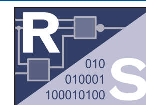
Unterstützung von Fließkommaarithmetik in CGRAs

Bachelorarbeit

Lars Stein

21. August 2017

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Erklärung gemäß § 22 Abs. 7 APB

Hiermit erkläre ich gemäß § 22 Abs. 7 der Allgemeinen Prüfungsbestimmungen (APB) der Technischen Universität Darmstadt in der Fassung der 5. Novelle vom 18. Mai 2016, dass ich die Arbeit selbstständig verfasst und alle genutzten Quellen angegeben habe und bestätige die Übereinstimmung von schriftlicher und elektronischer Fassung.

Darmstadt, den 21. August 2017
Ort, Datum

Name

Fachbereich Elektro- und Informationstechnik
Institut für Datentechnik
Fachgebiet Rechnersysteme
Prüfer: Prof. Dr.-Ing. Christian Hochberger
Betreuer: M.Sc. Dennis Wolf

Inhaltsverzeichnis

1 Aufgabenstellung 3

2 Grundlagen 4
- 2.1 AMIDAR 4
- 2.2 UltraSynth 5
- 2.3 Coarse Grained Reconfigurable Array 5
  - 2.3.1 Aufbau CGRA 6
  - 2.3.2 Aufbau PE 6
  - 2.3.3 CGRA-Generator 8
- 2.4 Fließkommazahlen 8
- 2.5 IEEE 754-Format 9
  - 2.5.1 Zahlendarstellung 9
  - 2.5.2 Sonderfälle 10
  - 2.5.3 Runden 11

3 Operatoren 12
- 3.1 Multiplikation 12
- 3.2 Addition & Subtraktion 12
- 3.3 Division 13

4 Implementierung 15
- 4.1 Vorgehen 15
- 4.2 Parametrisierung 15
- 4.3 Multiplikation 17
- 4.4 Addition & Subtraktion 20
- 4.5 Division 22

5 Evaluation 26

6 Zusammenfassung 29

1 Aufgabenstellung

Im Rahmen dieser Bachelorarbeit sollen die vier Fließkommaoperationen Addition, Subtraktion, Multiplikation und Division für das CGRA mittels Hardware-Beschreibungssprache implementiert werden. Diese Implementierung soll in einen Verilog-Generator integriert werden, der die Verilog Beschreibung für jede beliebige CGRA-Komposition erstellen kann. Die implementierten Operationen sollen Operanden beliebiger Bitbreite entgegegennehmen können. Außerdem soll die Möglichkeit bestehen, auch Operanden entgegennehmen zu können, die breiter sind als die Eingangsbitbreite der ALU.

Konkret sollte zunächste für jede Operation eine Implementierung fester Bitbreite erstellt und getestet werden. Für die Division stand hierbei bereits eine Goldschmidt-Implementierung zur Verfügung. Diese Implementierungen sollten in einem nächsten Schritt auf variable Bitbreiten angepasst und in den Verilog-Generator integriert werden. Dabei sind beide Operanden sowie das Ergebnis vom gleichen Format. Dabei mussten allerdings keine Optimierungen in Bezug auf Laufzeit oder Ressourcenverbrauch getätigt werden.

Abschließend sollten die einzelnen Operationen mit verschiedenen Operandengrößen und unterschiedlichen Exponent/Mantissen Kombinationen getestet werden. Ebenfalls sollten die Operationen sowohl einzeln, als auch integriert im CGRA, evaluiert werden.

2 Grundlagen

In diesem Kapitel werden alle nötigen Grundlagen erläutert und vorgestellt, welche für die dann folgende Ausarbeitung notwendig sind. Beginnend mit den Prozessoren, die das *Coarse Grained Reconfigurabel Array* (CGRA) später verwenden werden, gefolgt von einem Überblick über das CGRA selbst. Abschließend wird die, für die gesamte Bearbeitung essentielle, IEEE 754-Norm vorgestellt.

2.1 AMIDAR

Der Name AMIDAR ist eine Abkürzung für *Adaptive Microinstruction Driven Architecture*. Der Aufbau eines AMIDAR-Prozessors unterscheidet sich grundlegend vom Aufbau üblicher Prozessoren. Er besteht aus mehreren *Functional Units* (FUs), einer *Token Machine*, einem *Token Distribution Network* (TDN) und einem Datenbus.

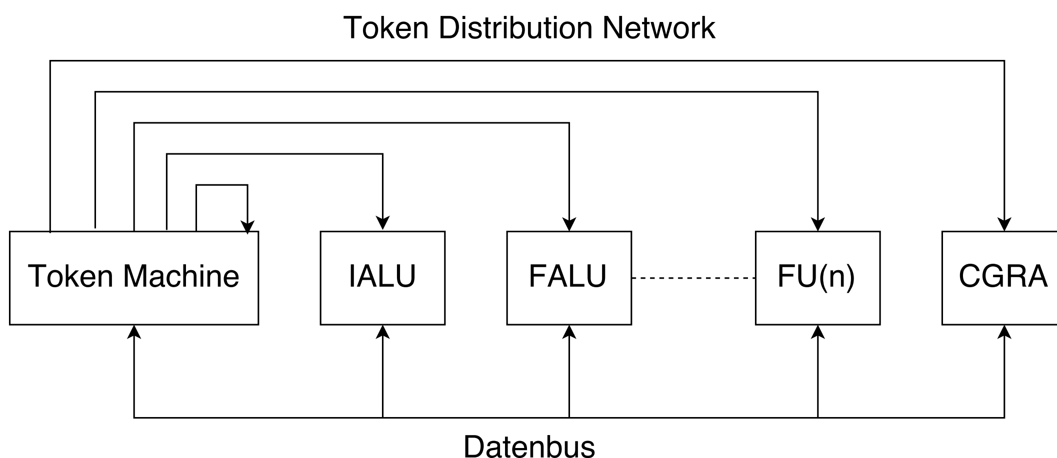


Abbildung 2.1: Aufbau des AMIDAR-Prozessors

Jede FU ist im Prinzip ein eigenständiges System, welches unterschiedliche Aufgaben realisieren kann. Vor allem werden die *Arithmetic Logic Units*(ALUs) für Interger-Zahlen (IALU) oder Floatingpointzahlen (FALU) als eigenständige FUs eingebunden. Beispielsweise werden spezielle FUs als Integer-ALUs oder Floatingpoint-ALUs eingebunden.

Die *Token Machine* wandelt jede einzelne Instruktion, im Falle von AMIDAR handelt es sich um Java Byte-Code Instruktionen, in ein oder mehrere Token um. Diese Token entsprechen im Grunde Mikroinstruktionen für einzelne FUs. Über das TDN werden sie automatisch an die FUs gesendet, welche die benötigten Operationen zur Verfügung stellen. Die Token werden dort, sobald die Eingangsdaten vorliegen, abgearbeitet.

Wenn verschiedene Instruktionen nicht voneinander abhängig sind, erlaubt es das AMIDAR-Konzept, diese Instruktionen automatisch und ohne viel Aufwand auf den entsprechenden FUs parallel auszuführen. Dafür sind auch keine näheren Angaben zur Laufzeit nötig, da die FUs

nur bei gültigen Eingangsdaten anfangen zu rechnen und andernfalls warten. Dies steigert die Effizienz im Vergleich zu herkömmlichen Prozessoren.

Als zusätzliche FU kann das CGRA in das System eingebunden werden. Es dient dort als Hardware-Beschleuniger und kommt unter bestimmten Bedingungen zum Einsatz. Während der Programmausführung erkennt ein Profiler, der in AMIDAR als Hardware-Einheit integriert ist, Schleifen beziehungsweise Codesequenzen, die sehr häufig ausgeführt werden. Sofern möglich, wird dann das CGRA entsprechend konfiguriert und die Codesequenz darauf ausgeführt.

Der AMIDAR-Prozessor existiert momentan nicht als physisches Hardwareprodukt. Er ist mittels Hardware-Beschreibungssprache beschrieben und wird mittels Synthese auf einem Field Programmable Gate Array (FPGA) abgebildet. [Ami]

2.2 UltraSynth

UltraSynth ist ein weiterer Prozessor, der das CGRA als Beschleuniger einsetzt. [Ult] Er wird am Fachgebiet für „Eingebette Systeme und ihre Anwendung“ an der TU-Darmstadt entwickelt. Die Besonderheit des UltraSynth Prozessors ist, dass er mit unterschiedlichen Bitbreiten rechnen kann. Diese Tatsache ist seinem Anwendungszweck in Messsystemen geschuldet. Dort sind nicht immer größtmögliche Genauigkeiten gefordert oder es werden generell weniger Bits benötigt um Messwerte zu speichern. Aus diesem Grund muss auch das CGRA seine Operationen in beliebigen Bitbreiten anbieten können. Das betrifft auch die im Rahmen dieser Bachelorarbeit implementierten Fließkommaoperationen.

2.3 Coarse Grained Reconfigurable Array

Ein CGRA ist eine Mikroarchitektur, die als Hardware-Beschleuniger verwendet wird. Ihr Hauptvorteil, entgegen anderen Architekturen, ist die parallele Ausführung mehrerer Instruktionen. Für gewöhnlich sind dies Instruktionen eines Schleifenkörpers.

Im Vergleich zu anderen Mikroarchitekturen von *Central processing units* (CPUs) und *Graphics processing units* (GPUs) im Hinblick auf Performance und Vielseitigkeit ist das CGRA dazwischen anzusiedeln. Es ist performanter als eine herkömmliche CPU; dafür weniger vielseitig. Auf der anderen Seite wird die Leistungsfähigkeit einer GPU nicht erreicht; dafür wird weniger Energie verbraucht.

Da das CGRA darauf ausgelegt ist, den Inhalt von Schleifen zu beschleunigen, kann es einen Prozessor nicht ersetzen. Dies ist zwar theoretisch möglich, aber sehr ineffizient. Daher wird es für gewöhnlich in einen Prozessor eingebunden, im Rahmen dieser Bachelorarbeit ist es der AMIDAR-Prozessor.

Der Prozessor arbeitet das Programm normal ab. Bei Ausführung einer Schleife wird versucht, diese auf dem CGRA abzubilden. Ob dies möglich ist, ist unter anderem davon abhängig, welche Operationen das CGRA zur Verfügung stellt. Für die Dauer der Ausführung übernimmt das CGRA die Kontrolle über den Programmfluss und der Prozessor bleibt inaktiv.

