1 Einleitung

Hennessy und Patterson spezifizieren in ihrem Buch „Computer Architecture - A Quantitative Approach“ [HePa90] einen 32-bit RISC Prozessor mit einer einfachen „load/store“-Architektur. Durch seinen simplen Befehlssatz und einer leicht verständlichen Architektur ist dieser Prozessor bestens zu Lehrzwecken geeignet.

Das bestehende Modell wurde von Martin Gumm am Institut für parallele und verteilte Höchstleistungsrechner (Abteilung Integrierter Systementwurf) erstellt. Abweichend von der Spezifikation nach Hennessy und Patterson wurde der Prozessor mit zusätzlicher Interrupt- und Ausnahmebearbeitung sowie drei verschiedenen Operationsmodi ausgestattet, jedoch ohne Fließkommaeinheit, wodurch das Modell den Namen DLXS („S“ für „Stuttgart“) erhielt.

Ziel der Arbeit war die Erweiterung des DLXS-Prozessormodells um eine 5-stufige Befehlspipeline.

2 Der DLXS-Prozessor

Zunächst soll das bisher bestehende Modell kurz vorgestellt werden. Eine ausführlichere Beschreibung findet sich in „VHDL-Modelling And Synthesis Of The DLXS RISC Processor“ [MaGu95].

Als Schnittstellen besitzt der DLXS einen Adress- sowie einen bidirektionalen Datenbus mit einer Breite von 32 Bit wie in Abbildung 2-1 dargestellt. Die Signale `RW`, `ENABLE` und `READY` werden als Speicherkontrolleitungen bezeichnet und dienen der Kommunikation des Prozessors mit dem Speicher. `phi1` und `phi2` müssen einen nicht-überschneidenden Zweiphasentakt zur Verfügung stellen und `ERROR` wird aktiviert, falls der Prozessor in einem irreparablen Fehlerzustand geraten ist.

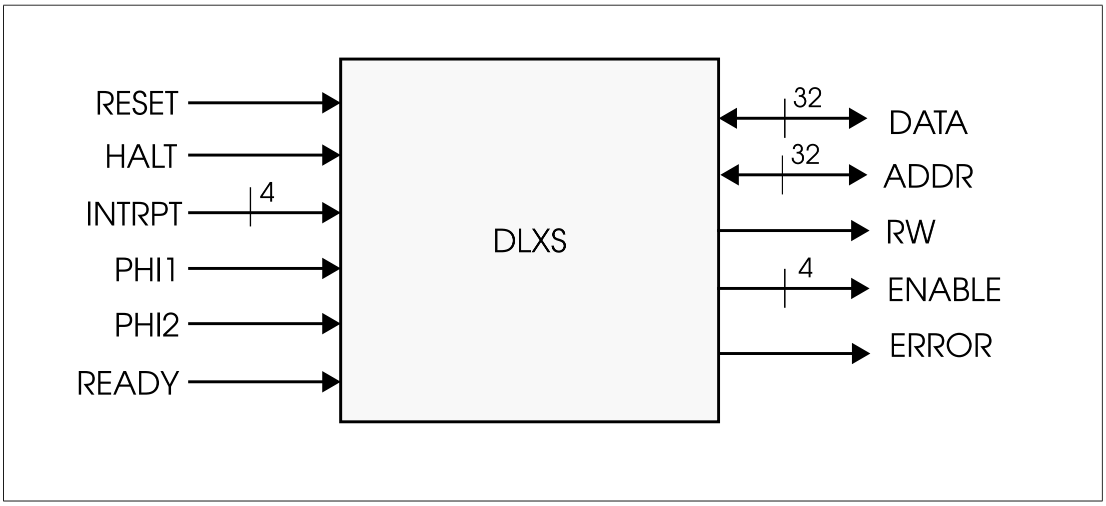

Abbildung 2-1: Schnittstellen des DLXS

2.1 Struktur

Der DLXS besitzt eine einfache interne Struktur , bestehend aus einem Datenpfad und einer Steuerung (Abbildung 2-2).

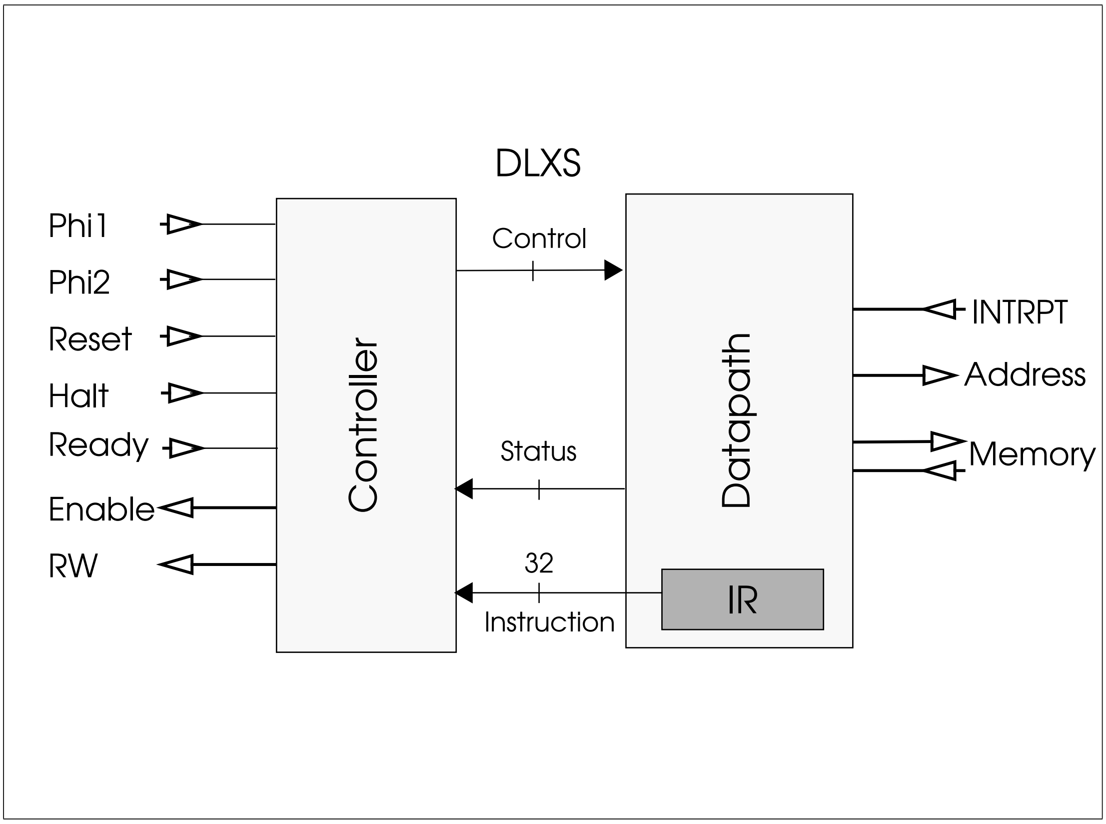

Abbildung 2-2: Die interne Struktur des DLXS

Der Datenpfad liefert der Steuerung das aus dem Speicher geholte 32 Bit breite Befehlswort (`Instruction`), sowie die zur Bearbeitung des Befehls notwendigen Statussignale (`Status`). Die Steuerung generiert daraus Kontrollsignale (`Control`) um die Komponenten im Datenpfad anzusteuern. Mit den Speicherkontrolleitungen `READY`, `ENABLE` und `RW` kann die Steuerung weitere Befehlsworte anfordern.

2.1.1 Der Datenpfad

Der Datenpfad ist in Abbildung 2-3 dargestellt. Er enthält die Register, eine arithmetisch/logische Verarbeitungseinheit (`ALU`) sowie drei Busse um alle Komponenten zu verbinden. Die Komponente `Register File` besteht aus 32 „General Purpose"-Registern.

Zur Bearbeitung von Interrupts und Ausnahmen (Exceptions) dienen die speziellen Register `ICR`, `IAR` und `TBR`. Das Trap-Base-Register `TBR` enthält die Basisadresse der Softwareroutinen des Betriebssystems, welches auf alle Unterbrechungen reagieren muß. Im ICR-Register werden Statusbits zur Identifizierung des jeweiligen Interrupts gespeichert. Das IAR enthält die Rücksprungadresse aus einer Unterbrechungsroutine.

Das Speicheradressregister `MAR` beinhaltet bei Speicherzugriffen die Zieladresse, während die zu lesenden oder schreibenden Daten in das Speicherdatenregister gelangen.

`PC` ist der Befehlszähler. Dieses Register enthält die Adresse des nächsten Befehls. Sie wird zu Beginn einer Befehlsausführung dem Speicher übergeben, damit dieser das Befehlswort zur Verfügung stellt. Das Befehlswort wird zur anschließenden Verarbeitung in der Steuerung in das Befehlsregister geladen.

Mit der Komponente `Constants` können konstante Werte erzeugt werden, die zur Initialisierung von einigen Registern und bei Schiebeoperationen benötigt werden.

2.1.2 Die Steuerung

Hauptsächlich besteht die Steuerung aus einem endlichen Automaten (`FSM`), drei Befehlsdekodierern (`DEC1`, `DEC2` und `DEC3`) und zusätzlicher Logik (Abbildung 2-4).

Ein zu bearbeitender Befehl wird zunächst durch die Befehlsdekodierer entschlüsselt, welche die Eingabe für den endlichen Automaten erstellen.

`DEC1` dekodiert alle Instruktionen, bis auf die Speicherzugriffe, welche von `DEC2` identifiziert werden. `DEC3` unterstützt die Dekodierung der Quell- und Zielregister des Befehls.

Mit steigender Taktflanke des `phi1`-Signals beginnt der Automat zu arbeiten, indem er abhängig von seinem Zustand, sowie dem entschlüsselten Befehl, Kontrollsignale zur Bearbeitung des Befehls generiert. Diese sind mit dem Datenpfad verbunden.

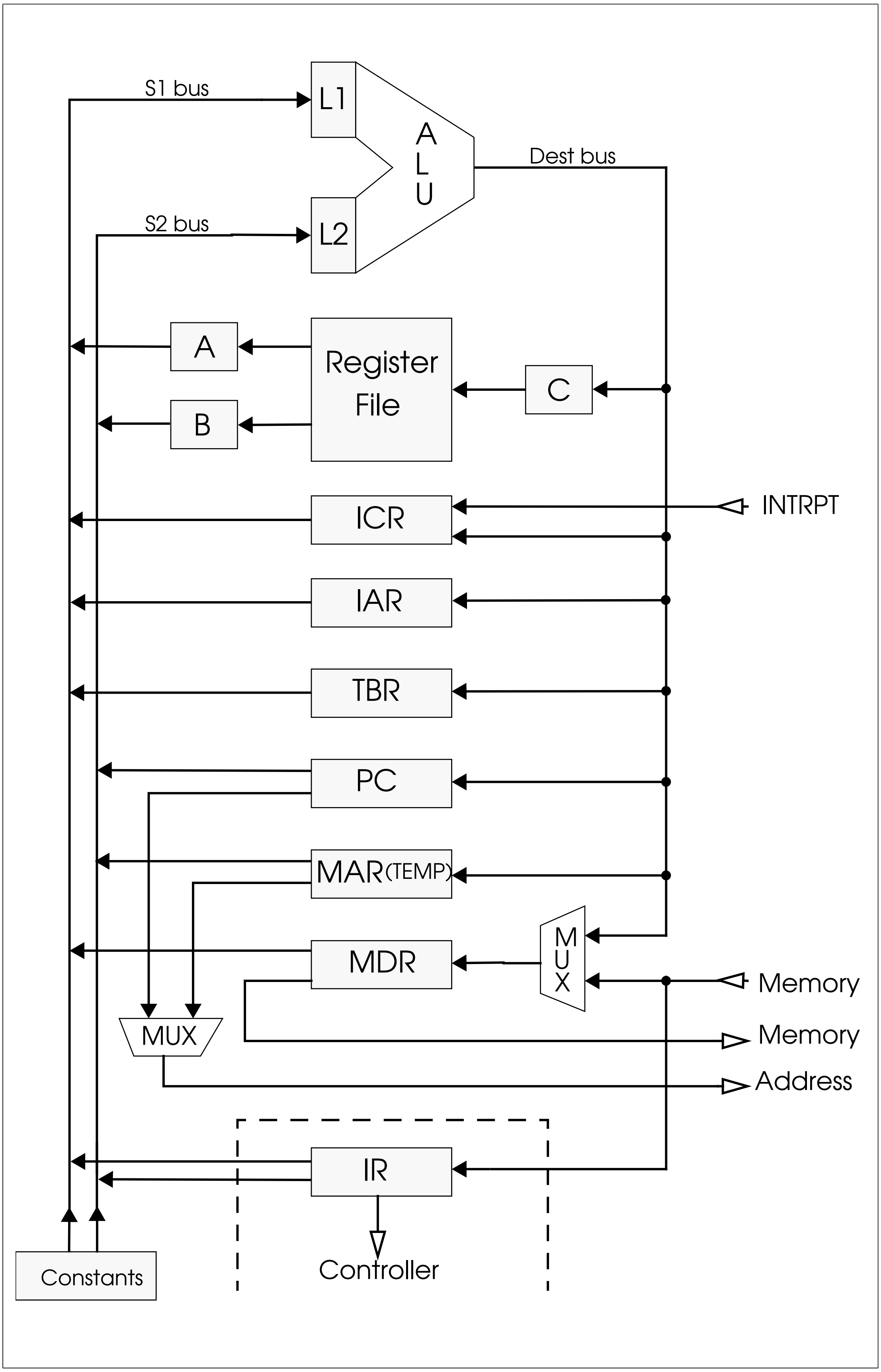

Abbildung 2-3: Der Datenpfad des DLXS

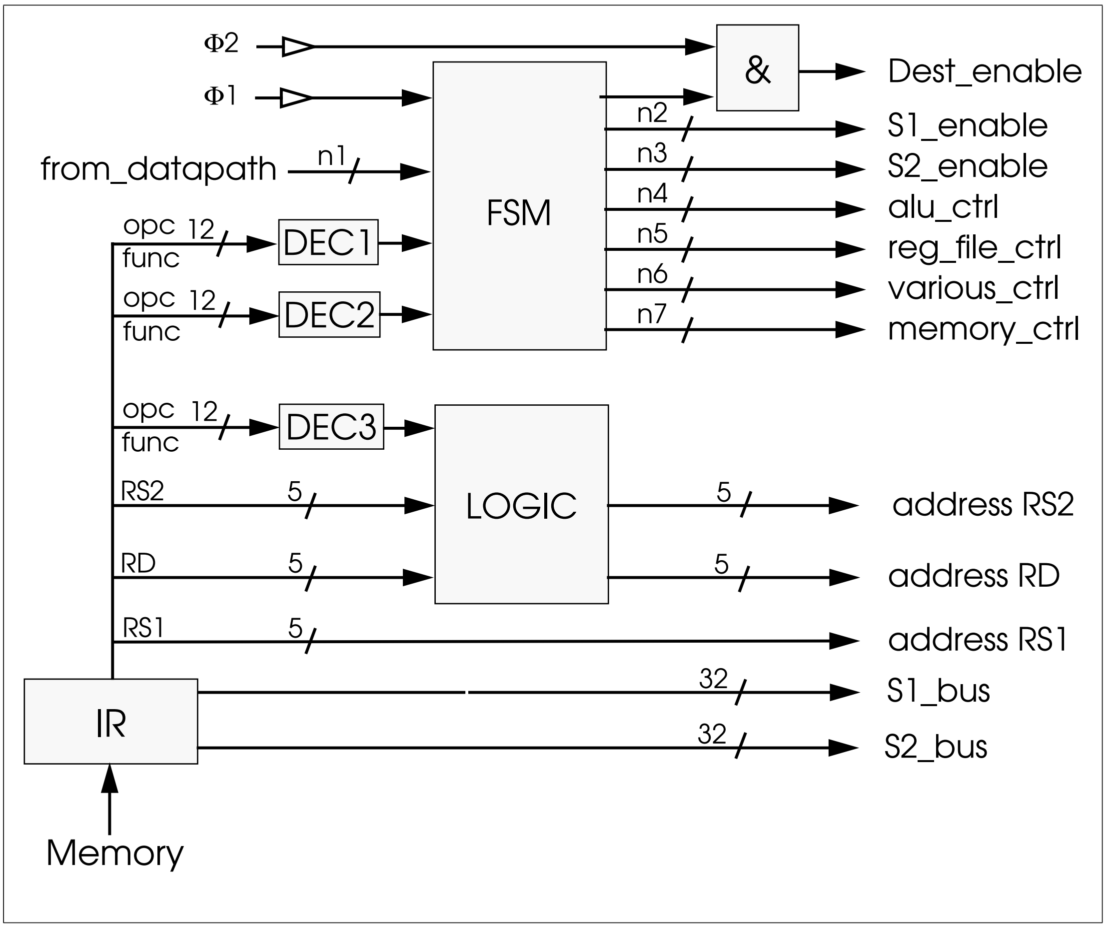

Abbildung 2-4: Die Steuerung des DLXS

Die Bearbeitung eines Befehls erfordert mehrere Arbeitsschritte, wobei jeder Schritt einem Zustand des Automaten entspricht.

