

**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 .....	8
2.1	Hauptrechenwerk.....	8
2.2	Modulo-Rechenwerk .....	12
3.	Verschaltung der HDL-Module.....	16
3.1	Hauptrechenwerk.....	16
3.2	Modulo-Rechenwerk .....	18
4.	Test und Verifikation .....	18
4.1	Funktionstests mit Hilfe von ModelSim .....	18
4.2	Funktionstests auf dem realen FPGA.....	25
4.3	Timing-Analyse.....	26
4.4	Analyse des Stromverbrauchs mit Quartus Prime .....	27

# 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:

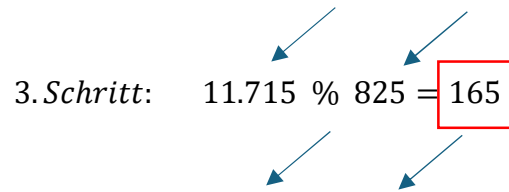
$$1. \text{Schritt: } 24.255 \% 12.540 = 11.715$$

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:

$$\begin{array}{lcl} 1. \text{Schritt:} & 24.255 \% 12.540 = 11.715 & \\ & \swarrow \quad \searrow & \\ 2. \text{Schritt:} & 12.540 \% 11.715 = 825 & \end{array}$$

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 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

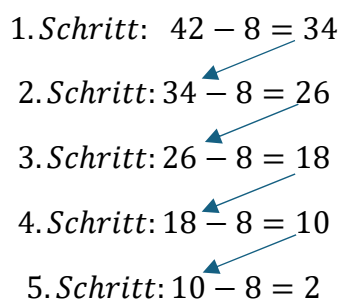
Der größte gemeinsame Teiler zweier gleicher Zahlen (zum Beispiel  $\text{ggT}(123, 123)$ ) ist immer die Zahl selbst. In diesem Fall ergibt die erste Modulo-Berechnung des Algorithmus sofort eine Null und das Ende des Algorithmus wird eingeleitet. Das Ergebnis des Algorithmus wird nun aus dem Register **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 Modulo-Funktionalität umzusetzen. Dies wird im Folgenden beschrieben.

Die Modulo-Berechnung kann umgesetzt werden, indem die kleinere Zahl von der größeren Zahl abgezogen wird und die größere Zahl als neuen Wert das Ergebnis dieser Differenz annimmt. Folglich wird wieder voneinander subtrahiert. Dies wird so lange durchgeführt, bis das Ergebnis kleiner ist als die kleinere Zahl. Der Rest entspricht dann der übriggebliebenen kleineren Zahl. An folgendem Beispiel soll dies anschaulicher gezeigt werden.

1. Schritt:  $42 - 8 = 34$   
2. Schritt:  $34 - 8 = 26$   
3. Schritt:  $26 - 8 = 18$   
4. Schritt:  $18 - 8 = 10$   
5. Schritt:  $10 - 8 = 2$



Da im fünften Schritt das Ergebnis kleiner ist als die zweite Zahl (8), wird die Berechnung hier beendet, das Ergebnis ist 2.

Dieser Algorithmus funktioniert ebenfalls nur, wenn die erste Zahl die größere der beiden Zahlen ist. Allerdings kann durch die Einbettung in das zuvor beschriebene Rechenwerk und den enthaltenen Teilcode zum Drehen der Zahlen davon ausgegangen werden, dass

diesem Modulo-Rechenwerk immer die korrekte Reihenfolge der Zahlen vorgegeben wird, sodass ein Drehen der Zahlen an dieser Stelle nicht notwendig ist.

## 2. Dokumentation der HDL-Module

### 2.1 Hauptrechenwerk

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**

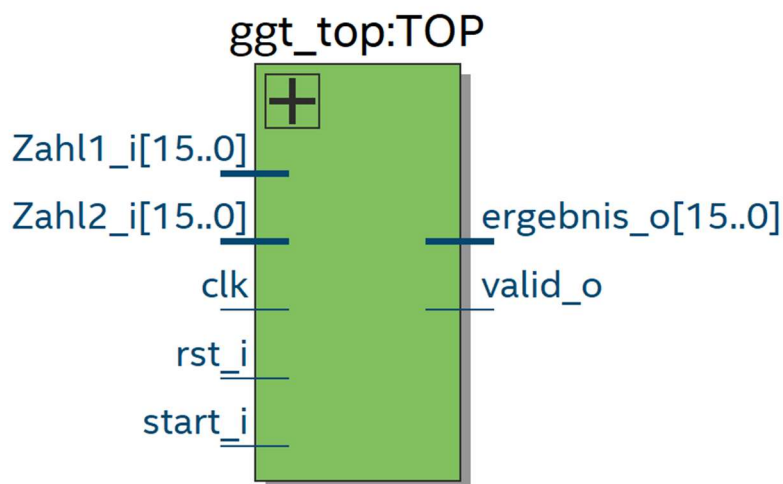


Abbildung 1: Mit dem RTL-Viewer generierter Block des Moduls `ggt_top.v`

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

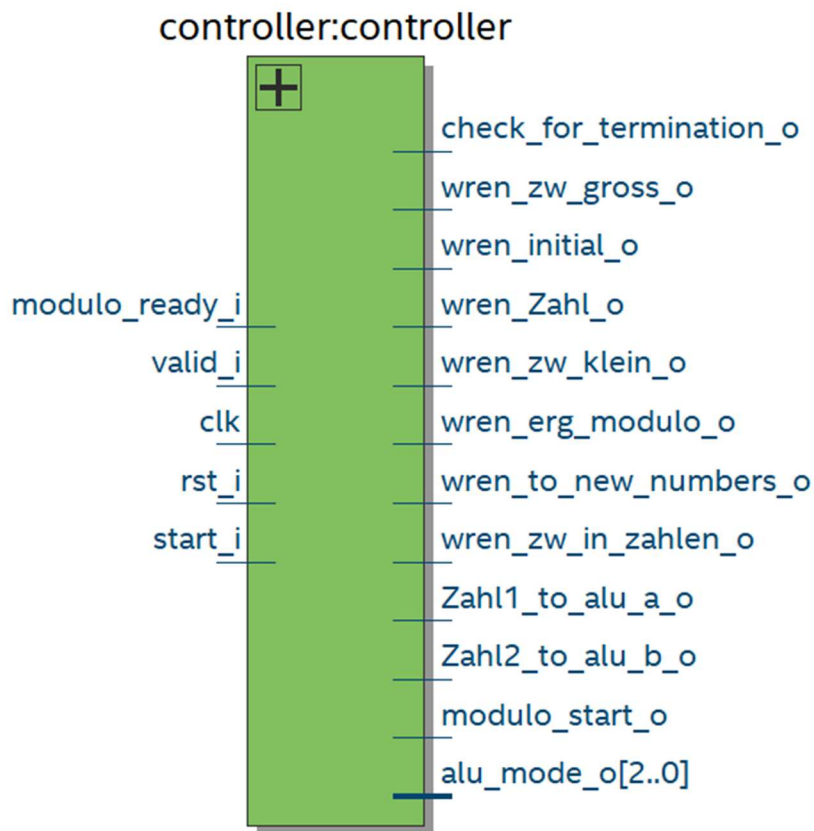


Abbildung 2: Mit dem RTL-Viewer generierter Block des Moduls controller.v

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, to datapath.v	O
alu_mode_o	3 bit output declaring which operation ALU should do, to datapath.v	O

### datapath.v

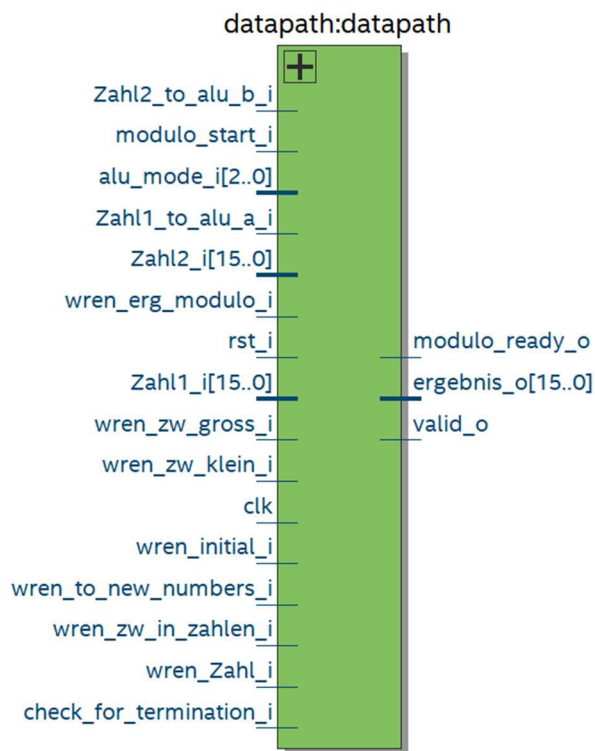


Abbildung 3: Mit dem RTL-Viewer generierter Block des Moduls datapath.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High active reset signal	I
wren_Zahl_i	High active Write-enable signal, from controller.v	I
wren_zw_klein_i	High active Write-enable signal, from controller.v	I
wren_zw_gross_i	High active Write-enable signal, from controller.v	I

wren_erg_modulo_i	High active Write-enable signal, from controller.v	I
wren_to_new_numbers_i	High active Write-enable signal, from controller.v	I
wren_zw_in_Zahlen_i	High active Write-enable signal, from controller.v	I
wren_initial_i	High active Write-enable signal, from controller.v	I
Zahl1_to_alu_a_i	High active Write-enable signal, writing to alu_a, from controller.v	I
Zahl2_to_alu_b_i	High active Write-enable signal, writing to alu_a, from controller.v	I
modulo_start_i	High active start signal to start modulo, from controller.v	I
alu_mode_i	3 bit input declaring which operation ALU should do, from controller.v	I
Zahl1_i	16 bit input for Zahl1	I
Zahl2_i	16 bit input for Zahl2	I
check_for_termination_i	High active signal to check if result of modulo = 0, from controller.v	I
modulo_ready_o	High active signal, if modulo operation is finished, to controller.v	O
valid_o	High active valid signal if ready, to controller.v and ggt_top.v	O
ergebnis_o	16 bit result of operation, to ggt_top.v	O

**alu.v**

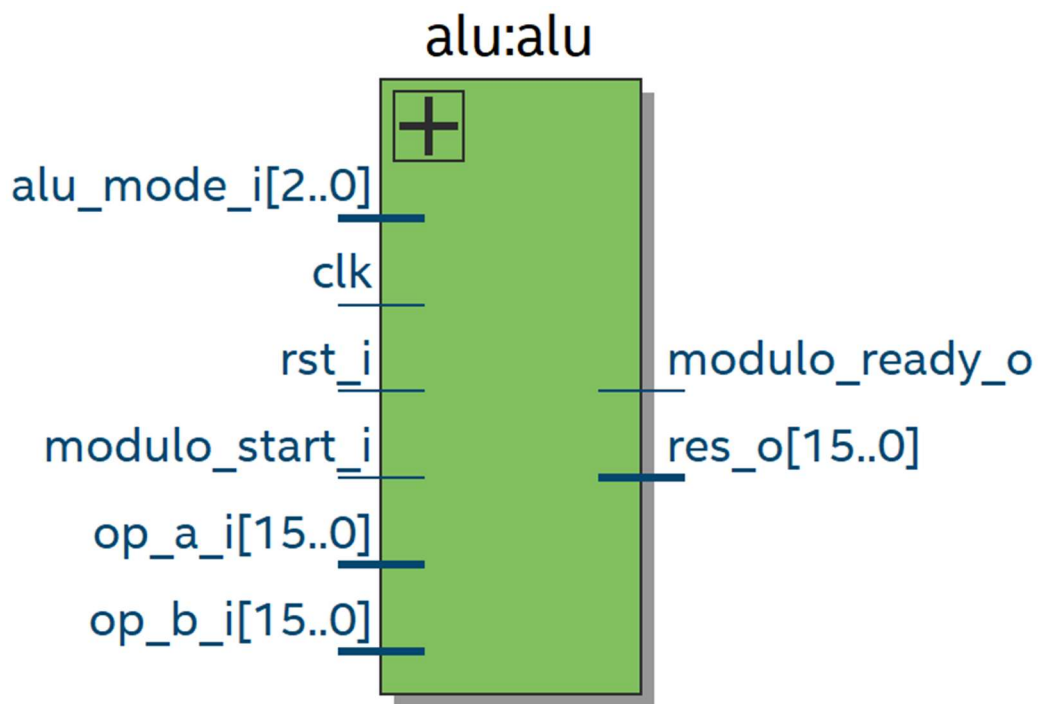


Abbildung 4: : Mit dem RTL-Viewer generierter Block des Moduls ALU.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High active reset signal	I
modulo_start_i	High active start signal to start modulo, from datapath.v	I
alu_mode_i	3 bit input declaring which operation ALU should do, from datapath.v	I
op_a_i	16 bit input for operator A	I
op_b_i	16 bit input for operator B	I
modulo_ready_o	High active signal, if modulo operation is finished, to datapath.v	O
res_o	16 bit operation result output, to datapath.v	O

## 2.2 Modulo-Rechenwerk

**modulo\_top.v**

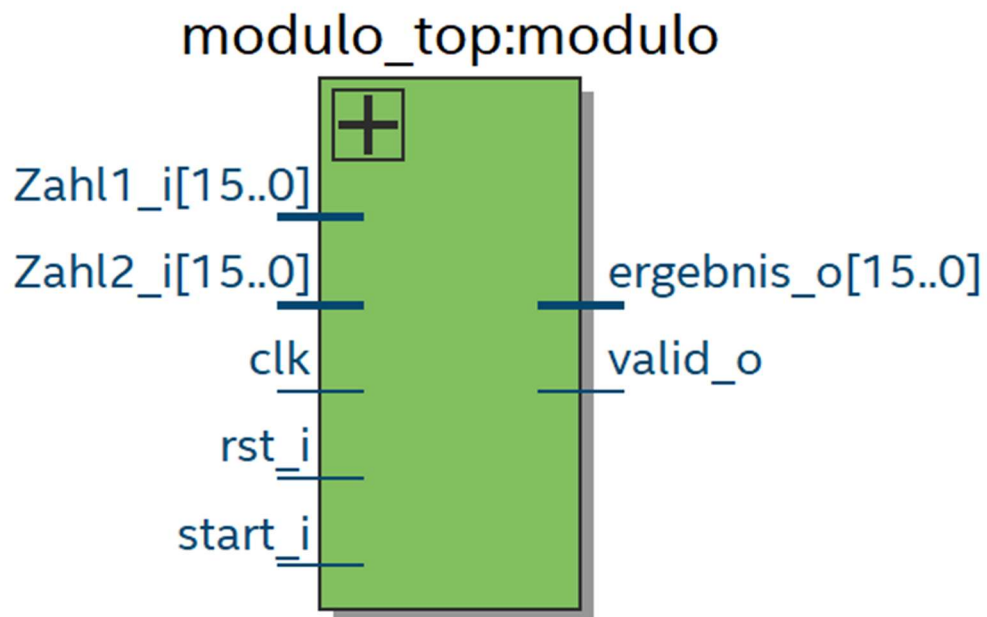


Abbildung 5: Mit dem RTL-Viewer generierter Block des Moduls modulo\_top.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High aktives reset signal	I
start_i	High aktives start signal	I
Zahl1_i	16 bit Eingang für Zahl1	I
Zahl2_i	16 bit Eingang für Zahl2	I
valid_o	High aktives valid Signal, wenn Berechnung erledigt	O
ergebnis_o	16 bit Ausgang, Ergebnis der Berechnung	O

**controller\_modulo.v**

## controller\_modulo:controller

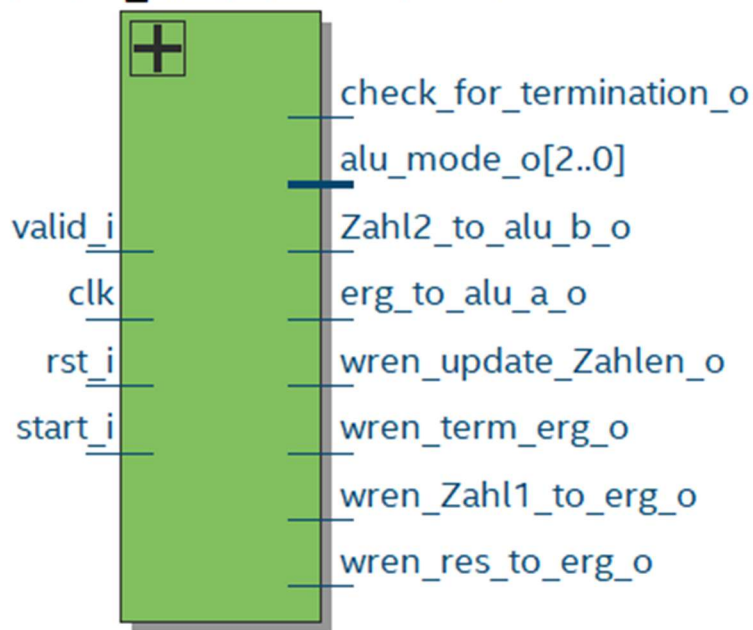


Abbildung 6: Mit dem RTL-Viewer generierter Block des Moduls controller\_modulo.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High aktives reset Signal	I
valid_i	High aktives valid Signal, wenn Berechnung erledigt	I
start_i	High aktives start Signal	I
check_for_termination_o	High aktives signal zum Checken, ob das Ende des Algorithmus eingeleitet werden soll	O
wren_update_Zahlen_o	High aktives Write-enable signal	O
wren_term_erg_o	High aktives Write-enable signal	O
wren_Zahl1_to_erg_o	High aktives Write-enable signal	O
wren_res_to_erg_o	High aktives Write-enable signal	O
erg_to_alu_a_o	High aktives Write-enable signal, schreiben auf alu_a	O
Zahl2_to_alu_b_o	High aktives Write-enable signal, schreiben auf alu_b	O
alu_mode_o	3 bit Ausgang, der beschreibt, welche Operation die ALU durchführen soll	O

## datapath\_modulo.v

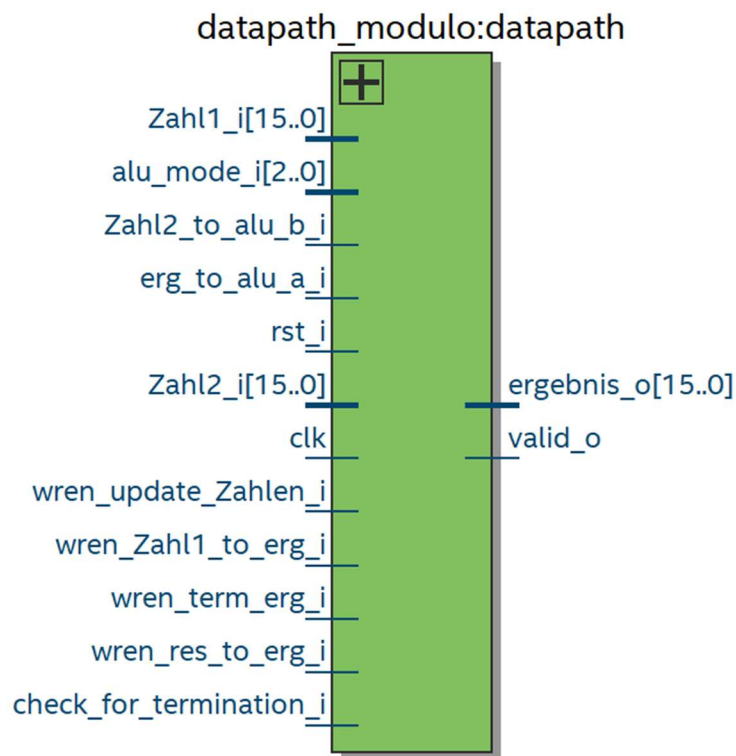


Abbildung 7: Mit dem RTL-Viewer generierter Block des Moduls datapath\_modulo.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i		I
check_for_termination_i	High aktives signal zum Checken, ob das Ende des Algorithmus eingeleitet werden soll	I
wren_update_Zahlen_i	High aktives Write-enable signal	I
wren_term_erg_i	High aktives Write-enable signal	I
wren_Zahl1_to_erg_i	High aktives Write-enable signal	I
wren_res_to_erg_i	High aktives Write-enable signal	I
alu_mode_i	3 bit Eingang, der beschreibt, welche Operation die ALU durchführen soll	I
erg_to_alu_a_i	High aktives Write-enable signal, schreiben auf alu_a	I
Zahl2_to_alu_b_i	High aktives Write-enable signal, schreiben auf alu_b	I
Zahl1_i	16 bit Eingang für Zahl1	I
Zahl2_i	16 bit Eingang für Zahl2	I
valid_o	High aktives valid Signal, wenn Berechnung erledigt	O
ergebnis_o	16 bit Ausgang, Ergebnis der Berechnung	O

**alu\_modulo.v**

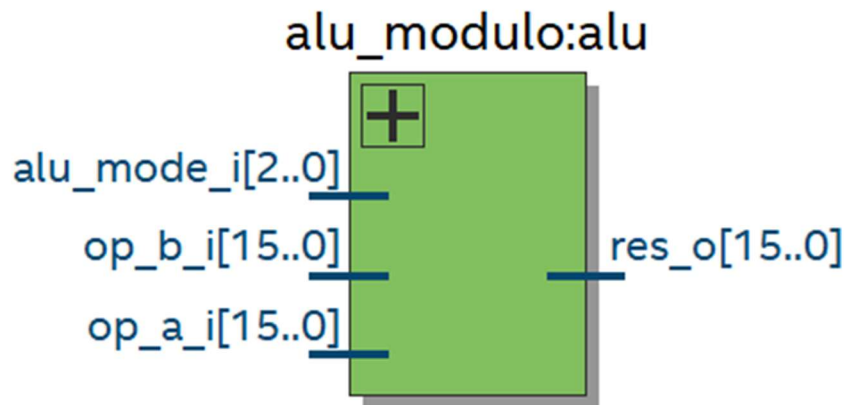


Abbildung 8: Mit dem RTL-Viewer generierter Block des Moduls alu\_modulo.v

Signal	Semantik	Input/Output
clk	Clock	I
rst_i	High aktives reset Signal	I
alu_mode_i	3 bit Eingang, der beschreibt, welche Operation die ALU durchführen soll	I
op_a_i	16 bit Eingang für alu_a	I
op_b_i	16 bit Eingang für alu_b	I
res_o	16 bit Ausgang, Ergebnis der jeweiligen Operation	O

## 3. Verschaltung der HDL-Module

### 3.1 Hauptrechenwerk

#### ggt\_top.v

Im Modul ggt\_top.v werden die beiden Module controller.v und datapath.v instanziiert und über wire miteinander verbunden.



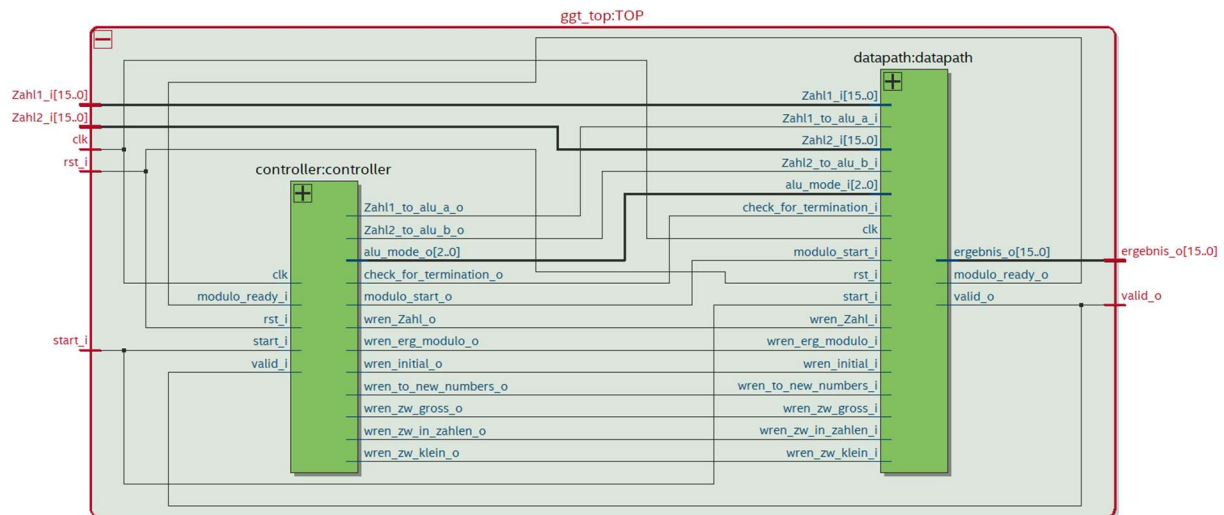


Abbildung 9: Interne Verschaltung im Modul `ggt_top.v`

## datapath.v

Aus Platzgründen wird hier auf die Darstellung des Moduls `datapath.v` verzichtet, denn die Darstellung wäre durch die große interne Schaltung wahrscheinlich kaum erkennbar. Zu erwähnen ist, dass das Modul `alu.v` im `datapath.v` instanziiert wird und dadurch die Werte an die ALU übergeben werden können.

Die Schaltung des `Datapath.v` kann allerdings in den mitgelieferten Quartus-Dateien eingesehen werden.

## alu.v

Auch für die `alu.v` muss auf eine bildliche Darstellung der internen Verschaltung verzichtet werden, obwohl die Darstellung durchaus interessant wäre. Denn die `alu.v` instanziiert das Modulo-Rechenwerk in Form vom Modul `modulo_top.v`. Das Eingangssignal `modulo_start_i` des Moduls `alu.v` wird an den Eingangsport `start_i` der `modulo_top.v` übergeben und die beiden angelegten Operanden `op_a_i` und `op_b_i` werden an die Ports `Zahl1_i` und `Zahl2_i` der `modulo_top.v` übergeben. Der Controller des Hauptrechenwerks bleibt dann so lange im aktuellen Schritt, bis das Modulo-Rechenwerk über `valid_o` das Signal gibt, mit der Berechnung fertig zu sein. Nun wird das Ergebnis dem `datapath.v` übergeben und der nächste Schritt wird eingeleitet.

## 3.2 Modulo-Rechenwerk

### modulo\_top.v

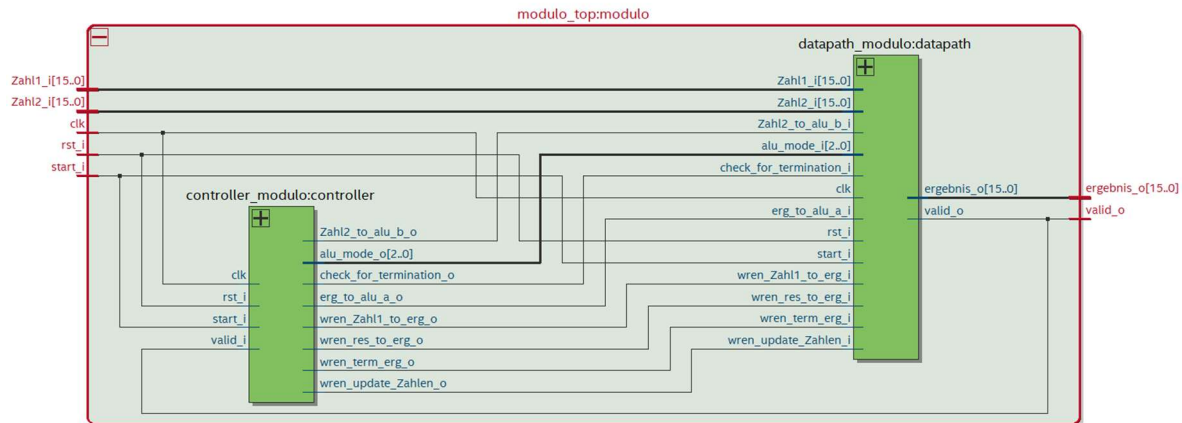


Abbildung 10: Interne Verschaltung im Modul `modulo_top.v`

Die Datei `modulo_top.v` instanziiert ähnlich zur `ggt_top.v` die beiden Module `controller_modulo.v` und `datapath_modulo.v` und verbindet diese mittels einiger wires.

### datapath\_modulo.v

Auch hier muss auf eine Darstellung der Schaltung verzichtet werden, da diese durch die enthaltene Logik zu umfangreich wäre, um sie bildlich darzustellen.

Die `datapath_modulo.v` instanziiert ebenfalls die ALU (`alu_modulo.v`). Die Schaltung kann ebenfalls in den Quartus-Dateien im RTL-Viewer eingesehen werden.

## 4. Test und Verifikation

### 4.1 Funktionstests mit Hilfe von ModelSim

Im ersten Schritt der Verifikation der Funktion wurde eine `testbench.v` geschrieben. Diese `testbench.v` instanziiert die `ggt_top.v` und übergibt dieser zwei vorgegebene, fest einprogrammierte Zahlenwerte (einen für `Zahl1`, den anderen für `Zahl2`). Weiterhin sorgt die Testbench dafür, dass wichtige Steuersignale, wie `clk`, `start_i` oder `rst_i` an die `ggt_top.v` übergeben werden.

Bevor eine Analyse mit Matlab möglich war, wurden nun erstmal mit der `testbench.v` einzelne Werte an die `ggt_top.v` übergeben, indem diese Werte im Code geschrieben wurde, das Projekt in ModelSim kompiliert und anschließend simuliert wurde. Am Beispiel der Zahlen 24255 und 12540 wird dies nun gezeigt:

```

module testbench(

);

    reg [15:0] Zahl1 = 16'd24255;
    reg [15:0] Zahl2 = 16'd12540;
    wire [15:0] ergebnis;
    reg start = 0;
    reg clk = 0;
    reg rst = 0;
    wire valid;

```

Abbildung 11: Initialisierung der beiden Zahlen im Code mittels eines festgeschriebenen Wertes

In dieser Abbildung sieht man, dass die beiden Werte 24255 und 12540 fest vorgegeben sind. Sie werden zur Laufzeit nicht mehr verändert.

Mit Hilfe eines online-Rechners ([ggT-Rechner - Matheretter](#)) wird vor der Simulation das erwartete Ergebnis ermittelt, dieses beträgt laut dem Rechner 165.

Im folgenden Schritt wird die Simulation mit ModelSim durchgeführt.

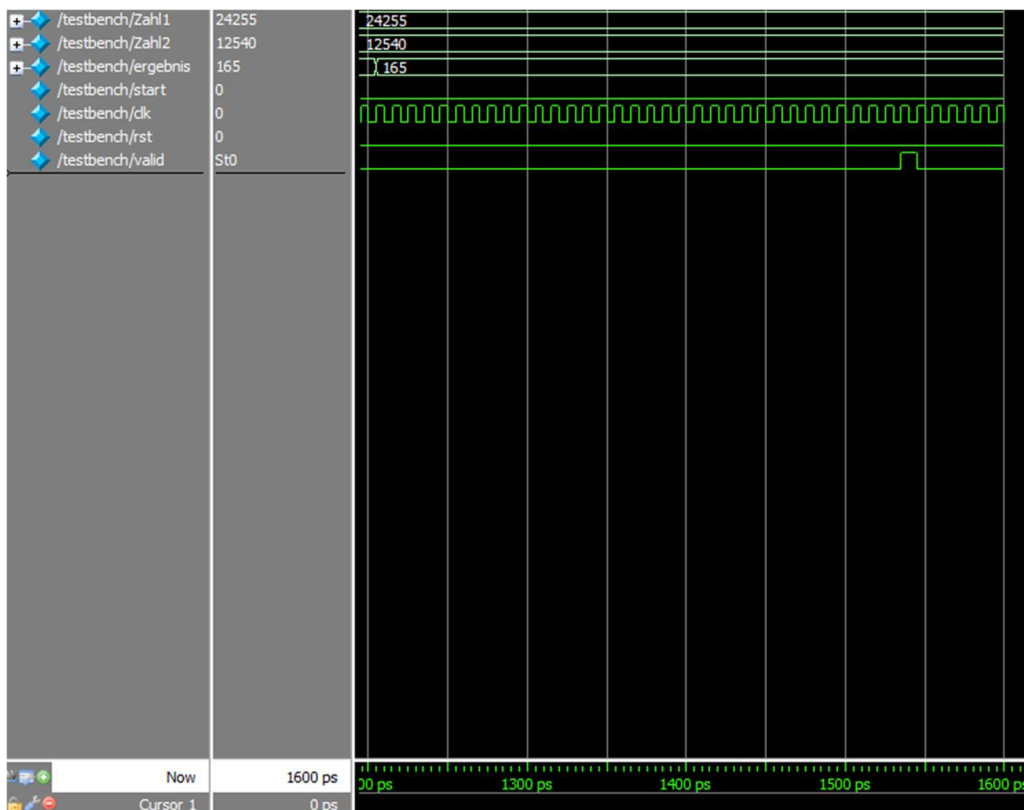


Abbildung 12: Durchgeführte Simulation in ModelSim mit dem Ergebnis 165, signalisiert durch steigende valid-Flanke

In dieser Abbildung ist zu sehen, dass bei der korrekten Reihenfolge der beiden Eingabezahlen das korrekte Ergebnis 165 nach etwa 1530 ps anliegt. Dabei gilt ein Hauptaugenmerk der steigenden Flanke des valid-Signals, da dieses signalisiert, dass das nun anliegende Ergebnis das korrekte ist. Dies ist notwendig, da auch schon vorher Zwischenergebnisse auf den Ausgangsport `ergebnis_o` geschrieben werden, diese sind aber noch nicht korrekt. Valid signalisiert, dass die Berechnung abgeschlossen und das Ergebnis das richtige ist.

Diese Simulation wird nun erneut durchgeführt, aber mit der vertauschten Eingabe der Zahlen, sodass `Zahl1` die kleinere und nicht wie üblich die größere der beiden Zahlen ist. Die Erwartungshaltung ist hier, dass das gleiche Ergebnis herauskommt.

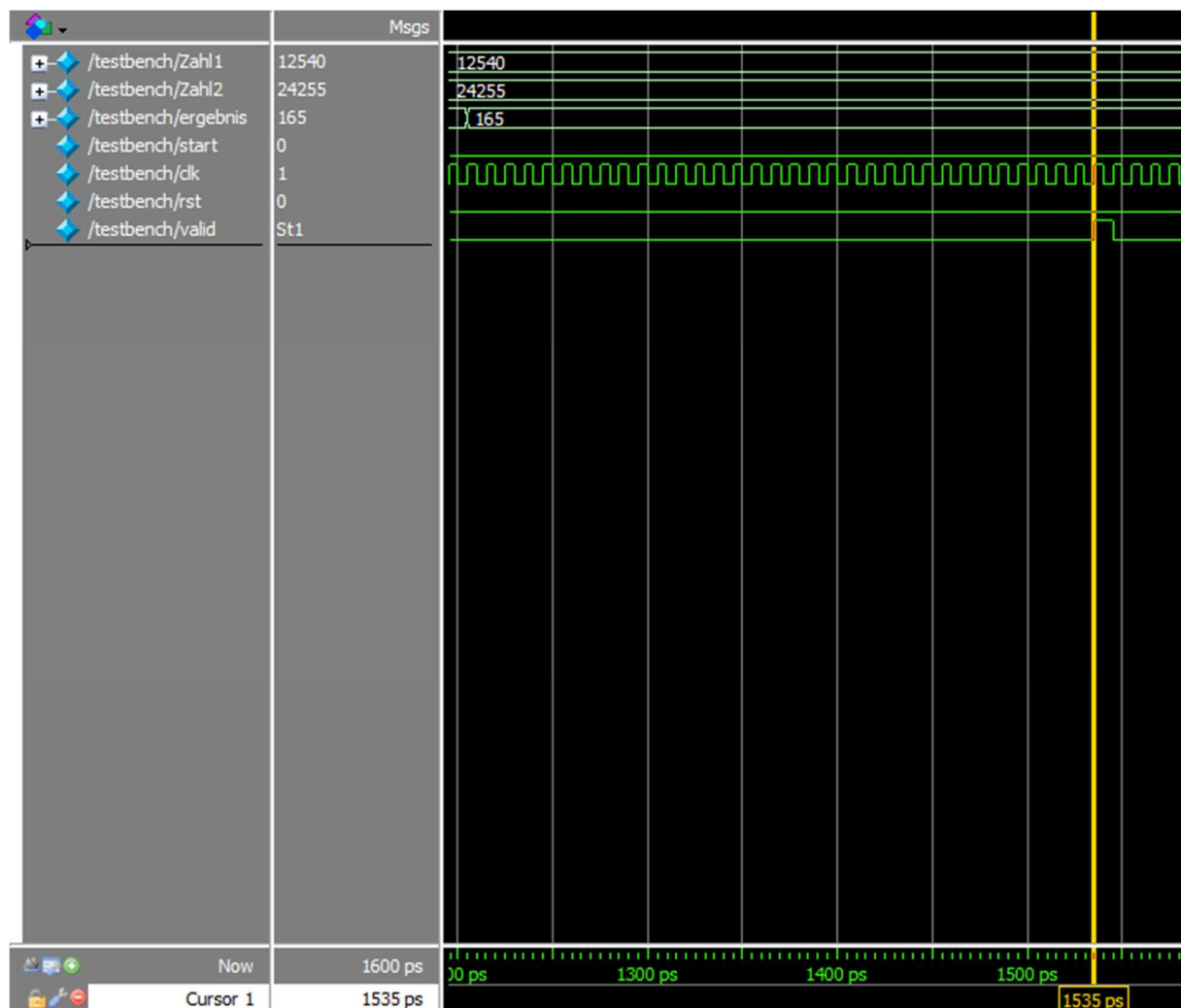


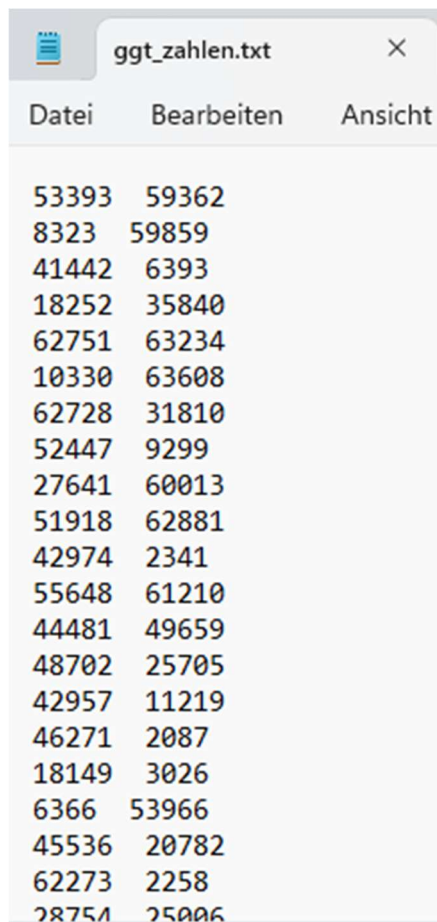
Abbildung 13: Durchgeführte Simulation mit vertauschten Eingaben in ModelSim mit dem Ergebnis 165, signalisiert durch steigende `valid`-Flanke

So sieht man auch in dieser Abbildung, dass das Ergebnis 165 nach 1535 ps korrekt am Ausgang anliegt. Dieser Simulationsaufbau wird nun für folgende Zahlen wiederholt:

Zahl1	Zahl2	Erwartetes Ergebnis	Korrekt in Sim?
554	192	2	korrekt
192	554	2	korrekt

