Studienarbeit

„Entwurf eines einfachen Prozessors zu Demonstrationszwecken“

im Studiengang

Elektrotechnik (Automation)

an der dualen Hochschule Baden-Württemberg Mannheim

von

Name, Vorname	Ditinger, Alexander
Abgabedatum	02.01.2025
Bearbeitungszeitraum	30.09.2024 – 02.01.2025
Matrikelnummer, Kurs	8168790, TEL22AT1
Ausbildungsfirma	SCHOTT AG, Mainz
Betreuer*in	Strahler, Tristan

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich meine Projektarbeit mit dem Thema „Entwurf eines einfachen Prozessors zu Demonstrationszwecken“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Studierende*r

Abstract

Deutsch:

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf eines einfachen Prozessors zu Demonstrationszwecken mit dem Ziel, einen funktionsfähigen Prozessor zu entwerfen und zu testen. Dazu werden im ersten Teil die notwendigen Grundlagen vermittelt. Dazu zählen der grundlegende Aufbau der Mikroarchitektur aktueller Prozessoren, das Vergleichen verschiedener Designphilosophien und das Analysieren typischer Befehlssätze von Mikroprozessoren im unteren Leistungsspektrum. Im zweiten Teil erfolgt der Entwurf des Prozessors basierend auf den Grundlagen und der Zieldefinition, indem ein entsprechender Befehlssatz und die Mikroarchitektur entworfen werden. Zuletzt erfolgt die Implementierung des Prozessors in einer Hardwarebeschreibungssprache zur Validierung des Designs.

English:

This paper deals with the design of a simple processor for demonstration purposes with the aim of designing and testing a functional processor. In the first part, the necessary basics are summarised. This includes the basic structure of the microarchitecture of current processors, the comparison of different design philosophies and the analysis of typical instruction sets of microprocessors in the lower performance spectrum. In the second part, the processor is designed by firstly implementing an instruction set and secondly implementing the corresponding microarchitecture Finally, the processor is implemented in a hardware description language to validate the design.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ALU	Arithmetisch Logische Einheit
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit, Hauptprozessor
GPR	General Purpose Register, Allzweckregister
HDL	Hardware Description Language, Hardwarebeschreibungssprache
IC	Integrated Circuit
IR	Instruction Register, Befehlsregister
ISA	Instruction Set Architecture, Befehlssatzarchitektur
LIFO	Last In First Out
MAR	Memory Address Register, Adressregister
MUX	Multiplexer
PC	Program Counter, Befehlszähler
Rd	Zielregister
RISC	Reduced Instruction Set Computer
Rs	Quellregister
SCR	Status and Control Register
SoC	System-on-a-Chip
SP	Stackpointer
UP	Unterprogramm

Abbildungsverzeichnis

1. Einführung

1.1 Motivation

Da digitale Systeme in allen Anwendungsgebieten immer komplexere Funktionen erfüllen müssen, ist der Einsatz von frei programmierbaren Prozessoren sehr stark verbreitet. Zu den Einsatzgebieten zählen unter anderem Embedded Systems (z.B. in der Haushaltsgeräteelektronik oder Fahrzeugtechnik), Mikrocontroller bzw. Systems-on-a-Chip (SoC) oder der Hauptprozessor als Komponente von Computern oder Laptops (Central Processing Unit, CPU). Die Komplexität dieser Geräte bzw. der eingesetzten Prozessoren nimmt stetig zu, sodass die genauere Funktionsweise nur noch schwer nachvollziehbar ist. Daher widmet sich diese Arbeit der Entwicklung eines einfachen Prozessors zu Demonstrationszwecken, der sich an den grundlegenden Funktionsmechanismen aktueller Prozessoren aus dem unteren Leistungsspektrum orientiert, um so die allgemeine Arbeits- und Funktionsweise von Prozessoren darzustellen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist der vollständige Entwurf eines funktionsfähigen Prozessors basierend auf aktuellen Entwicklungen und Implementierungen. Dazu zählen das Befehlsset als primäre Beschreibungsschnittstelle des Prozessors und eine passende Mikroarchitektur zur Implementierung des Befehlssets als fertiger Prozessor. Das Befehlsset soll sowohl grundlegende Befehle als auch einige weiterführende Befehle unterstützen und erweiterbar sein. Auch die Mikroarchitektur soll einfach gehalten werden, aber gleichzeitig auf weiterführende Prinzipien eingehen und auch erweiterbar sein. Der fertige Entwurf soll real umsetzbar und funktionsfähig sein und für weitere, an diese Arbeit anknüpfende Arbeiten verwendbar sein (z.B. Erweiterung des Befehlssatzes, reale Implementierung, Compiler, …)

1.3 Vorgehensweise

Zuerst werden die notwendigen Grundlagen für einen Prozessorentwurf gelegt. Dazu zählt der grundlegende Prozessoraufbau, der Inhalt eines Befehlssatzes, die Analyse existierender Befehlssätze und das Vorstellen verschiedener Designphilosophien von Prozessoren.

Anschließend wird aufbauend auf den Grundlagen ein neuer Befehlssatz für den Prozessor dieser Arbeit entworfen, der seinen Funktionsumfang festlegt. Basierend

auf dem Befehlssatz wird eine entsprechende Mikroarchitektur entwickelt, die den Befehlssatz implementiert.

Im Anschluss an die theoretische Modellbildung des Prozessors folgt die Implementierung des Aufbaus mithilfe einer Hardwarebeschreibungssprache (Hardware Description Language, HDL) und die anschließende Simulation, um die Funktionalität des Designs zu validieren. Der HDL-Code soll die Erweiterbarkeit des Befehlssatzes bzw. der Mikroarchitektur beibehalten.

2. Grundlagen

Die folgenden Unterkapitel widmen sich dem allgemeinen Entwicklungsstand aktueller Prozessoren. Da laut Zielsetzung dieser Arbeit lediglich ein einfacher Prozessor entworfen werden soll, der die grundlegenden Prinzipien der Arbeitsweise von Prozessoren vermittelt, beschränken sich die Grundlagen auf solche Konzepte. CPUs, die in Computern eingesetzt werden (beispielsweise solche von Intel oder AMD), verwenden weitere, deutlich komplexere Konzepte, um entsprechende Verarbeitungsgeschwindigkeiten zu erreichen, die jedoch den Rahmen dieser Arbeit überschreiten und nicht in die Zielsetzung, einen Prozessor zu Demonstrationszwecken zu entwickeln, passen.

Zunächst folgt eine Abgrenzung unterschiedlicher Begrifflichkeiten zum Thema.

Prozessor bezeichnet allgemein eine komplexe, digitale Schaltung, die in der Lage ist, ein Programm (bestehend aus einzelnen Befehlen) abzuarbeiten.

Die meisten komplexeren Prozessoren bestehen aus mehreren einzelnen *Prozessorkernen*, wobei jeder dieser Prozessorkerne (grob) aus einer *Steuer-* und einer *Recheneinheit* besteht. Prinzipiell ist jeder dieser Kerne selbst ein Prozessor, der im Falle eines mehrkernigen Prozessors wiederverwendbar und modular konzeptioniert ist, und mit entsprechender Beschaltung daher den mehrkernigen Prozessor bildet. Ein solcher Prozessor wird oft als *CPU* bezeichnet, die von weiteren Coprozessoren (z.B. Gleitkommaarithmetik, Grafik, …) unterstützt wird.

Mikroprozessoren bezeichnen allgemein verschiedene Prozessoren, die auf einem einzelnen Mikrochip realisiert sind. Da heutzutage alle Prozessoren als Integrated Circuit (IC) ausgeführt sind, sind die Begriffe Mikroprozessor und Prozessor gleichbedeutend.

Mikrocontroller sind Mikroprozessoren, die um weitere notwendige Komponenten ergänzt wurden und so ein Gesamtsystem aus Prozessor und Peripheriefunktionen (z.B. IO-Ports oder auch komplexen Kommunikationsschnittstellen) auf einen Mikrochip bilden. Diese finden häufig in Embedded Systems Einsatz und bilden die Grundlage vieler technischen Geräte. Der Übergang zwischen dem Begriff Mikroprozessor und Mikrocontroller kann nicht klar gezogen werden, da viele

Mikroprozessoren bzw. CPUs immer mehr Komponenten integrieren wie z.B. Peripheriecontroller zum Ansteuern des Hauptspeichers.

Viele Mobilgeräte verwenden mittlerweile *SoCs (System on a Chip)*, die neben Prozessor und Peripheriefunktionen weitere Komponenten wie Speicher, Grafik, Audio und mehr auf einen Chip enthalten, um Kosten und Platz einzusparen.

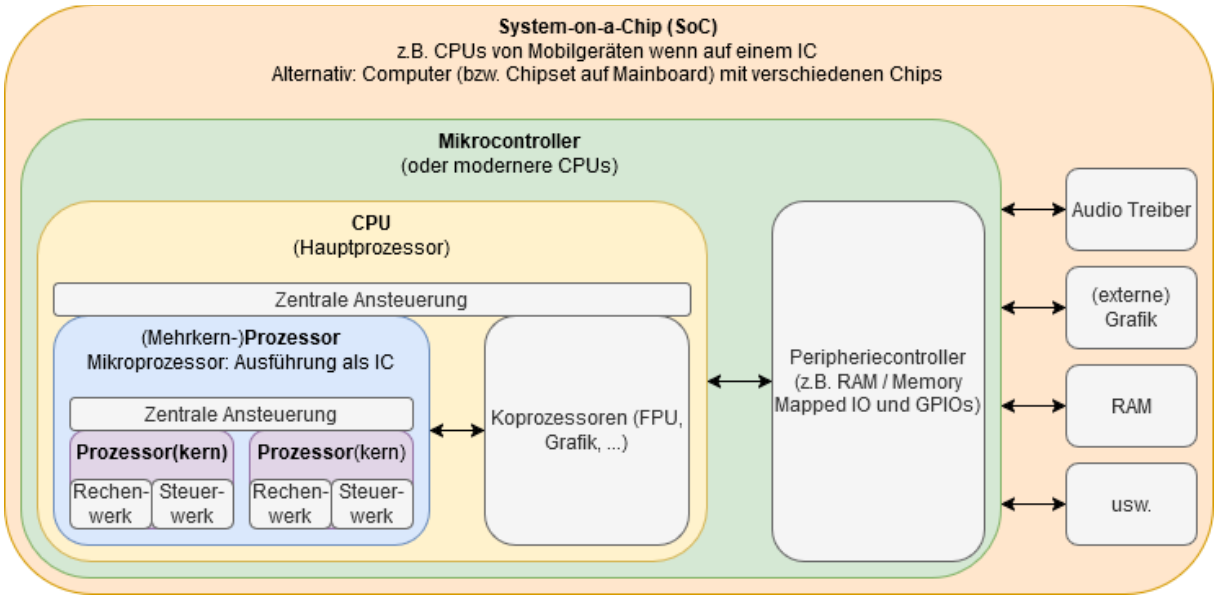


Abbildung 1: Begriffsabgrenzungen von Prozessoren nach [1, S. 2]

Alle Begriffe haben als gemeinsame Grundlage den Prozessor bzw. Prozessorkern, dessen Grundlagen im Folgenden betrachtet werden. Die Begriffe Prozessor, Mikroprozessor und CPU werden dabei synonym verwendet. Abbildung 1 verdeutlicht die Begrifflichkeiten.

2.1 Ebenen des Prozessormodells

Um das Gesamtsystem eines Prozessors genau zu definieren und zu dokumentieren, werden von [1, S. 17] zwei wichtige Begriffe genannt:

Die *Prozessorarchitektur* definiere die Grenze zwischen Hardware und Software und umfasse alle für Systemprogrammierer*innen und den Compiler wichtigen Komponenten und Informationen über das System. Synonym dazu sei der Begriff *Befehlssatzarchitektur* (eng.: Instruction Set Architecture, *ISA*), der im Folgenden verwendet wird. Die ISA umfasse unter anderem den Befehlssatz (d.h. die verfügbaren Befehle), das Befehlsformat (d.h. Codierung der Befehle in Maschinencode), die Adressierungsarten, das Interruptsystem und das Speichermodell (d.h. Register und Adressraumaufbau).

Die *Mikroarchitektur* (engl.: microarchitecture) bezeichne die genaue Implementierung einer bestimmten ISA auf der Hardwareseite. Zur Mikroarchitektur gehörten nach [1, S. 17] unter anderem die Hardwarestruktur der logischen Schaltungen, der Verlauf der Kontroll- und Datenpfade, die Art der Befehlsabarbeitung, die genauen zeitlichen Abläufe und die Organisation von internen Register- bzw. Speicherelementen. Das Wissen über die genaue Implementierung der Mikroarchitektur sei auf der Anwendungsseite nicht zwingend notwendig. Lediglich für eine starke Optimierung von Programmcode durch einen Compiler bzw. Systemprogrammierer*innen sei genaueres Wissen zur Mikroarchitektur notwendig.

Eine einzige ISA kann nach dieser Definition unterschiedliche Implementierungen durch verschiedene Mikroarchitekturen besitzen. Programmcode, der entsprechend dieser ISA geschrieben wurde, ist auf beiden Systemen ausführbar. Dieses Konzept wird auch in der Praxis angewandt und bringt u.a. folgende Vorteile mit sich: Auf der einen Seite wird eine Abwärtskompatibilität ermöglicht, wenn sich die Hardwaretechnologie eines Prozessors ändert und dieser beispielsweise besser und schneller wird. So sind Programme weiterhin ausführbar, wenn sich aus der Sicht des Programmes nichts an der ISA ändert. Auf der anderen Seite kann eine Hersteller- bzw. Hardwareunabhängigkeit realisiert werden. Wird Software für eine bestimmte ISA geschrieben, so kann diese auf jedem Prozessor, der diese ISA realisiert, ausgeführt werden und muss nicht auf viele verschiedene Systeme portiert werden. Ein Beispiel hierfür ist die quelloffene RISC-V ISA, die von unterschiedlichen CPU-Herstellern verwendet wird und eine einheitliche Richtlinie für eine ISA vorgibt. Es gibt verschiedene Implementierungen von RISC-V-Prozessoren, die sich stark unterscheiden können, jedoch alle RISC-V konform sind und entsprechend codierte Programme ausführen können. Auf die RISC-V ISA wird in dieser Arbeit stellenweise Bezug genommen.

Da das Ziel dieser Arbeit den Entwurf eines funktionsfähigen Prozessors umfasst, zählt neben der Entwicklung einer einfachen ISA auch das Design einer entsprechenden Mikroarchitektur zur Implementierung der ISA dazu. Die Mikroarchitektur muss mindestens so umfangreich sein, als dass eine Realisierung durch eine Hardwarebeschreibungssprache möglich ist. Eine genaue Implementierung durch einzelne Logikgatter ist somit nicht notwendig, da diese von dem HDL-Compiler durchgeführt wird. Zu den grundlegenden strukturellen Komponenten der

Mikroarchitektur zählen z.B. Register, Addier- bzw. Rechenwerke, Auswahlschaltungen und Schaltnetze, die als gegeben betrachtet werden können.

2.2 Grundlegender Aufbau der Mikroarchitektur

Die Entwicklung einer ISA und einer entsprechenden Mikroarchitektur sind generell aneinandergekoppelt. Unterschiedliche Designansätze der Mikroarchitektur ermöglichen unterschiedliche Befehle oder Befehlsgruppen; bestimmte Befehle, die die ISA enthalten muss, erfordern wiederum von der Mikroarchitektur das Vorhandensein bestimmter Hardwarestrukturen. Zudem bestimmen unterschiedliche Designansätze maßgeblich die Performance des Prozessors oder die Adressierung von Operanden bei Befehlen der ISA. Solche Designansätze werden in „2.3 Vergleich unterschiedlicher Designphilosophien“ verglichen. Jedoch existieren einige Kernkomponenten einer CPU, die für ihre Grundfunktionen notwendig sind und im Allgemeinen bei jedem Prozessordesign vorzufinden sind. [2, S. 172f] zählt die folgenden Hardwarekomponenten auf, die außerdem in Abbildung 3 dargestellt sind:

- *Arithmetisch Logische Einheit (engl.: arithmetic logic unit, ALU):*

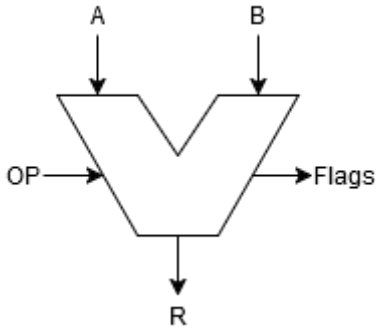


Abbildung 2: Schaltsymbol einer ALU mit Ein- und Ausgängen

Die ALU diene dem Ausführen einfacher arithmetischer und logischer Operationen für bis zu zwei gegebene Operanden. Abbildung 2 zeigt das typische Schaltsymbol einer ALU mit den entsprechenden Ein- und Ausgängen. A und B bezeichnen die beiden Operanden, auf die die Operation von der ALU ausgeführt werden soll. Über den Eingang OP wird die entsprechende Operation angewählt. Das Ergebnis wird über den Ausgang R ausgegeben. Ein weiterer Ausgang dient der Ausgabe von sogenannten Statusbits (z.B. ein Überlauf bei Additionen). Genaue Details zu den Statusbits, den verfügbaren Operationen und zu Datenbreite der Operanden sind erst bei einer genauen Implementierung der Mikroarchitektur bzw. nach Definition über die ISA bestimmbar. Typische Operationen, die von der ALU ausgeführt werden können, sind:

  - Logische Verknüpfungen (`and`, `or`, `xor`, `not`)
  - Schiebeoperationen (Bits nach links/rechts verschieben)

- Arithmetische Operationen (Addition, Subtraktion, optional: ganzzahlige Multiplikation und Division)

Die ALU dient primär der Ausführung von Ganzzahloperationen und wird z.B. bei Gleitkommazahloperationen von einem Coprozessor unterstützt.

- *Register:*

  [2, S. 172] beschreibt die Register eines Prozessors als kleine (Zwischen-) Speicherzellen, auf die schnell zugegriffen werden könne. Aufgrund dieser Geschwindigkeit dienten sie primär dem Zwischenspeichern von Operanden und Ergebnissen der ALU, die von dort aus in den Datenspeicher geladen werden können. Es würde unterschieden werden zwischen den *Allzweckregistern* (engl.: general purpose registers, *GPR*) und weiteren *Spezialregistern*. GPRs dienten primär dem oben genanntem Zweck und seien frei verfügbar zur Verwendung. Ihr Inhalt bzw. dessen Bedeutung werde durch den Compiler bzw. den*die Systemprogrammierer*in zugewiesen. In einem Prozessor existierten typischerweise mehrere Register, die gemeinsam zu einem *Registersatz* (engl.: register file) gruppiert würden und dadurch gemeinsam angesteuert werden könnten.

  Spezialregister enthielten immer Daten, die nur einem bestimmten Verwendungszweck dienten, der durch das Prozessordesign bestimmt ist. Der Zugriff auf die Spezialregister sei teilweise nur eingeschränkt möglich (z.B. nur lesen auf Statusbits) oder gar nicht möglich (z.B. auf interne Register, die nur vom Prozessor selbst verwendet werden) Zu den wichtigsten Spezialregistern gehörten:

  - *Befehlszähler* (engl.: program counter, *PC*): Adresse des nächsten Befehls im (Programm-)Speicher
  - *Befehlsregister* (engl.: instruction register, *IR*): Aktuell vom Prozessor auszuführender Befehl)
  - *Adressregister* (memory address register, *MAR*): Adresse zum Lesen von/Schreiben in Hauptspeicher

  Weitere Spezialregister seien abhängig von der ISA bzw. der Mikroarchitektur.

- *Hauptspeicher:*

  Der Hauptspeicher diene (im Gegensatz zu den Registern) der längerfristigen Speicherung von Daten. Während die Zugriffszeiten langsamer sind, ist die Speichergröße deutlich größer. [2, S. 173] weist deutlich darauf hin, dass der

eigentliche Speicher selbst keine Komponente sei, die sich im Prozessor befinde. Jedoch sei der Speicher notwendig für die Funktion des Prozessors. Außerdem sei die Schnittstelle zum Speicher ein wichtiger Planungsbestandteil der Mikroarchitektur. Entweder bestehe die Schnittstelle aus einem einfachen Datenbusinterface oder die Ansteuerung des Speichers (ein sog. Memory Controller) sei direkt im Prozessor integriert. Der Hauptspeicher bestehe aus einzelnen Zellen, die je einen Wert mit der Datenbreite des Prozessors speichern könnten und über eine Adresse eindeutig identifiziert werden könnten.

- *Datenbusse:*
  Datenbusse dienten der Verbindung verschiedener Komponenten der CPU zum Datenaustausch. Abhängig von der Mikroarchitektur und der ISA seien unterschiedliche Bustypen erforderlich und die Anzahl an Komponenten pro Bus schwanke.
  Beispielsweise könne ein Datenbus die ALU-Operanden und das Ergebnis zwischen ALU und Registersatz transferieren, ein weiterer Bus im gleichen Prozessor könne dem Datenaustausch zwischen CPU und Hauptspeicher ermöglichen, indem Daten und Adressen ausgetauscht werden.
- *Ein-/Ausgabe-Peripherie:*
  Um verschiedene externe Geräte an einen Prozessor anbinden zu können und einen Datenaustausch zu ermöglichen, sei ein extra I/O-Controller notwendig. Dieser könne unterschiedliche Speicherregionen der CPU einem E/A-Gerät zuweisen, anstatt einer tatsächlichen Speicherregion im Hauptspeicher.
- *Steuerwerk:*
  Das Steuerwerk diene der zentralen Steuerung aller Komponenten des Prozessors. Zur Realisierung einer solchen Steuerung sei das Steuerwerk mit allen Komponenten über sog. Steuersignale verbunden (z.B. OP der ALU zur Anwahl einer bestimmten Operation). Abhängig vom aktuell auszuführenden Befehl und bestimmten Zuständen (Statusbits) des Prozessors werden die Steuersignale gesetzt, um so die Abläufe zu steuern und einen bestimmen Befehl auszuführen.

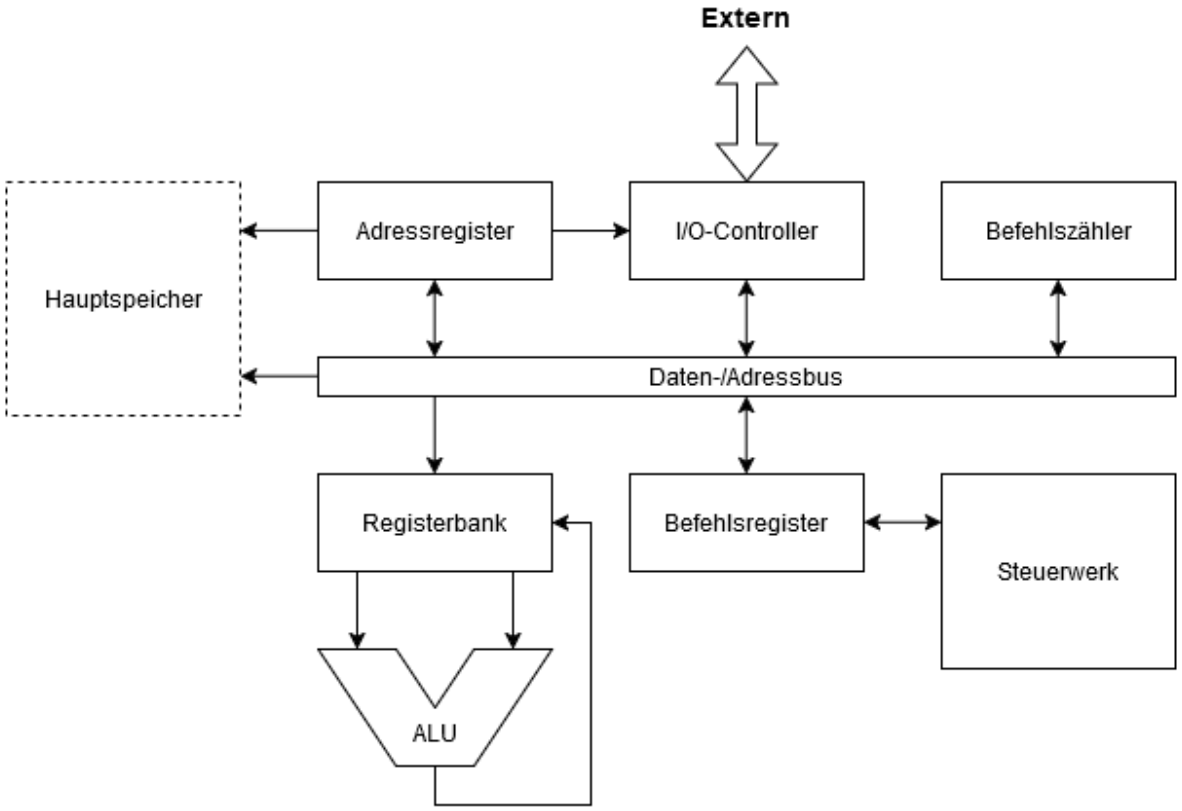


Abbildung 3: Beispielhafter Aufbau eines einfachen Mikroprozessors mit sequenzieller Ablaufstruktur und einem Hauptdatenbus zur Veranschaulichung der Kernkomponenten eines Prozessors nach [1, S. 44]

Abbildung 3 verdeutlicht den Aufbau eines einfachen Prozessors, der aus den genannten Kernkomponenten besteht. Dort könnte die beispielhafte Abarbeitung eines Befehls folgendermaßen aussehen:

1. Befehl laden (*Fetch*):
    a. Adresse des nächsten Befehls aus dem Befehlszähler in das Adressregister kopieren.
    b. Befehl aus Hauptspeicher auslesen und in Befehlsregister übernehmen.
2. Befehl dekodieren (*Decode*):
    c. Eventuell im Befehl kodierte Operanden dekodieren und befehlsabhängige Steuersignale im Steuerwerk generieren.
3. Befehl ausführen (*Execute*):
    d. Ausgewählte Operanden aus dem Registersatz an die ALU weitergeben.
    e. Entsprechende Operation (z.B. Addieren) durchführen.
4. Ergebnisse speichern (*Writeback*):
    f. Ergebnis der ALU wiederum in dem Registersatz speichern.

