各静的分岐を区別せず（グローバル）に，分岐方向の履歴を保持
- ◇ 直前に実行した分岐の方向をどんどん保存していく
- ◇ あとはローカル履歴予測器と同じ

# ローカル履歴予測器とグローバル履歴予測器



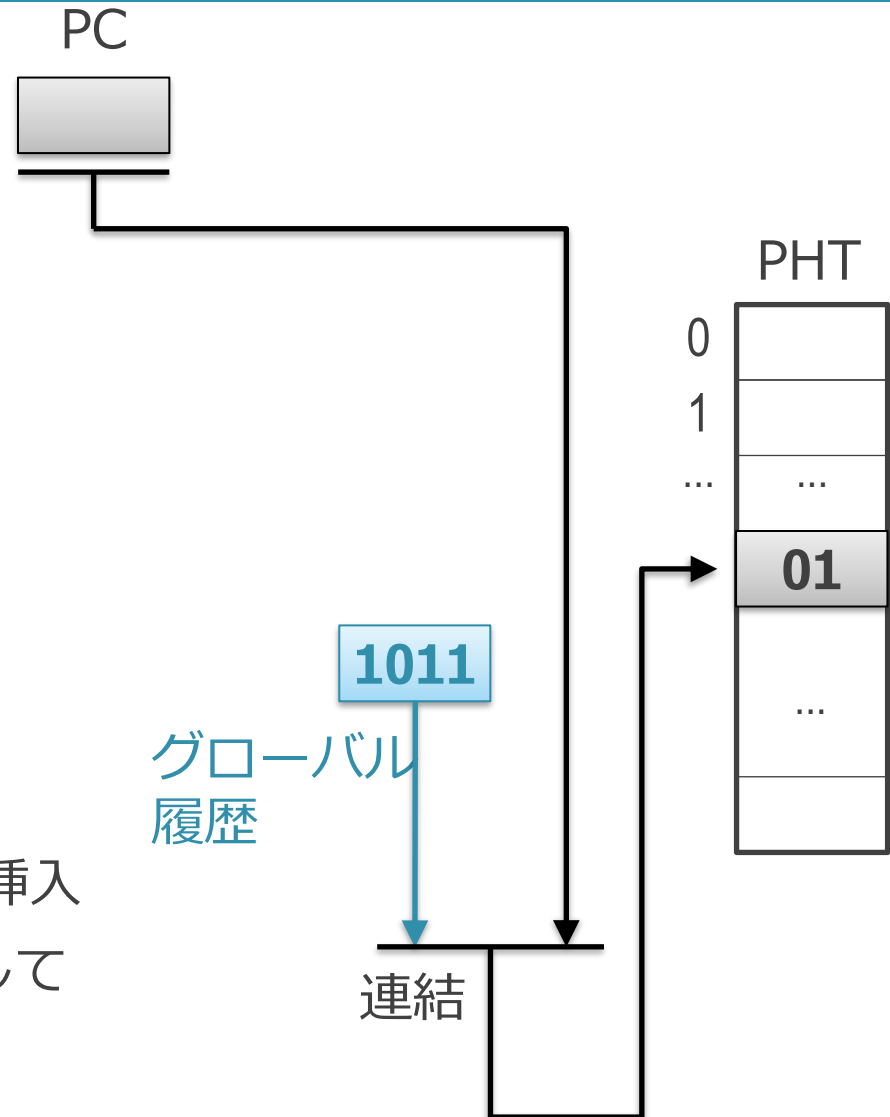
- 1 エントリのローカル履歴表を全員で共有しているイメージ

# グローバル履歴予測器

```
if (option == 1) {} // 1
if (option != 1) {} // 0
if (...) {} // 1
if (...) {} // 1
if (...) {} // ???
```

