IT-Infrastruktur

WS 2014/15

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz D': v1.0, 2014/11/13

- Rechnerstrukturen, Teil 2 (Pipelining)

Dieser Foliensatz

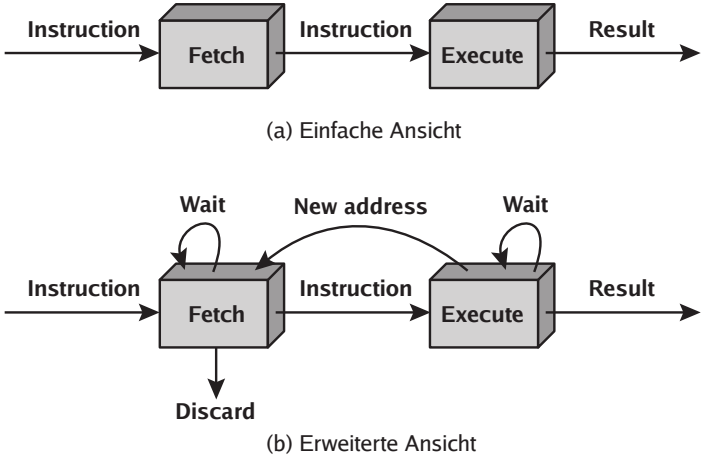
Vorlesungsübersicht
Seminar
Wiss. Arbeiten
Datenformate und Wandlung
PC als Arbeitsplatz
Ergonomie und Arbeitsschutz
Folien D' → Rechnerstrukturen
(Telekommunikation)
Infrastruktur-Technologie
Zentrale / verteilte IT-Infrastrukturen

Pipelining

Pipelining – Ziele

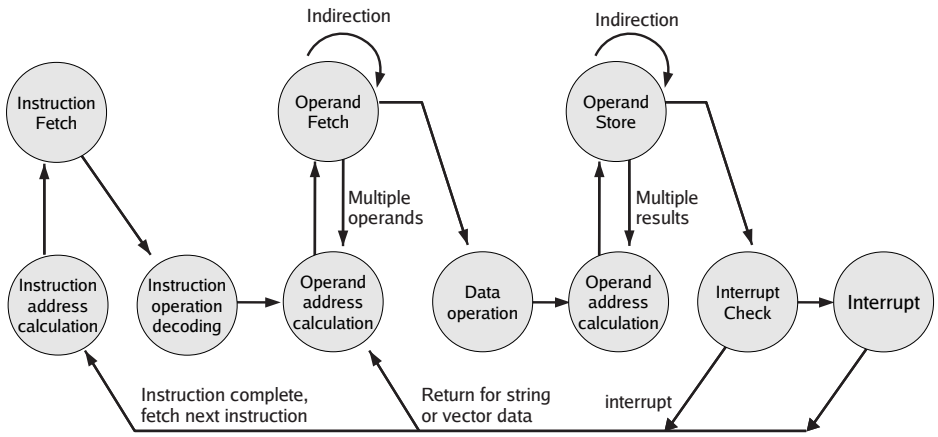
- Wesentliche Begriffe und Fakten über Pipelining kennen
- Vorteile des Pipelining kennen
- Bearbeitung von Sprungbefehlen – Arbeitsschritte für alle Befehle
  - Pipelining Hemmnisse verstehen: Sprünge, Speicherzugriff, Langläufer, Datenabhängigkeiten
- Zusammenhang Arbeitsschritte – Taktrate:
  - Implementierung analysieren können
  - MMIX-Programmstück analysieren und die Ausführung auf einer 5-stufigen Pipeline nachvollziehen können

Einleitung – einfache Pipeline



Quelle: Stallings, Computer Organization and Architecture, 8th ed., 2010, S. 463

Instruction Cycle State Diagram



Quelle: Stallings, Computer Organization and Architecture, 8th ed., 2010, S. 460

Pipeline-Designs

- 5-stufige RISC-Pipeline
  - Fetch – Decode – Execute – Memory Access – Write Back
  - siehe auch http://en.wikipedia.org/wiki/Classic_RISC_pipeline
- 6-stufige CISC-Pipeline
  - Fetch Instruction – Decode – Calculate Operands – Fetch Operands – Execute – Write Back
  - siehe Stallings, S. 464

Klassische 5-Stufen-RISC-Pipeline (1/3)

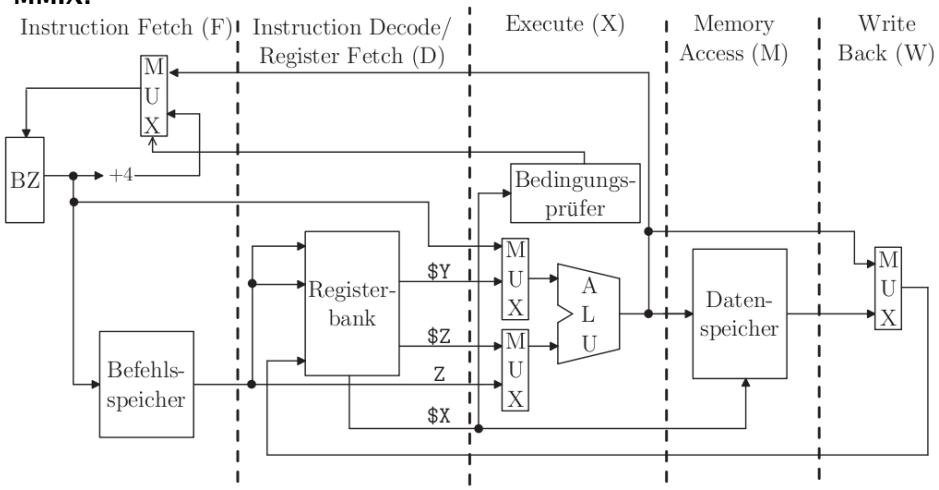
- Holen des Befehls – Fetch (**F**)
  Befehlszähler liefert Adresse für Speicherzugriff und wird (um vier) erhöht.
- Dekodieren des Befehls – Decode (**D**)
  Bei RISC üblicherweise „fixed field decoding“
  - 2 Register-Operanden (OP $X,**$Y,$Z**):
    Bereitstellen der beiden Operanden aus Registern
  - 1 Register-Operand, 1 Direktoperand (OP $X,**$Y,Z**):
    Register $Y auslesen; Z aus Befehlswort
  - Speicherzugriff: Adresse aus Register- und/oder Direktoperand auslesen.
    Bei schreibendem Zugriff: Wert aus $X lesen
  - Sprung: Für Bedingung $X auslesen; Direktoperand aus Befehlswort (für Sprungzielberechnung)

