

ERA Praktikum SS 2017
Digital Audio - Pegelanzeige
Seriell/Parallel-Konverter

ERA Praktikum SS 2017	1
1. Anwenderdokumentation	3
1.1 Einleitung	3
1.2 Beschreibung der Funktionalität des Programms	3
1.3 Bedienung	4
2. Entwicklerdokumentation	7
2.1 Einleitung	7
2.2 Implementierung	7
2.3 Test	10
2.4 Lösungsidee	10
2.5 Erweiterungsoptionen	11
3. Verfasser	12

1. Anwenderdokumentation

1.1 Einleitung

Im Rahmen des ERA Praktikums an der TU München wird für eine digitale Audiopegelanzeige das Teilprojekt der Implementierung eines Seriell/Parallel-Konverters durchgeführt. Um die Funktionsweise und Anwendung des Programms näher zu verdeutlichen, wird im Folgenden die Anwenderdokumentation vorgestellt.

1.2 Beschreibung der Funktionalität des Programms

Zur digitalen Tonverarbeitung wird in Musikgeräten die Übertragung von Audiosignalen genutzt. Um die Tonfrequenzen visuell auf einer Segmentanzeige darzustellen, kann eine Audiopegelanzeige angeschlossen werden.

Ziel solch einer Anzeige ist es, serialisierte, digitale 18-Bit-Audiodaten zu einer Pegelanzeige mit Spitzenwertfunktion umzuwandeln. Die Daten sind vorzeichenbehaftete Werte im Intervall zwischen - 131072 bis 131071.

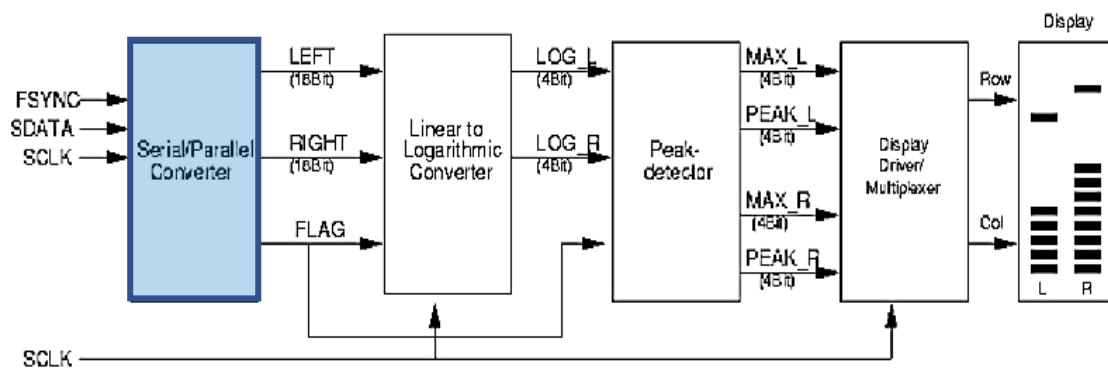


Abbildung 1

Da die Audiodaten von einer digitalen Tonquelle (beispielsweise einem Musikwiedergabegerät) als Stereosignale (siehe Abb. 2) ausgegeben werden, einfacher gesagt in unseren Baustein bitweise einzeln und aufeinanderfolgend eintreffen, ist der Seriell/Parallel-Konverter der erste Baustein der digitalen Audiopegelanzeige.

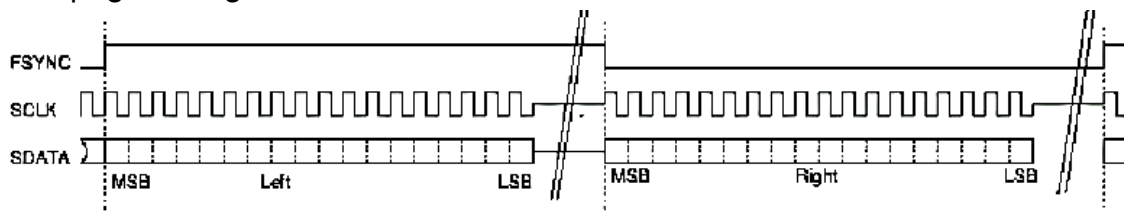


Abbildung 2

Um den seriellen Datenstrom in Datensätze von je 18 Bit umzuwandeln, wurde in VHDL eine Architecture (mit vorausgehender Entity) implementiert, die die einzelnen Bits sammelt und in regelmäßigen Abständen in zwei 18 Bit-Paketen (aufgefasst als links und rechts) zurückgibt. Basierend auf diesen beiden Werten wird zum Schluss die Anzeige generiert (siehe Abb.1 rechts).

Als Eingangssignale werden in der Entity der Datenfluss als SDATA, eine Clock SCLK und ein Framesynchronizer FSYNC definiert (s. Abb 3). An SDATA liegt der Datenstrom an, FSYNC wird mit der Samplefrequenz von 44.1 kHz gekoppelt und SCLK wird so angelegt, dass es pro FSYNC-Takt 64 mal wechselt. Diese Daten werden durch das Testprogramm zur Verfügung gestellt.

```
entity audiostp is
  port (
    sclk, fsync, sdata : in std_logic;
    left, right : out signed (17 downto 0);
    flag : out std_logic
  );
end audiostp;
```

Nach jeweils 36 Takten von SCLK werden die zwei 18-Bit-Datensätze in Abhängigkeit von FSYNC an den Signalen RIGHT bzw. LEFT ausgegeben, die in der Entity als Ausgänge festgelegt werden. Dies geschieht so, dass die ersten 18 Bit der eingehenden Daten in LEFT abgelegt werden und die nächsten 18 Bit in RIGHT. Außerdem wird das Flag, welches während der Bitsammelphase den Wert 0 hat, wieder auf 1 gesetzt, nachdem ein Samplepaar ausgegeben wurde (dies dient dem nachfolgenden Baustein des Logarithmisierers).

Falls das Programm in der Mitte der Bitgrenze startet, beginnt die Konvertierung erst zur neuen wechselnden Flanke von FSYNC.

Genauere Beschreibung zur Funktionsweise, besonders innerhalb der Architecture, befindet sich in der Entwicklerdokumentation (2.2 Implementierung).

1.3 Bedienung

Zum Kompilieren wird GHDL verwendet, ein Opensource Simulator für VHDL. Zum visuellen Verfolgen des Programms dient GTKWave (ein sogenannter Waveviewer), welches viele Hardwarebeschreibungssprachen in „Wellenform“, genauer die typischen Signaldiagramme (s. Abb 2) darstellt.

Es sollte sichergestellt sein, dass sowohl GTKWave, als auch GHDL für das entsprechende Betriebssystem installiert ist.

Vor Programmstart sollten sich das Makefile, das Testprogramm sowie die eigentliche Codedatei im selben Verzeichnis befinden (siehe markierte Dateien in Abb.3).






Semester 2 > erapraktikum > ss17-g45 > Projekt2 > Implementierung			
Name	Änderungsdatum	Typ	Größe
 Makefile	09.07.2017 20:12	Datei	1 KB
 README	15.07.2017 21:23	Datei	2 KB
 screenshot	15.07.2017 21:23	PNG-Datei	631 KB
 stp	09.07.2017 20:12	VHDL-Datei	5 KB
 stp_tb	09.07.2017 20:12	VHDL-Datei	13 KB

Abbildung 3

Dann muss über das Terminal (Eingabeaufforderung) in diesen Ordner navigiert werden (Beispiel Abb. 4).

```
C:\Users\Thea>cd C:\Users\Thea\Dropbox\Uni\Semester 2\erapraktikum\ss17-g45\Projekt2\Implementierung
C:\Users\Thea\Dropbox\Uni\Semester 2\erapraktikum\ss17-g45\Projekt2\Implementierung>_
```

Abbildung 4

In eben diesem Verzeichnis soll der "make" Befehl aufgerufen werden. Sobald GTKWave gestartet ist, muss man "stp.vcd" in GTKWave öffnen (per Doppelklick auf die Datei) und dort den audioconverter im linken oberen Feld auswählen (s. Abb 5). Es sollten sämtliche Ein- und Ausgabesignale im linken unteren Bereich angezeigt werden. Nun wählt man alle diese Signale aus und fügt sie rechts daneben per insert in das Programm ein.