## ■ 動作 :

- ◇ 分岐命令が実行されるたびに、分岐方向をグローバル履歴に挿入
- ◇ グローバル履歴と PC を連結して PHT にアクセスし、予測



# グローバル予測器の利点

## ■ 利点：

1. 異なる静的分岐の間にある相関を拾える
  - 例：～行目の if と ～ 行目の if は常に同じ方向
  - ローカル履歴予測器では拾えない
2. ローカル履歴に対応する相関も拾える
  - 直前に実行された動的分岐の方向を区別なく使用
  - なのでループのような同じアドレスの分岐も内包している

# 履歴長と予測精度

- 一般に、グローバル履歴長を長くするほど精度はあがる
  - ◇ より遠い分岐の相関が拾えるようになる
  - ◇ 履歴長が1000以上のところに相関がある場合もある
    - ある関数で分岐した後、色んな所にいったてまた来るとか
- 実際にはハードウェア（特に PHT の大きさ）の制約がある
  - ◇ 1 サイクル内にアクセス可能な大きさに限られる
    - 最大数K エントリ程度
    - （最近はもうちょっと大きいかも）
  - ◇ 履歴長に対し、2 の累乗のオーダーでエントリ数が増加

# g-share 予測器

## ■ グローバル履歴予測器の一種

- ◇ より長い履歴長でも, エントリ数が大きくならないようにしたもの

## ■ モチベーション: グローバル履歴のパターン自体に偏りがある

- ◇ たとえば履歴長が16ビットだとして, 2の16乗の全てのパターンは通常現れない

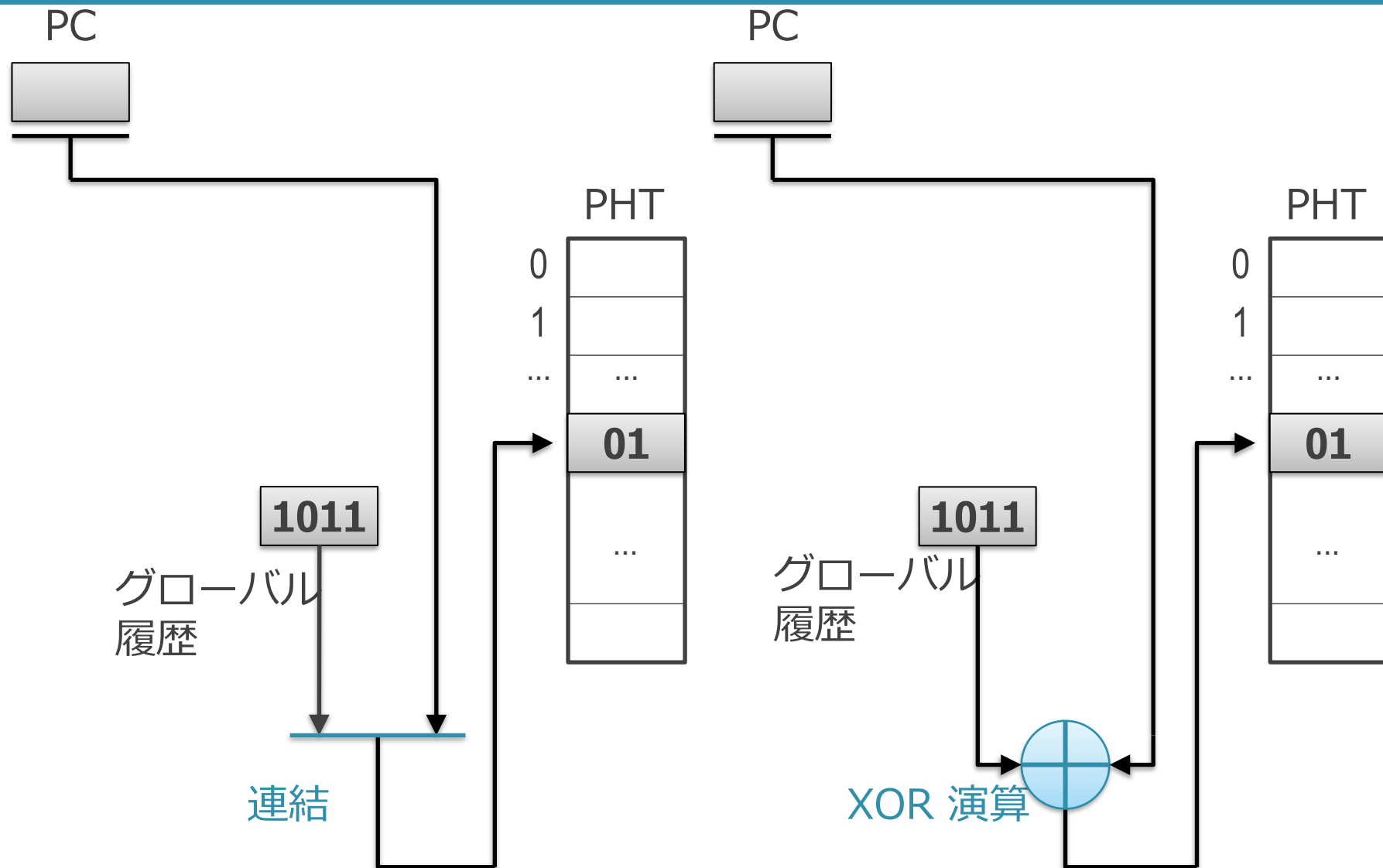