Klassische 5-Stufen-RISC-Pipeline (2/3)

- Ausführen des Befehls – Execute (**X**)
  - Im Befehl spezifizierte ALU-Operation ausführen
  - Bei Speicherzugriff: (ggf.) Adresse berechnen
  - Bei Verzweigungen: Bedingungen prüfen, Adresse des Folgebefehls festlegen
- Speicherzugriff – Memory Access (**M**)
  Lesender oder schreibender Speicherzugriff, falls erforderlich
- Ergebnis zurückschreiben – Result Write-back (**W**)
  Ergebnis einer Operation in ein Register schreiben, falls erforderlich (ALU-Output oder Speicherwort)

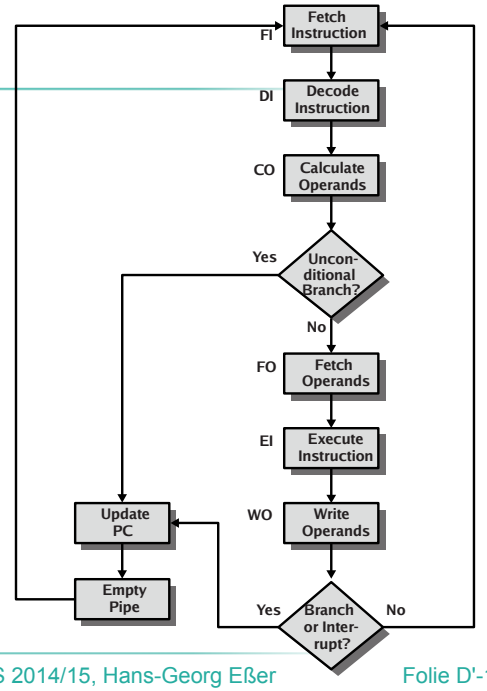
Klassische 5-Stufen-RISC-Pipeline (3/3)

MMIX:



MUX: Multiplexer

6-Stufen-CISC-Pipeline

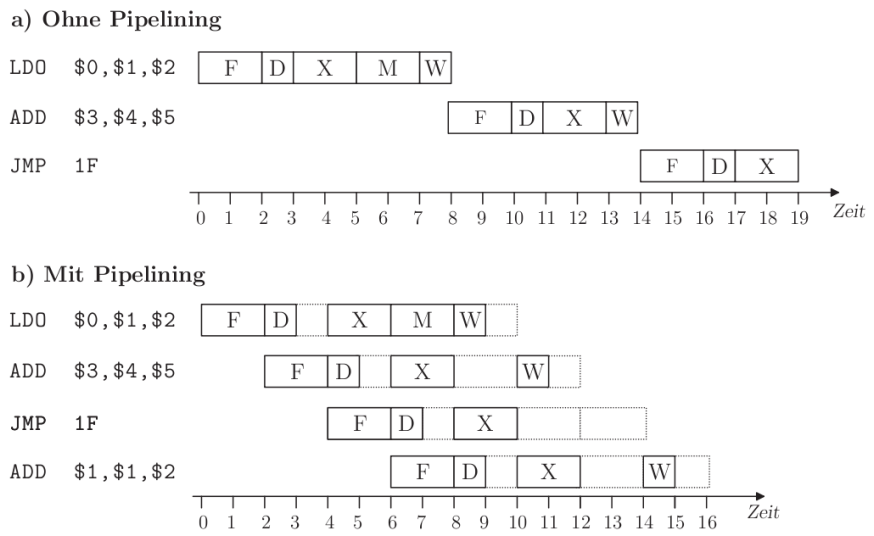


Quelle: Stallings, Computer Organization and Architecture

Takt

- Übergang zwischen Pipeline-Stufen muss getaktet sein
- Bei Übergang Werte-Weitergabe (CPU-interne Pipeline-Register)
- Annahmen:
  - Fetch, Execute, Memory Access: je 2 *t*
  - Decode, Write Back: je *t*
  - Dauer einer Takteinheit hängt von längster Stufe ab, hier also 2 *t*

Ausführung mit und ohne Pipelining



Pipeline-Diagramme („Reservation Tables“)

a) aus Sicht der Stufen

	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6	Takt 7
Fetch	Befehl 1	Befehl 2	Befehl 3	Befehl 4	Befehl 5	Befehl 6	Befehl 7
Decode		Befehl 1	Befehl 2	Befehl 3	Befehl 4	Befehl 5	Befehl 6
Execute			Befehl 1	Befehl 2	Befehl 3	Befehl 4	Befehl 5
Memory				Befehl 1	Befehl 2	Befehl 3	Befehl 4
Write Back					Befehl 1	Befehl 2	Befehl 3

b) aus Sicht der Befehle

	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6	Takt 7
Befehl 1	F	D	X	M	W		
Befehl 2		F	D	X	M	W	
Befehl 3			F	D	X	M	W
Befehl 4				F	D	X	M
Befehl 5					F	D	X

Pipeline-Hemmnisse

- Theoretisch: In jedem Takt ein Befehl fertig
- Praktisch: „Hemmnisse“ („Hazards“) verhindern das:
  - Strukturelle Hemmnisse
  - Datenabhängigkeiten (data hazard)
  - Ablaufbedingte Hemmnisse

Pipeline-Hemmnisse: strukturell (1)

- Nur ein Speicherzugriff pro Takt möglich

	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6	Takt 7
Befehl 1	F	D	X	M	W		
Befehl 2		F	D	X	M	W	
Befehl 3			F	D	X	M	W
Befehl 4				---	---	---	F
Befehl 5					---	---	---

- komplexe Befehle (z. B. Gleitkomma)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SETH $3,#4000	F	D	X	M	W									
FMUL $4,$4,$3		F	D	X	X	X	X	M	W					
FADD $4,$4,$2			F	D	--	--	--	X	X	X	X	M	W	
SET $3, $10				F	--	--	--	D	--	--	--	X	M	W

Pipeline-Hemmnisse: strukturell (2)

- Beispiele für Länge der Execute-Phase komplexer Befehle (bei MMIX):
  - Integer-Multiplikation: 10 Takte
  - Integer-Division: 60 Takte
  - Gleitkommabefehle: 4 Takte (i.d.R.)
- Taktrate auf Dauer des längsten Befehls anheben?

