

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

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）

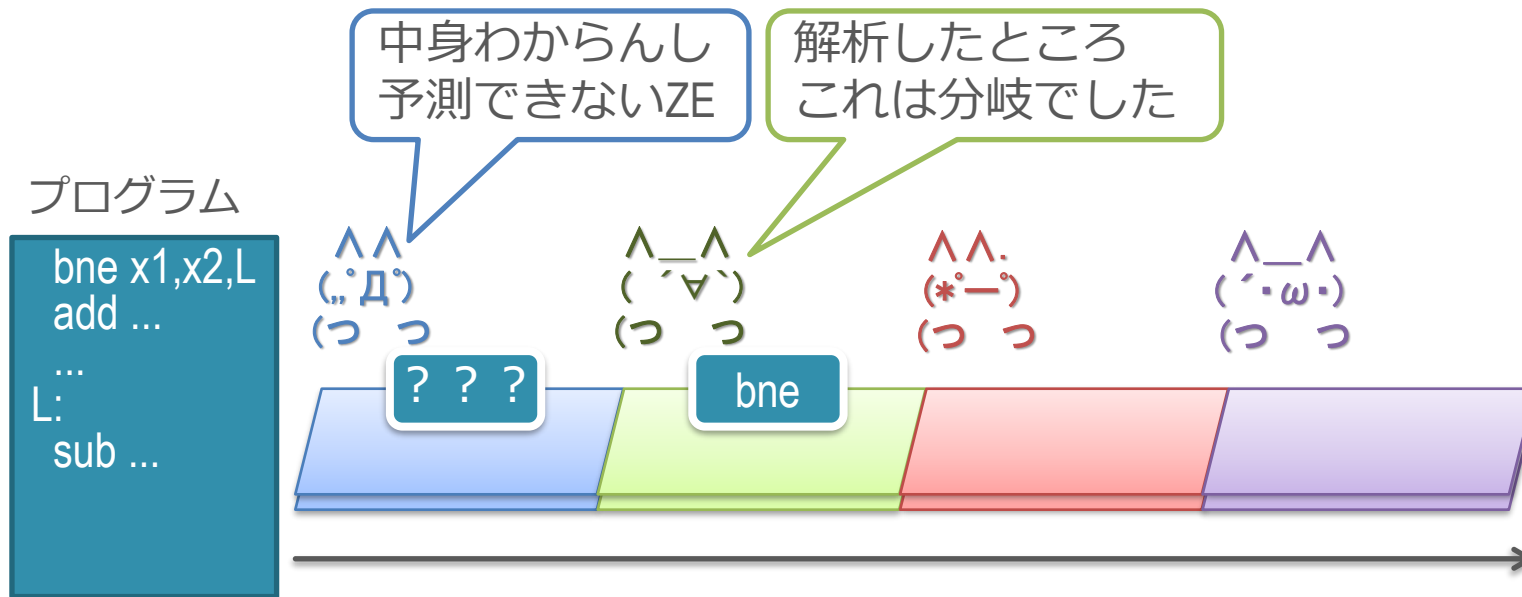
- 分岐先 アドレス or ターゲット
 - ◇ 分岐が成立した際の飛び先のアドレスのこと
- 前方分岐：
 - ◇ 分岐先ターゲットが分岐自身のアドレスよりも大きい分岐のこと
 - ◇ プログラムの進行方向に対して前方に飛ぶことから
- 後方分岐：
 - ◇ 分岐先ターゲットが分岐自身のアドレスよりも小さい分岐のこと
 - ◇ 後方に飛ぶ = ループを作る



