System-Level Design Exploration of an FM Radio Receiver

Michael Wurm

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Embedded Systems Design

in Hagenberg

im August 2021

Advisor:

DI (FH) Dr. Florian Eibensteiner

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, August 23, 2021

Michael Wurm

Contents

Abstract

This should be a 1-page (maximum) summary of your work in English.

Kurzfassung

An dieser Stelle steht eine Zusammenfassung der Arbeit, Umfang max. 1 Seite. ...

Chapter 1

Introduction

1.1 Motivation

In the master's degree program Embedded Systems Design, a wide range of topics relate to digital signal processing, chip design and software development. All of these disciplines find their application in an embedded system. Multiple different methods of implementation are introduced for each topic and their usage is tested in practical sessions. However, in the setting of a university course it is not always possible to look at the currently handled topic as a part of a larger, integrated system. Instead, it is often treated as a standalone part that fulfills its own, specific functionality. Therefore, one of the main motivation points for this thesis is to combine several disciplines and multiple implementation methods into a single system design, to find a universal workflow for design and verification, and treat them in this common project context.

In a more general perspective, the current electronics market situation is a huge motivational factor as well. Smart devices, smart sensors, Internet of Things (IoT) and many similar terms are well-known and are becoming more and more present in the public's daily lives. To some degree, all of these devices represent embedded systems, since they all consist of processing units, combined with sensors and interfaces to communicate with their surrounding world. In the future, the number of devices and the range of applications in this area is going to grow from today's point of view. This will require a lot of engineering work to advance the state of technology we have today, to push the limits and explore new areas of applications.

Since the development of an embedded system combines so many displines, it is of advantage to have knowledge of as many as possible of them. However, not only understanding single parts, but to have an overview and understanding of the entire, integrated system and the interaction between those single parts is important in an embedded system.

Those major reasons lead to the choice of topic for this thesis – having the competence for all the single parts by themselves, while maintaining the understanding over the integrated embedded system that fulfills a task as a whole.

1.2 Choice of Application

In order to elaborate on the theoretical topics of this thesis, a comprehensive practical project is developed along with it. The project of choice is a digital FM broadcast radio receiver. It combines several disciplines that appear in the master's program Embedded Systems Design, such as digital signal processing, software and hardware development, as well as system design and therefore is a perfect candidate for this project.

FM broadcast radio is chosen as the input signal, because it is a common radio frequency signal that is available almost everywhere. Furthermore, the demodulated signal is an audio signal, usually music or speech, that can be listened to. This is subjectively more attractive than a generic data stream. Additionally, it is a comparably simple signal to demodulate, but still requires several techniques of signal processing to achieve that.

1.3 Objective

The main objective of this thesis is to elaborate a system design for an embedded system, that combines all the knowledge of different disciplines and topics which are accumulated over the time of the university program, in a single project. This includes the development of a system model in a tool like Matlab, the practical usage and implementation of a digital signal processing (DSP) chain in an FPGA, the development of a digital receiver and the deployment on actual hardware.

The receiver implementation on the FPGA is to be done in two variants. The first variant is traditional, manually written hardware description language (HDL) in the programming language VHDL. For the second variant, high-level synthesis (HLS) is employed, which generates the low-level HDL from high-level C++ code. These two methods are implemented and compared on the basis of usability and some metrics.

In the view of a system design approach, another focus is put on a sustainable testing and development strategy for the entire system. Above mentioned implementation variants should ideally share as much as possible of testing and verification code, in order to save time and effort in implementation. Also, as many tasks as possible should be automated by scripts, to save time in manual labor for the engineers working on the development. All that enables an efficient development cycle of the product.

By the end of this thesis, the implementation of an FM radio receiver in multiple methods and abstraction levels should be achieved, as well as a good understanding of the overall system level design. Knowledge of each method's advantages and disadvantages, as well as efficiency and useability should be gained.

1.4 Outline

Chapter 2 describes the signal processing theory that is required to demodulate an FM broadcast radio signal. This includes an introduction to receiver types, the description of an FM signal generally and also specifically as it is used in FM audio broadcasting, as well as various general topics of signal processing. Chapter 3 covers the topic of a certain system design process, as well as system design tools that specifically target FPGAs. Chapter 4 is about high-level synthesis, which includes a general introduction to the state of the art, as well as a description of the workflow, inclusively the coding and testbench features and characteristics. Chapter 5 describes the chosen system architecture and concept for the project. The actual implementation, with a detailed explanation of some selected components, is covered in Chapter 6. Chapter 7 gives an overview of the required parts of development that are necessary to deploy the deeloped system on actual hardware. The comparison and achieved results are covered in Chapter 8. Chapter 9 concludes this thesis with a retrospective and gives an outlook of potential future extensions to the project.

Chapter 2

Signal Processing Theory

This chapter describes techniques of signal processing, which are required to receive and decode an FM broadcast radio signal. The entire signal processing chain from the antenna, through the analog front-end, to the digital processing parts is explained. However, the main focus is put on the digital signal processing part, since this is a major part of the practical implementation of this thesis.

2.1 Overview

Wireless transmission systems usually transmit their signals over the air. Various frequency bands and different types of modulation are being used for this purpose. The choice of frequency, as well as the choice of the modulation type, have impacts on the transmission characteristics. Depending on these choices, positive effects can be achieved in the amount of data that can be sent over the transmission channel, for example. Another advantage is the ability to adapt the ongoing transmission to the changing characteristics of the channel. This is especially useful for mobile receivers, since the signals' path is inherently changing over time, which leads to challenging conditions for the receiver. Additionally, varying signal strength, or multipath propagation may occur. Multipath propagation is caused by reflections of the signal, which, for example, can occur on a house facade, or a large vehicle that is passing by. The receiver then gets a signal that is a summation of the direct path and the reflected path – a signal from multiple paths. Besides that, environmental impacts, such as rain or fog have an effect on the transmission channel and thus the received signal.

Many more factors could be listed, and elaborations over the various modulation types fill entire books. However, with all these different facts, there are things that they all have in common. On the transmitter side, all these signals are transmitted in a band-limited, high frequency signal. The band limitation needs to be given at the transmitting antenna, to minimize interference to other bands in the frequency spectrum. The frequency needs to be high, to radiate the signal by an antenna. On the receiver side, once the signal reaches the antenna of a receiver, there are further similarities. The antenna signal needs to be amplified and filtered at first. Both steps depend on the antenna characteristics, but usually the received signal is of relatively low power,

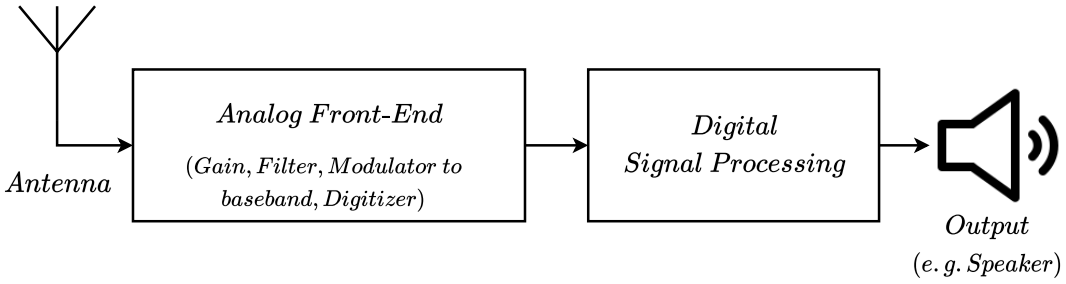


Figure 2.1: High-level block diagram of a signal processing chain.

and the antenna receives a wider range of frequencies than required. The filter therefor selects the frequency area that is of interest. Another reason for the filter is to avoid spectral replicas in the subsequent modulation, or down-conversion. Here, the front-end often down-converts to a low intermediate frequency (IF), instead of directly going to baseband. The intermediate frequency is then shifted down to baseband by a modulator later in the digital domain, which brings advantages in the downstream processing chain. These three steps – amplification, filtering and down-conversion to an IF – are often done in the analog domain, since digital circuits cannot work with such high frequencies. After the analog signal is down-converted to baseband, it needs to be digitized. This is done by an analog-digital-converter (ADC). After the signal is processed to digital baseband, the received signal can finally be decoded to get the actual data content. This decoding process depends on the respective modulation type that is used.

The block diagram in Figure 2.1 gives a high-level overview of a receiver structure.

2.2 Receiver Types

Multiple different receiver architectures exist and are implemented in receivers. Two common examples are explained in the following sections. The main source of information for this section is [20].

2.2.1 Superheterodyne Receiver

The superheterodyne receiver is often referred to as *superhet*, or *super* in the literature. The main characteristic of a superhet is that it converts the radio-frequency (RF) antenna signal to an IF in a first stage, instead of going to baseband directly.

The IF is usually set to a fixed frequency, which results in the advantage, that the subsequent IF filter can be optimized, since it does not have to be adapted in frequency. Therefore, a superhet receiver is able to select narrow band signals, even if the bands' surrounding frequencies are being used by other signals. This is one of the superhets' main advantages. A disadvantage is that the signal noise around the image frequency band is translated into the IF output band. Thus, the filter requirements are relatively

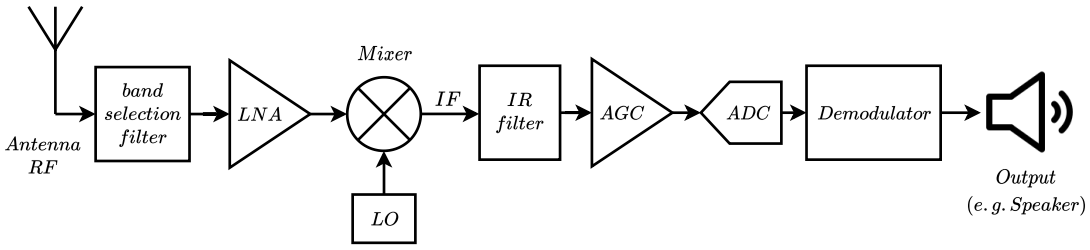


Figure 2.2: Block design of a superhetorodyne receiver [20]

high, since they need to deliver a strong attenuation in order to suppress noise outside the desired band.

Figure 2.2 shows a block diagram of a superhet receiver. The antenna signal first runs through a band selection filter. It selects the desired frequency band and filters out any unwanted signals outside of this band. A low-noise amplifier (LNA) may be used to boost the signal strenght. The next step is the mixer, which is fed by a local oscillator (LO). This shifts, or down-converts, the antenna signal down to the IF. Afterwards, the IF signal is amplified and filtered again to remove the side-products of the mixer, namely the image replicas. Thus, it is also called an image rejection filter. The demodulator stage can now perform any further signal processing on the signal, until the required signal is retrieved.

The depicted block diagram represents a superhet receiver structure, for example as it can be used in an FM broadcast receiver, where an audio output is generated from the antenna input.

2.2.2 Direct-Conversion Receiver

A direct-conversion receiver, as its name suggests, converts an RF signal directly to baseband. It can also be called homodyne, synchrodyne, or zero-IF receiver.

In order to achieve direct down-conversion, the local oscillator of the mixer needs to be set exactly to the center frequency of the desired signal. The main advantage of direct-conversion receivers is, that their output is an image-free signal. This is due to the properties of the mixer. The strongest disadvantages are the LO leakage, which leads to a DC offset at the mixer's output, and the flicker noise, also called $1/f$-noise, which induces strong noise at low frequencies. These effects are not explained into more detail here, since this is not the focus of this thesis. Figure 2.3 shows a block diagram of a direct-conversion receiver.

2.3 Down-Conversion

The process of down-conversion is a technique of amplitude modulation (AM). It is used to convert an RF signal to a signal of lower frequency, while preserving its information

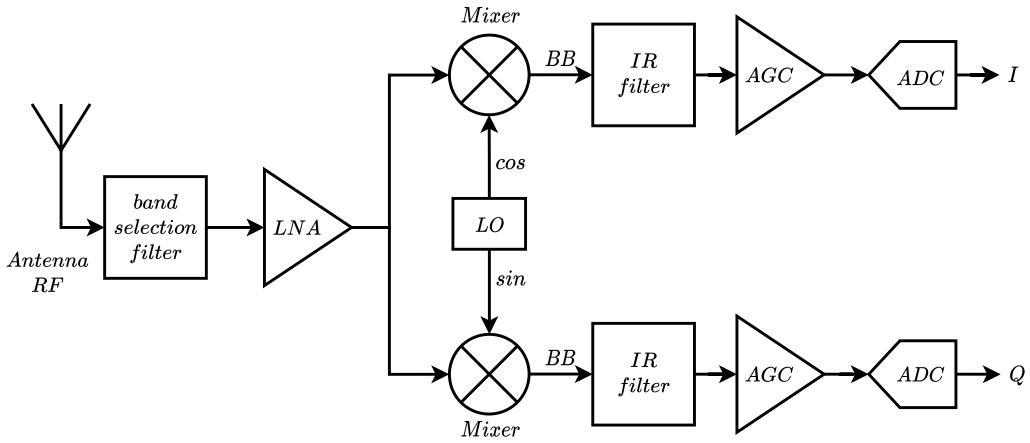


Figure 2.3: Block design of a direct-conversion receiver [20].

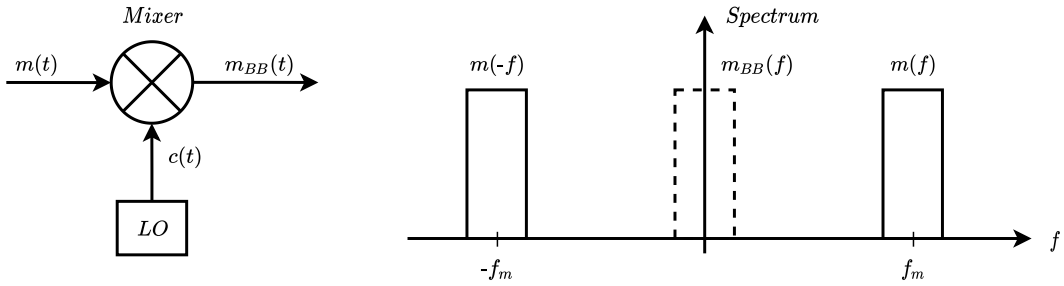


Figure 2.4: Block diagram and signal spectrum of down-conversion.

content. This is the explanation in the time domain. However, it may be easier look at the process in the frequency domain. In the frequency domain, the band-limited RF signal is shifted down to a lower frequency. This lower frequency can either be zero, i.e. baseband, or an intermediate frequency, such as in the application of a superheterodyne receiver. In order to achieve this, the incoming signal needs to be multiplicated with a carrier signal, that is generated locally in the receiver, by a local oscillator (LO). Depending on the frequency f_{LO} of this local carrier, the resulting signal will be shifted to baseband, or an IF. Note, that in the case of down-conversion, the local oscillator frequency f_{LO} is interchangeably denoted as carrier frequency f_c.

A block diagram, as well as the signal spectrum of the process is shown in Figure 2.4. In this example, the frequency of the LO is set to shift the signal from f_m, which is drawn in solid lines, to baseband, which is drawn in dashed lines.

2.3.1 Mathematical Description

The equation for a generic AM signal is $S_{AM}(t) = A_m m(t) \cdot A_c c(t)$, where $m(t)$ represents the message signal, and $c(t)$ is the high-frequency carrier. Their respective amplitudes are A_m and A_c. This equation is depicted as a block diagram and signal spectrum in

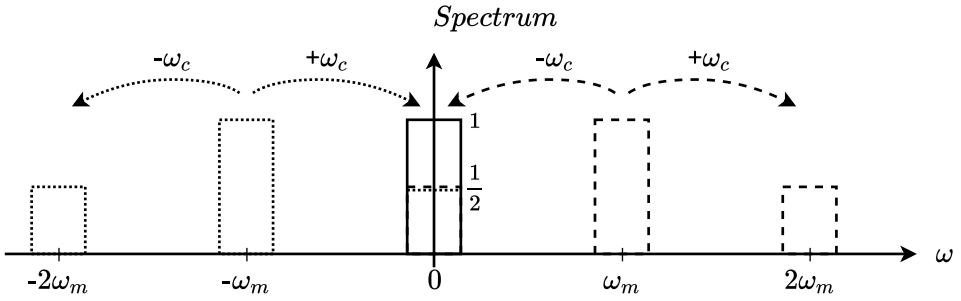


Figure 2.5: Signal spectrum, explaining the process of down-conversion.

Figure 2.4, in which S_{AM} is denoted as the baseband signal m_{BB}, because it is already down-converted. The original HF signal m, before the down-conversion, is a bandpass signal, such as an antenna signal that was filtered by a bandpass filter. This received signal is a real signal, and thus, its negative frequency spectrum is an exact mirror of the positive frequencies. This fact can be explained in two simple equations.

It is well-known that the real signal $cos(wt) = \frac{1}{2}\left(e^{jw} + e^{-jw}\right)$, which follows from $e^{jw} = cos(wt) + jsin(wt)$. Looking at the frequencies ω of the first equation, the negative and positive parts can be seen.

The process of down-conversion is described in the following formulas. The denotation of the subscripts is assuming a direct conversion to baseband.

$$m_{BB}(t) = A_m m(t) \cdot A_c c(t) \tag{2.1}$$

Assuming that $m(t) = cos(\omega_m t)$, $c(t) = cos(\omega_c t)$ and $A_m = A_c = 1$,

$$m_{BB}(t) = cos(\omega_m t) \cdot cos(\omega_c t) \tag{2.2}$$

follows. By applying the mathematical theorem

$$cos(\alpha) \cdot cos(\beta) = \frac{1}{2} \cdot \big[cos(\alpha - \beta) + cos(\alpha + \beta)\big] \tag{2.3}$$

the final formula of the conversion can be represented as

$$m_{BB}(t) = \frac{1}{2} \cdot \Big[cos\big((w_m - w_c)t\big) + cos\big((w_m + w_c)t\big)\Big] \tag{2.4}$$

The graphical representation in the frequency spectrum is shown in Figure 2.5. As mentioned above, the assumption is a direct conversion to baseband, so $w_m = w_c$. Looking at the positive original part, it is shown that only half of its energy is translated into the desired baseband. However, half of the energy of the original negative part is also shifted to baseband. The superposition of the two parts results in an amount of energy that is again equal to the original signal.

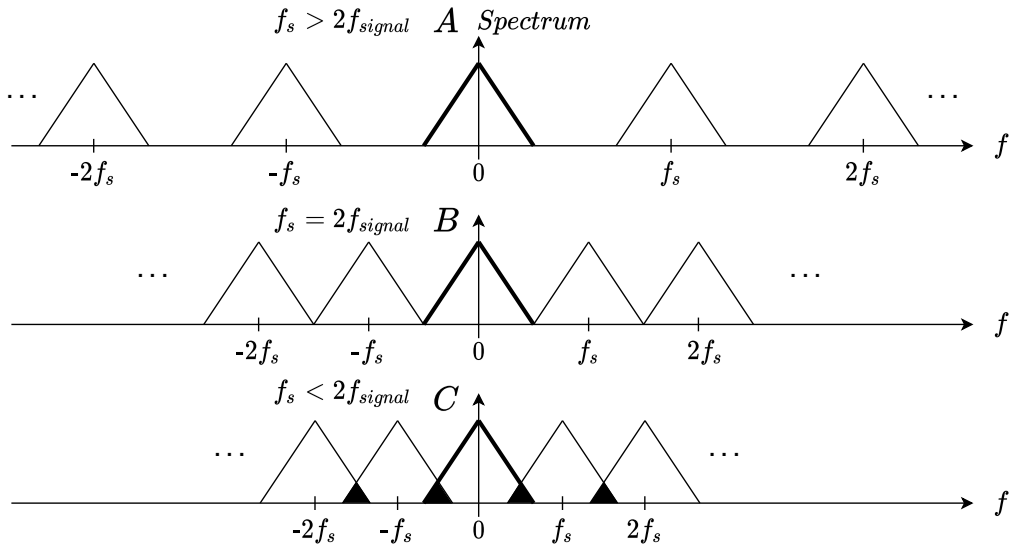


Figure 2.6: Frequency spectrum of the sampled baseband signal with its replicas at integer multiples of f_s [23].
A) $f_s > 2f_{signal}$: f_s is high enough, and thus, meets the theorem.
B) $f_s = 2f_{signal}$: f_s exactly meets the theorem.
C) $f_s < 2f_{signal}$: f_s is too low, and thus, creates overlaps of the replicas – aliasing.

2.4 Sampling Theorem

Signal processing is often done in the digital domain, using digital signals. A digital signal is obtained by sampling an analog signal in a periodic interval, the so-called sampling frequency f_s. The sampling frequency needs to meet the sampling theorem, in order to correctly represent the analog signal. The sampling theorem, by Nyquist and Shannon mainly consists of the following formula.

$$f_s > 2 \cdot f_{signal} \tag{2.5}$$

In words: The sampling frequency must be higher than twice the frequency of the signal's highest frequency content. Any digital signal processing technique must hold the specification of the sampling theorem [3, Ch. 4.2]. In case the theorem is violated, the effect of aliasing occurs on the generated signal. The sampling process introduces spectral replicas at integer multiples of the sampling frequency f_s. These replicas overlap, if f_s does not meet the theorem, which results in signal distortion. Figure 2.6 illustrates the effect of aliasing in the frequency spectrum [23].

2.5 Sample Rate Reduction

Sample rate reduction is also called downsampling. It is a way to reduce the amount of data that needs to be processed, without losing any of the signals' information. In some applications, the digital signal is oversampled by a large factor. However, if the frequency spectrum of the desired signal is a narrow band, a high oversampling rate

is not necessary and the signal can be downsampled without losing any information content. To guarantee this, the Nyquist sampling theorem must still be met after the downsampling process.

A lower sample rate is especially useful for efficient filter design. Factors such as the transition bandwidth or the width of the passband are important there. Short transition bandwidths, or a narrow passband require a high filter order to fulfill the specifications. This translates into a high computational effort. In such cases, it is of advantage to downsample the input, in order to create better conditions for the filter design. Downsampling spreads the spectrum of interest, in terms of the relative bandwidth between passband and sampling frequency. Therefore, the filter can be designed with wider transition bandwidth and passband, which significantly reduces the required filter order.

As an example, assuming the signal of interest has a bandwidth of 20 kHz. The Nyquist theorem thus requires a remaining sampling rate of at least twice the signal's frequency, so 40 kHz. The signal is sampled with an oversampling factor of 10 – meaning a sample rate of 200 kHz. A downsampling factor of 4 is now applied, which results in a sample rate of 50 kHz. The resulting signal is now downsampled and can be processed at this lower data rate, while none of the original data content is lost.

The process consists of two parts: filtering and skipping samples. Let us assume downsampling by a factor of $M = 4$. In the first step, a low-pass filter with a normalized cut-off frequency of $1/M$ is applied. This is required in order to avoid aliasing later in the process. The second step simply skips a number of M-1 samples in the sequence.

More detailed information can be found in the literature [3, Ch. 6.9], [13, Ch. 4.1] and [23, Ch. 10.2.2].

2.6 Frequency Modulation in Broadcasting

Frequency modulation (FM) is a widely used standard to transmit data streams. The probably best known usecase therefor is commercial broadcast radio, where an audio stream is transmitted. Devices to receive these streams are available for low prices to the public. This section describes the main properties of broadcast FM, such as the mathematical description, frequency bands that are used, or the specific frequency parts within a channel spectrum.

2.6.1 Mathematical description

The mathematical description of an FM baseband signal can be described with the formulas presented in this section. FM encodes the information content in its instantaneous frequency. This means that the measured frequency at any moment in time represents a specific value of a transmitted message. For a general classification, FM belongs to the group of angle-, or phase modulated signals (PM). The simple reason therefor is that a frequency has a direct relationship to an angle, if the signal is seen on a unit circle.

The instantaneous frequency f_i of an FM signal can be described as

$$f_i = f_c + \Delta f \cdot m(t) \tag{2.6}$$

where f_c is the carrier- or center frequency, Δf is the maximum frequency deviation and $m(t)$ is the information or message signal that is to be transmitted. Simply looking at this equation, the instantaneous frequency varies in the range between $f_c \pm \Delta f$, which is sometimes also called the swing. Considering the relationship between frequency and angle, as described above, and after some simple substitutions which will not be described into detail here, the equation for a generic frequency modulated wave is the following

$$s(t) = A_c \; cos\Big(2\pi f_c t + 2\pi k_f \int_0^t m(\tau) d\tau\Big), \tag{2.7}$$

where A_c is the amplitude of the resulting FM signal and k_f is the frequency sensitivity factor. This sensitivity factor has a direct relationship with the modulation index β.

$$\beta = \frac{\Delta f}{f_m} = \frac{k_f A_m}{f_m}, \tag{2.8}$$

where f_m is the highest existing frequency, or the bandwidth of the information signal. Equation (2.7) describes a frequency modulated signal with a generic message signal $m(t)$. A widely used example application of FM is broadcast radio, where audio streams are transmitted. An audio stream can be described as a cosine wave. Therefore, the information signal that is to be transmitted in broadcast radio can be represented as a cosine wave in the form of

$$m(t) = A_m \; cos(2\pi f_m t). \tag{2.9}$$

By inserting this message signal into the generic FM Equation (2.7), the final formula for an FM signal transforms into the following form.

$$y(t) = A_c \; cos\Big(2\pi f_c t + \beta \; sin(2\pi f_m t)\Big) \tag{2.10}$$

Several formulas, as well as their derivations are described according to [18, pg.54-55].

2.6.2 Frequency Band

The frequency band that is used for FM broadcasting is defined worldwide and spans from 87.5 to 108 MHz. However, some countries only partially use the band [2, RR5-39]. The range is located within the so-called Very-High-Frequency (VHF) band, which is open to the public in terms of usage. This means, that any transmitter may use the specified frequency range freely. Austria allowed the legal usage in 2006 [6]. However, the transmission power needs to be limited, so that neighboring receivers are not disturbed in receiving any existing channels. The European Commision specifies the power limit as 50 nW of effective radiated power (ERP) [7]. Another limitation is, that a transmitter must be capable to transmit on multiple center frequencies within the broadcasting band. It must not be fixed to a single center frequency. Through this specification, a transmitter is capable of switching its transmission center frequency, in order to avoid interference with another transmitters signal.

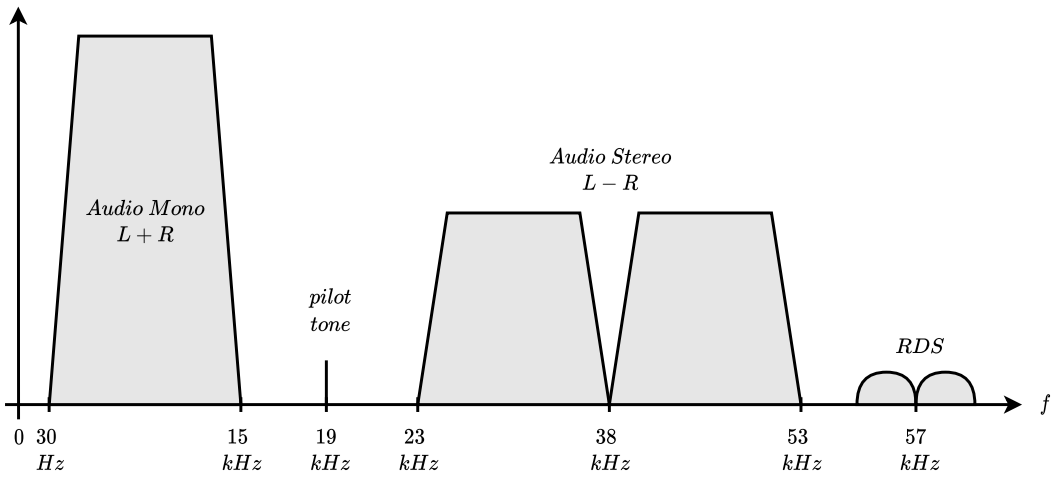


Figure 2.7: Allocation of frequencies in an FM channel [24].

In a practical usecase this means that a transmitter needs to select a channel or frequency range, that is not already occupied by an official transmitter. If a free range is found, the transmitter may start sending an FM signal with the mentioned power limitations [1].

2.6.3 Frequency Spectrum of a Broadcast FM Channel

The entire FM broadcasting band from 87.5 to 108 MHz divides into multiple channels which can be used. In Europe, the European Telecommunications Standards Institute (ETSI) sets the respective standards for the usage of these frequencies. The main specifications for a single FM broadcasting channel are explained in this section.

Each channel may allocate a bandwidth of 200 kHz. Within one of these channels, the maximum deviation from the center frequency shall not exceed 75 kHz. This leaves a guard band of 25 kHz on either side, to minimize interference with adjacent channels. Center frequencies may be allocated on multiples of 100 kHz. However, neighboring channels need to be considered therefore, since this may result in an overlap in the spectrum.

Figure 2.7 shows the frequency spectrum of a single channel, located at its carrier- or center frequency. For a better overview, only the positive spectum part is shown. The negative part is an exact mirror, because the transmitted and received signals from the antenna are real signals – they do not have an imaginary part. The spectrum consists of multiple parts. Therefore, it is also referred to as 'multiplex signal' in the literature. Several of these parts are described in the following paragraphs, according to [1],[16] and [33].

Mono Audio Part

The mono audio stream is located between 30 Hz and 15 kHz. This signal is built by the sum of the left and right audio channels. The upper limit at 15 kHz is chosen to maintain a sufficient spacing to the first subcarrier. For audio streaming, as in FM broadcasting, this is not a limitation, since it is unlikely to have frequencies higher than 15 kHz in an audio stream. Also, this already reaches the upper limit of the human ears' bandwidth, so higher spectral parts will not be heard anyway.

Pilot Tone

The first subcarrier is allocated at an offset of 19 kHz from the center frequency. This subcarrier is also called the pilot tone, since it is a continuous signal. It is independent of any varying message signal content. The pilot tone is used for stereo audio demodulation. To regenerate a stereo audio signal, the left and right audio channels need to be recovered correctly.

Stereo Audio Difference

A signal that is constructed by the difference between the left and right audio channel is modulated on a 38 kHz subcarrier. This subcarrier is an integer multiple of the pilot tone at 19 kHz, for practical reasons. However, the 38 kHz carrier is suppressed and thus not visible in the received spectrum. The modulation technique that is used for this spectral part is called dual-sideband suppressed-carrier (DSB-SC). Even though the carrier is suppressed, it can still be recovered, because it is phase coherent with the 19 kHz pilot tone per definition.

The bandwidth for this difference-signal spans 15 kHz on either side of the subcarrier. It is used to generate a stereo audio signal, in combination with the mono signal. This process is explained into more detail in Chapter 2.8.

Additional Services

Considering all these parts in the spectrum, there is still free bandwidth available to use up to the maximum channel bandwitdh of 100 kHz in one sideband. Because of that, additional services were added to the pure audio transmission, to provide additional data services and information.

Services that were implemented are the Data Radio Channel (DARC), which is mostly used in Japan and the USA, the Subsidary Communication Authorization (SCA) and the Radio Data System (RDS) [22]. Out of these, RDS is the most significant service in Europe. It is used to transmit additional information about the channel, such as the radio stations' names, the currently playing song's title or traffic information.

2.7 Algorithms for Digital FM Demodulation

An FM signal that is received, for example from an antenna, needs to be demodulated in order to decode its actual data content. In general, an FM demodulator serves the purpose to transform a frequency modulated signal (FM) into an amplitude modulated signal (AM), so that AM signal processing techniques can be applied afterwards.

The radio frequency antenna signal usually needs to be amplified, bandpass-filtered, and down-converted to baseband, using subsampling and quadrature-mixing, or similar strategies. This part of the signal processing chain is assumed to be working correctly and is out of the scope of this chapter. The FM signal that is evaluated here is assumed to be a quadrature-mixed signal in baseband, which means that inphase and quadrature (I/Q) signals are available. In the following sections, two digital FM demodulator variants are described.

2.7.1 Frequency Discriminator

A frequency discriminator generates an output that is directly proportional to the frequency of the input FM signal. In other words, it converts FM to AM. The discriminator consists of two main parts – a differentiator and an envelope detector. Different strategies can be implemented for both parts. A block diagram for a frequency discriminator with its two main parts is shown in Figure 2.8.

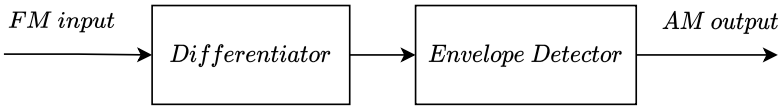


Figure 2.8: Frequency discriminator block diagram [25].

Differentiator

The differentiator already converts the FM signal into an AM signal, but still leaves an FM content. The transformation that happens here can be expressed in a formula, by simply differentiating Equation (2.7). This operation, with some additional simple algebraic transformations, results in

$$\frac{ds(t)}{dt} = A_c 2\pi \Big[f_c + k_f m(t) \Big] sin\Big(\omega_c t + 2\pi k_f \int_0^t m(\tau) d\tau - \pi \Big) \tag{2.11}$$

This formula shows AM and FM contents clearly. The FM part is described by the sine function, with its argument. The more important part is the AM part, which resembles the message signal. It is described by the term within the square brackets. A time-domain example for this signal is shown in the top left diagram in Figure 2.9.

An important requirement for the differentiator is, that the input signal is of a constant amplitude [21]. The transmitted signal is subject to random additive noise

on the transmission channel. The differentiator would deliver wrong results because of this noise, when substracting two consecutive samples, as described in Equation (2.14). To achieve a constant amplitude, the received complex baseband signal $s(t)$ can be normalized to a value of one.

$$s_{norm} = \frac{s}{|s|} = \frac{A(n)\ e^{j\phi_{\mathrm{FM}}(n)}}{|A(n)\ e^{j\phi_{\mathrm{FM}}(n)}|} = e^{j\phi_{\mathrm{FM}}(n)} \tag{2.12}$$

$$|e^{j\phi_{FM}(n)}| = 1 \tag{2.13}$$

In a hardware implementation, a differentiation is simply performed by substracting two consecutive samples, like

$$\frac{ds(t)}{dt} = s(t) - s(t - \Delta t), \tag{2.14}$$

where Δt is the inverse of the sample frequency.

Envelope Detector

The differentiated signal still consists of a FM part, as explained above, using Equation (2.11). In order to remove this high-frequeny part and extract the low-frequency envelope, the signal needs to be fed into an envelope detector. A block diagram is shown in Figure 2.8.

The method implemented in Figure 2.9 is one of the most simple ones and is usually referred to as 'Asynchronous Half-Wave Envelope Detector' [32]. It consists of a thresholding unit to remove negative values and a conventional lowpass filter. In analog implementations, the thresholding unit is a diode. The cutoff frequency of the lowpass needs to be adapted to the envelope signals' maximum frequency. It needs to be able to follow the signal, but also sufficiently smooth out the rectified signal.

Different methods may be implemented for the envelope detector. For example, an 'Asynchronous Full-Wave Envelope Detector' can be used. Therefor, the previous method only needs to swap the thresholding unit with an unit that calculates the absolut value. In that architecture, a full-wave rectification is performed on the signal, which significantly improves the envelope detection accuracy [32]. The envelope is followed more exactly, because of the higher power that is available in the signal, since the entire signal is taken into account, and not only the positive half, as in the previous half-wave method.

2.7.2 Phase-Locked Loop

A Phase-Locked Loop (PLL) can also be used to transform an FM signal to an AM signal. A PLL is a feedback loop that is usually used to generate an output signal that has a fixed phase reference to its input. In the case of FM demodulation, the loop filter in the feedback branch is configured to be able to follow the frequency variations on

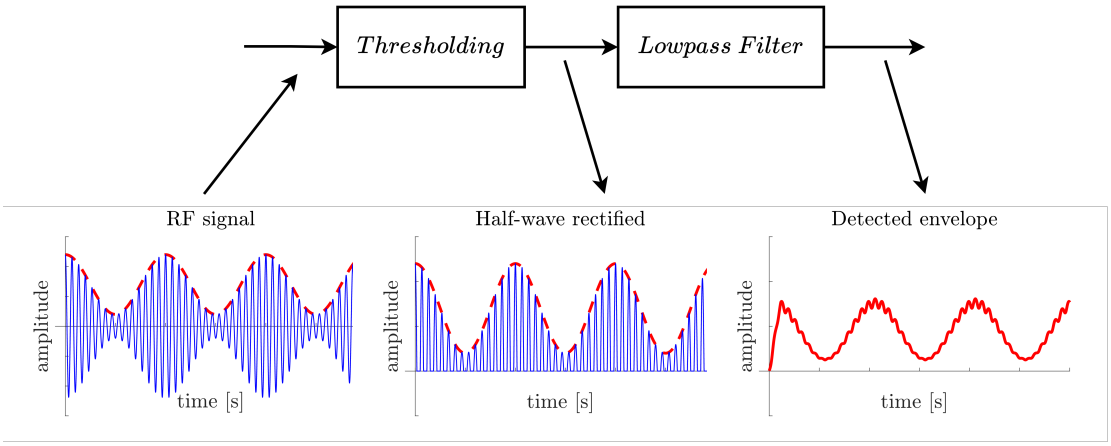


Figure 2.9: Time-domain signals in envelope detection [37].

the input, which is directly related to the encoded information. The output of the PLL directly delivers an AM signal, which corresponds to the transmitted information. [21]

FM demodulation with a PLL is not described into more detail, since this thesis is focussing on the concept of an entire system architecture.

2.8 Demodulation of Broadcast FM

This section describes the demodulation of the information content of an FM broadcast channel, as it is used in commercial FM radio broadcasting. The assumption here is, that the actual FM demodulation, as described in Chapter 2.7 is already done correctly. So at this point, the FM signal is converted to an AM signal. Thus, traditional AM DSP techniques can be employed to decode the FM broadcast channel.

A frequency spectrum of the multiplex signal of an FM channel is shown in Figure 2.7.

2.8.1 Mono Audio

The first part of the channel frequency spectrum is the mono audio signal. It is the summation signal of the left and right audio channel. Thus, it contains enough information to replay an audio stream in mono quality. A mono receiver can thus simply replay the so-called multiplex signal that is produced by the FM demodulator, without any further filtering or processing. The 19 kHz pilot tone, and any higher frequency components, will not be audible by most people, because it is outside of the range of a human ear. Besides that, any used speaker may also be unable to produce a frequency in this range.

Consequently, in order to demodulate this spectral part, a lowpass filter is required. The filter needs to have a cutoff frequency of 15 kHz, to allow the entire mono audio spectrum to pass. Sufficient attenutation must be available before 19 kHz, in order to suppress the pilot tone.

2.8.2 Stereo Audio

The demodulation of an FM channel to recover a stereo audio signal, requires a more sophisticated approach and multiple steps in signal processing. Two parts need to be combined in order to achieve stereo. The sum and the difference of the left and right audio channel signals – the mono and the stereo difference part, respectively. The following equations need to be applied, to compute a left and a right channel.

$$\begin{aligned}(L+R)+(L-R)&=(2)L\\(L+R)-(L-R)&=(2)R\end{aligned} \tag{2.15}$$

where $(L+R)$ represents the mono part and $(L-R)$ the stereo difference.

The block diagram in Figure 2.10 illustrates the signal processing chain for stereo audio. In the block diagram, $x(t)$ represents the multiplex signal from the FM demodulator. The signals $x_L(t)$ and $x_R(t)$ describe the left and right audio signal, respectively.

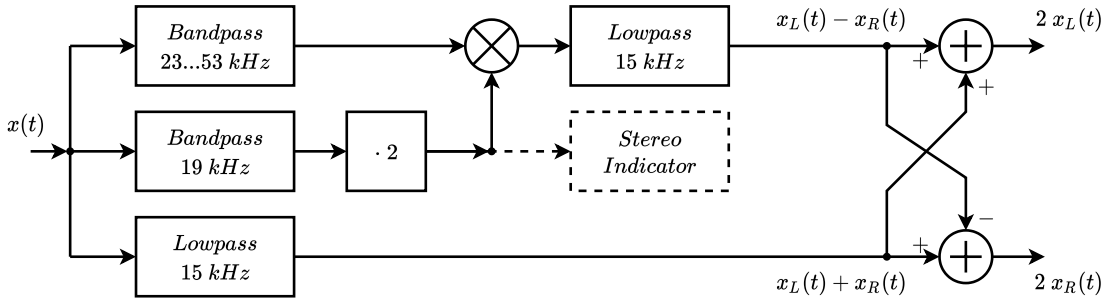


Figure 2.10: Block diagram of FM stereo audio demodulation [37].

The central branch selects the pilot tone at 19 kHz with a bandpass filter. This bandpass needs to have a frequency response that is sharp enough to have a sufficient attenuation at ±4 kHz, since this is where the mono and difference signals' frequency spectra are allocated. Afterwards, the pilot tone frequency is doubled to achieve a 38 kHz subcarrier. This guarantees a phase-coherency between pilot and subcarrier, which is required to correctly shift the difference signal down to baseband.

The frequency doubling can be implemented with multiple methods. The pilot tone can be modulated by simply multiplying it with itself. Another option is to use a PLL and configure it to generate the double frequency at its output. The PLL lock indicator can then be re-used as an indicator, whether a pilot tone exists.

In the upper branch, a bandpass filter selects the difference signal with ranges from 23 to 53 kHz. It is then modulated to baseband by the previously generated 38 kHz subcarrier. Subsequently, a lowpass limits signal bandwidth to 15 kHz, which removes the generated modulation artifacts. The lower branch lowpass-filters the summation, or mono signal, as described in Section 2.8.1. The rightmost part in the diagram performs the combination of the mono and the difference signal, according to Equation (2.15).

Chapter 3

System Design Process

The process of system design is a central topic in the development of any product. This chapter describes a general technique for system design, which can however also be directly applied to FPGA design. Additionally, tools that are specifically used in FPGA design are introduced.

3.1 General

The following sections describe a generalized approach to system design. A process called SIMILAR is explained, as well as selected topics that are relevant. The researched information for this section is mainly based on [4].

3.1.1 The SIMILAR Process

The SIMILAR process provides a general approach to problem solving. It can be applied to any kind of problem that can be of technical-, as well as non-technical nature, in order to successfully achieve the specified requirements. The concept was developed by A.T. Bahill and B. Gissing, who evaluated and compared multiple existing processes to find their similarities.

The acronym SIMILAR consists of the following components.

- **S**tate the Problem
- **I**nvestigate Alternatives
- **M**odel the System
- **I**ntegrate Components
- **L**aunch the System
- **A**ssess Performance
- **R**e-evaluate

Each of these terms represents a separate step in the process diagram shown in Figure 3.1. The steps can be seen as iterative and parallel, as the process runs during the product development cycle.

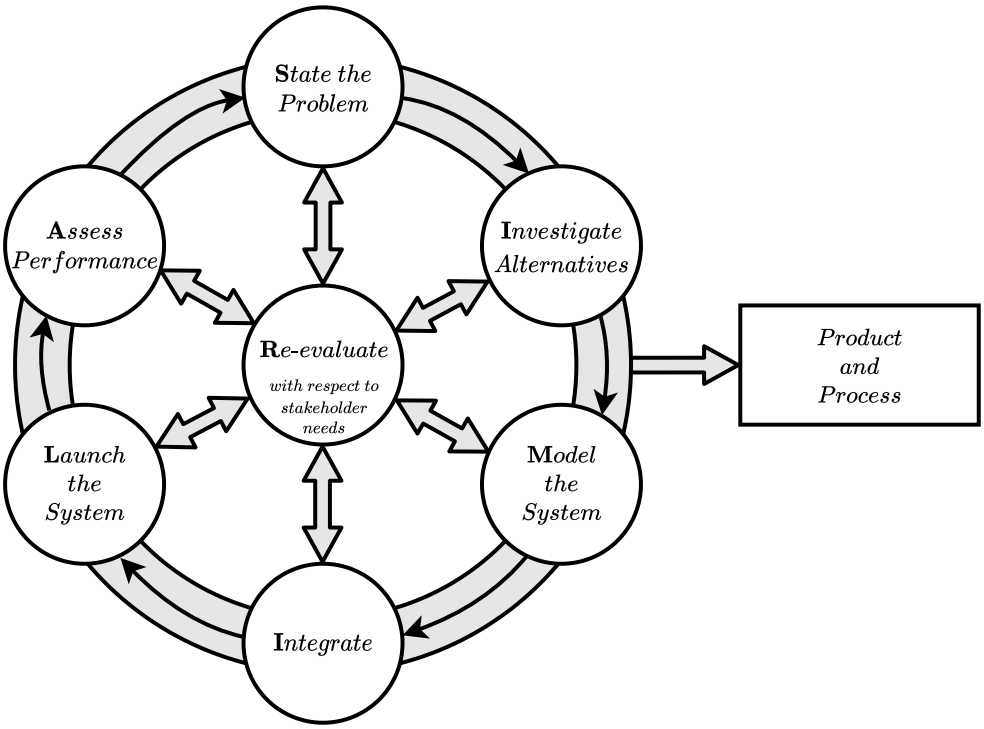


Figure 3.1: The SIMILAR process in a graphical representation [4].

State the Problem

The very first step of the process is to state the problem that is to be solved. This can be the initial definition of a new product, but can also be the definition of an improvement that is to be made for an existing product. Because of that, input for the problem statement can be generated by various parties, such as customers, product developers, regulatory agencies or manufacturers, just to name a few. In the process diagram, all these possible input parties are labeled as stakeholders.

Generally speaking, the problem statement serves the purpose to build a common understanding of the problem between the stakeholders and the developers. In case there is a mismatch, an engineer may spend valuable time to develop a sophisticated solution to a problem that was not even a requirement. Thus, any time spent to develop this feature is worthless, which highlights the importance of a correct problem statement.

An important guideline for the specification of this problem statement is, that each requirement is descibed in the way of *what* needs to be done, not *how* it needs to be done.

The problem statement is updated and improved by the Re-evaluate step, which is described later.

Investigate Alternatives

In almost any kind of problem, there are multiple ways in order to achieve a successful solution. However, some solutions may be preferred over others because of certain criteria that are evaluated. Such criteria may be cost, risk, schedule or performance. Based on the combined evaluation of these factors, a decision in favor of one selected alternative is made. This part of the process is also called concept exploration.

It is important to note that this step is a repeated, iterative step. This is necessary, since there may only be a few known facts available at the initial intestigation. Further design steps, such as a first estimation of the intended design implementation, simulation results, or a prototype can provide more data and facts. These can be used to re-evaluate the alternatives and subsequently optimize the design process.

Model the System

Based on the previous steps, a model for the system is developed here. This includes a process model, that is used for strategic planning towards cost reduction, optimization of the implementation effort, or the schedule of subprojects, and a product model, which supports trade-offs and the identification of risks, or helps to explain the system as a whole.

Typically, a coarse model is created for almost all alternative designs that are found in the previous step. A more detailed model is only developed for the single, selected design. The model itself may be represented in many different ways, such as state machines, block diagrams, flow diagrams, simulations or object-oriented diagrams.

Integrate Components

This step combines multiple single components into a larger, integrated system, which can then be treated as a whole. In order to be able to do that, interfaces between the components and subsystems need to be defined. Ideally, these interfaces are designed in a way that the amount of data that needs to be exchanged is kept to a minimum. Therefor, it is of advantage, if a subsystem only transfers finished products that it produced to another subsystem. The aim to develop simple interfaces is of special advantage in systems that include a feedback loop. There, managing the feedback loop connections is easier if there is a clear interface, so the loop can be connected around an entire subsystem, instead of interconnecting within the subsystems.

Launch the System

The system launch actually runs the system in a defined environment, so that it can perform what it is intended and designed to do. The defined environment can be the final, operational environment, or just a simulation which provides an output that is detailed enough for an evaluation. In a product that involves hardware development, this step may include the installation of commercial off-the-shelf products with a modified prototype software. However, since the SIMILAR process is applicable for any kind of

problem, this might also be a business process, where the launch step may be the roll-out of the subprocesses and tasks in the respective departments of a company.

Assess Performance

This step is all about measurements. The performance of a system can only be evaluated, if it can be measured by using different metrics. Evaluation criteria are used, based on the measurements of the technical performance, which can help to find trade-offs or to mitigate risk in design and manufacturing.

Re-evaluate

The re-evaluate step is a central building block in the SIMILAR concept. All of the other steps have a connection to it, in the form of a feedback loop. However, depending on the nature of the project and the point in the timeline of its life cycle, not all of these loops may be exercised all the time. The idea of re-evaluation and the subsequent improvement of a process is a widely applied and well-known concept. Its main function is to observe an output and to use this information to improve the system's input.

3.1.2 Design for Reuse

A design principle that is often used is design for reuse. It enables a faster and more efficient development of products, which leads to an overall decrease in development cost. Additionally, the product is likely less error-prone, since the reused parts of the system are already tested at this point. After all these positive arguments, there is a counter argument to it, though: designing a system with reuseable subsystems comes with cost. Additional effort is required to design the subsystems in a way so that they can be reused in another project at all. However, this additional effort may potentially already be paid off, when a single subsystem can be reused in a future product and thus, it is usually worth the effort.

3.1.3 Validation and Verification

The terms validation and verification are closely related, but have a different meaning in system design. Verification tests the properties of the developed product against the previously determined specification and requirements. It strives to build the *system right*. Validation on the other hand tries to build the *right system*. It proves that the product does what it is intended to do. This is important in order to develop the product according to what the customer actually needs. To avoid to divert from these customer needs, validation is done throughout the product's development cycle, from the beginning to the end. Various artifacts can be collected during the process, in order to have a traceable documentation of the decision that are made.

Validation may also fail, which means that the chosen system may be the wrong system for the respective customers needs. As an example, the system may be very sensitive to variations of a parameter that a customer cannot influence. Another example is, when a system is more sensitive to certain parameters than to its inputs. In both

cases, the system may be the wrong system for the specific problem and should urgently be re-evaluated.

In order to expose such failures as early as possible, any test engineer should be instructed to always keep an eye on the process of validation, which means to check if the team is building the *right system*. Therefore, a test engineer should be familiar with the problem statement.

3.2 System Design Targeting FPGAs

A wide range of system-level design approaches, which are targeting FPGAs, are available. In this section, some selected approaches are described. The research for this section is mainly based on [12].

3.2.1 V-Modell

The development of a product requires an organized approach, in order to successfully achieve the requirements. The common term for such an organized approach is the process model, which represents a guideline on how to organize and structure the development cycle.

A well-known process model is the so-called V-Model. It is based on the similarly well-known Waterfall Model, and has its name because of the distinct V-shape of its process diagram, as it is shown in Figure 3.2. One of the main characteristics of the V-Model is, that one phase needs to be completed before the following phase can start. This classifies the model as a static process model and is one of the main disadvantages at the same time, because it makes the model inflexible to adaptions of the requirements. The advantages however are that it is easy to install and apply, each phase has its specific deliverables, good results can be achieved for large projects with simple requirements, and the early test phases.

The disadvantages of the inflexibility regarding changes of requirements, and the fact that it is difficult to already consider all eventualities that may occur in the creation of the V-model, led to the innovation of modern process models, which are iterative and thus more flexible.

The V-model applies to general software development, or project development, but can directly be mapped to FPGA development. The model is covered in this thesis section, because it shows the clear testing structure of module tests (unit tests), integration tests and system tests, which are especially important in projects that target hardware, such as FPGA projects.

Furthermore, the V-model diagram in Figure 3.2 shows the difference between validation and verification, as it is explained in the previous Section 3.1.3.

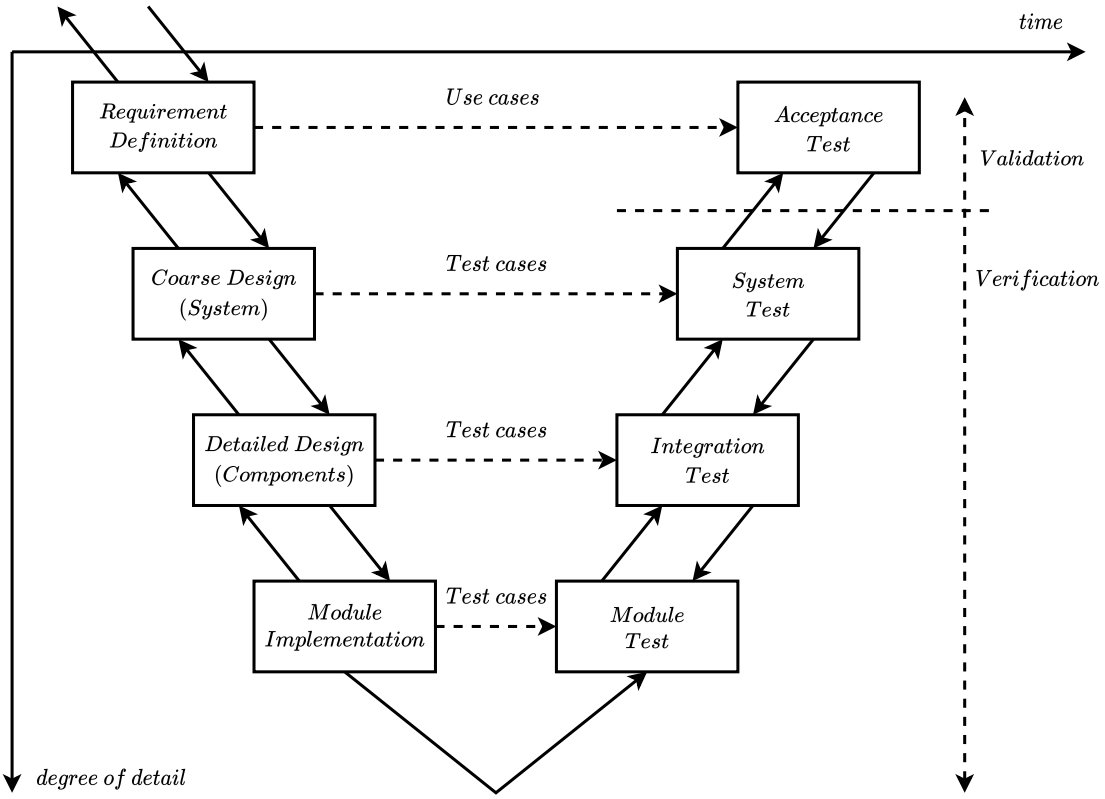


Figure 3.2: The V-Modell process diagram [12, Abbildung 3.5].

3.2.2 High-Level Synthesis

High-Level Synthes (HLS) is an FPGA design methodology that provides a high level of abstraction from the low-level hardware, and thus can be seen as a system level design method. Different languages are used for HLS – two well-known languages are SystemC, which is a library for C++ to provide modelling and simulation features, and plain C++. A comprehensive list of available HLS tools and languages can be found in Section 3.3 of [9].

The main idea of HLS is, that the hardware design can be written in abstract modules, which represent the actual functional meaning of a feature or algorithm, rather than the complex, low-level hardware details. HLS achieves this by allowing to develop the code in a high-level language, such as C++. This high-level code is then automatically transformed into hardware description language (HDL) by the respective HLS tool.

In this thesis, only the HLS of plain C++ is covered. SystemC is not discussed, since it may potentially be less important in the future. In the latest Xilinx HLS tools *Vitis HLS* and *Vivado HLS*, the language SystemC is already deprecated [31].

HLS is described into more detail in Chapter 4.

3.2.3 Direct Implementation

The implementation of low-level HDL in the register transfer level (RTL) is denoted as *direct*, or *manual* implementation in this thesis.

Two commonly used languages therefor are VHDL and Verilog. Both have similar features to describe hardware in the signal level, with parallel structures and timing-dependent architectures. The languages can describe a design in a hierarchical order, so that the separate modules can be constructed in a way that an overview of the functionality of the system remains given. Synthesis, as well as simulation is supported, which allows an efficient development of RTL designs.

3.2.4 Implementation with Tool Support

The implementation of FPGA designs with tool support focuses on the development as a model-based approach, that is often featured by a graphical programmers interface. This is often used for prototyping in an early phase of product development. A well-known example for such a model-based approach is Matlab and its graphical tool Simulink.

Matlab can be used to describe and develop hardware-oriented algorithms, since it supports various library functions, such as the fixed point representation of numbers. A major advantage is, that fast prototyping of an algorithm, as well as a fast simulation of the system is achievable. Simulink is a graphical editor, that allows to connect functional blocks in a block diagram and create a system in that way. Multiple toolboxes support additional blocks, that abstract complex functionalities, so that fast prototyping is possible here as well.

A powerful feature of Matlab and Simulink is, that it can be used similar to an HLS tool. Matlab code, as well as Simulink block diagrams can be transformed into low-level RTL code for an FPGA design.

Simulink System Generator

A full system-level design approach can be covered with the Simulink System Generator. This includes the high-level development of the system as a block diagram in Simulink, the automated generation of low-level RTL code for the FPGA hardware, and the verification in simulation and in hardware, in the form of hardware-in-the-loop. This workflow is supported for Xilinx FPGAs and is specifically targeting system design engineers, who work in rapid prototyping to generate hardware utilization estimates for the final product's hardware.

Figure 3.3 shows this entire design approach in a block diagram. The abbreviation ISE stands for Integrated Software Environment, and is Xilinx' development environment that includes the FPGA synthesis tool. ModelSim is a HDL simulator software, that is developed by Mentor Graphics.

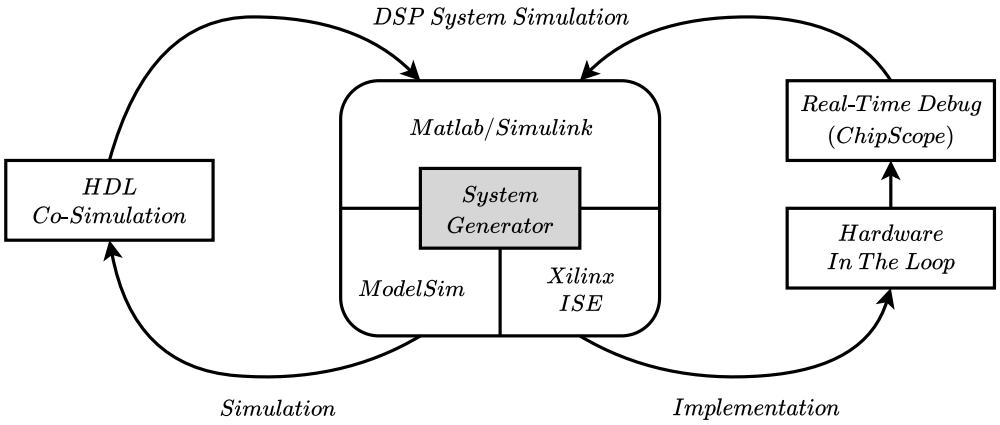


Figure 3.3: Capabilities of the Simulink System Generator [12, Abbildung 5.11].

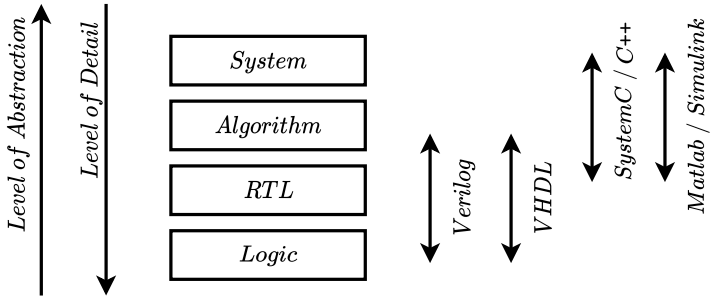


Figure 3.4: Abstraction levels of the implementation methods [12, Abbildung 5.1].

Simulink HDL Coder

The Simulink HDL Coder tool can generate VHDL or Verilog code from Matlab functions or Simulink block diagrams and state diagrams. The generated code can be used in Xilinx, as well as Intel/Altera FPGA hardware. This allows prototyping of algorithms, while continuously having detailed information of the target implementation, such as an estimate of the hardware logic utilization, or even the critical path in the logic.

3.2.5 Level of Abstraction

Multiple implementation methods are introduced in the previous sections, which all have different levels of abstraction. VHDL and Verilog require, or allow, a very detailed description of the RTL code, while SystemC, C++, or Matlab and Simulink provide a high level of abstraction. In Figure 3.4, the various implementation methods are compared. The respective level of abstraction is higher towards the upper end, while the level of detail increases towards the bottom of the figure.

In conclusion, it is an engineer's choice which implementation method to use to develop a product. While decisions and trade-offs may not always be obvious, they should consistently be based on the fundamental requirements of the final product.

Chapter 4

High-Level Synthesis

This chapter covers the High-Level Synthesis of C++ in general, but also describes details that specifically apply to HLS tools that are provided by Xilinx. An overview of the most important steps in the development workflow, as well as main features and possibilities of the code implementation and testbench are given. The main sources of information for this chapter are found in [14] and [15].

4.1 Introduction

In software development, engineers are used to writing code in a high-level language, such as C or C++. With these languages, specific hardware details are abstracted and thus, very little knowledge of the target hardware is required. The compiler takes over these details as it transforms the code by using the underlying, supported instruction set of the target hardware. This has been the standard for many years by now. In the earlier times however, it was common practise to develop programs in the low-level assembly language. The software engineer had to implement the software by directly using the supported instruction set of the target CPU.

A direct comparison can be drawn to FPGA design. An FPGA engineer historically develops code in the register-transfer level (RTL), which is common practise until today. This can be compared to the development of software in the assembly language. However, the recent efforts towards automation and code generation brought up High-Level Synthesis (HLS). There, the FPGA design can be described in a high-level language, such as C or C++, which is convenient and allows to focus on the algorithm, rather than hardware details. This high-level code is then transformed into the low-level RTL automatically by the HLS tool.

The usage of HLS tries to bring software- and hardware development closer together and tries to make these two worlds more similar. In software development, there are different compilers for multiple processor architectures, which can be used to port a program to another target hardware. The idea of HLS is to create the same level of abstraction – write code in a high-level language, which is agnostic of its target hardware architecture.

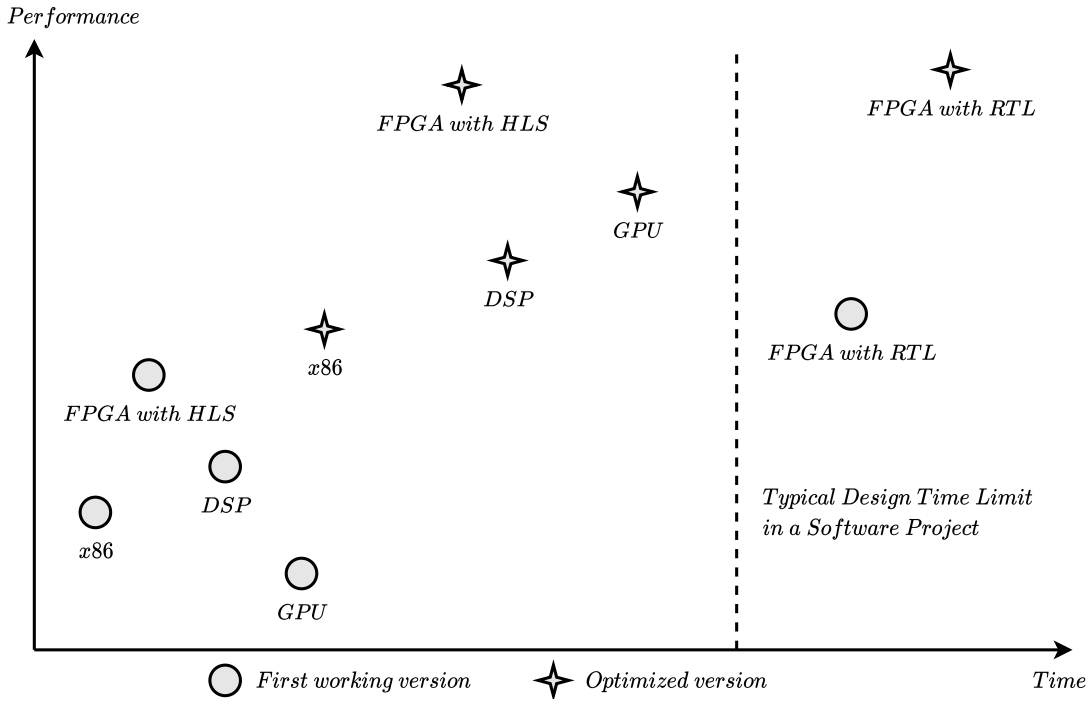


Figure 4.1: Design Time vs. Application Performance, if the same functionality is implemented in different methods for various targets [14, Figure 1-1].

A major achievement in the usage of HLS is the shorter development time. The time that it takes to develop a first working version of the intended design can be significantly reduced, compared to the development of RTL code directly. This is especially important if an FPGA design is combined with a software part, like in a System On Chip, because software development is comparably fast and thus, it is important to also be fast on the hardware design side, to be able to bring the two parts together as soon as possible.

Figure 4.1 schematically displays the difference in development time, over performance and degree of optimization, when the same functionality of a prototype is implemented in different methods for various targets. The compared targets are an x86 processor CPU, a digital signal processor (DSP), a graphics processing unit (GPU) and the FPGA design in RTL and HLS implementation. It is clearly visible, that the FPGA design with the HLS design approach achieves a first working prototype much earlier than the direct RTL approach. Even the optimized HLS version can be achieved within comparably short time, while both RTL versions, the first working version and the optimization, are beyond the typical design time limit in a software project.

4.2 State Of The Art

Information for this section is found in [5], [17] and [19].

HLS has been developed over 20 years ago in a first version, but has not yet been able to replace the manual, traditional implementation of RTL, by using the languages VHDL or Verilog. Unfortunately, no published numbers about the market share of HLS in relation to traditional implementations could be found during the research in this thesis. However, according to the given sources it is becoming more and more popular in recent years. This is mainly based on the fact that the HLS tools keep producing results of better quality as they evolve. This especially regards the efficiency of the produced RTL code, the useability of the tools, as well as the better understandable correlation between the high-level code and the generated RTL code.

Xilinx is the largest manufacturer of FPGAs globally and therefore its tool called *Vivado HLS* is widely used. Intel, the second largest FPGA manufacturer, also has its HLS tool called *Intel oneAPI* [28]. Both companies are conducting serious effort to push the HLS implementation method. Xilinx recently open-sourced the code for the front-end of their HLS compiler. The front-end processes the HLS code, specifically C++, and creates an intermediate representation from it. The company expects to expand the community engagement and the encouragement of innovation, driven by users, with this step. Also Intel with its oneAPI toolkit is trying to introduce a wider usage of HLS to the community.

In terms of language support, several tools settle for C++. Intel oneAPI uses library APIs and Data-Parallel C++ (DPC++), which is an extension to the ISO C++ standard. Xilinx also decided in favor of the C++ language for their tools *Vivado HLS* and the latest tool *Vitis HLS*. In older versions, SystemC was supported for synthesis as well, but is now deprecated in the respective latest versions [31].

Another feature of HLS is the support of libraries for complex functionalities, such as video processing. Xilinx HLS for example provides a port of the OpenCV image- and video processing library, that enables the development of complex functions in this area, inclusively the verification in the testbench. Also, other libraries are built-in, which feature commonly used functions, such as an FIR filter, for example.

Summing up, HLS is gaining more and more attention as it becomes more efficient. Its usage is being promoted and pushed forward with serious effort by the major FPGA chip manufacturers and tool vendors.

4.3 Workflow

This section describes the workflow for Xilinx HLS, as it is recommended in the user guide *Introduction to FPGA Design with Vivado High-Level Synthesis* [14].

Generally, as the user guide states, the quality and the correctness of the generated

output can only be as good as its input. Consequently, it is very important to follow the design recommendations and adhere to specific coding rules in order to produce a high-quality input software.

The following list presents the main steps that are required in the implementation workflow of an HLS IP core project.

- **Software Testbench**
  The developed HLS IP code needs to be verified and checked for correct functionality in a software testbench, before it gets processed to RTL code. This can be done by using conventional software development tools. Xilinx specifically recommends dynamic code checker tools like *valgrind* and *Coverity*. They support a developer in finding memory leaks, out-of-bounds memory access, uninitialized variables and similar issues. Furthermore, the application of code coverage tools like *gcov* is recommended as well. In that way, the percentage of executed code lines of testbench and HLS code can be measured, to ensure a sufficient level of testing. The software testbench is also called *C Simulation*.

  The implementation of the HLS IP code is subject to restrictions. As an example, dynamic memory allocation is not supported, since the entire code must be analyzable at runtime to be processed into RTL code. In contrast, the testbench code does not have any restrictions. Anything that is supported in C/C++ may be used here.

- **Co-Simulation**
  Co-Simulation serves the purpose to verify the correct functionality of the generated RTL code. It is not the goal to verify the algorithm, though. This is already done in the previous C simulation step. However, the issue is that the C simulation is executed like a regular program, which means it is executed sequentially, instruction by instruction and thus parallelism cannot be represented.

  This is where the Co-Simulation becomes neccessary. The RTL code is simulated with an HDL simulator software, which can actually simulate the parallelism as it exists in the FPGA hardware. In that way it can be ensured, that the user did not break the functional correctness of the algorithm, by giving wrong parallelization guidance to the HLS tool.

  The external simulator software has to run the complex HDL simulation and return data results, which is a time-consuming task. Therefore, the execution time of this simulation type is expected to be approximately 10.000 times slower, compared to the C simulation.

- **Integration of the generated IP**
  At this point, the developed HLS code is fully verified and can successfully be transformed into working RTL code. Xilinx HLS exports the generated RTL code

in an IP file- and folder structure that is supported by its IP integrator tool, that is used in Vivado or Vitis. Therefore, the developed IP core can be imported to the block design as another processing block, get connected with other IPs, and in that way get integrated into a larger design.

- **Synthesize FPGA bitstream**
  Like in any other FPGA development cycle, the created block design can now be synthesized. This generates a bitstream file, which can be programmed into the actual FPGA hardware. Depending on the architecture of the system, the developed IP may now interface with the on-chip CPU, in the example of a System-On-Chip.

These steps represent the standard workflow for the development of an integrated HLS IP. The successful deployment of hardware can be achieved if they are completed accurately. There may be more and deeper knowledge necessary, in order to use this design approach. This thesis however focuses on system design, not on HLS specifically, and thus a comprehensive study of the referenced literature is recommended.

4.4 Coding

This section gives a brief overview of some HLS-specific C++ features that are supported with Xilinx tools. More detailed information can be found in the respective user guide [15].

4.4.1 Data Types

HLS supports special data types, which are not available in standard C++. The focus of these data types is mainly to support an optimized translation into RTL, which requires the least amount of logic cells in the FPGA.

Standard C++ only supports variables in the bitwidth of byte-boundaries, so in multiples of 8 bit. However, this may be inefficient to fit into the available FPGA target hardware. One example therefor is the DSP48 macro that Xilinx FPGAs have built-in. They support the multiplication of only up to 18 bit. In an example application, the reachable number range of a variable might be limited to 17 bit for the multiplication. However, in order to represent that, using a 32 bit data type would be necessary in standard C++. The HLS compiler does not have this prior knowledge of the number range limit and would thus implement a 32 bit multiplication, which is a waste of resources in this case. Because of this reason, a major feature of HLS is the possibility to define data types with arbitrary bitwidths. In that way, the above example can be optimized to use a data type that has a bitwidth of 17 bit, which is exactly the bitwidth that is required for the application. This results in a lower logic utilization, which in turn results in a higher maximum clock frequency that can be used. Consequently, more functionality an be implemented in the FPGA device. The referenced user guide shows a design example where an area reduction of 75% is achieved by just reducing the bitwidths.

The argumentation is equally valid for integer data types, as well as fixed point data types. In calculations it is often necessary to represent fractional numbers. In standard C++, floating point data types are used therefor. However, floating point operations require a large amount of logic to implement and thus, HLS provides fixed point data types. These can be instantiated in arbitrary bitwidths, where integer and fractional bitwidths can be specified separately. The fixed point format is often described in the format *integer.fractional*, like *2.14*, which represents a 16 bit fixed point data type that has 2 integer bits and 14 fractional bits.

An example for the usage of integer and fixed point data types is given in the following code section.

```
1  #include <ap_int.h>
2  #include <ap_fixed.h>
3
4  ap_int<9>   custom_integer_var1;          //  9 bit integer, signed
5  ap_uint<17> custom_integer_var2;          // 17 bit integer, unsigned
6
7  ap_fixed<16,2>  custom_fixedpoint_var1;  // 16 bit fixed point, 2.14 format, signed
8  ap_ufixed<14,4> custom_fixedpoint_var2;  // 12 bit fixed point, 4.10 format, unsigned
```

4.4.2 Optimization Directives

The HLS code can be instumented with optimization directives. This is used to provide the HLS compiler with additional information and to instruct it on how to implement certain parts of the code in RTL.

Directives can be applied to interfaces, functions, loops, arrays or code regions. They can be written in a Tcl file that is provided to the HLS project, or in the design code directly, in the form of a pre-processor pragma. Both variants have their advantages and disadvantages.

Examples for such directives are optimization strategies for throughput and dataflow optimization, such as pipelining or loop unrolling. Also the implementation of the required interface types, such as various AXI interfaces, is controlled with directives.

Since this topic is very comprehensive, it cannot be covered into more detail here. For further information, refer to the user guide [15].

4.4.3 Interfaces

Interfaces are an important asset to any IP design, since this is where they communicate with their surroundings. This may be another IP that processes data, but may also be a CPU that reads status information or configures the IPs' settings.

Xilinx HLS implements two different types of interfaces: block-level, and port-level interfaces. The block-level interface is a general interface to the IP, that can provide

Program 4.1: Implementation of an AXI4-Lite memory-mapped slave with a separation of read-only status registers and read-write configuration registers.

```
1  typedef struct {
2    ap_int<NUM_LEDS> led_ctrl;
3    uint8_t code;
4  } config_t;
5
6  typedef struct {
7    uint32_t uptime;
8    uint32_t result;
9  } status_t;
10
11 void hls_ip_top_level(config_t& config,
12                        status_t* status) {
13     #pragma HLS INTERFACE s_axilite port = status bundle = API
14     #pragma HLS INTERFACE s_axilite port = config bundle = API
15
16     if (config.code == 0xFF) {
17       status->result = 0;
18       ...
19     }
20     ...
21 }
```

a handshaking functionality for when it is idle and ready to start a new operation, or when the operation is completed. The port-level interfaces on the other hand are specific to input and output ports. They can also provide handshaking functionality, such as data-valid indicators, acknowledge signals, or implement a FIFO for the IOs. However, port-level interfaces can also be implemented as AXI interfaces, such as AXI4-Stream, AXI4-Lite Slave or an AXI4 Master. This is an especially useful function, since the implementation of the actual interface logic is abstracted from the user. Additionally, software drivers are automatically generated along with the respective interfaces, which provides a fast and simple integration into a larger system.

The code example in Program 4.1 shows the specification of an AXI4-Lite memory-mapped slave interface. The provided registers are separated into configuration registers and status registers. The configuration is readable and writeable, while the status registers are implemented as read-only, which requires less logic in the RTL implementation.

The second code example, Program 4.2, demonstrates the equally simple definition of an AXI4-Stream interface.

HLS stream interface variables can be accessed via read- and write-functions. One very important property of these functions is, that they are blocking calls. The HLS streams are implemented with an underlying FIFO. Thus, values can only be written into it when there is an empty spot, and can only be read if there is a at least one value available. If the respective condition is not satisfied, the functions block the execution, until the FIFO state changes. During simulation in the testbench, this may result in an

Program 4.2: Implementation of an AXI4-Stream interface.

```
1  #include <hls_stream.h>
2
3  void hls_ip_top_level(hls::stream<uint32_t>& sample_in,
4                        hls::stream<uint32_t>& audio_out) {
5      #pragma HLS INTERFACE axis port = sample_in
6      #pragma HLS INTERFACE axis port = audio_out
7
8      auto input = sample_in.read();
9      ...
10     audio_out.write(audio);
11 }
```

endlessly blocking behaviour.

4.5 Testbench

The general significance of the simulation testbench for a HDL design is explained here, as well as the different levels of abstraction in which a HLS testbench can be executed. These different levels of abstraction in the testbench execution are a major difference between a VHDL testbench and an HLS testbench. The most important impact thereof is the execution speed – the time it takes to simulate a number of test cases on the DUT.

Section 4.3 above already mentioned the topics of testbenches and simulation types. The following descriptions shall be seen as a comprehensive extension around these topics.

4.5.1 General

In the development of software that runs on a CPU, the tool of step-by-step debugging is commonly used during the development process. Various verification and testing libraries exist, in order to create unit tests, for example. The advantage is, that the code that is developed there, can be compiled and run on the target CPU directly, while maintaining the direct instruction-by-instruction representation of the code.

However, in the development of hardware with any HDL, this is not possible. HDL languages provide a syntax and semantic representation that allow to create parallel structures and a dependency on timing, such as a system clock. The design is based on signals that perform logic operations and transfer values into registers. Thus, HDL code is often called register-transfer level, or short, RTL code. Before this code can be run on the target hardware, i.e. an FPGA, it needs to run through a process called synthesis. In synthesis, the HDL code is translated into a netlist and is consequently mapped into hardware logic cells that are available in the target device. Various optimizations are done during this process, so that the direct connection to the code representation is lost. Additionally, it is impossible to access every single signal within the design during

debugging on the hardware, which is probably also one of the main issues. Synthesis tools like Xilinx Vivado provide support to make debugging possible in hardware, but require a large effort to implement them, especially in terms of synthesis time. Another big factor that speaks against debugging in hardware generally, is that the process of synthesis takes a comparably large amount of time to transform code into the target technology. Each time the code is adapted, the entire synthesis process needs to be started over again. Because of all of these reasons, debugging HDL code step-by-step as in software is not possible.

Thus, in order to allow debugging and verification of HDL code, the technology of simulation was invented. Here, the HDL is also translated into a netlist that represents all the parallel processes just like in hardware, but the subsequent step of synthesis is not taken. Therefore, simulation comes to a result much faster. Since HDL designs actually only consist of electrical wires and some basic block primitives, like flipflops or lookup tables, it is difficult to verify correct values. They would need to be compared in their binary bit-pattern format, at the correct point in time. Of course the HDL language provides support with different datatypes, that abstract the binary number representation and make the numbers human-readable. However, it remains an enormous effort to manually write testing code to verify a bus interface, for example. In order to simplify such complex operations, various testbench frameworks and libraries are available to developers.

4.5.2 C Simulation

In C Simulation, the HLS compiler is used to create an executable binary that includes the testbench code, as well as the IP design of the DUT. This binary can be run just like any other program. The most important note here is, that the HLS code is *not* transformed into RTL at this point. Instead, the HLS code is treated as a regular C++ code, that uses some libraries, like the fixed-point library. This means, that the HLS IP cores' top-level function is used as a regular function call in the testbench.

The main advantage of this type of simulation is the major increase of execution speed. The reason for this speed-up is the abstraction of timing, such as clock cycles or similar constraints. In fact, there is absolutely no timing information that needs to be taken into account in this simulation.

4.5.3 C/RTL Co-Simulation

This type of simulation actually transforms the HLS IP code into RTL code, such as Verilog or VHDL, for the testbench. This is done in three steps.

- **Create input vectors**
  In this first step, the above explained, regular C simulation is executed. Hereby, the inputs that are sent to the DUT are stored as *input vectors*.
- **Perform RTL simulation**
  Next, the previously generated input vectors are applied to the DUT. At this point,

the DUT is already translated into RTL code, so the code that is used here is either VHDL or Verilog code. The default simulator is the Vivado Simulator (XSim). However, various other simulators like ModelSim are supported. The generated outputs of this step are stored as *output vectors*.

- **Verify the results**
  The final verification step sources the output vectors back to the C testbench, which compares the outputs with the expected results.

This type of simulation takes a significantly longer time to execute, because timing information now actually needs to be considered for the RTL code.

Chapter 5

System Architecture and Concept

This chapter introduces the FM radio receiver project that is implemented as a major part of this thesis. It specifically talks about the system level approach that is taken in the implementation of the project. The main concept, as well as the architecture are explained.

5.1 Overview

In the development of a project that includes multiple functional parts, it is always of advantage to start with a system level overview. Major decisions can be made at this stage, before any implementation has begun. This can have a direct impact on the efficiency of the entire development, which concequently has an impact on the final quality of the product. A thorough system level design can also prevent from issues that would otherwise arise while the development is already ongoing. As an example it could turn out that one of the functional parts is not able to fit into the integrated system, in the way it is implemented. This leads to the necessity to re-engineer this entire functional block. As a result, it takes more effort to implement the functionality, which directly correlates with the development cost and the time-to-market, in order to deliver the final product. Summing up, a comprehensive system level architecture and concept can pave the way for a successful implementation of a project from start to finish.

In order to create a system level design, considerations around the high-level, final goal of the product are a good starting point. This includes the definition of its functional range. Once this is done, the target system or target hardware should be considered, because there are various ways to implement a certain functionality. All of these variants will eventually result in the same output, but will, however, heavily differ in their implementation effort, efficiency, or their applicability for the products' aim in general. Thus, things like the available hardware platform, implementation time, degree of optimization, output quality, and similar things are important factors at this stage. Of course, there are many more factors to consider when a product is to be developed. However, only a subset of these is covered here, since this chapter is focused on how this thesis' accompanying project was developed and is thus written in that context.

5.2 The Project

The aim of this thesis is develop a system architecture and concept, that allows the implementation of an FM radio receiver in multiple different ways, while providing an elegant way to compare the different solutions. The final goal thereby is always be the same, that is: Listen to the music of a radio station that is transmitted over FM broadcast radio.

A very high-level overview of the entire system is given by the block diagram in Figure 5.1. It shows the front end, which processes the received antenna signal, so that it can be used in the FM radio receiver block. The radio receiver itself is implemented in multiple different variants. The produced audio output is sent to a speaker, since the ultimate goal is to listen to a radio stations' audio signal.

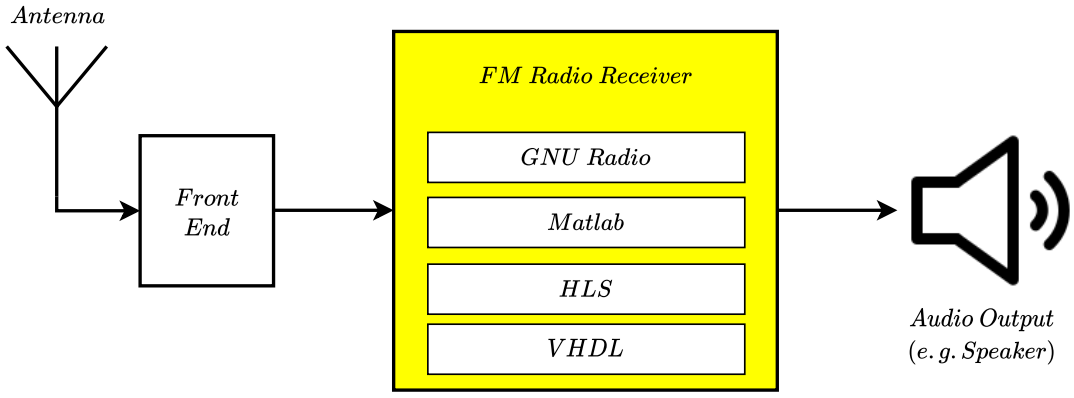


Figure 5.1: High-level overview of the system.

In the context of a system level design approach, several steps are necessary to find a solution that is ready to be implemented. As a very first step, the inner workings of an FM receiver need to be understood. This mainly regards the signal processing theory, to be able to decode the signal that is received by the antenna. Chapter 2 describes this part into detail.

As already mentioned above, multiple ways of technology and implementation can lead to a similar result. Thus, the next step is to define a set of possible ways which will actually be implemented. For this thesis, the decision was made to implement the FM radio receiver by using the following methods.

- **GNU Radio**
  GNU Radio is a free and open-source software development toolkit to implement a software-defined radio. The implementation is done by connecting functional blocks in a block design. This abstracts the inner workings of each functional block and thus provides an implementation approach on a very high level of abstraction. GNU Radio supports interfaces to external RF hardware [34].
- **Matlab**
  Matlab is a software platform that can work with matrices and compute numerical solutions. Matlab also includes various toolboxes, that support developers with

complex calculations and tasks on datasets. This includes a range of toolboxes for signal processing, such as filter designers or FFT functions, as well as convenient ways to visualize data [30].

- **FPGA: C++ High-Level Synthesis**
  In order to use an FPGA, a hardware design needs to be developed for it. This can be done using high-level synthesis. This thesis focuses in C++and does not implement a SystemC variant, because Xilinx has deprecated SystemC support for synthesis in their latest HLS tools. This, and the details of High-level synthesis (HLS) are explained in Chapter 4.
- **FPGA: VHDL**
  Similar to the HLS approach, this is a variant to write a hardware design for an FPGA. The language VHDL is used in this approach.

In order to implement an FM radio receiver, all of the listed methods require deep knowledge about the neccessary details of signal processing. However, the implementation of the various different methods require more or less knowledge and effort on the exact underlying implementation of each signal processing part. This depends on their respective level of abstraction.

5.3 System Concept

The overall system concept has to be developed in a way, to fit the respective projects' needs and requirements. There should be a clear structure, in order to help to create a working system successfully. This structure should include a description of the data source, the data processing blocks and stages, commonly used parameters, as well as a strategy to verify the correct functionality of the system. A good system structure tries to share as many parts as possible in the system. This is to minimize the points of failure, as well as to reduce the effort of implementation.

In the apparent projects' case, a single block of the system is implemented in multiple ways, which should, however, all compute an equal output. Thus, it is of advantage to define the data source to be common for all of these variants, to create a unified situation at the input. Additionally, the data source may be chosen to be recorded, to have a known, reproducable set of input data. This significantly eases the verification of the functionality in the final step, since all variants are expected to create the same output in the optimum, fully functioning case. Not only the data source is important to be common, but also the way of verification. In order to achieve that, the verification block needs to be able to compare multiple data outputs with the verification data, which is chosen to serve as the reference.

An example of a system concept and its architecture is developed and shown in Figure 5.2. Here, a common data source is provided by the block called *Front End*. It records data from an antenna and saves it to a file, in a format that the consequent blocks require at their input. Therefore, a reproducable input is ensured. Within the *FM Radio Receiver* block, all the different implementation variants are shown in parallel. They all work on this common input source and generate their respective output. This

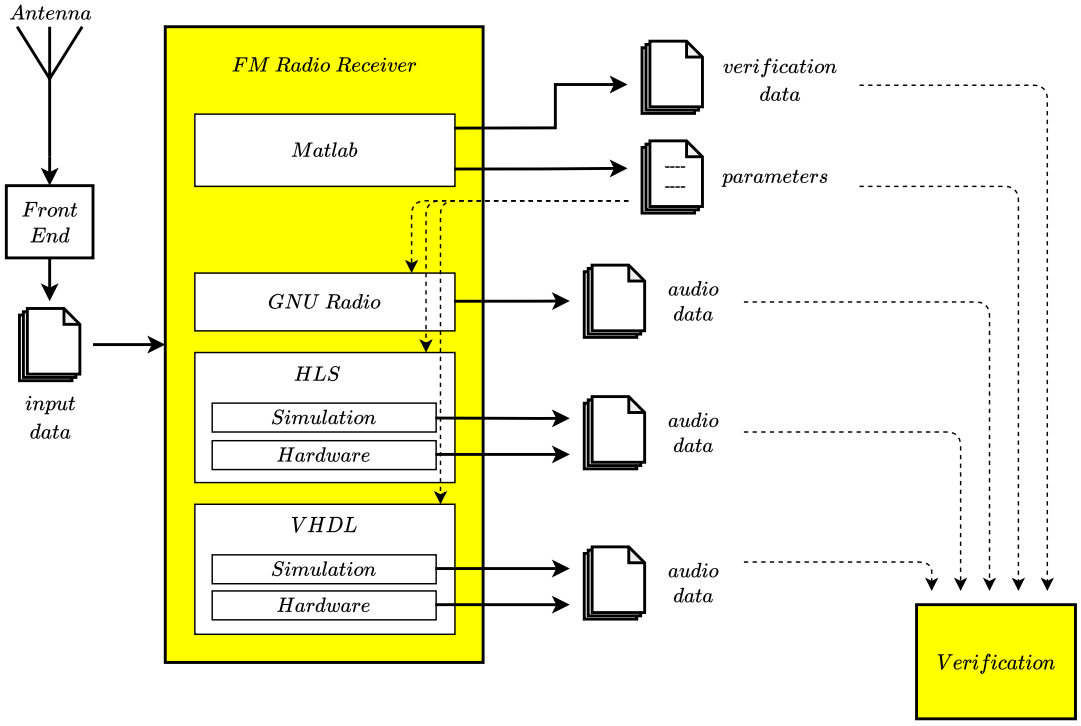


Figure 5.2: Block diagram of the overall system concept.

output data is again written to a file. Hereby it is important to note, that the output of the *Matlab* implementation is chosen to be the *golden reference model* of correct output. Its output is used as the verification data later in the process. This is because Matlab was used to research decoding algorithms and to develop the DSP chain, which is used in the GNU Radio, HLS and VHDL implementations. In addition to the verification data, the Matlab model also writes common parameters to a file. These parameters are directly used in the other implementations, to ensure a common setup. The final block is the *Verification* entity, which compares the output files of all the variants, with respect to the chosen model parameters.

A major point in this system concept is, that it works independently of the respective implementation methodology. This is especially important, since there is no possible way to run all these different variants in a single verification environment. Instead, each methodology writes an output file, which is a supported functionality that is available anywhere.

Chapter 6

Implementation

This chapter describes the most important details of each implementation variant. The implementation variants in High-Level Synthesis and VHDL are highlighted especially, since they are the main focus of this thesis.

6.1 Digital Signal Processing Chain

The theory behind the demodulation of an FM signal is already explained in Chapter 2. The demodulation of an FM broadcast channel specifically, is described in Section 2.8. In this section however, the DSP chain is described into more detail and with more information, as it is implemented in the various methods.

Figure 6.1 displays the detailed DSP chain as a block diagram.

The following list explains implementation details about the respective DSP blocks.

- **Data Source**
  Matlab is used to generate the data source file, which is used as an input to the system. An audio file is read and modulated as an FM channel signal.
- **FM Demodulator**
  TODO
  The FM demodulator is implemented as a Baseband Delay Modulator, as described in Section **??**.
- **De-Emphasis Filter**
  The above described Matlab source generation does not include an Emphasis filter. Thus, a de-emphasis filter is not implemented as it is not required.
- **Downsample**
  Downsampling is implemented in a simplified version, without an anti-aliasing filter. It is not required, because of the following explanation. At this point the sampling frequency is set to 960 kHz. The input is band-limited at 75 kHz, which is the maximum frequency in an FM channel, as explained in Section 2.6.3. Thus, the input has an oversampling rate of 12.8, while downsampling is only performed with a factor of 8.

- **Filters**
  Several filters are implemented as FIR filters. The 15 kHz lowpass (LP) filter is instantiated twice, with the exact same coefficients.
- **Filter Group Delay Compensation**
  FIR filters introduce a delay on the output signal, with respect to the input. The group delay in number of samples can be computed with the following formula.

$$group\ delay = \frac{filter\ order}{2} = \frac{(N\ coefficients - 1)}{2} \tag{6.1}$$

  This is due to the property of the FIR filters' linear phase response [10]. Since phase is directly proportional to time, the term *group delay* is often used. This delay needs to be accounted for, depending on where the signal is used subsequently.
  Looking at the block diagram, the group delay needs to be compensated at two places.
  - **Between the branches *Recover LR diff* and *Recover Mono*:**
    The *Mono* branch only contains a single LP filter, while the *LR diff* branch contains two filters, namely a BP and an LP filter. The latter branches' signal is therefore delayed, with respect to the *Mono* branch.
    The issue here is, that both signals are added and substracted at the end of the processing chain, to compute the left and right channel signals, respectively. Thus, they need to be aligned in time at that point. Consequently, a delay block needs to be inserted in the *Mono* branch, to compensate the group delay of the BP.
  - **Between the branches *Recover LR diff* and *Recover Carriers*:**
    Both branches only contain a single BP filter. However, they have properties, like the transition bandwidth and the passband. Thus, their filter order may be different as well, which leads to a non-matching group delay. Since the two signals are multiplicated in the subsequent mixer, and an important factor is the phase-coherency of the recovered carrier (as explained in Section 2.8), the group delay compensation becomes necessary here.

    To overcome the issue at this point in the actual hardware implementation, the filters are manually adapted to have a matching order.
- **Carrier Recovery**
  The 38 kHz carrier can be recovered from the 19 kHz pilot tone. It is important that the recovered carrier is phase-coherent with the received input, as described in Section 2.8. This can be achieved, by simply multiplying the pilot tone with itself, which is the way it is implemented here.

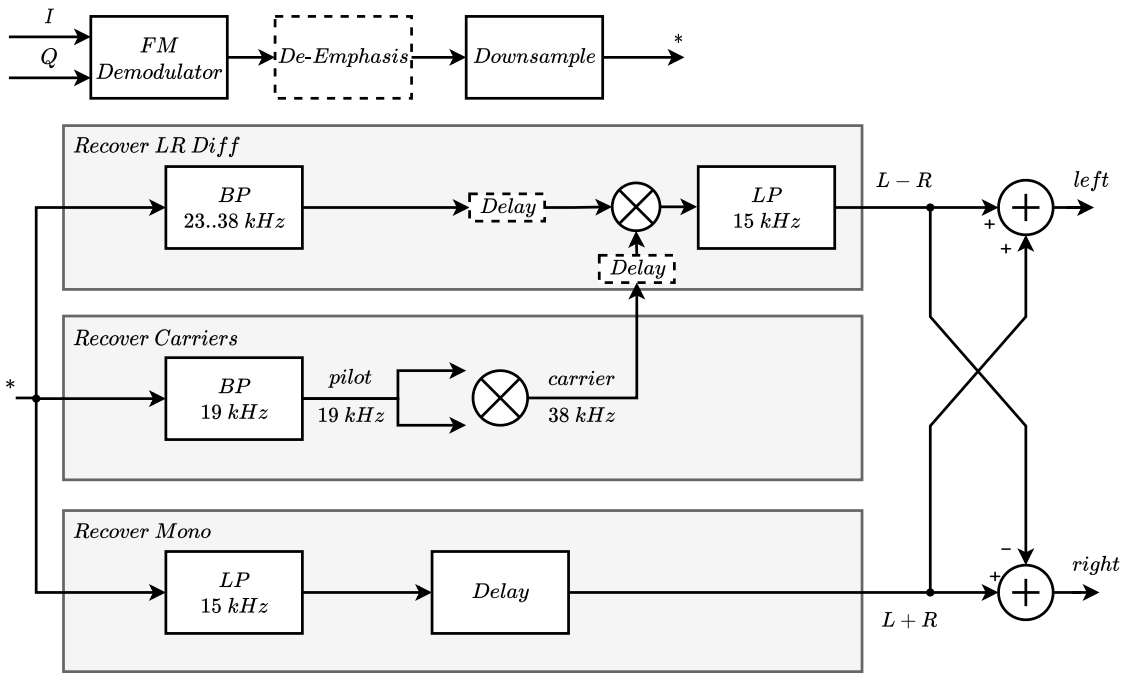


Figure 6.1: Detailed block diagram of the digital signal processing chain.

Several of the listed details are implemented, in order to achieve a successful demodulation of the FM signal.

6.2 Matlab Model

Matlab is used to create a model of the receiver, which is used as a reference for the other implementation variants. Here, DSP algorithms are tested and evaluated, to find a method that is suitable to implement in hardware, specifically in an FPGA.

The Matlab script consists of a transmitter, the receiver model, and additional code for analysis.

6.2.1 Transmitter

The transmitter is implemented to provide a reproducable source of data, which can be used to feed the receiver. Another option would be to use external hardware as a software-defined radio and record some antenna data. However, the main advantage of a locally implemented transmitter over a recording is, that the raw data input is known. Thus, the expected output of the receiver model is this raw data, that was previously modulated by the local transmitter. In the case of a recording, the original data is not known, which does not allow to actually verify the receiver.

In the implemented version, an audio file is modulated into the FM channel. The audio file contains data that represents speech with the words *Left channel... Right*

channel.... In terms of audio channels, the words exactly match the respective channels' output – the first part is modulated on the left audio channel only, while the second part only targets the right audio channel. This is particulary useful for verification, especially to verify the functionality of channel separation. From the experience during development, the channel separation is a good indicator of whether the system is functioning correctly. This left-right indication also simplifies the verification process, because a developer can recognize it with their visual and acoustic senses. It is possible to audibly verify it by listening to the output. It is also possible to just visually inspect the output data, where a clear separation is visible between the channels.

The transmitter is implemented according to the structure that is depicted as a block diagram in Figure 6.2. To facilitate an easy understanding of the block diagram, it is of advantage to keep in mind that it exactly resembles the frequency spectrum that is previously shown in Figure 2.7.

The transmitter reads an audio file, including a left and a right channel in the first step. The audio file contains data in a sample rate of 48 kHz. Next, the sample rate is increased by an oversampling factor of 20, to achieve a sufficient sample rate for the subsequent DSP procedures.

The achieved sample rate of 960 kHz could actually be chosen much lower, since an FM channels' maximum frequency is 75 kHz (see Section 2.6.3), and according to the Nyquist theorem the sample rate thus only needs to have 150 kHz (see Section 2.4). However, the higher rate was chosen, mainly because a higher number of signal samples beautify any plots that are created during analysis. Since a thorough optimization of the DSP chain is not the focus of this thesis, this was an acceptable trade-off.

The next step is the pre-emphasis filter, which is implemented, but is disabled in the receiver. This is done to reduce some complexity in the hardware implementation. The following stage generates the Mono and LR Diff signals, by simply calculating the summation and diffence of the left and right channels, respectively.

The next section in the block diagram represents multiple processes in parallel. The LR Diff part is shifted to 38 kHz in the spectrum. This is done by modulating the signal with its sub-carrier of 38 kHz. A subsequent BP filter serves to remove the artifacts of the modulation, such as the image replica. It band-limits the spectrum of the LR Diff part between 23 and 53 kHz. The BP filter introduces a constant group delay of the signal. In order to maintain the alignment of all signals, this side-effect needs to be accounted for and be compensated in the other signal branches. Thus, the Mono signal, as well as the Pilot Tone need to be delayed by the amount of samples defined by the BP filters' group delay. The subsequent amplifier blocks create the defined proportions of the sub-band, such as 90% for the audio parts, and 10% for the Pilot Tone [8].

One part in this block diagram is the so-called Hinz Triller, which was part of the Autofahrer-Rundfunk-Information service (ARI, German for Automotive-Driver's-Broadcasting-Information). It was used as an indicator for traffic announcements. It is

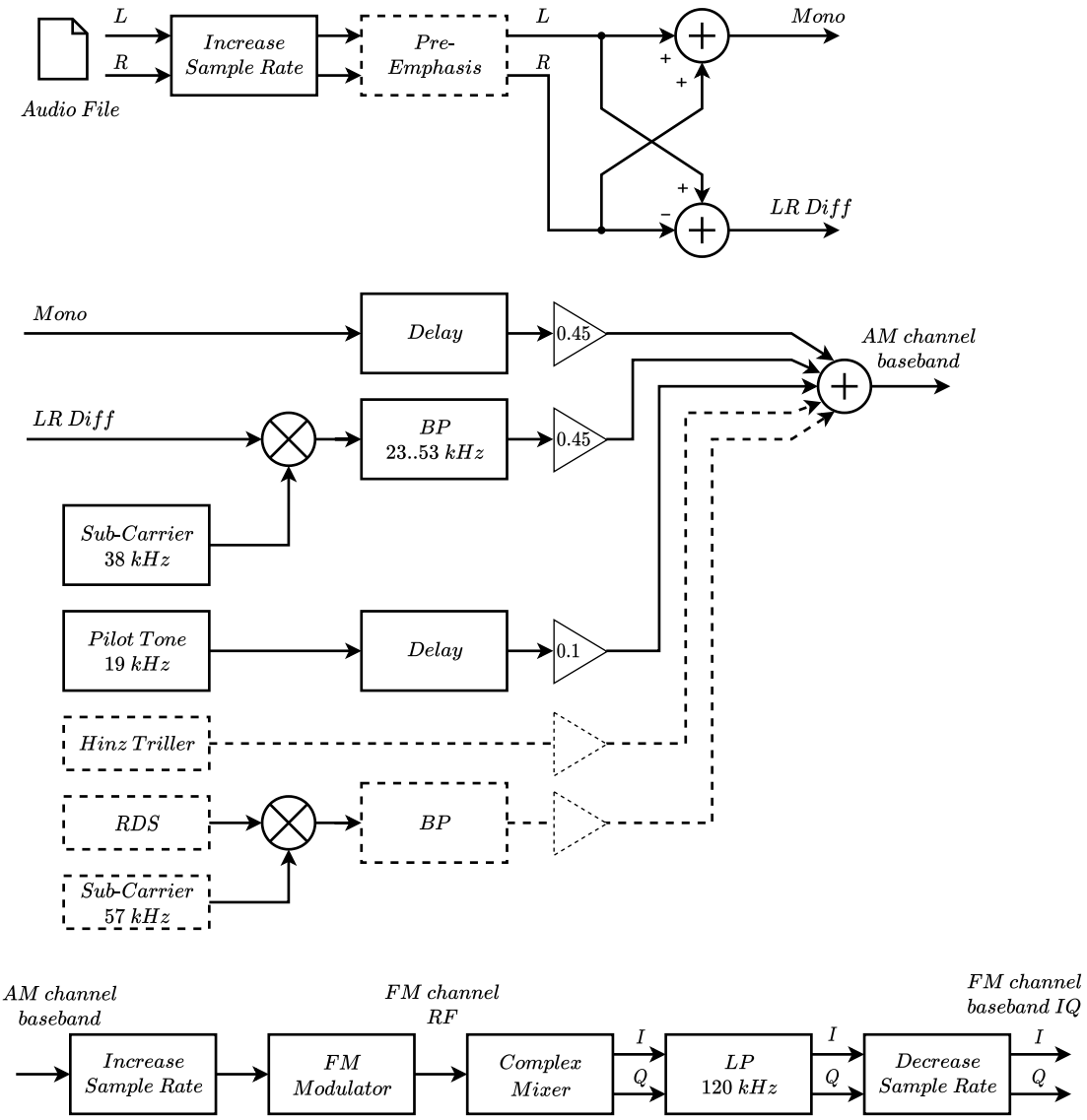


Figure 6.2: Block diagram of the implemented transmitter in Matlab.

now obsolete and is replaced by the more modern Radio Data System (RDS). However, the Hinz Triller is still transmitted in many radio stations and can often be heard before and after the traffic announcements [39]. In the implemented transmitter, the Hinz Triller is implemented, but is disabled, since it only represents a temporarily transmitted signal.

The Radio Data System (RDS) signal is not implemented in the transmitter, since it consists of a comprehensive protocol and multiple steps of modulation, such as differentially-coded BPSK [11] and is therefor too time-consuming to implement. Furthermore, it would not contribute any additional value to this project, since this is not the main focus.

At this point in the DSP chain, the signal resembles the frequency spectrum that is displayed in Figure 2.7.

In the final chain of DSP blocks, the sample rate is increased again, to be able to frequency-modulate the signal. The resulting frequency modulated signal represents the signal that is received by an antenna – the RF signal of an FM channel. This signal is then processed by a complex baseband mixer, which results in inphase-, and quadrature components in the baseband. An LP filter band-limits the signal and removes artifacts of the mixer. As a final step, the sample rate is decreased again, back to the chosen sampling rate of 960 kHz. This signal can now be processed in a digital FM receiver.

6.2.2 Fixed Point Arithmetic

The target hardware platform of the FM radio receiver project is an FPGA. These are devices that have a limited number of logic cells, which can be used to implement logical and computational functions. Any kind of computation can be implemented within these logic cells. However, depending on its detailed implementation, the amount of required logic cells may differ by a large factor. In order to overcome this issue, mathematical operations are often implemented in fixed point arithmetic, rather than floating point. This is due to the fact, that fixed point operations can be implemented with much less logic cells than floating point operations. The reason therefor is, that the bitwidth of input and output numbers is previously known in fixed point. Floating point calculations do not have this assumption and thus need to be more flexible, which requires a larger amount of logic cells.

The Matlab model is therefore developed by keeping in mind this limitation. Matlab provides special datatypes for fixed point number representation. However, these were not used in this implementation. Instead, everything is calculated in floating point representation, using the standard *double* datatype. Then, in certain positions in the DSP chain, the numbers are rounded to the resolution of a fixed point datatype. A fixed point format of *2.14*, meaning 2 bit for integer- and 14 bit for fractional number representation is chosen. This is sufficient for the FM receiver and could potentially be lowered to a lower bitwidth. Again, an optimization of the DSP chain is not the focus of this thesis.

Summing up, the Matlab model is not implemented in fixed point arithmetic, but is implemented in a way, that is close enough to the hardware implementation. Therefore, it can serve as a reference for the hardware implementation.

6.2.3 Receiver

The receiver is implemented, following the block diagram in Figure 6.1. Further assumptions and simplifications, such as the fixed point arithmetic are described into detail above.

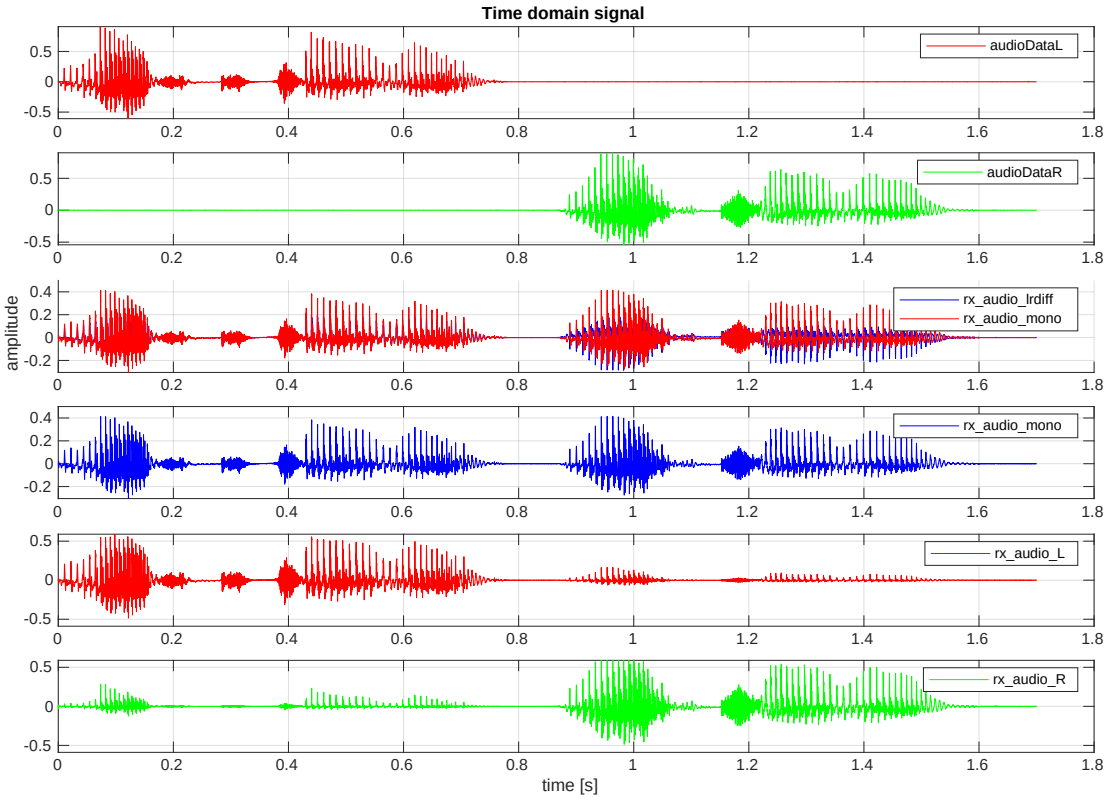


Figure 6.3: Matlab analysis in the time domain.

6.2.4 Analysis

The Matlab implementation also includes a comprehensive range of analysis, especially in terms of visual plots. These plots are useful during the development of new DSP algorithms, in order to verify the correct thereof.

An example of a plot that is created by the analysis is depicted in Figure 6.3. The diagram shows the respective signals in the time domain. Above two timelines display the input data, as it is read from the input audio file. The two diagrams in the center plot the Mono and LR Diff signals. The bottom diagrams then represent the result after the entire signal processing chain. It can be seen that the DSP chain is not able to demodulate the original signal without any error. However, the result shows a clear separation of the left and right channels, as resembles the original input very closely, which is good enough for this case study.

Further analysis is done in the frequency domain, especially in order to better understand the modulation of the different spectral parts in the FM channel. An example thereof is shown in Figure 6.4. The spectrum of an FM channel multiplex signal, as it is generated by a transmitter before the modulation, is shown in this diagram. The spectrum after the demodulation through a receiver is shown in the lower diagram. Spectral parts, such as the pilot tone at 19 kHz, or the LR Diff part around 38 kHz can be seen

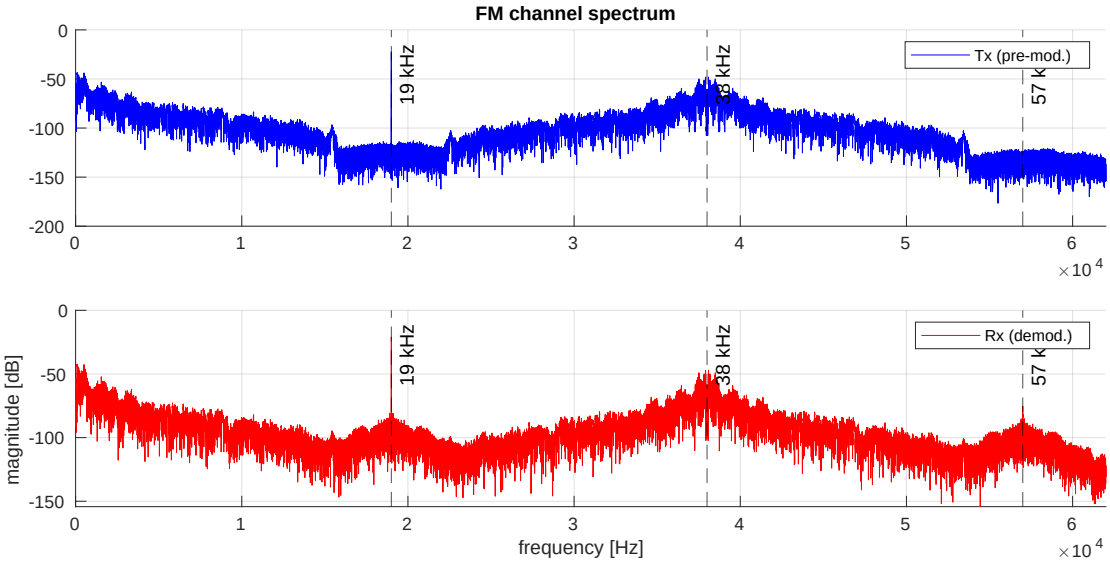


Figure 6.4: Matlab analysis in the frequency domain.

clearly. The demodulated spectrum shows an artifact at 57 kHz, which does not exist in the original transmitter spectrum. This is an artifact of the demodulation process, specifically the frequency-shift of the LR Diff part to baseband. The artifact is very low in energy, because it is attenuated by a BP filter in the down-conversion process, and can thus be ignored for further processing, even though it can still be seen in the spectral analysis.

6.3 GNU Radio

This section describes the implementation in GNU Radio. A transmitter, as well as a receiver are implemented using this software. Both can be deployed on actual hardware, so a continuous transmitter or receiver is built in a real environment.

6.3.1 Introduction

GNU Radio is a free and open-source software development toolkit to implement a software-defined radio. The implementation is done by connecting functional blocks in a block design. This abstracts the inner workings of each functional block and thus provides an implementation approach on a very high level of abstraction. GNU Radio supports interfaces to external RF hardware [34].

6.3.2 Transmitter

GNU Radio provides a comprehensive set of DSP blocks, which are available to build various functionalities. Multiple blocks are connected to a block diagram, that is very similar to the one in Figure 6.2. The USRP device is used to broadcast the RF signal over the air.

Again, an audio file is used as an input, which is modulated as an FM channel signal. The entire multiplex signal is then modulated onto an actual carrier frequency of 99 MHz, that is within the FM radio broadcast band. This entire DSP chain is being run in the GNU Radio software, on a host PC. Only the final signal – FM modulated, and on a carrier of 99 MHz – is then sent to the USRP device in the form of digital signal samples. The USRP hardware then transmits the signal via its antenna.

Special care needs to be taken when transmitting signals over the air. The frequency spectrum is strongly limited by the radio regulations and may be monitored by the authorities. Please make sure to adapt your setup to the respective local regulations, to avoid any legal issues. Some of the regulations are mentioned in Section 2.6.2, but may differ in various countries, however.

Figure 6.5 shows a block diagram of the transmitter application.

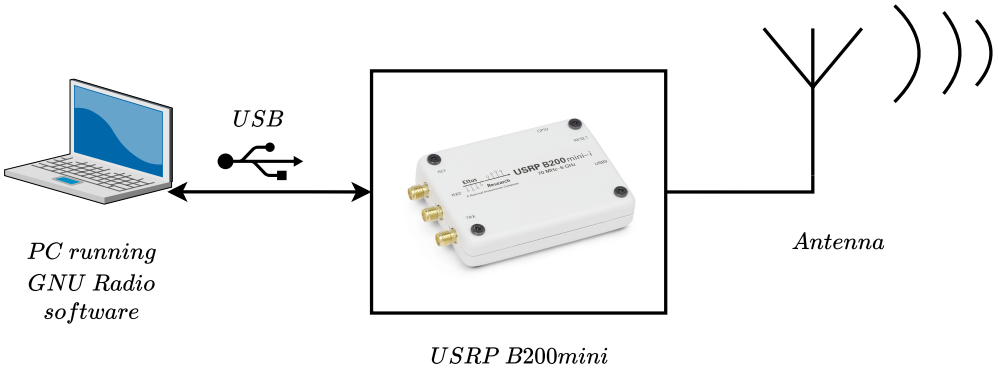


Figure 6.5: Block diagram of the GNU Radio transmitter setup.

6.3.3 Receiver

The receiver is also implemented, by using standard DSP blocks that are available in GNU Radio. The implemented structure however does not represent a stereo receiver. Instead, a mono receiver is implemented. The antenna signal is FM-demodulated, decimated and then directly sent to the speakers as the output, without any further processing. This simple receiver was intended as a proof-of-concept and a first example of the cooperation of the external hardware and the GNU Radio software. Thus, the mono receiver was sufficient, as the focus of the GNU Radio application is put on the transmitter side.

This receiver works with both devices, the USRP b200mini and the RTL-SDR. Figure 6.6 shows a block diagram of the receiver application.

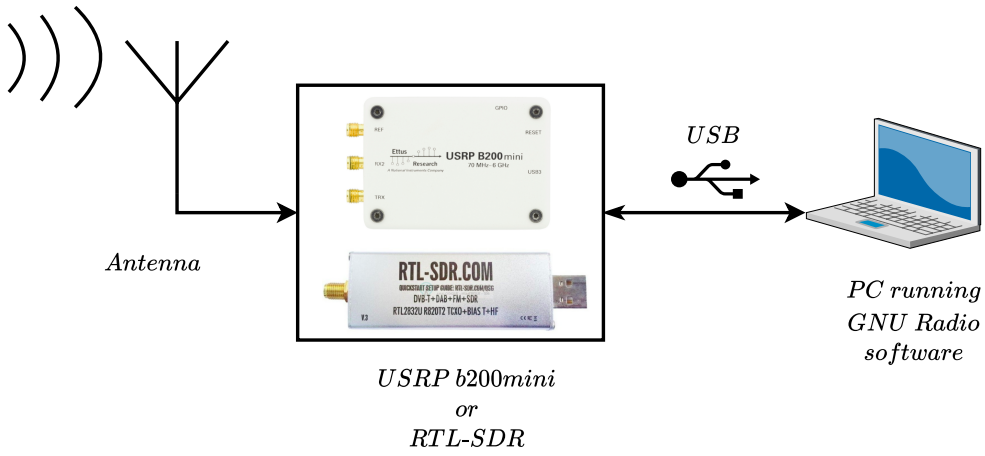


Figure 6.6: Block diagram of the GNU Radio receiver setup.

6.3.4 System Level Integration for Verification

The usage of a software like GNU Radio in combination with external hardware is an interesting tool for a use-case like this project. An FM receiver is to be implemented in multiple different versions. All of them need to be verified somehow – this is where the SDR transmitter created with GNU Radio and external hardware opens up an entire new possibility from the viewpoint of system level design integration.

Previously it was explained that it is not possible to use recorded antenna data for verification. The reason was, that the underlying original data is not known and therefor a comparison of the original versus the received, decoded data is not feasible. Consequently, the Matlab transmitter model was justified by the fact, that its originally encoded audio data is well-known.

However, with the current approach of using a GNU Radio SDR transmitter, the encoded data *is* well-known, because it is generated locally. Thus, the SDR can be used to transmit a verification signal. This signal could be recorded to file by another SDR, or sourced into the DUT (device under test) directly.

6.4 VHDL

In this section, the most important details of the VHDL implementation are described. This implementation variant is one of the two major implementation focus points of this thesis.

6.4.1 Testbench Simulator and Framework

The HDL compiler and the testbench framework that are used in this project are introduced here.

Simulator GHDL

Multiple different simulators are available on the market, such as Mentor Graphics ModelSim or QuestaSim, Synopsys VCS or the Cadence Simulator. The issue with most simulators is, that they are subject to a license fee, depending on the required set of features, or simulation speed.

However, there are simulators available, that are free to use. An example therefor is GHDL, which is an analyzer, compiler and simulator for VHDL and is developed as an open-source project. GHDL translates the VHDL code to a binary file that contains machine code, which can be executed. This is supported for a all common operating systems such as Linux, Windows and macOS. The simulator further supports a VPI (Verilog Procedural Interface), which can be used to control the simulator [29].

Testbench framework cocotb

The testbench framework cocotb is developed as a free, open-source project. It can be used to verify VHDL and SystemVerilog RTL code, in combination with an external simulator. Various different simulators can be used with cocotb – amongst others, GHDL is supported. The interface that is used to the simulator is the industry-standard VPI.

One of the main advantages of cocotb is, that test cases for the respective RTL code are written in the language Python. It is therefore also called a co-simulation framework. Using Python provides a large factor of flexibility and possibilities, since it is a comprehensive language, considering all its extension libraries. Additionally, Python may be more familiar to engineers in comparison to VHDL and thus, writing testbenches is made simpler with cocotb.

The test cases in Python are implemented as coroutines. By the way, this is where cocotb gets its name from. It is constructed by the words *CO*routine, *CO*simulation and *T*est*B*ench [27].

Figure 6.7 gives an overview of how cocotb interfaces with a simulator. The Python test cases are shown as a list of coroutines, which are managed by a scheduler. This Python code interfaces with the simulator through standard interfaces like VPI. The block called *GPI* is developed by cocotb and abstracts the different simulator interfaces, such as VPI, VHPI or FLI. In the simulator, the device under test (DUT) is instantiated. The RTL simulation itself is managed by the simulators' scheduler.

6.4.2 Testbench Architecture

The architecture for this testbench is created under the assumptions and according to the guidelines that are developed in Chapter 5.

Figure 6.8 shows a block diagram of the implemented architecture. The split between the Python environment and the VHDL implementation is clearly visible but is represented from the developers perspective. The actual simulator interfaces, as described in

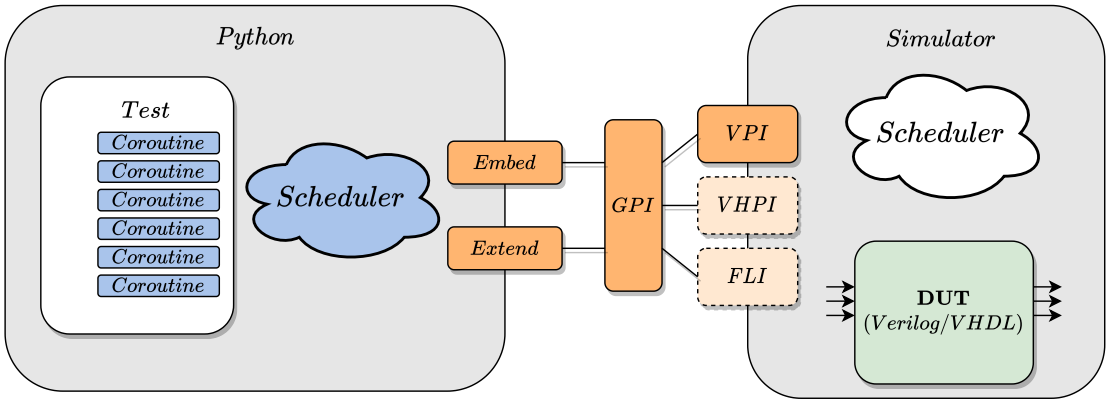


Figure 6.7: Block diagram of the interface between cocotb and a simulator [27].

Section 6.4.1, are hidden to maintain the overview of the implemented architecture.

The test cases are using the external verification data, that was previously generated by the Matlab model. This data is supplied to the input of the DUT. The DUTs' input and output is implemented as an AXI Stream interface, which is an interface that features a valid-, as well as a ready-signal along with the data. The cocotb framework supports both an AXI Stream Master, and an AXI Stream Slave class, to write and read the input and output, respectively. In that way, the interface signals are abstracted to simple read- and write-function calls in the Python code, while the actual signal handling is done in the mentioned cocotb library classes. A clock generator class is also available from cocotb, which is used to provide the system clock to the DUT.

The Python classes *tb_analyzer_helper*, *tb_data_handler* and *fm_receiver_model* are implemented to create a better structure of the testbench and to provide code that is reuseable for the testbench of the HLS implementation. The *tb_analyzer_helper* class provides functions to analyze the data, such as comparison against defined error thresholds or the creation of graphical plots with Pythons' matplotlib library. The *tb_data_handler* only works as a data storage class, that encapsulates the data that is captured from the DUT. The class called *fm_receiver_model* serves as the golden model reference that is used to compare the DUT against. Internally, it loads the Matlab verification data. This could be implemented differently, since Python can be used to actually implement an FM receiver including the entire DSP functionality, just like in Matlab. Because the Matlab model already exists, this is not done here – loading the Matlab data is sufficient as it is been seen as the single source of truth in this project.

At the end of the test cases, the DUTs' output data is written to a file. This data is later used to directly compare the different implementation variants against each other.

6.4.3 Testing Strategies

Common strategies for testing are unit tests and integration tests, which represent different levels of testing. Unit tests specifically target a single unit, while integration tests

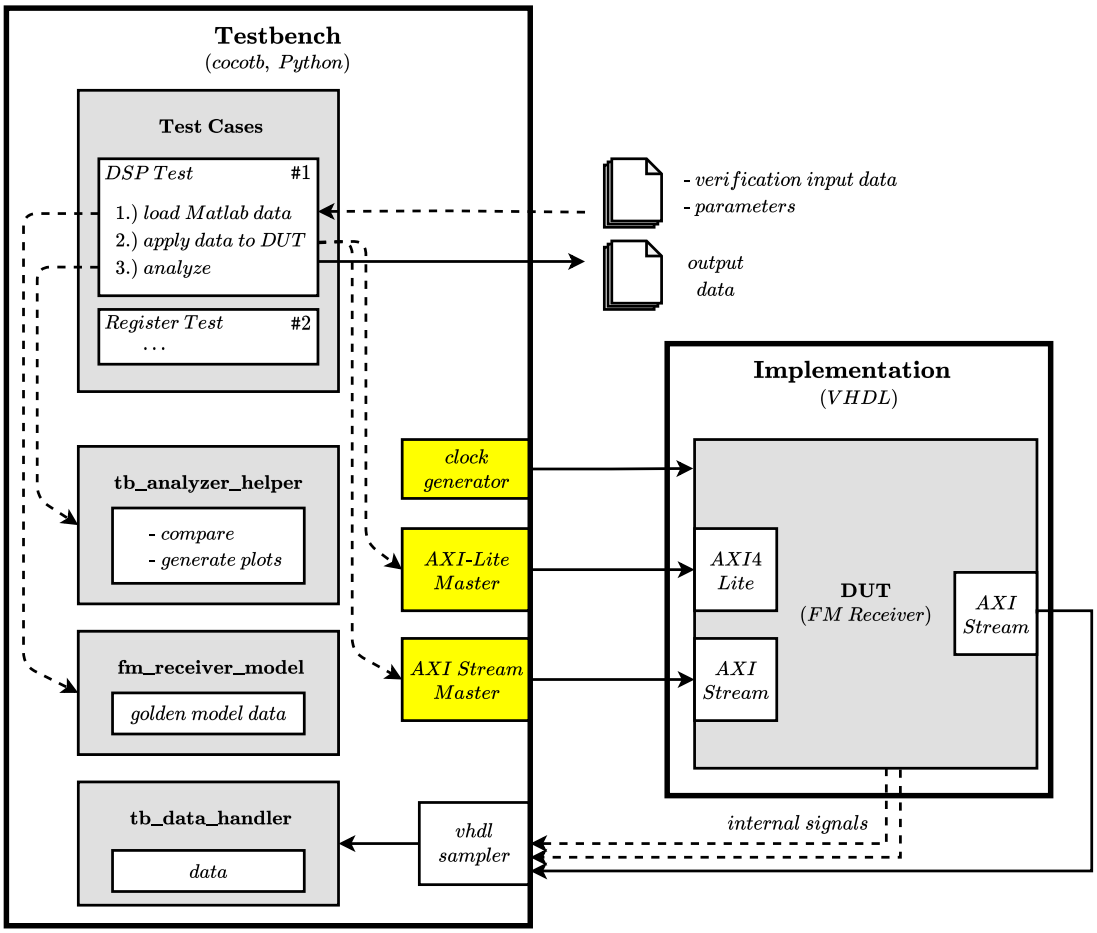


Figure 6.8: Block diagram of the implemented testbench architecture.

verify the combination of multiple units as a whole. Both of these stragegies are important. Given all unit tests being successful, it is ensured that all these single units are implemented correctly. However, their combination and interaction may still be subject to error – this is where integration tests unfold their potential. Implementation errors in the glue logic between the units, as well as the interaction of the respective interfaces can be found in these integration tests.

The testbench architecture presented above represents an integration test, since the DUT is tested as an entire block. The advantage thereof is, that it tests the DUT in the same way as it will be integrated within the target hardware, the FPGA. However, in case the DUT fails to produce a correct output in the integration test, it is not always visible where exactly the error is located. The source of error may be hidden anywhere within the nested and interconnected units. Thus, in addition to the DUTs' interfaces, further signals are logged from the inside of the DUT. These internal signals can be accessed through the Python testbench.

In this context, there are further keywords that need to be mentioned – black box testing and white box testing. These are testing strategies, and describe how the DUT is treated. In the case of black box, the DUTs internals, such as implementation details, are not known. Thus, only the exposed interfaces can be tested. In the white box as a contrary, every detail of the internal structure is known. Thus, even single signals can be accessed, which is what is done in this Python testbench. The advantage of the white box is, that these internals can be tested. However, any time the implementation changes, even if it is just the naming of a signal, the testbench code needs to be adapted as well. In black box testing, the testbench code is independent of the detailed implementation of the DUT, as long as it maintains its required behaviour at the exposed interfaces. However, the disadvantage is that internal states cannot be observed.

The Python testbench that is implemented combines the advantages of both worlds and uses black box testing, as well as white box testing.

6.4.4 Unit Tests

A unit test only targets a single unit, or module, of the system, as the name already describes. This is in the contrary to an integration test, which tests the combination of several units into an integrated system, as explained above.

In this project, the only module that has its dedicated unit test is the FIR filter. The reason therefor is, that it is the most complex unit within the entire design and is used in multiple instances. All the other design units mostly consist of simple delay lines, additions or substractions and thus a special unit test is not implemented.

6.4.5 Interfaces

The main interfaces of the FM radio receiver IP are two AXI4-Stream interfaces and one AXI4-Lite memory-mapped slave interface. The streaming interfaces are used for the input of IQ samples and output of the processed audio samples, while the memory-mapped interface provides configuration options and status information.

AXI4-Stream

An AXI4-Stream interface in the minimum configuration includes signals for the ready-indicator, and the data with its valid-indicator. The ready-indicator is used as a flag to signalize whether the interface can process data or not at this time, which allows handshaking between subsequent processing blocks. An IP may indicate that it is not ready, while its current operation is still running, for example.

In the VHDL version of the FM radio receiver IP, this handshaking is not implemented according to the AXI stream specification. Specifically, the ready signal handshake is implemented with a workaround, to simplify the development. The DSP chain of the IP is not fully pipelined and thus cannot consume an input sample on every clock cycle. This is mainly because of the FIR filter implementation – it takes multiple clock cycles to compute an output sample, before it can take a new sample. Ideally, this

situation should be handled by the ready signal, since this is exactly what it is intended for. However, in order to simplify the implementation a strobe generator is used, which is a counter that generates a repetitive pulse with a defined period. This pulse, in combination with some additional logic, is used as the ready signal, to guarantee enough spacing between two consecutive input samples.

AXI4-Lite Slave

The IP provides configuration registers and status registers on this memory-mapped bus interface. It is connected with the CPU, which uses these registers in the application software.

The VHDL code for this register interface is generated by a Python script which is named *Register Engine*. The registers can be defined in a YAML-file. Additionally, documentation in the form of a Markdown language file is generated, and – more importantly – a C/C++ code header, which defines all the register addresses and can be used in the application software.

Figure 6.9 gives an overview of how the Register Engine script works. The Python script takes the register definitions from the YAML file, and uses templates to generate the respecitive output files.

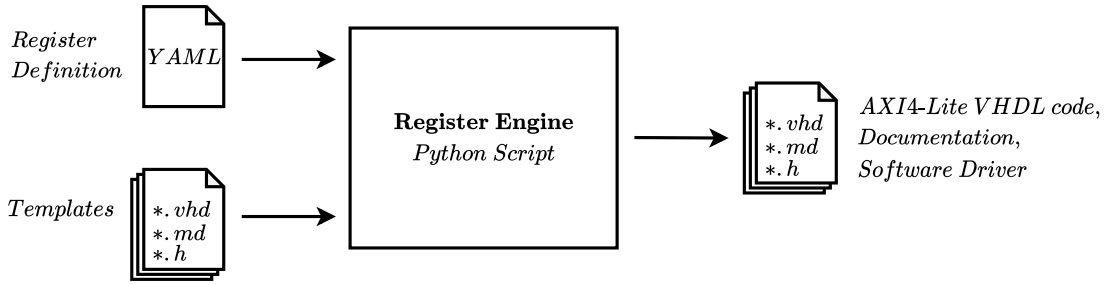


Figure 6.9: Overview of the Register Engine script.

6.5 High-Level Synthesis

This section describes the HLS version of the FM radio receiver implementation. The implemented testbench structure, as well as some specific implementation details are explained here.

6.5.1 Testbench

A testbench is equally important in the HLS implementation, as it is for the VHDL implementation. The HDL code for the IP is written in C++ and thus the testbench is also written in this language. The execution of the testbench is done in the Xilinx HLS software. A detailed explanation of how an HLS testbench works in general is given in

Chapter 4.

The testbench is developed in a similar manner as the VHDL testbench. It applies input data to the DUT and generates output data of multiple signals, which need to be observed. This output data is in turn saved to output files, which are analyzed in a subsequent step.

The actual implementation to write the HLS signal data to a file, is implemented as a file-writer C++ class within the HLS design code. These code parts are only evaluated during simulation, which is achieved by using pre-processor defines. The following code shows an example of this file-writer class.

```
1 audio_sample_t fm_receiver( ... ) {
2   ...
3 #ifndef __SYNTHESIS__
4   static DataWriter writer("data_out_audio.txt");
5   writer.write(audio);
6 #endif
7   ...
8   return audio;
9 }
```

6.5.2 Design For Reuse

The testbench is especially designed around the *Design For Reuse* segment of the SIMILAR concept, which is introduced in Section 3.1.1. This is the case because of two major parts.

Testbench

In the previous Section 6.5.1 it is explained, that the HLS testbench generates output files during its execution. These output files are analyzed with the exact same Python analysis code, that is already used in the VHDL testbench analysis.

Application Software Code

The testbench is developed in the language C++ and does not have any restrictions in the usage thereof, as Section 4.3 described. Thus, the C++ code that is later used in the application software to interface with the IP, can already be used in the testbench here, for the same purpose. This is especially useful in the development process of such a combined system, consisting of software and hardware, that relies on these two parts working together successfully. The fact that the application software can directly be used in the testbench, enables the development of software that interfaces with the hardware in a very early stage of the project, before any actual hardware exists. It also allows that the software part is already tested before the deployment on hardware.

Figure 6.10 gives an overview of the testbench and specifically points out the reuse of the application software. The diagram displays the HLS IP in the testbench and in

the hardware with dotted lines, to emphasize that they are actually the same.

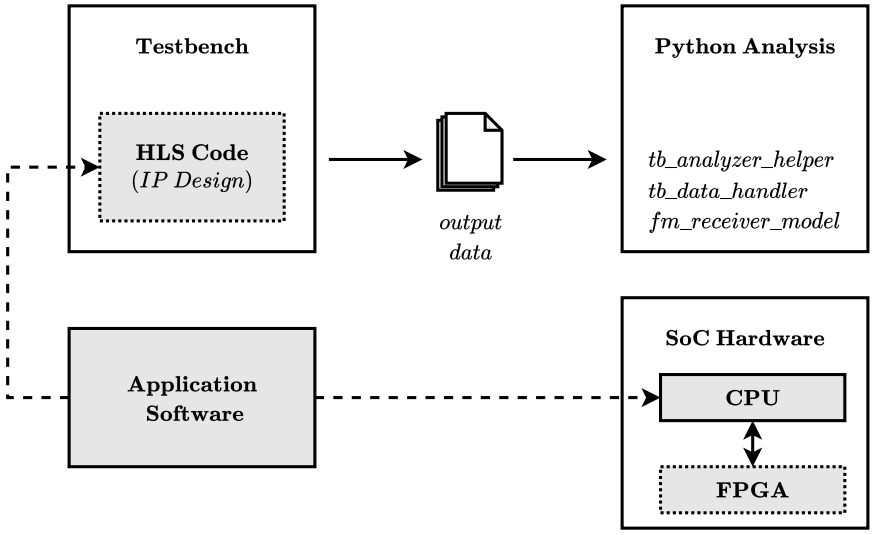


Figure 6.10: Overview of the testbench and the reuse of the application software. The HLS IP is displayed with dotted lines, to point out that it is the same in testbench and hardware.

6.5.3 Interfaces

HLS can automatically implement interfaces, as described in Section 4.4.3. This is a major feature that is used in the implementation of the HLS IP.

The design uses an AXI4-Lite memory mapped interface, as well as multiple AXI4-Streams. Additionally, a data-only interface is used for the connected LEDs. The actual implementation is demonstrated in Program 6.1. These few lines of code already implement the entire functionality of each interface type.

The streams use the types `iq_sample_t` and `audio_sample_t`, which are structs of two 16-bit fixed-point variables and thus represent a width of 32 bit. The configuration and status registers are also implemented as structs, that contain multiple variables to represent the separate registers. For the registers, a special detail is the distinction between the register types read-write and read-only. This difference is made by using a value-by-reference or a pointer value, for configuration and status interface, respectively. Consequently, the configuration registers become read-writeable and the status registers remain read-only.

The `bundle` keyword in the pragma-directives combine the two register groups into a single AXI4-Lite bus interface, instead of creating one bus each. This may be of advantage in the application software later, since it only needs to handle a single base address. The interface for the LEDs that can be connected is implemented as a data-only type, which does not have any protocol assigned. It simply represents the data value

Program 6.1: Implementation of sample rate reduction, using for-loops and AXI stream variables with their underlying FIFO behaviour.

```
1 void fm_receiver_hls(hls::stream<iq_sample_t>& iq_in,
2                       hls::stream<audio_sample_t>& audio_out,
3                       config_t& config,
4                       status_t* status,
5                       ap_int<NUM_LEDS>* led_out) {
6
7   /*------------ HLS interface settings ------------*/
8   #pragma HLS INTERFACE ap_ctrl_hs port = return
9
10  #pragma HLS INTERFACE axis port = iq_in
11  #pragma HLS DATA_PACK variable  = iq_in
12
13  #pragma HLS INTERFACE axis port = audio_out
14  #pragma HLS DATA_PACK variable  = audio_out
15
16  #pragma HLS INTERFACE s_axilite port = status bundle = API
17  #pragma HLS INTERFACE s_axilite port = config bundle = API
18
19  #pragma HLS INTERFACE ap_none port = led_out
20
21  ...
22 }
```

that is written to the C++ variable. The usage of a data type with variable bitwidth, such as `ap_int` in the example, is of advantage since only the actually required number of connections is implemented in the interface.

6.5.4 Sample Rate Reduction

Multiple steps in the DSP chain require the operation of sample rate reduction. This is explained in Section 6.1.

In the VHDL implementation, this is implemented with a counter that counts the number of incoming samples. If the counter reaches the number that matches the down-sampling rate, is forwards the current sample. Otherwise, the incoming samples are discarded.

In the HLS implementation however, this is done in another way. Here the previous block, which still runs in the oversampled domain, is executed N times, where N is the oversampling rate, before a single output sample is forwarded. This behaviour is possible because of the FIFO that is implemented in an AXI stream interface.

The implementation of the sample rate reduction can be shown in a short code example in Program 6.2. The FM Demodulator runs in the oversampled domain and is executed `OSR_RX` times in the inner for-loop. Only the last sample is written to the `fm_channel_data` variable, which itself is an AXI stream type with an underlying FIFO

Program 6.2: Implementation of sample rate reduction, using for-loops and AXI stream variables with their underlying FIFO behaviour.

```
audio_sample_t fm_receiver(hls::stream<iq_sample_t>& iq_in) {

  hls::stream<sample_t> fm_channel_data;
  #pragma HLS STREAM depth = OSR_AUDIO variable = fm_channel_data

  sample_t fm_demod;
  for (uint8_t i = 0; i < OSR_AUDIO; i++) {
    for (uint8_t k = 0; k < OSR_RX; k++) {
      iq_sample_t iq = iq_in.read();
      fm_demod       = fm_demodulator(iq);

      if (k == OSR_RX - 1) {
        fm_channel_data.write(fm_demod);
      }
    }
  }

  audio_sample_t audio_sample = channel_decoder(fm_channel_data);

  return audio_sample;
}


audio_sample_t channel_decoder(hls::stream<sample_t>& in_sample) {
  sample_t audio_mono;

  for (uint8_t i = 0; i < OSR_AUDIO; i++) {
    sample_t sample = in_sample.read();

    audio_mono = recover_mono(sample);
    ...
  }
  ...
  return audio;
}
```

depth of `OSR_AUDIO`, as defined with the accompanying pragma. Thus, the inner for-loop is executed `OSR_AUDIO` times by the outer loop, to fill this FIFO until it is full. The subsequent Channel Decoder then reads the entire FIFO and also implements downsampling with a similar behaviour. Several samples are processed, where only the last one is forwarded to the subsequent processing step. The FIFO behaviour of AXI stream variables is described in Section 4.4.3.

Chapter 7

Deployment on Hardware

This chapter describes the additional parts of implementation that are necessary in order to run the implemented FM radio on actual hardware.

7.1 GNU Radio

The GNU Radio implementations, both receiver and transmitter, are used in combination with external hardware. The specific hardware that is used is briefly described here.

In order to interface with the real world, two different hardware devices are used. Both devices are supported by GNU Radio and are briefly described here.

7.1.1 Ettus Research USRP b200mini

The USRP b200mini is a device that can be used to create a software-defined radio. It resembles the size of a business card and is developed by Ettus Research. A signal range from 70 MHz to 6 GHz can be covered by the front end. The board further features a Xilinx Spartan-6 FPGA that is user-programmable, which makes the device very flexible for various applications with the received signal [36].

GNU Radio supports this device with the blocks *UHD: USRP Sink* and *UHD: USRP Source*. Therefore, it can be used as both a receiver and a transmitter.

7.1.2 RTL-SDR

The RTL-SDR is a cheap USB dongle, that was originally developed as a DVB-T TV tuner. It is based on the RTL2832U chipset, which includes an 8-bit ADC and a digital signal processor. The device delivers IQ data via the USB interface [38].

Based on its hardware, the RTL-SDR can only be used as a receiver.

7.2 FPGA Hardware Platform

The ZedBoard is a development board that is developed by Xilinx. It is specifically designed around the Xilinx Zynq-7000 SoC, which contains a dual-core ARM Cortex A9 processor and a programmable logic fabric, the FPGA. The board further offers a range of features from external DDR3 memory, over various connectors, such as RJ45 Ethernet, USB, 3.5 mm audio, HDMI, Pmod, to buttons, LEDs and a display.

The most important features for this project are the audio line-out connector and the SD card port. Additionally, the LEDs are used to display various status information during runtime of the FM receiver.

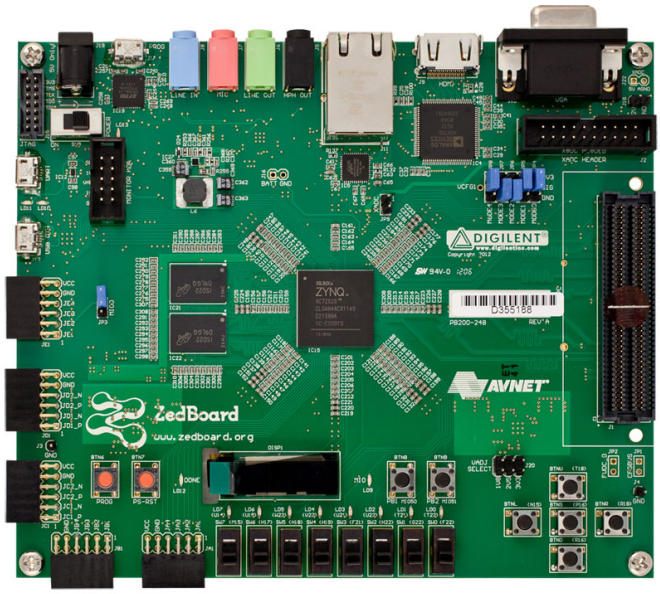


Figure 7.1: The ZedBoard PCB [26].

7.3 Product Design

The implemented version of the FM Radio Receiver on the ZedBoard is a working prototype at this point. The implementation does not support live data streaming. Instead, antenna data is recorded to a file with the RTL-SDR hardware, using a regular PC. The antenna data is recorded as IQ data in the baseband. The file is then stored onto an SD card, that is plugged into the ZedBoard. The CPU on the SoC then reads it and replays the antenna data to the FPGA. In the future, this design may be extended to use an embedded Linux operating system. In that way, the RTL-SDR could be connected to the ZedBoard PCB directly via USB, which would enable a live data stream from the antenna to the FPGA. However, the chosen solution is sufficient for the aims of this thesis.

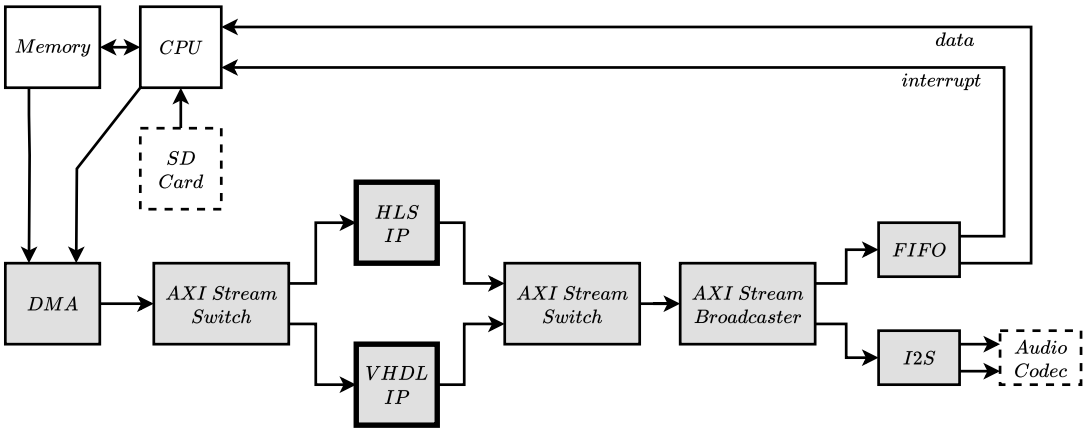


Figure 7.2: Block diagram of the Vivado implementation.

7.4 System-On-Chip Design

Chapter 6 already described the development of the main IP core, which is the FM radio receiver in the VHDL and HLS variants. In order to embed these IPs on the actual FPGA, or rather SoC hardware, a significant additional amount of implementation is neccessary. This section describes these additional parts of the implementation, which specifically enable and support the deployment on the SoC.

7.4.1 Architecture

The block diagram in Figure 7.2 shows the implemented architecture that is used in the SoC. The various design domains in and around the SoC are depicted in different visual representations. The FPGA domain blocks are highlighted in gray, with the main IPs being surrounded by emphasized lines. The processor system domain is shown as white blocks. Blocks that are external to the SoC are drawn with dashed lines.

7.4.2 Functionality

The FM receiver IP is the main functionality of the design. However, the design around it needs to function properly in order for the IP to work correctly, which is explained in the following passage.

The CPU is the main actor that determines any actions on the system. First, it reads a file from the external SD card and stores this data in a dedicated area in the memory. Next, the DMA is configured and its data transfer mode is initiated, so that the DMA starts to transfer data from the mentioned memory area towards the FPGA. The DMA is configured in a mode that creates a continuous transfer, by looping through the data – once the end is reached, the transfer starts at the beginning of the memory area. Any data connection in the following processing chain is implemented as an AXI stream bus. The first block after the DMA is an AXI Stream Switch. It can switch its single input to be forwarded to either of the outputs. The switch is configured by the

CPU, to enable one output at a time. Consequently, one of the FM radio receiver IPs is provided with data. The IP processes this data to produce an audio output, which is delivered to another AXI Stream Switch. It fulfills the function to either select the output of above or below IP, in coordination with the first switch, obviously. The next block is an AXI Stream Broadcaster, which takes a single stream input and simply duplicates it at its output, so that the outputs are an exact copy of the input stream. One of these duplicates is sourced to an AXI-Stream-To-I2S converter, which generates the I2S signals for the external audio codec chip. The second duplicate is connected to a FIFO, which simply stores the generated output data. Once the FIFO is almost full, it asserts an interrupt to the CPU, which triggers it to read out the entire FIFO buffer. The CPU then stores this data on the external SD card again. In that way, the loop of data is closed, which enables the verification of the respective IPs' correct functionality. Therefore, this system design can be described as a hardware-in-the-loop system, in the point of view of the FPGA IPs. This ability refers to the knowledge gained with the SIMILAR concept, that is introduced in Section 3.1.1. Specifically, this implemented system design enables the application of the steps *Assess Performance* and *Re-evaluate*, since the IPs' performance can be measured and their design can be re-evaluated.

It is to note, that Xilinx provides all the AXI infrastructure IPs, and thus the only self-implemented IPs are the FM radio receivers, as well as the I2S interface. Nevertheless, the correct design and implementation of the architecture still needs to be formulated by the user.

7.4.3 Software

The application software is running on a single core of the dual-core ARM Cortex A9 CPU and is built upon a real-time operating system, namely FreeRTOS. Xilinx provides a ported version of this RTOS with the installation of their software development kit, the Xilinx SDK. The SDK also includes drivers to support various functionalities, such as support for accessing a FAT file system on the external SD card.

The respective software drivers for several IPs that are used in the FPGA design are automatically imported into the SDK project. They are embedded in the so-called hardware definition file (HDF) that is generated by Vivado after the synthesis process, and provided together with the bitstream file. It is a file with the extension **.hdf* and can be opened like a regular archive file, with any ZIP archive viewer.

Generally, the application software for this project is developed as an object-oriented C++ program. It uses common software development patterns and a class structure that fits the purpose. The main tasks of the application are to configure several IPs in the FPGA, so that they can fulfill their respective purpose in the system, and to read the recorded data from the SD card, which is then streamed to the FPGA. It also provides a basic user interface over the serial console, with which a user can interact with the FM Radio Receiver product.

However, software design is not the main focus of this thesis, and thus no further

details are discussed here.

Chapter 8

Comparison and Results

This chapter presents the results that are achieved in the development of the FM Radio Receiver project, with the chosen different methods. However, the main focus is to compare the results that are achieved with HLS and VHDL, since these are the major implementation variants of the project.

8.1 General

A main objective of this thesis is to implement an FM radio receiver in multiple different methods. This includes GNU Radio, Matlab, VHDL and HLS, which all have their advantages and disadvantages and surface different challenges to the developer. The target functionality is to listen to the audio broadcast signal of an FM radio station.

All of the listed methods are implemented and the target functionality is achieved with all of them. However, the level of detail that is required in the implementation, and the resulting effort and time it takes to implement the respective variant, differs by a large factor. This is mainly due to the different levels of abstraction, so that the low-level algorithms do not need to be known.

In the following sections, especially the HLS and VHDL implementations are compared on the basis of metrics, but also based on the experience during development.

8.2 Functionality

Generally, both tools – HLS and VHDL – provide the capabilities to implement any functionality in one or the other method. However, the implementation is done in a different level. VHDL uses an approach that is very close to the hardware, such as clock cycles and flipflops, while HLS describes the logic on an algorithmic level.

The implementation is split into two main parts, the communication interfaces and the DSP. The results are presented in the following sections.

8.2.1 Interfaces

The AXI4-Lite memory-mapped bus is implemented to have an exactly equal behaviour in both variants. There are read-only and read-writeable registers, which are all mapped onto a single base address.

The AXI stream interface however does not show the exact same behaviour. Here, the HLS variant is using the AXI stream according to its protocol, while the VHDL variant is implemented differently. It uses a simplified logic for the ready-flag, which reduces the effort in implementation. However, from the perspective of the communicating blocks, the interface is usable like a regular AXI stream. The implementation details are explained in Section 6.4.5.

In the final, integrated system on the SoC hardware, the CPU is able to communicate with both IPs via their interfaces successfully. Status and configuration data can be read and written through the AXI4-Lite interface, and the streaming data for the DSP chain is successfully sent through the AXI streams as well.

8.2.2 Audio Output

The audio output of the HLS and VHDL variants is compared with the Matlab model, which serves as the reference. Additionally, the results of the testbench are compared with their respective result in the actual hardware. In summary, it can be stated that the DSP chain produces a very similar audio output in all the compared data sets. However, differences remain, which are presented in the following paragraphs and diagrams.

Comparing against the Matlab reference

In the comparison against the Matlab model, the HLS variant achieves a very exact match of the output signal. However, the VHDL implementation differs by a certain amount. There seems to be an issue in the signal separation between the left and right audio channels. This is visible in the analysis of the audio signal, as shown is Figure 8.1. The signal strength of the left channel in the upper diagram is weaker than expected, while the right channel in the lower diagram is stronger. This imbalance can also be observed by listening to the audio output via speakers.

The cause for this issue may be located in several components in the design. This includes the FM demodulator, the carrier recovery, i.e. the 38 kHz carrier, as well as a sample timing shift in the final summation and substraction to recover the left and right channels. The FIR filter has a successfully passing unit test and is thus assumed to be correct. Also the fixed point data type with its overflow- and rounding behaviour is suspected to be a potential issue. Due to the time limitation in the elaboration of this thesis, this issue can not be traced down to the root cause and thus still persists in the current implementation. However, from the system-level point of view of this thesis, the IP works and can be integrated into the final system.

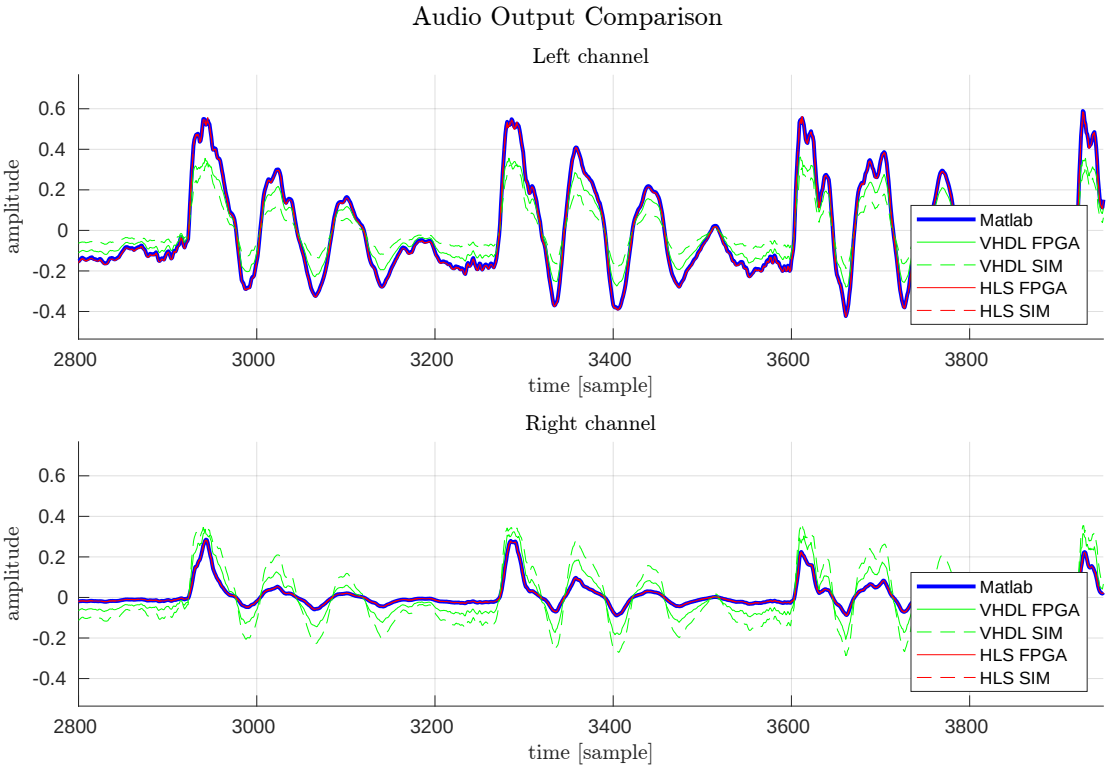


Figure 8.1: Comparison of the IP audio output signals, in simulation and hardware, against the Matlab model. The HLS variant matches the model, while the VHDL output diverts by a certain amount. Also, VHDL differs between simulation- and hardware results.

Comparing simulation- against hardware results

Here, the results of the simulation testbenches are compared against the values that are read from the FPGA directly. Ideally, the values should be exactly the same. However, Figure 8.1 shows that this is not always the case. Again, the HLS implementation does have a matching result, whereas the VHDL variant has deviating values. The explanation here may be linked with above suspicions, but may also be linked to the initial reset values that exist in the hardware. In the HLS implementation, the reset logic of several registers is implemented automatically, whereas in VHDL these conditions need to be implemented manually. Therefore, the IPs internal state, i.e. the register values, may be different at the beginning, which leads to an error propagation throughout the design. The entire DSP chain is built like a pipeline and therefor it takes a number of samples to process, before the wrong, intermediate values are flushed out. All that may be the cause for the deviating values in the VHDL design.

Again, the time limitation in the elaboration of this thesis does not allow a deeper analysis and investigation of this issue. However, the end-to-end system design enables the developer to analyze an issue like this in a convenient way, in quick iterations. Any adaptions in the VHDL design can be simulated and integrated into the FPGA with automated scripts. Both results can then be analyzed and compared, which may lead

to further adaptions.

8.3 Code Development

In the development of a software project, the metric for the lines of code can be used to give an idea of the scale, or volume of the project. The time it takes to implement a certain set of features can also give such an insight and a hint of the project's complexity. It is a subjective metric though, since it is heavily influenced by the respective developer's experience in the specific application. However, both metrics are applied to the FM Radio Receiver project.

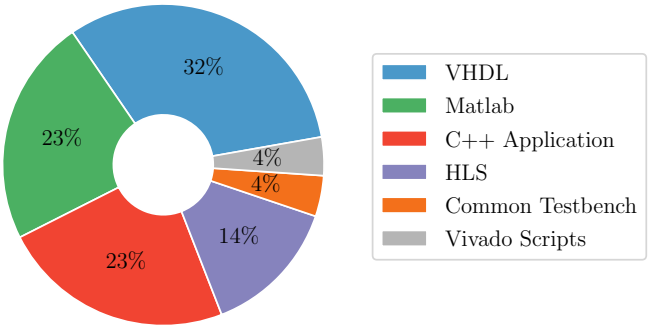
At this point, it is important to note that the results that are presented in this section are very specific to the chosen architecture, testbench framework, and the implementation style generally. Therefore, the numbers that are presented here may vary widely by modifying any of these factors, i.e. using a different testbench framework for VHDL. Thus, a generalization in the art of *HLS always requires less code than VHDL* can not be stated. However, the results in this section can give an idea and overview of this topic and provide a detailed comparison for this specific project.

8.3.1 Lines of Code

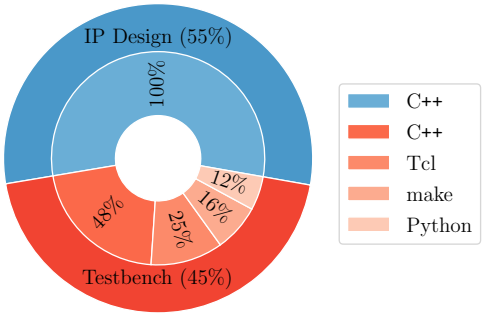
Several parts of the system design are taken into account to analyze the lines of code, to give an impression over the largest code parts in the project. Figure 8.2 shows multiple pie charts, which represent the proportions of the lines of code in the respective system design parts. Table 8.1 displays the actual numbers of code lines, to quantify the size of the respective parts.

Interesting details can be found by analyzing these charts. The upper chart shows that the implementation of the hardware support, in the form of Vivado project scripts and the C++ application software, takes a large part of the code with 27%. The remaining majority of the code belongs to the development of the IPs. This includes the implementation of the Matlab model, the VHDL and HLS code, as well as the common testbench. The chart also reveals that the VHDL implementation with its accompanying testbench requires more than twice the amount of code compared to the HLS version, i.e. 32% versus 14%.

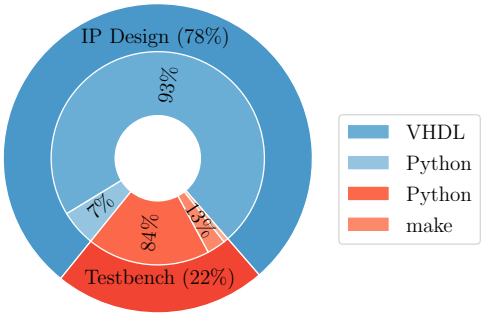
The lower charts highlight several differences between HLS and VHDL. Specifically, the code proportions between IP design and testbench are pointed out. The charts show that HLS requires about the same amount of code for IP design and testbench. VHDL requires the largest part of code in the IP design, while the testbench code amount is relatively small. It is to be noted, that the common testbench consists of 4%, which equals 630 lines of Python code. Therefor, the testbench code of both IPs is relatively small, because the entire analysis part is shared.



(a) Overview of the proportions between all the implemented code parts of the entire project.



(b) HLS in detail.



(c) VHDL in detail.

Figure 8.2: Lines of code of the respective code parts in the entire project, presented in multiple pie charts. The percentages of code in IP design and testbench for the HLS and VHDL implementations are shown in detail in the lower diagrams.

8.3.2 Implementation Time

The metric of implementation time – the time it takes to implement a specific feature – is a subjective measure, since it heavily depends on the respective developer's experience, as already noted above. In the present case, there is a strong background of knowledge and practical experience in VHDL design, while there is only very little of such in HLS design. However, this metric is used in this section to specifically compare HLS with VHDL in this project. Certain features are chosen for this comparison and their implementation times were logged during the actual implementation process. The implementation is always considered as the HDL description, including the corresponding testbench code, to verify the respective correct functionality. Table 8.2 shows the direct comparison between HLS and VHDL implementation times for these features.

The following paragraphs present an insight into how the implementation times of the listed code parts arise.

	IP Design	Testbench
VHDL	3000	850
HLS	920	740
Common Testbench	630	
Matlab Model	2730	
Vivado Scripts	490	
C++ Application	2800	

Table 8.1: The actual number of code lines for each code part, split in IP design and testbench, where applicable. All numbers are rounded up to tens.

	Time [h]	
Feature	**HLS**	**VHDL**
AXI Stream Interface	8	12
AXI4-Lite Interface	2	10
LED Control Register	1	2
Build Information Register	8	5
Pass-Through Mode Register	1	1
FIR Filter	13	7
Sample Decimation	15	1
Receiver Structure	13	33
Sum	**61**	**71**

Table 8.2: Implementation time logging for a specific set of features, for the direct comparison between HLS and VHDL.

AXI Stream Inferface

This is the main interface for the data input and -output of the DSP chain. In HLS it can be automatically inferred by adding pragma directives to the respective function parameter variables. Also the testbench code is implemented like a regular function call. Therefore, the actual interface logic is completely hidden from the developer. In VHDL on the other hand, this logic has to be implemented manually. Also the testbench has to be hooked up with the correct signals and have a verificator logic, that supports the functional verification of an AXI stream interface. Otherwise, this logic would have to be implemented manually as well. In the time comparison, the larger amount of manual implementation in VHDL is clearly visible. It is to note that this comparison does not compare functionally equivalent implementations, because the actually implemented logic in VHDL does not use the ready-flag correctly, as explained in the previous chapters. According to this fact, a further increase in implementation hours should actually be considered for VHDL. However, also the HLS implementation has its particular characteristics that need to be understood for a fully functional implementation. Following the respective user guides and implementation examples provides a good starting point for that.

AXI4-Lite Interface

This interface is used to control IP settings and to read status information. Similar to the previous AXI Stream interface, also this AXI4-Lite memory-mapped interface is automatically generated by HLS, while this has to be done manually in VHDL. To create this process more efficiently and extendable, a Python script is used, which can also automatically generate VHDL code for the registers' memory map. This turns out to be a multiplicator in productivity for the implementation of registers. The register generator script was previously used in projects and is therefor already well-known and tested before it is used in this project. However, the VHDL code only represents one part of the interface – the second part is the firmware driver that is used in the application software. Even though the register generator script also generates a C++ header file that includes several address offset definitions of the registers, the actual driver functions need to be implemented manually. Several files need to be integrated into a pre-defined folder structure, in order to enable a correct integration of the IP in a Vivado block diagram.

Again, the amount of manual steps in the VHDL implementation are clearly represented in the number of hours that are spent.

Registers

Once the AXI4-Lite memory-mapped interface's main functionality is implemented, further registers can be added. In the specific implementation for this project, registers for LED control, the DSP mode, as well as a build information register are added. The latter holds information about the build date and -time of the IPs, and is particularly time-consuming because of the implementation of the respective drivers to convert the date format in both variants. In contrast, the addition of the LED control and DSP mode registers is simple and fast, because of the auto-generated code in HLS and VHDL with usage of the register engine. The single bits in the LED control register directly represent the LEDs' values on the hardware. The DSP mode register can select the regular FM radio mode and a pass-through mode, which simply forwards the streaming data without any processing. Summing up, the addition of registers to the design can be time-consuming in both variants, depending on the required driver functionality. However, the general tendency is that VHDL requires more time, because of this driver issue.

FIR Filter

The FIR filter is one of the main DSP blocks of the design, as it is used in multiple instances throughout the design. Therefore, its implementation was done with special care, in order to guarantee its correct functionality. Consequently, a unit test is implemented in the testbench, which is also included in the listed number of implementation hours.

For the C++ version, a code example for an FIR filter, which is provided by Xilinx HLS, is taken and modified for the present purpose. The VHDL variant is taken

from an older DSP project, which was previously developed during a university course. However, the integration and testbench code is developed for this project, specifically. Here, the HLS implementation took significantly longer, because of the limited knowledge regarding the usage of an AXI stream interface. It is the first time in the project, that this type of interface is applied in the IP design and used in the testbench. The VHDL implementation was straightforward in comparison. The testbenches compare the FIR filters' output with the Matlab model's pilot tone output. Both variants match the output results of the Matlab model successfully.

Sample Decimation

The implementation of the sample decimation functionality differs slightly between the two variants, HLS and VHDL. This is explained in Section 6.5.4. The implementation time of the HLS version is severely larger than the VHDL version. This is mainly due to the limited knowledge of the functionality of the AXI stream interface at this stage in the project's implementation, just as previously mentioned with the FIR filter. The initial version was implemented like the VHDL version – a counter that only triggers the subsequent DSP blocks upon a certain number of samples was received. However, this led to an endlessly blocking AXI stream in the testbench execution. The reason was, that the read-function can only read a sample when the respective stream contains a sample, and that it is a blocking function. Because the sample decimator did not write a sample every time, the subsequent read-function blocked indefinitely. A thorough investigation in several user guides, in combination with step-by-step debugging, the issue was found and led to the current, working implementation, which processes a number of samples, before it forwards one sample. The process of finding this issue explains the largely varying implementation times, as listed in Table 8.2.

Receiver Structure

The implementation of the receiver structure requires the combination of several previously implemented blocks, such as the FIR filter and any interfaces. This is done according to the Matlab model, to resemble its functionality. The VHDL version is significantly more time consuming in comparison to HLS. However, it is to note at this point that the VHDL version is implemented first, and thus some issues are already solved by the time of the HLS implementation. Nevertheless, the VHDL implementation takes longer, as it requires more detailed care in the signal level. This specifically includes the clock-alignment of any signals that need to be used in a combination. For example, if two parallel FIR filters produce output signals that are to be added, they need to be aligned so that they are valid at the time of the summation. This often requires analysis and debugging in a signal wave form. In HLS this is not required, since the validity of a signal is determined internally by the HLS compiler, according to the usage of the respective data variables. Another factor that adds up to the implementation time is the number of lines of code, as presented in Table 8.1. The VHDL IP design requires 3000 lines, while the HLS IP design only takes 920 for the same functionality. Again, this is a subjective measure – several factors such as code style, or the developer's experience play a role – but still shows a good representation of the implementation effort for this project.

In the final summation over several implementation times of all the listed features, the two variants HLS and VHDL only differ by 10 hours, which is around 15%. However, as the previous paragraphs explain, prior knowledge regarding HLS was limited and had to be gathered during the ongoing implementation. Nevertheless, even consiering this fact, the HLS implementation is faster than the VHDL equivalent. Thus, with an equal amount of prior knowledge and experience, it is assumed that an HLS implementation can lead to a successfully functioning result significantly faster.

8.4 Simulation Speed

In this section, the simulation speed of the multiple implementation variants in their various abstraction levels are compared. Simulation speed in this context refers to the number of input samples that the DUT can process in a time unit, like one second. In the current application, the antenna signal is recorded and replayed to the DUT in the simulation. This ensures a reproducable result, so that the different variants can be compared against each other reliably.

Figure 8.3 shows this direct comparison of processed samples per second for the implementation variants. The measurements always include the testbench setup-phase, and the run-phase. The setup-phase mainly consists of loading the recorded data and its preparation, so that it can be fed into the DUT. The run-phase then actually applies this test data to the DUT and stores the respective output results. All time measurements are taken without the result analysis, since this is the exact same, shared code between all variants and thus takes the same amount of time everywhere. Several measurements are done multiple times, to achieve an average result, that is independent of the current processor load of the host PC.

The diagram shows an immense difference of simulation speed between the variants. It is important to note that the y-axis has a logarithmic scale, as a method to enable the presentation of the values in multiple orders of magnitude. VHDL only achieves around 20 samples per second. This is by far the slowest method, which is expected because of the way of how VHDL is simulated. This can not be explained in detail here, but one main feature of a VHDL simulator is its capability to simulate parallel processes, as they occur in actual parallel hardware. The VHDL simulator therefor needs to evaluate several concurrent processes and statements, whenever any of the interacting signals change their value, and this is time-consuming. In the current case, there is an additional interface between the VHDL simulator and the Python testbench framework, which likely additionally slows down the simulation. The HLS testbench in comparison achieves around 50000 samples per second, which is a speed-up factor of 2500. This is specifically talking about the HLS C/C++-simulation. The main reason for the speed-up therefor is, that in this type of simulation no timing behaviour is simulated – any processes are treated as C/C++ functions that runs like a regular executable on the host CPU. Also, the HLS compiler is using native C++ data types to represent the signals, where possible, which results in the faster execution of the code. Finally, the simulation in Matlab achieves a further speed-up by a factor of 70. Matlab is specifically

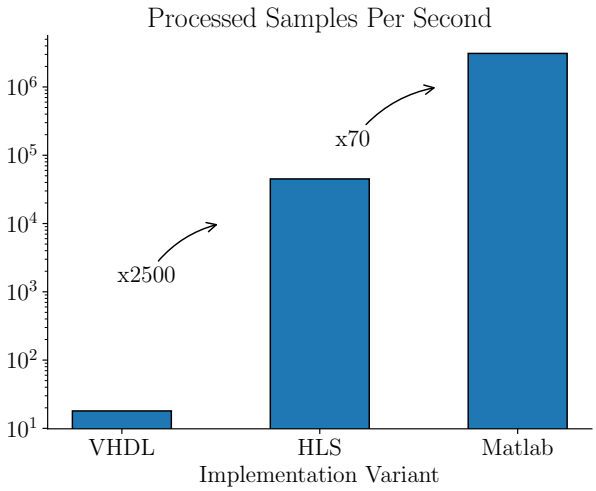


Figure 8.3: The measured simulation speed of the different implementation variants, in samples per second. The arrows show the speed-up factors. Note the logarithmic scale of the y-axis.

designed for applications which require matrix calculations and utilizes highly optimized algorithms in its underlying functions therefor. Thus, Matlab runs very efficiently and achieves the highest simulation speed of around 3 million samples per second.

8.5 System Design

The system design architecture that is chosen in this project covers several parts of the system. This includes a reference model, the development of the IP itself, including the simulation testbench, as well as the final integration into the productive system. Because of this modular and structured approach, that is given by this system design, it can be applied to almost any DSP project. The main processing IP can simply be replaced with another implementation, while the environment does not need to be modified. The reason therefor is, that DSP projects almost always use streaming interfaces for their inputs and outputs. Also a configuration interface is commonly used, to adapt certain parameters of the IP during runtime.

8.6 Deployment on Hardware

The VHDL and HLS implementation variants are implemented, simulated in their testbenches to verify the correct functionality, as well as deployed on hardware, to test their performance in an actual application. Important parts of this end-to-end integration are presented here.

8.6.1 Integration

The aim is to create IP blocks, that can be integrated into a block design in the Xilinx Vivado software. The IP blocks should be able to be connected with their AXI interfaces, in order to successfully communicate and interact with the embedded hard-processor, the ARM CPU. In the case of HLS, several files that are required for this integration are generated automatically. The files are already placed into the specific folder structure, that Vivado requires to recognize it as an IP block. In the case of VHDL on the other hand, this needs to be done manually. There is an assistant, the IP Packager, which helps the developer to create the IP. However, at this point the folder structure already needs to be in place, so that the assistant works correctly, which again requires manual action. Regarding the top-level interfaces, the IP Packager supports an auto-detection of the signals that belong to the respective interface types, as long as they follow the expected naming guidelines. However, the interface itself needs to be defined and updated manually in the IP Packager. This is especially important, if the interfaces change in the VHDL code.

However, also the auto-generation of the HLS IP has characteristics that require special attention. One issue during the implementation was, that the auto-generated software driver did not get imported into the Vivado SDK. The reason therefor was, that the top-level function of the IP did not have the exact same naming as the Vivado HLS project. For issues like that, the Eclipse-based SDK does not give any error output or any kind of information. This however is a general issue with the SDK, where error handling is poor and error log outputs are not existent, or at least not useful to the developer, so that the SDK sometimes crashes without an obvious reason. It is to note, that version v2018.2 is used for the entire Vivado toolchain, which is not the latest version that is currently available. Thus, some of the occuring issues may already be fixed in the newer versions.

However, both IPs could successfully be integrated into the Vivado block design, and the tests on the hardware show that they function correctly within the system.

8.6.2 Hardware Utilization

The hardware utilization of the HLS and VHDL implementations are compared in detail in this section. The numbers are taken from the final utilization report, that is created by Vivado at the end of the bitstream creation, since this is assumed to be the most accurate report that shows the actual utilization.

Figure 8.4 shows the utilization numbers of the different FPGA resources. It reveals that the manual VHDL implementation clearly requires more resources in the FPGA. The study of the detailed utilization report however does not enable a deeper direct comparison of the respective single blocks, such as VHDL FIR filter against HLS FIR filter. This is because the HLS compiler automatically applies optimizations, so that the direct connection between code and resource report is partially lost. Examples for optimizations are inlining of functions, re-ordering of operations and time-sharing. The latter has an especially large impact on the resource utilization. Generally, in hardware

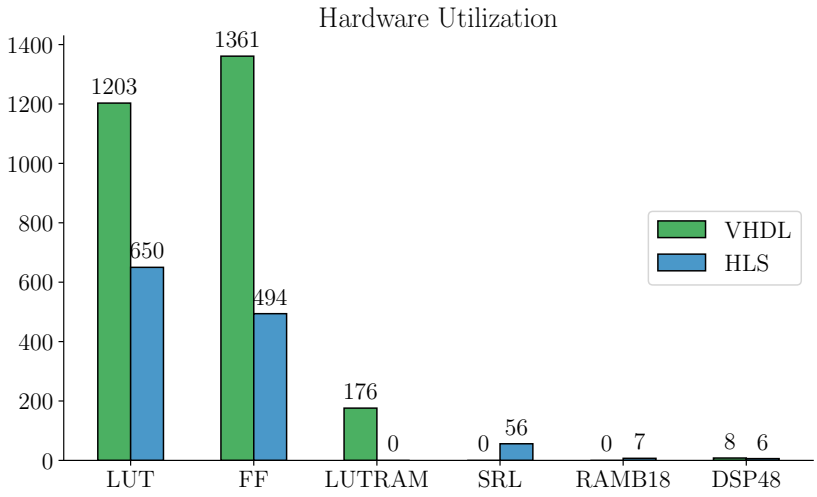


Figure 8.4: A comparison of the resource utilization of the VHDL and HLS implementations. The VHDL variant requires significantly more resources.

design there is a trade-off between resource utilization and throughput, which leads to time-sharing. For example, an entirely parallel-duplicated pipeline can process the double amount of data, but also requires the double amount of resources. On the other hand, time-sharing a resource may take the double amount of time, thus only processes half of the data, but also only requires half of the resource utilization. In HLS, this trade-off can be influenced and determined by the developer. Depending on the requirements of the design, the optimization may either be set towards a lower resource utilization, or towards a high throughput. In the current HLS implementation, no measures are taken towards any optimization and thus, the compiler is free to decide. However, the only requirement is the correct functionality within the given situation, of course. This specifically means that the throughput needs to be high enough, in order to process the audio signal output accordingly.

Nevertheless, the resource report allows the comparison of the configuration interface, which is the AXI4-Lite interface. In VHDL, it requires 44 LUTs and 54 FFs, while HLS consumes only 36 LUTs and 37 FFs. So it seems that the manual VHDL implementation requires more resources in all the comparable parts. By further investigating the report, the VHDL version apparently utilizes a large amount of LUTs for the delays, which are used to clock-align the FIR output signals. For an optimization of the VHDL design, this would be a good point to start at. However, for this project, both implementation variants are compared without any optimizations from the developer's side.

8.6.3 Latency

Latency is the time it takes for the design to produce an output sample. The latency of the HLS version is taken from the report that is produced after the C-synthesis of the HLS compiler. The report shows an estimated resource utilization, as well as the latency

as a number of clock cycles. The latency is given with 2712 clock cycles. For the VHDL version, a similar report is not available. Instead, the testbench of the integration test is executed, which produces a signal wave form. The latency is measured from this wave form and results in a number of 2470 clock cycles. With regards to latency, the two implementation variants appear to have a similar behaviour.

Chapter 9

Conclusion and Prospect

This thesis covers a comprehensive variety of topics, that ranges from digital signal processing to system modeling and -design, IP design in different methods, verification, to hardware-software interaction in hardware. The implementation of the FM radio receiver in multiple different methods, namely GNU Radio, Matlab, VHDL and HLS so that they achieve the same functionality, is a challenge, especially with the latter methods. However, this challenge is solved within an end-to-end system design, that strongly supports the development process.

GNU Radio is a software, that allows an efficient development of a software-defined radio as a prototype, which can run on actual hardware in a live environment. It is definitely worth to use it during the development of a project like the one in this thesis, since it allows to achieve a functioning result with relatively low effort.

The usage of Matlab as a tool to research DSP algorithms and strategies to demodulate a signal, in order to ultimately find a system model that can be implemented in hardware, turns out to be a powerful asset. It allows the development in fast iterations, through short simulation times, to achieve a result quickly. Based on this system model, the implementation of actual hardware IPs, with VHDL and HLS can be done more efficiently and in a direct way, because issues in the algorithm level are already fixed in the model at this point. In combination with the simulation testbenches that share major parts, such as the input source, as well as a the analysis tool, the IP development works smoothly and efficiently. The verification as a comparison against the Matlab system model gives instant feedback of whether the IP functions correctly. Even if the system model changes, the testbench automatically adapts, since the input source files are re-generated by the model and thus picked up at the next iteration of the testbench simulation, so the hardware engineer can also adapt the IP code. Once the IP is deployed on actual hardware, it is tested by the firmware, in order to check its correct functionality again, which closes the loop from the system model and the IP simulation to the hardware deployment. This entire process proved itself to be a successful approach.

Regarding the implementation of the VHDL and HLS IPs and the accompanying comparison, the result turns out to be unexpected. The initial expectation, that HLS is inefficient in terms of resource utilization, is contradicted. Also the amount of time

spent in the implementation and the metric of lines of code create a positive picture of HLS. Further, the automatic generation of several files that are required to integrate the IP into a system design, such as interface definitions or software drivers, support this attitude. VHDL may have its advantages in specific areas of application, but HLS is definitely a powerful tool that can achieve great results in less time. This is especially true, when it comes to the development of complex functionalities, such as image- or video processing, with the support of built-in libraries like OpenCV.

However, the often heard statement that HLS enables any software developer to develop hardware IPs, can not be supported. A developer that writes HLS code, still needs to have a strong hardware background, to be able to understand what is being generated in hardware. It is important to know about the limitations of the target hardware, i.e. FPGA, in order to write an efficient design. Therefore, the connection between specific HLS C++ code lines, as well as the general structure and algorithm, and the hardware target need to be understood.

In prospect, there are many ideas on how the project can potentially be extended. Additional implementation methods could be used to develop the FM radio receiver IP. For example, Matlab can be used to create VHDL code, in a high-level synthesis fashion, which can be used to create the IP. Regarding the verification environment, also the GNU Radio implementation could be included. The common input that is produced by the Matlab model, and the common output analysis, which are both used for the VHDL and HLS testbenches, can be used in GNU Radio as well. This would allow another comparison. Looking at the HLS code, it could be evaluated how compiler-independent it is. The question could be, if it is possible to write code that can be compiled with multiple compilers for different target platforms and vendors, e.g. Xilinx and Intel FPGAs, at the same time. Furthermore, the various optimization strategies that Xilinx HLS provides could be evaluated on the existing HLS code. Also, currently manually implemented functions, such as the FIR filter, could be replaced with build-in library functions. Finally, the application software could be upgraded from the current bare-metal, real-time OS, to an embedded Linux OS which runs on the ARM CPU. This would additionally allow the support of the RTL-SDR device to be used for live data streaming, instead of the current replay of the recorded antenna data from the SD card.

The amount of ideas for potential extensions to the project shows how comprehensive the project is and how many topics it covers. Summing up, it can be said that the project revealed a range of interesting and partially unexpected insights into multiple topics, and that a lot of knowledge was gained during the elaboration of this thesis.

Appendix A

Supplementary Materials

List of supplementary data submitted to the degree-granting institution for archival storage (in ZIP format).

A.1 PDF Files

Path: /

thesis.pdf . . . . . Master/Bachelor thesis (complete document)

A.2 Media Files

Path: /media

*.ai, *.pdf . . . . . Adobe Illustrator files
*.jpg, *.png . . . . . raster images
*.mp3 . . . . . audio files
*.mp4 . . . . . video files

A.3 Online Sources (PDF Captures)

Path: /online-sources

Reliquienschrein-Wikipedia.pdf [35]

List of Abbreviations

ADC	Analog Digital Converter
AM	Amplitude Modulation
AXI	Advanced Extensible Interface
BP	Bandpass (Filter)
CI	Continuous Integration
CPU	Central Processing Unit
DC	Direct Current
DMA	Direct Memory Access
DSB-SC	Dual-Sideband Suppressed-Carrier
DUT	Device Under Test
ERP	Effective Radiated Power
FM	Frequency Modulation
FPGA	Field-programmable Gate Array
HDL	Hardware Description Language
HLS	High-Level Synthesis
IF	Intermediate Frequency
IO	Input and Output
IoT	Internet of Things
IP	Intellectual Property
IQ, I/Q	Inphase and Quadrature components
LED	Light Emitting Diode
LP	Lowpass (Filter)
LSB	Least Significant Bit
LNA	Low Noise Amplifier
LO	Local Oscillator
OS	Operating System
PM	Phase Modulation
RTL	Register Transfer Level
RDS	Radio Data System
SDK	Software Development Kit
SDR	Software Defined Radio
SoC	System On Chip
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VHF	Very High Frequency

References

Literature

[1] European Broadcasting Union (EBU). "Low-power FM modulators in Europe". EBU-R120-2007 (2007), p. 5. URL: https://tech.ebu.ch/docs/r/r120.pdf (cit. on p. 12).

[2] International Telecommunication Union (ITU). *Radio Regulations Articles, Edition of 2020*. eng. Vol. I-EA5. 2020. ISBN: 978-92-61-30301-3 (cit. on p. 11).

[3] Silvia Maria Alessio. *Digital Signal Processing and Spectral Analysis for Scientists : Concepts and Applications*. eng. 1st ed. 2016.. Signals and Communication Technology. 2016. ISBN: 3-319-25468-5 (cit. on pp. 9, 10).

[4] A. Terry Bahill. *Tradeoff Decisions in System Design*. eng. 1st ed. 2017.. 2017. ISBN: 3-319-43712-7 (cit. on pp. 18, 19).

[5] Brian Bailey. "The Evolution Of High-Level Synthesis". *Semiconductor Engineering* (Aug. 27, 2020). URL: https://semiengineering.com/the-evolution-of-high-level-synthesis/ (visited on 07/21/2021) (cit. on p. 28).

[6] Innovation und Technologie Bundesministerium für Verkehr. "Frequenznutzungsverordnung 2013 - FNV 2013". BGBl. II Nr. 63/2014, Anlage 2 (2014), p. 99. URL: https://www.ris.bka.gv.at/Dokumente/Bundesnormen/NOR40219692/II_390_2016_Anlage_2.pdf (cit. on p. 11).

[7] European Commission. "Commission Decision on harmonisation of the radio spectrum for use by short-range devices". 2006/771/EC (Aug. 13, 2019), p. 11. URL: https://eur-lex.europa.eu/eli/dec/2006/771(2)/2019-08-13 (cit. on p. 11).

[8] Leonard Feldman. *FM Multiplexing for Stereo*. Vol. FMS-1. 62-13498 (cit. on p. 43).

[9] *FPGAs for Software Programmers*. eng. 1st ed. 2016.. 2016. ISBN: 3-319-26408-7 (cit. on p. 23).

[10] Orhan Gazi. *Understanding Digital Signal Processing*. eng. 1st ed. 2018.. Springer Topics in Signal Processing. 2018. ISBN: 981-10-4962-9 (cit. on p. 41).

[11] Liang Ge, EK Tan, and Joe Kelly. "Introduction to FM-Stereo-RDS Modulation" (2016). URL: https://www3.advantest.com/documents/11348/7898f05e-0a52-4e68-9221-3b8b75595436 (visited on 07/14/2021) (cit. on p. 44).

[12] Ralf Gessler. *Entwicklung Eingebetteter Systeme : Vergleich von Entwicklungsprozessen für FPGA- und Mikroprozessor-Systeme Entwurf auf Systemebene*. ger. 2014. ISBN: 9783834820808 (cit. on pp. 22, 23, 25).

[13] E. S Gopi. *Multi-Disciplinary Digital Signal Processing : A Functional Approach Using Matlab*. eng. 2018. ISBN: 9783319574301 (cit. on p. 10).

[14] Xilinx Inc. *Introduction to FPGA Design with Vivado High-Level Synthesis*. UG998. Version v1.1. Jan. 22, 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf (cit. on pp. 26–28).

[15] Xilinx Inc. *Vivado Design Suite User Guide - High-Level Synthesis*. UG902. Version v2018.2. July 2, 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf (cit. on pp. 26, 30, 31).

[16] European Telecommunications Standards Institute. "Transmitting equipment for the Frequency Modulated (FM) sound broadcasting service; Harmonised Standard covering the essential requirements of article 3.2 of Directive 2014/53/EU". v2.1.1.ETSI EN 302 018 (Apr. 2017) (cit. on p. 12).

[17] Kevin Krewell. "Xilinx Opens Up Vitis HLS Tool for FPGAs". *EE Times* (Mar. 10, 2021). URL: https://www.eetimes.com/xilinx-opens-up-vitis-hls-tool-for-fpgas/ (visited on 07/21/2021) (cit. on p. 28).

[18] Mohammad A Matin. *Communication Systems for Electrical Engineers*. eng. SpringerBriefs in Electrical and Computer Engineering. 2018. ISBN: 978-3-319-70129-5 (cit. on p. 11).

[19] Roberto Millón, Fernando Frati, and Enzo Rucci. "A Comparative Study between HLS and HDL on SoC for Image Processing Applications". *Elektron* 4 (Dec. 2020), pp. 100–106 (cit. on p. 28).

[20] Fábio Passos. *Automated Hierarchical Synthesis of Radio-Frequency Integrated Circuits and Systems : A Systematic and Multilevel Approach*. eng. 1st ed. 2020. 2020. ISBN: 978-3-030-47247-4 (cit. on pp. 5–7).

[21] Franz Schnyder and Christoph Haller. "Implementation of FM Demodulator Algorithms on a High Performance Digital Signal Processor". diploma thesis. HSR Hochschule für Technik Rapperswil, Elektrotechnik. URL: https://www.veron.nl/wp-content/uploads/2014/01/FmDemodulator.pdf (visited on 07/06/2021) (cit. on pp. 14, 16).

[22] Rohde & Schwarz. "Messungen an FM-Sendern für Abnahme, Inbetriebnahme oder Wartung". 7BM105_0D (Sep-13). URL: https://cdn.rohde-schwarz.com/pws/dl_downloads/dl_application/application_notes/7bm105/7BM105_0D.pdf (visited on 07/05/2021) (cit. on p. 13).

[23] K.S Thyagarajan. *Introduction to Digital Signal Processing Using MATLAB with Application to Digital Communications*. eng. 2019, pp. 39–44. ISBN: 9783319760292 (cit. on pp. 9, 10).

Media

[24] *Basisbandsignal eines UKW-Rundfunkprogrammes*. URL: https://de.wikipedia.org/wiki/UKW-Rundfunk#Technische_Details (visited on 07/05/2021) (cit. on p. 12).

[25] *Frequency Discriminator Block Diagram*. URL: https://www.tutorialspoint.com/analog_communication/analog_communication_fm_demodulators.htm (visited on 12/16/2020) (cit. on p. 14).

[26] Xilinx Inc. *Xilinx ZedBoard Product*. URL: https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html (visited on 07/20/2021) (cit. on p. 60).

Online sources

[27] cocotb contributors. *Welcome to cocotb's documentation!* Version ec99a877. URL: https://docs.cocotb.org/en/v1.5.2/ (visited on 07/15/2021) (cit. on pp. 50, 51).

[28] Intel Corporation. *Intel oneAPI Overview*. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi.html (visited on 07/21/2021) (cit. on p. 28).

[29] Tristan Gingold and contributors. *About GHDL*. Version 18104277. URL: https://ghdl.readthedocs.io/en/latest/about.html#what-is-ghdl (visited on 07/15/2021) (cit. on p. 50).

[30] The MathWorks Inc. *MATLAB Overview*. URL: https://www.mathworks.com/products/matlab.html (visited on 07/09/2021) (cit. on p. 38).

[31] Xilinx Inc. *SystemC Design Entry for HLS is deprecated in 2020.2*. URL: https://www.xilinx.com/support/answers/73613.html (visited on 07/20/2021) (cit. on pp. 23, 28).

[32] Rick Lyons. *Digital Envelope Detection: The Good, the Bad, and the Ugly*. Apr. 3, 2016. URL: https://www.dsprelated.com/showarticle/938.php (visited on 07/06/2021) (cit. on p. 15).

[33] Electronics Notes. *Broadcast VHF FM Tutorial & Basics*. URL: https://www.electronics-notes.com/articles/audio-video/broadcast-audio/vhf-fm-frequency-modulation-basics.php (visited on 07/05/2021) (cit. on p. 12).

[34] GNU Radio project. *About GNU Radio*. URL: https://www.gnuradio.org/about/ (visited on 07/09/2021) (cit. on pp. 37, 47).

[35] *Reliquienschrein*. Sept. 2018. URL: https://de.wikipedia.org/wiki/Reliquienschrein (visited on 02/28/2019).

[36] Ettus Research. *Product USRP B200mini-i*. URL: https://www.ettus.com/all-products/usrp-b200mini-i-2/ (visited on 07/14/2021) (cit. on p. 59).

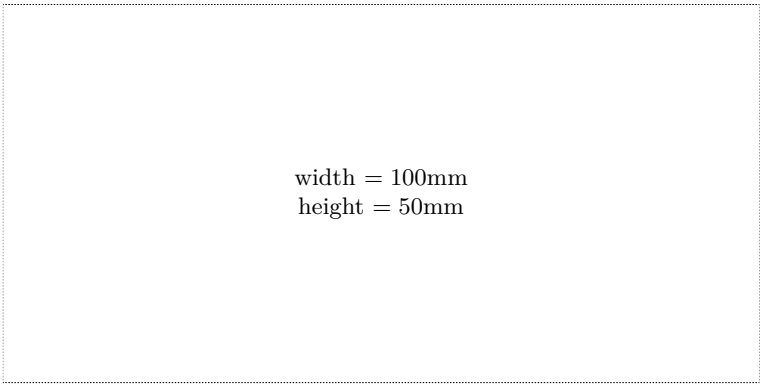
[37] Carsten Roppel. *Analoge Modulationsverfahren und Rundfunktechnik; Begleitmaterial zum Buch 'Grundlagen der digitalen Kommunikationstechnik, Übertragungstechnik - Signalverarbeitung - Netze'*. Aug. 24, 2010. URL: https://www.hs-schmalkalden.de/fileadmin/portal/Dokumente/Fakult%C3%A4t_ET/Personal/Roppel/Buch/Analoge_Modulationsverfahren.pdf (visited on 07/06/2021) (cit. on pp. 16, 17).

[38] RTL-SDR. *About RTL-SDR*. URL: https://www.rtl-sdr.com/about-rtl-sdr/ (visited on 07/14/2021) (cit. on p. 59).

[39] Wikipedia. *Autofahrer-Runkfunk-Information: Hinz Triller*. URL: https://en.wikipedia.org/wiki/Autofahrer-Rundfunk-Informationssystem,%20https://de.wikipedia.org/wiki/Autofahrer-Rundfunk-Information (visited on 07/09/2021) (cit. on p. 44).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —