HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Fakultät Ingenieurwissenschaften

Schaltkreisentwurf

Packet Detector

Author Nico Beyer (nico.beyer@htwk-leipzig.de)
Betreuer Professor Krondorf

15. März 2025

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit der Implementierung eines Paketdetektor-Algorithmus in Verilog. Der entwickelte Algorithmus dient dazu, digitale Signale auf das Vorhandensein eines definierten Präambels zu untersuchen und anhand dessen den Beginn des Payloads zu identifizieren. Damit bildet er eine wichtige Grundlage in der digitalen Signalverarbeitung und Kommunikationssystemen.

Ein wesentlicher Unterschied zu klassischen Paketdetektoren besteht darin, dass die Eingangsdaten nicht als digitale Signale im binären Format (0 und 1) vorliegen, sondern als komplexwertige Eingangssignale. Diese erfordern eine speziell angepasste Verarbeitung, da sowohl Real- als auch Imaginärteile berücksichtigt werden müssen.

Ziel dieser Arbeit ist es, einen robusten und effizienten Paketdetektor zu realisieren, der die Anforderungen an Echtzeitfähigkeit und Ressourcennutzung erfüllt.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist die möglichst effiziente Umsetzung eines Packet Detectors in Verilog. Im Kern soll der Algorithmus ein fest einprogrammiertes Präambel erkennen, welches nicht verändert werden kann. Dabei wird besonderer Wert auf die Echtzeitfähigkeit des Systems gelegt, sodass der Detektor in der Lage ist, Datenpakete in laufenden Signalen ohne allzu große Verzögerungen zu identifizieren.

Da die Länge des Payloads flexibel gestaltet werden kann, beinhaltet die Zielsetzung auch die dynamische Auswertung der Payloadlänge. Für jede Erkennung des Präambels wird die Länge des Präambels ermittelt und ausgegeben, um so eine Weiterverarbeitung der empfangenen Daten zu ermöglichen.

2 Theoretische Grundlagen und Konzept

Bei der digitalen Paketdetektion erfolgt die Erkennung durch die Überprüfung der Reihenfolge von zwei Zuständen, was einen relativ einfachen Erkennungsprozess ermöglicht. Im Gegensatz dazu sind die Eingangssignale in diesem Projekt komplexwertig und können durch Messrauschen zusätzlich verfälscht werden, sodass keine exakt identischen Eingangspakete vorliegen. Daher ist es erforderlich, eine Korrelationsmetrik zu nutzen, um die zuverlässige Erkennung der komplexen Signale zu gewährleisten. Hierbei kommt die Kreuzkorrelation zum Einsatz, die als Maß zur Bestimmung der Ähnlichkeit zwischen dem empfangenen Signal und einem vorher definierten Präambel dient. Diese wird im Folgenden als Metrik bezeichnet und wie in Gleichung 1 berechnet:

$$\mathrm{m[k]} = \frac{1}{L_P} \sum_{l=0}^{L_p-1} y(k-l) \cdot s_p^*(L_p - l - 1) \tag{1}$$

Zur Bewertung, ob eine Präambel erkannt wurde, wird für jedes neue Sample die berechnete Metrik analysiert. Eine Präambel liegt vor, wenn der Metrik-Wert signifikant vom Durchschnitt der bisherigen Metrik-Berechnungen abweicht. Im einfachsten Fall kann hierfür ein Schwellwert definiert werden, ab dessen Überschreiten die Präambel als erkannt gilt.

Der Schwellwert muss so gewählt werden, dass Fehlklassifikationen vermieden werden. Wird der Schwellwert zu niedrig angesetzt, können zufällige Nicht-Präambel-Samples fälschlicherweise als Präambel interpretiert werden. Wird er hingegen zu hoch festgelegt, besteht die Gefahr, dass echte Präambeln aufgrund eines zu geringen Metrik-Werts nicht erkannt werden. Daher ist die Wahl des Schwellwerts entscheidend für die korrekte Paketerkennung.

Aufgrund der unterschiedlichen Energiepegel und Rauschbeiträge der Eingangssignale können gleiche Signalstrukturen in unterschiedlichen Amplituden auftreten, was zu variierenden Metrikwerten führt. Daher ist es erforderlich, die Metrik vor der Auswertung zu normalisieren. Durch diese Normalisierung wird sichergestellt, dass der festgelegte Schwellwert nicht nur für ein spezifisches Signal, sondern generell anwendbar ist. Die Normalisierung erfolgt durch Division der Metrik durch den quadratischen Mittelwert der Energie des Signals. Die normierte Metrik wird dabei wie folgt berechnet:

$$m'[k] = \frac{|m[k]|^2}{\sigma_y^2} \tag{2}$$

Die Berechnung des quadratischen Mittelwerts der Energie des Signals ist durch Gleichung 3 definiert.

$$\sigma_y^2 = \frac{1}{L_P} \sum_{l=0}^{L_P-1} |y(k-l)|^2 \tag{3}$$

Durch diese Normalisierung wird sichergestellt, dass die Metrikwerte unabhängig von der Signalstärke vergleichbar bleiben (siehe Gleichung (3)).

Um zu überprüfen, ob eine Präambel erkannt wurde, wird die normierte Metrik mit dem vorgegebenen Schwellwert T verglichen. Eine Präambel wird dann als erkannt angesehen, wenn

$$m'[k] > T \tag{4}$$

gilt, wobei T den definierten Schwellwert darstellt.

2.1 Änderungen an der Ungleichung

Um die Berechnung möglichst effizient auf einem FPGA ausführen zu können, wurden Modifikationen an der in Kapitel 2 beschriebenen Berechnung vorgenommen. Zunächst wurde die Gleichung zur Berechnung der normierten Metrik betrachtet:

$$m'[k] = \frac{|m[k]|^2}{\sigma_y^2} = \frac{\left| \frac{1}{L_P} \sum_{l=0}^{L_P-1} y(k-l) \cdot s_p^*(L_P-l-1) \right|^2}{\frac{1}{L_P} \sum_{l=0}^{L_P-1} |y(k-l)|^2} \tag{5}$$

Hierbei ist zu erkennen, dass der Faktor L_p sowohl im Zähler als auch im Nenner vorkommt und daher gekürzt werden kann. Dies führt zur vereinfachten Form:

$$m'[k] = \frac{\left| \sum_{l=0}^{L_P-1} y(k-l) \cdot s_p^*(L_P-l-1) \right|^2}{L_P \sum_{l=0}^{L_P-1} |y(k-l)|^2} \tag{6}$$

Diese Umformung erhöht die Effizienz der Berechnung, was insbesondere bei der Implementierung auf einem FPGA von Vorteil ist. Da Divisionen in Hardware entweder ressourcenintensiv oder zeitaufwändig sind, wurde die finale Ungleichung entsprechend umgestellt, um diesen Rechenaufwand zu vermeiden.

$$\frac{|\mathbf{m}[k]|^2}{\sigma_y^2} > T \tag{7}$$

Durch Umformung ergibt sich:

$$|\mathbf{m}[k]|^2 > T \cdot \sigma_y^2 \tag{8}$$

Schließlich führt dies zur vereinfachten Darstellung:

$$\left| \sum_{l=0}^{L_P-1} y(k-l) \cdot s_p^*(L_P - l - 1) \right|^2 > T \cdot L_P \cdot \sum_{l=0}^{L_P-1} |y(k-l)|^2 \tag{9}$$

Diese Umformung ermöglicht eine effizientere Berechnung, insbesondere für eine FPGA-Implementierung, da auf eine Division verzichtet werden kann.

2.2 Festlegungen

Da sich auf der rechten Seite der Ungleichung die Multiplikation der Länge der Präambel (L_P) mit dem Schwellwert (T) befindet und beide während der Laufzeit nicht geändert werden müssen, kann diese Multiplikation bereits vor der Synthese berechnet und direkt im Programcode festgelegt werden. Bei der Analyse des von MATLAB erstellten Graphen zum ersten Datensatz, welcher in Abbildung 1 dargestellt ist, zeigt sich:

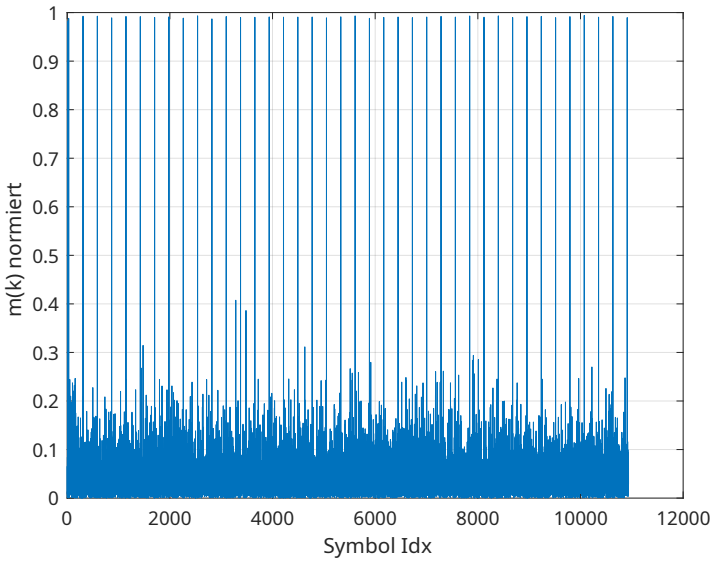


Abbildung 1: Graphische Darstellung der normierten Metrikwerte mittel Matlab (test_signal_1.mat)

Die Spitzen im Graphen markieren die erkannten Samples, die sich deutlich von den übrigen berechneten Metriken abheben. Daher würde ein Schwellwert von etwa 0,9 ausreichen, um die Präambel sicher zu erkennen. Daraus ergibt sich:

$$T \cdot L_P = 0,9 \cdot 23 = 20,7 \tag{10}$$

Der Wert 20,7 liegt nahe bei 16. Dies hat den Vorteil, dass bei der Multiplikation und Division mit einer ganzzahligen Zweierpotenz die Multiplikation in eine Bit-Shift-Operation überführt werden

kann. Diese Operation ist auf FPGAs äußerst effizient und erfordert nur minimalen Hardwareaufwand. Da die Länge der Präambel L_P festgelegt ist, wurde ein neuer Schwellwert gesucht, sodass das Produkt $L_P \cdot T$ exakt 16 ergibt.

Mit der folgenden Formel kann der benötigte Schwellwert berechnet werden:

$$T = \frac{16}{L_P} = \frac{16}{23} = 0,695 \tag{11}$$

Dadurch wird der Schwellwert zwar deutlich kleiner, was jedoch für den ersten Datensatz unproblematisch ist, da hier das restliche Signal deutlich unter 0,7 und die detektierten Präambeln deutlich über 0,7 liegen. Somit wurde eine äußerst effiziente Alternative zur herkömmlichen Multiplikation gefunden. Allerdings hat diese Methode den Nachteil, dass der Schwellwert nicht mehr flexibel wählbar ist.

2.3 Umgang mit komplexen Zahlen

Da die Eingangssamples als komplexe Zahlen vorliegen und nicht als reine reelle Werte, ist eine spezielle Behandlung notwendig. In Verilog existiert kein direkter Datentyp für komplexe Zahlen, wie dies beispielsweise in MATLAB der Fall ist. Daher werden komplexe Zahlen durch zwei separate Variablen dargestellt – eine für den Realteil und eine für den Imaginärteil. Diese Vorgehensweise führt zu einem zusätzlichen Rechenaufwand, da Operationen wie Addition bei komplexen Zahlen jeweils separat für beide Anteile durchgeführt werden müssen.

Ein großes Problem stellt jedoch die Berechnung des Betrags einer komplexen Zahl dar. Diese würde nach folgender Gleichung erfolgen:

$$|z| = \sqrt{\text{Real}^2 + \text{Imag}^2} \tag{12}$$

Das Hauptproblem dieser Berechnung ist die Wurzeloperation, da diese auf einem FPGA mit erheblichem Rechenaufwand und großem Ressourcenverbrauch verbunden ist. Daher wurde eine alternative Methode zur Approximation des Betrags einer komplexen Zahl verwendet, um die Berechnung zu vereinfachen und zu beschleunigen. Diese Methode basiert auf dem Verfahren von:

https://dspguru.com/dsp/tricks/magnitude-estimator/

Anstelle der exakten Berechnung wird hierbei der größere der beiden Werte (Realteil oder Imaginärteil) mit einem Faktor α multipliziert und der kleinere mit einem Faktor β. Anschließend werden beide addiert:

$$|z| \approx \alpha \cdot \max(|\text{Real}|, |\text{Imag}|) + \beta \cdot \min(|\text{Real}|, |\text{Imag}|) \tag{13}$$

Die Faktoren α und β werden so gewählt, dass sie ganzzahlige Zweierpotenzen sind. Dadurch können die Multiplikationen effizient als Bit-Shift-Operationen implementiert werden, die auf einem FPGA mit minimalem Hardwareaufwand durchgeführt werden können. Somit reduziert sich die Berechnung auf einen Vergleich zwischen Real- und Imaginärteil, gefolgt von einer Bitverschiebung und einer Addition.

Obwohl diese Methode im Durchschnitt Abweichungen von weniger als 10 % gegenüber dem exakten Wert aufweist, ergaben erste Tests, dass die Ungenauigkeiten in den Zwischenergebnissen so groß sind, dass sie zu signifikanten Abweichungen von der exakten Lösung führen. In einigen Fällen wurde sogar festgestellt, dass die Präambel nicht mehr zuverlässig erkannt wird.

Um diese Probleme zu umgehen, wurde eine alternative Methode gewählt, die anstelle der direkten Betragsberechnung verwendet wird. Die Berechnung des Betrags erfordert eine Wurzeloperation. Da im nächsten Schritt das Betragsquadrat benötigt wird, kann auf die Wurzeloperation verzichtet

werden, ohne dass das Endergebnis verfälscht wird. Dieser Zusammenhang ist in Gleichung 14 dargestellt.

$$\left(\sqrt{\text{Real}^2 + \text{Imag}^2}\right)^2 = \text{Real}^2 + \text{Imag}^2 \tag{14}$$

Diese Umformung hat den entscheidenden Vorteil, dass keine Wurzelberechnung erforderlich ist. Stattdessen kann die Berechnung effizient durch zwei Multiplikationen und eine Addition realisiert werden, was auf Hardwareebene exakt berechnet werden kann, anstatt nur approximiert wie bei der vorherigen Methode.

2.4 Berechnung der Payload-Länge

Eine weitere gewünschte Funktion des Packet Detectors besteht darin, nach der Identifizierung einer Präambel die Anzahl der Samples bis zur nächsten Präambelerkennung zu ermitteln. Dieser Zählwert definiert die Payload-Länge, da er alle Samples umfasst, die zwischen dem letzten Sample der aktuellen Präambel und dem ersten Sample der nächsten Präambel liegen. Dementsprechend lässt sich die Berechnung der Payload-Länge wie folgt formulieren:

$$L_{\text{payload}} = N_{\text{präamble}} - n_{\text{präamble}} - L_P \tag{15}$$

Hierbei gilt: - $N_{\text{präamble}}$ ist die Sample-Nummer der aktuell erkannten Präambel. - $n_{\text{präamble}}$ ist die Sample-Nummer der vorhergehenden erkannten Präambel. - L_P ist die Länge der Präambel.

Da die Länge der Präambel bereits in der Berechnung enthalten ist, muss sie zusätzlich subtrahiert werden.

Da es sich hier um einen kontinuierlichen Datenstrom handelt, wäre es für den FPGA nachteilig, absolute Sample-Nummern für die Berechnung zu verwenden. Da die Anzahl der eingehenden Samples sehr groß sein kann, wären entsprechend große Zähler erforderlich, um Überläufe zu vermeiden. Stattdessen wird der relative Abstand zwischen zwei Präambeln genutzt.

Dies bedeutet, dass der FPGA ab der Erkennung einer Präambel einen Zähler hochzählen lässt, bis die nächste Präambel erkannt wird. Da jedoch die Präambel selbst mitgezählt wird, müsste am Ende die Länge der Präambel subtrahiert werden. Diese zusätzliche Rechenoperation kann jedoch vermieden werden, indem der Zähler nach der Erkennung einer Präambel nicht auf 0 gesetzt wird, sondern auf $-L_P$. Dadurch entfällt die Notwendigkeit der nachträglichen Subtraktion.

Durch diese Umformungen und Festlegungen wurden effiziente Berechnungsvorschriften entwickelt, die sich gut auf einem FPGA umsetzen lassen. Die konkrete Implementierung dieser Berechnung in Verilog wird in Kapitel 3 beschrieben.

3 Verilog-Umsetzung und Dokumentation der HDL-Module

Für die Umsetzung des Paketdetektors in Verilog wurden zwei verschiedene Implementierungsansätze verwendet: die Pipeline-Implementierung und die Rechenwerk-Implementierung

Der Vorteil der Pipeline-Implementierung besteht darin, dass pro Takt ein neues Ergebnis erzeugt wird. Im Gegensatz dazu benötigt die Rechenwerk-Implementierung mehrere Takte, um ein Ergebnis zu berechnen. Allerdings hat die Pipeline-Implementierung den Nachteil, dass sie einen deutlich höheren Logikgatterverbrauch erfordert. Dies liegt daran, dass für jede Rechenoperation eine eigene Hardwareeinheit bereitgestellt werden muss. In der Rechenwerk-Implementierung können Berechnungen hingegen iterativ durchgeführt werden, sodass dieselben Rechenoperationen mehrfach genutzt werden können. Dadurch reduziert sich der benötigte Logikgatteraufwand er-

heblich, da sich wiederholende Berechnungen nicht mehrfach, sondern nur einmal hardwareseitig implementiert werden müssen.

Da der Eingangsdatenstrom mit einer Frequenz von 1 MHz vorliegt, müssen die Berechnungen innerhalb eines Takts dieses Eingangssignals abgeschlossen sein. Da hier mit iterativen Berechnungen gearbeitet wird, muss die Taktfrequenz des FPGAs entsprechend höher gewählt werden, um die notwendigen Berechnungen rechtzeitig durchführen zu können.

3.1 Vorgaben festlegen

Als Festkomma-Format wurde 4.12 gewählt, d.h. 4 Vorkommabits und 12 Nachkommabits. Dieses Format wird für alle Variablen genutzt, mit Ausnahme der Zähler.

Bei Multiplikationen ist es jedoch so, dass das Ergebnis das Format 8.24 annimmt. Um das ursprüngliche 4.12-Format beizubehalten, wird das Ergebnis durch eine Shift-Operation wieder zurückgewandelt.

Die Präambel besteht aus einer Folge von 1 und -1 und wurde ebenfalls im 4.12-Format als 'localparam' gespeichert:

```
localparam signed [15:0] P0 = 16'b0001000000000000; // Repräsentation von +1
localparam signed [15:0] P1 = 16'b1111000000000000; // Repräsentation von -1
```

Da die Berechnung der Metrik eine Multiplikation der Eingangs-Samples mit der Präambel erfordert, wäre eine direkte Multiplikation auf einem FPGA ineffizient. Stattdessen kann die Multiplikation durch eine Überprüfung des MSB (Most Significant Bit) und anschließender kleiner Berechnung erfolgen:

Falls das MSB (Most Significant Bit) = 0 ist, bleibt das Ergebnis unverändert (entspricht der Multiplikation mit 1). Falls das MSB = 1 ist, handelt es sich um eine negative Zahl, und das Ergebnis muss negiert werden (entspricht der Multiplikation mit -1).

Die Negierung erfolgt durch Bit-Invertierung und Addition von 1 (Zweierkomplement-Darstellung). Dies kann in Verilog effizient wie folgt umgesetzt werden:

Listing 1: Alternative berechnung der Multiplikation der Metrik

```
1    term0_R  = (P0[15]  ? (~r0 + 1) : r0);
```

Diese Methode spart FPGA-Ressourcen, da eine Multiplikation durch eine einfache Bedingung und eine nachfolgende Berechnung ersetzt wird. Da bei dieser Berechnungsmethode lediglich das MSB benötigt wird, sind die restlichen Bits eigentlich überflüssig. Daher könnte die Präambel, anstatt mit +1 und -1, auch einfach mit 0 und 1 codiert werden.

Zusätzlich wurde in der Implementierung ein synchroner, active-high Reset verwendet. Dies bedeutet, dass der Reset nur zur steigenden Clock Taktflanke aktiviert werden kann und durch ein High-Signal ausgelöst wird.

3.2 Grundlegende Rechenoperationen

Zur Berechnung der Ungleichung 9 wird eine Summation über so viele Summanden benötigt, wie die Präambellänge L_P beträgt. Dabei wurde entschieden, diese Summation nicht iterativ über mehrere Takte hinweg durchzuführen, sondern innerhalb eines einzigen Takts.

Um den Hardwareaufwand möglichst gering zu halten, wäre es grundsätzlich effizienter, die Summation über die gesamte Präambellänge iterativ zu berechnen, da hierfür nur ein einzelner Additionsblock benötigt wird. Dies hätte jedoch zur Folge, dass so viele Takte benötigt würden, wie die Präambellänge beträgt – in diesem Fall also 23. Da diese Summation mindestens dreimal ausgeführt

werden muss, würde dies zu einer benötigten Taktzahl von: $3 \cdot 23 = 69$ führen. Das bedeutet, dass allein für die Summationsberechnungen 69 Takte erforderlich wären. Bei einer Eingangssymbolrate von 1 MHz müsste der FPGA daher mit mindestens 69 MHz takten, nur um die Berechnung der Addition durchführen zu können. Aus diesem Grund wurde die Berechnung so optimiert, dass die Summation innerhalb eines einzigen Takts ausgeführt wird.

Bevor die Berechnung der Metrik durchgeführt werden kann, müssen die Eingangs-Samples mit der Präambel multipliziert werden. Diese Berechnung wurde bereits in Kapitel 3.1 beschrieben. Damit sind nun alle notwendigen Werte vorhanden, um die linke Seite der Ungleichung zu berechnen.

Um die mittlere Energie des Signals zu bestimmen, wird ebenfalls eine Summation über 23 Elemente benötigt. Bevor diese Summation durchgeführt werden kann, muss jedoch zunächst das Betragsquadrat der einzelnen Elemente berechnet werden. Auch hier wurde die Berechnung so umgesetzt, dass sie innerhalb eines einzigen Takts erfolgt. Dies wird im folgenden Verilog-Code dargestellt:

Listing 2: Berechnung des Betragsquadrats

```
1 abs_r = (sample0[15] == 1'b1) ? (~sample0 + 16'd1) : sample0;
2 abs_i = (sample1[15] == 1'b1) ? (~sample1 + 16'd1) : sample1;
3
4 mult_result = (abs_r * abs_r) + (abs_i * abs_i);
5
6 res_o = mult_result >>> 12;
```

Das Ergebnis dieser Berechnung wird in einem Schieberegister gespeichert. Die anschließende Summation der 23 Elemente erfolgt mit den Werten aus diesem Schieberegister.

Zu guter Letzt fehlt nur noch die Multiplikation mit $T \cdot L_P$, die, wie in Kapitel 3.1 beschrieben, durch eine Bit-Shift-Operation realisiert wird. Dabei wurde entschieden, nicht direkt mit der mittleren Energie zu multiplizieren, sondern stattdessen $T \cdot L_P$ auf die andere Seite der Ungleichung zu bringen. Dies entspricht einer Multiplikation mit dem Kehrwert, was bedeutet, dass die Bit-Shift-Operation nach rechts statt nach links durchgeführt werden muss.

Diese Entscheidung wurde getroffen, da es bei einer direkten Multiplikation mit der mittleren Energie aufgrund der geringen Bitbreite zu Überläufen kommen kann. Durch das Verschieben in die entgegengesetzte Richtung wird dies vermieden.

Damit sind nun alle grundlegenden Rechenoperationen definiert. Diese müssen nun lediglich in der richtigen Reihenfolge ausgeführt werden, was im nächsten Kapitel 3.3 erläutert wird.

3.3 Zustandsmaschine

Da die Berechnungen in einem Rechenwerk erfolgen, wird eine Zustandsmaschine benötigt, die die einzelnen Zustände steuert und die Berechnungen schrittweise ausführt. Im Folgenden ist die Zustandsmaschine als Schaubild dargestellt, um einen Überblick über alle Zustände zu erhalten.

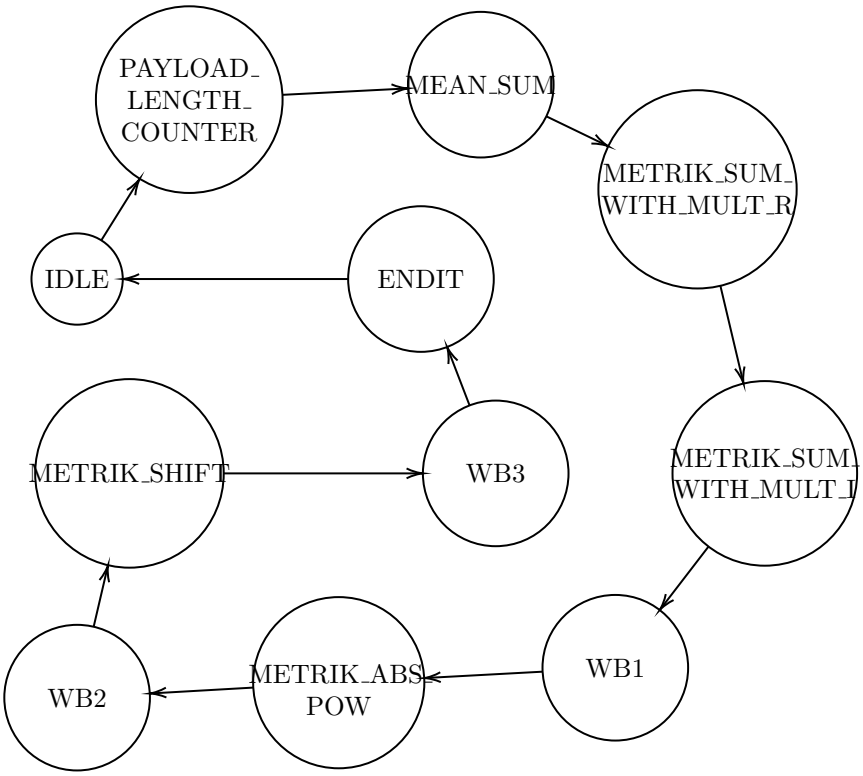


Abbildung 2: Zustandsdiagram

In der folgenden Darstellung ist die Zustandsmaschine in Tabellenform aufgeführt, um die Zustandsübergänge sowie die Zustandssemantik übersichtlich darzustellen.

Zustand	**Operation**	**Register Transfer / ALU CMD**	**Folgezustand**
IDLE	Warten auf Startsignal	Kein Register-Transfer, mode_o = ALU_IDLE	MEAN_ABS_POW
MEAN_ABS_POW	Berechnung des Betrags der komplexen Metrik	r_i auf alu_a, i_i auf alu_b, mode_o = CMPLX_ABS_POW	PAYLOAD_LENGTH_COUNTER
PAYLOAD_LENGTH_COUNTER	Zählen der Payload-Länge	payload_length_counter auf alu_a, one auf alu_b, mode_o = SUM_23, wren_mean_abs_pow_o = 1	MEAN_SUM
MEAN_SUM	Berechnung des Mittelwerts der Energie	mean_samples auf alu_a, mode_o = SUM_23, wren_payload_length_counter_o = 1	METRIK_SUM_WITH_MULT_R
METRIK_SUM_WITH_MULT_R	Berechnung der Metrik-Summe (Realteil)	metrik_samples_R auf alu_a, mode_o = SUM_23, wren_mean_sum_o = 1	METRIK_SUM_WITH_MULT_I
METRIK_SUM_WITH_MULT_I	Berechnung der Metrik-Summe (Imaginärteil)	metrik_samples_I auf alu_a, mode_o = SUM_23, wren_metrik_sum_R_o = 1	WB1
WB1	Write-Back der Metrik-Summe (Imaginärteil)	wren_metrik_sum_I_o = 1	METRIK_ABS_POW
METRIK_ABS_POW	Berechnung des Metrik-Betrags	metrik_sum_R auf alu_a, metrik_sum_I auf alu_b, mode_o = CMPLX_ABS_POW	WB2
WB2	Write-Back des Metrik-Absolutwerts	wren_metrik_abs_pow_o = 1	METRIK_SHIFT
METRIK_SHIFT	Shift der Metrik-Werte	metrik_abs auf alu_a, number_of_shifts auf alu_b, mode_o = SHIFT_RIGHT	WB3
WB3	Write-Back des Shift-Resultats	wren_metrik_shift_o = 1	ENDIT
ENDIT	Vergleich der normierten Metrik mit dem Schwellwert	check_for_packet_detect_o = 1, Falls detect_o = 1 → wren_reset_payload_lenght_o = 1	IDLE

Tabelle 1: Zustandsmaschine des Moduls packet_detector_controller mit mathematischen Operationen

3.4 Datenflussgraf

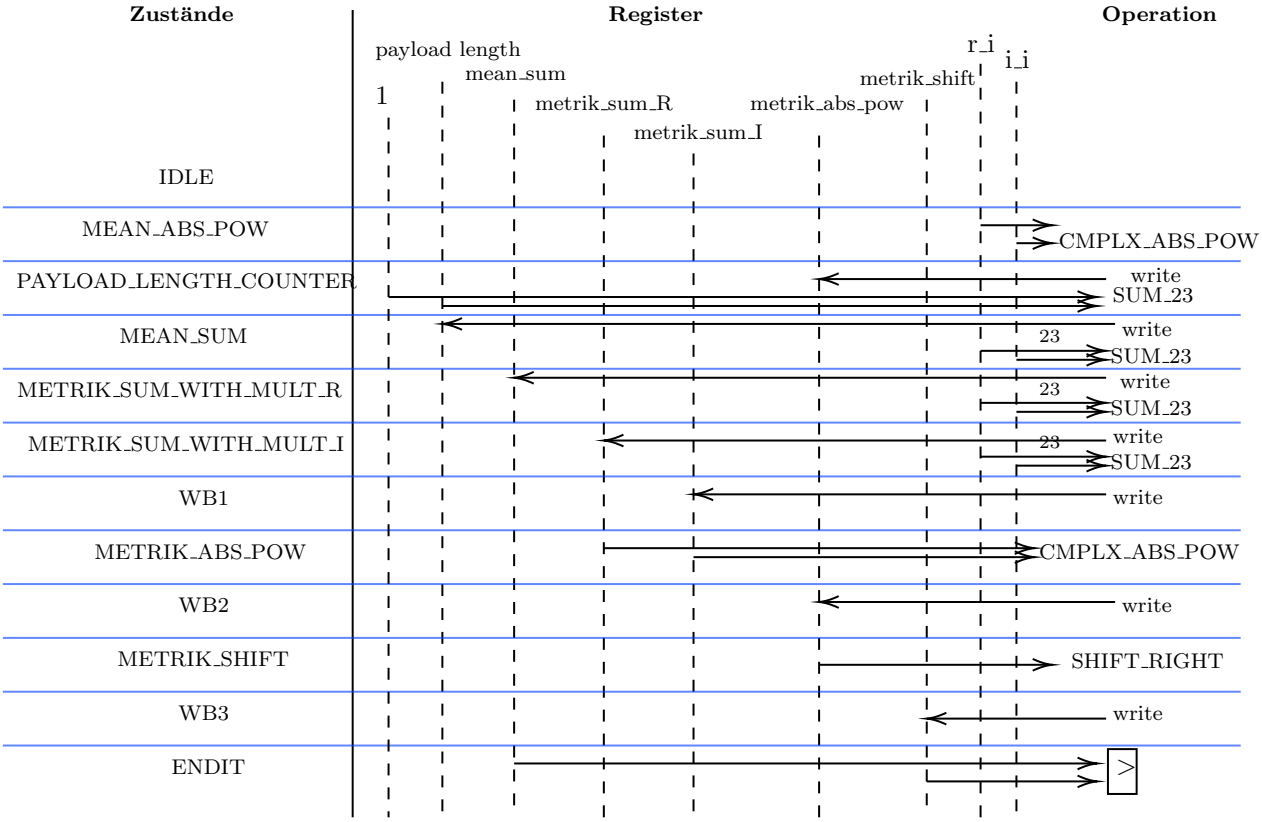


Abbildung 3: Datenflussgraf

3.5 Module

In diesem Kapitel werden alle verwendeten Module sowohl grafisch als auch tabellarisch übersichtlich dargestellt. Dabei werden ihre Funktionen, Ein- und Ausgänge sowie Zustandsübergänge erläutert.

Datei	**Kommentar**
`packet_detector_top.v`	Top-Level Modul der Paketdetektion
`packet_detector_controller.v`	Ablaufsteuerung (FSM) für die Paketdetektion
`packet_detector_datapath.v`	Datenpfad für die Paketdetektion
`packet_detector_ALU.v`	Kombinatorische ALU innerhalb des Datenpfades
`packet_detector_top_tb.v`	Testbench für das Top-Level-Modul

Tabelle 2: Dateistruktur des Paketdetektionssystems

3.5.1 ALU-Modul

Das ALU-Modul ist für die Durchführung der Rechenoperationen zuständig. Es besitzt insgesamt 23 Eingänge, die für verschiedene Berechnungen genutzt werden können. Im Folgenden ist das Schaubild des Moduls dargestellt:

Abbildung 4: Schaubild des ALU-Moduls

Zusätzlich ist in der folgenden Tabelle die genaue Spezifikation des ALU-Moduls aufgeführt:

Signal	**Semantik**	**I/O**
`clk`	Takt-Signal (ALU Clock)	I
`rst`	Aktives High Reset-Signal	I
`sample0, sample1, ..., sample22`	23 Signale, jeweils `signed [15:0]` (z.B. für Eingangswerte, Filter, Berechnungen etc.)	I
`mode_i`	Betriebsmodus (`[2:0]`) zur Steuerung der ALU-Funktion	I
`res_o`	Ergebnis der ALU (`reg [15:0]`)	O

Tabelle 3: Signale und Semantik des Moduls `packet_detector_ALU`

Eine Übersicht über die Zustände der ALU ist hier dargestellt:

Operation	**Erklärung**	**Verilog local Parameter**	**Zahlenwert**
Summation über 23 Werte	Berechnung der Summe aus 23 Eingangswerten	`SUM_23`	`3'd0`
Betragsquadrat	Berechnung des Betragsquadrat einer komplexen Zahl	`CMPLX_ABS_POW`	`3'd1`
Multiplikation	Multiplikation zweier Werte	`MULT`	`3'd2`
Rechtsshift	Rechtsshift einer Zahl um eine bestimmte Anzahl an Bits	`SHIFT_RIGHT`	`3'd3`
ALU IDLE	Keine Operation (NOP)	`ALU_IDLE`	`3'd4`

Tabelle 4: ALU-Operationen und ihre lokalen Parameter

3.5.2 Datenpfad-Modul

Das Datenpfad-Modul übernimmt die Verarbeitung und Weiterleitung der Daten zwischen den einzelnen Komponenten. Das folgende Schaubild veranschaulicht die Struktur dieses Moduls:

Abbildung 5: Schaubild des Datenpfad-Moduls

Die zugehörige tabellarische Beschreibung findet sich hier:

Signal	**Semantik**	**I/O**
`clk`	Takt-Signal	I
`rst`	Aktives High Reset-Signal	I
`start_i`	Startsignal zur Initialisierung der Verarbeitung	I
`r_i, i_i`	Real- und Imaginärteil des Eingangssignals (`signed [15:0]`)	I
`mode_i`	Betriebsmodus (`[2:0]`) zur Steuerung des Datenpfads	I
`Write Back, Register Transfer flags`	Register Transfer innerhalb des Datenpfades	I
`check_for_packet_detect_i`	Signal zur Prüfung, ob ein Paket detektiert wurde	I
`payload_length_o`	Ausgabe der berechneten Payload-Länge (`signed [15:0]`)	O
`detect_o`	Signal zur Anzeige einer erfolgreichen Paketdetektion	O

Tabelle 5: Signale und Semantik des Moduls `packet_detector_datapath`

3.5.3 Controller-Modul

Das Controller-Modul steuert die Abläufe innerhalb des Systems und verwaltet die Zustandsübergänge der Module. Das folgende Schaubild zeigt seine Struktur:

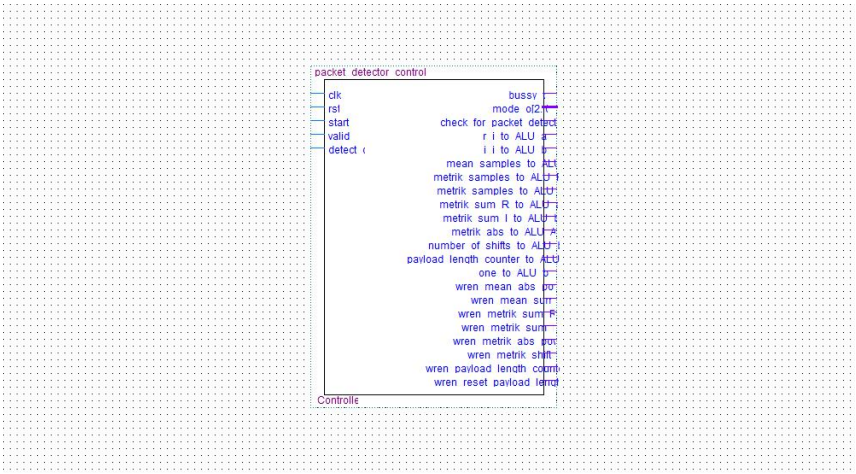


Abbildung 6: Schaubild des Controller-Moduls

Die genaue Beschreibung der Funktionsweise ist in der folgenden Tabelle aufgeführt:

Signal	**Semantik**	**I/O**
`clk`	Takt-Signal	I
`rst`	Aktives High Reset-Signal	I
`start_i`	Startsignal zur Initialisierung der Verarbeitung	I
`valid_i`	Signal, das gültige Eingangsdaten anzeigt	I
`detect_o`	Signal zur Anzeige einer erfolgreichen Paketdetektion	I
`bussy_o`	Signal, das angibt, dass der Controller gerade arbeitet	O
`Write Back, Register Transfer flags`	Register Transfer innerhalb des Controllers	O
`mode_o`	Betriebsmodus (`[2:0]`) zur Steuerung des Datenpfads	O
`check_for_packet_detect_o`	Signal zur Prüfung, ob ein Paket detektiert wurde	O

Tabelle 6: Signale und Semantik des Moduls `packet_detector_controller`

3.5.4 Top-Modul

Das Top-Modul stellt die oberste Hierarchieebene dar und verbindet die einzelnen Module miteinander. Es koordiniert die Datenflüsse und steuert das Gesamtsystem. Das folgende Schaubild zeigt die Struktur des Top-Moduls:

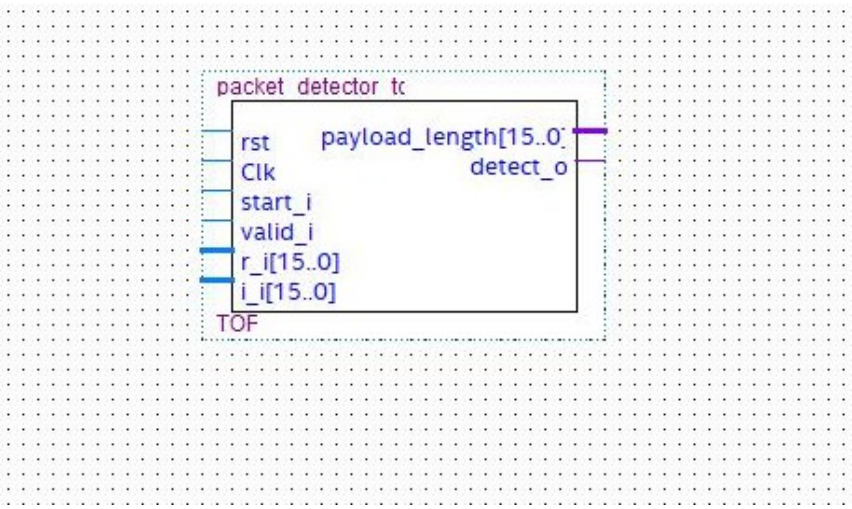


Abbildung 7: Schaubild des Top-Moduls

Die tabellarische Übersicht zu diesem Modul findet sich hier:

Signal	**Semantik**	**I/O**
`rst`	Aktives High Reset-Signal	I
`Clk`	System-Taktsignal	I
`start_i`	Startsignal zur Initialisierung der Verarbeitung	I
`valid_i`	Valid-Signal zur Angabe gültiger Eingangsdaten (z. B. für Testdaten)	I
`r_i`	Realteil des Eingangssignals (`signed [15:0]`, Q4.12)	I
`i_i`	Imaginärteil des Eingangssignals (`signed [15:0]`, Q4.12)	I
`payload_length`	Berechnete Payload-Länge (`signed [15:0]`)	O
`detect_o`	Signal zur Anzeige einer erfolgreichen Paketdetektion	O

Tabelle 7: Signale und Semantik des Moduls `packet_detector_top`

3.6 Simulation

Die Simulation zur Überprüfung des Paketdetektor-Moduls wurde mit Hilfe von ModelSim durchgeführt. Dabei wurden die bereitgestellten Datensätze, die im .mat-Format vorlagen, in eine Textdatei konvertiert und anschließend in ModelSim eingelesen.

Da ModelSim keine .mat-Dateien direkt verarbeiten kann, wurde ein selbstgeschriebenes MATLAB-Skript verwendet, um die Konvertierung in eine .txt-Datei vorzunehmen. Das Skript wandelt die komplexwertigen Zahlen aus der .mat-Datei in zwei separate binäre Zahlen um – eine für den Realteil und eine für den Imaginärteil.

Das MATLAB-Skript sieht wie folgt aus:

```
1  load('test_signal_1.mat');
2
3  dataList = symbole_rx;
4
5  n = 4;          % number bits for integer part of your number
6  m = 12;          % number bits for fraction part of your number
7
8
9  fileID = fopen('test_signal.txt', 'w');
10 if fileID == -1
11     error('Fehler beim oeffnen der Datei.');
```

```
end

for idx = 1:length(symbole_rx) - Lp
    re = real(dataList(idx));
    im = imag(dataList(idx));
    conv_re = 0;
    conv_im = 0;
    if re < 0
        re = -1*re;
        conv_re = 1;
    end

    if im < 0
        im = -1*im;
        conv_im = 1;
    end

    re = [ fix(rem(fix(re)*pow2(-(n-1):0),2)), fix(rem( rem(re,1)*
       pow2(1:m),2))];
    im = [ fix(rem(fix(im)*pow2(-(n-1):0),2)), fix(rem( rem(im,1)*
       pow2(1:m),2))];

    if conv_re
        re = 1-re;
        carry = 1;
        for i = length(re):-1:1
            sum_val = re(i) + carry;
            re(i) = mod(sum_val, 2);    % Neue Bit-Wert
            carry = floor(sum_val/2);
        end
    end

    if conv_im
        im = 1-im;
        carry = 1;
        for i = length(im):-1:1
            sum_val = im(i) + carry;
            im(i) = mod(sum_val, 2);    % Neue Bit-Wert
            carry = floor(sum_val/2);
        end
    end

    re_s = compose("%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d",re);
    im_s = compose("%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d",im);

    fprintf(fileID, '%s, %s\n', re_s, im_s);
end

fclose(fileID);
```

Dieses Skript sorgt dafür, dass die komplexen Zahlen aus der .mat-Datei in zwei getrennte binäre Werte umgewandelt werden. In der erzeugten .txt-Datei stehen diese binären Zahlen kommagetrennt, wobei jede neu eingelesene Komplexe Zahl in einer eigenen Zeile gespeichert wird. Dadurch entspricht das Format einer CSV-Datei, die dann von ModelSim eingelesen werden kann.

Oben im Skript kann außerdem festgelegt werden, welches Festkomma-Format verwendet wird, also wie viele Bits für den Ganzzahl- und Nachkommateil vorgesehen sind.

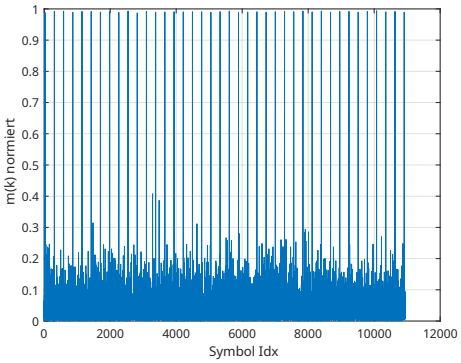
In der Testbench für das Topmodul wird die erstellte .txt-Datei geöffnet und pro Takt eine Zeile ausgelesen. In der Simulation wurde ein Basistakt von 16 MHz verwendet, was einem typischen FPGA-Takt entspricht und über der erforderlichen Mindestfrequenz von 12 MHz liegt. Um die Eingangssymbolrate zu steuern, wurde ein Zähler implementiert. Dieser zählt bis 16 und liest anschließend eine neue Zeile aus der Datei. Da die Eingangssymbolrate 1 MHz beträgt, wird alle 16 Takte eine neue Eingangsprobe verarbeitet. Auf diese Weise wird sichergestellt, dass die Eingangssymbole mit einer Frequenz von genau 1 MHz in die FPGA-Logik eingespeist werden.

Durch diese Methode kann die Paketerkennung unter realistischen Bedingungen getestet und mit der MATLAB-Simulation verglichen werden.

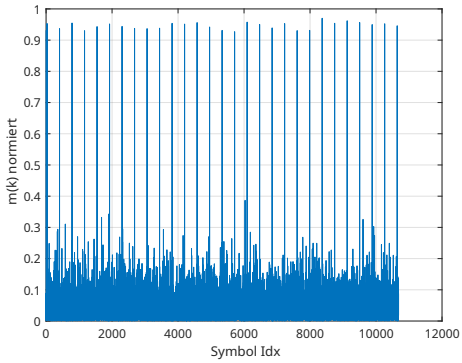
4 Test und Verifikation

Die Überprüfung der Richtigkeit des Algorithmus zur Paketerkennung wurde mit Hilfe eines MATLAB-Skripts durchgeführt. Dabei standen insgesamt vier Datensätze zur Verfügung, die zur Evaluierung genutzt wurden.

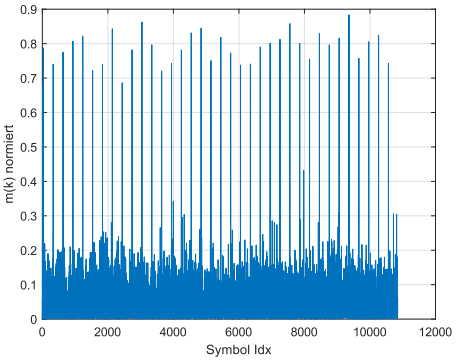
Zusätzlich wurde ein Skript bereitgestellt, das die normierte Metrik berechnet. Die Ergebnisse dieses Skripts sind in Abbildung 8 dargestellt.



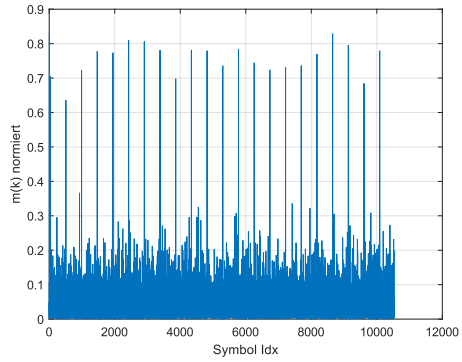
(a) Normierte Metrik für test_signal_1.mat



(b) Normierte Metrik für test_signal_2.mat



(c) Normierte Metrik für test_signal_3.mat



(d) Normierte Metrik für test_signal_4.mat

Abbildung 8: Grafische Darstellung der Berechnung der normierten Metrik für die 4 Datensätze

In den Abbildungen ist zu erkennen, dass vier verschiedene Eingangssignale aufgenommen wurden. Dies zeigt sich daran, dass die Payloadlänge in jedem Datensatz unterschiedlich ist und auch die berechneten Ergebnisse variieren. Besonders auffällig ist, dass die Metrikwerte für die korrekte Erkennung einer Präambel von Datensatz zu Datensatz abnehmen. Dadurch muss für jeden Datensatz ein angepasster Schwellwert gewählt werden, um eine zuverlässige Paketerkennung zu

gewährleisten. Die vier Datensätze bieten somit eine solide Grundlage zur umfassenden Evaluierung des Packet Detector Algorithmus.

4.1 Überprüfung der Richtigkeit der gemachten Detektion

Zur Überprüfung des Packetdetektor-Algorithmus wurde in der Testbench eine Ausgabe hinzugefügt, die bei jeder erkannten Präambel die folgenden Informationen ausgibt:

```
DETECT BEI: Count = 1725, payload_length = 256, r_i = 1111111010000111,
i_i = 0000000010100111, detect_o = 1
DETECT BEI: Count = 2004, payload_length = 256, r_i = 1111111010111100,
i_i = 0000000100100001, detect_o = 1
DETECT BEI: Count = 2283, payload_length = 256, r_i = 1111111100111101,
i_i = 0000000110011101, detect_o = 1
```

Die wichtigsten Informationen in dieser Ausgabe sind die Variablen Count und payload_length. Die Variable Count gibt an, bei welchem eingelesenen Sample der Paketdetektor ein Paket erkannt hat. Dies ermöglicht einen direkten Abgleich mit MATLAB, um die Korrektheit des entworfenen Algorithmus zu überprüfen. Ein weiteres wichtiges Kriterium war die Ausgabe der Payload-Länge, die hier ebenfalls dargestellt wird.

Für die Berechnung der Metrik und auch der Überprüfung der neuen Berechnung, welche in Kapitel 2 vorgestellt wurde, wurde ein neues Matlab Script geschrieben. Dessen grafische Ausgabe ist in Abbildung 9 zu sehen.

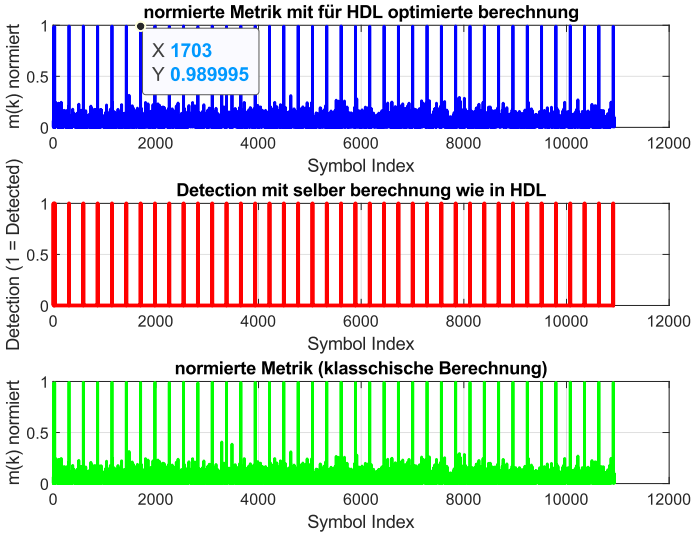


Abbildung 9: Grafische Darstellung der normierten Metrikwerte mittels neuem Matlab Script (test_signal_1.mat)

Im oberen Graphen ist die Berechnung der normierten Metrik zu sehen, die unter Berücksichtigung der in Kapitel 2 beschriebenen Modifikationen durchgeführt wurde. Der mittlere Graph zeigt die Ausgabe der Ungleichung, also das Ergebnis des Paketdetektors, das anzeigt, wenn ein Paket erkannt wird. Im unteren Graphen ist die normierte Metrik dargestellt, die auf klassische Weise berechnet wurde und somit als Referenz dient.

Ein Vergleich der Graphen zeigt, dass die oberen und unteren Darstellungen identisch sind. Dies bedeutet, dass die mathematischen Vereinfachungen für den FPGA äquivalent zur klassischen

Berechnung sind. Im oberen Graphen wurde zudem ein Punkt markiert, der ein erkanntes Präambel darstellt. Die Position dieses erkannten Präambels beträgt in MATLAB 1703.

Vergleicht man nun die Positionen der erkannten Präambel in MATLAB mit den in ModelSim simulierten Werten, stellt man eine Differenz fest. Während MATLAB eine Position von 1703 ausgibt, liefert die Simulation in ModelSim einen Wert von 1725. Diese Abweichung lässt sich dadurch erklären, dass MATLAB die Präambelberechnung auf Basis des aktuellen Samples und der folgenden 22 Samples durchführt, während in der ModelSim-Simulation das aktuelle Sample und die vorherigen 22 Samples berücksichtigt werden. Dadurch ergibt sich eine Differenz von 23.

4.2 Überprüfung der Berechnung der Payloadlänge

Für die Berechnung der Payload-Länge wurde folgender MATLAB-Code zur Verifikation genutzt:

```
1 if (real(temp_sum)^2 + imag(temp_sum)^2) > T_Lp * power_estimate
2         payload_estimate = it - last_detect - Lp;
3         last_detect = it;
4 end
```

Dabei wurde die gleiche Berechnung verwendet, die auch auf dem FPGA ausgeführt wird. Die hier genutzte Methode zur Ermittlung der Payload-Länge basiert darauf, die absolute Position der Samples in der Datei auszuwerten und voneinander zu subtrahieren. Dies unterscheidet sich zwar von der Implementierung auf dem FPGA, da dort eine andere Herangehensweise zur Berechnung genutzt wird, jedoch dient dieser MATLAB-Code lediglich dem Vergleich der Ergebnisse. Die Simulationsergebnisse zeigen, dass die mit diesem Skript berechneten Werte exakt mit den FPGA-Ergebnissen übereinstimmen.

4.3 Limitationen

Untersuchungen zur Genauigkeit der Berechnung wurden nicht durchgeführt, da das exakte Ergebnis in diesem Fall nicht von entscheidender Bedeutung ist. Die Berechnung basiert auf einer Ungleichung, die entweder wahr oder falsch sein kann. Da bei allen Berechnungen keine Näherungen verwendet werden, sondern stets die exakte Lösung bestimmt wird, besteht die einzige potenzielle Fehlerquelle im Quantisierungsfehler.

Bisher konnten erfolgreiche Tests nur mit dem ersten Datensatz durchgeführt werden. Bei den anderen Datensätzen konnte der aktuelle Paketdetektor-Algorithmus keine Präambel erkennen. Während der Analyse wurde festgestellt, dass es bei der Berechnung zu Überläufen in den Variablen kommen kann, die sowohl zur Speicherung der Metrik als auch der mittleren Energie des Signals verwendet werden.

Ein Überlauf führt dazu, dass das höchstwertige Bit verloren geht. Da die Berechnung auf einer Ungleichung basiert, bei der geprüft wird, welcher Wert größer ist, kann der Verlust des höchstwertigen Bits zu erheblichen Abweichungen im Ergebnis führen. Die Ursache dieses Problems liegt in der zu geringen Bitbreite der verwendeten Variablen.

4.4 Echtzeitfähigkeit

Es wurde gefordert, dass das Modul echtzeitfähig ist. Das bedeutet, dass die zeitliche Differenz zwischen dem Eingang eines Samples und der darauf basierenden Berechnung möglichst gering sein muss. Eine klare Definition, welche Verzögerung maximal zulässig ist, um ein System als echtzeitfähig zu bezeichnen, existiert zwar nicht, jedoch sollte in diesem Fall die Berechnung innerhalb eines Takts des Eingangssignals abgeschlossen sein. Da die Eingangssamples mit einer Frequenz von 1 MHz eintreffen, ergibt sich eine verfügbare Berechnungszeit von 1 µs.

Um Logikgatter zu sparen, wird die Berechnung iterativ durchgeführt, wodurch der FPGA einen höheren internen Basistakt benötigt. Für die hier umgesetzte Variante des Paketdetektors ist ein Takt von mindestens 12 MHz erforderlich, da der Algorithmus, wie in Kapitel 3.3 zu sehen, 12 Zustände umfasst. Dieser Takt ist für FPGAs vergleichsweise niedrig, weshalb die Implementierung noch Optimierungspotenzial bietet, insbesondere in Bezug auf die Reduzierung des Logikgatteraufwands.

Mögliche Verbesserungen könnten darin bestehen, die Summation der 23 Elemente in kleinere Teilsummen zu unterteilen. Beispielsweise könnte zunächst eine Summation über 5 Elemente erfolgen, wodurch der Logikgatteraufwand reduziert und gleichzeitig eine höhere Taktfrequenz ermöglicht wird. Eine weitere Optimierung wäre die Aufspaltung der Berechnung des Betragsquadrats der komplexen Zahlen in separate Multiplikations- und Additionsschritte, da beide Rechenoperationen bereits als Zustände in der ALU vorhanden sind.

5 Zusammenfassung und Fazit

In dieser Arbeit konnte gezeigt werden, dass ein funktionsfähiger Paketdetektor-Algorithmus erfolgreich in Verilog implementiert wurde. Die grundlegende Formel zur Berechnung der Metrik wurde gezielt optimiert, um eine möglichst effiziente Ausführung auf einem FPGA zu ermöglichen. Dabei wurden verschiedene Methoden angewandt, um die Berechnungen sowohl in Bezug auf Geschwindigkeit als auch auf den Ressourcenverbrauch zu verbessern.

Tests haben bestätigt, dass der Algorithmus grundsätzlich funktioniert. Allerdings wurden Schwächen in der Implementierung festgestellt, insbesondere im Hinblick auf die zu geringe Bitbreite der Variablen. Diese führte in einigen Fällen zu Überläufen, wodurch die Erkennung der Präambel bei bestimmten Datensätzen nicht zuverlässig erfolgte. Trotz dieser Einschränkungen konnte mit dem ersten Datensatz nachgewiesen werden, dass sowohl die Paketerkennung als auch die Berechnung der Payload-Länge korrekt durchgeführt wurden. Damit wurden die gestellten Anforderungen erfüllt.

5.1 Mögliche Verbesserungen

Zur Optimierung des Algorithmus und der Implementierung auf dem FPGA lassen sich mehrere Verbesserungsansätze identifizieren:

Optimierung des Zustandsautomaten

Der Zustandsautomat zur Berechnung der Metrik könnte effizienter gestaltet werden, um die Anzahl der benötigten Zustände zu reduzieren. Eine optimierte Struktur könnte die Berechnungen in weniger Taktzyklen ausführen.

Aufteilung der Berechnung des Betragsquadrats

Die Berechnung des Betragsquadrats einer komplexen Zahl könnte in Teilschritte unterteilt werden. Anstatt die gesamte Berechnung in einem Schritt durchzuführen, könnte sie in zwei Multiplikationen und eine Addition aufgeteilt werden. Dies könnte die Ressourcennutzung auf dem FPGA verbessern.

Anpassung der Multiplikation und Division von $L_P \cdot T$

Derzeit wird die Berechnung von $L_P \cdot T$ durch eine Bit-Shift-Operation umgesetzt, um eine schnelle und effiziente Berechnung zu ermöglichen. Diese Methode führt jedoch dazu, dass der Schwellwert

auf etwa 0,7 festgelegt ist. Wie in Kapitel 4 gezeigt, kann dieser Schwellwert für einige Eingangssignale zu hoch sein und dadurch die Paketerkennung negativ beeinflussen.

Eine alternative Möglichkeit wäre, die Berechnung stattdessen über eine herkömmliche Multiplikation durchzuführen. Dies hätte den zusätzlichen Vorteil, dass die Multiplikation in der ALU mehrfach genutzt werden könnte, insbesondere wenn die Berechnung des Betragsquadrats wie zuvor beschrieben optimiert wird. Dadurch ließe sich der Einsatz von Logikgattern insgesamt reduzieren und die Effizienz der FPGA-Implementierung weiter steigern.

Dynamische Anpassung des Schwellwerts

Falls der Schwellwert durch eine Multiplikation flexibel eingebunden wird, könnte er während der Laufzeit automatisch an das Signal angepasst werden. Eine Möglichkeit hierfür wäre die gleitende Mittelwertberechnung des Signals. Der Schwellwert könnte dann beispielsweise auf einen bestimmten Prozentwert über dem gleitenden Mittelwert gesetzt werden. Da die Strukturen zur Berechnung des gleitenden Mittelwerts bereits vorhanden sind, würde diese Anpassung nur einen geringen zusätzlichen Logikaufwand erfordern.

Erhöhung der benötigten Takte und Reduzierung des Logikgatterverbrauchs

Derzeit wird die Summation der 23 Werte gleichzeitig durchgeführt, was zu einem hohen Logikgatterverbrauch führt. Eine mögliche Optimierung besteht darin, die Summation schrittweise durchzuführen und sie über mehrere Taktzyklen zu verteilen.

Durch diese Anpassung könnte die benötigte Hardware reduziert werden, da weniger parallele Additionsoperationen erforderlich wären. Allerdings würde dies eine höhere Anzahl an Taktzyklen erfordern. Eine mögliche Lösung wäre es, mehrere Additionen pro Takt durchzuführen, beispielsweise fünf Summationen innerhalb eines Takts. Die konkrete Umsetzung hängt dabei von der verwendeten Taktfrequenz des FPGAs ab und muss entsprechend angepasst werden.

Diese Methode könnte zusätzlich mit der optimierten Berechnung des Betragsquadrats kombiniert werden, da auch dort eine Summation erforderlich ist. Durch eine geschickte Anordnung der beiden Berechnungen könnte der Ressourcenverbrauch weiter reduziert und die Effizienz der Implementierung verbessert werden.

Verbesserung der allgemeinen Implementierung

Während der Analyse wurde festgestellt, dass noch einige Probleme in der Implementierung bestehen. Insbesondere wurden Berechnungen innerhalb des Datenpfads ausgeführt, obwohl diese ausschließlich in der ALU durchgeführt werden sollten.

Um die Modularität und Effizienz der Implementierung zu verbessern, sollten alle Rechenoperationen konsequent in der ALU erfolgen. Dadurch lässt sich nicht nur der Datenpfad vereinfachen, sondern auch die Wiederverwendbarkeit und Wartbarkeit des Designs erhöhen. Zudem kann eine strengere Trennung zwischen Steuer- und Recheneinheiten dazu beitragen, die Ressourcen auf dem FPGA effizienter zu nutzen.