Script zum Seminarvortrag

„Out of Order Execution (Tomasulo)“

September/Oktober 1997

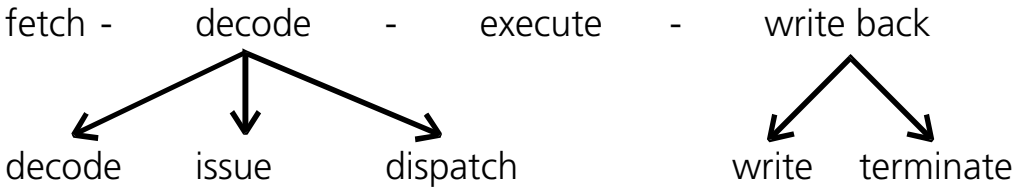
Daniel Kröning

Überblick

1. Einführung in Out of Order Execution

Phasen der Abarbeitung einer Instruktion

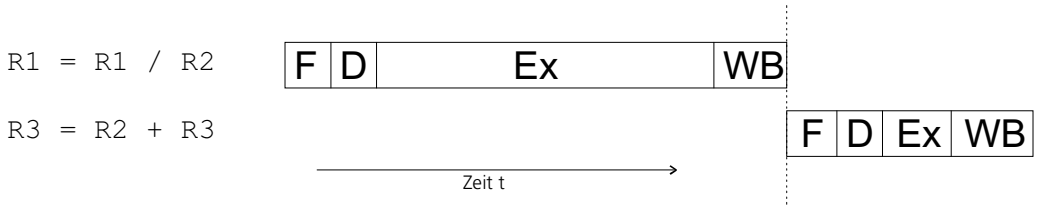
Die Abarbeitung einer Instruktion läßt sich allgemein in folgende Phasen aufteilen:



Sequentielle Maschine

Bei der sequentiellen Abarbeitung der Instruktionen werden diese Schritte in exakt dieser Reihenfolge für jede Instruktion in der ursprünglichen Reihenfolge der Instruktionen durchgeführt. Die nächste Instruktion wird also erst geladen (fetch), wenn die vorausgehende Instruktion vollständig ausgeführt ist, also wenn das Writeback abgeschlossen ist.

Beispiel 1:



Dies bedeutet, daß weite Teile der CPU die meiste Zeit unbenutzt bleiben, obwohl sie ihre Aufgabe längst beendet haben und zur Aufnahme weiterer Instruktionen oder Daten bereit wären.

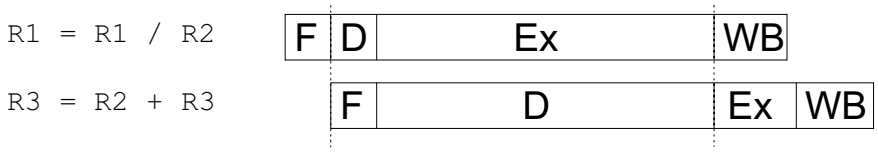
Abarbeitung mit Pipeline

Durch Überlappung der Ausführung (Pipelining) können mehrere Instruktionen gleichzeitig bearbeitet werden. Dadurch werden die Function Units besser genutzt und somit der Durchsatz verbessert.

Die einzelnen Phasen der einzelnen Instruktionen werden dabei in folgender Reihenfolge ausgeführt:

Fetch1	Decode1	Execute1	WriteBack1		
	Fetch2	Decode2	Execute2	WriteBack2	
		Fetch3	Decode3	Execute3	WriteBack3

Beispiel 2:



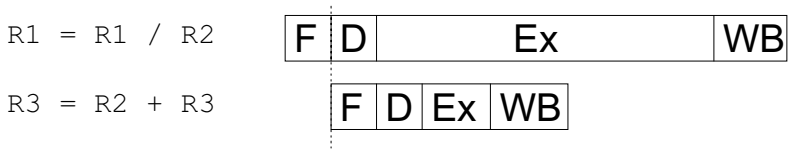
Die Phase „Decode“ der zweiten Instruktion wird in diesem Beispiel verlängert, da die Einheit, die für die Phase Execute notwendig ist, noch durch die (sehr lange dauernde) Division belegt ist.

Beim Pipelining kann es also immer noch zu erheblichen Idle-Zeiten kommen, und zwar

- bei Datenabhängigkeiten
- weil die Function Units stark unterschiedliche Latenzen haben (im Beispiel: die Division dauert deutlich länger als die Addtion).

Out of Order Execution

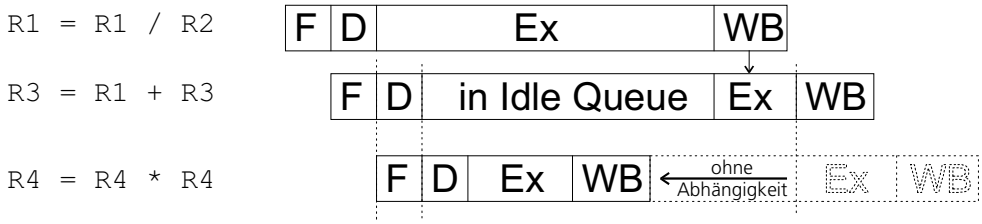
Beim Beispiel 2 bestehen keinerlei Abhängigkeiten zwischen den beiden Instruktionen. Die Division und die Addition könnten also eigentlich gleichzeigig bearbeitet werden. Das Ergebnis der Addition könnte sogar vor dem der Division ins Zielregister geschrieben werden:



Die Instruktionen werden dann allerdings nicht mehr in der Programmreihenfolge ausgeführt, man spicht dann von **Out of Order Execution**.

Wenn Abhängigkeiten auftreten, muß die Kontrolle geeignete Maßnahmen ergreifen, um eine korrekte Ausführung der Instruktionen sicherzustellen. Normalerweise bedeutet dies, daß in bestimmten Phasen einer Instruktion auf die Beendigung anderer Instruktionen gewartet werden muß.

Beispiel 3:



In diesem Beispiel muß mit der eigentlichen Berechnung der Summe (Phase „Execute“) in der zweiten Instruktion gewartet werden, bis das Ergebnis der ersten Instruktion zur Verfügung steht. Ohne diese Wartezeit würde nach der Berechnung im Register R3 die falsche Summe stehen. Die dritte Instruktion (die Multiplikation) kann wieder ohne Wartezeit ausgeführt werden, da sie in keinerlei Abhängigkeit zu den beiden anderen Instruktionen steht. Sie wird auch vor den anderen beendet („Out of Order Termination“), präzise Interrupts sind dann nur mit zusätzlichem Hardwareaufwand realisierbar [SP88][1].

Ziel

An einen Scheduling-Mechanismus werden also folgende Anforderungen gestellt:

- Schnelle Abarbeitung der Instruktionen
- Gute Ausnutzung der Function Units
- Schutz der Daten (Register/RAM) bei Abhängigkeiten

[1] Der Algorithmus von Tomasulo implementiert im Original Out of Order Termination. Durch die zusätzliche Verwendung von Recordbuffern kann In Order Termination und damit präzise Interrupts realisiert werden.

2. Grundlagen Tomasulo Algorithmus

Geschichte

Der Algorithmus wurde 1967 von Robert M. Tomasulo für eine IBM 360/91 spezifiziert [TO67]. Verwendung fanden im „Original“ Operationen mit zwei Registern (z.B. R1+=R2) und mehrere sequentielle Function Units für jede Rechenoperation.

Der Algorithmus ist, wie im Folgenden beschrieben, auf die heute üblichen Befehle mit drei Registern (z.B. R1=R2+R3) und pipelined Function Units erweiterbar. Er wird in vielen CPUs benutzt, so z.B. im PowerPC, Pentium-Pro oder AMD K5.

Kurzbeschreibung

- Der Algorithmus von Tomasulo ist ein Hardware Alogithmus zur automatischen Ausnutzung mehrerer Execution Units [TO67].
- Der Algorithmus ist für eine beliebige Anzahl von Registern und Execution Units geeignet, braucht nur wenige, globale Datenstrukturen und ist daher gut skalierbar.
- Hardware-Mechanismen: Die Grundlegende Idee des Verfahrens ist der **Common Data Bus** (CDB). Über den CDB werden Registerinhalte transportiert, die nach Herkunft (Ergebnis einer Execution Unit oder Inhalt eines Load Buffers) über **Tags** identifiziert werden.
- Schreibzugriff auf den CDB wird von einer zentralen, prioritätengesteuerten Kontrolle auf Anfrage vergeben. Lesezugriff kann gleichzeitig an mehreren Stellen durch **Bus-Snooping** erfolgen.

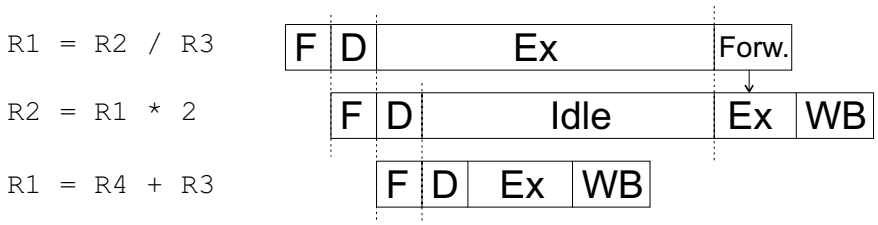
Besondere Eigenschaften

Der Algorithmus von Tomasulo verfügt über zwei besondere Eingenschaften:

- Forwarding von Ergebnissen: Gleichzeitige Weitergabe von Rechenergebnissen über den CDB an alle Function Units und Register, die dieses Datum benötigen.

- Der Algorithmus von Tomasulo implementiert Register Renaming. Instuktionen mit WAW-Abhängigkeiten[2] können gleichzeitig ausgeführt werden. Veraltete Daten werden erst gar nicht ins Register File geschrieben.

Beispiel:

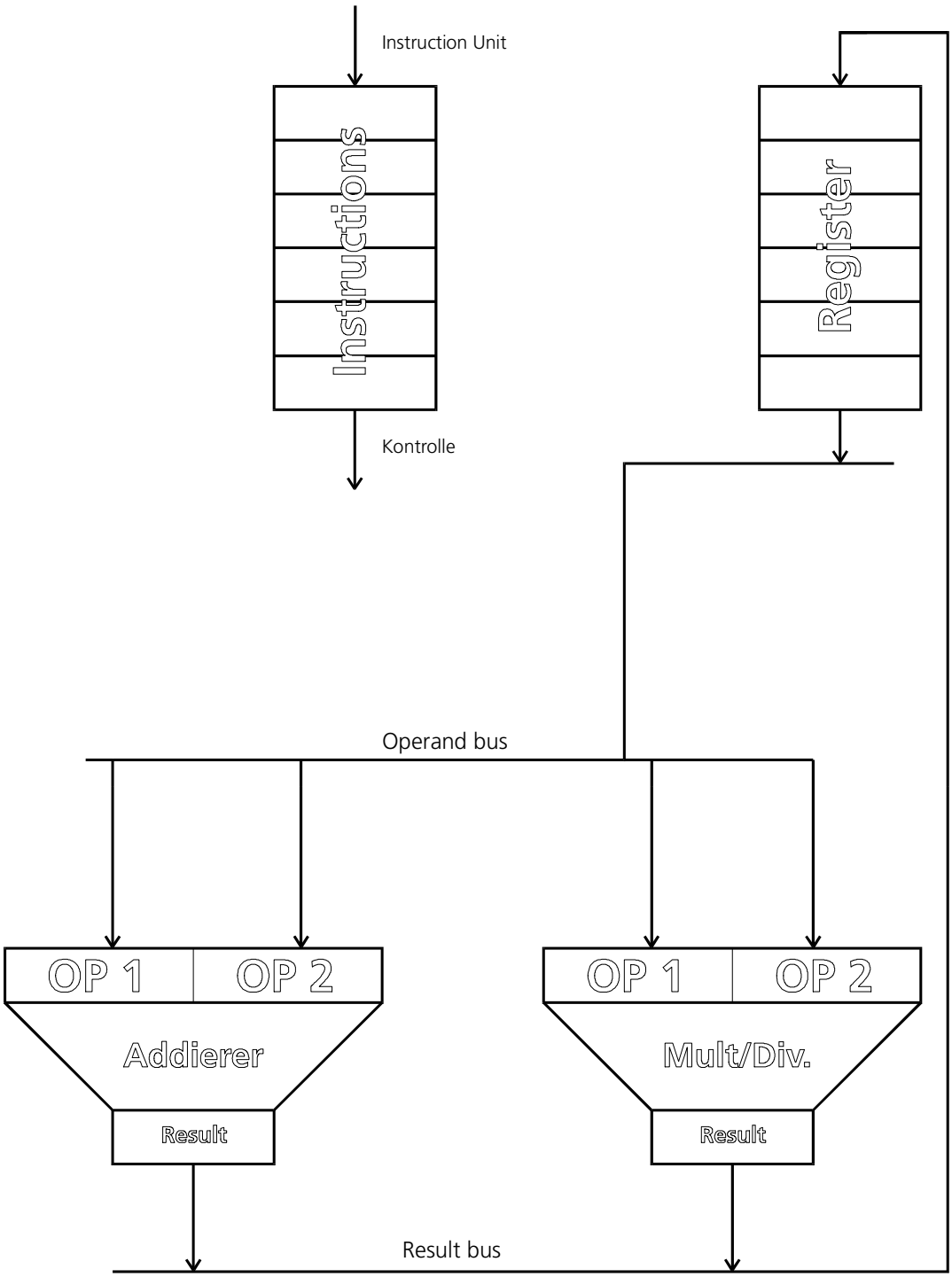


Das Ergebnis der ersten Instruktion wird nicht ins Register geschrieben, sondern nur an die zweite Instruktion weitergeleitet (Forwarding).

[2] Write after Write data hazard: D.h. die Gefahr, daß ein veraltetes Rechenergebnis den aktuellen Wert eines Registers überschreibt.

Aufbau der skalaren CPU

In der folgenden Abbildung ist der schematische Aufbau einer skalaren CPU (ohne Memory-Interface) gegeben:



Nicht angegeben sind die für die Kontrolle notwendigen Datenpfade.

[Exkurs: Busy-Bits]

Eine einfache Methode, die parallele Ausführung von Instruktionen zu implementieren, ist das Hinzufügen je eines „Busy-Bits“ zu jedem Register. Die Busy-Bits implementieren eine Regel zur Erkennung von Abhängigkeiten von Instruktionen auf einfache Weise:

- Eine Operation wird ausgeführt, wenn keines der beteiligten Register ein gesetztes Busy-Bit hat.
- Vor der Ausführung einer Operation wird beim Zielregister (Sink) das Busy-Bit gesetzt.
- Nach der Beendigung der Operation wird das Bit wieder gelöscht.

Unabhängige Instruktionen lassen sich so ohne großen Aufwand auf mehrere Instruktionseinheiten verteilen. Tritt jedoch eine Abhängigkeit auf, geht der Vorteil gegenüber der sequentiellen Abarbeitung verloren, da dann bis zur Beendigung der betroffenen Instruktion gewartet werden muß. Um einen Nutzen von dem „Busy-Bits“ Verfahren zu haben, muß der Programmierer (oder der Compiler) also ggf. durch Umsortieren dafür sorgen, daß die Instruktionen möglichst wenig Abhängigkeiten haben (alternativ kann auch Register Renaming in Hardware implementiert werden).

Beispiel:

Berechnung von R7=R1+R2+R3+R4+R5*R6

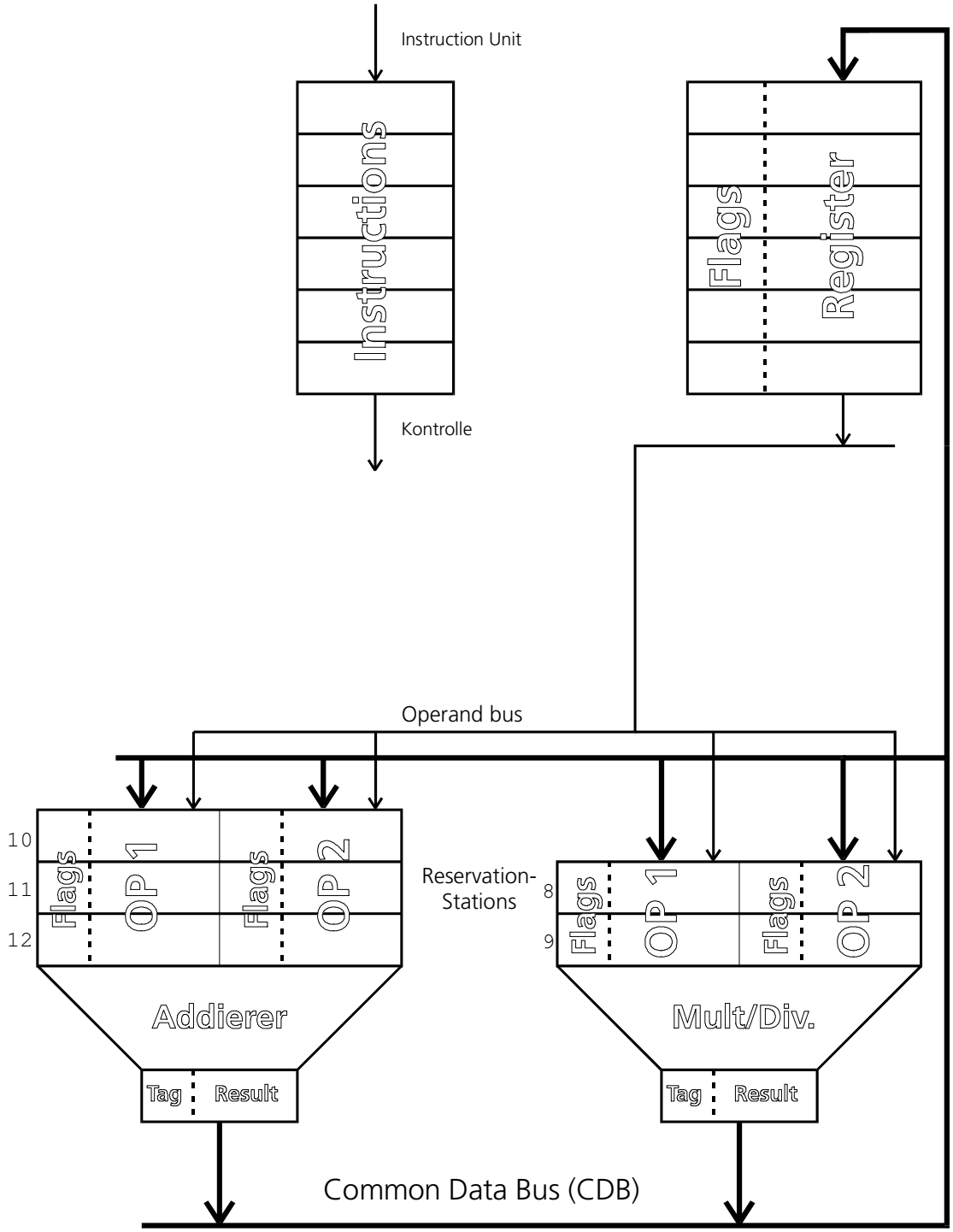
ohne Umsortierung	mit Umsortierung
`R7=R1+R2`	`R8=R5*R6`
`R7=R7+R3`	`R9=R1+R2`
`R7=R7+R4`	`R10=R3+R4`
`R8=R5*R6`	`R9=R9+R10`
`R7=R7+R8`	`R7=R8+R9`

Obwohl mit Umsortierung mehr Register benötigt werden, kann dieser Code auf einem Rechner mit Busy-Bits dennoch deutlich schneller ausgeführt werden als ohne Umsortierung, da die Multiplikation parallel zu den Additionen durchgeführt werden kann. Parallele Ausführung kann also durch „Busy-Bits“ auf einfache Weise

implementiert werden. Um einen Geschwindigkeitsvorteil zu erreichen ist dabei die aber Mithilfe des Programmieres/Compilers oder zusätzliche Hardware nötig.

Erweiterungen der skalaren FPU nach Tomasulo

Zur Implementierung des Algorithmus von Tomasulo müssen folgende Datenpfade und Strukturen hinzugefügt werden (siehe auch die folgende Abbildung):



- Von den Registern zu den Function Units wird der Operand Bus gelegt. Die Function Units können so direkt mit Registerinhalten versorgt werden.

- Der CDB ersetzt den Result Bus der skalaren CPU. Zusätzlich zu den eigentlichen Daten wird auf dem CDB parallel dazu das Tag transportiert. Bei Bedarf können mehrere CDBs verwendet werden, um mehrere Datenformate (Integer/Float) zu transportieren oder um die Leistung zu steigern.

- Die Register erhalten jeweils Speicherzellen für ein Tag und je eine Speicherzelle für ein „Valid Flag“. Das Valid-Flag wird immer dann auf 0 gesetzt, wenn für das Register ein Tag abgelegt ist. Es hat den Wert 1, wenn im Register gültige Daten abgelegt sind.

- Jeder Function Unit wird eine **Reservation Station** mit einem oder mehreren Einträgen zugeordnet. Pro Reservation Station kann jeweils ein Kommando von der Pipeline parallel bearbeitet werden. Um die Ergebnisse der Berechnungen wieder in Registern ablegen zu können, erhält jede Reservation Station Schreibzugriff auf den CDB. Jeder Reservation Station wird ein Tag zugeordnet (in der Abbildung die Tags 8-12).

Jeder Eintrag in der Reservation Station kann genau eine Instruktion aufnehmen. Die Reservation Station erhält daher noch zusätzlich folgende Datenstrukturen:

- Für beide Operanden Speicherplatz für je ein Tag, das Valid-Flag und die eigentlichen Daten.

- Ein Full- und ein Busy-Flag. Das Full-Flag wird gesetzt, wenn der Eintrag belegt ist. Das Busy-Flag wird gesetzt, wenn die dazugehörige Function Unit gerade mit der Berechnung des Ergebnisses beschäftigt ist.

- Speicherplatz für die Nummer des Zielregisters (Destination) und die Art der auszuführenden Operation (Addition, etc.).

Rahmenbedingungen

Gegeben ist ein Programm P mit

$P = \{ I_1, I_2, ..., I_p \}$

Die einzelnen Instruktionen I_i sind gegeben durch

I_i : D(i)=S1(i) op(i) S2(i)

Dabei ist D(i) das Zielregister (sink) und S1, S2 sind die Quellregister (source) der Instruktion. op(i) ist die ausgeführte Operation (*+-/)[3]. Die Werte dieser Register werden mit d(i), s1(i) und s2(i) bezeichnet.

Die Instruktion I_i wird durch die Function Unit F(i) bzw. die Reservation Station RS(i) ausgeführt.

Datenstrukturen

Für jede Reservation Station RS wird gespeichert:

- op (*+-/)
- free bzw. full (mit free:=/full)
- Dest (Zielregister der Operation)
- busy (Daten in der Pipeline)
- Vx, Datx, Tagx mit $x \in \{1, 2\}$ (Quelloperanden)

Die Valid Flags Vx werden gesetzt, sobald sich je ein Operand in Datx befindet. Sobald beide Flags V1,2 gesetzt sind, werden die Daten an die Pipeline übergeben und das Busy-Flag gesetzt. Adr wird benötigt, da im Register File wegen der großen Anzahl der Register kein echtes Bus-Snooping kostengünstig realisierbar ist[4]. Das Bus-Snooping im Register File kann ersetzt werden, da immer nur ein Register Daten vom CDB

---

[3] Der Algorithmus von Tomasulo ist für beliebige Operationen anwendbar; denkbar wäre z.B. eine Erweiterung um eine Funktion Unit für trigonometrische Funktionen.

[4] Im Original-Design von Tomasulo ist Bus-Snooping auch im Register-File vorgesehen; Adr wird daher dort nicht verwendet.

übernimmt. Bei einer kleinen Anzahl Register ist im Register File Bus-Snooping implementierbar; dann kann auf Adr verzichtet werden.

Für jedes Register R wird gespeichert:

- Tag
- Dat
- V (gesetzt, wenn gültige Daten in Dat)

Bezeichnungen

cdb(i) Zeitpunkt, zu dem das Ergebnis d(i) der Instruktion i auf dem CDB erscheint.

issue(i) Zeitpunkt, zu dem sich die Instruktion i in der Phase issue befindet.

dispatch(i) Zeitpunkt, zu dem sich die Instruktion i in der Phase dispatch befindet.

Protokoll des Tomasulo Algorithmus

Instruction Issue

```
while (I_i not issued) ∧ (Ii-1 issued)
 { if ( ∃ r mit RS_r ∈ RS_op(i) und RS.full = 0)          ①
     { RS(i):=RS_r ;          RS(i).Adr:= D(i);            ②
       RS(i).full:=1;         RS(i).busy:=0;               ③
       RS(i).op:=op(i);       RS(i).Vx=Sx(i).V;            ④
       RS(i).tag:=Sx(i).tag;                               ⑤
       D(i).tag:=r ;   D(i).V:=0;                          ⑥
       if ( Sx(i).V = 1 ) RS(i).Datx:=Sx(i).Dat            ⑦      (Operand Bus!)
     }
 }
```

Das Instruction Issue erfolgt in der ursprünglichen Programmreihenfolge. Beim Instruction Issue von Instruktion I_i wird wie folgt vorgegangen: Sobald ein Reservation Station Eintrag einer geeigneten Function Unit frei ist (①), wird dieser Eintrag initialisiert (②-⑥); dabei werden Quell- und Zielregister und die auszuführende Operation gesetzt (④). Wenn die Quelloperanden im Register File stehen (Valid Flag!) werden die Daten direkt über den Operand Bus zu dem Registration Stations übertragen (⑦). Ansonsten wird das Tag übernommen (⑤).

Instruction Dispatch (Execution)

```
while ( RS.full ∧ /RS.busy )                    ①
 { if ( RS.V1 ∧ RS.V2 )                         ②
    {  FU F(i) wird angefordert                 ③
       wenn F(i) frei:                          ④
        {  RS.busy:=1;                          ⑤
           RS.Datx, RS.op, RS.tag an F(i) übergeben
        }
    }
 }
```

Die eigentliche Berechnung wird wie folgt gestartet: Wenn ein Reservation Station Eintrag belegt ist (full) und die Daten noch nicht an die Pipeline übergeben wurden (/busy) (①), wird überprüft ob schon beide Operanden verfügbar sind (V1 ∧ V2) (②). Wenn ja, wird gewartet, bis die Pipeline ein Operandenpaar aufnehmen kann (③, ④). Dann wird das Busy-Flag gesetzt; die Operanden, die Operation und das Tag werden an die Pipeline übergeben. Das Tag muß an die Pipeline übergeben werden, damit das Ergebnis der Pipeline einem Register zugeordnet werden kann[5]. Da die Operanden von mehreren Reservation Station Einträgen gleichzeitig verfügbar werden können, muß ein geeignetes Protokoll und damit eine Priorität festgelegt werden (z.B. können die Daten des jeweils ältesten Eintrags zuerst an die Pipeline übergeben werden).

[5] Wenn - wie im Origial-Tomasulo-Design - mehrere sequentielle Funtion Units benutzt werden, kann das Tag hardcodiert werden.

Write Back (unit F)

```
while ( F hat Ergebnis für RS_r )
 { if ( CDB für F freigegeben )                    ①
     { CDB.data:=result;    CDB.tag:= r;           ②
       RSr.busy:=0;         RSr.full:=0;           ③
       R sei das Register RS_r.Adr                 ④
       if ( R.tag=r )                              ⑤
        {
          R.tag:=0; R.V:=1;                        ⑥
          R.Dat:=CDB.data;     (d.h. result)       ⑦
        }
     }
 }
```

Das Write Back wird wie folgt abgewickelt: Sobald die Function Unit Ergebnis berechnet hat, wird der CDB bei der Kontrolle angefordert (①). Nach Zuteilung des CDB wird das Ergebnis und das Tag der Reservation Station auf den CDB geschrieben (②). Die Reservation Station wird dann zurückgesetzt (③) (→Sender).

Zusätzlich wird überprüft, ob das in RS_r.Adr gespeicherte Register immer noch das Tag r hat (⑤). Wenn dies der Fall ist, wird das Tag des Registers gelöscht und die Daten auf dem CDB (also das Ergebnis der Operation) in das Register übertragen. Da sich dann gültige Daten im Register befinden, wird das Valid Flag gesetzt (⑥,⑦) (→Empfänger).

Snooping on CDB

```
while ( RS.full ∧ /RS.Vx ∧ (CDB.tag ≠ 0))
 { if (RS.tagx = CDB.tag)
    { RS.Datx:=CDB.Dat;
      RS.Vx:=1;
      RS.tagx:=0;
    }
 }
```

Reservation Stations, die noch nicht alle Operanden haben, „lauschen“ auf dem CDB (Bus-Snooping). Bei jedem Wert, der auf dem CDB erscheint, wird das Tag des CDB mit denen in der Reservation Station gespeicherten Tags (je eins für jeden Operand) verglichen. Wenn ein Tag übereinstimmt werden die Daten in die Reservation Station übernommen, das Tag gelöscht und das Valid Flag gesetzt (→Empfänger).

Beispiel

Das folgende Beispiel soll die Benutzung des CDB und der Reservation Stations verdeutlichen:

Auszuführen ist die Instruktion

```
R1 = R2 + R3
```

Dabei seien die Register R2 mit 9 und R3 mit 11 vorbelegt. Außerdem führt gerade eine andere Function Unit eine Instruktion aus, die das Register R3 als Zielregister hat. Das Ergebnis habe Tag 8. Das Register R3 enthält daher zur Zeit t=0 noch keine gültigen Daten. Die Instruktion wird wie folgt ausgeführt (t ist der Zyklus, S_1, S_2 sind die Quelloperandenspeicher der Reservation Station mit dem Tag 10, geänderte Werte sind grau unterlegt):

t	CDB		Reservation Station 10, Addierer									Register File								
						Operand 1			Operand 2			R1			R2			R3		
	Data	Tag	Busy	Op	Dest	Data	Tag	Valid	Data	Tag	Valid	Data	Tag	Valid	Data	Tag	Valid	Data	Tag	Valid
0	-	-	0	+	R1	9	0	1	-	8	0	-	10	0	9	0	1	-	8	0
1	11	8	0	+	R1	9	0	1	11	0	1	-	10	0	9	0	1	11	0	1
2	-	-	1	+	R1	9	0	1	11	0	1	-	10	0	9	0	1	11	0	1
3	-	-	1	+	R1	9	0	1	11	0	1	-	10	0	9	0	1	11	0	1
4	-	-	1	+	R1	9	0	1	11	0	1	-	10	0	9	0	1	11	0	1
5	-	-	1	+	R1	9	0	1	11	0	1	-	10	0	9	0	1	11	0	1
6	20	10	0	+	R1	9	0	1	11	0	1	20	0	1	9	0	1	11	0	1

Zur Zeit t=0 wird die Reservation Station 10 mit den Daten der Instruktion initialisiert. Der erste Operand R2 aus dem Register wird über den Operand Bus in die Reservation Station geladen. Gleichzeitig wird das Tag von Register R1 auf 10 gesetzt und das Valid-Flag gelöscht. Das Tag- und Valid-Flag von Register R3 werden in die Reservation Station übernommen.

Zur Zeit t=1 erscheit das Ergebnis der vorhergehenden Instruktion auf dem CDB. So wird der zweite Operand in die Reservation Station 10 geladen (Bus Snooping). Dann kann die eigentliche Addition beginnen, die im Beispiel vier Takte dauert (die Takte 2 bis 5). Kurz vor Ende der Addition fordert die Reservation Station bei der Kontrolle den CDB an. Im Beispiel wird der Bus direkt zugeteilt. Zusammen mit dem Ergebnis wird das Tag 10 mit auf den CDB geschrieben. Der Wert wird im Register R1 abgespeichert und das Valid-Flag gesetzt und das Tag des Registers gelöscht.

Zusammenfassung

- Ergebnisse von Rechenoperationen werden über den CDB transportiert
- Daten, die direkt aus Registern stammen, werden über den Operand Bus transportiert

6. Korrektheit

Invarianten des Tomasulo Algorithmus zum Zeitpunkt t

Folgende Invarianten lassen sich direkt aus dem Protokoll ablesen [SM96]:

Register R:

$$R.V = 0 \Leftrightarrow \exists\, r \neq 0 \text{ mit } R.tag = r$$

$$\wedge\ \exists\, i : issue(i) < t \leq cdb(i) \wedge D(i) = R \wedge RS(i) = RSr$$

$$\wedge\ \forall\, k>i : (\ D(i) \neq R \vee issue(k) \geq t\ )$$

$$R.V=1 \Leftrightarrow R.tag = 0 \wedge \exists\, j : cdb(j) < t \wedge D(j) = R$$

$$\wedge\ R.dat = d'(j)\ (\ = cdb(j).data\ )$$

$$\wedge\ \forall\, k>j : (\ D(i) \neq R \vee issue(k) \geq t\ )$$

Reservation Station RS:

$$RS.full = 1 \Leftrightarrow \exists\, i : RS(i) = RS \wedge issue(i) < t \leq cdb(i) \wedge RS.Dest = D(i)$$

$$RS.busy =1 \Leftrightarrow \exists\, i : RS(i) = RS \wedge dispatch(i) < t \leq cdb(i) \wedge RS.Dest = D(i)$$

Sei RS die Reservationstation zu Is und RS.full = 1:

$$RS.Vx = 0 \Leftrightarrow R.tagx = r \neq 0$$

$$\wedge\ \exists\, i < s : issue(i) < t \leq cdb(i) \wedge D(i) = Sx(s) \wedge RS(i) = RS$$

$$\wedge\ \forall\, i<k<s : D(i) \neq D(k)$$

$$RS.Vx = 1 \Leftrightarrow R.tagx = 0$$

$$\wedge\ \exists\, i < s : cdb(i) < t \wedge D(i) = Sx(s) \wedge RS.datx = d'(i)$$

$$\wedge\ \forall\, i<k<s : D(i) \neq D(k)$$

Korrektheitsbeweis

Zum Beweis der Korrektheit des Algorithmus sind zwei Schritte erforderlich [SM96]:

1. Beweis der Datenkonsistenz, d.h., daß der Algorithmus keine Auswirkungen auf die Rechenergebnisse hat. Trotz Scheduling muß also jede Instruktion die gleichen Operanden lesen und das gleiche Ergebnis produzieren wie bei squentieller Abarbeitung. Zusätzlich muß die Schreibreihenfolge pro Speicherwert (Memory, Register) gleich bleiben.

2. Beweis der Terminierung, also daß keine Deadlocks entstehen.

Datenkonsistenz

Um zu zeigen, daß der Algorithmus keine Auswirkungen auf die Rechenergebnisse hat, genügt es zu zeigen, daß die Quell- und Zielregister bei allen Operationen bei Verwendung des Algorithmus von Tomasulo denen der sequentiellen Ausführung entsprechen. Unter der Annahme, daß die Function Units deterministisch arbeiten, werden dann auch die selben Ergebnisse errechnet.

Zur Erinnerung:

Ausgeführt werden Instruktionen I_i mit

I_i : D(i)=S1(i) op(i) S2(i)

Für den Beweis werden folgende Bezeichnungen verwendet:

Wert von	D(i)	Sx(i)
sequentiell	d(i)	sx(i)
Tomasulo	d’(i)	sx’(i)

$x \in \{1,2\}$

Zu zeigen ist also:

① $\forall$ i : d(i)=d’(i) (Zielregister)

② $\forall$ i : sx(i)=sx’(i) (Quellregister)

Der Beweis erfolgt über Induktion über die Zeit t (Anzahl der Zyklen):

IA: $t=0$ Ok (Reset, es haben noch keine Operationen stattgefunden)

IS: $t \rightarrow t+1$

IVor: Für alle Instruktionen, die bis zur Zeit t ausgeführt wurden, gilt ① und ②.

IBeh: Für alle Instruktionen, die bis zur Zeit $t+1$ ausgeführt wurden, gilt ① und ②.

Beweis:

zu ① (Writes):

1. Fall: Zum Zeitpunkt t wird das Ergebnis der Instruktion I_j auf den CDB geschrieben, d.h. $cdb(j)=t+1$

   $\Rightarrow$ Die Quelloperanden von Befehl I_j wurden zur Zeit t oder früher geladen

   $\Rightarrow$ (IVor $\Rightarrow$ $sx(i)=s'x(i)$, Korrektheit der Function Units) $d'(j)=d(j)$

2. Fall: sonst: Siehe IVor

zu ② (Reads):

I_i liest Operand $Sx(i)$. Sei $R:=Sx(i)$

1. Fall: Operand wird über Register Bus gelesen

   $\Leftrightarrow t+1 = issue(i) \wedge R.V=1$

   I_k sei die letzte Instruktion (d.h. $k<i$), die R geändert hat

   $\Rightarrow R.dat = d'(k) = d(k)$ (wg IVor) $\wedge$ $\forall h$: $k<h<i$ gilt: $D(h) \neq R$

2. Fall: Operand wird im Zyxlus t+1 über den CDB gelesen:

   $\Leftrightarrow t+1 > issue(i) \wedge RS(i).tagx=cdb(j).tag$

   $\Rightarrow cbd(j).dat = d'(j) = d(j)$ (wg ①) $\wedge$ $D(j)=Sx(i)$ (Algoritmus!) $\wedge$

$\forall h: j<h<i: D(h) \neq D(j)$ (sonst Widerspruch zu RS(i).tagx=cdb(j).tag)

$\Rightarrow s'x(i)=sx(i)$

Terminierung

1. Definition

T(i) sei die kleinste Zahl t für die gilt:

alle Instruktionen $I_1 \ldots I_{i-1}$ sind bis zum Zyklus t terminiert

2. Lemma 1: CDB-Schedule

Sei f = # units. Wenn zur Zeit t ein CDB-Request stattfindet, dann wird der CDB spätestens zur Zeit t+f-1 zugeteilt, da der CDB im Round-Robin Verfahren vergeben wird.

3. Lemma 2

$issue(i) \leq T(i) + 1$

Wenn alle Instruktionen $I_1 \ldots I_{i-1}$ terminiert sind, stehen im nächsten Zyklus auf jeden Fall genug Ressourcen (CDB, Function Units) zur Verfügung, um das Issue der Instruktion i durchzuführen.

4. Beweis der Terminierung

Mit dem Dispatch der Instruktion i kann frühestens nach dem Issue und spätestens nach der Terminierung der vorherigen Instruktionen begonnen werden, plus die Zeit, die benötigt wird, bis garantiert eine Function Unit frei ist (Emax, also die Zeit die eine Function Unit maximal rechnet, plus die Zeit, die benötigt wird um den CDB zu erhalten):

$$dispatch(i) \leq \max\{issue(i), T(i)\} + 1 + Emax + f \underset{\text{Lemma 2}}{=} T(i) + 2 + Emax + f$$

Das Ergebnis der Operation wird spätestens zum folgenden Zeitpunkt auf den CDB geschrieben:

$$cdb(i) \leq dispatch(i) + (Emax-1) \cdot f + 1 + f - 1$$

Dabei muß angenommen werden, daß die Pipeline vollständig gefüllt ist und daß jedes Ergebnis der Pipeline mit einer Wartezeit von f Zyklen auf den CDB geschrieben werden kann (Lemma 1). Anschließend muß ebenfalls wieder f Zyklen gewartet werden bis der CDB für das Ergebnis der Instruktion i frei ist.

Durch Einsetzen von $dispatch(i) \leq T(i) + 2 + Emax + f$ erhält man:

$$cdb(i) \leq T(i) + 2 + Emax + f + (Emax-1) \cdot f + 1 + f - 1$$

$$cdb(i) \leq T(i) + 2 + Emax + (Emax-1) \cdot f + 2\,f$$

$$cdb(i) \leq T(i) + 2 + Emax\,(f+1) + f$$

Zum Beweis:

Zu zeigen ist: $\exists\, \alpha$ mit $T(i+1) \leq T(i) + \alpha \leq i\,\alpha < \infty$

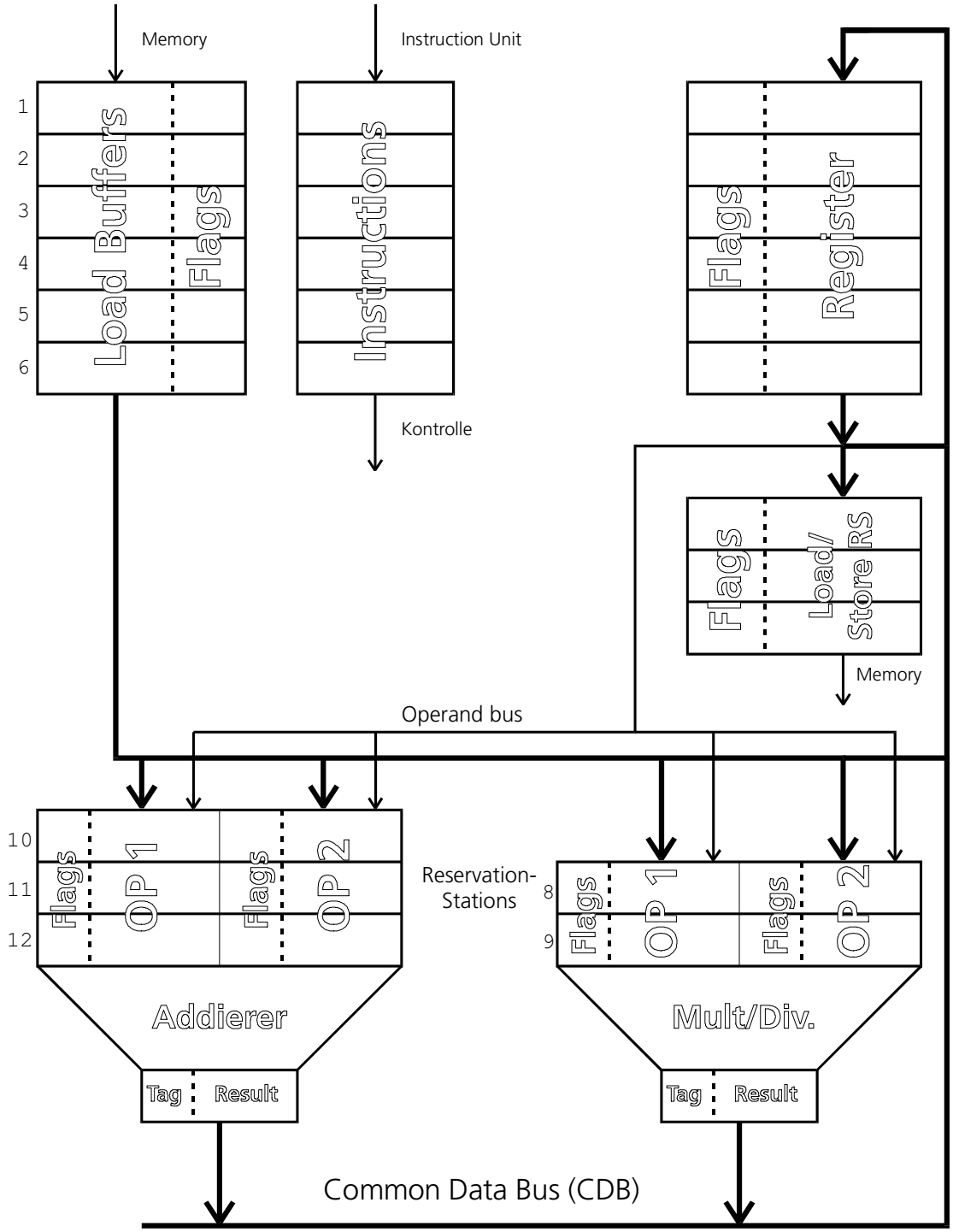
Wähle $\alpha = 2 + Emax\,(f+1) + f$ (konstant):

Offens. gilt: $T(i+1) = \max\{cdb(i), T(i)\} \leq \max\{T(i)+\alpha, T(i)\} = T(i)+\alpha \leq i \cdot \alpha$

(Beweis der letzten Ungleichung durch vollständige Induktion)

4. Erweiterung um Speichersystem

Um die CPU um ein Memory-Interface zu erweitern, müssen folgende Datenpfade und Strukturen hinzugefügt werden (siehe folgende Abbildung):



Die für die Kontrolle notwendigen Datenpfade sind nicht angegeben.

Load-Buffer

Die Load-Buffer erhalten Schreibzugriff auf den CDB. Zur Identifikation der Daten, die die Load-Buffers auf den CDB schreiben, wird jedem Eintrag ein Tag zugeordnet (in der Abbildung die Tags 1 bis 6).

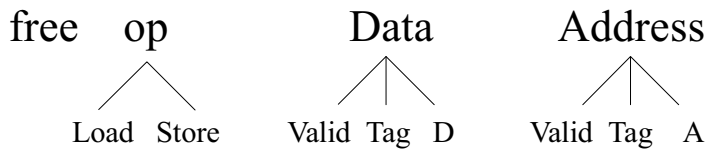
Analog zu den Reservation Stations werden für jeden Eintrag eines Load Buffers folgende Daten gespeichert:

- free bzw. full (mit free:=/full)
- Dest (Zielregister der Operation)
- Valid
- Data

Die Bedeutung der Flags entspricht der Bedeutung der entsprechenden Flags bei den Reservation Stations.

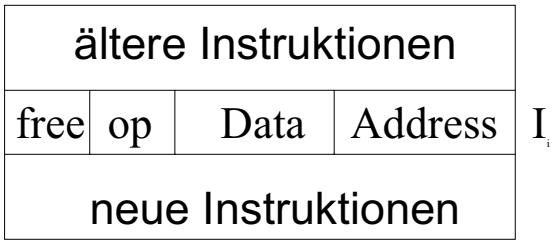
Load/Store-Reservation Station

Die Load/Store-Reservation Station wird mit Speicherplatz für folgende Datenstrukturen versehen:



Bei Loads wird Data.Tag auf das Tag des Load Buffers und - um Snooping auf dem CDB zu verhindern - Data.Valid auf 1 gesetzt.

Die Daten werden in der Reihenfolge der Instruktionen abgelegt:



Bedingung für einen Store-Dispatch

(1) $\text{Data.Valid} = \text{Address.Valid} = 1$

(2) $\forall j<i : I_j.\text{Address.Valid} = 1$ (Adressen vorheriger Einträge gültig)

(3) $\forall j<i : I_j.\text{Address.A} \neq I_i.\text{Address.A}$ (... und verschieden von der betrachteten)

Bedingungen für einen Load-Dispatch

(1) $\text{Data.Valid} = \text{Address.Valid} = 1$

(2) $\forall j<i : I_j.\text{Address.Valid} = 1 \vee I_j.\text{op} = \text{Load}$

(3) $\forall j<i : I_j.\text{Address.A} \neq I_i.\text{Address.A} \vee I_j.\text{op} = \text{Load}$

Zu (2) und (3): Bei Loads muß die Reihenfolge nicht erhalten bleiben.

Optimierung bei Load: Forwarding

(4) $(1\text{-}2) \wedge \exists\, j<i: (I_j.\text{Address.A} = I_i.\text{Address.A} \wedge I_j.\text{Address.Valid} = 1 \wedge$

$I_j.\text{Data.Valid} = 1 \wedge \forall k: j<k<i : I_j.\text{Address.A} \neq I_k.\text{Address.A})$

$\Rightarrow I_j.\text{Data.D}$ an Load Buffer weiterreichen

Diese Bedingung ist eine Optimierung und verhindert Loads von Daten, die noch in einem Store Buffer stehen.

Scheduling von Loads und Stores

Die oben genannten Bedingungen können für mehrere Instruktionen erfüllt sein. Die Kontrolle muß dann eine auswählen und könnte dann z.B. nach folgenden Kriterien vorgehen:

- In Programmreihenfolge
- Priorisierung von Loads

Literatur

[SM96] Silvia M. Müller, Vorlesung RA-II, WS 96/97

[TO67] R.M. Tomasulo, „An Efficient Algorithm for Exploiting Multiple Arithmetic Units,“ IBM Journal, January 1967

[HP90] J.L. Hennessy and D.A. Patterson, „Computer Architecture: A Quantitative Approach,“ Morgan Kaufmann Publishers, INC., San Mateo, CA, 1990.

[SP88] J.E. Smith, A.R. Pleszkun, „Implementing Precise Interrupts in Pipelined Processors,“ IEEE Trans. Comp. Vol 37(5): 562-573, 1988