ERA Praktikum SS 2017

Digital Audio - Pegelanzeige

Seriell/Parallel-Konverter

1. Einleitung

Im Rahmen des ERA-Praktikums an der TU München sollen zwei Projekte in einem Team von drei Studierenden bearbeitet werden.

Dies ist die Spezifikation zum Projekt 1 “Digital Audio - Pegelanzeige: Seriell/Parallel-Konverter”.

2. Aufgabenkurzbeschreibung

Die Grundlage der Aufgabenstellung bietet eine Audioübertragung durch ein serielles Protokoll. Dafür werden drei Eingangssignale definiert: die zu übertragenden Daten (`SDATA`), die getaktete Shift Clock (`SCLK`) sowie ein Frame Sync zur Aufteilung des Datenstroms. Im Detail werden die vorzeichenbehafteten Audiodateien `SDATA` bei steigender Flanke von `SCLK` ausgegeben. Zusätzlich wechselt bei jedem neuen Dateiabschnitt `FSYNC`, um den neuen Anfang zu signalisieren. Letztgenannte Leitung ist als einzige zeitlich abhängig von der Samplefrequenz von `FSYNC` = 44.1 kHz. Pro FrameSync-Takt treten 64 `SCLK`-Takte auf.

Ziel der Aufgabe ist es, einen Seriell/Parallel-Konverter in VHDL zu implementieren. Hierbei wird der serielle Datenfluss als Eingangssignal angelegt, somit in der Entity definiert. Im als Konverter fungierenden Architectureabschnitt werden diese Signale umgewandelt in ein Ausgangssignal. Dieses stellt zwei 18-Bitwerte dar (`LEFT` und `RIGHT`). Nach jeder Datenstromsequenz wird ein Flag gesetzt, um den Beginn eines neuen Abschnitts zu kennzeichnen.

Um dies zu verwirklichen, werden im Folgenden zwei Vorgehensweisen vorgestellt.

3. Lösungsansätze

Nach genauerer Abwägung kristallisieren sich zwei Herangehensweisen heraus, die sich im Hinblick auf die Architektur unterscheiden: Beide Lösungsansätze werden durch Funktionalitätsbeschreibung mit Prozessen umgesetzt. Bei Lösungsweg A wird der Datenstrom in einen internen 36-Bit großen Vektor eingelesen, ein `Counter` dient zur Terminierung. Bei Lösungsweg B hingegen werden die Daten intern in zwei verschiedenen 18-Bit großen Vektoren gespeichert, der internen Steuerung dienen verschiedene interne Flags.

Bei beiden Ansätzen wird eine Entity geschrieben, in der `SDATA`, `SCLK` und `FSYNC` als Eingangssignale definiert werden. Durch den Eingang `SDATA` gehen 18-Bit Werte im Intervall von -131072 bis 131071 als serieller Datenstrom ein.

2.1 Lösungsweg A

In der Architecture werden zunächst zwei interne Signale definiert: Ein Zähler `signal counter: signed(5 downto 0) = 35` wird festgelegt, der die Bits des Eingangs `SDATA` zählt, um `LEFT` und `RIGHT` gleichzeitig als Vektoren ausgeben zu können. Außerdem wird der interne Vektor `output`, in denen die einzelnen Bits, bis zur Ausgabe gespeichert werden, als `signal output: signed (35 downto 0)`, definiert.

Nun wird ein von `SCLK` abhängiger Prozess gestartet, der dem Outputvektor beginnend mit dem höchstwertigen Bit nacheinander die einzelnen Bits an der Stelle des `Counters` zuweist. Nach jeder Zuweisung wird der `Counter` dekrementiert.

Ist der `Counter` bei 0 angelangt, wird das Ausgabeflag gesetzt und den Ausgängen `LEFT` und `RIGHT` die jeweilige Hälfte des internen Vektors `Output` zugewiesen.

Außerdem wird bei jeder steigenden Flanke von `FSYNC` der `Counter` wieder auf 35 gesetzt.

Falls der Datenstrom nicht an der Bitgrenze beginnt, sondern bei Bit $x \neq MSB$, so werden in diesem Ansatz die ersten Bits bis zum nächsten Beginn falsch konvertiert.

Im Folgenden ist eine beispielhafte Umsetzung in VHDL-Code zu sehen.

```
1
2  entity audio_s2p is
3      port (
4          sclk, fsync, sdata : in std_logic;              -- Eingänge definieren
5          left, right : out signed ( 17 downto 0 );       -- 18 Bit Ausgabewerte
6          flag : out std_logic                            -- 1 Bit Ausgabeflag
7      );
8  end audio_s2p;
9
10
11 architecture converter of audio_s2p is
12     signal output : signed ( 35 downto 0 );             -- interne Zwischenspeicherung der seriellen Daten
13     signal counter : unsigned ( 5 downto 0 ) = 35;      -- Zähler um serielle Daten in Vektor einzufügen
14
15     process ( sclk )                                    -- nur wenn sclk sich ändert, wird process betreten
16     begin
17         if falling_edge ( sclk )                        -- wenn es sich um eine fallende Kante von sclk handelt
18             output(counter) <= sdata;                   -- keine Unterscheidung von left / right zu diesem Zeitpunkt
19             counter <= counter - 1 ;                    -- nach dem Einfügen wird counter um 1 dekrementiert
20         end if;
21     end process;
22
23     process ( fsync )                                   -- nur wenn sich fsync ändert, wird process betreten
24     begin
25         if rising_edge ( fsync ) then
26             counter = 35;                               -- nur bei steigender Flanke fsync wird counter zurückgesetzt
27                                                         -- also erst wenn sowohl left als auch right eingelesen sind
28         end if;
29     end process;
30
31     right <= output ( 17 downto 0 ) when counter = '0'  -- wenn counter auf 0 steht, also left und
32                                                         -- right eingelesen wurden, werden die unteren 18 bit
33                                                         -- von output right zugewiesen
34                 else right;                             -- ansonsten bleibt der Wert von right beibehalten.
35     left <= output ( 35 downto 18 ) when counter = '0'  -- analog zur Zuweisung von right werden left die oberen
36                                                         -- 18 bit von output zugewiesen wenn counter auf 0 steht
37                 else left;                              -- ansonsten bleibt der Wert von right beibehalten
38     flag <= '1' when counter = '0'                      -- Ausgabeflag wird für einen Takt auf 1 gesetzt
39                                                         -- wenn counter auf 0 steht
40                 else '0';                               -- (im nächsten Durchlauf wurde counter resettet
41                                                         -- und flag wird wieder auf 0 gesetzt)
42 end converter;
43
```

2.1 Lösungsweg B

Da in diesem Lösungsansatz die eingelesenen Werte in zwei 18-Bit Vektoren `i_left` und `i_right` zwischengespeichert werden, muss hier nur ein 5-Bit Counter `signal counter: signed (4 down to 0)` definiert werden. Bei fallender Flanke der Clock und wenn `FSYNC` auf null gesetzt ist, werden im Prozess die Bits von `SDATA` `i_right` zugewiesen, ansonsten `i_left`.

Schließlich gibt es noch zwei interne Flagsignale `h_flag, i_flag: unsigned=0`, um die Ausgänge richtig setzen zu können. Wenn `i_flag` und `h_flag` beide null sind, wird dem Ausgangsflag 1 zugewiesen, sonst 0. Weitere Details sind im unten zu findenden Code beschrieben. Wie in Lösungsansatz A wird nach jeder Zuweisung der Counter um eins dekrementiert.

Gleichzeitig wechselt das interne Flag `i_flag` seinen Wert, wenn der `Counter` 0 ist, somit nach jedem 18-Bit Paket des Datenstroms. Sobald `i_flag` wieder auf 0 gesetzt ist, ist also ein kompletter `LEFT / RIGHT` Zyklus vorbei und den Ausgängen `LEFT` und `RIGHT` werden die Werte der internen Zwischenspeicher `i_left` und `i_right` zugewiesen.

In einem Concurrent Statement wird das interne Hilfsflag `h_flag` auf 1 gesetzt, wenn `i_flag` auf 0 gesetzt ist. Durch die Eigenschaft eines Concurrent Statements wird diese Änderung an `h_flag` für die anderen Concurrent Statements erst im nächsten Durchlauf sichtbar.

Der Ausgang FLAG wird auf 1 gesetzt, wenn sowohl `h_flag` als auch `i_flag` 0 sind. Da dies durch die Zuweisung von `h_flag` nur einen Durchlauf der Fall ist, wird `FLAG` im nächsten Durchlauf wieder auf 0 gesetzt.

Da der Counter bei jeder Änderung von `FSYNC` auf den Wert 18 zurückgesetzt wird, ist es kein Problem, wenn der Datenstrom in der Mitte eines 18-Bit Wertes beginnt. In diesem Fall würde erst mit der Ausgabe begonnen werden, wenn der nächste vollständige Abschnitt beginnt.

Auf der nächsten Seite folgt ein erster Implementierungsansatz in VHDL-Code.

```
1
2  entity audio_s2p is
3      port (
4          sclk, fsync, sdata : in std_logic;              -- Eingänge definieren
5          left, right : out signed ( 17 downto 0 );       -- 18 Bit Ausgabewerte
6          flag : out std_logic                            -- 1 Bit Ausgabeflag
7      );
8  end audio_s2p;
9
10
11 architecture converter of audio_s2p is
12     signal i_left, i_right : signed ( 17 downto 0 );    -- interne Zwischenspeicherung der seriellen Daten
13     signal i_flag : unsigned ( 0 downto 0 ) = 0;        -- internes Flag um zu speichern, ob sowohl left, als auch
14                                                         -- right eingelesen wurden und zur Ausgabe bereit sind
15     signal h_flag : std_logic;                          -- internes Hilfsflag
16     signal counter : unsigned ( 4 downto 0 ) = 17;      -- Zähler um serielle Daten in Vektor einzufügen
17
18     process ( sclk )                                    -- nur wenn sclk sich ändert, wird process betreten
19     begin
20         if falling_edge ( sclk )                        -- wenn es sich um eine fallende Kante von sclk handelt
21             case fsync is                               -- Je nach aktuellem Wert von fsync wird ...
22                 when '1' => i_left ( counter ) <= sdata; -- ... sdata in den 'linken' Vektor an Stelle 'counter' eingefügt
23                 when others => i_right ( counter ) <= sdata; -- ... oder in den 'rechten' Vektor eingefügt
24             end case;
25             counter <= counter - 1 ;                    -- nach dem Einfügen wird counter um 1 dekrementiert
26             if counter = 0 then                         -- wenn counter = 0,
27                 i_flag <= i_flag + 1;                   -- wird i_flag invertiert (markiert, dass left bzw.
28                                                         -- right nun vollständig ist)
29             end if;
30         end if;
31     end process;
32
33     process ( fsync )                                   -- nur wenn sich fsync ändert, wird process betreten
34     begin
35         counter = 17;                                   -- wenn fsync sich ändert, wird counter zurückgesetzt
36                                                         -- (um Fehler zu vermeiden, falls Programm nicht direkt
37                                                         -- beim Beginn eines 18 Bit Datenstroms gestartet wird)
38     end process;
39
40     h_flag <= '1' when i_flag = '0'                     -- Wenn i_flag auf 0 gesetzt ist, wird h_flag auf 1 gesetzt.
41                                                             -- Da dies in einem Concurrent Statement geschiet, wird die
42                                                             -- Änderung des h_flags erst beim nächsten Durchlauf für
43                                                             -- die anderen Statements sichtbar, was ermöglich, dass
44                                                             -- das Ausgabeflag im nächsten Takt wieder auf 0 gesetzt wird.
45                                                         -- wenn i_flag auf 1 gesetzt ist, wird h_flag auf wieder auf 0 gesetzt
46             else '0';
47
48     right <= i_right when i_flag = '0' and h_flag = '0' -- erst wenn i_flag wieder auf 0 gesetzt wurde (also nach
49                                                         -- einem kompletten left-right Zyklus) und das Hilfsflag
50                                                         -- auf 0 steht (siehe h_flag), wird dem Ausgang 'right' der
51                                                         -- neue Wert von i_right zugewiesen
52                 else right;                             -- ansonsten bleibt der Wert von right beibehalten.
53     left <= i_left when i_flag = '0' and h_flag = '0'   -- analog zur Zuweisung von right
54                 else left;
55     flag <= '1' when i_flag = '0' and h_flag = '0'      -- Ausgabeflag wird für einen Takt auf 1 gesetzt wenn i_flag
56                                                         -- auf 0 steht (also nach einem kompletten left-right Zyklus)
57                                                         -- und h_flag auch auf 0 steht.
58                 else '0';                               -- Da h_flag im nächsten Durchlauf bereits auf 1 steht (siehe h_flag)
59                                                         -- wird Ausgabeflag im nächsten Durchlauf wieder auf 0 gesetzt.
60
61 end converter;
62
```

4. Bewertung der Ansätze

	Lösungsansatz A	Lösungsansatz B
Vorteile	• Hohe Effizienz durch simplere interne Signalverarbeitung • Einfache Implementierung ohne interne Flags	• Signalbeginn ist irrelevant • Fehlerunanfälliger da `LEFT` und `RIGHT` intern getrennt
Nachteile	• Bei Beginn abweichend vom Signalbeginn kommt es zu Fehlern • Somit ist Konvertierung nur von Signalbeginn an möglich	• Leicht ineffizient im Vergleich zu Ansatz A • Aufwändigere Implementierung als Ansatz A

5. Entscheidungswahl

Lösungsansatz A wäre weniger aufwändig zu implementieren und etwas effizienter als Ansatz B, da nur ein internes 36-Bit größtes Signal zur Speicherung der `LEFT` und `RIGHT` Datenströme benötigt wird und auf die internen Flags verzichtet werden kann.

Problematisch ist jedoch, dass, falls die Konvertierung nicht genau am Anfang des `SDATA` erfolgt, der 36-Bit `Counter` versetzt wird und die `LEFT` und `RIGHT` Datenströme sich überlappen.

Unserer Meinung nach ist die Stabilität und das Funktionieren des Programmes unabhängig von dem Zeitpunkt der Konvertierung bei Lösungsansatz B wichtiger, als die geringfügig leichtere Implementierung oder der Effizienzgewinn bei Lösungsansatz A, da letztendlich unsere Komponente auch Teil der Audiopegelanzeige ist und die restlichen Komponenten sich auf unsere Konvertierung verlassen müssen.

Wir wählen also Lösungsansatz B zur Verwirklichung des Seriell/Parallel-Konverters in VHDL.