- ◇ 単純に PC と連結すると, 使われないエントリの方が圧倒的に多い

## ■ ビット連結ではなく, XOR 演算により結合

- ◇ ビット連結: PC 下位16ビット + 履歴16ビット → 32ビット

- ◇ XOR 演算: PC 下位16ビット + 履歴16ビット → 16ビット

# g-share 予測器



- ビットを単純に連結するかわりに, XOR 演算して結合



# なぜ XOR 演算なのか？

AND

$a$	$b$	$z$
0	0	0
0	1	0
1	0	0
1	1	1

OR

$a$	$b$	$z$
0	0	0
0	1	1
1	0	1
1	1	1

XOR

$a$	$b$	$z$
0	0	0
0	1	1
1	0	1
1	1	0

## ■ 要求：

- ◇ 軽量の演算であること → 論理演算が良い
- ◇ 2つの値がよく混じってくれること  
(0と1が均等に現れること)

## ■ 要求を満たす論理演算は、XOR か XNOR しかない

- ◇ AND や OR では、結果が0か1に偏る
- ◇ それ以外は、 $a$  か  $b$  そのものか、それらの反転になってしまう<sup>89</sup>

# 分岐方向予測

1. 静的分岐予測
2. 動的分岐予測
  1. n ビット・カウンタ
    1. 1ビット・カウンタ予測器
    2. 2ビット・カウンタ予測器
  2. 履歴を用いたもの
    1. ローカル履歴予測器
    2. グローバル履歴予測器
    3. より高度な予測器