Der Zyklus *Fetch → Decode → Execute → Writeback* findet sich im Allgemeinen in jedem Prozessor wieder und zeigt seine grundlegende Arbeitsweise auf. Abhängig von der Mikroarchitektur können einzelne Teile dieses Zyklus in einem Takt geschehen (z.B. Decode und Execute) oder parallelisiert werden (*Pipelining*, s. „3.1 Grundlegende Festlegungen zur Mikroarchitektur“).

2.3 Vergleich unterschiedlicher Designphilosophien

Zur Implementierung einer Mikroarchitektur existieren grundlegende Designansätze bzw. Designphilosophien, die maßgeblich die Arbeitsweise, Performance und Komplexität des Prozessors bestimmen. Während in der Theorie eine strikte Trennung dieser Designansätze vorgenommen werde, würden praktische Implementierungen häufig Mischformen solcher Designs beinhalten, wie [1, S. 44] hervorhebt.

RISC vs. CISC:

[2, S. 199] beschreibt, dass die Vor- und Nachteile unterschiedlicher Designphilosophien bzgl. der genannten Punkte vor allem eine primäre Auswirkung auf die ISA haben: Es sei ein Abwiegen zwischen der Einfachheit des Designs des Prozessors (d.h. primär Mikroarchitektur) und der daraus resultierenden Komplexität im Programm (d.h. verfügbare Befehle der ISA). Entsprechend hätten sich zwei grundlegende Unterscheidungen von Prozessoren herauskristallisiert, die jeweils ein Extrem dieser Abwägung darstellen würden:

1. *Reduced Instruction Set Computer (RISC):*
   Ein RISC-Prozessor sei nach [2, S. 199] darauf ausgelegt, eine ISA mit einfachen Operationen und Befehlen bereitzustellen. Das resultiere auf der einen Seite in einem umfangreicheren Programmcode bestehend aus mehreren kleinen Befehlen, ermögliche aber auf der anderen Seite eine schnelle Abarbeitung einzelner Befehle mithilfe einer einfach implementierbaren Mikroarchitektur.
2. *Complex Instruction Set Computer (CISC):*
   Ein CISC-Prozessor sei im Gegenzug darauf ausgelegt, eine ISA mit komplexen Befehlen bereitzustellen (z.B. ALU-Operationen direkt auf Speicherinhalte anwenden oder ein Befehl zur Wurzelberechnung, der bei RISC eine Implementierung mithilfe eines Algorithmus erfordert). Das hätte entsprechend

einen kürzeren Programmcode, aber eine deutlich komplexere Mikroarchitektur zur Folge, wie [2, S. 200] weiter ausführt.

RISC-Designs seien laut [2, S. 200] in der Praxis mehr verbreitet und besäßen, ergänzt um Punkte von [1, S. 35], folgende primäre RISC-Merkmale:

- Befehlssatz mit wenigen, einfachen Befehlen
- Alle Befehle haben die gleiche Wortbreite und lassen sich in wenige Kategorien bzgl. der Kodierung von Operanden einteilen
    - ⇨ Resultiert ein einer einfachen Dekodierung und einem einfacheren Steuerwerk
- Ein Befehl kann (meistens) in einem Takt abgearbeitet werden
- Es werden viele GPRs bereitgestellt, um Daten Zwischenzuspeichern
- Speicherzugriff ist nur durch Lade-/Speicherbefehle möglich. Andere Befehle (z.B. ALU-Befehle) können nur auf Daten aus den GPRs zugreifen

CISC-Designs haben dementsprechend gegensätzliche Merkmale zu einem RISC-Design. Die entsprechenden Vor- und Nachteile ergeben sich aus diesen Merkmalen und müssen je nach Anforderung an den Prozessor entsprechend gewichtet werden.

Harvard vs. Von-Neumann

Ein weiterer Designansatz bezüglich der Mikroarchitektur betrifft die Organisation von Daten- und Programmspeicher, d.h. den Speicherorten, an denen das Programm (⇒ *Programmspeicher*) und Daten (⇒ *Datenspeicher*) gespeichert werden. [3, S. 86] unterscheidet in diesem Fall zwischen der *Harvard-Architektur* und der *Von-Neumann-Architektur*. Bei der Harvard-Architektur seien Programm- und Datenspeicher in zwei getrennten Speicherbereichen realisiert. Im klassischen Sinne sei dies eine Realisierung mittels zweier physisch getrennter Speicherbausteine, im moderneren Sinne könne dies auch ein Speicherbaustein mit getrennten Adressräumen für Daten und Programm sein. Die Von-Neumann-Architektur sieht hingegen einen gemeinsamen Speicherbereich sowohl für das Programm als auch für die Daten vor.

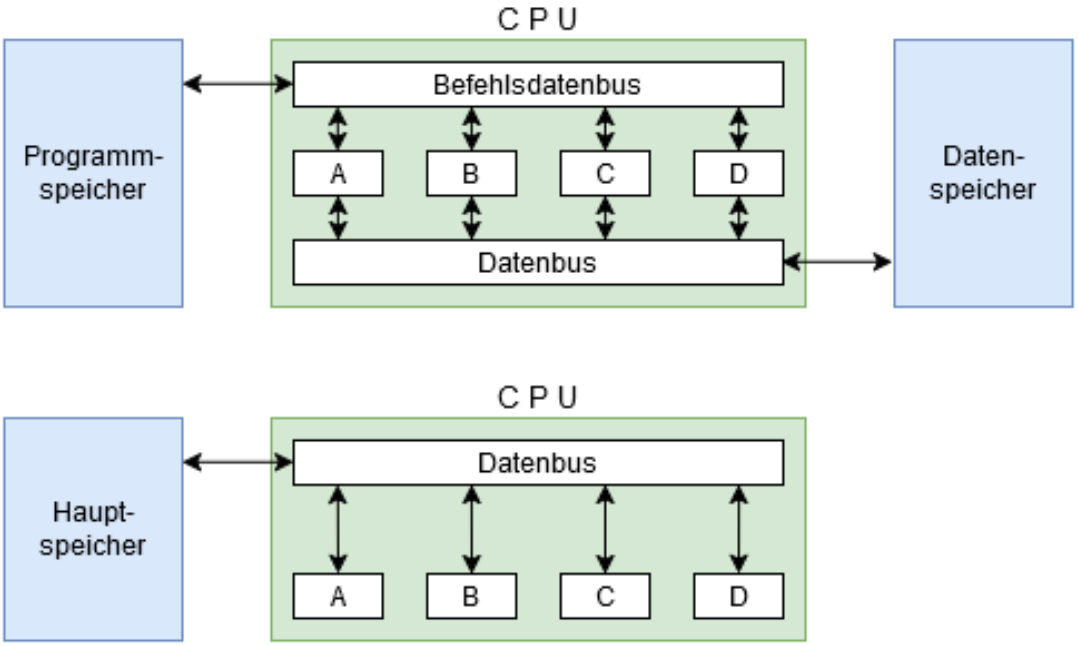


Abbildung 4: Unterschied in der Grundstruktur eines Prozessors nach Harvard-Architektur (oben) und Von-Neumann-Architektur (unten)

Abbildung 4 zeigt die grundlegenden strukturellen Unterschiede zwischen Von-Neumann-Prozessor und Harvard-Prozessor auf. Daraus lassen sich auch die Vorteile der einzelnen Architekturen ableiten. Ein Von-Neumann-Prozessor besitzt einen einfacheren Aufbau, da lediglich ein gemeinsamer Datenbus für Befehle und Daten notwendig ist. Außerdem ermöglicht ein gemeinsamer Hauptspeicher eine flexible Aufteilung zwischen Daten und Programm, da ein einzelner Speicherbaustein nicht strikt begrenzt ist.

Ein Harvard-Prozessor ermöglicht den gleichzeitigen Zugriff auf Programm und Daten, was eine Grundvoraussetzung für die im folgenden Abschnitt beschriebene Pipeline-Verarbeitung ist, und dadurch eine weitaus schnellere Verarbeitungsgeschwindigkeit des Prozessors ermöglicht. Weiterhin ist eine unterschiedliche Wortbreite für Befehle und Daten möglich. Das ist besonders bei einfacheren Mikrocontrollern von Vorteil, wenn die Datenbreite gering ist, um die Recheneinheit kompakt zu halten, jedoch eine größere Wortbreite für Befehle gewünscht ist, um diese sinnvoll kodieren zu können.

[3, S. 86] weist darauf hin, dass die Von-Neumann-Architektur eher bei kostengünstigen Prozessoren zum Einsatz käme aufgrund der weniger komplexen Implementierung und der Notwendigkeit von lediglich einem Speicherbaustein. Moderne, leistungsfähige CPUs orientierten sich vermehrt an der Harvard-Architektur

aufgrund der Geschwindigkeitsvorteile, die sich bei gleichzeitigem Zugriff auf Programm- und Datenspeicher ergäben.

Pipelining:

Im einfachsten Fall arbeiten Prozessoren streng sequenziell. Das bedeutet, dass die Ausführung des nächsten Befehls erst dann beginnt, wenn der vorherige Befehl vollständig abgearbeitet wurde, d.h. der Fetch, Decode, Execute, Writeback-Zyklus ein ganzes Mal durchlaufen wurde. Unter der Annahme, dass für die einzelnen Ausführungsschritte unterschiedliche Komponenten im Prozessor verwendet würden (ALU für Execute, Registersatz für Writeback, usw.), könne mithilfe der sogenannten *Fließbandverarbeitung* (engl. *Pipelining*) eine Steigerung der Taktfrequenz und damit eine Erhöhung des Durchsatzes der Befehlsausführung erreicht werden, wie [3, S. 89] beschreibt. Die Phasen können parallel (für jeweils unterschiedliche Befehle) ablaufen, da die beteiligten Komponenten parallel im Prozessor arbeiten können. Ein Prozessor, der mithilfe von Pipelining einen Befehl abarbeitet, besitzt entsprechend eine veränderte Grundstruktur, die in Abbildung 5 dargestellt ist.

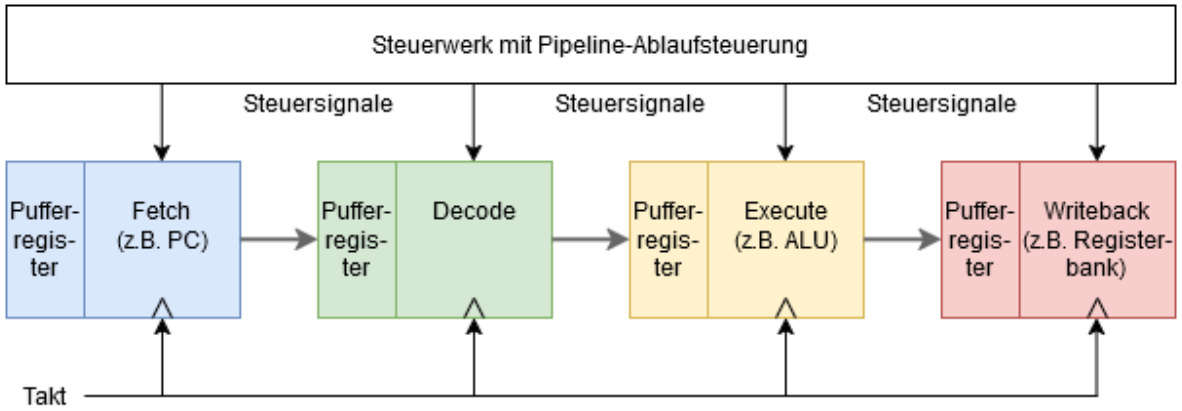


Abbildung 5: Struktur der Mikroarchitektur eines Prozessors mit vierstufiger Pipeline nach [1, S. 47]

Abbildung 5 zeigt, dass die einzelnen Komponenten des Prozessors nach ihrer Zugehörigkeit zu der jeweiligen Ausführungsphase (Fetch, Decode, Execute, Writeback) gruppiert werden. Ein Befehl wird durch die einzelnen Phasen bzw. Stufen der Pipeline hindurchgereicht und ist am Ende der Pipeline vollständig ausgeführt. Der Übergang bzw. das Weiterreichen eines Befehls von einem in die nächste Stufe findet immer zu jeder positiven oder negativen Taktflanke des Prozessortaktes statt, sodass der Befehl genau einen Takt in jeder Stufe verweilt. Damit dies möglich ist, muss jede Stufe der Pipeline mit Pipeline-Registern ausgestattet werden, die die relevanten

Daten zur Ausführung des Befehls auf der jeweiligen Stufe benötigen. Fetch benötigt beispielsweise den PC-Wert, Decode speichert den geladenen Befehl, Execute speichert den dekodierten Befehl mitsamt Operanden und Writeback benötigt das zu speichernde Ergebnis der Execute-Phase.

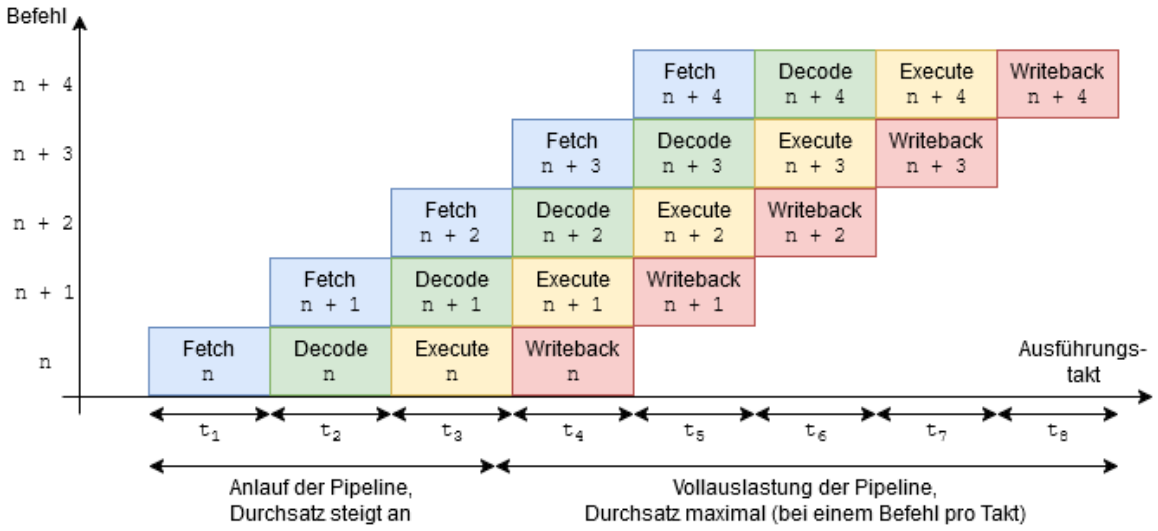


Abbildung 6: Zeitlicher Ablauf bei der Abarbeitung von Befehlen der vierstufigen Pipeline aus Abbildung 5

Der zeitliche Vorteil eine Fließbandverarbeitung ergibt sich bei der Betrachtung eines Zeitdiagrammes der Pipeline wie es in Abbildung 6 dargestellt ist. Da jede Komponente nur einer Stufe zugehörig ist, können alle Stufen parallel jeweils einen unterschiedlichen Befehl abarbeiten. Das bedeutet, dass nach dem Laden des Befehls n im Takt t_1 im nächsten Takt t_2 dessen Dekodierung stattfindet. Parallel dazu befände sich die Fetch-Stufe im Leerlauf und kann stattdessen im gleichen Takt den Befehl $n + 1$ laden. Geht in t_3 der Befehl n in die Execution-Phase, kann $n + 1$ dekodiert und gleichzeitig $n + 2$ geladen werden. Das resultiert nach dem Anlauf der Pipeline (d.h. nach Beenden des dritten Taktes) in einer vollständigen Auslastung der Pipeline und daher in einem fertig ausgeführten Befehl pro Takt (im Gegensatz zu einem Befehl alle vier Takte bei streng sequenzieller Abarbeitung).

Der Umfang der Pipeline ist stark abhängig von den Zielen beim Prozessorentwurf. Eine Erhöhung der Anzahl an Pipeline-Phasen (z.B. durch eine feinere Aufteilung der Abarbeitung in mehrere, aber einfachere Schritte) ermögliche nach [1, S. 48] eine weitaus schnellere Pipeline-Taktung aufgrund der geringeren Signallaufzeiten eines Signales durch das Schaltnetz der jeweiligen Pipelinestufe. Der Nachteil sei jedoch die weitaus komplexere Verwaltung der Pipeline und das Auftreten von sogenannten

Pipeline-Konflikten (engl. *Pipeline Hazards*). Auf solche Pipeline-Konflikte wird an dieser Stelle nicht näher eingegangen, da der Prozessorentwurf auf eine einfache Pipelinestruktur setzt, die solche Konflikte vermeidet. Generell sorgen Pipeline-Konflikte für Latenzzeiten in der Programmabarbeitung, da auf Daten von vorherigen Befehlen gewartet werden muss (Datenflusskonflikte) oder falsche Befehle nach bedingten Sprungbefehlen geladen wurden (Kontrollflusskonflikte). Um solche Konflikte zu lösen bzw. zu vermeiden, existieren einige Konzepte, die die Zielsetzung dieser Arbeit, einen einfachen Prozessor zu entwickeln, deutlich überschreiten.

Auch eine Verringerung der Pipeline-Stufen ist möglich, indem beispielsweise mehrere Phasen, die hintereinandergeschaltet sind und jeweils aus einem Schaltnetz bestehen, zusammengefasst werden. So wird auf der einen Seite die Komplexität der Pipelinesteuerung verringert, auf der anderen Seite jedoch die Taktzeit erhöht, da höhere Signallaufzeiten die einzelnen Stufen verlangsamen. Ein Pipelinedesign eines Prozessors ist daher von vielen Faktoren abhängig, die bei sehr komplexen und leistungsstarken Prozessordesigns beachtet werden müssen.

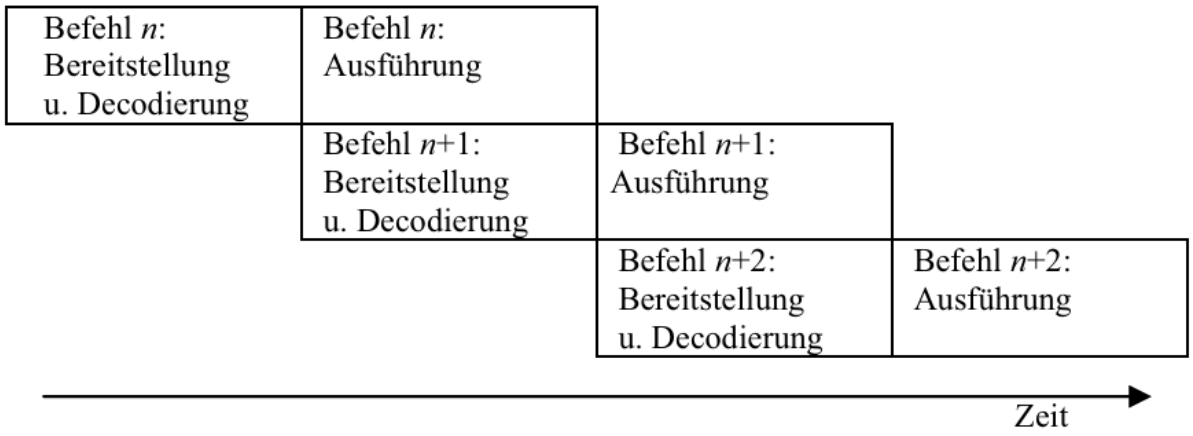


Abbildung 7: Einfache Pipeline mit zwei Stufen aus [1, S. 46]

2.4 Typische Befehle eines Befehlssatzes

Nachdem in den vorangehenden Kapiteln die grundlegende Struktur der Mikroarchitektur von Prozessoren und verschiedene Designansätze vorgestellt wurden, geht dieser Abschnitt auf die typischen Befehle eines Befehlssatzes ein. Die Grundlagen zur Mikroarchitektur sind notwendig, da verschiedene Befehle nicht nur einfache Berechnungen ausführen können, sondern auch Auswirkungen auf andere Komponenten des Prozessors haben. Entsprechend ihrer Wirkungsweise im Prozessor bzw. der Operation des Befehls zählt [1, S. 23-26] verschiedene

Befehlsgruppen auf, in die sich die verschiedenen Befehle der ISA eines Prozessors gruppieren lassen und die im Allgemeinen notwendig für einen einfachen Befehlssatz sind:

- *Datenbewegungsbefehle (data movement):*
  Dienten dem Verschieben von Daten zwischen den verschiedenen Speicherbereichen im Prozessor, wie z.B. Register ⇔ Datenspeicher.
- *Arithmetisch-logische Befehle (Integer arithmetic and logic):*
  Im Folgenden auch als ALU-Befehle bezeichnet. Alle Befehle, die mithilfe der ALU arithmetische und logische Operationen auf Ganzzahloperanden ausführten (z.B. Addition, UND-Verknüpfung, …), zählten in diese Kategorie.
- *Schiebe- und Rotationsbefehle (shift and rotate):*
  Seien tendenziell auch ALU-Befehle. Solche Befehle dienten dem Anwenden von logischen (oder arithmetischen) Schiebe- und Rotationsoperationen auf einen Operanden mithilfe von entsprechenden Schaltnetzen/-werken, die auch in der ALU integriert sein könnten (z.B. logisch nach links schieben um eine Bitstelle).
- *Programmsteuerbefehle (control transfer instructions):*
  Dazu zählten Befehle zum Beeinflussen des Programmablaufes, z.B. bedingte und unbedingte Sprungbefehle und Unterprogrammaufrufe. Solche Befehle könnten die Reihenfolge der Abarbeitung der Befehle im Programmspeicher verändern und dadurch komplexe Programmlogik ermöglichen.
- *Systemsteuerbefehle (system control instructions):*
  Systemsteuerbefehle würden direkten Einfluss auf die einzelnen Prozessorhardwarekomponenten nehmen, z.B. auf den I/O-Controller zur Konfiguration der Memory Mapped I/O-Geräte oder auf den Power Mode des Prozessors für verschiedene Ruhemodi. Auch ein sog. Halt-Befehl zum Anhalten der Befehlsausführung des Prozessors gehöre dazu.