Zur Beschleunigung von Codesequenzen muss nicht grundsätzlich ein CGRA herangezogen werden. Theoretisch ist auch eine Beschleunigung durch ein FPGA denkbar.
Der zu beschleunigende Code muss hierfür in eine Hardware-Beschreibung umgewandelt wer-

den und mittels Synthesetool auf einem FPGA abgebildet werden. Die Synthese dieser FPGA-Konfiguration dauert in der Regel allerdings mehrere Minuten bis Stunden. Daher ist diese Lösung zur Laufzeit von Programmen nicht praktikabel. [TRH]

Hier liegt der Vorteil des CGRAs. Es lässt sich mit wenigen Bits neu konfigurieren. In jedem Takt kann eine neue Konfiguration geladen werden. Dieser geringe Aufwand macht es überhaupt erst möglich, dass der AMIDAR-Prozessor Schleifen zur Laufzeit erkennt und beschleunigt.

2.3.1 Aufbau CGRA

Ein CGRA besteht aus einer bestimmten Anzahl sogenannter Processing Elements (PEs). Diese sind in einem Array angeordnet und können unterschiedliche Operationen zur Verfügung stellen. Mittels eines vermaschten Netzwerkes, auch Interconnect genannt, sind sie miteinander verbunden. Alle möglichen Verbindungsarten sind denkbar. Man unterscheidet hierbei zwischen einem regulären und einem irregulären Aufbau. Abb. 2.2 zeigt beispielhaft ein CGRA, das „von Neumann“ verbunden ist, das heißt jedes PE ist mit all seinen direkten Nachbarn verbunden.

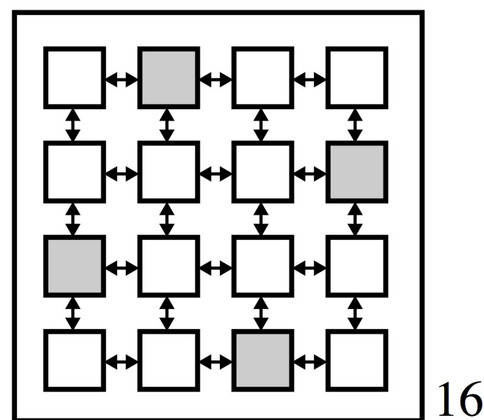


Abbildung 2.2: CGRA mit 16 PEs und „von Neumann“ Interconnect

Die einzelnen PEs müssen darüber hinaus nicht alle die gleichen Funktionen bereitstellen. Hier wird zwischen einem homogenen und einem heterogen Satz PEs unterschieden. In Abb. 2.2 haben beispielsweise nur die grau eingezeichneten PEs Speicherzugriff.

Jede mögliche Kombination von PEs und Interconnect beschreibt das CGRA fast vollständig und wird Komposition genannt. [TRH]

2.3.2 Aufbau PE

Wie bereits erwähnt, kann jedes einzelne PE einen unterschiedlichen Satz an Operationen bereitstellen. Dies kann im Prinzip jede arithmetische oder logische Operation sein. Auch Typkonvertierungen und Speicheroperationen sind möglich.

Ein PE besteht aus einer ALU, einem Registerfile(RF) und mehreren Multiplexern. In Abb. 2.3 ist ein einzelnes PE dargestellt, in dem alle konfigurierbaren Elemente eingefärbt sind.

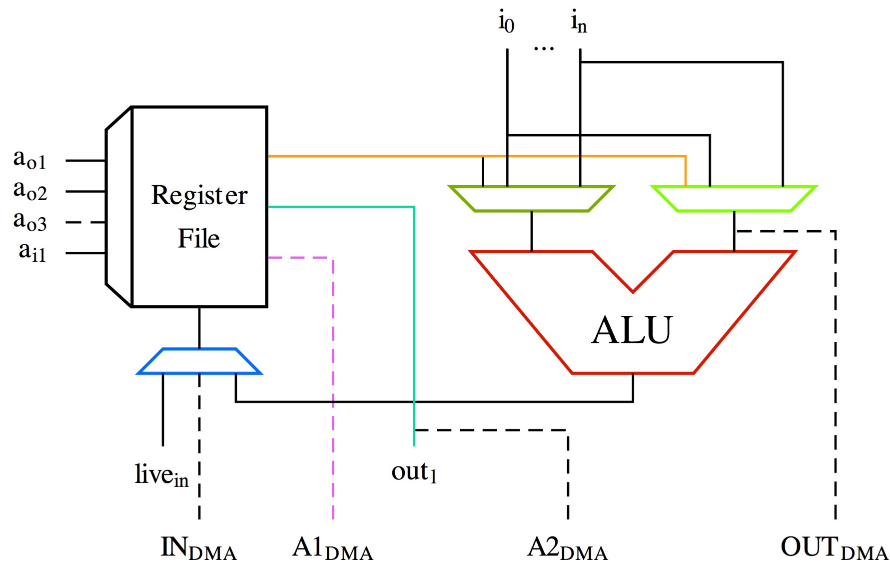


Abbildung 2.3: Aufbau eines PEs

Wie in Abb. 2.3 ersichtlich, benötigen die einzelnen Elemente nur sehr wenige Bits zur Konfiguration. Eine solche Konfiguration wird Kontext genannt. Jedes PE besitzt einen sogenannten Kontext-Memory in dem alle Kontexte für die aktuelle Programmausführung gespeichert sind. In jedem Takt kann daraus ein neuer Kontext geladen werden und das PE folglich in jedem Takt eine andere Aufgabe ausführen.

Zu beachten ist, dass PEs mit Speicherzugriff zusätzliche Elemente benötigen. Deren Aufbau ist für das Verständnis dieser Arbeit jedoch nicht essentiell.

Die ALU der PEs besteht aus mehreren eigenständigen Modulen. Für jede einzelne Operation gibt es ein Modul. Die Operationen können in drei Typen eingeteilt werden. Zum einen in arithmetische Opertionen mit einem oder zwei Eingängen, die ihr Ergebnis über einen Multiplexer an das RF zurückgeben. Zum anderen in Kontrollflussoperationen, wie zum Beispiel „kleiner gleich", die ihr Ergebnis direkt an eine Condition-Box zurückgeben. Diese sitzt zentral für alle PEs außerhalb im CGRA und steuert den Programmfluss. [TRH] Speicheroperationen bilden den letzten Typ. Abb. 2.4 zeigt eine exemplarische ALU mit allen drei unterschiedlichen Typen.

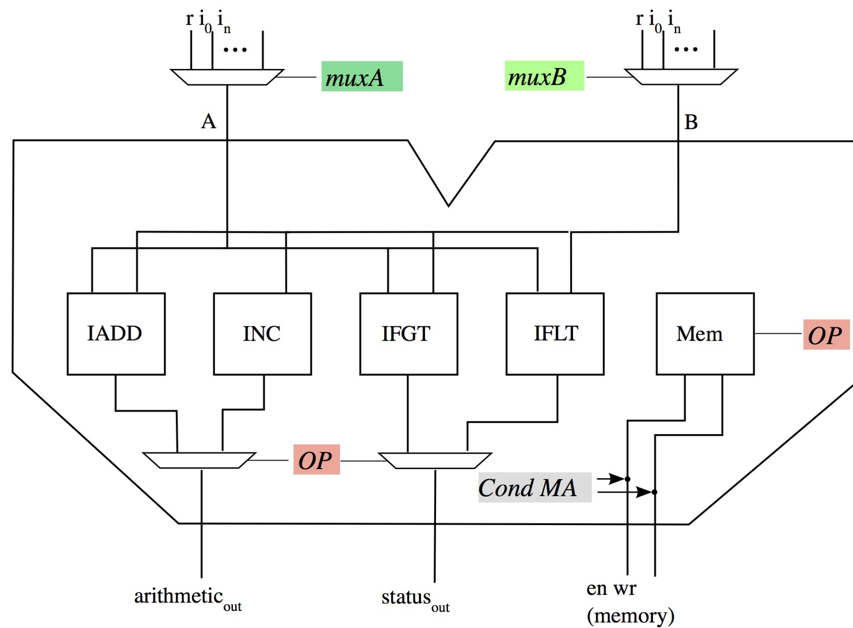


Abbildung 2.4: Exemplarische ALU eines PEs

2.3.3 CGRA-Generator

Da das CGRA selbst nicht als eigenständige Hardware existiert, wird es mittels Verilog Beschreibung und Synthesetools auf einem FPGA abgebildet. Um nicht für jede neue CGRA-Komposition einzelne Verilog Beschreibungen anpassen zu müssen, wurde eine Generator entwickelt, der diese Aufgabe übernimmt.

Das CGRA wird hierfür, mittels weniger JSON-Dateien mit Angaben zur PE-Anzahl, dem Interconnect und ihren Operationen, vollständig beschrieben. Der in Java programmierte Generator liest diese JSON-Dateien ein und erzeugt eine Verilog Beschreibung für jeden einzelnen Bestandteil des CGRAs.

Durch diese einfache Generierung ist es auch möglich, ohne viel Aufwand, unterschiedliche CGRA-Compositionen für eine Anwendung zu testen. So kann effizient getestet werden wie sich die unterschiedlichen PEs, ihre Vernetzung und nicht zuletzt die Größe des CGRAs auf die mögliche Beschleunigung auswirken.

2.4 Fließkommazahlen

Um rationale Zahlen darzustellen, gibt es unterschiedliche Möglichkeiten. Eine ist die sogenannte Fließkommadarstellung. Hierbei ist die Position des Kommas variabel und wird gesondert angegeben. Die allgemeine Darstellung einer Fließkommazahl ist in Gleichung (2.1) gezeigt.

$$x = m * b^e \tag{2.1}$$

m wird als Mantisse, b als Basis des Zahlensystems und e als Exponent bezeichnet. Der Exponent gibt im Prinzip an, um wie viele Stellen das Komma innerhalb der Mantisse nach links oder

rechts verschoben wird. Gleichung (2.2) zeigt ein Beispiel für eine dezimale Fließkommazahl, Gleichung (2.3) für eine binäre Fließkommazahl.

$$x = 1262,5_{10} * 10^{-2} \tag{2.2}$$

$$x = 11,00101_2 * 2^2 \tag{2.3}$$

In den oben gezeigten Beispielen ist die Position des Kommas innerhalb der Mantisse noch beliebig. Für die spätere Anwendung muss sie innerhalb der Mantisse aber an der selben Position sein. Um das zu erreichen, werden die Fließkommazahlen normalisiert. Das bedeutet im Allgemeinen, dass für die Mantisse die Bedingung $1 \leq m < b$ erfüllt sein muss. Gleichung (2.4) und Gleichung (2.5) zeigen die vorangegangenen Beispiele in normalisierter Form.

$$x = 1,2625_{10} * 10^1 \tag{2.4}$$

$$x = 1,100101_2 * 2^3 \tag{2.5}$$

2.5 IEEE 754-Format

IEEE 754 ist eine 1985 vom *Institute of Electrical and Electronics Engineers* eingeführte Norm zur Darstellung und Berechnung von Fließkommazahlen, beziehungsweise Floatingpoint Zahlen, in Digitalsystemen. Sie gibt außerdem an wie verschiedene mathematische Operationen durchgeführt und Zahlen gerundet werden. Ziel war es, die auf jedem System unterschiedlichen Darstellungen und Behandlungen von rationalen Zahlen durch einen gemeinsamen Standard abzulösen, um auf verschiedenen Digitalsystemen bei identischen mathematischen Operationen die gleichen Ergebnisse zu erhalten.

