HØGSKOLEN I BERGEN

BERGEN UNIVERSITY COLLEGE

ASYNCHRONOUS CLOCK DOMAIN CROSSINGS IN DIGITAL DESIGNS

Magna Karina Morales Nordgard

HEEL12

June 2015

Document Control Sheet

Thesis Title: Metastability and Clock Domain Crossing in Digital Designs	*Date/Version* 30.05.15/v2.00
	Thesis Code: BOE15E-14
Author: Magna Karina Morales Nordgard	*Study:* HEEL12
	Total page number (incl. Appendixs): 59
Faculty Advisor: Johan Alme (Johan.Alme@hib.no)	*Confidentiality:* Public
Comments: I give Bergen University College permission to publish this thesis.	

Employing Corporation: ProgBit AS
Technical Advisor: *Hans Strømsøyen (hans@progbit.no)*

Acknowledgments

I would like to express my deepest gratitude to M.Sc Hans Strømsøyen, whose extensive guidance and patient advice made this project possible, and to Dr. Johan Alme, who has been a fantastic teacher and provided invaluable support and encouragement throughout the course of this project.

I would also like to extend my gratitude to M.Sc Rune Langøy and Dr. Jørgen Lien for their lessons and advice during my time at Buskerud and Vestfold University College. Their influence has been one of the chief inspirations to my academical achievements.

I must extend special thanks to my father, Magne Nordgard, for making it possible for me to come to Norway and for always helping me to the best of his ability. It is through his continued support I am able to pursue my dreams.

To my mother, Benita Morales, I owe so much that I would have to write another twelve thousand words to begin acknowledging her, but even if I did try to write it all down, it would not be enough. It is my hope that she can see for herself how much of the person I have become resembles her.

Thanks to my siblings Karla Calderón, Ronald Brenes and Mai Nordgard for their love and for bringing my nephews to this world; they are the stars of my life. Thanks to Øyvind Lye as well for giving me his unconditional support and for accompanying me in this journey.

Abstract

The phenomenon of metastability, which occurs due to failure to meet the timing constraints of a given register in a design, is an unavoidable hazard when designing asynchronous systems. Meanwhile, few electronic systems besides the most elementary can be designed using a single clock source.

The potential dangers of metastable events can be averted through efficient design methods. In clock domain crossings, where data is transferred from logic driven by one clock to logic driven by another clock, the preferable design method will vary in concordance with the frequencies of the clocks involved in the crossing, the nature of the data which must be transmitted, and whether or not it is crucial to capture all values of said data.

This project contains a review of various methods which can be used to tackle such situations according to the needs of the design; from simple synchronizers for one bit signals going from slow to fast clock domains, to asynchronous FIFO to transfer multibit signals reliably across domains without losing data.

To illustrate the potential danger of metastability, a system has been built which contains a circuit which deliberately causes metastability to occur. The expected number of metastable events, when letting the system run for 0.2 seconds, is a staggering 2202. In this particular circuit the frequencies of the clocks involved in the crossing are the same, but they have an inherent phase difference because the clocks originate from different sources, and metastable events occur in bursts when the clocks glide in and out of phase. Hence, the amount of metastable events provoked by the system illustrate the scope of the dormant probability of disaster should a designer fail to account for potential phase differences in their design.

Table of Contents

1 Introduction

1.1 Employing Company

ProgBit AS is a consulting company based in Bergen, Norway. The company was founded in April 2014 and is currently comprised of three partners with broad expertise and experience within the field of digital design and FPGA technology.

1.2 Problem Statement

Metastability is a phenomenon that can cause both structural failure and data corruption when dealing with field programmable gate arrays, or FPGAs. It occurs when the timing requirements of the registers in the FPGA are not met, producing undefined and invalid results for an unpredictable amount of time [1].

A design may display metastability if an asynchronous signal is used within the system. Such a signal might arrive at any time at the input of a given register, which means that its arrival might occur at a time that violates the timing requirements of the register. This signal might be a single asynchronous input in an otherwise synchronous system, or a signal crossing between two unrelated clock domains[1].

Metastability is unavoidable in systems presenting clock domain crossing paths (henceforth referred to as CDC-paths) [2], such as the one illustrated in figure 1. Here, DA is the CDC signal, while CLK A and CLK B are clocks originating from independent sources. This means that they do not present any fixed or precisely predictable phase relationship, making it impossible to create timing constraints that can help secure the correct transfer of data [1], thus, all values entering the second clock domain are perceived as asynchronous inputs.

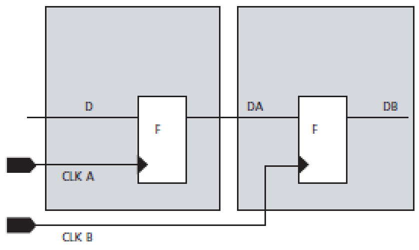


Figure 1 CDC Path

Increasing FPGA density, as well as the implementation of faster transistor technologies with lower operating voltages, set the conditions for growing probability for metastability in digital designs. [1] Meanwhile, the rising complexity of modern system-on-chip (SoC) designs challenges developers by making it necessary to operate different parts of the design at different clock rates. [3] Therefore, it is necessary for those pursuing a career in digital design to have good knowledge and understanding of the techniques that can help them make his or her system more robust against metastability failure when working with multiple clock domains.

[1] A synchronous clock domain consists of a section of logic driven by a clock, as well as sections of logic driven by other clocks that have been derived from the same source and have a fixed phase-relationship. [7]

1.3 Proposed Procedure

The purpose of this thesis project is to discuss known methods to handle metastability in CDC paths from a theoretical perspective, as well as deliberately causing the phenomenon to occur in a real system in such a manner that allows for some of the factors that affect the occurrence of metastability to be studied. Some of the theoretically outlined techniques will subsequently be tested.

The project will primarily serve an educational purpose to appease the concern raised by the employer that most newly educated junior electronics engineers are not acquainted in a sufficient manner, if at all, with the issues that arise from CDC crossings.

2 Assignment Specification

This assignment will introduce the phenomenon of metastability in flip-flop devices and explain why it is important for developers to consciously design to reduce the probability of its occurrence. It will focus on failures that occur due to clock domain crossings, and discuss the methods that decrease the mean time between failures in digital designs, namely flip-flop synchronizers, gray encoding and the use of asynchronous first-in-first-out (FIFO) registers.

Furthermore, a physical demonstrator will be developed on an FPGA board. Two different versions of the demonstrator will be developed:

- a simpler one, tailored for use in pedagogical context, which should clearly demonstrate metastability failure
- a slightly more complex one to study the methods that can be applied to decrease the probability of such failure occurring.

The simple version of the demonstrator will be created to be used in a class and should include a version of the test circuit that has been modified with synchronizing techniques, as well as a version of the test circuit with no synchronizing whatsoever. It should provide the means to compare the results from both circuits using software through a processor core. For this demonstrator, the input to the register that will be tested for metastability will be a simple toggled sequence.

Statistics showing the correlation between metastability failure rate and clock frequency will be generated from this demonstrator with help of the processor core. As a starting point, the demonstrator will be implemented for clock domains whose frequency is similar but uncorrelated. This frequency should be the highest possible frequency feasible for the design and the selected FPGA platform. It will subsequently be possible to expand the project by studying the cases where the clock frequencies of the various clock domains are very different. The clock frequencies will be set using phase-locked loops (PLL), which will be configured using the processor core as well.

If possible, statistics showing the correlation between temperature changes and metastability failure rate are also desirable. This is however dependent on the availability of suitable equipment to conduct such tests, which will be discussed in next chapter.

The more complex version of the demonstrator will use a pseudo-random bit-sequence (PRBS) generator at the originating clock domain and a PRBS checker following the crossing, which will allow for data hazards to be observed. The employer has offered to provide the code for the PRBS generator and checker.

This setup will be used to test at least two possible approaches to clock domain crossing. These should exemplify both good solutions and less ideal solutions. Two approaches have been proposed:

- a case where data transfer occurs at a moderate rate and is aided by a control signal
- a case with faster data transfer rate, aided by both a control signal and an asynchronous FIFO register

As a stretch goal, an IEEE-format paper condensing the information collected will be written and submitted for publishing in an academic journal or to a conference given that the results are both satisfactory and sufficient in terms of value of the data gathered during the execution of the project.

3 Project Analysis

3.1 Literature Review Methodology

A brief literature review must be conducted to discuss flip-flop metastability and to give a theoretical outline of the approaches that can be used to diminish the probability of metastability failure occurring in digital designs. A traditional approach will be used, meaning that a relevant body of literature will be selected, summarized and synthetized to provide the reader with the desired amount insight in this topics.

3.2 Demonstrator Design

In chapter 2, it was specified that both a simple and a more advanced demonstrator would be developed for different purposes. From now on, the simple demonstrator will be referred to as Type A Demonstrator, whereas the more complex one will be referred to as Type B demonstrator.

Both types have in common that they will be developed using the same SoC platform and development tools, and will require design and integration of both hardware and software.

3.2.1 SoC-platform, Design Tools and Equipment

The demonstrators will be designed for TerASIC's DE2-115 development board, which features a Cyclone IV-E FPGA from Altera [4]. Designing for this board will require access to Altera's design software tool, Quartus II. This tool is available in two editions, one licensed subscription edition and a free web edition. The project doesn't require any of the enhancements or features available only for the subscription edition, and thus the Web Edition will suffice.

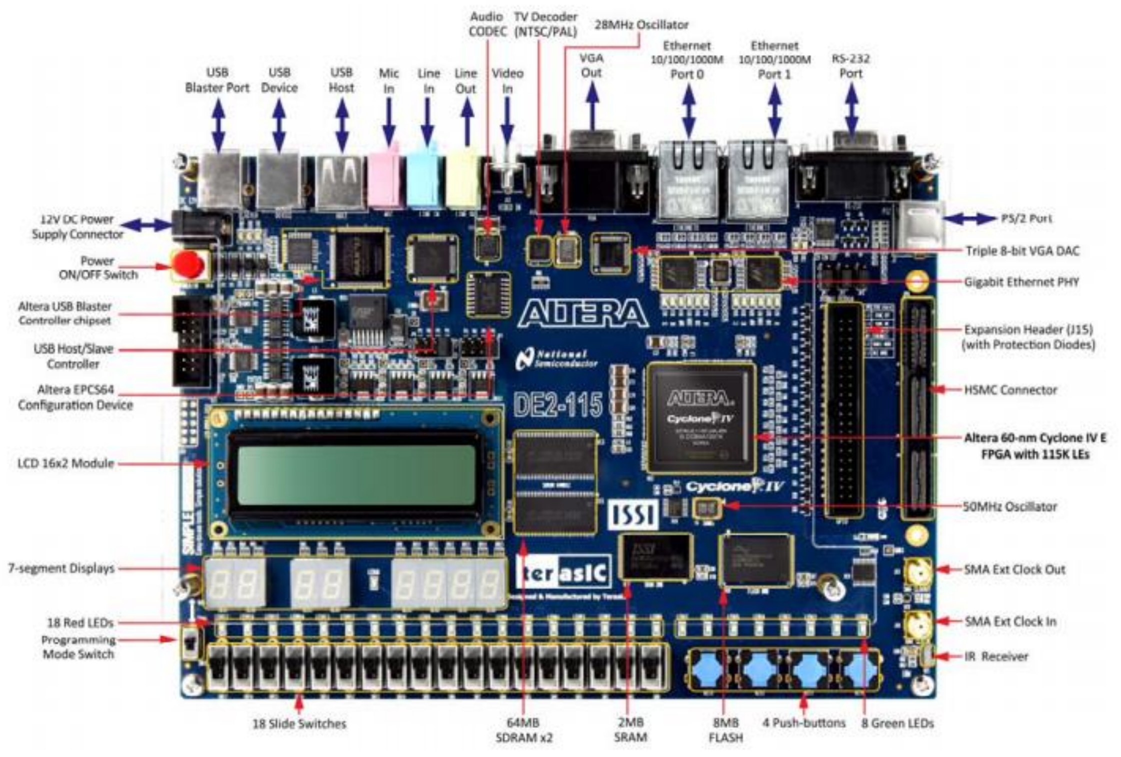


Figure 2 The DE2-115 Development Board

The DE1-SoC development board (also from TerASIC) was briefly considered for the project. This board features an Altera Cyclone V FPGA and a dual-core ARM Cortex-A9 Processor. The author has a special interest for ARM processor architecture, and since the project specification calls for a processor core, the feature made it appealing to choose the DE1-SoC board over the DE2-115.

The DE2-115 board doesn't feature a hard processor core like the DE1-SoC board, yet for the purposes of this project using the Altera-developed NIOS II soft processor is more than sufficient. Design tools for the NIOS II processor are supported in the web edition of Quartus II, which the author is acquainted with, as opposed to the design tools for hard ARM processor of the DE1-SoC board.

Another sensible reason to choose the DE2-115 board over the DE1-SoC board is that, in the classes in which the demonstrator Type A might be used, the students will have DE2-115 boards and not DE1-SoC boards. Since it is desirable that the Type A demonstrator is developed in a way that allows for laboratory exercises to be derived, it should be made for the available platform. However, this would not impede for the Type B demonstrator to be implemented for the DE1-SoC board.

Neither board contains more than one independent clock source, i.e. all internal clocks originate from the same oscillator crystal have thus a fixed phase relationship. A critical specification for this project is to use two completely uncorrelated clock sources for the clock domains involved. Connectivity for an external clock source is therefore required.

The DE1-SoC board fails to qualify as a suitable platform for the project at this point. It is not possible to connect an external clock source to this board. However, the DE2-115 board has an SMA connector which can be used to drive an external clock into the board's clock circuitry. This input is labelled as SMA_CLKIN in the figure below, which shows the clock distribution in the DE-115 board. The input is also visible in the right-bottom corner of figure 2, where it is labelled as SMA Ext Clock In.

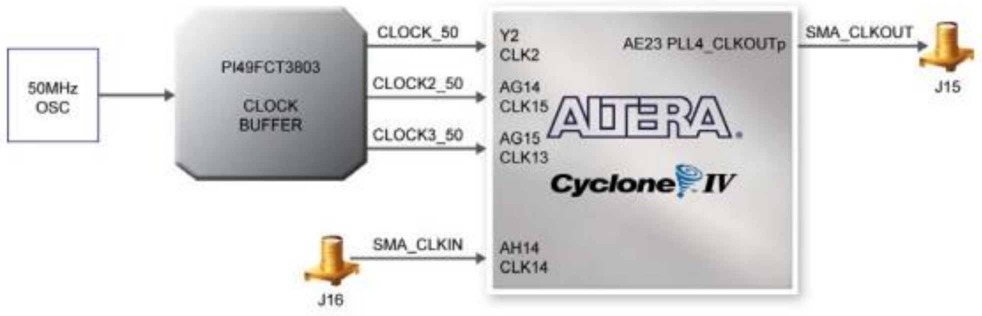


Figure 3 DE-115 Board Clock Distribution

A printed circuit board (PCB) has been ordered from Adafruit Industries featuring a Si5351A clock generator from Silicon Labs. This is a programmable clock generator that can generate frequencies between 8MHz and 160MHz. It is programmed through an Inter-Integrated-Cicuit (I2C) communication protocol. The desired frequency for the clock output is configured through this protocol by setting PLLs and clock dividers within the clock generator IC. This will be done from the NIOS II processor core.

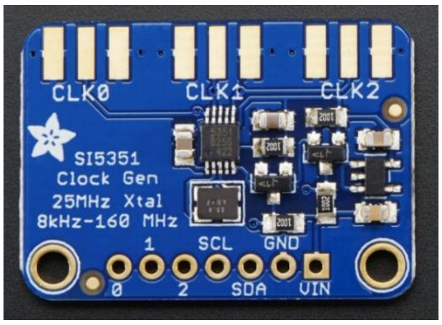


Figure 4 Clock Generator Breakout Board from Adafruit Industries [5]

1.6 mm thick SubMiniature-VersionA (SMA) edge launch connectors will be soldered to the topmost outputs of the clock generator PCB. These can then be connected to the clock input of the DE2-115 development board. This clock input is labelled as SMA_CLKIN in figure3.

The general form of the system as it is envisioned, regardless of the type of the demonstrator, is outlined below.

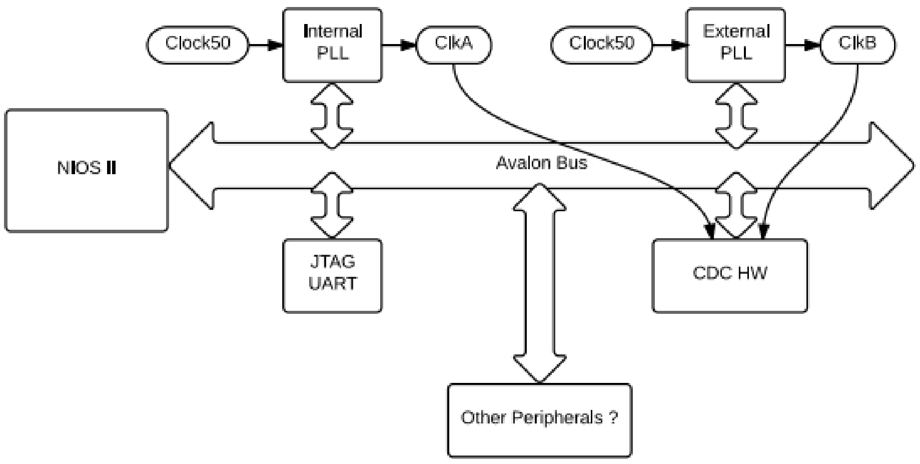


Figure 5 Overview of the System

The NIOS II processor[2] will be used to control the frequencies of both clock domains, dictated by ClkA and ClkB, controlling PLL through the Avalon Bus. The Avalon Bus is Altera's own bus protocol. For the internal clock of the development board, an internal PLL will be used. The external clock generator has its own PLL, which will be used to set the frequency of the external clock. The NIOS processor will also be connected to the designed hardware, which may be hardware displaying metastability at a clock domain crossing, or hardware designed to avoid metastability at said crossing. The joint-test action-group universal asynchronous receiver-transmitter (JTAG UART) will allow for the processor to display results in the console of the NIOS II. Other peripherals might be added to suit needs that might arise, such as displaying the results on the development board. For this purpose the LCD screen of the development board might be a suitable component to add to the system.

[2] For more information about the NIOS II processor, see chapter 5.4.

3.2.2 Demonstrator Type A

The starting point for the demonstrator type A was suggested by the employer and is the circuit shown below, which will be used to display metastability failure..

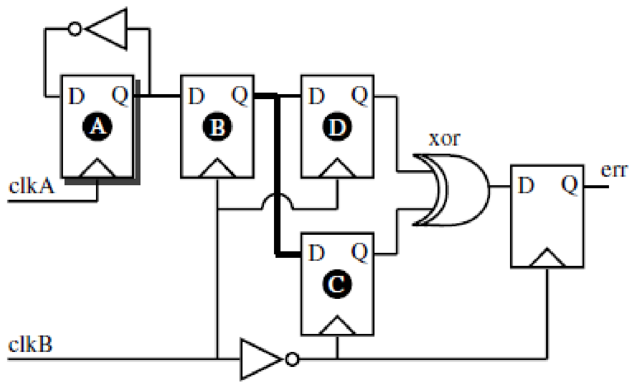


Figure 6 Test Circuit for Demonstrator Type A [1]

In this circuit, register B is the register under test, meaning that the circuit is designed to detect metastability occurring in this register. Metastability might occur elsewhere in the circuit, but this is not accounted for.

The input to register B comes from clock domain A, and changes every clock cycle. However, this register samples its input with a frequency dictated by clock B. Thus, the output of register A is asynchronous to the register that samples it, and represents a clock domain crossing.

The output of register B is sampled by two different registers, C and D. Sampling at the input of Register C has a delay of a half clock period relative to register D. These registers would sample different values should the output from register B become metastable. The exclusive-or circuit comparing the outputs from C and D would then detect that the registers sampled differing values and output a HIGH signal, which would indicate error. Discretizing and counting the amount of times the error signal goes high should allow for statistics regarding metastability to be elaborated.

In the assignment specification it was stated it is desirable to obtain statistics showing the correlation between metastability failure and frequency as well temperature from this demonstrator. To vary input frequencies, PLL can be used to set the frequencies of the clock domains. This should be easily achieved using the processor core given the envisioned system. Testing the relationship between metastability failure and temperature changes requires access to suitable temperature cabinets. It is known to the author that the university college is in possession of temperature cabinets, however whether or not they are available for use at the moment is unknown. Knowledge of this is however not critical at the moment given the time constraints of the project; it is not yet decided whether or not it will be possible to fulfil this goal.

3.2.3 Demonstrator Type B

The goal of this demonstrator type is to show CDC methods in practice. Two approaches were suggested by the employer for this purpose:

- Approach 1: Data transfer between clock domains at a moderate rate. Transfer aided by control signals.
- Approach 2: Data transfer between clock domains at higher rate. Transfer aided by control signals and asynchronous FIFO.

The form these approaches would take in hardware has not been outlined yet. However it has been specified that the data to be transferred across clock domains should be generated by a PRBS generator and checked at the other side of the crossing by the corresponding checker, as shown in the figure below.

