

# ***Branch Prediction***

Peter Marwedel  
Informatik 12  
TU Dortmund

2014/05/02

# Dynamisches Scheduling: Sprungvorhersage

---

- Bisher betrachtet: Techniken, Datenkonflikte (aufgrund von Abhängigkeiten) zu reduzieren
- Aber: Kontrollflussänderungen sind häufig in realen Programmen (ca. 10-15 Befehle pro Basisblock)
  - ☞ Potentielle Verzögerungen durch Sprünge behandeln
- Je mehr ILP ausgenützt werden soll, desto mehr sind Kontrollkonflikte limitierender Faktor!  
Betrachtete Methoden sind
  - wichtig für *single-issue* Prozessoren (wie bisher, pro Takt ein Befehl ausgegeben)
  - aber essentiell für *multiple-issue* (später genauer) ( $n$  Befehle pro Takt ausgegeben  
=> Sprunghäufigkeit  $n$ -mal so hoch)

# Sprungvorhersage

---

## Methoden zur Sprungvorhersage

### 1. Statisch (d.h. vom Compiler implementiert)

- Defaultannahmen über Sprungverhalten zur Laufzeit
  - *Default not taken* bzw.
  - *Default taken*
- *Delayed branches*

### 2. Dynamisch, d.h. in Abhängigkeit:

- Vom tatsächlichen Verhalten des Sprungs bzw.
- Vom gesamten dynamischen Ausführungskontext des Programms

Was wäre die optimale Methode zur Sprungvorhersage?

Welche Aspekte des Ausführungskontextes können/ sollten betrachtet werden?



*"Vorhersagen sind schwierig, besonders wenn sie die Zukunft betreffen" [Mark Twain]*

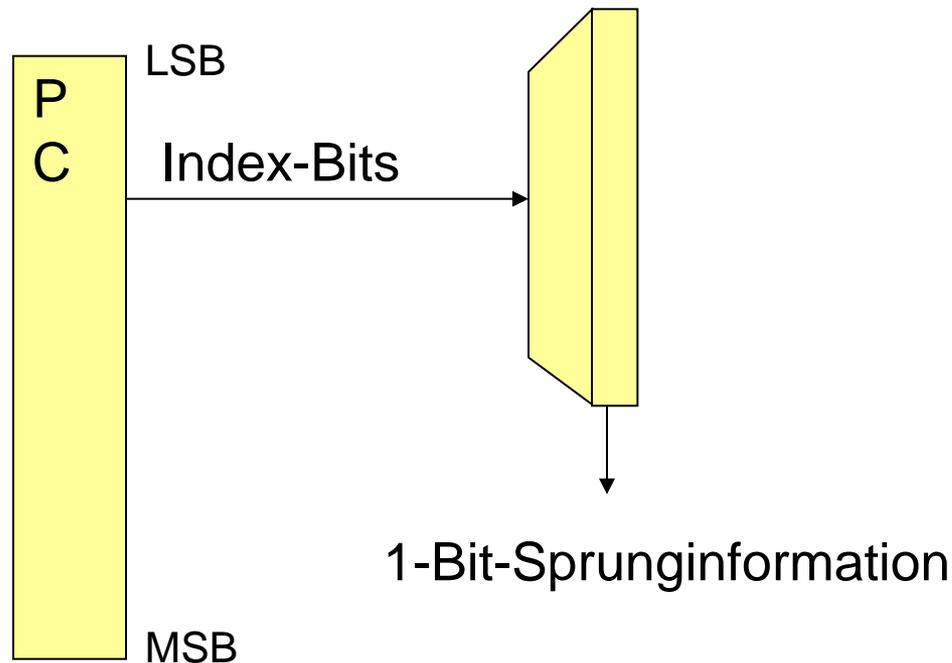
# Einfache Sprungvorhersage

---

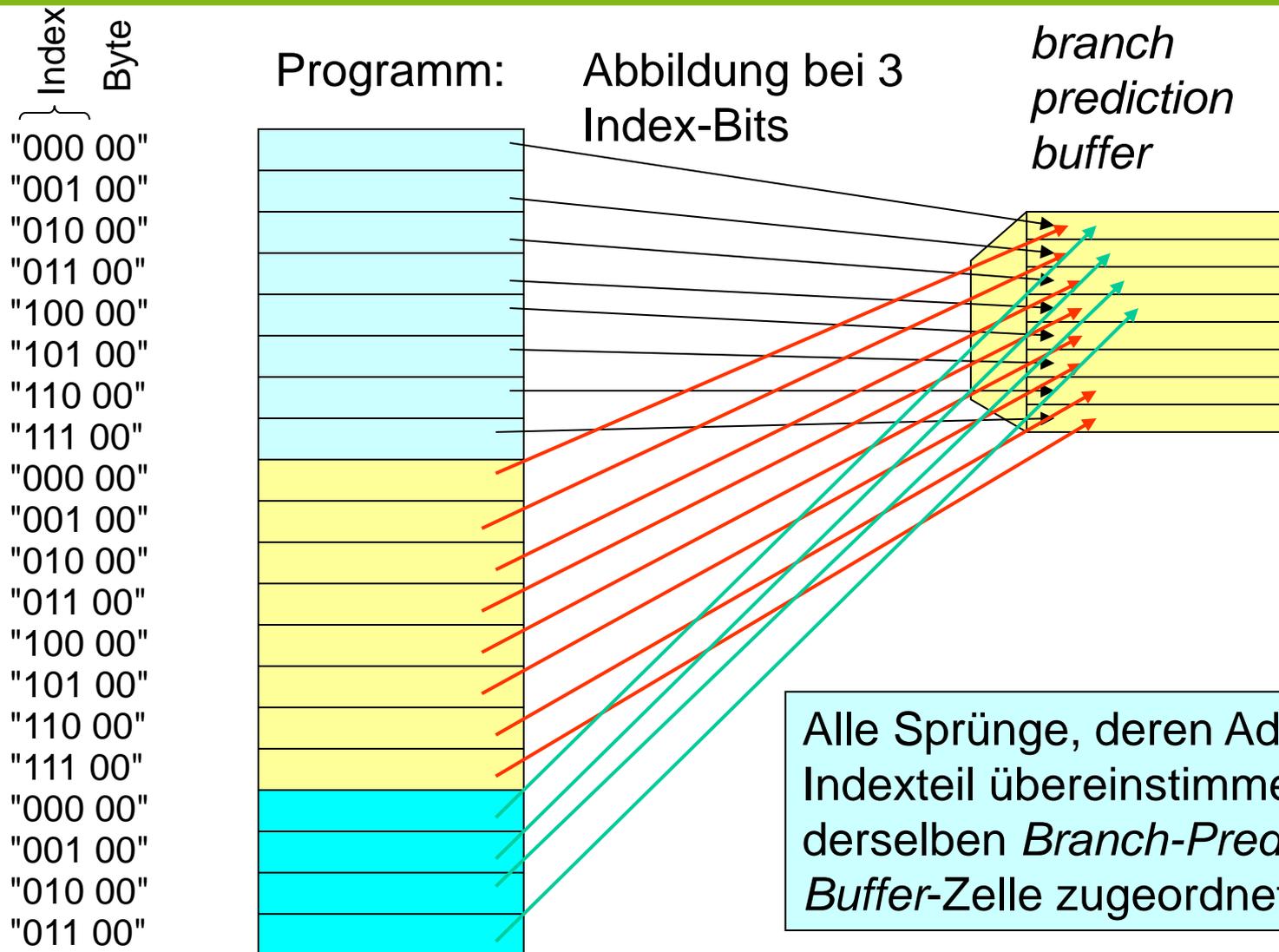
- Sprungvorhersage-Puffer (*branch-prediction buffer*) oder *branch-history table*
- Kleiner Speicher, der mit (Teil der) Adresse des Sprungbefehls indiziert wird
  - Verwendet nur wenige untere Bits der Adresse
  - Enthält 1 Bit: Sprung beim letzten Mal ausgeführt (*taken*) oder nicht (*not taken*)
- Prädiktion: Sprung verhält sich wie beim letzten Mal (alternativ: wie meistens, aber aufwendiger!)
- Nachfolgebefehle ab vorhergesagter Adresse holen
- Falls Prädiktion fehlerhaft: Prädiktionsbit invertieren