Pipeline-Hemmnisse: strukturell (3)

- Lösung für Speicherzugriff: Prefetching
  - In jeder Fetch-Phase mehrere Befehle aus dem RAM lesen und puffern („Fetch Buffer“)
  - z. B. MMIX: Befehlslänge 4 Bytes, eine Leseoperation liefert aber 8 Bytes
  - Ungenutzte Memory-Phasen (Befehle ohne Speicherzugriff) für weitere Fetch-Operationen verwenden
- getrennte Prozessor-Caches für Befehle und Daten (→ Pseudo-Harvard-Architektur)

Pipeline-Hemmnisse: Datenabhängigkeiten (1)

- Data hazards: Befehl benötigt ein Ergebnis, das noch nicht fertig berechnet wurde
- Beispiel: Tausch von zwei Werten (aus Quicksort)

	1	2	3	4	5	6	7	
`XOR l,l,r`	F	D	X	M	W			
`XOR r,l,r`		F	D	--	--	X	M	(l nicht bekannt)
`XOR l,l,r`			F	--	--	D	--	(r nicht bekannt)
`CMP tmp,l,pivot`				--	--	F	--	

Read-after-Write- (**RAW**-) Konflikt (zweiter Befehl braucht das im ersten berechnete l)

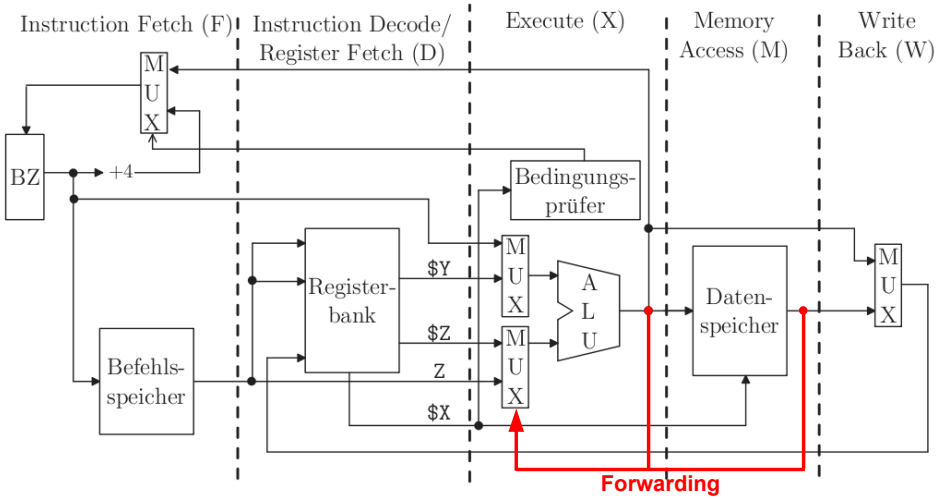
- Lösung: Result Forwarding / Bypassing

Pipeline-Hemmnisse: Datenabhängigkeiten (2)

Result Forwarding / Bypassing

- Ergebnisse von X- und M-Phasen immer als ALU-Input zurückreichen („feed back“)
- Spezielle Steuerlogik: Falls erforderlich, die zurückgereichten Ergebnisse (anstelle der Registerinhalte) verwenden

Pipeline-Hemmnisse: Datenabhängigkeiten (3)



Pipeline-Hemmnisse: Datenabhängigkeiten (4)

- ohne Forwarding

	1	2	3	4	5	6	7
`XOR l,l,r`	F	D	X	M	W		
`XOR r,l,r`		F	D	--	--	X	M
`XOR l,l,r`			F	--	--	D	--
`CMP tmp,l,pivot`				--	--	F	--

(l nicht bekannt)
(r nicht bekannt)

- mit Forwarding

	1	2	3	4	5	6	7
`XOR l,l,r`	F	D	X	M	W		
`XOR r,l,r`		F	D	X	M	W	
`XOR l,l,r`			F	D	X	M	W
`CMP tmp,l,pivot`				F	D	X	M

Pipeline-Hemmnisse: Datenabhängigkeiten (5)

- Result Forwarding hilft nicht immer
  - Lade-Befehle: Erst nach M-Phase steht Ergebnis zur Verfügung

	1	2	3	4	5	6	7
`LDO $1,base,off`	F	D	X	M	W		
`ADD $1,$1,$2`		F	D	--	X	M	W
`SUB $3,$4,$5`			F	--	D	X	M

Pipeline-Hemmnisse: ablaufbedingt (1)

- bei bedingtem Sprungbefehl: unklar, an welcher Stelle es weiter geht
- Sprungziel steht erst nach Execution-Phase des Sprungbefehls fest
- Frage: Was in die Pipeline schreiben?
  - einfach: Immer davon ausgehen, dass *nicht* gesprungen wird
  - MMIX: „probable branch“ vs. „branch“
  - komplizierter: Sprungvorhersage (eigenes Thema)

Pipeline-Hemmnisse: ablaufbedingt (2)

Beispiel: Berechne $\sum_{i=1}^{9} i$ und $2\sum_{i=1}^{9} i$

		1	2	3	4	5	6	7	8	9	10	11	12
i	IS $1												
sum	IS $2												
	SET i,10	F	D	X	M	W							
	SET sum,0		F	D	X	M	W						
loop	SUB i,i,1			F	D	X	M	W					
	ADD sum,sum,i				F	D	X	M	W				
	BNZ i,loop					F	D	X	M	W			
cont	STB sum,erg						~~F~~	~~D~~					
	ADD $3,sum,sum							~~F~~					
	STB $3,erg2												

