

Kap.3 Mikroarchitektur

Prozessoren, interne Sicht

1

3.1 Elementare Datentypen, Operationen
und ihre Realisierung (siehe 2.1)

3.2 Mikroprogrammierung

3.3 Einfache Implementierung von MIPS

3.4 Pipelining

3.5 Superskalare Befehlsausführung

Übersicht

Performanz von Rechnern lässt sich durch Pipelining steigern

- Prinzip der Fließbandverarbeitung
- Hardware-Realisierung des MIPS-Datenpfades mit Fließband
- Probleme bei Fließbandverarbeitung

JR - RA - SS02

Kap. 3.4

3

Das Prinzip an einem alltäglichen Beispiel

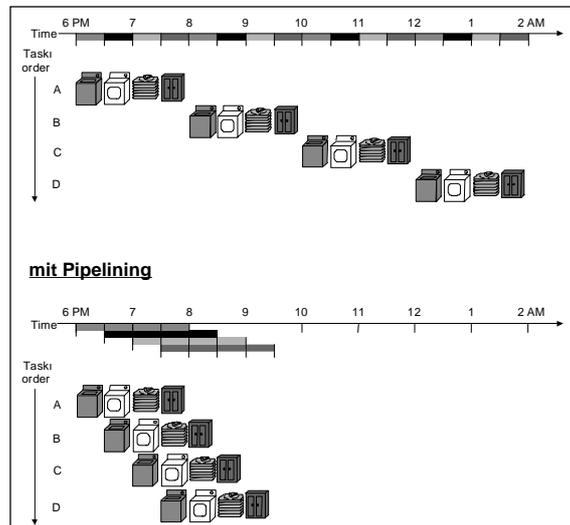
- Sie kommen aus dem Urlaub und es ist viel schmutzige Wäsche zu waschen!
- Zur Verfügung stehen:
 - eine Waschmaschine (1/2 Stunde Laufzeit)
 - ein Trockner (1/2 Stunde Laufzeit)
 - eine Bügelmaschine (1/2 Stunde Arbeit zum Bügeln)
 - ein Wäscheschrank (1/2 Stunde Arbeit zum Einräumen)
- jeder der Personen A, B, C, D aus dem Haushalt wäscht seine Wäsche selbst
- Es gibt zwei Möglichkeiten die vier Waschvorgänge auszuführen!

JR - RA - SS02

Kap. 3.4

4

Das Prinzip an einem Beispiel



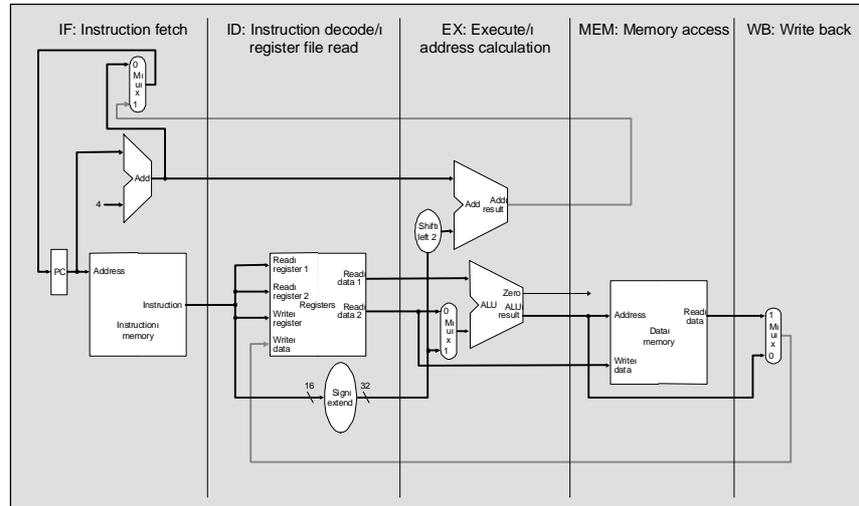
Dauer der Arbeiten:
8 Stunden

Dauer der Arbeiten:
3 1/2 Stunden

Aufteilung der Befehlsabarbeitung in Phasen

- Um Pipelining im Datenpfad ausnutzen zu können, muß die Abarbeitung eines Maschinenbefehls in mehrere Phasen mit möglichst gleicher Dauer aufgeteilt werden.
- Eine sinnvolle Aufteilung ist abhängig vom Befehlssatz und der verwendeten Hardware.
- Beispiel: Abarbeitung in 5 Schritten:
 - Befehls-Holphase (instruction fetch)
 - Dekodierphase / Lesen von Operanden aus Registern
 - Ausführung / Adressberechnung
 - Speicherzugriff (memory access)
 - Abspeicherphase (result write back phase)
- Bei CISC ist Pipelining schwer möglich, da die Dauer der Dekodier- und Ausführungsphase (evtl. mehrere Mikroprogrammbefehle) bei den verschiedenen Maschinenbefehlen sehr unterschiedlich ist.

Aufteilung des Datenpfades in 5 Phasen



Pipelining: Illustration

- Annahme: Aufteilung der Befehlsabarbeitung in 5 gleichlange Phasen

Befehl 1:	P1	P2	P3	P4	P5						
Befehl 2:		P1	P2	P3	P4	P5					
Befehl 3:			P1	P2	P3	P4	P5				
Befehl 4:				P1	P2	P3	P4	P5			
Befehl 5:					P1	P2	P3	P4	P5		
Befehl 6:						P1	P2	P3	P4	P5	
Befehl 7:							P1	P2	P3	P4	P5
Zeitschritt:	1	2	3	4	5	6	7	8	9	10	11

Pipelining: Speedup (1)

■ Annahmen:

- Abarbeitungszeit eines Befehls ohne Pipelining: t
- k Pipelinestufen, gleiche Laufzeit der Stufen
- Laufzeit einer Stufe der Pipeline: t/k

■ Beschleunigung bei m auszuführenden Instruktionen:

- $m = 1$: Laufzeit mit Pipeline = $k \cdot t/k = t$
⇒ keine Beschleunigung
- $m = 2$: Laufzeit mit Pipeline = $t + t/k = (k+1)t/k$
⇒ Beschleunigung um Faktor $2k/(k+1)$
- $m \geq 1$: Laufzeit mit Pipeline = $t + (m-1)t/k$,
Laufzeit ohne Pipeline = mt
- ⇒ Beschleunigung um $\frac{mt}{t + (m-1) \cdot \frac{t}{k}} = \frac{mk}{m+k-1} = k - \frac{k(k-1)}{m+k-1}$

JR - RA - SS02

Kap. 3.4

9

Pipelining: Speedup (2)

■ Beschleunigung bei m auszuführenden Instruktionen:

- $m \geq 1$: Laufzeit mit Pipeline = $t + (m-1)t/k$,
- Laufzeit ohne Pipeline = mt
- ⇒ Beschleunigung um $\frac{mt}{t + (m-1) \cdot \frac{t}{k}} = \frac{mk}{m+k-1} = k - \frac{k(k-1)}{m+k-1}$

- Ergebnis: Für $m \gg k$ nähert sich der Speedup also der Anzahl k der Pipelinestufen.

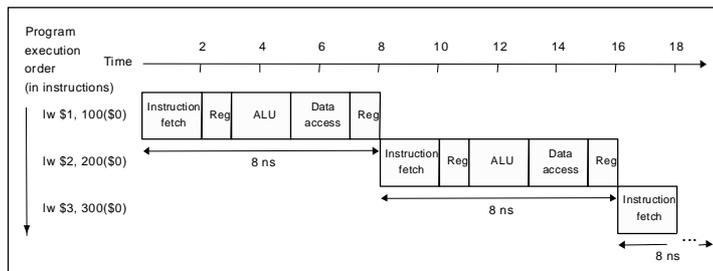
- Es wurde vorausgesetzt, daß sich die Ausführung der Befehle ohne weiteres „verzahnen“ läßt. Dies ist in der Praxis nicht ohne weiteres der Fall !

JR - RA - SS02

Kap. 3.4

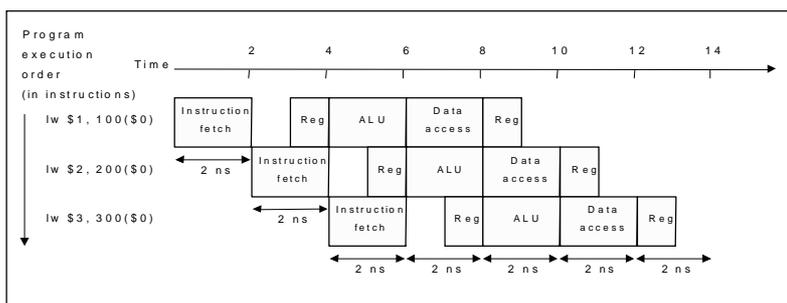
10

Programmablauf ohne Pipelining



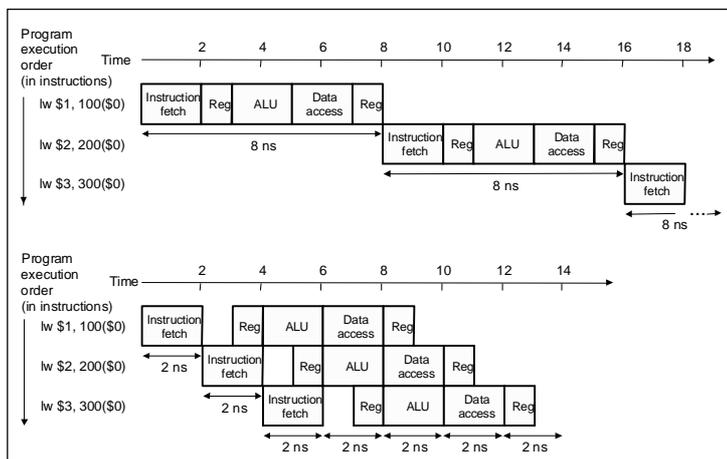
Instruction (class)	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
lw	2ns	1ns	2ns	2ns	1ns	8ns
sw	2ns	1ns	2ns	2ns		7ns
R-type	2ns	1ns	2ns		1ns	6ns
branch	2ns	1ns	2ns			5ns

Programmablauf mit Pipelining



Instruction (class)	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
lw	2ns	1ns	2ns	2ns	1ns	8ns
sw	2ns	1ns	2ns	2ns		7ns
R-type	2ns	1ns	2ns		1ns	6ns
branch	2ns	1ns	2ns			5ns

