Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Torgeir Leithe

Recreating lab measurements as RTL stimulus

Master's thesis in Electronic Systems Design
Supervisor: Kjetil Svarstad
June 2019

NTNU
Norwegian University of Science and Technology

Torgeir Leithe

Recreating lab measurements as RTL stimulus

Master's thesis in Electronic Systems Design
Supervisor: Kjetil Svarstad
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

Most of modern IC design verification is conducted as RTL simulations. However, the final verification steps must be done on physical silicon. If unexpected behavior is found at this stage, understanding the problem can be a challenge. The understanding of the problem is limited by all the internal signals and states in the DUT being hidden. Only the external signals can be probed. By recording all the stimulus applied to the IC, the conditions triggering the issue can be recreated in the RTL simulations. This enables a greater understanding of the inner workings of the IC at the time the issue occurs.

This thesis investigates ways of assisting the designer in this process. By taking advantage of most stimulus applied to the DUT being part of a communication protocol, focus can be moved from the signals them self, to the information transmitted by the signals. Decoding the signals to a protocol level allows for reuse of existing testbenches. The information being presented at a protocol level also makes analysis of the problem simpler for the designer.

Assuming a UVM testbench for the design already exist, the workflow proposed in this thesis can be implemented with limited effort from the designer. Waveform files recorded with a logic analyzer are decoded with easy to configure python scripts controlling protocol decoders from the free/open source project Sigrok. Examples are presented on how to adapt an existing testbench to generate stimulus based on the decoded waveforms.

Sammendrag

Mesteparten av moderne IC verifikasjon utføres som RTL simuleringer. Imidlertid må de endelige verifikasjonstegene alltid gjennomføres på fysisk silisium. Hvis uventet oppførsel blir avdekket på dette stadiet, kan forståelse av problemet bli en utfordring. Forståelsen av problemet er begrenset av at alle interne signaler og tilstander i DUT ikke er tiljengelig. Bare de eksterne signalene kan måles. Ved å gjøre opptak av all stimulus som påtrykkes ICen, kan betingelsene som utløser problemet gjenskapes i RTL-simuleringene. Dette muliggjør en større forståelse av ICens indre mekanismer på det tidspunkt problemet oppstår.

Denne oppgaven undersøker måter å bistå designeren på i denne prosessen. Ved å utnytte at mesteparten av stimulus som påtrykkes DUT er en del av en kommunikasjonsprotokoll, kan fokus flyttes fra de individuelle signalene, til informasjonen som blir overført av disse signalene. Dekoding av signalene til et protokollnivå muliggjør gjenbruk av eksisterende testbenker. Det at informasjonen presenteres på et protokollnivå gjør også analyse av problemet enklere for designeren.

Forutsatt at en UVM testbenk for designet allerede eksisterer, kan arbeidsflyten som foreslås i denne oppgaven implementeres med begrenset innsats fra designeren. Waveform-filer tatt opp med en logikkanalysator dekodes med enkle python-skript som kontrollerer protokolldekodere fra åpen kildekode-prosjektet Sigrok. Eksempler presenteres på hvordan man tilpasser en eksisterende testbenk til å generere stimulans basert på dekodede Waveform-filer.

Preface

This thesis concludes a 2-year Master of Science degree at the Department of Electronic Systems at the Norwegian University of Science and Technology. The work was conducted during the first half of 2019.

I would like to thank Nordic Semiconductor for proposing an interesting and challenging problem. Especially I give my thanks to my two supervisors at Nordic semiconductor, Isael Diaz and Christoffer Ramstad, for their valuable guidance and input to refining the work. I would also like to thank Kjetil Svarstad, my supervisor at NTNU, for his guidance and good advice.

-Torgeir Leithe

Contents

Abstract **I**

Sammendrag **III**

Preface **V**

List of Figures **IX**

List of Listings **X**

Abbreviations **XI**

Definitions **XI**

1 Introduction **1**
1.1 The problem . . . 1
1.2 Project goals . . . 1
1.3 Thesis outline . . . 2

2 Theory and Background **3**
2.1 Data sampling . . . 3
2.2 UVM . . . 4
2.2.1 UVM Driver . . . 5
2.2.2 UVM Sequencer . . . 5
2.2.3 UVM Sequence . . . 6
2.2.4 UVM Sequence item . . . 6
2.3 Sigrok . . . 7

3 System requirements **9**
3.1 Signal handling . . . 9
3.1.1 Unavoidable limitations of logic analyzer and the analog world . . . 9
3.2 Rawdata playback . . . 10
3.2.1 Bidirectional signals . . . 11
3.2.2 Master/slave . . . 11
3.3 Protocol decoding . . . 12
3.3.1 Advantages of protocol decoding . . . 12
3.3.2 Limitations of protocol decoding . . . 13
3.3.3 Advanced protocol decoding techniques . . . 14
3.4 Creating the testbench . . . 15

3.5 DUT state . . . 15

4 System implementation **17**
4.1 Choosing a path . . . 17
4.2 Protocol decoder . . . 18
4.2.1 Metadata . . . 18
4.2.2 Sigrok . . . 19
4.2.3 Sigrok-cli . . . 19
4.2.3.1 Timing . . . 20
4.2.3.2 Data content . . . 20
4.2.4 Stacking decoder . . . 21
4.2.4.1 making the stacking decoder . . . 23
4.2.5 Automating sigrok . . . 23
4.2.5.1 Transforming metadata . . . 25
4.3 The testbench . . . 27
4.3.1 UVM framework testbench . . . 27
4.3.2 Adapting a UVM sequence . . . 28
4.3.2.1 Verifying response . . . 29
4.3.3 Setting the starttime . . . 30
4.3.4 Configuration . . . 30
4.3.4.1 Non fixed configuration data . . . 31
4.4 Comparison of recorded and generated waveform . . . 32
4.4.1 Magnified waveform . . . 33

5 Discussion and Conclusion **35**
5.1 Rawdata or decode . . . 35
5.2 System implementation . . . 35
5.3 Achieved goals . . . 36
5.4 Further work . . . 36

Bibliography **37**

List of Figures

2.1 Adequate sampling frequency 3
2.2 Inadequate sampling frequency 4
2.3 Waveform recreated from samples 4
2.4 UVM Framework 5
2.5 Sequence item data flow [7] 6

3.1 Typical ideal waveform 13
3.2 Typical jittery waveform 13
3.3 SWD write package [11] 14
3.4 SWD wait acknowledge [11] 14

4.1 Test testbench setup 18
4.2 Waveform with protocol decoder shown in PulseView 19
4.3 Automated waveform decoding with sigrok 24
4.4 Waveform as generated in simulation 32
4.5 Waveform as recorded by logic analyzer 32
4.6 Zoomed in waveform as generated in simulation 33
4.7 Zoomed in waveform as recorded by logic analyzer 33

List of Listings

4.1 Sigrok-cli output simple command 19
4.2 Sigrok-cli output complete command 20
4.3 Sigrok-cli output for SWD 21
4.4 Sigrok-cli output with JSON encoder 22
4.5 Sigrok-cli output with JSON encoder for SWD 22
4.6 JSON_print decode function 23
4.7 TorgeSigrok usage spi 25
4.8 spidata.txt 25
4.9 configfile.txt 25
4.10 SpiSigrok 26
4.11 Translated configfile.txt 26
4.12 UVM sequence reading data from file 28
4.13 Verifying the response 29
4.14 UVM log for SEQ and SEQ-RSP 30
4.15 Driver waiting until starttime 30
4.16 Configuring the testbench 31

Abbreviations

RTL	=	Register transfer language
IC	=	Integrated circuit
DUT	=	Device under test
MCU	=	Microcontroller unit
TLM	=	Transaction-level modeling
SPI	=	Serial Peripheral Interface Bus
SWD	=	Serial Wire Debug
UVC	=	UVM Verification Component
UVM	=	Universal Verification Methodology
UVMF	=	UVM Framework

Definitions

How terms are used in this report

Signal: The information conveyed by a single data line. In the physical world this is the voltage in a wire or net. In the digital domain it is a single net that is either high or low.

Waveform: A representation of the value of one or multiple signals, at all times between a start and end time.

Raw data: A unaltered waveform, as it was captured by a logic analyzer.

Protocol decoding: The act of changing the raw data into another representation, based on the rules defined by a protocol.

1 Introduction

With the ever-rising complexity of modern Integrated circuit designs, more and more effort is needed to maintain the desired quality of verification.[1] As the speed and power requirements of ICs increase, the resulting rise in complexity is a huge challenge when verifying the design functionality. "Historically, about 30% of IC/ASIC projects are able to achieve first silicon success, and most successful designs are productized on the second silicon spin[2]."

1.1 The problem

If an issue is found during testing of the physical IC, it would be beneficial to recreate the same conditions in a simulated environment. When the IC is simulated, all the internal signals of the IC can be viewed, and the designer can gain insight into the inner workings of the IC at the time when the issue occurred. For this to be achieved, the simulated IC must be provided with the exact same stimulus as was applied to the physical one. Manually programming the testbench to generate this stimulus is tedious, resource intensive and error-prone. By automating this process, the work of the designer can be simplified, and more focus moved from recreating the issue, to understanding and solving it.

1.2 Project goals

The goal for this project is to show how the process can be automated. The first step will be an extensive analysis of different strategies for turning recorded waveforms into testbench stimulus. It will look into benefits and drawbacks with different approaches to solving the problem, and propose a workflow for a design with a UVM testbench. The second part is implementing the example workflow and discussing whether it is a suitable solution. The Ideal outcome is to create a robust and universal framework that works for all scenarios.

1.3 Thesis outline

This report explores several different approaches to the solution. Chapter 2 gives a summary of the knowledge needed to understand the report. In chapter 3, several different approaches to the different parts of a complete solution are proposed. Chapter 4 explains the different steps needed to implement a solution and show the result of the implementation. Chapter 5 gives a summary of what is archived by the work and suggestions for future work.

2 Theory and Background

This chapter is based on the project report on the same subject made the previous semester. Some parts were added or removed to better fit with this thesis.

2.1 Data sampling

Physical signals are converted into digital data by a logic analyzer. Unlike an oscilloscope, that typically has a vertical resolution of 8 - 12 bits for each sample, logic analyzers are made for capturing digital signals, and therefore only need one bit resolution. Hence a logic-analyzer will only samples if a signal is above or below a threshold voltage. In order to be able to recreate the original signal from the samples, the sampling frequency must be greater than 2 times the frequency of the signal[3]. However, to ensure decent timing resolution, and in case the duty cycle of the signal is not exactly 50%, the sampling frequency should be at least 4-6 times the switching frequency of the signal. [4]

Some high-end logic analyzers have the ability to use one of the input channels as the sample clock instead of using a fixed internal clock, resulting in the sample interval being nonuniform[5]. This can significantly reduce the size of the sample data as the sampling frequency can now equal the signal frequency. The downside is that the time of the sample must be saved together with the value, this is not needed when using a fixed sampling frequency as the time for a sample is always the time for the previous sample plus the sample period.

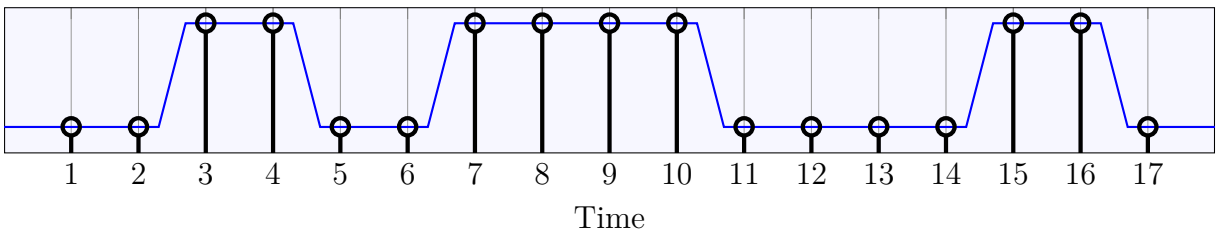


Figure 2.1: Adequate sampling frequency

Figure 2.1 show a signal sampled with a high enough sampling rate. Figure 2.2 show what can happen if the sampling rate is too low. Both sample-sets will be saved as the waveform shown in Figure 2.3. It is not possible to tell from the saved waveform if the

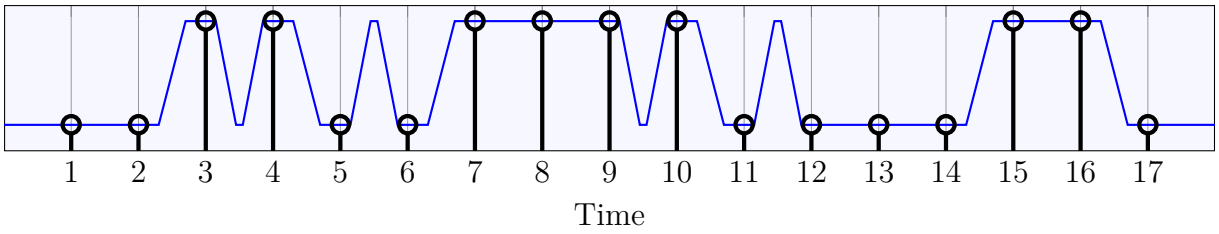


Figure 2.2: Inadequate sampling frequency

sampling frequency was sufficient. Therefore it is important to consider if the sampling rate is high enough before recording the waveform.

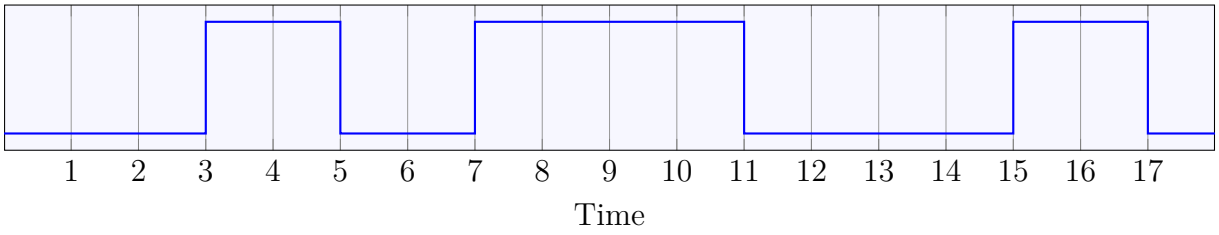


Figure 2.3: Waveform recreated from samples

Even if the sampling rate is adequate in regards to the frequency of the signal, noise/glitches on the signal might cause the sampled data to be different from the data perceived by the IC. Especially if the signal is asynchronous like an interrupt or a clock signal, it may trigger the IC without registering on the logic analyzer. A difference in threshold voltage in the IC and the logic analyzer can also cause such a discrepancy.[6]

2.2 UVM

In February 2011, the 1.0 version of UVM was approved by Accellera. UVM is an extension to SystemVerilog providing an object-oriented approach to designing testbenches. It provides a well-defined testbench structure were each component serves a single predefined purpose. Communication between the modules is transaction based, and this facilitates code reuse. An overview of a typical UVM testbench is shown in figure 2.4. The different classes are created by extending a UVM base class. This report will only focus on the parts of UVM concerned with generating stimulus.

The input data is generated by the UVM_sequence, which means that the UVM_env can stay the same, regardless of the input. This makes it simpler to take an existing simulation setup utilizing UVM, and substitute in another input sequence.

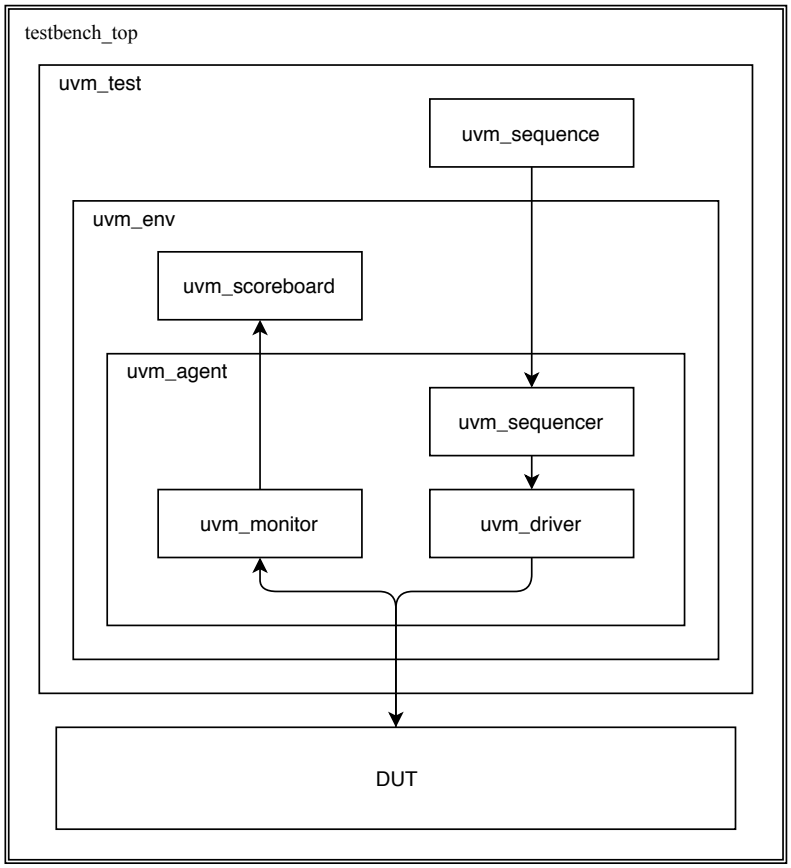


Figure 2.4: UVM Framework

2.2.1 UVM Driver

In a UVM simulation, the DUT is connected to the testbench via the UVM driver. It is the interface between the UVM sequencer and the DUT. Normally one driver exists for each interface on the DUT. The driver receives data to output to the DUT as sequence_items from the sequencer. Then it translates the sequence_item into the corresponding input signals to apply to the DUT. It can also read signals from the DUT, react to those signals, and send the response back to the sequencer.

2.2.2 UVM Sequencer

The UVM Sequencer is the transaction level arbiter for the driver. It controls the flow of sequence_items from the UVM sequences to the driver.

2.2.3 UVM Sequence

A UVM Sequence generates stimulus as a series of sequence_item and passes them to the UVM Driver through the UVM sequencer. As shown in figure 2.4, the UVM_Sequence is in the top layer of the UVM hierarchy, making it independent of the rest of the UVM setup. This makes it trivial to change the test data without altering the rest of the testbench. It is possible to make a Sequence of sequences for generating larger input data series.

2.2.4 UVM Sequence item

UVM sequence items are how information is passed between the sequence, sequencer, and driver. Each of them configured to work with a specific sequence item type. It is therefore not possible to mix sequence item types for one driver. The sequence item defines all data field that are part of the transaction, which of them can be randomized and constraining those randomizations. It can also include functions for setting, copying, comparing, and printing the values in itself. This can make it simpler for the UVM sequence to set the correct values to the sequence item, and for the driver to get the values.

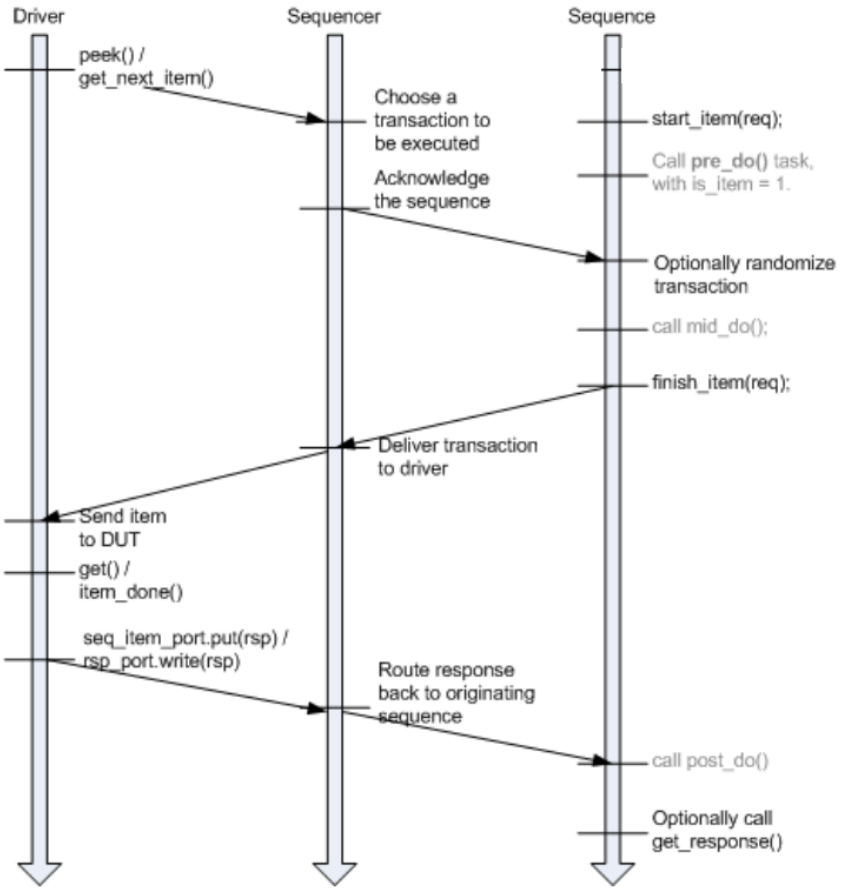


Figure 2.5: Sequence item data flow [7]

Figure 2.5 shows how the sequence, sequencer, and driver communicate using the transaction based sequence items. The graph shows how one sequence item gets requested from the sequencer by the driver. Then the sequencer calls the correct sequence, and the sequence generates (and randomizes) the sequence item before it is sent back to the driver. Finally, the driver drives the input in the sequence item to the DUT. Optionally the driver can return to the stimulus to the sequence. Typically the driver will be coded to loop this process so that it runs until the sequence has reached its endpoint, and no longer generates new sequence items.

2.3 Sigrok

The Sigrok project is a free/open source project that aims to simplify interacting with many different test instruments. One of the main products of the Sigrok project is the GUI Pulseview, which can control many different logic analyzers, display the captured waveform, and display the decoded data of those waveforms. It has support for decoding more than 100 different protocols.

Another product from the sigrok project is their command line interface, the sigrok-cli. It allows for much of the same functionality as Pulseview, but controlled from the terminal instead of the GUI. This allows for easy automating the use of sigrok, and controlling it as part of other software.[8]

3 System requirements

Performing a simulation based on stimulus captured with a logic analyzer can be split into a two-step process. The first step is to parse and adapt the data file generated by the logic analyzer so that it is usable in by the simulation environment. The second step is to create a testbench for the DUT that generates stimulus based on the adapted data. How one of the steps should function depends heavily on the functionality of the other. It is therefore ideal that a process/workflow is found that allows both steps to be performed without unnecessary complexity.

3.1 Signal handling

There are two different main paths for achieving this. One is to naively recreate every signal as the logic analyzer captured it. The other is to instead of focusing on the individual signals, focus on the information transferred by those signals. This requires decoding the waveforms in the first step and then reading the decoded waveforms in the second step. At first, this may seem like it introduces unneeded complexity to the process. However, there are several advantages by doing this, that are not instantly apparent.

Section 3.2 discusses the possible implementations for the naive path, and section 3.3 does the same for the protocol decoding path. The amount of work and challenges regarding designing and implementing testbenches for the various discussed solutions is discussed in section 3.4.

3.1.1 Unavoidable limitations of logic analyzer and the analog world

If the waveforms captured by the logic analyzer were perfectly recreated in the testbench, this would still not be a perfect recreation of the waveforms perceived by the physical DUT. In the physical world, even signals used to transfer digital information, are analog. When the logic analyzer converts the analog signals to their digital representation, both timing and signal level information is lost. Several causes for inaccuracies are mentioned in section 2.1. More details on logic analyzer inaccuracies and probing are outside the scope of this report.

Even with the inherent inaccuracies of analog ignored, an impossible ideal logic analyzer will not generate a perfect waveform file. The logic analyzer sampling clock will directly affect the timing resolution of the recreated signal. With a typical sampling rate of 50 MS/s, signals are updated by the logic analyzer every 20 ns. Therefore a signal level change in the physical signal can be shown in the generated waveform-file anywhere from 0 to 20 ns later. For a clock signal with a frequency of 5 MHz, this is up to 10 % of the clock period.

