Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Realisierung der virtuellen Hardware-Maschine in VHDL

Diplomarbeit

Nick Bierwisch

Leipzig, den 5. Juni 2003

Zusammenfassung

Die Verarbeitung von Anwendungen mittels Software ist ein sehr flexibler und mächtiger Weg, jedoch ist der Energieverbrauch sehr hoch aber die Leistung nicht dementsprechend. Eine Realisierung der selben Anwendung in Hardware, falls möglich, erlaubt meist eine schnellere Abarbeitung bei gleichzeitig wesentlich niedriger Leistungsaufnahme. Jedoch sind Hardwarerealisierungen nicht sehr flexibel. Dieser Nachteil wurde vermindert, indem rekonfigurierbare Hardware, wie field programmable gate arrays(FPGAs), entwickelt wurde. Ihre grosse Anzahl und der unterschiedliche Aufbau verhindern jedoch eine standardisierte Schnittstelle. Um nicht für jedes FPGA eine Schaltung komplett neu zu entwickeln, wurde in der Diplomarbeit von Sebastian Lange[Lan02] die virtuelle Hardware-Maschine eingeführt. Sie stellt eine Mittelschicht zwischen der zugrunde liegenden Architektur und der Schaltungsentwicklung dar. In dieser Arbeit wurde versucht eine lauffähige Version dieser VHM zu erstellen. Dafür wurde in VHDL das Verhalten beschrieben und ein Design synthetisiert, welches dann auf einem FPGA getestet wurde. Da eine vollständige Implementierung zwar erstellt, aber nicht zum Laufen gebracht werden konnte, lassen sich keine Geschwindigkeiten messen. Es können nur Abschätzungen der Geschwindigkeit und Aussagen über den Ressourcenverbrauch der Implementierung getroffen werden. Diese Arbeit beschreibt, wie die VHM implementiert wurde und trifft einige Aussagen über die zu erwartende Geschwindigkeit und den Ressourcenverbrauch einer solchen Implementierung gegenüber den bisher benutzten direkten Implementierungen der Schaltungen auf den verschiedenen FPGAs.

Danksagung

Hiermit möchte ich mich bei Prof. Dr. Udo Kebschull für die Möglichkeit diese Arbeit zu schreiben und seine Unterstützung dabei bedanken.
Ebenso danke ich meinem Betreuer Sebastian Lange für seine Unterstützung und die ständige Bereitschaft zu helfen.
Den restlichen Mitarbeitern der Abteilung Technische Informatik an der Universität Leipzig gilt ebenfalls mein Dank für die Hilfe bei vielen kleineren Problemen mit den benötigten Hard- und Softwarekomponenten.
Als letztes Danke ich meinen Eltern und meinen Geschwistern für die Unterstützung und das Verständnis auf meinem Weg durch das Studium.

Inhaltsverzeichnis

1 Einleitung/Motivation

Durch das Aufkommen rekonfigurierbarer Hardware wie programmierbaren Logikbausteinen(PLD[1]s) und field programmable gate arrays(FPGAs) ist es jetzt auch möglich die Hardwarelogik selbst, nicht nur die Softwarekomponenten, während der Laufzeit zu verändern. Diese Anpassungen können für den Benutzer völlig transparent vorgenommen werden. Das bedeutet, dass der Benutzer nicht wissen muss, dass die Hardware neu konfiguriert wird und auch keinerlei Schritte unternehmen muss, um diesen Prozess in Gang zu setzen beziehungsweise, ihn zu steuern. Um verschiedenen Einheiten die Möglichkeit zu geben aufeinander einzuwirken, sollten Hardwarekomponenten austauschbar sein, ohne von der zugrunde liegenden Architektur abhängig zu sein. Die große Vielfalt der FPGAs und anderen allgemeinen programmierbaren Geräten führt zu einem Fehlen einer standardisierten Schnittstelle, was es problematisch macht, Logik auf diese Weise zu definieren.

Die zunehmende Bedeutung von Geräten mit niedrigem Stromverbrauch ist ein weiterer wichtiger Aspekt. Das Auftreten der drahtlosen Kommunikation und die Verbreitung von Handheld Computern(PDA[2]s) erschuf einen neuen Bereich. Es wird viel Rechenleistung benötigt, um Informationen zu verarbeiten oder zu beschaffen und das wann immer und wo immer es nötig ist. Um diese Leistung zur Verfügung zu stellen, kann der Einsatz spezialisierter Hardware in Betracht gezogen werden. Die alleinige Ausführung in Software würde immer höhere Taktfrequenzen erfordern, welche den Energieverbrauch drastisch erhöhen würden. Allerdings kann die Hardware immer nur ganz bestimmte Aufgaben bearbeiten.

Der virtuelle Hardware-Bytecode(VHBC), dessen Entwicklung in der Arbeit[Lan02] von Sebastian Lange beschrieben wird, liefert Mittel, die die Tugenden der Hardwareentwicklung mit der Flexibilität der Software kombinieren. Schaltungen werden als Hardware entworfen, aber auf einer speziellen Hardwarearchitektur, welche als virtuelle Hardware-Maschine bezeichnet wird, von einem speziellen Hardware-Bytecode Compiler abgebildet. Die virtuelle Hardware-Maschine wurde entworfen, um auf praktisch jeder

[1] programmable logic device
[2] personal digital assistent

zugrunde liegenden Hardwarearchitektur leicht eingeführt zu werden und dient als ein Vermittler zwischen einer abstrakten algorithmischen Beschreibung und der Hardware selbst.

1.1 Eingebettete Systeme

Einer der bekanntesten Begriffe auf dem Gebiet des Hardwaredesigns ist der des eingebetteten Systems. Er bezeichnet einen spezifischen entworfenen Bestandteil, welcher selbstständig innerhalb eines komplexen Systems arbeitet. Folglich werden eingebettete Systeme speziell, für die Aufgaben die sie später bearbeiten sollen, entwickelt. Durch das Erstellen von Blöcken aus eingebetteten Systemen, können die Entwickler komplexere Systeme erstellen. Solche Systeme sollten in der Lage sein, genau die benötigte Leistung für den niedrigsten Preis zur Verfügung zu stellen.

Als nächstes stellt sich die Frage, welche Teile eines Systems in Hardware realisiert werden und wofür Standardprozessoren benutzt werden. Einige der daraus resultierenden Eigenschaften sind die reale Größe der Lösung, die Taktfrequenz, die Leistungsaufnahme sowie die Zeit bis zur Markteinführung und die Kosten der Implementierung. Sehr kritische Werte waren schon immer die Größe eines Designs und dessen Kosten. Neue Technologien erlauben zwar das Benutzen immer kleinerer Komponenten, jedoch werden die Systeme im Gegenzug auch immer komplexer.

Heutzutage wird bei der Realisierung einer Anwendung in Hardware meist eine anwendungsspezifische integrierte Schaltung(ASIC[3]) entworfen. Hardwarerealisierungen neigen zu einer schnelleren Abarbeitung als ihre Softwaregegenstücke, da ein System das auf seine wesentlichen Teile begrenzt wird eine hohe Effizienz erlaubt, daraus resultierend eine niedrigere Leistungsaufnahme entsteht und eine höherer Parallelitätsgrad ausgenutzt wird. Wird ein ASIC in Massenproduktion hergestellt, sind die Produktionskosten kaum höher als die Summe der Kosten der einzelnen Komponenten. Dies und die hohe Wahrscheinlichkeit, durch die hohe Verbreitung des Designs, Konstruktionsfehler zu erkennen, macht ein ASIC zu einem idealen Baustein komplexer Systeme. Den geringen Produktionskosten und der hohen Geschwindigkeit stehen jedoch der Verbrauch des Siliziums und die Entwicklungskosten gegenüber. Ein ASIC kann immer nur genau eine Aufgabe erfüllen. Eine Änderung der Anwendung erfordert eine Neuentwicklung des ASIC. Das ist das Hauptproblem von anwendungsspezifischen integrierten Schaltungen. Ihr Einsatz lohnt sich also vor allem für Anwendungen die vielfach benutzt werden sollen und an denen keinerlei Änderungen vorgenommen werden müssen.

[3] application specific integrated circuit

Die Geräte schrumpfen immer weiter in ihrer Größe, während ihre Komplexität weiter wächst und immer mehr Funktionen auf der gleichen begrenzten Menge an Ressourcen untergebracht werden sollen. Wie oben beschrieben passt sich Hardware nicht gut an verschiedene Anforderungen an, aber die Benutzer erwarten Geräte, welche Sie über einen langen Zeitraum benutzen können. Durch die schnell voranschreitende Entwicklung entsteht hier ein Problem. Neue Standards entstehen und nicht alle Teile können leicht diesen Änderungen angepasst werden.

Eine andere Methode zum Einführen von mehr Flexibilität ist die Benutzung von Software. Es wird eine abstrakte Beschreibung der Funktionalität von einem Prozessor zur Laufzeit ausgewertet und abgearbeitet. Diese abstrakte, nichtphysikalische Beschreibung der Funktionalität bietet einige Vorteile gegenüber Hardwarerealisierungen und die virtuelle Natur von Software erlaubt das Ersetzen von alten Versionen sowie den Austausch von Software zwischen kompatiblen Systemen zur Laufzeit. Dadurch können erkannte Fehler auf eine sehr leichte und bequeme Art behoben werden. Software wird normalerweise mit den höheren Programmiersprachen beschrieben, die den Prozess des Softwaredesigns weiter von zugrunde liegenden Hardwareplattformen entziehen und sich auf reine Funktionalität konzentrieren. Sobald ein Algorithmus in einer höheren Programmiersprache wie C beschrieben worden ist, kann er in einer Menge von Lösungen auf unterschiedlichen Systemen entfaltet werden. Software-Designs sind folglich extrem flexibel und in hohem Grade anpassbar. Der Fokus der Software auf Funktionalität anstatt physikalischer Eigenschaften lässt eine einfachere Beschreibung komplexer Systeme zu. Software hat sich folglich als Designplattform der Wahl für viele Systemplaner entwickelt, einen Markt gleichmäßig starker und produktiver Entwicklungswerkzeuge sowie eine ausgedehnte Wissensbasis innerhalb der Technikwelt aufgebaut. Die Flexibilität und die Anpassungsfähigkeit von Software hat aber auch ihren Preis. Weil Software-Designs nur virtuelle Beschreibungen von Funktionalität sind, ist eine Vorrichtung erforderlich, die die Konzepte verwirklicht. Solch eine Vorrichtung wird Prozessor genannt und ist in sich eine ziemlich komplizierte Maschine. Ein beträchtlicher Teil der vorhandenen Hardwareressourcen muss folglich für die Dekodierung der Software-Beschreibung benutzt werden, anstatt sich mit der Verarbeitung von Daten zu beschäftigen. Dies erfordert hohe Taktfrequenzen und daraus folgend eine höhere Leistungsaufnahme. Softwarebeschreibungen werden in atomare Teile zerlegt, welche Anweisungen genannt werden. Diese Anweisungen werden mit dem festgelegten Taktzyklus an die Funktionseinheiten des Prozessors verteilt. Beim Hardwaredesign kann aber die vorhandene Parallelität sehr viel besser ausgenutzt werden und somit der Energieverbrauch und die Taktfrequenz viel niedriger gehalten werden.

Software kann in zwei Kategorien, entsprechend dem spezifischen Zielgebiet der Prozessoren, die sie ausführen, aufgeteilt werden. Einerseits existieren universelle Prozessoren, die optimiert werden, um eine hohe Leistung für alle möglichen Softwaredesigns zur Verfügung zu stellen. Folglich wird der Aspekt der besseren parallelen Durchführung häufig gegen eine breitere Anzahl der untergestützten Operationen getauscht. In eingebetteten Systemen, die gewöhnlich für die Signalaufbereitung benutzt werden, müssen die Daten jedoch schnell verarbeitet werden. Das bedeutet, dass die eintreffenden Informationen innerhalb einer bestimmten Zeitspanne verarbeitet sein müssen, um ein Funktionieren des Systems zu gewährleisten. Deshalb wurden spezielle Prozessoren entworfen um die Aufgaben der Signalverarbeitung zu erfüllen. Diese Prozessoren werden Signalprozessoren(DSP[4]) genannt. Sie besitzen gewöhnlich Datenwege, die spezifisch entworfen werden, um schnelle und in hohem Grade parallele Verarbeitungen zuzulassen.

Der Gebrauch von Software um Funktionalität zu beschreiben ist sehr leistungsfähig. Er liefert eine große Abstraktion der Beschreibung des Gerätes, welches benutzt wird, um die Aufgabe auszuführen. Dadurch kann Software außerdem wiederholt ausgeführt und geändert werden und ebenso in Systemen verteilt werden. Die niedrige Effektivität der Softwaredesigns bei Leistung und Energieverbrauch im Vergleich zu Hardware wie ASICs führte zur Entwicklung von Hardware, welche ein Netz einfacher Hardwarekomponenten bereitstellt, die konfiguriert werden können, um bestimmte logische Funktionen auszuführen. Solche Vorrichtungen werden im Allgemeinen PLDs genannt und besitzen den Vorteil der schnellen Durchführung, welche fast im Bereich von ASICs liegt. PLDs erlauben die Spezifikation des Hardwaredesigns in einem transportfähigen Format. Diese Designs können folglich geändert werden und lassen eine Flexibilität der ausführbaren Anwendungen zu, die fast genauso hoch wie die von Software ist.

Tabelle 1.1 vergleicht die beschrieben Entwicklungsmöglichkeiten hinsichtlich ihrer Rechenleistung, der Anpassungsfähigkeit und dem Energieverbrauch.

	CPU	DSP	FPGA	ASIC
Rechenleistung	begrenzt	$\geq$ CPU	$\approx$ ASIC	sehr hoch
Anpassungsfähigkeit	sehr hoch	hoch	begrenzt	keine
Leistungsaufnahme	sehr hoch	hoch	$\leq$ DSP	niedrig

Tabelle 1.1: Vergleich bestehender Entwicklungsplattformen

[4]digital signal processor

1.2 Virtuelle Hardware

Virtuelle Hardware wurde eingeführt, da die Ressourcen, die durch rekonfigurierbare Hardware zur Verfügung gestellt wurden, nicht mehr ausreichten. Durch die Einführung von PLDs und FPGAs, welche ihre Funktionalität während der Laufzeit schnell ändern können, war es möglich, immer mehr Teile eines Systems durch Hardwarerealisierungen zu beschleunigen. Das Prinzip der virtuellen Hardware gleicht dem Prinzip des virtuellen Speichers in Computersystemen. Wenn zu wenig Speicher zur Verfügung stand, wurden nicht benutzte Teile des Speichers auf langsamere statische Speicherkomponenten(meist die Festplatte) ausgelagert. Dafür wurde der Speicher in Seiten aufgeteilt. Bei rekonfigurierbarer Hardware bedeutet dies die Virtualisierung der Konfigurationsdaten. Diese werden in virtuelle Hardware Seiten[5] aufgeteilt und in den externen Konfigurations-RAM[6]s abgelegt. Diese können mehrere Konfigurationen speichern. Dadurch kann zur Laufzeit die Funktionalität verändert werden und so ein viel größerer Aufgabenbereich, als es die vorhandenen Ressourcen eigentlich erlauben, emuliert werden. Es können aber ebenfalls auch Designs erstellt werden die sehr flexibel sein müssen, in dem einfach die ausgelagerten Seiten geändert werden. Wie schon in Abschnitt 1 beschrieben ist keine einheitliche Schnittstelle zu FPGAs vorhanden. Da die Konfigurationsdaten stark mit dem physikalischen Aufbau der FPGAs gekoppelt sind, ist eine Nutzung eines anderen FPGAs oft mit dem Benutzen einer anderen Technologie verbunden. Außerdem ist durch die unterschiedliche Anzahl der externen Ausgänge und deren räumlichen festgelegten Positionen sowie der Benutzung von höherer Logik, wie vollständiger CPUs auf dem Xilinx Virtex Pro, eine direkte Übersetzung der Konfigurationsdaten der einzelnen FPGAs nicht möglich. Ein Neuanlegen der Konfigurationsdaten kann mit der jetzigen Technologie leicht mehrere Stunden dauern und das Vorhalten verschiedener Versionen für jedes FPGA ist aufgrund der hohen Anzahl von field programmable gate arrays nicht praktikabel. Erforderlich wäre also eine allgemeine, bewegliche Beschreibung der Hardwareeinheiten, die in den laufenden Systemen benutzt werden könnte und es so eingebetteten Systemen erlaubt, die leistungsfähigen Vorteile des rekonfigurierbaren Rechnens und besonders der virtuellen Hardware auszunutzen.

[5] virtual hardware pages
[6] random access memory

1.3 Aufgabenstellung der Arbeit

Ziel dieser Arbeit ist es eine Implementierung der virtuellen Hardware-Maschine, beschrieben in der Diplomarbeit von Sebastian Lange[Lan02], zu erstellen. Dies geschieht indem in VHDL eine Beschreibung der Funktionsweise erarbeitet wird. Diese wir d dann synthethisiert und kann anschließend auf einem FPGA abgebildet und getestet werden.

1.4 Der Ansatz und die Ziele

Durch die Einführung der virtuellen Hardware-Maschine wird eine neue Schicht zwischen der Schaltungsbeschreibung und der Abbildung der Funktionalität auf der entsprechenden Hardware geschaffen. Dadurch soll es möglich werden, eine für einen bestimmten Algorithmus oder eine bestimmte Aufgabe, entwickelte Schaltung auf verschiedenen Hardwareressourcen zu nutzen. Durch die stark unterschiedlichen Konfigurationsdaten der verschiedenen FPGAs ist eine direkte Umsetzung der verschiedenen Formate nicht möglich. Um nicht für jedes verwendete FPGA alle, meist sehr zeitaufwendige, Schritte des Schaltungsdesign durchführen zu müssen, wird die virtuelle Hardware-Maschine als Mittelschicht eingesetzt. Die entwickelte Schaltung wird durch den virtual Hardware-Bytecode(VHBC) beschrieben und dann in der virtuellen Hardware-Maschine auf dem jeweiligen FPGA ausgeführt.

1.5 Was wurde im Rahmen der Diplomarbeit erreicht?

Das Design der virtuellen Hardware-Maschine(VHM) wurde in VHDL erstellt. Es wurde flexibel gestaltet um die VHM verschiedenen Aufgabengebieten anzupassen. So lassen sich der maximale Parallelitätsgrad, die maximale Verarbeitungstiefe und die Anzahl der Datenregister leicht anpassen. Dadurch kann die VHM die vorhandenen Hardwareressourcen besser ausnutzen und den zu erwartenden Anwendungen angepasst werden. Die Verarbeitungsgeschwindigkeit der VHM lässt sich abschätzen. Genau gemessen werden konnte sie nicht, da zum Zeitpunkt der Tests kein Logic Analyzer zur Verfügung stand. Die komplette VHM lies sich auf dem FPGA nicht zum laufen bringen, so das die Geschwindigkeit des Dekodiervorgangs nicht bestimmt werden konnte. In Kapitel 7 sind einige Aussagen über die zu erwartenden Geschwindigkeiten und die Größe der VHM im Vergleich zu direkten Implementierungen getroffen.

2 Stand der Technik

2.1 Hardware Emulatoren

Um während der Entwicklung einer Schaltung deren Korrektheit zu überprüfen und eine Aussage über die zu erwartende Geschwindigkeit zu treffen, wurden bisher 2 Methoden benutzt. Die eine ist die Entwicklung eines Prototypen, welche sehr kostenintensiv ist und einen langen Zeitraum in Anspruch nimmt. Simulation mittels Software ist der andere Weg. Im Gegensatz zu einem Prototypen ist die Softwaresimulation sehr flexibel, aber auch sehr langsam. Um genaue Aussagen über das Verhalten der Schaltung zu bekommen, müssen viele Schritte simuliert werden. Dies erfordert dann sehr viel Zeit. Um die Kosten gering zu halten und die Zeit bis zur Markteinführung[1] ebenfalls zu senken, wird versucht die Vorteile beider Methoden zu vereinen.

Ein Weg ist ein Hardware Emulator. Er stellt eine konfigurierbare Plattform dar, auf der die Schaltung emuliert werden kann. Meist bestehen diese Emulatoren aus vielen FPGAs auf denen die zu entwickelnde Schaltung abgebildet und dann getestet werden kann. Durch die Verwendung der FPGAs ist der Emulator ähnlich flexibel wie die Simulation mittels Software, kann aber mit wesentlich höherer Geschwindigkeit ausgeführt werden. Die Entwicklungskosten werden auch erheblich gesenkt da der Emulator, im Gegensatz zum Prototypen, angepasst und immer wieder verwendet werden kann. Der Emulator agiert wie das zu emulierende Gerät bzw. die emulierende Schaltung, wodurch der Anwender ihn wie das 'echte' Gerät benutzen kann. Verschiedene Programme und Abläufe können so simuliert werden als benutze man das Gerät selbst. Nur die Geschwindigkeit der Ausführung ist geringer, da hier rekonfigurierbare Hardware und nicht spezialisierte Komponenten benutzt werden.

Ein paar Hardware Emulatoren werden im Folgenden kurz vorgestellt. Genauere Informationen können in der angegebenen Literatur gefunden werden.

[1] time to market

2.1.1 RPM

RPM steht für Rapid Prototyping engine for Multiprocessors und wurde entwickelt um verschiedene MIMD-Systeme mit bis zu 8 Prozessoren zu emulieren. 9 Boards bilden die RPM, wobei 8 jeweils einen Prozessor emulieren und das neunte Board ist für die Ein- und Ausgaben sowie die Programmierung der FPGAs zuständig. Dieses ist über einen SCSI-Bus mit einen Host-Rechner - einen SUN SPARC - verbunden. Die Emulationsboards sind mit einen 64-Bit FUTUREBUS+ verbunden. Auf jedem dieser Boards befindet sich ebenfalls ein SPARC-Prozessor. Die FPGAs auf den Boards, Xilinx XC4013 Chips mit 192 Ein-/Ausgabeverbindungen, werden nur als Speichercontroller oder Cache benutzt. Dadurch lassen sich verschiedene Multiprozessormodelle, wie CC-NUMA[2],NCC-NUMA[3],COMA[4],MPS[5], emulieren. Näheres zu den verschiedenen Architekturen findet im Vortrag von Mikko Lipasti[Lip02]. Die Homepage des RPM-Projektes mit einer genaueren Beschreibung der Architektur, Ergebnissen und Information zur Erweiterung RPM-2, kann man hier[Dub96] erreichen.

2.1.2 Mentor Graphics

Die Firma Mentor Graphics[6] bietet 2 Lösungen zur Entwicklungsbeschleunigung mit Hardwareemulatoren an. Einige Eigenschaften der beiden Emulatoren VStation und Celaro werden in den folgenden Abschnitten vorgestellt. Auf der Homepage der Firma kann man unter den Punkten *Functional Verification-> Emulation* mehr über die Emulatoren und andere Entwicklungshilfen erfahren. Genauere Literaturangaben werden in den folgenden Abschnitten angegeben.

VStation

Weltweit wurden seit 1999 mehr als 100 VStation Systeme installiert. Damit ist VStation einer der erfolgreichsten Emulatoren auf dem Markt. Es gibt verschiedene Generationen der VStation, wobei ich hier nur etwas näher auf die 5. Generation, die VStation-30M, eingehen werde. Diese VStation besteht aus bis zu 30 Millionen Gattern, 288 MByte Speicher und 4608 Ein-/Ausgabeverbindungen. Diese Zahl wird erreicht, wenn das System aus der maximalen Anzahl von 9 Boards besteht. Der Emulator kann mit einer Geschwindigkeit von 500 kHz bis 2 MHz die Schaltungen emulieren. Dies entspricht einer Steigerung gegenüber Softwaresimulationen von dem 10000 bis 100000-fachen. Ein

[2] cache coherent non-uniform memory access
[3] non cache coherent non uniform memory access
[4] cache only memory architecture
[5] message passing system
[6] siehe www.mentor.com

Hochgeschwindigkeitscompiler, der bis zu 3 Millionen Gatter pro Stunde verarbeiten kann, verkürzt zusätzlich die Simulationszeit. Bevorzugt verbunden ist VStation mit einem SUN Blade 1000 Rechner über einen 16 Bit SCSI Bus. An diesem Rechner kann der Entwickler die Schaltung überprüfen, Änderungen vornehmen und einzelne Signale zur Laufzeit beobachten. Durch die VirtualWires Technologie können auch Systeme mit mehr als den vom Emulator gelieferten Ein- bzw. Ausgängen emuliert werden. Genauere Informationen zur VStation findet man auf der Website der Firma Mentor unter [Div03b].