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：

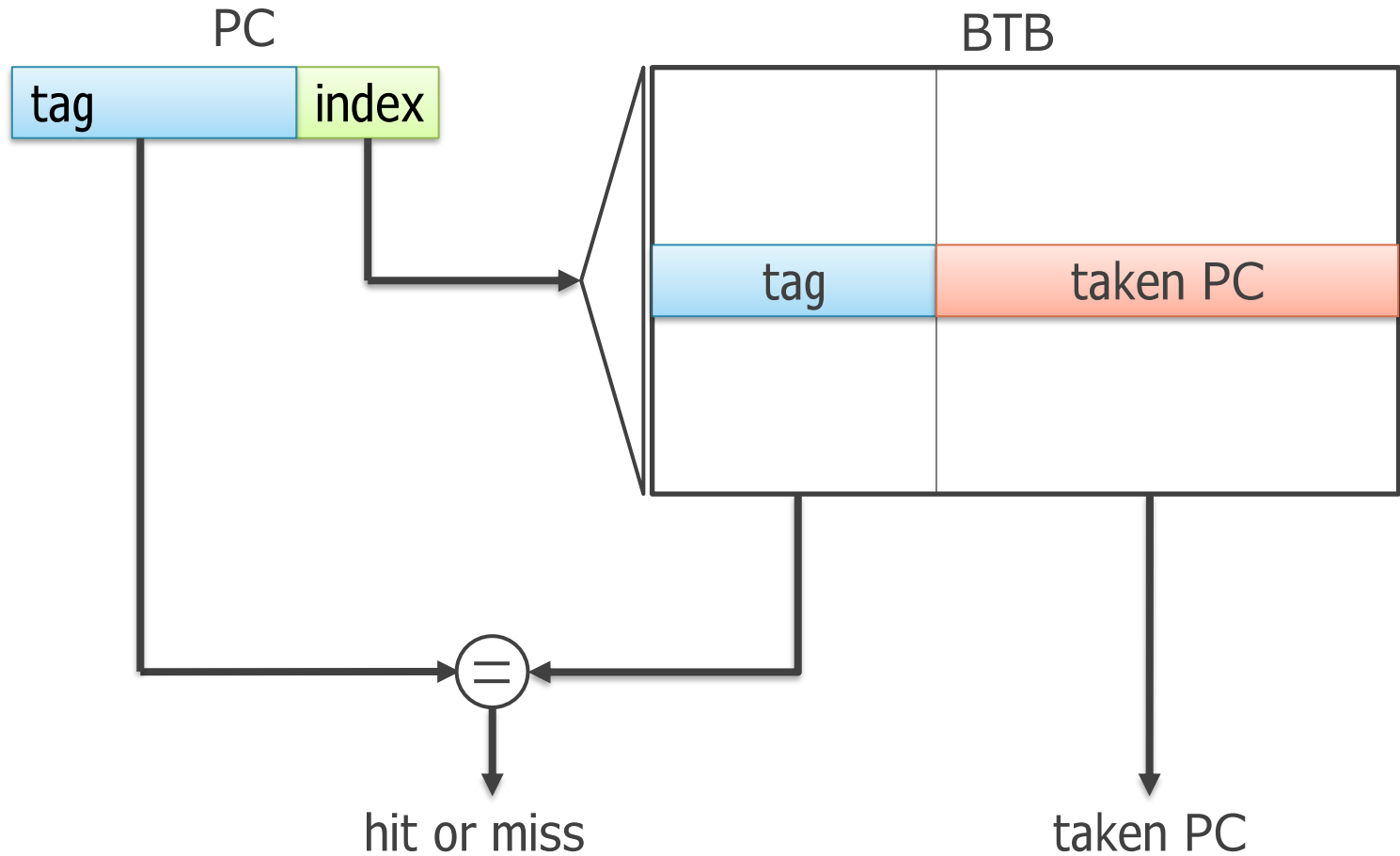
- hit or miss

- 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

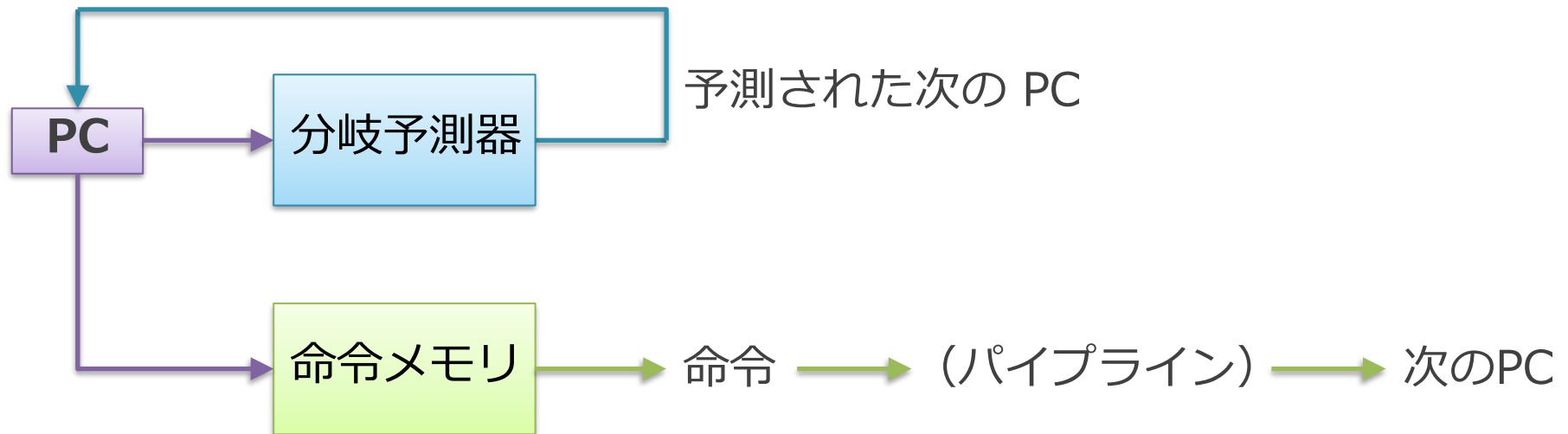
分岐予測の補足

1. 分岐予測器の全体構造
2. パイプラインとしての動作

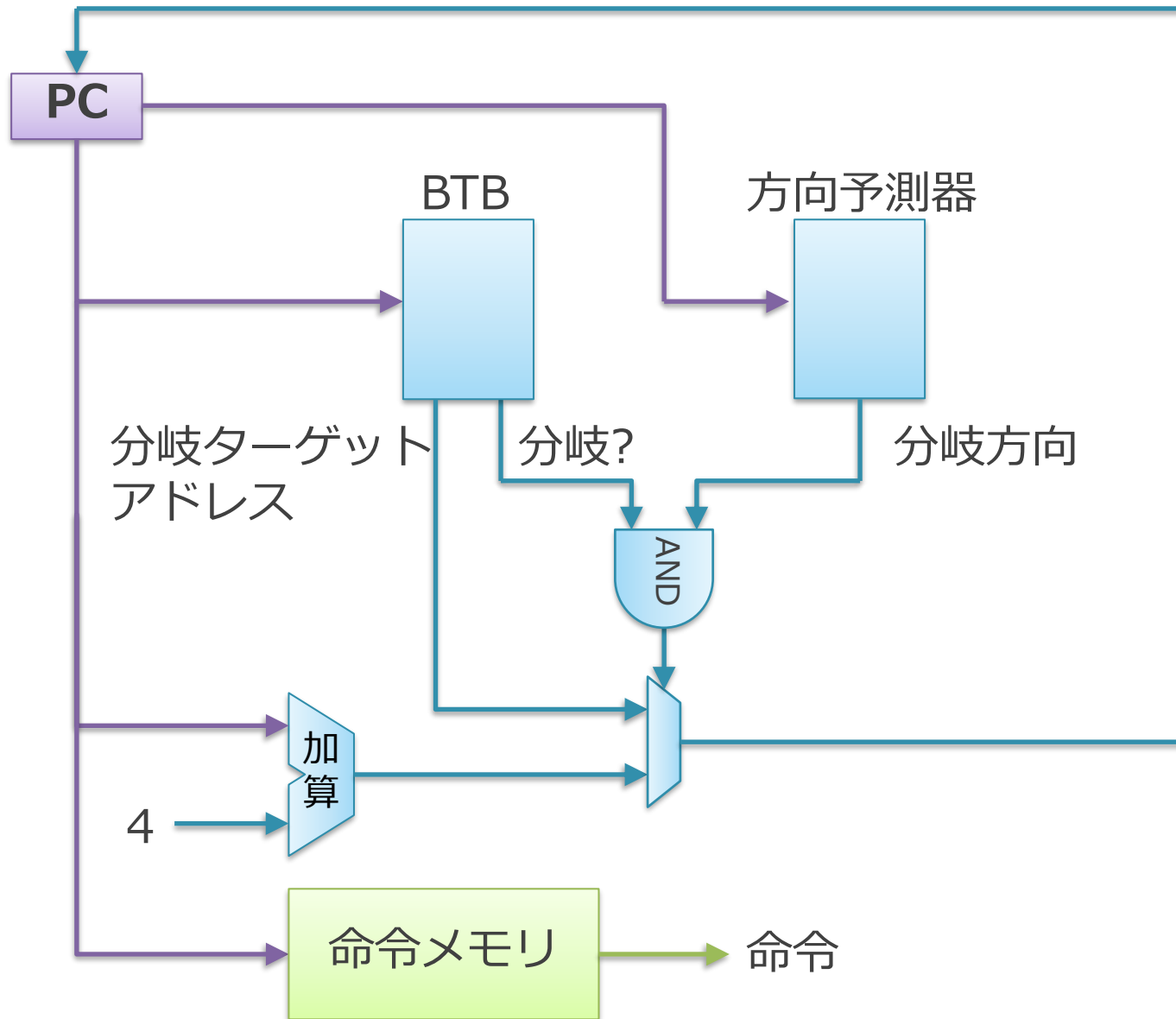
分岐予測器の全体構造

■ 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

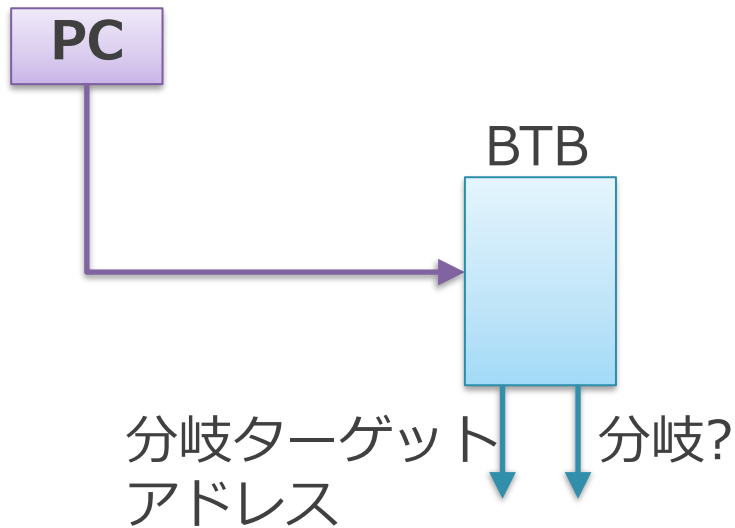


分岐予測器の全体構造



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

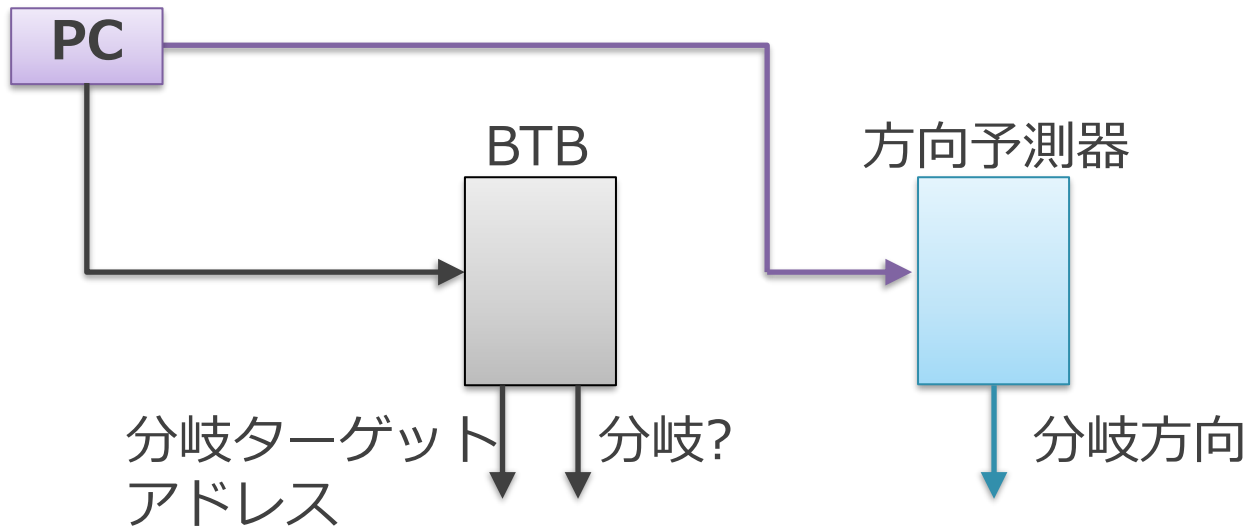


■ BTB により以下を予測

- ◇ 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）

