Versuchsvorbereitung Prozessor

Grundlagen des DLXJ-RISC-Prozessors

Andreas Reinsch

Friedrich-Schiller-Universität Jena
Institut für Informatik
Lehrstuhl für Advanced Computing

2. Juni 2020

Inhaltsverzeichnis

1. Einleitung

Die DLX[1]-Prozessorarchitektur wurde von Hennessy und Patterson [3, 4, 5] spezifiziert. Aufgrund seiner Einfachheit ist der DLX-Prozessor gut geeignet, um die Funktionsweise und die wesentlichen Elemente von RISC-Prozessoren (RISC - Reduced Instruction Set Computer) leicht verständlich darzustellen, mit VHDL zu modellieren und in Hardware umzusetzen. Zu Lehrzwecken wurde die DLX-Architektur mehrfach mit VHDL synthesefähig spezifiziert [1, 2], jedoch nicht in Hardware realisiert. Die Beschreibung nach [2] verzichtet auf die Implementierung von Gleitkommaarithmetik und vorzeichenlosen arithmetischen Befehlen, sieht aber gegenüber der Spezifikation von Hennessy und Patterson eine Ausnahme- und Unterbrechungsbehandlung vor.

Die „DLXJ“ -Prozessor-Architektur (der Buchstabe „J“ steht für „Jenaer“ Version) ist mit dem Ziel entwickelt worden, einerseits die wesentlichen Merkmale der DLX-Architektur nach Hennessy und Patterson zu zeigen, andererseits durch konsequente Beschränkung des Befehlssatzes zu einem sehr einfachen und damit leicht verständlichen Prozessor für Lehrzwecke zu gelangen. Alle RISC-typischen Architekturmerkmale wurden beibehalten. Der Befehlssatz wurde auf nur 7 Befehle reduziert, jedoch so, daß die drei Befehlsklassen Datentransport, arithmetisch-logische Operationen und Steuerung vorhanden sind. Gleitkomma-Operationen sowie eine Ausnahme- und Unterbrechungsbehandlung wurden nicht implementiert. Der Befehlssatz enthält Befehle aller drei Befehlsformate der DLX-Architektur. Der Entwurf wurde auf der algorithmischen Ebene und auf der Register-Transfer-Ebene in der Hardwarebeschreibungssprache VHDL durchgeführt. Einige Baugruppen mußten den Anforderungen der Schaltkreis-Architektur speziell angepaßt werden. Die Spezifikation der Register-Transfer-Ebene ist synthesefähig und gestattet die Implementierung in einen rekonfigurierbaren integrierten Schaltkreis (*Field Programmable Gate Array*, FPGA). Zusätzlich zum DLXJ-Prozessorkern wurden auf diesem Schaltkreis, der Teil eines einfachen Experimentalsystems ist, auch ein Arbeitsspeicher, ein Displayinterface sowie ein Interface zu einem Mikrocontroller realisiert. Das Experimentalsystem enthält weiterhin eine Displayeinheit und einen im System programmierbaren Mikrocontroller. Der Mikrocontroller dient als Bindeglied zwischen dem FPGA und einem über eine serielle Schnittstelle angeschlossenen Hostrechner. Die Programmentwicklung wird durch einen Assembler unterstützt. Die Arbeitsweise des Prozessors und der Ablauf einfacher kleiner Programme kann entweder am Digitalsimulator oder direkt an der Hardware mit Hilfe eines Debuggers untersucht werden. Die Spezifikation in VHDL ermöglicht auf einfache Weise die Erweiterung der Funktionalität des Prozessors (die Implementierung weiterer Befehle) oder die Modifikation der ebenfalls auf dem FPGA vorhandenen Prozessorperipherie (Erweiterung der Debug-Möglichkeiten).

[1] Die Bezeichnung „DLX“ (ausgesprochen „Deluxe“) steht für die römischen Ziffern DLX und ergibt sich als Mittelwert einer Zahl von experimentellen und kommerziellen Maschinen, die in ihrer Philosophie der DLX sehr ähnlich sind: (AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260)/13= $560 = DLX$ (nach [4]).

2. Das Programmiermodell

Die Architektur des DLXJ-Prozessors wurde aus dem von Gumm [2] entwickelten Modell abgeleitet. Der Befehlsumfang wurde stark reduziert.

- Die Architektur enthält 32 32-Bit-Universalregister (*General-Purpose Register, GPR*). Der Wert von Register R0 ist immer null.
- Der Speicher ist wortadressiert mit 32-Bit-Adressen (4 GByte Adreßraum). Es wird im Modus „Big Endian“ gearbeitet. Demnach ist das niederwertigste Datenbit im DLXJ-Wort der Stelle MSB (*Most Significant Bit*) zugeordnet und das höchstwertige Bit der Stelle LSB (*Least Significant Bit*). Alle Speicherzugriffe müssen ausgerichtet (aligned), also bei Wortzugriffen durch 4 teilbar sein.
- Alle Befehle sind 32 Bit lang.

Der Prozessor hat eine einfache Lade/Speicher-Architektur. In solchen Architekturen erfolgen alle arithmetischen und logischen Operationen zwischen den GP-Registern, Speicherzugriffe werden ausschließlich über GP-Register realisiert.

2.1. Befehlsklassen

Die Tabelle 1 zeigt die Zuordnung der vorhandenen 7 Befehle zu drei Befehlsklassen, den zugehörigen Assemblerbefehl, die Wirkung des Befehls, das Befehlsformat und den Befehlscode.

Befehl	Assemblerbefehl	Bedeutung	Typ	Opcode rr_func
		Datentransport		
load word	$LW.I \quad Rd, I(Rs)$	$Rd \leftarrow M(Rs + I)$ [a]	I	000100
store word	$SW.I \quad I(Rs), Rd$	$M(Rs + I) \leftarrow Rd$ [a]	I	001000
		Arithmetisch-logisch		
sub-traction	$SUB \quad Rd, Rs1, Rs2$	$Rd \leftarrow Rs1 - Rs2$	R	000000 000110
set if lower than	$SLT \quad Rd, Rs1, Rs2$	$Rs1 < Rs2$: $Rd \leftarrow hex00000001$ else: $Rd \leftarrow hex00000000$	R	000000 010100
no operation	NOP		R	000000 000000
		Steuerung		
branch if equal zero	$BEQZ Rs1, Label$	$Rs1 = 0$: $PC \leftarrow PC + 4 + Label$ [b] $Rs1 \neq 0$: $PC \leftarrow PC + 4$	I	010000
jump	$J \quad Label$	$PC \leftarrow PC + 4 + Label$ [c]	J	001100

[a] I: vorzeichenerweiterter 16-bit Offset, wortausgerichtete Adresse: letzte zwei Bit = 00
[b] Label: vorzeichenerweiterter 16-bit Offset
[c] Label: vorzeichenerweiterter 26-bit Offset

Tabelle 1: Befehlsklassen des DLXJ

Datentransport-Operationen dienen dem Datentransport zwischen den GP-Registern und dem Speicher. Jedes der GPR kann geladen oder gespeichert werden. Das Laden von R0 hat keine Wirkung. DLX-Speicherzugriffe können wort- halbwort- oder byteweise erfolgen. Die DLXJ-Implementierung ist auf zwei wortweise Zugriffe beschränkt.

Arithmetisch logische Operationen sowie Verschiebe- und Testoperationen bilden eine weitere Befehlsklasse. Der DLX-Befehlssatz enthält neben Operationen zu den Grundrechenarten, die logischen Operationen, logische und arithmetische Schiebeoperationen sowie Vergleichsoperationen.

In den DLXJ-Befehlssatz wurden ein Subtraktionsbefehl, eine Vergleichsoperation und die Nulloperation (no operation) aufgenommen.

Die Steuerung des Programmablaufs wird durch Sprung- und Verzweigeoperationen realisiert. Spünge können beim DLX-Prozessor als einfacher Sprung oder bei Prozeduraufrufen als Sprung mit Rückkehrverbindung ausgeführt werden. Alle Verzweigungen sind bedingt. Sprünge sind an keine Bedingung geknüpft.

Beim DLXJ-Prozessor wird der Programmablauf durch einen Sprungbefehl und einen Verzweigebefehl gesteuert.

Gleitkomma-Operationen beinhaltet der Befehlssatz des DLX-Prozessors für die Grundrechenarten und für Vergleiche sowohl mit einfacher (32 Bit) als auch mit doppelter (64 Bit) Genauigkeit.

Im Befehlssatz des DLXJ-Prozessors sind Gleitkomma-Operationen nicht vorgesehen.

2.2. Befehlsformate

Das DLX-Befehlslayout wurde unverändert übernommen. Abbildung 1 zeigt die drei verschiedenen Befehlsformate und die Zuordnung der Befehle.

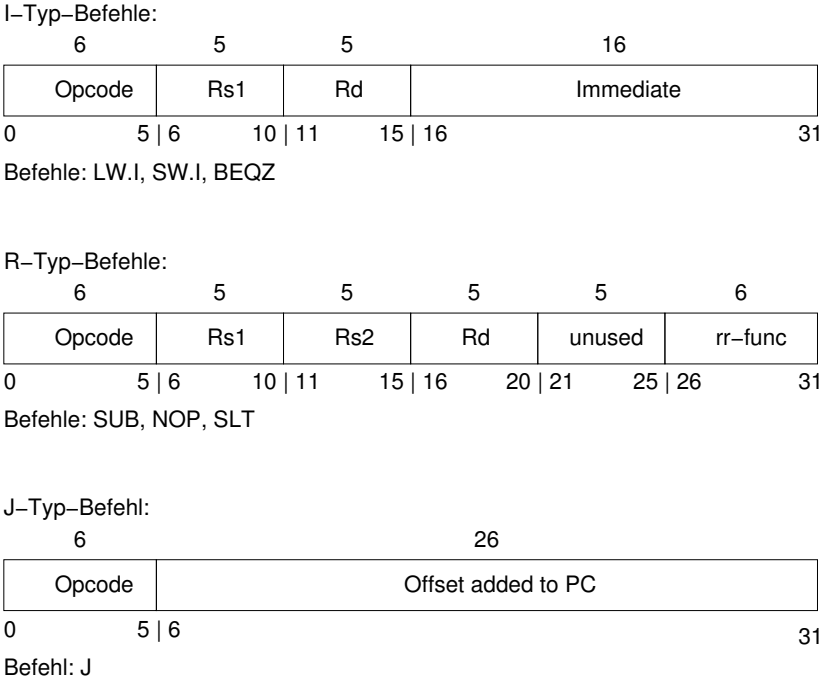

Abbildung 1: Die drei Befehlsformate des DLXJ

Der primäre Operationscode (Opcode) ist bei allen drei Formaten einheitlich durch die Bits $0 \ldots 5$ festgelegt. Die Verwendung der verbleibenden 26 Bit ist vom Befehlstyp abhängig:

Das R-Typ Format gestattet die Adressierung von zwei Quellregistern *Rs1, Rs2* (Source Register) und einem Zielregister *Rd* (Destination Register) mit jeweils 5 Bit. Die auszuführende Register-Register-ALU Operation (RRA) wird durch das Bitfeld rr_func (DLX: 11 bit) bestimmt: $Rd \leftarrow Rs1$ rr_func $Rs2$.

Die R-Typ-Befehle des DLXJ-Prozessor sind die Subtraktion, eine Vergleichsoperation und die Nulloperation. Bei der Subtraktion wird der Inhalt des Quellregisters $Rs2$ vorzeichenrichtig vom Quellregister $Rs1$ subtrahiert und das Ergebnis im Zielregister Rd abgelegt ($Rd \leftarrow Rs1 - Rs2$). Die Vergleichsoperation testet die Bedingung $Rs1 < Rs2$, ist sie erfüllt, wird das Zielregister auf 1, andernfalls auf 0 gesetzt.

Das I-Typ Format stellt je 5 Bit für die Adressierung eines Quellregisters $Rs1$ und eines Zielregisters Rd zur Verfügung. Die Verwendung der verbleibenden 16 Bit Immediate ist befehlsabhängig.

Die Datentransport-Operationen und der Verzweigebefehl des DLXJ-Prozessors sind I-Typ-Befehle. Die Speicheradresse der Transportoperationen wird bestimmt, indem der 16 Bit Immediate-Wert vorzeichenerweitert zum Quellregister addiert wird. Das Zielregister enthält das geladene oder das zu speichernde Datenwort. Der Verzweigebefehl testet das Quellregister und verzweigt, falls es Null ist. Die Verzweigezieladresse wird durch vorzeichenerweiterte Addition des Immediate-Werts zum Befehlszähler berechnet.

