Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

12-2019

Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures

Thomas Cenova
twc9519@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

Cenova, Thomas, "Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures" (2019). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures

Thomas Cenova

Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures

THOMAS CENOVA

December 2019

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

R·I·T | KATE GLEASON
College of ENGINEERING

Department of Computer Engineering

Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures

Thomas Cenova

Committee Approval:

Dr. Sonia López Alarcón *Advisor* Date
RIT, Department of Computer Engineering

Dr. Marcin Łukowiak Date
RIT, Department of Computer Engineering

Dr. Andrés Kwasinski Date
RIT, Department of Computer Engineering

Abstract

Software defined radio (SDR) platforms implement many digital signal processing algorithms. These can be accelerated on an FPGA to meet performance requirements. Due to the flexibility of SDR's and continually evolving communications protocols, high level synthesis (HLS) is a promising alternative to standard handcrafted design flows. A crucial component in any SDR is the error correction codes (ECC). Turbo codes are a common ECC that are implemented on an FPGA due to their computational complexity. The goal of this thesis is to explore the HLS coding techniques required to produce a design that targets the desired hardware architecture and can reach handcrafted levels of performance.

This work implemented three existing turbo decoder architectures with HLS to produce quality hardware which reaches handcrafted performance. Each targeted design was analyzed to determine its functionality and algorithm so a C implementation could be developed. Then the C code was modified and HLS directives were added to refine the design through the HLS tools. The process of code modification and processing through the HLS tools continued until the desired architecture and performance were reached.

Each design was implemented and the bottlenecks were identified and dealt with through appropriate usage of directives and C style. The use of pipelining to bypass bottlenecks added a small overhead from the ramp-up and ramp-down of the pipeline, reducing the performance by at most 1.24%. The impact of the clock constraint set within the HLS tools was also explored. It was found that the clock period and resource usage estimate generated by the HLS tools is not accurate and all evaluations should occur after hardware synthesis.

Contents

Signature Sheet i

Abstract ii

Table of Contents iii

List of Figures v

List of Tables 1

1 Introduction 2

1.1 Motivation . . . 2

2 Background 6

2.1 Software Defined Radios . . . 6

2.1.1 SDR Platforms and Libraries . . . 8

2.2 Turbo Codes . . . 11

2.2.1 Turbo Encoding . . . 12

2.2.2 Turbo Decoding . . . 13

2.3 High Level Synthesis . . . 18

2.3.1 HLS Development vs Software Development . . . 19

2.3.2 Limitations of HLS Tools . . . 23

2.3.3 HDL Design Flow vs HLS Design Flow . . . 25

2.4 Related Work . . . 27

3 Methodology 30

3.1 HLS Design Considerations . . . 30

3.1.1 HLS Design Flow . . . 30

3.1.2 Vivado HLS Steps . . . 32

3.1.3 Directives . . . 33

3.1.4 Data Types . . . 36

3.1.5 Memory Design . . . 37

3.1.6 Coding Structures . . . 38

3.2 Clock Constraint Exploration . . . 40

3.3 Methodology . . . 40

4 Architectures **43**
4.1 Design 1: Sequential Decoder 43
4.1.1 Handcrafted Serial Architecture 43
4.1.2 Serial Decoder HLS Implementation 45
4.1.3 Refining the Design 48
4.2 Design 2: Parallel Decoder 50
4.2.1 Handcrafted Parallel Architecture (HPA) 50
4.2.2 Parallel Decoder HLS Implementation 53
4.3 Design 3: Parallel Decoder with Double Buffering 56
4.3.1 Double Buffering Handcrafted Architecture (DBHA) 56
4.3.2 Double Buffering HLS Implementation 58
4.4 Summary of Designs 60

5 Results and Analysis **61**
5.1 Clock Constraint Impact on Designs 66

6 Conclusion and Future Work **74**
6.1 Conclusions 74
6.2 Future Work 75

Bibliography **77**

List of Figures

1.1 Design flow of an SDR utilizing HLS and open source libraries 3

2.1 Components of a modern communications system with areas marked as programmable implemented with software or reconfigurable hardware [1] 6
2.2 Turbo encoder 12
2.3 LTE turbo decoder [2] 14
2.4 LTE turbo code trellis [2] 16
2.5 High level mapping of CPU architectures and code execution to a hardware architecture 19
2.6 General HLS algorithm data flow to produce quality results 20
2.7 Simple multiplexer example of sharing an adder hardware resource . . 22
2.8 General handcrafted RTL hardware design flow for an FPGA 25
2.9 General HLS design flow from hardware oriented software to a high performance hardware design 26

3.1 HLS design flow 31
3.2 Array partitioning effect 34
3.3 Array reshaping effect [3] 35
3.4 General design flow with multiple memory accesses 39
3.5 Methodology used to replicate handcrafted turbo decoder designs with HLS 41

4.1 Serial turbo decoder block diagram [2] 43
4.2 Serial turbo decoder timing diagram [2] 44
4.3 Data flow and looping structure of initial serial turbo decoder architecture modeled in C 48
4.4 Improved data flow and looping structure of serial turbo decoder architecture modeled in C 50
4.5 Parallel turbo decoder timing diagram [2] 51
4.6 Parallel turbo decoder memory architecture [2] 52
4.7 Parallel turbo decoder memory access architecture with a parallelism of two 53
4.8 Parallel turbo decoder final architecture block diagram 55
4.9 Parallel extrinsic calculation [4] 56

4.10 Parallel forward and backward metric calculations [4] 57
4.11 Double buffering of turbo decoder [4] 57
4.12 HLS turbo decoder block diagram for double buffering 58

5.1 Throughput comparison between handcrafted and HLS implementations 63
5.2 Throughput comparison between handcrafted and HLS implementations with alternate latency for Design 3 65
5.3 Graph of the clock constraints impact on the designs minimum clock period reported after HLS synthesis and implementation 69
5.4 Bar graph of the clock constraints impact on the FF usage reported after HLS synthesis and implementation. 70
5.5 Bar graph of the clock constraints impact on the LUT usage reported after HLS synthesis and after implementation. 71
5.6 Graph of the implemented clock period's impact on maximum throughput for each design. The block size is 6144 and the parallelism is 8 for Design 2 and Design 3. 73

List of Tables

4.1 A summary of all design architectures 60

5.1 Results for all handcrafted and HLS designs 62
5.2 Alternate handcrafted throughput for Design 3 based on the latency of each iteration shown in Figure 4.11 64
5.3 Comparison of resource usage between HLS designs 65
5.4 Design 1 HLS and implementation results for clock constraints from 2ns to 10ns with a block size of 6144 67
5.5 Design 2 HLS and implementation results for clock constraints from 2ns to 10ns with a block size of 6144 with a parallelism of 8 68
5.6 Design 3 HLS and implementation results for clock constraints from 2ns to 10ns with a block size of 6144 and parallelism of 8 68

Chapter 1

Introduction

1.1 Motivation

Wireless communication systems are continually evolving to meet new standards with greater throughput over increasingly noisy channels. Software defined radios (SDRs) provide a current solution to this problem by allowing the radio to be upgraded over time with new, better algorithms. This can allow an exiting hardware platform to adapt to new standards and implement new protocols. Error correction codes (ECC) are an integral component in wireless communications systems that require reliable data transmissions. Forward error correction codes (FECs) include extra data in the initial transmissions to allow for error correction, rather than relying on retransmission which can be costly and undesired. Turbo codes are a common FEC used in high performance and noisy environments such as deep space communications [5] and long time evolution (LTE) mobile networks [6] due to their flexibility in performance, throughput, and resource usage.