2.2 Die Befehlsausführung

Die Abarbeitung eines Befehls des DLXS kann in fünf Arbeitsschritte aufgeteilt werden:

- Instruction Fetch
- Instruction Decode
- Execution
- Memory Access
- Write Back

Jeder dieser Schritte benötigt mindestens einen T aktzyklus. Der endliche Automat in der Steuerung sorgt mit den Kontrollsignalen dafür , daß der Datenpfad die Arbeitsschritte ausführt. Jeder Arbeitsschritt entspricht einem oder mehreren Zuständen des Automaten. Im folgenden werden diese Schritte näher erklärt. Die Aktionen jedes Arbeitsschritts sind in der

Register -Transfer -Notation nach Hennessy und Patterson [HePa90] dargestellt.

2.2.1 Instruction Fetch (IF)

Im ersten Schritt zur Bearbeitung eines Befehls, muß dieser zunächst aus dem Speicher geholt werden (engl. „fetch“). Dabei wird dem Speicher die Adresse des aktuellen Befehls, welche sich im Befehlszähler befindet, übergeben.

```
IR <- Memory[PC]
```

Die FSM des DLXS befindet sich dabei im Zustand `fetch`.

2.2.2 Instruction Decode (ID)

Anschließend muß der Befehl dekodiert werden. In diesem Schritt wird festgestellt, um welche Art von Befehl es sich handelt und welche Register er anspricht. Er entspricht dem Zustand `decode_pcincr_ab` der FSM. Außerdem wird der Befehlszähler um vier erhöht, um auf den nächsten Befehl zu zeigen.

```
A <- Rs1;
B <- Rs2;
PC <- PC + 4;
```

Die Register A und B werden mit dem Inhalt der Register Rs1 und Rs2 aus dem Registerfile belegt.

2.2.3 Execution Stage (EXE)

In dieser Bearbeitungsphase wird die ALU eingesetzt. Die Befehle lassen sich grob in drei Klassen einteilen:

- Speicherzugriffe
- logische und arithmetische Befehle
- Sprünge

Im Falle eines Speicherzugriffs muß die Zieladresse der zu schreibenden oder lesenden Daten ermittelt werden. Diese berechnet sich durch Addition eines Registers mit einer im Befehl enthaltenen Konstanten oder ist ganz einfach der Inhalt eines Registers. Die errechnete Zieladresse muß in das Memory Address Register (MAR), während im Memory Data Register (MDR), im Falle eines Schreibvorgangs, die zu schreibenden Daten abgelegt werden.

```
MAR <- A + immed;
MAR <- A;
MDR <- RD;
```

Bei logischen und arithmetischen Befehlen führt die ALU eine Operation über zwei Registerinhalte oder über ein Register und eine Konstante aus. Das Ergebnis kommt in späteren Schritten über Register C wieder zurück in das Registerfile.

```
C <- A op B;
C <- A op immed;
```

Bei einem Sprung wird mit Hilfe der ALU das Sprungziel berechnet. Dazu addiert die ALU die im Sprungbefehl enthaltene Konstante mit dem Inhalt des Befehlszählers, im Falle eines Softwareinterrupts, mit dem Inhalt des TRAP-Basisregisters. Das Ergebnis wird in den Befehlszähler geschrieben.

```
PC <- PC + immed;
PC <- PC + TBR;
```

Für jede Befehlsklasse erzeugt `DEC1` einen eigenen Folgezustand von `decode_pcincr_ab`, der die Steuersignale zur V errichtung der oben angesprochenen Aktionen erzeugt.

2.2.4 Memory Access (MEM)

In der vorletzten Phase kommuniziert der Prozessor mit dem Speicher . Dies geschieht jedoch nur dann, wenn ein Lade-/Speicherbefehl vorliegt. Ansonsten wird dieser Arbeitsschritt übergangen. Da der DLXS verschiedene Arten des Speicherzugriffs kennt, repräsentieren mehrere verschiedene Zustände diesen Arbeitsschritt. DEC2 ermittelt welchen Zustand der Automat hier einnimmt.

In der vorhergehenden Execution Stage wurde die Speicheradresse errechnet und im MAR abgelegt. Nun werden entweder die Daten aus dem MDR in den Speicher geschrieben, oder aus dem Speicher in das MDR gehohlt.

```
MDR <- M[MAR]; C <- MDR; -- Lesezugriff
M[MAR] <- MDR; -- Schreibzugriff
```

2.2.5 Write Back (WB)

Das Ergebnis des Speicherzugriffs, oder aber der arithmetisch/logischen Operation, wird in der letzten Phase in das Registerfile zurückgeschrieben.

```
RD <- C
```

Dabei befindet sich die FSM im Zustand `write_back`, dem anschließend wieder Zustand `fetch` folgt.

3 Pipelining

Die Grundidee beim Pipelining ist es, die Befehle nicht einzeln hintereinander, sondern wie an einem Fließband abzuarbeiten. Wenn ein Befehl dekodiert wird, könnte zum Beispiel schon der nächste aus dem Speicher geholt werden, während die ALU gleichzeitig die Operation eines Vorgängerbefehls ausführt. Betrachtet man die fünf Arbeitsschritte kann die Abarbeitung hintereinanderliegender Befehle nach folgendem Schema ablaufen (Tabelle 1):

Tabelle 1: Bearbeitung der Befehle in einer Pipeline

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`add R1, R2, R3`	IF	ID	EXE	MEM	WB	
`seq.i R4, R5, X"0007"`		IF	ID	EXE	MEM	WB
`bnez loop`			IF	ID	EXE	MEM
`add R1, R3, R4`				IF	ID	EXE
`sub R7, R5, R3`					IF	ID

Jeder Befehl benötigt zwar weiterhin fünf Arbeitsschritte, aber in einem Taktzyklus bearbeitet der Prozessor fünf verschiedene Befehle parallel. Effektiv wird so in jedem Takt ein Befehl abgearbeitet. Somit ist der Prozessor um ein fünfaches schneller !

Für den Datenpfad bedeutet dies, daß die Daten von fünf verschiedenen Befehlen gleichzeitig, in verschiedenen Komponenten, bearbeitet werden müssen. Die Steuerung muß also jeden Takt Steuersignale für alle fünf Arbeitsschritte senden.

3.1 Eine Pipeline für den DLXS

Damit der DLXS seine Befehle in einer Pipeline bearbeitet, muß jeden Takt ein neuer Befehl geladen werden. Die fünf Arbeitsschritte, die ein Befehl dann durchläuft, werden als Pipelinephasen bezeichnet. Im folgenden werden diese Phasen abgekürzt als IF, ID, EXE, MEM und WB. Wenn sich ein Befehl in einer bestimmten Pipelinephase befindet, heißt das, daß der entsprechende Arbeitsschritt ausgeführt wird. Das neue Modell des DLXS wurde DLXS/P genannt („P“ für „Pipeline“).

3.2 Strukturänderungen

Bei der Abarbeitung eines Befehls durch den DLXS werden manche Komponenten in mehreren Arbeitsschritten benutzt. Dies führt bei einer Pipelinerealisierung zu Problemen, da sich dann mehrere Befehle gleichzeitig in verschiedenen Bearbeitungsphasen befinden. Gelangt zum Beispiel ein `ADD`-Befehl in seine EXE-Phase wird die ALU zur Addition der beiden Operanden benötigt. Im gleichen Takt muß jedoch zum „Instruction Fetch“ der Befehlszähler von der ALU erhöht werden.

Daher muß der Datenpfad um neue Komponenten, hier zum Beispiel um einen zusätzlichen Inkrementierer, erweitert werden. Um diese fehlenden Komponenten zu bestimmen, werden alle DLXS-Befehle in Klassen eingeteilt, die sich bezüglich der Abarbeitung und der benutzten Ressourcen und Komponenten unterscheiden.

Dann wird jede Komponente des Datenpfads genau einer Arbeitsphase zugeteilt, in der sie arbeiten kann. Wenn eine Komponente in mehreren Arbeitsphasen benötigt wird, muß für jede dieser Phasen eine zusätzliche Instanz eingesetzt werden, um Kollisionen bei gleichzeitigen Zugriff mehrerer Befehle auf diese Komponente zu vermeiden.

Um eine günstige Zuteilung der Komponenten zu den Arbeitsschritten bzw. Pipelinephasen zu finden, muß der Befehlssatz des DLXS analysiert werden.

3.3 Befehlsklassen

Die verschiedenen Befehle des DLXS lassen sich in Befehlsklassen aufteilen, so daß Befehle einer Klasse die gleichen Bearbeitungsschritte und Komponenten benötigen. Zu jeder Klasse wird ein Bearbeitungsschema angegeben, welches die nötigen Aktionen des Datenpfads in jedem Arbeitsschritt angibt. Hier wird ersichtlich, welche Komponenten der Befehl benötigt, und vor allem zu welchem Arbeitsschritt diese benötigt werden.

Die Bearbeitungsschemen aller Klassen haben zwei Arbeitsschritte gemeinsam, IF und ID. Da zu diesen Zeitpunkten noch nicht bekannt ist, um welchen Befehl es sich handelt, kann es natürlich noch keine Varianten in der Bearbeitung geben.

Der Befehlszähler muß bereits in der IF-Phase erhöht werden, da sonst im nächsten Takt wieder derselbe Befehl geladen würde (Tabelle 2).

Tabelle 2: Allen Befehlen gemeinsame Arbeitsschritte

Phase	Aktionen
IF	IR <- M[PC]; PC <- PC + 4

Tabelle 2: Allen Befehlen gemeinsame Arbeitsschritte

Phase	Aktionen
ID	A <- Rs1; B <- Rs2

Dazu dient ein zusätzlicher einfacher Inkrementierer `add_4_adder.vhd`, da die ALU bereits in der EXE-Phase eingesetzt wird. Der Inkrementierer ist mit dem Ausgang des Befehlszählers verbunden und leitet sein Ergebnis über einen MUX-Baustein `pc_mux` wieder in den Befehlszähler zurück. Bei einem Sprung muß das `pc_mux` umgeschalten werden, damit das von der ALU berechnete Sprungziel übernommen werden kann (Abbildung 3-1).

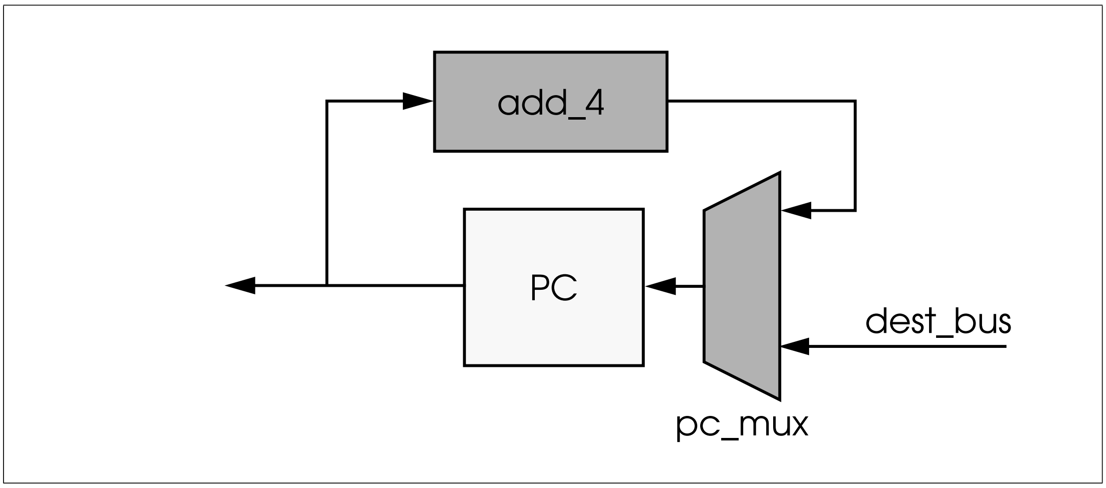

Abbildung 3-1: Der zusätzliche Inkrementierer

3.3.1 Load / Store

Ladebefehls benötigen stets alle fünf Arbeitsschritte. Deshalb kann das Registerfile immer erst in der fünften Pipelinephase beschrieben werden. In der MEM-Phase erhält das Datenregister (MDR) das Datenwort aus dem Speicher (T abelle 3).

Tabelle 3: Ladebefehle

Phase	Aktionen
EXE	MAR <- A + immed
MEM	MDR <- MEM[MAR]
WB	RD <- LMDR

Sieht man im Vergleich hierzu einen Speicherbefehl, erkennt man, daß das zu speichernde Datenwort bereits in der EXE-Phase im Datenregister bereitgestellt werden muß, damit es am Anfang der MEM-Phase zur Verfügung steht (Tabelle 4).

Tabelle 4: Speicherbefehle

Phase	Aktionen
EXE	MAR <- A + immed SMDR <- B
MEM	MEM[MAR] <- SMDR
WB	

Das MDR wurde daher aufgeteilt in ein Load-Memory-Data-Register (LMDR) und ein Store-Memory-Data-Register (SMDR).

Bei den Ladebefehlen kommt es zu drei weiteren Problemen:

Erstens gelangen geladene Daten nur über die ALU vom MDR ins Registerfile, jedoch ist `RD <- LMDR` nicht so ohne weiteres ausführbar, da die ALU in der EXE- und nicht in der WB-Phase arbeitet.

Zweitens gelangen die Daten nicht direkt in das Registerfile, sondern erst in das C-Register, wodurch die Ausführung von Ladebefehlen auf dem DLXS sechs Takte benötigt.

Drittens müssen bei Halbwort- und Bytezugriff durch eine logische Schiebeoperation die unbesetzten Bitstellen der Daten durch Vorzeichenbits besetzt werden.

Deshalb muß im Datenpfad ein zusätzlicher direkter Weg vom LMDR in das Registerfile bestehen. Ein zusätzlicher 2:1 MUX-Baustein `wb_mux` schaltet bei einem Ladebefehl auf das Ladedatenregister um (Abbildung 3-2).

Die Schiebeoperation wurde als zusätzliche Logik des MDR (`mdr2-dataflow.vhd`), implementiert.

Speicherbefehle werden beim DLXS mit zwei „Execution Stages“ abgehandelt. In der ersten Phase wird die Zieladresse für die zu speichernden Daten berechnet und im MAR abgelegt. In der zweiten Phase werden die Daten aus dem B-Register über die ALU ins MDR gebracht. Diese zweite Phase muß ebenfalls durch einen zusätzlichen Pfad ohne die ALU gelöst werden.

Das B-Register bekommt daher einen zweiten Ausgangsport, der mit dem SMDR direkt verbunden ist (Abbildung 3-3).