2.5.1 Zahlendarstellung

Im Wesentlichen legt die Norm die Aufteilung der binären Zahl in Vorzeichen, Exponent und Mantisse fest. [iee] Beispielhaft ist dies in Abbildung Abb. 2.5 und Abb. 2.6 für 32-bit und 64-bit breite Zahlen vorgestellt. Diese Darstellungen heißen in der Norm *single-* beziehungsweise *double-precision*. Im Java Standard entsprechen sie den Datenformaten *float* und *double*. Im Jahr 2008 wurden weitere Darstellungen in die Norm aufgenommen. Der grundsätzliche Aufbau ist gleich, lediglich die Breite der Zahl und entsprechend die Anzahl der Bits für die Teilbitgruppen sind verschieden.

VZ	Exponent (8 Bit)	Mantisse (23Bit)
MSB		LSB

Abbildung 2.5: Schema der Bitgruppen von float-Zahlen

VZ	Exponent (11 Bit)	Mantisse (52 Bit)
MSB		LSB

Abbildung 2.6: Schema der Bitgruppen von double-Zahlen

Generell ist das höchstwertige Bit immer das Vorzeichenbit (VZB). Eine Null bezeichnet eine positive, eine Eins eine negative Zahl.

Der Exponent wird als vorzeichenlose Binärzahl gespeichert. Um auch negative Exponenten eindeutig auf den vorzeichenlosen Zahlenbereich abzubilden, wird auf den tatsächlichen Exponenten der sogenannte Bias-Wert addiert. Der resultierende Wert, im folgenden *verschobener Exponent* genannt, wird in der Floatingpoint-Darstellung gespeichert.

Zu beachten ist, dass sowohl der maximal als auch der minimal verschobene Exponent zur Kodierung von Sonderfällen (Abschnitt 2.5.2) genutzt wird.

Der in der Mantisse gespeicherte Wert kann auf zwei Arten interpretiert werden. Sofern möglich, wird eine Floatingpoint Zahl in ihrer normalisierten Form gespeichert. Da hierbei an erster Stelle immer eine Eins steht, muss diese nicht explizit gespeichert werden und man gewinnt ein Bit zusätzliche Genauigkeit. Man nennt diese Stelle *hidden-bit*. In diesem Fall wird gemäß der Norm nur der Teil hinter dem Komma in der Mantisse gespeichert.

Aus den einzelnen Teilen der IEEE 754 Darstellung lässt sich dann gemäß der Gleichung (2.6) die binäre Fließkommazahl berechnen.

$$x = (-1)^{(VZB)} * 1,[Mantisse]_2 * 2^{versch.Exponent-Bias} \tag{2.6}$$

Da durch die normalisierte Darstellung Werte in der näheren Umgebung um Null nicht dargestellt werden können, ist es nötig, auch denormaliserte Zahlen darstellen zu können. Eine denormalisierte Zahl wir mit einem auf Null gesetzten Exponenten angezeigt. In diesem Fall interpretiert man das *hidden-bit* nicht als Eins, sondern als Null. Eine denormalisierte Zahl wird gemäß Gleichung (2.7) umgerechnet.

$$x = (-1)^{(VZB)} * 0,[Mantisse]_2 * 2^{1-Bias} \tag{2.7}$$

2.5.2 Sonderfälle

Neben normalisierten und denormalisierten Zahlen gibt es noch zwei weitere Fälle, die gesondert betrachtet werden.

Je nach Eingabedaten kann es beim Rechnen mit Floatingpoint Zahlen zu einem Überlauf kommen. Der Exponent übersteigt dabei den maximal zulässigen Wertebereich und die Zahl kann nicht mit der gegebenen Genauigkeit gespeichert werden. In diesem Fall gibt es eine Kodierung für „*Unendlich*“. Je nachdem ob das exakte Ergebnis positiv oder negativ ist, wird zwischen „+*Unendlich*“ und „-*Unendlich*“ unterschieden. Alle Bits des Exponenten werden auf Eins, die der Mantisse auf Null gesetzt. Das Vorzeichenbit berechnet sich gemäß der aktuellen Rechenoperation.

Als letztes gibt es noch eine Kodierung für ungültige oder undefinierte Ergebnisse namens *Not a*

Number (NaN). Diese Darstellung wird bespielsweise bei einer Division von Null durch Null zurückgegeben. Hierbei ist es unerheblich ob das VZB gesetzt ist oder nicht. Der Exponent wird auf den maximal möglichen Wert gesetzt und die Mantisse muss einen Wert ungleich Null haben.

Bezeichnung	Vorzeichen	Exponent	Mantisse
Normalisierte Zahl	0 bzw. 1	$0 < E < max$	≥ 0
Denormalisierte Zahl	0 bzw. 1	$= 0$	$\neq 0$
+/- Unendlich	0 bzw. 1	$= max$	$= 0$
NaN	beliebig	$= max$	$\neq 0$

Tabelle 2.1: Vollständiges Schema zur Kodierung von IEEE 754 Zahlen

2.5.3 Runden

Da beim Rechnen mit Fließkomma-Zahlen immer Zwischenergebnisse höherer Genauigkeit mit größeren Bitbreiten auftreten, ist es unausweichlich, diese Zwischenergebnisse zu runden. Der IEEE-Standard sieht vier verschiedene Rundungsmodi vor:

- Round to Zero
- Round to +Infinity
- Round to -Infinity
- Round to nearest (tie to even)

Keiner der vier Rundungsmodi rundet im mathmatischen Sinn. Die ersten drei Modi runden entweder immer in Richtung Null, „+Unendlich“ oder „-Unendlich“. Es ist dabei unerheblich, ob das exakte Ergebnis in Wirklichkeit näher an der Zahl in der entgegengesetzten Richtung liegt. Der letzte Modus *round to nearest* rundet bis auf eine Ausnahme im mathematischen Sinn. Liegt das exakte Ergebnis genau zwischen zwei darstellbaren Zahlen, wird zur nächsten geraden Zahl gerundet. Dies hat den Vorteil, dass statistisch gesehen genauso oft ab- wie aufgerundet wird.

3 Operatoren

Im Folgenden werden die Algorithmen der verschiedenen Operationen im Allgemeinen vorgestellt. Detailiert werden sie dann anhand der konkreten Implementierung in Kapitel 4 beschrieben.

3.1 Multiplikation

Die Multiplikation lässt sich am einfachsten verstehen, wenn man beide Operanden in ihrer binären Fließkommadarstellung miteinander multipliziert.

$$(-1)^{VZB_A \bigoplus VZB_B} * (1,[Mantisse_A]_2 * 1,[Mantisse_B]_2) * 2^{(Exp_A+BIAS+Exp_B+BIAS)-BIAS} \tag{3.1}$$

Man sieht, dass lediglich die Exponenten (Exp_A & Exp_B) addiert und die Mantissen multipliziert werden müssen. Da bei diesem Vorgang der Bias-Wert doppelt im Ergebnis der Exponentenaddition vorkommt, muss er einmal abgezogen werden. Das Vorzeichen ergibt sich durch eine einfache XOR-Verknüpfung. Danach muss das Ergebnis gegebenenfalls normiert und auf jeden Fall gerundet werden. Falls es normiert nicht dargestellt werden kann, wird es wenn möglich in eine denormalisierte Zahl umgewandelt. Zuletzt werden alle möglichen Spezialfälle abgefangen und das Ergebnis ausgegeben. Die möglichen Kombinationen, die zu einem vordefinierten Ergebnis führen, sind in Tabelle 3.1 aufgelistet. Zu beachten ist, dass das Vorzeichen auch hier auf normalem Weg durch eine XOR-Verknüpfung berechnet wird.

A \ B	NaN	Unendlich	Null	andere
NaN	NaN	NaN	NaN	NaN
Unendlich	NaN	Unendlich	NaN	Unendlich
Null	NaN	NaN	Null	Null
andere	NaN	Unendlich	Null	Ergebnis

Tabelle 3.1: Spezialfälle der Mulitplikation und ihr vordefiniertes Ergebnis

3.2 Addition & Subtraktion

Für die Addition betrachten wir zunächst ein kleines Zahlenbeispiel.

$$a = 1,101_2 * 2^1 \tag{3.2}$$

$$b = 1,110_2 * 2^2 \tag{3.3}$$

$$a^* = 0,1101 * 2^2 \tag{3.4}$$

$$x = a^* + b = 1,110_2 * 2^2 + 0,1101_2 * 2^2 \tag{3.5}$$

Man sieht, dass für die Addition zunächst die Exponenten angeglichen werden müssen (a^*). Durch das Angleichen der Exponenten verschiebt sich die Kommastelle der Mantisse entsprechend. Sind beide Exponenten gleich, kann man diesen ausklammern und es müssen nur noch die Mantissen addiert beziehungsweise subtrahiert werden. Das Vorzeichen kann nicht unabhängig vom Rest der Zahl berechnet werden und muss bei der Addtion der Mantissen berücksichtigt werden. Anschließend wird das Ergebnis analog zur Multiplikation normalisiert und gerundet. Vordefinierte Ergebnisse für die Addition und die Subtraktion sind in Tabelle 3.2 aufgelistet. [JSS]

A \ B	NaN	+ Unendlich	- Unendlich	andere
NaN	NaN	NaN	NaN	NaN
+ Unendlich	NaN	+ Unendlich (NaN)	NaN (+ Unendlich)	+ Unendlich (+Unendlich)
- Unendlich	NaN	NaN (- Unendlich)	- Unendlich (NaN)	- Unendlich
andere	NaN	+ Unendlich	- Unendlich	Ergebnis

Tabelle 3.2: Spezialfälle der Addition (Subtraktion) und ihr vordefiniertes Ergebnis

3.3 Division

Ähnlich wie bei der Multiplikation werden Exponent und Mantisse getrennt betrachtet. Statt zu multiplizieren, wird dividiert und die Exponenten werden nicht addiert, sondern subtrahiert.