Innerhalb der einzelnen Befehlsgruppen existieren teilweise weiterere Untergruppen oder Befehle mit verschiedenen Auswirkungen. Die Datenbewegungsbefehlen lassen sich beispielsweise weiter aufteilen in die folgenden Kategorien (wobei jede Kategorie implementierungsabhängig auch nur durch einen Befehl realisiert sein kann):

1. Register ⇔ Register Transfer (*move-Befehle*)
   Solche Befehle verschieben Daten zwischen den einzelnen Registern des Registersatzes.
2. Register ⇔ Datenspeicher Transfer (*load-/store-Befehle*)
   In RISC-Architekturen können ALU-Befehle nur auf Registerdaten und nicht auf Daten aus dem Speicher zugreifen. Entsprechend sind Befehle zum Datentransfer zwischen des Registersatzes und dem Datenspeicher notwendig. Load-Befehle, d.h. Ladebefehle, laden Werte aus dem Speicher in ein Register; Store-Befehle, d.h. Speicherbefehle, speichern einen Wert aus einem Register im Speicher.
3. *Immediate Befehle* (load immediate: Daten aus Befehl ⇒ Register)
   Damit ein Prozessor mit Anfangswerten für Berechnungen versorgt werden kann, sind sog. Immediate-Befehle notwendig, die einen im Befehl kodierten Wert in ein Register laden.

Der Umfang an arithmetisch-logischen Befehlen hängt vom Umfang der ALU ab. Für jede Operation, die die ALU ausführen kann, existiert in der ISA meistens ein Befehl, um diese Operation anzuwählen. Alternativ kann eine ausgewählte Operation im Befehl kodiert werden. ALU-Befehle benötigen im Allgemeinen bis zu drei zusätzliche Angaben, d.h. Operanden: Zwei Quellenangaben zu den beiden Operanden und eine Zielangabe, wo das Ergebnis gespeichert werden soll. Das nächste Kapitel „2.5 Adressierungsmöglichkeiten im Befehlssatz“ diskutiert unterschiedliche Ansätze, diese Angaben im Befehl zu kodieren und sie in der Mikroarchitektur zu implementieren.

Programmsteuerbefehle unterscheiden sich prinzipiell in zwei Merkmalen:

1. *Bedingt* oder *unbedingt*:
   Die Verzweigung bzw. der Sprung im Programm wird entweder *immer* ausgeführt (*unbedingt*) oder nur dann ausgeführt, wenn eine bestimmte *Bedingung erfüllt* ist (*bedingt*). Solche Bedingungen sind je nach ISA unterschiedlich implementiert und können beispielsweise über die Flags der ALU realisiert werden (s. [4, S. 21]). Bedingte Sprungbefehle ermöglichen sog. If/else-Ausdrücke in Hochsprachen.
2. *Absolut* oder *relativ*:

Das Sprungziel des Befehls wird entweder als *absolute Adresse* im Programmspeicher angegeben, zu der direkt gesprungen wird (*absolut*) oder in Form eines *Offsets*, d.h. einem Wert, der zur aktuellen Adresse des Befehls addiert wird (*relativ*). Dieser Offset ist vorzeichenbehaftet, d.h. Rückwärtssprünge sind möglich, und benötigt weniger Bitstellen in der Kodierung des Befehls. Dafür ist die Sprungweite begrenzt.

Auch Befehle, die zu Unterprogrammen springen, zählen in die Kategorie der Programmsteuerbefehle. Diese sind unbedingte, meist absolute Sprünge, die zusätzlich weitere Schritte unternehmen, um nach Beenden des Unterprogrammes wieder an die richtige Stelle im Hauptprogramm zurückspringen zu können. Ein Unterprogramm (UP) ist ein Programmabschnitt, der getrennt vom Hauptprogramm im Programmspeicher abgelegt ist und einen an mehreren Stellen wiederverwendbaren Programmcode darstellt. Damit das UP unabhängig vom Zustand des Prozessors im Hauptprogramm arbeiten kann, muss der sog. *call-Befehl*, der ein UP aufruft, neben dem Sprung zum UP zusätzlich die *Rücksprungadresse* speichern. Die Rücksprungadresse ist die Adresse des nächsten Befehls im Hauptprogramm nach Beenden des Unterprogrammes und wird in einem nach dem Last-In-First-Out-Prinzip (LIFO) organisierten Speicher (der sog. Stack) abgelegt. Am Ende des Unterprogrammes wird über den *return-Befehl* diese Rücksprungadresse geladen und die Abarbeitung des Hauptprogrammes fortgesetzt. Im Kontext von Unterprogrammen sind zwei weitere Befehle, die zur Gruppe der Datenbewegungsbefehle gehören, notwendig. *Push* speichert den Inhalt von bestimmten Registern des Registersatzes auch auf dem Stack, wenn diese in dem Unterprogramm verändert werden sollten. Mittels *pull* werden die Werte der Register aus dem Hauptprogramm wiederhergestellt, sodass eine reibungslose Abarbeitung sichergestellt wird. Push und pull garantieren, dass keine Speicherzelle überschrieben wird, wenn ein UP mehrmals aufgerufen wird und ein Wert immer an die gleiche Adresse geschrieben würde. Daher sind diese Befehle anstelle eines Store-Befehls mit fester Adressangabe in UPs zum Sichern von Registerinhalten zu verwenden. Die Implementierung eines Stacks als LIFO-Speicher wird im nächsten Kapitel thematisiert. Abbildung 8 verdeutlicht die Abläufe bei dem Aufruf eines Unterprogrammes.

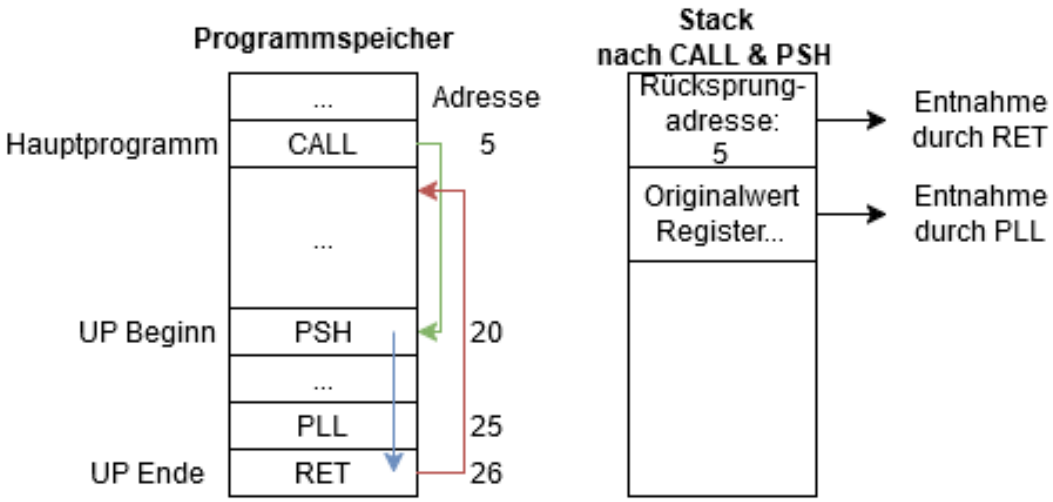


Abbildung 8: Abarbeitungsroutine von Unterprogrammen in einem Prozessor

Systemsteuerbefehle können direkten Einfluss auf die Hardwarekomponenten des Prozessors nehmen und ermöglichen dadurch die Konfiguration dieser Komponenten von extern. Realisiert werden sie, indem ein bestimmtes Datenwort als Steuerwort in ein Konfigurationsregister geschrieben wird. So können beispielsweise verschiedene Funktionen aktiviert oder deaktiviert werden, indem Bits in bestimmten Registern auf null oder eins gesetzt werden.

Die ISA beinhaltet auch die Kodierung der einzelnen Befehle, d.h. die Übersetzung eines Befehls in eine vom Prozessor lesbare Darstellung, den sogenannten Maschinencode. Dieser liegt dem Prozessor in binärer Darstellung vor, zwecks kürzerer Darstellung wird er oft hexadezimal angegeben. Wie genau ein Befehl kodiert werden soll, lege nach [1, S. 26] das sogenannte *Befehlsformat* der ISA fest. Ein kodierter Befehl bestehe aus zwei Teilen. Er beginne nach immer mit dem sog. Opcode des entsprechenden Befehls, einer Bitfolge, die den auszuführenden Befehl selbst eindeutig identifiziere. Danach folgten, basierend auf dem Opcode bzw. dem Befehl, unterschiedliche Operanden des Befehls (z.B. die Quell- und Zielangaben bei ALU-Befehlen). In einer Pseudoassemblersprache würde der Befehl `add r0, r1, r2`, bestehend aus der sog. Mnemonik `add` und den Operanden `r0, r1, r2` (Registerangaben), kodiert werden, indem die Mnemonik `add` in den entsprechenden Opcode laut ISA übersetzt wird und die Operanden nachfolgend codiert werden. Der genaue Aufbau eines Befehls ist abhängig von der ISA. Eine wichtige Variable bei der Erstellung einer ISA ist die Befehlsbreite, d.h. die Anzahl an Bits, in denen jeder Befehl kodiert wird. Sie kann z.B. von der Komplexität der Befehle, der notwendigen

Operanden, der Befehlsanzahl und der Adressbreite der Speicher abhängen. Eine solche beispielshafte Kodierung eines Befehls einer fiktiven ISA ist in Abbildung 9 dargestellt.

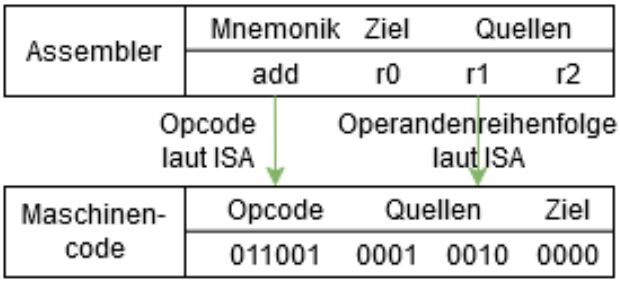


Abbildung 9: Beispielhafte Befehlskodierung des Befehls `add r0, r1, r2` in einer fiktiven ISA

2.5 Adressierungsmöglichkeiten im Befehlssatz

Zur Ausführung von ALU-Befehlen benötigt der Prozessor im Allgemeinen drei Angaben: Zwei Quellenangaben für die beiden Operanden der ALU und eine Zielangabe, wo das Ergebnis gespeichert werden soll. [1, S. 26] unterscheidet zwischen vier Klassen von Befehlssätzen, die ein entsprechendes Befehlsformat für ALU-Befehle vorgeben:

1. Das *Dreiadressformat* bestehe neben dem Opcode aus zwei Quelloperandenbezeichnern (`Src1`, `Src2`) und einem Zieloperandenbezeichner (`Dest`) zur Angabe aller notwendigen Informationen:

`Opcode`	`Dest`	`Src1`	`Src2`

2. Das *Zweiadressformat* bestehe neben dem Opcode aus einem Quelloperandenbezeichner und einem gemeinsamen Quell-/Zieloperandenbezeichner für den ersten Quelloperanden und den Zieloperanden:

`Opcode`	`Dest/Src1`	`Src2`

3. Das *Einadressformat* bestehe neben dem Opcode aus einem einzigen Quelloperandenbezeichner. Eine entsprechende Mikroarchitektur beinhalte immer das sogenannte *Akkumulatorregister*, das automatisch den ersten Quelloperanden beinhalte und in dem das Ergebnis gespeichert werde:

Opcode	Src2

4. Das *Nulladressformat* bestehe immer lediglich aus dem Opcode. Für die Implementierung einer solchen ISA sei eine besondere Mikroarchitektur notwendig, die sog. „*Kellerarchitektur*“, auf die an dieser Stelle nicht näher eingegangen wird:

Opcode

Bei einer RISC-Architektur (ALU-Befehle ohne Speicherzugriff) bezeichnen `Src1`, `Src2` und `Dest` ein Register des Registersatzes und werden mithilfe der Registernummer angegeben. Anstelle von `Src1`, `Src2` und `Dest` wird stattdessen von `Rs1`, `Rs2` und `Rd` gesprochen.

Die Auswahl einer der genannten Klassifikationen beeinflusst maßgeblich die Befehlsbreite der ISA und hänge nach [1, S. 27] eng mit unterschiedlichen Mikroarchitekturen zusammen. ISAs mit Null- oder Einadressformat benötigten eine besondere Mikroarchitektur, während ISAs mit Zwei- oder Dreiadressformat sich stark ähneln könnten und aktuellen Prozessorarchitekturen entsprächen.

Weitere Unterschiede in einzelnen Befehlsformaten existieren bezüglich der Adressierung von Speicherzellen im Datenspeicher des Prozessors. Zwei grundlegende Unterscheidungen dieser Adressierung werden von [1, S. 30] vorgenommen:

- Wird im Befehlswort die Adresse des Operanden aus dem Datenspeicher angegeben, spreche man von der absoluten oder *direkten Adressierung* (s. Abbildung 10).

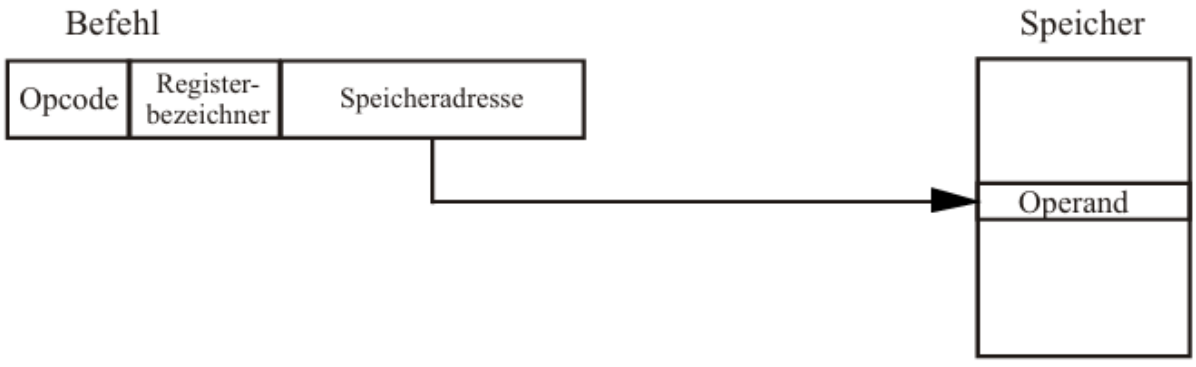


Abbildung 10: Funktionsweise der direkten Adressierung eines Speicheroperanden aus [1, S. 30]

- Wird im Befehlswort ein Registerbezeichner angegeben, stehe die Operandenadresse in diesem Register, d.h. das Register diene als Zeiger auf die Speicheradresse, Dies werde als *(register-)indirekte Adressierung* bezeichnet (s. Abbildung 11).

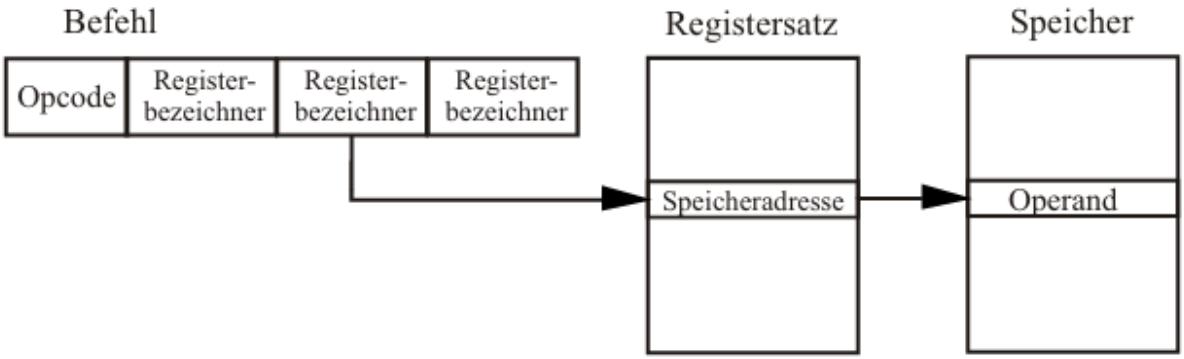


Abbildung 11: Funktionsweise der indirekten Adressierung eines Speicheroperanden aus [1, S. 30]

Weiterhin verweist [1, S. 31] auf einige Sonderformen der indirekten Adressierung. Zu diesen Spezialfällen würden unter anderem die folgenden Varianten zählen:

- Als registerindirekte Adressierung *mit Autoinkrement/Autodekrement* bezeichne man eine Variante, bei der die über ein Register angegebene Adresse vor oder nach dem Speicherzugriff (Prä-/Postinkrement/-dekrement) um den Wert Eins verringert oder erhöht werde. Diese Variante finde häufig Anwendung beim Zugriff auf Arrays mittels einer Schleife, indem der Registerinhalt anfangs auf den Anfang oder das Ende des Arrays zeige und bei jedem Zugriff auf ein Element automatisch erhöht oder verringert werde.
- Als registerindirekte Adressierung *mit Verschiebung* bezeichne man eine Variante, bei der die Speicheradresse errechnet werde aus der Summe aus dem Registerinhalt und einem im Befehl angegebenen Offset. Auch diese Variante finde Anwendung bei Operationen mit Arrays und ist in Abbildung 12 dargestellt.

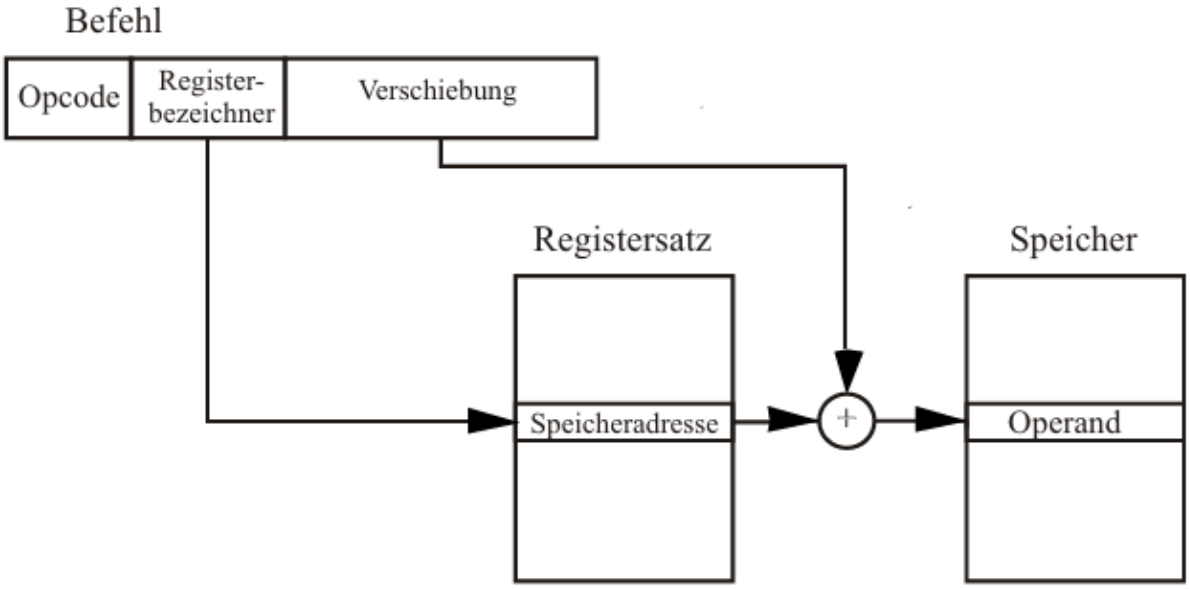


Abbildung 12: Funktionsweise der registerindirekten Adressierung mit Verschiebung aus [3, S. 31]

- Als *indizierte Adressierung* bezeichne man eine Variante, bei der die verwendete Speicheradresse errechnet werde aus der Summe aus dem angegebenen Registerinhalt und einem weiteren Register. Dieses weitere Register könne ein festes Register aus dem Registersatz sein oder ein spezielles Indexregister (Spezialregister). Mit dieser Variante ließen sich Tabellen und beliebig große Datenstrukturen einfach durchlaufen.

Stack-Speicher

Neben diesen adressierten Zugriffen auf den Hauptspeicher kann in einem Prozessor die Möglichkeit vorhanden sein, einen sogenannten *Stack-Speicher* (kurz *Stack*, dt.: *Kellerspeicher*) zu verwenden. Dieser sei nach [2, S. 401f] eine LIFO-Struktur, die vom Prozessor selbst mithilfe des sog. *Stackpointers (SP)* verwaltet werde. Der Stackpointer sei ein Register und entweder ein GPR als Teil des Registersatzes oder ein Spezialregister. Der aktuelle Wert des SPs verweise auf den obersten Stack-Eintrag, d.h. das letzte Element der LIFO-Struktur. Je nach Implementierung sei dies entweder der nächste freie oder der letzte belegte Stackeintrag. Nach dem Hinzufügen eines Elementes könne der SP inkrementiert oder dekrementiert werden, d.h. der Stack wächst entweder nach oben oder nach unten. Je nach Implementierung müsse der SP mit einer entsprechenden Startadresse für den Stack initialisiert werden. Zugriff auf den Stack ermöglichen die bereits erwähnten Befehle push (Wert auf Stack ablegen) und pop (Wert von Stack entnehmen). Abbildung 13 zeigt die Funktionsweise

eines Stack-Speichers, der nach unten wächst und dessen SP immer auf den nächsten freien Eintrag verweist.

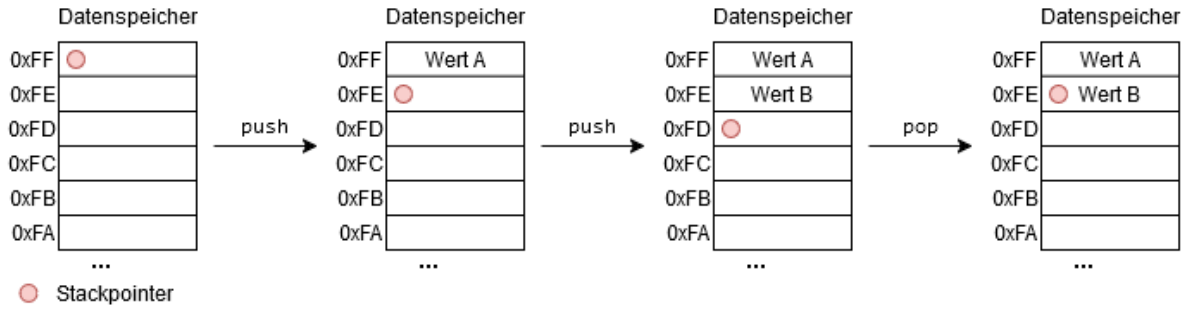


Abbildung 13: Funktionsweise eine Stack-Speichers. Der SP verweist auf den nächsten freien Eintrag und wächst nach unten

Neben einer schnellen Zwischenspeichermöglichkeit ohne Adressen beachten zu müssen mittels push und pop bietet der Stack-Speicher eine Möglichkeit zur Implementierung von Unterprogrammen mittels call und return (s. „2.4 Typische Befehle eines Befehlssatzes“) und zur Einbindung von Interrupts, die nach einem ähnlichen Prinzip wie call und return den Kontext des Hauptprogrammes wiederherstellen müssen. Somit ist der Stack-Speicher eine wichtige Speicherstruktur innerhalb eines Prozessors. Der Stack kann sich entweder im gleichen Adressraum und auf dem gleichen Chip wie der Hauptspeicher befinden oder (seltener) auf einem gesonderten Chip.

3. Entwurf des Prozessors

Nachdem in dem vorangehenden Kapitel wichtige Grundlagen und Konzepte zu ISAs und Mikroarchitekturen von Prozessoren diskutiert wurden, beschäftigt sich das folgende Kapitel mit dem Entwurf des Prozessors dieser Studienarbeit. Dabei sollen aus den vorgestellten Konzepten diejenigen ausgewählt werden, die zur Erfüllung der Zielsetzung beitragen, einen einfachen Prozessor zu Demonstrationszwecken zu entwerfen. Dabei werden zunächst einige grundlegende Festlegungen zur Mikroarchitektur getroffen, die zur Erfüllung der Zielsetzung notwendig sind und vor dem Entwurf der ISA definiert werden müssen. Anschließend wird eine ISA entworfen, die die Programmierschnittstelle des Prozessors definiert. Zuletzt wird auf Basis der ersten Festlegungen eine Mikroarchitektur entworfen, die die entwickelte ISA als Prozessors implementiert.

3.1 Grundlegende Festlegungen zur Mikroarchitektur

Die erste Festlegung bezüglich des Prozessors ist der Entwurf eines RISC-Designs. Die Verwendung einer RISC-Architektur resultiert daraus, dass die Argumente bzw. die Eigenschaften eines RISC-Prozessors deutlich besser zur Zielsetzung passen als die eines CISC-Prozessors. Ein RISC-Design ist simpler in der Implementierung der Mikroarchitektur, deutlich häufiger anzutreffen in praktischen Prozessordesigns und besitzt zudem weniger komplexe Befehlssets, die entsprechend kompakter kodiert werden können.

Im zweiten Schritt wird festgelegt, dass sich an der Harvard-Architektur bezüglich der Organisation von Programm- und Datenspeicher orientiert werden soll. Dies wird vor allem begründet mit dem Argument, dass unterschiedliche Wortbreiten für Befehle und Daten möglich sind. Somit kann für den einfachen Prozessor eine kleine Datenbreite verwendet werden, gleichzeitig wird die Kodierung der Befehle nicht eingeschränkt. Außerdem ermöglicht die Harvard-Architektur Pipelining zur Befehlsabarbeitung; ein Konzept, was dieser Prozessor definitiv unterstützen sollte. Pipelining ist ein Konzept, was in jedem modernen Prozessor angewandt wird. Trotz der etwas komplexeren Struktur sollte daher eine einfache Pipeline implementiert werden, um das Konzept in diesem Prozessor zu demonstrieren. Näheres zur Pipeline ist in „3.3 Entwurf der Mikroarchitektur“ zu finden, da eine Pipeline-Verarbeitung idealerweise nicht auf ISA-Ebene zu erkennen ist und dementsprechend keine Rolle in Bezug auf den Entwurf

der ISA spielt. Die Verwendung des Harvard-Prinzips resultiert in diesem einfachen Prozessor primär darin, wie die Befehls- und Datenbusse aufzubauen sind und welche Schnittstellen zu den Speicherbausteinen vorhanden sein müssen, da diese nicht Teil des Prozessors sind und dieser als einfache CPU auch keinen Cache besitzen wird.

Aus diesen Festlegungen folgt, dass der Prozessor zumindest alle Grundkomponenten einer CPU, wie sie in Abbildung 3 dargestellt sind, besitzt. Lediglich die Struktur der Anordnung wird sich aufgrund des Pipelinings unterscheiden. Allerdings wird ein I/O-Controller kein Bestandteil dieser Arbeit sein. Für die grundlegende Arbeitsweise eines Prozessors ist dieser nicht notwendig. Dank Memory Mapped I/O werden alle Befehle bzw. Mechanismen zum Zugriff auf Peripherie verfügbar sein und eine Implementierung eines I/O-Controllers kann Bestandteil einer weiteren Arbeit zur Erweiterung des Prozessors sein.

3.2 Entwurf des Befehlssatzes

Die folgenden Abbildungen sind Ausschnitte aus der fertigen ISA des Prozessors dieser Arbeit. Die gesamte ISA besteht aus drei Teilen (allgemeine Definitionen, Befehlsformate und Befehlsset) und ist im Anhang 1 (und Anhang 2) zu finden.

Registersatz

Die ISA definiert zuerst den Registersatz bestehend aus den GPRs. Eine RISC-Architektur verlangt eine ausreichend große Anzahl an GPRs zum Zwischenspeichern von Operanden und Ergebnissen, da keine Speicherzugriffe von anderen Befehlen außer Speicherzugriffsbefehlen möglich sind. Der Registersatz der RISC-V-Architektur bestehe nach [5, S. 9f] beispielsweise aus 32 Registern (31 GPRs und ein Nullregister mit dem konstanten Wert 0). Um Register zu adressieren, sind dafür fünf Bits notwendig. Ein einfacher RISC-Prozessor kann mit weniger GPRs auskommen und wodurch sich auch die Kodierung kompakter gestaltet. Daher fällt die Entscheidung für die Registerzahl dieses Designs auf acht. So werden drei Bits zum Kodieren perfekt ausgenutzt und es ist eine ausreichende Registeranzahl im Registersatz verfügbar.

GPRs	Kodierung	Beschreibung
r0	000	General Purpose Register
r1	001	General Purpose Register
r2	010	General Purpose Register
r3	011	General Purpose Register
r4	100	General Purpose Register
r5	101	General Purpose Register
r6	110	General Purpose Register. Aber: bei P=1 in CSR werden die 4 LSBs in PR geschrieben
r7	111	General Purpose Register. Aber: r7 bildet bei load/store die 8 LSBs der Adresse

Abbildung 14: Beschreibung der einzelnen Register des Registersatzes

Abbildung 14 zeigt die genaue Beschreibung der einzelnen Register des Registersatzes. Prinzipiell sind alle acht Register GPRs und entsprechend frei verfügbar. Register sechs (r6) und Register sieben (r7) haben einen weiteren Nutzen. Der Prozessor soll registerindirekte Adressierung des Speichers unterstützen, damit der Zugriff auf Arrays möglich ist. Dafür wird r7 bei registerindirekten Speicherzugriffen implizit adressiert und der Inhalt von r7 wird immer als Adresse für den registerindirekten Speicherzugriff verwendet. Die Besonderheit von r6 wird im weiteren Verlauf diskutiert. Wichtig ist, dass die vier niederwertigsten Bits bei Bedarf in ein vier Bit breites Spezialregister (*Page Register*) übernommen werden können.

Die Wortbreite der GPRs hängt von der Datenwortbreite des Prozessors ab. Für RISC-V werden zwei Varianten, 32 Bit und 64 Bit vorgeschlagen. Einfache Prozessoren unterstützen oft nur geringere Wortbreiten. Um beispielsweise die Logik für ALU-Operationen einfach zu halten, beschränken sich viele kompakte Prozessoren auf 16 Bit oder weniger. Diese einfache Implementierung eines Prozessors kann mit acht Bit Datenwortbreite auskommen. Daher ist die Datenbreite der GPRs acht Bit.

Spezialregister

Im nächsten Schritt definiert die ISA die verfügbaren Spezialregister des Prozessors, die in Abbildung 15 zu finden sind.

Special Register		
PC	11 Bit	Programm Counter
IR	16 Bit	Instruction Register
SCR	8 Bit	Status and Control Register (8 Steuer/Zustandsbits)
SP	12 Bit	Stack Pointer (für Stackzugriffe)
PR	4 Bit	Page Register (4 MSBs für 12 Bit Speicherzugriffe)

Abbildung 15: Verfügbare Spezialregister und ihre Beschreibung

Die Notwendigkeit von PC, IR und SP wurde bereits diskutiert. Diese Register sind allerdings prozessorintern und es ist kein Lese- bzw. Schreibzugriff auf diese möglich. Sie werden lediglich aus Gründen der Vollständigkeit in dieser Tabelle aufgeführt und sind aufgrund des eingeschränkten Zugriffes nicht relevant für die ISA. Ihre Wortbreite resultiert aus Größen, die im weiten Verlauf festgelegt werden (Adressbreite für Programm- und Datenspeicher, Befehlsbreite). Das *Status-and-Control-Register (SCR)* ist ein acht Bit Register, das neben Statusbits ein weiteres Kontrollbit beinhaltet, die alle in Abbildung 16 aufgelistet sind.

Status and Control Register		
C (Carry)	0	Statusbit: Carry Bit der letzten 8 Bit Operation gesetzt
V (oVerflow)	1	Statusbit: Letzte Operation hat ein Overflow ausgelöst (signed)
N (Negative)	2	Statusbit: Vorzeichenbit der letzten Operation
S (Signed)	3	Statusbit: Vorzeichenbit des korrekten Ergebnisses der letzten Operation
Z (Zero)	4	Statusbis: Letztes Ergebnis ist 0
P (write Page select)	5	Steuerbit: Schreibe r6 4 LSBs in PR im nächsten Takt. Bit wird automatisch cleared
	6	----
	7	----

Abbildung 16: Beschreibung der einzelnen Bits des SCR

Die ersten fünf Bits sind Statusbits der ALU, die gespeichert werden, um den Zustand des Prozessors bzw. der ALU nach einer vorangehenden Rechenoperation festzuhalten. Sie orientieren sich an den Statusbits der ALU von AVR-Mikrocontrollern, wie sie von [4, S. 1] genannt werden und werden in der ALU generiert. Das nächste Bit (*P-Bit, write page select*) dient dazu, die bereits angesprochenen, vier niederwertigsten Bits von r6 in das folgende Spezialregister, das Page-Register, zu schreiben. Diese Operation dauert einen Takt, was bedeutet, dass nach dem Setzen von `P=1` der Wert von r6 im nächsten Takt erst in das Page-Register übernommen werden kann. Die zwei letzten Bits sind aktuell noch unbelegt.

Das letzte Spezialregister, das *Page-Register*, wird in Bezug auf indirekte Speicherzugriffe verwendet. Bei registerindirekten Speicherzugriffen ist die Adressbreite des Speichers (bzw. des darüber adressierbaren Speichers) auf die Wortbreite der GPRs limitiert, in diesem Fall acht Bit oder 256 Speicherzellen. Der Prozessor soll aber einen größeren Datenspeicher als 256x1 Byte unterstützen. Die Lösung erfolgt über das Page-Register. Die tatsächliche Adresse bei der registerindirekten Adressierung ergibt sich bei dem Prozessor dieser Arbeit aus den vier Bits des Page-Registers gefolgt von acht weiteren Bits aus r6. Anders ausgedrückt: $Addr = Page \ll 8 + r6$. So wird in diesem Fall ein Datenspeicher mit

einer Adressbreite von zwölf Bits unterstützt. Auf die Größen der Speicherbereiche geht der folgende Abschnitt ein.

Sprungbedingungen

Im nächsten Abschnitt der ISA wird eine Maske für die Sprungbedingungen von bedingten Sprungbefehlen festgelegt. Mit dieser Maske können Statusbits im SCR angewählt werden, die bestimmen, ob ein Sprung ausgeführt werden soll oder nicht. Für die Sprungbedingung werden drei Bits im Befehlsformat reserviert, die zwischen acht verschiedenen Bedingungen unterscheiden können. Diese sind in Abbildung 17 aufgelistet.

Maske für Cond	Zugriff auf einzelne Statusbits bzw. Vergleichsoperatoren			
000	C		S=any	aus SCR
001	V		S=any	aus SCR
010	N		S=any	aus SCR
011	S		S=any	aus SCR
100	Z		S=any	aus SCR
101	==	inv: <>	S=any	Z==1 unsigned und signed
110	<	inv: >=	S=0	C==1
111	>	inv: <=	S=0	C or Z == 0
110	<	inv: >=	S=1	S == 1
111	>	inv: <=	S=1	S and Z == 0

Abbildung 17: Die in der ISA des Prozessors definierten Sprungbedingungen

Verschiedene ISA haben unterschiedliche Ansätze für das Definieren von Sprungbedingungen. Um zwei Möglichkeiten kombinieren zu können, sind die ersten fünf Sprungbedingungen das einfache Anwählen der Statusbits, d.h. die Abfrage, ob ein bestimmtes Bit im SCR gesetzt ist oder nicht. Die ISA der AVR-Mikrocontroller, die die Vorlage für die Statusbits dieses Prozessors ist, definiere nach [4, S. 21] Sprungbedingungen, die sich an logischen Vergleichsoperatoren wie $\geq$ orientierten. Nach der Subtraktion zweier Werte $a - b$ werden entsprechend die Statusbits von der ALU berechnet. Darauf basierend seien entsprechende Verknüpfungen der Statusbits notwendig, um z.B. $a > b$ abzufragen. Diese sind in den Sprungbedingungen sechs bis acht festgehalten. [4, S. 21] weist ergänzend darauf hin, dass bei solchen Vergleichsoperationen zwingend eine Unterscheidung zwischen vorzeichenbehafteten und vorzeichenlosen Operationen geschehen muss, da die Bedingungen dafür anders überprüft werden müssen. Dafür wurde im Befehlsformat der bedingten Sprünge eine weitere Bitstelle reserviert, die angibt, ob die Sprungbedingung für

vorzeichenbehaftete oder vorzeichenlose Vergleiche erfolgen soll. Abbildung 17 gibt nur den Überblick, wie die drei Bits für die Sprungbedingungen und das Bit zur Auswahl des Vorzeichens gesetzt werden müssen. Das genaue Befehlsformat wird nachfolgend definiert.

Speichergrößen

Der nächste Schritt ist die Festlegung der Speichergrößen, die adressiert werden sollen (s. Abbildung 18). Diese bestimmen maßgeblich die Busgrößen und einige Registergrößen des Prozessors.

Speicher	Adressbreite	Datenbreite	Gesamtspeicher
Programmspeicher	11	16 Bit	4 kB
Datenspeicher	12	8 Bit	4 kB

Abbildung 18: Speichergrößen des Prozessors

Für den Datenspeicher gilt eine Wortbreite von acht Bit, da dies bereits die festgelegte Datenbreite des Prozessors ist. Die Adressbreite beträgt bei registerindirekter Adressierung mindestens acht Bit. Ergänzt durch die vier Bit des Page-Registers ergeben sich zwölf Bit, eine mehr oder weniger frei gewählt Festlegung. Der Datenspeicher hat daher eine Speicherkapazität von $2^{12} * 1\ Byte = 4096\ Byte = 4kB$.

Für den Programmspeicher hängt die Datenbreite von der Befehlsbreite ab, die Adressbreite unter anderem davon, wie viel Speicher über Sprungbefehle adressiert werden kann, d.h. auch teilweise von der Befehlsbreite. Die genauen Festlegungen hierzu ergeben sich aus Abwägungen zu diesen verschiedenen Größen und wurden schließlich auf 16 Bit Befehlsbreite (s. weiter unten in diesem Kapitel) und elf Bit Adressbreite festgelegt, was auch hier einer Speicherkapazität von 4kB entspricht.

Damit sind auch die Datenbreiten von SP, PC und IR festgelegt (s. Abbildung 15).

Adressierungsmöglichkeiten

Bezüglich der Kodierung der Operanden bei ALU-Befehlen wird das Zweiadressformat gewählt. Ein Dreiadressformat sei bei den meisten RISC-Prozessoren wie bei RISC-V nach [5, S. 11] üblich, die AVR-ISA verwende, wie [4, S. 15] darstellt, z.B. ein Zweiadressformat. Für diese Architektur fällt die Entscheidung auf ein Zweiadressformat, da diese ein guter Kompromiss zwischen der Flexibilität in der Registerauswahl auf der einen Seite darstellt und auch die Mikroarchitektur vereinfacht

und auf der anderen Seite das Befehlsformat kompakt hält (sechs Bits für das Kodieren von Zielregister Rd und Quellregister Rs).

Bezüglich der Adressierung des Datenspeichers des Prozessors wurden die beiden wichtigsten Konzepte ausgewählt. Es soll sowohl eine direkte Adressierung über den Befehl ermöglicht werden als auch eine indirekte Adressierung über einen Registeriwert. Bei beiden Varianten werden die angegebenen acht Bit um die vier Bit aus dem Page-Register zu den zwölf Adressbits des Datenspeichers ergänzt. Die Angabe eines Offsets bei registerindirekten Befehlen soll zusätzlich den Arrayzugriff vereinfachen. Die Adressierungsmöglichkeiten des Datenspeichers sind in Abbildung 19 dargestellt.

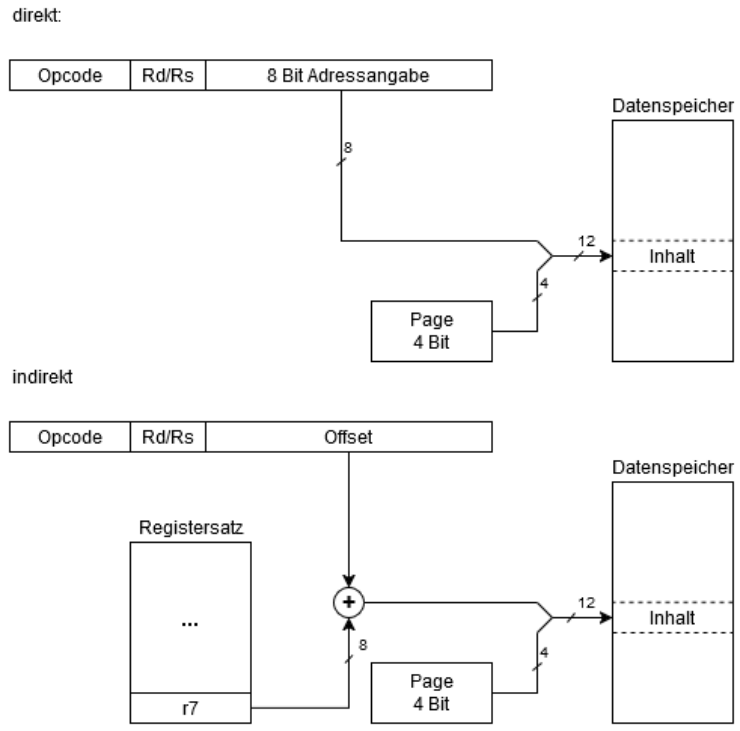


Abbildung 19: Speicherzugriffsmöglichkeiten des Prozessors: direkter und indirekter Zugriff#

Befehlskodierung und Befehlsformate

Im nächsten Schritt soll eine passende Kodierung für die in „2.4 Typische Befehle eines Befehlssatzes“ vorgestellten Befehlsgruppen eingeführt werden. Der gesamte Befehlssatz soll Befehle aus allen Befehlsgruppen enthalten. Zur Kodierung müssen die notwendigen Operanden wie `Rd` und `Rs` bei ALU-Befehlen oder Adressangaben bei Speicherzugriffen im Befehlsformat enthalten sein. Die Notwendigkeit von entsprechenden Operanden ergibt sich aus den Erläuterungen zu den Befehlsgruppen

und den getroffenen Definitionen zur ISA. Die Länge des Opcodes und die daraus resultierende Befehlsmenge wurde auf fünf Bit, d.h. 32 Befehle festgelegt, die für einen einfachen RISC-Prozessor vollkommen ausreichen. Ursprünglich wurden zwei unterschiedliche Kodierungen zu dieser ISA festgelegt, eine 16-Bit-Variante und eine 12-Bit-Variante. Beide Varianten sind mit Erläuterungen zu den Operanden im Anhang 1.2 (16 Bit) und im Anhang 2.1 (Zwölf Bit) zu finden.

Die Entscheidung zwischen beiden Varianten fiel zugunsten der 16-Bit-ISA, weshalb bereits einige Festlegungen auf dieser basieren. Ursprünglich wurde die 12-Bit-Variante entwickelt, da die meisten Befehle maximal 6 Bits zur Kodierung der Operanden verwenden (`Rd` und `Rs`). Für einige andere Befehle, die Adressen kodieren müssen, wird ein erweiterter Befehl verwendet, der aus zwei Teilen besteht und im zweiten Teil die Operanden mitliefert, d.h. die Speicheradresse. Die 16-Bit-Variante verzichtet hingegen auf erweiterte Befehle und kodiert alle Befehle einheitlich in 16 Bit, wobei in Kauf genommen wird, dass einige Bits bei den meisten Befehlen ungenutzt bleiben. Argumente für die Verwendung der 16-Bit-Variante sind, dass alle Befehle nach RISC-Vorgaben die gleiche Länge besitzen und entsprechend die gleiche Zeit benötigen, um geladen zu werden in der Fetch-Phase. Das hält die Fetch-Schaltung simpel und die notwendige Ladezeit geringer. Außerdem sind die meisten Speicherbausteine in n Byte Speicherzellen strukturiert, weshalb ein Programm mit zwölf Bit Befehlsbreite trotzdem in einem Speicher mit 16 Bit Wortbreite untergebracht werden müsste. Aus diesen Gründen fällt die Entscheidung auf die Verwendung der 16-Bit-Variante zur Kodierung der Befehle, die grob in Abbildung 20 zu finden ist.

Befehlsgruppe	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arithmetisc-logische Befehle	Opcode					Rd			X	X	X	X	X	Rs		
Schiebe- und Rotationsbefehle	Opcode					Rd			X	X	X	X	X	X	X	X
Datenbewegungsbefehle																
Register-Register	Opcode					Rd			X	X	X	X	X	Rs		
Register-Speicher indirekt	Opcode					Rd			Offset					Rs		
Register-Speicher direkt	Opcode					Rd/Rs			Address							
Immediate	Opcode					Rd			Data							
Programmsteuerbefehle																
Absolute Sprungbefehle	Opcode					Address										
Relative Sprungbefehle	Opcode					S	X	X	Offset					Cond		
Systemsteuerbefehle	Opcode					Rd			X	X	X	X	X	Rs		

Abbildung 20: 16-Bit Befehlsformate des Prozessors dieser Arbeit

Vollständiges Befehlsset

Im letzten Schritt der Definition der ISA werden verschiedene notwendige Befehle den einzelnen Befehlsgruppen zugeordnet. Dabei wurden verschiedene Auflistungen von wichtigen Befehlen aus [2, S. 170], der RISC-V ISA und der AVR-ISA in jeweils stark heruntergebrochener Form verwendet, um notwendige Befehle zu identifizieren. Alle Befehle sind mit Kommentaren im Anhang 1.3 zu finden. Zu den Befehlen zählen unter anderem:

- Verschiedene ALU-Befehle mit den wichtigsten arithmetischen und logischen Operationen: `add`, `sub`, `inc`, `dec`, `and`, `or`, `xor`, `not`
- Eine Möglichkeit zum Addieren/Subtrahieren mit vorangehendem Übertrag, um Ganzzahlen größer als acht Bit zu verrechnen: `addc`, `subc`
- Logische Schiebeoperationen nach links und rechts: `sll`, `slr`
- Datenbewegungsbefehle zwischen Register ⇔ Register (`mov`), Register ⇔ Speicher direkt (`ld`, `st`) und indirekt (`stz`, `ldz`) und ein immediate-Befehl zum Laden einer Konstante (`ldi`)
- Befehle für Stackzugriff (`psh`, `pll`)
- Sprung Befehle, bedingt/unbedingt und absolut/relativ: `jpa`, `bra`, `brs`, `brc`
- Befehle für Unterprogramme: `call`, `ret`
- Befehle für den Zugriff auf das SCR: `lcr`, `stcr`

Mit diesen Definitionen ist die ISA vollständig und kann im nächsten Schritt durch den Entwurf einer Mikroarchitektur implementiert werden.

3.3 Entwurf der Mikroarchitektur

Zu Beginn des Entwurfs der Mikroarchitektur wird eine Festlegung vorgenommen bezüglich des Speichers. Das Speichermodell, dass der Prozessor sowohl für den Daten- als auch für den Programmspeicher unterstützt, ist ein asynchroner Speicher, der den Inhalt der angegebenen Adresse ausgibt an seinem Ausgang, sobald die Adresse geändert wurde bzw. angelegt wurde. Die Schreibvorgänge des Speichers sind synchron zu dem Takt der CPU und müssen entsprechend in einem Takt der CPU erfolgen.

Im zweiten Schritt wird festgelegt, dass die Abarbeitung der Befehle in dem Prozessor nach dem Pipeline-Prinzip erfolgen soll, damit dieses Konzept zu

Demonstrationszwecken implementiert ist. Da Pipelining die Vorgänge im Prozessor und die übergeordnete Logik deutlich komplexer gestalten kann (s. „2.3 Vergleich unterschiedlicher Designphilosophien“), fällt die Entscheidung auf eine einfache, zweistufige Pipeline. In der ersten Stufe wird ein Befehl geladen (Fetch), in der zweiten Stufe wird er dekodiert und ausgeführt (Decode und Execute). Da die Execute-Phase ein komplexes Schaltnetz mit langen Signallaufzeiten darstellt, ergibt sich in Kombination mit der Decode-Phase eine sehr lange Signallaufzeit durch diese Pipelinestufe. Würde die Decode-Stufe in der Fetch-Stufe implementiert werden, würden mehr Pufferregister in der Execute-Stufe benötigt werden, was das Design komplexer gestaltet. Da die Pipeline nur zu Demonstrationszwecken dienen soll, bleibt es bei dem Pipelineentwurf Fetch →Decode/Execute. Daraus ergibt sich die folgende Pipeline-Grundstruktur:

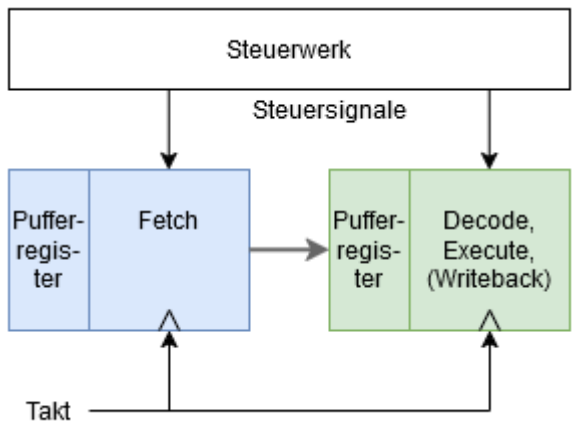


Abbildung 21: Pipeline Grundstruktur der Mikroarchitektur

Der Inhalt der Pufferregister ergibt sich aus den für die Stufe notwendigen Daten. Um einen Befehl laden zu können, benötigt die Fetch-Phase Informationen über die Adresse des nächsten Befehls im Programmspeicher. Diese ist im PC zu finden und ist zusätzlich abhängig von den Zieladressen oder dem Offset von Sprungbefehlen. Daher ist diese Information relevant für die Fetch-Phase und der PC kann als Pufferregister für die Fetch-Phase angenommen werden. Weiterhin benötigt Fetch Zugriff auf den Programmspeicher.

