FHS
HOCHSCHULE FÜR TECHNIK, WIRTSCHAFT
UND SOZIALE ARBEIT ST.GALLEN

TECHNIK

D i p l o m a r b e i t 2 0 0 1

R I S C - P r o z e s s o r f ü r F P G A S i g n a l v e r a r b e i t u n g s s y s t e m e

Fachbereich Elektrotechnik

Diplomand: Manuel Imhof

Betreuer: Dipl. Ing. ETH Björn Schaltegger

Experte: Dipl. Ing. ETH Hans Rissi

Präsentation: 30. Oktober 2001

1 Inhaltsverzeichnis

2 Zusammenfassung

Signalverarbeitung spielt im heutigen Kommunikationszeitalter eine sehr wichtige Rolle, sei dies im Natel, in der Bildverarbeitung oder im Automobilbereich. Um Signalverarbeitungsalgorithmen zu lösen stehen spezielle digitale Signalprozessoren (DSPs) zur Verfügung. In neuen Entwürfen von Signalverarbeitungssystemen werden vermehrt auch RISC-Prozessoren eingesetzt.

Die vorliegende Diplomarbeit beschäftigt sich mit dem Design eines solchen RISC-Prozessors für die Implementierung auf einem FPGA. Das Ziel bestand nicht nur darin, eine RISC-Core zu entwickeln, sondern auch die Möglichkeiten dieser zukunfts-weisenden Technologie zu zeigen. Aus diesem Grund werden in dieser Arbeit auch Erweiterungen für die RISC-Prozessorcore und Ideen für Systemumgebungen diskutiert.

Ausgehend von den Grundprinzipien eines Prozessors wurde eine 16-Bit Harvard-Register-Architektur mit einem 4 Phasen-Zyklus entworfen. Das für RISC-Prozessoren typische Phasenpipelining wurde vom Modell bis hin zur konzeptionellen Ebene analysiert. Als spezielle Einheit für die Signalverarbeitung wurde der RISC-Core ein Parallel-Multiplizierer beigefügt.

Das für Prozessoren sehr wichtige Zeitverhalten wurde anhand einer Post-Route-Simulation untersucht. Die Post-Route-Simulation verwendet die physikalischen Eigenschaften des eingesetzten FPGAs.

Anschliessend wurde ein Assembler und ein Loader entwickelt, welche die System-umgebung komplettieren. Der Assembler kennt die grundlegenden Direktiven und übersetzt den Quell- in den Maschinencode. Damit ist es möglich eine Quellcodedatei zu assemblieren und die erzeugte Codedatei mit dem Loader über den PCI-Bus in den Programmspeicher zu laden.

3 Problem- und Aufgabenstellung

3.1 Problemstellung

Field Programmable Gate Arrays (FPGAs) eröffnen neue Möglichkeiten in der bisher von Hardwarelösungen mit Signalprozessoren (DSPs) oder Applikationsspezifischen Integrierten Schaltungen (ASICs) dominierten Signalverarbeitung. Insbesondere die neueste Generation von Field Programmable Gate Arrays (FPGAs) ermöglicht eine anwendungsspezifische Entwicklung vollständiger Signalverarbeitungssysteme, die auf einem oder mehreren FPGAs implementiert werden können. Die Rekonfigurierbarkeit der FPGA-Hardware erlaubt ausserdem eine schnelle Anpassung und Umsetzung der benötigten Signalverarbeitungsalgorithmen auf das jeweilige Problem.

Eine Hauptkomponente eines solchen Signalverarbeitungssystems ist häufig ein auf die Problemstellung angepasster RISC-Prozessor mit seinen für die Peripheriebausteine (z.B. AD/DA-Wandler) notwendigen Schnittstellen. Im Rahmen dieser Diplomarbeit soll ein solcher RISC-Prozessor für FPGA Signalverarbeitungssysteme entworfen und implementiert werden.

3.2 Aufgabenstellung

Es ist ein RISC-Prozessor für FPGA Signalverarbeitungssysteme inklusive zugehöriger Schnittstellen für Peripheriebausteine zu entwerfen und auf der Xilinx PCI-Karte der Fachhochschule für Technik St.Gallen zu implementieren. Das Design des Prozessors ist in VHDL durchzuführen. Es ist ein einfacher Assembler für den Prozessor zur Verfügung zu stellen. Die Funktionsweise des Prozessors ist anhand einer Signalverarbeitungsaufgabe zu demonstrieren.

4 Einleitung

Seit der Einführung des ersten Mikroprozessors vor 30 Jahren haben sich diese in unserem Alltag stark verbreitet. Die Vielfalt reicht von 4-Bit Nibblerechner für einfachste Steuerungsaufgaben, über 64-Bit Prozessoren für 3-D fähige Spielkonsolen bis hin zu Grossrechnern für wissenschaftliche Berechnungen und Simulationen. Ein Auskommen ohne sie ist heute praktisch nicht mehr denkbar. In einem modernen Auto sind bis zu 300 Prozessoren im Einsatz.

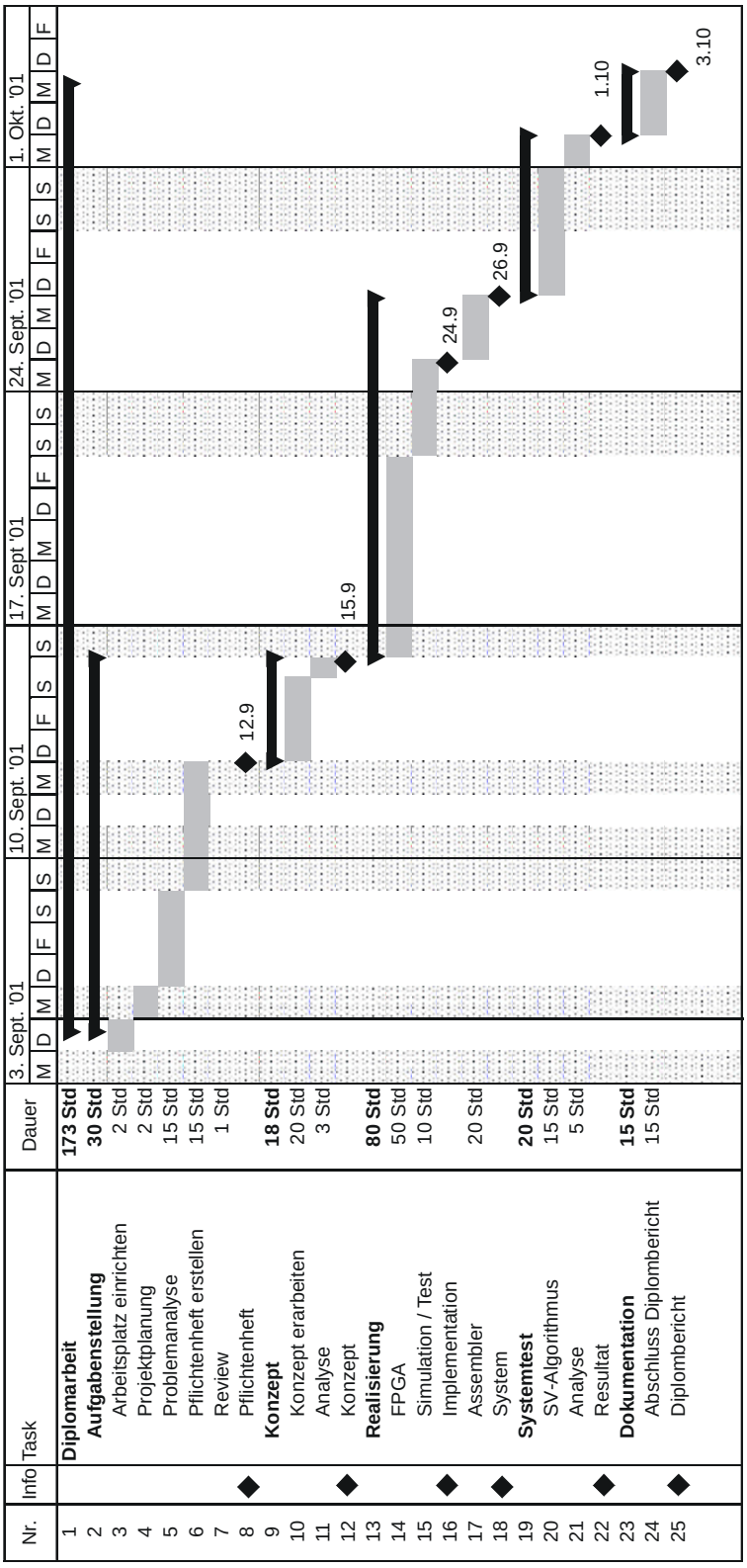
Mikrocontroller sind einem Prozessorsystem nachempfundene, programmierbare Bausteine. Sie brauchen keine externen Komponenten wie zum Beispiel Speicher- oder Schnittstellenbausteine. Man spricht auch vom Single-Chip-Computer.

Man unterscheidet Prozessoren und Controller grundsätzlich nach ihrer parallel verarbeitbaren Wortbreite von 4, 8, 16, 32 oder 64 Bit. Weiter gibt es Standard- und kundenspezifische Ausführungen. Die Standardtypen können weitgehend als anwendungsunabhängig betrachtet werden. Kundenspezifische Prozessoren (ASPs) enthalten zusätzlich noch applikationsabhängige Komponenten. Typische Beispiele dafür sind die in sehr grosser Stückzahl produzierten 4-Bit Controller für Armbanduhren und Taschenrechner, 8-Bit-Rechner für Fernsehgeräte oder 32-Bit-Prozessoren für Funktelephone und Modems. In der Signalverarbeitung wird hauptsächlich mit digitalen Signalprozessoren (DSPs) oder RISC-Prozessoren gearbeitet. Dies sind Maschinen, welche auf die Abarbeitung von digitalen Algorithmen (z.B. Filterung, Codierung) getrimmt sind.

Um diesen unterschiedlichsten Anforderungen Rechnung zu tragen, wäre eine Plattform wünschenswert, welche auf die gewünschte Aufgabe angepasst werden kann.

Eine völlig andere Klasse von programmierbaren Systemen - die feld-programmierbaren Bausteine - sind technisch mittlerweilen soweit gereift, dass sie für eine Integration von Single-Chip-Computern geeignet sind. Eine applikations-spezifische Lösung ist somit möglich, ohne eine Fabrik im Hintergrund zu haben.

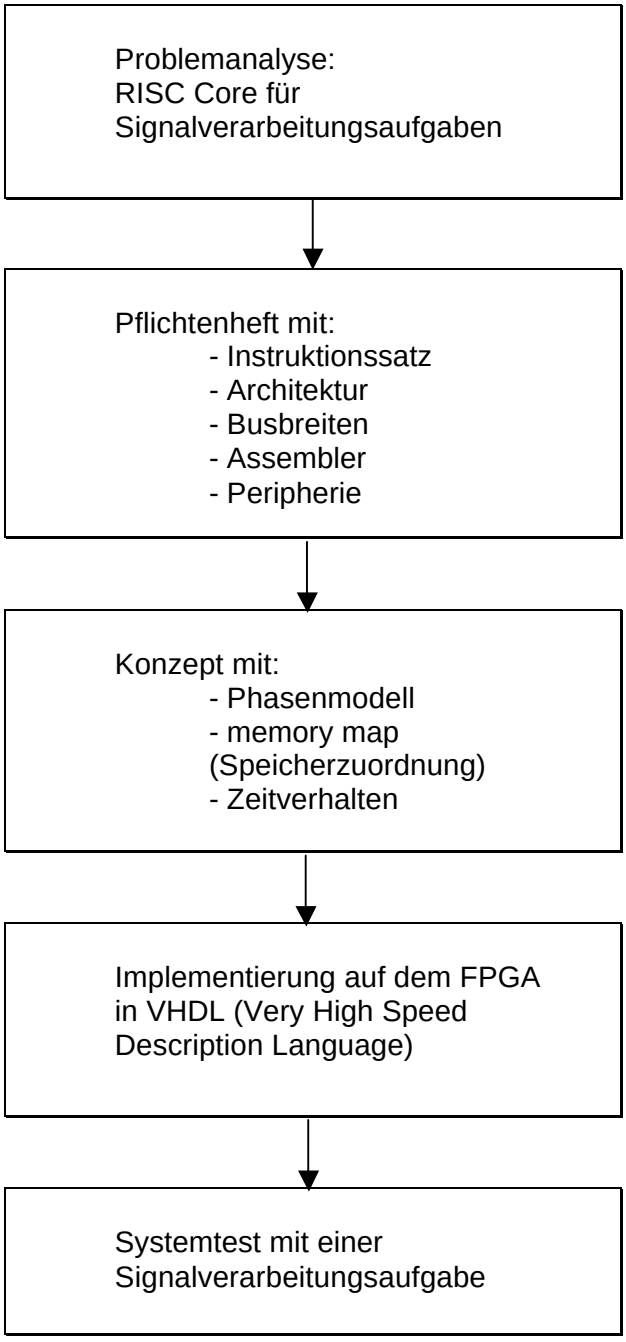
5 Zeitplan


Nr.
Info
Task
Dauer
3. Sept. '01
10. Sept. '01
17. Sept '01
24. Sept. '01
1. Okt. '01
M D M D F S S M D M D F S S M D M D F S S M D M D F
1 Diplomarbeit 173 Std
2 Aufgabenstellung 30 Std
3 Arbeitsplatz einrichten 2 Std
4 Projektplanung 2 Std
5 Problemanalyse 15 Std
6 Pflichtenheft erstellen 15 Std
7 Review 1 Std
8 Pflichtenheft
9 Konzept 18 Std
10 Konzept erarbeiten 20 Std
11 Analyse 3 Std
12 Konzept
13 Realisierung 80 Std
14 FPGA 50 Std
15 Simulation / Test 10 Std
16 Implementation
17 Assembler 20 Std
18 System
19 Systemtest 20 Std
20 SV-Algorithmus 15 Std
21 Analyse 5 Std
22 Resultat
23 Dokumentation 15 Std
24 Abschluss Diplombericht
25 Diplombericht
12.9
15.9
24.9
26.9
1.10
3.10

6 Entwicklungsphasen beim Entwurf einer Prozessor-Core

Wird ein Verarbeitungsblock mit einer Hardware-Beschreibungssprache realisiert, so spricht man von einem Core. Die Vielfalt solcher Verarbeitungsblöcke reicht von Signalverarbeitungsalgorithmen (z.B. FIR-Filter, Fast Fourier Transformation) bis hin zu Prozessoren.

Die folgende Grafik zeigt den Designprozess, der für die Entwicklung einer Prozessor-Core erarbeitet wurde.



7 Problemanalyse

Prozessoren, wie sie heute gebaut und eingesetzt werden, basieren alle auf dem gleichen Grundprinzip. Die Daten werden über eine Eingabeeinheit oder aus dem Speicher gelesen, in der Zentraleinheit (CPU) verarbeitet und über die Ausgabeeinheit ausgegeben oder wieder in den Speicher zurückgeschrieben. Die Zentraleinheit wird aufgeteilt in eine Kontrolleinheit (CU) und eine arithmetische, logische Einheit (ALU). Abbildung 1 zeigt diesen Sachverhalt auf.

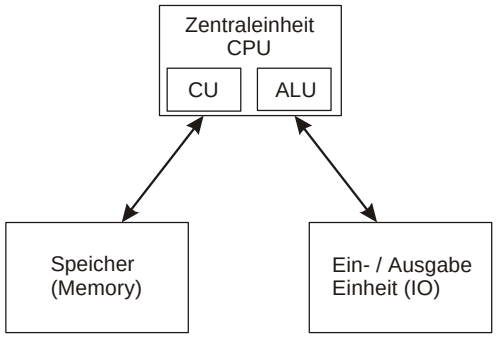


Abbildung 1: Prinzipieller Aufbau eines Prozessors

Der Datenaustausch zwischen den einzelnen Blöcken übernimmt ein Bussystem. Das Bussystem ist zuständig dafür, dass die Daten an den richtigen Ort gelangen. Der Adressbus bestimmt den Ausgangsort, beziehungsweise den Zielort für die Daten, welche auf dem Datenbus fliessen. Den zeitlichen Ablauf und die Koordination übernimmt der sogenannte Kontrollbus.

Um eine Verarbeitung zu ermöglichen, müssen die Daten zwischengespeichert werden. Die Register, welche diese Aufgabe übernehmen, sind nicht über den Datenbus, sondern direkt mit der Zentraleinheit verbunden, um langwierige Zugriffe zu vermeiden.

Abbildung 2 zeigt den strukturierten Aufbau eines Prozessors mit Bussystem und Register.

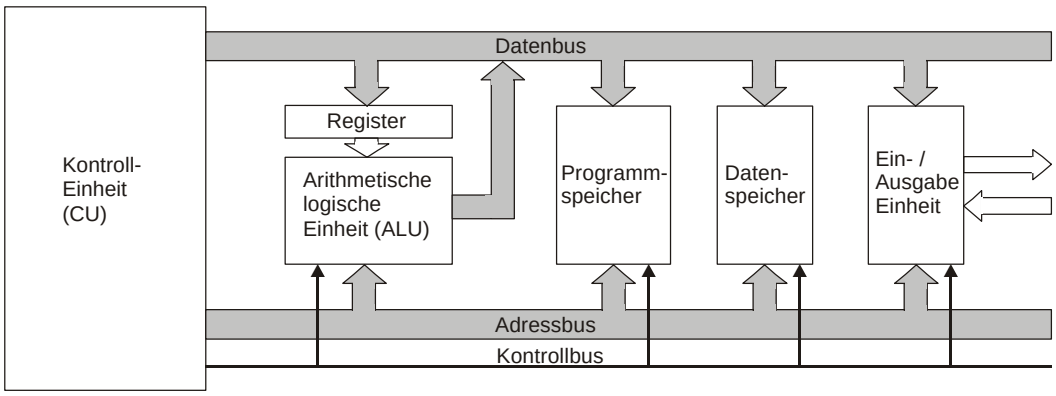


Abbildung 2: Bussystem eines Prozessors

Anhand der Busstruktur wird die Prozessorarchitektur unterschieden zwischen der von Neumann- und der Harvard-Architektur. Bei der Harvard-Architektur sind alle Busse doppelt ausgeführt: einmal für die Daten und einmal für den Programmcode. Im Vergleich zur von Neumann-Architektur, wo sowohl Daten wie auch Programmcode auf dem gleichen Bus fliessen, führt diese Duplizierung zu einem geringeren Verkehr auf den einzelnen Pfaden. Diese Parallelisierung führt zu einer Steigerung der Verarbeitungsgeschwindigkeit, benötigt jedoch mehr Chipfläche und verringert die Flexibilität, da nicht mehr alle Blöcke am selben Bus hängen.

General Purpose Prozessoren weisen normalerweise eine von Neuman-Architektur auf, da diese klein und billig sein sollten. Power- und Signalprozessoren arbeiten mit der Harvard-Architektur. Da diese eine grössere Chipfläche in Anspruch nimmt, ist der Preis dementsprechend höher.

Prozessor	Hersteller	Anwendung	Architektur
80C51	Infineon	General Purpose	Von Neuman
68HC11	Motorola	General Purpose	Von Neuman
Pentium I..IV	Intel	PC	Von Neuman
Tiger SHARC	Analog Device	Digitale Signalverarbeitung	Harvard
Alpha	DEC	Hochleistung	Harvard
Power PC	Motorola	Hochleistung	Harvard

Tabelle 1: Typische Beispiele

7.1 Prozessorarchitektur

Der innere Aufbau, der zeitliche Ablauf sowie das Hardware-Software-Interface eines Prozessors werden durch das Auslegen der folgenden Modelle bestimmt:

- Programmiermodell
- Speichermodell
- Form der Interaktionen
- Phasenmodell

Alle vier Komponenten haben direkten Einfluss aufeinander. Unter dem Hardware-Software-Interface versteht man die Funktionalität des Prozessors, welche direkt durch den Programmierer mittels Assemblerbefehlen genutzt werden kann.

7.1.1 Programmiermodell

Das Programmiermodell beinhaltet die direkt beeinflussbaren Register, die Maschinenbefehle (Instruction set) mit ihrer mnemotechnischen Beschreibung und dem Operationscode (Opcode) sowie die Adressierungsarten.

