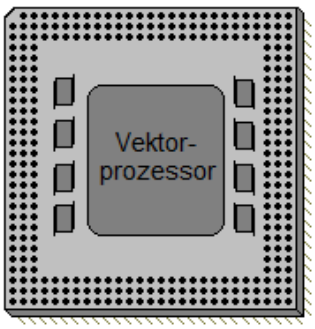

Diplomarbeit

Entwicklung eines konfigurierbaren Vektorprozessors

Harald Manske
Fachhochschule Augsburg

19. Dezember 2007

Erstellungserklärung

Hiermit erkläre ich, dass die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet wurden.

Abstract

When it comes to personal computers, processors with SIMD-extensions are already widely spread. With their help it is possible to considerably increase the performance of processing multimedia-based data compared to common, scalar-only architectures. With a processor like that, one single command can be applied to multiple operands, located in vector-registers. A vector-register contains several independent data words.

Since demand for increased performance is rising rapidly even in embedded systems, vector-processors or processors with vector-extensions become useful that field, too. This diploma thesis describes the development of a vector-processor as a configurable soft-core which can be integrated flexibly (e.g. in FPGAs) in embedded applications.

Written in VHDL, the processor is based on a simple accumulator-architecture, working together with a vector-unit. The instruction coding is designed to allow specifying a scalar and a vector command within a single 32-bit-instruction-word. Both commands will be executed either simultaneously or in cooperation of the two units.

To allow the processor to be tailored for the needs of different applications, the amount of datawords within one vector-register as well as the amount of vector-registers are configurable. The access to the memory, shared by the two units, is handled by a memory interface which is controlled and addressed by the scalar-unit.

Besides the processor itself, an assembler as well as a simple on-chip debugging system is implemented.

As final step, the performance of the processor is evaluated and compared to CPUs of typical personal computers and a Motorola 68000.

Kurzfassung

In Heim-PCs sind Prozessoren mit Vektorerweiterungen bereits weit verbreitet. Mit ihrer Hilfe können Berechnungen wie die Aufbereitung von multimedialen Daten im Vergleich zu herkömmlichen, skalaren Architekturen wesentlich beschleunigt werden. Ein einzelner Befehl kann in einem solchen Prozessor auf mehrere Operanden wirken, die sich in mehrere Wortlängen umfassenden Vektorregistern befinden.

Da auch in eingebetteten Systemen nach und nach mehr Rechenleistung gefordert wird, liegt es nahe, Vektorprozessoren oder Prozessoren mit Vektorerweiterungen auch in diesem Umfeld zu verwenden. Diese Diplomarbeit behandelt die Entwicklung eines solchen Prozessors, der sich als konfigurierbarer Soft-Core flexibel (z. B. in FPGAs) einsetzen lässt und sich damit zur Anwendung in eingebetteten Systemen eignet.

Der als VHDL-Modell entwickelte Prozessor besteht aus einer einfachen, skalaren Akkumulator-Architektur, welche mit einer Vektoreinheit zusammenarbeitet. Die Befehlskodierung ist so gewählt, dass ein 32-Bit-Befehl aus je einer Komponente für die Skalar- und für die Vektoreinheit besteht. Diese beiden Teilbefehle werden vom Prozessor entweder gleichzeitig oder in Kooperation der zwei Einheiten bearbeitet.

Um den Prozessor auf die Bedürfnisse der Anwendung anpassen zu können, kann die Anzahl an 32-Bit-Wörtern pro Vektorregister sowie die Anzahl an Vektorregistern konfiguriert werden. Der Zugriff auf den gemeinsamen Speicher geschieht über eine Speicherschnittstelle und wird über die Skalareinheit adressiert und gesteuert.

Neben dem Prozessor selbst werden auch ein Assembler sowie ein einfacher Debugger entwickelt.

Die anschließende Leistungsbewertung des in Betrieb genommenen Prozessors vergleicht dessen Performanz mit aktuellen Desktop-Prozessoren und einem Motorola 68000.

Inhaltsverzeichnis

1 Einleitung

1.1 Überblick

1.1.1 Multimediale Anwendungen

Die Möglichkeit, Bilder und Filme überall aufnehmen zu können, wird von immer mehr Menschen genutzt. Digitale Kameras dominieren mittlerweile den Markt und selbst Mobiltelefone sind in der Lage, Bilder mit bis zu fünf Megapixeln einzufangen. Dabei ist die Aufnahme allerdings nur der erste Schritt, die entstehenden Mediendaten müssen auch komprimiert und abgespeichert werden. Oft findet auch eine visuelle Verbesserung des Bildes durch Bildverarbeitungsalgorithmen statt.

Um diese Schritte ohne große Verzögerungen durchführen zu können, muss genug Rechenleistung zur Verfügung stehen. Ein Prozessor, der die Daten rein sequentiell bearbeiten kann, benötigt dafür eine sehr hohe Taktfrequenz. Hohe Taktfrequenzen resultieren allerdings auch in hoher Leistungsaufnahme, ein Umstand der in mobilen Anwendungen wegen der Abhängigkeit von Strom aus Batterien äußerst ungünstig ist.

Stattdessen bietet sich ein anderer Ansatz an. Die für jeden Bildpunkt notwendigen Berechnungen sind weitgehend voneinander unabhängig, können also parallel durchgeführt werden. Ein Prozessor, der diesen Umstand ausnutzt und die Werte mehrerer Bildpunkte gleichzeitig berechnet, benötigt keine so hohen Taktfrequenzen und spart damit Energie.

Dieses Prinzip ist nicht nur auf digitale Kameras beschränkt, sondern lässt sich in vielen Anwendungen umsetzen, in denen Mediendaten, seien es Bilder, Filme oder auch Ton, verarbeitet werden sollen. Oft können mit Hilfe solcher Prozessoren, auch mathematische Probleme (z. B. Berechnungen mit Matrizen) sehr effizient gelöst werden.

1.1.2 Hardwareentwicklung und Open Source

Im Bereich der Hardwareentwicklung wird zunehmend programmierbare Hardware wie z. B. FPGAs eingesetzt. Es ist auch der Trend zu beobachten, dass komplette Systeme auf einem einzigen Chip realisiert werden. Dies geschieht, indem man nach einem Baukastenprinzip passende IP-Cores (wiederverwendbare Beschreibungen von Halbleiterbauteilen) auswählt und miteinander verbindet.

Solche IP-Cores können zum einen von kommerziellen IP-Herstellern erworben oder selbst entwickelt werden. Zum anderen gibt es aber auch die

Möglichkeit, fertige IP-Cores zu verwenden, die unter einer Open-Source-Lizenz veröffentlicht wurden. Hierfür gibt es im Internet bereits etablierte Webseiten, auf denen eine Vielzahl verschiedener Bausteine zum Download angeboten wird. Das Spektrum reicht von Multiplizierern über Ethernet-Schnittstellen bis hin zu Mikroprozessoren, die teilweise schon als komplettes System mit Speicher, Controllern und vielen Schnittstellen vorliegen.

Was man in diesem Bereich allerdings momentan noch vergeblich sucht, sind für multimediale Anwendungen geeignete SIMD-Prozessoren.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung eines SIMD-Prozessors für eingebettete Anwendungen, welcher folgende Eigenschaften aufweist und in der Lage ist, sehr viele Daten parallel zu verarbeiten:

- Der Prozessor lässt sich flexibel als Soft-Core z. B. in FPGAs einsetzen.
- Die Befehlskodierung und der Befehlssatz sind erweiterbar ausgelegt.
- Die Menge an Daten, die parallel verarbeitet werden kann, ist konfigurierbar.
- Teile der Funktionalität können ein- bzw. ausgeschaltet werden, um ihn an verschiedene Anwendungen anpassen zu können.

Der Prozessor muss validiert werden und in realer Hardware, beispielsweise einem FPGA, lauffähig sein. Hierfür wird eine Testumgebung in Form von Speicher und einer On-Chip-Debugging-Einheit benötigt.

Die Entwicklung eines Assemblers und der Debugging-Software für den Entwicklungsrechner ist ebenfalls Teil der Aufgabe.

Zusätzlich dazu muss die Leistungsfähigkeit des Prozessors anhand einer Beispielanwendung ermittelt werden.

1.3 Aufbau des Dokuments

Diese Arbeit ist in sieben Kapitel unterteilt. Kapitel 1 führt zum Thema hin, beschreibt das Ziel der Arbeit und gibt einen Überblick über den Aufbau des Dokuments. Kapitel 2 behandelt die Grundlagen, die für das weitere Verständnis notwendig sind. Außerdem wird hier der aktuelle Stand der Technik, was Vektor-Erweiterungen angeht, beschrieben. Kapitel 3 befasst sich mit dem Entwurf des Prozessors. Es werden Grundprinzipien, Befehlssatz, Mikroarchitektur und vorhandene Schnittstellen erläutert. Details zur Implementierung sind in Kapitel 4 zu finden. Hier geht es um Optimierung,

welche Komponenten bei der Implementierung Probleme bereitet haben und wie diese gelöst wurden. Kapitel 5 beschreibt die Umgebung, in der der Prozessor eingebunden wurde sowie die enstandenen Entwicklungswerkzeuge. Auch die Validierung des Gesamtsystems wird hier erläutert. Die Ergebnisse der durchgeführten Leistungsanalyse und die Auswirkungen, die die verschiedenen Parameter der Konfiguration auf die Synthese haben, sind in Kapitel 6 zu finden. Abgeschlossen wird die Arbeit durch das Kapitel 7. Die Ergebnisse der Entwicklung werden zusammengefasst und noch nicht umgesetzte Ideen für den Prozessor vorgestellt.

2 Grundlagen

2.1 Programmierbare Hardware

Wenn in der Elektronikbranche ein digitales System entwickelt wird, stehen zwei relevante Herstellungsprozesse zur Auswahl. Es kann auf geeignete Standardbausteine (z. B. Mikroprozessoren, Speicherbausteine) zurückgegriffen und diese programmiert, aber auch eine eigene Anwendungsspezifische Integrierte Schaltungen (ASICs) hergestellt werden. Da Anforderungen, zum Beispiel an die Geschwindigkeit oder den Stromverbrauch eines Systems, den Einsatz von Standardkomponenten nicht immer zulassen, ist der Entwurf von eigenen, integrierten Schaltungen in vielen Fällen unvermeidlich [Sie01].

Die Verwendung von ASICs bietet sich vor allem für Geräte an, die in hohen Stückzahlen abgesetzt werden können. Der hohe Entwicklungsaufwand kann dann durch die Kostenersparnis bei der Fertigung der einzelnen Chips ausgeglichen werden.

Für Geräte, die in kleinen Serien gefertigt werden, gibt es allerdings auch Alternativen. Programmierbare Hardware wie CPLDs oder FPGAs erlauben die schnelle und kostengünstige Entwicklung von anwendungsspezifischen Schaltungen, was allerdings mit einem wesentlich höheren Preis für den einzelnen Chip erkauft wird.

FPGA steht für „Field Programmable Gate Array“. Ein solcher Baustein hat nicht wie ein ASIC ab seiner Fertigung eine feste Funktion, sondern kann jederzeit neu konfiguriert werden. Damit lassen sich die verschiedenartigsten Schaltungen, von einem einfachen Addierer bis zu einem kompletten Computersystem (System-On-a-Programmable-Chip), realisieren. Wegen der Rekonfigurierbarkeit eignen sich FPGAs sehr gut in Systemen, an denen ohne Anpassung der Hardware Veränderungen vorgenommen werden sollen. Dadurch lassen sich z. B. die Funktionalität nachträglich erweitern bzw. ändern oder Fehler in bereits ausgelieferten Geräten beheben. Zusätzlich kann der selbe Chip für mehrere verschiedene Aufgaben verwendet, sowie die Entwicklung neuer Anwendungen, durch die Möglichkeit jederzeit testen zu können, wesentlich vereinfacht werden.

Die Konfiguration für FPGAs wird im Normalfall mit Hilfe einer Hardwarebeschreibungssprache wie VHDL oder Verilog erstellt. Seltener wird auch die Möglichkeit genutzt, Schaltungen durch einen graphischen Editor oder auf Basis der Programmiersprache C zu definieren.

Ein FPGA basiert hauptsächlich auf vielen programmierbaren Logikblöcken, welche gewöhnlich aus Look-Up-Tables (LUT), einem oder mehreren Flipflops und Multiplexern bestehen (Abbildung 1). Diese Blöcke werden dann

durch die vom Programmierer definierte Konfiguration miteinander verknüpft, um die gewünschte Funktionalität zu erhalten. Aktuelle FPGAs bieten dem Anwender bis zu mehrere zehntausend Logikblöcke. Die Konfiguration wird überlicherweise durch SRAM-Speicherzellen realisiert und bei jedem Bootvorgang des FPGAs neu geladen.

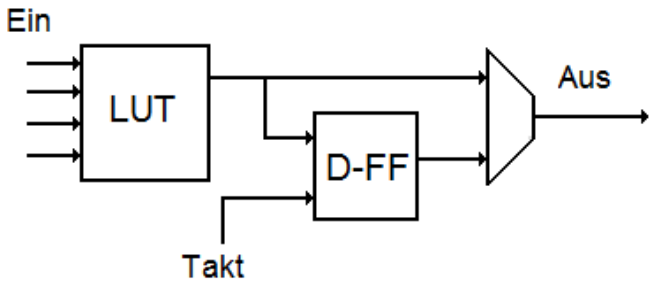

Abbildung 1: Vereinfachte Darstellung eines Logikblocks

Eine LUT definiert eine kombinatorische Funktion (NAND, AND, OR, XOR, Multiplexer etc.) aus mehreren Eingangssignalen. Die Anzahl an zur Verfügung stehenden Eingängen hängt hierbei vom verwendeten FPGA ab. Gängige Werte sind zwischen 4 und 6 Eingängen pro LUT. Reichen die Eingänge einer LUT für eine Funktion nicht aus, können mehrere LUTs direkt miteinander verschaltet werden.

Flipflops hingegen dienen der Speicherung von Signalen. Jedes Flipflop kann den Wert von einem Bit sichern, damit darauf in den folgenden Takten wieder zugegriffen werden kann.

Neben Logikblöcken können FPGAs, je nach Typ, auch noch weitere Komponenten enthalten.

Ein- und Ausgangs-Blöcke (IO-Blöcke) dienen als Schnittstelle des FPGAs zur Außenwelt, inklusive verschiedener TTL-Pegel, Pullup-Widerstände und verschiedener Signalstandards.

RAM-Blöcke erlauben die Speicherung von größeren Datenmengen, wodurch in vielen Situationen Flipflops und damit auch Logikblöcke eingespart werden können. Dieser, oft dualport-fähige RAM, ist in mehreren kleinen Blöcken organisiert und lässt sich dadurch flexibel zu größeren Einheiten zusammenschalten.

Multiplizierer erlauben die besonders schnelle Ausführung der Multiplikations-Operation, ohne dafür Logikzellen des FPGAs zu belegen. Dies ist besonders interessant in der digitalen Signalverarbeitung.

Um das Taktsignal synchron auf dem FPGA zur Verfügung zu stellen, dessen Frequenz zu teilen bzw. vervielfachen oder die Phase zu ändern, bieten FPGAs im Normalfall auch Elemente zur Taktaufbereitung.

Die Frage nach der Geschwindigkeit eines FPGAs ist nicht einfach zu beantworten. Zwar können FPGAs mit maximalen Systemtaktfrequenzen von mehreren hundert MHz verbaut werden, die tatsächliche Taktfrequenz der Schaltung ergibt sich aber zu großen Teilen aus dem Design der auf dem FPGA angewandten Konfiguration. Im Allgemeinen sind FPGAs deutlich langsamer als ASICs mit einer vergleichbaren Schaltung.

2.2 VHDL

Mit der Hardwarebeschreibungssprache VHDL (kurz für Very High Speed Integrated Circuit Hardware Description Language) können komplexe Schaltungen auf einem hohen Abstraktionsniveau entworfen werden. Mit dieser Sprache beschriebene Schaltungen lassen sich durch Software-Werkzeuge simulieren, verifizieren und in eine Netzliste umsetzen (Synthese).

Die Netzliste einer Schaltung beschreibt die Verbindungen zwischen den einzelnen Komponenten. Im Falle einer Netzliste für einen FPGA sind das die Verbindungen zwischen den Logik-, RAM-, IO-Blöcken oder Ähnlichem.

Um diese Netzliste zu erstellen wird ein Syntheseprogramm sowie die Bibliothek des Chipherstellers benötigt. VHDL selbst ist technologieunabhängig. Das bedeutet, eine mit VHDL entwickelte Schaltung kann auf verschiedene Weisen realisiert werden. Dies beinhaltet das Konfigurieren programmierbarer Logikbausteine wie FPGAs oder CPLDs, aber auch die Fertigung von ASICs mit Standardzellen sowie Gate Arrays.

Die Sprache VHDL macht es möglich eine Schaltung hierarchisch aufzubauen. Das folgendes Beispiel soll zeigen, wie aus vielen einzelnen kleinen Komponenten ein kompletter Prozessor zusammengebaut werden kann.

Mit Hilfe einiger Volladdierer wird ein Addiernetz gebildet. Dieses wird dann mit einem Multiplizierer und einer Schiebelogik zu einer arithmetisch-logischen Einheit (ALU) zusammengesetzt. Aus der ALU, mehreren Registern, Speicher und einem Steuerwerk entsteht dann der komplette Prozessor. Hierbei ist zu beachten, dass die einzelnen Komponenten (z. B. Register) wiederum aus kleineren Teilkomponenten bestehen können.

Die Schnittstelle einer Komponente (Entity) nach außen wird durch deren Port festgelegt. Dabei handelt es sich um eine Liste mit Ein- und Ausgabesignalen. Das Innenleben hingegen (Architecture) wird mit Hilfe von Sprachkonstrukten wie Nebenläufigkeit, sequentielle Prozessen und durch

das Einbinden anderer Komponenten beschrieben. Aber auch Funktionen und Prozeduren können verwendet werden.

In VHDL werden Zuweisungen von Signalen, im Gegensatz zu Zuweisungen in herkömmlichen Programmiersprachen wie C, nebenläufig (Abbildung 2) ausgeführt.

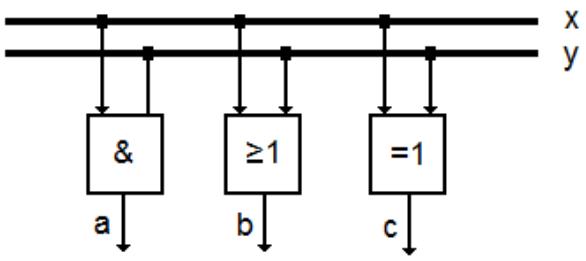

Abbildung 2: Nebenläufige Zuweisung in Hardware

Beispiel: Nebenläufige Zuweisung von a, b und c

```
architecture rtl of beispiel is
begin
    a <= x and y;
    b <= x or  y;
    c <= x xor y;
end rtl;
```

Zusätzlich gibt es die Möglichkeit, Prozesse zu definieren, in denen Anweisungen sequentiell abgearbeitet werden. Dies ermöglicht z. B. die Verwendung der Kontrollstrukturen *if*, *case*, Schleifen und die Erzeugung von Flipflops. Ein Prozess hat entweder eine Sensitivitätsliste und wird nur nach Änderung von darin aufgeführten Signalen gestartet, oder er wartet bis ein Ereignis, wie z. B. eine steigende Taktflanke, aufgetreten ist. Es können auch innerhalb einer Komponente mehrere nebenläufige Prozesse existieren. Oft stehen diese Prozesse auch miteinander in Verbindung und synchronisieren sich mit Hilfe von Signalen.

Beispiel: Vergleicher

```
architecture rtl of beispiel is
begin
    process(x,y)
    begin
        if x = y then
            z <= '1';
```

```
        else
            z <= '0';
        end if;
    end process;
end rtl;
```

VHDL erlaubt es, bereits existierende Komponenten in eine andere Komponente zu integrieren. Dazu wird eine Instanz der verwendeten Teilschaltung erzeugt und deren Ein- und Ausgabesignale mit Signalen der einbindenden Komponente verbunden. Auf diese Weise kann in VHDL der hierarchische Aufbau einer Schaltung realisiert werden.

Beispiel: Einbinden eines Halbaddierers

```
architecture struktur of beispiel is
    -- schnittstelle:
    component halbaddierer
        port(
            x: in std_logic;
            y: in std_logic;
            s: out std_logic;
            c: out std_logic
        );
    end component;

    -- definition von signalen
    signal x: std_logic;
    signal y: std_logic;
    signal s: std_logic;
    signal c: std_logic;
begin
    -- verbinden der komponente mit den signalen
    ha: halbaddierer port map (x => x, y => y, s => s, c => c);
end struktur;
```

Ein wichtiges Merkmal von VHDL ist es, dass damit in dieser Sprache entwickelte Komponenten validiert werden können. Dazu sind Anweisungen vorhanden, die zwar nicht synthetisiert werden können, aber von Simulatoren verstanden werden. Beispielsweise vergleicht die Assert-Anweisung den Wert eines Signals mit einem in Programmtext vorgegebenen Referenzwert und kann bei einer Abweichung Warnungen ausgeben oder sogar die Simulation stoppen.