CelaroPro

CelaroPro besitzt einen Compiler der bis zu 4 Million Gatter pro Stunde verarbeiten kann. Allerdings kann dieser Prozess auf mehrere Arbeitsrechner, wenn diese vorhanden sind, verteilt werden, um eine höhere Verarbeitungsgeschwindigkeit zu erreichen. Die Emulationsgeschwindigkeit kann bei Celaro bis zu 2 MHz erreichen. Insgesamt können bis zu 30 Millionen Gatter emuliert werden, wobei diese für bis zu 4 unabhängige Modelle benutzt werden. So können gleichzeitig mehrere Nutzer ihre Schaltungen emulieren. Durch eine sehr hohe Speicherbestückung von bis zu 128MB pro Board lassen sich sehr leicht verschiedene Speichermodelle erstellen und emulieren. Im Datasheet unter [Div03a] lassen sich genauere Informationen nachlesen.

2.2 Virtual FPGA

Die physikalischen Vorraussetzungen, genauer gesagt die Anzahl der Logikblöcke und der Ein- und Ausgänge, von FPGAs sind begrenzt. Das Ziel ist es ein virtuelles Gerät einzuführen, dass die Eigenschaften des FPGAs abstrahiert und aus der Sicht der Anwendung noch bessere Eigenschaften besitzt. Dieses Modell wird als virtuelles FPGA(virtual FPGA) bezeichnet und ist vergleichbar mit dem Prinzip der virtuellen Hardware(1.2). Wie es Betriebssysteme bei der Speicherverwaltung in Multitaskingsystemen tun, wird das FPGA virtualisiert, indem immer die jeweilige, von dem aktuellen Anwendungsteil benötigte, Konfiguration in das FPGA geladen wird. Es kann somit vom Betriebssystem wie jede andere geteilte Hardwarekomponente behandelt werden. Um die Anwendungsprogrammierung zu vereinfachen, sollte jeder gerade ausgeführte Programmteil das virtuelle FPGA als vollständig zu sich selbst zugeordnet betrachten. Dies erspart dem Programmierer die notwendige Synchronisation zwischen den einzelnen Programmteilen die das FPGA benutzen, überträgt diese Aufgabe aber dem Betriebssystem, wie es auch bei den anderen geteilt genutzten Hardwareressourcen der Fall ist. Der Anwendungsentwickler und der Programmierer können sich nun auf die für Anwendung wichtigen

Aspekte der spezifischen Aufgabe konzentrieren, ohne sich um die weiteren Aufgaben sowie die Probleme zu kümmern, die nicht auf die Anwendung bezogen sind, sondern auf die Architektur, welche die Anwendung selbst laufen lässt.

Es wird versucht die 2 typischen Ziele für geteilte Hardwareressourcen - eine abstrakte Ansicht der Hardwarekomponente(normalerweise mit besseren Eigenschaften als die reale) und eine virtuelle Ansicht der Komponente, welche jeder einzelnen Aufgabe vollständig zugeordnet ist - zu erreichen.

Um dies zu erreichen wurden einige, aus der Literatur über Betriebssysteme bekannte, Konzepte auf FPGAs übertragen.

dynamisches Laden: veranlasst das Laden der vom aktuell ausgeführten Programmteil benötigten Konfiguration bei einem Systemaufruf oder sobald der Programmteil gestartet wird

Aufteilung: teilt die Funktions-Logikblöcke des FPGA in Gruppen, für die das Betriebssystem unabhängig voneinander die Konfiguration, für die entsprechenden Gruppen von Aufgaben, laden kann

Bedeckung: erstellt Teile des FPGA, um allgemeine Funktionen zu berechnen, die häufig verwendet werden, während der restliche Teil benutzt wird, um spezifische Funktionen zu laden, die gewöhnlich selten verwendet werden

Segmentierung: zerlegt die zu ladenden Funktionen in kleinere Teile, welche eine selbstständige Unterfunktion berechnen und verschiedene Größen erreichen können

Seitenerstellung: teilt die zu ladenden Funktionen in kleinere Teile festgelegter Größe

Ein- und Ausgangszuordnung: weist der aktuell ausgeführten logischen Funktion die Ein- und Ausgänge des bearbeiteten Programmteils zu oder erhöht die Anzahl der Ein- und Ausgangssignale, wenn physikalisch nicht genug Ressourcen vorhanden sind

Zwei dieser beschriebenen Merkmale werden in den folgenden Abschnitten näher beschrieben ,um ein besseren Einblick in die Funktionsweise eines virtuellen FPGAs zu bekommen.

2.2.1 Dynamisches Laden

In den Multitaskingsystemen müssen gleichzeitige Teilaufgaben das FPGA verwenden, um spezifische Anwendungen(normalerweise unabhängig und ohne Bezug zueinander) in Hardware auszuführen, damit die Leistung, die die entsprechenden Anwendungen benötigen, erzielt wird. In einigen Fällen kann eine Anwendung beschleunigt werden, weil unterschiedliche unabhängige Anwendungen an den unterschiedlichen Punkten der Aufgabe selbst auf dem FPGA ausgeführt werden. In anderen Fällen ist es interessant, in der Lage zu sein, einen Service-Algorithmus in Hardware für alle Aufgaben im System laufen zu lassen, indem man den gewünschten Algorithmus aus einer gegebenen Menge auswählt. Dieses ist gewöhnlich der Fall bei einem Gerätetreiber, wenn verschiedene Nutzungsmöglichkeiten vorhanden sind(z. B. encoding/decoding, compression/decompression, Netzwerkgeräte).

In beiden Fällen wird die Möglichkeit, die entsprechende Konfiguration des FPGAs zu laden, wenn die jeweilige Anwendung ausgeführt wird, benötigt. Der einfachste Fall ist ein FPGA mit genügend Ressourcen. Es werden alle Konfigurationen zu einer Schaltung zusammengefasst und die aktuell ausgeführte Anwendung betrachtet nur die interessanten Ausgänge und ignoriert die Restlichen.

Die allgemeine Lösung ist das dynamische Laden der gewünschten Konfiguration in das FPGA. Die Grundidee besteht aus dem Ändern der FPGA Konfiguration, je nachdem welche von den verschiedenen gleichzeitigen Aufgaben benötigt wird. Jede Aufgabe im System muss die Anwendungen angeben, die es im FPGA ausgeführten möchte und eine verwendbare Beschreibung für das FPGA zur Verfügung stellen(als RAM-Konfiguration ausgedrückt). Der Anwendungsentwickler muss diese Anforderungen in den Designschritten in Betracht ziehen und dem Betriebssystem die notwendigen Informationen zur Verfügung stellen, damit dieses das FPGA exakt handhaben kann, genauso wie das Betriebssystem dies für alle anderen geteilten Betriebsmittel tut. Beim Starten einer Teilaufgabe muss also die verwendete Konfiguration dem Betriebssystem mitgeteilt werden und entsprechend gespeichert werden. Dies kann durch einen speziellen Betriebssystembefehl oder durch implizites Ausführen beim Starten des Prozesses geschehen.

Wird eine Teilaufgabe dem FPGA zugeordnet wird, setzt das Betriebssystem die Umgebung so, als würde nur dieser Prozess und kein anderer das Gerät benutzen. Für das FPGA bedeutet dies, dass das Betriebssystem die gewünschte Konfiguration in das FPGA-KonfigurationsRAM lädt, indem es die Informationen verwendet, die, wie oben besprochen, in den Betriebssystemtabellen gespeichert sind. Das Betriebssystem kann nun, durch aktualisieren der entsprechenden Prozessorregister, die Teilaufgabe ausführen.

Allerdings kann die Abarbeitung dieser Teilaufgabe in einem Multitaskingsystem nicht einfach unterbrochen werden. Ist die Verarbeitung noch nicht abgeschlossen, wären alle Zwischenergebnisse verloren und die Endergebnisse würden niemals berechnet. Damit dieser Fall nicht eintritt, könnte man ein Prioritätensystem einführen oder jede FPGA Konfiguration liefert ein Signal zurück, welches anzeigt das die Verarbeitung abgeschlossen ist.

Systeme die aber eine Unterbrechung der Prozesse erlauben, müssen die Informationen, die für ein Zurücksetzen und Neustarts der Teilaufgabe erforderlich sind, zwischenspeichern, um den Prozess an einem späteren Zeitpunkt erneut ausführen zu können. Kompliziert wird diese Aufgabe, wenn die Ergebnisse der Berechnung nicht nur von den aktuellen Eingaben, sondern auch von vorherigen Eingaben und Berechnungen abhängen. In diesem Fall muss eine Möglichkeit existieren, welche den gesamten Status der Konfiguration auslesen kann. Dies erfordert schon bei der Entwicklung ein hohes Maß an Arbeit. Außerdem müssen beim Neustart der Aufgabe alle Werte auch wieder gesetzt werden können. Damit sich der Aufwand überhaupt lohnt, muss aber ebenfalls die benötigte Zeit für das Lesen und Setzen der Signale möglichst gering sein. Andernfalls wären die Reaktivierungszeiten einfach zu hoch.

Die Anwendbarkeit des Prinzips des dynamischen Ladens ist also von der möglichen Reaktivierungszeit und der Komplexität der verschiedenen Aufgaben abhängig. Bei kurzen nur einmal auszuführenden Prozeduren(z. B. Initialisierungen) lässt es sich gut anwenden. In Systemen mit vielen Aufgabenwechseln oder Multitaskingsystem mit relativ kurzen Zeitscheiben ist der Vorteil dagegen eher gering.

2.2.2 Aufteilung

Sehr häufiges dynamisches Laden der FPGA Konfiguration kann eine Managementtätigkeit werden, die für das Betriebssystem zu zeitraubend ist. Wenn die abzubildende Schaltung groß ist, kann die Konfigurationszeit im Vergleich zur Nutzungszeit der Schaltung unannehmbar hoch sein. Wann immer der Verwaltungsaufwand bedeutend wird, sollten andere Annäherungen oder sogar die Programmierung des Algorithmus mittels Software betrachtet werden.

Die drastischere Lösung, um die Verwaltungskosten zu verringern, die durch Multitasking verursacht werden, verhindert den geteilten FPGA Gebrauch. Dann kann das FPGA allerdings erst von der nächsten Teilaufgabe benutzt werden, wenn die aktuelle Aufgabe komplett verarbeitet ist. Alle Teilaufgaben werden in eine Warteschlange eingereiht. Diese exklusive Art der FPGA-Nutzung führt also zu einer FIFO[7]-Ausführung der anfallenden Aufgaben und kann somit keinerlei Parallelität ausnutzen.

Wenn zwei oder mehr Schaltungen von den Teilaufgaben der Anwendung benötigt werden und gleichzeitig auf dem FPGA untergebracht werden können, ist das Aufteilen des FPGAs eine wirkungsvolle Technik. Dadurch lässt die Anzahl der Lade- und Speichervorgänge verringern und die nutzbare Rechenzeit erhöhen, ohne die Parallelität zu beeinflussen.

Die Grundidee ist das Teilen der FPGA-Ressourcen in verschiedene unterschiedliche Mengen, eine für jeden Teil des FPGA. Das Betriebssystem muss jeden dieser Teile, unabhängig von den Anderen, mit einer von der Anwendung benötigten Schaltung konfigurieren können. Jeder Teil entspricht einer Gruppe von Speicherzellen des FPGA-Konfigurationsspeichers, welcher die Logik-Blöcke bzw. die Weiterleitungen der Signale konfiguriert.

Die Größen der verschiedenen Teile des FPGAs können gleich groß oder unterschiedlich groß sein. Sie können außerdem festgelegt oder während der Laufzeit veränderbar sein. Im ersten Fall werden die Teile durch das Betriebssystem bei der Initialisierung, eine Systemkonfigurationsdatei benutzend, angelegt. Das Erstellen eines Teiles beinhaltet das Ablegen der, dem FPGA-Teil zugehörigen, Speicherzellen in den internen Betriebssystemtabellen, um das Laden der FPGA Konfigurationen zu ermöglichen. Die Zuordnung wird nur geändert, wenn das System mit einer anderen Systemkonfiguration neu gestartet wird.

Bei einer variablen Teilgröße, können die Grenzen der einzelnen Teile so lange dynamisch verändert werden, wie diese die Durchführung eines zurzeit geladenen Algorithmus nicht beeinflussen. Beim Starten des System wird eine Standardaufteilung des FPGAs

[7] first in first out

angelegt. Es wird ein Teil, welcher das gesamte FPGA bedeckt, erstellt. Auf Anfrage des Betriebssystems wird ein unbenutzter Teil genommen und in 2 Teile aufgespaltet. Danach werden die Einträge in den Betriebssystemtabellen aktualisiert. Ist kein genügend großer unbenutzter Teil vorhanden, wird das Ausführen der Teilaufgabe verschoben, bis ein verwendbarer Teil vorhanden ist. Dies kann allerdings dazu führen das eine Aufgabe unbestimmt lange wartet, wenn keiner der vorhandenen Teile groß genug ist. Akzeptierbar ist diese Arbeitsweise, wenn ein genügend großer Teil vorhanden ist, aber von einer anderen Teilaufgabe verwendet wird. Nicht erwünscht ist dieses Warten, wenn zwar kein genügend großer Teil, aber mehrere kleinere unbenutzte Teile, vorhanden sind. Ein Verfahren zur Erkennung und dem, falls benötigt, Zusammenlegen von mehreren unbenutzten Teilen zu einem zusammenhängenden Teil wird also gebraucht. Das Verschieben von Schaltungen auf andere Teile ist eine zeitraubende Tätigkeit. Folglich kann dies nur selten angewandt werden, um die Höhe der Verwaltungskosten zu begrenzen.

Das Laden einer Schaltung, welche von der Anwendung dem FPGA zugeordnet wurde, kann ähnlich der Art und Weise wie sie beim dynamischen Laden besprochen wurde, durchgeführt werden. Eine Teilaufgabe kann den gewünschten Teil des FPGAs angeben und die Konfiguration wird in diesen Teil geladen, falls er gerade verfügbar ist. Ist dies nicht der Fall, wird das Ausführen der Teilaufgabe verschoben, bis der entsprechende Teil verfügbar ist. Bei festgelegten Größen der Teile des FPGAs können die Aufgaben statisch den einzelnen Teilen zugeordnet werden, damit nicht jede Teilaufgabe den benutzten Teil angeben muss. Wird kein spezieller Teil angegeben, übernimmt im Allgemeinen das Betriebssystem die Auswahl eines geeigneten verfügbaren Teils. Lässt sich kein Bereich finden, wird die Aufgabe verschoben bis ein geeigneter Bereich vorhanden ist. Im Falle der dynamischen Aufteilung wird dann das Verfahren zur Zusammenlegung von Teilen begonnen. Das Erstellen von neuen Teilen muss nicht durch einen Systemaufruf erfolgen, sondern kann auch immer dann ausgeführt werden, wenn eine Schaltung in einen Teil geladen wird der zu groß ist. Ebenso kann das Zusammenlegen immer implizit versucht werden, sobald eine Teilaufgabe warten muss.

Bei der Verwendung von veränderbaren Bereichen muss beachtet werden, dass sich dadurch eventuell bestimmte Einschränkungen für die Teilaufgaben ergeben. Bei fester Zuordnung können die Teilaufgaben angepasst und entsprechend kompiliert und synthetisiert werden. Bei dynamischer Aufteilung und dem eventuellen Zusammenlegen von FPGA-Teilen, ist dies nicht so einfach möglich.

2.3 Was ist an diesem Ansatz neu

Das Neue am Ansatz der virtuellen Hardware-Maschine ist der Versuch eine Verbindungsschicht zwischen der Schaltungsbeschreibung und der Technologieabbildung zu erstellen. Bisher gab es 2 Ansätze um Schaltungen in Hardware zu realisieren. Die Erstellung einer speziell angepassten Hardware[8] oder die Verwendung von FPGA[9]s. Ein ASIC kann nur genau eine Aufgabe erfüllen und für keine anderen Anwendungen wieder verwendet werden, kann aber für genau diese Aufgabe optimiert werden. FPGAs lassen sich dynamisch anpassen, um verschiedene Aufgaben zu erledigen. Es wird nur etwas Zeit zur Neukonfiguration benötigt und dann kann die neue Schaltung benutzt werden. Durch die große Anzahl verschiedener FPGAs und ihrer inkompatiblen Programmierdatensätze, ist ein hoher Aufwand für die Anpassung der Anwendungen bei einem Wechsel des FPGAs notwendig. Um diese Nachteile zu entfernen wurde die VHM entwickelt. Eine gewünschte Schaltung wird mit Hilfe des Bytecode-Compilers in eine Darstellung als VHBC[10] umgewandelt, welchen die VHM dann ausführen kann. Die VHM kann als ASIC implementiert werden oder wie es in dieser Arbeit zum Testen getan wird auf einem FPGA abgebildet werden. Werden nun verschiedene FPGAs benutzt, müssen nur für die VHM die einzelnen Schritte des Schaltungsdesigns wiederholt werden. Dieser Ansatz vereinfacht die Entwicklung und Portierung von Schaltungen, jedoch lassen sich diese nicht mehr so effizient optimieren, da der direkte Bezug zur Zielarchitektur fehlt. Die nötigen Optimierungen müssen im Hardware-Bytecode Compiler, möglichst optimale Darstellung des Algorithmus als Hardware-Bytecode, und bei der Anpassung der VHM an die benutzte Architektur geschehen. Ziel ist es, eine möglichst flexible und trotzdem effiziente Architektur zur Verarbeitung verschiedener Schaltungen zu entwickeln.

[8] ASIC - anwendungsspezifische integrierte Schaltung
[9] field programmable gate array
[10] virtual hardware bytecode

3 Vorraussetzungen

3.1 Hardware-Bytecode

Der virtual Hardware-Bytecode(VHBC) ist eine Zwischenform zur Darstellung von Schaltungen. In diesem Dokument wird er im Weiteren nur noch als Bytecode bezeichnet. Diese Darstellungsform wurde entwickelt, um Schaltungen in einer hardwareunabhängigen Form zu beschreiben. Er besteht aus dem Bytecode-Header, den Instruktionsblöcken und einem abschließenden End-of-Block-Statement. Dieses Statement besteht aus 4 Bit die jeweils auf '0' gesetzt sind. Die Werte im Header und deren Anordnung sind bekannt, so das diese immer in derselben Weise ausgelesen werden können. Nach dem Header folgt eine variable Anzahl von Instruktionsblöcken. Ein Instruktionsblock beinhaltet eine Folge von Instruktionen, die durch ein End-of-Block-Statement beendet wird. Die Reihenfolge der Instruktionen bestimmt die Zuordnung zu der jeweiligen Functional Unit(↗5.3). Das bedeutet, die 1. Instruktion im Block geht an die 1. FU und so weiter. Sind weniger Instruktionen als FUs in einem Block enthalten, werden die restlichen FUs mit einer NOP-Operation gefüllt. Ausführlich beschrieben ist der Bytecode in [Lan02].

3.1.1 Header

Im Header sind die veränderlichen Werte des Bytecodes festgelegt. Dazu gehören die Anzahl der benutzten Register, Ein-/Ausgabeports und Konstanten. Werden keine Konstanten benutzt, ist der von *constant_number* gleich 0 und das Feld *constant_pool* taucht im Header nicht auf. Alle anderen Felder sind immer im Header des Bytecodes vorhanden. Durch diese Informationen kann eine Implementierung der VHM entscheiden, ob ein spezieller Bytecode auf ihr ausführbar ist. Ist dies nicht möglich, kennzeichnet dies ein entsprechender zurückgegebener Fehlercode(↗A.4). In der Tabelle 3.1 sind die Felder des Bytecode-Headers aufgelistet. Ihre Reihenfolge entspricht dem Vorkommen im Header des Bytecodes. Die Länge eines Eintrages wird in Bit angegeben.

Feld	Länge in Bit	Beschreibung
magic	32	Kennzeichen das Bytecode folgt - 'VHBC'
Version	16	Version des Bytecode als X.Y
address_width	32	gibt die Anzahl(*bpa*) der Bits an, mit denen die Operanden adressiert werden
register_number	32	beinhaltet die Zahl der vom Bytecode benutzten Register
in_port_number	32	gibt an wie viele Eingabewerte vom Bytecode benutzt werden
out_port_number	32	gibt an wie viele Ausgabewert vom Bytecode benutzt werden
constant_offset	bpa	bezeichnet die Stelle des Registers ab dem die Konstanten gesetzt werden
constant_number	16	gibt die Anzahl(*cst_num*) der Konstanten an
constant_pool	8	gibt die Konstanten des Bytecodes an dieses Feld existiert cst_num mal im Header

Tabelle 3.1: Felder des Hardware-Bytecode Headers

3.1.2 Instruktionen

Die Instruktionen des Bytecodes bestehen aus dem Operationscode und zwei Operanden. Der OpCode besteht aus 4 Bit und die Länge der Operanden bestimmt der Wert des Feldes *address_width* im Header, welcher in diesem Dokument auch als *bpa* bezeichnet wird. Dadurch kann die Länge einer Instruktion bei verschiedenen Bytecodes unterschiedlich sein. Eine Anpassung der Instruktionen an das feste Format der VHM kann dadurch nötig sein und wird im Abschnitt 6.1 erläutert. Eine Beschreibung der Instruktionen befindet sich in Tabelle 3.2, welche auch den 4-stelligen Binärcode der Operation benennt. Dieser wird in der Ausarbeitung meist als Operationscode bzw. Opcode bezeichnet. Die Tabelle 3.2 entspricht dem Stand der Diplomarbeit von Sebastian Lange[Lan02]. Allerdings wurden bei der Realisierung einige Änderungen vorgenommen, welche im Abschnitt 5.1 näher beschrieben werden.

Opcode	Instruktion	Beschreibung
0000_2	EOB	kennzeichnet das Ende eines Instruktionsblocks
0001_2	NOP	es wird keine Operation ausgeführt
0010_2	MOV	der 1. Operand wird als Ergebnis zurückgegeben
0011_2	NOT	liefert die Negation des 1. Operanden als Ergebnis
0100_2	AND	berechnet die UND-Verknüpfung der Operanden
0101_2	NAND	liefert die negierte AND-Operation zurück
0110_2	OR	berechnet die ODER-Verknüpfung der Operanden
0111_2	NOR	gibt die Negation der ODER-Verknüpfung zurück
1000_2	XOR	führt eine exklusiv-oder-Operation aus
1001_2	EQ	überprüft die Gleichheit der Operanden
1010_2	IMP	liefert das Ergebnis der Implikation
1011_2	NIMP	liefert das Ergebnis der Negation der Implikation
1100_2	CMOV	übernimmt als Ergebnis den 2. Operanden, wenn der 1. als wahr ausgewertet wird

Tabelle 3.2: Die Instruktionen des Hardware-Bytecodes

3.2 VHDL

VHDL steht für VHSIC Hardware Description Language, wobei VHSIC als Abkürzung für Very High Speed Integrated Circuits benutzt wird. Wie der Name schon sagt, ist VHDL eine Sprache zur Beschreibung digitaler elektronischer Systeme. Man kann mit VHDL die Struktur und Funktionalität integrierter Schaltungen beschreiben. Ein gutes Buch zur Einführung in VHDL ist *The Designers Guide to VHDL* [Ash02]. Um den Umgang mit VHDL und die Codebeispiele im Anhang D besser zu verstehen, werden hier einige Befehle und Prinzipien von VHDL erläutert. Es gibt in VHDL zwei Möglichkeiten der Beschreibung eines System. Die Verhaltensbeschreibung und die Strukturbeschreibung. In dieser Arbeit wurde das Verhalten der einzelnen Komponenten beschrieben und das Abbilden des Systems auf die entsprechende Hardware dem Synthesewerkzeug überlassen. In bestimmten Fällen kann aber eine optimierte Strukturbeschreibung bestimmter Teile die Größe des Designs verringern. Daten werden in VHDL mit Signalen transportiert. Signale erhalten den zuletzt zugewiesenen Wert am Ende des Prozesses. Im Gegensatz dazu erhalten Variablen sofort den Wert und eignen sich daher zur Zwischenspeicherung von Ergebnissen. Ein Prozess beschreibt einen Teilablauf des Systems. Alle parallelen Abläufe werden in VHDL als Prozesse bezeichnet. Jeder Prozess enthält eine Sensitivitätsliste, in der alle Signale enthalten sind, die eine Abarbeitung des Prozes-

ses aktivieren. D. h. ändert sich ein Signal dieser Liste wird der Prozess durchlaufen. Existieren mehrere Prozesse, so werden erst alle abgearbeitet und dann die geänderten Signale gesetzt. Dadurch werden wieder Prozesse aktiviert und verarbeitet. So kann ein System auf die Eingaben reagieren und die beschriebenen Aufgaben bearbeiten. Ebenso wie bei anderen Sprachen existieren Anweisungen zur Auswertung von Signalen bzw. Variablen, zum Abarbeiten von Schleifen und zur Bearbeitung der Werte. Hier werden nur einige Anweisungen kurz angesprochen, da sie ähnlich zu schon bekannten Programmiersprachen aufgebaut sind. Für eine wenn-dann Verzweigung benutzt man bei VDHL das *if-then-else-endif* Konstrukt. Verzweigungen mit mehreren Möglichkeiten lassen sich durch mehrere *if-then-elsif-then-. . . -endif* oder eine *case-when-end case* Anweisung ausdrücken. Schleifen werden meist durch eine *for-loop-end loop* Anweisung beschrieben. Die Syntax der beschriebenen Befehle ist im Folgenden noch einmal aufgelistet:

Prozess: <name>: process(<Signal_1>,. . . ,<Signal_n>)
begin
⋮
end process <name>;

Signal: signal <name >: <Datentyp >;
Zuweisung: <signal_1 > <= <signal_2 >;

Variable: variable <name >: <Datentyp >;
Zuweisung: <var_1 > := <var_2 >;

Wenn-Dann: if <Bedingung >then . . . else . . . end if;
oder if <Bedingung >then
⋮
{ elsif <Bedingung >then . . . }
else . . . end if;

Case: case <Bedingung >is
when <opt_1 >=>. . .
⋮
when <opt_n >=>. . .
others =>. . .
end case;

Schleife: for <Zähler >in <Bereich >
loop
⋮
end loop;

3.3 Das Synthesewerkzeug/Die Synthese

Als Synthesewerkzeug wurde die Software XiLinx ISE 4.2 verwendet. Das zentrale Werkzeug der Software ist der Project Navigator. In ihm wird der VHDL-Code bearbeitet, das Projekt erstellt und es werden die anderen Werkzeuge gestartet. Zu diesen Werkzeugen gehören die Simulationssoftware ModelSim und die benutzten VHDL-Compiler. Im Verlauf der Diplomarbeit wurden die VHDL Compiler XST, FPGAExpress und SpectrumVHDL der Firmen Xilinx[1], Synopsys[2] und Mentor Graphics[3] benutzt. Die Arbeitsfläche des Project Navigators besteht aus 4 Bereichen. Diese sind die Konsole in der die Ausgaben der jeweiligen Kommandos angezeigt werden, das Prozessfenster in dem die einzelnen Prozesse sichtbar sind und die entsprechenden Kommandos bzw. Werkzeuge gestartet werden, das Projektfenster in dem die einzelnen Teile des Projekts und deren Zusammenhänge dargestellt werden und der Quellcode-Editor in dem die VHDL-Dateien bearbeitet werden können. Im Projektfenster wird die Hierarchie der Quellen dargestellt und die Zielarchitektur sowie der verwendete VHDL-Compiler bestimmt. In diesem Fall wird *xcv800-6bg432-XST VHDL* ausgewählt. Durch markieren einer Komponente wird festgelegt, auf welchen Teil des Projekts sich die Angaben im Prozessfenster beziehen. Der Begriff Prozess bezieht sich hier nicht auf einen Prozess in VHDL, sondern auf eine Teilaufgabe in der Entwicklung. D. h. in diesem Fenster können nun Werkzeuge gestartet werden. Unter *Design Entry Utilities->Launch ModelSim Simulator* wird die Simulationssoftware ModelSim gestartet. Unter *Synthesize* kann getestet werden, ob die entsprechende VHDL-Beschreibung synthetisierbar, d. h. in Hardware realisierbar, ist, da in VHDL auch Modelle beschrieben werden können, die sich nur simulieren lassen. Dies ist z. B. der Fall, wenn in der Beschreibung bei der Signalzuweisung Verzögerungen angegeben werden, um das Laufzeitverhalten oder den Einfluss externer Komponenten zu testen. Solche Angaben lassen sich natürlich nicht synthetisieren. Ob das erstellte Design auf den angegebenen Chip passt, also genügend Ressourcen für die Abbildung vorhanden sind, kann unter *Implement Design-> Map* getestet werden. Hier wird die Zuordnung der Ressourcen des jeweiligen Chips angegeben und es lässt sich erkennen, wie viele der vorhandenen Ressourcen bereits belegt sind. Wenn die Simulation der Beschreibung funktioniert und das Design die Grenzen der Emulationshardware nicht übersteigt kann unter *Generate Programming File* ein Bitfile erstellt werden, mit dem dann das FPGA programmiert werden kann. Die Ausgaben der Werkzeuge können werden in der Konsole angezeigt, könne aber auch in Reporten, die auf der Festplatte abgelegt werden, nachgelesen werden.

[1] siehe www.xilinx.com
[2] siehe www.synopsys.com
[3] siehe www.mentor.com

3.4 Emulationshardware

Um die VHM während der Entwicklung zu testen, wurde ein FPGA der Virtex-Reihe der Firma XiLinx benutzt. Die genaue Bezeichnung lautet XCV800BG432, das bedeutet dieser Chip besitzt 432 Ein-/Ausgabe-Pins. Eine genauere Beschreibung kann unter [Xil01] gefunden werden. Die folgende Tabelle stellt die verschiedenen FPGAs der Virtex-Reihe vor. Man sieht, dass der hier verwendete Chip relativ viele Ressourcen besitzt. Dies ermöglicht die Realisierung und den Test verschiedener Varianten der virtuellen Hardware-Maschine.

Device	System Gates	CLB Array	Logic Cells	Maximum Avail. I/O	Block RAM Bits	Maximum SelectRAM+^{TM}Bits
XCV50	57,906	16x24	1,728	180	32,768	24,576
XCV100	108,904	20x30	2,700	180	40,960	38,400
XCV150	164,674	24x36	3,888	260	49,152	55,296
XCV200	236,666	28x42	5,292	284	57,344	75,264
XCV300	322,970	32x48	6,912	316	65,536	98,304
XCV400	468,252	40x60	10,800	404	81,920	153,600
XCV600	661,111	48x72	15,552	512	98,304	221,184
XCV800	888,439	56x84	21,168	512	114,688	301,056
XCV1000	1,124,022	64x96	27,648	512	131,072	393,216

Tabelle 3.3: Virtex field programmable gate array family members aus [Xil01]

3.4.1 Architektur des Virtex FPGA

Die FPGAs der Virtex-Reihe sind aus konfigurierbaren Logikblöcken(CLB[4]s), Ein- und Ausgabeblöcken(IOB[5]s), block RAM[6]s(BRAMs) und 4 delay locked loops(DLLs) aufgebaut. Diese DLLs gleichen Verzögerungen während der Taktverteilung aus. Die einzelnen Blöcke können variabel miteinander verbunden werden. Da diese Konfiguration auf SRAMs basiert, sind unendlich viele Konfigurationen möglich. Die BRAMs dienen der Speicherung von größeren Datenmengen für die CLBs. Abbildung 3.1 stellt den Aufbau eines Virtex FPGAs dar. Sie ist aus der Virtexkonfigurationsbeschreibung[Xil00] übernommen. Hier können auch noch genauere Informationen über das verwendete FPGA und den Aufbau des Konfigurationsstroms nachgelesen werden.

Ein CLB besteht aus 4 Logikzellen, von denen jeweils 2 zu einer Scheibe(Slice) zusammengefasst werden. Jede Scheibe besteht aus 2 Funktionsgeneratoren mit jeweils

[4]configurable logic block
[5]input-output block
[6]random access memory

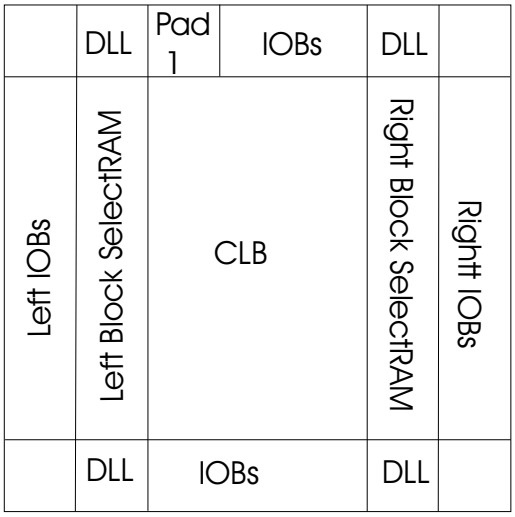

Abbildung 3.1: Aufbau des Virtex FPGA

4 Eingängen. Diese sind als Funktionstabellen(LUT[7]) implementiert, können aber auch als synchroner RAM verwendet werden. Außerdem sind 2 Register und eine Weiterleitungslogik(carry logic) enthalten, welche wie alle Elemente, durch SRAMs konfiguriert werden können. Mit mehreren Multiplexern lassen sich so die verschiedenen Funktionen realisieren. Eine genauere Beschreibung des Aufbaus der einzelnen Komponenten sowie ihre Möglichkeiten und Grenzen können der Diplomarbeit von Andreas Haase[Haa01] entnommen werden.

[7] look up table

4 Die virtuelle Hardware-Maschine

Im folgenden Kapitel wird ein allgemeines Design der virtuellen Hardware-Maschine, wie es in der Diplomarbeit von Sebastian Lange[Lan02] entwickelt wurde, vorgeschlagen, indem man das grundlegende Konzept einer Maschine vorstellt, welche in der Lage ist eine VHBC(virtual hardware bytecode)-Abbildung leistungsfähig auszuführen. Der erste Abschnitt nennt die Bestandteile der VHM, wie sie während der Definition des VHBC entwickelt wurden und der Zweite beschreibt die Funktionsweise der Komponenten. Dieses Kapitel beschreibt das Konzept einer VHM, wie es während der Entwicklung des Bytecodes entstand. Eine genaue Beschreibung der Entwicklung der Hardwarerealisierung, der tatsächlich erstellten Komponenten, sowie deren Aufbau und Funktionsweise findet man in den folgenden Kapiteln.

4.1 Die Architektur

Die VHBC-Entwicklung setzt voraus, dass Schaltungsbeschreibungen in einen Bytecode, welcher von einer speziellen Hardware - der virtuellen Hardware-Maschine - ausgeführt wird, übersetzt werden können. Das Design der VHM ist entsprechend an die Eigenschaften des Hardware-Bytecodes, vor allem den einfachen Operationen und einem hohen Maß an Parallelität, angepasst. Dies bedingt eine VLIW(very large instruction words)-ähnliche Struktur mit vielen Ausführungseinheiten. Das Konzept der VHM ist ein Allgemeines. Es zielt darauf ab, auf einer Vielzahl von zugrunde liegenden Hardwareplattformen leicht anpassungsfähig zu sein und reicht von den Standard CMOS Implementierungen bis zu unterschiedlichen rekonfigurierbaren Hardwarearchitekturen wie FPGAs. Um die vielen Ausführungseinheiten auf den begrenzten Ressourcen unterbringen zu können, müssen diese sehr einfach aufgebaut sein. Bei der VHM können die Ausführungseinheiten nur einfache logische Operationen(↗3.1.2) durchführen. Dadurch müssen zwar alle Schaltungen auf diese wenigen Funktionen abgebildet werden, aber diese können dann mit hohem Parallelitätsgrad ausgeführt werden, sofern keine Datenabhängigkeiten vorhanden sind. Außerdem sollten die Werte sehr schnell aus dem Register gelesen und in das Register geschrieben werden. Diese Ausführung muss innerhalb eines Taktes

geschehen, damit keine zusätzliche Synchronisation notwendig ist. Alle Datenabhängigkeiten und andere Probleme, wie z. B. das Kopieren von Zwischenergebnissen, müssen im VHBC-Compiler gelöst werden, damit die VHM keinerlei Überprüfungen dieser Art vornehmen muss. Die Zuordnung der Register zu den FUs ist starr festgelegt und kann nicht für bestimmte Situationen verändert werden. Jede FU kann auf alle Registerzellen lesend zugreifen, aber nur in eine Registerzelle schreiben. Diese Zuordnung besagt das FU 0 nur in Registerzelle 0 schreibt und FU n-1 nur in Registerzelle n-1. In einen Instruktionsblock des Bytecodes erhält die 1. Functional Unit(FU 0) die 1. Instruktion und so weiter. Dadurch wird auch der Aufwand für die Registerverwaltung gering gehalten, so das das Hauptaugenmerk der VHM weiter auf einer möglichst schnellen Abarbeitung der im Bytecode beschriebenen Schaltung liegt.

Um die genannten Anforderungen zu erfüllen, ergeben sich 5 Komponenten. Ein Decoder, der den Bytecode auswertet und die Operationen an die Functional Units verteilt, welche diese Instruktionen ausführen und die Daten verändern. Im Register werden diese Daten zwischengespeichert, die Eingabewerte gelesen und die Ausgabewerte ausgegeben. Der Sequenzer steuert den Ablauf und das Verbindungsnetzwerk transportiert die Daten zwischen den einzelnen Komponenten. Diesen Aufbau stellt die Grafik 4.1 aus der Diplomarbeit von Sebastian Lange[Lan02] noch einmal dar.

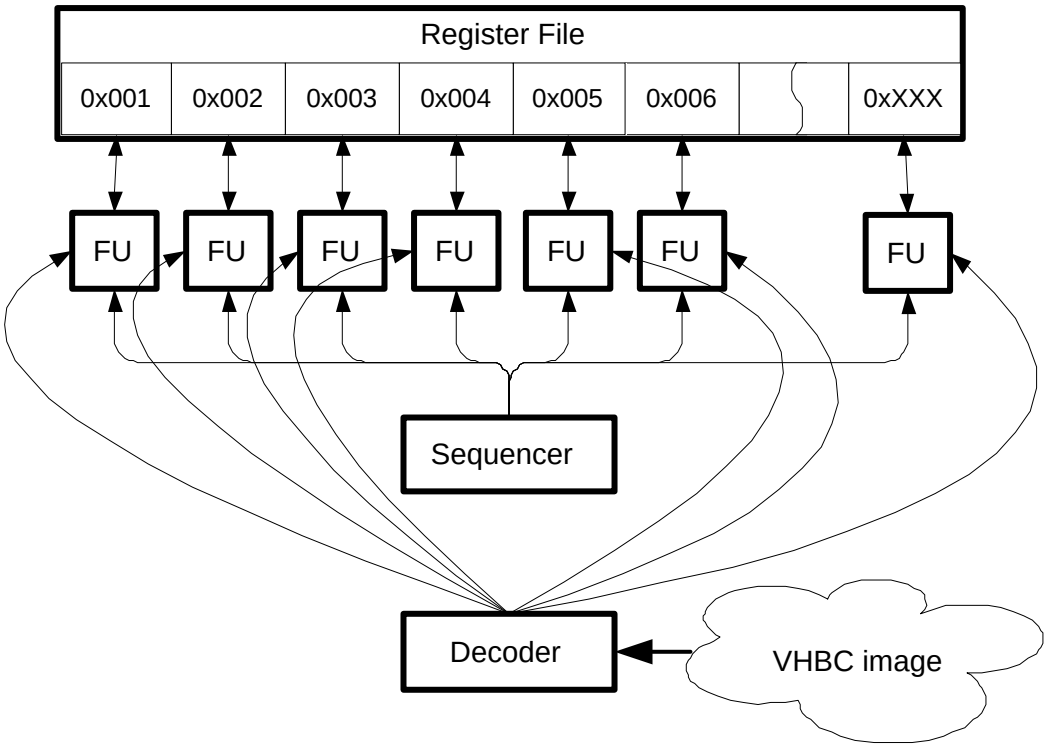

Abbildung 4.1: Aufbau der VHM

4.1.1 Decoder

Der Decoder teilt den Bytecode, welcher als Inputstream von Bits vorliegt, in die Instruktionsblöcke auf und verteilt diese Instruktionen an die entsprechenden FUs. Außerdem werden die Werte des Headers ausgelesen, um ggf. einige Anpassungen, wie z. B. die Registeradressierung, vornehmen zu können.

4.1.2 Functional Unit

Die einzelnen FUs führen die Instruktionen des Bytecodes aus. Dafür ist es nötig diese Instruktionen abzuspeichern. Aus diesem Grund besitzt jede FU einen eigenen Instruktionscache in dem die Instruktionen abgelegt werden. Ist der Instruktionscache der FU nicht groß genug, um alle Instruktionen zu speichern, können die entsprechenden fehlenden Operationen, zu gegebenen Zeitpunkt, nachgeladen und der Instruktionscache entsprechend angepasst werden. Weitere Bestandteile der FU sind der Ausführungskern, in dem die eigentliche Datenmanipulation stattfindet und ein Sequenzer welcher den Ablauf der Berechnung steuert. Diese Komponenten werden in der Abbildung 4.2 aus Sebastian Lange's Diplomarbeit[Lan02] grafisch dargestellt. Im Kern funktioniert die

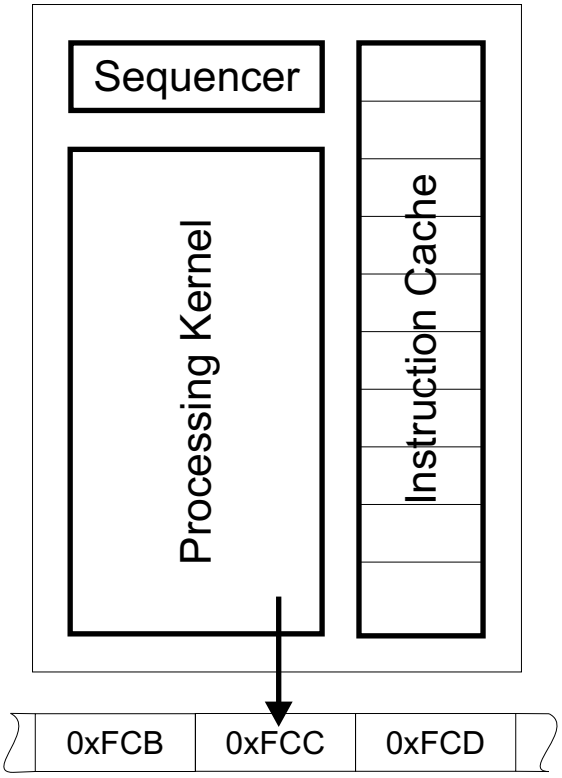

Abbildung 4.2: Aufbau der Functional Unit

Ausführung einer Operation in 4 Schritten. In den ersten beiden Schritten werden die

Adressen der beiden Operanden bestimmt und die entsprechenden Daten gelesen. Im dritten Schritt wird die eigentliche Instruktion ausgeführt und das Ergebnis wird dann im vierten Schritt zurück in das Register geschrieben. Dieses Zurückschreiben darf erst starten, wenn alle FUs die ersten 3 Schritte abgeschlossen haben, um die Richtigkeit der Daten zu gewährleisten.

4.1.3 Register

Das Register besteht aus mehreren einzelnen Zellen, welche zurzeit jeweils eine Breite von einem Bit besitzen. Ist ein Makrozyklus(eine komplette Verarbeitung des Bytecodes) abgeschlossen, werden die Ausgabewerte gesetzt und die neuen Eingabewerte übernommen. Während eines Mikrozyklusses(eine Instruktion wird verarbeitet) werden die Ergebnisse zwischengespeichert und können von den FUs bei der Abarbeitung der nächsten Instruktion gelesen werden. Die Werte der externen Verbindungen der VHM werden zu diesen Zeitpunkten allerdings nicht verändert.

4.1.4 Sequenzer

Durch den Sequenzer werden die einzelnen Komponenten synchronisiert und die Verarbeitung des Bytecodes gesteuert. Die Abarbeitung kann unterbrochen oder gestoppt werden, um einen neuen Bytecode zu verarbeiten oder einige Instruktionen nachzuladen.

4.1.5 Verbindungsnetzwerk

Den Lesezugriff auf alle Register für eine FU erlaubt das Verbindungsnetzwerk. Außerdem stellt es die Verbindung einer FU zu der entsprechenden Registerzelle zum Schreiben des Ergebnisses her. Zurzeit sind alle FU mit dem Registerzellen direkt verbunden. Dies erlaubt einen sehr schnellen, da gleichzeitigen, Lesezugriff, verbraucht aber viele Ressourcen und entsprechend viel Leistung.

4.2 Funktionsweise

Die Ausführung eines Bytecodes auf einer virtuellen Hardware-Maschine geschieht in 2 Schritten. Dem Einlesen und Dekodieren des Bytecodes und dem Abarbeiten der enthaltenen Instruktionen. Entsprechend der Größe des Datenbusses werden die Daten in Paketen dieser Länge eingelesen. In der ersten Phase werden die Informationen des Bytecodeheaders ausgelesen(1). Hier sind einige Statusinformationen über den Bytecode enthalten. Falls hier auch Konstanten angegeben sind, werden diese in das Register eingetragen(2). Danach werden die einzelnen Instruktionen ausgelesen(3) und ihre Registeradressierung an das Format der VHM angepasst. Jede Implementierung einer virtuellen Hardware-Maschine besitzt ein festes Format der Registeradressierung, um die Komplexität der FUs möglichst gering zu halten. Der Decoder passt die ggf. unterschiedlichen Adressierungsarten einander an. Die bearbeiteten Instruktionen werden in den Instruktionscache der FUs geschrieben(4). Wenn der komplette Bytecode abgearbeitet wurde, kann mit der Berechnung begonnen werden. Die Abbildung 4.3 aus der Diplomarbeit von Sebastian Lange[Lan02], in welcher der VHBC und die VHM entwickelt wurden, stellt diesen Ablauf noch einmal grafisch dar. Die in der Grafik benutzten Nummern entsprechen den in der Beschreibung verwendeten. Der erste Schritt einer jeden Berechnung

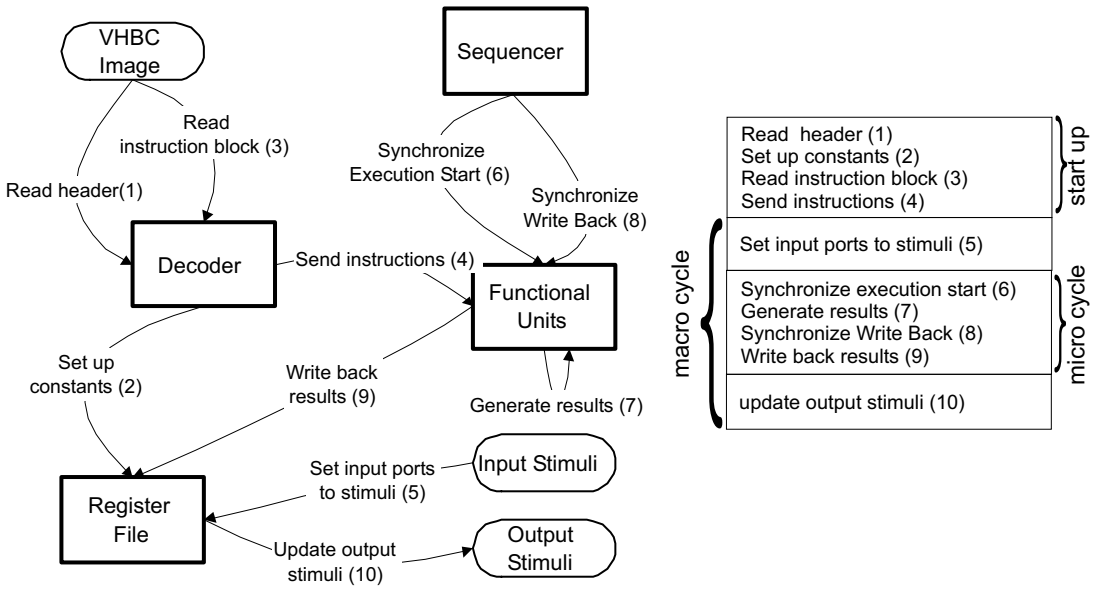

Abbildung 4.3: Funktionsweise der VHM

übernimmt die aktuellen Eingabewerte(5). Der Sequenzer steuert dann die einzelnen FUs. Sie werden synchronisiert und gestartet(6), führen ihre Instruktion aus(7) und schreiben dann die ermittelten Ergebnisse zurück(9) in die jeweilige Registerzelle. Dieser Schreibvorgang wird ebenfalls synchronisiert(8), damit keine Fehlinformationen erzeugt

werden können. Sind noch nicht alle Instruktionen des auszuführenden Bytecodes verarbeitet, wiederholen sich die Schritte 6 bis 9. Dieser Abschnitt der Berechnung wird als Mikrozyklus bezeichnet. Nachdem alle Instruktionen bearbeitet wurden, werden die Ausgabewerte gesetzt(10), die neuen Eingabewerte übernommen und die Instruktionen erneut komplett verarbeitet. Den Zyklus von 5 bis 10 nennt man Makrozyklus. Dieser entspricht der Zeit, die die VHM benötigt, um die von außen angelegten Daten zu verarbeiten und zurückzugeben.

5 Aufgaben und Funktionsweise der Komponenten

Nachdem in den vorherigen Kapiteln die Vorraussetzungen erläutert, die VHM allgemein beschrieben, die notwendigen Werkzeuge erklärt und die Gründe für diese Arbeit benannt wurden, wird in den folgenden Kapiteln die Hardwarerealisierung der VHM beschrieben. In diesem Kapitel werden die einzelnen erstellten Komponenten, sowie ihre Aufgaben und Funktionsweisen beschrieben. Das Zusammenspiel der Komponentenrealisierungen wird im nachfolgenden Kapitel erläutert.

Die Hardwarerealisierung der virtuellen Hardware-Maschine besteht aus 7 Komponenten und einem VHDL-Package, welches ihre Eckdaten bestimmt. Dadurch lassen sich verschiedene Implementierungen der VHM erzeugen, in denen sich zum Beispiel die Anzahl der benutzten Berechnungseinheiten(FU[1]s) unterscheiden. Es wird neben den Aufgaben der Komponenten auch beschrieben, wie diese abgearbeitet werden. Einige Komponenten hätte man zusammenfassen können, aber dies hätte das entstehende Design nicht verändert. Die Aufteilung in diese verschiedenen Komponenten erleichtert das Verstehen und Warten des VHDL-Codes. Als einzige Komponente ist die FU mehr als einmal in der VHM enthalten. Bevor jedoch die, in alphabetischer Reihenfolge aufgelisteten, Komponenten beschrieben werden, werden die Änderungen des in Abschnitt 3.1 beschriebenen Bytecodes benannt.

5.1 Notwendige Änderungen am Bytecode

Damit die Implementierung einer virtuellen Hardware-Maschine den Bytecode verarbeiten kann, musste ein Detail am Bytecode geändert werden. Ein Abarbeiten der bisherigen Struktur des Bytecodes hätte sich nur in wesentlich komplizierterer Weise in Hardware realisieren lassen.

[1] functional unit

5.1.1 Ablageformat der Operanden

Um den Bytecode besser verarbeiten zu können, musste die Art, in der die Adressen der Operanden abgelegt werden, verändert werden. Die Adressen werden jetzt mit dem niederwertigsten Bit(LSB - least significant Bit) zuerst abgelegt. Dies ist nötig, um die variable Adresslänge an die festgelegte Adressbreite der VHM anzupassen. Ein Auslesen der Bits in der bisherigen Reihenfolge ist nicht möglich, da zur Designzeit die Anzahl der zu lesenden Bits bekannt sein muss. Das genaue Verfahren ist in den Abschnitten 5.2.4 und 5.4.2 näher beschrieben.

5.1.2 Ablageformat des Operationscodes

Damit die Richtung der Operanden und des Operationscodes nicht unterschiedlich ist, wurde dessen Ablageformat ebenfalls verändert. Die zurzeit verwendeten Operationscodes sind also nicht mit der Übersicht in 3.1.2 und der Beschreibung von Sebastian Lange[Lan02] identisch. Die aktuell verwendeten Operationscodes sind in der Tabelle 5.1 dargestellt.

OpCode	Instruktion	OpCode	Instruktion	OpCode	Instruktion
0000_2	EOB	1000_2	NOP	0100_2	MOV
1100_2	NOT	0010_2	AND	1010_2	NAND
0110_2	OR	1110_2	NOR	0001_2	XOR
1001_2	EQ	0101_2	IMP	1101_2	NIMP
0011_2	CMOV				

Tabelle 5.1: Aktuelle Operationscodes des Hardware-Bytecodes

5.2 Decoder

Die komplexeste Komponente der VHM ist der Decoder. Er dekodiert den angelegten Bytecode. Dies beinhaltet das Auslesen des Headers und das Verteilen der Instruktionen an die verschiedenen FUs. Die Werte des Headers werden genutzt, um die Ein- und Ausgabewerte zu bestimmen, einige Statuswerte für die Verarbeitung anzulegen und die Kompatibilität zwischen der VHM und dem Bytecode zu überprüfen. Stehen nicht genug Ressourcen für die Abarbeitung des Bytecodes zur Verfügung, wird die Verarbeitung abgebrochen und ein entsprechender Fehlercode(A.4) ausgegeben. Ist der Header verarbeitet, werden die Instruktionen ausgelesen und an die FUs weitergeleitet(↗6.1). Nachdem ein Wert ausgelesen wurde, muss der Datenstrom nach links geschoben werden, damit die aktuellen Daten immer am Anfang des Datenstroms anliegen. Dies geschieht

mit Hilfe des Schieberegisters(5.7) und wird im Abschnitt 6.4 näher beschrieben. Wenn die Verarbeitung des Bytecodes abgeschlossen ist, wird die Berechnung der neuen Schaltung gestartet(↗6.2). Die einzelnen Teilaufgaben werden in den folgenden Abschnitten genauer beschrieben.

Abbildung 5.1 zeigt den Zusammenhang der einzelnen Prozesse des Decoders. Der

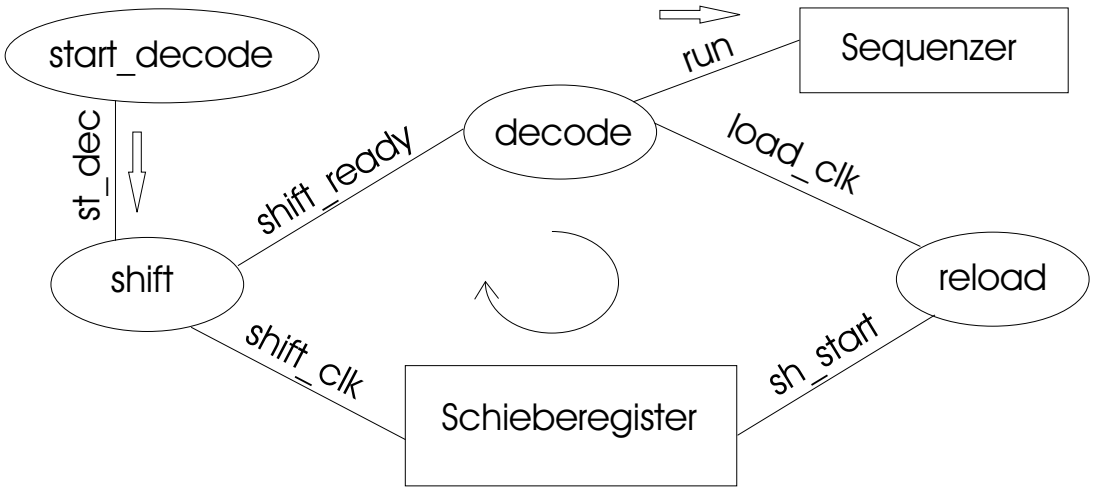

Abbildung 5.1: Die Prozesse des Decoders

Prozess start_decode wird ausgeführt, wenn das Signal *update_in* der VHM auf '1' geht. Dieser Fall tritt ein, wenn ein neu zu verarbeitender Bytecode an die VHM gesendet wird. Durch setzen des Signals *st_dec* auf '1' wird der Prozess shift gestartet. Dieser setzt wiederum *shift_ready* auf '1' und beginnt somit das Dekodieren, da durch das Verändern dieses Signals der Prozess decode aktiviert wird. Ist ein Teil des Datenstroms verarbeitet, wird das Signal *load_clk* verändert. Dadurch wird der Prozess reload aktiviert, welcher überprüft, ob neue Daten nachgeladen werden müssen, und dies ggf. erledigt. Eine Änderung des Signals *sh_start* veranlasst das Schieberegister zum Schieben des Datenstroms. Ist das Verschieben beendet, wird dies durch *shift_clk* signalisiert und im Prozess shift werden die Datenströme für das weitere Dekodieren vorbereitet. Jetzt ist ein Umlauf dieses Verarbeitungskreislaufes abgeschlossen. Dieser Vorgang wiederholt sich solange, bis der komplette Bytecode verarbeitet ist. Dann wird das Signal *load_clk* nicht mehr verändert und der Kreislauf ist beendet. Das Signal *run* wird auf den Wert '1' gesetzt und der Sequenzer startet die Abarbeitung der dekodierten Anwendung. Die Funktionsweise der hier genannten Prozesse wird in den folgenden Abschnitten erläutert.

5.2.1 Prozess decode

In diesem Prozess wird der Bytecode verarbeitet. Es werden die Daten aus dem Datenstrom(Signal *data_load*) den entsprechenden Signalen zugeordnet. Um die aktuellen Daten zuordnen zu können, werden einige Statusvariablen benutzt. Die Variable *fetch* ist ein Zahlenwert zwischen 0 und 5, welcher den aktuellen Bereich im Bytecode bestimmt. Welches Element des Headers gerade bearbeitet wird, bestimmt die Variable *header_pos*. Sie ist ebenfalls ein Zahlenwert, diesmal im Bereich von 0 bis 7. Die Bedeutung der einzelnen Werte der beschriebenen Variablen wird später in dieser Arbeit erläutert. Um beim Auslesen der Instruktionen festzustellen welcher Teil der Operation gerade bearbeitet wird, benutzt man die Variable *operand_pos*. Sie besitzt den Wert 0, wenn als nächstes der Operationscode ausgelesen wird, den Wert 1 für den 1. Operanden und den Wert 2 für den 2. Operanden. Einer FU wird immer dieselbe Instruktion innerhalb eines Instruktionsblockes zugewiesen. Diese Position wird in der Variable *instr_pos* gespeichert. Der Wert 0 steht für die 1. FU(FU Nr. 0) und der Wert FUZ-1(↗A.5) für die letzte FU(FU Nr. FUZ-1). Mit diesen Zustandsvariablen kann der Prozess jederzeit bestimmen, an welcher Stelle im Bytecode er sich gerade befindet und kann so die Daten den entsprechenden Signalen zuordnen.

Im Prinzip besteht der Prozess decode aus verzweigten case-Anweisungen(↗3.2), welche die oben genannten Statusvariablen auswerten. Als erstes wird das Signal *fetch* ausgewertet. Es beschreibt grob, an welcher Stelle sich der Dekodierungsprozess befindet. Eine genaue Auflistung der einzelnen Werte befindet sich in Tabelle 5.2.

Wert	Beschreibung
0	Beginn, die ersten Daten werden geladen
1	Die nächsten Daten werden geladen um den Puffer zu füllen
2	Die Werte des Header werden ausgewertet
3	Die Konstanten werden übernommen
4	Die Instruktionen des Bytecodes werden verarbeitet
5	Das Dekodieren ist beendet, die Berechnung wird gestartet

Tabelle 5.2: Werte der Statusvariable fetch

Am Anfang des Dekodierens werden nur Daten geladen, *fetch* besitzt die Werte 0 bzw. 1. Nachdem der Datenstrom die doppelte Länge des Datenbusses(↗A.3) besitzt, beginnt die Verarbeitung des Headers. Die Reihenfolge der Werte des Headers ist festgelegt und entspricht der im Kapitel 3 in Tabelle 3.1 beschriebenen. Die Felder magic und Version werden bisher noch nicht ausgewertet, da versucht wurde eine möglichst kompakte Version der virtuellen Hardware-Maschine zu entwickeln, und somit zuerst nur die Teile

implementiert wurden, welche für ein Funktionieren der VHM notwendig sind.

Wird der Header abgearbeitet, wertet eine 2. case-Anweisung die Variable *header_pos* aus. Die genaue Bedeutung der Werte stellt die Tabelle 5.3 dar.

Wert	Beschreibung
0	Liest die ersten Bytes des Headers 'VHBC'
1	Bestimmt die Version des Bytecodes
2	Die Adressbreite der Registeradressierung wird ausgelesen
3	Die Anzahl der benutzten Registerzellen wird übernommen
4	Die Anzahl der Eingabewerte wird gespeichert
5	Die Anzahl der Ausgabewerte wird ausgewertet
6	Die Startposition der Konstanten wird ermittelt
7	Die Anzahl der Konstanten wird berechnet

Tabelle 5.3: Werte der Zustandsvariable header_pos

Bei jedem Auslesen eines der Felder des Bytecode-Headers wird die Kompatibilität geprüft. Das bedeutet, es werden die ausgelesenen Werte mit denen im Package constant(5.9) festgelegten Größen der Implementierung verglichen. Ist ein Wert des Bytecodes größer als der Vergleichswert, wird das Signal *error* mit dem entsprechenden Fehlercode(↗A.4) gesetzt und *fetch* der Wert 5 zugewiesen. Die Abarbeitung wird also gestoppt. Wenn die Implementierung einer VHM den entsprechenden Bytecode ausführen kann, wird die Anzahl der Konstanten bestimmt. Es gibt nun 2 mögliche Fälle, es existieren Konstanten in diesem Bytecode oder es wurden keine Konstanten angegeben. Im 2. Fall erhält die Variable *fetch* den Wert 4 und das Verarbeiten der Instruktionen beginnt. Sind allerdings Konstanten angegeben, wird deren Anzahl - Signal *const_count* - bestimmt und *fetch* mit dem Wert 3 belegt.

Die Werte der Konstanten befinden sich am Ende des Headers. Sie werden, falls welche benutzt werden, als letzte Werte des Headers ausgewertet. Zurzeit bestehen die Konstanten noch aus einem Bit, aber in der Definition des Bytecodes sind schon 8 Bit für jede Konstante vorgesehen. Beim Auslesen wird also nur das letzte Bit dieser 8 betrachtet und im nächsten Aufruf des Prozesses wird die nächste Konstante bestimmt. Dies geschieht dann genau *const_count*-mal und der Variable *fetch* wird der Wert 4 zugewiesen.

Nachdem alle Werte des Headers verarbeitet wurden, werden die Instruktionen ausgelesen und an die FUs verteilt. Die beiden benötigten Zustandsvariablen *operand_pos* und *instr_pos* wurden weiter oben schon beschrieben. Eine weitere case-Anweisung wertet die Variable *operand_pos* aus. Es entstehen nun die 3 folgenden Möglichkeiten:

1. Der Operationscode wird ausgewertet.

2. Der 1. Operand wird bestimmt.

3. Der 2. Operand wird bestimmt.

Im ersten Fall wird der Operationscode, dessen Länge zurzeit auf 4 Bit festgelegt ist, aus dem Datenstrom gelesen. Es wird nun überprüft, ob dieser Wert dem End-of-Block(EOB)-Statement entspricht. Ist dies der Fall, ist das Ende eines Instruktionsblockes erreicht. Gekennzeichnet wird dies, indem das Signal *last_read_eob* auf '1' gesetzt wird. Dieses Verfahren wird benutzt, um zu erkennen, wann 2 EOB-Statements nacheinander gelesen wurden, was das Ende des Bytecodes bedeuten würde. Als nächstes wird überprüft, ob *last_read_eob* '1' ist. Um hier mögliche Verwirrungen auszuschließen, gehe ich noch einmal auf das Zuweisungsverfahren von Signalen in VDHL ein. Da *last_read_eob* ein Signal ist, wird die eben beschriebene Zuweisung des Wertes '1' erst nach Abarbeitung aller Prozesse dieses Deltazyklusses ausgeführt. Folgende Codezeilen ergeben also nicht eine automatische Auswertung mit wahr in der if-Anweisung.

```
--den Opcode auswaehlen
case operand_pos is
when 0 =>
  opcode_test:=data_load(0 to opcode_length-1);
  if opcode_test="0000"
  then
    last_read_eob<='1'; --als letztes EOB gelesen
    if last_read_eob='1'
    then
      fetch<=5; --2mal EOB gelesen -> ENDE des Bytecodes
    else
```

Wird die if-Anweisung mit wahr ausgewertet, bedeutet dies, dass die Verarbeitung des Bytecodes abgeschlossen ist. Wurde aber davor eine Instruktion ausgewertet, ist dieser Instruktionsblock hiermit abgeschlossen. Immer wenn ein EOB gelesen wurde, brauchen keine Operanden mehr ausgewertet zu werden. Damit alle FUs dieselbe Anzahl von Operation in ihrem jeweiligen Instruktionscache besitzen, wird an die restlichen FUs eine MOV-Operation weitergeleitet. Die Aufgabe übernimmt der FU-Multiplexer(5.4). Entspricht der ausgewertete OpCode nicht dem EOB-Statement, werden die beiden Operanden ausgelesen. Eine Besonderheit des Auslesens der Operanden und ein passendes Beispiel sind im Abschnitt 5.2.4 beschrieben. Die erstellte Instruktion und der aktuelle Wert von *instr_pos* werden an den FU-Multiplexer weitergeleitet. Dieser weist die Instruktion der jeweiligen FU zu, wenn dem Signal *clk_instr* der Wert '1' zugewiesen wurde.

Nach jeder Auswertung eines Teils des Bytecodes wird dem Signal *sh_sel* die Anzahl der gelesenen Bits als 6 Bit-Zahl zugewiesen. Das Schieberegister benutzt dieses Signal als Verschiebungsbreite. Ist ein Wert dem entsprechenden Signal zugeordnet worden, wird durch Ändern des Signals *load_clk* der in Abbildung 5.1 beschriebene Kreislauf fortgesetzt. Dadurch kann Stück für Stück der angelegte Bytecode dekodiert werden.

5.2.2 Prozess reload

Im Prozess wird überprüft, ob noch genug Daten im Datenstrom vorhanden sind. Um diese Aufgabe zu bewältigen, wird in der Variable *pos* die aktuelle Länge des Bytecodes abgelegt. Dafür wird die Anzahl - sie ist im Signal *sh_sel* abgelegt - der im Prozess decode(5.2.1) ausgewerteten Bits bestimmt. Es entstehen 2 mögliche Abarbeitungsmethoden.

Die Erste tritt ein, wenn *sh_sel* den Wert $000000_2 = 0$ besitzt. Dies ist nur am Anfang der Fall und bedeutet, dass die Daten geladen werden sollen. Wie weiter oben beschrieben, wird der Datenstrom erst mit Daten der doppelten Datenbusbreite gefüllt. Der Fall *sh_sel*=0 tritt also 2-mal ein. Danach besitzt *pos* den Wert 2xDB(↗A.3).

Werden die Werte des Bytecodes dekodiert, besitzt *sh_sel* einen Wert der größer als 0 ist. Der Prozess reload bestimmt nun, ob neue Daten nachgeladen werden müssen(*da_mir*='1') oder nur der Datenstrom mit Hilfe des Schieberegisters geschoben werden muss(*da_mir*='0'). In beiden Fällen wird der Wert des Signals *sh_start* verändert und somit dem Schieberegister signalisiert, dass der Vorgang gestartet werden soll. Eine genauere Beschreibung dieser Überprüfung findet man im Abschnitt 5.2.5. Dort ist auch ein Beispiel zum besseren Verständnis angegeben.

5.2.3 Prozess shift

Nachdem die Datenströme entsprechend geschoben wurden, müssen die aktuellen und die nachgeladenen Daten miteinander verbunden werden. Der Prozess shift wertet hierfür das Signal *da_mir*, welches im Prozess reload(↗5.2.2) entsprechend gesetzt wurde, aus.

Besitzt es den Wert '1' wurden Daten nachgeladen und müssen mit dem aktuellen Datenstrom verknüpft werden. Im Schieberegister wurden die Datenströme so geschoben, dass keine Überschneidungen auftreten. Also können die Datenströme - die Signale *bsh_out* und *bsh_mir_out* - mit einer einfachen Oder-Verknüpfung verbunden werden. So entsteht ein neuer Datenstrom(Signal *data_load* - wird vom Prozess decode ausgewertet), welcher den aktuellen Bereich des Bytecodes enthält.

Ist der Wert des Signals *da_mir* gleich '0', wurde nur der Datenstrom, um die entsprechende Anzahl Bits, nach links geschoben. Da keine Daten nachgeladen wurden,

kann einfach das komplette Ausgangssignal(*bsh_out*) des Schieberegisters in das Signal *data_load* übernommen werden.

Nach dem *data_load* gesetzt wurde, führt eine Änderung des Signals *shift_ready* zur Aktivierung des Prozesses decode(5.2.1). Die Abarbeitung des nächsten Teils des Bytecodes wird wie beschrieben durchgeführt.

5.2.4 Auslesen der Operanden

Um die variable Länge der Operanden an die Registeradressbreite[2](↗A.10) anzupassen, müssen die Operanden speziell ausgelesen werden. Die Funktionsweise soll anhand des folgenden Beispiels näher erläutert werden. In diesem Beispiel beträgt die RAB 5 Bit und die Länge eines Operanden 3 Bit. Dieser Wert ist im Header(↗3.1.1) des Bytecodes im Feld address_width abgelegt und wird im weiteren als bpa[3] bezeichnet. Aus diesen Zahlen folgt, dass der ausgelesene Wert von 3 auf 5 Bit aufgepumpt werden muss, damit er in der entsprechenden Instruktion abgelegt werden kann. Der Decoder legt dafür eine 5 Bit Maske an, die mit bpa vielen Einsen beginnt und mit $rab - bpa$ vielen Nullen endet. Hier würde sie also '11100' lauten. Der zugehörige VHDL-Code kann unter D.2 betrachtet werden. Nun werden 5 Bit aus dem Datenstrom gelesen und mit der Maske mit Hilfe der UND-Verknüpfung verbunden. D. h. die Bits des Operanden bleiben erhalten und die zuviel gelesenen Bits werden zu '0'. Sind 2 Operanden so verarbeitet, wird die entsprechende Instruktion an den FU-Multiplexer(5.4) weitergeleitet.

5.2.5 Nachladen und Schieben der Daten

Da nicht der komplette Bytecode im Decoder vorhanden sein kann, muss dieser nach und nach verarbeitet werden. Um dies zu erreichen, wird ein Datenstrom(*data_load*) der doppelten Datenbusbreite(DB), in diesem Fall 64 Bit, bereitgehalten. Wenn ein Teil des Datenstroms ausgelesen und bearbeitet wurde, werden die Daten mit Hilfe des Schieberegisters(5.7) nach links geschoben. Dazu bestimmt der Decoder den Wert der Verschiebung(*sh_sel*), welcher maximal dem Wert DB(↗A.3) entsprechen kann. Außerdem wird die Anzahl der vorhandenen Bits(*pos*) des Bytecodes gespeichert. Ist die Differenz(DIF) der vorhandenen Bits und der Verschiebung größer als DB, also 32, werden die Daten dem Schieberegister übergeben und geschoben. Wenn dies nicht der Fall ist, werden zuerst noch die nächsten 32 Bit des Bytecodes eingelesen und dem externen Speicher wird signalisiert, dass die nächsten Daten bereitgestellt werden sollen. Nun müssen der Datenstrom(*bsh_in*) und die nachgeladenen Daten(*bsh_mir_in*) geschoben werden, um sie

[2] RAB
[3] bits per address

lückenlos verknüpfen zu können. Die vorhandenen Daten werden, wie sonst auch, um die Anzahl der zuletzt ausgelesenen Bits geschoben. Damit die neuen Daten daran angehangen werden können, müssen sie um DB-DIF Bits verschoben werden. Nachdem die Daten wie in 6.4 erläutert geschoben wurden, kann mit dem Dekodieren des Bytecodes fortgefahren werden.

Anhand eines Beispiels wird diese Vorgehensweise noch einmal verdeutlicht. Es wird ein Bytecode mit 5 bpa(↗3.1.1) verwendet. Daten werden im Folgenden mit 'X' bezeichnet und '0' bedeutet, dass keine Daten anliegen. Im Datenstrom befinden sich 40 Bit und als nächstes wird eine Instruktion ausgelesen. Aus dem Datenstrom $(X)_{40}(0)_{24}$ werden also als erstes 4 Bit für den OpCode ausgelesen. Daraus ergibt sich ein Wert für DIF von $40 - 4 = 36$. Es folgt daraus, dass in diesem Fall nur der Datenstrom um 4 Stellen nach links geschoben werden muss. Dieser sieht dann wie folgt aus: $(X)_{36}(0)_{28}$. Der Operand besteht aus 5 Bit, d. h. der neue Wert für DIF beträgt $36 - 5 = 31$. Jetzt müssen also neue Daten nachgeladen werden. Dafür wird ein 32 Bit Register benutzt$[(X)_{32}]$. Nun wird der Wert der Verschiebung für diese Daten bestimmt: $32 - 31 = 1$. Dem Schieberegister werden diese Signale übergeben und als Ergebnis liefert es jetzt 2 Signale. Den Datenstrom $(X)_{31}(0)_{33}$ und die neuen Daten $(0)_{31}(X)_{32}(0)_1$. Diese werden nun im Decoder mittels einer ODER-Operation verbunden. Es entsteht als neuer Datenstrom $(X)_{63}(0)_1$. Mit diesem Datenstrom kann nun der Bytecode weiter dekodiert werden. Die Beschreibung des Schieberegisters als VHDL-Code ist im Abschnitt D.4 abgelegt.

5.3 Functional Unit(FU)

Die FUs dienen der Berechnung der, im übergebenen Bytecode enthaltenen, Schaltung. Jede FU besitzt einen eigenen Instruktionscache, in dem die angegebenen Operationen gespeichert werden. Wird dieser gefüllt, erhöht sich die Instruktionenanzahl(A.6) um eins. Jede Instruktion besteht aus dem Operationscode und 2 Registeradressen. Die Länge des Operationscodes ist zurzeit immer 4 Bit und die Länge der Registeradressen ist gleich dem Wert der Registeradressbreite(A.10). Beim Berechnen der Schaltung lädt eine FU die, in der Instruktion angegebenen, Operanden aus den entsprechenden Registerzellen und setzt dann das Ergebnis in die zugeordnete Registerzelle. Diese Zuordnung entspricht der Nummerierung der einzelnen FUs. Das bedeutet, FU Nr. i setzt das Ergebnis in Registerzelle i, wobei i einem Zahlenwert aus dem Bereich von 0 bis FUZ-1(↗A.5) entspricht. Diese beiden Teilaufgaben werden in den folgenden Abschnitten beschrieben.

5.3.1 Füllen des Instruktionscaches

Das Füllen des Instruktionscaches ist ein sehr einfacher Prozess. Wird durch den FU-Multiplexer(5.4) das Signal *instr_clk* auf den Wert '1' gesetzt, übernimmt die FU den Wert des Signals *instr* in den Instruktionscache. Dieser ist ein Array von Instruktionen mit der im Package constant(5.9) festgelegten Größe. Außerdem wird der Wert der Instruktionenzahl(fc) um eins erhöht. Beim Dekodieren eines neuen Bytecodes setzt der Decoder *clear_cache* auf '1'. Dadurch wird das Signal fc wieder mit dem Wert 0 initialisiert. Die Signale im Instruktionscache werden nicht gelöscht, da die bei der Berechnung benötigten Instruktion, während des Dekodieren ja neu gesetzt werden. Andere, jetzt eventuell nicht benutzte, Bereiche des Instruktionscaches, können bei der Berechnung nicht erreicht werden. Dadurch ist ein Löschen dieser Instruktionen nicht notwendig.

5.3.2 Berechnen der Operationen

Ist der Bytecode dekodiert und der Instruktionscache entsprechend gefüllt, kann die Berechnung der Schaltung starten. Der Sequenzer(5.8) hatte dem Signal *reset* während der Dekodierung den Wert '1' zugewiesen. Dadurch ist in einer FU der Befehlszähler(pc) auf den Wert 0 gesetzt. Startet die Berechnung, besitzt *reset* den Wert '0' und die FU wertet das Signal *clk* aus. Geht es vom Wert '0' auf '1' über, wird die aktuelle Operation ausgeführt. Dafür liest die FU die Instruktion an der Stelle pc aus dem Instruktionscache. Eine case-Anweisung wertet den Operationscode der Instruktion aus und führt die entsprechende Operation aus. Dafür werden die Operanden aus den entsprechenden Registerzellen geladen. Das Ergebnis der Operation wird dem Signal *reg_out* zugewiesen. Bei fallender Flanke - Übergang von '1' auf '0' - des Signals *clk* wird dieser Wert im Register übernommen. Nach der Berechnung wird noch überprüft, ob die eben ausgeführte Operation die Letzte für die Berechnung benötigte war. Dies ist der Fall wenn, der Wert von pc dem Wert von $fc-1$ entspricht. Nun muss der Wert des Befehlszählers wieder auf 0 gesetzt werden, da die Berechnung abgeschlossen ist. Außerdem wird das Signal *restart* auf '1' gesetzt. Es signalisiert dem Register, dass die Berechnung abgeschlossen ist - Ende eines Schaltungstaktes(↗A.11). Ist die Berechnung noch nicht abgeschlossen, wird der Wert von pc um eins erhöht und *restart* der Wert '0' zugewiesen. Nur die Fu-Nr. 0 besitzt dieses Signal und ist mit dem Register verbunden. Es ist nur eine Verbindung nötig, da die verschiedenen FUs durch den Takt synchronisiert betrieben werden. Dadurch existieren auch 2 verschiedene VHDL-Beschreibungen für die FU-Nr. 0 und die restlichen FUs, die sich aber nur in der Rückgabe des Signals *restart* unterscheiden. Bei den restlichen FUs ist dieses Ausgabesignal nicht vorhanden.

5.4 FU-Multiplexer

Dem Decoder(5.2) und den einzelnen FUs(5.3) ist der FU-Multiplexer zwischengeschaltet. Er verteilt die dekodierte Instruktion an die entsprechende FU. Ist ein Instruktionsblock abgeschlossen, werden die restlichen FUs mit einer MOV-Operation gefüllt. Dadurch enthalten immer alle FUs die gleiche Anzahl von Instruktionen. Eine weitere Aufgabe ist das Umdrehen der Operanden einer Instruktion. Dies ist nötig um den Zugriff, auf die im Bytecode angegebenen Register, sicherzustellen. Die Anordnung des Operationscodes wird nicht geändert.

5.4.1 Verteilen der Instruktionen

Um die Instruktionen an die entsprechenden FUs zu verteilen, benutzt der FU-Multiplexer die Signale *instr_out* und *sel_out*. Das Signal *sel_out* stellt eine Bit-Maske dar, in der eine '1' bedeutet, dass diese Instruktion übernommen wird und eine '0' bedeutet, dass die angelegte Instruktion nicht gespeichert wird. Mit jedem Bit von *sel_out* ist eine FU(5.3) verbunden. Somit ist festgelegt, ob die entsprechende FU die aktuelle Instruktion in ihren Instruktionscache übernimmt oder nicht. Die Instruktion steht im Signal *instr_out*, welches ein Array von Instruktionen darstellt. Jedes Arrayelement ist wieder mit einer FU verbunden. Hat der Decoder(5.2) eine Instruktion dekodiert, signalisiert er dies dem FU-Multiplexer und übergibt die entsprechende Instruktion(*instr_in*) und die Ziel-FU(*fu_sel*). Die Operanden von *instr_in* werden wie in 5.4.2 beschrieben bearbeitet und die Instruktion wird im Signal *instr_out* an der Stelle *fu_sel* gesetzt. Ebenso wird *sel_out(fu_sel)* der Wert '1' zugewiesen und den restlichen Stellen der Wert '0'. Dadurch reagiert nur die angesprochene FU auf die aktuelle Instruktion. Etwas anders verhält sich der FU-Multiplexer, wenn der Dekoder das Ende eines Instruktionsblockes signalisiert. In diesem Fall besitzt das Signal *fu_nop* den Wert '1'. Die Werte von *sel_out* werden, ab der Stelle die dem Wert von *fu_sel* entspricht, mit '1' belegt. Ähnlich ist das Verhalten bei der Zuweisung des Signals *instr_out*, hier werden die entsprechenden Elemente des Arrays mit einer MOV-Operation belegt. Die MOV-Operation wird benutzt, da beim Benutzen einer NOP-Operation unter Umständen das Ergebnis der FU noch in einem undefinierten Zustand sein kann. Damit dies keine Folgefehler in der Berechnung hervorruft, wurde dieser Mechanismus gewählt. Durch das mehrfache Vorkommen des Wertes '1' in *sel_out* reagieren nun alle FUs, die in diesem Instruktionsblock noch keine Instruktion erhalten, auf die MOV-Operation. Es ist sichergestellt, dass alle FUs nach dem Dekodieren und Verteilen der Instruktionen dieselbe Anzahl von Operationen in ihrem Instruktionscache besitzen.

5.4.2 Umdrehen der Operanden

Die im Decoder verarbeiteten Instruktionen, werden im FU-Multiplexer noch einmal angepasst. Der Operationscode muss nicht verändert werden, aber die Operanden müssen noch umgedreht werden. Das bedeutet, sie werden vom least significant bit(LSB) zum most significant bit(MSB) abgelegt.

Auch hierzu ein Beispiel. Es werden dieselben Werte wie in Abschnitt 5.2.4 verwendet. Das Umdrehen der Operanden ist notwendig, da sonst keine eindeutige Zuordnung der Operanden zu den Registerzellen in verschiedenen Implementierungen einer VHM möglich wäre. Im Bytecode ist als Operand Registerzelle 3 angegeben. Bei einer 3 Bit Adressierung lautet der Operand also '011'. Die Decoder zweier VHM-Implementierungen mit einer RAB[4](↗A.10) von 4 bzw. 5 würden das Zielregister in den Instruktionen mit '0110' bzw. '01100' ablegen. Die benötigte Registerzelle würde also von den FUs einmal als $0110_2 = 6$ bzw. $01100_2 = 12$ interpretiert. Derselbe Bytecode liefert also auf verschiedenen Implementierungen der virtuellen Hardware-Maschine unterschiedliche Ergebnisse. Dies ist nicht das erwünschte Verhalten der VHM. Um dies zu verhindern, werden die Operanden schon umgedreht im Bytecode(↗5.1) abgelegt. In diesem Fall lautet der Operand also '110' nicht '011'. Daraus ergeben sich nach dem Dekodieren die Registerzellen '1100' und '11000'. Diese werden nun im FU-Multiplexer nochmals umgedreht. Es ergeben sich also $0011_2 = 3$ und $00011_2 = 3$ bei 2 verschiedenen Implementierungen der VHM. Durch diesen Mechanismus ist der Bytecode auf verschiedenen VHMs lauffähig. Einzige Bedingung ist, dass der Wert der Registeradressierung(bpa) nicht größer als die RAB der jeweiligen virtuellen Hardware-Maschine ist. Damit dies nicht geschehen kann, vergleicht der Decoder beim Auslesen des Bytecodeheaders diese Werte und bricht die Verarbeitung ggf. ab.

Tabelle 5.4 stellt die Abläufe während des Verarbeitens mit und ohne Umdrehen der Operanden dar(↗VHDL-Code unter D.3).

	alter BC	4Bit VHM	5Bit VHM	neuer BC	4Bit VHM	5Bit VHM
Operand	011			110		
Decoder		0110	01100		1100	11000
FU-MUX		0110	01100		0011	00011
Ergebnis		6	12		3	3

Tabelle 5.4: Verarbeitung der Operanden

[4]Registeradressbreite

5.5 Konstanten-Multiplexer

Die Eingaben an der VHM werden im Konstanten-Multiplexer mit den im Bytecode enthaltenen Konstanten verknüpft. Die benötigten Werte erhält dieser vom Decoder(5.2). Diese sind die Position ab der die Konstanten stehen(*const_pos*), die Konstanten des Bytecodes(*const*) und die Eingaben der VHM(*input*). Werden keine Konstanten für die Verarbeitung der Schaltung benötigt, wird im Decoder das Signal *no_const* auf '1' gesetzt. Der Prozess multiplex - siehe VHDL-Beschreibung im Abschnitt D.5 auf Seite 77 - setzt das Ausgabesignal *output* mit den Werten von input, wenn *no_const* den Wert '1' besitzt. Ansonsten werden die Daten nach folgendem Prinzip bearbeitet. In einer Schleife werden alle Bits des Ausgangssignals durchlaufen. Ist die Position des aktuellen Bits - nachfolgend mit AB bezeichnet - niedriger als der Wert von const_pos(CP), wird das entsprechende Bit des Signals *input* übernommen. Ab der Position, die dem Wert des Signals *const_pos* entspricht, wird das Bit der Position AB-CP des *const*-Signals an das Register weitergeleitet. Durch dieses Verfahren sind zu jedem Zeitpunkt die aktuell angelegten Eingabewerte mit den im Bytecode vorhandenen Konstanten verbunden und das Register kann den FUs die gewünschten Operanden bereitstellen.

5.6 Register

Das Register hält die aktuellen Zwischenergebnisse vor. Am Anfang eines Schaltungstaktes(A.11) beinhaltet es die vom Konstanten-Multiplexer(5.5) bearbeiteten Eingaben und am Ende die Ergebnisse der ausgeführten Schaltung. Um Rechenfehler zu vermeiden, werden dem Register die Anzahl der benutzten Eingabewerte(*in_offset*) - entspricht Summe der Anzahl der benutzten Eingaberegister und der Anzahl der Konstanten des aktuellen Bytecodes - und die Anzahl der benutzten Ausgaberegister(*out_offset*) vom Decoder mitgeteilt. Das Signal *restart* teilt dem Register das Ende eines Schaltungstaktes mit. Hat es den Wert '1' werden an die ersten *out_offset*-vielen Bits des Registerinhaltes an das Signal *ext_output* übergeben und die restlichen Bits mit '0' initialisiert. Dies geschieht da die VHM, abhängig vom verwendeten Bytecode, verschieden viele Ausgaberegister benutzten kann. So entstehen nur Veränderungen an den relevanten Stellen der Ausgabe und Fehler beim Auslesen der Daten können vermieden werden. Ähnlich funktioniert das Verfahren beim Übernehmen der Eingabewerte. Es werden die ersten *in_offset*-vielen Bits in das Register übernommen und die restlichen Registerzellen werden mit '0' initialisiert. Dies ist nötig, da eventuell vorhandene Werte aus vorherigen Rechenschritten das Ergebnis beeinflussen könnten. Durch den Berechnungstakt(A.2) wird der VHDL Prozess update(↗D.6), welcher die genannten Funktionen beinhaltet

gesteuert. Bei jeder Änderung wird überprüft ob *restart* den Wert '1' - eben beschriebenes Verhalten - besitzt oder der Berechnungstakt den Wert '0'. Ist dies der Fall haben die FUs eine Operation ausgeführt und die berechneten Zwischenergebnisse werden in das Register übernommen. Nachzulesen ist die VHDL-Beschreibung dieses Prozesses im Anhang unter D.6.

5.7 Schieberegister

Da die zu verarbeitenden Daten als Bitstrom vorliegen, muss dieser nach einer erfolgreichen Verarbeitung eines Teils des Bytecodes nach links geschoben werden. Dadurch stehen immer die aktuellen Daten am Anfang des Datenstroms. Das Schieberegister kann einen Datenstrom der doppelten Datenbusbreite(↗A.3), um maximal diese nach links schieben. Dafür wird dem Schieberegister ein Bitstrom der Länge 2xDB und eine 6 Bit Zahl übergeben, da das Register zurzeit für eine Datenbusbreite von 32 Bit ausgelegt ist. Um die nachgeladenen Daten an den vorhandenen Datenstrom anhängen zu können, müssen diese eventuell auch geschoben werden. Deshalb kann das Schieberegister parallel zum Schieben des Datenstroms auch noch ein Signal der Länge des Datenbusses um DB-viele Bits nach links schieben und ein Signal der doppelten Datenbusbreite zurückgeben. Es wird ein zweites 6 Bit Signal benutzt, um die zweite Verschiebungsweite zu bestimmen. Die zurückgelieferten Signale können dann zu dem neuen Datenstrom verknüpft werden. Dies geschieht im Decoder(5.2) im Prozess shift(5.2.3), durch das Verbinden der beiden Ströme mit Hilfe einer ODER-Operation. Ein Beispiel ist im Abschnitt 5.2.5 beschrieben und der VHDL-Code dieser Komponente kann unter D.4 betrachtet werden.

5.8 Sequenzer

Der Sequenzer steuert die FUs(5.3) und das Register(5.6). Dies geschieht, indem der von außen angelegte Takt(*clk_in*), welcher in dieser Ausarbeitung auch als Berechnungstakt bezeichnet wird, als Signal *clk_out* an die FUs und das Register weitergeleitet wird. Er unterbricht die Verarbeitung der Daten, d. h. *clk_out* wird auf '0' gesetzt, wenn der Decoder(5.2) einen neuen Bytecode dekodiert. Das Signal *run* besitzt während der Dekodierung den Wert '0'. Ist dies der Fall wird außerdem den FUs mitgeteilt, dass der Befehlszähler auf 0 zurückgesetzt werden soll. Dafür benutzt der Sequenzer das Signal *reset_fu*. Es hat den Wert '1', wenn die FUs zurückgesetzt werden sollen und den Wert '0', wenn die Verarbeitung durchgeführt werden soll. Das Signal *reset_fu* wird auf '0' gesetzt, wenn der Takt als Signal *clk_out* an die anderen Komponenten weitergeleitet wird. Die genaue Beschreibung des Prozesses synch findet man im Abschnitt 6.2.

5.9 Package constants

In diesem Package werden die Eckdaten der VHM zur Designzeit festgelegt. Die Daten legen die Anzahl der benutzten Komponenten, deren Größe sowie die Größe der Datenpfade fest. Anhang A beschreibt diese und einige andere interne Signale näher. Tabelle 5.5 zeigt die zurzeit im Package definierten Signale.

Signal	Beschreibung
Datenbusbreite	Breite des externen Datenbusses(↗A.3)
FU-Adressbreite	Bits um FU-Nummer zu adressieren(↗A.5)
FU-Anzahl	Anzahl der FUs(↗A.5)
Registeradressbreite	Bits um Register zu adressieren(↗A.10)
Registeranzahl	Anzahl der vorhandenen Register(↗A.10)
Opcodelänge	Länge des Operationscodes(↗A.9)
Instruktionslänge	Bitanzahl einer Instruktion(↗A.8)
Instruktionscachegrösse	Anzahl der Instruktionen im Cache(↗A.7)

Tabelle 5.5: Die Signale des Package constants

Aus diesen Signalen ergeben sich einige Datentypen, die bei der Erstellung der Komponenten benutzt wurden. Diese erleichtern die Wartung des VHDL-Codes, da bei einer Änderung eines Signals automatisch die entsprechenden Komponenten angepasst werden. Dadurch werden Fehler vermieden und die Anpassung der Komponenten vereinfacht. Die definierten Datentypen sind in Tabelle 5.6 aufgelistet.

Datentyp	Abgeleitet von
vhm_reg	std_logic_vector(0 to reg_count-1)
op_code	std_logic_vector(0 to opcode_length-1)
operand	std_logic_vector(0 to reg_address-1)
instruction	record(OpCode:op_code; Input1:operand; Input2:operand)
complete_instruction	array(0 to fu_count-1) of instruction
instruction_cache	array(0 to fu_cache-1) of instruction
choose_fu	natural range 0 to fu_count-1
reg_offset	natural range 0 to reg_count

Tabelle 5.6: Die Datentypen des Package constants

5.10 VHM

Die VHM ist eigentlich keine richtige Komponente. Sie dient als Rahmen für die einzelnen Komponenten und als Top-Level-Entity[5] des VHDL-Projektes. Sie stellt die Verbindung zur Umgebung dar. Nur ihre Schnittstelle ist von außen sichtbar. Die Struktur dieser Schnittstelle ist im Abschnitt B.8 aufgelistet. Ihr werden die Daten des Bytecodes übergeben(data_in), die Eingabewerte(input) gesetzt, die Ergebnisse(output) und die Statusmeldungen(error) ausgelesen und der Takt(vhm_clk_in) festgelegt. Außerdem übernimmt sie die Aufgaben des Verbindungsnetzwerkes. In ihr wird festgelegt, welche Ein- und Ausgangssignale welchen internen Komponenten zugeordnet werden und wie diese Komponenten untereinander verbunden sind. In der jetzigen Version ist jede FU vollständig mit dem Register verbunden. Das bedeutet, jede FU kann lesend auf jede Registerzelle zugreifen. Diese Verbindungsart erlaubt das Ausführen aller Bytecodes, verbraucht aber bei größerer Anzahl der FUs oder sehr vielen Registerzellen relativ viele Ressourcen. Andere Möglichkeiten werden im Abschnitt 8.3.2 diskutiert.

[5] 1. Stelle in der Hierarchie der Komponenten

6 Das Zusammenspiel der Komponenten

Im vorherigen Kaptitel wurden die erstellten Komponenten und ihre Aufgaben vorgestellt. Ihre Funktionsweise wurde ebenfalls erläutert. Nun wird beschrieben, wie die Komponenten miteinander interagieren und welche Signale dabei transportiert werden, um die Aufgabe der virtuellen Hardware-Maschine, die Verarbeitung einer Schaltungsbeschreibung durch den VHBC, zu erfüllen. Wiederum ist die Auflistung der Abschnitte alphabetisch angeordnet und hat keinerlei Bezug zur Komplexität oder Wertigkeit der beschriebenen Vorgänge. Im Anhang B sind ab Seite 67 die Schnittstellen der Komponenten dargestellt. Damit und mit den in diesem Kapitel gezeigten Grafiken können die, von den Komponenten zur Kommunikation, benutzten Signale erkannt und die Funktionsweise leichter verstanden werden.

6.1 Decoder, FU und FU-Multiplexer

Die vom Decoder(5.2) aus dem Bytecode erstellte Instruktion(*instruction*) wird an den FU-Multiplexer(5.4) weitergeleitet. Für welche FU diese Instruktion bestimmt ist wird im Decoder mitgezählt(*choose_fu*) und ebenfalls dem FU-Multiplexer mitgeteilt. Dieser übergibt dann diese Instruktion(*instruction(choose_fu)*) an die entsprechende FU(5.3). Dort wird sie in deren Instruktionscache eingetragen. Wird eine EOB[1]-Instruktion übergeben, leitet der FU-Multiplexer eine MOV[2]-Instruktion an alle verbleibenden FUs weiter. Dadurch ist gewährleistet, dass alle FUs dieselbe Anzahl von Operationen in ihren jeweiligen Instruktionscaches besitzen. Um den Zugriff auf die richtigen Register zu gewährleisten, müssen die übergebenen Operanden nochmals umgedreht werden. Das bedeutet, sie werden nun vom niederwertigsten Bit(LSB[3]) zum höchstwertigsten Bit(MSB[4]) angeordnet. Die umgedrehte Anordnung der Daten stammt aus der Dekodierung des Bytecodes im Decoder(5.2) und ist im Abschnitt 5.4.2 beschrieben. Durch diesen Vorgang greifen die einzelnen FUs auf die, im Bytecode angegebenen, Register

[1] end of block
[2] den Eingabewert unverändert ausgeben
[3] least significant Bit
[4] most significant Bit

beim Laden der Operanden zu. Abbildung 6.1 stellt grafisch die Verbindungen der benannten Komponenten dar.

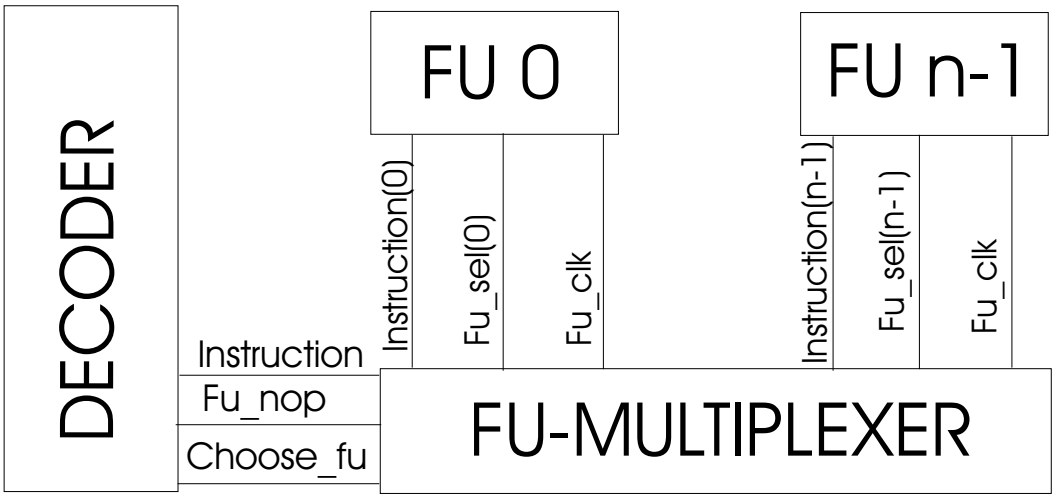

Abbildung 6.1: Decoder, FU und FU-Multiplexer

6.2 Decoder, FU, Register und Sequenzer

Der von außen angelegte Takt(Signal *clk*), wird vom Sequenzer(5.8) an die FUs(5.3) und das Register(5.6) weitergegeben. Falls der Decoder(5.2) gerade einen neuen Bytecode verarbeitet(*run* besitzt den Wert '0'), setzt der Sequenzer das Taktsignal auf den Wert '0', womit die Verarbeitung gestoppt wird. Durch setzen des Signals *reset* auf den Wert '1', wird den FUs mitgeteilt, dass neue Instruktionen decodiert werden, welche dann in dem jeweiligen Instruktionscache der FUs abgelegt werden. Ist die Dekodierung beendet, wird der Takt wieder weitergeleitet und die Berechnung der neuen Schaltung ausgeführt. Die Ergebnisse der FUs werden im Register abgelegt, wobei jede FU exklusiv auf genau eine Registerzelle schreibend zugreift(*reg(FU-NR.)*). Dagegen kann jede FU auf alle Registerzellen(*reg_in*) lesend zugreifen, um die Operanden der Operation zu holen. Löst die FU-Nr. 0 den Schaltungstakt(*restart*='1') aus, werden die Ergebnisse nach außen weitergegeben und die neuen Eingabewerte in das Register übernommen. Die einzelnen Signale sind in Grafik 6.2 abgebildet.

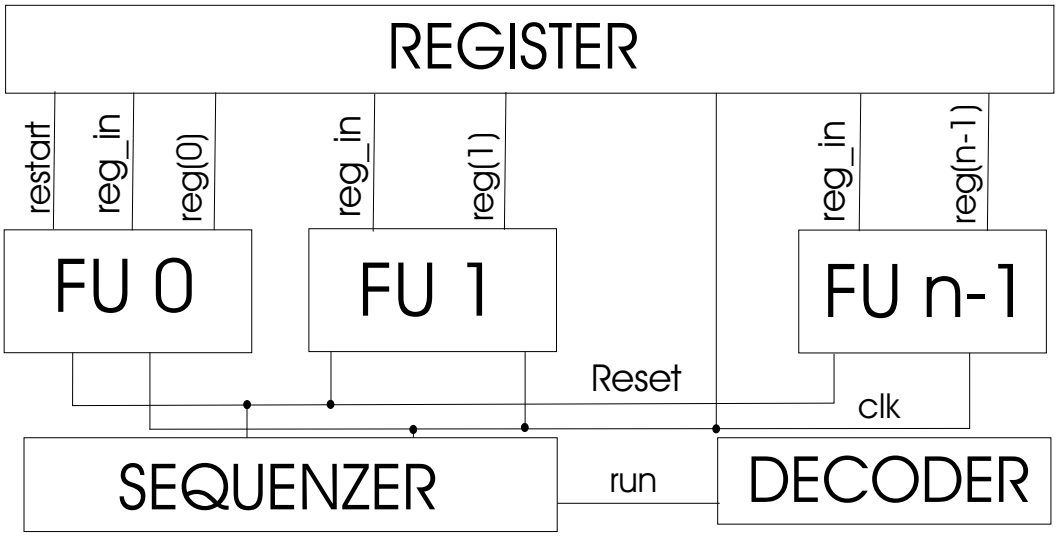

Abbildung 6.2: Decoder, FU, Register und Sequenzer

6.3 Decoder, Konstanten-Multiplexer und Register

Beim Dekodieren des Headers, des angegebenen Bytecodes, werden die Werte für die Konstanten(*const_reg*), sowie die Anzahl der Eingabewerte(*in_offset*) und der Ausgabewerte(*out_offset*) im Decoder(5.2) gespeichert. Im Konstanten-Multiplexer(5.5) werden die Konstanten mit den Eingabewerten verbunden und dann als Eingaben(*cm_output*) an das Register(5.6) weitergegeben. Das Register erhält vom Decoder die Anzahl der Ausgabewerte(AW) und die Summe(EW) aus der Anzahl der Konstanten(*const_pos*) und der Anzahl der Eingabewerte. Es werden die ersten EW-Bits aus dem Konstanten-Multiplexer in das Register übernommen und die ersten AW-Bits als Ausgabewerte aus dem Register gelesen. Die restlichen Bits werden mit '0' initialisiert. Dadurch wird versucht Fehlberechnungen und falsche Rückgabewerte zu vermeiden. Die verwendeten Signale stellt Grafik 6.3 dar.

6.4 Decoder und Schieberegister

Um immer die aktuellen Daten am Anfang des Datenstroms zu haben, muss dieser, nach der Abarbeitung, um die Anzahl der zuletzt benutzten Bits nach links geschoben werden. Dazu werden dem Schieberegister(5.7) der aktuelle Datenstrom(*data_in*), die Verschiebungsbreite(*sel*), die nachgeladenen Daten(*data_in_mir*) und deren Verschiebungsbreite(*sel_mir*) übergeben. Die Verschiebungsbreiten sind jeweils 6 Bit Zahlen, da das Register zurzeit auf einen 32 Bit breiten Datenbus ausgelegt ist. Das heißt, der aktuelle Datenstrom beträgt 64 Bit und die nachgeladenen Daten sind 32 Bit lang. Ver-

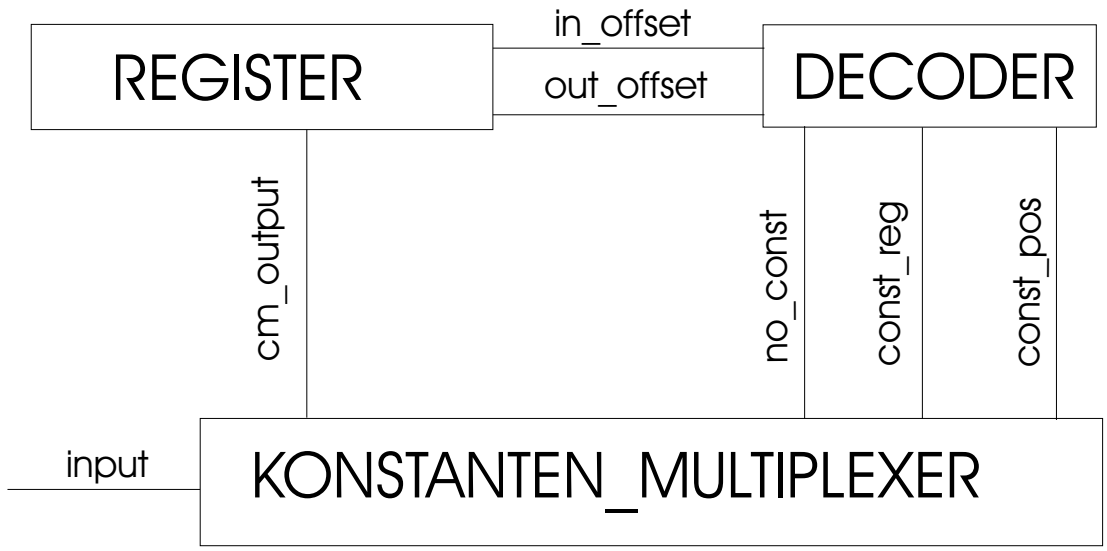

Abbildung 6.3: Decoder,Konstanten-Multiplexer und Register

schoben werden sie um maximal 32 Stellen nach links. Hat der Decoder einen Wert aus dem Datenstrom dekodiert, werden die entsprechenden Signale belegt und das Signal *sh_clk* verändert. Ist der Schiebevorgang beendet, wird dies durch eine Veränderung des Signals *sh_ready* angezeigt. Der Dekoder verbindet nun eventuell die geschobenen Daten(*data_out* und *data_out_mir*) mittels einer oder-Verknüpfung und bestimmt den nächsten Wert. Dies geschieht solange, bis das Ende des Bytecodes erreicht ist. Abbildung 6.4 stellt grafisch die benutzten Signale dar.

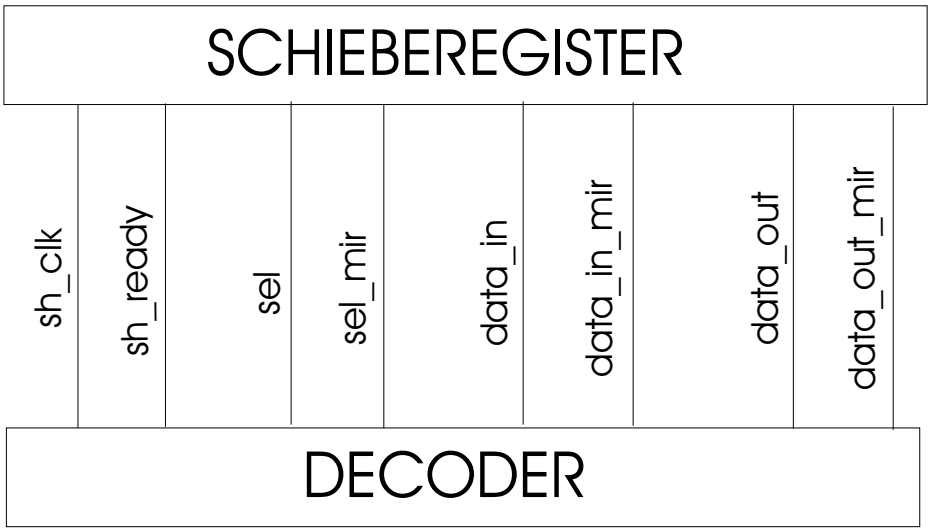

Abbildung 6.4: Decoder und Schieberegister

6.5 VHM mit der Umgebung

Die VHM wird von außen mit einem Takt(*vhm_clk*) versorgt, der die Berechnungsgeschwindigkeit der VHM bestimmt. Außerdem werden die Eingaben(*input*) an die VHM angelegt und die Ausgaben(*output*) der Berechnung gelesen. Das Signal *update* startet die Dekodierung eines neuen Bytecodes und durch das Signal *error* kann der Status der Verarbeitung(↗A.4) überprüft werden. Über den Datenbus(*bc_data*) kann der Decoder(5.2) den Bytecode nach und nach einlesen. Auch hier noch einmal die Darstellung der Signale in Grafik 6.5.

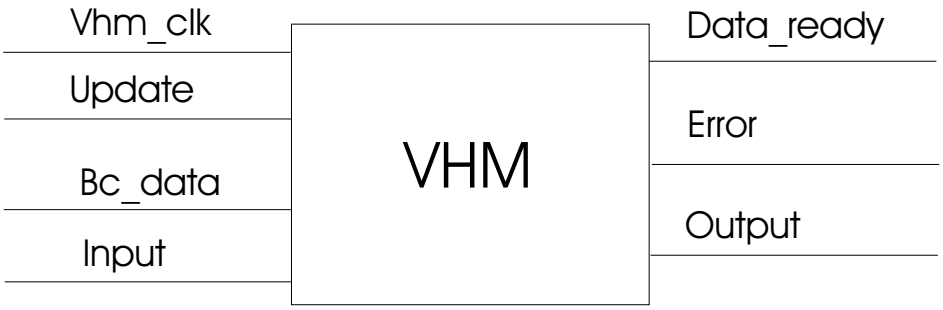

Abbildung 6.5: VHM

7 Ergebnisse

Dieses Kapitel stellt einige Ergebnisse der Diplomarbeit dar. Dies sind zum einen die bei der Entwicklung des Designs entstandenen Versionen des Schieberegisters - einer zentralen Komponente der VHM. Zum anderen die Flexibilität der VHM und die dadurch entstehenden verschiedenen Implementierungen auf derselben Hardwareressource. Als letzter Teil wird die VHM, hinsichtlich Ressourcenverbrauch und Geschwindigkeit, mit direkt in VHDL beschriebenen Schaltungen verglichen.

7.1 Umsetzung der virtuellen Hardware-Maschine

7.1.1 Anzahl der FUs und Größe des Instruktionscache

Beim Design der virtuellen Hardware-Maschine wurde darauf geachtet, dass es möglichst flexibel gestaltet wird. Das bedeutet, die Anzahl der Functional Units, die Anzahl der Instruktionen im Instruktionscache und die Breite des Registers sind leicht zu verändern. So kann eine Implementierung der VHM auf die Zielarchitektur und die zu erwartenden Aufgaben angepasst werden. Ist auf dem verwendeten FPGA nicht genug Platz, kann eine Implementierung mit weniger FUs benutzt werden. Ebenso kann die VHM auf stark parallelisierende aber relativ kurze Schaltungen oder umgekehrt ausgelegt werden. Alle Implementierungen benutzen denselben Hardware-Bytecode und entscheiden selbstständig, ob dieser auf ihr ausführbar ist. Im Package constants(5.9) werden diese Eckdaten festgelegt. Die Anzahl der FUs wird durch das Signal FUZ[1](↗A.5) bestimmt. Diese ist in den folgenden Tests immer gleich der Registerbreite(↗A.10), da ja jede FU auf genau ein Register schreibend zugreift. Die Breite des Registers darf nicht kleiner, kann aber größer sein. Allerdings können diese Werte des Registers dann nicht von den FUs verändert werden. Es würde sich dann anbieten, dort Konstanten oder Eingabewerte bei Schaltungen mit vielen Eingaben und wenig Ausgaben abzulegen. Das Signal ICG[2](↗A.7) bestimmt die Größe des Instruktionscaches. Dadurch wird bestimmt, aus wie vielen Instruktionsblöcken eine Schaltung bestehen kann. D. h. , wie groß die Anzahl

[1] Anzahl der benutzten FUs
[2] Instruktionscachegrösse

der Operationen ist die eine FU ausführt. In der Tabelle 7.1 sind ein paar mögliche Konfigurationen der VHM aufgelistet. Um die Daten besser einschätzen zu können, wird nicht der reine Ressourcenverbrauch, sondern die Auslastung eines Beispiel-FGPAs, angegeben. Benutzt wird hierfür ein Chip der Virtex-Reihe der Firma XiLinx[3], genauer gesagt der XCV800BG432. Ein Datenblatt lässt sich unter [Xil01] herunterladen.

FU-Anzahl	Cachegrösse	Slices	Slice Register	Lookup Tables
2	16	7,50%	4,23%	5,94%
2	32	10,09%	5,27%	6,78%
2	50	13,05%	6,45%	7,68%
2	100	21,89%	9,78%	10,42%
2	256	45,83%	19,87%	17,72%
4	4	7%	3,84%	6%
4	16	13,29%	6,22%	7,96%
4	32	20,31%	9,00%	10,00%
4	50	28,48%	12,15%	12,40%
4	100	49,89%	20,96%	18,59%
4	200	92,39%	38,39%	30,81%
8	1	7,63%	3,67%	6,92%
8	5	13,5%	5,52%	10,5%
8	16	21,81%	10,82%	12,15%
8	32	37,09%	17,77%	17,05%
8	50	54,84%	25,58%	22,79%
8	100	99,98%	47,57%	37,63%
11	7	20,16%	8,52%	14,72%
16	16	45,59%	21,41%	22,23%
16	32	84,20%	38,00%	33,54%
16	50	101,35%	56,79%	46,98%
32	16	97,41%	45,72%	46,62%
32	32	136,13%	84,52%	72,31%
32	50	209,95%	127,88%	102,94%

Tabelle 7.1: Die Implementierungsgrößen der VHM

[3] siehe auch www.xilinx.com

7.1.2 Einfluss des Schieberegisters

Das Schieberegister ist ein relativ komplexer Teil der virtuellen Hardware-Maschine, da zur Designzeit die Werte, um die die Daten geschoben werden sollen, nicht bekannt sind. Zurzeit ist es auf eine Datenlänge von maximal 64 Bit ausgelegt. Die Daten können um maximal 32 Bit nach links geschoben werden. Die Länge ergibt sich aus der aktuellen Breite des Datenbusses, welche 32 Bit beträgt. Dieser Wert wird benutzt, da die größten Werte im Header des Bytecodes 32 Bit lang sind. Ein kleinerer Datenbus würde zu häufigerem Nachladen führen, während ein größerer Datenbus die Größe des Designs stark ansteigen lassen würde. Durch den variablen Wert der Verschiebung, muss das Register in der Lage sein, die Daten um 0 bis 32 Stellen nach links zu schieben. Für die Angabe des Wertes wird eine 6 Bit Zahl benutzt. Es ergeben sich nun mehrere Möglichkeiten, die Daten zu verschieben. Durch die Wahl der Anzahl der Schritte, in denen die Daten geschoben werden, wird die Größe der erstellten Komponente maßgeblich beeinflusst. Bei einer 6 Bit Zahl bieten sich hier die Schrittanzahlen 1,2,3 und 6 an. Bei diesen Werten wird in jedem Schritt eine gleichgroße Anzahl von Bits ausgewertet. Die Tabelle 7.2 zeigt einige Eckdaten der dadurch entstehenden Komponente.

	1 Takt	2 Takte	3 Takte	6 Takte	unsigned
Slices	917	307	289	345	1220
Slice Flip Flops	95	100	213	546	1
4 Input LUT's	1804	608	568	664	2402
Total Gate Count	15013	464990	6692	9392	234868

Tabelle 7.2: Die Implementationsgrössen des Schieberegisters

Man erkennt, dass ein Verschieben der Daten in 3 Schritten die günstigste Lösung ist. Hier werden jeweils 2 Bit ausgewertet und die Daten dann entsprechend geschoben. Das Auswerten von 2 Bit bedeutet, dass es 4 mögliche Zustände gibt. Es müssen also 3 mal 4 mögliche Verschiebungen realisierbar sein. Wenn man die Verschiebung in 2 Schritten durchführt, das heißt jeweils 3 Bit betrachtet, sind dies 2 mal 8 Zustände. Die Spalte mit dem Namen unsigned stellt die Werte einer Implementierung mit dem Standarddatentyp unsigned dar. Dieser Datentyp besitzt eine shift-Funktion, welche es erlaubt die Länge der Verschiebung mit anzugeben. Der in diesem Abschnitt verwendete Takt ist nicht der Takt den die VHM benutzt. Er ist als Anzahl der benötigten Schritte zum Schieben der Daten zu werten. Die Geschwindigkeit hängt hier von der verwendeten Hardware ab, denn während des Dekodierens ist der von außen angelegte Takt durch den Sequenzer unterbrochen(↗6.2). Die shift_left-Funktion des Datentyps unsigned versucht, die Daten in einem Schritt zu verschieben und verbraucht deshalb so viele Ressourcen. Zur besseren

Übersicht am Ende noch der prozentuale Vergleich der verbrauchten Ressourcen. Das verwendete Design mit den 3 Takten wird als 100%-Marke benutzt. Tabelle 7.3 stellt die beschriebene Übersicht dar.

	1 Takt	2 Takte	3 Takte	6 Takte	unsigned
Slices	317%	106%	100%	119%	422%
Slice Flip Flops	44,6%	46,9%	100%	256%	0,47%
4 Input LUT's	318%	107%	100%	117%	423%
Total Gate Count	224%	6948%	100%	140%	3510%

Tabelle 7.3: Der prozentuale Ressourcenverbrauch des Schieberegisters

7.2 Vergleich mit direkten FPGA-Implementierungen

Es ist natürlich klar, dass eine Implementierung der virtuellen Hardware-Maschine im Vergleich zu direkten FPGA-Implementierungen mehr Ressourcen verbraucht und mit geringerer Geschwindigkeit abläuft. Dies kommt durch die fehlenden Optimierungsmöglichkeiten für die einzelnen Anwendungen. Auf einer VHM sollen ja alle Schaltungen abbildbar sein. Es können nicht immer alle FU komplett ausgenutzt werden und da jede FU mehrere verschiedene Instruktionen ausführen kann, ist hier ebenfalls ein beträchtlicher Overhead vorhanden. Bei einer direkten Implementierung kann das Synthesetool alle Möglichkeiten des FPGAs ausnutzen, um diese eine Anwendung optimal laufen zu lassen. Ein beträchtlicher Unterschied im Ressourcenverbrauch und der Geschwindigkeit ist also zu erwarten. Interessant ist es aber zu sehen, wie groß dieser Unterschied ist. In den folgenden Abschnitten werden der Ressourcenverbrauch und die Geschwindigkeit anhand von einigen kleinen Beispielen verglichen. Der Vorteil der VHM liegt dagegen in der Flexibilität. Sie kann verschiedene Anwendungen ausführen. Es muss nur der entsprechende Bytecode und nicht eine komplette FPGA-Konfiguration erstellt werden. Außerdem ist eine Konfiguration für den hier zum Testen benutzten Xilinx XCV800 FPGA 575KB groß. Ein Bytecode belegt je nach Anzahl der Operationen jedoch nur wenige KB. Das bedeutet, dass die Übertragung zum FPGA schneller abläuft und da nur einige Register(der Instruktionscache), nicht das komplette FPGA, beim Dekodieren des Bytecode neu beschrieben werden müssen, läuft der gesamte Neukonfigurationsvorgang wesentlich schneller ab. Da sich keine vollständig lauffähige VHM-Implementierung erstellen ließ, kann hier leider keine Zeitersparnis angegeben werden.

7.2.1 Ressourcenverbrauch

Wie oben schon beschrieben, verbraucht eine Implementierung einer VHM mehr Ressourcen als die direkte Implementierung auf dem FPGA, ist dafür aber flexibler. Im Folgenden werden einige Beispiele genannt und deren Ressourcenverbrauch dargestellt. Verglichen werden diese Werte jeweils mit denen der jeweiligen, zur Ausführung dieser Schaltung benötigten, VHM. Dargestellt wird der Ressourcenverbrauch der direkten Implementierungen in der Tabelle 7.4 und der der verschiedenen VHMs in Tabelle 7.5. Der Ressourcenverbrauch weiterer Beispielimplementierungen der VHM wurde bereits in der Tabelle 7.1 aufgelistet. Die Erklärung der Spaltennamen folgt unterhalb der Tabellen.

	Add	2BitCount	Count	7Seg	shift	Fulladd
Slices	2	1	2	4	16	1
4 Input LUT's	4	1	3	7	22	2

Tabelle 7.4: Ressourcenverbrauch bei einfachen Schaltungen

	Add	2BitCount	Count	7Seg	shift	Fulladd
Slices	1897	1897	659		718	1270
4 Input LUT's	2769	2769	1129		1302	1039

Tabelle 7.5: Ressourcenverbrauch der VHM bei einfachen Schaltungen

Add: bezeichnet die Addition zweier 4 Bit Eingaben zu einer 5 Bit Zahl. Eine VHM benötigt 11 FUs und 7 Takte um diese Addition auszuführen.

2BitCount: Es wird ein 2 Bitzähler hochgezählt. Dafür wird eine VHM mit 11 FUs gebraucht die den aktuellen Wert des Zählers dann nach 7 Takten wieder ausgibt.

Count: steht für einen 4 Bit Counter. Es wird die Anzahl der Einsen in der 4 Bit Eingabe gezählt und als 3 Bit Ausgabe zurückgegeben. Um diesen Algorithmus in einer VHM zu implementieren, werden 4 FUs und ein Instruktionscache von 3 Anweisungen benötigt.

7Seg: stellt eine 7 Segmentanzeige dar, bei der eine 4 Bit Zahl angezeigt wird. Die Werte werden als Hexadezimalzahl angezeigt. Eingaben die größer als 9 sind, werden also durch die Buchstaben a bis f dargestellt. Durch die Funktionsweise des Compilers entsteht ein Bytecode der 69 FUs benötigen würde und dann 10 Takte bräuchte, um das Ergebnis auf der 7-Segmentanzeige auszugeben. In Tabelle 7.1 erkennt man allerdings leicht, dass sich eine so große VHM nicht auf dem verwendeten FPGA abbilden lässt.

Shift: wurde benutzt, da dieser Algorithmus die Parallelität der VHM gut ausnutzt. Es wird eine 8 Bit Eingabe um eine Stelle nach links verschoben. Das erste Bit der Eingabe kommt an die letzte Stelle der Ausgabe. Hierfür wird eine VHM mit 8 FUs und nur einer Instruktion pro Instruktionscache benötigt.

Fulladd: bezeichnet einen Volladdierer. Es werden 3 Bit als Eingabe benutzt und 2 Bit als Ausgabe. Erstellt man hieraus einen VHBC, würde man 19 FUs und 11 Takte benötigten. Erstellt man den Bytecode von Hand durch die Darstelltung der Funktion als DNF[4], werden nur noch 8 FUs und 5 Takte benötigt. Nutzt man jetzt auch noch die Funktionsvielfalt der FUs aus und benutzt die Operationen XOR und EQ, werden sogar nur noch 6 FUs und 4 Takte gebraucht, um die Schaltung zu berechen. In den folgenden Tabellen werden die Werte für eine VHM mit 8 FUs benutzt, da beim Erstellen meist Potenzen von 2 für die FUZ[5](↗A.5) und die RB[6](↗A.10) benutzt wurden.

Tabelle 7.6 stellt dar, wie viel Prozent der verbrauchten Ressourcen einer VHM-Implementierung bei der direkten Implementierung benötigt werden. Das heißt, ein Wert von 10% bedeutet, dass die direkte Implementierung 10% der Ressourcen der VHM-Implementierung verbraucht.

	Add	2BitCount	Count	7Seg	Shift	Fulladd
Slices	0,11%	0,05%	0,3%		2,23%	0,08%
4 Input LUT's	0,14%	0,04%	0,27%		1,67%	0,19%

Tabelle 7.6: Vergleich des Ressourcenverbrauchs bei einfachen Schaltungen

7.2.2 Geschwindigkeit

Die Geschwindigkeit der VHM und der einfachen Beispiele konnte nicht gemessen werden. Die hier benutzten Werte sind die bei der Synthese erhaltenen Werte für die schnellstmögliche Abarbeitung. Es wurden die Synthesewerkezeuge XST[7], FPGAExpress und SpectrumVHDL benutzt. Wenn bei mehreren Programmen verwertbare Abschätzungen herauskamen wurde ein Mittelwert gebildet. Die benutzten Beispiele sind dieselben wie im vorherigen Abschnitt. Die Berechnungsgeschwindigket der VHM liegt ungefähr bei 80 MHz pro Takt. Tabelle 7.7 vergleicht die Geschwindigkeit der Beispielschaltungen mit der ihrer äquivalenten VHM-Implementierungen. Da die VHM die Schaltungen

[4] disjunktive Normalform
[5] Anzahl der FUs
[6] Anzahl der Registerzellen
[7] xilinx synthesis technology

nicht in einem Takt berechnen, kann wird die Anzahl der benötigten Takte aufgelistet. Die Zeile Performance gibt an, mit wie viel Prozent der Geschwindigkeit der direkten Implementierung die entsprechende VHM läuft. Ein Wert von 50% bedeutet also, dass die VHM die Schaltung halb so schnell berechnet wie eine direkte Implementierung auf dem FPGA.

	Add	2BitCount	Count	7Seg	Shift	Fulladd
Geschwindigkeit in MHz	230	233	265	285	400	300
Taktanzahl	7	7	3	11	1	5(4)
Performance in %	4,97	4,90	10,06	2,55	20	5,33(6,67)

Tabelle 7.7: Vergleich der Geschwindigkeit bei einfachen Schaltungen

7.2.3 Zusammenfassung

	RB=8 & ICG=4	RB=4 & ICG=4
Slices	63	42
4 Input LUT's	61	33

Tabelle 7.8: Ressourcenverbrauchs einer einzelnen FU

Wie in Tabelle 7.8 zu erkennen ist, verbraucht eine einzige FU schon mehr Ressourcen als die Beispielanwendungen. Es konnten aber auch keine komplexeren Schaltungen gewählt werden, da der VHBC-Compiler im Momemt völlig auf möglichst schnelle Abarbeitung getrimmt ist. Da die Anzahl der vorhandenen FUs nicht eingeschränkt ist, erstellt der Compiler so viele FUs wie er gerade benötigt. Diese werden aber meist nur in einem Takt benutzt. Danach werden den meisten nur NOP[8]s zugewiesen. Da alle Ausgabewerte am Anfang des Registers stehen, müssen die berechneten Zwischenergebnisse erst wieder an den Anfang gebracht werden. Dies kostet auch mindestens einen Takt. Deshalb konnten nur Beispiele benutzt werden, für die ein relativ kompakter Bytecode entsteht. Bei komplexeren Schaltungen ist auch eine Optimierung von Hand nicht mehr möglich. Wie viel dies bringen kann, sieht man am Beispiel des Volladdierers, wobei der von Hand erstellte Bytecode wahrscheinlich noch nicht einmal das Optimum ist. Wenn die Parallelität der VHM gut ausgenutzt werden kann, liegt die Geschwindigkeit, im Vergleich zu einer direkten Implementierung, gar nicht so weit darunter. Allerdings wird sich dieser Fall nicht allzuoft realisieren lassen. Man muss den Compiler dazu bringen die vorhandenen FUs gleichmässiger auszulasten. Entweder man kennt die vorhandene Anzahl

[8]no operation - keinen Befehl ausführen

der FUs und setzt eine Obergrenze, oder man benutzt einen Optimierungsalgorithmus der versucht einen Kompromiss zwischen der Taktanzahl und der maximalen Anzahl der FUs zu finden. Da ein Bytecode auf verschiedenen Implementierungen der VHM lauffähig sein soll, ist der erste Weg wahrscheinlich nicht akzeptabel. Für die 2. Variante müssten sicherlich im Rahmen einer Diplomarbeit vorhandene Optimierungslösungen überprüft und getestet werden, um herauszufinden mit welchem Optimierungsalgorithmus sich die Fähigkeiten der VHM am besten ausnutzen lassen. Der zweite wichtige Ansatzpunkt zur Verbesserung der VHM ist die Größe der FU zu verringern. Wie gesehen, verbrauchen diese durch den Instruktionscache und die variable Funktionsauswahl schon relativ viele Ressourcen. Functional Units die weniger Instruktionen beherrschen, aber dafür kleiner und schneller sind, könnten die Leistung verbessern. Weniger Flexibilität bei den FUs bedeutet aber wieder mehr Aufwand im Compiler. Außerdem muss darauf geachtet, dass der Bytecode weiterhin aus verschiedenen Implementierungen lauffähig bleibt und alle möglchen Schaltungen als Bytecode darstellbar sind. Diese und einige andere Verbesserungsvorschläge sind im nachfolgenden Kapitel noch einmal genauer angegeben.

8 Schlussfolgerungen

Abschließend werden in diesem Kapitel die in dieser Diplomarbeit erreichten Ziele noch einmal benannt, ein Ausblick auf zukünftige Erweiterungen der entwickelten Maschine gegeben, mögliche Verbesserungen des zur Schaltungsbeschreibung benutzten VHBC[1]s vorgeschlagen und die während der Arbeit entstanden Kenntnisse vorgestellt.

8.1 Was wurde erreicht

Eine VHDL-Beschreibung der virtuellen Hardware-Maschine wurde entwickelt. Diese lässt sich durch wenige Parameter flexibel an die vorhandene Hardware bzw. den gewünschten Verwendungszweck(↗7.1.1) anpassen. Eine komplett testfähige VHM-Implementierung konnte nicht erstellt werden, so das keine Geschwindigkeiten gemessen werden konnten. Diese konnten nur abgeschätzt werden. Da sich die VHM-Implementierungen jedoch synthetisieren ließen, konnte deren Ressourcenverbrauch bestimmt werden.

8.2 Was habe ich gelernt

Während dieser Arbeit habe ich einen Einblick in die Schaltungsentwicklung mit VHDL bekommen. Ich erlernte die notwendigen Sprachkenntnisse, um eine Schaltung beschreiben und in Hardware abbilden zu können. Außerdem zeigte mir die Arbeit mit FPGAs die Vor- und Nachteile von Hardwarerealisierungen im Vergleich zu den von mir bisher ausschließlich gekannten und benutzen Softwareimplementierungen. Ein Umdenken, im Vergleich zur Softwareentwicklung, ist nötig, um Schaltungen auf Hardware abzubilden, da die Möglichkeiten der Hardware beachtet werden müssen, denn hier übernimmt nicht der Compiler einer höheren Programmiersprache die Abbildung auf die benutzte Architektur. VHDL übernimmt einen Teil dieser Arbeit, da man nur das Verhalten beschreiben muss, aber die Grenzen der Architektur werden einem schneller bewusst. Hardwarerealisierungen laufen wesentlich schneller ab, aber nicht alle Anwendungen können reali-

[1] virtual hardware bytecode

siert werden. Die Synthese und das Zuordnen der Ressourcen[2], können bei sehr großen Designs - nahezu komplette Auslastung des verwendeten Xilinx Chips - auf heutigen PC-Systemen durchaus sehr lange dauern. Insgesamt hat mir diese Arbeit einen Eindruck einer anderen Art der Anwendungsentwicklung gegeben, welche durch die größere Hardwarenähe auch ein besseres Verständnis der zugrunde liegenden Architektur in Computersystemen vermittelt.

8.3 Verbesserungen/Noch zu tun

In dieser Arbeit wurde eine erste synthetisierbare Version einer virtuellen Hardware-Maschine entwickelt. Vollständig lauffähig ist das erstellte Design noch nicht, so dass sich die Geschwindigkeit nur abschätzen lässt. Ein bearbeiten des VHDL-Codes, um die VHM komplett zum Laufen zu bekommen ist also noch nötig. Um die VHM benutzen zu können, muss vor allem noch die Kommunikation mit der Umgebung definiert und entsprechend umgesetzt werden. Mögliche Verbesserungsvorschläge, die die Funktionen der VHM erweitern, sind in diesen Abschnitt ebenfalls noch angegeben.

8.3.1 Erweiterungen des Hardware-Bytecodes

In diesen Abschnitt werden einige Ideen zur Veränderung des Bytecodes vorgestellt. Die VHM erwartet diese Änderungen nicht, aber sie sollten bei einer Erweiterung der Funktionalität betrachtet werden.

Opcodelänge abspeichern: erlaubt die Erweiterung der Opcodes, da z. Zt. 13 der 16 möglichen Opcodes benutzt werden. Eine Implementierung einer VHM könnte dann entscheiden, ob sie die benötigten Opcodes kennt und den Bytecode ausführen kann. Die Dekodierung könnte analog zu der der Operanden(↗5.4.2) funktionieren.

FU-Anzahl abspeichern: im Moment wird nur angegeben, wie viele Registerzellen benötigt werden und davon ausgegangen das genau so viele FUs benutzt werden. Es ist aber vorstellbar das mehr Registerzellen als FUs benutzt werden. Diese Art von Schaltungen könnte dann auch von der VHM ausgeführt werden.

[2]mapping und place&route

Instruktionenanzahl abspeichern: wenn die Anzahl der Instruktionen im Header des Bytecodes enthalten wäre, könnte die jeweilige VHM-Implementierung überprüfen ob der Cache der FUs ausreichend ist. Bisher werden die Daten eingelesen und es wird davon ausgegangen, dass genug Ressourcen vorhanden sind. Wie in Abschnitt 7.1.1 zu sehen ist, können aber sehr unterschiedliche Implementierungen der virtuellen Hardware-Maschine erstellt und benutzt werden.

8.3.2 Erweiterungen der VHM

Einige Änderungen an der VHM müssen auch noch vorgenommen werden, um sie in der gewünschten Umgebung benutzen zu können. Die bisherige Version ist soweit entwickelt, dass sie in einer Testumgebung den angelegten Bytecode verarbeiten kann. Es werden nur die zum Betrieb nötigen Werte verarbeitet, um eine möglichst Ressourcensparende Implementierung zu entwickeln. Die nachfolgenden Abschnitte beschreiben näher, welche Anpassungen noch entwickelt werden müssen, um sie zukünftig in der vorgesehen Umgebung reibungsfrei benutzen zu können.

Speicherinterface

Die erste Anpassung betrifft das Speicherinterface. Zurzeit wird auf der Testumgebung eine Komponente generiert, die den Bytecode enthält und mit der VHM kommuniziert. Bei der späteren Anwendung ist diese Komponente aber ein Bestandteil der umgebenden Architektur. Da noch nicht genau festgelegt ist, wie diese Kommunikation abläuft, muss wahrscheinlich der Prozess des Nachladens der Daten(↗5.2.2) angepasst werden. Es könnten Störungen oder Verzögerungen auftreten, auf die die virtuelle Hardware-Maschine entsprechend reagieren muss.

Vollständige Kompatibilität

Diese Anpassungen sind nicht erforderlich, um die virtuelle Hardware-Maschine laufen zu lassen, sollten aber um eine bessere Kompatibilität zu gewährleisten noch entwickelt werden. Bisher wird die im Bytecode vorhandene Versionsnummer nicht ausgewertet, da noch keine verschiedenen Bytecodeversionen existieren. Werden aber Bytecodeversionen, mit z. B. unterschiedlicher Länge des Operationscodes, entwickelt, könnte eine Auswertung dieses Headerwertes mögliche Fehler vermeiden. Außerdem ist in der Bytecodedefinition festgelegt, dass jeder Bytecode mit 4 Bytes, die dem Text 'VHBC' entsprechen, eingeleitet wird. Eine Überprüfung dieser Werte kann ebenfalls stattfinden, um die angelegten Daten als Bytecode zu verifizieren.

8.3.3 Änderungen des FU-Aufbaus

Den Bytecode kompakter gestalten oder die Komplexität der Functional Units zu verringern, könnte sich vorteilhaft auf die Leistung und den Ressourcenverbrauch der VHM auswirken. Einige Möglichkeiten werden im Folgenden vorgestellt.

FU feste Instruktion zuordnen

Um den Bytecode kompakter werden zu lassen, wäre es möglich jeder FU genau einen Befehl zuzuordnen. Dadurch könnte der Operationscode weggelassen werden. Allerdings ist erst eine Untersuchung notwendig, in welchem Verhältnis die verschiedenen Instruktionen im Durchschnitt benutzt werden, um möglichst viele Schaltungen effizient abbilden zu können.

Ausgangsregister mitangeben

Eine andere Möglichkeit ist es den FUs auch das Ausgangsregister mit anzugeben. Dadurch würde zwar der Bytecode aufgebläht, aber vielleicht entstehen so mehr Optimierungsmöglichkeiten, falls nicht wie bisher jede FU auf alle Register zugreifen kann. Andere Verdrahtungsmöglichkeiten werden später in diesem Kapitel aufgezeigt.

Feste Instruktion und Ausgangsregister angeben

Lässt man, wie im vorherigen Punkt beschrieben, den Operationscode weg und gibt wie oben aufgezeigt auch das Ausgangsregister der Operation mit an, würde sich die Größe des Bytecodes kaum verändern und er könnte vielleicht besser optimiert werden. Der Vorteil läge dann in der Benutzung von kleineren, und wahrscheinlich schnelleren, Funktionseinheiten. Allerdings wäre auch der Verdrahtungsaufwand an den Ausgängen der FUs wesentlich höher und es muss sichergestellt werden, dass immer nur eine FU auf eine Registerzelle schreibend zugreift.

Änderungen des Verbindungsnetzwerkes

Als Verbindungsnetzwerk wird die Verbindung zwischen den Registern und der FUs bezeichnet. Da in der Entwicklung der virtuellen Hardware-Maschine festgelegt ist, dass jede FU das Ergebnis in genau eine - in immer die gleiche - Registerzelle schreibt, kann an diesem Teil des Verbindungsnetzwerkes nichts verändert werden. Möglichkeiten zum Ressourcensparen bleiben also nur beim lesenden Zugriff auf die Register. In der aktuellen Implementierung kann jede FU auf alle Registerzellen zugreifen. Dies erlaubt das ausführen aller möglichen Bytecodes, verbraucht aber für jede FU RB[3](↗A.10)-viele Leitungen. Es werden also $LZ = FUZ * RB$-viele Leitungen für das Auslesen der Operanden benutzt.

Es gibt 2 Ansätze diese Zahl zu verringern. Ein vollständiger Zugriff auf alle Operanden bleibt erhalten oder der Zugriff einer FU auf die Registerzellen wird begrenzt. Der 2. Fall bedingt allerdings eine Anpassung des Compilers, da die Art des Registerzugriffs betrachtet werden muss.

Eine möglicher Zugriff der FUs auf alle Registerzellen könnte durch ein Crossbar Matrix Switch implementiert werden. Als Eingänge dienen die Registerzellen und für jede FU würden 2 Ausgänge benötigt. Die Anzahl der Leitungen würde sich also durch die folgende Gleichung bestimmen: $LZ = RB + 2 * FUZ$. Allerdings ist der Verdrahtungsaufwand eines solchen Switches sehr hoch und zusätzlich müssen noch für jeden Ausgang Steuerleitungen den entsprechenden Operanden signalisieren. Damit der aktuelle Operand auch zu Beginn der Operation anliegt, muss das Steuersignal vorher schon entsprechend angepasst sein. Dies wäre durch Einführung eines Pipeline-Mechanismusses in der FU möglich. Wahrscheinlich ist der gesamte Ressourcenverbrauch eines solchen Switches und der dadurch notwendigen Anpassungen der VHM jedoch höher als bei der bisherigen Implementierung.

Der andere Weg besteht darin, einer FU nicht mehr den Zugriff auf alle Registerzellen zu erlauben. Der Compiler wird dadurch allerdings bei der Zuordnung der Operation zu den FUs eingeschränkt, da auf eine Registerzelle nur eine FU schreibend zugreifen kann. Damit dieser Weg eingeschlagen werden kann, ist zu untersuchen ob es eine Einschränkung des Zugriffes gibt, die alle Klassen von Schaltungen abbilden kann. Falls so eine Einschränkung existiert, muss abzuschätzen sein, ob und um wie viel sich die Abarbeitung verlängert, da es sehr wahrscheinlich ist, dass sich durch den geringeren Freiheitsgrad bei der Zuordnung der Operationen, die Anzahl der Instruktionsblöcke erhöht. Ein optimaler Kompromiss zwischen Ressourcenverbrauch und Verarbeitungsgeschwindigkeit(Anzahl der entstehenden Instruktionsblöcke) könnte somit gefunden wer-

[3]Registerbreite

den. Diese Auswahl kann auch von der Art der zu erwartenden Schaltungen abhängen. Eine Optimierung für den häufigsten Fall und schlechtere Ergebnisse für seltene Fälle, ergeben wahrscheinlich ein besseres Gesamtbild, als eine VHM die alle Arten von Schaltungen möglichst gleich schnell verarbeiten kann. Diese Überlegungen sind aber zu komplex und nicht Teil dieser Arbeit. Im Rahmen einer anderen Diplomarbeit, könnten aber diese Untersuchungen vorgenommen werden und eventuell eine bessere Möglichkeit der Verbindungsstruktur gefunden werden. Eine paar mögliche Zuordnungen werden im Folgenden angegeben.

Jede FU greift nur auf Registerzellen der Umgebung zu. Das bedeutet, FU Nr. x kann nur auf die Registerzellen r mit $x - i \leq r \leq x + j$ zugreifen. Eventuell kann dieser Zugriff auch zyklisch erfolgen. Wenn also $x - i < 0$ oder $x + j > RB$ eintritt, werden die Registerzellen am Anfang bzw. am Ende des Registers angesprochen. FU Nr. x hat demzufolge Zugriff auf die Zellen r mit $x - i \bmod RB \leq r \leq x + j \bmod RB$. Auf das Ergebnis von FU Nr. x können also nur benachbarte FUs zugreifen. Allerdings werden auch nur noch $LZ = FUZ * (i + j + 1)$-viele Leitung benötigt.

Ein andere Möglichkeit besteht darin, dass die FUs auf Registerzellen mit einem bestimmten Abstand untereinander zuzugreifen. Die Menge der erreichbaren Zellen ist in diesem Fall $R = \{r | r = x + z * i; i \in \mathbf{N}; z \in \mathbf{Z}\}$. Da hier x ein Teil der Menge ist, können auf das Ergebnis nur die FU Nr. x und die FUs mit einem Abstand von einen vielfachen von i zugreifen. Man kann eine Verschiebung y einbauen, wodurch $R = \{r | r = x + y + z * i; i \in \mathbf{N}; z, y \in \mathbf{Z}\}$ entsteht. Nun kann eine FU nicht mehr auf ihr Ergebnis der letzten Operationen zugreifen, aber die benachbarten FUs mit einem Abstand von y. Die Anzahl der benutzten Leitungen LZ errechnet sich in diesem Falle durch $LZ = RB/i$.

Die erste vorgeschlagene Zuordnung ist sicher sehr ungünstig, da alle Eingabewerte am Anfang des Registers stehen und auch die Ergebnisse laut Definition am Ende der Berechnung an den ersten Stellen der Register stehen. Dadurch werden die letzten FUs zu wenig ausgelastet und deren Ergebnisse müssen in mehreren Schritten an den Anfang des Registers gebracht werden. Im zweiten Fall gelingt dies besser, da auch FUs mit hohen Nummern auf die Eingabewerte zugreifen können. Die Ressourcenersparnis hängt jedoch stark von der Wahl des Parameters i ab. Ein kleiner Wert für i verringert die Verdrahtungskomplexität nur wenig, lässt aber eine relativ hohen Freiheitsgrad bei der Zuordnung der Operationen zu den FUs. Bei einem großen Wert für i ist es umgekehrt.

A Wichtige Zustandssignale

Bei der Erläuterung der Funktionsweise der VHM und ihrer Komponenten wurden einige Signale verwendet, deren Bedeutung in diesem Kapitel näher beschrieben wird. Es handelt sich um zur Designzeit festgelegte Signale aus dem Package constants(5.9) und interne Steuersignale zur Kommunikation und Synchronisation der Komponenten.

A.1 Der Befehlszähler(BZ)

Der Befehlszähler bestimmt den aktuell auszuführenden Befehl. Er ist Bestandteil der FUs. Erreicht er den Wert der IZ[1](A.6) wird er auf den Wert '0' zurückgesetzt und ein Schaltungstakt(A.11) wird ausgelöst.

A.2 Der Berechnungstakt(BT)

Der Berechnungstakt entspricht dem von außen angelegten Takt. Bei steigender Flanke wird in den FUs eine Berechnung ausgeführt und bei fallender Flanke werden die Ergebnisse in das Register übernommen.

A.3 Die Datenbusbreite(DB)

Dieses Signal beschreibt, wie viele Bits beim Lesen der Daten des Bytecodes von außen übertragen werden. Es wird im Package constants(5.9) zur Designzeit festgelegt. In dieser Version beträgt die Datenbusbreite 32 Bit.

A.4 Die Fehlercodes(FC)

Durch die Ausgabe eines Fehlercodes kann der Anwender sehen, ob und welches Problem beim Dekodieren des Bytecodes auftrat. In der Tabelle A.1 sind die verwendeten Codes dargestellt.

[1] Instruktionenanzahl

Code	Beschreibung
00_2	Abarbeitung erfolgreich
01_2	zu wenig Register vorhanden
10_2	zu viele Eingabewerte angegeben
11_2	zu viele Ausgabewerte angegeben

Tabelle A.1: Die Fehlercodes der VHM

A.5 Die FU-Adressbreite(FAB) und FU-Anzahl(FUZ)

Die Anzahl der vorhandenen FUs(5.3) wird durch die FU-Anzahl bestimmt. Diese berechnet sich durch $FUZ = 2^{FAB}$. Die FU-Adressbreite wird beim Design der VHM festgelegt und entspricht normalerweise der RAB(A.10). Sie darf jedoch nicht größer als Diese sein, da sonst kein Register zum Setzen des Ergebnisses vorhanden wäre.