Die Fetch-Phase lädt einen Befehl aus den Programmspeicher und gibt diesen an die Execute-Phase weiter. Diese muss den aktuellen Befehl speichern, da die Fetch-Phase bereits den nächsten Befehl lädt. Das Speichern des aktuellen Befehls übernimmt das IR, weshalb dieses das Pufferregister der Execute-Phase darstellt. Da auch der Datenspeicherzugriff (Writeback) teil der Befehlsausführung ist, benötigt Execute zudem Zugriff auf den Datenspeicher. Die erweiterte Grundstruktur der Pipeline sieht nun folgendermaßen aus:

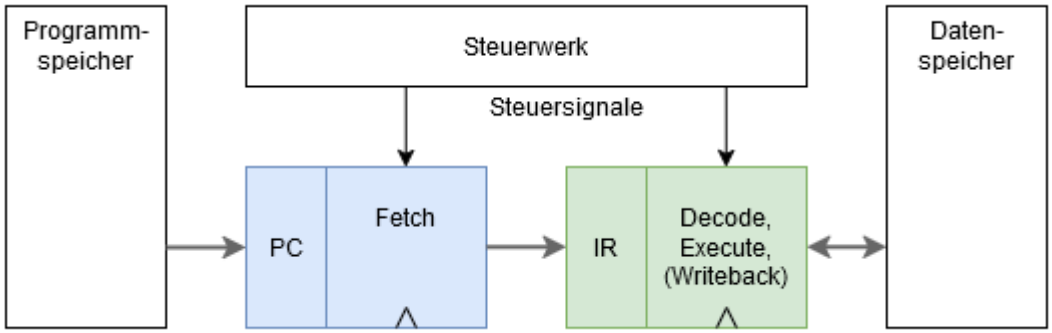


Abbildung 22: Erweitere Pipelinestruktur mit Pufferregistern und Speicherschnittstellen

Im Folgenden wird die Hardwarestruktur der beiden Pipelinephasen näher betrachtet. Dazu werden Ausschnitte aus der fertigen Mikroarchitektur (Blockdiagramm) verwendet, die im Anhang 3 zu finden ist.

Fetch:

In der Fetch-Phase sind zwei Funktionen zu beachten. Zunächst findet die Adressierung des Programmspeichers nicht immer direkt über den Befehlszähler statt, sondern kann auch über Sprungbefehle ablaufen. Damit ein Befehl, der einem Sprungbefehl im Programmspeicher folgt, direkt in der Fetch-Phase geladen werden kann, muss direkt die Zieladresse des dekodierten Befehls aus dem Decoder an die Fetch-Phase übergeben werden. Der PC zeigt im Normalfall nur auf die dem Befehl folgende Adresse, die im Falle eines Sprungs daher nicht den korrekten Befehl enthält. Die vier Möglichkeiten zur Adressierung des Programmspeichers sind:

1. Adresse kommt vom Stack (d.h. Datenspeicher) im Falle eines return-Befehls.
2. Adresse berechnet sich aus einem Offset vom aktuellen Befehl ausgehend. Dazu muss zum Inhalt des PCs (zeigt auf nächste Adresse) der Offset addiert werden und um eins verringert werden, da der Offset vom aktuellen Befehl (PC – 1, wenn dieser schon in der Execute-Phase ist) berechnet wird.
3. Nächster Befehl im Programmspeicher ohne Sprung. Dies ist einfach der Inhalt des PCs.
4. Adresse wird im Befehl kodiert (absoluter Sprung) und kann unverändert übernommen werden.

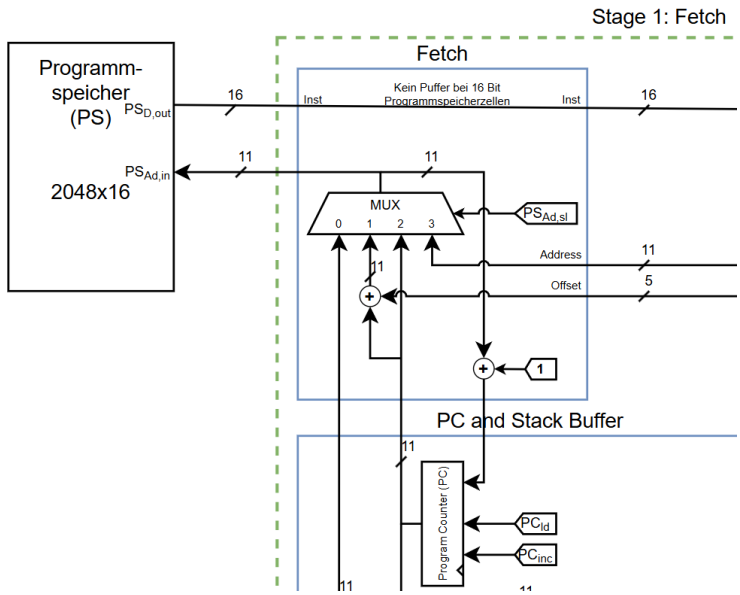


Abbildung 23: Berechnung der Programmspeicheradressen

Basierend auf dem aktuellen Befehl muss daher ein entsprechende Adressierungsmethode ausgewählt werden, um zum nächsten Befehl zu gelangen. Alle Adressen werden parallel bereitgestellt und über einen Multiplexer (Auswahlschaltung, MUX) kann das Steuerwerk eine Adresse auswählen. Die Schaltung ist in Abbildung 23 gezeigt. Anschluss 0 des MUX geht zum Stack, von dem die Rücksprungadresse bei return geladen wird. Anschluss 1 berechnet die Summe aus Offset-1 und PC. Die absolute Adresse und der Offset kommen vom Decoder. Der PC ist unten in der Abbildung zu sehen. Damit der PC nach dem Laden eines Befehls aus einer Adresse A im nächsten Takt, wenn sich A in der Execute-Phase befindet, auf die Adresse $A + 1$ zeigt, wird der Registerinhalt des PCs zu Beginn des nächsten Taktes aktualisiert und auf den Wert $A + 1$ gesetzt. Dazu wird unabhängig von der Adressierungsmethode allgemein der MUX-Ausgang um eins erhöht und in den PC geladen.

Um bei `ret` eine elf Bit breite Programmspeicheradresse vom acht Bit breiten Datenspeicher zu holen bzw. um bei `call` entsprechend diese Adresse im Datenspeicher ablegen zu können, ist eine Schnittstelle notwendig, die die elf Bit Adresse aufteilt und in zwei Takten nacheinander an den Stack weitergibt. Das hat zur Folge, dass die Ausführung von `ret` bzw. `call` jeweils zwei Takte in Anspruch nimmt, bis die Speicherschnittstelle des PCs die Adressen übertragen hat. Diese Schnittstelle besteht aus einem MUX und einem Pufferregister, das den zweiten Teil der Adresse zwischenspeichert. Die Ansteuerung dieser Schnittstelle sowie das Anhalten der Pipeline für den zweiten Takt ist Aufgabe des Steuerwerks. Ein Zeitdiagramm zur

Veranschaulichung zeigt Abbildung 24. Das Signal „PC to Stack“ gibt beispielsweise nacheinander die Adresse 301 (Adresse des nächsten Befehls nach `call 100`, 0x012D) aus. Im ersten Takt, in dem sich `call` im IR befindet (Execute-Phase), wird 0x01 übergeben, im zweiten Takt wird 0x2D übergeben. Dabei bleibt die Pipeline stehen, denn in der Zeit wird kein neuer Befehl in das IR geladen. Die genaue Implementierung der Pufferschnittstelle ist der vollständigen Mikroarchitektur im Angang zu entnehmen.

call/ret							
Takt	0	1	2	3	4	5	6
PC	300	301	301	101	102	102	302
PC_in	301	101	101	102	103	302	303
PS_Add,in	300	100	100	101	102	301	302
PS_D,out	CALL 100	PS(100)	PS(100)	RET		PS(301)	
PC Buffer Out	0	0	0x2D	0x2D	0x2D	0x2D	0x2D
PC to Stack		0x01	0x2D				
PC Buffer In	0	0	0	0	0	0x2D	0x2D
PC from Stack					0x2D	0x01	
IR		CALL 100	CALL 100	PS(100)	RET	RET	PS(301)
Opcode		CALL	CALL	z.B. NOP	RET	RET	
Address (PS)		100	100				

Abbildung 24: Zeitdiagramm bei call/ret-Befehlen mit Pipeline Stopp und Datenfluss zwischen PC und Stack

Decode und Execute

In diesem Abschnitt wird die zweite Pipeline-Stufe entwickelt. Diese decodiert mithilfe eines Schaltwerkes (Decoder) den Befehl und führt diesen über die ALU aus. Diese Stufe enthält daher die ALU und den Registersatz.

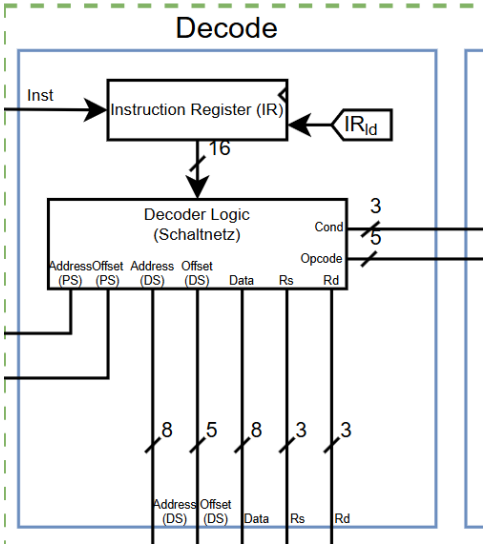


Abbildung 25: Symbolschaltbild des Decoders mit IR in der zweiten Pipelinestufe des Prozessors

Der Decoder besteht aus dem Pufferregister dieser Stufe, dem IR, und einem einfachen Schaltwerk bzw. einer Signalverschaltung, damit aus einem Befehl alle in den Befehlsformaten genannten Operanden dekodiert werden können (s. Abbildung 25). Der Decoder gibt unabhängig vom Opcode immer die Bitfolge aus, an der sich die entsprechenden Operanden im Befehl befinden. Das heißt, dass einige dekodierte Operanden keinen Sinn ergeben bzw. keine Bedeutung haben, wenn ein Befehl diese Operanden nicht enthält (s. Abbildung 26). Das ist kein Problem, da das Steuerwerk so arbeitet, dass die anderen Hardwarekomponenten nur die für den aktuellen Befehl notwendigen Operanden verarbeiten. Abbildung 26 verdeutlicht die Arbeitsweise des Decoders am Beispiel des Befehls `ld r5, 0x37`.

Maschinenbefehl																
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Opcode					**Operands**											
1	0	0	1	1	1	0	1	0	0	1	1	1	1	0	0	Eigentliche Bedeutung: ld r5, 0x37
					1	0	1	0	0	1	1	1	1	0	0	Programmspeicher Adresse (ohne Bedeutung)
								0	0	1	1	1				Offset Programmspeicher (ohne Bedeutung)
								0	0	1	1	1	1	0	0	Datenspeicher Adresse (wird verwendet)
								0	0	1	1	1				Datenspeicher Offset (ohne Bedeutung)
								0	0	1	1	1	1	0	0	Immediate Daten (ohne Bedeutung)
													1	0	0	Quellregister Rs (ohne Beduetung)
					1	0	1									Zielregister Rd (wird verwendet)
													1	0	0	Sprungbedingung (ohne Bedeutung)
							1									vorzeichenbehafteter Sprung (ohne Bedeutung)

Abbildung 26: Arbeitsweise des Decoders am Beispiel des Befehls ld r5, 0x37

Der Decoder ist so verschaltet, sodass die verschiedenen Operanden innerhalb der CPU an die Stellen verteilt werden, wo sie benötigt werden. Die Programmspeicheradresse bei absoluten Sprungbefehlen und der Offset bei relativen Sprungbefehlen wird beispielsweise an die Fetch-Phase der Pipeline weitergereicht und dort wie bereits beschrieben verwendet. Der Opcode und die Sprungbedingung sind mit dem Steuerwerk verbunden, damit dieses basierend auf dem Opcode Steuersignale zur Befehlsausführung setzen kann.

Der nächste Block ist der Execute-Block in dieser Pipelinestufe. Dieser besteht aus der Recheneinheit (ALU) und dem Registersatz mit entsprechender Verschaltung.

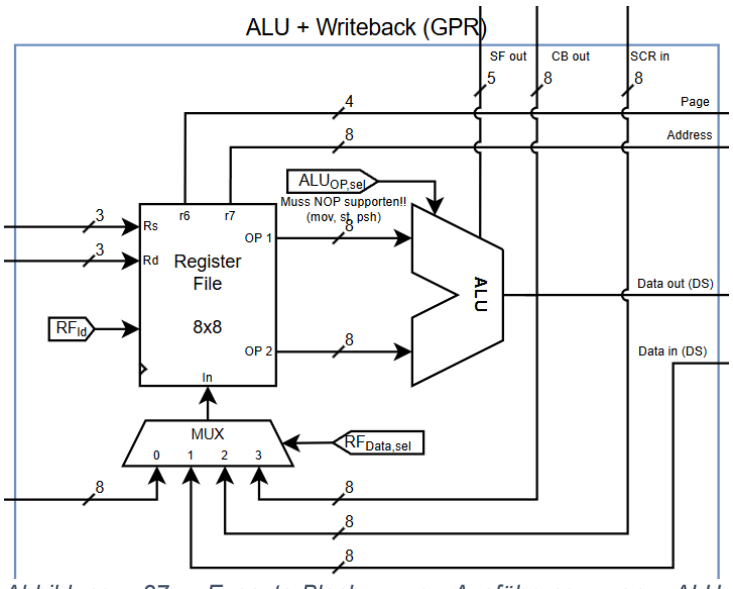


Abbildung 27: Execute-Block zur Ausführung von ALU-Berechnungen

Der Registersatz ist zentraler Bestandteil des Execute-Blocks und besteht aus den acht GPRs mit entsprechender Beschaltung. Über die beiden Eingänge wird ein `Rd` und ein `Rs` ausgewählt, die beim Zweiadressformat die ALU-Operanden adressieren. Daher wird der Inhalt des über `Rd` adressierten Registers über den Ausgang OP 1 an die ALU übergeben, der Inhalt von `Rs` übe OP 2 als zweiter Operand für die ALU. Außerdem verfügt der Registersatz über einen weiteren Eingang (In). Ist das Steuersignal RF_{ld} gesetzt, wird das acht Bit Datenwort an diesem Eingang bei einer positiven Taktflanke

in das Register `Rd` übernommen. So wird die Writeback-Funktion des Registersatzes realisiert. Nach der Ausführung der Berechnung (bzw. der Bereitstellung der neuen Daten für `Rd` über In) werden die neuen Daten am Ende des Pipelinetaktes übernommen. Über einen MUX vor In werden die in `Rd` zu schreibenden Daten ausgewählt, da nicht nur das ALU-Ergebnis in `Rd` geschrieben werden kann. Über den MUX-Anschluss 0 wird die Immediate-Konstante vom Decoder (bei `ldi`) an den Registersatz übergeben, über 1 wird ein Datenwort vom Datenspeicher übernommen (bei `ld`, `ldz`, `pll`) und über den Anschluss 2 wird der Inhalt des SCR ausgelesen und in `Rd` transferiert. Anschluss 3 gibt das ALU-Ergebnis an den Registersatz weiter.

Die ALU wird wie bereits angesprochen direkt vom Registersatz basierend auf `Rs` und `Rd` mit den Operanden versorgt und gib das Ergebnis wieder an diesen zurück. Um Daten aus `Rs` in den Datenspeicher zu übertragen oder in das SCR zu schreiben, wird direkt der Ausgang der ALU verwendet. Indem neben den Rechenoperationen auch eine Nulloperation (d.h. es wird direkt `Rs` am ALU-Ausgang ausgegeben) möglich ist, muss nicht direkt ein Ausgang vom Registersatz verwendete werden, um Daten vom Registersatz zum Datenspeicher und zum SCR zu übertragen. Die Rechenoperation, die die ALU ausführen soll, wird über ein Steuersignal festgelegt und kann entsprechend vom Steuerwerk gewählt werden. Die möglichen Operationen richten sich nach den ALU-Befehlen, die in der ISA spezifiziert sind.

Die ISA spezifiziert für die Register r6 und r7 des Registersatzes eine weitere Bedeutung neben ihrer Rolle als SCR bei der Adressierung des Datenspeichers. Daher werden die Inhalte von r6 und r7 direkt an die Ansteuerung des Datenspeichers weitergegeben.

Writeback (Data Storage)

Die letzte Komponente der zweiten Pipeline-Stufe ist der Writeback-Bereich für den Datenspeicher. Dieser beinhaltet die Adressgenerierung basierend auf der Adressierungsart des aktuellen Befehls, die Auswahl der zu schreibenden Daten für den Datenspeicher und die Verwaltung des Stacks.

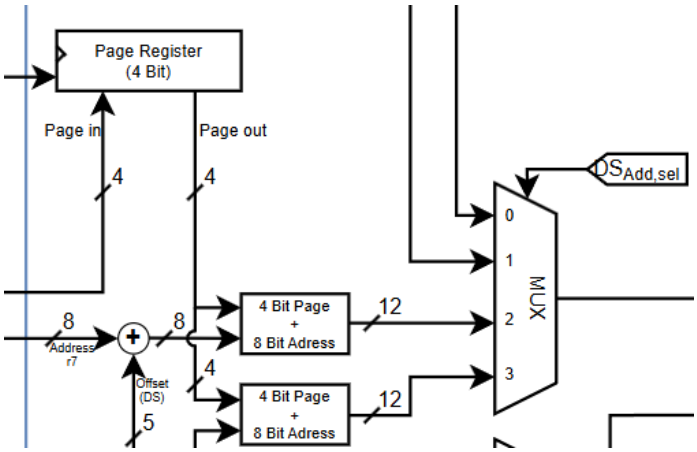


Abbildung 28: Adressberechnung und -auswahl der Mikroarchitektur

Die Berechnung der Adresse bei direkter Adressierung erfolgt durch das Voranstellen des Inhaltes des Page-Registers. Bei indirekter Adressierung wird der ISA entsprechend ein Offset zu r7 addiert und das Page-Register vorangestellt. Über einen MUX wird vom Steuerwerk ausgewählt, ob eine direkte Adressierung, eine indirekte Adressierung oder ein Zugriff auf den Stack vorliegt und die errechnete Adresse entsprechend an den Adresseingang des Datenspeicher geleitet (s. Abbildung 28). Die Anschlüsse 0 und 1 des MUX kommen vom Stack und entsprechen dem nächsten freien Stackeintrag (`psh`, Anschluss 0) bzw. dem ersten belegten Stackeintrag (`pll`, Anschluss 1).

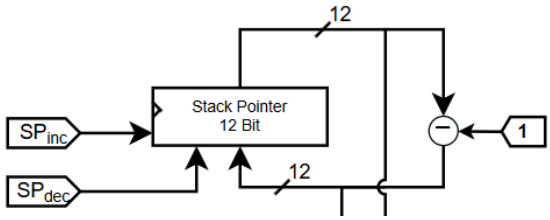


Abbildung 29: Stackpointer des Prozessors

Die Stackverwaltung mittels des SPs ist in Abbildung 29 dargestellt. Es wird festgelegt, dass der SP immer auf den nächsten freien Stackeintrag zeigt und nach oben wächst. So muss der SP nicht mit einem bestimmten Wert initialisiert werden, sondern kann mit Null initialisiert werden. Aus dieser Festlegung folgt, dass der SP bei `psh` inkrementiert werden muss und bei `pll` dekrementiert werden muss. Außerdem muss bei `pll` im Gegensatz zu `psh` bereits das Dekrementieren stattgefunden haben, bevor der Zugriff auf den Datenspeicher stattfindet. Daher werden zwei unterschiedliche Adresssignale an den MUX zur Auswahl der Speicheradresse übergegen.

Der synchrone Schreibzugriff auf den Datenspeicher wird über die Steuerlogik gesteuert. Die zu schreibenden Daten kommen aus dem Execute-Block oder aus der Fetch-Phase (der Pufferschaltung zum Stack). Die Datenauswahl findet auch durch das Steuerwerk statt.

Steuerwerk

Zuletzt folgt der Entwurf des Steuerwerks. Das Steuerwerk dient der Steuerung der Vorgänge im Prozessor, indem unter anderem die bereits genannten Steuersignale entsprechend gesetzt werden. Auch die Ablaufsteuerung der Pipeline gehört zur Aufgabe des Steuerwerks.

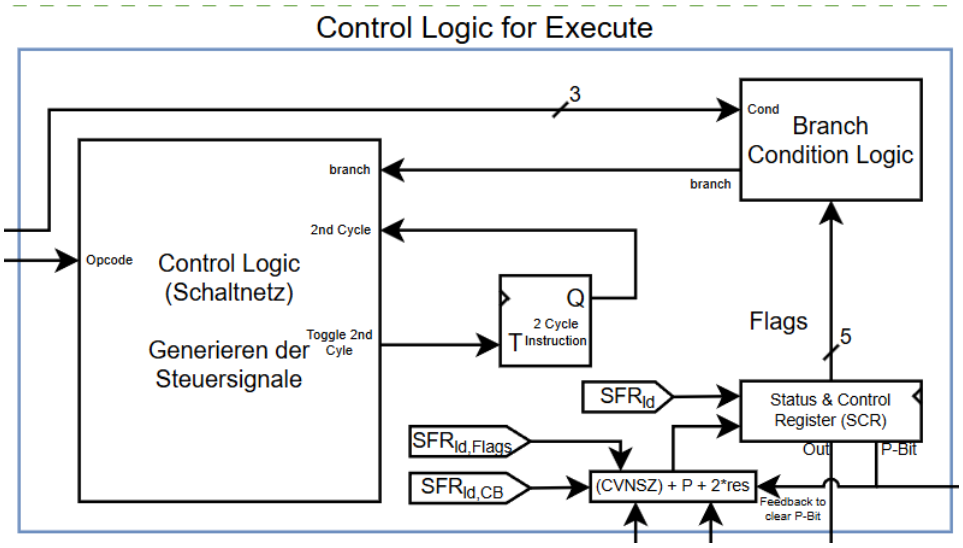


Abbildung 30: Das Steuerwerk des Prozessors mit seinen Komponenten

Abbildung 30 zeigt die Komponenten des Steuerwerks. Das SCR befindet sich im Steuerwerk und wird nach jeder ALU-Operation mit den entsprechenden Flags der ALU beschrieben. Auch Schreibzugriffe per Befehl auf das SCR werden von der Steuerlogik vorgenommen. Ein Überschreiben der Statusbits wird verhindert und das automatische Löschen des P-Bits nach Schreibzugriff auf das Page-Registers wird vorgenommen. Mithilfe der Statusbits aus dem SCR und der angewählten Sprungbedingung wertet die „Branch Condition Logic“ aus, ob die Sprungbedingung für einen bedingten Sprung erfüllt ist und dieser ausgeführt werden soll. Außerdem wird ein Ein-Bit-Register („2-Cycle Instruction“) verwendet, um bei call/return-Befehlen (Dauer: 2 Take wie bereits ausgeführt) festzuhalten, in welchem Ausführungstakt sich der Prozessor befindet. Die entsprechenden Steuersignale zur Ausführung eines bestimmten Befehls werden von einem Schaltnetz generiert. Sie sind neben dem Opcode, der den Befehl selbst bestimmt, abhängig vom Ergebnis der „Branch Condition Logic“ (bei Sprungbefehlen) und vom aktuellen Ausführungstakt bei call/return.

