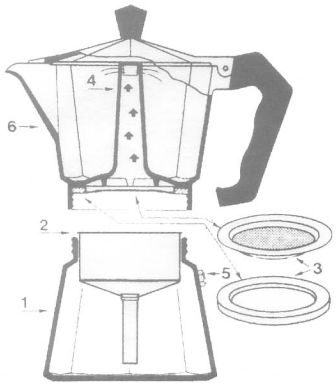


Die Espressomaschine wird in 7 Größen hergestellt.

Gebrauchsanweisung: Füllen Sie den unteren Behälter der Kaffeemaschine bis zum Rand mit Wasser. Dann den Siebeinsatz ...

Ein einwandfreies Funktionieren kann nur gewährleistet werden, wenn Sie die Kaffeemaschine laut nebenstehender Skizze in der angeführten Reihenfolge zusammensetzen. Vergessen Sie keinen Teil!

Um einen echten „Espresso Kaffee“ zu erhalten, wird empfohlen, nur *gemahlenen* Kaffe zu verwenden und die Heizquelle auf die *kleinste* Stufe zu stellen.

Systemsimulation

Der Codec macht mittlerweile (hoffentlich) was *Sie* wollen und die Audiodaten stehen innerhalb des FPGA an einer einfachen parallelen Schnittstelle zur Verfügung. Die Umgebung für die praktischen Tests von Modulen zur digitalen Signalverarbeitung steht also bereit.

Abb. 1 zeigt eine Unit zur digitalen Signalverarbeitung. Am Eingang liegt ein Strom von Audiodaten (*dry*, also unverarbeitet) an, der in der Unit verarbeitet wird. Das Ergebnis (*wet*, also verarbeitet) steht am Ausgang zur Verfügung. Sowohl Ein- als auch Ausgang dieser einfachen Unit verarbeiten Monosignale. Es sind zwar auch Units denkbar, welche Stereosignale verarbeiten, dies erhöht jedoch die Komplexität der Signalverarbeitungsaufgaben beträchtlich.

Eine mögliche Vorgehensweise bei der Entwicklung dieser Signalverarbeitungsmodule wäre nun, die einzelnen Module in VHDL zu beschreiben und auf dem Prototypingboard auszuprobieren. Eine entsprechende Struktur zeigt Abb. 2. Diese Vorgehensweise wird bei (*sehr*) einfachen Modulen

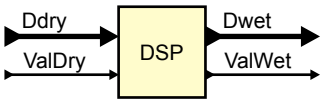


Abbildung 1: Unit zur digitalen Signalverarbeitung

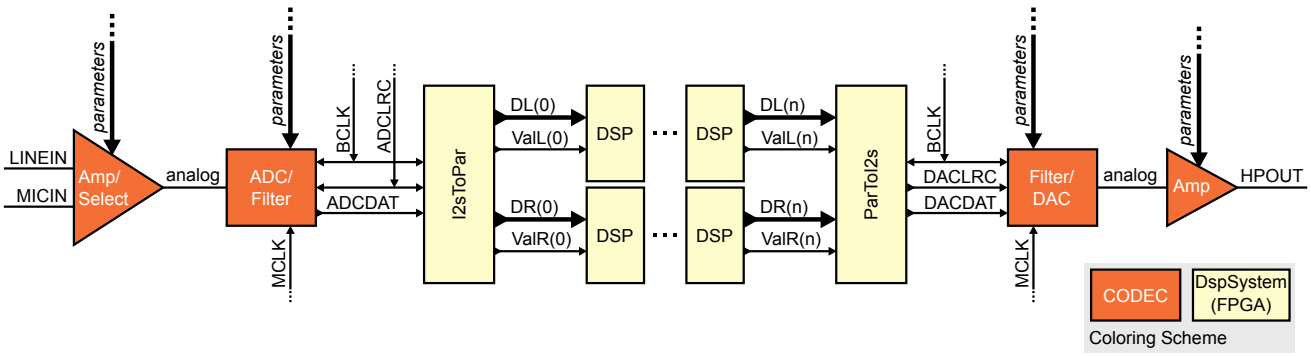


Abbildung 2: DSP-Gesamtsystem auf dem Prototypingboard. Es wird aus einer beliebige Anzahl von Blöcken zur digitalen Signalverarbeitung („*DSP*“) aufgebaut. Die Verarbeitung der Audiodaten erfolgt getrennt für den linken und rechten Kanal des Stereosignals.

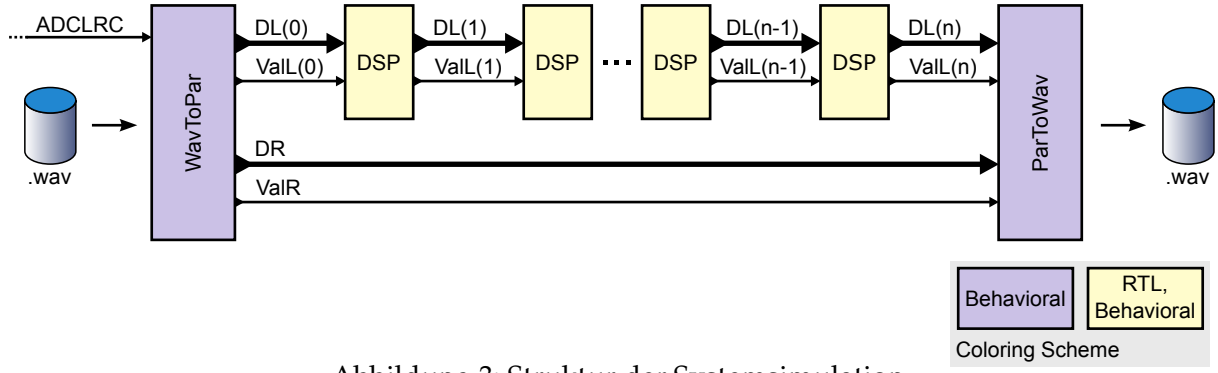


Abbildung 3: Struktur der Systemsimulation

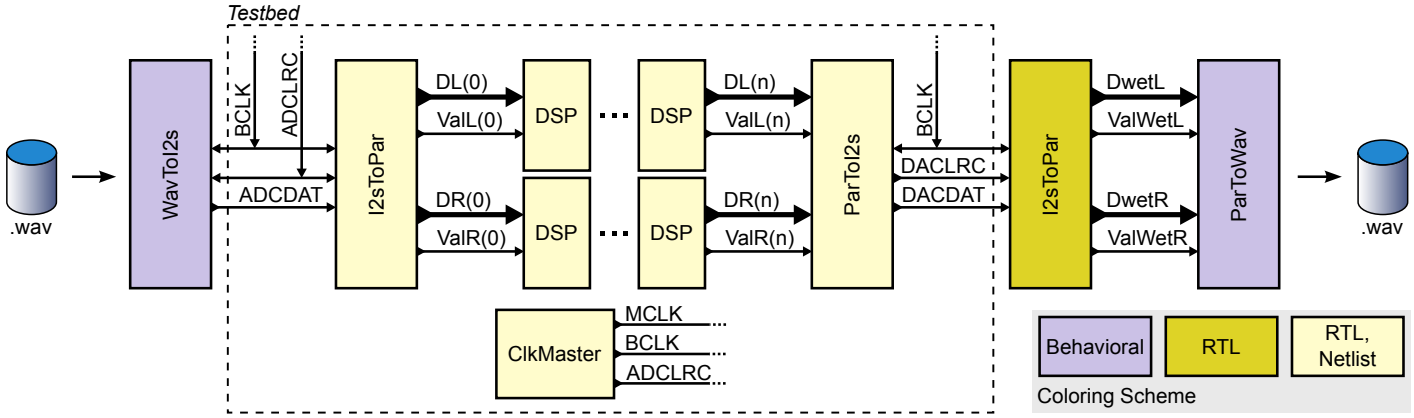


Abbildung 4: Simulationsumgebung für das komplette System, so wie es auf dem Rapid Prototyping Board umgesetzt wird. Neben den Units `WavToPar` und `ParToWav` steht auch die Unit `WavToI2s` fertig zur Verfügung.

wahrscheinlich sogar zum Erfolg führen. Aber selbst ein einfaches FIR-Filter stellt bereits einen zu komplexen Entwurf für diese Vorgehensweise dar. Eine Simulationsmöglichkeit würde die Fehlersuche daher erheblich erleichtern.

Betrachtet man die Arbeit auf der Systemebene, wo mit Matlab oder Simulink bereits sehr früh simuliert werden kann, kommt automatisch der Wunsch auf, diese Arbeitsweise auch im VHDL-Entwurf einsetzen zu können. Das hauptsächliche Problem ist hierbei die Ein- und Ausgabe der Audiodaten.

In Matlab werden hierzu Dateien des `.wav`-Formats eingesetzt. Auch VHDL bietet eine Möglichkeit zum Lesen und Schreiben von Dateien. Es müssten also lediglich eine Unit zum Lesen und eine zum Schreiben einer `.wav`-Datei erstellt werden. Die Struktur für eine komplette Systemsimulation könnte damit nach Abb. 3 aufgebaut werden. Auch das Testbed (d.h. das Gesamtsystem mit der Schnittstelle des FPGA auf dem Rapid Prototyping Board) oder eine daraus generierte Netzliste zur Post-Synthesis- bzw. Post-Layout-Simulation lassen sich auf diesem Weg in eine Simulationsumgebung einbetten. Eine mögliche Struktur zeigt Abb. 4. Die Schnittstelle zur Testbench entspricht in diesem Fall exakt der Schnittstelle des FPGA am Board.

Da der Umstieg von der einen auf die andere Umgebung an sich bereits ein erhebliches Problem darstellen kann, sollte zunächst immer versucht werden, die Simulation, welche in Matlab bereits einwandfrei funktioniert, in VHDL mit geringstem Aufwand (also möglichst ähnlich) nachzustellen. Es geht also *keinesfalls* darum, gleich ein synthesefähiges Modell zu erstellen. Stattdessen eignen sich Modelle, welche regen Gebrauch von sequentieller Abarbeitung und Subprograms machen, am besten für diesen Schritt.

Die Ergebnisse dieser Art der Systemsimulation lassen sich, weil als `.wav`-Datei vorhanden, leicht mit Matlab oder einem Wave-Editor analysieren und hörbar machen.

1. Aufgabe *Das IEEE-Package* `fixed_pkg`

Bei Verwendung des Typs `real` in der VHDL-Beschreibung kann mit der Struktur aus Abb. 3 auf einer hohen Abstraktionsebene simuliert werden. Es ergibt sich sogar eine Überschneidung mit dem Abstraktionsbereich von Matlab.

Bei Anwendung des Typs `real` ist in Hinsicht auf dessen Genauigkeit jedoch Vorsicht geboten. Das *Language Reference Manual* für die Sprachversionen 1076-1987 und 1076-1993 verlangt lediglich eine Genauigkeit von zumindest 6 dezimalen Nachkommastellen. Im Vergleich zur Darstellung in Matlab sind Fließkommazahlen in VHDL also äußerst grobe Gesellen. Erst mit der Sprachversion 1076-2002 sind die Anforderungen, die in VHDL an Fließkommazahlen gestellt werden, auf die Höhe der Genauigkeit von Matlab angehoben worden. Es werden nun 64-Bit-Fließkommazahlen verwendet. Da die Anforderungen der älteren Sprachversionen nur *Mindestanforderungen* waren, verwenden die meisten aktuellen Werkzeuge das 64-Bit-Format aber ohnehin für alle VHDL-Beschreibungen.

Der sinnvollste Punkt zum Umstieg von der Matlab- in die VHDL-Simulation liegt dennoch eher bei der Anwendung von Festkommazahlen (*fixed point*). Für diese Zahlendarstellung gibt es in Matlab/Simulink eine Toolbox, mit der die Auswirkungen von Genauigkeit und Wertebereich des verwendeten Festkommaformats beurteilt werden können.

In VHDL stellt das Package `fixed_pkg` den Typ `u_sfixed` mit zahlreichen Operatoren (unter anderem `"*"`, `"+"` und `"-"`) zur Verfügung. Basis ist der Typ `signed` aus dem Package `numeric_std`. Im Unterschied zu diesem erstreckt sich der Indexbereich einer Zahl vom Typ `u_sfixed` jedoch außer auf den positiven auch auf den negativen Ganzzahlbereich.

Festkommazahlen vom Typ `u_sfixed` lassen sich demensprechend sowohl hinsichtlich des Wertebereiches (Stellen vor dem Dezimalpunkt) als auch der Genauigkeit (Stellen nach dem Dezimalpunkt) an die Erfordernisse der Anwendung anpassen. Der positive Indexbereich (einschließlich 0) gibt dabei die Vorkommastellen (Integer-Bits) und der negative Indexbereich die Nachkommastellen (Factional-Bits) an. Der Datentyp `u_sfixed(N downto -M)` hat daher $N+1$ Vorkommastellen und M Nachkommastellen. Der Datentyp `u_sfixed(0 downto -8)` entspricht also dem 1.8-Festkommaformat (1 Vorkomma- und 8 Nachkommastellen); der Datentyp `u_sfixed(3 downto -5)` dem 4.5-Festkommaformat.

Hinweis: Das Package `fixed_pkg` gehört seit Version 1076-2008 zum VHDL-Standard. Es ist in der Library `IEEE` enthalten. Bedenken Sie aber, dass VHDL-2008 zum Teil leider noch immer nur in begrenztem Ausmaß in die Simulations- und Synthesewerkzeuge integriert ist. Speziell in den kostenlosen Versionen der Werkzeuge ist oft nur eine sehr eingeschränkte Unterstützung vorhanden.

Viele aktuelle Simulations- und Synthesewerkzeuge liefern jedoch auch für VHDL-93 und VHDL-2002 eine Version des Package `fixed_pkg` mit. In diesem Fall ist das Package in der Library `IEEE_proposed` enthalten. Sie können es daher mit `use ieee_proposed.fixed_pkg.all` in Ihre Beschreibungen einbinden.