7.1.1.1 Adressierungsarten

- Implizite Adressierung (Registeradressierung)
  Diese Adressierungsart ist einfach und sehr weit verbreitet. Im Opcode ist das Ziel des Zugriffes direkt bestimmt.

- Registerdirekte Adressierung
  Im Vergleich zur impliziten Adressierung wird hier das Ziel explizit angegeben. Da die Codierung des jeweiligen Registers sehr kompakt möglich ist, kann sie als Teil des Opcodes erfolgen. Somit geht die Registerdirekte zur impliziten Adressierung über.
  Bsp: Registerkopie, das Zielregister (A) ist im Opcode festgehalten.
  MOV A, B;

- Registerindirekte Adressierung
  Der Wert des zweiten Operanden wird mittels Register referenziert. Der Inhalt des Referenzregisters wird während der Load-Phase auf den Adressbus gelegt. Der Inhalt der angesprochenen Speicherzelle wird ins Zielregister geladen.

- Unmittelbare Adressierung (immediate)
  Bei dieser Adressierungsart folgt direkt auf den Opcode ein Literal (Konstante).
  Bsp: Lade den Akku mit dem Wert 10.
  MOV A, #10;

- Speicherdirekte Adressierung

  Die Referenzadresse, welche den Operanden enthält bzw. den Operanden aufnehmen soll, folgt direkt auf den Opcode. In der Literatur wird diese Adressierung normalerweise als *direct* bezeichnet, da die Registerdirekte mit der impliziten Adressierung korrespondiert und somit keine Verwechslung möglich ist.

  Bsp: Kopiere den Inhalt der Speicherstelle 200h in das Register D.
  MOV D, 200h;

- Speicherindirekte Adressierung

  Von dieser Adressierungsart wird normalerweise abgesehen, da zweimalig auf Adressen und erst im dritten Zyklus auf den Operanden zugegriffen werden kann.

- Indizierte Adressierung

  Ähnlich der Speicherdirekten ist die indizierte Adressierung. Die Referenzadresse wird aus einer Basisadresse und einem Offset berechnet. Diese Art des Speicherzugriffes eignet sich für Tabellenverarbeitungen.
  Befindet sich der Offset in einem internen Register, wird die Anzahl der Speicherzyklen nicht erhöht!
  Bsp: MOV A, 100h+B

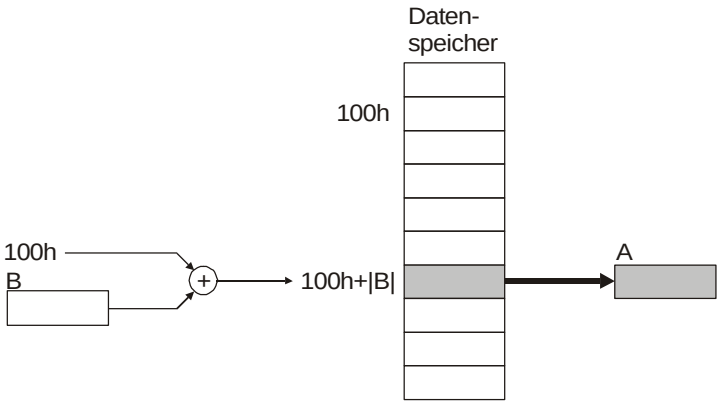


Abbildung 3: Indizierte Adressierung

- Absolute Adressierung

  Die absolute Adressierung wird im Rahmen von Sprungbefehlen genutzt. Der Programmcounter wird direkt mit dem Wert des Operanden geladen. Diese Adressierungsart benötigt exakt einen Speicherzugriff, um den Operanden zu laden.

  Bsp: Springe an die Stelle A020h
  JUMP A020h;

- Absolut indirekte Adressierung

  Das Argument der indirekten Adressierung wird als Adresse interpretiert, an welcher der Operand gespeichert ist.

- Relative Adressierung

  Die relative Adressierung wird insbesondere für Verzweigungsbefehle genutzt. Der Programmcounter wird um den Wert des Operanden variiert.

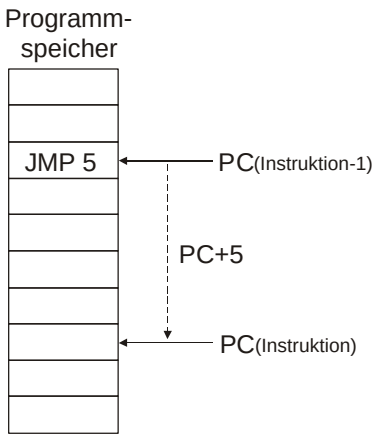


Abbildung 4: Relative Adressierung

Die Addition des Programmcounter mit dem Operanden ist vorzeichenbehaftet, damit auch Rückwertssprünge realisiert werden können.

7.1.2 Speichermodell

Im Speichermodell werden die Busbreiten festgelegt. Die Adressbusbreite wird durch den maximal ansprechbaren Speicherbereich bestimmt. Der Code- und der Datenzugriff werden bestimmt durch die Datenbusbreite. Bei der Harvardstruktur sind Programm- und Datenbus voneinander getrennt. Es kann also eine unterschiedliche Breite gewählt werden. Die Programmbusbreite wird dann durch das Instruction Set bestimmt.

⇒ Für ein Instruction Set mit 32 Befehlen ist eine minimale Programmbusbreite von $n = \log_2(32) = 5$ nötig.

In der Praxis wird der Programmbus oft breiter gewählt. Somit können gleichzeitig mit dem Instruktionswort auch noch Daten übertragen werden.

◆

Beispiel 1: Programmbusbreite für ein Instruction Set mit 60 Befehlen

Für ein Instruction Set mit 60 Befehlen und gleichzeitiger Übertragung einer 8-Bit Konstanten bei der unmittelbaren Adressierung wird ein

$$n = \log_2(60) + 8 = 14$$

Bit breiter Programmbus benötigt.

◆

Wird von einem 8-, 16- oder 32-Bit Prozessor gesprochen, meint man mit der Bitanzahl die Datenbusbreite. Die Datenbusbreite wird als Potenz mit Basis 2 gewählt, da in der Praxis nur 8- und 16-Bit Speicherbausteine erhältlich sind.

7.1.3 Form der Interaktionen

Mit Form der Interaktion ist das Verfahren gemeint, mit welchem der Operandenzugriff ausgeführt wird. Bei einer *Stack-Architektur* werden die Quell- und Zieloperanden auf dem Stack gespeichert und auch von dort zugegriffen. Die *Akkumulator-Architektur* zeichnet sich durch ein zentrales Register aus, welches implizit codiert ist und als Quell- und Zielregister dienen kann. Im Rahmen der *Register-Architekturen* werden alle Operanden explizit angegeben, egal ob es sich um ein Register oder eine Speicheradresse handelt.

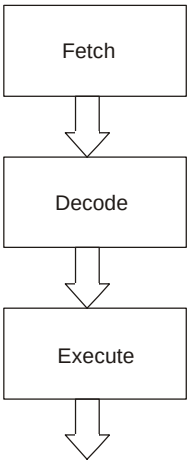
Typ	Vorteile	Nachteile
Stack	Keine explizite Angabe von Operanden notwendig, daher minimale Opcodelänge.	Erheblicher Datenfluss vom und zum Stack nötig. Deshalb wachsende Codegrösse.
Akkumulator	Einfachste Architektur, es ist sogar ein Verzicht auf einen Stack möglich. Der Zieloperand bei Arithmetikoperationen kann implizit angegeben werden.	Der Akkumulator ist das zentrale Register, deshalb wird eine Zwischenspeicherung der Resultate nötig. Erheblicher Datenfluss und wachsende Codegrösse.
Register-Speicher	Daten können ohne Ladezugriff für Operationen genutzt werden.	Ladezugriffe im Speicherbereich benötigen wesentlich mehr Zeit. Daher ggf. Verzögerung gegenüber Registerzugriff.
Register-Register	Schnellstmögliche Operation durch Registeroperanden.	Opcodelänge wächst durch explizite Operandenangabe.

Tabelle 2: Verfahren für Operandenzugriff

In der Praxis finden nur die Akkumulator- und die Register-Architektur (meist kombinierte Register-Speicher- und Register-Register-Architektur) Verwendung. Beide werden im Kapitel 7.2 genauer behandelt.

7.1.4 Phasenmodell

Das Phasenmodell bestimmt die Verarbeitungsgeschwindigkeit des Prozessors. Es legt die zeitliche Abarbeitung der Befehlsphasen und die Anzahl Takte pro Maschinenzyklus fest. Die Befehlsausführung kann in drei grundlegende Vorgänge unterteilt werden.



Fetch:	Lesen des Befehles und Ablegen im Instruktionsregister.
Decode:	Umsetzen des Maschinenbefehls.
Execute:	Der Sequenzer verarbeitet den Maschinenbefehl und führt die geforderten Aktionen aus.

Abbildung 5: Vorgänge der Befehlsausführung

Um die Verarbeitungsgeschwindigkeit zu steigern, gibt es unterschiedliche Methoden. Wird die Taktfrequenz erhöht, steigt linear dazu die Leistung des Prozessors. Auftretende Störungen und Wärmeprobleme können diese „einfache“ Methode sehr schnell erschweren.

Eine elegante Methode, wie sie auch bei RISC Prozessoren angewandt wird, ist das Pipelining. Beim Pipelining werden unterschiedliche Phasen der Befehle zeitlich übereinander geschoben. Dies ist nur möglich, wenn die verschiedenen Phasen auch verschiedene Hardwareblöcke beanspruchen.

Digitale Signalprozessoren können die meisten Befehle in einem Taktzyklus abarbeiten. Dies liegt an der starken, zusätzlichen Parallelisierung in der Hardwarestruktur.

7.1.4.1 Pipelining

Abbildung 6 zeigt das Phasenpipelining anhand von MIPS-Instruktionen. MIPS ist der klassische Befehlssatz für RISC Prozessoren, welcher von David A. Patterson und John L. Hennessy ([L2], [L5]) an der Universität Berkeley entworfen wurde.

1. Instruktion aus dem Speicher laden (IF).
2. Register lesen, während die Instruktion dekodiert wird (Reg).
3. Operation ausführen oder eine Adresse berechnen (EX).
4. Zugriff auf einen Operanden im Datenspeicher (DA).
5. Resultat in ein Register schreiben (WB).

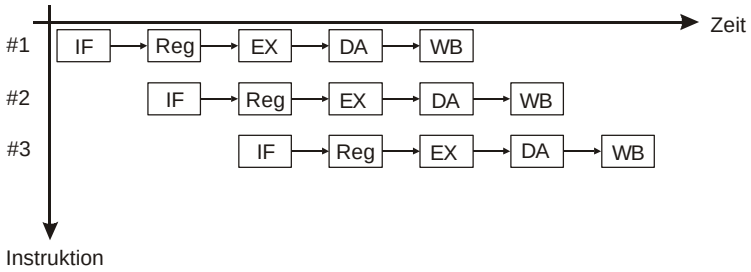


Abbildung 6: Phasenpipelining

☞

Pipelining optimiert die Abarbeitungsfolge der aufeinanderfolgenden Instruktionen.

$$Zeit\ zwischen\ den\ Instruktionen_{pipelined} = \frac{Zeit\ zwischen\ den\ Instruktionen_{nonpipelined}}{Anzahl\ Phasenüberlappungen}$$

Die Schwierigkeit beim Entwurf von Pipelining ist die zeitliche Verschachtelung der Instruktionen. Tritt ein gleichzeitiger Zugriff von zwei Befehlen auf die gleiche Ressource auf, spricht man von Hazards.

7.1.4.2 Pipelinehazards

Werden die einzelnen Instruktionen zeitlich übereinandergelegt, kann es passieren, dass zwei Phasen auf die gleichen Ressourcen zugreifen. (z.B. *fetch* und *store* bei einer von Neumann-Architektur, die beide den Datenbus benötigen!) Dieses Vorkommnis wird als Pipeline Hazard bezeichnet. Es gibt drei mögliche Problemfälle:

- In der Struktur
- Im Kontrollfluss
- Im Datenfluss

7.1.4.2.1 Strukturhazard

Strukturhazards treten auf, wenn die Hardware eine Kombination von Instruktionen nicht unterstützen kann. Jede Phase muss explizit einer Ressource zuteilbar sein.

7.1.4.2.2 Kontrollflusshazard

Kontrollflusshazards treten hauptsächlich bei Sprungbefehlen auf. Bis die Phase des Sprunges erreicht wird, sind bereits neue Instruktionen geladen worden. Abhilfe können Wartezyklen – sogenannte *stalls* – schaffen. Jeder eingefügte Wartezyklus erhöht aber das CPI (clocks per instruction) und vermindert dadurch die Verarbeitungsgeschwindigkeit des Prozessors.

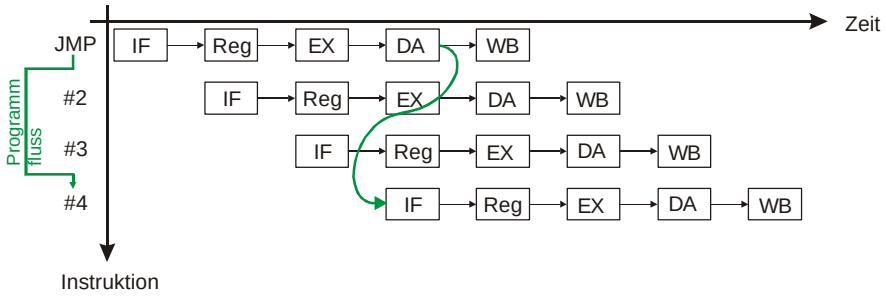


Abbildung 7: Kontrollflusshazard

7.1.4.2.3 Datenflusshazard

Durch gemeinsames Nutzen von Registern können Datenabhängigkeiten von aufeinander folgenden Befehlen entstehen. Ein Zerstören der Daten ist nicht ausgeschlossen. Dieses Problem wird Datenflusshazard genannt.

7.2 RISC Prozessor

Reduced **I**nstruction **S**et **C**omputer (RISC) basieren, wie der Name schon sagt, auf einem stark minimierten Befehlssatz. Im Vergleich zu einem CISC (Complex Instruction Set Computer) Prozessor stehen nur noch die elementaren Befehle zur Verfügung. Diese Reduzierung hat erhebliche Auswirkung auf die Assemblerprogrammierung. Sie wiederspiegelt sich aber auch in einer andersartigen Architektur.

Die komplexen Operationen (z.B. Multiplikation) eines CISC Prozessors werden durch kleine Mikroprogramme ausgeführt. Dies führt zu einer stark vereinfachten Programmierung und zu einem übersichtlicheren Programmcode. Um die Busstruktur möglichst flexibel zu halten, basieren CISC Prozessoren auf der von Neumann-Akkumulator-Architektur.

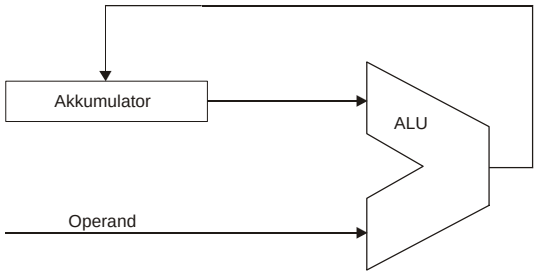


Abbildung 8: Akku/ALU Einheit

Aufgrund des minimierten Befehlssatzes bei RISC Prozessoren, kann eine sehr gute Koordination der Funktionsabläufe erzielt werden. Dies führt zu dem bereits oben erwähnten Phasenpipelining. Um eine genügende Parallelisierung des Datenflusses zu erreichen, muss von der Akkumulator-Architektur abgesehen werden. Es wird eine Harvard-Register-Architektur benötigt. Während bei der Akkumulator-Architektur immer der Akkumulator als Zielregister dient, kann bei der Register-Architektur das Zielregister explizit im Befehl angegeben werden. Somit fällt das Sichern des Akkumulators nach einem arithmetischen Befehl weg.

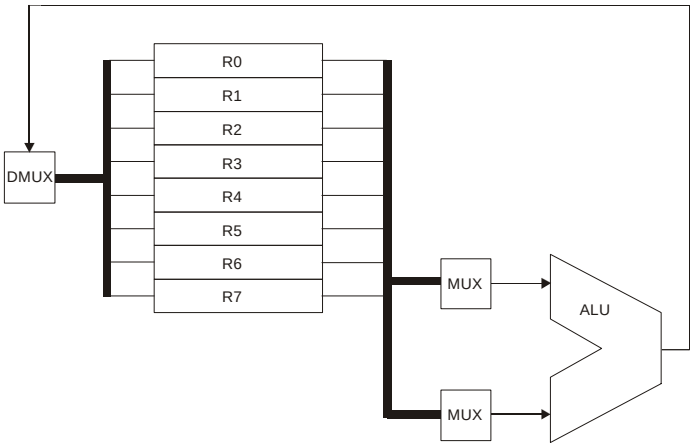


Abbildung 9: Register/ALU Einheit

7.3 Assembler

Bei der Programmierung von Prozessoren wird zur leichteren Handhabung und zum besseren Verständnis eine symbolische Schreibweise für die Befehlsdarstellung gewählt. Die der Hardware am nächsten liegende Befehlsbeschreibung ist die sogenannte Assemblersprache. Sie ist durch den Befehlssatz des Mikroprozessors geprägt, jedoch in Semantik und Syntax von der Hardware unabhängig. Die Übersetzung dieses Programmcodes übernimmt ein Übersetzungsprogramm, der Assembler. Bei einem 1 zu 1 Übersetzer entspricht ein symbolischer Befehl gerade einem Maschinenbefehl. Assemblersprachen heutiger Mikroprozessoren erlauben zum Teil auch die 1 zu n Übersetzung sogenannter Makrobefehle. Solche Makrobefehle entsprechen nicht mehr einem Maschinenbefehl, sondern bestehen aus einem mehrzeiligen Assemblercode.

Um eine sinnvolle und brauchbare Programmierung eines Prozessorsystems zu erreichen, sind zusätzliche Assembleranweisungen nötig, welche keinen korrespondierenden Maschienenbefehl besitzen. Diese Erweiterungen werden Assemblerdirektiven, Assembleranweisungen oder Pseudobefehle genannt. Assemblerdirektiven dienen neben der Speicherdeklaration auch zur Steuerung des Übersetzungsvorganges. Im Gegensatz zu den Maschinenbefehlen erzeugen Assembleranweisungen bei der Übersetzung keinen Maschinencode.

7.4 Implementierung der Prozessor-Core auf der Xilinx PCI Karte

Die Xilinx PCI Karte wurde an der FHS entwickelt[1]. Sie ist auf das Design von Signalverarbeitungssystemen in einer Hardwarebeschreibungssprache (z.B. VHDL) ausgelegt.

Abbildung 10 zeigt den Aufbau sowie die zur Verfügung stehenden Ressourcen.

