



ulm university universität
uulm

Fakultät für
Ingenieurwissenschaften,
Informatik und Psy-
chologie
Institut für Eingebettete
Systeme/Echtzeitsyste-
me

Entwicklung einer arithmetisch-logischen Einheit für ganze Zahlen mit skalierbarer Wortbreite in VHDL

Projektarbeit (16 LP)

Vorgelegt von:
Max Brand
max.brand@uni-ulm.de
Matrikelnummer: 750690

29. Januar 2017

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	1
1.3 Aufbau	2
2 Grundlagen	3
2.1 Field-Programmable-Gate-Array	3
2.2 Zahlendarstellung	3
2.2.1 Positive Darstellung	4
2.2.2 Negative Darstellung	4
2.2.3 Konvertieren	4
2.3 Bewertung der Komponenten	5
2.3.1 Fläche	5
2.3.2 Time-Critical-Path	5
Maximale Taktfrequenz	5
3 Addierer	7
3.1 Funktionsweise	7
3.2 Aufbau	8
3.2.1 Halbaddierer	8
Funktion	8
Komponenten	8
Verwendung	8
3.2.2 Volladdierer	8
Funktion	8
Komponenten	9
Verwendung	9
3.2.3 Carry-Ripple-Adder	9
Funktion	9
Komponenten	10

Verwendung	10
3.2.4 Carry-Select-Adder-Block	11
Funktion	11
Komponenten	12
Verwendung	12
3.2.5 Carry-Select-Adder	13
Funktion	13
Komponenten	13
Verwendung	15
3.3 Bewertung	16
3.3.1 Fläche	16
3.3.2 Time-Critical-Path	17
4 Multiplizierer	18
4.1 Funktionsweise	18
4.2 Aufbau	19
4.2.1 Multiplier-Base-Cell	19
Funktion	19
Komponenten	20
Verwendung	20
4.2.2 Multiplier-Slice	21
Funktion	21
Komponenten	21
Verwendung	21
4.2.3 Vier-Quadranten-Multiplizierer	22
Funktion	22
Komponenten	23
Verwendung	24
4.3 Bewertung	24
4.3.1 Fläche	24
4.3.2 Time-Critical-Path	24
5 Dividierer	26
5.1 Funktionsweise	26
5.2 Aufbau	27
5.2.1 Zweier-Komplement-Konverter	27
Funktion	27
Komponenten	28

Verwendung	28
5.2.2 Absolut-Funktion	29
Funktion	29
Komponenten	29
Verwendung	29
5.2.3 Divider-Slice	29
Funktion	29
Komponenten	30
Verwendung	31
5.2.4 Restoring-Divider	31
Funktion	31
Komponenten	32
Verwendung	32
5.2.5 Dividierer	34
Funktion	34
Komponenten	34
Verwendung	34
5.3 Bewertung	36
5.3.1 Fläche	36
5.3.2 Time-Critical-Path	36
6 Komparator	37
6.1 Funktionsweise	37
6.2 Aufbau	37
6.2.1 Komparator	37
Funktion	37
Komponenten	38
Verwendung	38
6.2.2 Comparing-Slice	38
Funktion	38
Komponenten	39
Verwendung	39
6.2.3 Tree-Comparator	40
Funktion	40
Komponenten	41
Verwendung	42
6.2.4 Word-Comparator	42
Funktion	42

Komponenten	42
Verwendung	43
6.3 Bewertung	43
6.3.1 Fläche	43
6.3.2 Time-Critical-Path	44
7 Rotieren und Schieben	45
7.1 Funktionsweise	45
7.2 Aufbau	46
7.2.1 Shifting-Slice	46
Funktion	46
Komponenten	46
Verwendung	46
7.2.2 Barrel-Shifter	48
Funktion	48
Komponenten	48
Verwendung	49
7.3 Bewertung	50
7.3.1 Fläche	50
7.3.2 Time-Critical-Path	50
8 Logische Verknüpfungen	51
8.1 Funktionsweise	51
8.1.1 Komponenten	51
8.1.2 Verwendung	51
8.2 Bewertung	52
8.2.1 Fläche	52
8.2.2 Time-Critical-Path	53
9 Validierung	54
9.1 Testaufbau	54
9.2 Beispiel	55
10 Verwendung	57
10.1 Übersicht und Schnittstelle	57
10.2 Befehlssatz	57
10.2.1 Interner Befehlssatz	57
10.2.2 Externer Befehlssatz	59
10.3 Demo	60

Inhaltsverzeichnis

11 Fazit und Ausblick	62
A Anhang	63
A.1 Befehlssatz	63
A.2 2-Bit-Komparator	65
Abbildungsverzeichnis	66
Tabellenverzeichnis	67
Literaturverzeichnis	68
Glossar	69

1 Einleitung

1.1 Motivation

Der Kern einer jeden Central-Processing-Unit (CPU) ist ihre Arithmetic-Logical-Unit (ALU), welche die verschiedenen Rechen- und Logikoperationen bereit stellt. Da die Implementation einer ALU sowie über die Geschwindigkeit¹ als auch über die Mächtigkeit² einer CPU entscheidet, ist dies ein wichtiges Teilgebiet der Rechnerarchitekturen.

Zu den notwendigen Grundfunktionen einer ALU zählen die Addition und das Vergleichen von Zahlen. Damit komplexere Programme ausführbar sind, sollte sie allerdings weitere Operationen wie die Multiplikation und Division sowie das Verschieben und Manipulieren von Bits beherrschen.

Je mehr verschiedene Operationen eine ALU beherrscht, desto mächtiger wird die CPU, in welcher sie eingesetzt wird. Ein Paradebeispiel dafür sind heutige CPUs wie sie in Heimcomputern eingesetzt werden. Diese beherrschen eine breite Palette an verschiedenen Funktionen, welche weit über die oben aufgeführten Operationen hinaus gehen. Eine weiteres Beispiel wäre ein digitaler Signalprozessor, dessen Befehlssatz stark von dem einer normalen CPU abweicht.

1.2 Zielsetzung

In dieser Projektarbeit soll eine einfache ALU, welche die Grundrechenarten sowie das Manipulieren und Verschieben von Bits und das Vergleichen von Zahlen beherrscht, entwickelt und in VHDL implementiert werden. Sie soll dabei aus klar strukturierten Teilkomponenten aufgebaut werden, welche die verschiedenen Rechen- und Logikoperatoren veranschaulichen und für den Einsatz zur Lehre geeignet sind. Weiterhin soll die Skalierbarkeit der Operatoren demonstriert werden,

¹Anzahl der ausführbaren Rechenoperationen innerhalb eines gegebenen Zeitmaßes

²Mögliche Operationen, welche von der CPU ausführbar sind

weshalb die gesamte ALU in ihrer Wortbreite konfigurierbar ist. Sie wird dabei direkt in Bits angegeben und muss die Form 2^w mit $w \in \mathbb{N}$ haben.

Bei der Implementierung soll auf die VHDL eigenen Generics und Components zurückgegriffen werden, welche den Code in seiner Lesbarkeit und Wartbarkeit erheblich verbessern und ebenfalls dem Lehrzweck entgegen kommen. Weiterhin ist ein offensichtlicher Vorteil bei der Verwendung von Components der, dass Teilkomponenten der ALU ausgetauscht oder allein verwendet werden können.

Neben der eigentlichen Entwicklung und Implementation sollen auch Testbenches, welche die Korrektheit der entwickelten Teilkomponenten verifizieren, sowie ein Demonstrationsprogramm für das Altera DE2 Board erstellt werden.

1.3 Aufbau

In den nachfolgenden Kapitel soll jeweils eine Teilkomponente für eine Rechenoperation bzw. Klasse³ von Rechenoperationen erarbeitet werden. Die Entwicklung der Teilkomponenten für die Rechen- und Logikoperatoren erfolgt dabei von unten nach oben. Dieses Vorgehen ermöglicht ein einfaches Verständnis der Teilkomponenten, da zu Beginn einfache Komponenten erarbeitet und zu immer komplexeren bis hin zur fertigen Teilkomponente zusammengesetzt werden.

Um die Lesbarkeit und Wartbarkeit des Codes weiter zu verbessern wurde eine Nomenklatur für alle verwendeten Signale und Ports eingeführt (Tabelle 1.1).

Form	Bedeutung
p_<name>	Port einer Komponente
g_<name>	Generic-Wert einer Komponente
s_<name>	Signal innerhalb einer Komponente

Tabelle 1.1: Nomenklatur VHDL-Code

³Beispiel: arithmetisch/logisch links und rechts Schieben sowie rotieren

2 Grundlagen

Folgend werden wichtige Grundlagen, welche zur Implementierung und Bewertung der zu entwickelnden ALU notwendig sind, erklärt.

2.1 Field-Programmable-Gate-Array

Ein Field-Programmable-Gate-Array (FPGA) ist ein durch Register (sog. Lookup-Tables, kurz: LUT) programmierbarer Mikrochip. Mit Hilfe dieser programmierbaren Lookup-Tables ist es möglich spezifische Funktionen umzusetzen. Da ein FPGA aus vielen hunderten bis tausenden LUTs besteht, können durch deren Verschaltung komplexe Strukturen bis hin zu kompletten Rechnerarchitekturen aufgebaut werden. Neben diesen Lookup-Tables besitzt ein FPGA im Normalfall noch weitere Bausteine, wie z.B. Arbeitsspeicher oder Permanentsspeicher, welche für die eigene Implementierung zur Verfügung stehen.

Die Architektur, welche auf dem FPGA erzeugt werden soll, wird mittels der Hardwarebeschreibungssprache VHDL beschrieben und synthetisiert. Damit kann anschließend der FPGA programmiert werden. Hierbei sei zu beachten, dass sich VHDL stark von herkömmlichen Programmiersprachen, welche sequentiell abgearbeitet werden unterscheidet, da es hier keine Ausführung von Programmcode auf einer Rechnerarchitektur gibt.

Um die geplante ALU zu implementieren wird VHDL verwendet.

2.2 Zahlendarstellung

Da eine ALU für gewöhnlich mit positiven sowie negativen Zahlen arbeiten kann, wird eine binäre Darstellung für beide Zahlenbereiche benötigt.

2.2.1 Positive Darstellung

Um positive Zahlen darzustellen, werden den einzelnen Stellen der Binärform jeweils ein Zahlenwert zur Basis Zwei zugewiesen. Die Zahlenwerte beginnen, bei einer Wortbreite von n , mit 2^0 für das Least Significant Bit (LSB) und enden mit 2^{n-1} für das Most Significant Bit (MSB).

Der Binärvektor 10010110_2 kann also wie folgt umgerechnet werden:

$$\begin{aligned}1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 \\= 128 + 16 + 4 + 2 = 150\end{aligned}$$

und entspricht damit der Zahl 150_{10} .

2.2.2 Negative Darstellung

Zur Darstellung negativer Zahlen, dient die selbe Form wie für positive, allerdings mit der Ausnahme, dass das MSB als negativer statt positiver Wert interpretiert wird. Das MSB besitzt also den Wert -2^{n-1} statt 2^{n-1} . Dadurch ist es möglich, Zahlen im Bereich von -2^{n-1} (nur das MSB ist gesetzt) bis $2^{n-1} - 1$ (alle Bits bis auf das MSB sind gesetzt) darzustellen.

Der Binärvektor aus 2.2.1 wird also wie folgt umgerechnet:

$$\begin{aligned}1 * -2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 \\= -128 + 16 + 4 + 2 = -106\end{aligned}$$

und entspricht damit der Zahl -106_{10} . Obwohl der Binärvektor der selbe ist, ist der Wert, welchen er repräsentiert ein anderer. Der Wert ist damit also abhängig von der Interpretation des Binärvektors.

Diese Form der Darstellung ist bekannt als das Zweier-Komplement.

2.2.3 Konvertieren

Um zwischen negativen und positiven Zahlen im Zweier-Komplement zu konvertieren (Vorzeichenwechsel), genügt es, den Binärvektor zu invertieren und mit 1 zu

addieren:

$$\begin{aligned}10010110_2 &= -106_{10} \\ \neg 10010110_2 + 1_2 &= 01101010_2 = 106_{10}\end{aligned}$$

2.3 Bewertung der Komponenten

Zur Bewertung der entwickelten Komponenten werden folgend zwei Maße gegeben. Diese entsprechen nur einem theoretischen Wert, welcher vom synthetisierten VHDL-Code abweicht.

Die Abweichung entsteht durch automatische Optimierungen seitens des Compilers, sowie durch die Umsetzung der Architektur innerhalb des FPGAs, da diese nicht auf Gattern, sondern auf LUTs (vgl. Abschnitt 2.1) basiert.

2.3.1 Fläche

Das Flächenmaß beschreibt die Größe der Komponente anhand der Anzahl von Gattern, welche für ihre Implementierung notwendig sind. Je mehr Gatter eine Komponente besitzt, desto mehr Platz benötigt sie später auf dem FPGA.

2.3.2 Time-Critical-Path

Als zweites Maß wird der Time-Critical-Path verwendet. Dieser beschreibt den längsten möglichen Pfad durch eine Komponente, welchen ein Signal durchlaufen kann (vgl. Worst-Case-Laufzeit bei Algorithmen). Die Länge dieses Pfades wird, ebenfalls wie die Fläche, als eine Anzahl von Gattern beschrieben. Die Länge des Time-Critical-Path (TCP) beeinflusst die spätere Geschwindigkeit der Komponente.

Maximale Taktfrequenz

Als maximale Taktfrequenz wird die Frequenz bezeichnet, bei welcher eine Komponente innerhalb eines Taktzyklus ihre Berechnung fertigstellen kann. Diese maximale Taktfrequenz berechnet sich nun aus dem TCP, da hier die größte Signalverzögerung der Komponente statt findet. Damit die Komponente mit ihrer Berechnung

innerhalb eines Taktes fertig wird, darf diese Signalverzögerung nicht länger als die Dauer zwischen zwei Taktzyklen sein.

Ausgehend von einer Signalverzögerung von $\Delta t = 20 \text{ ps}$ eines Gatters, kann nun die maximal mögliche Taktfrequenz wie folgt berechnet werden:

$$\frac{1}{\Delta t \cdot G_{TCP}} = \frac{1}{20 \text{ ps} \cdot 10} = 5 \text{ GHz}$$

wobei G_{TCP} der Länge des TCP in Gattern entspricht. Für das Beispiel wurde für G_{TCP} ein Wert von 10 gewählt.

Die maximale Taktfrequenz, bei welcher eine Komponente mit einem TCP der Länge 10 innerhalb eines Taktes mit ihrer Berechnung fertig wird, beträgt 5 GHz.

3 Addierer

Als Addierer wird ein Carry-Select-Adder (CSA) verwendet. Dieser wird aus $2 \cdot m$ Carry-Ripple-Adder (CRA) der Breite p aufgebaut, wobei immer zwei CRA zu einem Block gehören. Die daraus resultierende Wortbreite entspricht $m \cdot p$ Bit, da die Operanden in m Blöcke aufgeteilt werden, welche jeweils eine Breite von p Bit besitzen. Diese Blöcke werden dann jeweils parallel addiert.

3.1 Funktionsweise

Der CSA teilt die beiden Operanden in m Blöcke mit einer Größe von p Bit auf. Innerhalb jedes Blocks finden zwei Additionen statt. Eine davon unter der Annahme, dass die Addition des vorhergehenden Blocks eine 0 als Übertrag liefert, die andere geht entgegengesetzt von einer 1 als Übertrag aus. Dadurch können große Teile der beiden Operanden parallel addiert werden. Die Auswahl, welches der Ergebnisse nun verwendet wird, findet durch einen Multiplexer statt, welcher den tatsächlichen Übertrag des vorhergehenden Blocks erhält[5].

Der Vorteil des CSA liegt darin, dass der Übertrag nicht durch die gesamte Addition der beiden Operanden propagiert werden muss, sondern nur durch deren Teilblöcke. Die Laufzeit hängt lediglich von der Wortbreite p der CRA und der Multiplexer ab. Der Nachteil des CSA liegt offensichtlich darin, dass er deutlich mehr Platz benötigt als ein CRA.

3.2 Aufbau

3.2.1 Halbaddierer

Funktion

Der Halbaddierer (HA) addiert zwei Bits, indem er die Summe s und den Übertrag c der beiden Eingabebits a und b berechnet:

$$\begin{aligned}s &= x \otimes y \\c &= x \wedge y\end{aligned}$$

Komponenten

Der HA besteht aus einem AND- und XOR-Gatter um die Summe und den Übertrag zu berechnen (Abbildung 3.1).

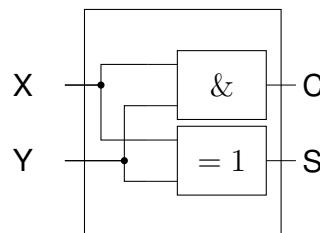


Abbildung 3.1: Halbaddierer

Verwendung

Der HA besitzt wie oben beschrieben die beiden Eingänge p_x und p_y sowie die Ausgänge p_s und p_c .