Classically, SDR implementations on an FPGA are handcrafted using a hardware description language (HDL) to describe the desired architecture at the register transfer level (RTL). This poses a challenge with implementation since the process can be very time consuming and requires specialized hardware engineers with the knowledge of the language, tools, and hardware platform being targeted. An alternative to the

standard RTL design approach exists through the use of high level synthesis (HLS). HLS converts a high level language (HLL) such as C into an RTL hardware model. This process allows for reduced development times and a simplified design process, just as the C language and compiler did with assembly programming. The development flow for implementing an SDR on an FPGA is shown in Figure 1.1. This diagram shows how a software library that is compatible with HLS tools could be tuned and then used to implement an SDR platform without the need for specialized hardware knowledge. SDR platforms are commonly implemented on system on a chips (SoCs) with CPU cores and FPGA fabric. Currently there are open source libraries for SDRs which are optimized for CPU and SIMD architectures, however, no such library exists for development on SoCs with CPUs and FPGA fabric. The use of architecture specific optimizations and reliance on complex data structures involving dynamic pointers within the current libraries cause incompatibility with current HLS tools. This results in the need for a new library with the goal of software and hardware support via HLS.

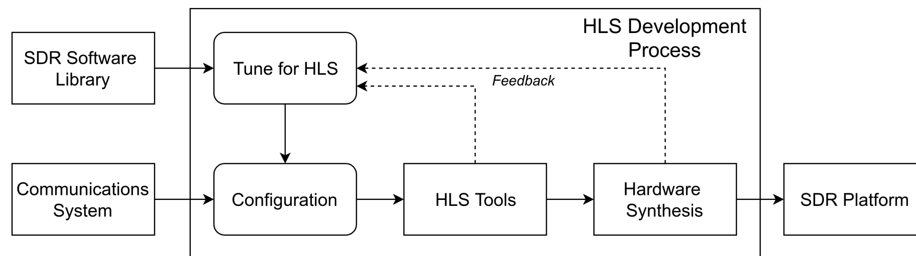


Figure 1.1: Design flow of an SDR utilizing HLS and open source libraries

HLS has the unique ability to bridge the gap between software and hardware development. From the perspective of a software domain, HLS can accelerate designs by allowing simplified hardware and software co-design. This has especially become prevalent in the cloud [7]. ie *Amazon* provides FPGAs for use with their computing nodes. This has been made accessible by offering similar development environments to software engineers via C/C++ and OpenCL, made possible by HLS tools. In the

hardware domain, HLS can be used as an alternative to HDLs to generate high performance designs. Utilizing HLS to replicate the results of an RTL design does come with the challenge of representing the desired hardware architecture in an HLL. HLLs do not describe how an operation is implemented in hardware, only its algorithmic function. Because of this, HLS tools must extract the data and control flow from an algorithm and generate hardware which implements it. This process can lead to inefficiencies since the tools may implement an algorithm with generic control logic to simplify the RTL generation process. To use HLS effectively, the code structure and design process are crucial to achieving quality hardware.

A case study of prior research shows that handcrafted RTL design flows still produce higher performing hardware than HLS [8]. The designs which were studied varied in how HLS was used, either to speedup software or to target the desired hardware architecture. Other past works also focused on using HLS to improve a software algorithm [9, 10]. The goal of these works was to use HLS optimizations and C style changes to produce high performance hardware. This approach can lead to bottlenecks in the design if there are software constructs that do not translate well into hardware. Without guiding the HLS tools to the desired hardware architecture, these bottlenecks can heavily impact the final performance and need to be mitigated.

This work explored the potential of an open source SDR library for HLS development and the design process required to produce high performance hardware from HLS. This was accomplished by modeling three existing handcrafted RTL turbo decoder architectures in C from a hardware engineer's perspective and using HLS to generate hardware which reaches RTL levels of performance. The resulting hardware architecture was analyzed for bottlenecks imposed by the HLS tools. These bottlenecks were then mitigated with code changes and HLS specific optimizations. By designing with a hardware architecture in mind, and then applying HLS optimizations, this work explored an alternate design approach to designing with HLS for

high performance designs. This is especially beneficial to turbo decoders, and SDRs in general due to the increased flexibility, decreased implementation difficulty, and fast time to market which an HLS design flow can provide.

Chapter 2

Background

2.1 Software Defined Radios

Software defined radios are platforms for implementing wireless communication systems with a combination of hardware and software. In an SDR, components that are classically implemented in hardware are instead executed on a processing device such as a CPU or FPGA. This allows an SDR to be more flexible than radio systems built with application specific integrated circuits (ASICs) or discrete components. The basic components of an SDR can be found in Figure 2.1.

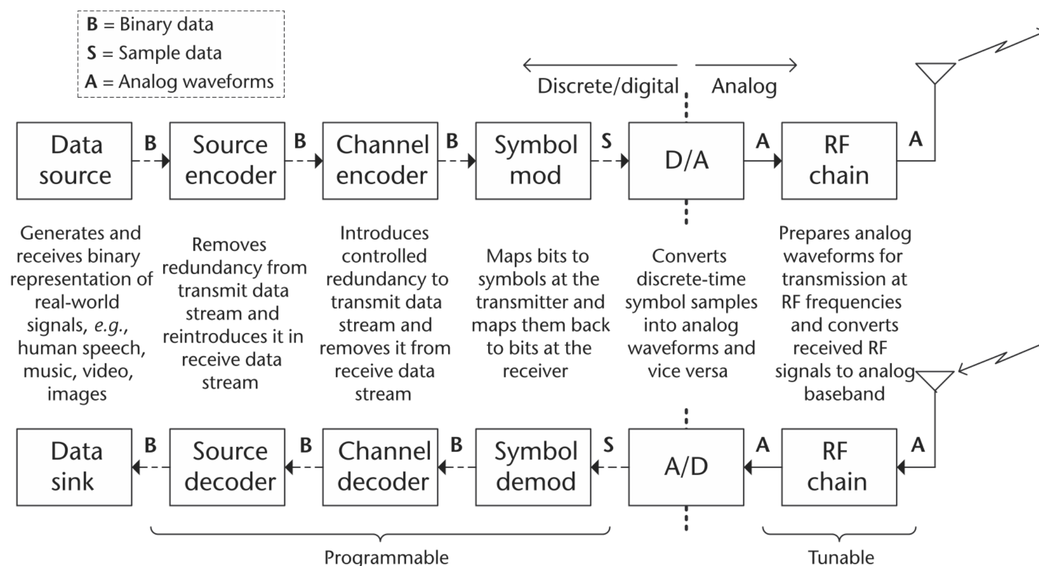


Figure 2.1: Components of a modern communications system with areas marked as programmable implemented with software or reconfigurable hardware [1]

Figure 2.1 shows the main components of a transmitter/receiver radio system. The sections labeled programmable are implemented in a processor system through software or a reconfigurable hardware system. The RF chain is also tunable in an SDR. This leads to a flexible system which can adapt and be upgraded to future communication systems. This can be especially useful for continuously evolving systems, systems with the need for upgradeability, and research and development of communication systems. In addition, as radio technology advances, remote systems are able to stay updated without requiring a hardware replacement.

Ground and in-flight space communications have utilized SDRs. For example, the Martian Curiosity rover utilized an Elextra and Elextra-lite SDR which allowed for communications systems to be upgraded in phases [11]. In phase 1 a conservative approach was used to communicate at 32kbps. Phase 2 upgraded communications to 128kbps and 256kbps, and finally phase 3 implemented Adaptive Data Rate and suppressed carrier modulation to increased the communications rate up to 2048kbps, doubling the mission requirements for data transfer for each day on mars. The unique ability of an SDR allows the radio to be tuned and adapt while in space. Similarly, the work presented in [12] discusses other SDR platforms are being created for onboard space applications by the European Space Agency. Not only are SDRs an ideal choice for remote systems that must be updated remotely, but they also have benefits for ground applications. Flexibility is still a benefit to allow for the evolution of algorithms and protocols vie loading new software on an existing hardware platform. SDRs also bring adaptability, easy integration, reduced time to market, reduced cost, lower obsolescence and economics of scale [12].

A key component in many communications systems is Error Correction Codes (ECC) which ensures that data is reliably transmitted. In Figure 2.1 the ECC component is found within the channel encoding block where the data has redundant information added and is then encoded into a final bitstream. ECC are one of the

most impactful upgrades to an SDR system as they can improve the reliability and data rate to overall increase reliable data throughput in a system. For example, the SDR on the Voyagers space probe was upgraded in-flight to include concatenated Reed-Solomon error correction which decreased the bit-error rate from 5×10^{-3} to 10^{-6} [13]. Turbo codes are more recently used ECC with better error correction performance but higher computational complexity. This has resulted in continuous research to decrease the implementation complexity to reach the throughputs required by state of the art wireless protocols and fit within small hardware devices.

2.1.1 SDR Platforms and Libraries

SDR Platforms

With the growing capabilities of FPGAs, there are many SDR platforms available ranging from the hobbyist level to commercial use radios. The RTL-SDK dongle is an entry level SDR which uses a re-purposed TV Tuner ASIC to convert radio signals into a digital representation sent via USB to a PC for processing [14]. This device has a relatively limited input frequency range of 22Mhz to 2.2Ghz. Its processing capabilities are limited by the computer attached and its bandwidth. The HackRF One[15] and LimeSDR[16] are more advanced hobbyist grade SDRs which use RF front ends and FPGAs to stream data over USB. These devices have a 1Mhz-6Ghz and 110kHz-3.8Ghz frequency range, respectively. The HackRF One only contains a small CPLD while the LimeSDR contains an FPGA with 40K logic cells allowing for more advanced algorithms to be offloaded to the FPGA, rather than an attached computer.

More advanced SDRs exist through the use of SoCs which contain both ARM CPU cores along with FPGA fabric. The Zedboard SDR II Evaluation Kit includes a Zedboard and an AD9361 Software-Defined Radio Evaluation Kit RF front end [17]. The Zedboard contains a Xilinx Zynq Z7020 SoC with 2 ARM9 cores with NEON

support and 85K logic cells. This SDR is tuned for an input frequency between 2.4Ghz to 2.5Ghz but can be configured from 70Mhz to 6Ghz. A large benefit comes from the use of an SoC with ARM cores integrated. This allows for a standalone SDR system that can implement all components of a communications system. The Raptor SDR builds upon this principle by using the same analog front end with a Xilinx's ZYNQ Ultrascale+ XCZU9EG SoC which has a quad-core ARM-A53 processor, dual-core ARM R5 real time processor, along with 600K logic cells [18]. This is a large upgrade compared to the Zedboard and can allow for more advanced communications systems to be implemented.

Xilinx also has SoCs designed for RF applications in the form of the Zynq Ultrascale+ RFSoCs [19]. These SoC's include ARM CPUs and FPGA fabric the same as the MPSoC, however, the RFSoC's also include RF digital to analog, analog to digital, and configurable forward error correction cores on the same chip. This allows for the potential of a single chip radio solution with a larger degree of reconfigurability. This expands the programmable region within Figure 2.1 to encompass all components except for the RF Chain. There does not seem to be software defined radio development boards available yet, however, Xilinx provides evaluation and characterization kits which could be used to explore their use in SDR contexts. Xilinx also recently released news of the second generation of RFSoCs which can support 5G communications, allowing future SDR platforms to support the newest wireless protocols in use today [20].

SDR Libraries

SDR platforms are only half of the recipe required to build a wireless communications system on an SDR. The other half requires the software to implement the encoding and decoding of data. There are many open source digital signal processing (DSP) and SDR libraries available to aid in this process, though they focus on CPU based

architectures. These libraries are ideal for use with SDR platforms which stream data to a computer for processing. With SoC based platforms these algorithms have the option of being accelerated in the FPGA fabric, however, libraries to accomplish this do not yet exist.

GNU Radio [21] is one of the largest open source projects with a graphical coding interface to build SDR radios. There is support for many signal processing blocks to build an SDR for many communications systems. Most of the library is written in C++ and Python making it ideal for CPU based architecture, however, it makes it difficult to interface with high level synthesis (HLS). The library makes heavy use of dynamic memory and pointers which HLS does not completely support. It also leverages many other software dependencies which would make porting to HLS more difficult.

Liquid-DSP [22] is also a library providing signal processing functions written in C. This library does not have the overhead of GNURadio and is very portable, however, it uses a CMAKE build system which is not compatible with current HLS tools. This library has the potential to be compatible with HLS, however, it would require work to allow for integration with current tools. The other issue with this library is the lack of support for turbo codes. There are turbo code libraries including TurboFEC [23] and AFF3CT[24], however, each library is heavily optimized for SIMD architectures on X86 and NEON for ARM. This would make posting to HLS difficult due to the focus on SIMD. AFF3CT also relied on C++11 which the tools do not support yet. Overall, while there are many open source libraries for HLS, each poses an issue for potential hardware/software integration. An ideal solution is a library optimized for both HLS and CPU implementations to allow for hardware and software co-design on an SoC platform. With an open source library for HLS a new design flow for SDRs and turbo codes could be created to allow for easy implementation and accessibility for more engineers.

2.2 Turbo Codes

Turbo codes are an iterative forward error correction technique with near Shannon's limit performance for use in noisy communication channels [5]. Turbo codes have become one of the defacto ECC used for high throughput and high noise communications including mobile networks and space communications. It is important to utilize an ECC, especially an FEC to correct errors rather than have to retransmit the same data over again until it is received correctly. Retransmissions can be costly, and take more time than introducing some overhead in the transmission in the form of an FEC.

Turbo codes define the error correction scheme and the encoding of data in blocks. On the transmitter side, a turbo encoder is used to generate the encoded bitstream with parity data and the receiver contains a turbo decoder to correct errors and extract the original data. Turbo codes are flexible due to configurable block size, the number of iterations, and the ability to be processed in parallel. This allows them to be used in a wide array of applications such as high throughput LTE mobile networks [6], and high noise deep space communications [5]. The turbo code algorithms have evolved to meet new specifications and performance metrics. For this document, the LTE turbo code implementation will be explored.

Turbo codes were chosen for this research due to their benefits from an HLS accelerated design process. There is a need for turbo codes to be flexible to meet many different standards and platforms. Turbo codes are also computationally intensive and are non-trivial to implement in hardware. There have been many hardware architectures proposed throughout the years to increase throughput without sacrificing the error correcting performance for use in the latest communications systems. Using HLS could decrease the complexity when improving current designs and therefore decrease the time to market which can be crucial for meeting the continually evolving

communications protocols.

2.2.1 Turbo Encoding

Turbo codes encode data at a 1/3 rate, such that the encoded data is 3 times the size of the original input. The encoded data is made up of the original bitstream, a parity stream, and a parity stream of the interleaved bitstream. Interleaving is a process to mix up the bit locations to remove the correlation of position to bit value. The specific operations performed for interleaving are discussed in a section below. The encoding operation is performed with a Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders and an interleaver (Figure 2.2). In this figure, c_k is bitstream to be encoded, and x_k, z_k, and z'_k are the output bitstream, parity, and interleaved parity respectively. All shift registers are zero when the encoding process begins. The bitstream is then processed in the encoder for $0 \leq k \leq K-1$ with the switches in the upper position, where K is the block size of the decoder.

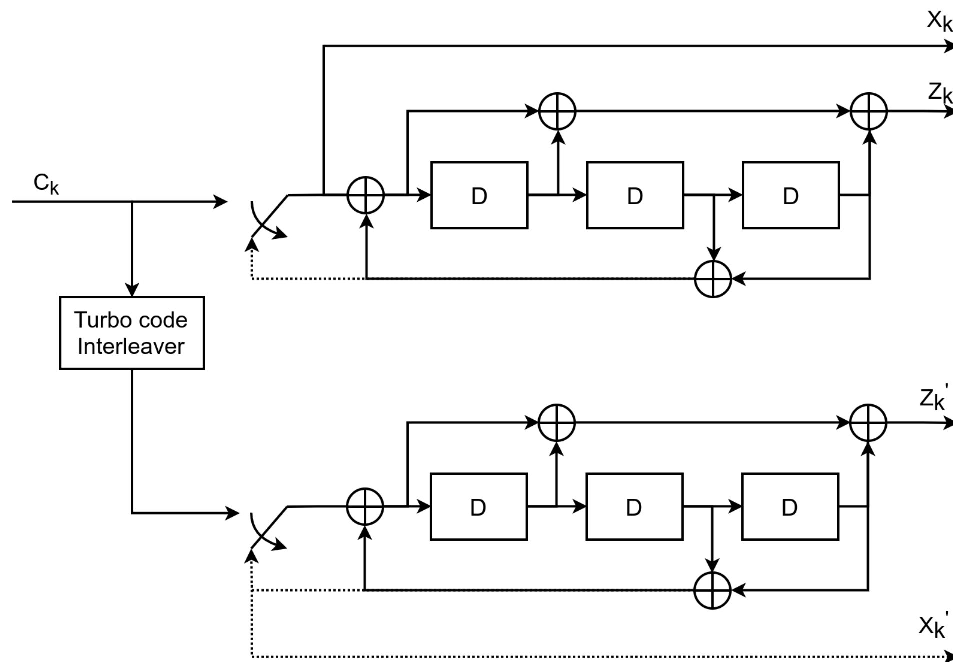


Figure 2.2: Turbo encoder

The LTE specification also terminates the encoder to ensure it ends on a known

state. The first 3 tail bits are used to terminate the first encoder with the upper switch in the lower position while the second constituent encoder is disabled. The last three bits are used to terminate the second encoder with the lower switch in the lower position with the first constituent encoder disabled. The termination bits are then reordered and concatenated with the bitstream and parity streams.

The interleaver component in turbo codes rearranges the order of the bits within a block of data. This occurs to allow for a second parity stream to be generated from a bitstream, but decorrelated from the original data. The design of an interleaver is crucial to the performance of turbo decoders because the algorithm for decoding relied on operating on uncorrelated data. In LTE turbo decoders a Quadratic Permutation Polynomial(QPP) interleaver is used. This design provides good decoding performance with contention free parallel memory accesses to allow for parallel turbo decoder hardware architectures [25]. The basis of the QPP interleaver is the use of a polynomial (2.1) to generate the addresses for interleaving memory.

$$\Pi(i) = (f_1(i + f_2 * i^2) \mod K) \tag{2.1}$$

Where i is the index of the bit to interleave and f_1 and f_2 are the coefficients of the polynomial. The coefficients change depending on the block size of data and all coefficients can be found in Table 5.1.3-3 of [6]

2.2.2 Turbo Decoding

Turbo decoding is based on the maximum a posteriori probability (MAP) algorithm, also known as the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm [26]. For turbo decoding, two MAP decoders, known as soft-input soft-output(SISO) decoders are used (Figure 2.3). This figure shows a turbo decoding architecture for the LTE specification with two SISO decoders along with the interleaver, deinterleaver, and a hard decision maker which converts the soft bits into a binary bitstream. The inputs to

the turbodecoder algorithm are made up of three streams of soft encoded bits, the bitstream itself, along with two parity streams, one which was generated from the interleaved bitstream. In Figure 2.3 they are labeled as $\Lambda^i(X_k), \Lambda^i(Z_k), \Lambda^i(Z'_k)$ for the bitstream, parity stream, and interleaved parity stream respectively.

The inputs to the SISO are a systematic stream along with a parity stream and the output is a log likelihood ratio (LLR) for each bit position. The systematic stream is a form of the original bitstream with additional information. For SISO1 the systematic input is $V_1(X_k)$ which is the original bitstream, $\Lambda^i(X_k)$, added together with the extrinsic value, $W(X_k)$. For SISO2, the systematic input is $V_s(X'_k)$ which is the output of the first SISO, with the extrinsic subtracted from it and then interleaved, $I\{\Lambda_1^o(X_k) - W(X_k)\}$. The parity inputs SISO1 and SISO2 are the respective parity inputs to the turbo decoder, $\Lambda^i(Z_k)$ and $\Lambda^i(Z'_k)$. The extrinsic is the additional information gathered during the turbo decoding algorithm used after each iteration to improve the overall LLR for each bit. This is calculated as follows, $W(X_k) = V_2(X_k) + DI\{\Lambda_2^o(X'_k)\}$, where DI is the deinterleaving process. Finally, the decision block takes the output from the final SISO, deinterleaves it and generates a binary string of '0's and '1's if the values are negative or positive, respectively.

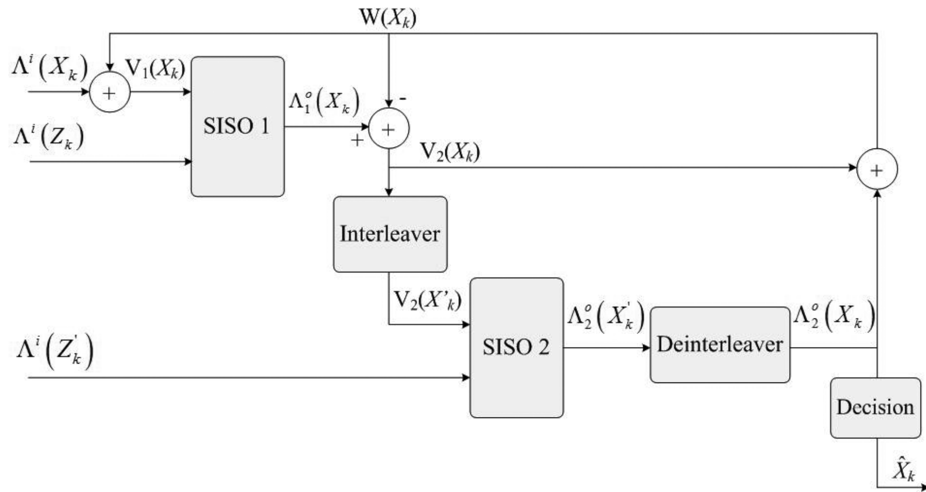


Figure 2.3: LTE turbo decoder [2]

The SISOs implement the MAP algorithm which is based on calculating the proba-

bilities of transitions along a trellis, which is a representation of the 8-state constituent encoders over time. The LTE turbo trellis is shown in Figure 2.4 with an example input and output when encoding. In this example, an input bitstream of 011 was encoded with a parity output of 010. The decoder must work backward with a received bitstream and parity steam which contains noise. In this case, the probabilities of each possible state transition through the trellis are calculated, along with the probability that a state transition occurred moving forward and backward through the trellis, given its neighbors. The maximum probability among all state transitions is used as the basis for the overall probability that the input bit was a ”0” or a ”1”. Finally, the LLR of each bit position is calculated. This alone will correct some errors due to limited paths through the trellis. The process of interleaving increases the number of unique paths through the trellis since both the original bitstream and interleaved bitstream have their own unique path.

The original MAP algorithm within [26] has a large implementation complexity and requires many bits to represent each soft input bit. Because of this, many suboptimal algorithms have been presented to lower the implementation complexity including Logarithmic MAP, Max Log MAP [27], Constant Log Map [28], and Linear Log Map [29]. LTE implements the Max Log Map variant.

MaxLogMap Algorithm

The Max Log MAP algorithm is based on the MAP algorithm in the log domain with a simplified Jacobian logarithm which is known as the max* function (2.2). The max* function can be estimated as the maximum of the two input values, and while this estimation will reduce the performance of the turbo decoder, it also reduces the implementation complexity to allow for lower area and high throughput designs.

$$max^*(x,y) = ln(e^x + e^y) = max(x,y) + ln(1 + e^{-|y-x|}) \approx max(x,y) \tag{2.2}$$

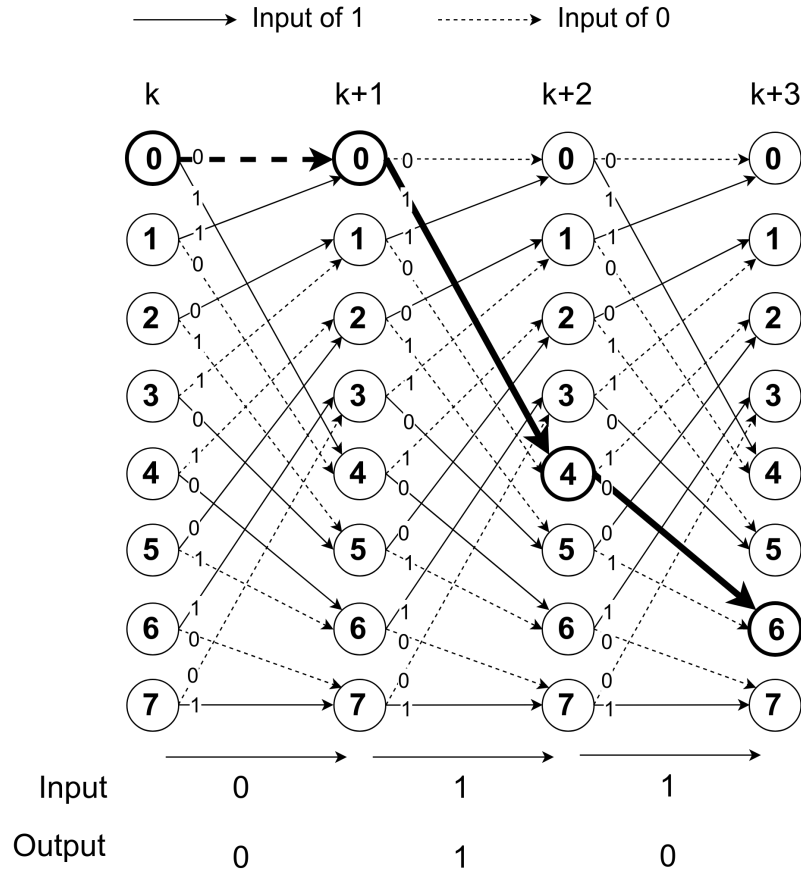


Figure 2.4: LTE turbo code trellis [2]

In the Max Log Map algorithm, a branch metric, along with forward and backward metrics are calculated to estimate the probabilities of transitions along a trellis to estimate the original bitstream. The calculation of the branch metric, γ, is based on the value of the bitstream and/or parity stream depending on the state transition on the trellis. For each state, there are 2 paths depending on if the input is a 0 or 1, thus, for each state there are 2 branch metrics calculated as follows in (2.3).

$$\gamma_{i,j} = V(X_k)X(i,j) + \Lambda^i(Z_k)Z(i,j) \tag{2.3}$$

$X(i,j)$ and $Z(i,j)$ represent the binary values within the bitstream and parity stream where i, j is the transition from state i to state j. In the example of Figure 2.4, the state transition from 4 to 6 is caused by an input of "1". This would result in

$X(0,4) = 1$, since the input was a "1" and $Z(0,4) = 0$ since the parity output is a "0". Since the only outcomes of X and Z are "0" or "1", the equation for gamma can be simplified into four calculations (2.4).

$$\begin{aligned} \gamma_{0,0} &= 0 \\ \gamma_{1,0} &= V(X_k) \\ \gamma_{0,1} &= \Lambda^i(Z_k) \\ \gamma_{1,0} &= V(X_k) + \Lambda^i(Z_k) \end{aligned} \tag{2.4}$$

The backward metric, $\beta_k(S_i)$ is derived from the values of $\gamma_{i,j}$ along with $\beta_{k+1}(S_j)$ for each state transition, $S_i \rightarrow S_j$ from $0 \leq k \leq K$. $\beta_k(S_i)$ is initialized to 0.0 for each state and is calculated as (2.5) for all other value of k, $0 \leq k \leq K-1$. The backward metrics are then normalized to $\beta_k(S_0)$ for each k, $\beta_k(S_i) = \hat{\beta}_k(S_i) - \hat{\beta}_k(S_0)$.

