Rechnerarchitektur

Marián Vajteršic und Helmut A. Mayer

Fachbereich Computerwissenschaften
Universität Salzburg
marian@cosy.sbg.ac.at und helmut@cosy.sbg.ac.at
Tel.: 8044-6344 und 8044-6315

28. September 2017

Pipelining

Pipelining

Pipelining ist eine Implementierungsmethode, bei der mehrere Befehle überlappt abgearbeitet werden.

Pipelining: Schlüsselimplementierungsmethode, um schnelle Prozessoren zu realisieren.

Pipelining

Überlappung:

- Autofließband: Zur gleichen Zeit befinden sich mehrere Autos in verschiedenen Phasen des Erzeugungsprozesses.
- - Addition von zwei Registern
  - Laden einer Speicherstelle in ein Register

  Beide Aktionen sind voneinander unabhängig:
  ALU addiert (1.) und CU macht Speicherzugriff (2.).
  Also überlappend (in der Zeit) ausführbar, d. h. gleichzeitig $\rightarrow$ Verbesserung der Rechnergeschwindigkeit

Wenn wir den Befehl in mehrere (verschiedene) Arbeitsschritte unterteilen können, dann können wir an mehreren Befehlen gleichzeitig arbeiten.

Pipelining

```
ADD R1, R2, R3
ADD R4, R5, R6
```
: geht gleichzeitig

```
ADD R1, R2, R3
ADD R4, R5, R1
```
: geht wegen Abhängigkeit (*R1*) nicht

Heutige Compiler versuchen herauszufinden, ob **parallele** Abarbeitung möglich ist

Parallele Abarbeitung: **Parallel**rechner SIMD, MIMD
Single Instruction Multiple Data
Multiple Instruction Multiple Data

Pipeline: Quasi-parallel

Befehle werden in mehreren Stufen bearbeitet

(Pipeline-)Stufe: Arbeitsschritt

T': Befehlsausführungszeit auf der Pipeline-Maschine

$$T'_{\text{Ideal}} = \frac{T_0}{n_p} = \frac{\text{Befehlsausführungszeit der ohne Pipeline implementierten Maschine}}{\text{Zahl der Pipeline-Stufen}}$$

$$\rightarrow \text{Speedup} = \frac{T_0}{T'}$$

$$\text{Speedup-Ideal} = \frac{T_0}{T'_{\text{Ideal}}} = \frac{T_0}{\frac{T_0}{n_p}} = n_p$$

Idealer Speedup nicht möglich: Obere Schranke (nicht alle Arbeitsschritte für alle möglichen Befehle sind voneinander unabhängig).
Wenn es gelingt, jeden Arbeitsschritt in einem Taktzyklus zu absolvieren, dann erzielt man die optimale CPI-Rate von 1.

Die DLX-Pipeline

Die Basis-Pipeline für die DLX-Architektur

Die Ausführung von DLX-Befehlen kann in **fünf** Grundschritten (Arbeitsschritten) implementiert werden:

1. Holen eines Befehls
   `IF` (**I**nstruction **F**etch)
   Die Instruktion wird aus der Speicherstelle, auf die der Programmcounter *PC* zeigt, in das Instruktion Register *IR* geholt und der neue Programmcounter *NPC* sowie auch der Programmcounter *PC* wird auf die Adresse des nächsten Befehls ($PC + 4$) gesetzt.
   (*PC*, *NPC* sind spezielle DLX-Register).
   Wir betrachten Befehlscode fester Länge, so entsteht *NPC* durch eine fixe Aufstockung von *PC* ($+4$).

Die Basis-Pipeline für die DLX-Architektur (Fortsetzung)

2. Befehlsdecodierung und Register holen
   `ID` (**I**nstruction **D**ecode und Register Fetch)
   Inhalte der Register und des Immediate werden in Prozessor geholt.
   Aus *IR* werden die Adressen der Register und ein (möglicher) Immediate-Wert extrahiert. Drei Wortregister (*A*, *B*, *IMM*) werden benötigt, um Operanden (Registerwerte und Immediate) des Befehls zwischenzuspeichern.
3. Ausführung und Adressberechnung
   `EX` (**EX**ecution and effective address calculation)
   In dieser Stufe wird der Befehl ausgeführt in Abhängigkeit von der jeweiligen Instruktion (dekodiert in der vorigen Stufe).

Die Basis-Pipeline für die DLX-Architektur (Fortsetzung)

4. Speicherzugriff
   `MEM` (**MEM**ory access)
   Der gewünschte Speicherzugriff wird hier durchgeführt.
5. Rückschreiben zum Zielregister
   `WB` (**W**rite **B**ack)
   Ein Datum aus dem Speicher (das im 4. Schritt geholt wurde), wird in das Zielregister transferiert (`LOAD`-Befehl), beziehungsweise das Ergebnis der ALU-Berechnung in das gewünschte Register eingetragen.

Die Basis-Pipeline für die DLX-Architektur (Fortsetzung)

Wir werden diese 5-stufige Bearbeitungssequenz für eine vereinfachte Version der DLX-Pipeline präzisieren.

Dabei beschränken wir uns auf:

- **Load-/Store-Befehle**
- **Integer-ALU-Operationen**
- **Branches**

Für die Beschreibung von Arbeitsschritten wird ein Pseudocode verwendet (J. L. Hennessy, D. A. Patterson: Rechnerarchitektur. Vieweg Verlag, 1996).

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- `LOAD`-Befehl
  - Befehl vom Typ I
  - Befehlsformat

Opcode	RS	RD	IMM
0 . . . 5	6 . . . 10	11 . . . 15	16 . . . 31

  - Beispiel: `LW R5, 100(R1)`
    $\rightarrow$ Opcode = `LOAD WORD`; $RS = R1$; $RD = R5$; $IMM = 100$
  - Die Bearbeitungsstufen:

`IF`: $IR \leftarrow MEM[PC]$
$NPC \leftarrow PC + 4$
$PC \leftarrow NPC$

`ID`: $A \leftarrow REGS[IR[6 \ldots 10]]$
$IMM \leftarrow IR[16 \ldots 31]$
(Inhalt des Source-Registers mit der Adresse $IR[6 \ldots 10]$, d. h. $R1$, wird im internen Register A abgespeichert)
(IMM-Register wird mit der Zahl 100 geladen)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- `LOAD`-Befehl
  - Die Bearbeitungsstufen (Fortsetzung):

`EX:`	$ALUOUTPUT \leftarrow A + IMM$ ([[Inhalt von $R1$] + 100] wird im internen Register $ALUOUTPUT$ gespeichert)
`MEM:`	$LMD \leftarrow MEM[ALUOUTPUT]$ (Das interne Register LMD wird mit dem Inhalt der Speicherstelle [[Inhalt von $R1$] + 100] geladen)
`WB:`	$REGS[IR[11 \ldots 15]] \leftarrow LMD$ (Das Zielregister mit der Adresse $IR[11 \ldots 15]$, d. h. $R5$, wird mit dem Wert des LMD-Registers beschrieben.)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- `STORE`-Befehl
  - Befehl vom Typ I
  - Befehlsformat

Opcode	RS	RD	IMM
0 . . . 5	6 . . . 10	11 . . . 15	16 . . . 31

  - Beispiel: `SW 10(R2), R1`
    $\rightarrow$ Opcode = `STORE WORD`; $RS = R2$; $RD = R1$; $IMM = 10$

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- `STORE`-Befehl (Fortsetzung)
  - Die Bearbeitungsstufen:

`IF`: $IR \leftarrow MEM[PC]$
$NPC \leftarrow PC + 4$
$PC \leftarrow NPC$

`ID`: $A \leftarrow REGS[IR[6 \ldots 10]]$
$B \leftarrow REGS[IR[11 \ldots 15]]$
$IMM \leftarrow IR[16 \ldots 31]$
(Inhalt des Source-Registers *R2*, wird im Register *A* gespeichert)
(Inhalt des Registers *R1*, wird im internen Register *B* gespeichert)
(*IMM*-Register bekommt den Wert 10)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- `STORE`-Befehl
  - Die Bearbeitungsstufen (Fortsetzung):

`EX`:	$ALUOUTPUT \leftarrow A + IMM$ ([[Inhalt von $R2$] + 10] wird im Register $ALUOUTPUT$ abgespeichert)
`MEM`:	$MEM[ALUOUTPUT] \leftarrow B$ (Die Speicherstelle [[Inhalt von $R2$] + 10] wird mit dem Wert des Registers B, d. h. $R1$, beschrieben)
`WB`:	In dieser Stufe ist **nichts** zu tun.

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- ALU-Register-Immediate
  - Befehl vom Typ I
  - Befehlsformat

Opcode	RS	RD	IMM
0 . . . 5	6 . . . 10	11 . . . 15	16 . . . 31

  - Beispiel: `ADDI` *R5*`,` *R4*`, #100`
    $\rightarrow$ Opcode = ALU-Register-Immediate-Operation (davon `OP` = Addition);
    $RS = R4; RD = R5; IMM = 100$

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- ALU-Register-Immediate (Fortsetzung)
  - Die Bearbeitungsstufen:

IF: $IR \leftarrow MEM[PC]$
$NPC \leftarrow PC + 4$
$PC \leftarrow NPC$

ID: $A \leftarrow REGS[IR[6 \ldots 10]]$
$IMM \leftarrow IR[16 \ldots 31]$
(Inhalt des Source-Registers *R4*, wird im internen Register *A* gespeichert)
(*IMM*-Register wird mit 100 geladen)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- ALU-Register-Immediate
  - Die Bearbeitungsstufen (Fortsetzung):

EX:	$ALUOUTPUT \leftarrow A$ OP IMM (ALU führt die Operation OP aus, die im Operationscode angegeben ist)
MEM:	In dieser Stufe ist **nichts** zu tun.
WB:	$REGS[IR[11 \ldots 15]] \leftarrow ALUOUTPUT$ (Das Ergebnis der ALU-Berechnung, d. h. [[Inhalt von $R4$] + 100], wird in das gewünschte Register $R5$ eingetragen)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- ALU-Register-Register-Befehl
  - Befehl vom Typ R
  - Befehlsformat

Opcode	RS1	RS2	RD	Func
0 . . . 5	6 . . . 10	11 . . . 15	16 . . . 20	21 . . . 31

  - Beispiel: `ADD` *R3*, *R2*, *R1*
    $\rightarrow$ Opcode = ALU-Register-Register-Operation;
    $RS1 = R2$; $RS2 = R1$; $RD = R3$; Func = Addition

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- ALU-Register-Register-Befehl (Fortsetzung)
  - Die Bearbeitungsstufen:

`IF:` $IR \leftarrow MEM[PC]$
$NPC \leftarrow PC + 4$
$PC \leftarrow NPC$

`ID:` $A \leftarrow REGS[IR[6 \ldots 10]]$
$B \leftarrow REGS[IR[11 \ldots 15]]$
(Inhalt des ersten Source-Registers (*R2*), wird im Register *A* gespeichert)
(Inhalt des zweiten Source-Registers (*R1*), wird im Register *B* gespeichert)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- ALU-Register-Register-Befehl
  - Die Bearbeitungsstufen (Fortsetzung):

EX:	*ALUOUTPUT* $\leftarrow$ *A* Func *B* (ALU führt die Funktion aus, die durch den Funktionscode im Befehlswort definiert ist (d. h. die Addition) und speichert das Ergebnis (Inhalt von *R2* + Inhalt von *R1*) in *ALUOUTPUT*)
MEM:	In dieser Stufe ist **nichts** zu tun.
WB:	*REGS*[*IR*[16 . . . 20]] $\leftarrow$ *ALUOUTPUT* (Das Zielregister (*R3*) wird mit dem Wert aus der EX-Stufe, (d. h. mit der Summe der Inhalte von *R2* und *R1*), beschrieben)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- Branch-Befehl
  - Befehl vom Typ I
  - Befehlsformat

Opcode	RS	RD	IMM
0 . . . 5	6 . . . 10	11 . . . 15	16 . . . 31

  - Beispiel: `BNEQZ` *R1*, 100
    → Opcode = Branch if Non EQual Zero (davon `OP` = non equal zero);
    RD = nicht benutzt; *IMM* = 100

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- Branch-Befehl (Fortsetzung)
  - Die Bearbeitungsstufen:

    IF: $IR \leftarrow MEM[PC]$
    $NPC \leftarrow PC + 4$
    $PC \leftarrow NPC$

    ID: $A \leftarrow REGS[IR[6 \ldots 10]]$
    $IMM \leftarrow IR[16 \ldots 31]$
    (Inhalt des Source-Registers *R1*, wird im Register *A* gespeichert)
    (*IMM* bekommt den Wert des Abstandes zwischen Sprungziel und nächstem Befehl)

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

- Branch-Befehl
  - Die Bearbeitungsstufen (Fortsetzung):

EX:	$ALUOUTPUT \leftarrow NPC + IMM$ (Die mögliche Sprungadresse wird berechnet, also $NPC + 100$) $COND \leftarrow (A$ OP $0)$ (Ob tatsächlich gesprungen wird, wird in $COND$ berechnet ($COND$ ist boolescher Wert) und enthält als Ergebnis den Vergleich des Registers A mit 0 (also ob $R1$ ungleich 0 ist oder nicht))
MEM:	IF($COND$) $PC \leftarrow ALUOUTPUT$ (Wenn die Bedingung für Branch erfüllt ist, dann wird der PC mit der vorhin berechneten Sprungadresse geladen; Also bleibt NPC als neuer PC für den Fall, dass nicht gesprungen wird.)
WB:	In dieser Stufe ist **nichts** zu tun.

Die Bearbeitungsstufen der Befehle auf der vereinfachten DLX-Pipeline

Die fünf Stufen `IF`, `ID`, `EX`, `MEM` und `WB` bilden die einzelnen Stufen der Befehlsabarbeitung.

Wenn jeder dieser Arbeitsschritte in **einem** Taktzyklus ausführbar ist, dann dauert ein DLX-Befehl maximal **fünf** Taktzyklen (Store und Branch nur **vier** Taktzyklen, weil für diese Instruktionen in der `WB`-Stufe nichts zu tun ist.)

Außerdem könnte für ALU-Befehle die `WB`-Phase in die `MEM`-Phase vorgezogen werden, da diese Befehle in der `MEM`-Phase „arbeitslos“ sind, was auch für ALU-Befehle eine Ausführungszeit von **vier** Taktzyklen bedeuten würde. (Das gilt für Integer-Befehle, bei FP-Befehlen dauern die Operationen mehr als 1 Takt.)

Prinzip der Befehlsabarbeitung in der Pipeline

	Taktzyklus							
	1	2	3	4	5	6	7	8
Befehl 01	IF	ID	EX	MEM	WB			
Befehl 02		IF	ID	EX	MEM	WB		
Befehl 03			IF	ID	EX	MEM	WB	
Befehl 04				IF	ID	EX	MEM	WB

Also: In Takt **4** sind die jeweiligen Befehle in folgenden Stufen (Arbeitsschritten):

Befehl 01	MEM
Befehl 02	EX
Befehl 03	ID
Befehl 04	IF

Prinzip der Befehlsabarbeitung in der Pipeline (Fortsetzung)

Wenn die vier Befehle ohne Probleme mit Pipelining bearbeitet werden können, dann werden insgesamt 8 Taktzyklen benötigt, was eine CPI-Rate von $\frac{8}{4} = 2$ ergibt.

Vergleich mit CPI-Rate für DLX ohne Pipeline ($4 \leq \text{CPI} \leq 5$) $\rightarrow$ Rechengeschwindigkeit mehr als verdoppelt.

Probleme der Befehlsabarbeitung in der Pipeline

Probleme durch Pipeline beim **Speicherzugriff**:
Die betroffenen Pipelinestufen sind hier `IF` und `MEM`.
In Taktzyklus 4 werden diese Stufen gleichzeitig bearbeitet (Befehl 04 und Befehl 01).
Wenn `IF` einen Befehl aus dem Speicher holen möchte und ein `LOAD`-Befehl einen Speicher-Zugriff in `MEM` nötig macht?

Wenn nur **ein** Bussystem als Verbindung zum Speicher zur Verfügung steht, dann ist ein gleichzeitiger Zugriff **nicht** möglich.

Als Lösung: Zwei separate Speicher in der CPU, einer speichert Befehle, der andere speichert Daten.

Probleme der Befehlsabarbeitung in der Pipeline (Fortsetzung)

Problem beim **gleichzeitigen Registerzugriff**: Das Problem kann bei gleichzeitiger Bearbeitung von `ID` und `WB` entstehen, weil ein Register nicht gleichzeitig gelesen und geschrieben werden kann.

Beispiel:

Register-Immediate-Befehl	IF	ID	EX	MEM	WB $REGS[IR[11\ldots15]] \leftarrow$ $ALUOUTPUT$		
⋮	⋮		⋮		⋮		
Store-Befehl				IF	ID $B \leftarrow$ $REGS[IR[11\ldots15]]$	EX	$\cdots$

Register $REGS[IR[11\ldots15]]$ kann **nicht gleichzeitig** beschrieben (Stufe `WB` des Register-Immediate-Befehls) und gelesen (Stufe `ID` des Store-Befehls) werden.

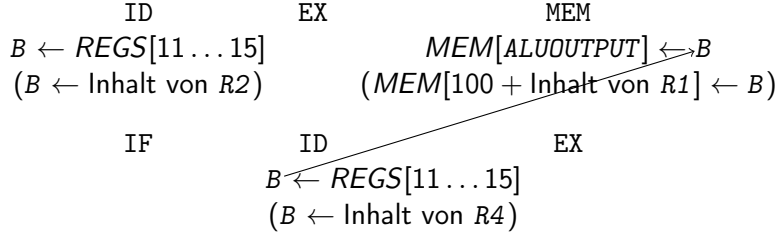
Probleme der Befehlsabarbeitung in der Pipeline (Fortsetzung)

Wenn es **zu wenige interne Register** gibt.

Betrachtet werden zwei hintereinander folgende Store-Befehle:

1. `SW 100(R1), R2`
2. `SW 10(R3), R4`

Ihre Pipeline-Bearbeitung (Stufen `ID`und `MEM`) sieht folgendermaßen aus:



Zu wenige interne Register (Fortsetzung)

Wir sehen, dass in der `MEM`-**Stufe** des **ersten** Befehls der Wert im Zwischenregister *B* **nicht aus seiner** `ID`-**Stufe**, sondern aus der `ID`-**Stufe** des nachfolgenden Befehls stammt. (Also: **Statt** des Inhalts von *R2* wird der Inhalt von *R4* gespeichert, was zu einer falschen Abarbeitung führt.)

Dieses Problem würde nicht entstehen, wenn wir genug Register in der Pipeline hätten. Es gibt deswegen sogenannte **Pipeline**-Register, die die benötigten Informationen abspeichern.

Zu wenige interne Register (Fortsetzung)

In unserem Beispiel wird das Problem mit Hilfe von zwei Registern *ID/EX.B* und *EX/MEM.B* gelöst, die zusätzlich in den ID- und EX-**Stufen** zur Verfügung stehen. Dort wird der richtige Wert von *B* abgespeichert und von der Stufe ID bis zur Stufe MEM transferiert.

Die Bearbeitung des 1. Befehls (nur *B*-Register betrachtet):

- IF
- ID:
  $B \leftarrow$ Inhalt von *R2*
  $ID/EX.B \leftarrow B$
- EX:
  $EX/MEM.B \leftarrow ID/EX.B$
- MEM:
  $MEM[ALUOUTPUT] \leftarrow EX/MEM.B$
- WB

Pipeline Hazards

Pipeline Hazards

Es gibt Situationen (genannt **Hazards**), die die Ausführung des nächsten Befehls aus dem Befehlsstrom im zugeordneten Taktzyklus **verhindern**.

Also: Ein **Hazard** ist ein Konflikt zwischen Pipelinestufen, wenn die Abhängigkeit zwischen diesen nicht beseitigt werden kann (oder die Lösung zu aufwendig wäre).

Pipeline Hazards (Fortsetzung)

Drei prinzipielle **Ursachen für Hazards**:

1. **Structure Hazards**
   Entstehen durch die **gleichzeitige Verwendung von Hardwareeinheiten durch zwei oder mehrere Stufen**.
2. **Data Hazards**
   Entstehen durch **Abhängigkeiten von Daten, die von einzelnen Befehlen zu einem Zeitpunkt benötigt werden, an denen diese nicht zur Verfügung stehen**.
3. **Control Hazards**
   Entstehen **durch verzögerte Erkennung** von Befehlen, die **den Program Counter** und damit die Programmabarbeitung verändern.

Pipeline Hazards (Fortsetzung)

Wenn ein Hazard auftritt, muss die Pipeline dafür sorgen, dass keine undefinierten (oder falschen) Zustände während der Befehlsabarbeitung eintreten.

$\rightarrow$ Die einfachste Methode: Die Stufe **blockieren**, die im Signalfluss **vor** der Stufe steht, mit der ein Konflikt auftritt.

Dieses **Blockieren** des Konflikt-Befehls: **Stall**

Alle **Befehle**, die **nach dem Stall kommen**, müssen auch **blockiert** werden. Die Befehle, die **schon** in der Pipeline sind (und vor dem blockierenden Befehl begonnen wurden), werden weiter durch die Pipeline geschickt.

Structure Hazards

Konflikt zwischen den Stufen IF und MEM.
Existiert nur ein Speicher, dann wird dieser Konflikt durch einen Stall gelöst (wenn Cache, dann kein Problem).

	Takt									
	1	2	3	4	5	6	7	8	9	10
Befehl LOAD	IF	ID	EX	MEM	WB					
Befehl 02		IF	ID	EX	MEM	WB				
Befehl 03			IF	ID	EX	MEM	WB			
Befehl 04				Stall	IF	ID	EX	MEM	WB	
Befehl 05						IF	ID	EX	MEM	WB

MEM: $LMD \leftarrow MEM[ALUOUTPUT]$

Stall: da wäre IF: auch Memory-Access
$IR \leftarrow MEM[PC]$

} → Stall

(Weil IF einen Konflikt mit MEM des LOAD-Befehls verursachen würde.)

Structure Hazards (Fortsetzung)

Nachfolgende Befehle (05) blockiert und um einen Taktzyklus verzögert.
Wenn Befehl $\underbrace{03}$ = `STORE` (03: *`MEM[ALUOUTPUT]`* $\leftarrow$ *`B`*, 05: `IF`, wieder Memory-Access) $\rightarrow$ Konflikt mit Befehl $\underbrace{05}$ $\rightarrow$ noch ein Stall nötig (d. h. Stall vor `IF` im Befehl 05).

Bemerkung: Das Schema gilt für Befehl 02, der kein Load/Store-Befehl ist (wenn das so wäre, haben wir Konflikt mit der `IF`-Stufe des Befehls 04 $\rightarrow$ noch ein Stall für Befehl 04 nötig).

Structure Hazards (Fortsetzung)

Beispiel (Abschätzung von Konsequenzen bei Structure Hazards)

Angenommen, dass eine Maschine ohne Auftreten von Hazards den idealen CPI-Wert 1.0 hätte. Eine Vergleichsmaschine hat separate Speicher (Caches) für Befehle und Daten. Die zusätzliche Hardware bedingt eine Verringerung der Taktrate um 5% gegenüber der ursprünglichen Maschine.
Welche der beiden Maschinen ist schneller, wenn in einem Programm 40% aller Befehle Loads sind?
(2 Programme (ident) auf beiden Maschinen ablaufen lassen: Wo läuft es schneller?)

Structure Hazards (Fortsetzung)

Beispiel (Fortsetzung)

Maschine 1: m (Memory)
(mit gemeinsamen Speicher)

Maschine 2: c (Cache)
(Maschine mit Cache)

CPI-Rate:
$CPI_c = 1.0$ (nie Stall)
$CPI_m = 1 + \underbrace{0.4}_{\text{bei 40\% Stall}} = 1.4$

Structure Hazards (Fortsetzung)

Beispiel (Fortsetzung)

Ausführungszeit (für ein Programm mit n Befehlen):

$t_c = n \underbrace{T_c}_{\substack{\text{Taktzykluszeit} \\ \text{für Maschine 2} \\ \text{(Periodendauer)}}} CPI_c \qquad t_m = n \underbrace{T_m}_{\substack{\text{Taktzykluszeit} \\ \text{für Maschine 1}}} CPI_m$

$T_c = \frac{1}{f_c} \qquad T_m = \frac{1}{f_m} \qquad f_c, f_m$: Frequenzen (Taktraten)

Laut Angabe: $f_c = 0.95 f_m$

$\rightarrow T_c = \frac{1}{0.95 f_m} \rightarrow s = \frac{t_m}{t_c} = \frac{n \frac{1}{f_m} CPI_m}{n \frac{1}{0.95 f_m} CPI_c} = \frac{0.95 \cdot 1.4}{1.0} = 1.33$

Maschine mit Cache ist schneller, aber höhere Hardware-Kosten.

Data Hazards

Verursacht durch **Vertauschen der Reihenfolge** von Read-/Write-Operationen von Daten (z. B. in Registern).

Beispiel:

```
ADD  R1, R2, R3
LW   R4, 0(R1)
SW   12(R1), R4
```

Konflikte bei der Pipeline-Ausführung:

- K1: Wert von *R1* wird in `LW` gebraucht
- K2: Wert von *R1* wird bei der Berechnung der Zieladresse von Store-Befehl benötigt
- K3: Store-Befehl soll *R4* abspeichern – Der Wert von *R4* wird in `LW` bereitgestellt

Lösung mit Stalls

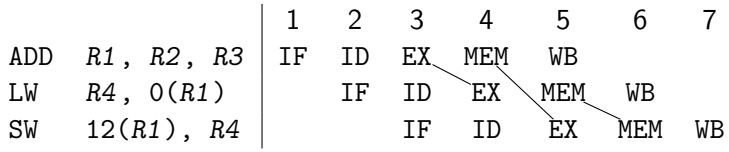
	1	2	3	4	5	6	7	8	9	10	11	12	13
ADD	IF	ID	EX	MEM	WB								
LW		Stall	Stall	Stall	IF	$\underbrace{\texttt{ID}}_{1}$	EX	MEM	WB				
SW						Stall	Stall	Stall	IF	$\underbrace{\texttt{ID}}_{2}$	EX	MEM	WB

- $\underbrace{}_{1}$: Der Wert von *R1* ist erst ab Takt **6** vorhanden und dieser Wert wird in der ID-Stufe des Load-Befehls gebraucht. Daher dürfen wir den LW-Befehl erst nach drei Stalls starten (damit $\texttt{ID}_{\texttt{LW}}$ im Takt **6** ausgeführt werden kann). Dadurch ist K1 beseitigt.
- $\underbrace{}_{2}$: (Wenn wir weniger Stalls hätten, dann wäre für $\texttt{ID}_{\texttt{LW}}$ der richtige Wert von *R1* noch nicht vorhanden, also würde $\texttt{ID}_{\texttt{LW}}$ den „alten Wert“ von *R1* nach *A* einlesen.) *R4* ist erst nach Takt 9 vorhanden und SW braucht es in der ID-Stufe: Deswegen SW nach 3 Stalls. Im Takt 10 ist auch R1 längst fertig.
- Damit sind alle drei Konfliktsituationen K1–K3 aufgelöst.

Forwarding

- Forwarding findet nur von einem Takt zum nächsten (also zwischen benachbarten Stufen) statt (sonst zusätzliche Hardware erforderlich).
- Technisch geschieht Forwarding **am Taktende** der übergebenden Stufe (z. B. wird nach der `EX`-Stufe das Ergebnis in das Pipelineregister *EX/MEM* geschrieben).
- Hardware muss Forwarding-Situation **erkennen**: Vergleich muss gemacht werden, ob das Zielregister des aktuellen Befehls ein Quellregister des folgenden Befehls ist (z. B. ist *R1* Zielregister beim `ADD`-Befehl und Input beim darauf folgenden `LOAD`-Befehl).
- Forwarding nicht immer möglich.

Lösung mit Forwarding



K1 beseitigt: Forwarding $\texttt{EX}_{\texttt{ADD}} \rightarrow \texttt{EX}_{\texttt{LW}}$
R1 nach EX-Stufe von ADD berechnet und in Phase EX von LW benutzt

K2 beseitigt: SW braucht *R1* in EX-Stufe.
Obwohl *R1* bereits in EX-Stufe von ADD berechnet ist, wird das Forwarding erst nach MEM von ADD gemacht (damit dies zwischen zwei benachbarten Stufen passiert).

K3 beseitigt: *R1* ist in *R4* nach MEM-Stufe von LW-Befehl und kann direkt forwarded werden nach $\texttt{MEM}_{\texttt{SW}}$.

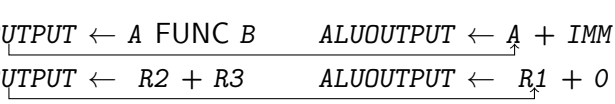
Ergebnis: keine Stalls (nur 7 Clocks)

Data Hazards (Forwarding Erklärung)

K1:

ADD *R1*, *R2*, *R3* → LW *R4*, 0(*R1*)

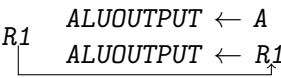
$\mathrm{EX}_{\mathrm{ADD}}$ → $\mathrm{EX}_{\mathrm{LW}}$



K2:

ADD *R1*, *R2*, *R3* → SW 12(*R1*), *R4*

$\mathrm{MEM}_{\mathrm{ADD}}$ → $\mathrm{EX}_{\mathrm{SW}}$



Fertig nach $\mathrm{EX}_{\mathrm{ADD}}$, aber Forwarding kann nur zwischen benachbarten Stufen durchgeführt werden; deswegen wird hier Forwarding nach der $\mathrm{MEM}_{\mathrm{ADD}}$-Stufe (die benachbart zur $\mathrm{EX}_{\mathrm{SW}}$-Stufe ist) durchgeführt.

Data Hazards (Fortsetzung Forwarding Erklärung)

K3:

LW *R4*, 0(*R1*) → SW 12(*R1*), *R4*

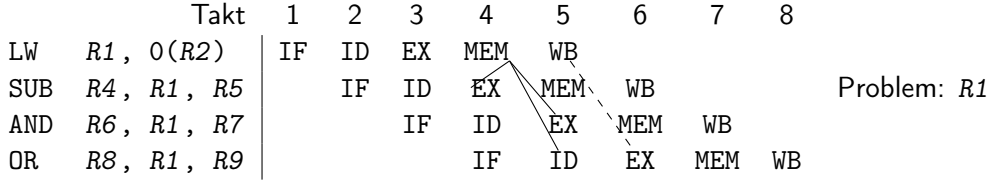
MEM_{LW} → MEM_{SW}

LMD ← *MEM[ALUOUTPUT]* *MEM[ALUOUTPUT]* ← *B*

LMD ← *MEM[0 + R1]* *MEM[12 + R1]* ← *R4*

Data Hazards (Forwarding)

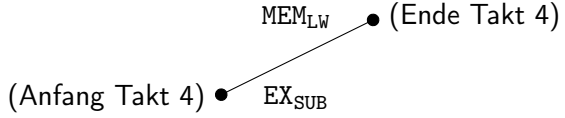
Beispiel, dass Forwarding **nicht immer** Stall vermeiden kann



- SUB-Befehl
  SUB benötigt *R1* in EX:
  EX_{SUB}: *ALUOUTPUT* ← *A* FUNC *B*
  ALUOUTPUT ← *R1* SUB *R5*
  R1 ist erst nach WB (Takt 5) verfügbar, aber bereits nach MEM (Takt 4) im internen Register *LMD* fertig.
  MEM_{LW}: *LMD* ← *MEM*[*ALUOUTPUT*]
  LMD ← *MEM*[0 + Inhalt *R2*]

Data Hazards (Forwarding)

- SUB-Befehl (Fortsetzung)
  Forwarding $\texttt{MEM}_{\texttt{LW}} \rightarrow \texttt{EX}_{\texttt{SUB}}$ nicht möglich, weil es eigentlich um Backwarding geht:



  $\rightarrow$ Hier muss die EX-Stufe des SUB-Befehls um einen Takt verzögert werden.
  $\rightarrow$ Stall kann man nicht vermeiden.
  Würde im SUB-Befehl *R1* mit *R2* ersetzt sein $\rightarrow$ kein Stall

- AND-Befehl
  R1 wird in EX-Stufe des AND-Befehls (Takt 5) gebraucht; Ergebnis *LMD* des LOAD-Befehls ist nach Takt 4 bekannt.
  $\rightarrow$ **Forwarding $\texttt{MEM}_{\texttt{LW}} \rightarrow \texttt{EX}_{\texttt{AND}}$ ist möglich.**

Data Hazards (Forwarding)

- OR-Befehl
  Dieser Befehl benötigt *R1*
  - Hier können wir folgendes Forwarding zwischen zwei benachbarten Stufen einführen:
    $\mathtt{MEM}_{\mathtt{LW}} \rightarrow \mathtt{ID}_{\mathtt{OR}}$
    (Der Wert von *R1* ist schon im *LMD*-Register (Stufe $\mathtt{MEM}_{\mathtt{LW}}$) vorhanden und dieser wird dann in der ID-Stufe des OR-Befehls in das interne Register *A* eingelesen.)
  - Ein anderes Forwarding:
    R1 ist nach $\mathtt{WB}_{\mathtt{LW}}$ fertig und sein Wert wird in der Stufe
    $\mathtt{EX}_{\mathtt{OR}}$: *ALUOUTPUT* $\leftarrow$ *A* FUNC *B*
    an der Stelle von *A* eingesetzt.

Takthalbierung

Für die Lösung des Problems mit *R1* im OR-Befehl des obigen Beispiels gibt es eine Möglichkeit **ohne** Forwarding. Es ist die Technik der **Takthalbierung**.

Wenn erlaubt ist, dass in der **ersten Hälfte** eines Taktes das Register *R1* in $\mathtt{WB}_{\mathtt{LW}}$ beschrieben wird und in der **zweiten Hälfte** dieses Taktes das selbe Register gelesen wird (in $\mathtt{ID}_{\mathtt{OR}}$), brauchen wir für die Ausführung des OR-Befehls keine Forwarding-Hardware.

Data Hazards (Takthalbierung)

Problem: *R1*
SUB-Befehl
Hier wird *R1* mit *R2* ersetzt, d. h.

LW *R1*, 0(*R2*)	IF ID EX MEM WB
SUB *R4*, *R2*, *R5*	IF ID EX MEM WB

Nach ID braucht LW *R2* nicht mehr

ID_{LW}: $A \leftarrow REGS[IR[6\ldots10]]$
$A \leftarrow R2$
$IMM \leftarrow 0$

SUB-Befehl braucht *R2* in EX-Stufe

EX_{SUB}: $ALUOUTPUT \leftarrow A$ FUNC B
$ALUOUTPUT \leftarrow R2$ SUB $R5$

Weil ID_{LW} und EX_{sub} um 2 Takte entfernt sind $\rightarrow$ kein Stall erforderlich.

Data Hazards in der Pipeline

Bei einer Maschine ohne Pipeline ist garantiert, dass **ein Befehl vollständig abgearbeitet ist, bevor der nächste bearbeitet wird**. Damit steht das Ergebnis des vorhergehenden Befehls sicher zur Verfügung. In einer Pipeline ist dies nicht mehr garantiert und dadurch können die sogenannten Data-Hazards entstehen.

Wir betrachten folgende Befehlsfolgen:

B1: `SUB R1, R2, R3`
B2: `ADD R3, R4, R5`

C1: `ADD R4, R0, R1`
C2: `SUB R2, R4, R1`

D1: `LOAD R2, 15(R3)`
D2: `MULT R2, R1, R2`

Data Hazards in der Pipeline

- **WAR** (Write After Read)

  Wenn die Befehle B1 und B2 in der Pipeline **parallel** laufen, kann es einen WAR-Hazard geben (bezüglich *R3*). Der Hazard entsteht, wenn der B2-Befehl schneller fertig ist, als B1 seine Operanden liest. In dem Fall liest B1 nicht den richtigen „alten“ Wert von *R3*, sondern den neuen, von B2 frisch erzeugten Wert. Dies führt zu einer fehlerhaften Ausführung des Codes.

Data Hazards in der Pipeline

- **RAW** (Read After Write)

  Parallele Ausführung der Befehle C1 und C2 könnte einen RAW-Hazard zur Folge haben. Es geschieht dann, wenn der zweite Befehl (C2) seinen Operanden *R4* liest, bevor C1 diesen Operanden fertig stellt. Dabei kann es zu einer fehlerhaften Ausführung des Codes kommen, weil der zweite Befehl den „alten“ Wert von *R4* lesen kann, obwohl es richtig wäre, den „neuen“ (von C1 erzeugten) Wert zu nehmen.

Data Hazards in der Pipeline

- **WAW** (Write After Write)

  Parallele Ausführung von Befehlen D1 und D2 kann zu einem WAW-Hazard führen. Wenn D2 früher sein Ergebnis schreibt als D1, dann werden die darauffolgenden Befehle nicht den richtigen Wert von *R2* (also den von D2) benutzen, sondern den von D1, was falsch wäre.

Static Scheduling

Umordnen von Befehlen, um **Data Hazards** (Load Delays) zu meiden. → Pipeline Scheduling
Dies geschieht zur Compilezeit → statische Methode.

Beispiel: Compilerarbeit

$a = b + c$
$d = e - f$ Übersetzung in DLX-Code

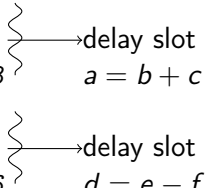
- Sequentielle Vorgangsweise

```
LW   R2, b
LW   R3, c
ADD  R1, R2, R3      a = b + c
SW   a, R1
LW   R5, e
LW   R6, f
SUB  R4, R5, R6      d = e - f
SW   d, R4
```

Static Scheduling (Fortsetzung)

→ **zwei** Load Delays (Stalls) → eliminieren

```
LW   R2, b
LW   R3, c
LW   R5, e
ADD  R1, R2, R3
LW   R6, f
SW   a, R1
SUB  R4, R5, R6
SW   d, R4
```



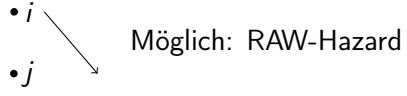
Nach dem Load-Befehl (dem Delay Slot) wird ein Befehl eingefügt, der das geladene Datum nicht verwendet.

Static Scheduling (Fortsetzung)

Damit „die passenden“ Befehle eingefügt werden, muss ein Compiler entscheiden können, welche Befehle **parallel** sind und **welche nicht**.
$\rightarrow$ Erkennung von Abhängigkeiten zwischen Befehlen.

3 Typen

- **Data dependence**
  Instruktion j benötigt ein Datum, das von Instruktion i produziert wird (möglicher RAW-Hazard).

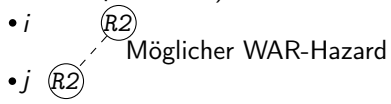


Static Scheduling (Fortsetzung)

- **Name dependence**
  Verwendung der selben Register oder Speicherstellen (**Name**) durch zwei oder mehrere Befehle, **ohne dass ein Datenfluss zwischen den Befehlen existiert**.
  - Antidependence
    Befehl i liest ein Datum und Befehl j schreibt auf den selben Namen (möglicher WAR-Hazard). Instruktion j übernimmt das Datum nicht, trotzdem kann ein Konflikt auftreten.

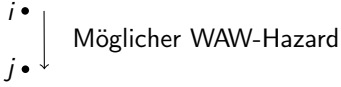
    Beispiel:
    ```
    ADD R3, R5, R2
    SUB R2, R4, R6
    ```

    Kein Datenfluss (*R2* wird in SUB nicht gebraucht, sondern produziert)

Static Scheduling (Fortsetzung)

- **Name dependence** (Fortsetzung)
  - Output Dependence
    Bezeichnet einen möglichen WAW-Hazard, beim Beschreiben des selben Namens durch Befehle i und j.



Eliminierung von **Name** dependencies

- Register Renaming
  Compiler verwendet ein anderes Register (nach Möglichkeit)
- **Control Dependence**
  Eine Abhängigkeit wenn Befehle nur **bedingt** ausgeführt werden

Eliminierung von **Name** dependencies

Static Scheduling
Limitiert durch Data Dependencies (weil diese Abhängigkeiten die Semantik des Programms repräsentieren, die der Compiler **nicht ändern darf**).
Nachteil: Große Abhängigkeit zwischen Compiler und Hardware (da der Compiler nur dann sinnvoll umordnen kann, wenn er alle Details der Hardware (insbesondere der Pipeline) kennt).

→ Idee: Das Umordnen von Befehlen während der Laufzeit des Programms **dem Prozessor** zu überlassen.
→ Dynamic Scheduling

Dynamisches Scheduling in Pipelines

(zur Umgehung von Hazards)

- Statisches versus dynamisches Scheduling
  Unsere bisherigen Pipeline-Techniken haben die Befehle in **normaler** Reihenfolge betrachtet. Wenn ein Befehl in der Pipeline angehalten wird, kann kein anderer Befehl fortsetzen. Falls es mehrere Funktionseinheiten gibt, können diese ungenutzt sein.
  Für das Scheduling von Befehlen (zur Minimierung von Wartezyklen) ist in diesem Falle die **Software** verantwortlich. Dieses Verfahren wird statisches Scheduling genannt.

  Beispiel (Statisches Scheduling)

```
DIVF F0, F2, F4
ADDF F10, F0, F8
SUBF F6, F6, F14
```

Dynamisches Scheduling in Pipelines

Der `SUBF`-Befehl muss wegen **Abhängigkeit** von `ADDF` zu `DIVF` warten. Aber `SUBF` hängt von nichts anderem in der Pipeline ab.
Diese Leistungsbegrenzung kann vermieden werden, wenn man die Abarbeitung von Befehlen **außer der Reihe** zulässt. So können `DIVF` und `SUBF` parallel (gleichzeitig) ausgeführt werden – eine außer-der-Reihe-Ausführung, weil `SUBF` vor `ADDF` bearbeitet wird.
Diese **außer-der-Reihe-Ausführung** (**out-of-order execution**) wird als **dynamisches Scheduling** bezeichnet: Hier reduziert die **Hardware** die Wartezyklen durch Umordnung der Befehlsausführung.

Das Prinzip der außer-der-Reihe-Ausführung

Die Structure und Data Hazards werden in der DLX-Pipeline in `ID`-Stufe getestet. Wenn ein Befehl wirklich ausgeführt werden könnte (in `EX`-Stufe), wird er von `ID` übergeben. Um den Beginn der Ausführung des `SUBF`-Befehls erlauben zu können, müssen wir diesen Übergabeprozess (`ID` $\rightarrow$ `EX`) in **zwei Teile** trennen:

- Testen auf Structure Hazards
- Warten auf Abwesenheit von Data Hazards

Das Prinzip der außer-der-Reihe-Ausführung (Fortsetzung)

Also werden in der außer-der-Reihe-Ausführung die Stufen `ID` (Befehlsdecodierung und Test auf alle Hazards und Operandenlesen) und `EX` (Befehlsausführung) der DLX-Pipeline durch die **drei** folgenden Stufen ersetzt:

1. **Issue** (Befehlsdecodierung und Test auf Structural Hazards)
2. **Read Operands** (Warten bis keine Data-Hazards auftreten, dann Operanden lesen)
3. **Execute** (Befehlsausführung).

Während die Stufe **1** (Issue) von allen Befehlen **in Reihe** durchlaufen wird (in-Reihe-Issue), können sich diese in der **zweiten** Stufe (Read Operands) gegenseitig anhalten und so in eine **außer-der-Reihe-Ausführung** eintreten.

Das Prinzip der außer-der-Reihe-Ausführung (Fortsetzung)

In unserem Beispiel:

```
DIVF F0, F2, F4
ADDF F10, F0, F8
SUBF F6, F6, F14
```

wird in der Read-Operands-Stufe festgestellt, dass `ADDF` das Ergebnis von `DIVF` (d. h. *F0*) liest. Wenn `DIVF` und `ADDF` parallel laufen, kann es zu einem (möglichen) **RAW-Hazard** kommen. Es kann passieren, dass das Read (Lesen von *F0* in `ADDF`) nicht nach dem Write (Schreiben von *F0* in `DIVF`), sondern zuvor kommen könnte, also wird `ADDF` von `DIVF` angehalten.
Für `SUBF` ist aber der „Weg frei“ für die außer-der-Reihe-Ausführung.

Das Prinzip der außer-der-Reihe-Ausführung (Fortsetzung)

Wir zeigen zwei Methoden für das **dynamische Scheduling** in Pipelines:

- Scoreboarding
- Tomasulo-Verfahren

Scoreboarding

Scoreboarding

Diese Methode braucht eine Hardware-Einheit: **das Scoreboard**.
Dies ist ein CPU-Manager, welcher die volle Verantwortung für die Befehlsübergabe und Befehlsausführung, einschließlich der Hazard-Erkennung, trägt.

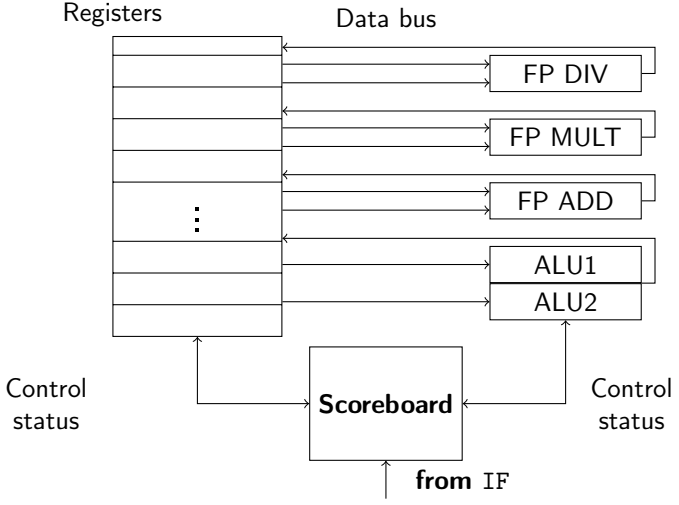
Jeder Befehl geht durch das Scoreboard, wobei die Datenabhängigkeiten **aufgezeichnet** werden.

Diese Aufzeichnungen bestimmen dann, wann der Befehl seine Operanden lesen und die Ausführung beginnen kann. Wenn das Scoreboard entscheidet, dass der Befehl nicht sofort ausgeführt werden kann, überwacht es jede Änderung der Hardware und entscheidet, wann der Befehl auszuführen ist.
Das Scoreboard überwacht auch, wann ein Befehl sein Ergebnis in das Zielregister schreiben kann.

Damit ist die gesamte Hazard-Erkennung und -Auflösung im Scoreboard zentralisiert.

Scoreboarding

Beispiel einer Architektur mit **mehreren Funktionseinheiten** und **einem Scoreboard**



Scoreboarding

Die Architektur beinhaltet:

- 3 FP Einheiten, 2 Integer ALUs, Scoreboard, Registerfile, Datenbusse

Alle Daten fließen zwischen dem Registerfile und den funktionalen Einheiten über **Busse**.

Status-Information über die Scoreboard-Steuerung der Befehlsausführung in den Funktionseinheiten sowie auch das Lesen und Schreiben von Daten im Registerfile ist in den **Befehlsstatus-**, **Funktionseinheitsstatus-** und **Ergebnisregisterstatustabellen aufgezeichnet**.
Die Aufzeichnungen in diesen Tabellen werden wir später (mit einem Beispiel) veranschaulichen.

Die vier Ausführungsschritte im Scoreboard-Verfahren

Diese Schritte sind:
Issue, *Read Operands*, *Execution* und *Write Result*

Issue und *Read Operands* ersetzen die `ID`-Stufe,
Execution die `EX`-Stufe und
Write Result die `WB`-Stufe der Standard-DLX-Pipeline.

Die `MEM`-Stufe der DLX-Pipeline wird Teil des *Execution*-Schrittes im Scoreboard (nur in Load-/Store-Befehlen relevant).

Die vier Ausführungsschritte im Scoreboard-Verfahren

1. Issue
   Das Scoreboard erhält den nächsten Befehl von der `IF`-Stufe und überprüft, ob eine funktionale Einheit für die Bearbeitung frei ist (**Structure Hazard Check**).
   Ist eine verfügbar, so wird auch geprüft, ob zur Zeit ein Befehl aktiv ist, der dasselbe Register zum Ziel hat (**WAW-Hazard Check**).
   Besteht kein Konflikt, so erfolgt die Übergabe (Issue) des Befehls an die funktionale Einheit.
   Besteht ein Konflikt, so gibt es auch hier einen Stall.

Die vier Ausführungsschritte im Scoreboard-Verfahren

2. Read Operands
   Das Scoreboard überwacht, ob eine andere Einheit Register schreiben wird, die von dem neu abgesetzten Befehl benötigt werden (**RAW-Hazard-Check**).

   Ist dies der Fall, so wird so lange gewartet, bis die Register geschrieben sind und erst dann wird die Ausführung des Befehls auf der Einheit gestartet.

   Dies hat aber auch zur Konsequenz, dass ein später abgesetzter Befehl früher bearbeitet werden kann (**Execution out of order**).

Die vier Ausführungsschritte im Scoreboard-Verfahren

3. Execution
   Die Einheit führt die Berechnung (des jeweiligen Befehls) durch und signalisiert dem Scoreboard deren Ende. Die Durchführungszeit ist abhängig von der jeweiligen Einheit.

4. Write Result
   Unter der Bedingung, dass kein vorher abgesetzter Befehl das Register benötigt und noch nicht gelesen hat (Read Operands), wird das Zielregister beschrieben.

   Wenn dieser **WAR-Hazard** entdeckt wird, blockiert das Scoreboard das Schreiben des Registers solange, bis der Konflikt aufgelöst ist.

Das Scoreboard

Das Scoreboard steuert die Befehlsabarbeitung von einem Schritt zum nächsten durch Kommunikation mit den **Funktionseinheiten** mittels **Bussystem**, das die Daten zwischen **Registern** und den einzelnen Einheiten transferiert.

Das Scoreboard hat **drei Teile**, die durch **drei Tabellen** aufgezeichnet sind:

- Befehlsstatustabelle
- Funktionseinheitsstatustabelle
- Ergebnisregisterstatustabelle

Befehlsstatustabelle

Für jeden bereits übergebenen oder noch zu übergebenden Befehl wird eine Zeile in der Tabelle reserviert.

Die Spalten der Tabelle entsprechen den Ausführungsschritten des Scoreboard-Verfahrens (`IF`, *Issue*, *Read Operands*, *Execution*, *Write Result*).

Die Einträge in der Tabelle protokollieren die Phasen und Taktzyklen, in welchen die Bearbeitung der jeweiligen Stufe des Befehls durchgeführt wurde.

Befehlsstatustabelle

Beispiel

Wenn ein Load-Befehl `LOAD F6, 34(R2)` eines Codes in der *Read Operands*-Stufe einen Eintrag ${}^{2}3^{3}$ hat,
bedeutet das, dass dieser Befehl in der 3. Bearbeitungsphase des Codes in der *Read Operands*-Stufe war,
wobei diese nach dem beendeten 2. Taktzyklus begann und bis zum beendeten 3. Taktzyklus dauerte.
(Also ist die Dauer der Bearbeitung des Befehls in der *Read Operands*-Stufe gleich 1 $(= 3 - 2)$ Taktzyklus.)

Funktionseinheitsstatustabelle

Für jede **Funktionseinheit** gibt es **eine Zeile** in der Tabelle. Wenn ein Befehl übergeben ist, werden seine Operanden in die Tabelle eingetragen.

Funktionseinheitsstatustabelle

Die Felder (Spalten) der Tabelle haben folgende Einträge:

- Busy:
  - Ja (wenn Funktionseinheit aktiv)
  - Nein (andernfalls)
- OP: Die in der Funktionseinheit auszuführende Funktion
- F_i: Das Zielregister des auf der Funktionseinheit bearbeiteten Befehls
- F_j, F_k: Die Operandenregister
- Q_j, Q_k: Die Nummern der Funktionseinheiten, die die Werte der Operandenregister erzeugen (leer wenn der Wert des Registers bereits vorhanden ist).
- R_j, R_k: Flag zur Anzeige der Bereitschaft von F_j, F_k:
  - Flag = Ja, wenn der Befehl auf den Wert des Registers wartet und ein anderer Befehl diesen Wert fertig berechnet hat.
    Also: Das Register wurde **bereits** geschrieben, aber sein Inhalt wurde noch **nicht** fertig gelesen.
  - Flag = Nein in allen anderen Fällen, d. h. entweder wir warten nicht auf den Wert des Registers oder wir warten zwar, aber der Wert des Registers ist noch nicht fertig.
    Also: Das Register wurde noch **nicht** geschrieben oder sein Inhalt wurde **bereits** fertig gelesen.

Ergebnisregisterstatustabelle

Es gibt **eine Zeile** mit **Funktionseinheiten-Nummern**.

Die Spalten entsprechen den **FP-Registern** (die Einträge zeigen die Nummern der Funktionseinheiten, welche die Werte der betrachteten Register erzeugen; andernfalls sind die Felder leer.).

Beispiel (Scoreboard-Algorithmus)

Betrachtet wird folgende Befehlsfolge:

```
LF F6, 34(R2)
LF F2, 45(R3)
MULTF F0, F2, F4
SUBF F8, F6, F2
DIVF F10, F0, F6
ADDF F6, F8, F2
```

Und 5 Funktionseinheiten:

FE1: Integer
FE2: MULT1
FE3: MULT2
FE4: ADD
FE5: DIV.

Wir zeigen Status-Tabellen nach **drei Etappen** der Abarbeitung des Codes.

Beispiel (Scoreboard-Algorithmus)

- Regel 1:
  Die Befehle sind gefetched in die Pipeline in der vorgeschriebenen Reihenfolge (LF1, LF2, `MULTF`, `SUBF`, `DIVF`, `ADDF`)
- Regel 2:
  Read Operands-Stufe kann starten, wenn alle Operands da sind (z. B. Phase 13 beim `DIVF`-Befehl)
- Regel 3:
  Write Result-Stufe kann beginnen, wenn kein WAR-Hazard entstehen kann (z. B. kann der `ADDF`-Befehl *F6* erst dann schreiben (Phase 14, Taktzyklus 20), wenn der „alte“ Wert von *F6* im `DIVF`-Befehl gelesen wurde (Phase 13, Taktzyklus 19))

Beispiel (Scoreboard-Algorithmus)

Befehlsstatustabelle

	IF	*Issue*	*Read Operands*	*Execution*	*Write Results*
`LF F6, 34(R2)`	$^{0}1^{1}$	$^{1}2^{2}$	$^{2}3^{3}$	$^{3}4^{4}$	$^{4}5^{5}$
`LF F2, 45(R3)`	$^{1}2^{2}$	$^{2}3^{3}$	$^{3}4^{4}$	$^{4}5^{5}$	$^{5}6^{6}$
`MULTF F0, F2, F4`	$^{2}3^{3}$	$^{3}4^{4}$	$^{6}7^{7}$	$^{7}8^{17}$	$^{17}12^{18}$
`SUBF F8, F6, F2`	$^{3}4^{4}$	$^{4}5^{5}$	$^{6}7^{7}$	$^{7}8^{9}$	$^{9}9^{10}$
`DIVF F10, F0, F6`	$^{4}5^{5}$	$^{5}6^{6}$	$^{18}13^{19}$	$^{19}14^{59}$	$^{59}15^{60}$
`ADDF F6, F8, F2`	$^{5}6^{6}$	$^{6}7^{7}$	$^{10}10^{11}$	$^{11}11^{13}$	$^{19}14^{20}$

Beispiel (Scoreboard-Algorithmus, Etappe 1)

Diese Etappe endet gerade bevor der zweite Load-Befehl zum Ergebnisschreiben übergeht, d. h. nach Taktzyklus 5. Angenommen wird, dass `IF`, *Issue*, *Read Operands*, *Write Result* jeweils 1 Taktzyklus benötigen und *Execution* bei `LOAD` ebenso 1 Taktzyklus.

- Befehlsstatustabelle
  In dieser Etappe werden insgesamt **fünf** Bearbeitungsphasen durchgeführt. (Diese sind in der gesamten Befehlsstatustabelle rot markiert.)

Beispiel (Scoreboard-Algorithmus, Etappe 1)

- Phase 1:
  - Der erste Load-Befehl (im Folgenden bezeichnet als LF1) ist aus dem Speicher gefetched
- Phase 2:
  - LF1 in der *Issue*-Stufe
  - Der zweite Load-Befehl (LF2) ist gefetched
- Phase 3:
  - LF1 in der *Read Operands*-Stufe (keine Wartezeit in der *Issue*-Stufe, weil dort kein Structure- oder WAW-Hazard auftritt)
  - LF2 in der *Issue*-Stufe
  - `MULTF`-Befehl geholt
- Phase 4:
  - LF1 hat die *Execution*-Stufe beendet
  - LF2 liest die Operanden
  - `MULTF` ist in der *Issue*-Stufe
  - `SUBF`-Befehl aus dem Speicher gefetched

Beispiel (Scoreboard-Algorithmus, Etappe 1)

- Phase 5: (Die letzte Phase bevor LF2 sein Ergebnis schreibt)
  - LF1 schreibt sein Ergebnis in *F6*
  - LF2 in der *Execution*-Stufe
  - `MULTF` muss in der *Issue*-Stufe aufgehalten werden (er kann nicht in die *Read Operands*-Stufe vorrücken, weil der Wert von *F2* aus dem LF2-Befehl noch nicht vorhanden ist)
  - `SUBF`-Befehl ist in der *Issue*-Stufe
  - `DIVF` wird geholt

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

(nach Beendigung der 1. Etappe, d. h. nach den ersten 5 Bearbeitungsphasen)

Anmerkung für R_j, R_k: Ja: Das Register wurde **bereits** geschrieben, aber sein Inhalt wurde noch **nicht** fertig gelesen.

Nein: Das Register wurde noch **nicht** geschrieben oder sein Inhalt wurde **bereits** fertig gelesen.

FE-Nr	Name	Busy	OP	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
1	Integer	Ja	`LOAD`	*F2*	*R3*				Nein	Nein
2	MULT1	Ja	`MULT`	*F0*	*F2*	*F4*	1		Nein	Ja
3	MULT2	Nein								
4	ADD	Ja	`SUB`	*F8*	*F6*	*F2*		1	Ja	Nein
5	DIV	Ja	`DIV`	*F10*	*F0*	*F6*	2		Nein	Ja

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

(nach Beendigung der 1. Etappe, d. h. nach den ersten 5 Bearbeitungsphasen)

- FE1 (Integer)
  Bearbeitet den zweiten Load-Befehl (LF2): `LF F2, 45(R3)`
  (Es wird angenommen, dass der erste Load-Befehl bereits fertig ist.)
- FE2 (MULT1)
  Bearbeitet den `MULTF`-Befehl: `MULTF F0, F2, F4`
- FE3 (MULT2)
  Bearbeitet keinen Befehl (Busy stets auf Nein gesetzt)
- FE4 (ADD)
  Bearbeitet den `SUBF`-Befehl: `SUBF F8, F6, F2`
- FE5 (DIV)
  Bearbeitet den `DIVF`-Befehl: `DIVF F10, F0, F6`

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

- 1. Zeile (FE1):
  - Busy = Ja (weil auf FE1 der zweite Load-Befehl in Bearbeitung ist)
  - OP = `LOAD`
  - $F_i = F2$ (das Zielregister des LF2-Befehls)
  - $F_j = R3$ (das erste Operandenregister)
  - F_k = leer (weil es kein zweites Operandenregister gibt)
  - Q_j, Q_k = leer (weil die Werte der Operandenregister in keiner der vorhandenen Funktionseinheiten erzeugt werden)
  - R_j, R_k = Nein (weil nicht auf die Operandenregister gewartet wird)
    R_j = Nein weil $F_j = R3$ fertig gelesen wurde
    R_k = Nein weil F_k kein Operand ist, also Register nicht geschrieben wurde

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

- 2. Zeile (FE2):
  - Busy = Ja (weil auf FE2 der `MULTF`-Befehl in Bearbeitung ist (in der *Issue*-Stufe))
  - OP = `MULT`
  - $F_i = F0$ (das Zielregister des `MULTF`-Befehls)
  - $F_j, F_k = F2, F4$ (Operandenregister)
  - $Q_j = 1$ (weil $F_j = F2$ in Erzeugung auf FE1 ist)
  - $Q_k =$ leer (weil $F_k = F4$ auf keiner der vorhandenen Funktionseinheiten produziert wird)
  - $R_j =$ Nein (es wird auf den Wert des Registers $F2$ gewartet, welcher noch nicht fertig ist, d. h. noch nicht geschrieben wurde)
  - $R_k =$ Ja (weil wir auf $F4$ warten, aber ein anderer Befehl (irgendwann vorher, außerhalb unseres Code-Windows) dieses bereits produziert hat, d. h. $F_k = F4$ wurde irgendwann vorher geschrieben)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

- 3. Zeile (FE3):
  - Leer (weil kein Befehl auf FE3 bearbeitet wird)
- 4. Zeile (FE4):
  - Busy = Ja (der `SUBF`-Befehl wird auf FE4 bearbeitet)
  - OP = `SUB`
  - $F_i = F8$ (das Zielregister des `SUBF`-Befehls)
  - $F_j, F_k = F6, F2$ (die Operandenregister)
  - $Q_j =$ leer (weil der Wert von $F6$ bereits fertig ist)
  - $Q_k = 1$ (weil FE1 den Wert des zweiten Operandenregisters $F_k = F2$ erzeugt)
  - $R_j =$ Ja (weil $F_j = F6$ zwar geschrieben wurde – durch den ersten Load-Befehl, aber noch nicht fertig gelesen wurde, d. h. die *Read Operands*-Stufe des `SUBF`-Befehls wurde noch nicht abgeschlossen)
  - $R_k =$ Nein (es wird auf $F2$ gewartet, dessen Wert FE1 noch nicht erzeugt hat, d. h. $F_k = F2$ wurde noch nicht geschrieben)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

- 5. Zeile (FE5):
  - Busy = Ja (der `DIVF`-Befehl ist auf FE5 in Bearbeitung)
  - OP = `DIV`
  - F_i = *F10* (das Zielregister des `DIVF`-Befehls)
  - F_j, F_k = *F0*, *F6* (die Operandenregister)
  - Q_j = 2 (weil FE2 den ersten Operanden produziert)
  - Q_k = leer (weil keine FE mehr den Wert von *F6* berechnet, welcher vorher durch FE1 fertig erzeugt wurde)
  - R_j = Nein (weil F_j = *F0* noch nicht fertig ist, d. h. noch nicht geschrieben wurde)
  - R_k = Ja (weil F_k = *F6* fertig ist, aber noch nicht fertig gelesen wurde, d. h. die *Read Operands*-Stufe des `DIVF`-Befehls noch nicht abgeschlossen ist – unsere Tabelle zeigt den Snapshot nach dem 5. Taktzyklus)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Ergebnisregisterstatus (nach Beendigung der 1. Etappe)

Register	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	.....	*F30*
FE-Nr	2	1			4	5		.....	

Der Wert von *F0* erzeugt durch FE2.
Der Wert von *F2* erzeugt durch FE1.
Der Wert von *F8* erzeugt durch FE4.
Der Wert von *F10* erzeugt durch FE5.

FE2: `MULTF F0, F2, F4`
FE1: `LF F2, 45(R3)`
FE4: `SUBF F8, F6, F2`
FE5: `DIVF F10, F0, F6`

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Etappe 2

Diese Etappe endet gerade wenn der `MULTF`-Befehl bereit ist, in den Zustand Ergebnisschreiben zu gehen.

Zu den Annahmen in der Etappe 1 wird hier weiter angenommen, dass die *Execution*-Stufe für die jeweiligen Befehle folgende Zeiten benötigen:

- Addition: 2 Taktzyklen
- Multiplikation: 10 Taktzyklen
- Division: 40 Taktzyklen

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Befehlsstatustabelle

In dieser Etappe werden insgesamt 6 Berechnungsphasen (Phasen 6–11) durchgeführt (in der Gesamttabelle grün markiert). Diese Etappe endet mit Taktzyklus 17.

Phase 6:

- LF2 schreibt sein Ergebnis in *F2*
- `MULTF` ist aufgehalten in der *Issue*-Stufe (er kann noch immer nicht in die Stufe *Read Operands* übergehen, weil der Wert von *F2* erst nach dieser Phase vorhanden ist)
- `SUBF` auch aufgehalten
- `DIVF` geht in die *Issue*-Stufe über
- `ADDF` wird gefetched

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Phase 7:

- `MULTF` und `SUBF` können ihre Operanden lesen
- `DIVF` wartet in der *Issue*-Stufe $(*1)$ (warten auf *F0*)
- `ADDF` geht in die *Issue*-Stufe über

Phase 8:

- `MULTF`, `SUBF` sind in der *Execution*-Stufe (gleichzeitig möglich, weil mehrere Funktionseinheiten vorhanden sind)
- `DIVF` aufgehalten in der *Issue*-Stufe (wegen $(*1)$)
- `ADDF` auch aufgehalten in der *Issue*-Stufe $(*2)$ (weil er für den Übergang in die *Read Operands*-Stufe *F8* braucht)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Phase 9:

- `SUBF` schreibt sein Ergebnis in *F8* (also ist *F8* nach 10 Taktzyklen fertig)
- `MULTF` ist in der *Execution*-Stufe $(*3)$ (weil diese Stufe erst nach dem Taktzyklus 17 fertig wird)
- `DIVF`, `ADDF` aufgehalten ($(*1)$, $(*2)$)

Phase 10:

- `ADDF` kann in die Stufe *Read Operands* vorrücken (weil sowohl *F2* als auch *F8* vorhanden sind)
- `MULTF` wie oben (wegen $(*3)$, weil wir uns erst im Taktzyklus 11 befinden)
- `DIVF` wie oben ($(*1)$)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Phase 11: (die letzte Phase bevor `MULT` sein Ergebnis in *F0* schreibt)

- `ADDF` ist in der *Execution*-Stufe
- `MULTF` wie oben ($(*3)$)
- `DIVF` wie oben ($(*1)$)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Status der Funktionseinheiten
(nach Abschluss der 2. Etappe, d. h. nach 11 Bearbeitungsphasen und 17 Taktzyklen)

Anmerkung für R_j, R_k: Ja: Das Register wurde **bereits** geschrieben, aber sein Inhalt wurde noch **nicht** fertig gelesen.

Nein: Das Register wurde noch **nicht** geschrieben oder sein Inhalt wurde **bereits** fertig gelesen.

FE-Nr	Name	Busy	OP	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
1	Integer	Nein								
2	MULT1	Ja	`MULT`	*F0*	*F2*	*F4*			Nein	Nein
3	MULT2	Nein								
4	ADD	Ja	`ADD`	*F6*	*F8*	*F2*			Nein	Nein
5	DIV	Ja	`DIV`	*F10*	*F0*	*F6*	2		Nein	Ja

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Detaillierte Erklärung für FE4:

4. Zeile (FE4):
   - Busy = Ja (der `ADDF`-Befehl ist in Bearbeitung auf FE4)
   - OP = `ADD`
   - F_i = *F6* (das Zielregister des `ADDF`-Befehls)
   - F_j, F_k = *F8*, *F2* (die Operandenregister)
   - Q_j, Q_k = leer (Werte beider Quellregister stehen bereits zur Verfügung)
   - R_j = Nein (weil F_j = *F8* bereits in Taktzyklus 11 vom `ADDF`-Befehl eingelesen wurde)
   - R_k = Nein (weil F_k = *F2* bereits in Taktzyklus 11 vom `ADDF`-Befehl eingelesen wurde, dieser Wert war bereits nach Taktzyklus 6 vorhanden)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Detaillierte Erklärung für die Spalten R_j, R_k der übrigen Zeilen der Tabelle:

2. Zeile (FE2, MULT1): `MULTF F0, F2, F4`
   - R_j = Nein (weil F_j = *F2* fertig gelesen wurde – bereits in Taktzyklus 7)
   - R_k = Nein (weil F_k = *F4* bereits fertig gelesen wurde)

5. Zeile (FE5, DIV): `DIVF F10, F0, F6`
   - R_j = Nein (weil F_j = *F0* noch nicht geschrieben wurde – wartet auf Schreiben, was erst in Taktzyklus 18 passieren wird)
   - R_k = Ja (weil F_k = *F6* bereits geschrieben wurde, aber vom `DIVF`-Befehl noch nicht fertig gelesen wurde – die *Read Operands*-Stufe wird erst in Taktzyklus 19 abgeschlossen)

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Ergebnisregisterstatus (nach Beendigung der 2. Etappe)

Register	*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	.....	*F30*
FE-Nr	2			4		5		.....	

- Der Wert von *F0* wird in FE2 erzeugt.
  FE2: `MULTF` *F0*, *F2*, *F4*
- Der „neue“ Wert von *F6* wird durch den `ADDF`-Befehl in FE4 erzeugt.
  FE4: `ADDF` *F6*, *F8*, *F2*
- Der Wert von *F10* (Ergebnis des `DIVF`-Befehls) wird in FE5 erzeugt.
  FE5: `DIVF` *F10*, *F0*, *F6*

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Etappe 3

Diese endet mit der Bearbeitung des Beispiel-Codes, also gerade, wenn der `DIVF`-Befehl die Stufe Ergebnisschreiben abgeschlossen hat.

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Befehlsstatustabelle

In dieser abschließenden Etappe werden 4 Berechnungsphasen (Phasen 12–15) durchgeführt.

Phase 12:

- `MULTF` schreibt sein Ergebnis in *F0*
- `DIVF` wartet (noch immer in der *Issue*-Stufe) auf das Ende der Schreiboperation von `MULTF`
- `ADDF` aufgehalten von der Schreibstufe $(*4)$ (weil wegen *F6* ein WAR-Hazard droht: wir brauchen in `DIVF` den „alten“ Wert von *F6* und nicht den „neuen“ (der durch `ADDF` erzeugt wird))

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Phase 13:

- `DIVF` kann Operanden lesen (*F6* aus dem ersten LOAD-Befehl und *F0* aus dem `MULTF`)
- `ADDF` wartet (wegen $\circledast 4$)

Phase 14:

- `DIVF` ist in der *Execution*-Stufe (wird nach 59 Taktzyklen fertig)
- `ADDF` kann sein Ergebnis in *F6* schreiben (weil `DIVF` in der vorigen Phase „*F6* alt“ gelesen hat)

Phase 15:

- `DIVF` in der Schreibstufe

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

- Status der Funktionseinheiten
  In der Tabelle sind alle Einträge in der Spalte Busy gleich Nein, sonst alles leer (weil alle Berechnungen abgeschlossen sind)
- Ergebnisregisterstatus
  Aus dem obigen Grund sind alle Einträge leer

Beispiel (Scoreboard-Algorithmus, Status der Funktionseinheiten)

Zusammenfassung

Das Scoreboard-Verfahren hat unseren Code nach insgesamt 60 Taktzyklen bearbeitet.

Das Verfahren hat folgende Umordnung der Befehlsreihenfolge bewirkt:

- `SUBF` vor `MULTF`
  (`SUBF` endet nach 10 Taktzyklen in der Phase 9 und `MULTF` nach 18 Taktzyklen in der Phase 12. In Taktzyklus 9 ist der Befehl `SUBF` in der letzten Bearbeitungsphase und der Befehl `MULTF` (der vor ihm steht) in der vorletzten.)
- `ADDF` vor `DIVF`
  (Phase 14 (Takt 20) versus Phase 15 (Takt 60), `ADDF` vor `DIVF` in Taktzyklus 19)

Trotz dieser dynamischen Umordnung der Befehlsabarbeitung haben wir richtige Ergebnisse bekommen.

Tomasulo-Verfahren

Tomasulo-Verfahren

Grundidee und Vergleich zum Scoreboarding

Methode der parallelen Ausführung, benannt nach dem Entwickler der IBM 360 Rechnerfamilie

- Verteilung der Hazard-Kontrolle und der Ausführungssteuerung. **Reservation Stations** (Reservierungstabellen) an jeder Funktionseinheit informieren die Steuerung, wenn ein Befehl mit der Ausführung an dieser Einheit beginnen kann. (Beim Scoreboarding ist diese Funktion zentralisiert.)
- Ergebnisse werden den Funktionseinheiten direkt übermittelt, anstatt dass sie durch Register gehen. Dies geschieht durch **Register Renaming**, wobei als Referenzziel nicht Register sondern Funktionseinheiten angegeben werden.

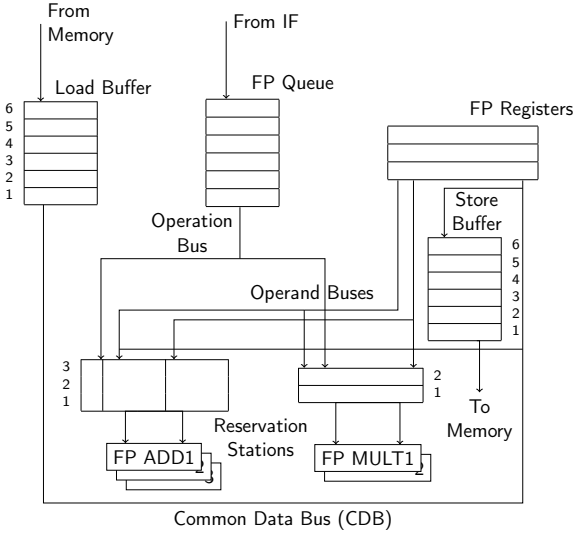
Tomasulo-Verfahren

Grundidee und Vergleich zum Scoreboarding

- Im Tomasulo-Verfahren gibt es einen gemeinsamen Ergebnisbus (genannt **Common Data Bus (CDB)**), der allen Einheiten ein simultanes Laden der Operanden erlaubt.
  Das Scoreboard schreibt Ergebnisse in Register, während andere Funktionseinheiten um sie konkurrieren.
- Das Scoreboard hat auch mehrere Busse zur Beendigung von Befehlen, während die Tomasulo-Einheit nur einen hat.

Die Grundstruktur einer Tomasulo-Pipeline-FP-Einheit

Ein Beispiel, welches nur einen Teil der gesamten CPU darstellt – es fehlen z. B. Funktionseinheiten für Integer- und Load-/Store-Operationen, die R-Register, etc.:



Die Grundstruktur einer Tomasulo-Pipeline-FP-Einheit

- FP-Operationen werden von der Befehlseinheit in eine Queue gesendet (FP Queue). Von dort werden sie über den Operation Bus an die Funktionseinheiten verteilt.
- Im Load Buffer werden Adressen wartender Ladebefehle gehalten.
- Im Store Buffer sind die Adressen offener, auf ihre Operanden wartende Speicherbefehle gehalten.
- FP-Einheiten: Vorhanden sind **drei** `ADD`-Einheiten (ADD1, ADD2, ADD3) für FP-Addition/Subtraktion und **zwei** `MULT`-Einheiten (MULT1, MULT2) für FP-Multiplikation/Division.
- Jede FP-Funktionseinheit ist durch eine Reservation Station überwacht.

Die Grundstruktur einer Tomasulo-Pipeline-FP-Einheit

- Aus den FP-Registern werden die Operanden (über Operand Buses) zu den FP-Einheiten gebracht (als Werte der Register V_j und V_k in den zuständigen Reservation Stations gespeichert).
  Für jede FE gibt es zwei Register in der FP-Register-Spalte für das Speichern von Werten in V_j und V_k.
- Alle Ergebnisse von den FP-Einheiten oder den Ladeeinheiten werden dem **Common Data Bus (CDB)** übergeben und zu dem FP-Register-File, den Reservation Stations und Speicher-Puffern übertragen.

Die Schritte des Tomasulo-Verfahrens für die Befehlsabarbeitung

Im Tomasulo-Verfahren gibt es **drei** Schritte (Stufen), die ein Befehl durchläuft: Issue, Execute und Write Result.

Der Schritt Issue entspricht der `ID`-Stufe, Execute der `EX`- und `MEM`-Stufe und Write Result der `WB`-Stufe einer einfachen Pipeline.

1. Issue (Übergabe)

In der FP-Queue, die von der IF-Stufe mit FP-Instruktionen versorgt wird, wird der nächste Befehl analysiert:
Ist es eine FP-Operation, so wird überprüft, ob eine Reservation Station bereit ist, den Befehl zu übernehmen.
Ist es ein Load/Store, dann wird geprüft, ob im jeweiligen Buffer noch ein Eintrag frei ist, um den Befehl auszuführen. Hier passiert auch das **Register Renaming** (Erklärung dazu später), indem einer **Reservation Station (RS)** mitgeteilt wird, woher die Operanden kommen.
Dadurch werden **WAW**- und **WAR**-Hazards eliminiert (Erklärung später).

2. Execute

Die Reservation Station stellt fest, ob alle Operanden für den Befehl zur Verfügung stehen.
Falls die Operanden noch nicht vorhanden sind, wartet die Reservation Station, bis über den Common Data Bus die Operanden eingetroffen sind (**RAW**-Hazard-Check – Erklärung später).
Stehen die Operanden zur Verfügung, dann wird der Befehl an die zuständige Funktionseinheit übergeben.

3. Write Result

Wenn ein Operand verfügbar ist (fertig berechnet oder geladen), dann wird er über den CDB an alle wartenden RS, in das Register-File und in einen wartenden Store-Befehl geschrieben. (Dies minimiert die Wartezeiten, da ein Ergebnis an alle wartenden Einheiten sofort und gleichzeitig übermittelt wird.)

Tomasulo-Verfahren

Beim Tomasulo-Verfahren wird die Information über den Bearbeitungsstatus der Befehle durch Einträge in speziellen Tabellen angegeben. Diese Tabellen werden im folgenden Beispiel näher präzisiert.

Beispiel (Tomasulo-Verfahren)

Behandelt wird der gleiche Code wie beim Scoreboard-Verfahren:

```
LF F6, 34(R2)
LF F2, 45(R3)
MULTF F0, F2, F4
SUBF F8, F6, F2
DIVF F10, F0, F6
ADDF F6, F8, F2
```

Die Execute-Zeiten für die Floating Point-Funktionseinheiten (FP-FE) sind:
2 Taktzyklen (`ADDF`), 4 Taktzyklen (`LOADF`), 10 Taktzyklen (`MULTF`) und 40 Taktzyklen (`DIVF`).

Die Issue- und Write Result-Stufen benötigen jeweils 1 Taktzyklus.

Beispiel (Tomasulo-Verfahren)

Wir zeigen den Verlauf der oberen Befehlsfolge laut Tomasulo-Algorithmus mittels **Befehlsstatus (BST)**-, **Reservierungs (RVT)**- und **Registerstatus (RET)**-Tabellen.

Bemerkung: Die BST ist nur zur Veranschaulichung da, ist aber nicht Teil des Tomasulo-Algorithmus.

Beispiel (Tomasulo-Verfahren)

BST (Befehlsstatustabelle)

- Zeilen: **6 Befehle** des Codes
- Spalten: **3 Stufen** (Issue, Execute, Write Result)
  (IF-Stufe wird nicht berücksichtigt, weil angenommen wird, dass die FP-Befehle bereits in der Einheit queued sind.)
- Einträge: **Die Phasennummern mit oberen Indizes der Taktzyklen** für Anfang und Ende der Bearbeitungsstufe (z. B. bedeutet ${}^{2}3^{3}$, dass die Stufe in der Phase 3 bearbeitet wird, die nach dem Taktzyklus 2 beginnt und im 3. Taktzyklus endet – ihre Dauer ist also $3 - 2 = 1$ Taktzyklen).

Beispiel (Tomasulo-Verfahren)

RVT (Reservierungstabellen)

- Zeilen: Für jede der **5**[*] arithmetischen FP-FEn (d. h. **ADD1, ADD2, ADD3, MULT1, MULT2**) eine Zeile.
  (Die FP-Load Einheiten **LOAD1** und **LOAD2** für die zwei Load-Befehle werden nicht gezeigt, damit die Tabellen übersichtlich sind.)

[*](Es handelt sich um die **Vereinigung zweier** Reservierungstabellen – einer **3**-zeiligen für die Additionseinheiten und einer **2**-zeiligen für die Multiplikationseinheiten.)

Beispiel (Tomasulo-Verfahren)

- Spalten: Felder (6) jeder Reservation Station (d. h. **BUSY**, **OP**, V_j, V_k, Q_j, Q_k)
  - BUSY: Ja (wenn FE aktiv), Nein (andernfalls)
  - OP: Die Operation, die auf der zuständigen FE in Ausführung ist
  - V_j bzw. V_k: (**Werte** des ersten bzw. des zweiten Operanden-Registers)
    - Eintrag **leer**, wenn der Wert des Operand-Registers nicht vorhanden ist
    - Eintrag **(X)**; **(X)** zeigt, dass das Feld der RVT das Ergebnis der Funktionseinheit **X** oder den Inhalt von Register **X** zum Zeitpunkt der Übergabe enthält.
  - Q_j bzw. Q_k:
    - Eintrag **X**; **X** ist die Funktionseinheit, wo der Wert des entsprechenden Operandenregisters (V_j bzw. V_k) in Bearbeitung ist
    - andernfalls **leer**

Beispiel (Tomasulo-Verfahren)

Bemerkung: In jedem Paar (V_l, Q_l), $l = j, k$, ist mindestens ein Eintrag leer.

Begründung: Entweder sind beide Einträge leer (der Befehl ist noch nicht in Bearbeitung, also noch nicht issued) **oder** V_l nicht-leer und Q_l leer (der Wert von V_l ist vorhanden und daher bearbeitet keine FE diesen Wert) **oder** V_l leer und Q_l nicht-leer (FE Q_l bearbeitet V_l).

Beispiel (Tomasulo-Verfahren)

RET (Registerstatustabelle)

- Zeilen: 2 Zeilen
  - Q_i:
    - **Name der FE**, die den Wert des betrachteten Registers erzeugt
    - Eintrag **leer**, wenn es keine FE gibt, die den Wert des Registers bearbeitet
  - BUSY:
    - Ja (wenn der Wert des Registers in Bearbeitung durch die im Feld Q_i angegebene FE ist)
    - leer (andernfalls)

Beispiel (Tomasulo-Verfahren)

Bemerkung: Die RVT- und RET-Tabellen ändern sich nur dann, wenn ein Befehl entweder issued ist, oder die Write Result-Stufe abgeschlossen ist.

Begründung:

- Issue:
  - **Wenn ein arithmetischer Befehl issued ist**, dann werden in der **Reservierungstabelle** (in die Zeile der ausführenden FE) folgende Einträge geschrieben:
    Ins Feld BUSY (= Ja), ins Feld OP (= der Name der Befehlsoperation) und jeweils ein Eintrag in die Paare (V_j, Q_j) und (V_k, Q_k) (je nach dem, ob der Registerwert vorhanden (V) oder in Bearbeitung ist (Q).
  - **Wenn ein Befehl (arithmetisch oder Load/Store) issued ist**, dann wird in der **Registerstatustabelle** unter dem Zielregister der Eintrag BUSY (= Ja) und Q_i (= Name der auszuführenden FE) geschrieben.

Beispiel (Tomasulo-Verfahren)

Begründung (Fortsetzung):

- Write Result:
  - **Wenn die Bearbeitung des arithmetischen Befehls abgeschlossen ist**, dann werden in der Zeile der auszuführenden FE in der **Reservierungstabelle** alle Einträge gelöscht.
  - **Wenn der Befehl (arithmetisch oder Load/Store) abgeschlossen ist** (also dessen Wert auf den CDB verschickt wurde), bekommen alle auf dieses Ergebnis wartenden Funktionseinheiten den Namen der ausführenden FE in der **Reservierungstabelle** unter V_l ($l = j$ oder $l = k$) eingetragen und unter Q_l gelöscht.
  - In der **Registerstatustabelle** werden für Zielregister **des abgeschlossenen Befehls (arithmetisch oder Load/Store)** die Einträge (BUSY, Q_i) gelöscht.

Beispiel (Tomasulo-Verfahren)

Beispiel (Register Renaming)

```
LF F2, 45(R3)
ADDF F6, F8, F2
```

Es wird angenommen, dass der Wert des Registers *F8* für den ADDF-Befehl bereits vorhanden ist, aber der Load-Befehl die Write Result-Stufe noch nicht abgeschlossen hat. (Also war der Wert des Registers *F2* noch nicht auf dem CDB und steht daher der ADD2-Funktionseinheit, die den ADDF-Befehl ausführt, noch nicht zur Verfügung.)
Die ADD2-Einheit wartet aber nicht auf das Schreiben von *F2*, sondern geht mit dem ADDF-Befehl in die Issue-Stufe über.

Das ist möglich, weil die Hardware der ADD2-Funktionseinheit die Information gibt, welche Einheit den fehlenden Operanden gerade berechnet.

Der fehlende Wert des Registers *F2* wird also in der Issue-Stufe durch den Namen der erzeugenden FE, d. h. LOAD2, ersetzt. Dieser Prozess heißt **Register Renaming**.

Beispiel (Tomasulo-Verfahren)

Beispiel (Register Renaming) (Fortsetzung)

In der Reservierungstabelle der ADD2 Funktionseinheit dokumentieren wir diesen Prozess folgendermaßen:

Name	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD2	Ja	`ADDF`	(*F8*)			LOAD2

Wenn der Load-Befehl seine Ausführung beendet hat, wir der Wert von *F2* auf die wartende ADD2 Einheit übertragen. **Diese Einheit kann jetzt mit dem Wert des Registers *F2* (unter dem Namen der ausführenden FE, d. h. LOAD2) in die Execute-Stufe vorrücken.**

Die Einträge in der Reservierungstabelle sehen jetzt so aus:

Name	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD2	Ja	`ADDF`	(*F8*)	(LOAD2)		

Beispiel (Tomasulo-Verfahren)

Die Bearbeitung des Code-Beispiels

Der Code wird in insgesamt 14 Phasen bearbeitet. Diese sind in der folgenden **Befehlsstatustabelle** aufgezeichnet:

Befehlsstatustabelle (BST)

	Issue	Execute	Write Result
`LF F6, 34(R2)`	$^{0}1^{1}$	$^{1}2^{5}$	$^{5}6^{6}$
`LF F2, 45(R3)`	$^{1}2^{2}$	$^{2}3^{6}$	$^{6}7^{7}$
`MULTF F0, F2, F4`	$^{2}3^{3}$	$^{7}8^{17}$	$^{17}12^{18}$
`SUBF F8, F6, F2`	$^{3}4^{4}$	$^{7}8^{9}$	$^{9}9^{10}$
`DIVF F10, F0, F6`	$^{4}5^{5}$	$^{18}13^{58}$	$^{58}14^{59}$
`ADDF F6, F8, F2`	$^{6}7^{7}$	$^{10}10^{12}$	$^{12}11^{13}$

Im folgenden wird für jede der jeweiligen Bearbeitungsphasen die **Reservierungstabelle** und die **Registerstatustabelle** gezeigt.

Beispiel (Tomasulo-Verfahren)

Phase 1

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Nein					
MULT2	Nein					

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	·····	*F30*
Q_i				LOAD1					
BUSY				Ja					

Beispiel (Tomasulo-Verfahren)

Phase 1

- BST:
  - Der erste Load-Befehl wird übergeben.
    (Nach Takt 1 ist die Übergabe-Stufe für diesen Befehl fertig.)
- RVT:
  - Keine der arithmetischen Funktionseinheiten ist Busy und daher sind alle übrigen Einträge in diesen Tabellen leer.
- RET:
  - LOAD1 ist unter *F6* in Q_i eingetragen und BUSY auf Ja gesetzt (weil der erste Load-Befehl (LF1) in der ersten (Issue) Bearbeitungsstufe auf der Funktionseinheit LOAD1 ist).

Beispiel (Tomasulo-Verfahren)

Phase 2

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Nein					
MULT2	Nein					

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i		LOAD2		LOAD1					
BUSY		Ja		Ja					

Beispiel (Tomasulo-Verfahren)

Phase 2

- BST:
  - Der zweite Load-Befehl wird übergeben.
  - Der erste Load-Befehl ist in der Execute-Stufe.
- RVT:
  - Die FP-Funktionseinheiten (ADD, MULT) nicht aktiv, deshalb sind die Einträge leer.
- RET:
  - Unter *F6*: Q_i = LOAD1; BUSY = Ja
    (weil der Wert von *F6* noch in Erzeugung durch LOAD1 ist).
  - Unter *F6*: Q_i = LOAD2; BUSY = Ja
    (weil LOAD2 mit der Erzeugung von *F2* beginnt).

Beispiel (Tomasulo-Verfahren)

Phase 3

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Ja	`MULTF`		(*F4*)	LOAD2	
MULT2	Nein					

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1	LOAD2		LOAD1					
BUSY	Ja	Ja		Ja					

Beispiel (Tomasulo-Verfahren)

Phase 3

- BST:
  - Der `MULTF`-Befehl wird übergeben.
    (Man weiß, woher die Operanden kommen, obwohl der Wert von *F2* nicht vorhanden ist.)
  - Der zweite Load-Befehl (LF2) ist in der EX-Stufe.
    (LF1 noch immer in der Execute-Stufe, weil diese bis zum Ende des Taktzyklus 5 dauert.)
- RVT:
  - Funktionseinheit MULT1 ist aktiv (Ausführung des `MULTF` Befehls).
  - Der erste Operand des `MULTF`-Befehls (*F2*) wird durch die FE LOAD2 erzeugt (deswegen LOAD2 im Feld Q_j eingetragen); der zweite Operand, d. h. *F4* ist vorhanden (irgendwo außerhalb des Codes produziert).
- RET:
  - *F6*, *F2* noch immer in Erzeugung.
  - Neuer Eintrag ist bei *F0*, welches in Erzeugung von MULT1 ist.

Beispiel (Tomasulo-Verfahren)

Phase 4

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Ja	`SUBF`			LOAD1	LOAD2
ADD2	Nein					
ADD3	Nein					
MULT1	Ja	`MULTF`		*(F4)*	LOAD2	
MULT2	Nein					

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1	LOAD2		LOAD1	ADD1				
BUSY	Ja	Ja		Ja	Ja				

Beispiel (Tomasulo-Verfahren)

Phase 4

- BST:
  - Der `SUBF`-Befehl wird an die LOAD2-Funktionseinheit übergeben.
  - Die beiden `LOADF`-Befehle sind noch immer in der Execute-Stufe (weil die Phase 4, d. h. die Übergabe des `SUBF`-Befehls, nach dem Taktzyklus 4 endet und die `LOADF`-Befehle erst nach Takt 5 bzw. 6 enden).
  - Der `MULTF`-Befehl wartet in der Issue-Stufe (solange bis *F2* vorhanden ist).

Beispiel (Tomasulo-Verfahren)

Phase 4

- RVT:
  - Funktionseinheit ADD1 ist Busy mit dem `SUBF`-Befehl, dessen Operanden *F6* bzw. *F2* sich in Bearbeitung von LOAD1 bzw. LOAD2 befinden.
  - MULT1 bleibt aktiv und wartet auf *F2*.
- RET:
  - *F6*, *F2*, *F0* noch immer in Bearbeitung
  - Neuer Eintrag ist ADD1 bei *F8* (aufgrund Issue des `ADDF`-Befehls)

Beispiel (Tomasulo-Verfahren)

Phase 5

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Ja	`SUBF`			LOAD1	LOAD2
ADD2	Nein					
ADD3	Nein					
MULT1	Ja	`MULTF`		*(F4)*	LOAD2	
MULT2	Ja	`DIVF`			MULT1	LOAD1

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1	LOAD2		LOAD1	ADD1	MULT2			
BUSY	Ja	Ja		Ja	Ja	Ja			

Beispiel (Tomasulo-Verfahren)

Phase 5

- BST:
  - Der `DIVF`-Befehl wird an MULT2 übergeben.
  - Die beiden Load-Befehle sind in der Execute-Stufe.
  - `MULTF`- und `SUBF`-Befehle warten in der Issue-Stufe (weil ihre Operanden nicht fertig sind, wodurch sie nicht in die Execute-Stufe vorrücken können).
- RVT:
  - MULT2 beginnt Busy zu sein mit dem neu übergebenen Befehl `DIVF`, dessen Operanden *F0*, *F6* sich in Bearbeitung von MULT1 bzw. LOAD1 befinden.
  - MULT1 und ADD1 bleiben aktiv.
- RET:
  - *F6*, *F2*, *F0* noch immer in Bearbeitung.
  - Neuer Eintrag ist Q_i = MULT2 und BUSY = Ja bei *F10* (weil der `DIVF`-Befehl übergeben wurde).

Beispiel (Tomasulo-Verfahren)

Phase 6

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Ja	SUBF	(LOAD1)			LOAD2
ADD2	Nein					
ADD3	Nein					
MULT1	Ja	MULTF		(*F4*)	LOAD2	
MULT2	Ja	DIVF		(LOAD1)	MULT1	

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1	LOAD2			ADD1	MULT2			
BUSY	Ja	Ja			Ja	Ja			

Beispiel (Tomasulo-Verfahren)

Phase 6

- BST:
  - Nach dem Taktzyklus 6 ist der Wert von *F6* vorhanden und wird auf dem Control Data Bus verschickt.
  - Der LF2-Befehl befindet sich noch immer in der Execute-Stufe.
  - Die `MULTF`-, `SUBF`- und `DIVF`-Befehle warten in der Issue-Stufe.
- RVT:
  - Weil LOAD1 den Wert für *F6* fertig erzeugt hat, tragen wir diese FE in Klammer an der Stelle V_j (ADD1-Zeile) und V_k (MULT2-Zeile) ein und LOAD1 wird dort aus den Spalten Q_j bzw. Q_k gelöscht.
  - MULT1 bleibt auch aktiv.
- RET:
  - Der Eintrag Q_i und BUSY unter *F6* wird gelöscht (weil *F6* in dieser Phase fertig geworden ist).

Beispiel (Tomasulo-Verfahren)

Phase 7

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Ja	`SUBF`	(LOAD1)	(LOAD2)		
ADD2	Ja	`ADDF`		(LOAD2)	ADD1	
ADD3	Nein					
MULT1	Ja	`MULTF`	(LOAD2)	(*F4*)		
MULT2	Ja	`DIVF`		(LOAD1)	MULT1	

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1			ADD2	ADD1	MULT2			
BUSY	Ja			Ja	Ja	Ja			

Beispiel (Tomasulo-Verfahren)

Phase 7

- BST:
  - Weil *F6* (von LF2) in der vorigen Phase freigegeben wurde, kann jetzt der `ADDF`-Befehl übergeben werden (es entsteht kein WAW-Hazard)
  - Im Taktzyklus 7 wird der Wert in *F2* produziert und danach auf CDB geschickt.
  - Die Befehle `MULTF`, `SUBF`, `DIVF` warten in der Issue-Stufe.
- RVT:
  - Weil LOAD2 den Wert in *F2* fertig erzeugt hat, wird diese FE geklammert an der Stelle von V_k (ADD1-Zeile) und V_j (MULT1-Zeile) eingetragen (und LOAD2 wird aus den entsprechenden Q-Spalten gelöscht).
  - ADD2 beginnt Busy zu sein (mit dem neu übergebenen Befehl `ADDF`, dessen Operand *F8* sich in Bearbeitung von ADD1 befindet und *F2* fertig ist (durch LOAD2)).

Beispiel (Tomasulo-Verfahren)

Phase 7

- RET:
  Unterschiede gegenüber der vorigen Phase 6:
  - Die Einträge Q_i, BUSY unter *F2* sind gelöscht, weil der Wert von *F2* nicht mehr in Bearbeitung ist.
  - Neu ist der Eintrag BUSY = Ja und Q_i = ADD2 bei *F6*, weil die ADD2-Einheit mit der Bearbeitung des `ADDF`-Befehls beginnt (dessen Ergebnis ist der Wert von *F6*).

Beispiel (Tomasulo-Verfahren)

Phase 8

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Ja	`SUBF`	(LOAD1)	(LOAD2)		
ADD2	Ja	`ADDF`		(LOAD2)	ADD1	
ADD3	Nein					
MULT1	Ja	`MULTF`	(LOAD2)	(*F4*)		
MULT2	Ja	`DIVF`		(LOAD1)	MULT1	

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1			ADD2	ADD1	MULT2			
BUSY	Ja			Ja	Ja	Ja			

Beispiel (Tomasulo-Verfahren)

Phase 8

- BST:
  - Weil der Wert von *F2* nun vorhanden ist, können die Befehle `MULTF` und `SUBF` in die Execute-Stufe vorrücken.
  - Die `DIVF`- und `ADDF`-Befehle sind in der Issue-Stufe angehalten (warten auf den Wert in *F0* bzw. *F8*).
- RVT, RET:
  - Keine Änderung gegenüber der vorigen Phase, weil keiner der Befehle beginnt issued zu sein oder seine Write Result-Stufe beendet.

Beispiel (Tomasulo-Verfahren)

Phase 9

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Ja	`ADDF`	(ADD1)	(LOAD2)		
ADD3	Nein					
MULT1	Ja	`MULTF`	(LOAD2)	(*F4*)		
MULT2	Ja	`DIVF`		(LOAD1)	MULT1	

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i	MULT1			ADD2		MULT2			
BUSY	Ja			Ja		Ja			

Beispiel (Tomasulo-Verfahren)

Phase 9

- BST:
  - Der `SUBF`-Befehl ist beendet (nach dem Taktzyklus 10).
  - Der `MULTF`-Befehl ist in der Execute-Stufe (dauert 10 Taktzyklen).
  - `DIVF`, `ADDF` bleiben in der Issue-Stufe.
- RVT:
  - Die Funktionseinheit ADD1 ist nicht mehr Busy (weil der `SUBF`-Befehl beendet ist).
  - In der ADD2-Zeile wird der Name der FE, die den Wert im Operandenregister *F8* erzeugt hat, in Klammer unter V_j eingetragen.
- RET:
  - Weil der Wert von *F8* nicht mehr in Bearbeitung ist, sind die Einträge unter *F8* frei geworden.

Beispiel (Tomasulo-Verfahren)

Phase 10

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Ja	`ADDF`	(ADD1)	(LOAD2)		
ADD3	Nein					
MULT1	Ja	`MULTF`	(LOAD2)	(*F4*)		
MULT2	Ja	`DIVF`		(LOAD1)	MULT1	

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	⋯	*F30*
Q_i	MULT1			ADD2		MULT2			
BUSY	Ja			Ja		Ja			

Beispiel (Tomasulo-Verfahren)

Phase 10

- BST:
  - Weil der Wert in *F8* vorhanden ist, kann der `ADDF`-Befehl in die Execute-Stufe übergehen.
  - Der `MULTF`-Befehl befindet sich noch immer in der Execute-Stufe.
  - Der `DIVF`-Befehl ist in der Issue-Stufe (weil er auf den Wert von *F0* wartet).
- RVT, RET:
  Wie in der vorigen Phase
  (weil kein Befehl issued wurde oder abgeschlossen wird).

Beispiel (Tomasulo-Verfahren)

Phase 11

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Ja	`MULTF`	(LOAD2)	(*F4*)		
MULT2	Ja	`DIVF`		(LOAD1)	MULT1	

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	⋯	*F30*
Q_i	MULT1					MULT2			
BUSY	Ja					Ja			

Beispiel (Tomasulo-Verfahren)

Phase 11

- BST:
  - Der `ADDF`-Befehl ist bearbeitet worden (nach insgesamt 13 Taktzyklen).
  - Der `MULTF`-Befehl ist in der Execute-Stufe.
  - Der `DIVF`-Befehl ist in der Issue-Stufe.
- RVT:
  - Die FE ADD2 hat den `ADDF`-Befehl bearbeitet, daher ist sie nicht mehr Busy.
  - Die MULT1, MULT2 Einheiten sind noch immer Busy (MULT1 bearbeitet den `MULTF`-Befehl, der sich in der Execution-Stufe befindet und MULT2 ist mit dem `DIVF`-Befehl in der Issue-Stufe).
- RET:
  - Die Einträge in der Spalte unter *F6* sind aufgrund der Beendigung des `ADDF`-Befehls frei geworden.

Beispiel (Tomasulo-Verfahren)

Phase 12

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Nein					
MULT2	Ja	`DIVF`	(MULT1)	(LOAD1)		

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i						MULT2			
BUSY						Ja			

Beispiel (Tomasulo-Verfahren)

Phase 12

- BST:
  - Die Ausführung des `MULTF`-Befehls ist abgeschlossen.
  - Der `DIVF`-Befehl wartet in der Issue-Phase.
    (Der Wert von *F0* wird für diesen Befehl erst in der nachfolgenden Phase über den CDB verschickt, und erst danach wird er in die EX-Stufe vorrücken können.)
- RST:
  - Die MULT1-Funktionseinheit ist nicht mehr Busy (weil der `MULTF`-Befehl fertig ist).
  - In der MULT2-Zeile wird der Name der FE, die den Wert im Operandenregister *F0* erzeugt hat, d. h. MULT1, geklammert unter V_j eingetragen.
- RET:
  - Die Einträge unter *F0* sind gelöscht worden (weil die Berechnung des Wertes in *F0* beendet ist).

Beispiel (Tomasulo-Verfahren)

Phase 13

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Nein					
MULT2	Ja	`DIVF`	(MULT1)	(LOAD1)		

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i						MULT2			
BUSY						Ja			

Beispiel (Tomasulo-Verfahren)

Phase 13

- BST:
  - Der `DIVF`-Befehl ist in der EX-Stufe
    (bleibt hier laut Angabe 40 Taktzyklen)
- RST, RET:
  - Unverändert im Vergleich zur Phase 12
    (kein Befehl issued oder beendet geworden)

Beispiel (Tomasulo-Verfahren)

Phase 14

Reservierungstabellen (RVT)

	BUSY	OP	V_j	V_k	Q_j	Q_k
ADD1	Nein					
ADD2	Nein					
ADD3	Nein					
MULT1	Nein					
MULT2	Nein					

Registerstatustabelle (RET)

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	$\cdots$	*F30*
Q_i									
BUSY									

Beispiel (Tomasulo-Verfahren)

Phase 14

- BST:
  - Als letzter wird der `DIVF`-Befehl beendet
    (nach insgesamt 59 Taktzyklen)
- RVT:
  - Auch die letzte aktive FE (MULT2) beendet ihre Tätigkeit.
- RET:
  - Weil der Wert von *F10* nicht mehr in Bearbeitung ist, sind die Einträge unter *F10* frei geworden.

Beispiel (Tomasulo-Verfahren)

Zusammenfassung des Beispiels

Der Tomasulo-Algorithmus hat die Bearbeitung unseres Beispiel-Codes nach 59 Taktzyklen beendet.

Es ist dabei zu folgenden **Befehlsumordnungen** gekommen:

- Der `SUBF`-Befehl wurde vor dem `MULTF`-Befehl ausgeführt.
  (`SUBF` endet nach **10** Taktzyklen in der **Phase 9** und `MULTF` nach **18** Taktzyklen in der Phase **12**)
- Der `ADDF`-Befehl wurde vor dem `MULTF`-Befehl sowie auch vor dem `DIVF`-Befehl beendet.
  (`ADDF` endet in der **Phase 11** (Taktzyklus **13**), die Befehle `MULTF` bzw. `DIVF` enden in der **Phase 12** (Taktzyklus **18**) bzw. in der **Phase 14** (Taktzyklus **59**).)

Beispiel (Beseitigung von **WAW**-Hazards (in der Issue-Stufe))

```
LF F6, 34(R2)
ADDF F6, F8, F2
```

- Der Hazard

  In unserer Befehlsfolge (siehe Befehlsstatustabelle) haben wir die Befehle `LF` und `ADDF` übergeben (in der Phase 1 bzw. Phase 7). Die Befehle laufen parallel, obwohl es einen **WAW**-Hazard bezüglich *F6* geben könnte, d. h. der `ADDF`-Befehl **könnte vor** dem Befehl `LF` schreiben und dadurch **könnten** die **nach** `LF` **folgenden Befehle** (`DIVF`) **statt dem alten Wert** (Ergebnis von `LF`) **den neuen Wert** (Ergebnis von `ADDF`) benutzen, was **falsch** wäre.

Beispiel (Beseitigung von **WAW**-Hazards (in der Issue-Stufe))

- Die Entfernung des möglichen Hazards

  WAW-Hazards werden in der Issue-Phase dadurch vermieden, dass in der Registerstatustabelle pro Register nur Platz für einen Eintrag (d. h. für den Namen einer Funktionseinheit) ist. Wenn ein anderer Befehl mit dem gleichen Zielregister (in unserem Beispiel ist es `ADDF` mit dem *F6* Register) in die Issue-Stufe übergehen möchte, **müsste der Platz Q_i in der Befehlsstatustabelle unter *F6* doppelt belegt werden**.
  Das geht nach dem Tomasulo-Algorithmus aber nicht. Deshalb muss der zweite Befehl (`ADDF`) warten, bis der erste Befehl (`LF`) fertig ist (d. h. solange BUSY unter *F6* auf **Ja** gesetzt ist).

Beispiel (Beseitigung von **WAW**-Hazards)

- Die Vermeidung des **WAW**-Hazards illustrieren die Befehlsstatus- und Registerstatustabellen:

  Der Tomasulo-Algorithmus erlaubt dem zweiten Befehl, welcher in *F6* schreiben soll (`ADDF`) nicht, früher übergeben zu werden (d. h. in den Platz Q_i unter *F6* den Namen seiner Funktionseinheit **ADD2** einzutragen), bevor der erste Befehl (`LF`) abgeschlossen ist (d. h. bevor dieser Platz in der Registerstatustabelle frei geworden ist (Phase 6)).

  Bevor aber der zweite Befehl in die Issue-Stufe übergeht und in die Befehlsstatustabelle der Name seiner FE (**ADD2**) eingetragen wird (Phase 7), ist das Ergebnis des `LF`-Befehls im Register V_k, das zur Funktionseinheit **ADD2** gehört, abgespeichert.

  Aufgrund der Vermeidung des WAW-Hazards kann auch der `DIVF`-Befehl den alten Wert von *F6* (Ergebnis des `LF`-Befehls) benutzen.

Beispiel (Beseitigung von **WAR**-Hazards (in der Issue-Stufe))

```
LF F6, 34(R2)
DIVF F10, F0, F6
ADDF F6, F8, F2
```

- Der Hazard

  In unserer Befehlsfolge (siehe Befehlsstatustabelle) haben wir die Befehle `DIVF` und `ADDF` übergeben (in der Phase 5 bzw. Phase 7) und parallel laufen lassen, obwohl es einen **WAR**-Hazard bezüglich *F6* geben kann. Wir haben also die Bearbeitung von `ADDF` gestartet, bevor der `DIVF`-Befehl fertig ist.

  Das könnte einen **WAR**-Hazard zur Folge haben, d. h. der `DIVF`-Befehl **könnte** einen falschen („neuen“, von `ADDF` erzeugten) Wert von *F6* einlesen.

Beispiel (Beseitigung von **WAR**-Hazards (in der Issue-Stufe))

- Die Entfernung des möglichen Hazards

  Wir nehmen an, dass der **Load**-Befehl noch nicht abgeschlossen ist. Dann wird in die Spalte Q_k der Reservierungstabelle der Funktionseinheit **MULT2** (die den Befehl `DIVF` überwacht) der Name der auszuführenden FE (d. h. **LOAD1**) eingetragen. Nach der Phase 6 ist die FE **LOAD1** fertig und sie schickt dann das Ergebnis über den **CDB** zu den Einheiten, die darauf warten.

  In der **Reservierungstabelle** der Funktionseinheit **MULT2** wird dann der Eintrag **LOAD1** bei Q_k gelöscht und **geklammert in die Spalte V_k eingetragen**. (Das heißt, der über den **CDB** übertragene Wert wird in V_k eingetragen.)

Beispiel (Beseitigung von **WAR**-Hazards (in der Issue-Stufe))

- (Fortsetzung)

  So ist der Wert von *F6* im Register V_k gespeichert und der `DIVF`-Befehl wird mit diesem Wert seine Bearbeitung (beginnend mit der Issue-Stufe) durchführen.
  Die Bezeichnung *F6* tritt im `DIVF`-Befehl nicht mehr auf, aber dessen Wert ist in V_k gespeichert oder in Q_k mit der FE LOAD1 verknüpft.

  Auch wenn der darauffolgende `ADDF`-Befehl dann parallel laufen würde, **kann es also nicht dazu kommen**, dass `DIVF` den **falschen** Wert von *F6* aus dem `ADDF`-Befehl liest. Dadurch wird der mögliche **WAR**-Hazard der Befehle `DIVF` und `ADDF` in der Issue-Stufe **vermieden**.

Beispiel (Beseitigung von **WAR**-Hazards (in der Issue-Stufe))

- Die Vermeidung des **WAR**-Hazards illustrieren die Befehlsstatus- und Registerstatustabellen:

  Der Wert von *F6* ist (nach der Phase 6) über CDB an die wartende **MULT2** verschickt.
  Der `ADDF`-Befehl schreibt den neuen Wert von *F6* in der Phase 11. Die Execution-Stufe des `DIVF`-Befehls (wo der Wert von *F6* gebraucht wird) startet erst in der Phase 13. Also könnte es passieren, dass `DIVF` den **neuen** Wert von *F6* benutzt (d. h. das Ergebnis von `ADDF`), was der **WAR**-Hazard wäre.

  Das geschieht aber nicht, weil der alte Wert von *F6* (Ergebnis von `LF`-Befehl) **bereits** (unter dem Namen **LOAD1**) **im Register** V_k **gespeichert ist** (siehe die Zeile **MULT2** in der Reservierungstabelle Phase 6).

Beispiel (Beseitigung von **RAW**-Hazards (in der Execute-Stufe))

```
LF F2, 45(R3)
MULTF F0, F2, F4
```

- Der Hazard

  In unserer Befehlsfolge (siehe die Befehlsstatustabelle) haben wir die Befehle `LF` und `MULTF` übergeben (in der Phase 2 bzw. Phase 3) und parallel laufen lassen, obwohl es einen **RAW**-Hazard bezüglich *F2* geben könnte. (Wir haben also die Bearbeitung von `MULTF` gestartet, bevor der `LF`-Befehl fertig ist und das **könnte** einen **RAW**-Hazard bezüglich *F2* verursachen (d. h. `MULTF` **könnte einen falschen („alten“) Wert von** *F2* **einlesen**).)

Beispiel (Beseitigung von **RAW**-Hazards (in der Execute-Stufe))

- Die Entfernung des möglichen Hazards

  Wir nehmen an, dass der Wert von *F2* noch nicht vorhanden ist, d. h. der `LF`-Befehl ist noch nicht fertig und der `MULTF`-Befehl möchte anfangen.
  Nach dem Tomasulo-Verfahren wird der Name der Funktionseinheit, welche den Wert von *F2* zur Verfügung stellt (in unserem Falle LOAD2), in die Spalte Q_j der Reservierungstabelle (in die Zeile, die der FE MULT1 entspricht) eingetragen.

Beispiel (Beseitigung von **RAW**-Hazards (in der Execute-Stufe))

- (Fortsetzung)

  Der `MULTF`-Befehl muss solange in der Issue-Stufe warten, bis alle Argumente verfügbar sind (also in unserem Falle bis der Wert von *F4* vorhanden ist und LOAD2 fertig geworden ist und das Ergebnis über den CDB verbreitet). Erst danach kann der `MULTF`-Befehl in die Execute-Stufe übergehen.
  Durch den Vorgang des Wartens auf das Ergebnis stellen wir sicher, dass `MULTF` erst lesen wird, nachdem `LF` schon geschrieben hat.
- Diese Vermeidung des RAW-Hazards illustriert die Befehlsstatustabelle:
  Der `MULTF`-Befehl geht in der 8. Phase des Algorithmus in die Execute-Stufe über, also erst nachdem der Wert von *F2* fertig geworden ist (Phase 7).

Tomasulo

Kann Speedup bringen, aber zusätzliche Hardware nötig: RS, CDB

$\rightarrow$ **auch hier größtes Problem: Branches**

Control Hazards

Control Hazards

Konflikt: Ob ein Branch verzweigt oder nicht (wenn dies erst nach einigen Pipelinestufen festgestellt werden kann)

Branch verzweigt: Branch Taken
Branch verzweigt nicht: Branch Untaken

Bis das Ergebnis feststeht, sind aber die nachfolgenden Befehle in der Pipeline.

Wenn Branch Taken
$\rightarrow$ **Diese Befehle falsch**
und sie können nicht ausgeführt werden (da sie Register verändern könnten, die dann von den Befehlen **im Taken Branch** verwendet werden)

Einfachste Lösung:
Stalls einfügen, sobald die Pipeline erkennt, dass ein Branch-Befehl in der Pipeline ist.

Control Hazards

	1	2	3	4	5	6	7	8	9	10
Branch	IF	ID	EX	MEM	WB					
Befehl 02		IF	**Stall**	**Stall**	IF	ID	EX	MEM	WB	
Befehl 03						IF	ID	EX	MEM	WB

Wenn nach ID festgestellt wird, dass ein **Branch**-Befehl in der Pipeline ist
→ 2 Stalls (weil erst nach der MEM-Stufe klar ist, **ob** und wohin gesprungen wird).

Nach den 2 Stalls: IF (für den Befehl nach Branch) wiederholt (**da bei ersten Mal der falsche Befehl geholt worden sein könnte**)

→ Delay **3** Taktzyklen

Control Hazards

Verbesserte Pipeline zur Ausführung der Branch-Befehle

Beim DLX:

- Branch erkannt: In der `ID`-Stufe
- Ob und wohin gesprungen wird: In `MEM` klar

Verbesserte Pipeline

Die Berechnungen von der `EX`- und `MEM`-Stufe in die `ID`-Stufe vorziehen

- Also hat in der verbesserten Pipeline der Branch-Befehl nur 2 Stufen:
  `IF` `ID`
- Die **neue** `ID`-Stufe des Branch-Befehls
  beinhaltet die Stufen `ID`, `EX` und `MEM` des ursprünglichen Branch-Befehls.
  (In der Stufe `WB` war schon beim ursprünglichen Branch-Befehl nichts zu tun.)

Control Hazards

- ID
  - ID
    - $A \leftarrow REGS[IR[6 \ldots 10]]$
    - (Inhalt des Source-Registers wird nach A gespeichert)
    - $IMM \leftarrow IR[16 \ldots 31]$
  - EX
    - $ALUOUTPUT \leftarrow NPC + IMM$ *
    - (Die mögliche Sprungadresse wird berechnet)
    - $COND \leftarrow A$ OP 0 **
    - (Ob tatsächlich gesprungen wird, wird in $COND$ berechnet; $COND$ (ein boolescher Wert) enthält als Ergebnis den Vergleich des Registers A mit 0)
  - MEM
    - IF($COND$) $PC \leftarrow ALUOUTPUT$ ***
    - (Wenn die Bedingung für den Branch erfüllt ist, dann wird der PC mit der vorhin berechneten Sprungadresse geladen; NPC (aus der IF-Stufe) bleibt als neuer PC für den Fall, dass nicht gesprungen wird)

Control Hazards

- In der `ID`-Stufe wird das Register *RS* (*A*) gelesen, das anzeigt, ob gesprungen wird
  $\rightarrow$ Die Operationen $**$, $***$ können in `ID` vorgezogen werden
- Auch Offset zum *PC* (*IMM*) wird in `ID` gelesen
  $\rightarrow$ $*$ kann in der `ID`-Stufe ausgeführt werden
  $\rightarrow$ Zusätzliche Hardware (ein Addierer) wird für die Adressberechnung $*$ benötigt.

Static Scheduling

Durch Hardwareerweiterungen dem Compiler die Möglichkeit zu geben, Code umzuordnen, um Stalls zu meiden.

Vier Methoden für die auf Branch bezogen **verbesserte Pipeline**:

- Freeze Pipeline
- Prediction (Branch Taken/Untaken)
- Delayed Branch
- Cancelling Branch.

Freeze Pipeline

Alle Befehle nach der Verzweigung werden angehalten, bis das Verzweigeziel bekannt ist (nach der ID-Stufe des Branch-Befehls).

	1	2	3	4	5	6	7	8
Branch	IF	ID						
Befehl 01		IF	IF	ID	EX	MEM	WB	
Befehl 02			Stall	IF	ID	EX	MEM	WB

- Befehl 01 ist bereits in der Pipeline (IF), wenn das Ziel der Verzweigung bekannt ist (nach der ID-Stufe des Branch-Befehls).
- Danach muss der richtige Befehl geholt werden (IF):
  - Ein neuer Befehl, wenn gesprungen wird.
  - Wenn kein Sprung, muss der alte Befehl (IF) wiederholt werden.
- Die darauf folgenden Befehle (ab Befehl 02) werden mit einem Stall fortgesetzt.

Freeze Pipeline

Bemerkung

Was die Befehlsabarbeitungsstufen anbelangt, wäre es das gleiche, anstelle des verworfenen `IF` einen Stall anzugeben.
Diese Lösung scheint einfacher zu sein, aber am Anfang von Takt 2 weiß man noch nicht, dass es sich beim Befehl davor um einen Branch handelt.

Prediction

Predict Not Taken

Für jeden **Branch** wird angenommen, dass er **nicht verzweigt**, d. h. die Abarbeitung wird beim nächsten Befehl (Fall-Through) fortgesetzt.

Situation 1: **Der Branch verzweigt nicht**

→ Hier wird der richtige Befehl nach dem Branch in die Pipeline geholt → keine Verzögerung.

Branch (Untaken)	IF	ID				
Befehl 01 (Fall-Through)		IF	ID	EX	MEM	WB
Befehl 02			IF	ID	EX	MEM	WB

Prediction

Predict Not Taken

Situation 2: **Der Branch verzweigt**

→ Das Verhalten ist ident zu Freeze, d. h. im Takt 2 wurde ein falscher Befehl (Fall-Through) geholt; dieser muss nach der ID-Stufe des Branch-Befehls durch den richtigen Befehl (Target) ersetzt werden → ein Verzögerungsschritt

Branch (Taken)	IF	ID						
Befehl 01		IF	IF	ID	EX	MEM	WB	
Befehl 02			Stall	IF	ID	EX	MEM	WB

Prediction

Predict Taken

Hier wird angenommen: **Jeder Branch verzweigt**

Situation 1: **Der Branch verzweigt nicht**
(also ist die Vorhersage Branch Taken falsch).

Branch	IF	ID					
(Untaken)							
Befehl 01		IF	Idle	Idle	Idle	Idle	
(Target 01)							
Befehl 02			IF	ID	EX	MEM	WB
(Fall-Through)							

Befehl 01 war falsch, also wird nach der Berechnung des Verzweigungsziels (nach der ID-Stufe des Branch-Befehls) Befehl 01 gestoppt und gleich (als Befehl 02) der Fall-Through geholt.

Prediction

Predict Taken

Situation 2:
Die Vorhersage stimmt, also **verzweigt der Branch** und als nächster Befehl wird der Zielbefehl in die Pipeline geholt (also haben wir richtig spekuliert). Die Zieladresse dieses Befehls ist aber erst nach der `ID`-Stufe des Branch-Befehls bekannt $\rightarrow$ wir müssen das Holen dieses Befehls um einen Takt verzögern (Stall)

Branch (Taken)	`IF`	`ID`					
Befehl 01 (Target 01)		Stall	`IF`	`ID`	`EX`	`MEM`	`WB`

Prediction

Fazit (Prediction)

Bei Predict Taken haben wir in beiden Situationen einen Verzögerungsschritt, d. h. diese Strategie bringt keinen Gewinn.

Bei Predict Not Taken ergibt sich in der Situation „Branch verzweigt“ ebenfalls ein Verzögerungsschritt, aber keine Verzögerung in der Situation „Branch verzweigt nicht“.

Delayed Branch

Hier muss der Compiler geeignete Befehle für den sogenannten **Delay Slot** finden.
Also: Ein Befehl wird hinter dem Branch platziert, der unabhängig vom Branch bearbeitet werden kann → Stall vermeiden
(Die Ursache des Stalls ist, dass man erst nach der ID-Stufe des Branch-Befehls weiß, ob und wohin gesprungen wird.)

Beispiel:

```
        ADD   R1, R2, R3
LOOP:   SUB   R4, R5, R6
        SUBI  R10, R10, #1
        BEQZ  R10, LOOP
        ADD   R5, R8, R9
        OR    R7, R8, R9
        JR    R31
```

Delayed Branch

Der Compiler hat **3 Möglichkeiten**, den **Delay Slot** zu füllen. In diesem Beispiel muss der Code umgeordnet werden (sonst kommt es zu einem falsch ausgeführten Programm).

- 1. Möglichkeit: **Einen Befehl von vor der Schleife** in den Delay Slot stellen:

```
LOOP:  SUB   R4, R5, R6
       SUBI  R10, R10, #1
       BEQZ  R10, LOOP
       ADD   R1, R2, R3      → Delay Slot filled from before
       ADD   R5, R8, R9
       OR    R7, R8, R9
       JR    R31
```

Das ist nur möglich, wenn es einen parallelen Befehl vor der Schleife gibt. Existiert ein solcher, wird jeder Stall des vorhergehenden Branch-Befehls eliminiert, doch bei mehrmaligem Durchlaufen der Schleife wird der Befehl mehrmals ausgeführt.

(Die mehrmalige Ausführung ist bei obigem `ADD`-Befehl kein Problem, ein Befehl wie z. B. `ADDI R1, R1, #1` wäre allerdings nicht für die Füllung des Delay Slots geeignet.)

- 2. Möglichkeit: Der Compiler nimmt an, dass **Branch Taken** eintritt und **stellt einen Befehl von der Sprungadresse in den Delay Slot**.

```
        ADD   R1, R2, R3
LOOP:   SUB   R4, R5, R6
        SUBI  R10, R10, #1
        BEQZ  R10, LOOP + 4
        SUB   R4, R5, R6      → Delay Slot filled from Target
        ADD   R5, R8, R9
        OR    R7, R8, R9
        JR    R31
```

Wieder muss dieser Befehl parallel und problemlos mehrmals ausführbar sein. Der Befehl sollte in den Delay Slot kopiert werden (weil die Sprungadresse auch von einer anderen Stelle angesprungen werden kann).

Verzweigt der Branch nicht, so wird zwar ein unnützer Befehl berechnet → aber **kein Problem**, wenn er auch zu den folgenden Befehlen parallel ist.

- 3. Bleibt die Möglichkeit, den **Delay Slot** mit Befehlen aus dem **Fall-Through** zu besetzen:

```
        ADD   R1, R2, R3
LOOP:   SUB   R4, R5, R6
        SUBI  R10, R10, #1
        BEQZ  R10, LOOP
        OR    R7, R8, R9      → Delay Slot filled from Fall-Through
        ADD   R5, R8, R9
        JR    R31
```

Bei der dritten Möglichkeit wurde der `OR`-Befehl (statt des `ADD`-Befehls) in den Delay Slot vorgezogen.

Weil bei einem **Taken Branch**
`LOOP: SUB` *R4*, *R5*, *R6*
das Register *R5* verwendet wird,
ist der `ADD`-Befehl `ADD` *R5*, *R8*, *R9* nicht für eine Platzierung im Delay Slot geeignet (nicht parallel zur Schleife).

$\rightarrow$ Also hätten wir eine falsche Ausführung des Programms, wenn wir keinen parallelen Befehl gefunden hätten. (In unserem Fall haben wir den parallelen `OR`-Befehl gefunden.)

→ Die Ausführung (bei **Branch Taken**):

Branch	`BEQZ` *R10*`, LOOP`	IF	ID						
(Taken)									
Delay Slot	`OR` *R7*`,` *R8*`,` *R9*		IF	ID	EX	MEM	WB		
Loop	`SUB` *R4*`,` *R5*`,` *R6*			IF	ID	EX	MEM	WB	

(Nach der `ID`-Stufe des Branch-Befehls ist die Sprungadresse bekannt, von welcher der Befehl geholt wird.)
Während der `ID`-Stufe des Branch-Befehls wird die `IF`-Stufe des parallelen `OR`-Befehls ausgeführt. Wenn die Pipeline darauf vorbereitet ist, dass der Befehl im Delay Slot immer ausgeführt wird (also auch dann, wenn wir keinen geeigneten parallelen Befehl finden), dann könnten wir das Programmsegment gar nicht korrekt auf der DLX ausführen.

→ Es gibt ein anderes Verfahren namens Cancelling Branch, wo nicht automatisch mit dem Delay Slot nach dem Branch fortgesetzt wird.

Cancelling Branch

Variante **Cancel if Untaken**

In dieser Variante wird bei Branch Untaken der Befehl im Delay Slot eingefroren (cancelled) und der nächste korrekte Befehl wird in die Pipeline geholt.

→ Hier wird also der Befehl im Delay Slot nur dann ausgeführt, wenn der Branch in die vermutete Richtung verzweigt.

Branch (**Taken**)	IF	ID					
Delay Slot		IF	ID	EX	MEM	WB	
Target 01			IF	ID	EX	MEM	WB

Branch (**Untaken**)	IF	ID					
Delay Slot		IF	Idle	Idle	Idle	Idle	
Fall-Through 01			IF	ID	EX	MEM	WB

Cancelling Branch

Variante **Cancel if Taken**

Hier wird der Befehl im Delay Slot nur bei Branch Untaken ausgeführt. Bei Branch Taken wird der Befehl im Delay Slot eingefroren (cancelled) und der nächste korrekte Befehl wird in die Pipeline geholt.

Branch	IF	ID					
(**Taken**)							
Delay Slot		IF	Idle	Idle	Idle	Idle	
Target 01			IF	ID	EX	MEM	WB

Branch	IF	ID					
(**Untaken**)							
Delay Slot		IF	ID	EX	MEM	WB	
Fall-Through 01			IF	ID	EX	MEM	WB

Cancelling Branch

Welche Variante von Cancelling Branch soll für **unser Beispiel** angewendet werden?

	ADD	*R1*, *R2*, *R3*	$R1 \leftarrow R2 + R3$
LOOP:	SUB	*R4*, *R5*, *R6*	$R4 \leftarrow R5 - R6$
	SUBI	*R10*, *R10*, #1	$R10 \leftarrow R10 - 1$
	BEQZ	*R10*, LOOP	Wenn $R10 = 0$ dann Sprung nach LOOP
	ADD	*R5*, *R8*, *R9*	$R5 \leftarrow R8 + R9$ (Delay Slot)
	OR	*R7*, *R8*, *R9*	$R7 \leftarrow R8$ OR $R9$
	JR	*R31*	Sprung nach Adresse in *R31*

Cancelling Branch

Der `ADD`-Befehl (der im Delay Slot steht) soll dann ausgeführt werden, wenn nicht (zur Marke `LOOP`) gesprungen wird, d. h. wenn Branch Untaken stattfindet. Umgekehrt soll der `ADD`-Befehl im Delay Slot eingefroren (cancelled) werden, wenn gesprungen wird, d. h. wenn Branch Taken passiert.

$\rightarrow$ Das geschieht bei der Variante **Cancel if Taken**. Also bräuchten wir für die korrekte Bearbeitung unseres Beispiels eine Pipeline mit **Cancel if Taken**.

Eliminierung von Branch Delays

Andere Technik zur Eliminierung von Branch Delays:
Eliminierung der Branches selbst.

→ **Loop Unrolling**
Versucht, die Schleifen durch Vervielfältigung des Codes im Schleifenkörper aufzulösen

Bei Maschinen mit großer Anzahl von Pipelinestufen ist ein **Branch Delay** üblicherweise größer als ein Taktzyklus.

→ Es ist schwer für den Compiler, genug parallele Befehle für die (jetzt) mehreren Delay Slots zu finden.

→ Viele Maschinen verzichten heute auf diese Technik und versuchen, **das Branchverhalten zur Laufzeit vorherzusagen**.
→ **Dynamic Branch Prediction**

Dynamic Branch Prediction

2 Dinge könnte man vorhersagen: **Die Branchbedingung** und die **Sprungadresse**.

Mittel zur Vorhersage der **Branchbedingung**:
Branch Prediction Buffer (Branch History Table)

- besteht in der einfachsten Form aus einem einzigen Bit
- **gibt an, ob der letzte Branch verzweigte oder nicht**

Prädiktoren

Prädiktoren

(zur Vorhersage der Branchbedingung)

- Unkorrelierende Prädiktoren
- Korrelierende Prädiktoren

Unkorrelierende Prädiktoren (Buffers)

1-Bit-Buffer

Dieser gibt an, ob der letzte Branch verzweigte oder nicht.

Beispiel (Programmsegment mit zwei Branches, b_1, b_2)

```
                ADDI  R1, R0, #1      ; x = 1
                ADDI  R2, R0, #0      ; y = 0
b1      up:     BNEZ  R1, next        ; b1 Taken, if x ≠ 0
                ADDI  R2, R2, #1      ; y++
b2      next:   BNEZ  R1, down        ; b2 Taken, if x ≠ 0
                ADDI  R2, R2, #1      ; y++
        down:   XORI  R1, R1, #1      ; flip x
                J     up
```

Unkorrelierende Prädiktoren (Buffers)

Für dieses Beispiel ergibt sich die folgende Vorhersagetabelle
(1-Bit (unkorrelierender) Prädiktor; P: Vorhersage, A: Verhalten)

		b_{1P}	b_{1A}	b_{2P}	b_{2A}
Iteration	$j=1$	0	1	0	1
	$j=2$	1	0	1	0
	$j=3$	0	1	0	1
	$j=4$	1	0	1	0

Die Vorhersage in der Iteration $j+1$ entspricht dem Verhalten in der vorigen Iteration j ($\swarrow$).

$b_{1A} = b_{2A}$, weil beide Branches davon abhängig sind, ob $R1 = 0$ (dann $b_{1A} = b_{2A} = 0$) oder $R1 = 1$ (dann $b_{1A} = b_{2A} = 1$).

Für $j = 1, 3$ ist $R1 = 1 \rightarrow b_{1A} = b_{2A}$ sind Taken.
Für $j = 2, 4$ ist $R1 = 0 \rightarrow b_{1A} = b_{2A}$ sind Untaken.

Unkorrelierende Prädiktoren (Buffers)

Das Gesamtergebnis:
In jeder Iteration sind die Vorhersagen inkorrekt.

Wenn man nicht **nur** den letzten Branch, sondern auch **vorige** mitprotokolliert, benutzt man n-Bit-Buffers.

Die einfachste Lösung: 2-Bit-Buffer.

Technische Realisierung des 2-Bit-Buffers

Untaken mit 0 kodiert.
Taken mit 1 kodiert.

$\rightarrow$ 4 Zustände des 2-Bit-Buffers:

0 0
0 1
1 0
1 1

Die Umwandlung für die Vorhersage (dezimal betrachtet):
Für $z \geq 2$ wird Taken, für $z < 2$ wird Untaken vorhergesagt.

Technische Realisierung des 2-Bit-Buffers

Die Vorhersage (unkorrelierend) wird folgendermaßen generiert:

	Prädiktor	Verhalten
Iter j	z	Taken
Iter $j+1$	$z \leftarrow z+1$	

(wenn $z = 3$, dann bleibt z unverändert auf 3)

Iter j	z	Untaken
Iter $j+1$	$z \leftarrow z-1$	

(wenn $z = 0$, dann bleibt z unverändert auf 0)

Also: Die Vorhersage ist in der Iteration j nicht nur vom Verhalten in der Iteration $j-1$ abhängig, sondern auch vom Wert z.
($z = 0, 1$: Untaken, $z = 2, 3$: Taken.)

Technische Realisierung des 2-Bit-Buffers

Hier wird eine längere Vorgeschichte protokolliert:
Wenn z. B. $z = 3$, dann war das Verhalten in der letzten Zeit öfter Taken als Untaken.

Also: Es muss mindestens **2-mal** Untaken sein, damit die Vorhersage auf Untaken wechselt (beim 1-Bit-Prädiktor hat 1-mal Untaken gereicht).

Korrelierende Prädiktoren

Wenn das Verhalten eines Branches auch von der Vorgeschichte eines anderen Branches abhängt, verwendet man korrelierende Prädiktoren.

- Der einfachste korrelierende Prädiktor besteht aus zwei Bits.
  Dieser Prädiktor wird in Form x/y ($x, y \in \{0, 1\}$) dargestellt, wobei für die Vorhersage x oder y genommen („fett“ geschrieben) wird.

Korrelierende Prädiktoren

- Die Konstruktionsregel für die Einträge der Prädiktionstabelle
  - Der neue Eintrag:
    An der Stelle des gewählten Bits („fette“ Stelle) wird das **Verhalten** des Branches **in seiner vorigen Iteration** eingetragen
  - Die Wahl des Vorhersagewertes (aus x/y):
    Abhängig vom **Verhaltenswert des globalen Branches** (beim 1-Bit-Prädiktor ist das der benachbarte Branch)
    - Wenn dieser Wert gleich **0** ist:
      Das linke Bit (x) wird für die Vorhersage ausgewählt (d. h. „fett“ geschrieben)
    - Wenn dieser Wert gleich **1** ist:
      Das rechte Bit (y) wird als Vorhersage genommen.

Korrelierende Prädiktoren

Die Vorhersagetabelle für das obige Beispiel mit einem korrelierenden 1-Bit-Prädiktor (A: Verhalten, P: Vorhersage („**fett**“ geschrieben))

	b_{1P}	b_{1A}	b_{2P}	b_{2A}
$j = 1$	$\mathbf{0}/0$	1	$0/\mathbf{0}$	1
$j = 2$	$1/\mathbf{0}$	0	$\mathbf{0}/1$	0
$j = 3$	$\mathbf{1}/0$	1	$\boxed{0/\mathbf{1}}$	1
$j = 4$	$1/\mathbf{0}$	0	$\mathbf{0}/1$	0

Das Verhalten (A) beider Branches ist beim korrelierenden Prädiktor das gleiche wie beim unkorrelierenden

Korrelierende Prädiktoren

1. Zeile:
   - $b_{1P} = \mathbf{0}/0$ (Startwert für b_{1P} ist 0/0).
     Weil der letzte Branch (irgendwo vorher) Untaken war, wird das linke Bit für die Vorhersage ausgewählt.
   - $b_{1A} = 1$, weil der erste Branch Taken ist.
   - $b_{2P} = 0/\mathbf{0}$, weil der vorherige Branch Taken war ($b_{1A} = 1$), wird im Start-Eintrag 0/0 für b_2 der rechte Wert ausgewählt.
   - $b_{2A} = 1$, im zweiten Branch wird gesprungen.

Korrelierende Prädiktoren

2. Zeile:

- $b_{1P} = \mathbf{1}/\underline{0}$, an die „fette“ Stelle in b_{1P} (in der ersten Zeile, d. h. links) kommt der Wert des Verhaltens $b_{1A} = 1$ aus der ersten Zeile. Die Wahl des rechten Bits für die Vorhersage erfolgt aufgrund von $b_{2A} = 1$ (d. h. der letzte globale Branch aus Sicht von b_1 (also b_2) war Taken).
- $b_{1A} = 0$.
- $b_{2P} = \mathbf{0}/1$ an die „fette“ Stelle aus der 1. Zeile kommt 1 und der linke Wert wird für die Vorhersage gewählt (weil $b_{1A} = 0$).
- $b_{2A} = 0$ (in b_2 wird in der zweiten Iteration nicht gesprungen).

Korrelierende Prädiktoren

- Das Gesamtergebnis:
  Der korrelierende Prädiktor liefert nur für die zwei ersten Vorhersagen ein falsches Ergebnis (in der ersten Zeile).
  Dieser ist also genauer als der unkorrelierende Prädiktor.
- Die Korrelation:
  Der Inhalt eines Eintrags (die „fette“ Stelle) wird durch das Verhalten des gegebenen Branches in der vorigen Iteration bestimmt, und die Wahl der Vorhersage ist vom Inhalt des globalen Branches (aus Sicht des betrachteten Branches) abhängig.

Korrelierende Prädiktoren

Beispiel:

3. Zeile $\boxed{b_{2P} = 0/\mathbf{1}}$

Die „fette“ Stelle war links (Zeile 2); die wird mit b_{2A} (aus der zweiten Zeile), d. h. mit 0 belegt. So entsteht der Eintrag 0/1. Aus diesem wird die rechte Stelle für die Vorhersage gewählt, weil für b_2 das Verhalten des globalen Branches (b_{1A} in der dritten Zeile) gleich 1 war.

Beispiel (1-Bit- versus 2-Bit-Prädiktor)

Betrachten wir Schleifen, die im Programm unmittelbar hintereinander ausgeführt werden. Die zweite Schleife wird **neunmal** durchlaufen und dann beendet. Wie groß ist die Vorhersagegenauigkeit der zweiten Schleife, die wir mit dem 1-Bit-Buffer erreichen können?

Beim ersten Branch in der zweiten Schleife wird **Not Taken** vorhergesagt (weil dies das Verhalten am Ende der ersten Schleife war), was falsch ist.
Dann werden 8 Verzweigungen korrekt vorhergesagt, und der Schleifenabbruch wird nicht erkannt. Dadurch sind **8** von **10** Vorhersagen richtig $\rightarrow$ Genauigkeit **80%**.

Beispiel (1-Bit- versus 2-Bit-Prädiktor)

Verbesserung: 2-Bit-Buffer

Vermeiden, dass beim Schleifeneintritt aufgrund des Austritts aus einer vorherigen Schleife die falsche Vorhersage Not Taken abgegeben wird.

Die Vorhersage des 2-Bit-Buffers ändert sich nur, wenn zweimal hintereinander eine falsche Vorhersage abgegeben worden ist.

Für das Beispiel:
Nach Austritt aus der ersten Schleife (**falsche Vorhersage Taken**) lautet die nächste Vorhersage immer noch **Taken** $\rightarrow$ Genauigkeit der Vorhersage ist **90%**.

Beispiel (1-Bit- versus 2-Bit-Prädiktor)

Skizze zum Beispiel

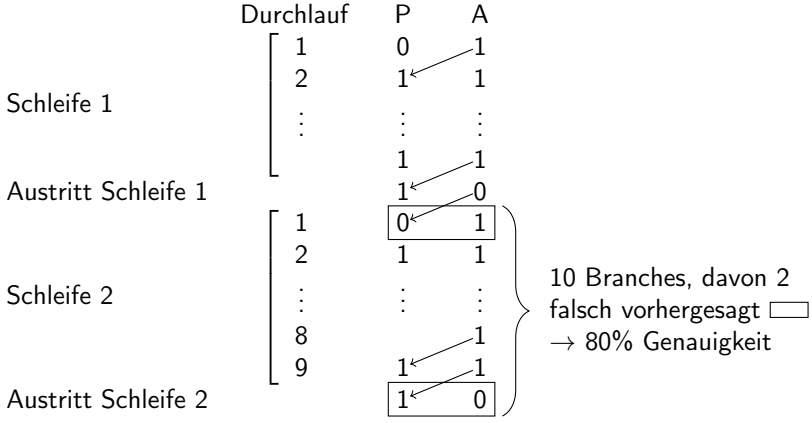
P: Vorhersage, A: Verhalten

- Schleife 1: Mindestens 3 Durchläufe
- Austritt Schleife 1
- Schleife 2: 9 Durchläufe
- Austritt Schleife 2

Beispiel (1-Bit- versus 2-Bit-Prädiktor)

Skizze zum Beispiel

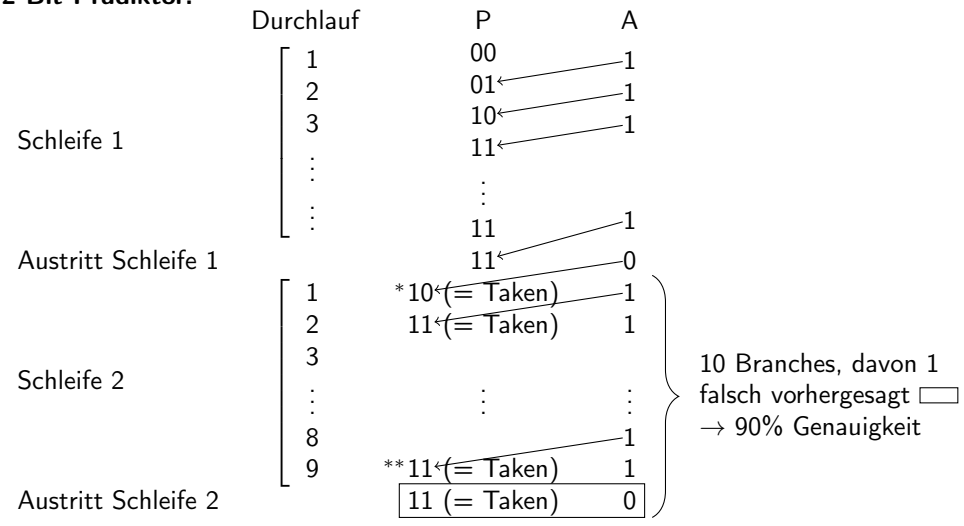
1-Bit-Prädiktor:



Beispiel (1-Bit- versus 2-Bit-Prädiktor)

Skizze zum Beispiel

2-Bit-Prädiktor:



Beispiel (1-Bit- versus 2-Bit-Prädiktor)

Skizze zum Beispiel

* $10 = 11 - 1$ (Bei Untaken (0) wird der Prädiktor in der nächsten Iteration dekrementiert)

** Wenn der Prädiktor den Wert 3 hat und das vorige Verhalten Taken (= 1) war, dann bleibt der Wert des Prädiktors auf 3 stehen.

Vorhersage der Sprungadresse

(mit Branch-Target Buffer (Verzweigezielpuffer))

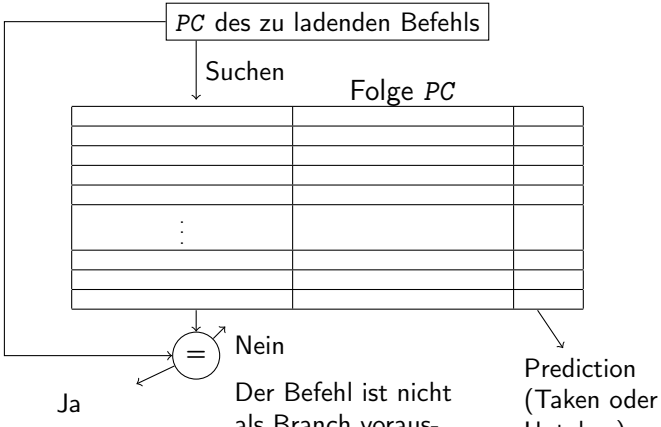
Um die Verzweigeverzögerung der DLX zu reduzieren, müssen wir am Ende von `IF` wissen, von welcher Adresse ein Befehl zu holen ist.

Also müssen wir wissen, ob der (noch nicht dekodierte) Befehl eine Verzweigung ist und wenn er eine Verzweigung ist, welchen Folgewert der *PC* haben sollte.

$\rightarrow$ Wenn der Befehl eine Verzweigung ist und wir wissen, wohin gesprungen wird (d. h. was der Folgewert des *PC* sein sollte), können wir ein Branch Delay (Verzweigeverzögerung) von null haben.

Der Puffer, der die vorausgesagte Adresse des nächsten Befehls nach einer Verzweigung speichert, wird Branch-Target Buffer genannt.

Vorhersage der Sprungadresse



Der Befehl ist ein Branch und der Folge-*PC* wird als nächster *PC* verwendet

Der Befehl ist nicht als Branch vorausgesagt

Vorhersage der Sprungadresse

Der *PC* des geholten (alten) Befehls wird mit einer Folge von Befehlsadressen verglichen, die in der ersten Spalte des Buffers gespeichert sind (d. h. mit den Adressen bekannter Verzweigungen).

Wenn *PC* mit einer dieser Adressen übereinstimmt, dann ist der geholte Befehl ein Branch. Wenn es ein Branch ist, dann erhält das zweite Feld die Adresse des Folge *PC* nach der Verzweigung.

Instruction Fetch (Befehlsholen des neuen Befehls) beginnt unmittelbar bei dieser Adresse.

Das dritte Feld zeigt an, ob der Branch als Taken (ausgeführt) oder Untaken (nicht ausgeführt) vorausgesagt war.

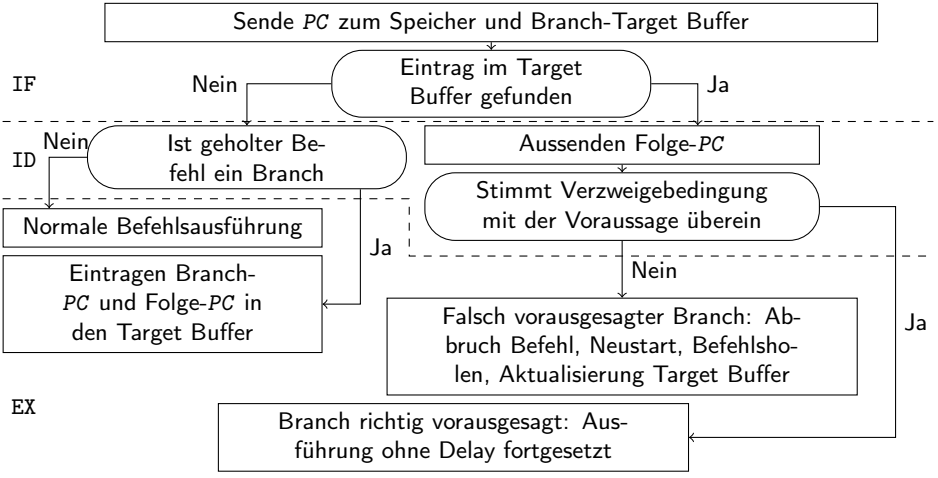
Vorhersage der Sprungadresse

Bemerkung: Der Branch-Target Buffer kann auch ohne Vorhersage-Spalte sein.

Dann wird das Adressen-Paar eingetragen, wenn der Branch-Befehl in der vorigen Iteration verzweigte.
Wenn dieser in der aktuellen Iteration als Branch erkannt wurde und nicht verzweigt, wird er aus der Liste gelöscht.

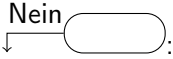
Vorhersage der Sprungadresse

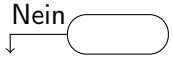
Alle möglichen Zustände der Vorhersage mit einem Target Buffer



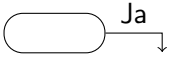
Vorhersage der Sprungadresse

Wenn der Eintrag (im linken Teil des Target Buffers) **nicht gefunden** wird



- 

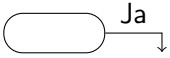
Der geholte (alte) Befehl **ist keine** Verzweigung. Dann folgt (in der EX-Phase) die normale Ausführung des Befehls (weil der Befehl kein Branch ist und daher keine Operation (Eintrag in den Buffer) nötig ist).

- 

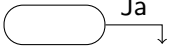
Der geholte (alte) Befehl **ist** ein Branch. Die Adresse dieses Befehls steht aber nicht im Buffer. Deswegen werden die Adressen des alten und des neuen Befehls in den Buffer eingetragen.

Vorhersage der Sprungadresse

Wenn die Adresse (im linken Teil des Target Buffers) **gefunden** wird

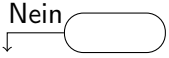
`IF`-Phase: 

Überprüfung der Branchbedingung: 2 Fälle

- `ID`-Phase: 

  Das Ergebnis der Auswertung der Branch-Bedingung stimmt mit der Vorhersage überein.
  Gewünschter Fall, d. h. die Vorhersage hat gestimmt.
  (Der neue Befehl ist richtig und befindet sich in der `IF`-Phase.)

Vorhersage der Sprungadresse

- ID-Phase: 

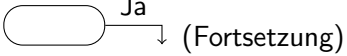
Hier sind die Auswertung der Branchbedingung und die Vorhersage unterschiedlich. Wir haben den neuen Befehl bereits in der Bearbeitung (Aussenden Folge-PC), aber dieser ist falsch.

  D. h.: **Entweder** ist es das Ziel des Sprungbefehls, aber es sollte der nächste Befehl (nach dem Branch) sein
  oder es ist der nächste Befehl (nach dem Branch), aber es sollte der Zielbefehl des Sprungbefehls sein.

  Deswegen muss man den aktuellen (neuen) Befehl abbrechen, den richtigen Befehl laden und die Prediction-Spalte des Buffers umändern.

Vorhersage der Sprungadresse

Wenn die Adresse (im linken Teil des Target Buffers) **gefunden** wird



Beim alten Befehl handelt es sich also um einen Branch und die Adresse des vorausgesagten *PC* (Adresse des neuen Befehls im rechten Teil des Buffers) wird zum Speicher und zum Target Buffer gesandt (deswegen zum Target Buffer, weil es sich wieder um einen Branch handeln könnte).

Vorhersage der Sprungadresse

Der Branch-Befehl (der alte Befehl) testet seine Branchbedingung, ob diese erfüllt oder nicht erfüllt ist (es handelt sich z. B. um eine der Bedingungen `BEQZ`, `BNEQZ`) und es wird ausgewertet, ob der Inhalt des Arguments zu der Bedingung passt.

Beispiel

`BEQZ` *R1*`, LOOP`

Wenn Wert des Arguments (*R1*) gleich 0 ist, dann wird die Branchbedingung erfüllt (also ist das Ergebnis dieser Auswertung dann gleich 1); sonst ist die Branchbedingung nicht erfüllt und das Ergebnis ist 0.

Dieses Ergebnis (1 oder 0) wird mit dem Vorhersage-Wert (dritter Teil des Buffers) verglichen.

Vorhersage der Sprungadresse

Verzögerung in allen möglichen Fällen (für einen Befehl, der ein Branch ist)

6 Fälle:
4 Fälle: Befehl im Buffer.
2 Fälle: Befehl nicht im Buffer.

- Wenn Befehl im Buffer:
  - Voraussage 1) — Taken (1); Untaken (0)
  - Aktuelle Verzweigung 2) — Taken (1); Untaken (0)

    (d. h. die Auswertung der Branchbedingung (des alten Befehls)).
- Wenn Befehl nicht im Buffer:
  - Voraussage: Nicht zutreffend, weil kein Eintrag im Buffer
  - Auswertung der Branchbedingung 3) — Taken (1); Untaken (0)

Vorhersage der Sprungadresse

	Befehl im Buffer	Prediction	Aktuelle Verzweigung	Verlustzyklen (Stalls)
1	Ja	Taken	Taken	0
2	Ja	Taken	Untaken	2
3	Ja	Untaken	Untaken	0
4	Ja	Untaken	Taken	2
5	Nein		Taken	2
6	Nein		Untaken	1

Zeilen 1–4: Kombinationen der Möglichkeiten 1), 2)
Zeilen 5–6: Möglichkeiten 3)

Vorhersage der Sprungadresse

- Zeile 1
  Die Auswertung der Verzweigungsbedingung stimmt mit der Vorhersage überein (weil beide Taken).
  Fall Ja Ja (siehe Skizze aller möglichen Zustände)
- Zeile 2
  Prediction ist Taken und Branch Untaken: Die Verzweigungsbedingung stimmt nicht mit der Vorhersage überein.
  Fall Ja Nein
- Zeile 3
  Prediction sowie auch die Auswertung der Branchbedingung ist 0 (Untaken). Also stimmt die Verzweigungsbedingung mit der Vorhersage überein.
  Fall Ja Ja

Vorhersage der Sprungadresse

- Zeile 4
  Prediction ist Untaken und Branch Taken: Die Verzweigungsbedingung stimmt nicht mit der Vorhersage überein.

	Ja
Fall	Nein

- Zeile 5
  Der Befehl ist nicht im Buffer, aber es ist eine Verzweigung (weil seine Auswertung ergibt, dass wir springen müssen).

	Nein
Fall	Ja

- Zeile 6
  Der Befehl ist nicht im Buffer, es ist aber ein Branch. Dessen Auswertung ergibt, dass wir nicht springen müssen.

	Nein
Fall	Ja

Vorhersage der Sprungadresse

Erklärung der Verzögerungen

- Zeile 1
  Der richtige Befehl wird (als neuer Befehl) geholt. Die Pipeline-Ausführung erfolgt **ohne Stalls**.
- Zeile 2
  1 Takt wird mit Abbruch des neuen Befehls (`IF`-Phase) verbraucht, weil dieser falsch ist. **Der zweite Takt** wird für das Update des Buffers (die dritte Spalte des Buffers) verbraucht. (Update muss eine unabhängige Operation sein, weil wir den Buffer nicht gleichzeitig lesen und schreiben können.)
- Zeile 3
  Übereinstimmung, was die Vorhersage und die Auswertung anbelangt. Wie in der Zeile 1: **keine Delays**, weil der richtige Befehl geladen wird.

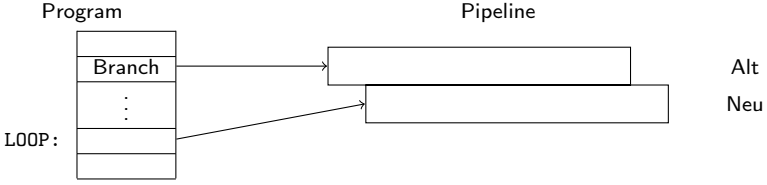
Vorhersage der Sprungadresse

- Zeile 4
  Ähnlich wie in Zeile 2: Falscher Befehl geladen (1 Takt verbraucht) und für den Buffer (Update) braucht man noch einen **zweiten** Takt.
- Zeile 5
  Es wird erst nach der `ID`-Phase erkannt, dass es sich um einen Branch-Befehl handelt, der verzweigen wird. Ein falscher (neuer) Befehl ist geladen, deswegen muss dieser Befehl weg aus der Pipeline (1 Takt Verlust). Im **zweiten Takt** (Stall) wird der Eintrag (Adresse des alten und neuen (Sprung) Befehls) im Buffer gemacht.
- Zeile 6
  Hier wurde der richtige neue Befehl geholt (weil der alte Befehl ein Branch ist, der aber nicht verzweigt) und deswegen kann der neue Befehl ausgeführt werden. 1 Takt Verlust für die Eintragung der Befehlsadressen in den Buffer.

Vorhersage der Sprungadresse

Skizzen

Zeile 1: Vorhersage Taken, Auswertung Taken

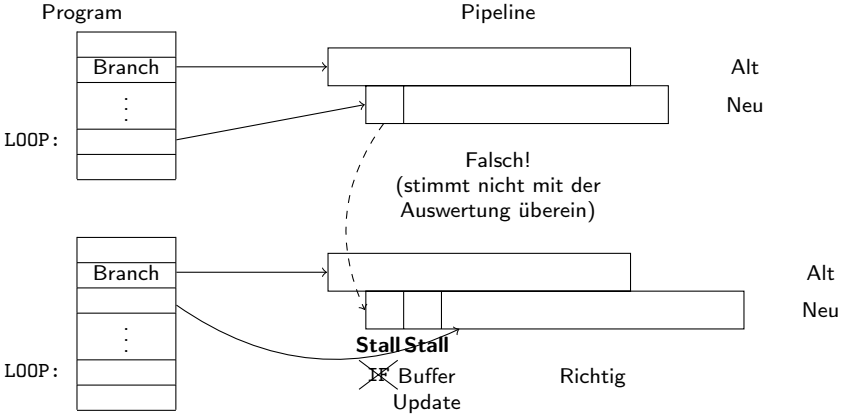


Kein Stall

Vorhersage der Sprungadresse

Skizzen

Zeile 2: Vorhersage Taken, Auswertung Untaken

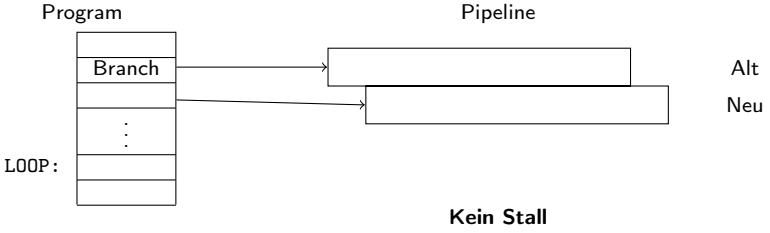


2 Stalls

Vorhersage der Sprungadresse

Skizzen

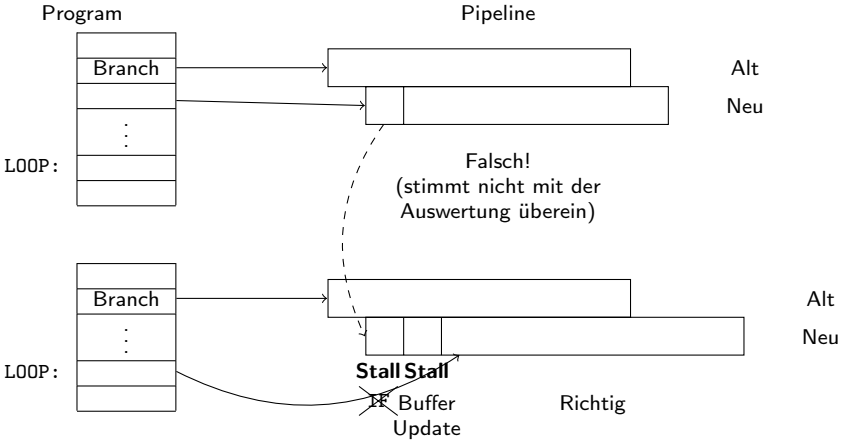
Zeile 3: Vorhersage Untaken, Auswertung Untaken



Vorhersage der Sprungadresse

Skizzen

Zeile 4: Vorhersage Untaken, Auswertung Taken

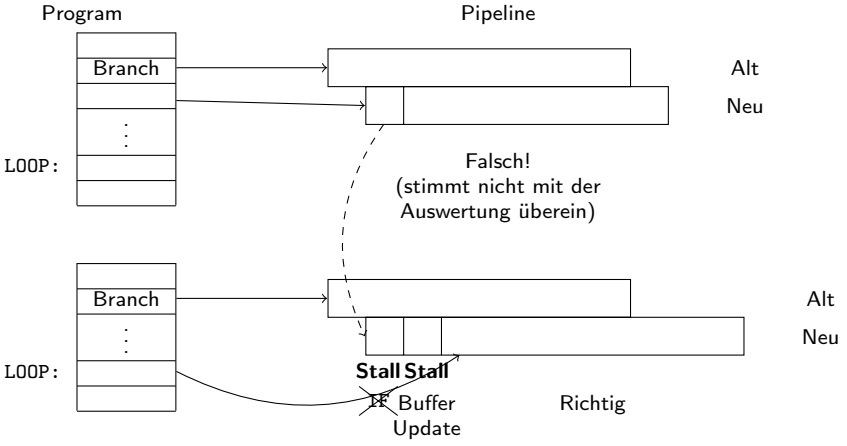


2 Stalls

Vorhersage der Sprungadresse

Skizzen

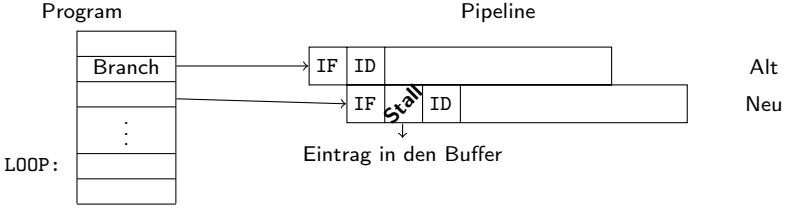
Zeile 5: keine Vorhersage, Auswertung Taken



Vorhersage der Sprungadresse

Skizzen

Zeile 6: keine Vorhersage, Auswertung Untaken



1 Stall