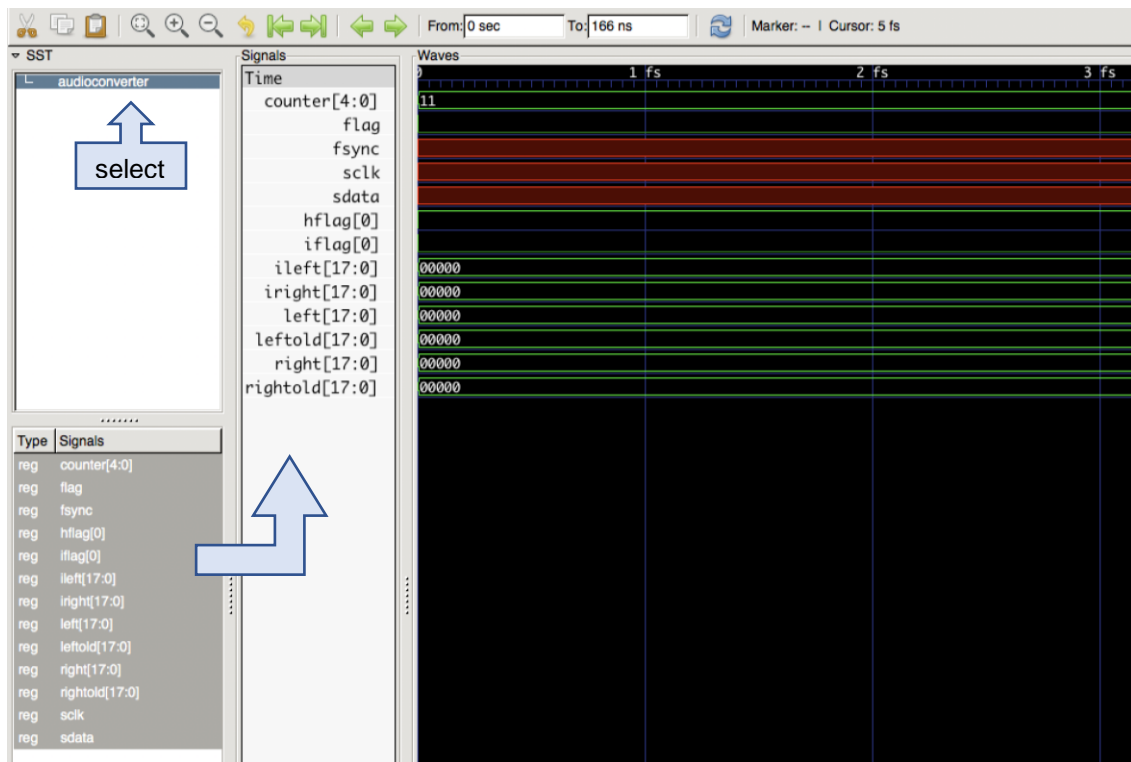
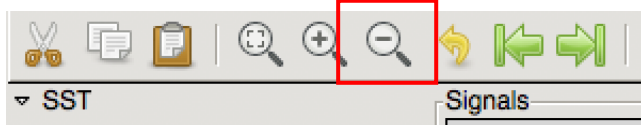


Abbildung 5

Um Veränderungen zu sehen, muss man in der Menüleiste so lange herauszoomen, bis das Diagramm in Nanosekundenschritten angezeigt wird:



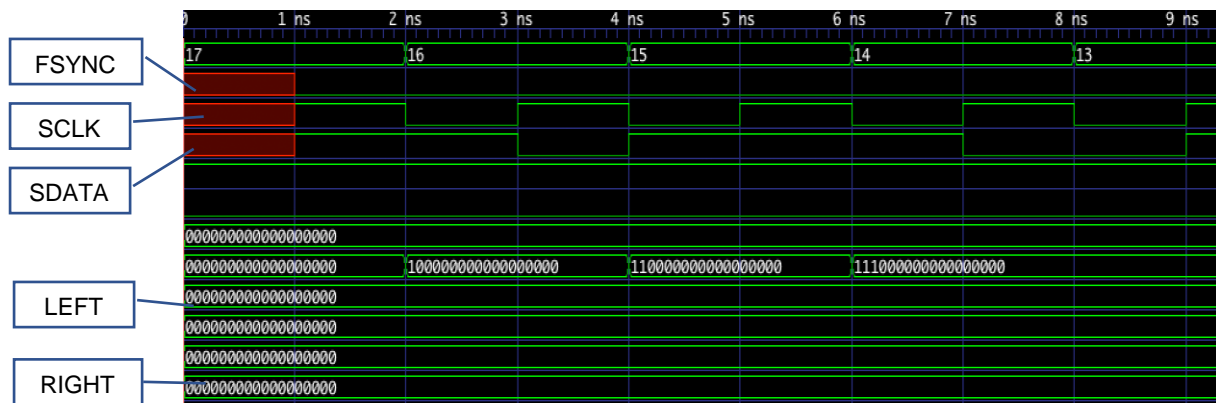


Abbildung 6

Im Diagramm sollten jegliche Signale und ihre Werte abhängig voneinander als Wellen dargestellt zu sehen sein. Zum besseren Verständnis können die eingefügten Signale noch per Auswahl und Rechtsklick je nach Belieben in das gewünschte Zahlensystem umgewandelt werden. Hierbei empfiehlt es sich binär für die LEFT- und RIGHT-Ausgänge, sowie dezimal für den Counter zu wählen. Auf diese Art lässt sich erkennen, wie in den beiden Kanälen direkt über LEFT (s. Abb 6) die einzelnen Bits identisch zu dem jeweiligen Wellenwert von SDATA bei jeder Counterdekrementierung (erster Kanal) zu den Vektoren hinzugefügt werden. Man sieht auch, wie diese Vektoren bei Flankenwechsel von FSYNC letztendlich auf LEFT und RIGHT übertragen werden, was Hauptziel des Programms ist. Über den Befehl "make clean" kann abschließend das Verzeichnis wieder in die Ausgangslage gebracht werden.

2. Entwicklerdokumentation

2.1 Einleitung

Angeleitet durch das ERA Praktikum an der TU München wird das Projekt eines Seriell/Parallelkonverters als Teil einer Audiopegelanzeige umgesetzt. Es folgt die Entwicklerdokumentation, die dazu dient, Entwicklern die Implementierungsweise verständlicher zu gestalten und eventuelle Adaptionen oder Weiterentwicklungen leichter und schneller umzusetzen.

2.2 Implementierung

Die Implementierung teilt sich auf in drei Dateien: die Makefile, das Testprogramm und das Hauptprogramm. Während der Implementierung von Haupt- und Testprogramm wurde Sublime Text genutzt, um den Code optisch leichter lesbar zu gestalten und gemeinsam arbeiten zu können. Jegliche hier gezeigten Screenshots der Codeausschnitte sind in der Sublime Text Umgebung entstanden.

Das Makefile dient zur leichteren Ausführung des Programms und automatisiert die einzelnen Schritte im Terminal (kompilieren, simulieren, ausführen), sodass das Programm lediglich mit dem Befehl `make` ausgeführt werden kann. Im Folgenden wird sich auf das Hauptprogramm konzentriert:

Wie schon in der Benutzerdokumentation erwähnt, werden als Eingangssignale `SDATA`, `SCLK` und `FSYNC` gesetzt.

Als Ausgangssignale werden `LEFT` und `RIGHT` definiert, an welchen die parallelisierten 18 Bitdatensätze ausgegeben werden, sowie ein Flag, das nachfolgendem Baustein zur Trennung der einzelnen Datensätze dient (es wird bei jeder neuen Ausgabe von `LEFT` und `RIGHT` für einen Takt auf 1 gesetzt).

```
entity audiostp is
  port (
    sclk, fsync, sdata : in std_logic;           -- Eingänge definieren
    left, right : out signed (17 downto 0);      -- 18 Bit Ausgabewerte
    flag : out std_logic                         -- 1 Bit Ausgabeflag
  );
end audiostp;
```

Die Architecture beginnt mit einer Signalzuweisung, um intern Eingangswerte zu speichern und Ausgangswerte richtig setzen zu können. Es werden zunächst zwei Vektoren `ileft` und `iright` benötigt, die die eingehenden einzelnen Bits von `SDATA` sammeln und schlussendlich `RIGHT` und `LEFT` zugewiesen werden.

Das Signal `iflag` wird gebraucht, um zu signalisieren, dass `ileft` und `iright` vollständig eingelesen wurden und der Ausgabe zugewiesen werden können. `hflag` dient, um das Ausgabeflag für nur einen Takt auf 1 zu setzen.

```

architecture converter of audiostp is

    -----SIGNAL DEFINITION-----
    signal ileft,  iright : signed (17 downto 0) := "000000000000000000";
    signal iflag : unsigned (0 downto 0) := "0";

    signal hflag : unsigned (0 downto 0) := "0";
    signal counter : unsigned(4 downto 0) := "10001";
    signal rightold : signed ( 17 downto 0 ) := "000000000000000000";
    signal leftold : signed ( 17 downto 0 ) := "000000000000000000";

    -----SIGNAL DEFINITION END-----

```

Das Signal counter bestimmt, ob die eingehenden Daten in ileft oder iright eingelesen werden. Rightold und rightnew fungieren lediglich als Zwischenspeicher für RIGHT und LEFT, da generell in Schaltungen Ausgänge nicht gelesen werden können. Alle Signale sind anfangs auf null gesetzt, bis auf counter, welcher bei 17 (binär 10001) beginnt, um danach runterzuzählen.

Gehen wir nun über zum eigentlichen Beginn der Architecture.

Es wird ein SCLK (und FSYNC) abhängiger Prozess gestartet, das heißt nur wenn SCLK (FSYNC wechselt alle 18 Takte zusammen mit SCLK) sich ändert, wird der Prozess ausgeführt, somit findet die gesamte Ausgabe getaktet statt. Bei jedem Wechsel von FSYNC wird der Counter wieder auf 17 gesetzt, da dies alle 18 Takte geschieht, zählt der Counter ab jedem 18-Bit-Paket neu abwärts und kennzeichnet so den Wechsel zu RIGHT bzw. LEFT.

```

47     process (fsync, sclk)
48     begin
49         if falling_edge(fsyc) or rising_edge(fsyc) then
50             counter <= "10001";
51         elsif falling_edge (sclk) then
52             case fsync is
53                 when '1' => ileft (to_integer(unsigned(counter))) <= sdata;
54                 when others => iright (to_integer(unsigned(counter))) <= sdata;
55             end case;
56             if counter > 0 then
57                 counter <= counter - 1 ;
58             end if;
59             if counter = 0 then
60                 iflag <= iflag + 1;
61             end if;
62         end if;
63     end if;
64     end process;
65

```


Falls FSYNC 1 ist, werden die eingehenden Bitwerte ileft zugewiesen, ist FSYNC 0, dann zu iright (Zeile 52-54). Da SDATA vom Datentyp std_logic ist und iright bzw. ileft unsigned-Vektoren sind, muss SDATA erst zu unsigned und dann zu einem Integer gecastet werden, was bei der Zuweisung in Z.53/54 geschieht.

In Zeile 57 wird der Counter, sofern er nicht null ist, pro Prozessaufruf um eins dekrementiert, zählt also die Anzahl noch nicht bearbeiteter Stellen in ileft bzw. iright. Wenn er bei null angekommen ist, wechselt iflag seinen Wert (Z. 60), iflag ist somit immer 0 wenn ileft bearbeitet wird, und 1 bei Bearbeitung von iright.

An dieser Stelle ist der Prozess zuende und das Concurrent Statement beginnt, welches den Ausgängen neue Werte zuweist. Die Zuweisungen eines Concurrent Statements (\leq) passieren im Gegensatz zu z.B. Signalzuweisungen innerhalb des Prozesses ($:=$) nicht sofort, sondern erst nach einer Zeit delta.

```
70 -----
71 -----OUTPUT-----
72 -----
73 hflag <= "1" when iflag = "0"
74
75     else "0";
76
77     rightold <= iright when iflag = "0" and hflag = "0"
78         else rightold;
79     leftold <= ileft when iflag = "0" and hflag = "0"
80         else leftold;
81
82     left <= leftold;
83     right <= rightold;
```

Diese Eigenschaft wird sich zunutze gemacht, indem hflag in Zeile 73 1 zugewiesen wird, falls iflag 0 ist, es sich aber erst später ändert und somit für die nächsten Zeilen noch den Wert 0 besitzt.

Rightold und leftold behalten somit stets ihre alten Werte bei bis hflag und iflag beide null sind, was nur einen Takt der Fall ist, da wenn iflag auf null gesetzt ist, hflag kurze Zeit später (nach Festlegung der Zuweisungen von rightold und leftold) auf eins gesetzt wird. Dadurch wird gewährleistet, dass sich LEFT und RIGHT nur einmal am Ende der beiden 18-Bit-Datensätze aktualisieren.

2.3 Test

Da in VHDL keine Eingabemöglichkeit über Register, Konsolen oder ähnliches möglich ist, wird eine Testbench benötigt, welche Daten an die Eingänge des Hauptprogramms legt und die Ausgänge liest. Der Aufbau unseres Testcodes `stp_tb.vhdl` besteht wie das Hauptprogramm aus Entity und Architecture, jedoch wird die Entity - wie für Testbenches üblich - leer belassen. Die Architecture startet mit einem component in dessen port nochmals alle Ein- und Ausgänge identisch zur Entity des Konverters definiert werden.

```

11
12 architecture test of audiostp_tb is
13     component audiostp
14     port (
15         sclk, fsync, sdata : in std_logic;
16         left, right : out signed (17 downto 0);
17         flag : out std_logic
18     );
19     end component;
20

```

Bei port map werden die Aus- und Eingänge des components (SCLK, FSYNC, SDATA, ...) auf entsprechend benannte Signale geführt.

```

~
21     signal sclk, fsync, sdata, flag: std_logic;
22     signal left, right: signed (17 downto 0);
23
24 begin
25     audioconverter: audiostp port map (sclk => sclk, fsync => fsync, sdata => sdata,
26         flag => flag, left => left, right => right);
27

```

Das Testprogramm wird gestoppt durch ein wait-Statement am Ende des Codes. Im Prozess wird anfangs SDATA, SCLK und FSYNC X zugewiesen, um einen neutralen Wert zu erzeugen, danach wird SCLK manuell im Abstand einer Nanosekunde abwechselnd 0 und 1 zugewiesen, um den Takt zu simulieren. FSYNC wechselt nach 18 Nanosekunden und SDATA werden zufällig ausgewählte Werte zugewiesen.

2.4 Lösungsidee

Der allgemeine Lösungsweg der Implementierung des Konverterbausteins beschäftigt sich hauptsächlich mit der Weise, wie die Bits intern gesammelt werden. Hierbei besteht die Schwierigkeit darin, die Bits nach LEFT und RIGHT zu trennen sowie das Flag für genau einen Takt bei der Änderung von LEFT und RIGHT zu setzen.

Es bietet sich einerseits an, in einem internen Vektor 36 Bit zu speichern und diese danach auf die beiden Ausgänge aufzuteilen oder zwei 18 Bit große Vektoren zur internen Speicherung zu nutzen.

Ein Ansatz wäre, mithilfe eines 6-Bit Counters (dezimal von 35 bis 0) die Bits innerhalb des Prozesses nacheinander in einem internen Vektor output zu speichern. Erreicht der Counter den Wert null, wird ein Ausgabeflag gesetzt und LEFT und RIGHT jeweils die Hälfte des output-Vektors zugewiesen, sowie der Counter bei der nächsten steigenden FSYNC Flanke wieder auf 35 gesetzt. Diese

Implementationsweise ist fehleranfälliger und unter Umständen beim Einstieg in das Programm fehlerhaft. Außerdem gestaltet es sich als schwierig, das Ausgabeflag für genau einen Takt zu setzen. Eine genauere Erläuterung zum alternativen Lösungsweg befindet sich in der Spezifikation dieses Projektes.

Wir entschieden uns für eine andere Herangehensweise. Der final implementierte Lösungsweg speichert wie in der Implementierung erklärt die seriellen Bits in zwei 18-Bit-Vektoren äquivalent zu den Ausgangsvektoren LEFT und RIGHT. Dadurch wird zusätzlich das Verständnis des Codes erleichtert, da die zwei internen Vektoren ileft und iright beim Lesen im Gehirn direkt mit RIGHT und LEFT assoziiert werden können. Dies ist hilfreich, da die Implementierung Teil eines größeren Projektes ist und unter Umständen Entwickler der anderen Bauteile mit unserem Code arbeiten.

Des Weiteren arbeitet unsere Lösung mithilfe eines 5-Bit-Vektors als Counter und zeichnet sich außerdem durch die zwei internen Flags hflag und iflag aus, um das Ausgangsflag richtig zu setzen. Durch die leicht verspätete Zuweisung der Concurrent Statements um einen Zeitfaktor delta sind hflag und iflag für einen kurzen Moment beide null, weshalb an die Ausgänge LEFT und RIGHT die neuen Werte angelegt werden können.

Wir achteten einerseits besonders darauf, dass zu jedem Zeitpunkt des Programms eine richtige Ausgabe der Bitstränge stattfindet, andererseits war uns das korrekte Setzen des Flags wichtig. Beide Kriterien werden in unserer Implementierung dank interner Flags und einem FSYNC abhängigen Counter stets erfüllt, zusätzlich ist sie leicht verständlich und intuitiv, weshalb wir diesen Lösungsweg als den besten erachten.

2.5 Erweiterungsoptionen

Weil das Programm Teil eines größeren Projekts ist, gibt es hinsichtlich des Hauptprogramms keine Erweiterungsmöglichkeiten, da man sonst die restlichen Bausteine beeinträchtigen würde. Jedoch ließe sich die Testbench noch weiter ausarbeiten, indem man die Clock automatisierte.

FSYNC ist abhängig von der Samplefrequenz 44,1 kHz, was umgerechnet 22675.74 ns sind. Also dauert ein Takt von FSYNC 22675.74 ns. Da SCLK pro FSYNC-Zyklus 64 mal wechselt, würde in der Testbench der SCLK-Takt auf $22675.74 \text{ ns} / 64 = 354 \text{ ns}$ gesetzt werden, um die Simulation noch praxisnaher zu gestalten (an der Funktion und Richtigkeit des Programmes ändert dies jedoch nichts). Um dies zu erreichen, wird in der vorhandenen Testbench dem Signal SCLK der Anfangswert 0 zugewiesen:

```
signal sclk: std_logic := '0';  
signal fsync, sdata, flag: std_logic;  
signal left, right: signed (17 downto 0);
```

Außerdem wird ein Clockprocess direkt nach begin und der port map eingefügt:

Somit wird die Taktvorgabe des Programms automatisiert und es muss nur FSYNC und SDATA zugewiesen werden (ähnlich zur bestehenden Testbench).

```
clock_process: process
  clk <= not clk after 354 ns;
  wait for 354 ns;
end process clock_process
```

3. Verfasser

Diese Dokumentation wurde von Thea Kramer verfasst, basiert auf selbstständiger Projektarbeit mit Florian Müller und Berzan Mikaili und wurde am 23.07.2017 fertig gestellt und eingereicht.