Kapitel 9 Befehlspipelining

9.1. Aufbau einer Befehlspipeline

Ein typischer Befehl in einer Maschine mit einem RISC-artigen Befehlssatz besteht aus den Operationen:

H : Befehl holen (instruction fetch) in einem Zugriff von Wortbreite

D : Befehl dekodieren

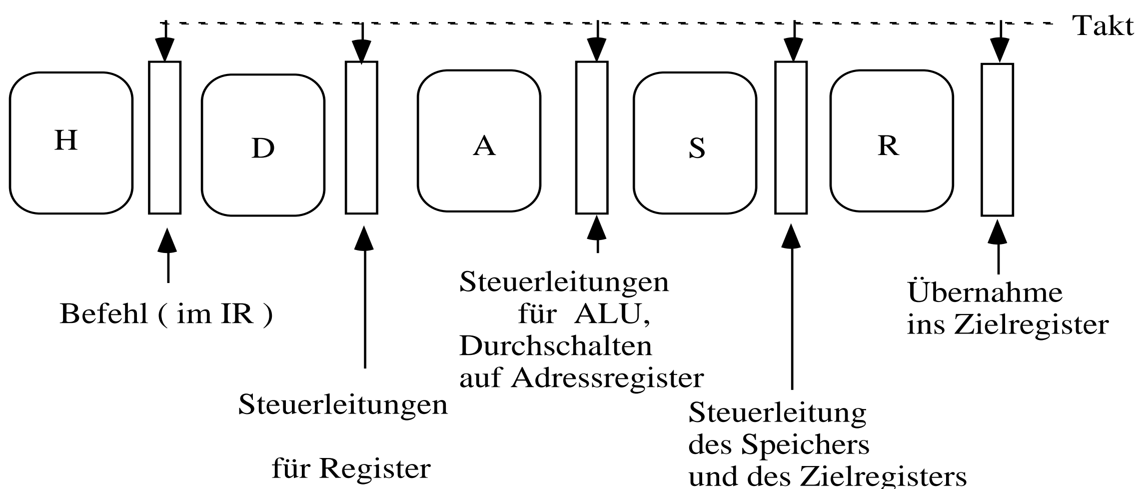
A : Befehl ausführen und Adressrechnungen durchführen

S : Speicherzugriff (bei Load-Store-Befehlen)

R : Rückschreiben

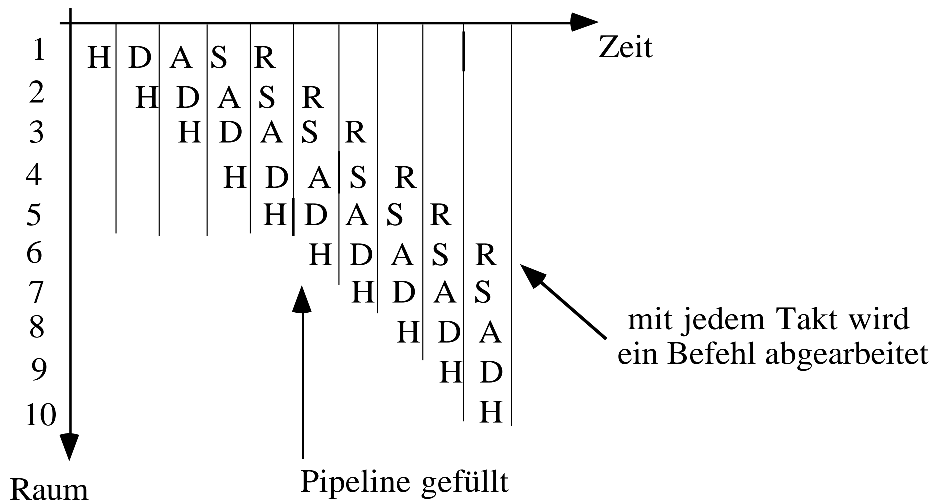
Bei CISC-Rechnern zerfällt das Holen in mehrere Abschnitte, ggf. muß mehrfach auf den Speicher zugegriffen werden, wenn der Befehl z. B. 6 Byte lang ist. Beim Ausführen muß ebenfalls mehrfach auf den Speicher zugegriffen werden, wenn MOVE-Befehle abgearbeitet werden.

Wenn Befehle aus den o. g. 5 Komponenten bestehen, kann die Abarbeitung in einer **Pipeline** parallelisiert werden: Die Teilergebnisse werden in Registern zwischengespeichert und die Abarbeitung überlappend durchgeführt: Wenn der erste Befehl dekodiert wird, kann der nächste schon geholt werden, ...



Die Geschwindigkeit wird bestimmt durch den langsamsten Verarbeitungsschritt; er bestimmt die Taktrate, mit der die Zwischenergebnisse in die Latches der Pipeline übernommen werden (Geleitzugprinzip: der langsamste Schiff bestimmt die Geschwindigkeit).

Nach 5 Schritten ist die Pipeline gefüllt und danach ist mit jedem Takt ein Befehl vollständig abgearbeitet.



Sei die mittlere Ausführungszeit für einen Befehl (HDASR)

4 x 50 ns + 1 x 60 ns = 260 ns.

In der Pipeline entsteht ein Overhead von 5 ns durch die Übernahme in das Latch.
Dann ist (Geleitzugprinzip) die Taktperiode für die Pipeline 65 ns und die Beschleunigung der Befehlspipeline

$$\frac{\text{mittlere Ausführungszeit ohne Pipeline}}{\text{Taktperiode der Pipeline}} = \frac{260}{65} \approx 4 .$$

Für die Beispielarchitektur der DLX-Maschine sind die Operationen in den Stufen der Pipeline hier dargestellt.

Stufe	**PC-Einheit**	**Speicher**	**Datenpfad**
H	PC <-- PC + 4	IR <-- Mem(PC)	
D	PC1 <-- PC	IR1 <-- IR	A <-- Rs1; B <-- Rs2
A			DMAR <-- A + $(IR1_{16})^{16}$ ## $IR_{16...31}$; SMDR oder ALUoutput <-- A *op* (B oder $(IR1_{16})^{16}$ ## $IR_{16...31}$) oder ALUoutput <-- PC1 + $(IR1_{16})^{16}$ ## $IR_{16...31}$ cond <op- (0 A) ;
S	if (cond) PC <-- ALUoutput	LMDR <-- Mem(DMAR) oder Mem(DMAR) <-- SMDR	ALU output1 <-- ALUoutput
R			Rd <-- ALUoutput1 oder LMDR

Dabei wird in jeder Stufe in PC-Einheit, Speicher und Datenpfad der Maschine ggf. parallel etwas ausgeführt.

9.2. Hazards

Das ideale Verhalten der Befehlsabarbeitung wird durch drei Arten von Störungen beeinträchtigt:

- Struktur-Hazards: Ressourcenkonflikte
- Daten-Hazards: Datenabhängigkeiten in Befehlen
- Steuer-Hazards: bedingte Sprünge im Programm.

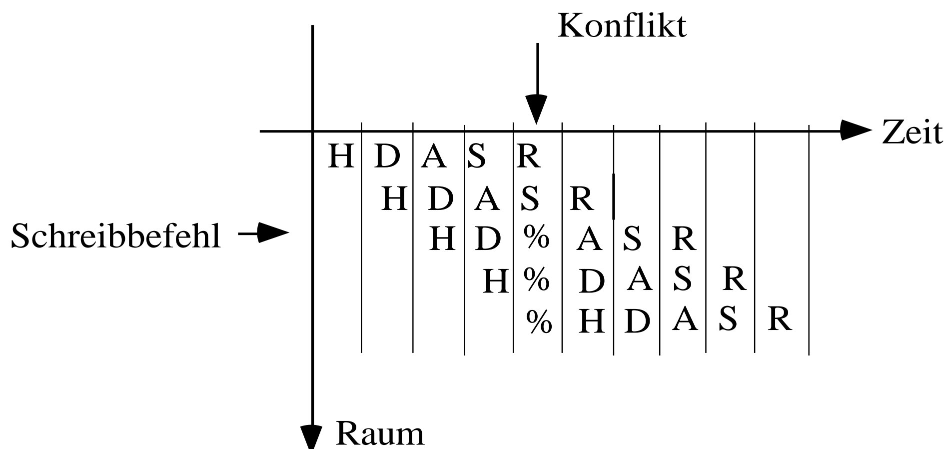
Wenn sie nicht durch konstruktive Maßnahmen abgefangen werden können, erzeugen sie Wartezyklen (stalls oder bubbles) in der Pipeline.

9.2.1. Struktur-Hazards

Sie entstehen durch Ressourcenkonflikte:
Befehle brauchen Ressourcen der Maschine und Befehlskombinationen können Ressourcen erfordern, die nicht zugleich angefordert werden können.

Beispiel:

In einer v. Neumann-Architektur das Holen eines Befehls und das Schreiben in den Speicher: Beide benötigen die gleichen Adressleitungen und Datenpfade.



Abhilfe:

Interne Harvard-Architektur durch einen Programm-Cache und davon getrennten Zugriff auf Daten im Speicher (Datencache oder Zugriff über MMU).

9.2.2. Daten-Hazards

Einem Register wird durch einen Befehl ein Datum zugewiesen. Das Register wird in einem Folgebefehl gelesen. Das darf erst geschehen, wenn der erste Befehl die Daten in das Register geschrieben hat, sonst wird ein älterer (falscher) Inhalt gelesen.

Beispiel:

ADD R1, R2, R3 - R1 ist Zielregister -

SUB R4, R5, R6

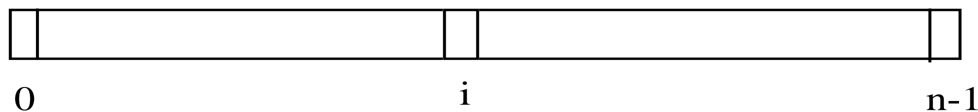
S R1, ADR - R1 darf erst gelesen werden, wenn der ADD-Befehl vollständig ausgeführt wurde.

Da solche Daten-Hazards unvermeidlich sind, müssen sie durch konstruktive Maßnahmen entweder sicher erkannt oder umgangen werden können.

9.2.2.1. Scoreboard

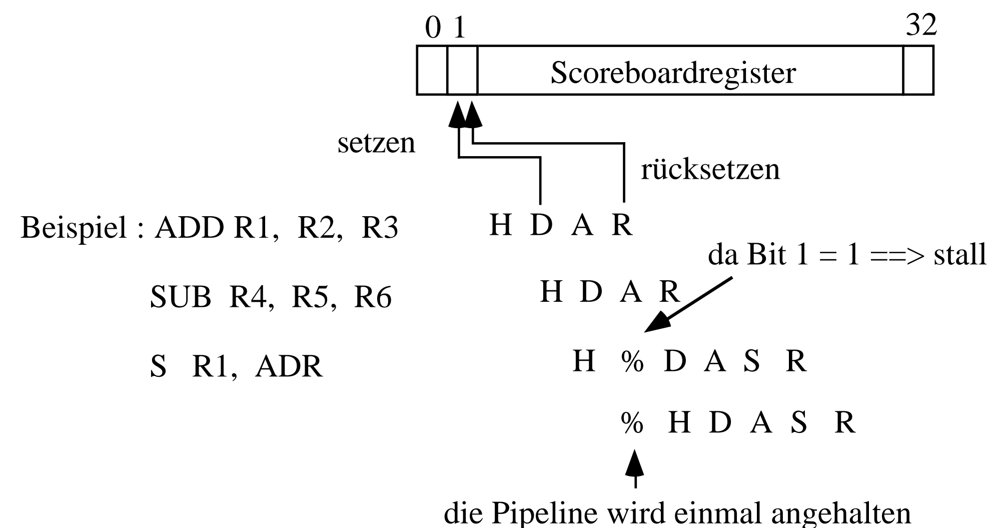
Man führt mit ein Register (Scoreboard) mit M Bit, wobei M die Anzahl der durch Befehle ansprechbaren Arbeitsregister ist (meist 32).

Jedem Register ist in dem Scoreboard ein Bit zugeordnet.



Es wird Bit i gesetzt, wenn Register i als Zielregister erkannt wurde, und gelöscht, wenn der zugehörige Befehl ausgeführt ist.

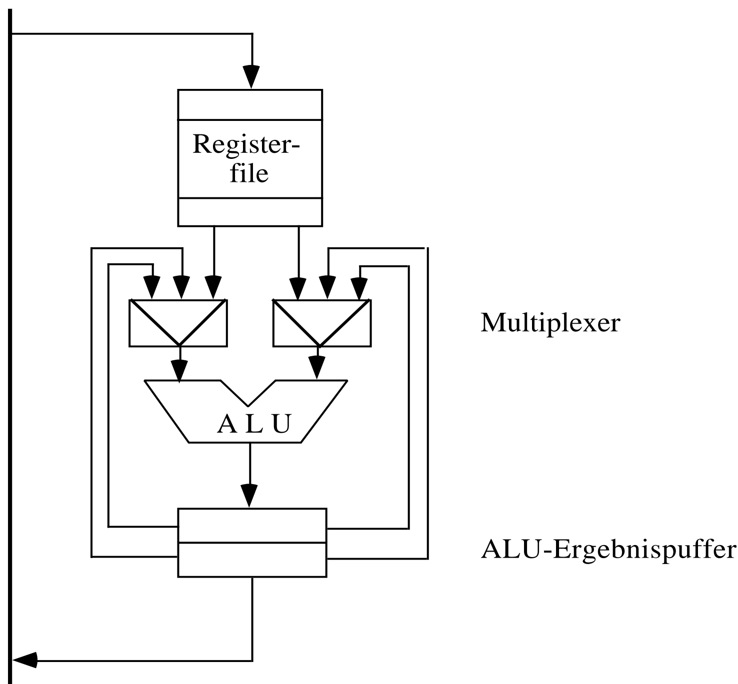
Ein Befehl, der Register i anspricht, muß warten bis das Bit i Null ist.



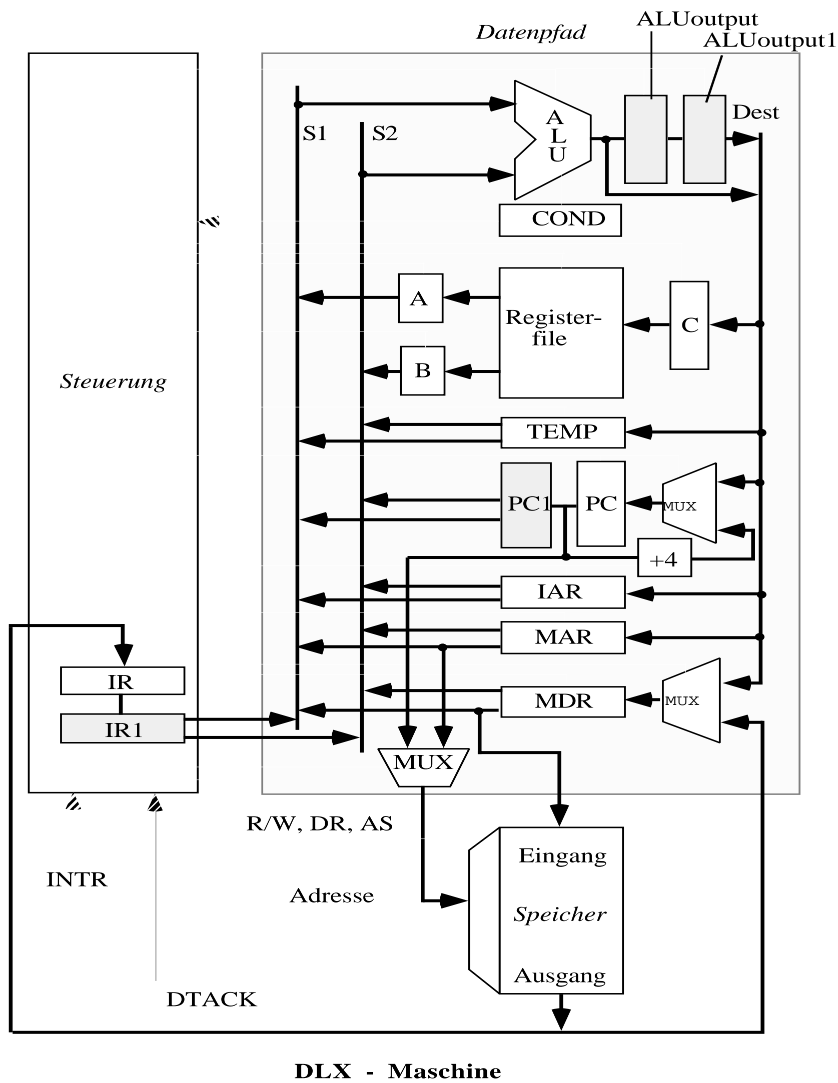
Die Pipeline wird hier einen Takt angehalten.

9.2.2.2. Bypasseinheit in der ALU

Sieht man in der ALU Ergebnispuffer vor, die ohne den Umweg über das Registerfile direkt auf die ALU rückgekoppelt werden können, dann kann ein Rechnerergebnis weiterverwendet werden, ohne daß ein Hazard auftritt.



Als Beispiel für einen Rechner mit Bypasseinheit möge die DLX-Maschine dienen:



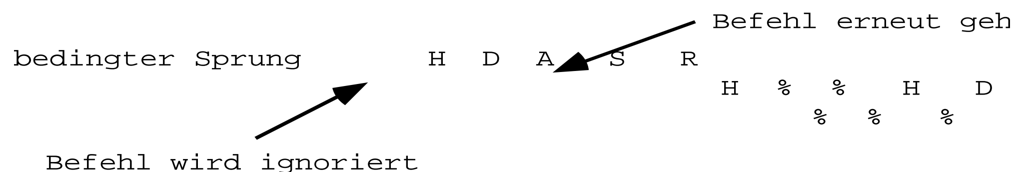
DLX - Maschine

9.2.2.3. Umordnen von Befehlen

Mögliche Daten-Hazards können vom Compiler erkannt und dadurch entschärft werden, daß er Befehlssequenzen umordnet, so daß die Datenabhängigkeiten erhalten bleiben aber keine Konflikte auftreten. Ggf. muß der Compiler NOP's einschieben.

9.2.3. Steuer-Hazards

Sie treten auf bei bedingten Sprungbefehlen:
Erst wenn die Bedingung ausgewertet ist, kann der Folgebefehl geholt werden, denn dann erst steht fest, ob die Folgeadresse die nächste Adresse oder das angegebene Sprungziel ist. Man muß die Pipeline leerlaufen lassen.



Um das Leerlaufen lassen zu vermeiden, gibt es eine Reihe von Strategien.

9.2.3.1. Zwei parallele Pipelines

Es werden in zwei parallelen Pipelines beide Folgeadressen berechnet und in beide Stränge die Folgebefehle geladen.

Es müssen aber datenverändernde Befehle solange verzögert werden, bis das Ergebnis der Abfrage der Bedingung feststeht, denn sonst würden Register ggf. mit falschen Daten überschrieben.

Wenn das Ergebnis der Abfrage feststeht, wird ein Strang verworfen.

Nachteil: Folgen von Sprungbefehlen erzeugen das Leerlaufen der Pipelines.

9.2.3.2. Intelligentes Raten

Bei einem bedingten Sprungbefehl wird, abhängig von der Art des Sprunges, die Folgeadresse geladen:

- Rücksprünge werden fast immer ausgeführt (repeat-until-Schleifen)
- Vorwärtssprünge werden selten ausgeführt
  (while-do-Schleifen, if-then-else-Konstrukte)

Die Pipeline wird angehalten, wenn datenverändernde Befehle auszuführen sind, bis das Sprungziel feststeht. Wurde falsch geraten, muß die Pipeline geleert und die andere Folgeadresse geladen werden.

9.2.3.3. Compiler-Optimierung

Man läßt durch den Compiler die Befehle bei bedingten Sprüngen so umordnen und verdoppeln, daß in beiden Strängen zunächst gleiche Folgebefehle stehen und das Sprungziel ermittelt ist, ohne daß Registeränderungen vorgenommen werden, die nur in einem Zweig vorkommen. Das kann auch durch Einfügen von NOP's geschehen.

Dann können keine Steuer-Hazards auftreten, aber es sind ggf. viele NOP's eingestreut worden.

9.2.3.4. Zusätzliches Befehlspuffer

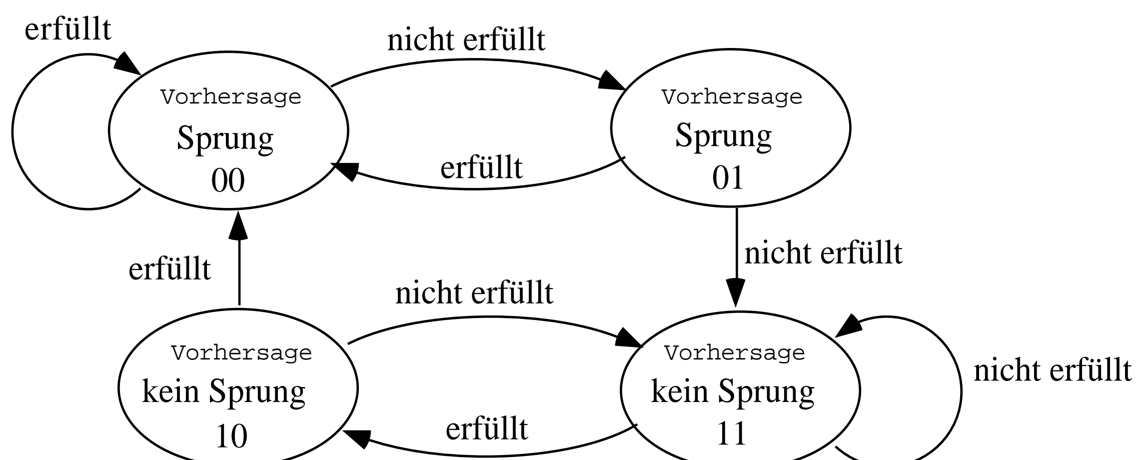
Man sieht einen zusätzlichen Befehlspuffer vor, in den der Folgebefehl < PC + 4 > eingelesen wird. Wenn nicht gesprungen wird, steht damit ohne neuen Holvorgang der Folgebefehl sofort zur Verfügung.

9.2.3.5. Dynamische Sprungvorhersage

Man führt einen einfachen Automaten (branch prediction buffer) mit, der für den nächsten Sprung eine Vorhersage macht.