Um eine Komponente zu validieren, wird im Regelfall eine sogenannte Testbench erstellt. Dabei handelt es sich um eine VDHL Komponente, welche das zu validierende Modul in sich einbindet. Innerhalb der Testbench werden dann die Eingangssignale der zu testenden Komponente mit Werten belegt

und deren Ausgangssignale per Assert-Anweisung überprüft. Bei einer Abweichung des Überprüften von dem Referenzwert, arbeitet die Komponente nicht nach Spezifikation.

2.3 SIMD- und Vektorprozessoren

Die Rechnerklassifikation nach Flynn [Fly72] unterteilt Architekturen anhand ihrer Anzahl an parallelen Befehls- und Datenströmen in vier Gruppen:

SISD - Single instruction stream, single data stream
SIMD - Single instruction stream, multiple data streams
MISD - Multiple instruction streams, single data stream
MISM - Multiple instruction streams, multiple data streams

Vetreter von SISD-Systemen sind Architekturen, bei denen ein einzelnes Leit- bzw. Steuerwerk genau ein Rechen- bzw. Operationswerk verwaltet. [Mär94] Die Operationen werden sequentiell abgearbeitet; in jedem Befehl kann also nur ein Operandenpaar verarbeitet werden. In Personal-Computern wurden bis zur Einführung des Intel Pentium MMX Prozessors, hauptsächlich reine SISD-Architekturen eingesetzt und in eingebetteten Systemen sind diese immer noch weit verbreitet.

SIMD-Rechner unterscheiden sich von SISD-Architekturen darin, dass das Steuerwerk anstatt einem mehrere Operationswerke mit Befehlen versorgt [Mär03]. Dadurch kann der selbe Befehl auf mehreren parallelen Operationswerken gleichzeitig ausgeführt und somit können mehrere Operandenpaare gleichzeitig verarbeitet werden. Unter die Kategorie SIMD fallen Feldrechner sowie Prozessoren mit Multimedia- bzw. Vektorerweiterungen.

Die beiden Kategorieren MISD und MISM des Flynn'schen Schemas haben für das weitere Verständnis dieser Arbeit keine Relevanz. Sie unterscheiden sich von den beiden bereits vorgestellten Architekturen dadurch, dass mehrere Befehlsströme vorhanden sind.

Der Begriff „Vektorprozessor“ wird mit zwei verschiedenen Bedeutungen verwendet: Zum einen als einen Rechnertyp mit pipelineartig aufgebauten Rechenwerk/en zur Verarbeitung von Vektoren an Daten [Mär94], zum anderen aber auch gelegentlich als allgemeinen Überbegriff für SIMD Systeme, welche Operationen auf Vektoren von Daten erlauben. Im Folgenden wird „Vektorprozessor“ und „Vektorrechner“ als Überbegriff für solche Systeme und „Vektorerweiterung“ als Synonym für SIMD-Erweiterung verwendet.

Bei dem in dieser Arbeit entwickelten Prozessor handelt es sich um eine skalare (SISD) Architektur mit Vektorerweiterung. Diese Erweiterung erlaubt es, einen einzigen Befehl auf mehrere Daten gleichzeitig anzuwenden. Das

parallel aufgebaute Operationswerk wirkt, anstatt auf einzelne Operanden, auf einen kompletten Vektor von Daten.

Jeder dieser Vektoren umfasst mehrere Daten-Wörter, deren Länge in vielen der heute üblichen SIMD-Prozessoren im Befehl kodierbar ist. Ein 128-Bit-Vektorregister könnte also beispielsweise in zwei 64-Bit-, vier 32-Bit-, acht 16-Bit- oder sechzehn 8-Bit-Wörter geteilt werden.

Die Abarbeitung einer Operation erfolgt für jedes Datum im Vektorregister unter Einbeziehung der Wortlänge parallel (Abbildung 3); dadurch ergeben sich ernome Leistungssteigerungen gegenüber skalaren Architekturen, welche jedes Datum in einem einzelnen Befehl bearbeiten müssen. Bei einem acht Wörter umfassenden Vektorregister können mit einem einzigen Additionsbefehl bis zu sieben Befehle eingespart werden. Dadurch wird eine Beschleunigung um bis zu Acht erreicht.

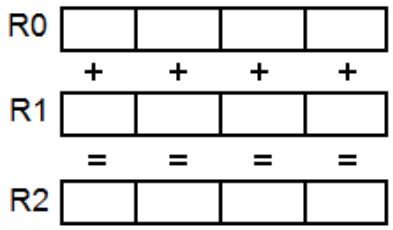

Abbildung 3: Addition von vier Wörtern umfassenden Vektorregistern

Der Gewinn an Laufzeit kann allerdings noch wesentlich größer sein. Oftmals werden für die Abarbeitung solcher Operationen in skalaren Prozessoren Schleifen verwendet um Programmspeicher und Programmieraufwand einzusparen. Dies resultiert in zusätzlichen Befehlen für Inkrementierungen, Vergleichsoperationen und bedingte Sprünge. Außerdem müssen die Operanden einzeln geladen und im Speicher abgelegt werden, während Vektorprozessoren (paralleler Speicherzugriff vorausgesetzt) dies ebenfalls in einem Befehl erledigen können.

Anhand eines Beispiels soll verdeutlicht werden, wie viele Befehle durch die Verwendung eines Vektorprozessors eingespart werden können. Es handelt sich dabei um die Addition von zwei, 16 Wörtern umfassenden Arrays. Der folgende C-Programmcode wird für den in dieser Diplomarbeit entwickelten Prozessor in zwei Ausführungen übersetzt. Zum einen in der Variante, die nur die Skalareinheit benutzt, zum anderen optimiert für die Vektoreinheit.

Algorithmus als C-Programm:

```
for(i=0; i<16; i++)
```

```
{
    res[i] = op1[i] + op2[i];
}
```

Algorithmus unter Verwendung der Skalareinheit:

```
        OR      Y, 0, 0       ; i=0
LOOP:   LD      A, [Y + OP1]  ; lade op1[i] in A
        LD      X, [Y + OP2]  ; lade op2[i] in X
        ADD     A, A, X       ; addiere A und X in A
        ST      [Y + RES], A  ; schreibe res[i]
        INC     Y, Y          ; i++
        SUB     0, Y, 16      ; vergleiche i mit 16
        JNZ     [0 + LOOP]    ; springe, wenn i < 16
```

Algorithmus unter Verwendung der Vektoreinheit:

```
        OR      A, 0, OP1     ; lade Adresse op1 in A
        OR      X, 0, OP2     ; lade Adresse op2 in X
        OR      Y, 0, RES     ; lade Adresse res in Y
        VLD     R0, [0 + A]   ; lade op1 in R0
        VLD     R1, [0 + X]   ; lade op2 in R1
        VADD.DW R2, R0, R1    ; addiere R0 und R1 in R2
        VST     [0 + Y], R2   ; schreibe res
```

Für die skalare Variante werden 113 Befehle zur Ausführung benötigt, während das Pendant unter Verwendung der Vektoreinheit mit lediglich sieben auskommt. Es ist also eine erhebliche Reduzierung an Befehlen beobachtbar, die sich in deutlich weniger Rechenzeit bemerkbar macht. In vielen Fällen kann auch Programmspeicher eingespart werden, da weniger Befehle kodiert werden müssen [Men05]. Leider konnte das im Beispiel kaum gezeigt werden, da die Vektoreinheit dieses Prozessors keine Direktwerte für Vektor-Operationen vorsieht. Dadurch wurden drei zusätzliche Befehle zum Laden der Adressen in Register notwendig.

Dieses Beispiel hat verdeutlicht, dass sich die Vektoreinheit schon bei sehr einfachen Problemen anwenden lässt. Es gibt aber auch bei komplexeren Problemen und insbesondere bei der Verarbeitung von Mediendaten viele Einsatzmöglichkeiten. In der Bildverarbeitung werden häufig Filter auf Bilder angewandt, um diese beispielsweise zu glätten oder Kanten hervorzuheben. Für eine Glättung (mit einem Box-Filter) müssen für jeden einzelnen Pixel dessen benachbarte Pixel aufaddiert und das Resultat durch den Faktor 9 geteilt werden, um einen Mittelwert zu bestimmen [Gon02]. Dieser Mittelwert ist dann der neue Wert des betrachteten Pixels. Hier können mit Vektorprozessoren hohe Leistungssteigerungen bei der Verarbeitung erzielt

werden, indem die Mittelwerte für mehrere Pixel gleichzeitig berechnet werden. Der Leistungsgewinn wird hierbei über die Länge des Vektorregisters limitiert. Je mehr Wörter in eines der Register passen, umso schneller wird die Verarbeitung.

2.4 Referenz-Architekturen

Prozessoren werden heutzutage überlicherweise nach dem von-Neumann-Referenzmodell aufgebaut. In diesem Modell befinden sich Daten und Programmcode im selben Speicher.

Eine Alternative zum von-Neumann-Rechner ist dessen, Harvard-Architektur genannte, Weiterentwicklung. Darin werden Daten und Programmcode nicht mehr in dem selben Speicher, sondern getrennt abgelegt. Auf Befehle und Daten kann damit gleichzeitig zugegriffen werden, während dies bei von-Neumann-Architekturen nacheinander erfolgen muss. Außerdem ist es mit einer solchen Architektur möglich, andere Wortbreiten für Daten und Befehle zu verwenden. Der Preis für einen Harvard-Rechner liegt darin, dass Befehle nicht mehr wie Daten behandelt und dadurch nicht mehr vom Programm verändert werden können. Zusätzlich werden zwei getrennte statt einem gemeinsamen Speicher notwendig.

2.5 Stand der Technik

Es gibt eine Vielzahl von verschiedenen Prozessoren für Computersysteme. Im Bereich der Personal Computer werden inzwischen von jedem wichtigen Hersteller Vektorerweiterungen in deren Prozessoren untergebracht. Dabei handelt es sich um verschiedene Erweiterungen mit unterschiedlichen Eigenschaften.

2.5.1 MMX

Die wohl bekannteste Erweiterung ist MMX [Int97] für Intel-Prozessoren, welche 1997 mit dem Pentium MMX eingeführt wurde.

Intel definierte für die MMX-Einheit neue Registerformate zur Datenrepräsentation. Die 64 Bit, welche in einem Register zur Verfügung stehen, lassen sich als „Packed quadword“ (1x 64 Bit), „Packed doubleword“ (2x 32 Bit), „Packed word“ (4x 16 Bit) bzw. „Packed byte“ (8x 8 Bit) interpretieren.

Für Daten der MMX-Befehle werden die bereits vorhandenen Register der Fließkommaeinheit der IA-32-Architektur benutzt. Dadurch benötigt die

MMX-Einheit keine zusätzliche Unterstützung durch das Betriebssystem. Leider ist es so aber auch nicht möglich, Fließkomma- und MMX-Operationen zur selben Zeit abzuarbeiten.

Es wird eine einfache Variante von bedingter Ausführung unterstützt. Vergleiche durch MMX-Anweisungen resultieren in einer Bitmaske, die der Länge der Operatoren entspricht. Diese Bitmaske kann dann mit Hilfe von logischen Verknüpfungen dazu verwendet werden, das Ergebnis zu bilden.

Folgendes Beispiel soll das verdeutlichen:

```
if (bedingung = wahr) Ra = Rb else Ra = Rc;
```

Die Ergebnismaske der Vergleichsoperation ist in *Rx* gespeichert. Diese enthält für alle Bits eines Elements den Wert Eins, wenn die Bedingung erfüllt und den Wert Null, wenn die Bedingung nicht erfüllt ist. Der neue Wert für den Vektor Ra lässt sich in dem Fall durch

```
Ra = (Rb and Rx) or (Rc and not Rx)
```

berechnen.

Mediendaten sind oft aus vielen kleinen Werten zusammengesetzt und Berechnungen mit deren Wortlänge führen sehr häufig zu Überläufen der Wertebereiche. Aus diesem Grund führt die MMX-Einheit viele Berechnungen mit einer Sättigungs-Arithmetik durch. Das bedeutet, wenn ein Überlauf bei einer Berechnung auftritt, wird als Ergebnis der Operation der größtmögliche und bei einem Unterlauf der kleinstmögliche Wert als Ergebnis festgehalten.

Es stehen Befehle zur Verfügung, mit denen Daten umsortiert und zwischen Registern verschoben werden können. Damit ist es beispielsweise möglich, Daten selektiert in ein anderes Register zu verschieben, Zwischenberechnungen mit einer höheren Wortbreite durchzuführen und das Ergebnis mit der ursprünglichen Wortbreite darzustellen.

MMX unterstüzt keine Fließkommaoperationen, sondern erlaubt Berechnungen nur auf Ganzzahl-Basis.

Mit Einführung von SSE2 wurde auch die MMX-Einheit etwas überarbeitet. MMX-Operationen können nun 128-Bit-Register angewandt werden.

2.5.2 SSE

Seit Einführung des Intel Pentium III ist auf den, für den Desktop-Betrieb gedachten, Prozessoren dieses Herstellers zusätzlich eine SSE-Einheit integriert. Diese erlaubt es, SIMD-Operationen auch auf Fließkommawerte anzuwenden. Dazu wurden acht (inzwischen ist diese Anzahl auf 16 gewachsen)

neue 128-Bit-Register auf den Chips untergebracht. Jedes dieser Register enthält vier Fließkommazahlen mit einfacher Genauigkeit [Int99].

Wegen der zusätzlichen Register wurde ein zusätzlicher Zustand im Prozessor eingebaut. Das bedeutet, dass das Betriebssystem SSE, im Gegensatz zu MMX, explizit unterstützen muss.

SSE wurde vielen inkrementellen Erweiterungen [Int06] unterzogen.

- SSE2 erlaubt Operationen auf Fließkommazahlen mit doppelter Genauigkeit. Zusätzlich wird auch die Verwendung von Daten in verschieden großen Integerformaten ermöglicht.
- SSE3 erweitert den Befehlssatz um mathematische Funktionen im Sinne von DSPs und um Befehle für die Handhabung von Threads.
- SSE4 [Int07] und SSE5 [AMD07] finden sich derzeit noch in keinem Prozessor auf dem Markt, sind aber bereits von Intel bzw. AMD angekündigt.

2.5.3 3DNow!

3DNow! ist eine SIMD-Erweiterung von AMD und wurde 1998 in der ersten Version auf dem Prozessor K6-2 vertrieben. 3DNow kann als Fließkommaerweiterung des MMX-Befehlssatzess verstanden werden. Die zu bearbeitenden Daten werden mit den selben Vor- und Nachteilen wie bei der MMX-Einheit in den normalen Fließkommaregistern gehalten.

3DNow! wurde ebenfalls mit mehreren Erweiterungen versehen und befindet sich auch in aktuellen AMD Desktop-CPUs noch im Einsatz. Allerdings werden von AMD seit dem Athlon-Prozessor auch SSE-Einheiten auf ihren Prozessoren integriert.

2.5.4 AltiVec

AltiVec (auch bekannt als VMX oder Velocity Engine) wurde in Kooperation der Firmen IBM, Apple und Motorola entwickelt und auf verschiedenen Varianten von PowerPC-Prozessoren sowie dem IBM Cell Prozessor implementiert. Der erste Prozessor mit AltiVec war der G4 von Motorola. Da der G3 im Vergleich zu den damaligen Prozessoren von Intel und AMD sehr wenig Chipfläche (unter einem Drittel eines Pentium III) beanspruchte, wurde die SIMD-Einheit des G4 mit dedizierter Hardware anstatt unter Benutzung vorhandener Ressourcen realisiert [Sto00].

Für AltiVec Operationen stehen 32 128-Bit-Register zur Verfügung. Darin können Daten entweder als Integerwerte verschiedener Größen aber auch als

32-Bit-Fließkommawerte interpretiert und verknüpft werden [Ful98]. Durch den Umstand, dass die AltiVec-Einheit komplett auf eigener Hardware basiert, können diese Daten auch in einem einzigen Taktzyklus verarbeitet werden. Das ist je nach Befehl und Mikroarchitektur bei SSE bzw. 3DNow! teilweise nicht der Fall.

Interessant ist auch die Tatsache, dass AltiVec-Operationen insgesamt vier Vektorregister (ein Zielregister, zwei Quellregister und ein Register für Filter/Modifikatoren) als Operanden erwarten können. Im Vergleich dazu hat ein MMX-, 3DNow!- bzw. SSE-Befehl lediglich zwei Vektorregister als Operand, wovon eines oft als Ziel- und Quellregister dienen muss.

Besonders mächtig ist AltiVec im Bereich von „horizontalen Operationen“, die auf Daten innerhalb eines Vektorregisters wirken. Damit können beispielsweise die Werte aller Elemente eines Vektorregisters aufsummiert werden. Solche „horizontalen Operationen“ werden auf Seite von Intel und AMD erst richtig seit der Einführung von SSE3 unterstützt [Wiki2].

Auch Permutationen bzw. das Umsortieren von Daten in Vektorregistern ist bei AltiVec sehr durchdacht. Hierbei wird ein Vektorregister mit Anweisungen gefüllt, wie umsortiert werden soll. Dieser Vektor wird dann als zusätzlicher Operand für die Permutationsoperation verwendet.

Eine leicht veränderte AltiVec-Einheit (VMX128) findet im Xenon Prozessor der XBOX 360 Verwendung. Diese wurde für 3D-Graphik- und Physikberechnungen optimiert und ist nicht vollständig kompatibel zu AltiVec [Bro05].

2.5.5 SIMD-Erweiterungen für eingebettete Systeme

Für eingebettete Systeme sind SIMD-Erweiterungen vor allen im Low-Cost-Bereich schwer zu finden. Bei teureren und schnelleren Prozessoren hingegegen sind Erweiterungen vorhanden, die sehr an die der Desktop-Systeme erinnern. Dazu gehören NEON [Neon] für ARM-Prozessoren, iwMMXt für XScale [iwMMXt] aber auch komplette AltiVec-Einheiten für CPUs von Freescale.

2.5.6 Open Source

Statt dem Einsatz dieser teuren Standardprozessoren gibt es aber auch die Möglichkeit, spezialisierte Hardware in Form von ASICs oder entsprechend konfigurierten FPGAs zu verwenden. Leider sieht es dazu im Open-Source-Bereich nicht gut aus. Auf den einschlägigen Webseiten (wie z. B. http://opencores.org) findet sich bisher kein Projekt eines freien Prozessors

mit ernsthafter SIMD-Unterstützung. Stattdessen sind dort eher Nachbauten oder Eigenentwicklungen von einfachen RISC-Prozessoren beziehbar.

Es gibt aber Ansätze [Gui07] einer SIMD-Erweiterung für Leon2-Prozessoren. Dort werden allerdings keine zusätzlichen, breiten Register in die Architektur eingefügt, sondern lediglich die Daten in den vorhandenen 32-Bit-Registern als vier mal 8-Bit bzw. zwei mal 16-Bit interpretiert und mit neuen Befehlen verarbeitet.

Außerdem ist im Internet eine Master-Arbeit [Sha04] aus den USA zu finden, in der ein, speziell auf Wavelet-Video-Kompressionen ausgelegter, konfigurierbarer Vektorprozessor beschrieben und dessen Leitungsfähigkeit untersucht wird. Ob der Prozessor allerdings frei vertrieben wird ist aus dem Dokument leider nicht ersichtlich.

3 Entwurf

3.1 Grundprinzipien

Der in dieser Arbeit entwickelte Prozessor basiert auf den Ideen der Parallelität, Konfigurierbarkeit, Erweiterbarkeit und Offenheit.

Er besteht aus den zwei Komponenten Skalareinheit und Vektoreinheit. Während die Skalareinheit für sich alleine gestellt als SISD klassifiziert werden kann, bearbeitet die Vektoreinheit mehrere Daten im Stil einer SIMD-Architektur. Parallelität bedeutet hier, dass beide Einheiten, voneinander unabhängige Aufgaben, gleichzeitig bearbeiten können.

Der Prozessor ist nicht für einen konkreten Einsatzzweck vorgesehen, sondern soll sich als Soft-Core flexibel für verschiedene Anwendungen einsetzen lassen. Da unterschiedliche Anwendungen oft unterschiedliche Anforderungen an einen Prozessor mit sich bringen, lässt sich dieser durch mehrere Parameter konfigurieren:

- Es kann die Anzahl an Wörtern (K) je Vektorregister eingestellt werden. Je höher dieser Wert ausgelegt ist, umso mehr Daten können mit einem Befehl parallel verarbeitet werden.
- Die Anzahl an Vektorregister (N) ist ebenfalls veränderbar. Mit mehr Vektorregistern lassen sich mehr Daten zwischenspeichern und dadurch Speicherzugriffe reduzieren.
- Die Multiplizierfunktionen in der Skalar- und in der Vektoreinheit lassen sich getrennt voneinander ein- bzw. ausschalten.
- Die Möglichkeit, Daten mit Hilfe des Shuffle-Befehls zu mischen, lässt sich ebenfalls aktivieren bzw. deaktivieren. Zusätzlich ist die Bit-Breite, mit welcher Daten maximal gemischt werden können, konfigurierbar. Das ist notwendig, um bei Werten für K, die keine Zweierpotenzen sind, sinnvolle Bit-Breiten der Ergebnisse zu erreichen (siehe hierzu Abschnitt 4.6)
- Um Daten über das ganze Vektorregister nach links bzw. rechts schieben zu können, muss dies per Konfiguration aktiviert werden. Hier kann außerdem die Anzahl an Bits festgelegt werden, um die mit einem solchen Befehl verschoben wird.

Die Grenzen der Konfiguration ergeben sich daraus, wie viel Logik bzw. Chipfläche der Prozessor in Anspruch nehmen darf. Zusätzlich beeinflusst die Konfiguration auch die maximal mögliche Taktfrequenz. Sollen dabei bestimmte Werte erreicht werden, müssen unter Umständen Abstriche bei der Konfiguration gemacht werden.

Da neue Anwendungen eventuell komplett neue Befehle notwendig machen, ist eine einfache Erweiterbarkeit des Prozessors gewährleistet. Das schließt sowohl eine Erweiterbarkeit des Befehlssatzes, als auch der Mikroarchitektur ein.

„Offenheit“ kann auf zwei verschiedene Weisen aufgefasst werden. Zum einen wird der Prozessor mit komplettem Quellcode unter einer Open-Source-Lizenz veröffentlicht, zum anderen bezieht es sich aber auch auf die Implementierung des Befehlssatzes. Dieser könnte, neben der Implementierung durch die hier vorgestelle Mikroarchitektur, auch mit Hilfe von Pipelining oder Out-of-order-execution realisiert werden.

3.2 Befehlssatzarchitektur

Der Befehlssatz definiert die Menge der auf einem Prozessor ausführbaren Befehle. Anhand der Befehlssatzarchitektur des hier entwickelten Prozessors können schon einige Rückschlüsse über die darunter liegende Implementierung gezogen werden. Es ist die Menge der Register, Adressierungsmöglichkeiten, die Art des Speicherzugriffs, das RISC-Design und der Umfang des Statusregisters erkennbar.

Da der Prozessor nicht für einen konkreten Einsatzzweck entwickelt wird, konnte sowohl die Mikroarchitektur als auch der Befehlssatz in weiten Teilen frei definiert werden. Es wurde bewusst auf komplexe und nicht essentielle Befehle verzichtet, um den Prozessor schnell und einfach zu halten.

Der im Entwicklungsprozess entstandene Befehlssatz lässt sich grob in drei Gruppen aufteilen. Es gibt Befehle für die Skalareinheit, Befehle für die Vektoreinheit und Befehle, welche in Kooperation der beiden Einheiten verarbeitet werden.

3.2.1 Register und Befehlsüberblick

Der skalare Teil des Prozessors verfügt über drei Datenregister. Da ein Befehlswort aus 32 Bit besteht, haben die Register und deren Datenpfade ebenfalls diese Länge.

Die Datenregister haben die Bezeichner A (Akkumulator), X und Y. Neben diesen Registern ist es bei skalaren oder kooperativen Operationen auch möglich, als skalare Quelle oder Ziel „0“ anzugeben. Dadurch wird entweder als Operand der Wert Null verwendet oder das Ergebnis der Operation nicht in ein Register geschrieben. Zusätzlich kann für viele Befehle ein Direktwert als Operand aus dem Befehlswort genutzt werden.

Die Anzahl der Datenregister ergibt sich daraus, dass im Befehl die Quell- und Zielregister mit je zwei Bit kodiert sind. Dadurch kann insgesamt eine Auswahl aus vier Quellen getroffen werden. Drei davon beziehen sich auf die Register, der vierte auf den Wert Null oder den Direktwert.

Die Vektoreinheit besitzt eine konfigurierbare Anzahl von Vektorregistern (R0 ... RN), die sowohl als Quell- als auch als Zielregister dienen.

Um den Befehlssatz übersichtlicher darstellen zu können, werden mögliche Operanden und Parameter von Befehlen zu Gruppen zusammengefasst:

- $\text{reg}_{axy} = A,\ X$ oder Y
- $\text{reg}_{axy0} = A,\ X,\ Y$ oder 0
- $\text{reg}_{axyi} = A,\ X,\ Y$ oder Direktwert (16 Bit)
- vreg_{l} = R0...R15
- vreg_{h} = R<0>...R<N>
- $\text{breite}_{bw} = B$(8 Bit) oder W(16 Bit)
- $\text{breite}_{voll} = B$(8 Bit), W(16 Bit), DW(32 Bit), QW(64 Bit)
- perm = 14-Bit-Permutationsbeschreibung für Mischoperationen (siehe Abschnitt 4.6)

Die in Tabelle 1 aufgeführten Befehle sind spezifiziert und im Prozessor implementiert.

Befehl		**Beschreibung**
LD	reg_{axy0}, [reg_{axy0} + reg_{axyi}]	Ladeanweisung Beispiel: LD A, [0+$FF42]
ST	[reg_{axy0} + reg_{axyi}], A	Speicheranweisung Beispiel: ST [X+Y], A
ADD	reg_{axy0}, reg_{axy0}, reg_{axyi}	Addition Beispiel: ADD X, X, 5
ADC	reg_{axy0}, reg_{axy0}, reg_{axyi}	Addition mit Carry Beispiel: ADC Y, X, A
INC	reg_{axy0}, reg_{axy0}	Inkrementierung Beispiel: INC Y, Y

SUB	reg_{axy0}, reg_{axy0}, reg_{axyi}	Subtraktion Beispiel: SUB 0, Y, $FF
SBC	reg_{axy0}, reg_{axy0}, reg_{axyi}	Subtraktion mit Carry Beispiel: SBC A, A, X
DEC	reg_{axy0}, reg_{axy0}	Dekrementierung Beispiel: DEC 0, Y
AND	reg_{axy0}, reg_{axy0}, reg_{axyi}	Und-Verknüpfung Beispiel: AND A, A, Y
OR	reg_{axy0}, reg_{axy0}, reg_{axyi}	Oder-Verknüpfung Beispiel: OR 0, Y, $FEAB
XOR	reg_{axy0}, reg_{axy0}, reg_{axyi}	Exklusiv-Oder-Verknüpfung Beispiel: XOR X, 0, A
LSL	reg_{axy0}, reg_{axy0}	Schiebeoperation links Beispiel: LSL X, X
LSR	reg_{axy0}, reg_{axy0}	Schiebeoperation rechts Beispiel: LSR Y, X
ROL	reg_{axy0}, reg_{axy0}	Schiebeoperation links (Carry einfügen) Beispiel: ROL A, 0
ROR	reg_{axy0}, reg_{axy0}	Schiebeoperation rechts (Carry einfügen) Beispiel: ROR Y, X
MUL	reg_{axy0}, reg_{axy0}, reg_{axyi}	Multiplizieren, optional Beispiel: MUL X, X, Y
JMP	$[\text{reg}_{axy0} + \text{reg}_{axyi}]$	Unbedingter Sprung Beispiel: JMP [0 + $AD38]
JAL	reg_{axy0}, $[\text{reg}_{axy0} + \text{reg}_{axyi}]$	„Jump-and-Link“ (für Unterprogramme) Beispiel: JAL A, [0 + $FB42]
JZ	$[\text{reg}_{axy0} + \text{reg}_{axyi}]$	Springe, wenn Zero-Flag = 1 Beispiel: JZ [0 + $F7B2]
JNZ	$[\text{reg}_{axy0} + \text{reg}_{axyi}]$	Springe, wenn Zero-Flag = 0 Beispiel: JNZ [X + Y]
JC	$[\text{reg}_{axy0} + \text{reg}_{axyi}]$	Springe, wenn Carry-Flag = 1

		Beispiel: JC [0 + A]
JNC	[reg_{axy0} + reg_{axyi}]	Springe, wenn Carry-Flag = 0 Beispiel: JNC [X + A]
CLZ		Lösche Zero-Flag
SEZ		Setze Zero-Flag
CLC		Lösche Carry-Flag
SEC		Setze Carry-Flag
HALT		Prozessor anhalten
NOP		Keine Skalar-Operation durchführen
VNOP		Keine Vektor-Operation durchführen
MOV	vreg_l (reg_{axy}), reg_{axy0}	Kopiere aus Skalar- in Vektoreinheit Beispiel: MOV R0(A), X
MOV	reg_{axy0}, vreg_l (reg_{axy})	Verschiebe aus Vektor- in Skalareinheit Beispiel: MOV Y, R1(X)
MOVA	vreg_l, reg_{axy0}	Kopiere K mal in Vektorregister Beispiel: MOVA R2, Y
VLD	vreg_l, [reg_{axy0} + reg_{axy}]	Vektor-Ladeanweisung Beispiel: VLD R0, [A + X]
VST	[reg_{axy0} + reg_{axy}], vreg_l	Vektor-Speicheranweisung Beispiel: VST [0 + Y], R0
VMOV	vreg_l, vreg_l	Kopieren von Vektorregistern Beispiel: VMOV R0, R1
VMOV	vreg_h, vreg_l	Beispiel: VMOV R<17>, R1
VMOV	vreg_l, vreg_h	Beispiel: VMOV R0, R<28>

VMOL	$vreg_l$, $vreg_l$	Verschieben von allen Datenwörtern im Vektorregister nach links Beispiel: VMOL R2, R1
VMOR	$vreg_l$, $vreg_l$	Verschieben von allen Datenwörtern im Vektorregister nach rechts Beispiel: VMOR R0, R3
VADD	.$breite_{voll}$ $vreg_l$, $vreg_l$, $vreg_l$	Vektor-Addition Beispiel: VADD.DW R2, R1, R0
VSUB	.$breite_{voll}$ $vreg_l$, $vreg_l$, $vreg_l$	Vektor-Subtraktion Beispiel: VSUB.QW R0, R0, R1
VAND	.$breite_{voll}$ $vreg_l$, $vreg_l$, $vreg_l$	Vektor-Und-Verknüpfung Beispiel: VAND.B R2, R1, R0
VOR	.$breite_{voll}$ $vreg_l$, $vreg_l$, $vreg_l$	Vektor-Oder-Verknüpfung Beispiel: VOR.DW R1, R0, R3
VXOR	.$breite_{voll}$ $vreg_l$, $vreg_l$, $vreg_l$	Vektor-Exklusiv-Oder-Verknüpfung Beispiel: VXOR.W R2, R1, R0
VLSL	.$breite_{voll}$ $vreg_l$, $vreg_l$	Vektor-Schiebeoperation links Beispiel: VLSL.DW R2, R2
VLSR	.$breite_{voll}$ $vreg_l$, $vreg_l$	Vektor-Schiebeoperation rechts Beispiel: VLSR.B R1, R2
VMUL	.$breite_{bw}$ $vreg_l$, $vreg_l$, $vreg_l$	Vektor-Multiplikation, optional Beispiel: VMUL.W R2, R1, R0
VSHUF	$vreg_l$, $vreg_l$, $vreg_l$, perm	Mischen von Vektorregistern Beispiel: VSHUF R2, R1, R0, 11110000011011

Tabelle 1 : Befehlssatz

3.2.2 Befehlskodierung

Die Wortlänge der Befehle für den Prozessor beträgt 32 Bit. Dieser Bereich ist in 12 Bit für Befehle für die Skalar- und 20 Bit für die Vektoreinheit aufgeteilt (Abbildung 4). Es kann in einem Wort also ein Teilbefehl für jede der beiden Einheiten untergebracht und diese auch gleichzeitig verarbeitet werden.

Sind die ersten drei Bit eines Teilbefehls mit 0 belegt, führt die Einheit lediglich einen NOP-Befehl durch. Ein solcher Befehl bewirkt keine Änderung

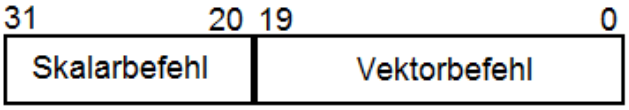

Abbildung 4: Aufteilung eines 32 Bit Befehlswortes

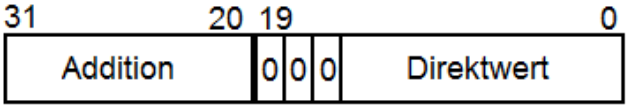

Abbildung 5: Additionsbefehl mit Direktwert

an Registern o.Ä., macht im Endeffekt also gar nichts. In einem solchen Fall können die restlichen Bits des Teilbefehls dafür verwendet werden, Direktwerte oder zusätzliche Parameter für den Befehl der anderen Einheit dort unterzubringen. Auf die Weise ist es z. B. möglich, einen 16-Bit Direktwert für eine Additionsoperation (Abbildung 5) direkt im Befehlswort anzugeben.

Die Befehle sind wie in Tabelle 2 dargestellt kodiert:

Befehl	**Kodierung**	**Kommentar**

Allgemein	`d ⇒ Zielregister`
	`s ⇒ 1.Quellregister bzw. Adresse`
	`t ⇒ 2.Quellregister bzw. Adresse`
	`r ⇒ Vektor-Zielregister`
	`v ⇒ 1.Vektor-Quellregister`
	`w ⇒ 2.Vektor-Quellregister`
	`n ⇒ Direktwert`	

LD	`1000--ddsstt --------------------` `1000--ddss00 000-nnnnnnnnnnnnnnnn`	
ST	`1010----sstt --------------------` `1010----ss00 000-nnnnnnnnnnnnnnnn`	

ALU-Befehle	`01xxxxddsstt --------------------`
	`01xxxxddss00 000-nnnnnnnnnnnnnnnn`
		x ⇒ Operation
		`ADD: 0000, ADC: 0001`
		`SUB: 0010, SBC: 0011`

		INC: 0100, DEC: 0110 AND: 1000, OR: 1001 XOR: 1010, MUL: 1011 LSL: 1100, LSR: 1110 ROL: 1101, ROR: 1111
Sprungbefehle	00xxxxddsstt ---------- 00xxxxddss00 000-nnnnnnnnnnnnnnnn	x ⇒ Sprungart JMP: 0000, JAL: 0000 JNC: 1100, JC: 1101 JNZ: 1110, JZ: 1111
Flag-Befehle	110000--eeff ----------	e ⇒ Flagauswahl f ⇒ neuer Wert CLC: 0100, SEC: 0101 CLZ: 1000, SEZ: 1010
HALT	00101------- ----------	
NOP	000--------- ----------	
VNOP	---------- 000----------	
MOV	101110--sstt 10--0100rrrr-------- 101111dd--tt 10--0101----vvvv----	
MOVA	101101--ss-- 10--0110rrrr--------	
VLD	1001----sstt 10--0010rrrr--------	
VST	101100--sstt 10--0011----vvvv----	
VMOV	---------- 001-0001rrrrvvvv---- 000-nnnnnnnn 001-0010rrrr-------- 000-nnnnnnnn 001-0011----vvvv----	VMOV r,v VMOV r,v(n) VMOV r(n),v
VMOL / VMOR	---------- 001-1000rrrrvvvv---- ---------- 001-1100rrrrvvvv----	VMOL r,v VMOR r,v

VALU-Befehle	`---------- 01bbxxxxrrrrvvvvwwww`	x ⇒ Operation
		`VADD: 0000, VSUB: 0010`
		`VAND: 1000, VOR: 1001`
		`VXOR: 1010, VMUL: 1011`
		`VLSL: 1100, VLSR: 1110`
		b ⇒ Wortbreite
		`8 Bit: 00`
		`16 Bit: 01`
		`32 Bit: 10`
		`64 Bit: 11`

VSHUF	`000-pppppppp 11bbssssrrrrvvvvwwww`	x ⇒ Wortbreite s ⇒ Register-Auswahl p ⇒ Permutation

Tabelle 2: Befehlskodierung

3.2.3 Befehle für die Skalareinheit

Die Befehle für den skalaren Teil des Prozessors lassen sich noch weiter in Lade- und Speicherbefehle, ALU-Befehle, Sprungbefehle sowie Befehle zur Manipulation des Statusregisters unterteilen.

Die Lade- und Speicherbefehle sind die einzigen Befehle, die auf den Speicher zugreifen dürfen. Alle anderen Befehle müssen mit Operanden aus den Registern auskommen, da die Adressierung des Speichers über die Additionsfunktion der ALU realisiert ist und diese damit nicht für sonstige Zwecke zur Verfügung steht. Durch die Verwendung der ALU ist die absolute, register-indirekte und die indizierte [Hen03] Adressierung möglich, ohne dass zusätzliche Logik dafür aufgewendet werden muss.

Die Ladeoperation holt ein Datum aus dem Speicher und legt dieses in einem Register ab. Die Adressierung kann aus einer (nicht komplett beliebigen) Kombination der Register A, X und Y sowie einem Direktwert oder Null erfolgen.

Beispiele für Ladeoperationen:

```
LD    X, [0 + 5]  ; lade Register X von Adresse 5
LD    Y, [0 + A]  ; lade Register Y von Adresse A
LD    A, [X + Y]  ; lade Register A von Adresse X + Y
```

Die Speicheroperation ist das Gegenstück zu Ladeoperation. Hier wird der Inhalt des Registers A (Akkumulator) in den Speicher geschrieben. Die Adressierung erfolgt analog zur Ladeoperation.

Beispiele für Speicheroperationen:

```
ST    A, [0 + 5]  ; speichere Register A an Adresse 5
ST    A, [0 + Y]  ; speichere Register A an Adresse Y
ST    A, [X + Y]  ; speichere Register A an Adresse X + Y
```

ALU-Befehle verknüpfen ein oder zwei Register durch eine logische oder arithmetische Funktion. Das Ergebnis wird in ein Register zurückgeschrieben, in manchen Fällen aber auch verworfen und nur der Inhalt des Statusregisters im nächsten Befehl verwertet.

Beispiele für ALU-Befehle:

```
ADD   A, X, Y     ; addiere X und Y, speichere in Register  A
SUB   A, A, 5     ; subtrahiere 5 von Register A
LSL   X, X        ; schiebe Register X nach links
SUB   0, A, 8     ; vergleiche Register A mit 8
```

Ein 16x16 Bit Multiplikationsbefehl (MUL) ist ebenfalls vorhanden. Da die Implementierung dieses Befehls aber von der Hardware abhängig ist und je nach deren Komponenten den Prozessor eventuell sehr langsam macht, kann die Multiplikation per Konfiguration ein- bzw. ausgeschaltet werden.

Sprungbefehle machen es möglich, Verzweigungen, Schleifen und Funktionen zu realisieren. Der Befehlsatz bietet für die beiden Bits im Statusregister, Carry und Zero, jeweils zwei bedingte Sprungbefehle. Außerdem sind zwei unbedingte Varianten, bei denen entweder einfach gesprungen oder zusätzlich der aktuelle Wert des Befehlszähler in einem Register gespeichert wird, vorhanden. Die zweite Variante kann dazu verwendet werden, Unterprogramme aufzurufen und anschließend wieder zurückzukehren. Das Ziel eines Sprungbefehls wird analog zur Adressierung der Lade- und Speicherbefehle mit Hilfe der ALU berechnet.

Beispiele für Sprungbefehle:

```
JMP   [0 + 5]     ; springe zu 5
JNZ   [Y + 5]     ; springe zu Y + 5, wenn Zero-Flag nicht gesetzt
JC    [X + Y]     ; springe zu X + Y, wenn Carry-Flag gesetzt
JAL   A, [0 + X]  ; springe zu X, hinterlege Befehlszähler in A
```

Die letzte Befehlsgruppe für die Skalareinheit dienen zur Modifikation der Bits im Statusregister. Es ist sowohl für das Carry- als auch das Zero-Flag jeweils ein Befehl zum Setzen oder Löschen vorhanden.

Beispiele für Befehle zur Modifikation des Statusregisters:

```
CLC             ; lösche carry flag
SEZ             ; setze zero flag
```

Von anderen Architekturen ist man als Softwareentwickler gewöhnt, dass Befehle zum Verschieben von Daten zwischen den Registern und zum Vergleich von Werten zur Verfügung stehen. Diese Aktionen lassen sich auch mit dem hier entwickelten Prozessor durchführen. Hierfür mussten keine extra Befehle definiert werden, da bereits vorhandene Instruktionen sich für solche Zwecke verwenden lassen.

Das Kopieren von Daten zwischen Registern kann mit Hilfe der skalaren ALU erfolgen. Eine Oder-Verknüpfung des Quellregisters mit Null und der Angabe eines Zielregisters, hat den gleichen Effekt, wie der von anderen Systemen bekannte *MOVE*-Befehl.

Kopieren von Daten zwischen Registern:

```
OR X, 0, Y      ; Register Y nach X kopieren
OR A, 0, X      ; Register X nach A kopieren
```

Vergleiche lassen sich mit Hilfe der Subtraktionsfunktion der ALU durchführen. Ergebnisse einer ALU-Operation müssen nicht zwingend in ein Register gespeichert werden. Wird als Zielregister einer Operation Null eingetragen, erfolgt nur eine Veränderung des Carry- und Zero-Flags. Die Ergebnisse von Größer-, Gleich- und Kleiner-Vergleichen können somit durch bedingte Sprünge im Programm verwertet werden.

Vergleiche:

```
SUB 0, A, 8     ; vergleiche Register A mit 8
JZ [0 + EQALS]  ; springe zu EQUALS wenn A = 8
```

3.2.4 Befehle für die Vektoreinheit

Die Vektoreinheit kommt mit einer wesentlich geringeren Anzahl an Befehlen aus. Es gibt lediglich Befehle für die Vektor-ALU, zum Transfer und zum Mischen von Daten zwischen den Vektorregistern. Lade- und Speicheroperationen existieren zwar ebenfalls für die Vektoreinheit, allerdings setzen diese die Zusammenarbeit mit dem skalaren Teil des Prozessors voraus und sind deswegen in Abschnitt 3.2.5 näher erläutert.

Wie in Abschnitt 2.3 bereits angesprochen, wirken ALU Befehle in der Vektoreinheit auf einen Vektor von Daten. Bei dem hier entwickelten Prozessor wird immer der komplette Vektor verarbeitet - lediglich Teile daraus auszuwählen ist nativ nicht möglich und muss vom Programmierer durch zusätzliche Befehle umgesetzt werden.

Die Anzahl der Wörter in einem Vektorregister ($R0 \ldots RN$) ist konfigurierbar und wird mit K referenziert. Der Wert für K muss mindestens Eins be-

tragen, nach oben wird er nur durch die Kapazität des verwendeten FPGAs begrenzt.

Bei jeder Vektor-ALU-Operation muss angegeben werden, auf welche Wortbreite der Befehl wirken soll. Zulässige Werte sind hierbei Byte (8 Bit), Word (16 Bit), DoubleWord (32 Bit) und QuadWord (64 Bit). Dadurch ergeben sich je nach Wahl der Wortbreite $K * 4$, $K * 2$, K oder $K / 2$ parallele Operationen. Für logische Verknüpfungen hat die Angabe einer Wortlänge keine Auswirkung.

Eine Anwendung für diese Aufteilung findet sich z. B. wieder in der Bildverarbeitung. Dort sind 32-Bit-Farbwerte für Pixel häufig in einem Format (Abbildung 6) abgespeichert, in dem 8 Bit für Alpha (Transparenz) und jeweils 8 Bit für die Farbkanäle Rot- Grün und Blau vorgesehen sind. Durch die Aufteilung eines 32-Bit-Wortes in vier 8-Bit-Wörter, kann eine Operation parallel aber auch für jeden Kanal getrennt erfolgen.

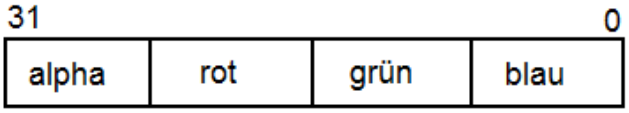

Abbildung 6: 32-Bit-Truecolor-Kodierung

Beispiele für Vektor-ALU-Befehle:

```
VADD.DW R2, R1, R0 ; addiere R1 und R0 in R2, 32 bit Wortlänge
VLSL.B  R1, R1     ; schiebe R1 nach links, 8 bit Wortlänge
VAND.W  R2, R3, R4 ; und-vernüpfen, Wortlänge hat keine Auswirkung
```

Wie bei der Skalareinheit kann auch bei der Vektoreinheit per Konfiguration der 8x8- bzw. 16x16-Bit-Multiplikationsbefehl (VMUL) aktiviert werden.

Einem Befehl für die Vektoreinheit stehen jeweils vier Bit zur Adressierung der Quell- und Zielregister zur Verfügung. Somit können lediglich die ersten 16 Register ($R0 \ldots R15$) direkt im Befehlswort verwendet werden, obwohl die Anzahl der Vektorregister bis 256 konfigurierbar ist. Um die restlichen Register ebenfalls einsetzen zu können, existieren Transferbefehle, mit denen es möglich wird, die nicht direkt adressierbaren auf die direkt adressierbaren Register (oder umgekehrt) zu kopiert.

Beispiele für Vektor-Transferbefehle:

```
VMOV    R1, R0     ; R0 nach R1 kopieren
VMOV    R1, R<42>  ; R42 nach R1 kopieren
VMOV    R<42>, R0  ; R0 nach R42 kopieren
```

Diesen Transferbefehlen sehr ähnlich sind die Anweisungen „VMOL“ und „VMOR“ . Es wird aber nicht, wie das bei „VMOV“ der Fall ist, das Wort an Stelle *i* aus dem Quellregister auch an Stelle *i* sondern an *i+1* bzw. *i-1* im Zielregister geschrieben. Damit können die Datenwörter eines Vektorregisters um jeweils ein Wort nach links oder rechts verschoben werden. Das Datenwort, das am Anfang bzw. Ende des Registers herausfällt, wird am anderen Ende wieder eingefügt. Die gewünschte Wortbreite für diese Operationen ist konfigurierbar, muss aber in den Quellen des Prozessors vorgenommen werden. Vom Programmierer einer Anwendung kann nur diese eine Wortbreite verwendet werden.

Beispiele für Vektor-Schiebeoperationen:

```
VMOL    R0, R0      ; R0 nach links schieben
VMOR    R1, R0      ; R0 nach rechts schieben, in R1 speichern
```

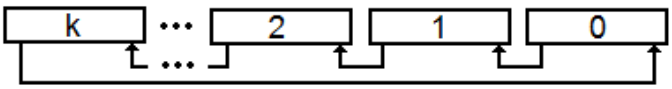

Abbildung 7: Auswirkung des Befehls „VMOL“ mit 32 Bit Wortbreite

Der *Shuffle*-Befehl „VSHUF“ (Abbildung 8) erlaubt das effiziente Mischen von Daten in Vektorregistern. Das Zielregister kann in vier Bereiche partitioniert und diese Bereiche können dann aus ausgewählten Teilen der beiden Quellregister gefüllt werden. Über den Parameter „Wortbreite“ kann die Größe des zu mischenden Bereichs entweder auf das komplette Register oder auf jeweils das erste Halbe, Viertel oder Achtel festgelegt werden. Wie gemischt werden soll, wird im Befehlswort mit 14 Bit festgelegt. Da der Vektoreinheit in einem Befehlswort aber nicht so viel Speicher zur Verfügung steht, werden 8 Bit davon im Teil der Skalareinheit untergebracht. Dadurch kann Shuffle nicht zusammen mit einem skalaren Befehl verwendet werden. Auf die Befehlskodierung wird in Abschnitt 3.2.2 eingegangen, die Operation selbst in Abschnitt 4.6 nochmals genau erläutert.

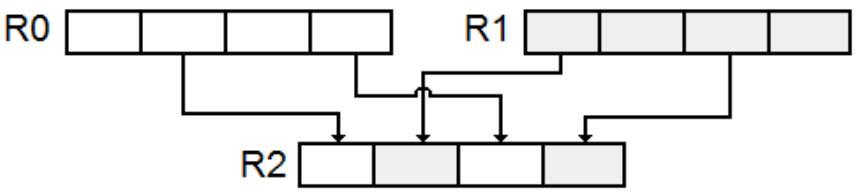

Abbildung 8: Umsortieren mit dem Shuffle-Befehl

3.2.5 Kooperationsbefehle

Die Vektor- und Skalareinheit des Prozessors sind weitgehend autonom, es gibt jedoch auch Befehle, die nur in Kooperation ausgeführt werden können. Dies dann ist dann der Fall, wenn Daten von oder zur Vektoreinheit transportiert werden müssen.

Um mit dem skalaren Teil des Prozessors Daten austauschen zu können, muss ein Wort aus einem Vektorregister selektiert werden. Je nach Richtung des Transfers wird dieses Wort dann entweder übertragen oder neu geschrieben. Die Auswahl des Wortes erfolgt durch die Skalareinheit. Der Index muss sich hierbei in einem der Register A, X oder Y befinden. Der MOVA Befehl benötigt keine Angabe eines Indizes. Hier wird das skalare Quellregister in alle Wörter eines Vektorregisters kopiert.

Beispiele für Transferbefehle:

```
MOV     R0(A), Y    ; Y nach Wort A in R0 kopieren
MOV     A, R1(X)    ; Wort X in R1 nach A kopieren
MOVA    R3, X       ; X in jedes Wort von R3 kopieren
```

Ohne Index kommt die Übertragung von Daten zwischen der Vektor-Einheit und der Speicherschnittstelle aus. Es wird immer ein komplettes Register auf einmal in den Speicher geschrieben bzw. daraus gelesen. Doch auch hier ist die Zuarbeit von der Skalareinheit notwendig. Wie bei den skalaren Lade- und Speicheroperationen wird die Adressierung des Speichers über deren ALU abgewickelt.

Beispiele für Lade- und Speicheroperationen der Vektoreinheit:

```
VLD     R0, A+X     ; Register R0 ab Adresse A+X laden
VST     X+Y, R1     ; Register R1 ab Adresse X+Y speichern
```

3.3 Mikroarchitektur

Wie in Abschnitt 3.1 bereits erkennbar war, besteht der Prozessor aus den Komponenten Skalareinheit und Vektoreinheit. Die Einheiten agieren weitgehend autonom und stehen nur über wenige Daten- und Steuerleitungen (Abbildung 9) miteinander in Kontakt. Bei der Speicherschnittstelle (ebenfalls in dieser Abbildung) handelt es sich zwar nicht um einen direkten Bestandteil des Prozessors, sie ist aber notwendig um die CPU in Betrieb nehmen zu können.

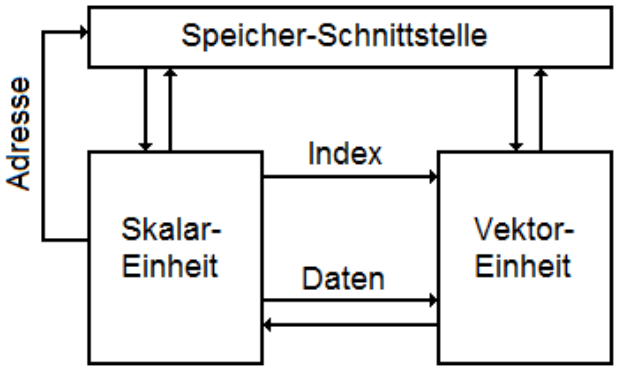

Abbildung 9: Datenverbindungen zwischen den Einheiten

3.3.1 Mikroarchitektur der Skalareinheit

Abbildung 10 zeigt die Struktur der Skalareinheit.

In Abbildung 11 sieht man vor beiden Eingängen der ALU jeweils einen Multiplexer. Die Steuersignale *ss* und *tt* stammen aus dem Befehlswort und kodieren die erste und zweite Datenquelle, welche für die Operation verwendet werden sollen.

Mit ss = „00“ kann der erste Eingang der ALU mit dem Wert „0“ belegt werden. Dies ist immer dann sinnvoll, wenn eine Operation den ersten Eingang der ALU ignorieren bzw. als Null ansehen soll. Solche Fälle wurden bereits in Abschnitt 3.2.3 erläutert.

Die Belegung des zweiten Steuersignals mit tt = „00“ hat wieder eine Sonderfunktion. In dem Fall wird der 16-Bit-Direktwert n aus dem Befehlsregister durchgeschaltet.

Für jede andere Kombination der Bits in *ss* und *tt* wird jeweils eines der Datenregister A, X oder Y ausgewählt.

Die Eingänge der Datenregister sind mit dem Ausgang der ALU verbunden (Abbildung 12). Das ebenfalls im Befehl kodierte Steuersignal *dd* gibt an, wohin das Ergebnis einer Operation gespeichert wird. Dies ist durch einen Demultiplexer realisiert, der je nach Wahl von *dd* einem der Register den Befehl gibt, den am Eingang anliegenden Wert als neuen Wert zu speichern. Im Fall dd = „00“ erhält kein Datenregister diese Anweisung. Das numerische Ergebnis der Operation geht damit zwar verloren, aber den Flags *Zero* und *Carry* im Statusregister (nicht auf der Abbildung) wird trotzdem ein neuer Wert zugewiesen. Dadurch kann der Prozessor Vergleiche zwischen

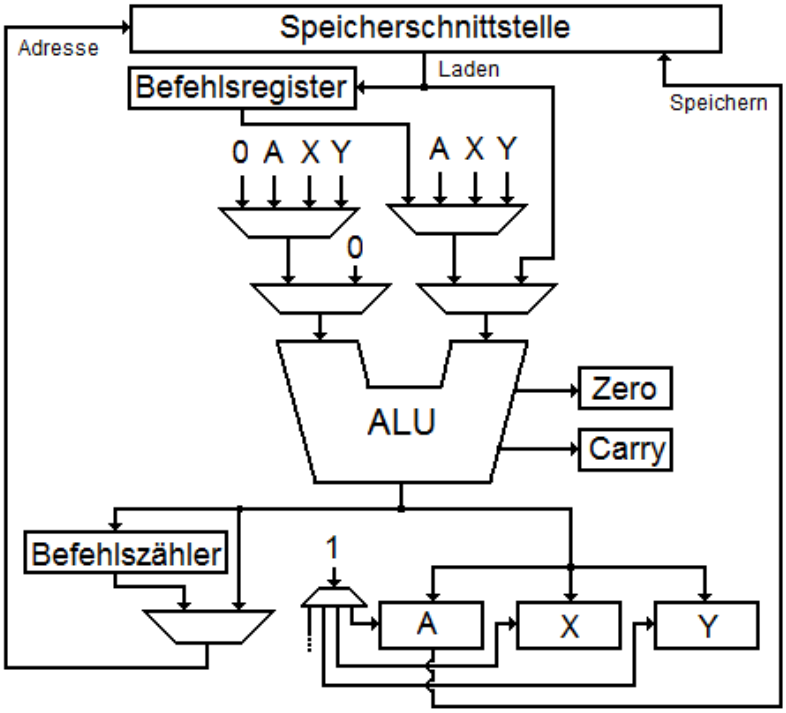

Abbildung 10: Skalareinheit mit Speicherschnittstelle (Vereinfacht)

zwei Werten durchführen, ohne für das Ergebnis den aktuellen Inhalt eines Registers aufgeben zu müssen.

Der Ausgang der ALU ist zusätzlich mit dem Eingang des Befehlszählers und der Adressleitung zur Speicherschnittstelle verbunden (Abbildung 11). Dadurch kann die ALU zur Berechnung der Adressen von Sprungbefehlen sowie von Lade- und Speicheroperationen verwendet werden.

Eine Ladeoperation läuft im Prozessor folgendermaßen ab: Nach Laden des Befehls in den Befehlsspeicher (Instruction Fetch) und dekodieren des Befehls (Decode) werden die beiden in Abbildung 11 dargestellten Multiplexer so geschaltet, dass am ALU Ausgang die Adresse des Befehls anliegt. Gleichzeitig dazu wird der Speicherschnittstelle der Befehl gegeben, den Wert an der anliegenden Adresse auf den Speicherausgang zu schalten. Sobald die Daten bereit stehen, wird der Speicherausgang auf einen Eingang der ALU geschaltet, der andere mit Null belegt und eine Addition ausgeführt (hierfür sind weitere Multiplexer notwendig, welche in Abbildung 11 nicht eingezeichnet sind). Das Speichern des Ergebnisses geschieht wie in Abbildung 12 gezeigt. Damit ist der Ladevorgang abgeschlossen.

Eine Speicheroperation hingegen ist vom Steuerwerk einfacher umzusetzen. Es muss lediglich die Adresse durch die ALU berechnet werden und der

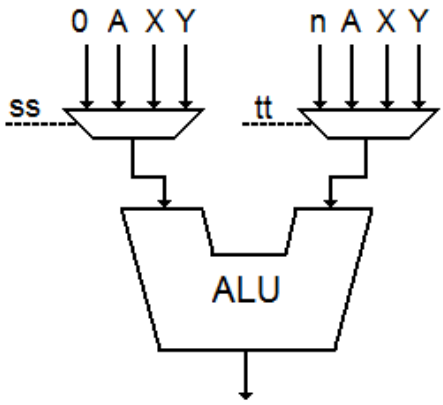

Abbildung 11: Multiplexer am ALU-Eingang (Vereinfacht)

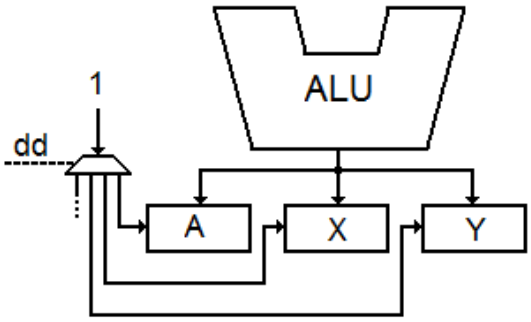

Abbildung 12: Datenregister am ALU-Ausgang

Speicherschnittstelle der Befehl zum Speichern gegeben werden. Der Wert des Akkumulator-Registers A liegt permanent am Dateneingang der Speicherschnittstelle an - es können also nur Daten aus diesem Register geschrieben werden. Dies hat wieder den einfachen Hintergrund, Logik und damit Chipfläche einzusparen.

Das Statusregister des Prozessors wird hauptsächlich für bedingte Sprünge verwendet und ist ebenfalls minimalistisch ausgelegt. Es sind nur die Flags für Carry und Zero vorhanden, welche entweder nach einer Lade-, Speicher- oder ALU-Operation automatisch gesetzt oder durch spezielle Befehle gezielt manipuliert werden.

Das Operationswerk der Skalareinheit steht über drei Datenleitungen in Verbindung mit der Vektoreinheit. Dabei handelt es sich um jeweils eine Leitung zum Senden und Empfangen von Daten sowie einer um ein Wort aus einem Vektorregister auszuwählen.

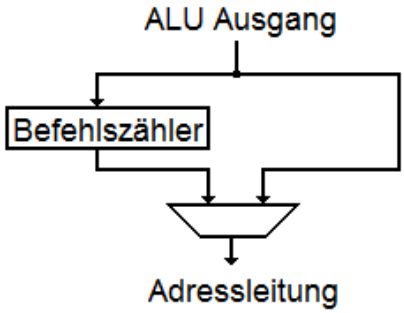

Abbildung 13: Befehlszähler und Adressleitung

3.3.2 Mikroarchitektur der Vektoreinheit

Die Vektoreinheit (Abbildung 14) ist einfacher aufgebaut als die Skalareinheit, da diese nur Daten verarbeiten muss und sich nicht um den Programmablauf und die Speicheradressierung kümmern muss. Das zentrale Element dieser Einheit ist der Vektorregistersatz, da jeder Befehl auf diese Ressource zugreift.

Der Vektorregistersatz beinhaltet mehrere Vektorregister, deren Anzahl (N) in der Konfiguration vorgegeben wird. Jedes dieser Vektorregister besteht aus beliebig vielen 32-Bit-Wörtern. Auch die Anzahl der Wörter (K) wird in der Konfigurationsdatei eingestellt. Höhere Werte für K und N schlagen sich allerdings in wesentlich mehr Bedarf an Chipfläche für den Prozessor nieder.

Abbildung 15 zeigt den Aufbau des Vektorregistersatzes. Eine Zeile stellt darin ein Vektorregister (R0 ... RN) dar, welches wiederum aus K einzelnen Wörtern besteht.

Die Vektoreinheit ist grob in zwei Gruppen aufteilbar.:

1. Logik, die nur einmal in der gesamten Einheit existiert.
2. Logik in 32-Bit-Scheiben, die für jedes Wort in einem Vektorregister (also K mal) instanziert wird.

Um beispielsweise alle Wörter aus zwei Vektorregistern wirklich gleichzeitig addieren zu können, werden K ALUs instanziert. Die Select-Unit auf der anderen Seite ist nur ein einziges Mal vorhanden, da diese aus allen Wörtern eines Vektorregisters lediglich eines auswählt und zur Skalareinheit weitergibt.

Eine 32-Bit-Scheibe (Abbildung 16) der Vektoreinheit besteht aus einer 32-Bit-Scheibe des Vektorregistersatzes, einer auf möglichst kleine Fläche

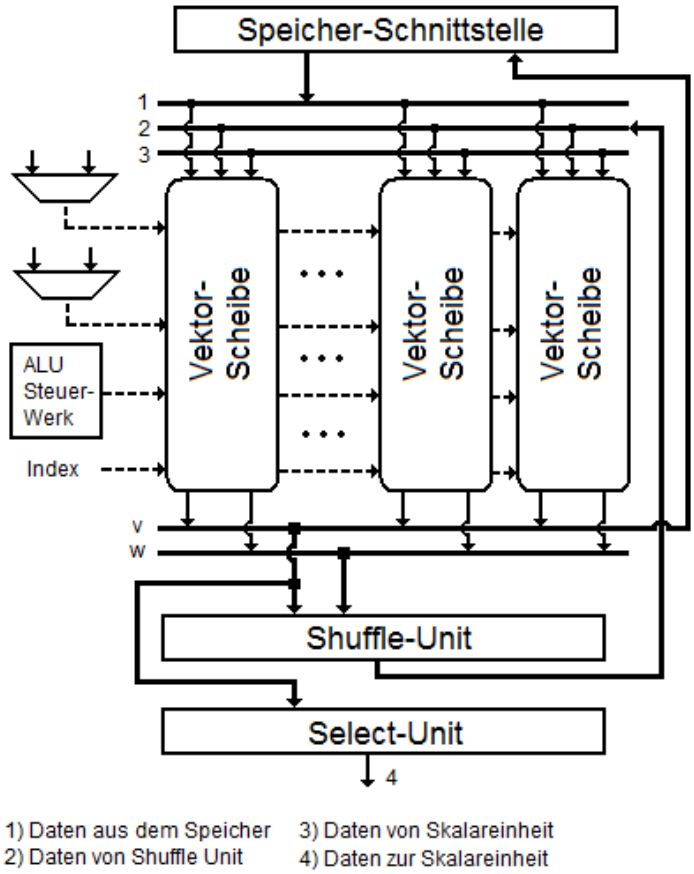

Abbildung 14: Vektor-Operationswerk

optimierten 32-Bit-ALU und einem Multiplexer. Mit Hilfe des Multiplexers wird entschieden, welcher Quelle die Eingangsdaten für die Scheiben des Registersatzes enstammen sollen. Hier kann zwischen dem Ausgang der Vektor-ALUs, der Speicherschnittstelle, der Shuffle-Unit oder der Skalareinheit gewählt werden. Es wurde versucht, die 32-Bit-Scheiben möglichst schlank auszulegen um auch für größere Werte von K möglichst wenig Chipfläche zu beanspruchen.

Jede Scheibe des Vektorregistersatzes hat einen Dateneingang, an dem neu zu speichernde Werte angelegt werden, zwei Datenausgänge, um zwei Register miteinander verknüpfen zu können, und insgesamt drei Leitungen zur Auswahl der Registernummer für Ein- (r) bzw. Ausgänge (v und w). Zusätzlich wird an jede Scheibe die Datenleitung zur Auswahl eines Wortes (Index) von der Skalareinheit angeschlossen. Damit können gezielt Nutzdaten von der Skalareinheit in die gewünschte Scheibe des Vektorregistersatzes kopiert werden. Jede der Vektor-Scheiben besitzt, abhängig von ihrer Positi-

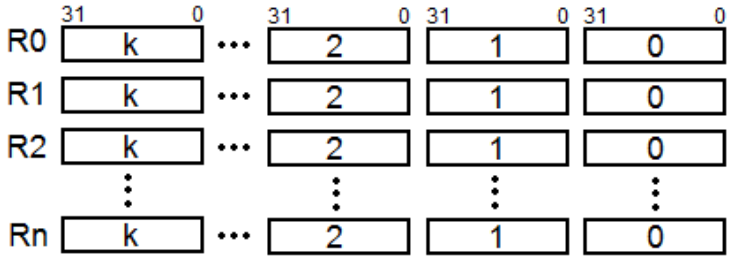

Abbildung 15: Aufbau des Vektorregistersatzes

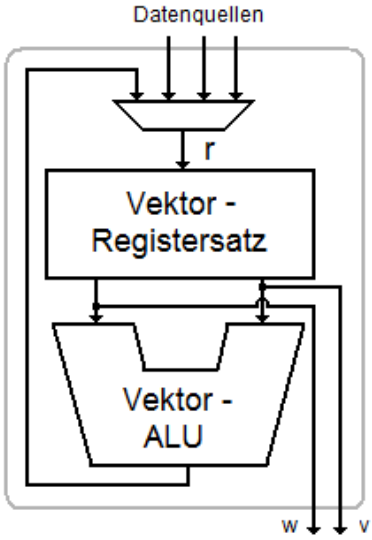

Abbildung 16: 32-Bit-Scheibe der Vektoreinheit

on im Vektor-Operationswerk eine Nummer zwischen 0 und *K-1*. Um Daten von der Skalar- in die Vektoreinheit zu kopieren, werden diese Daten an den Eingang jeder Scheibe angelegt und nur die Scheibe, deren Nummer identisch mit dem Index ist, speichert diese ab.

Komponenten, die nur einmal in der Vektoreinheit instanziert werden, sind die bereits erwähnte Select-Unit zum Auswählen eines Wortes aus einem Vektorregister, die Shuffle-Unit zum Umsortieren bzw. Mischen, ein gemeinsames Steuerwerk (siehe Abschnitt 4.4) für die ALUs und zwei Multiplexer anhand denen entschieden wird, ob die Auswahl des Ziel- bzw. Quellregisters für eine Kopier-Operation aus der dafür vorgesehenen Stelle des Befehlswortes oder dem Direktwert-Bereich vorgenommen werden soll. Außerdem verfügt die Vektoreinheit über einen jeweils K Wörter breiten Datenbus, der zur Speicherschnittstelle und zurück führt und auf die 32-Bit-Scheiben

aufgeteilt wird.

3.4 Speicherschnittstelle

Die Speicherschnittstelle abstrahiert den Zugriff auf den Arbeitsspeicher. Dem Prozessor muss nicht bekannt sein, welche technische Umsetzung von Speicher (Distributed RAM, Block-RAM, externer SRAM, externer DRAM, etc.) tatsächlich verwendet wird. Dadurch kann die Art des RAM leicht geändert werden, ohne dabei das Steuerwerk des Prozessors anpassen zu müssen.

Außerdem hat die Speicherschnittstelle die Aufgabe, der Vektoreinheit des Prozessors parallelen Zugriff auf den Speicher zu ermöglichen. In der Implementierung wird das in manchen Fällen zwar darauf hinauslaufen, dass die Daten intern nur seriell abgerufen/geschrieben werden, extern gesehen stellt die Komponente die Daten aber immer parallel zur Verfügung.

Da der Prozessor niemals gleichzeitig Daten für die Vektor- und die Skalareinheit aus dem Speicher lesen bzw. schreiben kann, hat die Speicherschnittstelle lediglich einen Adresseingang. Die gewünschte Art des Zugriffs wird über eine drei Bit umfassende Signalleitung übermittelt. Dabei handelt es sich um ein Bit, welches bestimmt, ob Daten geschrieben werden sollen (WE), eines das bestimmt, ob Daten gelesen werden sollen (OE) und ein Bit gibt an, ob sich die beiden anderen Bits auf ein einzelnes Datenwort für die Skalareinheit oder einen kompletten Vektor für die Vektoreinheit beziehen (US).

Die Speicherschnittstelle (Abbildung 17) hat sowohl getrennte Leitungen zur Skalar- und Vektoreinheit als auch getrennte Leitungen für ein- und ausgehende Daten.

Da das Steuerwerk des Prozessors wissen muss, wann die Daten an der Speicherschnittstelle zur Abholung bereit stehen, teilt diese ihren Status mit Hilfe des *ready*-Signals mit. Wenn die Speicherschnittstelle bereit ist Befehle entgegenzunehmen, wird das *ready*-Signal gesetzt und bleibt so lange auf diesem Wert, bis ein Befehl eingegangen ist. Sobald dies der Fall ist, setzt die Speicherschnittstelle *ready* zurück, bis die Daten am Ausgang anliegen.

3.5 Schnittstelle zwischen Skalar- und Vektoreinheit

Die Skalar- und die Vektoreinheit verfügen jeweils über ein eigenes Steuerwerk, welche weitgehend autonom arbeiten aber nicht komplett gleichberechtigt sind. Das Steuerwerk der Skalareinheit übernimmt neben der Ansteuerung des skalaren Operationswerkes zusätzlich die Aufgabe den Ablauf

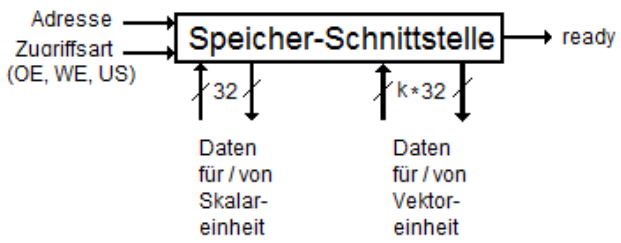

Abbildung 17: Externe Sicht der Speicherschnittstele

des Programmes umzusetzen. Das bedeutet, es ist zusätzlich verantwortlich, die Befehle in den Befehlsspeicher zu laden, bedingte und unbedingte Sprünge auszuführen bzw. den Befehlszähler zu inkrementieren, den Zugriff auf den Speicher für und den Datentransfer zwischen den beiden Einheiten zu koordinieren als auch sich mit dem Steuerwerk der Vektoreinheit zu synchronisieren. Außerdem wird von diesem Steuerwerk der Reset des Prozessors umgesetzt.

Zur Synchronisation zwischen Skalar- und Vektoreinheit stehen sechs Signale zur Verfügung:

1. *ir_ready, Skalareinheit* $\Rightarrow$ *Vektoreinheit:* Der Befehl wurde in den Befehlsspeicher geladen, die Vektoreinheit kann mit ihrer Arbeit beginnen.
2. *v_done, Vektoreinheit* $\Rightarrow$ *Skalareinheit:* Der Befehl wurde vollständig abgearbeitet, die Skalareinheit kann den nächsten Befehl laden, sobald sie ebenfalls fertig ist.
3. *s_ready, Skalareinheit* $\Rightarrow$ *Vektoreinheit:* Daten für die Vektoreinheit stehen zur Verfügung. Dabei kann es sich sowhol um Daten an der Speicherschnittstelle als auch um Daten zum Transfer zwischen den beiden Einheiten handeln.
4. *v_fetched, Vektoreinheit* $\Rightarrow$ *Skalareinheit:* Die Daten wurden von der Vektoreinheit abgeholt, die Skalareinheit kann ihre Arbeit fortsetzen.
5. *v_ready, Vektoreinheit* $\Rightarrow$ *Skalareinheit:* Die Daten an der Vektoreinheit stehen zur Verfügung und können von anderen Komponenten gelesen werden. Dabei handelt es sich entweder um die Speicherschnittstelle oder die Skalareinheit,
6. *s_fetched, Skalareinheit* $\Rightarrow$ *Vektoreinheit:* Die Daten an der Vektoreinheit wurden abgeholt, sie kann ihre Arbeit fortsetzen.

Das Steuerwerk der Skalareinheit hat neben diesen Signalen nur noch wenige Eingänge. Dabei handelt es sich um das Befehlswort, das *ready*-Signal der Speicherschnittstelle, das *Reset*-Signal und um die beiden Flags *Carry* und *Zero* des Statusregisters. Das skalare Operationswerk ist so ausgelegt, dass es nur mit einzelnen Steuersignalen auskommt und viele Multiplexer direkt aus Teilen des Befehlswortes geschaltet werden.

Der Zustandsautomat im skalaren Steuerwerk wurde mit den Phasen *Instruction Fetch*, *Instruction Decode* und *Execute* entworfen. Darauf aufbauend ist zusätzlich der Zustand *Synchronisieren* vorhanden, in dem das Steuerwerk darauf wartet, dass auch die Vektoreinheit mit ihrer Arbeit fertig ist. Nach dem Einschalten der Hardware befindet sich der Prozessor aber zunächst im *Halte*-Zustand und startet mit der Abarbeitung des Programmes erst, nachdem er das *Reset*-Signal empfangen hat.

Im Steuerwerk der Vektoreinheit ist die Phase *Instruction Fetch* dadurch ersetzt, dass lediglich auf die Meldung der Skalareinheit gewartet wird, ob sich der neue Befehl im Befehlsregister befindet. Die Vektoreinheit benötigt auch keinen Synchronisationszustand, da nach Abarbeitung einer Operation einfach wieder in den Wartezustand gewechselt werden kann.

4 Implementierung

Eine Voraussetzung an den Prozessor ist, dass dieser einfach erweiterbar sein soll. Das wird durch drei Ansätze erreicht:

1. Die Befehlskodierung ist so ausgelegt, dass noch Platz für neue Befehle frei ist bzw. leicht geschaffen werden kann.
2. Der Prozessor wird als Soft-Core in der Hardwarebeschreibungssprache VHDL (siehe Abschnitt 2.2) implementiert und kann damit auch von demjenigen modifiziert werden, der den Prozessor einsetzen will.
3. Der Quellcode ist in mehreren, klar voneinander getrennten Komponenten organisiert. Dadurch kann die richtige Stelle für Änderungen schnell gefunden werden und das Risiko, unerwünschte Auswirkungen auf andere Bereiche des Prozessors zu verursachen, wird minimiert.

Neben dem Prozessor selbst wurden Programmierwerkzeuge in Form eines Assemblers und eines Debuggers in der Programmiersprache Python realisiert. Damit konnten Programme erzeugt und mit Hilfe des Debuggers auf der Testhardware, einem Xilinx Spartan 3A (XC3S700A) FPGA (siehe Abschnitt 2.1) ausgeführt und validiert werden.

4.1 Hierarchischer Aufbau

Da der Quellcode für den Prozessor doch einen beträchtlichen Umfang hat, wurden die Komponenten hierarchisch über mehrere Ebenen aufgebaut. Dadurch wird das Programm übersichtlicher und die Wahrscheinlichkeit von Fehlern bei Signalzuweisungen reduziert, da diese oft innerhalb einer Komponente erfolgen kann und nicht alles in einer Ebene erfolgen muss. Zusätzlich können einige der Komponenten mit Hilfe von Testbenches in kleineren Gruppen und nicht nur einzeln oder im Gesamtsystem getestet werden.

Die oberste Ebene wird als System bezeichnet. Hier werden die Komponenten „Debugger“ (siehe Abschnitt 5.3) und „Speicher“ an den Prozessor angeschlossen.

Die Prozessor- oder CPU-Ebene besteht wiederum aus den Komponenten „Skalar-Steuerwerk“, „Vektor-Steuerwerk“, „Vektor-Operationswerk“ und verschiedenen Komponentengruppen für das skalare Operationswerk (ALU, Registersatz, etc.).

Das Vektor-Operationswerk besteht aus einmal instanzierten Komponenten wie dem Vektor-ALU-Steuerwerk und den mehrfach (K mal) erzeugten

Scheiben, welche aus Vektor-ALUs, Multiplexern und einem Registersatz aufgebaut sind.

4.2 Konfigurierbarkeit

Was am Prozessor konfiguriert werden kann, wurde bereits in Kapitel 3.1 aufgeführt. Jeder dieser Parameter wirkt sich aber nicht nur auf die Funktionalität, sondern auch auf die benötigte Chipfläche und teilweise auch auf die Geschwindigkeit der Schaltung aus. Es muss also je nach Anwendung eine geeignete Konfiguration erstellt werden. Damit kann gemeint sein, einen kompletten Chip zu füllen um die maximale Rechenleistung zu erzielen, oder aber möglichst wenig Platz zu belegen, um genau die geforderte Leistung zu erreichen.

Die Konfiguration wird zentral für den gesamten Prozessor in der Datei *config.vhd* vorgenommen. Es müssen keine VDHL-Komponenten ausgetauscht sondern lediglich die Werte von Konstanten angepasst werden.

Beispiel-Konfiguration

```
package cfg is
    constant vector_registers   : integer   := 8;  -- N
    constant vector_size        : integer   := 16; -- K
    constant use_debugger       : boolean   := true;
    constant use_scalar_mult    : boolean   := false;
    constant use_vector_mult    : boolean   := false;
    constant use_shuffle        : boolean   := true;
    constant max_shuffle_width  : integer   := 512;
    constant use_vectorshift    : boolean   := true;
    constant vectorshift_width  : integer   := 32;
end cfg;
```

Die Umsetzung dieser Konstanten erfolgt im VHDL-Quellcode entweder, indem die Konstante direkt als Zahlenwert (z. B. als Arraygröße) verwendet oder als Bedingung für eine *generate*-Anweisung eingesetzt wird. Mit *generate* lassen sich in einer Schleife mehrere Instanzen von der selben Komponente bilden. Außerdem ist es damit möglich Quellcode-Abschnitte nur dann zu synthetisieren, wenn eine Bedingung zutrifft.

Konstante *N* als Arraygröße

```
type regfile_type is array(0 to N-1) of std_logic_vector(31 downto 0);
```

Konstante *K* für *generate*-Schleife

```
vector_slice_impl: for i in K-1 downto 0 generate
```

```
    slice: vector_slice
        port map (
            ...
        );
end generate ;
```

Konstante *use_skalar_mult* für *generate*-Bedingung

```
mult_gen: if use_scalar_mult generate
    mult_res <= left(15 downto 0) * right(15 downto 0);
end generate;

not_mult_gen: if not use_scalar_mult generate
    mult_res <= (others => '0');
end generate;
```

4.3 Optimierung der Steuerwerke

In die Optimierung des Prozessors ist ein sehr großer Teil der Zeit geflossen, die für die gesamte Entwicklung aufgewendet werden musste. Neben der Steigerung der Taktfrequenz wurde auch darauf geachtet, möglichst wenig Chipfläche zu beanspruchen und die Anzahl an Takten, die für einen Befehl notwendig sind, zu reduzieren.

An manchen Stellen war es aber auch sinnvoll, weniger genutzte Befehle auf mehrere Takte zu verteilen, um die Taktfrequenz des Gesamtsystems erhöhen zu können oder viel Chipfläche einzusparen.

Die Steuerwerke des Prozessors selbst (Skalar- und Vektoreinheit) wurden unter Verwendung von endlichen Automaten realisiert. Hier stellte sich die Alternative zwischen Moore- und Mealy-Automaten.

Bei Moore-Automaten hängt dessen Ausgabe lediglich von seinem aktuellen Zustand ab, während die Ausgabe eines Mealy-Automaten durch seinen Zustand und seine Eingabe bestimmt wird.

Das Steuerwerk der Skalareinheit wurde zunächst als Mealy-Automat realisiert. Durch die Möglichkeit, Aktionen beim Übergang zwischen den Zuständen auszuführen und auf Ereignisse direkt zu reagieren, können wertvolle Takte bei der Abarbeitung der Befehle eingespart werden.

Allerdings hat sich herausgestellt, dass der Mealy-Automat zwar weniger Zustände benötigt, dadurch aber auch die Pfade für Signale länger werden und die mögliche Taktfreuenz des gesamten Prozessors sinkt. Außerdem besteht bei einem Mealy-Automat die Gefahr, ungewollt kombinatorische Schleifen zu erzeugen.

Aus diesen Gründen wurden die Steuerwerke auf einen Moore-Automat umgestellt und dadurch eine Steigerung der Taktfrequenz von 19 auf 28 MHz bei (abhängig vom Befehl) vorerst ein bis zwei zusätzlichen Takten erreicht.

Neben der Umstellung auf einen Moore-Automaten wurde vor allen Dingen das Zusammenspiel mit der Speicherschnittstelle verbessert. Das Steuerwerk der Speicherschnittstelle wurde von einem Moore- in einen Mealy-Automat konvertiert, um der Skalareinheit möglichst schnell Änderungen an deren Zustand mitteilen zu können. Zusätzlich wird vor einem Zugriff auf die Speicherschnittstelle nicht mehr darauf gewartet, dass diese ihre Bereitschaft mit dem ready-Signal bestätigt. Da die Skalareinheit prinzipiell die einzige Einheit ist, die dort Zugriffe veranlasst, kann davon ausgegangen werden, dass die Speicherschnittstelle vor dem Lesen bzw. Schreiben von Befehlen und Daten immer bereit sein muss. Zwar wird auch vom Debugger auf den Speicher zugegriffen, dieser sorgt aber dafür, dass die Speicherschnittstelle nach einer Operation wieder in dem Zustand ist, den das Steuerwerk erwartet. Dies ist dadurch möglich, dass der Debugger den Prozessor unabhängig von Speicher und Speicherschnittstelle takten kann. Wollen andere Komponenten des Systems ebenfalls auf den Speicher zugreifen, ist es die Aufgabe der Speicherschnittstelle auf die Bereitschaft des Speichers zu warten. Durch diese Änderungen können Lade- und Speicheranweisungen in zwei statt bisher vier Taktzyklen abgeschlossen werden.

4.4 Optimierung der Vektor-ALU

Die 32 Bit breiten ALUs der Vektoreinheit müssen in der Lage sein, ihre Operationen auf Daten im 8, 16, 32 und 64 Bit Format anzuwenden. Aus diesem Grund wurde jede dieser 32-Bit-ALUs aus jeweils vier 8-Bit-ALUs (Abbildung 18) zusammengesetzt, welche über eine Carry Leitung und eine Leitung für den Übertrag der Schiebeoperationen nach rechts miteinander in Verbindung stehen. Dadurch müssen, falls eine Operation nicht auf die gesamte Breite durchgeführt werden soll, lediglich die Übertragsleitungen getrennt werden.

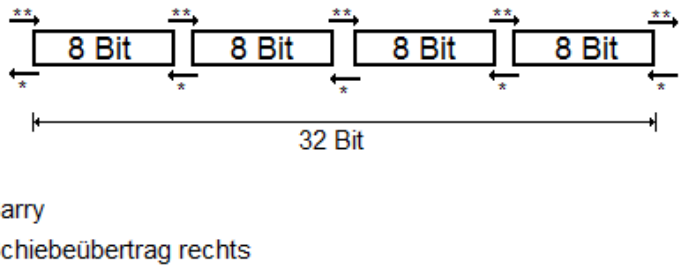

Abbildung 18: Zusammengesetzte 32-Bit-Vektor-ALU

Um auch Daten mit 64 Bit verarbeiten zu können, müssen auch jeweils zwei 32-Bit-ALUs bzw. Vektor-Scheiben durch die beiden oben genannten Übertragsleitungen miteinander in Verbindung (Abbildung 19) stehen. Das wurde im VHDL-Quellcode durch ein *generate*-Statement realisiert, bei dem für gerade Werte der Indexvariable die Übertragsleitung anders als bei ungeraden Werten verbunden werden. Der Carry-Ausgang der geraden Instanz geht an den Carry-Eingang der ungeraden, der Carry-Ausgang der ungeraden geht an den Eingang der geraden Instanz, der für den Übertrag von Schiebeoperationen nach rechts vorgesehenen ist.

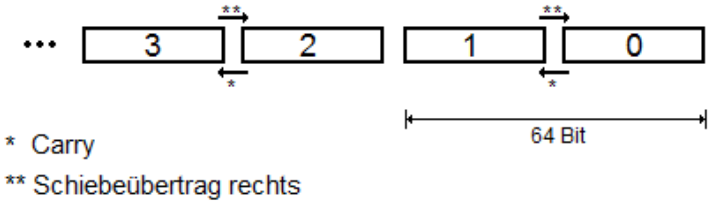

Abbildung 19: Verbindung der Vektor-ALUs für 64-Bit-Modus

Durch die Aufteilung einer 32-Bit-ALU in 8-Bit-Komponenten werden darin vom Synthesetool leider keine optimierten 32-Bit Rechenwerke (z. B. Carry-Look-Ahead-Addierer) gebildet, sondern vier 8-Bit-Versionen davon in Reihe geschaltet. Im Fall einer Addition mit 64 Bit Datenbreite ergeben sich also acht in Reihe geschaltete 8-Bit-Addierer. Diese Reihenschaltung sorgt für sehr hohe Signallaufzeiten und war der Grund, warum der Prozessor auch nach Optimierung der Steuerwerke nur mit 28 MHz lief.

Aus diesem Grund wurden zunächst die 8-Bit-ALUs auf das Carry-Select-Prinzip (parallelisierte Vorabberechnung, Abbildung 20) umgestellt. Jede 8-Bit-ALU wurde dabei doppelt erzeugt. An einem Block liegt an den Übertragseingängen fest Null, an dem anderen fest Eins an. Es werden also die Ergebnisse für beide Fälle parallel berechnet. Die Auswahl, welches der beiden Ergebnisse tatsächlich verwendet werden soll, geschieht über einen Multiplexer, der anhand des Übertragswertes des Vorgängerblocks geschaltet wird. Dadurch wird die Signallaufzeit darauf beschränkt, eine 8-Bit-Operation auszuführen und die Multiplexer an den restlichen Blöcke nacheinander umzuschalten.

Durch diese Methode konnte zwar die Taktfrequenz von 28 MHz auf 49 MHz gesteigert werden, allerdings wurde auch die für die ALUs benötigte Chipfläche ungefähr verdoppelt. Aus diesem Grund konnte in der Konfiguration eingestellt werden, ob die ALUs nach dem Carry-Select-Prinzip aufgebaut werden sollten oder nicht. Zusätzlich konnte auch der 64-Bit-Modus ausgeschaltet und damit auch ohne Carry-Select eine höhere Taktfrequenz erreicht werden.

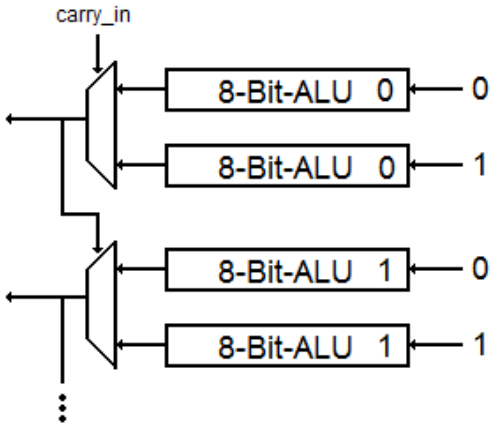

Abbildung 20: Carry-Select-Prinzip

Ein anderer, allgemeiner Ansatz zur Erhöhung der Taktfrequenz ist, eine Berechnung auf mehrere Takte zu verteilen und dabei in jedem Takt nur ein Teilproblem zu lösen.

Dieser Ansatz wurde umgesetzt, indem die 32-Bit-ALUs komplett neu und als Schaltwerk (Abbildung 21) statt wie bisher als Schaltnetz aufgebaut wurden. Anstatt vier 8-Bit-Operationen parallel in vier 8-Bit-ALUs auszuführen, werden in vier Takten nacheinander immer 8 Bit aus den Eingangsdaten ausgewählt, diese von einem einzigen Rechenwerk verarbeitet und das Ergebnis in ein 33-Bit-Register (Übertragbit + 32 Datenbits) geschoben.

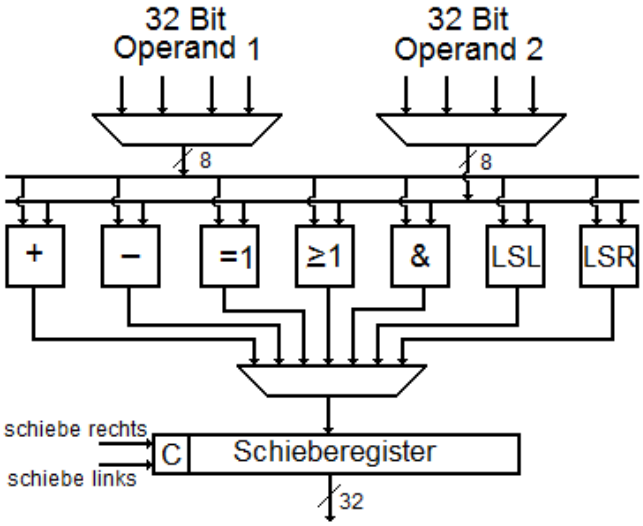

Abbildung 21: Vereinfachte Darstellung: Vektor-ALU Schaltwerk

Dadurch entfallen zwar in jeder 32-Bit-ALU drei der vier 8-Bit-ALUs, allerdings ist auch zusätzliche Logik für das Register, die Multiplexer und das Steuerwerk der Vektor-ALUs notwendig. Da die Steuerung der Vektor-ALU-Befehle unabhängig von den daran anliegenden Daten ist, existiert ein gemeinsames Vektor-ALU-Steuerwerk, welches sich außerhalb der 32-Bit-Scheiben befindet.

Da die Handhabung einer ALU-Implementierung in Forum eines Schaltwerkes schwieriger zu handhaben ist als ein Schaltnetz, müssen zusätzliche Steuersignale von dem Steuerwerk der Vektoreinheit zum Steuerwerk der Vektor-ALUs und zurück führen. Dabei bestimmt ein Signal, dass die Vektor-ALUs mit ihrer Arbeit beginnen sollen. Ein anderes Statussignal bestätigt, wenn die Vektor-ALUs ihre Arbeit durchgeführt haben und die Daten am Ausgang abgeholt werden können. Dieses Signal könnte prinzipiell entfallen, allerdings müsste das Steuerwerk der Vektoreinheit dann wissen, wie lange die einzelnen ALU-Operationen dauern. 64-Bit-Operationen benötigen mehr Takte als 8-, 16- oder 32-Bit-Operationen. Aus diesem Grund und unter Berücksichtigung des Umstandes, dass der Prozessor leicht erweiterbar sein soll, wurde dieses Signal eingeführt. Das bedeutet, dass Operationen, die mehr oder weniger Takte benötigen (z. B. Multiplizieren durch Additionen), später leichter integriert werden können, da das Steuerwerk der Vektoreinheit nicht angepasst werden muss, sondern die Änderungen lokal in der ALU vorgenommen werden können.

Insgesamt konnte durch diese Maßnahme die Taktfrequenz auf von 49 auf 67 MHz angehoben und die Chipfläche für die Vekoreinheit auf gut ein Drittel reduziert werden. Das alles wird damit bezahlt, dass ein Vektor-ALU-Befehl fünf zusätzliche Takte benötigt. Da die Chipfläche auf diese Weise aber dafür verwendet werden kann, die Anzahl an Worten pro Vektorregister um ca. das Dreifache zu erhöhen, die Performanz für alle anderen Befehle durch die erhöhte Taktfreuenz steigt und der 64-Bit-Modus keine weiteren Kosten verursacht, wurde diese Implementierung beibehalten und das vorhergehende System mit Carry-Select-Prinzip komplett verworfen.

4.5 Multiplizierer

Um auch die Multiplizier-Funktion in die Vektor-ALUs zu integrieren, wurde vor dem 33-Bit-Register innerhalb der Vektor-ALUs ein zusätzlicher Multiplexer eingefügt (Abbildung 22). Dieser kann so umgeschaltet werden, dass entweder das Ergebnis aus der Multiplikation oder das der anderen Operationen anliegt.

Da die Multiplizierer die Signallaufzeiten stark verlängern und dadurch auch Block-RAM-Ressourcen aufgewendet werden müssen, kann das Multiplizie-

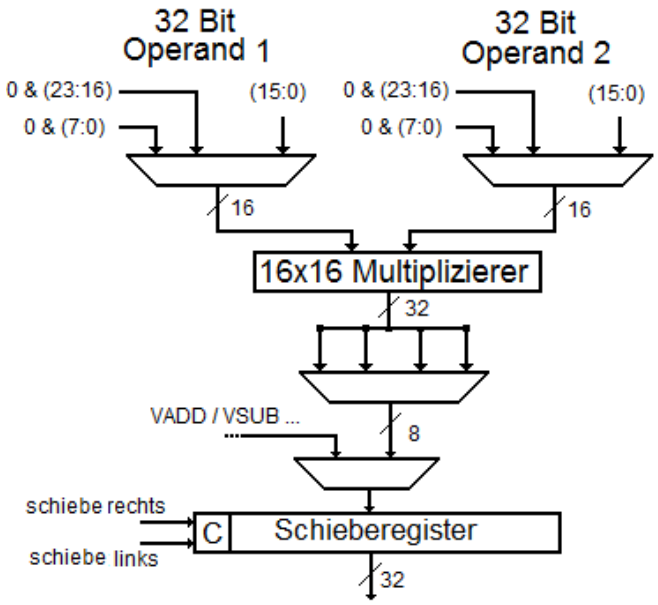

Abbildung 22: Einbindung des Multiplizierers

ren per Konfiguration aktiviert bzw. deaktiviert werden. Wird keine Multiplikation verwendet, setzt der Multiplikationsbefehl die Zielregister auf den Wert Null.

4.6 Shuffle

Die Shuffle-Unit ist für das horizontale Verschiebem von Daten verantwortlich. Für den Mischvorgang können im Befehl verschiedene Wortbreiten angegeben werden. Um die folgende Erklärung einfacher nachvollziehen zu können, wird zunächst die Wirkung des Shuffle-Befehls mit voller Wortbreite erläutert. Außerdem wird davon ausgegangen, dass die Shuffle-Unit mit der maximal möglichen Bit-Breite konfiguriert und synthetisiert wurde.

4.6.1 Definition der Operation

Die Shuffle-Operation arbeitet mit den Daten aus zwei Quell-Vektorregistern v und w. Diese beiden Vektorregister werden jeweils in vier gleich große Teile zerlegt und aus den enstandenen Teilen in beliebiger Reihenfolge das Ausgabe-Vektorregister r gebildet. Das Ausgabe-Vektorregister hat dieselbe Größe wie ein Quell-Vektorregister. Das bedeutet, es haben maximal vier der acht Teile darin Platz. Ein Teil aus v oder w kann aber auch zwei-, drei-, oder viermal in an verschiedene Positionen in r kopiert werden.

Die Definition des Mischvorgangs (Abbildung 23) erfolgt mit Hilfe der 14-Bit-Perumtationsbeschreibung des „VSHUF“-Befehls. Von Links gesehen geben deren erste zwei Bits (*ww*) die Wortlänge an. Anhand der nächsten vier Bits (*ssss*) wird entschieden, welcher Teil von r aus v oder aus w kopiert werden soll. Bei einer binären Null werden Daten aus v, bei einer Eins Daten aus w verwendet. Die restlichen acht Bits (*nnnnnnnn*) geben die Position der Teile an, von denen die Daten aus den Quellregistern kopiert werden sollen. Darin sind jeweils zwei Bit für eine Position vorgesehen, wobei „00“ den niederwertigsten Teil und „11“ den höchstwertigsten Teil der Quellregister bezeichnet.

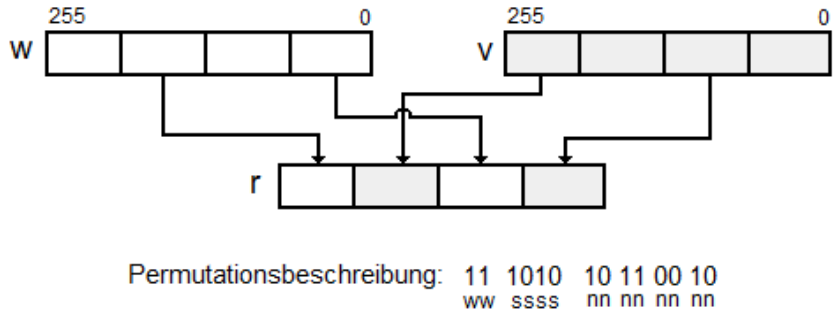

Abbildung 23: Shuffle-Operation bei voller Wortbreite und K=8

Die Wortbreite für die Shuffle-Operation kann man in vier Schritten vorgeben. Volle Wortbreite (*ww*=„11“) bedeutet, dass ganze Vektorregister in vier Teile zerlegt werden. Bei halber Wortbreite (*ww*=„10“) unterteilt die Shuffle-Unit die Vektorregister in zwei Hälften und behandelt diese so, als wären sie ein ganzes Vektorregister. Das bedeutet, die Hälften werden jeweils in vier Teile zerlegt, nach dem oben beschriebenen Verfahren gemischt und das Resultat in die Hälfte des Vektorregisters r geschrieben. Bei den verbleibenden Wortbreiten *ww*=„01“ und *ww*=„00“ arbeitet die Shuffle-Einheit analog dazu auf jeweils einem Viertel bzw. einem Achtel der Größe eines kompletten Registers.

In der Konfiguration kann die Bit-Breite eingestellt werden, welche die Shuffle-Unit maximal mischen kann. Ist dieser Wert kleiner als die Länge eines Vektorregisters, behandelt die Einheit die Register nur bis zu dieser Schranke. Der Rest des Ausgabe-Vektorregisters r wird mit Daten aus v aufgefüllt, die sich an entsprechender Position befinden.

4.6.2 Umsetzung der Shuffle-Unit

Bei einer einfachen Implementierung der Shuffle-Unit als Schaltnetz hat sich gezeigt, dass diese extrem viel Chipfläche (ca. 4100 Slices und 8200 LUTs auf

dem Test-FPGA XC3S700A bei K=16) in Anspruch nimmt. Dies ist darin begründet, dass für jedes der Quellregister vier Multiplexer pro Wortbreite existieren. Zusätzlich muss ausgewählt werden, ob die Daten aus v oder w übernommen werden. Da die Vektorregister, je nach Wahl des Parameters K in der Konfiguration, eine beträchtliche Breite haben können, entstehen daraus sehr große Multiplexerstrukturen. Aus diesem Grund wurde die Abarbeitung der Operation auf mehrere Takte aufgeteilt.

Es hat sich folgender Ansatz (Abbildung 24) als der geeignetste der getesteten erwiesen:

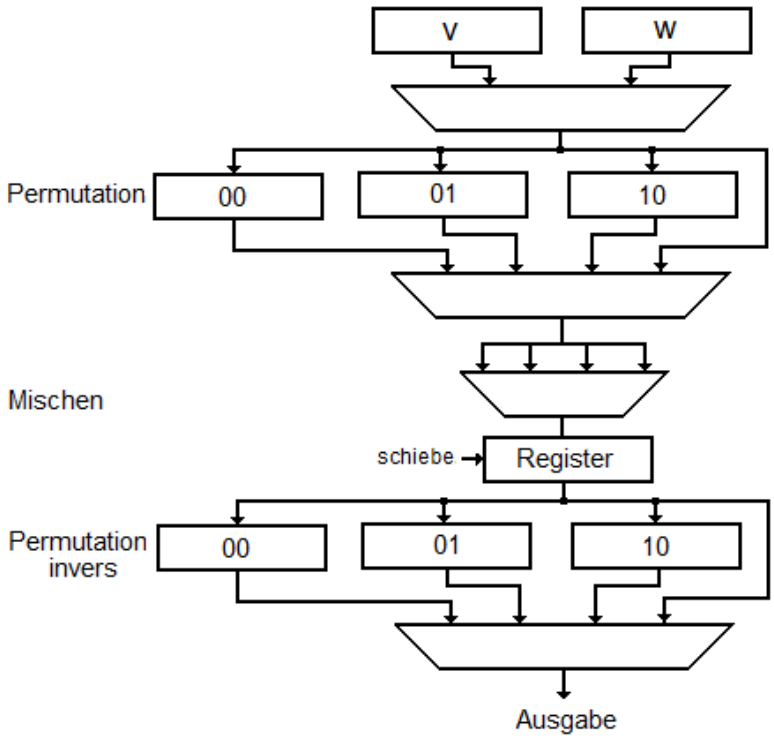

Abbildung 24: Struktur der Shuffle-Unit

Die Abarbeitung lässt sich in drei logischen Schritten beschreiben. Die Datenquellen werden mittels einer festen Logik permutiert, gemischt und anschließend mit der inversen Logik erneut permutiert.

Die Permutation (Abbildung 25) ist abhängig von der Wortbreite der Shuffle-Operation, in der höchsten Wortbreite kann diese sogar ganz entfallen. Das Quellregister wird als partitioniert in vier gleich große Teile betrachtet. Je nach Wortbreite der Operation werden diese vier Teile wiederum in zwei (ww=„10“), vier (ww=„01“) oder acht (ww=„00“) Bereiche unterteilt.

Durch die Permutation wird die Reihenfolge der Daten im Quellregister dann so verändert, dass das erste Viertel des Ergebnisses aus jeweils den ersten Bereichen der vier großen Teile besteht. Dasselbe Prinzip gilt auch für die restlichen drei Viertel. Diese setzen sich aus den entsprechenden Bereichen innerhalb der verbleibenden drei großen Teile zusammen.

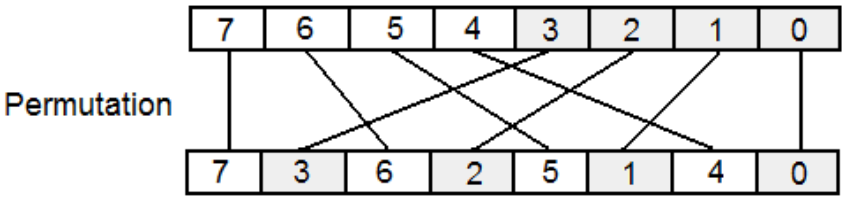

Abbildung 25: Permutation bei Wortbreite „10“

Beim Mischen (Abbildung 26) werden die vier, durch die Permutation entstandenen Teile in vier Schritten in ein Schieberegister eingefügt. Die Reihenfolge der Teile wird dabei entsprechend der Angabe von *nnnnnnnn* im Shuffle-Befehl verändert. Dieser Schritt ist nicht abhängig von der Wortbreite.

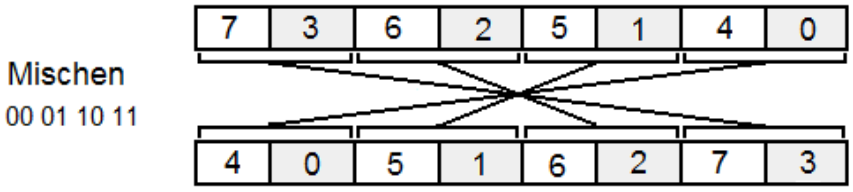

Abbildung 26: Mischen bei Wortbreite „10“

Der letzte Schritt (Abbildung 27) für die Shuffle-Operation besteht darin, die Daten aus dem Schieberegister anhand einer Vorschrift, die genau invers zur bereits angewandten Permutation ist, erneut umzusortieren.

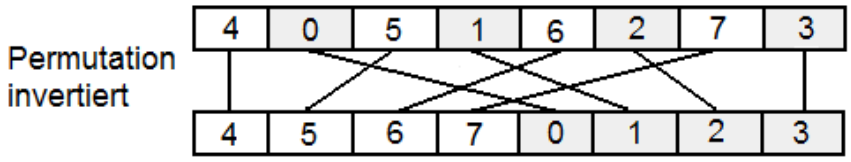

Abbildung 27: Inverse Permutation bei Wortbreite „10“

Damit steht das Ergebnis der Shuffle-Operation am Ausgang bereit. Die Chipfläche, die für die Permutationen zusätzlich aufgewendet werden muss, wird dadurch wieder eingespart, dass für Schiebeoperationen im Register

und das eigentliche Mischen nur eine einzige Wortbreite verwendet werden muss. Dadurch kommt diese Implementierung auf weniger benötigte Chipfläche (1135 Slices und 2383 LUTs auf dem Test-FPGA XC3S700A bei K=16) als alle anderen getesteten Verfahren und führt den Mischvorgang unabhängig von der Wortbreite in vier Takten durch.

Interessant wäre auch die Implementierung der Shuffle-Operation, in der das Mischen in 8-Bit-Einheiten innerhalb der 32-Bit-Vektor-Scheiben abgewickelt wird. Logik in den Vektor-Scheiben kann aufgrund des geringeren Umfangs vom Synthesetool besser optimiert werden als Logik, die ganze Vektorregister als Ein- bzw. Ausgabe verwendet. Dies konnte jedoch leider aus zeitlichen Gründen nicht mehr umgesetzt werden.

4.7 Validierung der Komponenten

Die Validierung eines Systems ist ein wichtiger Entwicklungsbestandteil und sollte so früh wie möglich begonnen werden.

Mittels des in Abschitt 2.2 vorgestellten Verfahrens zur Validierung von VHDL-Schaltungen konnten die einzelnen Komponenten des Prozessors bereits untersucht werden, als das System als Ganzes noch nicht lauffähig war.

Dadurch wurde sichergestellt, dass Fehler, die beim Testen des Gesamtsystems aufgetreten sind, nicht durch das Innenleben fehlerhafter Komponenten, sondern durch das Zusammenschalten entstanden sind.

5 Testumgebung und Entwicklungswerkzeuge

5.1 Testumgebung

Da der Prozessor alleine nicht lauffähig ist, musste zusätzlich Speicher bereit gestellt werden, in dem Programme, Zwischen- und Endergebnisse abgelegt werden können. Der erzeugte Speicher wird von der Speicherschnittstelle angesteuert und liegt, aus Gründen der Zeitersparnis, nur als SRAM Implementierung vor.

Diese Implementierung basiert auf einem 32 Bit breiten Array. Die Länge des Arrays wird so gewählt, dass sowohl das Programm als auch Zwischenwerte und Ergebnisse dort Platz finden. Da nur synchron auf die Daten zugegriffen wird, kann vom Synthese-Werkzeug automatisch das Block-RAM eines FPGAs für den Speicher verwendet werden. Allerdings ist es auch möglich das SRAM in LUTs abzubilden, um das Block-RAM für die Multiplizierer aufzusparen.

Es ist auch denkbar, anstatt echtem Speicher zusätzlich sogenannte „Memory-Mapped-IOs“ anzusprechen. Das bedeutet, dass sich hinter bestimmten Speicheradressen kein Speicher verbirgt, sondern Ein- bzw. Ausgänge des Chips. Damit könnte beispielsweise abgefragt werden, ob ein Taster gedrückt wurde oder es könnte sich ein LCD-Display ansteuern lassen. Diese Möglichkeit kann im „Top-Level-Modul“ des Systems geschaffen werden, indem anhand der Adresse des Speicherzugriffs entschieden wird, ob die Anfrage an das SRAM-Modul oder an die Ein- bzw. Ausgabesignale gerichtet ist.

5.2 Assembler

Mit Hilfe des Assemblers lassen sich Assemblerprogramme, wie sie in diesem Dokument an einigen Stellen zu finden sind, in Maschinenprogramme umsetzen. Der Assembler wurde mit der Programmiersprache Python geschrieben und ist somit plattformunabhängig.

Der Assembler unterstützt neben dem kompletten, in Abschnitt 3.2.1 aufgelisteten Befehlssatz auch die Assembler-Direktiven „ORG“, „EQU“ und „DC“.

Mit „ORG“ lässt sich der Assembler anweisen, die folgenden Maschinenbefehle oder Werte ab einer bestimmten Adresse abzulegen bzw. Sprungmarken dort zu setzen. Damit ist es insbesondere möglich, größere Bereiche im Speicher für Zwischenergebnisse frei zu halten.

```
V_DATA1: ORG $10 ; Marke V_DATA1 bei Adresse 0x10 setzen
V_DATA2: ORG $20 ; Marke V_DATA2 bei Adresse 0x20 setzen
```

Die Anweisung „EQU“ entspricht der Definition einer Konstanten, die dann im Programm als Direktwert verwendet werden kann.

```
EQU K 8      ; Konstante K mit Wert 8 definieren
SUB 0, A, K  ; Register A mit 8 vergleichen
```

Mit Hilfe von „DC“ können Werte direkt im Speicher abgelegt werden. Das ist z. B. dann notwendig, wenn ein benötigter Wert zu groß für einen Direktwert im Befehlswort ist.

```
MASK_FF: DC 255  ; Wert 0xFF an Marke MASK_FF ablegen
```

Direktwerte und Konstanten werden entweder als Dezimalzahl oder durch Voranstellen eines $-Symbols im Hexadezimal- bzw. eines %-Symbols im Binärformat angegeben.

Bei Aufruf des Assemblers lässt sich als Parameter angeben, in welches Ausgabeformat das Programm übersetzt werden soll.

- *symboltable*: Es wird eine Symboltabelle erstellt, die von dem Debugger eingebunden werden kann, der im nächsten Abschnitt vorgestellt wird.
- *coe*: Es wird eine .COE-Datei erstellt, welche dazu verwendet werden kann, vom Xilinx-Core-Generator instanzierte Speicherbausteine zu initialisieren.
- *vhdl*: Erstellt den VHDL-Code eines kompletten SRAM-Bausteins, der das Programm enthält. Dieser VHDL-Code kann unverändert in das SRAM-Modul des Systems kopiert werden.
- *bin*: Das Programm wird in Maschinenbefehle im Binärformat übersetzt. Die enstehende .BIN-Datei kann mit Hilfe des Debuggers in den Speicher des Systems transferiert werden.

5.3 Debugger

Ein Softwareentwickler setzt bei einem System voraus, darauf nicht nur Programme ausführen zu können, sondern auch eine Rückmeldung über den Verlauf zu bekommen. Es ist wichtig, herausfinden zu können, an welchen Stellen und warum ein Fehler aufgetreten ist. Während ein Entwickler für die PC-Plattform bereits unzählige Werkzeuge zur Verfügung hat, müssen für eingebettete Systeme oftmals selbst Möglichkeiten gefunden werden, an die gewünschten Informationen zu gelangen.

Aus diesem Grund wurde eine On-Chip-Debugging-Erweiterung als zusätzliche Hardwarekomponente (Debugger) in das System integriert. Der Debugger stellt den Vermittler zwischen dem Prozessor und einer seriellen (RS232)

Schnittstelle dar. Über diese Schnittstelle können vom einem anderen Computersystem Befehle an den Debugger gesendet werden. Dieser reagiert auf die Befehle, indem er die gewünschten Aktionen durchführt und anschließend eine Antwort zurücksendet.

5.3.1 Generierung der Taktsignale

Wurde der Prozessor mit Debugger synthetisiert, ist er nach dem Einschalten in einem Zustand, in dem jeder Takt explizit über die serielle Schnittstelle vorgegeben werden muss. Über eine Debugger-Anweisung besteht aber die Möglichkeit, den Prozessor in einen Freilaufzustand zu versetzen. Das bedeutet, es wird nicht mehr auf Takt-Befehle gewartet, sondern von der Debugger-Einheit werden so lange automatisch Taktsignale generiert, bis über die Schnittstelle der Befehl zum Verlassen des Freilaufzustands empfangen wird. Die Frequenz des Taktes, der am Debugger anliegt, muss 50 MHz betragen, da die Länge der Signale für die Kommunikation über die RS232-Schnittstelle daraus abgeleitet werden. Im Freilaufzustand wird der Prozessor mit der Hälfte dieser Frequenz (25 MHz) angesprochen. Ähnlich dem Freilaufzustand unterstützt der Debugger noch einen Zustand, in dem er so lange selbstständig Takte generiert, bis der Befehlszähler einen vorgegebenen Wert erreicht hat. Auch hierbei beträgt die Taktfrequenz 25 MHz.

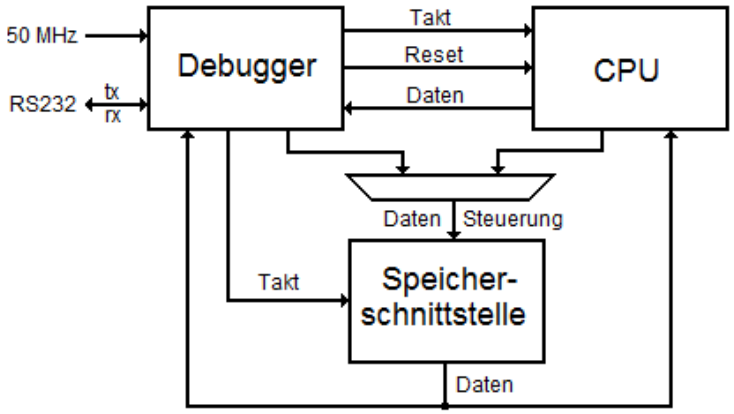

Abbildung 28: Einbindung des Debuggers

5.3.2 Speicheranbindung

Da der Debugger auf den Speicher zugreift, existiert für die Speicherschnittstelle inklusive Speicher eine eigene Taktleitung, die unabhängig vom Rest des Systems geschaltet werden kann. Dies ist nötig, damit der Zugriff auf

den Speicher keinen Einfluss auf die Ablaufsteuerung oder Daten innerhalb der CPU bewirkt. Will der Debugger die Speicherschnittstelle ansprechen, muss zunächst die Operation fertiggestellt werden, die die CPU begonnen hat. Während dieser Zeit werden nur Taktsignale für die Speicherschnittstelle und den Speicher selbst erzeugt. Anschließend führt der Debugger seine Anfrage durch. Ist diese abgeschlossen, müsste wieder der Zustand hergestellt werden, den die Speicherschnittstelle vor Eingriff des Debuggers hatte. Die Wiederherstellung ist aber sehr aufwändig, da der interne Zustand der Speicherschnittstelle nach außen nicht bekannt ist. Stattdessen sorgt der Debugger dafür, dass der Zugriff auf den Speicher abgeschlossen ist, sobald die Kontrolle darüber wieder an die CPU zurückgegeben wird. Dadurch kann die CPU in jedem Fall weiterarbeiten, auch wenn unter Umständen einige Takte beim Speicherzugriff eingespart wurden.

5.3.3 Kommunikation und Befehle

Die Befehle sind als 8-Bit-ASCII kodiert; die Antwort darauf folgt entweder ebenfalls in diesem Format oder als 32-Bit-Zahlenwert mit 8-Bit-Checksumme (XOR), aufgeteilt auf fünf Pakete. Bei manchen Befehlen wird zusätzlich zu dem Befehl im ASCII-Format noch ein 32-Bit-Wert ohne Checksumme mit übertragen. Eine Checksumme ist in diesem Fall nicht nötig, da der Debugger jedes eingehende Byte auch als Antwort zurücksendet.

Befehl	**Antwort**	**Aktion**
E	E	Echo, bestätigt die Bereitschaft des Debuggers.
C	C	Erzeugt ein Taktsignal für das System.
R	R	Erzeugt ein Reset- gleichzeitig mit einem Taktsignal.
G	G	Versetzt den Prozessor in einen Freilaufzustand.
P	P	Beendet den Freilaufzustand.
F	F	Gibt den aktuellen Wert der Flags *carry*, *zero*, *halted*, *mem_access*, *mem_ready* und *ir_ready* zurück.

A	Register A	Gibt den Wert des Registers A zurück.
X	Register X	Gibt den Wert des Registers X zurück.
Y	Register Y	Gibt den Wert des Registers Y zurück.
I	Befehlsregister	Gibt den Wert des Befehlsregisters zurück.
J	Befehlszähler	Gibt den Wert des Befehlszählers zurück.
M	Speicherbus	Gibt den Wert zurück, der aktuell am Speicherbus anliegt.
N	Speicheradresse	Gibt die Adresse zurück, die aktuell am Speicher anliegt.
0 + Adresse	0 + Adresse	Überträgt eine 32-Bit-Speicheradresse an den Debugger.
1 + Wert	0 + Wert	Überträgt einen 32-Bit-Wert an den Debugger.
2	Wert	Liefert den Wert an der zuvor übertragenen Speicheradresse zurück.
3	3	Schreibt den zuvor übertragenen Wert an die zuvor übertragene Adresse im Speicher.
4	4	Starte Taktzählung: Takte, bis Befehlszähler die zuvor übertragene Adresse erreicht hat.
5	5	Breche Taktzählung ab.
6	0 bzw. 1	Gebe den Status der Taktzählung zurück.

7	Anzahl Takte	Gebe die Anzahl an gezählten Takten zurück.

Tabelle 3: Befehle für die On-Chip-Debugging-Hardware

5.3.4 PC-Software

Die Softwarekomponente des Debuggers für den Entwicklungsrechner wurde, wie auch der Assembler, in der Programmiersprache Python realisiert. Der Anwender kann darin mittels einer Eingabeaufforderung Befehle absetzen. Die Debugger-Software unterstützt alle in der vorherigen Tabelle aufgeführten Befehle. Zusätzlich stehen aber auch Aktionen zur Verfügung, die durch Kombination dieser elementaren Operationen abgewickelt werden.

Befehl	**Aktion**
STEP *Anzahl*	Taktet den Prozessor so lange, bis *Anzahl* komplette Instruktionen abgearbeitet wurden.
JUMP *Adresse*	Taktet den Prozssor so lange, bis der Befehlszähler den Wert *Adresse* erreicht hat.
TRACE *Anzahl*	Gibt die Werte des Befehlszählers während der letzten *Anzahl* Instruktionen aus.
PROFILE	Zeigt an, welche Instruktionen (gruppiert) seit dem letzten Reset wie oft (absolut + prozentuell) vom Prozessor ausgeführt wurden.
STATUS	Listet alle abrufbaren Daten auf.
SYMBOLS	Listet alle bekannten Symbole und die zugehörigen Werte auf. Symbole können beim *JUMP*-Befehl verwendet werden. Die Symbole werden aus einer Datei importiert, die beim Programmstart als Parameter übergeben wird.
READ *Adresse*	Liest den Wert an *Adresse* aus dem Speicher.

WRITE *Adresse Wert*	Schreibt den *Wert* an *Adresse* im Speicher.
UPLOAD *Adresse*	Überträgt eine Datei in den Speicher ab *Adresse*.
DOWNLOAD *Adresse Anzahl*	Liest *Anzahl* Datenwörter aus dem Speicher ab *Adresse* und speichert diese in einer Datei ab.
CCSTART *Adresse*	Starte Taktzählung (in Hardware): Takte, bis der Befehlszähler den Wert *Adresse* erreicht hat.
CCSTATUS	Zeige Status bzw. Ergebnis der Taktzählung (in Hardware) an.
CCSTOP	Breche Taktzählung (in Hardware) ab.

Tabelle 4: Unterstützte Befehle der Debugging-Software

Wird der Prozessor ohne Debugger synthetisiert, müssen auszuführende Programme in den VHDL Quellen mittels Initialisierungswert des Speicher-Arrays fest eingetragen werden und lassen sich nach der Synthese nicht mehr verändern. Eine Speicher-Initialisierung bei der Synthese ist zwar auch mit integriertem Debugger noch möglich, jedoch ist es für die Entwicklung neuer Anwendungen wesentlich einfacher und schneller, Programme über die Upload-Funktion des Debuggers zu übertragen. Prinzipiell steht damit auch der Weg zu anderen Speichertechnologien als dem internen Block-RAM offen.

5.4 Validierung des Gesamtsystems

Das Gesamtsystem wurde zunächst mittels VHDL-Testbenches validiert. Der Prozessor wurde zusammen mit Arbeitsspeicher simuliert, in welchem sich einfache Programme befanden. Ob der Prozessor richtig gearbeitet hat, konnte anhand der Inhalte der Datenregister und der Statusflags (Carry, Zero) erkannt werden.

Da die Simulation des Prozessors aber eventuell auftretende Timing-Probleme verbirgt, musste dieser in echter Hardware realisiert und getestet werden. Der Prozessor wurde für den Xilinx Spartan 3A (XC3S700A) FPGA synthetisiert und auf einem solchen Chip aufgespielt. Mit Hilfe des On-Chip-

Debuggers konnten verschiedene Programme ausgeführt und die Arbeitsweise nachvollzogen werden.

Um den Prozessor nach Änderungen schnell und einfach validieren zu können, wurde ein Programm erstellt, welches alle möglichen Befehle ausführt und Ergebnisse der Operationen auf Gültigkeit überprüft. Dies wurde folgendermaßen realisiert:

Zunächst wird im Programm die korrekte Arbeitsweise von bedingten Sprüngen, unbedingten Sprüngen und Befehlen zur Modifikation des Statusregisters überprüft. Tritt an einer Stelle ein Fehler auf, läuft der Prozessor automatisch in einen Befehl, der ihn in den Haltezustand versetzt.

Sind diese Schritte erfolgreich abgeschlossen, fährt das Programm mit der Überprüfung der Lade- und Speicheroperation fort. Anschließend werden die Ergebnisse von ALU-Operationen der Skalareinheit auf Gültigkeit überprüft, indem sie mit Referenzwerten aus dem Speicher oder mit einem Direktwert verglichen werden.

Mit Hilfe der bereits überprüften Funktionalität validiert das Programm die korrekte Arbeitsweise der restlichen Befehle. Die korrekten Ergebnisse für Vektor-ALU-Operationen liegen nicht vorberechnet im Speicher. Stattdessen wird die skalare ALU dazu verwendet, Datum für Datum ein zweites Mal zu berechnen und die Ergebnisse mit denen der Vektoreinheit zu vergleichen.

Validiert wird der Prozessor, indem das Programm an die Konfiguration des Prozessors angepasst, übersetzt und per Debugger-Software in den Speicher geladen wird. Da, je nach Konfiguration, viele tausend Befehle ausgeführt werden, ist es am sinnvollsten, den Prozessor die Anweisungen im Freilaufzustand (siehe Abschnitt 5.3) abarbeiten zu lassen. Ob das Programm erfolgreich durchlaufen wurde ist daran erkennbar, dass der Befehlszähler stabil den Wert der Adresse des Programmendes („FINISH“-Symbol) annimmt.

5.5 Verwendete Ressourcen

- Xilinx ISE WebPACK 9.2i
- Xilinx Spartan-3A FPGA Starter Kit
- Python 2.5
- Motorola 68000 Simulator EASy68K
- Standard-PC mit Windows-Vista als Entwicklungsrechner
- Mehrere Desktop-Computer mit Linux Kernel 2.6 für Messungen zum Leistungsvergleich

6 Ergebnisse

6.1 Synthese-Ergebnisse in Abhängigkeit der Konfiguration

Um die Auswirkungen in Zahlen fassen zu können, wurde der Prozessor mit verschiedenen Konfigurationen für den Xilinx-Spartan3-FPGA XC3S1500 synthetisiert und die Anzahl an benötigten Slices ermittelt.

K	Slices	LUTs	Slices/K	LUTs/K
2	1345	2549	Bezugspunkt	Bezugspunkt
4	2021	3749	338,0	600,0
8	3256	6186	318,5	606,2
12	4517	8557	317,2	600,8
16	5788	10944	317,4	599,6
20	7019	13270	315,2	595,6
24	8360	15674	318,9	596,6
28	9735	18168	322,7	600,7
32	11002	20375	321,9	594,2
36	12269	22860	321,3	597,4

Tabelle 5: Abhängigkeit der Anzahl an Slices und LUTs von K

N	Slices	LUTs	Slices/N	LUTs/N
2	2204	4044	Bezugspunkt	Bezugspunkt
4	2985	4569	390,5	262,5
8	3969	5576	294,2	255,3
12	5238	7223	303,4	317,9
16	6255	7703	298,4	261,4
20	7187	8520	276,8	248,7
24	8033	9064	265,0	228,2
28	9015	9898	262,0	225,2
32	8604	7949	213,3	130,2
36	10556	10753	245,7	197,2
40	11426	11323	242,7	191,6

Tabelle 6: Abhängigkeit der Anzahl an Slices und LUTs von N

- Die Daten aus Tabelle 5 belegen, dass durch das Erhöhen der Anzahl an Wörtern (K) je Vektorregister um den Wert Eins, die Anzahl an benötigten Slices für die Vektoreinheit auf dem oben genannten FPGA um durchschnittlich 321 und die Anzahl an LUTs um 599 steigt. (Anmerkung: Die Spalten „Slices“ und „LUTs“ beziehen sich auf das Gesamtsystem.) Diese hohen Werte ergeben sich daraus, dass für jedes Wort eine eigene 32-Bit-Scheibe instanziert wird und Multiplexerstrukturen in der Select-Unit bzw. der Shuffle-Unit größer werden.

Vernachlässigt man den nicht von K abhängigen Teil der Vektoreinheit (z. B. deren Steuerwerk), steigt die Chipfläche damit zumindest im getesteten Bereich (K=4...36) in etwa linear mit K an. Zu beachten ist, dass für K wegen dem 64-Bit-Modus des Prozessors nur gerade Werte verwendet werden dürfen. Eine Erhöhung um den Wert Eins ist damit also nur rechnerisch möglich.

Getestet wurde mit folgender Konfiguration: Shuffle mit maximaler Breite aktiviert - Vektorshift mit 32 Bit Breite aktiviert - 8 Vektorregister - keine Multiplizierer - Optimierung auf Geschwindigkeit

- Der Einfluss, den die Veränderung der Anzahl an Vektorregistern (N) hat, ist stark abhängig von der Realisierung der Register bei der Synthese. Hier bieten sich bei FPGAs die beiden Möglichkeiten, Register im Block-RAM oder mit Hilfe vieler LUTs zu erzeugen (Parameter bei der Synthese). Im ersten Fall erhöht sich die Anzahl an benötigten LUTs und Slices nur minimal mit N, dafür werden aber wertvolle Block-RAM Ressourcen belegt. Im zweiten Fall ist dies genau umgekehrt. Tabelle 6 zeigt, dass die Zunahme an Slices und LUTs nicht linear mit N steigt sondern bei höheren Werten umgerechnet weniger Ressourcen dafür belegt werden. Die Anzahl an benötigten Slices pro Vektorregister bewegt sich im Bereich von 391 bis 213, die Anzahl an LUTs zwischen 318 und 130. Zudem zeigen die Ergebnisse bei N=32, dass durch günstige Wahl der Parameter eine sehr gute Optimierung erreicht werden kann.

  Getestet wurde mit folgender Konfiguration: 8 Wörter je Vektorregister - Shuffle deaktiviert - Vektorshift deaktiviert - keine Multiplizierer - Optimierung auf Geschwindigkeit - deaktivierte „RAM-Extraction“.

- Der Multiplizierer in der ALU der Skalareinheit kann ein- bzw. ausgeschaltet werden, da dieser, selbst wenn der Ziel-FPGA Hardware-Multiplizierer bietet, eine erhebliche Verlängerung der Signallaufzeiten im Prozessor verursachen kann. Sollten allerdings öfter Multiplikationen durchgeführt werden müssen, kann ein System mit Multiplizierer trotz der geringeren Taktfrequenz wesentlich schneller sein, da Multiplikationen per Software viele Befehle nötig machen. Das Aktivieren des Multiplizierers in einer Konfiguration für den oben genannten FPGA resultiert in einem Einbruch der Taktfrequenz um ca. 6,5 MHz.

- Problematisch bei Multiplizierern für die ALUs der Vektoreinheit ist, dass die Hardware-Multiplizierer in vielen FPGAs mit dem Block-RAM assoziiert sind und der Block-RAM deswegen nicht komplett für andere Komponenten (Arbeitsspeicher, Vektorregister) verwendet werden kann. Zusätzlich gilt hier aber auch, dass Multiplizierer die Signallaufzeiten erhöhen und deswegen die mögliche Taktfrequenz re-

duziert wird.

- Das Mischen bzw. Umsortieren von Vektorregistern erfordert große Multiplexerstrukturen. Das bedeutet, in Abhängigkeit von K kann die Unterstützung des Shuffle-Befehls sehr viel Chipfläche in Anspruch nehmen. Eine Reduzierung der Fläche kann durch Einschränken der Bit-Breite erreicht werden. In einer Testkonfiguration benötigt die Shuffle-Einheit bei K=16 und voller Bit-Breite 1408 Slices und 2970 LUTs. Bei Halbierung der Bit-Breite werden noch 767 Slices und 1457 LUTs belegt.
- Die Möglichkeit, Datenwörter im Vektorregister nach links oder rechts zu verschieben, beansprucht bei K=16 488 Slices und 1034 LUTs. Die Anzahl der Bits, um die verschoben wird, hat auf die benötigte Chipfläche keinen Einfluss.
- Die Debugging-Unit, mit der der Prozessor über eine RS232-Schnittstelle getaktet und Werte von Registern, Flags und dem Speicherbus abgefragt werden können, kann per Konfiguration ein- bzw. ausgeschaltet werden. Die Chipfläche für den Prozessor steigt durch die Debugging-Unit um einen festen Wert an. Ausschalten des Debuggers ist zum einen sinnvoll, wenn der Prozessor in Betrieb genommen wird und selbstständig arbeiten soll, zum anderen aber auch, um ihn mit einem VHDL-Simulator validieren zu können.

6.2 Leistungsanalyse

Um die Leistungsfähigkeit des Prozessors gegenüber anderen Architekturen zu ermitteln wurde eine einfache Form der Leistungsbewertung angewandt. Dabei handelt es sich um die Implementierung und den Vergleich der Laufzeit einer Beispielanwendung für den hier entwickelten Prozessor sowie moderner Dektop-Computer und einem Motorola 68000 Mikrocontroller.

6.2.1 Beispielanwendung

In der Beispielanwendung wird ein 1600x1600 Pixel großes Bild durch einen Filter im Orts-Raum geglättet.

Der eingesetzte Filterkern ist in Abbildung 29 dargestellt.

Für die Filterung wird jedem Pixel im Bild ein Wert zugewiesen. Dieser Wert ergibt sich aus der gewichteten Summe der Werte des Pixels selbst und seiner acht benachbarten Pixel im Ursprungsbild. Die Gewichtung hängt dabei von der Position und dem dafür vorgesehenen Wert im Filterkern ab.

$\frac{1}{16}$

1	2	1
2	4	2
1	2	1

Abbildung 29: 3x3 Filterkern

Soll beispielsweise der neue Wert des zentralen Pixel aus Abbildung 30 bestimmt werden, passiert dies durch folgende Rechnung:

$$\text{Neuer Wert} = \frac{1*30+2*16+1*12+2*34+4*18+2*18+1*22+2*16+1*16}{16} = 20$$

30	16	12
34	**18**	18
22	16	16

Abbildung 30: Beispieldaten für Bildverarbeitungsfilter

Die Filtermaske wurde so ausgewählt, dass Multiplikationen bzw. Divisionen im Programm durch Schiebebefehle ersetzt werden können. Der Arbeitsspeicher für den Prozessor ist auf dem FPGA als Block-RAM realisiert und teilt sich somit Ressourcen mit den Hardware-Multiplizieren. Aus diesem Grund konnte leider keine Unterstützung für Multiplikationen in Hardware für den Test-FPGA aufgebracht werden. Da Schiebeoperationen und Multiplikationen beim hier entwickelten Prozessor die selbe Anzahl an Takten benötigen, entsteht dadurch allerdings auch keine Verfälschung der Ergebnisse.

Die Anwendung ist gut für SIMD-Prozessoren parallelisierbar. Anstatt einem Pixel pro Schleifendurchlauf zu berechnen, können jeweils K-2 Pixel gleichzeitig von der Vektoreinheit verarbeitet werden. K-2 ergibt sich daraus, dass der erste und letzte Pixel im Vektorregister ungültige Werte enthält, da die Daten für die Bildpunkte links und rechts des zentralen Pixels um ein Wort im Vektorregister verschoben werden müssen.

Pixel werden in der Beispielanwendung als 32-Bit-Ganzzahl interpretiert, da der entwickelte Prozessor bisher über keine Sättigungsarithmetik verfügt. Ein realistischerer Ansatz wäre es, wenn jeder Pixel aus vier, jeweils mit 8-bit-kodierten, Farbwerten bestehen und die Verarbeitung auf Basis der Wortlänge „Byte“ erfolgen würde.

Leider sind auch im parallelisierten Algorithmus ca. 40 Prozent aller Befehle nicht vektorisiert, sondern werden allein von der Skalareinheit abgearbeitet.

Dabei handelt es sich um Kontrollstrukturen für Schleifen und die Notwendigkeit, Pixel am Bildschirmrand zu spiegeln.

Der im Block-RAM abgelegte Arbeitsspeicher für den hier entwickelten Prozessor ist nicht groß genug, um ein komplettes 1600x1600-Pixel-Bild aufzunehmen. Um die Leistungsbewertung dennoch durchführen zu können, wird lediglich eine einzige Zeile des Bildes im Speicher abgelegt. Der Algorithmus filtert dann das Bild so, als würde jede Zeile des Bildes die selben Daten wie die Zeile im Speicher enthalten. Dadurch ergibt sich die gleiche Laufzeit, die auch beim Filtern eines richtigen Bildes entstanden wäre. Beim Motorola 68000 wurde ebenfalls diese Methodik angewandt.

Die Beispielanwendung wurde für den konfigurierbaren Vektorprozessor und den Motorola 68000 als Assemblerprogramm und für die Dektop-Prozessoren mit der Programmiersprache C geschrieben. Das C-Programm ist mit dem Compiler GCC auf Laufzeit (GCC-Flag -O3) und für den jeweiligen Prozessortyp (GCC-Flag -mtune=*CPU-Typ*) optimiert worden. MMX, 3DNow!, SSE oder der zweite Prozessor des Intel Core 2 Duos wurden vom Compiler nicht zur Beschleunigung der Anwendung eingesetzt.

6.2.2 Ergebnisse der Leistungsanalyse

Um die Ergebnisse vergleichen zu können, wurde die Beispielanwendung auf verschiedenen Systemen ausgeführt und erfasst, wie lange das Filtern eines 1600x1600-Pixel-Bildes dauert. Die Messungen für die Desktop-Prozessoren wurden unter Linux mit Kernel 2.6 durchgeführt, beim Motorola 68000 und dem hier entwickelten Prozessor war kein Betriebssystem aktiv. Aus der gemessenen Zeit konnte die Anzahl an Takten errechnet werden, die die Prozessoren für die Verarbeitung eines Bildpunktes benötigen. In der folgenden Tabelle ist außerdem aufgeführt, mit welcher Taktfrequenz die getesteten Prozessoren betrieben werden und aus wie vielen Transistoren bzw. FPGA-Ressourcen sie aufgebaut sind.

Prozessor	**Transistoren bzw. Ressourcen**	**Taktfrequenz**	**Dauer**	**Takte je Pixel**
Intel Core 2 Duo	167 Mil. Transistoren	2200 MHz	0,07 s	ca. 58
AMD Athlon64	68,5 Mil. Transistoren	2200 MHz	0.08 s	ca. 65
Intel Pentium III	28,1 Mil. Transistoren	800 MHz	0,29 s	ca. 92

Motorola 68000 (EASy68K-Simulator)	70000 Transistoren	16 MHz	49,6 s	ca. 310
Konfigurierbarer Vektorprozessor (K=20)	5783 Slices, 10171 LUTs (XCS700A)	50 MHz	1,1s	ca. 21

Tabelle 7: Vergleich mit anderen Prozessoren

Die benötigte Anzahl an Takten für die Berechnung eines Pixels mit dem konfigurierbaren Vektorprozessor kann in Abhängigkeit von K mit einer Formel abgeschätzt werden. Dabei ist zu beachten, dass damit nur die maximal mögliche Leistung berechnet werden kann. Bei großen Werten für K besteht zunehmend die Gefahr, wertvolle Rechenzeit wegen nur teilweise mit Nutzdaten gefüllten Registern zu verschwenden. Aus diesem Grund sollte K so gewählt werden, dass die Anzahl an Bildpunkten je Zeile durch K-2 ohne Rest teilbar ist.

Takte je Pixel $= \frac{342+2*K}{K-2}$

Daraus ergeben sich bei einer Taktfrequenz von 50 MHz diese Werte:

K	**Dauer**	**Takte je Pixel**
20	1,09 s	21,2
40	0,57 s	11,1
80	0,33 s	6,4
160	0,22 s	4,2

Tabelle 8: Auswirkung von k

Das bedeutet, dass ungefähr ab $K = 90$ die Geschwindigkeit eines Pentium III mit 800 MHz erreicht werden könnte. Allerdings ist dieser Vergleich nicht unbedingt fair gegenüber den Desktop-Prozessoren. Der Pentium III könnte ebenfalls seine SIMD Befehle einsetzen und dadurch die Geschwindigkeit der Verarbeitung steigern. Auch der Umstand, dass die Anwendung für die Desktop-Prozessoren mit der Programmiersprache C geschrieben wurde

spricht für den Pentium. Programme, die direkt in Assemblersprache implementiert wurden, sind im Regelfall performanter.

Die unterschiedlichen Speichertechnologien machen einen exakten Vergleich der verschiedenen Plattformen nur schwer möglich. Die Geschwindigkeit der Bildverarbeitungsanwendung hängt auch sehr stark von der Geschwindigkeit der Speicherzugriffe ab. Desktop-Prozessoren werden mit viel höheren Taktraten als externe SDRAM-Bausteine betrieben. Das sorgt dafür, dass Zugriffe auf den Speicher verhältnismäßig viel Zeit benötigen, wenn sich die Daten nicht bereits im Cache befinden. Da für den hier entwickelten Prozessor der Speicher im Block-RAM des FPGAs untergebracht ist, kann dieser mit der vollen Taktfrequenz der CPU angesprochen werden. Allerdings ist diese Taktfrequenz mit beispielsweise 50 MHz immer noch wesentlich langsamer als die 100 MHz, mit der der getestete Pentium III auf seinen Speicher zugreifen kann.

7 Zusammenfassung und Ausblicke

7.1 Ergebnis dieser Arbeit

In dieser Diplomarbeit wurde ein neuer Prozessor entwickelt, zu dem es im Open-Source-Bereich momentan nichts Vergleichbares gibt. Der dabei entstandene Befehlssatz ist Turing-vollständig und unabhängig von der Breite oder Anzahl der Vektorregister.

Der Prozessor ist validiert und in realer Hardware lauffähig. Dies wurde mit Hilfe von VHDL-Testbenches und einem Xilinx-FPGA überprüft.

Neben dem Prozessor selbst wurde ein Assembler und ein On-Chip-Debugger geschaffen. Damit stehen die grundlegenden Software-Werkzeuge zur Verfügung, welche für die Entwicklung von Anwendungen notwendig sind.

Anhand eines Beispielprogrammes aus dem Bildverarbeitungsbereich wurde die Leistungsfähigkeit untersucht und gezeigt, dass auf dem konfigurierbaren Prozessor tatsächlich reale Anwendungen ausgeführt werden können.

7.2 Mögliche Erweiterungen

Leider war es im begrenzten Zeitraum nicht möglich, alle offenen Ideen für den hier vorgestellten Prozessor auch umzusetzen. Seine Entwicklung muss aber mit Abgabe dieser Arbeit nicht zwingend enden. Er wird voraussichtlich als Open-Source-Projekt im Internet veröffentlicht und kann somit beliebig eingesetzt und weiterentwickelt werden.

Ein Punkt, der für neue Anwendungsmöglichkeiten des Prozessors sorgen würde, wäre die Erweiterung, dass Unterbrechungen unterstützt werden. Dadurch könnte man den Prozessor auch in einem Umfeld einsetzen, in dem schnelle Reaktionen auf Ereignisse wichtig ist.

Der Vergleich mit anderen SIMD-Architekturen zeigt weitere Stellen auf, an denen der Prozessor noch erweitert werden kann. So benutzt er bei Vektoroperationen keine Sättigungsarithmetik. Das bedeutet, Überläufe der Wertebereiche sind in vielen Situationen wahrscheinlich und müssen durch die Anwendung abgefangen werden. Aber auch im Bereich von „horizontalen“ Operationen oder der Verarbeitung von Fließkommawerten kann noch viel von diesen Architekturen gelernt werden.

Interessant wäre sicherlich auch die Entwicklung und Leistungsanalyse eines Prozessors, der auf demselben Befehlssatz operiert, dessen Struktur aber pipelineartig aufgebaut ist. Eine Pipeline im skalaren Teil des Prozessors würde den Durchsatz an skalaren Befehlen pro Takt erhöhen, während bei

einer pipelineartig aufgebauten Vektoreinheit unter Umständen eine höhere Parallelität bei gleicher Chipfläche erreicht werden könnte.

Auch außerhalb der CPU gibt es noch Bereiche, in denen Verbesserungen eingebracht werden können. Beispielsweise kann die Speicherschnittstelle bisher nur den internen Block-RAMs verwenden. Da der Prozessor aber dafür konzipiert ist, Mediendaten zu verarbeiten, wäre es sehr sinnvoll, auch auf externe SDRAM-Bausteine mit hoher Speicherkapazität zugreifen zu können.

Dennoch liegt am Ende des Zeitraus dieser Diplomarbeit ein funktionierender Prozessor vor, der im Open-Source-Bereich bisher seinesgleichen sucht.

Literaturverzeichnis

[Fly72] M.J. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, Vol. c-21, No. 9, pp. 948-960, 1972

[Mär94] C. Märtin: Rechnerarchtitektur, Carl Hanser Verlag, 1994

[Mär03] C. Märtin: Einführung in die Rechnerarchitektur, Carl Hanser Verlag, 2003

[Men05] M. Menge: Moderne Prozessorarchitekturen, Springer-Verlag, 2005

[Gon02] R.C. Gonzalez, R.E. Woods: Digital Image Processing, Prentice Hall, 2002

[Sie01] C. Siemers: Hardwaremodellierung, Carl Hanser Verlag, 2001

[Dud01] Duden Informatik, Bibliographisches Institut F.A. Brockhaus AG, 2001

[Hen03] J.L. Hennessy, D.A. Patterson, Computer Architecture - A Quantitative Approach, Morgan Kaufmann Publishers, 2003

[Sto00] J. Stokes, SIMD architectures, http://arstechnica.com/articles/paedia/cpu/simd.ars/, 2000, zuletzt aufgerufen am 10.12.2007

[Int97] Intel Technology Journal, MMX Technology Architecture Overview, http://download.intel.com/technology/itj/q31997/pdf/archite.pdf, zuletzt aufgerufen am 10.12.2007

[Int99] Intel Technology Journal, The Internet Streaming SIMD Extensions, http://download.intel.com/technology/itj/Q21999/PDF/simd_ext.pdf

[Int04] Prescott New Instructions Software Developer's Guide, http://cache-www.intel.com/cd/00/00/06/67/66753_66753.pdf, zuletzt aufgerufen am 10.12.2007

[iwMMXt] Intel XScale Technology: Intel Wireless MMX 2 Coprocessor, http://download.intel.com/design/intelxscale/31451001.pdf, zuletzt aufgerufen am 17.12.2007

[Int06] Processors - Define SSE2 and SSE3, http://www.intel.com/support/processors/sb/cs-001650.htm, zuletzt aufgerufen am 17.12.2007

[AMD07] AMD64 Technology - 128-Bit SSE5 Instruction Set, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/43479.pdf, zuletzt aufgerufen am 17.12.2007

[Int07] Intel SSE4 Programming Reference,

http://softwarecommunity.intel.com/isn/Downloads/Intel SSE4 Programming Reference.pdf, zuletzt aufgerufen am 17.12.2007

[Bro05] J. Brown, Application-customized CPU design - The Microsoft Xbox 360 CPU story,
http://www-128.ibm.com/developerworks/power/library/pa-fpfxbox/?ca=dgr-lnxw09XBoxDesign, zuletzt aufgerufen am 17.12.2007

[Ful98] S. Fuller, Motorola's AltiVec Technology,
http://www.freescale.com/files/32bit/doc/fact_sheet/ALTIVECWP.pdf, zuletzt aufgerufen am 17.12.2007

[Neon] ARM NEON Technology, http://www.arm.com/miscPDFs/6629.pdf, zuletzt aufgerufen am 10.12.2007

[Gui07] P. Guironnet de Massas, P. Amblard, F. Pétrot, On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration, VLSI Design, Vol. 2007, Hindawi Publishing Corporation

[Sha04] J. Shafer, Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression,
http://www.jeffshafer.com/publications/shafer-masters_thesis.pdf, 2004, zuletzt aufgerufen am 10.12.2007

Abbildungsverzeichnis