$$\frac{(-1)^{VZB_A} * (1,[Mantissa_A]_2) * 2^{Exp_A - Bias}}{(-1)^{VZB_B} * (1,[Mantissa_B]_2) * 2^{Exp_B - Bias}} = (-1)^{VZB_A \oplus VZB_B} * 2^{(Exp_A - Exp_B) + BIAS} * \frac{1,[Mantisse_A]_2}{1,[Mantisse_B]_2} \tag{3.6}$$

Die Division der Mantissen lässt sich auf verschiedene Weisen lösen. Für die Bearbeitung stand eine Implementierung des Goldschmidt-Algorithmus zur Verfügung. Dieser führt die Division auf eine Multiplikation zurück. Dabei werden Nenner und Zähler des Bruches wiederholt mit bestimmten Faktoren erweitert. Diese sind so gewählt, dass der Nenner gegen Eins konvergiert. Folglich konvergiert der Zähler gegen den Quotienten. Dies wird in Gleichung (3.7) verdeutlicht.

$$Q = \frac{N}{D} * \frac{F_1}{F_1} * \frac{F_2}{F_2} * \ldots * \frac{F_n}{F_n} \text{ mit } D * F_1 * F_2 * \ldots * F_n \rightarrow 1 \tag{3.7}$$

Die Faktoren F_n bestimmen sich wie folgt:

$$F_{n+1} = 2 - D_n \tag{3.8}$$

Zu beachten ist hierbei, dass der Bruch so skaliert sein muss, dass $0 < D \leq 1$ gilt.[Kar16] Da diese Bedingung der Normalisierungsbedingung entspricht, muss für D die normalisierte Mantisse benutzt werden. Mit jeder Berechnung steigt die Genauigkeit des Ergebnisses. In der ursprünglich vorliegenden Implementierung wurden für das Format Float neun und für Double elf Wiederholungen implementiert. Nach diesen Iterationen entspricht der Zähler des Bruches dem Ergebnis der Division.

Darüber hinaus verwendet die vorliegende Implementierung eine effizientere Methode zur Berechnung der Faktoren. Setzt man $D = 1 - x$ und wählt man als Faktor $1 + x$ so ergibt sich nach einer Iteration die Gleichung (3.9).

$$\frac{N}{D} = \frac{N * (1+x)}{(1-x) * (1+x)} = \frac{N * (1+x)}{1-x^2} \tag{3.9}$$

Für n-Iterationen berechnet sich der Bruch, durch n-maliges erweitern mit $1 + x$, wie in Gleichung (3.10) gezeigt.

$$\frac{N}{D} = \frac{N(1+x^{2^{n-1}})}{1-x^{2^n}} \tag{3.10}$$

Man sieht in Gleichung (3.10), dass durch Ausmultiplizieren der binomischen Formel keine Multiplikation im Nenner mehr durchgeführt werden muss. Zwar muss die Potenz von x berechnet werden, dies ist für die Multiplikation aber sowieso immer noch nötig. Hierfür muss allerdings die Bedingung $\frac{1}{2} \leq D \leq 1$ gelten. Diese ist einfach zu erfüllen, indem die normalisierte Mantisse (D) um ein Bit nach rechts geschoben wird. Das ehemalige Vorkommabit steht nun dahinter, womit die Mantisse mindestens den Wert 0,5 hat.

4 Implementierung

Die folgenden Kapitel beschreiben die einzelnen Entwicklungsschritte, die benutzten Tools und die konkreten Implementierungen.

4.1 Vorgehen

Im Rahmen dieser Arbeit wurde die Implementierung für die einzelnen Operationen in mehreren Teilschritten erarbeitet und getestet. Hierdurch sollte zunächst ein Verständnis für die Algorithmen geschaffen werden, um die Korrektheit der Implementierung der folgenden Teilschritte sicherzustellen.

Zuerst wurden die Operationen zur Multiplikation, Addition und Subtraktion mit fester Bitbreite implementiert. Gewählt wurde hierfür eine Standard-Operandengröße von 32-Bit. Diese wurde gewählt, da hierfür mittels Java relativ einfach zufällige Testfälle generiert werden können. Mittels bereits vorhandener Java Routinen konnten zufällige Fließkommazahlen im Float-Format erstellt werden, deren Ergebnis berechnet und die Werte in hexadezimaler Darstellung in eine Verilog-Testbench exportiert wurden. Mittels des freien Tools „Icarus Verilog“ [Ica] wurden die Implentierungen und Testbenches simuliert und die resultierenden Signalverläufe mittels „GTKWave“ [GTK] zur Analyse benutzt. Zu diesem Zeitpunkt wurden außerdem schon spezielle Testfälle für die Randfälle der eigenen Implementierung erstellt und getestet. Das heißt konkret, dass händisch Testfälle berechnet wurden, sodass alle möglichen *it-then-else* Fälle mindestens einmal betreten wurden. Dies wurde mittels des Verilog-Befehls „**$display**("debug msq")“, welcher den entsprechenden String auf der Konsole ausgibt, überprüft.

Im nächsten Schritt wurde diese Lösung parametrisiert. Ab diesem Zeitpunkt konnten beliebige Bitbreiten getestet werden. Einzige Beschränkung hierbei ist jedoch, gemäß Aufgabenstellung, dass beide Eingabeoperanden im selben Format vorliegen. Auch das Ergebnis wird nur in diesem Format zurückgegeben.

Alle Module beschreiben zunächst den Datenpfad der jeweiligen Operation und im Anschluss einen endlichen Automaten, der den Datenfluss steuert.

Als letztes wurden die Verilog Beschreibungen in den CGRA-Generator integriert.

Zur Synthese der einzelnen Module und des kompletten CGRAs kam das Synthesetool „Vivado“ des FPGA-Herstellers „Xilinx“ zum Einsatz. [Viv]

4.2 Parametrisierung

Wie bereits beschrieben, wird für jede CGRA Komposition eine komplett eigenständigte Verilog Beschreibung generiert. Es werden Verilog Dateien zur Beschreibung der einzelnen PEs, ALUs, des Interconnects und für jede mathematische oder arithmetische sowie für jede Speicher-Operation erstellt. Für jeden Operator und für jede Operandengröße gibt es eine eigene Beschreibung. Für den AMIDAR-Prozessor würde das bedeuten, dass für jede Floatingpoint-Operation zwei Verilog Dateien, eine für das Java Datenformat Float und eine für das Double-

Format, vorliegen müssen. Da das CGRA aber auch in UltraSynth verwendet wird, müssen auch Operandengrößen unterstützt werden, die nicht dem Standard entsprechen und eine beliebige Bitbreite aufweisen. Für jede Bitbreite ist darüber hinaus jede Kombination von Exponenten- und Mantissengröße denkbar. Ebenfalls variabel ist die Datenbusgröße des CGRAs. Das heißt, dass die Verbindung zwischen den einzelnen PEs und auch die Verbindung zwischen CGRA und Prozessor unterschiedliche Bitbreiten haben kann. Dadurch kann es sein, dass die Operanden breiter als der Datenpfad sind und in mehreren Takten geladen werden müssen. Es ist also nicht praktikabel für jeden möglichen Fall eine statische Beschreibung bereit zu halten.

Der Generator passt für jeden Operator eine vorliegende Verilog-Beschreibung entsprechend der Komposition an. Im Wesentlichen müssen hierfür die verschiedenen Register- und Datenbusgrößen angepasst werden. Darüber hinaus müssen Shift-Weiten, Normierungs- und Rundungsbedingungen entsprechend modifiziert werden. Auch die spezifischen konstanten Bitmuster für die Darstellung der Sonderformen und der Biaswert müssen dem aktuellen Format angepasst werden.

Um diese variable Code-Generierung zu realisieren, standen zwei mögliche Lösungswege zur Wahl. Ein möglicher Weg bestand darin, während der Generierung an den entsprechenden Stellen die Registerbreiten und alle anderen Werte aus Java-Variablen zu berechnen und direkt im generierten Verilog-Code zu ersetzen.

Die andere und implementierte Möglichkeit ist, den Verilog-Code selbst zu parametrisieren. Das bedeutet, alle varible Registerbreiten sind von Parametern abhängig, die zu Beginn des Verilog-Moduls definiert werden. So ist die Beschreibung für jede mögliche Exponenten/Mantissen Kombination exakt gleich. Nur wenige Parameter müssen während der Generierung durch den jeweiligen variablen Wert ersetzt werden.

Hauptvorteil dieser Lösung ist es, dass der entstehende Code praktisch selbst dokumentiert ist. Man erkennt die variablen Stellen anhand der Platzhalter für die Parameter und kann anhand ihrer Namen die Funktion der Register oder Signalbündel ableiten. Auch die Integration in den Generator gestaltete sich einfacher, da nur an drei bis fünf Stellen Java-Code für die Ersetzung zuständig ist, der Rest des Verilog-Codes ist unveränderlich.

Konkret werden für jeden Operanden nur drei Parameter durch den Generator gesetzt und zwar die Größe des Datenpfades, des Exponenten und der Mantisse. Davon werden die Gesamtoperandengrößen, der Biaswert und die Kodierungen für NaN, +/- Null und +/- Unendlich abgeleitet.

```
    localparam  OP_EXPONENT = 8,
                OP_MANTISSA = 23,
                DP_WIDTH = 32,
                OP_WIDTH = 1 + OP_EXPONENT + OP_MANTISSA,
                POS_ZERO = {1'b0, {(OP_WIDTH-1){1'b0}}},
                POS_INF = {1'b0, {(OP_EXPONENT){1'b1}}, {(OP_MANTISSA){1'b0}}},
                NAN = {1'b0, {(OP_EXPONENT){1'b1}}, 1'b1, {(OP_MANTISSA-1){1'b1}}};
```

Quelltext 4.1: Beispiel der drei Parameter und den daraus Abgeleiteten

Wenn die Operandengröße die Datenbusbreite übersteigt, müssen die Operanden in mehreren Takten geladen werden. Dabei werden zuerst die *Least Significant Bits* (LSBs) und in jedem weiteren Takt die nächst höherwertigen Bits geladen. Die Anzahl der Bits, die pro Takt geladen werden, hängt von der Datenbusgröße ab.