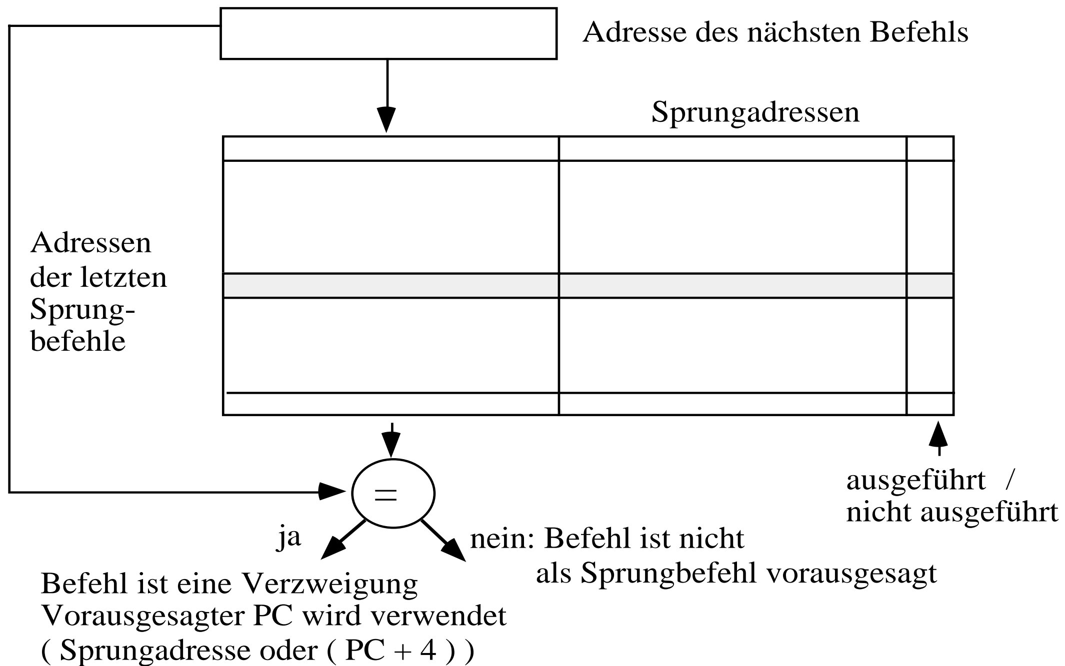
Erst nach zwei falschen Vorhersagen wird der Zustand gewechselt.

Der Automat macht eine Vorhersage, ob die Folgeadresse oder das Sprungziel geladen werden soll, wenn ein bedingter Sprung dekodiert wurde.



9.2.3.6. Cache für Sprünge

Man führt in einem Cache die letzten Adressen der Sprungbefehle mit.

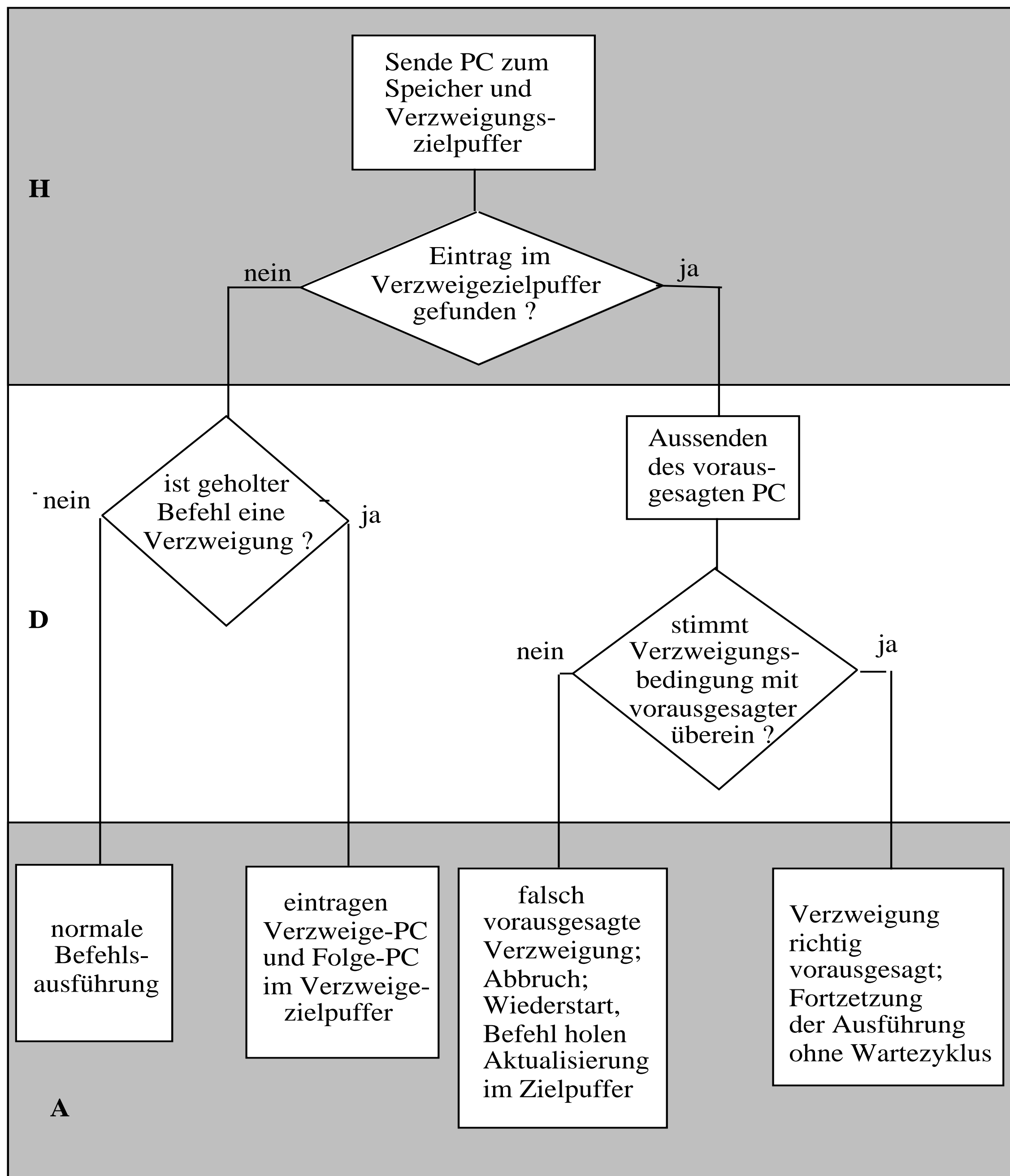


Der Normalfall ist, daß die Adresse des nächsten Befehls keinen Sprungbefehl adressiert. Dann bleibt der Cache wirkungslos.

Wenn der nächste Befehl ein Sprungbefehl ist, dann wird das vorhergesagte Sprungziel aus dem Cache verwendet, wenn das Ausführungsbit gesetzt ist, sonst wird die Folgeadresse als Sprungziel benutzt.

Änderungen in Cache:

1. Das Sprungziel wurde ausgeführt wie vorhergesagt
   => keine Änderung;
   das Sprungziel wurde nicht angesprungen
   => Änderung des Bits.
2. Der Befehl wurde nicht als Sprungbefehl vorausgesagt, war aber ein Sprung.
   => Er wird in den Cache eingeschrieben und eine alte Sprungadresse verdrängt. Im Datenteil des Cache wird das Sprungziel eingetragen und im Ausführungsbit vermerkt, ob gesprungen wurde oder nicht.



9.3. Behandlung von Interrupts

Man kann auf einen Interrupt in zwei Weisen reagieren:

- Es wird die Pipeline gerettet, dazu das PSW, und dann die Trapvektoradresse geladen. Das erfordert zusätzlichen Hardwareaufwand, sichert aber die sofortige Reaktion.
- Man verzögert den Interrupt, lädt keinen neuen Befehl, läßt die Pipeline leerlaufen und rettet das PSW wie gehabt.
  Dann erst wird die Trapvektoradresse geladen.