Erst nach Schritt 7 ist klar, dass gesprungen wird

Zwei Stufen der Pipeline löschen

Interrupts (1)

- Auf externe (asynchrone) Interrupts muss die CPU schnell reagieren und in den Interrupt-Handler verzweigen
- Es dürfen keine Befehle unterbrochen werden, die bereits einen Teil der Zustandsänderungen bewirkt haben
- Beispiel: `STB $1, label`
  - nicht zwischen M- und W-Phase unterbrechen!
  - M-Phase: Wert aus $1 in Speicher schreiben
  - W-Phase: Falls Überlauf, Register `rA` (*arithmetic status register*) anpassen

Interrupts (2)

- Darum Befehle in „später“ Pipeline-Phase (X oder M), noch – vor Interrupt-Behandlung – fertig bearbeiten
- Befehle, die gerade erst dekodiert werden, können aber wieder verworfen werden

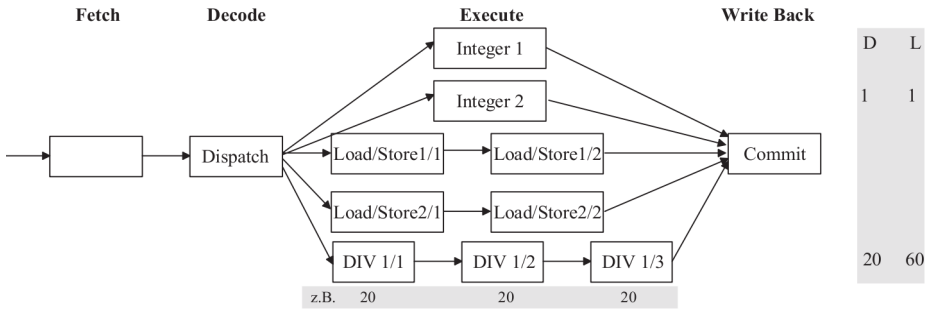
Superskalare Architekturen

Superskalare Architekturen (1)

- Idee: Parallelisierung durch mehrere (potenziell) parallel arbeitende Ausführungseinheiten (Funktionseinheiten)
- Die Funktionseinheiten können für sich wiederum mehrere Pipelinestufen enthalten
- Der Prozessor „entdeckt“ Möglichkeiten zur Parallelverarbeitung (*impliziter* Parallelismus)

  (*expliziter* Parallelismus: vom Programmierer festgelegt; Intel EPIC: **E**xplicit **P**arallel **I**nstruction **C**omputing; VLIW-Befehle – **V**ery **L**arge **I**nstruction **W**ord)

Superskalare Architekturen (2)



- **Durchsatzzeit** (**D**): Minimaler Abstand zwischen zwei Befehlen, die eine Ausführungseinheit verlassen
- **Latenzzeit** (**L**): Minimale Laufzeit eines Befehls durch eine Ausführungseinheit

Einführendes Beispiel: Parallele Pipelines (1/3)

- abhängige Pipelines
- unabhängige Pipelines

Beispiel: Rechnung aus Mandelbrot-Programm

```
23          FADD   p,p,plow
24          FADD   q,q,qlow
25          SET    xk,0
26          SET    yk,0
27          SET    k,0
28     * Nächste Iteration: x_{k+1} = x_k^2-y_k^2+p
29     1H   INCL   k,1
```


Einführendes Beispiel: Parallele Pipelines (2/3)

- abhängige Pipelines (z. B. Pentium)
  - Instruktionen müssen synchron durch die Pipelines laufen: gleichzeitiger Wechsel in nächste Stufe
  - Befehle müssen also ggf. „aufeinander warten“
  - Pipelines heißen **U** und **V** (wie bei Pentium)

	1	2	3	4	5	6	7	8	9	Pipeline
FADD q,q,qlow	F	D	X	X	X	X	M	W		**U**
SET xk,0	F	D	X	**	**	**	M	W		**V**
SET yk,0		F	D	**	**	**	X	M	W	**U**
SET k,0		F	D	**	**	**	X	M	W	**V**
INCL k,1			F	**	**	**	D	X	M	**U**

Einführendes Beispiel: Parallele Pipelines (3/3)

- unabhängige Pipelines
  - nicht synchron durch die Stufen; kein Warten
  - mehr „Logistik“ für Steuerung der Stufen nötig

	1	2	3	4	5	6	7	8	Pipeline
`FADD q,q,qlow`	F	D	X	X	X	X	M	W	**U**
`SET xk,0`	F	D	X	M	W				**V**
`SET yk,0`		F	D	**	**	**	X	M	**U**
`SET k,0`		F	D	X	M	W			**V**
`INCL k,1`			F	**	**	**	D	X	**U**

  - ◯: „**out-of-order completion**“ (Fertigstellung in falscher Reihenfolge)

Out-of-order completion

- Problem: Ergebnis eines späten Befehls vor Fertigstellung eines vorangehenden Befehls verfügbar
- Befehle sind aber voneinander abhängig
- Pipeline muss Befehl evtl. anhalten, bis Ergebnisse aus logisch vorangehenden Befehlen vorliegen
  - kann wieder Result Forwarding verwenden

Parallele Ausführung

- Zwei Pipelines (U, V); abhängig
- Zwei Pipelines; unabhängig
  ⇒ out of order completion
- Superskalare Pipelines allgemein: unabhängig und evtl. mehr als zwei

  ⇒ Es kann (wie bei zwei unabhäng. Pipelines) verschiedene Datenabhängigkeiten geben
- Ziel ist immer: Ergebnis muss identisch mit dem bei sequentieller Ausführung sein

Datenabhängigkeiten (1)

- Keine Abhängigkeit:

```
30    FMUL temp1,xk,xk      x_k²
31    FMUL temp2,yk,yk      y_k²
```

  (kann man einfach vertauschen)
- **Read-after-Write**-Abhängigkeit (**RAW**):

```
50    ADD  temp2,temp2,bildx
51    STBU k,    temp1,temp2
```

  tauschen verboten (sonst wird in #51 der Wert an der falschen Adresse `temp1+temp2` gespeichert)

Datenabhängigkeiten (2)

