Compiler Techniken für Hazards

Viele Stau-Typen sind recht häufig

Beispiel:

A = B + C

LW R1, B	**IF**	**ID**	**EX**	**MEM**	**WB**			
LW R2, C	**IF**	**ID**	**EX**	**MEM**	**WB**			
ADD R3, R1, R2		**IF**	**ID**	**stall**	**EX**	**MEM**	**WB**	
SW A, R3			**IF**	**stall**	**ID**	**EX**	**MEM**	**WB**

LW R1,B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	EX	MEM	WB			
ADD R3, R1, R2			IF	ID	stall	EX	MEM	WB	
SW A,R3				IF	stall	ID	EX	MEM	WB

Beispiel: Erzeuge gutes Maschinenprogramm für

A = B + C;
D = E - F;

```
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
```

Beide interlocks LW R3, C zu ADD R1, R2, R3 und LW R6, F zu SUB R4, R5, R6 sind nun eleminiert. Es gibt einen Hazard zwischen SUB und SW, aber der kann mit forwarding behoben werden.

Def: Pipeline Scheduling ist die Umstellung der Reihenfolge von Befehlen zur Vermeidung von Pipeline-Staus. Das Verhalten des Programms muss dabei erhalten bleiben.

Man kann an diesem Beispiel beobachten, daß die Benutzung unterschiedlicher Register für R2, R3 und R5, R6 notwendig ist. Insbesondere wegen der Befehlsumstellung würde eine doppelte Benutzung von R2 oder R3 für R5 zu falschem Verhalten führen.

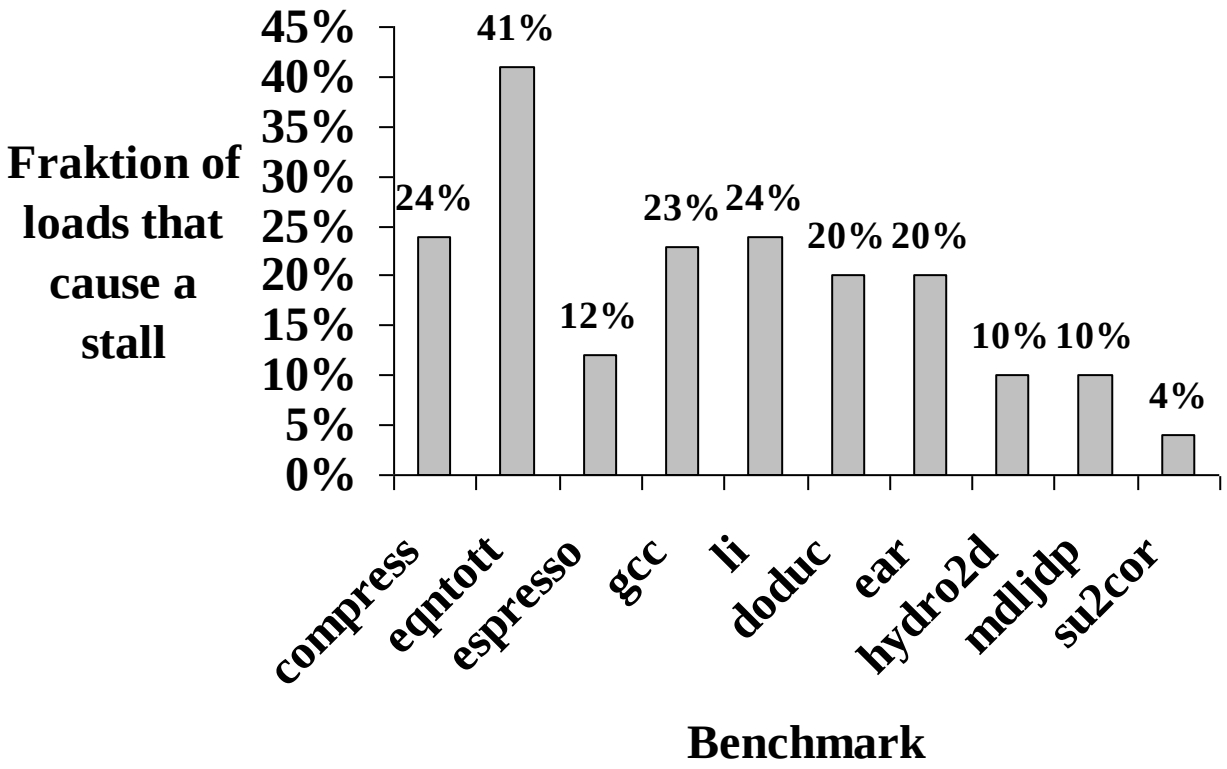
Wir lernen daraus:

Pipeline Scheduling kostet im allgemeinen zusätzliche Register.

Moderne Compiler benutzen Pipeline Scheduling in der Phase der *Lokalen Optimierung* in *basic blocks*. Das sind Code-Segmente linear nacheinander auszuführender Instruktionen ohne Ein- und Aussprünge, außer am Anfang und am Ende.

Für so einfache Pipelines wie unsere DLX (mit nur kurzen Stall-Verzögerungen, nur 1 bei load) ist dies ein adäquates Vorgehen. Wir müssen lediglich die Abhängigkeiten der Befehle als einen Graphen mit gerichteten Kanten zeichnen (Beispiel) und müssen die Reihenfolge so festlegen, dass alle Abhängigkeiten erfüllt sind, ohne dass ein Stau-Konflikt auftaucht.

Die Häufigkeit von Load-Staus wie oben ist aus folgender Folie für die Spec-Benchmarks


Fraktion of loads that cause a stall
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
24%
41%
12%
23%
24%
20%
20%
10%
10%
4%
compress
eqntott
espresso
gcc
li
doduc
ear
hydro2d
mdljdp
su2cor
Benchmark

Implementierung in der DLX Pipeline

Während der ID-Phase werden die Steuersignale erzeugt, die notwendig sind, um auf Hazards zu reagieren. (entweder forwarding oder interlock).

Je eher in der Pipeline man eine solche Situation erkennt, desto eher kann man reagieren. Wichtig ist, dass nie eine Situation auftritt, bei der man eine Veränderung des Prozessorstatus schon vorgenommen hat, die man dann aufgrund von einem Stau wieder zurücknehmen muss. Dies wäre nämlich sehr aufwendig.

Beispiel:
Wie reagieren wir auf einen load interlock?
Wir erkennen das Problem in der ID-Phase des Befehls, der u.U. gestallt werden muss.

Die folgende Tabelle zeigt die Situationen, die auftreten können.

Die erste und vierte Situation sind unkritisch.

Situation	**Example code sequence**	**Action**

No dependence	LW **R1,** 45 (R2)
	ADD R5, R6, R7
	SUB RS, R6, R7
	OR R9, R6, R7
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	LW **R1,** 45 (R2) ADD R5, **R1,** R7 SUB R8, R6, R7 OR R9, R6, R7	Comparators detect the use of R l in the ADD and stall the ADD (and SUB and QR) before the ADD begins EX.
Dependence overcome by forwarding	LW **R1, 45** (R2) ADD R5, R6, R7 SUB R8, **R1,** R7 OR R9, R6, R7	Comparators detect use of R l in SUB and for-ward result of load to ALU in time for SUB to begin EX.

Dependence with accesses in order	LW **R1,** 45 (R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, **R1,** R7
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Die erste und vierte Situation sind unkritisch.

Die zweite erfordert einen Stau. Dieser wird erkannt, wenn ADD in der ID-Phase ist und gleichzeitig LW in der EX-Phase.

Die folgende kleine Tabelle zeigt alle Problemsituationen anhand der in den Registern stehenden Bits:

Opcode field of ID/EX (ID/EX.IR$_{0..5}$)	**Opcode field of IF/ID (IF/ID.IRp $_{0..5}$)**	**Matching operand fields**
Load	Register-register ALU	ID/EX.IR$_{11.15}$ = IF/ID.IR$_{6..10}$
Load	Register-register ALU	ID/EX.IR$_{11.15}$ = IF/ID.IR$_{11..15}$
Load	Load, store, ALU immediate, or branch	ID/EX.IR$_{11..15}$ = IF/ID.IR$_{6..10}$

Für das forwarding geht es ähnlich, aber es gibt mehr Fälle. Alle diese sind in der folgenden Folie aufgeführt.