$$\hat{\beta}_k(S_i) = \max\{(\beta_{k+1}(S_{j1}) + \gamma_{ij1}), (\beta_{k+1}(S_{j2}) + \gamma_{ij2})\} \tag{2.5}$$

Similarly, the forward metric, $\alpha_k(S_j)$ is derived from $\gamma_{i,j}$ along with the prior forward metric, $\alpha_{k-1}(S_i)$ for each state transition $S_i \rightarrow S_j$ from $0 \leq k \leq K$. $\alpha_0(S_i)$ for all states is initialized to 0.0 and for $1 \leq k \leq K$, $\alpha_k(S_j)$ is calculated as in (2.6). The forward metrics are then normalized to $\alpha_k(S_0)$ for each k, $\alpha_k(S_i) = \hat{\alpha}_k(S_i) - \hat{\alpha}_k(S_0)$.

$$\hat{\alpha}_k(S_i) = \max\{(\alpha_{k-1}(S_{j1}) + \gamma_{ij1}), (\alpha_{k-1}(S_{j2}) + \gamma_{ij2})\} \tag{2.6}$$

Once the branch, forward, and backward metrics are calculated, the final step in the SISO decoders is to calculate the probability for each state transition (2.7).

$$Z_k(S_i \rightarrow S_j) = \alpha_{k-1}(S_i) + \gamma_{ij} + \beta_k(S_j) \tag{2.7}$$

Then the probabilities are summed for the transitions representing an input bit of a 0 and 1, forming an overall probability for each bit value. These values are then subtracted to form an LLR for whether the bit is likely a 0 or 1 (2.8). If the

probability of the '0' transition is higher than the '1' transition, then the overall LLR will be negative, which represents a 0 when converted to binary, and vice versa for a positive LLR.

$$\Lambda^o(X_k) = \max_{(S_i \to S_j):X_i=1} \{Z_k(S_i \to S_j)\} - \max_{(S_i \to S_j):X_i=0} \{Z_k(S_i \to S_j)\} \tag{2.8}$$

In summary, turbo codes are a common ECC that use a computationally intensive algorithm that is commonly implemented in reconfigurable hardware to meet throughput requirements. While suboptimal algorithms are utilized for simplified hardware implementation, the design process for FPGAs is still very time consuming and required highly specialized engineers. HLS provides an alternative design approach that can allow for a more efficient and more accessible implementation of hardware architectures, including turbo codes.

2.3 High Level Synthesis

HLS is the process of converting a behavior model of an architecture developed with an high level language (HLL) into a hardware RTL model. Current tools can convert C/C++ or OpenCL into synthesizable VHDL or Verilog models. The challenge with this conversion is that HLLs are sequential in nature, whereas developing with an HDL, hardware can be implemented in parallel. Tools must extract the control flow and data flow from software and convert them into parallel hardware with a control unit. The quality of this process depends heavily on the code structure as many software optimizations and constructs do not translate well into hardware. Because of this, care must be taken when developing code for HLS in order for quality hardware models to be generated.

Current use cases for HLS mainly exist for software oriented engineers to quickly accelerate software designs. This is especially common in cloud computing environments where performance is crucial for new designs. Graphics Processing Units

(GPUs) and Application Specific Integrated Circuits (ACIS) [30] are also used in the cloud but GPUs are not suited for all workloads and ASIC's can be cost prohibitive, especially for evolving algorithms. FPGAs are becoming more present in the cloud [7] and there are tools such as SDAccel from Xilinx [31] which can accelerate software designs with little to no hardware development knowledge. For local platforms on an SoC, there is also SDSoC from Xilinx [32] which accelerates portions of a software design within FPGA fabric. The performance may not be as high with HLS based tools versus a handcrafted implementation, but the design can be completed must faster and cheaper, while still providing a speedup over software.

2.3.1 HLS Development vs Software Development

While HLS may use an HLL to facilitate describing an architecture, only a subset of the language may be supported. Common coding constructs used in software for a CPU architecture may require HLS specific formatting to allow the tools to produce quality hardware. The main areas which must be modeled differently are memory accesses and overall program flow. The design process parallels designing a hardware architecture, as thought must be given to how an algorithm is implemented in hardware, especially when operations can occur in parallel.

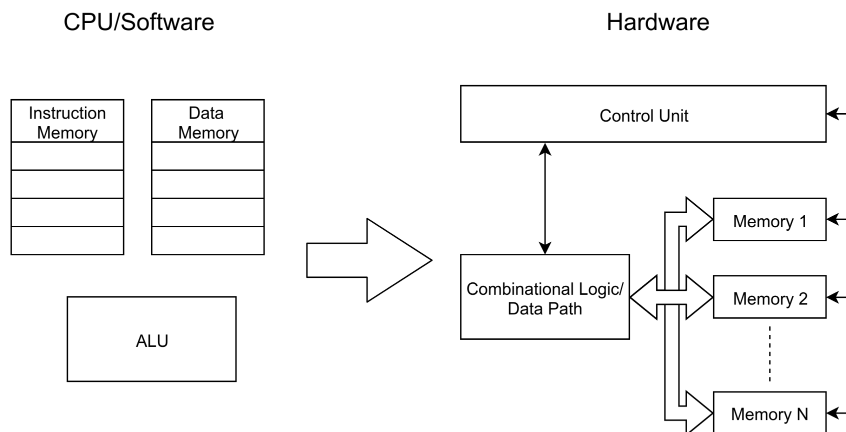


Figure 2.5: High level mapping of CPU architectures and code execution to a hardware architecture

This can be demonstrated by comparing how an algorithm is executed on a CPU versus implemented in hardware. Figure 2.5 demonstrates how, in a simplified model of a CPU, there is an instruction memory storing the operations to perform, the data memory holding all values to operate on, and an ALU to do the operations. In hardware, this translates into a control unit, which defines ”operations” similar to instructions, memories to store data, and combinational logic to perform ALU operations suited to a specific application. In hardware, there are many memories utilized, rather than a single data memory. This is the reason why pointers are not supported in HLS as they are in software since they don't actually point to a place in a single memory space, but define a new memory interface. The limitations when using pointers will also be discussed in depth later in this document.

General Data Flow

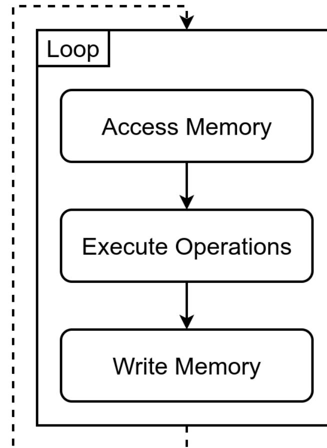


Figure 2.6: General HLS algorithm data flow to produce quality results

The general data flow to produce quality hardware follows a standard pipeline flow, where data is accessed, operated on, and then stored (Figure 2.6). This order of operations can be applied to a small portion of a program within a loop as shown in Figure 2.6, or to an overall program. This helps guide the tools when extracting memory allocations and dependencies, by making them more explicit similarly

to HDL development. If left implicit and there are other memory dependencies in an algorithm, the tools may delay the scheduling of operations as it processes code sequentially. If there is a memory dependency then all operations below are scheduled after the dependent operation. This is done to ensure that the final hardware implemented the same algorithm, though opens an area for optimization.

The approach described above may seem wasteful to a software programmer as it requires many extra variables and registers to be stored, however, in hardware this will allow for reading data in parallel and at the cost of only FFs for temporary storage. This general approach also allows for hardware pipelining, and additional pipeline parallelism since there must not be inter or intra memory dependencies for successful pipelining of a design.

Pointers and Multiplexing Data

A common construct in C programming is the usage of pointers to pass data by reference. While this is very useful in software, care must be taken when programming for HLS since the tools are limited in pointer support. In HLS pointers are used for all arrays and can be used for variables as well, however, all pointers must be static and are therefore immutable and serve only to represent an interface to data. This limitation exists as the HLS tools must allocate and route to all memory used in a design.

Multiplexors (MUXs) are heavily used in hardware development to address between different logic elements. In software this translates to an *if-else* statement or function call with different parameters. Figure 2.7 shows an example using multiplexors to share an adder resource. Listing 1 shows an example of using multiple function calls within an *if-else* to model the MUXs. Another approach is to move the *if-else* within the function call and pass all possible values in (Listing 2). While this approach increases the complexity of modeling a MUX, it allows for more oper-

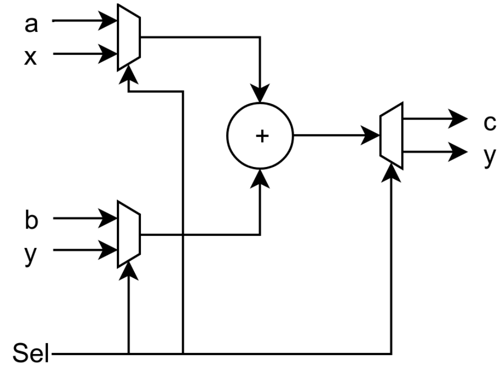


Figure 2.7: Simple multiplexer example of sharing an adder hardware resource

ations to occur depending on the select input. This method could also lead to more optimizations since all operations are within the scope of one function, and the HLS tools optimize each function on its own [3].

Listing 1 Multiplexing data with a function call

```
function ADD(int in1, int in2, int out)
    out = in1 + in2

if sel == 0 then
    ADD(a,b,c)
else
    ADD(x,y,z)
```

Listing 2 Multiplexing data with if else statement within a function

```
function ADD(in1_a, in1_b, in2_a, in2_b, out_a, out_b, sel)
    if sel == 0 then
        out_a = in1_a + in2_a
    else
        out_b = in1_b + in2_b

ADD(a,x,b,y,c,z,sel)
```

Variable Bounded Loops

A common construct in software and hardware alike is to loop with some conditional bound. In software, this could be accomplished via a *for-loop* as in Listing 3. For

HLS to generate quality hardware, static bounds must be defined for a loop. This is required to generate the smallest counters and adders required for a loop to be implemented in hardware. To ensure that this can still occur and have a conditional bound, a loop with a max iteration bound and an exit condition within will produce the best hardware (Listing 4). This works by allowing the HLS tools to generate minimized hardware to work with a bound up to some max while still allowing for the loop to exit. It is critical for the exit condition to be the last statement within the loop for the hardware to not waste any cycles. If the exit condition was first, the check would delay the contents of a function for a full clock cycle, whereas if it is at the end, the check can occur while the body is being evaluated [33].

Listing 3 Standard software for loop with variable end condition

```
for x ← 0 to end_condition do
    Operations
```

Listing 4 HLS optimized for loop with variable end condition

```
for x ← 0 to MAX_LOOP_ITERS do
    Operations
    if x > end_condition then
        break;
```

2.3.2 Limitations of HLS Tools

While HLS tools have been developed to support a large portion of language constructs and hardware elements, the current tools still contain limitations which must be mitigated. One of them, already mentioned, is that all pointers must be static and many of the benefits of modularity around pointers are lost. Similarly, all memory must be static as there is no such thing as a heap or stack in hardware like on CPU based systems. While there are not comparable use cases for dynamic memory in hardware, support for it may allow current software libraries to be ported to HLS.

Data dependency is also an area with limitations in relation to the scope in which the tools look for them. Localized dependencies within a function or loop are normally captured correctly, however, for complex dependencies the tools tend to be over cautious and may flash false dependencies. There are directives to allow the programmer to specify the data dependency for a variable, but this only works within a small scope. Dependencies between functions using the same memory are limited in functionality. The tools will optimize and schedule functions if all memory is read and written in order and only once. Otherwise, the tools default to assuming a dependence and schedule functions sequentially. This poses a large issue for parallel out-of-order memory access which is contention free. As a result, it becomes challenging to model a parallel architecture with out-of-order memory access, as simply duplicating function calls will not work. Breaking up loops with inner loops to add parallelism allows this to be bypassed, but increases the complexity of the code.

When describing an architecture with an HDL, it is common to make the design hierarchical. Sections of an algorithm can be implemented as separate entities with input, output, and control signals. Then a top level design would piece components together and contain a state machine that controls the design. There may be multiple levels within a design depending on its complexity. When designing with HLS, the ability to design hardware with this hierarchy is reduced. Functions in C are clocked and each one has its own state machine. This can allow subsections of an algorithm to be implemented, however, no purely combinational functions can be modeled with hardware reuse. This is possible through the use of the inline directive which allows the HLS scheduler to duplicate hardware and optimize as if it were inline to the overall algorithm. This may cause some hardware to be unnecessarily duplicated, however, this is a trade-off with the use of HLS.

2.3.3 HDL Design Flow vs HLS Design Flow

Designing a hardware architecture on an FPGA in an HDL versus HLS requires a different design flow to achieve quality results. A generic flow to produce handcrafted hardware is shown in Figure 2.8. The first step is to design the architecture which implements the desired functionality. A hierarchical approach is commonly taken, building up an architecture from its basic functional units. Once a architecture is planned out it can then be described in an HDL. The next step is to verify the HDLs functionality and ensure it implements the architecture as designed. Synthesis, place, and route then take place which implements the architecture via FPGA primitives. Finally the design can be evaluated on the FPGA for its area usage, cost and performance.