Pipelining versus Non-Pipelining



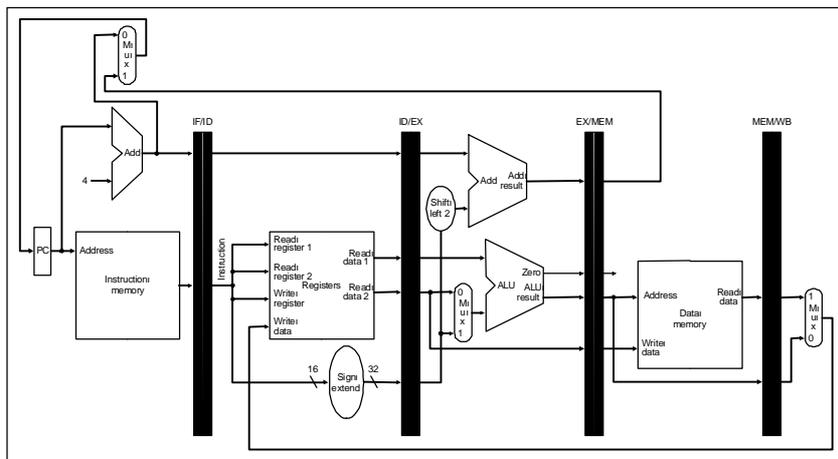
JR - RA - SS02

Kap. 3.4

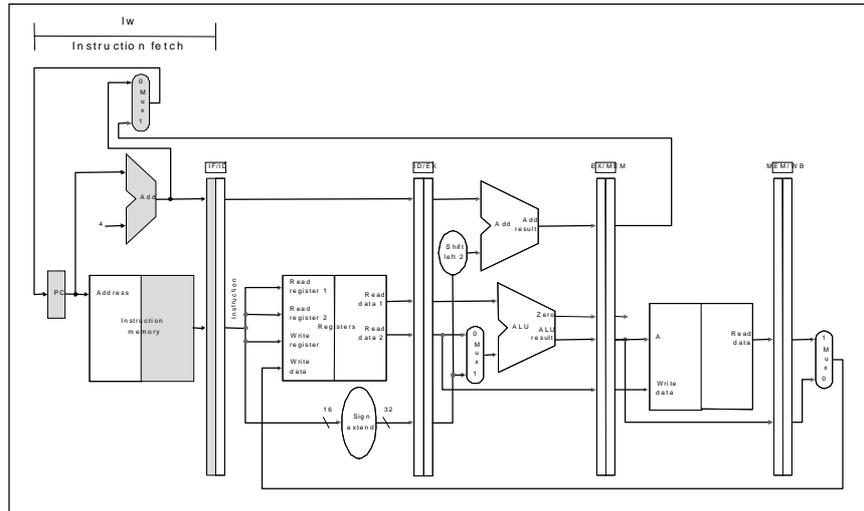
13

Hardwarerealisierung der Unterteilung

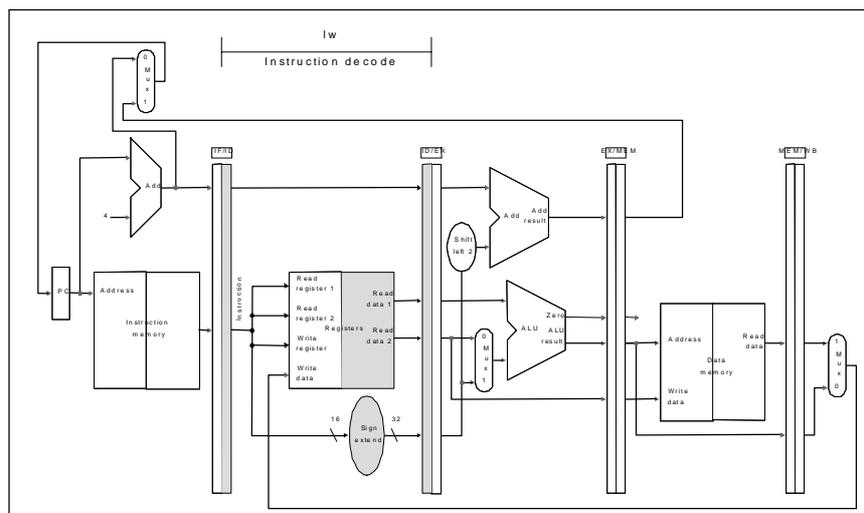
- Füge Registerbänke ein, um die Phasen hardwaremässig zu trennen



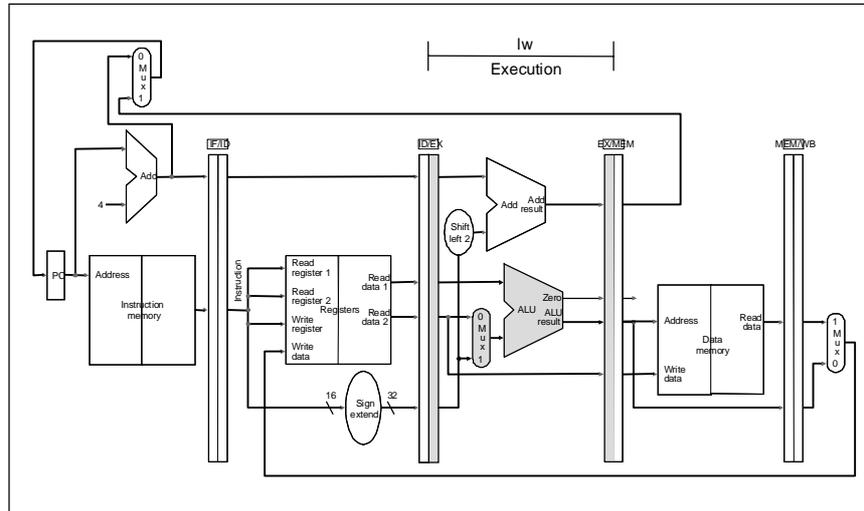
Abarbeitung einer Instruktion: Instruction fetch



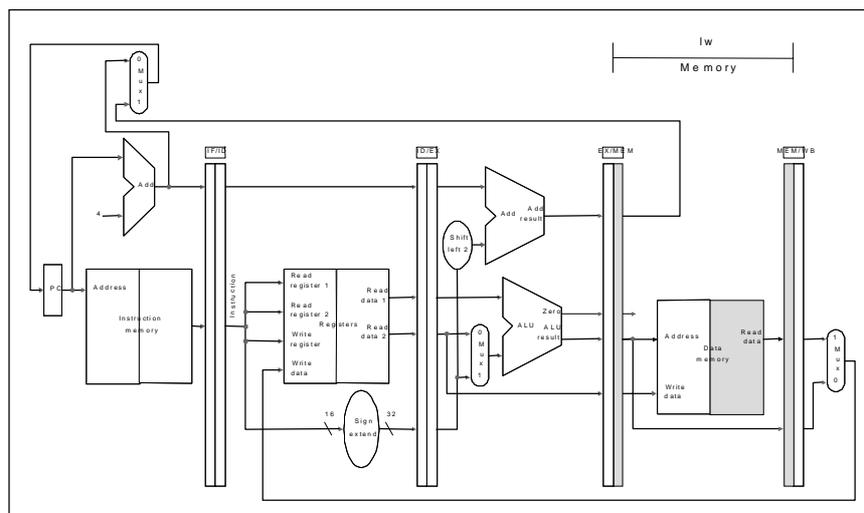
Abarbeitung einer Instruktion: Instruction decode and Register file read



Abarbeitung einer Instruktion: Execution



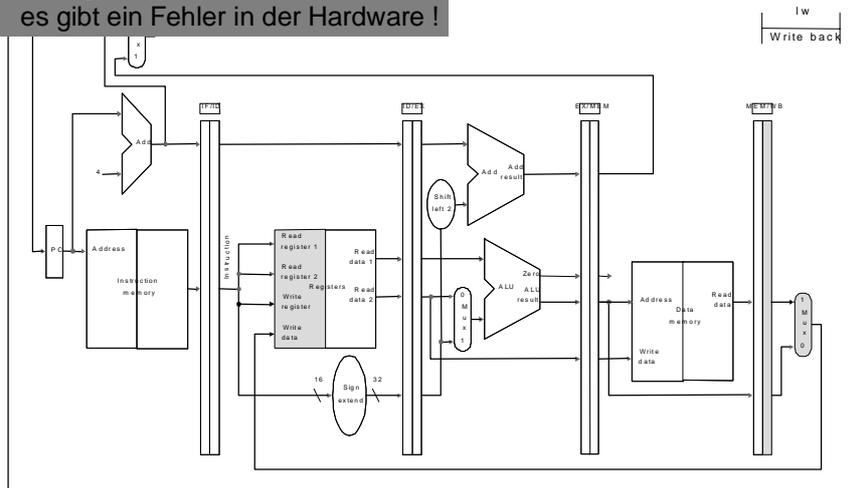
Abarbeitung einer Instruktion: Memory Access



Abarbeitung einer Instruktion: Write back

Write back

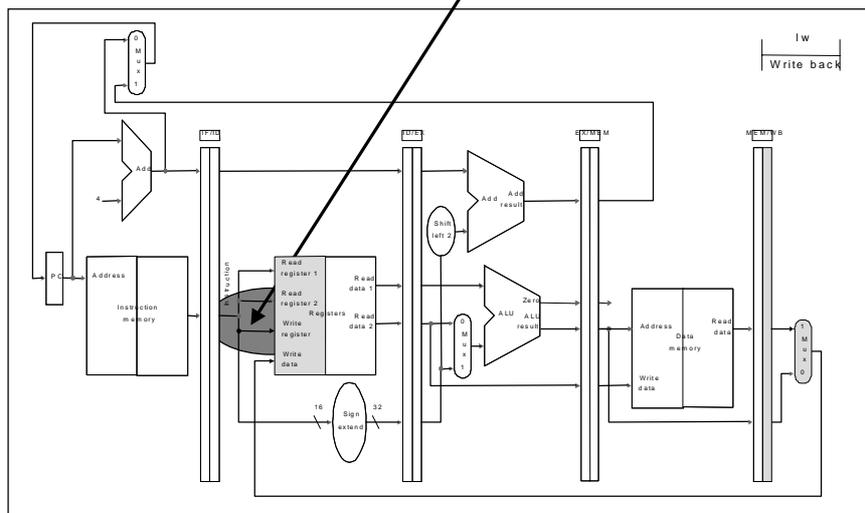
Achtung:
es gibt ein Fehler in der Hardware !



Fehler in der Hardware

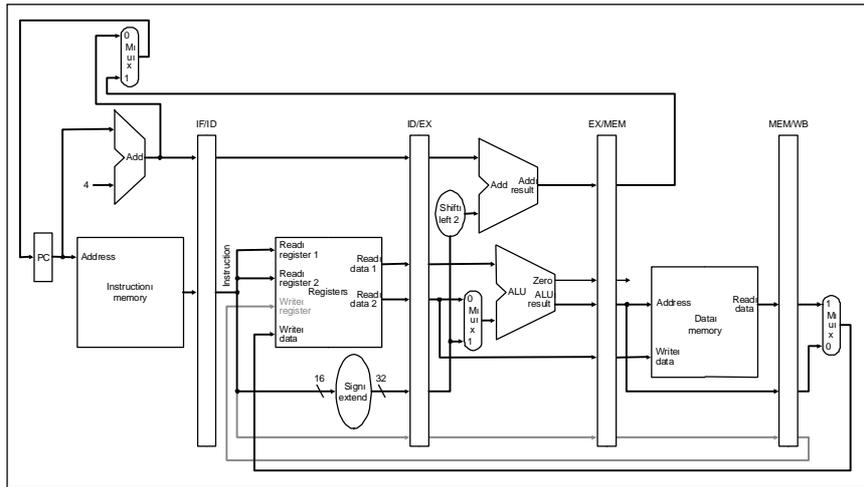
Der LOAD-Befehl schreibt das Ergebnis in das falsche Register!