Eine weitere Kollision entsteht bei gleichzeitigem „Instruction Fetch“, während der Lade- oder Speicheroperation da der DLXS nur einen Daten- und eine Adressport besitzt. Auf dieses Problem wird in Kapitel 4

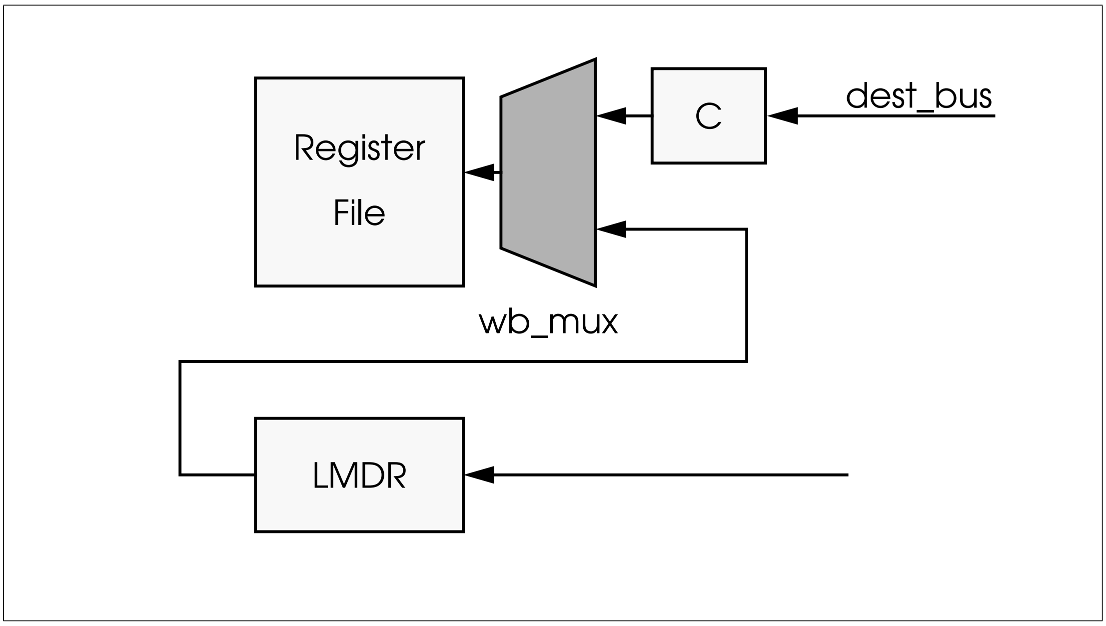

Abbildung 3-2: Zusätzliche Pfade ins Registerfile

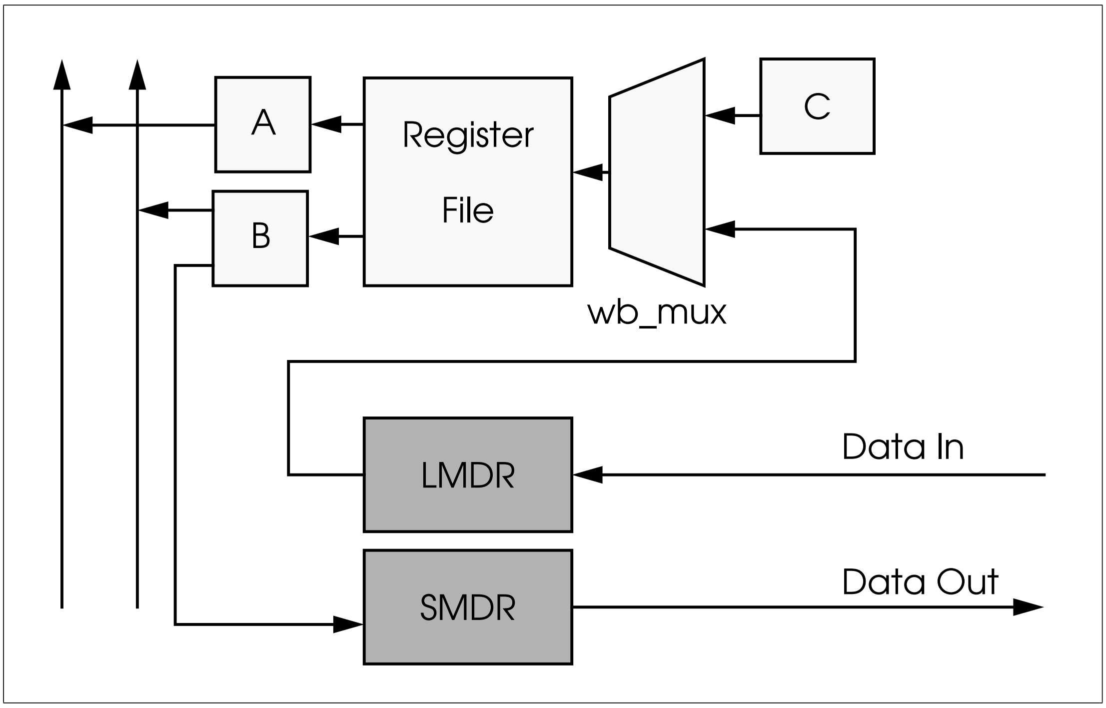

Abbildung 3-3: LMDR und SMDR Register

(„Hasards“) näher eingegangen.

3.3.2 arithmetische und logische Instruktionen

Diese Befehle benötigen keine MEM-Phase (Tabelle 5), könnten ihr Ergebnis folglich gleich nach der EXE-Phase in das Registerfile schreiben.

Tabelle 5: Arithmetische und logische Operationen

Phase	Aktionen
EXE	C <- A op B
MEM	
WB	RD <- C

Dies aber führt zu einer Kollision, falls ein vorhergehender Ladebefehl, der auch das Registerfile beschreiben muß, gerade seine MEM-Phase beendet hat. Daher muß das Ergebnis zwischengespeichert werden. Wir benötigen ein Warteregister `aluout_reg` für das ALU-Ergebnis (Tabelle 6; Abbildung 3-4).

Tabelle 6: Arithmetische und logische Operationen

Phase	Aktion
EXE	Aluout <- A op B
MEM	C <- Aluout
WB	RD <- C

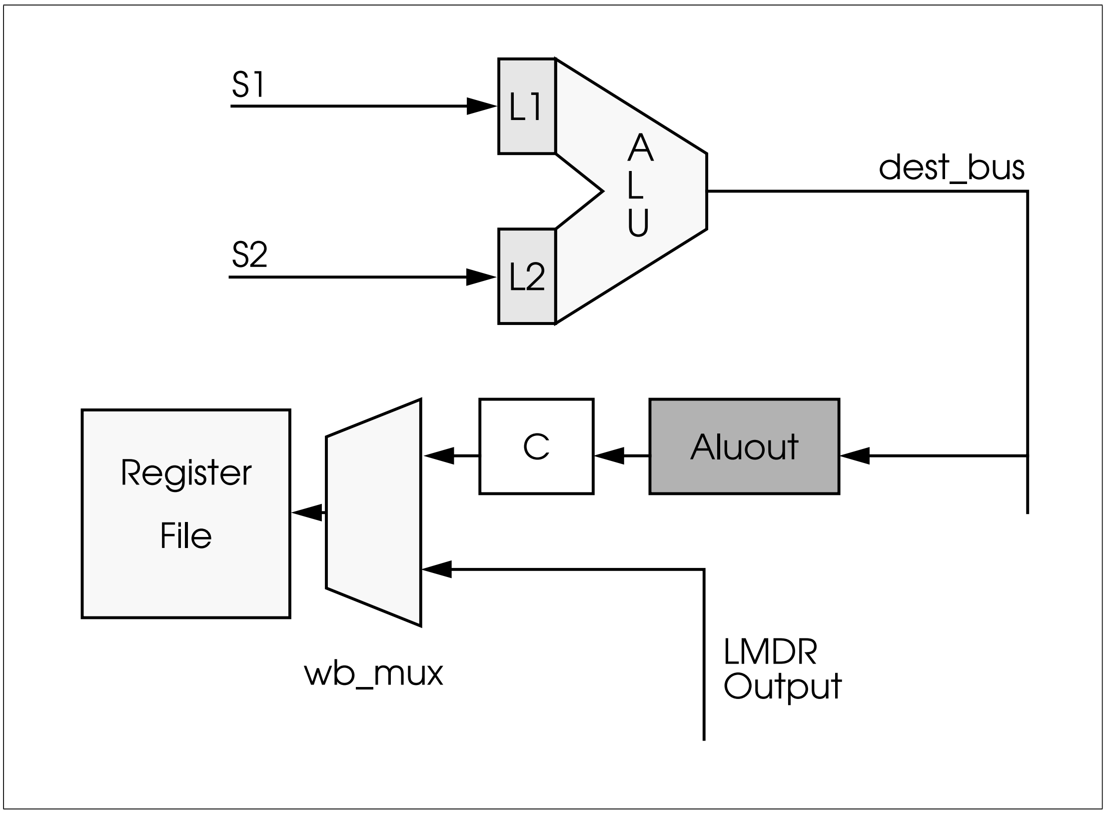

Abbildung 3-4: Aluout-Register um die MEM-Phase zu überbrücken

3.3.3 Test and Set

„Test and Set“-Instruktionen sind ein Sonderfall arithmetischer Operationen. Hier werden zwei Quellregister subtrahiert, das Ergebnis wird aber nicht in das Zielregister geschrieben. Das Zielregister erhält abhängig von der Art der „Test and Set“-Instruktion (`seq`, `sne`, `slt`...), sowie dem Ergebnis der Subtraktion den Wert „0“ oder „1“.

Im DLXS wird diese Befehlsklasse mit einer zweiten EXE-Phase bearbeitet. Zunächst wird die Subtraktion in der ALU ausgeführt, anschließend wird abhängig von deren Ergebnis ein Wert „0“ oder „1“ von den Buskonstanten `const` generiert. Dieser gelangt über die ALU ins C-Register und in der WB-Phase in das Registerfile.

Auch hier muß ein Weg gefunden werden, die Operation auf eine EXE-Phase zu beschränken. Da die Buskonstanten mit den S1 und S2 Bussen verbunden sind, auf denen bereits die Quellregister zur Subtraktion geschrieben werden, wurden zwei neue Buskonstanten, `const_0` und `const_1`, eingebaut (Abbildung 3-5).

Diese schreiben über zwei MUX-Bausteine in das Aluoutput-Register. Wenn ein „Test and Set“-Befehl dekodiert wurde, wird der Eingang des Aluout-Registers durch das Signal `test_and_set` auf den Ausgang des `set_mux` gelegt, welches durch das Signal `test_result` angesteuert, eine „0“ oder „1“ bereitstellt. Das Signal `test_result` kann erst nach der Subtraktion, aber noch innerhalb der EXE-Phase, bereitgestellt werden nachdem der ALU-Status (`alu_zero`, `alu_neg`) mit der Instruktion verglichen wurde.

Das oben eingeführte Aluout-Register muß auch hier als Zwischenspeicher aushelfen, da das C-Register nur in der MEM-Phase beschrieben werden darf (Tabelle 7).

Tabelle 7: Testoperationen

Phase	Aktion
EXE	Rs1 sub Rs2 / immed Aluout <- ‚1‘ / ‚0‘
MEM	C <- Aluout
WB	RD <- C

3.3.4 Branch / Jump

Sprünge sind unbedingte Verzweigungen und können wie Verzweigungen behandelt werden. Die ALU steht nur in der EXE-Phase zur Verfügung,

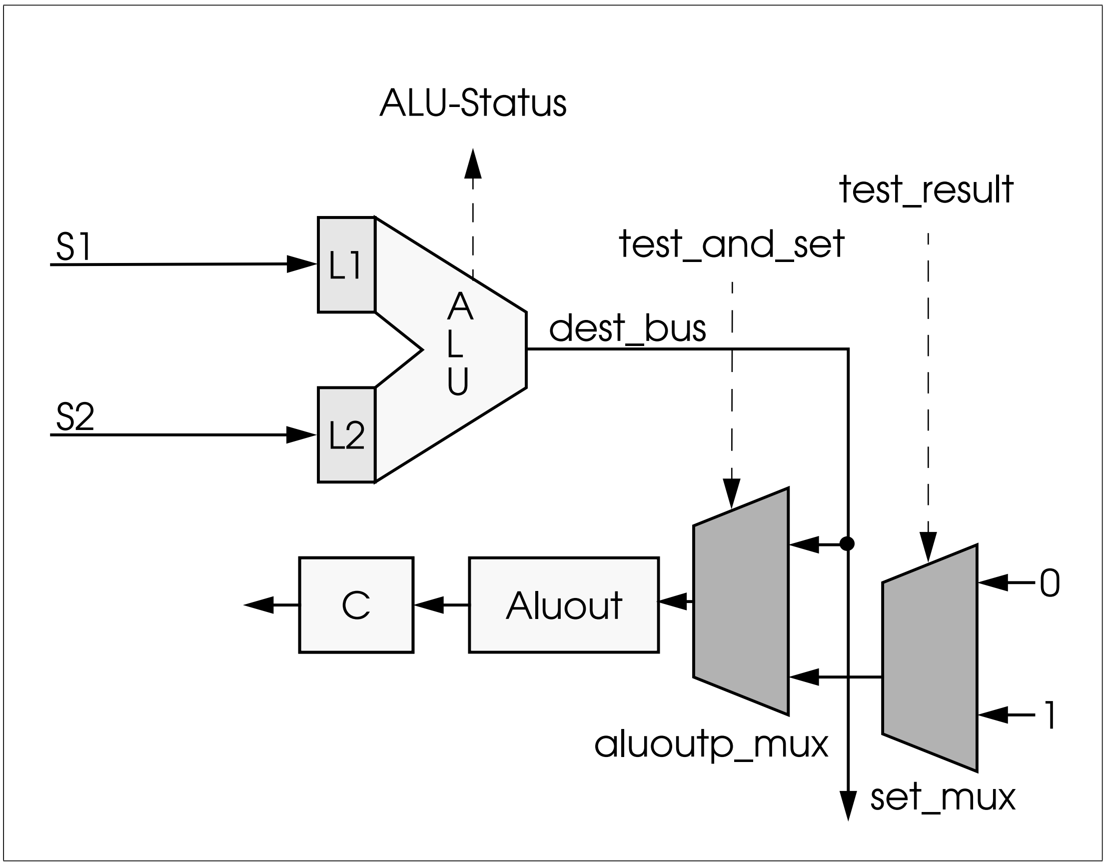

Abbildung 3-5: zusätzliche Buskonstanten

daher wird das Sprungziel auch bei einem „branch“ berechnet, obwohl die Bedingung erst in der MEM-Phase geprüft werden muß .

Bis ein Befehl die EXE-Phase erreicht, wurde der Befehlszähler allerdings schon inkrementiert. Deshalb muß er von der ID-Phase in Register `PC_1` zwischengespeichert werden (Tabelle 8).

Tabelle 8: Bedingte und unbedingte Sprünge

Phase	Aktion
ID	PC_1 <- PC
EXE	MAR <- immed + PC_1
MEM	PC <- MAR (if cond)
WB	

Die Übernahme des Sprungziels in den Befehlszähler geschieht dann in der MEM-Phase.

Da der Datenpfad von MAR nach PC nur über die ALU führt, muß ein zusätzlicher Pfad MAR -> PC eingebaut werden. Gleichzeitig dient das MAR aber nur noch als Speicheradressregister und nicht mehr als temporäres Zwischenregister für Operationen mit mehr als einer EXE-Phase. Da wir auf fünf Pipelinephasen beschränkt sind mußten diese zusätzlichen Aktionen durch neue Pfade überbrückt werden. Deshalb kann man das MAR vom S2-Bus trennen. (Abbildung 3-6).

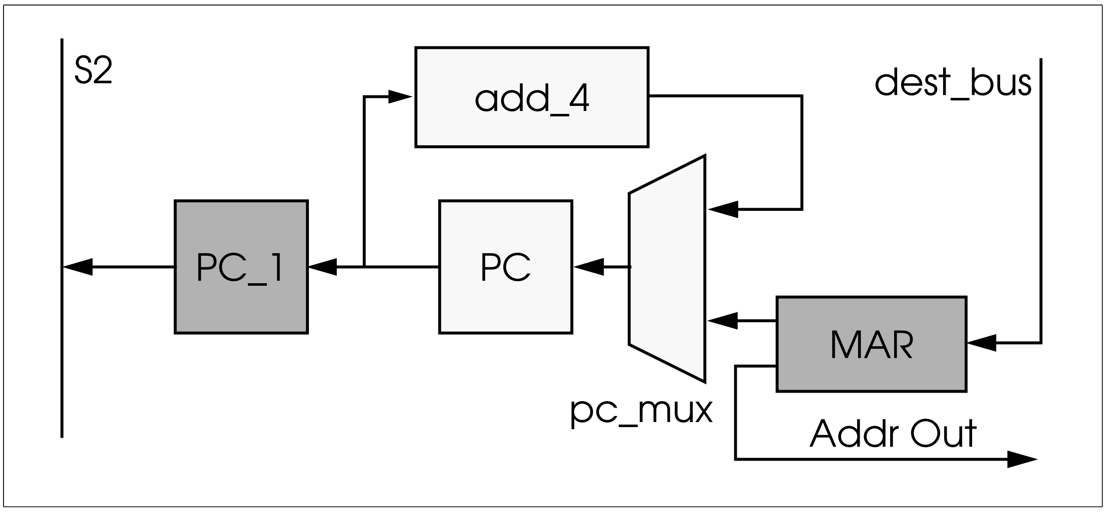

Abbildung 3-6: Sicherung des Befehlszählers

Die Auswertung der Bedingung bei einer Verzweigung darf allerdings nicht in der ALU erfolgen, da sie nur in der EXE-Phase zur Verfügung steht. Die Lösung dieses Problems wird im Kapitel 4, „Hasards“, beschrieben da sie eng mit diesem Thema verknüpft ist.

Der Sprung mit Angabe der Adresse in einem Zielregister ist ein einfacherer Fall des unbedingten Sprungs, da die ALU das Sprungziel nicht berechnen muß. Sie wird einfach zum Weiterreichen des A-Registers benutzt (Tabelle 9).

Tabelle 9: Sprung nach Zielregister

Phase	Aktion
EXE	MAR <- A
MEM	PC <- MAR
WB	

3.3.5 Jump and Link

„Jump and Link“-Befehle sind ebenfalls unbedingte Sprünge, jedoch mit Sicherung des Befehlszählers im Registerfile. Die ALU kann den Inhalt des Befehlszählers nicht weiterreichen, da sie das Sprungziel berechnet, daher benötigen wir einen weiteren Pfad PC -> C (Tabelle 10).

Tabelle 10: Sprung mit Sicherung des PC

Phase	Aktion
EXE	MAR <- imm + PC
MEM	C <- PC PC <- MAR
WB	RD <- C

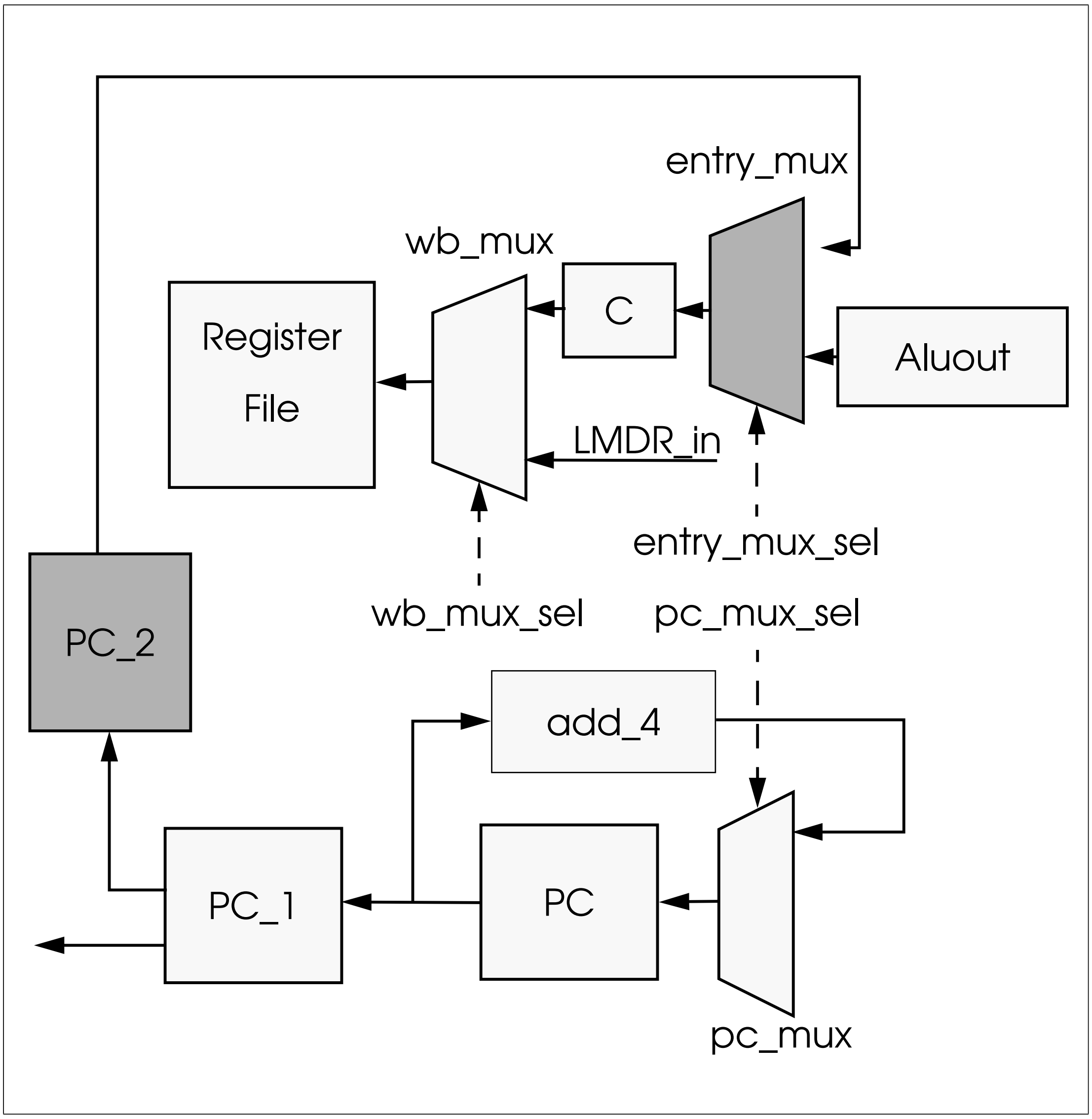

Abbildung 3-7: Doppelte Zwischenspeicherung des Befehlszählers

Da das C-Register erst in der MEM-Phase beschrieben werden darf, bis zu dem Zeitpunkt aber drei Takte vergangen sind, muß der Befehlszählerinhalt zweimal zwischengespeichert werden. Zunächst in `PC_1` und anschließend in `PC_2` (Abbildung 3-7).

Schließlich gehört in diese Befehlsklasse noch der `JALR`−Befehl, der ebenfalls den Befehlszähler im Registerfile sichert. Hier wird die ALU nur zum Weiterreichen der Daten benötigt (Tabelle 11).

Tabelle 11: Sprung nach Zielregister mit Sicherung des PC

Phase	Aktion
EXE	MAR <- A
MEM	C <- PC PC <- MAR
WB	RD <- C

In der MEM-Phase wird der Befehlszähler einerseits mit dem neuen Ergebnis beschrieben, andererseits muß er den alten Inhalt weiterreichen. Um ein „Durchrutschen“ der Daten zu vermeiden, was auch an anderen Stellen des Datenpfades möglich ist, wurden im Datenpfad alle Register durch Master-Slave-Register ersetzt.

3.3.6 Trap

Das Sprungziel bei einem Softwareinterrupt wird in der ALU, und damit in der EXE-Phase berechnet. Bevor in der MEM-Phase das Ziel in den Befehlszähler übernommen werden kann, müssen allerdings noch der alte Inhalt des Befehlszählers und die Interrupt-Kontrollbits gesichert werden.

Tabelle 12: Softwareinterrupts

Phase	Aktion
EXE	MAR <- TBR + immed ICR <- ICR shifted IAR <- PC_1
MEM	PC <- MAR
WB	

Zusammen mit der Berechnung des Sprungziels werden die speziellen Register `IAR` und `ICR` in der EXE-Phase beschrieben. Dadurch benötigen wir wieder einen „Bypass“ PC -> IAR, bzw. PC_1 -> IAR, da erst in der EXE-Phase der Befehlszähler gelesen wird. Das iar_mux lenkt die Daten in das IAR-Register.

Außerdem muß ein zusätzlicher Bit-shifter die ICR-Bits in die obere Hälfte des Registers schieben, da die ALU bereits belegt ist (Abbildung X).

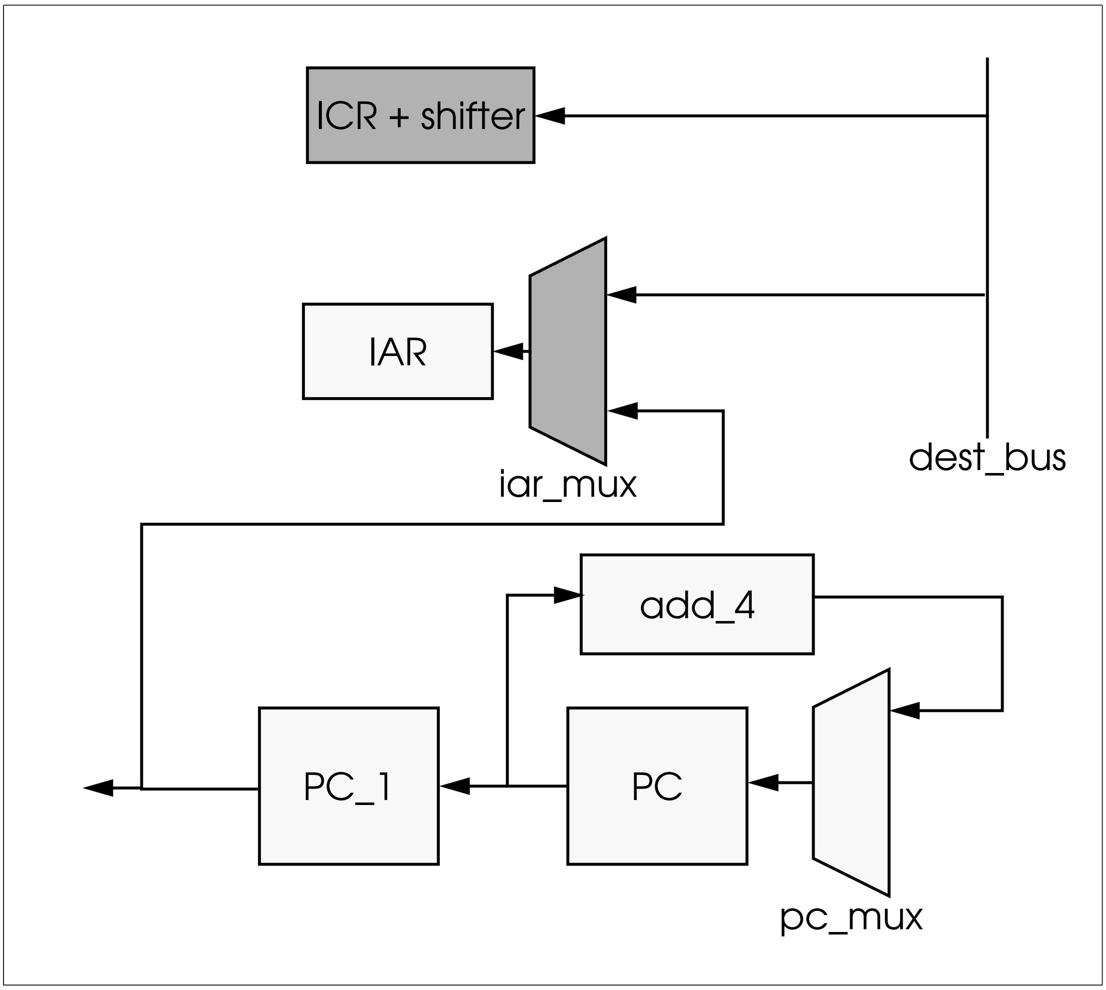

Abbildung 3-8: Sicherung des PC bei Softwareinterrupts

3.3.7 RFE

Bei der Rückkehr aus einem Interrupt läßt die ALU das IAR passieren und kann auch hier nicht die ICR-Bits verschieben. Der oben erwähnte Bit-shifter muß also die Bits auch rückwärts schieben können (Tabelle 13).

Tabelle 13: Rückkehr aus einem Interrupt

Phase	Aktion
EXE	ICR <- ICR shifted MAR <- IAR
MEM	PC <- MAR
WB	

3.3.8 MOVi2s

Dies ist ein einfacher Befehl, der nur die EXE-Phase benötigt. Die ALU schiebt einen Registerinhalt in eines der speziellen Register . Da dies ebenfalls in der EXE-Phase geschieht, kann es nicht zur Kollision mit TRAP-Befehlen kommen (T abelle 14).

Tabelle 14: Spezielles Register beschreiben

Phase	Aktion
EXE	IAR / ICR / TBR <- A
MEM	
WB	

3.3.9 MOVs2i

Um die speziellen Register zu sichern, werden sie über die ALU in das Registerfile bewegt. Um Kollisionen zu vermeiden, darf wie bei allen anderen Befehlen, das C-Register erst in der MEM-Phase beschrieben werden (Tabelle 15).

Tabelle 15: Spezielles Register im Registerfile speichern

Phase	Aktion
EXE	Aluout <- IAR / ICR / TBR
MEM	C <- Aluout
WB	RD <- C

3.4 Der neue Datenpfad

Alle bisher angesprochenen zusätzlichen Pfade und Register werden nun in den Datenpfad eingebaut. Abbildung 3-9 zeigt den neuen, für die Pipeline erweiterten Datenpfad. Aus Gründen der Übersichtlichkeit wurden die `test_and_set`-Multiplexer nicht eingezeichnet. Diese können auch als T eil der ALU implementiert werden.

Zu bemerken sind die neuen Schnittstellen `Data Address` für die Zieladresse eines Speicherbefehls und `Instruction Address` für die Adresse des nächsten Befehlswortes. Außerdem besitzt der Datenpfad nun zwei Datenbusse. `Instruction In` transportiert Instruktionen und `Data In` bzw. `Data Out` zu lesende oder schreibende Daten. Die Speicherkontrollleitungen sind nicht eingezeichnet, da sie zur Steuerung gehören. Beide Datenbusse besitzen ihre eigenen Kontrolleitungen, da gleichzeitig ein Schreibzugriff der MEM-Phase und ein Lesezugriff der IF-Phase auftreten können, die verschiedene Kontrollsignale benötigen.

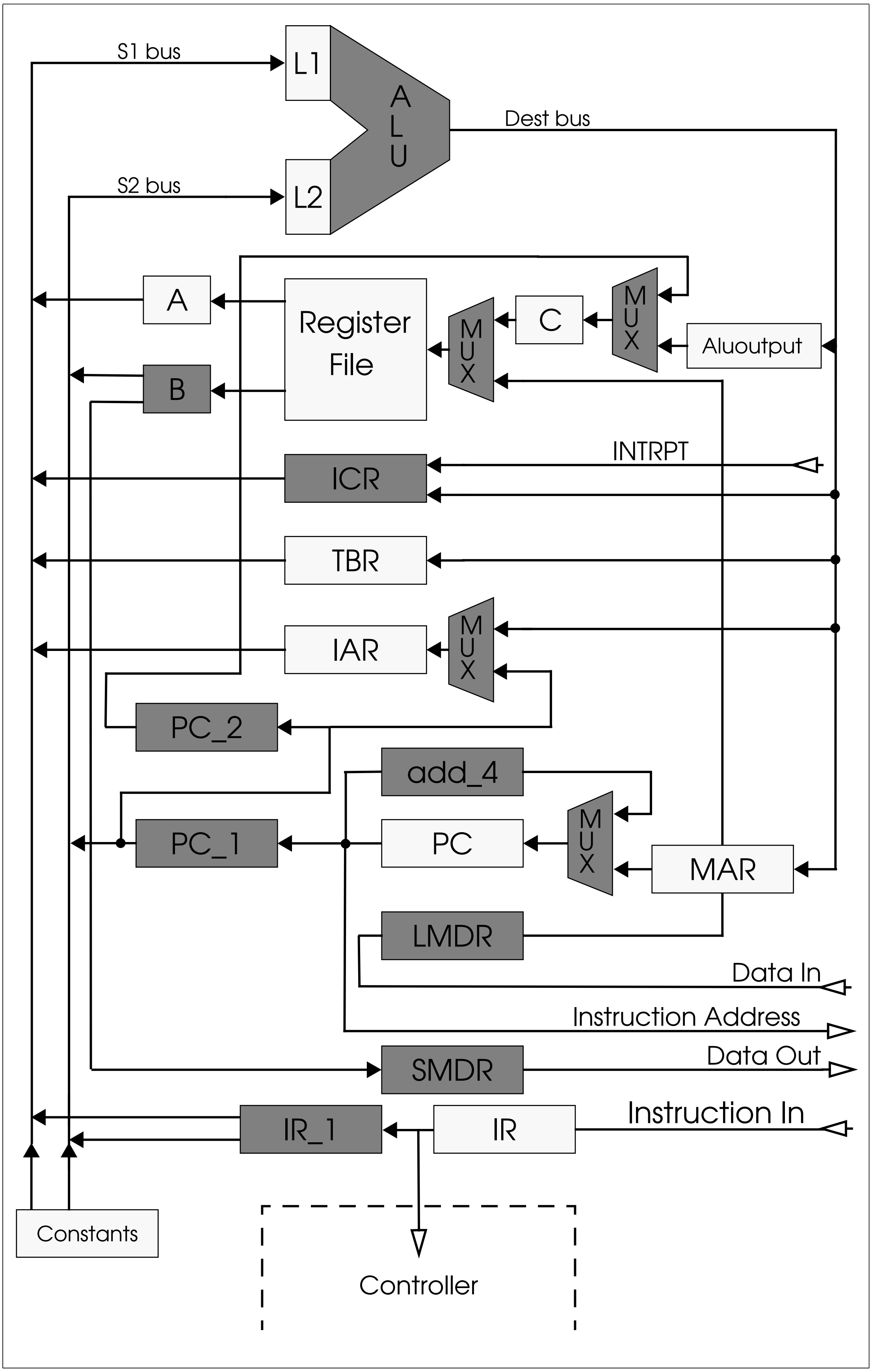

Abbildung 3-9: Der neue Datenpfad

Um pinkompatibel zum DLXS ohne Pipeline zu bleiben, wurden diese Datenleitungen mit einer Datenweiche auf die originalen Schnittstellen geleitet. So mußten keine Änderungen an der Testbench des Prozessors vorgenommen werden.[1]

Alle Komponenten im Datenpfad benötigen zu ihrer Aktivierung Kontrollsignale, die von der Steuerung geliefert werden. Die Steuerung muß den Datenfluß im Datenpfad so steuern, daß die zu unterschiedlichen Befehlen gehörenden Daten, die sich gleichzeitig im Datenpfad befinden, nicht kollidieren.

3.5 Die neue Steuerung

Der endliche Automat (FSM) der Steuerung des DLXS liefert jeden Takt die Kontrollsignale für den aktuellen Arbeitsschritt des zu bearbeitenden Befehls, abhängig von dem Befehl. Zur Steuerung einer Pipeline müßte er jeden Takt die Kontrollsignale für unterschiedliche Arbeitsschritte zu unterschiedlichen Befehlen, die sich gleichzeitig im Datenpfad befinden, liefern. Bei 13 verschiedenen Arbeitsabläufen in jeweils fünf Schritten benötigt der Automat $13^5 = 371293$ verschiedene Zustände. Er ist also ungeeignet für die Kontrolle der Pipeline.

Der endliche Automat wurde ersetzt durch 4 Register, die die Kontrollsignale für die Bearbeitung eines Befehls in fünf Takten erhalten, weiterschieben und dabei die für den jeweiligen Arbeitsschritt notwendigen Signale abzweigen. Dazu kommt eine Kontrolleinheit, die diese Register überwacht. Die vier Register sind in Abbildung X nach den Pipelinephasen benannt, da ihre Aufgabe darin besteht, die Arbeitsschritte der jeweiligen Phase auszuführen (Abbildung 3-10).

Mit steigender Taktflanke von phi1 beginnen alle Einheiten zu arbeiten und ihre jeweiligen Kontrollsignale zu generieren.

3.5.1 Instruction Fetch (fetch.vhd)

Die Komponente IF ist dafür verantwortlich, daß jeden Takt ein neues Befehlswort aus dem Speicher in das Instruktionsregister geladen wird, sowie für die Inkrementierung des Befehlszählers. Sie besitzt einen Eingang `stall_fetch`, der diese Operation verhindern kann. Außerdem enthält sie einen `READY`-Eingang, mit dem ihr der Speicher mitteilt, daß das Befehlswort für das Instruktionsregister bereitsteht. Wenn dies innerhalb eines Taktes nicht der Fall ist, meldet IF über das Signal `fetch_memory_not_ready`, daß diesen Takt kein Befehlswort aus dem

1. Ähnliche Kompromisse wurden bei Intel 8088 Prozessoren geschlossen, deren 16 Bit breiter interner Adressbus an 8 Bit breite Schnittstellen geleitet werden mußte, um die neuen Prozessoren auf alten 8 Bit Motherboards zu verwenden.

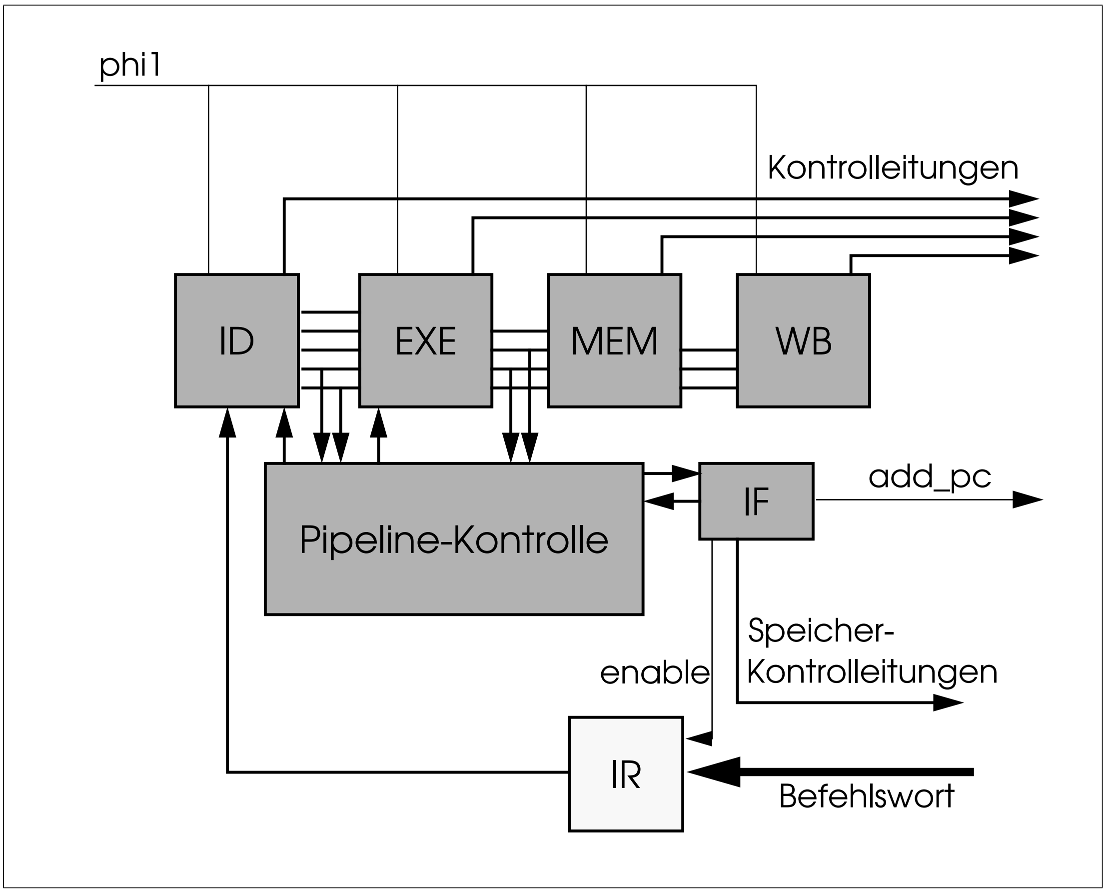

Abbildung 3-10: Die neue Steuerung

Speicher geholt werden konnte. Damit unterbricht IF die Inkrementierung des Befehlszählers (Abbildung 3-11).

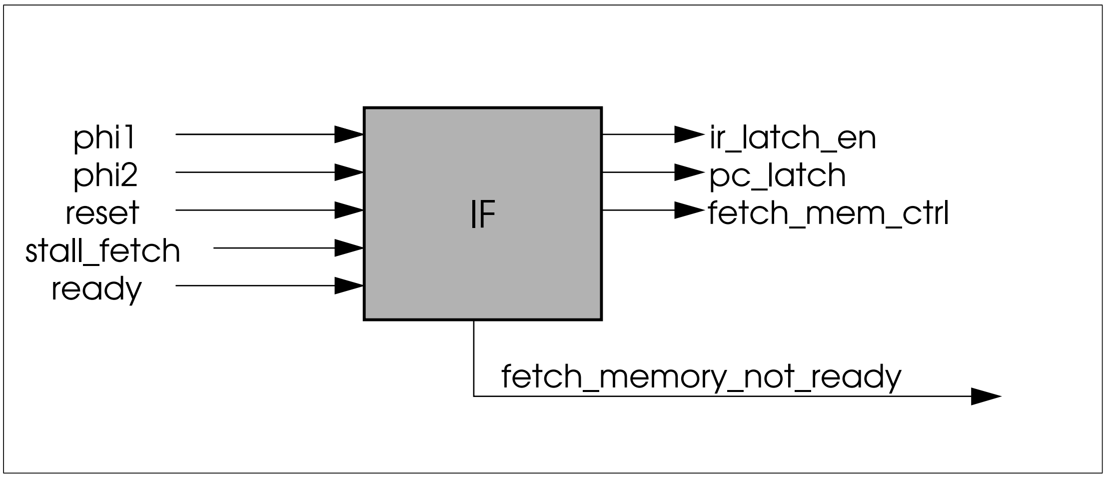

Abbildung 3-11: Komponente IF

3.5.2 Instruction Decode (decode.vhd)

Der durch IF geholte Befehl gelangt über den 32 Bit breiten Bus `instr_in` zur Komponente ID. ID enthält die drei Instruktionsdekodierer des DLXS um festzustellen, um welchen Befehl es sich handelt.

Abhängig von den Ergebnissen der Dekodierer werden alle zur Bearbeitung des Befehls notwendigen Kontrollsignale generiert, auch die erst in späteren Phasen benötigten. Die zu diesem Zeitpunkt notwendigen Signale werden dem Datenpfad zur Verfügung gestellt.

Dazu gehören die Nummern der Quellregister des Befehls, welche an das Registerfile geleitet werden, um `A`- und `B`-Register zu laden, sowie die Signale zur Zwischenspeicherung des Befehlswortes von `IR` nach `IR_1`, da dieses im nächsten Takt eventuell benötigt wird (Abbildung 3-12).

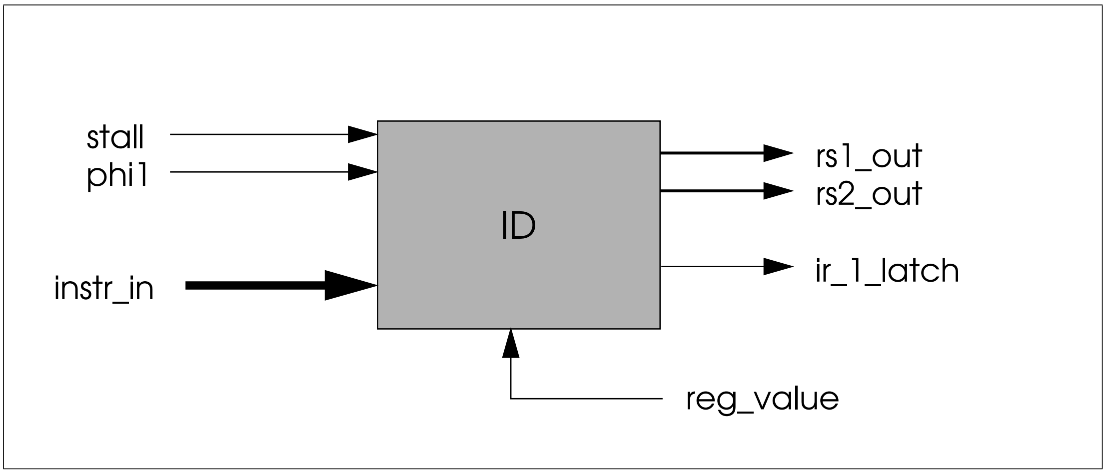

Abbildung 3-12: Komponente ID

Mit dem Signall `stall` kann ID gezwungen werden, einen `nop`-befehl zu dekodieren.

`reg_value` stammt aus dem Datenpfad. Es gibt an, ob das von `rs1_out` identifizierte Register einen Wert ungleich Null enthält.

3.5.3 Execution Stage (exec.vhd)

ID leitet die Kontrollsignale, die erst später benötigt werden an Komponente EXE weiter. Hier werden die Kontrollsignale zur Ausführung der Instruktion an den Datenpfad abgezweigt (Abbildung 3-13).

Mit `s1_enab` und `s2_enab` wird festgelegt, welche Komponente die Quelloperatoren des Befehls liefert (`A`, `B`, `iar`, `icr`, `tbr`, `ir_1` oder eine Buskonstante). `alu_op_sel` bestimmt die ALU-Operation und `dest_enab` schließlich das Zielregister (`aluout`, `TBR`, `ICR`, `IAR` oder `MAR`). `const_sel` bestimmt, welche Buskonstante auf welchen Bus geschrieben werden soll, `immed_sel` wird benötigt, wenn der Befehl einen unmittelbaren Wert als Quelloperand beinhaltet.

Bei einem Schreibzugriff auf den Speicher wird mit `smdr_latch` das Speicherdatenregister beschrieben und `smdr_ctrl` liefert die Art des Schreib-

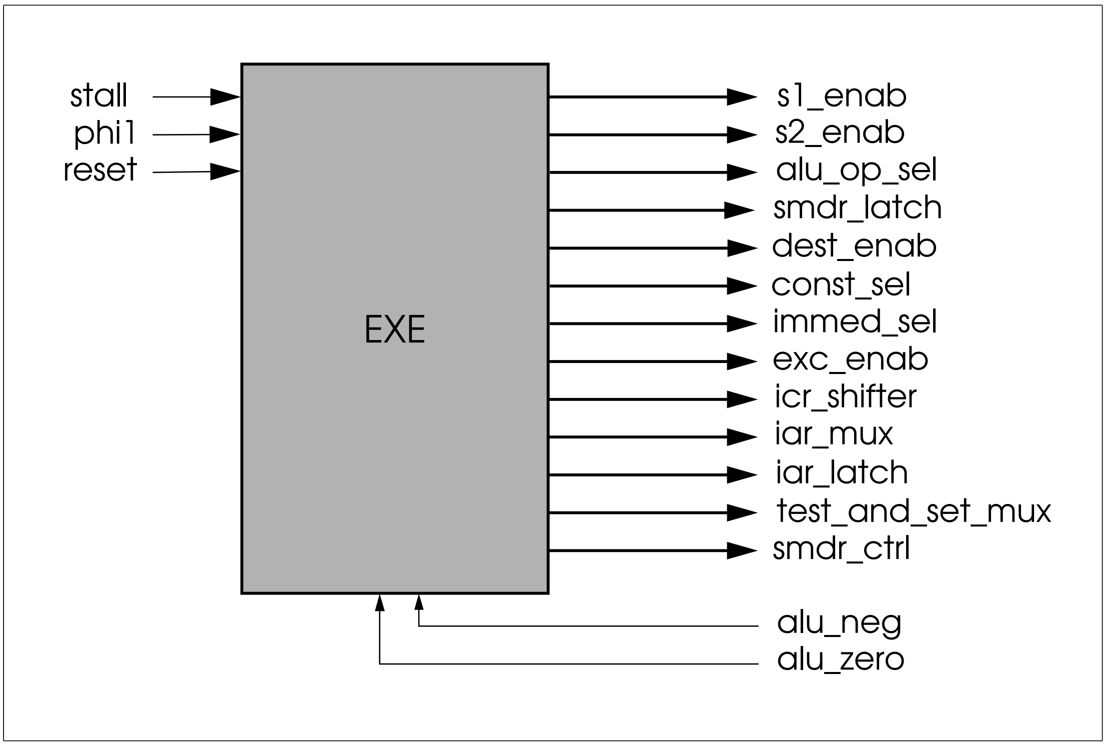

Abbildung 3-13: Die EXE-Phase

zugriffs (W ort-, Halbwort oder Bytezugriff).

`exc_enab` setzt ein Ausnahmebit im `ICR`-Register und `icr_shifter` verschiebt die Ausnahmebits in die obere oder untere Hälfte des Registers. Mit `iar_mux` kann der W ert des Befehlszählers in das `IAR`-Register gelenkt werden.

EXE erhält durch `alu_zero` und `alu_neg` ein Feedback der ALU-Operation die sie gerade veranlaßt hat. Es wird zur Ausführung der sogenannten „Test and Set“-Befehle benötigt. Prozeß `mux_selector` ist verantwortlich für die korrekte Ansteuerung der Mulitiplexer , die die Konstanten in das Aluout-Register lenken. Ihre Steuersignale werden durch `test_and_set_mux` ausgegeben.

Bei aktivem `stall`-Eingang gibt EXE Kontrollsignale zur Bearbeitung eines `NOP`-Befehls an den Datenpfad und an die nachfolgenden Phasen weiter . Der aktuelle Befehl bleibt jedoch gespeichert. Bei einem `RESET`-Signal, initialisiert EXE das `ICR`-Register und erzeugt die Null-Konstante, die in den Befehlszähler geladen wird.

3.5.4 Memory Access (memory.vhd)

Von EXE werden die restlichen Kontrollsignale an MEM weitergeleitet (Abbildung 3-14). Hier werden die Kontrollsignale für den Speicherzugriff erzeugt. Sie konnten nicht bereits durch ID generiert werden, da erst im vorherigen T akt die Zieladresse der Speicherzelle berechnet und ins Speicheradressregister `MAR` gebracht wurde.