# Branch-Prediction-Buffer

Speicherung der Historie, Befehlsadressen als Zugriffsschlüssel:



# Abbildung Programmadressen auf *Branch-Prediction Buffer*-Einträge

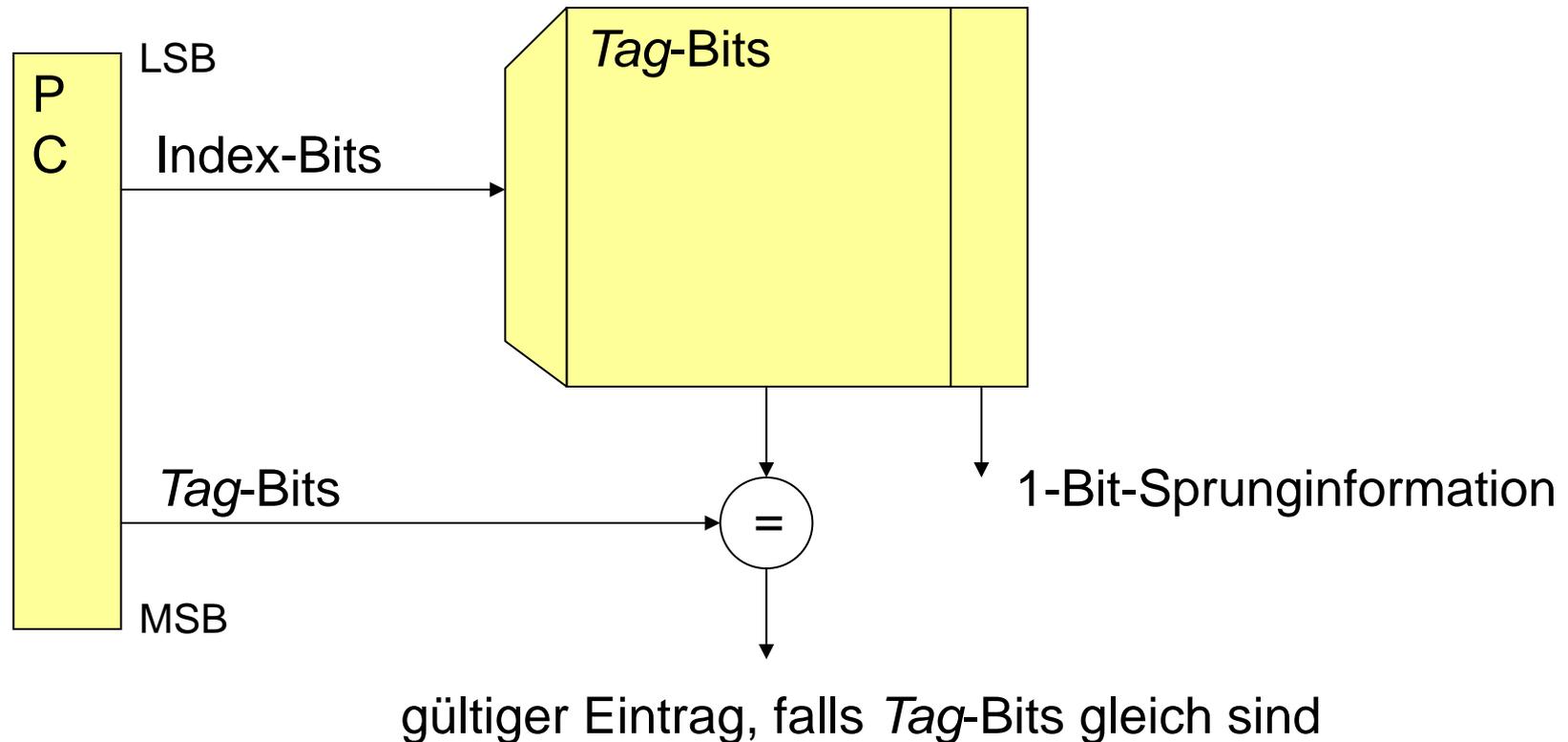


# Eigenschaften der einfachen Sprungvorhersage

---

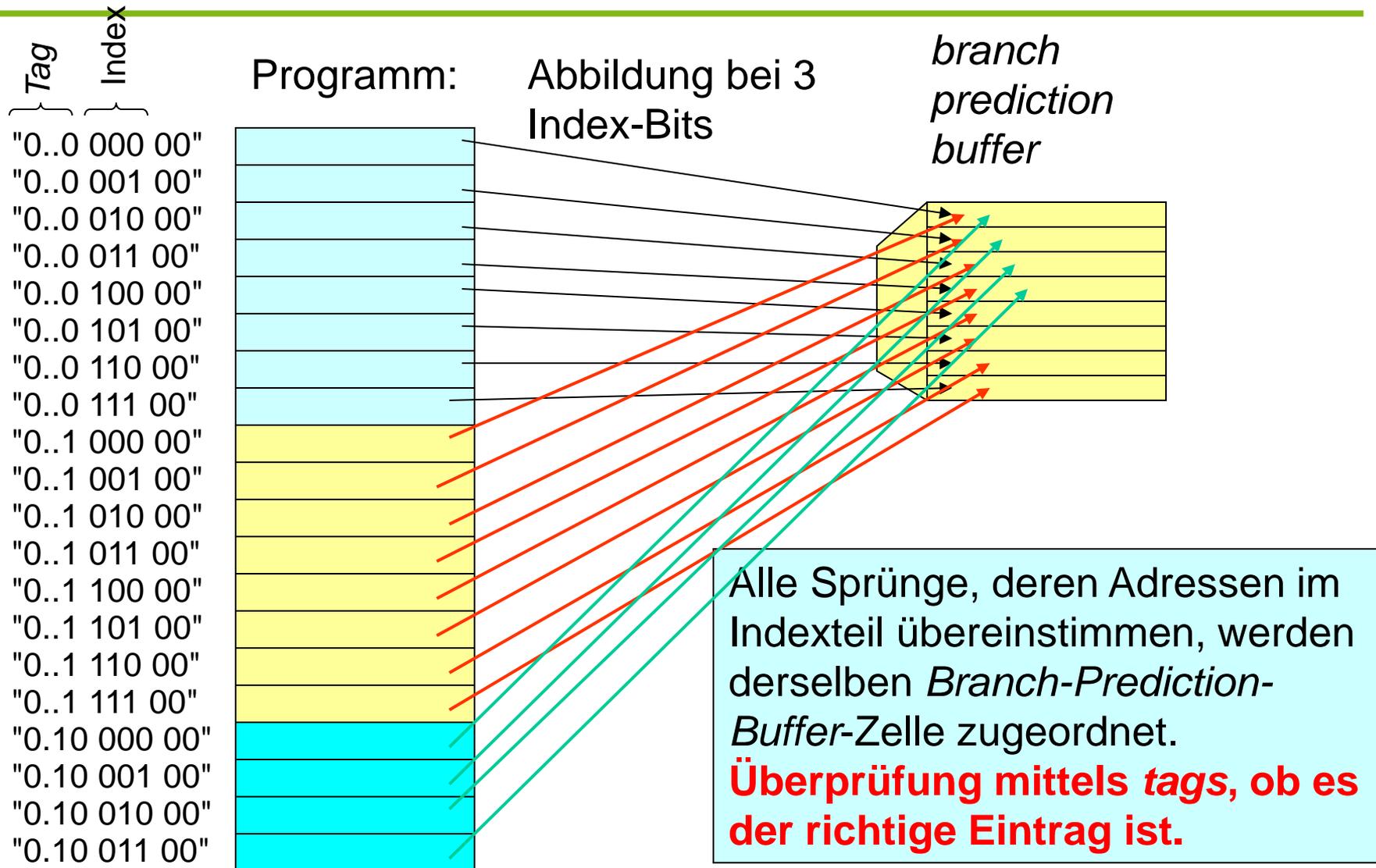
- Einfachste Art von Puffer (keine *Tags*, d.h. keine Überprüfung, ob Adresse tatsächlich im Puffer)
  - Entspricht einfachem Cache
  - Hat eine bestimmte Kapazität
    - ☞ Kann nicht für alle Sprünge (aktuelle) Einträge enthalten
- Reduziert *branch penalty* nur, wenn *branch delay* länger als Berechnung der Zieladresse
- Prädiktion kann fehlerhaft sein
- Prädiktion kann von anderem Sprungbefehl stammen (mit gleichen niederwertigen Adressbits)