3.2.2 Volladdierer

Funktion

Der Volladdierer (VA) berechnet die Summe s und den Übertrag c_{out} aus den beiden Bits a und b sowie einem zusätzlichen Übertrags-Bit c_{in} , welches von einem

```

1 PORT (
2     p_x : in  std_logic;
3     p_y : in  std_logic;
4     p_s : out std_logic;
5     p_c : out std_logic
6 );
7

```

Quellcode 3.1: Halbaddierer-Schnittstelle

vorhergehenden VA geliefert wird.

Komponenten

Der VA wird mit Hilfe von zwei HAs und einem OR-Gatter implementiert, welche bereits in 3.2.1 definiert wurden (Abbildung 3.2).

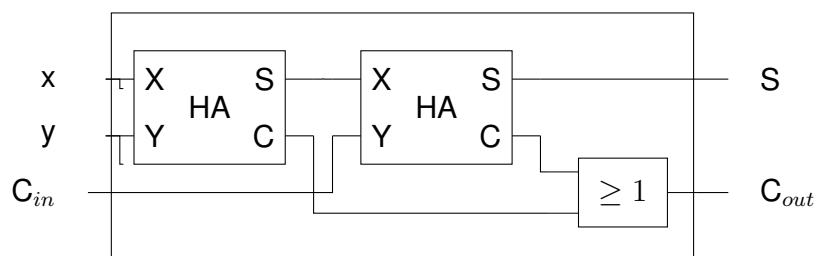


Abbildung 3.2: Volladdierer

Verwendung

Der VA besitzt wie im Schaltbild die drei Eingänge p_x , p_y und p_c_{in} sowie die beiden Ausgänge p_s und p_c_{out} .

3.2.3 Carry-Ripple-Adder

Funktion

Mit Hilfe des zuvor implementierten VA kann nun ein CRA implementiert werden. Hierfür werden, abhängig von der Wortbreite, p VA aneinander gereiht. Dabei wird

```

1 PORT (
2     p_x      : in  std_logic;
3     p_y      : in  std_logic;
4     p_c_in   : in  std_logic;
5     p_s      : out std_logic;
6     p_c_out  : out std_logic
7 );
8

```

Quellcode 3.2: Volladdierer-Schnittstelle

der c_{out} Ausgang eines VA an den c_{in} Eingang des nachfolgenden angeschlossen. Durch das weiter geben des Übertrags von VA zu VA kommt die Addition der beiden Operanden zustande (vgl. schriftliche Addition).

Komponenten

Es werden p VA für einen p -Bit breiten CRA verwendet (Abbildung 3.3).

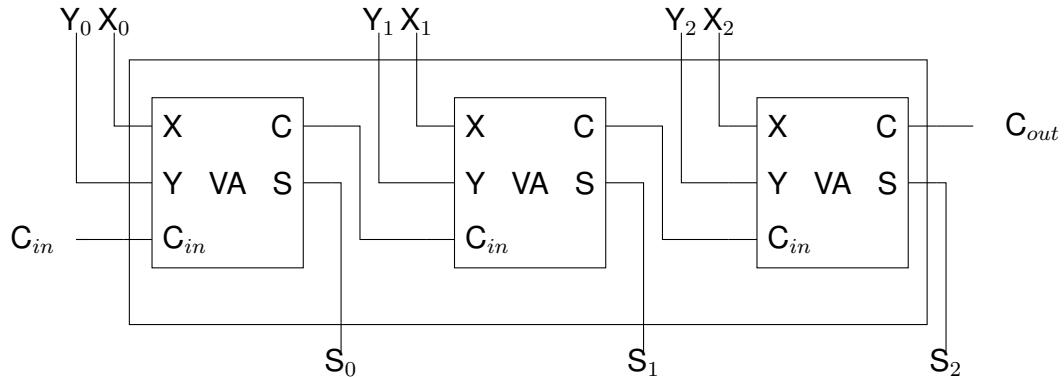


Abbildung 3.3: 3-Bit-Carry-Ripple-Adder

Verwendung

Der CRA besitzt die beiden p -Bit-Operanden p_op_1 , p_op_2 und p_c_in als Eingang. Als Ausgang besitzt er das p -Bit-Ergebnis p_result und ein Bit p_c_out für den Übertrag. Außerdem werden bei dieser Komponente zum ersten mal in die VHDL eigenen Generics verwendet. Mit Hilfe dieser wird die Größe des CRA in Bit

angegeben (`g_size`).

Zu beachten ist, dass die übergebene Größe um ein Bit kleiner ist als die tatsächliche vom CRA implementierte Größe. Für einen 8-Bit-CRA wird also der Wert von `g_size` auf 7 gesetzt. Dies liegt dem verwendeten Datentyp `std_logic_vector` zugrunde, welcher wie ein Array bei den meisten Programmiersprachen einen Startindex von 0 besitzt. Im Gegensatz zu einer Programmiersprache wie z.B. C++ wird hier aber nicht die Größe des Vektors n sondern die Spanne, also 0 bis $n - 1$, angegeben.

```
1 GENERIC (
2     g_size : integer := 7
3 );
4 PORT (
5     p_c_in      : in  std_logic;
6     p_op_1      : in  std_logic_vector(g_size DOWNTO 0);
7     p_op_2      : in  std_logic_vector(g_size DOWNTO 0);
8     p_result    : out std_logic_vector(g_size DOWNTO 0);
9     p_c_out     : out std_logic
10 );
11
```

Quellcode 3.3: Carry-Ripple-Adder-Schnittstelle

3.2.4 Carry-Select-Adder-Block

Funktion

Auf Basis des CRA kann nun der Carry-Select-Adder-Block (CSAB) entworfen werden. Dieser vereint zwei n -Bit CRA und einen Multiplexer zu einer Komponente. Der CSAB berechnet mit den beiden CRA jeweils das Ergebnis der Operanden unter der Annahme, dass der vorhergehende CSAB eine 1 oder 0 als Übertragbit liefert. Durch das Übertragbit des vorgehenden CSAB wird dann ein Multiplexer angesteuert, welcher das richtige der berechneten Ergebnisse an den Ausgang weitergibt. Das Übertragbit des richtigen Ergebnisses wird dann ebenfalls an den nächsten CSAB gegeben.

Komponenten

Es werden zwei p -Bit-CRA und ein Multiplexer für den CSAB verwendet (Abbildung 3.4).

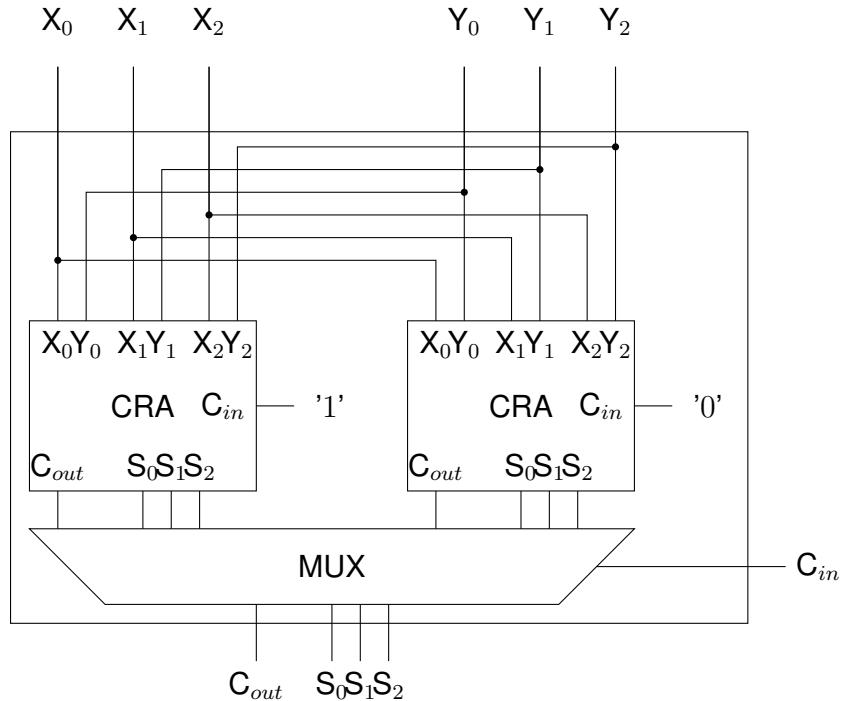


Abbildung 3.4: Carry-Select-Adder-Block mit 3-Bit Wortbreite

Verwendung

Das Interface des CSAB ist identisch mit dem des CRA. Der Unterschied findet sich nur in der Implementierung. Im direkten Vergleich der beiden Komponenten ist ihre Funktion ebenfalls identisch. Der CSAB benötigt allerdings doppelt so viel Platz wie der CRA, was offensichtlich ein Nachteil ist. Dieser wird durch eine bessere Performance, welche beim Verbund mehrerer CSAB erreicht wird, wieder aufgewogen (Abschnitt 3.2.5).

```

1  GENERIC (
2      g_size  : integer := 3
3  );
4  PORT (
5      p_c_in    : in  std_logic;
6      p_op_1    : in  std_logic_vector(g_size DOWNTO 0);
7      p_op_2    : in  std_logic_vector(g_size DOWNTO 0);
8      p_result  : out std_logic_vector(g_size DOWNTO 0);
9      p_cout    : out std_logic
10 );
11

```

Quellcode 3.4: Carry-Select-Adder-Block-Schnittstelle

3.2.5 Carry-Select-Adder

Funktion

Wie in 3.1 bereits erklärt, kann durch das Aneinanderreihen von mehreren CSAB nun der eigentliche CSA implementiert werden. Zusätzlich zum eigentlichen CSA-Addierwerk wird die Funktion einer Subtraktion mit im CSA umgesetzt. Hierfür gibt es ein 1-Bit breites Signal, welches das Addierwerk dazu anweist, zu subtrahieren. Für die Subtraktion wird das Signal als Übertrag in den CSA geleitet und zum Invertieren des zweiten Operanden mit Hilfe eines XOR-Gatters verwendet. Durch diese Konstruktion kann nun der zweite Operand in das sogenannte Zweier-Komplement überführt werden. Es wird der zweite Operand vom ersten subtrahiert.

Der CSA besitzt zudem ein zusätzliches 1-Bit breites Signal als Ausgang, welches einen Überlauf signalisiert. Dieses wird im Fall einer vorzeichenlosen Addition/Subtraktion durch die drei MSB der beiden Operanden und des Ergebnisses berechnet. Ist die Addition/Subtraktion vorzeichenbehaftet, so kann hierfür direkt das Übertragbit des CSA verwendet werden. Damit der Überlauf richtig berechnet werden kann, benötigt das Addierwerk zusätzlich ein Bit welches an gibt, ob die Operanden vorzeichenbehaftet sind oder nicht. [1]

Komponenten

Das CSA-Addierwerk wird aus m p -Bit-CSAB zusammengesetzt. Die Bit-Breite des CSA errechnet sich mit $n = p \cdot m$. Außerdem werden zusätzliche Gatter benötigt,

3 Addierer

um den Überlauf zu berechnen und die Subtraktion zu ermöglichen (Abbildung 3.5).

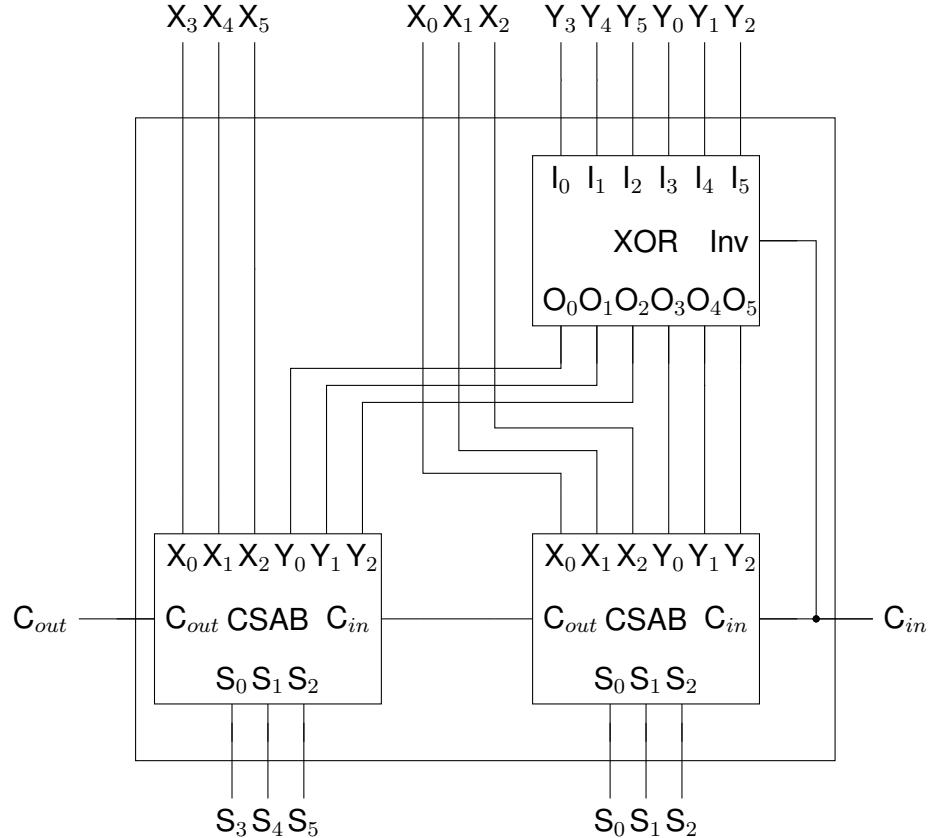


Abbildung 3.5: 6-Bit-CSA

Die Berechnung des Überlaufes und das dazu notwendige Vorzeichen-Bit wurden in Abbildung 3.5 zum Zwecke der Übersichtlichkeit weggelassen. Wie oben erwähnt, wäre der Überlauf bei einer vorzeichenbehafteten Addition/ Subtraktion direkt das Übertrags-Bit des CSA. Bei einer vorzeichenlosen Operation wird es durch folgende logische Verknüpfung berechnet:

$$ovflw = (X_{n-1} \wedge Y_{n-1}) \wedge \neg S_{n-1} \vee (\neg X_{n-1} \wedge \neg Y_{n-1}) \wedge S_{n-1}$$

Verwendung

Die Wortbreite des CSA-Addierwerks wird nicht, wie bisher über einen Parameter angegeben, sondern über zwei. Der erste, `g_block_size`, bezeichnet die Wortbreite der einzelnen CSABs, aus welchen der CSA aufgebaut wird. Als zweiter Parameter `g_blocks` wird angegeben, aus wie vielen CSABs das Addierwerk besteht. Insgesamt ergibt sich damit eine Wortbreite von $g_block_size \cdot g_blocks$ Bit.

Neben den aus den CRA/CSAB bekannten Ports (`p_op_1`, `p_op_2` und `p_result`) existieren noch drei weitere. Das Signal `p_sgnd` gibt an, ob die Operation vorzeichenlos ($= 0$) oder vorzeichenbehaftet ($= 1$) ist, `p_ovflw` repräsentiert einen Überlauf und mit dem Signal `p_sub` kann eine Subtraktion angeordnet werden.

Die Größe für die beiden Operanden und das Ergebnis des CSA müssen außerdem in Abhängigkeit von p (`g_block_size`) und m (`g_blocks`) angegeben werden. Da die erwartete Größe $m \cdot p$ ist, muss nun der obere Wert der Spanne mit $p \cdot m - 1$ (bzw. $(g_block_size + 1) \cdot (g_blocks + 1) - 1$) berechnet werden.

```

1  GENERIC (
2      g_block_size  : integer := 3;
3      g_blocks      : integer := 1
4  );
5  PORT (
6      p_sgnd      : in  std_logic;
7      p_sub       : in  std_logic;
8      p_op_1      : in  std_logic_vector((g_block_size + 1) *
9                                         (g_blocks + 1) - 1 DOWNTO 0);
10     p_op_2      : in  std_logic_vector((g_block_size + 1) *
11                                         (g_blocks + 1) - 1 DOWNTO 0);
12     p_result    : out std_logic_vector((g_block_size + 1) *
13                                         (g_blocks + 1) - 1 DOWNTO 0);
14     p_ovflw     : out std_logic
15  );
16

```

Quellcode 3.5: Carry-Select-Adder Schnittstelle

3.3 Bewertung

Zur Bewertung des Addierwerks werden die Fläche und der TCP verwendet.

3.3.1 Fläche

Man sieht dass der CSA im Vergleich zum CRA wesentlich mehr Fläche benötigt. Auch wenn man die erweiterten Funktionen des hier implementierten CSA ignoriert (Überlauf und Subtraktion), ist er immer noch doppelt so groß wie der CRA.

Komponente	Zusammensetzung	Formel [Gatter]
HA	AND + XOR	2
VA	$2 \cdot \text{HA} + \text{OR}$	5
CRA	$p \cdot \text{VA}$	$p \cdot 5$
CSAB	$2 \cdot \text{CRA} + \text{MUX}$	$p \cdot 13 + 3$
CSA	$m \cdot \text{CSAB} + \text{XOR} + \text{MUX} + 5$	$p \cdot m \cdot 14 + m \cdot 3 + 11$

Tabelle 3.1: Flächenverbrauch Carry-Select-Adder

3.3.2 Time-Critical-Path

Als TCP wird der längste Signalweg innerhalb einer abgeschlossenen Komponente bezeichnet. Also das Signal, das die Geschwindigkeit/Verzögerung der Komponente maßgeblich bestimmt. Beim CRA ist dies das Weiterreichen des Übertragbits zwischen den VA. Bei einem 32-Bit-CRA muss der Übertrag beginnend beim ersten VA bis zum letzten VA, also 32 mal, übertragen werden. Dies entspricht $32 \cdot 3 = 96$ Gattern. Ausgehend von einer durchschnittlichen Gatterlaufzeit von 20 ps, wäre das eine Verzögerung von $96 \cdot 20 \text{ ps} = 1,92 \text{ ns}$. Damit ist die maximal zulässige Taktfrequenz, in welcher das Addierwerk die Operation innerhalb eines Taktzyklus abschließen kann, auf $\frac{1}{1,92 \text{ ns}} = 520 \text{ MHz}$ beschränkt.

Der große Vorteil des CSA gegenüber dem CRA ist, dass größere Blöcke der Operanden gleichzeitig berechnet werden können. Es muss also nur einmal der Übertrag im ersten Block berechnet werden, welcher dann alle weiteren Ausgaben bestimmt, da alle anderen Blöcke zusammen mit dem ersten Block gleichzeitig berechnet werden. Im Vergleich zu einem 32-Bit-CRA, ergibt sich bei einem CSA mit $8 \cdot 4$ Bit, also 8 Blöcke mit jeweils 4 Bit Breite eine Verzögerung von $(4 \cdot 3 + 8) \cdot 20 \text{ ps} = 400 \text{ ps}$. Dies entspricht einer maximalen Taktfrequenz von $\frac{1}{400 \text{ ps}} = 2500 \text{ MHz}$, also fast dem Fünffachen der des CRA.

Sofern auf der Zielplattform genügend Fläche vorhanden ist, sollte der CSA verwendet werden, da dieser deutlich schneller ist als der CRA.

4 Multiplizierer

Zum Multiplizieren soll ein Ein-Quadranten-Multiplizierer (EQM) für vorzeichenlose bzw. ein Vier-Quadranten-Multiplizierer (VQM) für vorzeichenbehaftete Operanden verwendet werden. Es wird die Implementation eines VQM verfolgt, welcher so modifiziert wird, dass er durch das Abschalten bestimmter Bereiche zu einem EQM wird. Das Umschalten zwischen diesen beiden Varianten wird verwendet, um die Multiplikation mit vorzeichenbehafteten und vorzeichenlosen Operanden durchführen zu können.

4.1 Funktionsweise

Diese Art des Multiplizierens ist die einfachste Implementation eines Multiplizierers und wurde aus Gründen des besseren Verständnisses zu Lehrzwecken gewählt. Zur Multiplikation wird einer der beiden n -Bit-Operanden um m Bit nach links geschoben und mit Bit m des anderen Operanden durch ein AND-Gatter verknüpft, was der Multiplikation von zwei Bit entspricht. Es gilt $m \in [0, n - 1]$. Das Ergebnis der Multiplikation wird berechnet, indem alle erzeugten Bit-Folgen miteinander addiert werden. Als Beispiel soll das Produkt $10_{10} \cdot 11_{10} = 1011_2 \cdot 1010_2$ berechnet werden:

$$\begin{array}{r} 1 \ 0 \ 1 \ 1 \cdot 1 \ 0 \ 1 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \\ 1 \ 0 \ 1 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ + \ 1 \ 0 \ 1 \ 1 \\ \hline 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \end{array}$$

Die Multiplikation ergibt $1101110_2 = 110_{10}$, was der Operation des EQM entspricht, da sich Operanden und Ergebnis nur im ersten Quadranten befinden. Im Fall einer vorzeichenbehafteten Multiplikation werden die beiden Operanden als Darstellung im Zweier-Komplement interpretiert und multipliziert: $1011_2 \cdot 0011_2 = -5_{10} \cdot 3_{10}$. Hier-

bei können beide Operanden und das Ergebnis positiv oder negativ sein, weshalb dies der Multiplikation des VQM entspricht, welcher in allen vier Quadranten arbeitet. Beim VQM werden zu Beginn beide Operanden um ein Bit erweitert (Vorzeichen muss beachtet werden) und nach jeder Stufe wird erneut eine Vorzeichenerweiterung durchgeführt (rot, fett):

$$\begin{array}{r}
 1 \ 1 \ 0 \ 1 \ 1 \cdot 0 \ 0 \ 0 \ 1 \ 1 \\
 \hline
 & \textbf{1} \ 1 \ 1 \ 0 \ 1 \ 1 \\
 + & 1 \ 1 \ 0 \ 1 \ 1 \\
 \hline
 & \textbf{1} \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \\
 + & 0 \ 0 \ 0 \ 0 \ 0 \\
 \hline
 & \textbf{1} \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \\
 + & 0 \ 0 \ 0 \ 0 \ 0 \\
 \hline
 & \textbf{1} \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \\
 + & 0 \ 0 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1
 \end{array}$$

Das Produkt der Multiplikation mit dem VQM ergibt $111110001_2 = -15_{10}$. Da der VQM die beiden Operanden immer als Zweier-Komplement interpretiert, werden sowohl der VQM als auch der EQM benötigt, damit vorzeichenlose und vorzeichenbehaftete Zahlen multipliziert werden können.

4.2 Aufbau

4.2.1 Multiplier-Base-Cell

Funktion

Die Multiplier-Base-Cell (MBC) ist die grundlegende Komponente, aus welcher der VQM aufgebaut wird. Sie wird dazu verwendet, zwei Bits in einer Stufe¹ zu addieren, wobei berücksichtigt wird, ob der zweite Operand eine Addition auf dieser Stufe vorsieht ($= 0$) oder nicht ($= 1$, vgl. Multiplikationstafeln).

Sie addiert jeweils das Bit der übergeordneten Stufe (S_{in}) und ein Bit des ersten Operanden (A) zusammen mit (C_{in}). Ist das Bit des zweiten Operanden (X) auf dieser Stufe 0, so wird keine Addition mit A durchgeführt, sondern nur die Addition von S_{in} und C_{in} .

¹entspricht einem Additionsschritt bei der Multiplikation

Komponenten

Die MBC besteht aus einem VA zum Addieren der beiden Bits sowie einem AND-Gatter, um die Addition vom ersten Operanden auszublenden, wenn der zweite Operand auf dieser Stufe eine 0 besitzt (Abbildung 4.1).

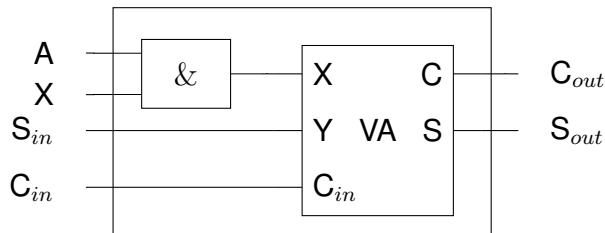


Abbildung 4.1: Multiplier-Base-Cell

Verwendung

Die Ein- und Ausgänge der Komponente sind wie in Abbildung 4.1 dargestellt. `p_s_in` ist die Summe aus der darüber liegenden Stufe (`p_s_out`), `p_a_in` ist ein Bit von einem der beiden Operanden und `p_x_in` das Bit des anderen Operanden, welches zum Ausblenden verwendet wird. `p_s_out` wird wie oben bereits erwähnt an die nächste Stufe weitergegeben und der Übertrag innerhalb einer Stufe wird über `p_c_in` und `p_c_out` durchgereicht.

```

1 PORT (
2     p_s_in  : in  std_logic;
3     p_a_in  : in  std_logic;
4     p_x_in  : in  std_logic;
5     p_c_in  : in  std_logic;
6     p_s_out : out std_logic;
7     p_c_out : out std_logic
8 );
9

```

Quellcode 4.1: Multiplier-Base-Cell Schnittstelle

4.2.2 Multiplier-Slice

Funktion

Das Multiplier-Slice (MS) implementiert eine Stufe des VQM, wobei sie bei einer Wortbreite von n Bit aus $n + 1$ miteinander verbundenen MBCs besteht. Sie erhält als Eingabe einen der beiden n Bit breiten Operanden und ein Bit zum Ausblenden der Addition, welches bei der Stufe m jeweils dem Bit m des zweiten Operanden entspricht. Weiterhin erhält sie eine n Bit breite Eingabe, welche der Summe der übergeordneten Stufe entspricht. Ist die aktuelle Stufe die oberste, so wird stattdessen entweder eine 0-Folge eingegeben oder ein dritter Operand. Durch die Möglichkeit, einen dritten Operanden in die oberste Stufe zu geben, ist der VQM in der Lage $x \cdot y + z$ zu berechnen statt nur die Multiplikation von x und y durch zu führen. Zusätzlich besitzt die Stufe ein Bit, welches zwischen dem VQM und EQM umschaltet, indem das MSB für die Ausgabe anders gewählt wird. Im Falle des VQM, wird die Summe der letzten MBC für das MSB verwendet, soll ein EQM realisiert werden, wird der Übertrag der vorletzten MBC als MSB verwendet.

Komponenten

Ein n Bit breites MS besteht aus $n + 1$ MBC und drei weiteren Logik-Gattern (Multiplexer) zum Auswählen des MSB der Ausgabe (Abbildung 4.2). Der Übertrag der letzten MBC wird nicht benötigt und bleibt deshalb offen.

Verwendung

In der Schnittstellendefinition der Komponente finden sich alle oben erwähnten Ein- und Ausgänge. Außerdem muss mit `g_size` die Wortbreite in Bit angegeben werden. `p_s_in` und `p_a_in` sind die beiden n Bit breiten Eingänge für einen der Operanden und das Ergebnis der übergeordneten Stufe. Das Ergebnis wird über `p_result` für die nächste Stufe bereitgestellt und über `p_sgnd` beeinflusst. `p_x_in` ist, wie bereits erklärt, Bit m des zweiten Operanden. `p_c_in` wird abgesehen von der letzten Stufe des VQM mit einer 0 beschalten.

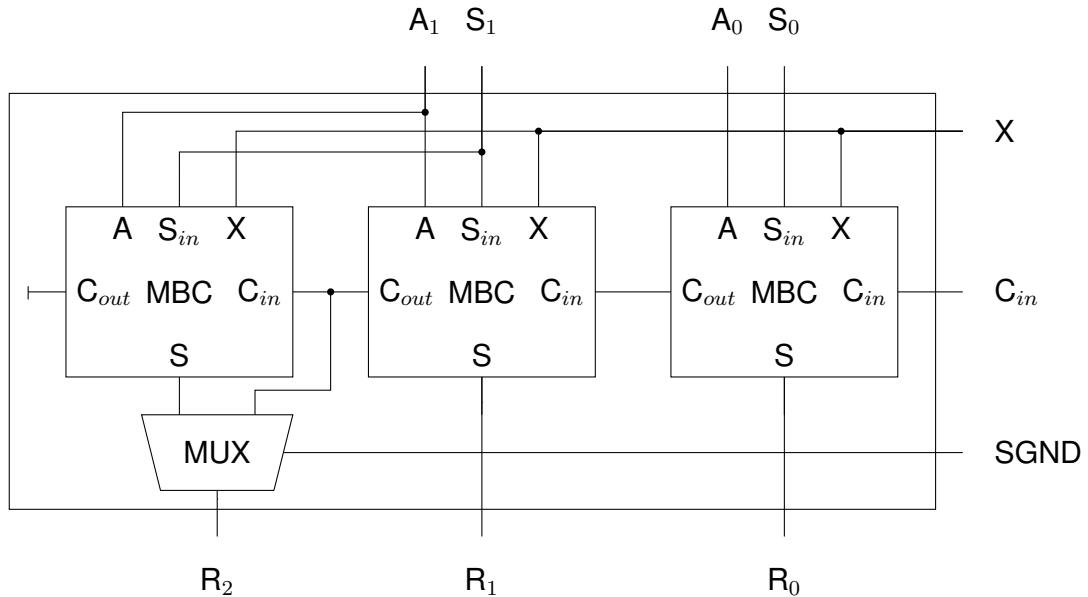


Abbildung 4.2: 2 Bit Multiplier-Slice

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_sgnd      : in  std_logic;
6      p_x_in       : in  std_logic;
7      p_c_in       : in  std_logic;
8      p_s_in       : in  std_logic_vector(g_size DOWNTO 0);
9      p_a_in       : in  std_logic_vector(g_size DOWNTO 0);
10     p_result    : out std_logic_vector(g_size + 1 DOWNTO 0)
11  );
12

```

Quellcode 4.2: Multiplier-Slice Schnittstelle

4.2.3 Vier-Quadranten-Multiplizierer

Funktion

Mit Hilfe des zuvor implementierten MS kann nun der VQM zusammengesetzt werden. Die genaue Funktionsweise wurde bereits in 4.1 beschrieben. Es muss dabei lediglich die letzte Stufe adaptiert werden, indem ihr p_c_in mit dem Bit für die vor-

zeichenbehaftete Berechnung verbunden wird und ihr Operand an p_a_in ebenfalls durch das Bit invertiert wird (vgl. bilden des Zweier-Komplements).

Komponenten

Der VQM besteht aus n MSs, wobei jedes eine Breite von n Bit aufweist. Weiterhin werden ein XOR-Gatter und ein AND-Gatter benötigt (Abbildung 4.3). Die Ausgabe, welche die doppelte Wortbreite der Eingabe besitzt, wurde in High (H_{n-1}, \dots, H_1, H_0) und Low (L_{n-1}, \dots, L_1, L_0) aufgeteilt.

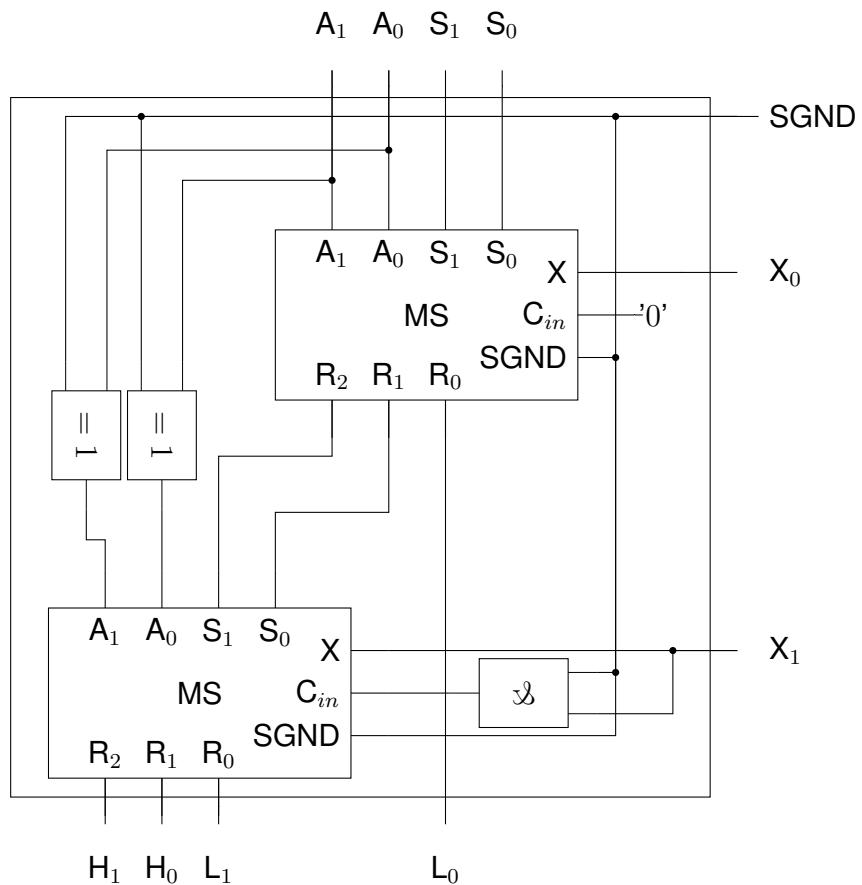


Abbildung 4.3: 2-Bit-Vier-Quadranten-Multiplizierer

Verwendung

Der VQM verfügt über vier Eingänge und zwei Ausgänge. Neben dem p_sgnd-Eingang zum Umschalten zwischen vorzeichenloser und vorzeichenbehafteter Multiplikation (EQM und VQM) besitzt er noch drei n Bit breite Eingänge für die beiden zu multiplizierenden Operanden und die kostenlose Addition welche die Implementierung mit sich bringt. Das Ergebnis wird an zwei separaten Ausgängen bereitgestellt, welche die n höherwertigen und die n niederwertigen Bit liefern, da die Multiplikation ein Ergebnis der Breite $2n$ liefert: p_result_hi und p_result_lo.