# より高度な予測器

1. ローカル・グローバルのハイブリッド予測器
2. パーセプトロン予測器
3. TAGE 予測器

- パーセプトロンと TAGE は基本的にはグローバル予測器が下敷き
  - ◇ XOR 演算は要素としてよく出てくる

# 予測器の精度

- 左上が g-share, 右下が TAGE の最新型

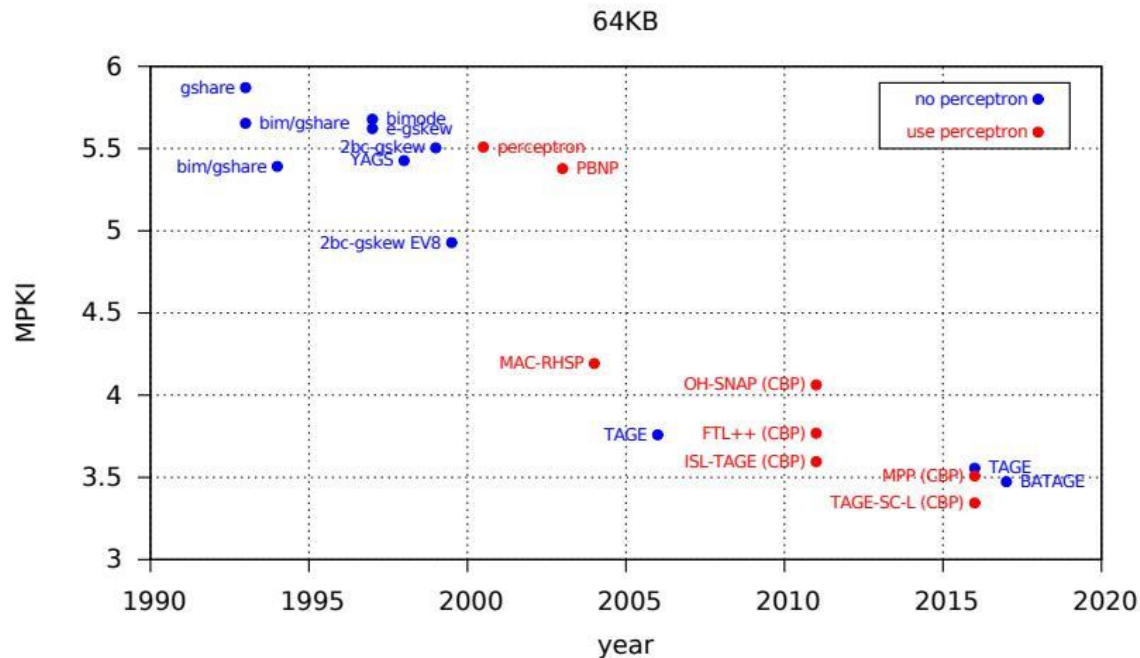


Figure 1: Average number of mispredictions per 1000 instructions (MPKI) for various conditional branch predictors on the CBP 2016 traces for 8KB, 32KB and 64KB storage budgets (see Appendix A).

# ローカル・グローバル・ハイブリッド予測器

- ローカル予測器とグローバル予測器のそれぞれが得意な分岐がある
  - ◇ 基本的にはグローバルの方が強い
    - グローバル予測機はローカルなパターンもうまく予測できる
  - ◇ ローカルが得意な分岐の例：間隔が長い場合
    - ある関数内の if 文は成立と不成立を交互に繰り返す
    - その関数はかなり時間をあけて呼ばれる
    - グローバル予測器では相当長い履歴が必要
- アプローチ
  - ◇ ローカル予測器とグローバル予測器を両方積んで、使い分ける

# ローカル・グローバル・ハイブリッド予測器

## ■ 要素

1. ローカル予測器
2. グローバル予測器
3. セレクタ

- PC をインデクスとしてアクセスされるカウンタのテーブル

## ■ 学習の動作

- ◇ 1. ローカル予測器と 2. グローバル予測器で並列に予測

- ◇ セレクタの更新

- 1. が当たってたらセレクタの対応エントリをデクリメント
- 2. が当たってたらセレクタの対応エントリをインクリメント

## ■ 予測時の動作

- ◇ セレクタの対応エントリと閾値を比較してどちらを使うか決定

# ローカル・グローバル・ハイブリッド予測器

- 問題点：容量効率が悪い
  - ◇ ローカルとグローバルが二重に存在
  - ◇ セレクタが追加される
- それほど予測精度は改善しない
  - ◇ しかし機構が割と単純なので結構いろんな CPU に乗っていた

# 方向分岐予測器のまとめ

- 分岐予測器
  - ◇ 静的予測
  - ◇ 動的予測
- 原始的なものから，最近の CPU で使われているものまで紹介
- 次回はパーセプトロン予測器と TAGE 予測器を紹介



# 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 :
- 意見や内容へのリクエストもあったら書いてください