

PA15_gelk_1 Polyphonic DDS Synthesizer mit MIDI Steuerung

ZÜRCHER HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN

INSTITUTE OF EMBEDDED SYSTEMS

Autoren Katrin Bächli

Hauptbetreuer

Nebenbetreuer

Datum 6. 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>

Inhaltsverzeichnis

1. Einleitung	4
1.1. Ausgangslage	4
1.2. Zielsetzung Aufgabenstellung Anforderungen	4
2. glitches	6
2.1. Definition Glitches	6
2.2. Ursache für Glitches	6
2.2.1. Asynchroner Input	6
2.2.2. Nachteil getakteter Prozesse	7
2.3. Glitches erzeugen	7
2.3.1. Glitches Aufgrund von Bauteiltoleranzen	7
2.3.2. Glitches Aufgrund von Pfadverzögerung	8
2.4. Resultat Glitches provozieren	9
2.4.1. Erzeugen über Bauteiltoleranzen	9
2.4.2. Erzeugen über Routing	10
3. Metastabilität	11
3.1. Definition Metastabilität	11
3.2. Ursache von Metastabilität	11
3.3. Metastabilität erzeugen	12
3.3.1. Ansatz	12
3.3.2. Implementation	13
3.4. Resultat Metastabilität provozieren	13
4. MIDI Steuerung	14
4.1. Einteilen der Blöcke und definieren der Schnittstellen	14
4.2. Das MIDI Kommunikationsprotokoll	15
4.2.1. <i>status bytes</i>	15
4.2.2. <i>data bytes</i>	15
4.2.3. Ungültige Bytes	15
4.3. Umsetzung "midi controlBlock"	15
4.4. Umsetzung "polyphonie outBlock"	18
5. Polyphonie	19
5.1. Midi Spezifikation	19
5.2. Umsetzung	19
5.2.1. software nahe	19
6. testbench	20
6.1. unit test midi control	20
6.2. unit test midi interface	20
7. Resultate der Projektarbeit	21
7.1. Generieren von Glitches	21
7.2. Zustand von Metastabilität provozieren	21
7.3. MIDI Controller entwickeln	21
7.4. Polyphonie Block	21
7.5. DDS Generatoren basierend auf Frequenzmodulation entwickeln	21
7.6. Textbasierte Testbench für alle entwickelten Blocks	21
8. Diskussion und Ausblick	22
8.1. Literaturverzeichnis	23

.1. Offizielle Aufgabenstellung	I
.2. CD mit vollständigem Bericht	I
A. Anhang: Top Synthesizer	II

Liste der noch zu erledigenden Punkte

besseres Bild def Glitch	6
Bild anpassen	8
Bild Asynchronem Zähler besser beschreiben	8
Bild FF1 verzögert, FF2 schneller	8
korrektes Wort ?(Concourent Assignment)	9
Timeanalyse für FF	10
einsetzen setup zeit gemäss glossar für	13

1. Einleitung

Die Projektarbeit bietet die Möglichkeit, sich vertieft in VHDL einzuarbeiten. Mit vertieft ist das Erstellen eines eignen Projektes wie auch das Kennenlernen der Eigenheiten der Sprache VHDL gemeint.

Der erste Teil der Projektarbeit umfasst die Auseinandersetzung mit der Sprache VHDL und deren eigenen Herausforderungen. Die zwei Fehlerquellen *Glitches* und *Metastabilität* werden künstlich erzeugt, damit ihre Ursache verstanden wird.

Der zweiten Teil der Projektarbeit beinhaltet eine Weiterentwicklung des Synthesizer-Projektes aus der Vorlesung Digitale Technik II. Konkret soll die Frequenzmodulation vertieft und der Midianschluss implementiert werden.

Eine Projektarbeit verdient nicht ihren Namen, wenn am Schluss der Arbeit nicht ausführlich die Resultate diskutiert und die verwendete Literatur genannt wird. Es folgt deshalb ein ausführlicher Anhang, in dem unter anderem auch der verwendete VHDL-Code abgebildet ist.

—ALT

1.1. Ausgangslage

Nennt bestehende Arbeiten zu diesem Thema (Literaturrecherche)
Stand der Technik: Bisherige Lösungen des Problems und deren Grenzen

MIDI bedeutet *musical instrument digital interface* und ist ein Standard, der sowohl die genaue Beschaffenheit der erforderlichen Hardware wie auch das Kommunikationsprotokoll der zu übermittelnden Daten festlegt (**Christian Braut: Das MIDI-Buch. Sybex, Düsseldorf u. a. 1993,**).
The MIDI Manufacturers Association, MIDI 1.0 Detailed Specification, Document Version 4.2, Revised September 1995, Los Angeles (1995)

Im Modul DTP2 entwickeln Studentinnen und Studenten ihren eigenen Synthesizer. Eine Option in diesem Projekt ist es, den Synthesizer über ein Keyboard per MIDI steuern zu können.

Am Institut for Embedded Systems bestand bereits die MIDI UART und die Aufgabe im zweiten Teil dieser Projektarbeit ist es, eine MIDI Steuerung zu entwickeln, die für das Modul DTP2 verwendet werden kann. Bild Top im Anhang

Am Institut for Embedded System bestand bereits die Hardware wie auch die MIDI UART. Diese detektiert die empfangenen Bytes und sendet ein valid-Flag, wenn das Byte korrekt ist. Das Byte wird als logic Vector übermittelt.

Jeder zu entwickelnde Block wird mit einer textbasierten *testbench* getestet.

1.2. Zielsetzung Aufgabenstellung Anforderungen

Formulierte Ziele der Arbeit
Verweist auf die **offizielle Aufgabenstellung des/der Dozenten im Anhang**

EV.Pflichtenheft, Spezifikation

EV. übersicht über die Arbeit: stellt die folgenden Teile kurz vor.

2. glitches

2.1. Definition Glitches

Im technischen Bereich bedeutet gemäss Cambridge Dictionaire ein *glitch*, eine ungewollte, flüchtige Signalspitze, die ein Fehlverhalten im System verursacht. Im Anhang ?? befindet sich der Originaltext wie auch noch eine weitere Defintion aus dem englischen Sprachraum.

In der digitalen Signalverarbeitung ist das Glitch ein bekannter Begriff und wird dort unter anderem leicht sarkastisch beschrieben:

"Als "Glitch" wird eine ungewollte, flüchtige "Signalspitze" bezeichnet, die Zähler aufwärts zählt, Register löscht oder einen ungewollten Prozess startet." key [?](Fletcher, Digital design, 472)

Am intuitivsten ist die bildliche Darstellung des soeben beschriebenen Fehlverhaltens (Abbildung 2.1). In dieser Signalabfolge treten zwei mal Glitches auf, die eigentlich nicht dort hingehören.



Abbildung 2.1.: Glitch-Signalspitzen

besseres Bild
def Glitch

Auf den ersten Blick scheinen solch temporäre Spannungsspitzen nicht zu stören. Doch wenn man Pech hat, sind Glitches der Auslöser für Abstürze oder zumindest für ein Fehlverhalten eines Gerätes. Aus diesem Grund, wird nun der Ursache dieser Spitzen nachgegangen.

2.2. Ursache für Glitches

Der Auslöser der flüchtigen Spannungsspitzen sind asynchrone Inputs vor einem asynchronen Bauteil oder verzögerte Signale. Trifft z. B. vor einem Dekoder von vier Leitungen, das Signal einer Leitung zu spät an, entschlüsselt der Dekoder kurzfristig einen falschen Wert. Obwohl die Störung nur kurz ist, übermittelt ein asynchroner den falschen Wert direkt an seinen Ausgang.

2.2.1. Asynchroner Input

Das ungleichzeitige Eintreffen von Signalen kann z.B. durch lange Signalpfade (Leitungen), unterschiedliche Durchlaufverzögerungen der vorangehenden Flip-Flops oder unterschiedliche Logik-Zeiten entstehen. Grundsätzlich gelten alle nicht-getakteten Prozesse als potenzielles Risiko für Glitches, da man bei ungetakteten Prozessen nicht weiss, wie lange sie dauern.

2.2.2. Nachteil getakteter Prozesse

Jeder getaktete Prozess verzögert die Verarbeitung. Aus diesem Grund wird abgewogen, wo Prozesse getaktet und wo sie asynchron getätigt werden. In VHDL gibt es viele asynchrone Vorgänge (wie ungetaktete Prozesse oder Singalzuweisungen), deshalb ist es vorteilhaft, wenn das Risiko asynchroner Prozesse bekannt ist.

Abbildung 2.2 zeigt ein leicht verzögertes (getaktetes) enable-Signal zu einem anders verzögerten (getakteten) Flip-Flop-Eingangssignal Q. Der Ausgang des Flip-Flops weist kurzzeitig Glitches auf.

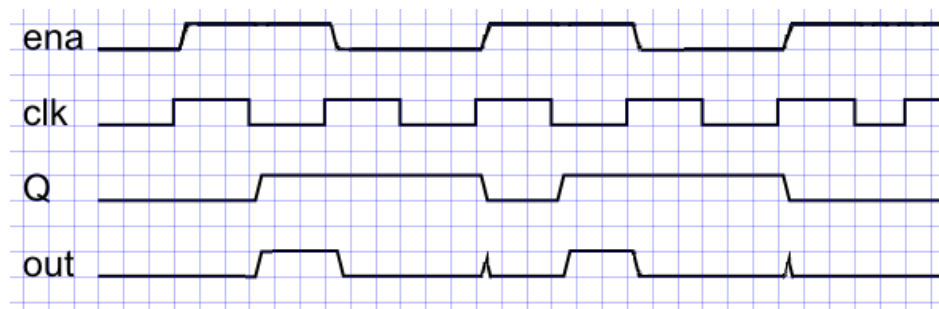


Abbildung 2.2.: Mögliches Bsp. für Glitches

2.3. Glitches erzeugen

Was in Glitch ist, ist relativ einfach zu beschreiben. Ein Glitch jedoch mit moderner Digitaltechnik zu erzeugen, erweist sich als etwas schwerer. Hier zwei Ansätze, die getestet werden.

2.3.1. Glitches Aufgrund von Bauteiltoleranzen

Der erste Ansatz ist, ein Zähler aus vier Flipflops mit asynchronem Dekoder zu implementieren.

Konzept

Die Erwartung ist, dass aufgrund der *Bauteiltoleranzen* der Flip-Flops die vier Ausgänge an den Flip-Flops nicht gleichzeitig ihren Wert übermitteln. Die einen sind leicht schneller, die anderen leicht verzögert. Dadurch ergibt sich kurzzeitig am asynchronen Dekoder einen falschen Wert.

Damit ein abnormaler Wert in einem Zähler erkannt wird, sendet der Dekoder bei der Zahl 7 einen Peak. Ohne Glitch entschlüsselt der Dekoder in regelmässigen Abständen von 160 ns diese Zahl. Aufgrund der Flip-Flop-Bauteiltoleranzen ist ein kurzzeitiges Dekodieren einer 7 *ausserhalb der Periode T* zu erwarten.

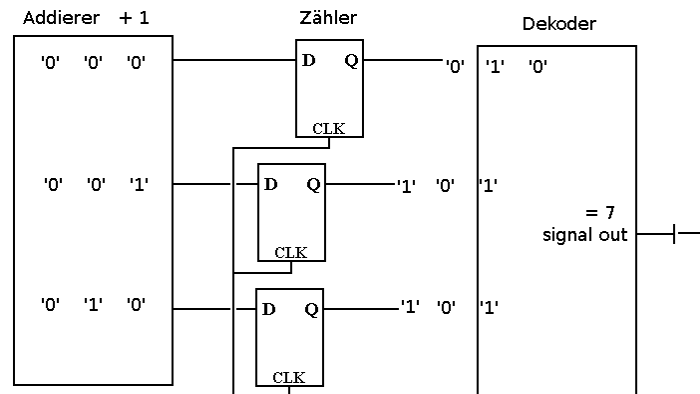


Abbildung 2.3.: Ausnutzen der Bauteilverzögerung

Implementation

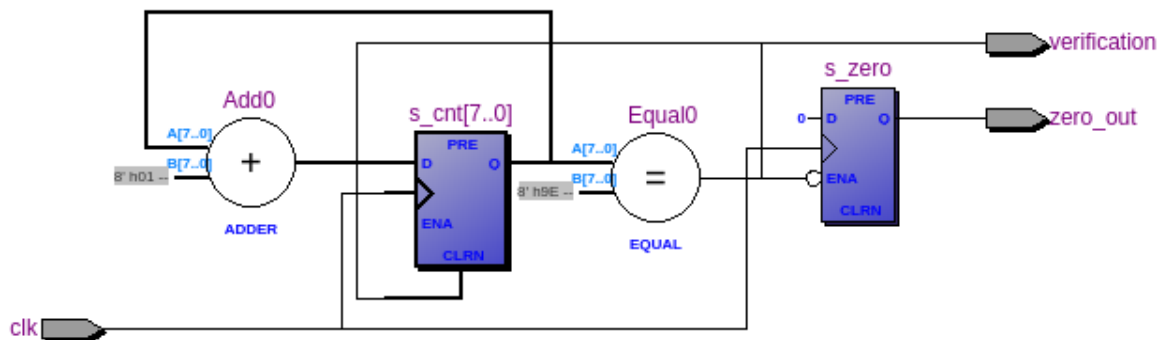


Abbildung 2.4.: RTL Zähler mit asynchronem Dekoder

Um die gewünschten Zahlenwerte von den Glitches zu unterscheiden, wird das asynchrone Signal getaktet. Dadurch erscheint der korrekte Zählwert mit einer Taktverzögerung. Die Periode ist 20 ns (CLK = 50 MHz).

2.3.2. Glitches Aufgrund von Pfadverzögerung

Der zweite Ansatz ist die gesuchte Bauteilverzögerungen über längere Signalfade zu simulieren. Der Dekoder des Zählers bleibt asynchron.

Konzept

Dekodiert wird die Zahl 15. Durch intelligentes Routing (FF 1 wird verzögert, FF 2 wird beschleunigt) wird der Zustand der Zahl 11 forciert

Bild anpassen

Bild Asynchronem Zähler besser beschreiben

Bild FF1 verzögert, FF2 schneller

Implementation

Cyclone II, Board De2. Quartus 13.0sp.

Die *Pfadverlängerung* wird über das Routing über die GPIO-Pins des Headers 1 gemacht (siehe Abbildung 2.6). Die obersten vier Doppel-Pins erhalten eine "Brücke", sodass das Signal links ausgegeben und rechts wieder eingespiessen wird.

Signalverkürzung ist eine direkte Signalzuweisung .

korrektes We
?(Concouren
Assignment)

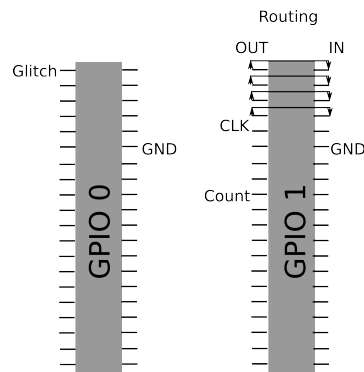


Abbildung 2.5.: GPIO Anschlüsse

Auf dem KO wird das asynchrone Glitch-Signal und das synchrone Zählersignal neben dem Takt ausgegeben. Weil der Zähler synchronisiert wurde, ist der Wert 1 Periode (= 20 ns) später als der Glitch.

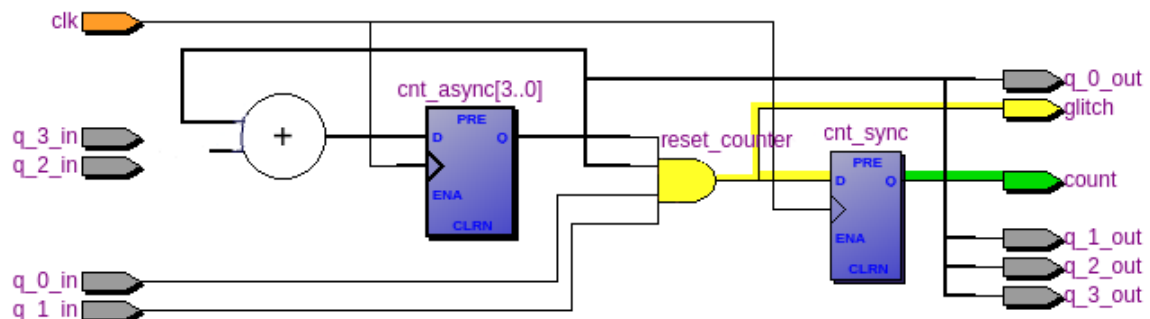


Abbildung 2.6.: Zähler mit Signal-Routing über GPIO

Im RTL-Diagramm sieht man deutlich den Unterschied zwischen dem asynchronen Zähler, der über das Gate *reset_counter* beim Wert 15 einen Impuls an den Ausgang *glitch* gibt und dem synchronisierten Zähler *cnt_sync* der dem asynchronen Ausgang nachgeschaltet ist und dieses Signal taktet. Das getaktete Zähl-Signal geht an den Ausgang *count*.

2.4. Resultat Glitches provozieren

2.4.1. Erzeugen über Bauteiltoleranzen

Der Ansatz, dass die Bauteiltoleranzen der Flip-flops eine Ursache für asynchrone Inputs in den Dekoder sind ist korrekt. Die Umsetzung zeigte sich jedoch als schwierig, da die heutigen Flip-Flops zu

schnell sind bzw. ihre Toleranzen zu klein um sichtbar zu werden. Aus diesem Grund entschlüsselte der asynchrone Dekoder trotz kleinen Verzögerungen die Werte stets korrekt.

Timeanalyse
für FF

2.4.2. Erzeugen über Routing

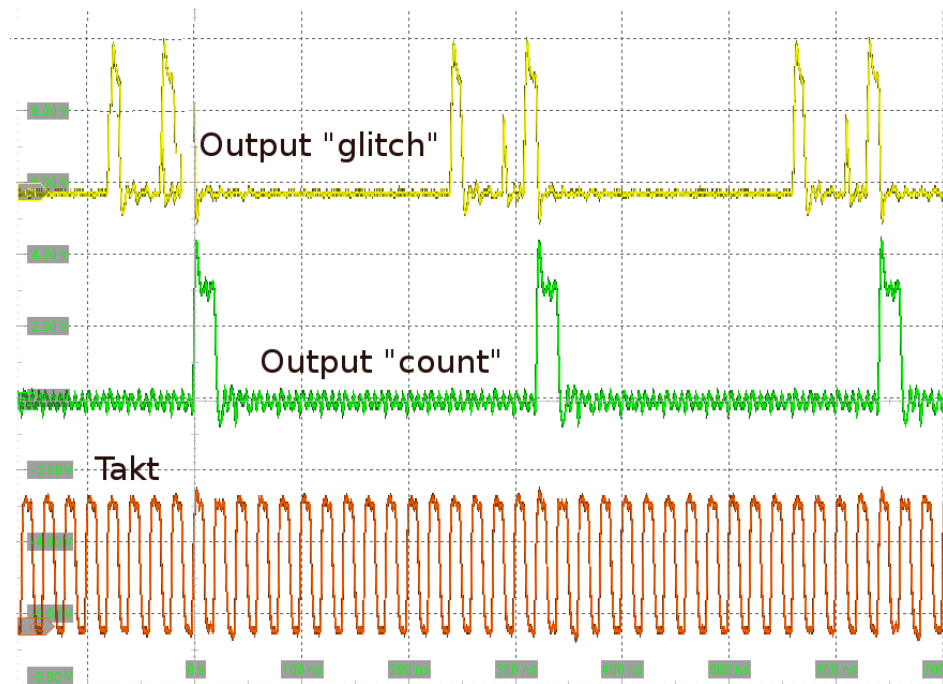


Abbildung 2.7.: Glitch (gelb), Zähler (grün) und Takt (orange)

Typisch ist, dass der synchrone Zähler eine Signalbreite von genau einer Periode hat, da dieses Signal getaktet ist. Dagegen hat der asynchrone Glitch keine konstante Breite.

1. Bei welchen Zählständen treten Glitches auf?
2. Wie hängen die Zählstände mit dem gewählten Routing zusammen?

3. Metastabilität

3.1. Definition Metastabilität

Metastabilität bedeutet, dass der Ausgang eines Flip-Flops nicht dem Eingang entsprechen muss. Wechselt das Inputsignal eines Flip-Flops zur falschen Zeit, ist der Wert des Ausgangssignal unsicher. Hier zwei kurze englische Beschreibungen, dieses Phänomens:

” 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” (Camara, S. 32-2)

”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” (Fletcher, 482.)

Im besten Fall wählt der Ausgang bei unklarem Eingangssignal selbst einen Wert an ('0' oder '1'). Im schlechten Fall “hängt” sich das Flip-Flop “auf” und toggelt permanent zwischen '0' und '1' oder setzt sogar beide Werte parallel.

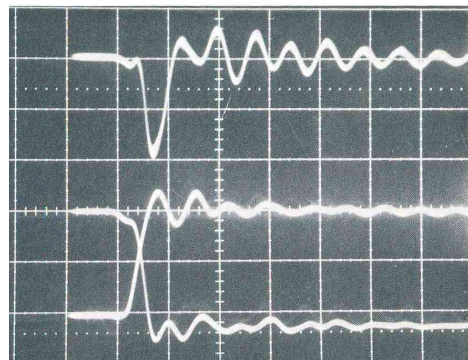


Abbildung 3.1.: Metastabilität schlimmster Fall (Fletcher, 482.)

3.2. Ursache von Metastabilität

Der Grund für Metastabilität ist, dass der angelgte Wert entweder zu spät eintrifft (verletzen der setup-Zeit) oder zu früh wieder verschwindet (verletzen der hold-Zeit). Metastabilität kann vermieden werden, wenn diese zwei Zeiten strikt eingehalten werden:

”Metastabilit 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.” (Camara, S. 32-2)

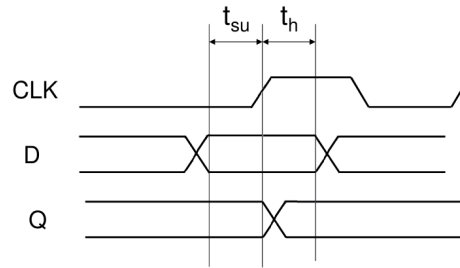


Abbildung 3.2.: Einhalten der Datenzeiten

Es gibt mehrere Gründe für das Nichteinhalten der geforderten setup-Zeit:

- Ein Logikpfad kann zu lange sein, bzw. die Taktfrequenz ist zu schnell
- Zwischen den Bauteilen liegen zu lange Pfade, die das Eintreffen der Daten verzögern
- Ein vorangehendes Bauteil hat eine zu lange Durchlaufverzögerung.

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

Als Alternative bietet sich eine Synchronisierungsschaltung an. Zwischen den zwei Takt-Flanken kann sich der metastabile Ausgang erholen und gelangt so stabil in den Verarbeitungspfad. Der Nachteil der Synchronisation ist jedoch, eine um einen Takt längere Verarbeitungszeit.

3.3. Metastabilität erzeugen

3.3.1. Ansatz

Aufgebaut wird ein System mit zwei Takten. Der zentrale Block hat eine Taktfrequenz von 50 MHz und beinhaltet eine State Machine. Diese wechselt bei jedem Impuls von einem Zustand in den anderen (Abbild: 3.4 Um die zwei Zustände zu erkennen, werden beiden Zuständen ein logischer Pegel zugefügt:

- Zustand 1: s0 = Logisch '0'
- Zustand 2: s1 = Logisch '1'

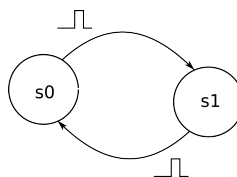


Abbildung 3.3.: Statemachine im zentralen Block

Der Impuls, der die State Machine steuert ist asynchron. Er wird von einem Zähler generiert, der mit der Taktfrequenz von 27 MHz läuft. Alle 37 ns sendet der Zähler einen Puls an die State Machine. Die State Machine selbst arbeitet mit einer Taktfrequenz von 20 ns. Der Impuls ist ihr gegenüber asynchron.

Erwartet wird, dass die setup-Zeit der State Machine-Flip-Flops regelmässig verletzt werden.



Abbildung 3.4.: Die zwei Taktzeiten

3.3.2. Implementation

3.4. Resultat Metastabilität provozieren

Was ist das Ergebnis beim Verletzen der setup Zeit? Beide Ausgänge immer an? Keiner von beiden? aufhängen des Systems? (Keine LED geht mehr).

Synchronisation Schaltung erhärtet die These b

einsetzen set
zeit gemäss
glossar für ..

4. MIDI Steuerung

4.1. Einteilen der Blöcke und definieren der Schnittstellen

Beschreiben der Schnittstellen

Vor Programmieren der units (blöcke) und unit test, schnittstellen klären

!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Die Abbildung A.1 (im Anhang A) zeigt, wie das zu entwickelnde MIDI Interface in die bestehenden Blöcke des Synthesizer-Projektes eingebaut wird. Die im Anhang direkt anschliessende Abbildung A.2 zeigt dann die geplante Umsetzung detaillierter.

Im folgenden wird nur auf den Block *midi interface* eingegangen, der die Umsetzung der MIDI Steuerung darstellt. Als erstes die Zusammenfassung der internen Blöcke (siehe Abbildung 4.1).

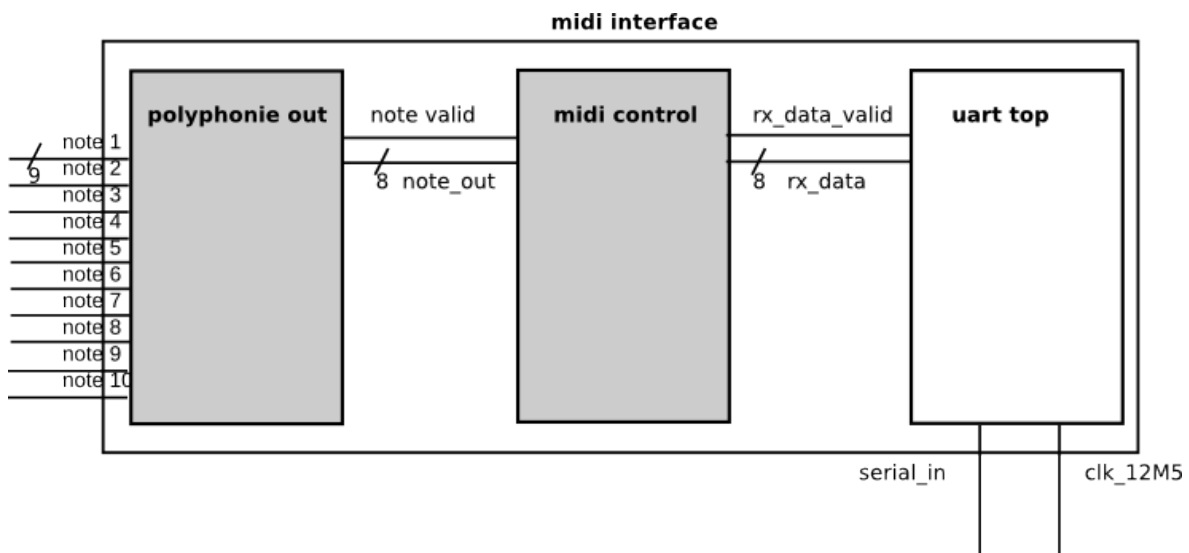


Abbildung 4.1.: Blockschaltbild MIDI Interface

Gegeben ist der UART Block, in der Abbildung als **uart top** bezeichnet. Dieser detektiert MIDI Bytes und sagt, ob diese gültig sind. Zu entwickeln sind die zwei Blöcke "**midi control**" und "**polyphonie out**".

Als erstes wird kurz generell auf das MIDI Kommunikationsprotokoll eingegangen, damit man die Kriterien bei der Umsetzung versteht. Danach wird die Umsetzung des *midi control*-Blocks erklärt und zum Schluss die Implementation des *polyphone out*-Blocks.

4.2. Das MIDI Kommunikationsprotokoll

Werden MIDI Daten übermittelt, so unterscheidet der Standard zwei Typen an Daten (Detaillierte Spezifikation S. 11 - 12).

4.2.1. status bytes

status bytes sind 8 Bit lang und das MSB ist immer logisch '1'. Die Status Bytes dienen dem Identifizieren der nachfolgenden *data bytes*. Das *status byte* sagt, von welcher Art und mit welcher Datenstruktur die folgenden *data bytes* sein werden.

MIDI behält einen Status so lange, bis ein neues *status byte* folgt. Dieses Verhalten wird als *running status* bezeichnet. Dieses Verhalten ist vor allem für die Polyphonie interessant, da dieser Zustand bleibt und viele *Data Bytes* (im Sinn von Noten) folgen können, ohne dass es eines neuen Status Bytes bedarf.

4.2.2. data bytes

Gemäss Spezifikation folgen einem *Status Byte* exakt 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 sind Noten, Geschwindigkeit des Anschlages und ...

Je nachdem welches *status byte* im Voraus gesetzt wurde, werden die Attribute anders interpretiert. Ist z.B. Polyphonie gesetzt, so bedeutet ein *data byte* mit einer Geschwindigkeit von 0, Note abstellen. Diese und andere Spezifikationen werden detailliert in **Detaillierte Spezifikation** beschrieben.

„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.“ (Spezifikation, S. 6).

4.2.3. Ungültige Bytes

„Alle *status bytes*, die nicht implementierte Funktionen enthalten und alle ihnen folgenden *Data Bytes* sollen vom Empfänger verworfen werden.“ (**Spezifikation, 6**).

MIDI Geräte sollen ausdrücklich beim Ein- und Abstellen darauf bedacht sein, dass keine undefinierten Bytes gesendet werden (**ebd**).

Diese Anforderung ist wichtig beim Implementieren einer Finite State Machine und der Testbench (siehe Kapitel).

4.3. Umsetzung "midi controlBlock

Ausgehend von der Spezifikation sind drei Eckpunkte für die *finite state machine* zu berücksichtigen:

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*

Zu beachten in der Verarbeitung der Daten ist, dass im *single mode* zuerst gesagt wird, ob eine Note an oder ab und diese Reihenfolge im *polyphony mode* gerade umgekehrt ist: zuerst kommt die Note, dann die Angabe, ob sie an oder ab ist.

Aus diesen Anforderungen ergab sich folgende *finite state machine*:

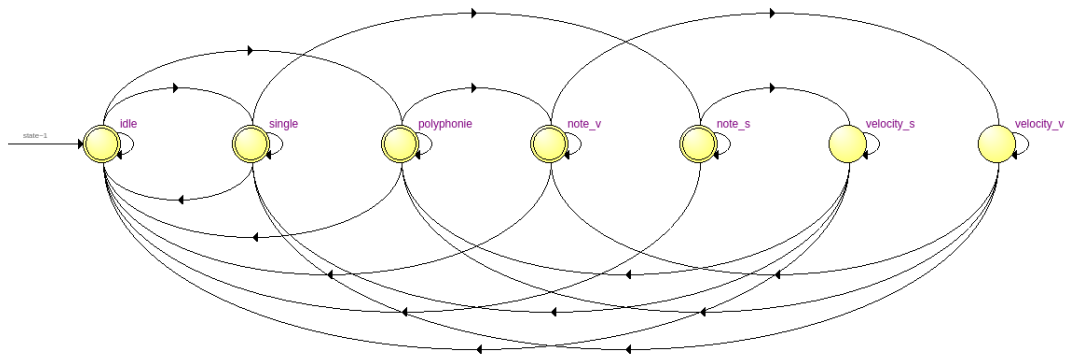


Abbildung 4.2.: fsm Übergänge

Alle drei Anforderungen sind implementiert:

Vor jedem *data byte* muss zuerst ein *status byte* eingegangen sein. Die *finite state machine* fragt im *idle* Zustand nur nach den *status bytes*. So weisen die oberen Nibbel mit den Werten "1001" für NOTE AN und "1000" für NOTE AUS auf den *single mode* hin und der Wert "1010" auf den *polyphony mode*. Nach diesen *status bytes* erwartet die *finite state machine* *data bytes*.

Die unterschiedliche Reihenfolge von Noten-Byte und Angaben zu an oder ab, wurde statusabhängig umgesetzt:

Im *single mode* wird das vierte Bit des *status nibbel* zum Setzen von AN und AB verwendet. Im *polyphony mode* wird das zweite *data byte*, das üblicherweise die Geschwindigkeit der Note bestimmt, für das Setzen von AN und AUS genommen. Ist der Wert der Geschwindigkeit gleich NULL, dann soll in diesem Zustand die Note als AUS gelten.

Gut sichtbar ist, dass die *finite state machine* bei ungültigen *bytes*, die Daten verwirft und in den *idle* Zustand zurückgeht.

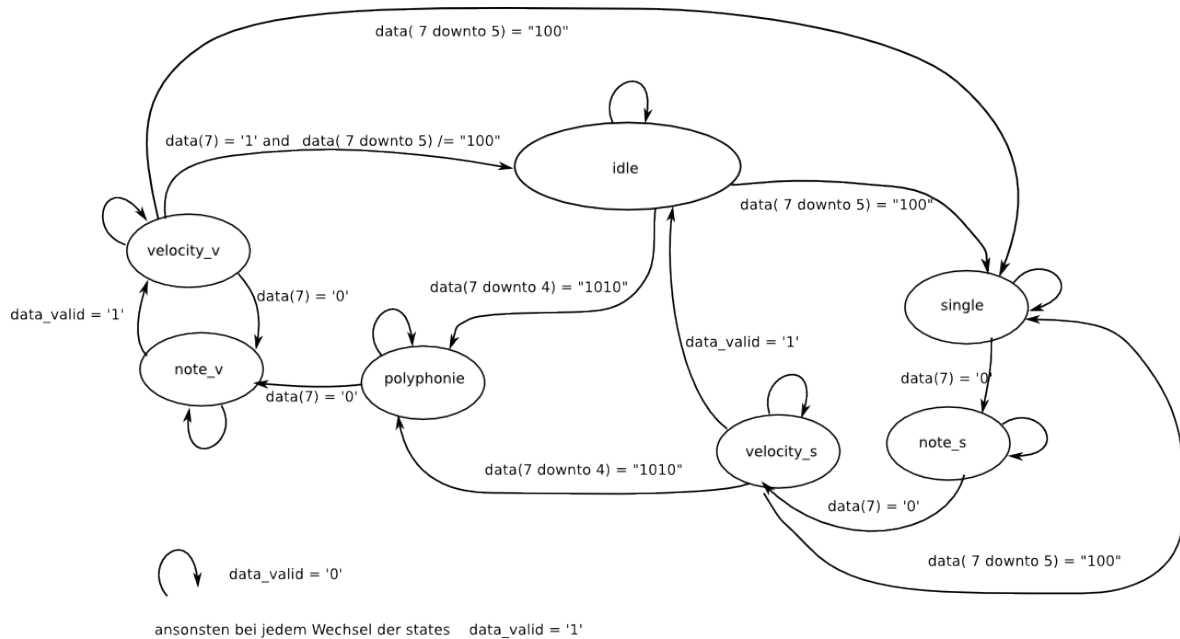


Abbildung 4.3.: fsm Übergangsbedingungen

Die Umsetzung der *fsm* sieht man in der Simulation in den zweit unteren Abbildungen gut. Nach dem Zustand *idle* folgt das *status byte* "single" (Abbildung 4.4) oder "polyphonie" (Abbildung 4.5)

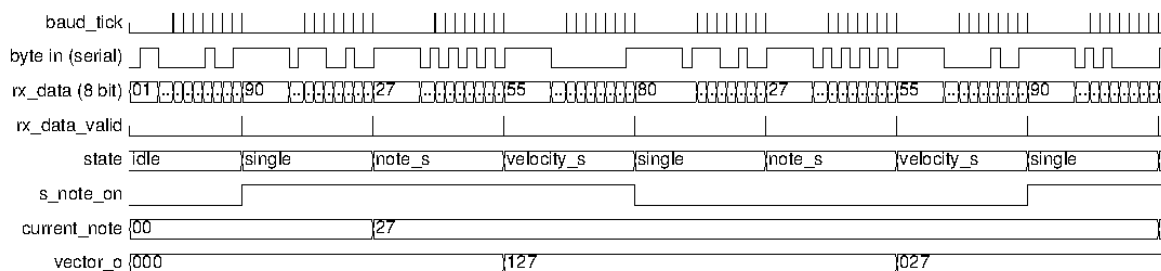


Abbildung 4.4.: fsm für single mode

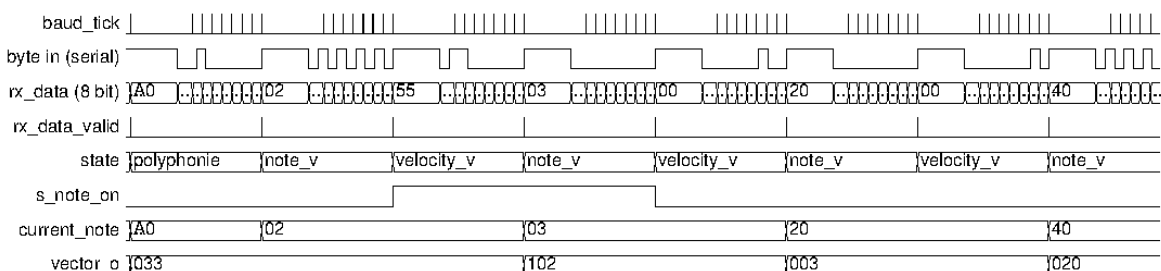


Abbildung 4.5.: fsm im polyphony mode

Die Simulation wurde mit möglichen Fehlübertragungen angereichert, damit man sieht, ob der *midi control* weiter korrekt funktioniert. Eine detaillierte Beschreibung zum Aufbau der Simulation befindet sich im Kapitel **testbench**.

4.4. Umsetzung "polyphonie outBlock"

5. Polyphonie



Abbildung 5.1.: Bildbeschreibung

5.1. Midi Spezifikation

5.2. Umsetzung

5.2.1. software nahe

hardware nahe

6. testbench

Inspiziert vom Konzept des *test driven development* wird stets parallel zur Entwicklung einer *unit* (im folgenden als Block genannt) der *unit-test* entwickelt. (Kent, 2013)

Nachdem im Voraus die Schnittstellen zwischen den Blocks geklärt sind (siehe Kapitel xXX) wird eine leere Hülle für jeden zu entwickelnden Block erstellt. Die *testbench* geht von Beginn weg vom Ziel, ein funktionstüchtiges *midi interface* aus. Solange ein Block nicht funktioniert, werden die zu testenden Signale direkt an den Ausgang der DUT geführt.

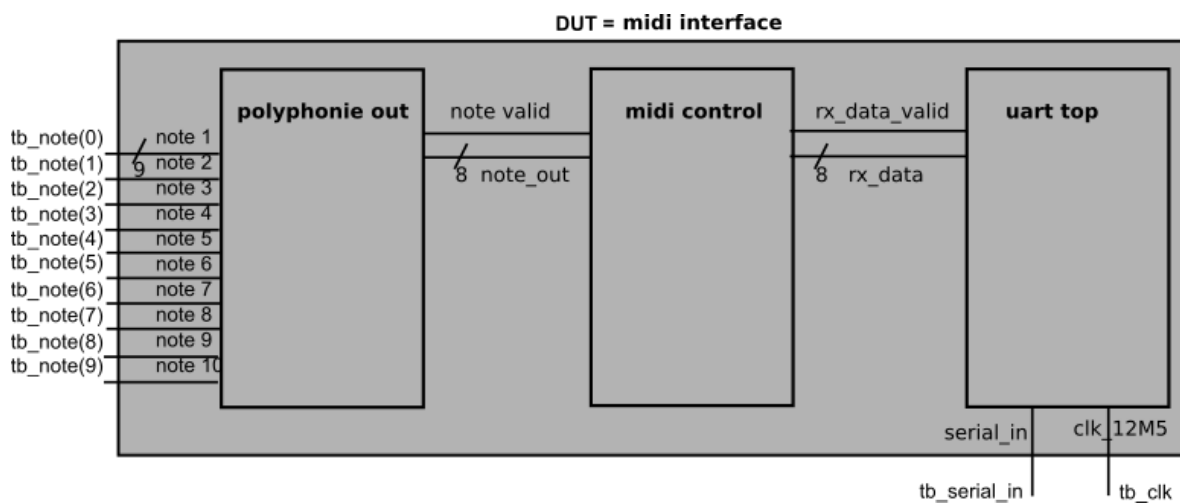


Abbildung 6.1.: Blockschaltbild Device under Test

In den nächsten zwei Unterkapiteln wird die Umsetzung der test driven Entwicklung beschrieben.

6.1. unit test midi control

6.2. unit test midi interface

7. Resultate der Projektarbeit

Zusammenfassung der Resultate

7.1. Generieren von Glitches

Beides erreicht. Viel Aufwand, da wenig Wissen wie ungewollter Zustand erzeugt werden kann. Es dauerte 4 Wochen (15. oktober + Doku), der insgesamt 16 Wochen PA.

7.2. Zustand von Metastabilität provozieren

Beides erreicht. Viel Aufwand, da wenig Wissen wie ungewollter Zustand erzeugt werden kann. Es dauerte 4 Wochen (15. oktober + Doku), der insgesamt 16 Wochen PA.

7.3. MIDI Controller entwickeln

7.4. Polyphonie Block

Midiansteuerung nach vielen Redesignes gelungen. Mehr in der Software geübt, ist das Timing in VHDL übungsbedürftig.

7.5. DDS Generatoren basierend auf Frequenzmodulation entwickeln

Sitzung: Nur 10 DDS einbauen. Schnittstellen da. Nicht da, Frequenzmodulation anstelle von LUT. Aus zeitgründen. Nur erster Entwurf, wie es umzusetzen ist.

7.6. Textbasierte Testbench für alle entwickelten Blocks

8. Diskussion und Ausblick

Bespricht die erzielten Ergebnisse bezüglich ihrer ERwartbarkeit, Aussagekraft und Relevanz
Interpretation und Validierung der Resultate
Rückblick auf Aufgabenstellung: erreicht nicht erreicht

Legt dar, wie die Resultate weiterhin genutzt werden können
an sie angeschlossen werden kann

Verzeichnis

8.1. Literaturverzeichnis

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 (Device Handbook, S. 8-19).

hold time

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

Die hold-Zeit beträgt beim Cyclone IV E 0 ns (Device Handbook, S. 8-19).

Pfadzeit

... (Unter 3.2. Metastabilität Ratschläge erwähnt)

quartus

IDE von altera zum Kompilieren, Synthesizieren und einbauen von IPs für die altera FPGAs.

setup time

minimale Zeit, in der Inputdaten stabil sein müssen *bevor* ein Taktflanke die Daten triggert.

Die setup-Zeit beträgt beim Cyclone IV E 10 ns (Device Handbook, S. 8-19)

.1. Offizielle Aufgabenstellung

Projektauftrag

.2. CD mit vollständigem Bericht

A. Anhang: Top Synthesizer

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

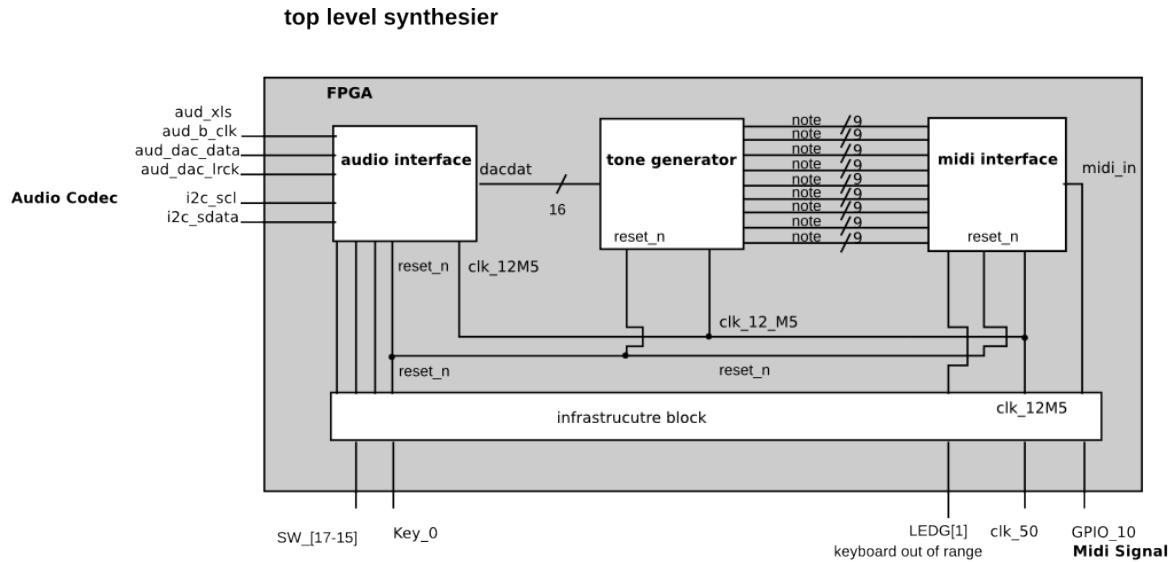


Abbildung A.1.: Top Synthesizer mit MIDI Interface: Blockschaltbild

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

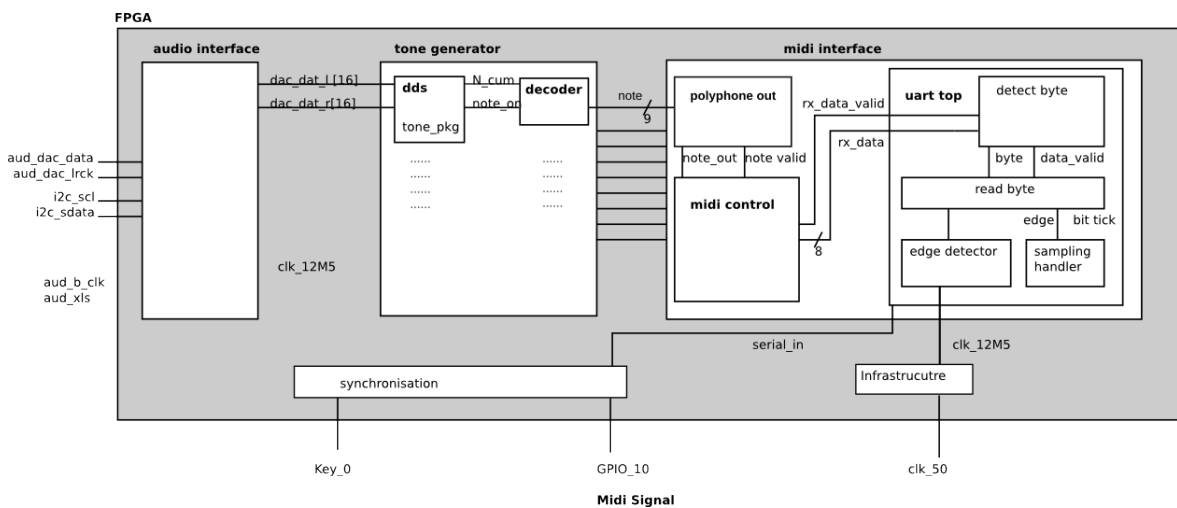


Abbildung A.2.: Top Synthesizer mit MIDI Interface: Detailansicht