```
1 GENERIC (
2     g_size : integer := 7
3 );
4 PORT (
5     p_sgnd      : in  std_logic;
6     p_op_1       : in  std_logic_vector(g_size DOWNTO 0);
7     p_op_2       : in  std_logic_vector(g_size DOWNTO 0);
8     p_add        : in  std_logic_vector(g_size DOWNTO 0);
9     p_result_lo : out std_logic_vector(g_size DOWNTO 0);
10    p_result_hi : out std_logic_vector(g_size DOWNTO 0)
11 );
12
```

Quellcode 4.3: Vier-Quadranten-Multiplizierer-Schnittstelle

4.3 Bewertung

4.3.1 Fläche

Die Fläche entspricht der Anzahl der verwendeten Gatter in Abhängigkeit der Wortbreite.

4.3.2 Time-Critical-Path

Der TCP des VQM entspricht wie beim CSA dem längsten Pfad des Übertrags-Bit. Dieses muss einmal durch die komplette erste Stufe geleitet werden und danach

Komponente	Zusammensetzung	Formel [Gatter]
MBC	AND + VA	6
MS	$(n + 1) \cdot \text{MBC} + \text{MUX}$	$n \cdot 6 + 3$
VQM	$n \cdot \text{MS} + n \cdot \text{XOR} + \text{AND}$	$6n^2 + 4n + 1$

Tabelle 4.1: Flächenverbrauch Vier-Quadranten-Multiplizierer

durch je zwei MBC der nachfolgenden Stufen. Innerhalb der MBC allerdings nur durch den VA, was einer Länge von 5 Gattern entspricht. Bei einem n -Bit-VQM entspricht dies $(n + 1) \cdot 5$ Gattern für die erste Stufe und $(n - 1) \cdot 10$ Gattern für die nachfolgenden Stufen. Insgesamt ergibt sich also ein TCP von $(n + 1) \cdot 5 + (n - 1) \cdot 10$ Gattern = $15 \cdot n$ Gattern.

Ausgehend von einer durchschnittlichen Gatterlaufzeit von 20 ps und einer Wortbreite von 32 Bit ergibt sich somit eine maximal mögliche Taktfrequenz von

$$\frac{1}{15 \cdot 32 \cdot 20 \text{ ps}} = 104 \text{ MHz}$$

Diese ist deutlich niedriger als beim Addierwerk, weshalb eine Multiplikation teurer als eine Addition ist.

5 Dividierer

Zum Dividieren soll ein Restoring-Divider (RD) eingesetzt werden. Dieser ist wie auch der VQM einfach zu verstehen und ist damit bestens zu Lehrzwecken geeignet. Der RD selbst beherrscht in der hier vorgestellten Implementierung nur die Division von zwei vorzeichenlosen Operanden. Um auch vorzeichenbehaftete dividieren zu können wurde hier absichtlich ein anderes Konzept umgesetzt, welches eine zweite Möglichkeit zum Umgang mit einer solchen Operation aufzeigen soll. Bei diesem Konzept soll der Dividierer nicht zwischen vorzeichenloser und vorzeichenbehafteter Division umgeschaltet werden, sondern mit positiven Zahlen arbeiten und das Vorzeichen nach der Berechnung anpassen.

5.1 Funktionsweise

Der RD ist wie auch bei der Multiplikation, eine der einfachsten Implementierungen, welche wieder dem Lehrzweck dienen soll. Es wird das LSB des Divisors zum MSB des Dividenden geschoben und eine Subtraktion durchgeführt. Ist das Ergebnis negativ, so wird der Divisor um eine Stelle nach rechts geschoben und der Vorgang wiederholt, bis das Ergebnis positiv ist. Bei einem positiven Resultat wird der Dividend mit dem resultierenden Ergebnis ersetzt und der Vorgang wiederholt, bis das LSB des Divisors auf gleicher Höhe wie das LSB des Dividenden ist.

Die Vorzeichen der einzelnen Schritte ergeben das invertierte Ergebnis der Division. Das Ergebnis der letzten Subtraktion entspricht dem Rest. Als Beispiel soll

$(\frac{11}{5})_{10} = (\frac{1011}{101})_2$ berechnet werden:

$$\begin{array}{r}
 & 0 & 0 & 1 & 0 & 1 & 1 \\
 - & 1 & 0 & 1 \\
 \hline
 & \textbf{1} & 0 & 0 \\
 & 0 & 1 & 0 & 1 & 1 \\
 - & 1 & 0 & 1 \\
 \hline
 & \textbf{1} & 0 & 1 \\
 & 1 & 0 & 1 & 1 \\
 - & 1 & 0 & 1 \\
 \hline
 & \textcolor{red}{0} & 0 & 0 \\
 & \textcolor{blue}{0} & \textcolor{blue}{0} & \textcolor{blue}{0} & 1 \\
 - & 1 & 0 & 1 \\
 \hline
 & \textcolor{red}{1} & 0 & 0
 \end{array}$$

Die Vorzeichen (rot, fett) von unten nach oben gelesen und invertiert ergeben den Quotienten: $0010_2 = 2_{10}$. Die letzte erfolgreiche Subtraktion (blau, kursiv) entspricht dem Rest der Division: $0001_2 = 1_{10}$.

Um eine Division mit vorzeichenbehafteten Zahlen durchzuführen, werden zuerst beide Operanden durch eine Absolut-Funktion (ABS) geleitet, damit beide positiv sind. Anschließend werden beide n Bit breiten Operanden mit Hilfe eines $n + 1$ Bit breiten RD dividiert und anschließend das Vorzeichen wieder angepasst. Der RD muss hierfür ein Bit breiter sein als die ursprünglichen Operanden, da sonst nicht der komplette Zahlenbereich ($[0, 2^n]$) abgedeckt werden kann. [5]

5.2 Aufbau

5.2.1 Zweier-Komplement-Konverter

Funktion

Der Zweier-Komplement-Konverter (TCC) wird dazu verwendet, Zahlen zwischen vorzeichenbehafteter (Zweier-Komplement-) und vorzeichenloser Darstellung zu konvertieren. Die Komponente verwendet dafür ein fest vorgegebenes Schema:

1. Das LSB bleibt bei jeder Konvertierung unverändert.

2. Die Position der 1, welche die geringste Distanz zum LSB besitzt, wird gespeichert.
3. Jedes Bit nach der gespeicherten Position (in Richtung des MSB) wird invertiert.

Dieses Verfahren ist identisch mit der häufig verwendeten Methode zum Bilden des Zweier-Komplements, bei welcher die Bitfolge zuerst invertiert, und dann eine 1 addiert wird (Abschnitt 2.2). Der Vorteil des hier vorgestellten Verfahrens ist, dass nicht addiert werden muss. Der Nachteil ist, dass das Verfahren in Hardware implementiert einen hohen Fan-In, genau genommen vom Grad n für das MSB, besitzt.

Komponenten

Die Umsetzung des Verfahrens erfolgt mit $n - 1$ XOR-Gattern und $n - 2$ OR-Gattern, wobei die OR-Gatter jeweils 2 bis $n - 1$ Eingänge besitzen. Bei einer Wortbreite von n Bit berechnet sich die Anzahl der Gatter mit $n - 1$ für die XOR-Gatter und $\frac{(n-1) \cdot n}{2}$ für die OR-Gatter, wenn man davon ausgeht, dass nur Gatter mit zwei Eingängen zur Verfügung stehen. Im Vergleich zum gängigen Verfahren (invertieren und addieren) benötigt dieses mehr Platz, ist aber schneller, da der längste Signallaufweg durch exakt n Gatter führt (vgl. Länge des TCP von CRA: $n \cdot 3$ Gatter).

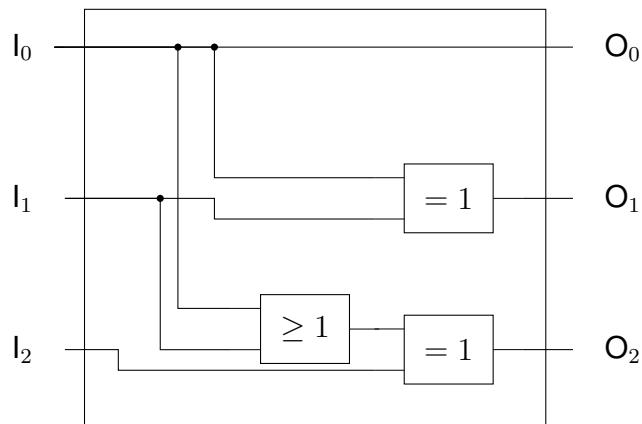


Abbildung 5.1: 3-Bit-Zweier-Komplement-Konverter

Verwendung

Die Schnittstelle benötigt nur die Angabe der Wortbreite (`g_size`), sowie einen je n Bit breiten Ein- und Ausgang (`p_in` und `p_out`).

```
1 GENERIC (
2     g_size  : integer := 7
3 );
4 PORT (
5     p_in   : in  std_logic_vector(g_size DOWNTO 0);
6     p_out  : out std_logic_vector(g_size DOWNTO 0)
7 );
8
```

Quellcode 5.1: Zweier-Komplement-Konverter-Schnittstelle

5.2.2 Absolut-Funktion

Funktion

Die Absolut-Funktion im mathematischen Sinne konvertiert eine negative Zahl (im Zweier-Komplement) in eine positive Zahl. Hierfür wird die Zahl abhängig von ihrem Vorzeichen direkt durch die Komponente geleitet (MSB = 0) oder einmal konvertiert (MSB = 1) und danach weitergeleitet.

Komponenten

Sie besteht aus einem TCC und einem Multiplexer. Die Zahl wird in jedem Fall (MSB = 1 und MSB = 0) einmal konvertiert. Abhängig vom Vorzeichen leitet der Multiplexer direkt die Eingabe oder die konvertierte Zahl durch (Abbildung 5.2).

Verwendung

Die Schnittstelle ist die gleiche wie die des TCC.

5.2.3 Divider-Slice

Funktion

Das Divider-Slice (DS) führt eine Subtraktion durch und entscheidet, ob die Differenz oder der ursprüngliche Minuend ausgegeben werden soll. Welcher der beiden

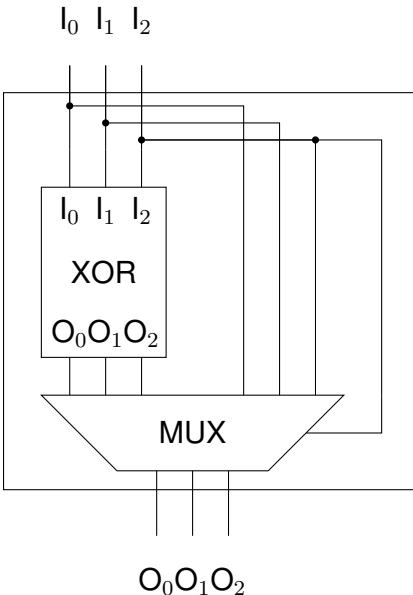


Abbildung 5.2: 3-Bit-Absolut-Funktion

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_in  : in  std_logic_vector(g_size DOWNTO 0);
6      p_out : out std_logic_vector(g_size DOWNTO 0)
7  );
8

```

Quellcode 5.2: Absolut-Funktion Schnittstelle

Werte am Ausgang angelegt wird, entscheidet sich durch einen ein Bit breiten Eingang M , an welchem das Übertrags-Bit des DS angeschlossen wird. Aus diesem DS kann später der RD aufgebaut werden.

Komponenten

Es besteht aus einem n Bit breiten CRA und einem Multiplexer, welcher die Ausgabe steuert. Der CRA wird dabei zum Subtrahieren verwendet, indem der zweite Operand vorher invertiert wird und der Eingang des Übertrags-Bit konstant auf 1 gesetzt wird (gängige Methode zum Bilden des Zweier-Komplement).

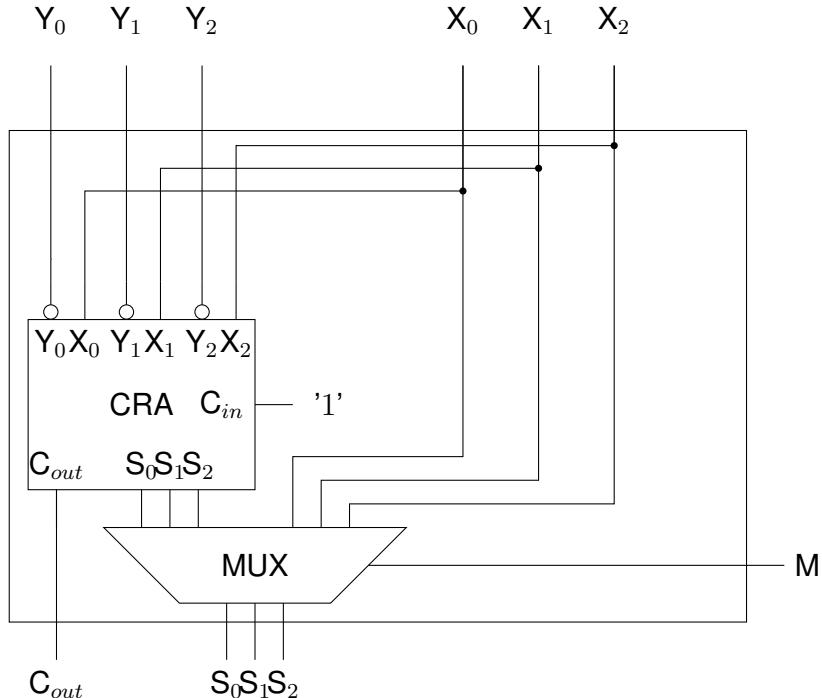


Abbildung 5.3: 3-Bit-Divider-Slice

Verwendung

Die Schnittstelle des DS besitzt, wie oben erwähnt, den Eingang p_m_in , um den Multiplexer zu steuern, und den Ausgang p_c_out , welcher dem Vorzeichen der Differenz entspricht. p_c_out wird an den p_m_in -Eingang geleitet. Außerdem wird das Übertrags-Bit später dazu verwendet, um den Quotienten der Division zu bilden. Weiterhin besitzt die Schnittstelle einen n Bit breiten Ausgang p_remain , welcher an den Eingang $p_dividend$ der nächsten Stufe angeschlossen wird. Zuletzt gibt es noch den Eingang $p_divisor$, welcher bei jeder Stufe die gleiche Eingabe erhält, nämlich den Divisor der eigentlichen Division. $p_dividend$ und $p_divisor$ besitzen ebenfalls die Wortbreite n .

5.2.4 Restoring-Divider

Funktion

Der RD ist der eigentliche Kern des Dividierers, er setzt das in Abschnitt 5.1 beschriebene Verfahren um, indem er bei einer Wortbreite von n Bit n DSs miteinander

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_m_in      : in  std_logic := '0';
6      p_dividend   : in  std_logic_vector(g_size DOWNTO 0);
7      p_divisor    : in  std_logic_vector(g_size DOWNTO 0);
8      p_c_out      : out std_logic;
9      p_remain     : out std_logic_vector(g_size DOWNTO 0)
10 );
11

```

Quellcode 5.3: Divider-Slice-Schnittstelle

verbindet. Dabei erhalten alle Stufen denselben Divisor als Eingang. Als Dividend erhalten die Stufen jeweils die Bit $n - 1$ bis 1 vom Ausgang der übergeordneten Stufe $i - 1$. Diese $n - 1$ Bit beginnen beim MSB der Stufe i . Das LSB entspricht dann dem $(n - i)$ -ten Bit des anfänglichen Dividenden.

Die erste Stufe erhält für den Dividend als LSB das MSB des eigentlichen Dividen- den und alle höherwertigen Bits erhalten eine 0 als Eingabe.

Komponenten

Für die Implementierung des RD werden bei einer Wortbreite von n Bit n DSs und n NOT-Gatter benötigt. Die DSs werden dabei wie oben erklärt miteinander verbun- den. Die NOT-Gatter werden benötigt, um die einzelnen Übertrags-Bit der Stufen zu invertieren. Diese invertierten Übertrags-Bit ergeben dann den Quotienten der Di- vision, wobei das LSB dem Übertrag der untersten Stufe entspricht.

Das Ergebnis der letzten Stufe entspricht außerdem dem Rest der Division und wird ebenfalls mit ausgegeben (Abbildung 5.4).

Verwendung

Die Schnittstelle benötigt neben der generischen Wortbreite `g_size` noch den Divi- denden `p_dividend` und Divisor `p_divisor`. Als Ausgang besitzt sie den Quotien- ten `p_result` und Rest `p_remain`.

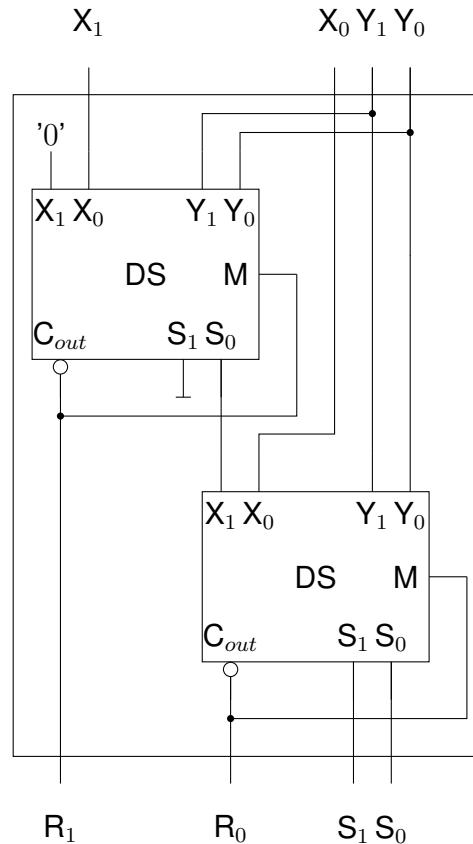


Abbildung 5.4: 4-Bit-Restoring-Divider

```

1  GENERIC (
2    g_size : integer := 7
3  );
4  PORT (
5    p_dividend  : in  std_logic_vector(g_size DOWNTO 0);
6    p_divisor   : in  std_logic_vector(g_size DOWNTO 0);
7    p_remain    : out std_logic_vector(g_size DOWNTO 0);
8    p_result   : out std_logic_vector(g_size DOWNTO 0)
9  );
10 
```

Quellcode 5.4: Restoring-Divider-Schnittstelle

5.2.5 Dividierer

Funktion

Der RD wird in eine übergeordneten Divisionskomponente eingebettet, um das Dividieren von vorzeichenbehafteten Zahlen zu ermöglichen. Hierfür wird intern im n Bit breiten Dividierer ein $n+1$ Bit breiter RD verwendet, welcher stets positive Zahlen dividiert. Der Divisor und Dividend werden durch eine Absolut-Funktion-Komponente geleitet, um sicherzustellen, dass die Eingaben an den RD positiv sind. Danach erfolgt eine Division der beiden Operanden. Der Quotient und der Rest der Division werden anschließend mit einem TCC in eine negative Zahl umgewandelt. Abhängig von den Vorzeichen der beiden Operanden bei der Eingabe, werden nun als Quotient und Rest die jeweils positiven oder negativen Werte ausgegeben. Bei der Ausgabe wird das MSB abgeschnitten, was die Zahl nicht verfälscht, da die Eingabe ebenfalls nur mit n Bit erfolgte und mit einer Vorzeichenerweiterung auf $n+1$ Bit hochgerechnet wurde.

Der RD benötigt intern ein Bit mehr, da sonst nicht der komplette Zahlenbereich abgedeckt wird: Soll zum Beispiel $\frac{-128}{64} = -2$ mit einer Wortbreite von 8 Bit berechnet werden, so würde die Berechnung nach der Absolut-Funktion $\frac{256}{64} = 4$ entsprechen, nach einer Vorzeichenanpassung erhält man entsprechend einen Quotienten von -4 , was offensichtlich falsch ist. Wird intern mit einem Bit mehr gerechnet, kann -128 zu 128 konvertiert und der Quotient richtig berechnet werden.

Komponenten

Der Dividierer besteht aus zwei ABS, zwei TCC, einem RD und zwei Multiplexern zur Auswahl der richtigen Ausgabe (Selektor). Außerdem wird ein weiteres Signal benötigt, welches eine vorzeichenbehaftete Division einleitet. Die Abbildung 5.5 zeigt den Dividierer mit seinen einzelnen Komponenten, es wurde hier ein Block-Schaltbild verwendet, da die Darstellung sonst zu komplex und damit zu unübersichtlich wäre. Die roten (gestrichelten) Leitungen in der Abbildung entsprechen $n+1$ Bit, die schwarzen n Bit und die gepunktete einem Bit.

Verwendung

Die Schnittstelle des Dividierers ist bis auf ein Bit identisch mit der des RD. Das zusätzliche Bit p_{sgnd} dient dazu, um zwischen einer vorzeichenbehafteten und

5 Dividierer

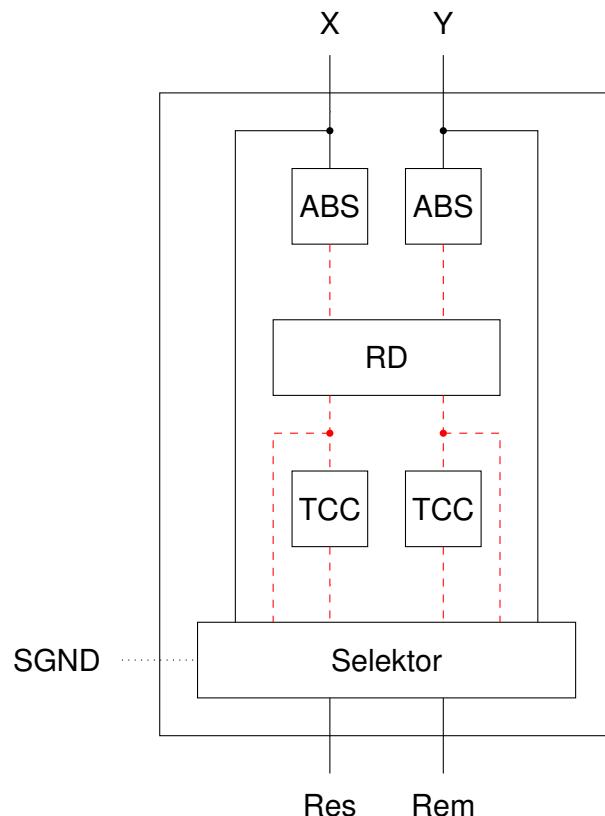


Abbildung 5.5: n -Bit-Divider

vorzeichenlosen Division umschalten zu können.

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_sgnd      : in  std_logic;
6      p_dividend   : in  std_logic_vector(g_size DOWNTO 0);
7      p_divisor    : in  std_logic_vector(g_size DOWNTO 0);
8      p_remain     : out std_logic_vector(g_size DOWNTO 0);
9      p_result     : out std_logic_vector(g_size DOWNTO 0)
10 );
11

```

Quellcode 5.5: Dividierer Schnittstelle

5.3 Bewertung

5.3.1 Fläche

Die Fläche entspricht der Anzahl der verwendeten Gatter in Abhängigkeit der Wortbreite.

Komponente	Zusammensetzung	Formel [Gatter]
TCC	$(n + 1) \cdot \text{XOR} + \frac{(n-1) \cdot n}{2} \cdot \text{OR}$	$\frac{n^2+n-2}{2}$
ABS	MUX + TCC	$\frac{n^2+7n-2}{2}$
DS	$n \cdot \text{NOT} + \text{MUX} + \text{CRA}$	$9n$
RD	$n \cdot \text{NOT} + n \cdot \text{DS}$	$9n^2 + n$
Dividierer	$2 \cdot \text{ABS} + 2 \cdot \text{TCC} + 2 \cdot \text{AND} + \text{MUX} + \text{RD}$	$11n^2 + 33n + 8$

Tabelle 5.1: Flächenverbrauch Dividierer

5.3.2 Time-Critical-Path

Der längste Signalweg des Dividierers setzt sich aus dem TCP der Absolut-Funktion, des Multiplexers und des RD zusammen. Der TCP durch den TCC entspricht der Erzeugung des MSB und wird mit $\lceil \log_2(n - 1) \rceil$ Gattern berechnet. Der TCP durch die Absolut-Funktion entspricht dem des TCC erweitert um einen Multiplexer, also $\lceil \log_2(n - 1) \rceil + 2$ Gatter. Der TCP durch den Multiplexer des Dividierers entspricht ebenfalls 2 Gattern. Zuletzt fehlt noch der TCP des RD, dieser ist der des Übertrags-Bits durch die CRA, welche zum Subtrahieren verwendet werden. Dieser TCP entspricht also $(n + 1) \cdot 3(n + 1) = 3n^2 + 6n + 3$ Gattern. Insgesamt ergibt sich damit ein TCP von $(\lceil \log_2(n - 1) \rceil + 2) + (3n^2 + 6n + 3) + (\lceil \log_2(n - 1) \rceil) + 2 = 2 \cdot \lceil \log_2(n - 1) \rceil + 3n^2 + 6n + 7$ Gattern.

Erneut unter der Annahme eines 32 Bit Dividierers und einer Gatterlaufzeit von 20ps, errechnet sich damit eine maximale Taktfrequenz von $\frac{1s}{3281 \cdot 20 \text{ ps}} = 15 \text{ MHz}$. Insgesamt ist der Dividierer damit langsamer als der Multiplizierer. Die maximale Taktfrequenz lässt sich noch steigern, indem man die einfachen Carry-Ripple-Adder der DS mit schnelleren Carry-Select-Adder austauscht.

6 Komparator

Ein Komparator dient dazu, zwei Operanden miteinander zu vergleichen. Mit ihm ist es möglich, zu entscheiden, welcher der beiden Operanden größer ist oder ob sie die gleiche Zahl repräsentieren. Dies funktioniert sowohl mit vorzeichenlosen als auch vorzeichenbehafteten Zahlen. Im Falle des vorzeichenbehafteten Falles muss lediglich bei einem Vergleich zweier unterschiedlicher Zahlen das Vorzeichen beachtet werden.

6.1 Funktionsweise

Zum Einsatz in der ALU kommt ein Tree-Comparator (TC), welcher aus einzelnen Zwei-Bit-Komparatoren, als Baumstruktur, aufgebaut wird. Dies ist möglich, da das Vergleichen der Bits von zwei Operanden parallel durchgeführt werden kann. Zum Vergleich: bei der Addition funktioniert das nicht, da stets der Übertrag beachtet werden muss. Die einzelnen Komparatoren können somit als Baum miteinander verschalten werden, um eine größere Wortbreite als zwei Bit zu erhalten. Der Vorteil dieser Baumstruktur liegt in der Laufzeit sowie in der Kompaktheit des synthetisierten Komparators [2][5].

6.2 Aufbau

6.2.1 Komparator

Funktion

Der Zwei-Bit Komparator Vergleicht zwei vorzeichenlose 2-Bit-Operanden miteinander und besitzt zwei Ausgänge. Die Ausgänge G und L geben an, welcher der

Operanden größer ist oder ob sie gleich sind. Die Interpretation von G (engl. Greater) und L (engl. Less) ist in Tabelle 6.1 dargestellt. Die logischen Gleichungen von

G	L	Interpretation
0	0	Beide Operanden sind gleich
0	1	Operand 2 ist größer als Operand 1
1	0	Operand 1 ist größer als Operand 2
1	1	Zustand nicht möglich

Tabelle 6.1: Komparator Interpretation

G und L lauten wie folgt:

$$G = (x_1 \wedge \neg y_1) \vee (x_1 \wedge x_0 \wedge \neg y_0) \vee (\neg y_1 \wedge x_0 \wedge \neg y_0)$$

$$L = (\neg x_1 \wedge y_1) \vee (\neg x_1 \wedge \neg x_0 \wedge y_0) \vee (y_1 \wedge x_0 \wedge \neg y_0)$$

Ihre Herleitung resultiert aus der Wahrheitstabelle zu finden in Anhang A.2.

Komponenten

Der Zwei-Bit-Komparator implementiert die angegebenen logischen Verknüpfungen mit Hilfe von Basis-Gattern (Abbildung 6.1).

Er ist damit eine atomare Komponente.

Verwendung

Die Schnittstelle der Komponente besitzt wie in den beiden Formeln angegeben, zwei 2-Bit-Operanden, welche auf vier 1-Bit-Signale aufgeteilt sind ($p_x_0, p_x_1, p_y_0, p_y_1$) sowie die beiden Ausgänge p_g und p_l .

6.2.2 Comparing-Slice

Funktion

Das Comparing-Slice (CS) reiht mehrere Zwei-Bit-Komparatoren nebeneinander, welche nicht miteinander verbunden sind. Das CS dient lediglich der einfacheren und übersichtlicheren Implementation in VHDL.

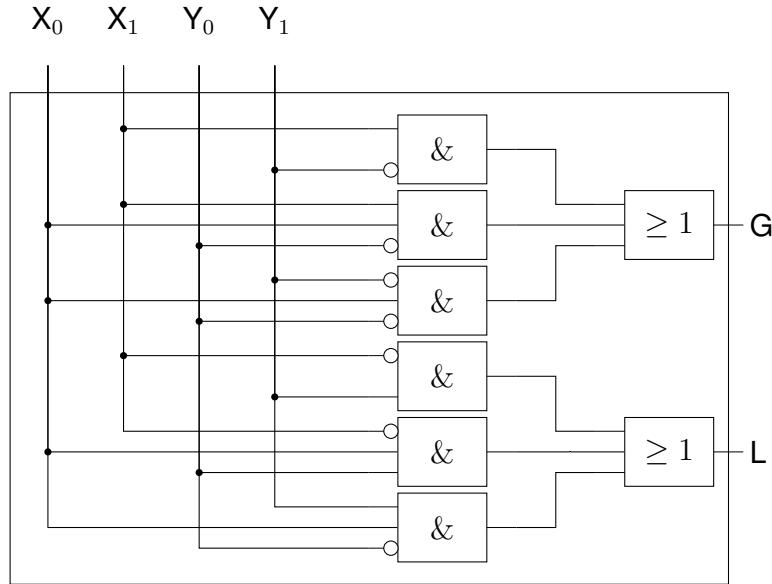


Abbildung 6.1: Komparator

```

1 PORT (
2   p_x_0 : in  std_logic;
3   p_y_0 : in  std_logic;
4   p_x_1 : in  std_logic;
5   p_y_1 : in  std_logic;
6   p_g   : out std_logic;
7   p_l   : out std_logic
8 );
9

```

Quellcode 6.1: Komparator Schnittstelle

Komponenten

Die Komponente besteht wie bereits erwähnt, aus mehreren Zwei-Bit Komparatoren. Bei einer angegebenen Wortbreite von n Bit, erzeugt die Komponente $\frac{n}{2}$ Zwei-Bit Komparatoren (Abbildung 6.2).

Verwendung

Die Schnittstelle ist im wesentlichen die selbe wie beim Zwei-Bit Komparator, aber mit dem Unterschied dass die Operanden als Vektoren zusammengefasst sind und

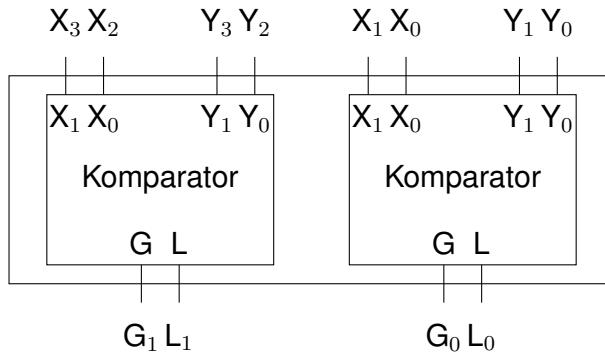


Abbildung 6.2: 4 Bit Comparing-Slice

eine breite von n Bit besitzen (p_op_1 , p_op_2). Die Wortbreite der Ausgabe von p_g und p_l entspricht nur der halben Wortbreite. Außerdem muss die Wortbreite g_size angegeben werden.

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_op_1  : in std_logic_vector(g_size DOWNTO 0);
6      p_op_2  : in std_logic_vector(g_size DOWNTO 0);
7      p_g     :
8          out std_logic_vector((g_size + 1) / 2 - 1 DOWNTO 0);
9      p_l     :
10         out std_logic_vector((g_size + 1) / 2 - 1 DOWNTO 0)
11  );
12

```

Quellcode 6.2: Comparing-Slice Schnittstelle

6.2.3 Tree-Comparator

Funktion

Der TC verwendet das vorher implementierte CS um damit einen Baum, bestehend aus Zwei-Bit Komparatoren, auf zu bauen. Hierfür werden beginnend bei der Wortbreite n , mehrere CSs mit jeweils der halben Wortbreite des übergeordneten CS

hintereinander geschaltet, bis die letzte Stufe nur noch die Wortbreite 2 besitzt und nur noch jeweils 1 Bit für G und L ausgibt. Die Ergebnisse der letzten Stufe werden im Falle eines vorzeichenbehafteten Vergleiches anschließend an die beiden Vorzeichen der Operanden angepasst. Der Vorzeichentest findet statt, in dem die beiden MSB der Operanden verglichen werden. Ist ein Operand positiv und der andere negativ, so ist offensichtlich, welcher Operand größer ist; G und L werden entsprechend angepasst. Sind beide Vorzeichen gleich oder es findet ein vorzeichenloser Vergleich statt, wird das vom TC errechnete Ergebnis direkt ausgegeben.

Komponenten

Die Komponente besteht bei einer Wortbreite von $n \in \{p|2^q, q \in \mathbb{N}\}$, aus $\log_2(n)$ Stufen, implementiert durch jeweils ein CS, gefolgt vom Vorzeichentest (Abbildung 6.3).

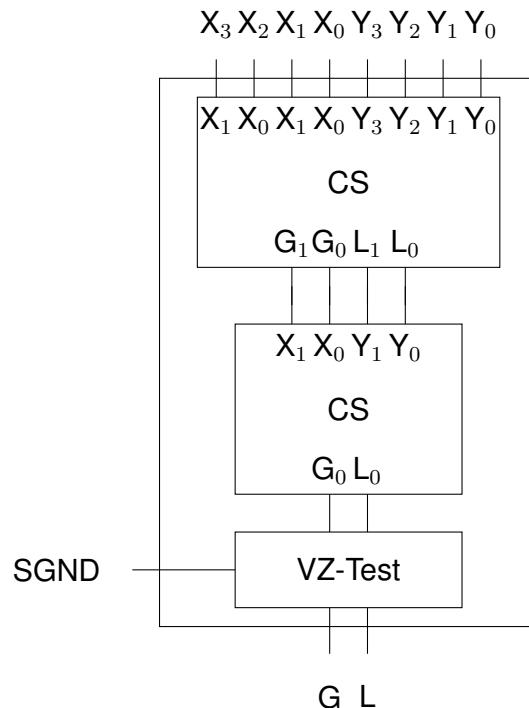


Abbildung 6.3: 4 Bit Tree-Comparator

Verwendung

Die Schnittstelle erhält als Eingabe wieder die beiden Operanden p_op_1 und p_op_2, sowie die Angabe, ob ein vorzeichenloser oder vorzeichenbehafteter Vergleich durchgeführt werden soll (p_sgnd). Als Ausgabe besitzt er nur noch jeweils ein Bit p_g und p_l. Ebenfalls wie beim CS muss die Wortbreite angegeben werden.

```
1 GENERIC (
2     g_size : integer := 7
3 );
4 PORT (
5     p_op_1  : in  std_logic_vector(g_size DOWNTO 0);
6     p_op_2  : in  std_logic_vector(g_size DOWNTO 0);
7     p_sgnd  : in  std_logic;
8     p_g     : out std_logic;
9     p_l     : out std_logic
10 );
11
```

Quellcode 6.3: Tree-Comparator Schnittstelle

6.2.4 Word-Comparator

Funktion

Der Word-Comparator (WC) verwendet den TC und erweitert diesen um einen Befehlssatz, mit dem der Vergleich ausgewählt werden kann, welcher zwischen den beiden Operanden erfolgen soll. Je nach gewähltem Befehl werden G und L des TC diesem entsprechend interpretiert. Falls der Vergleich wahr ist, wird das Ausgabe Bit des WC auf 1 gesetzt, ansonsten bleibt es auf 0. Die Befehle sind in Tabelle 6.2 aufgeführt.

Komponenten

Die Komponente besteht aus einem TC und einem zweiten Teil (CMD-Test), welcher den Befehlssatz der Komponente implementiert (Abbildung 6.4). Die schwarzen Leitungen entsprechen 1 Bit und die rote (gestrichelte) CMD-Leitung für den Befehlssatz 3 Bit.

Befehl	Vergleich
000	$op_1 > op_2$
001	$op_1 < op_2$
010	$op_1 \geq op_2$
011	$op_1 \leq op_2$
100	$op_1 = op_2$

Tabelle 6.2: Befehlssatz des Word-Comparator

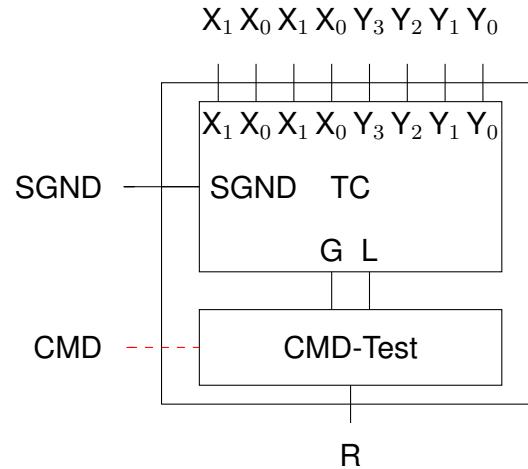


Abbildung 6.4: 4 Bit Word-Comparator

Verwendung

Die Schnittstelle entspricht der selben wie der des TC, allerdings erweitert um den Befehlsvektor CMD. Die Ausgabe der Komponente ist nur noch ein Bit, welches angibt, ob der Vergleich wahr oder falsch ist.

6.3 Bewertung

6.3.1 Fläche

Die Fläche entspricht der Anzahl der verwendeten Gatter in Abhängigkeit der Wortbreite.

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_op_1      : in  std_logic_vector(g_size DOWNTO 0);
6      p_op_2      : in  std_logic_vector(g_size DOWNTO 0);
7      p_cmd       : in  std_logic_vector(2 DOWNTO 0);
8      p_sgnd     : in  std_logic;
9      p_result    : out std_logic
10 );
11

```

Quellcode 6.4: Word-Comparator Schnittstelle

Komponente	Zusammensetzung	Formel
Komparator	Gatter zur Berechnung von G und L	14
CS	$n \cdot$ Komparator	$n \cdot 14$
TC	$\log_2(n) \cdot CS +$ Vorzeichentest ¹	$\sum_{i=0}^{\log_2(n)-1} (2^i \cdot 14) + 22$
WC	TC + CMD-Test	$\sum_{i=0}^{\log_2(n)-1} (2^i \cdot 14) + 60$

Tabelle 6.3: Flächenverbrauch Word-Comparator

6.3.2 Time-Critical-Path

Der TCP berechnet sich durch die Tiefe des TC, des Vorzeichentests und der Ergebnisberechnung abhängig vom gewählten Befehl. Die Anzahl der Gatter, welche durch den Baum durchlaufen werden müssen, errechnen sich mit $\log_2(n) \cdot 5$ Gattern. Für den Vorzeichentest müssen 5 und für die Befehlsauswertung 7 Gatter durchlaufen werden. Damit ergibt sich ein TCP von $\log_2(n) \cdot 5 + 12$ Gattern. Bei einer Wortbreite von 32 Bit und einer Gatterlaufzeit von 20 ps ergibt sich eine maximale Taktfrequenz von $\frac{1}{265 \text{ ps}} = 3773 \text{ MHz}$.

¹Größe der CS halbiert sich auf jeder Stufe

7 Rotieren und Schieben

Eine weitere wichtige Komponente der ALU ist die, welche das Schieben und Rotieren von Bits ermöglicht. Die Einfachste Art des Schiebens und Rotierens in Hardware, entspricht dem Neuverdrahten von Leitungen. Diese bringt allerdings den Nachteil mit sich, dass nicht eingestellt werden kann, um wie viele Positionen geschoben bzw. rotiert werden soll. Deshalb soll ein Barrel-Shifter eingesetzt werden, durch welchen es möglich ist den ersten Operand um m Bits schieben/rotieren zu können, wobei m durch den zweiten Operand festgelegt wird. Dabei sei zu beachten, dass der zweite Operand nicht in seiner vollen Wortbreite verwendet wird, da das Schieben/Rotieren eines n Bit breiten Wortes sich ab $m = n$ wiederholt.

7.1 Funktionsweise

Der Barrel-Shifter (BS) besteht aus $p = \log_2(n)$ einzelnen Stufen (Shifting-Slice (SS)), wobei jede den Operand um $2^i, i \in [0, p]$ stellen nach links oder rechts schieben/rotieren kann. Die einzelnen SSs werden dabei mit Hilfe von Multiplexern¹ umgesetzt. Die Stufen schieben/rotieren das Wort entsprechend dem zweiten Operanden: Ist Bit i des Operanden gesetzt wird Stufe i dazu angewiesen das Wort um 2^i Bits zu schieben/rotieren. Alle Stufen hintereinander geschaltet ergeben damit den BS[2].

¹Implementiert durch when-Statements in VHDL

7.2 Aufbau

7.2.1 Shifting-Slice

Funktion

Das SS implementiert, wie in 7.1 bereits beschrieben, die einzelnen Stufen des BS. Die Implementierung selbst ist ein Neuverdrahten des Eingabeoperanden mit Hilfe von Multiplexern. Weiterhin existieren innerhalb der Komponente zwei weitere Leitungen l_{in} und r_{in} , welche beim Schieben/Rotieren das neue MSB bzw. LSB angeben. Die Eingabe für l_{in} und r_{in} hängt von der aktuellen Operation ab (Tabelle 7.1).

Befehl	l_{in}	r_{in}
logisches Linksschieben	×	0
logisches Rechtsschieben	0	×
arithmetisches Linksschieben	×	0
arithmetisches Rechtsschieben	MSB	×
nach links rotieren	×	<i>MSB</i>
nach rechts rotieren	<i>LSB</i>	×

Tabelle 7.1: Bestimmung von l_{in} und r_{in}

Komponenten

Das SS besteht nur aus Multiplexern. Die rote (gestrichelte) CMD Leitung in Abbildung 7.1 und 7.2 besitzt eine Breite von zwei Bit und steuert die Multiplexer der gewünschten Operation entsprechend an. Die beiden Leitungen l_{in} und r_{in} existieren wie bereits erwähnt nur intern und werden von außen mit den beiden Leitungen rot, ari und der Tabelle 7.1 angesteuert. Die Abbildungen wurden zum Zweck der Übersichtlichkeit vereinfacht.

Verwendung

Die Schnittstelle für die Verwendung besitzt die beiden oben erwähnten Eingänge p_arith und p_rotate. Der zwei Bit breite Eingang p_ctrl steuert direkt die

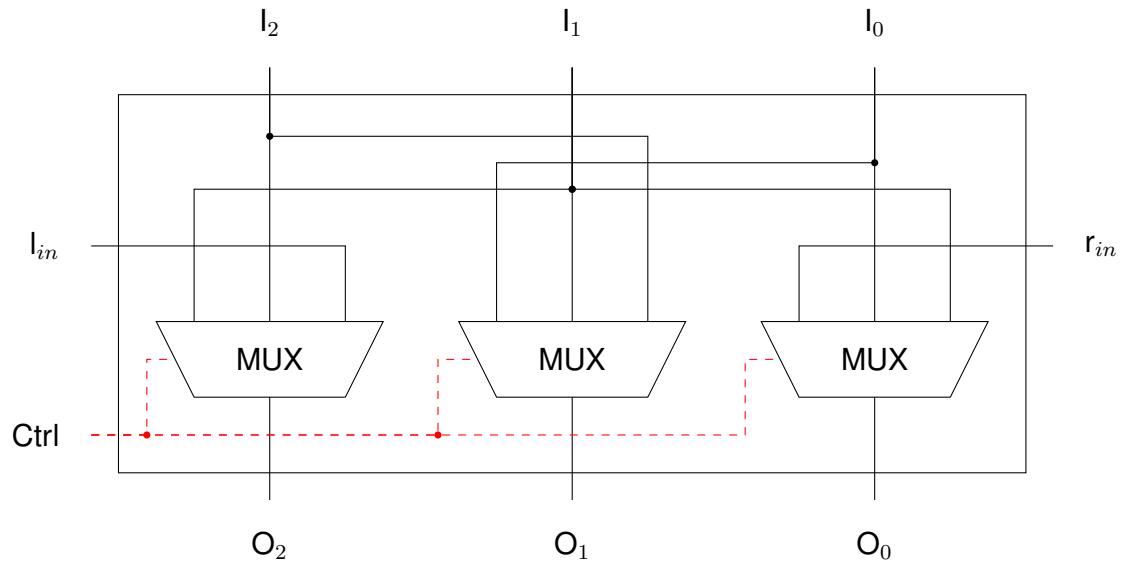


Abbildung 7.1: 3-Bit Shifting-Slice (Stufe 1)

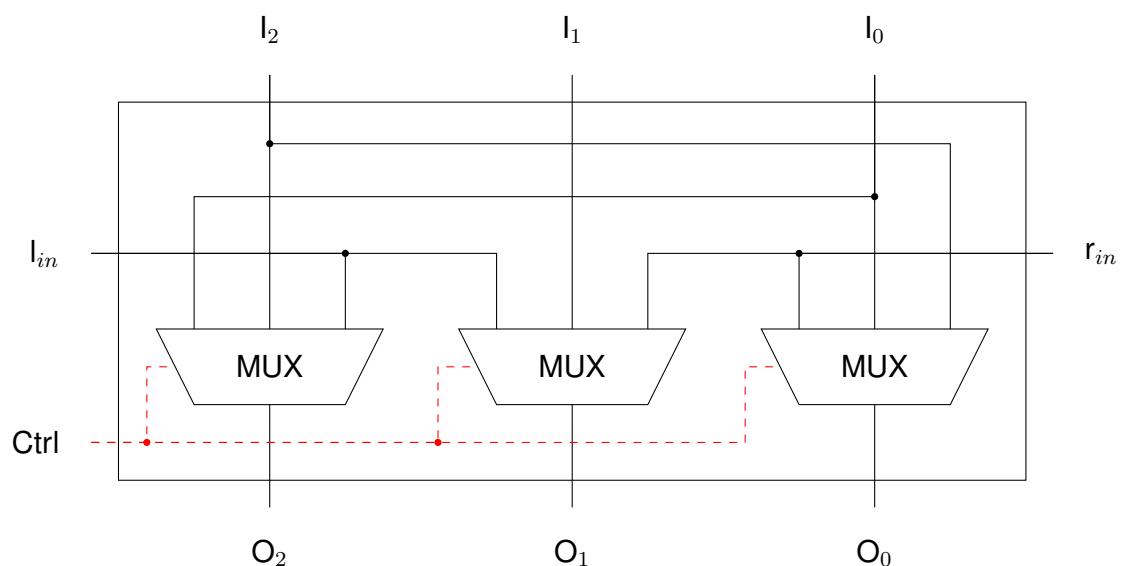


Abbildung 7.2: 3-Bit Shifting-Slice (Stufe 2)

Multiplexer an. Neben des Operanden p_op und dem Ergebnis p_result muss noch die Wortbreite g_size und der Stufenindex g_step angegeben werden. Der Stufenindex entspricht dabei der Position des SS innerhalb des BS und wird dazu verwendet, um die Verdrahtung in der Komponente entsprechend ihrer Position zu

bestimmten.

```

1  GENERIC (
2      g_size : integer := 7;
3      g_step : integer := 1
4  );
5  PORT (
6      p_arith    : in  std_logic;
7      p_rotate   : in  std_logic;
8      p_ctrl     : in  std_logic_vector(1 DOWNTO 0);
9      p_op       : in  std_logic_vector(g_size DOWNTO 0);
10     p_result   : out std_logic_vector(g_size DOWNTO 0)
11 );
12

```

Quellcode 7.1: Shifting-Slice Schnittstelle

7.2.2 Barrel-Shifter

Funktion

Der BS ist die eigentliche Komponente zum schieben und rotieren. Er schaltet nun $\log_2(n)$ SSs entsprechend der Wortbreite n hintereinander. Weiterhin werden hier für jede Stufe der aktuelle Befehl, welcher die Multiplexer der SSs ansteuert, maskiert. Für die Maskierung wird der Befehl, welcher an die Stufe i weitergegeben wird, zuerst mit Bit i des zweiten Operanden Bitweise mit einem AND verknüpft. Ist bei einem Schiebe- oder Rotiervorgang nun das Bit i des zweiten Operanden 0, wird der Befehl für die Stufe i ebenfalls auf 00 gesetzt und die Stufe leitet dann die Eingabe direkt an den Ausgang weiter ohne sie zu verändern.

Komponenten

Die in Abbildung 7.3 abgebildete Komponente besteht wie bereits erwähnt aus mehreren SSs und AND-Gattern zum maskieren des Befehls. Die Leitungen ARI und ROT werden dabei direkt zu den einzelnen Stufen weitergeleitet. Die roten (gestrichelten) Leitungen entsprechen zwei Bit und werden Bitweise mit Y_0 bzw. Y_1 verknüpft.

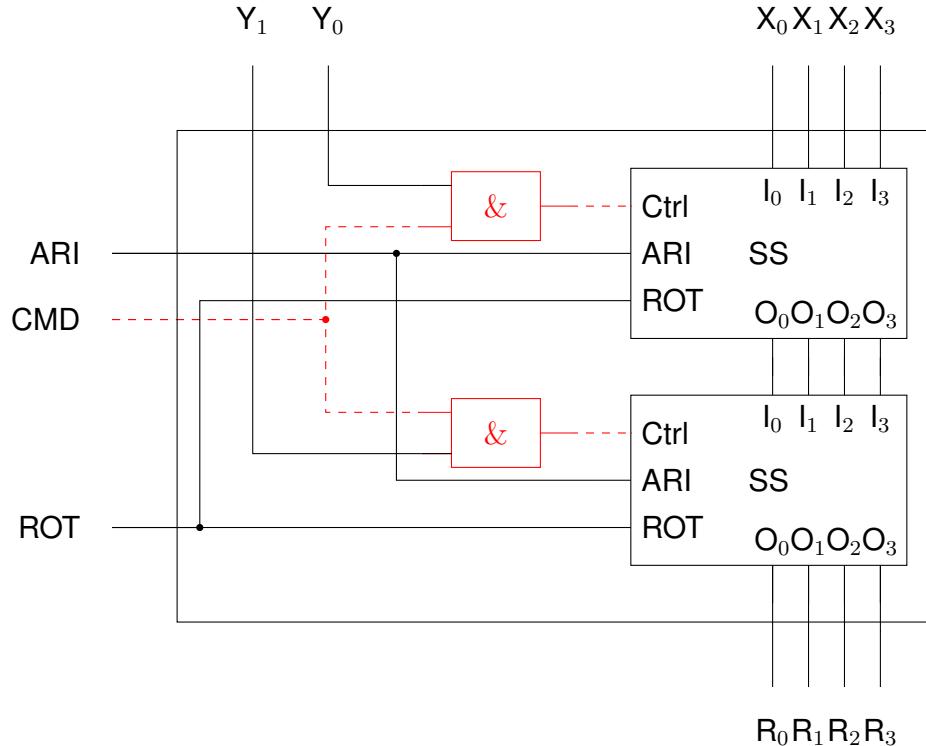


Abbildung 7.3: 4 Bit Barrel-Shifter

Verwendung

Die Schnittstelle des BS ist fast identisch zu der des SS. Die `p_ctrl`-Leitung wurde durch die `p_cmd`-Leitung ersetzt und der zweite Operand `p_op_2` hinzugefügt. Außerdem muss nur noch die Wortbreite angegeben werden. Durch die drei Eingänge `p_cmd`, `p_arith` und `p_rotate` kann die Komponente nun entsprechend Tabelle 7.2 angesteuert werden.

Befehl	<code>p_cmd</code>	<code>p_arith</code>	<code>p_rotate</code>
logisches Linksschieben	01	0	0
logisches Rechtsschieben	01	0	0
arithmetisches Linksschieben	10	1	0
arithmetisches Rechtsschieben	10	1	0
nach links rotieren	01	×	1
nach rechts rotieren	10	×	1
nop	00	×	×

Tabelle 7.2: Befehlssatz Barrel-Shifter

```

1  GENERIC (
2      g_size : integer := 7
3  );
4  PORT (
5      p_cmd      : in  std_logic_vector(1 DOWNTO 0);
6      p_arith    : in  std_logic;
7      p_rotate   : in  std_logic;
8      p_op_1     : in  std_logic_vector(g_size DOWNTO 0);
9      p_op_2     : in  std_logic_vector(g_size DOWNTO 0);
10     p_result   : out std_logic_vector(g_size DOWNTO 0)
11 );
12

```

Quellcode 7.2: Barrel-Shifter Schnittstelle

7.3 Bewertung

7.3.1 Fläche

Die Fläche entspricht der Anzahl der verwendeten Gatter in Abhängigkeit der Wortbreite.

Komponente	Zusammensetzung	Formel
SS	n 4-zu-1 Multiplexer	$9n$
BS	$\log_2(n) \cdot SS + 2 \log_2(n) \cdot \text{AND-Gatter}$	$(9n + 2) \log_2(n)$

Tabelle 7.3: Flächenverbrauch Word-Comparator

7.3.2 Time-Critical-Path

Der TCP durch den BS ergibt sich aus der Anzahl der Stufen, aus denen er besteht. Die Latenz innerhalb einer Stufe wird nur durch den Multiplexer verursacht. Somit ergibt sich ein TCP von $2 \log_2(n)$ Gattern. Bei einer Wortbreite von $n = 32$ Bit und einer Gatterlaufzeit von 20 ps entspricht dies einer Verzögerung von 200 ps und damit einer maximal möglichen Taktfrequenz von $\frac{1}{200\text{ ps}} = 5\text{ GHz}$.

8 Logische Verknüpfungen

Logik auf Bit-Ebene ist essentiell für eine CPU und damit auch für eine ALU. Der Aufbau einer Logikkomponente ist aber im Vergleich zu anderen Komponenten der ALU trivial.

8.1 Funktionsweise

Die entwickelte ALU stellt 4 logische Operationen zur Verfügung, welche durch den Bit-Manipulator (BM) implementiert werden. Er besteht aus jeweils n einzelnen AND-, OR-, XOR- und NOT-Gattern, welche jeweils die beiden Operanden miteinander verknüpfen (bei der NOT-Verknüpfung nur ein Operand). Die Ausgänge der einzelnen Verknüpfungen werden danach mit Hilfe eines Multiplexers an den Ausgang der Komponente weiter geleitet. Der Multiplexer besitzt dabei zwei Bit als Adresseingang, da er vier Eingänge besitzt, entsprechend den vier logischen Verknüpfungen.

8.1.1 Komponenten

Der BM besteht wie oben erwähnt aus $4 \cdot n$ Gattern für die logischen Verknüpfungen und einem 4-zu-1 Multiplexer (Abbildung 8.1). Die rote (gestrichelte) Leitung (CMD) besitzt eine Breite von zwei Bit.

8.1.2 Verwendung

Die Schnittstelle des BM bekommt die zu verwendende Wortbreite `g_size`, die beiden Operanden `p_op_1` und `p_op_2` sowie einen Steuerbefehl `p_cmd` übergeben. Als Ausgang besitzt er nur das Ergebnis der logischen Verknüpfung, abhängig vom gewählten Befehl, `p_result`. Der Befehlssatz ist in Tabelle 8.1 dargestellt.

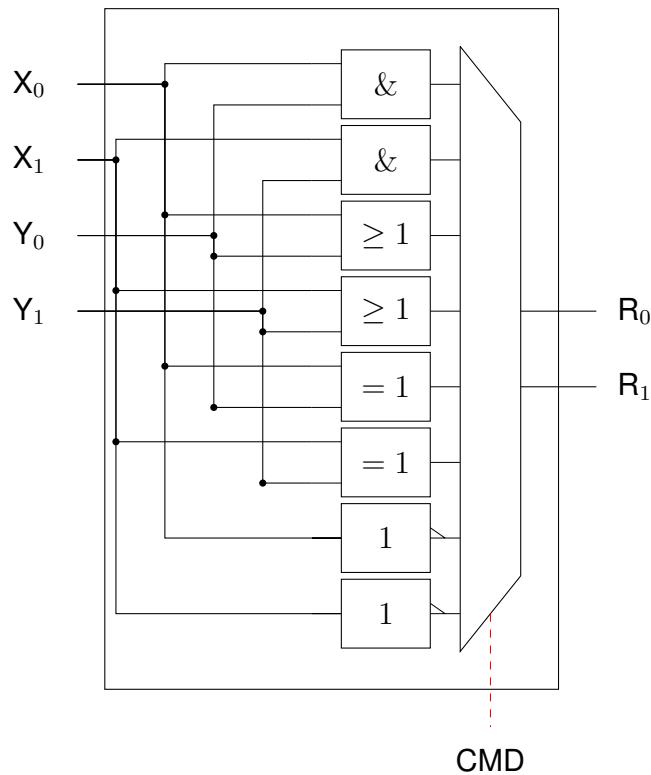


Abbildung 8.1: 2 Bit BM

Befehl	Verknüpfung
00	AND
01	OR
10	XOR
11	NOT

Tabelle 8.1: Befehlssatz Bit-Manipulator

8.2 Bewertung

8.2.1 Fläche

Die Fläche berechnet sich beim BM aus den n Gattern je Verknüpfung und dem Multiplexer. Da der Multiplexer 4 Eingänge besitzt, benötigt dieser $n \cdot 11$ Gatter. Insgesamt ergibt sich damit eine Fläche von $15n$ Gattern bei einer Wortbreite von n Bit.

```
1 GENERIC (
2     g_size : integer := 7
3 );
4 PORT (
5     p_op_1      : in std_logic_vector(g_size DOWNTO 0);
6     p_op_2      : in std_logic_vector(g_size DOWNTO 0);
7     p_cmd       : in std_logic_vector(1 DOWNTO 0);
8     p_result    : out std_logic_vector(g_size DOWNTO 0)
9 );
10
```

Quellcode 8.1: Halbaddierer Schnittstelle

8.2.2 Time-Critical-Path

Der TCP des BM entspricht genau 5 Gattern, eines davon von der jeweiligen logischen Verknüpfung und die restlichen vier durch den Multiplexer.

9 Validierung

Damit sichergestellt ist, dass die ALU keine Rechenfehler macht, wurde auf empirische Tests mit Hilfe von ModelSim zurückgegriffen. Hierfür wurde für jede Komponente der ALU (mit Ausnahme der Logik-Operatoren) ein sogenannter Testbench erstellt, welcher die Komponente in allen Verwendungsmöglichkeiten testet. Die Tests umfassen demnach auch die Unterscheidung von vorzeichenlosen und vorzeichenbehafteten Berechnungen. Alle Komponenten wurden mit einer Wortbreite von 8 Bit getestet ($2 \cdot 4$ Bit beim CSA).

9.1 Testaufbau

Die Tests der verschiedenen Komponenten werden wie bereits erwähnt, mit Hilfe von ModelSim durchgeführt. Der Testablauf wird anhand von Pseudocode erklärt (Quellcode 9.1).

Hierfür wird die zu testende VHDL-Komponente in den Testbench eingebunden. Für jeden Eingabeoperand existiert eine For-Schleife, welche ineinander verschachtelt werden (Zeile 1 und 2). Diese erzeugen für die Eingabeoperanden alle möglichen Kombinationen, sodass jeder Fall im Test abgedeckt wird. Im inneren der For-Schleifen wird die erzeugte Kombination an die zu testende Komponente gereicht, welche das Ergebnis berechnet (Zeile 3). Neben dem Resultat der Komponente, wird außerdem ein Vergleichswert mit Hilfe der Rechenoperatoren aus `ieee.numeric_std.all` berechnet (Zeile 4). Die beiden berechneten Werte werden anschließend miteinander verglichen (Zeile 5). Weichen die errechneten Ergebnisse voneinander ab, wird der laufende Testbench abgebrochen (Zeile 6). Die Kommutativität der Operationen wird bei den Tests bewusst ignoriert, um sicherzustellen, dass die Komponenten beim vertauschen der Operanden dennoch die richtigen Ergebnisse berechnen. Zum Beispiel wird bei der Addition sowohl $1 + 2 = 3$ als auch $2 + 1 = 3$ getestet und verifiziert.

Wird eine Komponente mit dem Testbench getestet und verläuft vollständig bis zum Ende, wurden alle Berechnungen korrekt durchgeführt. Die Komponente ist damit

```

1 FOR op1 FROM 0 TO 256
2   FOR op2 FROM 0 TO 256
3     result = comp(op1, op2);
4     control = ieee.numeric_std.comp(op1, op2);
5     IF (result != control)
6       RETURN -1;
7   END
8 END
9
10

```

Quellcode 9.1: Pseudocode Komponententest

validiert und kann in der ALU verwendet werden.

Mit ModelSim ist es beim Ausführen eines Testbenches auch möglich, die Signale genauer zu betrachten, welche erzeugt werden (vgl. Abschnitt 9.2).

9.2 Beispiel

In Abbildung 9.1 ist ein Ausschnitt aus dem Test zur Multiplikation zu sehen: tb_op1 und tb_op2 sind hierbei die Eingabeoperanden, tb_add die kostenlose Addition des VQM, tb_result das Ergebnis der Komponente und control_result der Vergleichswert. tb_sgnd gibt an ob es sich um eine vorzeichenlose oder vorzeichenbehaftete Operation handelt.

Zu erkennen ist, dass im gezeigten Ausschnitt das Ergebnis (tb_result) und sein

+◆ /tb/control_result	2997	2970	2943	2916	2889	2862	2835	2808	2781
+◆ /tb/tb_add	0								
+◆ /tb/tb_op1	-27								
+◆ /tb/tb_op2	-111	-110	-109	-108	-107	-106	-105	-104	-103
+◆ /tb/tb_result	2997	2970	2943	2916	2889	2862	2835	2808	2781
◆ /tb/tb_sgnd									

Abbildung 9.1: Ausschnitt Multiplikationstest

Kontrollwert (control_result) übereinstimmen. Die Komponente zur Multiplikation wurde in dem gezeigten Ausschnitt validiert. Ebenfalls zu sehen ist, dass es sich um einen vorzeichenbehafteten Test handelt, da beide Operanden negativ sind und

tb_sgnd den Wert 1 besitzt. Die kostenlose Addition (tb_add) beträgt in dem gezeigten Beispiel 0, wird aber im gesamten Testbench berücksichtigt und ebenfalls validiert.

10 Verwendung

10.1 Übersicht und Schnittstelle

Die ALU-Komponente fügt alle entwickelten Teilkomponenten, welche in den Kapiteln 3 bis 8 eingeführt und erklärt wurden, mit einem Multiplexer zusammen. Außerdem übernimmt sie die Übersetzung zwischen dem internen und externen Befehlsatz (Abbildung 10.1). Die Schnittstelle der ALU besitzt neben den trivialen Signalen und Angaben (`g_size`, `p_op_1`, `p_op_2`, `p_result_lo`, `p_result_hi`) noch einen Eingabevektor `p_cmd` für den Befehl, ein Ausgabebit `p_ovflw` für das Überlauf-Bit des Addierers und ein Ausgabebit `p_cmp_f`, welches angibt ob der Vergleich des Komparators Wahr oder Falsch ist (Quellcode 10.1).

10.2 Befehlssatz

Die implementierte ALU beherrscht insgesamt 28 verschiedene Operationen bzw. Vergleiche, welche mit dem entsprechenden Befehl aus dem Befehlssatz verwendet werden können. Es wird dabei zwischen dem internen und externen Befehlssatz unterschieden. Der interne Befehlssatz steuert die Komponenten innerhalb der ALU. Dieser wird nach außen durch eine Tabelle ersetzt, wodurch der externe Befehlssatz erzeugt wird. Die ALU verwendet also intern einen anderen Befehlssatz, als der, mit dem man sie ansteuern kann. Das Ganze bietet den Vorteil, dass durch das Ändern der Einträge in der Befehlssatzübersetzung, die ALU an jeden beliebigen Opcode angepasst werden kann.

10.2.1 Interner Befehlssatz

Da die ALU über keinen Takt verfügt und damit asynchron arbeitet, berechnen alle Teilkomponenten stets das Ergebnis der beiden Eingabeoperanden, unabhängig

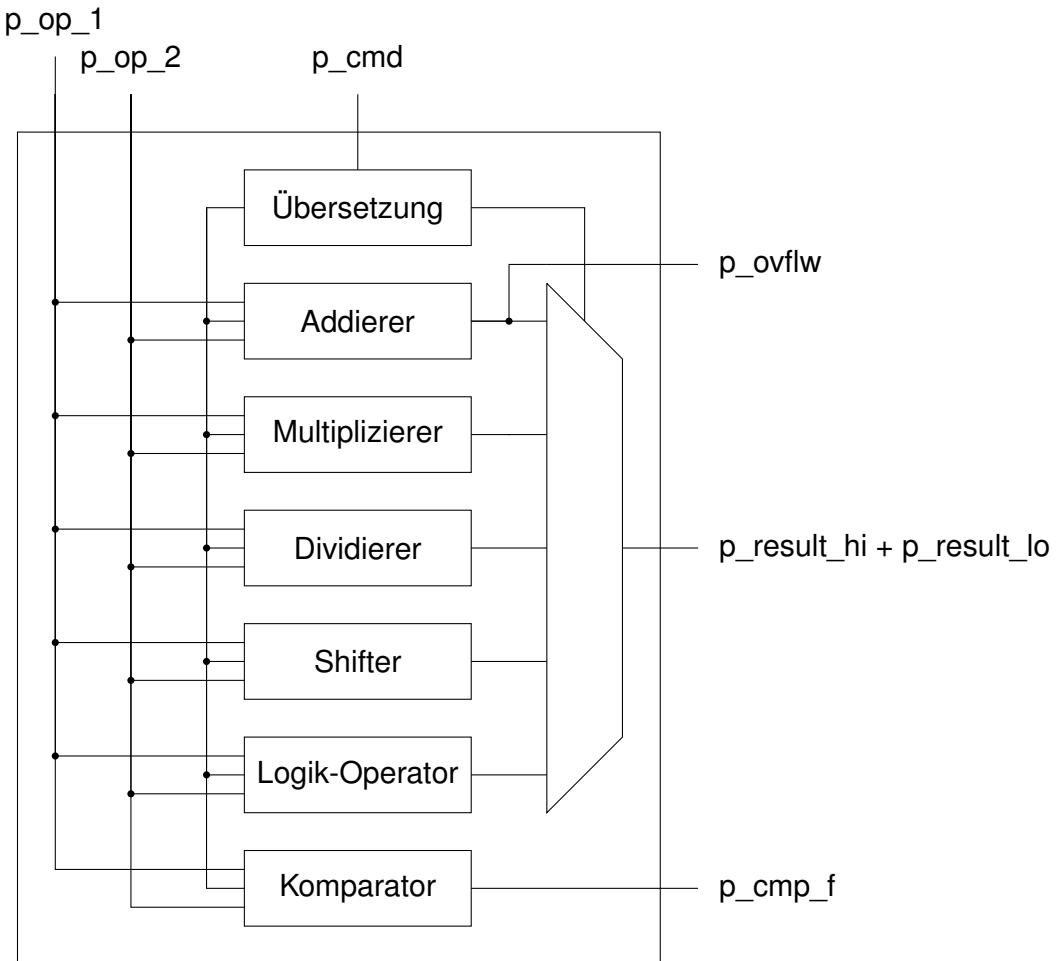


Abbildung 10.1: Blockschaltbild ALU

von der gewählten Operation. Damit das richtige Ergebnis am Ausgang der ALU ausgegeben wird, steuert ein Teil des internen Befehls den Multiplexer an, welcher die Ausgänge aller Teilkomponenten als Eingang besitzt. Die zweite Aufgabe des internen Befehlssatzes ist es, die Teilkomponenten entsprechend dem gewählten Befehl zu konfigurieren (Bsp.: den Komparator auf kleiner-gleich vergleichen lassen).

