Performanceverbesserung durch Pipelining

am Beispiel des MIPS-Prozessors

Wozu Pipelining?

Pipelining beschleunigt den Prozessor durch Erhöhung des Befehlsdurchsatzes.
Dies wird durch die Parallelausführung mehrerer Instruktionen erreicht.

Pipelined Datapath

Der Data path (siehe Rückseite) wird in fünf Phasen zerlegt:

1. IF = Instruction fetch: Ein Befehl wird aus dem Speicher geladen.
2. ID = Instruction decode: Der Befehl wird interpretiert: Die Kontrollsignale werden erzeugt und die Register gelesen.
3. EX = Execution: Die ALU Operation wird ausgeführt
4. MEM = Memory access: Speicherzugriff
5. WB = Write Back: Ergebnis wird gegebenenfalls in das Register geschrieben.

Zwischen den Phasen müssen Pipeline Register eingefügt werden, damit die Daten jeweils weiter gegeben werden können.

Im Pipelining können Probleme auftreten, wenn Befehle von vorherigen Ergebnissen abhängig sind. Man unterscheidet Datenkonflikte und Sprungkonflikte.

Datenkonflikte

Das Ergebnis einer Berechnung wird von einem darauffolgenden Befehl benötigt, bevor es in ein Register gespeichert worden ist. Durch Forwarding (Daten werden aus einem Pipeline Register direkt zur ALU umgeleitet) lässt sich dieses Problem meistens lösen.
Wird das Ergebnis eines Load-Befehls gebraucht, lässt sich dieses nicht gleich forwarden. Die Pipeline muss in die Ex-Phase ein NOP (No OPeration) eingefügt werden.

Sprungkonflikte

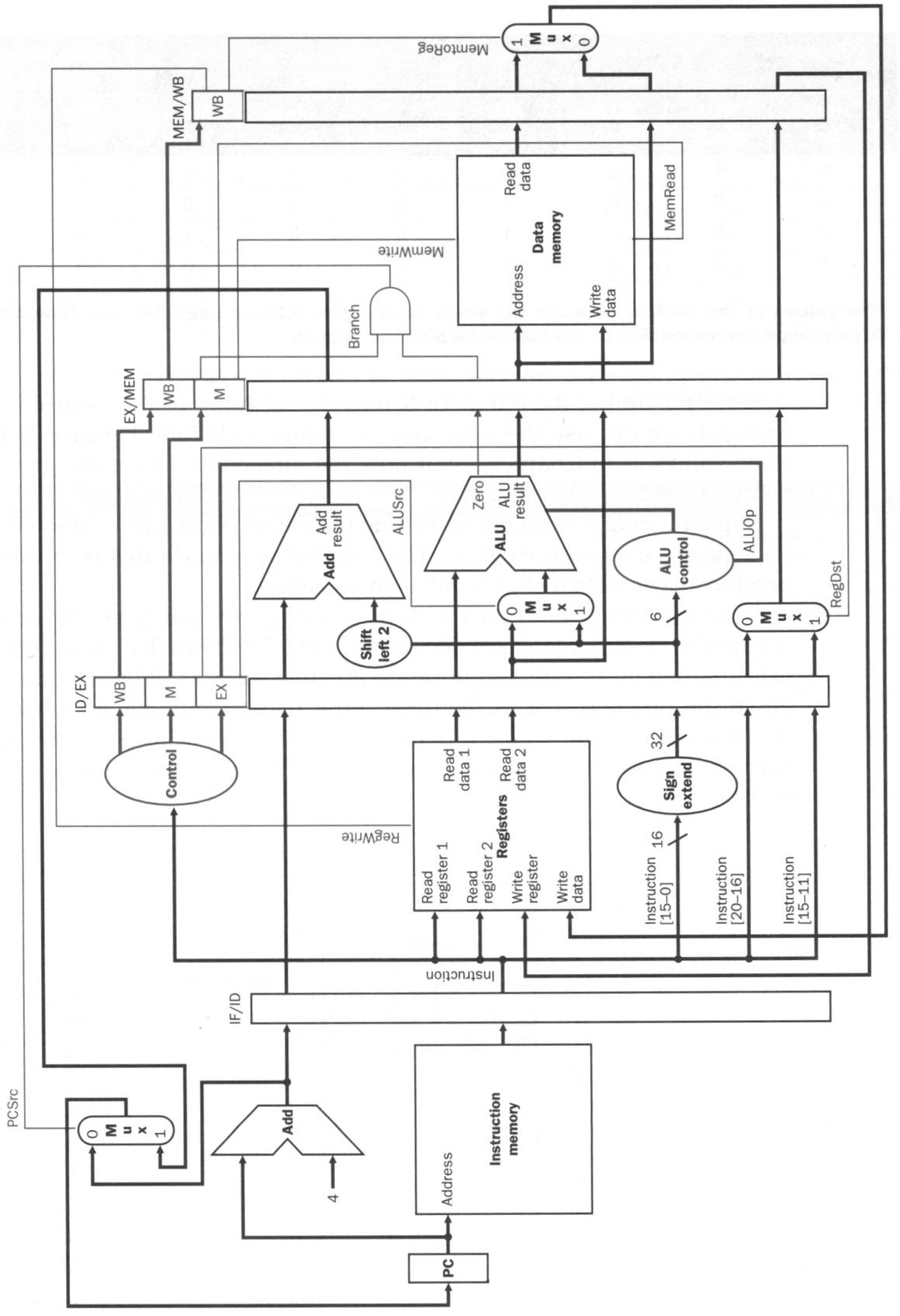
Kommt in einer Befehlsfolge ein Sprungbefehl vor, ist noch nicht klar, bei welchem Befehl weitergemacht werden soll. Mittels Branch Prediction (Vorhersage unter Berücksichtigung, ob die letzen Male gesprungen oder nicht gesprungen wurde) wird versucht, Verzögerung durch Falschraten zu reduzieren. Die Hardware wird so gestaltet, dass die Entscheidung möglichst früh fällt, damit bei Irrtum möglichst wenig Befehle umsonst aufgeführt werden.

Ausblick: Superscalar und Dynamic Pipelining

Beim Superscalar Pipelining wird versucht, durch Duplizierung der Hardware in jeder Phase mehrere Befehle auszuführen.
Dynamic Pipelining versucht Konflikte durch nichtlineares Bearbeiten zu lösen. Die Befehle werden auf unterschiedliche Funktionseinheiten verteilt und dort gleichzeitig bearbeitet. Eine Commit Unit sortiert hinterher die Befehle wieder in die richtige Reihenfolge.

Verwendete Literatur

David A. Patterson and John L. Hennessy:
Computer Organization & Design – The Hardware/Software Interface
Morgan Kaufmann, 1998 (2nd Ed.), 1-55860-491-X


MemtoReg
MEM/WB
WB
Read data
Data memory
Address
Write data
MemRead
MemWrite
Branch
EX/MEM
WB
M
Add result
Add
ALUSrc
Zero
ALU result
ALU
ALU control
ALUOp
RegDst
Shift left 2
ID/EX
WB
M
EX
Control
Read data 1
Read data 2
Registers
Read register 1
Read register 2
Write register
Write data
RegWrite
Sign extend
32
16
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Instruction
IF/ID
PCSrc
Add
4
Instruction memory
Address
PC

Performanceverbesserung durch Pipelining

am Beispiel des MIPS-Prozessors

Stefan Conrad Andreas Tyart

1. Ein Überblick über Pipelining

Gibt es für verschiedene Aufgaben unterschiedliche Hardware, lässt sich die Performance durch Parallelarbeit weiter steigern.

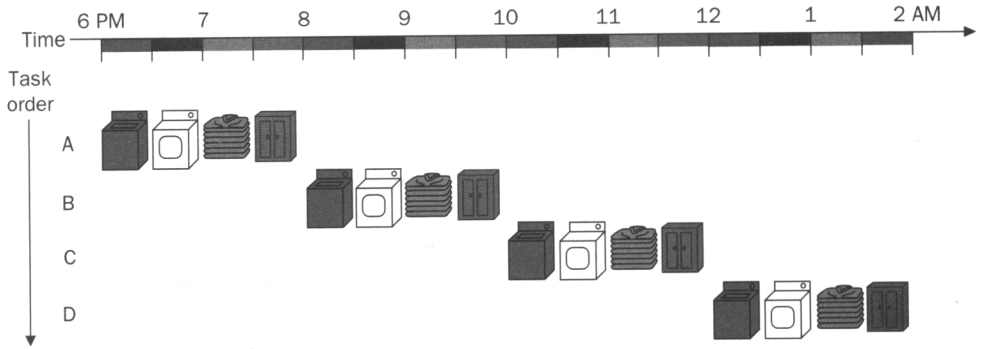
1.1. Ein Beispiel aus dem Alltag

Wäsche waschen: Es gibt vier Arbeitsschritte

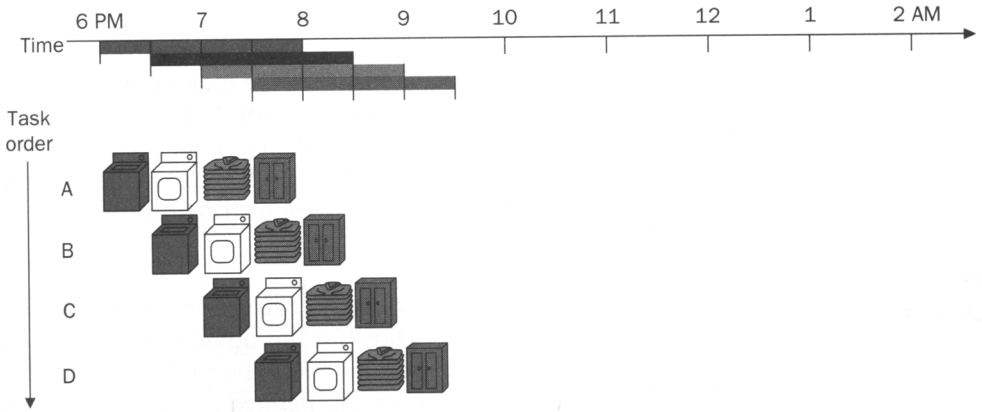
- Waschmaschine
- Trockner
- Bügelautomat
- Wegsortieren

4 Personen wollen waschen.

1.2. Hintereinanderausführung



1.3. Parallelarbeit (Pipelining)



→ Die Arbeitszeit für ein Waschdurchgang ist gleich geblieben, aber die gesamte Waschdauer ist mehr als halbiert.