Für dieses multizyklische Laden ist es notwendig aus der Operanden- und Datenpfadbreite die Anzahl der dafür benötigten Takte zu berechnen. Die Anzahl wird ebenfalls als Parameter namens „CYCLE“ berechnet und gespeichert. Hierfür muss die Operandengröße lediglich durch die Busbreite geteilt werden. Da die Verilog Parameter ganzzahlig sind und in jedem Fall abgerundet werden, kann alleine mit diesem Ergebnis keine zuverlässige Aussage über die Anzahl benötigter Takte getroffen werden. Für den Fall, dass die Operandengröße kein Vielfaches der Busbreite ist, gibt der Parameter „ODD“ an, ob in einem zusätzlichen Takt noch ein Teil des Operanden, der nicht den gesamten Bus füllt, geladen werden muss.

```
localparam   CYCLE = OP_WIDTH / DP_WIDTH,
             ODD = OP_WIDTH % DP_WIDTH;
```

Quelltext 4.2: Parameter für multizyklisches Laden

Mit Hilfe beider Parameter kann später in der Implementierung bestimmt werden, wie viele Takte zum Laden der Operanden gebraucht werden.

4.3 Multiplikation

Nach dem Einlesen der Operanden müssen diese für die folgenden Berechnungen vorbereitet werden. Denormalisierte Operanden müssen erkannt und das hidden-bit muss entsprechend ergänzt werden.
Wenn der Exponent eine denormalisierte Zahl signalisiert, wird die Mantisse von links mit einer Null erweitert, bei einer normalisierten Zahl mit einer Eins. Im denormaliserten Fall wird der Exponent für die folgende Addition auf den kleinstmöglichen Wert des Wertebereichs gesetzt. Das entspricht dem verschobenen Exponenten $1-BIAS$. Dies ist, unabhängig von der Breite des Exponenten, immer die dezimale Eins. Bei einer normalisierten Zahl muss der Exponent nicht weiter angepasst werden.

```
wire [OP_MANTISSA:0] ext_man_a;
assign ext_man_a[OP_MANTISSA-1:0] = man_a;
assign ext_man_a[OP_MANTISSA] = (exp_a != 0) ? 1 : 0;
```

Quelltext 4.3: Anpassung Mantisse als Beispiel an Operand a

Mit diesen angepassten Werten können beide Mantissen ohne Weiteres multipliziert werden. Per Definition ist das binäre Ergebnis dieser Operation doppelt so breit wie die Breite der erweiterten Mantissen. Zwar kann dieser Wert nicht komplett als Ergebnis ausgegeben werden, da hierfür die Breite der Ergebnismantisse nicht ausreicht, für weitere Umformungen und Rundungen wird er aber trotzdem komplett berechnet und zwischengespeichert.

Die angepassten Exponenten werden parallel zur Mantissenmultiplikation addiert. Da beide Exponenten noch den BIAS-Wert enthalten, muss darauf geachtet werden, dass an dieser Stelle der BIAS-Wert einmal abgezogen wird, damit das Ergebnis der erwarteten Darstellung entspricht. Da es durch bestimmte Eingangskombinationen, zum Beispiel zweier denormalisierter Zahlen, auch zu negativen Ergebnissen kommen kann, wird der Ergebnisexponent als Zweier-Komplement Zahl interpretiert. Außerdem ist der Ergebnisexponent um ein Bit breiter

als die Eingangsexponenten, um Pufferüberläufe zu erkennen. Das zusätzliche Bit ermöglicht es auch, zu große Exponenten zu speichern. Dies ist nötig, da der Ergebnisexponent durch die spätere Normierung wieder in den zulässigen Wertebereich kommen kann.

Nach Addition und Multiplikation muss das Ergebnis noch möglichst normalisiert werden. Da die Mantissen um ihre Vorkommastelle erweitert wurden, hat auch das Ergebnis eine „gedachte" Kommastelle und zwar nach den ersten beiden *Most Significant Bits*(MSBs). Um nun die Normalisierung durchzuführen werden drei verschiedene Fälle unterschieden. Sollten diese beiden Bits die Kombination „01" darstellen, ist das Ergebnis bereits normalisiert und es kann mit dem nächsten Schritt fortgefahren werden. Für die Fälle „11" und „10" trifft dies nicht zu. Die Ergebnismantisse ist nicht normalisiert und muss um eine Stelle nach rechts verschoben werden. Das entspricht dem Verschieben des Kommas um eine Stelle nach links. Aufgrund dieser Verschiebung muss der Exponent folglich um Eins erhöht werden.
Sind die beiden MSBs gleich „00", wird die Mantisse so weit nach links geschoben, bis der geforderte Fall „01" eintritt. Die Anzahl der Linksshifts muss in diesem Fall vom Exponenten abgezogen werden, um den Wert des Ergebnisses nicht zu verändern.
Dieser Linksshift während der Normalisierung macht ersichtlich warum das Zwischenergebnis in kompletter Genauigkeit gespeichert wurde. Die LSBs des exakten Ergebnisses werden so in den für das spätere Ergebnis relevanten Teil des Mantissenergebnisses geshiftet(Abb. 4.1).

Beschrieben wurde dieser Linksshift mittels einer while-Schleife, welche die Anzahl der führenden Nullen ermittelt. Eine Beschreibung mittels Case-Statement wäre auch möglich gewesen, ließe sich mittels reiner Verilog-Parametrisierung jedoch nur sehr aufwendig realisieren. Die while-Schleife wird vom Synthese Werkzeug in einen Dekoder umgewandelt.

```
reg [$clog2(2*OP_MANTISSA)-1:0] l_shift;
integer i;
always@(*) begin
  i = 0;
  while (i <= 2*OP_MANTISSA && mult_man[2*OP_MANTISSA-i] == 0) begin
    i = i + 1;
  end
  l_shift = i;
end
```

Quelltext 4.4: While-Schleife ermittelt führende Nullen

Bei der Normalisierung kann es durch den möglichen Rechts-Shift der Mantisse und dem damit verbundenen Erhöhen des Ergebnisexponenten zu einem Overflow des Ergebnisexponenten kommen. Da mit dem verschobenen Exponenten gerechnet wird, also der Bias-Wert schon enthalten ist, ist der Ergebnisexponent im Fall eines Overflow größer oder gleich dem maximalen Wert, der mit den Exponentenbits dargestellt werden kann. Ein möglicher Overflow des Exponenten muss an dieser Stelle nicht weiter berücksichtigt werden, da es während dem späteren Runden ein weiteres Mal zu einem Overflow kommen kann und zu diesem Zeitpunkt abgefangen wird.

Sollte der Ergebnisexponent nach der Normalisierung zu klein werden, also kleiner oder gleich Null werden, bedeutet das an dieser Stelle nicht, dass das Ergebnis zu klein ist, um in der gegebenen Bitbreite zurückgegeben werden zu können, sondern nur, dass das Ergebnis nicht in normalisierter Form gespeichert werden kann.

Es kann jedoch möglich sein das Ergebnis in denormalisierter Form anzugeben. Da nach der erfolglosen Normalisierung der Ergebnisexponent einen negativen Wert angenommen hat, wird zunächst der Betrag des Ergebnisexponenten gebildet. Um diesen Wert wird die Mantisse nach rechts geshiftet. Die MSB werden dabei mit Nullen aufgefüllt. Außerdem wird der Wert auf den Ergebnisexponenten aufaddiert. Der Ergebnisexponent hat somit den Wert Null. Da der Exponentenwert Null aber wie in Abschnitt 2.5.2 beschrieben nicht zum Wertebereich des verschobenen Exponenten gehört, muss der Ergebnisexponent um Eins erhöht und die Mantisse ein weiteres Mal nach rechts geshiftet werden. Da die Ergebnismantisse nun aber denormalisiert ist, wird der Ergebnisexponent wieder auf Null gesetzt. Da bei dieser Denormalisierung der Ergebnisexponent auf jeden Fall auf Null gesetzt wird, kann auf die Additon verzichtet werden.

```
wire underflow;
wire [2*OP_MANTISSA+1:0] underflow_shift_man;
wire signed [OP_EXPONENT+1:0] underflow_exp;
assign underflow = (add_exp <= 0) ? 1 : 0;
assign underflow_shift_man = mult_man >> -add_exp + 1;
assign underflow_exp = {(OP_EXPONENT+2){1'd0}};
```

Quelltext 4.5: Denormalisierung des Ergebnisses

Um nun das Resultat ausgeben zu können, muss das exakt zwischengespeicherte Ergebnis auf die Mantissenbreite der Operanden angepasst werden und gegebenenfalls gerundet werden. In Abb. 4.1 sieht man welches die relevanten Bits sind. Darüber hinaus sind *round-* und *sticky-bit* eingezeichnet, welche für die Rundung von Bedeutung sind.

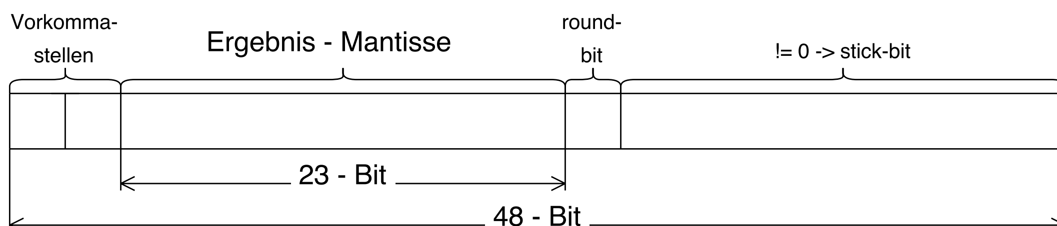


Abbildung 4.1: Aufteilung des Ergebnisses der Mantissenmultiplikation

Im einfachsten Fall ist das *round-bit* gleich Null und es muss nicht gerundet werden. Ist es gleich Eins und eines der darauffolgenden Bits ist ungleich Null, dann ist das exakte Ergebnis näher an der nächst größeren darstellbaren Floatingpoint Zahl als an der nächst niedrigeren darstellbaren Zahl und es muss aufgerundet werden. Dafür wird der für das Ergebnis relevante Teil um Eins erhöht.
Ist das Round-bit gleich Eins und alle restlichen Bits sind Null, dann ist das exakte Ergebnis genau zwischen zwei darstellbaren Zahlen. Dieser Konflikt wird im gewählten Verfahren *(tie to even)* so gelöst, dass zur nächsten geraden Zahl gerundet wird. Ist das LSB des relevanten Teils gleich Eins, muss aufgerundet, also plus Eins gerechnet werden. Ist es Null, wird abgerundet und es muss nichts weiter gemacht werden.

Durch ein mögliches Aufrunden kann es ein letztes Mal zu einem Übertrag in die zweite Vorkommastelle kommen, wodurch das Ergebnis denormalisiert wird. Im letzten Schritt wird das Ergebnis daher noch einmal normalisiert und der Exponent danach auf einen Overflow

überprüft. Ebenfalls werden alle Eingangskombinationen überprüft, die per Definition ein vorbestimmtes Ergebnis liefern.