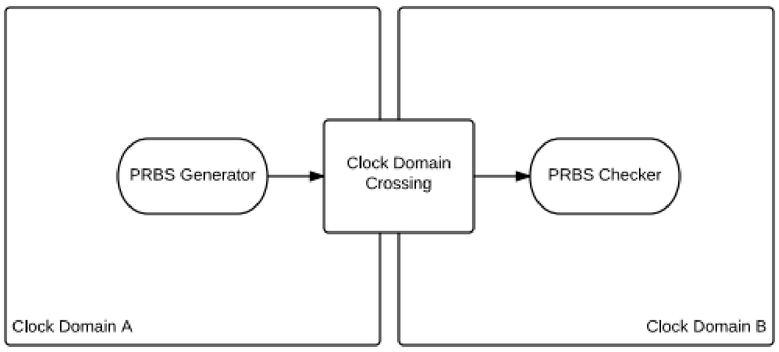


Figure 7 General outline of Demonstrator Type B

4 Theoretical Background

4.1 Metastability and Asynchronous Clock Domain Crossings

Metastability is, as stated in section 1.3, a phenomenon that occurs due to failure to meet the timing constrains of the flip-flip flops used to implement the design, which leads to undesired and unpredictable output behaviour, where the output signal does not assume a stable logic value for an unbounded amount of time.

During normal operation, a flip-flop copies the input D to the output Q on the active edge of the clock. Normal operation relies on the stability of the input signal D. To be regarded as stable, signal D must remain at a valid HIGH or LOW value for a fixed amount of time before and after the active edge of the clock driving the flip-flop, namely the *set-up* and *hold* times of the flip-flop. [6]

When dealing with designs that do not present asynchronous inputs, it is possible to constrain the input of every flip-flop, which in turn ensures the predetermined behaviour of the flip-flop outputs and the correct transfer of the data. However, when the design requires the handling of asynchronous signals, it is impossible to predict when the signal will transition relative to the active edge of the clock, meaning that timing constraints cannot be verified, and the transfer of data cannot be secured [1].

Metastability can have fatal consequences in a digital design: It might lead to different values of the same signal being read into different locations in the design, making the design behave in unpredictable ways. Furthermore, feeding unstable data to several places in a design is known in some cases to lead to high current flow; the worst case scenario being chip burnout [7] .

Asynchronous clock domain crossings occur when data is fed from a flip-flop driven by one clock to a flip-flop driven by another asynchronous clock. Asynchronous clocks originate from independent sources and have therefore no predictable or fixed phase-relationship. Metastability is unavoidable at CDC paths, but techniques can be employed to diminish the impact the phenomenon has in the design.

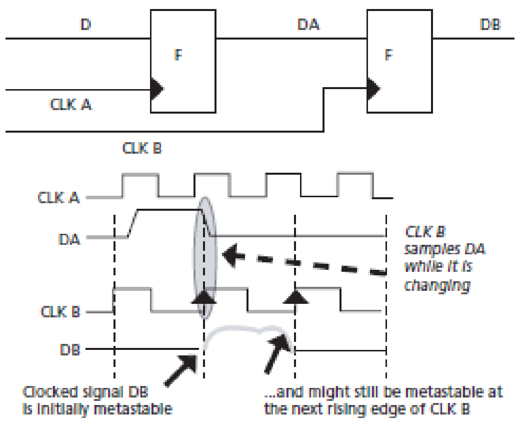

D
F
DA
F
DB
CLK A
CLK B
CLK A
DA
CLK B
DB
CLK B samples DA while it is changing
Clocked signal DB is initially metastable
...and might still be metastable at the next rising edge of CLK B

Figure 8 Metastability in a CDC

In the figure above, a clock domain crossing is depicted. The signal DA, originating from CLK A, is sampled by CLK B while it is changing, leading to metastability, and potential data loss in the second flip flop. In this case, the originating clock domain is slightly faster than the receiving clock domain. The relationship between the frequencies, as well as the nature of the signals we attempt to pass across domains, are the factors that we must consider when we choose how to secure the crossing.

4.2 Dealing with Clock Domain Crossings: Selected Methods

4.2.1 Transferring One-Bit Signals with Flip-Flop Synchronizers

According to Dally and Poulton [8], a synchronizer is a device that samples an asynchronous signal and outputs a version of the signal that has transitions synchronized to a local or sample clock.

When the originating clock domain is sufficiently slow related to the destination domain, two flip-flops in series might be used to synchronize single bits crossing the CDC path.

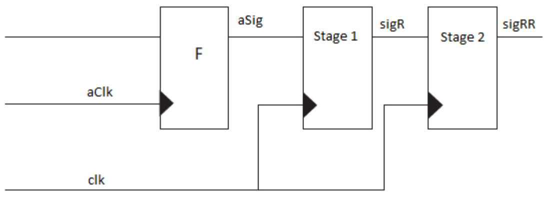


Figure 9 Two Flip-Flop Synchronizer

The purpose of the two flip-flop synchronizer is to buy time for metastable states to settle into a stable logic value [7] .

Regard figure 8. Signal *aSig* can transition anytime, which might lead to a violation of the set-up or hold timing requirements of the first flip-flop in the synchronizer chain (stage 1). Its output *sigR* will then become metastable. After a full clock-cycle, sigR is likely to have settled into a stable value, which is now sampled by the second flip-flop in the synchronizer. Thus, the output of the second flip-flop is a stable value, safe for further use in the receiving domain. In very high speed designs, two flip-flops might not provide enough time to ensure that the metastable state will resolve. In such cases, a third flip-flop might be added.

How do we know when clkA is sufficiently slow relative to clkb? A rule of thumb, known as the *three-edge requirement,* is that the clock period of the originating domain should be at least 1.5 times the length of the period of the destination domain [9]. Thus, a pulse from the slow domain should last enough for the faster clock to transition three times (therefrom the name three-edge requirement), ensuring that the signal can be safely and *correctly* captured by the synchronizer. Should this requirement not be fulfilled, then data might be lost, as shown in the figure below, where the period of aClk is only as long as that of clk. (Due to limitations in the waveform drawing software, it was not possible to draw an example where the period of aClk was slightly larger than that of clk yet still under a factor 1.5, however, the same situation might arise had aClk had $t_{aClk}=1.25*t_{clk}$).

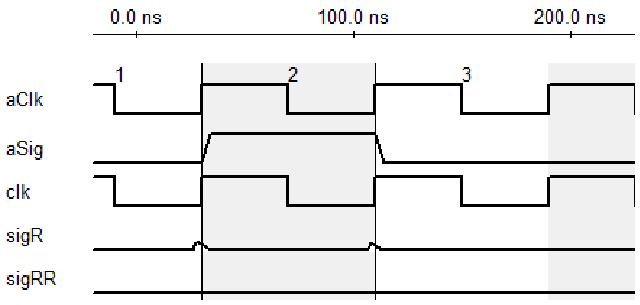


Figure 10 Data loss due to failure to meet the three-edge requirement

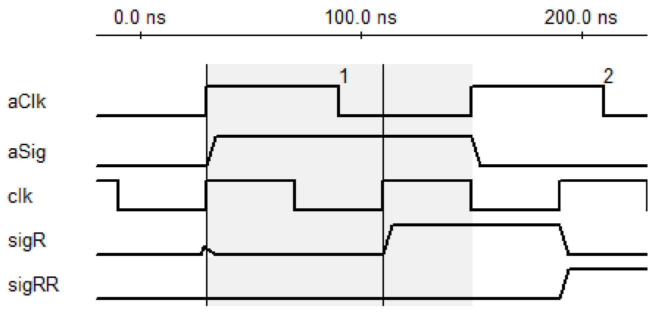
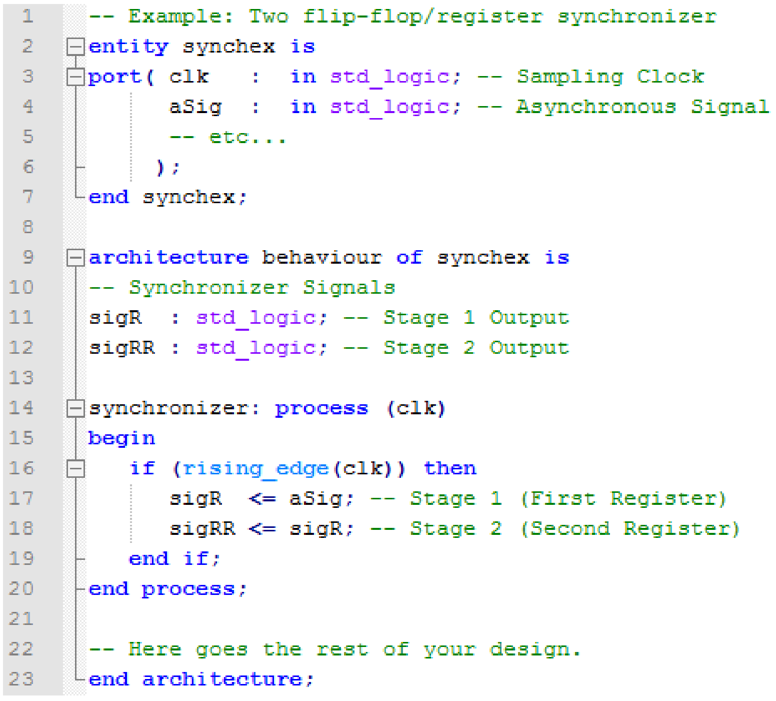


Figure 11 Data capture

Figure 11 shows an example where the first attempt to sample aSig results in metastability. However, since t_{aClk} (and thus the pulse aSig) lasts 1.5* t_{clk}, the three-edge requirement is fulfilled. The data is then correctly captured in the next active edge of clk, and propagated to the rest of the design.



```
-- Example: Two flip-flop/register synchronizer
entity synchex is
port( clk   :  in std_logic; -- Sampling Clock
      aSig  :  in std_logic; -- Asynchronous Signal
      -- etc...
     );
end synchex;

architecture behaviour of synchex is
-- Synchronizer Signals
sigR  : std_logic; -- Stage 1 Output
sigRR : std_logic; -- Stage 2 Output

synchronizer: process (clk)
begin
   if (rising_edge(clk)) then
      sigR  <= aSig; -- Stage 1 (First Register)
      sigRR <= sigR; -- Stage 2 (Second Register)
   end if;
end process;

-- Here goes the rest of your design.
end architecture;
```

Figure 12 Two Flip-Flop Synchronizer using VHDL

In figure 12, the VHDL code to implement a two register synchronizer is shown. Notice that the synchronizer process is both short and simple, and you should normally not use a separate module to implement this kind of synchronizer. Notice also that, as a naming convention, we append the letter “R” to the name of the output of stage 1 to indicate it has passed through one register. We append “RR” to the name of the output of stage 2 to indicate it has passed through both registers. Should we implement a synchronizer chain with more registers, we would increase the number of R’s we append to reflect the number of registers used.

It is possible to transfer one-bit signals from a fast clock domain to a slow clock domain using flip-flop synchronizers as described, provided that these signals are asserted long enough to satisfy the three-edge requirement. This is therefore only possible if the frequencies of the involved clock-domains are predefined, so that their relationship is used to determine the architecture of the logic used to assert the signals.

4.2.2 Transferring Multiple Bit Signals with Gray Encoding

When transferring signals composed of multiple bits across a CDC path, one must consider the importance of performing a synchronized sampling of the CDC bits. Attempting to transfer such signals by using separate flip-flop synchronizers for each bit in the signal might result in data incoherency [7]. You could run into a situation where some of the bits are captured at the first sampling attempt and others, by virtue of metastability, are not. Thus, the composed signal will be corrupt and data coherency lost. This situation is shown below.

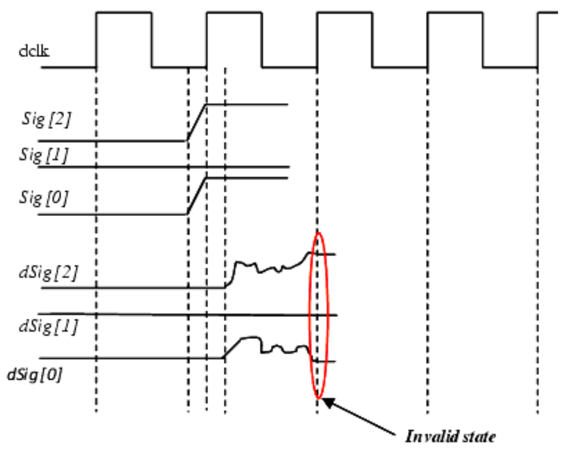


Figure 13 Data Incoherency [10]

In this figure, we see that signal *Sig* changes its value from 000 to 101 close to the sampling edge of dclk, leading to metastability in the flip-flops capturing the changing bits Sig[0] and Sig[1]. When the metastable state resolves, dSig[0] assumes an erroneous value, and the data is corrupted.

This problem would be remediated if the designer could guarantee that only one of the bits conforming the signal changed every clock cycle. In that case, if the first synchronizing flip-flop didn't initially register the transition, the output of the synchronizer would remain at the previous value until a correct signal value was captured.

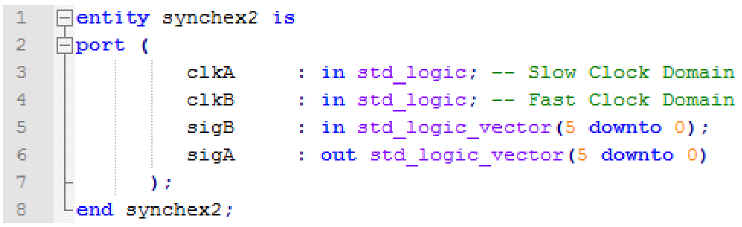
This can be achieved using gray encoding. Such an approach is suitable for control signals that count up or down (such as address buses), as gray code ensures that only a single bit is changed for every update of the signal.

4.2.3 Enabling the Sampling of Multiple Bit Signals using Synchronized Control Signals

Sometimes, we do not benefit from using gray code on multiple-bit signals across CDC paths. The originating clock domain might be much faster than the receiving domain, so that the receiving domain doesn't perceive the signal as changing one bit at a time. Neither are all multiple-bit signals of the kind that counts up or down, which is the requirement that lets us guarantee that no more than one bit will change whenever the value of the signal is updated.

In such cases, we might want to use control signals to allow for the receiving clock domain to sample the CDC signal at a time at which we know for certain that the signal will be stable.

An approach to tackling this kind of crossing will now be suggested, using a VHDL code example. The principle written in the previous paragraph is simple, however, many small factors must be taken in consideration when applying it to a design!

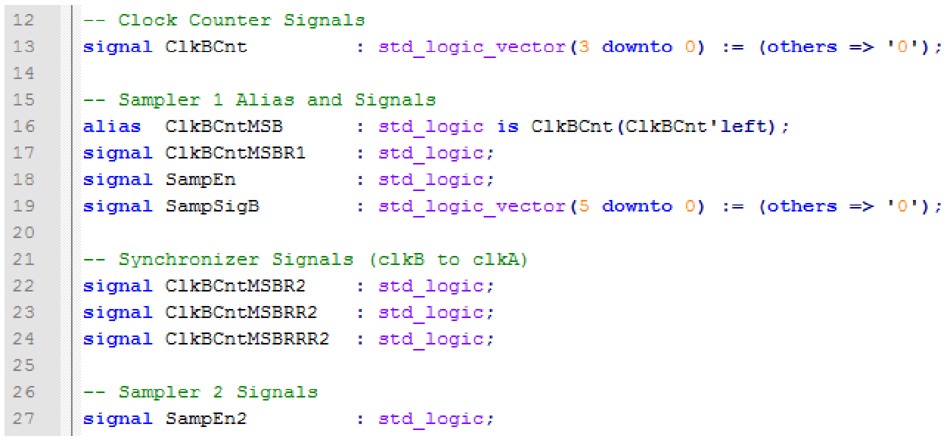


```vhdl
entity synchex2 is
port (
        clkA     : in std_logic; -- Slow Clock Domain
        clkB     : in std_logic; -- Fast Clock Domain
        sigB     : in std_logic_vector(5 downto 0);
        sigA     : out std_logic_vector(5 downto 0)
      );
end synchex2;
```

Figure 14 Entity

In this example, we will transfer sigB, which originates at the clkB domain, to the clkA domain, as the output sigA. The architecture will be broken down into six simple, individual processes. Each of these processes has a clear and individual purpose. In a real design it might be desirable to merge some of these processes together. Remember that writing readable code is key to producing good designs, and you must consider whether breaking down the code this much will be beneficial or detrimental to the readability of your design.

We now introduce the signal declarations. These will be referred to as the signals appear and/or are used in the later processes.

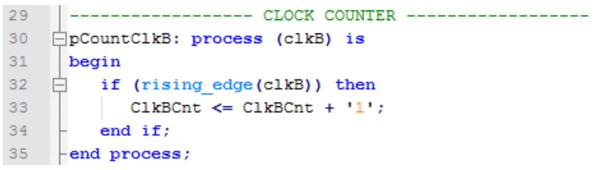


```vhdl
-- Clock Counter Signals
signal ClkBCnt          : std_logic_vector(3 downto 0) := (others => '0');

-- Sampler 1 Alias and Signals
alias  ClkBCntMSB       : std_logic is ClkBCnt(ClkBCnt'left);
signal ClkBCntMSBR1     : std_logic;
signal SampEn           : std_logic;
signal SampSigB         : std_logic_vector(5 downto 0) := (others => '0');

-- Synchronizer Signals (clkB to clkA)
signal ClkBCntMSBR2     : std_logic;
signal ClkBCntMSBRR2    : std_logic;
signal ClkBCntMSBRRR2   : std_logic;

-- Sampler 2 Signals
signal SampEn2          : std_logic;
```

Figure 15 Signal Declarations

We must start by analysing the relationship between clkA and clkB. How long must we assert a signal in the fast clkB domain to safely capture it in the slower clkA domain?

We will implement a counter to count the clkB, and take advantage in the fact that the most significant bit (MSB) of the counter vector will go from HIGH to LOW and from LOW to HIGH at a rate defined by the width of the vector. This change rate should correspond *at least* to 3 times the period of clkA (twice the three edge requirement), since we will exploit both transitions.



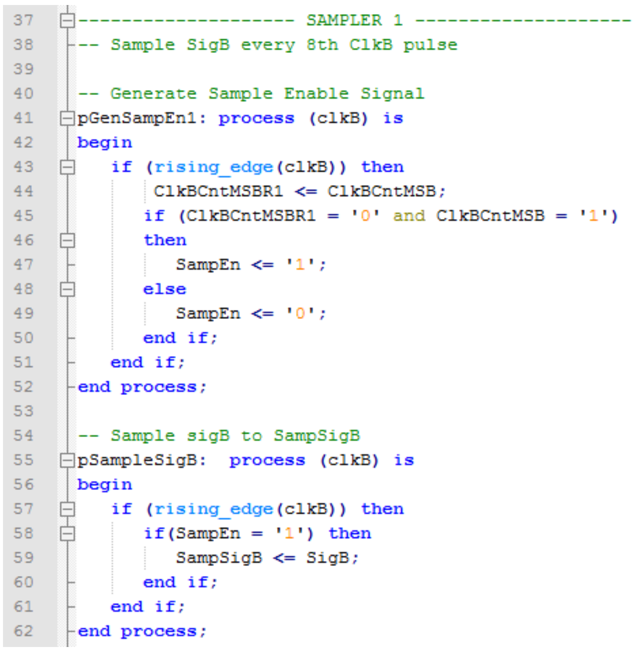
```
------------------ CLOCK COUNTER ------------------
pCountClkB: process (clkB) is
begin
    if (rising_edge(clkB)) then
        ClkBCnt <= ClkBCnt + '1';
    end if;
end process;
```

Figure 16 First Process: Clock Counter

In this example, the counter vector is called ClkBCount, and declared in line 13 (Figure 14). We have chosen the vector to have a width of 4 bits, meaning that the MSB will change every ninth clkB pulse. The MSB of ClkBCnt will look like a clock with a period 16 times as big as the period of ClkB. This division factor is suitable for an application displaying a dramatic difference between clkA and clkB.

An example would be if clkA was 50MHz and clkB was 160MHz. Then the rate of change of the MSB of the clkB counter would be $\frac{160MHz}{16} = 10Mhz$, and its period $tMSBclkB = 100ns$. Thus the relationship between both periods would be $(\frac{tMSBclkB}{tclkA} = \frac{100ns}{20ns} = 5) > 3$, satisfying, and exceeding, the edge requirement stated above.

Notice how in figure 14, line 16 we find the MSB of ClkBCnt using the ‘left VHDL-attribute, and give it the alias ClkBCntMSB to improve readability further down in the code. We could just as easily have picked the MSB using ClkBCnt(3), however, using the attribute method, we can change the width of ClkBCnt without having to worry about changing the code elsewhere.



```
-------------------- SAMPLER 1 --------------------
-- Sample SigB every 8th ClkB pulse

-- Generate Sample Enable Signal
pGenSampEn1: process (clkB) is
begin
    if (rising_edge(clkB)) then
        ClkBCntMSBR1 <= ClkBCntMSB;
        if (ClkBCntMSBR1 = '0' and ClkBCntMSB = '1')
        then
            SampEn <= '1';
        else
            SampEn <= '0';
        end if;
    end if;
end process;

-- Sample sigB to SampSigB
pSampleSigB:  process (clkB) is
begin
    if (rising_edge(clkB)) then
        if(SampEn = '1') then
            SampSigB <= SigB;
        end if;
    end if;
end process;
```

Figure 17 Second and Third Processes: First Sampling Sequence

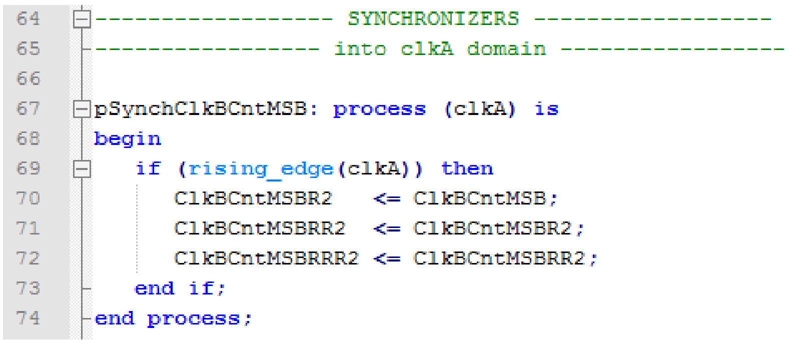
We now want to *detect* when ClkBCntMSB makes a transition from LOW to HIGH, and use this to generate a sample enable signal, SampEn, to enable the sampling of sigB. Thus, we will sample sigB after every eight pulse of clkB.

The process pGenSamp1 is used to generate SampEn. First, we pass ClkBCntMSB through a register. This is because we want to compare the present value of ClkBCntMSB with the value it had the preceding clkB period (and the output of the register, ClkBCntMSBR1, is ClkBCntMSB delayed by a clkB period). When we detect that ClkBCntMSB has gone from LOW to HIGH, we pulse SampEn HIGH.

In the next process, pSampleSigB, we sample the value of sigB to SampSigB every time SampEn is HIGH. The rate of change of SampSigB corresponds to the rate of change of ClkBCntMSB. SampSigB is the signal we wish to pass through our CDC path.

We need to generate a second sample enable signal, SampEn2, to determine the moment at which we read (sample) SampSigB into the clkA domain. We want this to happen at a time when we know SampSigB has a stable value, so we generate the SampEn2 when we detect that ClkBCntMSB makes a transition from HIGH to LOW. However, SampEn2 *must* be generated within the clkA domain.

It follows that ClkBCntMSB needs to be synchronized into the clkA domain. This can be done using a simple two flip-flop synchronizer, as ClkBCntMSBs' rate of change is so slow related to clkA.

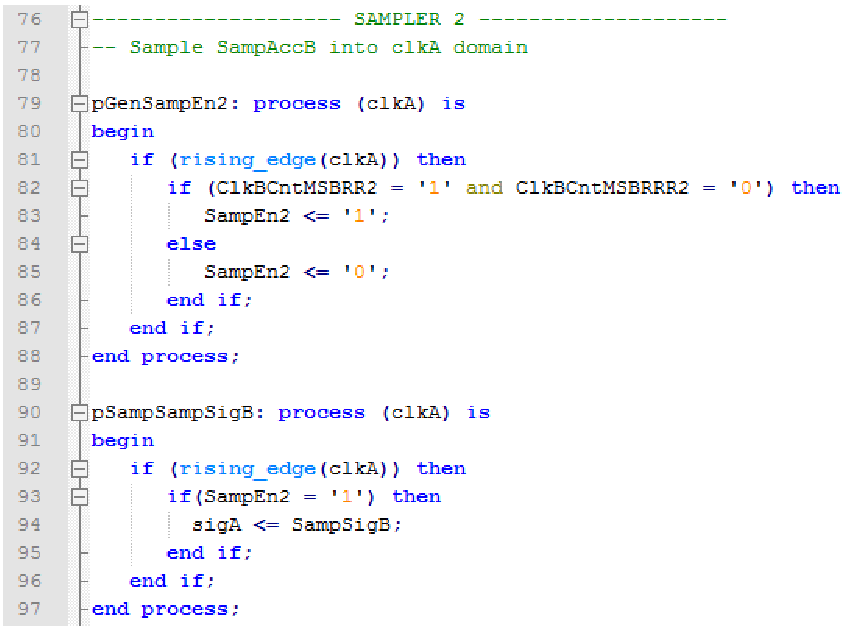


```vhdl
------------------ SYNCHRONIZERS ------------------
----------------- into clkA domain -----------------

pSynchClkBCntMSB: process (clkA) is
begin
   if (rising_edge(clkA)) then
      ClkBCntMSBR2    <= ClkBCntMSB;
      ClkBCntMSBRR2   <= ClkBCntMSBR2;
      ClkBCntMSBRRR2  <= ClkBCntMSBRR2;
   end if;
end process;
```

Figure 18 Fourth process: Synchronizer into ClkA domain

The process shown above contains three flip-flops. The reason is that we once more need to compare a value of ClkBCntMSB to its previous value. We cannot use ClkBCntMSB directly, as the signal is not synchronized into clkA. Neither can we use ClkBCntMSBR2, as the signal might display metastability. Thus, for the "current" value, we must use ClkBCntMSBRR2, and for the delayed value, ClkBCntMSBRRR2, the output of the third flop.

To generate SampEn2, we use the same procedure as used to generate SampEn. The only difference is that pGenSampEn2 is clocked by clkA instead of clkB.



```
-------------------- SAMPLER 2 --------------------
-- Sample SampAccB into clkA domain

pGenSampEn2: process (clkA) is
begin
   if (rising_edge(clkA)) then
      if (ClkBCntMSBRR2 = '1' and ClkBCntMSBRRR2 = '0') then
         SampEn2 <= '1';
      else
         SampEn2 <= '0';
      end if;
   end if;
end process;

pSampSampSigB: process (clkA) is
begin
   if (rising_edge(clkA)) then
      if(SampEn2 = '1') then
        sigA <= SampSigB;
      end if;
   end if;
end process;
```

Figure 19 Fifth and Sixth Processes: Second Sampling Sequence

Thus the crossing is secured, and sigB is safely read from a fast domain to a much slower domain.

Simulating the example in ModelSim produces following waveform patterns for one sampling of SigB over to SigA. (A screenshot of a larger simulation showing two transactions is found in Appendix E)

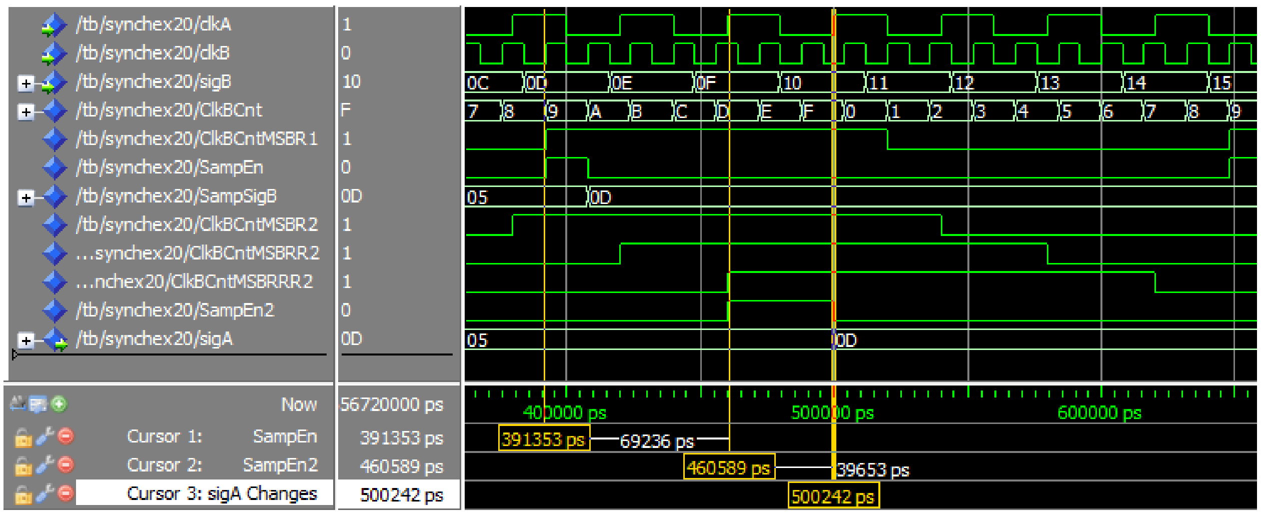

/tb/synchex20/clkA 1
/tb/synchex20/clkB 0
/tb/synchex20/sigB 10
/tb/synchex20/ClkBCnt F
/tb/synchex20/ClkBCntMSBR1 1
/tb/synchex20/SampEn 0
/tb/synchex20/SampSigB 0D
/tb/synchex20/ClkBCntMSBR2 1
...synchex20/ClkBCntMSBRR2 1
...nchex20/ClkBCntMSBRRR2 1
/tb/synchex20/SampEn2 0
/tb/synchex20/sigA 0D
0C 0D 0E 0F 10 11 12 13 14 15
7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9
05 0D
05 0D
Now 56720000 ps
Cursor 1: SampEn 391353 ps
Cursor 2: SampEn2 460589 ps
Cursor 3: sigA Changes 500242 ps
400000 ps
500000 ps
600000 ps
391353 ps
69236 ps
460589 ps
39653 ps
500242 ps

Figure 20 Waveform patterns for SynchEx2

4.2.4 Using Asynchronous FIFO Synchronizers

A method like the one described in 4.2.3 will sample a correct value when the sampling is enabled, but will not discern all states the input data signal goes through. Such an approach is therefore not suited for applications were it is important to capture all input data values; in such a case the use of an asynchronous FIFO is necessary to ensure no data is lost.

In an asynchronous FIFO, one clock domain writes data values to a FIFO buffer, yet the values are read from the buffer using another clock domain, which is asynchronous to the originating domain.

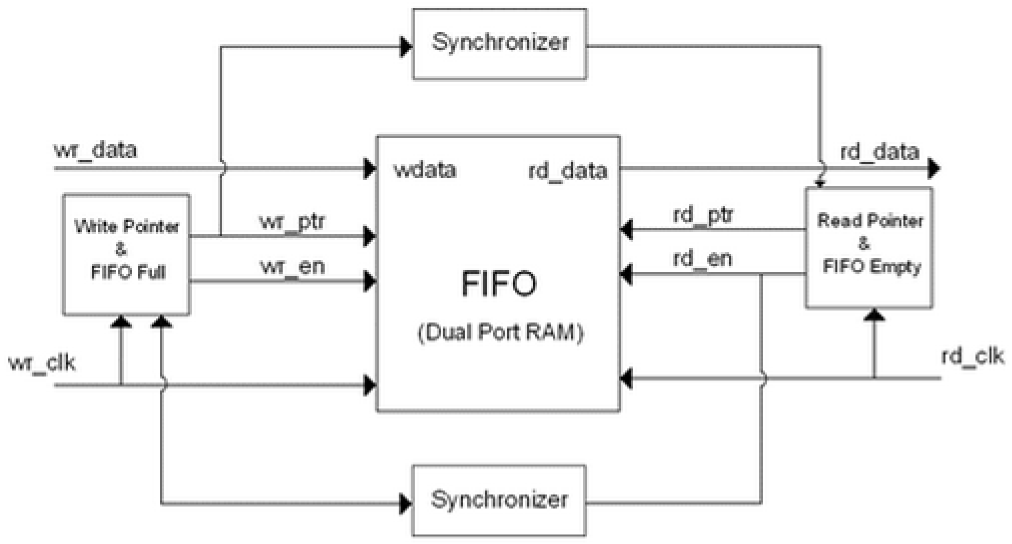


Figure 21 Asynchronous FIFO

The generation of full and empty flags in asynchronous FIFO queues is a sensitive issue. Since the write and read functions are governed by different clock domains, one such flag might be generated too late compared to the other clock domain and thus data corruption or loss might occur. To ensure the reliability of the design, these flags are in practice generated to mean “almost full” or “almost empty”, rather than truly specifying when the queue is full or empty.

In asynchronous FIFO design, it is important to use gray code for the incrementing and decrementing of the read and write pointers to the memory locations in the FIFO buffer. This is so because, as described in section 4.2.2, making sure that only one bit changes at a time decreases the probability of errors occurring.

5 Realization of the system

5.1 Interfacing the Clock Generator to the System

The Si5351 clock generator chip from Silicon Labs is an I2C-configurable clock generator featuring a 25Mhz crystal, three output channels, two internal phase locked loops, and multisynth-based fractional divider architecture. It can generate frequencies between 8kHz and 160Mhz.

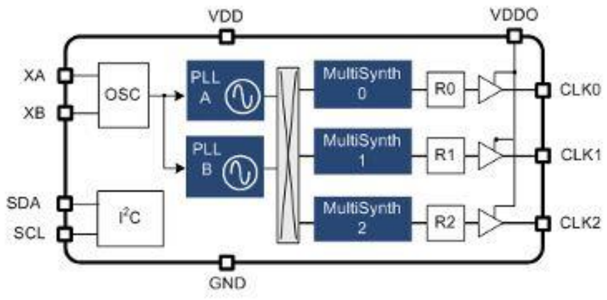


Figure 22 Inside the Si5351 Clock Generator

Adafruit Industries manufactures a breakout board for this chip, featuring SMA connectors which can be connected to the DE2-115 board, and thus no time has been devoted to the physical interfacing of the chip generator. However, to talk with the chip and configure the clock outputs, I2C functionality had to be added to the NIOS II system, and a specialized library for the Si5351 had to be written.

5.1.1 Overview of the I2C protocol

The Inter-Integrated-Circuit (I2C) serial interface bus, originally developed by Phillips (now NXP) [11], is an interface which uses only two wires. The bus consists of a clock line (SCL) and a data line (SDA), which both must be connected to the VDD0 supply with pull-up resistors.

The SCL line is one-directional and asserted by the master. The SDA line is bidirectional, and asserted as shown below.

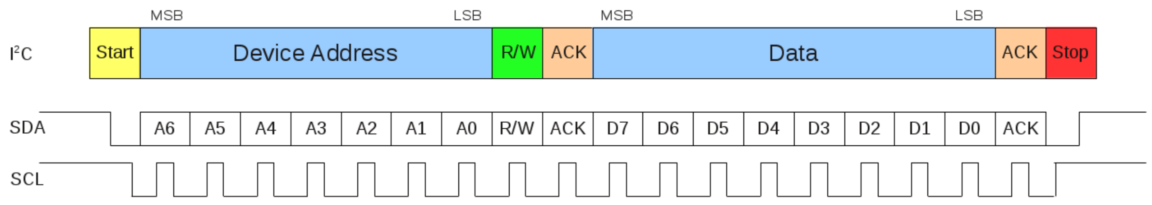


Figure 23 I2C Protocol

To start communication, the master sends a start bit, followed by a byte containing the slave address (A6:A0) and the direction of the transfer (R/W'). The addressed slave issues then an acknowledge bit (ACK). Subsequent bytes flow from sender to receiver depending on the transfer direction specified by the R/W' bit, asserted by the sender and acknowledged by the receiver, until the master issues a stop bit to stop the communication.

5.1.2 Adding I2C functionality to the NIOS II system

To add I2C functionality to the processor system, an intellectual property (IP) core was downloaded from AlteraWiki. The core is a modified version of an IP core found at OpenCores.org. The original core is a wishbone compliant core, and the AlteraWiki version has a "layer" on top of the Wishbone structure for interfacing with the Avalon bus.

The AlteraWiki IP core contained errors which required debugging, most notably, the name of the design files, as they were listed in the sw.tcl file (which specifies the drivers of the core), had a slight error which was not easy to notice immediately when checking that the path to the files was correct. Where the paths should have read: *include_source inc/i2c_opencores_regs.h*, they instead read *include_source inc/opencores_i2c_regs.h.*

During the debugging process other aspects of the design were modified. The header file listing the registers of the core was revisited using the specifications found on OpenCores.org. It is not clear whether or not it was only the error in the drivers' path in the sw.tcl file that was decisive to making the IP core work.

The driver functions corresponding to the IP core are listed below.

```
1. void I2C_init(alt_u32 base,alt_u32 clk,alt_u32 speed);
```

This function initializes the prescaler for the SCL line and enables the core. This function must be run before any other I2C code is executed.

The *base* input corresponds to the base address of the core. The *clk* input corresponds to the frequency given in Hz to the clock driving the component, and the *speed* input gives the desired SCL speed in Hz.

```
2. int I2C_start(alt_u32 base, alt_u32 add, alt_u32 read);
```

Sets the start bit and then sends the first byte, which is the address of the device and the write bit.

The *base* input corresponds to the base address of the core. The *add* parameter gives the address of the I2C device. *Read* can have the value 0 or 1 and determines whether the next operation is a read or write operation, correspondingly. The return value of the function is 0 if the slave address is acknowledged and 1 if the address is not acknowledged.

```
3. alt_u32 I2C_read(alt_u32 base,alt_u32 last);
```

This function can only be run after the I2C has been initialized (with I2C_init) and the particular slave has been addressed (through the I2C_start function) with the read parameter set to 1. It reads a single byte of data. To address a specific register you wish to read from, you must first write the address to the slave using the I2C_write function below.

```
4. alt_u32 I2C_write(alt_u32 base,alt_u8 data, alt_u32 last);
```

Just like I2C_read, this function assumes the core has been initialized and the slave has been successfully addressed, but in this case the *read* parameter should have been set to 0. The function writes a byte of data to the slave.

5.1.3 The Si5351 library

The register map used to read status, control, and configure the Si5351 is comprised of 255 registers, whereas about 109 of these are used in the library. The general procedure required to successfully configure the generator with a desired output frequency is outlined in the figure below, which is taken from the Si5351 datasheet.

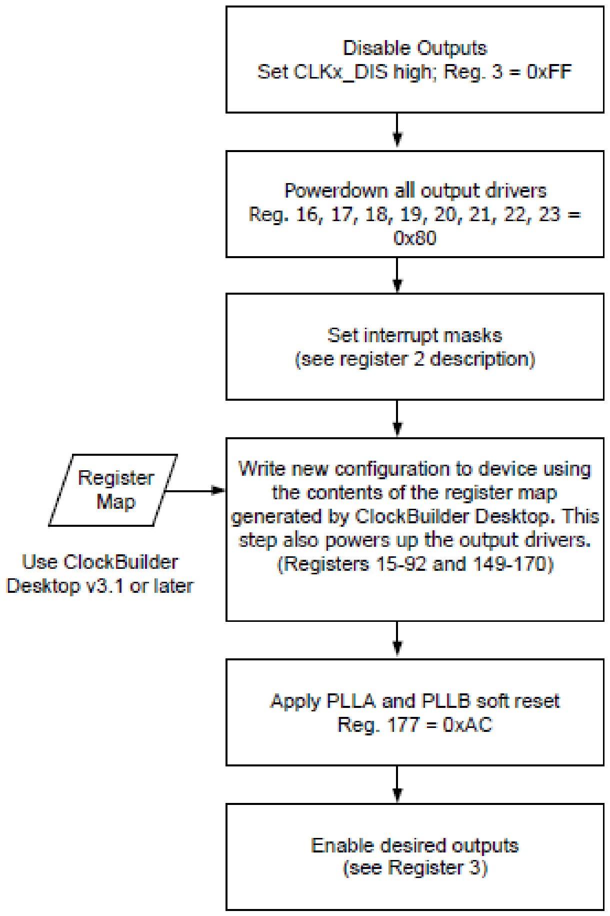
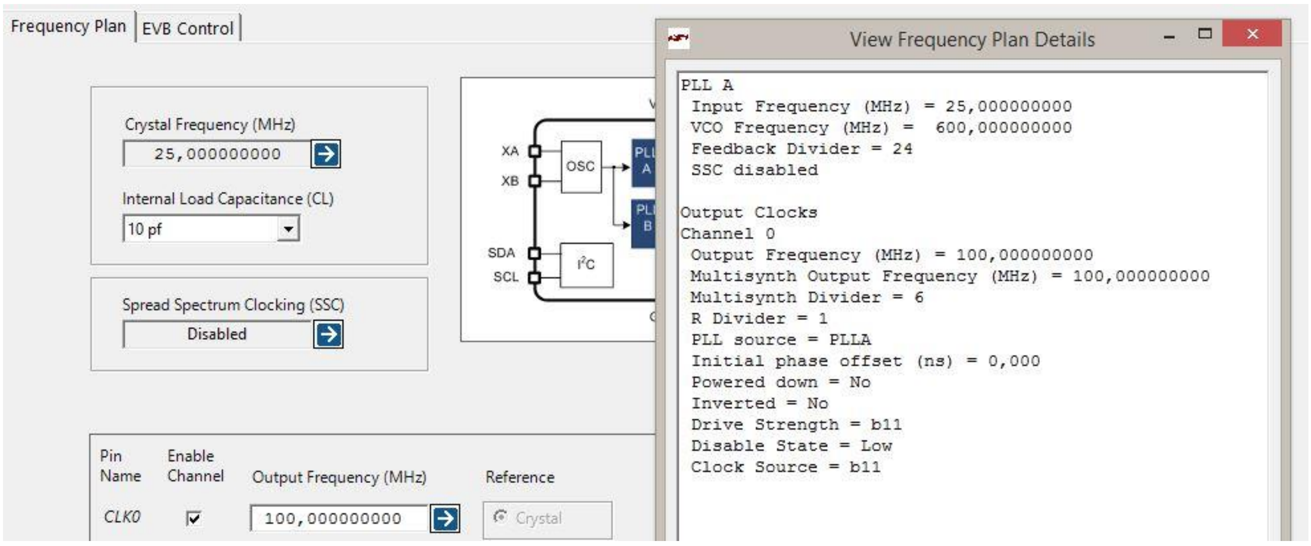

Disable Outputs
Set CLKx_DIS high; Reg. 3 = 0xFF
Powerdown all output drivers
Reg. 16, 17, 18, 19, 20, 21, 22, 23 = 0x80
Set interrupt masks
(see register 2 description)
Register Map
Use ClockBuilder Desktop v3.1 or later
Write new configuration to device using the contents of the register map generated by ClockBuilder Desktop. This step also powers up the output drivers. (Registers 15-92 and 149-170)
Apply PLLA and PLLB soft reset
Reg. 177 = 0xAC
Enable desired outputs
(see Register 3)

Figure 24 I2C programming procedure for the Si5351

The first three steps of the procedure are covered by the function called `Si5351_begin`, which also calls the `I2C_init` function that initiates the IP core in the system. The fourth step is covered by three functions that allow for the PLL, multisynth dividers, and R divider to be programmed. The last two steps are carried by the `enableOutputs` function.

Notice that step four mentions a "Clock Builder Desktop". This is a program developed by silicon labs which generates the parameters which must be programmed onto the chip to attain a desired frequency. Figure 25 features a frequency plan from the program for an output of 100 MHz.



```
PLL A
 Input Frequency (MHz) = 25,000000000
 VCO Frequency (MHz) =  600,000000000
 Feedback Divider = 24
 SSC disabled

Output Clocks
Channel 0
 Output Frequency (MHz) = 100,000000000
 Multisynth Output Frequency (MHz) = 100,000000000
 Multisynth Divider = 6
 R Divider = 1
 PLL source = PLLA
 Initial phase offset (ns) = 0,000
 Powered down = No
 Inverted = No
 Drive Strength = b11
 Disable State = Low
 Clock Source = b11
```

Figure 25 Clock Builder Desktop frequency plan

5.1.3.1 Overview of the Si5351 library functions

Below is an overview of the functions used to program the Si5351 chip. This library, written in C, is based on the C++ library written by Limor Fried from Adafruit Industries to interface the Si5351 breakout board with an Arduino microcontroller.

```
1. int SI5351_begin();
```

This function starts the I2C core in the NIOSII processor by calling `I2C_init`. It subsequently disables the outputs of the breakout board and sets the load capacitance for the crystal. The load capacitance is set by default to 10pF, but it can easily be changed to 6pF or 8pF if desired.

```
2. int SetTestConfiguration();
```

This function programs a test configuration to the board for testing purposes. Running this function gives following outputs:

Channel 0: 120.00 MHz **Channel 1:** 12.00 MHz **Channel 2:** 13.56 MHz

```
3. int setupPLL(int pll, alt_u8 mult, alt_u32 num, alt_u32 denom);
```

This function can be used to configure the multiplier of a either PLLA or PLLB. The *pll* parameter can be used to specify which PLL to configure, and must be either *SI5351_PLL_A*[3] for PLLA or *SI5351_PPL_B* for PLLB.

The output fVCO of the chosen PLL is configured, using the parameters mult, num, and denom, following the equation:

$$fVCO = 25MHz * (mult + (\frac{num}{denom}))$$

[3] *SI5351_PLL_A is defined to be 0, and SI5351_PPL_B to be 1. You can also write either 0 or 1 as the pll parameter, but that would make the code less intuitive/harder to read.*

The parameter *mult* must have a value between 15 and 90, *num* must have a value between 0 and 1048575, and denom must have a value between 1 and 1048575.

```
4. int setupMultisynth(alt_u8 output, int pllSource, alt_u32 div, alt_u32 num, alt_u32 denom);
```

The *output* parameter determines which output channel to use, and must be either 0, 1, or 2. The *pllSource* parameter must be either *SI5351_PLL_A* or *SI5351_PPL_B,* and must coincide with a previously configured PLL, since it specifies which PLL the multisynth divider will get its input from.

The multisynth divider can be configured in either integer or fractional mode. The advantage of using integer mode is that it reduces clock jitter.

For use in integer mode, the *div* parameter must be either 4, 6 or 8. The multisynth output fOUT is then given by :

$$fOUT = \frac{fVCO}{div}$$

For use in fractional mode, the value of *div* can range between 8 and 900. The limitations to *num* and *denom* are the same as in `setupPLL`, and the output frequency is given by the equation:

$$fOUT = fVCO * \frac{1}{div + (num/denom)}$$

```
5. int setupRdiv(alt_u8  output, int div);
```

The R divider can be used when it is necessary to achieve frequencies below 100KHz, which can divide fOUT once more by a fixed integer number. The *output* parameter is the same as in `setupMultisynth`, and the *div* parameter has to be an integer between 0 and 7.

```
6. int enableOutputs(int enable);
```

This function must be run after configuring the PLL and the dividers to enable the board’s outputs.

5.2 Designing the IP core for the Metastability Inducing Hardware

5.2.1 Hardware

The hardware part of the IP core consists of four vhdl files, whereas one of them is a package specifying names and register addresses. Each design unit will be commented below in inverse hierarchical order (meaning that the lowest level entity comes first and the highest level entity last). The hdl code can be found in the Appendixs.

5.2.1.1 Test Circuit: ms_hw.vhd

This design unit is simple and contains a single copy of the test circuit used to induce metastability. The RTL view of the circuit is shown below.

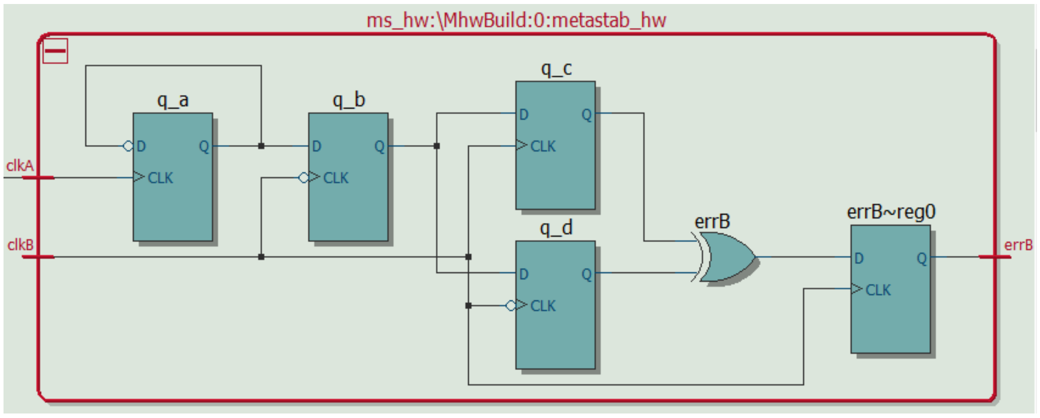


Figure 26 RTL view of test circuit

The input to register q_b comes from the domain of A, and changes every clock cycle of clkA. However, this register samples its input with a frequency dictated by clkB. Thus, the output of register q_a is asynchronous to the register that samples it, and represents a clock domain crossing.

The output of q_B is sampled by two different registers, q_c and q_d. Sampling at the input of Register q_c has a delay of a half clock period relative to register q_d. These registers would sample different values should the output from register B become metastable, assuming the metastable state lasted at least a time equivalent to the sum of half the clock period of clkB, the timing delay of the metastable signal through the network and the eventual clock skew of clkB. The exclusive-or circuit comparing the outputs from q_c and q_d would then detect that the registers sampled differing values and output a HIGH signal, which would indicate error.

5.2.1.2 Counting Circuitry: ms_fcount.vhd

This design unit is used to instantiate a number of test circuit copies specified by a generic variable in the top level entity. It is also used to scan the output of all circuit copies for failure and count the number of error events. The unit contains two clock domain crossings: one from the domain of the NIOS II processor clock running at 50MHz to the fast clkB domain, and one from the fast clkB domain to the 50MHz domain. The first one involves only one-bit control signals, and since the domain they originate at is so much slower than the receiving clkB domain, it is sufficient to use synchronizer chains into the receiving domain. The second crossing, however, is not only challenging because the originating domain is so much faster than the receiving domain, but also because the signals to be transferred are multibit signals. In this case, the approach described in section 4.2.3 was applied.

5.2.1.3 *Avalon Bus Interface: ms_top.vhd*

This design unit handles the connection with the Avalon bus, and thus the connection with the rest of the system.

5.2.2 Software

The IP core contains three registers, defined below. The functions in the software part of the core are the means by which these registers are read or written to.

1. Metastability Occurrence Count Register

Name	Mode	Address Offset	31-0
COUNT_REG	Read	0x00	Count Data

2. Control Register

Name	Mode	Address Offset	31	29-0
CONTROL_REG	Write	0x01	Control Bit	-

Setting the control bit to 1 starts the hardware, whereas clearing it stops the hardware.

3. Reset Register

Name	Mode	Address Offset	31-0
RESET_REG	Trigger Write	0x02	-

The functions are derived from the register definitions, and defined as follows:

`MS_TEST_READ`:	Reads the value stored in the count register.
`MS_TEST_STOP`:	Writes 0 to the most significant bit of the control register, and stops the test.
`MS_TEST_START`:	Writes 1 to the most significant bit of the control register, which starts the test.
`MS_TEST_RESET`:	Resets the count register when written to.

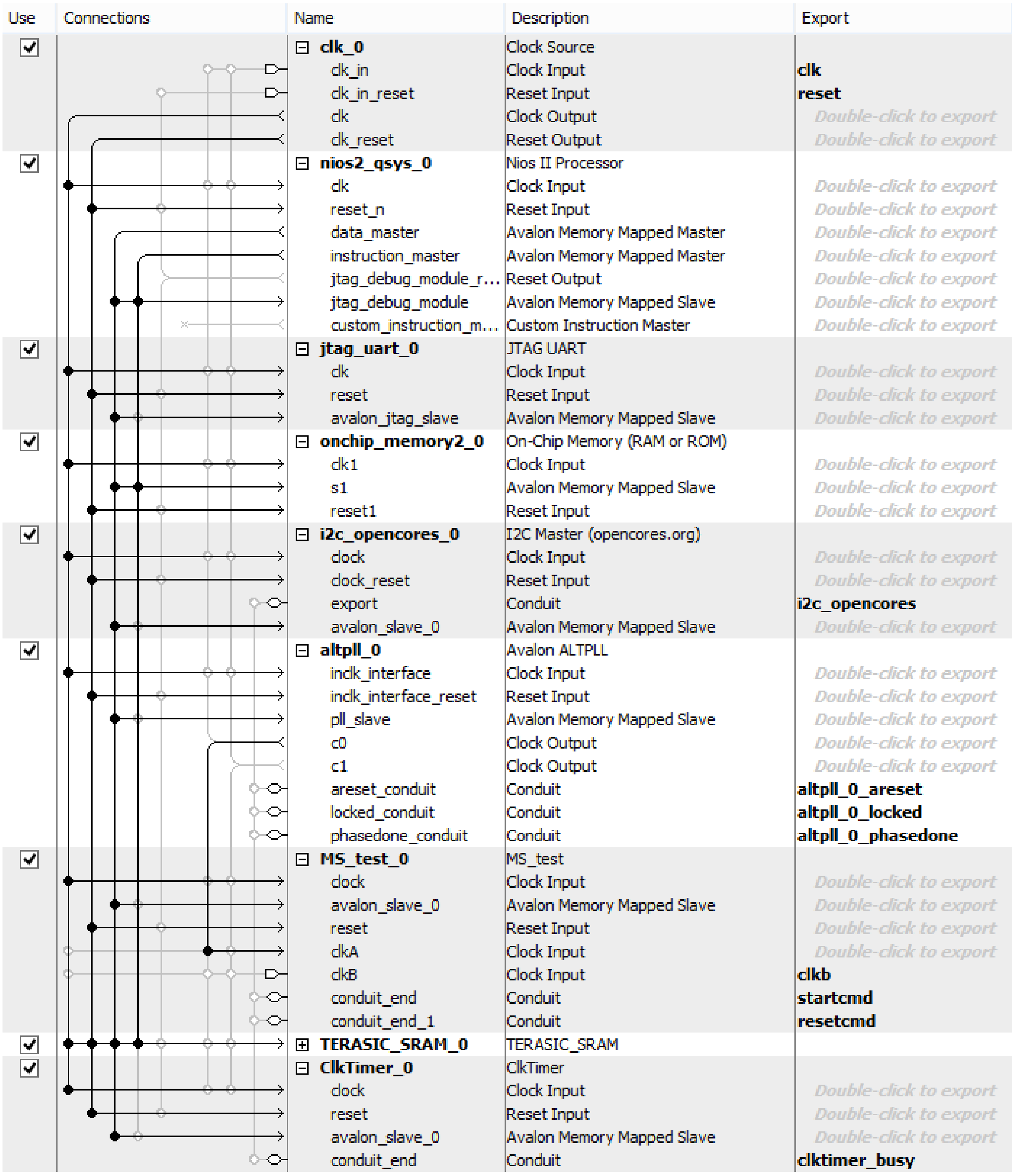
5.3 System Timing

To add timing to the system, and be able to monitor number of failure occurrences against time, a simple timer IP core was added to the system. The timing is done in terms of clock cycles, and is thus in principle only a simple counter triggered by the rising edge of the system clock with an Avalon bus interface on top. The timer is governed by start, stop, read and reset functions in software, which are very much like those outlined for the metastability inducing IP core above.

5.4 Nios II System

5.4.1 Hardware

The NIOS II processor is a soft processor developed by Altera. The processor can be made as complex as necessary by adding components and peripherals using Qsys, a tool which makes it simple to create of Avalon connections and which is included in the Quartus design software package.



Use	Connections	Name	Description	Export
✓		**clk_0**	Clock Source	
		clk_in	Clock Input	**clk**
		clk_in_reset	Reset Input	**reset**
		clk	Clock Output	Double-click to export
		clk_reset	Reset Output	Double-click to export
✓		**nios2_qsys_0**	Nios II Processor	
		clk	Clock Input	Double-click to export
		reset_n	Reset Input	Double-click to export
		data_master	Avalon Memory Mapped Master	Double-click to export
		instruction_master	Avalon Memory Mapped Master	Double-click to export
		jtag_debug_module_r...	Reset Output	Double-click to export
		jtag_debug_module	Avalon Memory Mapped Slave	Double-click to export
		custom_instruction_m...	Custom Instruction Master	Double-click to export
✓		**jtag_uart_0**	JTAG UART	
		clk	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)	
		clk1	Clock Input	Double-click to export
		s1	Avalon Memory Mapped Slave	Double-click to export
		reset1	Reset Input	Double-click to export
✓		**i2c_opencores_0**	I2C Master (opencores.org)	
		clock	Clock Input	Double-click to export
		clock_reset	Reset Input	Double-click to export
		export	Conduit	**i2c_opencores**
		avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export
✓		**altpll_0**	Avalon ALTPLL	
		inclk_interface	Clock Input	Double-click to export
		inclk_interface_reset	Reset Input	Double-click to export
		pll_slave	Avalon Memory Mapped Slave	Double-click to export
		c0	Clock Output	Double-click to export
		c1	Clock Output	Double-click to export
		areset_conduit	Conduit	**altpll_0_areset**
		locked_conduit	Conduit	**altpll_0_locked**
		phasedone_conduit	Conduit	**altpll_0_phasedone**
✓		**MS_test_0**	MS_test	
		clock	Clock Input	Double-click to export
		avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export
		reset	Reset Input	Double-click to export
		clkA	Clock Input	Double-click to export
		clkB	Clock Input	**clkb**
		conduit_end	Conduit	**startcmd**
		conduit_end_1	Conduit	**resetcmd**
✓		**TERASIC_SRAM_0**	TERASIC_SRAM	
✓		**ClkTimer_0**	ClkTimer	
		clock	Clock Input	Double-click to export
		reset	Reset Input	Double-click to export
		avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export
		conduit_end	Conduit	**clktimer_busy**

Figure 27 Screenshot from Qsys, showing the finished Demonstrator A Nios II system

The figure above shows the different components that were added to the Nios system to achieve the desired functionality of the demonstrator.

The topmost component clk_0 is the clock source driving the Nios II processor and provides the reset interface as well. Below is the Nios processor. In this case, it is the Nios II/e version which is the simplest version of the Nios II processor and does not include more sophisticated processor functionality such as instruction cache and branch prediction for pipeline stages.

The JTAG UART is necessary for the processor to be able to print messages to the console in Eclipse, and is therefore included. On-chip memory is essential to provide the processor with the minimum amount of memory to function. More memory is also added to the system by interfacing with an external SRAM, which ensures sufficient capability to store sampled data.

The I2C master is the core described in section 5.1.2. The MS_Test component is the metastability inducing hardware described in section 5.2. Notice how startCmd and resetCmd have been taken out as conduits. These are sent to LED's on the development board to monitor their status during testing. Finally, the ClkTimer is the core providing timing capability in the form of the clock tick counter named in section 5.3.

From Qsys, a system-on-a-programmable-chip (SOPC) information file and a block diagram symbol are generated. The block diagram symbol is used to interface conduit ends to physical I/Os on the development board. It also produces a hardware tcl file, which is the means by which Qsys detects the presence of the component in the projects file system.

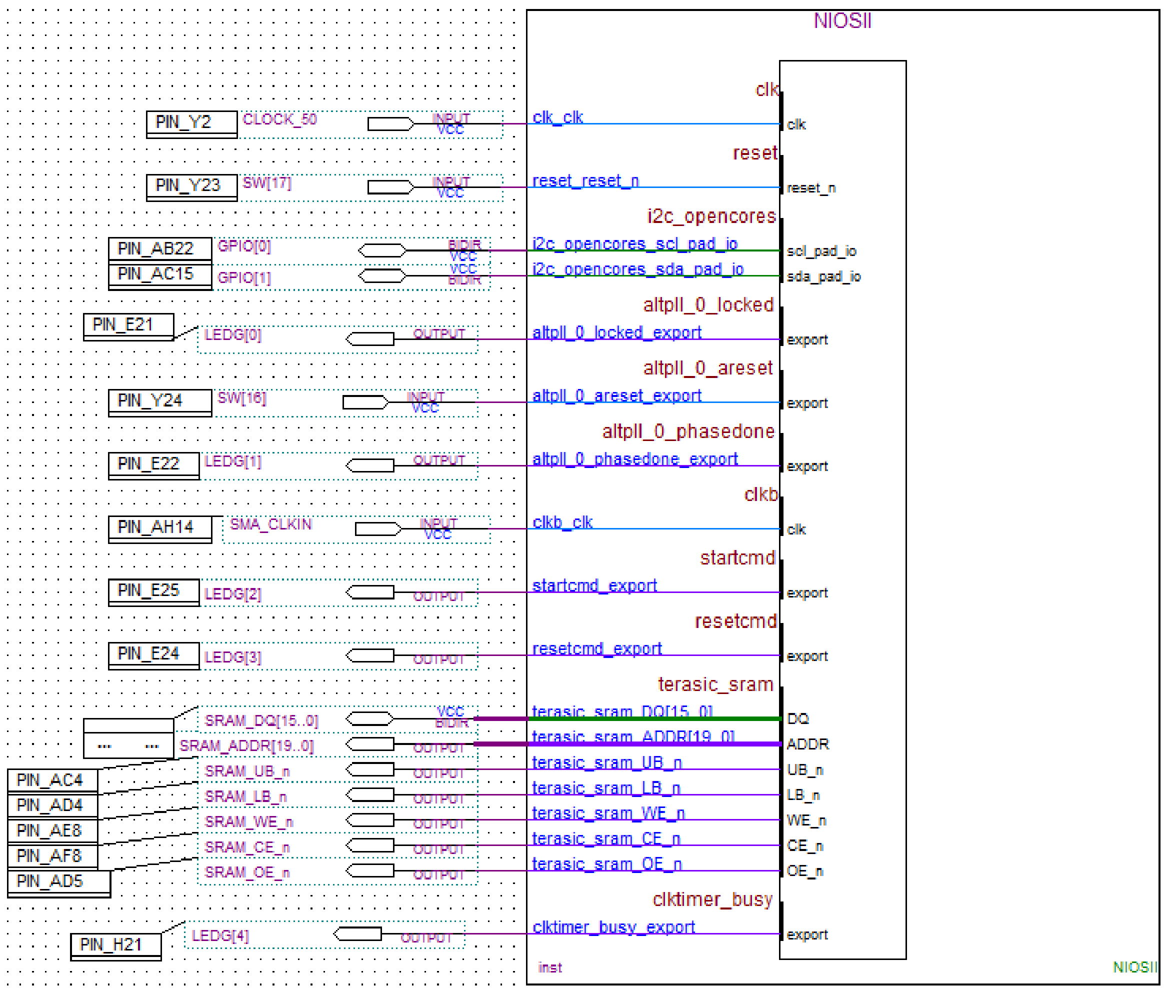


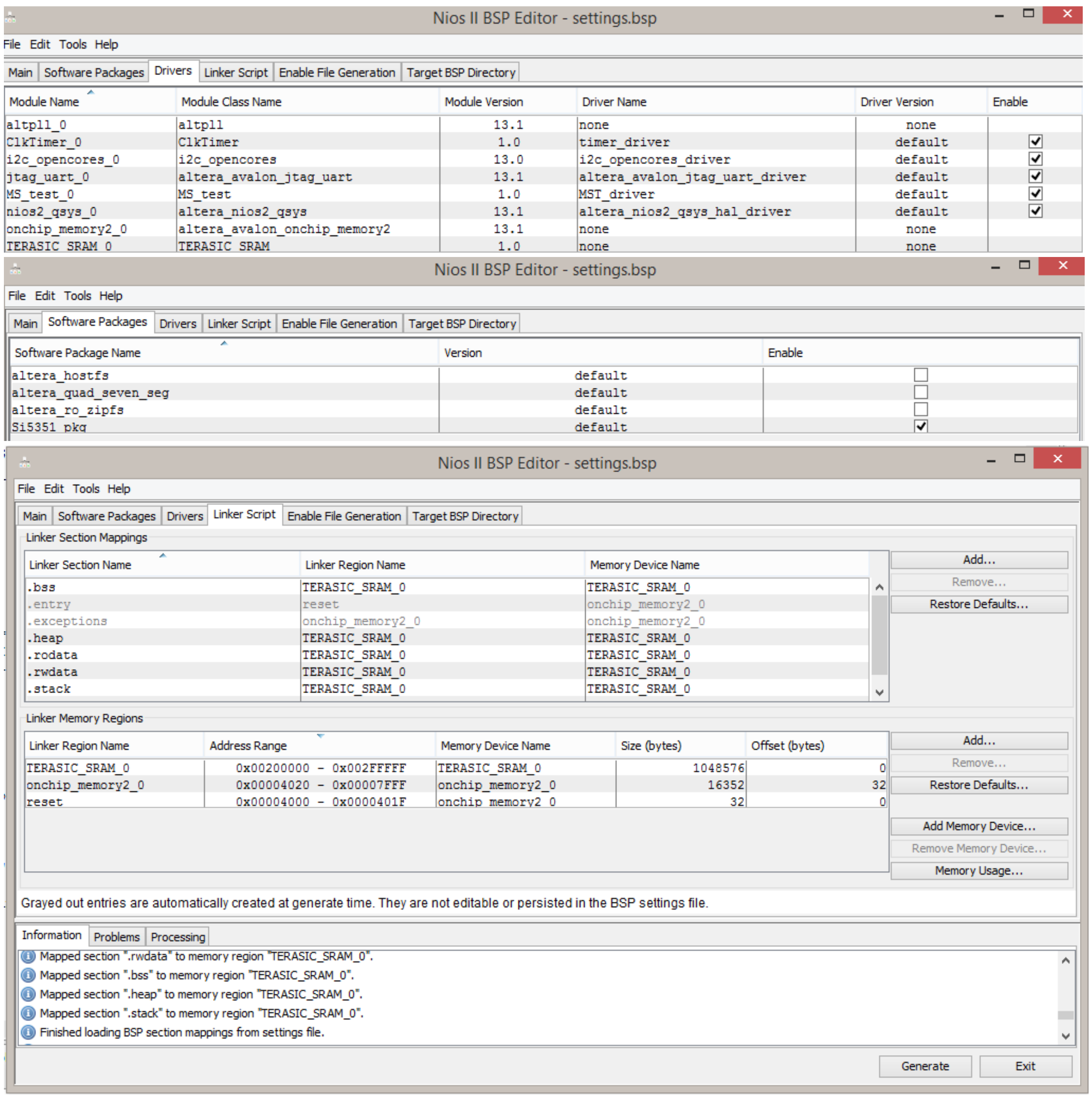
Figure 28 Screenshot from Quartus, showing the finished Demonstrator A system

We can see from the figure above that the SCL and SDA lines for the I2C interface are taken out on GPIO pins, which allow for connection to the external breakout board for the Si5351 chip. Furthermore, we see that the clkB input to the MS_test component is taken from the SMA_CLKIN pin, not as a conduit but in the manner of a clock sink, which is important to ensure that the clock is routed to the clock network and not as standard logic.

5.4.2 Board Support Package and System Software

The NIOS II software build tools for Eclipse is the tool Altera provides for the development of software projects for NIOS II systems. Out of a board support package (BSP) project and an application project, the tool builds an executable and linking format file (.elf) that can be uploaded to the NIOS II system.

The board support package is initially generated from the .SOPC information file, and can be further customized using the NIOS II BSP editor. The BSP editor can be used to adjust settings for the hardware abstraction layer (HAL), which provides the interface that allows programs to successfully access the underlying hardware. It can also be used to manage the software packages and other drivers which correspond to the project, and define memory regions for the linker script, amongst others.


Nios II BSP Editor - settings.bsp
File Edit Tools Help
Main Software Packages Drivers Linker Script Enable File Generation Target BSP Directory

Module Name	Module Class Name	Module Version	Driver Name	Driver Version	Enable
altpll_0	altpll	13.1	none	none	
ClkTimer_0	ClkTimer	1.0	timer_driver	default	✓
i2c_opencores_0	i2c_opencores	13.0	i2c_opencores_driver	default	✓
jtag_uart_0	altera_avalon_jtag_uart	13.1	altera_avalon_jtag_uart_driver	default	✓
MS_test_0	MS_test	1.0	MST_driver	default	✓
nios2_qsys_0	altera_nios2_qsys	13.1	altera_nios2_qsys_hal_driver	default	✓
onchip_memory2_0	altera_avalon_onchip_memory2	13.1	none	none	
TERASIC_SRAM_0	TERASIC_SRAM	1.0	none	none	

Nios II BSP Editor - settings.bsp
File Edit Tools Help
Main Software Packages Drivers Linker Script Enable File Generation Target BSP Directory

Software Package Name	Version	Enable
altera_hostfs	default	☐
altera_quad_seven_seg	default	☐
altera_ro_zipfs	default	☐
Si5351_pkg	default	✓

Nios II BSP Editor - settings.bsp
File Edit Tools Help
Main Software Packages Drivers Linker Script Enable File Generation Target BSP Directory
Linker Section Mappings
Add...
Remove...
Restore Defaults...

Linker Section Name	Linker Region Name	Memory Device Name
.bss	TERASIC_SRAM_0	TERASIC_SRAM_0
.entry	reset	onchip_memory2_0
.exceptions	onchip_memory2_0	onchip_memory2_0
.heap	TERASIC_SRAM_0	TERASIC_SRAM_0
.rodata	TERASIC_SRAM_0	TERASIC_SRAM_0
.rwdata	TERASIC_SRAM_0	TERASIC_SRAM_0
.stack	TERASIC_SRAM_0	TERASIC_SRAM_0

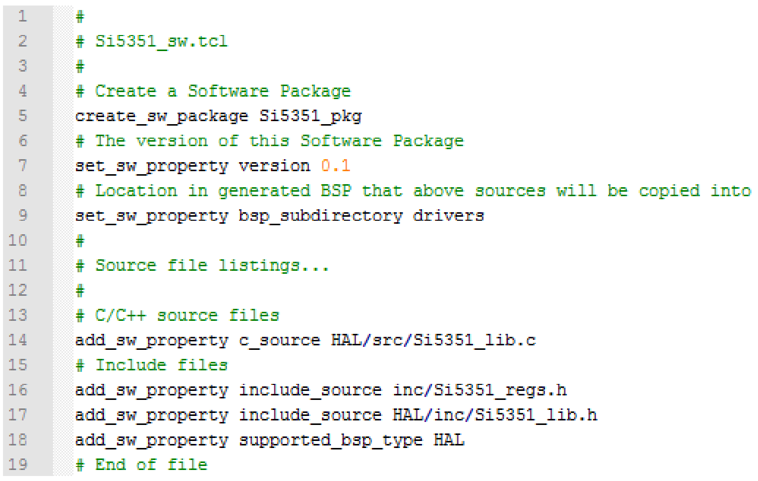
Linker Region Name	Address Range	Memory Device Name	Size (bytes)	Offset (bytes)
TERASIC_SRAM_0	0x00200000 - 0x002FFFFF	TERASIC_SRAM_0	1048576	0
onchip_memory2_0	0x00004020 - 0x00007FFF	onchip_memory2_0	16352	32
reset	0x00004000 - 0x0000401F	onchip_memory2_0	32	0

Grayed out entries are automatically created at generate time. They are not editable or persisted in the BSP settings file.

```
Mapped section ".rwdata" to memory region "TERASIC_SRAM_0".
Mapped section ".bss" to memory region "TERASIC_SRAM_0".
Mapped section ".heap" to memory region "TERASIC_SRAM_0".
Mapped section ".stack" to memory region "TERASIC_SRAM_0".
Finished loading BSP section mappings from settings file.
```

Figure 29 Screenshots from the BSP Editor

Figure 29 shows some of the options provided by the BSP editor. Notice that the Si5351 library goes under the option “software package”. To add a software package to a project a tcl file must be written, and then the software package must be activated inside the BSP editor. This activation is necessary because software packages are not added automatically to the project. Below, the tcl code which was used to add the Si5351 software package to the project is shown.



```
#
# Si5351_sw.tcl
#
# Create a Software Package
create_sw_package Si5351_pkg
# The version of this Software Package
set_sw_property version 0.1
# Location in generated BSP that above sources will be copied into
set_sw_property bsp_subdirectory drivers
#
# Source file listings...
#
# C/C++ source files
add_sw_property c_source HAL/src/Si5351_lib.c
# Include files
add_sw_property include_source inc/Si5351_regs.h
add_sw_property include_source HAL/inc/Si5351_lib.h
add_sw_property supported_bsp_type HAL
# End of file
```

Figure 30 Tcl file used to add the Si5351 software package to the project

Tcl files must also be written for the BSP builder and editor to detect the presence of software drivers for the hard components of the system. However, if written correctly, the BSP builder will add the drivers automatically to the BSP, so that it is not necessary to manually activate the drivers from the BSP editor.

6 Testing

Testing was performed continuously during the design of the demonstrator systems, as is necessary to ensure that the components behave as they are designed to do.

6.1 I2C Interface

Testing of the I2C interface was done primarily by connecting the system to the I2C device and attempting to communicate using the I2C IP core drivers in debug mode. Thus, a message was printed to the console of the NIOS II system which showed whether the device address was acknowledged by the slave, and thus whether communication succeeded or not. However, the signals were also monitored using SignalTap, which allowed the for the protocol waveforms to be observed and to ensure they were generated correctly. Additionally, the SCL and SDA signals out of the development board where monitored using an oscilloscope to ensure that the routing and pull-up resistor setup were correct.

6.2 Si5351 library

Once the I2C Interface was proven to work satisfactorily, the `SI5351_begin()` and `SetTestConfiguration()` were written. The testing consisted on monitoring the output from the clock generator using an oscilloscope until the expected results where produced. This implied also learning the correct procedure to use the I2C driver functions.

In the beginning, only the outputs of channel two and three where monitored. This was so because at the time only a Red Pitaya[4] was available to conduct the tests, which was a very inaccurate testing method, but served for the purpose at that stage. Later, a Tektronix MS05204 Mixed Signal Oscilloscope with a 2GHz bandwidth was made available for the project, which was much better suited for work at high frequencies than the humble Red Pitaya.

After the correct output was achieved using `SetTestConfiguration()`, work with the rest of the library was resumed. To start with, the library was written to only support integer mode configuration, and when this was achieved and verified with the oscilloscope, the library was expanded to support integer mode configuration, which facilitated the use of the Clock Builder Desktop. Once this was done, it was easy to rigorously check the concordance between parameters sent to the functions of the library and expected results using the Tektronix MS05204 Mixed Signal oscilloscope.

[4] "Red Pitaya is an open source project developed around a reconfigurable measurement instrument in size of a credit card. It can replace many expensive laboratory measurement and control instruments. The users can […] view and modify the published source code in order to develop new applications and share their results with the community. The Red Pitaya unit is a network attached device based on Linux operating system. It includes RF signal acquisition and generation technologies, FPGA, Digital Signal Processing and CPU processing." [14]

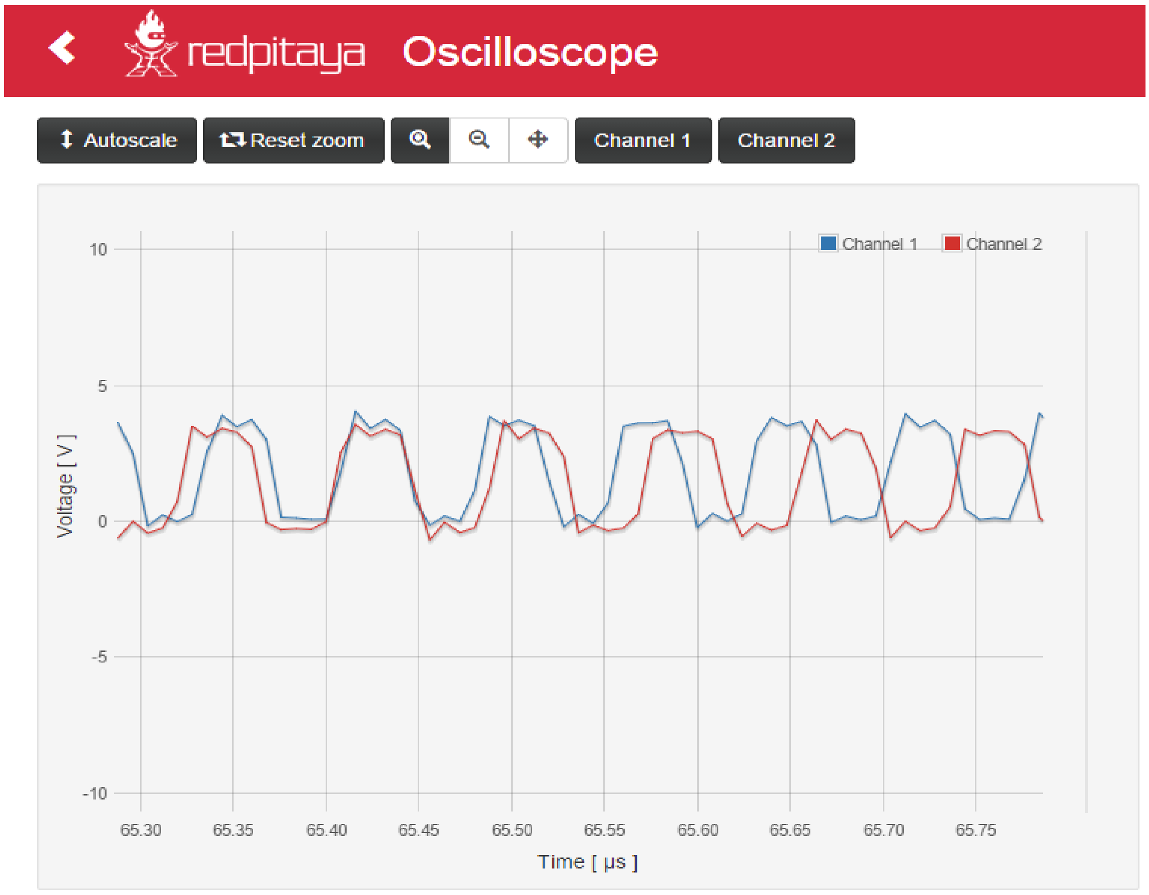


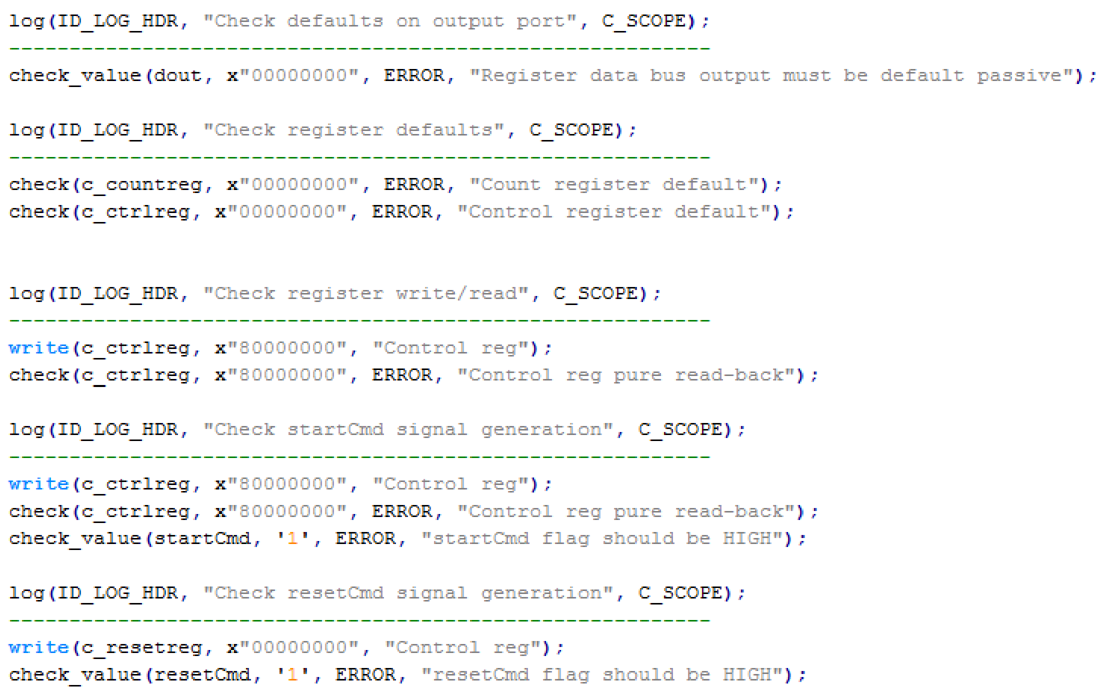
Figure 31 First successful programming of the clock generator. Screenshot from the Red Pitaya Oscilloscope.

6.3 IP Core for Metastability Inducing Hardware

For the design of the IP Core for Metastability Inducing Hardware, extensive support and feedback was provided by the technical advisor, which drastically improved the quality of the design at the time at which testing began. Both sub entities of the IP core (ms_hw.vhd and ms_fcount.vhd) required no adjustments at that point.

The design was simulated using ModelSim. The testbench for simulation was written using the Bitvis utility library, and the aim of the testbench was to verify the proper function of the Avalon bus handler. The tests implemented in the testbench ultimately led to the discovery of a bug in the drivers written for the IP core. These tests were not implemented before after trying to use the IP core within the system. This proved to be unwise, as it delayed the discovery and solution of the bug in the IP core drivers.

The implemented test-cases are shown below.



```
log(ID_LOG_HDR, "Check defaults on output port", C_SCOPE);
------------------------------------------------------------
check_value(dout, x"00000000", ERROR, "Register data bus output must be default passive");

log(ID_LOG_HDR, "Check register defaults", C_SCOPE);
------------------------------------------------------------
check(c_countreg, x"00000000", ERROR, "Count register default");
check(c_ctrlreg, x"00000000", ERROR, "Control register default");


log(ID_LOG_HDR, "Check register write/read", C_SCOPE);
------------------------------------------------------------
write(c_ctrlreg, x"80000000", "Control reg");
check(c_ctrlreg, x"80000000", ERROR, "Control reg pure read-back");

log(ID_LOG_HDR, "Check startCmd signal generation", C_SCOPE);
------------------------------------------------------------
write(c_ctrlreg, x"80000000", "Control reg");
check(c_ctrlreg, x"80000000", ERROR, "Control reg pure read-back");
check_value(startCmd, '1', ERROR, "startCmd flag should be HIGH");

log(ID_LOG_HDR, "Check resetCmd signal generation", C_SCOPE);
------------------------------------------------------------
write(c_resetreg, x"00000000", "Control reg");
check_value(resetCmd, '1', ERROR, "resetCmd flag should be HIGH");
```

Figure 32 Test cases implemented in the testbench for the Avalon bus handler of the IP core.

Running the testbench returned no failures, which suggested two possible situations: that errors in fact failed to be generated by the test circuits, or that the problem resided in the software rather than the hardware. The bus handler had already been rewritten once before testing with the testbench, and since the failure ultimately was proved to be in the software, it might not have been necessary to rewrite the bus handler in first place.

After testing with the testbench, the demonstrator was run again and the Avalon bus handler was monitored using SignalTap. It was apparent from the waveforms produced by the logic analyser that the StartCmd flag remained low for an indefinite amount of time. Checking the device drivers against the testbench showed clearly that the `MS_TEST_START` driver function attempted to write x”10000000” rather than x”80000000” to the control register, and thus the test was never properly started.

6.4 Demonstrator Type A as a whole

Testing of the Demonstrator Type A was in the beginning attempted by trying to start the metastability inducing circuitry, wait for some clock cycles and read the failure count register, but the count register was empty every time.

The signals form the ms_hw.vhd entity were subsequently monitored using SignalTap. An extra clock was programmed using the Altera PLL to provide a sampling clock for the logic analyser running significantly faster than the clocks in the metastability inducing circuitry. I attempted to sample the clocks in the crossing as well as the rest of the involved signals (including the error output signal). This was unwise and lead to misleading results: apparently, no error flag was being generated out of the metastability inducing circuit. The test was left to run for long periods of time, with all from 3 to 3000 copies of the circuit, and no error was generated which could be detected on SignalTap or read from the count register. It became apparent later in a later stage of testing that the reason no failures were observed was not that error signals weren't being generated, but that attempting to monitor the clocks in the crossing simultaneously was impractical at best.

This approach to monitoring the signals led to a significant delay, since the apparent lack of error flags in the low level entity could not be attributed to errors in the design; the test circuit is simple enough that the HDL had been checked thoroughly several times, and the RTL view had been scrutinized to ensure the Quartus Tool was not optimizing in a way that changed the functionality of the circuit. It seemed unlikely that the problem could be found in a higher level entity when the lower entity apparently failed to produce error flags to begin with.

The Avalon bus of the IP core was revisited as described in section 6.3 only after several days had been squandered expecting more copies of the circuit or longer running times to produce failures in the low level ms_hw.vhd entity. However, when this was done, the system seemed to work as it should, and reported an extremely high number of metastability occurrences in short periods of time. However, error flags were still not being observed in SignalTap, but this was only because monitoring the clocks in the crossing was still being attempted at that point.

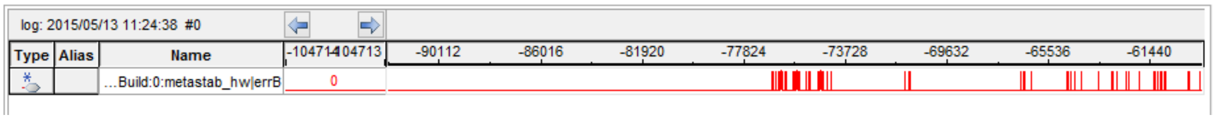


Figure 33 Error signals can be observed in SignalTap when monitored on their own

From this point onwards the changes done to the demonstrator took place primarily in software, making small adjustments to improve the consistency of the sampling rate and period. At the beginning, both reading and printing the values of the time and failure registers was done in a loop, which was inconvenient because the NIOS II processor had need for too much time to print the values to the console and this resulted in a poor sampling rate. The sampling rate was also not consistent, in such a manner that sometimes long time intervals without sampled values would appear in the data.

The solution to this problem was easy: the data samples were read into in a read loop, and then printed to the console on write loops. When this was done the amount of data collected in the determined test period, which was 0.2s, was too much, and a little wait loop had to be inserted to the sample reading loop to adjust the amount of data so that it was possible to print the arrays to the console and transfer the data manually to MATLAB.

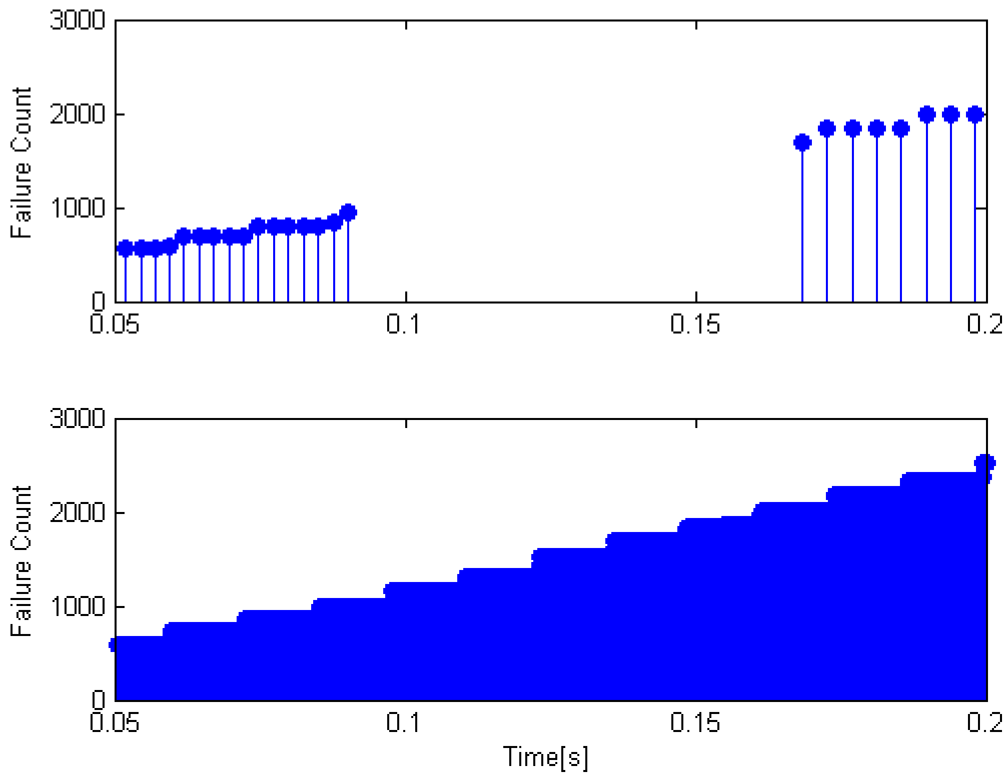


Figure 34 Data collected from the demonstrator system

In the figure above it is possible to observe the dramatic difference that the discussed adjustments produced regarding the density and consistency of the collected data sample points. The topmost stem plot depicts the results from a test where the data was acquired and displayed by means of the same loop, whereas the lowermost stem plot shows the result of a test where both functions where separated to different loops. From the topmost plot it is also possible to observe how failing to allocate the functions to different loops resulted in long time periods without data samples.

7 Discussion

The three main goals of this project were as follows:

- To produce a brief reference for digital designers with little to no experience regarding metastability in clock domain crossings. (Literature review)
- To make a system which allows the phenomenon to be observed and studied. (Demonstrator Type A)
- To make a system in which data transfer across clock domains might be done and tested using various methods. (Demonstrator Type B)

Initially, seven weeks were allocated to work with the literature review and the type A demonstrator in parallel. Of these seven weeks, four and a half were to be devoted specifically to designing the system, one and a half should have covered the writing of the literature review, and the last one was left unplanned.

However, the plan didn't account for the time needed to include I2C functionality to the system and interface the clock generator. These two tasks actually required four weeks of work to be completed satisfactorily.

Neither did the plan contemplate the difficulties that were encountered while testing the demonstrator system. The system was deemed ready for testing at week 13, which was about four weeks behind schedule, yet didn't produce the expected results before week 16. It was first deemed finished late in week 17, about eight weeks behind schedule.

The analysis to the project risks (Appendix D.4) contemplated the possibility that the workload required by the project goals was unrealistic in regard to the available time and work capacity. Considering the above described delays, this assumption was not unfounded.

7.1 Demonstrator Type A: Results

The data collected from the demonstrator type A shows that, at regular intervals, the failure count increases in great amounts, but is next to constant between said intervals. The brief bursts of metastable events generated correspond to the times at which the clocks involved in the crossing glide in and out of phase.

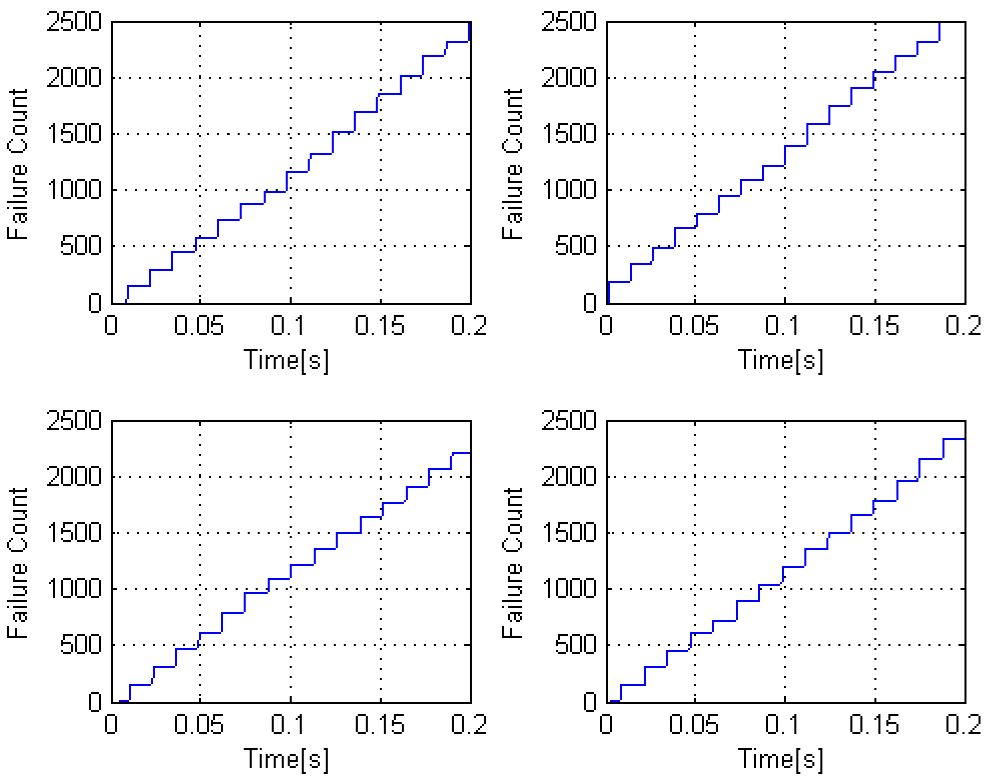


Figure 35 Selected plots from test results

The high frequencies make it difficult to sample the increments occurring during these bursts of metastable event. Figure 36 is a detail stem plot, showing the sampled values in one of such metastable event burst.The data used to make the stem plot in figure 36 is a subset of the data used for the rightmost graph in the top row of figure 35. Between the start and the end of the burst, only three data points were captured.

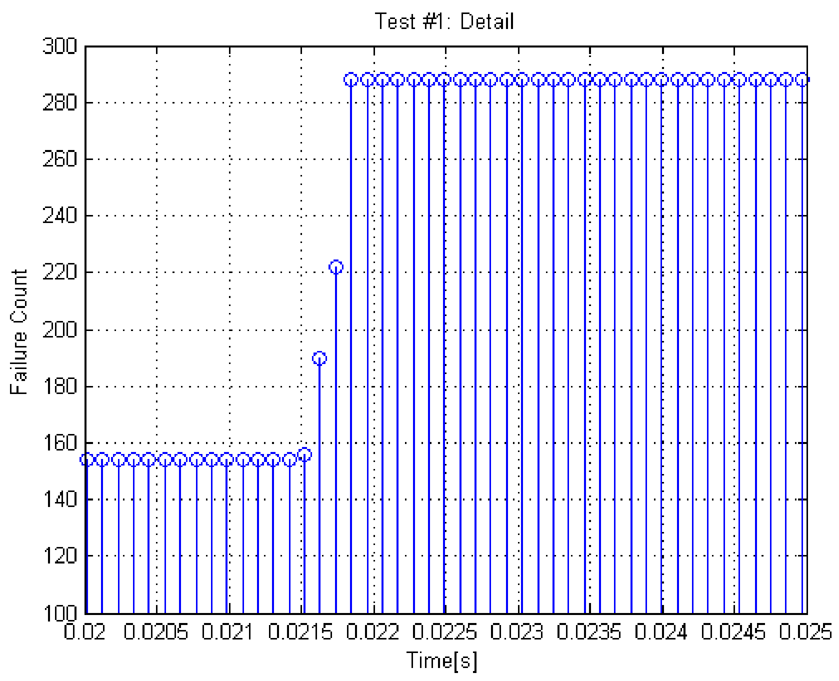


Figure 36 Detail stem plot of metastable event burst

Changing the test parameters to make the test interval shorter and the sample density higher results in following stem plot for a metastable event burst.

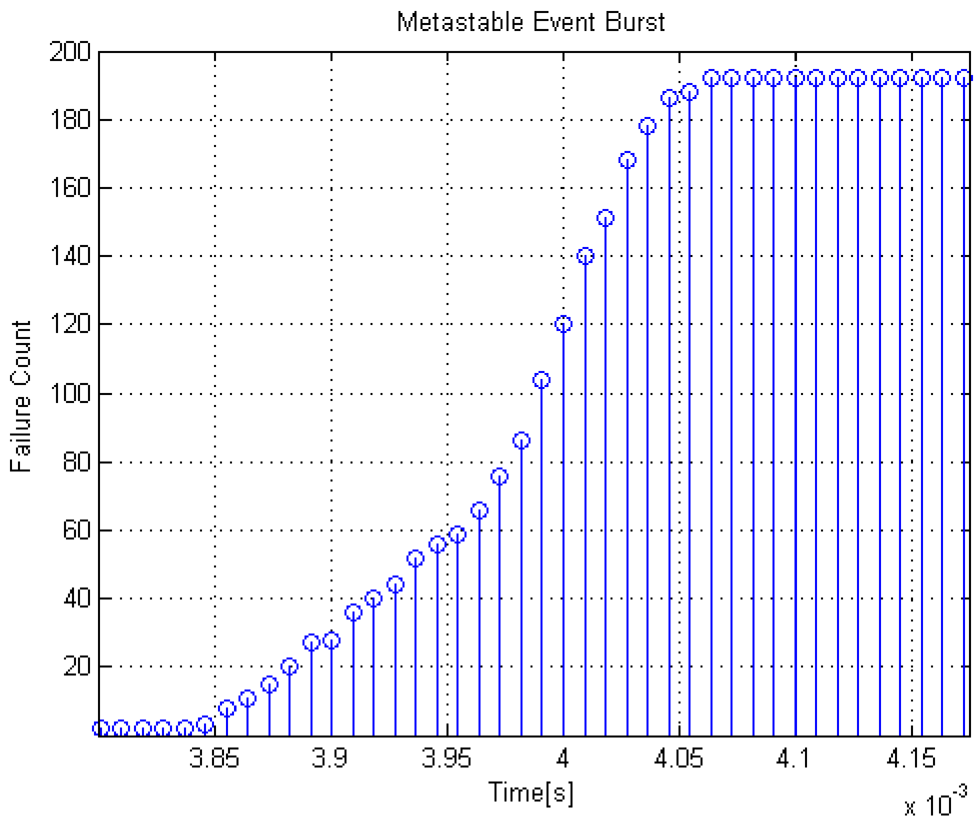


Figure 37 Detail stem plot of metastable event burst using a higher sampling rate

For a test interval of 0.2 seconds, the resultant metastable event count was recorded 3000 times. These results are presented in the histogram below, which is fitted with a Gaussian distribution.

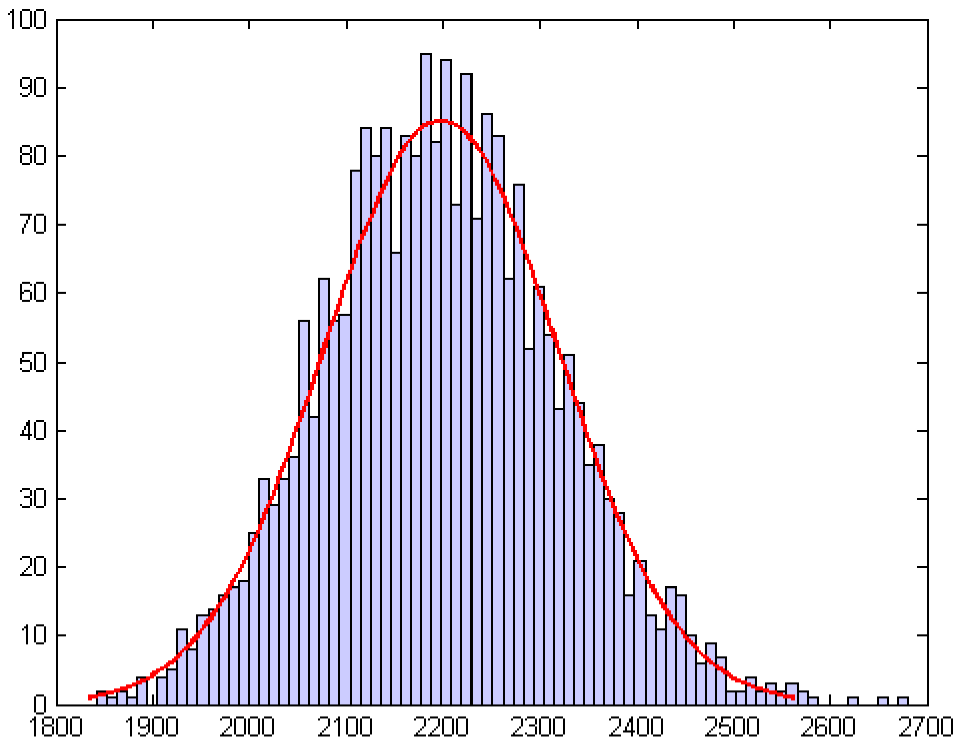


Figure 38 Spread of number of metastable events for an interval of 0.2 seconds

The fit of the histogram displays an expectation value of 2202 with a standard deviation of 121.37.

8 Conclusion

Knowing when and how to apply a particular technique when designing a clock domain crossing is a skill that distinguishes an experienced digital engineer, because no design tool can replace the insight and critical thought which is required to successfully identify the needs and hazards of a particular CDC situation.

The purpose of this project was first and foremost to make the topic more accessible to beginners, by trying to write in such a way that emulated the guidance of a mentor. This was, in truth, a reproduction of the guidance received by the technical advisor of the project and reflects the fact that, the dissemination of this kind of knowledge is best performed when a person is guided through a problem rather than by textbooks or articles. Such information can help the beginner designer gain insight, yet it cannot show him how to meet the requirements of a particular system.

The results from the demonstrator type A are unsurprising, and yet, they help put in perspective the potential danger imposed by the phenomenon of metastability in clock domain crossings. Seeing an example of mean time to failure so minuscule makes the hazard seem less distant to the designer.

It is a sad thing that the second demonstrator could not be completed within the time frame of the project, as doing so would have allowed for more techniques to be experimented with, and thus for more insight to be gained.

Therefore, it is advised that coming students take up the work which has been done here and expand it by continuing on the demonstrator type B: the necessary groundwork is already done. Said groundwork is comprised first and foremost by the interfacing of the external clock source, yet some efforts have been done to interface a PRSB generator and checker to the NIOS II system which have not been mentioned otherwise in this rapport given that the work is unfinished, but this work can easily be continued by coming students. The students would have an advantage in that they would have the opportunity to start right at the core of the problem statement, which is the study of CDC techniques, rather than spend much time setting up the basic system requirements.

Appendix A References

[1] D. Chen, D. Singh, J. Chromczak, D. Lewis, R. Fung, D. Neto and V. Betz, “A Comprehensive Approach to Modeling, Characterizing and Optimizing for Metastability in FPGAs,” *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '10,* February 2010.

[2] Real Intent Inc., Sunburst Design Inc. , «Clock Domain Crossing Demystified: The Second Generation Solution for CDC Verification».

[3] G. Tarawneh, «Solutions and Application Areas of Flip-Flop Metastability,» Newcastle University, UK, 2013.

[4] Terasic Technologies Inc., “DE-115 User Manual,” 2010. [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/13.0/Boards/DE2-115/DE2_115_User_Manual.pdf. [Accessed 26 January 2015].

[5] L. Fried, “Adafruit Industries: Learn,” 18 October 2014. [Online]. Available: https://learn.adafruit.com/downloads/pdf/adafruit-si5351-clock-generator-breakout.pdf. [Accessed 26 January 2015].

[6] S. L. Harris og D. M. Harris, Digital Design and Computer Architecture, Waltham: Elsevier, 2013.

[7] S. Verma og A. S. Dabare, «Understanding Clock Domain Issues,» *EE Times India,* December 2007.

[8] W. J. Dally og J. W. Poulton, Digital Systems Engineering, Cambridge University Press, 1998.

[9] C. E. Cummings, «Clock Domain Crossing Design & Verification Techniques Using System Verilog,» i *SNUG* , Boston, 2008.

[10] “Electrical Engineering Support and Information Wiki,” [Online]. Available: https://filebox.ece.vt.edu/~athanas/4514/ledadoc/html/pol_cdc.html. [Accessed 18 03 2015].

[11] P. Horrowitz og W. Hill, The Art of Electronics, New York: Cambridge University Press, 2015.

[12] Maxim Integrated, “Jitter Specification Made Easy: A Heuristic Discussion of Fibre Channel and Gigabit Ethernet Methods,” 04 2008. [Online]. Available: http://pdfserv.maximintegrated.com/en/an/AN377.pdf. [Accessed 05 Mai 2015].

[13] Cadence Design Systems, Inc., «Clock Domain Crossing: Closing the Loop on Clock Domain Functional Implementation Problems,» San Jose, CA, 2004.

[14] Red Pitaya, “Red Pitaya Wiki: User Manual,” [Online]. Available: http://wiki.redpitaya.com/index.php?title=User_Manual. [Accessed 05 05 2015].

Appendix B Abbreviations

CDC	Clock Domain Crossing
ELF	Executable and linking format file
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
IP Core	Intellectual Property Core
I2C	Inter-Integrated-Circuit
JTAG UART	Joint Access Test Group Universal Asynchronous Receiver Transmitter
MSB	Most Significant Bit
MTBF	Mean time between failures
PCB	Printed Circuit Board
PLL	Phase-Locked Loop
RF	Radio Frequency
RISC	Reduced Instruction Set Architecture
SMA	SubMiniature Version A (A Connector Type)
SOPC	System on a programmable chip

Appendix C Glossary

Asynchronous Signal: A signal whose value changes independently from the clock of the logic capturing the signal.

Clock Domain Crossing: A clock domain crossing occurs whenever data is transferred from a flip-flop driven by one clock to a flip-flop driven by another clock. [7]

Clock Jitter: The time deviation from the ideal timing of a clock transition. [12]

Flip-Flop: A flip-flop copies the input D to the output Q on the active edge of the clock that drives it. [6]

Hard Processor: A processor which is implemented on the silicon die.

I2C Interface Bus: A serial interface protocol. (See chapter 5.1.1 for detailed information.)

Intellectual Property Core: A reusable unit of logic that is the intellectual property of one party.

Metastability: A phenomenon that occurs due to failure to meet the timing constrains of the flip-flip flops used to implement the design, which leads to undesired and unpredictable output behaviour, where the output signal does not assume a stable logic value for an unbounded amount of time.

Mean time between failures: The predicted or average time between failures of a system during operation.

Register: See "Flip-Flop".

Soft Processor: A processor which is implemented in the FPGA fabric using logic synthesis.

***Three-edge requirement*:** To safely synchronize a CDC signal with a flip-flop synchronizer, the signal width must be at least 1.5 times the cycle width or period of the destination clock. This is known as the three-edge requirement.

Appendix D Project Managment and Control

A.1 Project Organization

This project is to be undertaken by a single person, requiring no sophisticated project organization or project form. The work of the author is, however, to be supervised by the faculty advisor who shall monitor the progress to ensure that the project comes to proper closure within the specified time margin. The role of the technical advisor will be to provide aid in case progress is compromised by practical difficulties

A.2 Progress Plan

Week	Planned Tasks
4	➢ Pre-Study Report
5	➢ Pre-Study Report: Corrections ➢ Demonstrator Type A: Metastability inducing HW ❖ Most simple Version, as in Ref. [1]
6	➢ Literature Review: Metastability & Flip-Flop Synchronizers ➢ Demonstrator Type A: HW Platform* ❖ Initiate NIOS II system ❖ Add metastability inducing HW ❖ Add I2C IP Core for External Clock Generator
7	➢ Demonstrator Type A: SW Platform* ❖ SW should detect metastability failure from HW ❖ Examine options for setting clocks from SW ❖ Examine options for generating statistics
8	No planned load for this week. (Due to exam)
9	➢ Demonstrator Type A: Metastability Solution Example
10	➢ Literature Review: Gray Encoding and asynch. FIFO
11	No planned load for this week. (Recuperate from eventual delays)
12	➢ Demonstrator Type A: Rapport Writing
13	➢ Demonstrator Type B: Initial Setup ❖ Understand use of PRSB gen & checker ❖ HW/SW platform for PLL /results
14	➢ Demonstrator Type B: Approach 1 to CDC ❖ Moderate rate data transfer w/ control signal
15	➢ Demonstrator Type B: Approach 1 to CDC ❖ Remove synchronizers and document effects
16	No planned load for this week. (Recuperate from eventual delays)
17	➢ Demonstrator Type B: Approach 2 to CDC ❖ Fast data transfer w/ control signal and FIFO
18	➢ Demonstrator Type B: Approach 2 to CDC
19	➢ Rapport Writing
20	➢ Rapport Writing
21	No planned load for this week. (Due to exam)

Table 1 Progress Plan

*Tasks likely to require overlapping

A.3 Progress Plan Update from Week 13

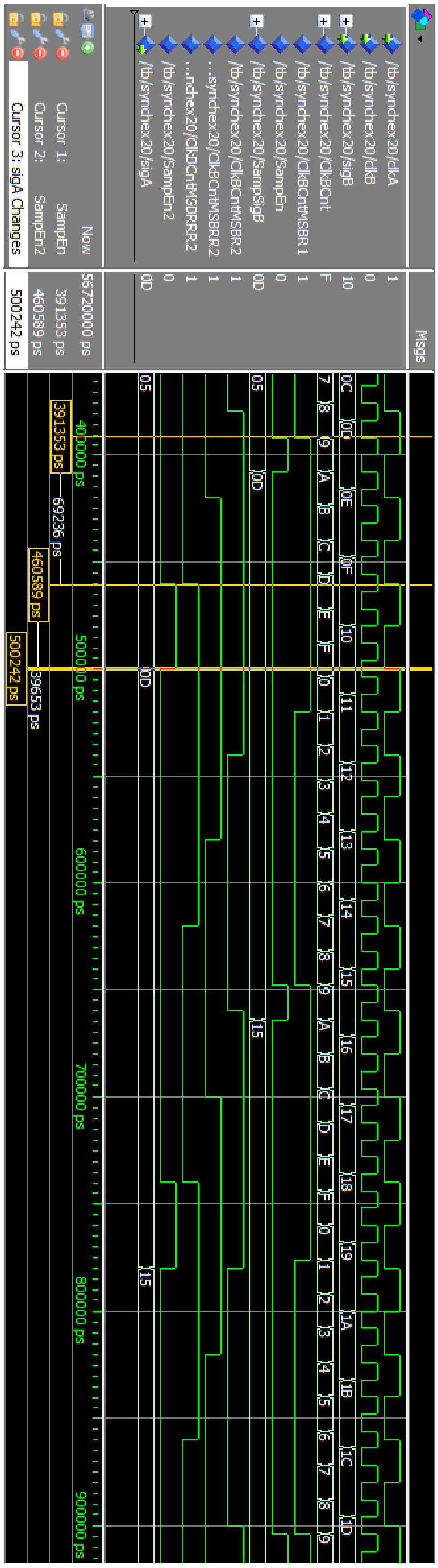
Week	Planned Tasks

13	➢ Demonstrator Type A: Testing
14	➢ Demonstrator Type A: Rapport Writing
15	No planned load for this week. (Recuperate from eventual delays)
16	➢ Demonstrator Type B: Initial Setup
17	➢ Demonstrator Type B: Approach 1 to CDC ➢ Moderate rate data transfer w/ control signal
18	No planned load for this week. (Recuperate from eventual delays)
19	➢ Demonstrator Type B: Approach 2 to CDC ➢ Fast data transfer w/ control signal and FIFO
20	➢ Demonstrator Type B: Approach 2 to CDC
21	➢ Rapport Writing

A.4 Risks

RISK	DESCRIPTION/COMMENT	SOLUTION
INADEQUATE WORK LOAD	Unrealistic expectations regarding amount of work that can be achieved.	Reduce work load. Options: • Eliminate either Approach 1 or Approach 2 from Demonstrator Type B. • Eliminate Demonstrator Type B alltogether.
ILLNESS	Would impact amount of work that can be achieved	Same as specified above.
DEFECTIVE CLOCK GENERATOR	Would compromise schedule / how early demonstrators can be programmed onto the evaluation board	Options: • Order a new PCB. Select an expensive shipping option. • Fabricate a cheap oscillator circuit at the lab.

Appendix E Simulation of SynchEx2 on ModelSim



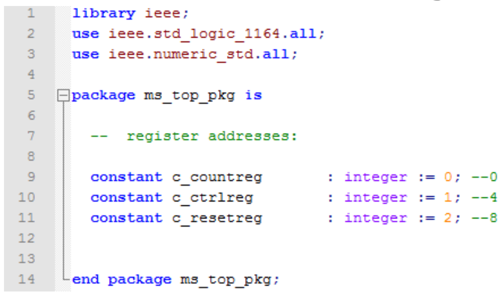
Signal	Msgs
/tb/synchex20/clkA	1
/tb/synchex20/clkB	0
/tb/synchex20/sigB	10
/tb/synchex20/ClkBCnt	F
/tb/synchex20/ClkBCntMSBR1	1
/tb/synchex20/SampEn	0
/tb/synchex20/SampSigB	0D
/tb/synchex20/ClkBCntMSBR2	1
...synchex20/ClkBCntMSBRR2	1
...nchex20/ClkBCntMSBRRR2	1
/tb/synchex20/SampEn2	0
/tb/synchex20/sigA	0D

Now	5672000 ps
Cursor 1: SampEn	391353 ps
Cursor 2: SampEn2	460589 ps
Cursor 3: sigA Changes	500242 ps

Appendix F HDL for the Metastability Test Component

The hardware of the metastability test component consists of one package and three design units. The HDL files will now be presented in hierarchical order.

A.1 Address package: ms_top_pkg.vhd



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package ms_top_pkg is

  --  register addresses:

  constant c_countreg       : integer := 0; --0
  constant c_ctrlreg        : integer := 1; --4
  constant c_resetreg       : integer := 2; --8


end package ms_top_pkg;
```

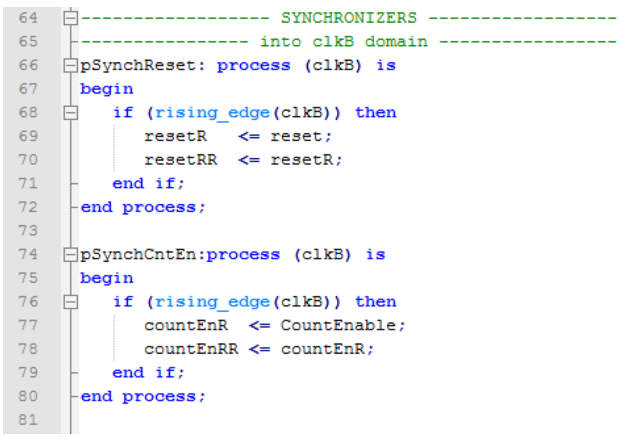
A.2 Avalon Bus Handler: ms_top.vhd

```vhdl
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

LIBRARY work;
use work.ms_top_pkg.all;

entity ms_top is
  port(
  -- For Avalon Bus Interface
   clk          : in std_logic;
   reset_n      : in std_logic;
   addr         : in std_logic_vector(3 downto 0);
   cs_n         : in std_logic;
   read_n       : in std_logic;
   din          : in std_logic_vector(31 downto 0);
   write_n      : in std_logic;
   dout         : out std_logic_vector(31 downto 0);
   -- For Clock Domain Crossing
   clkA         : in std_logic;
   clkB         : in std_logic;
   -- For Testbench
   startCmd_tb : out std_logic;
   resetCmd_tb : out std_logic
  );
end ms_top;
```

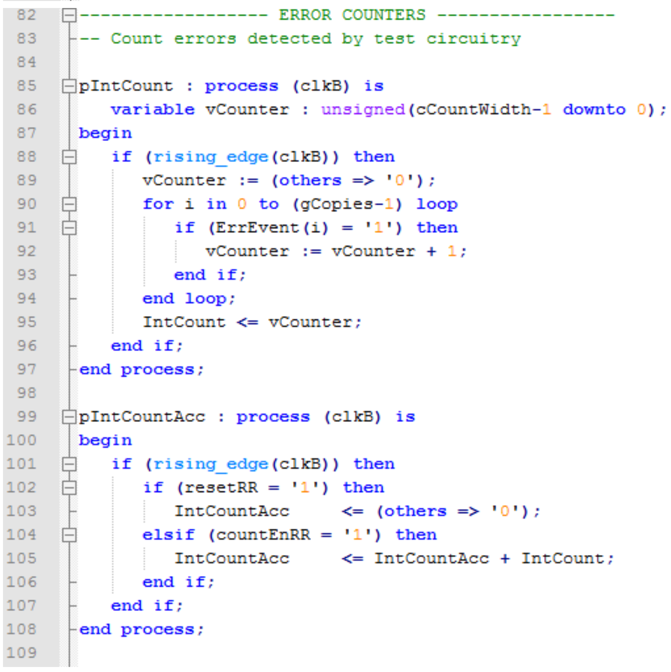
```vhdl
architecture rtl of ms_top is

signal CountReg : std_logic_vector(31 downto 0) := (others => '0');
signal CtrlReg  : std_logic := '0';
signal resetCmd : std_logic := '0';

alias  startCmd : std_logic is CtrlReg;


begin

startCmd_tb <= startCmd;
resetCmd_tb <= resetCmd;

 metastab_hw_count: entity work.ms_fcount
generic map (gCopies    =>  1)
port map    (clk         => clk,
             clkA        => clkA,
             clkB        => clkB,
             reset       => resetCmd,
             CountEnable => startCmd,
             fCount      => CountReg);


  p_avalon_interface: process(clk,reset_n)
  begin
    if (reset_n = '0') then
     CtrlReg     <= '0';
     dout        <= (others => '0');
    elsif rising_edge(clk) then
     resetCmd    <= '0';
     if (cs_n = '0') then
        if (write_n = '0') then
          case to_integer(unsigned(addr)) is
            when c_ctrlreg          =>  CtrlReg  <= din(31);
            when c_resetreg         =>  resetCmd <= '1';
            when others             =>  null;
          end case;
        elsif (read_n = '0') then
          case to_integer(unsigned(addr)) is
            when c_countreg         =>  dout  <= CountReg;
            when c_ctrlreg          =>  dout  <= CtrlReg & "000" & x"0000000" ;
            when others             =>  dout  <= (others => '0');
          end case;
        end if;
      end if;
    end if;
  end process p_avalon_interface;


end rtl;
```

A.3 Counters and CDC: ms_fcount.vhd

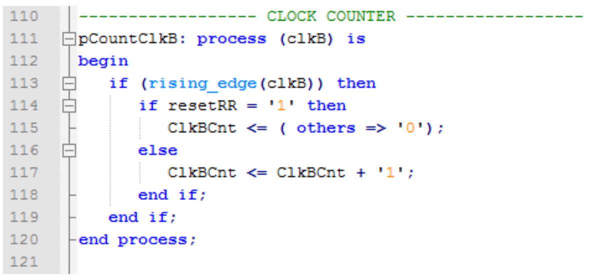
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity ms_fcount is
   generic (
      gCopies      : integer := 1000
   );
   port    (
      clk           : in std_logic;
      clkA          : in std_logic;
      clkB          : in std_logic;
      reset         : in std_logic;
      CountEnable   : in std_logic;
      fCount        : out std_logic_vector(31 downto 0) := (others => '0')
   );
end ms_fcount;

architecture rtl of ms_fcount is

-- Synchronizer Signals (clk to clkB)
signal resetR          : std_logic;
signal resetRR         : std_logic;
signal countEnR        : std_logic;
signal countEnRR       : std_logic;

-- Counter Constants and Signals
constant cCountWidth   : natural := 4;
signal ErrEvent        : std_logic_vector(gCopies-1 downto 0) := (others => '0');
signal IntCount        : unsigned(cCountWidth-1 downto 0) := (others => '0');
signal IntCountAcc     : unsigned(31 downto 0) := (others => '0');

-- Clock Counter Signals
signal ClkBCnt         : std_logic_vector(3 downto 0);

-- Sampler 1 Alias and Signals
alias  ClkBCntMSB      : std_logic is ClkBCnt(ClkBCnt'left);
signal ClkBCntMSBR1    : std_logic;
signal SampAccEn       : std_logic;
signal SampAccB        : unsigned(31 downto 0) := (others => '0');

-- Synchronizer Signals (clkB to clk)
signal ClkBCntMSBR2    : std_logic;
signal ClkBCntMSBRR2   : std_logic;
signal ClkBCntMSBRRR2  : std_logic;

-- Sampler 2 Signals
signal SampEn          : std_logic;


begin
------------- TEST CIRCUIT GENERATION -------------
MhwBuild : for i in 0 to (gCopies-1) generate
   begin
      metastab_hw : entity work.ms_hw
      port map (
         clkA   => clkA,
         clkB   => clkB,
         errB   => ErrEvent(i)
      );
   end generate MhwBuild;

```



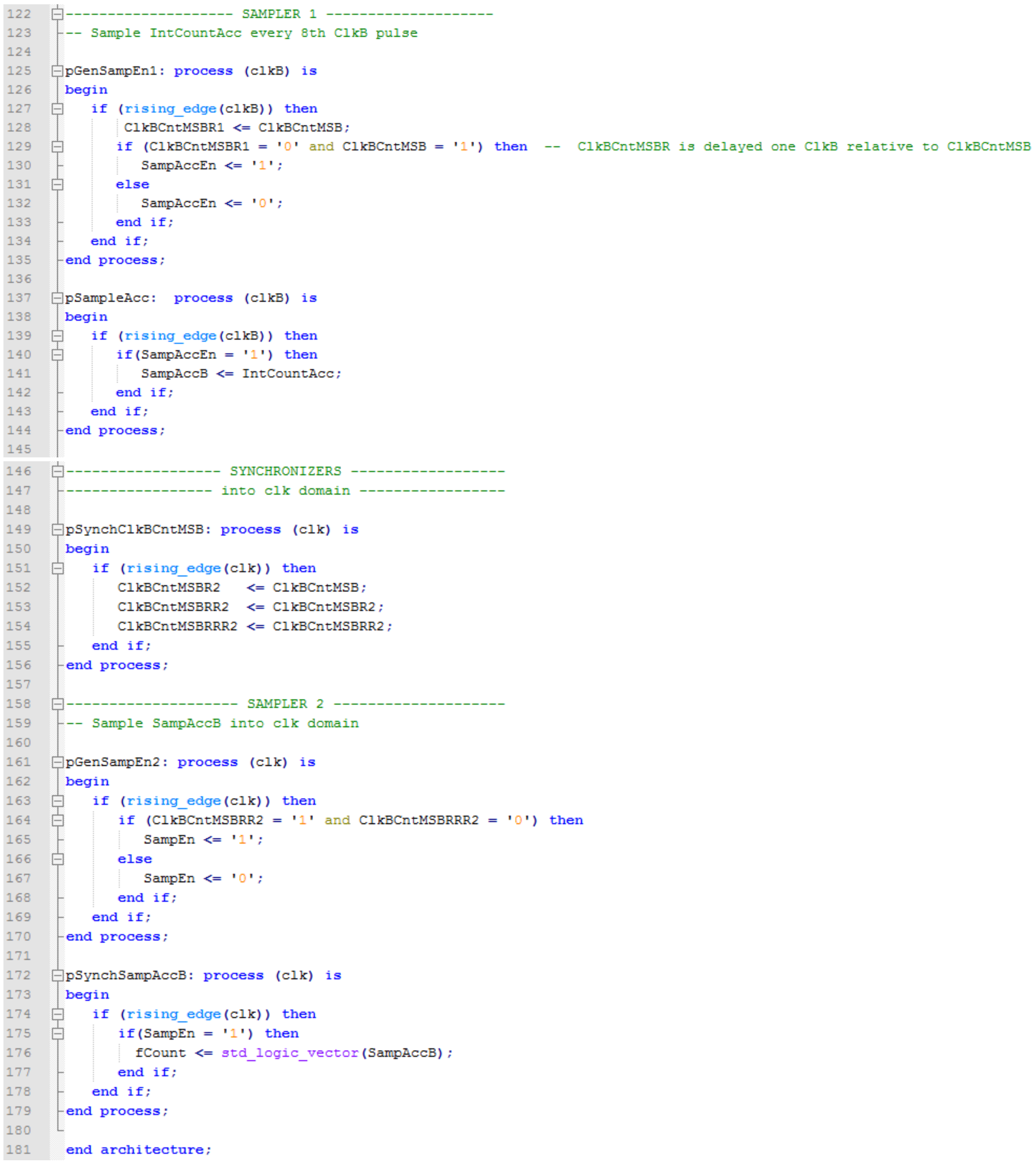
```vhdl
------------------ SYNCHRONIZERS ------------------
---------------- into clkB domain -----------------
pSynchReset: process (clkB) is
begin
   if (rising_edge(clkB)) then
      resetR   <= reset;
      resetRR  <= resetR;
   end if;
end process;

pSynchCntEn:process (clkB) is
begin
   if (rising_edge(clkB)) then
      countEnR  <= CountEnable;
      countEnRR <= countEnR;
   end if;
end process;

```



```vhdl
------------------ ERROR COUNTERS -----------------
-- Count errors detected by test circuitry

pIntCount : process (clkB) is
   variable vCounter : unsigned(cCountWidth-1 downto 0);
begin
   if (rising_edge(clkB)) then
      vCounter := (others => '0');
      for i in 0 to (gCopies-1) loop
         if (ErrEvent(i) = '1') then
            vCounter := vCounter + 1;
         end if;
      end loop;
      IntCount <= vCounter;
   end if;
end process;

pIntCountAcc : process (clkB) is
begin
   if (rising_edge(clkB)) then
      if (resetRR = '1') then
         IntCountAcc     <= (others => '0');
      elsif (countEnRR = '1') then
         IntCountAcc     <= IntCountAcc + IntCount;
      end if;
   end if;
end process;

```

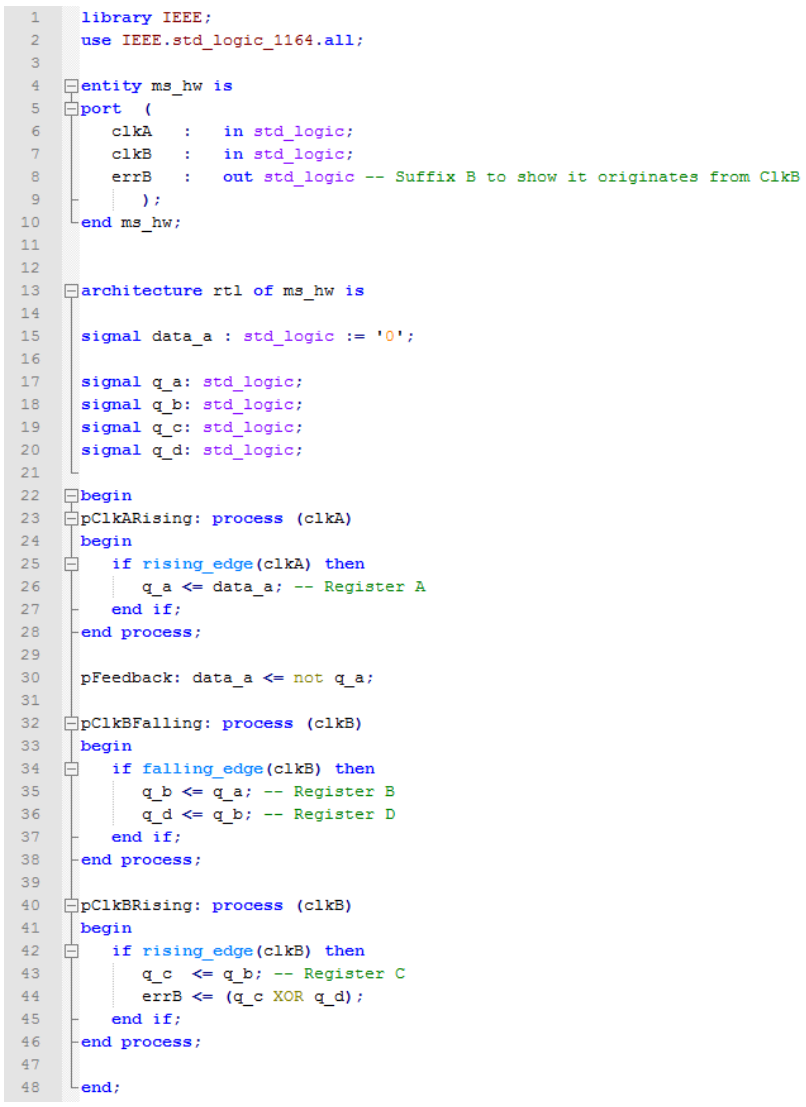


```vhdl
------------------ CLOCK COUNTER ------------------
pCountClkB: process (clkB) is
begin
   if (rising_edge(clkB)) then
      if resetRR = '1' then
         ClkBCnt <= ( others => '0');
      else
         ClkBCnt <= ClkBCnt + '1';
      end if;
   end if;
end process;

```



```
-------------------- SAMPLER 1 --------------------
-- Sample IntCountAcc every 8th ClkB pulse

pGenSampEn1: process (clkB) is
begin
   if (rising_edge(clkB)) then
      ClkBCntMSBR1 <= ClkBCntMSB;
      if (ClkBCntMSBR1 = '0' and ClkBCntMSB = '1') then  --  ClkBCntMSBR is delayed one ClkB relative to ClkBCntMSB
         SampAccEn <= '1';
      else
         SampAccEn <= '0';
      end if;
   end if;
end process;

pSampleAcc:  process (clkB) is
begin
   if (rising_edge(clkB)) then
      if(SampAccEn = '1') then
         SampAccB <= IntCountAcc;
      end if;
   end if;
end process;

------------------ SYNCHRONIZERS ------------------
----------------- into clk domain -----------------

pSynchClkBCntMSB: process (clk) is
begin
   if (rising_edge(clk)) then
      ClkBCntMSBR2    <= ClkBCntMSB;
      ClkBCntMSBRR2   <= ClkBCntMSBR2;
      ClkBCntMSBRRR2 <= ClkBCntMSBRR2;
   end if;
end process;

------------------- SAMPLER 2 --------------------
-- Sample SampAccB into clk domain

pGenSampEn2: process (clk) is
begin
   if (rising_edge(clk)) then
      if (ClkBCntMSBRR2 = '1' and ClkBCntMSBRRR2 = '0') then
         SampEn <= '1';
      else
         SampEn <= '0';
      end if;
   end if;
end process;

pSynchSampAccB: process (clk) is
begin
   if (rising_edge(clk)) then
      if(SampEn = '1') then
        fCount <= std_logic_vector(SampAccB);
      end if;
   end if;
end process;

end architecture;
```

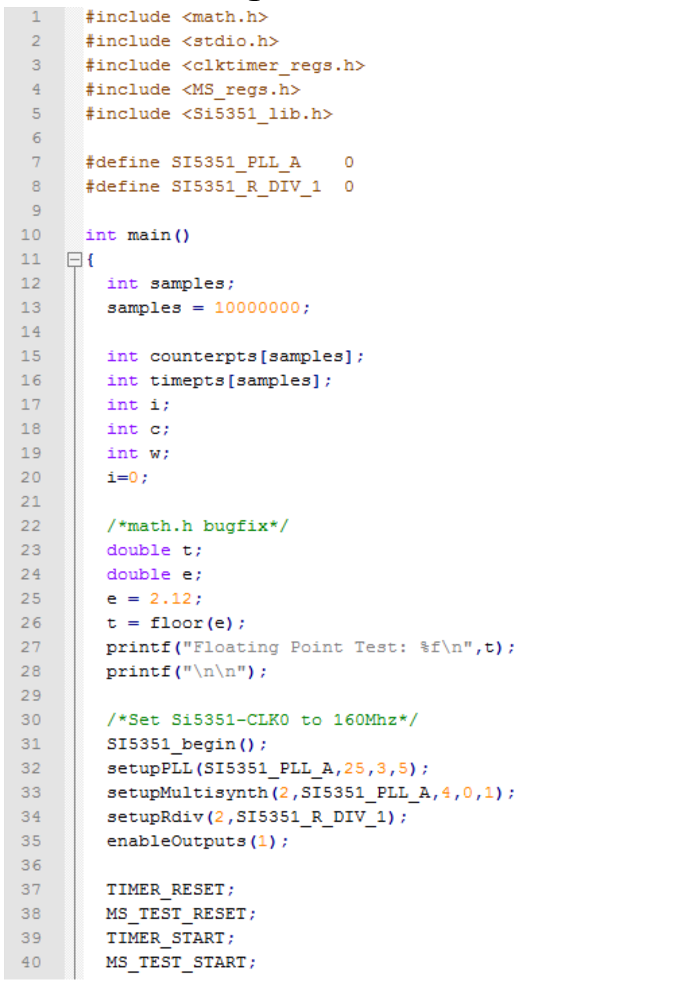
A.4 Metastability Inducing Circuit: ms_hw.vhd



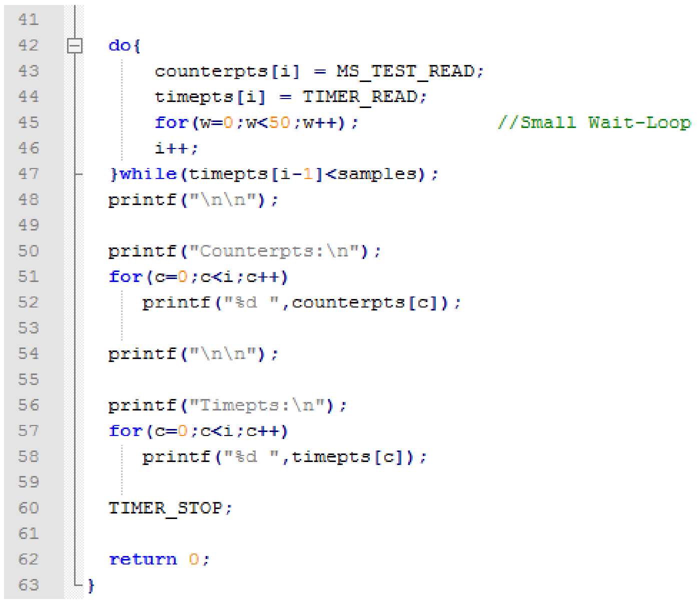
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity ms_hw is
port   (
    clkA    :    in std_logic;
    clkB    :    in std_logic;
    errB    :    out std_logic -- Suffix B to show it originates from ClkB
       );
end ms_hw;


architecture rtl of ms_hw is

signal data_a : std_logic := '0';

signal q_a: std_logic;
signal q_b: std_logic;
signal q_c: std_logic;
signal q_d: std_logic;

begin
pClkARising: process (clkA)
begin
    if rising_edge(clkA) then
       q_a <= data_a; -- Register A
    end if;
end process;

pFeedback: data_a <= not q_a;

pClkBFalling: process (clkB)
begin
    if falling_edge(clkB) then
       q_b <= q_a; -- Register B
       q_d <= q_b; -- Register D
    end if;
end process;

pClkBRising: process (clkB)
begin
    if rising_edge(clkB) then
       q_c  <= q_b; -- Register C
       errB <= (q_c XOR q_d);
    end if;
end process;

end;
```

Appendix G Test Software

A.1 Single test.

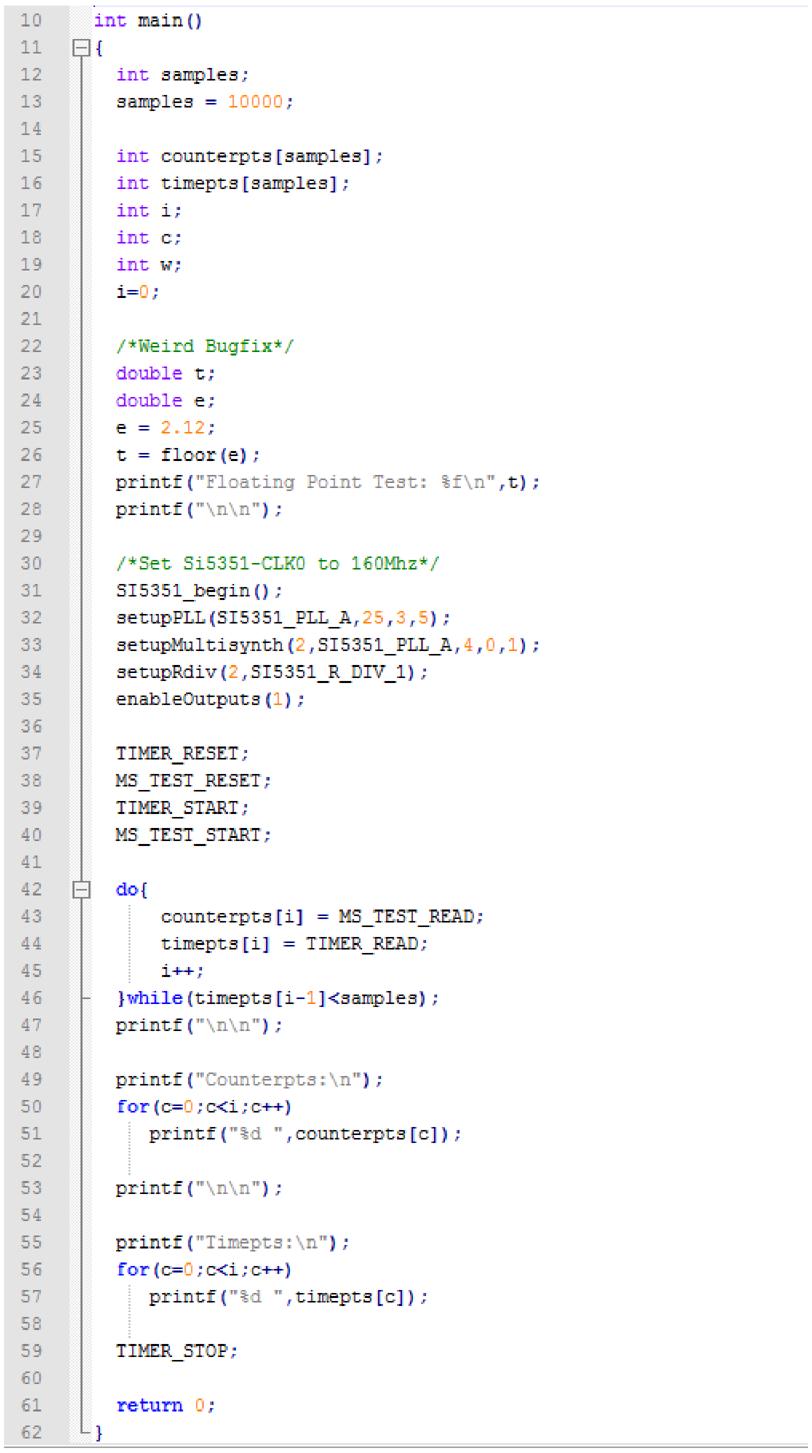


```c
#include <math.h>
#include <stdio.h>
#include <clktimer_regs.h>
#include <MS_regs.h>
#include <Si5351_lib.h>

#define SI5351_PLL_A    0
#define SI5351_R_DIV_1  0

int main()
{
    int samples;
    samples = 10000000;

    int counterpts[samples];
    int timepts[samples];
    int i;
    int c;
    int w;
    i=0;

    /*math.h bugfix*/
    double t;
    double e;
    e = 2.12;
    t = floor(e);
    printf("Floating Point Test: %f\n",t);
    printf("\n\n");

    /*Set Si5351-CLK0 to 160Mhz*/
    SI5351_begin();
    setupPLL(SI5351_PLL_A,25,3,5);
    setupMultisynth(2,SI5351_PLL_A,4,0,1);
    setupRdiv(2,SI5351_R_DIV_1);
    enableOutputs(1);

    TIMER_RESET;
    MS_TEST_RESET;
    TIMER_START;
    MS_TEST_START;
```



```c

    do{
        counterpts[i] = MS_TEST_READ;
        timepts[i] = TIMER_READ;
        for(w=0;w<50;w++);           //Small Wait-Loop
        i++;
    }while(timepts[i-1]<samples);
    printf("\n\n");

    printf("Counterpts:\n");
    for(c=0;c<i;c++)
        printf("%d ",counterpts[c]);

    printf("\n\n");

    printf("Timepts:\n");
    for(c=0;c<i;c++)
        printf("%d ",timepts[c]);

    TIMER_STOP;

    return 0;
}
```

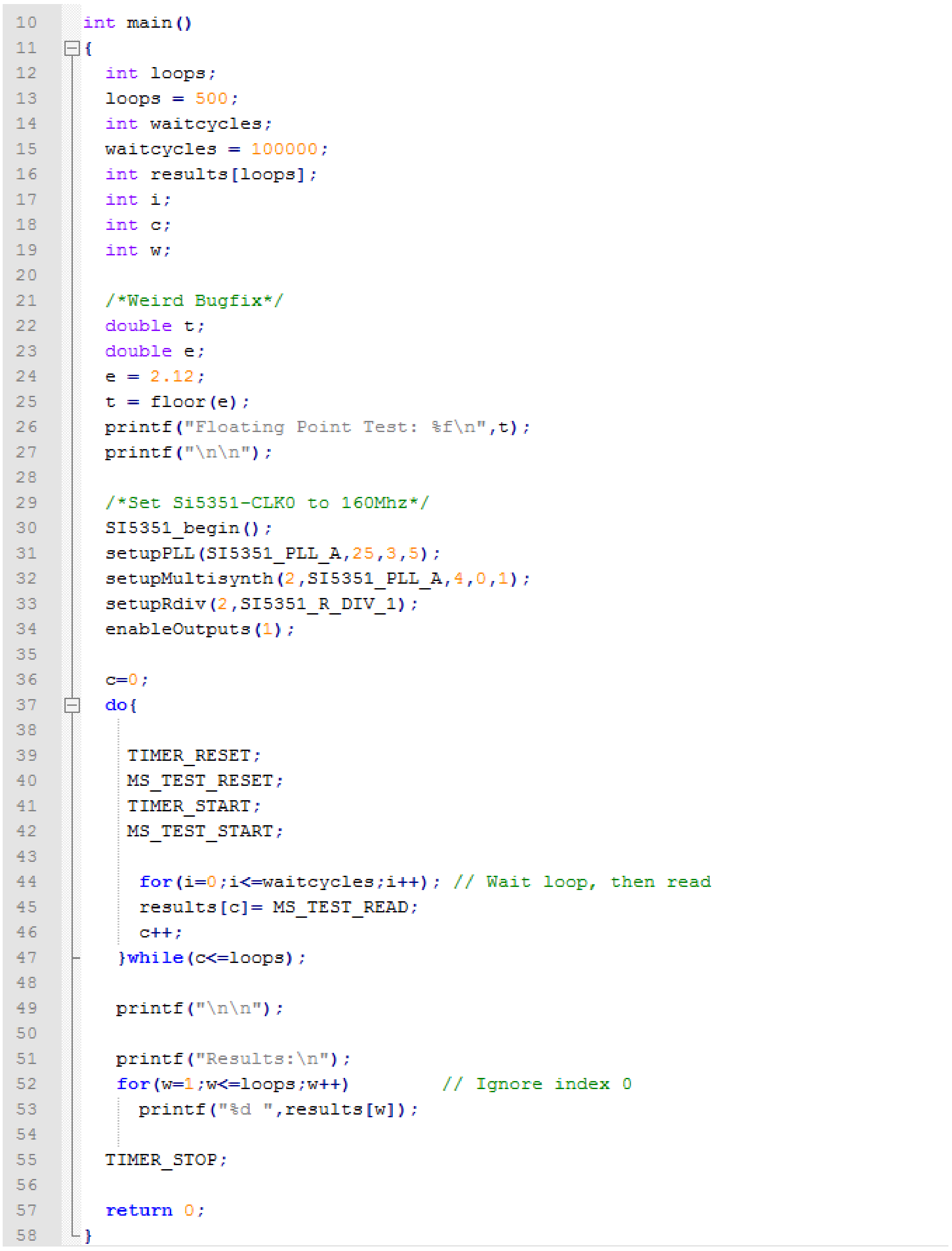
A.2 Single metastable event burst.



```c
int main()
{
    int samples;
    samples = 10000;

    int counterpts[samples];
    int timepts[samples];
    int i;
    int c;
    int w;
    i=0;

    /*Weird Bugfix*/
    double t;
    double e;
    e = 2.12;
    t = floor(e);
    printf("Floating Point Test: %f\n",t);
    printf("\n\n");

    /*Set Si5351-CLK0 to 160Mhz*/
    SI5351_begin();
    setupPLL(SI5351_PLL_A,25,3,5);
    setupMultisynth(2,SI5351_PLL_A,4,0,1);
    setupRdiv(2,SI5351_R_DIV_1);
    enableOutputs(1);

    TIMER_RESET;
    MS_TEST_RESET;
    TIMER_START;
    MS_TEST_START;

    do{
        counterpts[i] = MS_TEST_READ;
        timepts[i] = TIMER_READ;
        i++;
    }while(timepts[i-1]<samples);
    printf("\n\n");

    printf("Counterpts:\n");
    for(c=0;c<i;c++)
        printf("%d ",counterpts[c]);

    printf("\n\n");

    printf("Timepts:\n");
    for(c=0;c<i;c++)
        printf("%d ",timepts[c]);

    TIMER_STOP;

    return 0;
}
```

A.3 Five hundred tests.

Collecting only the final value of the metastability event counter. The results from six such tests were concatenated to produce the data for the histogram in figure 38.



```
int main()
{
    int loops;
    loops = 500;
    int waitcycles;
    waitcycles = 100000;
    int results[loops];
    int i;
    int c;
    int w;

    /*Weird Bugfix*/
    double t;
    double e;
    e = 2.12;
    t = floor(e);
    printf("Floating Point Test: %f\n",t);
    printf("\n\n");

    /*Set Si5351-CLK0 to 160Mhz*/
    SI5351_begin();
    setupPLL(SI5351_PLL_A,25,3,5);
    setupMultisynth(2,SI5351_PLL_A,4,0,1);
    setupRdiv(2,SI5351_R_DIV_1);
    enableOutputs(1);

    c=0;
    do{

        TIMER_RESET;
        MS_TEST_RESET;
        TIMER_START;
        MS_TEST_START;

         for(i=0;i<=waitcycles;i++); // Wait loop, then read
         results[c]= MS_TEST_READ;
         c++;
      }while(c<=loops);

     printf("\n\n");

     printf("Results:\n");
     for(w=1;w<=loops;w++)         // Ignore index 0
         printf("%d ",results[w]);

    TIMER_STOP;

    return 0;
}
```