2. Datapath beim Pipelining

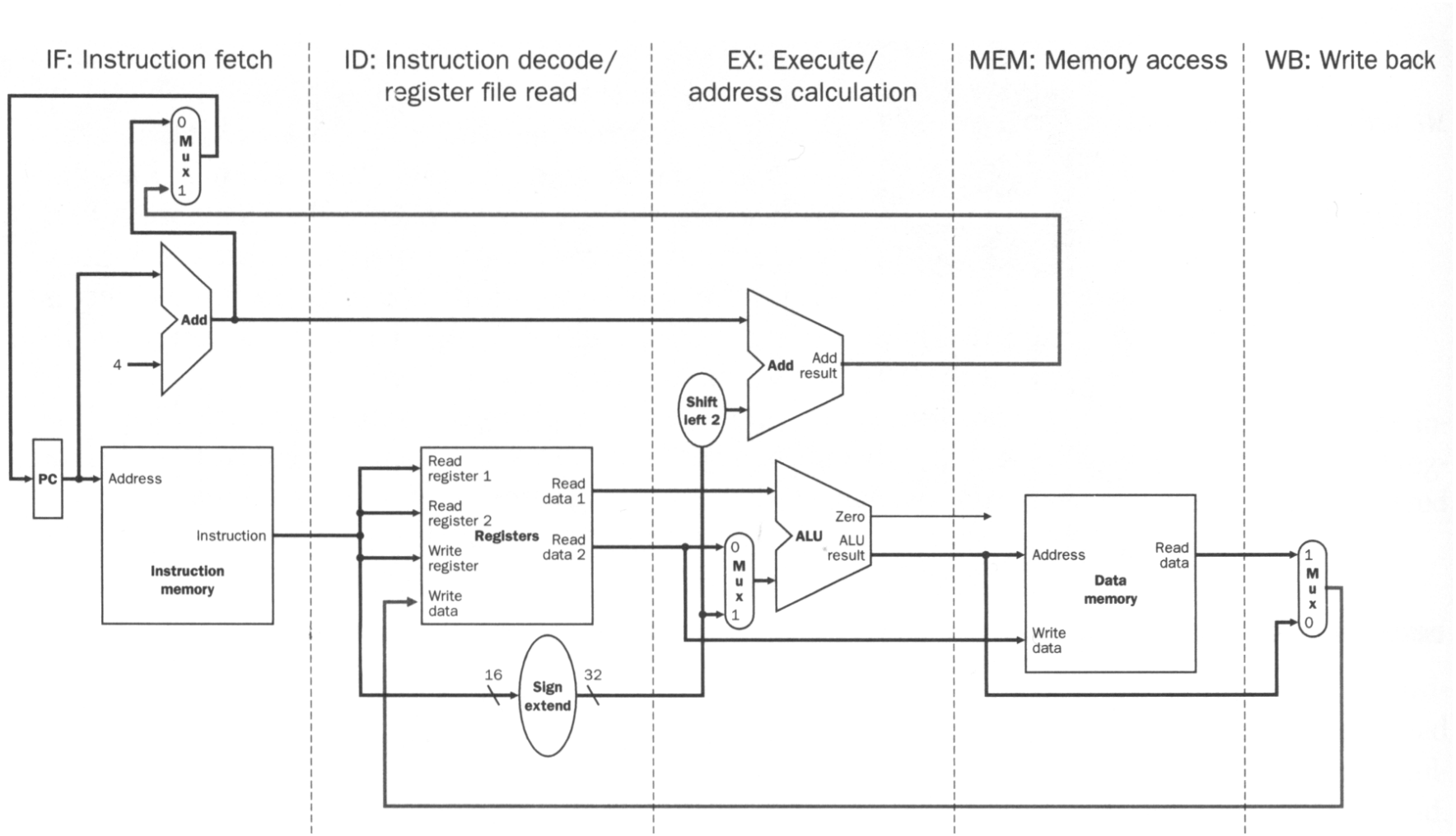
2.1. Einteilung der Phasen beim MIPS

Wir betrachten hier nur die Befehle lw, sw, add, sub, and, or, slt und beq

Instruction class	Instruction fetch	Register read	ALU Operation	Data access	Register write	Total time
Load	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store	2 ns	1 ns	2 ns	2 ns		7 ns
R-Format	2 ns	1 ns	2 ns		1 ns	6 ns
Branch	2 ns	1 ns	2 ns			5 ns

Jede Phase muss die gleiche Länge haben, ebenso jede Instruktion.
→ Phasenlänge 2 ns, Dauer für die gesamte Instruktion: 10 ns