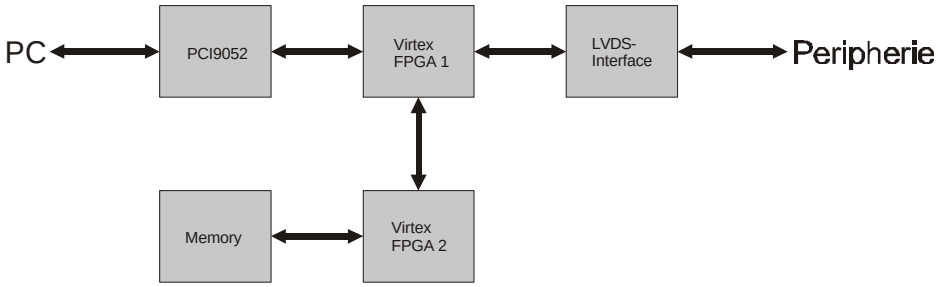


Abbildung 10: Blockschaltbild der Xilinx PCI Karte

PCI-Bus	V2.1, 33MHz, 32Bit
LVDS-Schnittstelle	Input: 12 Bit Output: 12 Bit Control: 6 Bit
Memory	4 x 128kByte RAM
FPGA1	Virtex V300 / V400, 50MHz
FPGA2	Virtex V50, 50MHz

Tabelle 3: Übersicht über die Ressourcen der Xilinx FPGA Karte

Wird die Xilinx PCI Karte als Plattform für einen Prozessor benützt, bietet sich die LVDS Schnittstelle als Ports für externe Peripherien an. Die LVDS Schnittstelle besitzt eine Bandbreite von 200MHz und lässt sich somit mit der selben Taktfrequenz wie die beiden FPGA's betreiben. Eine weitere, elegante Methode, um externe Peripherien anzusteuern, ist eine serielle, synchrone Schnittstelle. Durch den Freiheitsgrad der FPGA's lassen sich über die LVDS Schnittstelle diverse Standards wie zum Beispiel SPI oder IIC[2] realisieren

[1] Xilinx PCI Karte, entwickelt von U.Broger für die Diplomarbeit 2000
[2] SPI bzw. IIC sind eingetragene Markenzeichen der Firma Motorola bzw. Philips

8 Pflichtenheft

8.1 Anforderungsprofil

Bereich	Anforderung	Muss / Wunsch
Plattform	Einsatz der „Xilinx PCI Karte der FHS“	Muss
RISC Architektur	Harvard-Register-Architektur mit grundlegendem Instruktionssatz	Muss
Pipelining	Anpassen der Struktur und des Sequenzers für ein 2- oder 4-Phasen Pipelining	Wunsch
Assembler	1 zu 1 Übersetzer mit Pseudobefehlen zur Programmsteuerung	Muss
Debugging	Auslesen und ändern vom Datenspeicher	Wunsch
Peripherie	SPI Schnittstelle zum Betreiben einer externen, analogen Schnittstelle	Muss
Erweiterung	Für Digitale Signalprozessor-Core wie z.B. Multiply accumulate unit (MAC), Barrelshifter	Wunsch
Systemtest	Durch einfache Signalverarbeitungsaufgabe wie z.B. FIR-Filter	Muss

Tabelle 4: Anforderungsprofil

Das Pflichtenheft wird anhand des Anforderungsprofils und nach den Kriterien, wie sie bei der Problemanalyse untersucht wurden, erstellt. Es unterteilt sich in folgende Bereiche:

- Instruktionssatz
- Architektur
- Pipelining
- Peripherie
- Assembler

Der Systemtest durch eine einfache Signalverarbeitungsaufgabe wird im Kapitel 11 behandelt.

8.2 Instruktionssatz

Wie es sich für RISC Prozessoren gehört, wird der Instruktionssatz auf ein Minimum gebracht. Folgende Befehle sollen implementiert werden:

Transferbefehle: Sind abhängig von den Adressierungsarten. Zu den verwendeten Adressierungsarten werden die dazugehörigen Transferbefehle realisiert. Dazu kommen PUSH und POP für den Stackzugriff.

Arithmetische und logische Befehle:
Addieren, Subtrahieren, Vergleichen, Inkrementieren, Dekrementieren, logisch und, logisch oder, logisch exklusiv oder, schieben und rotieren.

Flagbefehle: Das Carry- und das Interruptflag sollen direkt beeinflussbar sein.

Kontrollflussbefehle: CALL / RET / RETI für Unterprogramm- und Interruptablauf, NOP um einen Maschinenzyklus zu verzögern,
absoluter Programmsprung mit JMP und
konditionelle Sprünge in Abhängigkeit des jeweiligen Flag.

8.2.1 Adressierungsarten

Die Operanden bei einer Register-Architektur werden explizit angegeben. Dies führt zur registerdirekten Adressierung. Weiter wird die unmittelbare (immediate) sowie die registerindirekte und die indizierte Adressierung für Speicherzugriffe unterstützt.

8.2.2 Binäre Kodierung

Bit 14 und 15 geben den Operationstyp an:

00 = Transferbefehle

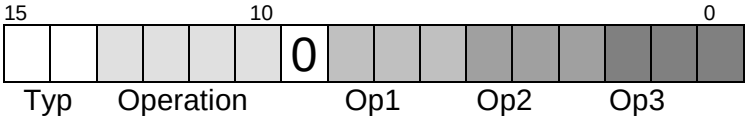
01 = Arithmetisch / logische Befehle

10 = Flagbefehle

11 = Kontrollflussbefehle

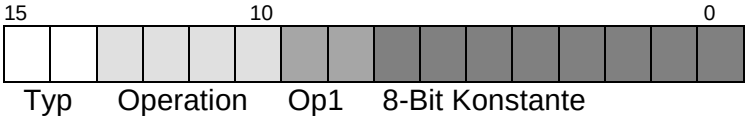
In den Bits 14 bis 10 wird die Operation kodiert. Darauf folgen die Operanden bzw. die unmittelbare Konstante.

8.2.2.1 Implizit / Register indirekt



Pro Operand stehen 3 Bits zur Verfügung. Das heisst, es ist eine maximale Registertiefe von 8 ansprechbar.

8.2.2.2 Unmittelbar / Indiziert



Als Operand 1 können nur Register 0 bis 3 angesprochen werden, da für die Kodierung nur 2 Bits zur Verfügung stehen. Um die Programmdatenbus-Breite für die indizierte Adressierung nicht erweitern zu müssen, wird auf die Grundlagen der Akkumulator-Architektur zurückgegriffen: R0 wird als fixes Indexregister definiert.

☞ R0..R3 dienen als Zielregister für die unmittelbare (immediate) Adressierung. R0 wird bei der indizierten Adressierung als Indexregister verwendet.

8.2.2.3 CALL / JMP

Im Vergleich zu den konditionellen Sprungbefehlen werden der Unterprogrammaufruf (CALL) und der absolute Sprung (JMP) mit einer 10 Bit Konstanten angegeben. Dies erlaubt Sprünge und die Platzierung der Unterprogramme im ganzen Programmspeicherbereich.

⇒ Der komplette Befehlssatz ist im Anhang zu finden.

8.3 Prozessor-Architektur

Bei der Problemanalyse wurde deutlich, dass eine Akkumulator-Architektur ungeeignet ist, um einen RISC-Prozessor zu entwerfen. Zur Unterstützung eines eventuellen Phasenpipelinings wird eine Harvard-Register-Architektur eingesetzt. Diese bietet aufgrund des geringsten Datenflusses auch die geringste Gefahr vor Struktur- und Datenflusshazards.

8.3.1 Stack

Die Initiierung von Unterprogrammen und Interrupt Anforderungen erfordert die Sicherung der Absprungadresse sowie der Register. Aus diesem Grund wird ein Stack eingeführt, welcher diese Daten vor dem Eintritt in die Routine (Unterprogramm, Interrupt) aufnimmt. Der Stack besitzt eine LIFO (Last In First Out) Struktur und gibt beim Rücksprung die gesicherten Daten zurück. Die Basiseinheit des Stacks ist der Stackpointer (SP). Die Stackpointerbreite wird auf 8 Bit beschränkt. Somit lassen sich die untersten 256 Datenwörter des Datenspeichers verwenden.

Aus der Stackpointerbreite lässt sich die maximale Verschachtelungstiefe eines Programms bestimmen:

$$Verschachtelungstiefe_{\max} = \frac{2^{Stackpointerbreite}}{zu\ sichernde\ Register} = \frac{256}{9} = 28$$

Mit „zu sichernde Register“ sind die Register 0..7, sowie die Rücksprungadresse gemeint.

8.3.2 Statusregister

Wird bei einer Addition ein Überlauf erzielt oder bei einer Subtraktion ein negatives Resultat erreicht, muss das erkannt werden. Die arithmetisch korrekte Weiterverarbeitung ist von diesen Erkenntnissen abhängig. Dazu wird ein Statusregister eingeführt, welches diese Zustände speichert und der nächsten Operation zur Verfügung stellt.

Auch für die konditionalen Sprünge wird ein Status aus einem Vergleich oder einer Berechnung benötigt. Deshalb wird zusätzlich das Zero-Flag (ZF) eingeführt.

Carry-Flag (CF):	Enthält den arithmetischen oder logischen Übertrag der jeweiligen Operation. Die exakte Beeinflussung wird im Befehlssatz angegeben.
Zero-Flag (ZF):	Das Zero-Flag wird auf 1 gesetzt, wenn die ausgeführte Operation Null ergab.
Negative-Flag (NF):	Wird für die Unterscheidung der negativen und positiven Zahlendarstellung verwendet. Das Negative-Flag ist gesetzt, wenn das Resultat der vorangegangenen Operation kleiner als Null ist. Das Resultat kann als Zweierkomplement der negativen Zahl interpretiert und für vorzeichenbehaftete Arithmetik weiterverwendet werden.
Overfolw-Flag (OF):	Dient zur Erkennung von Überläufen. Die exakte Beeinflussung wird im Befehlssatz angegeben.
Interrupt-Flag (IF):	Kennzeichnet den Status der Interruptbehandlung. Ist das IF Null, wird kein Interrupt ausgeführt.

Abbildung 11 zeigt den bitweisen Aufbau des Statusregisters.

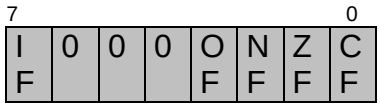


Abbildung 11: Aufbau des Statusregisters

8.3.3 RISC_Core I Architektur

Abbildung 12 zeigt die aus den obigen Überlegungen heraus erarbeitete Prozessorarchitektur.

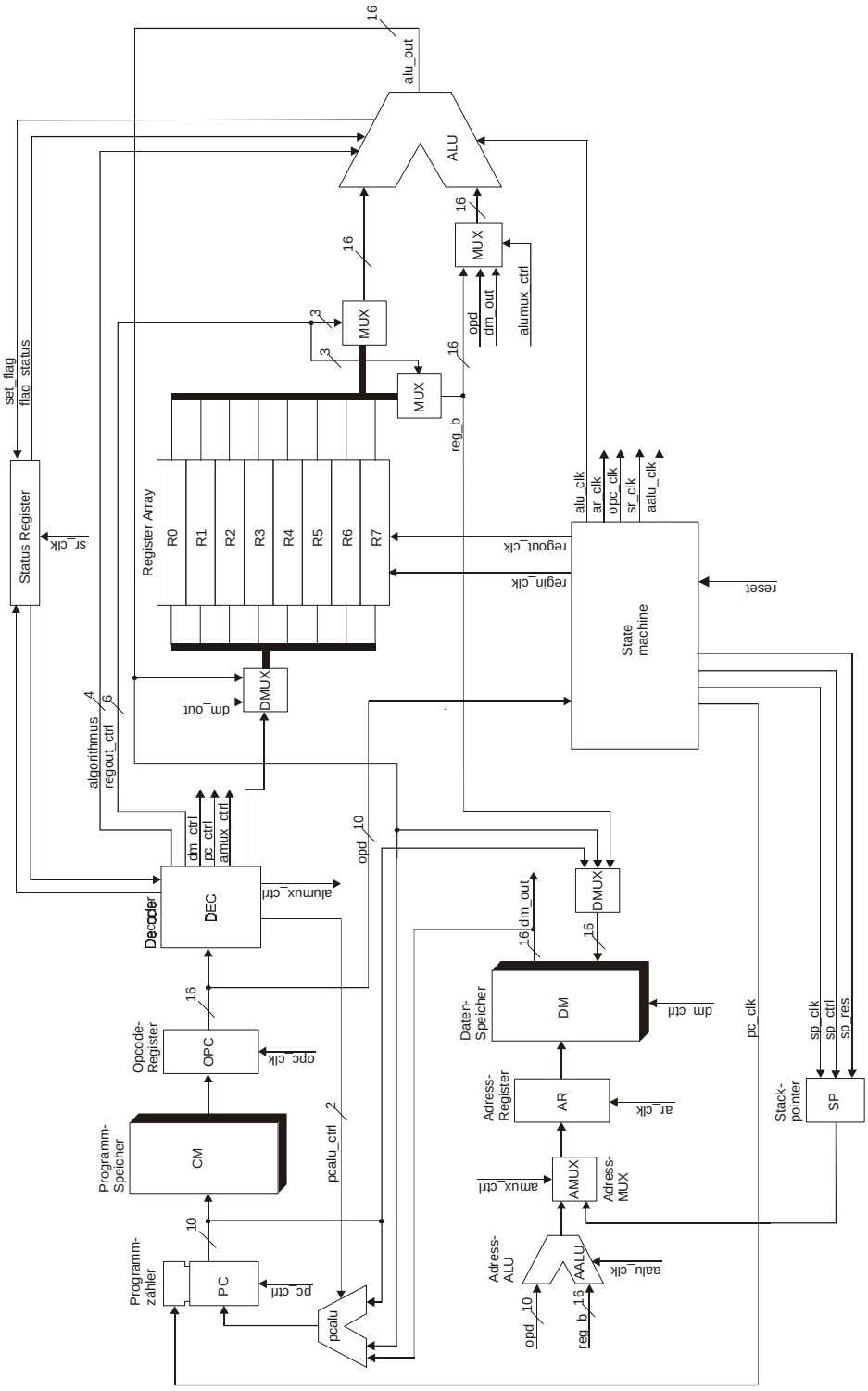


Abbildung 12: Die RISC_Core I Architektur

8.4 Peripherie

Um mit der RISC_Core auch Signalverarbeitungsaufgaben lösen zu können, muss eine Verbindung zur Aussenwelt hergestellt werden.

Als Ankopplung an die Xilinx PCI Karte steht das AD/DA-Peripherie-Modul [H4] zur Verfügung. Dieses Modul besitzt einen 8-Bit Eingangs- und einen 8-Bit Ausgangsport. An den jeweils unteren 4 Bits dieser Ports sind ein AD- und ein DA-Wandler angeschlossen, welche seriell angesprochen werden. Der analoge Eingang besitzt ein Anti-aliasing- und der analoge Ausgang ein Anit-imaging-Filter. Beide Filter sind ausgelegt als Butterworth-Typen 3. Ordnung mit einer Grenzfrequenz von 12kHz.

Da dieses Modul sehr gut für Signalverarbeitung geeignet ist, soll die RISC_Core so ausgelegt werden, dass diese Hardware angesprochen werden kann.

8.5 Assembler

Die Anforderungen aus dem Anforderungsprofil müssen noch genauer spezifiziert werden. Es soll ein 1 zu 1 Übersetzer mit Pseudobefehlen zur Programmsteuerung entstehen.

Der Assembler wird in einfachster Manier gehalten. Er übersetzt die – in einem Texteditor geschriebene – Assemblerdatei in den Maschinencode und generiert daraus eine Codedatei.

8.5.1 Assemblerdirektiven

END: Diese Direktive beendet die Assemblierphase. Nachfolgender Quellcode wird nicht mehr übersetzt.

ORG: Der ORG-Befehl dient zur Adresspegelsteuerung. Er legt die Adresse des nachfolgenden Befehls fest.

Bsp: ORG #100;

EQU: EQU dient als Symboldefinition. Der Assembler ersetzt textmässig das Symbol im Label-Feld durch den Wert des Ausdrucks im Operandenfeld.

Bsp: EQU P1, 80;

Das Symbol P1 wird im ganzen Programmcode durch den Wert 80 ersetzt. Symboldefinitionen müssen vor deren erster Benützung gemacht werden.

8.5.2 Syntax

Aufgrund der Tatsache, dass der Assembler ein Zeileninterpreter ist, müssen nur Konventionen über eine einzelne Zeile erstellt werden. Diese Konventionen müssen beim Programmieren für jede einzelne Zeile eingehalten werden.

Eine Assemblerzeile wird in folgende Zeichenfelder aufgeteilt:

Label	Opcode	Operanden	Kommentar

- Das Label-Feld dient als symbolische Adressierung einer Programmzeile. Es wird vom Assembler durch einen Doppelpunkt erkannt und in eine fixe Programmadresse umgewandelt. Das Label darf nicht mehr als 8 Zeichen umfassen.
- Das Opcode-Feld nimmt den mnemotechnischen Operationscode auf. Der Assembler sucht aus einer Tabelle den zugehörigen Maschinencode.
- Im Operanden-Feld stehen die Operanden in absoluter oder symbolischer Form. Sie müssen durch mindestens einen Abstand oder Tabulator vom Opcode getrennt sein. Sie werden stets in hexadezimaler Form angegeben.
- Im Kommentar-Feld kann Text zur Erläuterung der Programmzeile angebracht werden. Er wird vom Assembler durch das Voranstehen eines Strichpunktes erkannt und für die Übersetzung nicht berücksichtigt.

Pro Programmzeile darf nur ein Befehl in dieser Art stehen:

```
LABEL: OPCODE      OP1[,OP2[,OP3]]      ;KOMMENTAR
```

9 Realisierungskonzept

Die Realisierung des RISC Prozessors wird in zwei Phasen unterteilt. In der ersten Phase wird ein Minimalst-Modell (RISC_Core S) konzipiert. Dieses Modell basiert auf dem RISC_Core I. Es besitzt jedoch nur die für eine sequenzielle Abarbeitung wirklich notwendigen Blöcke. Der Stack, das Statusregister und auch die indizierte Adressierung werden nicht eingeschlossen. Das RISC_Core S Modell kann somit keine Kontrollflussbefehle ausführen. Ausgenommen davon ist der absolute Sprungbefehl (jmp). Dieser wird benötigt, um den Programmablauf in einer Endlosschleife zu fangen.

9.1 RISC_Core S

Das RISC_Core S Konzept wird so aufgestellt, dass es später direkt zum RISC_Core I Modell erweitert werden kann.

9.1.1 Phasenmodell

Um das CPI (clock's per instruction) zu bestimmen, muss von jedem Befehl der Datenfluss untersucht werden. Pro Register, das vom Datenfluss betroffen ist, wird ein Takt benötigt. Das CPI wird schlussendlich durch den Befehl vorgegeben, welcher am meisten Takte benötigt. Die unten aufgeführte Liste zeigt diese Untersuchung, welche anhand der RISC_Core I Architektur (Abbildung 12) durchgeführt wurde. Zusätzlich soll diese Untersuchung anhand eines Beispiels verdeutlicht werden.

◆

Beispiel 2: Bestimmen des CPI's einer Addition

Der Maschinencode muss aus dem Datenspeicher ins Opcode-Register geladen werden (Takt #1). Steht der Maschinencode im Opcode-Register, wird dieser dekodiert und bestimmt somit den Algorithmus für die ALU sowie die Operanden und das Zielregister. Der nächste Takt (Takt #2) gilt dem Register-Array, welches der ALU die Operanden zur Verfügung stellt. Mit dem dritten Takt (Takt #3) wird die Addition in der ALU durchgeführt. Das Resultat steht nun im Ausgangsregister der ALU. Der vierte Takt (Takt #4), welcher die Write-back-Phase darstellt, schreibt das Resultat zurück in das Zielregister.

◆

Die Flagbefehle sind in dieser Liste bei den arithmetischen / logischen Befehlen eingeschlossen.

- Transferbefehle:
  - Opcode laden.
  - Operand laden, entweder aus dem Datenspeicher oder aus einem Register. / Programmcounter inkrementieren. / Stackpointer dekrementieren (nur bei POP–Befehl)
  - Daten weiterleiten (der Datenpfad bei Transferbefehlen führt immer durch die ALU)
  - Operand zurückschreiben, entweder in ein Register, oder in den Datenspeicher. / Stackpointer inkrementieren (nur bei PUSH–Befehl).

- Arithmetische / logische Befehle:
  - Opcode laden.
  - Operanden laden (aus Register). / Programmcounter inkrementieren.
  - Operation ausführen.
  - Statusregister beeinflussen. / Resultat speichern (ausser bei compare-Befehlen).

- Kontrollflussbefehle:
  - Opcode laden.
  - Programmcode Adresse speichern (nur beim CALL-Befehl). / Stackpointer dekrementieren (nur bei RET- und RETI-Befehl).
  - Daten durch die ALU weiterleiten. / Stackpointer inkrementieren (nur bei CALL-Befehl).
  - Programmcounter laden (bei absoluten Sprüngen, und bei konditionellen Sprüngen mit erfüllter Bedingung). / Programmcounter inkrementieren (beim NOP-Befehl und bei konditionellen Sprüngen mit nicht erfüllter Bedingung).

Wie aus dieser Liste ersichtlich ist, können alle Befehle innerhalb von 4 Taktzyklen abgearbeitet werden. Somit wird das CPI der RISC_Core auf 4 festgelegt.

Zusätzlich müssen noch die Phasen für den Resetfall bestimmt werden. Der Reset initiiert einen Neustart des Programmes. Dazu ist es notwendig den Programmcounter sowie den Stackpointer zurückzusetzen. Dies geschieht in drei Phasen:

- Setzen der Reseteingänge von Programm- und Stackpointer.
- Der globale Reset wird zur Bestimmung der übernächsten Phase kontrolliert. Ist der globale Reset gesetzt, beginnt der Resetzyklus anschliessend von Neuem.
- Löschen der Reseteingänge von Programm- und Stackpointer.

Das Phasenmodell ist im Kapitel 10.2 anhand der state machine anschaulich dargestellt.

9.1.2 Speicher und Memory Map

Auf der Xilinx PCI Karte der FHS lässt sich mit externem Speicher nur eine von Neumann Struktur implementieren. Deshalb werden beide Speicherbereiche im FPGA Virtex 300 implementiert. Der Virtex 300 Baustein bietet 65536-Bit BlockRAM. Davon werden beim RISC_Core S Modell beiden Speicherteilen je zwei 4096-Bit dualported RAM mit 16-Bit Orientierung zur Verfügung gestellt.

$\Rightarrow$ Programm-/Datenspeicher: $\frac{2 \cdot 4096Bit}{8Bit} = 1kByte \equiv \frac{2 \cdot 4096Bit}{16Bit} = 512Word$

Wird mehr Speicher benötigt, kann ein Teil auf den Virtex 50 Baustein bzw. auf die externen RAM's verschoben und über den 34-Bit breiten FPGA Link angesprochen werden. Tabelle 5 zeigt eine Übersicht über die möglichen Speichereinteilungen.

	BlockRAM	Daten-/Programmspeicher
RISC_Core S	16384-Bit	je 1kByte
Virtex 300 maximal	65536-Bit	8kByte gesamt
Virtex 400[3] maximal	81920-Bit	10kByte gesamt
Virtex 50 maximal	32768-Bit	4kByte gesamt
Externes RAM		4x128kByte
Maximale Speichergrösse mit Xilinx PCI Karte (Virtex 400)		Programmspeicher: 10kB Datenspeicher : 516kB

Tabelle 5: Speicherübersicht

Das Auslegen der Speicher als Dualport RAM bringt zwei Vorteile mit sich:

- Zum Herunterladen des Programmcodes wird keine zusätzliche Logik benötigt, welche den Daten- und Adressbus prozessorseitig in einen Tristatezustand setzt.
- Datenseitig eröffnen sich Debuggmöglichkeiten über den PCI Bus. Der Datenspeicher kann während der Programmlaufzeit gelesen oder sogar verändert werden.

[3] Xilinx PCI Karte mit einem Virtex 400 anstelle des Virtex 300 bestückt.

Aufgrund der verschiedenen Busbreitenorientierung findet zwischen dem Local[4]- und dem RISC-Bus eine Adressumrechnung stat.

Auf dem Local-Bus bilden 4 Bytes ein 32-Bit langes Datenwort. Aus den unteren zwei Bytes wird das 16-Bit breite Datenwort auf dem RISC-Bus gebildet. Das heisst, dass die oberen 2 Bytes des Local-Busses nicht verwendet werden können! Anhand eines Beispiels soll dieser Sachverhalt verdeutlicht werden.

◆

Beispiel 3: Bildung eines 16-Bit Datenwortes auf dem RISC-Bus:

Die Daten an den Adressen 1000h und 1001h des Local-Busses bilden das 16-Bit breite Datenwort an der Adresse 000h des RISC-Busses. Die Werte ‚34h' und ‚1Fh' werden nicht verwendet! Weiter bilden die Daten an den Adressen 1004h und 1005h des Local-Busses das nächste Datenwort an der Adresse 001h des RISC-Busses.

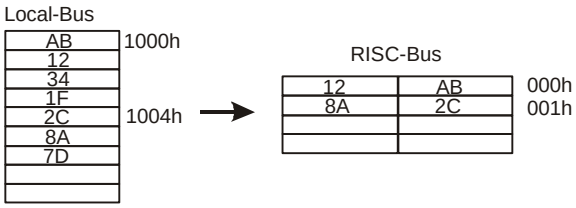


Abbildung 13: 16-Bit Datenwort auf dem RISC-Bus

◆

Das Mappen der Daten vom PCI- über den Local- bis hin zum RISC-Bus wird vom PCI-Controller übernommen. Dessen Konfiguration ist im Anhang zu finden.

Abbildung 14 zeigt das memory map von der PCI Schnittstelle bis hin zum RISC-Bus. Darauf ist die Adressumrechnung zwischen dem Local-Bus und dem RISC-Bus gut zu erkennen!

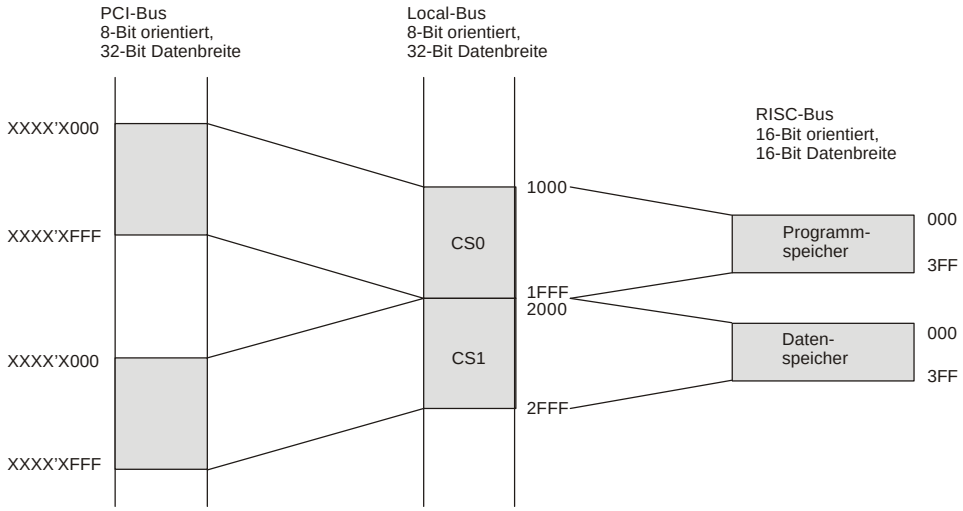


Abbildung 14: Memory map der PCI Schnittstelle

[4] Als Local-Bus wird der Adressbus zwischen dem PCI-Controller und dem FPGA verstanden.

9.1.3 Reset

Um die Kontrolle über den Programmstart zu erlangen, wird ein Reset-Register eingeführt. Dieses Reset-Register liegt an der obersten Datenspeicheradresse, also 200h [→ Anhang: Schema RISC Core]. Wird in dieses Register der Wert 1h geschrieben, liegt an der CPU der high-aktive Resetstatus an.

9.1.4 Zeitverhalten und Taktfrequenz

Die Taktfrequenz gilt oft als Inbegriff der Leistung eines Prozessors. Dies ist insofern richtig, als dass mit doppelter Taktfrequenz natürlich auch die Verarbeitungsgeschwindigkeit verdoppelt wird. Bei Prozessoren verschiedener Bauart darf jedoch kein direkter Vergleich über diese Eigenschaft erfolgen.

FPGA's werden bei der Produktion nach ihrer Gatterlaufzeit selektiert. Xilinx Virtex Bausteine sind erhältlich mit der Option 4, 5 oder 6ns. Die auf der Xilinx PCI Karte eingesetzten Virtex gehören der schnellsten Gruppe mit 4ns Verzögerung an. Um die maximale Durchlaufzeit und somit die maximale Taktfrequenz des Prozessors zu bestimmen, müssen alle Datenpfade untersucht werden. Besondere Vorsicht ist bei der Verteilung des Taktsignals geboten.

Werden interne Signale mit unterschiedlicher Verzögerung behaftet, können dieselben Effekte auftreten wie beim Phasenpipelining. Datenhazards treten auf, weil während der sensitiven Flanke des Taktsignals die zu verarbeitenden Daten noch „unterwegs" sind.

Die einfachste Methode dieses Problem zu untersuchen, bietet die Post Route Simulation[5], welche das Synthesetool [S2] zur Verfügung stellt. Diese Methode kann jedoch erst während der Realisierung angewandt werden! Im Kapitel 10.3 wird das Zeitverhalten des Prozessorcores untersucht.

[5] Post Route Simulation wird im Glossar erwähnt.

9.2 Erweiterung zum RISC_Core I Modell

Das RISC_Core S Modell wird nun mit einem Stack, einem Statusregister und einem Peripherieblock zum RISC_Core I Modell erweitert. Damit das Phasenmodell für die konditionellen Sprünge beibehalten werden kann, wird eine kleine arithmetische Einheit (PCALU) eingebaut, welche die neue Programmcode Adresse berechnet.

9.3 Berechnung der Programmcode Adresse bei konditionellen Sprüngen

Bei konditionellen Sprungbefehlen hängt die folgende Programmcode Adresse von den Statusflag ab. Stimmt die Kondition mit dem Status überein, wird gesprungen. Herrscht keine Übereinstimmung, wird der Programmcounter um eins erhöht, und der nächste Befehl wird abgearbeitet.

Bei Schleifenanweisungen ist es oft der Fall, dass die Bedingung am Ende der zu wiederholenden Sequenz steht. Dies bedeutet, dass bei Übereinstimmung ein Rückwärtssprung erfolgt. Die Sprungweite muss, um einen Rückwärtssprung zu ermöglichen, vorzeichenbehaftet sein. Rückwärtssprünge müssen im Zweierkomplement angegeben werden.

◆

Beispiel 4: Springe bei Gleichheit von der Programmcode Adresse A7h nach 93h zurück.

Sprungweite:	A7h – 93h = 14h
Zweierkomplement:	14h $\xrightarrow{ZK}$ ECh
Sprungbefehl:	jne #ec;
Neue Programmcode Adresse:	<PC> = A7h + ECh = 193h

Der Übertrag aus der vorzeichenbehafteten Addition wird ignoriert. Somit ergibt sich die gewünschte Rücksprungadresse.

◆

9.3.1 Parallelport und Stack

Die Xilinx PCI Karte bietet als Verbindung zur Aussenwelt die LVDS Schnittstelle an. Von der 30-Bit breiten Schnittstelle werden je 8 Bit für einen Eingangs- und einen Ausgangsport verwendet. Für die Ansteuerung der Ports werden spezielle Funktionsregister im unteren Bereich des Datenspeichers angelegt. Benötigt werden je ein Port Register.

Abbildung 15 zeigt das neue memory map des Datenspeichers mit den speziellen Funktionsregistern.

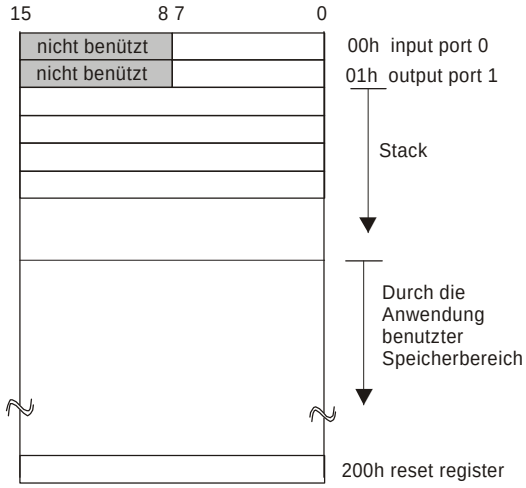


Abbildung 15: Funktionsregister im Datenspeicher

Somit sind die Adressen 00h und 01h von den Port-Registern belegt und dürfen nicht mehr vom Stack benützt werden! Der Stackpointer muss nach einem Reset mit dem Wert 02h initialisiert werden.

☞ Der Reset-Wert des Stackpointers beträgt 02h.

9.4 Assembler

Um den Assemblerquellcode in den Maschinencode übersetzen zu können, wird eine Codetabelle benötigt. Die Codetabelle listet die mnemonischen Symbole mit der entsprechenden binären Codierung auf. Aus dieser Tabelle und den im Quellcode angegebenen Operanden setzt der Assembler das Datenwort zusammen. Dieses Datenwort wird in der Codedatei direkt unter die zugehörige Adresse geschrieben.

Abbildung 16 zeigt den Ablauf vom Quellcode bis hin zum programmierten RISC_Core.

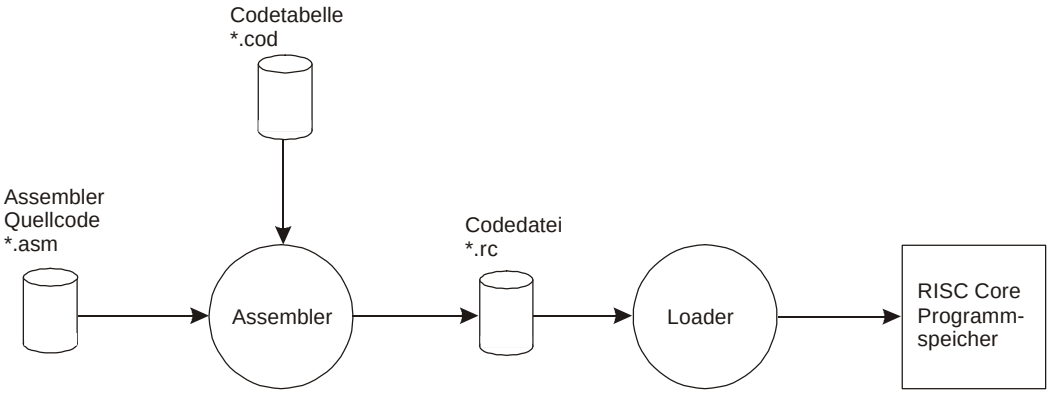


Abbildung 16: Programmierablauf

Der Loader wird benötigt, um die Codedatei in den Programmspeicher des RISC Prozessors zu schreiben. Er verarbeitet die Codedatei in der Manier eines Stapelverarbeitungsprogramms. Weiter bietet der Loader die bereits erwähnten Debug Möglichkeit durch das Zurücklesen des Datenspeichers.

10 Implementierung

Implementiert wird das Modell mit der Entwicklungsumgebung von Xilinx, dem Foundation ISE Software Paket. Die Speicherblöcke sowie die PCI-local-Bus Ankopplung werden schematisch, die Zentraleinheit (CPU) hingegen in VHDL erstellt [→ Anhang: Schema RISC_Core I].

Um die Zentraleinheit zu programmieren, wird die Prozessorarchitektur auf die Register Transfer Ebene gebracht und ein Flussdiagramm der state machine erstellt. Die Register Transfer Ebene zeigt die einzelnen Blöcke und die verbindenden Datenflüsse auf.

10.1 Register Transfer Level (RTL)

Die Register Transfer Ebene ist unter VHDL eine gängige Methode, um Blockschaltbilder zu implementieren und die einzelnen Blöcke als Code auszuarbeiten. Theoretisch besitzen die Blöcke der Register Transfer Ebene an ihren Ein- und Ausgängen ein Register. Diese Konvention einzuhalten ist jedoch nur möglich, wenn ein komplett synchrones System aufgebaut wird. In diesem Fall ist das nicht möglich, da der Decoder eine asynchrone Maschine ist.

Sowohl die Blöcke wie auch die Datenflüsse sind im Quellcode auf der obersten Ebene wiederzufinden.

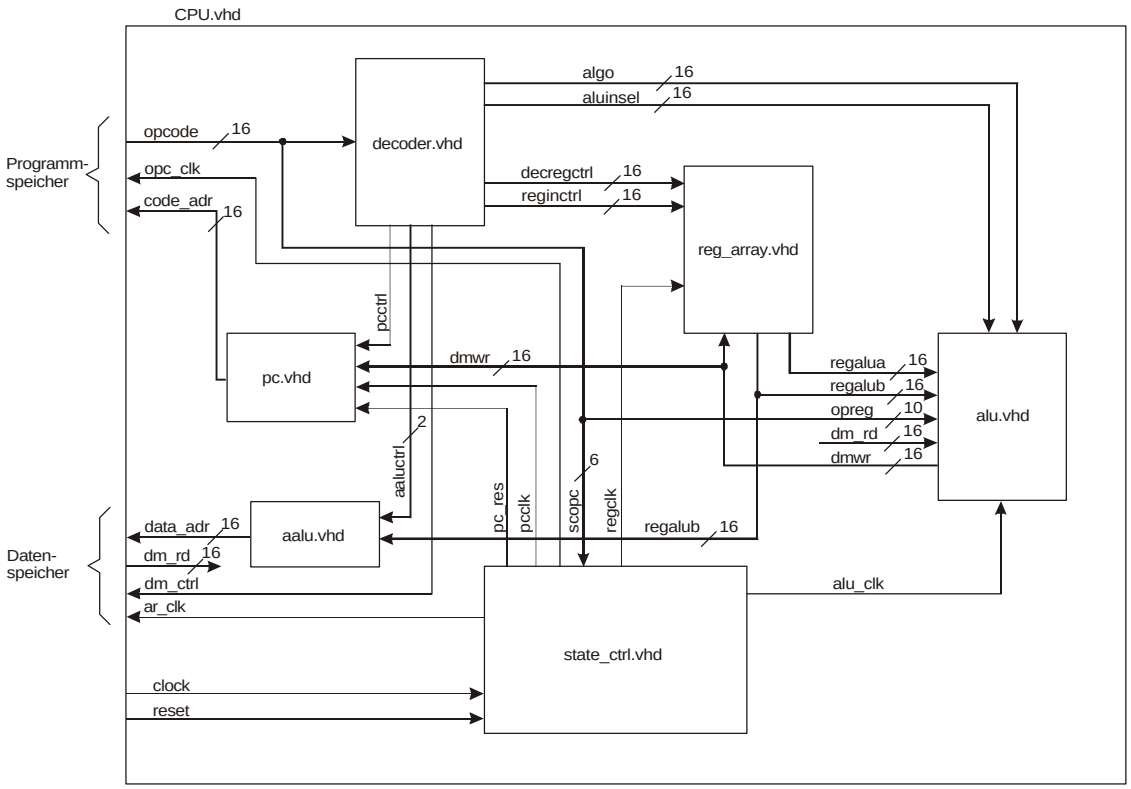


Abbildung 17: Register Transfer Level des RISC_Core S

10.2 State machine

Die state machine ist das Herz des Prozessors. Sie verteilt den Takt an die einzelnen Blöcke und ist somit verantwortlich für den zeitlichen Ablauf der Instruktionen. Aus diesem Grund wurde das Flussdiagramm in Abbildung 18 erstellt.

Die state machine representiert das Phasenmodell des Prozessors und der Begriff ‚state' korrespondiert mit dem Begriff der Phase.

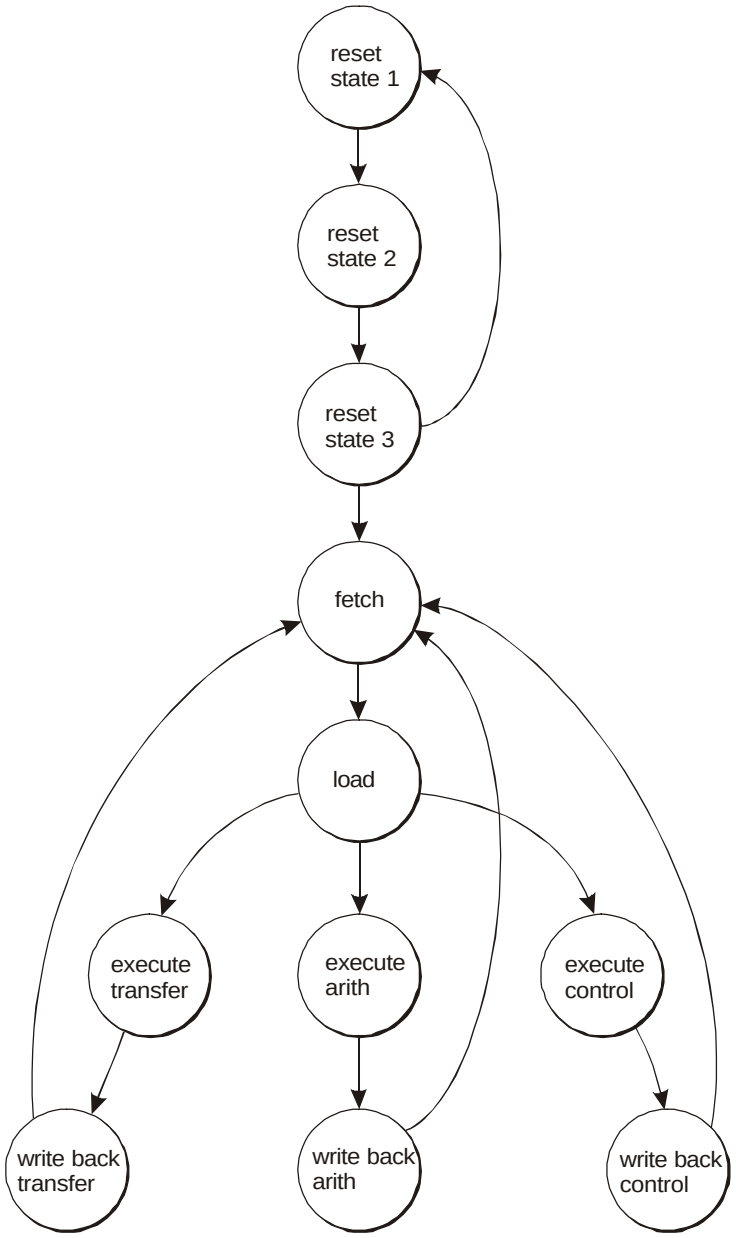


Abbildung 18: state machine

10.3 Untersuchen des Zeitverhaltens mit Hilfe der Post Route Simulation

Abbildung 19 zeigt das Zeitverhalten des RISC_Core S Modells anhand der Post Route Daten, welche mit dem Xilinx Synthese Tool (XST) erzeugt wurden. Die Daten entsprechen jeweils dem kritischen Pfad, ohne Berücksichtigung der Signallaufzeiten auf den Verbindungspfaden.

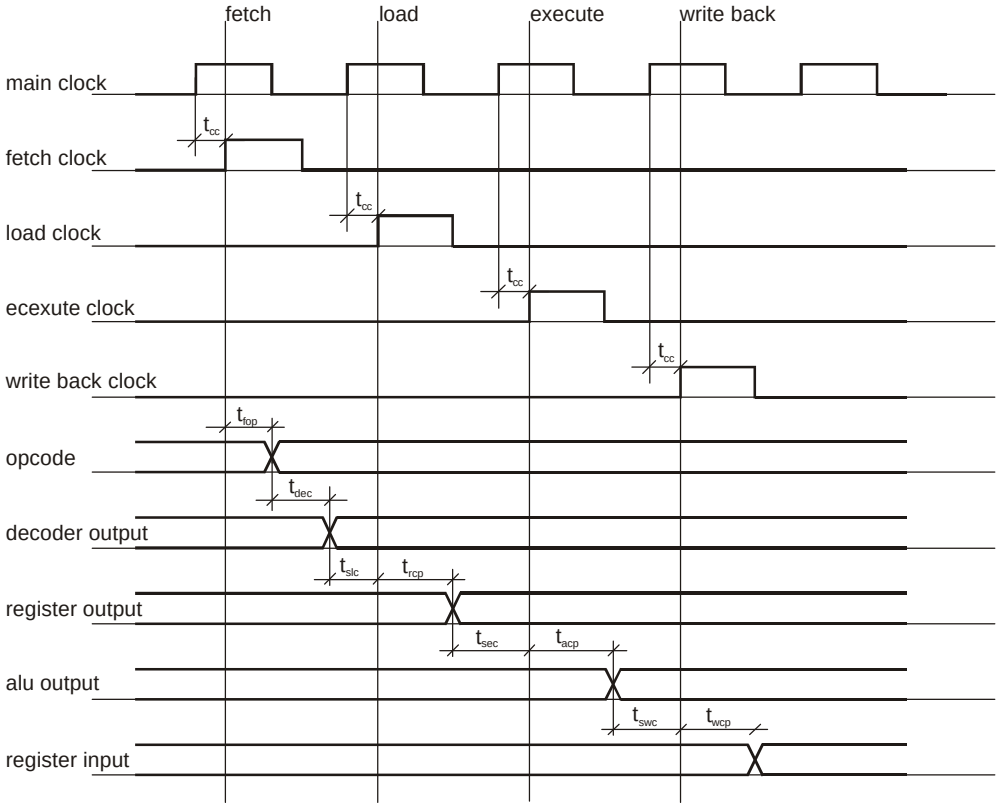


Abbildung 19: Zeitverhalten des RISC_Core S Modells anhand der Post Route Daten

t_{cc}	main_clock to phase_clock delay	10ns
t_{fop}	opcode to pad	10ns
t_{dec}	decoder delay	15ns
t_{slc}	setup load to clock	8ns
t_{rcp}	register array clock to pad	11ns
t_{sec}	setup execute to clock	13ns
t_{acp}	alu clock to pad	10ns
t_{swc}	setup write back to clk	9ns
t_{wcp}	write back clock to pad	12ns

Tabelle 6: Legende zu Abbildung 19

Der kritische Punkt liegt zwischen ‚fetch' und ‚load'. Wird noch eine Signallaufzeit von 10ns zwischen den Blöcken einbezogen, beträgt die gesamte Verzögerung 43ns.

$$f_{\max} = \frac{1}{t_{delay\_synthese} + t_{delay\_constraint}} = \frac{1}{33ns + 10ns} = 23MHz$$

Für den 50MHz Takt der Xilinx PCI Karte wird ein Teiler von 4 gewählt. Dies entspricht einem CPU-Takt von 12.5MHz.

☞ Die RISC_Core I wird mit 12.5 MHz getaktet.

Das Einfügen eines zusätzlichen Taktes (stall) zwischen den Phasen ‚fetch' und ‚load' erscheint im ersten Moment sinnlos. Wird aber das Zeitverhalten danach nochmals untersucht, zeigt sich, dass sich der kritische Pfad verschoben hat. Dieser liegt jetzt mit 24ns zwischen den Phasen ‚load' und ‚execute'. Wiederum wird eine zusätzliche Signallaufzeit von 10ns berücksichtigt.

$$f_{\max} = \frac{1}{t_{delay\_synthese} + t_{delay\_constraint}} = \frac{1}{24ns + 10ns} = 29MHz$$

Die Taktfrequenz kann nun auf 29MHz erhöht werden, dafür umfasst ein Maschinenzyklus 5 Takte.

$$Maschinenzyklus(4\ Takte,\ 23MHz) = 4 \cdot 43ns = 172ns$$
$$Maschinenzyklus(5\ Takte,\ 29MHz) = 5 \cdot 34ns = 170ns$$

Die Verarbeitungsgeschwindigkeit lässt sich mit dieser Methode erhöhen, jedoch nur um 1%.

Daraus folgt, dass ein Weg gesucht werden muss, um das Zeitverhalten zu verbessern. Denn, wird das Zeitverhalten besser, kann auch die Taktfrequenz erhöht werden.

10.3.1 Methoden zur Verbesserung des Zeitverhalten

Um das Zeitverhalten zu verbessern gibt es noch zwei weitere Methoden. Die eine ist das manuelle Routen des Chips, die andere das Ändern der Hardwarestruktur mittels Beschreibungssprache.

- Manuelles Routen: Die Xilinx Entwicklungsumgebung Foundation ISE [S3] bietet mit dem Floorplaner die Möglichkeit, das Chipdesign selber zu bestimmen. Somit können alle Gatter manuell plaziert und die Verbindungsleitungen von Hand gezogen werden. Das Chipdesign ist so zu optimieren, dass alle relevanten Datenpfade die selbe Verzögerungszeit aufweisen. Ist dies der Fall, spielt die absolute Verzögerungszeit keine Rolle mehr, da alle Daten zum selben Zeitpunkt eintreffen. Besondere Vorsicht ist bei der Verteilung des Taktsignales geboten.
- Ändern der Hardwarestruktur: Bei der Entwicklung eines Codesignes wird der VHDL Code in eine Hardwarestruktur übersetzt. Durch geschicktes Programmieren kann das Zeitverhalten auf dem Chip verbessert werden. Dazu muss jedoch der Zusammenhang zwischen der Software- und der Hardwarestruktur bekannt sein. Das heisst, es muss bekannt sein, welche Hardwarestruktur der Compiler anstrebt. Dabei stellt auch die Entwicklungsumgebung Optionen zur Verfügung, welche genau betrachtet werden müssen.

◆

Beispiel 5: Ändern der Hardwarestruktur eines Addierers mit Xilin Foundation

Eine Addition in VHDL übersetzt das Foundation [S3] standardmässig in einen ‚ripple-carry-adder'. Wird die Option „Synthesis Optimization Effort" auf high gestellt, erzeugt der Compiler einen ‚carry-lock-ahead-adder'. Der ‚carry-lock-ahead-adder' ist natürlich deutlich schneller als der ‚ripple-carry-adder', benötig aber etwa die doppelte Chipfläche.

◆

10.4 Assembler und Loader

In einem ersten Durchgang sucht sich der Assembler alle Labels aus dem Quellcodefile. Dies ist nötig, damit beim Assemblieren auch die Labels bekannt sind, welche erst nach ihrer Benützung deklariert werden (z.B. Unterprogrammaufruf).

Im zweiten Durchlauf wird bei jeder Zeile anhand des Strichpunktes entschieden, ob diese zu interpretieren ist. Ist dies der Fall, wird der Befehl beziehungsweise die Direktive bestimmt. Ist eine Direktive erkannt worden, wird diese entsprechend ausgewertet. Ist ein Befehl erkannt worden, wird in der Codetabelle der entsprechende Maschinencode ausgelesen. Der Maschinencode und die zugehörige Programmcode-Adresse werden, nach dem in Abbildung 20 gezeigten Muster, in die Codedatei geschrieben.

```
        EQU   P1, #1;

        MOVI  R1, #0;
        MOVI  R0, P1;
        MOV   &R0, R1;

LOOP:   NOP;
        JMP   LOOP;
```

```
0
1100
1
1000
2
808
3
e000
4
fc03
```

Abbildung 20: Beispiel eines Assembler Quellcodes und der assemblierten Codedatei.

Der Loader ist zuständig für das Beschreiben des Programmspeichers. Er basiert auf den mit WinDriver ([S3], [D4]) generierten Funktionen. In einem ersten Schritt wird die Xilinx PCI Karte auf dem PCI-Bus lokalisiert und geöffnet. Ist dieser Schritt erfolgreich abgeschlossen, beginnt das interpretieren der Codedatei. Der Loader verarbeitet die Codedatei in der Manier eines Stapelverarbeitungsprogramms. Dabei werden immer zwei Zeilen gelesen, bevor der Maschinencode an die entsprechende Adresse geschrieben wird.

Weiter bietet der Loader die Funktion des Debugging. Das Debugging beschränkt sich auf das Lesen des Datenspeichers und das Ein- und Ausschalten des Resets.

11 Systemtest

Getestet wurde das System mittels Debbug-Möglichkeit und mit dem AD/DA-Peripherie-Modul. Der Systemtest wurde in zwei Abschnitte eingeteilt:

- Test der RISC_Core I anhand von Testprogrammen
- Implementierung einer Signalverarbeitungsaufgabe

11.1 Test der RISC_Core I anhand von Testprogrammen

Die Testprogramme wurden so ausgelegt, dass alle Hardwarekomponenten der RISC_Core I mit ihrer Beeinflussung überprüft werden konnten. Besonderers Augenmerk wurde auf das Statusregister gerichtet, da die konditionellen Sprünge bei der Assemblerprogrammierung eine zentrale Rolle spielen.

◆

Beispiel 6: Testprogramm, welches die Beeinflussung des ‚sub' Befehls auf das Statusregister untersucht.

```
        MOVI  R0, #10;
        MOVI  R1, #ab;
        MOVI  R2, #12;
        SUB   R3, R2, R1;
        JNS   #2;
        MOVI  R3, #5;
        SUB   R4, R1, R2;
        JNC   #2;
        MOVI  R3, #5;
        MOV   &R0, R3;
LOOP:   NOP;
        JMP   LOOP;
```

Steht nach der Abarbeitung des Testprogramms der Wert ff67h an der Adresse 10h, kann daraus geschlossen werden, dass die Beeinflussung des Statusregister durch den ‚sub' Befehl korrekt ist.

◆

Resultat: Es wurden alle implementierten Befehle erfolgreich ausgetestet!

11.2 Implementierung einer Signalverarbeitungsaufgabe

Die Weiterführung des Tests basiert auf der Implementierung einer Signalverarbeitungsaufgabe auf dem System.

Als Aufgabe wurde ein Allpass gewählt, welcher über den A/D-Wandler des AD/DA-Peripherie-Modules einen analogen Wert einliest, und denselben über den D/A-Wandler des AD/DA-Peripherie-Modules wieder ausgibt.

Der Schwerpunkt dieser Aufgabe liegt bei der Generierung der Signale für die serielle SPI Schnittstelle [→ H4]. Dazu werden viele Bitmanipulationen und Sprungbefehle benötigt.

Für die Bitmanipulationen werden vorwiegend logische Operationen (and, or) mit der unmittelbaren und der Register-direkten Adressierung verwendet. Die Sprungbefehle sind abhängig vom Statusregister und der Korrektheit der vorzeichenbehafteten Arithmetik (→ Kapitel 9.3 ‚Berechnung der Programmcode Adresse bei konditionellen Sprüngen).

Durch die vielseitige Benützung der Befehle resultiert aus diesem Test ein recht eindeutiges Ergebnis über die Funktionsfähigkeit der entwickelten RISC-Core.

⇒ Die implementierte Signalverarbeitungsaufgabe wird korrekt ausgeführt. Daraus ist zu schliessen, dass die eingesetzten Systemkomponenten, sprich Assembler, Loader und RISC_Core I, fehlerfrei sind. Es ist jedoch zu bemerken, dass diese Fehlerfreiheit nicht garantiert werden kann!

Das Testen eines digitalen Systems anhand eines Beispiels ist in der Praxis keine gängige Methode, denn es sollte die Funktionsfähigkeit annähernd vollständig garantiert werden können. Zur Verifikation von digitalen Systemen bestehen zwei Varianten: die eine ist das Testen mittels Testvektoren und die andere das Aufstellen der boolschen Funktionsgleichung.

Bei den Testvektoren handelt es sich um Datenwörter, welche systemintern in spezielle Register geschrieben werden, um somit einen Zustand zu simulieren. Durch einen Vergleich dieses Zustandes mit der Reaktion des Systems kann die Korrektheit verifiziert werden.

Ein neuer Ansatz ist das Aufstellen der boolschen Funktionsgleichung, welche das ganze digitale System beschreibt. Diese Methode bietet eine höhere Sicherheit und den Vorteil, dass das System von Aussen getestet werden kann.

12 Erweiterungen der Prozessor-Core für Signalverarbeitungssysteme

Dieses Kapitel zeigt mögliche Erweiterungen für die RISC Prozessor-Core, welche die Implementation von Signalverarbeitungssystemen vereinfachen und die Effizienz steigern würden.

12.1 Phasenpipelining

Das Phasenpipelining ist die effektivste Methode um die Leistung eines Prozessors zu steigern. Sie ist jedoch sehr aufwendig und benötigt eine gute Koordination der Phasenabläufe um Hazards zu verhindern.

Die einzelnen Phasen müssen klar voneinander getrennt werden können. Trotzdem ist es nicht zu vermeiden, dass gleichzeitig auf die selben Ressourcen zugegriffen werden soll. Prädestiniert, um Kollisionen hervorzurufen, sind die Load- und die Write-back-Phase, welche beide auf den Datenspeicher zugreifen (bei Speicheroperationen). Um diesem Problem aus dem Weg zu gehen, kann ein Stall (Wartezyklus) eingefügt oder ein Feedforward eingeleitet werden.

Stall: Ein Stall verzögert die nächste anliegende Phase. Stalls sind sehr wichtig bei Sprungbefehlen. Natürlich können sie auch eingesetzt werden um Struktur- und Datenflusshazards zu verhindern.

Feedforward: Unter Feedforward versteht man einen Datenvorschub. Um Datenflusshazards zu vermeiden, müssen Informationen zum Teil frühzeitig bekannt sein. Aus diesem Grund werden Feedforward-Pfade eingeführt, welche das Phasenmodell umgehen und die Daten direkt der nächsten Operation zur Verfügung stellen.

Im Beispiel 7 werden beide Methoden gezeigt. Abbildung 22 zeigt ein Phasenpipelining mit einem Feedforward-Pfad. Das Resultat der Addition (zukünftiger Inhalt von R1) wird über den Feedforward-Pfad bereits der Subraktion zur Verfügung gestellt. Dieselbe Befehlsfolge ist in Abbildung 21 mit Stalls gelöst. Es müssen 2 Verzögerungstakte eingefügt werden, um auf den korrekten Registerinhalt zugreifen zu können.

◆

Beispiel 7: Verhindern eines Datenflusshazards

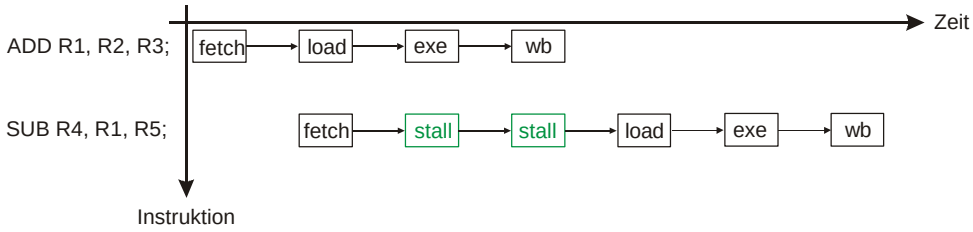


Abbildung 21: Pipelining mit Hilfe von Stalls

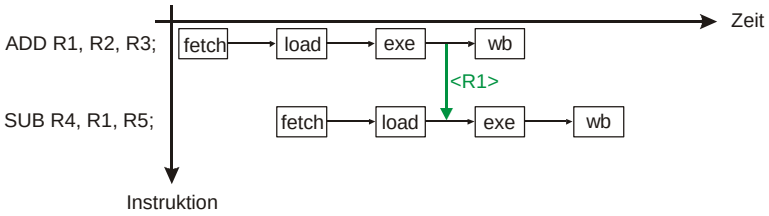


Abbildung 22: Pipelining mit einem Feedforward-Pfad

◆

Das Beispiel zeigt die Effizienz eines Feedforward-Pfades. Es entsteht kein Unterbruch im Pipelining!

12.2 Multiply / Accumulate Unit (MAC)

Die MAC ist eine Multiplizier- und Additionseinheit, wie sie in jedem DSP eingesetzt wird. Das gemeinsame Multiplizieren und Addieren in einer Einheit wird bei vielen Signalverarbeitungsalgorithmen wie zum Beispiel dem FIR-Filter benötigt.

Nutzen der MAC gezeigt anhand des FIR-Algorithmus:

$$FIR-Algorithmus: \quad y[n] = \sum_{i=0}^{k} a_i \cdot x[n-i] = a_0 \cdot x[n] + a_1 \cdot x[n-1] + ... + a_k \cdot x[n-k]$$

Mit der MAC kann nun gleichzeitig zu der Multiplikation $a_i \cdot x[n-i]$ der bereits bekannte Wert $a_{i-1} \cdot x[n-i-1]$ addiert werden. Dank dieser Kombination kann ein FIR-Filter 50. Ordnung mit 50 Operationen berechnet werden.

Eine MAC kann folgende Operationen ausführen:

- (signed) Operator * (signed) Operator
- (signed) Operator * (unsigned) Operator
- (unsigned) Operator * (unsigned) Operator
- (signed) Operator * (signed) Operator + Register
- (signed) Operator * (unsigned) Operator + Register
- (unsigned) Operator * (unsigned) Operator + Register

Das Resultat einer Multiplikation ist immer so breit wie die Breite beider Operanden zusammen. Bei einer binären 16-Bit Multiplikation wird das Resultat also 32-Bit breit.

Eine MAC macht natürlich nur dann Sinn, wenn die Multiplikation in einem Taktzyklus ausgeführt werden kann. Dazu ist es nötig, den Multiplizierer parallel aufzubauen. Der Preis, der dafür bezahlt wird, ist eine grosse Chipfläche.

Abbildung 23 zeigt den aufwendigen Aufbau eines parallelen 4-Bit Multiplizierers. Y0..Y3 und X0..X3 kennzeichnen die 4-Bit breiten Eingänge. Der 8-Bit breite Ausgang ist mit m0..m7 beschriftet. Die mit einem + gekennzeichneten Blöcke stellen Volladdierer dar.