Das exakte Ergebnis wird dann aus Vorzeichen, Exponent und Mantisse zusammengesetzt, in ein Register geladen und von dort in einem oder mehreren Takten ausgegeben.

4.4 Addition & Subtraktion

Da Addition und Subtraktion sich in der Implementierung nur geringfügig unterscheiden, wird hier zuerst die Addition ausführlich beschrieben und nachfolgend werden die Änderungen ergänzt, die für die Subtraktion nötig sind. Dies ist besonders einfach, da sowohl Addition als auch Subtraktion auf eine Zweier-Komplement (2K) Addition zurückgeführt werden.

Wie bereits in Abschnitt 3.2 beschrieben müssen die Exponenten beider Operanden einander angeglichen werden. In der Implementierung im Rahmen dieser Bachelorarbeit wird die Zahl mit dem kleineren Exponenten an die Zahl mit dem größeren Exponenten angepasst.

In einem ersten Schritt werden hierfür beide Exponenten miteinander verglichen und die Zahl mit dem kleineren Exponenten als Operand „A“ gespeichert. Da es sein kann, dass der Datenpfad kleiner als die Operandengröße ist und die Operanden dadurch in mehreren Takten eingelesen werden, werden diese zunächst in einem Register zwischengespeichert. Da der Exponent in den MSBs des Operanden gespeichert ist, muss die Zahl zunächst vollständig geladen sein. In einem zusätzlichen Takt werden die Operanden dann, falls erforderlich, miteinander vertauscht. Diesen zusätzlichen Takt kann man vermeiden, wenn man die Hardware, die zum Anpassen der Operanden nötig ist, für beide Operandenregister vorsieht. Eine weitere Möglichkeit könnte sein, die Operanden zuerst mit den MSBs von außen in das Modul zu laden. Hierdurch könnte man schon nach dem Laden der ersten Bits die Exponenten miteinander vergleichen und bereits im nächsten Takt des Ladens in den entsprechend richtigen Registern speichern. Diese Variante ist aber nicht immer praktikabel, da sie nicht funktioniert, sobald der Exponent breiter als der Datenpfad ist.

Bevor nun die Zahlen aneinander angepasst werden können, werden zunächst wieder denormalisierte Zahlen erkannt und die Mantissen und Exponenten angepasst. Die Anpassung der Exponenten erfolgt identisch zur Anpassung bei der Multiplikation. Die Mantissen werden allerdings nicht nur um eine Vorkommastelle, sondern um drei Vorkommastellen erweitert. Das MSB speichert das Vorzeichen für die folgende Addition und wird zunächst auf Null gesetzt. Das nächste Bit entspricht der zweiten Vorkommastelle und wird auch auf Null gesetzt. Das dritte Bit entspricht der ersten Vorkommastelle und wird, je nachdem, ob die Zahl normalisiert oder denormalisiert ist, auf Eins oder Null gesetzt. Desweiteren werden die Mantissen auf der rechten Seite um drei weitere LSBs erweitert. Diese werden auf Null gesetzt und sind für die spätere Angleichung der beiden Zahlen von Bedeutung.

Nach diesen Vorbereitungen ist sichergestellt, dass der Operand mit dem kleineren Exponenten im Register für Operand A gespeichert ist und alle Mantissen und Exponenten entsprechend vorbereitet sind. Für die Anpassung der Exponenten wird zunächst die Differenz von Exponent B und Exponent A gebildet. Um den Wert dieser Differenz wird die Mantisse von Operand A nach rechts geshiftet. Durch diesen Rechtsshift muss auch der Exponent von A um die gleiche Anzahl erhöht werden. Da Exponent A dadurch den gleichen Wert wie Exponent B annimmt, muss diese Rechnung nicht durchgeführt werden. Ab diesem Zeitpunkt sind die Exponenten angepasst und der Exponent von Operand B entspricht dem Ergebnisexponenten.

Liegen beide Exponenten sehr weit auseinander wird auch die Differenz und damit die Anzahl der Rechtsshifts sehr groß. Möchte man nun ein absolut exaktes Ergebnis berechnen würde die Mantisse von Operand A dadarch sehr breit und entsprechend würde auch der Addierer für die beiden Mantissen sehr groß. Schon bei einfacher Genauigkeit, also 32 Bit breiten Zahlen, müsste ein Shifter mit mehr als 200 Stellen implementiert werden. Da das Ergebnis nach der Addition aber sowieso auf 23 Stellen gerundet werden muss, kann dies beim Shiften der Mantisse von Operand A direkt berücksichtigt werden. Im Allgemeinen werden zur Rundung nach IEEE 754 drei LSBs mehr benötigt als im exakten Ergebnis gespeichert werden können. Dies sind die drei Bits um welche die Mantissen auf der rechten Seite erweitert wurden. Von links nach rechts heißen sie „*Guard-*“, „*Round-*“ und „*Sticky-Bit* “. Zusammen entscheiden diese Bits später ob auf- oder abgerundet wird. Guard- und Round-Bit bestimmen sich nur durch den Rechtsshift. Das Sticky-Bit bleibt Eins, sobald es während des Rechtsshifts einmal auf Eins gesetzt wurde.

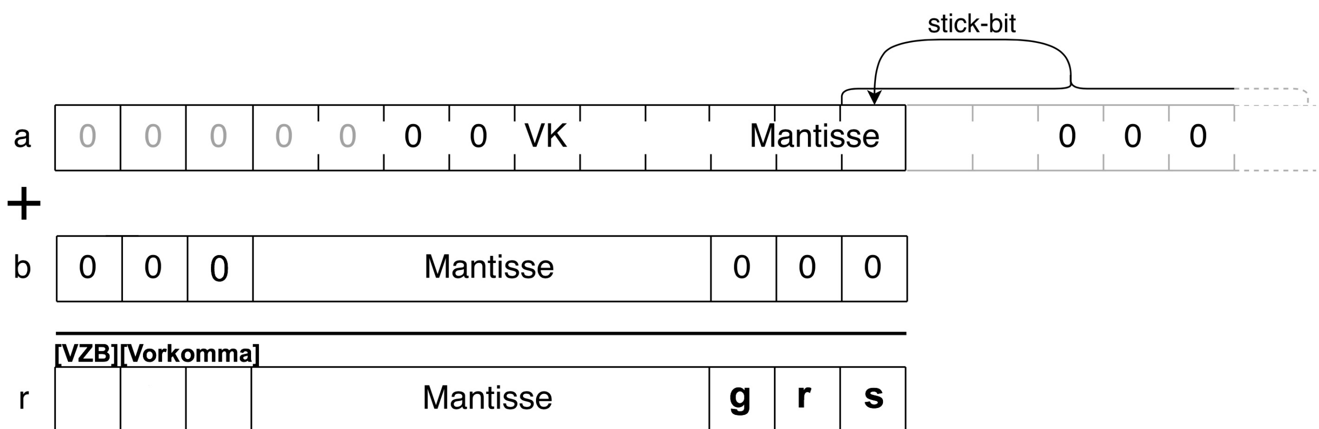


Abbildung 4.2: Anpassung von Mantisse A und Addition beider Mantissen

Bei einer Mehrtaktimplementierung des Rechtsshifts könnte das aktuelle Sticky-Bit mit dem Bit nach dem Shift verodert werden, um das neue Sticky-Bit zu bestimmen. Da der Shift aber in einem Takt durchgeführt wird, muss das Sticky-Bit berechnet werden. Dazu werden drei Fälle unterschieden. Bei einem Shift um weniger als drei Stellen bleibt das Sticky-Bit in jedem Fall Null. Bei einem Shift um mehr als die Mantissenbreite inklusive der drei zusätzlichen LSBs und des Vorkommabits, wird das Sticky-Bit auf Eins gesetzt, wenn die erweiterte Mantisse ungleich Null ist. Ist sie gleich Null wurden nur Nullen nach rechts geschoben und das Sticky-Bit berechnet sich somit ebenfalls zu Null. In allen anderen Fällen müssen die Bits, die nach rechts aus der Mantisse rausgeschoben wurden miteinander verodert werden. Das Ergebnis dieser Operation ist das Sticky-Bit.

Als Letztes werden beide Mantissen, falls das jeweilige Vorzeichenbit gesetzt ist, negiert. Anschließend werden beide Mantissen addiert. Das Ergebnis dieser Addition ist wieder eine 2K-Zahl. Das Vorzeichenbit der Mantissenaddition kann direkt als Vorzeichen des endgültigen Ergebnisses übernommen werden. Ist es gesetzt muss das Ergebnis der Additon negiert werden, da gemäß IEEE 754 die Mantisse betragsmäßig gespeichert wird.

Ist der Betrag gebildet und das Vorzeichen extrahiert wird die Ergebnismantisse analog zur Multiplikation genormt. Zu beachten ist, dass bei einem Rechtsshift der Ergebnismantisse das Sticky-Bit weiterhin gesondert berechnet werden muss.

Die Ergebnismantisse wird ähnlich zur Multiplikation gerundet. Ist das Guard-Bit gesetzt muss entschieden werden, ob gerundet werden muss oder nicht. Sind in diesem Fall Round-

oder Sticky-Bit gesetzt wird aufgerundet. Sind diese beiden Bits allerdings Null liegt das berechnete Ergebnis genau zwischen zwei darstellbaren Zahlen. Da zur nächsten geraden Zahl gerundet wird, entscheidet das LSB des relevanten Teils der Ergebnismantisse. Ist es gesetzt, ist die Zahl im relevanten Teil ungerade und es muss aufgerundet werden. Sollte das Guard-Bit nicht gesetzt sein wird abgerundet. Es muss nichts weiter getan werden als den relevaten Teil des Additionsergebnisses zu selektieren.

Zuletzt werden alle vordefinierten Ergebnisse abgefangen, genauso wie ein Overflow des Ergebnisexponenten.

Subtraktion

Durch die Bildung des Zweierkomplements der erweiterten Mantissen, ist die Subtraktion eigentlich schon in der bisher beschriebenen Implementierung vorhanden. Die Addition einer negativen Zahl ist letztendlich eine Subtraktion.
Werden aber zwei Zahlen subtrahiert und der Subtrahend ist negativ wird der Subtrahend letztlich addiert. Daher wird, wenn das VZB von Operand B gesetzt ist, nicht die negative Mantisse, sondern die positive Mantisse an den Addierer übergeben. Desweiteren muss der Fall beachtet werden, in dem beide Operanden getauscht wurden. In diesem Fall entspricht das Vorzeichen der Mantissenaddition nicht dem Ergebnisvorzeichen, sondern dessen Negation. Diese muss, während das Ergebnisvorzeichen berechnet wird, rückgängig gemacht werden.