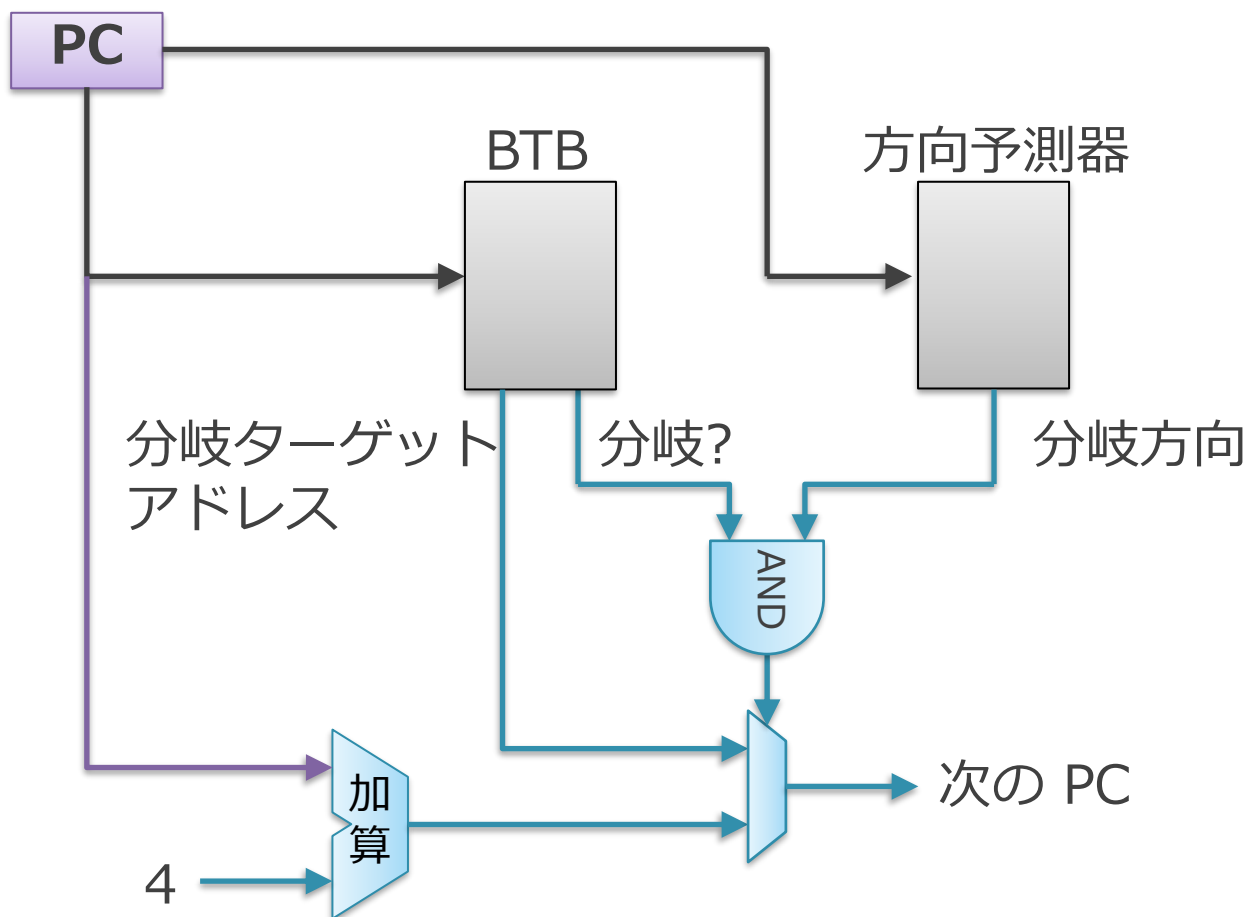
方向予測器による分岐方向の予測



- 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



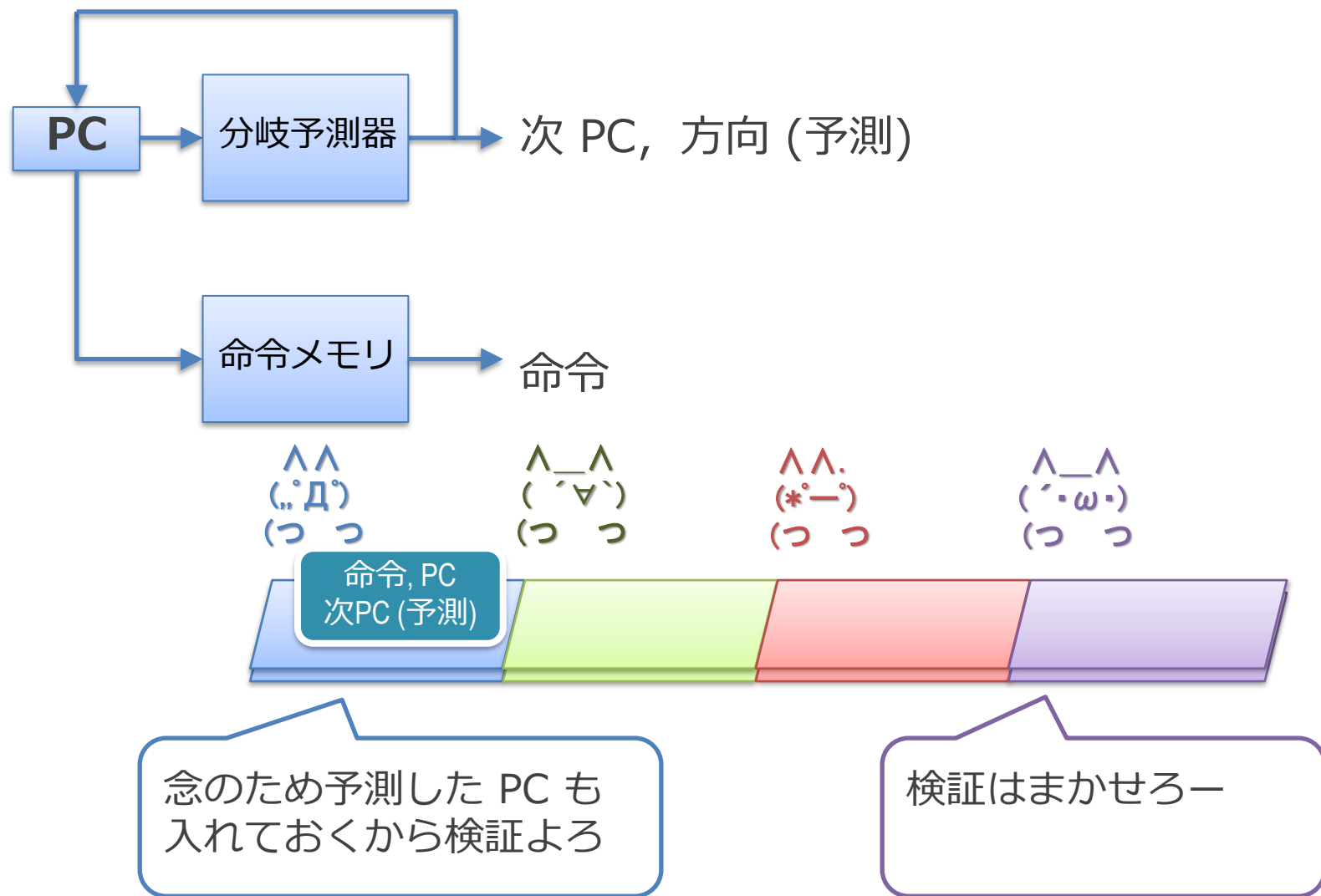
- 次の PC をマルチプレクサにより選択
 - ◇ 分岐命令かつ分岐が成立なら, ターゲット・アドレスを選択
 - ◇ そうでなければ $PC + 4$ を選択

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

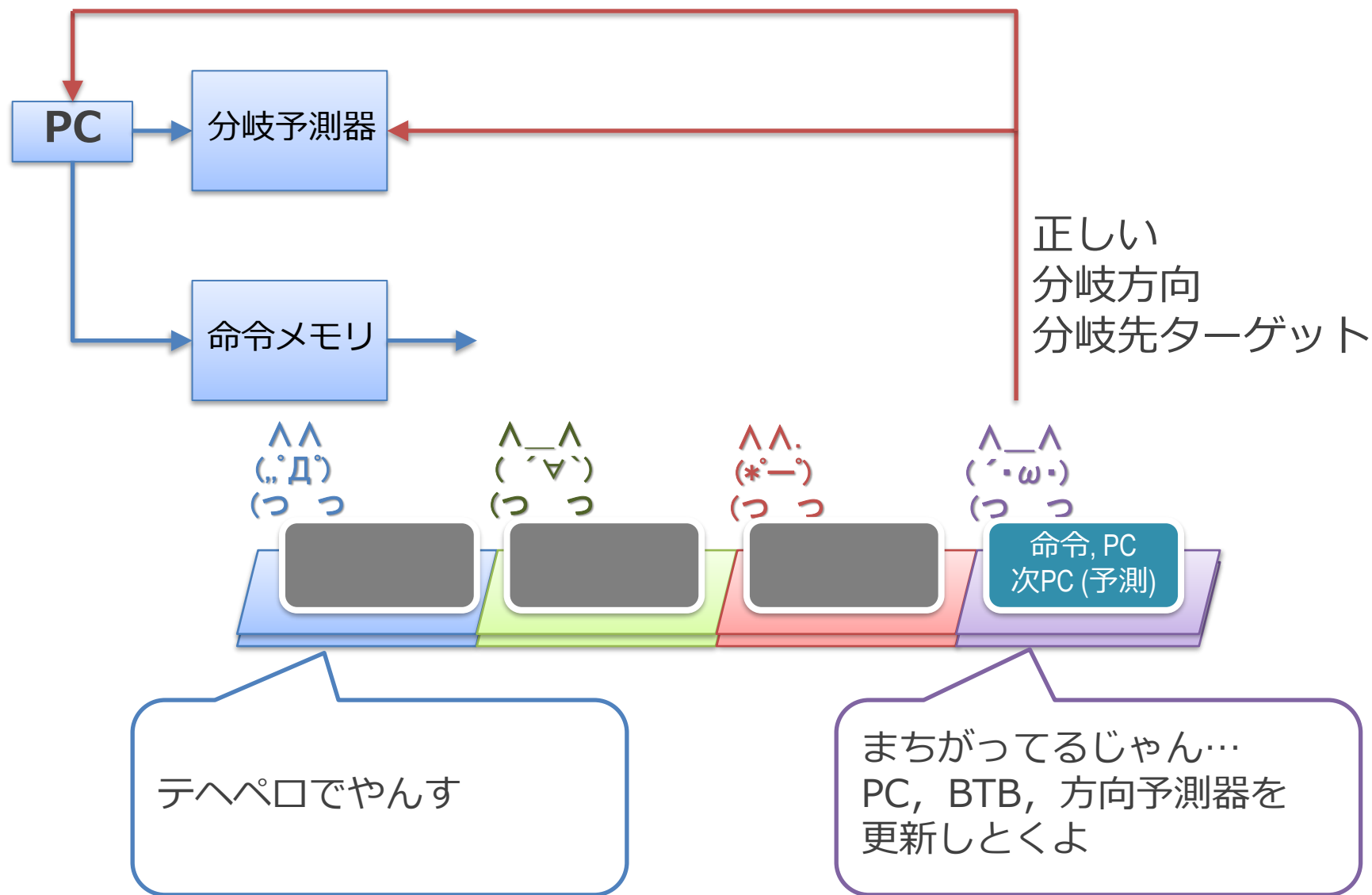
パイプラインとしての動作 (1)

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

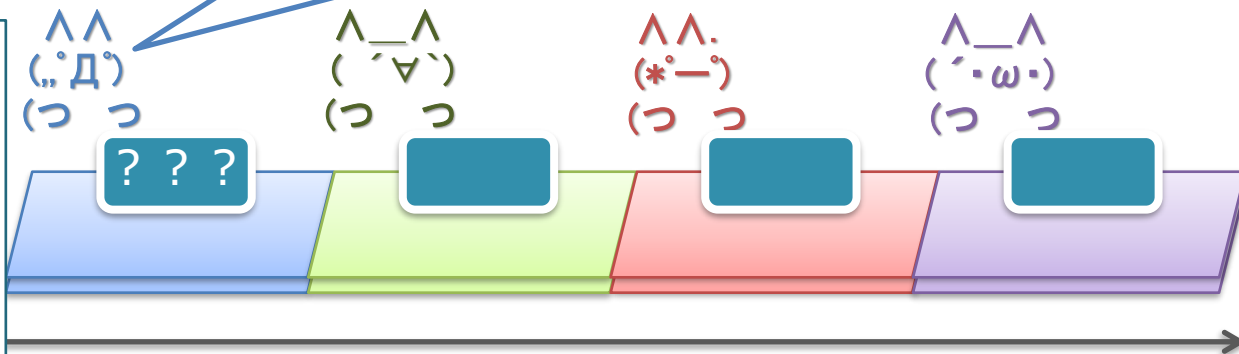
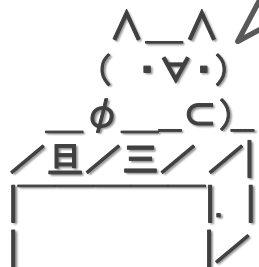
「後方分岐かどうか」 「成立/不成立のヒント」の予測

ヒントをうめておいたので
これでヨシ！

いやその、ここではまだ
中身わからないんですけど...

プログラム

```
bne + hint  
add ...  
...  
L:  
sub ...
```



■ 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？
 - ◇ 後述の動的な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

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

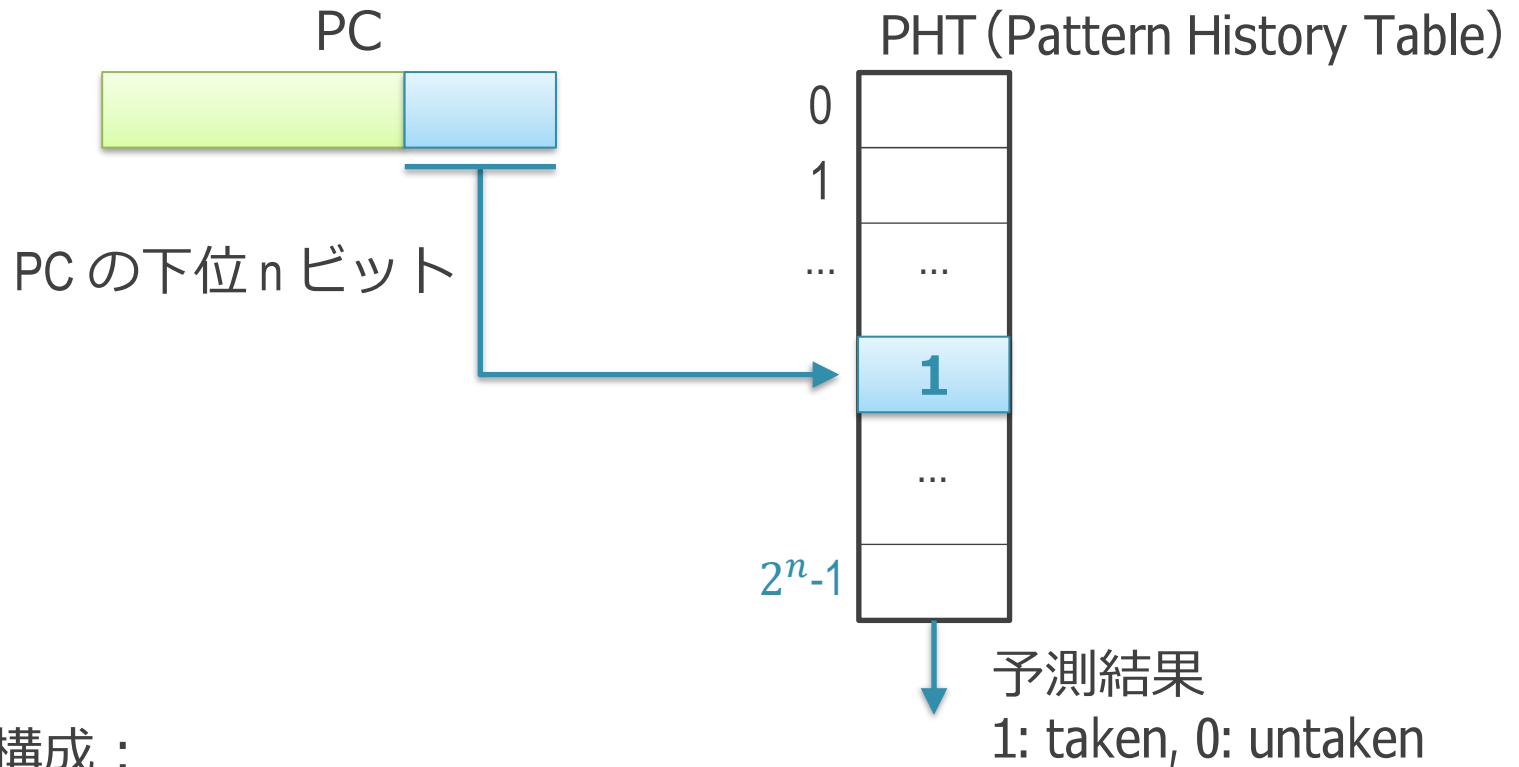
2. 履歴を用いたもの

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

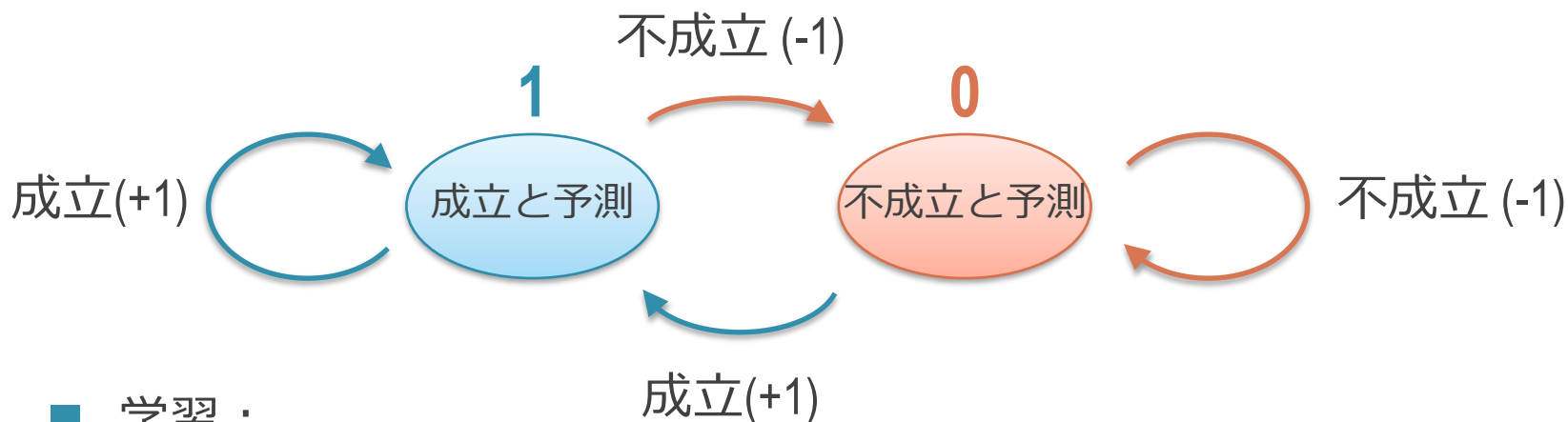
1ビット・カウンタ予測器



■ 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

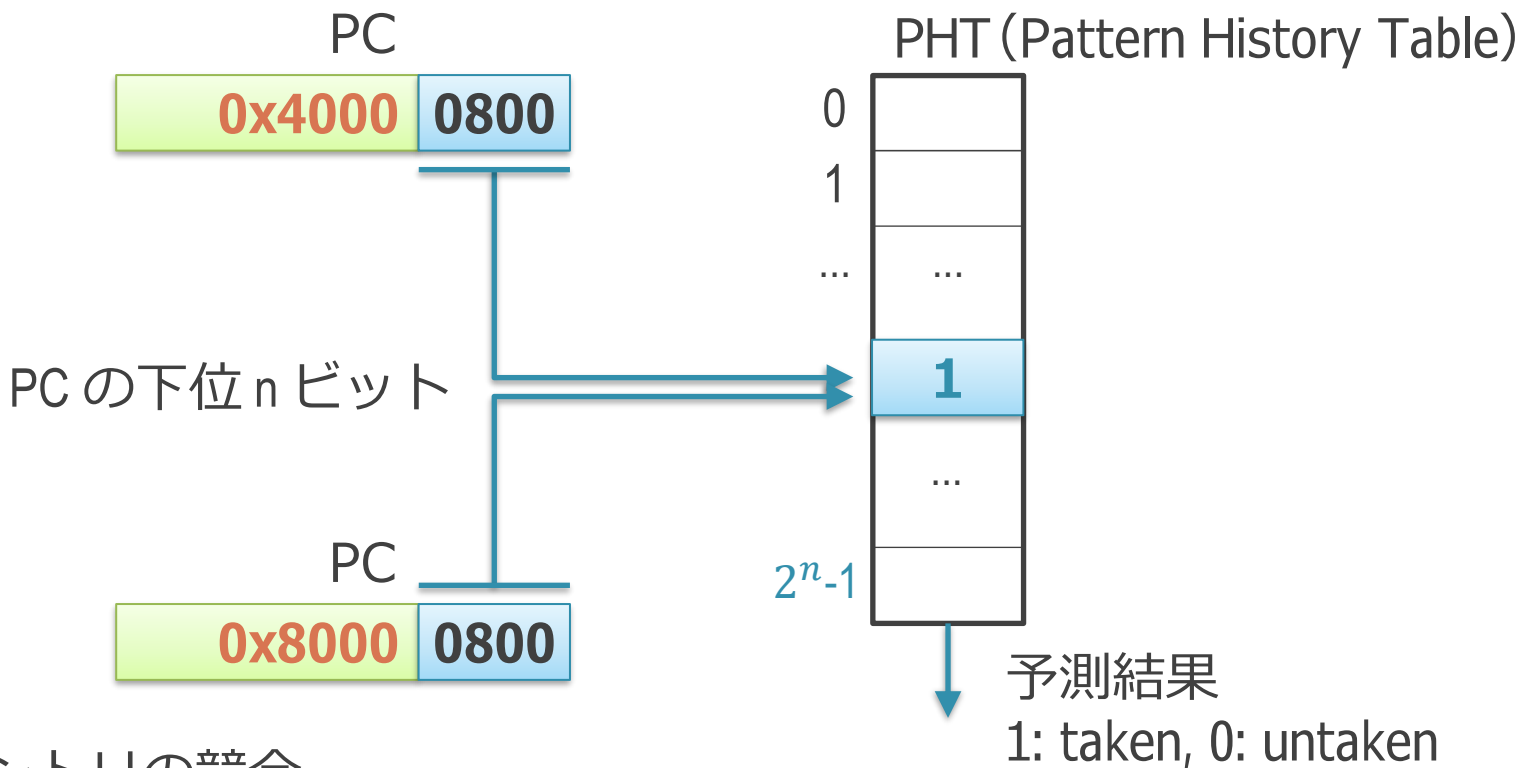
1ビット・カウンタ予測器の意味

- 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

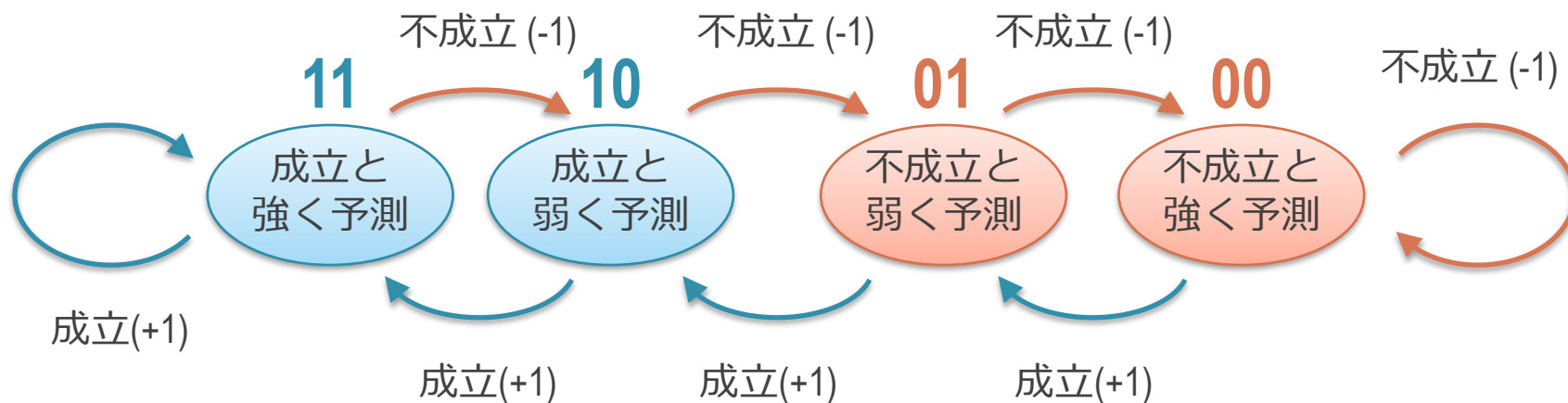
1ビット・カウンタ予測器の問題点：無駄な遷移

- 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

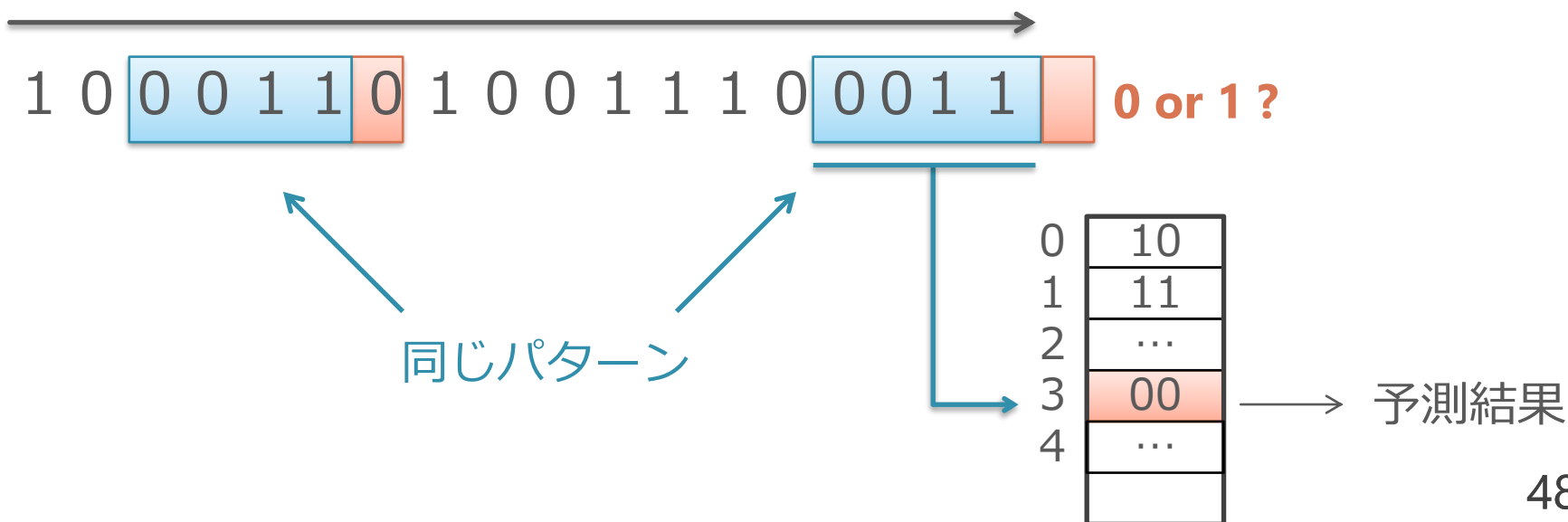
```
for (i = 0; i < 4; i++) {  
    ...  
}
```

- 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

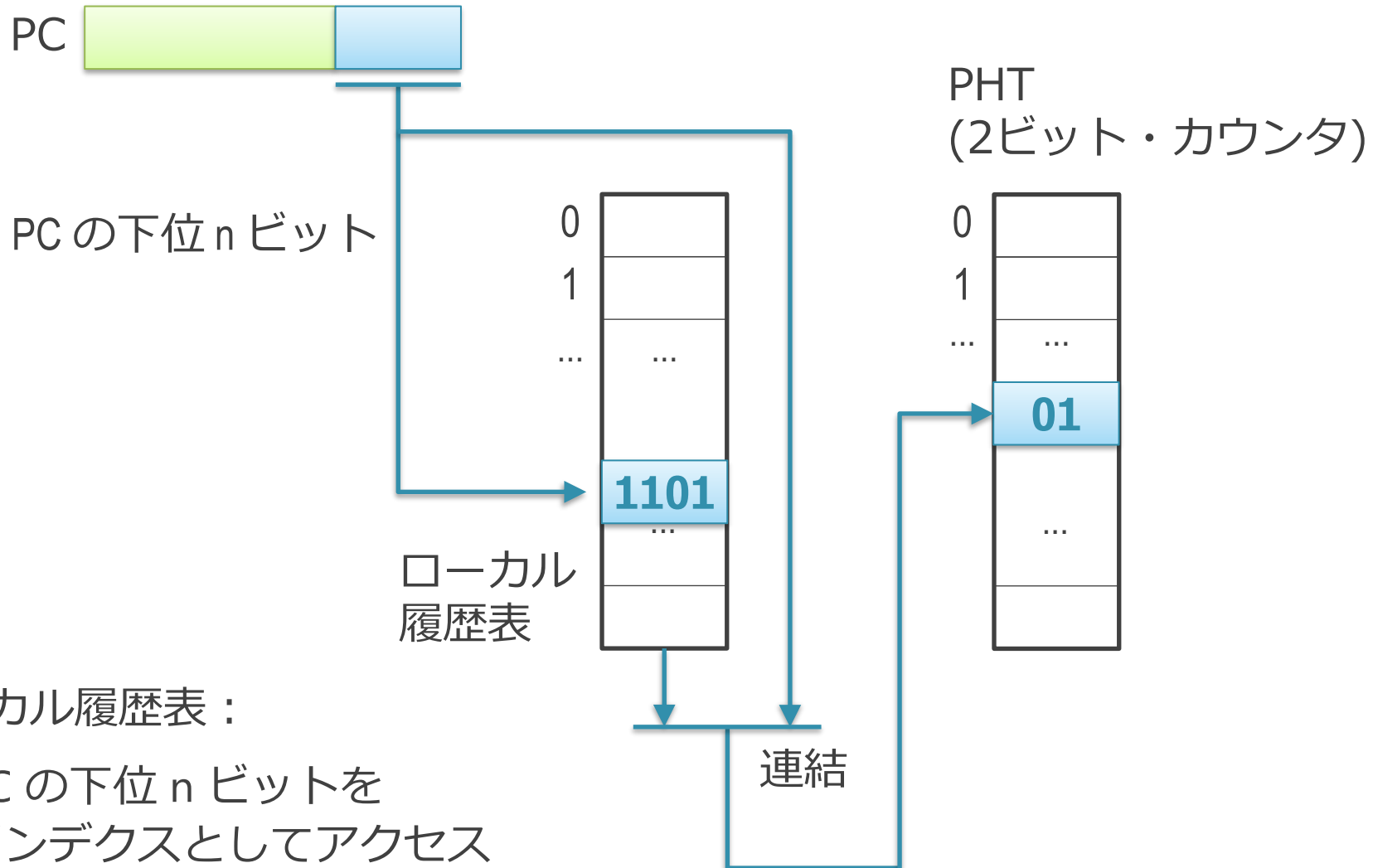
1. ローカル履歴予測器

- 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

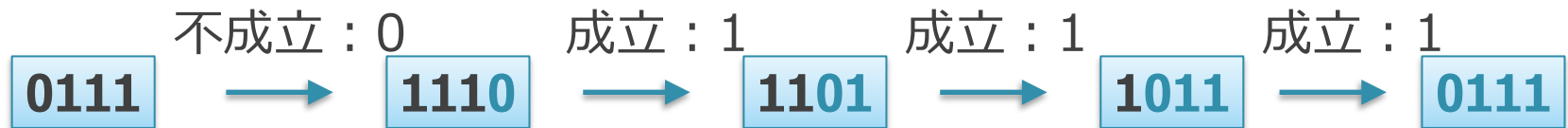


■ ローカル履歴表：

- ◇ 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++) {  
    ...  
}
```

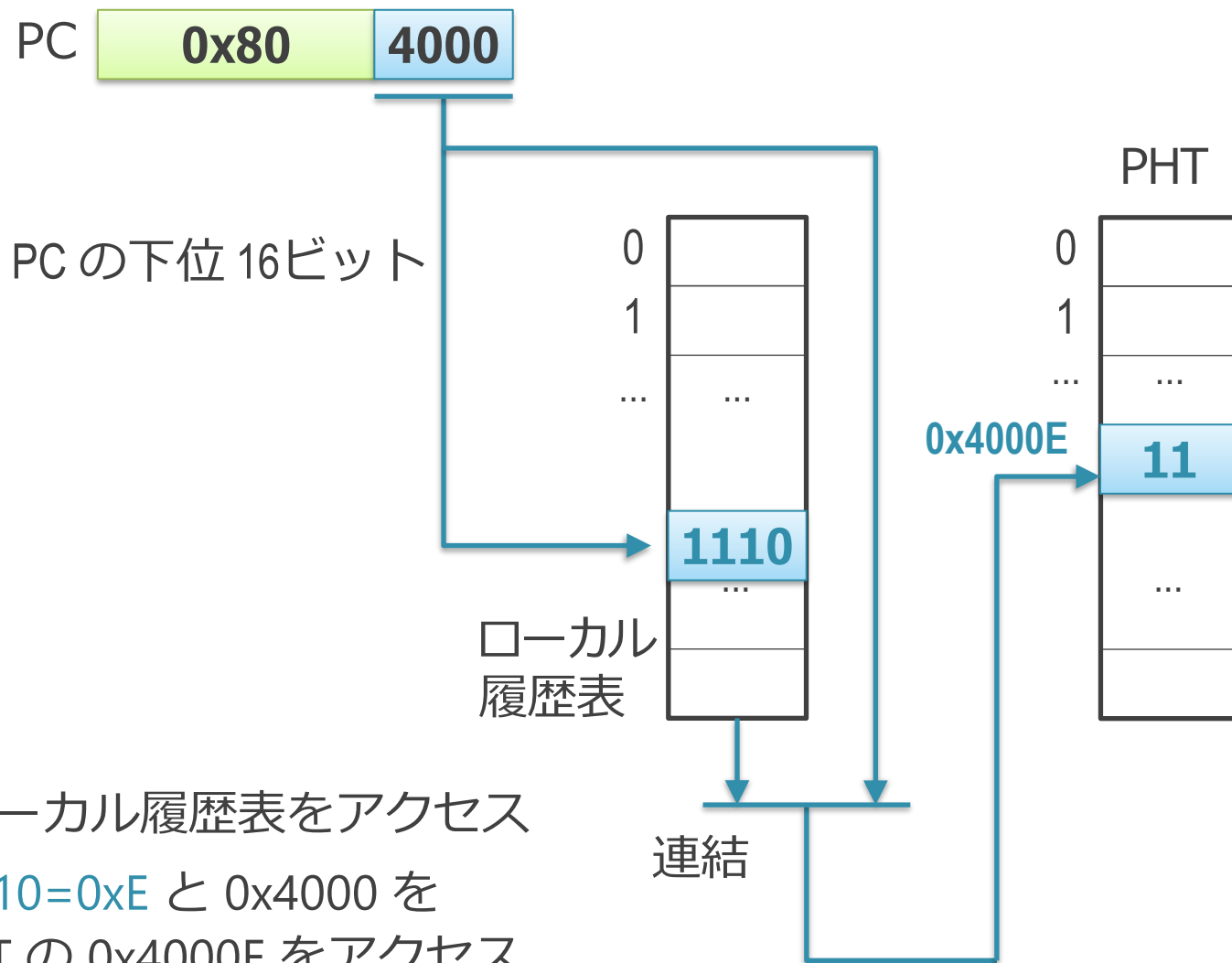
// 4回まわるループ

```
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

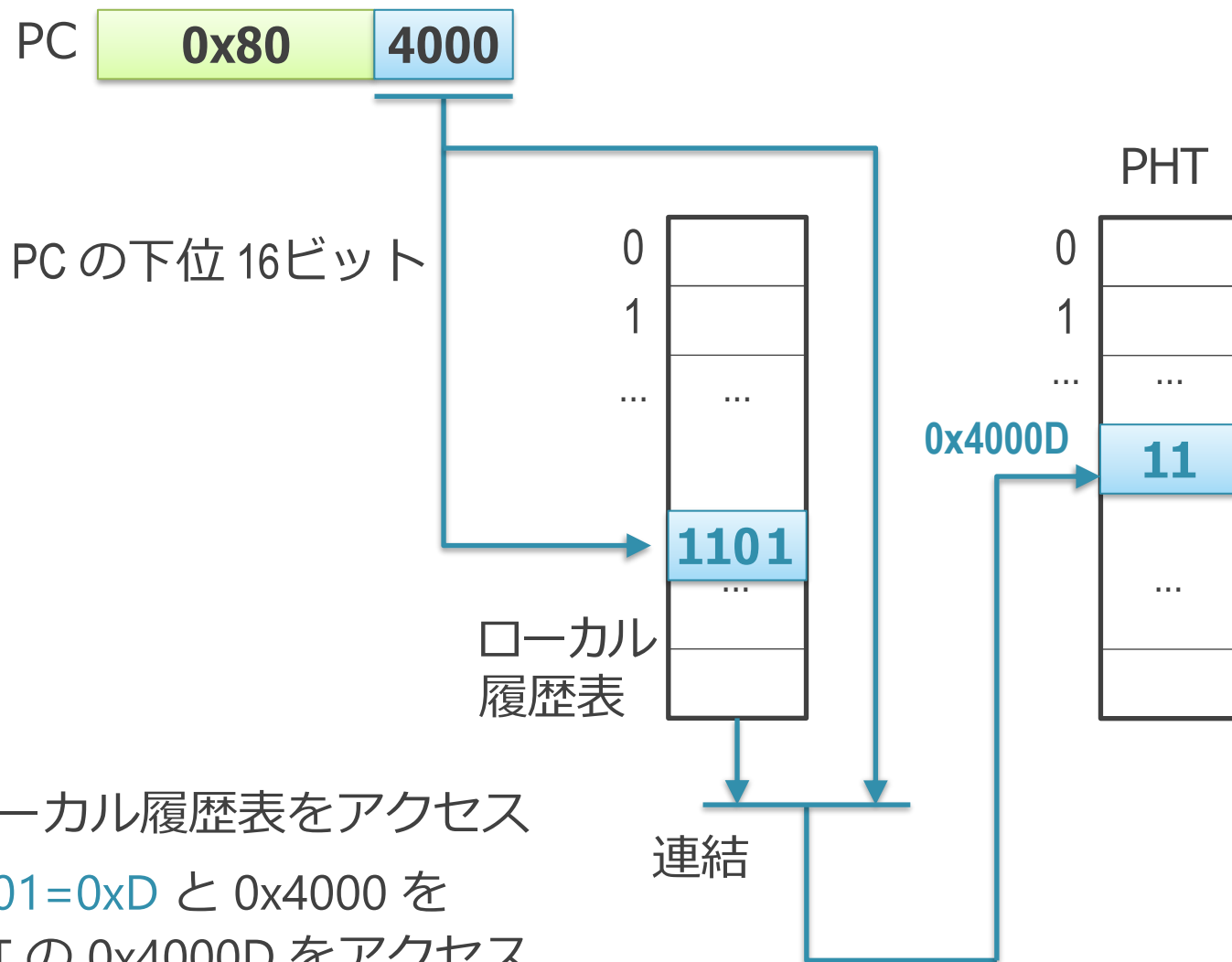
ローカル履歴予測器の動作例（２）



■ 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

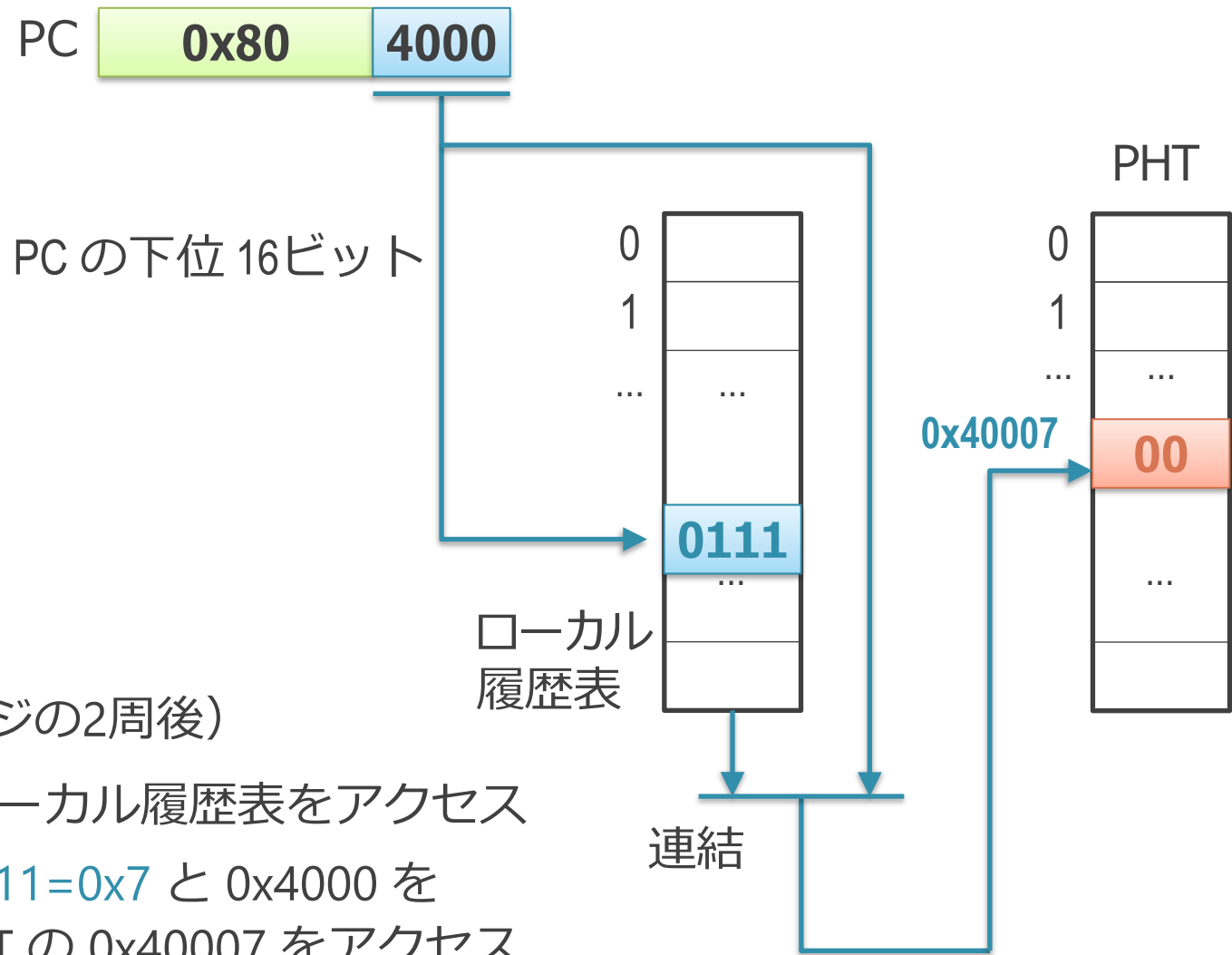
ローカル履歴予測器の動作例（3）



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

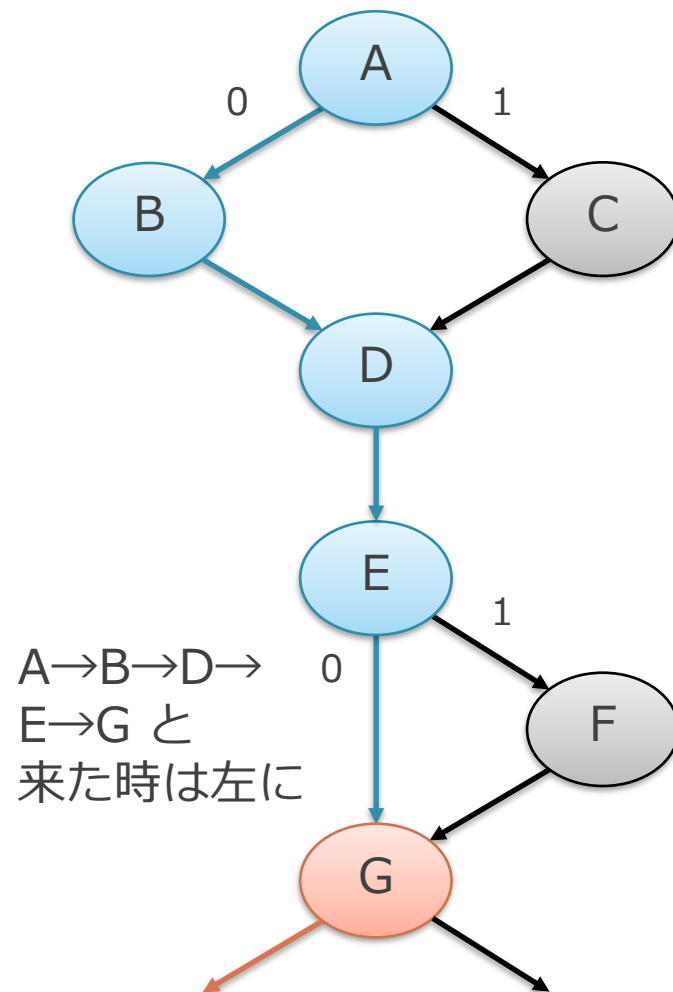
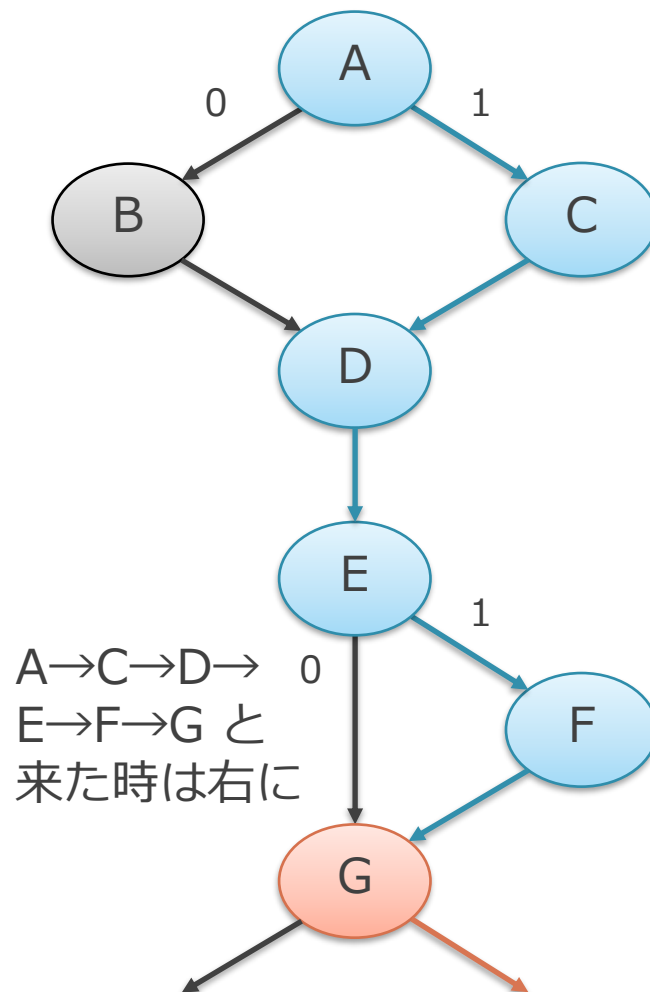
グローバル履歴予測器のモチベーション

```
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

```
if (A) {  
    B...  
}  
else{  
    C...  
}  
D...  
if (E) {  
    F...  
}  
if (G) {
```



- 「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

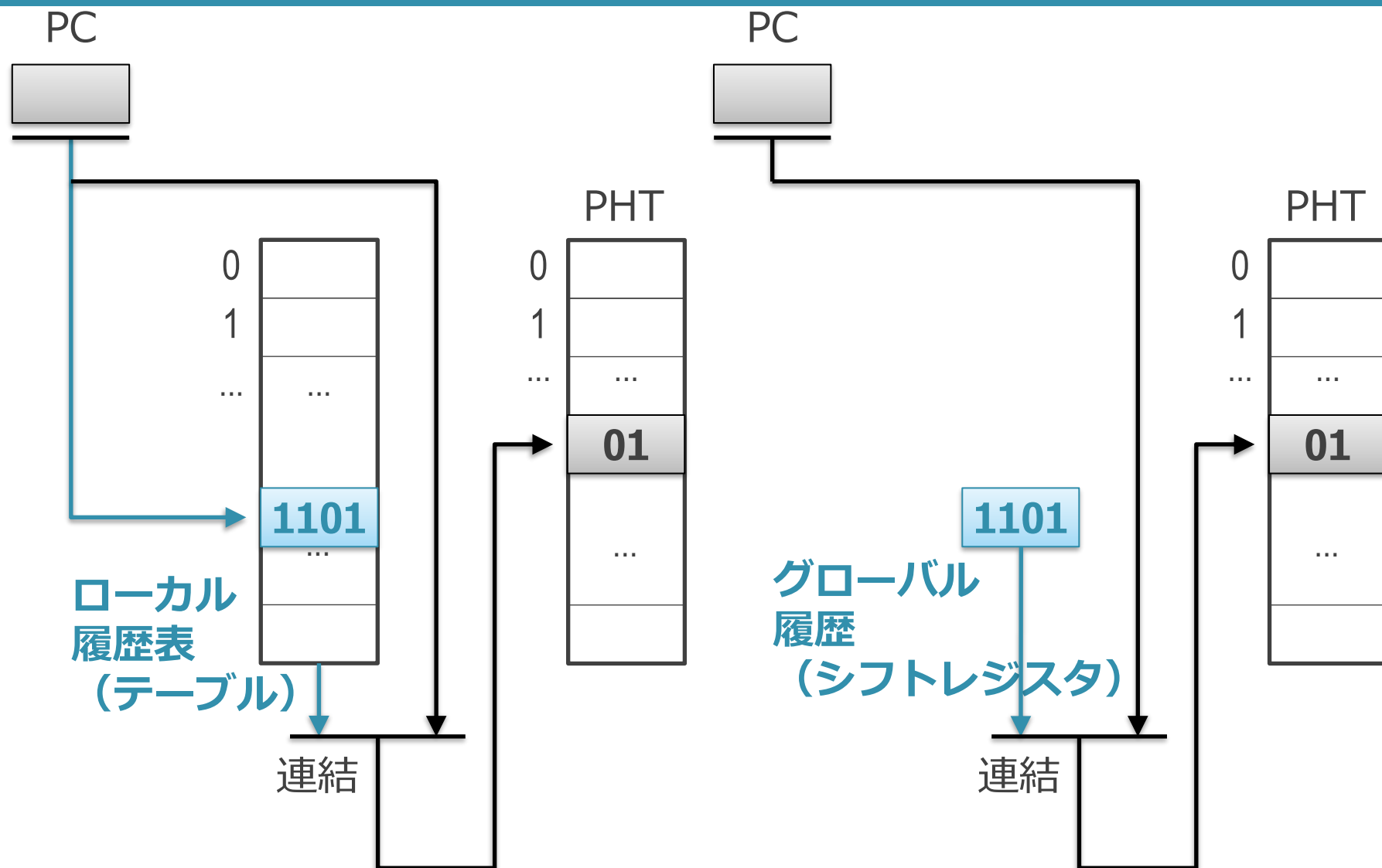
■ ローカル履歴予測器

- ◇ 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



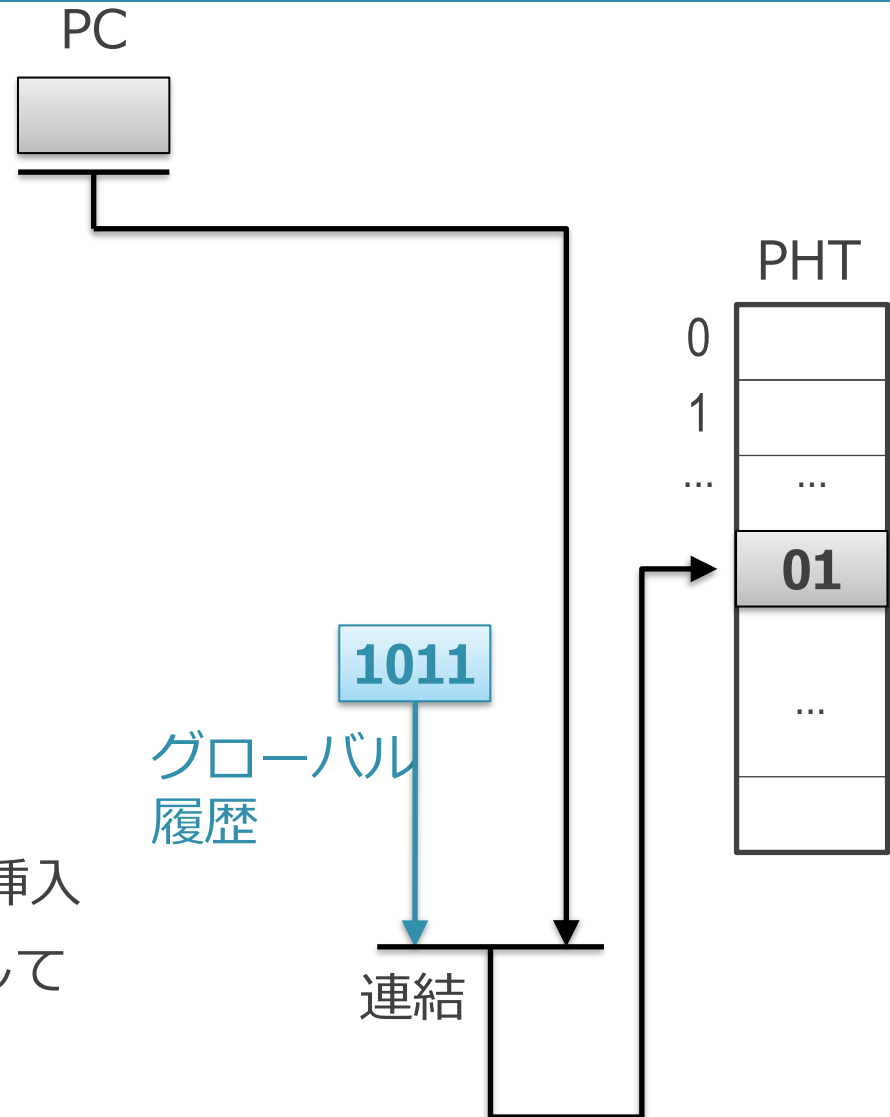
- 1 エントリのローカル履歴表を全員で共有しているイメージ

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

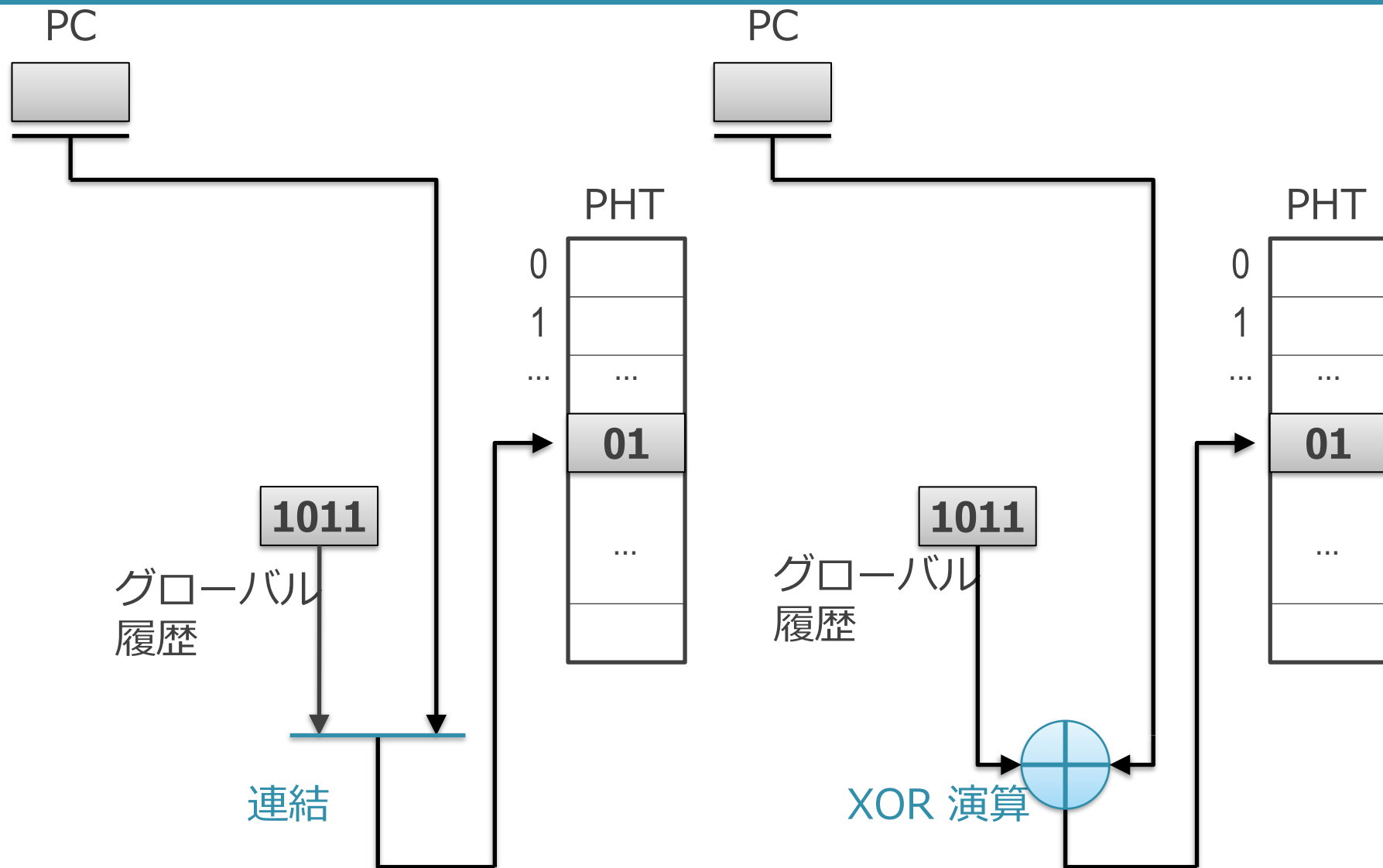
- ◇ 単純に PC と連結すると, 使われないエントリの方が圧倒的に多い

■ 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₆₉

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

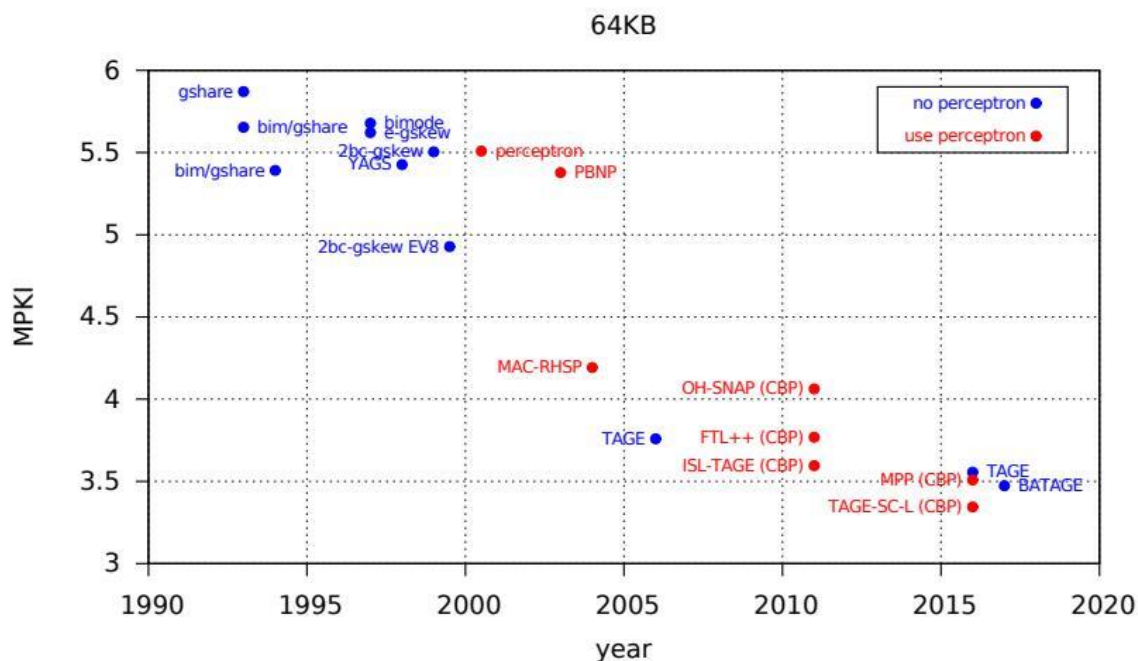


Figure 1: Average number of mispredictions per 1000 instructions (MPKI) for various conditional branch predictors on the CBP 2016 traces for 8KB, 32KB and 64KB storage budgets (see Appendix A).

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

- 分岐予測技術は限界に達していると聞いていますが、新しいワークロードの場合に効率を向上させるためには、hardware software co-designが必要ではないでしょうか。

- パイプラインの段数を大きくしても高速化には限界があるとの話でしたが、分岐予測ミスによるペナルティが大きくなることによるデメリットはないのかなと思いました

- 今日の授業を聞いていると投機実行の実現の大変さとペナルティを考えると投機実行を行うメリットが感じられませんでした。実際に投機実行のあるなしでどれくらいスループットが向上するのでしょうか？ あとCPUを自作する話に興味があるので演習の資料があれば共有していただけると嬉しいです。

◇ <https://github.com/shioya-lab/cpu-exercise>

- BTBは少しキャッシュに似ているという印象を持ちました。

- GPUのマルチスレッディングについてなのですが、NVIDIA GPUのハードウェア用語との対応に混乱しています。例えばNVIDIA A4090には100個のStreaming Multiprocessor があり各SMは64 registers, 96K shared memory, 2048 threadsのリソースを持つという資料を見たのですが、この2048 threadsというのが授業資料p27のTh0-Th3に相当するものでしょうか？