- **Write-after-Read**-Abhängigkeit (**WAR**):

```
35    FMUL yk,   xk,yk
36    SETH xk,   #4000      2,0 (Gleitkommawert!)
37    FMUL yk,   yk,xk      2 * y_k
```

  - #36 muss zwischen #35 und #37 bleiben
  - #35/#36 tauschbar, falls anderes Register verwendet wird
  - auch: **Antidependence**, „falsche“ Abhängigkeit
  - außerdem: **Name Dependence**

Datenabhängigkeiten (3)

- **Write-after-Write**-Abhängigkeit (**WAW**):

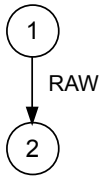
```
37    FMUL yk,   yk,xk    2 * y_k
38    FADD yk,   yk,q
```

  - Beide Befehle schreiben yk. Würde #38 vor #37 fertiggestellt (Write-Back-Phase), wäre das Ergebnis falsch
  - auch: **Output Dependence**
  - außerdem: **Name Dependence**

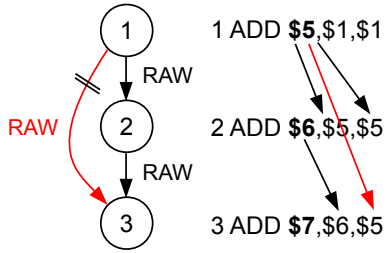
Datenabhängigkeiten, Übersicht

RAW	**Read-After-Write** true dependence	Befehl *B* verwendet das Ergebnis von Befehl *A* (Vorgänger). Konflikt, falls Ergebnis noch nicht da
WAR	**Write-After-Read** antidependence (name dependence)	*B* überschreibt Wert, den der Vorgänger *A* benötigt. Konflikt, falls *B* schreibt, bevor *A* liest
WAW	**Write-After-Write** output dependence (name dependence)	*A* und *B* schreiben in dasselbe Register (oder dieselbe Speicherzelle) Konflikt, wenn *B* vor *A* schreibt
RAR	***Read-After-Read***	*A und B lesen denselben Wert; kein Konflikt*

Abhängigkeitsgraph (1)

- Abhängigkeiten grafisch darstellen
  - 

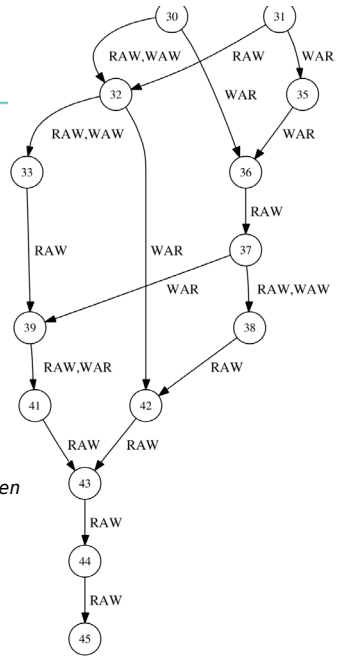
    bedeutet eine RAW-Abhängigkeit zwischen Instruktionen 1 und 2, also: (2) liest Wert, den (1) schreibt
  - mehrere Abhängigkeiten gleichzeitig: alle angeben (z. B. „RAW, WAW“)
  - keine transitiven Abhängigkeiten einzeichnen



Abhängigkeitsgraph (2)

Auszug aus einem Beispiel-Programm

```
30      FMUL   temp1,xk,xk        x_k^2
31      FMUL   temp2,yk,yk        y_k^2
32      FSUB   temp1,temp1,temp2
33      FADD   temp1,temp1,p      x_{k+1}
34 * y_{k+1}=2x_k y_k+q
35      FMUL   yk,   xk,yk
36      SETH   xk,   #4000        2,0 (Gleit-
                                  kommawert!)
37      FMUL   yk,   yk,xk        2 * y_k
38      FADD   yk,   yk,q
39      SET    xk,   temp1        x_k kann über-
                                  schrieben werden
40 * r=x_{k+1}^2+y_{k+1}^2
41      FMUL   temp1,xk,xk
42      FMUL   temp2,yk,yk
43      FADD   r,    temp1,temp2
44      FCMP   test, r,:M
45      BNP          test, 2F
```

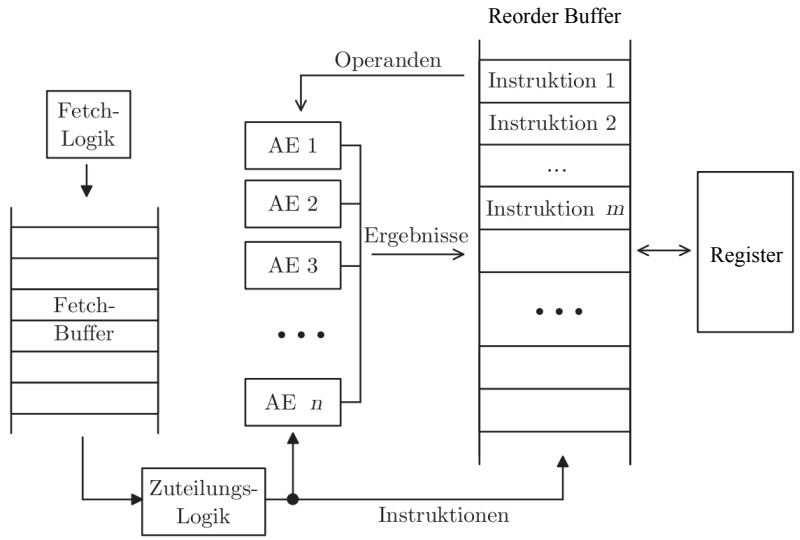


Phasen superskalarer Pipelines

- **Instruction Fetch:** Es steht ein Puffer zur Verfügung (Fetch buffer), in dem mehrere Instruktionen bereit stehen können.
- **Instruction Issue:** Instruktionen werden der Reihe nach (in order) auf freie Ausführungseinheiten (functional units, FUs) verteilt („issued“).

