

PA15_gelk_1 Polyphonic DDS Synthesizer mit MIDI Steuerung

ZÜRCHER HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN

INSTITUTE OF EMBEDDED SYSTEMS

Autorin	Katrin Bächli
Hauptbetreuer	Prof. Hans-Joachim Gelke
Nebenbetreuer	Dr. Matthias Rosenthal
Datum	17. Dezember 2015

Kontakt Adresse

c/o Inst. of Embedded Systems (InES)
Zürcher Hochschule für Angewandte Wissenschaften
Technikumstrasse 22
CH-8401 Winterthur

Tel.: +41 (0)58 934 75 25

Fax.: +41 (0)58 935 75 25

E-Mail: katrin.baechli@zhaw.ch

Homepage: <http://www.ines.zhaw.ch>

Zusammenfassung

Die hardwarenahe Programmiersprache VHDL ist ein wichtiger Bestandteil der digitale Signalverarbeitung. Die Projektarbeit setzt zwei unabhängige Aufgaben in VHDL um.

- Bilden zweier hardwarenahen Fehlerquellen: *glitches* und *metastability*
- Implementation eines *midi interfaces*, dessen Entwicklung auf einer *textbasierten testbench* basiert

Glitches werden künstlich Herbeigeführt in dem auf einem Cylcone II-FPGA die Pfade einzelner Signale verlängert sind. Dadurch treffen Werte verzögert ein und der asynchrone *decoder* verarbeitet falsche Werte. Falsche Signale gelangen auf die Leitung und sogenannte *glitch* entstehen.

Der metastabilen Zustand in einem System entsteht, durch das Verletzen der *setup* und *hold time*. Forciert ist dies durch zwei unterschiedliche Takte zweier VHDL-Logik-Blöcke, deren Takt-Phasen nicht übereinstimmen. Das Ausgangssignal des ersten Logik-Blocks ist als asynchroner Impuls auf den zweiten Logik-Block geführt, der eine *finite state machine* enthält. Dekodiert die *fsm* keinen definierten Zustand, befindet sich das System in einen undefinierten Zustand. Metastabilität trifft ein.

Der zweiten Teil der Projektarbeit beinhaltet ein *midi interface*, das Polyphonie detektiert. Die textbasierte *testbench* begleitet die Entwicklung des *midi controller*. Das *midi interface* detektiert die *status bytes* NOTE ON, NOTE OFF und POLYPHONY und die VHDL-Einheit *polyphony out* gibt 10 gedrückte Noten parallel aus.

Abstract

An important field in digital signal processing is the hardware-related programming language VHDL. In this thesis, two independent tasks have been drawn up and implemented using VHDL.

On one hand was the inducing of hardware-related glitches and metastability, and on the other hand the implementation of a MIDI-interface, whose development is built on a text-based testbench.

By extending the paths of individual signals on a Cyclone II-FPGA it is possible to generate artificial glitches. Hence, some signals arrive delayed at the asynchronous decoder. Invalid information is processed and put on the signal lines, which occurs in so-called glitches.

Metastable states are caused by clocking two VHDL logic blocks with two independent clocks, where no clock is a multiple of the other one. The output signal of the first block is connected as a asynchronous input of the second block. Since the two blocks work in a different clock domain, the finite state machine can fall in an undefined state, in other words the finite state machine is in a metastable state.

The second part of this thesis is focused on the development of a midi interface, which is able to detect polyphony. The text-based testbench is used for test driven development of the midi controller. The midi interface detects and handles the status bytes NOTE ON, NOTE OFF and POLYPHONE. The VHDL entity polyphony out is able to process up to 10 pressed keys in parallel.

Vorwort

Meine Motivation ist das vertiefte Kennenlernen der Sprache VHDL. Diese hardwarenahe Sprache beinhaltet mit der kombinatorischen Logik und der auch nicht-sequentiellen Prozessverarbeitung Eigenheiten, mit denen ich vertraut werden will.

Der erste Teil der Projektarbeit, das Provozieren von Signalfehlern, lässt mich in die asynchrone Signalverarbeitung einblicken und wird meinen VHDL-Coderstil nachhaltig prägen. Im zweiten Teil, dem Entwickeln eines *midi interfaces* lerne ich ein Protokoll zu durchleuchten. Besonders interessant ist die textbasierte *testbench*, welche die Implementation auf Herz und Nieren testet.

Ich möchte Prof. Hans-Joachim Gelke Dank aussprechen. Er lehrt mich viel über kombinatorische Logik. Ebenfalls möchte ich Dr. Matthias Rosenthal danken, der die Arbeit und den Entwicklungsprozess mitträgt.

Aus meiner Sicht ist diese Arbeit vor allem für Software Ingenieure interessant, da sie einen Einblick in die hardwarenahe Programmierung gibt.

Ich freue mich auf kommende VHDL-Projekte.

Inhaltsverzeichnis

1. Einleitung	6
1.1. Ausgangslage	6
1.2. Aufgabenstellung	6
2. Glitches	7
2.1. Glitches in der Digitalen Signalverarbeitung	7
2.2. Ursache für Glitches	7
2.3. Glitches durch Pfadverzögerung	8
2.4. Resultat	10
3. Metastabilität	11
3.1. Metastabiler Zustand	11
3.2. Ursache von Metastabilität	11
3.3. Metastabilität erzeugen	12
3.3.1. Konzept	12
3.3.2. Umsetzung	12
3.4. Resultat Metastabilität provozieren	14
4. Testbench	17
4.1. Device Under Test	17
4.2. Struktur der Input-Datei	17
4.3. Aufstellen der Fehler	18
4.3.1. Einzelne Noten testen	18
4.4. Code Testbench	19
4.4.1. Erstellen eines Package	20
4.4.2. Prozessoptimierung	20
4.5. Ergebnisse Simulation	20
4.5.1. Block Midi Control	20
4.5.2. Block Polyphonie Out	21
5. MIDI Steuerung	22
5.1. Blockschaltbild und Schnittstellen	22
5.2. Das MIDI Kommunikationsprotokoll	23
5.2.1. MIDI Daten Typen	23
5.2.2. Zwei MIDI-Noten-Modi	24
5.3. Umsetzung Midi Control-Block	25
5.3.1. Anforderung an die Finite State Machine und Skizze	25
5.3.2. Implementation Finite State Machine	25
5.4. Resultat Midi Control-Block	27
5.4.1. Implementierte Finite State Machine	27
5.4.2. Simulation Single Mode	27
5.4.3. Simulation Polyphony Mode	28
5.5. Umsetzung Polyphone Out-Block	29
5.5.1. Funktionsbeschreibung	29
5.5.2. Konzept	29
5.5.3. Implementation	30
5.5.4. Resultat Polyphonie Out-Block	30
5.5.5. Simulation parallel Noten ausgeben	30
6. Diskussion und Ausblick	33

7. Verzeichnis	34
7.1. Literatur	34
7.2. Glossar	34
A. Offizielle Aufgabenstellung	I
B. Aufgabenspezifikation für den zweiten Teil	II
C. CD mit Projektdateien	III
D. Top Synthesizer	IV
E. In- und Outputdatei der textbasierte Testbench	V

1. Einleitung

1.1. Ausgangslage

Für den ersten Teil der Arbeit, die Timing Artefakte *glitch* und *metastability* zu demonstrieren, gibt es wenige Referenzprojekte. Da die Zustände ungewollt sind, finden sie als Fehlerquellen Erwähnung in der Literatur [5], [6], [7]. Nur ein Dokument wurde gefunden, das die Erzeugung von *metastability* behandelt [9]. Aus diesem Grund bezieht sich der Nachweis der Timing Artefakte auf Anregungen und auf die Erfahrung von Prof. Hans-Joachim Gelke.

Im zweiten Teil geht es um den Aufbau eines *midi interfaces*. MIDI bedeutet *musical instrument digital interface* und ist ein Standard, der die Beschaffenheit der Hardware wie auch das Kommunikationsprotokoll festlegt [4]. Die MIDI Manufacturers Association dokumentiert die mehrfachen Erweiterungen des MIDI 1.0 Standard [2]. Diese Spezifikationen bildet die Grundlage für den Block *midi control*. Am Institut for Embedded Systems besteht ein *midi uart top*-Block in VHDL von Armin Weiss. In dieser Projektarbeit zu entwickeln sind die zwei Einheiten *midi control* und *polyphony out*. Und anschliessend diese Blocks in das bestehende Synthesizer-Projekt einzubauen. Bei beiden Blocks basiert die Entwicklung auf einer textbasierten *testbench*.

1.2. Aufgabenstellung

Die offizielle Aufgabenstellung befindet sich im Anhang A.

- Erzeugung von Glitches mit einem Zähler und nachgeschaltetem Dekoder. Sichtbarmachung der Glitches mit einem Oszilloskop. Betätigen des asynchronen Resets vom Decoder aus.
- Provozieren und sichtbarmachung von metastabilen Zuständen. Hierfür kann z.B. eine Schaltung mit zwei asynchronen externen Takten aufgebaut werden.

Nach Fertigstellen des ersten Teils, wird die Aufgabenstellung für den zweiten Teil präzisiert (siehe Anhang B).

- Midi Interface for Keyboard für Polyphonie nach Konzept von gelk
 - 10 Frequenz Control Ausgänge zur Steuerung der Tonhöhe des Generators
 - 10 On/Off Ausgänge Ton on/off
 - UART wird geliefert von gelk
 - VHDL wird von Grund auf neu erstellt.
- 10 DDS implementieren und mit Mischer mischen
- Script basierte Testbench. Testbench erzeugt serielle Midi Daten, so wie sie auf dem DIN Stecker vorkommen (logisch)
- Testbench liest eine Testscript Datei ein, in welcher die Tastendrücke eines Keyboards abgebildet werden können.
- Midi Poliphony Spec muss durch die Testbench unterstützt werden können.
- Velocity muss nicht unterstützt werden.
- Kein VHDL code ohne Testbench.
- Block level testbench. Unit Tests.

2. Glitch

2.1. Glitch in der Digitalen Signalverarbeitung

In der Digitalen Signalverarbeitung ist ein *glitch* ein bekanntes, ungewolltes Verhalten, das William I. Fletscher folgendermassen beschreibt:

”Als *glitch* wird eine ungewollte, flüchtige ”Signalspitze” bezeichnet, die Zähler aufwärts zählt, Register löscht oder einen ungewollten Prozess startet.” [6]

Abbildung 2.1 zeigt zwei *glitches* in einem Ausgangssignal.



Abbildung 2.1.: Zwei Glitches im Ausgangssignal

2.2. Ursache für Glitches

Der Auslöser sind ungleichzeitig eintreffende Signale, die durch

1. unterschiedlich lange Signalpfade,
2. unterschiedliche Durchlaufverzögerungen der vorangehenden Flip-Flops oder
3. unterschiedliche Logik-Zeiten

entstehen, und die in ein asynchrones Bauteil geführt werden. Der Dekoder im asynchronen Bauteil entschlüsselt dadurch kurzfristig einen falschen Wert.

Abbildung 2.2 zeigt ein leicht verzögertes enable-Signal (ena) zu einem anders verzögerten Flip-Flop-Eingangssignal (Q). Beide Signale sind mit der Taktfrequenz clk getaktet. Der Ausgang (out) des Flip-Flops weist aufgrund der unterschiedlichen Verzögerungen kurzzeitig *glitches* auf.