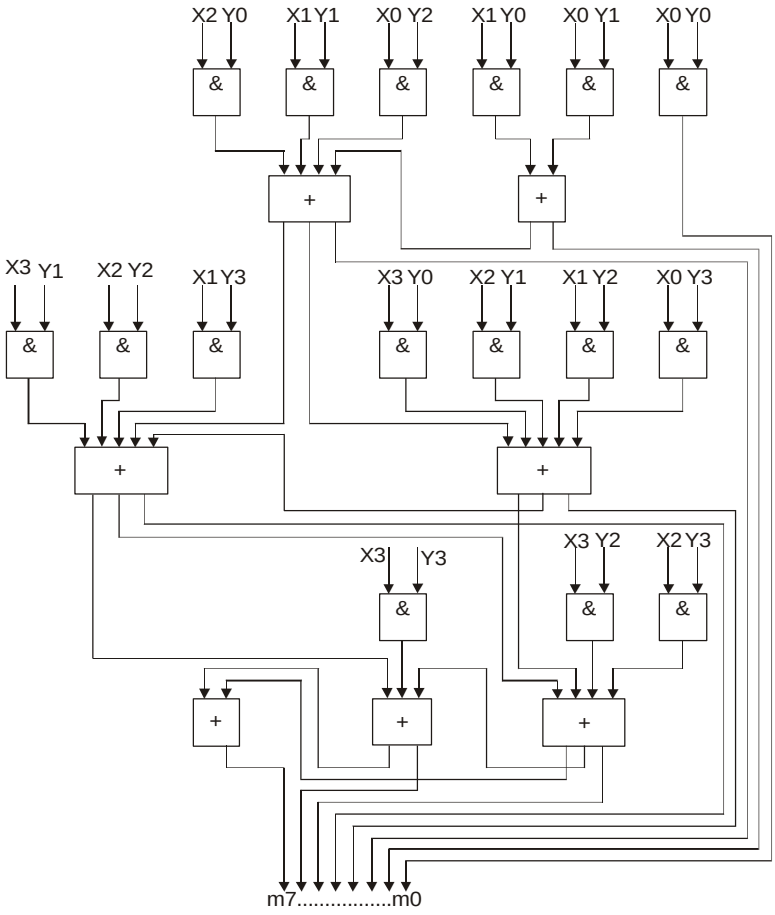


Abbildung 23: 4-Bit parallel Multiplizierer.

12.2.1 Multiplizieren mit der RISC_Core I

Da die RISC_Core I nicht mit vorzeichenbehafteten Werten arbeitet, kann in diesem Fall eine einfache Lösung gefunden werden. Die ALU kann mit einem zusätzlichen Multiplizierer ausgestattet werden. Dies würde nicht den selben Konfort bieten wie eine MAC, aber der Vorteil, eine Multiplikation nicht iterativ lösen zu müssen, wäre gegeben.

Es gilt: Die ALU der RISC_Core I ist 16-Bit breit. Also lässt sich maximal eine 8*8 Bit Multiplikation lösen.

Die Multiplikation zählt zu den Arithmetischen / logischen Befehlen und gehorcht den selben sequentiellen Abläufen wie zum Beispiel eine Addition. Eine Anpassung der State machine ist somit nicht nötig. Eine Modifikation des Instruktionssatzes und somit des Dekoders, sowie die Erweiterung der ALU genügen, um einen Parallel-Multiplizierer zu implementieren.

12.3 Registershadowing

Im Kapitel 8.3.1 wurde der Sinn und Zweck des Stacks behandelt. Eine weitere Methode, die Register und die Absprungadresse bei einem Unterprogrammaufruf oder bei einer Interruptanforderung zu sichern, ist das sogenannte Registershadowing. Das Registershadowing basiert auch auf einer Parallelisierung in der Hardwarestruktur. Anstatt die Register in den Stack zu schreiben, wird bei einem Routinenaufruf (Unterprogramm oder Interrupt) auf eine zweite Registerbank umgeschaltet. Dies bedeutet, dass für jede Verschachtelungsebene eine Registerbank zur Verfügung stehen muss. Parallel dazu muss ausserdem eine zusätzliche Registerbank vorhanden sein, welche die Absprungadressen aufnehmen kann. Die Anzahl dieser Register wird durch die maximal zulässige Verschachtelungstiefe bestimmt.

Abbildung 24 zeigt die Hardwarestruktur für ein Registershadowing mit einer maximalen Verschachtelungstiefe von 7.

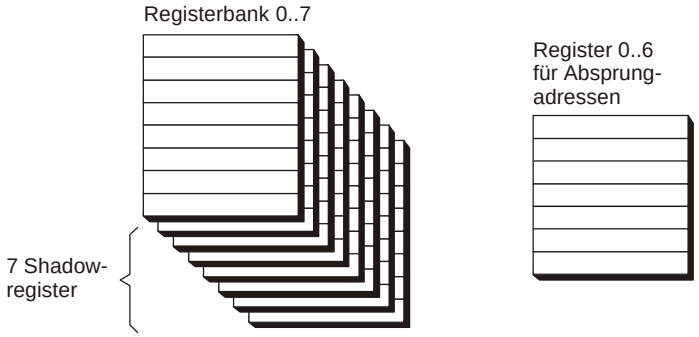


Abbildung 24: Registershadowing

12.4 Multiprozessorsystem

Dieses Kapitel behandelt die Frage, wie sich die Leistung von Signalverarbeitungssysten um ein Mehrfaches steigern lässt.

Der logische Weg, die Leistung eines Verarbeitungssystems, das auf Prozessortechnologie basiert (Embedded System), zu steigern, ist der Einsatz von mehreren Prozessoren. Seit der Preis der Mikroprozessoren nicht mehr die entscheidende Rolle eines Embedded Systems spielt, werden solche Multiprozessorsysteme gebaut. Grundsätzlich gibt es zwei verschiedene Methoden, um zwei oder mehrere Prozessoren miteinander zu koppeln.

- Multiprozessorsystem mit gemeinsamem Speicher

  Bei Multiprozessorsystemen mit wenig Prozessoren ist es möglich, einen zentralen Speicher zu benützen. Die Prozessoren müssen jedoch über ein Mehr-level-cache verfügen, damit der Zentralspeicher alle Anfragen verarbeiten mag.

- Multiprozessorsystem mit geteiltem Speicher

  Bei Multiprozessorsystemen mit einer grösseren Anzahl von Prozessoren reicht die Bandbreite eines zentralen Speichers nicht mehr aus. Jeder Prozessor erhält seinen eigenen Speicher. Ein Netzwerk verbindet alle diese Prozessor-Speicher-Einheiten und sorgt für den Datenaustausch.

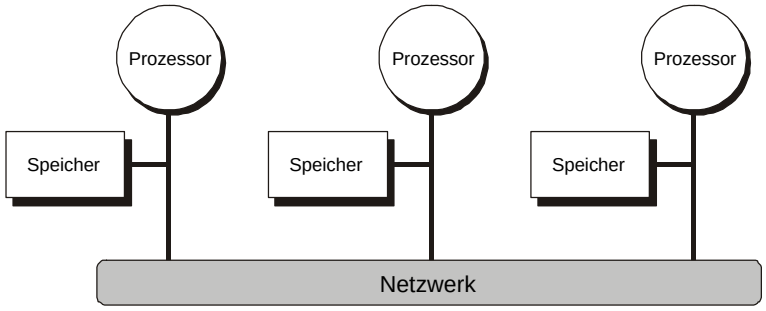


Abbildung 25: Multiprozessorsystem mit geteiltem Speicher

Das grössere Problem als die Hardware ist jedoch die Software. Oft ist es sehr schwierig einen Algorithmus für eine sinnvolle, parallele Abarbeitung aufzuteilen.

12.4.1 Mulitprozessorsystem bezogen auf die RISC_Core I in einem FPGA

Besteht eine Prozessor-Core als VHDL Beschreibung, ist der Schritt nicht mehr weit, dieselbe Core zweimal in einem FPGA abzubilden. Natürlich muss die Prozessor-Core so erweitert werden, dass sie eines der oben erwähnten Modelle erfüllen kann. Als Vorbild dazu könnte zum Beispiel der DSP ADSP-210X von Analog Device [L8] dienen. Diese DSP Familie ist mit einem sogenannten Prozessor-Link-Port ausgestattet, welcher es erlaubt, einen getrennten Speicher gegeneinander abzugleichen.

Wird die RISC_Core I mit einem solchen Link-Port erweitert, könnte ein Multiprozessor-system-on-Chip (mehrere Prozessoren auf dem selben Chip) erstellt werden. Viele Grossrechner besitzen auch mehrere Prozessorenkerne auf dem selben Chip. Dies spart viel Platz und die Kosten von teuren Gehäusen. Ein FPGA bietet diese Möglichkeit nun auch ohne die Notwendigkeit, ein ASP (Application Specific Processor) produzieren zu lassen.

13 Ergebnis

Als Abschliessender Teil dieser Diplomarbeit werden in diesem Kapitel die erzielten Ergebnisse besprochen. Dazu wird die Aufgabenstellung in sechs Bereiche aufgeteilt. Die unten aufgeführte Tabelle zeigt diese Aufgabenbereiche und die erreichten Ergebnisse.

Aufgabenbereich	Ergebnis
RISC-Prozessor für FPGA Signalver-arbeitungssysteme	Das Design des RISC-Prozessors (RISC-Core) wurde in einer Kombination aus einer schematischen Darstellung und einer VHDL Beschreibung gelöst. Der Speicherbereich sowie die PCI Ankopplung sind schematisch und die Zentraleinheit in VHDL gelöst. Der erarbeitete Prozessor besitzt eine RISC typische Harvard-Register-Architektur mit 8 Zentralregistern. Der Daten-, sowie der Programmcode-Bus sind 16 Bit breit. Auch die ALU unterstützt die Verarbeitung von 16 Bit breiten Datenwörtern. Die Taktfrequenz der RISC_Core I beträgt 12.5 MHz. Jegliche Befehle können innerhalb von einem Maschinenzyklus – welcher 4 Takte umfasst – abgearbeitet werden. Der Instruktionssatz ist auf 34 Befehle beschränkt und unterstützt Transfer-, Arithmetische / logische- sowie Kontroll-flussbefehl. Die RISC_Core I ist in der Lage konditionelle Sprünge zu tätigen und Unterprogramme aufzurufen. Als Erweiterung für Signalverarbeitungsaufgaben besitzt die RISC_Core I eine Multipliziereinheit.
Schnittstelle für Peripheriebausteine	Als Peripheriebaustein wurde für diese Diplomarbeit das AD/DA-Peripherie-Modul eingesetzt. Dementsprechend sind zwei 8-Bit Ports implementiert, welche über die LVDS Schnittstelle der Xilinx FPGA Karte mit diesem Modul kommunizieren können.
Implementierung auf der Xilinx PCI Karte	Die RISC_Core I wurde für den Xilinx Virtex 300 Baustein, welcher sich auf der PCI Karte befindet, entwickelt. Benutzt wird auch die LVDS Schnittstelle für das oben erwähnte Peripherie-Modul. Das Ansprechen des Programm- und Datenspeichers erfolgt über den PCI-Bus.
Assembler	Der Assembler ist als Zeileninterpreter für eine Textdatei ausgeführt. Er kennt die Direktiven EQU, ORG, END und erlaubt das Verwenden von Label. Weiter sind Registerbezeichnungen R0..R7 hart codiert und müssen nicht extra deklariert werden.

	Zusätzlich zum Assembler ist noch ein Loader erstellt worden, welcher benötigt wird, um den Programmcode in den Programmspeicher zu laden und als Debbug-Möglichkeit das Auslesen des Datenspeicher bietet.
Funktionsweise anhand einer Signalverarbeitungs-aufgabe	Um die Funktionsweise des Gesamtsystems anhand einer Signalverarbeitungsaufgabe zu demonstrieren, wurde ein Allpass realisiert. Als Allpass versteht sich das Einlesen eines analogen Wertes über die serielle SPI-Schnittstelle des AD/DA-Peripherie-Modules und die Ausgabe desselben Wertes über die selbe Schnittstelle.

13.1 Ausblick

Einen kleinen Ausblick, was mit einem solchen Design realisiert werden kann, hat schon das Kapitel 12 ‚Erweiterungen der RISC-Core für Signalverarbeitungssysteme' geboten. Es wäre aber möglich, den Freiheitsgrad eines solchen Systemes auf einem FPGA weiter zu erhöhen.

Eine Möglichkeit wäre das generische Erzeugen der Prozessor-Core durch die Angabe der Busbreite und der Speichergrösse. Eine generische Erzeugung heisst, dass während der Synthese alle Blöcke an die angegebene Busbreite angepasst werden. Wird eine Datenbusbreite von 4 gewählt, wird auch eine 4-Bit ALU generiert. Dasselbe gilt auch für den Speicher, solange dieser im FPGA realisiert wird. Aus der gewünschten Speicherangabe kann dann noch die Anzahl der Adressleitungen bestimmt werden.

Mit dieser Methode liessen sich Prozessoren mit einer beliebigen Busbreite generieren. In der Praxis hat sich jedoch gezeigt, dass eine Abweichung von einem 2^n-Bit breiten Bus wenig Sinn macht. Somit kann man sagen, dass von einem 4-Bit Nibblerechner bis hin zu einer 64-Bit Maschine (ev. sogar noch mehr, falls die Ressourcen des FPGA ausreichen) innerhalb kurzer Zeit der gewünscht Prozessor generiert werden kann. Jeder generierte Prozessor besitzt dieselbe Hardwarestruktur, denselben Instruktionssatz und auch dasselbe Phasenmodell. Natürlich macht es keinen Sinn, einen 4-Bit Rechner mit derselben Opcodelänge zu betreiben wie einen 64-Bit Rechner. Eine zusätzliche generische Erzeugung der Opcodelänge – nicht aber des Instruktionssatzes – wäre notwendig.

Eine sehr elegante Methode eines Designs wäre die Benutzung einer grafischen Oberfläche. Unter der Benutzung einer grafischen Oberfläche kann man sich quasi einen Schemaeditor vorstellen, in welchem einzelne Blöcke zusammengefügt werden können. Die oben erwähnte generische Erzeugung würde die Synthese der zusammengefügten Blöcke übernehmen. Dies würde dazu führen, dass an eine universelle Prozessor-Core (z.B. der RISC_Core I) die unterschiedlichsten Signalverarbeitungsblöcke (z.B. eine MAC oder ein FFT-Block) angeschlossen werden könnten. Das Design eines Multiprozessorsystem-on-Chip wäre mit einer solchen Entwicklungsumgebung denkbar einfach.

14 Verzeichnisse

14.1 Abbildungsverzeichnis

14.2 Tabellenverzeichnis

14.3 Verzeichnis der Beispiele

14.4 Eingesetzte Mittel

14.4.1 Hardware

[H1] Standard ATX PC mit Betriebssystem Windows 2000

[H2] Xilinx PCI-Karte der Fachhoschule für Technik St. Gallen

[H3] Xilinx Multilinx-Kabel

[H4] AD/DA Peripherie

14.4.2 Software

[S1] Xilinx: Foundation 3.3i mit Servicepack 8

[S2] Modeltech: Modelsim for Xilinx

[S3] Jungo: Windriver for PCI

[S4] PLXTECH: plxmon

[S5] Microsoft Visual C++ 6.0

[S6] Corel: Corel draw

[S7] Microsoft: Office 2000

14.4.3 Messgeräte

[M1] LeCroy Oszilloskop LT264M

14.5 Literaturverzeichnis

[L1] Christian Siemers: Prozessorbau, 1999, Carl Hanser München,

ISBN 3-446-19330-8

[L2] David A. Patterson und John L. Hennessy: Computer Organization & Design, 1998, Morgan Kaufmann, San Francisco

ISBN 1-55860-428-6

[L3] Jan Gray: Hands-on Computer Architecture – Teaching Processor and Integrated Systems Design with FPGA, 2000, Gray Research LLC,

www. fpgacpu.org

[L4] Thomas Flik und Hans Liebig: Mikroprozessortechnik, 1998, Springer-Verlag, Berlin

ISBN 3-54064019

[L5] David A. Patterson und John L. Hennessy: Computer Architecture a quantitative approach, 1996, Morgan Kaufmann, San Francisco,

ISBN 1-55860-329-8

[L6] Douglas Perry: VHDL, 1998, Mc Graw-Hill New York

ISBN 0-07-049436-3

[L7] Heinkel, Padeffke, Haas, Buerner, Braisz, Gentner and Grassmann: The VHDL Reference, 2000, John Wiley & Sons Chichester

ISBN 0-47-189972-0

[L8] ADSP-2100 Family User's Manual, Analog Devices Inc., 1995, Norwood Massachussetts

[L9] André Willms, C Programmierung, Addison-Wesley, 1998, Bonn

ISBN 3-8273-1405-4

14.6 Dokumente – Verzeichnis

[D1] PLXTECH: PCI9052 Data book / Erata sheet / Design notes

[D2] Xilinx Synthesis Technology (XST) User Guide

[D3] Diplomarbeit 2000, von U. Broger, Rekonfigurierbarer Basisbandprozessor

[D4] Semesterarbeit 2001, von M. Imhof, Inbetriebnahme eines Xilinx DSP-Boards

[D5] Mikroprozessortechnik Teil 1 von R. Widmer, 1999

14.7 Internet Links

[I1] www.arm.com

[I2] www.jungo.com

[I3] www.xilinx.xom

[I4] www.plxtech.com

[I5] www.mkp.com/books_catalog/catalog.asp

[I6] www.national.com

[I7] www.maxim-ic.com

15 Glossar

CISC:	Complex Instruction Set Computer. Die komplexen Befehle (z.B. Multiplikation) beruhen auf Mikroprogrammen, da mehrere Schritte pro Befehl zu bearbeiten sind. CISC steht im Gegensatz zu RISC.
CPI:	Clocks Per Instruction. Anzahl Takte pro Maschinenzyklus.
CPU:	Control Processing Unit.
Dualport RAM:	RAM Speicherbaustein mit doppelt ausgeführten Ports. Es kann von zwei Bussystemen aus auf den gleichen Speicherbereich zugegriffen werden.
Feedforward:	Datenvorschub, welcher bei einem Phasenpipelining eingeführt wird, um Datenhazards zu vermeiden.
FPGA:	Field Programmable Gate Array.
Harvard - Architektur:	Besonderheit der Harvard-, im Vergleich zur von Neuman-Architektur, ist eine Trennung im Bussystem von Daten- und Programmzugriff.
Memory map:	Zeigt die Speichereinteilung eines digitalen Systems.
MIPS:	MIPS ist eine klassische Assemblersprache mit expliziter Operandenangabe. Sie wurde von David Patterson und John Hennessy an der Universität Berkley entwickelt.
Opcode:	Operationscode. Opcode wird für den Maschinencode und den dafür stehenden mnemonischen Ausdruck verwendet.
PCI:	Peripheral Component Bus. 32-Bit Standard Plug and Play PC-Bus
Post Route Simulation:	Simulation des Zeitverhalten eines Digitalsystems unter Berücksichtigung der physikalischen Eigenschaften der Bauteile.
RISC:	Reduced Instruction Set Computer. Die RISC-Architektur beruht auf einer Minimierung des Befehlssatzes und Ausführung der Befehle durch eine optimierte Hardware.
Stall:	Wartezyklus, welcher bei einem Phasenpipelining eingefügt wird, um Datenhazards zu vermeiden.
Von Neumann - Architektur:	Grundlegendes Rechnermodell bestehend aus CPU, Ein- / Ausgabeeinheit und Speicher. Dieses Modell wurde 1946 von Burks, Goldstine und von Neumann veröffentlicht.
VHDL:	Very High Speed Integrated Circuit Description Language. Beschreibungssprache für binäre Digitalsysteme.

16 Anhang

16.1 Befehlssatz der RISC_Core I

16.1.1 Übersicht

Mnemonic	Argumente		Adressierungsart	Maschinencode	Flags
Transferbefehle					
MOV	Op1	Op2	Reg. Direkt	00 00 00 0 Op1 Op2 000	
MOVI	Op1 (R0..R3)	#const8	Immediate	00 01 00 Op1 #const8	
MOVIH	Op1 (R0..R3)	#const8	Immediate	00 01 01 Op1 #const8	
MOV	Op1	&Op2	Reg. Indirekt	00 00 01 0 Op1 Op2 000	
MOV	Op1 (R0..R3)	#const8+R0	Indiziert	00 01 10 Op1 #const8	
MOV	&Op1	Op2	Reg. Indirekt	00 00 10 0 Op1 Op2 000	
MOV	#const8+R0	Op1 (R0..R3)	Indiziert	00 01 11 Op1 #const8	
PUSH	Op1		Reg. Direkt	00 10 00 0 Op1 000 000	
POP	Op1		Reg. Direkt	00 10 01 0 Op1 000 000	
Arithmetische/logische Befehle					
ADD	Op1 Op2 Op3		Reg. Direkt	01 00 00 0 Op1 Op2 Op3	OV
ADDI	Op1 (R0..R3)	#const8	Immediate	01 00 01 Op1 #const8	OV
SUB	Op1 Op2 Op3		Reg. Direkt	01 00 10 0 Op1 Op2 Op3	NF
SUBI	Op1 (R0..R3)	#const8	Immediate	01 00 11 Op1 #const8	NF
CMP	Op1	Op2	Reg. Direkt	01 01 00 0 Op1 Op2 000	ZF, NF
CMPI	Op1 (R0..R3)	#const8	Immediate	01 01 01 Op1 #const8	ZF, NF
DEC	Op1		Reg. Direkt	01 01 10 0 Op1 000 000	ZF, NF
INC	Op1		Reg. Direkt	01 01 11 0 Op1 000 000	OV
AND	Op1 Op2 Op3		Reg. Direkt	01 10 00 0 Op1 Op2 Op3	CF
ANDI	Op1 (R0..R3)	#const8	Immediate	01 10 01 Op1 #const8	CF
EOR	Op1 Op2 Op3		Reg. Direkt	01 10 10 0 Op1 Op2 Op3	
EORI	Op1 (R0..R3)	#const8	Immediate	01 10 11 Op1 #const8	
OR	Op1 Op2 Op3		Reg. Direkt	01 11 00 0 Op1 Op2 Op3	
ORI	Op1 (R0..R3)	#const8	Immediate	01 11 01 Op1 #const8	
SL	Op1		Reg. Direkt	10 00 00 0 Op1 000 000	CF
SR	Op1		Reg. Direkt	10 00 01 0 Op1 000 000	CF
RL	Op1		Reg. Direkt	10 00 10 0 Op1 000 000	
RR	Op1		Reg. Direkt	10 00 11 0 Op1 000 000	
MUL	Op1 Op2 Op3		Reg. Direkt	10 01 00 0 Op1 000 000	
Flagbefehle					
SF	Flag (Flg)		Implizit	10 10 01 0 Flg 000 000	
CF	Flag (Flg)		Implizit	10 10 00 0 Flg 000 000	
Kontrollflussbefehle					
NOP	-			11 10 00 00 0000 0000	
JMP	#const10		Absolut	11 11 11 #const10	
JCS	#const8		Relativ	11 01 01 00 #const8	
JCC	#const8		Relativ	11 01 00 00 #const8	
JEQ	#const8		Relativ	11 01 11 00 #const8	
JNE	#const8		Relativ	11 01 10 00 #const8	
JOS	#const8		Relativ	11 00 01 00 #const8	
JOC	#const8		Relativ	11 00 00 00 #const8	
JNS	#const8		Relativ	11 00 11 00 #const8	
JNC	#const8		Relativ	11 00 10 00 #const8	
CALL	#const10		Absolut	11 11 00 #const10	
RET	[Zielreg(pc)]	[Quellreg(stack)]	Implizit	11 11 01 00 0000 0000	
RETI	[Zielreg(pc)]	[Quellreg(stack)]	Implizit	11 11 10 00 0000 0000	

16.1.2 Beschreibung

16.1.2.1 Transferbefehle

mov Op1, Op2;
Das Register Op1 wird mit dem Wert vom Register Op2 geladen.
<Op1> ← <Op2>

---

mov Op1, &Op2;
Das Register Op1 wird mit dem Wert aus der Speicherzelle geladen, auf die das Register Op2 zeigt.
<Op1> ← Data(Op2)

---

mov Op1, &#const8
Das Register Op1 (R0..R3) wird mit dem Wert aus der Speicherzelle mit der Adresse #const8 + R0 geladen.
<Op1> ← Data(#const8 + R0)

---

mov &Op1, Op2;
Die Speicherzelle mit der Adresse Op1 wird mit dem Inhalt vom Register Op2 (R0..R8) geladen.
Data(Op1) ← <Op2>

---

mov &#const8, Op2;
Die Speicherzelle mit der Adresse #const8 + R0 wird mit dem Wert vom Register Op2 (R0..R3) geladen.
Data(#const8 + R0) ← <Op2>

---

movi Op1, #const8;

Das Register Op1 (R0..R3) wird mit einem konstanten 8 Bit Wert (#const8) geladen.

<Op1> ← #const8

---

movih Op1, #const8;
Das Highbyte des Registers Op1 (R0..R3) wird mit einem konstanten 8 Bit Wert (#const8) geladen. Das Lowbyte wird dabei nicht überschrieben!
<Op1 (Highbyte)> ← #const8

---

push Op1;
Das Register Op1 wird auf den Stack geschrieben.
Stack ← <Op1>, Stackpointer ← Stackpointer + 1

---

pop Op1;
Das Register Op1 wird mit dem Inhalt des Stacks geladen.
<Op1> ← Stack(Stackpointer-1)

16.1.2.2 Arithmetische und logische Befehle

add Op1, Op2, Op3;
Addiert die Register Op2 und Op3 und schreibt das Resultat ins Register Op1. Das OV Flag wird gesetzt, falls die Addition einen Überlauf erzielt.
<Op1> ← <Op2> + <Op3>

---

addi Op1, #const8;
Addiert eine 8-Bit Konstant zum Register Op1 (R0..R3). Das Resultat wird ins Register R0 geschrieben. Das OV Flag wird gesetzt, falls die Addition einen Überlauf erzielt.
<R0> ← <Op1> + #const8

---

sub Op1, Op2, Op3;
Subtrahiert das Register Op3 vom Register Op2 und schreibt das Resultat ins Register Op3. Das NF Flag wird gesetzt wenn Op3 grösser Op2 ist. Sind Op2 und Op3 gleich gross, wird das ZF Flag gesetzt.
<Op1> ← <Op2> - <Op3>

---

subi Op1, #const8;
Subtrahiert eine 8-Bit Konstante vom Register Op1. Das Resultat wird ins Register 0 geschrieben. Die Flagbeeinflussung ist dieselbe wie beim sub-Befehl.
<R0> ← <Op2> - #const8

---

and Op1, Op2, Op3;
Verknüpft das Register Op2 und Op3 durch ein bitweises, logisches UND. Das Resultat wird ins Register Op1 geschrieben. Das Carry Flag wird gesetzt, falls ein logischer Übertrag stattfindet.
<Op1> ← <Op2> • <Op3>

---

andi Op1, #const8;
Verknüpft das Register Op1 mit einer 8-Bit Konstanten durch ein bitweises, logisches UND. Das Resultat wird ins Register 0 geschrieben. Das Carry Flag wird gesetzt, falls ein logischer Übertrag stattfindet.
<R0> ← <Op1> • #const8

---

or Op1, Op2, Op3;
Verknüpft das Register Op2 und Op3 durch ein bitweises, logisches ODER und schreibt das Resultat ins Register Op1.
<Op1> ← <Op2> + <Op3>

---

ori Op1, #const8;
Verknüpft das Register Op1 mit einer 8-Bit Konstanten durch ein bitweises, logisches ODER und schreibt das Resultat ins Register R0.
<R0> ← <Op1> + #const8

---

eor Op1, Op2, Op3;
Verknüpft das Register Op2 und Op3 durch ein bitweises, logisches EXKLUSIV ODER und schreibt das Resultat ins Register Op1.
<Op1> ← <Op2> ⊕ <Op3>

---

eori Op1, #const8;
Verknüpft das Register Op1 mit einer 8-Bit Konstanten durch ein bitweises, logisches EXKLUSIV ODER und schreibt das Resultat ins Register R0.
<R0> ← <Op1> ⊕ #const8

cmp Op1, Op2;
Vergleicht das Register Op1 und Op2. Bei Gleichheit wird das ZF Flag gesetzt. Ist Op2 grösser Op1 wird das NF Flag gesetzt.

cmpi Op1, #const8;
Vergleicht das Register Op1 mit einer 8-Bit Konstanten. Bei Gleichheit wird das ZF Flag gesetzt. Ist Op2 grösser Op1 wird das NF Flag gesetzt.

dec Op1;
Vermindert den Inhalt des Registers Op1 um 1. Das ZF Flag wird gesetzt wenn das Resultat 0 ergibt. Das NF Flag wird gesetzt, wenn das Resultat FFFF ergibt.

inc Op1;
Erhöht den Inhalt des Registers Op1 um 1. Bei einem Überlauf wird das OV Flag gesetzt.

sl Op1;
Schiebt den Inhalt des Registers Op1 um 1 nach links. Das MSB wird ins Carry Flag verschoben.

sr Op1;
Schiebt den Inhalt des Registers Op1 um 1 nach rechts. Das LSB wird ins Carry Flag verschoben.

rl Op1;
Rotiert den Inhalt des Registers Op1 um 1 nach links.

rr Op1;
Rotiert den Inhalt des Registers Op1 um 1 nach rechts.

mul Op1, Op2, Op3;
Multipliziert das Register Op2 mit dem Register Op3 und schreibt das Resultat ins Register Op1.
<Op1> = <Op2> * <Op3>

16.1.2.3 Flagbefehle

sf Flag;
Setzt das durch <Flag> angegebene Flag. Flag ist eine 3-Bit Konstante mit folgender Auflösung:

"000"	→	Carry Flag
"001"	→	Zero Flag
"010"	→	Negativ Flag
"011"	→	Overflow Flag
"100"	→	Interrupt Flag

cf Flag;
Löscht das angegebene Flag. Die Auflösung der Konstante <Flag> entspricht dieser des sf Befehls.

16.1.2.4 Kontrollflussbefehle

nop;
Der nop Befehl führt während einem Maschinenzyklus – bis auf das Erhöhen des Programmzählers - keine Aktionen aus.

jcs #const8; oder jcs <label>;
Verursacht einen relativen Sprung zum momentanten Programmpunkt, wenn das Carry-Flag gesetzt ist. Die Sprungweite wird durch eine 8-Bit Konstante angegeben. Wird ein Label verwendet, errechnet der Assembler die Sprungweite.

<pc> = <pc> + #const8, <pc> = <pc> + <label – pc>

jcc #const8; oder jcc <label>;
Relativer Sprung zum momentanen Programmpunkt, wenn das Carry-Flag gelöscht ist. Die Sprungweite wird durch eine 8-Bit Konstante angegeben. Sprungangabe und Berechnung wie beim jcs Befehl.

jeq #const8; oder jzs <label>;
Relativer Sprung zum momentanen Programmpunkt, wenn das Zero-Flag gesetzt ist. Sprungangabe und Berechnung wie beim jcs Befehl.

jne #const8; oder jzc <label>;
Relativer Sprung zum momentanen Programmpunkt, wenn das Zero-Flag gelöscht ist. Sprungangabe und Berechnung wie beim jcs Befehl.

jns #const8; oder jns <label>;
jump if **n**egativ flag **s**et. Relativer Sprung zum momentanen Programmpunkt, wenn das Negativ-Flag gesetzt ist. Sprungangabe und Berechnung wie beim jcs Befehl.

jnc #const8; oder jnc <label>;
jump if **n**egativ flag **c**lear. Relativer Sprung zum momentanen Programmpunkt, wenn das Negativ-Flag gelöscht ist. Sprungangabe und Berechnung wie beim jcs Befehl.

jos #const8; oder jos <label>;
jump if **o**verflow flag **s**et. Relativer Sprung zum momentanen Programmpunkt, wenn das Overflow-Flag gesetzt ist. Sprungangabe und Berechnung wie beim jcs Befehl.

joc #const8; oder joc <label>;
jump if **o**verflow flag **c**lear. Relativer Sprung zum momentanen Programmpunkt, wenn das Overflow-Flag gelöscht ist. Sprungangabe und Berechnung wie beim jcs Befehl.

jmp #const8; oder jmp <label>;
Absoluter Sprung an die Programmadresse #const8 bzw <label>

<pc> ← #const8, <label>

call #const8; oder call <label>;
Absoluter Sprung an die Programmadresse #const8 bzw. <label> und Sicherung der Absprungadresse auf dem Stack.

<pc> ← #const8, <label>

ret;
Unterprogramm Rücksprung. Absoluter Sprung an die im Stack abgelegte Programmadresse.

<pc> ← <stack(stackpointer – 1)>

16.2 Konfiguration des PCI Controllers PCI9052

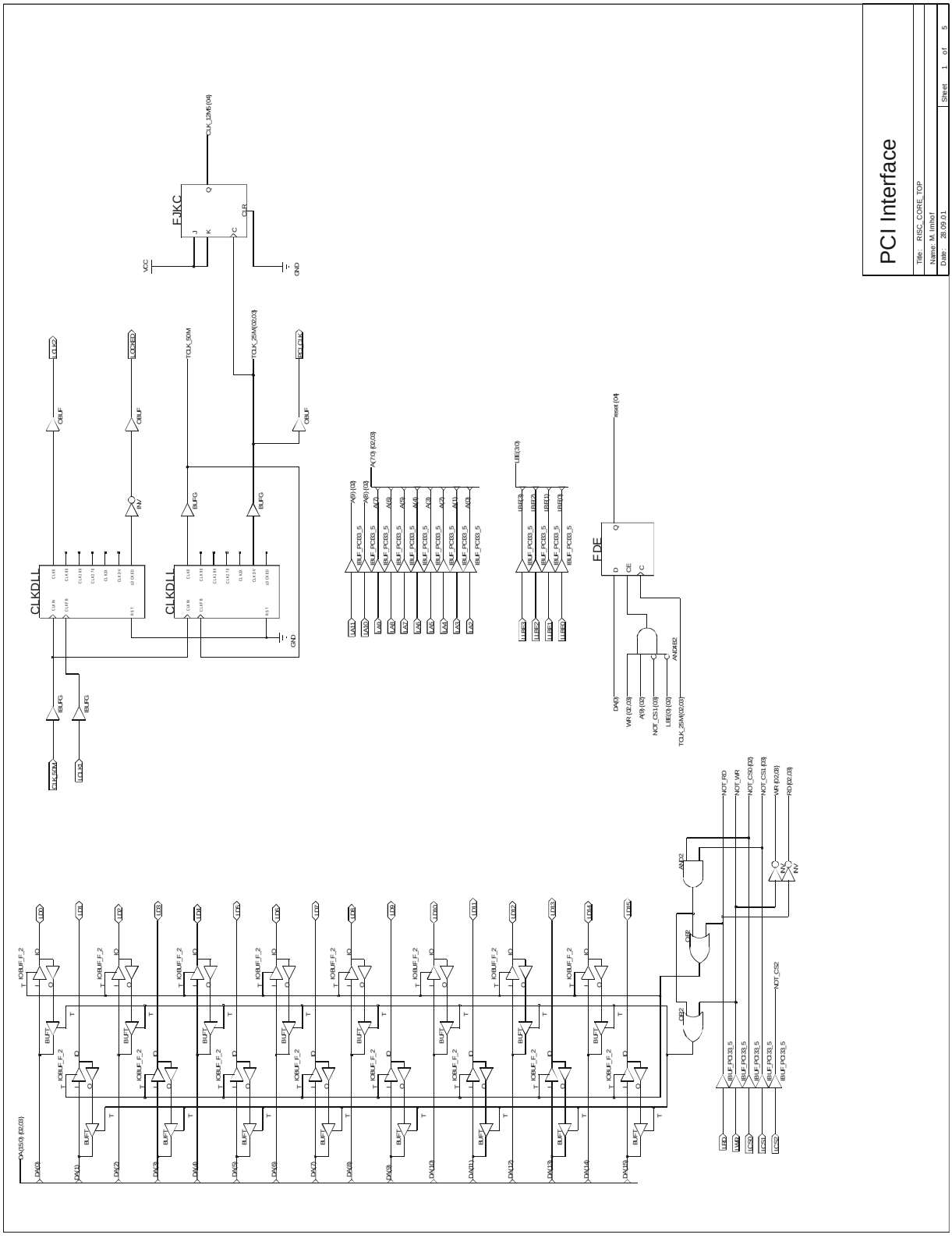
Für die Realisierung des memory map (vgl. Realisierungskonzept) muss der PCI Controller konfiguriert werden. Dieser lädt die Registerinhalte beim Aufstarten aus einem externen EEPROM welches mit Hilfe des Programms „plxmon.exe“ [S4] beschrieben werden kann. Das Programm „plxmon.exe“ wird vom Hersteller des Controllers zur Verfügung gestellt.

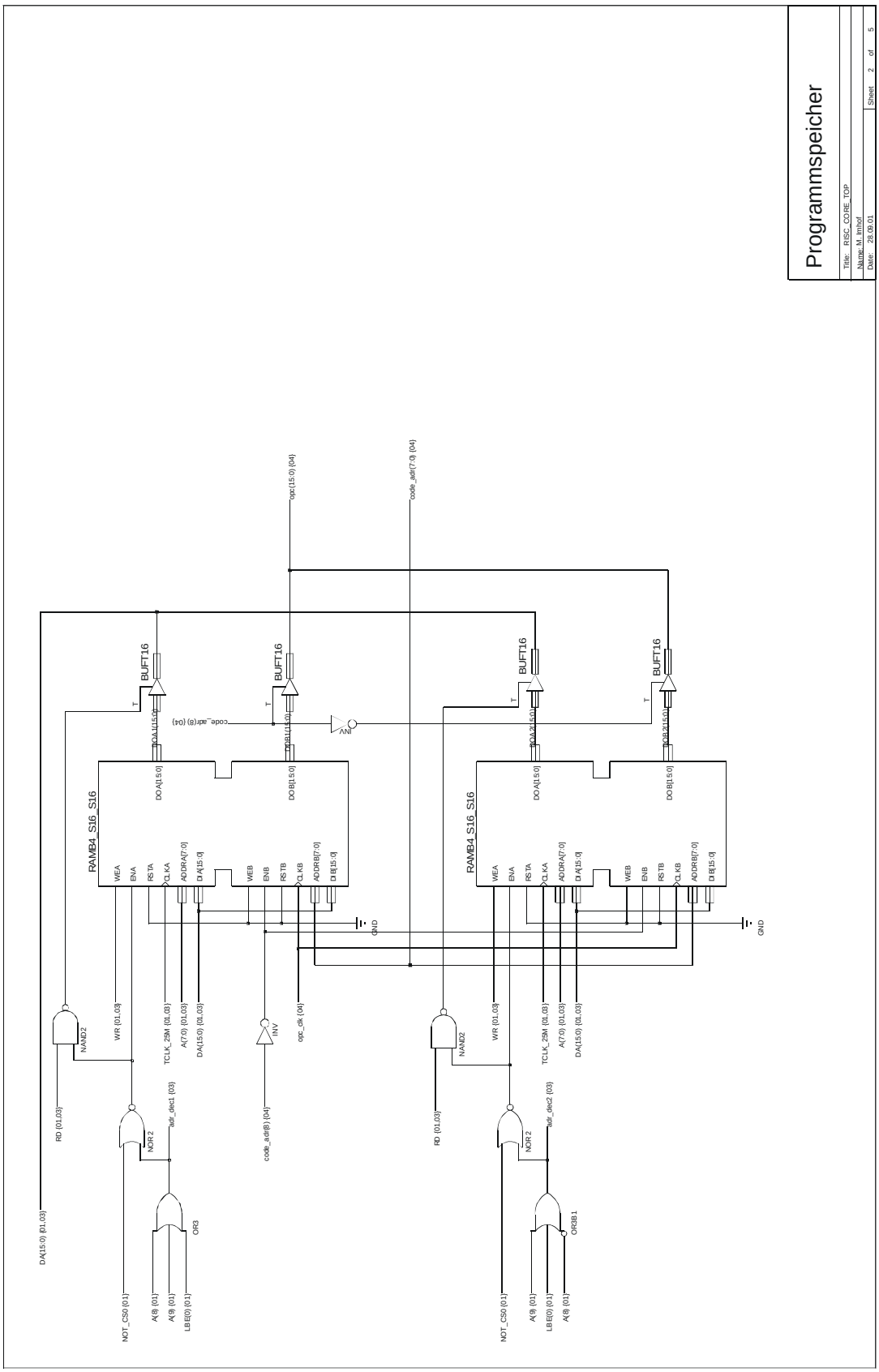
Adresse	31 **Konfigurations Register** 0	**Wert**
00h	Local Addr. Space 0 Range	FFFFF000h
04h	Local Addr. Space 1 Range	FFFFF000h
08h	Local Addr. Space 2 Range	00000000h
0Ch	Local Addr. Space 3 Range	00000000h
10h	Local Expansion R=M Range	00000000h
14h	Local Addr. Space 0 Local Base Addr.	00001001h
18h	Local Addr. Space 1 Local Base Addr.	00002001h
1Ch	Local Addr. Space 2 Local Base Addr.	00000000h
20h	Local Addr. Space 3 Local Base Addr.	00000000h
24h	Expansion ROM Local Base Addr.	00000000h
28h	Local Addr. Space 0 Bus Descriptors	00400045h
2Ch	Local Addr. Space 1 Bus Descriptors	00400045h
30h	Local Addr. Space 2 Bus Descriptors	00000000h
34h	Local Addr. Space 3 Bus Descriptors	00000000h
38h	Expansion ROM Bus Descriptors	00000000h
3Ch	Chip Select 0 Base Addr.	00001801h
40h	Chip Select 1 Base Addr.	00002801h
44h	Chip Select 2 Base Addr.	00000000h
48h	Chip Select 3 Base Addr.	00000000h
4Ch	Interrrupt Control / Status	00000041h
50h	Serial EEPROM Control / PCI Slave	08784AC6h

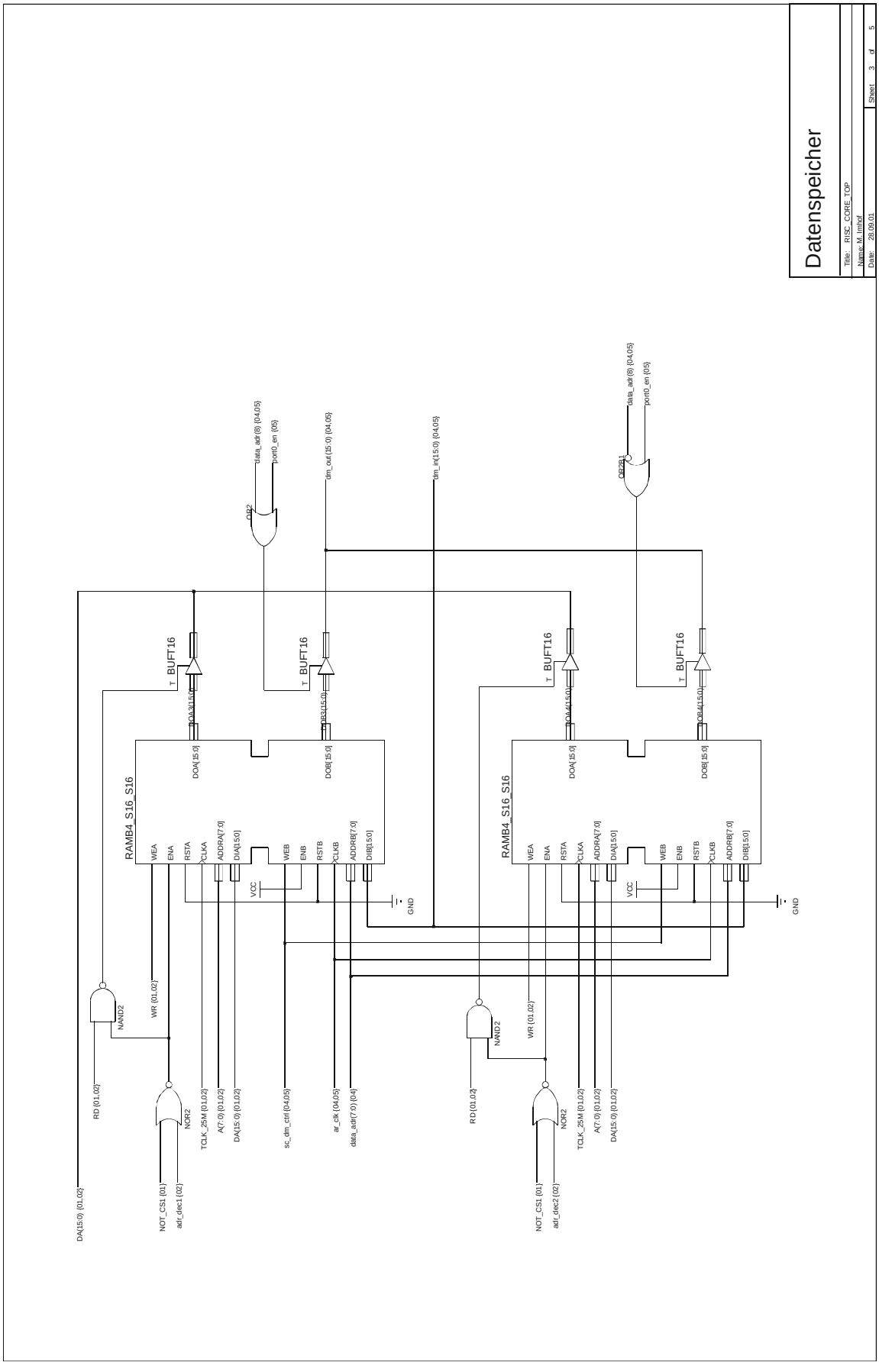
Abbildung 26: Konfiguration PCI Controller

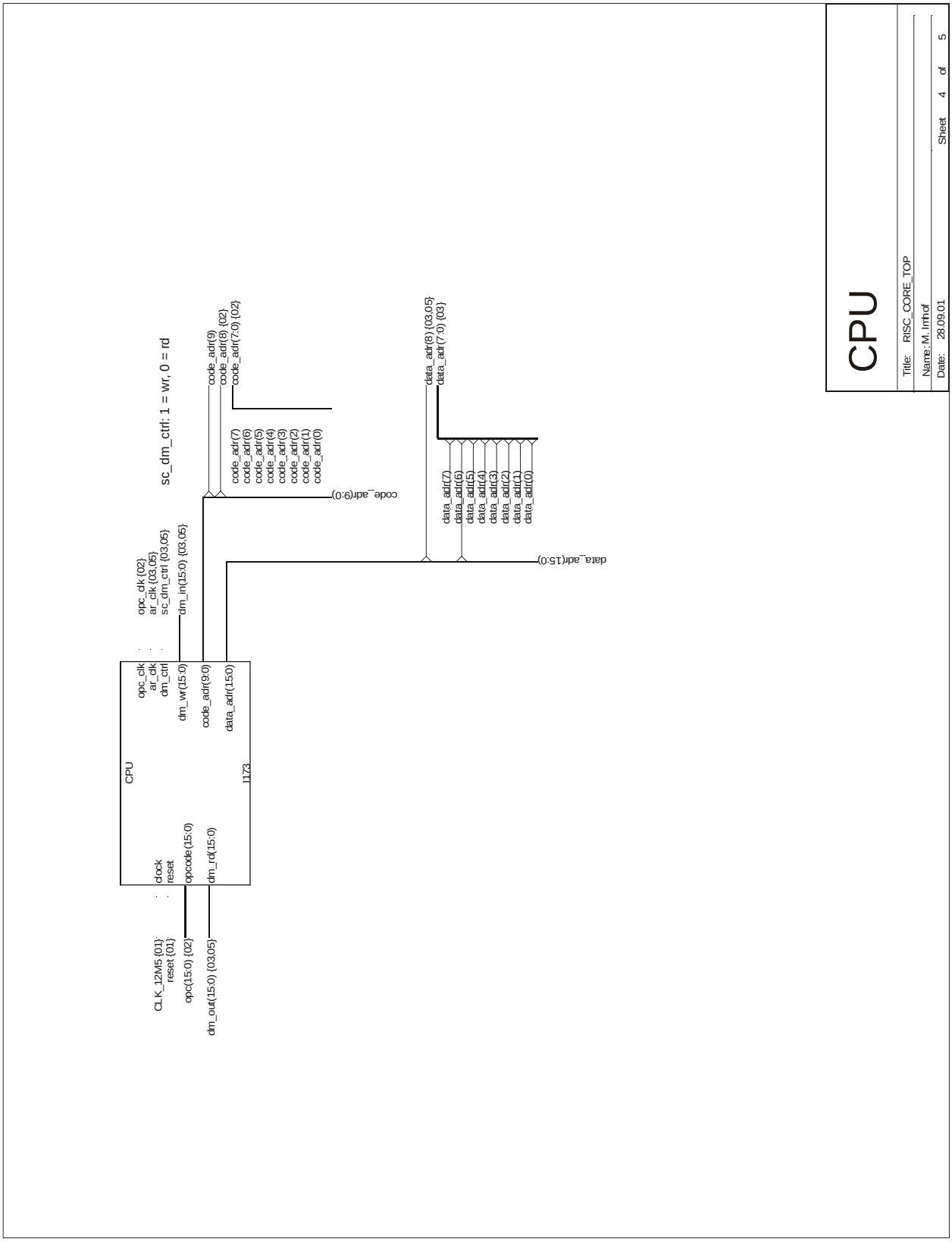
Die Bedeutung der massgebenden Registerinhalte kann dem Datenblatt des PCI-Controllers [D1] entnommen werden.

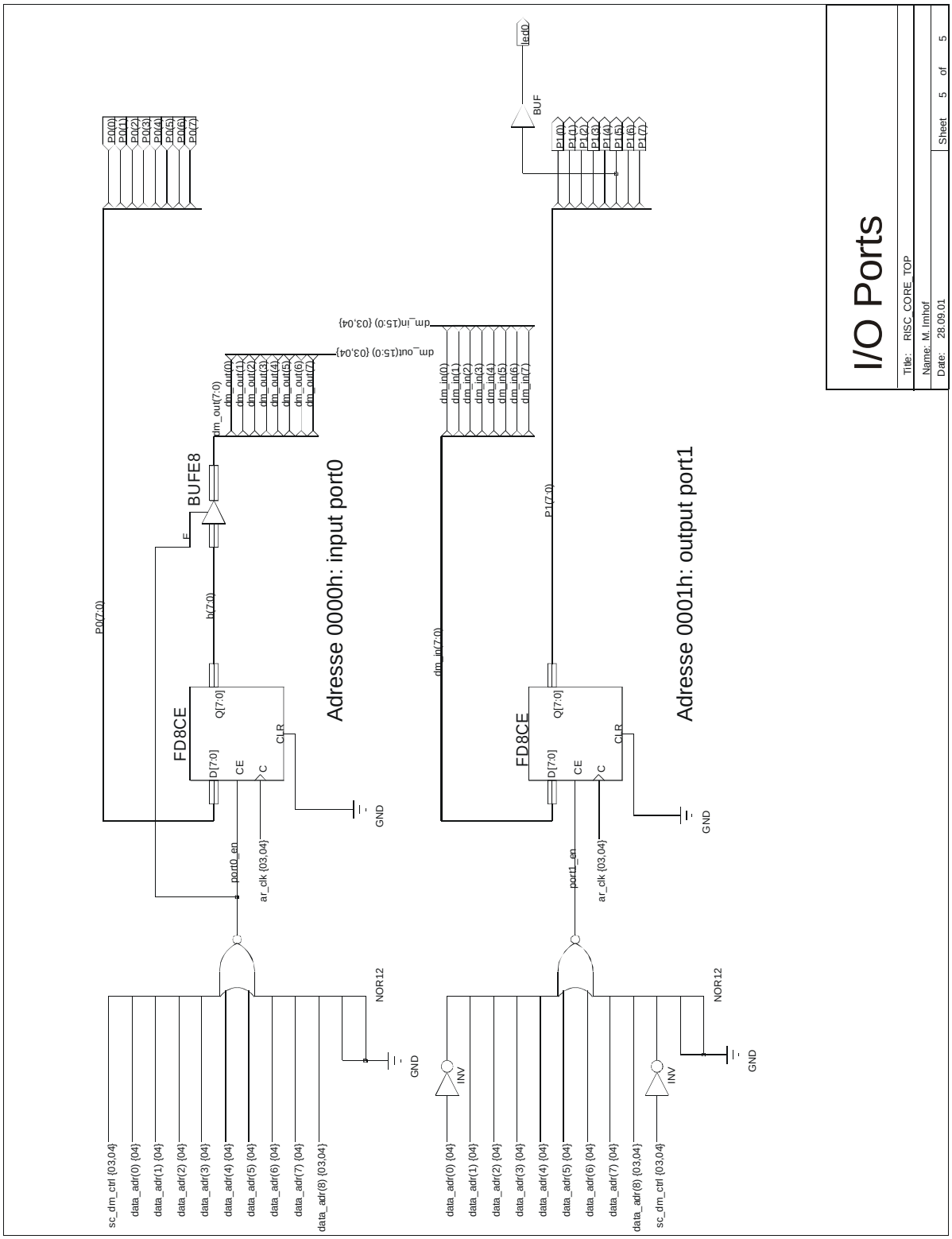
16.3 Schema RISC_Core I



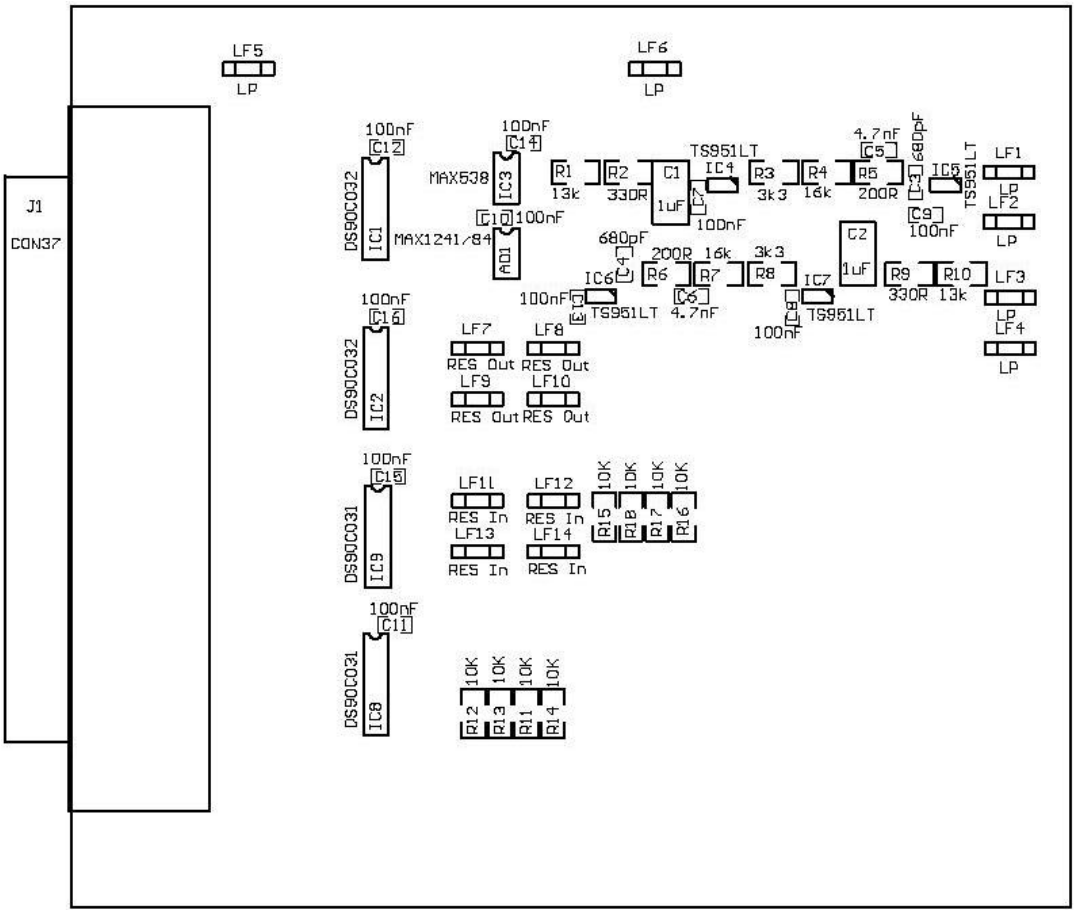

Programmspeicher
Title: RISC_CORE_TOP
Name: M. Imhof
Date: 28.09.01
Sheet 2 of 5
opc(15:0) (04)
code_adr(7:0) (04)
BUFT16
BUFT16
BUFT16
BUFT16
code_adr(8) (04)
INV
RAMB4_S16_S16
RAMB4_S16_S16
WEA
ENA
RSTA
CLKA
ADDRA[7:0]
DI A[15:0]
DOA[15:0]
WEB
ENB
RSTB
CLKB
ADDRB[7:0]
DI B[15:0]
DOB[15:0]
GND
GND
NAND2
NAND2
INV
NOR2
NOR2
OR3
OR3B1
WR (01,03)
TCLK_29M (01,03)
A(7:0) (01,03)
DA(15:0) (01,03)
opc_clk (04)
RD (01,03)
adr_dec1 (03)
adr_dec2 (03)
DA(15:0) (01,03)
NOT_CS0 (0:1)
A(9) (01)
A(9) (01)
LBE(0) (0:1)
A(9) (01)


Datenspeicher
Title: RISC_CORE_TOP
Name: M. Imhof
Date: 28.09.01
Sheet 3 of 5
RAMB4_S16_S16
BUFT16
NAND2
NOR2
OR2
WEA
ENA
RSTA
CLKA
ADDRA[7:0]
DIA[15:0]
DOA[15:0]
WEB
ENB
RSTB
CLKB
ADDRB[7:0]
DIB[15:0]
DOB[15:0]
VCC
GND
WR (01,02)
RD (01,02)
TCLK_25M (01,02)
A(7:0) (01,02)
DA(15:0) (01,02)
NOT_CS1 (01)
adr_dec1 (02)
adr_dec2 (02)
sc_dm_ctrl (04,05)
ar_clk (04,05)
data_adr(7:0) (04)
dm_out(15:0) (04,05)
dm_in(15:0) (04,05)
data_adr(8) (04,05)
port0_en (05)


CPU
Title: RISC_CORE_TOP
Name: M. Imhof
Date: 28.09.01
Sheet 4 of 5
CPU
I173
opc_clk
ar_clk
dm_ctrl
dm_wr(15:0)
code_adr(9:0)
data_adr(15:0)
clock
reset
opcode(15:0)
dm_rd(15:0)
opc_clk {02}
ar_clk {03,05}
sc_dm_ctrl {03,05}
dm_in(15:0) {03,05}
sc_dm_ctrl: 1 = wr, 0 = rd
code_adr(9)
code_adr(8) {02}
code_adr(7:0) {02}
code_adr(7)
code_adr(6)
code_adr(5)
code_adr(4)
code_adr(3)
code_adr(2)
code_adr(1)
code_adr(0)
code_adr(9:0)
data_adr(8) {03,05}
data_adr(7:0) {03}
data_adr(7)
data_adr(6)
data_adr(5)
data_adr(4)
data_adr(3)
data_adr(2)
data_adr(1)
data_adr(0)
data_adr(15:0)
CLK_12M5 {01}
reset {01}
opc(15:0) {02}
dm_out(15:0) {03,05}


I/O Ports
Title: RISC_CORE_TOP
Name: M. Imhof
Date: 28.09.01
Sheet 5 of 5
Adresse 0000h: input port0
Adresse 0001h: output port1
BUFE8
FD8CE
NOR12
INV
BUF
GND
P0(7:0)
P1(7:0)
dm_out(7:0)
dm_in(7:0)
dm_out(15:0) {03,04}
dm_in(15:0) {03,04}
port0_en
port1_en
ar_clk {03,04}
led0
sc_dm_ctrl {03,04}
data_adr(0) {04}
data_adr(1) {04}
data_adr(2) {04}
data_adr(3) {04}
data_adr(4) {04}
data_adr(5) {04}
data_adr(6) {04}
data_adr(7) {04}
data_adr(8) {03,04}

16.4 AD/DA Peripherie



17 Ehrenwörtliche Versicherung

Ich versichere hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig und nur unter Benützung der angeführten Quellen und Hilfsmittel angefertigt habe. Sämtliche Entlehnungen sind durch Quellenangaben kenntlich gemacht.

Ort, Datum: 3. Oktober 2001

Unterschrift: