

Abschlussprojekt im Modul

E980 Schaltkreisentwurf u. Simulation elektronischer Schaltungen

Entwurf eines Rechenwerks zur Umsetzung des Euklidischen Algorithmus

Johannes Trummer

Johannes.trummer@stud.htwk-leipzig.de

Wallroda 29

06647 Bad Bibra

Betreuender Professor

Prof. Dr.-Ing. Marco Krondorf

Inhaltsverzeichnis

1.	Erklärung der algorithmischen Idee.....	3
1.1	Einleitung.....	3
1.2	Algorithmik für eine Simulation.....	4
1.3	Algorithmik für einen realen FPGA.....	7
2.	Dokumentation der HDL-Module	7

1. Erklärung der algorithmischen Idee

1.1 Einleitung

Der Euklidische Algorithmus ist ein iterativer Algorithmus zur Bestimmung des größten gemeinsamen Teilers (ggT, engl. gcd) zweier Zahlen. Er ist effizient und schnell und kann optimal mittels eines Rechenwerkes, bestehend aus Controller, Datapath und ALU, umgesetzt werden.

Der Algorithmus basiert auf der iterativen Restbildung der beiden Zahlen. Das bedeutet, die erste Zahl wird durch die zweite Zahl dividiert, sodass ein ganzzahliges Ergebnis entsteht. Anschließend wird der Restbetrag identifiziert. Diese Operation wird auch als Modulo bezeichnet und wird in dieser Dokumentation noch eine besondere Rolle einnehmen. Eine Voraussetzung für diesen Algorithmus ist, dass die erste Zahl (Zahl1) die größere der beiden Zahlen sein muss. Sonst würde die Restberechnung nicht funktionieren. An dieser Stelle erscheint eine Definition sinnvoll, ob der Algorithmus nur funktionieren soll, wenn die beschriebene Voraussetzung gegeben ist, oder ob er automatisch die Zahlen tauschen soll, sodass die Berechnung anschließend möglich ist.

Im Falle dieser Dokumentation wird festgelegt, dass der umgesetzte Algorithmus die Zahlen automatisch tauscht, sollten sie in falscher Reihenfolge eingegeben worden sein.

Ein weiteres wichtiges Kriterium ist die Festlegung des Zahlenbereichs, der abgedeckt werden soll. Es erscheint hier sinnvoll, den Bereich der natürlichen Zahlen zu wählen, da einerseits die negativen Zahlen keine sinnvolle Erweiterung des Bereichs, sondern nur eine Dopplung wären und andererseits ist die Erweiterung auf die reellen Zahlen ebenfalls nicht sinnvoll, da die Berechnung des ggT zweier reeller Zahlen nicht nur unüblich ist, sondern auch weitere Probleme und Schwierigkeiten mit sich bringt. Da das Rechenwerk auf einem FPGA umgesetzt werden soll, muss zudem die Bitbreite der Zahlen festgelegt werden. Mit einer Bitbreite von 16 Bit sind Berechnung von Zahlen bis 65.535. Dies erscheint ausreichend. Folglich wird der Wertebereich auf natürliche Zahlen von 0 bis 65.535 festgelegt.

Im folgenden Schema ist eine beispielhafte Restberechnung mittels Modulo-Operator (%-Operator) dargestellt:

$$24.255 : 12.540 = 1,9342105 \rightarrow 1 \text{ Rest } 11.715$$

$$24.255 \% 12.540 = 11.715$$

1.2 Algorithmik für eine Simulation

Auch wenn eine Unterteilung in eine simulierte und eine real umsetzbare Algorithmik auf den ersten Blick unsinnig erscheint, macht es durchaus Sinn, mit einer Simulation zu beginnen. Das Kriterium, welches die Simulation von der echten Synthese unterscheidet, ist nämlich der Modulo-Operator, oder genauer gesagt die Division, die im Modulo-Operator verwendet wird.

In einem FPGA kann die Division nicht ohne weiteres mittels bekannten Operators wie beispielsweise in Python oder C genutzt werden. Um eine Modulo-Operation durchzuführen, muss dafür ein eigenes Rechenwerk umgesetzt werden. Diese Überlegung ist ein Ergebnis aus dem Gedanken, dass sich das umgesetzte Rechenwerk nicht auf Zahlen beschränken soll, die der Menge aller Zahlen 2^n entspringen. Mit diesen Zahlen wäre es deutlich einfacher, da Division hier durch Shiften möglich ist. Allerdings soll das Rechenwerk den größten gemeinsamen Teiler von verschiedensten Zahlen berechnen können.


Im ersten Schritt, dem Entwurf für Simulation mit ModelSim kann der Modulo-Operator allerdings verwendet werden, denn die ModelSim-Software unterstützt diesen. Die Simulation wird nun also verwendet, um die Architektur des Euklidischen Algorithmus in prinzipieller Form umzusetzen, ohne der Modulo-Operation zu früh eine zu große Bedeutung zu verleihen.

Im folgenden Schema soll nun die Struktur und Algorithmik der Berechnung gezeigt werden. Wie bereits erwähnt, ist die größere der beiden Zahlen als Zahl1 zu wählen. Anschließend wird im ersten Schritt die erste Modulo-Berechnung durchgeführt, wie ebenfalls bereits gezeigt:

$$1. \text{ Schritt: } 24.255 \% 12.540 = 11.715$$


Im folgenden Schritt wird die Modulo-Operation erneut durchgeführt, allerdings mit anderen Zahlen. Zahl2 (12.540) wird nun zu Zahl1 und das Ergebnis der Berechnung wird zu Zahl2:

$$1. \text{ Schritt: } 24.255 \% 12.540 = 11.715$$


$$2. \text{ Schritt: } 12.540 \% 11.715 = 825$$

Diese Berechnung wird so lange fortgeführt, bis das Ergebnis der Modulo-Operation Null beträgt:

$$2. \text{ Schritt: } 12.540 \% 11.715 = 825$$


$$3. \text{ Schritt: } 11.715 \% 825 = 165$$

$$4. \text{ Schritt: } 825 \% 165 = 0$$

Da dies im vierten Schritt der Fall ist, wird nun das Ergebnis des vorherigen Schrittes, also dem dritten Schritt (165) herangezogen und dieses Zwischenergebnis bildet den größten gemeinsamen Teiler der beiden Zahlen 24.255 und 12.540.

$$\rightarrow \text{ggT}(24255, 12540) = 165$$

Mit dieser iterativen Berechnung kann nun ein Datenflussgraph erstellt werden, mit dem die Architektur des Rechenwerks umgesetzt werden kann. Aus Platzgründen wird der Graph hier nur stellenweise erwähnt und wird stattdessen vollumfänglich im Anhang abgebildet.

Schritte des Controllers

In der linken Spalte befinden sich die einzelnen Schritte des Controllers mit den zugehörigen, im Code vergebenen localparams und den passenden Ziffern. Die folgende Abbildung zeigt die Definition dieser lokalen Parameter in der Datei controller.v:

Write-Enable-Flags vom Controller an den Datapath

Die Write-Enable-Flags bilden einen Teil der blauen waagerechten Pfeile im Datenflussgraph. Sie sind mit „wren_...“ beschriftet und zeigen an, dass ein neuer Wert in ein Register geschrieben werden soll. Dieser neue Wert kann sowohl der Ergebniswert aus der ALU sein, aber auch der Wert, der ein anderes Register hält.

...to_alu -Flags vom Controller an den Datapath

Die ...to_alu-Flags bilden den zweiten Teil der blauen Pfeile im Datenflussgraph und haben immer die Richtung nach rechts. Sie symbolisieren, dass ein gewisser Wert auf einen Operanden der ALU geladen wird. Diese Flags sind immer mit einer ALU-Operation verbunden.

ALU-Modes

Die verschiedenen ALU-Modes sind ganz rechts dargestellt und ebenfalls mit Namen gekennzeichnet. Der ALU-Mode „ALU_give_back_bigger“ gibt beispielsweise den größeren der beiden angelegten Operanden zurück.

Der Datenflussgraph hat insgesamt neun Schritte, einen weiteren als IDLE-Schritt, der hier allerdings nicht aufgeführt ist. Begonnen wird der Algorithmus mit einem initialen Schreiben der registerten Eingänge auf die Register `Zahl1_r` und `Zahl2_r` im Schritt **STATE_initial_write**. Im nächsten Schritt werden diese beiden Zahlen an die ALU übergeben und die ALU gibt als Ergebnis die größere der beiden Zahlen aus. In Schritt **STATE_find_smaller** das Ergebnis der vorherigen Operation in das Register **zwischen_groß_r** geschrieben und erneut werden beide Zahlen (`Zahl1_r` und `Zahl2_r`) an die ALU übergeben, die diesmal die kleinere Zahl als Ergebnis ausgibt. Das Ergebnis wird im nächsten Schritt in **zwischen_klein_r** geschrieben. Anschließend in **STATE_write_zwischenspeicher** werden die Werte beider Zwischenregister (`zwischen_klein_r` und `zwischen_groß_r`) in die Register `Zahl1_r` und `Zahl2_r` geschrieben. `Zwischen_groß_r` in `Zahl1_r` und `zwischen_klein_r` in `Zahl2_r`. Weiterhin wird in diesem Schritt der Wert von `zwischen_klein_r` in das Register `ergebnis_zuvor_r` geschrieben. Mehr zu dieser Besonderheit unter dem Punkt gleiche Zahlen.

Damit sind die Zahlen nun in jedem Fall so gedreht, dass die größere in `Zahl1_r` und die kleinere in `Zahl2_r` steht und der Algorithmus kann beginnen.

Der Schritt **STATE_calc** übergibt über die beiden Flags die Register `Zahl1_r` und `Zahl2_r` an die ALU. Diese führt die Modulo-Operation durch und gibt das Ergebnis zurück an den Datapath. Im Folgenden Schritt (**STATE_write_erg**) wird das Ergebnis ins Register `erg_modulo_r` geschrieben. Nun wird im kommenden Schritt (**STATE_check_if_zero**) überprüft, ob das eben geschriebene Ergebnis Null ist. Dies erfolgt über das Flag **check_for_termination**, welches der Controller dem Datapath übergibt. Sollte das Ergebnis Null sein, wird der nächste Schritt **STATE_IDLE** sein, von dem aus eine neue Berechnung gestartet werden kann. Ist das Ergebnis nicht Null, werden die Zahlen, verteilt über zwei Schritte, wie oben gezeigt verschoben: `Zahl1_r` nimmt den Wert von `Zahl2_r` an, welches nun den Wert von `erg_modulo_r` annimmt. Der gleiche Wert wird in `erg_zuvor_r` geschrieben, um im nächsten Schritt auf dieses Ergebnis zugreifen zu können, sollte die Berechnung fertig sein.

Sonderfall zwei gleiche Zahlen

Der größte gemeinsame Teiler zweier gleicher Zahlen (zum Beispiel $\text{ggT}(123, 123)$) ist immer die Zahl selbst. In diesem Fall ergibt die erste Modulo-Berechnung des Algorithmus sofort eine Null und das Ende des Algorithmus wird eingeleitet. Das Ergebnis des Algorithmus wird nun aus dem Register `erg_zuvor_r` bezogen. Da dieses aber ohne eine vorherige Berechnung noch nicht beschrieben wurde und somit keinen Wert hält, würde dies nicht funktionieren. Aus diesem Grund wird das Register

erg_zuvor_r im Schritt **STATE_write_zwischenspeicher** mit dem Wert des Registers zwischen_klein_r beschrieben, sodass dieser Wert dann als Ergebnis ausgegeben werden kann.

1.3 Algorithmik für einen realen FPGA

Möchte man den vorgestellten Algorithmus für einen FPGA synthetisieren, so muss man auf den Modulo-Operator (%) verzichten und andere Wege der Berechnung finden. Da eine Begrenzung der Eingaben ausgeschlossen wurde, ist es hier naheliegend ein eigenes Rechenwerk für die Funktionalität umzusetzen. Dies wird im Folgenden beschrieben.

2. Dokumentation der HDL-Module

Im Folgenden werden die einzelnen Verilog-Module anhand ihrer Ein- und Ausgangsports vorgestellt. Die Blockbilder wurden mit dem Quartus-internen RTL-Viewer automatisch anhand des Codes erstellt. Zwar wird hier nicht mit Pfeilen dargestellt, ob es sich um einen Eingangs- oder Ausgangsport handelt, allerdings befinden sich alle Eingangsports auf der linken Seite des Blocks und alle Ausgangsports auf der rechten Seite des Blocks. Weiterhin gibt die Endung der Namen Aufschluss darüber, ob es sich um einen Eingang oder Ausgang handelt. Die Namen der Eingänge enden mit `_i`, die Namen der Ausgänge enden auf `_o`.

ggt_top.v

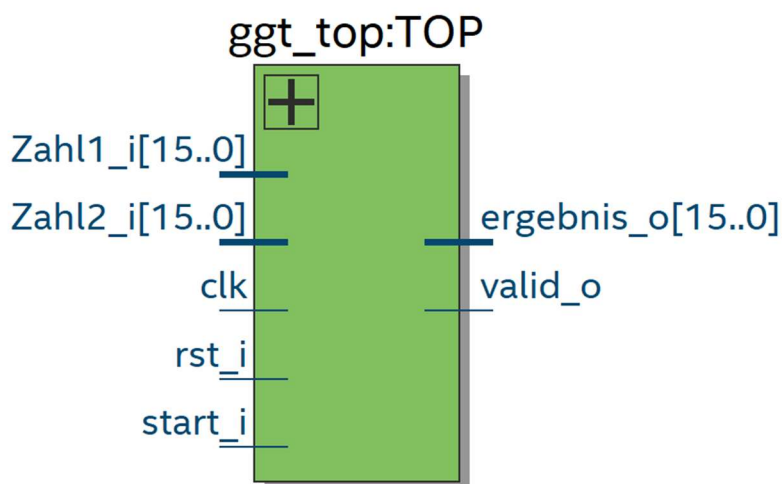


Abbildung 1: Mit dem RTL-Viewer generierter Block des Moduls ggt_top.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High active reset signal	I
start_i	High active start signal	I
Zahl1_i	16 bit number Zahl1	I
Zahl2_i	16 bit number Zahl2	I
ergebnis_o	16 bit number result	O
valid_o	High active valid signal if ready	O

controller.v

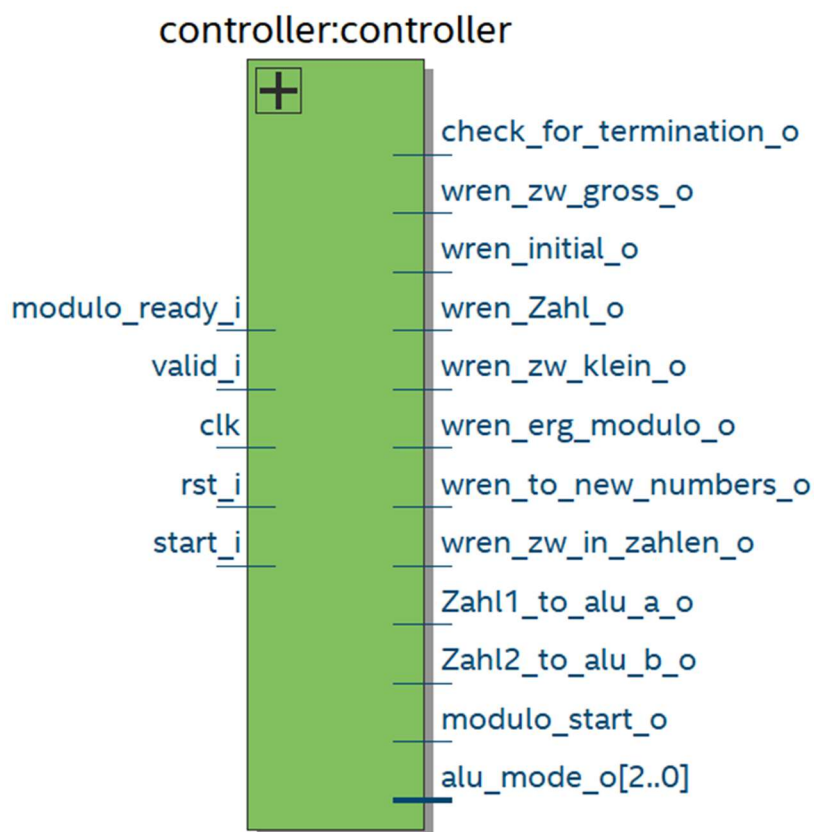


Abbildung 2: Mit dem RTL-Viewer generierter Block des Moduls controller.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High active reset signal	I
start_i	High active start signal	I
valid_i	High active valid signal if ready, from datapath.v	I
modulo_ready_i	High active signal, if modulo operation is finished, from datapath.v	I

check_for_termination_o	High active signal to check if result of modulo = 0, to datapath.v	O
wren_Zahl_o	High active Write-enable signal, to datapath.v	O
wren_zw_klein_o	High active Write-enable signal, to datapath.v	O
wren_zw_gross_o	High active Write-enable signal, to datapath.v	O
wren_erg_modulo_o	High active Write-enable signal, to datapath.v	O
wren_to_new_numbers_o	High active Write-enable signal, to datapath.v	O
wren_zw_in_zahlen_o	High active Write-enable signal, to datapath.v	O
wren_initial_o	High active Write-enable signal, to datapath.v	O
Zahl1_to_alu_a_o	High active Write-enable signal, writing to alu_a, to datapath.v	O
Zahl2_to_alu_b_o	High active Write-enable signal, writing to alu_b, to datapath.v	O
modulo_start_o	High active start signal to start modulo	O
alu_mode_o	3 bit output declaring what operation ALU should do, to datapath.v	O

datapath.v

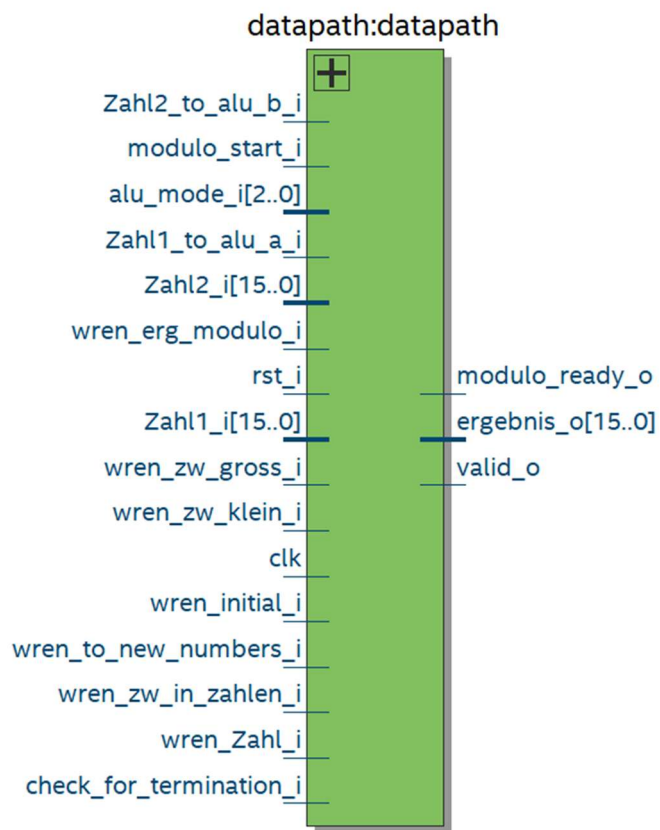


Abbildung 3: Mit dem RTL-Viewer generierter Block des Moduls datapath.v