Um eine Wahrheitstabelle für die Steuersignale zu erstellen bzw. um zu definieren, welche Steuersignale für die Ausführung welches Befehls gesetzt werden müssen, hilft zunächst eine Übersicht über die Steuersignale und ihre Wirkung. Die folgenden Signalbezeichnungen stammen aus dem Blockschaltbild der Mikroarchitektur. Zusätzlich ist in Klammern die Bezeichnung in der später folgenden VHDL-Implementierung angegeben:

- $PS_{Ad,sl}$ (PrStrAddrSel): Steuersignal am MUX zur Adressauswahl des Programmspeichers.
  - "00": Anwahl gespeicherte Adresse vom Stack
  - "01": Anwahl berechnete Adresse für relativen Sprung
  - "10": Anwahl Adresse für nächsten Befehl im Programmspeicher (PC)
  - "11": Anwahl Adresse für absoluten Sprung von Decoder
- PC_{ld} (PCload): Steuersignal zum Laden des neuen Wertes für den PC (d.h. inkrementieren bzw. bei Sprung neuen Sprungwert laden)
- $PC_{ld,in}$ (PCldInBuf): Eingangspufferregister für Schnittstelle Stack zu PC laden, um zweiten Teil der Adresse zwischenzuspeichern am Eingang
- $PC_{ld,out}$ (PCldOutBuf): Ausgangspufferregister für Schnittstelle PC zu Stack laden, um zweiten Teil der Adresse zwischenzuspeichern am Ausgang
- $PC_{sel,out}$ (PCselOutBuf). Anwahl MUX am Ausgangspuffer, damit einmal der untere und einmal der obere Adressteil zum Stack übertragen wird.
- IR_{ld} (IRload): Befehlsregister laden. Neuen Befehl von Fetch-Stufe in Execute-Stufe übernehmen.
- RF_{ld} (RegFileLoad): Wert am Eingang In des Registersatzes in das über Rd angewählte Register übernehmen.
- $ALU_{OP,sel}$ (AluOpSel): Auszuführende ALU-Operation auswählen (s. Abbildung 31)
- $RF_{Data,sel}$ (RegFileDataSel): Auswahl der zu ladenden Daten für den Registersatz:
  - "00": Immediate-Konstante vom Decoder
  - "01": Daten bei Lesezugriff Datenspeicher
  - "10": Aktueller Wert SCR

```
-- 0b0000   : nop
-- 0b0001   : add
-- 0b0010   : addc
-- 0b0011   : sub
-- 0b0100   : subc
-- 0b0101   : inc
-- 0b0110   : dec
-- 0b0111   : and
-- 0b1000   : or
-- 0b1001   : xor
-- 0b1010   : not
-- 0b1011   : sll
-- 0b1100   : slr
```

Abbildung 31: ALU-Operationen des Prozessors

    - "11": Ergebnis von ALU
- SP_{inc} (SPinc): Stackpointer inkrementieren
- SP_{dec} (SPdec): Stackpointer dekrementieren
- $DS_{Add,sel}$ (DataStrAddrSel): Steuersignal am MUX zur Adressauswahl des Datenspeichers:
    - "00": Wert SP bei `psh`-Zugriff auf Stack
    - "01": Wert SP - 1 bei `pll`-Zugriff auf Stack
    - "10": Berechnete Adresse bei direkter Adressierung
    - "11": Berechnete Adresse bei indirekter Adressierung
- $DS_{in,sel}$ (DataStrInSel): Auswahl der in den Datenspeicher zu schreibenden Daten. Entweder aus ALU, d.h. Register, oder vom PC
- DS_{ld} (DataStrLoad): Steuersignal für Schreibzugriff auf Datenspeicher
- $SFR_{ld,CB}$ (SFRldCB): Schreibzugriff auf die Steuerbits (P-Bit)
- $SFR_{ld,F}$ (SFRldFlags): Steuersignal zum Speichern der ALU-Statusbits im SCR
- (set2ndCyle): Steuersignal zum Einleiten des zweiten Ausführungstaktes von call/ret

Diese Übersicht bietet einen guten Überblick, welche Möglichkeiten das Steuerwerk hat, die Abläufe im Prozessor zu steuern. Im zweiten Schritt werden Zeitdiagramme zur Ausführung der einzelnen Befehle erstellt, die die Datenflüsse im Prozessor zur Abarbeitung der einzelnen Befehle darstellen. Abbildung 32 zeigt beispielhaft das Zeitdiagramm, wie der Befehl `pll r0` abgearbeitet werden muss in der Mikroarchitektur des Prozessors. Daraus lassen sich die notwendigen Steuersignale für die Abarbeitung des Befehls ableiten.

pll (Stack - Register)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	PLL r0			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	PLL r0		
Opcode		PLL		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd		r0		
Rs				
RF_in		5		
r0	0	0	5	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR	0	0	0	
SP	1	1	0	
DS_Add,in		0		
DS_D,out		5		
DS_D,in				
DS(0)	5	5	5	
DS(1)				

Abbildung 32: Timing Diagramm für den Befehl `pll r0`

Zu Beginn von Takt 0 befindet sich der Prozessor im Ausgangszustand. Der Befehl befindet sich in der Fetch-Phase. Die Steuersignale werden (noch) vom aktuellen Befehl bestimmt, da dieser vorgibt, ob „normal“ geladen werden soll. Der pull-Befehl liegt am Datenausgang des Programmspeichers an und wird mit der nächsten positiven Flanke in das IR übernommen, d.h. die Execution-Phase des Befehls beginnt und die Steuersignale werden basierend auf dem Opcode von `pll` gesetzt. Der decodierte Operand ist `Rd`, das Zielregister, in das der Wert vom Stack geladen wird. Da der pull-Befehl kein Sprungbefehl ist, soll der im Speicher darauffolgende Befehl in der Fetch-Stufe geladen werden und die Adresse im PC verwendet werden. Daher muss $PC_{Add,sel} = \text{"00"}$ sein. Der Befehl dauert in der Ausführung nur einen Takt, weshalb der PC inkrementiert werden muss und $PC_{ld} = 1$ sein muss. Außerdem muss deswegen das IR am Ende der Execution-Phase mit der nächsten Flanke geladen werden, damit der neue Befehl von der Fetch-Stufe übernommen wird ($IR_{ld} = 1$). Da die aktuelle Adresse vom PC nicht auf dem Stack gespeichert werden muss (nur `call`/`ret`), müssen die drei nachfolgenden Steuersignale nicht gesetzt werden, da diese die Schnittstelle PC ⇔ Stack betreffen. $RF_{ld} = 1$ sorgt dafür, dass der vom Stack geladene Wert in das Zielregister `Rd` übernommen wird. Die ALU muss für den pull-Befehl keine Operation ausführen, daher ist das Steuersignal $ALU_{Op,sel}$ irrelevant. Um den Wert vom Stack (=Datenspeicher) an den Dateneingang des Registersatzes zu übergeben, muss $RF_{Data,sel} = \text{"01"}$ gesetzt werden. Bei einem pull-Zugriff auf den

Stack muss der nach oben wachsende Stackpointer dekrementiert werden, also $SP_{inc} = 0$ und $SP_{dec} = 1$ und $DS_{Add,sel} = "01"$ per obiger Definition. Da kein Schreibzugriff auf den Datenspeicher geschieht, ist der Wert für $DS_{in,sel}$ irrelevant und $DS_{ld} = 0$. Für `pll` müssen keine Bits im SCR gesetzt werden, da es kein Rechenbefehl der ALU ist. Somit gilt $SFR_{ld,CB} = SFR_{ld,F} = 0$. Das Setzen dieser Steuersignale ergibt sich aus deren Definition und dem Verhalten von `pll` im Zeitdiagramm. So wird der Wert „5“, der auf dem Stack abgelegt war, an den Eingang vom Registersatz angelegt und nach der Execute-Phase in diesen übernommen.

Alle weiteren Zeitdiagramme und die notwendigen Steuersignale für die Befehle sind im Anhang 4 und im Anhang 5 zu finden und wurden nach dem Muster des obigen Beispiels erstellt. Mit der Definition der Steuersignale zur Ausführung der einzelnen Befehle ist die Wahrheitstabelle der Steuerlogik im Steuerwerk vollständig definiert und die Mikroarchitektur des Prozessors dieser Arbeit abgeschlossen.

4. Implementierung in VHDL

Im Anschluss an den Entwurf des Prozessors mit ISA und Mikroarchitektur soll dieser in einer Hardwarebeschreibungssprache implementiert werden, damit die Funktionalität des Prozessors simuliert und verifiziert werden kann. Die verwendete HDL für diese Implementierung ist VHDL. In diesem Abschnitt werden die Kenntnisse von VHDL vorausgesetzt und lediglich einige Besonderheiten der Implementierung dokumentiert. Der gezeigt Programmcode wird minimal gehalten, damit stattdessen der Aufbau und die Strukturierung des VHDL-Codes verdeutlicht wird. Die Implementierung selbst wird auf Basis des ausführlichen Blockdiagramms zur Mikroarchitektur erstellt und kann ohne viele weitere Definitionen erfolgen.[1]

Grundsätzlich wurde das Design in VHDL trotz der vorangehenden Definition von verschiedenen Größen gemäß der Zielsetzung erweiterbar gehalten. Das bedeutet, dass verschiedene Größen wie die Daten-/Adressbreiten oder die Registerzahl im Registersatz mithilfe von sog. Generics variabel bleiben, jedoch mit den definierten Werten als Standardwert initialisiert werden. Die nach außen hin sichtbaren Generics des Prozessors sind:

- `N: integer := 8;` -- Datenbreite des Prozessors
- `DataAddrWidth: integer := 12;` -- Adressbreite Datenspeicher
- `InstWidth: integer := 16;` -- Befehlsbreite
- `InstAddrWidth: integer := 11;` -- Adressbreite Programmspeicher
- `OpcodeSize: integer := 5;` -- Bits für Opcode im Befehl
- `rCount: integer := 8;` -- Anzahl GPRs im Registersatz

Nach Angabe dieser Generics müssen noch zwei weitere Voraussetzungen gegeben sein, damit der Prozessor Befehle eigenständig abarbeiten kann:

1. Der Hauptspeicher (hier Programm- und Datenspeicher) ist kein Teil eines Prozessors und ist entsprechend nicht in der gleichen VHDL-Datei wie der Prozessor selbst implementiert. Daher müssen zwei Speicher instanziiert

[1] Hinweis: Der gesamte VHDL-Programmcode sowie weitere ergänzende Dokumentationen zum Prozessor (ISA, Mikroarchitektur, VHDL-Signalbenennungen, Programmierung, …) sind in GitHub unter dem folgenden Link zu finden: https://github.com/alexditi/cpu-design

werden, die die Voraussetzungen für Speichermodule dieser CPU erfüllen, und der CPU zur Verfügung gestellt werden.

2. Es muss ein externes Reset-Signal und ein Taktsignal angegeben werden. Um die CPU „neustarten“ zu können, ist ein externes Reset-Signal notwendig, die alle Registerinhalte und den Zustand der CPU auf definierte Anfangswerte zurücksetzt, damit die Verarbeitung des Programmes im Programmspeicher von vorne beginnen kann. Auch das Taktsignal der CPU wird klassisch extern z.B. einem Quarzoszillator generiert und an die CPU übergeben. Beim Instanziieren der CPU müssen daher eine Quelle für Reset- und Taktsignal angegeben werden.

Sind diese Voraussetzungen erfüllt (entweder in VHDL simuliert oder bei einer realen Implementierung an die CPU angeschlossen), kann die CPU Befehle abarbeiten und die Funktionsweise erfolgreich simuliert werden. Am Ende dieses Kapitels werden Testergebnisse gezeigt.

Struktur der VHDL-Implementierung

Entsprechend des Aufbaus der Mikroarchitektur, ist die Implementierung der CPU modular gehalten und in thematisch bzw. räumlich zueinander passende Module gegliedert. Die Abhängigkeiten der einzelnen Komponenten sind in der Abbildung im Anhang 6 dargestellt.

Durch diese modulare Implementierung ist es möglich, einzelne Module wie ein Register an verschiedenen Stellen wiederzuverwenden. Außerdem können so einzelne Verarbeitungsstufen unabhängig voneinander getestet werden, wie z.B. die ALU oder die Fetch-Stufe. So werden Fehler im fertigen CPU-Modul minimiert. Für jedes einzelne Modul existiert ein eigenes Testprogramm, eine sog. Testbench. Für die ALU sind die Ergebnisse der Testbench in Form eines Zeitdiagramms in Abbildung 33 gezeigt.

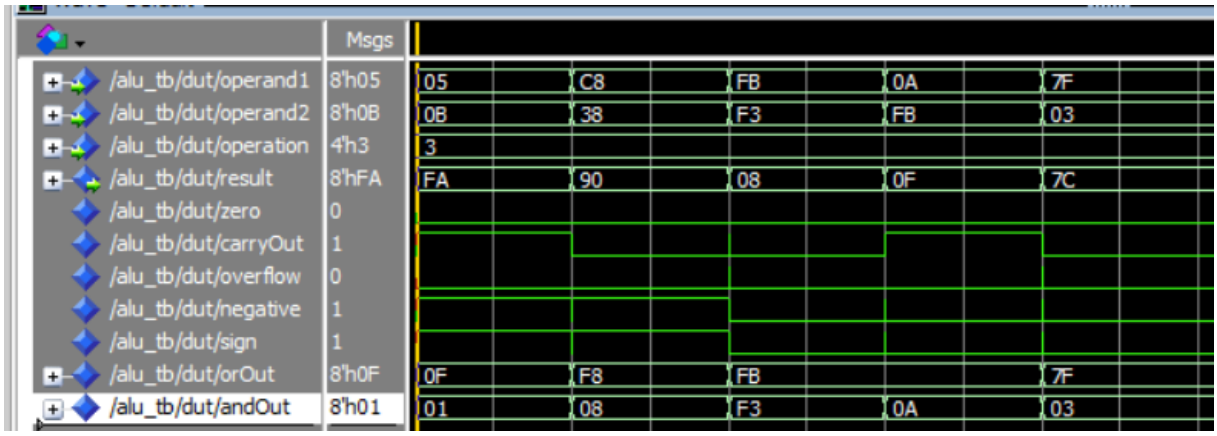


Abbildung 33: Ergebnis der ALU-Testbench in Form eines Zeitdiagramms

In den ersten beiden Zeilen werden über `operand1` und `operand2` verschiedene Werte an die ALU übergeben. Über `operation` in der dritten Zeile wird die auszuführende Operation angewählt. In diesem Fall bedeutet `operation=3`, dass eine Subtraktion durchgeführt wird. Im ersten Beispiel wird $05_{16} - 0B_{16}$ berechnet. Das Ergebnis wird in der vierten Zeile unter `result` ausgegeben und lautet „FA_{16}“. Die darunterliegenden Flags `carryOut` und `overflow` geben Auskunft über das Ergebnis. Betrachtet man die Operanden als vorzeichenlose Zahl, wird die Rechnung $5 - 11$ durchgeführt. Das korrekte Ergebnis müsste -6 lauten. Bei vorzeichenlosen Operanden muss auch das Ergebnis vorzeichenlos betrachtet werden. Dann gilt $FA_{16} = 250_{10}$. Das Ergebnis ist falsch, was uns durch `carryOut=1` angezeigt wird. Es zeigt für vorzeichenlose Operationen einen Überlauf an. Das korrekte Ergebnis -6 liegt außerhalb der Wertemenge von Null bis 255 bei vorzeichenlosen 8-Bit-Ganzzahlen. Das angezeigte Ergebnis ist daher falsch, was durch `carryOut=1` signalisiert wird. Bei einer vorzeichenbehafteten Rechnung wird auch die Rechnung $5 - 11$ durchgeführt. Allerdings muss dann auch das Ergebnis als vorzeichenbehaftete Zahl angesehen werden. Dann gilt $FA_{16} = -6_{10}$. Das ist das korrekte Ergebnis, was auch durch `overflow=0` signalisiert wird. Bei vorzeichenbehafteten Zahlen liegt bei dieser Rechnung kein Überlauf vor. In den letzten beiden Zeilen ist außerdem zu erkennen, dass die ALU auch die anderen Operationen parallel durchführt, wie z.B. or und and. Auch diese Ergebnisse sind korrekt:

- **or:** $05_{16} \vee 0B_{16} = 0000\ 0101_{16} \vee 0000\ 1011_{16} = 0000\ 1111_{16} = 0F_{16}$
- **and:** $05_{16} \wedge 0B_{16} = 0000\ 0101_{16} \wedge 0000\ 1011_{16} = 0000\ 0001_{16} = 01_{16}$

Mithilfe solcher Zeitdiagramme und den Testbenches kann daher über ein Simulationsprogramm für VHDL (hier: Intel QuestaSim) die Funktionalität einzelner Prozessorkomponenten verifiziert werden.

Als weiteres Beispiel soll die Implementierung des Speichers gezeigt werden. Da die Anforderungen bzgl. des Verhaltens beider Speichermodule gleich sind (s. „3.3 Entwurf der Mikroarchitektur“), sind beide Speicher Instanzen der VHDL-Architektur `ram_block.vhdl`. In VHDL ist dieser Speicher als Array von einzelnen Registern anzusehen. Dabei haben die Register die Datenbreite des Speichers und die Anzahl an Registern entspricht der Anzahl an verfügbaren Adressen im Speicherblock. Damit ist es möglich, in einem Takt ein Datenwort im Speicher zu speichern und asynchron Daten am Datenausgang auszugeben. Abbildung 34 zeigt den VHDL-Programmcode zur Implementierung des Speicherblocks. Die angewählte Adresse wird in Zeile 64 von Binärsignal am Eingang in einen Integer für einfache Handhabung umgewandelt. In Zeile 80 ist der asynchrone Lesezugriff realisiert. Unabhängig vom Taktsignal wird im Array `ram` der Inhalt der angewählten Adresse ausgelesen und am Datenausgang ausgegeben. Der Schreibzugriff ist synchron zum Taktsignal. Bei einer positiven Flanke (Zeile 69) und angewählten Schreibsignal (`WriteEn`, Zeile 70), wird in das `ram`-Array der aktuelle Wert am Dateneingang taktsynchron und ohne Verzögerung geschrieben.

```
61  begin
62
63          -- Convert Address
64          Addr <= to_integer(unsigned(AddrIn));
65
66          -- Process: Synchrones Schreiben
67          ram_process: process(clk) is
68          begin
69                  if rising_edge(clk) then
70                          if WriteEn = '1' then
71                                  ram(Addr) <= DataIn;
72                          elsif rst = '1' and initPS = '1' then
73                                  -- Programmspeicher mit Programm initialisieren
74                                  ram <= initRam;
75                          end if;
76                  end if;
77          end process ram_process;
78
79          -- Daten asynchron lesen
80          DataOut <= ram(Addr);
81
82  end rtl;
```

Abbildung 34: Implementierung des RAM-Speichermoduls in VHDL

Beim Instanziieren des Programmspeichers wird zusätzlich der Generic-Parameter `initPS=1` angegeben. Beim Prozessorreset (`rst=1`) wird in Zeile 74 des

Programmcodes im Falle des Programmspeichers bei `initPS=1` eine Initialisierungsroutine für den Programmspeicher aufgerufen, in der das aktuelle Programm für den Prozessor aus einer Textdatei `program.txt` in den Programmspeicher übertragen wird. Das erleichtert Testfunktionen für die CPU, indem in der Testbench der CPU nicht mehr zunächst der Programmspeicher manuell beschrieben werden muss. Abbildung 35 zeigt das asynchrone Leseverhalten des Programmspeichers. Sobald eine Änderung der Adresse am Adresseingang `PrStrAddrIn` vorliegt, ändern sich die Daten am Datenausgang `PrStrDataOut`.

Abbildung 35: Verhalten des (Programm-)speichers beim Ladevorgang

Testbench der gesamten CPU

Zuletzt soll anhand eines Beispielprogrammes die Arbeitsweise der CPU in der Simulation gezeigt werden. Das Beispielprogramm beinhaltet einige simple Befehle, um die grundlegenden Abläufe in der CPU zu demonstrieren und ist in Abbildung 36 gezeigt.

Mnemonic	Opcode Binär	Opcode Hex	Wirkung
T E S T 1 (General)			
ldi r0, 0x05	10100 000 00000101	0xA005	r0=5
mov r1, r0	01101 001 00000 000	0x6900	r1=5
st 0xFE, r0	10010 11111110 000	0x97F0	ram(FE) = 5
ld r2, 0xFE	10011 010 11111110	0x9AFE	r2=5
add r0, r1	00001 000 00000 001	0x0801	r0=10
sub r1, r2	00011 001 00000 010	0x1902	r1=0
inc r2	00101 010 00000000	0x2A00	r2=6
dec r0	00110 000 00000000	0x3000	r0=9

Abbildung 36: Beispielprogramm zur Ausführung auf der CPU in Pseudoassemblercode und codiert in Maschinencode

Über `ldi` wird der Wert 05_{16} in das Register `r0` geladen. Danach wird er mithilfe von `mov` kopiert in das Register `r1` und schließlich in die Speicherzelle an Adresse FE_{16} in den Datenspeicher geschrieben. Aus dieser Adresse wird der Wert in das Register `r2` geladen. Nach diesen Befehlen steht in den Registern `r0`, `r1` und `r2` der Wert 05_{16} zur Verfügung. Zunächst wird der Datenfluss zwischen den Pipelinestufen betrachtet und die Wege der Befehle durch die CPU erläutert.

Abbildung 37: Datenfluss zwischen den Pipelinestufen bei der Abarbeitung des Beispielprogrammes

Nach dem Reset gibt der PC den Wert „0“ (`PCout`) als Adresse für den Programmspeicher aus. Sobald diese an den Adresseingang des Programmspeichers angelegt wird (`PrStrAddrIn`), gibt dieser den ersten Befehl $\mathrm{A001}_{16}$ aus. Mit der nächsten positiven Flanke wird dieser Befehl an die zweite Pipelinestufe übergeben und wird im IR gespeichert (`Instruction`). In Abbildung 37 wird deutlich, dass immer bei einer positiven Flanke der Befehl am Ausgang des Programmspeichers (`PrStrDataOut`) in das IR übernommen wird. Während der aktuelle Befehl im IR dekodiert und ausgeführt wird, kann parallel in der Fetch-Phase der nächste Befehl aus dem Programmspeicher geladen werden, indem zunächst die korrekte Adresse berechnet oder aus dem PC geladen wird und an den `PrStrAddrIn` angelegt wird.

Abbildung 38: Verhalten des Decoders während der Ausführung des Beispielprogramms

Während sich ein Befehl in der zweiten Pipelinestufe befindet, wird dieser zunächst vom Decoder dekodiert. Die dekodierten Operanden der einzelnen Befehle sind in Abbildung 38 dargestellt. Beim ersten Befehl (`ldi r0, 0x05.` Zu finden unter `Instruction=A005`) wird unter `RdOut` das korrekte Zielregister (`r0`) dekodiert, unter `DataImmediate` ist der Wert $\mathrm{05}_{16}$ zu finden, der in `r0` geladen werden soll. Beim nächsten Befehl (`mov r1, r0`) wird das Zielregister `r1` und das Quellregister `r0` dekodiert. Bei den folgenden beiden Befehlen mit Datenspeicherzugriff wird z.B. die korrekte Adresse (FE_{16}) dekodiert unter `DataAddrOut`.

Sobald der Opcode dekodiert ist, generiert das Steuerwerk die passenden Steuersignale für den Befehl in der zweiten Pipelinestufe. Diese werden entsprechend der Vorgaben aus „3.3 Entwurf der Mikroarchitektur“ generiert. Das Zeitdiagramm des Steuerwerks zum Beispielprogramm ist im Anhang 7 zu finden.

Abbildung 39 zeigt das Verhalten des Execute-Blocks im Zeitdiagramm.