Das J-Typ Format wird ausschließlich von Sprungbefehlen genutzt. Die Zieladresse bei Ausführung des Sprungbefehls ergibt sich aus einem zum Befehlszähler addierten 26-Bit vorzeichenerweiterten Offset (Offset added to PC) zu der dem Sprungbefehl folgenden Adresse.

Der DLXJ-Prozessor verfügt über einen Sprungbefehl. Prozeduraufrufe mit Rückkehrverbindung sind damit nicht möglich.

3. Die Struktur des DLXJ-Prozessors

Der DLXJ-Prozessor wurde als Einchiprechner realisiert. Auf einem einzigen FPGA-Schaltkreis sind zusätzlich zum Prozessorkern auch der Arbeitsspeicher und zwei Interfaces integriert. Das Blockschaltbild (Abbildung 2) zeigt die vier Baugruppen. Damit enthält das FPGA alle wesentlichen Funktionseinheiten eines Rechners. Der FPGA-Schaltkreis ist Teil eines Experimentalsystems, das für die Implementierung des Prozessors außerdem einen PC als Hostrechner, einen Mikrocontroller, ein LCD-Display mit 2x16 Zeichen und einen Quarz-Taktgenerator nutzt (Abbildung 3).

Der Mikrocontroller ist einerseits über eine USB-Verbindung mit dem Hostrechner und andererseits über mehrere 8-Bit-Parallelports mit dem FPGA verbunden. Auf diesem Weg werden

- Programme zum DLXJ-Prozessor übertragen,

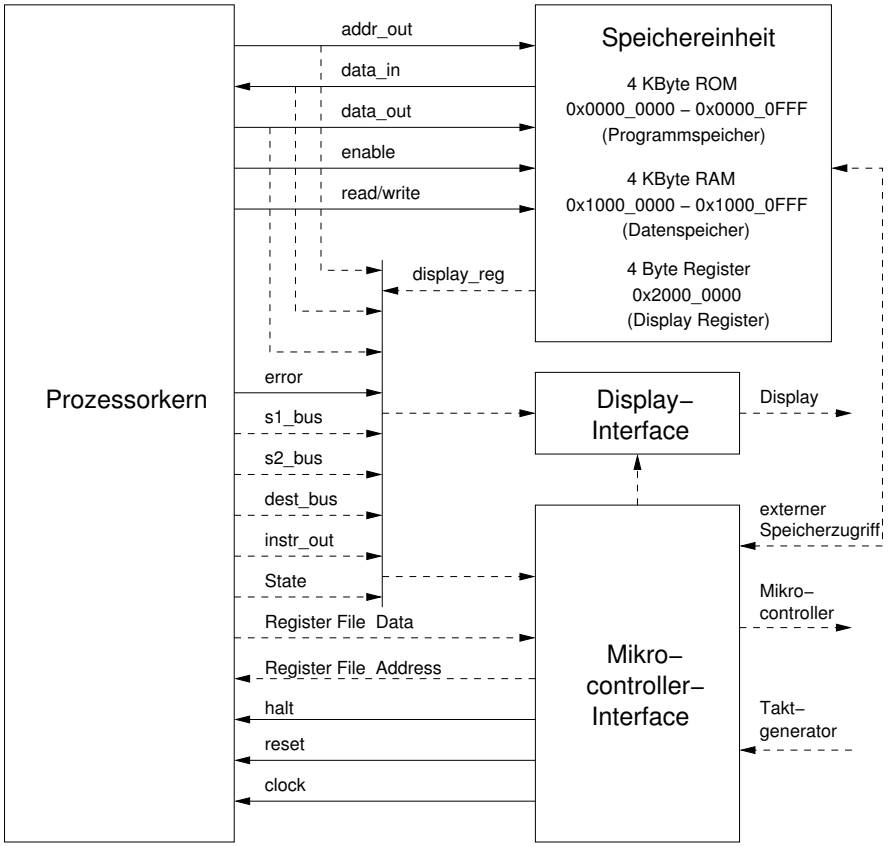

Abbildung 2: Die Struktur des DLXJ-Prozessors

- Daten zwischen dem DLXJ-Prozessor und dem Hostrechner ausgetauscht und
- Steuersignale an den DLXJ-Prozessor gesendet.

Mit einer zusätzlichen USB-Verbindung zwischen dem FPGA-Board und dem Host-PC wird das FPGA konfiguriert. Die Konfigurationsdaten des FPGA bleiben nur solange erhalten, solange das Experimentalsystem mit Spannung versorgt wird. Nach jeder Betriebsunterbrechung muß das FPGA neu konfiguriert werden, um so die Funktionalität des DLXJ-Prozessors wieder herzustellen.

Weiterhin besteht eine Netzwerkverbindung zwischen dem Host-PC und einer Workstation. Alle für den Hardwareentwurf benötigten Programme werden von der Workstation zur Verfügung gestellt.

3.1. Der Prozessorkern

Der Prozessorkern ist über einen Adreßbus ($addr\_out$, 32 Bit), zwei Datenbusse ($data\_out$, $data\_in$, je 32 Bit) und zwei Steuersignale ($enable$, $read/write$) mit der Speichereinheit verbunden.

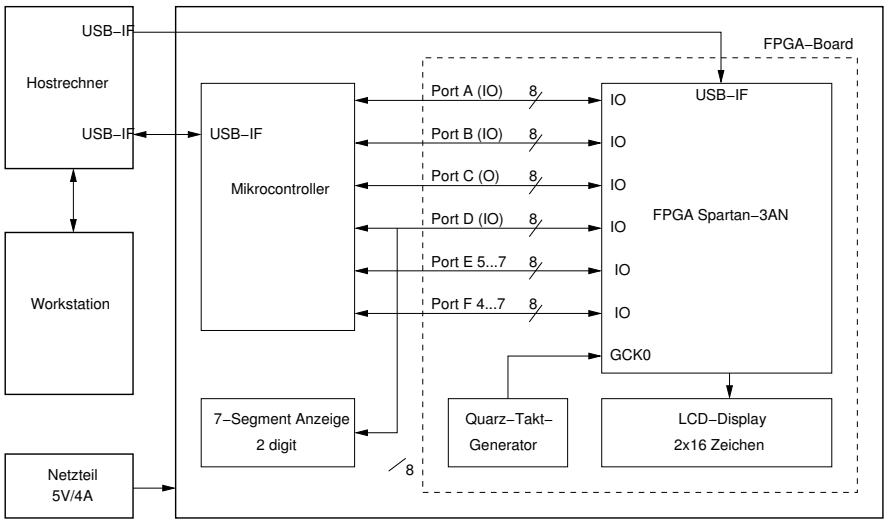

Abbildung 3: Das Experimentalsystem

Das Signal $reset$ versetzt den Prozessor in einen definierten Anfangszustand, d.h. der Programmzähler, das Speicheradreßregister und das Speicherdatenregister werden zurückgesetzt. Folglich wird die Programmausführung an der Adresse 0 fortgesetzt, vorausgesetzt das $halt$-Signal wurde zurückgesetzt.

Das Signal $halt$ löst eine Unterbrechung der Programmabarbeitung aus. Der gerade ausgeführte Befehl wird beendet und die Programmabarbeitung wird mit dem folgenden Befehl erst dann fortgesetzt, wenn das $halt$-Signal nicht mehr aktiv ist.

Das Signal $error$ zeigt an, daß der Prozessor einen nicht decodierbaren Befehl erhalten hat und die Programmabarbeitung abgebrochen wurde oder daß Implementierungsfehler im Schaltwerk (Abschnitt 4.4) vorliegen.

Der Zweiphasentakt $clock$ besteht aus zwei periodischen Signalen, deren Impulsdauer und Phasenlage so zueinander angeordnet sind, daß niemals beide Signale gleichzeitig 1-Pegel annehmen. Gegenüber einem Einphasentakt kann der Prozessor so einen insgesamt höheren Datendurchsatz erreichen.

Im Abschnitt 4 wird die Funktionsweise des Prozessorkerns ausführlich beschrieben.

Anmerkung: Alle gestrichelt gezeichneten Signale der Abbildungen 3, 4 und 6 sind für die Funktion des Prozessors nicht erforderlich. Sie werden benötigt, um prozessor-interne Abläufe experimentellen Untersuchungen zugänglich zu machen.

3.2. Die Speichereinheit

Die Speichereinheit erhält vom Prozessorkern ihre Adreßinformationen ($addr\_out$). Die Daten gelangen über je einen Bus vom Prozessor zum Speicher ($data\_out$) und vom Speicher zum Prozessor($data\_in$). (vgl. Abschnitt 4.1) Die Speicherfreigabe geschieht durch das Signal $enable$, das Signal $read/write$ entscheidet darüber, ob Daten vom Speicher gelesen

oder in den Speicher geschrieben werden (vgl. Abschnitt 4.2).

Die Speichereinheit beinhaltet einen Adreßdecoder und drei verschiedene Speicherblöcke: einen Datenspeicher, einen Programmspeicher und ein Register. Der Adreßdecoder ordnet den drei Speichereinheiten jeweils einen Adreßbereich zu. Die Adreßbereiche können der Abbildung 2 entnommen werden.

Der Programmspeicher und der Datenspeicher mit je 4 KByte Speicherkapazität (d.h. je 1K Worte) sind Zweitor-Speicher mit wahlfreiem Zugriff (Dual Port RAM DP-RAM). Ein Port jedes Speichers ist mit dem Prozessorkern verbunden. Der Programmspeicher wird prozessorseitig als ROM (Read Only Memory) betrieben, kann also vom Prozessorkern nur gelesen werden. Damit wird verhindert, daß die Programmdaten durch ein fehlerhaftes Programm überschrieben werden können. Der Datenspeicher wird prozessorseitig als RAM (Random Access Memory) betrieben. Hier bestimmt der Zustand des Signals $read/write$, ob Daten gelesen oder geschrieben werden. Das zweite Port jedes Dual Port RAM ist mit dem Mikrocontroller-Interface verbunden. Das Interface hat Schreib- und Lesezugriff auf beide Speicherblöcke, so daß über diese Verbindung Programme und Daten mit dem Hostrechner ausgetauscht werden können.

Das Register (Display Register) dient der Ergebnisanzeige. Es ist eine einzelne Speicherzelle (ebenfalls 32 bit) im Adreßraum des Prozessors, auf die nur geschrieben werden kann. Der Wert dieses Registers kann über das Display-Interface auf dem LCD-Display angezeigt werden.

3.3. Das Display-Interface

Das Display-Interface wird für die direkte Anzeige von Daten und von Steuerinformationen des Prozessors verwendet. Mit einem Multiplexer (8x32 auf 1x32) können 8 verschiedene 32 Bit-Datenquellen ($addr\_out$, $data\_in$, $data\_out$, $display\_reg$, $s1\_bus$, $s2\_bus$, $dest\_bus$, $instr\_out$) auf das LCD-Display geschaltet werden (vgl. Abschnitt 4.1). Die Auswahl des Multiplexer-Eingangs, der zur Anzeige gelangen soll, wird über das Mikrokontroller-Interface vom Hostrechner aus festgelegt. Der momentane Zustand der Steuerung (State) und das Signal $error$ des Prozessorkerns werden ebenfalls auf dem LCD-Display angezeigt.

3.4. Das Mikrocontroller-Interface

Das Mikrocontroller-Interface verbindet den DLXJ-Prozessor mit dem Mikrocontroller des Experimentalsystems. Hierzu werden die Parallelports des Controllers genutzt. So erfolgt der Datenaustausch mit der Speichereinheit, die Steuerung des Prozessorkerns, die Erzeugung von Einzeltakt-Impulsen und die Übertragung von Debuginformationen (Registerinhalte, Speicherinhalte, alle prozessorinternen Buszustände und der aktuelle Zustand der Steuerung) über drei 8 Bit Ports (A, B und C; Abbildung 3).

Im wesentlichen besteht das Mikrocontroller-Interface aus mehreren Multiplexern und Registern. Die Multiplexer schalten die jeweils zu übertragende Datenquelle oder Datensenke auf die Ports des Controllers. In den Registern werden die Steuerinformationen für die Multiplexer und den Prozessorkern gespeichert.

4. Die interne Struktur des Prozessorkerns

Der Prozessorkern besteht aus Datenpfad und Steuerung (Abbildung 4). Der Datenpfad ent-

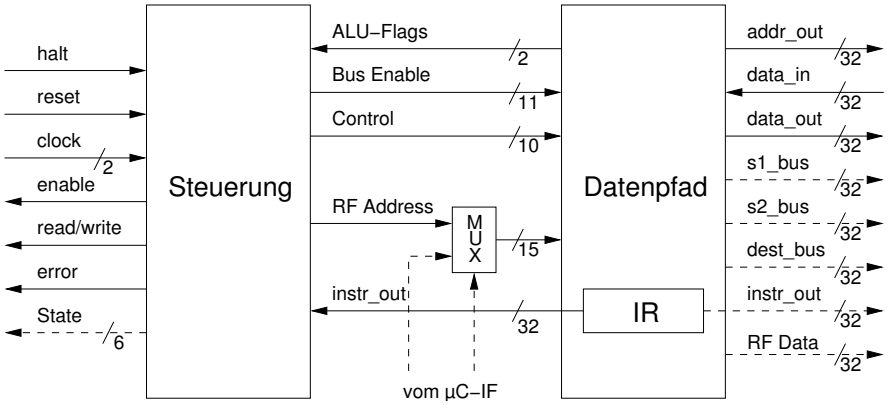

Abbildung 4: Der Prozessorkern

hält mehrere Register. Das Befehlsregister (Instruction Register - IR) ist außerdem mit der Steuerung verbunden. Es wird zu Beginn eines jeden Befehls mit dem aktuellen Befehlswort geladen. Weitere wesentliche Bestandteile des Datenpfads sind die ALU und ein Bussystem. Der Datenpfad führt alle Operationen an den Daten aus, liefert der Steuerung Statusinformationen (ALU-Flags) und das aktuelle Befehlswort.

Die Steuerung besteht neben dem Befehlsregister aus drei Decodern und einem Schaltwerk. Das Schaltwerk wird auch als binärer Automat oder als Finite State Machine (FSM) bezeichnet. Aus dem Befehlswort und den Statusinformationen werden Signale für die Ansteuerung des Datenpfads ($Bus\ Enable$, $Control$, $RF\ Address$), für Zugriffe auf die Speichereinheit ($enable$, $read/write$) und ein Signal für die Fehleranzeige ($error$) erzeugt. Die Steuerung wird mit einem Zweiphasentakt ($clock$) betrieben.

4.1. Der Datenpfad

Der Datenpfad ist in der Abbildung 5 dargestellt.

Die GPR0 ... GPR31 werden zu einem Block zusammengefaßt und als Registerdatei (Registerfile - RF) bezeichnet. Der Datenpfad hat drei interne Busse: die beiden Quellbusse S1, S2 ($s1\_bus$, $s2\_bus$) und den Zielbus Dest ($dest\_bus$) Die grundlegende Operation des Datenpfades ist das Lesen von Operanden aus dem RF über die Busse S1 und S2, deren Verarbeitung in der ALU und das Zurückspeichern des Ergebnisses in das RF über den Bus Dest. Da das RF nicht in jedem Taktzyklus gelesen und geschrieben werden muß, wird diese Folge unterteilt. Deshalb gibt es im Datenpfad die Register A und B an den zwei Ausgängen und ein Register C am Eingang des RF. Weiterhin sind zwei Latches L1, L2 an den Eingängen der ALU vorgesehen. Die befehlsspezifischen Aufgaben des RF als Quell- und Zielregister wurden bereits in Abschnitt 2 beschrieben. Die Aufgaben der mit dem RF in unmittelbarem Zusammenhang stehenden Register A, B und C werden im Abschnitt 4.3 erklärt.

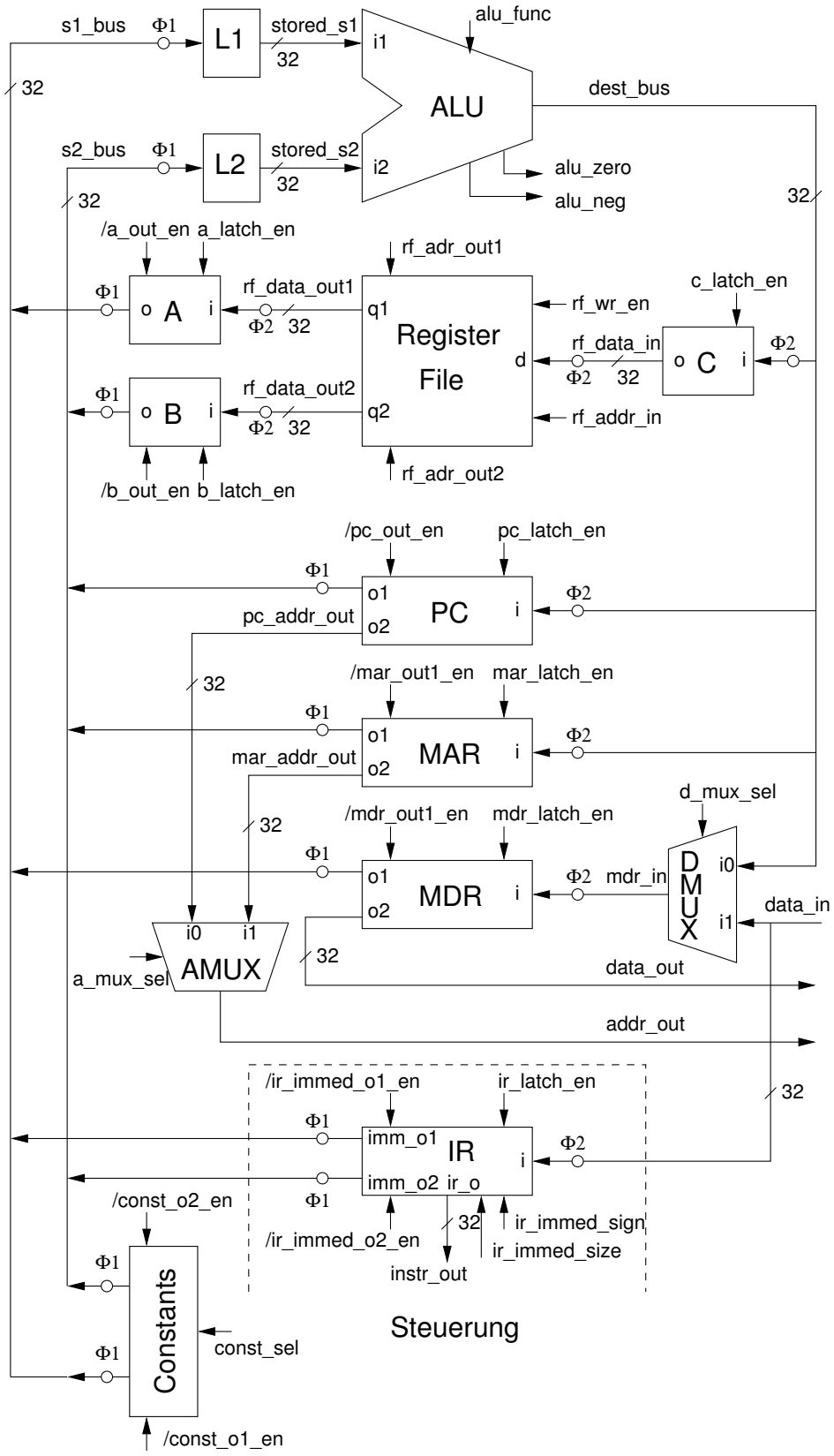

Abbildung 5: Der Datenpfad des DLXJ

Zu beachten ist, daß der einzige Weg von den Quellbussen zum Zielbus über die ALU führt. In der Tabelle 2 sind alle implementierten ALU-Operation aufgelistet.

Die Realisierung der ALU durch eine synthetisierbare VHDL-Beschreibung zeigt der Abschnitt A.1. Für die arithmetischen Operationen und für die Schiebeoperationen werden Prozeduren aus dem Package $stdl1164\_vector\_arithmetic$ [2] verwendet.

Bezeichnung	ALU-Operation
alu_pass_s1 [a]	Dest := S1
alu_pass_s2 [a]	Dest := S2
alu_s1_add_s2 [a,b,c]	Dest := S1 + S2
alu_s1_sub_s2 [a,c]	Dest := S1 − S2
alu_s1_and_s2	Dest := S1 AND S2
alu_s1_or_s2	Dest := S1 OR S2
alu_sll_s1_s2 [d]	Dest := S1 log. links um S2 geschoben
alu_srl_s1_s2 [d]	Dest := S1 log. rechts um S2 geschoben
alu_dcare	Dest ist beliebig

[a] für die Befehle nach Tab.1 erforderlich
[b] bei den Befehlen nach Tab.1 nur für befehlsinterne Operationen
[c] Zweierkomplement
[d] nur die 5 niederwertigsten Bits von S2 werden verwendet

Tabelle 2: ALU-Operationen

Der Befehlszähler (Program Counter - PC) zeigt auf den nächsten auszuführenden Befehl. Der Inhalt dieser Speicherzelle wird zu Beginn eines jeden Befehls in das IR geladen. Das IR enthält somit immer den aktuellen Befehl.

Bei Datentransport-Operationen werden die Daten im Speicher-Daten-Register (Memory Data Register - MDR) zwischengespeichert. Der Multiplexer DMUX schaltet im Verlauf einer Schreiboperation den internen Bus ($dest\_bus$) und im Verlauf einer Leseoperation den Speicher-Datenbus ($data\_in$) auf den Eingang des MDR. Die für den Speicherzugriff erforderliche Adresse wird im Speicher-Adreß-Register (Memory Adress Register - MAR) abgelegt. In Abhängigkeit davon, ob das IR mit dem Folgebefehl geladen werden soll oder ob eine Datentransport-Operation durchgeführt wird, schaltet der Multiplexer AMUX entweder den Inhalt des PC oder des MAR als Speicheradresse ($addr\_out$) an den Speicherblock.

Die Baugruppe Constants stellt die Konstanten 0, 1 und 4 zur Verfügung. Bei der Initialisierung ($reset$) werden die Register PC, MAR, MDR auf 0 gesetzt, im Verlauf des SLT-Befehls werden die entsprechenden RF-Register mit 0 oder 1 geladen und das Inkrementieren des PC erfordert die Konstante 4.

Anmerkung In der Abbildung 5 sind die Steuersignale bei den einzelnen Komponenten mit angegeben. Die detailierte Beschreibung der Ansteuerung aller Baugruppen des Datenpfads ist für das Verständnis der prinzipiellen Funktionsweise des DLXJ-Prozessors nicht erforderlich und nicht Gegenstand dieser Abhandlung.

4.2. Die Steuerung

Die Steuerung (Abbildung 6) enthält außer dem Schaltwerk auch drei Decoder und das IR. Zur Verdeutlichung der Befehlsdecodierung sind die VHDL-Beschreibungen der drei Decoder im Abschnitt A.2 bis A.4 beigefügt.

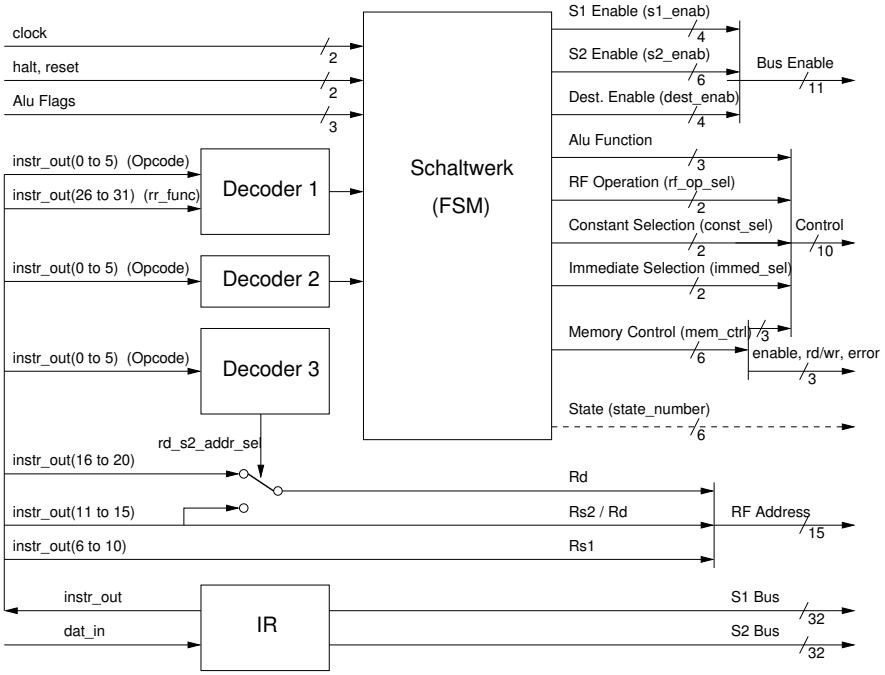

Abbildung 6: Die Steuerung des DLXJ

Der Decoder 1 entschlüsselt den aktuellen Befehl aus dem Bitfeld Opcode. Gehört der Befehl zu den Register-Register-ALU Operationen, dann entscheidet das Bitfeld rr_func darüber, welche ALU-Operation auszuführen ist. Das Ergebnis dieser Decodierung ist eine dem Befehl entsprechende Eingabe für das Schaltwerk (vgl. Tabelle 1 und Abbildung 1).

Der Decoder 2 decodiert die Datentransport-Operationen weiter aus, da diese als zusätzliches Eingangssignal für das Schaltwerk benötigt werden.

Der Decoder 3 erzeugt ein Steuersignal ($rd\_s2\_addr\_sel$), mit dem in Abhängigkeit vom Befehlsformat das entsprechende Bitfeld des Befehlswortes als RF-Adresse gewählt wird (vgl. Abbildung 1). Wird ein R-Typ-Befehl erkannt, dann wird der GPR-Adresse Rd das Bitfeld IR(16 ... 20) zugeordnet. Entsprechend ergibt sich Rd zu IR(11 ... 15), wenn der Befehl kein R-Typ-Befehl ist. Für Befehle, die kein Ergebnis in das GPR zurückschreiben, ist eine Zuordnung nicht erforderlich, so daß nicht weiter zwischen I- und J-Typ-Befehlen unterschieden werden muß. Die Steuerung stellt somit dem Datenpfad die dem Befehl entsprechenden Bitfelder $Rs1$, $Rs2$ und Rd als GPR-Adressen ($RF\ Address$) zur Verfügung.

Das Signal $rd\_s2\_addr\_sel$ wird auch für die Auswahl der S2 Bus-Quelle ($S2\ Enable$) bei arithmetisch-logischen Befehlen genutzt (nicht in Abb. 6 dargestellt). R-Typ-Befehlen wird Register B als Quelle zugeordnet, bei I-Typ-Befehlen wird das 16-Bit Immediate des IR als S2 Bus-Quelle verwendet.

Im Verlauf des Experiments „RISC-Prozessor“ soll der Befehlssatz um die in der Tabelle 3 spezifizierten Befehle erweitert werden. Diese Befehle werden bei den folgenden Betrachtungen mit berücksichtigt.

Bei Operationen, die den Inhalt des Registers A mit dem Immediate des Registers IR arithmetisch verknüpfen ($LW.I$, $SW.I$, J, $BEQZ$, $ADD.I$, $SUB.I$), werden die 16 Bit des Immediate im IR vorzeichenrichtig auf 32 Bit erweitert. Bei logischen Operationen ($AND.I$, $OR.I$) werden die fehlenden 16 Bit mit Nullen aufgefüllt.

Das Befehlsregister wird zu Beginn einer jeden Befehlsabarbeitung in einem Speicher-Lese-Zugriff mit dem aktuellen Befehl geladen. Die hierfür notwendigen Steuersignale generiert das Schaltwerk. Das IR stellt die befehlsbezogenen Informationen für die Steuerung ($Opcode$, rr_func, $Rs1$, $Rs2$, Rd) bereit. Die Bitfelder „Immediate“ (16 Bit) und „Offset added to PC“ (26 Bit) werden vom IR dem Befehl entsprechend auf 32 Bit erweitert und können als ALU-Operanden sowohl am S1 Bus als auch am S2 Bus genutzt werden.

Das Schaltwerk (ausführlich beschrieben im Abschnitt 4.4) wertet alle Eingangssignale ($halt$, $reset$, $Alu\ Flags$, Ausgangssignale der Decoder 1 und 2) aus. Mit diesen Informationen werden folgende Signale erzeugt (siehe auch Abschnitt A.5):

Bus Enable Freigabe der Ausgänge von A, MDR, IR oder Constants am S1 Bus durch die Signale $S1\ Enable$, Freigabe der Ausgänge von B, PC, MAR, IR oder Constants am S2 Bus mit den Signalen $S2\ Enable$ und Freigabe der Eingänge von C, PC, MAR oder MDR am Zielbus mithilfe der Signale $Dest\ Enable$.

Control Auswahl der ALU-Operation ($Alu\ Function$), Steuerung der Zugriffe auf das GPR ($RF\ Operation$), Auswahl einer der Konstanten 0, 1 oder 4 ($Constant\ Selection$), Auswahl entweder des Bitfeldes „Immediate“ (16 Bit) oder des Bitfeldes „Offset added to PC“ (26 Bit) aus dem Befehlswort ($Immediate\ Selection$) und der Datenpfad-internen Speicher-Steuersignale für das IR und die Multiplexer DMUX und AMUX ($Memory\ Control$)

Enable, Rd/Wr, Error Steuerung der Speichereinheit außerhalb des Prozessorkerns ($enable$, $read/write$) und Statusanzeige ($error$).

State Anzeige des aktuellen Zustands der Steuerung [2]

[2] nicht für die Funktion des Prozessors erforderlich

Befehl	Assemblerbefehl	Bedeutung	Typ	Opcode rr_func
subtraction	$SUB.I\ Rd, Rs1, I$	$Rd \leftarrow Rs1 - I$[a]	I	010110
addition	$ADD\ Rd, Rs1, Rs2$	$Rd \leftarrow Rs1 + Rs2$	R	000000 000100
addition	$ADD.I\ Rd, Rs1, I$	$Rd \leftarrow Rs1 + I$[a]	I	010100
logical AND	$AND\ Rd, Rs1, Rs2$	$Rd \leftarrow Rs1\ AND\ Rs2$	R	000000 001000
logical AND	$AND.I\ Rd, Rs1, I$	$Rd \leftarrow Rs1\ AND\ I$[b]	I	011000
logical OR	$OR\ Rd, Rs1, Rs2$	$Rd \leftarrow Rs1\ OR\ Rs2$	R	000000 001001
logical OR	$OR.I\ Rd, Rs1, I$	$Rd \leftarrow Rs1\ OR\ I$[b]	I	011001
shift right logical	$SRL\ Rd, Rs1, Rs2$	$Rd \leftarrow Rs1\ log.\ rechts\ geschoben\ mit\ Rs2$	R	000000 001110
shift right logical	$SRL.I\ Rd, Rs1, I$	$Rd \leftarrow Rs1\ log.\ rechts\ geschoben\ mit\ I$[c]	I	011110
shift left logical	$SLL\ Rd, Rs1, Rs2$	$Rd \leftarrow Rs1\ log.\ links\ geschoben\ mit\ Rs2$	R	000000 001100
shift left logical	$SLL.I\ Rd, Rs1, I$	$Rd \leftarrow Rs1\ log.\ links\ geschoben\ mit\ I$[c]	I	011100

[a] vorzeichenerweiterter 16-bit Offset
[b] mit Nullen erweiterter 16-bit Offset
[c] nur die 5 niederwertigsten Bit sind signifikant

Tabelle 3: Zusätzlich zu implementierende arithmetisch-logische Befehle

4.3. Grundschritte der Befehlsausführung

Bei der Abarbeitung der Befehle unterscheidet man 5 Grundschritte:

- Befehl holen (fetch),
- Befehl decodieren (decode),
- Befehl ausführen (execute),
- Speicher zugreifen (memory) und
- Resultat schreiben (write back).

Das gilt sowohl für den DLXJ-Prozessor als auch für den DLX-Prozessor. Hier werden nur die Operationen beschrieben, die für die DLXJ-Befehle notwendig sind, einschließlich der Erweiterung des Befehlssatzes.

1. Befehlsholeschritt (fetch):

$$IR \leftarrow M[PC] \quad \text{(alle Befehle)}$$

   Der Inhalt des PC wird über den AMUX als Adresse an den Speicher angelegt (Abbildung 5). Die Daten der so adressierten Speicherzelle (das Befehlswort) werden in das IR geladen.

2. Befehlsdecodierung/Registerholeschritt (decode):

$$A \leftarrow Rs1; \quad B \leftarrow Rs2; \quad PC \leftarrow PC + 4 \quad \text{(alle Befehle)}$$

   Der Befehl wird decodiert, die Daten werden aus dem RF in die Register A und B geladen und der PC wird um 4 erhöht.

   Gleichzeitig mit der Decodierung werden die Register A und B geladen. Das Befehlsformat ist zu diesem Zeitpunkt noch nicht bekannt. Die Register A und B können dennoch geladen werden, weil die Quellregister-Adressen immer an derselben Stelle im Befehlswort angeordnet sind (Abbildung 1). Diese Art der Registeradressierung wird als Festfeldcodierung bezeichnet. Wurde gerade kein R-Typ-Befehl geladen, wird der Ladevorgang aus dem RF nach A und B genauso durchgeführt, als wäre es ein R-Typ-Befehl. Im Verlauf der weiteren Befehlsabarbeitung bleiben die fälschlich geladenen Register ja ungenutzt.

3. Ausführungs-/Effektivadreß-Schritt (execute):

   Die ALU operiert mit den im vorangegangenen Schritt vorbereiteten Operanden abhängig von der Befehlsklasse in einer der folgenden drei Funktionen.

   a) Speicherzugriff:

$$\begin{aligned} MAR &\leftarrow A + (IR_{16})^{16} \#\# IR_{16\ldots31} && (LW.I,\ SW.I) \\ MDR &\leftarrow RD && (SW.I) \end{aligned}$$

   Die ALU addiert die Operanden zur Bildung der effektiven Adresse, die in das MAR geladen wird. Das MDR wird immer dann geladen, wenn der Speicherzugriff ein Schreibzugriff ist.

b) ALU-Befehl:

$$C \leftarrow A\ op\ B \qquad (ADD,\ SUB,\ AND,\ OR,\ SLL,\ SRL)$$
$$C \leftarrow A\ op\ (IR_{16})^{16}\#\#IR_{16\ldots31} \qquad (ADD.I,\ SUB.I,\ SLL.I,\ SRL.I)$$
$$C \leftarrow A\ op\ (0)^{16}\#\#IR_{16\ldots31} \qquad (AND.I,\ OR.I)$$

Die ALU führt die dem Befehl entsprechende arithmetische oder logische Operation op mit den Werten in A($Rs1$) und in B($Rs2$) oder mit dem um das Vorzeichen bzw. um 0 erweiterten 16-Bit Immediate aus. Bei den Schiebeoperationen sind nur die 5 niederwertigsten Bit des Registers B signifikant.

c) Verzweigung, Sprung:

$$cond \leftarrow A\ op\ 0 \qquad (BEQZ)$$
$$PC \leftarrow PC + (IR_6)^6\#\#IR_{6\ldots31} \qquad (J)$$

Im Falle der bedingten Verzweigung wird das Register A auf null getestet. Bei einem unbedingten Sprung addiert die ALU den 26-Bit-Offset vorzeichenerweitert zum PC und speichert das Ergebnis (die Sprungadresse) im PC ab.

4. Speicherzugriff, Verzweige-Komplettierungsschritt (memory)

Dieser Schritt ist nur für Speicherzugriffe und bedingte Verzweigungen relevant.

a) Speicherzugriff:

$$MDR \leftarrow M[MAR]; \quad C \leftarrow MDR \qquad (LW.I)$$
$$M[MAR] \leftarrow MDR \qquad (SW.I)$$

Bei einem Ladebefehl werden die Daten zunächst vom Speicher in das MDR eingelesen und dann vom MDR nach C geschrieben. Wird ein Schreibbefehl ausgeführt, so werden die Daten aus dem MDR in den Speicher geschrieben. Die Speicheradresse steht in beiden Fällen im MAR.

b) Verzweigung:

$$if\ (cond)\ \ PC \leftarrow PC + (IR_{16})^{16}\#\#IR_{16\ldots31} \qquad (BEQZ)$$

Falls die Bedingung $cond$ wahr ist, addiert die ALU den 16-Bit-Offset vorzeichenerweitert zum PC und speichert das Ergebnis (die Sprungadresse) im PC ab.

5. Rückschreibe-Schritt (write back)

$$Rd \leftarrow C \quad \text{(SLT, LW.I, alle arithmetisch-logischen Befehle)}$$

Das vom Speicher oder der ALU kommende Resultat wird in das GPR mit der Adresse Rd geschrieben. Das betrifft die Befehle SLT, $LW.I$, ADD, $ADD.I$, SUB, $SUB.I$, AND, $AND.I$, OR, $OR.I$, SLL, $SLL.I$, SRL und $SRL.I$.

4.4. Das Schaltwerk

Das Schaltwerk ist als Moore-Automat realisiert. Ein Moore-Automat ist dadurch gekennzeichnet, daß die Ausgabesignale nur vom aktuellen Zustand, nicht aber von der Eingabe abhängig sind. Ein Automat wird als binärer Automat bezeichnet, wenn Eingaben, Ausgaben und Zustände durch Binärvektoren dargestellt werden. Abbildung 7 zeigt den prinzipiellen Aufbau, der durch die Automatengleichungen

$$\boldsymbol{z}(t+1) = f\big(\boldsymbol{x}(t), \boldsymbol{z}(t)\big) \tag{1}$$

für die Zustands- oder Speicherüberführungsfunktion und

$$\boldsymbol{y}(t) = g\big(\boldsymbol{z}(t)\big) \tag{2}$$

für die Ausgabe- oder Ergebnisfunktion beschrieben wird.

Mit dem Eingabevektor $\boldsymbol{x}(t)$ werden alle Eingangssignale berücksichtigt, der Ausgabevektor $\boldsymbol{y}(t)$ enthält alle Ausgangssignale. Der Zustandsvektor $\boldsymbol{z}(t)$ wird durch den Zustandsspeicher (ein Register) repräsentiert. Mit jeder positiven Taktflanke geht der Folgezustand $\boldsymbol{z}(t+1)$ in den Momentanzustand $\boldsymbol{z}(t)$ über. Die Zustandsüberführungsfunktion f und die Ergebnisfunktion g werden durch je ein Schaltnetz (Kombinatorik) realisiert.

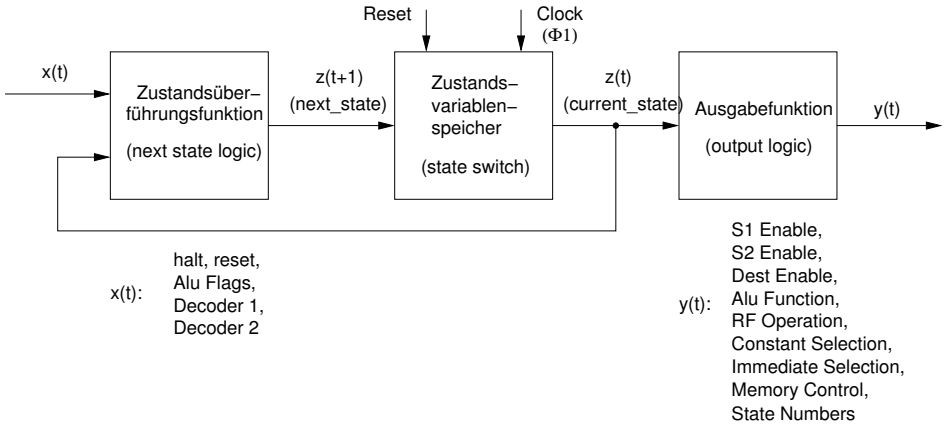

Abbildung 7: Binärer Moore-Automat

Der Zustandsspeicher hat noch eine weitere Aufgabe. Bei jedem Zustandsübergang wird das Reset-Signal überprüft. Falls es aktiv ist, wird anstatt des Folgezustands der Reset-Zustand eingenommen und erst bei inaktivem Reset wieder verlassen. Die Abbildung 8 zeigt den vollständige Zustandsgraphen des DLXJ-Prozessors einschließlich des Verhaltens bei aktivem $reset$- oder $halt$-Signal.

Die Zustandsknoten enthalten im oberen Teil die Bezeichnung des Zustandes, der am LCD-Display des Experimentalaufbaus und bei einigen Debuggerfunktionen am Hostrechner ausgegeben wird. Der untere Teil des Zustandsknotens beschreibt die Aufgabe des Zustandes. An den Kanten des Zustandsgraphen sind bei bedingten Übergängen die Bedingungen für den jeweiligen Übergang vermerkt.

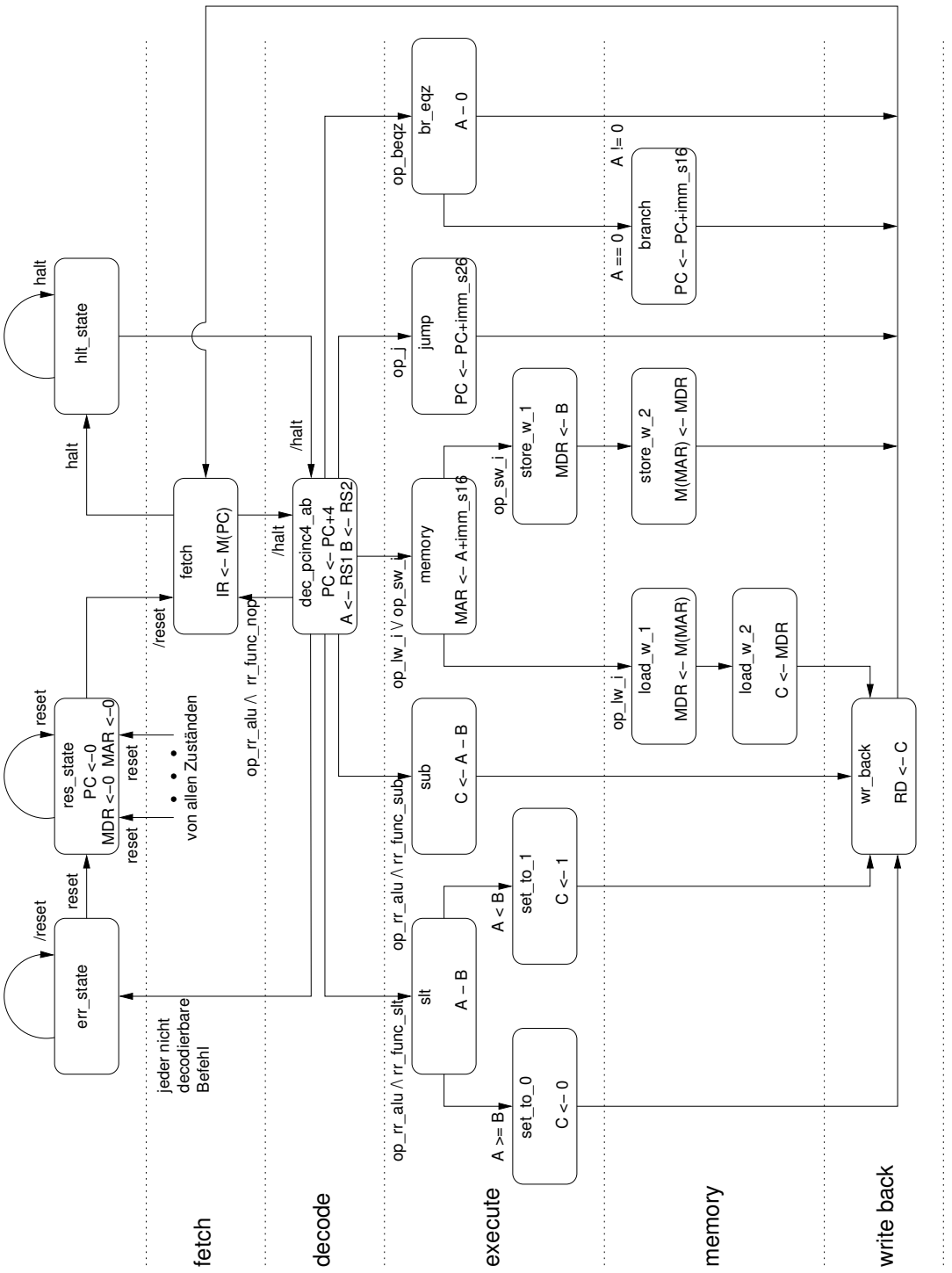

Abbildung 8: Vollständiger Zustandsgraph des DLXJ

Die Funktionsweise des Schaltwerks bei der Befehlsabarbeitung wird nun an einem Beispiel gezeigt.

Auf der ROM-Adresse $0x0000\_0100$ möge der Befehlscode für den Assemblerbefehl $SUB\ R10, R20, R30$ gespeichert sein. Die Signale $halt$ und $reset$ seien inaktiv. Die Befehlsausführung (vgl. Abschnitt 4.3) durchläuft dann folgende Schritte:

1. Befehlsholeschritt (fetch)

   Zustand: $fetch$

   $$IR \leftarrow M[0x0000\_0100]$$

   Ergebnis: Der Befehlscode für die Anweisung $SUB\ R10, R20, R30$ befindet sich im IR.

   Der Übergang zum nächsten Zustand ist bedingt; er findet bei inaktivem Halt-Signal zum Zustand dec_pcinc4_ab und bei aktivem Halt zum Halt-Zustand hlt_state statt.

2. Befehlsdecodierung/Registerholeschritt (decode)

   Zustand: dec_pcinc4_ab

   $$A \leftarrow GPR20; \quad B \leftarrow GPR30; \quad PC \leftarrow 0x0000\_0104$$

   Ergebnis: Opcode $op\_rr\_alu = 000000$,
   rr_func $rr\_func\_sub = 000110$.

   Der Übergang zum nächsten Zustand ist bedingt; er wird durch den Decoder 1 bestimmt. Dem Decodierungsergebnis ($op\_rr\_alu \wedge rr\_func\_sub$) entsprechend wird zum Folgezustand sub übergegangen.

3. Ausführungs-/Effektivadreß-Schritt (execute)

   Zustand: sub

   $$C \leftarrow A - B$$

   Ergebnis: Die Differenz aus den Werten in A und B ist in C gespeichert.

   Der Übergang zum Folgezustand wr_back ist an keine Bedingung geknüpft.

4. Speicherzugriff, Verzweige-Komplettierungsschritt (memory)

   Dieser Schritt ist für RRA-Befehle nicht relevant.

5. Rückschreibe-Schritt (write back)

   Zustand: wr_back

   $$GPR10 \leftarrow C$$

   Ergebnis: Das in der ALU berechnete und bereits in C zwischengespeicherte Resultat wird in das GPR10 geschrieben.

   Der Übergang zum nächsten Zustand ist an keine Bedingung geknüpft; der Folgezustand ist $fetch$.

Das Schaltwerk ist durch je eine VHDL-Beschreibung für die Zustandsüberführungsfunktion (Abschnitt A.6), für den Zustandsspeicher (Abschnitt A.7) und für die Ergebnisfunktion (Abschnitt A.8) spezifiziert. Die Beschreibungen für die Zustandsüberführungsfunktion und für den Zustandsspeicher sind in Verbindung mit dem Zustandsgraphen der Abbildung 8 weitgehend selbsterklärend.

In der Beschreibung der Ergebnisfunktion werden, abhängig vom aktuellen Zustand, den Ausgangssignalen vordefinierte Konstanten (Abschnitt A.5) zugewiesen. Die Ergebnisfunktion ist für den Zustand sub ausführlich kommentiert (Abschnitt A.8), so daß die Zusammenhänge in Verbindung mit den Ausgabekonstanten (Abschnitt A.5) und den Steuersignalen des Datenpfads (Abb. 5) nachvollzogen werden können.

An einem Bus darf auf einer Leitung zur selben Zeit immer nur eine Signalquelle aktiv sein. So wird zum Beispiel im Zustand sub den Ausgängen $S1\ Enable$ (Signalvektor $s1\_enab$) der Konstantenvektor $s1\_a$ zugewiesen. Mit diesem Konstantenvektor ist das Enablesignal a_out_en aktiv ($'0'$), d.h. der Inhalt von Register A liegt am S1 Bus (Abb. 5) an. Die Signale mdr_out1_en (MDR), $ir\_immed\_o1\_en$ (IR) und $const\_o1\_en$ (Constants) sind inaktiv ($'1'$). Folglich sind die Ausgänge dieser Register nicht wirksam, so daß nur eine aktive Signalquelle am S1 Bus existiert.

5. Die Entwicklungsumgebung

Die Entwicklungsumgebung zum DLXJ-Prozessor besteht aus einem Digitalsimulator, einem Assembler, einem Debugger, Synthesewerkzeugen und einem FPGA-Experimentalsystem. Die so gegebenen Voraussetzungen gestatten

- die Erweiterung des Befehlssatzes,
- die Veränderung der Struktur des Prozessorsystems (Prozessorkern, Speicher, Interfaces),
- die Entwicklung einfacher Assemblerprogramme,
- die funktionale Simulation des gesamten Systems auf verschiedenen Implementierungsstufen einschließlich der Simulation einfacher Programme,
- die Übersetzung der Prozessorspezifikation (VHDL) in Hardware (FPGA-Konfiguration),
- das Debuggen von Assemblerprogrammen am Digitalsimulator und an einem FPGA-Experimentalsystem.

Die verschiedenen Werkzeuge der Entwicklungsumgebung können über Shell-Scripte gestartet werden. Die Bezeichnungen sind weitgehend selbsterklärend. Die Aufrufmöglichkeiten und die Abhängigkeiten zwischen den einzelnen Werkzeugen sind in der Abbildung 9 dargestellt und werden in den folgenden Abschnitten erläutert.

5.1. Assemblierung von Programmen

Der Assembler [1, 2] unterstützt den kompletten Befehlssatz des DLX nach Hennessy und Patterson. Die zu übersetzende Datei wird eingelesen und auf Syntaxfehler geprüft. Eventuell vorhandene Fehler werden mit Angabe der Zeilennummer angezeigt. Nach erfolgreicher Syntaxprüfung generiert der Assembler eine VHDL-Beschreibung für einen ROM. Der Maschinencode in Hex-Format ist Bestandteil dieser VHDL-Beschreibung. In einem weiteren Schritt wird der Maschinencode daraus extrahiert und steht nun für die Verwendung im Simulator und im Experimentalsystem zur Verfügung. Mit der Anweisung

```
dlxj_assemble_program <filename>
```

werden Assemblerprogramme in Maschinencode übersetzt. Das Format der bereits implementierten und der noch zu implementierenden Assemblerbefehle kann den Tabellen 1 und 3 entnommen werden. Ein Beispiel für ein Assemblerprogramm ist als Abschnitt B beigefügt.

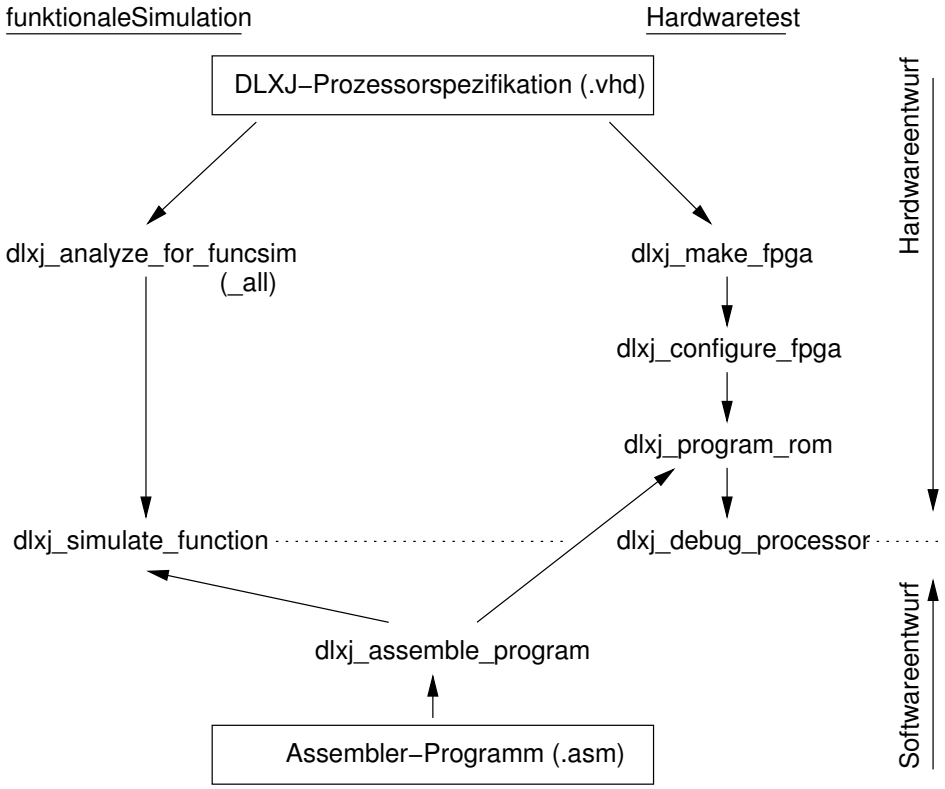

Abbildung 9: Der Entwurfsfluß

5.2. Analyse und Synthese

Die Darstellung der Analyse und der Synthese von Hardware-Beschreibungen ist nicht Gegenstand dieser Abhandlung. Für die Analyse und für die Synthese des DLXJ-Prozessors sind

entsprechende Scripte erstellt worden, die im Verlauf des Entwurfsprozesses immer dann auszuführen sind, wenn Änderungen an der VHDL-Spezifikation des DLXJ-Prozessors oder an dem aktuell zu untersuchenden Assemblerprogramm vorgenommen wurden (Abschnitte 5.3 und 5.4).

5.3. Simulation

Die Simulation zeigt genau die Zusammenhänge, die bereits durch die VHDL-Beschreibung festgelegt wurden, d.h. es wird eine funktionale Simulation durchgeführt. Dazu genügt es, die VHDL-Spezifikation direkt in ein Simulationsmodell zu übersetzen. Das geschieht, indem das Shell-Scripts `dlxj_analyze_for_funcsim` gestartet wird. Die Abbildung 9 zeigt die Vorgehensweise für die funktionale Simulation und für den Hardwaretest. Jede Änderung der Prozessorspezifikation oder des Assemblerprogramms erfordert die erneute Ausführung der erforderlichen Zwischenschritte (Abbildung 9). Wurde z.B. das Assemblerprogramm geändert, so ist das Programm zu assemblieren, die Simulation zu beenden und erneut zu starten (`dlxj_simulate_function`).

Mit dem Start des Simulationsprogramms wird das zu simulierende Prozessormodell einschließlich des zuletzt übersetzten Assemblerprogramms in den Simulator geladen. Der Simulator wird initialisiert und zwei Fenster werden geöffnet:

DVE Die grafische Benutzeroberfläche DVE (Discovery Visual Environment) startet, wenn das Script `dlxj_simulate_function` ausgeführt wird. Das Wave-Fenster zeigt sofort alle benötigten Signalverläufe an. Im rechten Teil der Menüleiste befinden sich zwei speziell für die Simulation des DLXJ-Prozessors implementierte Schaltflächen:

- **`NI`** Mit dem Befehl **`NI`** wird der Simulator veranlaßt, einen vollständigen DLXJ-Befehl abzuarbeiten, so daß die Simulation immer vom Zustand $fetch$ eines Befehls bis zum Zustand $fetch$ des folgenden Befehls fortschreitet. Die Anzahl der mit **`NI`** ausgelöste Abarbeitung von Zuständen ist befehlsabhängig. Durch wiederholte Betätigung von **`NI`** kann das geladene Programm Befehl für Befehl simuliert werden.
- **`80ns`** Die Schaltfläche **`80ns`** bewirkt, daß die Simulation um $80ns$ voranschreitet. Das entspricht einem Taktzyklus des Prozessors.

DLXJ Instructions Dieses Fenster zeigt den in die Speichereinheit geladenen Befehlscode (Hexformat) und die nacheinander abgearbeiteten Assemblerbefehle mit den zugehörigen Registerinhalten fortlaufend an. Der Fensterinhalt kann ausgedruckt werden (`dlxj_print_funcsim_results`).

5.4. Der DLXJ-Debugger

Der Debugger wurde für den Test der DLXJ-Prozessorhardware und für den Test von Assemblerprogrammen entwickelt. Bevor der Debugger gestartet werden kann (`dlxj_debug_processor`), sind folgende Vorbereitungen (Abbildung 9) notwendig:

Nach erfolgreich durchgeführter Synthese (`dlxj_make_fpga`), steht eine Konfigurationsdatei für das FPGA des Experimentalsystems zur Verfügung. Diese Datei ist nun mit Hilfe

des Scripts `dlxj_configure_fpga` in das FPGA zu übertragen. Jetzt hat das FPGA die Funktionalität des DLXJ-Prozessorsystems erhalten und das Programm kann in den Speicher des Prozessors übertragen werden. Der hierfür erforderliche Maschinencode wird bei der Assemblierung (`dlxj_assemble_program`) des zu testenden Programms generiert und mit dem Script `dlxj_program_rom` in den Speicherblock des DLXJ-Prozessors geschrieben.

Die Funktionen des Debuggers Der Debugger wird über die Standardeingabe bedient. Die Ausgabe von Informationen erfolgt über das LCD-Display des Experimentalaufbaus und an der Standardausgabe des Hostrechners.

Der Debugger stellt folgende Funktionen zur Verfügung:

Allgemeine Funktionen:

- `da` : der Code des geladenen Programms wird disassembliert und angezeigt
- `srr` : Register- und Datenspeicherinhalte werden abgespeichert
- `q` : der Debugger wird verlassen
- `h` : Anzeige aller Funktionen mit einer kurzen Erklärung.

Anzeige interner Bussignale, sowie von Speicher- und Registerinhalten:

- `bus` : alle oben aufgeführten Bussignale
- `rom` : der Inhalt des Teils des Programmspeichers, der mit Programmcode belegt ist
- `ram` : der Inhalt des Datenspeichers beginnend bei der Adresse $0x1000\_0000$. Die Anzahl der anzuzeigenden Datenworte kann eingegeben werden (Voreinstellung 32 Worte)
- `reg` : Inhalt aller 32 GPR, der Register PC, MAR, MDR, IR und des Display Registers

Steuerung des Prozessors:

`hlt` : das Signal *halt* wird aktiviert
`res` : das Signal *reset* wird aktiviert
`nrh` : die Signale *reset* und *halt* werden deaktiviert
`cc` : Umschaltung von Einzeltakt-Betrieb auf Quarzgenerator-Takt
`#` : es wird genau eine Taktperiode erzeugt und der aktuelle Zustand angezeigt
`#reg` : `#` und `reg` werden nacheinander ausgeführt
`ni` : der folgende Befehl wird ausgeführt und disassembliert; die durchlaufenen Zustände, die beteiligten Register und die Registerinhalte werden angezeigt
`nb` : das Programm wird bis zum nächsten Unterbrechungspunkt abgearbeitet (Abbruch mit Ctrl-c); die durchlaufenen Anweisungen weden disassembliert, Zielregisterinhalte werden angezeigt
`sb1` : Unterbrechungspunkt 1: Eingabe der Zeile im Assemblerprogramm, bis zu der das Programm abgearbeitet werden soll; die wirksamen Zeilen werden mit dem Befehl `da` (siehe dort) angezeigt
`sb2` : Unterbrechungspunkt 2: ein weiterer Unterbrechungspunkt
`db1` : Unterbrechungspunkt 1 wird gelöscht
`db2` : Unterbrechungspunkt 2 wird gelöscht
`srn` : Eingabe der Anzahl n Taktperioden, die bei Eingabe von `rn` erzeugt werden
`rn` : es werden n Taktperioden erzeugt

Anhang A VHDL-Beschreibungen

In den folgenden Abschnitten werden die VHDL-Beschreibungen einiger Baugruppen des DLXJ-Prozessorkerns angegeben.

A.1 ALU

```
LIBRARY IEEE;  USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_signed.ALL;
USE WORK.control_types.ALL;
LIBRARY IEEE_EXTD;
USE IEEE_EXTD.stdl1164_vector_arithmetic.ALL;

ENTITY alu_core IS
  PORT (
    stored_s1   : IN std_logic_vector (0 TO 31);
    stored_s2   : IN std_logic_vector (0 TO 31);
    alu_op      : IN alu_func_type;
    result      : OUT std_logic_vector (0 TO 31);
    negative    : OUT std_logic;
    zero        : OUT std_logic);
END alu_core;

ARCHITECTURE behaviour OF alu_core IS
BEGIN
  s1_func_s2: PROCESS(stored_s1, stored_s2, alu_op)
    VARIABLE tmp_res : std_logic_vector(0 TO 31);
  BEGIN
    CASE alu_op IS
      WHEN alu_pass_s1
        =>      tmp_res := stored_s1;
      WHEN alu_pass_s2
        =>      tmp_res := stored_s2;
      WHEN alu_s1_add_s2
        =>      sv_add(stored_s1, stored_s2, tmp_res);
      WHEN alu_s1_sub_s2
        =>      sv_sub(stored_s1, stored_s2, tmp_res);
      WHEN alu_s1_and_s2
        =>      tmp_res := stored_s1 AND stored_s2;
      WHEN alu_s1_or_s2
        =>      tmp_res := stored_s1 OR stored_s2;
      WHEN alu_sll_s1_s2
        =>      sv_sll(stored_s1, tmp_res, stored_s2(27 to 31));
      WHEN alu_srl_s1_s2
        =>      sv_srl(stored_s1, tmp_res, stored_s2(27 to 31));
      WHEN OTHERS
        =>      NULL;
    END CASE;
    negative  <= tmp_res(0);
    result    <= tmp_res;
  END PROCESS s1_func_s2;
  zero      <= '1' WHEN (stored_s1 = 0) ELSE '0';
END behaviour;
```

A.2 Decoder 1

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;
USE WORK.dlx_instructions.ALL;
USE WORK.control_types.ALL;

ENTITY ir_decode_1 IS
  PORT (
    instr_in : IN  dlx_word;
    dec1_out : OUT fsm_states);
END ir_decode_1;

ARCHITECTURE behaviour OF ir_decode_1 IS
BEGIN
  decode_1 : PROCESS(instr_in)
  BEGIN
    CASE instr_in(0 TO 5) IS       -- Opcode from IR
      WHEN op_rr_alu =>            -- |-> any RRA operation
        CASE instr_in(26 TO 31) IS -- |   RRA Function (rr_func)
          WHEN rr_func_nop =>      -- |   |-> no operation
            dec1_out <= fetch;     -- |   |    -> fetch
          WHEN rr_func_sub =>      -- |   |-> subtraction
            dec1_out <= sub;       -- |   |    -> sub
          WHEN rr_func_slt =>      -- |   |-> set if lower than
            dec1_out <= slt;       -- |   |    -> slt
          WHEN OTHERS      =>      -- |   |-> unkown
            dec1_out <= err_state; -- |        -> err_state
        END CASE;                  -- |
      WHEN op_lw_i |               -- |-> load word  OR
           op_sw_i   =>            -- |   store word
        dec1_out     <= memory;    -- |        -> memory
      WHEN op_beqz   =>            -- |-> branch if equal zero
        dec1_out     <= br_eqz;    -- |        -> br_eqz
      WHEN op_j      =>            -- |-> jump
        dec1_out     <= jump;      -- |        ->jump
      WHEN OTHERS    =>            -- |-> unkown
        dec1_out     <= err_state; -- |        -> err_state
    END CASE;
  END PROCESS decode_1;
END behaviour;
```

A.3 Decoder 2

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;
USE WORK.dlx_instructions.ALL;
USE WORK.control_types.ALL;

ENTITY ir_decode_2 IS
  PORT (
    instr_in    : IN  dlx_word;
    dec2_out    : OUT fsm_states);
END ir_decode_2;

ARCHITECTURE behaviour OF ir_decode_2 IS
BEGIN
  decode_2 : PROCESS(instr_in)
  BEGIN
    CASE instr_in(0 TO 5) IS       -- Opcode from IR
      WHEN op_lw_i =>              -- |-> load word
        dec2_out <= load_w_1;      -- |    -> load_w_1
      WHEN op_sw_i =>              -- |-> store word
        dec2_out <= store_w_1;     -- |    -> store_w_1
      WHEN OTHERS =>               -- |-> other Opcode
        dec2_out <= err_state;     -- |    -> err_state
    END CASE;
  END PROCESS decode_2;
END behaviour;
```

A.4 Decoder 3

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;
USE WORK.dlx_instructions.ALL;
USE WORK.control_types.ALL;

ENTITY ir_decode_3 IS
  PORT (
    instr_in      : IN  dlx_word;
    rd_s2_adr_sel : OUT std_logic
    );
END ir_decode_3;

ARCHITECTURE behaviour OF ir_decode_3 IS
BEGIN
  decode_3 : PROCESS(instr_in)
  BEGIN
    CASE instr_in(0 TO 5) IS       -- Opcode from IR
      WHEN op_rr_alu =>            -- |-> any RRA operation
            rd_s2_adr_sel <= '1';  -- |   |    -> Rd = Rd(Rtype)
      WHEN OTHERS    =>            -- |-> other Opcode
        rd_s2_adr_sel     <= '0';  -- |        -> Rd = Rd(Itype)
    END CASE;
  END PROCESS decode_3;
END behaviour;
```

A.5 Steuerkonstanten

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

PACKAGE control_types IS

-- Alu Function:
   ------------
  TYPE alu_func_type  IS
    (alu_pass_s1,
     alu_pass_s2,
     alu_s1_add_s2,
     alu_s1_sub_s2,
     alu_s1_and_s2,
     alu_s1_or_s2,
     alu_sll_s1_s2,
     alu_srl_s1_s2,
     alu_dcare);
  ATTRIBUTE enum_encoding : string;
  ATTRIBUTE enum_encoding OF alu_func_type :
    TYPE IS ("000 001 010 011 100 101 110 111 000");



-- S1 Enable:
   ---------
-- source for s1 bus (low activ signals)
-- |a|mdr|immed|bus_const| (b/pc/mar not connected to s1 bus)
-- activated with phi1 positive edge (fsm_next)
-- signal vector: s1_enab                              0123
  CONSTANT s1_a       : std_logic_vector(0 TO 3) := "0111";
  CONSTANT s1_mdr     : std_logic_vector(0 TO 3) := "1011";
  CONSTANT s1_immed   : std_logic_vector(0 TO 3) := "1101";
  CONSTANT s1_const   : std_logic_vector(0 TO 3) := "1110";
  CONSTANT s1_none    : std_logic_vector(0 TO 3) := "1111";--don't care not
  --      a_out_en (A) _____________________________||||     allowed (only
  --      mdr_out1_en (MDR) ________________________|||     one driver!)
  --      ir_immed_o1_en (IR) ______________________||
  --      const_o1_en (Constants) __________________|
```

```
-- S2 Enable:
   ---------
-- source for s2 bus (low activ signals)
-- |B|immed|B or immed|PC|MAR|Constants| (A, MDR not connected to s2 bus)
-- b_out_en or ir_immed_o2_en depends on instr.type, i.e. on rd_s2_addr_sel
-- activated with phi1 positive edge (fsm_next)
-- signal vector: s2_enab:                          012345
  CONSTANT s2_b      : std_logic_vector(0 TO 5) := "011111";
  CONSTANT s2_immed  : std_logic_vector(0 TO 5) := "101111";
  CONSTANT s2_Y      : std_logic_vector(0 TO 5) := "110111";
  CONSTANT s2_pc     : std_logic_vector(0 TO 5) := "111011";
  CONSTANT s2_mar    : std_logic_vector(0 TO 5) := "111101";
  CONSTANT s2_const  : std_logic_vector(0 TO 5) := "111110";
  CONSTANT s2_none   : std_logic_vector(0 TO 5) := "111111";--don't care not
  --      b_out_en (B) _____________________________||||||    allowed (only
  --      ir_immed_o2_en (IR) ______________________|||||    one driver!)
  --      b_out_en or ir_immed_o2_en (B or IR) _____||||
  --      pc_out_en (PC) ___________________________|||
  --      mar_out1_en (MAR) ________________________||
  --      const_o2_en (Constants) __________________|


-- Dest Enable:
   -----------
-- destination from dest bus (high active signals)
-- all inputs are gated with phi2 high level
-- signal vector: dest_enab                          0123
  CONSTANT dest_c     : std_logic_vector(0 TO 3) := "1000";
  CONSTANT dest_pc    : std_logic_vector(0 TO 3) := "0100";
  CONSTANT dest_mar   : std_logic_vector(0 TO 3) := "0010";
  CONSTANT dest_mdr   : std_logic_vector(0 TO 3) := "0001";
  CONSTANT dest_res   : std_logic_vector(0 TO 3) := "0111"; -- don't care not
  CONSTANT dest_none  : std_logic_vector(0 TO 3) := "0000"; -- allowed
  --      c_latch_en (C) ___________________________||||
  --      pc_latch_en (PC) _________________________|||
  --      mar_latch_en (MAR) _______________________||
  --      mdr_latch_en (MDR) _______________________|


-- Constant Selection:
   ------------------
-- signal vector: const_sel                           01
  CONSTANT const_00    : std_logic_vector(0 TO 1) := "00";
  CONSTANT const_01    : std_logic_vector(0 TO 1) := "01";
  CONSTANT const_04    : std_logic_vector(0 TO 1) := "10";
  CONSTANT const_dcare : std_logic_vector(0 TO 1) := "--";
  --      const_sel -> "00": tmp1 <= 0x00000000 ______||
  --                   "01": tmp1 <= 0x00000001 ______||
  --                   "10": tmp1 <= 0x00000004 ______||
  -- s1_bus <= tmp1       if enabled (const_o1_en)
  -- s2_bus <= 0x00000000 if enabled (const_o2_en)
```

```
-- Immediate Selection:
   -------------------
-- Sign extension: replicate the sign bit of the operand into
-- the most significant bits of the result.
-- Zero extension: replicate zero bits into the most significant
-- bits of the result.
-- signal vector: immed_sel
  CONSTANT imm_s16   : std_logic_vector(0 to 1) := "01";
  CONSTANT imm_u16   : std_logic_vector(0 to 1) := "00";
  CONSTANT imm_s26   : std_logic_vector(0 to 1) := "11";
  CONSTANT imm_u26   : std_logic_vector(0 to 1) := "10";
  CONSTANT imm_dcare : std_logic_vector(0 to 1) := "--";
  --      ir_immed_size: '0': 16 bit _________________||
  --                     '1': 26 bit _________________||
  --      ir_immed_sign: '0': zero extended __________|
  --                     '1': sign extended __________|



-- RF Operation:
   ------------
-- a_latch_en, b_latch_en: high active, gated with phi2
-- rf_wr_en: low active clocked with phi2 neg. edge
-- signal vector: rf_op_sel                          01
  CONSTANT rfop_none   : std_logic_vector(0 TO 1) := "01";
  CONSTANT rfop_ab_rf  : std_logic_vector(0 TO 1) := "11";
  CONSTANT rfop_rf_c   : std_logic_vector(0 TO 1) := "00";
  --      a_latch_en, b_latch_en (A, B) ______________||
  --      rf_wr_en (RF)_____________________________|


-- Memory Control:
   --------------
-- ir_latch_en: high active, gated with phi2 high
-- enable: memory access, high active, clocked with phi1 pos. edge
-- error pad signal: high active
-- signal vector: mem_ctrl                          012345
  CONSTANT mem_none  : std_logic_vector(0 TO 5) := "000010";
  CONSTANT mem_fetch : std_logic_vector(0 TO 5) := "100110";
  CONSTANT mem_ldst  : std_logic_vector(0 TO 5) := "010000";
  CONSTANT mem_lw    : std_logic_vector(0 TO 5) := "011110";
  CONSTANT mem_sw    : std_logic_vector(0 TO 5) := "010100";
  CONSTANT mem_error : std_logic_vector(0 TO 5) := "000011";
  --   ir_latch_en ___________________________________||||||
  --      addr_mux_sel -> '0': Mem. addr. <= PC ______|||||
  --                      '1': Mem. addr. <= MAR _____|||||
  --      mdr_mux_sel  -> '0': MDR <= Dest bus ________||||
  --                      '1': MDR <= Memory __________||||
  --      enable (memory) ____________________________|||
  --      rw (memory)  -> '1': read, '0': write _________||
  --      error ________________________________________|
```

```
-- FSM States:
   ----------
  TYPE fsm_states IS (
    res_state, fetch,     dec_pcinc4_ab, memory, load_w_1,
    load_w_2,  store_w_1, store_w_2,     br_eqz, branch,
    jump,      sub,       slt,           set_to_1,
     set_to_0, wr_back,   hlt_state,     err_state,
-- States for additional instructions:
    add, and_1, or_1, sll_1, srl_1 );


-- State:
   -----
-- for debugging purposes, assign a constant number to each state
-- signal vector: state_number
  SUBTYPE  fsm_state_numbers IS std_logic_vector(5 DOWNTO 0);
  CONSTANT res_state_no         : fsm_state_numbers := "000001"; -- state 1
  CONSTANT fetch_no             : fsm_state_numbers := "000010"; -- state 2
  CONSTANT dec_pcinc4_ab_no     : fsm_state_numbers := "000011"; -- state 3
  CONSTANT memory_no            : fsm_state_numbers := "000100"; -- state 4
  CONSTANT load_w_1_no          : fsm_state_numbers := "000101"; -- state 5
  CONSTANT load_w_2_no          : fsm_state_numbers := "000110"; -- state 6
  CONSTANT store_w_1_no         : fsm_state_numbers := "000111"; -- state 7
  CONSTANT store_w_2_no         : fsm_state_numbers := "001000"; -- state 8
  CONSTANT br_eqz_no            : fsm_state_numbers := "001001"; -- state 9
  CONSTANT branch_no            : fsm_state_numbers := "010000"; -- state 10
  CONSTANT jump_no              : fsm_state_numbers := "010001"; -- state 11
  CONSTANT sub_no               : fsm_state_numbers := "010010"; -- state 12
  CONSTANT slt_no               : fsm_state_numbers := "010011"; -- state 13
  CONSTANT set_to_1_no          : fsm_state_numbers := "010100"; -- state 14
  CONSTANT set_to_0_no          : fsm_state_numbers := "010101"; -- state 15
  CONSTANT wr_back_no           : fsm_state_numbers := "010110"; -- state 16
  CONSTANT hlt_state_no         : fsm_state_numbers := "010111"; -- state 17
  CONSTANT err_state_no         : fsm_state_numbers := "011000"; -- state 18
  CONSTANT add_no               : fsm_state_numbers := "011001"; -- state 19
  CONSTANT and_no               : fsm_state_numbers := "100000"; -- state 20
  CONSTANT or_no                : fsm_state_numbers := "100001"; -- state 21
  CONSTANT sll_no               : fsm_state_numbers := "100010"; -- state 22
  CONSTANT srl_no               : fsm_state_numbers := "100011"; -- state 23
END control_types;
```

A.6 Zustandsüberführungsfunktion

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;
USE WORK.control_types.ALL;

ENTITY fsm_next IS
  PORT (
    reset         : IN  std_logic;      -- reset input
    halt          : IN  std_logic;      -- halt input
    alu_zero      : IN  std_logic;      -- alu zero bit
    alu_neg       : IN  std_logic;      -- alu negative bit
    dec_1_in      : IN  fsm_states;     -- input from ir_decode_1
    dec_2_in      : IN  fsm_states;     -- input from ir_decode_2
    current_state : IN  fsm_states;     -- the current state
    next_state    : OUT fsm_states);    -- the next state
END fsm_next;

ARCHITECTURE dataflow OF fsm_next IS
BEGIN
  next_state_logic :
  PROCESS (current_state, reset, halt, alu_neg, alu_zero,
                                       dec_1_in, dec_2_in)
  BEGIN
    CASE current_state IS
      WHEN res_state =>
        next_state <= fetch;
      WHEN fetch =>
        IF halt = '1' THEN
          next_state <= hlt_state;
        ELSE
          next_state <= dec_pcinc4_ab;
        END IF;
      WHEN dec_pcinc4_ab =>
        next_state <= dec_1_in;
      WHEN wr_back =>
        next_state <= fetch;
      WHEN sub =>
        next_state <= wr_back;
      WHEN memory =>
        next_state <= dec_2_in;
      WHEN load_w_1 =>
        next_state <= load_w_2;
      WHEN load_w_2 =>
        next_state <= wr_back;
      WHEN store_w_1 =>
        next_state <= store_w_2;
      WHEN store_w_2 =>
        next_state <= fetch;
```

```
      WHEN br_eqz =>
        IF alu_zero = '0' THEN
          next_state <= fetch;
        ELSE
          next_state <= branch;
        END IF;
      WHEN branch =>
        next_state <= fetch;
      WHEN jump =>
        next_state <= fetch;
      WHEN slt =>
        IF alu_neg = '1' THEN
          next_state <= set_to_1;
        ELSE
          next_state <= set_to_0;
        END IF;
      WHEN set_to_1 =>
        next_state <= wr_back;
      WHEN set_to_0 =>
        next_state <= wr_back;
      WHEN hlt_state =>
        IF halt = '0' THEN
          next_state <= dec_pcinc4_ab;
        ELSE
          next_state <= hlt_state;
        END IF;
      WHEN err_state =>
        IF reset = '0' THEN
          next_state <= err_state;
        ELSE
          next_state <= res_state;
        END IF;
      WHEN OTHERS =>
        IF reset = '0' THEN
          next_state <= err_state;
        ELSE
          next_state <= res_state;
        END IF;
    END CASE;
  END PROCESS next_state_logic;
END dataflow;
```

A.7 Zustandsspeicher

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;
USE WORK.control_types.ALL;

ENTITY fsm_switch IS
  PORT (
    phi1          : IN  std_logic;   -- clock input
    reset         : IN  std_logic;   -- reset input
    next_state    : IN  fsm_states;  -- the next state
    current_state : OUT fsm_states); -- the current state

END fsm_switch;

ARCHITECTURE dataflow OF fsm_switch IS
BEGIN
  state_switch : PROCESS(phi1, reset)
  BEGIN
    IF (phi1'event AND phi1 = '1') THEN
      IF reset = '1' THEN
        current_state <= res_state;
      ELSE
        current_state <= next_state;
      END IF;
    END IF;
  END PROCESS state_switch;
END dataflow;
```

A.8 Ergebnisfunktion

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;
USE WORK.control_types.ALL;

ENTITY fsm_output IS

  PORT (
    current_state : IN  fsm_states;                -- the current state
    s1_enab       : OUT std_logic_vector(0 TO 3); -- select s1 source
    s2_enab       : OUT std_logic_vector(0 TO 4); -- select s2_source
    dest_enab     : OUT std_logic_vector(0 TO 3); -- select destination
    alu_op_sel    : OUT alu_func_type;            -- alu operation
    const_sel     : OUT std_logic_vector(0 TO 1); -- select constant
    rf_op_sel     : OUT std_logic_vector(0 TO 1); -- select reg file op.
    immed_sel     : OUT std_logic;                -- select immed. from ir
    mem_ctrl      : OUT std_logic_vector(0 TO 5); -- memory control lines
    state_number  : OUT fsm_state_numbers         -- state numbers
    );
END fsm_output;


ARCHITECTURE dataflow OF fsm_output IS

BEGIN
  output_logic : PROCESS (current_state)
  BEGIN
    CASE current_state IS

      WHEN res_state =>
        s1_enab         <= s1_const;
        s2_enab         <= s2_none;
        alu_op_sel      <= alu_pass_s1;
        dest_enab       <= dest_res;
        const_sel       <= const_00;
        rf_op_sel       <= rfop_none;
        immed_sel       <= imm_dcare;
        mem_ctrl        <= mem_none;
        state_number    <= res_state_no;

      WHEN fetch =>
        s1_enab         <= s1_none;
        s2_enab         <= s2_none;
        alu_op_sel      <= alu_dcare;
        dest_enab       <= dest_none;
        const_sel       <= const_dcare;
        rf_op_sel       <= rfop_none;
        immed_sel       <= imm_dcare;
        mem_ctrl        <= mem_fetch;
        state_number    <= fetch_no;
```

```
WHEN dec_pcinc4_ab =>
  s1_enab         <= s1_const;
  s2_enab         <= s2_pc;
  alu_op_sel      <= alu_s1_add_s2;
  dest_enab       <= dest_pc;
  const_sel       <= const_04;
  rf_op_sel       <= rfop_ab_rf;
  immed_sel       <= imm_dcare;
  mem_ctrl        <= mem_none;
  state_number    <= dec_pcinc4_ab_no;

  ------------------------
  -- state "sub":
  ------------------------
WHEN sub           =>
  --
  -- add: c <- a - y  / y : b or immed (Rtype: B, Itype: immed)
  --                    depends on rd_s2_addr_sel
  ------------------------
  -- available constants:
  -- s1_a, s1_mdr, s1_immed, s1_const, s1_none
  --
  -- select S1 bus source:
  s1_enab      <= s1_a;
  ------------------------
  -- available constants:
  -- s2_b, s2_immed, s2_Y, s2_pc, s2_mar,
  -- s2_const, s2_none
  --
  -- select S2 bus source:
  --  Y : b or immed (Rtype: B, Itype: immed)
  --  depends on rd_s2_addr_sel
  s2_enab      <= s2_Y;
  ------------------------
  -- available constants:
  -- alu_pass_s1, alu_pass_s2, alu_s1_add_s2,
  -- alu_s1_sub_s2,alu_s1_and_s2, alu_s1_or_s2,
  -- alu_sll_s1_s2, alu_srl_s1_s2, alu_dcare
  --
  -- select Alu operation:
  alu_op_sel   <= alu_s1_sub_s2;
  ------------------------
  -- available constants:
  -- dest_c, dest_pc, dest_mar, dest_mdr,
  -- dest_res (reset), dest_none
  --
  -- select Dest bus destination register:
  dest_enab    <= dest_c;
  ------------------------
  -- available constants:
  -- const_00, const_01, const_04, const_dcare
  --
  -- select constant:
  const_sel    <= const_dcare;
  ------------------------
```

```
        -- available constants:
        -- rfop_none, rfop_ab_rf, rfop_rf_c
        --
        -- select register file operation:
        rf_op_sel    <= rfop_none;
        ------------------------
        -- available constants:
        -- imm_s16, imm_s26,imm_dcare
        --
        -- select immediate (16 or 26 bit from IR):
        immed_sel    <= imm_s16;
        ------------------------
        -- available constants:
        -- mem_none, mem_fetch, mem_ldst, mem_lw,
        -- mem_sw, mem_error
        --
        -- select memory control:
        mem_ctrl     <= mem_none;
        ------------------------
        -- available constants:
        -- (for debugging purposes,
        --  one number for each state)
        -- res_state_no, fetch_no, dec_pcinc4_ab_no,
        -- memory_no, load_w_1_no, load_w_2_no,
        -- store_w_1_no, store_w_2_no, br_eqz_no,
        -- branch_no, jump_no, sub_no, add_no,
        -- slt_1_no, set_to_1_no, set_to_0_no,
        -- write_back_no, hlt_state_no, err_state_no
        -- and_no, or_no, sll_no, srl_no,
        --
        -- assign a number to state:
        state_number <= sub_no;
        ------------------------
              .
              .
              .
      WHEN OTHERS    =>
        --
        -- set all outputs to high impedance/ set error
        --
        s1_enab      <= s1_none;
        s2_enab      <= s2_none;
        alu_op_sel   <= alu_dcare;
        dest_enab    <= dest_none;
        const_sel    <= const_dcare;
        rf_op_sel    <= rfop_none;
        immed_sel    <= imm_dcare;
        mem_ctrl     <= mem_error;
        state_number <= err_state_no;
        --------------------------
    END CASE;
  END PROCESS output_logic;
END dataflow;
```

Anhang B Programmbeispiel

```
;-------------------------------------------------------------;
;                                                             ;
; Datei         : inc2_dec1.asm                               ;
; Datum         : Mar. 2010                                   ;
; Beschreibung  : Ein kleines Testprogram, mit dem die Folge  ;
;                 0 2 1 3 2 4 3 5 ...                         ;
;                 auf dem Display ausgegeben wird.            ;
;-------------------------------------------------------------;

        org   X"00000000"  ; legt die Programmstartadresse fest
        start init         ; markiert den ersten Befehl

; Immediate-Werte fuer Speicheroperationen:
        zero  equ  X"0000" ; Offset 0
        disp  equ  X"0FFC" ; Offset 0x0FFC
        ram   equ  X"0FF8" ; Offset 0x0FF8
;
; Es gibt nur eine Adressierungsart: Basisregister + 16 Bit Offset
; mit Vorzeichen. Demzufolge kann auf Adressen, deren Wert 15 Bit
; uebersteigt, nur zugegriffen werden, wenn zuvor das Basisregister
; entsprechend initialisiert wurde. Deshalb sind die letzten beiden
; ROM-Speicherzellen mit zwei Konstanten initialisiert. In der
; Speicherzelle mit der Adresse 0x0000_0FFC steht die Adresse fuer
; das Display Latch:
;  M[0x0000_0FFC] = 0x2000_0000
; und in Speicherzelle mit der Adresse 0x0000_0FF8 die Basisadresse
; des RAM:
; M[0x0000_0FF8] = 0x1000_0000

init:
        lw.i    r31, disp(r0)   ; r31     <- M[0x0000_0FFC]
        sw.i    zero(r31), r31  ; Display <- 0x2000_0000
        slt     r2, r0, r31     ; r2      <- 1 wegen r0 < r31
        sub     r1, r0, r2      ; r1      <- r0 - r2
        sub     r3, r1, r2      ; r3      <- r1 - r2
        slt     r4, r0, r0      ; r4      <- 0
label_1 sub     r4, r4, r3      ; r4      <- r4 - r3
        sw.i    zero(r31), r4   ; Display <- r4
        sub     r4, r4, r2      ; r4      <- r4 - r2
        sw.i    zero(r31), r4   ; Display <- r4
        j label_1               ; springe zu label_1
        end
```

Literatur

[1] P. Bazargan et al. A Tutorial for Advanced VLSI Course: Designing the 32-bit DLX Microprocessor with the ALLIANCE CAD System. In *Proc. of the 5. Eurochip Workshop on VLSI Design*, Dresden, Germany, 1994.

[2] M. Gumm. *VHDL Modelling and Synthesis of the DLXS RISC Processor*. Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart, 1995.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 1990.

[4] J. L. Hennessy and D. A. Patterson. *Rechnerarchitektur – Analyse, Entwurf, Implementierung, Bewertung.* Friedr. Vieweg Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 1994.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 1996.