# Einführung von *Tag*-Bits



Beseitigt eines der Probleme

# Abbildung Programmadressen auf *Branch-Prediction Buffer*-Einträge



# Eigenschaften der einfachen Sprungvorhersage (2)

---

Nachteile des einfachen 1-bit Vorhersageschemas:

- Hat höhere Fehlerrate als möglich, wenn Häufigkeit der Sprungentscheidungen betrachtet
- D.h. auch wenn Sprung fast immer ausgeführt (*taken*), entstehen 2 Fehler anstatt 1, wenn nicht ausgeführt

Beispiel:

- Betrachten Rücksprung in Schleife  
9x ausgeführt, 1x nicht
- Welche Fehlerrate der Prädiktion ergibt sich, wenn Vorhersagebit im Puffer verbleibt?  
(d.h. wird nicht von anderem Sprung überschrieben)

# Eigenschaften der einfachen Sprungvorhersage (3)

---

Beispiel (cont.):

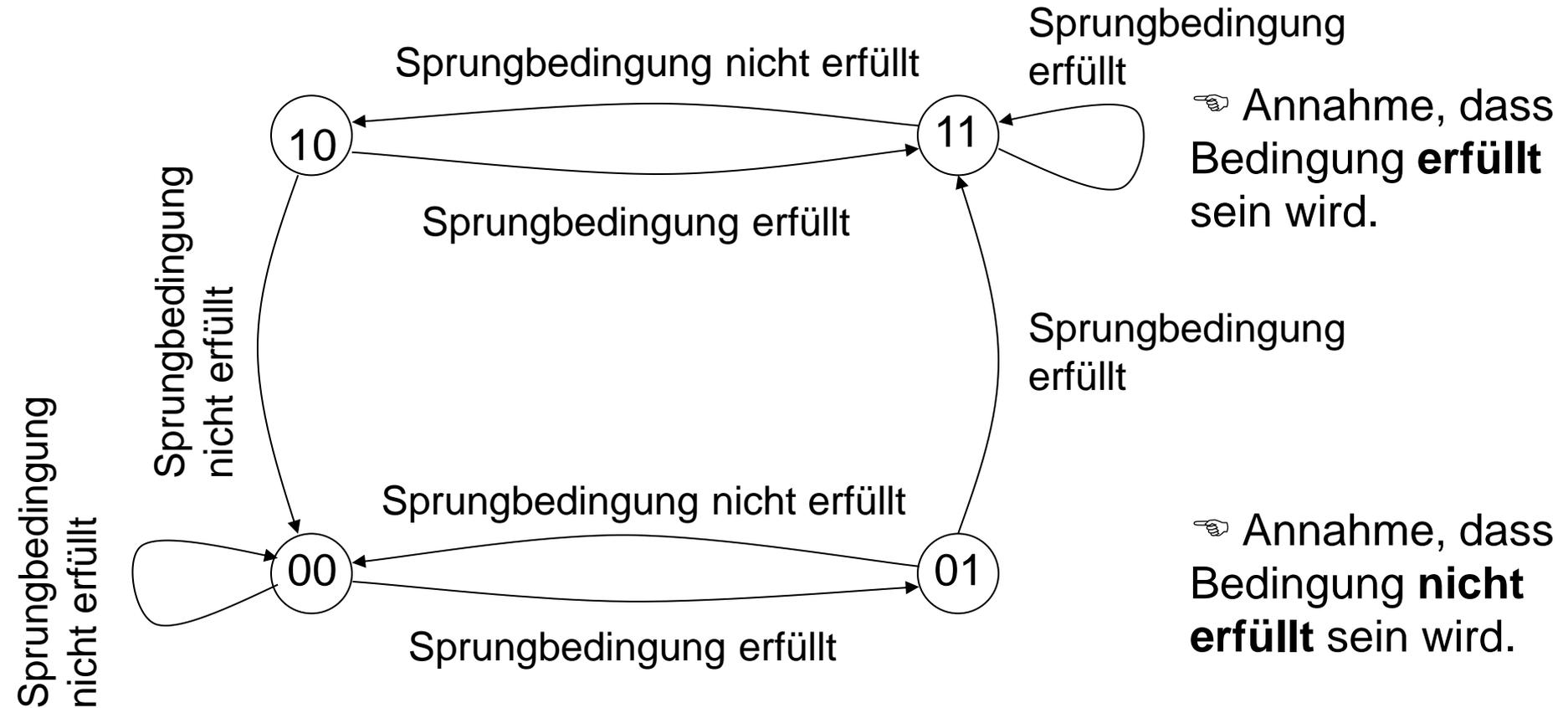
Eingeschwungener Zustand:

- ☞ Vorhersage falsch bei erster und letzter Iteration der Schleife
  - Bei letzter Iteration (praktisch) unausweichlich, da Sprung 9x in Folge ausgeführt
  - Bei erster Iteration wegen Umschaltung nach Verlassen der Schleife (aufgrund nur einer Observation!)
- ☞ Vorhersagegenauigkeit für in 90% der Fälle ausgeführten Sprung ist 80%

Allgemein: Fehlerrate von 1-bit Prädiktor ist für Sprünge in Schleifenkonstrukten doppelt so hoch, wie Anzahl nicht ausgeführter Sprünge ☞ unschön!

# Verbesserung: 2-Bit-Vorhersage

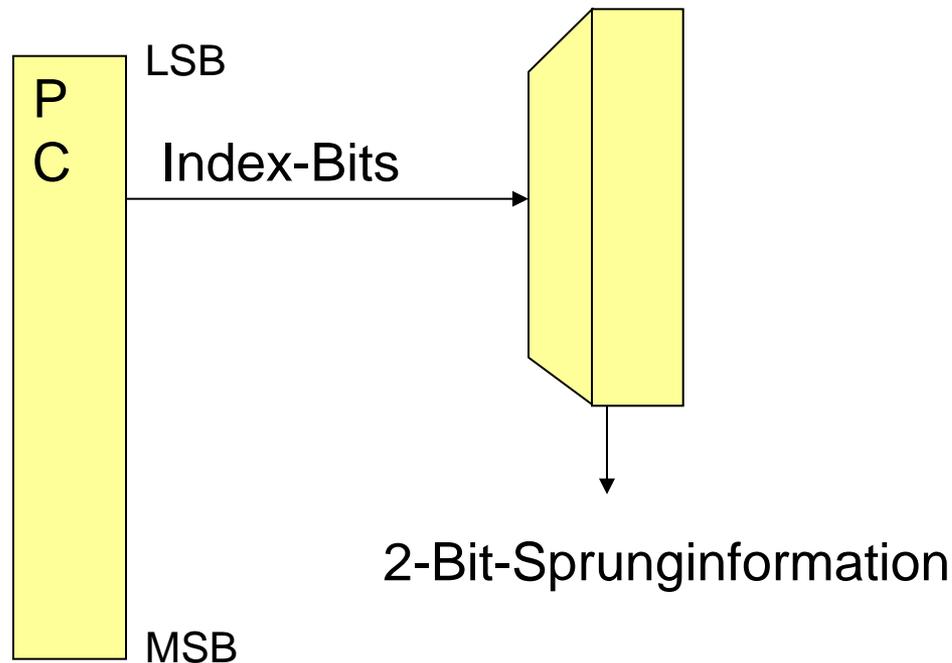
Änderung der Vorhersage nur, wenn 2 falsche Vorhersagen in Folge



