Student Guide for EE465
Design of a DSP System

Original version created by Eric Salt and Ha Nguyen
for the inaugural offering of EE465 in January 2015

Document Revision:
07/01/2021

Revision History

```
Revision History:
Revised spring/summer 2015: Major revisions based on feedback
                            from the class of 2015.
Revised December 2016:      Major revisions based on feedback
                            from the class of 2016.
Revised Jan 3, 2017:  Corrected typos
Revised Jan 9, 2017:  Added the matlab code used in the
                      discussion for deliverable 1.
                      Also changed the length of RC and SRRC
                      filters used in deliverable 1.
Revised Jan 10, 2017: Revised method for steering discussions
                      for deliverable 1.
                      The original method of discussing a list
                      of fairly pointed questions that were provided
                      in advance did not seem to be working that well.
                      That has been changed to listing a few broader
                      topics for discussion.
Revised Jan 13, 2017: Added detail to the specification for
                      deliverable 1 so that it can be designed,
                      built and tested.
Revised Jan 19, 2017: Changed stop band attenuation for deliverable 1
                      from 62 to 40 dB.
Revised Jan 26, 2017: Learned from the class of 2017 (yet to graduate)
                      that the suggestions from the class of 2016 were
                      not implemented properly. Therefore, deliverable 1
                      was substantially revised.
Revised Feb 5, 2017:  Modified Deliverable 1 in preparation for 2017-2018
                      Modified Deliverable 2 to reflect the changes
                      explained in class.
Revised Feb 15, 2017: Made substantial revisions to Deliverable 2 to clearify
                      issues raised by the first students to get it working.
                      This interrupted the revisions for Deliverable 3
                      that were in progress. Those revisions will be finished
                      later.
Revised Feb 21, 2017: Finished adding the notes for lecture on making baseband
                      measurements for Deliverable 3 based on the RF
                      specifications given for Deliverable 4.
Revised Mar 20, 2017: (By Salt/Gowen) Revised the wording in Deliverable 4 and
                      added the pre-exam information form for Deliverable 4.
Revised Mar 22, 2017: (By Rory Gowen) Added Table 4 to pre-exam information form
                      for Deliverable 4.
Revised June 25, 2017: Major revisions by Eric Salt(20 or 30 additional pages)
                       Reorganized the document into three chapters.
```

```
                      The body of chapter 3 was moved to a separate .tex file.
                      Rory Gowen expanded the .tex file for chapter 3.
                      Extensive notes on timing recovery were also added.
Revised June 28, 2017: Modified the high level block diagram, which is the first
                      figure referenced in the section on project over view.
                      Rory Gowen added the Lab exam for deliverable 5 (chap 3)
Revised August 17, 2017: Changed a "-" to a "+" in a verbatim Matlab script
                        Also corrected RC impulse response equation
            (due to Brian Daku)
Revised Jan 5, 2018: (By Brian Berscheid)
                     Initial version of package for 2018 class
                     Fixing typos, adding outline of notes for some sections
                     Revising lab exam descriptions for 2018 session of class
Revised Jan 26, 2018: (By Brian Berscheid)
                      Fixing errors and adding detail in the verilog review notes
                      Adding review notes on Modelsim and functional simulation
                      In process of adding notes on bit growth in FIR filters
                      Updates and clarifications to Lab Exam 1
Revised Feb 1, 2018: (By Brian Berscheid)
                     Updating description and diagram of Lab Exam 2 SUT
                     Added 'hours worked' question to each Lab Exam
                     Finished notes on bit growth in FIR filters
                     In process of adding notes on linear phase filters
Revised Feb 16, 2018: (By Brian Berscheid)
                     Added notes on FIR filter structures
                     Added notes on pipelining in filters
                     Corrected error in Deliverable 2 block diagram
Revised Mar. 11, 2018: (By Rory Gowen)
                     Added a disable for the adjacent channel in Deliverable
                     4 block diagram.
                     Corrected notes on BER testing to reference
                     $2^(20) symbols instead of 2^(22) symbols.
                     Added steps to test BER and MER with and without
                     adjacent channels.
Revised April 12, 2018: (By Brian Berscheid)
                        Changed length of Kickstart and D1 filter to 21
                        Added 55dB measurement range requirement for D2 circuit
Revised Feb 28, 2019: (By Brian Berscheid)
                      Added notes regarding OOB emissions requirements (D#3)
                      Added material on timing closure and the use of TimeQuest
Revised April 4, 2019: (By Brian Berscheid)
                       Adding lecture notes on phase and frequency recovery
Revised Jan 6, 2020:  (By Brian Berscheid)
                      Fixed error in D1 matlab discovered by Brian Daku
                      Initial version of document for 2020 offering
```

```
Revised Jan 7, 2021: (By Brian Berscheid)
                      Fixed a few minor typos
                      Initial version of document for 2021 offering
```

Contents

Acknowledgements

The creators of this document must thank the classes of 2014-2015 and 2015-2016 for their very considerate and well thought out suggestions for improving the class. The creators hold both classes, especially the class of 2014-2015, in high regard for being very understanding and maintaining a positive attitude while fighting through a class with organizational growing pains.

The creators would also like to thank Vecima Networks and in particular Dr. Brian Berscheid for their behind-the-scenes help.

Perspective

From an engineering project point of view, the objective is to design, build and test a 16-QAM modulator and 16-QAM demodulator for a CATV system in piece-by-piece fashion over 12 weeks. The course has a 3L-3P classification, which has the following student workload implications:

1. 3L classes have 3 hours of lectures per week and students are expected to attend all lectures.

2. 3L classes have assignments that are expected to consume 3 hours per week of student effort.

3. Normally classes with a 3P designation have a 3 hour per week practicum component that involves students making measurements in the labs for 3 hours per week and writing the report outside the lab, which takes another 1.5 hours per week.

4. The student workload for a normal 3L-3P class consists of 3 hrs lectures/week, 3 hrs assignments/week, 3 hrs labs/week and 1.5 hrs report writing/week. The total workload is 3 hrs lectures/week plus 7.5 hrs/week outside the lectures.

EE465 is not a regular 3L-3P class so does not have the standard 4 component, but the 3L-3P designation carries the implication of 7.5 hours/week of work outside of the lectures. Over a 12-week period this amounts to 90 hours of work outside the lectures.

This class, being a guided design, does not fit the standard class template where typically 12 assignments and 12 labs are issued at weekly intervals. Furthermore, the assignments and labs are bonded with the outcome being a few well-specified deliverables. The students are expected to spend 7.5 hours per week designing, building and testing circuits. The proportion of time spent in the lab to time spent outside the lab will vary from week to week.

The class has six milestones, each of which having predefined deliverables in the form of a working circuit. The time-line will vary from year to year, but the students will have to demonstrate that the circuit associated with each milestone works properly.

Project Overview

The general block diagram in Figure 1 provides an overview of the CATV communication system that will be designed and built. Its operation will be described in the lectures.

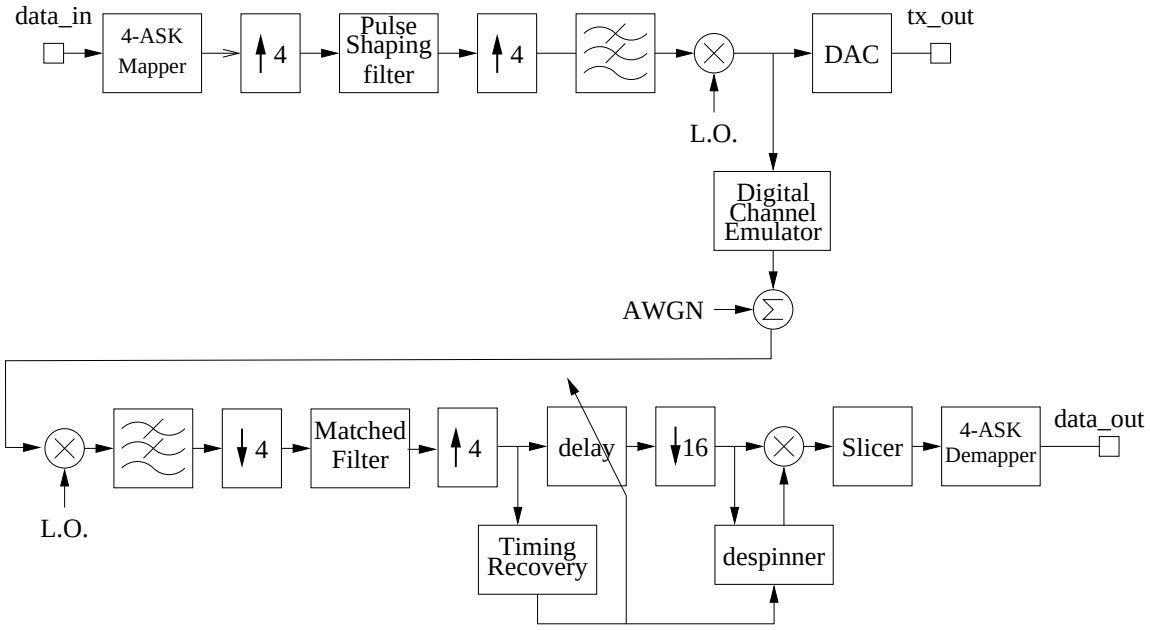

Figure 1: A high level block diagram for one channel (in-phase or quadrature) of a 16-QAM CATV communication system.

Chapter 1

Lecture Notes

1 History

The inaugural EE465 class was given in term 2 of the 2014-2015 school year. That class was structured as a guided design class and did not have conventional lectures and lecture notes. Every student in the class was tasked with designing a CATV modem to meet a specification issued by the instructor. At that time it was believed the students would have learned all that was necessary to design the CATV modem in the prerequisite classes, which were EE265, EE362, CME341, EE365, EE456 and EE461.

The design was broken down into three deliverables, which were referred to as Deliverable 1, Deliverable 2 and Deliverable 3. These deliverables were described in a 19 page document in portable document format (pdf). The deliverables were examined and graded in three "show-and-tell" type lab exams, where the students demonstrated to the class professor and DSP stream engineer that their deliverables met the specification.

Feedback from the students in the inaugural class suggested that Deliverable 1 encompassed too much and suggested it be broken down into two deliverables. This was done prior to the 2015-2016 year and EE465 had 4 deliverables for 2015-2016.

Feedback from the students in the 2015-2016 class indicated that deliverable 1, which was the first part of the original Deliverable 1, was too difficult and took them too long to get control of what to do. They suggested Deliverable 1 be further divided into two deliverables.

The class of 2015-2016 also strongly suggested that the workload of EE465 be moved forward so that it didn't peak at the same time as that of the capstone design class.

The student feedback was taken seriously and EE465 was restructured/reorganized for the 2016-2017 year. The first deliverable was made easier breaking the original Deliverable 1 into three sequential deliverables. The original Deliverable 1 was now Deliverable 3. The workload was moved forward by using the first 3 hour scheduled lab period to deliver lectures and cancelling the lectures during the week of the capstone design presentations. As stated earlier, the reason for doing this was to reduce the peak in the workload.

Very loud feedback from the students in the 2016-2017 class made it clear that, while the material covered in the prerequisite classes was sufficient to do the design, much of it was forgotten by the time they reached term 2 of their 4^{th} year. At different times throughout the year they found themselves in difficulty and asked that certain material be reviewed

in lectures. Of course these requests were granted, but the reviews would have been more beneficial had they came earlier. The class suggested that in the years to come the review material be presented in a timely manner in the EE465 lectures. They felt the request-then-present approach had the review material coming too late. They also suggested adding a lab at the beginning of the class with a quick due date to jump start the students and get them into the rhythm of the class.

In response to the feedback from the students in the 2016-2017 class three changes were made in preparation for the 2017-2018 class.

1. A new "kick start" lab assignment was added with the examination date being in the second (or third) week of the term.

2. Some review material was added to the student guide .pdf document. This material will be revised and added to in the future as the instructors get a better sense of exactly what review material is required.

3. The structure of the document was reorganized into three chapters: one for the review notes, one for the design specifications and one for the lab examinations.

In general, the "lecture time" in this class is used for two activities:

1. lectures (mostly review, but some new material) and

2. interactive discussions, where the instructor poses questions and the class sorts out the answers through discussion and questions of their own.

As the term progresses a larger portion of the "lecture time" will be used for discussion.

The lecture material and its ordering may vary from year to year based on student feedback and requests. The sections below are intended to provide a starting point and general guidelines for the lectures and discussions.

2 Lecture / Discussion Topics for Kick-Start Lab

2.1 Plan for Kick-Start Lab

The first three-hour lab slot that occurs after the first lecture will used for 3 hours of lectures. Depending on what day of the week the term begins, the first lab slot could happen before the first lecture, in which case the first lab will not be used and the second scheduled lab of the term will be used for three hours of lectures. The room used for the lecture that replaces the lab slot will be announced in class.

The topics to be covered in the lectures corresponding to the kick-start lab are:

2.2 Frequency response of FIR filters

One of the key properties of an FIR filter is the frequency response. This topic has been discussed at length in EE 362 and EE 461, but a brief summary is provided here as a review. For more details, please refer to the notes from those prior classes.

By definition, the frequency response of a filter with impulse response $h[n]$ is:

$$H\left(e^{j\omega}\right) = \sum_{n=0}^{N-1} h[n]e^{-j\omega n} \tag{1.1}$$

where N is the length of the filter impulse response. Equation 1.1 indicates that the frequency response of a filter may be calculated by taking the Fourier transform of the impulse response coefficients.

In general, equation 1.1 produces a complex number which depends on the digital frequency parameter ω and the impulse response of the filter. This complex number is generally expressed in polar form as:

$$H\left(e^{j\omega}\right) = \left|H\left(e^{j\omega}\right)\right| e^{j\angle H\left(e^{j\omega}\right)}$$

where $\left|H\left(e^{j\omega}\right)\right|$ and $\angle H\left(e^{j\omega}\right)$ are known as the magnitude response and phase response of the filter, respectively. The physical interpretation of these quantities is as follows: if the input to a filter is a sinusoid at a particular frequency ω_{in}, the output of the filter will be a sinusoid at the same frequency, but with magnitude and phase modified according to the magnitude and phase response of the filter evaluated at ω_{in}.

$$x[n] = e^{j\omega_{in}n} \rightarrow y[n] = \left|H\left(e^{j\omega_{in}}\right)\right| e^{j(\omega_{in}n+\angle H\left(e^{j\omega}\right))} \tag{1.2}$$

The frequency response of a filter is important because Fourier analysis allows us to view any arbitrary input signal as the sum of a set of sinusoids. Since FIR filters are linear systems, the we can compute the output of the filter in response to each of these sinusoids, then sum the results in order to compute the filter output for complex input signals such as those used in digital communication systems.

One of the main topics covered in the kickstart lab/assignment is the scaling of a scaling of a sinusoidal input through an FIR filter.

Some questions on this topic that could be discussed in class:

1. How does the frequency response of a filter change if all of its coefficients are scaled by a constant factor K? i.e., $h_{new}[n] = Kh_{old}[n]]$

2. If a sinusoid with a peak value of 1 is passed through a filter, what is the peak value of the output sinusoid?

3. Given a set of filter coefficients, how can we find the frequency response in Matlab? Is there a built-in function which can provide this information?

4. Given a set of filter coefficients, how can we perform a Matlab simulation to verify the result in that was given by the built-in function above?

5. Why is looking at the peak value of a real output sinusoid in response to a real input sinusoid not a very accurate method of finding the frequency response? What alternatives are there?

6. How might we test the frequency response of an FIR filter that is running in an FPGA?

2.3 Designing a Practical FIR filter

Linear Phase FIR Filters

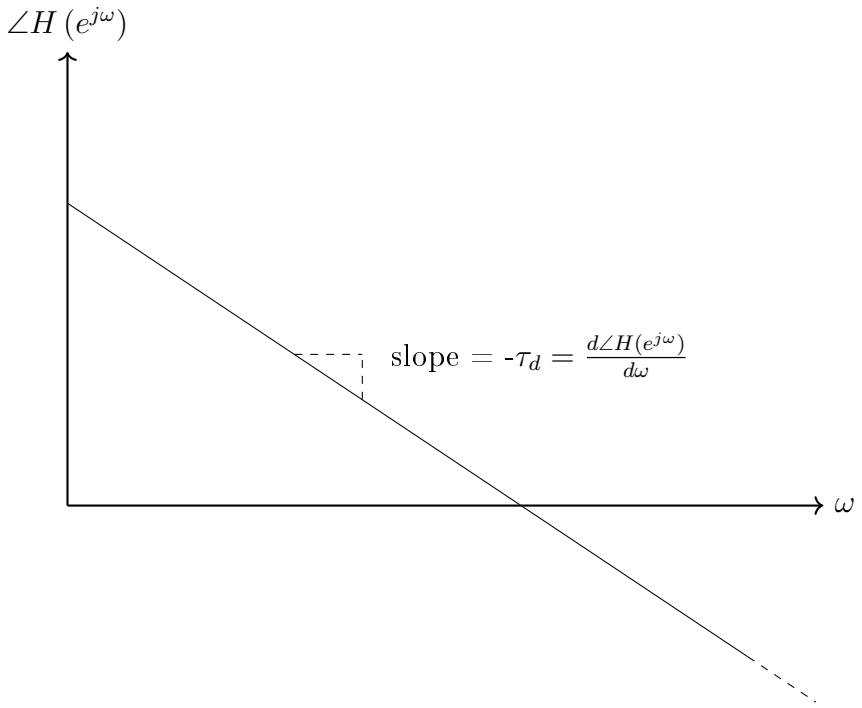

Figure 1.1: Phase response of a linear phase filter.

The filters used in digital communication systems are typically linear phase filters. The topic of linear phase filters has been covered in EE 362 (and likely EE 365 and 461), but a very brief review is provided here. For more details, please refer to your EE 362 notes or a standard DSP textbook such as *Discrete Time Signal Processing* by Oppenheim and Schafer.

The term "linear phase" in the context of a digital filter means that the phase response of the filter is linear (with respect to frequency). Linear phase is an important property because a linear phase response in the passband of a filter is equivalent to a pure delay. That is, to signals falling within the passband of a linear phase filter, the output of the filter is a delayed version of the filter input. All sinusoids are delayed by an equal amount by a

linear phase filter, which means that the shape (time domain waveform) of a signal which falls within the passband of the filter is not distorted.

Conversely, a filter which is not linear phase delays some sinusoids more than others. This has the effect of changing the instants in time in which the sinusoids add constructively and destructively. Thus, a filter which is not linear phase typically changes the time domain shape of its input waveform. In a digital communication system, the shapes of the transmitted waveforms are used to convey information, so such distortion to the waveform shape potentially hinders the communication process, making such distortion highly undesirable.

The delay experienced by a signal passing through a linear phase filter is:

$$\tau_d = -\frac{d\angle H(e^{j\omega})}{d\omega} \tag{1.3}$$

The quantity τ_d is commonly referred to as the "group delay" or "transport delay" of the filter. As shown in Figure 1.1, since the phase response $H(e^{j\omega})$ is linear with respect to frequency, equation 1.3 yields a group delay which is constant with respect to frequency for a linear phase filter.

For an FIR filter to be linear phase, its impulse response must be either symmetric or antisymmetric around the center point in the impulse response. The delay τ_d through such a filter is $(N-1)/2$, where N is the length of the impulse response. In such a filter, τ_d can be interpreted as the delay from the input to the middle of the corresponding output pulse.

For example, consider the illustration shown in Figure 1.2, which shows the group delay of an $N = 3$ linear phase filter with system function $H(z) = 0.5 + z^{-1} + 0.5z^{-2}$. It is clear from the figure that the output of the system in response to an impulse is a pulse 3 samples in duration. The midpoint of the pulse is at sample $(N-1)/2 = 1$, which represents a 1 sample delay relative to the input impulse, which was injected at time $n = 0$.

Similarly, Figure 1.3 illustrates the group delay of a linear phase filter with an even length impulse response. The specific filter under consideration has $N = 4$ with system function $H(z) = 0.4 + 0.8z^{-1} + 0.8z^{-2} + 0.4z^{-3}$. For such a system, the output in response to an impulse at time $n = 0$ is a 4 sample pulse, spanning $n = 0$ to $n = 3$. The midpoint of pulse is at sample $(4-1)/2 = 1.5$, indicating that this filter applies a 1.5 sample delay to its input signal.

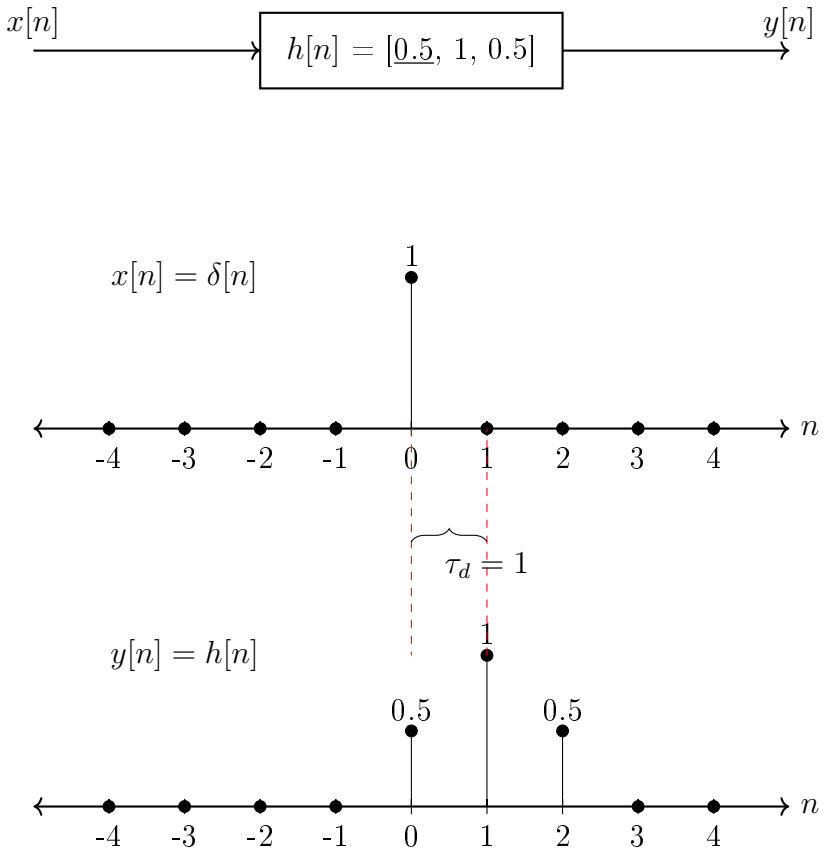

Figure 1.2: Impulse response of an odd length linear phase filter.

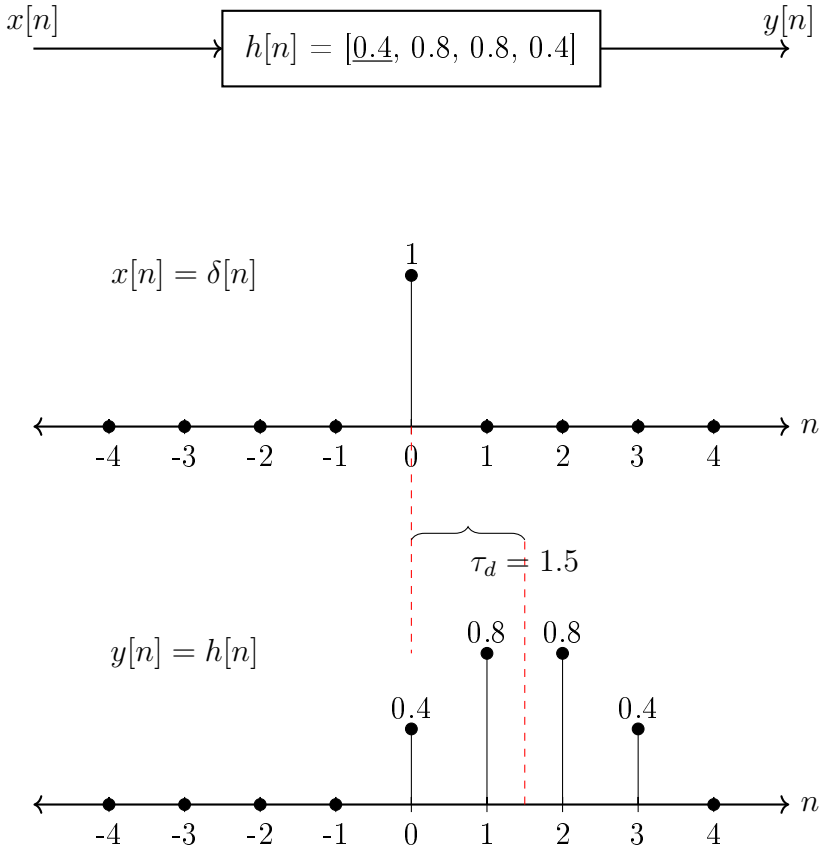

Figure 1.3: Impulse response of an even length linear phase filter.

Structures for FIR Filters

This section highlights a few basic structures for FIR filters. To help illustrate the concepts, a concrete example is considered, consisting of a single filter with system function $H(z) = 0.4 + 0.8z^{-1} + 0.8z^{-2} + 0.4z^{-3}$. As noted above, this system function corresponds to a linear phase filter of length $N = 4$.

It is obvious that any structure used to implement an FIR filter must compute the result represented by the convolution equation:

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{1.4}$$

For the specific filter under consideration, the general convolution equation of 1.4 simplifies to:

$$\begin{aligned} y[n] &= h[0]x[n] + h[1]x[n-1] + h[2]x[n-2] + h[3]x[n-3] \\ y[n] &= 0.4x[n] + 0.8x[n-1] + 0.8x[n-2] + 0.4x[n-3] \end{aligned}$$

The most straightforward structure used to represent an FIR filter directly implements this equation, as shown in Figure 1.4. This structure is known as the "direct form".

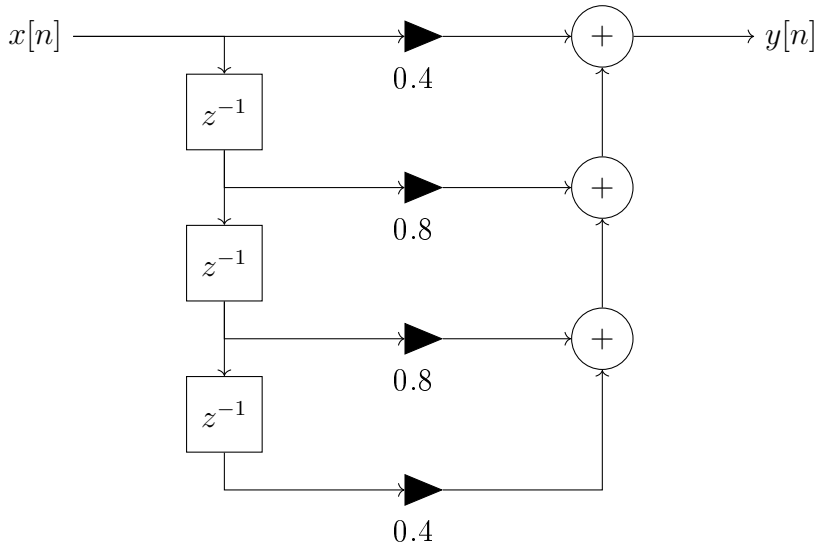

Figure 1.4: Direct form structure for an FIR filter.

The direct form structure consists of three main parts: a set of registers which delays the input signal, a set of multipliers which compute the product the delayed input signal values and the filter coefficients, and a set of adders which sum the multiplier outputs to generate the final output $y[n]$.

The symmetry of the impulse response of linear phase filters can be exploited when they are implemented in hardware. Since multipliers are an important resource in FPGAs, designers strive to optimize their use when possible. In the case of a linear phase filter, each coefficient unique coefficient value appears in the convolution equation twice. Consequently, the convolution equation can be factored into the following form:

$$y[n] = 0.4(x[n] + x[n-3]) + 0.8(x[n-1] + x[n-2]) \tag{1.5}$$

The physical interpretation of this equation is that we can add pairs of input samples prior to performing the multiplications, which reduces the number of multipliers required to implement the circuit. The corresponding hardware structure is shown in Figure 1.5.

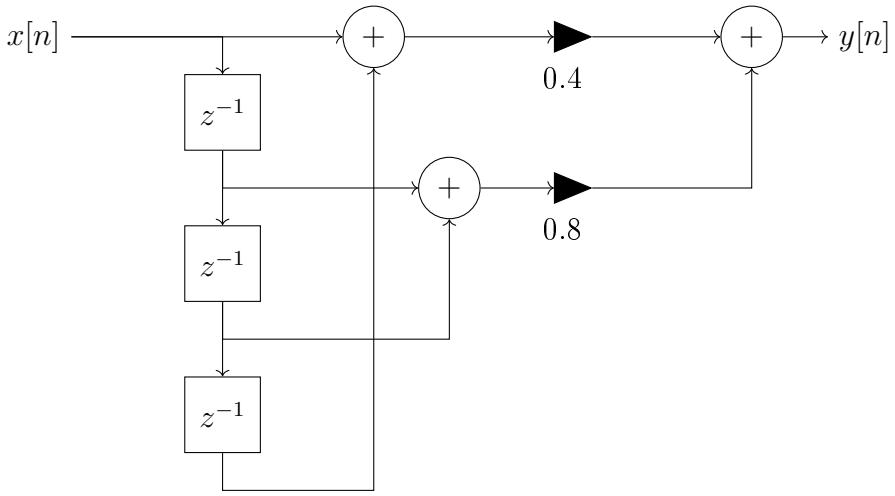

Figure 1.5: Symmetry exploitation structure for linear phase FIR filters.

Another FIR filter structure which is sometimes used in industry is the so-called "systolic" filter structure, which moves the filter delay elements after the multipliers, as shown in Figure 1.6. The reasoning for doing this is as follows.

Many modern FPGAs contain circuit elements that are highly optimized for DSP operations. In addition to embedded multipliers, these DSP blocks commonly contain adders and registers which can act upon the output of the multipliers. By rearranging the filter in the manner shown in Figure 1.6, it is possible to implement the delay chain and the adders inside the DSP blocks (as well as the multipliers). This ideally allows the filter to be implemented exclusively inside the DSP blocks.

Implementing the delay chain and adders of a filter inside the DSP blocks is desirable for two reasons. First, the DSP blocks are optimized for higher clock speeds and lower power consumption than the general reconfigurable logic blocks (LUTs and flip-flops). Second, by performing these operations inside the DSP blocks, the general purpose reconfigurable logic that would have otherwise been consumed by these functions is left available for other uses.

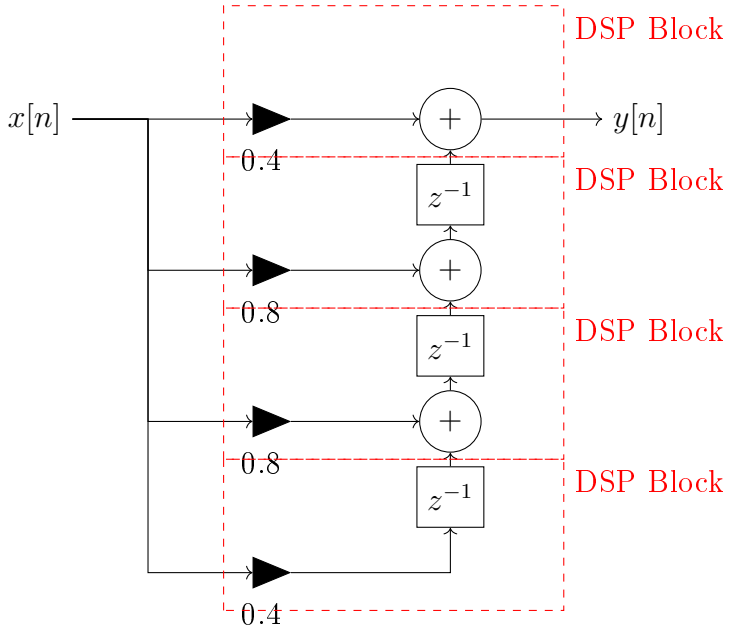

Figure 1.6: Systolic FIR filter structure.

The systolic filter structure is covered here for informational purposes only and is not recommended for use in your EE 465 design. The Cyclone IV FPGA used in EE 465 does not contain DSP blocks with the built-in register and post-adder necessary for the systolic structure to be advantageous. In the Cyclone IV, the registers, post-adders, and pre-adders will all be built using general purpose FPGA logic resources, regardless of which filter structure is used.

Questions to be discussed by the class:

- What is the cost in terms of registers, adders, and multipliers of a direct form FIR filter?
- What is the cost in terms of registers, adders, and multipliers of an FIR filter using the symmetry exploitation form?
- What is the cost in terms of registers, adders, and multipliers of an FIR filter using the systolic structure?
- Can you think of any disadvantages of the symmetry exploitation structure?
- Can you think of any disadvantages of the systolic structure?

Pipelining in FIR Filters

It is sometimes desirable to add extra registers in FIR filter modules. These registers, known as pipeline registers, are used to help the FPGA compiler (Quartus) ensure that the setup

requirements for all of the flip flops in the design can be met.

In general, the term "pipelining" refers to the process of breaking down a large operation into multiple smaller operations that can each run in a single clock cycle. The general concept is illustrated in Figure 1.7.

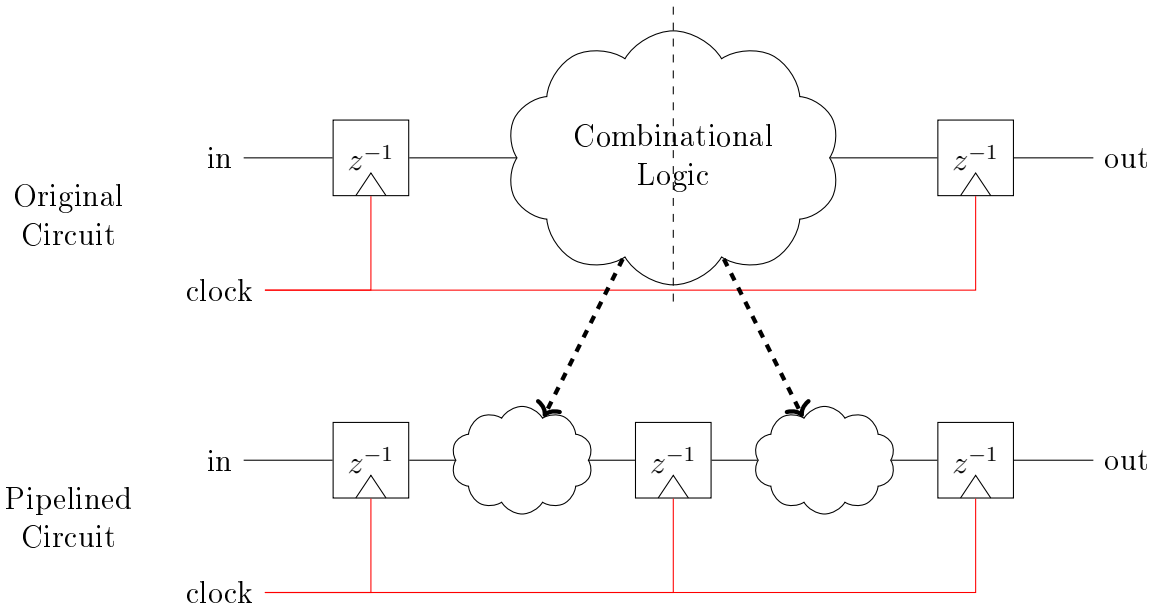

Figure 1.7: Concept of pipelining of digital logic.

In order for the digital logic to function correctly, the data which is clocked out of the source register must propagate through the combinational logic and reach the destination register t_{setup} prior to the next clock pulse. If the combinational logic is relatively complex or the clock frequency is relatively high, this setup requirement can be difficult to achieve. In such cases, pipelining can be necessary in order to achieve a functional circuit.

Consider a standard direct form FIR implementation, shown in Figure 1.8. For illustration purposes, a hypothetical destination register has been added at the output of the filter.

It can be observed from Figure 1.8 that there is a combinational path from $x[n-3]$ to the output register which includes 1 multiply operation and 3 additions. The longest path is commonly referred to as the "critical timing path" or "critical path", as it is the most challenging path for Quartus to place and route in order to meet the setup requirement at the output register. If the digital word lengths are long and/or the clock frequency is high, the propagation delay through the critical path may approach the clock period, potentially causing setup failures.

In practice, the critical path length often determines the maximum clock frequency that can be used for a design (and therefore the data rates that can be achieved for the design). For this reason, FPGA developers in industry spend a significant amount of time optimizing

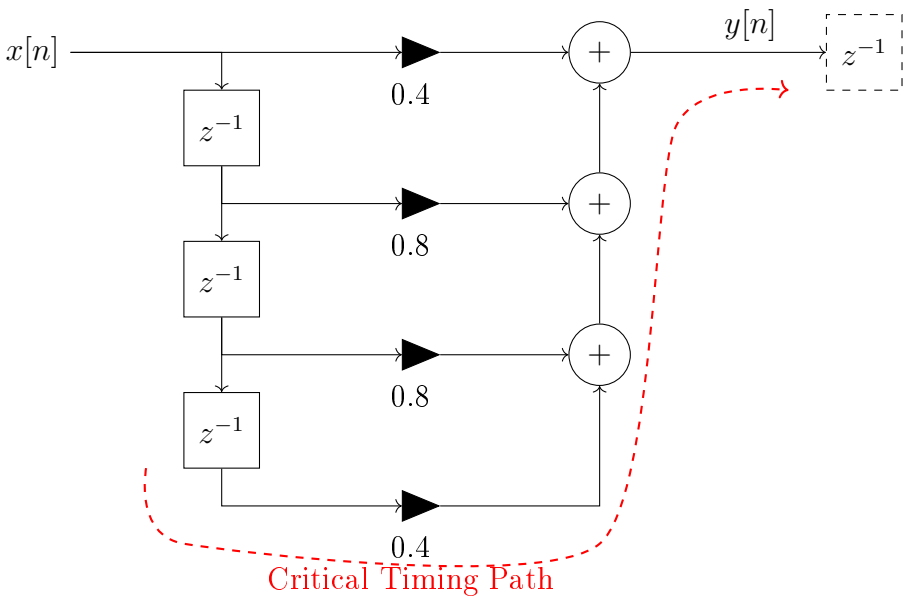

Figure 1.8: Critical timing path of a direct form FIR filter.

their designs in order to limit the length of the critical path. Pipelining is one common method of achieving this goal.

When adding pipelining registers, it is important to be sure the functionality of the circuit is not changed overall. Of course, the output will be delayed by a number of samples equal to the number of pipeline registers added, but the calculations perfomed by the circuit must not change. For exampl, consider the FIR filter implementation shown in Figure 1.9, in which a single register (shown in red) has been added within the filter.

Does the pipeline register shown in red in Figure 1.9 reduce the critical timing path length?

Does the pipeline register shown in red in Figure 1.9 change the functionality of the circuit? Hint: compute the difference equation for the block diagram to find out...

The obvious question at this point is: how can we safely add pipelining to FIR filters without changing their functionality? There are two main approaches to achieving this goal. The first is more intuitive, but may involve trial and error, while the second is more formal.

Manual Method

One method of adding pipeline registers is to start by freely adding registers to the input and output of the filter. As many registers as desired can be added at this stage. The registers can then be manually distributed throughout the filter by shifting them around in the block diagram. The rules which must be followed during this register repositioning process are as follows:

- A register at the input to a splitting node (a point at which a signal splits into multiple directions) is equivalent to registers on each of the outputs of the splitting node.

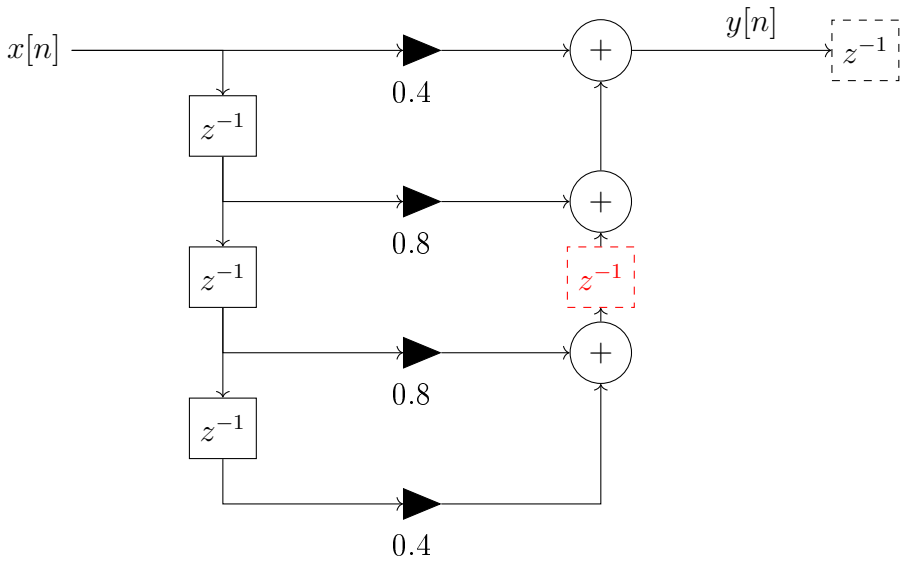

Figure 1.9: An attempt at adding pipelining to a direct form FIR filter. Is the filter functionality changed by the pipeline register?

- A register at the output of a combining node (an adder) is equivalent to registers on each of the inputs of the combining node.
- Registers can be freely moved on paths that do not contain splitting nodes or combining nodes.
- Registers can be freely moved through constant coefficient multipliers.

These rules are illustrated graphically in Figure 1.10. By moving delays around within the filter according to these rules, it is possible to reduce the length of the critical path in the system.

Cutset Method

The first step in the "cutset method" is to draw a line through the block diagram representation of a digital system. The line must meet the following conditions:

- The line must cut the block diagram into two distinct pieces. It is not necessary for the two pieces to be equal (or even close to equal) in size.
- The direction of signal flow must be the same every time a wire crosses the line.

Once an appropriate line has been selected, the designer should add a pipeline register at every place the line cut through the diagram. This process can be repeated as many times as desired.

For example, consider the three potential cutset lines shown in Figure 1.11.

Questions to be discussed in class:

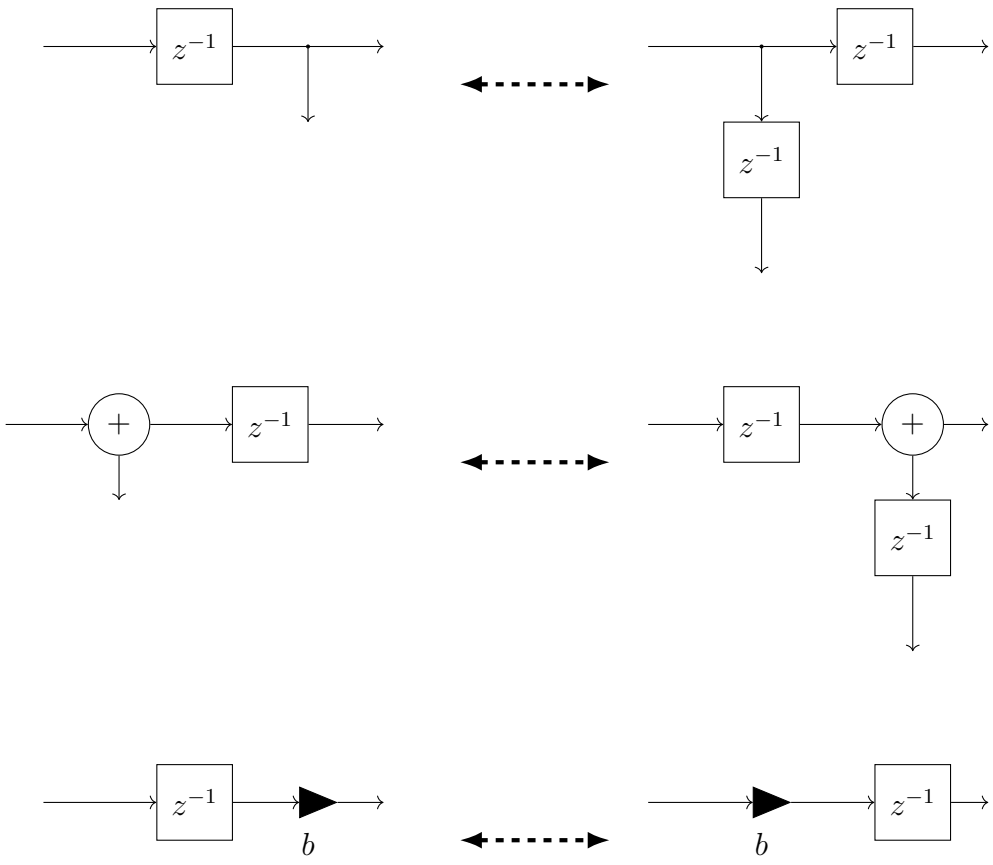

Figure 1.10: Graphical illustration of rules for manual pipelining.

- Which of the three lines shown in Figure 1.11 represents a valid cutset selection?
- For each of the valid cutsets, draw the filter structure after the addition of pipelining registers. Compute the difference equation for the resulting structures and verify that the filtering operation is unchanged (aside from the addition of a delay to the output signal).
- Which of the valid cutsets is more useful in reducing the critical timing path?

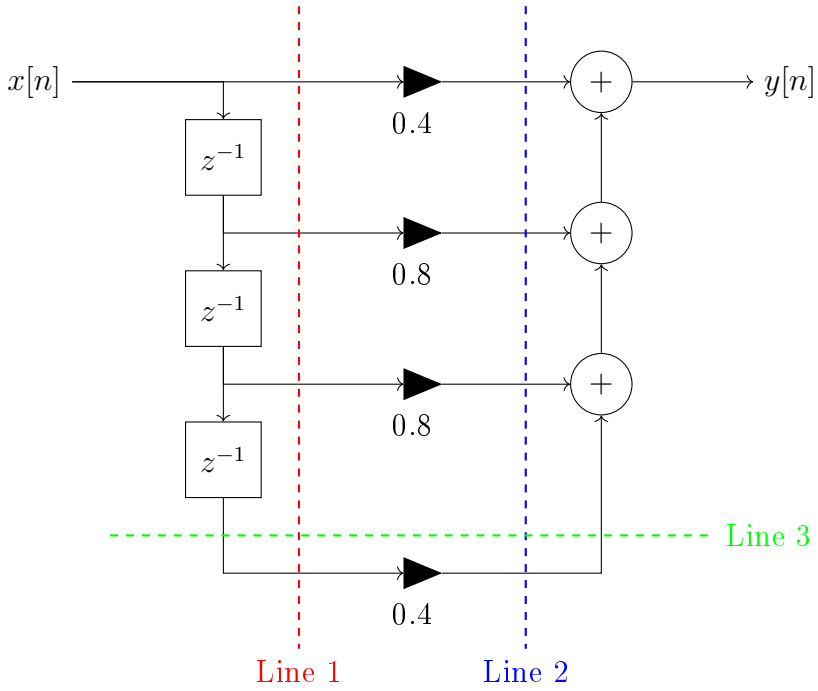

Figure 1.11: Three proposed applications of the cutset method of adding pipelining to an FIR filter.

Bit Growth in FIR Filters

Motivation

When designing and implementing FIR filters in Verilog, it is very important to carefully choose the bit widths used to represent each signal (both the data signals and the coefficients) in the filter. Recall from EE 365 that digital signals are commonly described based on the number of bits used to represent the integer and fractional portions of the signal. For example:

- A 2s2 number has a total of 4 bits, with 2 integer bits and 2 fractional bits. (Signed number)
- A 1s17 number has a total of 18 bits, with 1 integer bit and 17 fractional bits. (Signed number)
- A 0u10 number has a total of 10 bits, all fractional. (Unsigned number)

Question for class: What range of values can be represented by each of these number formats? How can we determine this?

There are two main questions that must be answered for each signal in the design:

1. How many integer bits should be used?

2. How many fractional bits should be used?

Many FPGAs have built-in multipliers which have a fixed number of bits. Since multiplication is expensive to construct using general-purpose FPGA logic, it is often convenient and economical to select number representations which match the width of the provided multipliers (typically 18 bits, but some Altera devices also have a 9-bit input mode in which a single 18x18 multiplier can be used to perform two 9x9 multiplies simultaneously). In this case, although the total bit width is fixed, the designer has the flexibility to decide where to place the binary point.

The positioning of a binary point in a fixed width signal involves a tradeoff. Moving the binary point to the left (fewer integer bits) reduces the quantization noise present in the representation, but also reduces the size of the largest number that can be represented in the digital word. Conversely, moving the binary point to the right (more integer bits) increases the quantization noise, but also increases the size of the largest value that can be represented in the digital word.

In order to make an appropriate selection for the binary point location, it is important to understand the growth of the input signal throughout the filter so that an appropriate number of integer bits can be selected. If the number of integer bits is too small, overflows may result, causing severe degradation to the signal. However, if the number of integer bits is too large, the quantization noise may be larger than is necessary.

The bit growth throughout the filter depends greatly on the characteristics of the input signal. The following sections consider two common cases: sinusoidal inputs and random inputs.

Bit Growth with Sinusoidal Inputs

The case of a sinusoidal input to an FIR filter was discussed in detail in EE 365. The following discussion is intended as a brief review of that material. For more details, please refer to your EE 365 notes.

When a sinusoidal input is passed into an FIR filter, the definition of frequency response states that the filter output willbe a sinusoid of the same frequency, but with an amplitude scaling and phase shift that are determined by the frequency response of the filter. Therefore, if the peak value of the input sinusoid is known, the peak value of the output can be calculated based on the value of the magnitude response at the input frequency. This information can then be used to size the output of the filter.

For example, consider the magnitude response of a hypothetical digital filter shown in Figure 1.12. If the input signal $x[n]$ to the filter is a sinusoid which is scaled such that its peak value is 0.5 as shown in Figure 1.13, what is the largest possible output value $y[n]$?

According to the definition of frequency response,

$$x[n] = 0.5\cos(\omega n) \rightarrow y[n] = 0.5\left|H\left(e^{j\omega}\right)\right|\cos(\omega n + \angle H(e^{j\omega}))$$

Clearly, since the value of the magnitude response varies with the input frequency, the peak of the output sinusoid will also vary with the input frequency. Based on the frequency response shown above, the peak of the sinusoid could be as large as $(0.5)(2.5) = 1.25$ or as small as $(0.5)(0.25) = .125$. To be safe and avoid overflows, we should consider the worst case (largest) gain when designing our filter. Therefore, we should select an output format

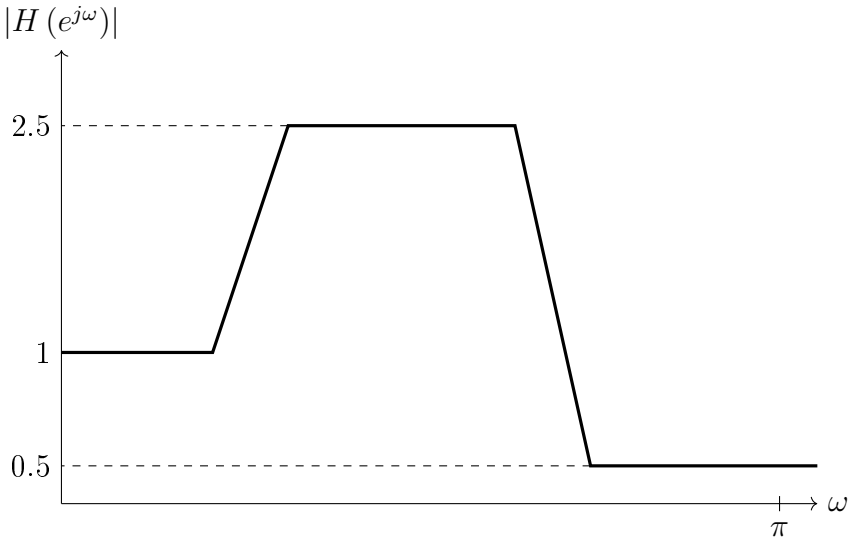

Figure 1.12: Magnitude response for a hypothetical digital filter.

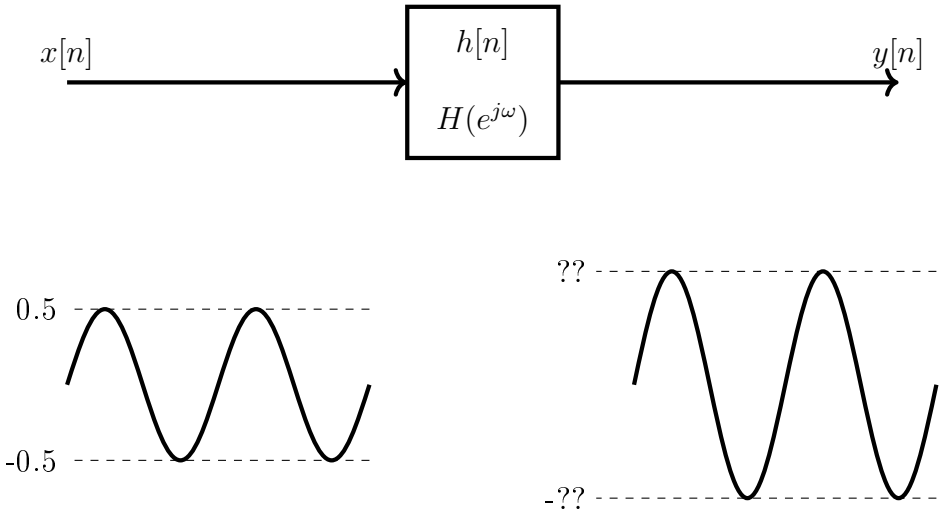

Figure 1.13: Input and output signals for a hypothetical digital filter.

which is capable of holding values as large as ± 1.25, which implies that at least 2 integer bits are required.

Note that if the filter coefficients are scaled, the magnitude response will also be scaled by an equivalent amount, which will change the maximum output value by the same scaling factor.

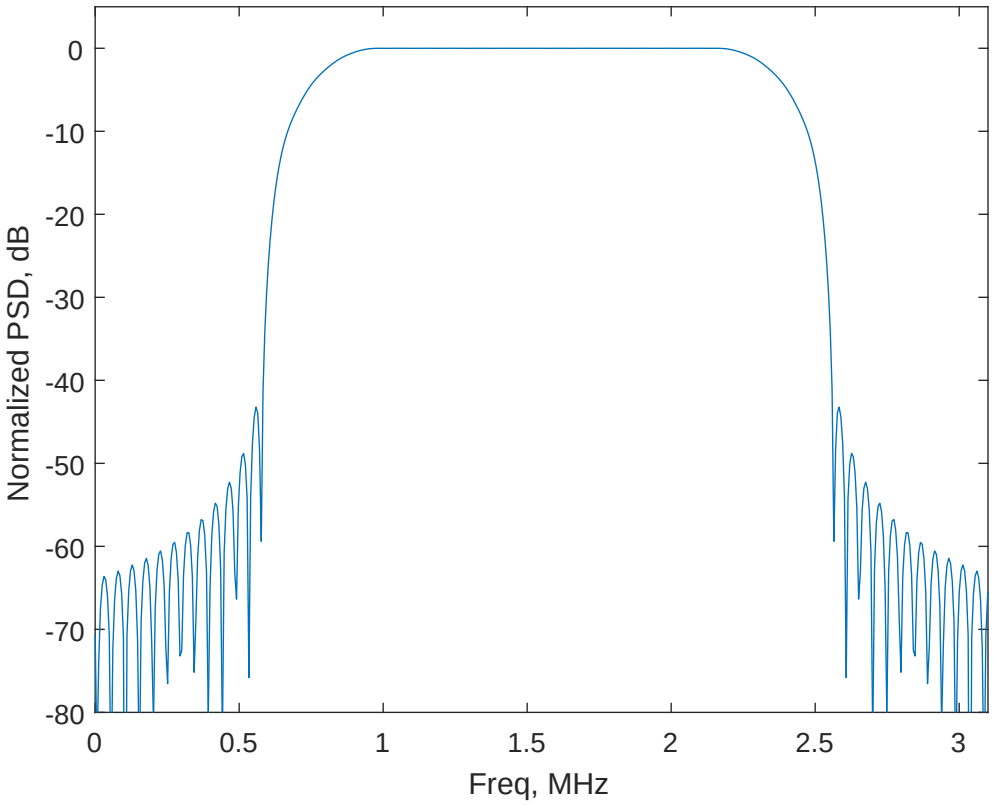

Figure 1.14: Example frequency spectrum of a QAM communication channel

Bit Growth with Random Inputs

Practical communication systems, generally handle signals that are much more complex than the sinusoidal inputs considered above. In fact, it is the complex and random nature of these signals that makes them suitable for transmitting information. As an example, consider a QAM channel transmitted over coaxial cable, which has a frequency spectrum similar to that shown in Figure 1.14:

Since the signals used in communications systems are random in nature, the outputs of filters that are processing these signals are also random. The method for calculating bit growth described above requires the input to be a predictable sinusoidal signal, so it does not apply to scenarios in which a practical communications waveform is being filtered.

The following scenario will be considered in class to illustrate the reasoning involved when designing an FIR filter which must handle a random or pseudorandom input signal:

A filter with impulse response $h[n] = [1, \ -0.5, \ 0.75]$ is used to process an input signal which is random. The signal is known to fit inside a 2s8 input word, but its actual contents will fluctuate unpredictably from sample to sample. How many integer bits are needed at the filter's output to guarantee there will never be any overflows?

Questions for the class to discuss:

- How can we mathematically express the filter output in terms of its input?

- Graphically express the filter structure using a block diagram or signal flow graph.
- What range of values can be represented in a 2s8 input?
- Consider the output of each multiplier (or equivalently each of the inputs to the adder chain which follows the multipliers). What is the largest possible value that can be obtained for each intermediate signal?
- What is the largest possible output value that can be obtained by summing the intermediate values discussed above?
- What sequence of input values is required in order to generate this largest possible output value?
- How would the result change if an extra filter coefficient is added to the filter, making the impulse $h_{new}[n] = [1, \ -0.5, \ 0.75, \ -0.2]$?
- Can we find a general mathematical representation for the maximum possible output value from a filter based on its impulse response and its input signal format?

2.4 Debugging in Modelsim

The following diagram illustrates the steps involved in compiling and testing an FPGA design.

As shown in the Figure 1.15, there are three main debugging methods:

- Signal Tap (JTAG debugging)
- Functional Simulation (using Modelsim)
- Timing Simulation (using Modelsim)

While functional simulations and timing simulations both use Modelsim, they are quite different in nature. A timing simulation, which is the type of simulation studied in CME341, operates using a low level structural representation of the routed design. The structural representation is typically stored in a file with extension .vo that is output by Quartus during compilation. A timing simulation models all of the internal logic propagation delays within the FPGA.

On the other hand, a functional simulation directly uses the Verilog source code that was written by the designer. A functional simulation does not model any internal propagation delays, which means all signal transitions and computations happen instantaneously when triggered by a clock edge or a change in the value of an input signal. While this has some minor disadvantages, functional simulation is by far the most common debugging tool in industry because:

- It is the fastest debugging technique by far. There is no need to run through the whole compilation flow to generate the programming file (for Signal Tap) or the timing simulation .vo file. As the size of the design increases, compilation can take a long time, so being able to simulate without running a full compilation can save many hours of time in the lab.

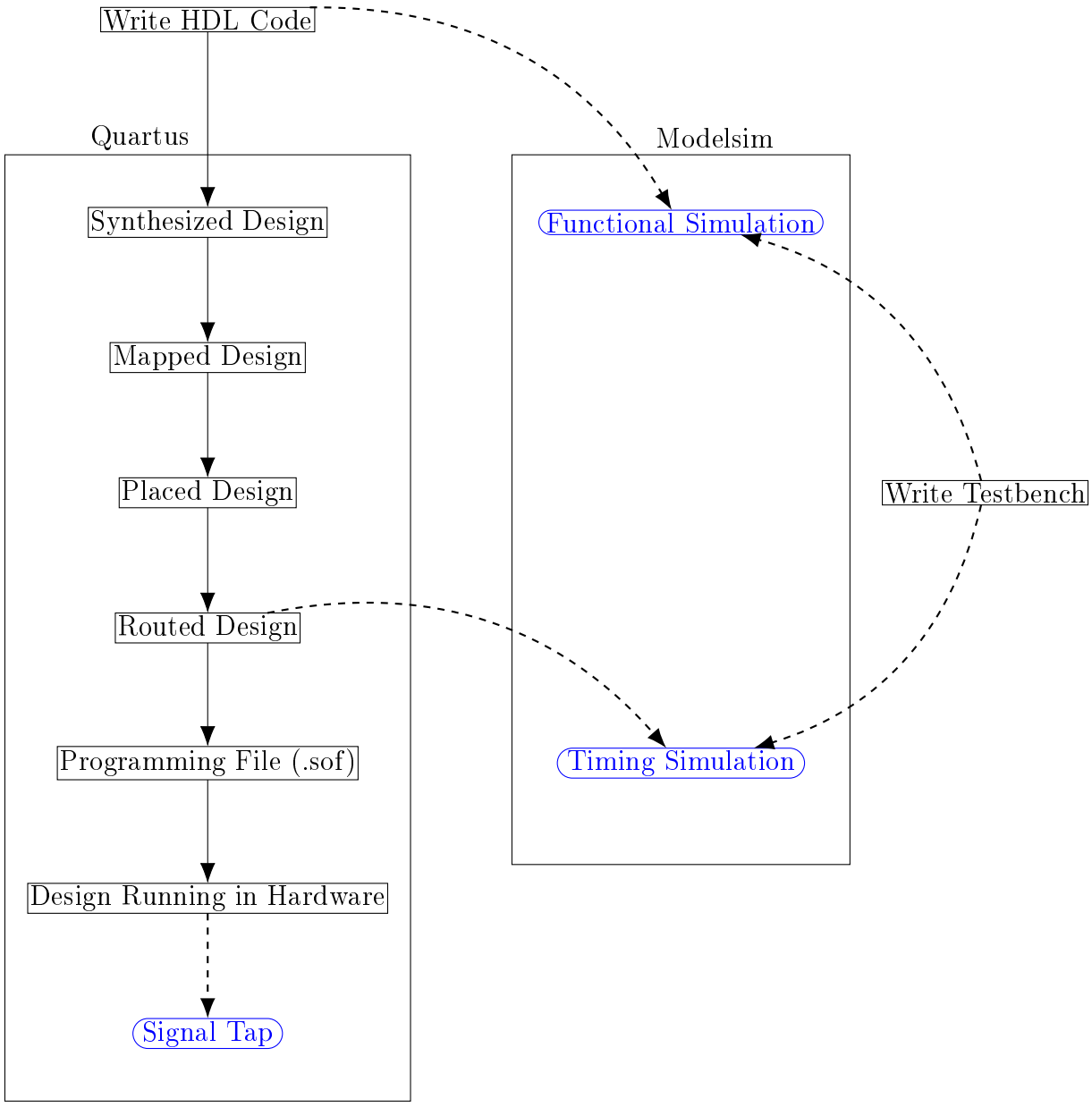

Figure 1.15: Methods of debugging an FPGA design.

- There is no need to recompile the entire design in order to bring additional signals in the debug waveform viewer.
- Modelsim includes waveform comparison and viewing tools which are far more powerful than those provided by Signal Tap.
- Modelsim can be configured to perform automatic checking and reporting of test results with little or no manual intervention required.

In EE465, students are strongly encouraged to use functional simulation to assist in the debugging process. In the past, students who have mastered this tool have saved a great deal of time throughout the term, have gotten further in the design, and have generally been more successful in the class.

Performing a Simulation in Modelsim

In order to perform a functional or timing simulation, it is necessary to write a testbench, which is a special Verilog module that acts as the top level entity for the simulation. The testbench is generally not synthesizable, and is intended only for use in simulation. The key responsibilities of the testbench are:

- to instantiate the module which is to be tested (module under test - MUT)
- to provide the MUT with the inputs required in order to execute the test
- to capture and process the outputs of the MUT

In practice, testbenches generally have no input or output ports, just a series of internal registers and wires. The testbench module can be viewed as a wrapper around the MUT. For each input port on the MUT, the testbench will have a corresponding register and a corresponding section of code which drives an appropriate sequence of values into that register. Similarly, for each output port on the MUT, the testbench will contain a wire to capture the output values.

Once a testbench has been created, a series of commands need to be run in Modelsim in order to execute the simulation. While it is possible to perform these commands by clicking on appropriate buttons in the Modelsim GUI (as was done in CME 341), it is generally desirable to set up a script to automate this repetitive process. An example of such a script has been generated by the instructor will be posted on the course website. For reference, it is also shown below:

```
# EE 465 Modelsim compilation script
# Instructions: 1.  Set the variables below to values appropriate for your design.
#                   The rest of the script should work correctly without
#                   any modifications.
#               2.  In Modelsim, navigate to the location of this script.
#                   (File menu -> Change Directory) or use 'cd' commands.
#               3.  Run 'do compile.do' on the Modelsim console.
```

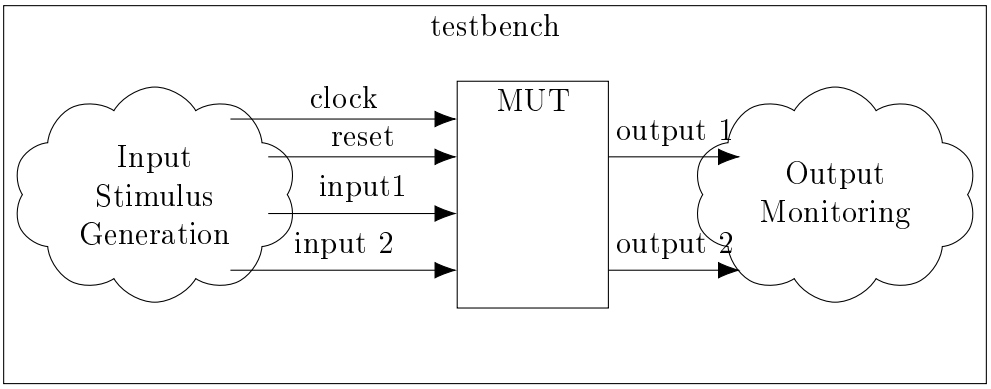

Figure 1.16: General structure of a testbench

```
#               4.  Add signals to waveform viewer as desired.
#               5.  Save waveform set up file as 'wave.do' using 'Save format...'
#                   from the file menu.
#                   Place it in the same directory as the compile.do file (this file).
#               6.  After adding signals, you may need to rerun 'do compile.do'
#                   to get the values to show up.
#               7.  Use the waveform window to debug your design as necessary.
#                   Try looking at the inputs and outputs of your module and
#                   any relevant intermediate signals.

# Variables to configure
set SIMULATION_LENGTH 10ms
set SOURCE_DIR "./"
set TB_DIR "./"
set TB_MODULE "test_tb"

# End of variables to configure

onerror {resume}
transcript on

# set up compilation library
if {[file exists rtl_work]} {
    vdel -lib rtl_work -all
}
vlib rtl_work
vmap work rtl_work
```

```
# compile source files
vlog -sv -work work ${SOURCE_DIR}/*.v

# compile tb file
vlog -sv -work work ${TB_DIR}/${TB_MODULE}.v

# initialize simulation
# add other libraries if necessary with -L lib_name
# if simulating megafunctions, add libraries specified by Quartus
vsim -t 1ns -L work ${TB_MODULE}

# open waveform viewer and populate with saved list of signals
do wave.do

# run simulation for specified amount of time
run ${SIMULATION_LENGTH}
```

An example testbench and a simple test module are provided on the course website along with the compilation script. These files can be used to get a feeling for how Modelsim works and how testbenches can be set up.

Modelsim Debugging Tips

Clock and reset generation can be done using hardcoded delay values inside initial blocks. For example, the following code generates a clock signal with a specified period.

```
localparam PERIOD = 10;
reg clk;

// Clock generation (OK to use hardcoded delays #)
initial
begin
  clk = 0;
  forever
    begin
      #(PERIOD/2);
      clk = ~clk;
    end
end
```

Note that any other input stimulus signals that need to be generated (aside from clocks and resets) should not use hardcoded delay values, as doing so can lead to some unexpected behavior. Due to the lack of propagation delays, many signals in a functional simulation tend to toggle simultaneously. Modelsim can occasionally get confused about which signal changes "come first", which can cause the simulation to produce incorrect results. The instructor will illustrate this effect during an in-class Modelsim demo.

To avoid this type of behavior, the input stimulus signals other than clocks and resets should be generated using `always @ (posedge clk)` blocks, as follows:

```
reg a;
always @(posedge clk)
  if(reset)
    begin
      a <= 1'b0;
    end
  else
    begin
      a <= ~a;
    end
```

Using this structure allows Modelsim to clearly identify that the signals are changing after (and in response to) the clock. Typically, more complex input signal patterns than that shown above will be required. One method of generating such signals is to generate a counter in the testbench which increments on clock edges, then assign values to the input stimulus signal based on the count value. For example:

```
reg [7:0] my_count;
reg my_input;

always @ (posedge clk)
  if(reset)
    begin
      my_count <= 8'd0;
      my_input <= 1'd0;
    end
  else
    begin
      my_count <= my_count + 1;
      if(my_count == <some constant value>)
        my_input = <some value>
      else
        my_input = <some other value>
    end
```

It is always a good idea to send a short (few clock cycles) reset pulse into your MUT at the start of the simulation. Provided all of your registers are resettable, this can prevent some problems which are caused by registers being initialized with don't care (x) values by Modelsim at the start of the simulation.

High impedance (z) and don't care (x) values in the waveform viewer often indicate that a signal isn't connected properly. This is sometimes caused by a testbench signal being generated with a width that doesn't match the corresponding signal in the MUT. Try to look at intermediate signals in order to verify proper connectivity and investigate the source of the problem. This can help to track down some very tricky bugs.

2.5 Review of Verilog operations involving signed and unsigned numbers

`These notes have been added at the request of Ian Abbott from class of 2016-2017.`

Hardware elements like adders, comparators and multipliers have well defined functions, but the hardware constructed from Verilog HDL depends on whether or not the inputs to the function are signed or unsigned numbers. Hardware built for unsigned inputs will not work if the inputs are signed numbers and visa versa. While this material was covered in CME341 and EE365, it slipped the minds of several students in the class of 2016-2017 and was the cause of several hours of frustration.

These notes are a review of the Verilog HDL rules that affect the hardware being built.

Signed and unsigned vectors

First it is pointed out that the Verilog 95 standard did not support signed numbers. Signed numbers were introduced in Verilog 2001. Verilog was and still is biased toward unsigned numbers and unsigned arithmetic.

In Verilog 2001 vectors could be declared as "signed" as part of the data type declaration. For example

```
reg signed [17:0] y;
wire signed [17:0] x;
```

The default is unsigned. That is if the word "signed" is omitted then the vector is unsigned.

Verilog 2001 also introduced two functions, which are really compiler directives, that override the sign type of a vector when it is used on the right hand side of an assignment. The functions `$signed(x)` and `$unsigned(x)` provide these override functions. Examples of their usage follow

```
assign y = $signed(x);
always @ * y = $signed(x) + 18'sd1233;
always @ * y = $unsigned(x);
assign y = $signed(x)+x;
```

The function `$signed(x)` instructs the compiler to override the number type assigned to `x` and make it a signed number in this one instance. Consider the last of the assignment examples above, in which `$signed(x)` is summed with `x`. The summand, `$signed(x)`, is `x` treated as a signed number and the second summand is `x` treated as whatever number type was specified in the data type declaration.

NB: The function `$signed(x)` does not change `x` to a signed number, it simply informs the compiler that `x` is to be treated as a signed number. For example, if `x` is the unsigned number `x = 3'd7` it has a decimal worth of 7. However, `$signed(x)` has a decimal worth of -1. The difference comes when the MSB is 1'b1, in which case the decimal worth of the MSB of `x` has a positive decimal worth while the MSB of `$signed(x)` has a negative decimal worth.

Assignments

Assignments in Verilog are

```
assign y = x;
always @ * y = x;
assign y = 10'd63;
assign y = 10'sd63;
```

The hardware built by an assignment is a direct bit-by-bit connection starting with the least significant bits, which are the right most bits.

If the width of `y` is the same as the width of the vector on the right hand side all the bits are connected. It does not matter if the vector on the right hand side is signed or unsigned.

If the width of `y` is less than the width of the vector on the right hand side the most significant bits (i.e. the left most bits) of the vector on the right hand side that are in excess of the width of `y` are not used. Again it does not matter if the vector on the right hand side is signed or unsigned.

If the width of `y` is greater than the width of the vector on the right hand side then the connection depends on whether the vector on the right hand side is signed or unsigned. In either case the bits of the vector on the right hand side are connected to the least significant bits of `y`. It is the connections to the most significant bits of `y` that are in excess of the width of the vector on the right hand side that depend on whether the vector on the right hand side is signed or unsigned.

If the vector on the right hand side is unsigned then the excess most significant bits of `y` are connected to logic zeros. This type of leftward extension is referred to as zero extension.

If the vector on the right hand side is signed then the excess most significant bits of `y` are connected to the most significant bit of vector on the right hand side. This type of leftward extension is referred to as sign extension.

Examples

```
wire signed [2:0] xs=3'sd7;// xs gets 3'b111 (decimal worth -1)
wire [2:0] x=3'd7;         // x gets 3'b111  (decimal worth 7)
wire [4:0] y1, y2, y3;

assign y1 = xs; // y1 gets 5'h1F i.e. {xs[2],xs[2],xs}
assign y2 = x; // y2 gets 5'h07 i.e. {2'b0,x}
assign y3 = $signed(x); // y3 gets 5'h1F i.e. {x[2],x[2],x}
```

Note that the number types of `y1`, `y2`, and `y3` do not affect assignment. If `y3` is used as a summand its worth would be 31, while the worth of `$signed(y3)` would be -1.

Adders

The only difference between adders that handle signed summands and those that handle unsigned summands is the logic that generates the carry-out bit. **However, the rules for Verilog addition make it so the carry out logic is never built.** The rules for addition

in Verilog leftward extend the summand vectors to be the same size as the sum vector so that the carry-out bit is not used and the carry-out logic is not built.

The extension rules, which are biased toward unsigned addition, are simple: If one or both of the summand vectors are unsigned both are treated as unsigned numbers and zero extended to the width of the sum vector. If both summand vectors are signed both are treated as signed numbers and sign extended to the width of the sum vector.

Examples

```
wire signed [2:0] xs=3'sd7;// xs gets 3'b111 (decimal worth -1)
wire [2:0] x=3'd7;         // x gets 3'b111  (decimal worth 7)
wire [4:0] y1, y2, y3, y4, y5, y6 y7;

assign y1 = xs + xs; // y1 = {xs[2],xs[2],xs} + {xs[2],xs[2],xs}
                     // y1 = 5'h1F + 5'h1F = 5'h1E

assign y2 = $signed(x) + xs;  // y2 = {x[2],x[2],x} + {xs[2],xs[2],xs}
                              // y2 = 5'h1F + 5'h1F = 5'h1E

assign y3 = x + xs;  // y3 = {0, 0, x} + {0, 0, xs}
                     // y3 = 5'h07 + 5'h07 = 5'h0E
```

If one or both of the summands is itself the result of an expression, there appears at first glance to be some ambiguity regarding the order of operations. As an example, consider the following lines of code:

```
assign y4 = -x + xs;
assign y5 = -xs + &x;
```

Based on the above discussion, it is clear that the summands will be extended to 5 bits when computing `y4` and `y5` and that the overall expressions will be treated as unsigned. However, one might question whether the two's complement in `y4` and bit reduction in `y5` are performed before or after the extension. It is left to the reader to verify that these two sequences of operations result in different results (`y4 = 5'b01000` vs `y4 = 5'b00000`, respectively).

When resolving scenarios involving sub-expressions like the one above, the Verilog standard defines two types of operands: context-determined and self-determined. The types differ in how the bit widths of the sub-expressions are chosen.

The bit width used when evaluating a context-determined operand is determined based on the largest signal in the entire line of code (not just the sub-expression). The unary negation (two's complement) operator used when computing `y4` above is a context-dependent operator, so the two's complement is performed as a 5 bit operation (since `y4` is 5 bits), which requires `x` to be extended to 5 bits before the two's complement is taken. As a result, `y4` is computed is follows:

```
assign y4 = -x + xs;    // y4 = -{0, 0, x} + {0, 0, xs}
                        // y4 = 5'b11001 + 5'b00111 = 5'h00
```

In contrast, the bit width used when evaluating a self-determined operand is determined based on only the sub-expression itself. For example, the bit reduction (&) operator used in the computation of `y5` is a self-determined operator and so the bit width of the bit reduction depends only on the length of `x`. The three bits of `x` are AND'ed together, generating a single bit result, which is then extended to 5 bits in preparation for the addition. Therefore, the complete computation of `y5` is as follows:

```
assign y5 = -xs + &x;    // y5 = -{0,0,xs} + {0,0,0,0,&x}
                         // y5 = 5'b11001 + 5'b00001 = 5'b11010 == 5'h1A
```

In general, operators which perform mathematical functions tend to act as context-dependent operators, while operators which perform logical functions (&, ~, |, !, &&, ||) tend to be self-dependent. For a complete list of operator types (as well as other interesting potential pitfalls in Verilog), please refer to:

http://www.lcdm-eng.com/papers/snug06_Verilog%20Gotchas%20Part1.pdf.

To avoid unexpected results due to context-dependent vs self-dependent sub-expressions, it is suggested students attempt to break down complex lines of code involving sub-expressions into multiple smaller lines of code using intermediate variables. This may result in code that is easier to read and less prone to confusion about the lengths and types of operations. For example, the `y4` computation above could be rewritten as:

```
assign x_neg = -x;
assign y4 = x_neg + x_s;
```

Additional examples:

```
assign y6 = -xs + $signed(&x); // y6 = -{xs[2],xs[2],xs} + { {5{&x}} }
                               // y6 = 5'h01 + 5'h1F = 5'h00
assign y7 = -xs + $signed({1'b0, &x});
                              // y7 = -{xs[2],xs[2],xs} + { 4'b0, &x }
                              // y7 = 5'h01 + 5'h01 = 5'h02
```

Note that `y5, y6` and `y7` all deal with a 1 bit logic signal that results from the reduction and of `x`, i.e. from `&x`. If the intent is to increment the signed number $-$`xs` by 1 if &`x` $= 1$, then `&x` must be zero extended by at least 1 bit and the zero extended quantity must be placed in the argument of `$signed()` so that it is interpreted as a positive signed number.

Comparators

Logic that compares signed numbers is quite different than logic that compares unsigned numbers. For example, the signed comparator synthesized by "`3'sb111 > 3'sb011`" makes the decimal worth comparison "$-1 > 3$" and produces an output of 1'b0. The unsigned comparator synthesized by "`3'b111 > 3'b011`", has exactly the same binary inputs, but makes the decimal worth comparison of "$7 > 3$" and produces an output of 1'b1.

The rules for synthesizing comparators in Verilog HDL are:

1. If one or both of the inputs have an unsigned format, then an unsigned comparator is synthesized.

2. If and only if both inputs have signed formats will the synthesizer build a signed comparator.

3. If an unsigned comparator is synthesized and one of the input vectors is shorter than the other, the shorter vector is leftward extended to the size of the longer with zeros.

4. If a signed comparator is synthesized and one of the input vectors is shorter than the other, the shorter vector is leftward extended to the size of the longer by sign extension.

Multipliers

The hardware circuitry that implements a multiplier for a signed inputs is quite different than the circuitry needed to implement a multiplier with unsigned inputs. The rule used in Verilog is if one or both inputs are unsigned numbers then an unsigned multiplier is constructed. If both inputs are signed then a signed multiplier is constructed. The product of an unsigned number is an unsigned number and the product of a signed multiplier is a signed number.

The input vectors of a Verilog multiplier are always interpreted as integers. The product vector is also an integer i.e. the value of the least significant bit is 1. The width of the product vector is the sum of the widths of the inputs. The most significant bit of the product of two signed numbers can be discarded in all by the special case where both inputs are negative full scale. This case so rarely occurs that the most significant bit of a signed product can be safely discarded. All the bits in an **unsigned product** are needed so the **MSB can not be discarded**.

If the inputs to a multiplier are not integers, but contain a binary point and have fraction bits, then the product will also contain a binary point located so that the number of its fraction bits is equal to the sum of the fraction bits of the input.

Actions that change the number type

As previously explained, Verilog is biased toward unsigned numbers. Any time a subset of the bits in a vector are reference, that subset is treated as an unsigned number regardless of whether the vector is signed or unsigned. Also any time a vector is expanded with a concatenation operator, whether or not all the numbers in the concatenation are signed, the expanded vector is treated as an unsigned number. For example

```
reg signed [17:0] xs;
reg signed [17:0] ys;

xs[17:0]     // this is viewed as a subset of xs and therefore
             // treated as an unsigned number
xs[16:0]     // unsigned number
{ 1'sb0, xs} // unsigned number
ys * xs[16:4]  // builds an unsiged multipler
```

```
ys + xs[14:0]  // xs[14:0] is unsigned so it is extended leftward
               // with zeros if the width of the sum exceeds 15 bits
```

Synopsis Design Constraints

Often there is combinational logic between the output of one register and the input of another. It is important that the propagation delay through the logic circuit not exceed the clock period. Should the propagation delay in the logic circuit exceed the clock period then the data transferred into the down steam register will be in error.

The Quartus compiler will do its best to synthesize a circuit that meets timing if it knows the clock period or clock frequency. If it can not meet the timing the TimeQuest analyser in Quartus will print a warning indicating as much. The designer then has to introduce pipelining in the data path to the design to work. The TimeQuest analyser can not print such a warning if it does not know the period or frequency of the clock.

The clock information needed for the TimeQuest analyser to do its job must be in a text file with a .sdc extension and that file must be in the project folder.

Two clocks are used in this class: `CLOCK_50` or perhaps `clock_50`, which originates from a pin and `sys_clk`, which is derived from `CLOCK_50`. The .sdc file needed for the project in this class is

```
create_clock -name {input_clock} -period 20.000 [get_ports {CLOCK_50}]
create_generated_clock  -name {sys_clock} [get_registers{sys_clk}]
                       -divide_by 2 -source[get_ports CLOCK_50]
```

If `sys_clk` is in another module then `[get_registers{sys_clk}]` must be changed to `[get_registersmodule_name:instance_name|sys_clk]`.

As described in more detail in the timing analysis discussion associated with Deliverable 3, it is also recommended to add the following commands to your SDC file in order to prevent warnings related to unconstrained paths from masking real timing errors as your design grows in size.

```
# adding false path constraints to all input ports
set_false_path -from [get_ports ADC_DA*]
set_false_path -from [get_ports ADC_DB*]
set_false_path -from [get_ports KEY*]
set_false_path -from [get_ports SW*]

# adding false path constraints to all output ports
set_false_path -to [get_ports ADC_CLK_*]
set_false_path -to [get_ports DAC_CLK_*]
set_false_path -to [get_ports DAC_DA*]
set_false_path -to [get_ports DAC_DB*]
set_false_path -to [get_ports DAC_WRT*]
set_false_path -to [get_ports LEDG*]
set_false_path -to [get_ports LEDR*]
```

Note that the port names above should match those in your top level Verilog file. If this is not the case, you may need to slightly modify the port names in the "set_false_path" commands in your SDC file.
End of Notes on Verilog Review

2.6 Common Errors from Previous Years

This section will list some common errors that many students have run into in the past and suggest ways to avoid them.

Inferred Single Bit Wires

In Verilog, variables are typically declared before they are used. If a designer attempts to use a variable that has not been declared, the result is generally a compilation error. However, there is one exception to this rule: Verilog by default allows designers to implicitly declare single bit wires by assigning values to them. This is dangerous, and can be the source of some very frustrating bugs.

For example, consider the following code:

```
module test (
  input wire a,
  input wire b,
  output wire c
);

assign cv = (a & b); // typo : was supposed to be c

endmodule
```

This code will not generate an error and will actually create a wire named `cv` which contains the result `a & b`. Of course, the wire `cv` doesn't actually connect to anything downstream, so it will just end up being pruned away by the compiler. The `c` output from this module will be connected to 0 instead of the intended `a & b`, which will likely cause problems with some downstream logic.

The simplest way to prevent this type of error is to add the statement

```
‘default_nettype none
```

as the first line of every Verilog source file. This statement tells the compiler to throw and error instead of inferring a wire when an undeclared variable is used in the code. The error can be easily used to identify the typo and fix the code.

It should be noted that some third-party code and/or Quartus Megafunctions may rely upon the inferred wire "feature". Therefore, it is prudent to restore the default behavior after our module has been compiled by placing the following statement as the last line of every Verilog source file.

```
‘default_nettype wire
```

Filter Coefficients in Simulation

The following code snippet illustrates a method of coding the constant coefficients required in FIR filters which has been used by many students past years.

```
reg [17:0] coeff_0;
reg [17:0] coeff_1;
// etc...

always @ (*)
  begin
    coeff_0 <= 18'h1_2345;
    coeff_1 <= 18'h2_beef;
    // etc...
  end
```

Unfortunately, there is a problem with this code which causes it to not behave as desired in simulation, which is often confusing to students, as the code functions functions "correctly" in synthesis/hardware.

To understand the reason for this mismatch, first recall that the values inside the parentheses in the `always @ (*)` statement are referred to as the "sensitivity list". The sensitivity list is used to control the execution of the always block. Specifically, when the value of any signal included in the sensitivity list changes, the always block "fires" and its statements are executed.

Since it is easy to make an error when manually inserting a long list of signals into the sensitivity list of an always block, Verilog provides the `*` shorthand, which instructs the interpreter to automatically set up an appropriate sensitivity list for the always block in question at compile time. The rule which is followed by the interpreter when constructing this sensitivity list can be expressed as follows: "Include the names of all signals which appear on the right hand side of an expression."

In the code snippet above, notice that the signals which appear on the RHS of the expressions inside the always block are all constants. Since the verilog simulator never sees the values of any of the signals in the sensitivity list change (because constants by definition never change), it never executes the statements within the always block. As a result, the registers remain at their default startup value of `x` (don't care) for the duration of the simulation.

There are a number of possible strategies for working around this issue. The first is to use an `initial` block to assign the values, as follows. Recall that an initial block is executed at the start of a simulation, which ensures that the coefficient registers take on the desired values immediately.

```
reg [17:0] coeff_0;
reg [17:0] coeff_1;
// etc...

initial
```

```
begin
  coeff_0 <= 18'h1_2345;
  coeff_1 <= 18'h2_beef;
  // etc...
end
```

An alternative solution is to declare the coefficients as type `wire` and then use the `assign` statement to implement the assignment instead of a combinational `always` block. This option is illustrated below.

```
wire [17:0] coeff_0;
wire [17:0] coeff_1;
// etc...

assign coeff_0 = 18'h1_2345;
assign coeff_1 = 18'h2_beef;
// etc...
```

Yet another solution is to define the coefficients as parameters within the module using the `localparam` functionality as shown below. Note that parameters are assigned in the same line they are declared and cannot be modified later in the code.

```
localparam [17:0] coeff_0 = 18'h1_2345;
localparam [17:0] coeff_1 = 18'h2_beef;
// etc...
```

Timing Errors

As you progress through your EE 465 design, you may occasionally see timing errors being reported by the TimeQuest Analyzer in Quartus. In such cases, your design will likely either not work at all in the FPGA or else will work only intermittently. This has been a cause of much frustration and many "wasted" hours in the lab for students in previous years. Thus, it is strongly recommended that you monitor the TimeQuest reports in order to verify that all timing requirements are being met.

If timing errors are reported (especially setup errors), you should inspect the path which is in error, verify that your SDC file is correct, and add pipelining as necessary until Quartus reports that the design fully meets timing.

Timing errors will likely become more of an issue in later deliverables, as the utilization of the FPGA resources for your design increases.

3 Lecture / Discussion Topics for Deliverable 1

3.1 Raised Cosine & Square Root Raised Cosine Filter Discussion

The first "discussion oriented" component of the lectures centers on the use and exploration of a Matlab script that is made available on the class website.

The Matlab script generates the coefficients for a finite-length raised cosine (RC) filter with a 6 dB bandwidth of $F_s/(2N_{\text{sps}})$ and also the coefficients for a finite-length SRRC filter with a 3 dB bandwidth of $F_s/(2N_{\text{sps}})$, for $N_{\text{sps}} = 4$. Here N_{sps} means the number of samples per symbol.

This Matlab script also generates the theoretical power spectra for the infinite-length RC and SRRC filters.

If during the discussion the class decides it needs some graphical or numerical information then the Matlab script can be modified in class, providing it can be written quickly, and the results displayed on the screen.

The Matlab script for the discussion is given below:

```
clear all

% parmeters for the filters
 N_sps = 4;   % number of samples per symbol
 beta = 0.25; % roll off factor, script requires
              % beta be less than 1
 N_rc = 33;   % length of the impulse response
              % of the raised cosine filter
 N_srrc = 17; % length of the impulse response of the
              % square root raised cosine filter
 F_s = 1;     % sampling rate in samples/second
 f_6db = 1/2/N_sps; % 6 dB down point in cycles/sample
 F_6db = F_s * f_6db; % 6 dB down point in Hz

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% time and frequency vectors
%
df = 1/2000;   % frequency increment in cycles/sample
f = [0:df:0.5-df/2]; % cycles/sample; 0 to almost 1/2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  magnitude response for RC and SCCR filters
Hrc_f = zeros(1,length(f)); % reserve space for
           % magnitude response of rc filter
Hsrrc_f = zeros(1,length(f)); % reserve space for
           % magnitude response of srrc filter
f1 = find(f < f_6db*(1-beta)); % indices where
                           % H_f = 1
f2 = find( (f_6db*(1-beta)<= f) & ( f <=...
          f_6db*(1+beta))); % indices where
                           % H_f is in transition
f3 = find(f > f_6db*(1+beta)); % indices where
```

```
                          % H_f = 0
Hrc_f(f1) = ones(1,length(f1));
Hrc_f(f2) = 0.5+0.5*cos(pi*(f2-f2(1))/(length(f2)-1));
Hrc_f(f3) = 0;

Hsrrc_f = sqrt(Hrc_f);

figure(1);
     plot(f,Hrc_f,'r', ...
          f,Hsrrc_f,'--b','LineWidth',2);
     xlabel('frequency in cycles/sample')
     ylabel('|H_{rc}(e^{2\pif})| and |H_{srrc}(e^{2\pif})|')
     grid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% find and plot the impulse responses

h_rc=firrcos(N_rc-1,F_s/8,beta,F_s,'rolloff');
    % impulse response of rc filter
h_srrc=firrcos(N_srrc-1,F_s/8,beta,F_s,'rolloff','sqrt');
    % impulse response of rc filter

figure(2)
plot(0:N_rc-1,h_rc,'r*', ...
     0:N_srrc-1,h_srrc,'bd', 'MarkerSize',8);
ylabel('h_{rc}[n] and h_{srrc}[n]');
xlabel('n');
grid;

% Find and plot the frequency repsonses of the
% finite length RC and SRRC filters

H_hat_rc = freqz(h_rc,1,2*pi*f);
H_hat_srrc = freqz(h_srrc,1,2*pi*f);

figure(3)
plot(f,abs(H_hat_rc),'r', ...
     f,abs(H_hat_srrc),'--b','LineWidth',2);
ylabel('H_{hat}(\Omega) for RC and SRRC');
xlabel('\Omega');
grid;
```

Some questions/topics of discussion for Deliverable 1 are given below:

1. What structure should be used to implement an SRRC filter.

2. How can this structure be tested in ModelSim. Can the impulse response be measured in Modelsim.

3. What is the 3 dB bandwidth of a SRRC filter with 4 samples symbol.

4. A communication system has a transmitter, a channel and a receiver. What is the purpose of the SRRC filter in the transmitter.

5. What is the purpose of the SRRC filter in the receiver.

6. At the transmit end of a communication system one symbol is generated every N_{sps} samples (for this discussion $N_{\text{sps}} = 4$ samples per symbol), with zeros inserted between symbols. Of course this sample sequence is filtered and then transmitted and then received and then filtered again. The filters must be designed so that every Nsps sample of the output of the filter in the receiver is equal to one of the data samples that were transmitted.

   How is this possible?

7. A raised cosine filter in in a class of filters referred to as Nyquist filters. The impulse response of Nyquist filters are special.
   What is special about them?

   The impulse response for a raised cosine filter is given below

$$h[n] = \begin{cases} \frac{\pi}{4N_{\text{sps}}}\text{sinc}\left(\frac{1}{2\beta}\right), & n = \pm\frac{N_{\text{sps}}}{2\beta} \\ \frac{1}{N_{\text{sps}}}\text{sinc}\left(\frac{n}{N_{\text{sps}}}\right)\frac{\cos\left(\frac{\pi\beta n}{N_{\text{sps}}}\right)}{1-\left(\frac{2\beta n}{N_{\text{sps}}}\right)^2}, & \text{otherwise,} \end{cases}$$

   where β is the roll-off factor.

8. What is critical about the 6 db bandwidth of the cascade of the transmit and receive SRRC filters and how does that relate to N_{sps}?

9. The Fourier transform of the product of two sequences is $1/(2\pi)$ times the circular convolution of the Fourier transforms of the two sequences with the interval of the circular convolution being 2π. What does that mean? What is the graphical interpretation of circular convolution? The Matlab function `cconv(x,y,N)` computes the circular convolution of two sequences. Can this be used to find the convolution of two continuous functions of frequency? Is $\texttt{cconv(x,y,N)} \times \Delta f$, where Δf is the sample spacing, a Riemann approximation to the analog convolution $X(f) \star Y(f)$?

10. If the impulse response of filter is multiplied by another sequence, say a window of some sort, what happens in the frequency domain. To start the discussion suppose the impulse response of an ideal low pass filter with bandwidth $F_s/8$ is multiplied by the Fourier transform of the positive half of a cosine that has zero crossings at $\pm F_s/16$. Is the resulting sequence the impulse response of a raised cosine filter, if so what is the roll-off factor.

    Will every 4th sample in the resulting impulse response be zero?

    Will every 4th sample in the result be zero for any type of window?

11. If an RC filter is windowed by a Hamming window to make the impulse response finite, say a length of 33, will every 4th sample still be zero? Why or why not?

    What does windowing do the transition band and what happens to the out-of-band power, i.e., the stop band power.

12. In theory an RC filter is equivalent to two SRRC filters in cascade. Are two length-17 rectangular windowed SRRC filters in cascade the same a one length 33 RC filter? What is the difference?

13. Define inter-symbol interference. Can it be measured from the impulse response? Is the rms level of ISI proportional to the amplitude of the signal?

14. Define MER. Is it a better measure of filter performance than ISI? Is MER, expressed in dB, given by

$$10 \log \frac{\mathrm{P_{rcvd\_data}}}{\mathrm{P_{ISI}}}?$$

    What is $P_{\mathrm{rcvd\_data}}$ w.r.t. P_{ISI} in dB?

15. What is an easy way to measure the ISI generated by two length-17 SRRC filters in cascade? Will windowing the transmit filter, say by a hamming window, increase the system ISI? Is so why is the transmit filter windowed?

16. Can the ISI caused by windowing the impulse response of a filter be minimized by adjusting the roll-off factor and/or the 3 dB bandwidths in the SRRC filters in the transmitter and/or the receiver. If so, how would one find the optimum adjustments?

17. After filters are constructed in Verilog, their impulse responses need to be verified. The first step is to verify the impulse response using ModelsimAltera. The filters can be quite long so hand checking the value of $h[n]$ for every n where $h[n] \neq 0$ can be tedious. Is there any easy way to check the impulse response for typographical errors in entering the coefficients (not to verify it meets spec)?

    Would looking at the output to a DC input be helpful?

    Would looking at the output to an input of alternating $+1$ and -1 be useful?

    Could such a quick check be implemented in ModelsimAltera?

    Could such a quick check be implemented in hardware using SignalTap?

18. Is it possible to import data from ModelsimAltera to Matlab? If so could that be helpful in debugging filters described in Verilog?

19. 16-QAM is generated using two 4-ASK signals that are filtered with separate SRRC filters, up-converted with quadrature sinusoids and then combined at IF. The input to each of the two filters in the transmitter consists of a data sample followed by $N_{\text{sps}} - 1$ zero samples. This means the output of just one of every N_{sps} multipliers is non-zero.

    Is there any way to share one multiplier among N_{sps} taps? Is there any way to reduce the number of registers used in the filter by a factor of N_{sps}?

20. Since the input to each of the two SRRC filters in the transmitter is a 4-ASK signal, the inputs can have only 1 of 4 values, which, for example, could be -3, -1, 1, or 3 every fourth sample. Is there any way to design the SRRC filter in the transmitter so that no multipliers are used, i.e., build a multiplierless filter.

21. The multiplierless filter described above works only for a 4-ASK input and is usually designed to accept the data going into the mapper, which is 2 bits, as opposed to a digital signal at the output of the mapper, which has many more bits. For example, a digital signal with a 1s17 format would have 18 bits.

    Can an impulse be applied to the multiplierless filter discussed above. An impulse, i.e. $\delta[n]$, would be the sequence $\ldots, 0, 0, 1 - 2^{-17}, 0, 0, \ldots$, where the non-zero value corresponds to $n = 0$. If not why not?

4 Lecture / Discussion Topics for Deliverable 2

The Deliverable 2 description which begins on page 103 is quite detailed, so the class time will mainly be dedicated to answering student questions and elaborating on the points raised in that section. It is anticipated that the students will take a proactive role in asking questions and requesting discussions on topics of particular use/interest to them. This may include the following topics:

4.1 Clock Generation

Methods of clock and/or clock enable generation
Advantages/disadvantages of multiple clocks vs clock enables

4.2 Data Generation

Need for data generation
Methods of data generation

4.3 Slicer Operation

General concept of a slicer
Need for a reference level in the slicer
Impact of reference level on MER result

4.4 Debugging strategies and hints from previous years:

- Since the full 2^{20}-symbol sequence used for the MER measurement can take a long time to simulate in Modelsim, it can be helpful to initially average over a shorter sequence (such as 8, 16, or 32 symbols). Of course, the MER estimates obtained using this shorter sequence will not be as accurate as those obtained from the full sequence, but this testing can help to quickly identify logical errors in students' Verilog code.

- Pay careful attention to variable data types and lengths when writing your Verilog code. Most of the issues students typically face on this deliverable can be traced back to incorrect use of variable types and lengths.

- Drawing a complete block diagram of the circuit to be constructed (including all signal types and word lengths) prior to writing any Verilog code has saved students large amounts of time in the past.

- It can be helpful to manually choose a short sequence of decision variables and compute by hand the expected result for each of the signals in the design (reference level, error, squared error, etc). A Modelsim simulation can then be set up to actually input those signals into your design. Comparing your hand-computed expected outputs to the actual values returned by Modelsim can be very helpful in tracking down errors in your code.

5 Lecture / Discussion Topics for Deliverable 3

5.1 Out of Band Power Requirements

Transmitters in communication systems are generally allowed to transmit over a fixed band of frequencies that is defined in a communication standard, such as the LTE Advanced standard for 4G wireless systems or the DOCSIS 3.1 standard for broadband cable internet systems. Ideally, the transmitter would transmit exactly zero power outside of the allocated frequency range. However, due to practical limitations, a transmitter will generally tend to emit a certain degree of power outside of its allocated channel. These unwanted emissions are known as **out of band emissions**.

The communication standards recognize that out of band emissions cannot be entirely eliminated. Rather, they impose restrictions on the magnitude of such emissions. For example, a standard may require that the out of band emissions inside a frequency band that is 1MHz wide and centered 1MHz away from the edge of a communication channel must have 40dB less power than the transmission within the allocated frequency range, as shown in

Figure 1.17. Note that the requirement is placed on the total power across the bandwidth (the areas in the Figure 1.17), not the PSDs of the various signals.

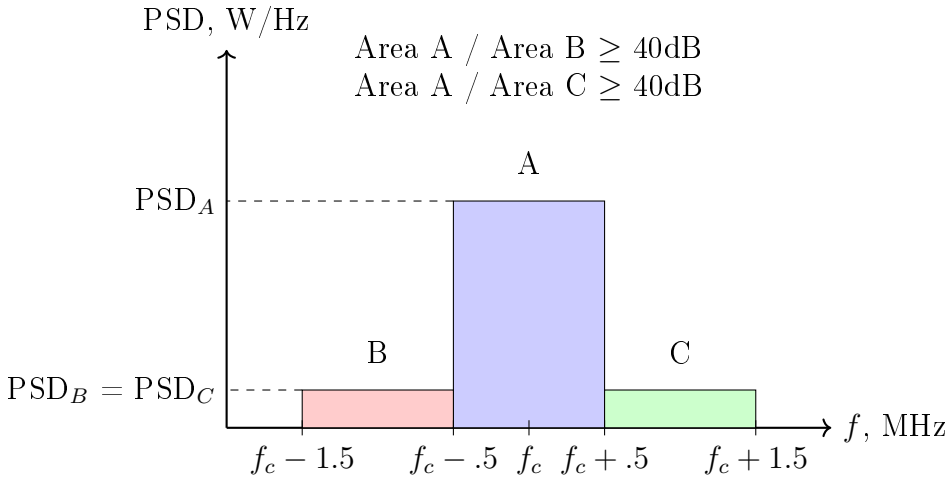

Figure 1.17: Illustration of an example Out of Band power requirement.

When building a transmitter, the designer must pay close attention to any out of band emissions requirements, as a failure to meet the requirements may prevent deployment of the transmitter. At the same time, a design which greatly exceeds the requirements is likely to be overly complex and costly, placing it at a competitive disadvantage in the marketplace. As is commonly the case in engineering, the key is to find the correct balance between these two extremes.

Some questions for discussion in class:

- Why are out of band requirements needed?
- What elements of a transmitter design are likely to impact out of band emissions performance?
- How can a designer verify that a design meets out of band emissions requirements?
- Referring to Figure 1.17, how do the area requirements relate to the PSDs of A, B, and C?
- Would the required PSD relationships change if the widths of bands B and C were doubled? How?

In Deliverable 3, students are asked to design pulse shaping and matched filters for a 16-QAM communication system. In doing so, one key factor to consider is how the pulse shaping filter impacts the out of band emissions for the transmitter. Therefore, it is important to understand how to first predict (during the design phase) and then measure (during the implementation phase) the out of band (OOB) emissions for a particular filter.

Measuring OOB Emissions in Matlab

If the class is interested, a student-driven in-class discussion will focus on how to measure OOB emissions for a given transmit filter in Matlab.

Measuring OOB Emissions with Spectrum Analyzer at RF

The simplest way of measuring OOB emissions involves connecting the transmitter's output to a spectrum analyzer, then using the spectrum analyzer's integrated bandwidth power measurement functions to compute the total power in the spectral bands of interest. This method requires the transmitter's output to be centered at an RF carrier frequency, rather than DC, so it requires the use of a complete transmitter (including an upconverter), as opposed to just a basic pulse shaping filter.

The specific set of steps which configure the spectrum analyzer to take OOB measurements is fairly lengthy. It has been documented in a video which is posted to the class website.

Measuring OOB Emissions with Spectrum Analyzer at Baseband

While the out-of-band emissions performance of the pulse shaping filter is specified at RF (centered around a carrier frequency), the pulse shaping filter is designed and operates at baseband (centered around 0 Hz / DC). A complete transmitter includes an upconverter which translates the baseband signal to a carrier frequency. Once the signal is at RF, a spectrum analyzer can be used to measure the out-of-band emissions of the signal as described in the previous section.

However, the upconverter and downconverter are not implemented until Deliverable 4. The approach taken in the lab exam for Deliverable 3 is for the instructor to provide a "known good" upconverter which can be instantiated in cascade with the student's transmitter. This allows the out-of-band emissions to be measured in the typical fashion previously discussed.

An alternative method of evaluating the filters designed and implemented in Deliverable 3 is to perform the measurements at baseband. This section will discuss the theory and methods for doing so.

Unfortunately there are a few issues associated with testing at baseband:

1. Baluns are used to convert an analog output from a chip with a single power supply to signal that can have a positive or negative voltage. Baluns (short for balanced to unbalanced converters) are transformers. In this case the primary winding is driven by a balanced current source from the chip with the single power supply and the secondary winding is connected to the SMA connector whose outside conductor is grounded making it unbalanced.

   The Baluns on the DAC/ADC daughter board are tiny transformers that, unfortunately, act as high pass filters with a gain of 0 at DC and a gain of 1 at high frequency. The 3 dB down point is somewhere around 100 kHz. At 300 kHz the pass band gain is nearly 1.

2. The spectrum analyser makes one-sided power measurements. This means when the spectrum shown in Figure 2.3 is translated to baseband (i.e. center frequency F_c is translated to 0) the power in the signal is obtained by measuring the power in the frequency band from 0 to 1.75/2 MHz. This also means the power in the two OB1 bands can not be measured separately. The power measured will be the sum of powers in the two OB1 bands. Therefore, the ratio of the signal power to the power in OB1 measured at baseband must be divided by 2 to get the ratio of the power in OB1 to the carrier power as illustrated in Figure 2.3.

The theoretical 3 dB bandwidth of the SRRC pulse shaping filter will be $F_s/8$. It so happens the noise bandwidth of a SRRC is its 3 dB bandwidth. (The noise bandwidth of a low pass filter is defined to be the bandwidth of an ideal brick wall low pass filter that has the same output power when both are excited with white noise.) This fact allows the power in the signal to be measured in a segment of the passband then prorating to get the total power in the signal. For example, the signal power could be measured in the band from 300 kHz to 300 kHz$+F_s/80$. The measurement bandwidth is then $F_s/80$, which is $(F_s/8)/10$ making it one tenth the noise bandwidth. The total signal power is therefore obtained by multiplying the measured power by 10 (or if the power is measured in dBm increasing it by 10 dB).

For Deliverable 3 the sampling rate is 6.25 MHz. The rate is changed to 25 MHz in Deliverable 4 by up-sampling, but it is only 6.25 MHz in Deliverable 3. This means only part of the out-of-band emissions specified at RF can be measured at baseband in Deliverable 3. The power in bands OB1 and OB2 can be measured, but the power in adjacent channel 2 can not.

There is yet one more problem. The RF specification assumes the $\sin(\pi F/F_s)/(\pi F/F_s)$ roll off caused by the DAC is corrected by a DAC correction filter in the FPGA. That filter will not constructed so the power measurements must take the DAC roll off into account.

The signal power was computed by extrapolating the power measured in the band between 300 kHz and 300 kHz $+ F_s/80$. This measurement band was chosen to be at low frequency so the DAC roll off would not have a significant effect. The roll off at the upper edge of this measurement band with respect to the roll off at DC is less than 0.06 dB.

The DAC roll off does, however, effect the OB1 and OB2 bands. The roll off in the center of the OB1 band is 0.36 dB. It would be reasonable to correct for the power measured in OB1 by adding 0.36 dB to the measurement.

The roll offs at the ends of OB2 are xxxx and yyyy. The difference is too large to use a single correction. It suggested that OB2 be divided into 5 bands and the power be measured in each band separately and then corrected using the roll offs for the center of each band.

5.2 Filter Optimization Techniques

Resource sharing in multirate filters

5.3 Debugging Strategies

Timequest and Timing Analysis

As the complexity of an FPGA design increases, it becomes more and more difficult for the compiler (Quartus in this case) to place and route the design elements in a manner which meets the setup and hold requirements for all registers in the design. This generally occurs for two main reasons:

1. The critical timing path of the circuit tends to become more complex. The concept of the critical path as the path between any two registers which contains the largest amount of combinational logic was introduced in the discussion prior to the kickstart lab. As the critical path becomes longer and more complex, the time it takes for a signal to propagate between the source and destination register increases, making the register setup requirements more difficult to achieve.

2. As the more and more of the available resources (FFs, LUTs, Multipliers, routing paths, etc) inside the FPGA are used by the design, it becomes more and more difficult for the compiler to place related logic elements close together. This tends to increase the propagation delays between elements, which makes the register setup time requirements more difficult to achieve.

One method of mitigating timing-related challenges is to use pipelining to reduce the amount of combinational logic between any two registers. A few methods of adding pipelining to circuits were introduced earlier in the term. Of course, pipelining the registers which are on the critical path will have the most impact on the compiler's ability to meet the timing requirements. However, this raises a few questions:

- In a complicated design, how can we identify the critical path so that we can focus our efforts appropriately?

- How can we tell whether a given Quartus compile has met our design's timing requirements?

- How are the timing requirements determined?

Fortunately, Quartus provides a utility called TimeQuest which can be used to help answer these questions. While TimeQuest is a very powerful tool, it can also be somewhat tricky and non-intuitive for a beginner. This section provides an overview of the Timequest tool and shows how it can be used to debug timing issues in an FPGA.

It should be understood that TimeQuest and timing analysis for FPGA designs in general is a deep and complex topic, so this introduction will only scratch the surface by highlighting a few best practices for use in EE 465 designs. Complete documentation and users guides (hundreds of pages!) are available online - relevant links will be posted on the class website.

Timing Closure Algorithm

Figure 1.18 illustrates the high-level procedure for achieving what is known in the industry as "timing closure", which essentially means that a design is meeting all timing requirements

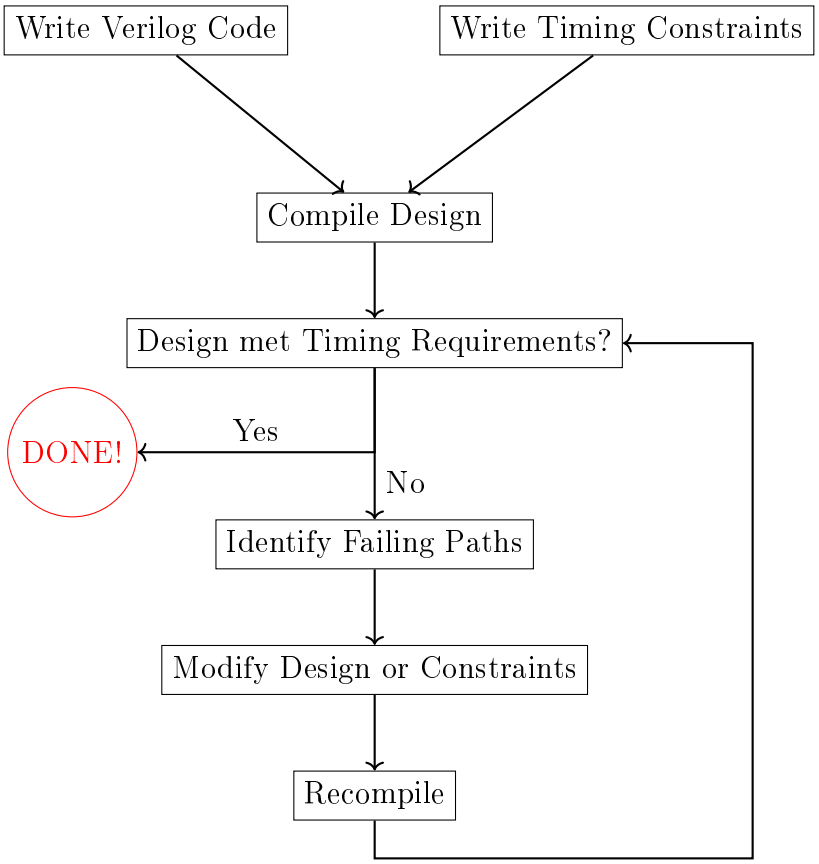

Figure 1.18: Process for achieving timing closure in an FPGA design.

and is now ready to reliably run in hardware. We will discuss each of the steps shown in the diagram individually.

Compile Design

The first step in the timing closure process is to compile the design in Quartus. Quartus performs timing analysis on your design by default after the place and route cycle is complete, so no special options or steps are required here. Just click on "Compile Design" or "Start Compilation" as always.

As a reminder, timing constraints are specified in an SDC file. If there is no SDC file in your project, Quartus assumes a default set of constraints that is likely very inappropriate for your design. *Make sure you have an SDC file in your project!*

Checking if Design Met Timing

After the compilation is complete, you should check whether Quartus was able to successfully meet all of the timing constraints that were set out in your SDC file. The simplest way of doing so is to view the "TimeQuest Timing Analyzer" section of the compilation report that pops up after the compilation finishes. If the words "TimeQuest Timing Analyzer" are written in black text, congratulations! Your design has achieved timing closure and it is safe to move on to hardware testing.

In the less ideal case that TimeQuest identified potential timing issues with your design, the words "TimeQuest Timing Analyzer" will show up in red, as shown in Figure 1.19. Technically, this means that under certain combinations of process variation (normal manufacturing uncertainties), supply voltage and temperature, Quartus can't guarantee that every register in your design will function correctly. Stated another way, some combination of FPGA manufacturing variations, voltage, and temperature may cause some parts of your design to not work.

Naturally, the conditions in the lab for your hardware test may not match up with those required to cause your design to malfunction, in which case your design is likely to work despite the warning from Quartus. However, it could also be the case that your design will indeed malfunction in hardware. These malfunctions can be intermittent over time and/or temperature, and the "symptoms" may vary greatly from compile to compile. Consequently, such issues tend to be very difficult to track down, and can be the source of much frustration. Many previous students in EE 465 have wasted significant amounts of time in the lab chasing phantom errors unrelated to their verilog code due to timing problems.

To be clear: *if the words "TimeQuest Timing Analyzer" are red in your compilation report, it is STRONGLY recommended that you work to achieve timing closure as discussed in the following sections before performing hardware testing. Failure to do so is likely to result in significant debugging challenges.*

Identifying Failing Paths

The process of identifying signal paths in your design which are failing to meet the timing constraints starts by expanding the "TimeQuest Timing Analyzer" portion of the compilation report. Any subsections of the report which show up in red indicate potential problems and should be investigated in more detail. For example, Figure 1.20 shows an example in which TimeQuest is indicating that two different scenarios (combinations of process, voltage, and temperature) are showing register setup time violations. The designer should investigate all of these issues.

The most common type of timing issue students in EE 465 face are be register setup time

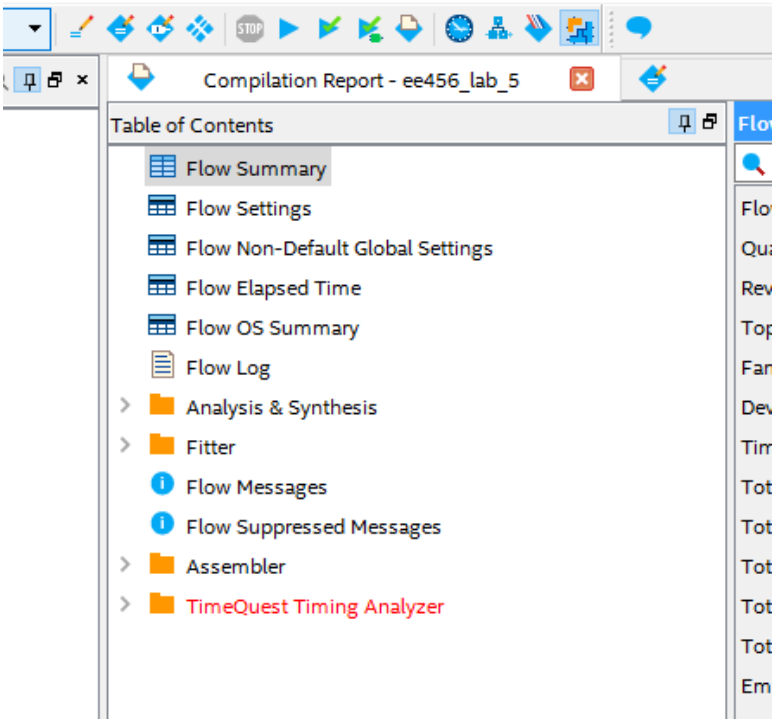

Figure 1.19: Indication of timing failure in Quartus compilation report.

violation issues, so this section will show an example of how to investigate setup issues. First, click on the red "Setup Summary" in the compilation report to see a summary table of the setup analysis results. The summary table should show results for each clock in your design which drives any registers. An example summary table is shown in Figure 1.21. Any clocks that are subject to timing failures are highlighted in red and should be further investigated.

To investigate the timing failures within a particular clock domain, right-click on the clock name (ie. sys_clk in the example shown in Figure 1.21) and choose "Report Timing" from the pop-up menu. A dialog box with a large number of options will appear, and you should notice that the "To clock" field has been set appropriately. The remainder of the options in this "Report Timing" dialog box will be left at their default settings in these instructions - feel free to explore other settings at your leisure.

Once the "Report Timing" command has completed, a series of windows will open, as shown in Figure 1.22. These windows contain a LOT of information and can be intimidating at first glance. In this tutorial we will focus on only the most essential information. As highlighted in Figure 1.22, the "Summary of Paths" window contains one line for each path that is being reported. (In the previous dialog box we chose to report only the 10 worst paths.)

For each path, you should note the name of the source register, the name of the destination register, the name of the source clock, and the name of the destination clock. Note that the destination clock will be the clock we selected earlier. As a sanity check, you should verify that there is a path between the specified registers in your Verilog code. The reported slack

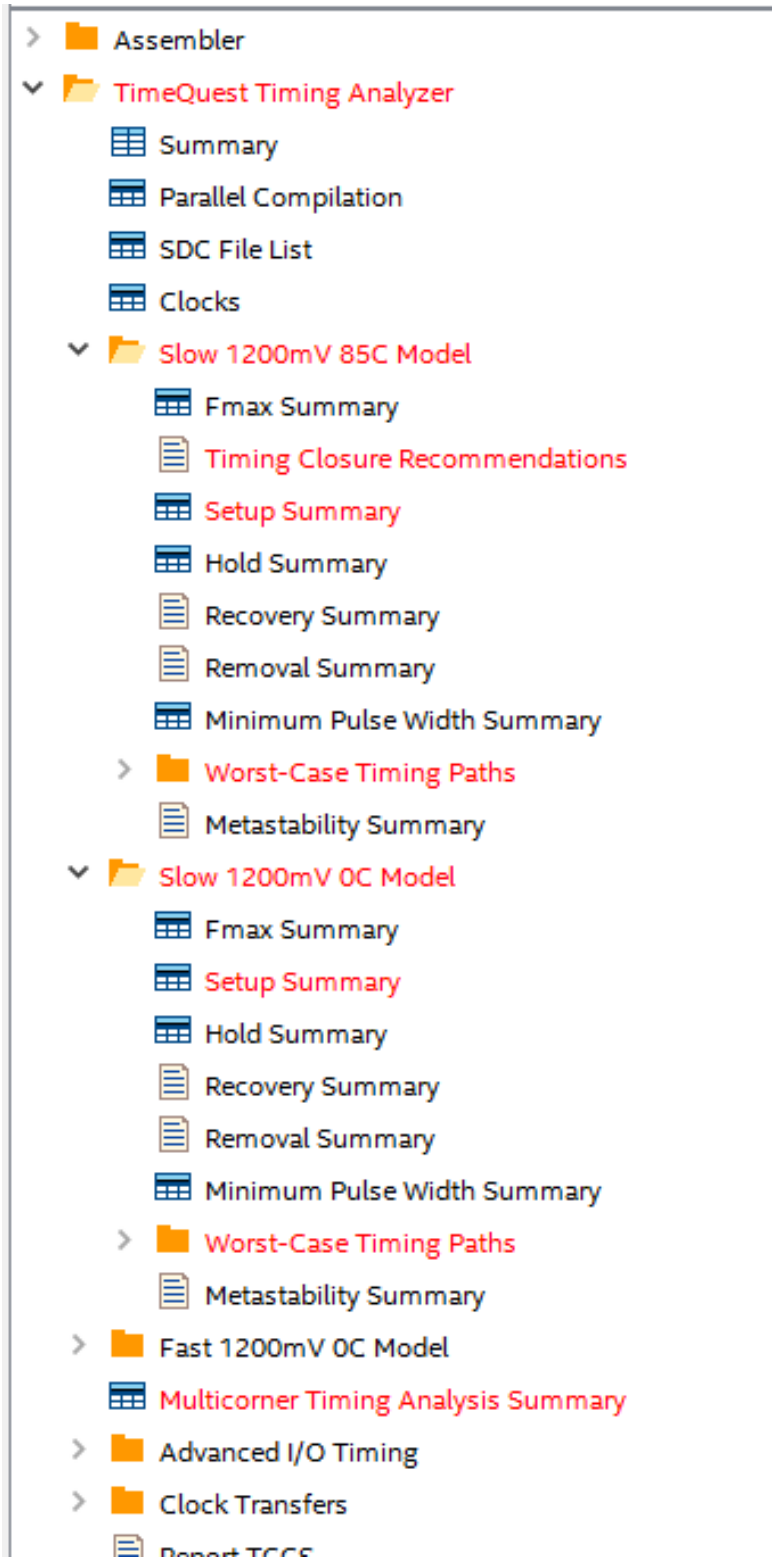

Figure 1.20: Expanded timing analysis report showing setup failures on two different process/voltage/temperature models, as indicated by red text.

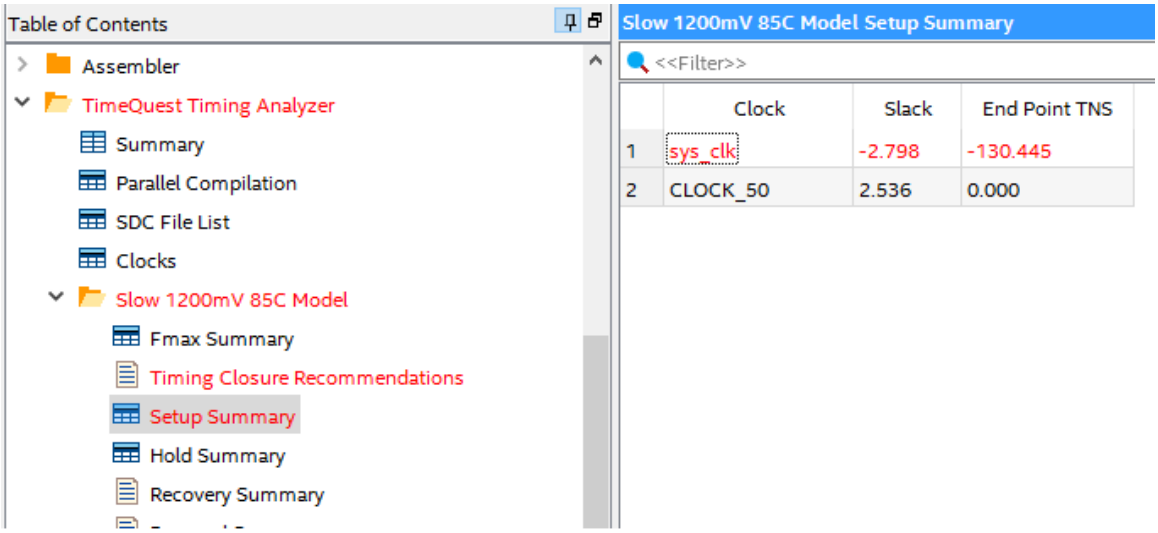

Figure 1.21: Expanded timing analysis report showing setup failures on two different process/voltage/temperature models, as indicated by red text.

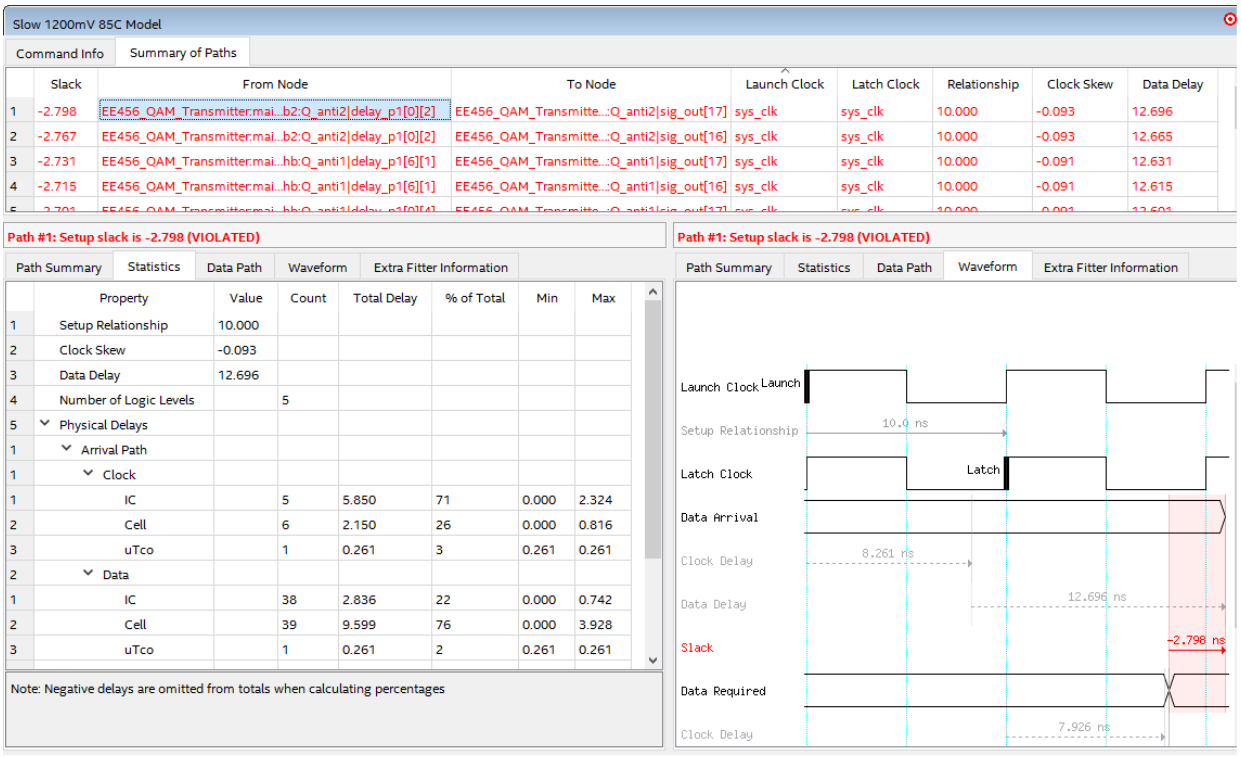

Figure 1.22: Illustration of information provided by TimeQuest GUI regarding critical timing paths.

for a path is the amount of margin the path has with respect to the timing requirement under consideration. A positive slack means that the timing requirement was met and a negative slack indicates that this path failed to meet the timing requirement. Our eventual goal is to eliminate all negative slack values.

To illustrate with an example, the first path shown in Figure 1.22 corresponds to a source register called "delay_p1" and a destination register called "sig_out", both inside a module instance named "Q_anti2". The registers share a common clock frequency of 100MHz, which means that the data should propagate between two registers within 10ns in order to be captured correctly at the destination. This is shown in the "Relationship" column in the table. However, the actual propagation delay between the delay_p1 and sig_out signals in the antialiasing filter module is greater than 12.69ns, as shown in the "Data Delay" column. Therefore, the timing constraint is not being met, as indicated by the reported negative slack value of -2.79ns. The developer would thus conclude that this path is a prime candidate for modification.

Modifying the Design or Constraints

Having identified the critical failing paths in the design, the next step is for the developer to make modifications to the design or its timing constraints to make it easier for Quartus to meet the timing requirements.

As discussed previously, the simplest and most common way to modify a design to facilitate timing closure is to add pipelining between the source and destination registers on the critical paths. Of course, when doing so, it is important to follow the rules discussed earlier in the term to ensure that the functionality of the design is not broken.

Other strategies for modifying a design to improve timing performance may include:

- Writing the code in such a way that the synthesizer is able to place more logic into the optimized "hard IP" blocks (such as block RAMs, DSP blocks, PLLs for clock generation, etc) that are included in the FPGA, rather than using generic FFs and LUTs to implement such functionality. The exact details of how this is done will vary a fair bit from FPGA to FPGA and synthsis tool to synthesis tool, so be sure to check the manufacturer's documentation to see if there are recommended coding structures for your scenario. Note that Quartus provides an extensive list of Verilog templates which synthesize to optimized structures (available from the code editor window.

- Manually duplicating some registers and/or logic in order to reduce the maximum number of fanout points from any any one register. This can enable the placer to locate the source and destination registers in close proximity to one another, thereby minimizing routing delay. Quartus also provides a global "Maximum fanout" setting in the Assignment Editor which can be used to achieve similar results.

- Registering the inputs and outputs to modules. This helps to keep timing issues local to an individual module, which is particularly important when working as part of a team in which modules from multiple developers are combined into a final overall design.

As an alternative to modifying the Verilog code for the design itself, it may be possible to modify the timing constraints instead. In general, the timing constraints in a design are determined by the clocking scheme used. If the wrong timing constraints are selected,

Quartus will be unable to optimize the design correctly, which may cause the design to not function correctly in hardware. If, after a careful review of the failing paths, you believe the timing constraint being applied by Quartus is incorrect, you should consider modifying the constraints defined in your SDC file in order to properly constrain the design. Some examples of the types of scenarios in which this may be a viable option are provided below:

- The timing "Relationship" shown for a path in TimeQuest doesn't match up with the actual period for the source/destination clock. You may wish to double-check the "create_clock" and "create_generated_clock" constraints in your SDC.
- A path is being passed between multiple clock domains and Quartus has incorrectly determined the relationship between the clocks. In addition to checking your clock constraints, you may wish to add multicycle path constraints using the "set_multicycle_path" command in your SDC.
- A signal is quasi-static, which means that it changes only rarely and then remains constant for a long period of time. By default, Quartus doesn't know that such signals are quasi-static, so it analyzes them as though they may change on every clock cycle. Options for constraining this type of signal are the "set_multicycle_path" and the "set_false_path" commands.

As mentioned before, the topic of timing constraint generation can be fairly complex, so the above cases are only scratching the surface of what is possible. Complete reference manuals for TimeQuest and the SDC language will be posted on the class website for reference. Fortunately, it should be possible in EE 465 to get away without many changes to the original basic SDC file. If you are planning to make changes to your timing constraints, it is probably wise to check with the instructor or lab engineer to ensure your changes will function as intended.

Recompiling

After modifying the design and/or its constraints, the final step in the timing closure process is to recompile the design and check if all of the timing constraints are now met. If so, you are done! If not, another round of analysis, modification, and compilation will be necessary.

Unconstrained Timing Paths

As described above, Quartus will report timing issues if any paths in the design are failing the specified timing constraints. Additionally, Quartus will also report errors if there are any paths in the design which are not covered by a timing constraint. This is done to inform the developer that Quartus was unable to analyze the paths in question, in the hope that the developer will update the constraints appropriately in order to constrain all paths.

One of the most common places this issue arises is the input and output ports of the top level module of an FPGA design. If the user provides sufficient information about the data rates and the circuit board design, Quartus has the ability to perform timing analysis on input and output signals in order to check whether the interfaces between the FPGA and any external devices will function correctly. However, if the user doesn't provide such information in the SDC file, Quartus will complain that it was unable to analyze these interface signals by raising warnings of what are known as "unconstrained paths".

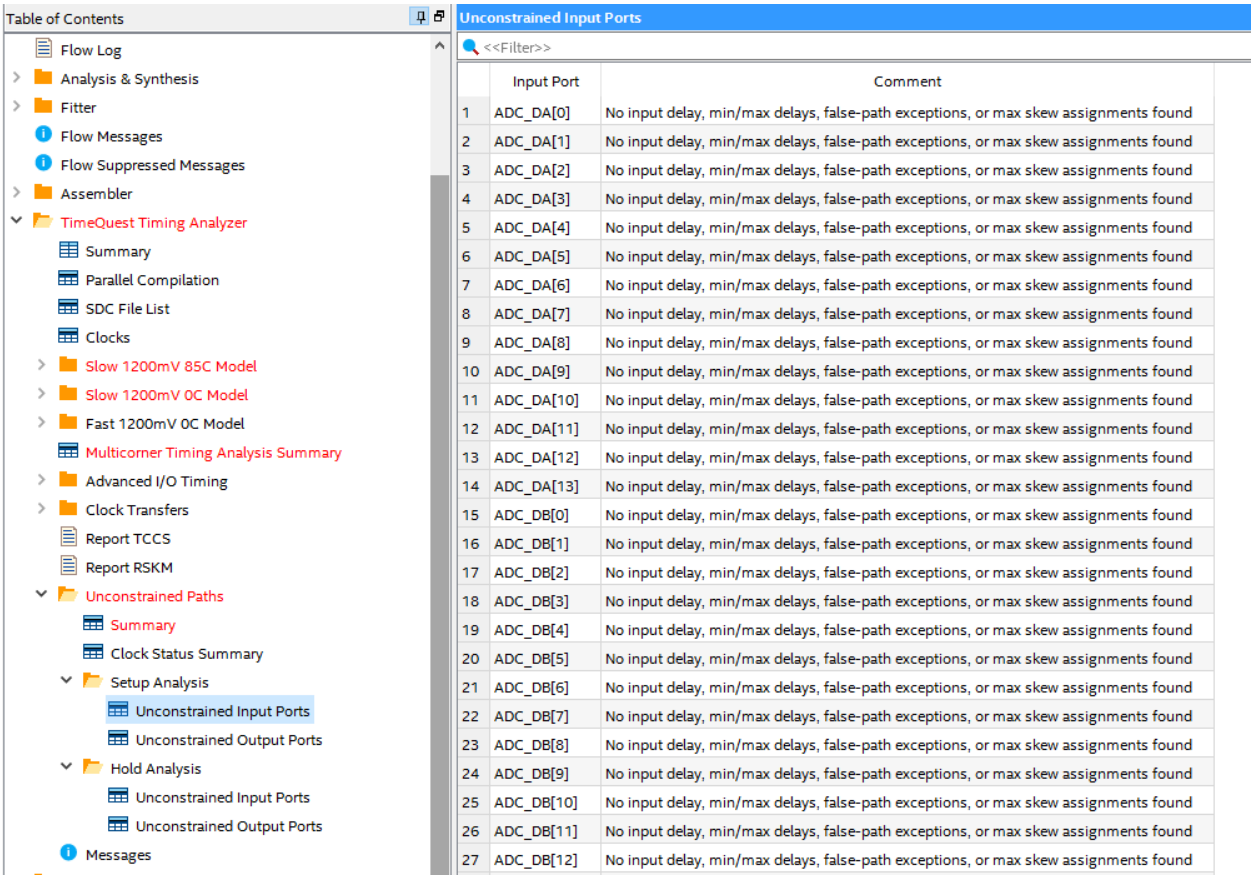

Figure 1.23: Unconstrained paths as reported in TimeQuest report.

It should be noted that unconstrained path warnings can sometimes be safely ignored if the signals are registered just prior to entry to / exit from the FPGA and clock rates are relatively low. However, these warnings can mask real timing errors and condition the user to ignore timing issues. Therefore, it is HIGHLY RECOMMENDED that developers remove any unconstrained path warnings so that the timing error indicator (TimeQuest Timing Analyzer showing up in red text) can function as intended.

To debug any unconstrained paths, browse the "Unconstrained Paths" section of the TimeQuest report, as shown in Figure 1.23. By clicking on the various subsections of the report, it is possible to see which paths are unconstrained. Figure 1.23 focuses on the unconstrained input ports, but a similar process can be used to see other types of unconstrained paths.

In EE 465, the most common place for unconstrained paths to show up is the input and output ports which connect to the ADC and DAC. As mentioned above, Quartus typically wants to analyze these ports to figure out whether the data will be transferred to/from the ADC and DAC correctly. Setting up this analysis correctly requires knowledge of the circuit board design, propagation delays on the circuit board, etc. The importance of this analysis depends greatly on the clock rate for the interface, as higher rate interfaces are much more sensitive to small variations in delays. In a real industrial design, the developer would work

with the PCB designer to ensure that the analysis is performed correctly and thoroughly, by adding "set_input_delay" and "set_output_delay" commands to the SDC file. However, in EE 465, we are dealing with relatively slow interfaces of 25MHz. Provided that the input and output ports are registered immediately upon entering / immediately prior to exiting the FPGA (as was done in the EE 456 MER lab template file which has been posted on the class website), there is a negligible chance of these interfaces failing.

Therefore, for EE 465, it is recommended that students tell Quartus to just ignore the input and output paths and not attempt to analyze them. This will prevent Quartus from raising unwanted warnings which can mask real timing errors. The command which is used to tell Quartus to ignore certain paths is called "set_false_path". Specifically, it is recommended that students add the following commands to their SDC file (note that the names of the ports may need to be changed if they don't match up with the names used in your top level Verilog module).

```
# adding false path constraints to all input ports
set_false_path -from [get_ports ADC_DA*]
set_false_path -from [get_ports ADC_DB*]
set_false_path -from [get_ports KEY*]
set_false_path -from [get_ports SW*]

# adding false path constraints to all output ports
set_false_path -to [get_ports ADC_CLK_*]
set_false_path -to [get_ports DAC_CLK_*]
set_false_path -to [get_ports DAC_DA*]
set_false_path -to [get_ports DAC_DB*]
set_false_path -to [get_ports DAC_WRT*]
set_false_path -to [get_ports LEDG*]
set_false_path -to [get_ports LEDR*]
```

6 Lecture / Discussion Topics for Deliverable 4

Some notes for this section are still in handwritten form and are posted as a separate .pdf document on the class website. The file with the notes is named
`Notes_on_Upsampling_EE465_June_28_2017.pdf`.

Figures 1.24 and 1.25 illustrate the spectrum of the digital signal at various points in the upsampling and downsampling structure that is constructed in Deliverable 4. For convenience, the frequency axis is shown in both analog and digital formats. Note that the

Questions for the class to discuss:

- What should the passband and stopband corner frequencies of the filter be in order to attenuate the images that are created during the upsampling process?
- How does the stopband attenuation of the filter relate to the amount of noise that aliases onto our channel of interest?

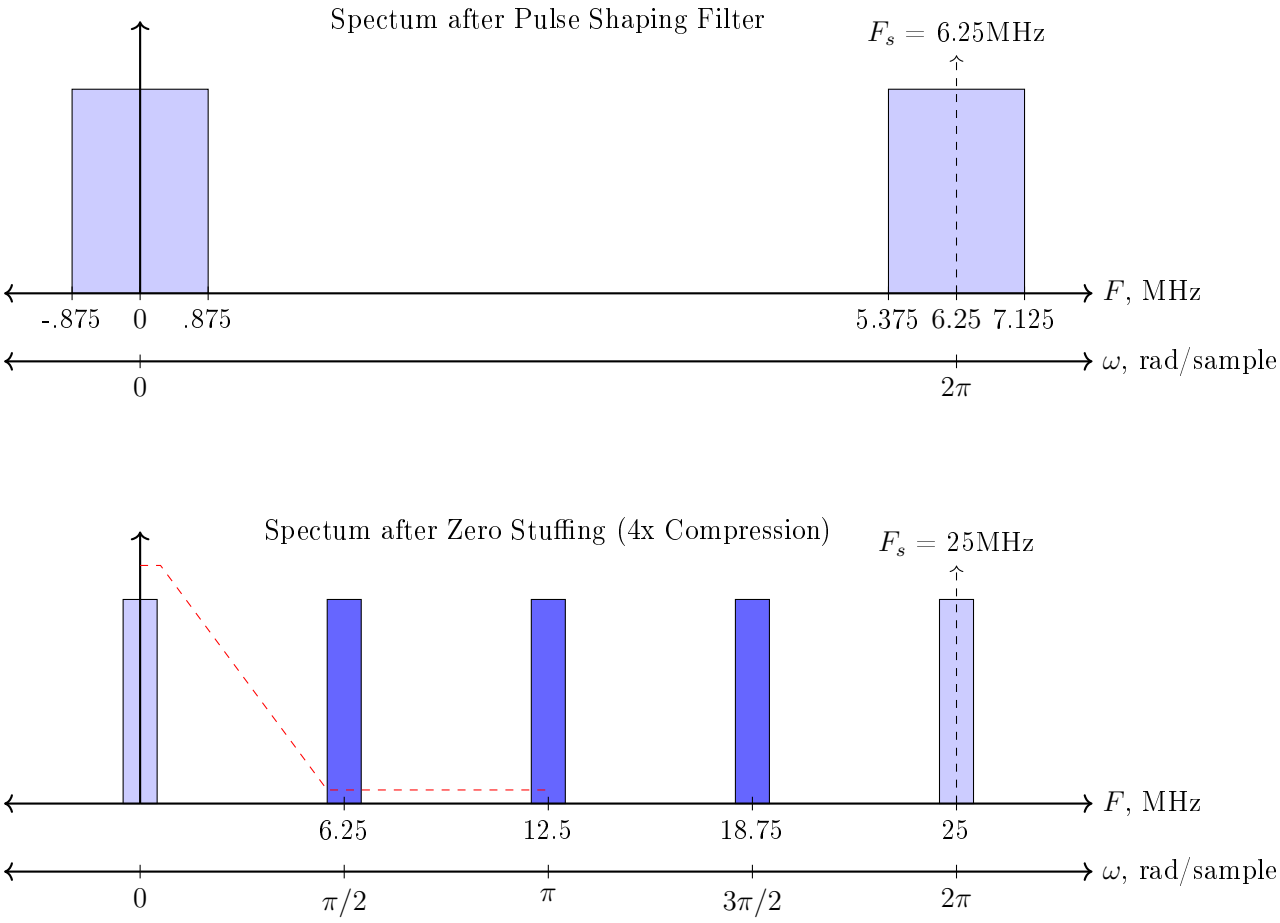

Figure 1.24: Spectrum during upsampling and downsampling process of Deliverable 4 (part 1 of 2).

- How does the upsampling factor influence the amount stopband attenuation required to achieve a particular signal quality?

6.1 Upconversion

Filter design and implementation
Timesharing of multipliers
NCO optimizations

6.2 AWGN and BER

Sources of noise
Techniques for measuring bit errors

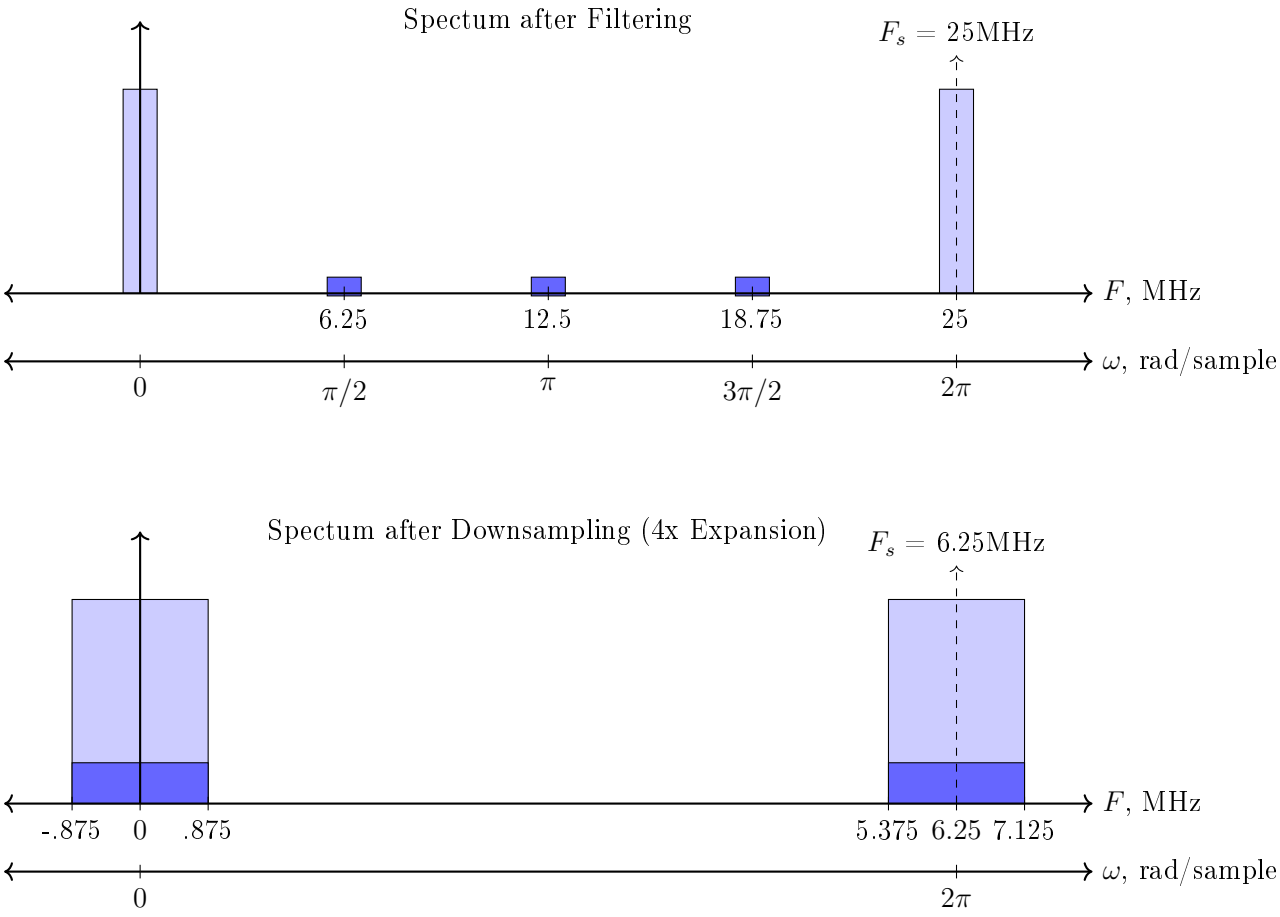

Figure 1.25: Spectrum during upsampling and downsampling process of Deliverable 4 (part 2 of 2).

7 Lecture / Discussion Topics for Deliverable 5

7.1 Timing Recovery

Background

The receiver must synchronize itself to the transmitter before data can be recovered from the received signal. Synchronizing to the transmitter involves "timing recovery" and "frequency and phase recovery". The phrase "timing recovery" means finding the correct time to sample the decision variable. The phrase "frequency and phase recovery" means finding the frequency and phase of the carrier. Knowledge of the frequency and phase of the carrier is used to set the frequency and phase of the two down-conversion local oscillators so that the resulting two baseband signals are the in-phase and quadrature signals that were transmitted.

In burst mode transmission, which is what we have here, a known symbol sequence is

prefixed to the data and used by the receiver to synchronize itself to the transmitter. This sequence of symbols is called the preamble. While each symbol in M-QAM can take on M values, only two of the M values are used in the preamble. The two values used correspond to the the points in the first and third quadrant of the constellation that are largest in magnitude. That is, the two points in the constellation farthest from the origin on the positive-slope diagonal are used. Since only two points in the constellation are used, the preamble symbol sequence can be described by a binary sequence with a "1" indicating the symbol that maps to the constellation point largest in magnitude on the diagonal in quadrant 1 and a "0" indicating the symbol that maps to the constellation point largest in magnitude on the diagonal in quadrant 3.

The preamble symbol sequence that will be used for this class is described by the 27-bit binary sequence

```
preamble = 110111__000_0101_0011_0111__000010,
```

where the right most bit is transmitted first. Note that this is opposite to the convention used in Matlab where the left most sample in a vector is transmitted first. Matlab's rule is very logical: the sample with the lowest vector index is transmitted first.

The preamble has three parts: a 6-bit suffix, a 15-bit unique word and 6-bit prefix, i.e.

```
preamble = suffix__uniqueword__prefix,
```

where

```
suffix = 110111
uniqueword = 000_0101_0011_0111
prefix = 000010
```

Only the unique word portion of the preamble is used for the synchronization.

The unique word is extended with a prefix and suffix to ensure the sharply peaked lobe in the output of the timing recovery filter that marks the time of the decision variable is approximately symmetric in time. The symmetry of this impulse-like lobe is affected by the symbols neighbouring the unique word since the pulse shaping and matched filters spread each symbol over an interval equal to the sum of their lengths. To be more precise, each symbol in the preamble is spread over an interval of $(N_{pulse-shaping} + N_{matched} - 1) \times T_{sample}$ seconds, where $N_{pulse-shaping}$ and $N_{matched}$ are the lengths in samples of the pulse shaping and matched filters and T_{sample} is the time in seconds between samples. The spreading effect diminishes with distance so symbols that the immediate neighbours to the unique word will contribute more to the time interval occupied by the unique word than more distant neighbours. The prefix and suffix, which are the closest and therefore the most influential neighbours, are chosen to make the critical lobe nearly symmetric about its peak. To ensure the critical lobe is completely symmetrical, the prefix and suffix would have occupy neighbours with any influence, which means they would need to be about $(N_{pulse-shaping} + N_{matched} - 1)/samples\_per\_symbol$ symbols long. However, the preamble is overhead and therefore comes with a cost, so only six symbols are used for each of the prefix and the suffix. The prefix and suffix are chosen to be cyclic extensions of the unique word to make the critical lobe nearly symmetrical.

It must be pointed out that the pulse shaping and matched filters were carefully designed so that the ISI is minimal. This means the prefix and suffix will have little affect on the samples that mark the decision variables for the symbols in the unique word, but will affect the samples between the decision variables.

Timing Recovery Theory

It will be shown that the location of a peak of a critical lobe in the output of yet another specially designed filter marks the time of a decision variable. This special filter is referred to as the timing recovery filter. Of course, circuitry must be designed to find the location of the peak and then use this location to adjust the down-sampler to sample the decision variables.

It turns out that the theoretical timing recovery filter can be approximated, with a high degree of accuracy, by an FIR filter that has only 15 non-zero coefficients. Furthermore, the coefficients are either $+1$ or -1, which means the approximated timing recovery filter can be implemented without any multipliers.

The coefficients for the efficient approximated timing recovery filter are a special subset of the coefficients for the theoretical timing recovery filter. To explain how and why the approximated timing recovery filter works requires an understanding of how and why the theoretical timing recovery filter works.

The principle of operation of timing recovery filters is best explained using matched filter theory. In essence this theory as it relates to the "matched" filter in the receiver can be summarized by:

> A filter in a receiver that has an impulse response that is the time reverse of the impulse response of the pulse shaping filter in the transmitter is by definition a matched filter. To be precise it is a filter matched to the pulse shaping filter, but is loosely referred to as the matched filter. A single QAM symbol applied to the pulse shaping filter produces a $\sin(kn)/kn$ type response at the output of the matched filter with the peak being the decision variable.

This means, if it was possible to transmit a single symbol in isolation, the output of the matched filter could be used to mark the location of the decision variable.

Unfortunately, a single symbol can not transmitted in isolation so the output of the matched filter can not be used to find timing. However, a more sophisticated matched filter placed after the matched filter can. This more sophisticated filter is matched to the "matched filter" output produced when the preamble is applied to the pulse shaping filter. However, to save cost it is not matched to the entire output, but rather a segment of the output.

The impulse response for the sophisticated matched filter, which will be referred to as the timing recovery filter, is obtained by applying the preamble to the pulse shaping filter and extracting the 14 symbol interval (actually the interval is 14 symbols plus 1 sample long) in the center of the output of the matched filter. This 14-symbol-plus-1-sample interval contains all 15 decision variables for the 15-symbol unique word. The samples at each end of the interval are the decision variables for the first and last symbol in the unique word. The extracted interval is time reversed to become the impulse response of the timing recovery filter.

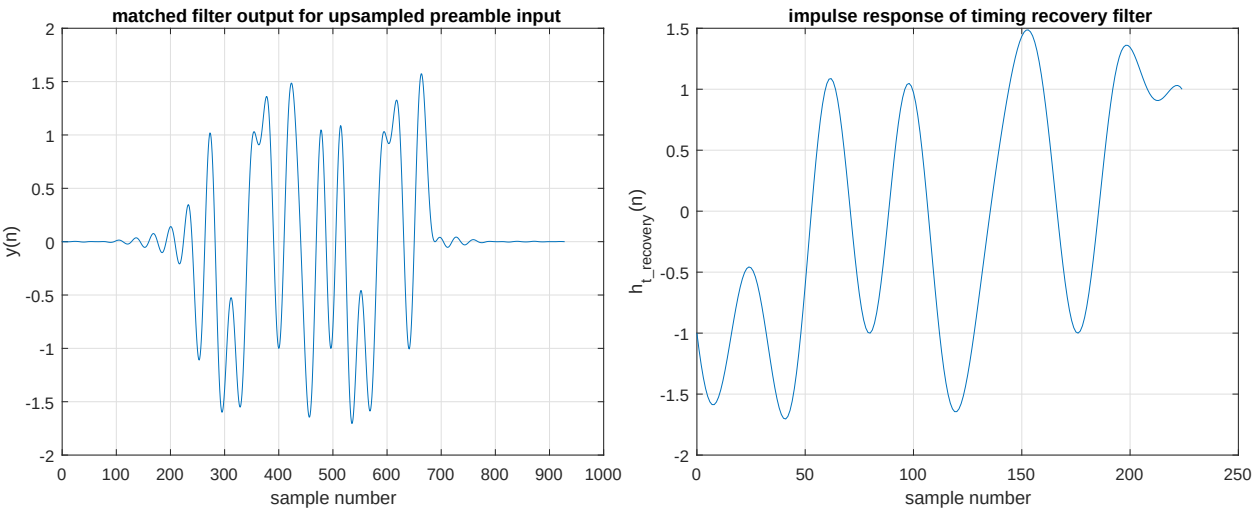

Figure 1.26: Left: Matched filter response to preamble. Right: Center 15-symbol (i.e $14 \times 16 + 1 = 225$ samples) interval flipped to become the impulse response of the unique-word filter.

To improve the resolution of timing detection the output of the matched filter is up-sampled by a factor of 4 prior to filtering by the timing recovery filter. The timing recovery filter must run at the up-sampled rate, which is 16 samples per symbol, and the length of the extracted interval, which is also the length of the timing recovery filter, must be 14 symbols $\times$ 16 samples/symbol + 1 sample = 225 samples. Therefore the timing recovery filter is an FIR filter with 225 coefficients.

The location of the up-sampler and the timing recovery filter within the receiver is illustrated in Figure 2.6 on page 115. The timing recovery filter is in the block labelled "Timing Detector".

The matched filter response to the preamble applied in the transmitter is illustrated on the left side of Figure 1.26[1]. The response is plotted after the matched filter output is up-sampled by a factor of 4, which is equivalent to the response of a matched filter running at 16 samples/symbol. The center 14-symbol-plus-1-sample interval is extracted and flipped to become the impulse response of the timing recovery filter. This response is plotted on the right side of Figure 1.26.

The output of the timing recovery filter is plotted in Figure 1.27[2]. Its length is 1153 samples, which, being the convolution of the matched filter output with the impulse response of the timing recovery filter, is one less than the sum of their respective lengths. The critical lobe is the tall sharp lobe located at its center. The peak of the critical lobe is located at the center sample in the timing recovery filter output.

The location of the peak of the critical lobe is used to synchronize the timing in the receiver to the transmitter. The peak occurs in the output of the timing recovery filter when

[1]The graphs in Figure 1.26 were generated by a Matlab script that is included as subsection 7.4 on page 74.

[2]The graphs in Figure 1.27 were generated by a Matlab script that is included as subsection 7.4 on page 74.

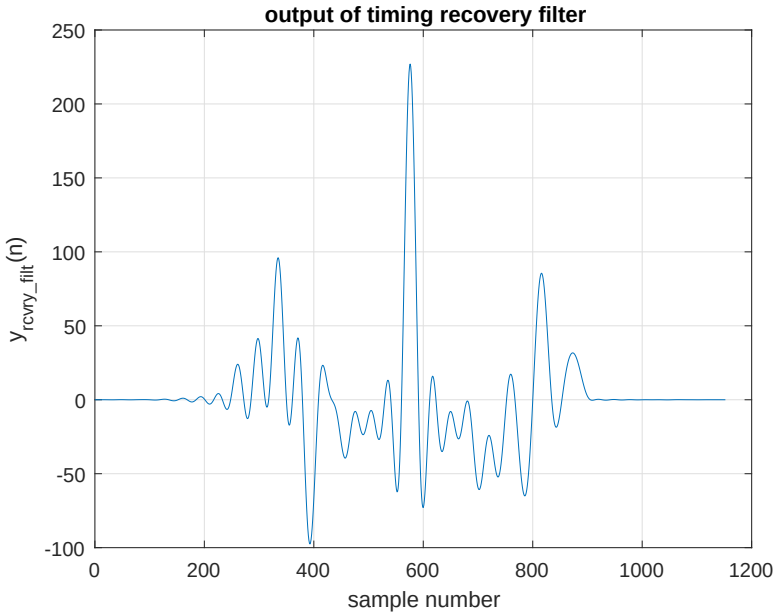

Figure 1.27: Output of the unique-word timing recovery filter in response to the preamble applied in the transmitter.

the 14-symbol-plus-1-sample segment in the center of the matched filter output occupies the registers of the timing recovery filter. This 14-symbol-plus-1-sample segment begins and ends with the decision variables for the first and last symbol in the unique word. Therefore, the samples in the first and last registers of the timing recovery filter at the time the peak occurs are the decision variables for the first and last symbol in the unique word. Therefore, the peak in the output of the timing recovery filter indicates a decision variable has just entered the timing recovery filter and can be used to set the timing in the receiver.

It has now been established that the timing recovery filter can be used to recover timing, but it is a very expensive FIR filter as it has 225 coefficients. To be practical, the number of coefficients must be reduced. It will be shown that the number of coefficients can be reduced to just 15 without significantly degrading the quality of the filter. The rationale and procedure for doing this is presented in the next subsection.

7.2 Practical Timing Recovery Filter

The matched filter has a low-pass frequency response so its output when the preamble is applied to the pulse shaping filter will be a low-pass signal. This is illustrated in the plot on the left side of Figure 1.28 where the magnitude of the Discrete Time Fourier Transform of the matched filter's output is plotted against frequency in cycles/sample.

The plots in all figures in this section were generated by the Matlab script printed in subsection 7.4 on page 74.

The impulse response of the timing recovery filter is a segment of the matched filter output. Therefore, since the matched filter output is a low-pass signal one would expect the timing recovery filter to be a low-pass filter. This is verified by the plot of its magnitude

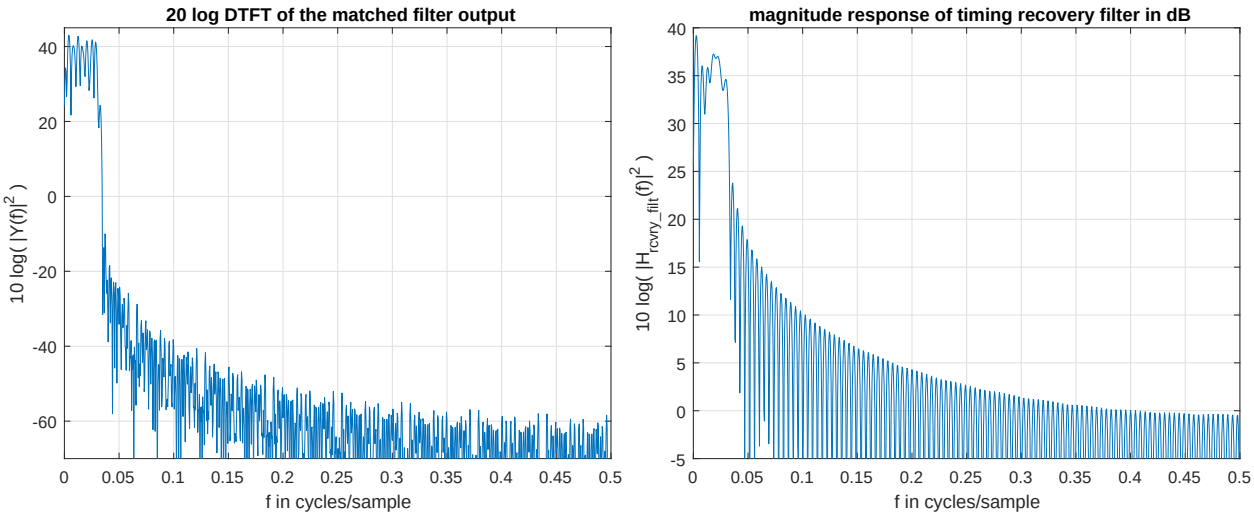

Figure 1.28: Left: Magnitude of the DTFT of matched filter output due to preamble. Right: Frequency response of the timing recover filter.

response given on the right side of Figure 1.28.

To make the timing recovery filter more implementation friendly, several of its coefficients could be disconnected or "zeroed out". Of course, doing this could have an adverse affect on the output. Perhaps surprisingly, "zeroing out" most of the coefficients does not significantly affect the output of the timing recovery filter. For example, suppose 7 out of every 8 coefficients are zeroed out. The impulse response of the "7/8 zeroed out" timing recovery filter is shown on the left side of Figure 1.29. Notice that the number of non-zero coefficients has been reduced from 225 to 29.

The magnitude response of the "7/8 zeroed out" timing recovery filter is shown on the right side of Figure 1.29. As predicted by up-sampling theory, the magnitude response is that of the timing recovery filter with the addition of 7 equally spaced images. Of course zeroing out 7 of 8 coefficient reduces the gain of the primary baseband response, as well as the images, by $10\log(\,(1/8)^2\,) = -18.1\,\text{dB}$. The difference between the timing recovery filter and its "7/8 zeroed out" approximation is the "7/8 zeroed out" approximation will pass energy in the frequencies bands under the images. Should the input signal have some energy in the image bands the output would be distorted. However, in this case the input is a low-pass signal that has very little energy in the image bands. Therefore, the output of the "7/8 zeroed out" timing recovery filter should be nearly the same as that of the timing recovery filter, except scaled in amplitude by 1/8.

The normalized outputs of the timing recovery filter and its "7/8 zeroed out" approximation are plotted in Figure 1.30. Both outputs are normalized to the peak value of their respective outputs. The two outputs are not identical as the input has a small amount of energy in the image bands and the images in the "7/8 zeroed out" approximation have skirts that, although small in magnitude, extend into the original baseband low-pass filter and change its response. Close examination of the two outputs show that the critical lobes are very nearly same.

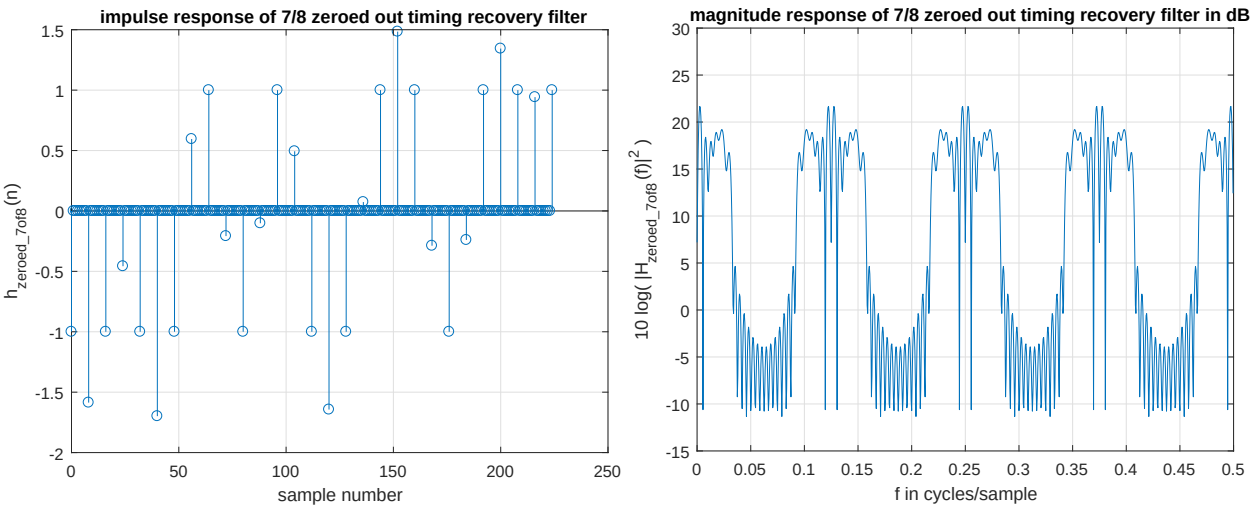

Figure 1.29: Left: Impulse response of the "7/8 zeroed out" timing recovery filter. Right: Magnitude response of the "7/8 zeroed out" timing recover filter.

Certainly it would be safe to use the "7/8 zeroed out" approximation for timing recovery and with only 29 coefficients its cost is reasonable. However, the cost can be further reduced by zeroing out 15 of every 16 coefficients in the timing recovery filter. In doing this the number of images in the "15/16 zeroed out" approximation doubles and slightly overlap each other. The magnitude response of the "15/16 zeroed out" approximation is plotted on the left side of Figure 1.31. The figure clearly shows the images are tightly packed. The best way to see if there is significant overlap is to compare its output to that of the timing recovery filter. The normalized outputs of the two filters are plotted on the right side of Figure 1.31. While there are differences, the critical lobe in the two responses seem to be the same.

Using a "15/16 zeroed out" approximation to the timing recovery filter offers a couple of significant advantages. The obvious advantage is the number of non-zero coefficients is reduced from 29 to 15. The other advantage, which will be demonstrated shortly, is the values of the 15 coefficients are either $+1$ or -1. This allows the filter to be implemented without multipliers as illustrated at the top of Figure 2.7 on page 116. The filter is shown to be a string of 224 registers with a tap on every 16^{th} register. The taps connect directly to either an adder or subtractor depending on whether the tap weight is $+1$ or -1. The output of the filter is labelled `match_strength_I`. The impulse response of this "15/16 zeroed out" approximation is plotted in Figure 1.32.

It is mentioned out of interest that the coefficients of the "15/16 zeroed out" filter are actually the decision variables for the symbols in the unique word.

In conclusion, since the input to the timing recovery filter is a low-pass signal, the "15/16 zeroed out" approximation to the timing recovery filter has very nearly the same time response on and around the critical lobe. Since the "15/16 zeroed out" approximation has only 15 non-zero coefficients and those coefficients are either $+1$ or -1 it is a compact filter that can be built without multipliers. For this reason the "15/16 zeroed out" approximation will be used for timing recovery and from this point forward may be referred to as simply the

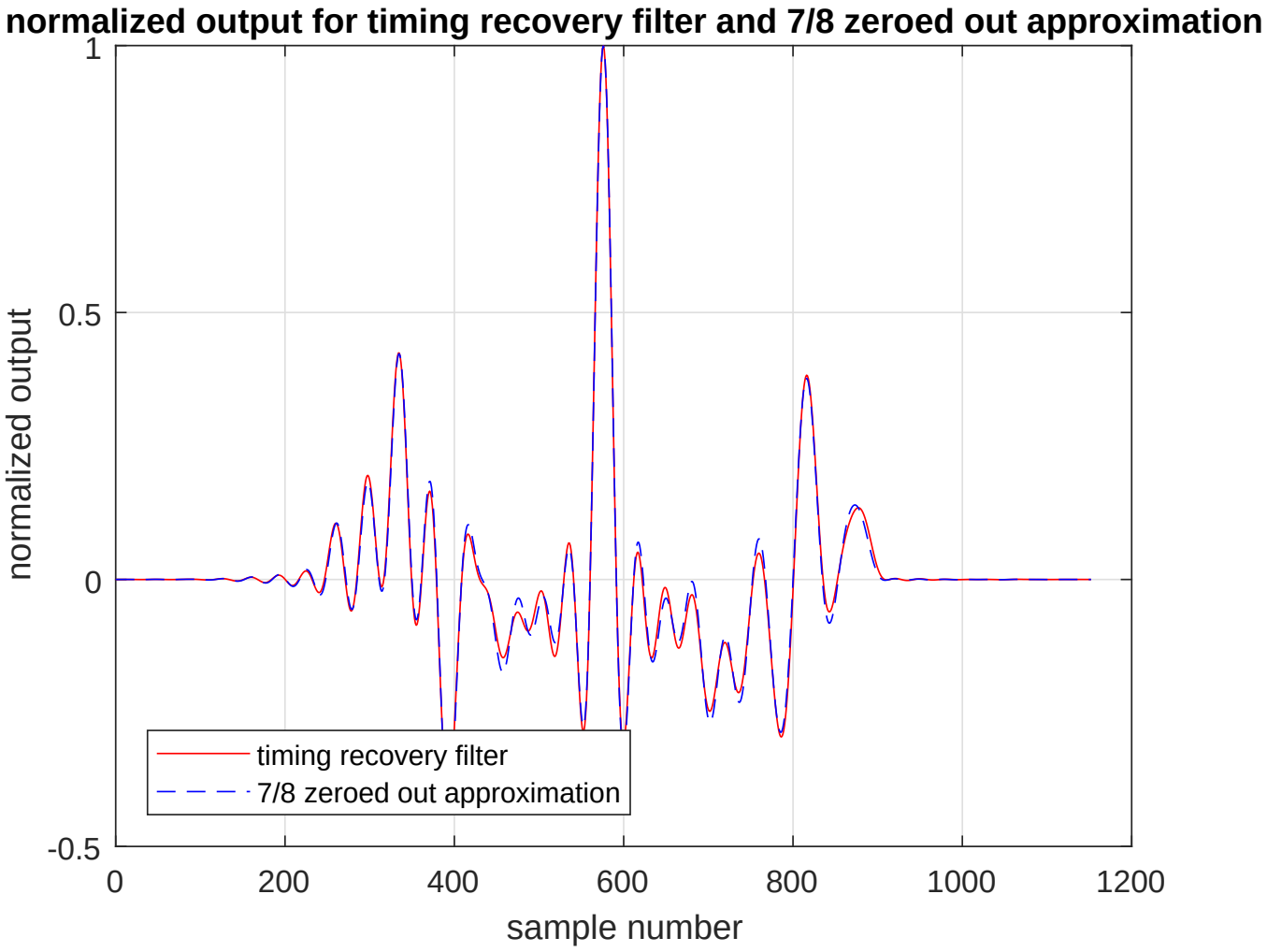

Figure 1.30: The outputs of the timing recovery filter (solid red) and its 7/8 zeroed out approximation (dashed blue) normalized to their respective peak values

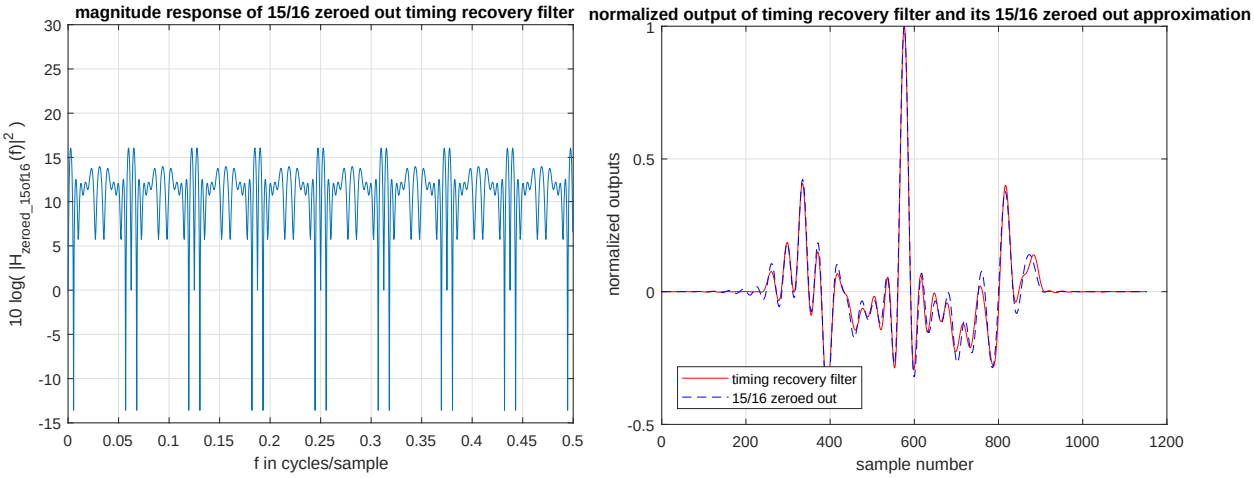

Figure 1.31: Left: Magnitude response of the "15/16 zeroed out" approximation to the timing recovery filter.
Right: The outputs of the timing recovery filter (solid red) and its 15/16 zeroed out approximation (dashed blue) normalized to their respective peak values.

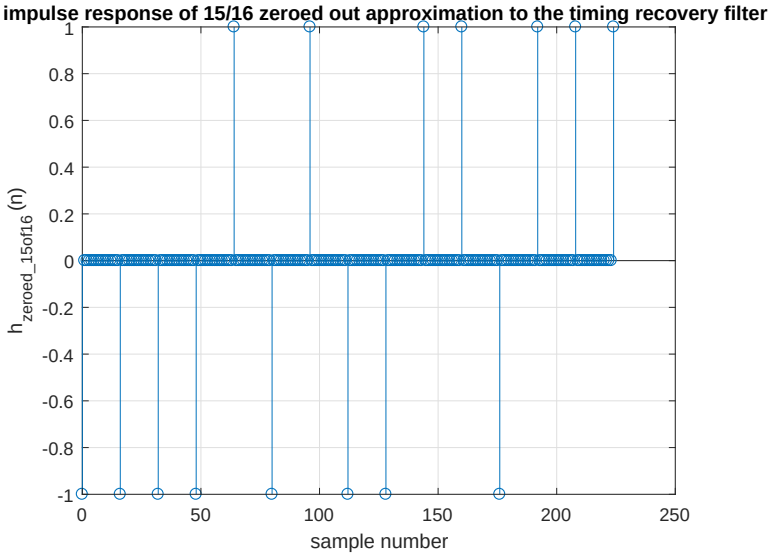

Figure 1.32: The impulse response of a 15/16 zeroed out approximation to the timing recovery filter

timing recovery filter.

7.3 Finding the location of the peak in the critical lobe

Recovering timing from the output of the "15/16 zeroed out" approximation to the timing recovery filter can be separated into to problems: (1) finding the critical lobe and (2) finding the location of the peak in that lobe. Schematic diagrams of circuits that solve each of the problems can be found inside Figure 2.7 on page 116. For purposes of explanations operation of the timing recovery circuitry assume neither the Q-channel matched filter nor the Q-channel timing recovery filter is not implemented and all signals in Figure 2.7 that have a name that ends in Q are 0.

The circuit that identifies the critical lobe uses the sign bits of the taps in the filter as input. The output of the circuit is labelled `match_I` and is a logic 1 when the filter output is some point on the critical lobe.

The peak detector circuit in Figure 2.7 has output labelled `local_peak`. The circuit monitors three successive samples as they are shifted through a 3-bit shift register and makes `local_peak` a logic 1, i.e. high, to indicate a local peak has been found when the center sample in the shift register is greater than its neighbours. The operation of the peak detector is obvious.

The circuit that identifies the critical lobe is more complicated and its principle of operation needs to be explained. First the Boolean logic of the circuit is explained and then it is shown that such a circuit identifies the critical lobe. The task of the circuit is to assert a logic signal, i.e. make a binary signal high, when the signs of the numbers in the 15 registers that have taps agree with the signs of the 15 symbols in the unique word. The circuit consists of 15 xnor gates followed by a 15 input `and` gate. A schematic of the circuit is shown inside the schematic of the broader circuit shown in Figure 2.7. The inputs to the circuit are the inputs to the 15 xnor gates. These inputs are the sign bits of the registers in the filter that have taps. These sign bits are compared to the signs of the set of "+1" or "−1" symbols that make up the unique word, which of course are known *a priori* and are therefore constants. The outputs of the xnor gates are `anded` with a 15 input `and` gate to produce the output labelled `match_I` in Figure 2.7. `match_I` is asserted, i.e. made high or a logic 1, when the signs of the numbers in the 15 taps agree with the signs of the 15 symbols in the unique word.

The symbol sequence in the unique word was carefully chosen so that `match_I` is high, at and only at, a time when the filter output is a sample located on the critical lobe in the vicinity of its peak. This fact is illustrated in Figure 1.33, which shows the number of xnor outputs that are high at the time of sample n on the "15/16 zeroed out" filter output. There is only one segment where all 15 xnors are high. That is a 10 sample segment from sample 572 to 581. This interval contains the peak of the critical lobe which is located at sample 576. There is one sample where 14 xnors are high, which is dangerously close to having all 15 xnor gates high, but that at sample number 817, which occurs after the peak in the critical lobe so will never be encountered.

The safety margin in the circuit is better illustrated in Figure 1.34 where both the number of xnors that are high and the filter output are plotted on the same graph. The plot shows there is one peak prior to the peak in the critical lobe that is coincident with an xnor count

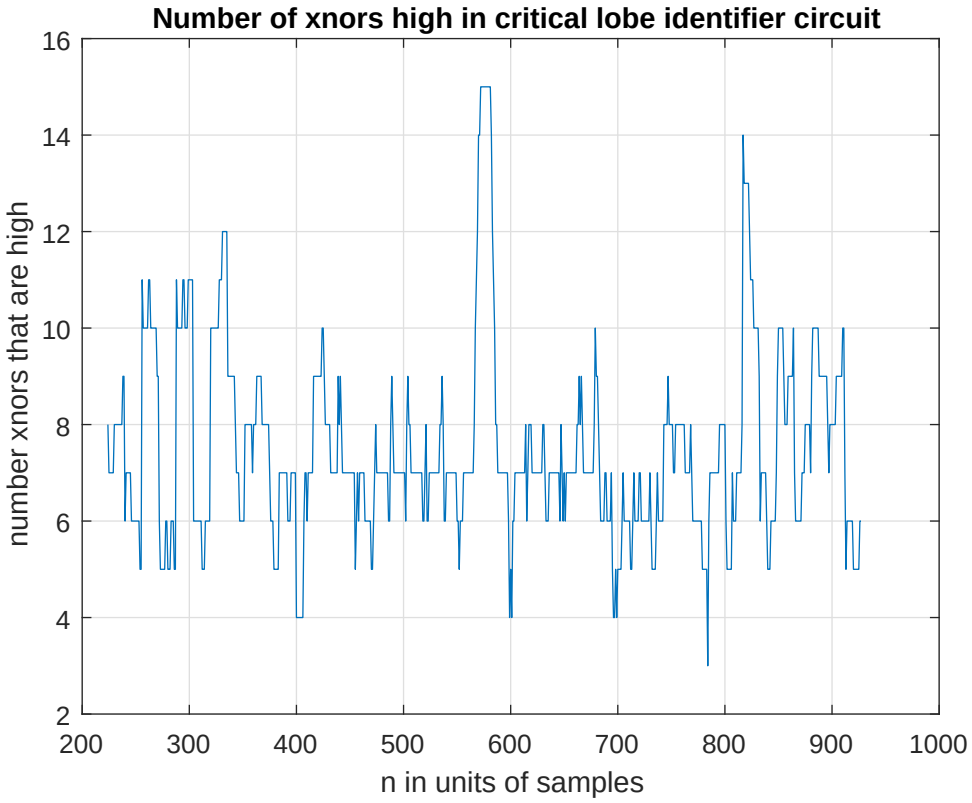

Figure 1.33: The number of xnor gates that are high at sample n of the "15/16 zeroed out" filter output.

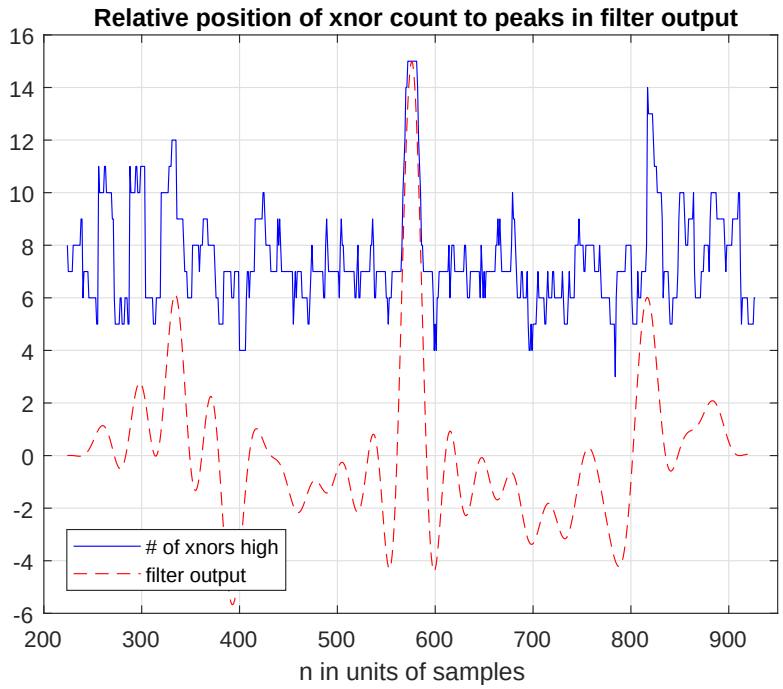

Figure 1.34: The number of xnor gates that are high at sample n of the "15/16 zeroed out" filter output.

of 12. That is at sample number 334. The xnor count at all other peaks prior to the peak in the critical lobe is less than 12. This provides a safety margin of 3. In other words the would have to be sufficient noise to toggle 3 specific xnor gates.

There are ways to increase this margin. For example, if the output of the peak detector circuit was delayed by 2 samples, then the peak detector output would not be active at the peak at sample 334 where the xnor count is 12. The peak detector would be active at sample 336 where the xnor count is 11. This would increase the safety margin by 1 making it 4.

In fact the peak detector circuit in Figure 2.7 does have a delay of two so the safety margin of the circuit is actually 4. Of course this two sample delay must be taken into account when calculating the time of occurrence of the decision variables.

The output of the timing detection circuit is a pulse that marks the location of the peak in the critical lobe, more precisely it marks the time of the second sample after the occurrence of the peak. The peak detection pulse signal is labelled `peak_detection_pulse` in Figure 2.7.

The `peak_detection_pulse` signal is the output of a 3-input `and` gate. One of the inputs is an enable that is set under the control of the MAC and cleared by `peak_detection_pulse` one sample after it is active. The enable signal together with the setting and clearing action are necessary to enable the peak detection just prior to the arrival of the preamble and to disable it right after the peak is detected so that the data that follows the preamble will not generate a false timing signal.

The other two inputs to the 3-input `and` are `local_peak` and `match_I`, which are logic 1's two samples after a local peak in the output of the "15/16 zeroed out" timing recovery filter

occurs and the inputs to the 15 xnors match the 15 symbols in the unique word, respectively. Both are high at the same time if and only if the sample at the output of the "15/16 zeroed out" filter is two samples after the peak of the critical lobe. When both are inputs active and the enable input is active the output of the 3-input and gate goes high to indicate "set the timing in the receiver now". The output going high clears the enable flip/flop that was set by the MAC on the next clock edge thereby disabling the 3-input and limiting the output of the 3-input `and` gate, i.e. the signal called `peak_detection_pulse`, to a pulse that is one sample time in width.

7.4 Matlab script for plots in timing recovery notes

The Matlab script that was used to generate the plot for the notes on timing recovery is given below;

```
%
% last modified June 25, 2017
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Graphs is support of the EE465 notes on timing recovery
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% parameters to be specified
prefix = [-1 -1 -1 -1  1 -1];
unique_word = [ -1 -1 -1 -1 1 -1 1 -1 -1 1 1  -1 1 1 1];
suffix = [1 1 -1 1 1 1];
preamble = [suffix, unique_word, prefix]; % NB: The EE465 student guide
                            % specifies the right most symbol in the
                            % preamble is to be transmitted first.
                            % NB:   Matlab uses the opposite convention.
                            % The left most number in a Matlab vector has
                            % the lowest sample number and therefore
                            % is transmitted first.

ROF = 0.12;    % roll off factor for raised cosine filter
span = 32;     % length of raised cosine filter in symbols
SPS = 16;      % number of samples per symbol

f = 0.5*[0:1:999]/1000; % frequency vector in cycles/sample

%end parameters
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% derived parameters
N_preamble = length(preamble);
N_uw = length(unique_word);
N_rcvry_filt = (N_uw-1)*SPS + 1; % number of sample between the first
                                 % and last symbol in the unique word
                                 % including the first and last symbol
n_rcvry_filt = [0:N_rcvry_filt-1];
N_filter = SPS*span + 1;  % number of coefficients in the filter
N_filter_input = SPS*(N_preamble-1) + 1; % length of input
                             % after zero stuffing
                             % note no zeros place before first or after
                             % the element in the preamble
N_filter_output = N_filter + N_filter_input - 1; % length of matched
                                                 % filter output
n_out = [0:N_filter_output-1];  % sample index vector for matched filter
                                %output
N_rcvry_filt_out = N_rcvry_filt + N_filter_output -1; % length of the
                                % output of the timing recovery filter
n_rcvry_filt_out = [0:N_rcvry_filt_out-1];

% find the filter input, which is zero stuffed preamble
filter_input = zeros(1,N_filter_input); % intial input with zeros
filter_input(1:SPS:end) = fliplr(preamble); % place non-zero values
                                 % flip the preamble so that the order
                                 % of transmission has first symbol
                                 % to be transmitted on the left
% end derived parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% find filter coefficients for both new and old versions of matlab

% for newer versions of Matlab use rcosdesign as follows:
% b = rcosdesign(ROF,span,SPS,'normal'); % coeficients for RC filter

% for older versions of matlab compute by hand as follows:
n_b = [1:N_filter] - (N_filter +1)/2; % sample index vector for filter
b = sinc(n_b/SPS) .* cos(pi*ROF/SPS*n_b) ./ (1 - (2*ROF*n_b/SPS).^2);

% b = b / sum(b); % make the DC gain 1, i.e. 0 dB
b = b / max(b);  % make the peak value of the impulse response 1
```

```
H_f = b * exp(-i*2*pi*n_b.'*f); % frequency response of raised cosine
                                % filter for frequencies in vector f


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% find the filter output
y_n = conv(filter_input,b); % output of the matched filter

Y_f = y_n * exp(-i*2*pi*n_out.'*f); % DTFT of the matched filter output
                                    % at frequencies in the vector f


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% extract the samples in the output that are inside the unique word
index_of_first_sample = (N_filter_output - N_rcvry_filt)/2+1;
index_of_last_sample = index_of_first_sample + N_rcvry_filt -1;
n_extraction = [index_of_first_sample:1: index_of_last_sample];
h_rcvry_filt = fliplr(y_n(n_extraction));
H_rcvry_filt = h_rcvry_filt * exp(-i*2*pi*n_rcvry_filt.'*f);
                          % frequency response of timing recovery filter
                          % for frequencies in the vector f

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% output of timing recovery filter
y_rcvry_filt = conv(h_rcvry_filt,y_n);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% construct more efficient timing recovery filters by zeroing out
% 7 of every 8 coeficients
h_zeroed_7of8 = zeros(1,N_rcvry_filt); % start with zeros
h_zeroed_7of8(1:8:end) = h_rcvry_filt(1:8:end); % zero out 7 of every
                                                % 8 samples
H_zeroed_7of8 = h_zeroed_7of8 * exp(-i*2*pi*n_rcvry_filt.'*f);
                          % frequency response of timing recovery filter
                          % with 7 of every 8 coefficients zeroed out
                          % for frequencies in the vector f

% find the ouput of the more efficient timing recover filter
y_zeroed_7of8 = conv(h_zeroed_7of8,y_n); % output of more efficient timing
                                         % recovery filter

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% construct most efficient timing recovery filters by zeroing out
% 15 of every 16 coeficients
```

```
h_zeroed_15of16 = zeros(1,N_rcvry_filt); % start with zeros
h_zeroed_15of16(1:16:end) = h_rcvry_filt(1:16:end); % zero out 15 of every 16 samples
H_zeroed_15of16 = h_zeroed_15of16 * exp(-i*2*pi*n_rcvry_filt.'*f);
                         % frequency response of timing recovery filter
                         % with 15 of every 16 coefficients zeroed out
                         % for frequencies in the vector f

 % find the ouput of the more efficient timing recover filter
 y_zeroed_15of16 = conv(h_zeroed_15of16,y_n); % output of more efficient timing
                                         % recovery filter

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % BACKGROUND
 % The circuit that indicates the output of the timing recover
 % circuit is a sample on the critical lobe consists of 15 xnors
 % that compare the sign bits of the 15 taps in the 15 of 16 zeroed
 % out filter to the unique word followed by a 15 input and gate.
 % The outputs of the xnors are anded by the 15 input and gate and
 % if all 15 xnor outputs are high, the output of the and gate is
 % high indicated the  output of the 15 of 16 timining recovery filter
 % is a sample that is located on the critical lobe.
 %
 % To see how much margin there is in this cicuit the number of
 % xnor outputs that are positive counted and that number is plotted
 % for each sample. If for some sample the number plotted is 14,
 % then for that sample 14 of the 15 xnor outputs were high. Ideally
 % one would like the number to be 15 for samples on the main lobe
 % and 0 for all other samples
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % Count the number of xnor outputs that are high

 % First discrimate the output on the basis of its 2's complement sign bit
 % make a sample +1 if the sign bit is zero and -1 is the sign bit is 1
discriminated_output = sign(sign(y_n)+.5); % 1 if y_n is 0 or positive
                                          % -1 if y_n is negative
 % find the difference between the number of xnor outputs that are
 % high and the number that are low
 % This done in a indirect way using convolution. The problems is the
 % results are only valid for the duration of the shifting in the
 % convolution when the filter registers filled with data
low_from_high = conv(h_zeroed_15of16,discriminated_output); % valid
                               % from sample 225 to sample end-224
                               % for reasons given above
```

```
% adjust the wrong_from_right to the number that are high
number_of_xnors_high = (low_from_high + 15)/2;
% end count number of xnors that are high
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% plot the figures
figure(1); plot(f,20*log10(abs(H_f))); grid;
   xlabel('f in cycles/sample');
   ylabel('10 log( |H(f)|^2 )');
   title('magnitude response of raised cosine filter');
   axis([0, 0.5, -85, 25]);
figure(2); plot(n_out,y_n); grid;
   xlabel('sample number');
   ylabel('y(n)');
   title('matched filter output for upsampled preamble input');
   print -depsc timing_recovery_1a.eps
figure(3); stem([0:(N_filter_output-1)/SPS],y_n(1:SPS:end)); grid;
   xlabel('n_{symbol} i.e. decision variable number');
   ylabel('y(n_{symbol}*SPS)');
   title('downsampled output of the matched filter');
figure(4); plot(f,20*log10(abs(Y_f))); grid;
   xlabel('f in cycles/sample');
   ylabel('10 log( |Y(f)|^2 )');
   title('20 log DTFT of the matched filter output');
   axis([0, 0.5, -70, 45]);
   print -depsc timing_recovery_3a.eps
figure(5); plot(n_rcvry_filt,h_rcvry_filt); grid;
   xlabel('sample number');
   ylabel('h_{t\_recovery}(n)');
   title('impulse response of timing recovery filter');
   print -depsc timing_recovery_1b.eps
figure(6); stem([0:(N_rcvry_filt-1)/SPS],h_rcvry_filt(1:SPS:end)); grid;
           ylabel('h_{t\_recovery}(n_{decision\_variable}*SPS)');
           xlabel('n_{decision\_variable}, i.e. indices for decision variables')
           title('downsampled impulse response of timiming recovery filter')
figure(7); plot(f, 20*log10(abs(H_rcvry_filt))); grid;
   xlabel('f in cycles/sample');
   ylabel('10 log( |H_{rcvry\_filt}(f)|^2 )');
   title('magnitude response of timing recovery filter in dB');
   axis([0, 0.5, -5, 40]);
   print -depsc timing_recovery_3b.eps
figure(8); plot(n_rcvry_filt_out,y_rcvry_filt); grid;
   xlabel('sample number');
   ylabel('y_{rcvry\_filt}(n)');
```

```
   title('output of timing recovery filter');
   print -depsc timing_recovery_2.eps
figure(9); stem(n_rcvry_filt,h_zeroed_7of8); grid;
   xlabel('sample number');
   ylabel('h_{zeroed\_7of8}(n)');
   title('impulse response of 7/8 zeroed out timing recovery filter');
   print -depsc timing_recovery_4a.eps
figure(10); plot(f, 20*log10(abs(H_zeroed_7of8))); grid;
   xlabel('f in cycles/sample');
   ylabel('10 log( |H_{zeroed\_7of8}(f)|^2 )');
   title(['magnitude response of 7/8 zeroed out timing ', ...
                 'recovery filter in dB']);
   axis([0, 0.5, -15, 30]);
   print -depsc timing_recovery_4b.eps
figure(11); plot(n_rcvry_filt_out,y_zeroed_7of8); grid;
   xlabel('sample number');
   ylabel('y_{zeroed\_7of8}(n)');
   title('output of efficient 7-of-8 zeroed timing recovery filter');
figure(12); plot(n_rcvry_filt_out,y_zeroed_7of8/max(abs(y_zeroed_7of8)), ...
                '-r', ...
   n_rcvry_filt_out,y_rcvry_filt/max(abs(y_rcvry_filt)), '--b'); grid;
   xlabel('sample number');
   ylabel('normalized output');
   legend('timing recovery filter', '7/8 zeroed out approximation', ...
              'location','southwest');
   title(['normalized output for timing recovery filter ', ...
               'and 7/8 zeroed out approximation']);
   print -depsc timing_recovery_5.eps
figure(13); stem(n_rcvry_filt,h_zeroed_15of16); grid;
   xlabel('sample number');
   ylabel('h_{zeroed\_15of16}(n)');
   title('impulse response of 15/16 zeroed out approximation to the timing recovery filt
   print -depsc timing_recovery_7.eps
figure(14); plot(f, 20*log10(abs(H_zeroed_15of16))); grid;
    xlabel('f in cycles/sample');
    ylabel('10 log( |H_{zeroed\_15of16}(f)|^2 )');
    title('magnitude response of 15/16 zeroed out timing recovery filter');
    axis([0, 0.5, -15, 30]);
    print -depsc timing_recovery_6a.eps
figure(15); plot(n_rcvry_filt_out,y_zeroed_15of16); grid;
   xlabel('sample number');
   ylabel('y_{zeroed\_15of16}(n)');
   title('output of efficient 15-of-16 zeroed timing recovery filter');
figure(16);
   plot(n_rcvry_filt_out,y_zeroed_15of16/max(abs(y_zeroed_15of16)), ...
```

```
    '-r', n_rcvry_filt_out,y_rcvry_filt/max(abs(y_rcvry_filt)), '--b');
   grid;
   xlabel('sample number');
   ylabel('normalized outputs');
   legend('timing recovery filter', '15/16 zeroed out ', ...
           'approximation',  'location','southwest');
   title(['normalized output of timing recovery filter ', ...
          'and its 15/16 zeroed out approximation']);
   print -depsc timing_recovery_6b.eps
figure(17); % window output to remove indeterminate portion of
            % convolution when filter registers are not filled with data
   plot(n_rcvry_filt_out(225:end-225), ...
                number_of_xnors_high(225:end-225)); grid;
   xlabel('n in units of samples');
   ylabel('number xnors that are high');
   title('Number of xnors high in critical lobe identifier circuit');
   print -depsc timing_recovery_8.eps
figure(18); % window output to remove indeterminate portion of
            % convolution when filter registers are not filled with data
   plot(n_rcvry_filt_out(225:end-225), ...
                number_of_xnors_high(225:end-225),'-b', ...
                n_rcvry_filt_out(225:end-225), ...
                y_zeroed_15of16(225:end-225),'--r');
   grid; axis([200,950,-6,16]);
   legend('# of xnors high', 'filter output','location','southwest');
   xlabel('n in units of samples');
   title('Relative position of xnor count to peaks in filter output');
   print -depsc timing_recovery_9.eps
```

8 Lecture / Discussion Topics for Deliverable 6

In EE 465, we have constructed an example communication system which has both the transmitter and receiver inside an FPGA development board. In such an ideal setting, the operation of the system is governed by simple and predictable rules and formulas. However, in a practical communication system, the transmitter and receiver are separate devices built out of distinct electronic components. Due to manufacturing variability, component aging, and environmental factors such as temperature variation and/or vibration, it is common for the actual behaviour of the circuits to deviate slightly from the designed specifications.

One practical example of this phenomenon may be observed in the oscillators which generate clock signals in the transmitter and receiver. Due to the factors listed above, it is exceedingly unlikely that the oscillation frequencies of two DE2 boards will be EXACTLY the same, even though they are each based off of crystal oscillators with the same nominal frequency (50MHz in this case). If we were to transmit a signal from one DE2 board to a receiver in a second DE2 board, this frequency difference could impair the receiver's ability

to recover the original data.

Deliverable 6 builds upon the receiver design from Deliverable 5 in order to make the system robust against error in the carrier frequency which arises from such variations in oscillator frequencies.

Please note that these lecture notes for Deliverable 6 are purposely made much more detailed than those of the prior deliverables. This is done because in certain years there may be limited lecture time available for Deliverable 6. By deviating from the model of student-driven discussions, it is hoped that the necessary background information for Deliverable 6 can be conveyed in an efficient manner through independent study.

Impact of Phase and Frequency Error

Consider the communication system shown in Figure 1.35, in which the oscillator driving the transmitter circuit is slightly inaccurate, resulting in a system clock rate of $F_{s,t} = 25\text{MHz} + \Delta F$, instead of the ideal value of 25MHz. In this example, the receiver clock frequency is assumed to be perfect, i.e. $F_{s,r} = 25\text{MHz}$.

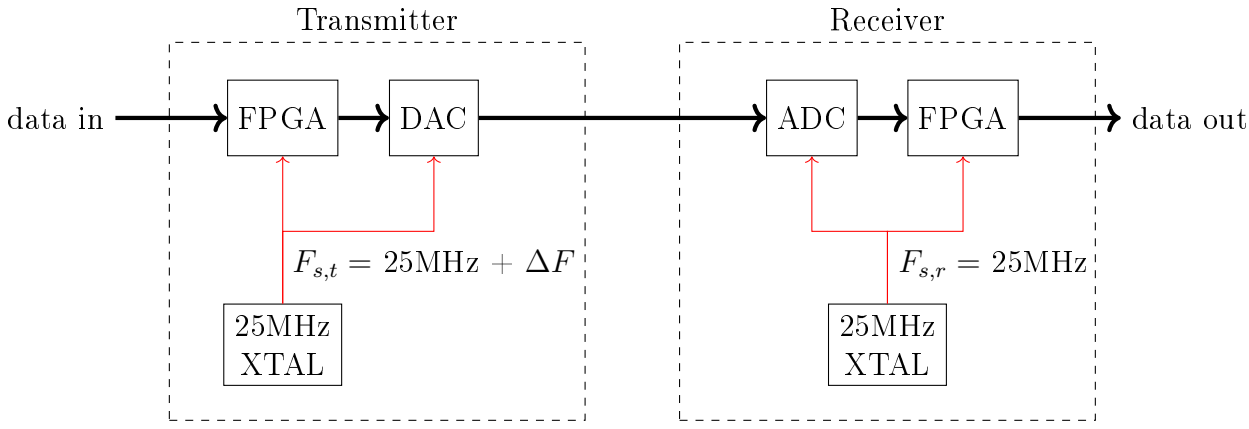

Figure 1.35: A communication system with an inaccurate transmitter oscillator.

Note that the frequency accuracy of a crystal oscillator is typically quoted by the manufacturer in a "parts per million" (ppm) specification. The specified ppm value may be viewed as a ratio which indicates how much the actual frequency of the oscillator may deviate from the nominal (ideal) value. The ratio is calculated as follows:

$$\text{ppm specification} = \frac{|\Delta F|}{\text{nominal frequency}} * 1e6 \tag{1.6}$$

For example, the actual frequency of oscillation of a 25MHz crystal which is specified at 100ppm could be any value in the range (25MHz - 2.5kHz) $\rightarrow$ (25MHz + 2.5kHz). Furthermore, the frequency of a particular oscillator could vary within this range over time and temperature. When designing the receiver in a communication system, it is important to understand the implications of this frequency error.

To illustrate the effects of frequency error, the following discussion considers the specific case of the transmitter constructed in EE 465, in which a baseband QAM signal is digitally upconverted inside an FPGA which runs at a nominal system clock rate of 25MHz. The resulting RF signal is sent to a DAC which runs at the same nominal rate of 25MHz.

Inside the transmitter FPGA, the frequency error has no impact on the values of the RF samples which are generated. The carrier frequency in the digital domain was selected in Deliverable 4 to be $\omega_c = 2\pi * \frac{6.25\text{MHz}}{25\text{MHz}} = \pi/2$ radians / sample. This digital carrier frequency is passed as a fixed value to the transmitter NCO / upconverter and so the digital carrier frequency is unchanged by the frequency error in the system clock. The frequency spectrum of a digital signal is normalized based on the sampling frequency, so the spectrum of the (real) digital signal just before it is sent to the DAC will look similar to that shown in Figure 1.36.

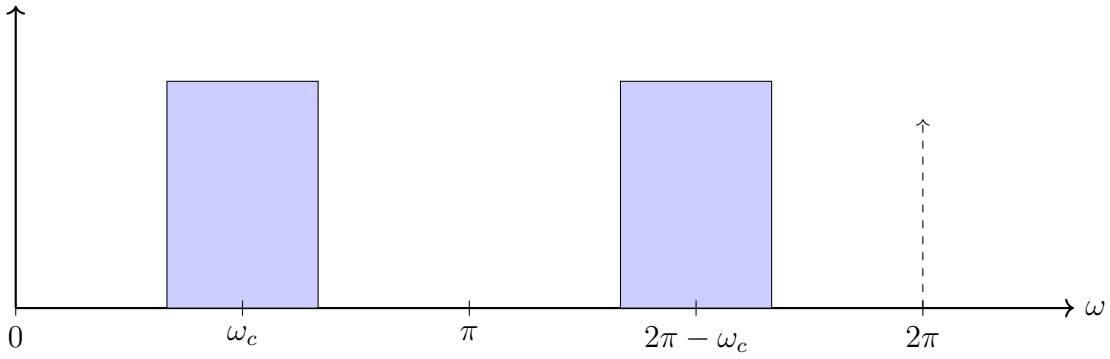

Figure 1.36: Frequency spectrum in transmitter after upconversion.

When the samples of the RF signal are sent from the FPGA to the DAC for conversion to the analog domain, the frequency axis in the spectral plot above is rescaled such that $F_{DAC} = F_{s,t} = 25\text{MHz} + \Delta F$ corresponds to $\omega = 2\pi$. This topic was discussed at length in EE 362 and EE 365, so please refer to your notes from those classes for more details if necessary. Note that to simplify this discussion the amplitude scaling and rolloff effects that occur during the digital-to-analog conversion process are ignored. The resulting spectrum at the DAC output is illustrated in Figure 1.37.

Due to the rescaling, the actual carrier frequency of the analog signal, which will be referred to as F_c', is shifted from its ideal value of $F_c = 6.25\text{MHz}$.

$$F_c' = \frac{\omega_c}{2\pi} F_{s,t} \tag{1.7}$$

$$F_c' = F_c + \frac{\omega_c}{2\pi}\Delta F = 6.25\text{MHz} + 0.25\Delta F \tag{1.8}$$

Thus, for this specific carrier frequency, one-quarter of the error in the frequency of the system clock shows up in the carrier frequency of the transmitted signal. In general, for an

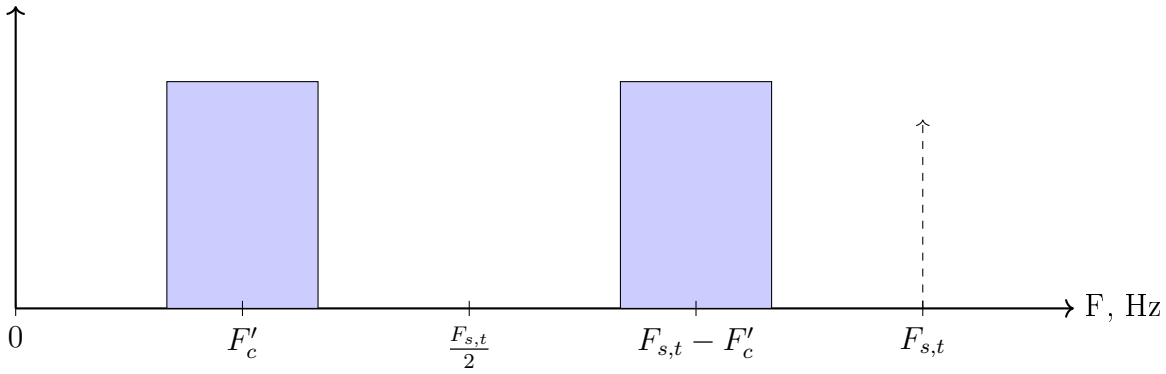

Figure 1.37: Frequency spectrum in transmitter after digital-to-analog conversion.

arbitrary digital carrier frequency of ω_c, the error in the analog carrier frequency due to the error in the system clock rate will be given by equation (1.7).

The DAC output signal is generally passed through a reconstruction filter and then sent to the receiver where it is sampled at a rate $F_{ADC} = F_{s,r}$, which is equal to exactly 25MHz in this example. This sampling process has the effect of rescaling the frequency axis of the spectrum shown in Figure 1.37, such that the digital frequency of the samples of an analog signal at frequency F becomes $\omega = \frac{F}{F_{s,r}} * 2\pi$. Therefore, the spectrum seen at the input to the receiver FPGA will be similar to that shown in Figure 1.38.

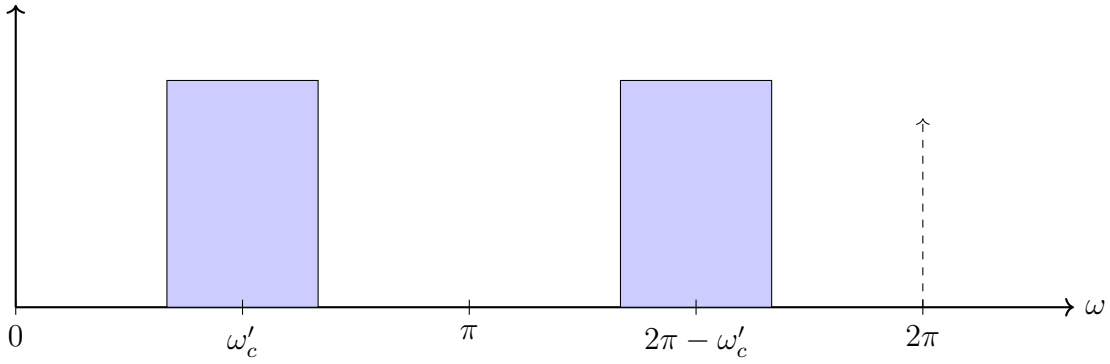

Figure 1.38: Frequency spectrum in receiver after analog-to-digital conversion.

The carrier frequency of the digital signal seen by the receiver, which will be referred to as ω_c' is given by:

$$\omega_c' = \frac{F_c'}{F_{s,r}} * 2\pi \tag{1.9}$$

$$\omega_c' = \frac{\frac{\omega_c}{2\pi} F_{s,t}}{F_{s,r}} * 2\pi \tag{1.10}$$

$$\omega_c' = \omega_c \frac{F_{s,t}}{F_{s,r}} \tag{1.11}$$

As shown above, the original digital carrier frequency ω_c has been scaled by a factor of $\frac{F_{s,t}}{F_{s,r}}$ due to the clock frequency variations. It is this scaled version of the digital carrier frequency which is observed by the receiver, rather than the ideal frequency. In our current example, it is clear that if ΔF is positive, the ratio $\frac{F_{s,t}}{F_{s,r}}$ will be greater than 1, meaning that $\omega_c' > \omega_c$. Conversely, if ΔF is negative, $\frac{F_{s,t}}{F_{s,r}}$ will be less than 1, so $\omega_c' < \omega_c$. Also, as a sanity check, note that if $\Delta F = 0$, $\omega_c' = \omega_c$, and the digital carrier frequency seen by the receiver matches its nominal value exactly.

For notational convenience, the error in the digital carrier frequency seen by the receiver due to oscillator variability will be denoted $\Delta\omega_c$ in the following sections, i.e. $\Delta\omega_c = \omega_c' - \omega_c$.

Questions for the class to discuss:

- Assuming a 50ppm transmitter oscillator, what is the absolute analog frequency error ΔF in our EE 465 system?
- What is the maximum error in the receiver digital carrier frequency $\Delta\omega_c = \omega_c' - \omega_c$?
- Repeat the above two questions for an ideal transmit oscillator and a 50ppm receiver oscillator.
- What is the worst-case value of $\Delta\omega_c$ if the transmitter and receiver oscillators both have 50ppm accuracy specifications?

As argued above, the combined effect of the upconversion, digital-to-analog conversion, and analog-to-digital conversion processes in the presence of oscillator inaccuracies is to upconvert the digital baseband channel to a carrier frequency of ω_c' instead of ω_c. The input to the receiver FPGA may therefore be represented as (assuming an ideal channel and no delay to simplify the notation):

$$x_r[n] = x_{bb,i}[n]\cos(\omega_c' n) - x_{bb,q}[n]\sin(\omega_c' n), \tag{1.12}$$

where $x_{bb,i}$ and $x_{bb,q}$ represent the in-phase and quadrature portions of the complex baseband signal in the transmitter just prior to upconversion.

Notice that in order to properly downconvert the received signal to baseband, the receiver should multiply the incoming signal by $e^{-j\omega_c' n} = \cos(\omega_c' n) - j\sin(\omega_c' n)$. However, the receiver constructed in Deliverables 4 and 5 makes no attempt to determine the value of $\Delta\omega_c$ or ω_c'. Rather, it merely downconverts using a NCO with a fixed digital frequency of ω_c. This process is depicted in Figure 1.39 below.

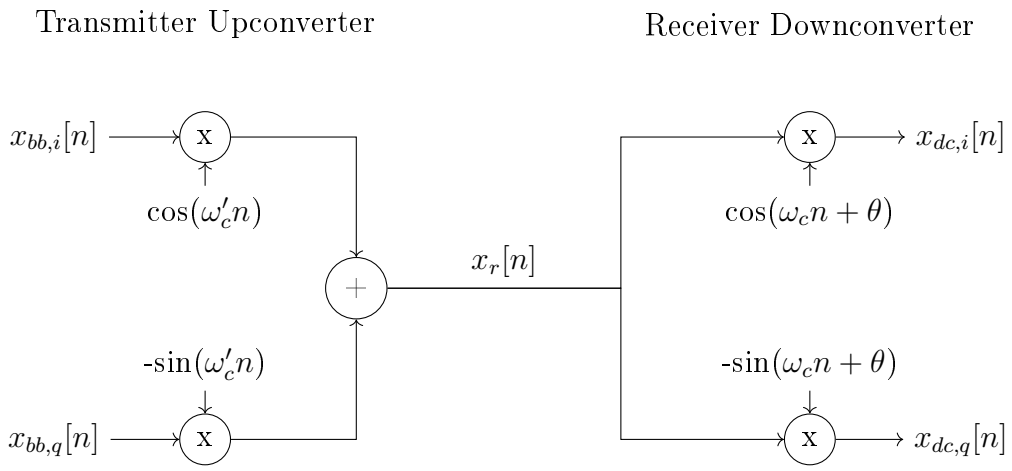

Figure 1.39: Mathematical model of upconversion and downconversion process in presence of frequency error

The downconversion process may be represented mathematically as follows. Note that the model includes an arbitrary phase offset θ, which is added to the carrier used for downconversion in the receiver to represent the fact that the two oscillators are not synchronized and are almost certain to be "out-of-phase" with one another.

The in-phase component of the downconverter output is:

$$x_{dc,i}[n] = x_r[n]\cos(\omega_c n + \theta) \tag{1.13}$$

$$x_{dc,i}[n] = (x_{bb,i}[n]\cos(\omega_c' n) - x_{bb,q}[n]\sin(\omega_c' n))\cos(\omega_c n + \theta) \tag{1.14}$$

$$x_{dc,i}[n] = x_{bb,i}[n]\cos(\omega_c' n)\cos(\omega_c n + \theta) - x_{bb,q}[n]\sin(\omega_c' n)\cos(\omega_c n + \theta) \tag{1.15}$$

$$x_{dc,i}[n] = \frac{1}{2}x_{bb,i}[n](\cos(\Delta\omega_c n - \theta) + \cos((2\omega_c + \Delta\omega_c)n + \theta)) - \frac{1}{2}x_{bb,q}[n](\sin(\Delta\omega_c n - \theta) + \sin((2\omega_c + \Delta\omega_c)n + \theta)) \tag{1.16}$$

Subsequent low-pass filtering blocks in the receiver which are associated with the downsampling and matched filtering operations will remove the double-frequency components, such that:

$$x_{dc,i}[n] \approx \frac{1}{2}x_{bb,i}[n]\cos(\Delta\omega_c n - \theta) - \frac{1}{2}x_{bb,q}[n]\sin(\Delta\omega_c n - \theta) \tag{1.17}$$

Similarly, the quadrature portion of the downconverter output can be expressed as:

$$x_{dc,q}[n] = -x_r[n]\sin(\omega_c n + \theta) \tag{1.18}$$

It is left as an exercise to show that after the double frequency components are removed, (1.18) reduces to:

$$x_{dc,q}[n] \approx \frac{1}{2}x_{bb,i}[n]\sin(\Delta\omega_c n - \theta) + \frac{1}{2}x_{bb,q}[n]\cos(\Delta\omega_c n - \theta) \tag{1.19}$$

Combining (1.17) and (1.19) and omitting the scaling factors of $\frac{1}{2}$ for notational convenience, the downconverter output may be written in terms of the original baseband signals from the transmitter in matrix format as:

$$\begin{bmatrix} x_{dc,i}[n] \\ x_{dc,q}[n] \end{bmatrix} = \begin{bmatrix} \cos(\Delta\omega_c n - \theta) & -\sin(\Delta\omega_c n - \theta) \\ \sin(\Delta\omega_c n - \theta) & \cos(\Delta\omega_c n - \theta) \end{bmatrix} \begin{bmatrix} x_{bb,i}[n] \\ x_{bb,q}[n] \end{bmatrix} \tag{1.20}$$

or, equivalently, in complex format with $x_{bb} = x_{b,i} + jx_{bb,q}$ and $x_{dc} = x_{dc,i} + jx_{dc,q}$:

$$x_{dc}[n] = x_{bb}[n]e^{j(\Delta\omega_c n - \theta)}. \tag{1.21}$$

Equations (1.20) and (1.21) should look familiar, as numerous expressions of this form were studied in EE 456. Such expressions correspond to rotating an input vector, $x_{dc}[n]$ in this case, by a specified angle, which in this case is $\Delta\omega_c n - \theta$.

Notice that the angle of rotation contains two components. The first is a constant component $-\theta$, which arises from the initial difference in phase between the transmitter and receiver oscillators at time $n = 0$. This term is commonly referred to as the "phase offset". The second component of the angle of rotation, $\Delta\omega_c n$, is referred to as the "frequency offset" and represents the difference in frequency between the two oscillators. The frequency offset yields a time-varying rotation at a rate of $\Delta\omega_c$ radians per sample.

To summarize the above analysis, it has been shown that phase and frequency mismatches between the transmitter and receiver oscillators cause the complex baseband communication signal to rotate in the I-Q plane. QAM communication systems encode information in both the amplitude and phase of the constellation, so an arbitrary phase rotation has the potential to cause major problems for the receiver of a QAM signal. Figure 1.40 shows an example of how this rotation can cause the received constellation points to cross the decision boundaries of a coherent QAM demodulator, such as that constructed in Deliverables 1-5 of EE 465. Due to this phenomenon, without an accurate method of estimating and correcting for the phase and frequency offsets, a QAM receiver is doomed to suffer from high bit and symbol error rates.

Questions for the class to discuss:

- Equation (1.21) describes the per-sample rotation seen at the receiver input. In our EE 465 receiver, this corresponds to the 25MHz system clock rate. How does the amount of per-symbol rotation relate to the per-sample rotation rate?
- If $\Delta\omega_c = 0.05$ radians per sample in our EE 465 receiver, what is the per-symbol rotation seen by the slicer?

Mitigating Frequency and Phase Error

As shown in the previous section, left unchecked, frequency and phase error can cause significant degradation to the performance of a QAM communication system. Since it is generally not possible to prevent frequency and phase error between distinct transmitter and receiver

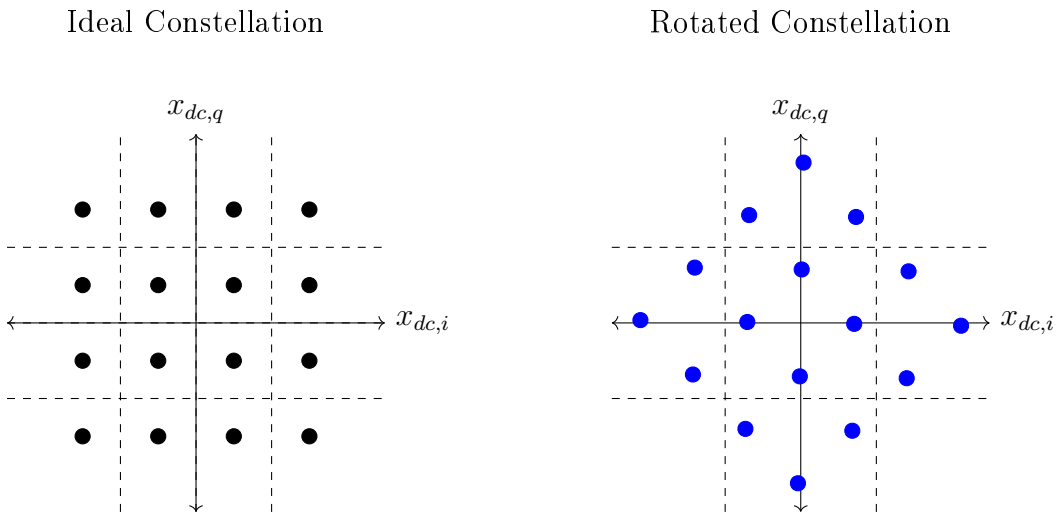

Figure 1.40: Illustration of the effect of constellation rotation for an example rotation of $\pi/2$ radians. Notice that the rotation can cause the received symbols to cross the decision boundaries, resulting in bit and symbol errors.

oscillators, it is necessary to modify our receiver to combat the effects of these phenomena. This generally involves two steps:

1. Estimating the frequency and phase offset from observations of the received signal.
2. Applying an equal and opposite rotation to the received signal in order to counteract the rotation caused by the offsets.

These two processes are described in the following sections.

Estimating Frequency and Phase Offset

The task of estimating the frequency and phase offset present in a particular received signal is conceptually simple, but can be difficult in practice. The design of specialized algorithms which generate accurate estimates across a variety of channel conditions, are robust against noise and interference, and are efficient to implement in hardware is a topic well beyond the scope of this course. Indeed, a large number of very bright people have dedicated their academic careers to solving this problem, and there exist multiple books containing many thousands of pages which describe the algorithms that have been developed.

In EE 465, a specific algorithm for frequency recovery will be provided in order to simplify the process. The algorithm and a circuit which implements it will be discussed in detail in a subsequent section. For the time being, it is sufficient to understand that the outputs of the circuit will be estimates of the true phase offset θ and true frequency offset $\Delta\omega_c$. By mathematical convention, estimates of parameter values are indicated by placing a $\hat{}$ on the symbol in question. Thus, the phase and frequency offset estimates that our circuit generates will be denoted as $\hat{\theta}$ and $\widehat{\Delta\omega_c}$.

Compensating for Frequency and Phase Offset

Once we have estimated the extent to which the constellation is rotating due to frequency and phase offset, we can counteract that rotation in order to bring the constellation points back to their ideal locations. Mathematically, multiplying by a complex exponential is equivalent to a rotation, so this so-called "derotation" operation can be expressed as

$$x_{derot}[n] = x_{dc}[n]e^{-j(\widehat{\Delta\omega_c}n-\hat{\theta})}, \tag{1.22}$$

where x_{derot} is the derotated signal. In practice, derotation is typically implemented through the following matrix equation, which is analogous to (1.20).

$$\begin{bmatrix} x_{comp,i}[n] \\ x_{comp,q}[n] \end{bmatrix} = \begin{bmatrix} \cos(\widehat{\Delta\omega_c}n-\hat{\theta}) & \sin(\widehat{\Delta\omega_c}n-\hat{\theta}) \\ -\sin(\widehat{\Delta\omega_c}n-\hat{\theta}) & \cos(\widehat{\Delta\omega_c}n-\hat{\theta}) \end{bmatrix} \begin{bmatrix} x_{dc,i}[n] \\ x_{dc,q}[n] \end{bmatrix} \tag{1.23}$$

The derotation process described by equation (1.23) is typically implemented using the structure shown in Figure 1.41 below.

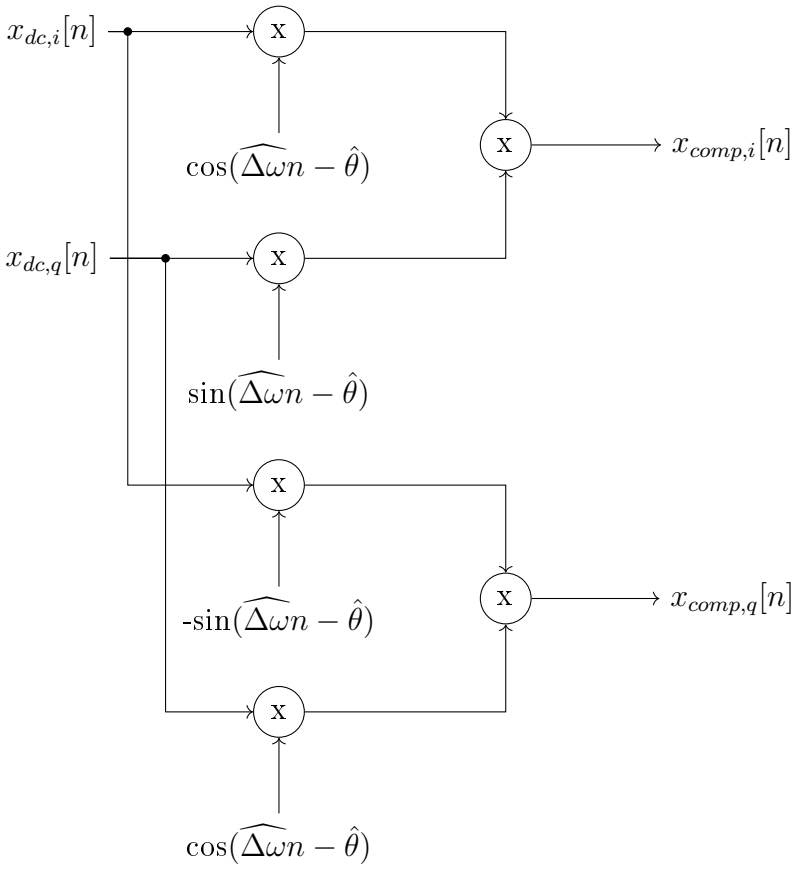

Figure 1.41: Implementation structure of derotation module.

Of course, the derotation process is only as good as the estimates of the phase and frequency offset that are provided. This may be seen mathematically by substituting the expression for x_{dc} from equation (1.21) into (1.22), which gives:

$$\begin{aligned} x_{derot}[n] &= x_{dc}[n]e^{-j(\widehat{\Delta\omega_c}n-\hat{\theta})} \\ &= x_{bb}[n]e^{j(\Delta\omega_c n-\theta)}e^{-j(\widehat{\Delta\omega_c}n-\hat{\theta})} \\ &= x_{bb}[n]e^{j((\Delta\omega_c-\widehat{\Delta\omega_c})n-(\theta-\hat{\theta}))} \end{aligned} \tag{1.24}$$

It is apparent from equation (1.24) that if the estimates are equal to the true frequency and phase offset, the derotation will perfectly cancel the original rotation, such that $x_{derot}[n] = x_{bb}[n]$. However, if there is some error in the estimates, residual offsets equal to $\theta - \hat{\theta}$ and $\Delta\omega_c - \widehat{\Delta\omega_c}$ will persist after derotation. These residual offsets can potentially degrade the slicer performance.

Initial Phase and Frequency Estimation Techniques

As with the slicer reference level recovery circuit developed in Deliverable 5, phase and frequency recovery algorithms generally include two components:

- An initial estimation phase which executes at the beginning of the communication process and produces a coarse estimate of the parameter of interest.
- A continuous refinement loop which tracks the parameters throughout the transmission and attempts to remove any residual offsets.

This section of the document focuses on techniques for performing the initial coarse estimation of phase and frequency offsets in the receiver. The continuous refinement process generally includes a phase locked loop (PLL), and is discussed in more detail in the following section of this document.

It was established earlier that phase and frequency offsets cause the constellation to rotate away from its ideal orientation. Therefore, the process of estimating these offsets conceptually involves determining how much the constellation has rotated and how quickly that rotation is occurring. This process can be facilitated through the transmission of a symbol or sequence of symbols which is known in advance by the receiver. The receiver can then compare the phase of the received signal to that of the known transmitted signal in order to generate the required estimates.

To illustrate how this could work, consider the following simple example, in which a preamble sequence consisting of two symbols is transmitted prior to the actual data symbols. Both preamble symbols correspond to the outer constellation point in the upper right portion of a 16-QAM constellation, which is equal to $1 + j$. As usual, the preamble symbols are followed by a data payload consisting of random symbols from our 16-QAM constellation. The overall packet format is depicted in Figure 1.42 below.

Due to phase and frequency offset, the input to the slicer in the receiver will not be exactly $1 + j$. Rather, the slicer input for the preamble symbols will be (assuming correct operation of the timing recovery circuit, ignoring noise and interference, and assuming unity channel gain):

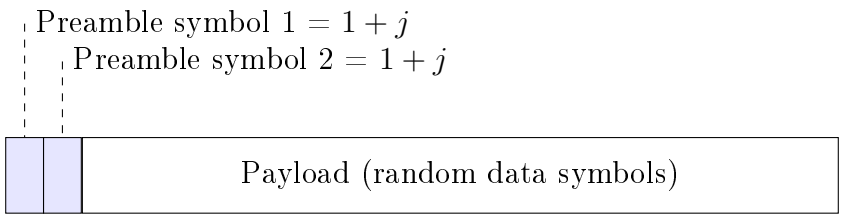

Figure 1.42: Packet format for simple phase and frequency estimation example.

$$x_{tr}[n] = (1+j)e^{j(16\Delta\omega_c n-\theta)} \quad ; \quad n \,\epsilon\, \{0,1\} \tag{1.25}$$
$$= (1+j)e^{j(\Delta\omega_c' n+\theta')} \quad ; \quad n \,\epsilon\, \{0,1\} \tag{1.26}$$

Note that the 16 in the equation above arises from the net downsample by 16 which occurs between the receiver downconverter (which is the reference point for the $\Delta\omega_c$ as defined earlier) and the slicer input. In effect, the frequency offset seen by the slicer on a per-symbol basis is 16x larger that that seen on a per-sample basis at the downconverter output. To simplify the notation, we have defined new variables $\Delta\omega_c' = 16\Delta\omega_c$ and $\theta' = -\theta$ in equation (1.26).

Evaluating equation (1.26) for $n = 0$ and $n = 1$, we have:

$$x_{tr}[0] = (1+j)e^{j\theta'} \tag{1.27}$$
$$x_{tr}[1] = (1+j)e^{j(\Delta\omega_c'+\theta')} \tag{1.28}$$

Figure 1.43 illustrates these transmitted and received preamble symbols graphically.

Knowing the expected and actual values of $x_{tr}[0]$ and $x_{tr}[1]$, one can generate estimates of the parameters θ' and $\Delta\omega_c'$ by comparing the angles of the received points to their expected values (the values we would observe if no frequency or phase offsets are present). Perhaps the simplest such approach would be to first estimate θ' based off of $x_{tr}[0]$ as follows:

$$\hat{\theta'} = \text{Ideal Angle of } x_{tr}[0] - \text{Actual Angle of } x_{tr}[0] \tag{1.29}$$
$$= \pi/4 - \text{Actual Angle of } x_{tr}[0] \tag{1.30}$$

As illustrated in Figure 1.43, an estimate of the frequency offset can be generated by looking at the difference in the angles of $x_{tr}[1]$ and $x_{tr}[0]$. Specifically,

$$\widehat{\Delta\omega_c} = \text{Actual Angle of } x_{tr}[1] - \text{Actual Angle of } x_{tr}[0] \tag{1.31}$$

Unfortunately, noise and other interference may corrupt the signal x_{tr}, causing the samples to deviate from their proper locations. Such noise is likely to change the angles of the samples that are being used for the phase and frequency estimation. For this reason, real communication systems typically use a preamble which is longer than two symbols, allowing

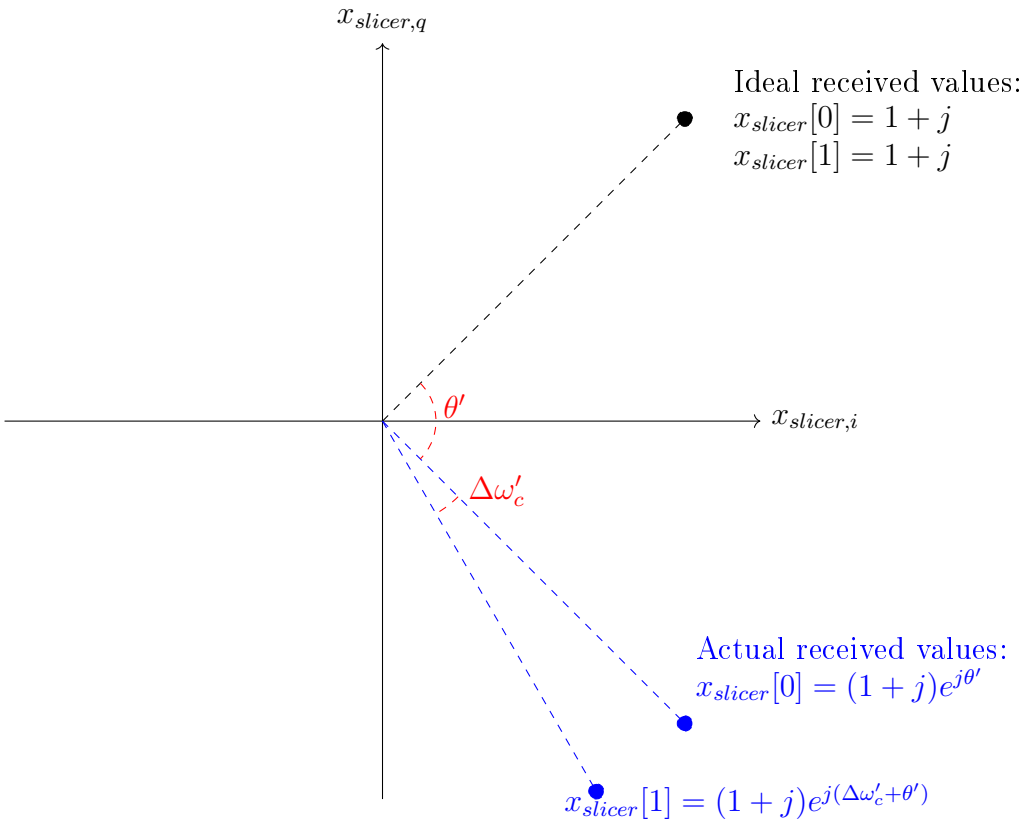

Figure 1.43: Illustration of effect of frequency and phase offset on ideal and actual received symbols for two-symbol preamble example.

the receiver to average multiple observations and thereby reduce the effect of the noise. In Deliverable 6, we will reuse the 27 symbol preamble from Deliverable 5.

Equations (1.29) and (1.31) require the determination of the angle of a complex input sample. Mathematically, this may be done through a $\tan^{-1}$(Imag part/Real part), although it is important to notice that the $\tan^{-1}$ function is periodic and therefore unable to uniquely specify an angle in the complex plane, i.e., $\tan^{-1}(\phi)$ =$\tan^{-1}(-\pi + \phi)$.

Questions for the class to discuss:

- Do the signs of the real and imaginary parts of the complex input tell us anything about the angle of the complex number?
- Is there a way we can use the signs to eliminate the ambiguity of the $\tan^{-1}$ function?

Since the task of finding the angle of a complex value is very common in signal processing, there are pre-built FPGA circuits in the form of IP cores / MegaFunctions that implement this functionality. Such circuits are commonly based on the COordinate Rotation DIgital Computer (CORDIC) algorithm, which iteratively computes the angle by performing

a series of computationally efficient rotations. While the algorithm is quite interesting and instructive, it is beyond the scope of EE 465. In this class, we will simply instantiate the ALTERA_CORDIC block that is provided by Altera for the purpose of computing the angles of complex signals.

Continuous Phase and Frequency Tracking

In a previous section, we saw that that small residual offsets will generally persist after the initial phase and frequency offset estimation and correction. Even if we are extremely lucky and our initial estimates are exactly correct, it is possible that the transmit and receive oscillators will slowly drift apart over time, causing the input to the slicer to rotate and leading to bit errors. To prevent such occurrences, a dedicated circuit is generally constructed to track these small changes.

Continuous tracking of the carrier phase and frequency in a digital demodulator circuit is generally performed using a phase locked loop (PLL). As discussed in detail in EE 461, PLLs are circuits that construct a periodic signal and manipulate its phase in order to "lock" the frequency and phase of the constructed signal to that of a periodic input reference signal. In our application, the PLL attempts to generate a sinusoidal signal which is perfectly matched in phase and frequency to the rotation of the timing recovery output so that such rotation can be canceled out through a derotation process.

The typical structure of a PLL is shown in Figure 1.44 below. As shown in the figure, the input sinusoid and constructed sinusoid are passed into a phase detector, which determines the phase error between the two sinusoids. The phase error is passed through a loop filter, which has the effect of suppressing noise on the input in order to generate a stable reference. Finally, the output of the loop filter, which is a phase value, is passed through a LUT, which performs a phase-to-sinusoid conversion in order to generate the constructed sinusoid.

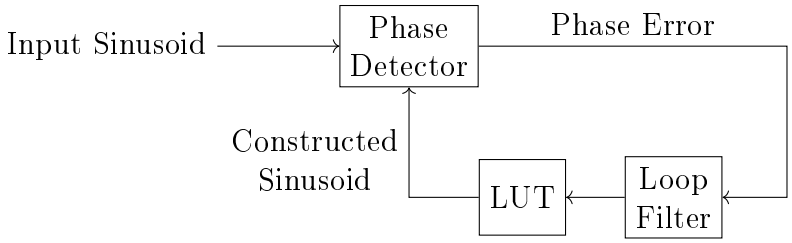

Figure 1.44: Structure of a basic PLL.

A common PLL structure for use in a digital communications receiver is shown in Figure 1.45. This circuit contains all of the elements of the basic PLL shown in Figure 1.44 above, although it may not appear that way at first glance. The fundamental reason for the differences is that the input signal to our PLL (which is the output of the timing recovery circuit) is of a somewhat different format than that considered in EE 461. The specific unique features of the PLL input signal are:

- It is a complex signal, rather than a purely real signal, as was considered in EE 461. Furthermore, the sinusoid which is being constructed by the PLL is also a complex signal. This means that phase detector must determine the phase difference between two complex signals, rather than two real signals.
- The PLL input is a modulated signal containing data symbols drawn from a 16-QAM constellation, rather than a pure sinusoid, as considered in EE 461. Mathematically, the PLL input may be represented as $x_{tr}[n] = s_n e^{j(\Delta\omega_c' n + \theta')}$, where s_n is the complex 16-QAM symbol transmitted during the n'th symbol interval.

As a result of these differences, the basic PLL studied in EE 461 will not work, and a more complicated structure, such as that shown in Figure 1.45, is required. The following sections discuss the individual sections of the PLL in more detail.

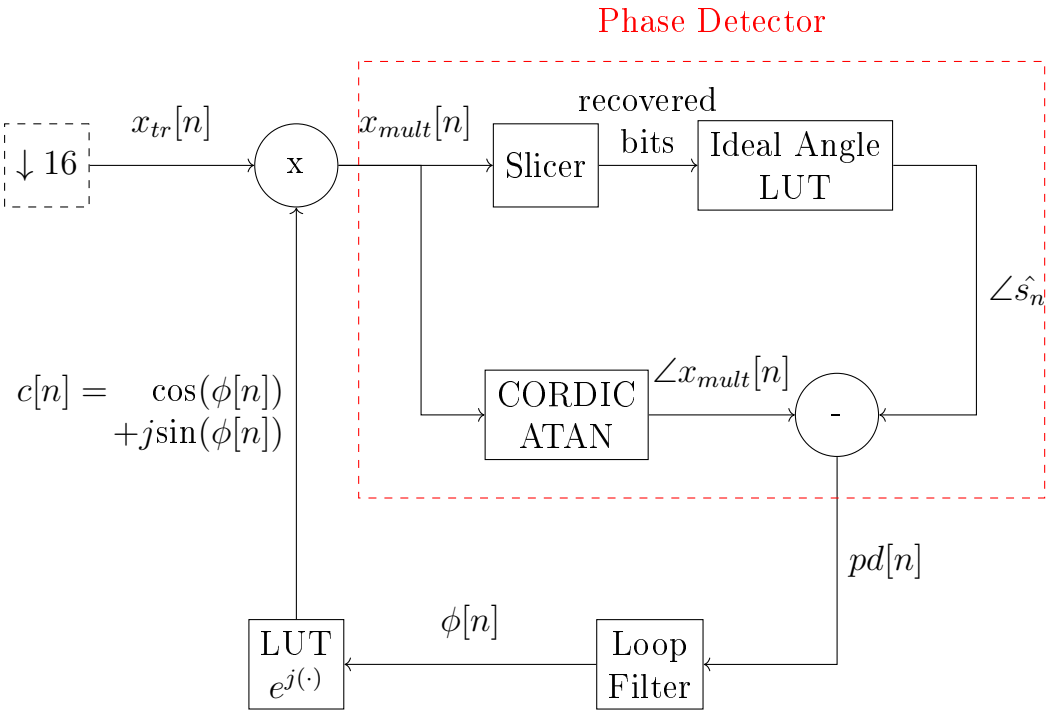

Figure 1.45: Structure of a practical PLL for recovering phase and frequency in a QAM system.

Phase-to-Sinusoid Conversion

As in a conventional PLL, the input to the phase-to-sinusoid block is a phase term which we will refer to as $\phi[n]$, representing the estimated phase of the input sinusoid. However, since the input to the PLL is a complex signal, the sinusoid which is generated by the PLL must also be complex. Therefore, the phase-to-sinusoid block must be modified to output not only the cosine of the input phase, but a true complex sinusoid of the form

$e^{j\phi[n]} = \cos(\phi[n]) + j\sin(\phi[n])$. One method of achieving this goal woudl be to implement two separate LUTs (one to generate sine, one to generate cosine) which are driven by a single phase input.

Phase Detector

The first component of the proposed phase detector shown in Figure 1.45 looks like a simple multiplier at first glance. However, it is important to note that since both of the phase detector inputs are complex signals, this block is actually performing a complex multiplication. From a hardware perspective, this means that the multiplication involves 4 real multiplications and 2 real additions.

In general, the angular difference between two complex numbers can be found by multiplying one number by the complex conjugate of the other. For two arbitrary complex numbers a and b, this may be expressed as:

$$\begin{aligned} p &= ab^* \\ &= |a|e^{j\angle a}|b|e^{-j\angle b} \\ &= |a||b|e^{j(\angle a - \angle b)} \end{aligned} \tag{1.32}$$

Thus, the result of the complex multiplication is a complex number whose angle is equal to the angular difference between the two inputs. In our particular case, the two inputs are actually equal to $x_{tr}[n] = s_n e^{j(\Delta\omega_c' n + \theta')}$ and $c[n] = e^{j\angle c[n]}$, and so the output from the complex multiplier will be:

$$\begin{aligned} x_{mult}[n] &= x_{tr}[n]c[n]^* \\ &= s_n e^{j(\Delta\omega_c' n + \theta')} e^{-j\angle c[n]} \\ &= s_n e^{j(\Delta\omega_c' n + \theta' - \angle c[n])} \qquad (1.33) \\ &= |s_n| e^{j(\Delta\omega_c' n + \theta' - \angle c[n] + \angle s_n)} \qquad (1.34) \end{aligned}$$

The goal of the frequency/phase recovery circuit in general and the PLL specifically is to generate a sinusoidal signal which is matched to the rotation of the timing recovery output signal. In essence, the job of the phase detector is to compute the difference between the phase of the input signal and the generated sinusoid so that that difference can eventually be driven to zero through a feedback process. This phase difference, which is the desired output of the phase detector, may be represented mathematically as:

$$pd_{desired}[n] = \Delta\omega_c' n + \theta' - \angle c[n]. \tag{1.35}$$

It was mentioned earlier that the fact that the PLL input is a modulated signal of the form $x_{tr}[n] = s_n e^{j(\Delta\omega_c' n + \theta')}$ complicates the implementation of the phase detector. If the PLL input signal was unmodulated (did not contain the s_n part), the angle of the complex output from the multiplier represented by equation (1.34) would exactly match $pd_{desired}[n]$. Indeed, if the PLL input signal had not been modulated, we could just compute the angle of the complex multiplier's output (using a CORDIC block, for example) and use the result as our phase detector output.

Unfortunately, it is apparent from equation (1.34) that the complex multiplier's output is corrupted by an unwanted $\angle s_n$ term representing the phase angle of the transmitted 16-QAM constellation point. In order to obtain the desired phase detector output, we need a way of removing $\angle s_n$ from equation (1.34).

The key insight that can be used to remove $\angle s_n$ from the output of the complex multiplier is that since s_n is drawn from a 16-QAM constellation, its angle must be exactly equal to that of one of the 16 possible constellation points. If we knew which point had been transmitted, we could easily subtract out its contribution to the angle of the complex multiplier output, similar to the way we subtracted out the angle of the known preamble constellation point in equation (1.29). Although the true transmitted constellation point is not known with certainty by the receiver, it is possible to generate a "best guess" of which of the 16 possible points was transmitted. In fact, processing an input signal and determining the most likely transmitted constellation point, $\hat{s_n}$ is precisely the job of the slicer.

Therefore, it is possible to generate an estimate of the phase error by passing the output of the complex multiplier into the slicer, as shown in Figure 1.45. The output of the slicer, which represents an estimate of the most likely constellation point is passed into a LUT which is pre-loaded with the correct angle for each of 16-QAM constellation points, according to Table 1.1 below. Therefore, after the most likely constellation point, $\hat{s_n}$, has been selected, $\angle\hat{s_n}$ may be read out of the LUT.

Constellation Point	Angle (°)	Constellation Point	Angle (°)
$(1+j)$	45.00	$(1-j)$	-45.00
$(3+j)$	18.43	$(3-j)$	-18.43
$(-1+j)$	135	$(-1-j)$	-135
$(-3+j)$	161.57	$(-3-j)$	-161.57
$(1+3j)$	71.57	$(1-3j)$	-71.57
$(3+3j)$	45.00	$(3-3j)$	-45
$(-1+3j)$	108.43	$(-1-3j)$	-108.43
$(-3+3j)$	135	$(-3-3j)$	-135

Table 1.1: Ideal angles of transmitted constellation points in a 16-QAM communication system.

The final step in the phase detector is to compute the angular difference between the input to the slicer and the output of the slicer. This involves first computing the angle of the slicer's input $x_{mult}[n]$ through the use of a CORDIC inverse tangent block. As per equation (1.34), this angle will be:

$$\angle x_{mult}[n] = \Delta\omega_c' n + \theta' - \angle c[n] + \angle s_n. \tag{1.36}$$

The phase difference may then be computed by subtracting the CORDIC output from the slicer's angle output, resulting in:

$$\begin{aligned} pd[n] &= \angle x_{mult}[n] - \angle\hat{s_n} \\ &= \Delta\omega_c' n + \theta' - \angle c[n] + \angle s_n - \angle\hat{s_n}. \end{aligned} \tag{1.37}$$

A comparison of equations (1.35) and (1.37) indicates that if $s_n = \hat{s_n}$, the phase detector generates the desired output, i.e. $pd[n] = pd_{desired}[n]$.

It should be understood that the operation of the phase detector described above relies upon the assumption that the slicer is making correct decisions. If the phase difference is

sufficiently small, this is a reasonable assumption. However, if the phase difference becomes too large, the constellation will be rotated to such an extent that the received signal is likely to cross one of the slicer's decision boundaries, leading to a symbol error. Consequently, the slicer's estimate of the s_n will be in error, causing an inaccurate phase difference value to be fed back into the loop filter.

Figure 1.46 illustrates graphically how a slicer error can cause an incorrect estimate of the phase difference. In the figure, the true transmitted constellation point is indicated in black. Due to phase and / or frequency offsets, the received point at the slicer input has been rotated counterclockwise by about 22°, resulting in the white point. The white point is now closer to the red constellation point than the black constellation point, so the slicer will decide that the red constellation point was actually the one that was sent by the transmitter. Thus, the phase detector will suggest that the phase difference is approximately 4° in a clockwise direction, rather than 22° in a counterclockwise direction. Such an error will be fed into the loop filter, causing phase adjustments in the wrong direction and potentially impeding the PLL locking process.

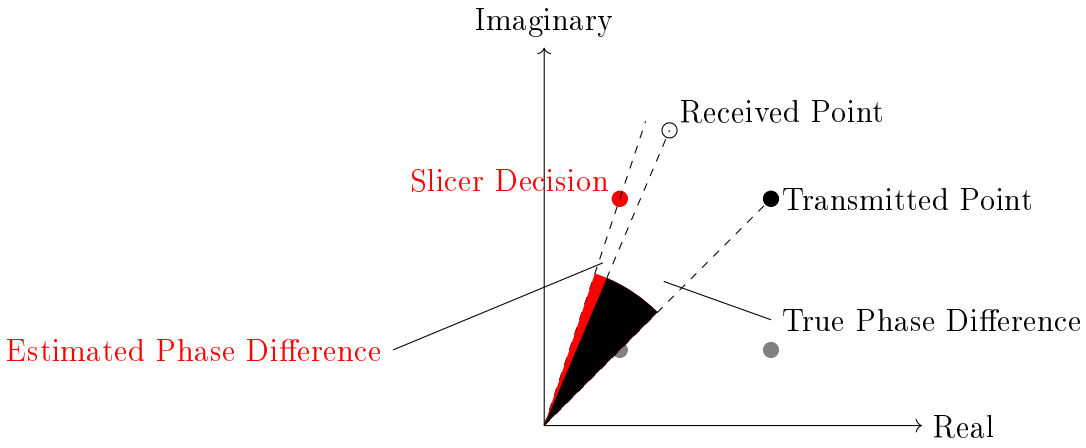

Figure 1.46: Illustration of how .

In the language of EE 461, one would say that the linear range of the phase detector illustrated in Figure 1.45 is limited. For phase differences large enough to cause symbol errors, the phase detector output will no longer be a linear function of the difference between the phases of its two inputs.

Once the system is locked, the phase difference should be very small and so symbol errors of the type described above should be rare. However, if care is not taken to correctly seed the PLL with initial estimates of the frequency and phase offsets at startup, such errors can greatly complicate the locking process. The seeding of the PLL is discussed in more detail in the following section.

Questions for the class to think about / discuss:

- What is the linear range of the phase detector described above for a 16-QAM system?

Stated another way, how much phase error can be present before the detector's output will not accurately represent the phase error?

- Would the linear range be different for a 64-QAM system? How and why?

Loop Filter

The task of the loop filter in a phase locked loop is to filter the phase difference signal $pd[n]$ in order to remove noise so that a cleaner and more stable sinusoid can be generated. Loop filters are generally classified in terms of the number of integrators they contain, as this number has significant implications for the types of signals the loop is able to track.

A "type 1" loop generally has the structure shown in Figure 1.47. Due to the fact that a type 1 loop has a single integrator, it is able to track a constant phase difference with zero steady state error. A type 1 loop is also able to accurately track the slope of a ramp phase input, but doing so requires the accumulation of a certain amount of phase error which persists when the loop is in steady state. Since the goal of our PLL is to achieve zero steady state error in tracking a phase input consisting of both a constant phase term and a phase ramp, a type 1 loop is not sufficient for this application.

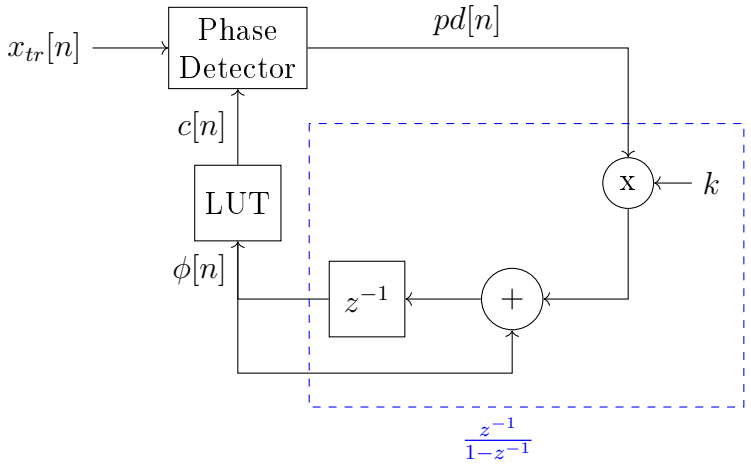

Figure 1.47: Structure of a type 1 phase locked loop.

Therefore, it is necessary to use a "type 2" loop filter inside our PLL, with a structure similar to that shown in Figure 1.48. A type 2 loop has two integrators and is therefore able to track a phase ramp with no steady state phase error, as required for our receiver. As shown in Figure 1.48, there are two constants, k_1 and k_2 which must be selected in order to design the loop filter. The choice of these constants controls the placement of the poles in the overall PLL system response, which in turn determines the noise bandwidth of the system and the speed of convergence. The designer generally needs to make a tradeoff between these two conflicting goals, as was discussed in detail in EE 461.

In the current application, both the PLL's convergence time and its ability to filter noise are of great importance. Furthermore, as discussed in the previous section, it is critical that the initial phase error is small in order to facilitate correct slicer decisions. At convergence, the two accumulators inside the loop filter will contain the true values of the phase and

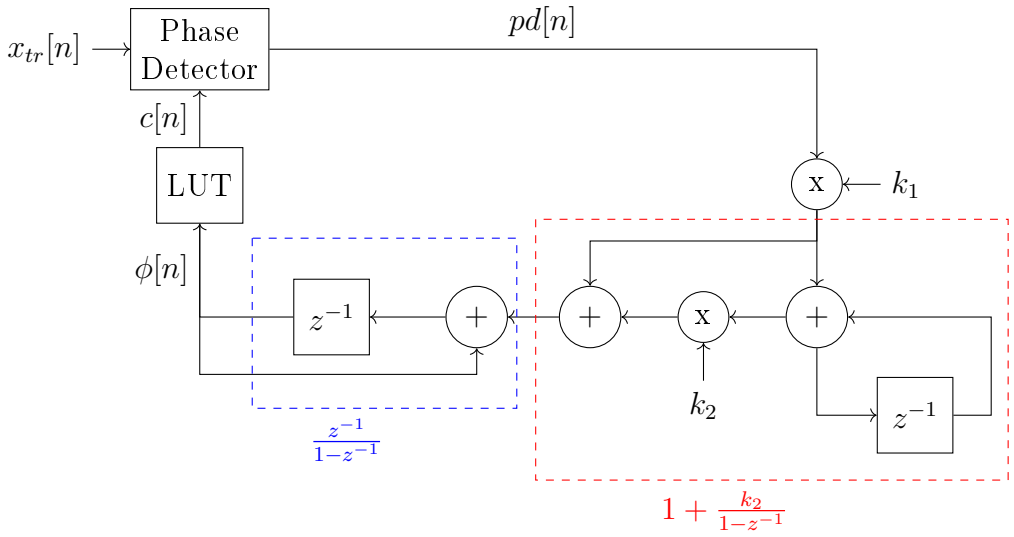

Figure 1.48: Structure of a type 2 phase locked loop.

frequency offset. By starting the PLL from a state which is closer to this final steady state, we can speed up the convergence process and minimize the chance of slicer decision errors while the PLL is converging. This may be done by loading these accumulators with initial estimates of the phase and frequency offset that are obtained based on the preamble symbols so that the PLL is close to convergence at the start of the data portion of the packet. The seeding process is illustrated in Figure 1.49 below.

Some questions for the class to discuss:

- If the initial phase offset estimate based on the preamble is 0.1 cycles, what value should be loaded into the phase accumulator? Is that value influenced by the loop constants k_1 and k_2? If so, how?
- If the initial frequency offset based on the preamble is 0.01 cycles / symbol, what value should be loaded into the frequency accumulator? Is that value influenced by the loop constants k_1 and k_2? If so, how?
- If you accidentally connected an incorrect seed value into one of the accumulators above, what would you expect to see? How could you debug the PLL?

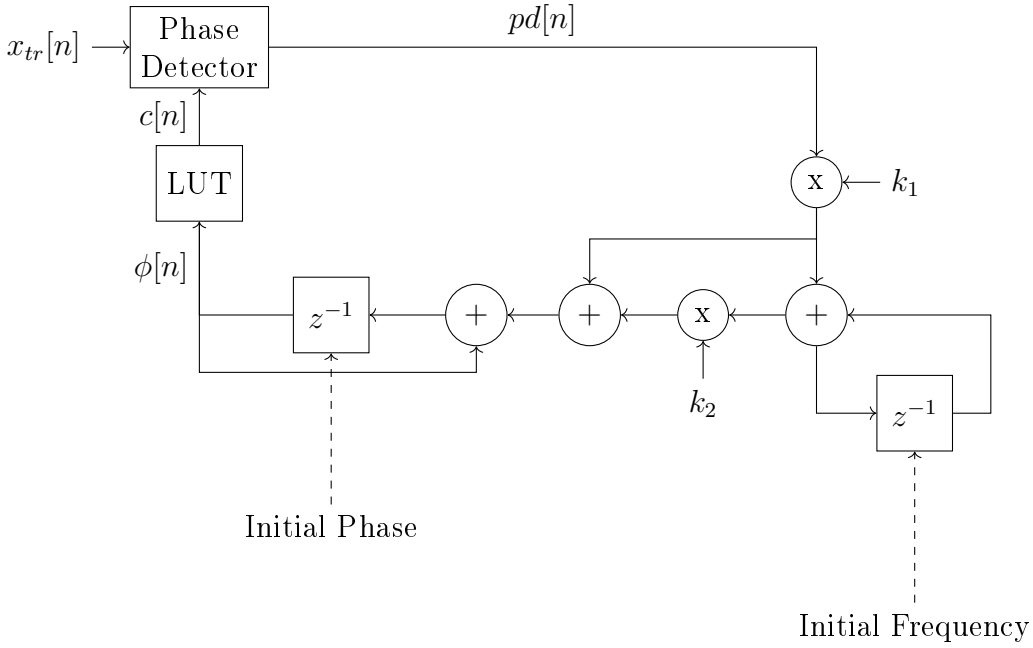

Figure 1.49: Structure of a type 2 phase locked loop.

Chapter 2

Deliverables

1 Kick Start Mini-Lab Assignment

The details of the lab exam for this Kick Start Mini-Lab can be found in Section 1 on page 123.

1.1 Part A: Sinusoidal Input

Design a length 21 Square-Root-Raised Cosine filter (an FIR filter with 21 coefficients) that has 4 samples per symbol and roll-off factor of $\beta = 0.25$.

The filter must be designed so that a full scale 1s17 sinusoidal input produces a 1s17 sinusoidal output. Furthermore the coefficients must be scaled so that the maximum of the magnitude response is exactly 1.

The filter must be designed, debugged and tested in three stages:

Stage 1: Find the filter coefficients and verify the magnitude response in Matlab.

Stage 2: Describe the filter in Verilog HDL and then write a testbench to verify its operation in Modelsim.

Stage 3: Implement the filter on the DE2-115 board and write a synthesizeable testbench to verify its magnitude response on the spectrum analyser. The ADC on the daughter board of the DE2-115 board is to be clocked at 25 MHz.

1.2 Part B: Managing Headroom

Please do the following:

1. Find the 21 sample input sequence that produces the largest possible peak value in the output. The 21 sample sequence will be referred to as the worst case input.

2. Determine the decimal worth of the peak output for worst case input.

3. Scale the coefficients of the filter so that its 1s17 output has a a peak value of 18'H1FFFF, which corresponds to a decimal worth of $1 - 2^{-17}$, for the worst case input.

4. Write a Verilog HDL description of a signal generator that generates a periodic worst case input. That is a sequence with period 21 whose values are the worst case input. The Verilog HDL does not need to be synthesizeable as it will only be used in ModelsimAltera.

5. Write a Verilog HDL testbench and verify that the peak output has a value of 18'H1FFFF and also verify that overflows do not occur.

2 Deliverable 1

The details of the lab exam for deliverable 1 are in Section 2 on page 126.

Design, build and test length-21 square-root raised cosine (SRRC) filters suitable for use in the transmitter and receiver of a 16-QAM digital communication system. The transmit filter should be constructed with the lab exam so please see Section 2 on page 126 before starting your design.

Multipliers are a precious resource on FPGAs and the inputs to some filters are such that filters can be implemented without multipliers. This is the case for the transmit SRRC filter. The students are required to redesign their transmit filter to make it multiplier free. The implementation of such a filter will depend on the order of modulation. Design the filter for 4-ASK (i.e., in-phase component of a 16-QAM modulator).

The list of deliverable are:

1. One RCV SRRC filter (see specification below).

2. One TX SRRC filter that uses multipliers (see specification below).

3. One TX SRRC filter that does not use multipliers (see specification below).

The design must meet the specification given below. The quality of the design will be evaluated based on the MER measured at the output of the RCV filter when the transmit and receive filters are connected in cascade.

2.1 Specification for Deliverable 1

1. Three length 21 Square Root Raised Cosine (SRRC) filters are to be built: two alternative implementations of a TX filter, which is the filter in the transmitter, and one implementation of an RCV filter, which is the filter in the receiver.
2. All coefficients in the filters must be less than or equal to 18 bits.
3. The input and output for each of the filters can be at most 18 bits.

4. The sampling rate for the filters is $N_{sps} = 4$ times the symbol rate. The subscript `sps` in N_{sps} signifies samples-per-symbol.
5. The RCV filter is to have a roll-off factor of $\beta = 0.25$.
6. The impulse response of the RCV filter is to be the infinite SRRC response truncated to a length of 21. Just to be clear, the center 21 coefficients of the infinite impulse are used so in effect the infinite impulse response is rectangularly windowed about its center value.
7. The TX filter is limited to a length of 21. The coefficients must be chosen to satisfy the restrictions that are about to be listed. In the end the coefficients are likely to to resemble those of a SRRC filter with a roll-off of 0.25.
8. The stop band of the TX filter starts at 0.2 cycles/sample and runs to 0.5 cycles/sample.
9. The magnitude response at all frequencies in the stop band of the TX filter must be 40 dB below the DC response.
10. The coefficients in each filter must be scaled so that the maximum possible output of the filter fits into a 1s17 format.
11. The first implementation of the TX filter may use up to 21 multipliers.
12. The second implementation of the TX filter must use 0 multipliers.

3 Deliverable 2

`This deliverable was introduced in the 2015-2016 year. It has been included at the request of the class of 2014-2015 and most notably Conor Kerslake.`

The details of the lab exam for deliverable 2 are in Section 3 on page 129.

The students are given a system that takes the output of the mapper in the transmitter and produces the decision variable in the receiver (both in-phase and quadrature components). They have to build the necessary circuits to measure the MER.

A block diagram of the baseband model of the system that the students will measure the MER is shown at the top of Figure 2.1. The circuits shown at the bottom of Figure 2.1 will have to be built to measure the MER.

The circuits that have to be built and the tasks that have to be done in preparation for measuring the MER are listed below:

1. Design, build and test a clocking system that allows circuits to be clocked at one of the three rates: 25 Msamples/second, 6.25 Msamples/second and 1.5625 Msamples/second.

   There are two ways to design multi-rate clocking systems: One way is to clock all registers with the 25 MHz system clock and use clock enables to control the rate at which a register is being loaded.

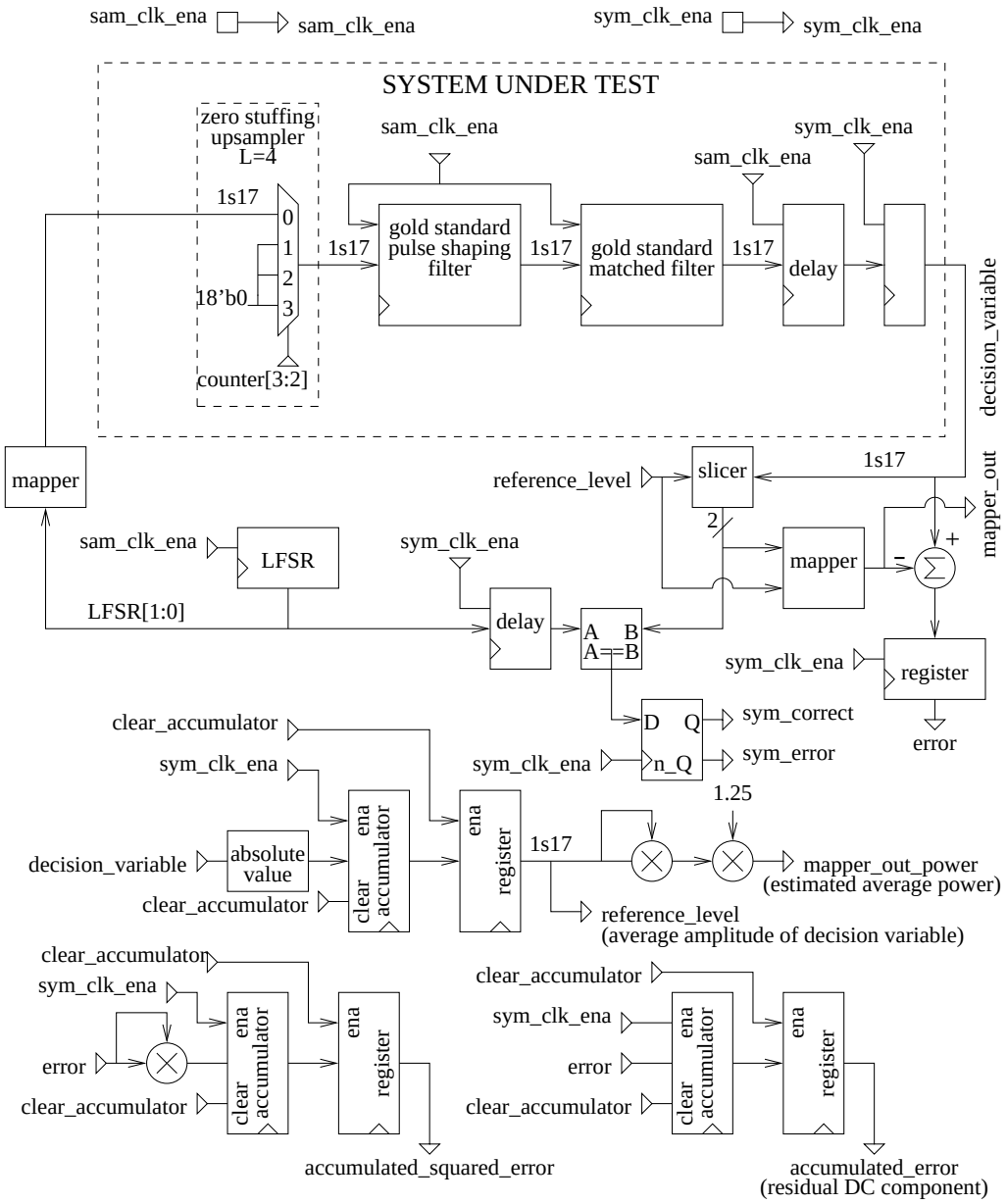

Figure 2.1: The circuits and system used as a reference in the measurement of the performance of a practical QAM transmitter and receiver.

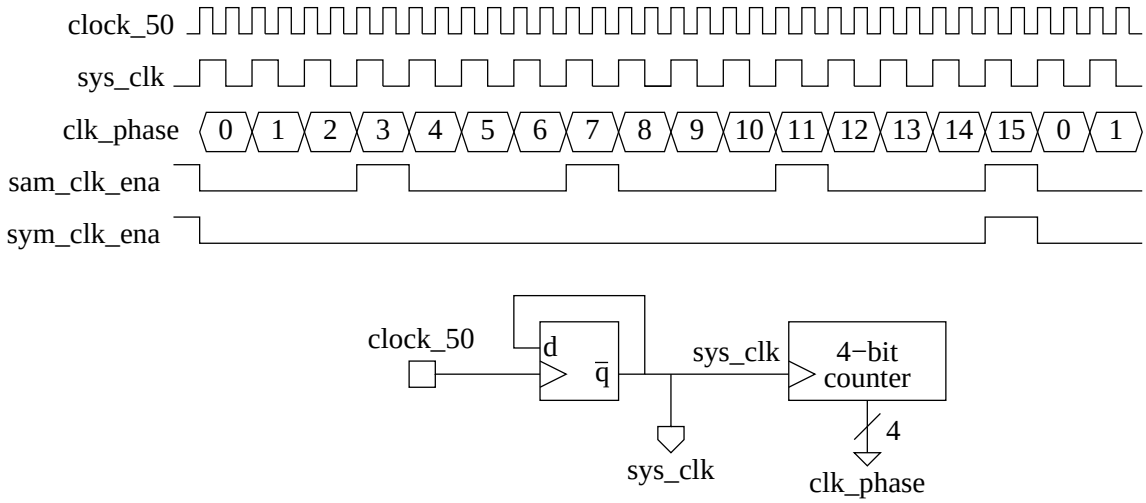

Figure 2.2: A circuit for generating the 25 MHz system clock and the 16 phases used to generating the enables for clocking at the sampling and symbol rates.

The other way is to generate three different clocks that run at three different frequencies and use the appropriate clock to clock each register. The clocks are synchronized so that data can be passed among the clock domains without error.

The clock enable method may consume more power and generate more heat, depending on the FPGA architecture. However, it is more straight forward and flexible so will be used in the design for this class. The clock generating circuit that should be used is shown in Figure 2.2.

The first task is to design build and test the circuits in Figure 2.2 along with the combinational logic circuits for the sampling clock enable, i.e. `sam_clk_ena`, and the symbol clock enable, i.e. `sym_clk_ena`.

2. Design, build and test a maximum length 22-bit LFSR.

3. Design, build and test a circuit that estimates the average magnitude of the decision variable. Since the optimum reference level for the slicer is related to the average magnitude of the decision variable by a multiplicative constant, the slicer reference level can be set from an estimate of the average magnitude of the decision variable.

4. Design, build and test a circuit that estimates the average error at the output of the mapper, which is the average power in the decision variable after the error has been removed.

   The circuit shown in Figure 2.1 estimates the average power of the output of the mapper using the relationship between the mapper output and the reference level to get

$$\mathrm{P}_{\mathrm{mapper\_output}} = 1.25 \times reference\_level^2.$$

Please calculate this power in another two ways and then compare the results. Measure the power in the output of the mapper directly and also measure the power in the decision variable and then subtract the power in the error (i.e. The power of the decision variable should be the power of the mapper output plus the power of the error).

5. Design, build and test a circuit that estimates the average power of error in the decision variable.

6. Find the MER using a calculator to compute

$$\text{MER} = 10 \log \left(\frac{\text{average power in the mapper output}}{\text{average power in the error}} \right)$$

7. Optional: construct a circuit to compute the value of the MER in dB using the equation above.

The MER measurement circuit designed in Deliverable 2 must be capable of accurately measuring MER values as high as 55dB. This will become important in Deliverables 3, 4, and 5, when the MER measurement circuit is used to test the performance of the designed 16-QAM communication system.

4 Deliverable 3

The deliverables are:

1. A gold standard for the pulse shaping filter for a CATV 16-QAM modem. (Just need one - not building both I and Q.)

2. A gold standard for the matched filter for a CATV 16-QAM modem. (Just need one - not building both I and Q.)

3. A practical cost effective pulse shaping filter. (Just need one - not building both I and Q.)

The specifications for the three deliverables are given below.

4.1 *Specifications for the Gold Standard and Practical Pulse Shaping Filter*

1. The pulse shaping filter must run at 4 times the symbol rate, i.e., 4 samples per symbol. This means the clock used for the filter has a rate 4 times that of the symbol clock, The clock used to clock the filter is referred to as the sampling clock and is denoted `sam_clk`.

2. The sampling rate is 1/4 the rate of the system clock. The system clock, referred to as `sys_clk`, is to run at 25 Msamples/second and the sampling rate is to be 6.25 Msamples/second.

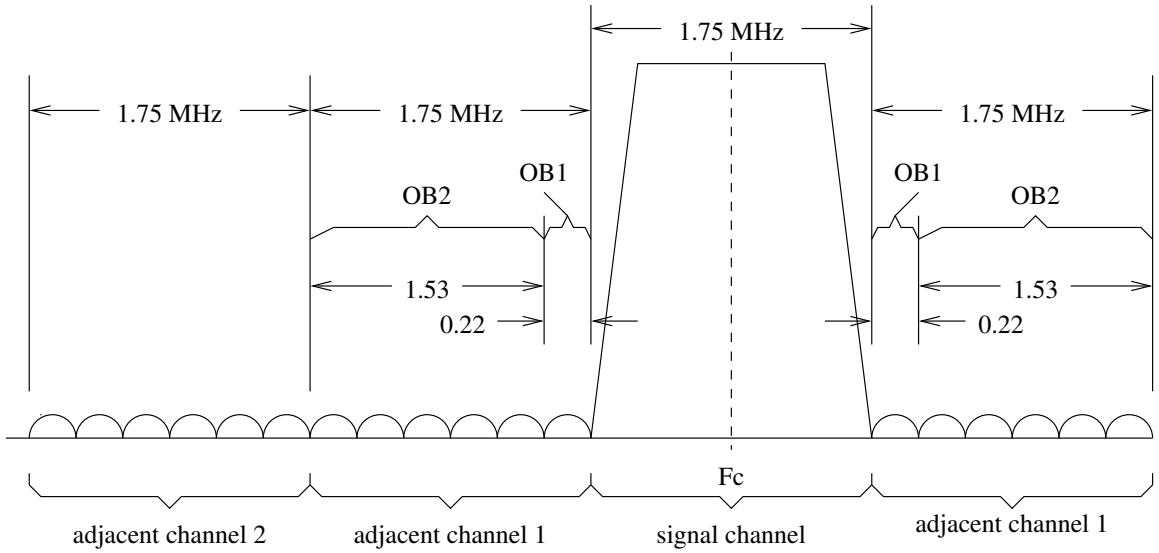

Figure 2.3: Illustration of out-of-band emission channels.

3. The modulation must be 16-QAM.

4. The nominal roll-off factor for the pulse shaping is $\beta = 0.12$

5. The channel bandwidth at RF is $\frac{(1+\beta)}{\texttt{samples/symbol}} \times \texttt{sampling rate} = \frac{(1+0.12)}{4} \times 6.25\,\text{Msam/sec} = 1.75$ MHz. The channel bandwidth at baseband is 1/2 the RF bandwidth, which is $\text{BW}_{\text{baseband}} = 1.75/2 = 0.875\,\text{MHz}$

6. **Practical Pulse Shaping filter only:**
   The out of band attenuation must meet certain specifications in the bands defined in Figure 2.3.

   (a) The power transmitted in <u>either</u> of the two OB1 bands, which are the 220 kHz bands that border the signal channel, must be at least 58 dB below the power in the signal. The signal power is the total power transmitted in the signal channel.

   (b) The power transmitted in <u>either</u> of the two OB2 bands, which are the two 1.53 MHz bands defined in Figure 2.3, must be at least 60 dB below the power in the signal.

   (c) The power transmitted in <u>either</u> of the two 1.75 MHz bands labelled adjacent channel 2 in Figure 2.3, must be at least 63 dB below the power in the signal.

7. **For the Gold Standard pulse shaping filter only:**
   In the absence of channel distortion and AWGN, the MER (modulation error ratio) must be greater than 50 dB when the gold standards for both the pulse shaping filter and matched filter are used. The MER is defined by

$$\text{MER} = \lim_{N\to\infty} 10\,\log_{10}\left(\frac{\sum_{n=-N}^{N}(I^2[n]+Q^2[n])}{\sum_{n=-N}^{N}[(I[n]-\widehat{I}[n])^2+(Q[n]-\widehat{Q}[n])^2}\right),$$

where $I[n]$ and $Q[n]$ are the ideal values of the decision variable and $\widehat{I}[n]$ and $\widehat{Q}[n]$ are the actual values of the decision variable.

The gold standard pulse shaping filter is not used in the final design so implementation cost is not an issue. However, since there are a limited number of multipliers on the FPGA the gold standard pulse shaping filter should be designed for multiplierless implementation, but the use of 10 18x18 multipliers is acceptable.

8. **For the Practical pulse shaping filter only:**
   In the absence of channel distortion and AWGN, the MER (modulation error ratio) must be greater than 40 dB when the the practical pulse shaping filter is paired with the gold standard matched filter.

9. The number of 18x18 multipliers used in the implementation of the practical pulse shaping filter should be zero, but must be less than 10.

4.2 *Specifications of the Gold Standard Matched Filter*

The gold standard for the matched filter is to be a rectangularly windowed (i.e. truncated) square root raised cosine filter that when paired directly with the Gold Standard pulse shaping filter yields an MER of 50 dB.

The gold standard matched filter should be designed so that the number of multipliers used in the implementation is as small as possible. While no specific target is provided, achieving a lower implementation cost will result in a higher mark for this item. Strategies for reducing the implementation cost of this filter will be discussed in class.

5 Deliverable 4

Deliverable 4 is the up-sampler and up-converter in the transmitter and the down-converter and down-sampler in the receiver. The system block diagram is shown in Figure 2.4. The transmitter is shown encased in a dashed box at the top of the Figure. The transmitter has an in-phase channel (I-Channel) and a quadrature phase channel (Q-Channel). The up-sampler is illustrated in both as a zero stuffing up-sampler followed by a low pass filter. The up-conversion for the I-Channel is accomplished by multiplication by the cosine output of `NCO_1` while the up-conversion for the Q-Channel is accomplished by multiplication by the sine output of `NCO_1`. For this design, the carrier frequency of the signal channel is to be set at $F_c = 6.25$ MHz which means `NCO_1` generates sinusoids at frequency of 6.25 MHz

The model for the channel is shown encased in the dashed box in the middle of Figure 2.4. The channel does three things:

1. Introduces adjacent channels, which of course cause interference.

2. Introduces AWGN, which is really introduced at the LNA in the receiver.

3. Introduces signal attenuation.

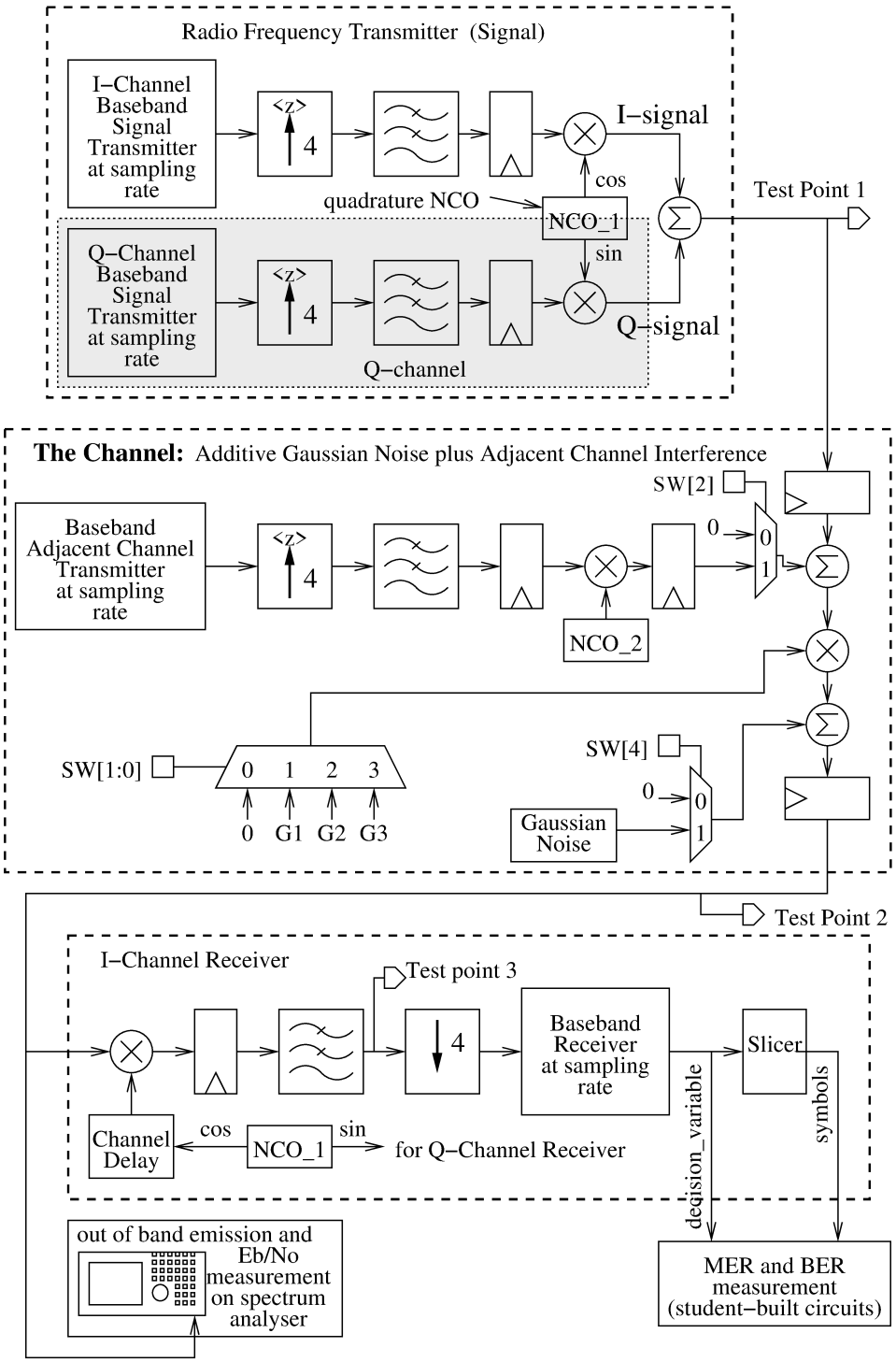

Figure 2.4: Block diagram showing the up/down sampling and up/down conversion.

The adjacent channel interference generator is simply another transmitter with the up-converter NCO set to an appropriate carrier frequency. The channel gain is modeled as one of four values: 0, G1, G2, or G3, which are selectable by switches SW[1:0].

The channel noise is introduced by a Gaussian noise generator supplied by the instructor.

The I-Channel receiver is shown in Figure 2.4 encased in the dashed box at the bottom of the Figure. Only the I-Channel Receiver is shown.

The specifications for the up-sampler/down sampler and up-converter/down-converter are given below:

1. In the absence of AWGN and channel distortion, with the practical pulse shaping filter and the gold standard matched filter employed in the transmitter and receiver, respectively, the process of up-sampling, up-converting, down-converting and down-sampling must not degrade the MER by more than 1 dB.

2. The total number of multipliers is limited to 14 or less.

A practical matched filter needs to be designed to suppress the adjacent channel. The adjacent channel is not suppressed by the filter that precedes the down-sampler. This means the practical matched filter should probably be the same as the pulse shaping filter, which had a sharp transition band. As with the pulse shaping filter the coefficients should probably be windowed to minimize the filter length.

Depending on the year and how the students are doing the design of a practical matched filter will usually be omitted as the students have already designed the very similar pulse shaping filter. If the practical matched filter is omitted, then it makes no sense to include the adjacent channel interference. Therefore if the practical matched filter is not designed the channel can be constructed without the adjacent channel interference.

Testing:

1. The Gaussian noise generator is supplied by the instructors. It has an 18-bit signed output that can be any signed number format. Whatever number format is chosen, the average power is 24 dB below full scale. That is, the power in the digital noise in units dBV is

$$P_{\text{noise\_total\_in\_dBV}} = P_{\text{full\_scale\_in\_dBV}} - 24 \text{ dB},$$

where

$$P_{\text{full\_scale\_in\_dBV}} = 10 \log \left(\frac{s^2}{1 \text{ V}^2} \right),$$

where s is the worth of the most significant bit.

The power spectral density of the noise at the output of the DAC in W/Hz is given by

$$N_{0\_\text{in\_W/Hz}} = \frac{P_{\text{noise\_total\_in\_Watts}}}{F_s/2 \text{ cycles/sample}} \times \frac{\sin(\pi F/F_s)}{\pi F/F_s},$$

where F_s is the sampling rate in units samples/second and $\sin(\pi F/F_s)/(\pi F/F_s)$ is the

DAC roll-off. The power spectral density in dBm/Hz is given by

$$\begin{aligned} N_{0\_\text{in}\_\text{dBm/Hz}} &= 10\log\left(\frac{\left(\frac{P_{\text{noise}\_\text{total}\_\text{in}\_\text{Watts}}}{F_s/2 \text{ cycles/sample}}\right)}{1 \text{ mW/Hz}} \times \frac{\sin(\pi F/F_s)}{\pi F/F_s}\right), \\ &= 10\log\left(\frac{P_{\text{noise}\_\text{total}\_\text{in}\_\text{watts}}}{1 \text{ mW}}\right) - 10\log\left(\frac{F_s/2 \text{ cycles/sample}}{\text{Hz}}\right) \\ &\qquad + 10\log\left(\frac{\sin(\pi F/F_s)}{\pi F/F_s}\right) \\ &= P_{\text{noise}\_\text{total}\_\text{in}\_\text{dBm}} - 10\log\left(\frac{F_s/2 \text{ cycles/sample}}{\text{Hz}}\right) + 10\log\left(\frac{\sin(\pi F/F_s)}{\pi F/F_s}\right) \end{aligned}$$

2. The signal-to-noise ratio (SNR) is measured at RF. This is accomplished by measuring the powers in the signal and noise separately and then taking the ratio. The signal power is measured from $F_c-(1+\beta)R_s/2$ to $F_c+(1+\beta)R_s/2$ with the noise turned off, where F_c is the frequency of the carrier, i.e., the center of the signal's spectrum, and R_s is the symbol rate. The noise power is measured across the *noise bandwidth* of the pulse shaping/matched filter with the signal turned off. This translates to measuring the power in the noise from $F_c - R_s/2$ to $F_c + R_s/2$ with the signal turned off.

   Both the power in the signal and the power in the noise can be measured on the spectrum analyzer using the marker function "Band interval power".

   The SNR is calculated as follows. First, the power in the signal, denoted P_s, is equal to the energy per symbol, denoted E_s times the symbol rate, denoted R_s. Second, the power in the noise is equal to the noise power spectral density, denoted $N_{0\_in\_W/Hz}$, times the noise bandwidth, which is R_s cycles/symbol. It then follows that the SNR is given by

$$\text{SNR} = \frac{P_s}{P_n} = \frac{E_s\, R_s}{N_{0\_\text{in}\_\text{W/Hz}}\, R_s \text{ cycles/symbol}} = \frac{E_s \text{ symbols/cycle}}{N_{0\_\text{in}\_\text{W/Hz}}}$$

3. The three values of attenuation/gain in the transmitter must be chosen to achieve three specific values of E_b/N_0: 7.88 dB, 10.52 dB, 12.20 dB. In linear scale the equivalent values are 6.14, 11.28, 16.61. With correctly chosen values, assuming Gray code mapping, the probabilities of error should be 10^{-2}, 10^{-3} and 10^{-4}, respectively.

   The attenuation/gain that achieves a specified value of E_b/N_0 can be calculated using the equations above. Of course E_s can be converted to E_b by $E_b = E_s/4$ if the RF signal contains both the I- and Q-signals or by by $E_b = E_s/2$ if the RF signal contains only the I-signal.

4. The bit error rate can be measured in a few different ways. There is a fairly simple way that does not involve accessing the symbols sent by the transmitter. Obviously, accessing the symbols in the transmitter, delaying them and comparing them to the

symbols received is impossible to do in practice. A circuit that measures BER without a "data connection" between the transmitter and receiver is illustrated in Figure 2.5.

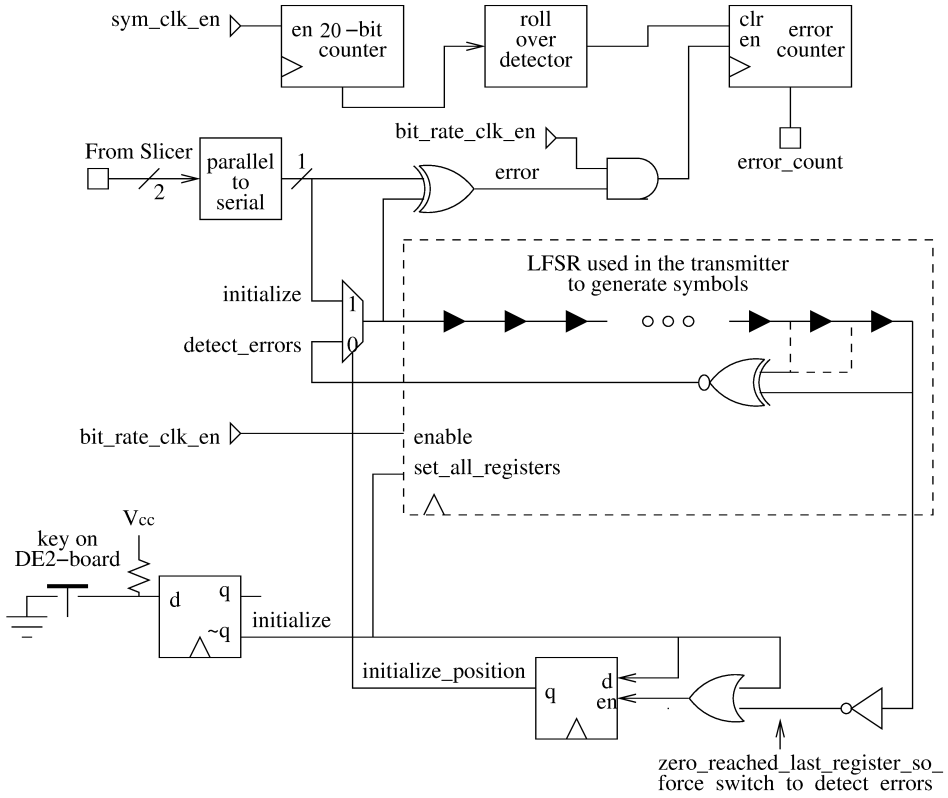

Figure 2.5: A simple circuit to calculate BER.

The circuit of Figure 2.5 is a bit tricky in that it runs at the bit rate, which is twice the symbol rate and half the sampling rate. It also requires the LFSR used in the transmitter be identical to the one shown in Figure 2.5. That is to say the serialized data leaving the parallel-to-serial converter in Figure 2.5 must be a delayed version of (but otherwise identical to) the output of the LFSR that generates the symbols in the transmitter's LFSR. The bottom line is the LFSR in the transmitter has to be modified to run at half the sampling rate (which is twice the symbol rate).

The principle of operation is quite straight forward. Once timing is found and the decision variable is identified, the momentary switch shown in Figure 2.5 is pressed to put the error measurement circuit in motion. Pressing the momentary switch sets the "initialize" flip/flop, which positions the data selector at the input to the LFSR to select the serialized data from the slicer. At the same time it sets all the registers in the LFSR to "1". When the momentary switch is released the serialized data from the slicer is loaded into the LFSR until the "initialize" flip/flop is cleared. When the "initialize" flip/flop is cleared the data selector switches to select the feedback network.

The parallel-to-serial converter takes the 2-bit symbols from the slicer, which obviously runs at the symbol rate, and produces a single bit output at the bit rate, which is

obviously twice the symbol rate. Once the receiver's LFSR is filled, assuming no error is the data, it will contain a pattern identical to one that was in transmitter's LFSR a little while ago.

As previously mentioned, pressing the momentary switch sets all registers in the LFSR to "1". This is done so that when a "0" finally reaches the last register in the chain of shift registers it acts as a flag to indicate that LFSR is filled with data. When a "0" finally reaches the last register in the LFSR, the "initialize" flip/flop is cleared and the MUX at the input to the LFSR changes to select the feedback network. The LFSR now functions as an LFSR identical to the one in the transmitter that is synchronized to the serialized data.

The output of the now free-running LFSR is compared to the serialized data coming from the slicer with an exclusive-OR. If the output of the exclusive-OR is "1" an error has occurred. The errors are accumulated in a counter that runs at the bit rate. The "error counter" is periodically reset with period 2^{20} symbols. The error rate is obtained by reading `error_count` in signal tap just prior to it being cleared and then dividing its value by 2×2^{20} to get the bit error rate.

5. Measure E_b/N_0 and the corresponding BER. Compare with the theoretical calculation of the BER according to $P[\text{bit error}] = \frac{3}{4}Q\left(\sqrt{0.8\frac{E_b}{N_0}}\right)$. *Perform these tests both with and without the adjacent channel enabled.*

6. Measure the MER at different settings of E_b/N_0, one of which is in the absence of noise, i.e., $E_b/N_0 = \infty$. *Perform these tests both with and without the adjacent channel enabled.*

6 Deliverable 5

Deliverable 5 is the system described by deliverable 4 with two additional features, both of which are for a packet-based system. The two additional features are timing recovery and slicer reference level recovery.

6.1 Timing Recovery

To add packet based timing recovery to the system requires the design of two circuits: a packet generator that resides in the transmitter and a timing recovery circuit that resides in the receiver.

Before a packet generator can be designed the structure/format of a packet must be defined. For the system designed here a packet consists of a 27 symbol preamble followed by an infinite length payload. Normally the payload would have a fixed length that would either be known *apriori* or would be transmitted in a special field in the preamble. For this class the packet ends when an external "reset" signal is made active. The "reset" is generated by pressing `key[3]` on the DE2-115 board.

The transmission of a packet begins with the transfer of the first symbol of the preamble, which is the first symbol in the cyclic prefix, to the mapper (or to the pulse shaping filter if

it is multiplier less). This transfer happens on the first positive clock edge of `sys_clk` that occurs after the reset signal goes low and `sym_clk_en` is active.

The 27-symbol preamble has three components: `cyclic_prefix`, `unique_word` and `cyclic_suffix`. The entire preamble is generated by a circuit described by the Verilog HDL below:

```
always @ (posedge sys_clk)
   if (reset == 1'b1)
       LFSR_4 = INITIAL_VALUE;  // INITIAL_VALUE is localparm
   else if (sym_clk_en == 1'b1)
       LFSR_4 = {LFSR_4[2:0], LFSR_4[3]^~LFSR_4[2]};
   else
       LFSR_4 = LFSR_4;

always @ *
   preamble_sequence = LFSR_4[3];
```

The local parameter `INITIAL_VALUE` is chosen so that `LFSR_4[3]` first generates the cyclic prefix, followed by the unique word, followed by the cyclic suffix. The preamble pattern to be generated is

```
preamble_sequence = suffix :   unique word    : prefix
preamble_sequence = 110111__0000_1010_0110_111__000010
                    |                                 |
                last sent                        first sent
```

The algorithm for switching from the preamble to the payload is described by the Verilog HDL below:

```
always @ (posedge sys_clk)
   if (reset == 1'b1)
       preamble_counter = 5'd0;
   else
       if (sym_clk_en == 1'b1)
          if (preamble_counter == 5'd27)
              preamble_counter = preamble_counter;
          else
              preamble_counter = preamble_counter + 5'd1;
       else
          preamble_counter = preamble_counter;

always @ (posedge sys_clk)
   if (sym_clk_en == 1'b1)
        if (preamble_counter == 5'd27)
            TX_symbol = random_data; // preamble finished so transmit payload
        else if (preamble_sequence == 1'b0)
            TX_symbol = MOST_POS_CONSTELLATION_POINT;
```

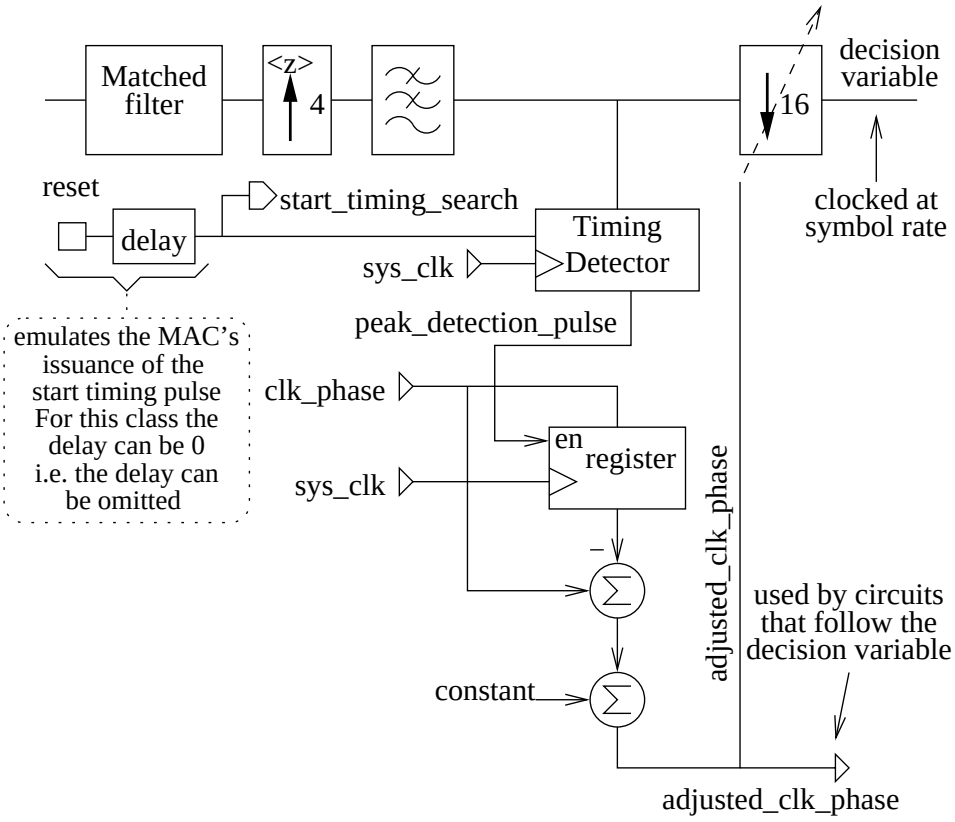

Figure 2.6: The architecture of a QAM receiver with timing recovery.

```
    else
        TX_symbol = MOST_NEG_CONSTELLATION_POINT;
else
    TX_symbol=TX_symbol;
```

There are many ways to design a timing recovery circuit. To ease the complexity of the task a receiver architecture that includes timing recovery is given in Figure 2.6.

To further ease the task of designing, building and testing a timing recovery circuit a block diagram for a timing recovery circuit is given in Figure 2.7.

There are two additional specifications that impact the timing recovery circuit.

1. A reset pulse of one system clock period duration is issued to the receiver. This pulse is a time critical pulse that would normally be generated by the MAC layer. The reset pulse is time critical in that it must coincide within tight limits of the arrival of the first symbol at the receiver. To be more specific, the reset pulse in the receiver occurs a fixed time after the "start of packet" reset pulse is issued in the transmitter. The time difference between pulses is the time from when a symbol enters the pulse shaping filter in the transmitter to the time the main tap of the response of the pulse shaping to that symbol reaches the receiver. This time depends on the length of the pulse shaping filter and the delay through the channel, which is unique to every customer.

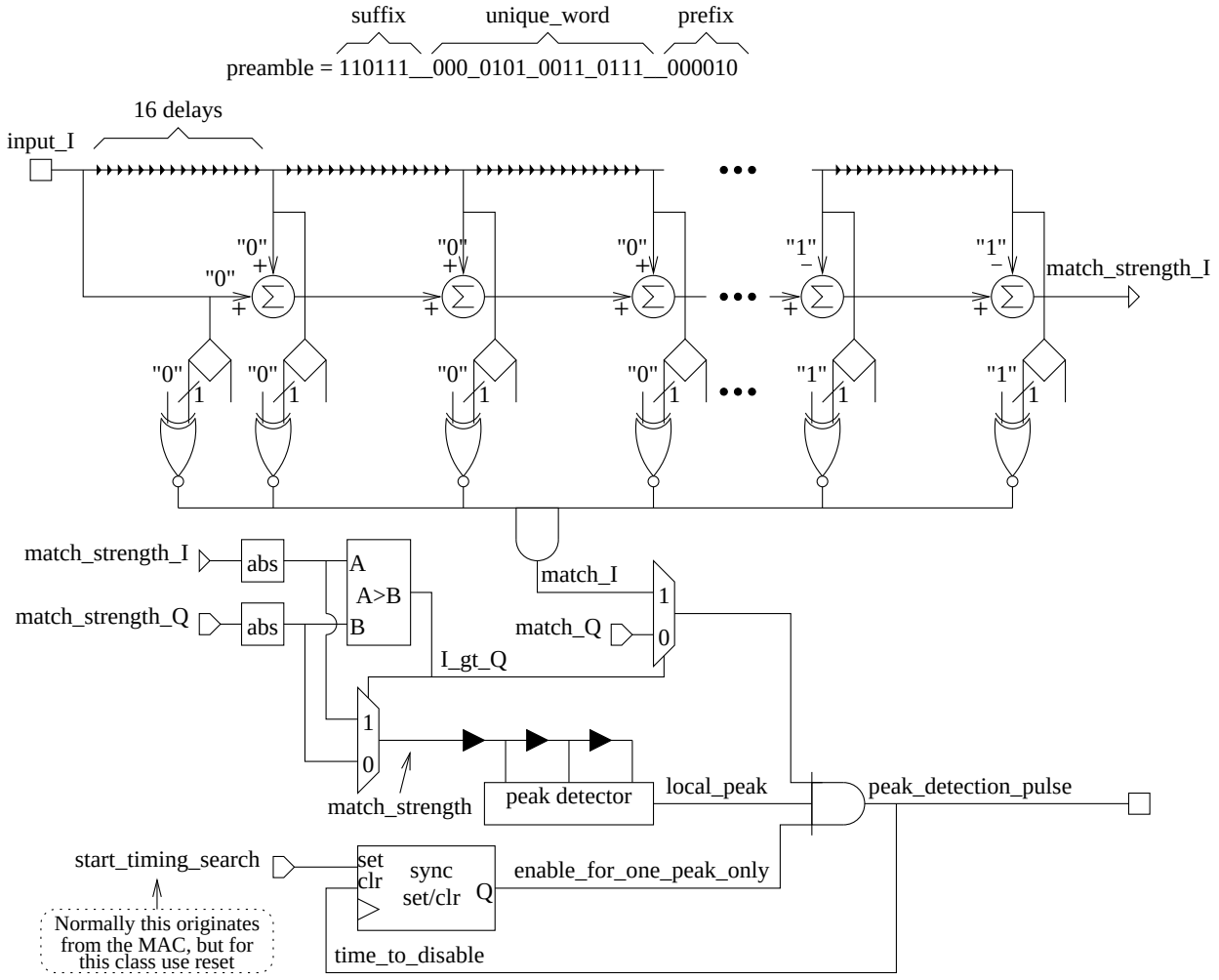

Figure 2.7: A block diagram of a unique word based timing recovery circuit.

The critical time is measured during a initialization process known as "ranging" and stored in the MAC layer.

For this project the reset pulse for the receiver will not be supplied by the MAC layer nor by the instructors. It will be designed and build along with the receiver by the student.

For this project the reset pulse (for the receiver) will coincide with the first symbol of the cyclic prefix reaching the timing recovery circuit.

The pulse width and tolerance on the time of issuance of the reset pulse in the receiver is specified by:

The pulse is a positive pulse with a duration of one system clock period.

The pulse is to be issued a fixed time after the "start of packet" pulse in the transmitter such that under conditions of no channel delay the negative edge of the pulse will go low sometime after (or coincident with) the time the main tap of the first symbol transmitted reaches the timing recovery circuit and before the second last symbol in cyclic prefix reaches the timing recovery circuit.

It is pointed out that during the testing in the lab exam a channel delay of any value up to 1 symbol will be inserted between the transmitter and receiver.

2. The timing recovery circuit must find the correct timing to within $\pm 1/32$ of a symbol time when there is no channel noise.

Note: The reset pulse arrives before the pipe line in the timing recovery circuit is filled with symbols. If the reset pulse is used for the reset in Figure 2.7 there will be a chance the unfilled pipeline will mimic the unique word and `peak_detection_pulse` **occur prematurely. Perhaps the reset in Figure 2.7 should be a delayed version of the reset pulse, i.e., delayed by 15 symbols.**

6.2 Slicer Reference Recovery

The reference level for the slicer must be recovered from the received data. In a packet based system a good estimate of the reference level must be recovered from the preamble (i.e. before the payload is received). Normally, the reference level is first estimated from during the preamble using the unique word and then continuously improved using the payload data. The initial estimate is normally based on the output of the peak detector, which depends on length of the unique word. The length of the unique word thereby determines the quality of the initial estimate of the reference level. The unique word is chosen long enough to give a reasonably good estimate, but, for reasons of efficiency, not so long that it gives a very good estimate. The error in the initial estimate is reduced over time using the data in the payload.

The decision variable, in the absence of noise, can be one of only 4 values: $\pm 1.5 v_{\text{ref}}$ or $\pm 0.5 v_{\text{ref}}$, where v_{ref} is the reference level for both the slicer and mapper. Only symbols that map to $\pm 1.5 v_{\text{ref}}$ are used in the preamble. Since there are 15 symbols in the unique word

and all symbols map to $\pm 1.5 v_{\text{ref}}$ the output of the peak detector is

$$\begin{aligned} v_{\text{pk\_det}} &= \sum_{n=0}^{14} \left(\frac{3}{2} v_{\text{ref}} + v_{\text{nse}}[n] \right) \\ &= \frac{45}{2} v_{\text{ref}} + \sum_{n=0}^{14} v_{\text{nse}}[n] \end{aligned}$$

where $v_{\text{pk\_det}}$ is the output of the peak detector and $v_{\text{nse}}[n]$ is the noise corrupting the n^{th} decision variable. An estimate of the decision variable is obtained by multiplying $v_{\text{pk\_det}}$ by $2/45$ to get

$$\widehat{v}_{\text{ref}} = v_{\text{ref}} + \frac{2}{45} \sum_{n=0}^{14} v_{\text{nse}}[n],$$

where $\widehat{v}_{\text{ref}}$ is an estimate of the correct reference level, which is v_{ref}. The noise corrupting $\widehat{v}_{\text{ref}}$ is $(2/45) \sum_{n=0}^{14} v_{\text{nse}}[n]$. It does not have a DC component, therefore its AC component has power

$$\begin{aligned} P_{\text{ref\_nse}} &= \overline{\left(\frac{2}{45} \sum_{n=0}^{14} v_{\text{nse}}[n] \right)^2} \\ &= \left(\frac{2}{45} \right)^2 \overline{\left(\sum_{n=0}^{14} v_{\text{nse}}[n] \right)^2} \\ &= \left(\frac{2}{45} \right)^2 \sum_{n=0}^{14} \overline{v_{\text{nse}}^2[n]} \\ &= \left(\frac{4}{2025} \right)^2 (15 \times \sigma_n^2) \\ &= 0.0296 \sigma_n^2 \end{aligned}$$

where $\sigma_n^2 = \overline{v_{\text{nse}}[n]^2}$ is the power in the noise corrupting the decision variable.

The data in the payload can be used to reduce the power in the noise corrupting $\widehat{v}_{\text{ref}}$ even further. There are, however, two significant differences between the preamble and the payload. The symbols in the preamble are known *apriori* where as the symbols in the payload are not and the symbols in the preamble are never mapped to constellation points as $\pm v_{\text{ref}}/2$ whereas the symbols in the payload are.

If the decision variable holds a constellation point at $\pm 0.5 v_{\text{ref}}$, then its absolute value has to be multiplied by 2 to get an estimate for v_{ref}. Whereas, if the decision variable holds a constellation point at $\pm 1.5 v_{\text{ref}}$, then the absolute value has to be multiplied by $1/1.5 = 2/3$ to get an estimate for v_{ref}. This means the constellation point must be ascertained before the decision variable can be used to estimate v_{ref}.

The constellation points for the payload can be ascertained using the reference $\widehat{v}_{\text{ref}}$ obtained in the preamble for the slicer. While $\widehat{v}_{\text{ref}}$ is not perfect it is good enough to categorize the symbols for purposes of improving the estimate $\widehat{v}_{\text{ref}}$ obtained in the preamble.

An interesting observation can be used to simplify the math. Summing the absolute values of two decision variables, where one represents a symbol mapped to $\pm 1.5 v_{\text{ref}}$ and the other represents a symbol mapped to $\pm 0.5 v_{\text{ref}}$ produces $((1.5 v_{\text{ref}} + v_{\text{nse}_1}) + (0.5 v_{\text{ref}} + v_{\text{nse}_2}))$, which is equivalent to $2 v_{\text{ref}} + v_{\text{nse}_1} + v_{\text{nse}_2}$. An estimate of the reference level is obtained by dividing the sum by two.

Using this approach an estimate for v_{ref} can be obtained by summing the absolute values of N decision variables taking care that exactly half of the N terms in the sum represent a constellation point at $\pm 0.5 v_{\text{ref}}$ and exactly half represent a constellation point at $\pm 1.5 v_{\text{ref}}$. Then v_{ref} is estimated by

$$\widehat{v}_{\text{ref}} = \frac{N v_{\text{ref}} + \sum_N v_{\text{nse}}[n]}{N},$$

where $\widehat{v}_{\text{ref}}$ is an estimate of v_{ref}. Choosing N to be a power of 2 makes the division a simple shift.

The power in the AC noise that corrupts $\widehat{v}_{\text{ref}}$ is given by

$$\begin{aligned} P_{\text{ref\_nse}} &= \overline{\left(\frac{\sum_N v_{\text{nse}}[n]}{N} \right)^2} = \frac{1}{N^2} \sum_N \overline{v_{\text{nse}}^2[n]} \\ &= \frac{1}{N^2} N \sigma_n^2 = \frac{\sigma_n^2}{N} \end{aligned}$$

Note that for $N = 32$ the noise power $0.0312 \sigma_n^2$, which is nearly the same as the power in the noise corrupting the initial estimate that was based on the symbols in the unique word.

A hardware efficient algorithm for recovering the reference level is given below:

Slicer Reference Level Recovery Algorithm

The initial estimate for the slicer reference is 2/45 of the peak value of the peak detector. This value is loaded into a register named `slicer_reference` that holds the current estimate of the slicer reference level. The register `slicer_reference` is also referred to a $\widehat{v}_{\text{ref}}$.

The slicer reference recovery circuit has four inputs: (1) a clock, (2) a reset pulse, (3) the absolute value of the decision variable, which is denoted $x_{\text{abs\_DV}}[n]$ and (4) a binary variable named $c[n]$, which is derived from $x_{\text{abs\_DV}}[n]$ as follows: if $x_{\text{abs\_DV}}[n] \geq \widehat{v}_{\text{ref}}$ then $c[n]$ is assigned a value of 1 otherwise it is assigned a value of 0.

The slicer reference recovery circuit has two outputs: (1) a clock enable named `clk_ena` and (2) an updated estimate of the reference level, which is named $\widehat{v}_{\text{ref\_out}}$.

The slicer reference recovery circuit has an accumulator named `ref_lvl_accumulator`, two counters named `counter_0` and `counter_1` and a register named `counter_milestone`.

The algorithm can be described by the logic that is applied to each new sample of the decision variable. That logic is described below:

1. On clock edges that occur while the reset is high:

   Load counters `counter_0` and `counter_1` with 16.

   Load `ref_lvl_accumulator` with 32 times $\widehat{v}_{\text{ref}}$.

   Load `counter_milestone` with 32.

2. If (`counter_0 == counter_milestone`) and (`counter_1==counter_milestone`) then

   Set `clk_ena = 1`.

   Assign $\widehat{v}_{\text{ref\_out}}$ the value in `ref_lvl_accumulator` divided by (2×`counter_milestone`).

   Load `counter_milestone` with 2×`counter_milestone`.

3. If ($c[n] == 0$) and (`counter_0`$\neq$`counter_milestone`) then add $x_{\text{abs\_DV}}[n]$ to `ref_lvl_accumulator` and increment `counter_0`.

4. If ($c[n] == 1$) and (`counter_1`$\neq$`counter_milestone`) then add $x_{\text{abs\_DV}}[n]$ to `ref_lvl_accumulator` and increment `counter_1`.

7 Deliverable 6

Deliverable 6 involves the construction of a series of circuits to add frequency and phase recovery capabilities to our receiver. The theory behind the problems of frequency and phase recovery is covered in detail in section 8 of this document.

The specific circuits that are to be designed, built, and integrated in the receiver are as follows:

- A circuit for estimating the frequency offset present at the timing recovery output based on the preamble.
- A circuit for estimating the phase offset present at the timing recovery output based on the preamble.
- A PLL for performing fine estimation and tracking of the phase and frequency offset.

A block diagram showing how these circuits will fit into the receiver is provided below.

Add receiver diagram

Additionally, in order to test your system, you will need to add a phase / frequency offset generation circuit to your transmitter. This circuit should be connected into the transmitter as shown Figure 2.8 below.

The phase and frequency offset generation circuit will be provided by the instructor as a Quartus partition (.qxp) file. The verilog module declaration is provided below to facilitate instantiation in the transmitter.

```
module tx_osc_error (
  input wire         clk,
  input wire         clk_en,
  input wire         reset,

  input wire  [31:0] phase,  // unsigned fraction, 32 bits
  input wire  [31:0] freq,   // unsigned fraction, 32 bits
  input wire  [17:0] i_in,   // 1s17
```

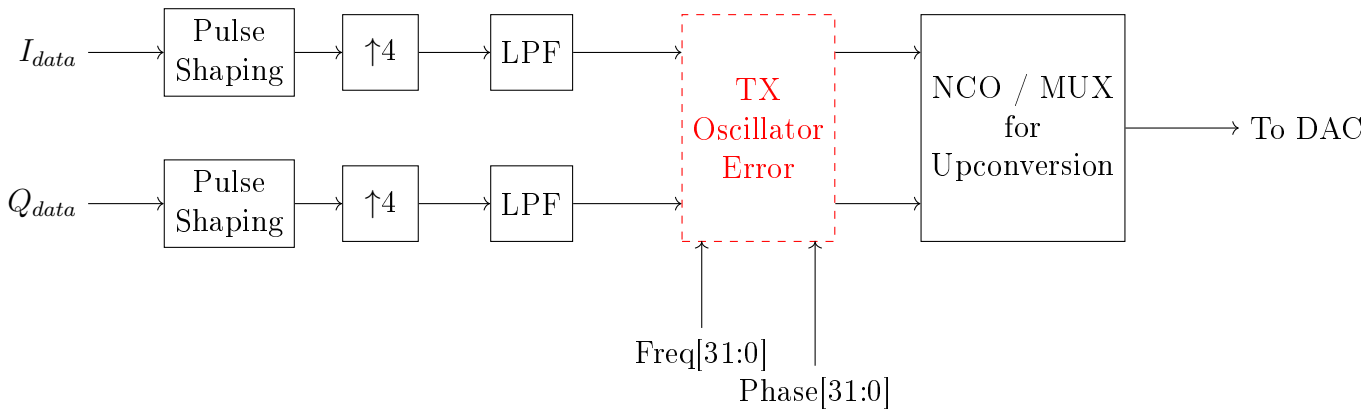

Figure 2.8: Connection diagam showing how phase / frequency offset generation circuit should be added to the transmitter.

```
  input wire  [17:0] q_in,   // 1s17

  output wire [17:0] i_out,  //1s17
  output wire [17:0] q_out   //1s17
);
```

Chapter 3

Lab Exams

```
Revised June 8, 2017, by Rory Gowen:
                       Added lab exams and instructions for kick start lab, Deliverable 1
                       and Deliverable 2.
Revised June 28, 2017, by Rory Gowen:
                     Added lab exam for Deliverable 5.
Revised Jan 5, 2018, by Brian Berscheid
                     Removed measurement of OOB emissions using SA from Deliverable 2
                     Removed baseband OOB emission measurements from Deliverable 3
                     Deliverable 3 OOB emission measurements done at RF
                       (black box upconverter provided to students)
```

1 Lab Exam for Kick Start Lab

1.1 Instructions For Kick Start Lab Exam

1. Print the exam sheet on the following page and bring it to the examination with you. Complete the steps listed below prior to coming to the scheduled lab exam time.

2. The following tasks pertain to Part A of the lab exam:

   (a) You will be asked to display the impulse response of your filter in Modelsim to the exam invigilator, be prepared to do so.

   (b) Fill in Table 3.1 with the peak value of the filter output for the two requested input sinusoids given in the table.

   (c) Instantiate the designed filter into a Quartus project that will place a swept frequency sinusoid as input to the filter. Compile the project and program the DE2 board. Use the signal analyzer in the lab to display the magnitude response of the designed filter. Show the results to the exam Invigilator(s) as requested.

3. The following tasks pertain to Part B of the lab exam:

   (a) You will be asked to display the impulse response of your filter in Modelsim to the exam invigilator, be prepared to do so.

(b) Display the worst case periodic signal to the exam Invigilator using Modelsim (i.e. show the repeating periodic signal samples in Modelsim).

(c) Display the output of the scaled filter in Modelsim when the input to the filter is the worst case periodic signal. Determine the peak value of the filter when this input signal is used and show it to the exam Invigilator (also record the value in Table 3.2).

(d) Fill in Table 3.2 with the peak value of the filter output for the two requested input sinusoids given in the table.

(e) For the lower frequency sinusoidal signal in the previous step determine the minimum headroom (i.e. ratio between the filter output peak value and the maximum value the filter can output without overflows) of the filter for the sinusoidal input. Record this value in 3.3.

(f) Instantiate the designed filter into a Quartus project that will place a swept frequency sinusoid as input to the filter. Compile the project and program the DE2 board. Use the signal analyzer in the lab to display the magnitude response of the designed filter. Show the results to the exam Invigilator(s) as requested.

Student Name (print): _____ (sign): _____

Table 3.1: Part A: Peak Output of Filter with Sinusoidal Input

Frequency	**Value**
$f_o = \frac{1}{16}$ cycles/sample	
f_{3dB}	

Table 3.2: Part B: Peak Output of Scaled Filter

Signal	**Value**
worst-case	
$f_o = \frac{1}{16}$ cycles/sample	
f_{3dB}	

Table 3.3: Part B: Headroom Calculation

Parameter	**Value**
$f_o = \frac{1}{16}$ cycles/sample	
Maximum filter output	
Headroom ratio	

Approximate number of hours spent on deliverable: ____

For Exam Invigilator Use: _____

- verify magnitude response of filter in part A: ____
- verify magnitude response of filter in part B: ____

2 Lab Exam for Deliverable 1

2.1 Lab Exam 1 Details

All items listed in **Deliverable 1** must be completed prior to entering the lab exam.

The exam will test the following items:

- The magnitude-frequency response of the TX filter.
- The magnitude-frequency response of the RCV filter.
- The MER of the TX and RCV filters connected in cascade.

Students are responsible for constructing a circuit or circuits to excite the TX and RCV filters in order to enable to measurements above. For example, these circuits may include a sinusoidal sweep generator and/or a pseudorandom data generator and/or a mapper and/or an impulse generator. Note that the stimulus generation requirements may be different for the TX filter containing multipliers and the multiplierless TX filter.

The magnitude response measurements should be performed in hardware using a spectrum analyzer, while the MER measurements should be performed in either Modelsim or Signal Tap (NOT using a spectrum analyzer). As discussed in class, some manual calculation will be necessary to generate MER readings from the raw measured data. (In deliverable 2, you will design a circuit to automate the MER measurement process.)

The exam will be graded using the following criteria:

- The TX filter meeting the stop-band attenuation specified/described in **Deliverable 1**.
- The TX and RCV filter using the number of coefficients specified in **Deliverable 1**.
- The RCV filter having the magnitude-frequency response specified in **Deliverable 1**.
- The TX filter's resource utilization (i.e. number of multipliers used for the multiplier based TX filter).
- The method used to determine coefficients for the TX filter and the method used to determine the theoretical MER of the cascaded filters.
- The method used to determine the practical MER of the cascaded TX and RCV filters.
- The comparison between theoretical (purely in Matlab) and practical (based on hardware measurements and calculations) MER values
- The quality of the TX filter will be gauged by MER that is achieved. For example, suppose filters from two students yield different MERs, the student whose filters yield the higher MER will receive a slightly higher grade.

Note: *It will be the responsibility of the Exam Invigilator(s) to verify the Student's results and determine the methods used during the filter design process for this exam.*

2.2 Instructions For Deliverable 1 Lab Exam

1. Print the exam sheet on the following page and bring it to the examination with you. Complete the steps listed below prior to coming to the scheduled lab exam time.

2. Fill in Table 3.4 with the values used when designing your TX filter as requested in Deliverable 1.

3. Fill in Table 3.5 with the theoretical value of the MER of your cascaded TX and RCV filters as determined in MATLAB. Also fill in the practical MER determined from measurements taken from the DE2 board. *You may be asked to explain how you determined the practical MER of your filters during the lab exam.*

4. Fill in Table 3.6 with the attenuation of the stop-band relative to the pass-band of the TX filter at the requested frequency in the table. Also write down the number of 18-bit by 18-bit multipliers used to create the practical TX filter. The number of multipliers used can be found in the resource usage section of the Quartus compilation report.

5. During the exam the Invigilator(s) will ask you questions about the design process and hardware implementations that you used. You may also be asked to demonstrate measurements that needed to be made when completing the above listed points.

Student Name (print): _____ (sign): _____

Table 3.4: Design Values for TX and RCV filters

Parameter	**TX filter**	**RCV filter**
f_{3dB}		$\frac{1}{8}$ cycles/sample
$r_{roll-off}$		0.25
β_{shape_factor}		0

Table 3.5: MER Values for Cascaded TX and RCV Filters

MER	**dB**
Theoretical	
Practical	

Table 3.6: Attenuation and Resource Usage of TX Filter

Parameter	**Value**
attenuation at 0.2 cycles/sample	
Multipliers used TX filter	

Approximate number of hours spent on deliverable: ____

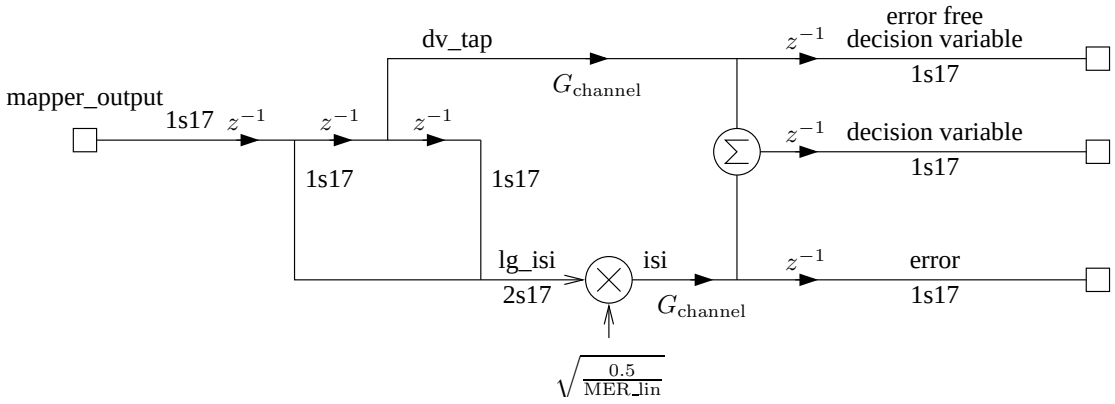

Figure 3.1: Schematic diagram of a SUT for the MER measurement in Deliverable 2

3 Lab Exam for Deliverable 2

Lab exam for Deliverable 2

The lab instructor has constructed a circuit that models all but the mapper of the system under test (SUT) shown in Figure 2.1. The schematic diagram of the SUT is shown in Figure 3.1.

The decision variable originates from the second last register in the chain shown in Figure 3.1, which is labeled `dv_tap` short for "decision variable tap". The ISI is derived from the registers neighbouring `dv_tap`: one register holds the next value for `dv_tap` and the other hold the past value of `dv_tap`. The signal named `lg_isi`, which is short for "large ISI" is the sum of the past and future values of `dv_tap`. Since the values in the sequence are independent, the average power in `lg_isi` is twice the power in `dv_tap`.

The large ISI, `lg_isi`, is scaled by $\sqrt{\frac{0.5}{\text{MER\_lin}}}$ to get `isi`. The power in `isi` is the power in `lg_isi` times $\left(\sqrt{\frac{0.5}{\text{MER\_lin}}}\right)^2$. That is

$$\begin{aligned} \text{P}_{\text{isi}} &= \text{P}_{\text{lg\_isi}} \times \frac{0.5}{\text{MER\_lin}} \\ &= \frac{\text{P}_{\text{dv\_tap}}}{\text{MER\_lin}} \end{aligned}$$

This means the ratio of the power in `dv_tap` to the power in `isi` is `MER_lin`.

Both `dv_tap` and `isi` undergo the same channel gain to become the error free decision variable and the error, respectively. This means the ratio of their powers do not change and the MER of decision variable is

$$\begin{aligned} \text{MER} &= 10 \log \left(\frac{\text{power in error free decision variable}}{\text{power in error}} \right) \\ &= 10 \log \left(\frac{\text{power in dv\_tap}}{\text{power in isi}} \right) \\ &= 10 \log \left(\text{MER\_lin} \right) \end{aligned}$$

The inputs and output of the model are described below:

1. The SUT model is driven by a 25 MHz system clock, `sys_clk`.

2. The SUT model has an ACTIVE LOW reset.

3. The SUT model uses `sym_clk_ena`, which must be provided.

4. The SUT model is for an ASK system. A QAM system has both in-phase and quadrature components so the circuit shown in Figure 3.1 must be instantiated twice to model a QAM channel.

5. The input to the SUT is the output of the mapper in 1s17 format. The mapper can use any constellation for 16-QAM, which is 4-ASK per channel, that fits into a 1s17 format without causing overflow. The model accepts the digital signal output of the mapper on the positive edge of `sys_clk` that is enabled by `sym_clk_ena`.

6. The error generated in the model is actually ISI from the two neighbouring symbols. Generating the error in this way makes its power proportional to the power in the constellation of the input.

   The taps in Figure 3.1 used to generate the ISI are first summed and then weighted by $\sqrt{\frac{0.5}{\text{MER}_{\text{linear}}}}$ so that the power in `isi` is $1/\text{MER}_{\text{linear}}$ times the power in `dv_tap`.

   Since `isi` and `dv_tap` are both multiplied by G_{channel} to become "error" and "error free decision variable", respectively, the MER of the decision variable is the ratio of the power in `dv_tap` to the power in `isi`, which is the MER used in the scaling factor $\sqrt{\frac{0.5}{\text{MER}_{\text{linear}}}}$.

7. The scaling factor $\sqrt{\frac{0.5}{\text{MER}_{\text{linear}}}}$ is controlled through the `isi_power` input to the provided Verilog module. This input scaling factor is treated as a 1s17 number by the Verilog code.

8. The decision variable is valid on the positive edge of `sys_clk` that is enabled by `sym_clk_ena`.

9. The reference level for the constellation at the output is G_{channel} times the reference level for the input constellation. This information should be useful for checking the circuit that calculates the reference for the output decision variable.

10. The model provides the error corrupting the decision variable on a separate output in 1s17 format. Again this is provided to help with debugging.

11. The provided Verilog code has a parameter `CHANNEL_GAIN`, which must be set to an integer (default = 1). This input controls the channel gain by performing a right shift by the specified number of bits. This results in an effective channel gain of $G_{\text{channel}} = 2^{-CHANNEL\_GAIN}$.

12. The MER of the output can be set to any value by changing the weights of the ISI taps in Figure 3.1.

13. The SUT model is designed to be instantiated in your prototype module and still work in Modelsim.

The Lab instructor will ask the student being examined to adjust the value of the multiplier coefficient in Figure 3.1 to one or more predetermined values and ask the students to measure the MER of the SUT using their MER circuit. *It is highly recommended that you connect the ISI scaling factor input to an "In-System Sources and Probes" module in your design so that the scaling can be modified in the FPGA without recompiling the entire design in Quartus.*

3.1 Instructions For Deliverable 2 Lab Exam

1. Print the exam sheet on the following page and bring it to the examination with you. Complete the steps listed below that you can prior to coming to the scheduled lab exam time.

2. Ensure that you have the following signals included in your SignalTap file so that the Invigilator may view them:

   - All clock signals *use the sys_ clk signal for Signal Tap.*
   - The output of your LFSR (i.e I and Q symbols).
   - The input values to the SUT being tested.
   - The reference levels for the I and Q decision variables.
   - The trigger for resetting the accumulators used in the MER circuit.
   - The symbol correct flag(s)
   - The accumulated error and the accumulated squared error values for both I and Q.
   - The MER in dB (if calculated in hardware - optional)

3. You will be asked to fill in Table 3.7 with the values obtained from your MER circuit. During the exam you will be asked to modify the multiplier value in Figure 3.1 in the SUT provided to you and measure the MER of the SUT after changing this value.

Student Name (print): _____ (sign): _____

Table 3.7: Values Obtained from the MER Circuit

Multiplier:	**I - Phase** value	**Q - Phase** value	**units**
reference_level			X'sdXXXXX
mapper_out_power			X'dXXXXX
accumulated_error			X'sdXXXXX
accumulated_squared_error			X'dXXXXX

Approximate number of hours spent on deliverable: ____

4 Lab Exam for Deliverable 3

4.1 Lab Exam 3 Details

While the out-of-band emissions performance of the pulse shaping filter is specified at RF (centered around a carrier frequency), the pulse shaping filter is designed and operates at baseband (centered around 0 Hz / DC). A complete transmitter includes an upconverter which translates the baseband signal to a carrier frequency. Once the signal is at RF, a spectrum analyzer can be used to measure the out-of-band emissions of the signal in a straightfoward fashion.

However, the upconverter and downconverter are not implemented until Deliverable 4. For this lab exam, the students will be provided with a black-box upconverter module that is to be instantiated in cascade with the designed pulse shaping filter. The output of this upconverter should be connected to the DAC so that the the out-of-band emissions generated by the pulse shaping filter can be measured in the typical fashion previously discussed.

Of course, the pulse shaping filter needs to be cascaded with the matched filter in order to measure the MER. Thus, in order to measure the MER and OOB performance simultaneously, the output of the pulse shaping filter should be split and sent to the upconverter and the matched filter in parallel. Your MER measurement circuit from Deliverable 2 should be used to take the MER measurement based on the matched filter output.

Prior to the lab exam, students should generate a programming file as described above which is capable of measuring the MER and the OOB emissions of their design. Additionally, any relevant intermediate signals that could help to verify the correct operation of the circuit should be either captured in Signal Tap or a Modelsim simulation.

4.2 Details of the Lab Exam for Deliverable 3

```
The Lab exam is generated by and the responsibitly of Mr. R. Gowen
who is the support engineer for the DSP stream.
```

The exam will be a show-and-tell type exam where the students demonstrate their working circuits to the instructor and answer questions asked by the instructor during the demonstration.

The grading is partitioned into two parts. The first concerns only the implementation cost. The second concerns the student's understanding of circuit and general knowledge of building and debugging circuits as well as the performance of the working circuit.

The students must complete the form below and present it to the instructor at the time of the show-and-tell demonstration. The information will be used by the examiners to help them ask meaningful questions.

Student Name (print): _____ (sign): _____

Implementation Costs - (3 marks)

1. Fill in the Table 3.8.

Table 3.8: Rough cost of Filters

Filter	**Number of Multipliers used**	**Filter length** (# coefficients)	β_{Kaiser_window}
Gold Standard - TX			$\beta = 0$
Gold Standard - RX			$\beta = 0$
Practical - TX			

2. During the exam you will be asked to show the resource usage in the compiler report to verify the numbers you listed in Table 3.8.

Knowledge and Performance of Filters - (12 marks)

1. Fill in the Table 3.9:

Table 3.9: Record of MER and out-of-band power

Filter	**MER** (dB)	**OB1** (dBc)	**OB2** (dBc)
Gold Standard		N/A	N/A
Practical			

2. Did you work on your design with other students or did you use modules written by other students? `yes` `no`

3. If the answer to question 2 was yes, what was your contribution (as a percentage) to the design? _____%

4. What mark (out of 15) do you feel you deserve for deliverable 3 under the assumption the class average is 12/15)? _____ /15

5. Answers to some questions asked during the exam may have to be written for reasons of confidentiality. Such answers are to be provided in the space below and on the reverse side of the page in necessary.

Approximate number of hours spent on deliverable: _____

5 Lab Exam for Deliverable 4

5.1 Pre-Exam Measurements for Deliverable 4

```
Created March 10, 2017, by Rory Gowen
Revised March 20, 2017, by Eric Salt/Rory Gowen
Revised March 21, 2017, by Rory Gowen
Revised March 11, 2018, by Rory Gowen
```

1. Print the blank form on page 137. Make the measurements described below and record the results in the appropriate table in the form that you printed. The completed form (all four tables) is to handed in to the examiner of deliverable 4 at the time of the exam. **Perform all measurements without the adjacent channel enabled. You may be asked to enable the adjacent channel and make measurements by the examiner.**

2. Measure the power in OB3 with respect to the carrier in units of dBc at two different test points: test point 1 denoted **tp1** and test point 2 denote **tp2** and record those measurements in Table 3.10. The location of the test points are shown in Figure 2.4.

   Also record the MER of the system measured (obviously measure it in the receiver) and record it in Table 3.10. Please set the channel gain as large as possible and turn off the AWGN prior to making the measurements.

3. Measure the bit error rates and calculate $\frac{E_b}{N_0}$ from measurements and record the results in Table 3.11. Please also provide the gains G_1, G_2 and G_3 that you used to get the specified E_b/N_o.

4. Measure the power in residuals of the images relative to the carrier power at test point 3 (i.e. **tp3**), which is located after the anti-aliasing, but prior to down-sampling as illustrated in Figure 2.4. Then record the largest of powers in the residual images in Table 3.12 under **t**p3.

   If you implemented a two stage down-sampler, down-sampling by two in each stage, then measure the residual power in the image at each stage. After making the measurements please record the results in Table 3.12 under **t**p3a and **t**p3b.

   Make sure the AWGN is turned off and the channel gain is set as large as possible.

5. Fill in Table 3.13 with the number of non-zero coefficients, registers and multipliers used to implement the down-sampler filter(s).
   If you used a single down-sampling stage only fill in the values for **filter 1**. If you used a two stage down-sampler fill in the values for both **filter 1** and **filter 2**.

Student Name (print): _____ (sign): _____

Table 3.10: First table for Deliverable 4

Parameter	**OB3** (dBc) @tp1	**OB3** (dBc) @tp2	**MER** (dB) no AWGN
Value			

Table 3.11: Second table for Deliverable 4

Gain / **parameter**	**0**	**G1:**	**G2:**	**G3:**
$\frac{E_b}{N_0}$				
BER				

Table 3.12: Third table for Deliverable 4

Parameter	**Carrier** (dB)	**image** (dBc) @tp3 or tp3a	**image** (dBc) @tp3b
Value			

Table 3.13: Fourth table for Deliverable 4

Filter	**Number of non-zero coefficients**	**Number of multipliers used**	**Number of registers used**
filter 1			
filter 2			

Approximate number of hours spent on deliverable: ____

6 Lab Exam for Deliverable 5

6.1 Pre-Exam Instructions for Deliverable 5

`Created June 28, 2017, by Rory Gowen`

1. Add an adjustable delay after the register located before `Test Point 2` in Figure 2.4 that can be varied between 0 to 15 sample delays of the channel sampling rate. Ensure that the delay can be adjusted by using switches or an instance of the `in-system probes and sources` mega-function in Quartus.

2. Add the following signals into SignalTap:

   - clk_phase
   - adjusted_clk_phase
   - MER
   - match_strength
   - peak_detection_pulse
   - reference_value

3. Set the adjustable delay to be 0 and trigger the timing recovery circuit by sending a preamble packet. Fill in Table 3.14 by adjusting the delay and filling in the values. **DO NOT RE-TRIGGER THE TIMING CIRCUIT FOR EACH DELAY VALUE! The table should be filled out after the timing has been recovered for the zero delay case in order to examine the effect of timing error on the MER of the system.**

4. Set the adjustable delay to the values required in Table 3.15 and **trigger the timing recovery circuit for each delay value** prior to filling in the table for that delay value.

5. If you have implemented the reference recovery portion of deliverable 5, fill in the values requested in Table 3.16.

Student Name (print): _____ (sign): _____

Table 3.14: First table for Deliverable 5

adjustable delay	0	1	2	15
clk_phase				
adjuseted_clk_phase				
difference between clock phases				
MER				

Table 3.15: Second table for Deliverable 5

adjustable delay	0	1	2	15
clk_phase				
adjuseted_clk_phase				
difference between clock phases				
MER				

Table 3.16: Third table for Deliverable 5

parameter	**signal format**	**value**
reference value (just after peak detect pulse)		
reference value (a long time after the peak detect pulse)		

Approximate number of hours spent on deliverable: ____