Zusätzlich zu Vergleichern und Schaltnetzen, die wir brauchen, um die Bedingungen zu überprüfen, ob eine Situation für forwarding vorliegt, müssen wir die Multiplexer an den Eingängen der ALU vergrößern und einen weiteren Multiplexer am Eingang des Speichers einfügen. Diese Multiplexer werden mit den Registern verbunden, aus denen die Zwischenergebnisse im forwarding-Fälle genommen werden müssen.
Dies sehen wir auf der (über)-nächsten Folie.

Pipeline-Register in dem die erste Operation ist	Instruktionstyp der ersten Operation	Pipeline-Register, in dem die zweite Operation ist	Instruktionstyp der zweiten Operation	Ziel des Ergebnisses des Forwarding	Bedingung für den Forwarding Fall
EX/MEM	Register-Register-ALU	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	3: Oberer ALU-Mux	$EX/MEM.IR_{16..20} = ID/EX.IR_{6..10}$
EX/MEM	Register-Register-ALU	ID/EX	Register-Register-ALU	8: Unterer ALU-Mux	$EX/MEM.IR_{16..20} = ID/EX.IR_{11..15}$
MEM/WB	Register-Register-ALU	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	2: Oberer ALU-Mux	$MEM/WB.IR_{16..20} = ID/EX.IR_{6..10}$
MEM/WB	Register-Register-ALU	ID/EX	Register-Register-ALU	9: Unterer ALU-Mux	$MEM/WB.IR_{16..20} = ID/EX.IR_{11..15}$
EX/MEM	ALU-immediate	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	3: Oberer ALU-Mux	$EX/MEM.IR_{11..15} = ID/EX.IR_{6..10}$
EX/MEM	ALU-immediate	ID/EX	Register-Register-ALU	8: Unterer ALU-Mux	$MEM/WB.IR_{11..15} = ID/EX.\ IR_{11..15}$
MEM/WB	ALU-immediate	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	2: Oberer ALU-Mux	$MEM/WB.IR_{11..15} = ID/EX.IR_{6..10}$
MEM/WB	ALU-immediate	ID/EX	Register-Register-ALU	9: Unterer ALU-Mux	$MEM/WB.IR_{11..15} = ID/EX.IR_{11..15}$
MEM/WB	Load	ID/EX	R-R-ALU, ALU-immediate, load, store, branch	1: Oberer ALU-Mux	$MEM/WB.IR_{11..15} = ID/EX.IR_{6..10}$
MEM/WB	Load	ID/EX	Register-Register-ALU	10: Unterer ALU-Mux	$MEM/WB.IR_{11..15} = ID/EX.IR_{11..15}$
EX/MEM	Register-Register-ALU	ID/EX	Store	13: DM-Mux	$EX/MEM.IR_{16..20} = ID/EX.IR_{11..15}$
EX/MEM	ALU-immediate	ID/EX	Store	12: DM-Mux	$EX/MEM.IR_{11..15}= ID/EX.IR_{11..15}$
EX/MEM	Load	ID/EX	Store	11: DM-Mux	$EX/MEM.IR_{11..15}= ID/EX.IR_{11..15}$

Beispiele:

Zeile 1:
```
ADD  R1,  R2,  R3
ADD  R4,  R1,  R5
```

Zeile 2:
```
ADD  R1,  R2,  R3
ADD  R4,  R5,  R1
```

Zeile 3:
```
ADD  R1,  R2,  R3
OR   R7,  R8,  R9
ADD  R4,  R1,  R5
```

Zeile 4:
```
ADD  R1,  R2,  R3
OR   R7,  R8,  R9
ADD  R4,  R5,  R1
```

Zeile 5:
```
ADDI R1,  R2, #1000
ADD  R4,  R1,  R5
```

Zeile 6:
```
ADDI R1,  R2, #1000
ADD  R4,  R5,  R1
```

Zeile7:
```
ADDI R1,  R2, #1000
OR   R7,  R8,  R9
ADD  R4,  R1,  R5
```

Zeile 8:
```
ADDI R1,  R2, #1000
OR   R7,  R8,  R9
ADD  R4,  R5,  R1
```

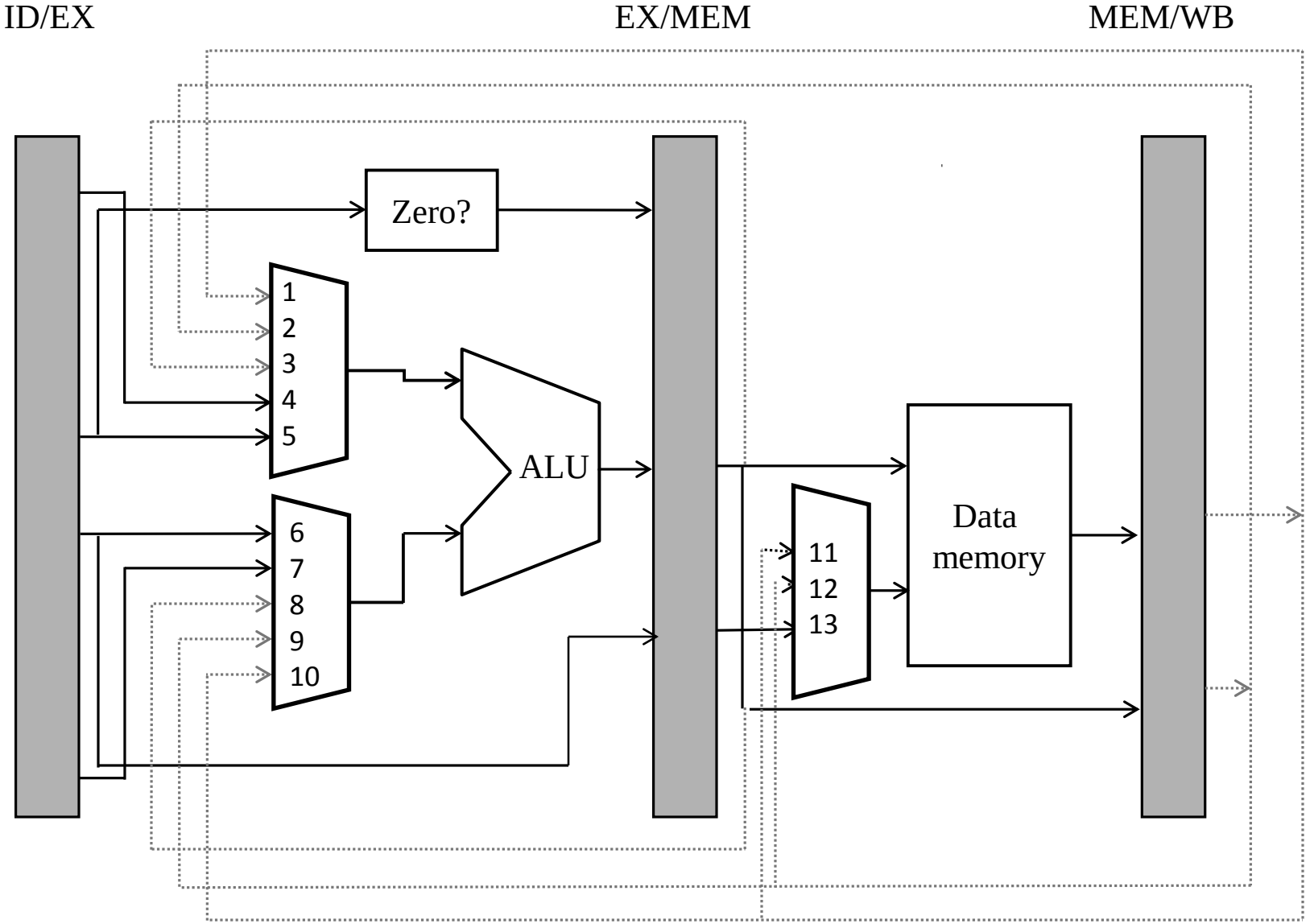
Zeile 9:
```
LW   R1,  1000(R2)
OR   R7,  R8,  R9
ADD  R4,  R1,  R5
```

Zeile 10:
```
LW   R1,  1000(R2)
OR   R7,  R8,  R9
ADD  R4,  R5,  R1
```

Zeile 11:
```
ADD  R1,  R2,  R3
SW   100(R4), R1
```

Zeile 12:
```
ADDI R1,  R2, #1000
SW   100(R4), R1
```

Zeile 13:
```
LW   R1,  1000(R2)
SW   100(R4), R1
```


ID/EX
EX/MEM
MEM/WB
Zero?
1
2
3
4
5
ALU
6
7
8
9
10
11
12
13
Data
memory

4.6. Behandlung von Verzweigungen

Kontrollhazards

kosten mehr pro Stau bei der DLX

Ausgeführte Verzweigung: Sprung zur Zieladresse (taken branch)
Nicht ausgeführte Verzweigung: PC+4 (not taken branch, untaken)

Erst in der Execute-Phase wird die Zieladresse berechnet, die Condition ausgewertet. D.h. erst in der MEM-Phase stehen der neue PC und das Cond-Register zur Verfügung.