2-Bit-Schieberegister, Einschieben aktueller Bedingung

# Branch-Prediction-Buffer

Speicherung der Historie, Befehlsadressen als Zugriffsschlüssel:



# $n$ -bit Prädiktor

---

Allgemein:  $n$ -bit Prädiktor (Spezialfall: 2-bit)

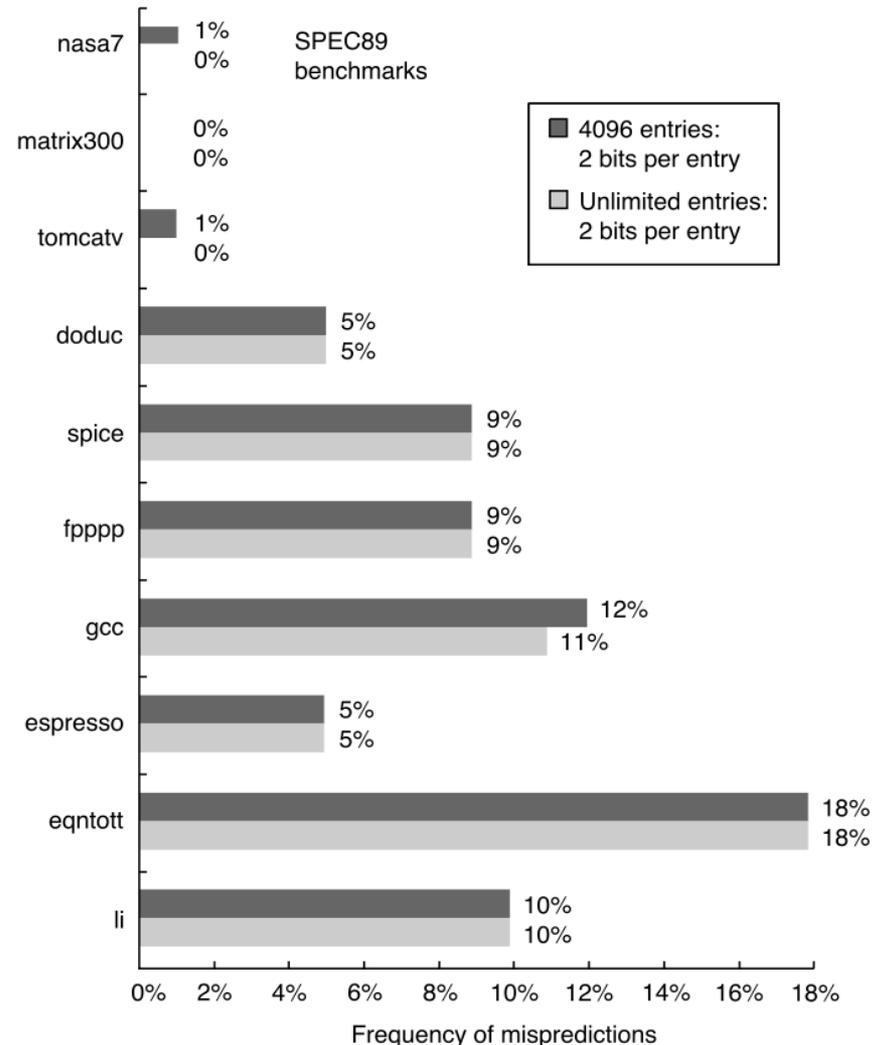
Verwendet  $n$ -bit Zähler

- Sättigungsarithmetik (kein wrap-around bei Überlauf)
- Kann Werte zwischen 0 und  $2^n - 1$  annehmen
- Wenn Zähler größer als Hälfte des Maximums ( $2^{n-1}$ ): Vorhersagen, dass Sprung ausgeführt wird
- Zähler wird bei ausgeführtem Sprung inkrementiert und bei nicht ausgeführtem dekrementiert

In der Praxis: 2-bit Prädiktor ähnlich gut wie  $n$ -bit Prädiktor  
☞ in den meisten Prozessoren heute: 2-bit Prädiktor für (lokale) Vorhersage

# Vorhersagequalität für 2-bit Prädiktor

- Studie von '92 für SPEC89 auf IBM Power-Architektur
- Qualität nicht durch die Größe des Speichers beschränkt
- Fehlerwahrscheinlichkeit höher für Integer-Programme
- Für Auswirkungen muss auch Sprungfrequenz betrachtet werden



# Korrelierende Prädiktoren

---

- Einschränkung des  $n$ -bit (bzw. 2-bit) Prädiktors:  
Betrachtet nur (vergangenes) Verhalten eines Sprungs, um dessen (zukünftiges) Verhalten vorherzusagen.  
☞ arbeitet rein lokal!
- Idee: Verbesserung durch Betrachtung des Verhaltens anderer Sprünge  
☞ man erhält so genannten korrelierenden Prädiktor (*correlating predictor*) oder zweistufigen Prädiktor
- Prinzip: Aufgrund globaler Information (andere Sprünge) wird einer von mehreren lokalen Prädiktoren ausgewählt

# Korrelierende Prädiktoren (2)

```
if (aa == 2)
aa = 0;
if (bb == 2)
bb = 0;
if (aa != bb) {
... }
```

Beispiel (aus Spec92 eqntott)

MIPS-Code: (aa in R1, bb in R2)

```
          DSUBI    R3, R1, 2      # aa =? 2
b1:       BNEZ    R3, L1         # Branch b1 (aa != 2)
          DADD     R1, R0, R0     # aa = 0
L1:       DSUBI    R3, R2, 2      # bb =? 2
b2:       BNEZ    R3, L2         # Branch b2 (bb != 2)
          DADD     R2, R0, R0     # bb = 0
L2:       DSUB    R3, R1, R2     # R3 = aa-bb
b3:       BEQZ    R3, L3         # Branch b3
```

Falls b1 und b2 nicht ausgeführt folgt hier: aa==bb und b3 wird immer(!) ausgeführt

☞ mit lokaler Prädiktion nicht vorherzusagen

# Korrelierende Prädiktoren (3)

```

if (d == 0)
d = 1;
if (d == 1)
...;
    
```

Arbeitsweise des zweistufigen Prädiktors (illustratives Beispiel):

- MIPS-Code:

```

b1:   BNEZ      R1,L1      # Branch b1 (d != 0)
      DADDI    R1,R0,1    # d == 0 => d = 1
L1:   DADDI    R3,R1,-1
b2:   BNEZ      R3,L2      # Branch b2 (d != 1)
      ...
L2:
    
```

- Ausführungsfolgen (d = 0,1,2):

Initiales d	d==0?	b1	d vor b2	d==1?	b2
0	ja	nein	1	ja	nein
1	nein	ja	1	ja	nein
2	nein	ja	2	nein	ja

- Falls b1 *not taken* ➡ b2 *not taken*

# Korrelierende Prädiktoren (4)

- Betrachten wiederholte Ausführung des Codefragments (ignorieren dabei alle anderen Sprünge inkl. dem für Wiederholung)
- Einschränkung: Statt aller möglichen Sequenzen:  $d$  wechselt zwischen 2 und 0

Verhalten des 1-bit Prädiktors (initialisiert zu *not taken*)

$d=?$	Präd.	Akt. Neue Präd.	Präd.	Akt. Neue Präd.
	$b1$	$b1$	$b2$	$b2$
2	NT	T	NT	T
0	T	NT	T	NT
2	NT	T	NT	T
0	T	NT	T	NT

Alle Vorhersagen falsch!