Die „Least Significant Bits" der Adresse werden als `adr_ls2(0 to 1)`-Eingangsvektor zur MEM-Phase geführt. Im Prozeß control_out wird mit

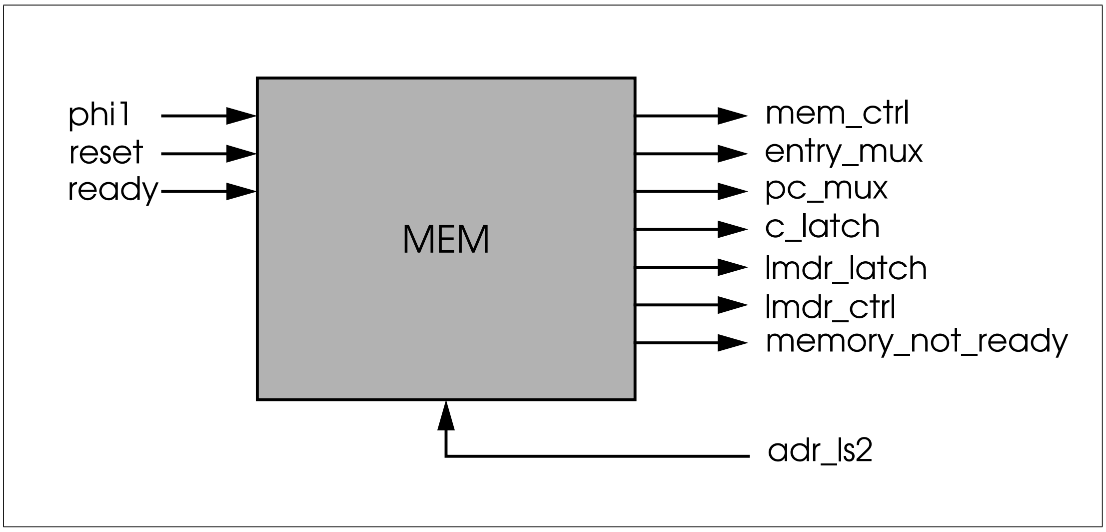

Abbildung 3-14: Die MEM-Phase

einem `case`-Statement aufgrund der Kombination von `adr_ls2` und der Art des Speicherzugriffs (Wortzugriff, Bytezugriff oder Halbwortzugriff) darüber entschieden, welches Byte bzw. welche Worthälfte der Speicher bereitstellen soll. Damit wird das `ENABLE`-Signal festgelegt. An den Speicher gelangt `mem_ctrl`, ein Vektor aus `ENABLE`, `ERROR` und `RW` Signalen.

Nach dem Setzen der Speicherkontrolleitungen erwartet MEM innerhalb eines Taktes das `READY`-Signal aus dem Speicher, sonst sendet MEM an die Pipeline-Kontrolle das Signal `memory_not_ready`. Mit `lmdr_latch` wird der vom Speicher gelieferte Wert in das Ladedatenregister (`LMDR`) übernommen und mit `lmdr_ctrl` die Vorzeichenbits bei Halbwort oder Bytezugriff gesetzt.

Für die restlichen Aktionen der MEM-Phase (siehe Kapitel X, „Befehlsklassen") werden die Signale `c_latch` und `entry_mux` ausgegeben, welche dafür sorgen, daß das Ergebnis einer ALU-Operation ins `C`-Register gelangt.

Bei einem Sprungbefehl wird durch `pc_mux` die Sprungzieladresse in den Befehlszähler geladen. Bei einem `RESET` gelangt die durch die EXE-Komponente erzeugte Null vom `MAR` in den Befehlszähler.

3.5.5 Write Back (WB)

WB sorgt dafür, daß bei einer Registeroperation das Ergebnis ins Zielregister gelangt (Abbildung 3-15). Dazu muß mit rd die Zielregisternummer angegeben und durch `rf_in` das Registerfile auf Dateneingang geschaltet werden. `wb_mux` lenkt bei einem Ladebefehl den Wert des `LMDR` in das Registerfile.

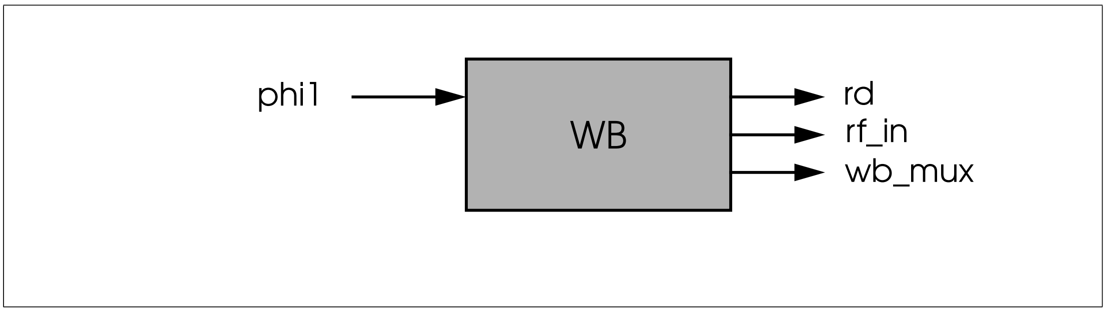

Abbildung 3-15: Die WB-Phase

3.5.6 Die Pipeline-Kontrolle

Die Pipeline-Kontrolle überwacht die Arbeit der Register und hat die Möglichkeit, das Weiterschieben der Kontrollsignale in die nächste Phase zu unterbrechen. Dazu dienen die low-aktiven Signale `stall_fetch`, `stall_decode` und `stall_exec`, welche die Register IF, ID und EXE anhalten. Die Auswirkungen auf die Register sind verschieden.

`stall_fetch` verhindert, daß der Befehlszähler inkrementiert wird, und das Instruktionsregister `IR` das Befehlswort übernimmt.

`stall_decode` veranlaßt die ID-Komponente, daß ein `nop`-Befehl dekodiert und die entsprechenden Steuersignale erstellt werden.

`stall_exec` verhindert die Übernahme der anstehenden Kontrollsignale der ID-Komponente in das EXE-Register. Bei aktiven `stall`-Eingang reicht das EXE-Register an das MEM-Register und an den Datenpfad Kontrollsignale zur Bearbeitung eines `nop`-Befehls weiter. Die zuletzt empfangenen Kontrollsignale bleiben aber im Register gespeichert.

Die Stall-Signale werden in der Pipeline-Kontrolle von der `hazard_fsm`, einem endlichen Automaten erzeugt. Im nächsten Kapitel wird die Funktion dieses Automaten erläutert.

4 Hasards

Wenn die Abarbeitung eines Befehls unabhängig von seinen Vorgängern und Nachfolgern ist, läuft die oben vorgestellte Pipeline fehlerfrei. Leider ist dies nur in den seltensten Fällen so. Meistens hängen die Befehle voneinander ab, indem sie Ergebnisse ihrer Vorgänger lesen und wiederum ihre Ergebnisse ihren Nachfolgern bereitstellen.

Erst wenn ein Befehl die WB-Phase verläßt ist er vollständig abgearbeitet. Ein nachfolgender Befehl muß also warten, bis das Ergebnis seines Vorgängers im Registerfile abgelegt wurde, falls er darauf zugreifen will. Wenn ein Befehl auf einen anderen Befehl warten muß, weil die korrekte Abarbeitung des Programms davon abhängt, spricht man von einem Pipeline-Hasard.

Es gibt drei Hauptgründe für eine Situation in der ein Befehl auf einen Vorgänger warten muß. Nach ihnen werden die Hasards eingeteilt in Struktur-Hasards, Steuer-Hasards und Daten-Hasards.

4.1 Struktur-Hasards

Ein Struktur-Hasard ist ein Betriebsmittelkonflikt. Er tritt auf, wenn mehrere Befehle gleichzeitig dieselbe Funktionseinheit nutzen wollen. Durch die Strukturänderungen im Datenpfad wurden zwar die meisten Struktur-Hasards beseitigt, dennoch gibt es noch bestehende Probleme.

Struktur-Hasards lassen sich, wie bereits im Datenpfad gezeigt, durch zusätzliche Betriebsmittel vermeiden. In diesem Fall hat der Datenpfad des DLXS/P zwei Daten und zwei Adressbusse, sowie zwei Bündel Speicherkontrolleitungen erhalten. Diese müßten jeweils in einen Daten- und einen Instruktionsspeicher führen.

Der Prozessorkern des DLXS besitzt jedoch nach wie vor nur einen Daten- sowie einen Adressport und ein Bündel Speicherkontrolleitungen. Bei einem Lese- oder Schreibzugriff muß die Pipeline für einen Takt auf den Instruction Fetch verzichten. Um eine eventuelle Erweiterung der Testbench des DLXS/P-Modells auf getrennte Daten und Instruktionsspeicher so einfach wie möglich zu gestalten wird dieser Hasard jedoch nicht durch die Pipeline-Kontrolle erkannt.

4.1.1 Die Datenweiche (data_siding.vhd)

Die Datenleitungen des DLXS/P wurden mit einer Datenweiche `data_siding.vhd` auf die originalen Schnittstellen geleitet. Diese Datenweiche gibt einem Speicherzugriff aus dem MEM-Register immer Vorrang vor einem „Instruction Fetch" der IF-Komponente. Sie liefert dabei der Steuerung nur das `READY`-Signal für den Datenzugriff und nicht für den Befehlswortzugriff (Abbildung 4-1).

Für die IF-Komponente der Steuerung erscheint diese Kollision wie ein zu langsamer Befehlswortspeicher. Deshalb sendet sie der Pipeline-Kontrolle das Fehlersignal `fetch_mem_not_ready`.

Das hat zur Folge, daß die `hazard_fsm` der Pipeline-Kontrolle in den Zustand `re_fetch` übergeht, und das `stall_decode`-Signal der ID-Komponente aktiviert. Sie bleibt so lange in diesem Zustand, bis das Signal `fetch_mem_not_ready` zurückgesetzt wird.

So lange das stall_decode-Signal aktiv ist, dekodiert die ID-Komponente NOP-Befehle, die sie an die nachfolgenden Pipelinephasen weiterreicht.

Das MEM-Register setzt die Speicherkontrolleitungen zurück, sobald das `READY`-Signal eingegangen ist. Dadurch schaltet die Datenweiche die Adress- und Datenports, sowie die Kontrolleitungen wieder zur IF-Kompo-

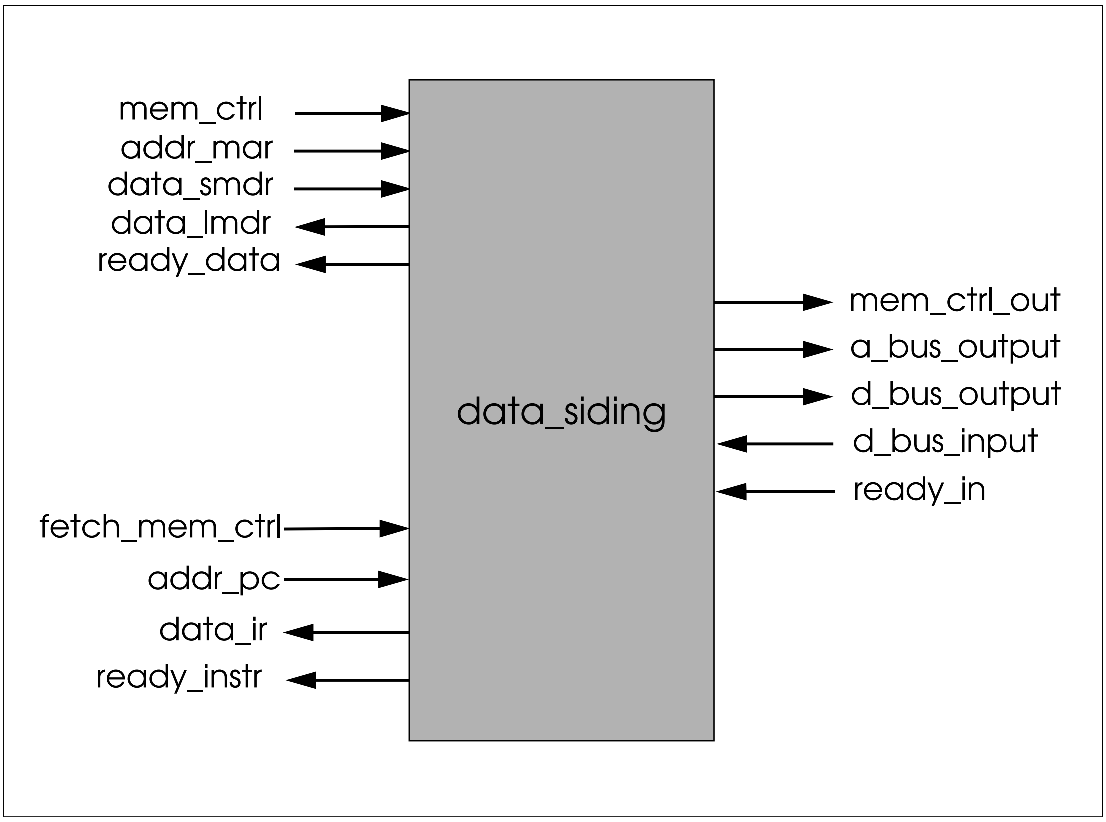

Abbildung 4-1: Die Datenweiche

nente um, welche im nächsten Takt das Befehlswort holen kann und damit `fetch_mem_not_ready` deaktiviert.

4.2 Daten-Hasards

Ein Daten-Hasard tritt auf, wenn ein Befehl in der EXE-Phase den Zieloperanden eines Vorgängerbefehls als Quelloperand lesen will. Er muß warten bis dieser das Zielregister in der WB-Phase beschrieben hat. Folgendes Programmbeispiel verdeutlicht das Problem:

```
sub R5, R4, R7
add R1, R2, R5
```

Der `add` Befehl liest als zweiten Quelloperanden das Register R5, welches Zieloperand des vorhergehenden `sub` Befehls ist. Da er bereits in der ID-Phase seine Quelloperanden einliest, während der `sub` Befehl Register R5 erst in der WB-Phase aktualisiert muß er mit der Ausführung warten (Tabelle 16).

Tabelle 16: Ein Daten-Hasard

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`sub R5, R4, R7`	IF	ID	EXE	MEM	WB $_{\text{R5 beschrieben}}$	

Tabelle 16: Ein Daten-Hasard

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`add R1, R2, R5`		IF	ID $_{\text{R5 gelesen}}$	EXE	MEM	WB

4.2.1 Die Forwarding-Hardware

Der Daten-Hasard wird durch eine Forwarding-Hardware aufgelöst. Sie besteht aus drei 4:1 MUX Bausteinen im Datenpfad und einer Forwarding-Logik, welche sich die Zielregister der letzten beiden Befehle merkt und damit feststellen kann, wo im Datenpfad der aktuelle Befehl seine Quelloperanden vorfindet. Die Forwarding-Logik befindet sich in der Pipeline-Kontrolle als Komponente `forward_logic.vhd` (Abbildung 4-2).

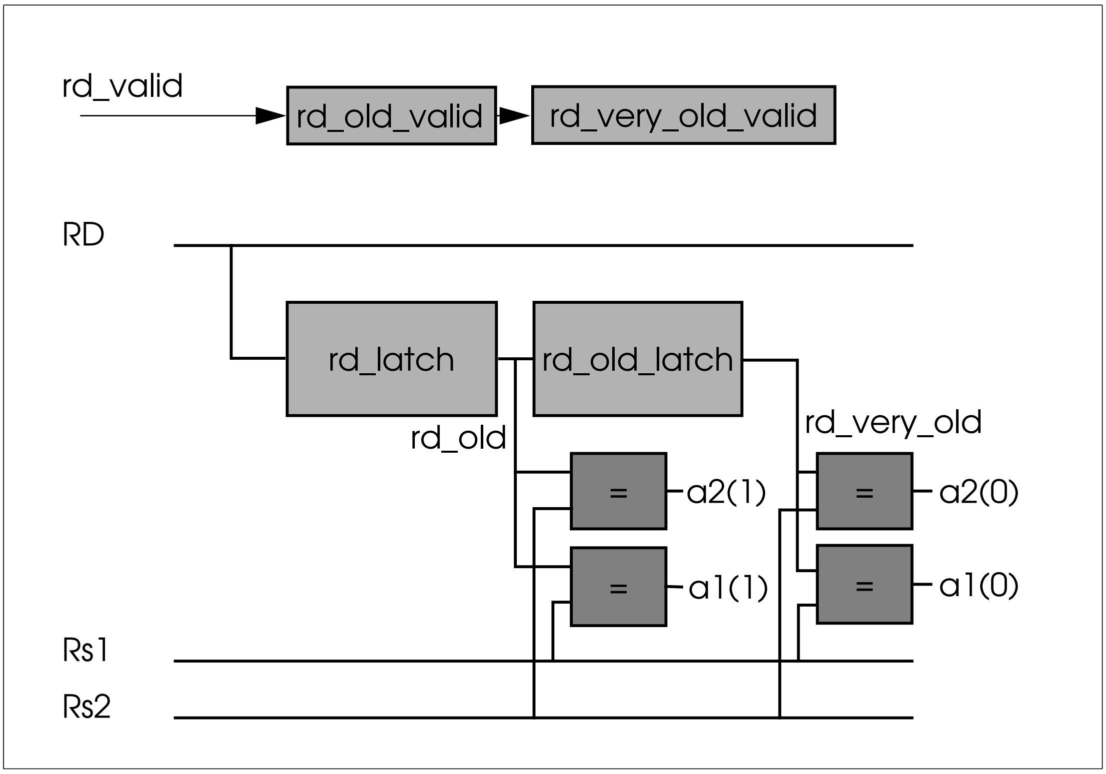

Abbildung 4-2: Die Forwarding-Logik

Die Forwarding-Logik steuert mit den Signalen `alu_mux_1_sel` und `alu_mux_2_sel` (in der Abbildung durch `a1` und `a2` abgekürzt) die MUX-Bausteine `alumux1` und `alumux2` an, wodurch die alten ALU-Ergebnisse zurück an den Eingang der ALU geleitet werden, sofern sie noch nicht ins Registerfile geschrieben wurden (Abbildung 4-3).

