Dokumentation zum Thema

FP10 Advanced Processor Design
G2 MAC-Unit

von Peter Konstantin Schöne, Thomas Alexander Hövelmann

Kurs:	FP10 Advanced Processor Design
Unterthema:	G2 MAC-Unit
Betreuer:	Mikail Yayla
Bearbeitungszeitraum:	Sommersemester 2022

Inhaltsverzeichnis

1 Einleitung

Ziel dieses Fachprojekts ist die Entwicklung und Analyse eines MAC-Unit. Eine MAC-Unit führt die Operation

$$acc := acc + a \cdot b$$

aus, bei welcher das Ergebnis der Multiplikation $a \cdot b$ auf einen Akkumulator acc addiert wird. Die Hardware-Unterstützung einer solchen Operation kann beispielsweise für die schnelle Berechnung von Matrix-Multiplikationen verwendet werden. Bei dem in diesem Fachprojekt vorgestellten Design wird die Addition durch einen Brent-Kung-Addierer und die Multiplikation durch einen Dadda-Multiplizierer realisiert. Um die Operation möglichst schnell durchführen zu können, wird beschrieben, wie die Addition des Akkumulators in die Multiplikation integriert werden kann. Um das Design zu bewerten, werden die Verzögerungen aller Bausteine in die Verzögerung von NAND-Gattern umgerechnet und die MAC-Unit wird mit einer MAC-Unit aus einem vergangenen Fachprojekt verglichen.
In Kapitel 2 werden einfache Grundbausteine beschrieben und in Kapitel 3 werden auf komplexere Funktionseinheiten eingegangen. Das Design der MAC-Unit wird in Kapitel 4 erläutert. Die MAC-Unit wird in Kapitel 5 evaluiert.

2 Grundbausteine

2.1 AND-Gatter

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt 2-Bit AND-Gatter im Dadda-Reduzierer (vgl. 3.3), im Brent-Kung-Addierer (vgl. 3.4) und im 2:1 Mux (vgl. 2.8).
Der Ausgang des 2-Bit AND-Gatters ist genau dann 1, wenn beide Eingänge 1 sind (Tabelle 1). Ein 2-Bit AND-Gatter kann durch zwei hintereinander geschaltete 2-Bit NAND-Gatter realisiert werden, wodurch sich eine Verzögerung von zwei NANDs ergibt.
Die Beschreibung in der zugehörigen VHDL-Datei „and2.vhdl“ ist funktional.

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 1: Berechnungstabelle für AND-Gatter.

2.2 NOT-Gatter

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt 1-Bit NOT-Gatter zur Overflow-Berechnung (vgl. 4) und im Dadda-Reduzierer zur Vorzeichenerweiterung (vgl.3.3).
Der Ausgang des Gatters, dessen funktionale Implementierung sich in der VHDL-Datei „not1.vhdl“ befindet, ist genau dann 1, wenn der Eingang 0 ist (Tabelle 2).
Ein 1-Bit NOT-Gatter kann durch ein NAND-Gatter realisiert werden, wodurch sich eine Verzögerung von einem NAND ergibt.

a	$\neg a$
0	1
1	0

Tabelle 2: Berechnungstabelle für NOT-Gatter.

2.3 OR-Gatter

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt 2-Bit AND-Gatter im Dadda-Reduzierer (vgl. 3.3), im Brent-Kung-Addierer (vgl. 3.4) und im 2:1 Mux (vgl. 2.8).
Der Ausgang des 2-Bit OR-Gatters ist genau dann 1, wenn mindestens einer der Eingänge 1 ist (Tabelle 3).
Ein OR-Gatter kann durch drei NAND-Gatter realisiert werden, von denen allerdings zwei parallel geschaltet sind, wodurch sich eine Verzögerung von zwei NANDs ergibt.
Die Beschreibung in der zugehörigen VHDL-Datei „or2.vhdl“ ist funktional.

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 3: Berechnungstabelle für OR-Gatter.

2.4 XOR-Gatter

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt 2-Bit und 3-Bit XOR-Gatter. Erstere werden im Halbaddierer 2.5 und im Komplementbilder 3.2 und letztere werde Brent-Kung-Addierer 3.4 verwendet.
Die Ausgabe eines 2-Bit Gatters ist genau dann 1, wenn genau einer der Eingänge 1 ist (Tabelle 4). Die Ausgabe eines 3-Bit Gatters ist genau dann 1, wenn entweder die XOR-Verknüpfung der ersten beiden Bits 1 und das dritte Bit 0 ist, oder die XOR-Verknüpfung der ersten beiden Bits 0 und das dritte Bit 1 ist (Tabelle 5).
Die Beschreibungen in den zugehörigen VHDL-Dateien „xor2.vhdl“ und „xor3.vhdl“ sind funktional.

2.5 Halbaddierer

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt Halbaddierer im Brent-Kung-Addierer 3.4, im Dadda-Reduzierer 3.3 und im Volladdierer 2.6.
Ein Halbaddierer berechnet die Summe zweier Bits und den möglichen Carry,

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 4: Berechnungstabelle für 2-Bit XOR-Gatter.

a	b	c	$a \oplus b \oplus c$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabelle 5: Berechnungstabelle für 3-Bit XOR-Gatter.

der genau dann entsteht, wenn beide eingehenden Bits 1 sind. Bezeichnen a und b die beiden eingehenden Bits, so ergeben sich das Summenbit s und der Carry c aus den folgenden Formeln:

$$s = a \oplus b,$$

$$c = a \vee b.$$

Der Addierer kann durch fünf NANDs realisiert werden, von denen zwei mal zwei parallel geschaltet sind, wodurch sich eine Verzögerung von drei NANDs ergibt.
Die Beschreibung in der zugehörigen VHDL-Datei „ha.vhdl“ ist funktional.

2.6 Volladdierer

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt Volladdierer im Dadda-Reduzierer 3.3 und im Komplementbilder 3.2.
Der Addierer berechnet in Abhängigkeit eines Carrys die Summe zweier Bits und den dadurch möglicherweise entstehenden Carry. Bezeichen a und b die

beiden eingehenden Bits und c_{in} den eingehenden Carry, so ergeben sich das Summenbit s und der ausgehende Carry c_{out} aus den folgenden Formeln:

$$s = a \oplus b \oplus c_{in},$$

$$c_{out} = (c_{in} \wedge (a \oplus b)) \vee (a \wedge b).$$

Ein Volladdierer kann durch neun NANDs realisiert werden, von denen drei mal zwei parallel geschaltet sind, wodurch sich eine Verzögerung von sechs NANDs ergibt.
Die Beschreibung in der zugehörigen VHDL-Datei „fa.vhdl“ ist funktional.

2.7 D-Flip-Flop

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt zwei D-Flip-Flops um zu speichern, ob es im zuletzt abgeschlossenen Berechnungszyklus zu einem Underflow oder einem Overflow gekommen ist (vgl. 4). Bei diesen D-Flip-Flops handelt es sich um gewöhnliche D-Flip-Flops mit je einem 1-Bit Dateneingang, einem 1-Bit Takteingang und einem 1-Bit Speicherausgang. Wenn am Takteingang eine steigende Flanke detektiert wird, wird der aktuell am Dateneingang anliegende Zustand gespeichert und liegt bis zur nächsten Änderung als neuer Zustand am Speicherausgang an. Die Beschreibung in der zugehörigen VHDL-Datei „d_ff.vhdl“ ist funktional. Da die D-Flip-Flops innerhalb des in dieser Projektarbeit entworfene MAC-Bausteins immer nur direkt beim Auftreten einer Taktflanke aktiv sind, können sie für die Abschätzung der maximalen Taktfrequenz für den MAC-Baustein vernachlässigt werden.

2.8 2:1-MUX

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt einen 2:1-Multiplexer um zu kontrollieren, ob bei der nächsten Taktflanke der Akkumulator-Wert im entsprechenden Register auf einen beliebigen aber festen Wert gesetzt werden soll oder ob stattdessen das Berechnungsergebnis des nächsten MAC-Zyklus im entsprechenden Register gespeichert werden soll (vgl. 4). Bei diesem 2:1-Multiplexer handelt es sich um einen gewöhnlichen 2:1-Multiplexer für 16-Bit Zahlen mit zwei 16-Bit Dateneingängen, einem 1-Bit Steuereingang und einem 16-Bit Datenausgang. Wenn am Steuereingang eine

logische 0 anliegt, dann hat der Datenausgang den Wert des ersten Dateneingangs, andernfalls hat der Datenausgang den Wert des zweiten Dateneingangs. Die Beschreibung in der zugehörigen VHDL-Datei „mux2_1.vhdl“ ist funktional. Unabhängig von der funktionalen Beschreibung, lässt sich ein 16-Bit 2:1-Multiplexer mithilfe eines NOT-Gatters, 32 AND-Gattern und 16 OR-Gattern realisieren, wobei der erste Dateneingang bitweise mit der Negation des Steuereingangs verundet wird, der zweite Dateneingang bitweise mit dem Wert des Steuereingangs verundet wird und die Ergebnisse dieser Verundungen bitweise, gemäß ihrer Stelligkeit, verodert werden. Unter Berücksichtigung der NAND-Verzögerungen von NOT-Gattern (vgl. 2.2), von AND-Gattern (vgl. 2.1) und OR-Gattern (vgl. 2.3) ergibt sich für einen 16-Bit 2:1-Multiplexer eine NAND-Verzögerung von 5 NANDs, da die Negation des Steuereingangs innerhalb des MAC-Bausteins parallel zu anderen deutlich langsameren Berechnungen erfolgen kann, ist die effektive Verzögerung jedoch lediglich 4 NANDs groß.

3 Komplexere Funktionseinheiten

3.1 PIPO-Register

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt für die Verwaltung der beiden Eingabe-Bitvektoren zwei 8-Bit PIPO-Register mit synchroner Set/Reset-Funktionalität (vgl. 4). Die Register haben einen 8-Bit Dateneingang, einen 1-Bit Set/Reset-Eingang, einen 1-Bit Taktgebereingang und einen 8-Bit Datenausgang. Beim Auftreten einer steigenden Flanke am Taktgebereingang, wird, falls am Set/Reset-Eingang eine 1 anliegt, der Wert 00000000 und ansonsten der Wert des Dateneingangs zum Datenausgang geleitet. Die Implementierung ist prozedural und befindet sich in der VHDL-Datei „pipo8.vhdl “.
Der Akkumulator des MAC-Bausteins wird durch ein 16-Bit PIPO-Register realisiert, dessen Wert asynchron zurückgesetzt werden kann (vgl. 4). Er hat einen 16-Bit Dateneingang, einen 1-Bit Reset-Eingang, einen 1-Bit Taktgebereingang und einen 16-Bit Datenausgang. Unabhängig vom Wert am Taktgebereingang wird der Wert am Datenausgang auf 0000000000000000 gesetzt, falls am Reset-Eingang eine 1 anliegt. Falls keine 1 anliegt, wird, sollte eine steigende Flanke vorliegen, der Wert des Dateneingangs zum Datenausgagn geleitet. Die Implementierung ist prozedural und befindet sich in der VHDL-Datei „pipo16.vhdl “.

3.2 Komplementbilder

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt einen 8-Bit Komplementbilder um im Zuge der Dadda-Reduktion das Zweierkomplement einer Binärzahl zu bilden, welches benötigt wird, damit der MAC-Baustein bezüglich des Rechnens mit Zweikomplementzahlen vollständig ist (vgl. 3.3). Das Zweierkomplement einer Binärzahl lässt sich bilden, indem von dieser Zahl die Zahl Eins abgezogen wird, was einer Addition mit der Zahl minus Eins entspricht (unter Vernachlässigung des Übertrags), und anschließend das Einerkompement des erhaltenen Ergebnisses gebildet wird, was einer bitweisen exklusiven Veroderung mit logisch 1 entspricht. Damit verfügt der Komplementbilder-Baustein über einen 8-Bit Dateneingang und einen 8-Bit Datenausgang, da innerhalb des vorliegenden MAC-Bausteins das Komplement nur unter bestimmten Voraussetzungen benötigt wird, verfügt der Komplementbilder über einen zusätzlichen 1-Bit Steuereingang. Der Wert, welcher

am Steuereingang anliegt, wird bitweise mit dem Ergebnis der beschriebenen Komplementbildung verundet, sodass bei Anliegen einer logischen 0 das Ergebnis Null ist (also als 8-Bit Repräsentation: 00000000) und andernfalls das Ergebnis der Komplementbildung am Datenausgang anliegt. Die Beschreibung der Funktion in der zugehörigen VHDL-Datei „complementer8.vhdl“ folgt dem in Abbildung 1 gezeigten Vorgehen. Unter Berücksichtigung der NAND-Verzögerungen vom 8-Bit Brent-Kung-Addierer (vgl. 3.4), von XOR-Gattern (vgl. 2.4) und AND-Gattern (vgl. 2.1) ergibt sich für einen 8-Bit Komplementbilder eine NAND-Verzögerung von 26 NANDs.

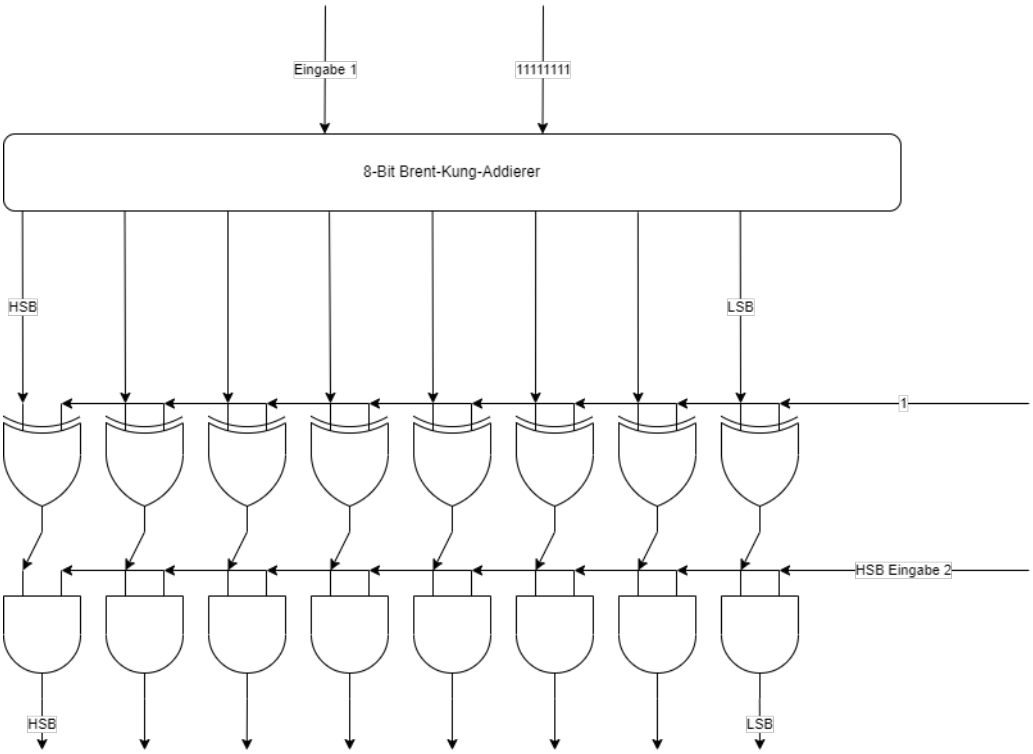

Abbildung 1: Layout des Komplementbilders; Dateneingang (Eingabe 1), Steuereingang (HSB Eingabe 2), Datenausgang (unten, highest significant bit (HSB) links bis least significant bit (LSB) rechts)

3.3 Dadda-Reduzierer

Der in dieser Projektarbeit entworfene MAC-Baustein nutzt einen Dadda-Reduzierer um die MAC-Operation zu vereinen (vgl. 4). Das Konzept des Dadda-Reduzierers wurde erstmals 1965 von Luigi Dadda für die Multiplikation zweier Binärzahlen vorgeschlagen[1]. Hierbei wird zunächst der erste Faktor bitweise mit jeder Bit-Stelle des zweiten Faktors gemäß der jeweiligen Stelligkeit verundet, wie es auch bei einer naiven Multiplikation der Fall wäre. Anschließend werden die so erhaltenen Binärzahlen in mehreren Schritten nach dem in Daddas Arbeit beschriebenen Verfahren so reduziert, dass am Ende nur noch zwei Binärzahlen verbleiben, die dann noch addiert werden müssen um das Multiplikationsergebnis zu erhalten. Obwohl das Verfahren ursprünglich für eine konventionelle Multiplikation gedacht war, lässt es sich auf den vorliegenden Anwendungsfall übertragen, da der geschwindigkeitsrelevante Gedanke hinter der Dadda-Reduktion die Vermeidung multipler Addition durch Reduktion dieser auf eine einzelne Addition ist. Der in dieser Projektarbeit implementierte Dadda-Reduzierer-Baustein besitzt daher neben den zwei 8-Bit Eingängen für die beiden Faktoren einen weiteren Eingang für den aktuellen 16-Bit Akkumulator-Wert, woraus dann zwei 16-Bit Zahlen berechnet werden, die dann an den zwei 16-Bit Ausgängen des Dadda-Reduzierer-Bausteins für eine anschließende Addition bereitgestellt werden. Dabei prozessiert der Dadda-Reduzierer-Baustein Binärzahlen in ihrer Zweierkomplementdarstellung um das Rechnen mit negativen Zahlen zu ermöglichen. Die grundlegende Funktionsweise des Dadda-Reduzierer-Bausteins ist in Abbildung 2 dargestellt, welcher auch die Beschreibung der Funktion in der zugehörigen VHDL-Datei „dadda_reducer.vhdl“ folgt.

Um das Produkt zweier Binärzahlen im Zweierkomplement zu erhalten ist es, abweichend zum Vorgehen bei der normalen Multiplikation zweier Binärzahlen, erforderlich für die Multiplikation, also der Bitweisen Verundung, des HSBs des zweiten Faktors mit dem ersten Faktor zuvor das Zweierkomplement des ersten Faktors zu bilden. Dafür wird der 8-Bit Komplementbilder (3.2) verwendet, weswegen an dessen Dateneingang der erste Faktor und an dessen Steuereingang das HSB des zweiten Faktors angelegt werden. Für alle weiteren Bits des zweiten Faktors wird die gewöhnliche bitweise Verundung mit dem ersten Faktor durchgeführt. Die weitere Verarbeitung erfordert es zusätzlich die Vorzeichenerweiterungen der so erhaltenen Binärzahlen zu berücksichtigen. Diese entsprechen für die ersten sieben Binärzahlen, die durch

[1] Dadda, L. (1965). Some Schemes for Parallel Multipliers. *Alta Freq., 34, 349-356.*

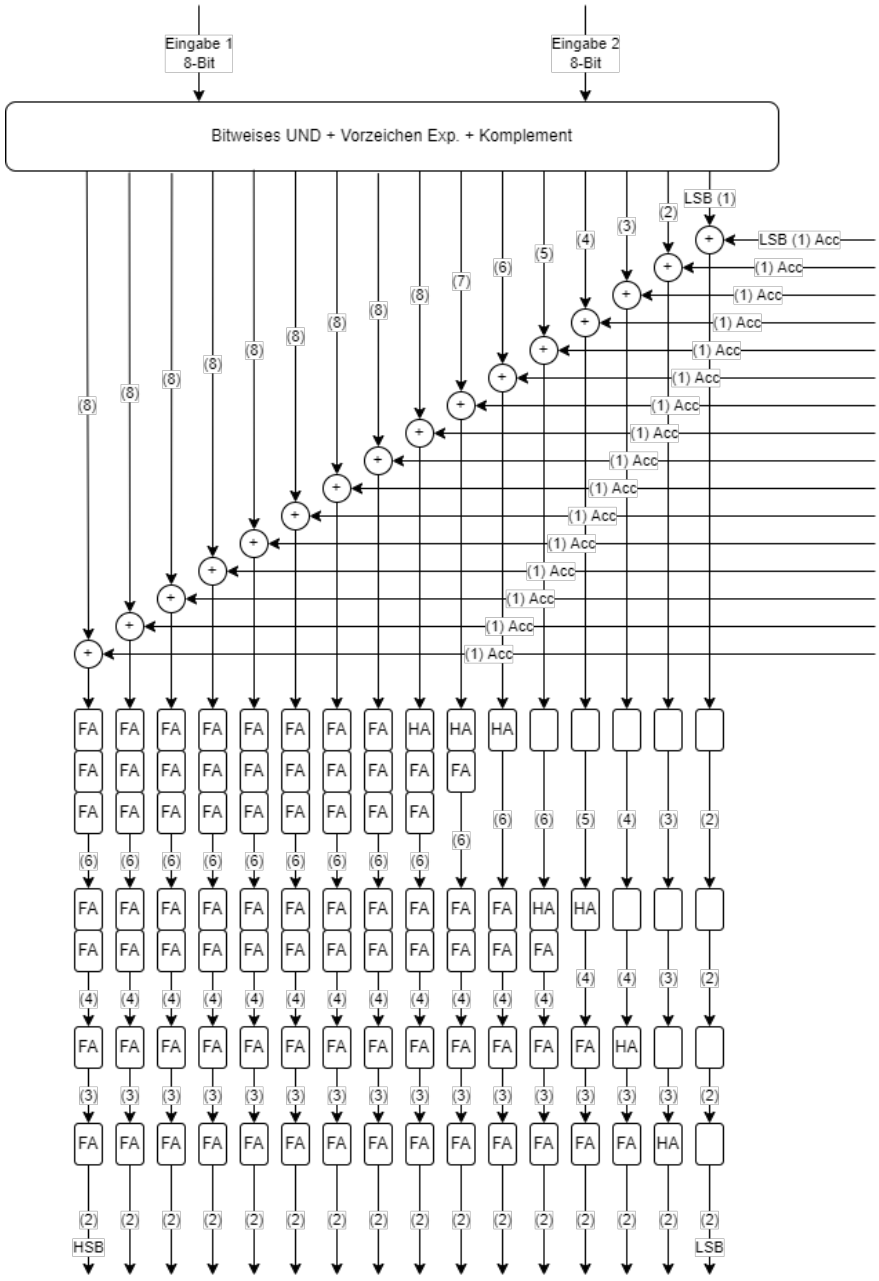

Abbildung 2: Layout des Dadda-Reduzierers; Dateneingänge (Eingabe 1, Eingabe 2, rechts Acc), Datenausgang (unten, zwei Zahlen, jeweils highest significant bit (HSB) links bis least significant bit (LSB) rechts)

die primitive Verundung erhalten wurden, einfach deren jeweiligem HSB. Für die verbleibende Binärzahl ist es jedoch notwendig die Vorzeichenerweiterung zu berechnen, denn hier entspricht die Vorzeichenerweiterung der Verundung des negierten HSB des ersten Faktors mit dem HSB des zweiten Faktors und dem HSB dieser verbleibenden Binärzahl. Diese aufwendigere Vorzeichenkonstruktion ist notwendig um für die Eigentlich notwendige Vertauschung des ersten und zweiten Faktors je nach deren aktueller Vorzeichenkombination zu kompensieren. Damit sind dann die in Abbildung 2 dargestellten Datenströme für die 16 verschiedenen Signifikanzniveaus berechnet, die den ersten Prozessblock verlassen. Es gibt auf dem untersten Signifikanzniveau (LSB) genau einen Datenstrom, der der Verundung des LSB des ersten Faktors mit dem LSB des zweiten Faktors entspricht, und auf dem höchsten Signifikanzniveau (ganz links in der Abbildung) acht verschiedene Datenströme, die den beschriebenen Vorzeichenerweiterungen der acht erhaltenen Binärzahlen entsprechen.

Um den Ablauf der Berechnung bis hierher zu verdeutlichen betrachten wir die Multiplikation der zwei 8-Bit Binärzahlen im Zweierkomplement -8 (1001) und -7 (1001). Das Zweierkomplement des ersten Faktors ist -8 (1000), denn 1000 addiert mit 1111 ergibt 0111 und das Einerkomplement von 0111 ist 1000. Es ergeben sich also folgende Datenströme, wobei in jeder Zeile eine Binärzahl angegeben ist und die Vorzeichenerweiterungen blau markiert sind:

1te Binärzahl	00001001
2te Binärzahl	0000000
3te Binärzahl	000000
4te Binärzahl	01001

Die Vorzeichenerweiterung der vierten Binärzahl ist dabei das Ergebnis der Verundung des negierten HSB des ersten Faktors, also 0, mit dem HSB des zweiten Faktors, also 1, und dem HSB der vierten Binärzahl ohne die Vorzeichenerweiterung, also 0, und damit 0. Im nächsten Schritt werden dann, wie in Abbildung 2 gezeigt, die Bits des aktuellen Akkumulatorwertes gemäß ihrer Stelligkeit den Datenstömen hinzugefügt, sodass sich die Anzahl an Datenströmen für jedes Signifikanzniveau genau um eins erhöht. Anschließend erfolgt die eigentliche Dadda-Reduktion in vier Stufen, wobei die Kästen jeweils andeuten, was mit den Datenströmen auf den einzelnen Signifikanzniveaus geschieht. Ein leerer Kasten deutet an, dass die eingehenden Da-

tenströme unverändert weiterverwendet werden. Ein HA-Kasten deutet die Verwendung eines Halbaddierers an, welcher zwei Datenströme des aktuellen Signifikanzniveaus nimmt und je einen Datenstrom für das gleiche und das nächst höhere Signifikanzniveau in der nächsten Reduktionsstufe erzeugt. Ein FA-Kasten deutet die Verwendung eines Volladdierers an, welcher drei Datenströme des aktuellen Signifikanzniveaus nimmt und je einen Datenstrom für das gleiche und das nächst höhere Signifikanzniveau in der nächsten Reduktionsstufe erzeugt. Final werden dann die zwei 16-Bit-Zahlen erhalten, die an den Addierer weitergegeben werden können. Die geschwindigkeitsbestimmenden Schritte des Dadda-Reduzierers sind zuerst die Komplementbildung mit Verundung für die Berechnung der Vorzeichenexpansion was, gemäß den NAND-Verzögerungen für den Komplementbilder (vgl. 3.2), zu einer Verzögerung von 26 NANDs führt. Dann folgt die eigentliche Reduktion, die eine maximale Tiefe von vier Volladdieren hat und damit eine Verzögerung von 28 NANDs (vgl. 2.6) aufweist. Das ergibt eine effektive Gesamtverzögerung des verwendeten Dadda-Reduzierers von 54 NANDs.

3.4 Brent-Kung-Addierer

Zur Umsetzung der Addition in der MAC-Unit wird ein Brent-Kung-Addierer verwendet, welcher 1982 von Richard P. Brent und H. T. Kung vorgestellt wurde [2]. Es handelt sich dabei um einen Präfix-Addierer, der die Carry-Bits in logarithmischer Zeit berechnet.
Es seien $a_1 \dots a_n$ und $b_1 \dots b_n$ zwei n-Bit Zahlen. Für $i = 1, \dots, n$ wird das n-te Summen-Bit s_n durch

$$s_n = a_i \oplus b_i \oplus c_{i-1}$$

berechnet, wobei c_{i-1} das $(i-1)$-te Carry-Bit darstellt.
Mit Hilfe der Generate- und Propagate-Bits

$$g_i := a_i \vee b_i$$

$$p_i := a_i \oplus b_i$$

[2]Brent / Kung A Regular Layout for Parallel Adders, 1982-03, IEEE Transactions on Computers , Vol. C-31, No. 3 Institute of Electrical and Electronics Engineers (IEEE) p. 260-264

kann durch

$$c_i = \begin{cases} 0 & , i = 0 \\ g_i \vee (p_i \wedge c_{i-1}) & , sonst \end{cases}$$

das i-te Carry-Bit berechnet werden. Die Grundidee des Brunt-Kung-Addierers ist es, die Carry-Bits so schnell wie möglich zu berechnen. Dazu wird der Operator o eingeführt. Für die Bits g, p, g' und p' ist

$$(g, p)o(g', p') := (g \vee (p \wedge g'), p \wedge p'$$

die Definition des Operators o. Als nächstes wird

$$(G_i, P_i) := \begin{cases} (g_i, p_i) & , i = 1 \\ (g_i, p_i)o(G_{i-1}, P_{i-1}) & , sonst \end{cases}$$

gesetzt. Dann gilt für alle $i = 1, \ldots, n$:

$$c_i = G_i.$$

Der Beweis wird mittels vollständiger Induktion über alle $i = 1, \ldots, n$ geführt. Ist $i = 1$, so ist

$$c_1 = g_1 \vee (p_1 \wedge c_0) = g_1 \vee (p_1 \wedge 0) = g_1 = G_1,$$

der Induktionsanfang ist also erfüllt. Es wird der Schritt $i-1 \mapsto i$ betrachtet. Dann ist

$$(G_i, P_i) = (g_i, p_i)o(G_{i-1}, P_{i-1}) = (g_i, p_i)o(c_{i-1}, P_{i-1}) = (g_i \vee (p_i \wedge c_{i-1}), (p_i \wedge P_{i-1}),$$

woraus

$$G_i = g_i \vee (p_i \wedge c_{i-1}) = c_i$$

folgt. Somit ist auch der Induktionsschritt erfüllt.
Zudem zeigt

$$\begin{aligned} [(g_3, p_3)o(g_2, p_2)]o(g_1, p_1) &= (g_3 \vee (p_3 \wedge g_2), p_3 \wedge p_2)o(g_1, p_1) \\ &= (g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge p_1), p_3 \wedge p_2 \wedge p_1) \\ &= (g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1), p_3 \wedge p_2 \wedge p_1) \\ &= (g_3 \vee (p_3 \wedge (g_2 \vee (p_2 \wedge g_1))), p_3 \wedge p_2 \wedge p_1) \\ &= (g_3, p_3)o(g_2 \vee (p_2 \wedge g_1), p_2 \wedge p_1) \\ &= (g_3, p_3)o[(g_2, p_2)o(g_1, p_1)], \end{aligned}$$

dass der Operator o assoziativ ist. Aufgrund der Assoziativität und da

$$(G_i, P_i) = (g_i, p_i) o (g_{i-1}, p_{i-1}) o \ldots o (g_1, p_1)$$

ist, kann das Paar (G_k, P_k) durch

$$\begin{aligned}(G_k, P_k) &= (g_k, p_k) o (g_{k-1}, p_{k-1}) o (g_{k-2}, p_{k-2}) o (g_{k-3}, p_{k-3}) \ldots (g_4, p_4) o (g_3, p_3) o (g_2, p_2) o (g_1, p_1) \\ &= ((g_k, p_k) o (g_{k-1}, p_{k-1})) o ((g_{k-2}, p_{k-2}) o (g_{k-3}, p_{k-3})) \ldots ((g_4, p_4) o (g_3, p_3)) o ((g_2, p_2) o (g_1, p_1)) \\ &= (((g_k, p_k) o (g_{k-1}, p_{k-1})) o ((g_{k-2}, p_{k-2}) o (g_{k-3}, p_{k-3}))) \ldots (((g_4, p_4) o (g_3, p_3)) o ((g_2, p_2) o (g_1, p_1)))\end{aligned}$$

berechnet werden, wobei sukzessive die Operanden des Operators o geklammert werden. Diese Berechnung kann durch einen Carry-Baum (Abbildung 3) realisiert werden. Die Knoten des Baums erhalten als Eingabe Generate- und Propagate-Bits, welche sie entweder unverändert weiterleiten oder aus ihnen neue Generate- und Propagate-Bits berechnen (Abbildung 4). Unter Verwendung eines invertierten Carry-Baums können für alle $i = 1, \ldots, n$ das Paar (G_i, P_i) berechnet werden (Abbildung 5), wodurch sich insbesondere der Carry c_i bestimmt lässt.
Jedes Carry-Bit und jedes Summen-Bit kann in $\mathcal{O}(\log(n))$ berechnet werden, wodurch zwei n-Bit-Zahlen in $\mathcal{O}(n \log(n))$ addiert werden können.
Für die MAC-Unit werden ein 8-Bit und ein 16-Bit Brent-Kung-Addierer verwendet. Die Implementierungen befinden sich in den Dateien „brent_kung_adder8.vhdl“ und „brent_kung_adder16.vhdl“ zu finden. Die weißen Knoten im Carry-Baum werden in der Implementierung durch Verbindungen repräsentiert, da sie die Generate- und Propagate-Bits nur weiterleiten. Die Implementierung der schwarzen Knoten ist in der Datei „brent_kung_processor.vhdl“ zu finden. Für den 8-Bit Brent-Kung-Addierer ergibt sich eine Verzögerung von 21 NANDs und für den 16-Bit Brent-Kung-Addierer ergibt sich eine Verzögerung von 25 NANDs.

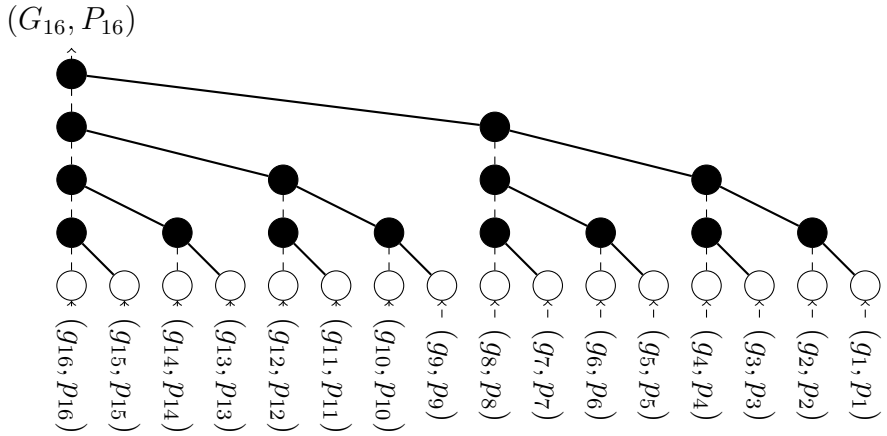

Abbildung 3: Mit Hilfe dieses Baum kann das Paar (G_{16}, P_{16}) und somit insbesondere der Carry c_{16} berechnet werden. Die Funktionalität der Knoten ist in Abbildung 4 dargestellt.

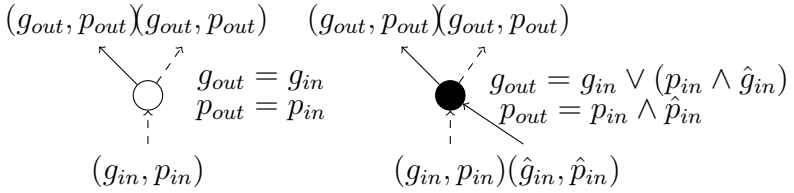

Abbildung 4: Der weiße Knoten des Carry-Baums leitet die eingehenden Generate- und Propagate-Bits g_{in} und p_{in} unverändert weiter. Der schwarze Knoten berechnet hingegen die ausgehenden Generate- und Propagate-Bits.

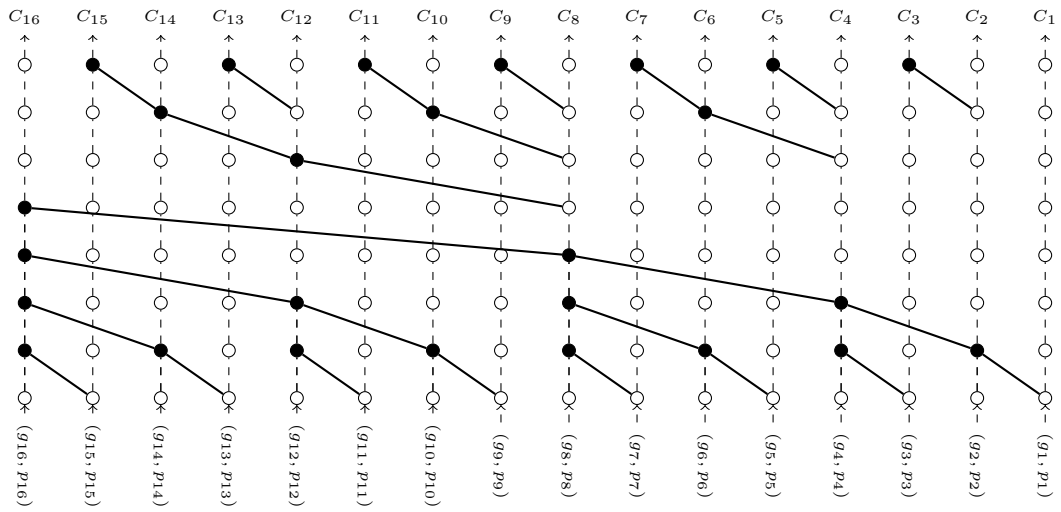

Abbildung 5: Durch die Kombination des Carry-Baums in der unteren Hälfte mit dem invertierten Carry-Baum in der oberen Hälfte können alle Carry-Bits berechnet werden.

4 MAC-Layout

Im folgenden wird das Layout des in dieser Projektarbeit entworfenen MAC-Bausteins anhand von Abbildung 6 erläutert, das auch auf diese Weise in der zugehörigen VHDL-Datei „mac.vhdl“ umgesetzt ist. Der MAC-Baustein ist in der Lage folgende Berechnung vorzunehmen:

$$\mathrm{acc}^{n+1} = \mathrm{acc}^{n} + \mathrm{f}_1 * \mathrm{f}_2$$

Es wird also der nächste Accumulatorzustand berechnet, indem zum aktuellen Akkumulatorzustand das Produkt der beiden übergebenen Faktoren addiert wird. Zu diesem Zweck stellt der MAC-Baustein nach außen diverse Schnittstellen zu Verfügung. Einerseits die Dateneingänge zum Setzen der beiden Faktoren (I1-Value, I2-Value), die in Abbildung 6 den Dateneingängen der zugehörigen PIPO-Register entsprechen, und ein Dateneingang zum Setzen des Akkumulators (Acc-Value). Um zu kontrollieren ob im nächsten Takt eine Berechnung durchgeführt werden soll (logisch 0) oder der Akkumulator auf den anliegenden Wert gesetzt werden soll (logisch 1), gibt es zusätzlichen einen Multiplexer, der zwischen diesen beiden Optionen gesteuert durch das Steuersignal Set Acc wählt. Die Register der beiden Faktoren besitzen zudem je einen synchronen Set/Reset-Steuersignaleingang, sodass bei der nächsten ansteigenden Taktflanke entweder der anliegende Datenwert in das jeweilige Register übernommen wird (logisch 0) oder das Register in den Null-Zustand überführt wird (logisch 1). Das Akkumulatorregister verfügt über einen asynchronen Reset-Steuereingang, mit welchem der Akkumulator jederzeit in den Null-Zustand überführt werden kann. Weiterhin gibt es einen 1-Bit Ausgang der angibt, ob es in der letzten Berechnung zu einem Overflow kam und einen 1-Bit Ausgang der angibt, ob es in der letzten Berechnung zu einem Underflow kam. Beide Informationen werden in je einem zugehörigen D-Flip-Flop gespeichert. Alle genannten getakteten Bausteinen werden durch das selbe Taktsignal gesteuert. Die eigentliche Berechnung nutzt die Zustande der drei großen PIPO-Register und erfolgt im Dadda-Reduzierer-Baustein (Multiplikation) und im Brent-Kung-Addierer-Baustein (finale Addition). Die genannten Bausteine werden in den jeweiligen Kapiteln dieses Berichtes detaillierter erläutert. Der einzig verbleibende Baustein, dessen Funktionsweise im Folgenden näher erläutert werden soll, berechnet ob es zu einem Overflow oder einem Underflow im aktuellen Takt gekommen ist.

Um zu entscheiden, ob es zu einem Overflow oder einem Underflow gekommen ist, sind lediglich die führenden Bits des aktuellen Akkumulatorwerts,

der beiden Faktoren und des Ergebnisses relevant. Dabei ist ein Overflow dasjenige Ereignis, bei dem die Addition zweier positiver Zahlen zu einem negativen Ergebnis führt, und ein Underflow dasjenige Ereignis, bei dem die Addition zweier negativer Zahlen zu einem positiven Ergebnis führt. Folgende Vorzeichenkombinationen können dabei auftreten:

f_1	f_2	acc^n	acc^{n+1}	Bewertung
-	-	-	-	Korrekt
-	-	-	+	Korrekt
-	-	+	-	Overflow
-	-	+	+	Korrekt
-	+	-	-	Korrekt
-	+	-	+	Underflow
-	+	+	-	Korrekt
-	+	+	+	Korrekt
+	-	-	-	Korrekt
+	-	-	+	Underflow
+	-	+	-	Korrekt
+	-	+	+	Korrekt
+	+	-	-	Korrekt
+	+	-	+	Korrekt
+	+	+	-	Overflow
+	+	+	+	Korrekt

Tabelle 6: Mögliche Zustände bei der Berechnung

Wie in Tabelle 6 zu erkennen, gibt es genau zwei Fälle in denen ein Overflow aufgetreten ist und genau zwei Fälle in denen ein Underflow aufgetreten ist. Ob diese Fälle eingetreten sind lässt sich gemäß der jeweiligen Normalformausdrücke berechnen, wobei zu beachten ist, dass ein - in der Tabelle eine logische 1 impliziert. Für die Overflows gilt damit:

$$(f_1 \wedge f_2 \wedge \neg acc^n \wedge acc^{n+1}) \vee (\neg f_1 \wedge \neg f_2 \wedge \neg acc^n \wedge acc^{n+1})$$

und für die Underflows folgerichtig:

$$(f_1 \wedge \neg f_2 \wedge acc^n \wedge \neg acc^{n+1}) \vee (\neg f_1 \wedge f_2 \wedge acc^n \wedge \neg acc^{n+1})$$

Da das Ergebnis der MAC-Berechnung als letztes zu Verfügung steht, ist es am Zeiteffizientesten die Verundung mit acc^{n+1} bzw. $\neg acc^{n+1}$ jeweils nach

den anderen Verundungen auszuführen. Ansonsten findet die Berechnung unter Verwendung der logischen Grundbausteinen AND, NOT und OR genau gemäß der angegebenen logischen Ausdrücke statt. So ausgeführt ist die Overflow und Underflow Berechnung nicht der geschwindigkeitsbestimmende Schritt der Berechnung. Die Gesamtverzögerung des MAC-Bausteins ergibt sich auf dem Pfad von den (Eingabe-)Registern über den Dadda-Reduzierer-Baustein (vgl. 3.3), den Brent-Kung-Addierer (3.4) und den Multiplexer (vgl. 2.8) bis zum Ausgaberegister und beträgt damit 54 NANDs + 25 NANDs + 4 NANDs = 83 NANDs.

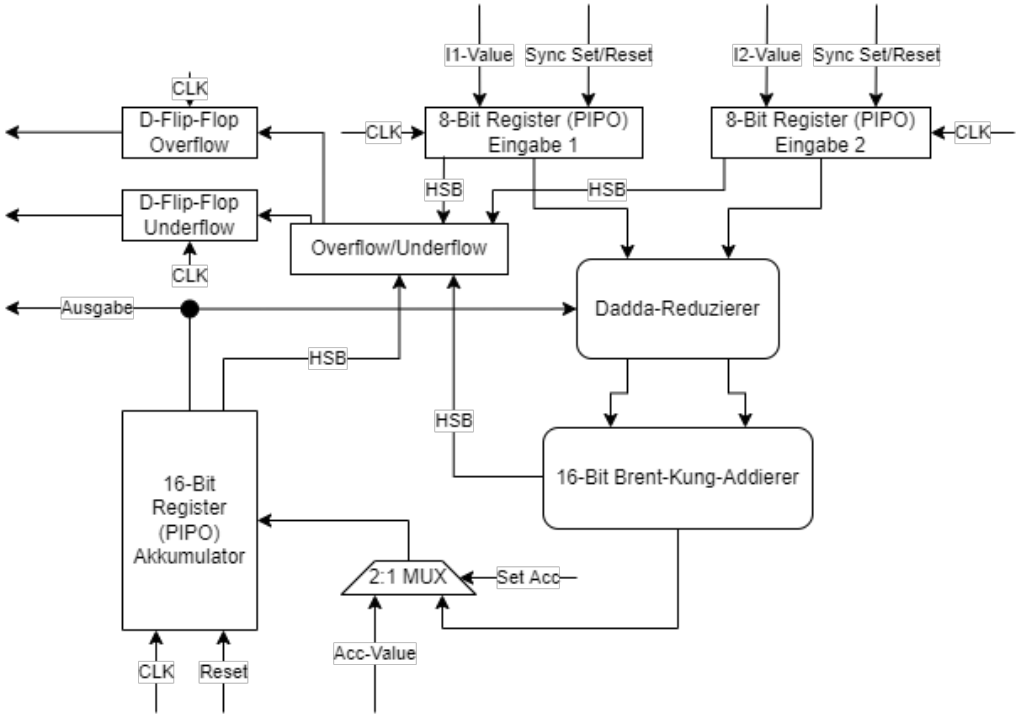

Abbildung 6: Layout des MAC-Bausteins; Dateneingänge (I1-Value, I2-Value, Acc-Value), Steuereingänge (Sync Set/Reset Eingabe 1, Sync Set/Reset Eingabe 2, Set Acc, Reset), Takteingang (CLK), Datenausgänge (Ausgabe, Overflow, Underflow)

5 Evaluation

Um das Design der MAC-Unit zu evaluieren, wird es mit einer MAC-Unit aus einem vergangenen Fachprojekt [3] verglichen. Dazu werden die theoretischen Verzögerungen beider Entwürfe in Verzögerungen von NAND-Gattern umgerechnet und dann gegenübergestellt. Die andere MAC-Unit verwendet anstelle des Brent-Kung-Addierers einen Kogge-Stone-Addierer und statt des Dadda-Multiplizierers wird ein Wallace-Tree-Multiplizierer verwendet. Beim Kogge-Stone-Addierer entfällt eine Verzögerung von einem XOR-Gatter auf die Generate- und Propagate-Logik. Dazu kommt noch vier mal die Verzögerung eines AND- und eines OR-Gatters. Zudem wird noch die Verzögerung eines XOR-Gatters hinzugerechnet. Für den Kogge-Stone-Addierer ergibt sich also eine Verzögerung von 22 NAND-Gattern.
Auf den Wallace-Tree-Multiplizierer entfällt eine Verzögerung von einem Kogge-Stone-Addierer, zwei XOR-Gattern, drei MUX 2:1, einem AND-Gatter, vier Volladdierern und 16 OR-Gattern. Dadurch ergibt sich für den Wallace-Tree-Multiplizierer eine Gesamtverzögerung von 145 NANDs.
Nach den Ausführungen in Kapitel 4 ergibt sich für das in diesem Projekt vorgestellt Design eine Verzögerung von 83 NAND-Gattern. Bei einer angenommen Verzögerung von 0,33 ps pro NAND-Gatter [4], ist die Verzögerung der hier vorgestellten MAC-Unit mit $83 \cdot 0,33 \text{ ps } = 27,39 \text{ ps}$ um $27,72$ ps schneller, als die Verzögerung der MAC-Unit aus dem anderen Fachprojekt, die $145 \cdot 0,33 = 55,11$ ps beträgt.

Zur praktischen Umsetzung des Designs wurde die MAC-Unit mittels des Programms Vivado [5] simuliert. Die Simulation ergab eine Gesamtverzögerung von $4,743$ ns, von der $0,907$ ns auf die logische Verzögerung und $3,836$ ns auf die Verbindungsverzögerung entfallen. Daraus ergibt sich eine maximale Taktfrequenz von $0,211$ GHz. Es ist zu erkennen, dass die auf NAND-Gattern basierende theoretische logische Verzögerung mit $27,39 \text{ ps } = 0,02739$ ns deutlich kürzer als die logische Verzögerung der Simulation ist. Es kann also geschlossen werden, dass die Errechnung der Verzögerung mittels NAND-Gattern nur eingeschränkt zur Abschätzung der realen Laufzeit geeignet ist.

[3] https://github.com/TUD-MLA/FP6-Computation-Unit/tree/main/VHDL

[4] Nirmal, Vijaya; Kumar, Sam Jabaraj. (2010). NAND GATE USING FINFET FOR NANOSCALE TECHNOLOGY.
Int. j. eng. sci. technol., Vol. 2(5), 1351-1358.

[5] https://www.xilinx.com/products/design-tools/vivado.html

6 Zusammenfassung

Im vorliegenden Fachprojekt wurden mittels Brent-Kung-Addierer und eines Dadda-Multiplizierers eine MAC-Unit entworfen und in VHDL implementiert. Das Besondere dieses Entwurfs ist die Integration des Addition in den Multiplikationsschritt. Die theoretische Laufzeit des Designs wurde durch Umrechnung auf die Verzögerung von NAND-Gattern bestimmt und mit der Laufezit einer MAC-Unit aus einem vergangenen Fachprojekt verglichen, mit dem Ergebnis, dass die in dieser Arbeit vorgestellte MAC-Unit schneller ist. Zudem wurde die MAC-Unit mittels des Programms Vivado simuliert.