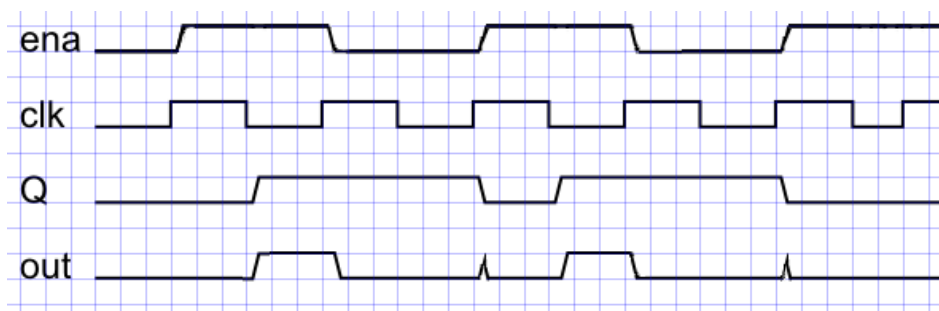


Abbildung 2.2.: Asynchrone Eingangssignale führen zu Glitches

2.3. Glitch erzeugen

2.3.1. Konzept

Glitches werden durch Pfadverlängerung einzelner Werte erzeugt. Ein asynchroner Zähler erhält verzögerte Bitwerte. Zählt man binär auf 15, so kann sich beim Übergang von der Zahl 11 zu 12, die falsche Zahl 15, ergeben, sofern die zwei höheren Bits der Zahl 11 verzögert ankommen (siehe Abbildung 2.3).

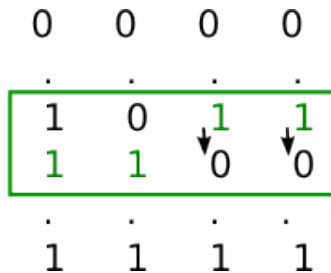


Abbildung 2.3.: Binärwerte des asynchronen Zählers

Die Verzögerung der zwei Bits, wird über Routing umgesetzt.

2.3.2. Implementation

Die Hardware ist das Altera board De2 mit dem FPGA Cyclone II. Kompiliert wird das Projekt mit Quartus 13.0sp, der ältesten Quartus-Version, die den Cyclone II unterstützt.

Die Pfadverlängerung wird über das Routing über die GPIO-Pins des Steckers 1 gemacht (siehe Abbildung 2.6. GPIO 1 ist der Stecker 1). Dekodiert die asynchrone Logik die Zahl 15, wird das Reset-Signal (aus der Logik `reset_counter~{2}`) an den Zähler gesendet und der Zähler beginnt wieder von 0 an zu zählen. Produziert der Dekoder zur falschen Zeit einen Reset, so ist dies eine Fehlkodierung: ein *glitch*.

Das RTL-Diagramm des asynchronen Zählers sieht wie folgt aus:

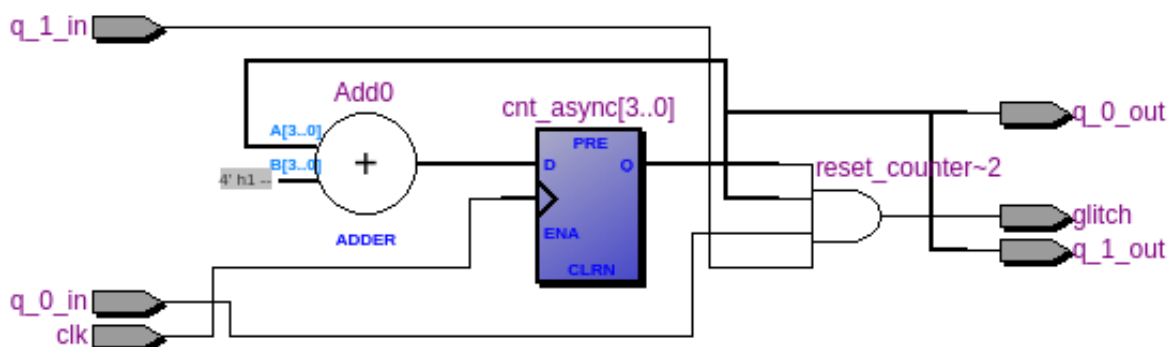


Abbildung 2.4.: Asynchroner Zähler mit Routing erzeugt Glitch

Um die Lösung gegen *glitches* aufzuzeigen, wird dem asynchronen Zähler zur Synchronisation ein Flip-Flop nachgeschaltet. Dadurch werden die asynchronen Zustände übersehen.

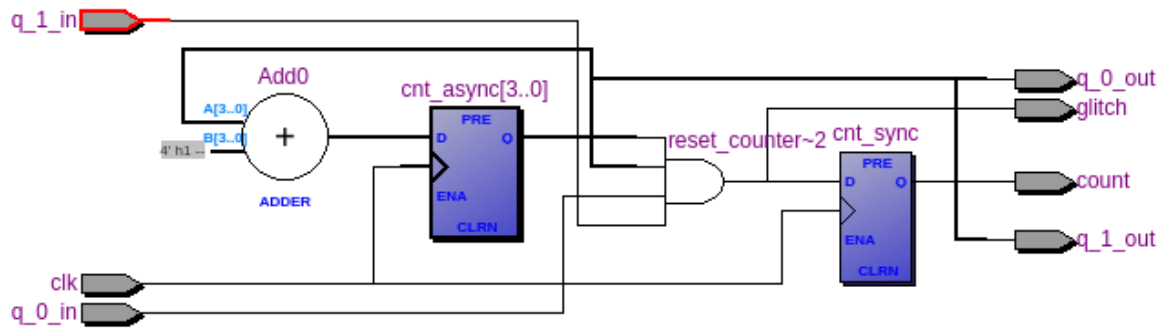


Abbildung 2.5.: Glitch-Zähler und synchroner Zähler dazu

Die Reset-Signale aus der Vergleichslogik (`reset_counter\{}2`) des asynchronen Zählers wie die des synchronisierten Zählers werden an die GPIO Stecker ausgegeben, ebenso der Systemtakt. In der Abbildung 2.6) wird das Signal des asynchronen Zählers als Glitch und das Signal des synchronisierten Zählers als Count benannt. In der GPIO-Pinbelegung sieht man auch die Nutzung der zwei oberen Pin-Reihen für das Routing (benannt mit Routing OUT, IN). Der Systemtakt wird als CLK ausgegeben.

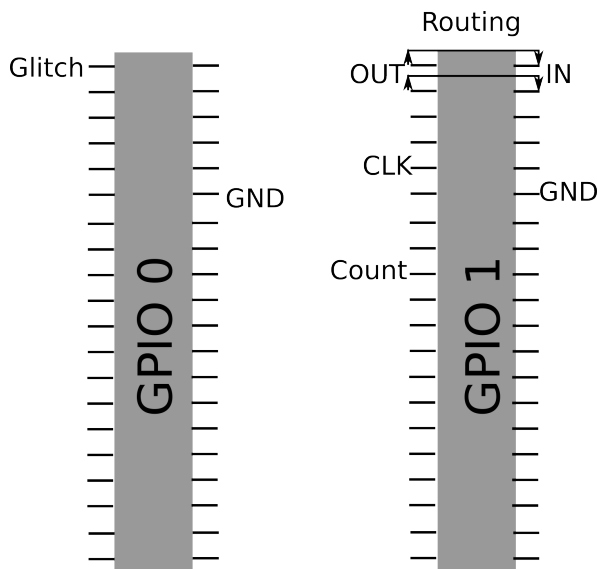


Abbildung 2.6.: GPIO Anschlüsse

2.4. Resultat

Der Reset des asynchronen Zählers (CH 1), der synchronisierte Reset (CH 2) und der Systemtakt (CH 3) werden am KO ausgegeben. Durch die Synchronisation wird der Wert um 1 Periode (= 20 ns) verzögert.

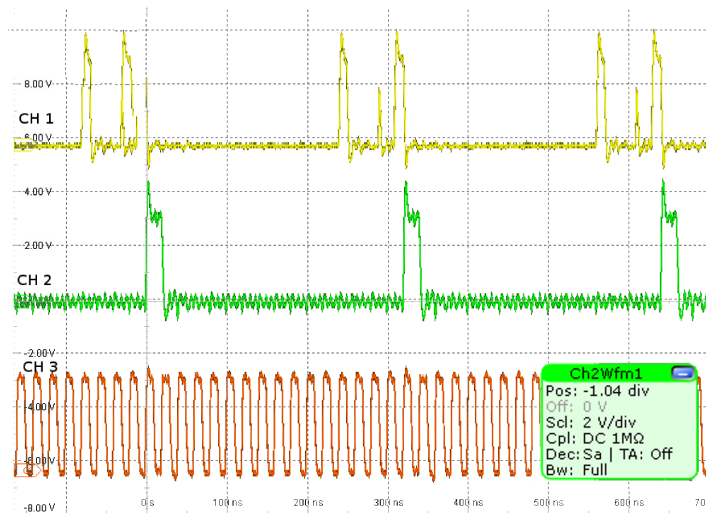


Abbildung 2.7.: Glitch (gelb), Zähler (grün) und Takt (orange)

Das *glitch* trifft in der im Übergang von der 11 zur 12 Periode (nach 240 ns) regelmässig auf. Dies ist das zu erwartende Ergebnis. Ein kurzzeitiges asynchrones Verhalten findet sich auch im Übergang von der 13 zur 14 Periode. Dies ist wenn der binäre Wert 1101 auf 1110 wechselt. Da die zwei niederwertigen Bits verzögert sind, ist das Dekodieren des Wertes 1111 plausibel.

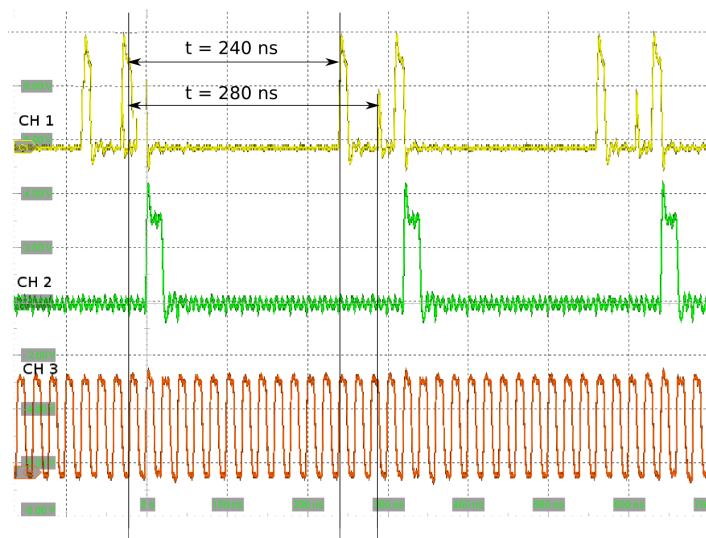


Abbildung 2.8.: Zeitanalyse Glitches

3. Metastabilität

3.1. Metastabiler Zustand

Metastabilität bedeutet, dass der Ausgang eines Flip-Flops nicht dem Eingang entsprechen *muss*. In einem metastabilen Zustand kann ein Ausgang korrekt sein, muss aber nicht. Im Idealfall wählt ein Flip-Flop seinen Ausgangswert selbst (siehe Abbildung 3.1 oberes Signal). Im schlechten Fall “hängt” sich das Flip-Flop “auf” und toggelt permanent zwischen ‘0’ und ‘1’ (Abbildung 3.1 unteres Signal).

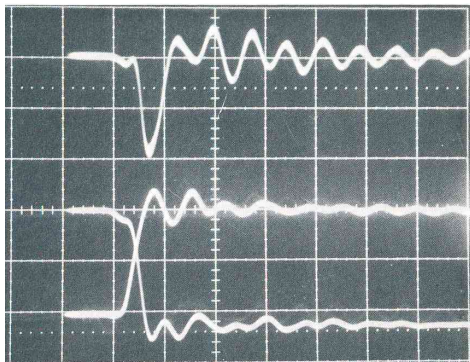


Abbildung 3.1.: Metastabilität schlimmster Fall [7]

3.2. Ursache von Metastabilität

Die Ursache unsicherer Ausgangswerte liegen darin, dass das Inputsignal eines Flip-Flops zur falschen Zeit wechselt.

”If data inputs to a flip-flop are changing at the instant of the clock pulse, a problem known as *metastability* may occur. In the metastable case, the flip-flop does not settle in to a stable state” [5]

”If the amplitude of the runt pulse is *exactly the threshold level of the SET input of the output cell*, the cell will be driven to its metastable state. The metastable state is the condition that is roughly defined as ”half SET and half RESET” [7]

Trifft der anzulegende Wert zu spät ein wird die *setup time* verletzt und wird der Signalwert zu früh entwendet, verletzt die *hold time*. Metastabilität kann vermieden werden, wenn diese zwei Zeiten strikt eingehalten werden:

”Metastability is avoided by holding the information stable before and after the clock pulse for a set period of time, called the setup time for the data line an the hold time for the control line.” [5]

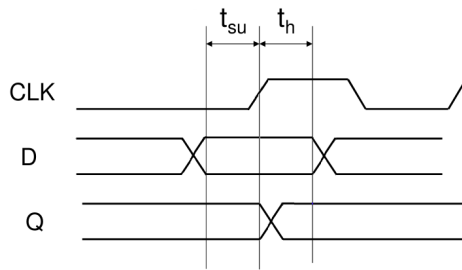


Abbildung 3.2.: Einhalten der Datenzeiten

Um Metastabilität zu vermeiden, sollte die Logik möglichst klein, die Bauteile nahe beieinander und der Systemtakt an die längste Pfadzeit angepasst werden. Der maximal erlaubte Systemtakt kann in quartus mit dem Timequest Time Analyser abgefragt werden.

3.3. Metastabilität erzeugen

3.3.1. Konzept

Aufgebaut wird ein System mit zwei *clock domains*. *Clock domain 1*, Gebiet 1 genannt, beinhaltet einen Zähler, der an die *clock domain 2*, als Gebiet 2 bezeichnet, asynchrone Impulse sendet. Gebiet 2 verarbeitet diese Impulse in einer *finite state machine*. Bei korrekter Funktionsweise wechselt die *fsm* zwischen den definierten *states*. Funktioniert sie falsch, fällt die *fsm* in einen *state*, den sie nicht implementiert hat.

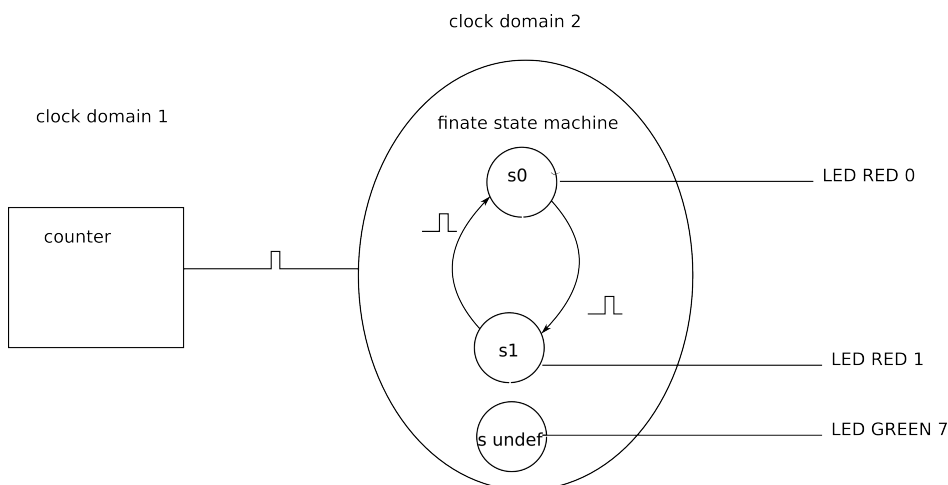


Abbildung 3.3.: Konzept Metastabilität nachweisen

3.3.2. Implementation

Als Hardware wird das altera board DE2 genommen und mit der Software quartus 13.Osp1 gearbeitet. Da die zwei Takte nicht Vielfache voneinander sein dürfen, wird für den Zähler ein Takt von 27 MHz und für die *fsm* ein Takt von 50 MHz gewählt. Der Takt des Zählers ist leicht schneller als die Hälfte des *fsm*-Taktes und schiebt sich vorwärts (siehe Abbildung 3.4). Das Verletzen der *setup time* ist eine Frage der Zeit.

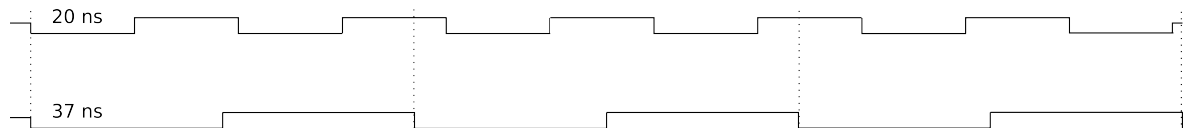


Abbildung 3.4.: Die zwei Taktzeiten

Die Zustandsüberprüfung erfolgt über das Ausgeben des aktuellen Zustands auf den zwei roten LEDs.

- Zustand = s0
Rote LED 0 ist an
- Zustand = s1
Rote LED 1 ist an
- Zustand = *OTHERS*
Grüne LED 17 ist an

Funktioniert die *fsm*, blinken die zwei roten LEDs abwechselungsweise. Fällt die *fsm* in einen undefinierten Zustand, leuchtet die grüne LED. Um die Ursache der Metastabilität, das Verletzten der *setup time* zu verhindern, wird eine optionale Synchronisation durch Switch 17 eingebaut. Ist Switch 17 auf '1', wird der Puls der *clock domain* 27 MHz durch ein Flip-Flop auf 50 MHz synchronisiert.

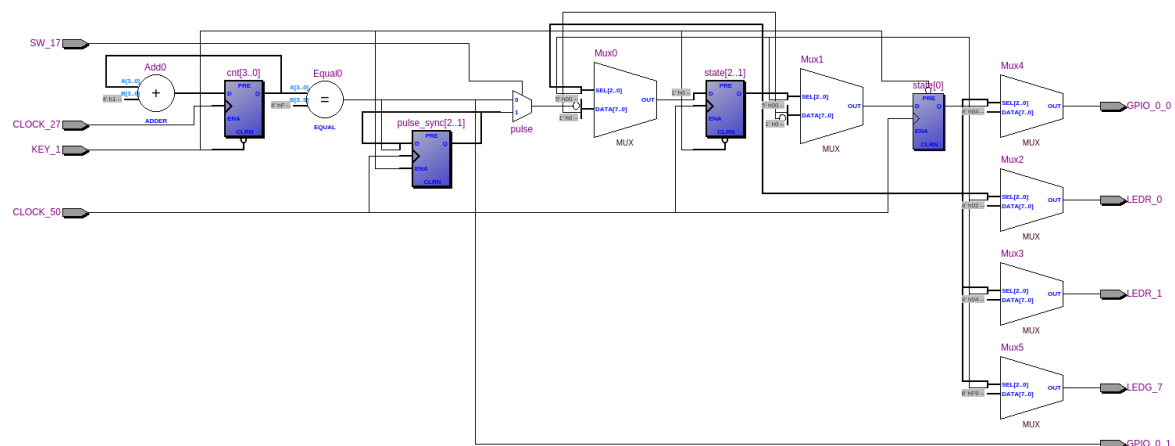


Abbildung 3.5.: RTL mit Synchronisations-Switch

3.4. Resultat

Das Resultat ist, dass das Board unmittelbar nach Einstellen in den metastabilen Zustand fällt und die grüne LED leuchtet. Wird Reset gedrückt, folgt ein kurzes Aufblinken der zwei roten LEDs und wieder die grüne LED.

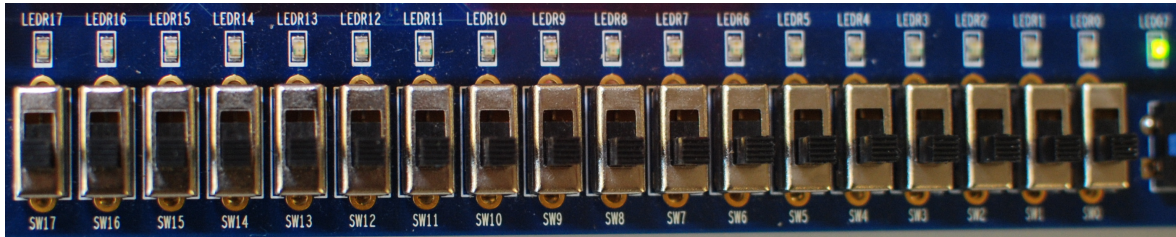


Abbildung 3.6.: Metastabiler Zustand

Wird die Synchronisations-Schaltung betätigt, leuchten beide roten LEDs auf. Die *fsm* wechselt zwischen den states s0 und s1 hin und her. Das Verbleiben in den zwei definierten Zuständen s0 und s1 funktioniert auch nach einem Tag noch.

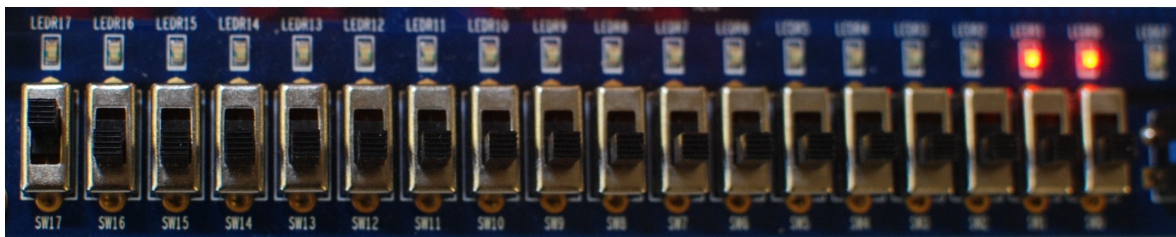


Abbildung 3.7.: Switchschalter ON: Rote LEDs leuchten

Wird das Wechseln zwischen den zwei *states* am KO ausgegeben, so erkennt man kein wiederkehrendes Muster im Wechseln. Der Grund ist, dass der Takt 27 MHz kein Bruchteil von 50 MHz.

CH 1 = Rote LED

CH 2 = Synchronisierter Puls

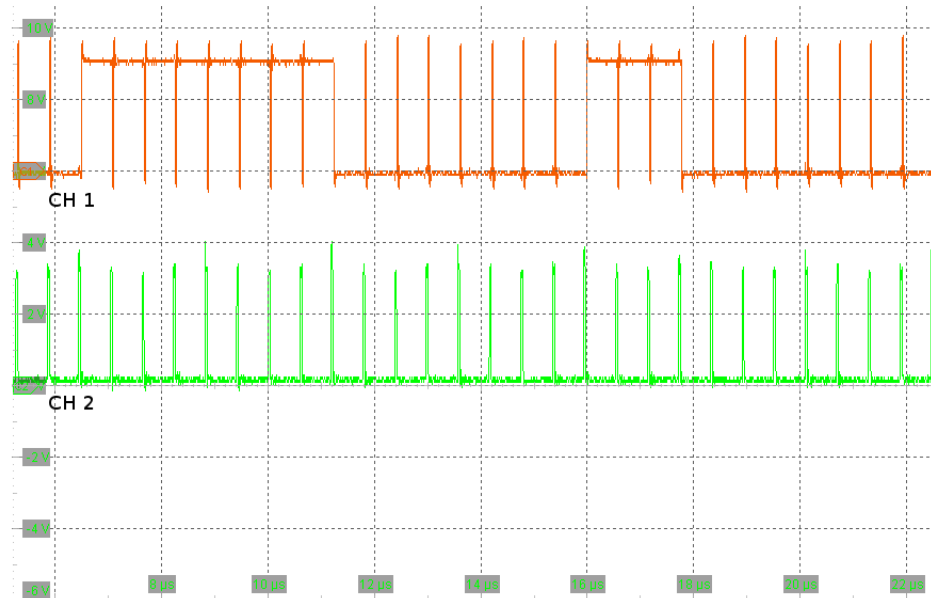


Abbildung 3.8.: Erster Teil des unregelmässigen Wechsel zwischen Zustand s0 und Zustand s1

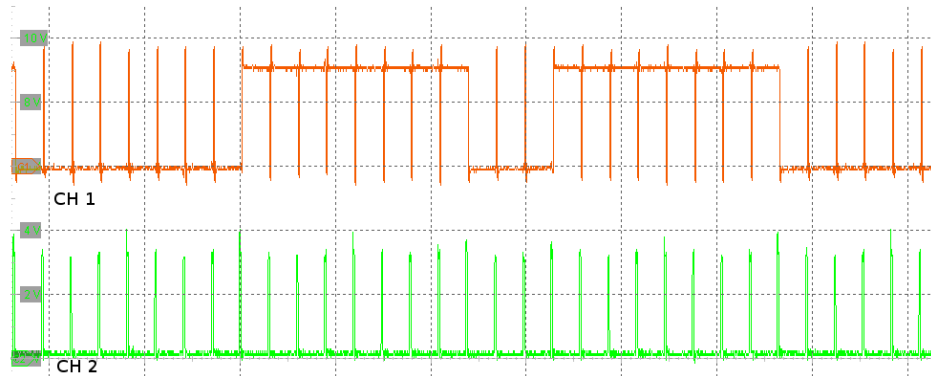


Abbildung 3.9.: Zweiter Teil des unregelmässigen Wechsel zwischen Zustand s0 und Zustand s1

Im Zustand der Metastabilität sind die Pulse nicht synchronisiert und die rote LED geht nicht an.