Um die Steuersignale der Multiplexer zu generieren wurden vier Komparatoren `comparator.vhd` eingesetzt, die die aktuellen Quellregisternummern mit den gespeicherten alten Zielregisternummern `rd_old` und `rd_very_old` vergleichen.

Der dritte Multiplexer `smdr_mux` leitet die ALU-Ergebnisse an das Spei-

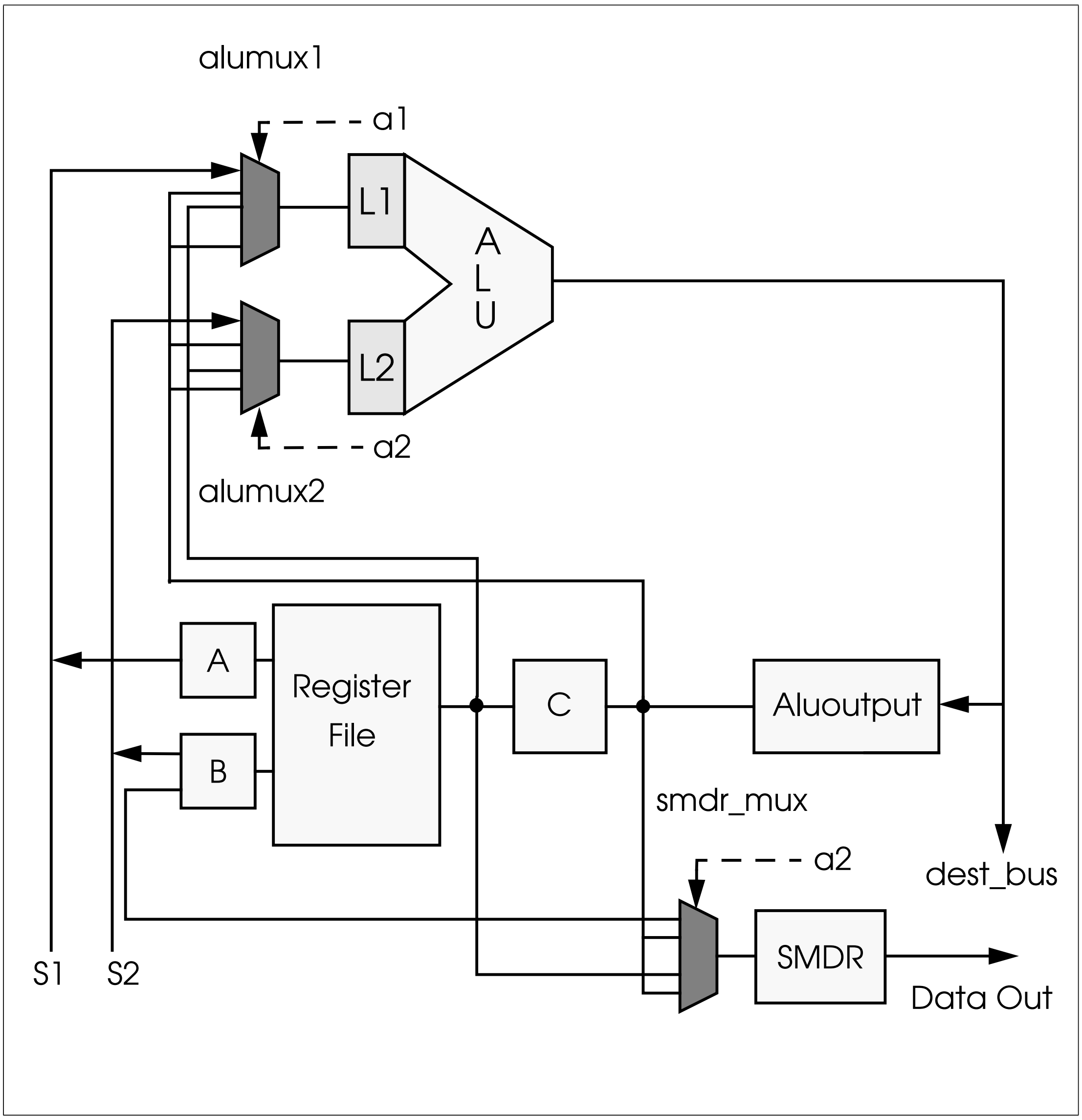

Abbildung 4-3: Die Multiplexer der Forwarding-Logik

cherdatenregister SMDR, falls unmittelbar nach den ALU-Operationen die Register gespeichert werden sollen:

```
add R1, R2, R3
sw 0(R1), R1
```

In diesem Beispiel muß das Ergebnis des `add` Befehls sowohl durch `alumux1` zur Zieladressberechnung, als auch durch das `smdr_mux` in das Speicherdatenregister aus dem Datenpfad weitergeleitet werden.

In einem DLXS-Speicherbefehl ist das zu schreibende Datenregister als Rs2 kodiert, deshalb reicht es das `smdr_mux` mit den Steuerleitungen für `alumux2` anzusteuern.

Die beiden Bit-Latches `rd_old_valid` und `rd_very_old_valid` wurden benötigt, da nicht immer sicher gestellt ist, daß die gespeicherte Zielregisternummer auch wirklich ein Zielregister eines Befehls war. Das Eingangssignal `rd_valid` ist nur dann aktiv, wenn der Befehl in der EXE-Phase das Aluout-Register oder in der MEM-Phase das LMDR beschreibt. Dies ist nur dann der Fall, wenn mindestens eine der `latch_enable` Leitungen für diese beiden Register aktiv ist. Wenn die Zielregister ungültig sind, werden die Forwarding-Multiplexer die Daten aus dem Registerfile bereitstellen.

Wenn das Registerfile in der ersten Hälfte des Takts mit `phi1` Daten einliest und in der zweiten Hälfte mit `phi2` bereitstellt, können sich innerhalb eines Taktes die Phasen WB und ID zweier aufeinanderfolgender datenabhängiger Befehle überschneiden (Tabelle 17).

Tabelle 17: Schreibzugriffe auf das Registerfile während phi1, Lesezugriffe während phi 2

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`ADD R1, R2, R3`	IF	ID	EXE	MEM	WB $_{\text{phi1}}$	
`SUB R4, R1, R5`		IF	ID	EXE	MEM	WB
`AND R6, R1, R7`			IF	ID	EXE	MEM
`OR R8, R1, R9`				IF	ID $_{\text{phi2}}$	EXE

Um dies zu erreichen wurde das Registerfile mit Master-Slave-Registern ausgestattet (`reg_file-behaviour_5.vhd`).

Es genügt also, sich die Zielregister der letzten zwei Befehle in der Forwarding-Logik zu merken. Nicht alle Daten-Hasards können ohne Eingriff der Pipeline-Kontrolle durch die Forwarding-Logik aufgelöst werden. Wenn eine Operation auf Daten eines Ladebefehls zugreift, muß sichergestellt sein, daß sich diese Daten schon im Speicher befinden. Folgende Programmsequenz ergibt einen solchen Hasard:

```
lw R1, 0(R2)
add R3, R7, R1
sub R4, R1, R3
```

Der Ladebefehl erhält seine Daten erst am Ende der MEM-Phase, während der nachfolgende `add`-Befehl bereits am Anfang der EXE-Phase Register 1 liest (Tabelle 18).

Tabelle 18: Ein Daten-Hasard der nicht gelöst werden kann

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`lw R1, 0(R2)`	IF	ID	EXE	MEM	WB	
`add R3, R7, R1`		IF	ID	EXE	MEM	WB

Tabelle 18: Ein Daten-Hasard der nicht gelöst werden kann

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`sub R4, R1, R3`			IF	ID	EXE	MEM

Um diesen Hasard zu erkennen, erhält die Forwarding-Logik einen Eingang `load_instr`, der von der ID-Komponente bei einem Ladebefehl gesetzt wird. Der Zustand dieses Eingangs wird für einen Takt im Bit-Latch `rd_old_from_memory` gespeichert.

Wenn der folgende Befehl Register liest, die zuvor Zielregister einer Ladeoperation waren (was durch den Zustand von `rd_old_from_memory` festgestellt werden kann), aktiviert die Forwarding-Logik das Eingangssignal `forward_from_memory` der `hazard_fsm`, die durch Anhalten von IF, ID und EXE die Ausführung der Folgebefehle um einen Takt verzögert. Dadurch gelangt der aus dem Speicher zu ladende Wert in das LMDR und steht im darauffolgenden Takt für die add-Instruktion im Registerfile zur Verfügung (Tabelle 19).

Tabelle 19: Behebung des Daten-Hasards durch die Pipeline-Kontrolle

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6	
`lw R1, 0(R2)`	IF	ID	EXE	MEM	WB		
`add R3, R7, R1`		IF	ID	stalled	EXE	MEM	WB
`sub R4, R1, R3`			IF	stalled	ID	EXE	MEM

4.2.2 Der Kontrollautomat (hazard_fsm.vhd)

Der Kontrollautomat, im folgenden `hazard_fsm` genannt, ist wesentlicher Bestandteil der Pipeline-Kontrolle. Er kann durch die Ausgabesignale stall_fetch, stall_decode und stall_exec die fortlaufende Pipelinebearbeitung anhalten, falls ein Problem in der Pipeline auftritt (Abbildung 4-4).

Abbildung 4-5 zeigt das Zustandsübergangsdiagramm der hazard_fsm. Es handelt sich hierbei um einen Moore-Automaten. Durch Zustand und Eingangssignale wird der Folgezustand bestimmt. Mit der fallenden Taktflanke von `phi2` geht die `hazard_fsm` in den neuen Zustand über. Damit sind für den nächsten Takt alle `stall`-Signale gesetzt.

Bei einem `RESET`-Signal sorgt die `hazard_fsm` für die Initialisierung der Pipeline. Dazu geht sie in den Zustand `reset_1` über, der die IF, ID und EXE-Phasen durch die `stall`-Signale anhält. Um den Befehlszähler auf „0“ zurückzusetzen werden zwei Takte benötigt, deshalb folgt ein weiterer Zustand `reset_2`.

Anschließend gelangt der Automat in den Arbeitszustand `fetch`. Nur bei

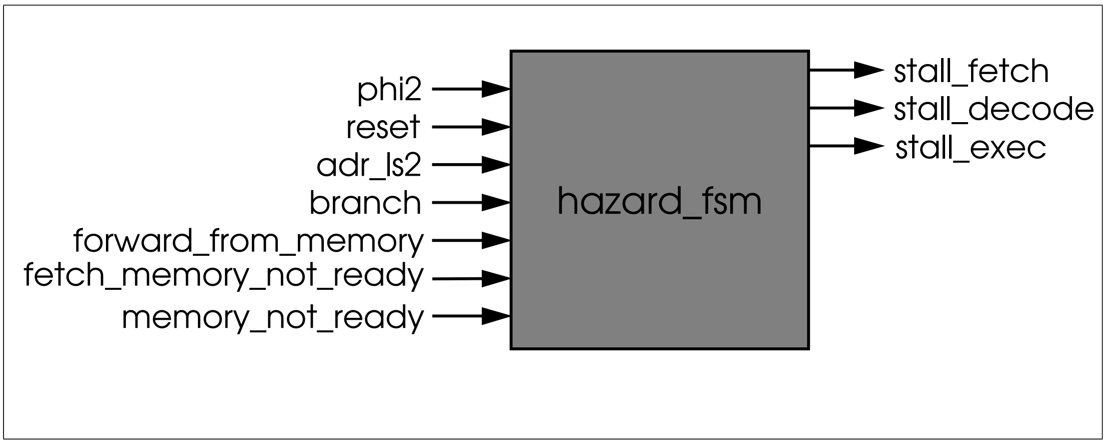

Abbildung 4-4: Der Kontrollautomat - hazard_fsm

einem Fehlersignal, bzw . bevorstehenden Hasard, geht er in einen der Behebungszustände über .

Durch das Fehlersignal `forward_from_memory` wechselt der Automat in den Zustand `stop_fetch_dec_exec`, welcher die Pipelinephasen IF , ID und EXE anhält. Nach einem T akt geht sie wieder in den Zustand `fetch` über , sofern das MEM-Register seine Daten lesen konnte, sonst wird das `mem_not_ready-Signal` gesetzt, und sie bleibt in diesem Zustand.

Das Fehlersignal `fetch_mem_not_ready` bringt den Automaten in den Zustand re_fetch, in dem die ID-Phase `NOP`-Befehle dekodiert, bis der Instruktionsspeicher die nächste Instruktion liefert. Dadurch wird das Fehlersignal zurückgesetzt, und Zustand `fetch` wieder eingenommen.

Im nächsten Kapitel wird auf das dritte Fehlersignal `branch` eingegangen, dabei werden die Aktionen der Pipeline-Kontrolle bei einem Sprung beschrieben.

4.3 Steuer-Hasards

Wenn der sequentielle Ablauf des Programmes durch einen Sprungbefehl geändert wird, muß die IF-Phase warten, bis der Befehlszähler die Adresse des Sprungziels beinhaltet. Diese Unterbrechung des Pipelineflusses bezeichnet man als Steuer -Hasard.

Bei einem unbedingten Sprung wird so lange mit Instruction Fetch und Instruction Decode gewartet, bis das Sprungziel berechnet und in den PC gelangt ist. Bei einem bedingten Sprung muß noch innerhalb des „Instruction Decode“ entschieden werden, ob ein V erzweigungs- oder Sprungbefehl vorliegt, und ob die V erzweigung ausgeführt wird oder nicht. Dadurch kann die Arbeit der Pipeline ununterbrochen fortgesetzt werden, falls keine Sprungbedingung vorliegt.