4.5 Division

Zur Implementierung der Division stand eine Referenzimplentierung eines Goldschmidt Dividierers aus dem AMIDAR-Prozessor zur Verfügung. Er wurde von Jakob Karg im Jahr 2016 im Rahmen einer Seminararbeit implementiert. [Kar16] Im Rahmen dieser Bachelorarbeit sollte dieser Dividierer so angepasst werden, dass er mit beliebigen Operandengrößen und beliebigen Kombinationen von Mantissen- und Exponentenbreiten rechnen kann. In der ursprünglichen Implementierung konnte dieser Dividierer lediglich Zahlen im Float- und Double-Format miteinander verrechnen. Der für die Implementierung zugrunde liegende Algorithmus ist in Abschnitt 3.3 beschrieben.

Die Referenzimplementierung ist modular aufgebaut. Im Datenpfad befinden sich fünf Module mit unterschiedlichen Aufgaben.

Das Modul „ExceptionChecker “ erkennt alle Eingangskombinationen, die zu einem vordefinierten Ergebnis führen. Er gibt außerdem mittels des Statussignal „*op_denorm_o*“ an, ob einer oder beide Operanden denormalisiert sind. Mittels „*res_denorm_o*“ zeigt er an, ob das Ergebnis denormalisiert ist.

Das Modul „Square_X“ berechnet das Quadrat des Wertes an seinem Eingang. Es wird für die Berechnung der einzelnen Faktoren benötigt mit denen der Bruch N/D (Gleichung (3.10)) erweitert wird. Das Modul „Mul_Q“ wird ebenfalls für diese Erweiterung benötigt, es multipliziert N und D. Der „DenormHandler“ normalsisiert, beziehungsweise denormalisiert, die beiden Eingangsmantissen. Da bei der Division der Exponent getrennt betrachtet werden kann und der Ergebnisexponent auch von der Anzahl der Shifts, die nötig sind, um die Mantissen zu

normalisieren, abhängig ist, wird er ebenfalls im „DenormHandler“ berechnet. Das letzte Modul „Result_Mul“ überprüft das berechnete Ergebnis auf den maximalen Fehler, der bei der Berechnung entstehen kann.

In der gesamten Referenzimplementierung sind die Register- und Busbreiten darauf ausgelegt, 64 Bit breite Operanden zu berechnen. Sollten jedoch 32 Bit breite Operanden berechnet werden müssen, werden die entsprechenden Signale mit LSBs der Wertigkeit Null soweit aufgefüllt, dass sie der Bitbreite von 64 Bit breiten Operanden entsprechen. So müssen die Berechnungen nicht angepasst werden und es muss nur entschieden werden welcher Teil der verschiedenen Zwischenergebnisse für Float- und welcher Teil für Double-Operanden relevant ist.

Aufgrund dieser Tatsache sind alle Register, die zur Berechnung des Exponenten erforderlich sind, 11 Bit (s. Abb. 2.6) breit. Die iterative Multiplikation der Mantissen mit den Faktoren benutzt 68 Bit breite Datenbusse. Dies sind 15 Bit mehr als für die Darstellung von Mantissen des Formats Double benötigt werden. Diese zusätzlichen Bits sind notwendig, um die nötige Genauigkeit bei der Berechnung und der anschließenden Rundung zu erreichen. [Kar16]

Als erstes wurde das Modul „ExceptionChecker“ angepasst. Es testet alle Eingangskombinationen auf vorbestimmte Ergebnisse. Es gibt außerdem an ob einer oder mehrere Operanden denormalisiert sind. Dafür wurde ein Vergleich auf diese Eingangskombination einmal für das Format Float und einmal für das Format Double implementiert.
Das Modul bekommt beide Operanden mit 64 Bit übergeben und entpackt Exponent und Mantisse für jedes Format getrennt. Daher gibt es im Modul „ExceptionChecker“ jeweils zwei Signalbündel für Mantisse und Exponent. Ein Bündel entspricht dem Float-Format. Das andere Bündel entspricht dem Double-Format. Je nachdem welches Format berechnet wird, wird entschieden welche Signalbündel zur Bestimmung der Ausgabe herangezogen werden. Dafür werden sie mit konstanten Bitmustern der Sonderfälle verglichen. Diese sind jeweils als lokale Parameter mit Bitbreite 32 und 64 gespeichert.

Um dieses Modul zu parametrisieren wurden zunächst die Eingänge von der aktuellen Operandengröße abhängig gemacht. Darauf folgend sind Signalbündel von Exponent und Mantisse von deren jeweiliger Größe abhängig. Damit dies möglich ist muss der „ExceptionChecker“ im übergeordneten Modul „fdiv“ mit den Werten für die Parameter „OP_EXPONENT“ und „OP_MANTISSA“ instanziert werden. Um die parametrisierten Mantissen und Exponenten auf Sonderfälle vergleichen zu können, wurden die lokalen Parameter für spezielle Bitmuster, analog zu Listing 4.1, berechnet. Um weitere Fälle, die zu einem vorbestimmten Ergebnis führen, zu bestimmen, müssen Exponenten miteinander verrechnet und die Mantissen beider Zahlen verglichen werden. All diese Operationen mussten nicht weiter angepasst werden.

Die Module „Sqare_X“ und „Mul_Q“ berechnen beide jeweils ein Produkt, dass für das iterative Erweitern des Bruches benötigt wird. Diese beiden Multiplikationen laufen parallel ab. „Mul_Q“ multipliziert den aktuellen Zähler mit dem aktuellen Faktor. Parallel berechnet „Square_X“ bereits das Quadrat von x_n für die nächste Iteration und addiert um Eins auf.

In der Referenzimplementierung wurden diese Muliplikationen mittels einer Blockmultiplikation und anschließender Addition durch einen Carry-Save-Addierer umgesetzt.

Da die Blockmulitplikation und vor allem der CSA-Addierer nur mit Mühe beliebig parametrisierbar sind, wurde zunächst die Blockmultiplikation durch eine kombinatorische Multiplikati-

on ersetzt. Da in der Dokumentation der Referenzimplementierung keine genaue Angabe über die Anzahl der benötigten zusätzlichen Bits gemacht wurde, wurden das Modul „Mul_Q“ und „Sqare_X“ um einen Parameter (MUL_WIDTH) erweitert, der die Größe beider Eingangsoperanden angibt. Dieser Parameter muss beiden Modulen zur Instanzierung übergeben werden. Im Top-Modul ist er lediglich von der Mantissenbreite und einer beliebig wählbaren Konstante abhängig. Diese Konstante kann variiert werden, sobald die Genauigkeit für exakte Ergebnisse nicht mehr ausreicht.

Wichtig ist, dass die Eingangssignale der Module „Mul_Q“ und „Sqare_X“ im Topmodul mit der Richtigen Anzahl an LSBs der Wertigkeit Null aufgefüllt werden.

Das Modul „DenormHandler“ denormalisiert die Mantissen beider Eingangsoperanden. Außerdem gibt es mittels Statussignalen an, ob es beim Denormalisieren zu einem Over- oder Underflow gekommen ist. Da für die Shiftoperationen der Exponent des jeweiligen Operanden angepasst wird, wird im Folgenden auch der Ergebnisexponent berechnet. Hauptsächlich mussten im Modul „DenormHandler“ die Registergrößen parametrisiert werden. Die einzelnen Berechnungen waren von der Parametrisierung nicht betrofffen. Die einzelnen Shiftoperationen waren mittels zwei Case-Statements realisiert. Eines zum Shiften von Mantissen im Float-Format und eines zum Shiften von Mantissen im Double-Format. Beide Case-Statements wurden durch eine While-Schleife ersetzt, welche die führenden Nullen zählt und ihre Anzahl zurückgibt. Dieses Vorgehen ist analog zu den Shift-Operationen in den Modulen der anderen Operatoren.

Das Modul „Result_Mul“ überprüft, ob das berechnete, angenäherte Ergebnis zu weit vom exakten Ergebnis abweicht. Die Parametrisierung des Moduls gestaltete sich schwierig, da, je nachdem welche Bitbreite die Operanden besaßen, unterschiedlich viele Bits des berechneten Ergebnisses zum Vergleich herangezogen wurden. Diese Aufgabe konnte nicht gelöst werden. Das Modul wurde aus der Implementierung entfernt. Das wirkt sich in seltenen Fällen auf die Genauigkeit aus. Sollte der Fehler des berechneten Ergebnisses zu groß werden, würde das Modul „Result_Mul“ dieses aufrunden. Diese Aufrundung entfällt und das berechnete Ergebnis ist somit minimal kleiner als erwartet.

Im Topmodul, in dem alle beschriebenen Submodule miteinander verschaltet sind, musste nach jeder Änderung der Submodule der Datenpfad angepasst werden. Auch die Vergleiche, die zwischen Float- und Double Datenformat unterschieden haben, mussten alle durch die entsprechend parametrisierte Form ersetzt werden.

Wie man in Abb. 4.3 sieht, kann die Division je nach Art der Eingangsoperanden nach einer unterschiedlichen Zahl an Takten zu einem Ergebnis kommen. Da das CGRA einen statischen Scheduler verwendet, müssen alle Operationen mit einer festen Taktzahl implementiert sein. Dafür wurde das Topmodul um einen Counter erweitert. Die maximal Zahl der Takte wird mittels Parameter übergeben. Erst wenn sie erreicht ist wird das Ergebnis ausgegeben.

