NTNU
Norwegian University of
Science and Technology

Master's thesis

2022

Kåre-Benjamin Hammervold Rørvik

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

Master's thesis

Kåre-Benjamin Hammervold Rørvik

Next Generation Compression Algorithm

July 2022

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Next Generation Compression Algorithm

Kåre-Benjamin Hammervold Rørvik

Electronic Systems Design
Submission date: July 2022
Supervisor: Milica Orlandic
Co-supervisor: Lars Erik Songe Paulsen

Norwegian University of Science and Technology
Department of Electronic Systems

Preface

This master thesis is my work for the spring of 2022. It was written at the Department of Electrical Engineering at the Norwegian University of Science and Technology. The thesis is credited 30 points and is part of course TFE4930. Councillors for this master thesis were Milicia Orlandic and Lars Erik Songe Paulsen.

This thesis's objective has been to create a synthesizable hardware implementation of the compression algorithm Asymmetric Numeral Systems (ANS), specifically the uniform binary variant (uABS) of the algorithm, and verify the design. Therefore a software model should also be made and used for prototyping purposes, and as a reference. My preparational project thesis introduced the concept of compression, which will be briefly summarized in this master thesis. Furthermore, I have created two new and improved software models based on the experiences gained from the project thesis. While the software models are new and improved, they are based on the software model of the project thesis. The UVM testbench was based on strategies and methodology proposed by Cadence in their course "SystemVerilog Accelerated Verification with UVM" [1] (completed while working with ARM).

My motivation for making a hardware implementation of the uniform binary variant is twofold:

- I want to learn more about hardware implementation, design, and verification processes. My ambition is to take an arbitrary algorithm and be able to work out a fully synthesizable design, thoroughly verify it, and implement it on an FPGA.
- This uniform binary variant (uABS) of ANS does not have any online open-source hardware implementation. This lack of available open-source implementations makes the work even more exciting. My work could be a meaningful contribution to the world of hardware compressors (e.g., an accelerator in an embedded system).

Acknowledgement

I would like to extend my deepest gratitude to my councillors Lars-Erik and Milica for being very helpful and accessible during the work of this master thesis. I am also grateful for the support and experience I received at ARM. Furthermore, I would like to thank my loving family, friends and colleagues for helping with proofreading and support.

Trondheim, 14.7.2022

Kåre-Benjamin Hammervold Rørvik

Abstract

Compression serves an instrumental role in today's computer systems. Next generation compressors seek to bring modern innovation to the field of compression and answer today's demand for transmitting more data between devices. Although rapid development is happening with software-based compression algorithms, hardware implementations may also bear significant gains. Such gains are improved computational speed, cheaper circuitry and decreased power consumption.

Historically, entropy-based compression techniques offer compression ratios close to the theoretical limits. However, the emergence of the Asymmetric Numeral Systems (ANS) algorithm in 2014 by Jarosław Duda provides a new and innovative take on the family of compressors [2]. Contrary to numerous other entropy-based compressors, the algorithm offers highly competitive compression ratios while still boasting low computation times.

The thesis's main objective is to create a high-quality synthesizable hardware implementation of the Uniform Binary Variant (uABS) of ANS. This is done in hardware descriptive language using industry-standard tools. The final hardware implementation was achieved through an iterative process, first by implementing a higher level software model and a lower level software model, before implementing the hardware model. An iterative approach was needed due to the increased complexity of implementing a hardware design and the lack of software implementation.

The hardware implementation was later optimized to achieve higher performance and to ensure lightweight circuitry. A balance between high performance and resource usage (Power, Performance and Area) had to be found. This balance was struck without sacrificing precision and robustness, thus not adversely affecting compression ratios.

Lastly, the hardware designs were verified with UVM and non-UVM testbenches. The hardware implementation was tested for correctness, accuracy, and performance(compression ratios), among other metrics. It revealed that both timing and results satisfy the conditions of a full-fledged ANS compressor, where the hardware implementation yielded compression ratios converging the theoretical limit while maintaining low processing times. The verification process also revealed how the compressor was suited to handle a considerable amount of inaccuracy in the probability modelling.

The implementation reports revealed peak encoding and decoding speeds in the mega-bit range. Being able to handle sizeable bitstreams of data situates the implementation as useful for modern world applications, especially given its low resource requirements. Therefore, based on the implementation and verification reports results, one may conclude that the hardware implementation is successful.

Sammendrag

Kompresjon har en viktig rolle i dagens datasystemer. Neste generasjons kompressorer forsøker å tilfredsstille dagens økende krav om å kunne håndtere stadig økende datamengder. På tross av rask utvikling innen programvarebaserte komprimeringsalgoritmer, kan maskinvareimplementasjoner gi betydelige gevinster. Eksempler på slike gevinster er forbedret beregningshastighet, rimeligere elektronikk og redusert strømforbruk.

Historisk sett tilbyr entropibaserte kompresjonsteknikker kompresjonsforhold nær de teoretiske grensene. Da Asymmetric Numeral Systems (ANS)-algoritmen av Jarosław Duda kom i 2014 fikk verden tilgang til et nytt og innovativt medlem av kompressorfamilien [2]. Sammenlignet med andre entropibaserte kompressorer tilbyr algoritmen meget konkurransedyktige kompresjonsforhold, samtidig som den kan skilte med hurtig beregning.

Avhandlingens hovedmål er å lage en syntetiserbar maskinvareimplementasjon av høy kvalitet basert på Uniform Binary Variant (uABS) av ANS. Dette gjøres ved hjelp av maskinvare-beskrivende språk og industristandardverktøy. Den endelige maskinvareimplementasjonen ble oppnådd gjennom en iterativ prosess, først ved å implementere en programvaremodell på høynivå og så en programvaremodell på lavnivå, før maskinvaremodellen ble implementert. En iterativ tilnærming ble brukt for å håndtere maskinvaredesignkompleksiteten og en grunnleggende mangel på liknende programvareimplementasjoner. Maskinvareimplementasjonen ble senere optimalisert for å oppnå høyere ytelse og for å sikre et enkelt kretsdesign samt en skalerbar arkitektur. Det måtte samtidig oppnås en god balanse mellom forbruk, ytelse og elektronikkareal (PPA). Denne balansen ble oppnådd uten å måtte ofre presisjon og robusthet, og uten negative påvirkning på kompresjonsforholdet. Til slutt ble maskinvaredesignene verifisert gjennom bruk av både UVM og ikke-UVM testbenker. Maskinvareimplementasjonen ble blant annet undersøkt med tanke på korrekthet, nøyaktighet og ytelse (kompresjonsforhold). Verifikasjonen viste at både timing og kompresjonsresultater tilfredsstiller kravene til en fullverdig ANS-kompressor. Den endelige maskinvareimplementasjonen ga kompresjonsforhold som konvergerte mot den teoretiske grensen, samtidig som lave behandlingstider ble beholdt. Verifikasjonsprosessen viste også at kompressoren kunne håndtere en betydelig økning i unøyaktighet i sannsynlighetsmodelleringen.

Implementasjonsrapportene viste encodings- og dekodingshastigheter i megabitområdet. Selv uten ytterligere optimaliseringer vil den dermed kunne håndtere betraktelige bitstrømmer med data, og gjøre implementasjonen nyttig for moderne anvendelser, spesielt gitt det lave ressursbehovet. Ut fra resultatene av implementasjons- og verifikasjonsrapportene kan det derfor konkluderes med en vellykket maskinvareimplementasjon.

Contents

1 Introduction

1.1 Background

Data storage and communication capacity are always limited and associated with a cost. A wide range of computing devices uses compression, ranging from powerful supercomputers to Internet of Things (IoT) microcontrollers and nodes. With the desire of transmitting more data between devices, compression of data becomes even more relevant. Compression can be implemented in software (SW) or hardware (HW), with compression innovation taking occurring more rapidly in software.

Software compression utilizes the most expensive computing resources on devices, such as the central processing units (CPUs, computing cores) and the graphical processing units (GPUs). These resources must also serve other tasks, such as the user applications (apps) and the operating system (OS).

Even though software compression may offer real savings in terms of storage and data transfer if used extensively, it does take a heavy toll on both processing capacity and associated power consumption.

User perception is also a practical side of it, especially noticeable for portable equipment where higher power consumption translates into shorter runs per charge and where a sluggish response may be at least partly due to background processes such as data compression when employed there.

The environmental footprint of software compression is also significant, both in terms of the power consumption (and associated CO_2 emissions) and for the manufacture of both new equipment and the recycling of old. Moreover, it appears fair to assume that the upgrade frequency of computing devices is somewhat higher in the lack of hardware compression, with compression as an extra burden on top of running apps and OS.

Implementing compression in hardware therefore potentially offers tangible relief on all those accounts. Moreover, hardware compression ultimately places electronics designers, design houses, and chipmakers in a very privileged position, being able to implement and market well-designed hardware-based compression solutions that will benefit users, enterprises, manufacturers, and the environment.

1.2 Compression uses cases with room for innovation

There are two categories of compression algorithms: The lossless and the lossy. The lossy ones might optimize away some of the original data, preserving just as much as is necessary to be of practical use when decompressed. Many hardware compressor designs exist, supporting compression standards such as AV1, VP9, AVS2, H.264, and H.265, which are a vital integral part of today's photographic equipment, video cameras, and smartphones [3]. No one would even suggest reverting to using general computing cores for such compression operations today, with increased cost, size, and power consumption.

Lossless compression is a strong contender for general hardware-based compression implementations. As there are also many use cases where lossy compression is undesirable. Lossless compression algorithms alleviate transfer and storage bottlenecks while unambiguously reproducing the original data's exact representation.

Some of the lossless compression algorithms are limited in terms of performance or unfit for random binary data. One reason is that many well-known techniques from lossless compression perform best on higher-level data with specific characteristics and use cases, such as text compression. However, they will not compress well in other settings and are typically demanding in terms of processing requirements and memory usage. They play an important role in the compression ecosystem, however, does not cover all use cases. In particular, the lowest level of data, where there might be typically relatively random binary datasets, with large companies keeping much of their accelerators proprietary. This use case has room for innovation and improvement and is an especially important use case for hardware compressors as naturally find themselves handling binary data.

1.3 New and innovative compression candidates

Entropy-based compression techniques are known to offer compression close to theoretical limits with relatively lightweight computations. The Asymmetric Numeral Systems (ANS) algorithm by Jarosław Duda (2014) offers highly competitive compression ratios while still boasting efficient computation. Whereas there are a handful of variants, such as uABS and rANS with different features and advantages. It is a family of compressors being praised highly as being next generation and seeing many new design additions in the past years.

The ANS entropy coding principles have also influenced previously existing compression algorithms and have influenced many of the most proliferated compression standards, such as JPEG XL, Facebook/Zstandard, Google pik, Apple/LZFSE, as well as the lowest layers of compression found in iOS and macOS [4]. Whereas macOS squarely points toward the benefits of a hardware compressor implementation as well, operating at the bottom layer of the OS's themselves to the benefit of all applications running on them.

ANS performs well with relatively random data, even those featuring an uneven distribution of 1s and 0s. The so-called uniform binary variant (uABS) of ANS works well for the compression of binary data (interpreted as natural numbers) as long as the probability of the number of 1s and 0s is known. The probability can easily be computed, making a software implementation relatively straightforward to model. Whereas the algorithm itself consists dominantly of arithmetic operations and conditional statements, which are convenient to model in software.

Overall, uABS appears to be well-posed for successful low-power hardware implementation. However, no hardware implementations appear to be published or readily available for reuse as of now. This is illustrated by worldwide web searches and using database queries of reusable electronic designs such as [5].

Narrowing the research to uABS appears to therefore be a good choice, focusing on the implementation challenges since possible alternative compressors have been well covered in the preceding project thesis [6].

1.4 Research questions and contributions

The thesis's main objective is to create a high-quality synthesizable hardware implementation of the compression algorithm Asymmetric Numeral Systems (ANS) using hardware descriptive language. Specifically the uniform binary variant of the algorithm, and to verify the design. Based on this following research questions were posed:

- Is it possible to design a modular hardware implementation of the uABS algorithm using industry-standard tools which are:
  - Configurable,
  - performs close to the theoretical limits of entropy-based compression
  - and are easily implementable on commercial FPGA?
- Is it possible to design a hardware decoder performant enough to be useful to decode large streams of data at a sufficient speed for real use cases?
- Will the hardware compressor be robust enough to compress datasets with inaccurate probability modelling?

The workflow of this thesis was to first implement higher-level models, before moving to the low level of abstraction with the hardware implementations. This led to four iterations: High-level software model (C++ model), low-level software model (SV model), unoptimized ANS hardware compressor and optimized ANS hardware compressor. Each preceding step was based on the previous one, and between each step, comparisons were made to ensure the robustness and accuracy of the solution. This led to the following contributions:

- An iterative approach towards implementing an optimized hardware design, basing it on high-level and low-level software implementations, and their performance.
- A high-performance open source modular plug-and-play hardware implementation of the uABS algorithm using industry-standard tools, which was fully synthesized in an EDA. It is also easily implementable on commercial FPGA, with a configurable encoder/decoder module. The final hardware implementation was optimised in order to achieve higher performance and make the circuitry lightweight, compared to an unoptimized implementation. In order to achieve this, a trade-off analysis was performed. The analysis balances high performance and low resource usage while keeping the compressor high in efficiency and accuracy.
- The optimized and unoptimized implementations were verified through a testbench. The top testbench was simulated using a reliable verification methodology (UVM). The simulations verified that:
  - Both timing and behavior act according to the requirements of a full-fledged ANS compressor. With performance approaching the theoretical limits of entropy-based compression.
  - Peak encoding and decoding speeds were measured to be in the mega-bit range, making them useful for modern world applications. The decoder in particular is expected to perform well enough to decode streams of video.
  - It was also proven that the ANS compressor achieves accurate compression behaviour even with inaccurate probability modelling (up to $\pm 10\%$ deviation in probability).
- Implementation of two easily configurable software reference models, one high and low level, in a well-designed software simulation environment. Both software models were verified through similar testbenches as the hardware model, where the testbench tested for correct compression behaviour.

2 Theoretical background

2.1 Compression theory

Some key aspects of compression theory are presented to provide the reader with a background for understanding the compression algorithms, which will be briefly presented thereafter. In addition to those presented below, the preparatory project thesis [6] delves into more detail on other compression algorithms.

2.1.1 Data compression

The hunger for storage and communication bandwidth is high and ever increasing, due to a broad range of data intensive applications ranging from high-power supercomputer scientific applications to ultra-low-power Internet of things (IoT) applications.

Reducing the number of bits stored or transferred, and still being able to retrieve or receive them exactly as they were originally (lossless) or sufficiently recognizable (lossy) is the goal of all compression/decompression techniques [7].

There are several ways to achieve data compression, with differences in areas such as implementation complexity and suitability for different types of data [8]. The following is a brief summary of notable compression algorithms. Special emphasis is given to the Asymmetric Numeral System (ANS) and uABS (uniform binary variant) [2], the latter being the main focus of the hardware implementation of this thesis.

Compression algorithms are implemented in ways that take advantage of characteristics of the input data, such as the type of data and how the different symbols of the data type occur in the input data. Data may exhibit repeating sequences of data and statistical properties of data, such as uniformly or non-uniformly distributed symbols.

In all cases, the capabilities of compression techniques are expressed using a few important metrics. One of them is the compression ratio, which is the ratio of uncompressed data versus compressed data size [9], given by equation (1):

$$Compressionratio = \frac{UncompressedSize}{CompressedSize} \tag{1}$$

Alternatively, in order to compare the compression ratio to the Shannon source coding theorem[10] directly, the compression ratio may be expressed as, as shown in equation (2):

$$Compressionratio = \frac{CompressedSize}{UncompressedSize} \tag{2}$$

Within the field of data communication, compression will be relative to the data rate [11, 12] of the data passing by as shown in equation (3):

$$CompressionRatio = \frac{UncompressedDataRate}{CompressedDataRate} \tag{3}$$

2.1.2 Shannon's source coding theorem

Shannon's source coding theorem [10] states that the smallest possible code that can express a given piece of information (such as string of symbols) with sufficient precision and reliably is limited, thus defining a theoretical limit of data compression. The average number of bits per symbol must be equal to or larger than this limit [13], as shown in equation (4) below, with the index ”i” representing a given symbol to get coded with a given probability p, which, in summation provides the entropy H over all possible symbols X:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{4}$$

2.1.3 Lossless compression

Shannon's discussion on entropy provides a solid groundwork for how much we can compress original data composed of discrete data, yet reproduce them exactly and losslessly. Examples of discrete data are human readable texts, CAD drawings, computer software and control signals.

2.1.4 Lossy compression

Lossy compression allows for and assumes that a certain degree of information will be lost during the compression/decompression cycle, yet with a decompressed variant of the original data sufficiently similar to be of practical use [14]. The compression task thus relies on eliminating the least necessary information, in essence reducing the entropy of the data. Lossy compression is therefore used where absolute accuracy is not required and/or feasible. One example is in the use of digital audio, where the level of compression can increase while still maintaining a sufficient degree of entropy to afford a good reproduction of the original [15].

2.2 Asymmetric Numeral Systems

The Asymmetric Numeral Systems (ANS) lossless entropy based encoding method [16] describes the conversion of sequences of symbols (messages) originating from a given alphabet, to natural numbers. This is done using statistical compression schemes, where the most frequent symbols are mapped in an optimal fashion onto shorter (i.e. small) natural numbers, and less frequent symbols onto longer natural numbers, thus providing high compression ratios if the implementation works properly[13].

As an introduction to elements of the ANS scheme, a simple but still fully working encoder, assuming a binary string with equal probability of the two symbols '0' and '1' can be demonstrated. The corresponding natural number of the string can be coded, one bit at a time based upon the input string of bits $b_1b_2b_3...b_i$ where b_i is MSB. In this simple coder, new bits b_{i+1} can be inserted after multiplying the already coded number by two and adding the new bit. Encoding is presented in equation (5). Decoding follows in a similar simple fashion with equation (6), as given by [13].

$$C(x_i, b_{i+1}) = 2x_i + b_{i+1} \tag{5}$$

$$(x_i, b_{i+1}) = D(x_{i+1}) := (\lfloor \frac{x_{i+1}}{2} \rfloor, x_{i+1} \bmod 2) \tag{6}$$

It is however important to note that in this case, the equations (5) and (6) provides optimal compression according to Shannon's source coding theorem only when the probabilities of "0" and "1" are equal, i.e. there is a uniform distribution of these two symbols. In this particular case entropy is given as $H(x) = -(2 \cdot \frac{1}{2} \log_2(\frac{1}{2})) = 1$, since $p_0 = p_1 = \frac{1}{2}$.

This indicates that there is 1 bit per binary digit in the coded natural number, excluding a starting bit for the natural number $x_0 = 1$, and therefore no real gain in terms of compression of the encoder.

This can be illustrated by recalculating the resulting entropy for a slightly skewed distribution using t his coder, with $p_0 = \frac{3}{8}$ and $p_1 = \frac{5}{8}$. The resulting entropy is $H(x) = -(\cdot \frac{3}{8} \log_2(\frac{3}{8}) + \cdot \frac{5}{8} \log_2(\frac{5}{8})) \approx 0.954$. The simple ANS implementation of (5) and (6) sorely lacks statistical coding that could take advantage of the skew in distribution, taking into advantage the overrepresentation of '1' versus '0' in this example.

Coding perfectly the (q_s) symbol distribution onto a (p_s) symbol sequence, this would produce an average of $\sum_s p_s lg(\frac{1}{q_s})$ bits per symbol. The difference between this value and the optimal one is called the Kullback - Leiber distance[2], in shorthand notation given as:

$$\Delta H = \sum_s p_s lg(\frac{p_s}{q_s}) \tag{7}$$

This can be simplified as follows, using a second order Taylor's expansion of the logarithm, around 1:

$$\Delta H = \sum_s \frac{-p_s}{ln(2)}((1 - \frac{q_s}{p_s}) - \frac{1}{2}(1 - \frac{q_s}{p_s})^2) \approx 0.72 \sum_s \frac{(p_s - q_s)^2}{p_s} \tag{8}$$

Note that in the case of true entropy coding, $q_s = p_s$, and hence the difference (and distance) will be zero. It can also be seen that if not, the Kullback-Leiber distance for non-entropic coding grows for each encoded symbol contributing to deviations from the ideal of $p_s - q_s = 0$.

Utilizing the statistical properties of a data set with ANS will be further discussed in subsection 2.2.1.

The natural number (also known as state) must always be given an initial value. In other systems this value is often refereed to as check sum. This initial value could theoretically be initialized to '0', however will lead to potential loss of data. This means that the case where the natural number is initialized as $x_0 = 0$, is not guaranteed to result in the indented compression behavior. Common practice is to initialize the natural number $x_0 = 1$ [6].

Note that one important feature of ANS is that it has been shown to be suited for hardware implementation, in an adaptation known as Low Complexity ANS (LOCO ANS) [17].

2.2.1 Solving issues of basic ANS

While the simple ANS algorithm discussed previously only works optimally with uniform distribution, most data deviate more or less from this assumption, that is for $p_0 \neq p_1$, such as $\boldsymbol{p_1} = p < 1 - p = \boldsymbol{p_0}$.

Looking at equation (5) for encoding, each bit should contribute with as little entropy as possible. Optimal encoding C_{opt} may be expressed using (9) according to [13]:

$$\begin{aligned} H(C_{\text{opt}}(x_i, b_{i+1})) &= H(x_{i+1}) = H(x_i) + H(b_{i+1}) \\ &= \log_2(x_i) - \log_2(p_{b_{i+1}}) \\ &\implies C_{\text{opt}}(x_i, b_{i+1}) \approx \frac{x_i}{p_{b_{i+1}}} \end{aligned} \tag{9}$$

Using this result, a coder true to the criterium of minimizing entropy can be implemented [13], mapping odd and even binary numbers according to probabilities (p, 1-p) using the coding function in equation (10). Decoding would then be using equation (11), using the result of (9):

$$x_{i+1} = C(x_i, b_{i+1}) \tag{10}$$

$$(x_i, b_{i+1}) = D(x_{i+1}) \tag{11}$$

The three first variants of ANS (chronologically) according to the distributions of the source symbols and methods of realization are [4]:

1. Uniform Asymmetric Binary System (uABS): Only for the processing of binary symbols
2. Range Asymmetric Numeral System (rANS): Is not only '0' and '1', but with sub-cycles (ranges) as well.
3. Table Asymmetric Numeral System (tANS): Tabularizing the ANS's encoding/decoding process

These will be explained very briefly in the following sections, with special emphasis on uABS, which is the basis of the hardware implementation of this paper.

2.2.2 uABS

uABS (uniform binary variant of ANS) can compress/decompress binary messages of finite size, providing an optimal entropy coding/decoding process covering all distributions; including non-even.

Assuming a binary alphabet, and focusing on the odds, $N \cdot p$ odd numbers can be mapped in the first N natural numbers with probability p=Pr(1), so that for any given N and $N+1$ [13]:

$$\lceil (N+1) \cdot p \rceil - \lceil N \cdot p \rceil = \begin{cases} 1 & \text{, if } N \text{ has an odd mapped} \\ 0 & \text{, otherwise} \end{cases} \tag{12}$$

Note that this may be interpreted as N satisfying the outcome of $[N * p]$ in respect of odd numbers with no odd numbers expected at $N+1$.

The encoding function is presented in equation (13), and satisfies the requirements for distribution as set forth by (12) [13].

$$C(x_i, b_{i+1}) = \begin{cases} \lceil \frac{x_i+1}{1-p} \rceil - 1 & \text{, if } b_{i+1} = 0 \\ \lfloor \frac{x_i}{p} \rfloor & \text{, otherwise} \end{cases} \tag{13}$$

The decoding function presented in equation (14) is the reverse calculation, and returns the original binary sequence [13].

$$\begin{aligned} (x_i, b_{i+1}) &= D(x_{i+1}) \\ b_{i+1} &= \lceil (x_{i+1}+1) \cdot p \rceil - \lceil x_{i+1} \cdot p \rceil \\ x_i &= \begin{cases} x_{i+1} - \lceil x_{i+1} \cdot p \rceil & \text{, if } b_{i+1} = 0 \\ \lceil x_{i+1} \cdot p \rceil & \text{, otherwise} \end{cases} \end{aligned} \tag{14}$$

Knowing the probability is vital to make uABS work correctly. Obviously readily available on the coding side, but must be made available to the decoding process work. In practice, p has to be stored or transferred along the compressed data, hence requiring space and thereby reducing the effective compression ratio.

An encoding example is shown in equations (15) of the 8-bit binary sequence 10011000 to a natural number, using the uABS variant adapted from [13]. Note that p of odds (counting 1s) is $p = \frac{3}{8}$ (and hence $p = \frac{5}{8}$ for evens), and initial value (also known as state) is $x_0 = 1$.

$$\begin{aligned} x_1 &= C(x_0, b_1) = \lfloor \frac{x_0}{p} \rfloor = \lfloor 1 \cdot \frac{8}{3} \rfloor = 2 \\ x_2 &= C(x_1, b_2) = \lceil \frac{x_1+1}{1-p} \rceil - 1 = \lceil (2+1)\frac{8}{5} \rceil - 1 = 4 \\ x_3 &= C(x_2, b_3) = \lceil \frac{x_2+1}{1-p} \rceil - 1 = \lceil (4+1)\frac{8}{5} \rceil - 1 = 7 \\ x_4 &= C(x_3, b_4) = \lfloor \frac{x_3}{p} \rfloor = \lfloor 7 \cdot \frac{8}{3} \rfloor = 18 \\ x_5 &= C(x_4, b_5) = \lfloor \frac{x_4}{p} \rfloor = \lfloor 19 \cdot \frac{8}{3} \rfloor = 48 \\ x_6 &= C(x_5, b_6) = \lceil \frac{x_5+1}{1-p} \rceil - 1 = \lceil (48+1)\frac{8}{5} \rceil - 1 = 78 \\ x_7 &= C(x_6, b_7) = \lceil \frac{x_6+1}{1-p} \rceil - 1 = \lceil (78+1)\frac{8}{5} \rceil - 1 = 126 \\ x_8 &= C(x_7, b_8) = \lceil \frac{x_7+1}{1-p} \rceil - 1 = \lceil (126+1)\frac{8}{5} \rceil - 1 = 203 \end{aligned} \tag{15}$$

The corresponding decoding of the natural number 203 of uABS variant is shown in (16) [13]. Note that it is generated in reversed order.

$$
\begin{aligned}
&b_8 = \lceil (x_8+1)\cdot p\rceil - \lceil x_8\cdot p\rceil = \lceil (203+1)\cdot \frac{3}{8}\rceil - \lceil 203\cdot \frac{3}{8}\rceil = 0\\
&x_7 = x_8 - \lceil x_8\cdot p\rceil = 203 - \lceil 203\cdot \frac{3}{8}\rceil = 126\\
&b_7 = \lceil (x_7+1)\cdot p\rceil - \lceil x_7\cdot p\rceil = \lceil (126+1)\cdot \frac{3}{8}\rceil - \lceil 126\cdot \frac{3}{8}\rceil = 0\\
&x_6 = x_7 - \lceil x_7\cdot p\rceil = 126 - \lceil 126\cdot \frac{3}{8}\rceil = 78\\
&b_6 = \lceil (x_6+1)\cdot p\rceil - \lceil x_6\cdot p\rceil = \lceil (78+1)\cdot \frac{3}{8}\rceil - \lceil 78\cdot \frac{3}{8}\rceil = 0\\
&x_5 = x_6 - \lceil x_6\cdot p\rceil = 78 - \lceil 78\cdot \frac{3}{8}\rceil = 48\\
&b_5 = \lceil (x_5+1)\cdot p\rceil - \lceil x_5\cdot p\rceil = \lceil (48+1)\cdot \frac{3}{8}\rceil - \lceil 48\cdot \frac{3}{8}\rceil = 1\\
&x_4 = \lceil x_5\cdot p\rceil = \lceil 48\cdot \frac{3}{8}\rceil = 18\\
&b_4 = \lceil (x_4+1)\cdot p\rceil - \lceil x_4\cdot p\rceil = \lceil (18+1)\cdot \frac{3}{8}\rceil - \lceil 18\cdot \frac{3}{8}\rceil = 1\\
&x_3 = \lceil x_4\cdot p\rceil = \lceil 18\cdot \frac{3}{8}\rceil = 7\\
&b_3 = \lceil (x_3+1)\cdot p\rceil - \lceil x_3\cdot p\rceil = \lceil (7+1)\cdot \frac{3}{8}\rceil - \lceil 7\cdot \frac{3}{8}\rceil = 0\\
&x_2 = x_3 - \lceil x_3\cdot p\rceil = 7 - \lceil 7\cdot \frac{3}{8}\rceil = 4\\
&b_2 = \lceil (x_2+1)\cdot p\rceil - \lceil x_2\cdot p\rceil = \lceil (4+1)\cdot \frac{3}{8}\rceil - \lceil 4\cdot \frac{3}{8}\rceil = 0\\
&x_1 = x_2 - \lceil x_2\cdot p\rceil = 4 - \lceil 4\cdot \frac{3}{8}\rceil = 2\\
&b_1 = \lceil (x_1+1)\cdot p\rceil - \lceil x_1\cdot p\rceil = \lceil (2+1)\cdot \frac{3}{8}\rceil - \lceil 2\cdot \frac{3}{8}\rceil = 1\\
&x_0 = \lceil x_1\cdot p\rceil = \lceil 2\cdot \frac{3}{8}\rceil = 1
\end{aligned}
\tag{16}
$$

2.3 General HW implementation

2.3.1 FPGA

A wide scale of standalone field programmable gate array (FPGA) chips, as well as embedded FPGA functionality on multi-function chips are manufactured. Several are well posed as possible targets for a hardware implementation of the compressor in this project.

As an introduction to this section of theory on FPGAs, one may illustrate the core functionality with figure 1, courtesy National Semiconductors [18]. This shows the three basic FPGA functionalities of logic, routing and I/O.

Logic, as embedded in so-called Configurable Logic Blocks (CLBs) typically contains a number of slices, each with flip/flops, look-up-tables (LUTs), registers, multiplexers, wiring, carry logic, arithmetic logic and storage logic and similar. Some CLB slices also include distributed RAM and 32-bit shift registers [19].

Block memory, DRAM, registers and FIFO queues are commonplace in FPGAs today, providing easy access to different types of storage, with different strengths and weaknesses.

I/O is organised in Input/Output Blocks (IOBs) where each of these handles a certain type of I/O [20]. Assigning IOBs to physical ports, they become bonded IOBs. IOBS is connected to an actual physical wire going out of the chip. Physical connect can be such as PXI backplanes or Real-Time System Integration (RTSI) connectors. Note that non-bondable IOBs encompass (among others)

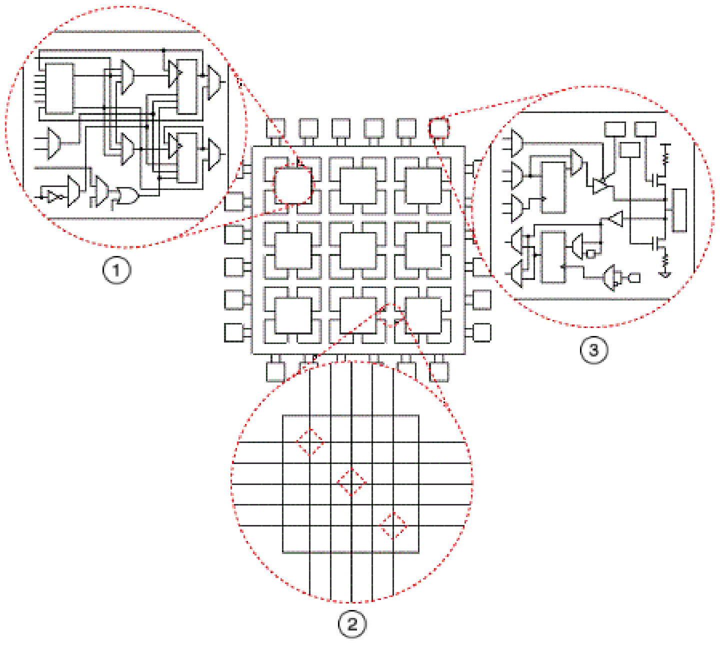


Figure 1: Configurable Logic Blocks, routing and I/O on FPGAs. Illustration taken from [18]

internal lines that provide I/O between configurable and non-configurable parts within the FPGA [20].

Routing allows for the transport of data to and from the different parts of the FPGA.

For small-scale system design, manually wiring components on an FPGA provides tight optimization but requires an intimate understanding of the FPGA components and their uses. To be productive at scale, high-level tools converting statement-based designs expressed in either VHDL or Verilog/SystemVerilog offers good results as well, given a performant EDA for synthesis.

The FPGAs range from traditional standalone chips such as the 7-series of Xilinx (starting with the Spartan-7) to a variety of FPGA functionality on multiprocessor on a chip (MPSoC) systems.

2.3.2 Xilinx target FPGAs

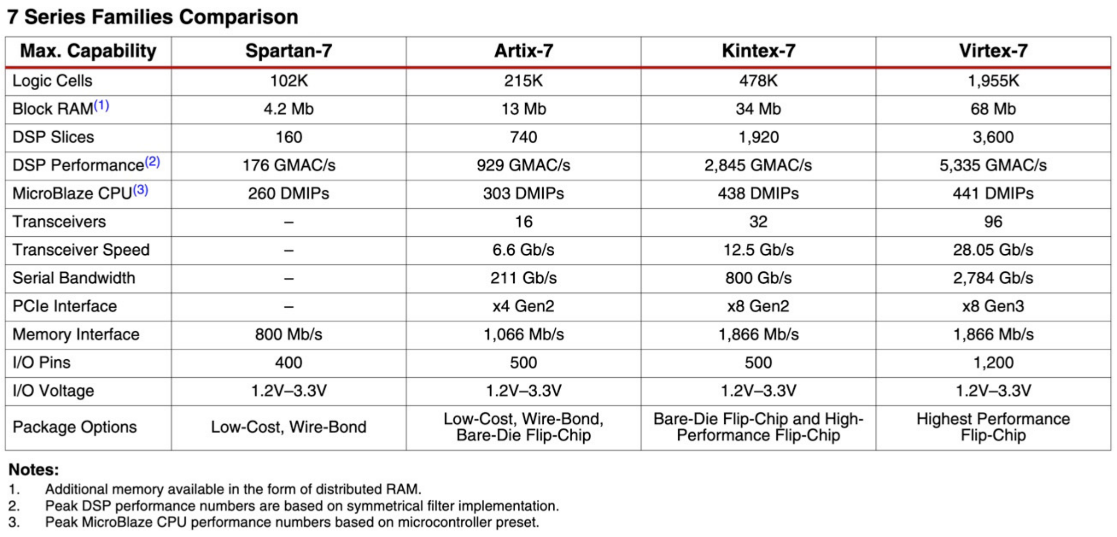
A possible standalone-FPGA target for a hardware implementation of the compressor in this project is the Xilinx XC7S100 Spartan-7, the 7-series entry-point low-power FPGA.

Figure 2: Xilinx Spartan-7 FPGA, illustration taken from [21]

The best quipped Spartan-7 has 1100KB DRAM and 1 analog to digital converter (XADC), based on a 28 nm process [21]. The 7 Series families is summarized in table 1 and 2 for supplemental info.

Xilinx offers the SP701 Evaluation Kit for prototyping and development using the top-of-the Spartan-7 variant Xilinx XC7S100. Third-party vendors are marketing other more or less elaborate development kits and hobby kits based on the Spartan-7, such as a small-factor board IAM

Table 1: Xilinx 7-series overview, illustration taken from [21]

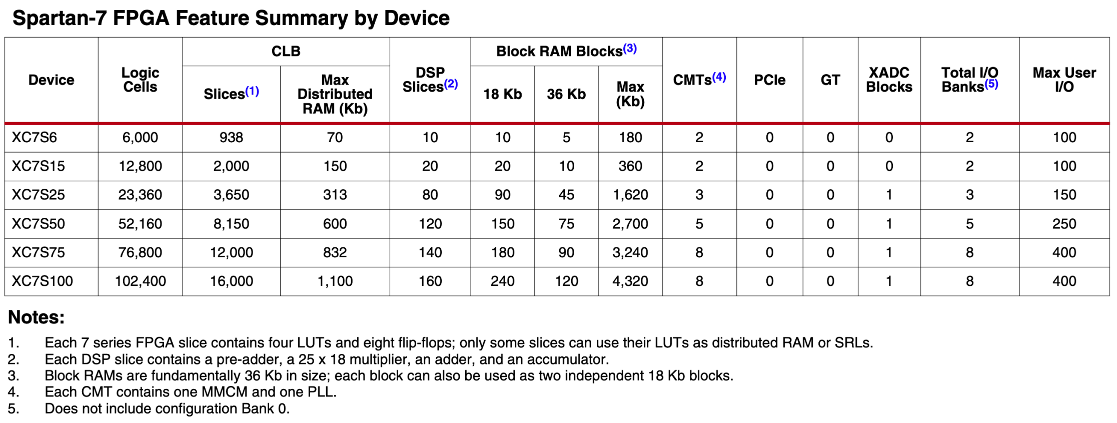


7 Series Families Comparison

Max. Capability	Spartan-7	Artix-7	Kintex-7	Virtex-7
Logic Cells	102K	215K	478K	1,955K
Block RAM(1)	4.2 Mb	13 Mb	34 Mb	68 Mb
DSP Slices	160	740	1,920	3,600
DSP Performance(2)	176 GMAC/s	929 GMAC/s	2,845 GMAC/s	5,335 GMAC/s
MicroBlaze CPU(3)	260 DMIPs	303 DMIPs	438 DMIPs	441 DMIPs
Transceivers	–	16	32	96
Transceiver Speed	–	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Serial Bandwidth	–	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	–	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	800 Mb/s	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	400	500	500	1,200
I/O Voltage	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V	1.2V–3.3V
Package Options	Low-Cost, Wire-Bond	Low-Cost, Wire-Bond, Bare-Die Flip-Chip	Bare-Die Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip

Notes:
1. Additional memory available in the form of distributed RAM.
2. Peak DSP performance numbers are based on symmetrical filter implementation.
3. Peak MicroBlaze CPU performance numbers based on microcontroller preset.

Table 2: Spartan-7 FPGA Feature Summary by Device, illustration taken from [21]



Spartan-7 FPGA Feature Summary by Device

Device	Logic Cells	CLB		DSP Slices(2)	Block RAM Blocks(3)			CMTs(4)	PCIe	GT	XADC Blocks	Total I/O Banks(5)	Max User I/O
		Slices(1)	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7S6	6,000	938	70	10	10	5	180	2	0	0	0	2	100
XC7S15	12,800	2,000	150	20	20	10	360	2	0	0	0	2	100
XC7S25	23,360	3,650	313	80	90	45	1,620	3	0	0	1	3	150
XC7S50	52,160	8,150	600	120	150	75	2,700	5	0	0	1	5	250
XC7S75	76,800	12,000	832	140	180	90	3,240	8	0	0	1	8	400
XC7S100	102,400	16,000	1,100	160	240	120	4,320	8	0	0	1	8	400

Notes:
1. Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
2. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.
3. Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
4. Each CMT contains one MMCM and one PLL.
5. Does not include configuration Bank 0.

Electronics [22] kit for under USD100.

The Xilinx Zynq UltraScale+ series MPSoC, which is based on a 16 nm-process [23], was chosen as the primary FPGA target in this project. It provides a host of applications and real-time Arm core processors as well as several memory and connectivity options.

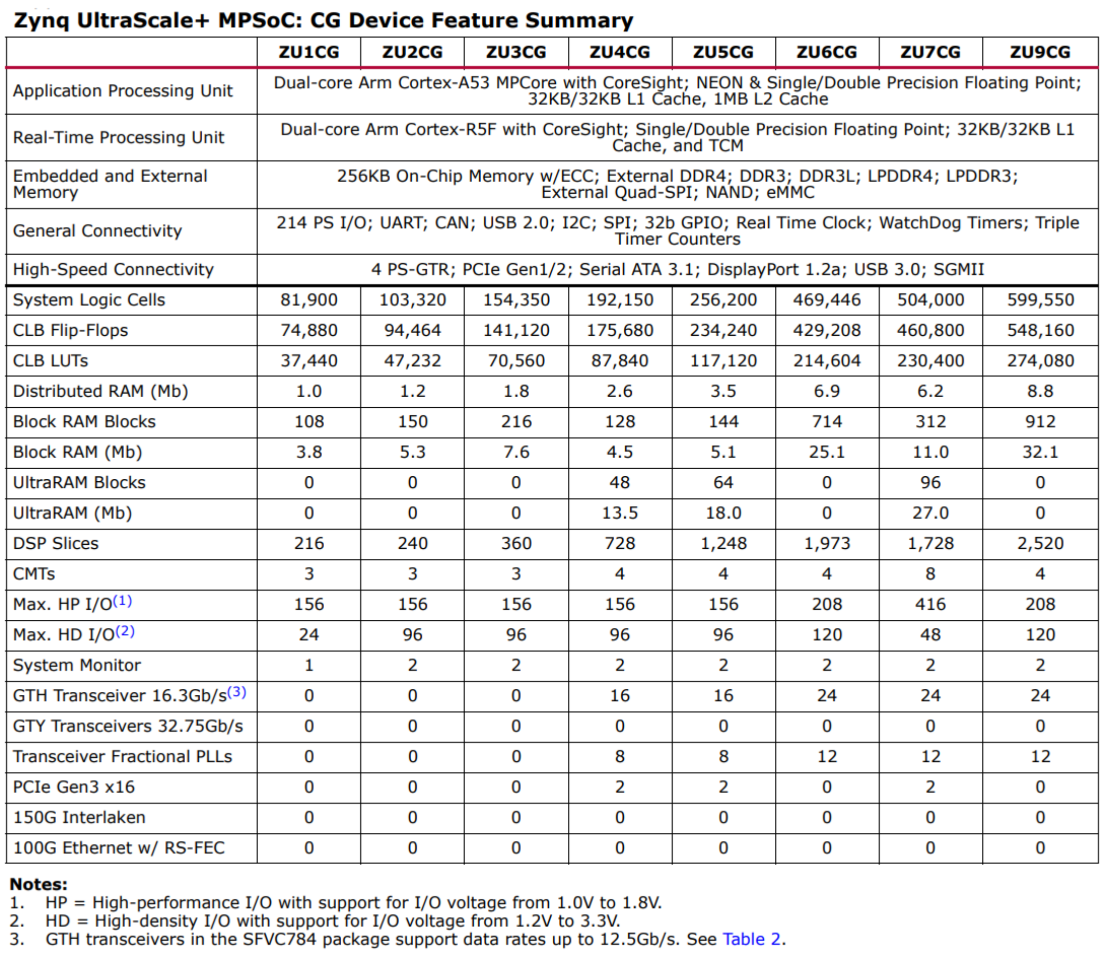
Figure 3: Xilinx Zynq UltraScale+, illustration taken from [23]

Table 3, Zync UltraScale+ MPSoC: CG Device Feature Summary for a presentation of the capabilities of the series, comparable to those of 7-Series in table 1 and of Spartan-7 in table 2.

Xilinx offers development kits based on Zynq UltraScale+ as well, such as the Zcu106 [24].

It may be worthwhile to note that the UltraScale+ series fans out at 600K system logic cells; while the 7-series goes on to 1955K.

Table 3: Xilinx Zynq UltraScale+ series overview, illustration taken from [23]



Zynq UltraScale+ MPSoC: CG Device Feature Summary

	ZU1CG	ZU2CG	ZU3CG	ZU4CG	ZU5CG	ZU6CG	ZU7CG	ZU9CG
Application Processing Unit	Dual-core Arm Cortex-A53 MPCore with CoreSight; NEON & Single/Double Precision Floating Point; 32KB/32KB L1 Cache, 1MB L2 Cache							
Real-Time Processing Unit	Dual-core Arm Cortex-R5F with CoreSight; Single/Double Precision Floating Point; 32KB/32KB L1 Cache, and TCM							
Embedded and External Memory	256KB On-Chip Memory w/ECC; External DDR4; DDR3; DDR3L; LPDDR4; LPDDR3; External Quad-SPI; NAND; eMMC							
General Connectivity	214 PS I/O; UART; CAN; USB 2.0; I2C; SPI; 32b GPIO; Real Time Clock; WatchDog Timers; Triple Timer Counters							
High-Speed Connectivity	4 PS-GTR; PCIe Gen1/2; Serial ATA 3.1; DisplayPort 1.2a; USB 3.0; SGMII							
System Logic Cells	81,900	103,320	154,350	192,150	256,200	469,446	504,000	599,550
CLB Flip-Flops	74,880	94,464	141,120	175,680	234,240	429,208	460,800	548,160
CLB LUTs	37,440	47,232	70,560	87,840	117,120	214,604	230,400	274,080
Distributed RAM (Mb)	1.0	1.2	1.8	2.6	3.5	6.9	6.2	8.8
Block RAM Blocks	108	150	216	128	144	714	312	912
Block RAM (Mb)	3.8	5.3	7.6	4.5	5.1	25.1	11.0	32.1
UltraRAM Blocks	0	0	0	48	64	0	96	0
UltraRAM (Mb)	0	0	0	13.5	18.0	0	27.0	0
DSP Slices	216	240	360	728	1,248	1,973	1,728	2,520
CMTs	3	3	3	4	4	4	8	4
Max. HP I/O(1)	156	156	156	156	156	208	416	208
Max. HD I/O(2)	24	96	96	96	96	120	48	120
System Monitor	1	2	2	2	2	2	2	2
GTH Transceiver 16.3Gb/s(3)	0	0	0	16	16	24	24	24
GTY Transceivers 32.75Gb/s	0	0	0	0	0	0	0	0
Transceiver Fractional PLLs	0	0	0	8	8	12	12	12
PCIe Gen3 x16	0	0	0	2	2	0	2	0
150G Interlaken	0	0	0	0	0	0	0	0
100G Ethernet w/ RS-FEC	0	0	0	0	0	0	0	0

Notes:
1. HP = High-performance I/O with support for I/O voltage from 1.0V to 1.8V.
2. HD = High-density I/O with support for I/O voltage from 1.2V to 3.3V.
3. GTH transceivers in the SFVC784 package support data rates up to 12.5Gb/s. See Table 2.

2.3.3 Briefly about thermal considerations

FPGA-related thermal design issues have not been taken into consideration in this project, due to time and project constraints. This can however be an important issue when designing applications for FPGAs. Agne & al [25] present an interesting discussion on the impact of different FPGA elements on power consumption, demonstrated by using the Xilinx S7 and other FPGAs. Certain uses can easily overheat the chip and/or create challenges for the power budget.

Apart from design changes to avoid some of the most notorious power guzzlers in the FPGA port logic design, some relief can be found by using processes with smaller feature sizes, such as 20 nm, 16 nm or below. One example is the Zynq Ultrascale+ [23] series based on 16 nm, providing 2–5X greater system-level performance/watt over 28nm devices [26].

In addition to smaller feature size, other design choices can lower the power consumption in the logic gate functions, with enhanced logic blocks for a target of 90% utilization for the UltraScale+ [26]. If on-chip elements such as processing cores can be used in a sensible manner, reducing the use of off-chip (MPSoC) functions, further power savings can be achieved. Note that this will not necessarily lower related chip heating issues per se since even more power consumption is crammed into a small space by using other functions on the MPSoC. But if for instance the CPU cores of the UltraScale+ can be used together with the logic gates on-chip, real power savings can be achieved compared to the total power consumption of a system forced to intensive I/O activity for intermediate result data transport passed via the power-wise expensive external lines of the chips involved.

2.3.4 Timing: Setup and hold slack

Signal propagation through high speed/high-frequency circuitry such as FPGA or ASIC does pose some challenges. Looking into a propagation path, one will either experience that signal propagation is well timed with data in place before being clocked (and locked) in place, or that it fails to do so.

There are two variants of setup and hold slack: Setup slack, and hold slack.

The difference between desired time (clock propagation) and achieved time (data propagation) for a timing path is called the setup slack. It may either be: positive, zero or negative. Negative simply means that the signals fail to propagate within the given time frame, while zero means that it does succeed but without any margin. Zero or positive means that the data has propagated successfully, and in time [27]. Please refer to figure 4 of [28] for an example of setup timing.

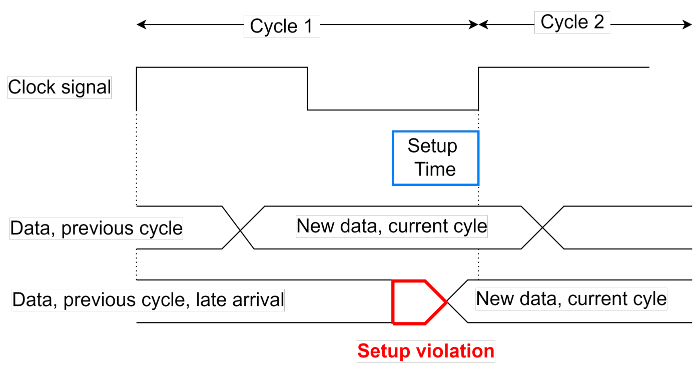


Figure 4: Signal setup timing

The difference between the achieved and desired time for locking the data is called hold slack. It simply means that if data are in place too early, and not stable during the hold, locking will fail. Please refer to figure 5 of [28] for an example of hold timing.

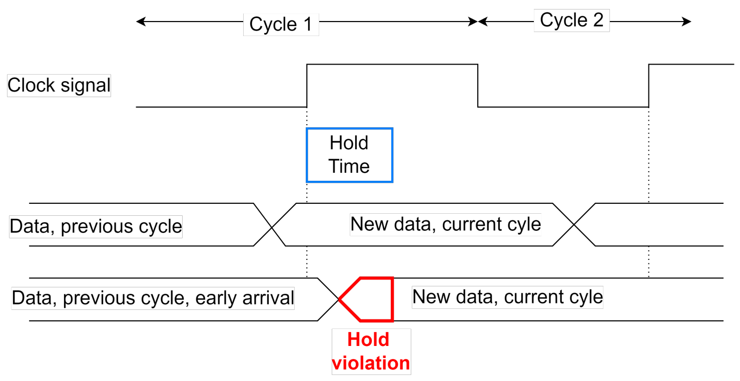


Figure 5: Signal hold timing

The libraries of the synthesis and verification tools know the different signal propagatíon data of the elements used. Proper design will minimize the probability of zero and negative setup and hold slack, and verification will minimize the chance of not finding any timing errors that might slip through.

2.3.5 The Ready-Valid data interfacing design pattern

One of the challenges of electronic design is proper handshaking when transferring data from a source to a sink. The handshake is used for the sink to signal that it is ready for data, and the source to signal that it will transmit data.

Simple solution patterns such as the ReadyValid data interfacing design patterns, have been with us for a while as exemplified by [29] and illustrated in figure 6.

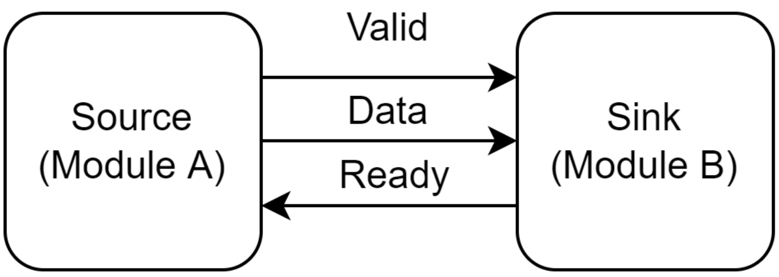


Figure 6: Ready/Valid interfacing design pattern principle with one source and one sink

This pattern relies on two handshake signals: Valid from source to sink, signalling that the data presented are to be consumed, and ready from sink to the source, signalling that the sink is indeed ready to consume. Both will have to be true in order to facilitate a successful data transfer, locking in (and holding) data on the sink. Note that the original design pattern relies on the rising edge of the clock to hold (lock) the data on the sink as the receiving end, as illustrated in figure 7.

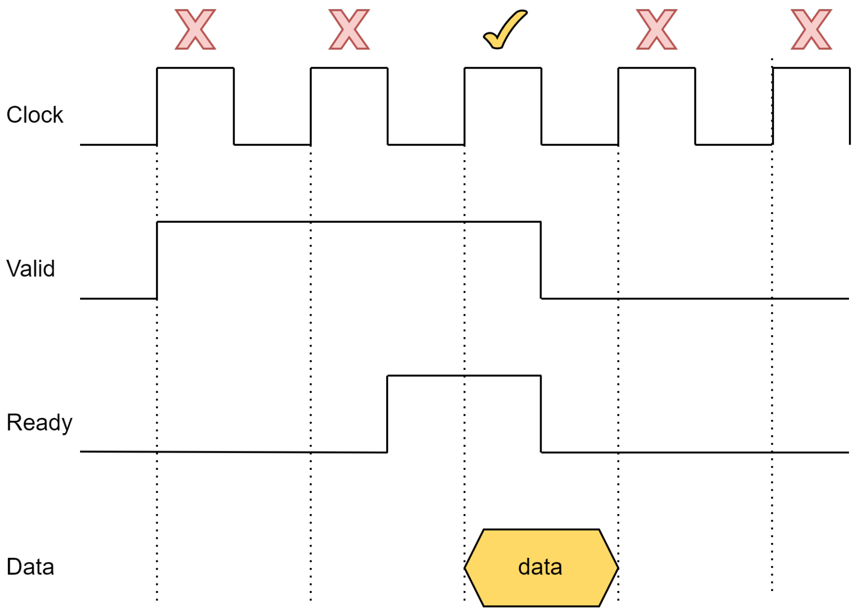


Figure 7: Ready/Valid signals vs. transfer on rising edge, one clock cycle data transfer

This pattern is not limited to the simple stop-and-go timing as shown here; but can string a number of transfers, as long as both Valid and Ready er true, as shown in figure 8.

Referring to the discussion of section 2.3.4, special caution must be exercised to avoid negative setup and hold slack.

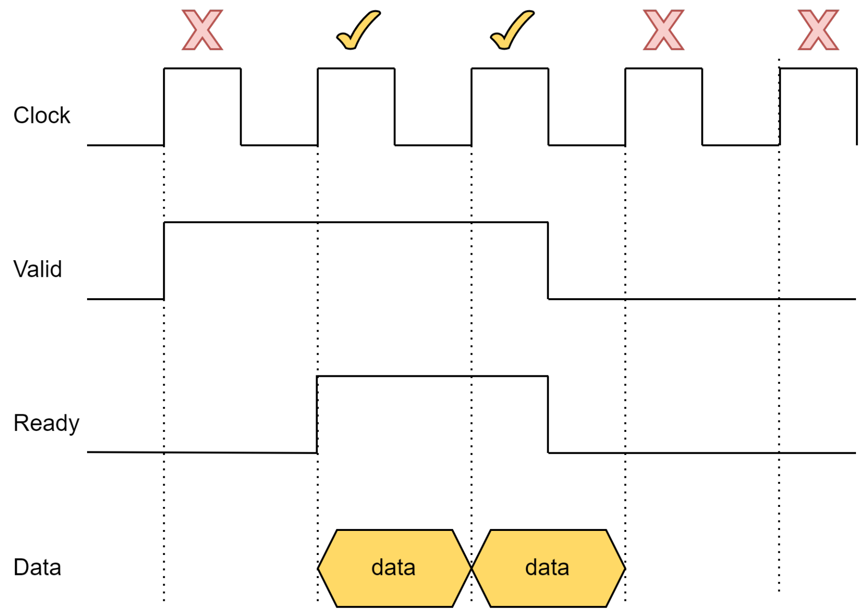


Figure 8: Ready/Valid signals vs. transfer on rising edge, two clock cycle data transfer

2.3.6 FPGA divide library

The different FPGAs are delivered with a wide array of functionalities as defined by the FPGA itself, and the standard libraries of synthesizable functions of the different vendors and third parties.

The fixed-point division is not available out-of-the-box with Verilog with FPGAs. ANS type compression requires division; and luckily, additional libraries are available.

One of the open libraries, licensed under an open-source MIT licence, is Project F: FPGA Dev [30]. This library for the fixed point division of binary integers requires one clock cycle per bit involved, up to 32 bits (depending on design parameters). The run-time compares favourably with a modern CPU such as Intel Skylake, requiring 42-95 cycles for a 64-bit signed integer division.

Traditional long division per K-9 elementary school recipe is used. A long division algorithm is shown in figure 9, according to [31]:

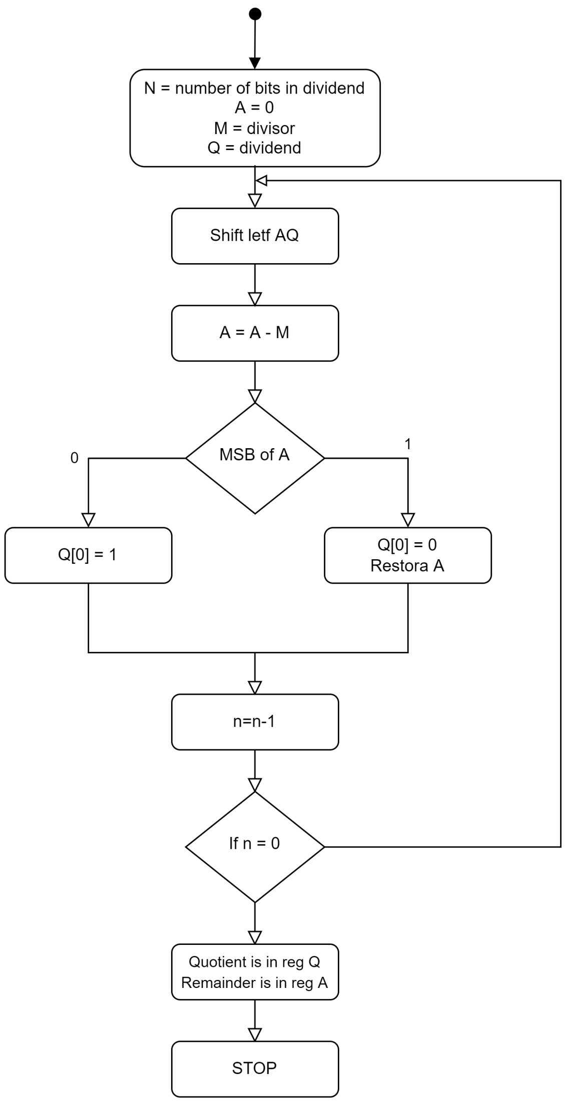


Figure 9: Binary division per long division method

2.3.7 FPGAs and fixed point numbers

In need of decimals, simple fixed point maths can often be a good alternative to computationally expensive floating point numbers, as long as the number range is relatively limited. Fixed point math is definable in Verilog, reaping the advantage of DSP blocks for fast multiplications and additions. Such libraries exist from third parties for example from Project F: FPGA Dev [32].

The bits in a FP integer are divided in two: The integer part, and the fractional part, as illustrated in figure 10:

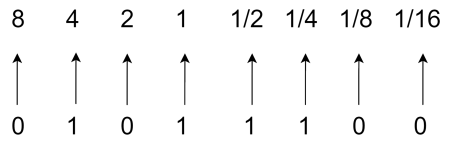


Figure 10: An example fixed point number 0100.1100_2 equalling 4.75_{10}

Range in fixed point math can be improved by adding digits in the integer part of the number, and precision by adding digits in the fractional part.

Extending range can also be achieved, by adding a scaling factor. This would have to be stored and taken into account whenever calculus is performed, but does go a long way towards floating point accuracy and range.

Implementing libraries for fixed point should also handle cases of over- and underflow, that is when the sign changes contrary to the operation, such as two positive fixed point numbers being added, ersulting in a negative

2.4 Using SystemVerilog for synthesis

SystemVerilog is a IEEE standard for both circuitry synthesis (design) and verification, existing side-by side with VHDL serving the same purposes, but doing so slightly differently [33]. One of the primary goals of the original Verilog, and of the later incarnation dubbed SystemVerilog has indeed been to facilitate implementations of complex hardware designs with better accuracy and fewer code lines. Synthesis has thus been a part of the standard from the start, with a steady evolution of mechanisms since the very inception in 1984, through the first standardisation in 1995 (known as Verilog-1995) up to the more recent SystemVerilogs (2005/2009/2012/2017).

There was a name change from Verilog to SystemVerilog in 2005, as a merge of the different developments, resulting in a host of new concepts for both design and verification. Versions 2009 and 2012 followed in the same trait, with the 2017 standard primarily as a roll-up with fixes and more firm and unambiguous definitions.

Todays SystemVerilog in it's different implementations offer a more hand-in-hand integrated design- and verification cycle than ever, as illustrated in figure 11.

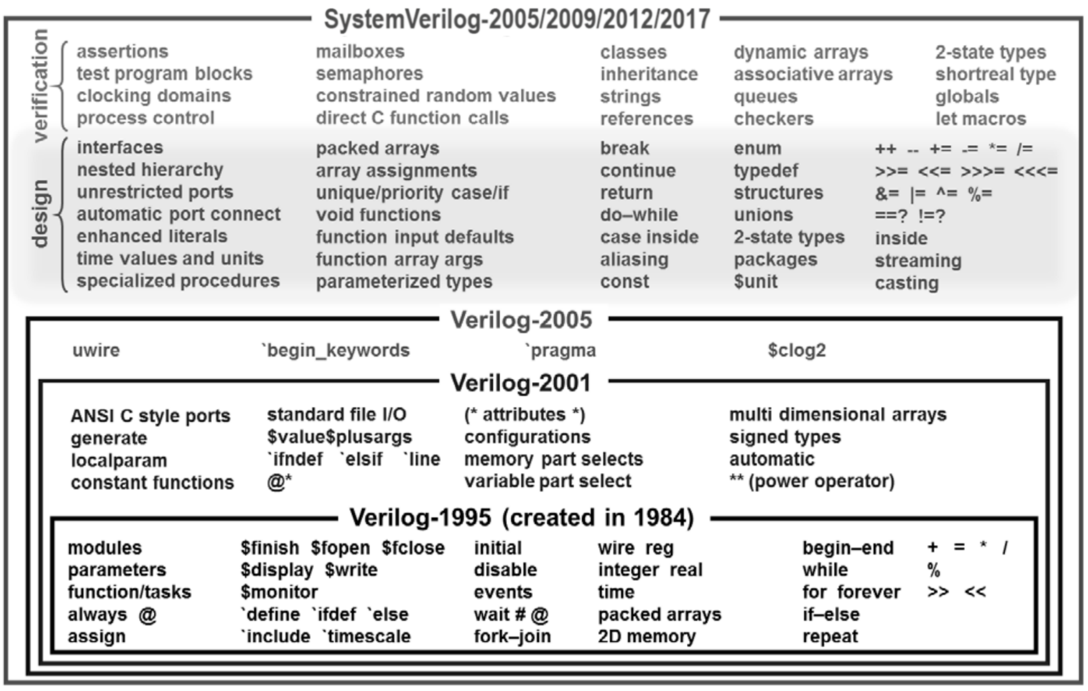


Figure 11: System Verilog ancestry for design and verification, versions 2005-2017. The illustration is taken from [34]

There exist several tools with SystemVerilog support, such as the suite of synthesis (design) and verification tools from Synopsis, highly praised by Sutherland and Mills [34]. The company Accel-era, originator of many of the concepts behind Verilog and SystemVerilog, still resides firmly at the center of much of the developments and standardisation activity, donating this basis to IEEE as one of the key organizers. There is a comprehensive (although somewhat outdated) list of vendors by EEtimes [35].

2.5 State machines

Since so much of the design work is directly or indirectly related to state machines, it may be in order to briefly sum up some vital and relevant state machine concepts; avoiding the more formal notations here, and focusing on the more computer/electronics related terms.

A very brief, but hopefully sufficiently firm definition is that a state machine is a concept (or abstraction) with a (normally limited [finite] set of) states S it may enter, starting with a state s_0 [36]. The state machine can receive responses to events (signals) based on rules associated with the states and transitions between them. It can either remain in the same state (after making some processing) and either return to the same state. There are a number of responses that may come from the state machine, such as whether the events were understood (valid/interpretative), outcomes, state information and so on.

There are several models for detailing state machines that are typically used when dealing with electronics and programming; each with its advantages, such as Moore and Mealy. Mealy was chosen, due to advantages such as allowing output to depend directly on the present state as well as present input (and hence saving cycles and providing faster reaction to input on transitions), allowing for a smaller number of states, but potentially at the expense of more design challenges [37].

The illustration of a Mealy state machine in figure 12, courtesy of a Verilog doc page, illustrates these concepts nicely [38]:

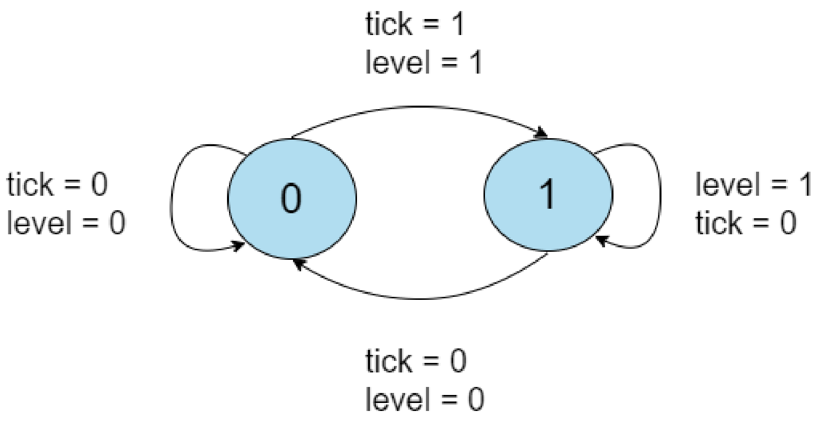


Figure 12: Example of a simple Mealy state machine [38]

The state machine will transition from state ”zero” to ”one” when the signals ”level” and ”tick” are both 1, while entering state ”zero” if both are 0. The reader will notice that level = 0 and tick = 0 will only loop back to ”zero” when already there, possibly returning an output to the surroundings in the transition back to where it were, and likewise return to state ”one” if level = 1 and tick = 0.

Other signal input combinations are undefined, and must be ignored during processing, and are often referred to as ”don´t-cares”.

Using SystemVerilog to define a state machine, verification can be planned and enforced in a coherent manner, please see chapter 2.6 for more details.

When finally on silicon, test vectors with all possible combinations of test signals (inputs) and outputs may be used to check hat the state machine is behaving as planned, ignoring the ”don´t cares” mentioned above.

2.6 UVM

Universal Verification Methodology (UVM) is standardised by the IEEE in 1800.1, and is a class-based verification library with support for reuse methodology for SystemVerilog [39]. UVM has a feature rich class library providing basic bulding blocks for creating verification data and components [40]. A typical workflow defines data stimulus with help from building blocks of the UVM class library, creating configurable (and reusable) UVM Verification Components (UVCs) for stimuli to device under test (DUT). Multiple UVCs can be combined in the verification environment, with scoreboards and register models. UVM contains a transaction library model (TLM) and configuration database, and enables faster development and more rapid verification [39].

UVM is primarily derived from Open Verification Methodology (OVM) [41]. It has seen widespread support by industry leading EDA vendors such as Aldec, Cadence, Mentor and Synopsis. An advantage with UVM has is its layer of abstraction where each distinct component int he verification environment has it's specific purpose. Such as where the driver class object only drives the signals to the Device Under Test (DUT). On the other hand the monitor has the role of monitoring the design interface. This division into classes with specific purposes makes the process of configuring and using UVM intuitive and well defined.

2.6.1 Standard UVM testbench architecture

The standard testbench architecture is seen in 13. This architecture follows the terminology recommended by standards organization Accellera. Their standards are widely adopted in the industry and a trusted reference [39].

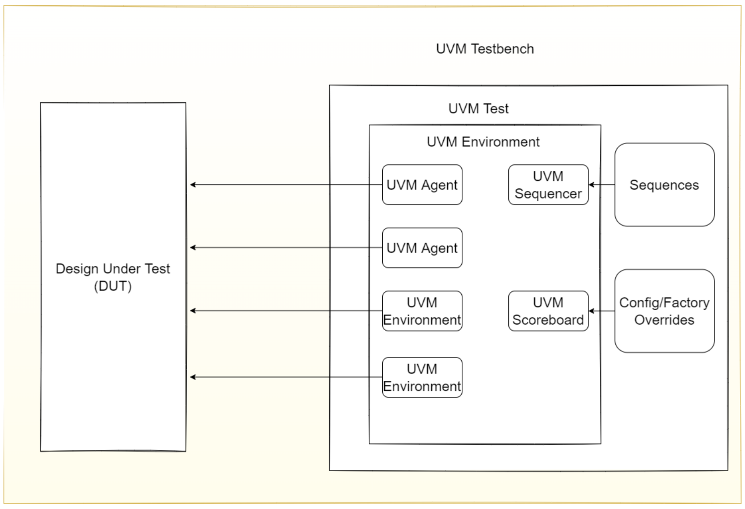


Figure 13: Standard UVM Testbench Architecture

2.6.2 UVM testbench

One of the most important roles of the UVM Testbench is to instantiate the DUT module and the UVM Test class [41]. It also configures the connection there between. Other verification modules may also be instantiated in the testbench as well. The UVM Testbench is instantiated dynamically during run-time. This feature allows the UVM Testbench to be compiled once and ran with any number of different tests. Lastly in some architectures the UVM testbench is used to refer to a special module encapsulating verification collaterals only, which are in turn integrated up with the DUT.

2.6.3 UVM test

The top-level UVM component in the UVM Testbench is the UVM Test [41]. Typically the UVM Test is tasked with performing three main functions. It instantiates the top-level environment, configures the environment and applies stimulus. The configuration may include factory overrides or configuration of database, and the invocation of UVM sequences are done through the environment to the DUT.

A standard approach is to utilise one base UVM Test with the UVM Environment instantiations including other common items. Following are individual tests extending the base test (e.g. extending the base class). This may be done to for instance configure the environment differently, select different sequences or change constraints.

2.6.4 UVM environment

In order to group together other interrelated verification components, the hierarchical component UVM Environment is required [41]. Some UVM components which are typically instantiated inside the UVM Environment are UVM Agents, UVM Scoreboards as well as other UVM Environments. As a rule of thumb, the top-level UVM Environment is desired to encapsulate the verification components targeting the DUT.

A typical application of UVM Environment for a SoC will entail one UVM Environment per IP. This could be for instance a PCIe Environment,USB Environment or a Memory Controller Environment. The usage of Cluster Environments is done in order to group together IP Environments. This could for instance take shape as an IO Environment or a Processor Environment, where a grouping of UVM Environments would be satisfactory.

2.6.5 UVM scoreboard

Behaviour of targeted DUTs are mainly checked by the UVM Scoareboard [41]. Usually the UVM Scoreboard receives transactions containing inputs and outputs of the DUT via UVM Agents analysis ports. While running the input transactions through a predictor, which is a reference model. Thereby producing expected transactions which compare the expected output to that of the actual output.

Different methodologies are employed for scoreboard implementation. In the industry, there are various approaches on how the reference model should work and how the communication should take place between the scoreboard and the testbench.

2.6.6 UVM agent

The UVM Agent works as a hierarchical component which groups other verification components together [41]. Being components that deal with a specific DUT interface. It operates both in active and passive mode. It is is capable of generating stimulus in active mode and could for instance in passive mode monitor the interface without actually controlling it. The typical UVM Agent includes the following components as seen in this list:

1. UVM Sequencer for stimulus flow management.
2. UVM Driver to drive the stimulus to the DUT interface.
3. UVM Monitor for monitoring the DUT interface.

Some less common yet important components may be seen in this list:

1. Coverage collectors

2. Protocol checkers
3. TLM model

2.6.7 UVM sequencer

The UVM Sequencer controls the flow of the UVM Sequence Item transactions, which are generated by one or more UVM sequences [41]. It is thus seen that transaction flow from multiple stimulus sequences is handled by the UVM Sequencer, where it may serve as an arbiter.

2.6.8 UVM sequence

Stimulus generation behavior is found within the UVM Sequence object, where the UVM Sequences are not seen as part of the component hierarchy [41]. The UVM Sequences can be persistent or transient. Where a UVM Sequence instance may exist for any given duration. It may be short-lived for a single transaction or be used to drive stimulus for the entirety of a simulation. The UVM Sequences may themselves be seen hierarchically. In this case, there may be a parent sequence which invokes a child sequence.

Each UVM sequence has to be at some point bound to a UVM sequencer during operation if it is intended for usage. Whereas several UVM Sequence instances may be bound to the very same UVM Sequencer.

2.6.9 UVM driver

Individual UVM Sequence item transactions from the UVM Sequencer are received by the UVM Driver [41]. The UVM Sequences are then driven to the DUT Interface. The UVM Driver thereby spans through the abstraction levels as it is converting transaction-level stimulus to pin-level stimulus. The Driver is also equipped with Transaction Level Modelling (TLM) ports to receive transactions from the Sequencer, and in order to gain access to the DUT interface to drive the signals.

2.6.10 UVM monitor

Samples from the DUT interface are collected by the UVM Monitor [41]. This information of the transactions are then relayed to the rest of the UVM Testbench for further analysis. In similar fashion to the UVM Driver the UVM Monitor may be seen to span several abstraction levels since it converts pin-level activity to transactions. Therefore the UVM Monitor is generally granted access to the DUT interface while still yielding a TLM analysis port to broadcast the created transactions.

UVM Monitors is able to perform some processing internally for the transactions. This could for instance include coverage collection, checking, logging or recording. These processes may also be delegated to dedicated components that are connected to the monitor's analysis port.

2.6.11 UVM phases

All testbench components derive from the uvm_component class, and all such unified verification classes (UVCs) must be able to respond to virtual functions (tasks) as defined per UVM Phases, [42] in figure 14, see also figure 14 slightly adapted from [43]. Rooted in the uvm_component class, all objects will be instantiated and ready to respond to the different phases of the simulation, see figure 14. Build phases create and connect testbench objects, before being connected. The actual time-consuming simulation then follows, in one or several "run" UVM phases, that will be executed in parallel if there are more than one. After simulation, data are extracted, before being checked versus expected values from design, and reported.

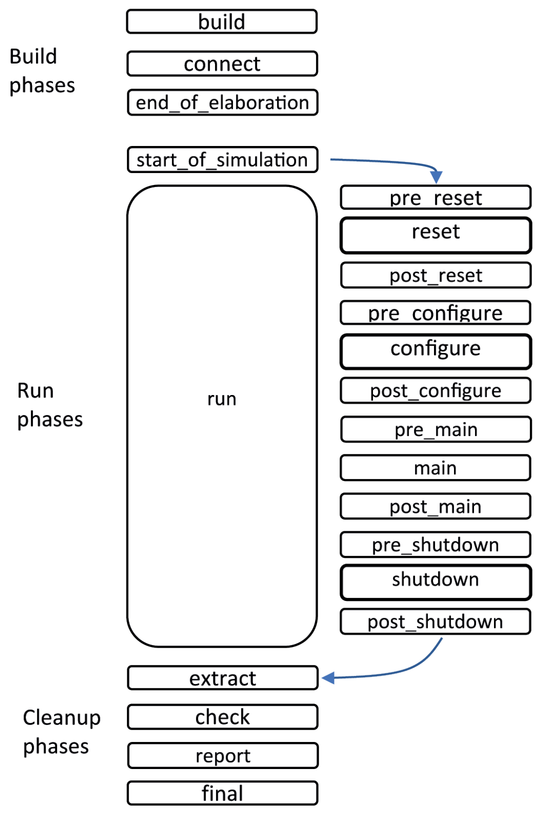


Figure 14: UVM phases

3 Implementation

The implementation chapter delves into the processes involved towards successful implementation, developing software- and hardware simulations and deliveries.

3.1 Introduction to implementation

Few if any hardware implementation of the uABS variant of ANS have yet been publicly made available, therefore a significant amount of development time of this project has been invested in experimentation and breaking new ground. This process has been carried out using already proven design concepts such as Finite state machines and combinatorial design practices. Throughout the incremental design process a set of metrics and criterias were employed to distinguish success from failure. The checklist below was developed to aid the designer on whether to revert to an earlier build of the design and rethink the introduced changes, or use as is.

- Is the design able to perform lossless compression without error rates?

  Is the Decoder able to reproduce the stimuli driven to the encoder?

- Does a new build of the compressor produce similar or better compression rates than former builds?

- Is the design faster, cheaper or smaller (e.g. Power, Performance and Area) than former builds?

- Does the new build implicate potential bugs or challenges which may prove problematic in the future?

In total, two software models and two hardware implementations are proposed. These instances serve an important role in the exploration and analysis of ANS compression. A strong argument for discovering the feasibility of hardware implementation through an incremental design process, with multiple software models and hardware implementations, is that it adds a wast amount of experience and data for future analysis. The experience can lead to optimizations and better understanding of ANS, as the systems grow increasingly complex with increasing abstraction levels. It is for example easier to understand a C++ software model, and make subsequently add algorithmic improvements to it, whereas a hardware implementation in SystemVerilog may grow increasingly difficult to improve algorithmically as the designer works at the RTL level. Conversely the RTL designer may also find potential improvements on the RTL level, that may be tested for validity in the software model. If the optimization or improvement works in both the software model and the hardware implementation, then the RTL designer may proceed with confidence. This method became crucial in the implementation of the ANS compressor, and was a key driving factor.

The following list illustrates a simplified work breakdown structure of the development process.

1. ANS C++ software model
   Verification
2. ANS SystemVerilog software model
   Verification
3. Unoptimized ANS hardware implementation
   Verification
4. Optimized ANS hardware implementation
   Verification

While verification is listed as a sub-task, it is recommended to create a proper test environment (e.g. UVM or a simple testbench) early in the development process. Testing is crucial during the development of compressors as small changes might lead to faulty compression behavior. These could be corner cases that are hard to track, and later down the development process might prove difficult to bug hunt as the systems grow increasingly complex. Therefore it is highly recommended to eliminate bugs and unintended behavior as soon as possible after their introduction.

Both the software models and hardware implementations stem from the uABS variant of the ANS algorithm, and will therefore inherently have many similarities as illustrated in the list below, and with requirements as shown:

- The design instance contains one Encoder and one Decoder. The Decoder should for any legally encoded state be able to decode the state and retrieve the original representation. Likewise the Encoder must be able to unambiguously encode a given legal stimuli, meaning if given an input the decoder must always produce the same output for a given probability.
- For every single encoding and decoding operation one unambiguous probability must be made available to the encoder and decoder.
- In these designs accurate probability modelling will be used, therefore probability distribution must be as accurate as possible in order to produce high compression ratios. The end result is dependent on as accurate as possible probability modelling. The desired solution was to tally the total number of 1's in the data set and divide it by the total number of bits in the stimuli.
- The probabilities are per definition in the range of greater than 0% and within 100%.
- Implementation must have high enough decimal precision to handle probabilities with sufficient precisions. For software models, this is handled by using floats (or preferred double) datatypes. For hardware implementation the desired solution was using fixed precision (e.g.

Q(48.8) format), parameterization is highly recommended as it offers re-usability and configurability. Lastly, it is important that the encoder and decoder use the same precision and handle fractions similarly during the majority of processing, it is highly recommended to use the same precision strategy for the encoder and decoder.

- The encoder and decoder pair must be able to operate separately and concurrently. For instance the decoder may decode one message, while the encoder may encode another distinct message simultaneously. This does however not mean that they must decode and encode the same message at the same time, which is neither useful nor currently feasible.

3.2 Systems overview

As briefly outlined in section 3.1, the standard ANS compressor is required to have at least one Encoder and Decoder module. This basic structure may be illustrated as in illustration 15. Both the software models and hardware implementations follows this structure, and are named according further visualized in illustration 16. The overview is in line with the work breakdown structure presented in section 3.1.

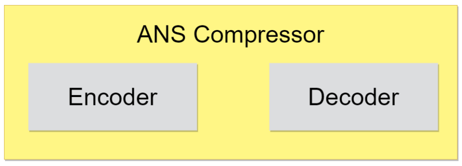


Figure 15: The standard ANS compressor unit, it features at least one Encoder and Decoder entity.

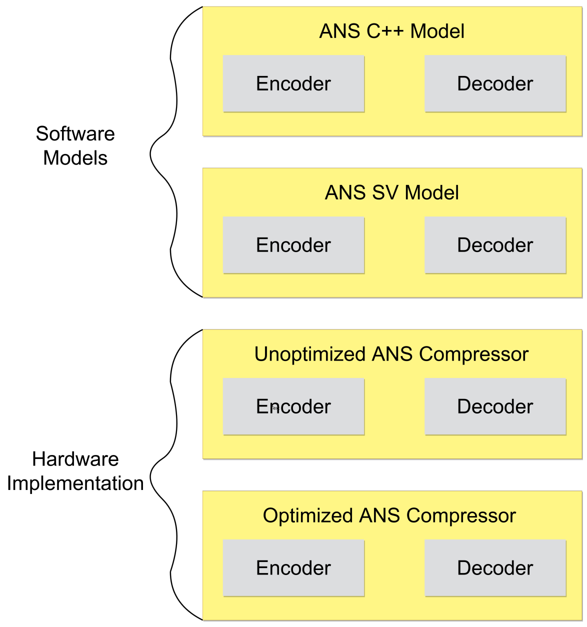


Figure 16: An overview of two software models and two hardware implementations. They are all uABS variants of the ANS algorithms and are the four compressors that were created.

The ANS c++ Model and ANS SV Model were named after the language they are implemented with, respectively C++ and SV (SystemVerilog). The Unoptimized- and Optimized ANS Compressors are implemented in Hardware using the hardware description language of SystemVerilog. This differentiates them from the SV Model as it is designed as a higher abstraction level simulation oriented model. It therefore serves the purpose of bridging the gap between the C++ software model and the Hardware implementations, and is only purposed for verification, not for synthesis. A more detailed overview of the software models is seen at illustration 16, and will be further detailed in section 3.3. Both software models has at least one testbench each, which drives the design with stimuli and examines the outputs, by methods such as equality (e.g. does the encoding/decoding produce a result which equates the expected result). Lastly the Encoder also produces two output files with the encoded and decoded representations.

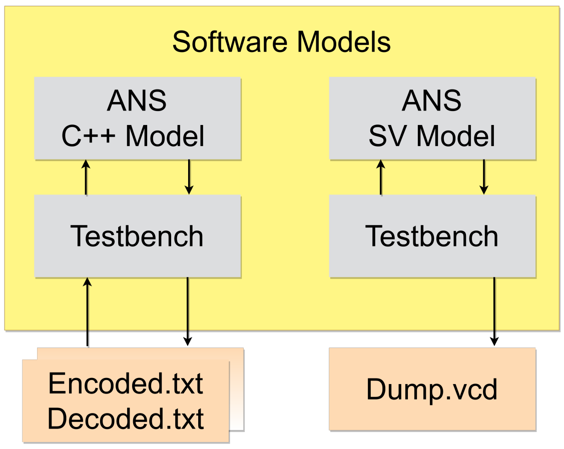


Figure 17: An overview of the software models and their interfacing. The ANS models interface with a verification environment. External files are deliberately created for the C++ Software Model which takes the form of Encoded.txt and Decoded.txt, these holds the encoded and decoded states as well as decoded bits

The hardware implementations are structured in largely the same manner as the software models. They are interface with a testbench and exports simulation data through external files. They differ in the sense that the testbenches for the hardware implementations also offers newer metrics such as the actual data and clock signals, and their corresponding waveforms. These waveforms may for instance be analyzed in timing analysis, as presented in section 4.3.1.2. Furthermore, many properties of the simulation are exported to .csv files. These makes external statistical analysis (e.g. Jupyter Notebook) easier to interface with. It is recommended to continuously analyze the performance of the compressor as development progresses. The illustration 18 displays these properties.

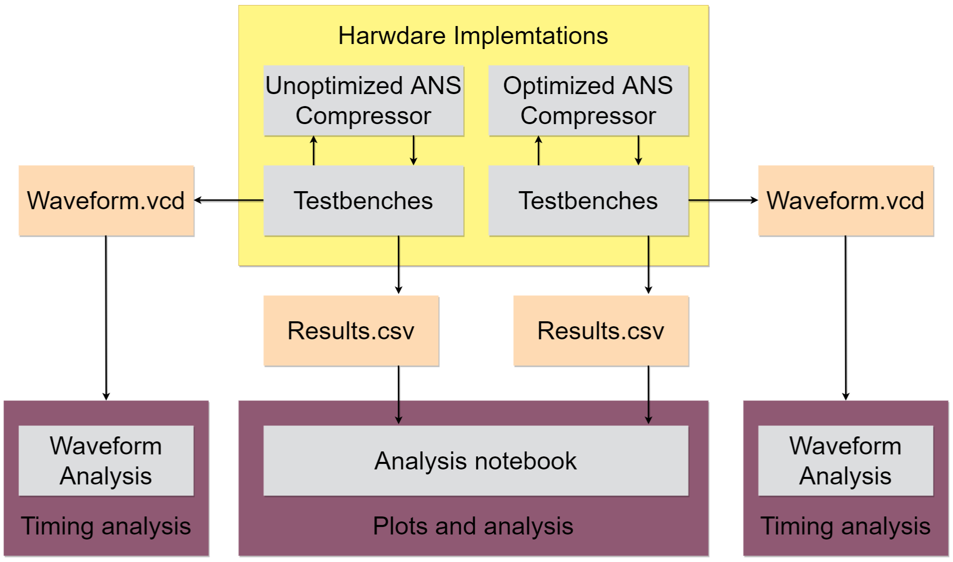
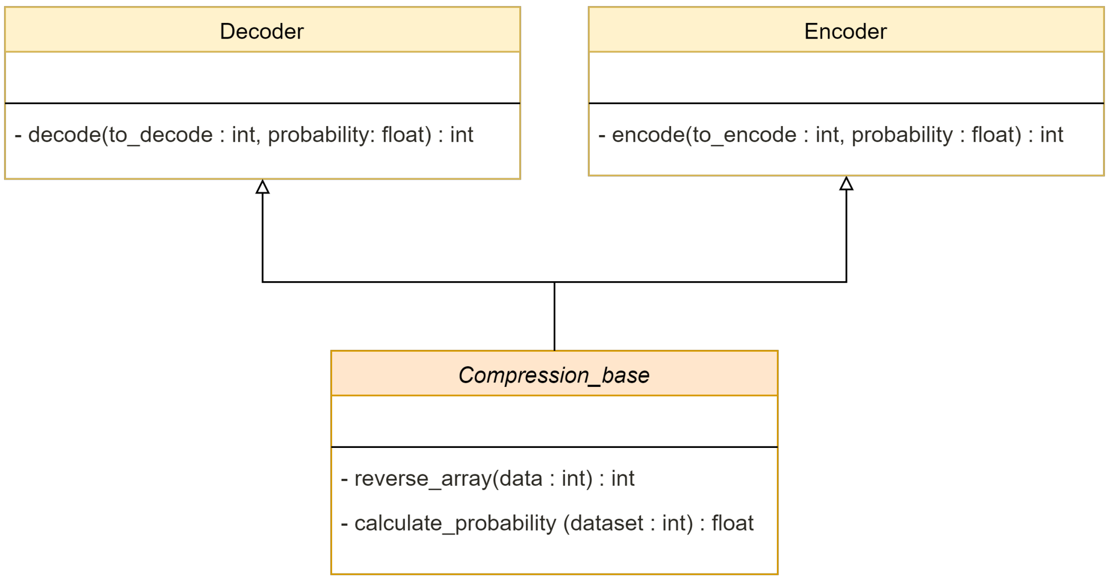


Figure 18: An overview of the hardware compressor implementations. The ANS compressors interfaces with a verification environment. External files are deliberately created for waveform and result exports. These are exported in order to interface directly with timing analysis and analysis entities.

3.3 Software models

The software models has been created based on the strategies and methodology presented in 3.1 and 3.2. Furthermore, the theoretical background presented in section 2.2 explains in detail the uABS variant of the ANS algorithm that is being modelled.

Both the ANS C++ Software model and the ANS SystemVerilog Software models contains much of the same classes, methods and overall functionality. A summarizing class diagram 19 presents essential classes common within both the models.



Decoder
- decode(to_decode : int, probability: float) : int
Encoder
- encode(to_encode : int, probability : float) : int
Compression_base
- reverse_array(data : int) : int
- calculate_probability (dataset : int) : float

Figure 19: Simplified class diagram of the essential classes inside a ANS software model.

There are no necessary attributes for the classes as probabilities and states are calculated for every compression cycle. For the C++ implementation however some attributes are declared. Furthermore there are only a handful of essential methods. The encode and decode methods performs their respective compression operations, and the compression base class contains two helper methods. The reverse_array method is needed to reverse the output of the decoder. This encoder and decoder iterates from MSB to LSB, producing the decoded bits from LSB to MSB. For correct endianness this order must be reversed, this could also be directly implemented into the Decoder. This revere bit order is due to the for loop structure and cannot be easily circumvented. Lastly the probability is calculated by the calculate_probability method.

3.3.1 ANS C++ software Model

The software model takes the basic class diagram 19 and adds some new classes needed for testing and exportation of data. Otherwise, it remains faithful to the basic overview. The software models has the purpose of testing the algorithm and specific aspects of its algorithmic properties. Therefore it adds a Simulator class as well as the C++ specific main.cpp file to execute the tests.

3.3.1.1 Creating the ANS C++ software Model

The Encode and Decoder classes are summarized in C++ pseudo code 1 and 2 respectively. The software implementations are identical to those presented in theory, and are relatively trivial arithmetic operations. The Encoder uses division, addition and subtraction operations whereas the Decoder uses multiplication as opposed to division. Following the uABS variants algorithmic steps in section 2.2.2 these may be directly coded into the model, and should not lead to any challenges. Lastly the correct rounding is handled by the floor and ceiling functions, and the code is wrapped inside the conditional for statement. The length of the for loop is determined by the number of bits to be compressed.

```
Encoder::encode_uabs(binary_sequence) {
  probability_of_one = get_probability_of_one();
  state = get_starting_number();
  string_length = length(binary_sequence);

  for (ii = string_length - 1; ii >= 0; --ii) {
    if (binary_sequence[ii]) {
      state = floor(state / probability_of_one);
    }
    else {
      state = ceil((state + 1) / (1 - probability_of_one)) - 1;
    }
  }
  return state;
}
```

Listing 1: uABS encoder c++ style pseudo code

```
Decoder::decode_uabs(state, probability_of_one) {
  string_length = length(state);
  decoded_binary_sequence(0);;

  for (ii = string_length - 1; ii >= 0; --ii) {
    decoded_binary_sequence[ii] = ceil(state * probability_of_one) - ceil(((double)
    state + 1) * probability_of_one);
    if (decoded_binary_sequence[ii]) {
      state = ceil(state * probability_of_one);
    }

    else {
      state = (state - ceil(state * probability_of_one));
    }
  }
  return binary_reverse(decoded_binary_sequence);
}
```

Listing 2: uABS decoder c++ style pseudo code

The encoder produces a state, whereas the decoder produces a state and an encoded bit, the state is returned by reference. The state however, is not returned from the Decoder as the compressor configured to loop over the entire BIT_WIDTH. The encoder will always start at the initial state which is commonly set to ’1’ 2.2. The decoder’s State should similarly be found to be ’1’ as the decoding concludes. The usage of initial state may be seen as a checksum, as both the encoder starts at the check sum and and decoder ends up at the check sum. If the decoder does not have the correct state value at the end of decoding, then the compression has been nugatory and is considered wrong.

The base class inherits from both classes inherits for binary_reverse method. The pseudo code for the base class is not shown in it’s entirety. Only the binary_reverse method is shown as it serves and important illustrational purpose.The pseudo in listing code3 shows how the binary vector is reversed iteratively throughout the for loop. It is then returned with the option of offsetting the end result, which is needed if there are leading or trailing 0’s in the data holding the result. For instance if the stimuli is 110011 and it is held in an array with greater width (i.e. not a dynamically allocated array), then leading 0’s are introduced during the compression process. If these 0’s are to be filtered out, then the offset should be adjusted accordingly. If the leading 0’s are to be maintained then the offset should be set to 0.

```
Compression_base::binary_reverse(binary_number, offset) {
  for (ii = 0; ii < (BIT_WIDTH / 2); ++ii) {
    temp = binary_number[ii];
    binary_number[ii] = binary_number[BIT_WIDTH - ii - 1];
    binary_number[BIT_WIDTH - ii - 1] = temp;
  }
  return (binary_number >> offset);
}
```

Listing 3: uABS base class inherited by encoder and decoder

The rest of the base class contains methods related to data management, analysis and interfacing with files/printing to monitor and are merely listed. They may be created in a multitude of ways based on personal preference, and are relatively trivial in nature.

- `print_compression_data(data, line_terminator, end_line, file_name)` - Print compression results to an external file. This is useful in order to analyze large datasets at a later stage. Here the raw data from the encoder/decoder may be adjusted to add formatting, such as formatting the data to a csv file. The `line_terminator` and `end_line` arguments should thus be chosen based on the target file format.
- `clear_file(filename)` - A method to clear external output or input files.

The modeller may make a set of decisions that will affect how the algorithm performs. The correctness of the implementation may be determined by comparing the stimuli provided to the encoder to the reproduced representation by the decoder. It is recommended that the designer performs these comparisons frequently during the design process.

The encoder and decoder are reliant on using the same probability. If the probability of '1's or '0's is not the same for both encoder and decoder, then the decoder will not be able to reproduce the original stimuli correctly. Furthermore, how accurate the probability is modelled will affect how well the encoder compresses the data. The compression ratio will deteriorate as the probability becomes less accurate. Therefore the designer should strive to keep the probability as precise as possible if strong compression ratios are prioritized.

The algorithm will only be able to compress consistently when the initial state is known by the encoder and decoder. The lowest possible recommended starting state is '1', however could in theory be any positive integer. Alternatively, this may requirement may be bypassed by requiring the user of the encoder to always apply stimuli (e.g. a dataset) which includes a leading one (high bit). In this case, then the encoders initial state may also be '0'. The advantage of the starting state being '0' is that the overall state becomes less in value, and may lead to greater compression ratios. If this scheme is selected, and the encoder uses a leading '1', then this information must be made available to the decoder. In this case the Decoder should end up on the state of '0' during its last decoding cycle. If this rule is not maintained, the decoder will lose leading '0's prior to the first '1' in the decoding process. This issue is not present in the hardware implementation as presented in section 3.4.

When introducing a for loop in the encoder and decoder structure, the code repeats the encode/decode function a set number of times. This setup is not strictly needed. However, it introduces some convenience when observing its performance in software. An alternative arrangement is having the encode and decode functions perform on a single bit at a time. Both configurations will provide the same results and are only different in execution. The hardware implementation presented in section 3.4 performs encode and decode operations on a per-bit basis, as opposed to the software models which operates on binary vectors. The vector reversal method is not required for the hardware implementation, as the decoded bit is merely one binary value.

The algorithm uses floating number arithmetic, which introduces issues related to rounding. As defined for the uABS variant of ANS 2.2, most of these issues are handled using ceil() and floor() at the specific algorithm steps. These functions provides predictability and accuracy. Failure to solve these rounding issues would induce wrongly encoded and or decoded representations at a high rate. Alternative rounding schemes are possible, however not explored in this thesis.

The software implementation may need to adjust for an offset on the decoded representation of the stimuli. This depends on how the algorithm has been implemented. Suppose the implementation has a static bit width on the decoded representation's data element. The result may need to be padded with an offset since the bits are calculated from LSB to MSB before the vector-reverse function. Consider the case where the encoder encodes 8 bits while the data element holds 12 bits. Then, the decoder would have to pad the decoded representation by 4 bits to make the decoded representation and original representation equivalent. The designer may ignore the offset issue if the implementation does not contain the for-loop structure, as only one bit is produced at once and naturally would not need to be reversed. This is similarly handled in pseudo code 3.

3.3.1.2 ANS C++ software model testbench

The testbench should be set up early during the development process to catch errors in compression behaviour, it is here known as the Simulator class. There are many useful metrics that should be taken into account when performing testing, such as compression ratio, probability and rate of faulty compression results (e.g. if decoded representation deviates from stimuli given to encoder). Helper methods may be employed to extract these metrics from the compressor. Some of these helper functions are already written as methods in the compression base class, as seen in pseudo code 3. After driving stimuli to the design and collecting and analyzing data, the design should write the test results to the terminal and external files, for instance with a .tx. or .CSV format.

The algorithm may be tested with pseudo-random or non-random stimuli. Both have their useful use-cases. Beyond stimuli selection, there are some useful methods which should be defined in the testbench class.

- `select_line_in_file(filename, line_number)` - This method is used to select lines in a targeted file, it may be seen as a helper method to be used in conjunction with other methods.
- `extract_data_from_file(filename)` - A method used to extract data from a file. This method uses the `select_line_in_file()` method. It returns the data in the desired formats (as designated by the designer). In this application relevant formats are vectors for extraction of states, strings or a binary data types for extraction of decoded representations (i.e. the results after decoding).
- `batch_run_and_print_to_file()` - This method encompasses all the aforementioned methods and executes both the testing as well as the compression itself. It must thus have access to properties such as starting state, stimuli, encoders results, decoders results and probability.

The last step of the design process is to set up a main file. This file should instantiate the simulator class and execute the `batch_run_and_print_to_file()` method. This main file may be short and easy to use. See pseudocode 4.

```cpp
int main() {
    start = start_state;
  Simulator simulator(start, "../../stimuli.txt", "../../encode.txt", "../..decode.
    txt");
  simulator.batch_run_and_print_to_file();
}
```

Listing 4: main file for the uABS software model

The main file concludes the software model. It is recommended to use this software model as a reference when working through the following sections, as the software model is made at a higher abstraction level. This aids the designer in keeping track during low level implementation, as well as a means of comparison. The designer may for instance apply the same stimuli to the software model and the hardware implementation, and compare the results from both encoder and decoder. If both the software model and the hardware implementation produces the same results, then the confidence in the work should rise. Lastly it may also be recommended to compare the outputs of the encoder and decoder to the calculated examples in the theory section 2. This may act as a third reference.

3.3.2 ANS SystemVerilog software Model

The SystemVerilog model presented in this thesis is essentially a simplified ported implementation of the C++ model, as the SystemVerilog language reaps advantage of features that are not present in the C++ language. The sole purpose of the SystemVerilog software model is thus to bridge the gap between the software C++ model and a hardware implementation created with SystemVerilog, as described in section 3.4. It is enough to prove that a given stimuli encoded in the C++ model is correctly decoded in the SV model and conversely.

3.3.2.1 Creating the ANS SystemVerilog software model

Taking inspiration from pseudocodes of the Encoder 1, Decoder 2 and Base class 3, the designer may use the equivalent strategies in SystemVerilog. This design step does not require many changes to the C++ code, as the sole purpose of the simulation only model is to adapt the software implementation from a high-level language (i.e. C++) to a low-level language (i.e. SystemVerilog). In essence, this model is merely used to quickly verify that the algorithm behaves similarly after changing the programming language.

On this stage, the designer will mostly make the design synchronous and add clock edge-sensitive blocks. The designer may quickly achieve sufficient granularity of synchronous logic by using enable flags in unison with the clock-edge triggered logic. Given the pseudocode provided in 5, a positive clock edge-triggered logic is combined with an enable signal. This process ensures that the encoder and decoder are only engaged when chosen, and only at the clock edge for easier compatibility with an interface, such as when interfacing with a testbench.

```
        module uabs_compressor (...)

        // At clock edge and decoder enable
          decode()

        // At clock edge and encoder enable
          calculate_probability()
          encode()

        endmodule : uabs_compressor
```

Listing 5: SystemVerilog pseudocode with the uabs compressor module. This module is intended for simulation purposes only

It is worth noticing that the probability in pseudocode 5 is calculated during encoding, and must be stored and subsequently be used in the decoder for properly decoding of the data.

3.3.2.2 ANS SV model testbench

Lastly a testbench must be created in order to drive the design under test (DUT) with stimuli, print and analyse the results as well as to create a dumpfile for waveform creation. This may be achieved by using a standard testbench. In comparison, the Hardware Implementations is verified with a UVM and a more complex testbench, as seen in section 3.4.3. Pseudo-code 6 shows how a simple test may be carried out.

```
module uabs_decoder_tb
    generate_clock()

    // Begin
    create_dumpfile()
    reset_signals()

    // Start compression
    drive_stimuli()
    enable_encoding()
    // wait one cycle
    enable_decoding()
    // wait one cycle

    // Check results
    check_compression;

endmodule : uabs_decoder_tb
```

Listing 6: SystemVerilog pseudocode with the uABS compressor testbench. It drives the DUT with stimuli and performs one encoding and decoding operation.

The designer should print the results to monitor and manually inspect the output waveforms, check for equallity or correctness. The resulting waveforms should match those presented in illustration

64. This bench carries out a single compression cycle, with WIDTH number of bits. The clock uses two cycles from positive edge to transition to the next positive edge, meaning there is one cycle between the negative and positive edge. The encoder is enable for one clock cycle and the decoder is enable at the following cycle, whereas the decoder is driven stimuli by the encoder.

3.4 Hardware implementations

The hardware implementation is at a lower level of abstraction, and in some areas substantially more complex than the software models. Nevertheless they still inherit the properties of ANS similarly, and the same algorithm requirements still applies. This section will therefore assume that the reader has familiarized itself with the software models prior to reading it. Basic concepts of the uABS variant, and the implications of working with it will therefore not be discussed as it has already been analyzed in the software model section 3.3.

There are many aspects that must be taken into account when assessing hardware implementation of an algorithm. Examples are algorithmic complexity, synthesis implications of code and PPA (power, performance and area). A convenient way of managing these challenges is to continuously synthesize the design during development. The algorithm is quite light, and does not require a lot of time to synthesize in popular EDAs. The designs was developed, synthesized and implemented using Vivado (2022.1), however was also tested verified with Cadence Xcelium (20.09) and Aldec Rivera Pro (2020.04). No vendor specific design features or IP blocks were used. This means that in theory the design should be compatible with most EDAs and remain highly re-usable and modular. The key takeaway is that synthesis and implementation results largely dictated design decisions, and the implementation strategies that were used. The principles of section 3.2 still applies.

Two iterations of the ANS compressor were created, one with hardware specific optimizations and one with a more straight forward implementation. The circuitries are modularized and based on a synchronous FSM controlling the flow in an arithmetic module with the compression logic. Both modules are instantiated within a top module which handles the I/O and facilitates the interfacing between the arithmetic and the FSM.

Both the encoder and decoder are designed to be convenient for integration into larger systems. This is accentuated by the largely parameterized modules, enumerated states and a Ready/Valid interface, as further evaluated in 5.8. The Ready/Valid strategy remains prevalent throughout the entire design process, and is described in section 2.3.5. The Ready/Valid interface is chosen as it is convenient to implement, interface with and thus less time consuming to verify. It is not only used at the I/O, but also employed internally within the circuitry. The Encoder and Decoder modules are, in their final form, implemented separately into their own top modules. This is highly advised, as they may be used, implemented and analyzed separately. The designer should thus end up with two top modules, one for the encoder and decoder respectively. For verification however, in some configurations the encoder and decoder are instantiated in one top module for convenience.

An open source fixed point division module was used with light modifications, the original code may be found at [32]. The module is re-usable for developers with the MIT licence and was created by Will Green. This module is passable and functional, however not highly performant. It became the main bottleneck of the Encoder circuitry, as further displayed in section 4.3.5 and discussed in section 5.3. The module requires n clock cycles to perform fixed point division due to it's synchronous operation, which contrasts with what is otherwise lightweight synchronous or (largely asynchronous) combinatorial logic.

3.4.1 Unoptimized ANS compressor

3.4.1.1 Overview

The unoptimized ANS Compressor was the original build of the ANS hardware implementation. It is a more straightforward implementation than its newer counterpart, the optimized ANS compressor. The design presented is instantiated in a single top module, encompassing both the encoder and decoder. This is convenient for testing and is not an issue as the unoptimized ANS compressor was not intended for rollout.

The modules are instantiated within the design.sv module, as seen in the list below. It shows a hierarchical overview. In addition there are two packages uabs_encoder_fsm_pkg.sv and uabs_decoder_fsm_pkg.sv. These packages contains enumeration literals, which makes the FSM cleaner to design and easier to understand.

- design.sv
  - uabs_decoder.sv
  - uabs_decoder_fsm.sv
  - uabs_encoder.sv
    - div.sv
  - uabs_encoder_fsm.sv

3.4.1.2 Interface

The interface of the unoptimized ANS Compressor is displayed in table 4. These signals are the I/O of the top module design.sv. These ports are a combination of flags and signals intended for carrying data. The resulting interface is presented in section as shown in the RTL analysis.

Table 4: Unoptimized encoder and decoder interface using a Ready/Valid strategy

Port	Direction	Type	Description
start_dec, start_enc	Input	Logic	Start encoding flag
vin_dec, vin_enc	Input	Logic	Valid input flag
rout_dec, rout_enc	Input	Logic	Ready for output flag
clk	Input	Logic	Clock
rst_n_dec, rst_n_enc	Input	Logic	Negative reset
to_decode	Input	Logic [WIDTH-1:0]	A state to decode
to_encode	Input	Logic	A bit to encode
probability	Input	Logic [WIDTH-1:0]	Probability of ones
decoder_en, encoder_en	Input	Logic	Enable flag
state_in_enc	Input	Logic [WIDTH-1:0]	ANS state input
state_out_dec	Output	Logic [WIDTH-1:0]	ANS state output
idle_dec, idle_enc	Output	Logic	Idle flag
ready_dec, ready_enc	Output	Logic	Ready for compression flag
vout_dec, vout_enc	Output	Logic	Valid out flag
decoded_bit	Output	Logic	A decoded bit (result of decoding)
encoded	Output	Logic [WIDTH-1:0]	Encoded result (result of encoding)
div_dbz	Output	Logic	Division module's divide by zero flag
div_ovf	Output	Logic	Division module's overflow flag

There is also an internal interface between the FSM and the arithmetic logic, as seen in table 5. These signals are internal to the top module design.sv and are exclusively flags.

3.4.1.3 Finite State Machine

The Finite State Machine (FSM) is the control logic of the implementation (e.g. uabs_decoder_fsm.sv, and directs the arithmetic modules flow (e.g. uabs_decoder.sv). The state machines for the Encoder and Decoder are largely similar, the major difference being declaration names. This is not the case for the Optimized counterpart, as presented in section 3.4.2. The Encoder FSM diagram is presented in illustration 30

Table 5: Unoptimized encoder and decoder, internal interface between the FSM and the Arithmetic modules, Using a Ready/Valid strategy

Port	Direction	Type	Description
check_dec, check_enc	internal	Logic	Check arithmetic unit flag
checked_dec, checked_enc	internal	Logic	Check has been performed flag
bit_is_one_dec, bit_is_one_enc	internal	Logic	Bit is one flag
comp_one_done_dec, comp_one_done_enc	internal	Logic	Compression of type one done flag
comp_zero_done_dec,comp_zero_done_enc	internal	Logic	Compression of type zero done flag
decode	internal	Logic	(start) Decode arithmetic flag
encode	internal	Logic	(start) Encode arithmetic flag
comp_ready_dec, comp_ready_enc	internal	Logic	Arithmetic unit ready flag
comp_busy_dec, comp_busy_enc	internal	Logic	Arithmetic unit busy flag
comp_init_dec, comp_init_enc	internal	Logic	Initialize arithmetic unit flag

The state machine has several states and is of the Mealy type, as the next state is affected by its current state and it's inputs. The state machine contains eight states. Some states are quite similar, such as the COMP_ONE_INIT and COMP_ZERO_INIT states. The main difference being their state transition, which are prompted based on whether the encoder is encoding '1' or '0'. Likewise, the $COMP\_ONE\_BUSY$ and $COMP\_ZERO\_BUSY$ states are almost identical. They are seen merged into one state in the optimized implementation, as presented in section 3.4.2.5.

The initial state is the IDLE state, which the state machine will remain in until the start signal has been asserted. This will prompt the $READ$ state which is used to begin the interfacing process for reading stimuli, and asserts the ready in flag as. Upon receiving a valid in signal the CHECK_ENC state commences. This state will remain until the arithmetic module has both analysed it's inputs (i.e. checked) and determined whether the incoming bit is zero or one, prior to transitioning the state flags the encode, and comp_init flags. The next state is the initial compression state known as COMP_ONE_INIT or alternatively COMP_ZERO_INIT if bit_is_one was low. This state triggers the comp_busy flag and transitions to the DONE_ENC state upon receiving the comp_ready flag. Upon receiving the comp_one_done the FSM transitions through COMP_ONE_BUSY and DONE_ENC while flagging their respective flags and returning to the IDLE state and remains there until the next compression cycle commences. The requirements for interfacing to and from the FSM is summarized in sequence diagram 21, which the FSM is responsible for.

The decoder FSM has the same state transitions and conditions as the encoder FSM 20, and is therefore not drawn up separately. The effeciency of the FSMs is evaluated in 5.6. The FSM RTL was modelled based on the FSM state Diagram.

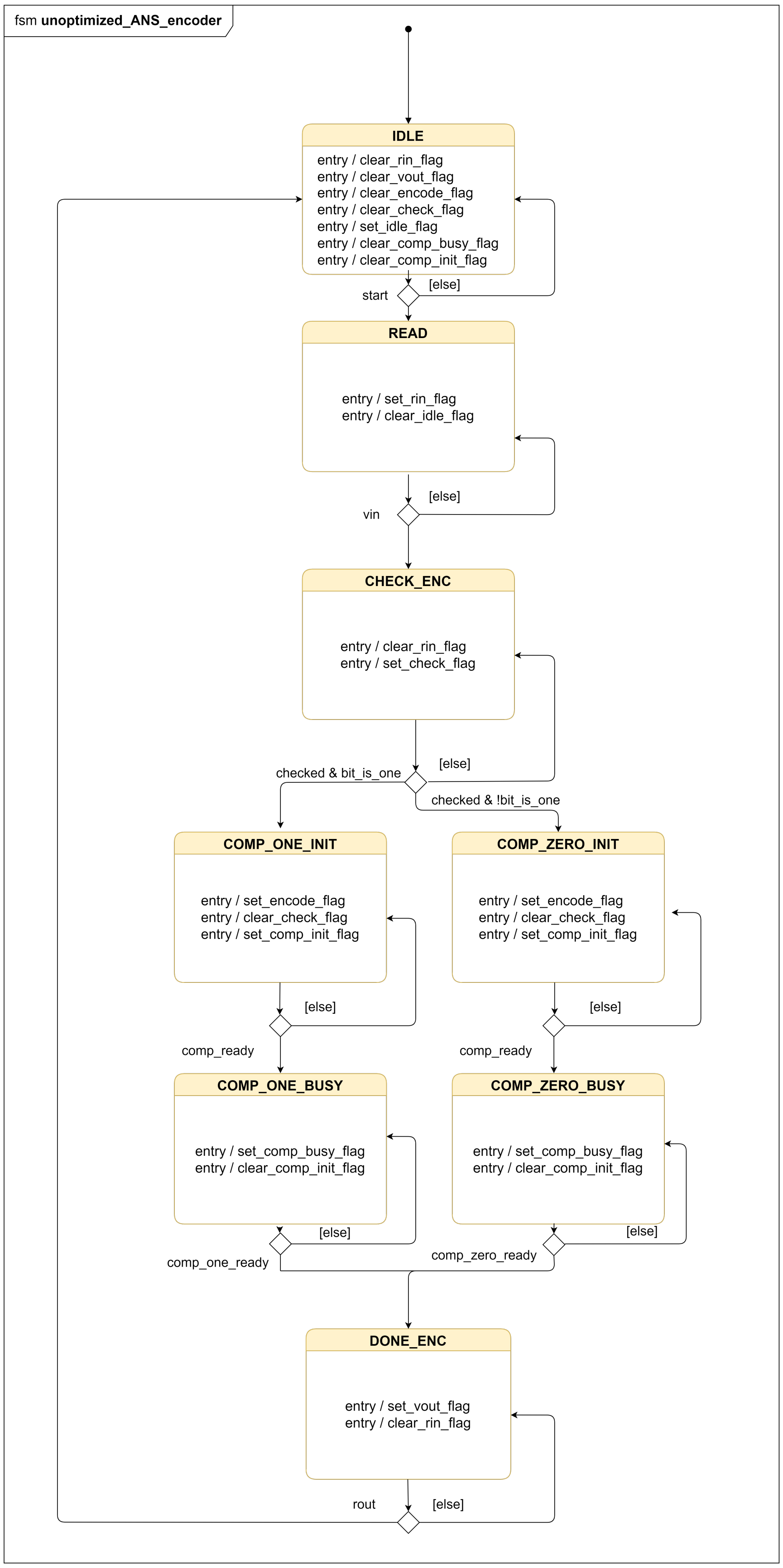


Figure 20: Unoptimized ANS Encoder State Machine Diagram. Designed UML style

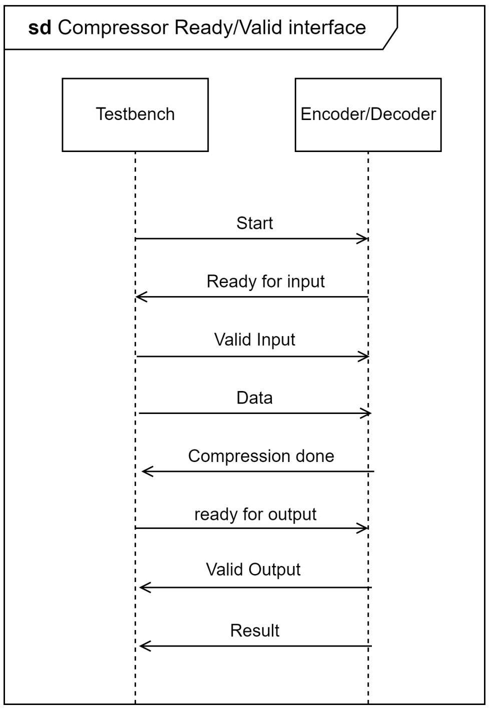


Figure 21: Illustration of the Ready/Valid requirements for the FSM. It visualizes how the testbench interfaces with the compressor, which is realized by the FSM

The hardware implementation follows the STM diagram 20 faithfully and is kept compact, as less code often leads to fewer bugs. Designs that are compact are furthermore easier for the designer to visualize, which is very important when modelling finite state machines. The logic datatype is used deliberately throughout the RTL as it does not allow the designer to mistakenly make multiple assignments to the same signal. The compiler will not allow this occurrence and throws an error or warning during simulation and synthesis, and thus saves time in bug fixing.

Combinations of flip-flop and arithmetic based logic is exclusively used in the FSM RTL to infer clocked and combinatorial logic. The SystemVerilog always_ff and always_comb procedures includes built-in analysis during synthesis. It will for instance display a warning should combinatorial logic not be inferrable. Secondly, for the circuit to be truly combinatorial all branches of a MUX must contain assigned outputs for the respective logic signals (e.g. flags in the FSM). The mux is represented by the CASE statement in the SystemVerilog code. This design strategy is reflected in the code, see appendix section G, and as is seen in section 4.3.6 during RTL analysis. As indicated, every possible branch that might be taken in the case statement drives a value to all flags, so that all possible paths are covered. If this was not true, then for instance latches might have been inferred and could have lead to unintended behavior. The entire list of flags are set to low at the top of the sequential always_comb statement, as this ensures high readability of the code and less chance of humanly induced errors, as opposed to placing all of them inside every States directly.

Furthermore, the state register is inferred from three short lines of code. The state transition is positive edge triggered with a negative edge triggered active low asynchronous reset. It is placed at the top of the design to make it clear to the reader how the FSM behaves prior to reading all the states. If the design was aimed at ASIC, then the flip-flops might be unwanted, as flip-flop adds to the blocks of clocked logic and does consume space. On the other hand for FPGA's flip-flops are cheap, whereas combinatorial logic is more costly. Nonetheless as the design is relatively lightweight this is not any issue for this FPGA implementation, observing that the majority of logic is combinatorial. Lastly OneHot encoding is chosen for the states as it requires smaller combinatorial logic (at the cost of more Flip Flops), which is also preferred by most synthesis tools (such as in Vivado).

Table 6: Unoptimized ANS Compressor primary clock constraints

Primary Clock	Preiod (ns)	Frequency [MHz]
clk	10.0	100.00

Table 7: Unoptimized ANS Compressor Input delays

Interface	Clock	Alignement	Data Rate and Edge	Min Delay [ns]	Max Delay[ns]
start_dec, start_enc	clk	Edge	Single Rise	2.000	3.000
vin_dec, vin_enc	clk	Edge	Single Rise	2.000	3.000
rout_dec, rout_enc	clk	Edge	Single Rise	2.000	3.000
rst_n_dec, rst_n_enc	clk	Edge	Single Rise	2.000	3.000
to_decode	clk	Edge	Single Rise	2.000	3.000
to_encode	clk	Edge	Single Rise	2.000	3.000
probability	clk	Edge	Single Rise	2.000	3.000
deocder_en_ encoder_en	clk	Edge	Single Rise	2.000	3.000

3.4.1.4 Arithmetic

The arithmetic unit follows much of the same design principles (for good RTL) as the FSM, as explained thoroughly in section 3.4.1.3. The task of the arithmetic modules is to perform the actual encoding and decoding. They are controlled by the FSM flags and the most critical design decisions were to ensure correct flow. Therefore many internal flags have been employed inside always positive edge triggered blocks, resulting in sequential synchronous logic. The setup used in the arithmetic modules of the unoptimized implementation was aimed to be accurate, bug free and easy to understand and verify. A detailed rundown of the module will therefore not be showcased, please refer to section 3.4.2.6 which contains a more deliberate and meticulous solution.

3.4.1.5 Preparations for Synthesis and Implementations

As the unoptimized implementation of the ANS compressor is largely a proof of concept, and not a product ready for roll out little to no preparations was necessary for synthesis and implementation. The only criteria for the circuitry was to be synthesizable without any significant warnings. Timing constraints was created based on the recommendations displayed in the EDA Vivado and is summarized in tables 6, 7 and 8. The constraints are neither strict nor relaxed, and are mostly aimed at giving an indication on how well the design performs. Most important is the clock constraint as it clearly indicates how light the sequential logic is. The highest achievable clock frequency also decreases as the input and output delays grows, which is why they are kept at an average level. Lastly these delays are also affected by the technology employed, better or worse FPGAs and EDAs might need required tighter or more relaxed constraints for realistic reports.

Table 8: Unoptimized ANS Compressor Output delays

Interface	Clock	Alignement	Data Rate and Edge	Delay min [ns]	Delay max [ns]
state_out_dec	clk	Edge	Single Rise	0.000	1.100
idle_dec, idle_enc	clk	Edge	Single Rise	0.000	1.100
ready_dec, ready_enc	clk	Edge	Single Rise	0.000	1.100
vout_dec, vout_enc	clk	Edge	Single Rise	0.000	1.100
decoded_bit	clk	Edge	Single Rise	0.000	1.100
encoded	clk	Edge	Single Rise	0.000	1.100
div_dbz	clk	Edge	Single Rise	0.000	1.100
div_ovf	clk	Edge	Single Rise	0.000	1.100

3.4.2 Optimized ANS compressor

The optimized ANS Compressor builds on the foundation of the unoptimized ANS Compressor, as presented in section 3.4. Only the differences between the two ANS hardware implementations will be highlighted here, as much of the design features are common between the two implementations. The main task the optimized ANS compressor seeks to solve is minimizing resource utilization, increasing performance and decreasing power consumption (i.e. achieving a better PPA). The implementation does not change or improve any algorithmic properties and uses synthesis and implementation reports as a measure of achieved improvements. The optimized ANS compressor does not add or remove already existing modules or drastically alter the design philosophy of the unoptimized ANS compressor. The advantage of having the two hardware implementations is found in verification, as the RTL itself has drastically changed especially in the arithmetic modules. Finding optimizations and executing them while also simultaneously implementing them based on a software model is a daunting task, it is therefore of high value to have both the optimized and the original (unoptimized) model available.

3.4.2.1 Overview

The most significant new design features are summarized in the list below. These optimizations were carried out on an individual basis one by one and compared during synthesis. It is difficult to gauge the efficiency of an implemented optimization if multiple changes are carried out at once. Therefore the best way to ensure that every change contributes to a better design is by introducing them one by one, running synthesis and comparing reports before and after. Some optimization techniques may work well in some modules (such as parallelization), while bringing no optimization or even slowing down other modules. This might occur due to the synthesis tool itself performing optimizations that the designer had either not intended or been aware of. Furthermore, it is also important to be aware that the synthesis tool might significantly improve a design despite it employing sub-optimal design practices. This does not indicate that poor design practices are recommended, but that the synthesis tool might solve many of the designers' mistakes. And conversely, improving a design might not yield any real life benefits if the synthesis tool does an excellent job at ironing out poor design decisions. Lastly, it is worth noticing that Synthesis is highly technology specific, and having better tools will in most cases yield better results with little effort on the part of the designer. It is therefore crucial to work with the synthesis tool, and not against it. Make a few changes at a time and analyse how the changes improved or depreciated the performance of the design. At some point, all the low hanging fruits are collected, and only complex and intricate time consuming improvements are left.

- Less I/O and fewer internal signals and registers.
- Fewer states with fewer state transition conditions.
- Shorter blocks of sequential logic, that may execute in parallel as opposed to fewer and longer sequential blocks.
- Only always_ff and always_comb procedures with one exception, as opposed to generic always Verilog procedures.
- Reduction of synchronous logic (e.g. Decoder arithmetic which became asynchronous and combinatorial).

3.4.2.2 Interface

The interfaces follow the same strategies and conventions as presented for the unoptimized implementation in section 3.4.1.2. The main difference is a reduction in unnecessary I/O. With the notable exclusion of the divide by zero and division module overflow flags. They were crucial for development and debugging, however not essential for normal operation. They are still accessible inside the Encoder arithmetic module itself, and may easily be added if the designer desires them.

They were removed to decrease the overall resource utilization and increase timing slack due to lessened complexity. Furthermore, the probability ports were made narrower as they only need to allow inputs of FBITS width, meaning they only accept floating point precision numbers that are understood as less than one (i.e. probabilities of 0-100%). The new interfaces are presented in table 9 and 10. The internal interfaces between the FSMs and the Arithmetic modules are affected similarly, as there are fewer signals.

3.4.2.3 Decoder

Table 9: Optimized Decoder interface using a Ready/Valid strategy

Port	Direction	Type	Description
start	Input	Logic	Start encoding flag
vin	Input	Logic	Valid input flag
rout	Input	Logic	Ready for output flag
clk	Input	Logic	Clock
rst_n	Input	Logic	Negative reset
to_decode	Input	Logic [WIDTH-1:0]	A state to decode
probability	Input	Logic [FBITS:0]	Probability used in decoding
state_out	Output	Logic [WIDTH-1:0]	Decoded state (result of decoding)
decoded_bit	Output	Logic	A decoded bit (result of decoding)
ready	Output	Logic	Ready flag
vout	Output	Logic	Valid output flag

3.4.2.4 Encoder

Table 10: Optimized Encoder interface using a Ready/Valid strategy

Port	Direction	Type	Description
start	Input	Logic	Start encoding flag
vin	Input	Logic	Valid input flag
rout	Input	Logic	Ready for output flag
clk	Input	Logic	Clock
rst_n	Input	Logic	Negative reset
to_encode	Input	Logic	One bit to decode
probability	Input	Logic [FBITS:0]	Probability used in encoding
state_in	Input	Logic [WIDTH-1:0]	A state used in encoding
idle	Output	Logic	Idle flag
ready	Output	Logic	Ready flag
vout	Output	Logic	Valid output flag
encoded	Output	Logic [WIDTH-1:0]	The encoded state (result of encoding)

3.4.2.5 FSM

The FSMs has been significantly improved and trimmed down. They contain less states, has less control logic, and overall reads and drives less flags. The decoder is shown in diagram 22 and diagram 23.

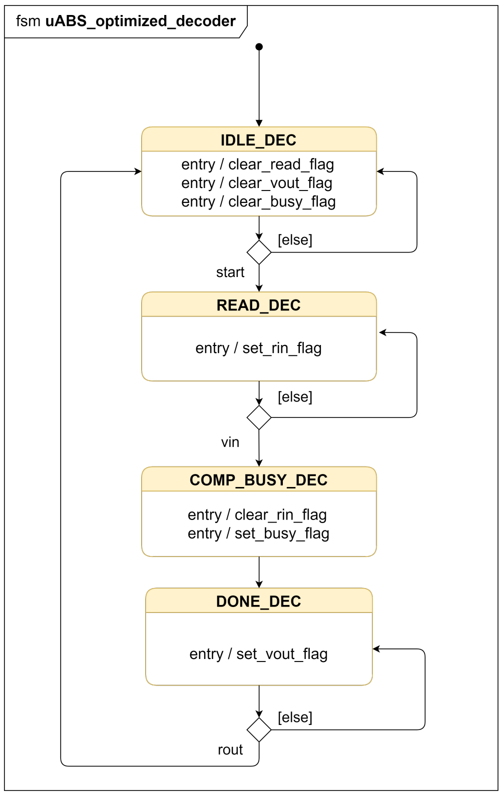


Figure 22: Optimized ANS decoder finite state machine diagram. Designed UML style

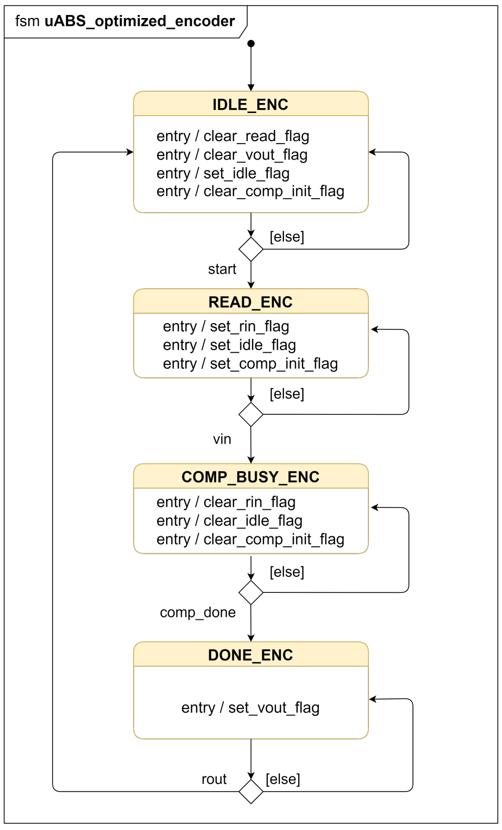


Figure 23: Optimized ANS encoder finite state machine diagram. Designed UML style

3.4.2.6 Arithmetic

The decoder arithmetic module was specifically more heavily optimized as it yielded the highest potential for optimizations. One challenge during optimization was the encoder module's reliance on div.sv (i.e. the fixed division module), which remained a bottleneck in the system. It is further discussed in section 5.3, whereas a proposed solution is presented in section 6. The code of the decoder arithmetic module is seen in appendix at section A.

3.4.2.7 Preparations for synthesis and implementations

The optimized hardware implementation was prepared for synthesis and implementation similarly as for the unoptimized implementation, as presented in section 3.4.1.5. The only difference was different signal names and higher clock frequencies reflected in the constraints. For the full code see for instance listing 16 in the Appendix.

3.4.3 Verification of Hardware Implementations

A multitude of testbenches have been made during development. They are crucial to understanding and correctly shaping the designs, and should ideally be made for every single module. The two most important verification additions were the master testbench and the UVM testbench. These testbenches were used to create large quantities of constrained pseudo random stimuli. Such large amounts of stimuli allows the developer to thoroughly analyze the performance in terms of compressor stability, exactness, speed, power consumption and architectural robustness, and get a better general understanding of the compression behavior and what affects it. Testbenches are crucial in finding bugs, both early and late into the design. Verification may even take more time to properly perform than the design process itself, and was at the core of the development strategy. The configuration of the ANS compressors during verification may be illustrated in illustration 24.

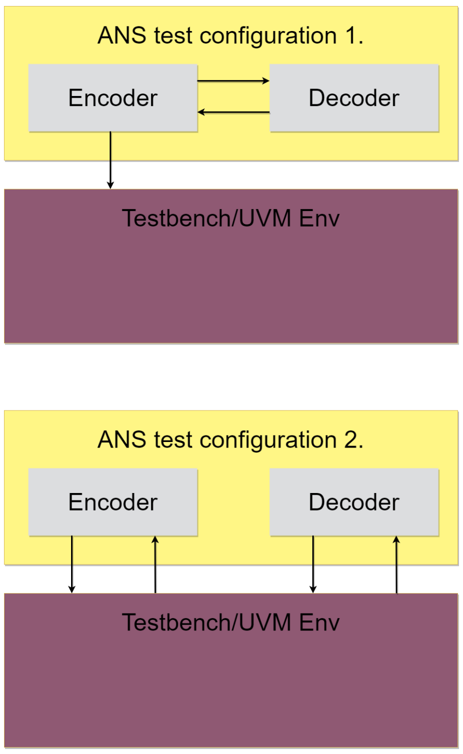


Figure 24: Third iteration of the encoder Finite State Machine diagram.

3.4.3.1 Master testbench

The master testbench's role is to generate large amounts of stimuli to drive the device under test (DUT), here being the ANS compressor. It must also be able to evaluate whether the compression succeeded and subsequently export data from the testbench for external analysis. The testbench relies on a set of helper functions and tasks to be able to function correctly:

- tb_reset() - Resets all variables within the testbench and drives a reset to the DUT
- calculate_probability() - Calculates the probability of the dataset (stimuli). The probability is converted from a floating number to a fixed_precision format. Furthermore, the function accepts a probability deviation factor, which offsets the probability by an amount n. This is important to gauge the effect of inaccurate probability modelling on the DUTs compression performance.
- encode_decode() - Drives the stimuli generated by the testbench to the DUT at appropriate timings. It also performs performance analysis, and records metrics such as compression ratio and the number of failed compression cycles.
  - single_run() - Is used to interface correctly with the DUT, and drives the flags for the Ready/Valid interface.
  - decode_one_bit() - Uses the single_run task to drive the appropriate flags at the appropriate timings to the Decoder.
  - encode_one_bit() - Uses the single_run task to drive the appropriate flags at the appropriate timings to the Decoder.

The SystemVerilog style pseudo-code in 7 demonstrates the functionality of the master testbench.

```
module master_tb #(...)(...);

  // Instantiate the DUT
  design_top #(...) design_top (.*);

  generate_clock();

  /* Stimuli */
  initial begin

    tb_reset;
    open_file_and_export_data();

    // Test 1 Pseudo Random A
    for (ii = 0; ii <= RANDOM_ITERATIONS - 1; ++ii) begin
      generate_random_stimuli();
      for(jj = 0; jj < random_constant; ++jj) begin
          position = random_position();
          compression_stimuli[position] = 1;
      end
      calculate_probility(...);
      encode_decode(...);
      export_results(...));
    end

    // Test 2 Random Stimuli B(not shown)
    // Test 3 Bit shifts (not shown)

endmodule : master_tb
```

Listing 7: SystemVerilog style pseudo code for the Master testbench

3.4.3.2 UVM testbench

The UVM testbench was built up from the ground up based on industry advice and experience. The testbench was based on strategies and methodology proposed by Cadence in their course "SystemVerilog Accelerated Verification with UVM" [1] (completed while working with ARM). The verification phase represents a vast amount of development time, where the development was influenced by having worked with cadence verification products prior to starting this master thesis (such as accelerated UVM). As opposed to the master testbench, the UVM testbench contains a handful of classes, functionalities and follows industry standards where applicable, as well as conventions and expectations for how a testbench performs as presented in section 2.6. Furthermore, debugging and ensuring the correct functionality is a major design hurdle when creating UVM testbenches. There are few open-source well documented setups freely available online, which comes in handy due to their complexity and often intricate setups that are either not well documented or hard to make from scratch. That is not to say that UVM is a bad standard, quite the opposite. It is a very agile industry standard, however complex and with few high quality and well documented open source full-blown examples.

The UVM testbench designed for the verification of the ANS compressor is rather intricate. It contains over 20 classes and more than two thousand lines of dense object oriented code (not counting the inherited UVM base classes). Some features are straight forward to understand, such as packet structure containing some features such as virtual interfaces and internal ports are far more complex. Fully outlining all design steps in an understandable manner is beyond the scope of this master thesis, as it either requires the reader to already have an understanding of how UVM works, or fully outline all the details that must be understood, probably with the same granularity as a commercial UVM course. Therefore, only an overview is provided, with the most notable design decisions and crucial features. The UVM in it's entirety is available in the appendix D.

The UVM testbench consists of the following core modules:

1. uabs
   (a) uabs_env.sv
   (b) uabs_if.sv
   (c) uabs_packet.sv
   (d) uabs_pkg.sv
   (e) uabs_scoreboard.sv
   (f) uabs_tx_agent.sv
   (g) uabs_tx_driver.sv
   (h) uabs_tx_monitor.sv
   (i) uabs_tx_seqs.sv
   (j) uabs_tx_sequencer.sv
2. tb
   (a) clkgen.sv
   (b) hw_top.sv
   (c) testbench.sv
   (d) uabs_tb.sv
   (e) uabs_test_lib.sv
3. channel
   (a) channel_env.sv
   (b) channel_if.sv
   (c) channel_packet.sv
   (d) channel_pkg.sv
   (e) channel_channel_resp.sv

(f) channel_rx_agent.sv
(g) channel_rx_driver.sv
(h) channel_rx_monitor.sv
(i) channel_rx_seqs.sv
(j) channel_rx_sequencer.sv

An overview of the UVM Testbench is seen in figure 25.

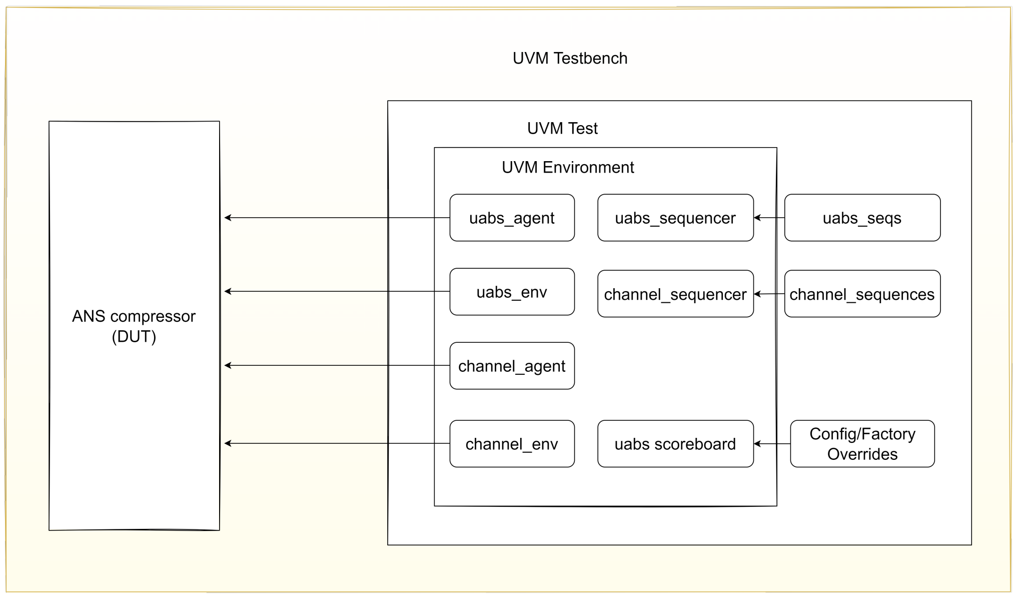


Figure 25: Overview of the UVM Testbench. Showing both the main model (uABS) and a channel, which may be used by a reference model.

The channels are configured, however not fully utilized in the current iteration of the UVM testbench. As channels they are set up to interface with a reference model, such as the SystemVerilog software model described in section 3.3.

The stimuli being driven to the systems are defined in the uabs_packet.sv. This is where the variables must be declared, the utility field macros must be set up and stimuli constrained. All the variables were defined as packed integral properties using the 'uvm_field_int macro. It is crucial that the constraints applied to the variables is correct, and neither too wide nor too narrow. For instance, the probability constraints is defined between 1 and (1 ¡¡ FBITS) - 1. This essentially translates to probabilities higher than 0 and less than 100%. It is important to have a good understanding of the behavior and properties of the DUT when creating packets. SystemVerilog style pseudocode is supplied in illustration in listing 8.

```
class uabs_packet extends uvm_sequence_item;
  parameters
  declarations

  uvm_component_macro

  class_constructor();

  // Some examples of constraints
  constraint to_encode_range {0 <= to_encode; to_encode <= 1;}
  ...
  constraint probability_range {1 <= probability; probability <= (1 << FBITS) - 1;}
  ...
  constraint start_enc_delay_range {MIN_INTERFACE_DELAY <= start_enc_delay;
    start_enc_delay
endclass : uabs_packet
```

Listing 8: SystemVerilog style pseudo code for uabs packet in the UVM testbench

The uabs_packet is seen in it's entirety in the appendix, at section D.

Two important modules are the drivers and the monitors. Both modules are configured relatively similarly, with the purposes of driving and monitoring the DUT, using the virtual interface (vif, as declared in uabs_if) to connect with the DUT. As for most classes they are extensions of the UVM base classes of the same types (the same goes for e.g. the packet), and it is important that they conform with the conventions of UVM. One example is the connect_phase where it is good practice to check whether the Interface is correctly configured, and during the run_phase when the packets must be driven to and monitored from the DUT. Lastly during the report_phase where the modules should report how many packets they have driven and/or monitored. This number should be equal for both the driver and monitor. Otherwise packets could have either been missed, dropped or driven at wrong times. Lastly, it is important to notice that it is good practice for the stimuli to be driven during the negative clock edge, assuming that the design operates on positive edge.

Cover points have been defined inside the uabs_monitor. These cover points are used to measure how well a packet attribute (stimuli) has been covered during the pseudo random directed stimuli generation, as seen inside the uabs_packet. These coverpoints are defined inside covergroups and are instantiated inside the class constructor, however if and only if coverage is enabled. Lastly, the coverage reports are presented during the report_phase where a percentage between 0 and 100 represents how well the attributes are covered. For attributes with more than two values, buckets must be defined. The buckets were defined precicely to create a coverage report with high granularity, which makes the data highly detailed. This comes at a trade off of longer processing times. The coverage could alternatively have been defined within it's own module, which would have been well advised for verification of large systems. The SystemVerilog style pseudo code 9 visualizes the core components of the monitor.

```
class uabs_tx_monitor extends uvm_monitor;

    declare_analysis_port();
    virtual interface uabs_if vif;

    uvm_component_macro

    covergroup cover_packet;
    covergroup cover_packet_states;
    covergroup cover_packet_flags;

    class_constructor();

    connect_phase();

    // Run_phase is the most important phase for monitor
    // Here packets are detected, collected, transactions handled and coverage
    performed.
    run_phase(uvm_phase phase);

    report_phase();

endclass : uabs_tx_monitor
```

Listing 9: SystemVerilog style pseudo code for uabs monitor in the UVM testbench

The uABS_if supplies the driver and monitor with methods to facilitate the transaction with the DUT. It may be implemented in a multitude of ways and is modelled here with three methods named collect_packet, send_to_dut and uabs_reset. Notably the collect and send methods use wait() statements to ensure proper interfacing when the DUT. It may require a number of clock cycles before receiving some signals, depending on where it is in its flow. For instance, under proper operation the Valid in signal should only arise from the testbench after the Ready-in flag has been raised by the DUT. Using wait() statements effectively ensures that the attributes of the packets (stimuli) are delivered at the correct timings. These functions are unfortunately sufficiently flexible to deliberately deliver stimuli at the incorrect timings. One example is to drive the input lines of the state or to_encode before the valid in flag is high. It is important to test how fragile or robust the circuit is with respect of missing timed flags or early/late stimuli arrival, given that the ANS compressor is configured with a Ready/Valid interface. Some timings are however made strict. For instance, the monitor will only sample the encoded and decoded representations upon receiving the Valid Output flag from the DUT. It will however not wait for the DUT to be ready before flagging valid in, and vice versa, since it is decided by the random delay attributes of the packets.

Lastly, the scoreboard clones the packets collected by the monitor and evaluates whether they were successful or failed. It could alternatively also report other metrics such as whether the packages were supposed to fail, and failed successfully, or compare them to the performance of a reference model. For this build of the uabs_scoreboard the inputs and outputs prior to and after compression were compared for equality. Essentially, the scoreboard will fail the packet if the reproduced representation of the stimuli does not match the original stimuli, meaning the compression had failed. It ensures that both the input bit and state were correct. Other factors may also be monitored such as correct interfacing, which here was solved through assertions in the uabs_if.

The expected output of the UVM Testbench is a summary made by the scoreboard, and its form factor is largely up to the designers preferences. A simple and effective layout is recommended, where the scoreboard should at least contain printouts as displayed in listing 10.

```
#  Scoreboard: Packet Statistics
#  Packets in 50000       Packets Dropped: 0
#      Channel 0 Total: 0 Pass: 50000 Miscompare 0 Dropped 0
#
# Simulation Passed
#
```

Listing 10: An example of how the scoreboard printout should like like upon a successful simulation

4 Results

4.1 Introduction to results

A modular and configurable hardware implementation of the uABS variant of the ANS compression algorithm has been created. It has been synthesized and implemented successfully using the EDA Vivado, and verified using an industry standard for verification, UVM, as well as more novel testbenches with detailed analysis and some coverage. The verification reports imply that under normal circumstances and with proper interfacing no bugs are to be expected. The compressor is able to achieve compression ratios approaching the theoretical limit. There are two hardware implementations available, one with and one without hardware-specific optimizations. Only the optimized synthesizable will be presented in its entirety with exhaustive testing and analysis. The unoptimized synthesizable ANS compressor is largely inferior to the optimized counterpart and is in essence presented as its concept, and therefore valuable as a basis for discussion and future work. Both hardware implementations have undergone much of the same verification testbenches and passed their criteria, however, the optimized implementation has undergone some additional tests. The result section will focus on the Optimized hardware implementation for discussions and conclusions, largely ignoring the unoptimized hardware implementation. Furthermore, two accompanying software models have been developed. These software models are intended as reference models, used for research, experimentation and to display the properties of the ANS algorithm. Their value is tied to their ability to bridge the gap between high and low abstraction levels as the developer works from the algorithms as equations to the hardware implementation as a circuit. The code for all four designs and their verification environments are found in the Appendix at 7. The illustration in figure 26 gives an overview of the four designs.

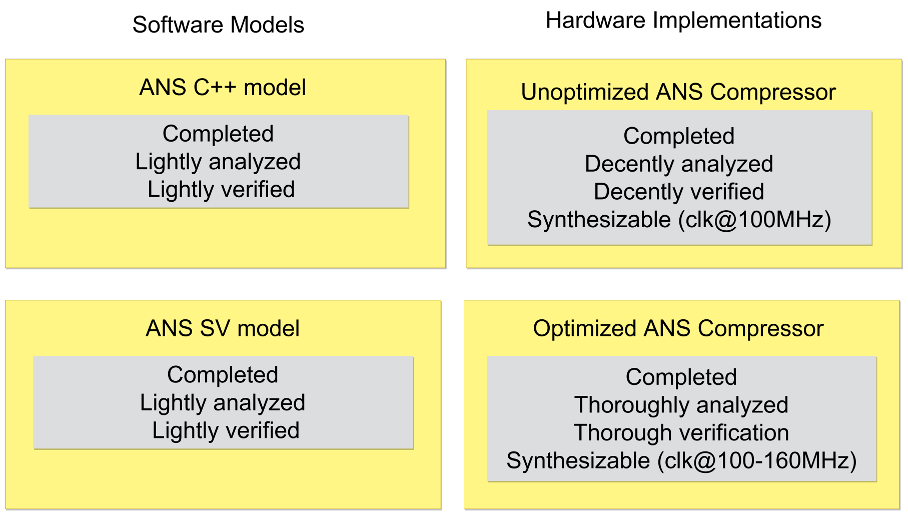


Figure 26: An overview of the four designs. Two hardware implementations and two software models.

4.2 Parameters

The designs are deliberately and thoroughly parameterized, this allows for high configurability and adds to their re-use factor. It also majorly affects at which conditions they are able to operate. For instance at greater WIDTHs they may allow greater inputs (stimuli) without overflowing and producing the wrong outputs. Unless otherwise stated the default parameters are as displayed in table 11:

Table 11: WIDTH is the default width of the Encoder/Decoder input output and intermediate result data types(reg/wire/logic), whereas FBITS is the fixed floating point decimal precision width.

Name	Synthesis and Implementation	Verification
WIDTH	24	180
FBITS	4	8

It is worth noting that the verification utilizes larger WIDTH and FBITS. This is employed in order to run more wide-covering exhaustive testing with a greater range of stimuli. The Synthesis and Implementation parameters are more in line with a realistic use case where re-normalization is employed (keeping the input state low). 4 bits FBIT was found to be low if accurate probability modelling is employed, while 8 was on the high side, and 6 worked well under all circumstances. Furthermore some additional parameters are required for the optimized encoder circuit:

Table 12: Bit sizes of all the input/output and intermediate result data types, see explanation below table.

Name	Synthesis and Implementation	Verification
WIDTH	24	180
FBITS	4	8
CLK_DIVISOR	4	4
DUTY_DIVISOR	4	4

Width signifies default width of Encoder/Decoder, input/output and intermediate result data types: reg, wire and logic. FBITS is fixed point decimal precision width. CLK_DIVISOR is frequency f/ CLK_DIVISOR for Encoder FSM operations. DUTY_DIVISOR is the divisor of the duty cycle, becoming (1/DUTY DIVISOR)*100%. For default parameters clock is slowed down by a divisor of 4 with a duty cycle of 25%

4.3 Optimized ANS compressor hardware implementation

The following section presents the final results of the ANS compressors hardware implementation. The first section 4.3.1 presents the verification of the top module (i.e. both Encoder and Decoder). The following section 4.3.2 presents the synthesis and implementation reports for the two target FPGA. Thereafter results from the performance analyses are presented, and lastly the RTL analysis is shown in section 4.3.6.

4.3.1 Waveforms and verification

A multitude of methods were employed for verification of the ANS compressor. Multiple testbenches were configured as presented in section 3.4, both with and without Universal Verification Methodology (UVM). These testbenches produced a set of verification reports alongside waveforms, terminal printouts, and data points for analysis. All verification environments were constrained to perform within the allowed operational range of the ANS compressor, whereas all the simulations passed within these frames. When combining all simulations, roughly 40 million encoding and decoding operations have been recorded (totalling 80 million). These results were captured using several commercial simulators such as Vivado, Cadenxe Xcelium, Aldec Rivera Pro and Synopsis VCS (courtesy of EDA Playground). Roughly 80% of these samples were stored for analysis, whereas the latter 20% could not be extracted as they were performed in the cloud at EDA Playground (with heavily restricted terminal printout limits). In total the combined run time of all the simulations lands at 100 hours. This estimate excludes the samples created during development, and only account for the verification done on the final build of the ANS compressor.

4.3.1.1 UVM environment reports

The UVM testbench was run with two different configurations, with and without interface delays and coverage. The environment was restricted to one minute of run time due to EDA vendor specific restrictions. The two configurations were able to run 50 000 and 140 000 encode and decode cycles respectively before being cut off. While not a very large sample pool, it is sufficient to draw some conclusions on the accuracy of the hardware implementation of the ANS compressor. Because of the runtime limitations of one minute, the circuit was tested at the standard synthesis parameters, as they are narrower. For verification with greater sample size see 4.3.1.2 which ran without run time restrictions. The standard uABS_packet was used as described in section 3.4.3.2.

As may be observed in the table 13 all the packets sent from the driver in the UVM Testbench resulted in passes at the scoreboard, and the test was thus unambiguously passed. No packets neither failed nor were dropped. Furthermore the table 14 confirms that all bit and probability combinations were present in the simulations. This means that all input bits (i.e. 1's or 0's) has been combined with all the computed optimal probabilities as well as those subjected to deviations to analyze less than optimal probabilities. The flag delays were also fully covered, defined in our case as tested with independent interfacing flag delays of up to 100 clock cycles. For instance ,one run the start flag may be prompted at the tenth clock cycle and the valid input flag at the hundredth clock cycle. Lastly ,the coverage of all possible legal input states is 93.75% at 50 000 packets, although deemed probable that this number would approximate 100% if the run time limitation of 1 minute was not present, EDA vendor specific restrictions prevented the testbench from reaching this last percentage of coverage. Nonetheless ,a high degree of coverage was achieved which bring confidence in the ANS compressors' ability to compress correctly. It is guaranteed to perform correctly for 100% of the probabilities and bit combinations as long as deviations around the optimal probability does not approach 20%, and for most, and probably all of the possible state input combinations. It is also guaranteed to operate correctly with any combination of any legal flag behavior at its interface, within 100 clock cycles of imitating the compression operation

Table 13: UVM Scoreboard report from 140 000 samples in the UVM testbench. With WIDTH = 24 and FBITS = 4

Packets sent	Packets passed	Packs failed	Packets dropped
140 000	140 000	0	0

Table 14: Coverage from 50 000 packets in the UVM testbench. With WIDTH = 24 and FBITS = 4

Coverage name	Coverage
Probabilities and encoded/decoded bit	100%
Flag delays	100%
States as stimuli (of all possible states)	93.75%

for both the encoder and decoder (if chained in total 100 + 100 clock cycles). Note that standard synthesis parameters for WIDTH and FBITS have been used.

4.3.1.2 Master testbench reports

The master testbench is the source of the majority of samples, as it was run locally without any runtime restrictions, it however was not run with UVM due to issues with Vivado and UVM, for further discussion see section 5.12 for more details. While a hurdle, this was not a roadblock as large quantities of constrained pseudo random stimuli could still be generated without UVM, that is with the Master testbench. Table 15 shows the tests that were run, their configurations and the results from them. The 224.683 bit-strings that were generated were 150 bits wide, and thus resulted in 33.702.450 encode and decode operations (for a total of 67 404 900). The tests were carried out with probability deviations divided into buckets spanning up to -20% to +20%, added to or subtracted from the ideal computed probability. For instance for the bit string 1000 the accurate probability of 1's is 25%, given a probability deviation of 10% the resulting probability becomes 35%. Of all the probability deviations, only samples with 20% probability deviation resulted in some failed compression (due to internal overflow). These tests are however not considered important for the ANS compressor itself, as the probabilities are heavily modified on purpose to gauge how inaccurate the probability may be modelled before it negatively alter the compressors accuracy and behavior. It was thus observed that the ANS compressor is able to safely encode and decode datasets with up to 10% inaccuracy in the probability modelling.

It is important to note the value of having different tests and the diversity they bring. The UVM testbench drives seemingly random states, random probabilities and a random bit into the Encoder. On the other hand, the master testbench drives carefully crafted stimuli with the purpose of leading to data driven external analysis. Here the stimuli are carefully crafted to explore and challenge various properties of the ANS compressor, which may better be analyzed in analysis tools externally, as presented in section 4.3.5. Following this strategy, both testbenches complements one another with different purposes, properties and test cases. The Master testbench adds to the results from the UVM Testbench and brings high confidence in the correct behavior of the ANS compressor, and it's ability to also compress stimuli with moderately sub-optimal probability (+/-10%) correctly.

Table 15: Verification report from the Master testbench.

Test name	Parameters	Forced P offset	E+D cycles	B Strings	Failed	Fail reason
Pseudo Random A	Max WIDTH	0%	3 747 000	24980	0	N/A
Pseudo Random A	Max WIDTH	1%	3 708 000	24720	0	N/A
Pseudo Random A	Max WIDTH	5%	3 827 700	25518	0	N/A
Pseudo Random A	Max WIDTH	10%	3 752 400	25016	0	N/A
Pseudo Random A	Max WIDTH	20%	3 710 700	24738	4.17%	Overflow
Pseudo Random A	Max WIDTH	-1%	3 766 800	25112	0	N/A
Pseudo Random A	Max WIDTH	-5%	3 744 450	24963	0	N/A
Pseudo Random A	Max WIDTH	-10%	3 727 950	24853	0	N/A
Pseudo Random A	Max WIDTH	-20%	3 717 450	24783	12.42%	Overflow
Bit shifts	Max WIDTH	0%	22 650	151	0	N/A
Bit shifts	Max WIDTH	1%	22 650	151	0	N/A
Bit shifts	Max WIDTH	5%	22 650	151	0	N/A
Bit shifts	Max WIDTH	10%	22 650	151	0	N/A
Bit shifts	Max WIDTH	20%	22 650	151	4.67%	Overflow
Bit shifts	Max WIDTH	-1%	22 650	151	0	N/A
Bit shifts	Max WIDTH	-5%	22 650	151	0	N/A
Bit shifts	Max WIDTH	-10%	22 650	151	0	N/A
Bit shifts	Max WIDTH	-20%	22 650	151	5.3%	Overflow

4.3.1.3 Decoder waveforms

The waveforms are created using Vivado simulation dump files. They have been cleaned up and highlighted using the standalone tool Timing Analyzer. The stimulus used is the binary string 11001100. This string is firstly encoded using the decoder, and then the encoded state is used to recreate the original representation by the decoder. The compression process is carried out using the probability $p = 0.5$. As there are 8 bits of stimuli with a uniform probability distribution, both the encoder and the decoder are required to carry out 8 encode/decode cycles each. As may be observed, the encoding and decoding process was carried out successfully as the decoder was able to successfully reproduce the original representation 11001100. The Ready/Valid interfacing is seen taking place as presented in section 3.4. The importance of visually inspecting that the interfacing is functioning correctly is not to be underestimated and constitutes a vital part of the qualitative verification preceding the quantitative, mass generated testing, or even as a final assessment.

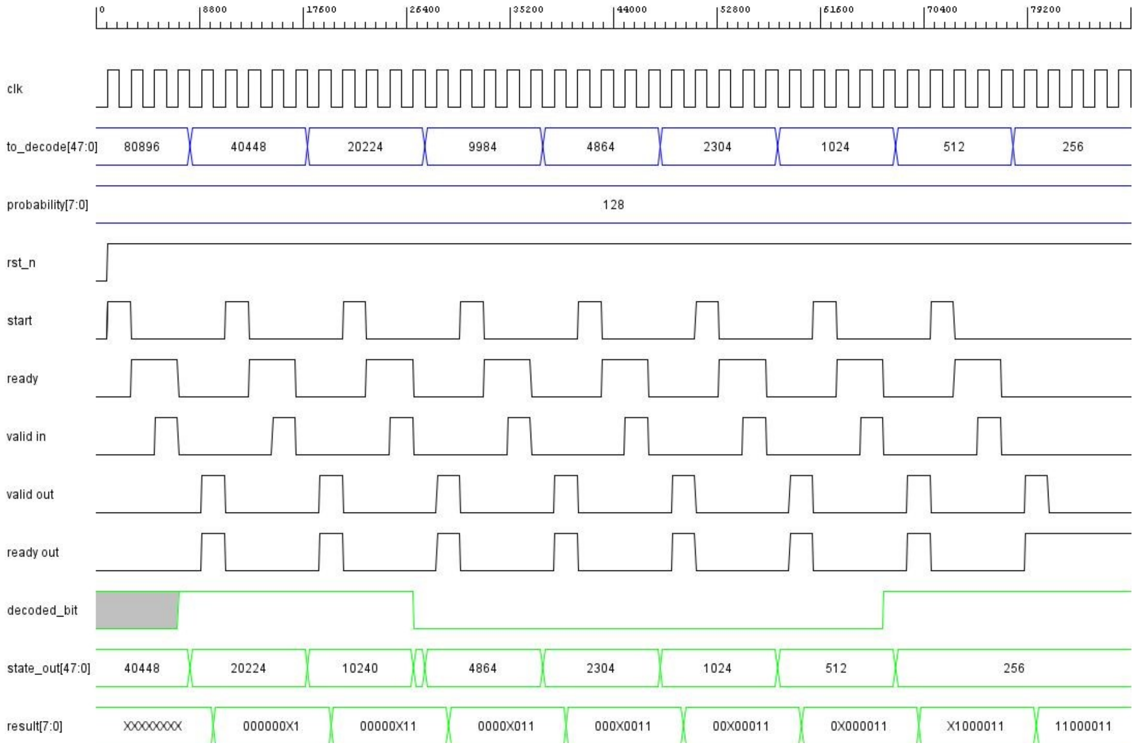


Figure 27: Waveforms showing eight decode cycles. The decoder is given an input state (to_decode) and a probability, and is able to correctly reproduce the original representations (result).

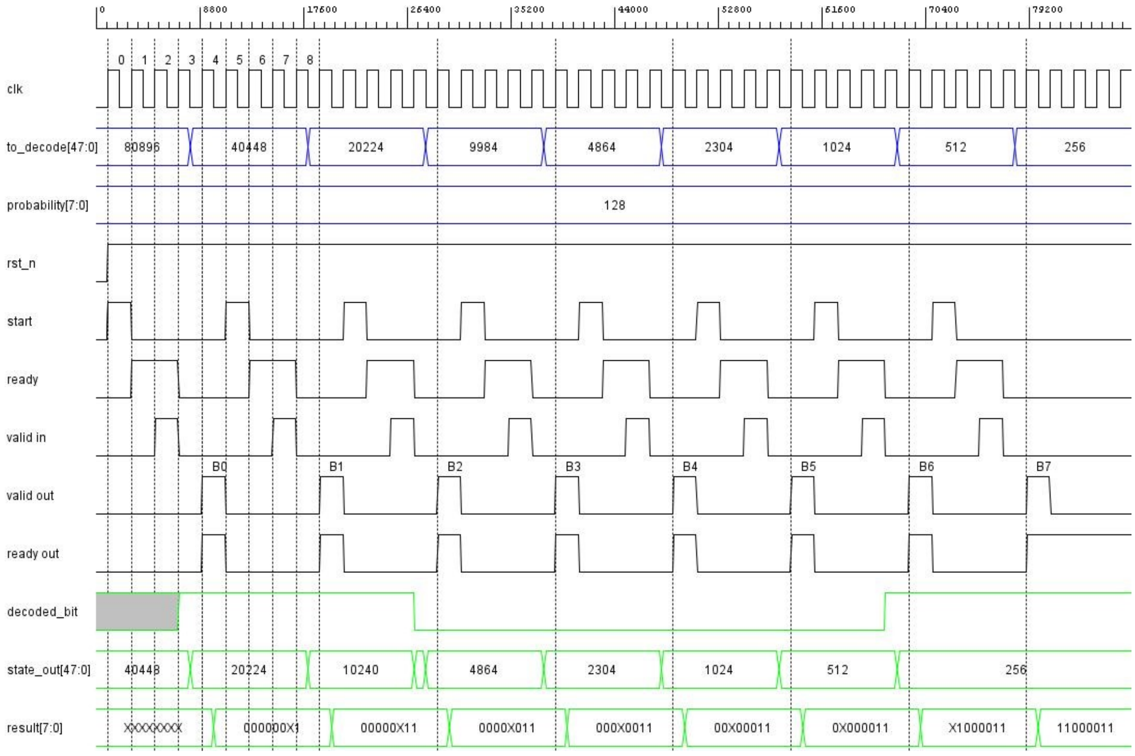


Figure 28: Waveforms showing eight decode cycles. Labels are placed on the positive edge of the clock cycles, revealing that one decode operation required five clock cycles.

4.3.1.4 Encoder waveforms

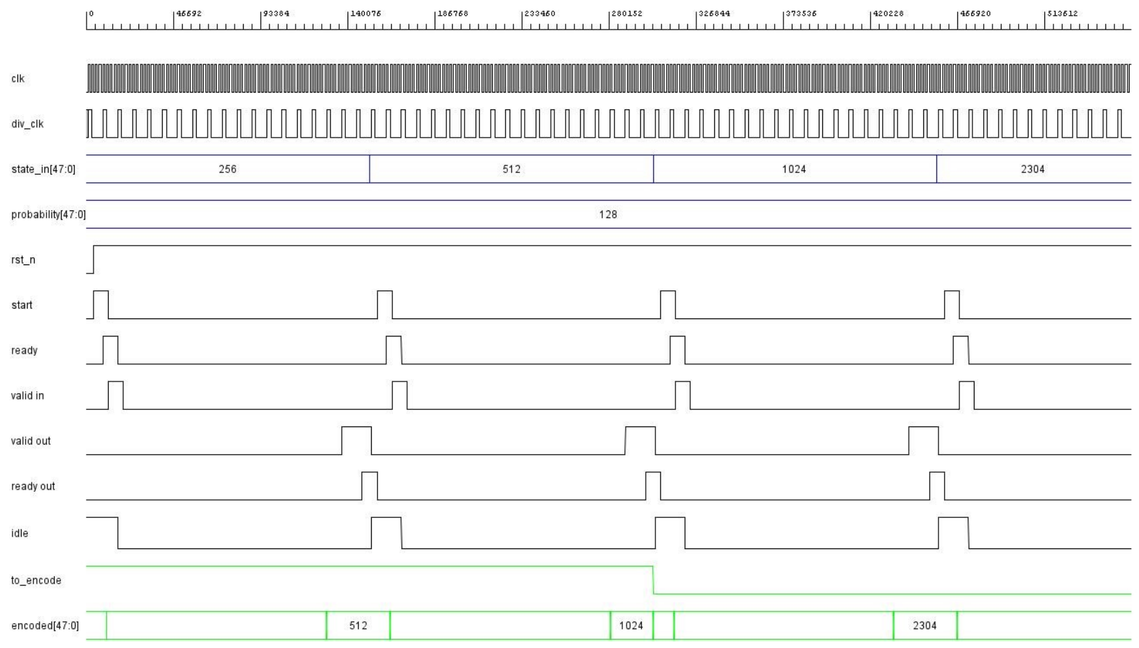


Figure 29: Waveforms showing four encode cycles. The encoder is given an input state (state_in), a bit to encode (to_encode) and a probability. It is able to correctly encode all bits, as seen in figure 30.

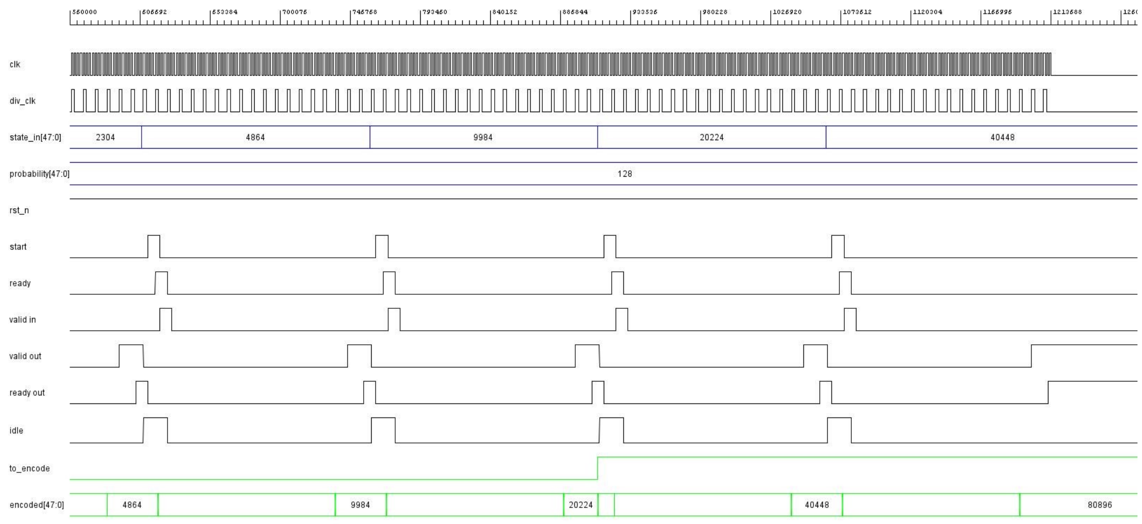


Figure 30: Waveforms showing four encode cycles, which continues from illustration 29. The encoder is given an input state (state_in), a bit to encode (to_encode) and a probability. It is able to correctly encode all bits.

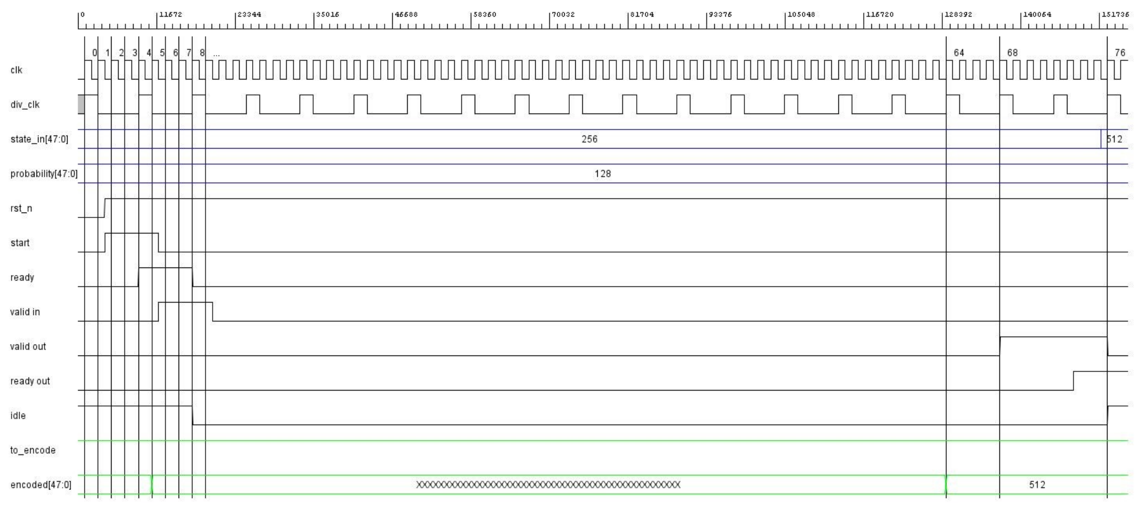


Figure 31: Waveforms showing one encode cycles. Labels are placed on the positive edge of the clock cycles, revealing that one encode operation required 76 clock cycles.

4.3.2 Synthesis and implementation reports

Synthesis and implementation was carried out using the Vivado toolset. The synthesis was done using the standard synthesis parameters as presented in section 4.2. Overall the results are very promising in terms of utilization, timings and power consumption. Furthermore it is observable that the decoder uses slightly less resources than the encoder, however only marginally. Furthermore the opposite is true for power consumption, where the Decoder uses more energy due to it's dominantly combinatorial logic (for instance it uses a DSP). Based on the findings of the synthesis and implementation reports it is fair to conclude that the implementation is inherently very lightweight, and operates at respectable frequencies.

The ANS Compressor was simulated for two FPGA development cards, the Xilinx Zynq UltraScale+ ZCU106 and the Xilinx Spartan-7-SP701. There is a significant performance difference between them, as the Zynq UltraScale outperforms the Spartan-7-SP701 in every category except on power consumption, and is equipped as a MPSoC with general and real-time processing cores, among others. This discrepancy in performance is largely caused by two different technologies being employed in the FPGAs, where the Zynq UltraScale platform is better suited for better performance at a higher price (3-4 times more expensive) and carrying already mentioned additional circuitry, adding to the static power consumption. For more information on the cards see section 2.3.2.

4.3.3 FPGA Zynq UltraScale+ ZCU106

The Xilinx Zynq UltraScale ZCU106 passed synthesis and implementation without any errors or severe warnings. The following sections present the results for both the Encoder and the Decoder. The RTL was largely translated into configurable logic blocks (CLBs), whereas most of the CLBs are synthesized as lookup tables (LUTS). There are also 4 DSPs and the mandatory Bonded Input output Buffers for interfacing with I/O. All the utilization reports are well within the margins of what the FPGA has available of resources. If you exclude the Bonded IoBs consumes, then resource consumption of every category is less than 1% of the resources available. Furthermore, it is observable that the FSM resulted in a very lightweight implementation, as seen in for instance table 16.

The dominant factor of resource consumption comes from the arithmetic module, which has the most complex logic. Notice that CARRY8 blocks are synthesized. These are eight MUXes and eight XORs which are connected to the LUTs. It is also somewhat noticeable in the visualization of the RTL analysis of section 4.3.6, as separate XOR gates and MUXes. Lastly, there's one global clock buffer used by the main clock.

As seen in the timing table after synthesis 18, as opposed to that after implementation 19, there appears to be some negative hold slack. This negative hold slack (see section 2.3.4) surfacing after synthesis but prior to implementation, and is a known hurdle to handle according to Xilinx (creator of the FPGAs and Vivado)30, and can largely be ignored if it is not prevalent during implementation [44]. As observed, this was not an issue and was fixed during implementation as seen in table 19.

4.3.3.1 Decoder

The synthesis of the Decoder ran successfully and yielded promising results, as seen from table 16 to 20. The power consumption is displayed in figure 32.

Table 16: Utilization report for synthesis of the optimized decoder circuit for Zynq UltraScale+ ZCU106

Name	CLB LUTs	CLB Registers	CARRY8	DSP	Bonded IOB	Global Clock Buffers
top	105	5	12	4	113	1
encoder_fsm	5	4	0	0	0	0
encoder	100	1	12	4	0	0

Table 17: Utilization report for implementation of the optimized decoder circuit for Zynq UltraScale+ ZCU106

Name	CLB LUTs	CLB Registers	CARRY8	CLB	LUT as Logic	DSP	Bonded IOB	Global Clock Buffers
top	104	6	12	26	104	4	113	1
encoder_fsm	4	4	0	3	4	0	0	0
encoder	100	2	12	23	100	4	0	0

Table 18: Synthesis timing report of the optimized decoder circuit for Zynq UltraScale+ ZCU106

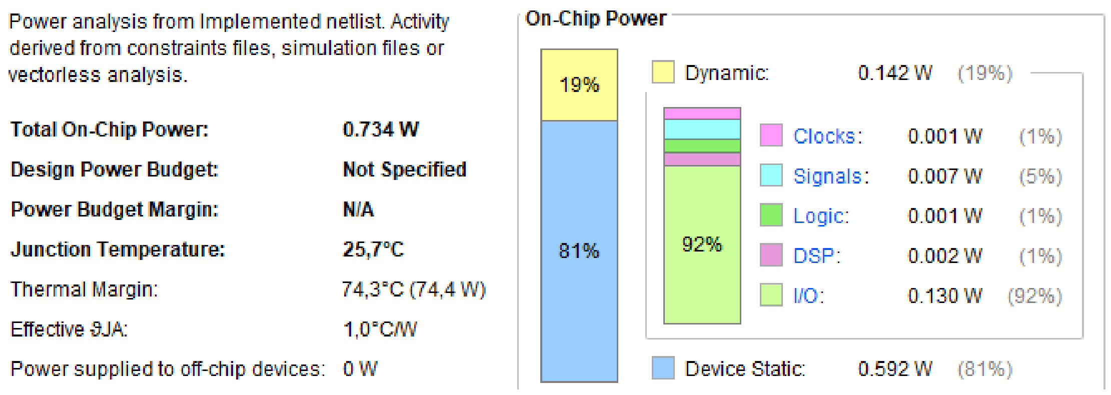
Setup	Hold	Pulse Width
WNS0.013ns	WHS-0.064ns	WPWS 2.700ns
TNS 0.000ns	THS-0.229ns	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 0	Failing Endpoints 0
Endpoints 10	Endpoints 10	Endpoints 5

Table 19: Implementation timing report of the optimized decoder circuit for Zynq UltraScale+ ZCU106

Setup	Hold	Pulse Width
WNS0.150ns	WHS0.047ns	WPWS 2.700ns
TNS 0.000ns	THS0	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 0	Failing Endpoints 0
Endpoints 10	Endpoints 10	Endpoints 5

Table 20: Clock summary report of the optimized decoder circuit for Zynq UltraScale+ ZCU106

Name	Waveform	Period(ns)	Frequency (MHz)
clk	{0.000 2.975}	5.950	168.067


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.734 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,7°C**
Thermal Margin:	74,3°C (74,4 W)
Effective ϑJA:	1,0°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

🟨	Dynamic:	0.142 W	(19%)
🟪	Clocks:	0.001 W	(1%)
🟦	Signals:	0.007 W	(5%)
🟩	Logic:	0.001 W	(1%)
🟪	DSP:	0.002 W	(1%)
🟩	I/O:	0.130 W	(92%)
🟦	Device Static:	0.592 W	(81%)

Figure 32: Implementation and synthesis power report of the optimized decoder circuit for Zynq UltraScale+ ZCU106

4.3.3.2 Encoder

The encoder exhibits many of the same features as those found in the decoder. Some notable differences are found in the inclusion of numerous more registers, and those are almost exclusively due to the fixed division module. One area where the Encoder beats the decoder is in the number of Bonded IoBs, only requiring 65 I/O ports. This leads to a significant saving in power consumption, as the I/O is the major power sink, as seen in figure 33 compared to 32. While the power consumption is largely static, which is a drawback of having a large MPSoC with FPGA functionality, reducing the I/O has significantly reduced the dynamic consumption from 142mW to 33mW. What the Decoder loses to the Encoder in speed and utilization, it makes up for in power consumption.

Table 21: Utilization report for synthesis of the optimized encoder circuit for Zynq UltraScale+ ZCU106

Name	CLB LUTs	CLB Registers	CARRY8	Bonded IOB	Global Clock Buffers
top	170	94	11	65	1
encoder	137	89	8	0	0
encoder.div	129	89	5	0	0
encoder_fsm	29	2	0	0	0
slow_clk	3	3	0	0	0

Table 22: Utilization report for implementation of the optimized encoder circuit for Zynq UltraScale+ ZCU106

Name	CLB LUTs	CLB Registers	CARRY8	CLB	LUT as logic	Bonde IOB	Global Clock Buffers
top	168	94	11	53	168	65	1
encoder	137	89	8	49	137	0	0
encoder.div	128	89	5	45	128	0	0
encoder_fsm	28	2	0	11	28	0	0
slow_clk	2	3	0	1	2	0	0

Table 23: Synthesis timing report of the optimized encoder circuit for Zynq UltraScale+ ZCU106

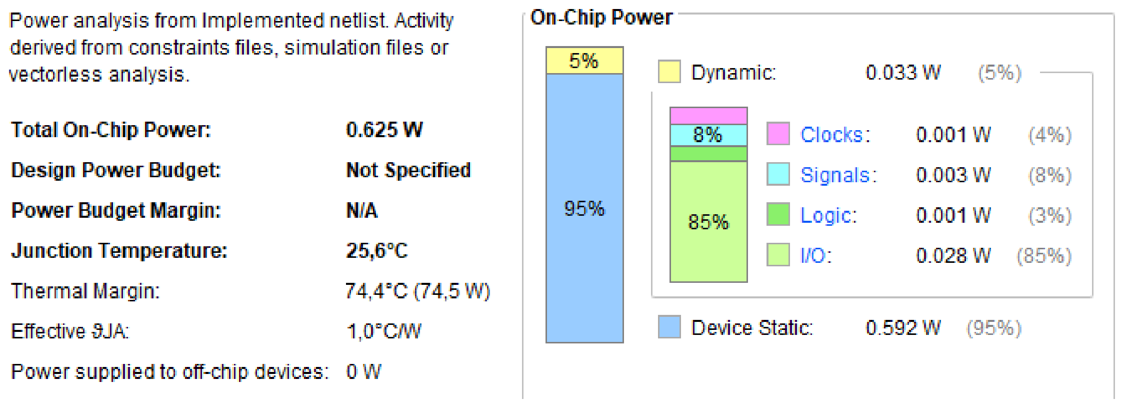
Setup	Hold	Pulse Width
WNS2.471ns	WHS-0.072	WPWS 4.625ns
TNS 0.000ns	THS-4.697ns	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 108	Failing Endpoints 0
Endpoints 266	Endpoints 266	Endpoints 115

Table 24: Implementation timing report of the optimized encoder circuit for Zynq UltraScale+ ZCU106

Setup	Hold	Pulse Width
WNS0.003ns	WHS0.053	WPWS 4.625ns
TNS 0.000ns	THS 0	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 0	Failing Endpoints 0
Endpoints 266	Endpoints 266	Endpoints 115

Table 25: Clock summary report of the optimized encoder circuit for Zynq UltraScale+ ZCU106

Name	Waveform	Period(ns)	Frequency (MHz)
clk	{0.000 4.925}	9.850	101.523


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.625 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,6°C**
Thermal Margin:	74,4°C (74,5 W)
Effective ϑJA:	1,0°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

🟨	Dynamic:	0.033 W	(5%)
🟪	Clocks:	0.001 W	(4%)
🟦	Signals:	0.003 W	(8%)
🟩	Logic:	0.001 W	(3%)
🟨	I/O:	0.028 W	(85%)
🟦	Device Static:	0.592 W	(95%)

Figure 33: Implementation and synthesis power report of the optimized encoder circuit for Zynq UltraScale+ ZCU106

4.3.4 Porting the solution to the FPGA Devboard Spartan-7-SP701

One important difference between the Spartan-7 and the Zynq UltraScale is the naming conventions and architecture. The Spartan-7 utilizes the splices, whereas a slice contains LUTs, flip-flops and multiplexers. They are still stated as LUTs or Registers as seen in for instance table 27.

4.3.4.1 Decoder

The Spartan-7 decoder is fairly comparable to the Zynq UltraScale decoder, with differences only in board specific optimizations and architecture. It operates at a significantly slower clock speed as seen in table 34.

Table 26: Utilization report for synthesis of the optimized decoder circuit for Spartan-7-SP701

Name	Slice LUTs	Slice Registers	DSP	Bonded IOB	BUFGCTRL
top	105	5	4	113	1
encoder_fsm	4	4	0	0	0
encoder	101	1	4	0	0

Table 27: Utilization report for implementation of the optimized decoder circuit for Spartan-7-SP701

Name	Slice LUTs	Slice Registers	Slice	LUT as Logic	DSP	Bonded IOB	BUFGCTRL
top	105	6	45	105	4	113	1
encoder_fsm	4	4	4	4	0	0	0
encoder	101	2	41	101	4	0	0

Table 28: Synthesis timing report of the optimized decoder circuit for Spartan-7-SP701

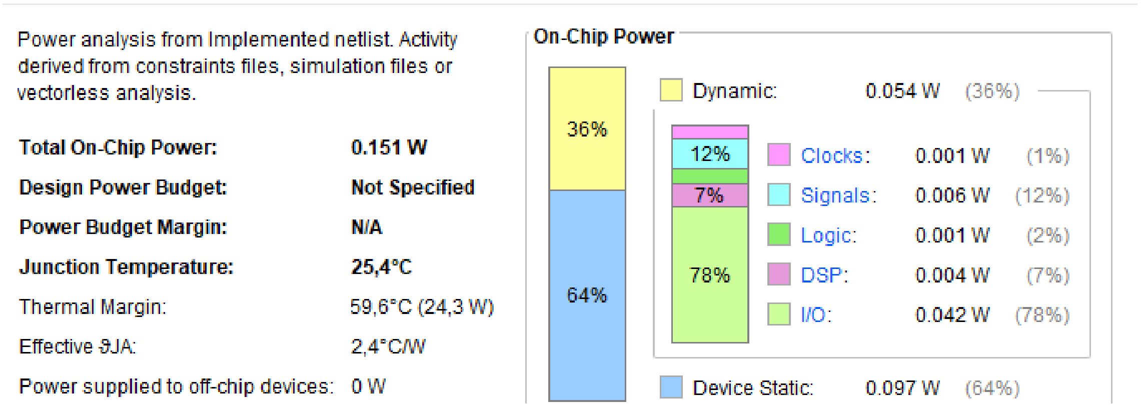
Setup	Hold	Pulse Width
WNS 2.814ns	WHS 0.117ns	WPWS 4.325ns
TNS 0.000ns	THS 0.00ns	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 0	Failing Endpoints 0
Endpoints 10	Endpoints 10	Endpoints 5

Table 29: Implementation timing report of the o/ptimized decoder circuit for Spartan-7-SP701

Setup	Hold	Pulse Width
WNS 0.002ns	WHS 0.212ns	WPWS 4.325ns
TNS 0.000ns	THS 0.000ns	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 0	Failing Endpoints 0
Endpoints 10	Endpoints 10	Endpoints 5

Table 30: Clock summary report of the optimized decoder circuit for Spartan-7-SP701

Name	Waveform	Period(ns)	Frequency (MHz)
clk	{0.000 4.825}	9.650	103.627


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.151 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,4°C**
Thermal Margin:	59,6°C (24,3 W)
Effective ϑJA:	2,4°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

🟨	Dynamic:	0.054 W	(36%)
🟪	Clocks:	0.001 W	(1%)
🟦	Signals:	0.006 W	(12%)
🟩	Logic:	0.001 W	(2%)
🟪	DSP:	0.004 W	(7%)
🟩	I/O:	0.042 W	(78%)
🟦	Device Static:	0.097 W	(64%)

Figure 34: Implementation and synthesis power report of the optimized decoder circuit for Spartan-7-SP701

4.3.4.2 Encoder

Although fairly comparable to the Zynq UltraScale encoder as previously noted in some technologies and board-specific optimizations, note that it does not utilize CARRY-8s, has fewer LUTs but a similar number of registers, as seen in table 32. It is observable that the Spartan-7 while slower, has some advantages over the larger and more powerful Zynq UltraScale.

Table 31: Utilization report for synthesis of the optimized encoder circuit for Spartan-7-SP701

Name	CLB LUTs	Slice Registers	Bonded IOB	Global Clock Buffers
top	147	94	65	1
encoder	115	89	0	0
encoder.div	114	89	0	0
encoder_fsm	29	2	0	0
slow_clk	2	3	0	0

Table 32: Utilization report for synthesis of the optimized encoder circuit for Spartan-7-SP701

Name	Slice LUTs	Slice Registers	Slice	LUT as Logic	Bonded IOB	BUFGCTRL
top	146	94	55	146	65	1
encoder	115	89	49	115	0	0
encoder.div	114	89	44	114	0	0
encoder_fsm	29	2	13	29	0	0
slow_clk	2	3	1	2	0	0

Table 33: Synthesis timing report of the optimized encoder circuit for Spartan-7-SP701

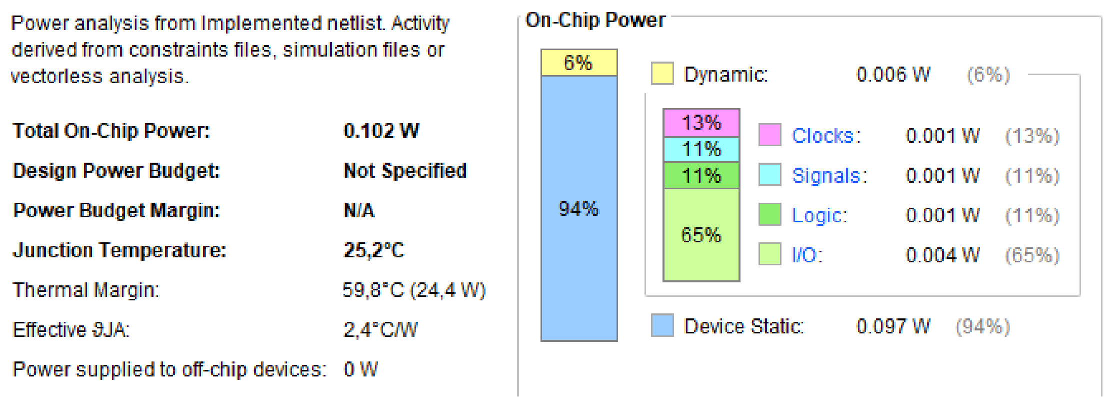
Setup	Hold	Pulse Width
WNS3.198ns	WHS0.127ns	WPWS 6.450ns
TNS 0.000ns	THS 0	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 151	Failing Endpoints 0
Endpoints 220	Endpoints 220	Endpoints 93

Table 34: Implementation timing report of the optimized encoder circuit for Spartan-7-SP701

Setup	Hold	Pulse Width
WNS0.011ns	WHS0.055	WPWS 6.450ns
TNS 0.000ns	THS 0.000ns	TPWS 0.000ns
Failing Endpoints 0	Failing endpoints 151	Failing Endpoints 0
Endpoints 220	Endpoints 220	Endpoints 93

Table 35: Clock summary report of the optimized encoder circuit for Spartan-7-SP701

Name	Waveform	Period(ns)	Frequency (MHz)
clk	{0.000 6.950}	13.900	71.942


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.102 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,2°C**
Thermal Margin:	59,8°C (24,4 W)
Effective ϑJA:	2,4°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

🟨	Dynamic:	0.006 W	(6%)
🟪	Clocks:	0.001 W	(13%)
🟦	Signals:	0.001 W	(11%)
🟩	Logic:	0.001 W	(11%)
🟩	I/O:	0.004 W	(65%)
🟦	Device Static:	0.097 W	(94%)

Figure 35: Implementation and synthesis power report of the optimized encoder circuit for Spartan-7-SP701

4.3.5 Performance analysis

Various analyses have been performed based on the data gathered from simulation, synthesis and implementation. These analyses helped gain an understanding of how the circuit behaves, and metrics such as speed, accuracy and compression ratio may be estimated.

4.3.5.1 Encoding and decoding speeds

Based on the wave forms 30, 31 and 28, presented in section 4.3.1.4 encoding and decoding speed may be calculated. By using the clock frequencies found during synthesis and implementation in section 4.3.2 the maximum number of Mb/s (Mega bits per second) may be calculated. As seen in table 36

Table 36: Peak Decoding Speed for optimized compressor

Board	Cycles per bit [b]	Clock Frequncy [MHz]	Peak Decoding speed [Mb/s]
Zynq UltraScale+ ZCU106	5.00	168.067	33.613
Spartan-7-SP701	5.00	103.627	20.725

If the circuit interfacing the encoder is able to both start the next encoding operation and receive the valid output during the same clock cycle, then the encoder may operate at increased decoding speed. This is achieved by reducing the required cycles per bit from 5 to 4, as seen in table 37.

Table 37: Peak Decoding Speed for optimized compressor assuming better interfacing

Board	Cycles per bit [b]	Clock Frequncy [MHz]	Peak Decoding speed [Mb/s]
Zynq UltraScale+ ZCU106	4.00	168.067	42.016
Spartan-7-SP701	4.00	103.627	25.906

The encoder operates at a significantly slower speed due to the lengthy fixed division calculations, which is the main bottleneck of the compressor. This is displayed in waveform 30 and 31. The processing time of the encoding arithmetic logic is 56 cycles, at clock cycles 8 through 64 (see the markers at CLK) for $WIDTH = 24$ and $FBITS = 4$. Higher WIDTH yields vastly longer processing time, and the opposite is also true for shorter WIDTH, this is because how the division module has to work through all the bits through synchronous operation (i.e. dependent on clk). The Decoder does not have the same problem as its fixed multiplication is solved in combinatorial blocks (which could be synchronous or asynchronous). The decoder is able to perform the encoding within one or two clock cycles as opposed to the encoders 56. The main bottleneck for the decoder is the FSM's synchronous operation. It is worth noting that in its basic form it only transitions from state to state on the positive clock edge.

Table 38: Peak Encoding speed for optimized compressor

Board	Cycles per bit [b]	Clock Frequency [MHz]	Peak Encoding speed [Mb/s]
Spartan-7-SP701	76.000	71.942	0.946
Zynq UltraScale+ ZCU106	76.000	101.523	1.335

4.3.5.2 Compression behavior

Data analysis has been carried out based on the data extracted from the master testbench, as presented in section 15. Table 39 visualizes key findings from the data analysis. A total of 24980 runs, of 150 bits each, were encoded and decoded. The average compression ratio achieved was 0.725. This means that the data were compressed from 150 bits to 109 bits on average, whereas the lowest compression ratio achieved was 0.7% This means that the data was compressed from 150 bits to 1 bit. On the other hand, the highest compression ratio was 101.3%, which means that the data was increased in size from 150 bits to 155 bits. Another important observation is that the average ANS compression ratio of 72.5% was only slightly higher than the average Shannon entropy of 71.8%. Note that lower compression ratios are considered better, as they reduce the number of bits in the encoded state by a greater number. See 2.1.1 for clarifications. This indicates that the compressor approaches the theoretical limit for entropy-based compression, and obviously cannot surpass it on average. These compression ratio calculations do not take into consideration the cost of transmitting metadata together with the encoded state, in particular in a real-life compression scenario the probability used by the decoder that needs to be transmitted to the decoder. This has not been accounted for in these compression ratio metrics. For larger datasets, however, these additional approximately 4-8 bits holding the probabilities will only marginally worsen the compression ratios.

Table 39: Statistics of properties in testbench set, including number of samples, mean, min and max value. Note that lower compression ratios are better. Compression ratios greater than 100% increases the size of the data, and ratios less than 100% reduces the data size.

	Count	Mean	Min	Max
ANS compression ratio	24980	72.50 %	0.70%	101.30 %
Shannon entropy	24980	71.80 %	1.10 %	100.00%
Probability of ones	24980	42.80 %	3.30 %	100.00 %
Encoded	24980	1.993e+44	1.000e+00	3.920e+45
Stimuli	24980	6.124e+44	3.382e+15	1.427e+45

The probabilities present in the data set span from 3.30% to 100%, and the mean probability was 42.8%. This indicates that the dataset had slightly more samples with probabilities lower than 50%, as is confirmed in figure 37. All of the encoding and decoding operations operated with accurate probability modelling, where the only inaccuracy was the conversion from floating-point probabilities to 8-bit fixed point fractions Q(180.8). Figure 37 illustrates the distribution of ANS compression ratios plotted with the Shannon entropies for the dataset. As is observed there are slightly more ANS compression ratio samples with worse compression ratios than the Shannon entropy. There are also numerous cases where the ANS Compressor outperforms the given Shannon entropy, as the Shannon entropy only defines the average theoretical limit. This is also confirmed in table 39 as the lowest compression ratio is significantly lower than the lowest Shannon entropy for the dataset.

Table 40 summarizes the number of samples which failed and number of samples which produced compression ratios lower than the Shannon Entropy. None of the 24980 samples of 150 bits failed during compression, and 4441 of them produced compression ratio lower (and hence better) than the Shannon entropy. This indicates that 17.78% of the runs compressed the data better than the

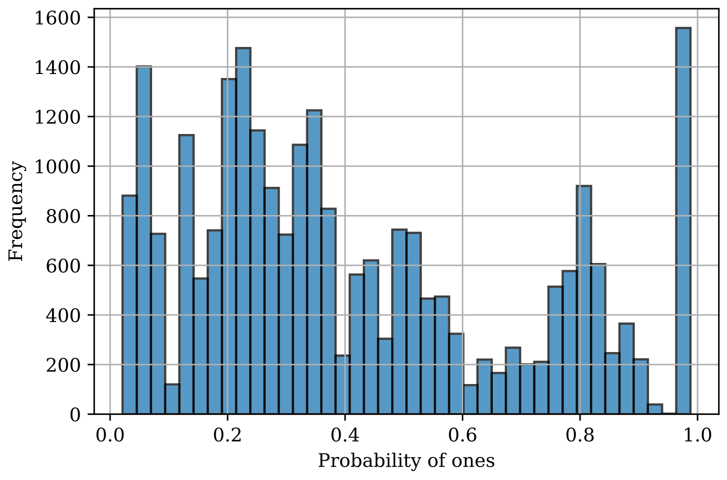


Figure 36: Histogram of distribution of probability of ones in testbench

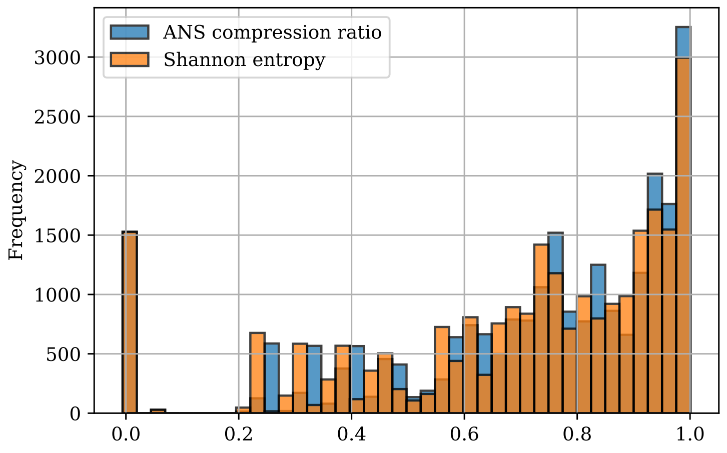


Figure 37: Histogram of compression ratios and Shannon entropy in testbench

average theoretical limit, whereas 82.22% of the runs compressed the data worse than the average Shannon entropy.

Table 40: Number of samples with failed compression and compression ratio less than Sharon entropy

	Compression Status	Compression ratio less than Shannon Entropy
True	24980	20539
False	0	4441

Histograms 38 and 39 illustrates which samples compression ratios are lower than the Shannon entropy. It is thus seen that there is a relation between how large the encoded states are, and how well the compressor had compressed the samples (larger encoded states are worse). This is due to lower values of the encoded state taking fewer bits to store, and thus having better compression ratios, sometimes surpassing that of the average theoretical limit. Similarly, it is observed that the compressor performed well where there were large stimuli, that was encoded into small states, as this results in strong compression.

Figure 40 compares the achieved ANS compression ratio to that of the Shannon entropy. As can be observed, and previously indicated, the ANS compression ratio approaches that of the theoretical limit for entropy based compressors. The ANS Compressor has thus achieved a highly satisfactory

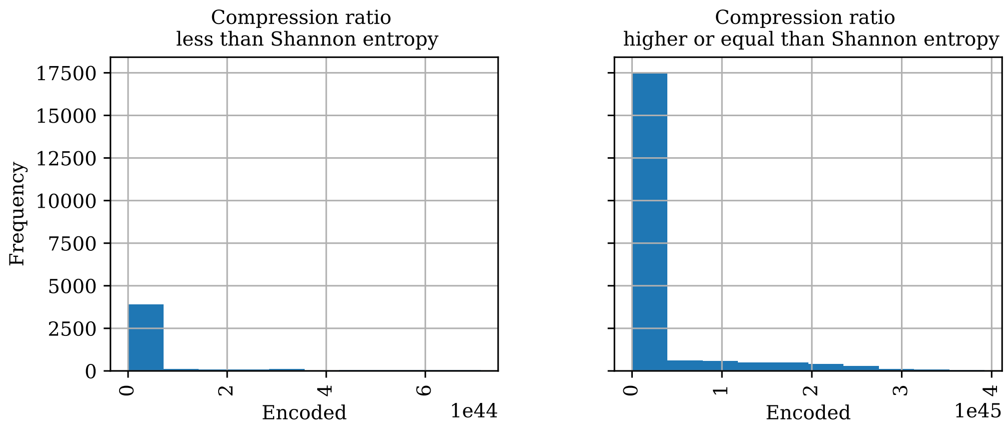


Figure 38: Histogram of encode values for compression failed or not

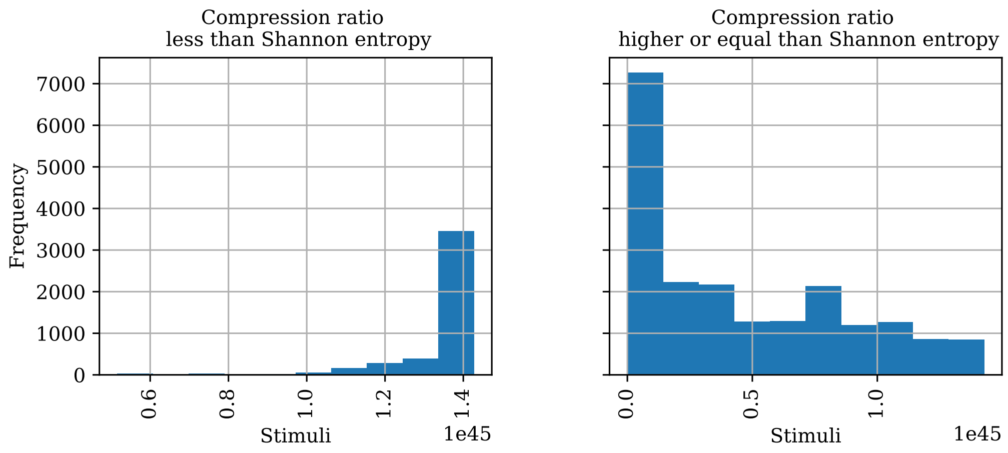


Figure 39: Histogram of stimuli values for compression failed or not

compression performance.

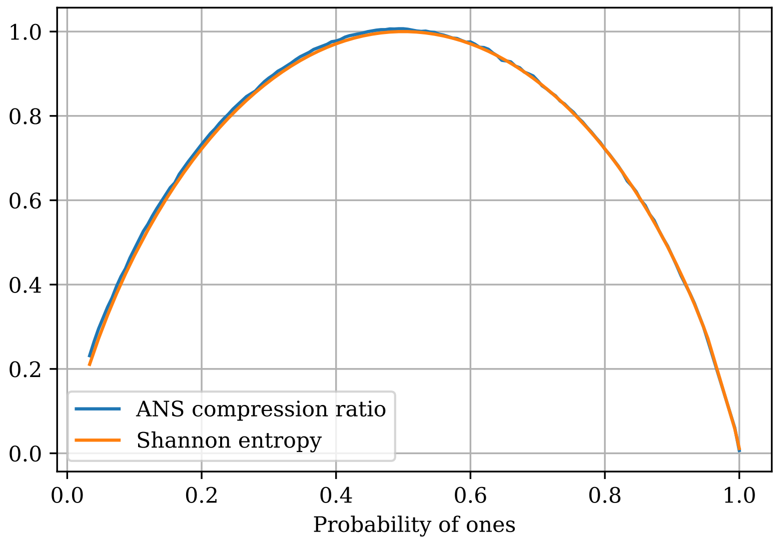


Figure 40: Plot of mean compression ratio and Shannon entropy as function of probability of ones of the testbench

Figure 41 displays both min, max and mean compression ratios of the dataset. The 17.78% of the cases where ANs compressor surpassed the Shannon entropy is seen as the orange curve occasionally undercutting the Shannon entropy. Interestingly, the cases where the ANS compressor outperformed the Shannon entropy are most notable in the region where the probability is greater than 50%. The absolute distance between the Shannon entropy and the ANS compressors' best performing samples also decreases in relation to the increase of the probability. The relative distance, however, is largely unaltered until the extremes of the probability are reached (i.e. under 10% and over 90% probability). This relationship is seen in figure 42.

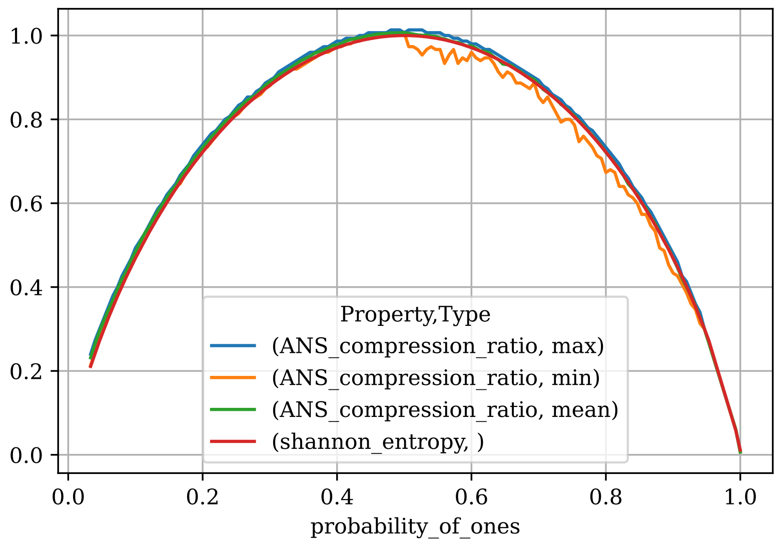


Figure 41: Plot of mean, max and min compression ratio as function of probability of ones of the testbench

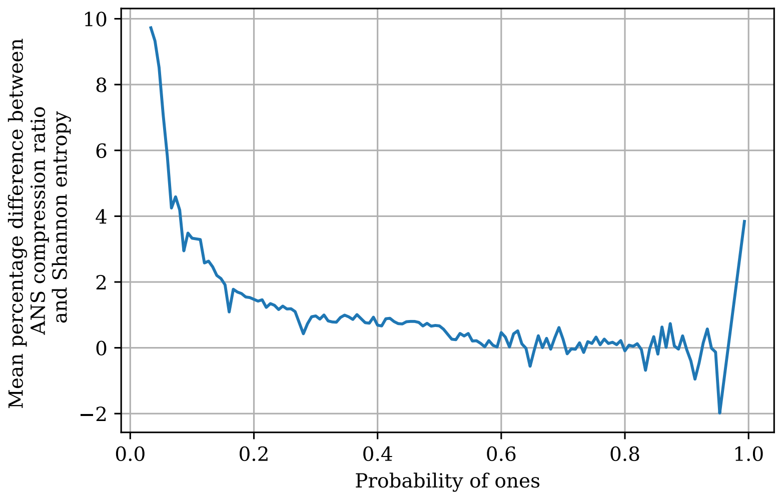


Figure 42: Mean percentage difference between compression ration and Shannon entropy per probability of ones

4.3.5.3 Compression behavior with probability deviations

The master testbench also generated a multitude of samples with forced probability deviations. These deviations are seen as increased or decreased probabilities, which purposefully decreases the accuracy of the probability modelling by +-20%, as presented in section 15. In total, the sample pool is based on 224.683 samples, which translates into 33.702.450 encode and decode operations (for a total of 67.404.900). The value of forcefully offsetting the probability of a sample is that it explores how inaccurate the probability modelling may be. The criteria are that the ANS compressor must still be able to produce an accurate and unambiguous state which is correctly decodable. The challenge with compressing samples with highly inaccurate probability modelling is that the state grows more rapidly, and if the state grows too wide for the ANS compressor to handle, then it results in a failed compression process. Table 41 visualizes the total number of samples at their respective probability deviation. Some samples with probability deviations of ±20%fail and are filtered out from further analysis. This indicates that based on this analysis, the compressor may safely encode and decode states with probability deviations of up to ±10%. This conclusion coincides with that of section 15. These results are also visualized in figure 43.

Table 41: Number of samples with successful and faulty compression and/or compression ratio higher or lower the Sharon entropy per deviation of probability of ones

Deviation	Value	Successful compression	Compression ratio > Shannon Entropy
-0.20	True	21705	21726
	False	3078	3057
-0.10	True	24853	24680
	False	0	173
-0.05	True	24963	23766
	False	0	1197
-0.01	True	25112	22281
	False	0	2831
0.00	True	24980	22067
	False	0	2913
0.01	True	24720	22198
	False	0	2522
0.05	True	25518	25075
	False	0	443
0.10	True	25016	24983
	False	0	33
0.20	True	23704	23798
	False	1034	940

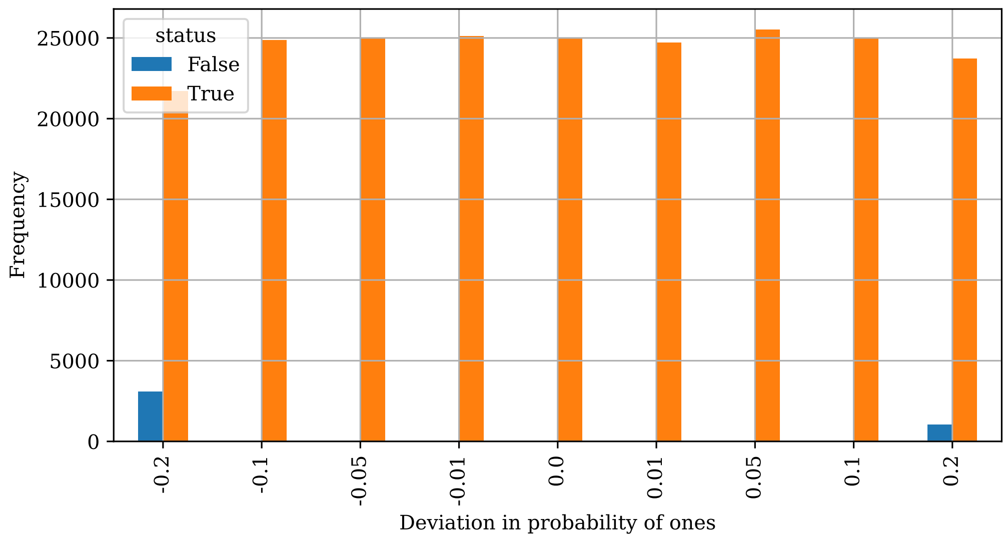


Figure 43: Number of samples per deviation where compression ratio higher (true) or lower (false) than Sharon entropy per deviation of probability of ones

Table 42: Number of samples after removing samples with faulty compression

Deviation in probability of ones	Count
-20%	21705
-10%	24853
-5%	24963
-1%	25112
0%	24980
1%	24720
5%	25518
10%	25016
20%	23704

Table 42 visualizes the number of samples that are left after purging the failed compressions at probabilities ±20% from the dataset.

Figure 44 shows the frequency of the samples of their given probability. It is seen that the samples are roughly equally distributed as for the tests without probability deviations, as seen in figure 36.

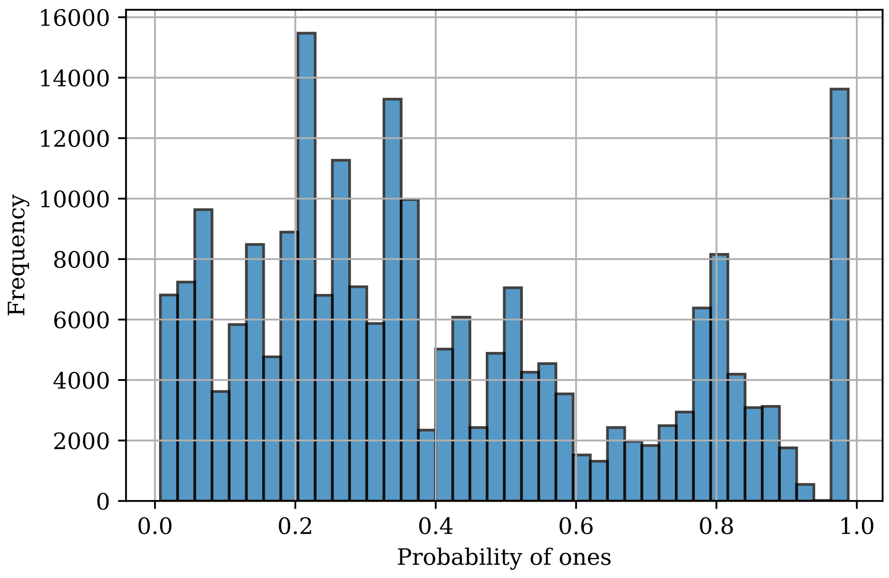


Figure 44: Histogram of probabilities of ones

Figure 45 illustrates the relationship between mean compression ratio and the deviation of probability. It is directly observable that inaccurate probability modelling leads to worse compression ratios and therefore worse compression performance. The compression ratios increase significantly (worsens) as the accuracy of the probability modelling decreases. The compression performance at +-20% is significantly worse. The relationship is further accentuated in figure 46 which shows the mean compression ratio difference to the Shannon Entropy. These ratios appear rather steep as they include the very edges of the probability spectrum (e.g. ratios of less than 99% and more than 1%). It becomes increasingly difficult to compress data with ANS as these extreme probabilities of the one percentile are encountered.

Figure 47 confirms that the sample set was given roughly equally sized stimuli on average.

Figure 48 plots the average compression ratios of the dataset against their probabilities. The graph visualizes the probability deviations which are especially significant around the probabilities of 25-75%. In these ranges, size increases of up to 18% are seen. Given a 150 bit sample, this translates to an encoded state consuming 177 bits, which is a significant size increase and completely defeats the purpose of data compression. It is however also noticeable how probability deviations of ±1-5%

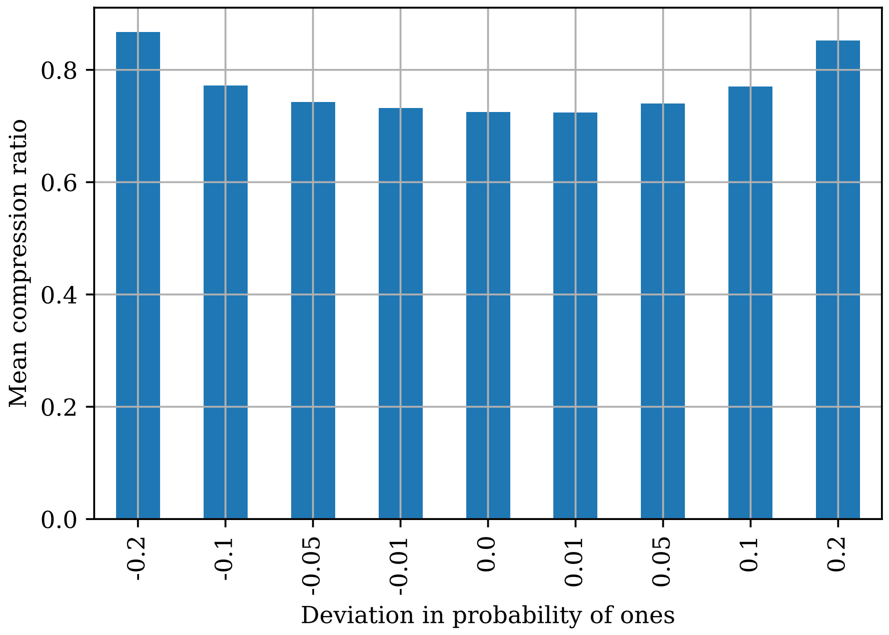


Figure 45: Mean compression ratio per deviation of probability

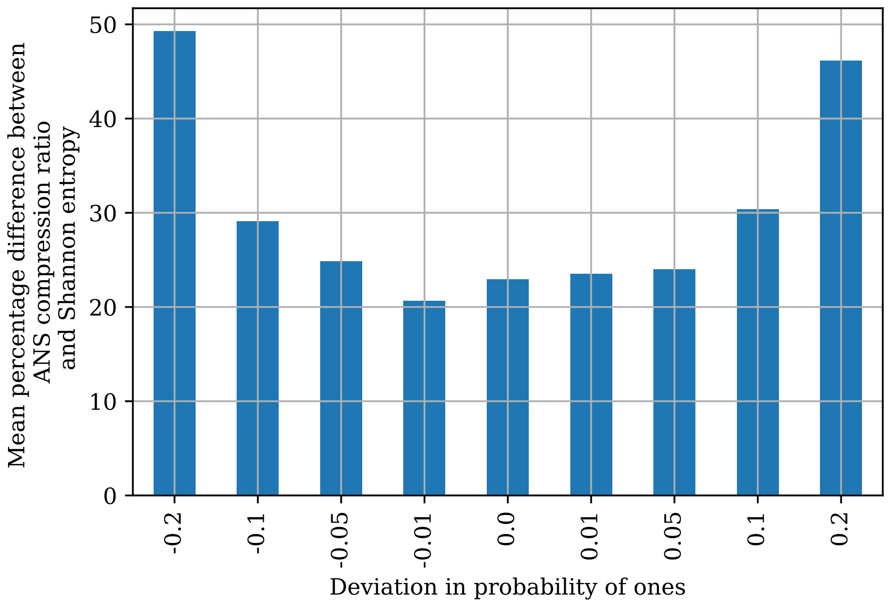


Figure 46: Mean compression ratio difference to Shannon entropy per deviation of probability

results in almost insignificant deterioration of the compression ratios, whereas ±10% results in a marginal deterioration of compression ratios. All these findings indicate that the compressor is able to comfortably handle data sets with probability modelling inaccuracies of ±1-5%, and may handle datasets with up to ±10% inaccuracy in probability modelling safely, although with deteriorated performance.

Figure 49 visualizes how samples with probability deviation still are able to outperform the average theoretical limit at select samples.

An overview of the compression ratios distance to the Shannon entropy is found in figure 50. Different perspectives are offered in figure 51 and 52. Lastly, figure 53 visualizes how the stimuli relates to the probability at which it was modeled including the forced deviations of probability of

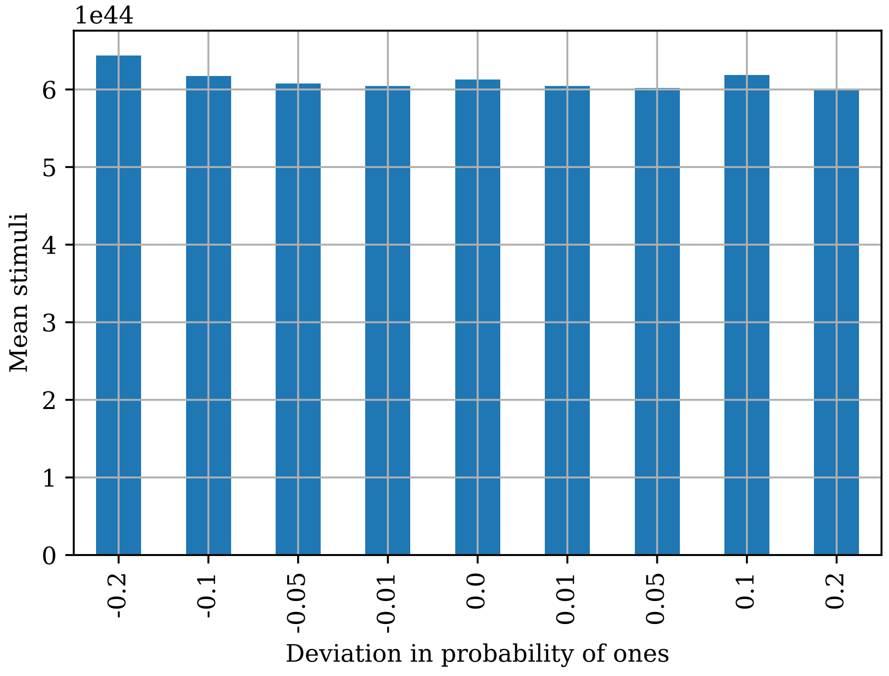


Figure 47: Mean stimuli per deviation of probability

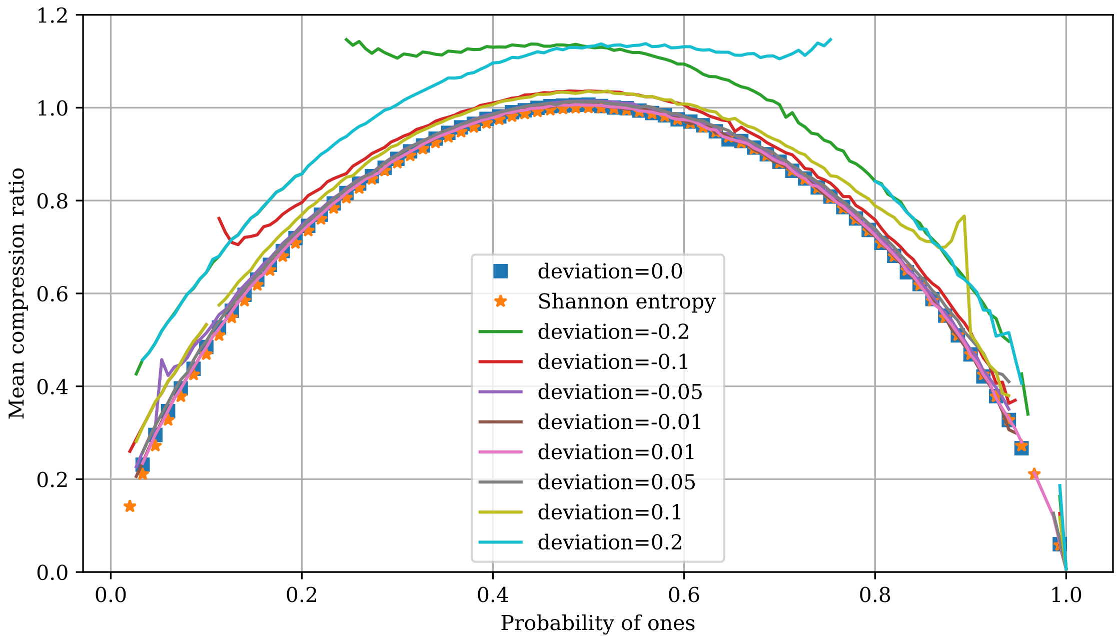


Figure 48: Mean compression ration as function of deviation of probability and probability of ones

ones.

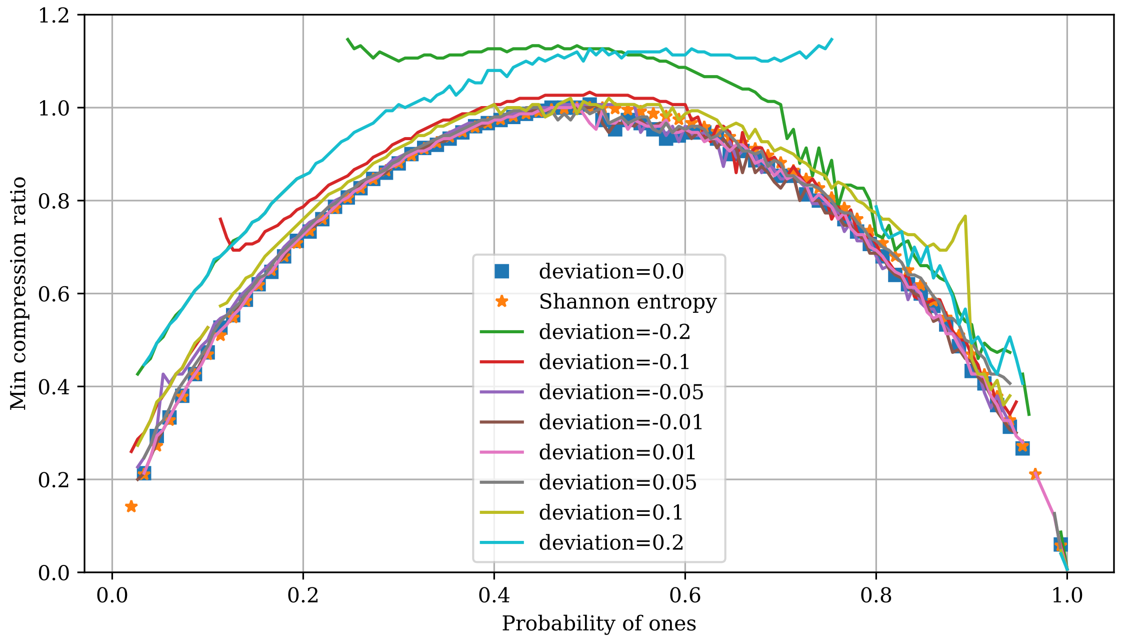


Figure 49: Min compression ration as function of deviation of probability and probability of ones

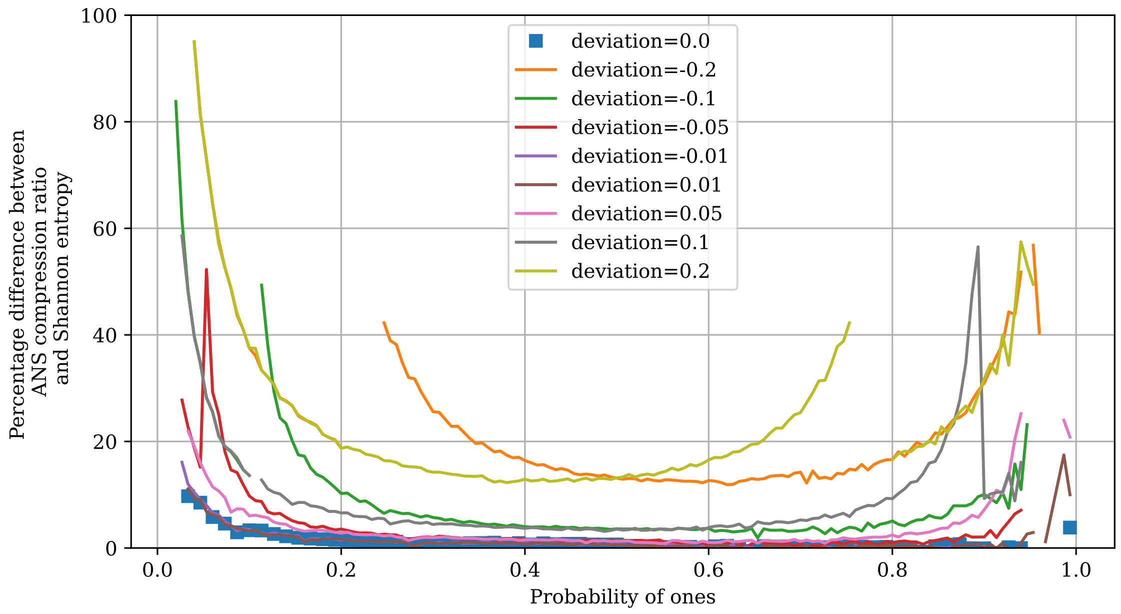


Figure 50: Mean percentage difference between compression ratio and Shannon entropy as a function of probability of ones and deviation in probability of ones

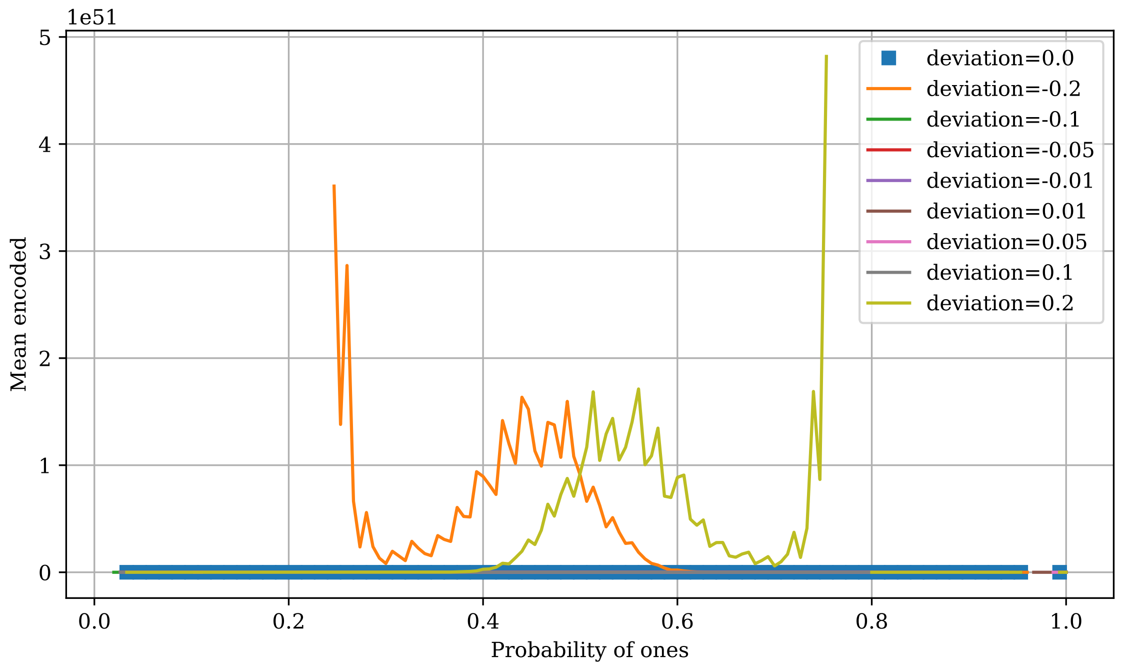


Figure 51: Mean encoded value as a function of probability of ones and deviation in probability of ones

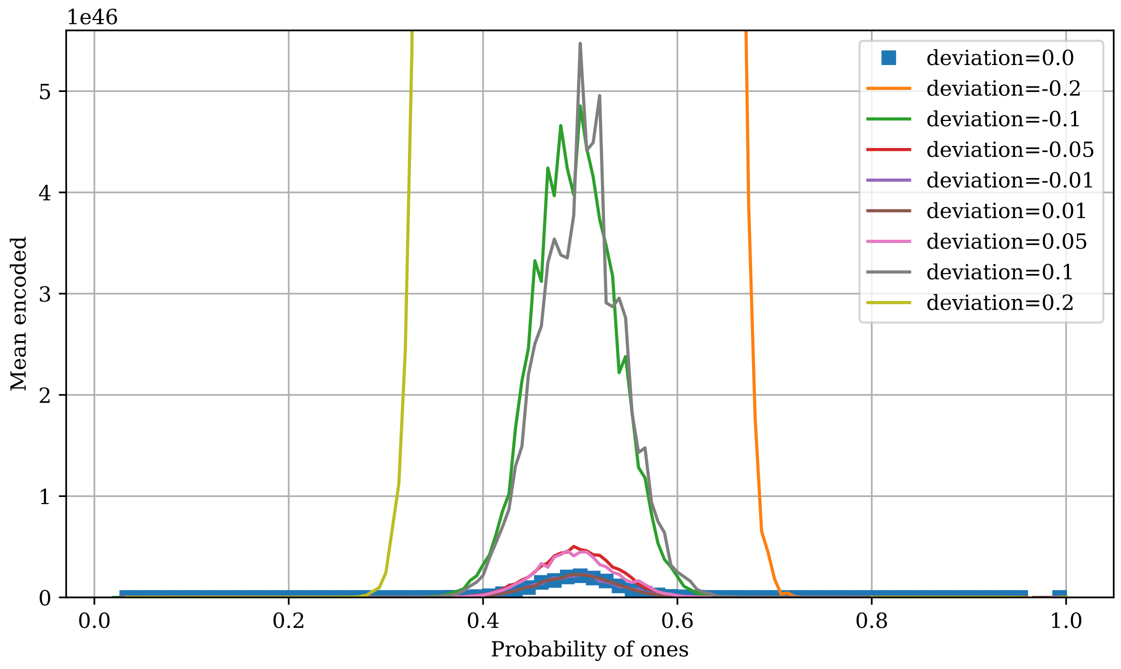


Figure 52: Mean encoded value as a function of probability of ones and deviation in probability of ones removing encoded values larger than $0.56 * 10^{47}$

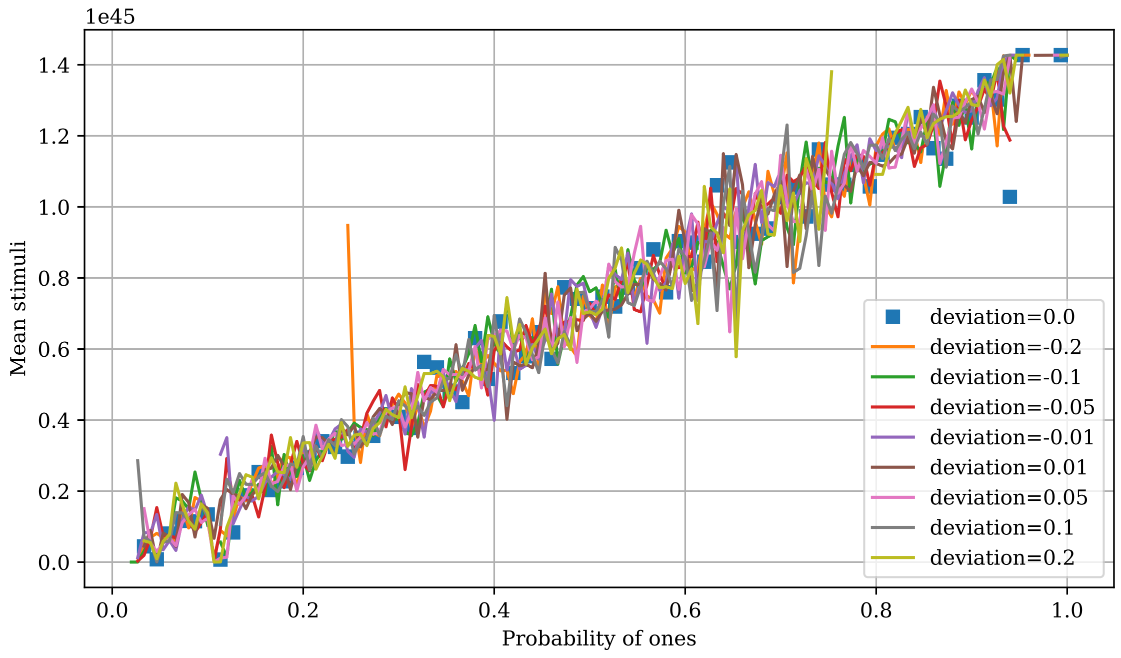


Figure 53: Mean stimuli value as a function of probability of ones and deviation in probability of ones

4.3.6 RTL analysis schematics

The RTL diagrams are created using the RTL Analysis tool in vivado. Firstly a top-level diagram is shown, following more detailed diagrams. The diagrams are generated with the default synthesis and implementation parameters.

4.3.6.1 Decoder

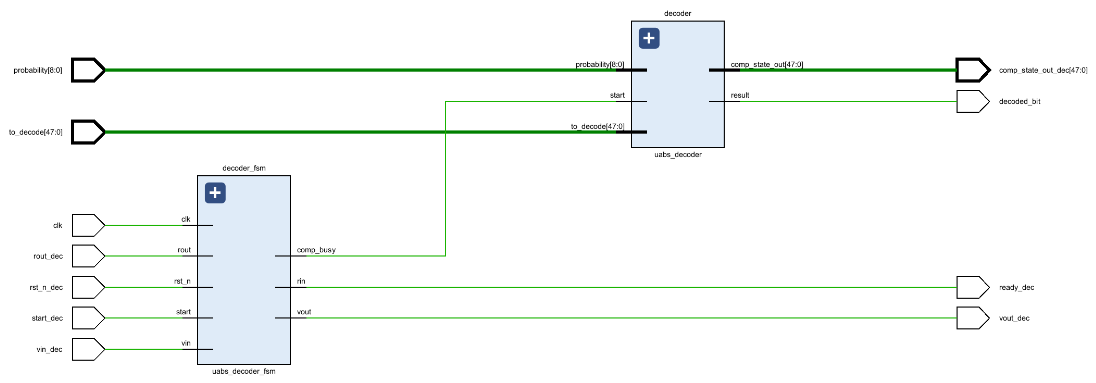


Figure 54: Top level Schematic from RTL Analysis of the optimized decoder circuit

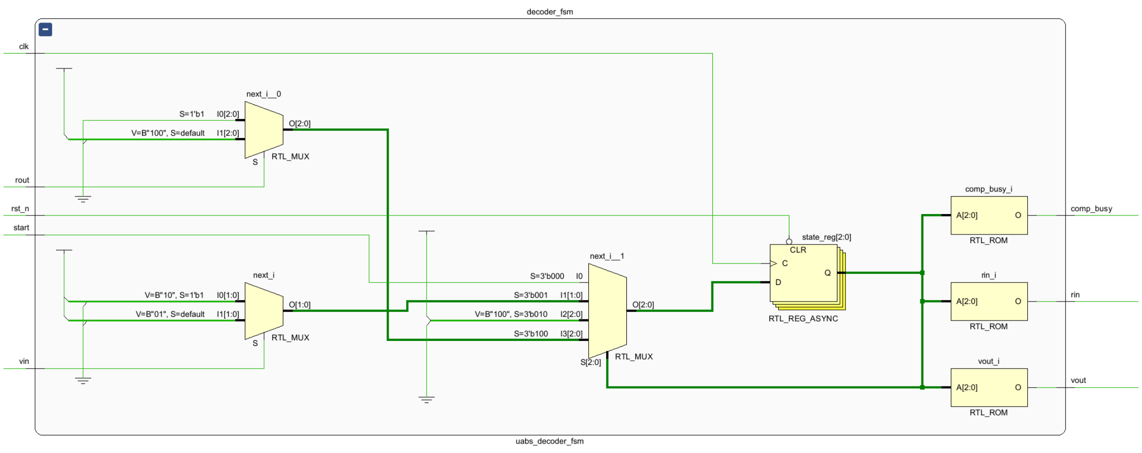


Figure 55: Finite State Machine Schematic from RTL Analysis of the optimized decoder circuit

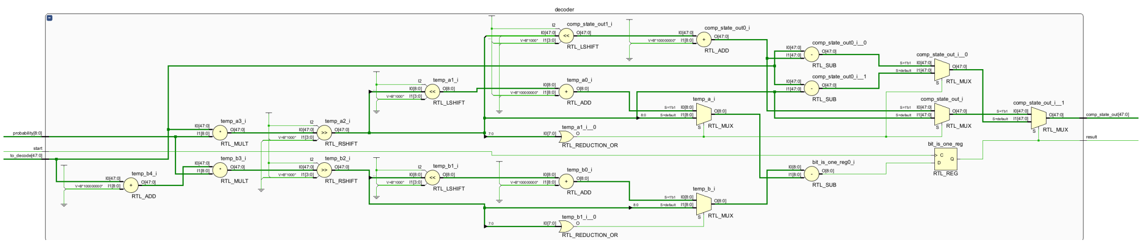


Figure 56: Encoder arithmetic Schematic from RTL Analysis of the optimized decoder circuit

4.3.6.2 Encoder

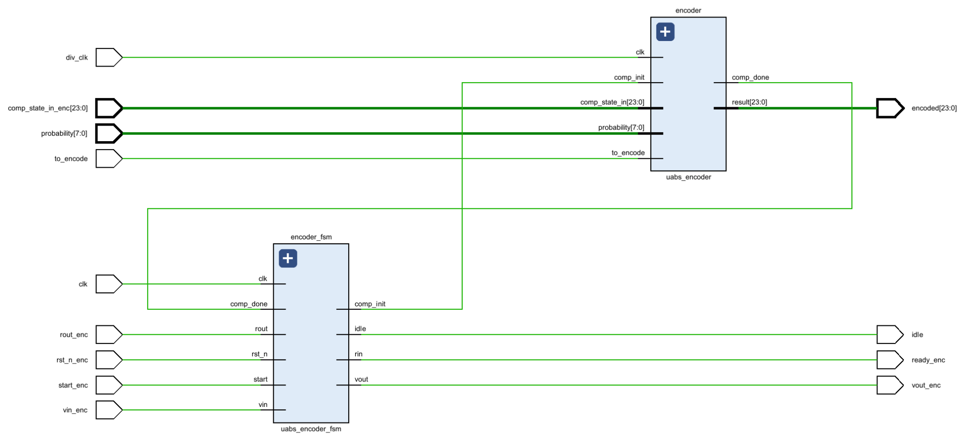


Figure 57: Top level Schematic from RTL Analysis of the optimized encoder circuit

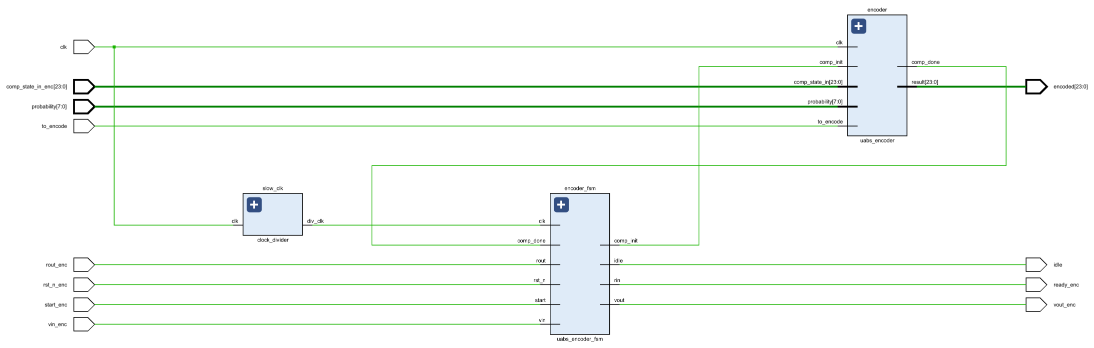


Figure 58: Encoder arithmetic Schematic from RTL Analysis of the optimized encoder circuit with clock divider circuit

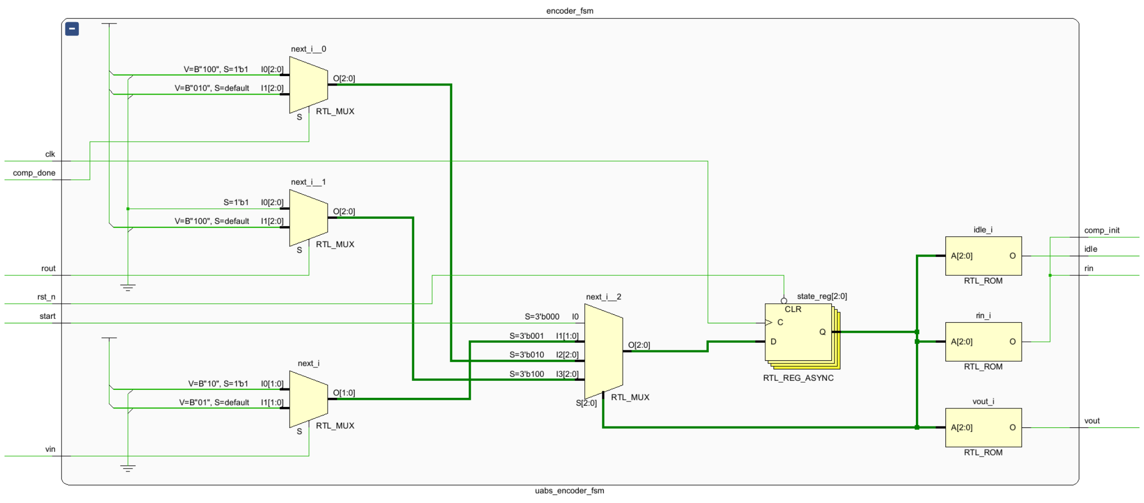


Figure 59: Finite State Machine Schematic from RTL Analysis of the optimized encoder circuit

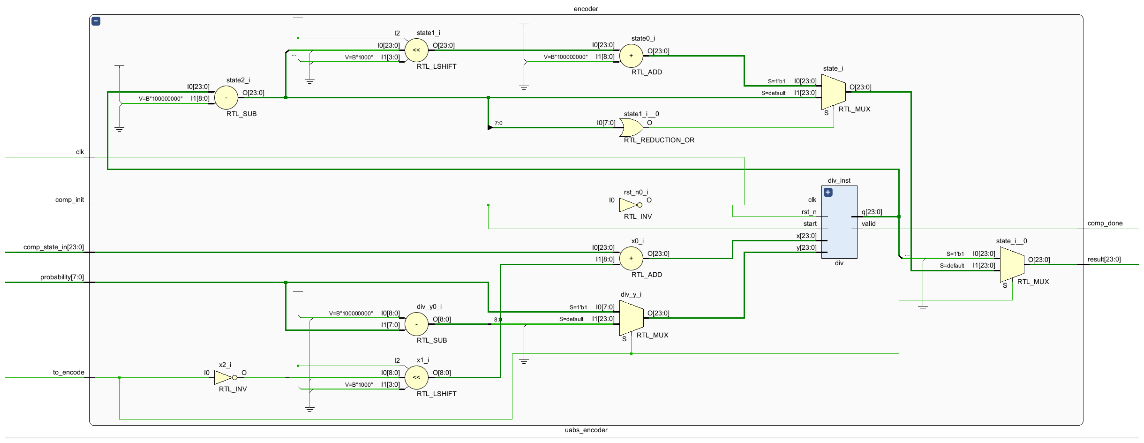


Figure 60: Encoder arithmetic Schematic from RTL Analysis of the optimized encoder circuit

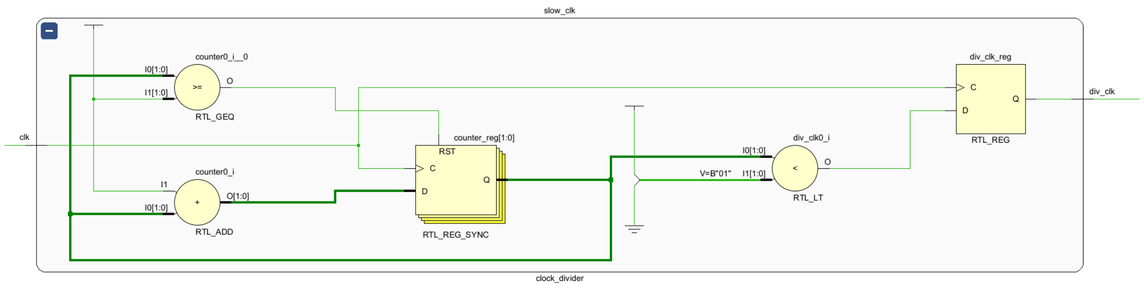


Figure 61: Encoder Slow Clock Schematic from RTL Analysis of the optimized encoder circuit. Given clock division of frequency/4 and duty cycle of 1/4. These are parameters in the module instantiation and could easily be tweaked by the designer.

4.4 Unoptimized ANS compressor hardware implementation

The unoptimized ANS compressor performs similarly to the optimized ANS compressor when correctness and compression ratios are estimated. It does however perform inferior to the optimized implementation when synthesis and implementation is estimated. Due to similar or inferior results being reported, it is deemed undesirable for neither rollout nor further analysis. It is therefore kept as a reference model and archived as a previous build. It does however hold value as a research object, as it has some features that are convenient for testing and prototyping. Examples are an abundance of I/O, which are convenient for prototyping and experimental development. It is also closer to the software model's structure and easier to understand for developers picking up the project. As compared to the optimized implementation, the RTL is of a higher abstraction level. The logic behind the optimized implementations RTL is obscure, as it to a lesser extent resembles the original algorithm's structure.

4.4.1 RTL Analysis schematics

An overview of the unoptimized implementation is supplied by the RTL analysis. It is noticeable how the unoptimized circuitry of for instance the decoder in figure 62 is using much more I/O than the optimized Decoder at figure 54. Much of the I/O was deemed unnecessary or unimportant and was optimized away. Less I/O leads to less interfacing, less code and circuitry to verify and less power consumption. However, it may come at the trade-off of less features and insight from interfacing modules, such as the encoder's previous overflow and divide by zero flags, which might be desired by designers.

4.4.1.1 Decoder

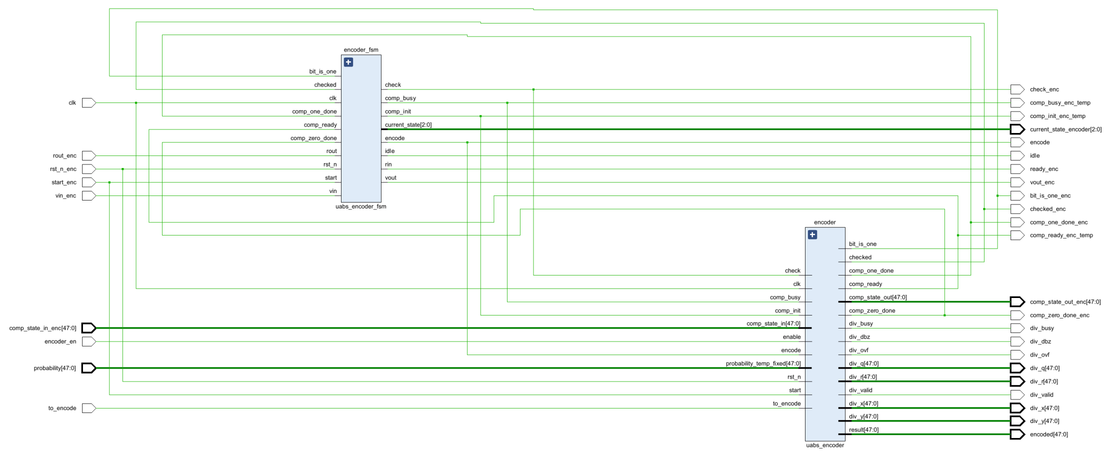


Figure 62: Top level Schematic from RTL Analysis of the decoder circuit

4.4.1.2 Encoder

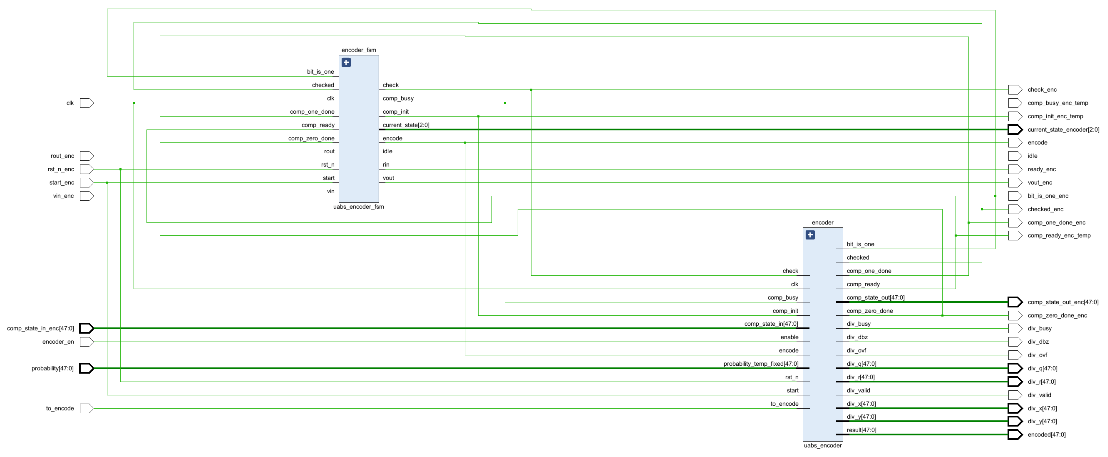


Figure 63: Top level Schematic from RTL Analysis of the encoder circuit

4.5 Software models

Similarly to the role the unoptimized Hardware implementation plays, the software models are dominantly seen as resources for the designer, and not products for rollout. This is further discussed in section 5.15. Their performance is in essence unimportant, beyond being functionally correct in behavior. And their most important contribution to the results is largely bridging the gap between algorithm and hardware circuit. They are verified to function for a handful of stimuli, which have been manually confirmed using for instance the calculated examples provided in section 2.2. They are thus lightly verified, however proven to work for the few test cases they are exposed to through development. The result section will therefore not obsessively present their performance and formally prove their correctness, as it is beyond the scope of this thesis.

4.5.1 C++ software model

Table 43 displays a set of decode and encode operations that were run. The tests results showcases successful compression behavior, as all encode and decode operations succeeded. More tests was ran during development, however was not stored.

Table 43: C++ software model example of test runs and their status

Stimuli.txt	Decoder.txt	Compression status
10000010	10000010	Success
11001100	11001100	Success
10101010	10101010	Success
11001100	11001100	Success
10000000	10000000	Success
00110011	00110011	Success
11000000	11000000	Success
11111111	11111111	Success
10000000	10000000	Success

4.5.2 SystemVerilog software model

Table 44 displays similar results to those of table 43. Both software models were able to decode and encode successfully the same stimuli. While it is not a thorough verification setup, it does bring some confidence in their correct behavior and modelling.

Table 44: C++ software model example of test runs and their status

Stimuli	Decoded	Compression status
10000010	10000010	Success
11001100	11001100	Success
10101010	10101010	Success
11001100	11001100	Success
10000000	10000000	Success
00110011	00110011	Success
11000000	11000000	Success
11111111	11111111	Success
10000000	10000000	Success

Waveform 64 displays one encoding and decoding operation. The compressor encodes the 8-bit string 1000_0000, which is shown as 128 in decimal in the simulation at to_encode. The encoding takes place using probability 0.125, which results in the encoded binary representation 11001, which is denoted as the decimal number 27. The original representation is reproduced at the following clock cycle, displaying how the compression process was carried out successfully. The process took two clock cycles, one for encoding and one for decoding (which is only obtained in simulation and is not realistic in hardware). This software model does not translate into actual hardware and is purely illustrational for modelling purposes.

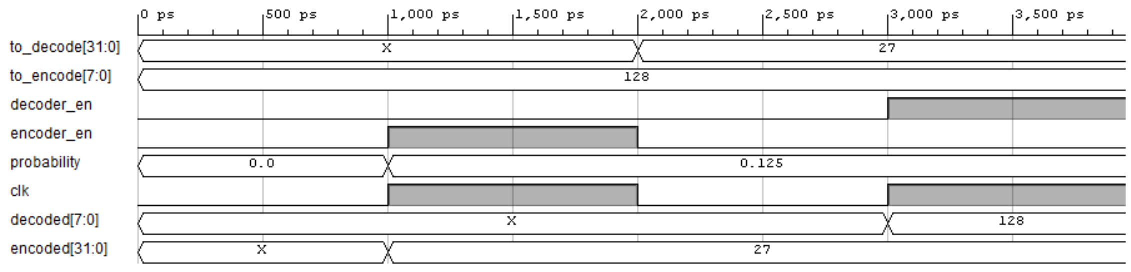


Figure 64: Top level Schematic from RTL Analysis of the encoder circuit

5 Discussion

5.1 Overall evaluation

One of the most challenging aspects of developing a hardware implementation of the uABS variant of the ANS algorithm is the lack of similar works readily available online. This forces the designer to start from scratch, and results in a lot of development time invested in experimentation. This forces the designer to create and compare multiple builds of the compressors against each other, as there is not necessarily initially a clear understanding of how the development process should take place, in order to make a respectably performing circuit. This is not to say that there is a total lack of references to other compressors already implemented in hardware from other algorithms, or of literature outlining how good hardware is made. It just means that the process becomes more tedious and time consuming.

Nevertheless, a well performing hardware implementation has been proposed in this project, namely the optimized uABS variant of the ANS compressor. Alongside it are a less optimized, but still operational unoptimized compressor and two software models. The synthesis data appears to be valid, and both timing analysis and a rather thorough verification processes has passed with good results. More measures could have been added to the analysis, larger data sets that could have been produced, and more corner cases could have been verified. Nonetheless, based on the present reports, it should be within fair reason to confidently propose the uABS variant of ANS as a potent algorithm of choice for hardware implementation.

5.2 Evaluation of the compression behavior

The compression behavior presents a highly satisfactory compression ratio, with a relatively high tolerance for inaccuracy in the probability modeling. It is of course bound by the theoretical limit set by the Shannon entropy on average, still approaching this closely and of course occasionally better than this as well.

No clear rule of thumb was found for which cases would lead to compression ratios exceeding the theoretical average limit. However, it appears intriguing to try to identify and characterize these cases. This could conceivably be used to aid algorithmic modifications to improve the compression ratio, or at the very least bring a better understanding of the nature of such cases.

Is is also possible to combine the ANS compressor with other compressor in a compression chain. This way a compressor, such as the pattern driven LZ77 could find low hanging fruits in the dataset, and then allow the ANS compressor to further compress the dataset. This could pose a potent combination of compressors, as the former compresses based on properties such as pattern and repetition, whereas the ANS compressor compresses based on frequencies of occurrence. In terms of redundancy in the dataset, these two compressors contribute based on two different strategies. An example of a common combination of other compression schemes, is using Huffman and LZ77 together, with Huffman coding as a common entropy coding technique, with much familiarity to the ANS compression scheme.

5.3 Evaluation of compression rates

The encoder operates at a significantly slower rate due to it's fixed division module, compared to the decoder, as seen in table 36 compared to table 38. The speed at which it operates is almost exclusively proportional to the number of bits (WIDTH + FBITS) the circuit is configured to, as the division module consumes the majority of clock cycles of the total run time for any reasonable bit widths.

Because of this property, the module was given a slow and fast clock, since the control logic itself made up only a handful of clock cycles of the overall runtime. This does also marginally increases the runtime, and could be removed if peak performance is desired. This is conveniently achieved by changing the parameters in the code of the module instantiating.

5.4 Cases of errors in compression

There are no cases of compression errors within the defined limitations of the compressor, however outside of these limitations faulty compression is to be expected. Circuitry could potentially have been designed to prevent these cases, or alert the user. The ANS compressor might raise a flag if an internal error is detected. This could be implemented as a few extra signals and some peripheral logic, most likely with not noticeable decrease in the performance of the compressor. The division module has already a divide by zero and overflow flag, which is internal to the Encoder module. These flags are not properly represented in the modules I/O, and could be placed at the ports if the designer has spare ports on it's target FPGA. Alternatively, if the ANS compressor is part of a larger system, it would come at almost no cost to include these warning flags internally. Having a flag indicate to the control logic that a faulty compression has occurred, would often prove useful in the case of the compressor being connected to a stream of continuous data.

5.5 Effect of BIT_WIDTH and FBITS

The WIDTH and FBITS parameters have a very direct impact on the circuitry as a whole. Larger parameter sizes enables it to accept larger stimuli, utilize probabilities with finer granularity as well as to generate a larger circuit. It is also worth to note that the WIDTH of many of the internal decoder are defined as WIDTH * 2. These are thus twice as wide as the general circuitry width. This was carried out as a convenience to reduce the number of overflow incidents at high probability deviations (i.e. highly inaccurately modelled probability).

It is furthermore important that the designer adjusts these parameters when the circuit is used in a larger system. A rule of thumb is that the WIDTH parameter of the circuit must be around 20-25% greater than the number of bits the circuit shall encode prior to renormalization of the state. Aggressive renormalization of the states will allow the circuit to operate with a low WIDTH. This was evident in the the UVM testbench, where random states were combined with random probabilities and random input bits without issues using a WIDTH of only 24 and FBITS merely 4. A WIDTH of 180 was used for the majority of the simulations, but was deemed largely unnecessary for a real world implementation, and only used for research such as assessment into precision and similar factors.

Lastly it is important to understand the effect of the number of FBITS on assessing the accuracy of the probabilities being captured. It may serve as a simple illustration to point out that whereas a 4-bit implementation basically allows 16 distinct probabilities to be captured, a 8-bit implementation allows 256. The difference between allowing 16 and 256 different probabilities heavily affects the compressors ability to produce competitive compression ratios, as was presented in section 4.3.5.3, and thus illustrating that a decent probability resolution is confirmed to be necessary.

5.6 Evaluation of FSMs

The FSMs are well performant and is deemed as having broadly speaking, little potential for optimizations, when seen as an isolated system. It is certainly, however, heavily optimizable if seen as part of a larger system. At the moment, the optimized hardware implementation are based around a four or five state FSM. These states each require at least one clock cycle to transition through, and there are bottlenecks especially for the Decoder. If these states were reduced from for instance 4, to 2 then there would be a speedup of 100%, bringing even more convincing decoding speeds. As seen in table 45. Being able to decode 82.032 Mb/s puts the decoder at a decent rate even with the at 168Mhz.

Theoretically speaking, if a more modern technology was employed, and the Decoder was optimized for it, decoding speeds could easily reach 100-200Mb/s if a two state FSM was employed. If combined with the FIFO on the Decoder/Encoders outputs it should still be able to comfortably interface with other modules if part of a larger system. Lastly, it could also be possible to implement a stateless control logic, this however has not been further researched and remains a topic of conversation for the future.

Table 45: Theoretical peak decoding speed for optimized compressor assuming a 2-state FSM

Board	Cycles per bit [b]	Clock Frequncy [MHz]	Peak Decoding speed [Mb/s]
Zynq UltraScale+ ZCU106	2.00	168.067	82.032
Spartan-7-SP701	2.00	103.627	51.812

5.7 Interface

The interfaces of the optimized ANS compressor and unoptimized ANS compressor represent two different design philosophies. The optimized ANS compressor uses resources sparingly, as few I/O ports as was deemed suitable, while the unoptimized ANS compressor has a more generous interface. During the development however, it was quickly seen that some I/O should have been kept in the optimized Encoder. Not only the previously mentioned divide by zero and overflow flags in inside the Encoder as mentioned in section 5.4, but also more enable signals on the top modules input. These enable signals would be handy for completely turning off the compression modules when desired, to save power.

It is observable during for instance verification, how some outputs will switch when new stimuli are applied, despite the compression process not being started. This is not technically an issue, as the output of the compressors is only regarded correct during the valid out flag. This is only visible during inspection of waveforms, but does however lead to unnecessary noise inside the circuitry and wasted energy. The unoptimized ANS compressor have enable signals remedying this issue, and it appears to be a needed feature in the optimized ANS compressor. It was however removed to push some extra marginal performance out of the system, which turned out to be an improvement which was negligible at best.

5.8 Evaluation of the Ready/Valid interface

The Read/Valid interface performed largely according to expectation, and proved a valuable design inclusion. It generally well regarded in the industry, and complemented the light ANS compressor. Other interfacing strategies might have tested, however there are other areas of improvements that might be more worthwhile to explore.

One common usage of the Ready/Valid interface is to use it with a FIFO. In this case the compression result are stored inside a FIFO, used in conjunction with the Ready/Valid interface. This would be a welcome design addition to the ANS Compressor, since the interface can be regarded as a FIFO interface.

5.9 Clock gating and further power saving

There might also be sections of the circuit that could be gated off during periods of time where operation is not required. Power consumption has in general not been an issue, as the circuitry is relatively lightweight, but is anyhow an area of interest always looking for improvements. Despite no signals being toggled, both digital and analog logic do consume static power. The static consumption is generally speaking more difficult to reduce. Factors such as subthreshold leakage and gate direct-tunneling can be dominant, depending on the technology employed. The static consumption is of course reducible on a hardware level, but is outside the scope of this thesis. The dynamic power is on the other hand more easily affected by the RTL design.

The arithmetic units could for instance have been completely gated off when not in use. The decoder is somewhat wasteful in terms of activity factor as the combinatorial logic is very responsive and always ready, at the cost of calculating temporary constants even when the compressor is not prompted to compress (i.e. receiving start and input valid). There should be multiple solutions available maintaining the high performance of the decoder (such as one or multiple enable signals) gating off the arithmetic, while not deteriorating it's performance. The only real trade off would be a higher utilization of FPGA resources (area), which are already respectably low and could easily be expanded. It is seen that the dominantly combinatorial circuitry of the decoder in figure 32 is more expensive than the more power friendly encoder 33. Lastly it is worth to mention that both the Zynq UltraScale and the Spartan-7 are both more than well equipped to handle the ANS compressor. They might be seen as overqualified, and the circuit could easily be implemented on much cheaper FPGA, unless using those resources (such as the ARM cores) giving an advantage to the total application of the compressor.

5.10 Effect of chosen FPGA on performance, utilization and power results

The choice of FPGA greatly affects the performance results, utilization and power consumption. New technology enables higher clock frequencies, more optimized circuitry and new techniques for reduced power consumption. It is also worth to notice that better synthesis tools enables better insight and greedier synthesis specific optimizations.

5.11 Evaluation of testbench

A lot of time was spent on verification, which also appears to be the trend in the industry at the moment. The verification gap is hard to close, especially for a one man development team. Nonetheless, a handful of testbenches was developed and used as part of the verification strategy. Testbenches have the convenient feature of allowing the designer to peak inside the designs, both to better understand them, but also to eliminate bugs as early as possible in the development cycles.

Testbenches were made for every individual module during development, as it can be difficult to verify all properties of a system purely from a top module of a larger system. There are states unreachable during a top level simulation, that might not be noticeable under normal operation. They could however, be present in corner cases and leave bugs in the system which are difficult to track down, as discovering the entire state space is difficult. Likewise, there will always be real possibilities of bugs in the system. However, what can be said with confidence, is that the current verification reports indicates that under normal operation, the ANS compressor appears to perform without compression/decompression errors, and with high compression ratios and acceptable performance.

As was apparent in graph 44, producing perfectly balanced, truly random samples for simulation is difficult. The sample set contained almost 225.000 samples of 150 bits, and still could not cover all possible input states. Exhaustive verification, in the sense of testing all possible inputs and producing all possible outputs at high design WIDTHs seems to be nearly impossible without a sufficiently large team of verification engineers and top of the line highly automated commercial simulators. What is possible however, and was performed is adjusting the design parameters to be sufficiently limited, yet sufficient for the purpose at hand. This allowed the design to cover most possible input combinations and produce the most important of the possible output combinations, as presented in section 14.

There are many aspects that could have been added to the UVM testbench, as well as the master testbench. For instance, the SystemVerilog reference model could have been added using the already pre-configured channels in the UVM testbench. It could thus be invoked in the scoreboard and used as yet another criteria for a packet to pass. Furthermore assertions could have been used more frequently inside the UVM testbench, as well as in the design itself. It is commonly regarded a good design practice for the RTL designer to declare assertions directly inside the design code, or inside external files, during development. This way some of the verification work is done early in the design process, and it also adds more confidence overall in the proper behavior of the design.

Furthermore different packets could have been declared, in combination with different sequences. Almost all terminal printouts were commented, and thus effectively removed from the code due to the restrictions on EDA playground, but certainly much of this could (and should) be brought back. It is a verification environment after all, and should provide useful information to the verification engineer. Lastly more could have been done in testing the interface, more randomness could have created interesting corner cases potentially finding unexplored areas of the statespace that has not yet been considered, which might still exist.

As an alternative or complement to UVM testbenches, formal verification could also have been utilized. Formal verification has some advantages over UVM testbench for certain use cases. Formal verification is a form of functional verification and uses static analysis to prove design functionality. The verification engineer is for instance not required to create stimuli as it is handled by the tool itself (e.g. JasperGold). The designer may however still constrain it. The utilization of assertions in tools such as JasperGold allows the designer to confidently carry out equivalence checking. Conversely, it is important to acknowledge that the formal verification tool might struggle with producing large amounts of stimuli, at times even more so than the the UVM testbench does. Discovering sequentially deep and large state spaces with formal verification could turn into a daunting task. The best solution would be to combine UVM testbenches and formal verification and play to their respective advantages.

5.12 Issues during simulation

There were some major challenges during simulation, which if solved, would have allowed even greater sample sets to be produced. Whether that would be necessary and would lead to other conclusions is another question.

Vivado was used as the EDA of choice with a local installation. Vivado has some convenient features, but does however produce a decent amount of dump during long simulations, with the longest run at roughly 15 hours. At that point dump files exceeded 100GB of storage, and was not only placed inside the work folder but probably also stored and or recycled in other places inside Windows. Nonetheless, this lead to many simulations crashing due to limited storage space on the development computer. This led to many overnight simulations crashing, and also due to other unknown unexpected issues. It is believed that the sample set could have been anywhere from 50-200% larger if not for these issues.

Another limitation was the run-time limit of one minute at EDA Playground. Sadly, the final build of the UVM Testbench had issues running on Vivado, due to unknown issues. Some error messages were produced, however could not be identified. Such as ERROR: [VRFC 10-2984]. It is of course possible to identify these issues without using the error messages, however, it proved tedious and after some hours of debugging, the hunt for a solution was abandoned. A temporary solution was to run the testbench on EDA Playground, which has commercial simulators (such as Cadence Xcelium and Aldec Riviera-PRO). These simulators successfully ran the UVM testbench and did not reveal any underlying errors which might have resulted in the failure of simulation in Vivado. Due to these unfortunate circumstances, the UVM testbench was restricted to 5000 terminal printouts and 1 minute of run-time per simulation execution.

5.13 Analysis of RTL views

The RTL views of the ANS compressor, as presented in section 4.3.6 revealed how the RTL was interpreted by Vivado. It is largely unsurprising and brings a convenient visual perspective of the RTL. The circuits contain the expected logic blocks, and neatly brings the circuit into a schematic. The inferred logic appears to be present as intended when designing the RTL, which brings confidence to the design process. It furthermore brings confidence in the uABS variant of ANS as a suitable hardware implementable algorithm, as it clearly displays how well the algorithm is suited for hardware.

5.14 The ANS compressor as part of a larger system.

The ANS compressor could be employed in a multitude of systems. It has a compact implementation, which allows it to fit in small IoT or even IoTT systems. The compressor is also boasting compression rates approaching the theoretical limit for entropy based compressors and is therefore also useful where chip size is not an issue, or where it can be embedded MPSoC- or even ASCIC-wise. The decoder and encoder both operate in the Mb/s range, whereas the decoder is potentially fast enough to decode video, and could be used for instance as a link from action cameras or security systems. The restrictive factor is how large the required data rates are, as the decoder must be able to output enough pixels to fill the monitors with sufficient frames per second. As a reference full HD video can have a bitstream of roughly 3,5Mb/s to 5,000 Mb/s, however it is important to note that video in real world use cases often is heavily compressed to lossy formats. However, still could be gaining an extra compression factor as experienced by the large companies adding ANS on top of existing solutions such as LZ compression (Apple etc.).

The most important issue to solve for the systems designer when implementing the ANS compressor as part of a larger system is good probability modelling. An example of the control logic of the encoder and decoder compressing video data is shown in figure 65 and 66 encoding and decoding video data. The control logic must handle data streams, estimate the probability and transmit the data from the encoder to the decoder. Examples exist in areas such as real time transmission of video from a drone, where there must be transmitters, receivers and so on. A challenge however with implementing the ANS compressor in a real time system, would be probability modelling, which must be handled quickly and carefully in order to not introduce too much latency. The decoder in today's implementation is positioned to handle decoding of some streams of data at 42Mb/s, however the encoder would struggle at 1.34Mb/s and probably bottleneck the system, due to the long division employed. A Newton-Raphson method fixed point division implementation could provide significant improvement. Improving the encoder, as discussed in section 5.3 can as noted be combined with other compression algorithms as discussed in section 5.2.

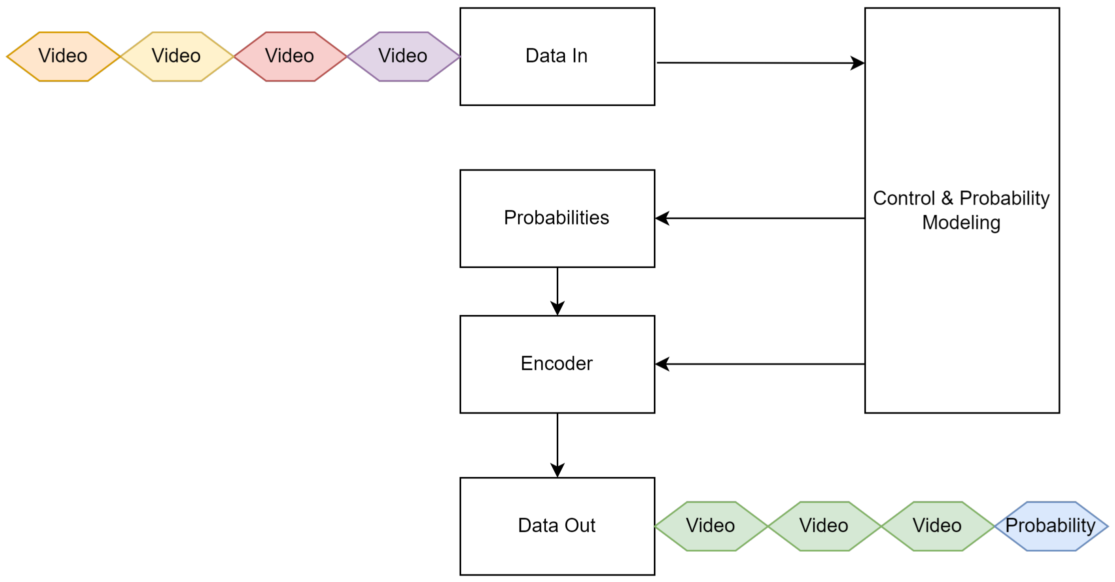


Figure 65: Suggested implementation of Ans encoder as seen part of a larger system.

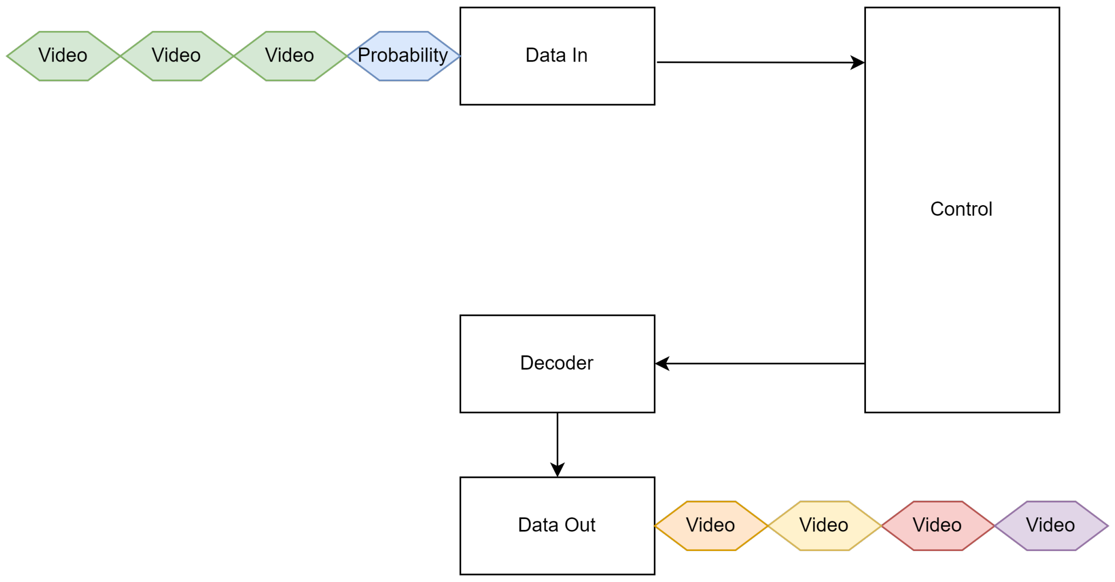


Figure 66: Suggested implementation of Ans decoder as seen part of a larger system.1

5.15 Experimental software Models

The software models are made at a higher level of abstraction than the hardware implementations, as discussed in section 3.4. This higher level of abstraction is desirable as it brings more value to the designs as reference models, and the software models should not aim to initially optimize the algorithms. However, later design iterations of the software models could lead to experimental algorithmic optimizations and other interesting discoveries. It could for instance lead to discoveries of more efficient handling or calculation of state. Further experiments could be carried out with taking a multitude of bits at once and encoding them simultaneously, which at the moment is restricted due to data dependencies.

6 Further work

A list of further work is presented based on the discussion of 5.

- Replacing the fixed division module with a more suitable one for high speed performance (div.sv). The module may be optimized, such as in the cases where there are a large amount of leading 0's or trailing 0's on both the divisor and dividend. In this case the module may quickly iterate through these and shift the result (pad it) by an appropriate to make up for it. Alternatively the module could be replaced completely with solutions based on other approaches for division, such as Newton Rhapson. One reference to a promising Fixed-Point Divider Using Newton Raphson Division module can be found here [45].

- Experiment with the fast and slow clock in the Encoder module. It does lead to some slow down, however is promising for low power and IoT applications. Furthermore it also allows the developer to change it's duty cycle, which further limits the activity factor of the circuitry and might lead to further power saving, at the cost of performance that should be further investigated. Such issues could even differ between target systems, and be used for optimizations based on performance, power and area (resources).

- Renormalization and probability modelling has not been implemented in the ANS compressor. This is strictly not needed to be performed by the compressor, however it could prove useful for designers desiring to use the ANS compressor in their own systems. Especially, good probability modeling might quickly become quite complex, however there are also solutions for this already in available open source. Nonetheless it would make the compressor module more of a complete offering and contribution to the hardware design community.

- More trials should be carried out to estimate where the ideal balance can be found for WIDTH and FBITS, with the possibility of suggesting a method for balancing these factors. Such methods should make it simple and safe to get to a point where the circuit is wide enough to prevent overflows, has large enough probability resolution while being as performant and small as possible.
- The ANS Compressor should be tested in combination with other algorithms, such as LZ77. Alternatively it is also imaginable to fuse their algorithms, rather than having them in a chain. More trials must be carried out, estimating the competiveness with existing solutions.
- More research should be done to rate the hardware implementation against existing solutions. This will both aid in understanding how good the performance actually is, as well as to identify the areas where the ANS compressor should be employed.
- The hardware implementation is based on estimates, synthesis and simulations carried out inside the EDA. The hardware implementation has not been tested on an actual FPGA, and this is a step that should be carried out to ensure that the synthesis and implementation has been conducted correctly. It would furthermore allow for real world testing, which should be performed in order to truly guarantee that it operates bug-free. Alternatively it could result in more development.
- Some new flags might be desired at the output of the ANS compressor, as discussed in section 5.4. These flags are welcome if the circuit is implemented as part of a larger system, and therefore would be connected to the control logic unit. Furthermore, enable flags should be added back to the optimized ANS compressor as it is a low cost measure with little deterioration effect on performance, and could save power. It was a design feature in the unoptimized ANS compressor which should be brought back to the optimized implementation.
- A FIFO could be added to the output of the Encoder and Decoder in conjunction with the Ready/Valid interface, as discussed in section 5.8. This would free up the FSM from blocking and awaiting for a Ready Out signal. Using the Ready/Valid interface with a FIFO as a temporary memory is a common design technique and improve throughput. It would seamlessly fit the already present signal flow and design strategy. In conclusion, it would allow the ANS compressor to interface more easily with other circuits, and act more flexibly as part of a larger system. The downside is added circuitry and overall complexity along with increased power consumption.
- A 2-state implementation of the FSM should be investigated, as suggested in section 30.
- Verification could be improved, both in terms of expanding and improving the UVM testbench as well as the other testbenches, but also to include formal verification, as discussed in section 5.11.
- The gate-level descriptions and netlists should be further investigated for the ANS compressors hardware implementation, as it has not been a focus during development due to time limitations.

7 Conclusion

This thesis proposed a hardware implementation of the compression algorithm Uniform Binary Variant (uABS) of ANS, representing a step toward the next generation line of compressors. The hardware implementation was implemented in an iterative approach, starting with creating high level and low level software models. Preceding steps were based on each other.

The ANS compressor circuit was designed to be modular, configurable and plug-and-play. It was implemented using Finite State Machine (FSM) control logic and fixed point arithmetic logic. The design was successfully synthesized for Zynq UltraScale+ ZCU106 and Spartan-7-SP701 with low power consumption and resource utilization. The implementation succeeded for both the optimized and unoptimized ANS compressors. Implementation reports concluded that the encoders and decoders utilized less than 1% of the target FPGAs resources. Namely, the encoder required less than 200 configurable logic blocks (CLB) and 11 CARRY-8s, and the decoder substituted some CLBs for Digital Signal Processors (DSPs). The designs reached clock frequencies of up to 168 MHz with conservative constraints. The clocks, DSP, internal logic and signals required 3-5mW for the encoder and 12mW for the decoder. The peak decoding speed was estimated to be 42.02Mb/s, and the peak Encoding speed was found to be 1.34Mb/s.

A testbench was created to analyze the hardware implementations' robustness and performance. The testbench consisted of 234 683 bit-strings of pseudo-random samples, where each bit-string was 150 bits wide. Making the design perform 33 702 450 encode and decode operations (in total 67 404 900). Analysis revealed that the compressor produced average compression ratios of only 1% worse than the dataset's theoretical limit (Shannon Entropy). The robustness was checked by inserting noise in the probability modelling. These results showed that one achieved comparable compression ratios with ±1 to ±5% deviation added to the actual probability of '1's. The breaking point of the inaccuracies in probability modelling was at inaccuracies ±20%, where the compressor could no longer correctly encode and decode. The Ready/Valid interface was also verified with random interfacing delays, which brings confidence in the robustness of the hardware implementation.

The software models were first implemented in C++ and thereafter in SystemVerilog. The compression behaviour of the software models was verified using similar techniques as with the testbench of the hardware implementations. Additionally, the software model's compatibility was confirmed by exchanging encoded and decoded messages between the two software models. These models proved essential as reference models during the development of the hardware implementation.

This thesis shows that the uniform binary variant (uABS) of Asymmetric Numeral Systems (ANS) proves well suited for hardware implementation. The provided hardware implementation is proven to produce compression ratios close to that of the theoretical limit for entropy-based compressors. Furthermore, it is also shown that the compressor is agile enough to be performant even with inaccurate probability modelling, satisfying both the timing and behaviour requirements of a full-fledged ANS compressor. The proposed hardware implementation is thus a valuable addition to the line of next-generation compressors. It boasts lossless decoding speeds sufficient to support the real-time decoding of video. It is a valuable addition to the line of hardware compressors, as it is both performant and lightweight in terms of power and FPGA area.

References

1. Cadence. SystemVerilog Accelerated Verification with UVM. Available from: https://cadence.com/en_US/home/training/all-courses/86070.html [Accessed on: 2022 Jul 11]
2. Duda J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. 2014. arXiv: 1311.2540 [cs.IT]. [Accessed on: 2021 Dec 12]
3. Design and Reuse. Design and Reuse, Search for Silicon IP, Hevc/h.265 IP Listing. Available from: https://www.design-reuse.com/sip/?q=hevc%5C%2Fh.265 [Accessed on: 2022 Jul 10]
4. Hsieh PA and Wu JL. A Review of the Asymmetric Numeral System and Its Applications to Digital Images. eng. Entropy (Basel, Switzerland) 2022; 24:375
5. Design and Reuse. Design and Reuse, Search for Silicon IP, UABS IP Listing. Available from: https://www.design-reuse.com/sip/?q=uABS [Accessed on: 2022 Jul 10]
6. Rørvik KBH. Next generation compression algorithm. 2021
7. P. C and Sliwa C. Data compression. Available from: https://searchstorage.techtarget.com/definition/compression [Accessed on: 2021 Sep 27]
8. Jayasankar U, Thirumal V and Ponnurangam D. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. eng. Journal of King Saud University. Computer and information sciences 2021; 33:119–40
9. Anup A, Ashok R and Raundale P. Comparative Study of Data Compression Techniques. International Journal of Computer Applications 2019 Jun; 178:15–9. DOI: 10.5120/ijca2019919104
10. Shannon CE. A mathematical theory of communication. The Bell System Technical Journal 1948; 27:623–56. DOI: 10.1002/j.1538-7305.1948.tb00917.x
11. Wade G. Signal Coding and Processing 2nd Edition. Cambridge University Press, 1994
12. Poynton C. Digital Video and HD: Algorithms and Interfaces 2nd Edition. Morgan Kaufmann Publishers, 2012
13. Keng B. Lossless Compression with Asymmetric Numeral Systems. Available from: https://bjlkeng.github.io/posts/lossless-compression-with-asymmetric-numeral-systems/ [Accessed on: 2021 Oct 20]
14. Roberts E. Lossy Compression. Available from: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/index.htm [Accessed on: 2021 Oct 4]
15. Roberts E. Lossless Compression: An Overview. Available from: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/index.htm [Accessed on: 2021 Oct 4]
16. Duda J. Asymmetric numeral systems. CoRR 2009; abs/0902.0271. arXiv: 0902.0271. Available from: http://arxiv.org/abs/0902.0271 [Accessed on: 2021 Dec 11]
17. Alonso T, Sutter G and López de Vergara JE. An FPGA-Based LOCO-ANS Implementation for Lossless and Near-Lossless Image Compression Using High-Level Synthesis. Electronics 2021; 10. DOI: 10.3390/electronics10232934. Available from: https://www.mdpi.com/2079-9292/10/23/2934 [Accessed on: 2021 Dec 11]
18. Semiconductor N. Introduction to FPGA Resources. Available from: https://www.ni.com/docs/en-US/bundle/labview-nxg-fpga-targets/page/intro-fpga-resources.html [Accessed on: 2022 Mar 22]
19. FPGAKey. FPGA Slices. Available from: https://www.fpgakey.com/wiki/details/52 [Accessed on: 2022 Jul 8]
20. FPGAKey. FPGA IOB. Available from: https://www.fpgakey.com/wiki/details/50 [Accessed on: 2022 Mar 22]
21. Inc X. Spartan-7 SP701 FPGA Evaluation Kit. Available from: https://www.xilinx.com/products/boards-and-kits/sp701.html [Accessed on: 2022 Jul 6]

22. IAMElectronics. Small form factor FPGA module with Xilinx Spartan-7. Available from: http://www.iamelectronic.com/products/T0006_Spartan-7_FPGA_Module/datasheet/ [Accessed on: 2022 Jul 8]
23. Inc X. UltraScale Architecture - Staying a Generation Ahead with an Extra Node of Value. Available from: https://www.xilinx.com/products/technology/ultrascale.html [Accessed on: 2022 Jul 7]
24. Inc X. Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. Available from: https://www.xilinx.com/products/boards-and-kits/zcu106.html#specifications [Accessed on: 2022 Jul 6]
25. Agne A, Hangmann H, Happe M, Platzner M and Plessl C. Seven recipes for setting your FPGA on fire – A cookbook on heat generators. eng. Microprocessors and microsystems 2014; 38:911–9
26. Xilinx. Xilinx 16nm UltraScale+ Devices Yield 2-5X Performance/Watt Advantage. Available from: https://www.xilinx.com/content/dam/xilinx/publications/archives/xcell/Xcell90.pdf [Accessed on: 2022 Jul 8]
27. Sadrusham NJ. Setup and hold slack. Available from: https://asic-soc.blogspot.com/2013/08/setup-and-hold-slack.html [Accessed on: 2022 Jul 8]
28. ICDesignTips. Setup and hold time. Available from: https://www.icdesigntips.com/2020/10/setup-and-hold-time-explained.html [Accessed on: 2020 Jan 1]
29. Fletcher CW. EECS150: Interfaces: "FIFO" (a.k.a. Ready/Valid). Available from: https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf [Accessed on: 2009 Feb 24]
30. Green W. Division in Verilog. Available from: https://projectf.io/posts/division-in-verilog/ [Accessed on: 2020 Jul 1]
31. Electrical4u. Binary Division. Available from: https://www.electrical4u.com/binary-division/ [Accessed on: 2021 Jan 31]
32. Green W. Fixed Point Numbers in Verilog. Available from: https://projectf.io/posts/fixed-point-numbers-in-verilog/ [Accessed on: 2020 May 26]
33. EmLogic. Digital Design and Verification services. Available from: https://emlogic.no/fpga/ [Accessed on: 2022 Jul 7]
34. Sutherland. Synthesizing SystemVerilog-Busting the Myth that SystemVerilog is only for Verification. Available from: https://sutherland-hdl.com/papers/2013-SNUG-SV_Synthesizable-SystemVerilog_paper.pdf [Accessed on: 2022 Jul 7]
35. EEtimes. EDA vendors reveal plans for SystemVerilog. Available from: https://www.eetimes.com/eda-vendors-reveal-plans-for-systemverilog/ [Accessed on: 2022 Jul 7]
36. Myers A. State machines. Available from: https://www.cs.cornell.edu/courses/cs211/2006sp/Lectures/L26-MoreGraphs/state_mach.html [Accessed on: 2022 Jul 7]
37. Warya A. Difference between Mealy machine and Moore machine. Available from: https://www.geeksforgeeks.org/difference-between-mealy-machine-and-moore-machine/ [Accessed on: 2022 Jul 7]
38. Patel MK. Figure 7.2: State diagrams for Edge detector : Mealy Design. Available from: https://verilogguide.readthedocs.io/en/latest/verilog/fsm.html/ [Accessed on: 2022 Jul 7]
39. Accellera. UVM users guide 1.2. Available from: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf [Accessed on: 2022 Feb 2]
40. Inc C. SystemVerilog Accelerated Verification with UVM. Available from: https://www.cadence.com/en_US/home/training/all-courses/86070.html [Accessed on: 2022 Jul 10]
41. Chipverify. uvm-tutorial. Available from: https://www.chipverify.com/uvm/uvm-tutorial [Accessed on: 2022 Feb 4]
42. Vora Y. UVM phases. Available from: https://semiconreferrals.com/uvm-phases/ [Accessed on: 2021 Mar 14]
43. ChipVerify. What are UVM phases? Available from: https://www.chipverify.com/uvm/uvm-phases [Accessed on: 2022 Jul 11]

44. Xilinx. Vivado Design Suite Tutorial: Using Constraints. Available from: %7Bhttps://docs.xilinx.com/r/2021.1-English/ug945-vivado-using-constraints-tutorial/Step-3-Creating-Timing-Constraints%7D [Accessed on: 2022 Mar 4]

45. Pandey PK. Fixed-Point Divider Using Newton Raphson Division Algorithm. Available from: https://link.springer.com/chapter/10.1007/978-981-16-0275-7_19#citeas [Accessed on: 2022 Jul 11]

Appendix A Optimized ANS compressor - decoder

```
//`include "uabs_decoder_fsm.sv"
//`include "uabs_decoder.sv"
//import uabs_decoder_fsm_pkg::*;

module top #(int WIDTH = 48, int FBITS = 8)(
    input logic start_dec,
    input logic vin_dec,
    input logic rout_dec,
    input logic clk,
    input logic rst_n_dec,
    input logic [WIDTH-1:0] to_decode,
    input logic [FBITS:0] probability,
    output logic [WIDTH-1:0] comp_state_out_dec,
    output logic ready_dec,
    output logic vout_dec,
    output logic decoded_bit
    );
    logic comp_busy_dec;

    // STM
    uabs_decoder_fsm decoder_fsm(
        .rin(ready_dec),
        .vout(vout_dec),
        .start(start_dec),
        .vin(vin_dec),
        .rout(rout_dec),
        .clk(clk),
        .rst_n(rst_n_dec),
        .comp_busy(comp_busy_dec)
    );

    // Decoder
    uabs_decoder #(WIDTH, FBITS) decoder(
      .to_decode(to_decode),
      .probability(probability),
      .result(decoded_bit),
      .comp_state_out(comp_state_out_dec),
      .start(comp_busy_dec)
    );
endmodule


```

Listing 11: Optimized hardware implementation of decoder, top module

```
module uabs_decoder #(int WIDTH = 24, int FBITS = 8)(
    input logic [WIDTH-1:0] to_decode,
    input logic [FBITS:0] probability,
    input logic start,
    output logic result,
    output logic [WIDTH-1:0] comp_state_out
    );

    // Variables
    logic bit_is_one;
    logic [WIDTH*INTERNAL_EXPANSION-1:0] temp_a;
    logic [WIDTH*INTERNAL_EXPANSION-1:0] temp_b;
    logic [WIDTH*INTERNAL_EXPANSION-1:0] temp_d;
    logic [WIDTH*INTERNAL_EXPANSION-1:0] temp_e;
    logic [WIDTH*INTERNAL_EXPANSION-1:0] temp_g;

    // Calculate variables asap to speed up the rest of the processes.
    always_comb begin
        temp_d = to_decode * probability;
        temp_e = temp_d >> FBITS;
        temp_g = (temp_e[WIDTH-1:FBITS] << FBITS) + (1 << FBITS);
    end

    // calculate temp_a for determening which bit it is...
    always_comb begin
```

```
            // Only adjusts for ceil if there are FBITS.
            if(temp_e[FBITS-1:0]) begin
                // Removing FBITS and adding + 1 if decimals are detected.
                temp_a = (temp_e[WIDTH-1:FBITS] << FBITS) + (1 << FBITS);
            end
            else begin
                temp_a = temp_e;
            end
        end

        // calculate temp_b for determening which bit it is...
        always_comb begin
            temp_b = (to_decode + (1 << FBITS));
            temp_b = ((temp_b * probability) >> FBITS);
            // Only adjusts for ceil if there are FBITS.
            if(temp_b[FBITS-1:0]) begin
                // Removing FBITS and adding + 1 if decimals are detected.
                temp_b = (temp_b[WIDTH-1:FBITS] << FBITS) + (1 << FBITS);
            end
        end

        // use temp_a and temp_b for determening which bit it is...
        always @(posedge start) begin
            logic [WIDTH-1:0] bit_is_one_temp;
            bit_is_one_temp = /*ceiled*/temp_b - /*ceiled*/temp_a; //
            bit_is_one = bit_is_one_temp[FBITS]; // We do this to get a '1' or '0'
    in 1's position.
            result = bit_is_one;
        end

        // Check stage
        always_comb begin
            // Calculate the new state
            if (bit_is_one) begin
                // Only adjusts for ceil if there are FBITS.
                if(temp_e[FBITS-1:0]) begin
                    // Removing FBITS and adding + 1 if decimals are detected.
                    comp_state_out = temp_g;
                end
                else begin
                    comp_state_out = temp_e;
                end
            end
            else begin
                // Only adjusts for ceil if there are FBITS.
                if(temp_e[FBITS-1:0]) begin
                    // Removing FBITS and adding + 1 if decimals are detected.
                    comp_state_out = to_decode - temp_g;
                end
                else begin
                    comp_state_out = to_decode - temp_e;
                end
            end
        end
    endmodule : uabs_decoder
```

Listing 12: Optimized hardware implementation of decoder, arithmetic logic

```
       //`include "uabs_decoder_fsm_pkg.sv"
    import uabs_decoder_fsm_pkg::*;

    module uabs_decoder_fsm(
        input logic start,
        input logic vin,
        input logic rout,
        input logic clk,
        input logic rst_n,
        output logic rin,
        output logic vout,
        output logic comp_busy
        );

        state_dec state, next;

        always_ff @(posedge clk, negedge rst_n)
            if (!rst_n)
                state <= IDLE_DEC;
            else
                state <= next;

        always_comb begin
            case (state)
                IDLE_DEC :  begin
                            rin <= 1'b0;
                            vout <= 1'b0;
                            comp_busy <= 1'b0;
                        if (start)
                              next = READ_DEC;
                          else
                              next = IDLE_DEC;
                        end
                READ_DEC :  begin
                            rin <= 1'b1;
                            vout <= 1'b0;
                            comp_busy <= 1'b0;
                            if (vin)
                              next = COMP_BUSY_DEC;
                          else
                              next = READ_DEC;
                            end
                COMP_BUSY_DEC:  begin
                            rin <= 1'b0;
                            vout <= 1'b0;
                            comp_busy <= 1'b1;
                            next = DONE_DEC;
                            end
                DONE_DEC :  begin
                            rin <= 1'b0;
                            vout <= 1'b1;
                            comp_busy <= 1'b0;
                            if (rout)
                                next = IDLE_DEC;
                            else
                                next = DONE_DEC;
                            end
                default:begin
                            rin <= 'x;
                            vout <= 'x;
                            comp_busy <= 'x;
                            next = XXX_DEC;
                        end
            endcase
        end
    endmodule : uabs_decoder_fsm

```

Listing 13: Optimized hardware implementation of decoder, FSM

```
    package uabs_decoder_fsm_pkg;
        typedef enum logic [2:0] {
            IDLE_DEC = 3'b000,
            READ_DEC = 3'b001,
            COMP_BUSY_DEC = 3'b010,
            DONE_DEC = 3'b100,
            XXX_DEC = 'x } state_dec;
    endpackage

```

Listing 14: Optimized hardware implementation of decoder, package

```
//`timescale 10ns/100ps
//`include "fsm1_pkg_b.sv"
module uabs_fsm_tb #(int WIDTH = 48, int FBITS = 8);
  //import uabs_decoder_fsm_pkg::*;
  logic [WIDTH-1:0] comp_state_out_dec;
  logic start_dec;
  logic vin_dec;
  logic rout_dec;
  logic clk;
  logic rst_n_dec;
  logic [WIDTH-1:0] to_decode;
  logic ready_dec;
  logic vout_dec;
  logic decoded_bit;
  logic [2:0] current_state_dec;
  logic comp_done_dec;

  parameter SF = 2.0**-FBITS;
  parameter COMPRESSION_BITS = 8;

  logic [WIDTH:0] probability;
  logic [7:0] compression_result;

  top #(WIDTH, FBITS) top (.*);

  /* Clock */
  initial forever
    #1 clk = ~clk;

  /* Stimuli */
  initial begin
    /* Start clock */
    tb_reset;
    #100; // Warmup delay

    probability = (1 << (FBITS-1));       // static probability here it is (1/2) =
    0.5
    to_decode = 316 << (FBITS); //decoded should be 1100_0011
    for(int i = COMPRESSION_BITS; i > 0; i--) begin
      decode_one_bit(to_decode, probability);
      compression_result[COMPRESSION_BITS - i] = decoded_bit;
    end
    $display("\t%d:\t Result %b", $time, compression_result);


    probability = (1 << (FBITS-1));       // static probability here it is (1/2) =
    0.5
    to_decode = 307 << (FBITS); //decoded should be 1100_1100
    for(int i = COMPRESSION_BITS; i > 0; i--) begin
      decode_one_bit(to_decode, probability);
      compression_result[COMPRESSION_BITS - i] = decoded_bit;
    end
    $display("\t%d:\t Result %b", $time, compression_result);


    #100; // Wait for waveform
    $finish;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end

  function void tb_reset;
    start_dec = 0;
    vin_dec = 0;
    rout_dec = 0;
    rst_n_dec = 0;
    //decoder_en = 0;
    clk = 0;
  endfunction : tb_reset

  task single_run (__start, __vin, __ack, __rst_n, nothing, __decoder_en, mode = 1)
    ;

    /* Decoder mode */
    if(mode == 1) begin
```

```
      #1
      start_dec = __start; vin_dec = !__vin /*ok...*/; rout_dec = __ack; rst_n_dec
    = __rst_n;
      //decoder_en = __decoder_en;

      //print_compression_data;
      //print_state(current_state_dec);
      //print_data(start_dec, vin_dec, rout_dec, rst_n_dec, ready_dec, decode,
    vout_dec, "decoder");
      #1;
    end
  endtask : single_run

  task decode_one_bit(logic __to_decode, logic [WIDTH-1:0] probability);
    /* Process stage
     * DECODING
     */
    begin
      //integer __to_decode;

      /* First round */
      single_run(0, 0, 0, 0, 0, 0, 1);
      single_run(0, 0, 0, 1, 0, 0, 1);
      single_run(0, 0, 0, 0, 0, 0, 1);

      /* READ */
      //to_decode = __to_decode; // Not necesarry...
      //comp_state_in_dec = state;
      //probability = probability;
      single_run(1, 0, 0, 1, 0, 0, 1);
      single_run(0, 0, 0, 1, 0, 0, 1);
      /* COMP */
      single_run(0, 1, 0, 1, 0, 1, 1);
      //#2;
      //#400;
      /* DONE */
      single_run(0, 0, 0, 1, 0, 0, 1);
      //$display("to_decode = %b and comp_state_out_dec %b", to_decode,
    comp_state_out_dec);
      to_decode = comp_state_out_dec;
      //$display("to_decode = %b and comp_state_out_dec %b", to_decode,
    comp_state_out_dec);
      /* IDLE */
      single_run(0, 0, 1, 1, 0, 0, 1);
      single_run(0, 0, 0, 1, 0, 0, 1);
      //single_run(0, 0, 0, 1, 0, 0, 1);
      //single_run(0, 0, 0, 1, 0, 0, 1);
    end

  endtask : decode_one_bit


endmodule

```

Listing 15: Optimized hardware implementation of decoder, testbench

```
    create_clock -period 5.950 -name clk -waveform {0.000 2.975} [get_ports clk]
    set_input_delay -clock [get_clocks clk] -min -add_delay 2.000 [get_ports {
    probability[*]}]
    set_input_delay -clock [get_clocks clk] -max -add_delay 3.000 [get_ports {
    probability[*]}]
    set_input_delay -clock [get_clocks clk] -min -add_delay 2.000 [get_ports {
    to_decode[*]}]
    set_input_delay -clock [get_clocks clk] -max -add_delay 3.000 [get_ports {
    to_decode[*]}]
    set_input_delay -clock [get_clocks clk] -min -add_delay 2.750 [get_ports
    rout_dec]
    set_input_delay -clock [get_clocks clk] -max -add_delay 3.000 [get_ports
    rout_dec]
    set_input_delay -clock [get_clocks clk] -min -add_delay 2.600 [get_ports
    rst_n_dec]
    set_input_delay -clock [get_clocks clk] -max -add_delay 3.000 [get_ports
    rst_n_dec]
    set_input_delay -clock [get_clocks clk] -min -add_delay 2.750 [get_ports
    start_dec]
    set_input_delay -clock [get_clocks clk] -max -add_delay 3.000 [get_ports
    start_dec]
    set_input_delay -clock [get_clocks clk] -min -add_delay 2.750 [get_ports
    vin_dec]
    set_input_delay -clock [get_clocks clk] -max -add_delay 3.000 [get_ports
    vin_dec]
    set_output_delay -clock [get_clocks clk] -min -add_delay 0.100 [get_ports
    decoded_bit]
    set_output_delay -clock [get_clocks clk] -max -add_delay 1.000 [get_ports
    decoded_bit]
    set_output_delay -clock [get_clocks clk] -min -add_delay 0.100 [get_ports
    ready_dec]
    set_output_delay -clock [get_clocks clk] -max -add_delay 1.000 [get_ports
    ready_dec]
    set_output_delay -clock [get_clocks clk] -min -add_delay 0.100 [get_ports
    vout_dec]
    set_output_delay -clock [get_clocks clk] -max -add_delay 1.000 [get_ports
    vout_dec]

```

Listing 16: Optimized hardware implementation of encoder, timing constraints

Appendix B Optimized ANS compressor - encoder

```
    //`include "uabs_encoder_fsm.sv"
    //`include "uabs_encoder.sv"
    import uabs_encoder_fsm_pkg::*;

    module top #(int WIDTH = 24, int FBITS = 8)( // If FBITS are too large causes
    error.
        input logic start_enc,
        input logic vin_enc,
        input logic rout_enc,
        input logic clk,
        input logic div_clk,
        input logic rst_n_enc,
        input logic to_encode,
        input logic [FBITS-1:0] probability,
        input logic [WIDTH-1:0] comp_state_in_enc,
        output logic idle,
        output logic ready_enc,
        output logic vout_enc,
        output logic [WIDTH-1:0] encoded
        );

        logic check_enc;
        logic encode;
        logic comp_init_enc;
        logic comp_done_enc;

        uabs_encoder_fsm encoder_fsm(
            .rin(ready_enc),
            .vout(vout_enc),
            .idle(idle),
            .start(start_enc),
            .vin(vin_enc),
            .comp_done(comp_done_enc),
            .rout(rout_enc),
            .clk(clk),
            .rst_n(rst_n_enc),
            .comp_init(comp_init_enc)
        );

        // Encoder
        uabs_encoder #(WIDTH, FBITS) encoder(
          .to_encode(to_encode),
          .probability(probability),
          .clk(clk),
          .result(encoded),
          .comp_state_in(comp_state_in_enc),
          .comp_done(comp_done_enc),
          .comp_init(comp_init_enc)
        );

    endmodule

```

Listing 17: Optimized hardware implementation of encoder, top module

```
//`include "div.sv"
module uabs_encoder #(int WIDTH = 24, int FBITS = 8)(
    input logic to_encode,
    input logic [FBITS-1:0] probability,
    input logic clk,
    input logic comp_init,
    input logic [WIDTH-1:0] comp_state_in,
    output logic comp_done, // quotient and remainder are valid
    output logic [WIDTH-1:0] result
    );

    logic div_busy;  // calculation in progress
    logic div_dbz;   // divide by zero flag
    logic div_ovf;   // overflow flag (fixed-point)
    logic [WIDTH-1:0] div_x; // dividend
    logic [WIDTH-1:0] div_y; // divisor
    logic [WIDTH-1:0] div_q; // quotient
    logic [WIDTH-1:0] div_r; // remainder

    logic [WIDTH-1:0] state;

    div #(.WIDTH(WIDTH), .FBITS(FBITS)) div_inst (
        .clk(clk),
        .start(comp_init),
        .busy(div_busy),
        .valid(comp_done),
        .dbz(div_dbz),
        .rst_n(!comp_init),
        .ovf(div_ovf),
        .x(div_x),
        .y(div_y),
        .q(div_q),
        .r(div_r));


    // Encoder preparations
    always_comb begin
        if(comp_init) begin
            if(to_encode)
                div_y <= probability;
            else
                div_y <= (1 << (FBITS)) - probability;
        end
        else
            div_y <= 'x;
    end

    always_comb begin
        if(comp_init)
            div_x <= comp_state_in + (!to_encode << (FBITS));
        else
            div_x <= 'x;
    end

    always_comb begin
        if(div_busy)
            state = 'x;
        else begin
            if (to_encode) begin
                state = div_q;
```

```
                    // Adjust for floor
                    state[FBITS-1:0] = '0;
                    //result = state;
                end
                else begin
                    state = div_q - (1 << (FBITS));
                    // Removing FBITS and adding + 1 if decimals are detected.
                    if(state[FBITS-1:0]) begin
                        state = (state[WIDTH-1:FBITS] << FBITS) + (1 << FBITS);
                    end
                    //result = state;
                end
            end
        end

        always_comb begin
            result <= state;
        end
    endmodule : uabs_encoder

```

Listing 18: Synthesizable optimized hardware implementation of encoder, arithmetic logic

```
    //`include "uabs_encoder_fsm_pkg.sv"
    import uabs_encoder_fsm_pkg::*;

    module uabs_encoder_fsm (
        input logic start,
        input logic vin,
        input logic comp_done,
        input logic rout,
        input logic clk,
        input logic rst_n,
        output logic rin,
        output logic vout,
        output logic idle,
        output logic comp_init);

        state_encoder state, next;

        always_ff @(posedge clk, negedge rst_n)
            if (!rst_n)
                state <= IDLE_ENC;
            else
                state <= next;

        always_comb begin
            case (state)
                IDLE_ENC :  begin
                            rin <= 1'b0;
                            vout <= 1'b0;
                            idle <= 1'b1;
                            comp_init <= 1'b0;
                        if (start)
                            next = READ_ENC;
                          else
                            next = IDLE_ENC;
                        end
                READ_ENC :  begin
                            rin <= 1'b1;
                            vout <= 1'b0;
                            idle <= 1'b1;
                            comp_init <= 1'b1;
                            if (vin)
                              next = COMP_BUSY;
                          else
                              next = READ_ENC;
                            end
                COMP_BUSY :  begin
                            rin <= 1'b0;
                            vout <= 1'b0;
                            idle <= 1'b0;
                            comp_init <= 1'b0;
                            if (comp_done)
                                next = DONE_ENC;
                            else
                                next = COMP_BUSY;
                            end
                DONE_ENC :  begin
                            rin <= 1'b0;
                            vout <= 1'b1;
                            idle <= 1'b0;
                            comp_init <= 1'b0;
                            if (rout)
                                next = IDLE_ENC;
                            else
                                next = DONE_ENC;
                            end
                default:begin
                            rin <= 'x;
                            vout <= 'x;
                            idle <= 'x;
                            comp_init <= 'x;
                            next = XXX_ENC;
                        end
            endcase
        end
    endmodule : uabs_encoder_fsm

```

Listing 19: Synthesizable optimized hardware implementation of encoder, FSM

```
    /*
     *   Original author
     *   link: https://projectf.io/posts/division-in-verilog/
     *   Edited by: Kare-Benjamin H. Rorvik
     *
     */
    module div #(
        parameter WIDTH=4,  // width of numbers in bits
        parameter FBITS=0   // fractional bits (for fixed point)
        ) (
        input wire logic clk,
        input wire logic rst_n, // Added for reset (added by KB)
        input wire logic start,          // start signal
        output      logic busy,          // calculation in progress
        output      logic valid,         // quotient and remainder are valid
        output      logic dbz,           // divide by zero flag
        output      logic ovf,           // overflow flag (fixed-point)
        input wire logic [WIDTH-1:0] x,  // dividend
        input wire logic [WIDTH-1:0] y,  // divisor
        output      logic [WIDTH-1:0] q,  // quotient
        output      logic [WIDTH-1:0] r   // remainder
        );

        // avoid negative vector width when fractional bits are not used
        localparam FBITSW = (FBITS) ? FBITS : 1;

        logic [WIDTH-1:0] y1;           // copy of divisor
        logic [WIDTH-1:0] q1, q1_next;  // intermediate quotient
        logic [WIDTH:0] ac, ac_next;    // accumulator (1 bit wider)

        localparam ITER = WIDTH+FBITS;  // iterations are dividend width +
    fractional bits
        logic [$clog2(ITER)-1:0] i;     // iteration counter

        always_comb begin
            if (ac >= {1'b0,y1}) begin
                ac_next = ac - y1;
                {ac_next, q1_next} = {ac_next[WIDTH-1:0], q1, 1'b1};
            end else begin
                {ac_next, q1_next} = {ac, q1} << 1;
            end
        end

        always_ff @(posedge clk) begin
```

```
            /* Added by KB */
            if (!rst_n) begin
                busy <= 0;
                valid <= 0;
                dbz <= 0;
                ovf <= 0;
                q <= '0;
                r <= '0;
            end
            if (start) begin
                //$display("\t%d:\t We got started!",$time);
                valid <= 0;
                ovf <= 0;
                i <= 0;
                if (y == 0) begin  // catch divide by zero
                    busy <= 0;
                    dbz <= 1;
                end else begin
                    busy <= 1;
                    dbz <= 0;
                    y1 <= y;
                    {ac, q1} <= {{WIDTH{1'b0}}, x, 1'b0};
                end
            end else if (busy) begin
                if (i == ITER-1) begin  // done
                    busy <= 0;
                    valid <= 1;
                    q <= q1_next;
                    r <= ac_next[WIDTH:1];  // undo final shift
                end else if (i == WIDTH-1 && q1_next[WIDTH-1:WIDTH-FBITSW]) begin
    // overflow?
                    busy <= 0;
                    ovf <= 1;
                    q <= 0;
                    r <= 0;
                end else begin  // next iteration
                    i <= i + 1;
                    ac <= ac_next;
                    q1 <= q1_next;
                end
            end
        end
    endmodule

```

Listing 20: Synthesizable optimized hardware implementation of encoder, fixed point division module

```
    package uabs_decoder_fsm_pkg;
        typedef enum logic [2:0] {
            IDLE_DEC = 3'b000,
            READ_DEC = 3'b001,
            COMP_BUSY_DEC = 3'b010,
            DONE_DEC = 3'b100,
            XXX_DEC = 'x } state_dec;
    endpackage

```

Listing 21: Synthesizable optimized hardware implementation of encoder, package

```
    //`include "fsm1_pkg_b.sv"
    module uabs_fsm_tb #(int WIDTH = 48, int FBITS = 4);
      import uabs_encoder_fsm_pkg::*;

      logic [WIDTH-1:0] comp_state_in_enc;
      logic [WIDTH-1:0] comp_state_out_enc;
      logic start_enc;
      logic vin_enc;
      logic rout_enc;
      logic clk;
      logic rst_n_enc;
      logic to_encode;
      logic check_enc;
      logic comp_one_done_enc;
      logic comp_zero_done_enc;
      logic ready_enc;
      logic vout_enc;
      logic encode;
      logic idle;
      logic [WIDTH-1:0] encoded;
      logic div_clk;
      logic comp_ready_enc_temp;
      logic comp_init_enc_temp;
      logic div_busy;
      logic div_valid;
      logic div_dbz;
      logic div_ovf;
      logic [WIDTH-1:0] div_x; // parameterise
      logic [WIDTH-1:0] div_y; // parameterise
      logic [WIDTH-1:0] div_q; // parameterise
      logic [WIDTH-1:0] div_r; // parameterise

      parameter SF = 2.0**-FBITS;
      parameter COMPRESSION_BITS = 8;


      logic [2:0] current_state_encoder;
      logic [WIDTH-1:0] probability;
      logic [WIDTH-1:0] temp_encoded;
      logic [COMPRESSION_BITS-1:0] compression_result;
      logic [COMPRESSION_BITS-1:0] compression_stimuli = 8'b1100_1100;
      top #(WIDTH, FBITS) top (.*);

      /* Clock */
      initial forever begin
        #1 clk = ~clk;
        div_clk = ~div_clk;
      end

      /* Stimuli */
      initial begin
        /* Start clock */
        tb_reset;

        #200;

        comp_state_in_enc = (1 << (FBITS)); // starting state, here it's 1.
        probability = (1 << (FBITS-1));       // static probability here it is
    (1/2) = 0.5
        encode_one_bit(compression_stimuli[COMPRESSION_BITS-1], comp_state_in_enc);
     // First round to insert +1
        for(int ii = COMPRESSION_BITS - 2; ii >= 0; ii--) begin
          encode_one_bit(compression_stimuli[ii], temp_encoded);
        end

        $display("\t%d:\t Result %d", $time, $itor(encoded*SF));

        compression_stimuli = 8'b1100_0011;
        comp_state_in_enc = (1 << (FBITS)); // starting state, here it's 1.
        probability = (1 << (FBITS-1));       // static probability here it is
    (1/2) = 0.5
        encode_one_bit(compression_stimuli[COMPRESSION_BITS-1], comp_state_in_enc);
     // First round to insert +1
        for(int ii = COMPRESSION_BITS - 2; ii >= 0; ii--) begin
          encode_one_bit(compression_stimuli[ii], temp_encoded);
        end
        $display("\t%d:\t Result %d", $time, $itor(encoded*SF));

        compression_stimuli = 8'b1111_0000;
        comp_state_in_enc = (1 << (FBITS)); // starting state, here it's 1.
        probability = (1 << (FBITS-1));       // static probability here it is
    (1/2) = 0.5
        encode_one_bit(compression_stimuli[COMPRESSION_BITS-1], comp_state_in_enc);
     // First round to insert +1
        for(int ii = COMPRESSION_BITS - 2; ii >= 0; ii--) begin
          encode_one_bit(compression_stimuli[ii], temp_encoded);
        end
        $display("\t%d:\t Result %d", $time, $itor(encoded*SF));

        $finish;
      end

      initial begin
        $dumpfile("dump.vcd");
        $dumpvars(1);
      end

      function void tb_reset;
        start_enc = 0;
        vin_enc = 0;
        rout_enc = 0;
        rst_n_enc = 0;
        //encoder_en = 0;
        clk = 0;
        div_clk = 0;
      endfunction : tb_reset

      task single_run (__start, __vin, __ack, __rst_n, nothing, nothing_2, mode = 1);

        /* Encoder mode */
        if(mode == 1) begin
          #1
          start_enc = __start; vin_enc = __vin; rout_enc = __ack; rst_n_enc = __rst_n;
          //encoder_en = __encoder_en;

          //print_compression_data;
          //print_state(current_state_encoder);
          //print_data(start_enc, vin_enc, rout_enc, rst_n_enc, ready_enc, encode, vout_enc, "encoder");
          #1;
        end
      endtask : single_run

      task encode_one_bit(logic __to_encode, logic [WIDTH-1:0] state);
        /* Process stage
         * ENCODING
         */
        begin
          logic [WIDTH-1:0] __encoded;

          /* First round */
          single_run(0, 0, 0, 0, 0, 0, 1);
          single_run(0, 0, 0, 1, 0, 0, 1);
          single_run(0, 0, 0, 0, 0, 0, 1);
          /* READ */
          to_encode = __to_encode;
          comp_state_in_enc = state;
          single_run(1, 0, 0, 1, 0, 0, 1);
          single_run(0, 0, 0, 1, 0, 0, 1);
          /* COMP */
          single_run(0, 1, 0, 1, 0, 1, 1);
          #400;
          /* DONE */
          single_run(0, 0, 0, 1, 0, 0, 1);
          temp_encoded = encoded;
          /* IDLE */
          single_run(0, 0, 1, 1, 0, 0, 1);
          single_run(0, 0, 0, 1, 0, 0, 1);
          single_run(0, 0, 0, 1, 0, 0, 1);
          single_run(0, 0, 0, 1, 0, 0, 1);
        end

      endtask : encode_one_bit


    endmodule
```

```

```
Listing 22: Synthesizable optimized hardware implementation of encoder, testbench

Appendix C Master testbench

```
//`include "fsm1_pkg_b.sv"
module uabs_fsm_tb #(parameter WIDTH = 180, parameter FBITS = 8, parameter
CLK_DIVISOR = 2, parameter DUTY_DIVISOR = 2, parameter CLK_DIVISOR_WIDTH =
$clog2(CLK_DIVISOR+1)); // Max recommended width 128 (vivado), largest tested
180 and 8 FBITS.
  import uabs_decoder_fsm_pkg::*;

  // Decoder
  logic [WIDTH-1:0] /*comp_state_in_dec, */comp_state_in_enc;
  logic [WIDTH-1:0] comp_state_out_dec/*, comp_state_out_enc*/;
  logic start_dec, start_enc;
  logic vin_dec, vin_enc;
  logic rout_dec, rout_enc;
  logic clk;
  logic rst_n_dec, rst_n_enc;
  logic [WIDTH-1:0] to_decode;
  logic to_encode;
  //logic decoder_en, encoder_en;
  //logic check_dec, check_enc;
  //logic checked_dec, checked_enc;
  //logic bit_is_one_dec, bit_is_one_enc;
  //logic comp_one_done_dec, comp_one_done_enc;
  //logic comp_zero_done_dec, comp_zero_done_enc;
  logic ready_dec, ready_enc;
  logic vout_dec, vout_enc;
  //logic decode, encode;
  logic /*idle_dec, */idle_enc;
  logic decoded_bit;
  logic [WIDTH-1:0] encoded;
  //logic comp_ready_dec_temp, comp_ready_enc_temp;
  //logic comp_busy_dec_temp, comp_busy_enc_temp;
  //logic comp_init_dec_temp, comp_init_enc_temp;
  //logic div_busy;
  //logic div_valid, div_valid_dec;
  //logic div_dbz;
  //logic div_ovf;
  //logic [WIDTH-1:0] div_x; // parameterise
  //logic [WIDTH-1:0] div_y; // parameterise
  //logic [WIDTH-1:0] div_q; // parameterise
  //logic [WIDTH-1:0] div_r; // parameterise
  //logic [WIDTH-1:0] div_q_dec; // parameterise

  parameter SF = 2.0**-FBITS;
  parameter COMPRESSION_BITS = 150; // Max 150 given optimal probabilities, 80
for +- probability deviation of 20%
  parameter RANDOM_ITERATIONS = 10;
  parameter EXTRA_RUNS = 32;
  parameter EXTRA_RUNS_DIVISOR = 64;
  parameter DELAY = 1;
  parameter DELAY_2 = 1;
  parameter FAST_CLOCK_DELAY = 1;
  parameter PROBABILITY_DEVIATION_OPTIONS = 9;
  parameter ENCODER_VOUT_MAX_WAIT_LIMIT = 1000;

  real probability_deviation_array[PROBABILITY_DEVIATION_OPTIONS] = {-0.2,-0.1,
 -0.05, -0.01, 0, 0.01, 0.05, 0.1, 0.20};
  integer probability_deviation_pointer = 0;

  logic div_clk;
  logic [WIDTH-1:0] probability;
  logic [COMPRESSION_BITS:0] temp_decoded; // parameterize ...
  logic [WIDTH-1:0] temp_encoded;
  logic [COMPRESSION_BITS-1:0] compression_result;
  logic [COMPRESSION_BITS-1:0] compression_stimuli;
```

```
    uabs_decoder_fsm_pkg::state_dec current_state_dec;
    uabs_encoder_fsm_pkg::state_encoder current_state_encoder;
    real temp_encoded_bits;
    logic [WIDTH-1:0] temp_temp_encoded_bits;
    real temp_Encoded_bits_real;
    real temp_temp_encoded_bits_real;
    real stimuli_temp_bit;
    real shannon;
    real shannon_temp;
    integer run_counter = 0;
    integer success_counter = 0;
    integer bit_value;
    integer position;
    integer number_of_ones;
    real probability_real;
    real probability_deviation = 0.20;
    real probability_without_deviation;
    logic compression_status;
    real compression_ratio;
    logic [WIDTH -1:0] encoded_copy;
    integer fd;
    real modifier_1, modifier_2, modifier_3, modifier_4;

    design_top #(WIDTH, FBITS, CLK_DIVISOR, DUTY_DIVISOR, CLK_DIVISOR_WIDTH)
  design_top (.*);

    /* Clock */
    initial forever begin
      #FAST_CLOCK_DELAY clk = ~clk;
    end

    /*
    initial forever begin
      #DELAY_2 div_clk = ~div_clk;
    end
    */
    /* Stimuli */
    initial begin

      tb_reset;

      /* Start clock */
      fd = $fopen("./simulation_result.csv", "w"); //"a" will amend, "w" will
  overwrite...
      $fdisplay(fd,"Compression Status;Number_of_ones;Compression ratio;
  Actual_bits_for_stimuli;Ideal Probability;Actual Probability;Forced Probability
   Deviation;Probability_with_deviation;Encoded;Stimuli");

      // Test 1
      for (int ii = 0; ii <= RANDOM_ITERATIONS - 1; ++ii) begin
        //compression_stimuli = 1 << (WIDTH-1);
        //compression_stimuli = 1 << COMPRESSION_BITS-1;
        comp_state_in_enc = (1 << (FBITS));
        compression_stimuli = 0;
        modifier_1 = $urandom_range(1,EXTRA_RUNS);
        modifier_2 = $urandom_range(1,EXTRA_RUNS_DIVISOR);
        modifier_3 = COMPRESSION_BITS;
        modifier_4 = modifier_3 * (modifier_1/modifier_2);
        probability_deviation_pointer = $urandom_range(0,
  PROBABILITY_DEVIATION_OPTIONS-1);
        probability_deviation = probability_deviation_array[
  probability_deviation_pointer];
        //$display("Random Modifier %0f", modifier_4);
        for(int jj = 0; jj < modifier_4; ++jj) begin
            //bit_value = 1;//$urandom_range(0,1);
            position = $urandom_range(0,COMPRESSION_BITS-1);
            //compression_stimuli[position] = bit_value;
            //$display("Compression stimuli = %b", compression_stimuli);
            //bit_value = $urandom_range(0,1);
            compression_stimuli[position] = 1;
        end
        calculate_probility(compression_stimuli, probability_deviation,
  probability, probability_real, number_of_ones, probability_without_deviation);
```

```
        encode_decode(compression_stimuli, comp_state_in_enc, probability,
    compression_status, compression_ratio);
        $fdisplay(fd,"%0d;%0d;%0f;%0d;%0d;%0f;%0f;%0f;%0d;%0d",
    compression_status, number_of_ones, compression_ratio, stimuli_temp_bit,
    probability, probability_without_deviation, probability_deviation ,
    probability_real, encoded_copy, compression_stimuli);
      end





      // Test 2 (Not currently in use)
      /*
      for (int ii = 0; ii <= RANDOM_ITERATIONS - 1; ++ii) begin
        //compression_stimuli = 1 << (WIDTH-1);
        //compression_stimuli = 1 << COMPRESSION_BITS-1;
        comp_state_in_enc = (1 << (FBITS));
        compression_stimuli = 0;
        for(int jj = 0; jj < (COMPRESSION_BITS * EXTRA_RUNS) + 1; ++jj) begin
            bit_value = $urandom_range(0,1);
            position = $urandom_range(COMPRESSION_BITS-1);
            compression_stimuli[position] = bit_value;
            //$display("Compression stimuli = %b", compression_stimuli);
            end
        calculate_probility(compression_stimuli, probability_deviation,
    probability, probability_real, number_of_ones, probability_without_deviation);
        encode_decode(compression_stimuli, comp_state_in_enc, probability,
    compression_status, compression_ratio);
        $fdisplay(fd,"%0d;%0d;%0f;%0d;%0d;%0f;%0f;%0f,%0d;%0d",
    compression_status, number_of_ones, compression_ratio, stimuli_temp_bit,
    probability, probability_without_deviation, probability_deviation ,
    probability_real, encoded_copy, compression_stimuli);
       end
      */

      // Test 3 (Not currently in use)
      // 11000011 Stimuli used for waveform generation and illustration
      //for (int ii = 0; ii <= RANDOM_ITERATIONS - 1; ++ii) begin
        //compression_stimuli = 1 << (WIDTH-1);
        //compression_stimuli = 1 << COMPRESSION_BITS-1;
        comp_state_in_enc = (1 << (FBITS));
        compression_stimuli = 8'b1111_1111;
        probability = 1 << FBITS-1;
        //calculate_probility(compression_stimuli, probability_deviation,
    probability, probability_real, number_of_ones, probability_without_deviation);
        encode_decode(compression_stimuli, comp_state_in_enc, probability,
    compression_status, compression_ratio);
        $fdisplay(fd,"%0d;%0d;%0f;%0d;%0d;%0f;%0f;%0f,%0d;%0d",
    compression_status, number_of_ones, compression_ratio, stimuli_temp_bit,
    probability, probability_without_deviation, probability_deviation ,
    probability_real, encoded_copy, compression_stimuli);
      */


      // Test 4 Bit shift ones
      comp_state_in_enc = (1 << (FBITS));
      compression_stimuli = 0;
      calculate_probility(compression_stimuli, probability_deviation, probability
    , probability_real, number_of_ones, probability_without_deviation);
      encode_decode(compression_stimuli, comp_state_in_enc, probability,
    compression_status, compression_ratio);
      $fdisplay(fd,"%0d;%0d;%0d;%0f;%0f;%0f;%0f", number_of_ones, encoded_copy,
    compression_status, compression_ratio, probability, probability_real,
    probability_deviation);
      for (int ii = 0; ii <= COMPRESSION_BITS - 1; ++ii) begin
        comp_state_in_enc = (1 << (FBITS));
        compression_stimuli[ii] = 1;
        calculate_probility(compression_stimuli, probability_deviation,
    probability, probability_real, number_of_ones, probability_without_deviation);
        encode_decode(compression_stimuli, comp_state_in_enc, probability,
    compression_status, compression_ratio);
        $fdisplay(fd,"%0d;%0d;%0f;%0d;%0d;%0f;%0f;%0f,%0d;%0d",
    compression_status, number_of_ones, compression_ratio, stimuli_temp_bit,
    probability, probability_without_deviation, probability_deviation ,
    probability_real, encoded_copy, compression_stimuli);
      end

      $fclose(fd);
      if(run_counter == success_counter)
        $display("\t%d:\t All %d runs succeeded!", $time, run_counter);
      else
        $error("\t%d:\%d runs failed!", $time, run_counter-success_counter);

      $finish;
    end



    task calculate_probility(input logic [WIDTH-1:0] data, input real inaccuracy,
   output logic [WIDTH-1:0] probability_fixed, output real
    probability_float_rounded, output integer number_of_ones, output real
    probability_accurate); begin
      real counter = 0; // Counter as float to help division later
      //logic [WIDTH-1:0] probability_fixed;
      real probability_float;
      number_of_ones = 0;
      for (int ii = COMPRESSION_BITS - 1; ii >= 0; --ii) begin
        //$display("Iteration %b", ii);
        if(data[ii])begin
          //$display("Found One");
          counter++;
        end
      end
      number_of_ones = counter;
      //$display("Counted %b", counter);
      //$display("Inacc %f", inaccuracy);
      probability_float = (counter / COMPRESSION_BITS);
      probability_accurate = probability_float;
      //$display("probability is %f", probability_float);
      if (probability_float >= (1.00 - inaccuracy)) begin
        probability_float = probability_float - inaccuracy;
        //$display("New probability after fix with - %f is %f", inaccuracy,
    probability_float);
      end
      else if (probability_float + inaccuracy <= 0) begin
        probability_float = probability_float - inaccuracy;
        //$display("We did it at probability %0f", probability_float);
      end
      else if (probability_float - inaccuracy <= 0) begin
        probability_float = probability_float + inaccuracy;
        //$display("We did it at probability %0f", probability_float);
      end
      else begin
        probability_float = probability_float + inaccuracy;
        //$display("New probability after fix with + %f is %f", inaccuracy,
    probability_float);
      end
      probability_fixed = probability_float * (1 << FBITS);
      probability_float_rounded = probability_fixed * SF;
      //$display("\t%d:\t Probability: %f", $time, probability_float);
      //$display("Probability fixed: %b", probability_fixed);
      //$display("\t%d:\t Probability fixed as float %f", $time,
    probability_float_rounded);
      // Convert from float to fixed
      // Multiply input number by 2^F, F is fraction bits.'
      counter = 0;
      end
    endtask : calculate_probility



    task encode_decode(logic [COMPRESSION_BITS-1:0] __stimuli, logic [WIDTH-1:0]
    __state_in, [COMPRESSION_BITS-1:0] __probability, output logic
    __compression_status, output real __compression_ratio); begin
      tb_reset;
      run_counter++;
      /* Compression */
      // The encoder encodes from MSB -> LSB.

      compression_stimuli = __stimuli; //'b1110_1110_1110_1110_1110_1110_1110_1110;  //'b1110_1110_1110_1110_1110_1110_1110_1110;
      comp_state_in_enc = __state_in; // starting state, here it's 1.
      probability = __probability;       // static probability here it is (1/4) = 0.5
      encode_one_bit(compression_stimuli[COMPRESSION_BITS-1], comp_state_in_enc); // First round to insert +1
      for(int ii = COMPRESSION_BITS - 2; ii >= 0; ii--) begin
        encode_one_bit(compression_stimuli[ii], temp_encoded);
      end

      tb_reset;

      /* Compression */
      //comp_state_in_dec = (1 << (FBITS)); // starting state, here it's 1.
      //probability = (1 << (FBITS-1));       // static probability here it is (1/2) = 0.5
      //to_decode = 316 << (FBITS);
      //$display(" to_decode = %d", encoded);
      to_decode = encoded;
      encoded_copy = SF*to_decode;
      //$display(" to_decode = %d", encoded_copy);

      for(int i = COMPRESSION_BITS; i > 0; i--) begin
        decode_one_bit;
        compression_result[COMPRESSION_BITS - i] = decoded_bit;
      end
      //$display("\t%d:\t Result %b", $time, compression_result);
      if(compression_result == compression_stimuli) begin
        //$display("\t%d:\t Compression succeeded! Stimuli %b == Reconstruction %b", $time, compression_stimuli, compression_result);
        success_counter++;
        __compression_status = 1'b1;
      end
      else
        __compression_status = 1'b0;
        //$error("\t%d:\t Compression failed. Stimuli %b == Reconstruction %b", $time, compression_stimuli, compression_result);

      //$display("\t%d:\t The encoded representation (state) was %d with p = %f", $time, $itor(temp_encoded * SF), SF*probability);
      temp_temp_encoded_bits = $ceil(SF*temp_encoded); // convert to an integer (must round up to whole number (but should never be a decimal...))
      temp_encoded_bits = $ceil($clog2(temp_temp_encoded_bits + 1)); //Required bits are log2(number) + 1.
      stimuli_temp_bit = $ceil($clog2(__stimuli + 1)); //already ceiled by clog2 so not needed?
      //$display("given bits = %f, calculated bits = %f", temp_encoded_bits, stimuli_temp_bit);
      //$display("\t%d:\t Encoded bits: %f, Decoded bits %f", $time, temp_encoded_bits, stimuli_temp_bit);
      temp_temp_encoded_bits_real = temp_encoded_bits;
      temp_Encoded_bits_real = temp_temp_encoded_bits_real/COMPRESSION_BITS;
      __compression_ratio = temp_Encoded_bits_real;
      //$display("\t%d:\t Compression ratio: %f", $time, temp_Encoded_bits_real);
    end
    endtask : encode_decode

    // Dump simulation data
    /*
    initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
    end
    */

    function void tb_reset;
      // Decoder
```

```
      start_dec = 0;
      vin_dec = 0;
      rout_dec = 0;
      rst_n_dec = 0;
      clk = 0;

      // Encoder
      start_enc = 0;
      vin_enc = 0;
      rout_enc = 0;
      rst_n_enc = 0;
      clk = 0;
    endfunction : tb_reset

    task single_run (__start, __vin, __ack, __rst_n, nothing, nothing_2, mode =
  1);

      /* Encoder mode */
      if(mode == 1) begin
        #1
        start_enc = __start; vin_enc = __vin; rout_enc = __ack; rst_n_enc =
  __rst_n;

        #1;
      end
      else begin
        #1
        start_dec = __start; vin_dec = __vin; rout_dec = __ack; rst_n_dec =
  __rst_n;
        #1;
      end
    endtask : single_run

    task decode_one_bit();
      /* Process stage
       * DECODING
       */
      begin
        /* First round */
        single_run(1, 0, 0, 1, 0, 0, 0);
        single_run(0, 0, 0, 1, 0, 0, 0);
        /* COMP */
        single_run(0, 1, 0, 1, 0, 1, 0);
        //#2;
        //#400;
        /* DONE */
        single_run(0, 0, 0, 1, 0, 0, 0);
        //$display("to_decode = %b and comp_state_out_dec %b", to_decode,
  comp_state_out_dec);
        to_decode = comp_state_out_dec;
        //$display("to_decode = %b and comp_state_out_dec %b", to_decode,
  comp_state_out_dec);
        /* IDLE */
        single_run(0, 0, 1, 1, 0, 0, 0);
        //single_run(0, 0, 0, 1, 0, 0, 1);
        //single_run(0, 0, 0, 1, 0, 0, 1);
        //single_run(0, 0, 0, 1, 0, 0, 1);
      end

    endtask : decode_one_bit

    task encode_one_bit(logic __to_encode, logic [WIDTH-1:0] state);
      /* Process stage
       * ENCODING
       */
      int stuck_in_while_protection = 0;
      begin
        logic [WIDTH-1:0] __encoded;

        /* First round */
        single_run(0, 0, 0, 0, 0, 0, 1);
        single_run(0, 0, 0, 1, 0, 0, 1);
        single_run(0, 0, 0, 0, 0, 0, 1);
        /* READ */
        to_encode = __to_encode;
        comp_state_in_enc = state;
        single_run(1, 0, 0, 1, 0, 0, 1);
        single_run(0, 0, 0, 1, 0, 0, 1);
        /* COMP */
        single_run(0, 1, 0, 1, 0, 1, 1);
        single_run(0, 0, 0, 1, 0, 1, 1);
        stuck_in_while_protection = 0;
        while(!vout_enc & (stuck_in_while_protection <
ENCODER_VOUT_MAX_WAIT_LIMIT)) begin
          #FAST_CLOCK_DELAY;
          #FAST_CLOCK_DELAY;
          stuck_in_while_protection++;
        end
        /* DONE */
        single_run(0, 0, 0, 1, 0, 0, 1);
        temp_encoded = encoded;
        /* IDLE */
        single_run(0, 0, 1, 1, 0, 0, 1);
        single_run(0, 0, 0, 1, 0, 0, 1);
        single_run(0, 0, 0, 1, 0, 0, 1);
        single_run(0, 0, 0, 1, 0, 0, 1);
      end

    endtask : encode_one_bit


  endmodule

```

Listing 23: Optimized ANS compressor, master testbench

Appendix D UVM testbench

The most important modules of the UVM testbench has been added to the appendix. The testbench was based on strategies and methodology proposed by Cadence in their course "SystemVerilog Accelerated Verification with UVM" [1] (completed while working with ARM)

D.1 UVM uABS

```
class uabs_env extends uvm_env;

// Components of the environment
uabs_tx_agent agent;
bit checks_enable = 1;

bit coverage_enable = 1;

// Component macro
//`uvm_component_utils(uabs_env)

`uvm_component_utils_begin(uabs_env)
    `uvm_field_int(checks_enable, UVM_ALL_ON)
    `uvm_field_int(coverage_enable, UVM_ALL_ON)
`uvm_component_utils_end

// Component constructor
function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction : new

// UVM build phase()
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agent = uabs_tx_agent::type_id::create("tx_agent", this);
endfunction : build_phase
```

```

    function void start_of_simulation_phase(uvm_phase phase);
        `uvm_info(get_type_name(), {"start of simulation for ", get_full_name()},
    UVM_HIGH)
    endfunction : start_of_simulation_phase

endclass : uabs_env


```

Listing 24: Optimized ANS compressor, UVM testbench, uabs_env

```
        interface uabs_if #(parameter WIDTH = 24, parameter FBITS = 4, parameter
    CLK_DIVISOR = 4,
                        parameter DUTY_DIVISOR = 4, parameter CLK_DIVISOR_WIDTH =
    $clog2(CLK_DIVISOR+1),
                        parameter MAX_ALLOWED_VOUT_WAIT_CYCLES = WIDTH*100)
    (input clock, input reset, input [WIDTH-1:0] encoded, input vout_enc, input
    idle_enc, input vout_dec, input decoded_bit, input [WIDTH-1:0]state_out_dec);
    timeunit 1ns;
    timeprecision 100ps;

    import uvm_pkg::*;
    `include "uvm_macros.svh"

    import uabs_pkg::*;

        // Actual signals
        logic to_encode;
        logic [WIDTH-1:0] to_decode;
        logic [WIDTH-1:0] to_decode_temp;
        //logic decoder_en;
        //logic encoder_en;
        logic start_enc;
        logic start_dec;
        logic vin_enc;
        logic vin_dec;
        logic rout_enc;
        logic rout_dec;
        logic [WIDTH-1:0] state_in_enc;
        logic [FBITS:0] probability;
        //input logic rst_n_enc
        logic in_suspend = 0; // Temp
        //logic [47:0] encoded;
        //integer decoded;
        logic encoder_done = 0;
        logic decoder_done = 0;
        logic rst_n_enc;
        logic rst_n_dec;

        integer start_enc_delay;
        integer vin_enc_delay;
        integer rout_enc_delay;
        integer start_dec_delay;
        integer vin_dec_delay;
        integer rout_dec_delay;

        int counter = 0;

        // signals for transaction recording
        bit monstart, drvstart;

        // local storage for payload
        logic [7:0] payload_mem [0:63];

        // local flags for drive and monitor:
        logic drive_done = 0;
        logic monitor_done = 0;

         // This number depends ont he WIDTH that is used... this parameter is an
    approximation, change if needed.

        task uabs_reset();
```

```
            @(posedge reset);
            to_encode <= 'hz;
            to_decode <= 'hz;
            //decoder_en <= 1'b0;
            //encoder_en <= 1'b0;
            start_enc <= 1'b0;
            start_dec <= 1'b0;
            vin_enc <= 1'b0;
            vin_dec <= 1'b0;
            rout_enc <= 1'b0;
            rout_dec <= 1'b0;
            rst_n_enc <= 1'b0;
            rst_n_dec <= 1'b0;
            //rst_n_enc <= 1'b0;
            disable send_to_dut;
        endtask : uabs_reset

        task simple_reset();
            to_encode <= 'hz;
            to_decode <= 'hz;
            //decoder_en <= 1'b0;
            //encoder_en <= 1'b0;
            start_enc <= 1'b0;
            start_dec <= 1'b0;
            vin_enc <= 1'b0;
            vin_dec <= 1'b0;
            rout_enc <= 1'b0;
            rout_dec <= 1'b0;
            state_in_enc <= 'hz;
            probability <= 'hz;
            rst_n_enc <= 1'b0;
            rst_n_dec <= 1'b0;
            //rst_n_enc <= 1'b0;
        endtask : simple_reset

        // Gets a packet and drive it into the DUT
        task send_to_dut(input  bit to_encode_in,
                                bit to_decode_in,
                                //bit decoder_en_in,
                                //bit encoder_en_in,
                                bit start_enc_in,
                                bit start_dec_in,
                                bit vin_enc_in,
                                bit vin_dec_in,
                                bit rout_enc_in,
                                bit rout_dec_in,
                                logic [WIDTH-1:0] state_in_enc_in,
                                logic [FBITS:0] probability_in,
                                integer start_enc_delay_in,
                                integer vin_enc_delay_in,
                                integer rout_enc_delay_in,
                                integer start_dec_delay_in,
                                integer vin_dec_delay_in,
                                integer rout_dec_delay_in);

            // Start to send packet if not in_suspend signal:
            @(negedge clock iff (!in_suspend));

            // Trigger for transaction recording
            drvstart = 1'b1;
            drive_done = 1'b0;
            rst_n_enc = 1'b1;
            rst_n_dec = 1'b1;
            start_enc_delay = start_enc_delay_in;
            vin_enc_delay = vin_enc_delay_in;
            rout_enc_delay = rout_enc_delay_in;
            start_dec_delay = start_dec_delay_in;
            vin_dec_delay = vin_dec_delay_in;
            rout_dec_delay = rout_dec_delay_in;

            `uvm_info("uabs_if", "sender is driving Encoder", UVM_HIGH)
            // Drive Encoder with DUT
            @(negedge clock iff (!in_suspend))begin
                //to_encode = to_encode_in;
                //probability = probability_in;
                //state_in_enc = state_in_enc_in;
                //decoder_en = decoder_en_in;
                //encoder_en = encoder_en_in;
                //start_enc = start_enc_in;
                //start_dec = start_dec_in;
                //vin_enc = vin_enc_in;
                //vin_dec = vin_dec_in;
                //rout_dec = rout_dec_in;
            end

            // Fork Join makes the encoder interfacing more random, as the interfacing is executed in paralell.
            fork

                // Encoder Start Interfacing
                begin
                    // Wait with delay then start encoder
                    repeat(start_enc_delay) begin
                        @(negedge clock iff (!in_suspend));
                    end
                    start_enc = start_enc_in;
                end

                // Encoder Valid Interfacing
                begin
                    // Wait with delay then make valid input and drive stimuli to encoder
                    repeat(vin_enc_delay) begin
                        @(negedge clock iff (!in_suspend));
                    end
                    vin_enc = vin_enc_in;
                    to_encode = to_encode_in;
                    probability = probability_in;
                    state_in_enc = state_in_enc_in;
                end

                // Encoder Ready Out Interfacing
                begin
                    // Wait for Encoder to be done
                    `uvm_info("uabs_if", "sender is waiting for encoder to be done", UVM_HIGH)
                    while(!encoder_done) begin
                        @(negedge clock iff (!in_suspend));
                    end

                    // Now drive the Encoded message to Decoder and intputs

                    // Wait with delay and drive Decoder and release Encoder
                    repeat(rout_enc_delay) begin
                        @(negedge clock iff (!in_suspend));
                    end
                        rout_enc = rout_enc_in;
                        to_decode_temp = encoded;
                        //vin_dec = vin_dec_in;
                        //rout_dec = rout_dec_in;
                end
            join

            fork

                // Decoder Start Interfacing
                begin
                    repeat(start_dec_delay) begin
                        @(negedge clock iff (!in_suspend));
                    end
                    `uvm_info("uabs_if", "sender is driving Decoder", UVM_HIGH)
                    //rout_enc = rout_enc_in;
                    start_dec = start_dec_in;
                end

                // Decoder Valid In Interfacing
                begin
                    repeat(vin_dec_delay) begin
                        @(negedge clock iff (!in_suspend));
                    end
                    vin_dec = vin_dec_in;
                    to_decode = to_decode_temp;
                end

                // Decoder Ready Out Interfacing
                begin
                    // Wait for Decoder to be done
                    `uvm_info("uabs_if", "sender is waiting for encoder to be done"
    , UVM_HIGH)
                    while(!decoder_done) begin
                        @(negedge clock iff (!in_suspend));
                    end
                    repeat(rout_dec_delay) begin
                        @(negedge clock iff (!in_suspend));
                    end
                    rout_dec = rout_dec_in;
                end
            join


            @(posedge clock iff (!in_suspend)) begin
                drive_done = 1'b1;
            end

            `uvm_info("uabs_if", "sender waiting for monitor to be done", UVM_HIGH)
            while(!(monitor_done & drive_done)) begin
                @(posedge clock iff (!in_suspend));
            end

            simple_reset;

            // Wait for monitor to catch up...
            @(posedge clock iff (!in_suspend))

            // reset trigger
            `uvm_info("uabs_if", "sender ending and resetting!", UVM_HIGH)
            drvstart = 1'b0;
        endtask : send_to_dut

        // Collect packets
        task collect_packet(output  bit to_encode_out,
                                    bit to_decode_out,
                                    bit start_enc_out,
                                    bit start_dec_out,
                                    bit vin_enc_out,
                                    bit vin_dec_out,
                                    bit rout_enc_out,
                                    bit rout_dec_out,
                                    logic [WIDTH-1:0] state_in_enc_out,
                                    logic [FBITS:0] probability_out,
                                    logic [WIDTH-1:0] encoded_out,
                                    logic decoded_bit_out,
                                    logic [WIDTH-1:0] state_out_dec_out,
                                    integer start_enc_delay_out,
                                    integer vin_enc_delay_out,
                                    integer rout_enc_delay_out,
                                    integer start_dec_delay_out,
                                    integer vin_dec_delay_out,
                                    integer rout_dec_delay_out);
            //Monitor looks at the bus on posedge (Driver uses negedge)
            //@(posedge in_data_vld);

            @(posedge clock iff(!in_suspend /* & in_data_vld */))
            // Trigger for transaction recording
            monstart = 1'b1;
            monitor_done = 1'b0;
            encoder_done = 1'b0;
            decoder_done = 1'b0;
            start_enc_delay_out = start_enc_delay;
```

```
            vin_enc_delay_out = vin_enc_delay;
            rout_enc_delay_out = rout_enc_delay;
            start_dec_delay_out = start_dec_delay;
            vin_dec_delay_out = vin_dec_delay;
            rout_dec_delay_out = rout_dec_delay;

            `uvm_info("uabs_if", "collector collect_packets", UVM_HIGH)
            // Collect Header {Length, Addr}

            //`uvm_info("uabs_if","ARE WE STUCK HERE?", UVM_HIGH) // Just for debug
     remove later

            // Wait one cycle ...
            @(posedge clock iff (!in_suspend))

            `uvm_info("uabs_if", "collector waiting for encoder vout ", UVM_HIGH)
            // Wait for encoder to have a valid output
            counter = 0;
            while(!vout_enc && (counter < 200)) begin
                @(posedge clock iff (!in_suspend));
               //`uvm_info("uabs_if", "still waiting for encoder vout... ",
    UVM_HIGH)
               counter++;
            end

            // Sample Encoder
            @(posedge clock iff (!in_suspend)) begin
                to_encode_out = to_encode;
                start_enc_out = start_enc;
                //start_dec_out = start_dec;
                vin_enc_out = vin_enc;
                //vin_dec_out = vin_dec;
                rout_enc_out = rout_enc;
                //rout_dec_out = rout_dec;
                encoded_out = encoded;
                probability_out = probability;
                state_in_enc_out = state_in_enc;
                //state_out_dec_out = state_out_dec;
            end

            /*rst_n_enc_out = rst_n_enc */

            // Encoder is done, but not idle
            encoder_done = 1'b1;

            counter = 0;
            while(!idle_enc && (counter < 200)) begin
                @(posedge clock iff (!in_suspend));
               //`uvm_info("uabs_if", "still waiting for encoder idle... ",
    UVM_HIGH)
               counter++;
            end

            // Wait for decoder to have valid out
            `uvm_info("uabs_if", "collector waiting for decoder vout", UVM_HIGH)
            while(!vout_dec) begin
                @(posedge clock iff (!in_suspend));
            end

            decoder_done = 1'b1;

            // Sample decoded data
            @(posedge clock iff (!in_suspend)) begin
                //start_enc_out = start_enc;
                start_dec_out = start_dec;
                //vin_enc_out = vin_enc;
                vin_dec_out = vin_dec;
                //rout_enc_out = rout_enc;
                rout_dec_out = rout_dec;
                decoded_bit_out = decoded_bit;
                state_out_dec_out = state_out_dec;
            end

            // Alert that monitor is done
            @(posedge clock iff (!in_suspend)) begin
                monitor_done = 1'b1;
            end

            //$display("collector %b waiting for sender %b to be done",
    monitor_done, drive_done);
            // Check that driver and monitor both are done...
            while(!(monitor_done & drive_done)) begin
                @(posedge clock iff (!in_suspend));
            end
            //$display("collector %b waiting for sender %b to be done",
    monitor_done, drive_done);

            /*
            * Do whatever else here with data from the DUT
            */

            // Wait for sender to catch up...
            @(posedge clock iff (!in_suspend));
            `uvm_info("uabs_if", "collection ending!", UVM_HIGH)
            monstart = 1'b0;
            //`uvm_info("uabs_if","WE MADE IT OUT of collect_packet =)", UVM_HIGH)
    // Just for debug remove later
        endtask : collect_packet

        // If the channel suspends ports are incorrectly connected/driver
        // the uabs input will be suspended and the simulation could hang.
        // This assertion checks for this and raises a (non-UVM) error message
        property uabs_suspended;
            @(posedge clock) !in_suspend ##1 in_suspend[*10] |=> !in_suspend;
        endproperty

        UABS_SUSPEND : assert property (uabs_suspended)
            else
            begin
                $error("\n** Assertion Error - uabs interface is suspended: Check
    channel suspend ports! \n ");
                $finish;
            end


        property uabs_encoder_stuck;
            @(posedge clock) !vout_enc & start_enc ##1 !vout_enc[*
    MAX_ALLOWED_VOUT_WAIT_CYCLES] |=> vout_enc;
        endproperty

        UABS_ENC_VOUT_STUCK : assert property (uabs_encoder_stuck)
            else
            begin
                $error("\n** Assertion Error - uabs encoder is stuck in waiting for
     vout: Check Encoder design!\n ");
                $finish;
            end

        property uabs_decoder_stuck;
            @(posedge clock) !vout_dec & start_dec ##1 (!vout_dec & start_dec)[*
    MAX_ALLOWED_VOUT_WAIT_CYCLES] |=> vout_dec;
        endproperty

        UABS_DEC_VOUT_STUCK : assert property (uabs_decoder_stuck)
            else
            begin
                $error("\n** Assertion Error - uabs decoder is stuck in waiting for
     vout: Check Decoder design!\n ");
                $finish;
            end


    endinterface : uabs_if

```

Listing 25: Optimized ANS compressor, UVM testbench, uabs_if

```
  class uabs_packet extends uvm_sequence_item;

parameter WIDTH = 24;
parameter FBITS = 4;
parameter int SAFETY_HEADROOM = $ceil((1.0 * WIDTH) * 0.20);
parameter CONSTRAINT_MIN = (1 << FBITS) - 1;
parameter CONSTRAINT_MAX = (1 << (WIDTH - SAFETY_HEADROOM)) - 1;
parameter MAX_INTERFACE_DELAY = 100;
parameter MIN_INTERFACE_DELAY = 0;

// Inputs
rand logic to_encode;
rand logic [WIDTH-1:0] state_in_enc;
rand logic [FBITS:0] probability;
// Outputs
logic [WIDTH-1:0] encoded;
logic decoded_bit;
logic [WIDTH-1:0] state_out_dec;
logic [WIDTH-1:0] to_decode; // Takes the encoded state, does not need stimuli.
// Less important inputs
rand logic start_enc;
rand logic vin_enc;
rand logic rout_enc;
rand logic start_dec;
rand logic vin_dec;
rand logic rout_dec;

rand integer start_enc_delay;
rand integer vin_enc_delay;
rand integer rout_enc_delay;
rand integer start_dec_delay;
rand integer vin_dec_delay;
rand integer rout_dec_delay;

/* rand logic rst_n_enc; */

`uvm_object_utils_begin(uabs_packet)
  //`uvm_field_int(decoder_en, UVM_ALL_ON)
  //`uvm_field_int(encoder_en, UVM_ALL_ON)
  `uvm_field_int(to_encode, UVM_ALL_ON)
  `uvm_field_int(start_enc, UVM_ALL_ON)
  `uvm_field_int(vin_enc, UVM_ALL_ON)
  `uvm_field_int(rout_enc, UVM_ALL_ON)
  `uvm_field_int(start_dec, UVM_ALL_ON)
  `uvm_field_int(vin_dec, UVM_ALL_ON)
  `uvm_field_int(rout_dec, UVM_ALL_ON)

  `uvm_field_int(start_enc_delay, UVM_ALL_ON)
  `uvm_field_int(vin_enc_delay, UVM_ALL_ON)
  `uvm_field_int(rout_enc_delay, UVM_ALL_ON)
  `uvm_field_int(start_dec_delay, UVM_ALL_ON)
  `uvm_field_int(vin_dec_delay, UVM_ALL_ON)
  `uvm_field_int(rout_dec_delay, UVM_ALL_ON)

  `uvm_field_int(encoded, UVM_ALL_ON)
  `uvm_field_int(to_decode, UVM_ALL_ON)
  `uvm_field_int(decoded_bit, UVM_ALL_ON)
  `uvm_field_int(state_out_dec, UVM_ALL_ON)
  `uvm_field_int(state_in_enc, UVM_ALL_ON)
  `uvm_field_int(probability, UVM_ALL_ON)
  /* `uvm_field_int(rst_n_enc, UVM_ALL_ON) */
`uvm_object_utils_end

function new (string name = "uabs_packet");
  super.new(name);
endfunction: new

constraint to_encode_range {0 <= to_encode; to_encode <= 1;}
//constraint to_decode_range {CONSTRAINT_MIN < to_decode ;to_decode <=
  CONSTRAINT_MAX;}
//constraint decoder_always_on {decoder_en != 0;}
//constraint encoder_always_on {encoder_en != 0;}
constraint start_enc_range {1 <= start_enc; start_enc <= 1;}
```

```
  constraint vin_enc_range {1 <= vin_enc; vin_enc <= 1;}
  constraint rout_enc_range {1 <= rout_enc; rout_enc <= 1;}
  constraint start_dec_range {1 <= start_dec; start_dec <= 1;}
  constraint vin_dec_range {1 <= vin_dec; vin_dec <= 1;}
  constraint rout_dec_range {1 <= rout_dec; rout_dec <= 1;}
  constraint state_in_range {(1 << FBITS) <= state_in_enc; state_in_enc <=
    CONSTRAINT_MAX; state_in_enc[FBITS:0] == 0;}
  constraint probability_range {1 <= probability; probability <= (1 << FBITS) - 1;}

  constraint start_enc_delay_range {MIN_INTERFACE_DELAY <= start_enc_delay;
    start_enc_delay <= MAX_INTERFACE_DELAY;}
  constraint start_dec_delay_range {MIN_INTERFACE_DELAY <= start_dec_delay;
    start_dec_delay <= MAX_INTERFACE_DELAY;}
  constraint vin_enc_delay_range {MIN_INTERFACE_DELAY <= vin_enc_delay;
    vin_enc_delay <= MAX_INTERFACE_DELAY;}
  constraint vin_dec_delay_range {MIN_INTERFACE_DELAY <= vin_dec_delay;
    vin_dec_delay <= MAX_INTERFACE_DELAY;}
  constraint rout_enc_delay_range {MIN_INTERFACE_DELAY <= rout_enc_delay;
    rout_enc_delay <= MAX_INTERFACE_DELAY;}
  constraint rout_dec_delay_range {MIN_INTERFACE_DELAY <= rout_dec_delay;
    rout_dec_delay <= MAX_INTERFACE_DELAY;}


  //rand logic [WIDTH-1:0] comp_state_in_enc;
  //rand logic [FBITS:0] probability


  //constraint decoded_range {0 < encoded; encoded <= 1;}
  /* constraint to_encode_range {0 < rst_n_enc; rst_n_enc <= 1;} */

endclass : uabs_packet

```

Listing 26: Optimized ANS compressor, UVM testbench, uabs_packet

```
    package uabs_pkg;
  import uvm_pkg::*;
    `include "uvm_macros.svh"

    typedef uvm_config_db#(virtual uabs_if) uabs_vif_config;
    `include "uabs_packet.sv"
    `include "uabs_tx_monitor.sv"
    `include "uabs_tx_sequencer.sv"
    `include "uabs_tx_seqs.sv"
    `include "uabs_tx_driver.sv"
    `include "uabs_tx_agent.sv"
    `include "uabs_env.sv"
endpackage : uabs_pkg

```

Listing 27: Optimized ANS compressor, UVM testbench, uabs_pkg

```
        typedef enum bit {EQUALITY, UVM} comp_t;
    //import uvm_pkg::*;
    //`include "uvm_macros.svh"
    //`include "uabs_pkg.sv"
    //import uabs_pkg::*;
    //import channel_pkg::*;
    //`include "uabs_tb.sv"
    //`include "uabs_test_lib.sv"

    class uabs_scoreboard extends uvm_scoreboard;


        // TLm Port Declarations
        `uvm_analysis_imp_decl(_uabs)
        //`uvm_analysis_imp_decl(_chan0)

        uvm_analysis_imp_uabs #(uabs_packet, uabs_scoreboard) sb_uabs_in;
        //uvm_analysis_imp_chan0 #(channel_packet, uabs_scoreboard) sb_chan0;

        // Scoreboard Packet Queues
        uabs_packet sb_queue0[$];
        //channel_packet cp_queue0[$];


        // Scoreboard Statistics
        int packets_in, in_dropped;
        int packets_ch0, compare_ch0, miscompare_ch0, dropped_ch0;

        // variable for comparer policy
        comp_t compare_policy = UVM;

        `uvm_component_utils_begin(uabs_scoreboard)
            `uvm_field_enum(comp_t, compare_policy, UVM_ALL_ON)
        `uvm_component_utils_end

        // Constructor
        function new(string name="", uvm_component parent=null);
            super.new(name, parent);
            sb_uabs_in = new("sb_uabs_in", this);
            //sb_chan0 = new("sb_chan0", this);
        endfunction : new

        /*
        // custom packet compare function uysing inequality operators
        function bit comp_equal (input uabs_packet ub, input channel_packet cp);
            // returns first mismatch only
            `uvm_info(get_type_name, "Inside compare UVM functions...", UVM_LOW)
            if(ub.encoder_en != cp.encoder_en) begin
                `uvm_error("PKT_COMPARE",$sformatf("Encoder_en mismatch uABS %0d
    Chan %0d",ub.encoder_en, cp.encoder_en))
                return(0);
            end
            return(1);
        endfunction : comp_equal
        */


        function bit comp_equal_2 (input uabs_packet ub);
            // returns first mismatch only
            `uvm_info(get_type_name, "Inside compare UVM functions...", UVM_HIGH)

            if(ub.to_encode != ub.decoded_bit) begin
                `uvm_error("PKT_COMPARE",$sformatf("to_encode %0d mismatch with
    decoded_bit %0d p %0b",ub.to_encode, ub.decoded_bit, ub.probability))
                `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s", ub.
    sprint()), UVM_LOW)
                return(0);
            end
            if(ub.state_in_enc != ub.state_out_dec) begin
                `uvm_error("PKT_COMPARE",$sformatf("state_in_enc %0b mismatch with
    state_out_dec Chan %0b at p %0b",ub.state_in_enc, ub.state_out_dec, ub.
    probability))
                `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s", ub.
    sprint()), UVM_LOW)
                return(0);
            end

            /*
            if(ub.start_enc != 1) begin
                `uvm_error("PKT_COMPARE",$sformatf("Encoder_en mismatch uABS %0d
    Chan %0d",ub.start_enc, 1))
                return(0);
            end
            */
            /*
            *
            * More compares
            *
            */
            return(1);
        endfunction : comp_equal_2


        // custom packet compare function using uvm_comparer methods
        function bit comp_uvm(input uabs_packet up/*, input channel_packet cp*/, uvm_comparer comparer = null);
            string str;
            `uvm_info(get_type_name, "Inside compare UVM functions...", UVM_HIGH)
            /*
            if (comparer == null)
                comparer = new();
            comp_uvm = comparer.compare_field("encoder_en", up.encoder_en, cp.encoder_en, 1);
            */
        endfunction : comp_uvm

        virtual function void write_uabs(uabs_packet packet);
            uabs_packet sb_packet;
            logic test_result;
            // Make a copy for storing in the scoreboard
            $cast( sb_packet, packet.clone()); // Clone retunrs uvm_object type
            packets_in++;
            // This will be odd with my implementation ...
            // Is meant to be case ... (sb_packet.enable)
            sb_queue0.push_back(sb_packet);
            `uvm_info(get_type_name, "Added packet to Scoreboard Queue 0", UVM_HIGH)

            test_result = comp_equal_2(packet);
            compare_ch0 += test_result;
            miscompare_ch0 += !test_result;

            void'(sb_queue0.pop_front());
                //`uvm_info(get_type_name(), $sformatf("Scoreboard Compare Match: Channel_0 Packet\n%s", packet.sprint()), UVM_LOW)
                ///compare_ch0++;
        endfunction : write_uabs

        // Channel 0 Packet Check (write) implementation
        /*
        virtual function void write_chan0(channel_packet packet);
            bit pktcompare;
            uabs_packet sb_packet;
            packets_ch0++;
            `uvm_info(get_type_name, "Inside write_chan0 UVM function...", UVM_LOW)
            if ( sb_queue0.size() == 0) begin
                `uvm_error(get_type_name(),
                $sformatf("Scoreboard Error [EMPTY]: Received Unexpected Channel_0 Packet!\n%s",packet.sprint()))
                dropped_ch0++;
                return;
            end;
            if(compare_policy == UVM)
                // use custom comparer with UVM methods
                pktcompare = comp_uvm(sb_queue0[0],packet);
            else
                // use customcomparer wtih equality operators
                pktcompare = comp_equal(sb_queue0[0],packet);

            if( pktcompare ) begin
                void'(sb_queue0.pop_front());
                `uvm_info(get_type_name(), $sformatf("Scoreboard Compare Match: Channel_0 Packet\n%s", packet.sprint()), UVM_LOW)
                compare_ch0++;
            end
            else begin
                    sb_packet = sb_queue0[0];
                    `uvm_warning(get_type_name(), $sformatf("Scoreboard Error [MISCOMPARE] : Received Channel_0 Packet:\n%s\nExpected Channel_0 Packet:\n%s", packet.sprint(), sb_packet.sprint()))
            end
        endfunction : write_chan0
        */


        // UVM check phase
        function void check_phase(uvm_phase phase);
```

```
            `uvm_info(get_type_name(), "Checking UABS Scoreboard", UVM_LOW)
            //write_chan0(sb_chan0);
            if (sb_queue0.size() /* can also add more channels here */)
                `uvm_error(get_type_name(), $sformatf("Check:\n\n WARNING : uABS
    Scoreboard Queue's NOT Empty :\n chan0 %0d", sb_queue0.size()))
            else
                `uvm_info(get_type_name(),"Check\n\n uABS Scoreboard Empty!\n",
    UVM_LOW)
        endfunction : check_phase

        // UVM report() phase
        function void report_phase(uvm_phase phase);
            `uvm_info(get_type_name(), $sformatf("Report:\n\n Scoreboard: Packet
    Statistics \n Packets in %0d       Packets Dropped: %0d \n     Channel 0 Total:
     %0d Pass: %0d Miscompare %0d Dropped %0d \n", packets_in, in_dropped,
    packets_ch0, compare_ch0, miscompare_ch0, dropped_ch0), UVM_LOW)
        if((miscompare_ch0 + dropped_ch0) > 0)
            `uvm_error(get_type_name(),"Status:\n\nSimulation FAILED\n")
        else
            `uvm_info(get_type_name(),"Status:\n\nSimulation Passed\n", UVM_LOW)
        endfunction : report_phase

    endclass : uabs_scoreboard

```

Listing 28: Optimized ANS compressor, UVM testbench, uabs_scoreboard

```
    class uabs_tx_agent extends uvm_agent;

    uabs_tx_monitor monitor;
    uabs_tx_sequencer sequencer;
    uabs_tx_driver driver;

    // component macro
    `uvm_component_utils_begin(uabs_tx_agent)
        `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
    `uvm_component_utils_end

    // constructor
    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    // UVM build phase() method
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        //uvm_top.print_topology();
        monitor = uabs_tx_monitor::type_id::create("monitor", this);
        if(is_active == UVM_ACTIVE) begin
            sequencer = uabs_tx_sequencer::type_id::create("sequencer", this);
            driver = uabs_tx_driver::type_id::create("driver", this);
        end
    endfunction : build_phase

    // UVM connect phase() method
    function void connect_phase(uvm_phase phase);
        if(is_active == UVM_ACTIVE)
            // Connect the driver and the sequencer
            driver.seq_item_port.connect(sequencer.seq_item_export);
    endfunction : connect_phase

    function void start_of_simulation_phase(uvm_phase phase);
        `uvm_info(get_type_name(), {"start of simulation for ", get_full_name()},
    UVM_HIGH);
    endfunction : start_of_simulation_phase

    // Assign the vertiual interfaces of the agent's children
    function void assign_vi(virtual interface uabs_if vif);
        monitor.vif = vif;
        if (is_active == UVM_ACTIVE)
            driver.vif = vif;
    endfunction : assign_vi

```

```
endclass : uabs_tx_agent

```

Listing 29: Optimized ANS compressor, UVM testbench, uabs_tx_agent

```
    class uabs_tx_driver extends uvm_driver #(uabs_packet);

    // Declare this property to count packages sent
    int num_sent;

    virtual interface uabs_if vif;

    // component macro
    `uvm_component_utils_begin(uabs_tx_driver)
        `uvm_field_int(num_sent, UVM_ALL_ON)
    `uvm_component_utils_end

    // constructor - required syntax for UVM automation and utilities
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    function void connect_phase(uvm_phase phase);
        //uvm_top.print_topology();
        if(!uabs_vif_config::get(this,"","vif", vif))
            `uvm_error("NOVIF", {"virtual interface must be set for: ",
    get_full_name(), ".vif"})
    endfunction : connect_phase

    // Start of simulation
    function void start_of_simulation_phase(uvm_phase phase);
        `uvm_info(get_type_name(), {"start of simulation for ", get_full_name()},
    UVM_HIGH)
    endfunction : start_of_simulation_phase

    // UVM run phase
    task run_phase (uvm_phase phase);
        fork
            get_and_drive();
            reset_signals();
        join
    endtask : run_phase

    // Gets packets from the sequencer and passes them to the driver.
    task get_and_drive();
        @(posedge vif.reset);
        @(negedge vif.reset);
        `uvm_info(get_type_name(), "Reset dropped", UVM_HIGH)
        forever begin
            // Get new item fro mthe sequencer
            seq_item_port.get_next_item(req);

            `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s", req.sprint
    ()), UVM_FULL)

            fork
                // Send packet
                begin
                    // Needed?
                    //vif.to_encode = req.to_encode;
                    //vif.encoder_en = req.encoder_en;
                    //vif.decoder_en = req.decoder_en;
                    //`uvm_info(get_type_name(), $sformatf("We are stuck here"),
    UVM_LOW)
                    vif.send_to_dut(req.to_encode,
                                    req.to_decode,
                                    req.start_enc,
                                    req.start_dec,
                                    req.vin_enc,
                                    req.vin_dec,
                                    req.rout_enc,
                                    req.rout_dec,
                                    req.state_in_enc,
                                    req.probability,
                                    req.start_enc_delay,
                                    req.vin_enc_delay,
                                    req.rout_enc_delay,
                                    req.start_dec_delay,
                                    req.vin_dec_delay,
                                    req.rout_dec_delay);
                end

                // Trigger transaction at start of packet (trigger signal from
    itnerface)
                @(posedge vif.drvstart) void'(begin_tr(req, "Driver_UABS_Packet"));
            join

        // End transaction recording
        end_tr(req);
        num_sent++;
        // Communicate item done to the sequencer
        seq_item_port.item_done();
        end
    endtask : get_and_drive

    task reset_signals();
        forever
            vif.uabs_reset();
    endtask : reset_signals

    // UVM report_phase
    function void report_phase(uvm_phase phase);
        `uvm_info(get_type_name(), $sformatf("Report: uabs driver sent %0d packets"
    , num_sent), UVM_FULL)
    endfunction : report_phase

endclass : uabs_tx_driver


```

Listing 30: Optimized ANS compressor, UVM testbench, uabs_tx_driver

```
    class uabs_tx_monitor extends uvm_monitor;

    parameter WIDTH = 24;
    parameter FBITS = 4;
    parameter int MAX_PROBABILITY = $pow(2,FBITS)-1;
    parameter int MAX_STATE = $pow(2,SAFETY_HEADROOM) - 1;
    parameter int SAFETY_HEADROOM = $ceil((1.0 * WIDTH) * 0.20);
    parameter CONSTRAINT_MAX = (1 << (WIDTH - SAFETY_HEADROOM)) - 1;
    parameter SAFE_WIDTH = WIDTH - SAFETY_HEADROOM - FBITS;
    parameter MAX_INTERFACE_DELAY = 100;
    parameter MIN_INTERFACE_DELAY = 0;

    // Collected Data Handle
    uabs_packet pkt;

    // Count packets collected
    int num_pkt_col;

    // analysis port for lab09*
    uvm_analysis_port#(uabs_packet) item_collected_port;

    virtual interface uabs_if vif;

    bit checks_enable = 1;
    bit coverage_enable = 1;
    bit local_coverage_enable = 1;

    // component macro
    `uvm_component_utils_begin(uabs_tx_monitor)
        `uvm_field_int(num_pkt_col, UVM_ALL_ON)
        `uvm_field_int(checks_enable, UVM_ALL_ON)
        `uvm_field_int(coverage_enable, UVM_ALL_ON)
    `uvm_component_utils_end

```

```
covergroup cover_packet;
    option.per_instance = 1;
    packet_to_encode : coverpoint pkt.to_encode;
    packet_decoded : coverpoint pkt.decoded_bit;
    packet_probability : coverpoint pkt.probability {
      bins range[] = {[MAX_PROBABILITY:1]};
    }
endgroup : cover_packet


covergroup cover_packet_states;
    option.per_instance = 1;
    packet_state_in_enc : coverpoint pkt.state_in_enc[WIDTH-SAFETY_HEADROOM:FBITS+1] {
      bins range[] = {[(1 << FBITS):MAX_STATE]};
    }

    packet_state_out_dec : coverpoint pkt.state_out_dec[WIDTH-SAFETY_HEADROOM:FBITS+1] {
      bins range[] = {[(1 << FBITS):MAX_STATE]};
    }
endgroup : cover_packet_states

covergroup cover_packet_flags;
    option.per_instance = 1;
    packet_start_enc_delay : coverpoint pkt.start_enc_delay{
      bins range[] = {[MIN_INTERFACE_DELAY:MAX_INTERFACE_DELAY]};
    }

    packet_start_dec_delay : coverpoint pkt.start_dec_delay{
      bins range[] = {[MIN_INTERFACE_DELAY:MAX_INTERFACE_DELAY]};
    }

    packet_vin_enc_delay : coverpoint pkt.vin_enc_delay{
      bins range[] = {[MIN_INTERFACE_DELAY:MAX_INTERFACE_DELAY]};
    }

    packet_vin_dec_delay : coverpoint pkt.vin_dec_delay{
      bins range[] = {[MIN_INTERFACE_DELAY:MAX_INTERFACE_DELAY]};
    }

    packet_rout_enc_delay : coverpoint pkt.rout_enc_delay{
      bins range[] = {[MIN_INTERFACE_DELAY:MAX_INTERFACE_DELAY]};
    }

    packet_rout_dec_delay : coverpoint pkt.rout_dec_delay{
      bins range[] = {[MIN_INTERFACE_DELAY:MAX_INTERFACE_DELAY]};
    }
endgroup : cover_packet_flags

// component constructor - required syntax for UVM automation and utilities
function new(string name, uvm_component parent);
    super.new(name, parent);
    if (!uvm_config_int::get(this,"","coverage_enable",coverage_enable))
        `uvm_info("NOCOV",{"Coverage not enabled for: ",get_full_name()},UVM_LOW)
    if(local_coverage_enable) begin
        cover_packet = new();
        cover_packet.start();
        cover_packet.set_inst_name({get_full_name(), ".cover_packet"});
        `uvm_info("NOCOV",{"Coverage started for: ",get_full_name()}, UVM_LOW)

        cover_packet_states = new();
        cover_packet_states.start();
        cover_packet_states.set_inst_name({get_full_name(), ".cover_packet_states"});
        `uvm_info("NOCOV",{"Coverage started for: ",get_full_name()}, UVM_LOW)

        cover_packet_flags = new();
        cover_packet_flags.start();
        cover_packet_flags.set_inst_name({get_full_name(), ".cover_packet_flags"});
        `uvm_info("NOCOV",{"Coverage started for: ",get_full_name()}, UVM_LOW)
    end
    item_collected_port = new("item_collected_port",this);
endfunction : new

function void connect_phase(uvm_phase phase);
    if(!uabs_vif_config::get(this, get_full_name(),"vif", vif))
        `uvm_error("NOVIF", {"virtual interface must be set for: ",
get_full_name(), ".vif"})
endfunction : connect_phase

// UVM run() phase
task run_phase(uvm_phase phase);
    // Look for packets after reset
    @(posedge vif.reset)
    @(negedge vif.reset)
    `uvm_info(get_type_name(), "Detected Reset Done", UVM_HIGH)
    forever begin
        // Create collected packet instance
        pkt = uabs_packet::type_id::create("pkt", this);

        fork
            // Collect packet
            vif.collect_packet(
                pkt.to_encode,
                pkt.to_decode,
                pkt.start_enc,
                pkt.start_dec,
                pkt.vin_enc,
                pkt.vin_dec,
                pkt.rout_enc,
                pkt.rout_dec,
                pkt.state_in_enc,
                pkt.probability,
                pkt.encoded,
                pkt.decoded_bit,
                pkt.state_out_dec,
                pkt.start_enc_delay,
                pkt.vin_enc_delay,
                pkt.rout_enc_delay,
                pkt.start_dec_delay,
                pkt.vin_dec_delay,
                pkt.rout_dec_delay);
            //`uvm_info(get_type_name(), "Made it out of fork/join", UVM_LOW)
// Just for debug remove later
            // Trigger transactio nat start of packet
            @(posedge vif.monstart) void'(begin_tr(pkt, "Monitor_uabs_packet"))
;
        join

        end_tr(pkt);
        `uvm_info(get_type_name(), $sformatf("Packet Collected :\n%s", pkt.
sprint()), UVM_FULL);
        item_collected_port.write(pkt);
        num_pkt_col++;
        `uvm_info(get_type_name(), $sformatf("We have this many packets :\n%0d"
, num_pkt_col), UVM_FULL);
        //perform_coverage();
        cover_packet.sample();
        cover_packet_states.sample();
        cover_packet_flags.sample();


    end
endtask : run_phase

// Triggers corverage events
/*
function void perform_coverage();

  cover_packet.get_inst_coverage());

endfunction : perform_coverage
*/

// UVM report phase
function void report_phase(uvm_phase phase);
    `uvm_info(get_type_name(), $sformatf("Report: uabs Monitor Collected %0d Packets", num_pkt_col), UVM_FULL)
    `uvm_info(get_type_name(), $sformatf("%3.2f%% coverage achieved for stimuli packet.", cover_packet.get_inst_coverage()), UVM_LOW);
    `uvm_info(get_type_name(), $sformatf("%3.2f%% coverage achieved for flags.", cover_packet_flags.get_inst_coverage()), UVM_LOW);
    `uvm_info(get_type_name(), $sformatf("%3.2f%% coverage achieved for states.", cover_packet_states.get_inst_coverage()), UVM_LOW);
endfunction: report_phase

endclass : uabs_tx_monitor

```

Listing 31: Optimized ANS compressor, UVM testbench, uabs_tx_monitor

```
    class uabs_base_seq extends uvm_sequence #(uabs_packet);

    // Required macro for sequences automation
    `uvm_object_utils(uabs_base_seq)

    // Constructor
    function new(string name = "uabs_base_seq");
        super.new(name);
    endfunction : new

    task pre_body();
        uvm_phase phase;
        `ifdef UVM_VERSION_1_2
            // In UVM 1.2, get starting phase from method
            phase = get_starting_phase();
        `else
            phase = starting_phase;
        `endif
        if (phase != null) begin
            phase.raise_objection(this, get_type_name());
            `uvm_info(get_type_name(), "raise objection", UVM_HIGH)
        end
    endtask : pre_body

    task post_body();
        uvm_phase phase;
        `ifdef UVM_VERSION_1_2
            // In UVM 1.2, get starting phase from method
            phase = get_starting_phase();
        `else
            phase = starting_phase;
        `endif
        if (phase != null) begin
            phase.drop_objection(this, get_type_name());
            `uvm_info(get_type_name(), "drop objection", UVM_HIGH)
        end
    endtask : post_body

endclass : uabs_base_seq

class uabs_5_packets extends uabs_base_seq;
    // Should have been renamed to uabs_35000_packet.
    // Required macro for sequences automation
    `uvm_object_utils(uabs_5_packets)

    // Constructor
    function new (string name = "uabs_5_packets");
        super.new(name);
    endfunction : new

    // Sequence body definition
    virtual task body();
        `uvm_info(get_type_name(), "Executing uabs_5_packets sequence", UVM_LOW)
      repeat(35_000)
            `uvm_do(req)
    endtask

endclass : uabs_5_packets

```

Listing 32: Optimized ANS compressor, UVM testbench, uabs_tx_seqs

```
    class uabs_tx_sequencer extends uvm_sequencer #(uabs_packet);

    uabs_packet packet;

    `uvm_component_utils(uabs_tx_sequencer)

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    function void start_of_simulation_phase(uvm_phase phase);
        `uvm_info(get_type_name(), {"start of simulation for", get_full_name()}, 
    UVM_HIGH)
    endfunction : start_of_simulation_phase

endclass : uabs_tx_sequencer

```

Listing 33: Optimized ANS compressor, UVM testbench, uabs_tx_sequencer

D.2 UVM testbench

```
    // Clock generation, if necessary

```

Listing 34: Optimized ANS compressor, UVM testbench, clkgen

```
    module hw_top #(parameter WIDTH = 24, parameter FBITS = 4, parameter 
    CLK_DIVISOR = 4, parameter DUTY_DIVISOR = 4, parameter CLK_DIVISOR_WIDTH = 
    $clog2(CLK_DIVISOR+1));

    //import fsm1_pkg_b::*;
    //import uabs_enc_fsm_pkg::*;

    // Clock and reset signals
    logic [31:0]     clock_period;
    logic            clock;
    logic            reset;

    /* Temp signals for waveform purposes */
    logic start_enc_temp;
    logic start_dec_temp;
    logic vin_enc_temp;
    logic vin_dec_temp;
    logic rout_enc_temp;
    logic rout_dec_temp;
    //logic ack_dec_temp;
    logic rst_n_enc_temp;
    logic rst_n_dec_temp;
    logic [WIDTH-1:0] to_decode_temp;
    logic to_encode_temp;
    //logic decoder_en_temp;
    //logic encoder_en_temp;
    //logic idle_temp;
    //logic loop_done_enc_temp;
    //logic check_enc_temp;
    //logic checked_enc_temp;
    //logic bit_is_one_enc_temp;
    //logic comp_one_done_enc_temp;
    //logic comp_zero_done_enc_temp;
    //integer increment_temp;
    logic ready_enc_temp;
    //logic read_dec_temp;
    logic vout_enc_temp;
    logic vout_dec_temp;
    //logic encode_temp;
    //logic decode_temp;
    //logic decoded_temp;
    logic [WIDTH-1:0] encoded_temp;
    //uabs_enc_fsm_pkg::state_enc current_state_enc_temp;
    //fsm1_pkg_b::state_e_b current_state_dec_temp;
    logic [FBITS:0] probability_temp;



    // Added 6.7.2022
    logic [WIDTH-1:0] state_out_dec_temp;
    logic ready_dec_temp;
    logic decoded_bit_temp;
    logic [WIDTH-1:0] state_in_enc_temp;
    logic idle_enc_temp;
    logic div_clk_temp;




    // uabs Interface to the DUT
    uabs_if in0(
        .clock(clock),
        .reset(reset),
        .encoded(encoded_temp),
        .vout_enc(vout_enc_temp),
        .idle_enc(idle_enc_temp),
        .vout_dec(vout_dec_temp),
        .decoded_bit(decoded_bit_temp),
        .state_out_dec(state_out_dec_temp)
        );

    // channel Interfaces to the DUT
    //channel_if ch0(clock, reset);

    // CLKGEN module generates clock
    clkgen clkgen (
        .clock(clock),
        .run_clock(1'b1),
        .clock_period(32'd10)
    );

    design_top #(WIDTH, FBITS, CLK_DIVISOR, DUTY_DIVISOR, CLK_DIVISOR_WIDTH)
    design_top (
        .start_dec(start_dec_temp),
        .vin_dec(vin_dec_temp),
        .rout_dec(rout_dec_temp),
        //input logic clk,
        .rst_n_dec(rst_n_dec_temp),
        .to_decode(to_decode_temp),
        //input logic [FBITS:0] probability,
        .comp_state_out_dec(state_out_dec_temp),
        .ready_dec(ready_dec_temp),
        .vout_dec(vout_dec_temp),
        .decoded_bit(decoded_bit_temp),
        .start_enc(start_enc_temp),
        .vin_enc(vin_enc_temp),
        .rout_enc(rout_enc_temp),
        .clk(clock),
        //input logic div_clk,
        .rst_n_enc(rst_n_enc_temp),
        .to_encode(to_encode_temp),
        .probability(probability_temp),
        .comp_state_in_enc(state_in_enc_temp),
        .idle_enc(idle_enc_temp),
        .ready_enc(ready_enc_temp),
        .vout_enc(vout_enc_temp),
        .div_clk(div_clk_temp),
        .encoded(encoded_temp)
    );
    initial begin
        reset <= 1'b0;
        in0.in_suspend <= 1'b0;
        @(negedge clock)
            #1 reset <= 1'b1;
        @(negedge clock)
            #1 reset <= 1'b0;
    end

    /*
    initial begin
        rst_n_enc_temp <= 1'b1;
        @(posedge clock)
            #1 rst_n_enc_temp <= 1'b0;
            rst_n_dec_temp <= 1'b0;
        @(posedge clock)
            #1 rst_n_enc_temp <= 1'b1;
            rst_n_dec_temp <= 1'b1;
    end
    */

    /*
    initial begin
        for(int i = 0; i < 100; i++) begin
            @(negedge clock) begin
                encoder_en_temp = in0.encoder_en;
                decoder_en_temp = in0.decoder_en;
                to_encode_temp = in0.to_encode;
            end
        end
    end

    */
    always_comb begin
        //encoder_en_temp = in0.encoder_en;
        //decoder_en_temp = in0.decoder_en;
        to_encode_temp = in0.to_encode;
        to_decode_temp = in0.to_decode;
        start_enc_temp = in0.start_enc;
        start_dec_temp = in0.start_dec;
        vin_enc_temp = in0.vin_enc;
        vin_dec_temp = in0.vin_dec;
        rout_enc_temp = in0.rout_enc;
        rout_dec_temp = in0.rout_dec;
        state_in_enc_temp = in0.state_in_enc;
        probability_temp = in0.probability;
        rst_n_dec_temp = in0.rst_n_dec;
        rst_n_enc_temp = in0.rst_n_enc;

        //decoded_temp = in0.decoded;
    end
endmodule : hw_top

```

Listing 35: Optimized ANS compressor, UVM testbench, hw_top_no_dut

```
    // Code your testbench here
// or browse Examples

//`include "uabs_pkg.sv"
//`include "channel_pkg.sv"
//`include "uabs_if.sv"
```

```
//`include "channel_if.sv"
`include "clkgen.sv"
//`include "uabs_scoreboard.sv"
`include "hw_top_no_dut.sv"

module tb_top #(WIDTH = 24);

  import uvm_pkg::*;
  `include "uvm_macros.svh"
  //`include "uabs_pkg.sv"
  import uabs_pkg::*;
  //import channel_pkg::*;
  //`include "uabs_scoreboard.sv"
  `include "uabs_tb.sv"
  `include "uabs_test_lib.sv"
  `include "uabs_scoreboard.sv"
  //`include "uabs_scoreboard.sv"
  //`include "uabs_tb.sv"
  //`include "uabs_test_lib.sv"

  initial begin
    //uvm_config_db#(virtual mem_if)::set(uvm_root::get(), "*", "memif", memif);
    uabs_vif_config::set(null,"*.tb.uabs.tx_agent.*","vif", hw_top.in0);
    //channel_vif_config::set(null,"*.tb.chan0.*","vif", hw_top.ch0);
    // Possibly more channels here
    run_test("base_test");
  end

  /*
  initial begin
    begin
    clk = 0;
    #1 clk = ~clk;
   end end
   */

  initial begin
    void'($urandom(1500));
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end

endmodule : tb_top


```

Listing 36: Optimized ANS compressor, UVM testbench, testbench

```
    class uabs_tb extends uvm_env;

    // Component makro
    `uvm_component_utils(uabs_tb)

    // uabs env
    uabs_env uabs;

    //Channel env
    //channel_env chan0;

    // uabs scoreboard
    uabs_scoreboard uabs_sb;

    // Constructor
    function new (string name, uvm_component parent = null);
        super.new(name, parent);
    endfunction : new

    // uvm build phase
    function void build_phase (uvm_phase phase);
        super.build_phase(phase);

        // Uabs UVC
        uabs = uabs_env::type_id::create("uabs", this);

        // Channel UVC - RX ONLY
        //uvm_config_int::set(this, "chan0", "channel_id", 0);
        //chan0 = channel_env::type_id::create("chan0", this);

        /* Possibly more channels here */

        /*
         * More stuff
         */

        // uabs scoreboard
        uabs_sb = uabs_scoreboard::type_id::create("uabs_sb", this);
    endfunction : build_phase


    function void connect_phase(uvm_phase phase);

        // Connect the TLM ports fro mthe uABS and Channel UVCs to the scoreboard
        /* Remember to change the name from "agent" to "tx_agent" */
        uabs.agent.monitor.item_collected_port.connect(uabs_sb.sb_uabs_in);
        //chan0.rx_agent.monitor.item_collected_port.connect(uabs_sb.sb_chan0);

    endfunction : connect_phase

endclass : uabs_tb

```

Listing 37: Optimized ANS compressor, UVM testbench, uabs_tb

```
    class base_test extends uvm_test;
    // component macro
    `uvm_component_utils(base_test)

    uabs_tb tb;

    // constructor
    function new (string name, uvm_component parent = null);
        super.new(name, parent);
    endfunction : new

    // UVM build() phase
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
      uvm_config_wrapper::set(this, "tb.uabs.tx_agent.sequencer.run_phase",
                                    "default_sequence",
                                    uabs_5_packets::get_type());
        tb = new("tb", this);
    endfunction : build_phase

    // End of eloaboration phase
    function void end_of_elaboration_phase(uvm_phase phase);
        //uvm_top.print_topology();
    endfunction : end_of_elaboration_phase

    function void start_of_simulation_phase(uvm_phase phase);
        `uvm_info(get_type_name(), {"start of simulation for", get_full_name()},
    UVM_HIGH)
    endfunction : start_of_simulation_phase

endclass : base_test

class test_2 extends base_test;

    // component macro
    `uvm_component_utils(test_2)

    // constructor
    function new (string name, uvm_component parent = null);
        super.new(name, parent);
    endfunction : new

endclass : test_2
```

```
```

Listing 38: Optimized ANS compressor, UVM testbench, uabs_test_lib

Appendix E SystemVerilog software model, both encoder and decoder

```
    class Decoder #(int BIT_WIDTH = 8) extends Compression_base #(BIT_WIDTH);
        /** ANS Compressor Decoder
        * Decodes an integer using the uABS variant of ANS.
        * @param [in] to_decode is the integer (state) that is going to be decoded.
        * @param [in] probability of ones in the dataset.
        * @returns decoded which is the reproduced representation.
        */
        function logic [BIT_WIDTH - 1:0] decode_integer(int to_decode, real probability);
            integer state_part_a; // Part sum A of the state
            integer state_part_b; // Part sum B of t he state
            integer state = 0;    // The state (the rest of the encoded message)
            logic [BIT_WIDTH - 1:0] decoded; // The bits that have been decoded
            begin
            // Sequential logic due to dependences
                state = to_decode;
                // Loop until all bits are reproduced
                for (int ii = BIT_WIDTH - 1; ii >= 0; --ii) begin
                    // Calculate part sums then the bit is found.
                    state_part_a = $ceil((state + 1) * probability);
                    state_part_b = $ceil(state * probability);
                    decoded[ii] = state_part_a - state_part_b;
                    // Determine the remaining state
                    if(decoded[ii]) begin
                        state = $ceil(state * probability);
                    end
                    else begin
                        state = state - $ceil(state * probability);
                    end
                end
                // Reverse the order of bits in the array, MSB becomes LSB.
                decoded[BIT_WIDTH - 1:0] = reverse_array(decoded[BIT_WIDTH - 1:0]);
                // All bits are found, return the decoded representation.
                return decoded[BIT_WIDTH - 1:0];
            end
        endfunction
    endclass


```

Listing 39: SystemVerilog software model, design

```
    class Compression_base #(BIT_WIDTH = 8);
     /** Reverses the array
        * @param [in] data to be reversed.
        * @returns data is the reversed data.
        */
      function int reverse_array(logic [BIT_WIDTH - 1:0] data);
        begin
          data = {<<{data}};
        end
          return data;
      endfunction

        /** Calculates the probability of ones.
        * Uses the total BIT_WIDTH to calculate the probabilty
        * @param [in] dataset to be analyzed.
        * @returns probability of ones.
        */
      function real calculate_probability(logic [BIT_WIDTH - 1:0] dataset);
        real probability = 0;
        begin
          for (int ii = 0; ii < BIT_WIDTH; ++ii)
            probability = probability + dataset[ii];
          probability /= BIT_WIDTH;
        end
        return probability;
      endfunction

    endclass

```

Listing 40: SystemVerilog software model, Compression_base

```
      class Decoder #(int BIT_WIDTH = 8) extends Compression_base #(BIT_WIDTH);
      /** ANS Compressor Decoder
      * Decodes an integer using the uABS variant of ANS.
      * @param [in] to_decode is the integer (state) that is going to be decoded.
      * @param [in] probability of ones in the dataset.
      * @returns decoded which is the reproduced representation.
      */
      function logic [BIT_WIDTH - 1:0] decode_integer(int to_decode, real
  probability);
          integer state_part_a; // Part sum A of the state
          integer state_part_b; // Part sum B of t he state
          integer state = 0;    // The state (the rest of the encoded message)
          logic [BIT_WIDTH - 1:0] decoded; // The bits that have been decoded
          begin
          // Sequential logic due to dependences
              state = to_decode;
              // Loop until all bits are reproduced
              for (int ii = BIT_WIDTH - 1; ii >= 0; --ii) begin
                  // Calculate part sums then the bit is found.
                  state_part_a = $ceil((state + 1) * probability);
                  state_part_b = $ceil(state * probability);
                  decoded[ii] = state_part_a - state_part_b;
                  // Determine the remaining state
                  if(decoded[ii]) begin
                      state = $ceil(state * probability);
                  end
                  else begin
                      state = state - $ceil(state * probability);
                  end
              end
              // Reverse the order of bits in the array, MSB becomes LSB.
              decoded[BIT_WIDTH - 1:0] = reverse_array(decoded[BIT_WIDTH - 1:0]);
              // All bits are found, return the decoded representation.
              return decoded[BIT_WIDTH - 1:0];
          end
      endfunction
  endclass


```

Listing 41: SystemVerilog software model, Decoder

```
  class Encoder #(int BIT_WIDTH = 8) extends Compression_base #(BIT_WIDTH);
      /** ANS Compressor Encoder
      * Encodes a binary vector using the uABS variant of ANS.
      * @param [in] to_encode is the binary vector that is going to be encoded.
      * @param [in] probability of ones in the dataset.
      * @returns state which is the encoded representation.
      */
      function int encode_binary(logic[BIT_WIDTH - 1:0] to_encode, real
  probability);
          integer state = 1; // The state is the (growing per iteration) encoded
  representation
          begin
              // Sequential logic due to dependences
              // Loop until all bits are reproduced
              for (int ii = BIT_WIDTH - 1; ii >= 0; --ii)
              begin
                  // Calculate the state
                  if(to_encode[ii])
                  begin
```

```
                        state = $floor(state / probability);
                    end
                    else
                    begin
                        state = $ceil((state + 1)/ (1- probability)) - 1;
                    end
                end
                return state;
            end
            endfunction
    endclass

```

Listing 42: SystemVerilog software model, Encoder

```
    module uabs_decoder_tb #(WIDTH = 8);
      integer to_decode;
      logic [WIDTH - 1:0] to_encode;
      logic decoder_en;
      logic encoder_en;
      real probability;
      logic clk;
      logic [WIDTH - 1:0] decoded;
      integer encoded;

      uabs_compressor #(WIDTH) uabs_compressor (.*);

      initial forever
        #1 clk = ~clk;

      initial begin
        //void'($urandom(1500));
        $dumpfile("dump.vcd");
        $dumpvars(1);
        clk = 0;

        // Drive stimuli
        to_encode = 8'b1000_0000;
        decoder_en = 0; encoder_en = 0; #1
        print_compression_data;

        // Start encoding
        decoder_en = 0; encoder_en = 1; #1
        print_compression_data;

        // Drive stimuli
        to_decode = encoded;
        decoder_en = 0; encoder_en = 0; #1
        print_compression_data;

        // Start decoding
        decoder_en = 1; encoder_en = 0; #1
        check_compression;

        $finish;
        //to_encode = 0; probability = 0.5;#1;
      end


      task print_compression_data;
        $display("to_encode:%0b, to_decode:%0b, probability:%0f, decoder_en:%0b,
    encoder_en:%0b", to_encode, to_decode, probability, decoder_en, encoder_en);
      endtask

      task check_compression;
        if (to_encode == decoded)
          $display("Compression successful");
        else
          $display("Compression failed");
      endtask

    endmodule
```

```
```
Listing 43: SystemVerilog software model, testbench

Appendix F C++ software model, both encoder and decoder

Note: Some sections of the software C++ implementation are based on work previously performed in the project thesis. While this master thesis has developed a new Object oriented implementation, some features such as the arithmetic is largely the same as in the project thesis, because of uABS still being the same algorithm in both. See the function oriented implementation of ANS which was previously developed at [6].

```
#include "header.h"
#include "Decoder.h"
#include "Encoder.h"
#include "Simulator.h"

int main() {

  // Stimuli
  bitset<BIT_WIDTH> original(136);
  bitset<BIT_WIDTH> original_2(243);
  bitset<BIT_WIDTH> original_3(31);
  vector<std::bitset<BIT_WIDTH>> binary_vec;
  int start = 1;

  binary_vec.push_back(11000000);
  binary_vec.push_back(10000000);
  binary_vec.push_back(10000001);
  binary_vec.push_back(10000010);

  // Object
  Simulator simulator(start, "../../stimuli.txt", "../../encode.txt", "../..
decode.txt");

  // Run
  simulator.batch_run_and_print_to_file();

  // End
    return 0;
}

```
Listing 44: ANS software c++ model, main

```
#include "header.h"
#include "Compression_base.h"
#include "Decoder.h"

Decoder::Decoder(int start)  // Constructor with parameters
  : Compression_base{ start }
{
}

/** Converts a natural number to a binary sequence.
* Decoding operation for a uABS.
*
* @param [in] state is the natural number that is going to be decoded.
* @param [in] increment is the number increments necesary for the main for body
.
* @param [in] probability_of_one is the probability for ones in the model.
* @param [in] starting number is the starting natural number of the conversion,
 '1' is the standard.
* @returns decoded_binary_sequence which is the decoded binary sequence.
*/
std::bitset<BIT_WIDTH> Decoder::decode_uabs(long state, double
probability_of_one) {
  int string_length = BIT_WIDTH;
  std::bitset<BIT_WIDTH> decoded_binary_sequence(0);;
  std::clog << "String Length: " << string_length << std::endl;

  for (int ii = string_length - 1; ii >= 0; --ii) {
    /* Binary bit */
    long part_1 = ceil(((double)state + 1) * probability_of_one);
    long part_2 = ceil(state * probability_of_one);
    decoded_binary_sequence[ii] = part_1 - part_2;
    std::clog << "b_" << ii + 1 << " = (" << state << " + 1 * " <<
probability_of_one << ") - ("
      << state << " * " << probability_of_one << ") = " << part_1 << " - " <<
part_2 << " = " << decoded_binary_sequence[ii] << std::endl;

    /* Natural number */
    if (decoded_binary_sequence[ii]) {
      std::clog << "(x_" << ii << ") = " << state << " * " <<
probability_of_one << " = ";
      state = ceil(state * probability_of_one);
      std::clog << state << std::endl;
    }

    else {
      state = (state - ceil(state * probability_of_one));
      std::clog << "(x_" << ii << ") = " << state << " - (" << state << " * "
<< probability_of_one << ") = " << state << std::endl;
    }
  }
  return binary_reverse(decoded_binary_sequence);
}

```

Listing 45: ANS software c++ model, decoder class

```
#ifndef DECODER_H
#define DECODER_H

#include "header.h"
#include "Compression_base.h"

class Decoder : public Compression_base {
public:
  std::bitset<BIT_WIDTH> decode_uabs(long natural_number, double
probability_of_one);
  Decoder(int start);
};

#endif


```

Listing 46: ANS software c++ model, header for the decoder class

```
#include "header.h"
#include "Compression_base.h"
#include "Encoder.h"


/** Converts a binary sequence to a natural number.
 * Encoding operation for a uABS.
 *
 * @param [in] binary_sequence is the binary string that is going to be encoded
.
 * @param [in] increment is the number increments necesary for the main for
body.
 * @param [in] probability_of_one is the probability for ones in the model.
 * @param [in] starting number is the starting natural number of the conversion
, '1' is the standard.
 * @returns state which is the encoded natural number.
 */
long Encoder::encode_uabs(std::bitset<BIT_WIDTH> binary_sequence, double
probability_of_one) {

  long state = get_starting_number();
  long string_length = BIT_WIDTH;


  std::clog << "Probability of ones: " << probability_of_one << std::endl;

  for (int ii = string_length - 1; ii >= 0; --ii) {

    if (binary_sequence[ii]) {
      std::clog << "x_" << string_length - ii << " =  2 * " << state << " / "
<< probability_of_one;
      state = floor(state / probability_of_one);
    }
    else {
      std::clog << "x_" << string_length - ii << " =  (" << state << " + 1) / (
" << "1" << -probability_of_one << ") - 1";
      state = ceil(((double)state + 1) / (1 - (double)(probability_of_one))) -
1;
    }

    std::clog << " = " << state << std::endl;

  }
  return state;
}

Encoder::Encoder(int start)  // Constructor with parameters
  : Compression_base {start}
{
}



```
Listing 47: ANS software c++ model, encoder class

```
#ifndef ENCODER_H
#define ENCODER_H

#include "header.h"
#include "Compression_base.h"

class Encoder : public Compression_base {
public:
  long encode_uabs(std::bitset<BIT_WIDTH> binary_sequence, double
probability_of_one);
  Encoder(int start);
};

#endif


```
Listing 48: ANS software c++ model, header for the encoder class

```

#include "Compression_base.h"
#include "header.h"

Compression_base::Compression_base(int start)
  : starting_number {start}
{
}

/** Reverses (flips) a bit sequence.
 * For instance MSB becomes LSB, and LSB becomes MSB.
 * @param [in] binary_number is the number to be reversed.
 * @returns binary_number which is the flipped bit sequence.
 */
std::bitset<BIT_WIDTH> Compression_base::binary_reverse(std::bitset<BIT_WIDTH>
binary_number, int offset) {
  for (int ii = 0; ii < (BIT_WIDTH / 2); ++ii) {
    bool temp = binary_number[ii];
    binary_number[ii] = binary_number[BIT_WIDTH - ii - 1];
    binary_number[BIT_WIDTH - ii - 1] = temp;
  }
  return (binary_number >> offset);
}

void Compression_base::int_print_to_file(int integer, bool line_terminator, bool end_line, std::string file_name) {  // Method/function defined inside the class
  ofstream myfile;
  /* Print integer */
  myfile.open(file_name, fstream::app);
  myfile << integer;

  /* Line termination */
  if (line_terminator) {
    myfile << LINE_TERMINATOR;
  }

  /* New Line */
  if (end_line) {
    myfile << NEW_LINE;
  }

  myfile.close();
}

void Compression_base::binary_print_to_file(std::bitset<BIT_WIDTH> binary, bool line_terminator, bool end_line, std::string file_name) {  // Method/function defined inside the class
  ofstream myfile;
  //myfile.open("example.txt");
  myfile.open(file_name, fstream::app);
  myfile << binary;

  /* Line termination */
  if (line_terminator) {
    myfile << LINE_TERMINATOR;
  }

  /* New Line */
  if (end_line) {
    myfile << NEW_LINE;
  }

  myfile.close();

}

void Compression_base:: string_print_to_file(std::string text, bool line_terminator, bool end_line, std::string file_name) {  // Method/function defined inside the class
  ofstream myfile;
  //myfile.open("example.txt");
  myfile.open(file_name, fstream::app);
  myfile << text;

  /* Line termination */
  if (line_terminator) {
    myfile << LINE_TERMINATOR;
  }

  /* New Line */
  if (end_line) {
    myfile << NEW_LINE;
  }

  myfile.close();
}


void Compression_base::clear_file(std::string filename) {
  std::ofstream ofs;
```

```cpp
  ofs.open(filename, std::ofstream::out | std::ofstream::trunc);
  ofs.close();
}





```

Listing 49: ANS software c++ model, base compression class

```cpp
#ifndef COMPRESSION_BASE_H
#define COMPRESSION_BASE_H
#include "header.h"

class Compression_base {
private:
  int starting_number{};
public :
  Compression_base(int start);
  int get_starting_number() const { return starting_number; }
  std::bitset<BIT_WIDTH> binary_reverse(std::bitset<BIT_WIDTH> binary_number, int offset = 0);
  void int_print_to_file(int integer, bool line_terminator = false, bool end_line = false, std::string file_name = DEFAULT_ENCODER_FILENAME);
  void binary_print_to_file(std::bitset<BIT_WIDTH> binary, bool line_terminator = false, bool end_line = false, std::string file_name = DEFAULT_ENCODER_FILENAME);
  void string_print_to_file(std::string text, bool line_terminator = false, bool end_line = false, std::string file_name = DEFAULT_ENCODER_FILENAME);
  void clear_file(std::string filename);
};

#endif

```

Listing 50: ANS software c++ model, header for the base compression class

```cpp
#include <iostream>
#include <fstream>
#include <bitset>
#include <limits>
#include <sstream>
#include <vector>

using namespace std;

/* Compression related */
#define BIT_WIDTH 8
#define STARTING_NUMBER_DEFAULT 1

/* Print related */
#define DEFAULT_ENCODER_FILENAME "../../encoder.txt"
#define LINE_TERMINATOR ";"
#define NEW_LINE "\n"

```

Listing 51: ANS software c++ model, header

```cpp
#include"header.h"
#include"simulator.h"

Simulator::Simulator(int start, std::string stimuli_file, std::string encoder_file, std::string decoder_file)
  : Decoder{ start},
  Encoder{ start}
{
}

/* Semi my function*/
std::fstream& select_line_in_file(std::fstream& filename, int line_number) {
    filename.seekg(std::ios::beg);
```

```cpp
        for (int ii = 0; ii < line_number - 1; ii++) {
            filename.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        }
        return filename;
    }

    /* Not my function */
    std::string extract_string_until(std::string const& s)
    {
        std::string::size_type pos = s.find(';');
        if (pos != std::string::npos)
        {
            return s.substr(0, pos);
        }
        else
        {
            return s;
        }
    }

    std::vector<int> extract_vector_from_file(std::string filename) {
        std::fstream file(filename);

        int ii = 1;
        std::vector <int> vect;
        while (file.peek() != EOF)
        {
            int my_integer;
            std::string text;

            /* Find and extract line */
            select_line_in_file(file, ii);
            file >> text;
            text = extract_string_until(text);
            std::stringstream integer(text);
            integer >> my_integer;

            /* Insert into vector */
            vect.push_back(my_integer);
            std::cout << vect[ii - 1] << std::endl;
            ii++;
        }
      std::cout << "End of file";
        return vect;
    }

    std::vector<std::bitset<BIT_WIDTH>> extract_binary_vector_from_file(std::string
     filename) {
        std::fstream file(filename);

        int ii = 1;
        std::vector <std::bitset<BIT_WIDTH>> vect;
        while (file.peek() != EOF)
        {
            std::bitset<BIT_WIDTH> my_binary;
            std::string text;

            /* Find and extract line */
            select_line_in_file(file, ii);
            file >> text;
            text = extract_string_until(text);
            std::stringstream integer(text);
            integer >> my_binary;

            /* Insert into vector */
            vect.push_back(my_binary);
            std::cout << vect[ii - 1] << std::endl;
            ii++;
        }
        std::cout << "End of file";
        return vect;
    }

```

```cpp
int Simulator::batch_run_and_print_to_file() {
  /* Run Compresion */

  vector <int> natural_vector;
  vector <bitset<BIT_WIDTH>> binary_vector;
  vector <int> natural_vector_2;

    vector <bitset<BIT_WIDTH>> binary_input_vector_from_file;

  Compression_base::clear_file("../../encoder.txt");
  Compression_base::clear_file("../../decoder.txt");
    binary_input_vector_from_file = extract_binary_vector_from_file("../../stimuli.txt");

  std::cout << "Inputs" << std::endl;
  double probability = 0;
  for (int ii = 0; ii < binary_input_vector_from_file.size(); ++ii) {
    probability += binary_input_vector_from_file[ii].count();
    std::cout << "Binary: " << binary_input_vector_from_file[ii] << " ones:" << binary_input_vector_from_file[ii].count() << std::endl;
  }

    probability = probability / (binary_input_vector_from_file.size() * BIT_WIDTH);

  std::cout << std::endl << "Overal p of ones" << probability << std::endl;

  for (int ii = 0; ii < binary_input_vector_from_file.size(); ++ii) {
    natural_vector.push_back(encode_uabs(binary_input_vector_from_file[ii], probability));
    Compression_base::int_print_to_file(natural_vector[ii], true, true, "../../encoder.txt");
    cout << "Encoded" << natural_vector[ii] << std::endl;
  }

  natural_vector_2 = extract_vector_from_file(".. / .. /encoder.txt");;

  for (int ii = 0; ii < natural_vector.size(); ++ii) {
    binary_vector.push_back(decode_uabs(natural_vector[ii], probability));
    Compression_base::binary_print_to_file(binary_vector[ii], true, true, "../../decoder.txt");
  }

  return 1;
}



```

Listing 52: ANS software c++ model, simulator class

```cpp
#ifndef SIMULATOR_H
#define SIMULATOR_H
#include "header.h"
#include "Decoder.h"
#include "Encoder.h"

class Simulator : public Encoder, Decoder {
private:
  std::string encoder_file_name;
  std::string decoder_file_name;
public:
  int batch_run_and_print_to_file();
  Simulator(int start, std::string stimuli_file, std::string encoder_file, std::string decoder_file);
};

#endif

```

Listing 53: ANS software c++ model, header for the simulator class

Appendix G   Unoptimized ANS compressor

Only the decoder FSM is shown, as the Optimized ANS compressor is the preferred implementation.

```
    //`include "uabs_decoder_fsm_pkg.sv"
import uabs_decoder_fsm_pkg::*;

module uabs_decoder_fsm (
    input logic start,
    input logic vin,
    input logic checked,
    input logic bit_is_one,
    input logic comp_one_done,
    input logic comp_zero_done,
    input logic rout,
    input logic clk,
    input logic rst_n,
    input logic comp_ready,
    output logic rin,
    output logic vout,
    output logic decode,
    output logic check,
    output logic comp_busy,
    output logic idle_dec,
    output logic comp_init);

    state_dec state, next;

    always_ff @(posedge clk, negedge rst_n)
        if (!rst_n)
            state <= IDLE_DEC;
        else
            state <= next;

    always_comb begin
        next = XXX_DEC; //@LB next = state;
        rin = 1'b0;
        vout = 1'b0;
        decode = 1'b0;
        check = 1'b0;
        idle_dec = 1'b0;
        comp_busy = 1'b0;
        comp_init = 1'b0;
        case (state)
            IDLE_DEC :  begin
                        idle_dec = 1'b1;
                    if (start)
                        next = READ_DEC;
                      else
                        next = IDLE_DEC;
                    end
            READ_DEC :  begin
                        rin = 1'b1;
                        if (vin)
                          next = CHECK_DEC;
                      else
                          next = READ_DEC;
                        end
            CHECK_DEC :  begin
                        check = 1'b1;
                        if (checked && !bit_is_one)
                            next = COMP_ZERO_INIT;
                        else if (checked && bit_is_one)
                            next = COMP_ONE_INIT;
                        else
                            next = CHECK_DEC;
                        end
            COMP_ONE_INIT :  begin
                        decode = 1'b1;
                        comp_init = 1'b1;
                        if (comp_ready)
                            next = COMP_ONE_BUSY;
                        else
                            next = COMP_ONE_INIT;
                        end
            COMP_ONE_BUSY :  begin
                        decode = 1'b1;
                        comp_busy = 1'b1;
                        if (comp_one_done)
                            next = DONE_DEC;
                        else
                            next = COMP_ONE_BUSY;
                        end
            COMP_ZERO_INIT :  begin
                        decode = 1'b1;
                        comp_init = 1'b1;
                        if (comp_ready)
                            next = COMP_ZERO_BUSY;
                        else
                            next = COMP_ZERO_INIT;
                        end
            COMP_ZERO_BUSY :  begin
                        decode = 1'b1;
                        comp_busy = 1'b1;
                        if (comp_zero_done)
                            next = DONE_DEC;
                        else
                            next = COMP_ZERO_BUSY;
                        end
            DONE_DEC :  begin
                        vout = 1'b1;
                        if (rout)
                            next = IDLE_DEC;
                        else
                            next = DONE_DEC;
                        end
            default:begin
                        rin = 'x;
                        vout = 'x;
                        decode = 'x;
                        check = 'x;
                        next = XXX_DEC;
                    end
        endcase
    end
endmodule : uabs_decoder_fsm


```
Listing 54: Unoptimized ANS compressor, FSM