Der interne Befehlssatz setzt sich aus 8 Bit zusammen (Tabelle 10.1).

Das MSB weist die ALU dazu an, mit oder ohne Vorzeichen zu rechnen. Die drei niedrigsten Bits teilen der ALU (bzw. dem Multiplexer) mit, welches Ergebnis der Teilkomponenten an den Ausgang weitergegeben wird. Die übrigen vier Bit in der Mitte sind dazu da um die Komponenten zu konfigurieren. Beim Shifter wird angegeben, ob es sich um einen arithmetischen oder logischen Shift handelt (R, A)

```

1  GENERIC (
2      g_size  : integer := 7
3  );
4  PORT (
5      p_op_1      : in  std_logic_vector(g_size DOWNTO 0);
6      p_op_2      : in  std_logic_vector(g_size DOWNTO 0);
7      p_cmd       : in  std_logic_vector(4 DOWNTO 0);
8      p_ovflw     : out std_logic;
9      p_cmp_f     : out std_logic;
10     p_result_lo : out std_logic_vector(g_size DOWNTO 0);
11     p_result_hi : out std_logic_vector(g_size DOWNTO 0)
12 );
13

```

Quellcode 10.1: ALU Schnittstelle

Komponente	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Allgemein	VZ	CMD ₃	CMD ₂	CMD ₁	CMD ₀	Mux ₂	Mux ₁	Mux ₀
Addierer	VZ	×	×	×	×	0	0	0
Multiplizierer	VZ	×	×	×	×	0	0	1
Dividierer	VZ	×	×	×	×	0	1	0
Shifter	×	R	A	C	C	0	1	1
Komparator	VZ	×	C	C	C	1	0	0
Logik-Op.	×	×	×	C	C	1	0	1

Tabelle 10.1: Interner Befehlssatz

und in welche Richtung der Shift durchgeführt wird (C, C). Beim Komparator wird lediglich angegeben, um was für einen Vergleich es sich handelt (C, C, C). Der Logik-Operator bekommt ebenfalls mitgeteilt, was für eine logische Verknüpfung durchgeführt werden soll (C, C).

A, C und R entsprechen einzelnen Bit aus Tabelle 10.1. Die genauen Belegungen davon können den einzelnen Komponenten entnommen werden. Der komplette Befehlssatz inklusive Opcode findet sich im Anhang A.

10.2.2 Externer Befehlssatz

Damit nun der externe Befehlssatz verwendet und angepasst werden kann, wird dieser über eine when-Struktur direkt in den internen Befehlssatz übersetzt (Quell-

code 10.2). Auf der linken Seite der when-Struktur steht der interne Befehlssatz,

```
1 "01001011" when p_cmd = "01001" else -- rl
2 "00010011" when p_cmd = "01010" else -- srl
3 "00001011" when p_cmd = "01011" else -- sll
4 "00110011" when p_cmd = "01100" else -- sra
5 "000101011" when p_cmd = "01101" else -- sla
6
```

Quellcode 10.2: Ausschnitt Übersetzungstabelle

auf der rechten Seite der externe. Durch ändern der Binärvektoren auf der rechten Seite kann nun der externe Befehlssatz beliebig angepasst werden. Der externe Befehlssatz besteht zudem nur aus 5 Bits, da diese ausreichend sind um 28 verschiedene Befehle zu kodieren.

10.3 Demo

Damit die ALU manuell getestet werden kann wurde ein Demoprogramm für das Altera DE2 Board implementiert. Die Eingabe erfolgt über die Schalter und Taster des Boards. Die Ausgabe über Sieben-Segment-Anzeigen, welche die Operanden und das Ergebnis zur Basis 16 anzeigen (Abbildung 10.2).

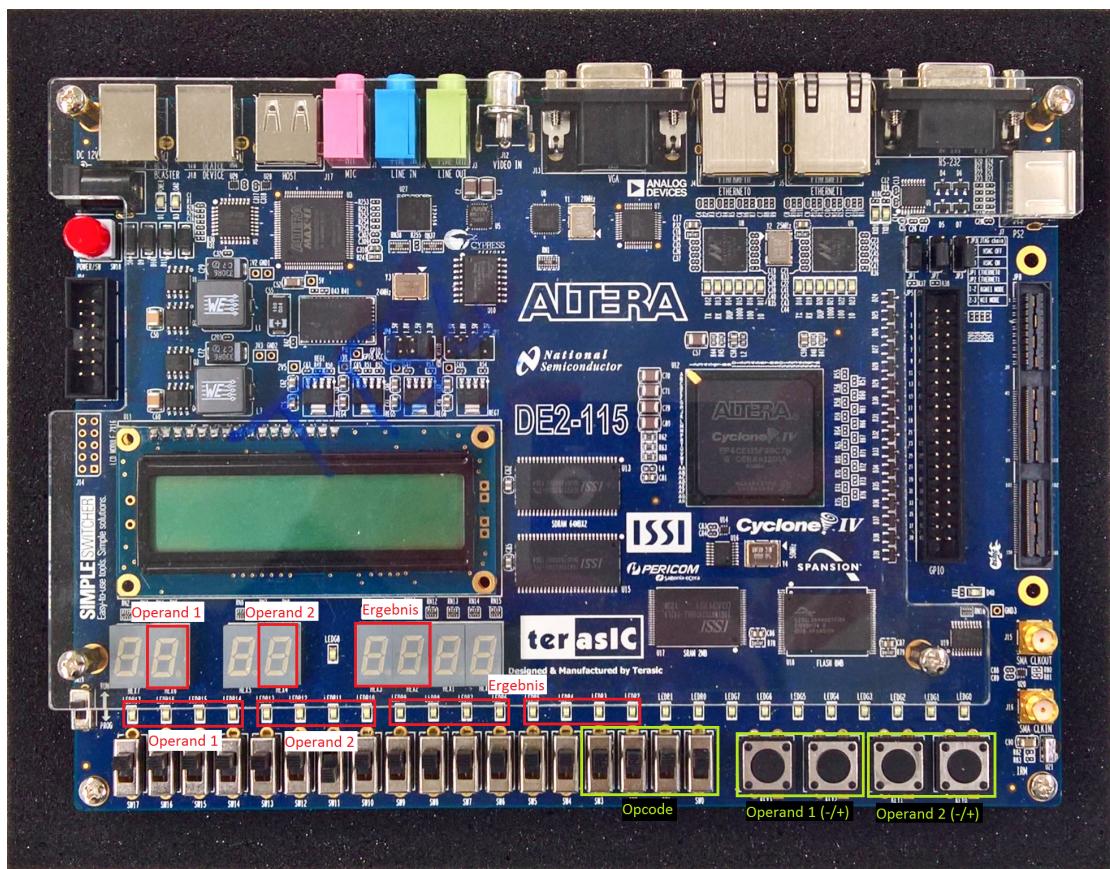


Abbildung 10.2: Altera DE2 Board

11 Fazit und Ausblick

Die entwickelte ALU beherrscht die wichtigsten und gängigen Rechenoperationen und ist damit sowohl für die Lehre als auch für den Einsatz in einer Rechenarchitektur verwendbar. Sie wurde zur besseren Übersicht und Lesbarkeit sauber strukturiert und ist in ihrer Wortbreite skalierbar. Damit sichergestellt werden kann, dass sie keine Rechenfehler besitzt, wurden ausgiebige Tests zur Validierung durchgeführt. Die Teilkomponenten sind komplett austauschbar und können auch einzeln verwendet werden.

Es wurde bei der Entwicklung darauf geachtet, dass die ALU als Folgeprojekt mit einem Pipelining ausgestattet werden kann. Weitere Folgeprojekte könnten das Messen der Latenz und des Platzbedarfs nach der Synthese sein. Eine Erweiterung der ALU um neue Komponenten wie zum Beispiel Fließkomma-Operationen oder BCD-Operationen sind ebenfalls möglich.

A Anhang

A.1 Befehlssatz

Die Opcodes sind in ihrer Binärdarstellung angegeben. Die Befehle mit dem Suffix `_u` sind vorzeichenlose Operationen, die ohne Suffix sind vorzeichenbehaftete Operationen (Tabelle A.1).

Kürzel	Befehl	Interner Opcode	Externer Opcode
add	Addition	10000000	00000
add_u	Addition	00000000	00001
sub	Subtraktion	10001000	00010
sub_u	Subtraktion	00001000	00011
mul	Multiplikation	10000001	00100
mul_u	Multiplikation	00000001	00101
div	Division	10000010	00110
div_u	Division	00000010	00111
rr	Rotate Right	01010011	01000
rl	Rotate Left	01001011	01001
srl	Shift Right Logic	00010011	01010
sll	Shift Left Logic	00001011	01011
sra	Shift Right Arithmetic	00110011	01100
sla	Shift Left Arithmetic	00101011	01101
and	Und-Verknüpfung	00000100	01110
or	Oder-Verknüpfung	00001100	01111
xor	Exklusive Oder-Verknüpfung	00010100	10000
not	Invertieren	00011100	10001
gt	Größer	10000101	10010
gt_u	Größer	00000101	10011
lt	Kleiner	10001101	10100
lt_u	Kleiner	00001101	10101
geq	Größer gleich	10010101	10110
geq_u	Größer gleich	00010101	10111
leq	Kleiner gleich	10011101	11000
leq_u	Kleiner gleich	00011101	11001
eq	Gleich	10100101	11010
eq_u	Gleich	00100101	11011

Tabelle A.1: Befehlssatz

A.2 2-Bit-Komparator

X_1	Y_1	X_0	Y_0	G	L
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	0	0

Tabelle A.2: Wahrheitstabelle 2-Bit-Komparator

Abbildungsverzeichnis

3.1 Halbaddierer	8
3.2 Volladdierer	9
3.3 3-Bit-Carry-Ripple-Adder	10
3.4 Carry-Select-Adder-Block mit 3-Bit Wortbreite	12
3.5 6-Bit-CSA	15
4.1 Multiplier-Base-Cell	20
4.2 2 Bit Multiplier-Slice	22
4.3 2-Bit-Vier-Quadranten-Multiplizierer	23
5.1 3-Bit-Zweier-Komplement-Konverter	28
5.2 3-Bit-Absolut-Funktion	30
5.3 3-Bit-Divider-Slice	31
5.4 4-Bit-Restoring-Divider	33
5.5 n -Bit-Divider	35
6.1 Komparator	39
6.2 4 Bit Comparing-Slice	40
6.3 4 Bit Tree-Comparator	41
6.4 4 Bit Word-Comparator	43
7.1 3-Bit Shifting-Slice (Stufe 1)	47
7.2 3-Bit Shifting-Slice (Stufe 2)	47
7.3 4 Bit Barrel-Shifter	49
8.1 2 Bit BM	52
9.1 Ausschnitt Multiplikationstest	55
10.1 Blockschaltbild ALU	58
10.2 Altera DE2 Board	61

Tabellenverzeichnis

1.1	Nomenklatur VHDL-Code	2
3.1	Flächenverbrauch Carry-Select-Adder	17
4.1	Flächenverbrauch Vier-Quadranten-Multiplizierer	25
5.1	Flächenverbrauch Dividierer	36
6.1	Komparator Interpretation	38
6.2	Befehlssatz des Word-Comparator	43
6.3	Flächenverbrauch Word-Comparator	44
7.1	Bestimmung von l_{in} und r_{in}	46
7.2	Befehlssatz Barrel-Shifter	49
7.3	Flächenverbrauch Word-Comparator	50
8.1	Befehlssatz Bit-Manipulator	52
10.1	Interner Befehlssatz	59
A.1	Befehlssatz	64
A.2	Wahrheitstabelle 2-Bit-Komparator	65

Literaturverzeichnis

- [1] *Overflow Detection for Adders.* <http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Comb/overflow.html>, Juni 2003. – [Abrufdatum: November 2016]
- [2] GAJSKI, Daniel D.: *Principles of Digital Design*. London : Prentice Hall, 1997. – ISBN 978–0–133–01144–9
- [3] KOREN, Israel: *Computer Arithmetic Algorithms*. 2. Auflage. Justus-Liebig-Universität Gießen : Taylor & Francis, 2001. – ISBN 978–1–568–81160–4
- [4] PARHAMI, Behrooz: *Computer Arithmetic - Algorithms and Hardware Designs*. 2. Auflage. New York : Oxford University Press, 2010. – ISBN 978–0–195–32848–6
- [5] ROTHERMEL, Prof. Dr.-Ing. A.: Entwurf integrierter Systeme / Institut für Mikroelektronik - Universität Ulm. Sommersemester 2015. – Vorlesungsskript

Glossar

Altera DE2 Board Entwicklungsboard von der Firma terasIC welches auf einem Cyclone-FPGA der Firma Altera basiert. 1, 60, 66

Arithmetic-Logical-Unit Die ALU ist der zentrale Rechenkern einer CPU.

C++ objektorientierte Hochsprache zum erzeugen von Maschinencode. 10

digitaler Signalprozessor Ein DSP dient der Verarbeitung und Bearbeitung von digitalen Signalen. Er hat einen speziell dafür angepassten Befehlssatz und Zusatzkomponenten. 1

Fan-Out Der Fan-Out ist das selbe wie der Fan-In, aber vom Ausgang des Gatters betrachtet, welches andere Gattereingänge treiben muss. Genaueres siehe Abschnitt 2.3.

Fan-In Als Fan-In bezeichnet man die Anzahl der Gattereingänge, welche von einem Signal getrieben werden müssen. Ein hoher Fan-In verringert die maximale Taktfrequenz. Genaueres siehe Abschnitt 2.3. 28

Field-Programmable-Gate-Array Ein Logikbaustein welcher mit VHDL programmiert werden kann. Genaueres siehe Abschnitt 2.1. ii, 3, 69

Logikgatter Ein logisches Gatter ist ein elektronisches Bauteil, welches eine logische Verknüpfung (AND, OR, NOT, XOR, NAND, NOR) umsetzt..

Lookup-Table Register innerhalb eines FPGAs, welches eine logische Funktion beschreibt. 3

ModelSim ModelSim ist ein Softwarewerkzeug von der Firma Altera, welches das Simulieren und damit auch das Testen von VHDL-Code ermöglicht.

Opcode Ein Opcode entspricht einem Zahlenwert, welcher einen Befehl aus einem Befehlssatz repräsentiert. 57

Glossar

VHDL Hardwarebeschreibungssprache zum Programmieren von Field-Programmable-Gate-Arrays. 1, 2, 3, 5, 10, 38, 54, 67, 69

Zweier-Komplement Das Zweier-Komplement ist eine binäre Darstellungsform von negativen ganzen Zahlen. Genaueres siehe Abschnitt 2.2. 4, 12, 27, 28, 30

Akronyme

ABS Absolut-Funktion. iv, 27, 28, 29, 32, 34, 35, 36, 66

ALU Arithmetic-Logical-Unit. 1, 3, 37, 45, 51, 54, 57, 60, 62, 66, 69

BM Bit-Manipulator. 51, 52, 66, 67

BS Barrel-Shifter. v, 45, 46, 48, 49, 50, 66, 67

CPU Central-Processing-Unit. 1, 51, 69

CRA Carry-Ripple-Adder. ii, 7, 9, 10, 11, 12, 15, 16, 28, 30, 36, 66

CS Comparing-Slice. iv, 38, 39, 40, 41, 43, 66

CSA Carry-Select-Adder. iii, 7, 12, 13, 15, 16, 24, 36, 54, 66, 67

CSAB Carry-Select-Adder-Block. iii, 11, 12, 13, 15, 16, 66

DS Divider-Slice. iv, 29, 30, 31, 30, 32, 35, 36, 66

EQM Ein-Quadranten-Multiplizierer. 18, 19, 20, 23

HA Halbaddierer. ii, 7, 8, 9, 16, 51, 66

LSB Least Significant Bit. 3, 26, 27, 31, 32, 45

MBC Multiplier-Base-Cell. iii, 19, 20, 21, 24, 66

MS Multiplier-Slice. iii, 20, 21, 23, 24, 66

MSB Most Significant Bit. 3, 4, 12, 20, 21, 26, 28, 29, 31, 32, 36, 40, 45, 57

RD Restoring-Divider. iv, 26, 27, 29, 31, 32, 34, 35, 36, 66

Akronyme

SS Shifting-Slice. v, 45, 46, 48, 49, 66

TC Tree-Comparator. iv, 37, 40, 41, 42, 43, 44, 66

TCC Zweier-Komplement-Konverter. iii, 27, 28, 29, 32, 34, 35, 36, 66

TCP Time-Critical-Path. ii, iii, iv, v, 5, 6, 16, 24, 28, 36, 43, 44, 50, 52

VA Volladdierer. ii, 8, 9, 10, 16, 19, 24, 66

VQM Vier-Quadranten-Multiplizierer. iii, 18, 19, 20, 21, 23, 24, 26, 55, 66, 67

WC Word-Comparator. iv, 42, 43, 49, 66, 67