  Dieser Prozess stoppt, falls keine freien FUs vorhanden sind.
- **Read Operands:** Instruktionen warten ggf. auf ihre Operanden (RAW); ab hier ist Überholen möglich.
- **Execute:** out of order (OOO). Instruktionen warten, bis alle WAR-Vorgänger fertig sind.
- **Write Back:** Endgültiges Zurückschreiben in order: Write-after-Write (WAW) beachten. Ergebnisse stehen allerdings schon vorher zur Verfügung (Forwarding).

Reorder Buffer



Reorder Buffer

1. Freie Ausführungseinheit suchen.
   Falls verfügbar, durch den Befehl belegen und einen Eintrag im **Re**order **B**uffer (**ROB**) anlegen
2. Operanden bereitstellen. Für jeden Operanden den ROB von unten nach oben durchsuchen:
   - Wert aus Register (kein Ergebnis eines Bef. im ROB) *oder*
   - Ergebnis eines vorangehenden (fertigen) Befehls *oder*
   - vorangehenden Befehl beobachten, falls er noch nicht fertig ausgeführt ist (Pause!)
3. Befehl ausführen. Ergebnis im Eintrag des Befehls im ROB vermerken
4. Befehl bestätigen (commit): Ergebnisse in die Register übernehmen

Aufbau des Reorder Buffer

t_0			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
`FMUL yk,yk,xk`	MUL	▶	$4	$3	—	—	$4 (ungültig)
`FADD yk,yk,q`	FPU	❚❚	—	$2	FMUL	—	$4 (ungültig)

```
q  = $2
xk = $3
yk = $4
```

- Befehl
- Unit (welche Ausführungseinheit?)
- Status: ▶ = läuft, ❚❚ = Pause, ✔ = fertig
- Operanden: Q_Y, Q_Z: Werte der zwei Operanden, evtl. noch nicht verfügbar
- V_Y, V_Z: ggf. Zeiger auf ROB-Eintrag ⇒ dann Q_Y bzw. Q_Z bereits (im ROB) verfügbar
- Ziel

Beispiel: Reorder Buffer (1)

- wieder unser Beispiel-Programm; diesmal Zeilen 34-41

```
34  * y_{k+1} = 2 x_k y_k + q
35  FMUL yk,xy,yk
36  SETH xk,#4000
37  FMUL yk,yk,xk
38  FADD yk,yk,q
39  SET  xk,temp1
40  * r = x_{k+1}^2 + y_{k+1}^2
41  FMUL temp1,xk,xk
```

das schauen wir uns an ...

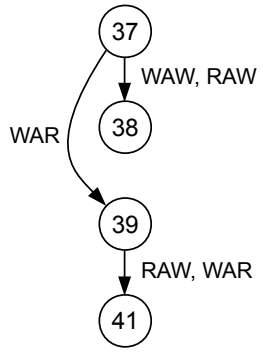
Legende

```
q     = $2
xk    = $3
yk    = $4
temp1 = $10
```


Beispiel: Reorder Buffer (2)

- Abhängigkeitsgraph:

```
37  FMUL yk,    yk,xk
38  FADD yk,    yk,q
39  SET  xk,    temp1
40
41  FMUL temp1,xk,xk
```



Beispiel: Reorder Buffer (3)

Annahmen:

- 1 FPU (alle Floating-Point-Ops außer FMUL)
- 2 Integer-Einheiten (INT1, INT2), alles außer Multiplikation
- 1 MUL (FP- und Integer-Multiplikation)
- FPU und MUL: je 4 Takte, INT: je 1 Takt
- Reorder Buffer hat Platz für 4 Einträge
- Je Takt max. 2 Befehle zuteilen (issue) und max. 2 Befehle bestätigen (commit)

q = $2, xk = $3, yk = $4, temp1 = $10

(4)

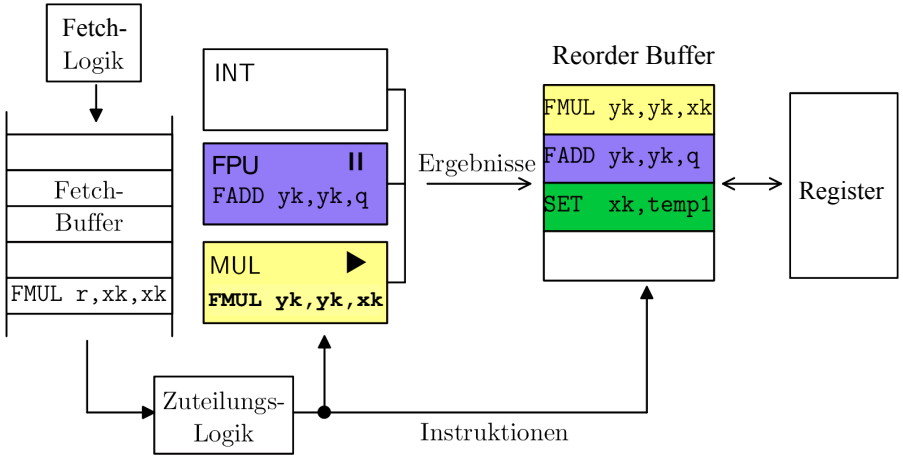
t_0			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
FMUL yk,yk,xk	MUL	▶	$4	$3	—	—	$4 (ungültig)
FADD yk,yk,q	FPU	❚❚	—	$2	FMUL	—	$4 (ungültig)
—							
—							

t_1			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
FMUL yk,yk,xk	MUL	▶	$4	$3	—	—	$4 (ungültig)
FADD yk,yk,q	FPU	❚❚	—	$2	FMUL	—	$4 (ungültig)
SET xk,temp1	INT1	▶	$10	×	—	—	$3 (ungültig)
—							

t_2 und t_3			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
FMUL yk,yk,xk	MUL	▶	$4	$3	—	—	$4 (ungültig)
FADD yk,yk,q	FPU	❚❚	—	$2	FMUL	—	$4 (ungültig)
SET xk,temp1	—	✔	$10	×	—	—	$3
—							

zu den Taktzeitpunkten t_2 und t_3:

(5)



q = $2, xk = $3, yk = $4, temp1 = $10

q = $2, xk = $3, yk = $4, temp1 = $10

(6)

t_4			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
FMUL yk,yk,xk	—	✔	$4	$3	—	—	$4
FADD yk,yk,q	FPU	▶	$4*	$2	—	—	$4 (ung.)
SET xk,temp1	INT1	✔	$10	×	—	—	$3
FMUL temp1,xk,xk	MUL	▶	$3*	$3*	—	—	—

t_5			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
FADD yk,yk,q	FPU	▶	$4*	$2	—	—	$4 (ung.)
SET xk,temp1	—	✔	$10	×	—	—	$3
FMUL temp1,xk,xk	MUL	▶	$3*	$3*	—	—	$10 (ung.)
—							

*) Wert aus ROB übernommen (Forwarding)

Beispiel: Reorder Buffer (7)

Klassisch:

			0	1	2	3	4	5	6	7	8	
FMUL **yk**, yk,xk	F	D	Xm	Xm	Xm	Xm	W					(Xm: 0-3)
FADD **yk**, yk,q	F	D	-	-	-	-	Xf	Xf	Xf	Xf	W	(Xf: 4-7)
SET **xk**, temp1		F	D	Xi	W							(Xi: 1)
FMUL **temp1**,xk,xk		F	D	-	-	-	Xm	Xm	Xm	Xm	W	(Xm: 4-7)

Leicht geändertes Programm:

			0	1	2	3	4	5	6	7	8		
FMUL **yk**, yk,xk	F	D	Xm	Xm	Xm	Xm	W						(Xm: 0-3)
FADD **yk**, yk,q	F	D	-	-	-	-	Xf	Xf	Xf	Xf	W		(Xf: 4-7)
SET **yk**, temp1		F	D	-	-	-	-	-	-	-	Xi	W	(Xi: 8)
FMUL **temp1**,yk,yk		F	D	-	-	-	-	-	-	-	-	Xm	(Xm: 9-)

Register Renaming (1)

- Jeder Eintrag im Reorder Buffer benötigt Register zum Speichern von Operanden und spekulativen (= noch nicht bestätigten) Ergebnissen.
- Diese heißen **Schattenregister** oder **Rename Registers** (im Unterschied zu den **Befehls-satz-Registern**; **Architectural Registers**)
- *Was* wird denn hier gespeichert?

Register Renaming (2)

- Was im Reorder Buffer steht...

Befehlssatz-Register 0..n

t_0			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
`FMUL yk,yk,xk`	MUL	▶	$4	$3	—	—	$4 (ungültig)
`FADD yk,yk,q`	FPU	❚❚	—	$2	FMUL	—	$4 (ungültig)
—							
—							

Register-Nummer (0..n) dito dito (nur Hinweis, wo Ergebnis später hin gehört; sowie Status)

Verweis auf Ergebnis im ROB Reg.-Nr. (0..n) für Ziel, Status **und(!)** Ergebniswert

t_4			Operanden				
Befehl	unit	Status	Q_Y	Q_Z	V_Y	V_Z	Ergebnis
`FMUL yk,yk,xk`	—	✔	$4	$3	—	—	$4
`FADD yk,yk,q`	FPU	▶	$4*	$2	—	—	$4 (ung.)
`SET xk,temp1`	INT1	✔	$10	×	—	—	$3
`FMUL temp1,xk,xk`	MUL	▶	$3*	$3*	—	—	—

Register Renaming (3)

- Pro „Zeile“ im ROB muss die Ziel-Spalte also Platz bieten für
  - eine Register-Nummer (Zielregister)
  - Status
  - einen berechneten Wert (in „Registergröße“, diese Teile des ROB heißen „Rename-Register“)
- Es ist ökonomisch sinnvoll, lediglich Pointer auf Rename Register in einem Register-Pool (für Zwischenergebnisse) zu speichern.

Register Renaming (4)

Register-Pool:

Wert	frei	gültig
	+	–
	–	+
	–	+
	–	–
	+	–
	–	+
	–	–
	–	+
	+	–
	–	+

Pointer aus ROB-Ziel-Spalte

frei: Eintrag nicht benutzt

gültig: enthält fertig berechneten Wert, kann für weitere Berechnungen genutzt werden

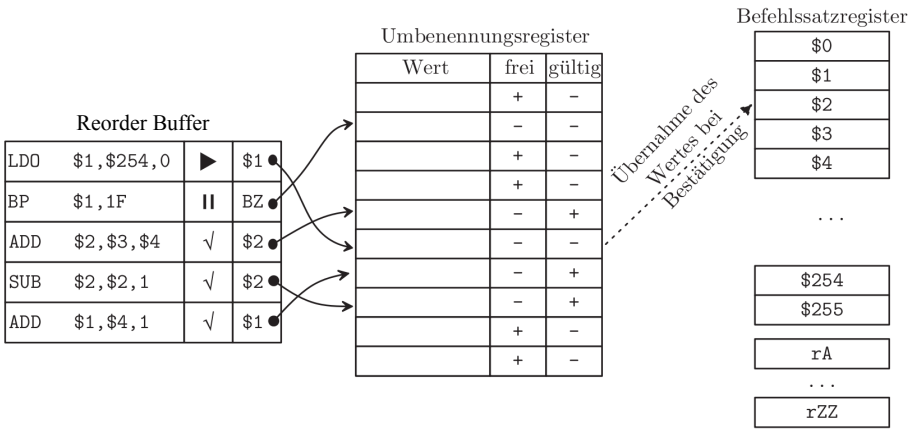
Register Renaming (5)

Implizite Umbenennung

- Zwei getrennte Sätze physischer Register für Befehlssatz-Register und Rename Register
- Es ist nirgends explizit vermerkt, wo sich der jüngste spekulative Wert eines Registers befindet. Finden des Wertes: ROB durchsuchen
  - Neuer Befehl `ADD $1,$3,$3` kommt in den ROB. „Welches“ $3 nehmen? Reg.-Inhalt? ROB-Inhalt? Warten auf Berechnung von $3?
- Das Verwerfen spekulativer Werte (bei Interrupts oder falsch vorhergesagten Sprüngen) ist einfach.