Die angelegte Registernummer gehört zu der Instruktion, die gerade erst in die Pipeline geschoben wurde!

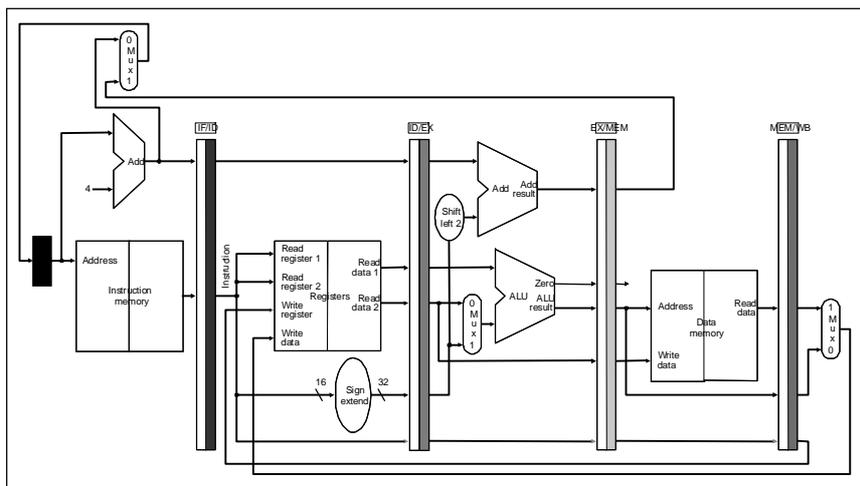


Korrigierte Hardware

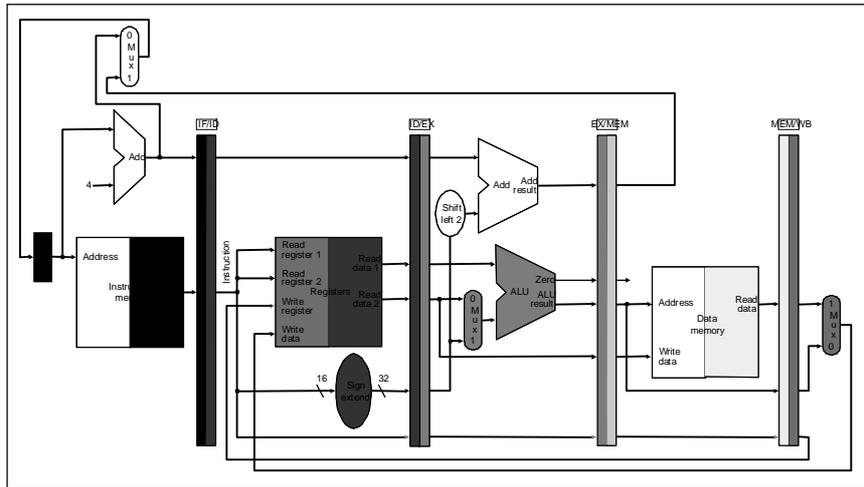
Die Registernummer muss über die ganzen Phasen in den entsprechenden Pipeline-Registern gettet werden



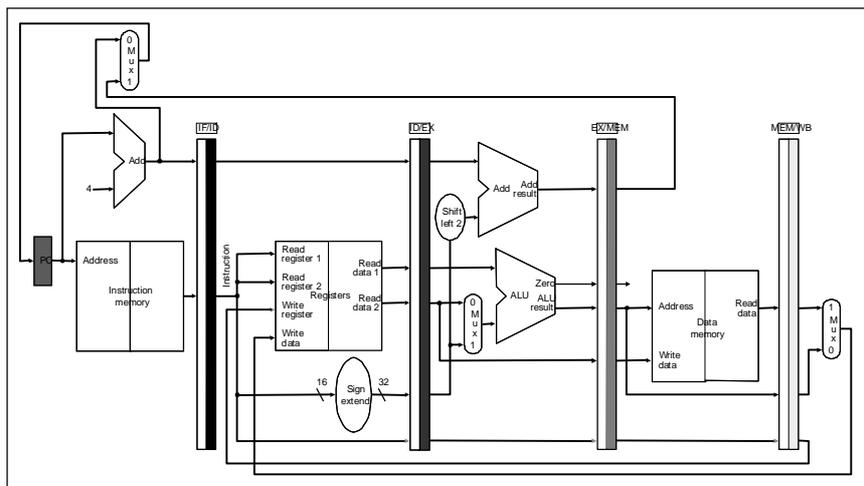
Das Pipelining-Prinzip (1)



Das Pipelining-Prinzip (2)

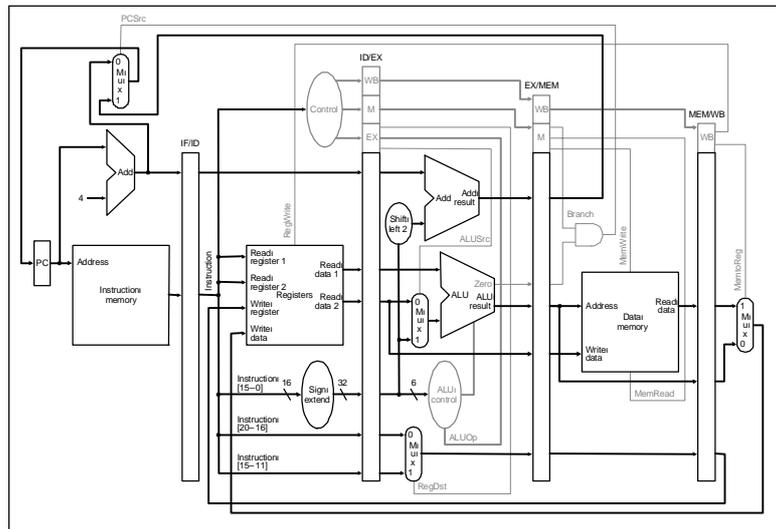


Das Pipelining-Prinzip (3)



Hardware mit Kontrolllogik

Die Kontrollsignale zu einer Instruktion müssen ebenfalls in den Pipeline-registern gerettet werden, bis sie benutzt werden können.



Probleme beim Pipelining: Datenabhängigkeit

- Betrachte folgendes Maschinenprogramm: **das erwartete Resultat**

```

sub $2 , $1, $3      # $2 = 23-3 = 20
and $12, $2, $5      # $12 = 20 and 7 = 4
or  $13, $6, $2      # $13 = 3 or 20 = 23
add $14, $2, $2      # $14 = 20+20= 40
sw  $15, 100($2)     # save $15 to 100(20)

```

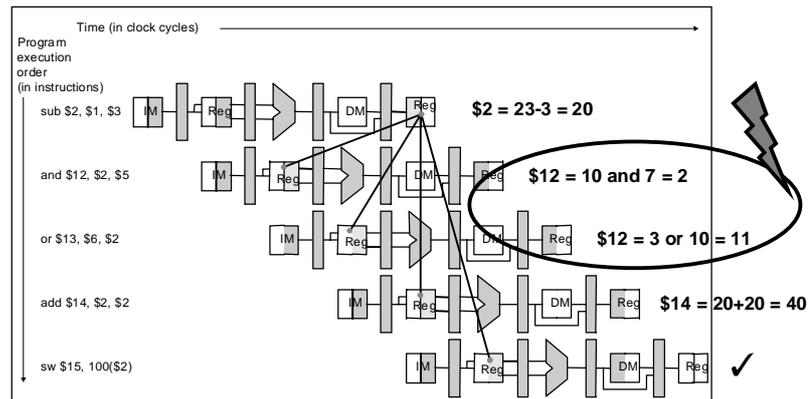
- Nehme an, am Anfang ist Register
 - \$1 mit 23,
 - \$2 mit 10,
 - \$3 mit 3,
 - \$5 mit 7,
 - \$6 mit 3

belegt.

Berechnung durch die Pipeline

Anfangsbelegung

\$1 mit 23
\$2 mit 10
\$3 mit 3
\$5 mit 7
\$6 mit 3



JR - RA - SS02

Kap. 3.4

27

Datenabhängigkeiten

■ Definition:

Gegeben zwei Anweisungen S_1, S_2 .

DEF_i sei die Menge der Variablen auf die S_i schreibend zugreift, USE_i sei die Menge der Variablen auf die S_i lesend zugreift.

S_1, S_2 heißen datenunabhängig, wenn

$$(DEF_1 \cap USE_2) \cup (DEF_2 \cap USE_1) \cup (DEF_1 \cap DEF_2) = \emptyset$$

JR - RA - SS02

Kap. 3.4

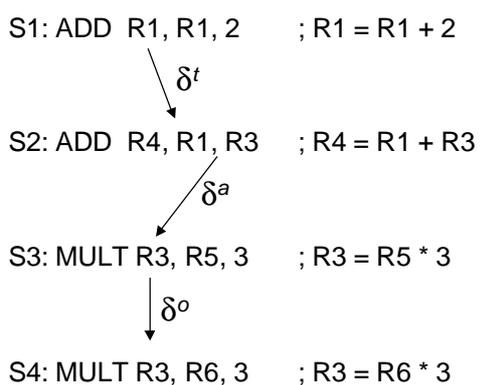
28

Datenabhängigkeiten

Wird S_1 vor S_2 ausgeführt dann besteht eine

- **True Dependence** (RAW) von S_1 nach S_2 wenn $DEF_1 \cap USE_2 \neq \emptyset$ gilt (geschrieben: $S_1 \xrightarrow{\delta^t} S_2$)
- **Anti Dependence** (WAR) von S_1 nach S_2 wenn $USE_1 \cap DEF_2 \neq \emptyset$ gilt (geschrieben: $S_1 \xrightarrow{\delta^a} S_2$)
- **Output Dependence** (WAW) von S_1 nach S_2 wenn $DEF_1 \cap DEF_2 \neq \emptyset$ gilt (geschrieben: $S_1 \xrightarrow{\delta^o} S_2$)

Datenabhängigkeiten: Beispiel



Datenabhängigkeiten

Datenabhängigkeiten führen zu Hazards (Hemmnissen) die ohne Behandlung zu fehlerhaften Ergebnissen führen

Output- und Anti Dependencies

- bereiten bei einfachen Pipelines keine Probleme
- machen erst bei Techniken wie superskalärer Ausführung oder *out-of-order execution* Schwierigkeiten
- können zum Teil vom Compiler durch Variablenumbenennungen und Variablenkopien aufgelöst werden
 - **Pseudo Dependencies**

Software-Lösung

Compiler müssen folgende Aufgaben übernehmen

- Entdecken von Daten Hazards im generierten Maschinenprogramm
- Einfügen von **N(o)-OP(eration)**-Instruktion

In unserem Beispiel:

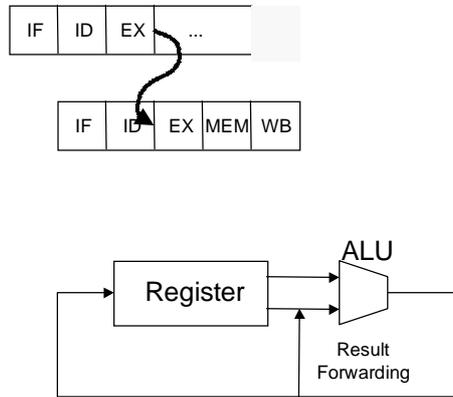
```

sub    $2, $1, $3
nop
nop
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)

```

→ Hardwarelösung

Result Forwarding/Register Bypassing



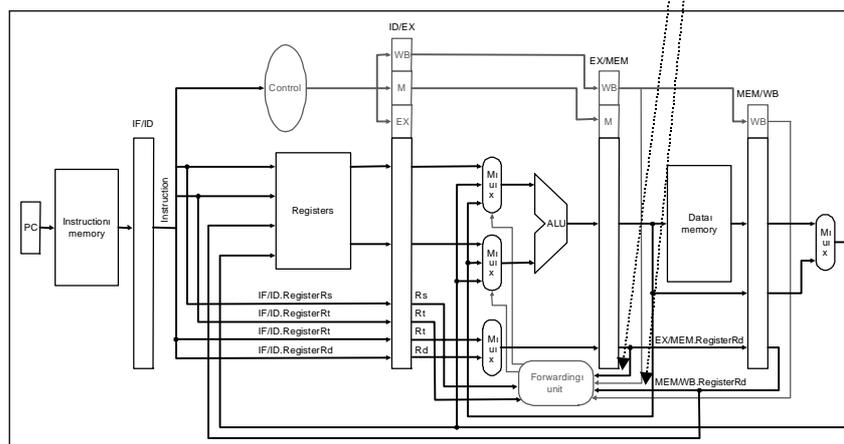
JR - RA - SS02

Kap. 3.4

33

Hardware-Lösung

Die Kontrolllogik benötigt die in den aktuellen Instruktionen benutzten Registernummern, um die Existenz und Art eines Daten Hazards zu erkennen.



JR - RA - SS02

Kap. 3.4

34

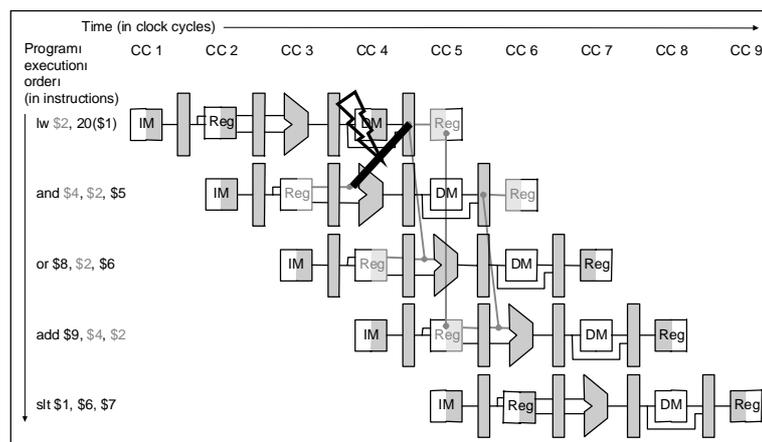
Vollständig ohne Stoppen der Pipeline geht es jedoch nicht !

Betrachte folgendes Programm

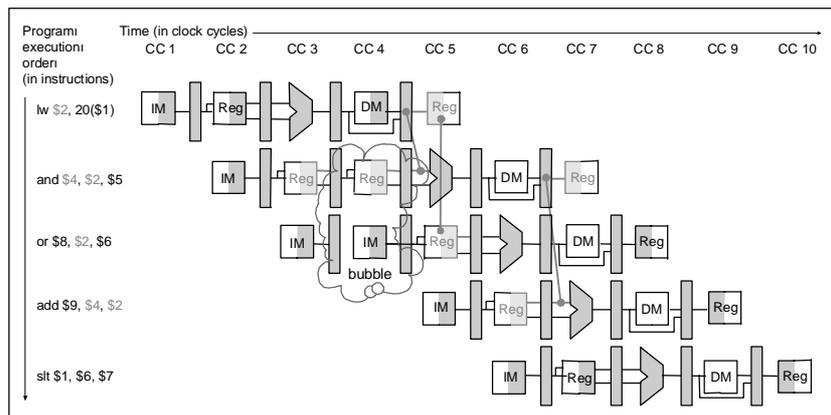
```
lw    $2, 20($1)
and   $4, $2, $5
or    $8, $2, $6
add   $9, $4, $2
slt   $1, $6, $7
```

Daten Hazard kann nicht hardwaremässig repariert werden, da zu dem Zeitpunkt, an dem \$2 von der AND-Instruktion gelesen werden soll, der neue Wert von \$2 nicht geladen ist !

Illustration



Einfügen eines Wartezyklus



JR - RA - SS02

Kap. 3.4

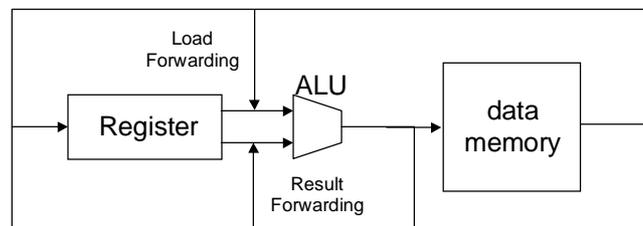
37

Forwarding/Bypassstechniken (2)

lw \$2, 20(\$1)

and \$4, \$2, \$5

- man kann diesen Hazard nicht beseitigen aber
- man kann seine Folgen abkürzen



JR - RA - SS02

Kap. 3.4

38

Probleme beim Pipelining: Control Hazards

Betrachte folgendes Maschinenprogramm:

```

40 beq $1, $3, 72
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
...
72 lw $4, 50($14)

```

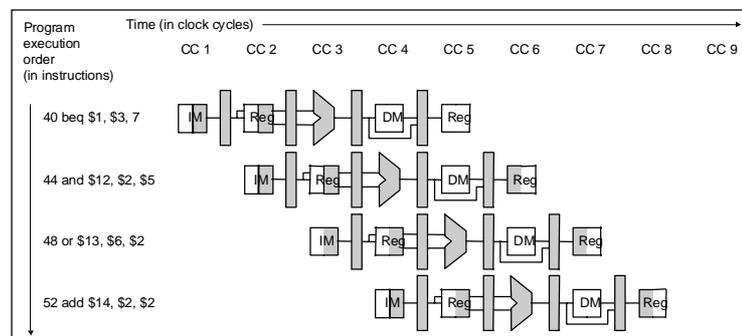
Frage

Welche Instruktion wird nach Instruktion 40 ausgeführt ?

Assume Branch not taken

In den einfachen Implementierungen wird angenommen, dass der Sprung nicht ausgeführt werden braucht !

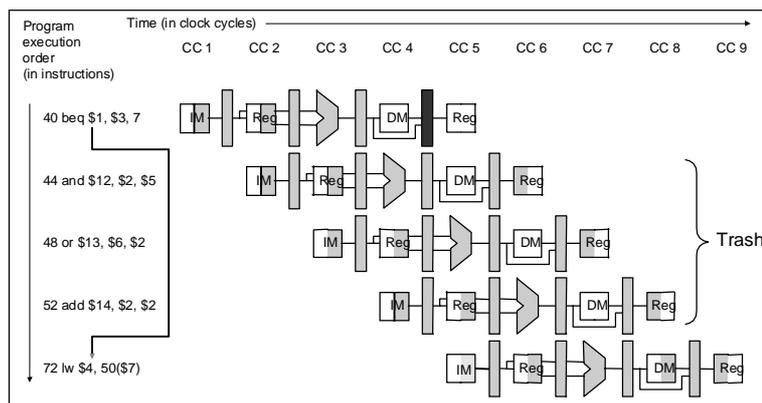
Die Folgeinstruktionen $\$pc+4$, $\$pc+8$ und $\$pc+12$ werden "auf Verdacht" (spekulativ) in die Pipeline geschoben !



Assume Branch not taken ff

Wird die Abzweigung doch genommen,

so muss die Pipeline geleert werden !



JR - RA - SS02

Kap. 3.4

41

Vermeidung von Kontrollhazards

■ Software

- Einfügen von NOPs durch den Compiler
- Delayed Branch Technik

■ Hardware

- „einfrieren“ der Pipeline (stalling)
- spekulative Ausführung
 - l mit fester Vorhersage (assume branch taken/ not taken)
 - l mit Sprungvorhersage
 - statisch (vom Compiler)
 - dynamisch

JR - RA - SS02

Kap. 3.4

42

Delayed Branch

- Externe Rechnerarchitektur definiert, daß der Befehl nach einem Sprungbefehl immer ausgeführt wird (unabhängig von der Sprungbedingung)
- Der Compiler muß durch Umordnung der Befehlssequenz einen datenunabhängigen Befehl finden und diesen hinter die Sprunganweisung setzen
 - ▮ Im schlimmsten Fall eine NOP-Anweisung
 - ▮ Untersuchungen zeigen, daß ein Befehl gut gefunden werden kann, zwei sind schon schwierig, drei fast unmöglich
- Beispiel: SPARC-Architektur: 1 delay slot

CLR R3
CMP R1, R2
BEQ 5000
NEXT

CMP R1, R2
BEQ 5000
CLR R3
NEXT

Sprungrichtungsvorhersage

- Statische Sprungvorhersage
 - ▮ ein Bit im Opcode des Sprungbefehls definiert, ob der Sprung spekulativ ausgeführt werden soll oder nicht
 - ▮ Schleifen: Wahrscheinlichkeit hoch, dass gesprungen wird
 - ▮ if-then-else: 2/3 der Ausführungen gehen in den ELSE-Teil
 - ▮ Vorhersagegenauigkeit: 55%-80%
- Dynamisch
 - ▮ Eine oder mehrere prozessorinterne Tabellen werden ständig aktualisiert und zur Erzeugung einer Vorhersage ausgewertet.
 - ▮ Tabellen mit Hilfe der Befehlsadresse des Sprungbefehls adressiert
 - ▮ Vorhersagegenauigkeit bis zu 97% (Literatur).

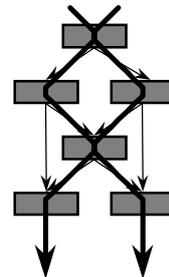
Informationsquellen für Sprungrichtungsvorhersage

■ Per-Adress Vorgeschichte

- Das Sprungverhalten jedes einzelnen Sprungs wird protokolliert und zur Vorhersage herangezogen.
- Der Sprung wird im wesentlichen wieder so vorhergesagt, wie er bei letzten n Ausführungen am häufigsten ausgeführt wurde.

■ Globale Vorgeschichte

- Pfad, den Programm durch die letzten m Kontrollblöcke beschriften hat, kann zusätzliche Information liefern.
- Wenn sich zwei Sprünge ähnlich verhalten (d.h. sie sind korreliert), kann dies erkannt und ausgenutzt werden.

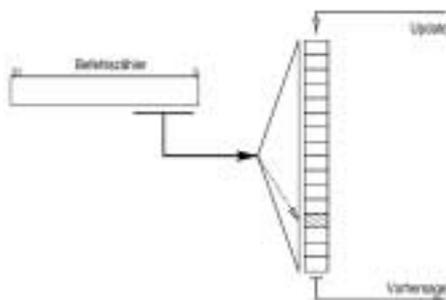


JR - RA - SS02

Kap. 3.4

45

Sprungrichtungsvorhersage durch eine branch history table



- Einfaches dynamisches Verfahren
- Ein Teil der Befehlsadresse wird als Hash-Index verwendet
- Kollisionen sind möglich (Verschiedene Sprünge mit dem selben niederwertigen Adressanteil.)
- Kollisionsbehandlung wäre nur durch Tags möglich
- Die gespeicherte Information wird nach jeder Sprungausführung erneuert
- Aus dieser Information wird bei Bedarf eine Vorhersage abgeleitet

JR - RA - SS02

Kap. 3.4

46

Vorhersageinformation

■ Letzte Sprungausführung - 1 Bit

- Vorhersage: Wie bei der letzten Ausführung
- Update: Ausgang der letzten Ausführung speichern

■ Muster der letzten n Sprungausführungen - n Bit

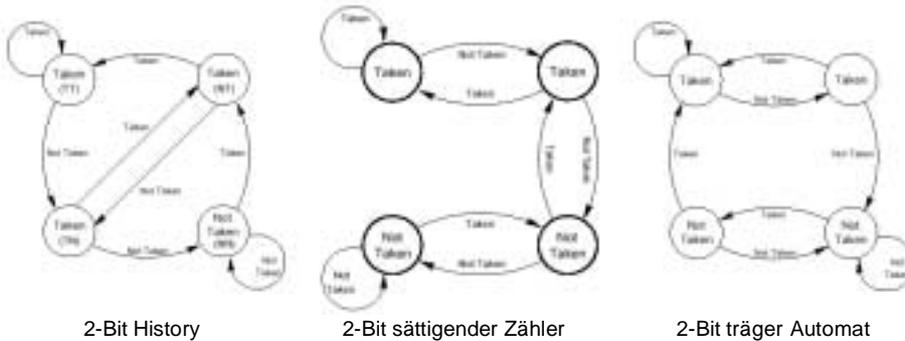
- Vorhersage: Aufgrund einer Heuristik, die aus Sprunghäufigkeiten gewonnen wird
- Update nach der Sprungausführung: Mit einem Schieberegister

...Vorhersageinformation

■ Endlicher Automat (Finite State Machine) - 2 oder 3 Bit

- Vorhersage: Vom Zustand abhängig, in dem sich der Automat befindet.
- Update nach der Sprungausführung: Gemäß einer Zustandsübergangsfunktion

Endliche Automaten für die Sprungvorhersage



- Zähler und träge Automaten erzielen in der Regel die besten Vorhersagen bei geringstem Platzbedarf
- Genauigkeit hängt auch vom Initialzustand der Automaten ab

JR - RA - SS02

Kap. 3.4

49

Sprungziel Speicher

- Berechnung der Sprungzieladresse kostet Zeit
 - ┆ Umsetzung der virtuellen in eine physikalische Adresse (Kap. 4)
 - ┆ Addition/Subtraktion der Sprungdistanz
- ➔ schneller Zugriff auf
 - ┆ die Zieladresse des Sprungs durch einen assoziativ Speicher BTAC (branch target address cache)
 - ┆ Adresse des Sprungbefehls → Adresse des Sprungziels
 - ┆ 1. Sprung löst Cachemiss aus, d.h. Speicher wird geladen
 - ┆ ab dem 2. Sprung: Sprungzieladresse steht zur Verfügung
 - ┆ die Befehle am Sprungziel durch einen assoziativ Speicher BTC (branch target cache):
 - ┆ Adresse des Sprungbefehls → k Befehle am Sprungziel
 - ┆ 1. Sprung löst Cachemiss aus, d.h. Speicher wird geladen
 - ┆ ab dem 2. Sprung: Befehle am Sprungziel stehen zur Verfügung

JR - RA - SS02

Kap. 3.4

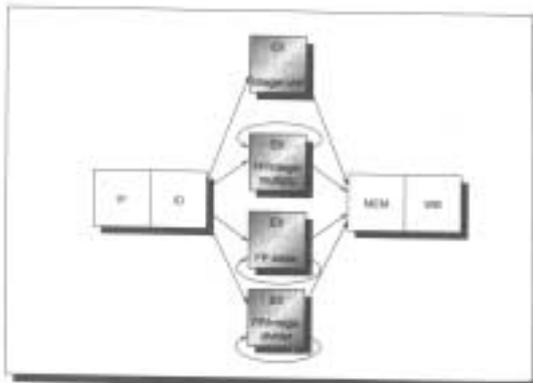
50

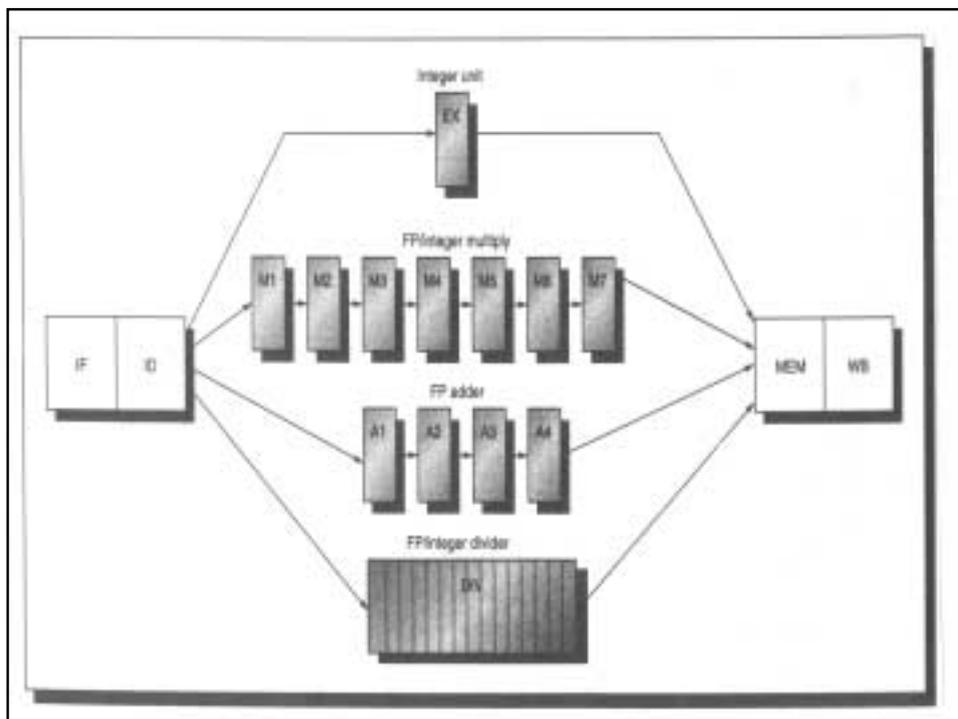
Zusammenfassung

- Beschleunigung um bis zu Faktor k durch Einsatz einer Pipeline (k = Anzahl der Pipeline-Stufen).
- Data- und control-Hazards verringern die Beschleunigung.
- Viele Möglichkeiten (z.T. in Hardware), Hazards zu vermeiden
 - Forwarding, Bypassing
 - Branch Prediction, BTC, ...
- Möglichkeiten für Code-optimierende Compiler, Hazards zu vermeiden:
 - Umstellen der Instruktionen eines Maschinenprogramms

Pipelining: Weiterentwicklungen

- Einführen mehrerer funktionaler Einheiten (FE) mit unterschiedlichen (gleichen) Aufgaben





Befehlszuordnung, -Ausführung, -Beendigung

- Werden die Befehle in ihrer Programmreihenfolge den Funktionalen Einheiten zugeordnet?
 - in-order issue
 - out-of-order issue
- Werden die Befehle in ihrer Reihenfolge ausgeführt?
 - in-order execution
 - out-of-order execution
- Wird die Befehlsausführung in der Reihenfolge beendet?
 - in-order completion
 - out-of-order completion
- Wird genau ein Befehl zu einem Zeitpunkt zugewiesen?
 - single-issue
 - multiple-issue (superskalare Ausführung)

Pipeline mit unterschiedlich langen EX-Phasen

MULT	IF	ID	M1	M2	M3	M4	M5	M6	MEM	WB
ADDD		IF	ID	A1	A2	A3	A4	MEM	WB	
LD			IF	ID	EX	MEM	WB			
SD				IF	ID	EX	MEM	WB		

- Befehle werden zu unterschiedlichen Zeitpunkten fertig
 - Datenabhängigkeiten werden in die Länge gezogen
 - trotz in-order-issue folgt out-of-order execution und out-of-order completion

Wer regelt, wann Befehle zugeordnet, gestartet und zurückgeschrieben werden?

Scheduling

Scheduling: ein Verfahren das entscheidet, wann eine Instruktion gestartet wird, seine Operanden liest und das Ergebnis zurückschreibt

Ziel: Umordnung von Instruktionen bei Daten- oder Kontrollabhängigkeiten

- **Static scheduling:** Aufgabe des Compilers
- **Dynamic scheduling:** Aufgabe der Hardware
- Kernidee: Instruktionen hinter *stalls* ausführen

```

DIVD  F0, F2, F4
ADDD  F10, F0, F8
SUBD  F12, F8, F14

```

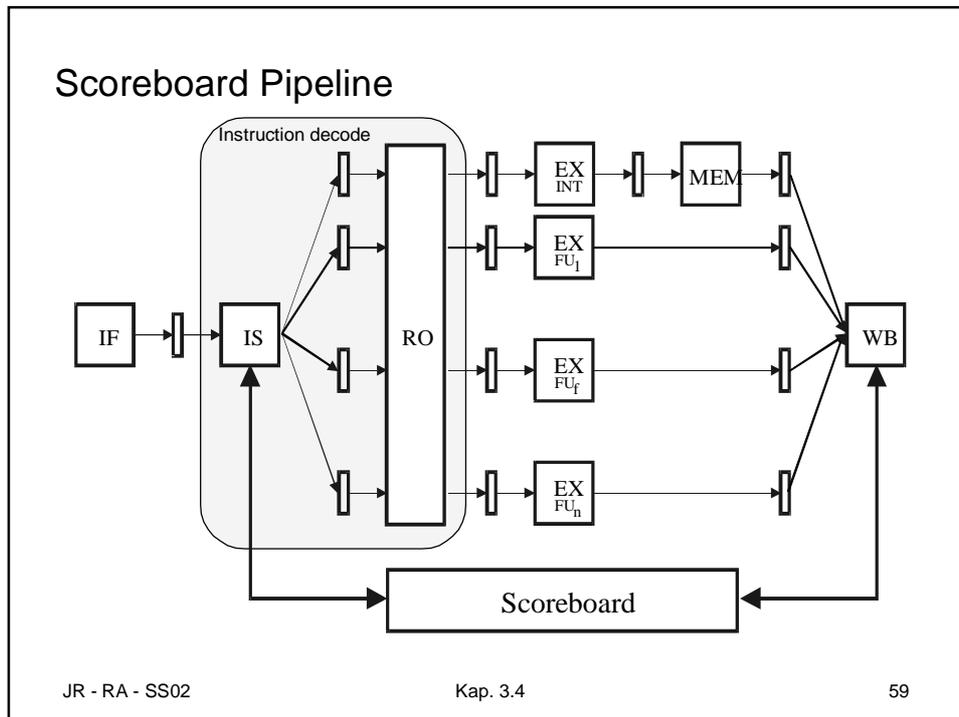
- SUBD hat keine Datenabhängigkeiten zu anderen Instruktionen
- zulassen von out-of-order execution ⇒ out-of-order completion
- ID-Phase prüft auf Struktur- und Datenabhängigkeiten

Dynamic Scheduling

- Dynamic scheduling arbeitet auch wenn *stalls* erscheinen, die zur Compilzeit nicht vorhersehbar sind, z.B. Cache Misses
- **Dynamic scheduling:**
 - **Control flow scheduling**, wird zentral bei der Dekodierung vorgenommen
z.B. scoreboarding in CDC 6600
 - **Dataflow scheduling**, wird verteilt in den funktionalen Einheiten zur Laufzeit ausgeführt. Befehle werden dekodiert und *reservation stations* zugeteilt, bis ihre Operanden verfügbar sind.
Tomasulo Algorithmus im IBM System/360 Model 91 Prozessor

Scoreboarding

- 1963 von Thornton für den CDC6600 eingeführt
- Ziel: Ausführung einer Instruktion pro Takt. Befehle so früh wie möglich ausführen
- Befehle werden out-of-order ausgeführt, wenn alle nötigen Ressourcen zur Verfügung stehen und keine Datenabhängigkeiten existieren
- Das Scoreboard ist eine HW-Einheit die registriert
 - welche Befehle sich in der Ausführung befinden
 - welche Funktionalen Einheiten aktuell benutzt werden
 - welche Register die Ergebnisse speichern werden
- Das Scoreboard führt zentral Konflikterkennung durch und kontrolliert dadurch den Programmfortschritt



Scoreboarding

- Die Dekodierphase wird aufgeteilt in,
 - **issue (IS) Phase (Zuordnungsphase):**
 Dekodieren, prüfen auf Struktur- und WAW-Hazards
 (Strukturhazards: eine Funktionale Einheit wird zu einem Zeitpunkt mehrfach benötigt)
 - **read operands (RO) Phase:**
 Warten bis keine Datenhazards existieren
 ⇒ Lesen der Operanden aus den Registern
- EX (Ausführung) und WB (Rückreiben) Phase sind um zusätzliche Registrierungs- und Überwachungsaufgaben erweitert

Scoreboarding (im Detail)

- **IS Phase:** wenn kein **struktureller** und kein **WAW Hazard** vorliegt,
 - das Scoreboard weist den Befehl der FE zu und aktualisiert seine internen Datenstrukturen
 - ansonsten wird die Zuordnung gestoppt, bis sich die Hazards aufgelöst haben (⇒ single issue and in-order issue!).
- **RO Phase:**
 - das Scoreboard überwacht die Verfügbarkeit der benötigten Operanden
 - wenn diese zur Verfügung stehen wird die FE aktiviert, um die Register zu lesen (⇒ kein Forwarding!!) und in der EX-Phase auszuführen
RAW Hazards werden dynamisch aufgelöst

Scoreboarding (im Detail)

- **EX Phase:**
Die FE startet die Ausführung (kann mehrere Zyklen dauern) und benachrichtigt das Scoreboard wenn die Ergebnisse zur Verfügung stehen (result ready flag).
- **WB Phase:**
Wenn das Scoreboard weiß, dass die FE die Ausführung beendet hat, prüft es auf **WAR Hazards** und stoppt die Registeraktualisierung falls nötig.
Andernfalls wird die FE aufgefordert, ihr Ergebnis in das Zielregister zu schreiben.

Scoreboard Implementierung

- **Register result status table (R):**
speichert, welche FE ein Ergebnis in welches Register schreibt, d.h. die Anzahl der Einträge in R ist gleich der Anzahl der Register
- **Functional unit status table (F):**
speichert die Bearbeitungsphase jeder Instruktion:
Phase: *Busy*, *RO*, *EX*, und *WB* für jede FE.
- **Instruction status table (auch F):** ein Eintrag pro FE:
 - Opcode der bearbeiteten Instruktion
 - Zielregister
 - Quellenregister
 - Verfügbarkeit der Quellenregister
 - sind Operanden nicht verfügbar, dann speichert die Tabelle die FE, die den Operanden gerade erzeugt
- Initial sind alle Einträge Null (leer).

JR - RA - SS02

Kap. 3.4

63

Scoreboarding Beispiel

registers

R	1	2	3	4	5	6

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

phase flags of the FUs

		Busy	RO	EX	WB
functional units	1 add				
	2 mul				
	3 div				

instruction information of the FUs

	Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2

cycle 0

all fields are initialized with 0
empty fields have the value 0

op	EX cycles
add	1
sub	1
mul	4
div	4

JR - RA - SS02

Kap. 3.4

64

Scoreboarding Beispiel

registers

R	1	2	3	4	5	6
	2					

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

functional units

F		Busy	RO	EX	WB
1 add					
2 mul	1				
3 div					

phase flags of the FUs

instruction information of the FUs

Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2
mul	1	3	1		5	1	

cycle 1

Scoreboarding Beispiel

registers

R	1	2	3	4	5	6
	2	1				

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

functional units

F		Busy	RO	EX	WB
1 add	1				
2 mul	1	1			
3 div					

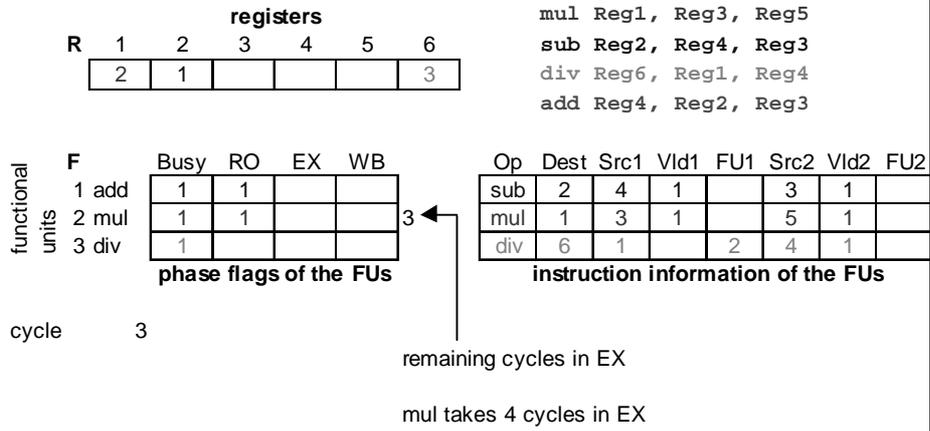
phase flags of the FUs

instruction information of the FUs

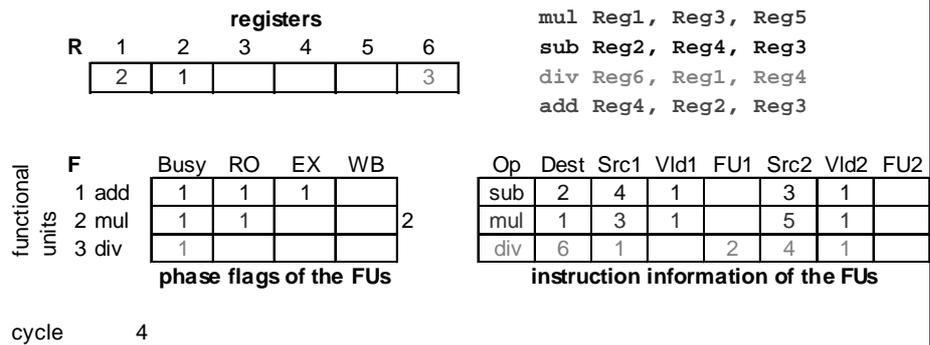
Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2
sub	2	4	1		3	1	
mul	1	3	1		5	1	

cycle 2

Scoreboarding Beispiel



Scoreboarding Beispiel



Scoreboarding Beispiel

registers

R	1	2	3	4	5	6
	2					3

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

phase flags of the FUs

functional units	F	Busy	RO	EX	WB	
	1 add		1	1	1	
	2 mul	1	1			1
	3 div	1				

instruction information of the FUs

Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2
mul	1	3	1		5	1	
div	6	1		2	4	1	

cycle 5

Scoreboarding Beispiel

registers

R	1	2	3	4	5	6
	2			1		3

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

phase flags of the FUs

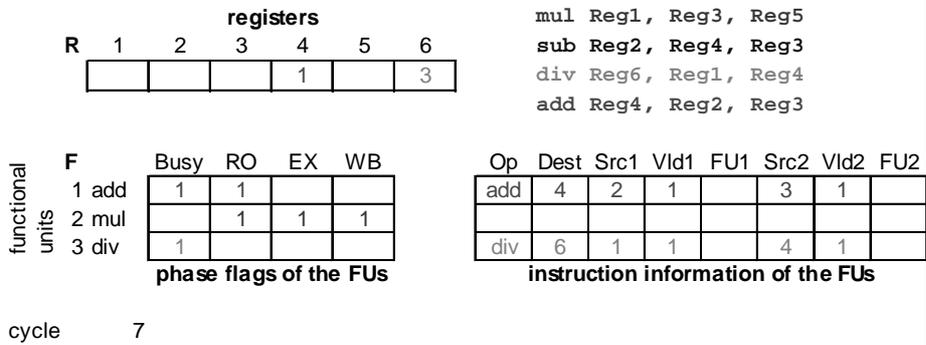
functional units	F	Busy	RO	EX	WB	
	1 add	1				
	2 mul	1	1	1		0
	3 div	1				

instruction information of the FUs

Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2
add	4	2	1		3	1	
mul	1	3	1		5	1	
div	6	1		2	4	1	

cycle 6

Scoreboarding Beispiel

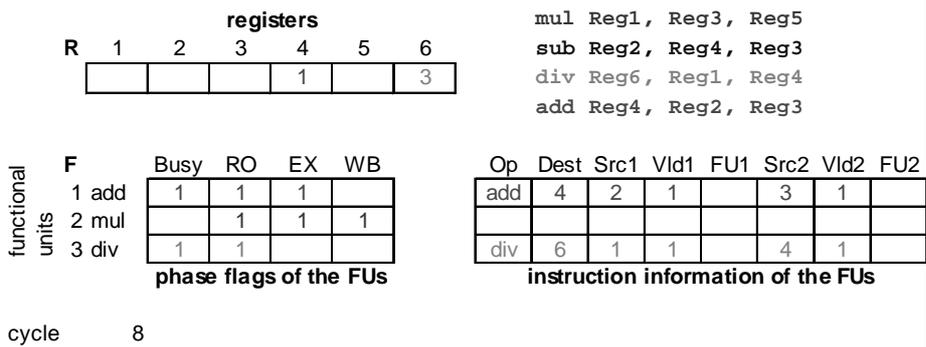


JR - RA - SS02

Kap. 3.4

71

Scoreboarding Beispiel

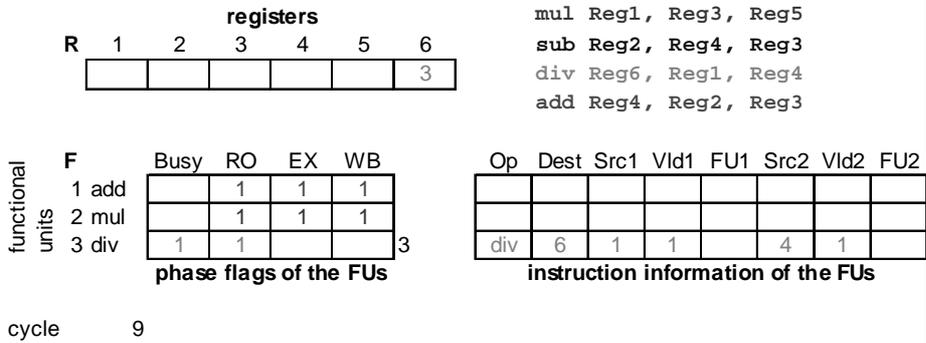


JR - RA - SS02

Kap. 3.4

72

Scoreboarding Beispiel

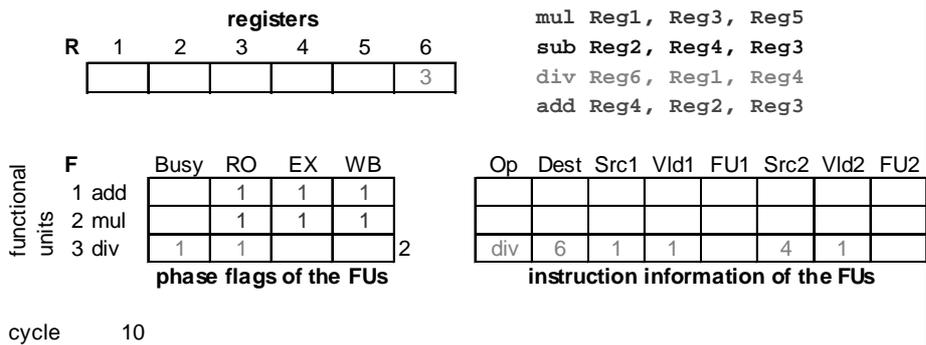


JR - RA - SS02

Kap. 3.4

73

Scoreboarding Beispiel



JR - RA - SS02

Kap. 3.4

74

Scoreboarding Beispiel

registers

R	1	2	3	4	5	6
						3

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

functional units

F	Busy	RO	EX	WB
1 add		1	1	1
2 mul		1	1	1
3 div	1	1		

phase flags of the FUs

Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2
div	6	1	1		4	1	

instruction information of the FUs

cycle 11

Scoreboarding Beispiel

registers

R	1	2	3	4	5	6
						3

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

functional units

F	Busy	RO	EX	WB
1 add		1	1	1
2 mul		1	1	1
3 div	1	1	1	

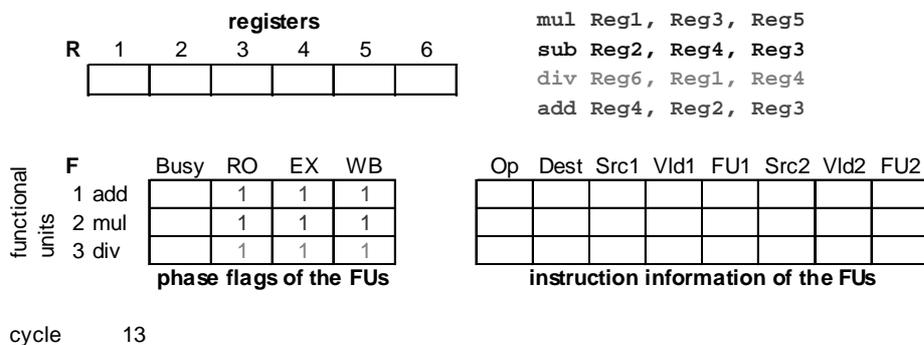
phase flags of the FUs

Op	Dest	Src1	Vld1	FU1	Src2	Vld2	FU2
div	6	1	1		4	1	

instruction information of the FUs

cycle 12

Scoreboarding Beispiel



JR - RA - SS02

Kap. 3.4

77

Scoreboard Zusammenfassung

■ Vorteil:

- verwaltet mehrerer FEs
- out-of-order Ausführung von Mehrtaktoperationen
- erkennen aller Datenabhängigkeiten (RAW, WAW, WAR)

■ Scoreboard Beschränkungen:

- single issue Ausführung (läßt sich auf multiple-issue erweitern)
- in-order issue
- kein Registerumbenennen ⇒ Pseudoabhängigkeiten können zu Blockierungen führen
- keine Forwardinghardware ⇒ alle Ergebnisse durch Register

■ generelle Beschränkungen

- Beschränkung der FEs führt zu strukturellen Hazards
- Grad der Parallelität im Programm (auf Instruktionsebene)

JR - RA - SS02

Kap. 3.4

78

Zur Erinnerung

Mehrzykleninstruktionen führen zu out-of-order Ausführung

■ Control flow scheduling,

zentral während des Dekodierens ausgeführt:
 ⇒ Scoreboarding Technik implementiert im CDC 6600 Prozessor

■ Dataflow scheduling,

wird verteilt in den FEs ausgeführt. Befehle werden dekodiert und anschließend den *reservation stations* zugeordnet, wo sie auf ihre Operanden warten.

⇒ Tomasulo Algorithmus

- benutzt Forwarding und Registerrenaming
- ist single and in-order issue
- Basis aller modernern Superskalarprozessoren

Register Renaming

- Eine Namensabhängigkeit entsteht, wenn zwei Instruktionen $Inst_1$ und $Inst_2$ die selben Register benutzen, aber keine Daten zwischen den Instruktionen ausgetauscht werden
- Wird ein Registeroperand umbenannt, so dass $Inst_1$ und $Inst_2$ keine Abhängigkeit mehr haben, dann können beide Instruktionen simultan ausgeführt oder umgeordnet werden
- Die Technik die Namensabhängigkeiten eliminieren um WAR und WAW Hazards zu vermeiden, heißt **register renaming**
- Register renaming kann statisch (vom Compiler) oder dynamisch (von der Hardware) ausgeführt werden
- Tomasulos Algorithmus führt register renaming durch die Hardware aus

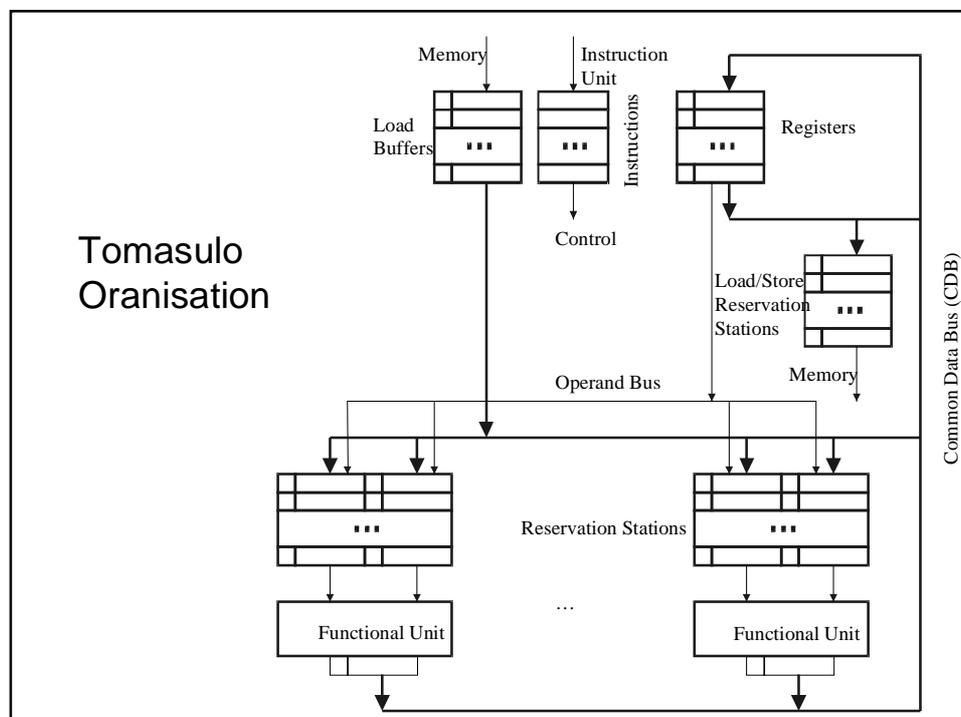
Tomasulo Algorithm

- Entwickelt für IBM 360/91 (1967 ca. 3 years nach CDC 6600)
- Hazard Erkennung und Ausführungskontrolle sind verteilt in den FEs
- **Reservation stations (RS)** vor jeder FE kontrollieren den Beginn der Instruktionausführung
- **Common Data Bus** verteilt alle Ergebnisse an alle reservation stations und an die Register
- Speicherzugriffe werden von speziellen FEs durchgeführt
- Jedes Register hat zusätzliche Kontrollflags

JR - RA - SS02

Kap. 3.4

81

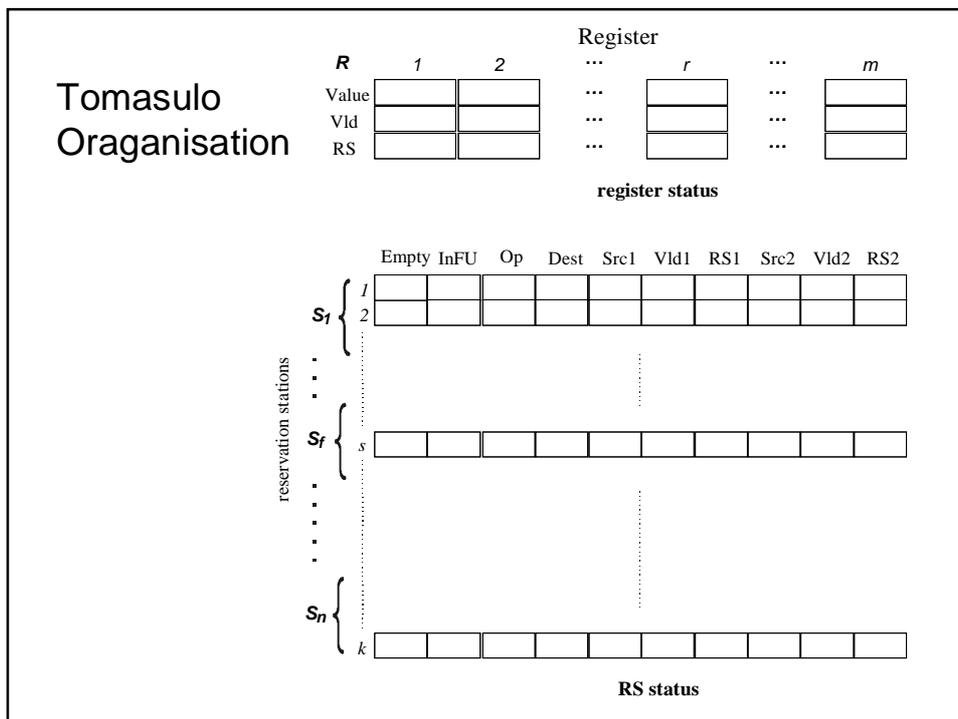


Reservation Stations

- Jede FE hat eine oder mehrere reservation stations
- Die reservation station beinhalten:
 - Befehle, die bereits zugewiesen wurden und auf ihre Ausführung in der FE warten
 - die Operanden des Befehls, soweit bereits berechnet (andernfalls wird die Quelle/FE des Operanden gespeichert)
 - ⇒ WAR Hazards werden umgangen, da die Werte des Registers bereits in der reservation station liegen auch wenn eine andere Operation (out-of-order) dieses Register überschreibt
 - ⇒ WAW Hazards werden umgangen, da Referenzen auf reservation stations als Tags auf dem CDB benutzt werden, anstatt Referenzen auf konkrete Register

Reservation Station Einträge

- Empty*: zeigt an, ob die reservation station leer ist
- InFU*: zeigt an, ob die FE die gespeicherte Instruktion gerade ausführt
- Op*: Operation, die von der FE auszuführen ist
- Dest*: Tag des reservierten Registers
- Src1, Src2*: Werte der Quelloperanden
- RS1, RS2*: Tag der reservation station, die den Operanden erzeugt
- Val1, Val2*: Valid Flags zeigen an, ob die Operanden gültig sind



CDB und Reservation Stations

- Nach der Ausführung eines Befehls einer RS wird ein Ergebnistoken generiert und über den *common data bus* (CDB) an die Register und direkt an alle RS geschickt (forwarding)
- Alle RS überwachen ständig die Daten auf dem CDB (snooping)
- Ein Ergebnis auf dem CDB wird in alle RS kopiert, die darauf warten
- CDB ermöglicht simultanes Laden von Ergebnissen sobald diese zur Verfügung stehen (Datenflußprinzip)

Drei Phasen des Tomasulo Algorithmus

1. Issue: Instruktion laden (von der instruction queue)

Wenn RS frei, dann wird die Instruktion zugewiesen und Operanden aus den Registern geladen (falls möglich)
 → In-order issue!

2. Execution: Ausführen des Befehls (EX)

Wenn beide Operanden bereit stehen, dann wird die Operation an die FE geleitet und dort ausgeführt (dispatch), wenn nicht, dann wird der CDB abgehört um Operanden abzufangen
 → Out-of-order dispatch und out-of-order execution!

3. Write result: Beendigung der Ausführung (WB)

Lege Ergebnis auf den CDB
 Freigabe der RS.

Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		1	1	1	1	1	1
RS		0	0	0	0	0	0

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	1									
	S _{add} 2	1									
	S _{mul} 3	1									
	S _{div} 4	1									



We assume:
mul and **div** need 4 EX cycles,
sub and **add** need 1 EX cycle.

Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		0	1	1	1	1	1
RS		3	0	0	0	0	0

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S_{add}	1									
		2									
	S_{mul}	3	0	0	mul	1	(R3)	1	0	(R5)	1
	S_{div}	4	1								

cycle 1



Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		0	0	1	1	1	1
RS		3	1	0	0	0	0

```
mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S_{add}	1	0	0	sub	2	(R4)	1	0	(R3)	1
		2	1								
	S_{mul}	3	0	1	mul	1	(R3)	1	0	(R5)	1
	S_{div}	4	1								

cycle 2



Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		0	0	1	1	1	0
RS		3	1	0	0	0	4

mul Reg1, Reg3, Reg5
 sub Reg2, Reg4, Reg3
 div Reg6, Reg1, Reg4
 add Reg4, Reg2, Reg3

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	0	1	sub	2	(R4)	1	0	(R3)	1	0
	S _{add} 2	1									
	S _{mul} 3	0	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div} 4	0	0	div	6		0	3	(R4)	1	0

cycle 3



remaining cycles in FU

Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	(R4)-(R3)	(R3)	-	(R5)	-
Vld		0	1	1	0	1	0
RS		3	0	0	2	0	4

mul Reg1, Reg3, Reg5
 sub Reg2, Reg4, Reg3
 div Reg6, Reg1, Reg4
 add Reg4, Reg2, Reg3

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	1	1	sub	2	(R4)	1	0	(R3)	1	0
	S _{add} 2	0	0	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul} 3	0	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div} 4	0	0	div	6		0	3	(R4)	1	0

cycle 4



sub schreibt auf CDB und gibt RS1 frei;
add wird RS2 zugewiesen und lädt den ersten Operanden im gleichen Zyklus vom CDB

Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	(R4)-(R3)	(R3)	-	(R5)	-
Vld		0	1	1	0	1	0
RS		3	0	0	2	0	4

mul Reg1, Reg3, Reg5
 sub Reg2, Reg4, Reg3
 div Reg6, Reg1, Reg4
 add Reg4, Reg2, Reg3

register status

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2	
reservation stations	S _{add}	1	1	sub	2	(R4)	1	0	(R3)	1	0	
		2	0	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul}	3	0	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div}	4	0	0	div	6		0	3	(R4)	1	0

RS status

cycle 5

CDB	token.tag
	token.data

Tomasulo Scheduling

		registers					
R		1	2	3	4	5	6
Value		-	(R4)-(R3)	(R3)	(R4)-(R3)+(R3)	(R5)	-
Vld		0	1	1	1	1	0
RS		3	0	0	0	0	4

mul Reg1, Reg3, Reg5
 sub Reg2, Reg4, Reg3
 div Reg6, Reg1, Reg4
 add Reg4, Reg2, Reg3

register status

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2	
reservation stations	S _{add}	1	1	sub	2	(R4)	1	0	(R3)	1	0	
		2	1	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul}	3	0	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div}	4	0	0	div	6		0	3	(R4)	1	0

RS status

cycle 6

CDB	token.tag	2
	token.data	(R4)-(R3)+(R3)

add and **mul** gleichzeitig fertig und konkurrieren um den CDB; **add** schreibt auf CDB, **mul** muß warten

WAR-Hazard wird automatisch aufgelöst: **add** schreibt Reg4 bevor **div** die Ausführung beginnt, allerdings hat **div** bereits den alten Registerwert in der RS gesichert

Tomasulo Scheduling

		registers									
R		1	2	3	4	5	6				
Value		(R3)*(R5)	(R4)-(R3)	(R3)	(R4)- (R3)+(R3)	(R5)	-				
Vld		1	1	1	1	1	0				
RS		0	0	0	0	0	4				

register status

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	1	1	sub	2	(R4)	1	0	(R3)	1	0
	2	1	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul} 3	1	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div} 4	0	0	div	6	(R3)*(R5)	1	0	(R4)	1	0

RS status

cycle 7

CDB	token.tag	3
	token.data	(R3)*(R5)

JR - RA - SS02

Kap. 3.4

95

Tomasulo Scheduling

		registers									
R		1	2	3	4	5	6				
Value		(R3)*(R5)	(R4)-(R3)	(R3)	(R4)- (R3)+(R3)	(R5)	-				
Vld		1	1	1	1	1	0				
RS		0	0	0	0	0	4				

register status

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	1	1	sub	2	(R4)	1	0	(R3)	1	0
	2	1	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul} 3	1	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div} 4	0	1	div	6	(R3)*(R5)	1	0	(R4)	1	0

RS status

cycle 8

CDB	token.tag	
	token.data	

JR - RA - SS02

Kap. 3.4

96

Tomasulo Scheduling

		registers									
R		1	2	3	4	5	6				
Value		(R3)*(R5)	(R4)-(R3)	(R3)	(R4)- (R3)+(R3)	(R5)	-				
Vld		1	1	1	1	1	0				
RS		0	0	0	0	0	4				

register status

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add}	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul}	3	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div}	4	0	div	6	(R3)*(R5)	1	0	(R4)	1	0

RS status

cycle 9

CDB	token.tag token.data
------------	-------------------------

JR - RA - SS02

Kap. 3.4

97

Tomasulo Scheduling

		registers									
R		1	2	3	4	5	6				
Value		(R3)*(R5)	(R4)-(R3)	(R3)	(R4)- (R3)+(R3)	(R5)	-				
Vld		1	1	1	1	1	0				
RS		0	0	0	0	0	4				

register status

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add}	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul}	3	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div}	4	0	div	6	(R3)*(R5)	1	0	(R4)	1	0

RS status

cycle 10

CDB	token.tag token.data
------------	-------------------------

JR - RA - SS02

Kap. 3.4

98

Tomasulo Scheduling

		registers								
R		1	2	3	4	5	6			
Value		(R3)*(R5)	(R4)-(R3)	(R3)	$\frac{(R4)}{(R3)+(R3)}$	(R5)	-			
Vld		1	1	1	1	1	0			
RS		0	0	0	0	0	4			

mul Reg1, Reg3, Reg5
 sub Reg2, Reg4, Reg3
 div Reg6, Reg1, Reg4
 add Reg4, Reg2, Reg3

		register status									
		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	1	1	sub	2	(R4)	1	0	(R3)	1	0
	S _{add} 2	1	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul} 3	1	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div} 4	0	1	div	6	$\frac{(R3)*(R5)}{(R3)+(R5)}$	1	0	(R4)	1	0

cycle 11

CDB	token.tag	
	token.data	

Tomasulo Scheduling

		registers								
R		1	2	3	4	5	6			
Value		(R3)*(R5)	(R4)-(R3)	(R3)	$\frac{(R4)}{(R3)+(R3)}$	(R5)	$\frac{(R3)*(R5)}{(R4)}$			
Vld		1	1	1	1	1	1			
RS		0	0	0	0	0	0			

mul Reg1, Reg3, Reg5
 sub Reg2, Reg4, Reg3
 div Reg6, Reg1, Reg4
 add Reg4, Reg2, Reg3

		register status									
		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	S _{add} 1	1	1	sub	2	(R4)	1	0	(R3)	1	0
	S _{add} 2	1	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	S _{mul} 3	1	1	mul	1	(R3)	1	0	(R5)	1	0
	S _{div} 4	1	1	div	6	$\frac{(R3)*(R5)}{(R3)+(R5)}$	1	0	(R4)	1	0

cycle 12

CDB	token.tag	4
	token.data	$\frac{(R3)*(R5)}{(R3)+(R5)}$

Tomasulo Zusammenfassung

- Verhindert WAR und WAW Hazards
- RAW werden durch CDB (forwarding) beschleunigt
- dynamisches scheduling
- Register renaming
- Jedoch: single-issue, in-order issue!
- Register sind nicht der Flaschenhals (forwarding vom CDB an die RS)
- aber CDB ist der Flaschenhals

- Implementierung in der IBM 360/91