Zusätzlich wurde mit der ersten Übung eine Version des Package `fixed_pkg` mitgeliefert, die Sie ebenso verwenden können.

a☐ Machen Sie sich mit dem Package `fixed_pkg` vertraut und analysieren Sie, wie der Datentyp `u_sfixed` bereits in Ihr Design integriert ist. Beachten Sie dabei wie das mitgelieferte Package mit fhlow in eine separate Library `ieee_proposed` eingebunden wird (`append ExtraLibraries`, `append Packages`). Sehen Sie sich zudem insbesondere auch die Definitionen von `aAudioData` und `fract_real` im Package `Global`, sowie die Datentypen des Par-Protokolls (z.B. anhand der Ports der Units `I2sToPar` und `ParToI2s`) an.

b☐₂ Welches Zahlenformat (Vor- und Nachkommastellen, Wertebereich) wird in Ihrem Design verwendet?

c☐₂ Was hat es mit dem Wertebereich des Datentyps `fract_real` auf sich? In welchem Zusammenhang steht dieser mit dem Format der Audiosamples im Par-Protokoll?

2. Aufgabe *WavToPar: Einlesen von Wave-Dateien*

Die Unit `WavToPar` liest eine Datei des Typs `.wav` ein und gibt die darin enthaltenen Audiodaten synchron zum Frame Clock `ADClrc` des Gesamtsystems aus. Dieses Signal bestimmt damit die Abtastfrequenz.

Die Unit selbst steht bereits fertig zur Verfügung. Diese Aufgabe dient hauptsächlich zum Kennenlernen des Dateiformats einer `.wav`-Datei und als Beispiel des Lesens einer Datei in der VHDL-Simulation. Hierzu bietet das Kapitel 18 in ASHENDEN genaue Informationen.

Das Format einer `.wav`-Datei kann zwar überaus kompliziert sein, jedoch wird zumeist nur ein kleiner Ausschnitt der geradezu chaotischen Vielfalt dieses Formats genutzt. Dieser Ausschnitt wird im Dokument `WaveFileFormatByStanfordCourse422.pdf` (`literature/AudioFileFormats/`) komprimiert wiedergegeben. In `WaveFileFormatByJGlatt.htm` findet sich eine genauere und allgemeinere Darstellung.

a☐ Simulieren Sie die Testbench der Unit `WavToPar` schrittweise und verfolgen Sie dabei, wie die Datei Stück für Stück eingelesen wird. Achten Sie insbesondere auf die Stellen, an denen die Datei geöffnet und gelesen wird. Vergleichen Sie hierbei mit dem Format der `.wav`-Dateien.

b☐ Stellen Sie den Dateinamen auf eine andere Datei ein und probieren Sie, ob das gewünschte Ergebnis erhalten wird.

c☐$_1$ Wie verhält sich die Unit `WavToPar` bei Dateien im Mono- oder Stereo-Format, wie bei Dateien im 8-Bit-Format?

d☐$_2$ Die Abtastrate für die Audiodaten ist in der Wave-Datei angegeben. Welchen Einfluss hat diese Angabe auf die Abtastrate, mit der die Daten in der Simulation auftreten?

Tipp: Untersuchen Sie die Unit nach Stellen, an denen die Abtastrate eingestellt wird (z.B. Daten aus der `.wav`-Datei, Generics) und wie die Audiodaten dann tatsächlich hinaus „getaktet“ werden. Was fällt Ihnen dabei auf?

3. Aufgabe *ParToWav: Ausgabe in Wave-Dateien*

Zur Vervollständigung der Umgebung für die Systemsimulation fehlt noch die Unit `ParToWav`, welche einen Audiodatenstrom wieder in einer `.wav`-Datei ablegt.

Die grundsätzliche Vorgehensweise ähnelt derjenigen in der Unit `ParToI2s`: Sobald die Gültigkeit der Audiodaten des linken Kanals angezeigt wird, werden dessen Daten von der parallelen Audioschnittstelle gelesen und in der Datei abgelegt. Eine Möglichkeit zum Prüfen der Gültigkeit wäre:

```
1  wait until iValL = cActivated and iClk'event and iClk = '1';
```

Damit ist das Gültigkeitssignal für den linken Kanal automatisch die Synchronisationsquelle für das Schreiben der Audiodaten. Die Daten des rechten Kanals werden jeweils unmittelbar nach den Daten des linken Kanals in die Wave-Datei geschrieben. Gültig sind diese Daten aber nur, wenn dies über den Port `iValR` angezeigt wird.

In der Beschreibung kann man nach Ankunft der Daten des linken Kanals nicht auf die Gültigkeit der Daten des rechten Kanals warten, da unter Umständen beide Kanäle zugleich gültig sein können.

a☐$_2$ Wie könnte es zu dieser Situation kommen?

Eine mögliche Lösung dieses Problems liegt darin, die Daten des rechten Kanals jeweils zu deren Gültigkeit abzuspeichern und den gespeicherten Wert immer zum Zeitpunkt der Gültigkeit der Daten des linken Kanals in die Wave-Datei zu schreiben. Die Unit `ParToI2s` setzt eine ähnliche Lösung ein.

Ein weiterer Stolperstein beim Schreiben von Wave-Dateien stellt deren prinzipieller Aufbau aus *Chunks* dar. Jeder Chunk enthält bereits zu Beginn die explizite Angabe seiner eigenen Länge. Dies ist vor dem Hintergrund der binären Kodierung des Dateiinhaltes verständlich. Wird eine solche Datei geschrieben, so muss *vorab* die Länge jedes Chunks bekannt sein. Dies ist in der Systemsimu-

lation jedoch aus unterschiedlichen Gründen nicht der Fall. Selbst wenn die Eingabedaten aus einer Wave-Datei stammen, deren Länge ohne weiteres auslesbar ist, gibt es Fälle, welche eine Bestimmung der Länge der zu schreibenden Datei schwierig machen. Ein Dirac-Impuls als Eingangssignal wird beispielsweise eine sehr kurze Dateilänge ergeben, während die Antwort des Signalverarbeitungssystems über einen relativ langen Zeitraum interessant sein kann.

Es ist also notwendig, alle Audiodaten, welche in die Ergebnisdatei geschrieben werden sollen, zwischenzuspeichern und den Inhalt dieses Zwischenspeichers erst am Ende der Simulation in eine Wave-Datei zu übertragen. Als Zwischenspeicher kommt entweder eine *Variable* oder ein *File* in Frage. Letzteres ist die bei weitem günstigere Variante, da die Variable enormen Speicherplatz im Arbeitsspeicher des Simulationsrechners belegt. Dieser Speicher ist aber bereits vom Simulator selbst stark belastet. Außerdem ist das Ablegen von Datensequenzen in einem File einfacher realisierbar, da keine Indizierung der Daten mitgeführt werden muss.

Da das Ende des Eingangsdatenstroms, wie oben schon dargestellt, nicht zugleich das Ende des Ausgangsdatenstroms bedeuten muss, stellt sich die Frage, wie dieses Ende in der Simulation überhaupt erkannt werden kann. Hierzu sollen zwei Möglichkeiten berücksichtigt werden:

a) Die Wave-Datei mit den Eingangsdaten ist komplett ausgelesen und es ist nach deren Ende zusätzlich ein mittels Generic einstellbarer Zeitraum vergangen.

b) Die Simulation wird von der Benutzerin oder dem Benutzer vorzeitig abgebrochen. Dennoch soll eine gültige Wave-Datei mit den bisher erzeugten Ergebnisdaten erstellt werden.

Die Unit `WavToPar` zeigt das Ende der Eingabedatei über den Port `oWaveAtEnd` an. Dieser ist vom Typ `boolean` und nimmt nach dem Ende der Eingangsdaten den Wert `true` an. Von diesem Zeitpunkt an sendet diese Unit Audiodaten, welche Abtastwerte für vollkommene Stille darstellen.

Die Unit `ParToWav` hat einen entsprechenden Eingang `iWavAtEnd`, welcher das Ende der eigentlichen Eingangsdaten der Simulation anzeigt. Gemessen vom Zeitpunkt zu dem dieser Eingang von `false` nach `true` wechselt, werden noch für die Dauer eines Zeitraums, der mit dem Generic `gRecordingDurationAftWaveEnd` definiert ist, Audiodaten aufgezeichnet. Nachdem auch dieser Zeitraum verstrichen ist, wird die Ausgabedatei mit Hilfe der Daten aus dem Zwischenspeicher erstellt und die Simulation wird mittels eines Assertion-Statements mit der Severity `failure` abgebrochen.

Soll die Simulation vorzeitig beendet werden, beispielsweise für den Fall einer sehr langen Eingabedatei, so kann dem Port `iWavAtEnd` im Simulator einfach der Wert `true` aufgezwungen werden (`force -freeze...`). Danach wird noch der Simulationszeitraum mit der Dauer `gRecordingDurationAftWaveEnd` vergehen und im Anschluss eine gültige Wave-Datei erstellt.

b☐ Die Unit `ParToWav` steht zusammen mit passender Testbench fertig zur Verfügung. Weisen Sie die Funktionsfähigkeit der Beschreibung in der Simulation nach.

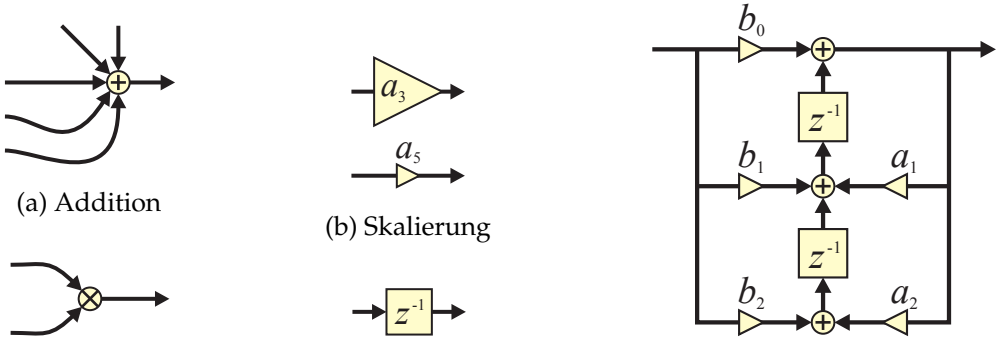
c☐ Brechen Sie die Simulation probehalber auf dem oben gezeigten Weg frühzeitig ab und stellen Sie sicher, dass die Ergebnisdatei eine gültige Wave-Datei ist.

4. Aufgabe *Die Unit DspScaleAndAdd*

Nun steht eine Umgebung zur Realisierung beliebiger signalverarbeitender Systeme zur Verfügung. Zunächst sollen einige Grundbausteine zur digitalen Signalverarbeitung implementiert und in dieser Umgebung getestet werden. Abschließend wird ein komplettes System realisiert.

Die graphische Nomenklatur der Grundelemente der digitalen Signalverarbeitung ist nicht genormt. In Abb. 5 findet sich eine Darstellung der im Rahmen der Übung verwendeten Symbolik.

Die Unit `DspScaleAndAdd` dient zur skalierten Addition zweier Signale aus unterschiedlichen Quellen (siehe Abb. 6). Es werden also zunächst die Eingangswerte mit jeweils einem konstanten Faktor multipliziert und anschließend die beiden Werte addiert.



(a) Addition (b) Skalierung (c) Multiplikation (d) Verzögerung (e) Beispiel: IIR-Block 2. Ordnung

Abbildung 5: Graphische Nomenklatur der Grundelemente der digitalen Signalverarbeitung.

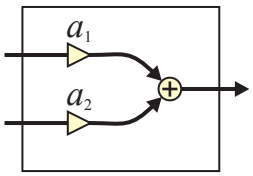


Abbildung 6: Die Struktur der Unit `DspScaleAndAdd`.

Bei der Skalierung wird ein Signal (Typ `u_sfixed`) mit einem konstanten Faktor Typ `fract_real` bzw. `real`) multipliziert. Das Package `ieee_proposed.fixed_pkg` definiert dazu den „*“-Operator:

```
function "*" (L : u_sfixed; R : real) return u_sfixed;
```

Dieser wandelt den `real`-Wert `R` in einen `u_sfixed`-Wert mit der selben Range wie `L` um. Anschließend werden die beiden `u_sfixed`-Werte miteinander multipliziert.

Die Multiplikation zweier `u_sfixed`-Werte mit den Ranges `(a downto b)` und `(x downto y)` liefert einen `u_sfixed`-Wert mit der Range `(a+x+1 downto b+y)`. Für die Multiplikation mit einem `real`-Wert ergibt sich daher ein Ergebnis mit der Range `(2*L'left+1 downto 2*L'right)`.

Bei Verwendung von Fraktionalzahlen der Form `u_sfixed(0 downto -N)` hat das Multiplikationsergebnis also die Form `u_sfixed(1 downto -2*N)`. Dieses Ergebnis muss nun mit einer geeigneten Resize-Operation wieder in den ursprünglichen Wertebereich (`u_sfixed(0 downto -N)`) der verwendeten Fraktionalzahlen gebracht werden, um in nachfolgenden Rechenschritten mit dem ursprünglichen Wertebereich weiterrechnen zu können.

Die einfachste Möglichkeit für dieses Resize ist das Abschneiden der überzähligen Bits:

```
function scale (L : u_sfixed; R : fract_real) return u_sfixed is
  variable result : u_sfixed(2*L'left+1 downto 2*L'right);
begin
  result := L * R;
  return result(L'range);
end function scale;
```

Dabei ergeben sich allerdings einige Probleme:

a) ***Überlaufverhalten:*** Für `sfixed(0 downto -(B-1))` ergibt sich als kleinste Zahl -1 ($'1.000'_b$, wenn $B = 4$ angenommen wird). Multipliziert man -1 mit -1 so ergibt das $+1$ ($'01.000000'_b$). Schneidet man nun die linke Seite dieser Zahl wieder auf den ursprünglichen Wertebereich ab ergibt sich daraus $'1.000000'_b$. Diese Zahl stellt aber -1 dar. Es kommt also zu einem Zweierkomplementüberlauf.

Insbesondere bei Systemen mit Rückkopplung können solche Überläufe gravierende Auswir-

kungen haben (z.B. große Grenzzyklen bei IIR-Filtern). Die Resize-Operation muss daher sicherstellen, dass statt diesem einfachen Überlaufverhalten eine Sättigungsfunktion angewendet wird. Das bedeutet, dass das Ergebnis auf den *nächst*möglichen Wert gebracht wird, hier also $'0.111111'_b$ ($= 0.984375$).

b) ***Quantisierungsverhalten:*** Auch auf der rechten Seite (Nachkommastellen) müssen Stellen abgeschnitten werden um auf den ursprünglichen Wertebereich zu gelangen. Aus $'0.111111'_b$ ($= 0.984375$) wird beispielsweise $'0.111'_b$ ($= 0.875$). Die Zahl verliert also an Genauigkeit.

Durch das Abschneiden der überzähligen Nachkommastellen wird immer der nächst*kleinere* Wert herangezogen. So ergibt beispielsweise $'1.000111'_b$ ($= -0.890625$) beim Abschneiden den Wert $'1.000'_b$ ($= -1$). Positive Zahlen werden also zum betragsmäßig kleineren Wert hin abgeschnitten, negative Zahlen hingegen zum betragsmäßig größeren Wert.

Eine alternative Möglichkeit wäre es, den Wert zum nächstgelegenen Wert hin zu runden. Dadurch wird der Quantisierungsfehler symmetrisch um den Wert verteilt und damit der betragsmäßige Fehler reduziert. Jedoch wird sowohl beim einfachen Abschneiden als auch beim Runden u.U. der Absolutbetrag des quantisierten Werts erhöht. Dadurch wird dem System zusätzliche Energie zugeführt, die bei Systemen mit Rückkopplung leicht zu Problemen führen kann (z.B. kleine Grenzzyklen bei IIR-Filtern).

Dieses Problem lässt sich durch ***Betragskorrektur***, also das Abrunden zum betragsmäßig kleineren Wert (auch „round-to-zero“ genannt) lösen.

Das Package `ieee_proposed.fixed_pkg` stellt bereits eine `resize`-Funktion bereit, auf die Sie im Rahmen dieser Aufgabe zurückgreifen können:

```
function resize (
  arg                    : u_sfixed;   -- input
  size_res               : u_sfixed;   -- for size only
  constant overflow_style : fixed_overflow_style_type := fixed_overflow_style;
  constant round_style    : fixed_round_style_type    := fixed_round_style)
  return u_sfixed;
```

Dabei gibt `overflow_style` das Überlaufverhalten und `round_style` das Quantisierungsverhalten an.

a☐1 Ermitteln[1] Sie, welche Einstellungsmöglichkeiten das Package `ieee_proposed.fixed_pkg` für die Parameter `overflow_style` und `round_style` hat und welche Einstellungen standardmäßig verwendet werden.

b☐2 Skizzieren Sie für jede dieser Möglichkeiten die Quantisierungs- bzw. Überlaufkennlinie. Skizzieren Sie auch die Quantisierungskennlinie für die Betragskorrektur. Ist dieses Verhalten mit der `resize`-Funktion realisierbar?

c☐3 Implementieren Sie die Resize-Operation mit Betragskorrektur und Sättigung als synthesefähige (!) Funktion `ResizeTruncAbsVal` in Ihrem Package `Global`. Greifen Sie dabei auf die Funktion `resize` zurück und schneiden Sie den Eingangswert damit zunächst auf den nächstkleineren Wert ab.

```
function ResizeTruncAbsVal (
  arg      : u_sfixed;  -- input
  size_res : u_sfixed)  -- for size only
  return sfixed;
```

d☐4 Erstellen Sie eine VHDL-Beschreibung für die Unit `DspScaleAndAdd`. Die Entity dieser Unit sieht folgendermaßen aus:

```
entity DspScaleAndAdd is
  generic (
    gAudioBitWidth     : natural                := cDefaultAudioBitWidth;
```

[1] Beispielsweise anhand der Dokumentation und des Sourcecodes des Package `ieee_proposed.fixed_pkg`.

```
    gScaleFactor       : aSetOfFactors(1 to 2) := (0.1, 0.3);
    gRegisteredOutputs : boolean               := false);
  port (
    -- Sequential logic inside this unit
    iClk         : in std_ulogic;
    inResetAsync : in std_ulogic;
    -- Input audio channels
    iDdry1, iDdry2 : in aAudioData(0 downto -(gAudioBitWidth-1));
    -- Input channels have to be valid at the same time;
    -- only one port needed to flag validity.
    iValDry        : in std_ulogic;
    -- Output audio channel
    oDwet    : out aAudioData(0 downto -(gAudioBitWidth-1));
    oValWet : out std_ulogic);
end DspScaleAndAdd;
```

Diese Beschreibung greift auf die folgende Deklaration innerhalb des Package `Global` zurück:

```
-- A set of factors, e.g. the set of filter coefficients for a cascade.
type aSetOfFactors is array (natural range <>) of fract_real;
```

Wenn diese Unit Bestandteil eines großen Systems werden soll, so ist es günstig, alle Ausgangssignale über Register zu führen. Unter Umständen kann es so aber auch zu unnötig vielen Registern kommen. Sie sollen daher den Einbau von Registern über den Generic `gRegisteredOutputs` steuerbar machen.

Sie können davon ausgehen, dass die beiden Eingangskanäle stets genau gleichzeitig gültige Daten liefern. Um dies sicherzustellen verwenden Sie die bereitgestellte Unit `DspSyncChannels`. In der signaltheoretischen Interpretation verschiebt diese Einheit die Phase des rechten Kanals soweit, dass die Abtastzeitpunkte beider Kanäle exakt zusammen fallen.

Linker und rechter Kanal werden also zunächst durch `DspSyncChannels` miteinander synchronisiert. Das Ergebnis dieses Schrittes dient als Eingabe von `DspScaleAndAdd`. Ausgabe dieses Schrittes ist ein einziger Kanal, der auf den linken Ausgabekanal des Gesamtsystems gelegt werden soll. Der rechte Kanal des Gesamtsystems wird unverändert vom Ein- auf den Ausgang durchgeschleift und kann zum Vergleich herangezogen werden.

e□1 Weisen Sie die Funktionsfähigkeit Ihres Entwurfs in der VHDL-Systemsimulation nach. Eine dazu geeignete Wave-Datei ist `SinL220HzR263Hz1sec.wav`.

Tipp: In QuestaSim (bzw. ebenso in ModelSim) lassen sich Audiosignale im *Waveform-Window* sehr gut im Analog-Format darstellen. Das Fenster in Abb. 7 erreicht man mit einem Rechtsklick auf den Signalnamen und Auswahl von *Format* → *Analog*...

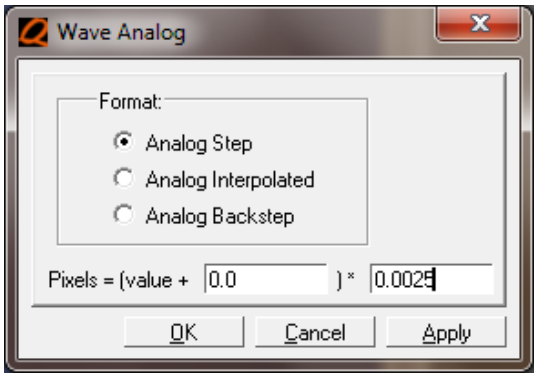


Abbildung 7: Fenster zur Parametrierung des Analog-Formats in QuestaSim/ModelSim.

f☐₂ Ist der Aufwand für die Skalierung abhängig von der Konstante, mit der multipliziert wird? Zeichnen Sie zunächst Ihre Einschätzung auf und verifizieren Sie anschließend mittels mehrerer Synthesedurchläufe.

Tipp: Überlegen Sie sich, wie verschiedene Skalierungsfaktoren (z.B. 0; 0,2; 0,25; 0,5; 0,55; 0,9; 1; diese Werte wurden nicht zufällig gewählt!) in Hardware realisiert werden.

g☐ Es steht eine Unit `TbdDspScaleAndAdd` zur Verfügung, welche das Testbed für `DspScaleAndAdd` darstellt. Für dieses Testbed existiert eine Testbench. Passen Sie dieses Testbed, falls notwendig, an Ihre Beschreibung an.

h☐₁ Führen Sie nun eine Simulation zur Validierung des Testbeds durch. Machen Sie sich bitte klar, was Sie eigentlich mit dieser Simulation zeigen: Die Schnittstelle des Testbeds entspricht exakt der Schnittstelle des FPGA auf dem Rapid Prototyping System. Vergleichen Sie hierbei auch mit der Darstellung in Abb. 4. Sie zeigen also mit dieser Simulation, dass Ihr Entwurf auch in seiner späteren Umgebung funktionieren sollte. Von den Problemen, welche durch die Umsetzung in der Synthese und den Einfluss von Signallaufzeiten hinzukommen können, bleibt Ihre Simulation derzeit natürlich noch unbeeinflusst.

i☐₁ Realisieren Sie den Entwurf auf dem Rapid Prototyping Board. Prüfen Sie die korrekte Funktion sowohl gehörmäßig, als auch mit dem Audiotester. Eine dazu geeignete Wave-Datei ist `SinL440Hz-R446Hz20sec.wav`.

Abgabe des Sourcecodes

Neben den regulären ***Commits*** im Laufe der Arbeit an jedem Übungszettel ist der Endstand jedes Übungszettels als Tag im Repository abzulegen. Jeder Tag hat dabei einen Namen der Form „ue-<Nr>“ zu tragen, wobei <Nr> die Nummer des jeweiligen Übungszettels ist.

a☐ Erstellen Sie bis spätestens ***23:59 Uhr am Vortag des Abgabetags*** (siehe *Semesterübersicht*) einen Tag `/tags/ue-04` mit dem Endstand Ihrer Übungsausarbeitung.