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

30. Mai 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*
    → 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:	$REGS[IR[16 \ldots 20]] \leftarrow ALUOUTPUT$ (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 keine Arbeit für diese Instruktionen in der `WB`-Stufe.)

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, weil 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*		
$\vdots$	$\vdots$		$\vdots$		$\vdots$		
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.

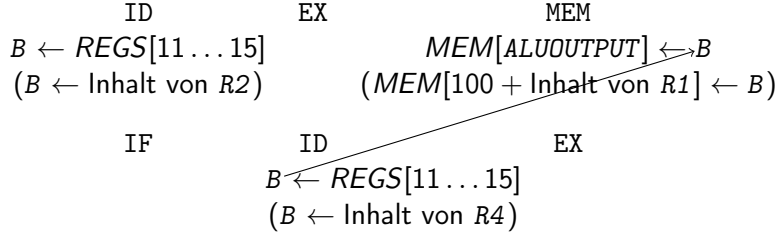
Probleme der Befehlsabarbeitung in der Pipeline (Fortsetzung)

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

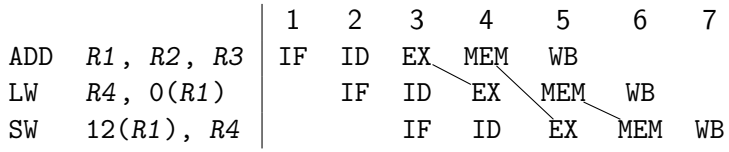
	1	2	3	4	5	6	7	8	9	10	11	12	13
ADD	IF	ID	EX	MEM	WB								
LW		Stall	Stall	Stall	IF	$\underbrace{\texttt{ID}}_{1}$	EX	MEM	WB				
SW						Stall	Stall	Stall	IF	$\underbrace{\texttt{ID}}_{2}$	EX	MEM	WB

- $\underbrace{}_{1}$: 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 $\texttt{MEM}_{\texttt{SW}}$.

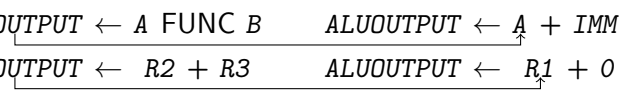
Ergebnis: keine Stalls (nur 7 Clocks)

Data Hazards (Forwarding Erklärung)

K1:

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

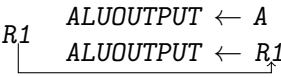
EX_{ADD} → EX_{LW}



K2:

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

MEM_{ADD} → EX_{SW}



Fertig nach EX_{ADD}, aber Forwarding kann nur zwischen benachbarten Stufen durchgeführt werden; deswegen wird hier Forwarding nach der MEM_{ADD}-Stufe (die benachbart zur EX_{SW}-Stufe ist) durchgeführt.

Data Hazards (Fortsetzung Forwarding Erklärung)

K3:

LW *R4*, 0(*R1*) → SW 12(*R1*), *R4*

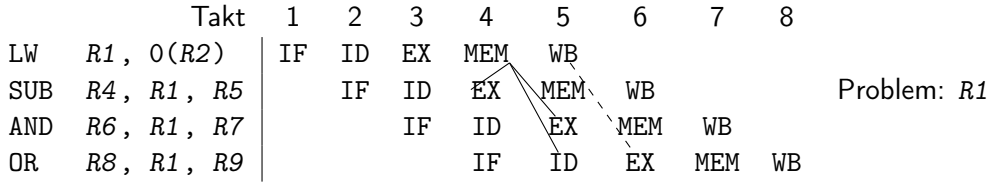
MEM_{LW} → MEM_{SW}

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

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

Data Hazards (Forwarding)

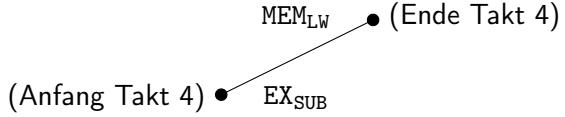
Beispiel, dass Forwarding **nicht immer** Stall vermeiden kann



- 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 geht eigentlich um Backwarding:



  → Hier muss die EX-Stufe des SUB-Befehls um einen Takt verzögert werden.
  → Stall kann man nicht vermeiden.
  Würde im SUB-Befehl *R1* mit *R2* ersetzt sein → 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.
  → **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 kann den „alten“ Wert von *R4* lesen, 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

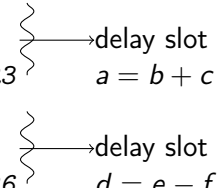
- Sequentielle Vorgangsweise

```
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
```



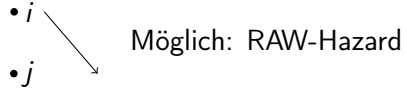
Nach dem Load-Befehl (dem Delay Slot) wird ein Befehl eingefügt, der das geladene Datum nicht verwendet.

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).



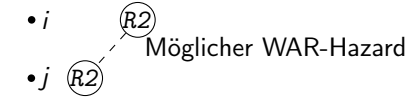
Static Scheduling (Fortsetzung)

- **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:

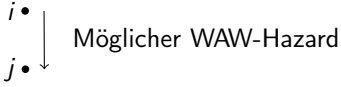
    ```
    ADD R3, R5, R2
    SUB R2, R4, R6
    ```

    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's Verfahren