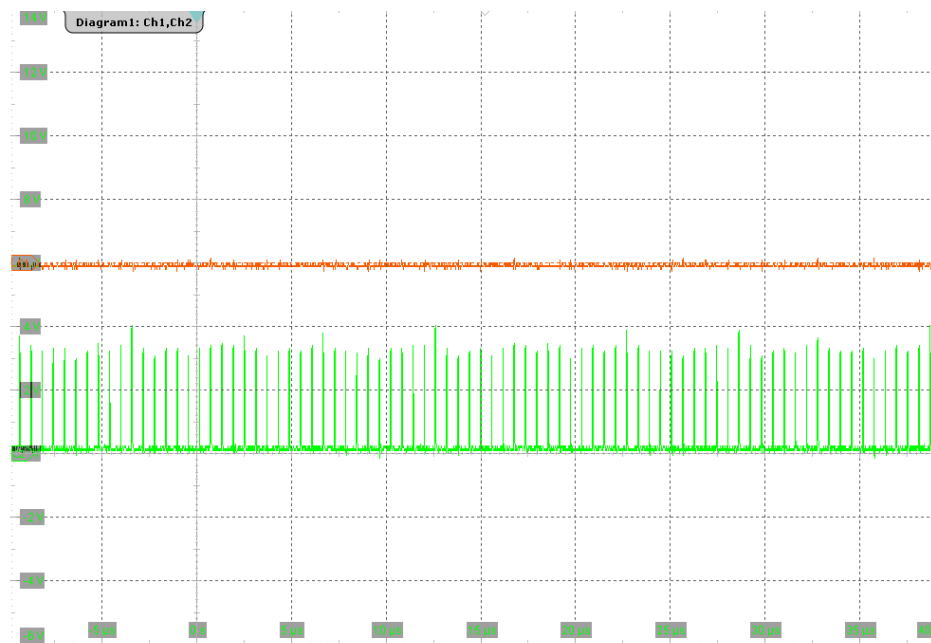


Abbildung 3.10.: Metastabiler Zustand

4. MIDI-Steuerung

4.1. Das MIDI-Kommunikationsprotokoll

Werden MIDI Daten übermittelt, so unterscheidet der Standard zwei Typen an Daten [2].

4.1.1. MIDI-Daten-Typen

Status Bytes

Status bytes sind 8 Bit lang und das MSB ist immer logisch '1'. *Status bytes* dienen dem Identifizieren der nachfolgenden *data bytes*. Das *status byte* definiert die Datenstruktur der folgenden *data bytes*.

MIDI behält einen Status, bis ein neues *status byte* folgt. Dieses Verhalten ist als *running status* bezeichnet. Dieses Verhalten ist für Polyphonie relevant, da der Zustand bleibt, bis ein neues *status byte* folgt.

Data Bytes

Gemäss Spezifikation folgen einem *status byte* ein oder zwei Bytes. Das MSB ist immer logisch '0'. Die Werte können von 0x00 bis 0x7F sein. Das bedeutet, dass MIDI maximal 128 Noten unterscheiden kann.

Data bytes können unterschiedliche Informationen enthalten. Im Kontroller sind Notenwerte, Geschwindigkeit des Anschlages relevant.

Je nach dem *status byte* werden die *data byte* anders interpretiert.

”Empfänger sollen so konzipiert sein, dass zuerst alle *data bytes* empfangen werden und ein neues *status byte* kommt. Danach werden ungültige Daten verworfen. Einzige Ausnahme ist der *running status*, bei dem nicht bis zum Ende gewartet wird.” [2].

Ungültige Bytes

”Alle *status bytes*, die nicht implementierte Funktionen enthalten und alle ihnen folgenden *Data Bytes* sollen vom Empfänger verworfen werden.” [2]

MIDI Geräte sollen ausdrücklich beim Ein- und Abstellen darauf bedacht sein, dass keine undefinierten Bytes gesendet werden [2].

Diese Anforderung ist wichtig beim Implementieren der *finite state machine* und der *testbench* (siehe Kapitel 4)

Midi Bytes binär

- ‘0xxx xxxx’: Definition *data byte*
- ‘1xxx xxxx’: Definition *status byte*
- ‘1000 xxxx’: Definition NOTE OFF
- ‘1001 xxxx’: Definition NOTE ON
- ‘1010 xxxx’: Definition POLYPHONY
- ‘100x xxxx’: Erste drei Bits der *status bytes* NOTE ON (0x90) und NOTE OFF (0x80)

4.1.2. Zwei MIDI-Noten-Modi

Datenstruktur

Die Datenstruktur der zwei MIDI-Noten-Modi beginnt mit dem *status byte* (grau in der Abbildung 5.3). Es folgt der Notenwert (hier Dummy-Wert von 0x11 eingetragen) und die Geschwindigkeit. Letztere hat im *single mode* keine spezifische Bedeutung, im *polyphony mode* bestimmt die Geschwindigkeit, ob die Note an oder ab ist.

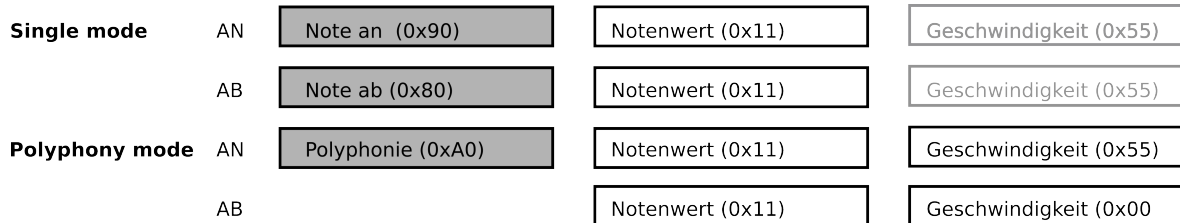


Abbildung 4.1.: MIDI Spezifikation für Datenstruktur einzelne Note und Polyphonie

Unterschiedlich zu behandeln ist die Funktion des *status bytes*. Im *single mode* wird mit dem *status byte* der Zustand an oder ab mitgegeben. Im *polyphony mode* wird nur der Noten-Modus mitgeteilt und das *status byte* hat keine weiteren Funktionalitäten. In der Abbildung wird der Platz von Note an oder ab bezüglich dem Noten-Byte durch graue Markierung veranschaulicht. Die zeitliche Reihenfolge der ist umgekehrt, was in der Token-Verarbeitung berücksichtigt werden muss.

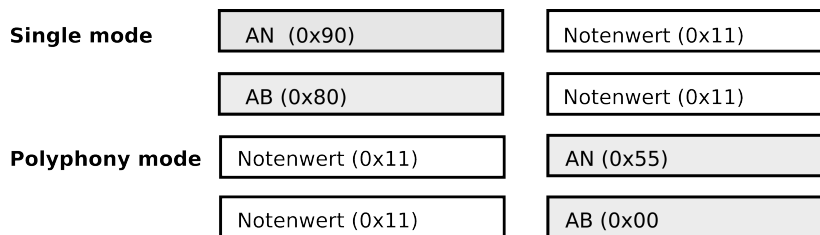


Abbildung 4.2.: Blockschaltbild Device unter Test

Wegen der unterschiedlichen Bedeutung der eingegangenen Token, behandelt die *fsm* die zwei Noten-Modi und deren Noten- und Geschwindigkeitszustände unabhängig voneinander.

4.2. MIDI-Steuerung: Blockschaltbild und Schnittstellen

Analog zum bestehenden Synthesizer-Projekt erhält der VHDL-Block einen englischen Namen: Die MIDI-Steuerung ist als Block mit dem Namen *midi interface* implementiert. Als erstes die Zusammenfassung der internen Blöcke. Die zwei entwickelten Blöcke *midi control* und *polyphony out* sind grau markiert (siehe Abbildung 5.1). Gegeben ist der Block *uart top*.

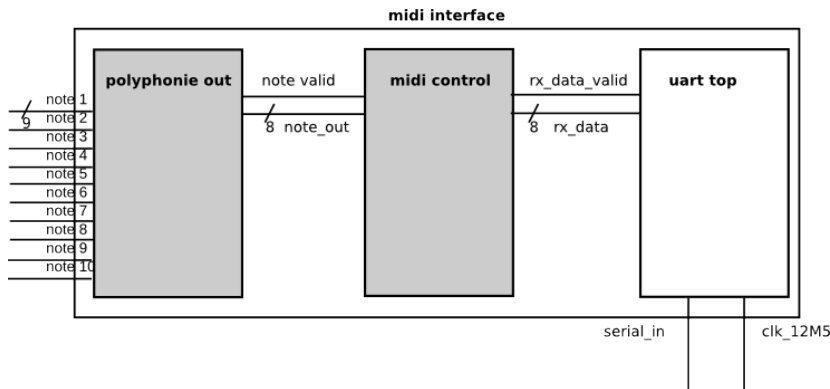


Abbildung 4.3.: Blockschaltbild Midi Interface

Definition der Schnittstellen

Uart Top Ausgang

- 8-Bit-Signal: Bytweise Dekodierung der Midi Daten
- 1-Bit-Signal: Übermittelt Gültigkeit der Daten

Midi Control Ein- und Ausgang

- Empfangen von 8-Bit Midi Daten,
- Empfangen, ob Daten korrekt sind (1 Bit)
- Übermittelt 9-Bit-Notenvektor (Struktur des Vektors in Abb.5.2)
- Übermitteln, ob Daten korrekt (1 Bit)

Polyphony Out Ein- und Ausgang

- Eingang eines 9-Bit-Notenvektors
- Eingang, ob Daten gültig sind
- Ausgabe von 10 Notenvektoren zu 9-Bit.

Im vorgegebenen Konzept für *polyphony out* (siehe Unterkapitel 5.5.2) ist die Schnittstelle zum Polyphony Out-Block als ein 9-Bit-Signal definiert. Das MSB dient als Flag, ob die übermittelte Note an oder ab ist.

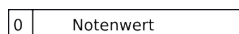


Abbildung 4.4.: Aufbau Notenvektor

Als nächstes wird die MIDI 1.0 Spezifikation, erklärt, nach der Block *midi control* aufgebaut ist. Die Umsetzung des *polyphony out*-Blocks bildet den Abschluss dieses Kapitels.

4.3. Midi Control-Block

4.3.1. Konzept mit Finite State Machine

Der Controller wird über eine *finite state machine* implementiert. Ausgehend von der Spezifikation 5.2 sind drei Eckpunkte berücksichtigt:

1. Unterscheiden von *status byte* und *data byte*
2. Unterschiedliche Interpretation der *data bytes* abhängig vom *status byte*.
3. Verwerfen aller falschen *status byte* oder *data bytes*

Vereinfacht verhält sich die *fsm* wie in Abbildung 5.5 gezeigt.

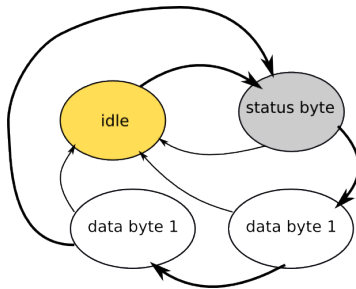


Abbildung 4.5.: Skizze der fsm

Startpunkt der Verarbeitung ist das *status byte* (grau hinterlegt). Danach führt die Verarbeitung durch die zwei *data bytes*.

In jedem Zustand, werden ungültige Daten verworfen, und führen zurück zu *idle*. Die Verarbeitung wird fortgesetzt, wenn das nächste *status byte* folgt.

Nicht angeschrieben sind die Übergangsbedingungen: `data_valid = '1'` wechselt zwischen den Zuständen und `data_valid = '0'` verbleibt im Zustand. Die breiteren Pfeile heben die fehlerfreie Datenverarbeitung hervor.

4.3.2. Implementation der Finite State Machine

Aufgrund der unterschiedlichen Datenstruktur für den *polyphony mode* und den *single mode* besitzen beide Noten-Modi ihre eigenen Zustände (siehe Abbildung 5.6).

Die implementierten Zustände sind

- **idle**: Alle nicht näher spezifizierten Vorfälle verwerfen
- **single**: Eintreten in *single mode* durch *status bytes* 0x80 oder 0x90
- **note_s**: Erstes *data byte* im *single mode*
- **velocity_s**: Zweites *data byte* im *single mode*
- **polyphonie**: Eintreten in *polyphony mode* durch *status byte* 0xA0
- **note_v**: Erstes *data byte* im *polyphony mode*
- **velocity_v**: Zweites *data byte* im *polyphony mode*

Abbildung 5.6 definiert die Übergangsbedingungen. Drei generelle Verhaltensweisen sind vereinfacht angegeben:

- `data_valid = '0'`
Im aktuellen Zustand bleiben.
Dargestellt mit Pfeil an Ort

- `data_valid = '1'`
Grundbedingung für Zustandswechsel
Gilt implizit zu jedem Pfeil und dessen Bedingung dazu
- `data(7) = '1' and (data(7 downto 5) /= '100' or data(7 downto 4) /= '1010')`
Status bytes, die nicht *polyphony* oder *single mode* bedeuten, verworfen
Dargestellt durch Pfeil oben rechts zu idle. Gilt für jeden Zustand

Die Übergangsbedingungen detektieren die Binärstruktur der MIDI-Daten, die im Unterkapitel 5.2.1 aufgelistet ist.

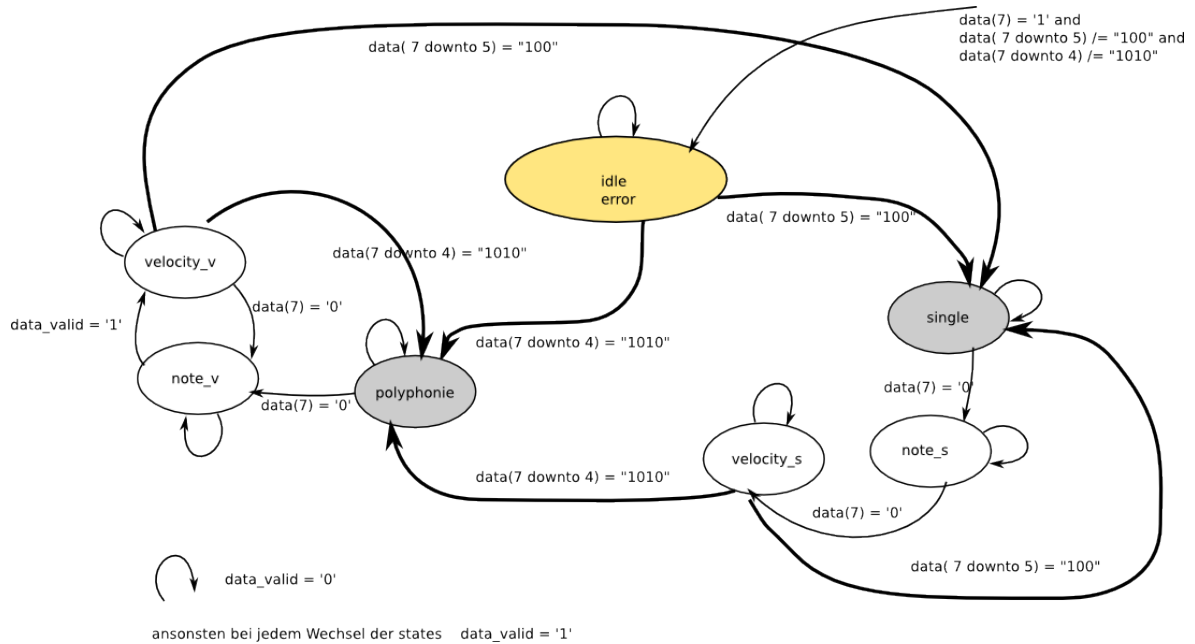


Abbildung 4.6.: Übergänge der fsm

Alle drei Anforderungen 5.3.1, die sich aus der MIDI-Spezifikation ergeben, sind implementiert:

1. Vor jedem *data byte* muss ein *status byte* eingegangen sein. Die *finite state machine* fragt im *idle* Zustand nur nach den *status bytes*. Nach dem *status byte* erwartet die *finite state machine* *data bytes*.
2. Die unterschiedliche Datenstruktur der zwei Noten-Modi ist *mode*-spezifisch implementiert:
Im *single mode* wird das vierte Bit des *status bytes* zum Setzen von an und ab verwendet.
Im *polyphony mode* wird das zweite *data byte*, die Geschwindigkeit zum Setzen der Note auf an oder ab verwendet.
Geschwindigkeit = NULL ist als Note aus implementiert.
3. Ungültige Bytes sind verworfen, und die *fsm* kehrt in den *idle* Zustand zurück.

4.4. Resultat

4.4.1. Implementierte Finite State Machine

Abbildung 5.7 zeigt die in quartus generierte fsm des Blocks *midi control*.

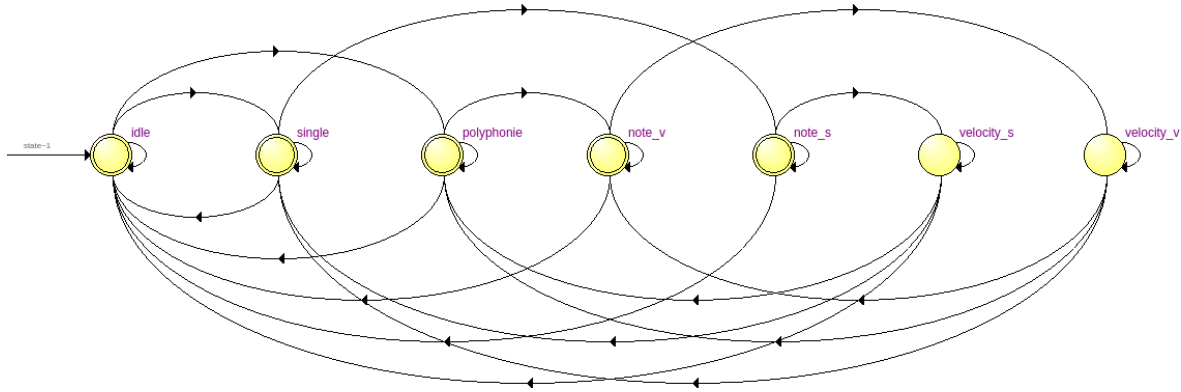


Abbildung 4.7.: Implementierte fsm im Block Midi Control

4.4.2. Simulation Single Mode

Der Aufbau der *testbench* und dadurch der Simulation wie auch der Input-Daten ist in Kapitel 4 eingehend beschrieben. Hier steht exemplarisch *eine* Auswertung für den *single mode* und im nächsten Unterkapitel für den *polyphony mode*, damit das korrekte Funktionieren der Implementation dokumentiert ist.

Input Daten (Zeile 3, siehe Anhang E)

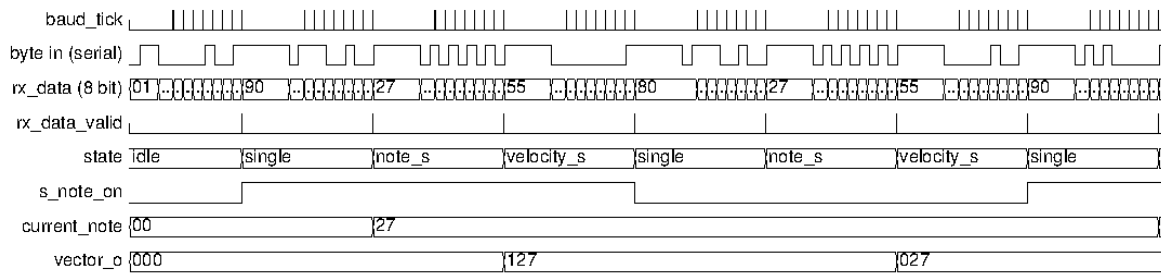
singl	55	90	27	80	27	90	02	00	00
-------	----	----	----	----	----	----	----	----	----

Beschreibung der Befehle

- 55 als Dummy-Velocity für alle Noten
- Note an
- Notenwert 27
- Note ab
- Notenwert 27
- Note an
- Notenwert 02
- Dummywerte

Erwartetes Resultat

Der Controller erkennt die Note 27, schaltet diese an und gibt am Ausgang den Vektor "Note-27-AN" aus. Dieselbe Note wird nochmals detektiert, diesmal als ab und der Vektor am Ausgang zeigt "Note-27-AB" an. Die nächste Note hat den Wert 2 und wird auf AN gesetzt. Der Ausgang gibt "Note-2-AN" aus.

Abbildung 4.8.: fsm für *single mode*

- Das Signal **rx_data** detektiert die Befehle (0x90) und (0x80).
- Der Controller interpretiert *single modus*.
- Die Zustandsabfolge ist korrekt: **single**, **note_s**, **velocity_s**.
- Die Zustände werden bei **rx_data_valid** = '1' geändert.

Die Simulation zeigt, dass die Notenwerte korrekt gespeichert sind und dass das An- und Abstellen der Noten funktioniert. Am Ausgang erscheint der zusammengesetzter Vektor aus den 8 Notenbits und einem vorangestellten Bit, das detektiert, ob die aktuelle Note an oder ab ist.

4.4.3. Simulation Polyphony Mode

Input Daten (Zeile 9, siehe Anhang E)

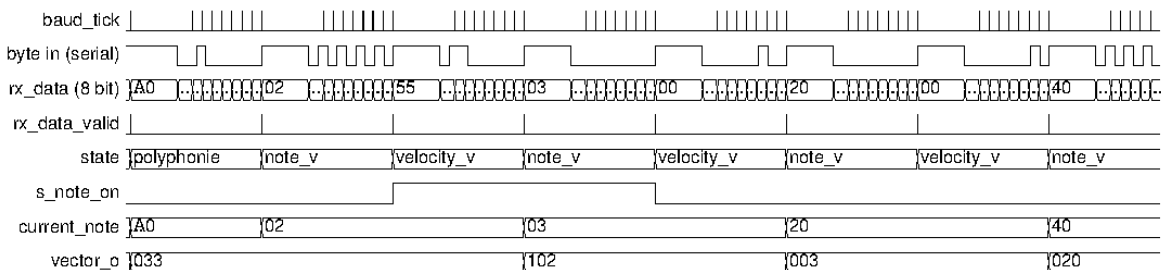
polyp 02 55 03 00 20 00 40 55 00

Beschreibung der Befehle

- Notenwert 02
- Note an
- Notenwert 03
- Note ab
- Notenwert 02
- Note ab
- Notenwert 4
- Note an

Erwartetes Resultat

Der Kontroller erkennt die Note 02, schaltet diese an und gibt am Ausgang den Vektor "Note-02-AN" aus. Die Note 03 wird detektiert, auf ab gesetzt und der Vektor am Ausgang zeigt "Note-03-AB". Die nächste Note hat den Wert 2 und wird auf ab gesetzt. Der Ausgang gibt "Note-2-Ab" aus. Als letztes folgt die Note 40, die angestellt wird.

Abbildung 4.9.: fsm im *polyphony mode*

- Das Signal `rx_data` detektiert den Befehle (0xA0).
- Der Controller interpretiert *polyphony mode*.
- Die Zustandsabfolge ist korrekt: `polyphonie`, `note_v`, `velocity_v`.
- Der Controller wartet mit dem Setzen der Note am Ausgang, bis klar ist, ob die Note an oder ab ist.
Keine kurzfristig falschen Noten am Ausgang, die ab sind.
- Zustände werden bei `rx_data_valid = '1'` die Zustände geändert.
- Noten können beliebig an- und abgestellt werden

4.5. Polyphony Out-Block

4.5.1. Funktionsbeschreibung

Der Polyphone Out-Block speichert die empfangenen Signale in 10 Registern. Bei jeder neuen Note wird geprüft, ob der Wert im Register besteht und ob das ON/OFF-Bit der gespeicherten neu gesetzt werden muss. Der Block gibt 10 Noten parallel aus.

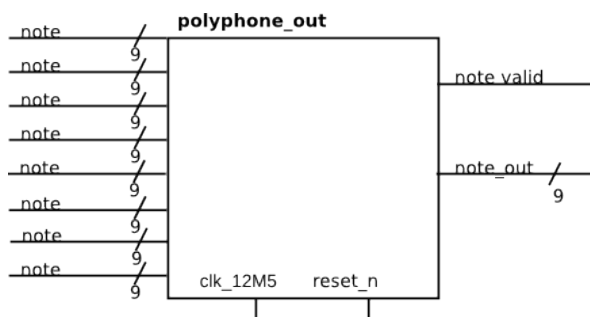


Abbildung 4.10.: Polyphone Out-Block

4.5.2. Konzept

Der empfangene Notenwert wird mit den gespeicherten Notenwerten verglichen. Ist eine Note vorhanden, wird das ON-OFF-Bit geprüft und aktualisiert. Keine Note darf zweimal gespeichert sein. Sind alle 10 Registerplätze besetzt, wird die neue Note in ein Register mit abgeschaltetem ON-OFF-Bit gesetzt.

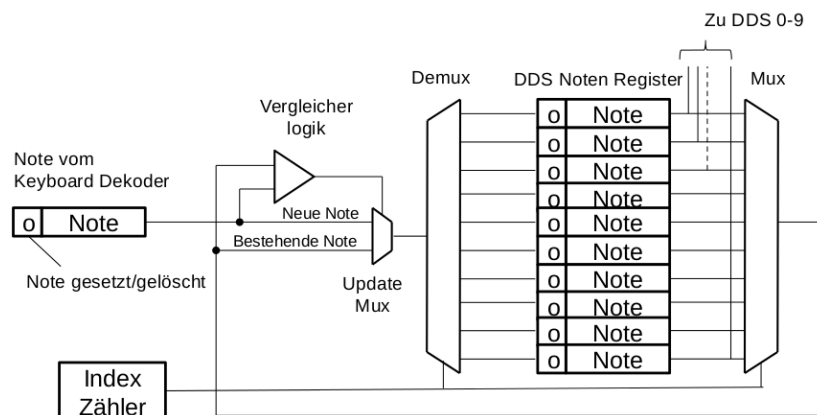


Abbildung 4.11.: Konzept Polyphonie Block [8]

4.5.3. Implementation

Der Ablauf des Speicher-Vorgangs der neuen Note ist aus der Abbildung 5.12 ersichtlich.

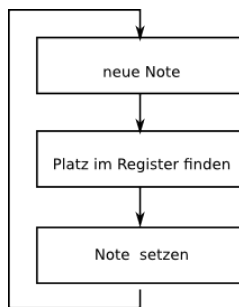


Abbildung 4.12.: Ablauf Note speichern

Umgesetzt wird das Suchen eines Speicherplatzes innerhalb der 10 Registern mit einer Input-Logik, die mit folgenden 3 Vergleichen arbeitet:

1. Liegt der Notenwert in einem Register ?
2. Ist ein Register unbenützt ?
3. Welches Register hat einen abgeschalteten Notenwert ?

Sobald eine Frage mit Ja beantwortet wird, wird der Registerindex gespeichert und als Output des Logik-Prozesses zur Verarbeitung weiter gegeben.

Durch den übermittelten Index-Wert weiss der Speicher-Prozess, in welches Register die neue Note gespeichert werden soll.

Die Werte aller 10 Register werden am Ausgang parallel ausgegeben.

4.6. Resultat

Die Ziele sind:

- dass jeder Notenwert in ein anderes Register gespeichert wird und kein Notenwert zweimal abgelegt wird.
- An und Ab müssen in dasselbe Register gespeichert werden.
- Dieses Konzept ist unabdingbar für den *polyphony mode* und stört die Notenausgabe im *single mode* nicht.

4.6.1. Simulation parallel Noten ausgeben

Input Daten (Zeile 3, 5, 7 und 9 siehe Anhang E)

singl	55	90	27	80	27	90	05	00	00
singl	55	90	73	80	73	90	16	00	00
polyp	71	55	02	55	33	55	08	00	00
polyp	20	55	03	00	20	00	40	55	00

Beschreibung der Befehle

- Dummywert Velocity 0x55
- Note an
- Notenwert 27
- Note ab
- Notenwert 27

- Note an
- Notenwert 05
- Dummywert Velocity 0x55
- Note an
- Notenwert 73
- Note ab
- Notenwert 73
- Note an
- Notenwert 16
- Notenwert 71
- Note an
- Notenwert 02
- Note an
- Notenwert 33
- Note an
- Notenwert 08
- Note ab
- Notenwert 20
- Note an
- Notenwert 03
- Note ab
- Notenwert 20
- Note ab
- Notenwert 40
- Note an

Erwartetes Resultat

- Note 27 wird angestellt (Vektor ist 127), Note 27 wird abgestellt (Vektor ist 027). An und Abstellen sind in demselben Register abgelegt.
- Note 05 wird angestellt und in neuem Register abgelegt, da dieses Register unbenutzt ist.
- Note 73 wird angestellt und nachher abgestellt. Beides in neuem Register.
- Note 16 wird angestellt und in neues Register gelegt.
- Note 71 wird angestellt und in neues Register gelegt.
- Note 02 wird angestellt und in neues Register gelegt.
- Note 33 wird angestellt und in neues Register gelegt.
- Note 08 wird abgestellt. (Neues Register stört nicht).
- Note 20 wird angestellt und in neues Register gelegt.
- Note 03 wird abgestellt. (Neues Register stört nicht).
- Note 20 wird abgestellt. Muss im 20-Notenregister sein.
- Note 40 wird angestellt und in neues Register gelegt.

tb_note_0	127	027
tb_note_1	000	105
tb_note_2	000	173 073
tb_note_3	000	116
tb_note_4	000	171 071
tb_note_5	000	102
tb_note_6	000	133 033
tb_note_7	000	008
tb_note_8	000	120 020
tb_note_9	000	003

Abbildung 4.13.: Simulation des Blocks Polyphonie Out

Die Simulation verhält sich exakt nach Spezifikation.

5. Testbench

Test Driven Development bedeutet, dass vor oder parallel zur Entwicklung einer *unit* (im Folgenden Block genannt) der *unit-test* entwickelt wird [3]. Beim textbasierten Testen stammen die Befehle aus einer Input-Datei, und die Ergebnisse werden in einer Datei abgelegt.

5.1. Device Under Test

Das Device Under Test (DUT) ist das midi interface (siehe Abbildung 4.1). Das Ziel ist, dass das MIDI-Signal in den Block geführt wird und am Ausgang 10 Notenvektoren anliegen mit je 8 Notenbits und einem Bit, das besagt, ob die Note an oder ab ist.

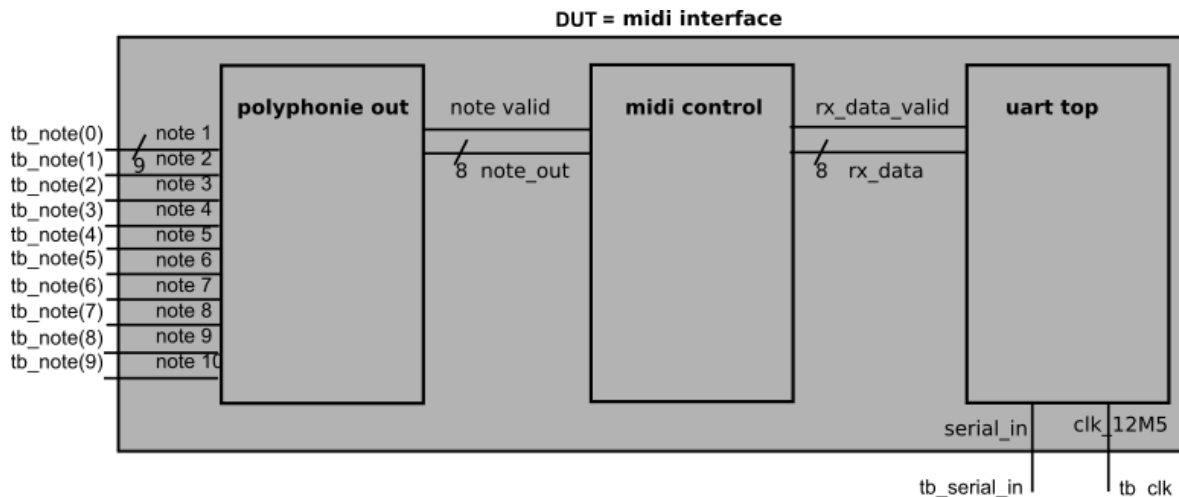


Abbildung 5.1.: Blockschaltbild Device under Test

Die *testbench* wird mit Daten der Input-Datei gespeisen. Die Endversion der Input-Datei und der Testbericht liegen im Anhang E. In den Unterkapiteln wird der Aufbau der Input-Datei, das Entwickeln der Test-Fälle, und die Umsetzung im VHDL-Code beschrieben.

5.2. Struktur der Input-Datei

Die Test-Datei ist zeilenweise strukturiert.

Verarbeitungsmodus

Jede Zeile beginnt mit dem Verarbeitungsmodus. Bei der Input-Datei besteht der Verarbeitungsmodus aus fünf Buchstaben.

```
reset    00  00  ...
singl    90  27  ...
polyp    71  55  ...
```

Tokenstruktur

Nach dem Bearbeitungsmodus folgen die Daten. Jede Zeile hat gleichviele Datenpakete (Tokens). Die *testbench* ordnet jedem MIDI-Datenpaket (siehe 5.2.1, Beschreibung der MIDI-Daten) innerhalb der Zeile eine Bedeutung zu. Je nach Bearbeitungsmodus ist die Bedeutung der Token anders.

Die Tokenstruktur leitet sich aus der MIDI-Datenstruktur im *polyphony mode* und im *single mode* ab (siehe 5.2.2). In den nachfolgenden zwei Token-Beispielen bezieht sich die obere Zeile auf den *polyphony mode* und die untere auf den *single mode*.

In der *testbench midi interface* haben Tokens folgende Bedeutung:

mode_p	Note	Velocity	Note	Velocity	Note	Velocity	Note	Velocity	Anzahl Noten
mode_s	Dummy	Status	Note	Status	Note	Status	Note	Dummy	Dummy

Dummy ist die Bezeichnung für das Einlesen nicht relevanter Werte. Diese nicht relevanten Werte sind in der Input-Datei gesetzt, um die Verarbeitungsstruktur beim Einlesen der Token zu vereinfachen. Der Dummywert wird beim Einlesen verworfen.

5.3. Aufstellen der Fehler

Zu Beginn hatten die Testfälle nur drei Tokens und testeten die Grundfunktionen:

```
single mode note an/ab
singl 90 27
singl 90 27
```

```
polyphone note an/ab
polyp 71 55
polyp 71 00
```

Komplexere Testfälle zeigten, dass eine Test-Zeile mehr Tokens braucht. Die Endversion der Input-Datei kann mit einer Zeile bis 4 Noten an und abstellen.

5.3.1. Einzelne Noten testen

Testfälle

Getestet sind auch Kombinationen unter den Fällen, die aus Übersichtlichkeit nicht alle aufgeschrieben werden.

- Einzelne Note an, Geschwindigkeits Byte folgt
- Einzelne Note an, Geschwindigkeits Byte folgt nicht
- Einzelne Note ab
- Einzelne Note an, direkt nach Reset
- Einzelne Note an, selbe Note nochmals an
- Einzelne Note an, wenn in *polyphonie mode*
- Einzelne Note an, nach ungültigem status byte
- Einzelne Note an, andere Note an, erste Note ab
- Einzelne Note an, diverse andere Noten setzen, erst bei nächster Zeile erste Note ab

Zu jedem Testfall wird auf der nächsten Zeile das zu erwartende Resultat vorgegeben. Die *testbench* prüft die ausgegebene Notenwerte am Ausgang des midi interfaces mit den vorgegebenen Werten.

Beispielzeile

singl	55	90	27	80	27	90	05	00	00
check	00	00	27	00	00	00	05	00	00

Die Sequenz bedeutet Note 27 an (0x90), dann ab (0x80) der Note 27 und am Schluss an Note 05. Überprüft (check) wird, ob am Ausgang die Noten 27 und 05 anliegen.

Im *single mode* ist die Geschwindigkeit für das An- oder Abstellen der Note nicht relevant und wird deshalb nicht als Befehl eingelesen. Die *testbench* hängt nach jeder Note einen Dummy-Geschwindigkeitswert von 0x55 an.

5.3.2. Polyphonie testen

Testfälle

In der Polyphony können mehrere Noten hintereinander an- und nur einzelne davon wieder abgestellt werden.

- Polyphoniestatus setzen, einzelne Note an
- Polyphoniestatus setzen, mehrere Noten an
- Polyphoniestatus setzen, mehrere Noten über mehrere Zeilen verteilt an
- Polyphoniestatus setzen, Note an, die bereits in Register ist
- Polyphoniestatus setzen, Note an, andere Note an, erste Note aus, dritte Note an
- Polyphoniestatus setzen, dritte Note aus, erste Note an, erste Note an
- Polyphoniestatus setzen, *single note an* status setzen, Note ohne Geschwindigkeit senden
- Polyphoniestatus setzen, falsches Statusbyte senden, Note an, Note aus,
- Polyphoniestatus setzen, Reset, Note setzen
- Polyphoniestatus setzen, 10 Noten an
- Polyphoniestatus setzen, 10 Noten in Register, eine ist aus. Neue Note an senden

Beispielzeile

polyp	71	55	02	55	33	55	08	00	00
check	71	00	02	00	33	00	00	00	03

In der Sequenz wird die Note 71, dann die Noten 02 und 33. Danach wird die Note 08 abgestellt. Die *testbench* prüft am Ausgang, ob die Noten 71, 02 und 33 an sind.

Im Verarbeitungsmodus Polyphonie sendet die *testbench* das *status byte* "10100000" (0xA0)

5.4. Code Testbench

Die automatisierte Datenverarbeitung erzeugt viele Werte (10 Noten mit je 9 Werten). Um einzelne Bits effizient zu setzen oder zu überprüfen, wird der Code einem *refactoring* unterzogen.

Im Gegensatz zum hardwarenahen Code der VHDL-Blocks, bei denen arrays und loop explizit vermieden wurden, baute die *testbench* bewusst auf softwarenahe Strukturen auf.

5.4.1. Package

Damit in allen VHDL-Dateien die Arrays und Konstanten gebraucht werden können, werden diese Definitionen in einem Package zusammengefasst. Das Package kann wie eine Library zu Beginn einer VHDL-Datei eingebaut werden.

- Werte der *status bytes* als Konstanten
- Ein- und Ausgänge als arrays
- Tokenstruktur als record

Bsp. Tokenstruktur

```
-- define midi_data
type t_midi_data is record
    token_note : std_logic_vector(7 downto 0);
    token_attribut : std_logic_vector(7 downto 0);
end record;

type t_midi_data_array is array (0 to 3) of t_midi_data;

-- define token structure
type t_token_line is record
    token_cmd : string(1 to 5);
    t_midi_data : t_midi_data_array;
    token_number : std_logic_vector(7 downto 0);
end record;

-- array with note structure (input/output)
type t_note_array is array (0 to 9) of std_logic_vector(8 downto 0);
```

5.4.2. Prozess-Optimierung

Um die einzelnen Bits in den arrays zu setzen, braucht es in der Ablaufstruktur Optimierungen.

- loops iterieren durch die arrays
- Einleseprozess wird vom Verarbeitungsprozess getrennt
- Flags wie `s_read_input_finished <= '1'` sichern das parallele Datenverarbeiten

5.5. Ergebnisse Simulation

Die Ausgabe der Signale in die Output-Datei bezieht sich auf den Zustand am Ausgang des DUT. Damit auch die beiden internen Blöcke midi control und polyphonie out korrekt funktionieren werden die Signale überprüft. Auch das Verhalten in den Blöcken entspricht den erwarteten Signalverläufen.

5.5.1. Block Midi Control

Gemäss der *fsm* durchläuft der *single mode* die Zustände `idle`, `note_s`, `velocity_s` und geht dann zurück in den `idle`-Zustand. Das Signal `s_note_on` wechselt nach einem *status byte* von (0x90) auf on und nach (0x80) auf ab.

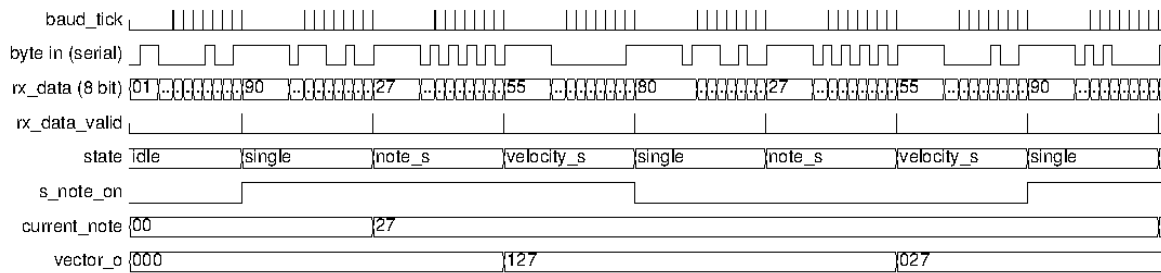


Abbildung 5.2.: Simulation Block Midi Control

Im *polyphony mode* existieren die Zustände *idle*, *note_v*, *velocity_v* und verbleibt in diesem Zustand. Nur durch ein *status byte* (oder ungültige *data bytes*) wird der Zustand der Polyphonie verlassen.

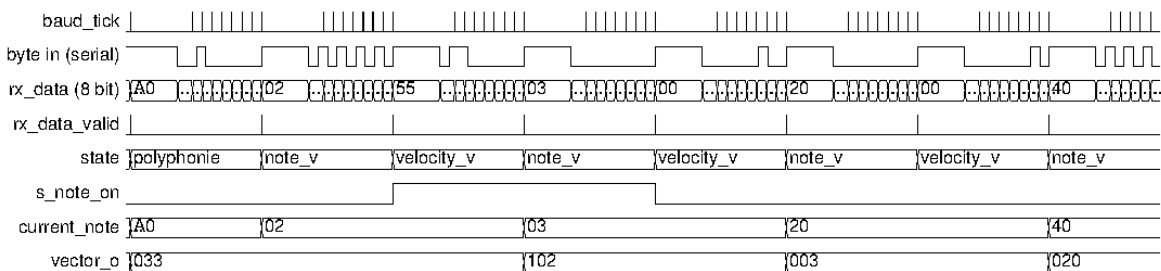


Abbildung 5.3.: Simulation Block Midi Control

5.5.2. Block Polyphony Out

Kriterien in der Polyphonie out sind, dass jede neue Note auf den nächsten freien Register-Platz gelegt wird. Zudem soll keine Note zwei Registerplätze belegen. Zudem soll, wenn alle Register-Plätze einen Notenwert haben, die neue Note an einen Registerplatz mit aktuell abgeschalteter Note besetzen. Alle Kriterien (siehe ??) sind erfüllt.

tb_note_0	127	027	
tb_note_1	000	105	
tb_note_2	000	173	073
tb_note_3	000	116	
tb_note_4	000	171	071
tb_note_5	000	102	
tb_note_6	000	133	033
tb_note_7	000	008	
tb_note_8	000	120	020
tb_note_9	000	003	

Abbildung 5.4.: Simulation Block Polyphonie Out

6. Diskussion und Ausblick

Das Provozieren von *glitch* und von *metastability* ist vollständig umgesetzt. Für das entwickelte *midi interface* fehlt die Integration des Blocks ins Synthesizer-Projekt.

6.1. Einbauen in das bestehende Synthesizer-Projekt

Ein erster Versuch, die 10 Noten über 10 *DDS* auszugeben scheiterte aus zeitlichen Gründen. Das Anschliessen der 10 DDS ist gemacht, ebenso eine Testbench über das Top des Synthesizers. Die Umsetzung scheiterte an der Implementation des Mischers, der die 10 parallelen Noten zusammenfügt.

Im Anhang befindet sich ein Block-Schaltbild, wie das *midi interface* in das bestehende Projekt eingebaut wird.

6.2. Frequenzmodulation mit vielfältiger Klangfarbe

Artikel zu empfehlen [?], [?]

7. Verzeichnis

7.1. Literatur

- [1] Altera. *Cyclone IV Device Handbook*, pages 8–19. San Jose, 2014.
- [2] The MIDI Manufacturers Association. *MIDI 1.0 Detailed Specification*. Los Angeles, 1995.
- [3] Kent Beck. *Test-Driven Development, By Example*, page ix. Addison Wesley Signature, 2013.
- [4] Christian Braut. *Das MIDI-Buch*, page 10. Sybex, 1993.
- [5] John A. Camara. *Engineering Reference Manual*, pages 32–2. Belmont, 2010. About metastability.
- [6] William I. Fletcher. *An Engineering Approach to Digital Design*, page 472. Utah State University, 1980. About glitch.
- [7] William I. Fletcher. *An Engineering Approach to Digital Design*, page 482. Utah State University, 1980. About metastability.
- [8] Hans-Joachim Gelke. *Polyphonie Erweiterung*. Als Datei übergeben, 2015.
- [9] Sandeep Mandarapu. *Measuring Metastability, Master Project*. Departement of Electrical and Computer Engineering, Southern Illinois University, 2012.
- [10] Dictionary of the English Language. *finite state machine*. American Heritage, 2011.
- [11] Cambridge Dictionaries Online. *glitch, specialized electronics*. www.dictionary.cambridge.org/dictionary/english/glitch, 02.11.2015.

7.2. Glossar

Das Glossar dient interessierten Software-Entwicklern, die elektrotechnik-spezifischen Worte zu verstehen.

Asynchrone Signale

Werden Signale unmittelbar zugewiesen sind sie vorerst ungetaktet, asynchron. Es ist nicht definiert, *wann* exakt das Signal den neuen Wert erhält. Erst wenn ein Signal durch ein Flip-Flop geführt wird, wird es getaktet und seine Signalzuweisung dadurch determinierbar.

Audio Codec

Bezeichnet im vorgegebenen Synthesizer-Projekt den, bezüglich dem FPGA, externen Audio-Baustein auf dem altera Development Board DE2-115. Es handelt sich um einen WM8731.

Clock Domain

Ein Bereich der Hardware, der mit demselben Takt läuft.

Controller

Bezeichnet ein Bauteil, das Eingangssignale gemäss einer Spezifikation verarbeitet und die entsprechenden Ausgangssignale setzt.

DDS

Bedeutet Direct Digital Synthesis und bezeichnet das digitale Erzeugen von periodischen Signalen.

Diese Signale können für die Tonerzeugung gebraucht werden.

Dekoder

Bezeichnet ein Bauteil, das einen oder mehrere Eingangswert(e) gemäss implementierter Logik in einen Ausgangswert wandelt.

Durchlaufverzögerung

Wird englisch *propagation delay* genannt und bezeichnet die Zeit, die Daten vom Eingang bis zum Ausgang des Bauteils brauchen.

Die Durchlaufverzögerung beträgt beim Cyclone IV 4 ns [1].

Finite State Machine (fsm)

"A model of a computational system, consisting of a set of states, a set of possible inputs, and a rule to map each state to another state, or to itself, for any of the possible inputs." [10]

Auf deutsch "Ein Model in Rechensystemen, das aus einem Satz aus Zuständen, möglichen Eingängen und Regeln wie man von einem Zustand zum nächsten, oder zu sich selbst, für alle möglichen Eingänge gelangt. "

Flip-Flops

Grundbaustein der Digitalen Logik. Das Flip-Flop speichert seinen Wert, den es am Eingang erhält am Ausgang.

Glitch

Im technischen Bereich bedeutet *glitch* gemäss Cambridge Dictionary "a sudden unexpected increase in electrical power, especially one that causes a fault in an electronic system" [11],

auf deutsch "eine plötzliche, unerwartete Spannungserhöhung, die insbesondere ein Fehlverhalten im elektronischen System verursacht".

Hold Time

Ist die minimale Zeit, in der die Inputdaten *nach* der Taktflanke stabil sein müssen.

Die *hold time* beträgt beim Cyclone IV E 0 ns [1].

Hot Plug

Bezieht sich auf die Hardware-Umsetzung einer *finite state machine*. Gewöhnlich braucht es für 2^n Zustände n Flip-Flops. Bei Hot Plug braucht es für n Zustände n Flip-Flops, denn jeder neue Zustand wird durch eine '1' am n -ten Flip-Flop detektiert. Alle anderen Flip-Flop-Werte sind auf '0'. Die logische Schaltung für eine *Hot Plug fsm* wird durch den direkten Bezug einer gesetzten '1' zum Zustand einfach.

Kathodenstrahl Oszilloskop, KO

Bezeichnet ein elektronisches Messgerät, das ein Signale analog als Spannungen mit deren zeitlichem Verlauf am Bildschirm ausgibt.

Metastabilität

Bezeichnet in der digitalen Signalverarbeitung einen unsicheren Zustand. Der Wert des Ausgangssignals ist nicht vorhersehbar, da beim Eingangssignal die Daten zu spät ankommen oder zu früh weggenommen werden.

Others

Bezeichnet in einem Switch-Case in VHDL alle anderen Möglichkeiten, die nicht abgefragt werden. Es dient dem System einen definierten Zustand zu geben, falls etwas Unerwartetes eintrifft.

Pfadzeit

Bezeichnet die Zeit, die ein Signal von einem Flip-Flop zum nächsten braucht.

Refactoring

Bezeichnet das Überarbeiten eines funktionierenden Codes. Ziele sind, den Code effizienter, verständlicher und sicherer zu gestalten.

Setup Time

Minimale Zeit, in der Inputdaten stabil sein müssen *bevor* eine Taktflanke die Daten triggert.

Die *setup time* beträgt beim Cyclone IV E 10 ns [1]

State

Bezeichnet den aktuellen Zustand einer *finite state machine*.

Textbasierte Testbench

In VHDL wird die Simulation der Signale in einer Testbench aufgesetzt. In der Testbench werden die Signalanregungen, stimuli, definiert, und die zeitlichen Abläufe unter Signalen. Für eine Testbench ist eine eigene Software notwendig.

Eine textbasierte Testbench liest die stimuli und die zu erwartenden Ergebnisse aus einer Datei ein. Die Ergebnisse werden ebenfalls in eine Datei ausgegeben.

Token

Bezeichnen Elemente in einer Reihe von strukturierten Daten.

Quartus

IDE von altera zum Kompilieren, Synthetisieren und Einbauen von IPs für die altera FPGAs.

A. Offizielle Aufgabenstellung

Beschreibung der Projektarbeit PA15_gelk_1

In dieser Projektarbeit sollen Versuche entwickelt werden, die für das Modul DTP2 verwendet werden können. Die Arbeit besteht aus zwei Teilen:

Im ersten Teil der Arbeit sollen Versuche entwickelt werden, mit denen folgende Timing Artefakte demonstriert werden können. Dies soll zum zu einem vertieften Verständnis der digitalen Design Grundlagen führen.

- Erzeugung von Glitches mit einem Zähler und nachgeschaltetem Dekoder. Sichtbarmachung der Glitches mit einem Oszilloskop. Betätigen des asynchronen Resets vom Decoder aus.
- Provozieren und sichtbarmachung von Metastabilen Zuständen. Hierfür kann z.B. eine Schaltung mit zwei asynchronen externen Takten aufgebaut werden.

Im zweiten Teil soll mit dem dem Direct Digital Synthesis Verfahren ein Synthesizer mit vielfältigen Klangfarben entwickelt werden. Damit kann anspruchsvolle digitale Schaltungstechnik umgesetzt werden. Zum Erreichen der Klangvielfalt können mehrere DDS Generatoren gleichzeitig, mit unterschiedlichen Frequenzen und Phasen betrieben werden. Möglich ist auch eine Frequenzmodulation mit einem zweiten Generator oder Ändern des Volumens mit einer Hüllkurve. Die Ansteuerung soll mit Hilfe eines MIDI Interfaces, welches Polyphonie (mehrere Klaviertasten gleichzeitig gedrückt) unterstützt. Die Implementierung soll im FPGA erfolgen. In der Implementierungsphase der Arbeit soll das Timing der FPGA Implementierung genau betrachtet werden.

Am Ende soll eine Referenzimplementierung in Anlehnung an den Yamaha DX7 für das Modul DTP2 entstehen

B. Aufgabenspezifikation für den zweiten Teil

- Midi Interface for Keyboard für Polyphonie nach Konzept von gelk
 - o 10 Frequenz Control Ausgänge zur Steuerung der Tonhöhe des Generators
 - o 10 On/Off Ausgänge Ton on/off
 - o UART wird geliefert von gelk
 - o VHDL wird von Grund auf neu erstellt.
- 10 DDS implementieren und mit Mischer Mischen
- Script basierte Testbench. Testbench erzeugt serielle Midi Daten, so wie sie auf dem DIN Stecker vorkommen (logisch)
- Testbench liest eine Testscript Datei ein, in welcher die Tastendrucke eines Keyboards abgebildet werden können. Midi Poliphony Spec muss durch die Testbench unterstützt werden können. Velocity muss nicht unterstützt werden.
- FM Modulation – Testbench im Matlab
- Kein VHDL code ohne Testbench.
- Block level testbench. Unit Tests.

Abgrenzung:

- Keine Hüllkurve
- Keine Ausgabe der Velocity auf Midi controller
- Kein Bluetooth

Zeitplan:

- 2.5 Wochen Midi Controller incl. 10 DDS
- 2.5 Wochen FM Synthese

Unterstützung:

- Midi Controller/gelk
- FM-Synthese/rosn

Falls Midi nicht zum geplanten Zeitpunkt fertig wird, wird FM-zurückgestellt. Alle oben genannten Punkte sind Pflicht. Nicht Fertigstellung hat Einfluss auf die Benotung.

C. CD mit Projektdateien

D. In- und Output-Datei textbasierte Testbench

Datei mit Testbefehlen für die Testbench

```
reset 00 00 00 00 00 00 00 00 00
check 00 00 00 00 00 00 00 00 00
singl 55 90 27 80 27 90 05 00 00
check 00 00 27 00 00 00 05 00 00
singl 55 90 73 80 73 90 16 00 00
check 00 00 00 00 00 00 16 00 00
polyp 71 55 02 55 33 55 08 00 00
check 71 02 33 00 00 00 00 00 03
polyp 20 55 03 00 20 00 40 55 00
check 20 00 16 00 40 00 00 00 03
polyp 71 00 16 55 20 55 33 00 00
check 00 00 16 00 20 00 00 00 04
```

Das Testergebnis in der Datei

Automatically generated outputfile

Read file with commands in

```
reset
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note number: 00
```

```
check
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
Read note:00
Read attribut: 00
```

Read note number: 00

singl

Read note:55

Read attribut: 90

Read note:27

Read attribut: 80

Read note:27

Read attribut: 90

Read note:05

Read attribut: 00

Read note number: 01

check

Read note:00

Read attribut: 00

Read note:27

Read attribut: 00

Read note:00

Read attribut: 00

Read note:05

Read attribut: 00

Read note number: 01

... etc

polyp

Read note:02

Read attribut: 55

Read note:03

Read attribut: 00

Read note:20

Read attribut: 00

Read note:40

Read attribut: 55

Read note number: 03

check

Read note:02

Read attribut: 00

Read note:16

Read attribut: 00

Read note:40

Read attribut: 00

Read note:00

Read attribut: 00

Read note number: 03

Number of read lines from file: 12

Finished read whole file

E. Blockschtlbild Top-Level Synthesizer-Projekt

In die bestehenden Blöcke und Signale wird das MIDI Interface wie folgt eingebaut:

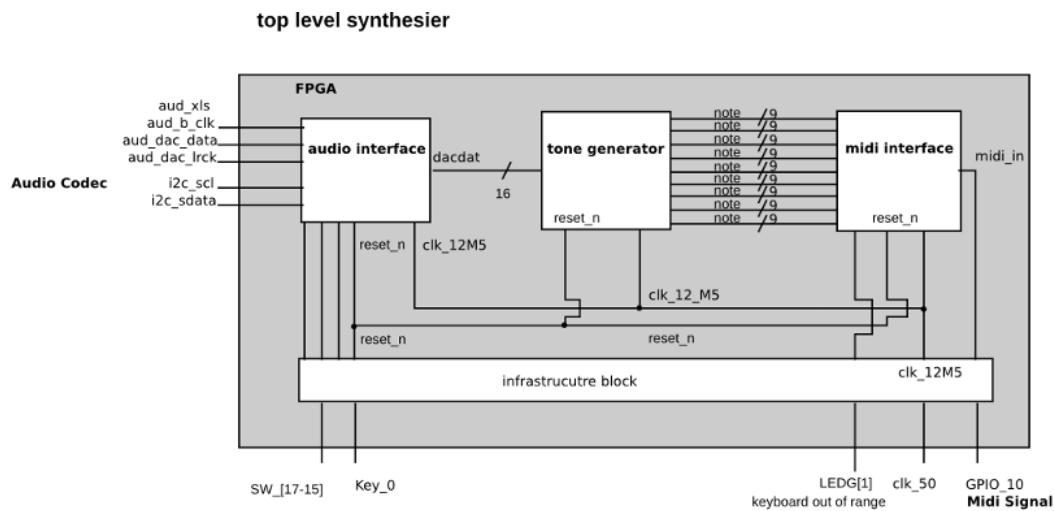


Abbildung E.1.: Top Synthesizer mit MIDI Interface: Blockschaltbild

Hier ist das Konzept der Umsetzung des MIDI Interface detaillierter beschrieben:

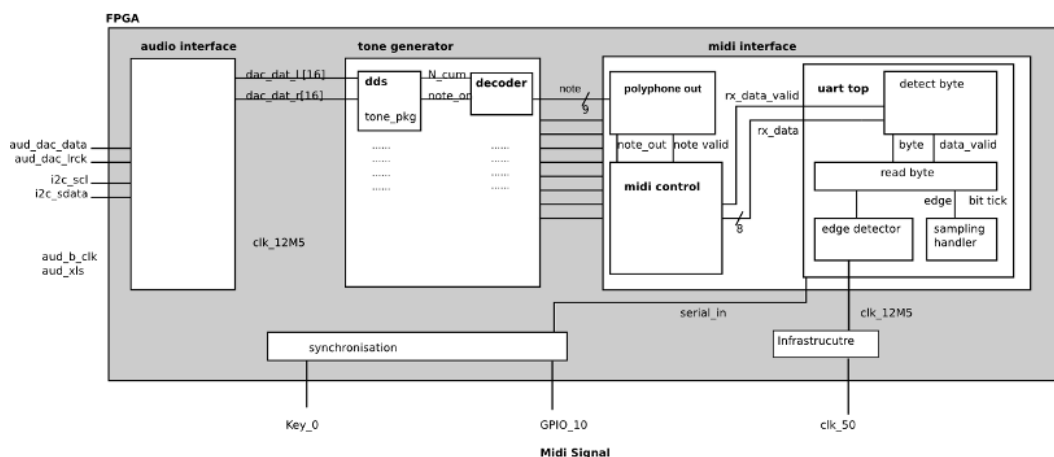


Abbildung E.2.: Top Synthesizer mit MIDI Interface: Detailansicht