Vor der ID-Phase wissen wir noch nicht, dass es ein Verzweigungsbefehl ist, daher können wir erst in der ID-Phase stauen. Das bedeutet, dass der Nachfolgebefehl noch bis in die IF-Phase kommt.
Das Stauen der Pipeline und Warten, ob die Verzweigung ausgeführt wird oder nicht resultiert in folgendem Ablauf:

Branch-Befehl	IF	ID	EX	MEM	WB				
Branch +1		IF	stall	stall	IF	ID	EX	MEM	WB
Branch +2						IF	ID	EX	MEM
Branch +3							IF	ID	EX

Der Stau in diesem Falle ist anders als beim Datenhazard, den die erste IF-Phase muß im Falle eines Verzweigungsbefehls ja durch eine zweite IF-Phase überschrieben werden. Daher muß die Hardware im Staufalle das Zielregister in der IF/ID-Register auf 0 setzen (no-op).

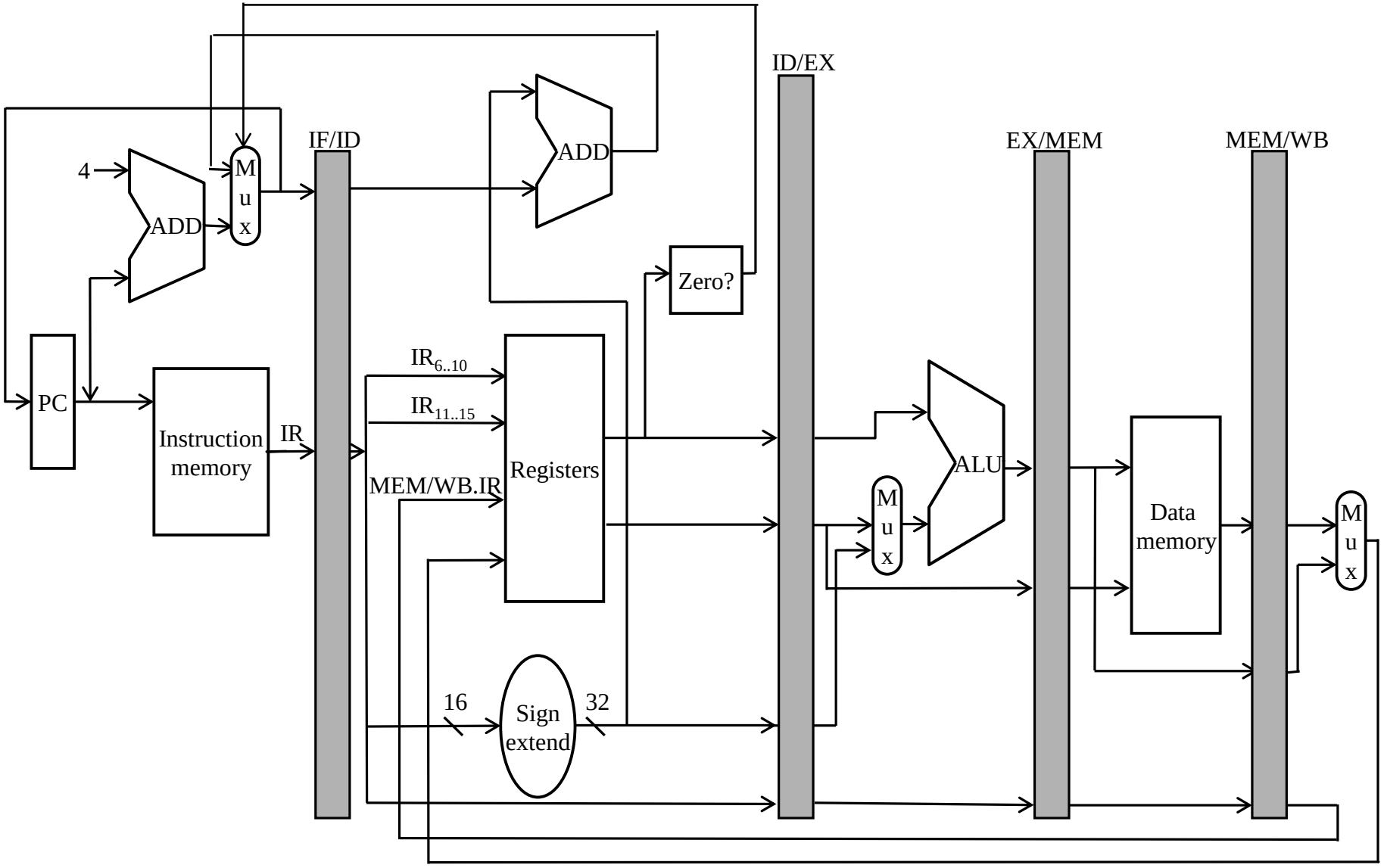
Wenn die Verzweigung nicht ausgeführt wird, ist der Stau eigentlich nicht erforderlich, denn wir haben mit dem Branch +1 Befehl bereits den richtigen Befehl gefetcht. Wir werden gleich sehen, wie man das ausnutzen kann, um Performance zu gewinnen.

Aber zuerst: **Wie können wir die Stau-Strafe vermindern?**

1. Früher herausfinden, ob verzweigt wird oder nicht.
2. Sprungziel eher berechnen.

Zusätzliche ALU (nur Adder) für die Berechnung des Verzweigungsziels bereits in der ID-Phase. Cond wird auch bereits in der ID-Phase berechnet.

Die nächste Folie zeigt den entsprechend veränderten Datenpfad.


4
ADD
M
u
x
IF/ID
ADD
ID/EX
EX/MEM
MEM/WB
Zero?
PC
Instruction
memory
IR
IR6..10
IR11..15
MEM/WB.IR
Registers
M
u
x
ALU
Data
memory
M
u
x
16
Sign
extend
32

Die Abfolge der Schaltvorgänge in der Pipeline wird sich dadurch folgendermaßen ändern:

Stufe	**Verzweigungsbefehl**
IF	IF/ID.NPC, PC ⟵ If(IF/ID.IR$_{0..5}$ = BRANCH und Regs[IF/ID.IR$_{6..10}$] op 0) {NPC + (IF/ID IR$_{16}$)^{16} ## IF/ID.IR$_{16..31}$} else {PC + 4}
	IF/ID.IR ⟵ MEM [PC]
ID	ID/EX.A ⟵ Regs[IF/ID.IR$_{6..10}$];
	ID/EX.B ⟵ Regs[IF/ID.IR$_{11..15}$];
	ID/EX.IR ⟵ IF/ID.IR;
EX	ID/EX.IMM ⟵ (IF/ID.IR$_{16}$)^{16}## IF/ID.IR$_{16..31}$
MEM	
WB	

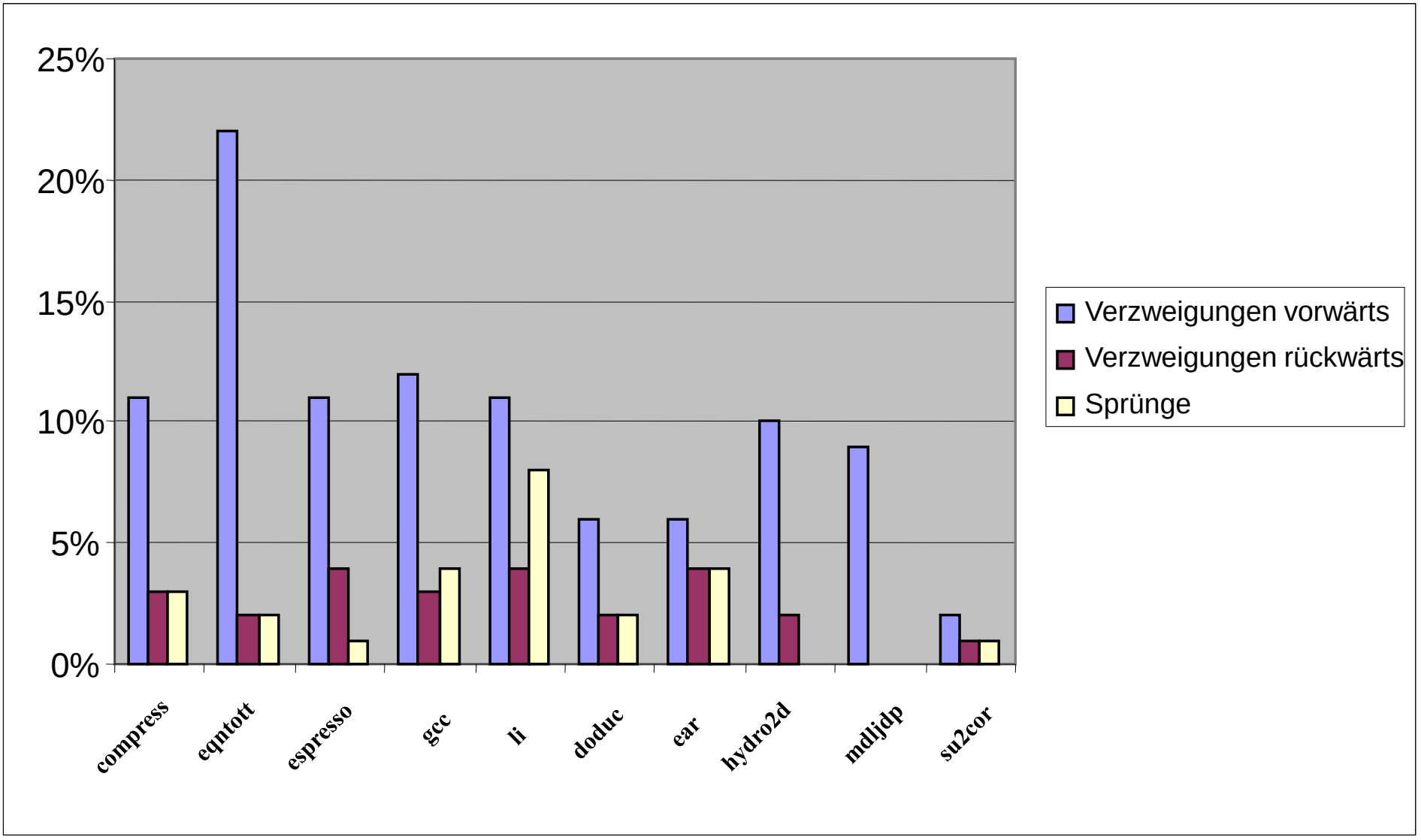
Weil Verzweigungsbefehle die Pipeline-Performance sehr stark beeinflussen können, stellen wir eine Analyse des Verhaltens von Programmen mit Sprüngen:

Auf der folgenden Folie sehen wir das Verhalten der SPEC-Programme bezüglich **Sprüngen** und **Verzweigungen**, die in vorwärts und rückwärts unterteilt sind.

Integer: 14-16% Branches, weniger Sprünge

FP: 3-12% Branches, weniger Sprünge

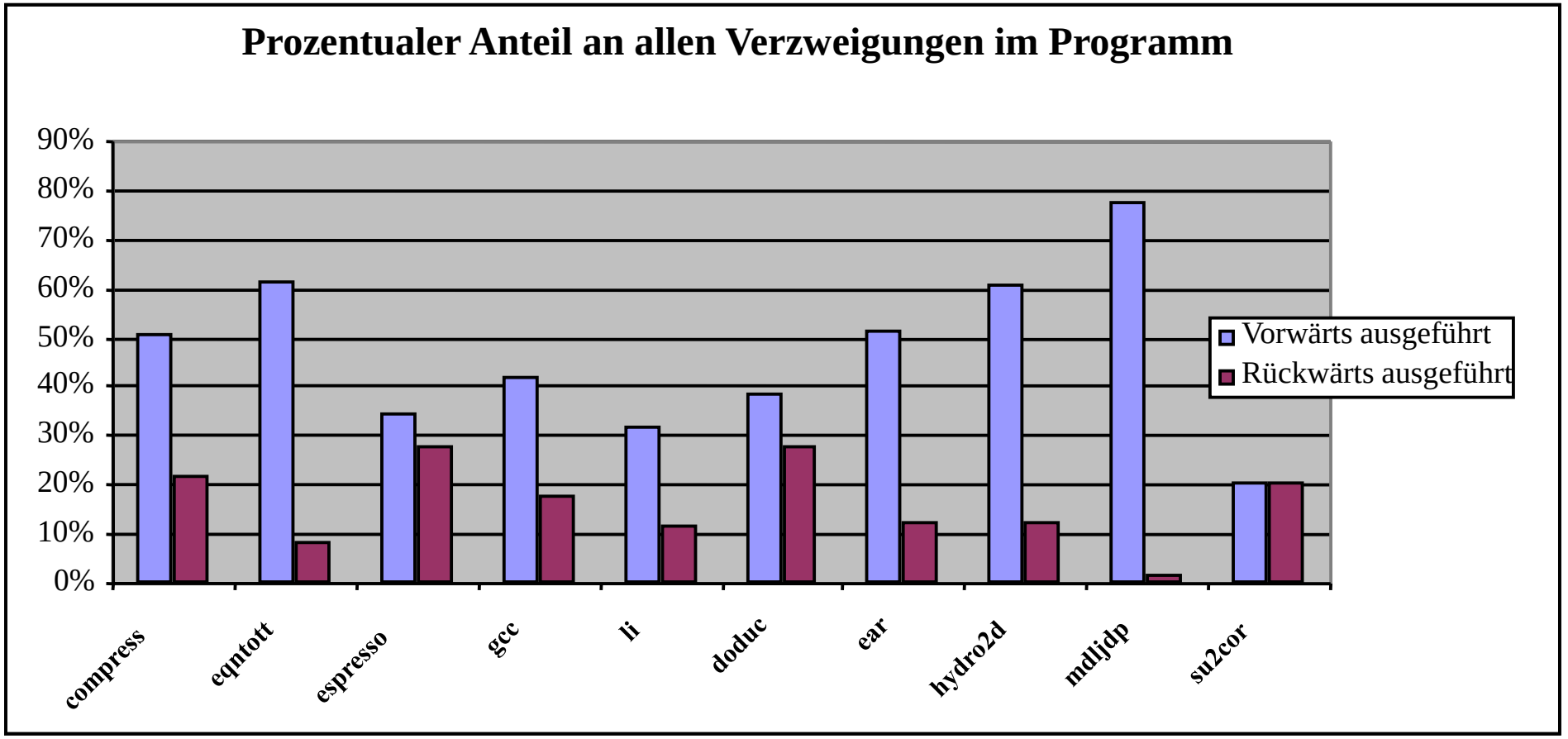
Vorwärts zu rückwärts ca. 3:1
das ist verwunderlich, da man ja bei Schleifen zurückspringt. Aber bei If vorwärts (2 Branches pro if-Statement). Daher dieses Verhältnis.


25%
20%
15%
10%
5%
0%
compress
eqntott
espresso
gcc
li
doduc
ear
hydro2d
mdljdp
su2cor
Verzweigungen vorwärts
Verzweigungen rückwärts
Sprünge

Für die Effizienz der Strategie, mit der man Branches behandelt, ist es wichtig zu wissen

wie groß ist der Anteil an ausgeführten Verzweigungen zu nicht ausgeführten Verzweigungen.

Dies sehen wir auf dem nächsten Schaubild.


Prozentualer Anteil an allen Verzweigungen im Programm
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
compress
eqntott
espresso
gcc
li
doduc
ear
hydro2d
mdljdp
su2cor
Vorwärts ausgeführt
Rückwärts ausgeführt

Wie kann man die Verzweigungsstrafe minimieren?

Vier Strategien, die zur Compilezeit gewählt werden können:

freeze
predict not taken, treat as not taken
predict taken, treat as taken
delayed branch

1. Freeze

Stauen bis das Ergebnis der Verzweigung bekannt ist. **Vorteil: Einfachheit für Soft- und Hardware. Nachteil: hoher CPI-Wert.**

2. Treat as not taken

Jede Verzweigung wird zunächst so behandelt, als werde sie nicht ausgeführt. Die Pipeline fährt fort, Instruktionen zu holen und zu Dekodieren, bis das Ergebnis des Tests vorliegt. Im Falle, dass der Sprung nicht genommen werden soll, wird normal (ohne Penalty) weitergemacht. Im Falle, dass gesprungen werden soll, werden die fälschlich gefetchten und dekodierten Befehle von der Hardware in no-ops umgewandelt, wodurch alle temporären Register und alle GPRs erhalten bleiben, bis das Sprungziel neu gefetcht worden ist.

Nicht ausgeführter Branch	IF	ID	EX	MEM	WB					
Branch +1		IF	ID	EX	MEM	WB				
Branch +2			IF	ID	EX	MEM	WB			
Branch +3				IF	ID	EX	MEM	WB		

Ausgeführter Branch	IF	ID	EX	MEM	WB					
Branch +1		IF	No-op	No-op	No-op	No-op				
Sprungziel			IF	ID	EX	MEM	WB			
Sprungziel +1					IF	ID	EX	MEM	WB	

Vorteil: Keine Strafe, wenn nicht genommene Verzweigung. Nachteil: mehr Verzweigungen werden ausgeführt als nicht ausgeführt. Daher:

3. Treat as taken

Jede Verzweigung wird zunächst so behandelt, als werde sie ausgeführt. Die Pipeline fährt fort, Instruktionen ab dem Sprungziel zu holen und zu Dekodieren, bis das Ergebnis des Tests vorliegt. Im Falle, dass der Sprung genommen werden soll, wird normal (ohne Penalty) weitergemacht. Im Falle, dass nicht gesprungen werden soll, werden die fälschlich gefetchten und dekodierten Befehle von der Hardware in no-ops umgewandelt, wodurch alle temporären Register und alle GPRs erhalten bleiben, bis neu gefetcht worden ist.

Bei unserer DLX bringt diese Strategie nichts, denn wir kennen das Sprungziel ja erst, wenn wir auch die Bedingung ausgewertet haben. Dies ist aber bei anderen Maschinen, z. B. welchen mit einer tieferen Pipeline, nicht der Fall. Daher gilt das Diagramm auf der folgenden Folie so nicht für die DLX.

Vorteil: Keine oder nur geringe Strafe bei Verzweigung. Nachteil: Kostet extra Hardware, früh herauszufinden, ob verzweigt wird.

Nicht ausgeführter Branch	IF	ID	EX	MEM	WB			
Sprungziel		IF	ID	No-op	No-op	No-op		
Branch +1			IF	ID	EX	MEM	WB	
Branch +2				IF	ID	EX	MEM	WB

Ausgeführter Branch	IF	ID	EX	MEM	WB			
Sprungziel		IF	ID	EX	MEM	WB		
Sprungziel +1			IF	ID	EX	MEM	WB	
Sprungziel +2				IF	ID	EX	MEM	WB

4. Die vierte Technik wird **delayed branch** genannt.

Hinter dem Verzweigungsbefehl gibt es sogenannte delay slots. Das sind die möglichen Nachfolgebefehle, die begonnen werden, bevor die Sprungentscheidung getroffen ist. Diese delay slots werden durch Scheduling mit anderen Instruktionen gefüllt, die auf jeden Fall (also unabhängig vom Ausgang des Vergleichs) ausgeführt werden.

Bei der DLX haben wir einen Delay-slot. (Dies ist üblich für Maschinen mit delayed branch)

Die Situation sieht also so aus wie in dem folgenden Diagramm für ausgeführte bzw. nicht ausgeführte Verzweigung.

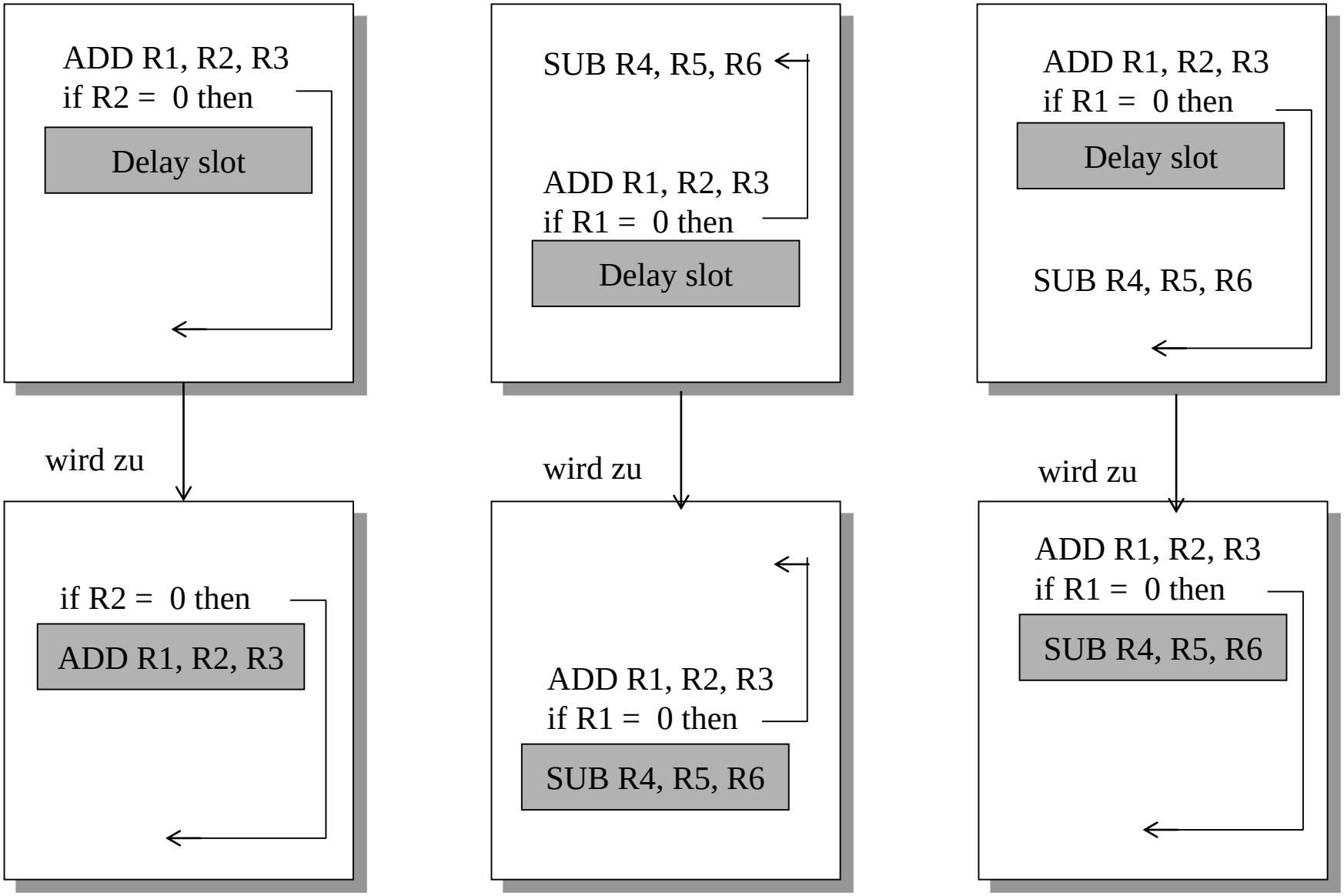
Nicht ausgeführteVerzweigung	IF	ID	EX	MEM	WB					
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB				
Instruction *i* +2			IF	ID	EX	MEM	WB			
Instruction *i* +3				IF	ID	EX	MEM	WB		
Instruction *i* +4					IF	ID	EX	MEM	WB	

AusgeführteVerzweigung	IF	ID	EX	MEM	WB					
Branch delay instruction (i+1)		IF	ID	EX	MEM	WB				
Branch target			IF	ID	EX	MEM	WB			
Branch target +1				IF	ID	EX	MEM	WB		
Branch target +2					IF	ID	EX	MEM	WB	

Der Compiler ist nun verantwortlich dafür. dass die Instruktion im Delay Slot valide und nützlich ist. Es gibt **drei Möglichkeiten**, die er hat, um die Validität mit Sicherheit und die Nützlichkeit mit größter zur Compilezeit ermittelbarer Wahrscheinlichkeit zu gewährleisten:

1. **Schedule from before**
2. **Schedule from target**
3. **Schedule from fall through**

Diese Möglichkeiten werden auf der folgenden Folie illustriert:


ADD R1, R2, R3
if R2 = 0 then
Delay slot
wird zu
if R2 = 0 then
ADD R1, R2, R3
SUB R4, R5, R6
ADD R1, R2, R3
if R1 = 0 then
Delay slot
wird zu
ADD R1, R2, R3
if R1 = 0 then
SUB R4, R5, R6
ADD R1, R2, R3
if R1 = 0 then
Delay slot
SUB R4, R5, R6
wird zu
ADD R1, R2, R3
if R1 = 0 then
SUB R4, R5, R6

From before: ist immer von Vorteil, wenn es möglich ist. Aber man muss einen Befehl, finden dessen Ergebnis nicht die Bedingung beeinflusst. Wenn from before nicht möglich ist, wählt der Compiler eine der anderen Möglichkeiten:

From target: Man wählt für den delay slot eine Instruktion vom Ziel des Sprunges. Diese Variante soll gewählt werden, wenn die Ausführung des Sprungs wahrscheinlich ist, z. B. bei Rücksprüngen für Schleifen. Dabei ist zu bedenken:

1. Die Instruktion muss kopiert werden, weil das Sprungziel auch von einer anderen Stelle aus angesprungen werden könnte.

2. Die Instruktion darf im Falle, dass der Sprung nicht ausgeführt wird, nicht falsch sein, d. h. das Zielregister muss in diesem Falle redundant sein.

3. Die Instruktion darf nicht noch einmal ausgeführt werden, falls sie durch den Schedule nach dem Branch ausgeführt wird.

From fall through: Man wählt für den delay slot einen Befehl nach dem Sprungbefehl. Diese Variante ist vorzuziehen, wenn der Sprung mit größter Wahrscheinlichkeit nicht ausgeführt wird.

Auch hier muss die Ausführung des delay-Befehls legal sein, falls der Sprung doch in unerwartete Richtung geht. Dies wäre z. B. der Fall, wenn R4 ein Register ist, dass zu dieser Zeit sonst von niemandem benutzt wird.

Scheduling Strategie	**Bedingung für legale Ausführung**	Verbessert Performance
from before	**Darf nicht auf das Register schreiben, wenn in der Bedingung abgefragt wird**	immer
from target	**Umgestellter Befehl muss ok sein, auch wenn nicht gesprungen wird**	Wenn gesprungen wird (kann Programm verlängern, Kopie des eingefügten Befehls)
from fall through	**Muss ok sein, auch wenn gesprungen wird**	Wenn nicht gesprungen wird (u. U. Kopie des eingefügten Befehls)

Worin liegt der Vorteil gegenüber dem normalen *treat as not taken?* 1. From before bringt immer Vorteil, 2. Man kann Nutzen daraus ziehen, wenn man etwas über die Wahrscheinlichkeit sagen kann, mit der der Sprung ausgeführt wird.

Je tiefer die Pipeline wird, desto geringer wird der Performancegewinn durch delayed branches. Für unsere DLX mit einem delay-slot gilt das noch nicht, denn für ein-slot-Systeme ist der Delayed Branch attraktiv wegen guter Performance bei geringem Hardwareaufwand.

Das Einzige, was man braucht, ist eine zusätzliche Kopie des PC. Warum?

Wenn bei der Ausführung des Befehls im delay-slot ein Interrupt auftritt, muss im Anschluss an die Interrupt-Behandlung der Befehl im delay-slot neu ausgeführt werden und der Befehl am Verzweigungsziel danach. Diese haben aber keine aufeinanderfolgenden Adressen. Daher müssen beide Adressen in zwei PCs gehalten werden, und in der Pipeline mitwandern, bis der Befehl im delay-slot fertig ist.

Aber bei heutigen Maschinen wird mehr Wert auf Hardware-Unterstützung bei der Verzweigungsvorhersage (branch prediction) gelegt. Dafür wird aber der delayed branch nicht mehr immer unterstützt.

Cancelling branch

Zusätzlich zur Instruktion wird dem Maschinenprogramm vom Compiler Informationen über das prognostizierte Sprungverhalten mitgegeben. Solange diese Prognose richtig liegt, wird der Befehl nach dem Sprung ausgeführt. Wenn sie falsch liegt, wird er gecancelt, d. h. durch einen no-op ersetzt.

Diese Technik nutzt ebenfalls die Möglichkeit der **Prognose von Sprungwahrscheinlichkeiten zur Compilezeit** aus.

Sie hat aber auf der anderen Seite nicht den Nachteil, das das Zielregister des Befehls im delay-slot redundant sein muss wie in der vorherigen Compiler Scheduling Technik.

Die meisten Maschinen bieten beide Möglichkeiten: nicht-cancelling und cancelling (meist cancel if not taken) delayed branches. Dies Kombination hat sich in der Praxis bewährt.

Predict taken cancelling branch

Nicht ausgeführteVerzweigung	IF	ID	EX	MEM	WB					
Branch delay instruction (i + 1)		IF	ID	idle	idle	idle				
Instruction *i* +2			IF	ID	EX	MEM	WB			
Instruction *i* +3				IF	ID	EX	MEM	WB		
Instruction *i* +4					IF	ID	EX	MEM	WB	

AusgeführteVerzweigung	IF	ID	EX	MEM	WB					
Branch delay instruction (i+1)	IF	ID	EX	MEM	WB					
Branch target		IF	ID	EX	MEM	WB				
Branch target +1			IF	ID	EX	MEM	WB			
ranch target +2				IF	ID	EX	MEM	WB		

Erweiterung der DLX-Pipeline für Floating-Point Arithmetik

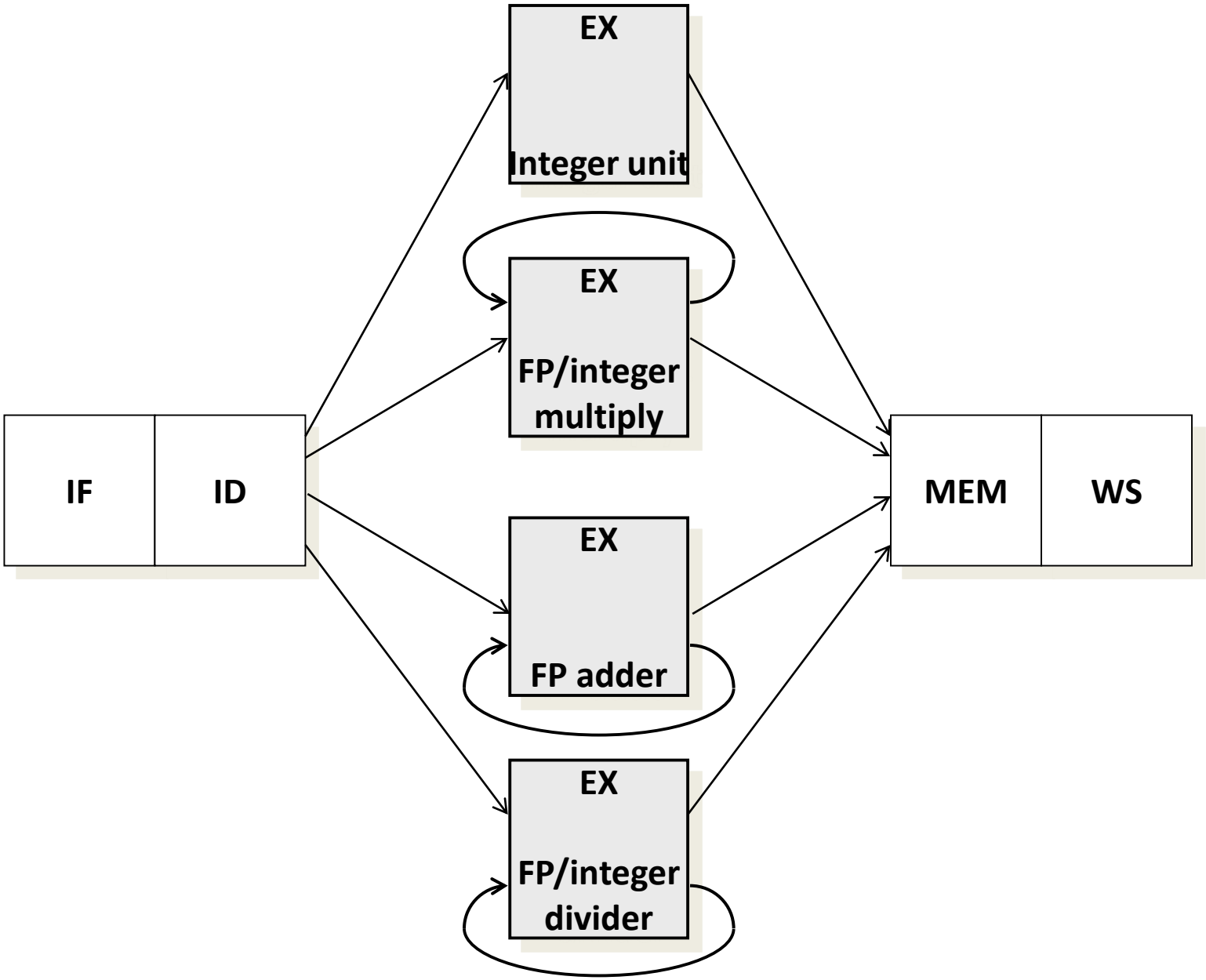
Unpraktisch wenn nicht unmöglich FP-Operationen in einem Takt zu machen. Dies würde bedeuten, daß ein enormer Hardwareaufwand getrieben werden müßte und der Takt sehr langsam sein müßte.