Um dies zu erreichen muß festgestellt werden, wo im Datenpfad sich der aktuelle Registerinhalt des über den Sprung entscheidenden Registers

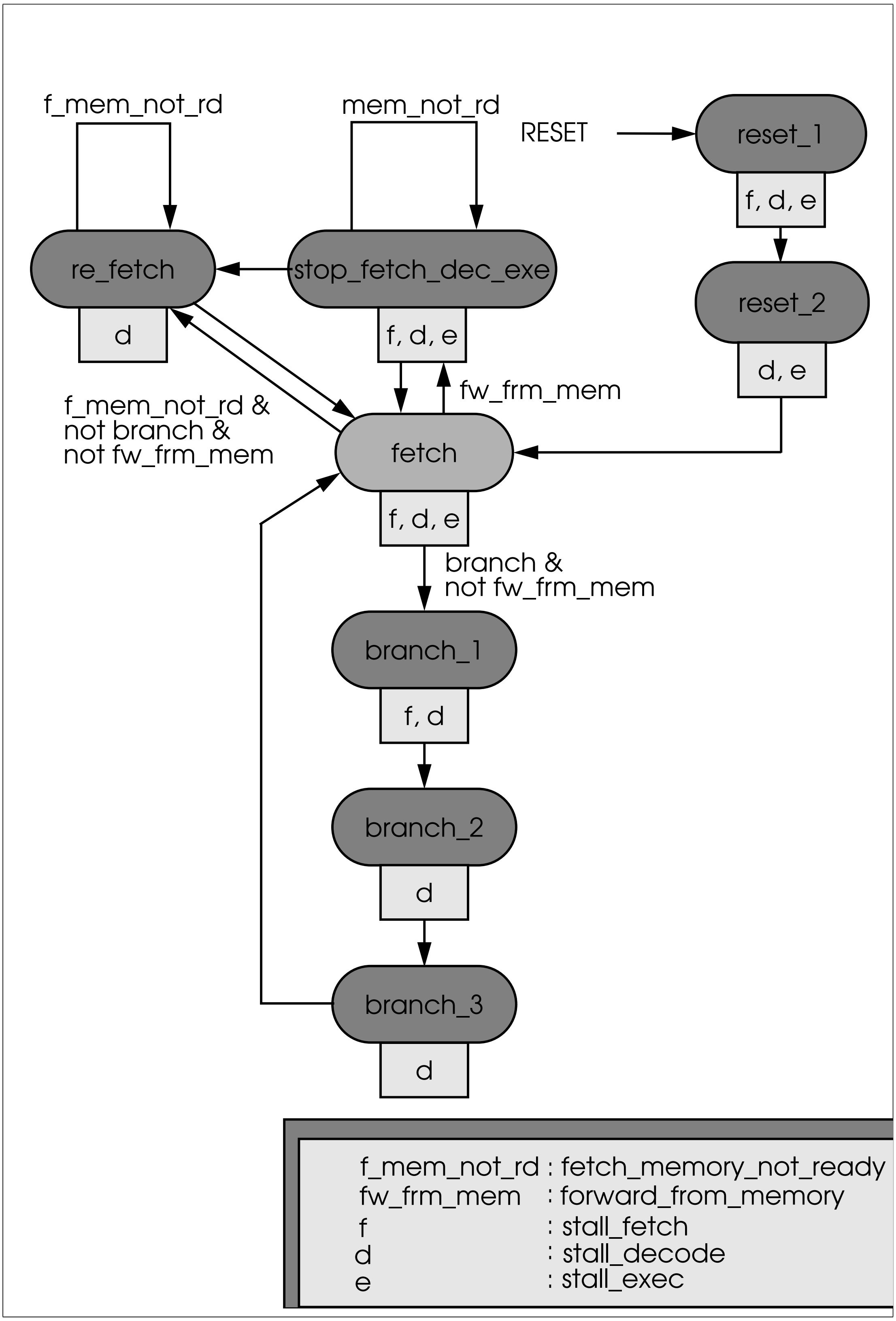

Abbildung 4-5: Die Zustände der hazard_fsm

befindet. Im folgenden Beispiel ist Register R1 entscheidend über die Ausführung des Sprungs:

```
sub R1, R2, R3
bnez R1, loop
```

Register R1 wird zwar erst im Write Back beschrieben, sein zukünftiger Inhalt ist jedoch schon in der Execution Stage dieses Befehls, d.h. im Verlauf des dritten Takts, bekannt (Tabelle 20).

Tabelle 20: Entscheidung über die Verzweigung in der ID-Phase

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6
`sub R1, R2, R3`	IF	ID	EXE	MEM	WB	
`bnez R1, loop`		IF	ID	EXE	MEM	WB
`add R3, R5, R7`			IF	ID	EXE	MEM

Um festzustellen wo sich der aktuelle Inhalt von Register R1 befindet, kann die Forwarding-Logik benutzt werden. Sie vergleicht das Quellregister des branch-Befehls mit den beiden zuletzt beschriebenen Zielregistern.

Bei einer Übereinstimmung mit einer der beiden Registernummern wird der Registerinhalt aus dem Datenpfad, bei keiner Übereinstimmung aus dem Registerfile, in eine „Null-Test"-Logik `zero_logic.vhd` geführt. Dies geschieht durch Ansteuerung des `branch_reg_mux` durch das von der Forwarding-Logik erzeugte Signal `branch_reg_sel`. Die Null-Test-Logik bestimmt, ob das Register einen Wert ungleich Null enthält und leitet ihr Ergebnis mit dem Signal `reg_value` zurück an Komponente ID (Abbildung 4-6).

Dort wird schließlich anhand der Art des dekodierten branch-Befehls (`beqz` oder `bnez`) darüber befunden, ob der Sprung ausgeführt wird oder nicht und das Signal `branch` entsprechend aktiviert.

Das Signal `branch` veranlaßt die MEM-Phase, das `pc_mux` umzuschalten, damit das Ergebnis vom `MAR` in den Befehlszähler geladen wird.

Außerdem ist `branch` ein Fehlersignal der Pipeline-Kontrolle, wodurch die `hazard_fsm` in den Zustand `branch_1` übergeht (Abbildung 4-5), in dem IF und ID `stall`-Signale erhalten. Die Sprungzieladresse gelangt erst in der MEM-Phase des `branch`-Befehls in den Befehlszähler. Daher müssen IF und ID einen weiteren Takt lang durch den Zustand `branch_2` angehalten werden. Im folgenden Zustand `branch_3` beginnt IF wieder zu arbeiten, ID erzeugt nach wie vor Kontrollsignale für `NOP`-Befehle. Schließlich kehrt die `hazard_fsm` wieder in den Arbeitszustand `fetch` zurück (Tabelle 21).

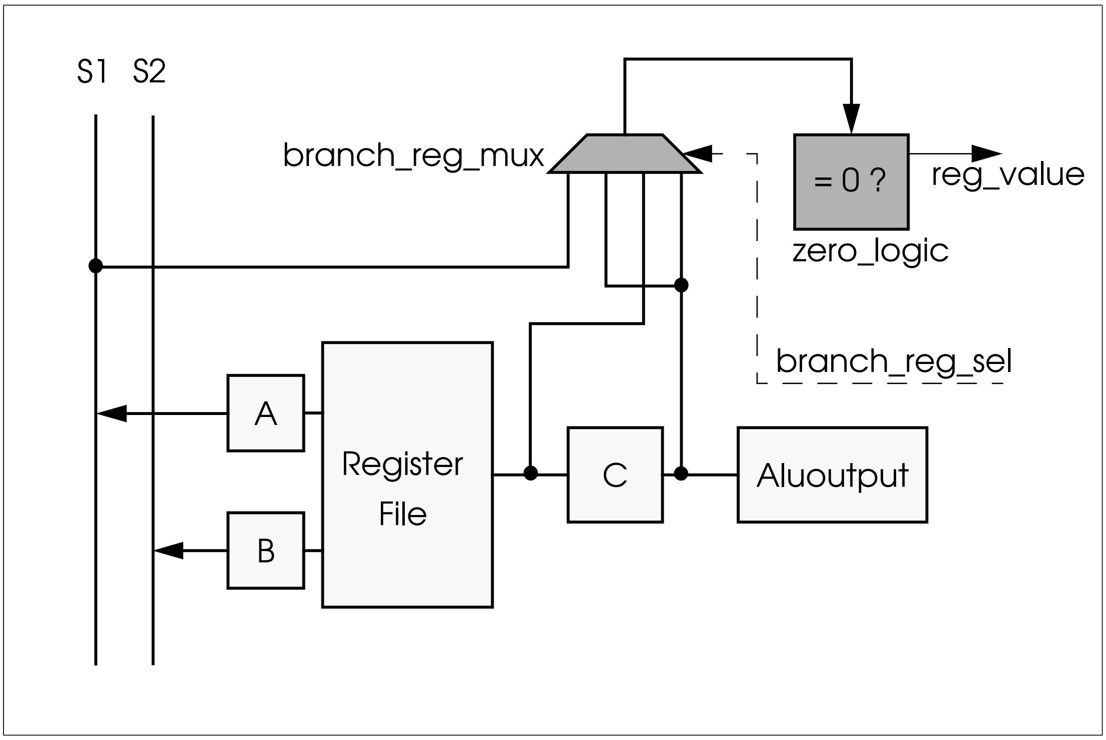

Abbildung 4-6: Auswertung des branch-Registers

Tabelle 21: Behandlung eines Steuer-Hasards

Befehl	Takt 1	Takt 2	Takt 3	Takt 4	Takt 5	Takt 6	Takt 7	Takt 8	Takt 9
`branch ziel`	IF	ID	EXE	MEM	WB				
`ziel`		IF	stall	stall	IF	ID	EXE	MEM	WB
`ziel + 1`			stall	stall	stall	IF	ID	EXE	MEM
`ziel + 2`				stall	stall	stall	IF	ID	EXE
`ziel + 3`					stall	stall	stall	IF	ID
`ziel + 4`						stall	stall	stall	IF

4.4 Interrupts und Exceptions

Aus zeitlichen Gründen konnte die sehr umfangreiche Bearbeitung der Hardware-Interrupt- und Ausnahmesignale (engl. „exceptions“) nicht mehr implementiert werden. Dennoch sollen im folgenden die Probleme und eventuelle Möglichkeiten zur Bearbeitung eines Interrupts angesprochen werden. Zunächst ist die Interruptbehandlung des DLXS zu betrachten:

Die Finite-State-Machine (FSM) des DLXS erwartet während dem „Instruction Fetch“ das READY -Signal um in den nächsten Zustand „Instruction Decode“ überzugehen. Sobald READY aktiv ist, werden `INTRPT`- und `HALT`-Leitung geprüft.

Liegt ein Interrupt vor , geht der Automat in den Zustand `intrpt_1` über , in welchem die ICR-Bits in die obere Hälfte des ICR-Registers geschoben werden. Im Folgezustand `intrpt_2` wird der Befehlszählerstand gesichert, und anschließend im Zustand `intrpt_3` der Inhalt des TBR-Registers in den Befehlszähler geschrieben. Dabei wird das „Supervisor“-Bit gesetzt.

Da der DLXS/P zumindest in der Lage ist, Softwareinterrupts auszuführen, die fast die gleichen Arbeitsschritte wie ein Hardwareinterrupt verlangen, kann ein Interrupt durch „V ortäuschen“ eines gelesenen `trap 0`-Befehls bearbeitet werden.

Dazu muß die `hazard_fsm` im Arbeitszustand `fetch` die INTRPT -Leitung prüfen. Liegt ein Interrupt vor , muß ein Folgezustand `intrpt_1` den `trap 0`-Befehl als Konstante über einen Multiplexer in das Instruktionsregister leiten. Im selben T akt darf IF den Befehlszähler nicht erhöhen, da durch den eingeschobenen Befehl der Folgebefehl überdeckt wird. Es muß also ein Signal `stall_PC` für IF geben. Anschließend kann in Arbeitszustand `fetch` zurückgekehrt werden, der auch Softwareinterrupts behandelt.

Eine Exception ist schwieriger zu behandeln. Im DLXS treten folgende Exceptions auf:

1. Überlauf des Befehlszählers
2. illegaler Opcode
3. illegale Register -Register ALU-Funktion
4. privilegierte Instruktion ohne Supervisor -Modus
5. Überlauf bei arithmetischer Instruktion
6. privilegierter Datenzugriff
7. Zugriff auf falsch ausgerichtete Speicheradresse bei Halbwort- und Wortzugriff
8. Sprungziel ist falsch ausgerichtete Speicheradresse

Die Exceptions müssen im DLXS/P von verschiedenen Pipelinephasen erkannt werden. Befehlszählerüberlauf von IF , illegale und privilegierte Instruktionen von ID, Überlauf der ALU von EXE, und die übrigen Exceptions von MEM.

Da es kein allgemeingültiges Schema zur Behandlung dieser Ausnahmesignale gibt, muß die `hazard_fsm` um eine V ielzahl von Zuständen erweitert werden. Folgende Entscheidungen müssen getroffen werden:

- Um welche Ausnahme handelt es sich, d.h. welches ICR-Bit muß gesetzt werden ?
- In welchem Register (`PC`, `PC_1`, `PC_2`) befindet sich die Adresse des Folgebefehls ?
- Wieviele T akte benötigen die in der Pipeline liegenden Befehle um beendet zu werden ?
- Auf welche Ausnahme wird reagiert, wenn innerhalb eines T aktes mehrere gleichzeitig auftreten ?

Hinzu kommt, daß sich bis zu fünf Befehle gleichzeitig in der Pipeline befinden. W enn ein Befehl eine Ausnahme beim Dekodieren erzeugt, wird ein früher geholter Befehl erst später eine Ausnahme bei einem Speicherzugriff erzeugen. Deshalb muß bei einer Behandlung einer Ausnahme in jedem Zustand geprüft werden, ob weitere vorliegen.

5 Validierung

Um das Prozessormodell zu validieren diente der Assembler `dlx_asm` von Pirouz, Bazargan und Sabet, der von Martin Gumm überarbeitet wurde [MaGu90].

Der Assembler übersetzt Assembler -Code für den DLXS/P in VHDL-Code. Dieser Code stellt die V erhaltensspezifikation eines ROM Bausteins in den das Assembler -Programm gebrannt wurde, dar .

Der Baustein muß als User -ROM in die Systemkonfiguration `system-config_rtl2.vhd` eingetragen werden. Mit einem Vhdl-Debugger kann anschließend die Bearbeitung des Programms welches sich auf dem ROM befindet, verfolgt werden [AsW e95].

Zum T est des DLX/P wurden mehrere kleine Assemblerprogramme erstellt, die insbesondere Daten-Hasards und Steuer -Hasards erzeugen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Ausgehend von dem Modell des DLXS wurde im Rahmen dieser Studienarbeit eine 5-stufige Pipeline auf RTL-Ebene in VHDL implementiert. Das neue Modell erhielt die Bezeichnung DLXS/P .

Dazu mußte der bestehende Datenpfad durch zahlreiche Komponenten erweitert und verändert werden. Diese notwendigen Änderungen wurden mit Hilfe einer umfassenden Analyse des Befehlssatzes des DLXS erkannt und vorgenommen. Dabei wurde eine im DLXS implementierte Erweiterung (Register -Register -Speicherbefehle) des DLX-Befehlssatzes nach Hennesy und Patterson nicht berücksichtigt, damit das bestehende Registerfile weiterverwendet werden konnte.

Die neu hinzugekommenen Ein-Ausgabeports des DLXS/P wurden aus Kompatiblitätsgründen durch eine Datenweiche auf die alten Schnittstellen des DLXS geleitet. Dadurch hat man die Möglichkeit, das Modell in der bestehenden T estbench zu validieren. Außerdem müssen so bei einer Erweiterung auf getrennten Daten und Instruktionsspeicher keine Änderungen am Prozessormodell vorgenommen werden.

6.2 Ausblick

Um vollständige Kompatiblität des DLXS/P zum DLXS zu erreichen, müssen noch weitere Änderungen an Steuerung und Datenpfad vorgenommen werden. Folgende Schwerpunkte könnten Thema weiterer Arbeiten sein:

6.2.1 Exception-Handling

Es muß ein allgemeines, erweiterbares Schema gefunden werden, auf die Ausnahmesignale verschiedener Pipelinephasen zu reagieren. Die in Kapitel 4.4 angesprochenen Ausnahmen müssen beabeitet werden, damit der DLXS/P korrekt arbeitet. Es ist zu prüfen ob sich die `hazard_fsm` für die Ausnahmebehandlung eignet, da sehr viele neue Zustände eingeführt werden müssen.

6.2.2 Register-Register-ALU-Befehle

Diese im DLXS eingeführten Befehle ermöglichen einen Speicherzugriff mit Registerindex zur Berechnung der Speicheradresse. Zur Ausführung muß während der EXE-Phase auf drei verschiedene Register zugegriffen werden. Das erzwingt eine Erweiterung des Registerfiles auf drei Ausgabeports, die mit zusätzlichen Kontrolleitungen anzusteuen sind. Dabei müssen auch Änderungen Forwarding-Hardware vorgenommen werden. Zur Dekodierung dieser Befehle und Ermittlung der Quell- und Zielregisternummern sei auf die Mechanismen im DLXS hingewiesen.

6.2.3 Instruction- / Data-Only Cache

Mit einem getrennten Cache-Speicher für Instruktionen und Daten kann die Datenweiche weggelassen werden. Dadurch erhöht sich die Durchgangsleistung der Pipeline bei häufigen Speicherzugriffen. Nur bei einem „Cache-Miss“ muß auf den nächsten Befehl gewartet werden. Diese Ausnahmesituation kann nur durch Erweiterung des Exception-Handling erfaßt und bearbeitet werden.

6.2.4 Floating-Point-Unit

Eine Erweiterung des DLXS-Befehlssatzes auf Fließkommarithmetik fordert weitere Änderungen des Datenpfads. Zusätzliche Fließkommaeinheiten werden benötigt, die ihre arithmetischen Operationen wahrscheinlich nicht innerhalb eines T akts durchführen können. Die nachfolgenden Befehle müssen so lange verzögert werden, bis die Pipeline-Kontrolle über die erfolgreiche Ausführung der mehrere Zyklen dauernden Fließkommaberechnung benachrichtigt wurde.

Die zusätzlichen arithmetischen Einheiten erzeugen eigene Ausnahmesignale, die ebenfalls nur durch eine Erweiterung des Exception-Handlings verarbeitet werden können.