728	259	7	korrekt
259	728	7	korrekt
944	648	8	korrekt
648	944	8	korrekt

Da alle Werte korrekt berechnet werden, kann auf Vorbehalt von einer richtigen Funktion ausgegangen werden. Dieser Test soll nun mit Hilfe von Matlab erweitert werden. Dafür wurde ein Matlab Skript (Matlab\_GGT.m) geschrieben, welches zufällige Zahlenpaare in eine Textdatei namens „ggt\_zahlen.txt“ schreibt. Die Anzahl der Zahlenpaare kann eigens festgelegt werden und wird hier auf 1000 gesetzt. Gleichzeitig wird zu jedem Zahlenpaar der ggT mittels Matlab-Funktion gcd() (engl. greatest common divisor) berechnet und zeilenweise in eine zweite Textdatei namens „ggt\_ergebnisse.txt“ geschrieben. Die beiden Textdateien sehen nach einmaligem Durchlaufen des Skriptes folgendermaßen aus:



Datei	Bearbeiten	Ansicht
53393	59362	
8323	59859	
41442	6393	
18252	35840	
62751	63234	
10330	63608	
62728	31810	
52447	9299	
27641	60013	
51918	62881	
42974	2341	
55648	61210	
44481	49659	
48702	25705	
42957	11219	
46271	2087	
18149	3026	
6366	53966	
45536	20782	
62273	2258	
28751	25006	

Abbildung 14: Zufallszahlenpaare in ggt\_zahlen.txt

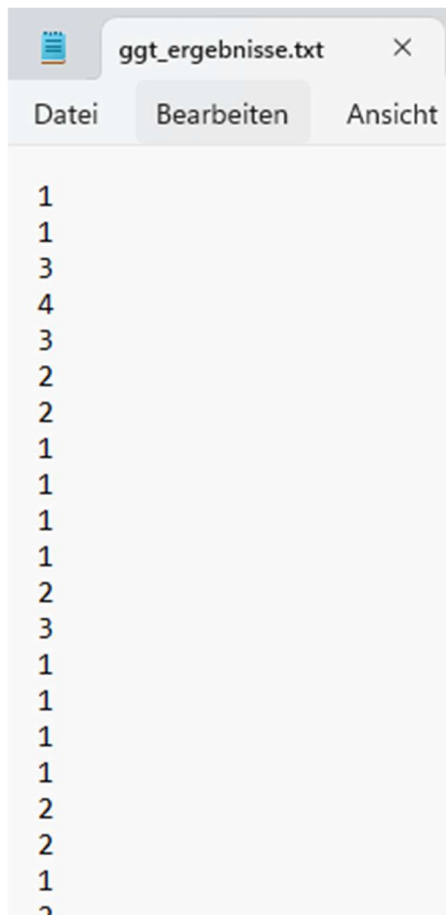


Abbildung 15: ggt-Ergebnisse der Zahlenpaare, berechnet mit der Matlab-Funktion gcd()

Nun wird auch die Testbench entsprechend so erweitert, dass sie die Textdatei „ggt\_zahlen.txt“ öffnen kann und die Werte zeilenweise aus dieser Datei liest, an ggt\_top.v übergibt und anschließend auf das valid-Signal wartet, bevor die nächste Zeile gelesen wird. Kommt das valid-Signal, wird der ergebnis-Wert aus der ggt\_top.v gelesen und in eine dritte Textdatei namens „ggt\_ergebnisse\_euklid.txt“ geschrieben. Ebenfalls zeilenweise. Die Testbench tut dies in einer Schleife so lange, bis eof, also End of File erreicht ist. Dann werden die beiden Textdateien geschlossen und die Simulation beendet.

In der Theorie müssten die beiden Textdateien „ggt\_ergebnisse.txt“ und „ggt\_ergebnisse\_euklid.txt“ exakt gleichen Inhalt haben. Da 1000 Zeilen aber manuell nicht verglichen werden können, wird ein zweites Matlab-Skript (Vergleich.m) geschrieben, welches die beiden Textdateien miteinander vergleicht und dem Anwender eine Zusammenfassung über die Gleichheit gibt. Die Ausgabe sieht im Ausgabe-Fenster von Matlab folgendermaßen aus:

```
Command Window

Zeile 967: ✓ Übereinstimmung
Zeile 968: ✓ Übereinstimmung
Zeile 969: ✓ Übereinstimmung
Zeile 970: ✓ Übereinstimmung
Zeile 971: ✓ Übereinstimmung
Zeile 972: ✓ Übereinstimmung
Zeile 973: ✓ Übereinstimmung
Zeile 974: ✓ Übereinstimmung
Zeile 975: ✓ Übereinstimmung
Zeile 976: ✓ Übereinstimmung
Zeile 977: ✓ Übereinstimmung
Zeile 978: ✓ Übereinstimmung
Zeile 979: ✓ Übereinstimmung
Zeile 980: ✓ Übereinstimmung
Zeile 981: ✓ Übereinstimmung
Zeile 982: ✓ Übereinstimmung
Zeile 983: ✓ Übereinstimmung
Zeile 984: ✓ Übereinstimmung
Zeile 985: ✓ Übereinstimmung
Zeile 986: ✓ Übereinstimmung
Zeile 987: ✓ Übereinstimmung
Zeile 988: ✓ Übereinstimmung
Zeile 989: ✓ Übereinstimmung
Zeile 990: ✓ Übereinstimmung
Zeile 991: ✓ Übereinstimmung
Zeile 992: ✓ Übereinstimmung
Zeile 993: ✓ Übereinstimmung
Zeile 994: ✓ Übereinstimmung
Zeile 995: ✓ Übereinstimmung
Zeile 996: ✓ Übereinstimmung
Zeile 997: ✓ Übereinstimmung
Zeile 998: ✓ Übereinstimmung
Zeile 999: ✓ Übereinstimmung
Zeile 1000: ✓ Übereinstimmung

✓ Die beiden Dateien sind vollständig identisch!
fx >>
```

Abbildung 16: Ausgabe des Matlab-Skripts Vergleich.m

Erfreulicherweise sind alle Werte als gleich bestimmt, was bedeuten würde, dass das Rechenwerk korrekt funktioniert und keinerlei Fehler ausweist.

Um das Vergleichsskript einem kleinen Test zu unterziehen, werden in einer der beiden Textdateien nun drei Werte per Hand verändert (zwischen Zeile 970 und Zeile 1000, sodass dies in der Ausgabe des Skripts gut sichtbar ist).



```
Command Window
Zeile 971: ✓ Übereinstimmung
Zeile 972: ✓ Übereinstimmung
Zeile 973: ✓ Übereinstimmung
Zeile 974: ✓ Übereinstimmung
Zeile 975: ✓ Übereinstimmung
Zeile 976: ✓ Übereinstimmung
Zeile 977: ✓ Übereinstimmung
Zeile 978: ✓ Übereinstimmung
Zeile 979: ✓ Übereinstimmung
Zeile 980: ✓ Übereinstimmung
Zeile 981: ✓ Übereinstimmung
Zeile 982: ✓ Übereinstimmung
Zeile 983: ✓ Übereinstimmung
Zeile 984: ✓ Übereinstimmung
Zeile 985: ✗ Unterschied gefunden
    Datei 1: 1
    Datei 2: 3
Zeile 986: ✓ Übereinstimmung
Zeile 987: ✓ Übereinstimmung
Zeile 988: ✓ Übereinstimmung
Zeile 989: ✓ Übereinstimmung
Zeile 990: ✓ Übereinstimmung
Zeile 991: ✓ Übereinstimmung
Zeile 992: ✗ Unterschied gefunden
    Datei 1: 2
    Datei 2: 1
Zeile 993: ✓ Übereinstimmung
Zeile 994: ✓ Übereinstimmung
Zeile 995: ✓ Übereinstimmung
Zeile 996: ✓ Übereinstimmung
Zeile 997: ✓ Übereinstimmung
Zeile 998: ✓ Übereinstimmung
Zeile 999: ✗ Unterschied gefunden
    Datei 1: 4
    Datei 2: 3
Zeile 1000: ✓ Übereinstimmung

⚠ Die Dateien haben 3 Unterschiede.
fx >>
```

Abbildung 17: Ausgabe des Matlab-Skripts Vergleich.m mit drei manuell veränderten Werten

Wie man in dieser Abbildung sieht, erkennt das Skript die drei manuell veränderten Werte. Dies verdeutlicht, dass es ordnungsgemäß funktioniert und die Werte aus der Matlab-Funktion gcd() und aus dem Rechenwerk übereinstimmen.

Die beiden Matlab-Skripte und die drei Textdateien liegen im Ordner Simulation\_Hauptrechenwerk, damit die Simulation in ModelSim auf die Textdateien zugreifen kann. Das Simulationsprojekt ist im gleichen Ordner unter Sim\_gesamt.mpf zu finden.



## 4.2 Funktionstests auf dem realen FPGA

Die aktuelle Testbench war mit den beinhalteten File-Handlings, der wait-Funktion und \$stop nicht kompilierbar. Aus diesem Grund wurde die testbench.v weiter umgebaut. Es wurde nun eine define-Präprozessor-Direktive eingebaut, mit der verschiedene Code-Teile „ein- oder ausgeblendet“ werden können. Die Unterteilung des Codes wird nach folgendem Schema vorgenommen:

```
`define SIMULATION; // Diese Zeile auskommentieren für Quartus

`ifndef SIMULATION
module testbench(
    input clk
);
    //hier der Quartus-Code

endmodule
`endif

`ifdef SIMULATION
module testbench(
);
    //hier der Code für die Simulation mit ModelSim

endmodule
`endif
```

Abbildung 18: Schema, nach dem die Unterteilung der testbench.v vorgenommen wird

So kann die Testbench.v für beide Zwecke genutzt werden und bleibt trotzdem übersichtlich.

Für den Code für Quartus Prime wurde nun noch eine ALTPLL sowie ein RAM-Block hinzugefügt, in den das Ergebnis geschrieben werden soll.

Auf dem realen FPGA ist eine automatisierte Überprüfung wie mit Matlab leider nicht möglich, daher muss wieder auf eine manuelle Prüfung zurückgegriffen werden, die etwas zeitintensiver ist, aber ihren Zweck ebenso erfüllt.

Also wurde folgende Tabelle aufgesetzt mit verschiedenen Zahlen. Hier wurde sowohl das Verdrehen berücksichtigt als auch gleiche Zahlen als Eingabe. Es wurde ebenso das erwartete Ergebnis in Hexadezimal-Darstellung ergänzt, da der RAM-Block das Ergebnis als Hex-Code ausgibt.

Zahl1	Zahl2	erwartetes Ergebnis (hex)	korrekt?
554	192	2	korrekt
192	554	2	korrekt
728	259	7	korrekt
259	728	7	korrekt
944	648	8	korrekt
648	944	8	korrekt
180	90	5A	korrekt
90	180	5A	korrekt
180	180	B4	korrekt
240	95	5	korrekt
95	240	5	korrekt
744	685	1	korrekt
685	744	1	korrekt

Auch hier sind alle Werte, die in den RAM-Block geschrieben werden, korrekt und erfüllen die Erwartungen.

## 4.3 Timing-Analyse

### 4.3.1 Mit ModelSim

Eine Timing-Analyse kann in ModelSim relativ einfach durchgeführt werden. Hierfür wird zuallererst die eingestellte Frequenz der Clock berechnet. Diese berechnet sich wie folgt:

$$f_{clock} = \frac{1}{T_{clock}} = \frac{1}{5 \text{ ps}} = 200 \text{ GHz}$$

Anhand folgender drei Beispiele, gemessen in ModelSim soll gezeigt werden, wie schnell die Schaltung arbeitet:

Zahl1	Zahl2	Zeit von start bis valid
53393	59362	3.715 ps
59859	8323	3.668 ps
41442	6393	2.518 ps
63520	1	3,176145 ms
400	200	248 ps

Die fünf Beispiele zeigen, dass die Zeit, die vom Signal start bis zum Signal valid vergeht, in einem gewissen Maße variieren kann. Diese Zeit hängt stark von der Wahl der Zahlen ab, wie man bei den letzten beiden Beispielen sehen kann. Das Beispiel mit den Zahlen 63520 und 1 benötigt deutlich mehr Durchläufe, um die Berechnung abzuschließen, wohingegen die Berechnung mit den Zahlen 400 und 200 sehr schnell durchgeführt wird.

Trotzdem ist auch die gezeigte sehr intensive Berechnung mit etwa 3 ms durchaus schnell ausgeführt worden.

An dieser Stelle ist zu bemerken, dass diese Simulation zusätzlich das Schreiben und Lesen in den Textdateien sowie das Schreiben mittels  $\$display$  beinhaltet. Ohne diese Operationen würde die Berechnung vermutlich noch um einiges schneller ablaufen können.

### 4.3.2 Mit Quartus

In Quartus Prime lässt sich die Durchlaufzeit mit dem aktuellen Aufbau leider nicht so einfach bestimmen.

Die verwendete ALTPLL nutzt eine Input-Taktrate von 12 MHz und teilt diese so, dass die ausgehende und anschließend verwendete `logic_clk` eine Taktrate von 1MHz besitzt.

## 4.4 Analyse des Stromverbrauchs mit Quartus Prime

Power Analyzer Summary	
<<Filter>>	
Power Analyzer Status	Successful - Sat Mar 15 14:55:31 2025
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	Euklidischer_Algorithmus
Top-level Entity Name	testbench
Family	MAX 10
Device	10M08SAU169C8G
Power Models	Final
Total Thermal Power Dissipation	131.61 mW
Core Dynamic Thermal Power Dissipation	1.34 mW
Core Static Thermal Power Dissipation	121.02 mW
I/O Thermal Power Dissipation	9.26 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data