Beispiel FP-ADD, FP-MULT

Daher macht man folgende Veränderungen für FP-Operationen.

Anstelle einer **EX-Einheit** stellt man vier verschiedene Einheiten zur Verfügung. Die erste ist die alte Integer-Einheit. Die zweite ist für FP- und Interger **Multiplikation**, die dritte für **FP-Addition** und die vierte für **FP- und Integer Division**.

Da die Operationen außerhalb der EX-Einheit nicht in einem Zyklus ausgeführt werden können, müssen die Teilergebnisse wieder an deren Eingang zurückgeführt werden, bis das Ergebnis vollständig berechnet ist.


EX
Integer unit
EX
FP/integer multiply
EX
FP adder
EX
FP/integer divider
IF
ID
MEM
WS

Nun liegt es nahe, auch die anderen Einheiten ihrerseits als Pipelines zu implementieren. Dazu muß man zunächst wissen, wie lange die jeweiligen Berechnungen dauern, d. h. wieviele Stufen die jeweilige Pipeline haben wird.

Latency ist die Anzahl der Takte, die der Befehl zusätzlich zur alten EX-Phase (ungepipelined für die Ausführung) braucht. Soviele+1 Stufen der Pipeline sieht man vor.

Die Tiefe der Pipeline ist also die Latency + 1.

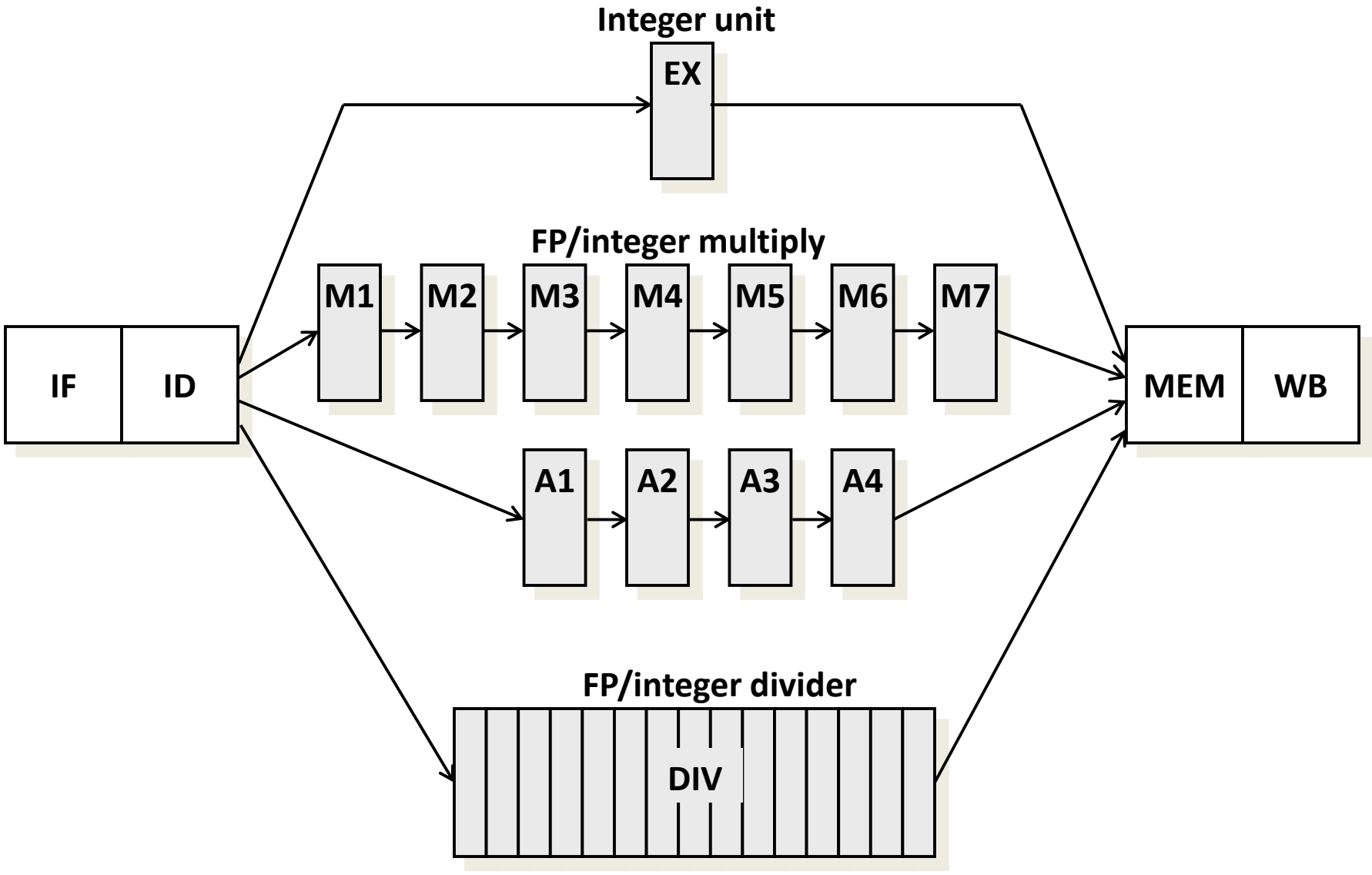
Mit **Initiation interval** wird die Anzahl der Takte bezeichnet, die erforderlich sind, bevor nach Beginn einer Berechnung die nächste Instanz der Berechnung in der Pipeline beginnen kann.

Wir sehen an der Tabelle, daß alle Einheiten voll in Pipelinestufen der Länge 1 unterteilt sind, außer der Division, die nicht gepipelined ist.

Definition:
Latency ist die Anzahl der Takte einer Verarbeitungseinheit, die der Befehl zusätzlich zur alten EX-Phase (ungepipelined für die Ausführung) braucht.

Definition:
Das **Initiation Interval** ist die Anzahl der Takte, die erforderlich sind, bevor nach Beginn einer Berechnung die nächste Instanz der Berechnung in der Pipeline beginnen kann.

Functional unit	Latency	Initiation intervall
Integer ALU	0	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide and FP sqrt)	14	15


Integer unit
EX
FP/integer multiply
M1
M2
M3
M4
M5
M6
M7
IF
ID
MEM
WB
A1
A2
A3
A4
FP/integer divider
DIV

Es werden jetzt natürlich **zusätzliche Pipeline-Register** notwendig: M1/M2, M2/M3, ..., A1/A2, ID/EX, ID/M1, ID/A1, ID/DIV, M7/MEM, A4/MEM, DIV/MEM

Die unterschiedlichen Pipelinetiefen verursachen nun Probleme.

Betrachten wir folgendes Beispiel: Folie

Bemerkung:
LD lädt 64 Bit. Wie kann man das in einem MEM-Zyklus schaffen? 64-Bit breiter Datenbus zum Memory. Ablage der Werte im Speicher so, daß 64-Bit breit zugegriffen werden kann.

	Takt										
Befehl	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	*EX*	**MEM**	WB			

1. **RAW-Staus werden wesentlich häufiger**. Die Kontrolle für Pipeline-Staus ist aufwendiger, obwohl sie im Prinzip mit den Methoden, die wir kennengelernt haben, zu bewältigen ist.

```
Beispiel oben:    MULTD   F0, F2, F4
                  ADD     F8, F6, F0

oder              MULTD   F0, F2, F4
                  ADD     F8, F6, F4
                  LD      F6, 400(R2)
                  SD      320(R1), F0
```

2. **Struktur-Hazards werden möglich**, da die Division nicht gepipelined ist.

```
Beispiel          DIVD    F0, F2, F4
                  DIVD    F6, F8, F10
```

3. **Weil die Operationen unterschiedlich lang sind, müssen mehr als eine Schreiboperation in den Registerfile pro Takt möglich sein.** Wenn dasselbe Register beschreiben werden soll entsteht ein **Struktur Hazard**.

4. **WAW Hazards werden möglich**.

Beispiel: MULTD F0, F2, F4
ADDD F0, F6, F8

5. **Instruktionen werden in anderer Reihenfolge fertig, als sie begonnen werden.** Dies verursacht **Probleme bei Interrupts**.

Im folgenden Beispiel sehen wir eine typische Folge von FP-Befehlen (daxpy-loop, **d**ouble precision **a*X p**lus **Y**)). Jede Instruktion muß hier auf den Vorgänger warten, obwohl schon alle Möglichkeiten für forwarding ausgenutzt sind.
Interessant ist ferner, daß das SD am Ende noch einen weiteren Takt gestallt werden muß, weil es nicht gleichzeitig in der MEM Phase sein kann wie das ADDD (denn die Leitungen sind nur einfach vorhanden, d.h. Daten können nicht von zwei Befehlen gleichzeitig in der MEM Phase verarbeitet werden.

	Takt																
Befehl	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
LD F4, 0 (R2)	IF	ID	EX	MEM	WB												
MULTD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADDD F2, F0,F8					IF	ID	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
SD 0(R2), F2							IF	ID	EX	stall	stall	stall	stall	stall	stall	stall	MEM

Zu 3. Weiteres Beispiel nächste Folie:
Drei Befehle kommen gleichzeitig in ihre MEM und WB-Phase. Die MEM-Phase ist unkritisch, denn die Konflikte können ohne großen Hardwareaufwand behoben werden (nur mehrfache Register und Leitungen).
In der WB-Phase allerdings kommt es zu Problemen.
1. Lösung: Mehrere Write-Ports. Teuer, denn der Fall kommt zu selten vor, als das dieser Aufwand gerechtfertigt wäre.
2. Lösung: Die späteren Befehle werden gestaut, so daß nacheinander geschrieben werden kann.
Was ist dazu notwendig?
Zwei Möglichkeiten:
1. Alle machen erst mal bis zur **MEM Phase** weiter und stauen dann. Vorteil: Man kann andere, u.U. zwischendurch aufgetretene Staus ausnutzen, so daß für WB kein weiterer Stau erforderlich ist. Nachteil: Wenn gestaut werden muß, müssen rückwärts in der Pipeline alle Stufen angehalten werden. Das ist sehr aufwendig. Viel einfacher ist es, wenn weiterhin alle Hazards in der ID-Phase des zweiten Befehls erkannt und behandelt werden. Daher
2. Ein Schieberegister mit der Länge n des längsten Befehls (Pipelinetiefe) in der ID-Phase führt Buch über die Benutzung des Write-ports. Jeder Befehl, der den Write-port braucht, schreibt eine 1 ins Schieberegister an der Stelle k, wenn er in n-k Takten schreiben will. Wenn dort schon eine Eins steht, wird er einen Takt gestaut, wartet einen Takt und versucht es wieder.

	Takt										
Befehl	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
MULTD FO, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2,F4,F6				IF	ID	Al	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F8,0(R2)							IF	ID	EX	MEM	WB

```
LOOP:   LD     F0, 0(R1)      1
        SUBI   R1, R1,#8      2
        ADDD   F4, F2, F0      3
        stall                  4
        BNEZ   R1, LOOP        5
        SD     8(R1), F4       6
```

Durch Scheduling konnten wir die fünf Stall-Zyklen also zu einem reduzieren.

Bemerkungen: Der Stau-Zyklus ist da, weil das SD auf das ADDD warten muß.
Die Umstellung des SUBI nach vorne, die den wesentlichen Anteil an dieser Optimierung hat, ist natürlich trickreich: Sie bedingt, daß die SD- Instruktion verändert wird. (von 0(R1) auf 8(R1)).
Dies ist natürlich nicht trivial, denn die meisten Compiler würden die Abhängigkeit von SD und SUBI sehen und daraufhin die Reihenfolge unverändert lassen.

In diesem Beispiel erledigen wir die vorher 10 Zyklen lange Schleife in 6 Zyklen. Aber von diesen sind immer noch nur 3 für die eigentliche Berechnung erforderlich. Dagegen SUBI, BNEZ und stall sind eigentlich nur für die Verwaltung der Schleife zuständig.

Die Technik, die man zur weiteren Optimierung benutzen kann, nennt sich loop unrolling. Die Schleifenrümpfe werden mehrfach hintereinanderkopiert, wodurch der Aufwand an branches und Subs vermindert wird:

```
LOOP:   LD      F0, 0(R1)
        ADDD    F4, F2, F0
        SD      0(R1), F4
        SUBI    R1, R1, #8
        LD      F0, 0(R1)
        ADDD    F4, F2, F0
        SD      0(R1), F4
        SUBI    R1, R1, #8
        LD      F0, 0(R1)
        ADDD    F4, F2, F0
        SD      0(R1), F4
        SUBI    R1, R1, #8
        LD      F0, 0(R1)
        ADDD    F4, F2, F0
        SD      0(R1), F4
        SUBI    R1, R1, #8
        BNEZ    R1, LOOP
```

```
LOOP:   LD      F0, 0(R1)
        ADDD    F4, F2, F0
        SD      0(R1), F4
        LD      F6, -8(R1)
        ADDD    F8, F2, F6
        SD      -8(R1), F8
        LD      F10, -16(R1)
        ADDD    F12, F2, F10
        SD      -16(R1), F12
        LD      F14, -24(R1)
        ADDD    F16, F2, F14
        SD      -24(R1), F16
        SUBI    R1, R1, #32
        BNEZ    R1, LOOP
```

Loop unrolling führt also zu einem Programmstück, bei dem vier Werte in 28 Zyklen errechnet werden, denn nach jedem LD ist ein Stall, nach jedem ADDD zwei, nach SUBI 1 und nach BNEZ 1.

Jetzt liegt es nahe, den so entstandenen größeren Basic Block wieder mit Scheduling (lokaler Optimierung) umzustellen, so daß die Anzahl der Stau Zyklen weiter vermindert wird:

```
LOOP:   LD      F0, 0(R1)
        LD      F6, -8(R1)
        LD      F10, -16(R1)
        LD      F14, -24(R1)
        ADDD    F4, F2, F0
        ADDD    F8, F2, F6
        ADDD    F12, F2, F10
        ADDD    F16, F2, F14
        SD      0(R1), F4
        SD      -8(R1), F8
        SUBI    R1, R1, #32
        SD      +16(R1), F12
        BNEZ    R1, LOOP
        SD      8(R1), F16
```

Zu bemerken: Das SUBI muß zwei Befehle vor dem Branch sein, weil sonst ein Stall dazwischen notwendig wird, denn der Branch braucht das Ergebnis von SUBI schon in der ID-Phase.

Insgesamt wird die Arbeit jetzt in 14 Zyklen erledigt, also die Berechnung eines Wertes in 3,5 Zyklen (verglichen mit 10 am Anfang).

Welches Vorgehen ist notwendig für Loop unrolling und Scheduling?

1. Erkenne, daß die Vertauschung von SD und SUBI legal ist und berechne die Veränderung im Offset in SD.

2. Erkenne, das Loop unrolling sinnvoll ist, weil die einzelnen Iterationen der Schleife unabhängig sind.

3. Benutze verschiedene Register, um Abhängigkeiten von aufeinanderfolgenden Iterationen zu verringern

4. Verringere Anzahl der Verzweigungen und Tests durch längere Schleifenrümpfe

5. Erkenne, welche Loads und Stores vertauscht werden können mit Loads und Stores aus anderen Iterationen.

6. Scheduling des Codes bei Erhaltung der verbleibenden Abhängigkeiten.