A.6 Die Instruktionenanzahl(IZ)

Beim Beschreiben des Instruktionscache wird die Anzahl der Operationen mitgezählt, um bei der Berechnung den Schaltungstakt(A.11) auslösen zu können. Sie bestimmt, wie viele Instruktionen für die Berechnung der Schaltung ausgeführt werden.

A.7 Die Instruktionscachegrösse(ICG)

Von der ICG hängt ab, aus wie vielen Operationen der Bytecode bestehen darf, damit die VHM diesen ausführen kann. Eine FU kann maximal ICG-viele Operationen speichern. Da alle FUs dieselbe Anzahl von Instruktionen berechnen, darf die Tiefe des Bytecodes nicht größer als die Instruktionscachegrösse sein. Diese wird ebenfalls im Package constants(5.9) festgelegt.

A.8 Die Instruktionenlänge(IL)

Da die Länge einer Instruktion von der Länge der Operanden abhängt ist diese nicht konstant. Sie wird nach folgender Methode bestimmt: $IL = OL^2 + 2 * RAB^3$. Sie bestimmt den Aufbau des Instruktionscache.

[2] Opcodelänge
[3] Registeradressbreite

A.9 Opcodelänge(OL)

Zurzeit ist die Länge des Operationscodes auf 4 Bit festgelegt. Eine Veränderung ist nicht geplant, da hier auch der Bytecode erst angepasst werden müsste. Trotzdem ist dieses Signal im Package constants(5.9) definiert, um die VHM möglichen Änderungen anpassen zu können.

A.10 Die Registeradressbreite(RAB) und Registerbreite(RB)

Die Größe des Registers(5.6) hängt von der Registeradressbreite ab. Sie bestimmt die Anzahl der Bits, die nötig sind, um eine Registerzelle zu adressieren. Das heißt die Registerbreite bestimmt sich durch $RB = 2^{RAB}$. Ein Register besteht also aus RB-Bits.

A.11 Der Schaltungstakt(ST)

Wenn alle Operationen eines kompletten Durchlaufs des Algorithmus beendet sind, wird der Schaltungstakt ausgelöst. Geschieht dies werden die berechneten Ergebnisse an die Ausgangsports geleitet und die aktuellen Eingabewerte ins Register(5.6) übernommen.

B Struktur der Komponenten

Dieser Abschnitt zeigt die Struktur der einzelnen Komponenten. Es werden die im Design festgelegten Signale, deren Datentypen und der Ein/Ausgabe-Modus, sowie die entsprechende Komponente mit der interagiert wird, angegeben. Dazu werden noch einmal kurz die Funktionen der Signale beschrieben. Jedoch wurde beim Entwickeln der virtuellen Hardware-Maschine versucht durch eine entsprechende Namensgebung die Funktion der Signale zu verdeutlichen und die Wartung des VHDL-Codes zu vereinfachen. Außerdem erfolgt die genaue Beschreibung in den Kapiteln 5 und 6.

B.1 Decoder

Die Signale mit denen mit der Umgebung kommuniziert wird, dienen dem Einlesen des neuen Bytecodes sowie der Statussignalisierung. An den FU-Multiplexer werden die Instruktionen und an den Konstanten-Multiplexer die ausgelesenen Konstanten verteilt. Das Register erhält durch die Signale *reg_in_offset* und *reg_out_offset* die Anzahl der Ein- und Ausgabewerte. Die Verarbeitung der Daten unterbricht der Sequenzer, wenn das Signal *run* auf '1' gesetzt ist. Tabelle B.1 stellt beschriebenen Signale dar.

B.2 Functional Unit

Das Signal *clear_cache* des Decoders löst ein Leeren des Instruktionscaches aus. Die zu setzenden Instruktionen kommen vom FU-Multiplexer und die Daten werden vom Register geholt und in Dieses wieder geschrieben. Der Ablauf wird durch die Signale des Sequenzers gesteuert. In Tabelle B.2 sind diese Signale nochmal aufgelistet.

B.3 FU-Multiplexer

Die im Decoder erstellten Instruktionen werden an die entsprechenden FUs weitergeleitet. Das Signal *fu_clk* weist die FUs an, die angelegten Daten in ihren Instruktionscache zu übernehmen. Eine Darstellung der benutzten Signale findet man in Tabelle B.3.

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
update	std_logic	Eingabe	Umgebung
data	std_logic_vector(0 to 31)	Eingabe	Umgebung
instr	instruction	Ausgabe	FU-MUX
clk_instr	std_logic	Ausgabe	FU-MUX
akt_fu	choose_fu	Ausgabe	FU-MUX
nop	std_logic	Ausgabe	FU-MUX
clear_fu	std_logic	Ausgabe	FU
dec_error	std_logic_vector(1 downto 0)	Ausgabe	Umgebung
no_const	std_logic	Ausgabe	K-MUX
const_reg	vhm_reg	Ausgabe	K-MUX
const_position	operand	Ausgabe	K-MUX
reg_in_offset	reg_offset	Ausgabe	Register
reg_out_offset	reg_offset	Ausgabe	Register
synchro	std_logic	Ausgabe	Umgebung
run	std_logic	Ausgabe	Sequenzer

Tabelle B.1: Struktur des Decoders

B.4 Konstanten-Multiplexer

Durch die Signale des Decoders wird festgelegt, welche Konstanten vorhanden sind und wie sie mit den Eingabewerten verbunden werden. Diese werden dann an das Register(Signal *cm_output*) weitergegeben. Alle benutzten Signale sind in Tabelle B.4 aufgelistet.

B.5 Register

Im Register werden die Daten zwischengespeichert. Die Signale zu den FUs verarbeiten diese Daten. Wenn das Signal *restart* auf den Wert '1' gesetzt wird, werden die Eingaben des Konstanten-Multiplexers übernommen und der Umgebung die aktuellen Ergebnisse übergeben. Die Signale des Decoders bestimmen, welche Daten gesetzt werden. Tabelle B.5 zeigt alle vom Register benutzten Ein-/Ausgabesignale.

B.6 Schieberegister

Das Schieberegister ist ein Teil des Decoders(5.2), da nur dieser dessen Möglichkeiten nutzen muss. Alle Signale außer *bsh_clk_in* und *sh_ready* dienen der Datenübergabe.

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
in_reg	vhm_reg	Eingabe	Register
clk	std_logic	Eingabe	Sequenzer
instr	instruction	Eingabe	FU-MUX
instr_clk	std_logic	Eingabe	FU-MUX
clear_cache	std_logic	Eingabe	Decoder
reset	std_logic	Eingabe	Sequenzer
restart	std_logic	Ausgabe	Register
out_reg	std_logic	Ausgabe	Register

Tabelle B.2: Struktur der Functional Unit

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
instr_in	instruction	Eingabe	Decoder
fu_sel	choose_fu	Eingabe	Decoder
fu_nop	std_logic	Eingabe	Decoder
instr_out	complete_instruction	Ausgabe	FU
sel_out	std_logic_vector(0 to fu_count-1)	Ausgabe	FU
fu_clk	std_logic	Ausgabe	FU

Tabelle B.3: Struktur des FU-Multiplexers

Mit Hilfe dieser beiden Signale werden die Komponenten synchronisiert. Das Schieben beginnt erst nach einer Änderung des Signals *bsh_clk_in* und am Ende dieses Vorgang wird das Signal *sh_ready* verändert. Erst danach übernimmt der Decoder die Daten und setzt das Dekodieren fort. Eine Übersicht dieser Signale liefert Tabelle B.6.

B.7 Sequenzer

Der Sequenzer leitet das Signal *clk_in* als *clk_out* an die FUs und das Register weiter. Diese Funktionsweise wird nur unterbrochen wenn run auf '1' gesetzt ist. Bei einem Wechsel dieses Signals auf '1' wird außerdem mit *reset_fu* den FUs angezeigt, dass ein neuer Bytecode anliegt. Tabelle B.7 benennt nochmal diese Signale.

B.8 VHM

Die virtuelle Hardware-Maschine(VHM) dient als Rahmen für die einzelnen Komponenten. Sie beschreibt die Verbindungen der einzelnen Komponenten untereinander und stellt die Verbindung zur Umgebung dar. Deshalb wird in der folgenden Tabelle auch

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
cm_input	vhm_reg	Eingabe	Umgebung
cm_const	vhm_reg	Eingabe	Decoder
no_const	std_logic	Eingabe	Decoder
cm_const_pos	operand	Eingabe	Decoder
cm_output	vhm_reg	Ausgabe	Register

Tabelle B.4: Struktur des Konstanten-Multiplexer

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
reg_in	vhm_reg	Eingabe	FU
clk	std_logic	Eingabe	Sequenzer
ext_input	vhm_reg	Eingabe	K-MUX
restart	std_logic	Eingabe	FU
in_offset	in reg_offset	Eingabe	Decoder
out_offset	in reg_offset	Eingabe	Decoder
reg_out	vhm_reg	Ausgabe	FU
ext_output	vhm_reg	Ausgabe	Umgebung

Tabelle B.5: Struktur des Registers

jeweils die Komponente angegeben, die mit der Umgebung in Beziehung steht. Der E/A-Typ bezieht hierbei auf die VHM. Das bedeutet Eingabe steht für von außen angelegte Daten und Ausgabe entsprechend für nach außen gegebene Signale. Diese E/A-Signale werden in Tabelle B.8 noch einmal aufgelistet.

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
bsh_clk_in	std_logic	Eingabe	Decoder
SEL	std_logic_vector(5 downto 0)	Eingabe	Decoder
sel_mir	in std_logic_vector(5 downto 0)	Eingabe	Decoder
b_in	std_logic_vector(0 to 63)	Eingabe	Decoder
b_in_mir	in std_logic_vector(0 to 31)	Eingabe	Decoder
b_out	std_logic_vector(0 to 63)	Ausgabe	Decoder
b_out_mir	std_logic_vector(0 to 63)	Ausgabe	Decoder
sh_ready	std_logic)	Ausgabe	Decoder

Tabelle B.6: Struktur des Schieberegisters

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
clk_in	std_logic	Eingabe	Umgebung
run	std_logic	Eingabe	Decoder
reset_fu	std_logic	Ausgabe	FU
clk_out	std_logic	Ausgabe	Register & FU

Tabelle B.7: Struktur des Sequenzers

Signalname	Datentyp	E/A-Typ	Ziel/Herkunft
data_in	std_logic_vector(0 to 31)	Eingabe	Decoder
data_ready	std_logic	Ausgabe	Decoder
update_in	std_logic	Eingabe	Decoder
vhm_clk_in	std_logic	Eingabe	Sequenzer
error	std_logic_vector(1 downto 0)	Ausgabe	Decoder
input	vhm_reg	Eingabe	K-MUX
output	vhm_reg	Ausgabe	Register

Tabelle B.8: Struktur der virtuellen Hardware-Maschine

C Inhalt der CD

Die VHDL-Dateien in denen die Komponenten beschrieben worden sind auf einer CD gespeichert. Im Verzeichnis VHDL/Aktuell liegen die zuletzt verwendeten und in dieser Arbeit beschriebenen Versionen. Unter VHDL/Versuche sind einige Versionen der Komponenten zu finden die nicht verwendet werden. Sie sind trotzdem auf der CD enthalten, da hier andere Implementationsmöglichkeiten getestet wurden. Unter anderem sind hier die verschiedenen Versionen des Schieberegisters(↗7.1.2) abgelegt. Die verwendeten Beispiele befinden sich im Verzeichnis VHDL/Beispiele.

Decoder: decoder_v9.vhd

Functional Unit: fu.vhd

Functional Unit Nr. 0: fu_0.vhd

FU-Multiplexer: fu_mux.vhd

Konstanten-Multiplexer: const_mux.vhd

Package constants: constants.vhd

Register: registerfile.vhd

Sequenzer: sequenzer.vhd

Schieberegister: bsh_t3.vhd

VHM: vhm_v9.vhd

Die Diplomarbeit - als pdf- und ps-Datei - befindet sich im Hauptverzeichnis der CD.

D Codebeispiele

Um die beschriebenen Funktionsweisen besser nachvollziehen zu können sind einige Teile des VHDL-Codes am Ende dieser Arbeit angefügt. Eine grobe Beschreibung der wichtigsten VDHL-Befehle befindet sich im Abschnitt 3.2 ab Seite 18.

D.1 Auslesen der Operanden im Decoder

```
case operand_pos is
when 0 => --den Opcode auswaehlen
  opcode_test:=data_load(0 to opcode_length-1);
  if opcode_test="0000" then --Ende des Blocks signalisiert
    last_read_eob<='1'; --als letztes EOB gelesen
    if last_read_eob='1' then
      --2mal EOB gelesen -> ENDE des Bytecodes
      fetch<=5;
    elsif instr_pos=0 then
      --letzte Instruktion des Blocks-> keine NOP's einfuegen
      sh_sel<=opcode_binary;
      clk_instr<='0'; --keine Instruktion setzen
      nop<='0'; --keine NOP's einfuegen
    else
      sh_sel<=opcode_binary;
      nop<='1'; --NOP's einfuegen
      clk_instr<='1'; --Instruktionen setzen
      instr_pos<=0;
    end if;
  else --kein EOB gelesen
    last_read_eob<='0';
    nop<='0'; --keine NOP's einfuegen
    instr.opcode<=opcode_test; --opcode setzen
    operand_pos:=1;
    sh_sel<=opcode_binary;
    clk_instr<='0'; --keine Instruktion setzen
  end if;
  when 1 => --1. Operand auslesen
```

```
    --Operand mit Maske verknuepfen
    instr.input1<=data_load(0 to reg_address-1) and mask;
    sh_sel<=operand_shift;
    operand_pos:=2;
    clk_instr<='0'; --keine Instruktion setzen
  when others => --2. Operand auslesen
    --Operand mit Maske verknuepfen
    instr.input2<=data_load(0 to reg_address-1) and mask;
    sh_sel<=operand_shift;
    operand_pos:=0;
    akt_fu<=instr_pos;
    --wenn letzte Instruktion des Blocks
    if instr_pos=fu_count-1
    then --Beginn von vorne
      instr_pos:=0;
    else --naechste FU ansprechen
      instr_pos:=instr_pos+1;
    end if;
    clk_instr<='1';
  end case;
  --schieben und evtl nachladen signalisieren
  load_clk<=not load_clk;
```

D.2 Erstellen der Operandenmaske im Decoder

```
--setzt die Maske fuer die Operanden, um diese an die Instruktion anzupassen
procedure set_mask(signal adr: in std_logic_vector(31 downto 0); variable m: out operand;
                   variable op_sh: out std_logic_vector(5 downto 0)) is
variable i : natural;
variable temp_mask : unsigned(0 to reg_address-1);
begin
  i:=to_integer(unsigned(adr)); --die Breite der Adressierung bestimmen
  temp_mask:=(others => '1');
  --die Maske bestimmen '1..10..0'
  temp_mask:=shift_left(temp_mask,reg_address-i);
  m:=std_logic_vector(temp_mask);
  op_sh:=adr(5 downto 0); --die Adressbreite merken
end procedure set_mask;
```

D.3 Verteilen der Instruktionen und Umdrehen der Operanden

```
--dieser Prozess setzt eine Instruktion
set_instr : process(fu_clk,fu_sel,fu_nop,instr_in)
-- bestimmt die Ziel-FUs
```

```
variable selected : std_logic_vector(0 to fu_count-1);
begin
  --latches vermindern
  for i in 0 to fu_count-1
  loop
    instr_out(i).opcode<=(others => '0');
    instr_out(i).input1<=(others => '0');
    instr_out(i).input2<=(others => '0');
  end loop;
  sel_out<=(others =>'0');
  --latches vermindert
  if fu_clk='1' then --wenn verteilt werden soll
    selected:=(others =>'0');
    if fu_nop='1' then --restliche FUs mit MOV setzen
      sel : for i in fu_count-1 downto 0
      loop
        if i=fu_sel then exit sel; end if; --schleife verlassen
        selected(i):='1'; -- Ziel-FU setzen
        instr_out(i).opcode<="0100"; --MOV Operation
        instr_out(i).input1<=std_logic_vector(to_unsigned(i,operand'length));
      end loop sel;
    else --nur eine Instruktion setzen
      selected(fu_sel):='1'; -- Ziel-FU setzen
      instr_out(fu_sel).opcode<=instr_in.opcode;
      --Adressen umdrehen
      for i in operand'range loop
        instr_out(fu_sel).input2(i)<=instr_in.input2(instr_in.input2'right-i);
        instr_out(fu_sel).input1(i)<=instr_in.input1(instr_in.input1'right-i);
      end loop;
    end if;
    sel_out<=selected; --Ziel(e) setzen
  end if;
end process set_instr;
```

D.4 Schieben der Datenströme im Schiebregister

```
--schiebt die Daten um 0 bis 3 Stellen nach links
shift_1 : process(bsh_clk_in,sel,s,b_in,sel_mir,b_in_mir)
begin
  if bsh_clk_in'event then --schieben starten
    if sel(0)/='U'then -- Schiebeposition angegeben
      case SEL(1 downto 0) is
        when "00" =>   -- um 0 Stellen schieben
          c <= b_in;
        when "01" =>   -- um 1 Stelle 1 schieben
```

```
          c(0 to 62) <= b_in(1 to 63);
        when "10" =>   -- um 2 Stellen schieben
          c(0 to 61) <= b_in(2 to 63);
        when others =>  -- um 3 Stellen schieben
          c(0 to 60) <= b_in(3 to 63);
      end case;
      c_mir<=(others=>'0');
      case SEL_mir(1 downto 0) is
        when "00" =>   -- um 0 Stellen schieben
          c_mir(32 to 63) <= b_in_mir;
        when "01" =>   -- um 1 Stelle schieben
          c_mir(31 to 62) <= b_in_mir;
        when "10" =>   -- um 2 Stellen schieben
          c_mir(30 to 61) <= b_in_mir;
        when others => -- um 3 Stellen schieben
          c_mir(29 to 60) <= b_in_mir;
      end case;
      if s='U' then s<='1'; else s<=not s; end if;
    end if;
  end if;
end process;

--schiebt um weitere 0 bis 12 Stellen um links
shift_2 : process(s)
begin
  if s'event then
    d<=(others=>'0');
    d_mir<=(others=>'0');
    case SEL(3 downto 2) is
      when "00" =>   -- um weitere 0 Stellen schieben
        d<=c;
      when "01" =>   -- um weitere 4 Stellen schieben
        d(0 to 59) <= c(4 to 63);
      when "10" =>   -- um weitere 8 Stellen schieben
        d(0 to 55) <= c(8 to 63);
      when others => -- um weitere 12 Stellen schieben
        d(0 to 51) <= c(12 to 63);
    end case;
    case SEL_mir(3 downto 2) is
      when "00" =>   -- um weitere 0 Stellen schieben
        d_mir(29 to 63) <= c_mir;
      when "01" =>   -- um weitere 4 Stellen schieben
        d_mir(25 to 59) <= c_mir(29 to 63);
      when "10" =>   -- um weitere 8 Stellen schieben
        d_mir(21 to 55) <= c_mir(29 to 63);
```

```
      when others => -- um weitere 12 Stellen schieben
        d_mir(17 to 51) <= c_mir(29 to 63);
    end case;
    s2<=s;
  end if;
end process;

--schiebt nochmal um weitere 0 bis 32 Stellen um Links
shift_3 : process(s2)
begin
  if s2'event then
    b_out<=(others=>'0');
    b_out_mir<=(others=>'0');
    case SEL(5 downto 4) is
      when "00" =>   -- um weitere 0 Stellen schieben
        b_out <= d;
      when "01" =>   -- um weitere 16 Stellen schieben
        b_out(0 to 47) <= d(16 to 63);
      when "10" =>   -- um weitere 32 Stellen schieben
        b_out(0 to 31) <= d(32 to 63);
      when others => -- um weitere 48 Stellen schieben
        b_out(0 to 15) <= d(48 to 63);
    end case;
    case SEL_mir(5 downto 4) is
      when "00" =>   -- um weitere 0 Stellen schieben
        b_out_mir(17 to 63) <= d_mir;
      when "01" =>   -- um weitere 16 Stellen schieben
        b_out_mir(1 to 47) <= d_mir(17 to 63);
      when others => -- um weitere 32 Stellen schieben
        b_out_mir(0 to 31) <= d_mir(32 to 63);
    end case;
    sh_ready<=s2; --Vorgangsende signalisieren
  end if;
end process;
```

D.5 Übernehmen der Konstanten im Konstanten-Multiplexer

```
-- vereint die Konstanten mit den Eingaben
multiplex : process(cm_input,cm_const,cm_const_pos,no_const)
variable j : reg_offset;
begin
  -- wenn keine Konstanten vorhanden sind
  if no_const='1' then
    cm_output<=cm_input; --Eingaben uebernehmen
  else
```

```
    --Position bestimmen
    j:=to_integer(unsigned(cm_const_pos));
    for i in cm_input'range
    loop
      if i<j --bis zur Position
      then
        cm_output(i)<=cm_input(i); --Eingaben uebernehmen
      else
        cm_output(i)<=cm_const(i-j); --danach die Konstanten setzen
      end if;
    end loop;
  end if;
end process;
```

D.6 Speichern der Ergebnisse im Register

```
-- uebernimmt die angelegten Werte
update : process(clk,restart)
begin
if restart='1'
then
  ext_output<=(others=>'0');  --die Ausgaben initialisieren
  for i in 0 to reg_count-1 loop
    if i<in_offset
    then
      reg_out(i)<=ext_input(i); -- die Eingaben werden uebernommen
    end if;
    if i<out_offset
    then
      ext_output(i)<=reg_in(i); -- die Ausgaben uebernehmen
    end if;
  end loop;
  -- bei fallender Flanke werden die Zwischenergebnisse gesetzt
  elsif   clk'event and clk='0'
  then
    reg_out<=reg_in; --die Zwischenergebnisse uebernehmen
  end if;
end process update;
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Ash02] Peter J. Ashenden. *The Designers Guide to VHDL*. Morgan Kaufmann, 2nd edition, 2002.

[Com02] Aldec The Design Verification Company. Active HDL On-Line Documentation. *Aldec Homepage*, 2002. www.aldec.com.

[Div03a] Mentor Emulation Division. CelaroPro Hardware Emulator datasheet. *Mentor Graphics Corporation*, 2003. http://www.mentor.com/celaro/documentation/celaropro_ds.pdf.

[Div03b] Mentor Emulation Division. Mentor graphics vstation. *Mentor Graphics Corporation*, 2003. http://www.mentor.com/vstation.

[Dub96] Michel Dubois. Rpm project homepage. *University of Southern California*, 1996. http://www.usc.edu/dept/ceng/dubois/RPM.html.

[FP98] William Fornaciari and Vincenzo Piuri. Virtual FPGAs: Some steps behind thy physical barriers. *Department of Electronics and Information, Politecnico di Milano*, 1998. ipdps.eece.unm.edu/1998/raw/fornacia.pdf.

[Haa01] Andreas Haase. Untersuchungen zur dynamischen Rekonfiguration von FPGAs. Master's thesis, Technische Universität Chemnitz, September 2001. http://www-user.tu-chemnitz.de/~anhaa/Diplom/Diplom.html.

[Lan02] Sebastian Lange. Design and implementation of a virtual hardware machine. Master's thesis, Universität Leipzig, November 2002.

[Lip02] Mikko H. Lipasti. Parallel processors. *University of Wisconsin-Madison*, 2002. http://www.cae.wisc.edu/~mikko/552/ch9.ppt.

[LLFP] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier. Placing, routing and editing virtual FPGAs. *Université de Bretagne Occidentale - Brest, IRISA / CNRS - Rennes.* www.irisa.fr/symbiose/people/lavenier/Publications/Lav01ch.pdf.

[SUA+00] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Moto-mura. A virtual hardware system on a dynamically reconfigurable logic device. *Proceedings of IEEE symposium on FPGAs for custom computing machines*, 2000.

[Xil00] Inc Xilinx. Virtex series configuraton architecture user guide v 1.5. *Xilinx*, September 2000. http://www.xilinx.com/xapp/xapp151.pdf.

[Xil01] Inc. Xilinx. Virtex datasheet. *Product Specification*, 2001. http://direct.xilinx.com/bvdocs/publications/ds003.pdf.

[Xil02] Inc. Xilinx. Xilinx product information, 2002. http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, den 4.6.2003

Nick Bierwisch