2.2. Datapath beim Pipelining

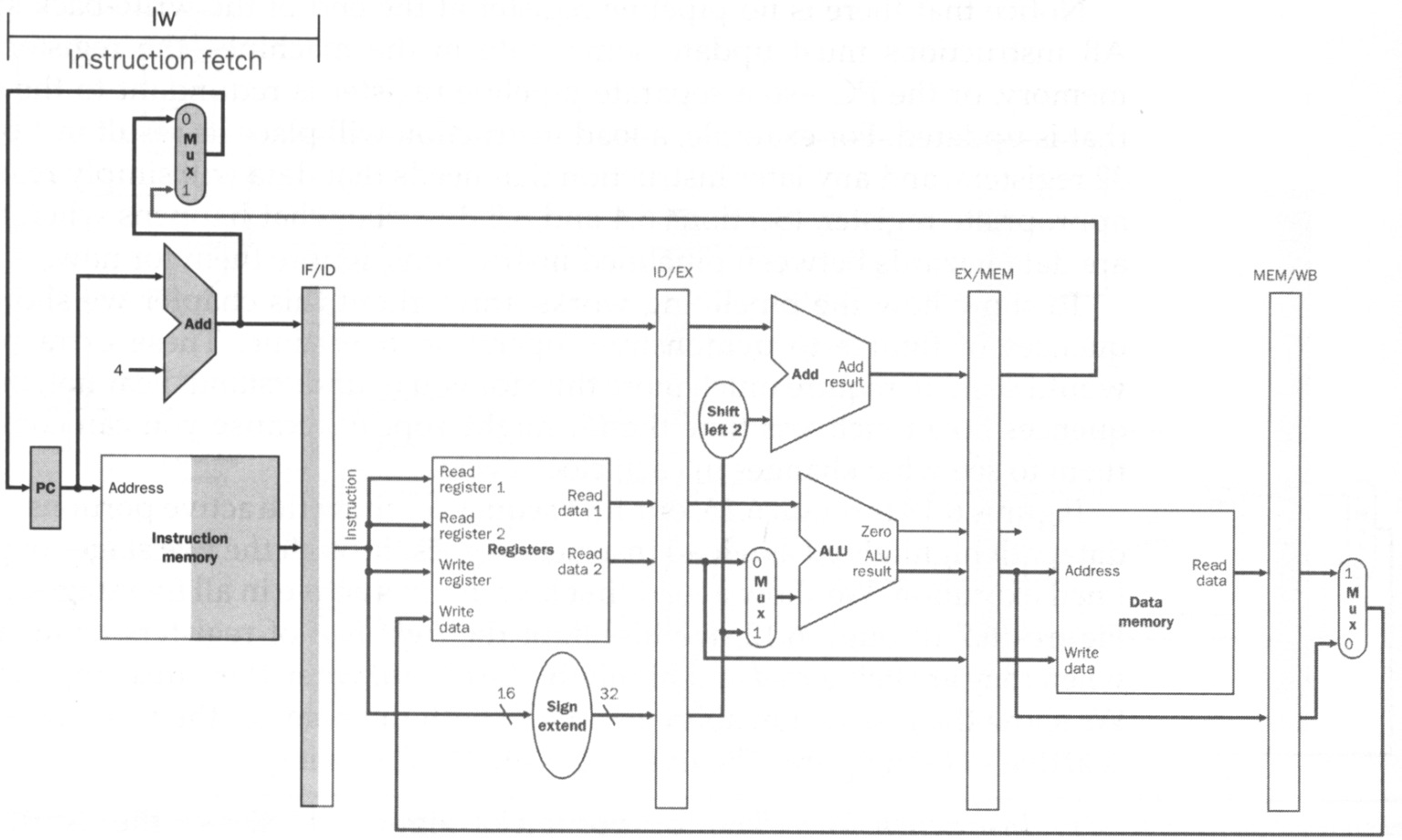


2.3. Pipeline Register

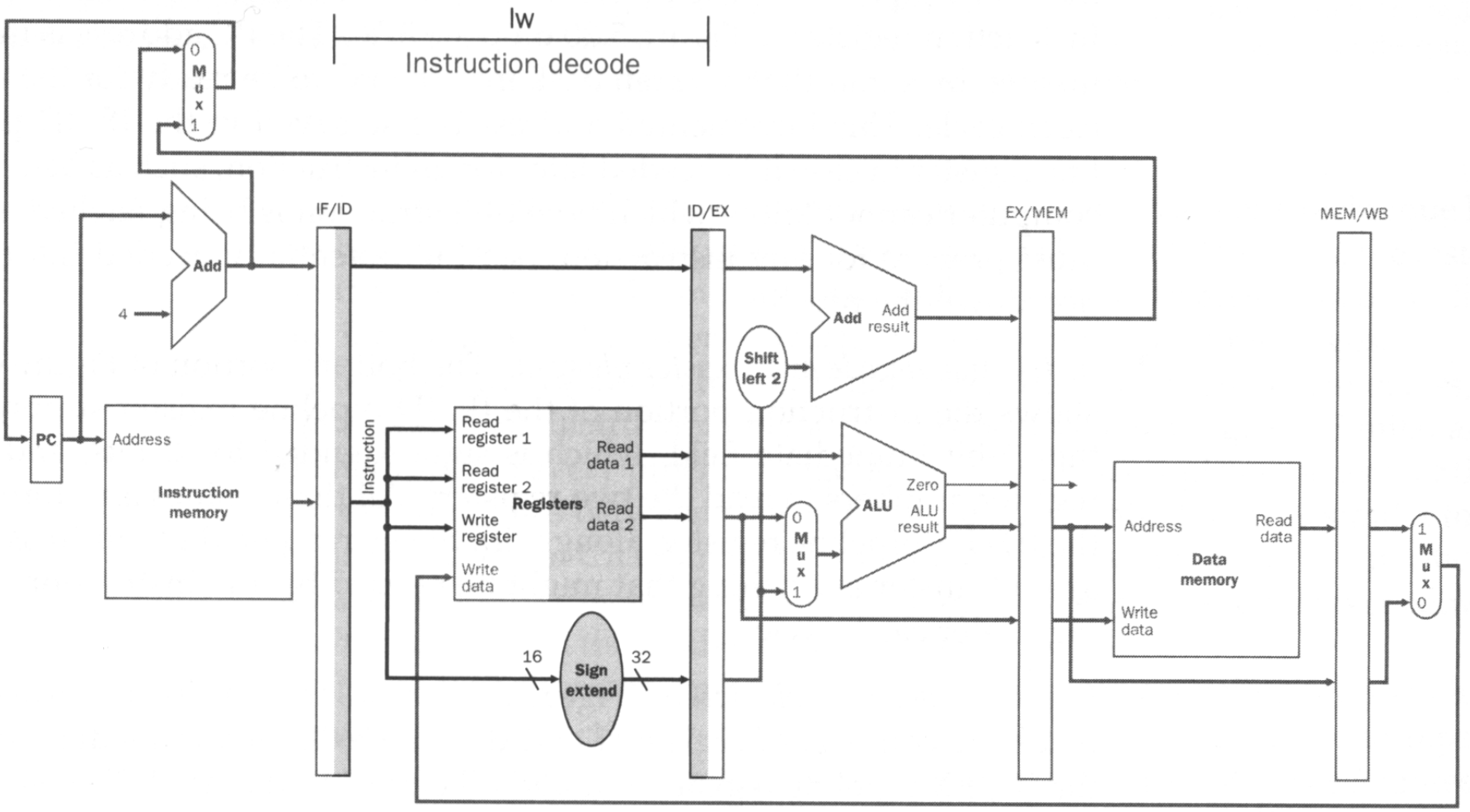
- Problem: Wohin mit den Zwischenergebnissen?
- Lösung: Pipeline Register für die Zwischenspeicherung wie bei Multicycle
- Analogie: Waschkörbe zwischen Waschmaschine, Trockner, Bügelautomat und Schrank

2.4. Beispiel load-Instruktion

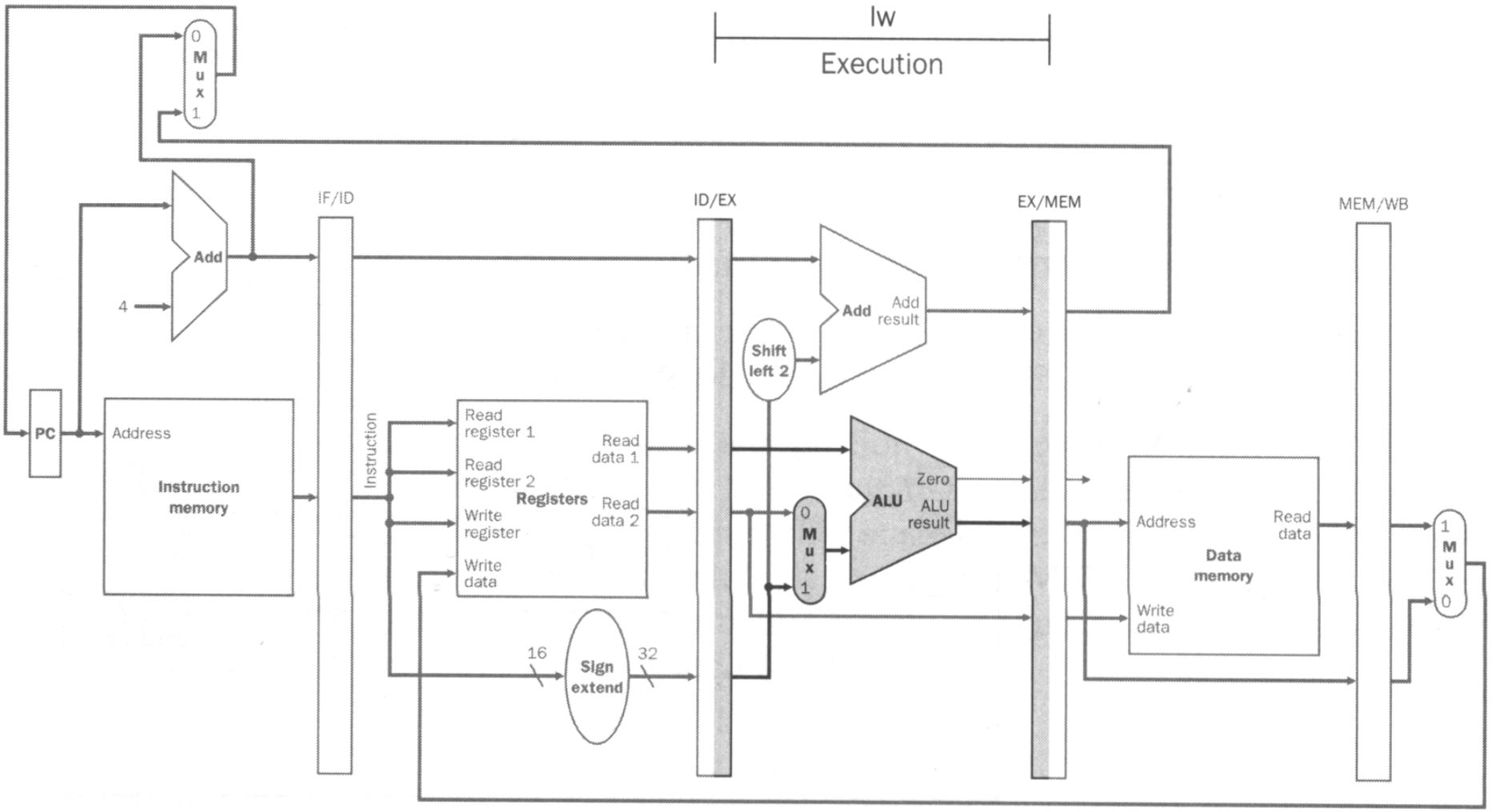
2.4.1. IF: Instruction load (Instruktion laden)



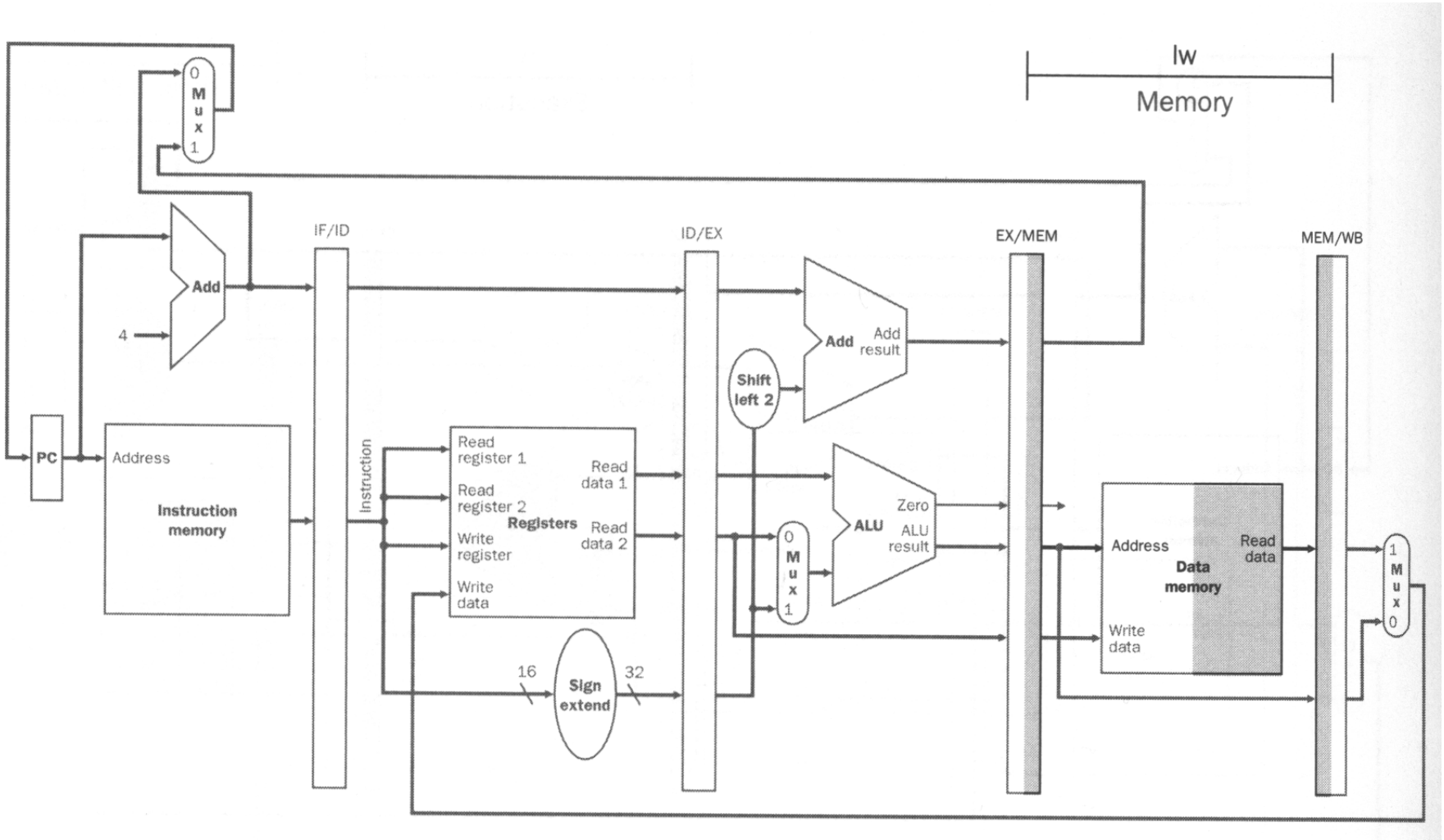
2.4.2. ID: Instruction decode (Instruktion dekodieren)



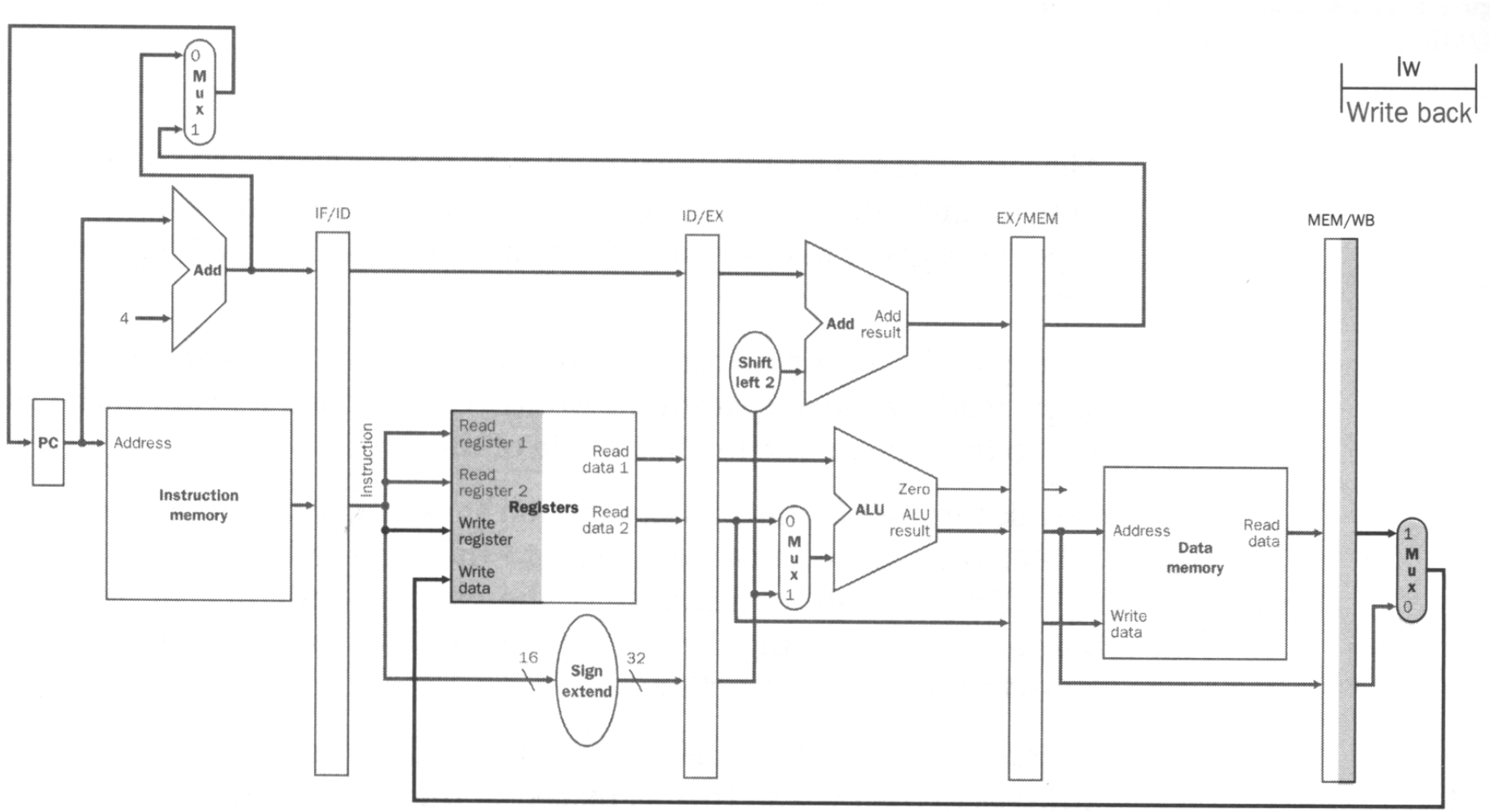
2.4.3. EX: Execute (Ausführen)



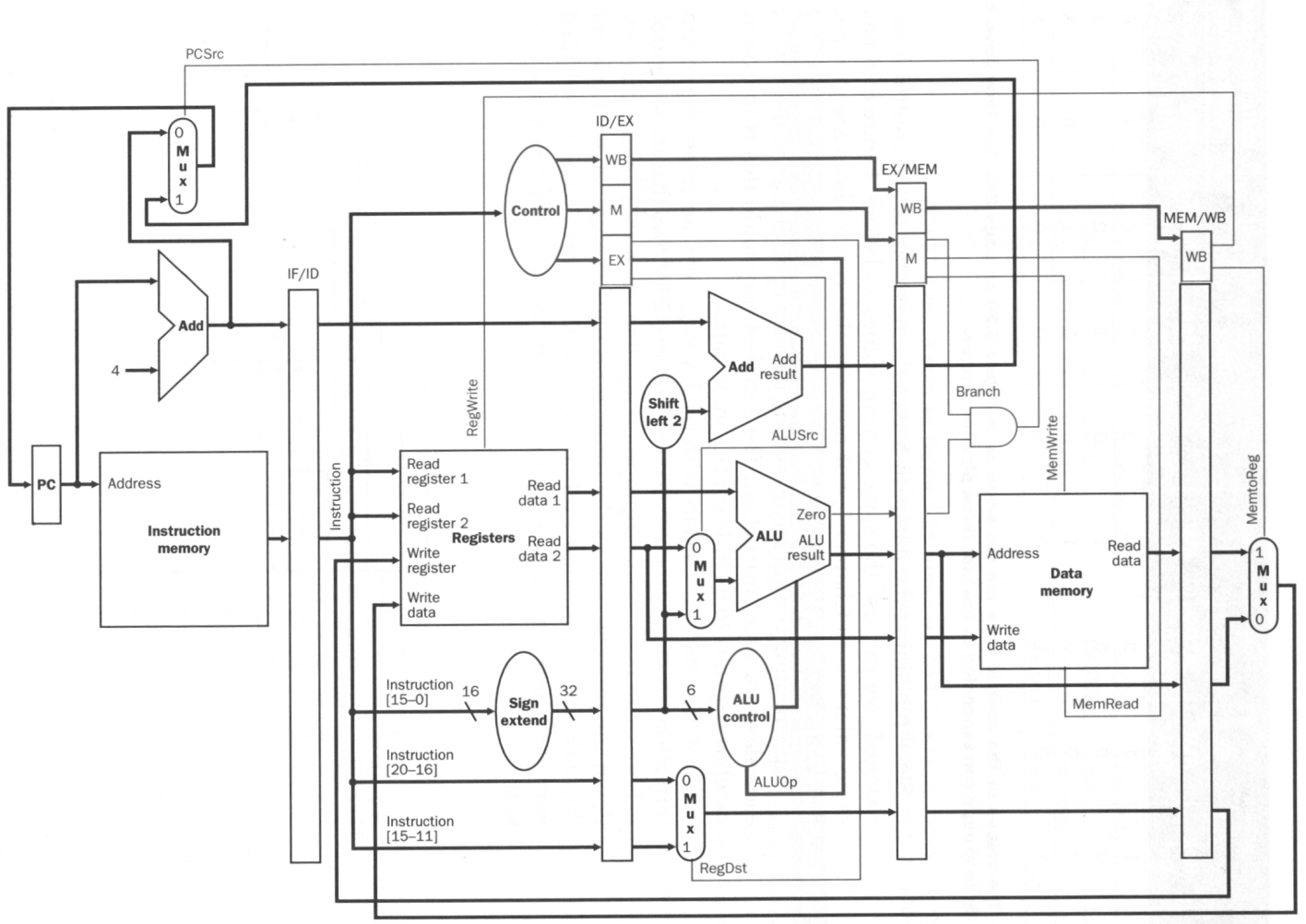
2.4.4. MEM: Memory access (Speicheroperation)



2.4.5. WB: Write back (Ergebnis speichern)



2.5. Pipelined Control



3. Die Probleme der Pipeline :

Datenkonflikte :

Diese entstehen, wenn Befehle von dem Ergebnis vorheriger abhängig sind.
Möglichkeiten diese zu umgehen :

- der Compiler ordnet die Befehle so, daß die Abhängigkeit umgangen wird (evtl. nop)
- Forwarding - die Ergebnisse werden benutzt, bevor sie im Zielregister sind
- Stalls - einige Abhängigkeiten lassen sich nicht durch Forwarding lösen, so daß einzelne Phasen sich im „Leerlauf“ befinden müssen

Sprungkonflikte:

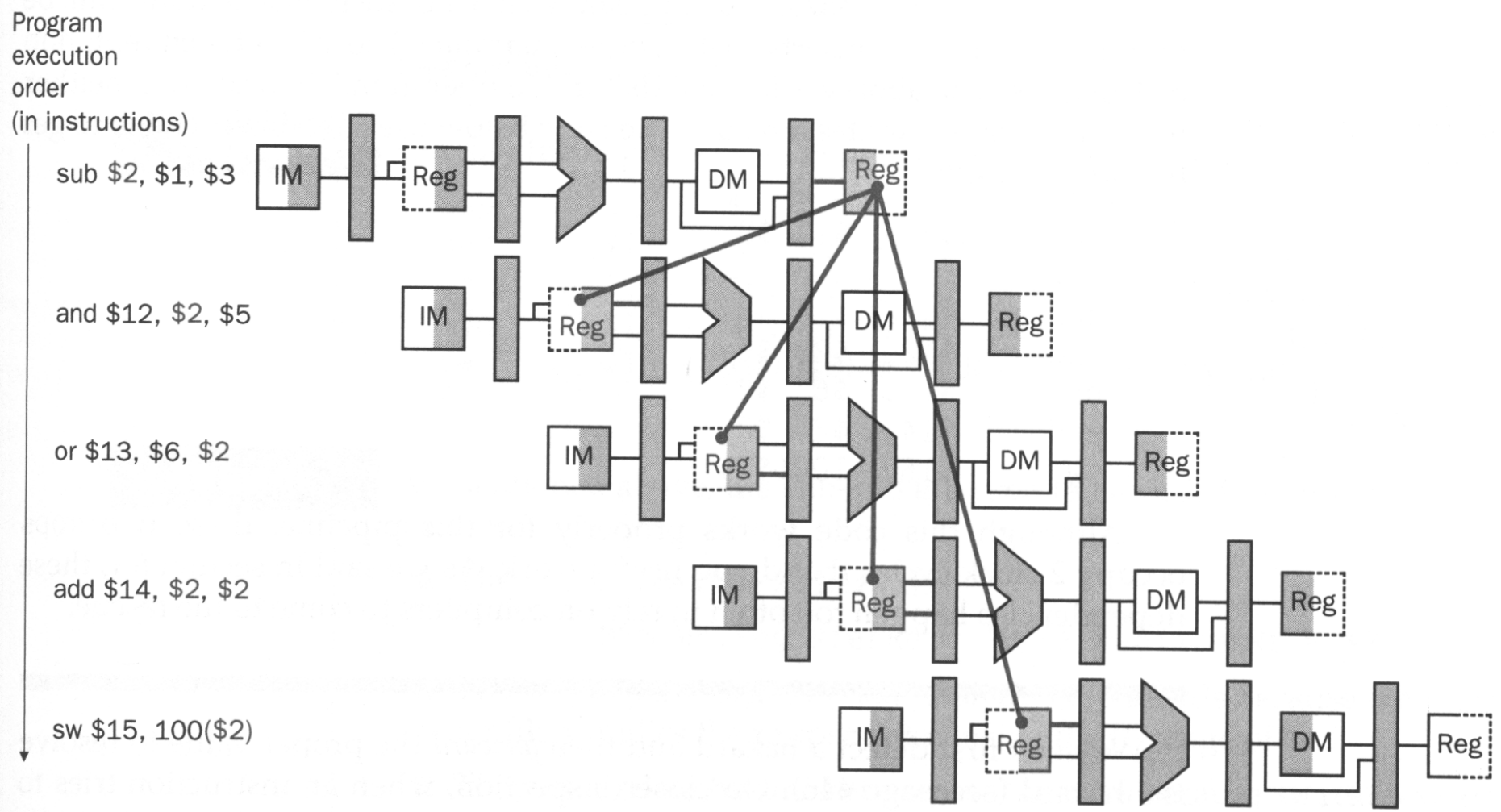
Wenn in der Befehlsfolge ein Sprungbefehl (z.B. beq) auftaucht, ist nicht klar, mit welchem Befehl es weitergeht. Hier gibt es keine Patentlösung wie bei den Datenkonflikten, sondern nur Schadenbegrenzung:

- Reducing the Delay of Branches
- Assume Branch not taken
- Dynamic Branch Prediction

3.1. Datenkonflikte (Data Hazards) :

Da viele Befehle von einander abhängig sind, bekommt man schnell Probleme.
Die folgende Befehlsfolge zeigt solche Abhängigkeiten.:

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


Program
execution
order
(in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM
Reg

3.2. Lösungsmöglichkeiten :

3.2.1. Die Softwarelösung :

Der Compiler läßt es nicht zu, daß solche Abhängigkeiten auftreten.

Umstellung des Programmcodes, einfügen von anderen Befehlen oder nop (no operation).

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

Die Softwarelösung :

- Vorteile :
  - Keine Änderung der Hardware notwendig

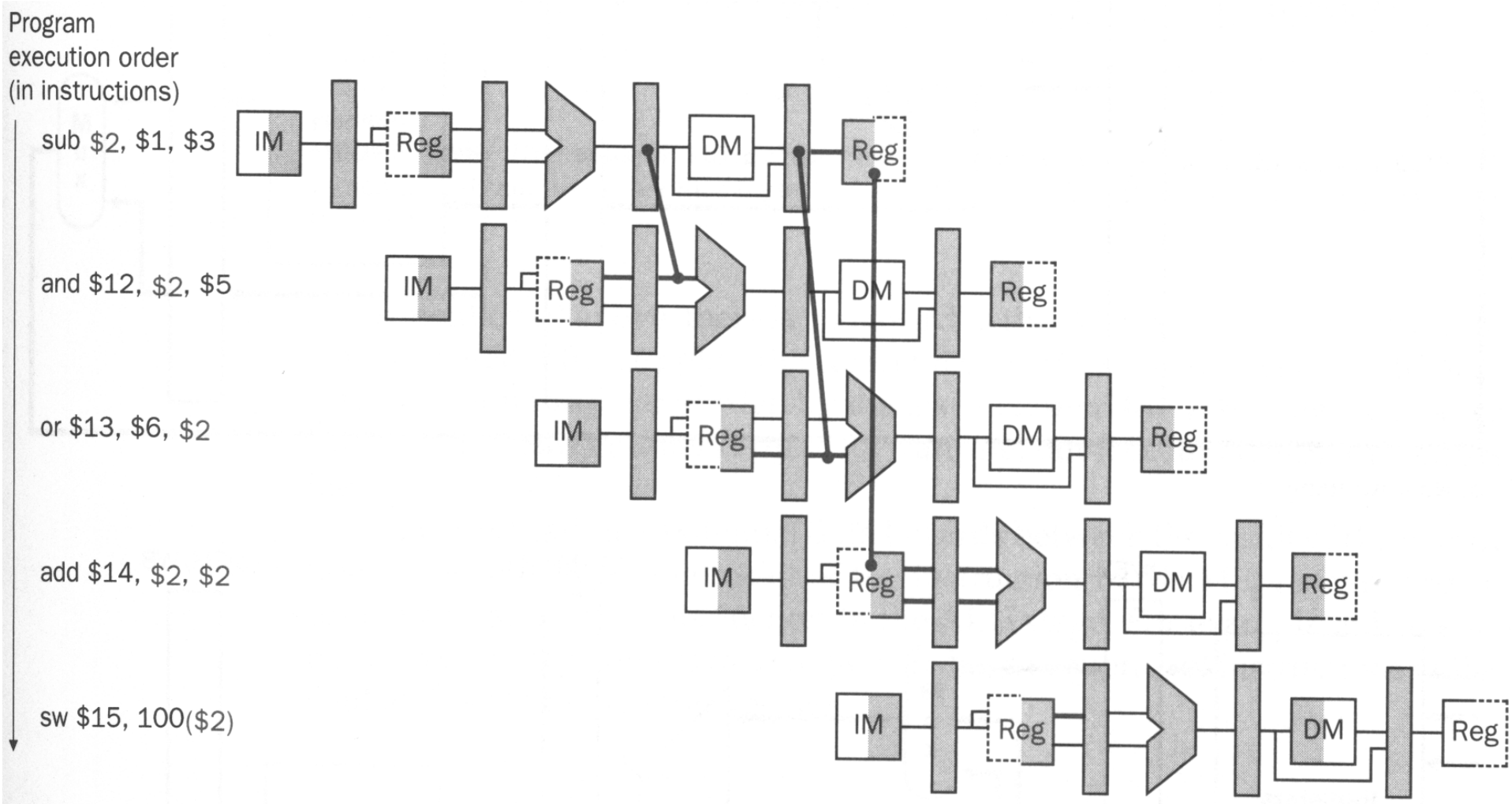
- Nachteile :
  - Solche Abhängigkeiten entstehen zu oft, als es sinnvoll ist, die Problemlösung nur dem Compiler zu überlassen.
  - Die nop- Befehle haben bedeuten Leerlauf des Prozessor.

3.2.2. Forwarding

- Daten früherer Berechnungen sind in Pipeline-Registern, bevor sie im Hauptregister sind

- Erkennen von Abhängigkeiten : (Einteilung in zwei Klassen):
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

- Sobald die Abhängigkeit erkannt ist, muß noch die Hardware entsprechend erweitert werden, dann können auch Pipeline Register als Alu-Input genutzt werden.


Program
execution order
(in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM
Reg

3.2.2.1 Technische Umsetzung: Die Forwarding Unit

1. Testen, ob und wenn ja, welche Abhängigkeit vorliegt, ggf. forwarden :
   1. Ex Hazards :

      if (EX/MEM.RegWrite
      and (EX/MEM.Register.Rd ≠ 0)
      and (EX/MEM.Register.Rd=ID/EX.RegisterRs)) ForwardA= 10

      if (EX/MEM.RegWrite
      and (EX/MEM.Register.Rd ≠ 0)
      and (EX/MEM.Register.Rd=ID/EX.RegisterRt)) ForwardB= 10
   2. Mem Hazards :

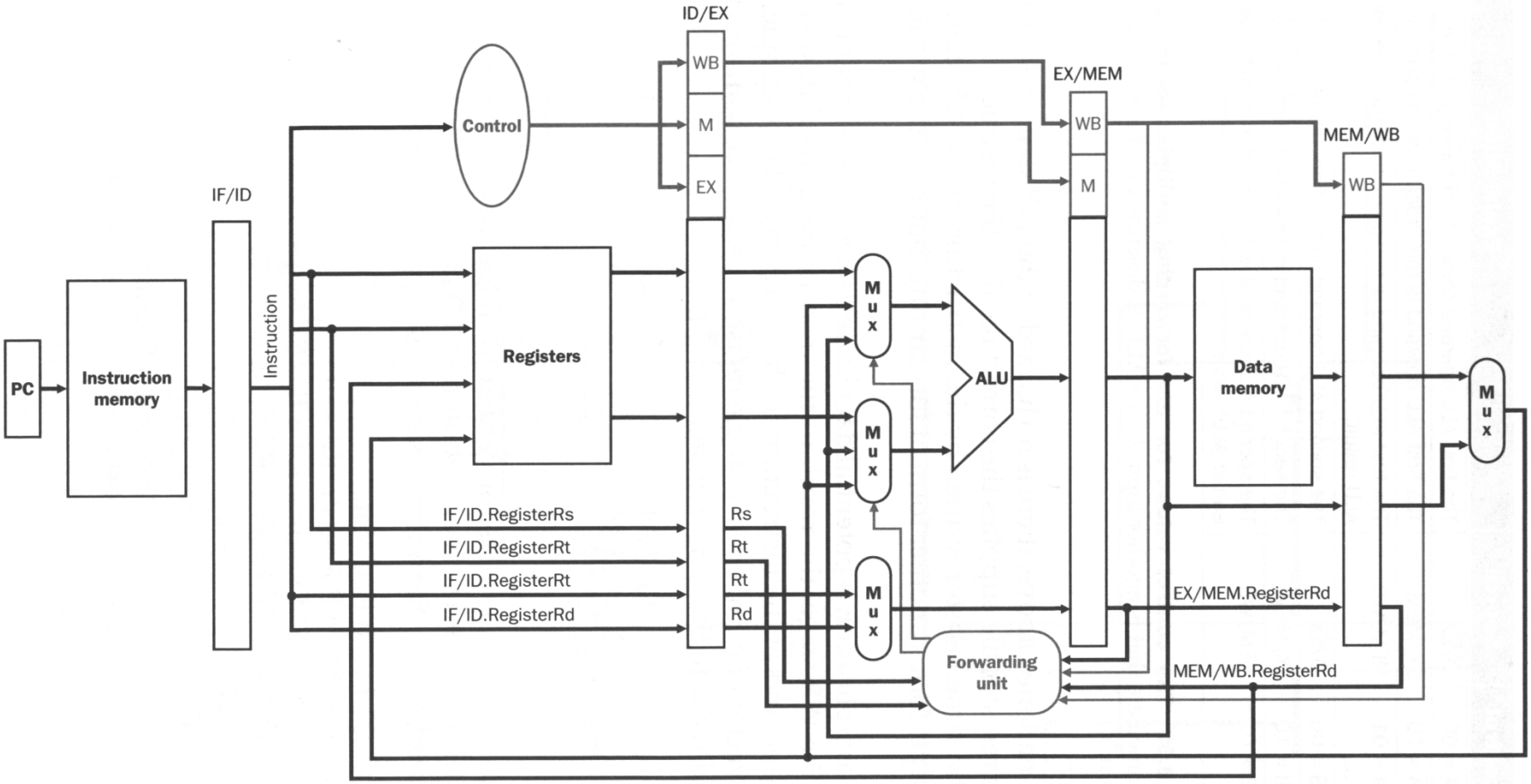
      if (MEM/WB.RegWrite
      and (MEM/WB.Register.Rd ≠ 0)
      and (MEM/WB.Register.Rd=ID/EX.RegisterRs)) ForwardA= 01