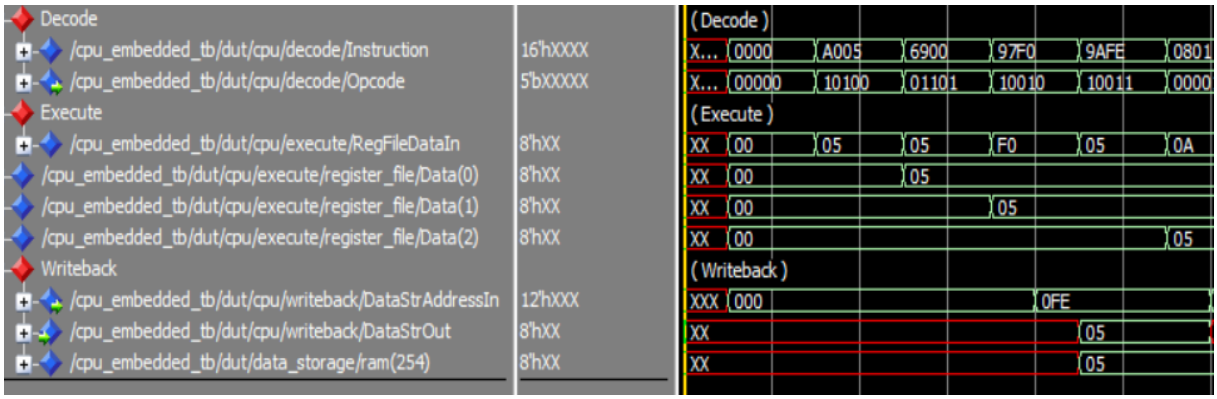


Abbildung 39: Funktionsweise des Execute-Blocks im Zeitdiagramm

Die Daten am Eingang zum Registersatz sind in der Zeile für `RegFileDataIn` zu finden. Dort werden vom MUX die passenden Daten zum Schreiben angewählt. Der Wert 05_{16} liegt bei den Befehlen `ldi`, `mov` und `ld` dort an und wird nach deren Execute-Phase (d.h. mit der nächsten positiven Flanke) in die Register übernommen (letzte drei Zeilen vor Writeback). Am Ende der Ausführung sind alle drei Register `r0`, `r1` und `r2` wie erwartet mit dem Wert 05_{16} beschrieben.

5. Fazit

Zusammenfassend lässt sich sagen, dass die Zielsetzung, eine funktionsfähige ISA mitsamt Mikroarchitektur zur Implementierung dieser ISA zu entwickeln, erfüllt wurde. Die ISA und die Mikroarchitektur enthalten verschiedene Grundkonzepte zur Demonstration von grundlegenden Designansätzen von Prozessoren, wie z.B. die wichtigsten Befehlsformate, eine Pipeline zur Befehlsabarbeitung und einen Stack-Speicherbereich.

Auch die Implementierung in einer HDL konnte erfolgen. Entsprechende Tests zum Verifizieren der Funktionsfähigkeit des Designs wurden durchgeführt. Außerdem wurde darauf geachtet, den Entwurf und die Implementierung erweiterbar zu gestaltet.

Eine kritische Betrachtung einiger Punkte bezüglich einer realen Implementierung ist jedoch an dieser Stelle durchzuführen.

Ein Punkt ist der Speicherzugriff auf Programm- und Datenspeicher. Sowohl der asynchrone Zugriff als auch der zum Prozessortakt synchrone Zugriff sind beide in der Praxis nicht realisierbar. Ein Speicherbaustein arbeitet normalerweise nach einem eigenen Takt, zu dem sowohl Lese-, als auch Schreibzugriff synchron ablaufen. Die Latenzzeiten sind weitaus größer und Zugriffe synchron zum Prozessor-/Pipelinetakt unmöglich. Daher werden verschiedene Möglichkeiten verwendet, um diese Latenzzeiten, zumindest von der ISA aus betrachtet, zu kompensieren.

Der zweite Punkt ist der sehr asymmetrische Aufbau der Pipeline. Das bedeutet, dass die Verarbeitungszeiten bzw. die Signallaufzeiten beider Pipelinestufen stark unterschiedlich sind. Um die Taktzeit des Prozessors verringern zu können, sollten alle Pipelinestufen gleiche bzw. ähnliche Laufzeiten haben, damit keine Leerlaufzeiten der Pipelinestufen mit den geringeren Laufzeiten entstehen. In dem Falle des Prozessors dieser Arbeit könnte der Deocde-Block zur Pipelinestufe eins verschoben werden. Alternativ kann in Kombination mit der verbesserten Ansteuerung für reale Speichermodule die Pipeline mit mehr Stufen ausgestattet werden. Das erfordert eine deutlich komplexere Pipelinesteuerung, die dann auftretende Pipelinekonflikt überwacht.

Der letzte Punkt betrifft die Datenpfade, welche in der aktuellen Implementierung recht komplex ausfallen. Diese können vereinfacht werden, indem Datenbusse z.B. für

Daten zum Programmspeicher zusammengefasst werden und um die notwendige Bussteuerung erweitert werden.

Während eine Implementierung in der Praxis daher noch stellenweise eine Überarbeitung des Prozessordesigns erfordert, waren erste Tests in der Simulation schon erfolgreich. Aufbauend auf diesem Design können somit Implementierungen von weiteren Grundkonzepten ergänzt werden.

Literaturverzeichnis

[1] U. Brinkschulte und T. Ungerer, Mikrocontroller und Mikroprozessoren, Heidelberg: Springer-Verlag Berlin Heidelberg, 2010.

[2] D. Page, A Practial Introduction to Computer Architecure, London: Springer-Verlag London Limited, 2009.

[3] M. Menge, Moderne Prozesssorarchitekturen - Prinzipien und ihre Realisierungen, Heidelberg: Springer-Verlag Berlin Heidelberg, 2005.

[4] Atmel Corporation, „Microchip Technology : Atmel Products,“ November 2016. [Online]. Available: https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf. [Zugriff am 10 10 2024].

[5] A. Waterman und K. Asanović, „The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2,“ RISC-V Foundation, Berkeley, 2017.

Anhangsverzeichnis

Anhang 1: 16 Bit ISA

Anhang 1.1: Allgemeine Definitionen

GPRs	**Kodierung**	**Beschreibung**
r0	000	General Purpose Register
r1	001	General Purpose Register
r2	010	General Purpose Register
r3	011	General Purpose Register
r4	100	General Purpose Register
r5	101	General Purpose Register
r6	110	General Purpose Register. Aber: bei P=1 in CSR werden die 4 LSBs in PR geschrieben
r7	111	General Purpose Register. Aber: r7 bildet bei load/store die 8 LSBs der Adresse
Special Register		
PC	11 Bit	Programm Counter
IR	16 Bit	Instruction Register
SCR	8 Bit	Status and Control Register (8 Steuer/Zustandsbits)
SP	10 Bit	Stack Pointer (für Stackzugriffe)
PR	4 Bit	Page Register (4 MSBs für 12 Bit Speicherzugriffe)
Status and Control Register		
C (Carry)	0	Statusbit: Carry Bit der letzten 8 Bit Operation gesetzt
V (oVerflow)	1	Statusbit: Letzte Operation hat ein Overflow ausgelöst (signed)
N (Negative)	2	Statusbit: Vorzeichenbit der letzten Operation
S (Signed)	3	Statusbit: Vorzeichenbit des korrekten Ergebnisses der letzten Operation
Z (Zero)	4	Statusbis: Letztes Ergebnis ist 0
P (write Page select)	5	Steuerbit: Schreibe r6 4 LSBs in PR im nächsten Takt. Bit wird automatisch cleared
	6	
	7	

Maske für Cond	**Zugriff auf einzelne Statusbits bzw. Vergleichsoperatoren**				
000	C		S=any	aus SCR	
001	V		S=any	aus SCR	
010	N		S=any	aus SCR	
011	S		S=any	aus SCR	
100	Z		S=any	aus SCR	
101	==	inv: <>	S=any	Z==1 unsigned und signed	
110	<	inv: >=	S=0	C==1	siehe ATMega ISA S.21
111	>	inv: <=	S=0	C or Z == 0	siehe ATMega ISA S.21
110	<	inv: >=	S=1	S == 1	siehe ATMega ISA S.21
111	>	inv: <=	S=1	S and Z == 0	siehe ATMega ISA S.21

Speicher	Adressbreite	Datenbreite	Gesamtspeicher
Programmspeicher	11	16 Bit	4 kB
Datenspeicher	12	8 Bit	4 kB

Datenspeicher Adressierung über 8 Bit im Register / im Befehl und weitere 4 Bit im Page-Register. Das Page-Register wird über das P-Bit im CR aktiviert. Page-Register: 4 LSBs von r6
Programmspeicher Adressierung über 10 Bit im PC. Bei jpa 10 Bit direkt im Befehl, bei brbs, brbc und bra wird ein relativer Offset von 5 Bit (max +-32) angegeben

Addressierung		
Programmspeicher:		
	A:	11 Bit PC im normalen Zugriffverfahren bei Programmablauf
	B:	11 Bit direkt im Befehl kodiert (jpr, jsr). Diese werden in den PC geladen
	C:	5 Bit Offset im Befehl kodiert, der zum aktuellen PC addiert wird (signed)
Datenspeicher:		
	A:	8 Bit direkt im Befehl kodiert (st/ld). Diese bilden die 8 LSBs, PR bildet die 4 MSBs
	B:	8 Bit aus r6 bilden die 8 LSBs, PR bildet die 4 MSBs (bei stz/ldz)

Anhang 1.2: Befehlsformate

Befehlsgruppe	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Arithmetisc-logische Befehle	**Opcode**						**Rd**		X	X	X	X	X	**Rs**			Rd: Zielregister	Rs: Quellregister	! Zielregister ist auch Operand 1
Schiebe- und Rotationsbefehle	**Opcode**						**Rd**		X	X	X	X	X	X	X	X	Rd: Zielregister		! Zielregister ist auch Operand
Datenbewegungsbefehle																			
Register-Register	**Opcode**						**Rd**		X	X	X	X	X	**Rs**			Rd: Zielregister	Rs: Quellregister	! psh/pll haben nur Rd oder Rs
Register-Speicher indirekt	**Opcode**						**Rd**		**Offset**					**Rs**			Rd: Zielregister	Rs: Quellregister	! stz/ldz nutzen r7 als Adressangabe. Entweder Rd oder Rs
Register-Speicher direkt	**Opcode**						**Rd/Rs**		**Address**								Rs/Rd: Ziel-/Quellr	Address: Adresse	! st/ld entweder Rd oder Rs. Diese sind dann an den entsprechenden Stellen
Immediate	**Opcode**						**Rd**		**Data**								Rd: Zielregister	Data: Daten	! Direktes Laden des 8 Bit Datenworts in Rd
Programmsteuerbefehle																			
Absolute Sprungbefehle	**Opcode**						**Address**												Address: Zieladresse ! Address ist eine absolute Adressangabe im Programmnspeicher
Relative Sprungbefehle	**Opcode**					**S**	X	X	**Offset**					**Cond**			Offset: Sprungweite	Cond: Sprungbedingt.	! Cond ist ein optionaler Parameter nur bei brbs, brbc. Nicht bra
Systemsteuerbefehle	**Opcode**						**Rd**		X	X	X	X	X	**Rs**			Rd: Zielregister	Rs: Quellregister	! Rd und Rs optional je nach Befehl (hlt, lcr, stcr)

Anhang 1.3: Befehlsset

Mnemonik	Maschinenbefehl																Beschreibung (intern)	Beschreibung (Text)
	Opcode				Operands													
nop	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	---	1 Wartetakt ohne Operation
add	0	0	0	0	1	Rd			X	X	X	X	X	X	X	Rs	Rd <= Rd + Rs	Addition der Registerinhalte Rd + Rs
addc	0	0	0	0	1	0	Rd		X	X	X	X	X	X	X	Rs	Rd <= Rd + Rs + C	Addition der Registerinhalte Rd + Rs mit vorangehendem Übertrag
sub	0	0	0	0	1	1	Rd		X	X	X	X	X	X	X	Rs	Rd <= Rd - Rs	Subtraktion der Registerinhalte Rd - Rs
subc	0	0	0	1	0	0	Rd		X	X	X	X	X	X	X	Rs	Rd <= Rd - Rs - C	Subtraktion der Registerinhalte Rd - Rs mit vorangehendem Übertrag
inc	0	0	0	1	0	1	Rd		X	X	X	X	X	X	X	X	Rd <= Rd + 1	Inkrementieren des Registerinhalts Rd um 1
dec	0	0	0	1	1	0	Rd		X	X	X	X	X	X	X	X	Rd <= Rd - 1	Dekrementieren des Registerinhalts Rd um 1
and	0	0	0	1	1	1	Rd		X	X	X	X	X	X	X	Rs	Rd <= Rd AND Rs	UND-Verknüpfung der Registerinhalte Rd AND Rs
or	0	1	0	0	0	0	Rd		X	X	X	X	X	X	X	Rs	Rd <= Rd OR Rs	ODER-Verknüpfung der Registerinhalte Rd OR Rs
xor	0	1	0	0	0	1	Rd		X	X	X	X	X	X	X	Rs	Rd <= Rd XOR Rs	XOR-Verknüpfung der Registerinhalte Rd XOR Rs
not	0	1	0	1	0	Rd			X	X	X	X	X	X	X	X	Rd <= NOR Rd	Logische Invertierung des Registerinhalts Rd
sll	0	1	0	1	1	Rd			X	X	X	X	X	X	X	X	Rd <= Rd << 1	Logisches Verschieben des Inhalts von Rd um 1 Bitstelle nach links
slr	0	1	1	0	0	Rd			X	X	X	X	X	X	X	X	Rd <= Rd >> 1	Logisches Verschieben des Inhalts von Rd um 1 Bitstelle nach rechts

mov	0	1	1	0	1	Rd		X	X	X	X	X	Rs				Rd <= Rs	Kopieren des Registerinhalts von Rs nach Rd
psh	0	1	1	1	0	X	X	X	X	X	X	X	Rs				MEM[SP] <= Rs; SP <= SP + 1	Speichern des Registerinhalts von Rs auf dem Stack
pll	0	1	1	1	1	Rd		X	X	X	X	X	X	X	X	X	Rd <= MEM[SP - 1]; SP <= SP - 1	Speichern des ersten Stackeintrags im Register Rd
stz	1	0	0	0	0	X	X	X	Offset				Rs				MEM[r7 + Offset] <= Rs	Speichern des Inhalts von Rs im Speicher an der Adresse in r7 + Offset
ldz	1	0	0	0	1	Rd		Offset					X	X	X		Rd <= MEM[r7 + Offset]	Laden des Inhalts von der Adresse in r7 + Offset des Speichers in Rd
st	1	0	0	1	0	Address							Rs				MEM[Address] <= Rs	Speichern des Inhalts von Rs im Speicher an der angegebenen Adresse
ld	1	0	0	1	1	Rd		Address									Rd <= MEM[Address]	Laden des Inhalts von der angegebenen Adresse im Speicher in Rd
ldi	1	0	1	0	0	Rd		Data									Rd <= Data	Laden des angegebenen Wertes in Rd
jpa	1	0	1	0	1	Address											PC <= Address	Springen zur absolut angegebenen Adresse im Programm
bra	1	0	1	1	0	X	X	X	Offset				X	X	X		PC <= PC + Offset	Springen um den relativ angegebenen Offset im Programm
brs	1	0	1	1	1	S	X	X	Offset				Cond				PC <= PC + Offset IF Cond=1	Springen um den relativ angegebenen Offset im Programm, wenn Cond true
brc	1	1	0	0	0	S	X	X	Offset				Cond				PC <= PC + Offset IF Cond=0	Springen um den relativ angegebenen Offset im Programm, wenn Cond false
call	1	1	0	0	1	Address											MEM[SP,+1] <= PC, SP <= SP + 2	Aufrufen der Subroutine an der angegebenen Adresse
ret	1	1	0	1	0	X	X	X	X	X	X	X	X	X	X		PC <= MEM[SP - 1,2], SP <= SP - 2	Zurückkehren von der Subroutine ins Hauptprogramm
res2	1	1	0	1	1	X	X	X	X	X	X	X	X	X	X	X	nop	freier Befehl (aktuell nop)
res3	1	1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	nop	freier Befehl (aktuell nop)
lcr	1	1	1	0	1	Rd		X	X	X	X	X	X	X	X	X	CR <= Rs	Lade den Inhalt des Control Registers in Rd. C, V, N, S, Z: write protected
stcr	1	1	1	1	0	X	X	X	X	X	X	X	Rs				Rd <= CR	Speichere den Inhalt von Rs im Control Register (CVNSZ read only)
hlt	1	1	1	1	1	X	X	X	X	X	X	X	X	X	X	X	CLK off	Gesamten Prozessor und Programmausführung anhalten

Anhang 2: 12 Bit ISA

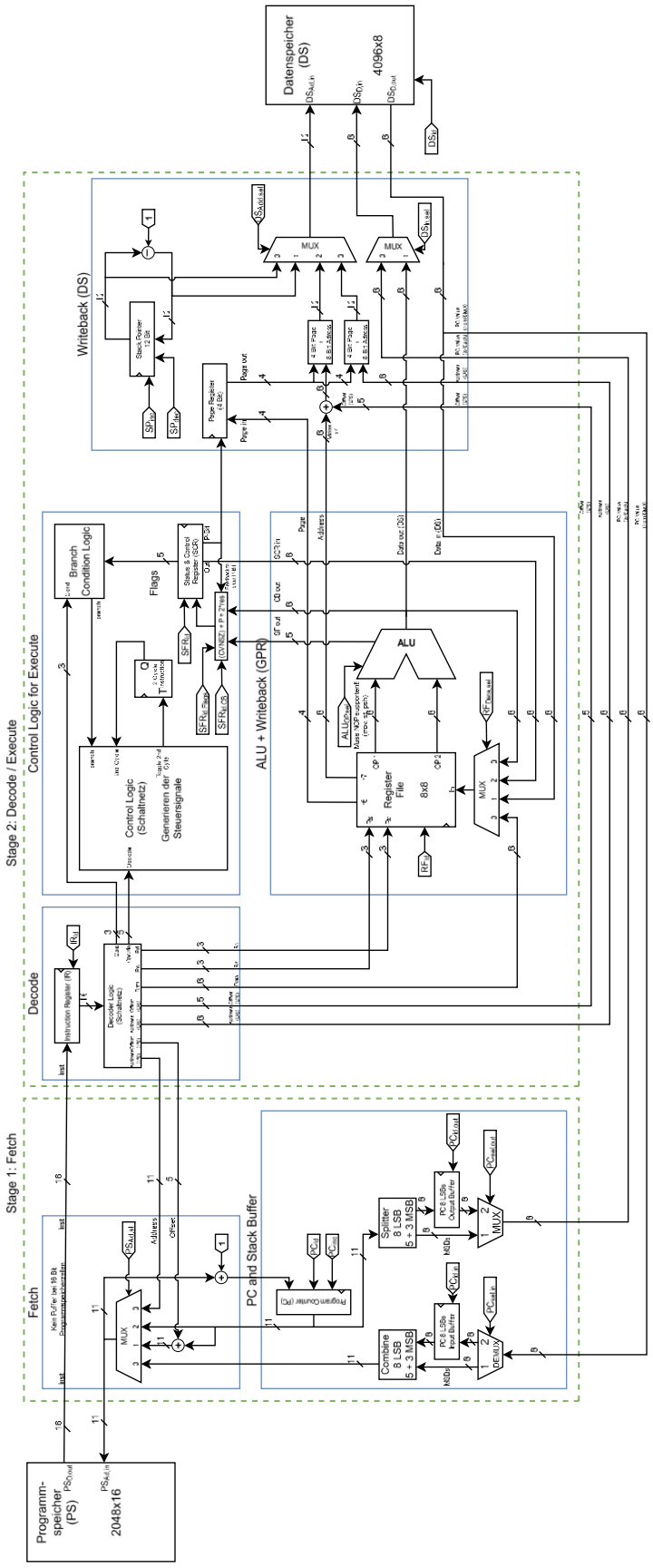
Anhang 2.1: Befehlsformate

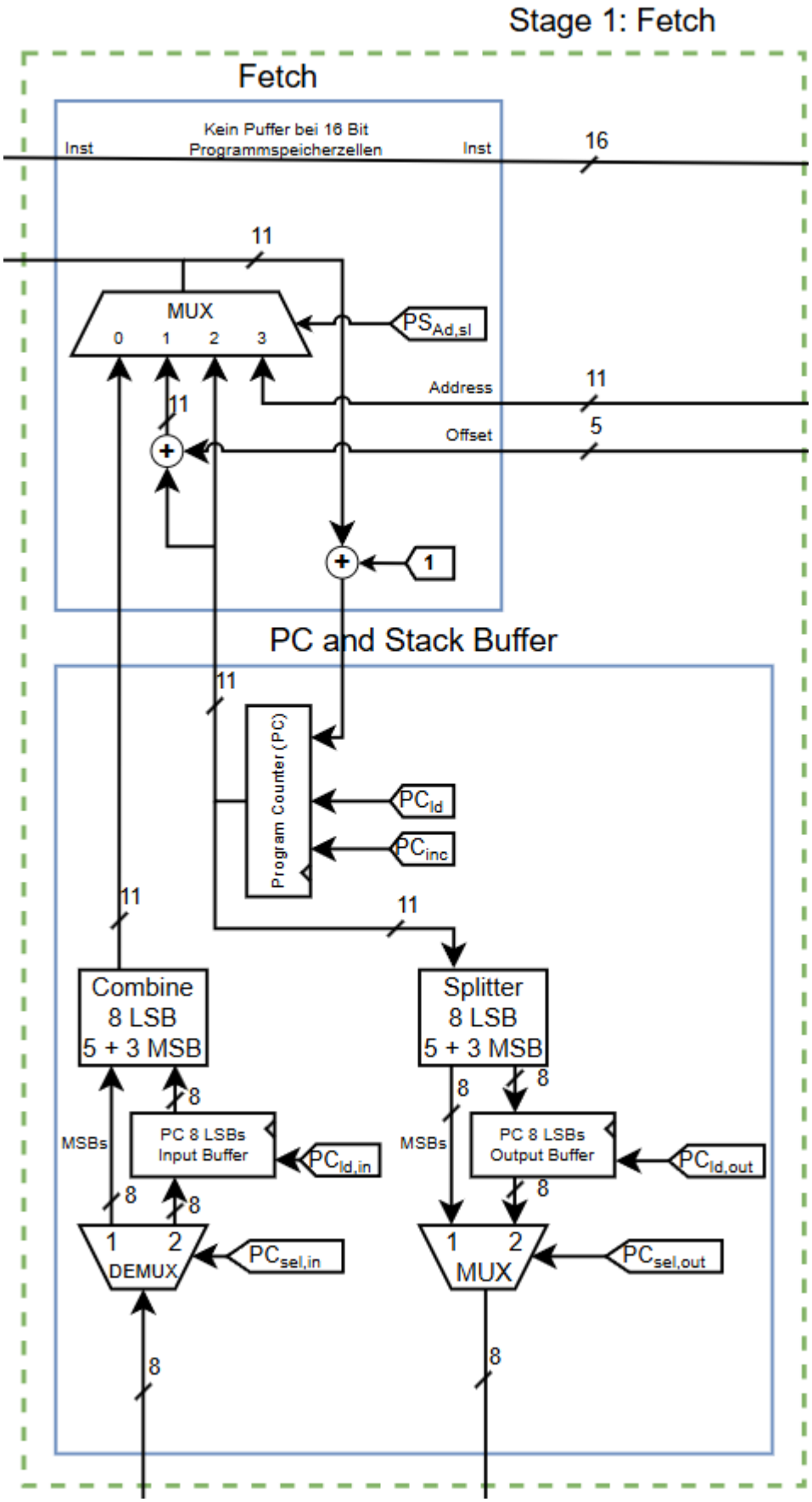
Befehlsgruppe	11	10	9	8	7	6	5	4	3	2	1	0			
Arithmetisc-logische Befehle	**Opcode**				**X**		**Rd**			**Rs**			Rd: Zielregister	Rs: Quellregister	! Rd ist auch Quellregister für Op1
Schiebe- und Rotationsbefehle	**Opcode**				**X**		**Rd**			**X**	**X**	**X**	Rd: Zielregister		! Rd ist auch Quellregister für Op1
Datenbewegungsbefehle															
Register-Register	**Opcode**				**X**		**Rd**			**Rs**			Rd: Zielregister	Rs: Quellregister	
Register-Speicher indirekt	**Opcode**				**X**		**Rd/Offset**			**Rs/Offset**			Rd: Zielregister	Rs: Quellregister	! Speicheradresse liegt in r6, r7. Offset wird zu r6/r7 addiert. R6 wird ignoriert wenn P=0
Register-Speicher direkt	**Opcode**				**E**		**Rd/Add0**			**Rs/Add0**			Rd: Zielregister	Rs: Quellregister	! Verwendung von Add0, wenn E=0
Register-Speicher direkt ext	**Opcode**				**Address**										! Verwendung von Address, wenn E=1. Doppelte Befehlsbreite
Immediate	**Opcode**				**E**		**Rd**			**Data0**			Rd: Zielregister		! Verwendung von Data0, wenn E=0
Immediate ext	**Data**														! Verwendung von Data wenn E=1. Doppelte Befehlsbreite
Programmsteuerbefehle															
unbedingte Sprungbefehle	**Opcode**				**E**		**Add0/Offset**						Add0: Sprungziel	Offset: rel. Sprung	! Verwendung von Add0, wenn E=0
unbedingte Sprungbefehle ext	**Address**												Address: Sprungziel		! Verwendung von Address wenn E=1. Doppelte Befehlsbreite
bedingte Sprungbefehle	**Opcode**				**Cond**					**Offset**			Cond: Sprungbedingung	Offset: rel. Sprung	Die Sprungbedingung wird auf der vorigen Tabelle definiert
Systemsteuerbefehle	**Opcode**				**X**		**Rd**			**Rs**					

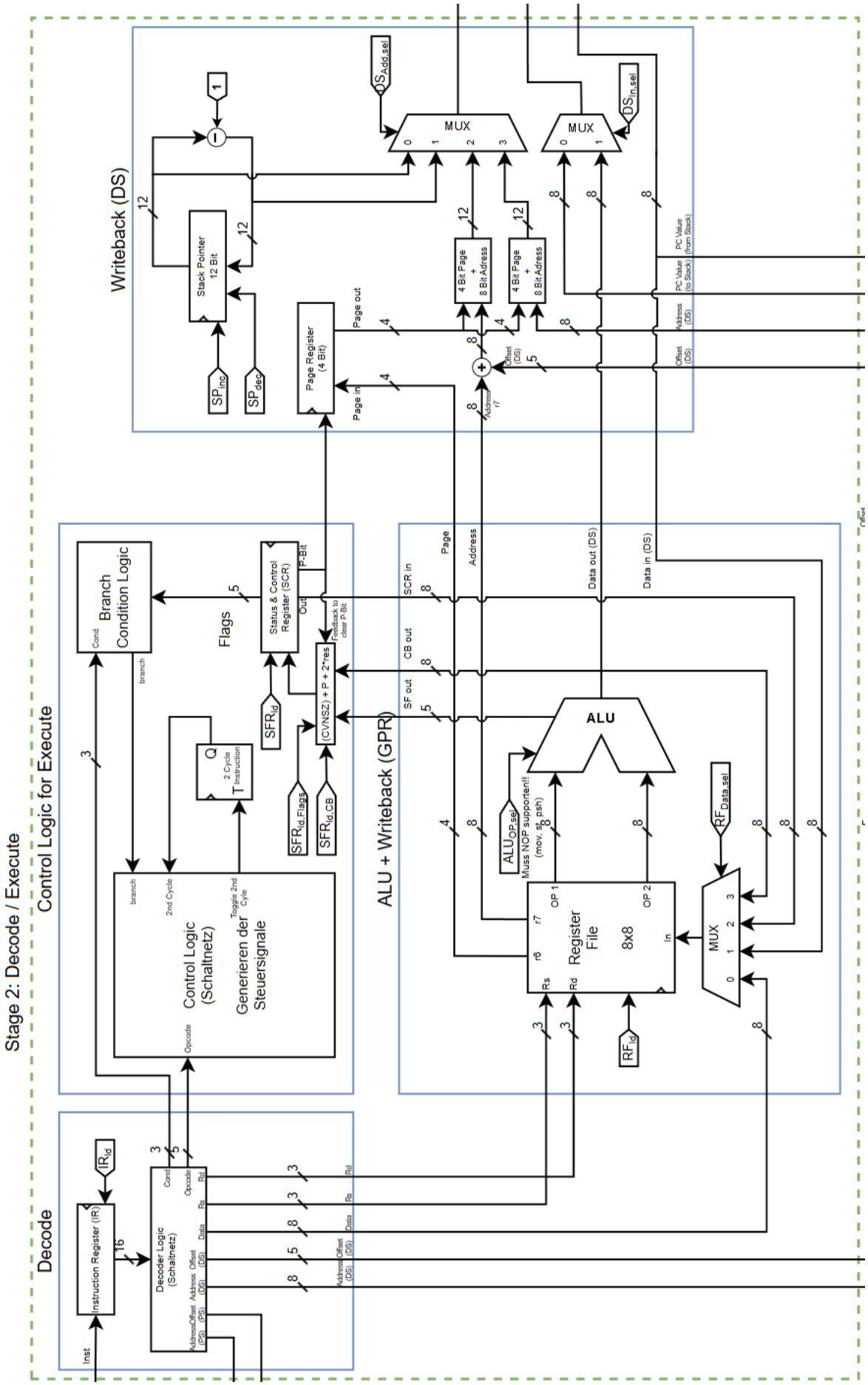
Anhang 2.2: Befehlsset

Mnemonik	Maschinenbefehl												Beschreibung
	Opcode					Operands							
nop	0	0	0	0	0	X	X	X	X	X	X	X	Keine Operation, Wartetakt
add	0	0	0	0	1	X	Rd			Rs			Rd <= Rd + Rs
addc	0	0	0	1	0	X	Rd			Rs			Rd <= Rd + Rs + C
sub	0	0	0	1	1	X	Rd			Rs			Rd <= Rd - Rs
subc	0	0	1	0	0	X	Rd			Rs			Rd <= Rd - Rs - C
inc	0	0	1	0	1	X	Rd			X	X	X	Rd <= Rd + 1
dec	0	0	1	1	0	X	Rd			X	X	X	Rd <= Rd - 1
and	0	0	1	1	1	X	Rd			Rs			Rd <= Rd AND Rs
or	0	1	0	0	0	X	Rd			Rs			Rd <= Rd OR Rs
xor	0	1	0	0	1	X	Rd			Rs			Rd <= Rd XOR Rs
not	0	1	0	1	0	X	Rd			X	X	X	Rd <= NOR Rd
sll	0	1	0	1	1	X	Rd			X	X	X	Rd <= Rd << 1
slr	0	1	1	0	0	X	Rd			X	X	X	Rd <= Rd >> 1
mov	0	1	1	0	1	X	Rd			Rs			Rd <= Rs
psh	0	1	1	1	0	X	X	X	X	Rs			MEM[SP] <= Rs; SP <= SP + 1
pll	0	1	1	1	1	X	Rd			X	X	X	Rd <= MEM[SP - 1]; SP <= SP - 1
stz	1	0	0	0	0	P	Offset			Rs			MEM[(r6 << 8 OR r7) + Offset] <= Rs ; ignore r6 IF P=0
ldz	1	0	0	0	1	P	Rd			Offset			Rd <= MEM[(r6 << 8 OR r7) + Offset] ; ignore r6 IF P=0
st	1	0	0	1	0	E	Add0			Rs			MEM[Add0] <= Rs IF E=0
	Address												MEM[Address] <= Rs IF E=1
ld	1	0	0	1	1	E	Rd			Add0			Rd <= MEM[Add0] IF E=0
	Address												Rd <= MEM[Address] IF E=1
ldi	1	0	1	0	0	E	Rd			Data0			Rd <= Data0 IF E=0
	Data								X	X	X	X	Rd <= Data IF E=1
jpa	1	0	1	0	1	X	X	X	X	X	X	X	PC <= Address
	Address												
bra	1	0	1	1	0	Offset							PC <= PC + Offset
brs	1	0	1	1	1	Cond			Offset				PC <= PC + Offset IF Cond=1
brc	1	1	0	0	0	Cond			Offset				PC <= PC + Offset IF Cond=0
jsr	1	1	0	0	1	E	Add0						MEM[SP] <= PC, SP <= SP + 1, PC <= Add0 IF E=0
	Address												MEM[SP] <= PC, SP <= SP + 1, PC <= Address IF E=1
ret	1	1	0	1	0	X	X	X	X	X	X	X	PC <= MEM[SP - 1], SP <= SP - 1
res0	1	1	0	1	1	X	X	X	X	X	X	X	nop
res1	1	1	1	0	0	X	X	X	X	X	X	X	nop
res2	1	1	1	0	1	X	X	X	X	X	X	X	nop
res3	1	1	1	1	0	X	X	X	X	X	X	X	nop
hlt	1	1	1	1	1	X	X	X	X	X	X	X	Alle Abläufe im Prozessor anhalten

Anhang 3: Mikroarchitektur des Prozessors




Stage 1: Fetch
Fetch
Kein Puffer bei 16 Bit Programmspeicherzellen
Inst
Inst
16
11
MUX
0 1 2 3
PS Ad,sl
Address
11
Offset
5
11
1
PC and Stack Buffer
11
Program Counter (PC)
PC ld
PC inc
11
11
Combine 8 LSB 5 + 3 MSB
Splitter 8 LSB 5 + 3 MSB
8
8
MSBs
PC 8 LSBs Input Buffer
PC ld,in
MSBs
PC 8 LSBs Output Buffer
PC ld,out
8
8
8
8
1 2
DEMUX
PC sel,in
1 2
MUX
PC sel,out
8
8


Stage 2: Decode / Execute
Decode
Control Logic for Execute
ALU + Writeback (GPR)
Writeback (DS)
Instruction Register (IR)
Decoder Logic (Schaltnetz)
Branch Condition Logic
Flags
Status & Control Register (SCR)
Control Logic (Schaltnetz)
Generieren der Steuersignale
ALU
Register File
8x8
MUX
Stack Pointer 12 Bit
Page Register (4 Bit)
4 Bit Page + 8 Bit Adress
Data in (DS)
Data out (DS)
Page
Address
SCR in
CB out
SF out
Page out
Page in
PC Value (to Stack)
PC Value (from Stack)
Address (DS)
Offset (DS)

Anhang 4: Zeitdiagramme

Anhang 4.1: Zeitdiagramme ALU-Befehle

ALU 2 Operanden				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	ADD r1, r2			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	ADD r1, r2		
Opcode		ADD		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd		r1		
Rs		r2		
RF_in		7		
r0	0	0	0	
r1	5	5	7	
r2	2	2	2	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1		5		
OP2		2		
ALU_out		7		
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in				
DS_D,out				
DS_D,in				

ALU 1 Operand				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	SLL r1			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	SLL r1		
Opcode		SLL		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd		r1		
Rs				
RF_in		10		
r0	0	0	0	
r1	5	5	10	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0		
OP1		5		
OP2				
ALU_out		10		
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in				
DS_D,out				
DS_D,in				

Anhang 4.2: Zeitdiagramme Datenbewegungsbefehle

mov (Register - Register)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	MOV r2, r0			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	MOV r2, r0		
Opcode		MOV		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd		r2		
Rs		r0		
RF_in		2		
r0	2	2	2	
r1	5	5	5	
r2	0	0	2	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1		0		
OP2		2		
ALU_out		2		
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in				
DS_D,out				
DS_D,in				

psh (Register - Stack)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	PSH r1			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	PSH		
Opcode				
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd				
Rs		r1		
RF_in				
r0	0	0	0	
r1	5	5	5	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2		5		
ALU_out		5		
Page_in				
PR	0	0	0	
SP	0	0	1	
DS_Add,in		0		
DS_D,out				
DS_D,in		5		
DS(0)			5	
DS(1)				

pll (Stack - Register)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	PLL r0			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	PLL r0		
Opcode		PLL		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd		r0		
Rs				
RF_in		5		
r0	0	0	5	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR	0	0	0	
SP	1	1	0	
DS_Add,in		0		
DS_D,out		5		
DS_D,in				
DS(0)	5	5	5	
DS(1)				

stz (Register - DS)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	STZ r0, 1			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	STZ r0, 1		
Opcode		STZ		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)		1		
Data				
Rd				
Rs		r0		
RF_in				
r0	7	7	7	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	29	29	29	
OP1				
OP2		7		
ALU_out		7		
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in		30		
DS_D,out				
DS_D,in		7		
DS(29)				
DS(30)			7	

ldz (DS - Register)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	LDZ r4, 2			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	LDZ r4, 2		
Opcode		LDZ		
Address (PS)				
Offset (PS)				
Address (DS)		2		
Offset (DS)				
Data				
Rd		r4		
Rs				
RF_in		3		
r0	0	0	0	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	3	
r5	0	0	0	
r6	0	0	0	
r7	29	29	29	
OP1				
OP2				
ALU_out				
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in		31		
DS_D,out		3		
DS_D,in				
DS(29)				
DS(30)				

st (Register - Datenspeicher)				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	ST 0, r2			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	ST 0, r2		
Opcode		ST		
Address (PS)				
Offset (PS)				
Address (DS)		0		
Offset (DS)				
Data				
Rd				
Rs		r2		
RF_in				
r0	0	0	0	
r1	0	0	0	
r2	46	46	46	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2		46		
ALU_out		46		
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in		0		
DS_D,out				
DS_D,in		46		
DS(0)			46	
DS(1)				

AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	AU
ld (Datenspeicher - Register)						**ldi (Immediate - Register)**				
Takt	0	1	2	3		Takt	0	1	2	3
PC	0	1	2			PC	0	1	2	
PC_in	1	2	3			PC_in	1	2	3	
PS_Add,in	0	1	2			PS_Add,in	0	1	2	
PS_D,out	LD r0, 1					PS_D,out	LDI r3, 7			
PC Buffer Out	0	0	0			PC Buffer Out	0	0	0	
PC to Stack						PC to Stack				
PC Buffer In	0	0	0			PC Buffer In	0	0	0	
PC from Stack						PC from Stack				
IR	---	LD r0, 1				IR	---	LDI r3, 7		
Opcode		LD				Opcode		LDI		
Address (PS)						Address (PS)				
Offset (PS)						Offset (PS)				
Address (DS)		1				Address (DS)				
Offset (DS)						Offset (DS)				
Data						Data		7		
Rd		r0				Rd		r3		
Rs						Rs				
RF_in		46				RF_in		7		
r0	0	0	46			r0	0	0	0	
r1	0	0	0			r1	0	0	0	
r2	0	0	0			r2	0	0	0	
r3	0	0	0			r3	0	0	7	
r4	0	0	0			r4	0	0	0	
r5	0	0	0			r5	0	0	0	
r6	0	0	0			r6	0	0	0	
r7	0	0	0			r7	0	0	0	
OP1						OP1				
OP2						OP2				
ALU_out						ALU_out				
Page_in						Page_in				
PR	0	0	0			PR	0	0	0	
SP	0	0	0			SP	0	0	0	
DS_Add,in		1				DS_Add,in				
DS_D,out		46				DS_D,out				
DS_D,in						DS_D,in				
DS(0)										
DS(1)	46	46	46							

Anhang 4.3: Zeitdiagramme Programmsteuerung

jpa				
Takt	0	1	2	3
PC	0	1	11	
PC_in	1	11	12	
PS_Add,in	0	10	11	
PS_D,out	JPA 10	PS(10)		
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR		JPA 10	PS(10)	
Opcode		JPA		
Address (PS)		10		
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd				
Rs				
RF_in				
r0	0	0	0	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR	0	0	0	0
SP	0	0	0	0
DS_Add,in				
DS_D,out				
DS_D,in				

bra				
Takt	0	1	2	3
PC	3	4	9	
PC_in	4	9	10	
PS_Add,in	3	8	9	
PS_D,out	BRA 5	PS(8)		
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR		BRA 5	PS(8)	
Opcode		BRA		
Address (PS)				
Offset (PS)		5 - 1		
Address (DS)				
Offset (DS)				
Data				
Rd				
Rs				
RF_in				
r0	0	0	0	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR	0	0	0	0
SP	0	0	0	0
DS_Add,in				
DS_D,out				
DS_D,in				

brbs / brbc (taken)				
Takt	0	1	2	3
PC	0	1	8	
PC_in	1	8	9	
PS_Add,in	0	7	8	
PS_D,out	BRBS Z, 7	PS(7)		
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR		BRBS Z, 7	PS(7)	
Opcode		BRBS		
Address (PS)				
Offset (PS)		7 - 1		
Address (DS)				
Offset (DS)				
Data				
Rd				
Rs				
RF_in				
r0	0	0	0	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR				
SP				
DS_Add,in				
DS_D,out				
DS_D,in				

brbs / brbc (not taken)				
Takt	0	1	2	3
PC	0	1	8	
PC_in	1	2	9	
PS_Add,in	0	1	8	
PS_D,out	BRBS Z, 7	PS(2)		
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR		BRBS Z, 7	PS(2)	
Opcode		BRBS		
Address (PS)				
Offset (PS)		7 - 1		
Address (DS)				
Offset (DS)				
Data				
Rd				
Rs				
RF_in				
r0	0	0	0	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR				
SP				
DS_Add,in				
DS_D,out				
DS_D,in				

call/ret							
Takt	0	1	2	3	4	5	6
PC	300	301	301	101	102	102	302
PC_in	301	101	101	102	103	302	303
PS_Add,in	300	100	100	101	102	301	302
PS_D,out	CALL 100	PS(100)	PS(100)	RET		PS(301)	
PC Buffer Out	0	0	0x2D	0x2D	0x2D	0x2D	0x2D
PC to Stack		0x01	0x2D				
PC Buffer In	0	0	0	0	0	0x2D	0x2D
PC from Stack					0x2D	0x01	
IR		CALL 100	CALL 100	PS(100)	RET	RET	PS(301)
Opcode		CALL	CALL	z.B. NOP	RET	RET	
Address (PS)		100	100				
Offset (PS)							
Address (DS)							
Offset (DS)							
Data							
Rd							
Rs							
RF_in							
r0	0	0	0	0	0	0	0
r1	0	0	0	0	0	0	0
r2	0	0	0	0	0	0	0
r3	0	0	0	0	0	0	0
r4	0	0	0	0	0	0	0
r5	0	0	0	0	0	0	0
r6	0	0	0	0	0	0	0
r7	0	0	0	0	0	0	0
OP1							
OP2							
ALU_out							
Page_in							
PR	0	0	0	0	0	0	0
SP	0	0	1	2	2	1	0
DS_Add,in		0	1		1	0	
DS_D,out					0x2C	0x01	
DS_D,in		0x01	0x2D				
DS(0)			0x01	0x01	0x01	0x01	0x01
DS(1)				0x2D	0x2D	0x2D	0x2D

Anhang 4.4: Systemsteuerung

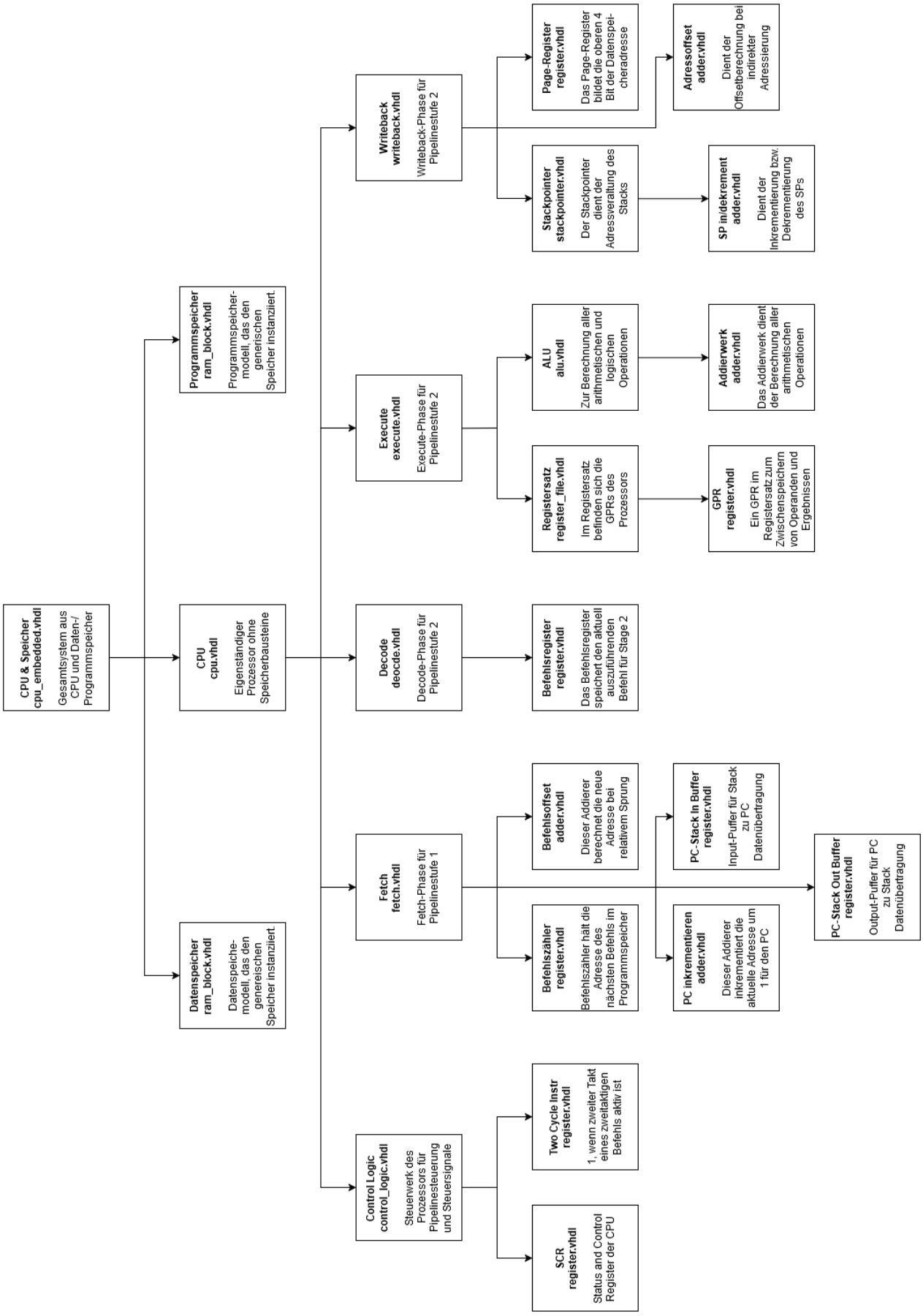
NOP				
Takt	0	1	2	3
PC	0	1	2	
PC_in	1	2	3	
PS_Add,in	0	1	2	
PS_D,out	NOP			
PC Buffer Out	0	0	0	
PC to Stack				
PC Buffer In	0	0	0	
PC from Stack				
IR	---	NOP		
Opcode		NOP		
Address (PS)				
Offset (PS)				
Address (DS)				
Offset (DS)				
Data				
Rd				
Rs				
RF_in				
r0	0	0	0	
r1	0	0	0	
r2	0	0	0	
r3	0	0	0	
r4	0	0	0	
r5	0	0	0	
r6	0	0	0	
r7	0	0	0	
OP1				
OP2				
ALU_out				
Page_in				
PR	0	0	0	
SP	0	0	0	
DS_Add,in				
DS_D,out				
DS_D,in				

Anhang 5: Steuerwort für alle Befehle

Steuerwort	Befehl	nop	add	addc	sub	subc	inc	dec	and	or	xor	not	sll	slr	mov	psh	pll	stz	ldz
	getestet?	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
PrStrAddrSel		"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"	"10"
Pcload		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PCldInBuf		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PCldOutBuf		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PCselOutBuf		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IRload		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
RegFileLoad		0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1
AluOpSel		"0000"	"0001"	"0010"	"0011"	"0100"	"0101"	"0110"	"0111"	"1000"	"1001"	"1010"	"1011"	"1100"	"0000"	"0000"	"0000"	"0000"	"0000"
RegFileDataSel		"00"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"11"	"00"	"01"	"00"	"01"
SPinc		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
SPdec		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
DataStrAddrSel		"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"01"	"10"	"10"
DataStrInSel		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
DataStrLoad		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
SFRldCB		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SFRldFlags		0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
set2ndCycle		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Steuerwort	Befehl	st	ld	ldi	jpa	bra	brs	brc	call.0	call.1	ret.0	ret.1	res1	res2	lcr	stcr	hlt
	getestet?	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
PrStrAddrSel		"10"	"10"	"10"	"11"	"01"	"01"	"01"	"11"	"11"	"00"	"00"	"10"	"10"	"10"	"10"	"00"
PcIoad		1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0
PCldInBuf		0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
PCldOutBuf		0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
PCselOutBuf		0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
IRload		1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0
RegFileLoad		0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0
AluOpSel		"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"	"0000"
RegFileDataSel		"00"	"01"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"10"	"00"	"00"
SPinc		0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
SPdec		0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
DataStrAddrSel		"11"	"11"	"00"	"00"	"00"	"00"	"00"	"00"	"00"	"01"	"01"	"00"	"00"	"00"	"00"	"00"
DataStrInSel		1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DataStrLoad		1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
SFRldCB		0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
SFRldFlags		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
set2ndCycle		0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Anhang 6: Struktur der VHDL-Implementierung



Anhang 7: Zeitdiagramm der Steuersignalgenerierung des Steuerwerks