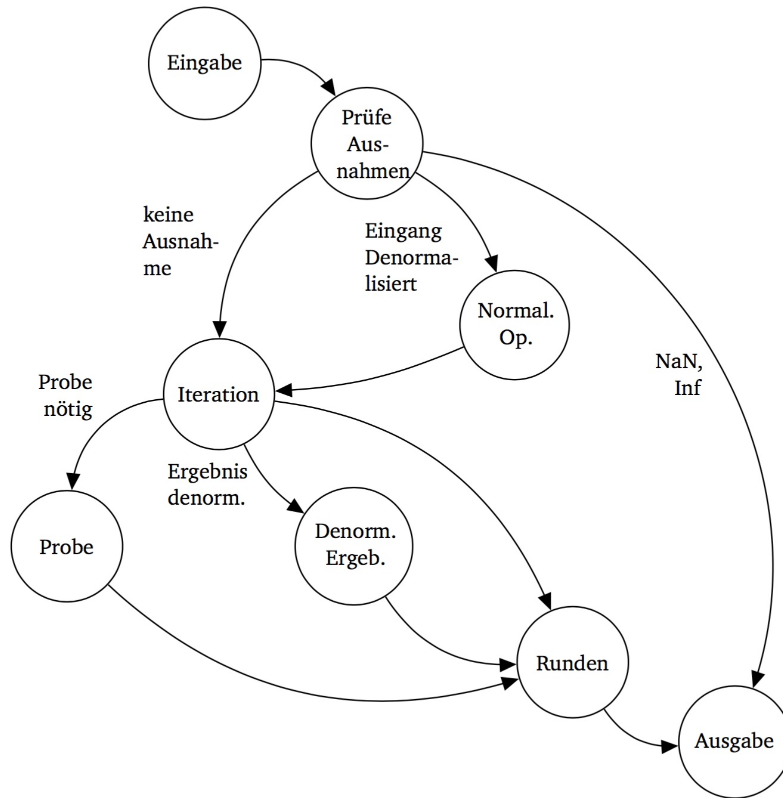


Abbildung 4.3: Ablauf der Division [Kar16]

In den folgenden vier Tabellen wird der Resourcenverbrauch der einzelnen Operationen mit unterschiedlichen Formaten auf einem realen FPGA aufgelistet. Die erste Zahl des Formates gibt die Exponentenbreite, die zweite Zahl die Mantissenbreite an. Für den Resourcenverbrauch werden die benötigten Lookup-Table (LUT), FlipFlops(FF) und DigitalSignalProcessors (DSP) betrachtet. Außerdem wird die ungefähre maximale Taktzahl angegeben. Sie kann aber nur zum Vergleich der einzelnen Formate der unterschiedlichen Operationen zu Rate gezogen werden, da sich bei Integration in ein vollständiges CGRA die längsten Signalpfade anders zusammensetzen können.

Format	LUTs	FF	DSP	MHz
float5x14	455	114	1	≈ 250
float8x23	738	178	2	≈ 208
float9x32	1088	229	4	≈ 166
float11x52	1878	366	9	≈ 143

Tabelle 5.1: Resourcenverbrauch Multiplikation

Format	LUTs	FF	DSP	MHz
float5x14	493	208	0	≈ 200
float8x23	739	320	0	≈ 181
float9x32	1026	417	0	≈ 153
float11x52	1819	632	0	≈ 133

Tabelle 5.2: Resourcenverbrauch Addition

Format	LUTs	FF	DSP	MHz
float5x14	478	208	0	≈ 200
float8x23	745	320	0	≈ 167
float9x32	948	417	0	≈ 125
float11x52	1671	632	0	≈ 111

Tabelle 5.3: Resourcenverbrauch Subtraktion

Format	LUTs	FF	DSP	MHz
float5x14	769	443	8	167
float8x23	1304	577	18	143
float9x32	1665	745	18	125
float11x52	2770	1001	34	111

Tabelle 5.4: Resourcenverbrauch Division

Man erkennt, dass der Ressourcenverbrauch mit steigender Operandengröße zunimmt. Dies ist der Tatsache geschuldet, dass alle mathematischen Operationen kombinatorisch implementiert sind. Bei steigender Bitbreite werden diese kombinatorischen Schaltungen größer. Außerdem fällt auf, dass bei steigenden Bitbreiten die maximal möglichen Frequenzen sinken. Dieser Effekt ist ebenfalls der größer werdenden Kombinatorik der mathematischen Operationen geschuldet. Diese werden in einem Takt ausgeführt. Wenn die Schaltung durch steigende Bitbreite größer wird, wird auch der längste Pfad durch die Schaltung länger. Der längste Pfad ist aber bestimmend für die Frequenz. Diese wird aus seinem Kehrwert bestimmt.

Was außerdem auffällt, ist, dass Addition und Subtraktion keine DSPs benötigen. Das Synthesetool benutzt sie lediglich zur Implementierung von Multiplikationen. Dies ist auch der Grund, weshalb die Division so viele davon benötigt.

Ebenfalls interessant ist, wie sich das Verhältnis von Exponentenbreite zur Mantissenbreite im Resourcenverbrauch niederschlägt.

Hierfür wurden pro Operation die Datenformate Float und Double betrachtet. Einmal mit normaler Aufteilung und einmal mit vertauschten Exponenten- und Mantissenbreiten.

Format	LUTs	FF	DSP
float8x23	738	178	2
float23x8	582	211	0
float11x52	1878	366	9
float52x11	1095	384	1

Tabelle 5.5: Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Multiplikation

Format	LUTs	FF	DSP
float8x23	739	320	0
float23x8	541	275	0
float11x52	1819	632	0
float52x11	1100	509	0

Tabelle 5.6: Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Addition

Format	LUTs	FF	DSP
float8x23	745	320	0
float23x8	536	275	0
float11x52	1671	632	0
float52x11	1095	509	0

Tabelle 5.7: Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Subtraktion

Format	LUTs	FF	DSP
float8x23	1304	577	18
float23x8	1107	612	8
float11x52	2770	1001	34
float52x11	1853	946	8

Tabelle 5.8: Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Division

Bei diesen Vergleichen sieht man sehr gut den Zusammenhang zwischen Breite der Multiplikation und den dafür benötigten DSPs. Ebenfalls ist auffällig, dass sich der Verbrauch an LUTs weitaus stärker ändert als der Verbrauch von FFs.

6 Zusammenfassung

Abschließend kann zusammengefasst werden, dass alle Operationen implementiert werden konnten. Auch die Parametrisierung konnte für alle Module, mit Ausnahme von einem, erfolgreich realisiert werden.

Die Verilog-Beschreibungen der Addition, Subtraktion und Multiplikation bestehen mit jeder beliebigen Exponent/Mantissen Kombination alle zufälligen Tests fehlerfrei. Bei der Implementierung dieser Operationen stellten sich die gewählten Zwischenziele als wertvolle Stütze heraus. Damit ist vor allem die Implementierung der Operationen in fester Bitbreite gemeint, da sie bereits zu Beginn viele Schwierigkeiten im Umgang mit IEEE 754 Floatingpoint Zahlen aufgezeigt hat.

Problematischer war hingegen das Anpassen der Reverenzimplementierung der Division. Dies war zum einen dem komplexen Aufbau des Dividierers geschuldet, zum anderen den teils nicht genau dokumentierten Designentscheidungen. Der Dividierer ist das einzige Modul, das die Tests nicht komplett fehlerfrei bestanden hat.

Im Ausblick ist die Anpassung des letzten Submoduls des Dividierers sicherlich der nächste Schritt, um eine komplett fehlerfreie FALU zu erhalten.

Darüber hinaus können die einzelnen kombinatorischen Berechnungen optimiert und beschleunigt werden. Dies trifft vor allem für die Multiplikationen der Division zu, da sie die Frequenz am stärksten negativ beeinflussen. Allerdings können für bestimmte Anwendungen mit geringen Bitbreiten diese kombinatorischen Schaltungen ausreichend sein. Daher wäre im Rahmen dieser Verbesserung zu prüfen, für welche Fälle überhaupt eine solche Beschleunigung sinnvoll ist.

Eine weitere Herausforderung war die Synthese des kompletten CGRAs. Dies ist dem Umstand geschuldet, dass es von vielen beteiligten Personen entwickelt wird und es dabei immer wieder zu Abstimmungsschwierigkeit kommen kann. Dies äußerte sich während der Evaluation durch unterschiedliche oder fehlende Signale innerhalb der ALUs. Diese mussten händisch angepasst werden.

Abschließend lässt sich sagen, dass die Implementierungen, trotz des fehlenden Moduls in der Division, als Bereicherung für das CGRA-Projekt angesehen werden können.

Abkürzungsverzeichnis

CGRA Coarse Grained Reconfigurable Array

AMIDAR Adaptive Microinstruction Driven Architecture

FU Functional Unit

TDN Token Distribution Network

ALU Arithmetic Logic Unit

IALU Integer Arithmetic Logic Unit

FALU Floatingpoint Arithmetic Logic Unit

FPGA Field Programmable Gate Array

CPU Central Processing Unit

GPU Graphics Processing Unit

PE Processing Element

RF Register File

IEEE Institute of Electrical and Electronics Engineers

VZB Vorzeichenbit

CSA Carry-Save-Adder

LUT Lookup-table

FF FlipFlop

DSP Digital Signal Processor

MSB Most Significant Bit

LSB Least Significant Bit

2K Zweier-Komplement

Abbildungsverzeichnis

2.1 Aufbau des AMIDAR-Prozessors 4
2.2 CGRA mit 16 PEs und „von Neumann“ Interconnect 6
2.3 Aufbau eines PEs 7
2.4 Exemplarische ALU eines PEs 8
2.5 Schema der Bitgruppen von float-Zahlen 9
2.6 Schema der Bitgruppen von double-Zahlen 10

4.1 Aufteilung des Ergebnisses der Mantissenmultiplikation 19
4.2 Anpassung von Mantisse A und Addition beider Mantissen 21
4.3 Ablauf der Division [Kar16] 25

Tabellenverzeichnis

2.1 Vollständiges Schema zur Kodierung von IEEE 754 Zahlen 11

3.1 Spezialfälle der Mulitplikation und ihr vordefiniertes Ergebnis 12
3.2 Spezialfälle der Addition (Subtraktion) und ihr vordefiniertes Ergebnis 13

5.1 Resourcenverbrauch Multiplikation 26
5.2 Resourcenverbrauch Addition 26
5.3 Resourcenverbrauch Subtraktion 26
5.4 Resourcenverbrauch Division 27
5.5 Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Multiplikation 27
5.6 Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Addition . . 27
5.7 Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Subtraktion 28
5.8 Resourcenverbrauch unterschiedlicher Exp./Man. Kombinationen bei Division . . 28

Literaturverzeichnis

[Ami] *AMIDAR*. http://www.amidar.de

[GTK] *GTK Wave*. http://gtkwave.sourceforge.net

[Ica] *Icarus Verilog*. http://iverilog.icarus.com

[iee] *IEEE Standard for Binary Floating-Point Arithmetic*

[JSS] Jyoti Sharma, Sambangi S. Pabbisetty Tarun T. Pabbisetty Tarun ; S, Sivanantham: *Fused Floating-Point Add and Subtract Unit*

[Kar16] Karg, Jakob P.: *FPGA Implementierung von Floatingpoint- und Integer-Dividierern für AMIDAR-Prozessoren*. 2016

[TRH] Tajas Ruschke, Dennis W. Lukas Johannes Jung J. Lukas Johannes Jung ; Hochberger, Christian: *Scheduler for Inhomogeneous and Irregular CGRAs with Support for Complex Control Flow*

[Ult] *UltraSynth*. https://www.ultrasynth.de

[Viv] *Vivado*. https://www.xilinx.com/products/design-tools/vivado.html