      if (MEM/WB.RegWrite
      and (MEM/WB.Register.Rd ≠ 0)
      and (MEM/WB.Register.Rd=ID/EX.RegisterRt)) ForwardB= 01

2. Kontrollwerte für die Forwarding-Multiplexors

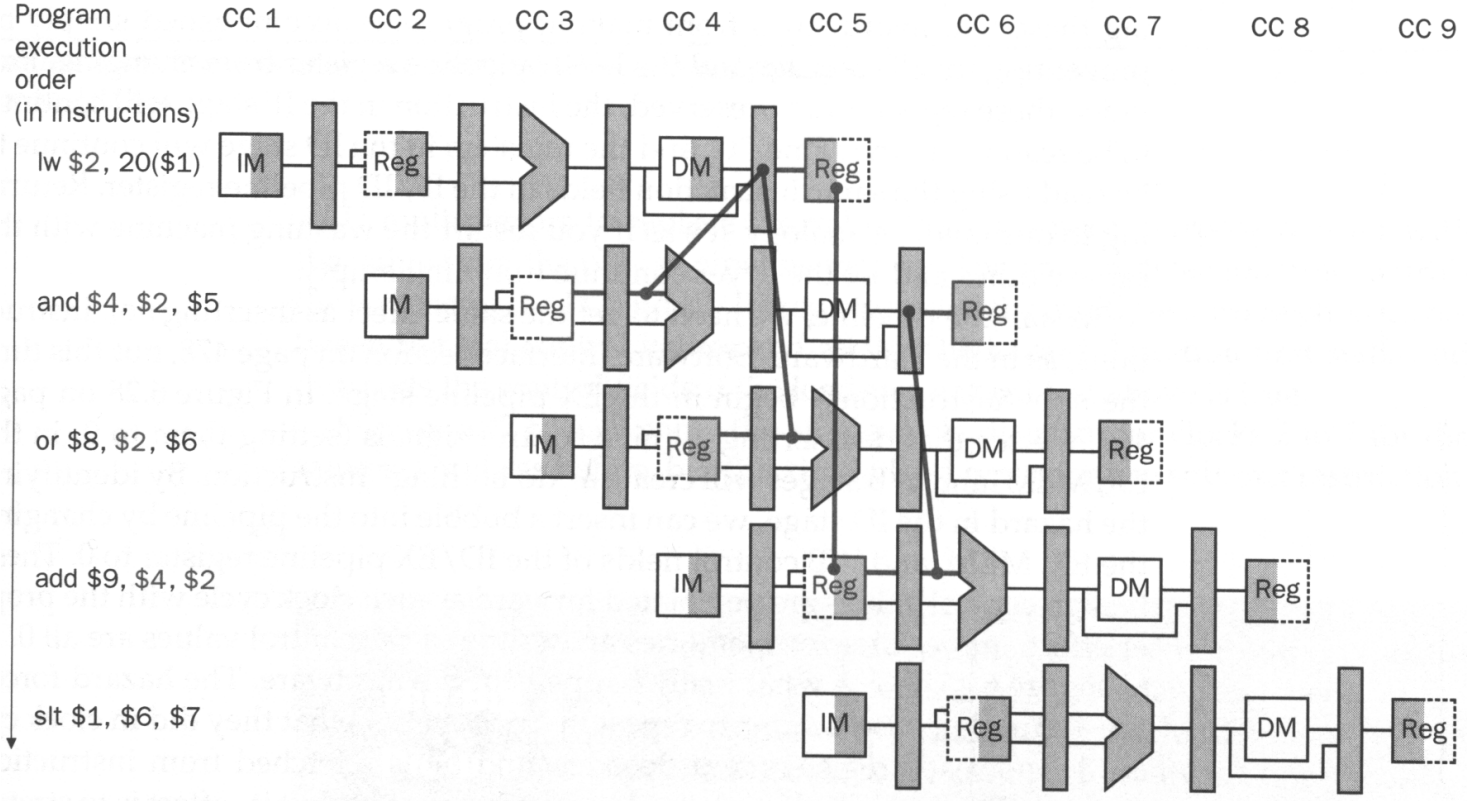
Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
ForwardB = 00	ID/EX	The second ALU operand comes from the register file
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result

3. Das Schaltbild



3.2.3. Datenabhängigkeit und Stall

- Folgt auf einen load-Befehl ein Befehl, der den zu schreibenden Register lesen will, so kann man diese Abhängigkeit nicht so einfach mit Forwarding lösen.



- Die Pipeline muß angehalten werden („stall“)

3.2.4. Hazard detection unit

- Die Hazard Detection Unit muß o.g. Abhängigkeiten erkennen und entsprechend reagieren.

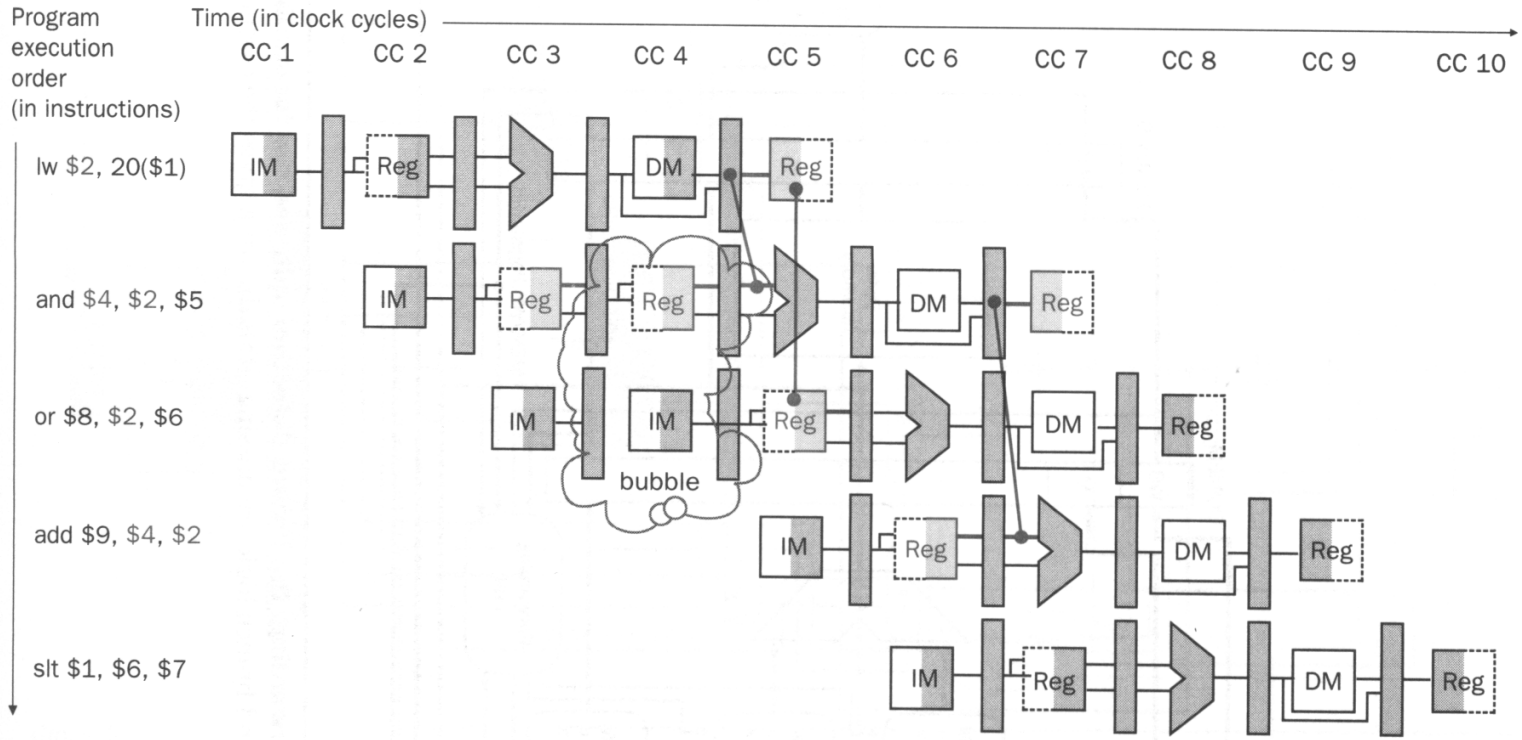
- Die Erkennung :

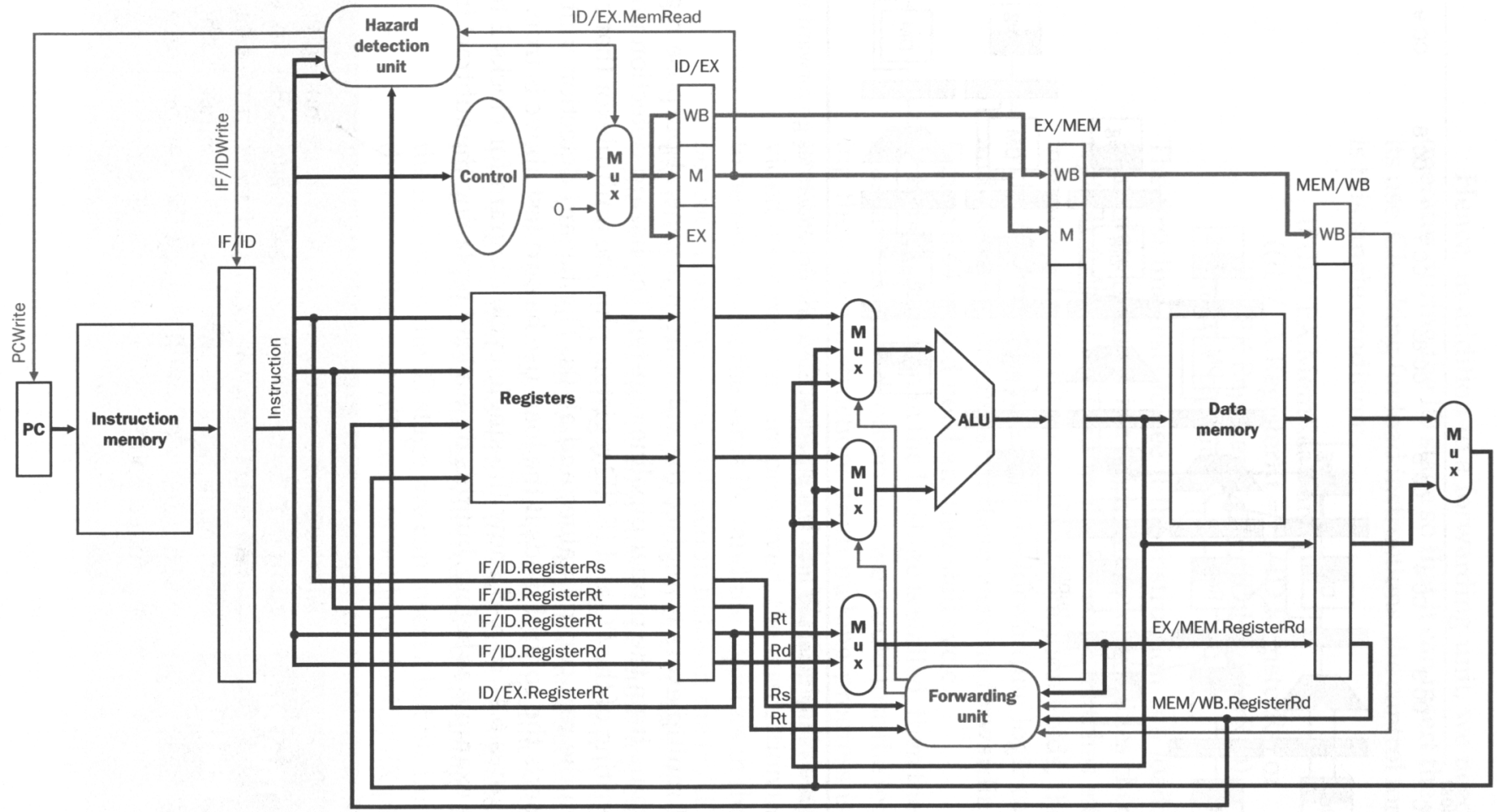
```
if ( ID/EX.MemRead and
((ID/EX.RegisterRt=IF/ID.RegisterRs) or
 (ID/EX.RegisterRt=IF/ID.RegisterRs)))
stall the Pipeline
```

3.2.5. Die technische Umsetzung:

- Wenn der Befehl in der ID-Phase gestoppt werden soll, muß auch der Befehl in der IF-Phase aufgehalten werden, da er sonst verlorengehen würde.

- In der EX-Phase wird praktisch ein „nop“ eingefügt.




Hazard detection unit
ID/EX.MemRead
ID/EX
WB
M
EX
Control
M u x
0
IF/IDWrite
IF/ID
EX/MEM
WB
M
MEM/WB
WB
PCWrite
PC
Instruction memory
Instruction
Registers
M u x
ALU
Data memory
M u x
M u x
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
ID/EX.RegisterRt
Rt
Rd
Rs
Rt
Forwarding unit
EX/MEM.RegisterRd
MEM/WB.RegisterRd

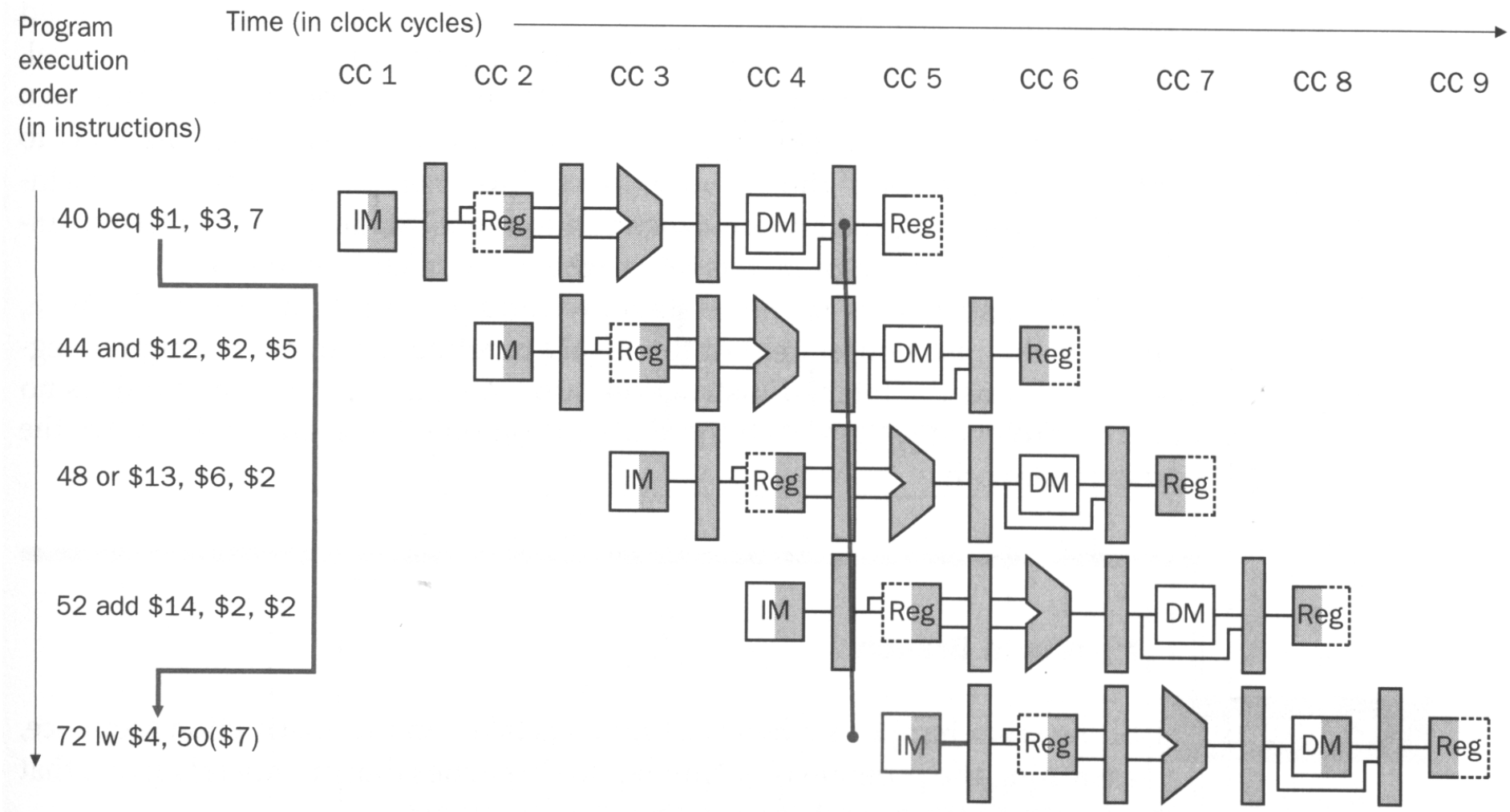
3.3. Sprungkonflikte (Branch Hazards)

Hier gibt es keine Möglichkeit, das Problem zu beheben wie bei den Data Hazards.
Tritt ein Branch-Befehl auf kann man :

- warten, bis entschieden ist an welcher Stelle weitergemacht wird

oder

- raten (besser, versuchen vorherzusagen), wo es weitergeht.
  - rät man richtig, macht man einfach weiter
  - hat man unrecht, müssen alle falschen Befehle gelöscht werden.


Program execution order (in instructions)
Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)
IM
Reg
DM
Reg

3.4. Reducing the Delay of Branches

- Die Branch Entscheidung wird in die ID-Phase vorverlegt :
  - Erhöht, den Hardwareaufwand
  - ist schneller
  - bei falscher Vorhersage muß nur ein Befehl gelöscht werden.

3.5. Branch Prediction

- „assume Branch not taken“ : Die einfachste Art der Sprungvorhersage ist davon auszugehen, die Verzweigung wird nicht genommen

- dynamik Branch Prediction : diese Sprungvorhersage ist besser, da sie sich auf vorherige Ereignisse beruft
  - die einfache Variante ist der 1-Bit Branch-prediction-Buffer
  - genauer ist die 2-Bit Methode

3.6. Sprungentscheidung: Am Beispiel der Schleife

Situation: In einer Schleife wird eine Verzweigung immer neun aufeinander folgenden Durchläufen genommen und dann einmal nicht

- assume branch not taken : 10 % Trefferquote

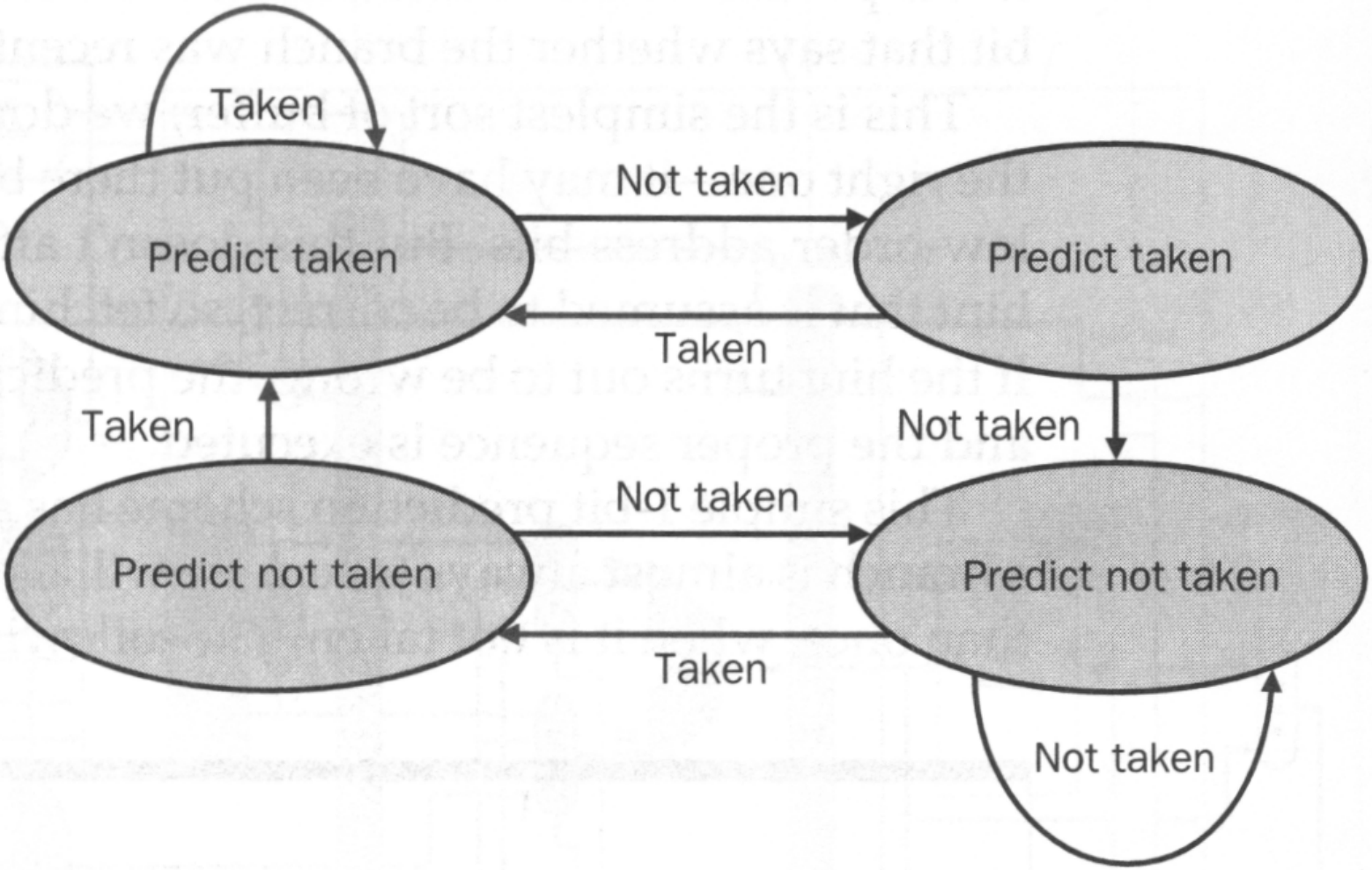
(da immer nur blind geraten wird)

- 1-Bit Methode : 80 % Trefferquote

(es wird nur die letzte Entscheidung berücksichtigt)

- 2-Bit Methode : 90 % Trefferquote

(es wird mehr als nur die letzte Entscheidung betrachtet)


Taken
Predict taken
Not taken
Predict taken
Taken
Taken
Not taken
Predict not taken
Not taken
Predict not taken
Taken
Not taken

4. Exceptions

4.1. Einige Gründe für Exceptions:

- Arithmetischer Overflow
- I/O-Gerät Anfrage
- Undefinierte Instruktion
- Technische Fehlfunktion

Reaktion:

- Die Adresse der die Exception verursachenden Instruktion wird im EPC-Register gespeichert
- Nachfolgende Instruktionen gestoppt
- Vorherige Instruktionen beendet
- Exceptionhandler aufgerufen.

4.2. Probleme bei Exceptions

- Es können mehrere Exceptions gleichzeitig auftreten → Prioritäten vergeben
- Zuordnung der Exceptions zu der sie verursachenden Instruktion

5. Zusammenfassung

5.1. Performancegewinn durch Pipelining

Typisches Beispiel für durchschnittliche Zeit zwischen zwei Instruktion:

Singlecycle: 8 ns
Multicycle: 6,3 ns
Pipelining: 2,34 ns

→ Pipelining ist 3,4 mal schneller als Singleycle
→ und 2,7 mal schneller als Multicycle

5.2. Ausblick: Weitere Verbesserungen

5.2.1. Superpipelining

→ Aufteilung des Datapath in mehr Phasen

Je mehr Phasen, je mehr Instruktionen kann man gleichzeitig ausführen.

Dabei ist darauf zu achten, dass die einzelnen Phasen ausgeglichen sind.

5.2.2. Superscalar Pipelining

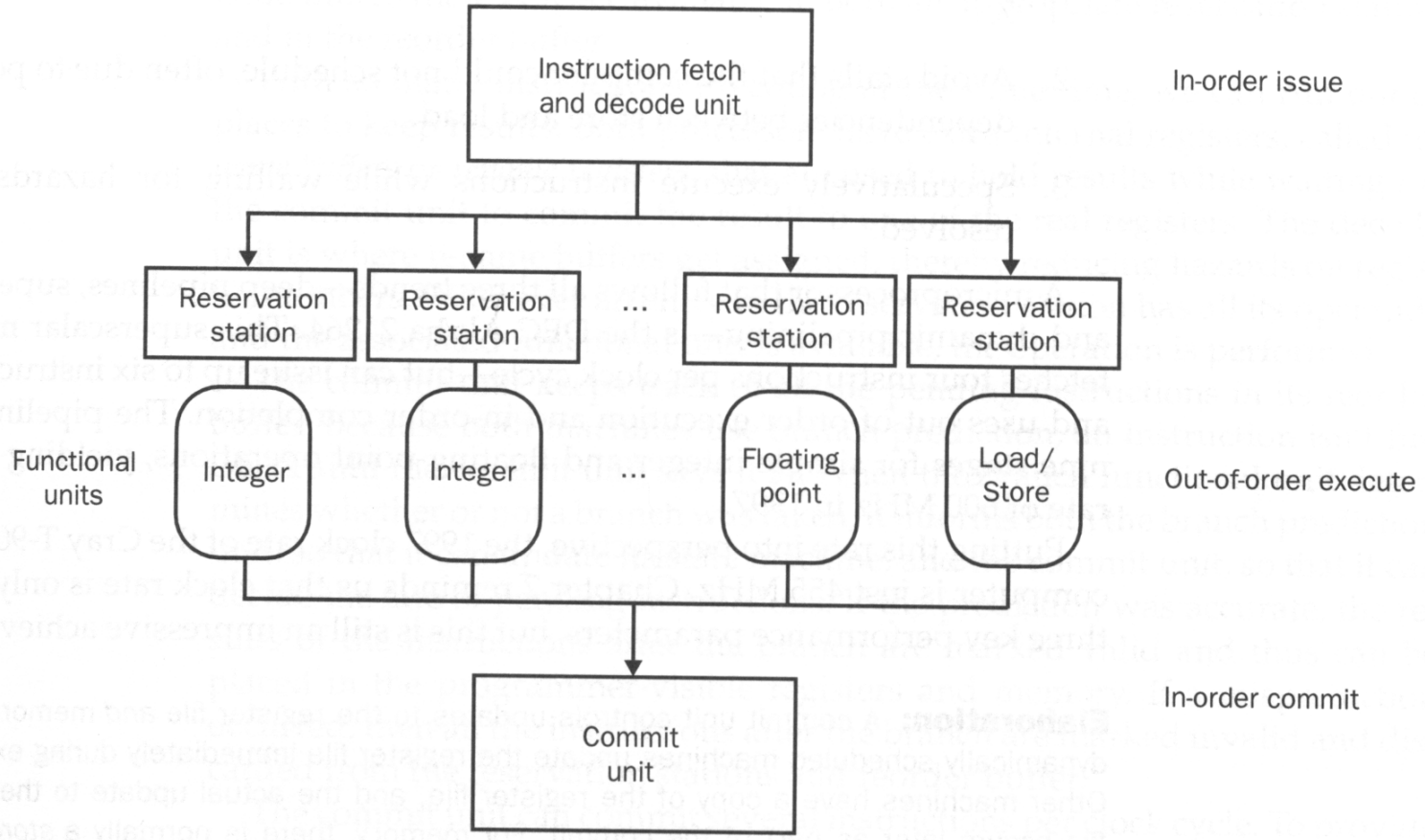
→ Durch Multiplizierung der Hardware können mehrere Instruktionen gleichzeitig ausgeführt werden.

Probleme:

- Durch Abhängigkeiten kann oft doch nur wenige oder sogar nur eine Instruktion ausgeführt werden
- Deutlich komplexere Hardware

5.2.3. Dynamic Pipelining

→ Versuch, Pipeline Konflikte durch “Out of order”-Bearbeitung zu vermeiden.



5.3. Zusammenfassung

- Durch Parallelarbeit erhöht Pipelining die Durchsatzrate an Instruktionen
- Daten- und Sprungkonflikte verringern die theoretisch erreichbare Geschwindigkeitsverbesserung und erhöhen die Komplexität des Datapath
- Datenkonflikte lassen sich größtenteils durch Forwarding lösen
- Bei Sprungkonflikten kann man durch Branchprediction die Verzögerung reduzieren.
- Der Übergang zu längeren Pipelines, Superscalar- und Dynamischem Pipelining hat stark zu den Performancegewinnen der letzten Jahre beigetragen.