Register Renaming (6)

Implizite Umbenennung



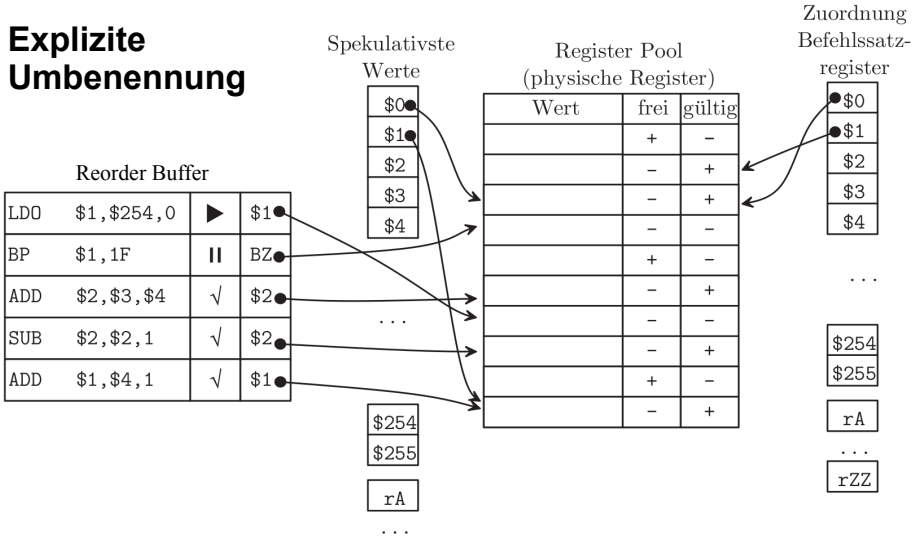
Register Renaming (7)

Explizite Umbenennung

- Es gibt nur einen Register-Pool (gemeinsam für Befehlssatz- und Rename-Register)
- Eine Tabelle gibt an, wo die jüngsten (spekulativsten) Werte der Register stehen
- Kein Umkopieren der Register nach Bestätigen eines Befehls;
  Pointer auf die physischen Register beschreiben Registerzustand

Register Renaming (8)

Explizite Umbenennung



Register Renaming (9)

Datenabhängigkeiten

- **Read-after-Write** (**RAW**):
  führt stets zum Stillstand, bis das Ergebnis zur Verfügung steht.
- **Write-after-Read** (**WAR**) und
  Write-after-Write (**WAW**):
  Schattenregister können Stillstand verhindern (wenn es genügend freie gibt)

Register Renaming (10)

- „Register Renaming“ auch direkt aus dem Wort verständlich:

```
1. R1=M[1024]
2. R1=R1+2
3. M[1032]=R1
4. R1=M[2048]
5. R1=R1+4
6. M[2056]=R1
```

Nach Umbenennen von R1 in R2:

```
1. R1=M[1024]     4. R2=M[2048]
2. R1=R1+2        5. R2=R2+4
3. M[1032]=R1     6. M[2056]=R2
```

WAR-Konflikt zwischen (3) und (4); Befehl (4) darf erst schreiben, wenn (3) fertig ist

„echte“ Pipelines

- echte Beispiel-Architekturen:
  - Pentium Pro: bis zu 16 Stufen
  - PowerPC 970: 16–24 Stufen

→ Übung

Anhang: Mandelbrot-Programm (1/2)

```
01              PREFIX  :Mandel:
02              LOC     #100
03    plow      GREG    #c002000000000000       -0,025
04    pdelta    GREG    #3f83333333333333       0,009375=(0,75- (-2,25))/320
05    qlow      GREG    #bff8000000000000       -1,5
06    qdelta    GREG    #3f8999999999999a       0,0125=(1,5 - (-1,5))/240
07    p         IS      $1
08    q         IS      $2
09    xk        IS      $3                      x_k
10    yk        IS      $4                      y_k
11    k         IS      $5                      Iteration k
12    r         IS      $6
13    bildx     IS      $7                      Gerätekoordinaten (integer)
14    bildy     IS      $8
15    test      IS      $9
16    temp1     IS      $10                     Zwischenergebnisse
17    temp2     IS      $11
18
19    :Mandel   FLOT    p,bildx
20              FLOT    q,bildy
21              FMUL    p,p,pdelta
22              FMUL    q,q,qdelta
23              FADD    p,p,plow
24              FADD    q,q,qlow
25              SET     xk,0
26              SET     yk,0
27              SET     k,0
28    * Nächste Iteration: x_{k+1}=x_k^2-y_k^2+p
29    1H        INCL    k,1
30              FMUL    temp1,xk,xk             x_k^2
31              FMUL    temp2,yk,yk             y_k^2
32              FSUB    temp1,temp1,temp2
33              FADD    temp1,temp1,p           x_{k+1}
```

Quelle: Böttcher, „Rechneraufbau und Rechnerarchitektur“

Anhang: Mandelbrot-Programm (2/2)

```
34    * y_{k+1}=2x_ky_k+q
35          FMUL     yk,xk,yk
36          SETH     xk,#4000              2,0 (Gleitkommawert!)
37          FMUL     yk,yk,xk              2 * y_k
38          FADD     yk,yk,q
39          SET      xk,temp1              xk kann überschrieben werden
40    * r=x_{k+1}^2+y_{k+1}^2
41          FMUL     temp1,xk,xk
42          FMUL     temp2,yk,yk
43          FADD     r,temp1,temp2
44          FCMP     test,r,:M
45          BNP      test,2F
46          LDA      temp1,:Bmp:data       Punkt in Farbe k setzen
47          SET      temp2,:WIDTH
48          MUL      temp2,bildy,temp2     so viele Bytes bis y
49          ADD      temp2,temp2,bildx
50          STBU     k,temp1,temp2
51          JMP      4F
52
53    2H    CMP      test,k,:K
54          BNZ      test,1B               noch eine Iteration
55    * sonst: farbe schwarz, also nichts zu tun
56    4H    INCL     bildx,1               Loop zuende
57          SET      test,:WIDTH
58          CMP      test,bildx,test       Zeilenende?
59          BNZ      test,:Mandel          nein
60          SET      bildx,0               ja: in nächste Zeile
61          INCL     bildy,1
62          CMP      test,bildy,:HEIGHT
63          PBNZ     test,:Mandel
64          POP      0,0                   Fertig!
```