- ◇ NVIDIA GPU におけるスレッドと、前回の講義のスレッドにはちょっと概念のズレがあります
- ◇ 上記ページのスレッドは、NVIDIA の GPU の場合は Warp という単位に概ね対応します
- ◇ 1 1 回目の講義で説明の予定です

- BTBでの分岐予測ってすごく単純に見えるのですが、これが最適なんですか？

- FPGAでCPUを実装した時に疑問に思っていたのですが、基本的な整数演算は1ステージということは、整数演算にかかる遅延より動作周波数を上げることはできないということでしょうか？ もちろん他に時間がかかる処理があれば別ですが、動作周波数を上げようとするとその部分がボトルネックとなるように思います。

- 熱が出るとなんとなくまずそうであることは分かるのですが、CPU に対して具体的にどのような影響が起こるのでしょうか。部品が溶ける？熱揺らぎによるビットエラー？

- ここで言うマルチスレッドはいわゆるIntelとかのHyper Threadingと別の概念ですか？

- 「分岐予測すること」自体にどれほどコストがかかっているのか、知りたいです。過去の実行結果などを参照し、そこから推論することに時間がかかるのかどうか、ということです。
- ◇ 1 サイクル（5GHz で動作する CPU なら 0.2 ナノ秒）～数サイクル以内に予測をしないとイケないので、複雑な推論はできません

- 同じ命令セットを動かそうとした場合、早くしようとするると大体似たようなアーキテクチャになると思うのですが、各社で違いが出やすいところはどこになるのでしょうか？

- 大企業には何十年と蓄積してきたノウハウがあるから簡単には追いつけないという話がありましたが、Appleが自社でチップを開発し始めて十年強で高性能なものを作れるようになったのはやはりお金や技術者の引き抜きによるところが大きいのでしょうか

- CPUの命令セットが内部でマイクロ命令に分解されるという関係が、CUDAだとユーザーが基本的にいじれるのがPTXまでで、ドライバの中？で真のハードウェアの命令セットに変換されるという関係にしているなあと感じました
- こういう感じで、ハードウェアの(真の)命令セットはRISC的にシンプルなパイプラインがし易いものにして、ユーザーが認識できる命令セットはそこそこ複雑なことができるようにして、ハードウェアの代わりにソフトウェアでハードの真の命令セットに変換するみたいなアプローチはCPUでは取れないのかなと思った

- パイプラインの段数を増やせば早くなるかの議論があったが、講義中ではどこまでも段数を増やせないとあったが、一概に段数が多い方が良いのかが気になった。段数が多いと、ストールした際により多くの実行が無駄になったりしないのかとか気になった。"

- 命令によってパイプラインの段数を動的に変化させることで実際の回路遅延にできる限り近づけるというのはもはや必須の技術だと思うが、次に来る命令の種類によって段数が変化してしまうと、同時に次のステージに到達してしまうハザードが発生するような気がした。この場合はどのように譲り合い(どちらが先に次のステージに入るのか)を制御するのか気になった。

- 最新のCPUの発表では、分岐予測に力を入れたことが多かったと思います。実際の分岐予測ではもっと複雑なアルゴリズムが働いていますか？
- ◇ 講義資料の付録（github の README の最後のほう）に、こみいった予測器の説明をおいてますので、次回の講義を聴いた後にも良ければみてみてください