The timescale of the testbench may theoretically affect the results in a similar way to the logic analyzer clock. However, it is as default set to 1 ps[9], and can be changed to be even lower. This is so small that it for any practical purpose can be ignored as a limitation, and as long as it is a whole number divisor of the sample clock of the logic analyzer, it can be used to recreate the raw data perfectly.

These inaccuracies will equally affect all possible paths when handling the captured data. Therefore the rest of this chapter will ignore these effects. It is worth noting that due to the captured signals not being entirely accurate to begin with, there is a relatively small drawback from choosing a path that slightly alters the signals further.

3.2 Rawdata playback

Let's first consider the naive approach of replaying the data exactly as it was captured, sample for sample, without doing any decoding or analysis of it. All the signals captured by the logic analyzer are either driven by the environment around the DUT, or the DUT drives them as a response to the inputs from the test environment. Only the signals in the capture file that are driven by the environment need to be used, the signals driven by the DUT can be ignored as the simulated DUT will generate new responses. This will result in the waveform shown in the simulation being exactly as reported by the logic analyzer.

Updating the simulation for every new sample taken by the logic analyzer will negatively impact simulation performance. As shown in figure 2.1, the number of samples is often much larger than the number of value changes, resulting in many unneeded updates in the simulation. A better solution is only to update the simulation when a value change occurs.

The data format "Value change dump"(.vcd) is defined as part of the Verilog standard[10]. Instead of storing all the samples, it stores the data as a series of times where the sampled value has changed. By using .vcd or a comparable format, all the samples in figure 2.1 can be turned into just storing the value changes shown in figure 2.3. This leads to smaller files for storing the data, and far fewer signal updates when running the simulation. Playing back this file can be done in a similar way to playing back all the samples, but instead of always waiting for one sample interval and then updating all the signals, the driver must wait until the next time a signal changes, and then update the signal.

3.2.1 Bidirectional signals

Some communication protocols like I2C and SWD have both the environment and the DUT driving the same signals at different times. This is where the naive approach starts to breakdown. The testbench will have no way of knowing if it or the DUT should drive a signal at a given time. One possible solution to this problem is to utilize signal drive strengths. By setting the drive strength of the testbench to be lower than the drive strength of the DUT. The testbench will be able to drive the signal as long as DUT is not trying to drive the signal, but as soon as the DUT drives the signal, it will take priority over the testbench and overwrite anything driven by the testbench.

A typical MCU have the possibility to configure the I/O pins with either pull-up or pull-down resistors in addition to the signal level. This further complicates the signal strength solution as the signal strength of the testbench needs to be weaker than the strength of a signal from the DUT, and at the same time stronger than the pull-ups. In Verilog, there is no signal strength between the default signal strength (strong), and the level typically used for pull-ups and pull-downs (pull) [10]. This is a challenge because it may require changing of the DUT get the correct behavior in the testbench.

Another thing to consider is that even if the strength levels are configured to work correctly, timing related issues may still cause this method to not work in some instances. Take a simple request-response situation where the testbench sends a request and the DUT answers with a response on the same data line. If the DUT in the simulation takes a slightly longer time to send the response than the physical DUT did, the end of the response from the DUT will overwrite the start of the next request by the testbench. If the protocol used does not specify a strict response time, this is still valid behavior by the DUT, but will break the simulation completely due to the DUT no longer getting the next request.

3.2.2 Master/slave

So far, the considerations have worked on the assumption that the testbench is the master of the communication, driving all signals as exactly when they occurred in the logic analyzer recording. When the testbench is the master, it initiates all the communication, and the DUT responds synchronized to the masters requests. If the DUT is the master, this order is changed, and the testbench must now respond synchronized to the requests of the DUT. Unlike the scenario where the testbench is the master, this can cause issues even if the request and response do not share the same signal.

Even without the danger of one node in the communication overwriting the other, timing related issues can cause failure. For the synchronized communication in figure 3.1, there are no issues as long as the testbench generates both the clock and data signal. However, if DUT is the master, generating the clock, and the testbench is the slave, generating the data, there are some difficult challenges to overcome. Firstly some advanced analysis of the waveform file from the logic analyzer must be done in order to figure out when

the DUT should request the data. The other part of the challenge is to get the DUT to request the data at the desired time. There will often not be a direct way of doing this, as the communication interface is only one part of a larger DUT, controlling the interface. If the time delay from telling this larger DUT to request the data, until the request is sent is varying or unknown, this becomes an impossible problem to solve using the naive method.

3.3 Protocol decoding

By recognizing that most signals are used to transfer information as part of some higher level communication protocol, it is possible to move focus from the individual signals, to the information carried by those signals. To achieve this, all the signals that are part of the communication must be converted from individual signals into a single information stream.

With protocol decoding, an additional step is needed before the simulation is started. The waveforms from the logic analyzer must be analyzed and decoded to find the information transferred by the waveform. How this information should be stored will depend on the protocol, since most protocols are packet-based, the general solution is to store the data as a series of packets. The complexity of the decoding process depends on the protocol. A robust solution should be able to handle most protocols. The open source Sigrok project[8] includes protocol decoders for most common protocols and should provide a good starting point for doing the protocol decoding.

In the testbench the data is now handled at the packet level, the UVM_sequense will generate one sequence_item for each packet, it needs to contain all the information needed to recreate the original waveform. The driver in the testbench implements all the protocol specific behavior and re-encodes the data in the sequence_item before driving it to the DUT.

3.3.1 Advantages of protocol decoding

By utilising protocol decoding the key challenges discussed in section 3.2.1 and 3.2.2 are solved. Since the data is decoded from a raw data level to information level, the synchronization issues can be solved by the testbench on a protocol level. For the example mentioned in 3.2.2, where the DUT is the master generating a clock, it is no longer critical that the DUT generates the clock edges at exactly the right time. When the testbench implements the protocol behavior it can drive the bits on the correct clock edge, regardless of if the clock edge occurs at the same time as in the original recording.

Another advantage gained by protocol decoding is that it makes the designers job more straightforward. It is easier for the designer to understand what is happening in the design with the information presented at a protocol level. If this step was not automated, it would require the designer to do this as a manual step as part of the debugging process.

3.3.2 Limitations of protocol decoding

Some low-level information about the signal is lost when adding this abstraction. Consider the typical waveform in figure 3.1 and the similar waveform in figure 3.2. When doing protocol decoding with data sampling on the rising edge of the clock, both of these packets will decode to exactly the same data, even though the waveforms are not the same.

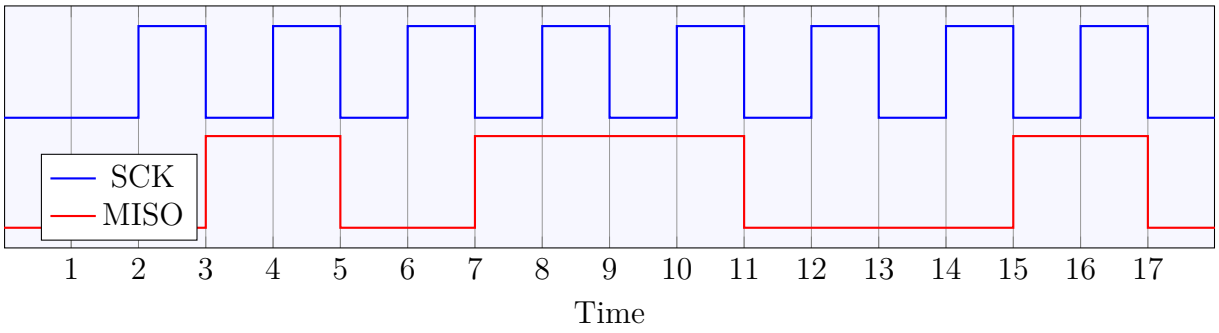


Figure 3.1: Typical ideal waveform

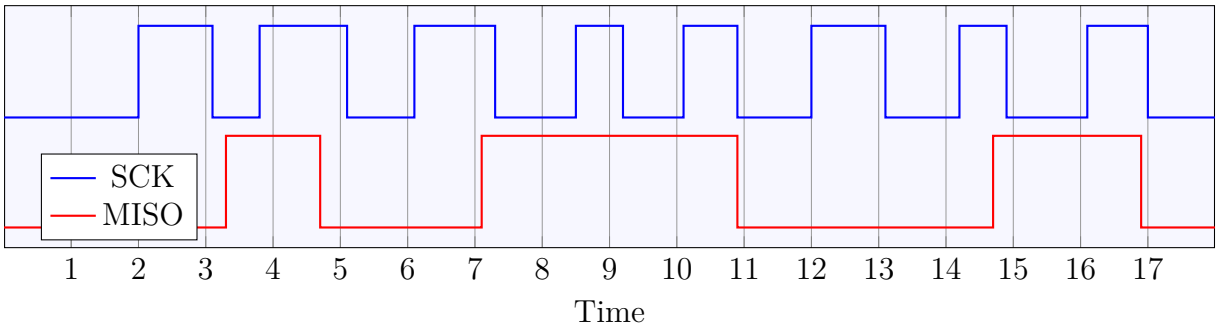


Figure 3.2: Typical jittery waveform

By storing the start time for each packet in the transaction the relative timing between the packets can be maintained correctly, there is however no simple way of doing protocol decoding while also keeping track of each individual signal change. The example in figure 3.2 is a rather extreme example, and any normal recorded waveform will be closer to the one in figure 3.1. As explained in section 3.1.1 this recorded waveform is not a 100 % correct representation of the physical waveform, therefore this loss of accuracy when decoding can be ignored in most use-cases.

The major limitation that makes protocol decoding unusable for some situations is that some signals are not part of a well-defined protocol. Signals like an asynchronous reset cannot be decoded as it is not part of any protocol. Even when the majority of signals in a recorded file will benefit from decoding, there may be some signals that can not be

decoded. In this case, it is possible to combine the two techniques, decoding the signals that are part of a protocol and keeping the others undecoded.

3.3.3 Advanced protocol decoding techniques

For more "advanced" protocols it is not enough to just decode the waveform to get the data for each packet. This is because the bus behavior changes depending on the packet type. To support this, the packet type must be found when decoding the packet. Protocols typically have the packet type information in the header field of the packet. The layout of an SWD packet is shown in figure 3.3. Just knowing the data content of the packet is not enough, as the testbench must check the value of the R/W bit to know if it or the DUT should generate the DATA field.

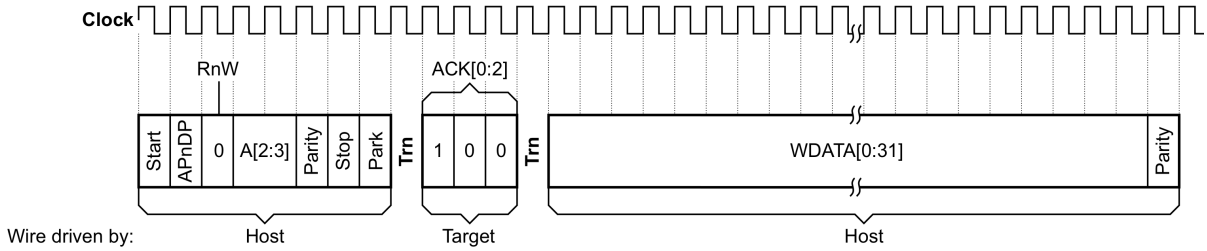


Figure 3.3: SWD write package [11]

It is possible to take the idea of transferring information rather than replaying the raw data even further by responding correctly to signals sent by the DUT. One example of this in the case of SWD could be by honoring the wait signal from the DUT. Figure 3.4 show the SWD host trying to do the same operation as in figure 3.3, but the DUT behaves differently by acknowledging with `0 1 0 (WAIT)` instead of `1 0 0 (OK)`. In this case disregarding the response from the DUT would cause the Host to try to write data to the DUT, while the DUT is not ready to receive it. For other protocols, it could be waiting for a ready signal from the DUT before sending the packet. Of course, if the test intends to verify what happens in the case where such wait signals are ignored, this logic should not be included.

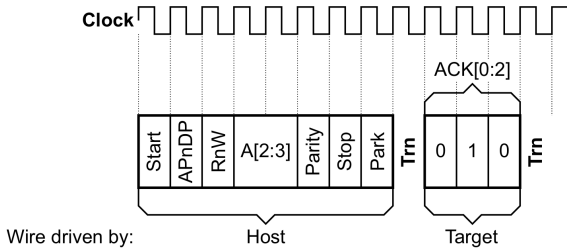


Figure 3.4: SWD wait acknowledge [11]

3.4 Creating the testbench

By the time a design has matured to the stage where tests can be performed outside of the simulated environment, either on an FPGA or as a prototype IC, the simulation testbenches have also matured to the same level. A UVM testbench for an IC will typically have one UVC for each interface on the DUT [12]. In this UVC the sequence items will consist of the information that should be transmitted by the driver. How the information is represented in the sequence_item will vary based on what is beneficial for that specific interface, it usually is at a protocol level as that makes the work of the designer simpler when design stimulus.

If the naive path is chosen, it allows for very limited reuse of the existing testbench. Since the existing UVC for the interface operates on a protocol level, it must be replaced with a completely new UVC specially designed to drive pins directly with the raw data. If this UVC is well made, it should be possible to only make it once and then reuse it between different testbenches. The drawback here is that all the verification elements of the original UVC are lost. Creating a Hybrid UVC with the stimulus driving components (sequencer, sequence, driver...) from the new UVC, and all the verification components (monitor, scoreboard, coverage...) from the original UVC, is challenging and will likely not be a universal solution.

Depending on the situation the new UVC is either used as the only component of a new standalone testbench, or it replaces the original UVC in a larger existing testbench. Which of these approaches are the best depends on if the other parts of the original testbench are required for the simulation to run correctly.

With the protocol decoding path, almost all of an existing UVC and testbench can be reused. If the UVC is well suited for adapting, only the UVM_sequence have to be replaced with one that is made for reading all the stimulus data from a file. When this sequence has been made once, it should be possible to use it as a template when adapting future UVCs, minimizing the amount of work needed to adapt the UVC. Keeping all the verification components of the original testbench makes it faster to verify if the DUT is behaving correctly.

3.5 DUT state

The goal of the simulation is to perfectly recreate how the physical IC should behave, given the captured stimulus. One requirement for this is that at the start of the simulation, the internal states are the same in the physical and simulated DUT. Some states will not be relevant for this particular simulation or are configured by the stimulus itself. Therefore ensuring that all of the states are the same might not be needed.

The simplest way to ensure this is to reset the DUT immediately before capturing the data, and doing the same when replaying it to the DUT. Making sure the reset signal is one of the signals in the captured waveform file is a simple way to ensure that the reset happens at the same time compared to the other stimulus. In some cases, this will work flawlessly. For other, more complex systems, the non-reset state of the DUT might be part of what one wants to investigate with the simulation. It is also a likely case that a part of the system the physical DUT is a part of is set to configure the DUT immediately after a reset. In both scenarios, care must be taken to ensure that the relevant states are the same at the start of the simulation.

Automating this process is likely very challenging, as what is a practical approach will vary significantly with different ICs and different interface on the same IC. In some cases, it will be possible to read the internal status registers directly using Jtag/SWD. In other cases, the only way to achieve knowledge of the state may be to keep track of all state changes in the DUT from the time of the last reset.

4 System implementation

This chapter will focus on implementing a solution to the problem presented in chapter 1, using the insights gained in chapter 3 to find and implement a universal solution. A path can be chosen based on the pros and cons of each approach discussed in chapter 3. There are obvious benefits to both paths.

4.1 Choosing a path

The major advantage of raw data playback is its simplicity. When data is never converted away from its original representation, no undesired inaccuracies are introduced. However, the multiple issues with synchronized communication are challenging to overcome. If this path is taken getting the first prototype up and running should be relatively fast, but after that, a considerable effort must be put into trying to solve those issues. They are likely not solvable for all use cases.

Protocol decoding solves those problems without introducing any new significant issues. It has the additional benefits of testbench reuse and ease of use for the designer. The main drawback is its higher upfront cost due to having to create a method for doing protocol decoding. If a universal and robust method is found, the rest of the process should be reasonably straight forward, assuming a testbench using the protocol already exists. If a testbench or UVC using the protocol does not exist, it might still be worth the effort to implement simple protocol functionality in a UVC.

Based on these considerations, it was decided to aim for an implementation primarily based on protocol decoding. If individual signals exist in the communication that are not part of a protocol, they can still be used in the same testbench by adding an additional UVC for driving undecoded signals. This UVC will be functionally similar to the ones that are for a protocol, but will not have any verification functionality. For the stimulus generating part, it will be very similar, basically defining a new protocol that is no protocol.

An overview of the proposed data flow for a given set of raw data is shown in figure 4.1. First, the data file from the logic analyzer is decoded and converted to a .csv file using a sigrok based python script. The UVM_sequence then parses this file in the testbench. Each row in the .csv file corresponding to one sequence item in the testbench.

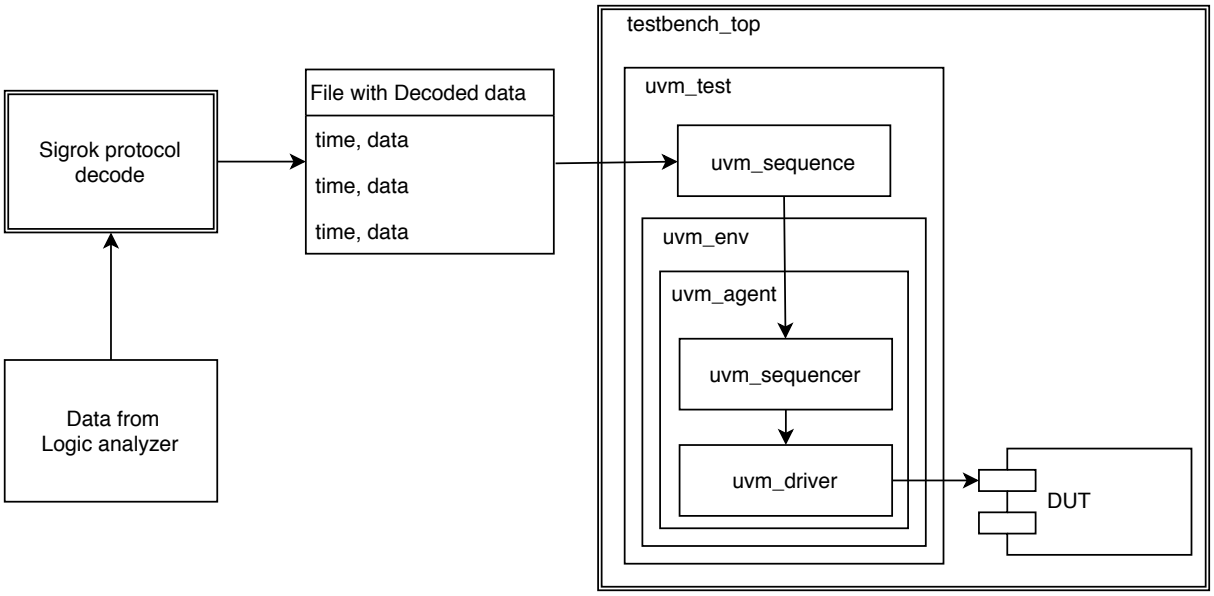


Figure 4.1: Test testbench setup

4.2 Protocol decoder

The goal for this part of the process is to take an inputfile generated by a logic analyzer and convert it into a file readable by a UVM testbench. This process should be the same regardless of whether the data in the .csv file is raw data or protocol decoded data. That might lead to an overly complex solution for the raw data case, but the benefit of only having one implementation should outweigh this drawback. The rest of this chapter will primarily focus on the situation with protocol decoding, as this is the most complex case and a good solution for it should be adaptable to the case whiteout protocol decoding.

4.2.1 Metadata

When configuring the protocol decoder, it must be provided with enough metadata to decode the waveforms correctly. This metadata must at minimum include which protocol it is, and which channel in the raw data is which signal in the protocol. In addition, some protocol specific metadata might be required. This will typically include whether signals are active high or low and if data is sampeled on rising or falling edge of the clock.

The UVM_driver also needs the same protocol specific metadata in order to recreate the waveform correctly. An excellent solution to avoid issues due to incorrectly entered metadata is only to specify the metadata one place, and use it from there both when doing protocol decoding and in the testbench. Since the metadata must be known at the time the decoded data file is generated, the best way to do this would be to at the same time save all the metadata used when decoding. Either as part of the header for the data file or as a separate configuration file.

4.2.2 Sigrok

By using tools made by the free/open source Sigrok project to do the actual protocol decoding, the most challenging part of the protocol decoding is already solved. Now the main task is interfacing with Sigrok and verifying that it works as needed, instead of implementing the actual protocol decoding. Sigrok directly supports input formats from many different logic analyzers, making it possible not to force the designer to use a specific logic analyzer for this process to work.

By opening a waveform in Sigroks GUI PulseVeiw, Sigroks capabilities can be tested. Figure 4.2 shows a waveform containing SPI communication opened in PulseView. By adding an SPI protocol decoder and configuring it with the correct metadata like word size, sample edge, and channel mapping, the information transferred by the waveform is also shown. It is evident by the figure that Sigrok can provide all the necessary information, mainly the start time and data for each packet.

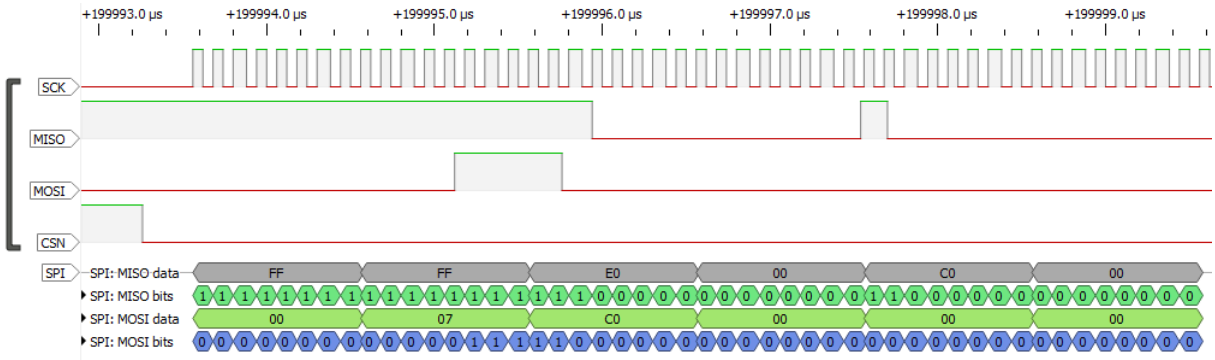


Figure 4.2: Waveform with protocol decoder shown in PulseView

4.2.3 Sigrok-cli

In order to extract this information the sigrok-cli most be used instead of the GUI. It can be started from, and output information to, the command line. A command for decoding the waveform in figure 4.2 will look like: `sigrok-cli -i spi_read.vcd -P spi:miso=CH1:mosi=CH2:clk=CH3:cs=CH4:wordsize=8:bitorder=msb-first:cpol=0`. and gives the output shown in listing 4.1

```
1   spi-1: 1
2   spi-1: 1
3   spi-1: 1
4   spi-1: 1
5   spi-1: 1
6   spi-1: 1
7   spi-1: 1
8   spi-1: 1
9   spi-1: 0
10  spi-1: 0
11  spi-1: 0
12  spi-1: 0
13  spi-1: 0
14  spi-1: 0
15  spi-1: 0
16  spi-1: 0
17  spi-1: FF
18  spi-1: 00
19  spi-1: 1
20  spi-1: 1
21  spi-1: 1
22  spi-1: 1
23  spi-1: 1
24  spi-1: 1
25  spi-1: 1
26  spi-1: 1
27  spi-1: 1
28  spi-1: 1
29  spi-1: 1
30  spi-1: 0
31  spi-1: 0
32  spi-1: 0
33  spi-1: 0
34  spi-1: 0
35  spi-1: FF
36  spi-1: 07
37  spi-1: 0
38  spi-1: 0
39  spi-1: 0
40  spi-1: 0
41  spi-1: 0
...
105 spi-1: 0
106 spi-1: 0
107 spi-1: 00
108 spi-1: 00
```

Listing 4.1: Sigrok-cli output simple command

4.2.3.1 Timing

The first observation is that this data does not contain the start time for each packet. This can be solved by adding `--protocol-decoder-samplenum` to the command. With this command, sigrok will output the start and end sample for each row of information in the output. The testbench will use the start sample for each packet, to start it at the correct time. It must therefore be configured with the sample rate to be able to translate between a sample number and time. Even though the raw data file from the logic analyzer contains what sample interval was used when recording the file, there is no command in the sigrok-cli to get the sample rate.

It is possible to use the time of the start and end sample to calculate the baud rate of a packet. Note that Sigroks intended use for the time tags is to draw the annotation boxes in the GUI. It is therefore not a requirement that the values are absolutely correct. For this SPI decoding, the start time of each packet is set as expected. However, the end time is set by assuming that the final clock cycle has the same length as the second to last clock cycle. The same decoding as in Listing 4.1, with the option to add the sample numbers, is shown in listing 4.2. Here it is obvious that this estimation is slightly incorrect as one byte ends after the next byte has begun. In this case, using the start and end times to calculate the baud rate would result in it being about 2% off. That can work fine in some application, but as a general rule, the baud rate should be specified explicitly, and not calculated, to avoid issues.

4.2.3.2 Data content

While it is possible to parse the output in listing 4.1 directly, this output has not been formatted to be easily machine parsable. For instance, the bit values are intertwined with the byte values, and there is no easy way of telling if a value is for the MISO or MOSI line. After opening the file in PulseView it is possible to deduce that the information is printed with the MISO bits together before the MOSI bits, then the MISO and MOSI byte values. It is clear that it easy to introduce errors when parsing this information. It is possible to reduce the amount of information printed by adding `-A spi=mosi-data`. Now only the MOSI byte values will be printed, avoiding the information confusion.

Adding all the proposes changes gives the following command:
`sigrok-cli -i spi_read.vcd -P spi:miso=CH1:mosi=CH2:clk=CH3:cs=CH4:cpol=0:wordsize=8:bitorder=msb-first --protocol-decoder-samplenum -A spi=mosi-data`
Which gives the output in listing 4.2.

```
1 199993560-199994580 spi-1: 00
2 199994560-199995580 spi-1: 07
3 199995560-199996580 spi-1: C0
4 199996560-199997580 spi-1: 00
5 199997560-199998580 spi-1: 00
6 199998560-199999580 spi-1: 00
```

Listing 4.2: Sigrok-cli output complete command

This output is a lot more workable than the original output, but it is still not ideal. If both the MISO and MOSI data is needed, the sigrok-cli must either be run multiple times or the confusion issue reintroduced. Listing 4.3 shows the output for a single SWD packet like the one in figure 3.3. The output for it has been split into three separate lines, all part of the same packet. This makes it obvious that the parser for the data would have to be modified for different protocols, and making a universal parser is impossible. The Sigrok project evens specifically states on their wiki that parsing this information is not a reliable method, as it can change between versions, and different protocol decoders use different ways of presenting the data [13]. Phrasing this data will, therefore, be tedious and error-prone.

```
1 134733113-134745363 swd-1: W CTRL/STAT
2 134747613-134751113 swd-1: OK
3 134764613-134818863 swd-1: 0x54000000
```

Listing 4.3: Sigrok-cli output for SWD

4.2.4 Stacking decoder

An alternative approach for extracting the data from Sigrok is the use of a stacking decoder. In sigrok, regular protocol decoders have 'logic' as the input format and can decode a waveform in some way. A stacking decoder is special because instead of having 'logic' as input, it can use the output of another regular protocol decoder as input. It is intended for cases like when communication is done with an SPI sensor. The SPI decoder would then decode the raw data, and stacked on top of that decoder would be another decoder specific for that sensor. Decoding the SPI-data to show the values measured by the sensor.

For this project, a simple stacking decoder for the purpose of extracting data was made. It can be put on top of any protocol decoder that supports stacking decoders and will output all data passed to it, with JSON formatting. The data passed to the decoder is always in the format `[<startsample>, <endsample>, <packet>]`. The content of the start and end sample fields are universal for all protocol decoders.

How the `<packet>` field is encoded depends on the protocol decoder. A general guideline is that it should have the format `[<packettype>, <packetdata>]`. However, this is not strictly enforced, and some decoders will deviate from it. For SPI a typical output will be `[199993560, 199994580, ["DATA", 0, 255]]`. Here the packet data is split in two, with the first element for MOSI data, and the second for MISO data. The only way to find information about how the packet field is formatted is to look in the header of the source code for that protocol decoder.

A command for running sigrok with the stacking decoder called `JSON_print` added will look like `sigrok-cli -i spi_read.vcd -P spi:miso=MISO:mosi=MOSI:clk=SCK:cs=CSN:wordsize=8:bitorder=msb-first:cpol=0,JSON_print -B JSON_print=json`. The output from the command is shown in listing 4.4. Note that by using the stacking

decoder, some information that was unavailable in the output from the SPI decoder is now available. The value for the chip-select signal and metadata is now output together with the decoded data. Metadata is output in the format `["METADATA", <datapoint>, <value>]`. Data point 1000 is the sampling rate, making it possible to extract the sampling rate from the raw data file,

The outputs made directly by the SPI and SWD decoder were of annotation-type and primarily made to be human readable. To avoid sigrok adding the header to each line showing which protocol has printed the data, and since showing the JSON encoded data in the GUI would just look ugly, the output from JSON_print is of binary-type. Unlike annotation options, where all outputs are enabled by default, all binary outputs are disabled by default. Hence the need for adding -B JSON_print=json to the command.

```
1  [0, 0, ["METADATA", 1000, 10000000]]
2  [0, 0, ["CS-CHANGE", NULL, 1]]
3  [199993260, 199993260, ["CS-CHANGE", 1, 0]]
4  [199993560, 199994580, ["BITS", [[0, 199994440, 199994580],
   [0, 199994300, 199994440], [0, 199994180, 199994300], [0, 199994060, 199994180],
   [0, 199993940, 199994060], [0, 199993800, 199993940], [0, 199993680, 199993800],
   [0, 199993560, 199993680]], [[1, 199994440, 199994580], [1, 199994300, 199994440],
   [1, 199994180, 199994300], [1, 199994060, 199994180], [1, 199993940, 199994060],
   [1, 199993800, 199993940], [1, 199993680, 199993800], [1, 199993560, 199993680]]]
5  [199993560, 199994580, ["DATA", 0, 255]]
6  [199994560, 199995580, ["BITS", [[1, 199995440, 199995580],
   [1, 199995300, 199995440], [1, 199995180, 199995300], [0, 199995060, 199995180],
   [0, 199994940, 199995060], [0, 199994800, 199994940], [0, 199994680, 199994800],
   [0, 199994560, 199994680]], [[1, 199995440, 199995580], [1, 199995300, 199995440],
   [1, 199995180, 199995300], [1, 199995060, 199995180], [1, 199994940, 199995060],
   [1, 199994800, 199994940], [1, 199994680, 199994800], [1, 199994560, 199994680]]]
7  [199994560, 199995580, ["DATA", 7, 255]]
...                                    ⋮
15 [199998560, 199999580, ["DATA", 0, 0]]
```

Listing 4.4: Sigrok-cli output with JSON encoder

Compared to the outputs in listings 4.1 and 4.2, parsing this output should be trivial, and no effort is needed to change the parser based on the protocol. For a SWD transmission of the same packet as in listing 4.3 the output with the stacking decoder is shown in listing 4.5. The first element of the data field is the address, the second is the data, and the third is the acknowledge status. The data value is different due to it being decimal encoded when output by the stacking decoder, while it was hexadecimal encoded by the SWD decoder.

```
1  [134733113, 134820613, ["DP_WRITE", [4, 1409286144, "OK"]]]
```

Listing 4.5: Sigrok-cli output with JSON encoder for SWD

4.2.4.1 making the stacking decoder

The JSON_print stacking decoder was implemented by adapting Sigroks template decoder. Most of Sigroks code is written C. However, the protocol decoders are written in Python. Since this decoder does not do any actual decoding, only a few changes were needed to the template decoder. First, the binary output had to be added as an option to the decoder, and registered to the sigrok framework in the initialization function of the decoder. Finally, the decode function itself needs to be updated to output all data passed to it into the binary data stream. The decode function is shown in listing 4.6. The formatting for the put function is (<startsample>, <endsample>, <stream type>, [<stream number>, <data>]).

```
1 def decode(self, startsample, endsample, data):
2   outputdata = json.dumps((startsample,endsample,list(data))) + '\n'
3   self.put(startsample, endsample, self.out_binary, [0, outputdata.encode()])
```

Listing 4.6: JSON_print decode function

The JSON_print decoder works fine with the limited functionality it has so far. It will be easy to add additional functionality if needed. One tempting possibility is to have the decoder directly output the data file that the testbench will read. If this was attempted, configurability of the stacking decoder would become a challenge. There are a plethora of different ways decoders can present the decoded data and multiple different data fields from one decoder that can be useful in the simulation. Therefore making the JSON_print decoder have enough options to be configurable for all use cases is not feasible, and changes to decoder itself will have to be made every time a new protocol is used.

All protocol decoders must be located inside a specific directory inside the install directory of the sigrok-cli. This further complicates this idea as you would quickly end up with different versions of the decoder for different projects, and get a manual step of replacing the decoder every time a new project is worked on.

4.2.5 Automating sigrok

The final implementation of the protocol decoding uses the stacking decoder discussed in section 4.2.4 and a python class, TorgeSigrok, for interfacing with the sigrok-cli. The TorgeSigrok class handles all direct interaction with the sigrok-cli; starting the sigrok-cli with the correct command with all options, and parsing the output data. The decoded data is returned from TorgeSigrok as a list of dicts. Each dict in the list have the fields {<startsample>, <endsample>, <packet type>, <data>}. An overview of the intended dataflow when using TorgeSigrok is shown in figure 4.3.

In addition to handling the waveforms and decoded data, TorgeSigrok has functionality to help ensure that metadata is consistent between Sigrok and the testbench. TorgeSigrok directly imports a copy of the protocol decoder for the specified protocol. This allows TorgeSigrok to be aware of all options that are available in the protocol decoder, and their

default values. If a setting is not specified for an option, TorgeSigrok will use the default value in the protocol decoder. The same default value is then used when generating the configfile with all the metadata. This method ensures that the configfile always contains all the information used to decode the waveform. It is also possible to add settings to the configfile that are not used for decoding. One common such setting will be baud rate for synchronous protocols, since it is not needed when decoding the waveform, but is required when recreating it.

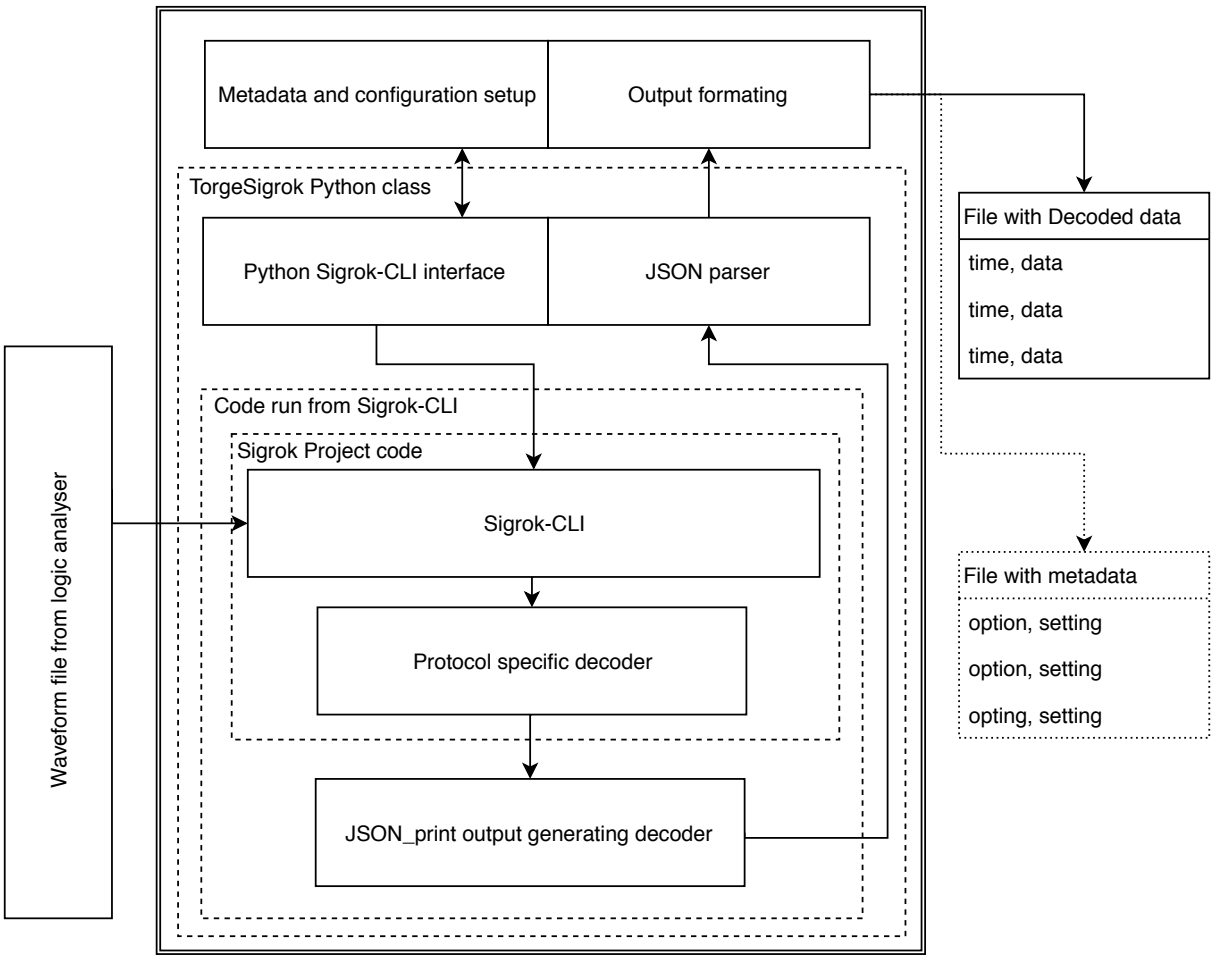


Figure 4.3: Automated waveform decoding with sigrok

Since all of the needed functionality is implemented in the TorgeSigrok class, very little additional code is needed to decode a waveform. A example use of TorgeSigrok, for decoding the SPI transaction from figure 4.2 is shown in listing 4.7. The two generated output files are shown in listing 4.8 and 4.9.

The code in listing 4.7 will make TorgiSigrok start the sigrok-cli with the command `sigrok -cli -i spi_read.vcd -P spi:miso=MISO:mosi=MOSI:clk=SCK:cs=CSN:wordsize=8: bitorder=msb-first:cpol=0,JSON_print -B JSON_print=json`, makeing the sigrok-cli produce the same output as in listing 4.4. Since channels and options are added in the same way when using the sigrok-cli, the only difference between a channel and a option in TorgeSigrok is that channels are only used with the sigrok-cli, while options are used with the sigrok-cli and saved to the configfile.

```
#!/usr/bin/python3
import TorgeSigrok

sr = TorgeSigrok.TorgeSigrok(protocol="spi")
sr.inputfile("spi.vcd")
sr.channels([['miso', 'MISO'], ['mosi', 'MOSI'], ['clk', 'SCK'], ['cs', 'CSN']])
sr.options ([['wordsize', 8], ['bitorder', 'msb-first'], ['cpol', '0']])

data_file_data = []
for elem in sr.run_sigrok():
  if elem['type'] == 'DATA':
    data_file_data.append("{} {:02x} {:02x}".format
    (elem['starttime']/sr.samplerate, elem['data'][0],elem['data'][1]))

sr.save_data(data_file_data,filename='spidata.txt')
sr.save_config(options=[['master_slave', 'master'], ['BAUDRATE', 8000000]])
```

Listing 4.7: TorgeSigrok usage spi

After starting the sigrok-cli, TorgeSigrok parses the JSON data output and puts it in the list of dicts returned by the run_sigrok function. The top function then loops thru all the dicts and formats how the data should be saved. Since the absolute time for the start sample depends on the sample rate, the start time is divided on the sample rate to get the time in seconds. Finally, the datafile and configfile are saved, when saving the configfile additional options that are only for the testbench can be added.

```
0.199993560 00 ff
0.199994560 07 ff
0.199995560 c0 e0
0.199996560 00 00
0.199997560 00 c0
0.199998560 00 00
```

Listing 4.8: spidata.txt

```
cpol 0
cpha 0
bitorder msb-first
wordsize 8
BAUDRATE 8000000
master_slave master
```

Listing 4.9: configfile.txt

An equivalent approach is possible for signals that are not to be decoded. A non-decoding decoder has been made that can be used instead of a protocol decoder. The non-decoding passes all the individual signal changes on to the JSON_print stacking decoder. TorgeSigrok can then be used in the same way to generate the datafile. Therefore the same method can be used both with and without protocol decoding.

4.2.5.1 Transforming metadata

While these output files will work with a compatible testbench, some additional formatting of the configfile might be needed. The variable names for the options are often different between sigrok and the testbench. Therefore a translation between the different option naming schemes is needed. This translation can either be done as part of the python code or as part of the testbench.

If the translation is done as part of the python code, it can be implemented as a new subclass of TorgeSigrok. This subclass will work the same way as the original TorgeSigrok but will implement the option translation as part of the new save_config function. This subclass can also implement an improved method of handling options that are only used in the simulations. The simulation only options are added with the same format as options in Sigroks protocol decoders. With this change, the simulation options can be used in the top python script in the same way as the decode options, and default values are used if no setting is selected for an option. Listing 4.10 show a subclass of TorgeSigrok made for SPI decoding. The configfile generated by the code in listing 4.7 changed to use SpiSigrok is shown in listing 4.11, the datafile still looks lie listing 4.8.

```
1  import TorgeSigrok
2
3  class SpiSigrok(TorgeSigrok.TorgeSigrok):
4
5   def __init__(self):
6    super().__init__("spi")
7
8    self.sim_options += ({'id': 'BAUDRATE', 'desc': 'SPI baudrate',
      'default' : 115000},)
10   self.sim_options += ({'id': 'master_slave', 'desc': 'Simulator option',
      'default' : '1', 'values' : ('1','0')},)
11   self.sim_options += ({'id': 'initiator_responder', 'desc': 'Simulator option',
      'default' : '1', 'values' : ('1','0')},)
12
13  def save_config(self,configfile="configfile.txt"):
14   self.config_file_data = []
15
16   for elem in options+self.use_decode_options+self.use_sim_options:
17    if elem[0] ==   "cpol":      elem[0] = "SPCR_CPOL"
18    elif elem[0] == "cpha":      elem[0] = "SPCR_CPHA"
19    elif elem[0] == "bitorder": elem[0] = "MSB_FIRST"
20      elem[1] = '1' if elem[1] == "msb-first" else '0'
21    self.config_file_data.append("{} {}".format(elem[0],elem[1]))
22
23   self._writefile(self.config_file_data,configfile)
```

Listing 4.10: SpiSigrok

The values for “bitorder” is converted from a string to an integer to make the parsing of the file simpler in the testbench. The parser in the testbench can now assume that all settings are numbers. The alternative is first to treat all settings strings and afterward convert the settings into integers for only some of the options.

```
1 SPCR_CPOL 0
2 SPCR_CPHA 0
3 MSB_FIRST 1
4 wordsize 8
5 BAUDRATE 8000000
6 master_slave 1
7 initiator_responder 1
```

Listing 4.11: Translated configfile.txt

4.3 The testbench

The goal of the testbench is to use input files generated by the method described in section 4.2, to generate stimulus that is equivalent to the stimulus that was applied to the physical DUT. This section will not focus on implementing a general UVM testbench. Instead, it will focus on special consideration needed for adapting a testbench to generate stimulus based on prerecorded data.

4.3.1 UVM framework testbench

The UVM Framework (UVMF) made by Mentor Graphics provides a jump-start for learning UVM and building UVM verification environments. It expands upon UVMs functionality by providing new base classes that extends UVMs existing base classes. By using these base classes instead of the UVM base classes, a minimal feature-complete testbench can be made with very little additional code. [14]

To get a feature-rich testbench working based on pure UVM, requires a lot of additional code different parts of the testbench. In the UVMF most of this code has been added in the extended base classes. The result of this is that a minimal setup with UVMF will have more features, but less flexibility and a more strict usage method.

Together with the UVMF, Mentor has made several example testbenches to show the UVMF works. One of the example testbenches is for a wishbone to SPI converter. It has one UVC for wishbone acting as a master, and one for SPI acting as a slave. A new testbench was made based on this testbench to test methods for recreating the decoded SPI waveform. By basing this testbench on an existing testbench instead of creating a new one, it also allowed insight into how to adapt an existing testbench into reading stimulus data from a file.

To make the scenario of adapting an existing testbench more realistic, the new testbench was made as similar as possible to the wishbone to SPI testbench. It used the original testbench as a starting point, removing the wishbone UVC and reconfiguring the SPI UVC to act as the master. More functionality for different configuration options was added to the driver bus functional model in the SPI UVC. The DUT was replaced with a minimal SPI DUT. It simply connects the MOSI and MISO line together. This means that it always responds with precisely the data that is sent to it. The rest of section 4.3 details how this testbench was adapted to read stimulus from the files generated in section 4.2.

4.3.2 Adapting a UVM sequence

The main task when adapting the testbench is adding a new UVM_sequence that reads stimulus data from a file. This sequence can either replace the original sequence in the testbench and be the only sequence, or it can be one of multiple sequences started by a master sequence. Which functions are used by the sequence to interact with the driver and sequencer is shown in figure 2.5.

The sequence is designed as a while loop that parses one row in the datafile for each pass and runs until it has reached the end of the file. Inside the loop it sets the spi_data and start_time variables in the sequence_item and passes the sequenze_item on to the sequencer. When the end of the datafile is reached, the sequence will terminate. The sequence is shown in listing 4.12.

```
1  class read_file_sequence extends spi_sequence_base #(.REQ(delay_spi_transaction),
       .RSP(spi_transaction));
2   `uvm_object_utils(read_file_sequence)
3
4   function new();
5    super.new();
6    $timeformat(1, , , )
7   endfunction : new
8
9   string datafile = "../spidata.txt";
10  REQ req_transaction; RSP rsp_transaction;
11
12  task body;
13   int status; int MISOdata; int MOSIdata; longint start_time; int fd;
14   req_transaction = delay_spi_transaction::type_id::create("req_transaction");
15
16   fd = $fopen(datafile,"r");
17   if (!fd) `uvm_error("SEQ","Could not open datafile" )
18
19   while (!$feof(fd)) begin
20     status = $fscanf(fd,"%t %h %h",start_time, MISOdata, MOSIdata);
21     if(status != 3) begin
22      `uvm_error("SEQ", $sformatf("Issue with datafile %d", status))
23       breake; //Abort if file is not formatted correctly
24     end
25
26     start_item(req_transaction);
27     req_transaction.spi_data = MOSIdata;
28     req_transaction.start_time = start_time;
29     `uvm_info("SEQ", req_transaction.convert2string, UVM_MEDIUM)
30     finish_item(req_transaction);
31    end
32
33    $fclose(fd);
34  endtask: body
35 endclass: read_file_sequence
```

Listing 4.12: UVM sequence reading data from file

The start time of the packet is parsed with the `"%t"` format specifier. By using this format specifier the time values are scaled to the timescale of the testbench. The value stored in start_time will, therefore, depend on the timescale set in the testbench. Having the value scaled to the testbench makes the functionality in the driver simpler, the value returned by the `$time` function can now be directly compared with the start time. The

`$timeformat` function is used to specify the timescale in the data file. Since this function also configures the formatting for when time is printed to the terminal, the drawback of this solution is that all the printouts in the testbench must use the same time format as the one that was used in the datafile.

For SPI, either the MISO or the MOSI data must be used as stimulus, MOSI if the testbench is the master, and MISO if the testbench is the slave. In this case, the testbench is the master, and therefor spi_data is set to be the MOSI data. It is possible to automate this selection based on whether the testbench is configured to be the master or the slave. For any new test case some reconfiguring of the sequence must be expected, and how to access the configuration data will vary for different testbenches. Automation this will therefore add more unneeded complexity for a minimal gain.

4.3.2.1 Verifying response

The data file contains both the stimulus data and the original DUTs response to that stimulus. It is possible to use this to check if the response from the simulated DUT is the same as the original response. This will be useful if it is expected or relevant that the response is the same. In a case where the DUT is a sensor, it is expected that the response is the same if it is responding with its configuration. However, if it is responding with its measured value, it is not expected. It is therefore ideal if the response is always compared to the original response, and the result of the comparison does not influence the rest of the testbench behavior.

A simple way to add this functionality that can be implemented without the need for altering the UVCs verification functionality is to add it to the sequence. This is possible because the same sequence_item that carried the message from the sequence to the driver, can also carry the response back from the driver to the sequence. Adding the checking functionality to the sequence in listing 4.12, is done by adding the content in listing 4.13 between `finish_item(req_transaction);` and the end of the while loop.

```
1 get_response(rsp_transaction);
2
3 if (rsp_transaction.spi_data == MISOdata) `uvm_info("SEQ-RSP", $sformatf("Response
      was: %h as expected", rsp_transaction.spi_data), UVM_MEDIUM)
4 else `uvm_warning("SEQ-RSP", $sformatf("Response was: %h, expected: %h",
     rsp_transaction.spi_data,r_data))
```

Listing 4.13: Verifying the response

The UVM logs for the testbench with the sequence in listing 4.12 with the addition in listing 4.13 for "SEQ" and "SEQ-RSP" when using the datafile in listing 4.8 is shown in listing 4.14. Most of the responses are correctly identified as not as expected due to the simulated DUT always responding with the data that was sent to it.

```
1  UVM_INFO sim/tb/src/tb_seq.svh(24) @ 0: uvm_test_top.environment.spi.
       sequencer@@read_file_sequence_s [SEQ] dir:TO_SPI spi_data:0xff start_time:
       0.199993560
2  UVM_WARNING sim/tb/src/tb_seq.svh(28) @ 0.199994490: uvm_test_top.environment.spi.
       sequencer@@read_file_sequence_s [SEQ-RSP] Response was: ff, expected: 00
3  UVM_INFO sim/tb/src/tb_seq.svh(24) @ 0.199994490: uvm_test_top.environment.spi.
       sequencer@@read_file_sequence_s [SEQ] dir:TO_SPI spi_data:0xff start_time:
       0.199994560
4  UVM_WARNING sim/tb/src/tb_seq.svh(28) @ 0.199995490: uvm_test_top.environment.spi.
       sequencer@@read_file_sequence_s [SEQ-RSP] Response was: ff, expected: 07

...                                        ⋮

11 UVM_INFO sim/tb/src/tb_seq.svh(24) @ 0.199998490: uvm_test_top.environment.spi.
       sequencer@@read_file_sequence_s [SEQ] dir:TO_SPI spi_data:0x00 start_time:
       0.199998560
12 UVM_INFO sim/tb/src/tb_seq.svh(27) @ 0.199999490: uvm_test_top.environment.spi.
       sequencer@@read_file_sequence_s [SEQ-RSP] Response was: 00 as expected
```

Listing 4.14: UVM log for SEQ and SEQ-RSP

4.3.3 Setting the starttime

The original UVC did not have functionality for setting the start time of a packet. It would always transmit one packet immediately after the previous. Two steps were needed to add this functionality. A field for the start time had to be added to the sequence_item, and the delay functionality added to the driver. To ensure backward compatibility with the other sequences that do not set the start time, the start time in the sequence item will default to 0, and the driver will transmit a packet immediately if the set start time has passed. Listing 4.15 shows the additional code added to the driver.

```
1  if(start_time>=$time) #(start_time - $time);
2  else `uvm_warning("Driver-bfm", $sformatf("Packet: %h start_time: %d started at
       time %d",spi_driver_dout,start_time,$time()))
```

Listing 4.15: Driver waiting until starttime

By default, the testbench will end the simulation immediately after the last sequence have been completed. This is undesirable because it is not possible to investigate the DUTs reaction to the final message. To fix this `run_phase.phase_done.set_drain_time(this, 5000ns);` is added to the top module of the UVC.

4.3.4 Configuration

In the same way that the stimulus data is generated based on a file, the configuration of the testbench can be set based on a file. This is optional, and for one-time use, it will be simpler to configure the values manually. If automating the configuration data entry is worth the additional complexity depends on if repeatedly changing the configuration is likely. If setup correctly, automating the configuration removes the danger of generating an incorrect waveform due to incorrect configuration of the metadata.

The code for reading the metadata is similar to the code used to read the stimulus data. A while loop parses the configfile one line at a time and sets the corresponding option. For the options where the setting is not a number, the setting must be translated into the format used in the testbench. Code implementing this is shown in listing 4.16. It can be added to the start of the sequence, keeping all of the additional code added to the testbench in one place.

```
string varible; string configfile = "../configfile.txt";
int status; int fd; int setting;

fd = $fopen(configfile,"r");
if (!fd) `uvm_error("tb-cfg", "Could not open configfile")

while (!$feof(fd)) begin
 status = $fscanf(fd,"%s %d",varible, setting);
 if(status != 2) begin
  `uvm_error("tb-cfg",  $sformatf("Issue with configfile %d", status))
  breake;
 end

 `uvm_info("tb-cfg", $sformatf("Setting %s -> \t %d",varible,setting), UVM_MEDIUM)
 case(varible)
  "SPCR_CPOL"     : spi_config.SPCR_CPOL     = setting;
  "SPCR_CPHA"     : spi_config.SPCR_CPHA     = setting;
  "BAUDRATE"      : spi_config.BAUDRATE      = setting;
  "MSB_FIRST"     : spi_config.MSB_FIRST     = setting;
  "master_slave"  : spi_config.master_slave = setting ? MASTER : SLAVE;
  "initiator_responder" : spi_config.initiator_responder = setting ? INITIATOR:
    RESPONDER;
  default     : `uvm_warning("tb-cfg", $sformatf("Unsupported configuration %s",
    varible))
 endcase // varible
end

spi_config.driver_bfm.configure(spi_config.to_struct()); //Update configuration
$fclose(fd); // Close file before finish
```

Listing 4.16: Configuring the testbench

4.3.4.1 Non fixed configuration data

The strategy of setting all the options at the start of the simulation works as long as the metadata does not need to be changed during a transmission. If the metadata needs to change, for example if different packets in a transmission have different baud rate, a more complex solution is required. In the testbench, the configuration is set by calling the configure function in the driver with a struct containing all the configuration option. The driver adapted from the UVMF changes an option immediately after configure has been called. It is therefore important to have an external synchronization mechanism to ensure that the configuration is only changed between packages.

There is no way to configure sigrok with changing metadata. If the changing metadata is something like baud rate that does not affect the output of the decoding, this is not an issue. In this case, the method for adding the changing metadata that requires the least change to the code would be to manually add lines in the datafile where the metadata changes. A simple way to do this is to set the start time to -1 and the data field to the

baud rate. In the sequence then add a check for if the start time is -1, then configure the baud rate to the data value.

If the changing metadata does affect the output of the decoder the only solution is to split up the decoding in sigrok into different parts, each with consistent metadata. This will produce multiple data and metadata file pairs. One way to recreate the waveform from this is to make the inputfiles to the sequence configurable, and then start the sequence on time for each file pair.

4.4 Comparison of recorded and generated waveform

The goal for this process is that the waveforms generated in the simulation are as close as possible to those captured by the logic analyzer. This was tested with the short 6 byte transmission from Figure 4.2, using the method described in section 4.2 and 4.3. The waveform captured by the logic-analyzer is shown in figure 4.4, and the waveform generated in the simulation is shown in figure 4.5. At a quick glance, they appear very similar. However, due to the way both of the waveforms are created, there are subtle differences between them. By focusing on the differences, the limitation of the protocol decoding technique is highlighted.

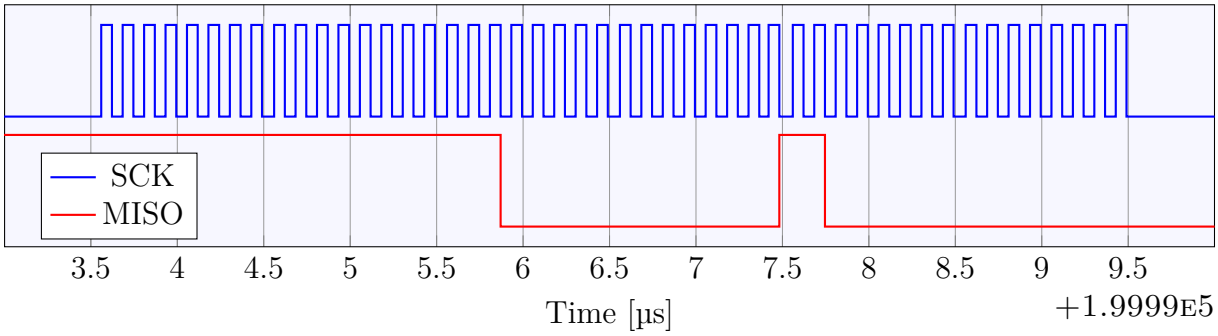


Figure 4.4: Waveform as generated in simulation

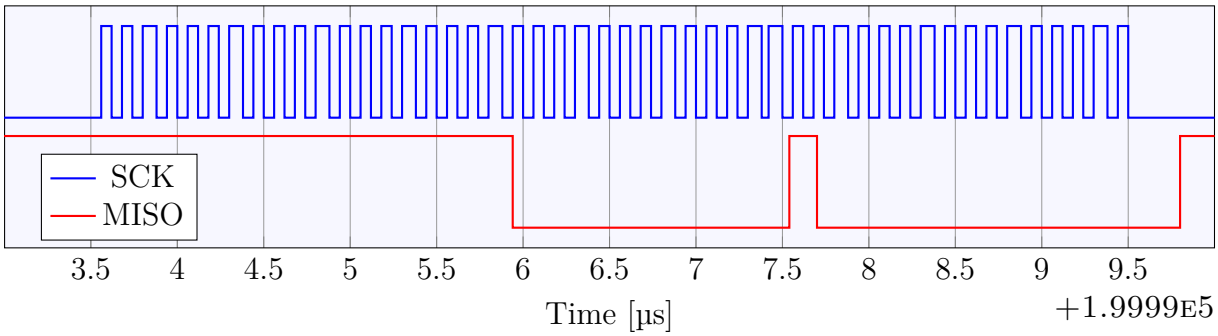


Figure 4.5: Waveform as recorded by logic analyzer

The most obvious difference between the recorded and generated waveforms, is that MISO goes high after the transmission is finished in the recorded waveform. There are simple reasons why this is not recreated in the generated waveform. The datafile made with TorgeSigrok does not contain any information on what MISO should be set to between the packages. Since sampling is done at the leading clock edge, the testbench will set a bit half a clock cycle after the previous bit has been sampled. The change on MISO happens after the last packet in the recording has been transferred. Therefore the testbench has no information on how to set the next bit. The change occurs after the last piece of information has been transferred. Consequently, this difference should not affect the simulation result.

4.4.1 Magnified waveform

Figure 4.6 and 4.7 show the same waveforms zoomed in at a few clock cycles. This shows the more subtle difference between the waveforms. In Figure 4.7 the gray tick lines indicates the sampling times in the logic analyzer used to record the waveform. The waveform can only be updated at the lines, and any change in value that actually occurred between two lines does not get updated in the waveform until the next line.

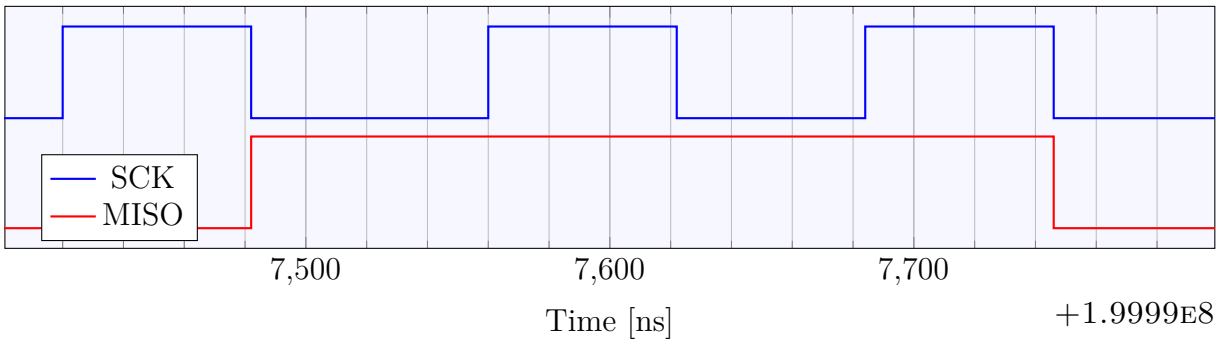


Figure 4.6: Zoomed in waveform as generated in simulation

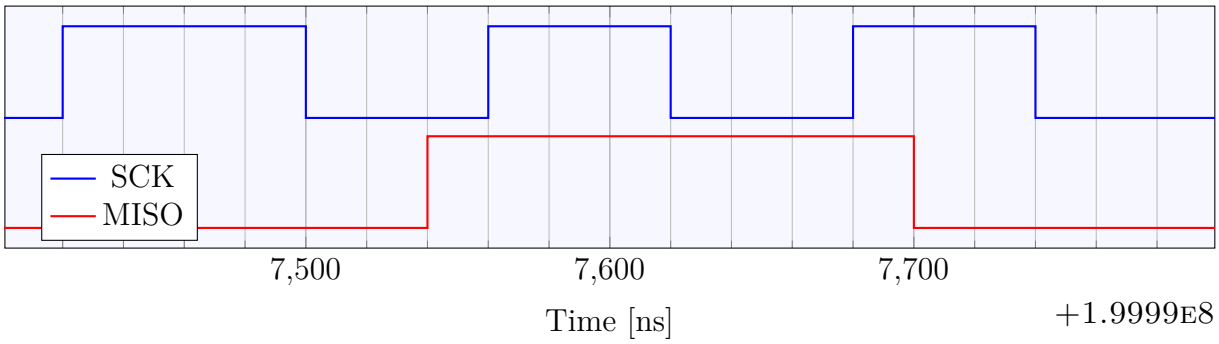


Figure 4.7: Zoomed in waveform as recorded by logic analyzer

A new byte starts at the time 7560 (+1.9999E8) ns. Therefore this clock edge is perfectly aligned between the two figures. The other clock edges are not aligned, and there is up to a 20 ns difference in when they occur. In the recorded waveform the clock edges are not uniformly spaced, this is due to the 50 MHz sampling clock in the logic analyzer not

perfectly synchronizing with the 8 MHz clock used to transmit the data. In the recording there are either 60 or 80 ns between the clock edges, averaging 62.5 ns.

In the generated waveform the clock edges are evenly spaced 61.5 ns apart, this is calculated based on the configured baud rate. The only exception to this is the half clock cycle between packets, due to the first clock edge in a packet not being calculated based on the previous clock edge but set based on the packet start time.

Since there is no guarantee that the start times of the packets are uniformly spaced even when transmitting continuously, trying to both have each packet take the correct length of time and start them at the correct time would lead to many warnings of packets not started at the correct time. Therefore the time used to transmit a package is set 1.5% lower than what is expected based on the baud rate, this allows every packet to be started at the time in the data file. The uneven timing caused by this is still lower than the timing unevenness caused by the sampling rate in the original recording.

There is also a difference in the MISO line. Data sampling is done on the rising clock edge. Therefore the generated waveform sets the MISO line on the falling edge. This also works for the first bit in a packet, where the bit is set at the final clock edge of the previous packet, if there is a delay between the packets, the bit if therefor set earlier than needed. In the recorded waveform the MISO line does not change precisely at a clock edge, this information is lost when utilizing protocol decoding.

5 Discussion and Conclusion

5.1 Rawdata or decode

The solution described in chapter 4, implementing the protocol decoding ideas, recreates the original waveform with only minor distortions. If the distortion is not acceptable, it is possible to eliminate it by doing raw data playback instead of protocol decoding. However, the distortionless playback is the only advantage, and unless it is critical for the current test scenario protocol decoding is a better choice.

With protocol decoding, the signals are handled at the information level. This allows for more testbench reuse and makes the job simpler for the designer. Protocol decoding solves the synchronization issues that can occur with raw data playback at the protocol level. The drawback of protocol decoding is the additional complexity added to the workflow.

5.2 System implementation

The baseline solution using sigrok for protocol decoding and a UVM_sequence for reading the datafile works well. When needed, it is possible to add more functionality and complexity to it, such as verifying the response and automating the metadata setup. By keeping the baseline functionality as simple as possible, the undesired complexity from the optional functionality can be avoided when it is not needed.

The implementation described in chapter 4 is for an SPI protocol in a UVMF testbench. No UVMF specific functionality has been used and no simplification made due to SPI being a simple protocol. All choices are made with the goal that the same methodology can be used for other protocols and in other UVM testbenches. Some cases are more suited to this approach than others. Due to the very different nature of different signals, it is not possible to create one solution that is ideal for every scenario. The workflow implemented produced results very similar to the original waveform. Most signal changes are within one sampling interval of the logic analyzer away from the original recording.

5.3 Achieved goals

The goal set at the start of this project was to automate this process. Complete automation has not been achieved, but a general workflow has been established. It allows for testbench and verification reuse for all signals that are part of a protocol. By adding a new UVC signals that are not part of a protocol can be handled with the same workflow. Since all the timing uses the absolute simulation time, multiple protocols and UVC can be tested at the same time.

Focus has been put on minimizing the effort needed from a designer implementing this workflow into an existing testbench. The usage of sigrok is simplified with TorgeSigrok, requiring less than 15 lines of Python for a basic use-case. Apart from creating a new sequence, very few changes are needed to the testbench. By having the required workflow in place, the designers' focus can immediately be moved from recreating the issue, to solving it.

5.4 Further work

The next step logical step for the work in this project is to use the techniques shown, in the implementation of real-life scenario. By implementing the process described the real usefulness of the workflow can be tested. Testing with protocols other than SPI or SWD may uncover additional challenges that were not encountered during the work on this project.

Bibliography

[1] Ram Narayan and Tom Symons. *I created the Verification Gap*. URL: http://events.dvcon.org/2015/proceedings/papers/10_1.pdf (visited on 05/20/2019).

[2] Harry Foster. *The 2018 Wilson Research Group Functional Verification Study*. URL: https://blogs.mentor.com/verificationhorizons/blog/2018/11/14/prologue-the-2018-wilson-research-group-functional-verification-study/ (visited on 03/02/2019).

[3] H. Nyquist. "Certain Topics in Telegraph Transmission Theory". In: *Transactions of the American Institute of Electrical Engineers, Volume 47, Issue 2, pp. 617-624* 47 (Apr. 1928), pp. 617–624. DOI: 10.1109/T-AIEE.1928.5055024.

[4] Tim Reyes. *What Sample Rate Is Required for a Given Signal?* 2018. URL: https://support.saleae.com/faq/technical-faq/what-sample-rate-is-required (visited on 12/08/2018).

[5] Joel Avrunin. *Putting the Logic in Logic Analyzers*. 2013. URL: https://www.tek.com/blog/putting-logic-logic-analyzers (visited on 12/08/2018).

[6] Tektronix. *Debugging Timing Problems with a Logic Analyzer*. 2009. URL: http://download.tek.com/document/52W_23621_0_Letter_LR.pdf (visited on 11/29/2018).

[7] Accellera. *Universal Verification Methodology (UVM) 1.2 Users Guide*. 2015. URL: http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf (visited on 09/29/2018).

[8] *Sigrok*. URL: https://sigrok.org/wiki/Main_Page (visited on 12/03/2018).

[9] Verification Academy. *Universal Verification Methodology - uvm_tlm_time*. URL: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/index.html (visited on 04/24/2019).

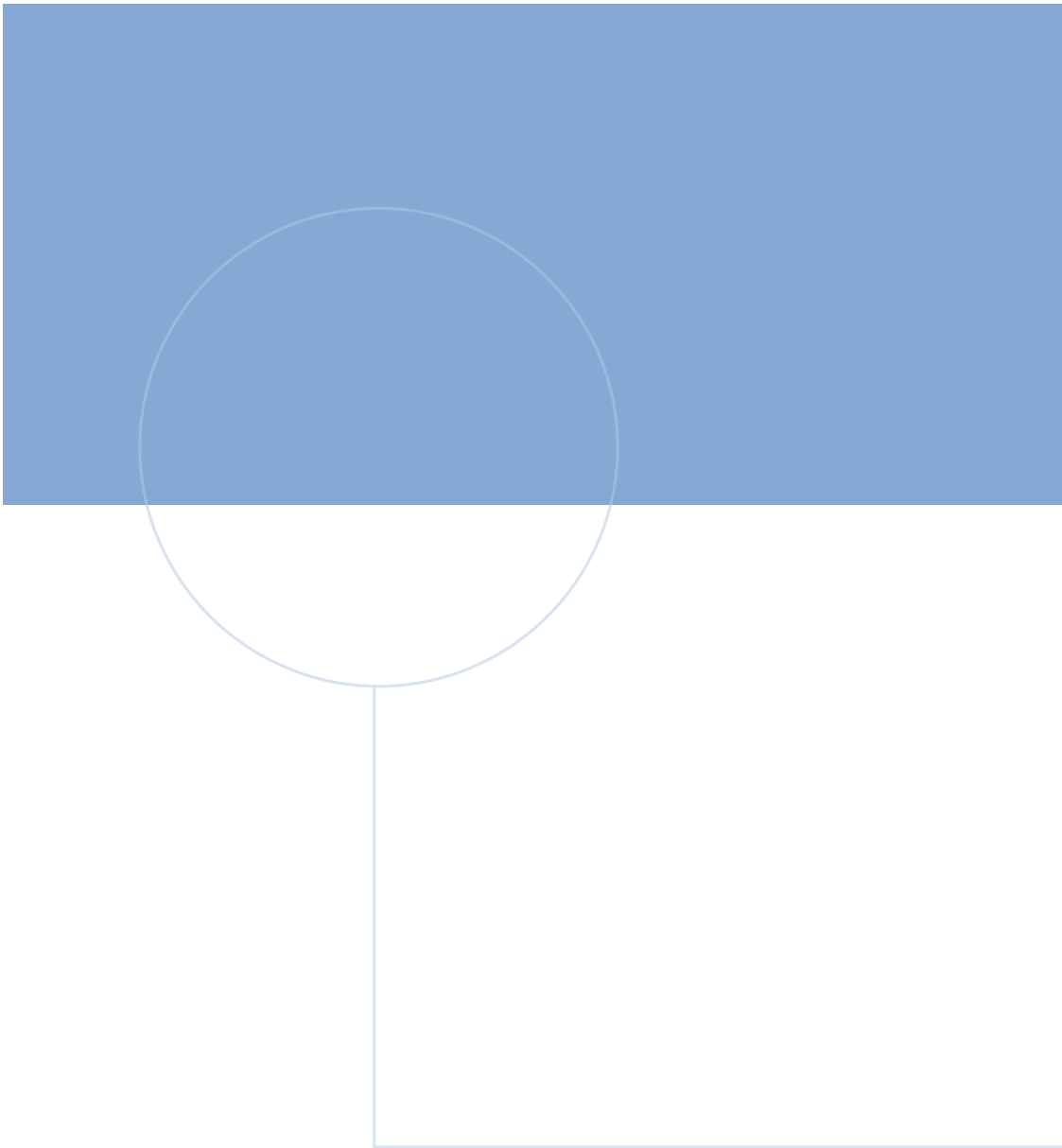
[10] "IEC/IEEE Behavioural Languages - Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001)". In: *IEEE 1364 IEC 61691-4 First edition 2004-10* (2004). DOI: 10.1109/IEEESTD.2004.95753.

[11] Peter Harrod et al. *Serial Wire Debug and the CoreSight™ Debug and Trace Architecture*. 2006. URL: https://www.arm.com/files/pdf/Serial_Wire_Debug.pdf (visited on 12/04/2018).

[12] Verification Academy. *UVC / UVMVerificationComponent*. URL: https://verificationacademy.com/cookbook/uvc/uvmverificationcomponent (visited on 04/11/2019).

[13] Sigrok project. *Protocol decoder output - Annotations*. URL: `https://sigrok.org/wiki/Protocol_decoder_output#Annotations` (visited on 05/07/2019).

[14] Mentor Graphics. *UVM Framework Users Guide Revision 3.6g_0*.



NTNU
Norwegian University of
Science and Technology