# Zweistufiger Prädiktor

---

Verwendet 1 Bit Kontextinformation

Ex. 2 lokale Prädiktoren (je 1-bit)

Kontext: Letzter (i.a. anderer) Sprung wurde ausgeführt/nicht ausgeführt (1 Bit)

Vorhersage aufgrund des Kontexts wählt lokalen Prädiktor für Sprungvorhersage aus

Letzter Sprung ist i.a. nicht gleich aktuellem, vorherzusagendem Sprung (nur in einfachen Schleifen)

Notation des Prädiktorstatus:  $\langle X \rangle / \langle Y \rangle$  mit

- $\langle X \rangle$ : Vorhersage, falls letzter Sprung *not taken* (NT)
- $\langle Y \rangle$ : Vorhersage, falls letzter Sprung *taken* (T)
- $\langle X \rangle$  und  $\langle Y \rangle$  Vorhersagen: T oder NT

# Verhalten des 2-stufigen Prädiktors

Beispiel: (1,1)-Prädiktor

- 1 Bit Kontext, d.h. nur vorheriger Sprung betrachtet
- 1 Bit lokale Vorhersage
- Initialisiert zu NT/NT

<i>d=?</i>	<i>Präd.</i>	<i>Akt.</i>	<i>Neue Präd.</i>	<i>Präd.</i>	<i>Akt.</i>	<i>Neue Präd.</i>
	<i>b1</i>	<i>b1</i>	<i>b1</i>	<i>b2</i>	<i>b2</i>	<i>b2</i>
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

Notation: **Verwendete Vorhersage**, **falsche Vorhersage**

# $(m,n)$ -Prädiktor

---

Allgemein:  $(m,n)$ -Prädiktor

- Betrachtet Verhalten der letzten  $m$  Sprünge um aus  $2^m$  lokalen Prädiktoren einen  $n$ -bit Prädiktor auszuwählen

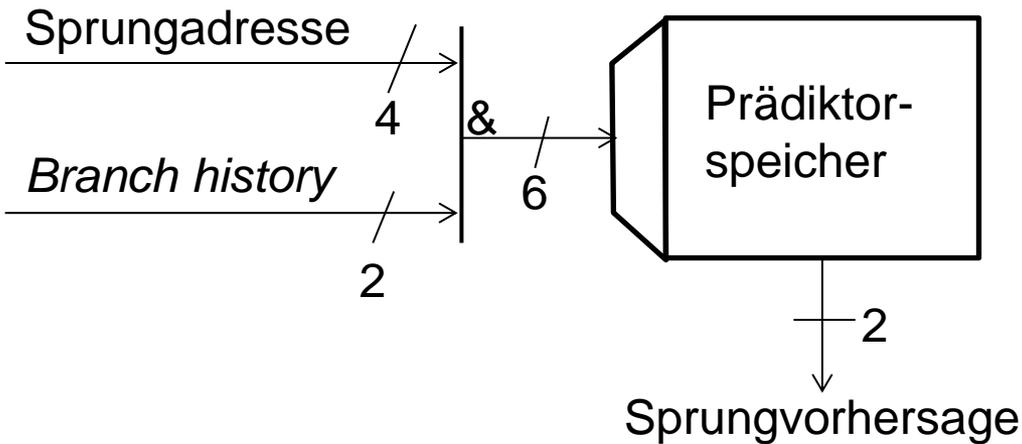
Vorteil gegenüber (rein lokalem) 2-bit Prädiktor

- Höhere Vorhersagegenauigkeit
- Erfordert kaum Hardwareaufwand
  - Sprunggeschichte („Ausgang“ vorangegangener Sprünge) kann in Schieberegister gespeichert werden (1 Bit pro Sprung, 1 wenn *taken*)
- Vorhersagepuffer adressiert via Konkatination von:
  - Unteren Adressbits der Sprungbefehlsadresse
  - $m$  Bit globaler Sprung-Geschichte

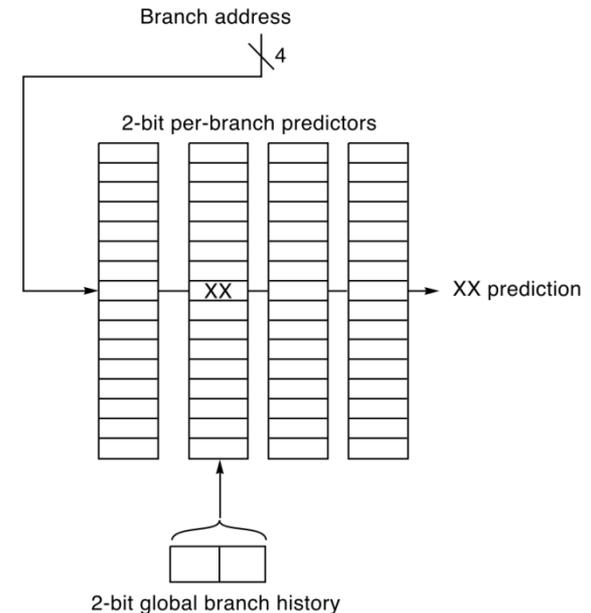
# Beispiel: (2,2)-Prädiktor

- 64 Einträge insgesamt, 4 je lokalem Prädiktor

Unsere Notation



Hennessy/Patterson



# Vergleich von Vorhersagemethoden

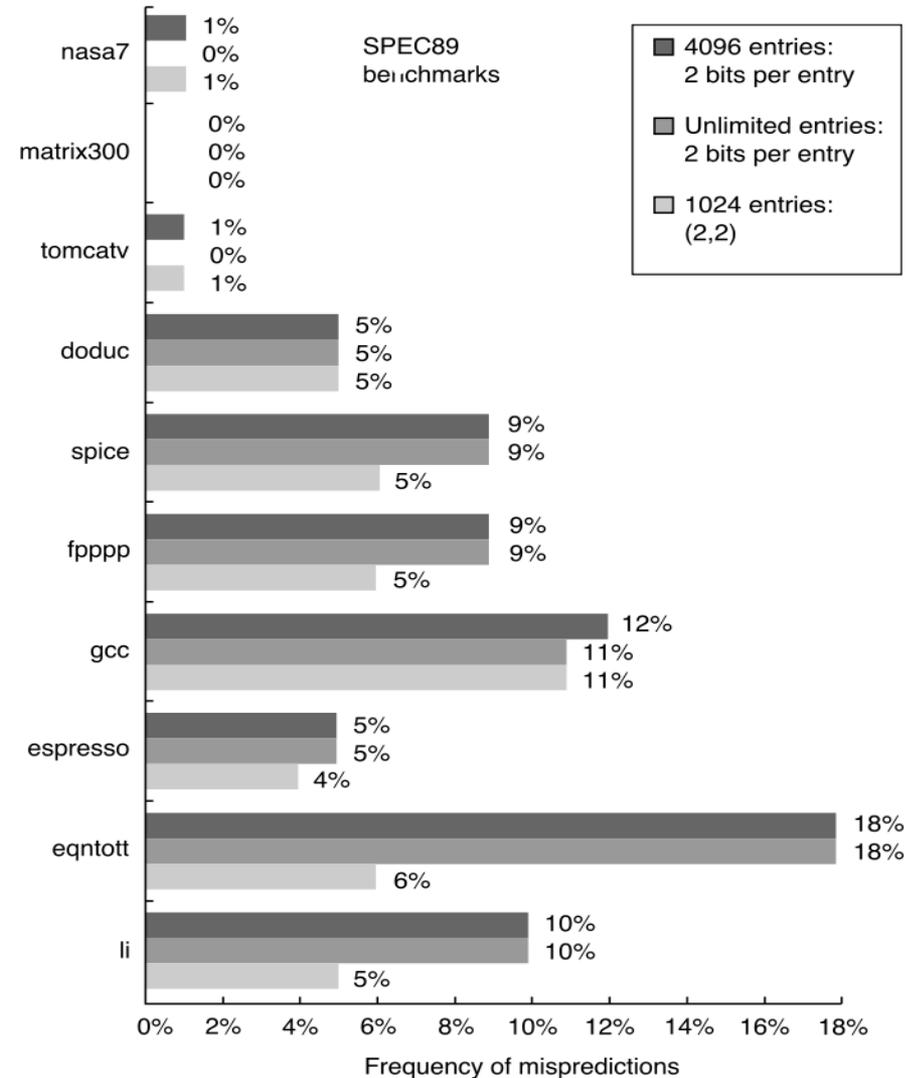
---

Nur fair, wenn gleiche Anzahl Zustandsbits verwendet!

- Anzahl Zustandsbits in  $(m, n)$  Prädiktor:  
 $2^m \times n \times \text{\#Einträge in lokaler Vorhersagetabelle}$
- Lokaler 2-bit Prädiktor ohne Sprung-Geschichte entspricht (0,2)-Prädiktor
  - ☞ 2-bit Prädiktor mit 4096 Einträgen (= 4K) enthält  $2 \times 4K = 8K$  Zustandsbits
- Vergleich mit (2,2) Prädiktor gleicher Größe:
  - ☞  $8k = 4 \times 2 \times \text{\#Einträge}$
  - ☞  $\text{\#Einträge} = 8K/8 = 1K$

# Vergleich (SPEC89)

- Einfacher 2-bit Prädiktor (4K Einträge)
- 2-bit Prädiktor mit unbegrenzter Anz. Einträge (☞ eindeutig)
- (2,2)-Prädiktor (je 1K Einträge)



# Tournament Prädiktor

---

Motivation für 2-stufige Prädiktoren:  
lokale Information häufig nicht ausreichend

☞ globale Information dazunehmen

Aber: Kombinationsschema fest!

Verallgemeinerung:

Verwendung mehrerer Prädiktoren und

Selektion zwischen diesen

☞ *Tournament* Prädiktor

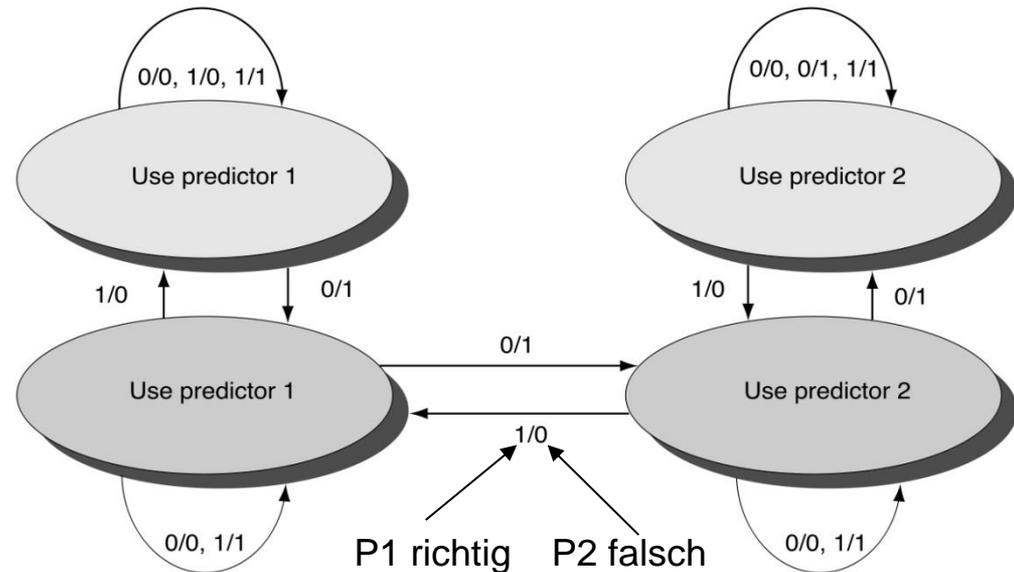
- i.d.R. Kombination von lokalem und globalem Prädiktor
- Verbreitetste Form mehrstufiger Prädiktoren

# Tournament Prädiktor (2)

Beispielkonfiguration:

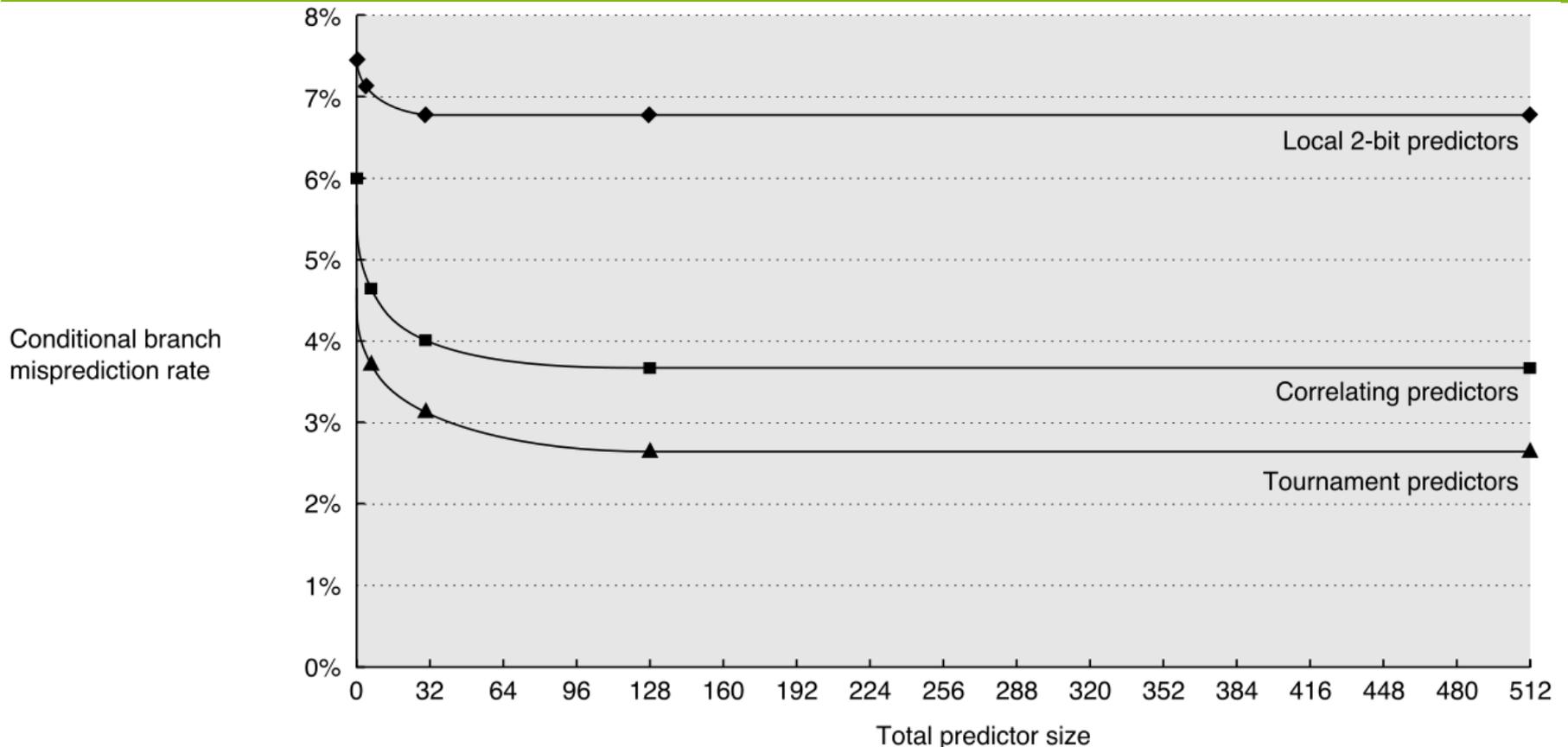
- 2-bit saturierender Zähler
- Inkrementiert, falls verwendeter Prädiktor korrekt und anderer inkorrekt vorhersagt
- dekrementiert im inversen Fall
- Sonst: unverändert

Beachte: Änderung nur, falls ein Prädiktor klar im Vorteil!



© 2003 Elsevier Science

# Sprungvorhersage: Vergleich verschiedener Prädiktoren



- Deutlicher Vorteil durch 2-stufige Prädiktion
- Weiterer leichter Vorteil durch *Tournament*-Prädiktor

© 2003 Elsevier Science

# Dekodierung mit Hochdurchsatz

---

In Hochleistungspipelines ist reine Vorhersage eines Sprungs i.d.R. nicht ausreichend

Insbesondere: Falls mehrere Befehle pro Takt auszugeben

☞ Befehlsstrom mit großer Bandbreite erforderlich

Idee: Kontrollflussabhängigkeiten dürfen nicht „wahrnehmbar“ sein

Maßnahmen dazu:

- Pufferung von Sprungzielen und nicht nur Vorhersage des Sprungverhaltens (*branch-target buffer*)
- Integrierte Einheit für das Holen der Befehle (d.h. nicht nur [relativ] einfache erste Stufe der *Pipeline*)
- Vorhersage von Rücksprungadressen (bei Prozeduraufruf)

# Branch Target Buffer

---

Betrachten 5-stufige Pipeline mit Sprungbedingungs-  
auswertung in EX  *branch delay* von 2 Takten

Mit Sprungvorhersage (*branch-prediction buffer*):

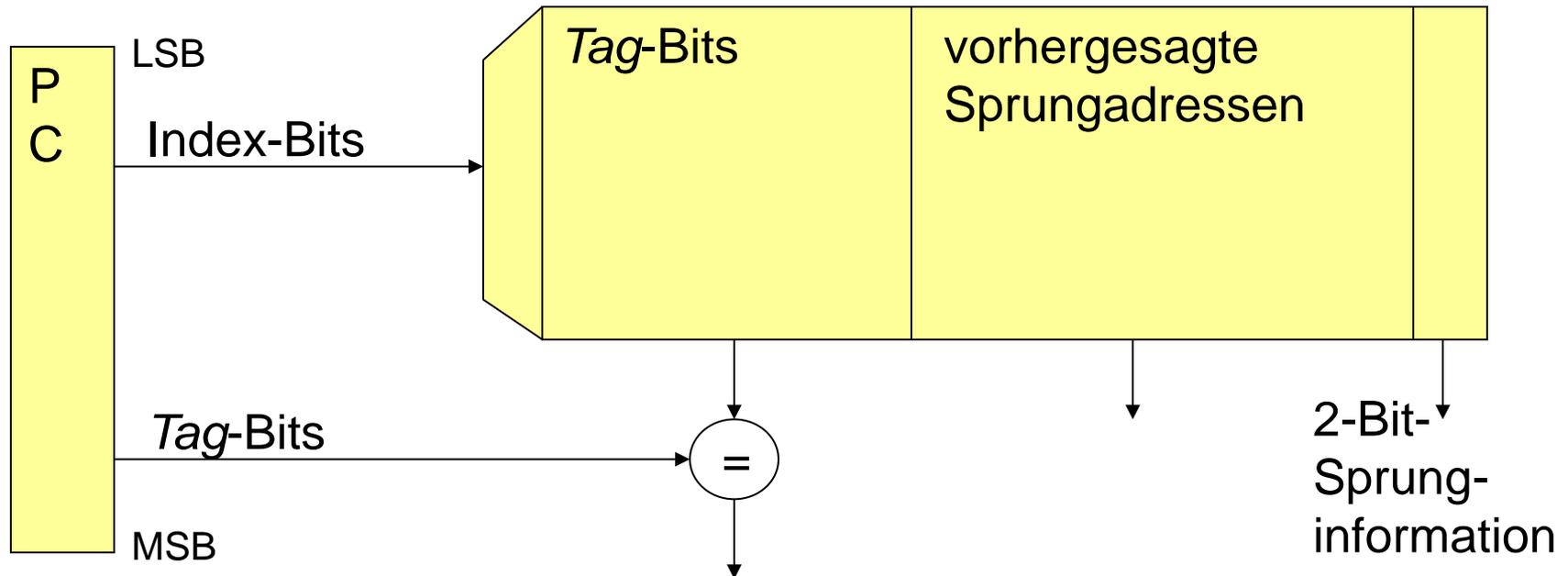
- Zugriff erfolgt in ID (Adresse des Sprungbefehls schon in IF bekannt, aber, ob Sprung, erst nach Befehlsdekodierung [ID])
- Nächste vorhergesagte Instruktion kann nach ID geholt werden => *branch delay* = 1, fall Prädiktion korrekt

Mit Pufferung des Sprungziels (*branch-target buffer*):

- Zugriff erfolgt in IF
- Verhalten wie „echter“ Cache, adressiert mit Sprungbefehlsadresse (Wichtig: Überprüft, ob Cache-Hit)
- Liefert vorhergesagte Adresse als Ergebnis, d.h. nächsten PC (d.h. nicht nur Vorhersage über Sprungverhalten)  
 keine Verzögerung, falls Prädiktion korrekt!

# Branch-Target-Buffer (2)

Zusätzliche Speicherung auch des Sprungziels, z.B. Kombination mit *branch prediction buffer*:



gültiger Eintrag, falls *Tag-Bits* gleich sind

Bei geschickter Organisation kann das Fließband immer gefüllt bleiben; die Sprünge kosten dann effektiv keine Zeit;  $\text{CPI} < 1$  möglich.

# Branch-Target-Buffer (3)

---

Eigenschaften:

- Verzögerung durch Sprung kann vollständig vermieden werden (sofern Vorhersage korrekt), da bereits in IF Entscheidung über nächsten Befehlszähler (PC)
- Da Entscheidung allein auf Basis der Befehlsadresse, muss überprüft werden, ob Adresse im Puffer (beachte: impliziert, dass Sprungbefehl vorliegt)
- Speicherung im Prinzip nur für Sprünge notwendig, die als ausgeführt vorhergesagt werden (*not taken* = normale sequentielle Dekodierung geht weiter)

Achtung: Bei falscher Vorhersage:

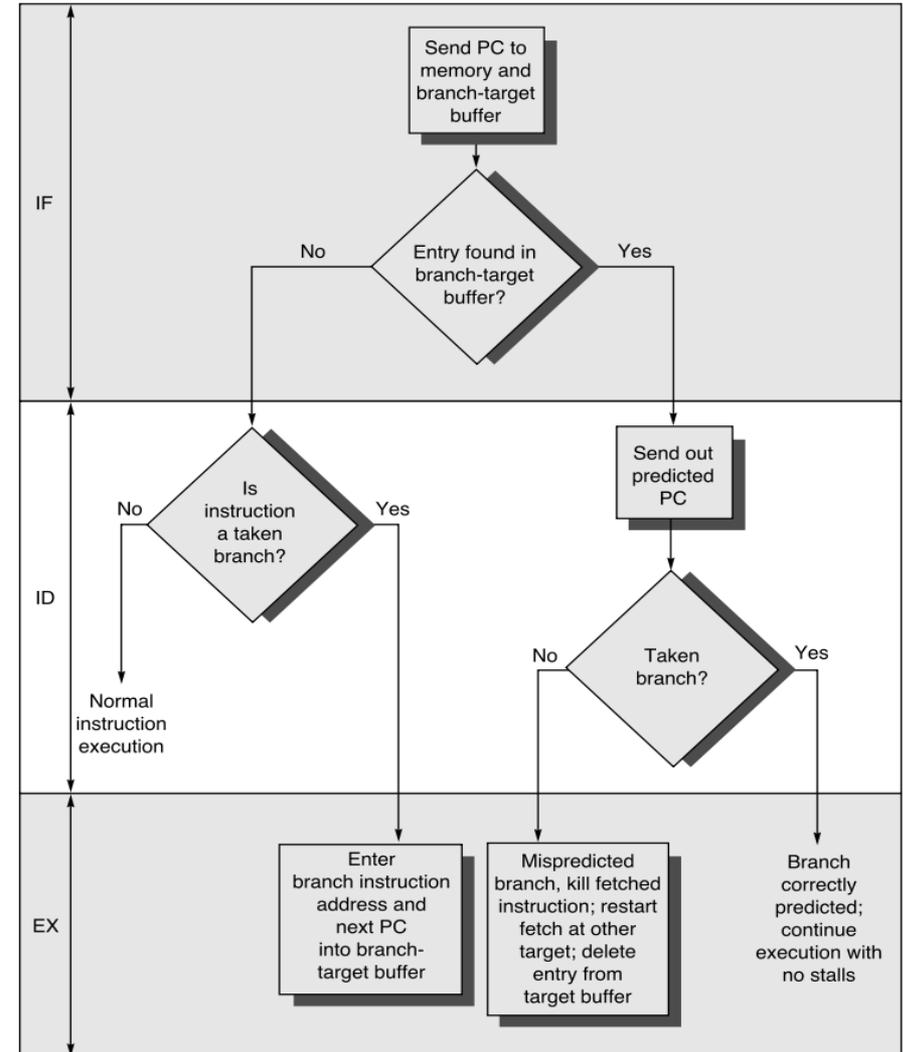
- Entsteht ursprüngliche Sprung-Verzögerung und
- Aufwand für Aktualisierung des Vorhersagepuffers

# Schritte in der Pipeline

- Rechter Zweig: Sprungbefehl bereits im Puffer
- Linker Zweig: Evtl. neuen Sprungbefehl eintragen

Hier Annahme:

- Adressrechnung und
- Bedingungsauswertung erfolgen in ID



© 2003 Elsevier Science

# Erweiterung

---

- Speicherung einer oder mehrerer Instruktionen ab dem Sprungziel zusätzlich zur vorhergesagten Adresse

Vorteile:

- Zugriff auf Puffer der Sprungzieladressen kann mehr Zeit in Anspruch nehmen, als Intervall zwischen zwei IF-Phasen
  - ☞ komplexerer / größerer Puffer möglich
- Erlaubt *branch folding*:
  - Bei unbedingtem Sprung kann Sprungbefehl nach Zugriff auf *branch-target buffer* direkt durch gehaltenen Befehl in der Pipeline ersetzt werden
    - ☞ *zero-cycle unconditional branch*
  - Ähnlich auch für bedingte Sprünge möglich, aber komplexer (wegen Bedingungsauswertung)

# Integrierte Befehls-Hol-Einheit (IF-Unit)

---

Insbesondere mit Blick auf *multiple-issue* Prozessoren eigene (autonome) funktionelle Einheit für Befehlsholphase

☞ führt Befehlscodes in *Pipeline* ein

Integrierte Funktionalitäten:

- Sprungvorhersage: Wird Teil der Befehlsholphase
- *Instruction Prefetch*: Insbes. um mehrere Befehle pro Takt liefern (und später ausgeben) zu können, geht Befehlsholen weiterer Dekodierung voraus (= *prefetch*)
- Zugriff auf Befehlsspeicher: Bei mehreren Befehlen pro Takt mehrere Zugriffe (bei Cache auf ggf. mehrere *cache lines*) erforderlich -> werden hier koordiniert/geplant
- Befehlspuffer: Befehle können hier (lokal im Prozessor!) von *Issue*-Stufe nach Bedarf abgerufen werden

# Vorhersage von Rücksprungadressen

---

Allgemeines Ziel: Vorhersage indirekter Sprünge  
(d.h. bzgl. Basisadresse in Register)

Hauptverwendung: Rückkehr aus Prozeduraufrufen  
(Bei MIPS: Aufruf JAL proc, Rückkehr JR \$31 o.Ä)

Vorhersage mit *Branch-Target* Buffer schlecht, da Aufruf aus unterschiedlichen Codeteilen möglich

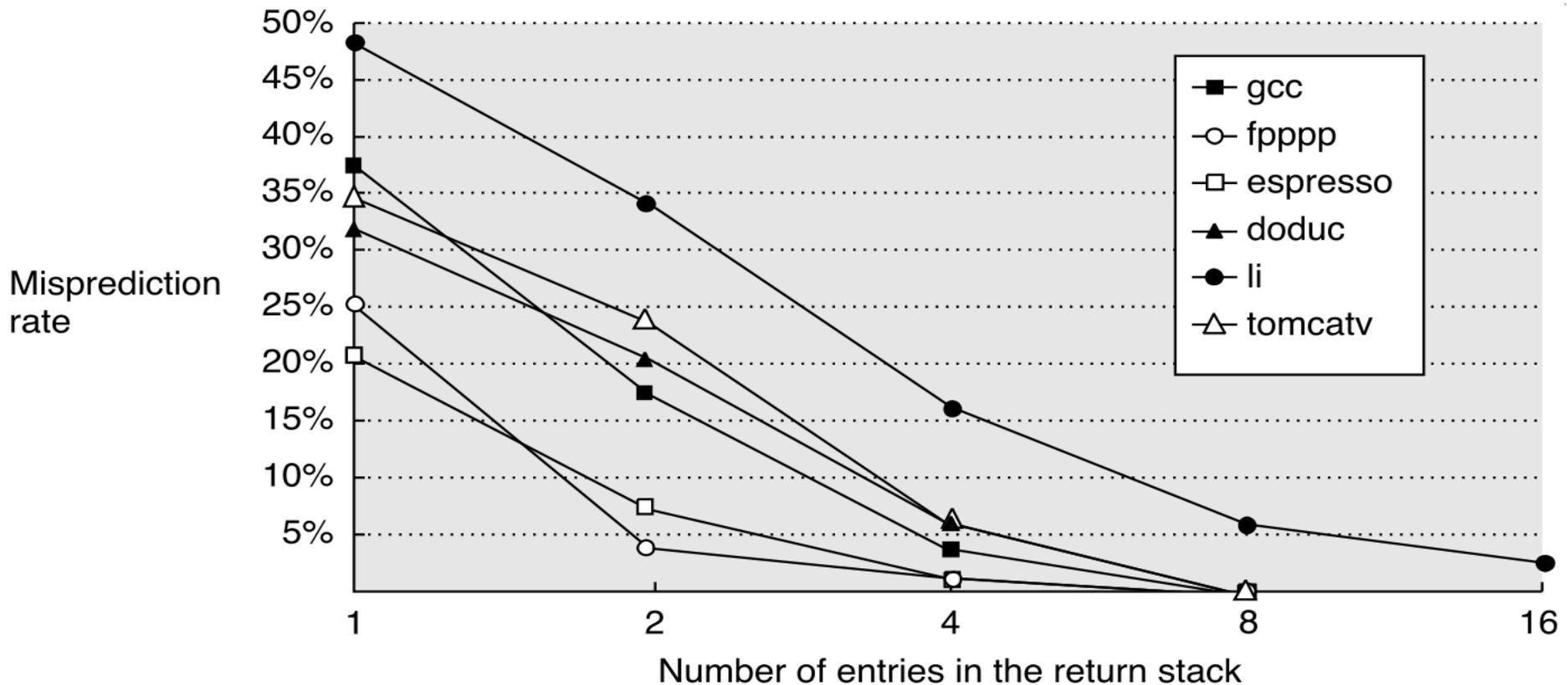
☞ Methode: (*Stack*-)Speicher für Rücksprungadressen

- Push, bei Prozeduraufruf (*call*) und
- Pop bei "*Return*"

Vorhersagequalität „perfekt“, wenn Puffer größer als max. Aufruftiefe

# Vorhersage von Rücksprungadressen (2)

Vorhersagequalität für Stack-Tiefe 1-16 auf verschiedenen SPEC-Benchmarks



© 2003 Elsevier Science

# Zusammenfassung

---

## Sprung-Vorhersage

- Einfache lokale Prädiktoren
  - 1-Bit Prädiktor
  - 2-Bit Prädiktor
  - $n$ -bit-Prädiktor
- Korrelierende Prädiktoren
- *Tournament-Prädiktoren*
- *Branch Target Buffer*
- *Return Address Buffer*