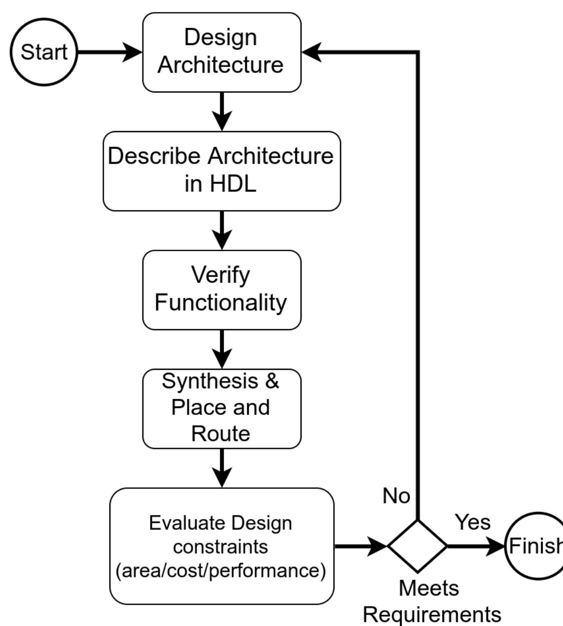


Figure 2.8: General handcrafted RTL hardware design flow for an FPGA

The design flow to produce hardware with HLS changes considerably due to the tools automating the scheduling and generation of the hardware. The general process flow to design hardware with HLS is shown in Figure 2.9. The *HLS Development* section in the diagram allows for rapid iterations of designs since the tools provide

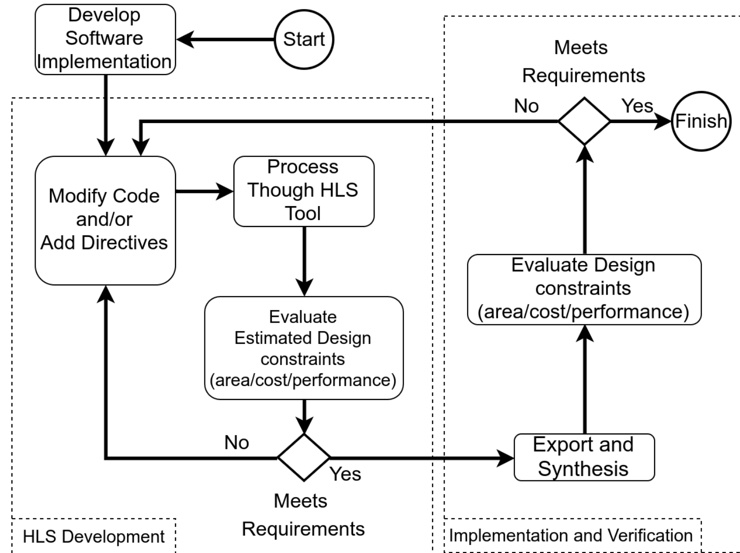


Figure 2.9: General HLS design flow from hardware oriented software to a high performance hardware design

instant feedback. This is due to the tools ability to process code and make large changes to the hardware and report on them must faster than a design could with an HDL. The tools also provide an estimation for evaluating design constraints. A clock constraint can be set, and an estimated clock is given after C synthesis. Similar to HDL development, once the HLS tools process the code and produce a hardware model, the design can be synthesized. After synthesis, the design can then be evaluated again. Verification is handled via a C simulation to verify software, and a co-simulation to verify the generated hardware replicates the functionality.

HLS has the potential to become a high performance tool for hardware design while also decreasing the learning curve by modeling architectures in an HLL. There are implicit challenges with this process since software and hardware development are fundamentally different, however, with a design flow geared towards developing code which describes hardware, the HLS tools can produce quality hardware. By modeling turbo decoder architectures in hardware, a complex and continually evolving algorithm can test this design process and analyze the resulting hardware design.

2.4 Related Work

The state of HLS is continually improving as tools evolve in processing HLL and applying hardware optimizations. HLS has demonstrated performance speedups when compared to software and is in use in cloud environments, however, research continues into the effectiveness of HLS as an alternative to handcrafting hardware designs. HLS has the potential to become a faster and easier development method for hardware design than writing RTL HDL [8, 9, 34, 35].

In 2011, [34] carried out a case study on the AutoPilot HLS tools and examined its effectiveness of speeding up software. This paper demonstrated that HLS was able to speedup stereo matching software codes along with common cryptographic algorithms by up to 126X versus a software implementation. They also concluded that the design effort to produce hardware designs was less than handcrafting RTL. Research into HLS with regards to accelerating software has led to uses within cloud computing environments and is a current use-case for HLS tools. Since the tools are continually improving and could expand into the hardware development domain, this thesis looks at using HLS from a hardware engineer's perspective with the goal of meeting handcrafted performance.

In 2014, [35] performed a case study on the effectiveness of HLS with regards to software speedup and the usage of SystemC and SystemVerilog in an HLS accelerated work flow. This paper concludes that the quality of the results is dependent on both software and hardware knowledge and a predefined architecture should be in mind to produce good source code for HLS. This paper explains that no standardized HLL is defined for HLS and explains that SystemC was the best choice at the time of evaluation. With the current HLS scene favoring C/C++ or OpenCL for languages, further research is required to determine the best methods for modeling hardware in C, rather than the more hardware oriented SystemC. Our work uses C/C++ to

model turbo decoder architectures, and in the process examines how hardware must be modeled to produce quality designs.

In 2018 a study of 46 research papers was conducted to determine the quality of HLS designs compared with HDL design flows, along with the required design effort [8]. Rather than a self coded case study, this paper looked at other authors and their efforts to produce quality hardware with HLS. While they concluded that handcrafted RTL design still outperforms HLS based designs, HLS required less design effort for the same architecture. One parameter not explored was how each design was modeled in an HLL to produce quality hardware. This work is an extension of the ongoing research to produce hardware which can reach handcrafted levels of performance, and the required design process to facilitate this.

The use of HLS in designing a complete SDR has been explored [36]. In this work, a full Zigbee radio was implemented with HLS on a Virtex-6 Perseus 6010 platform. The paper concluded that it is possible to implement an SDR in a C language and that HLS provides a noticeable design potential for flexible wireless platforms. The work did not provide performance metrics and focused on the feasibility of implementation. This thesis focuses on one block within an SDR, but with the goal of matching handcrafted levels of performance.

The feasibility of developing ECCs within HLS has been explored in various works such as [9, 10, 37]. In [10] the implementation of low-density parity-check (LDPC) decoders with HLS was explored. In this paper, HLS architectures were proposed and successfully implemented with RTL levels of performance, however with higher logic utilization. This work implements a turbo decoder based on existing hardware architectures to approach HLS from another perspective. Turbo codes are also more computationally intensive than LDPC decoders [38] and together with exploring the design process, this work will help validate and explore the performance of HLS in more domains.

In [37] the feasibility of implementing turbo codes with HLS was explored. In this paper, the goal was to implement a turbo decoder in a rapid manner without concern for performance. In [9] many revisions of a turbo decoder were implemented to produce a high performance design. Performance metrics were only compared to a software design and the design process revolved around improving a software implementation. This work focuses on hardware architectures and how HLS can model them and produce a performance that equals that of handcrafted designs.

Chapter 3

Methodology

In order to develop C code which models a hardware architecture, it is crucial to understand the specific coding style for HLS tools. While it is possible to take a software implementation of an algorithm and synthesize it in HLS, it will generate a functional, but suboptimal hardware model. This is due to standard software constructs not taking full advantage of the hardware. By understanding some HLS specific guidelines to follow, a software implementation can be improved via code style changes or through the use of directives to help guide specific hardware to be generated.

3.1 HLS Design Considerations

3.1.1 HLS Design Flow

Designing a system for HLS tools requires a new design flow compared to standard software or hardware development. With almost instant feedback from the tools, the development follows an iterative process with a rapid development cycle compared to standard handcrafted hardware designs [8]. Without a standard for describing hardware architectures in HLL, each HLS tool provides unique guidelines and optimization techniques to produce quality results. For this work, Vivado HLS 2018.2 is utilized due to its leading HLS performance for C languages and integration with

Xilinx's standard HDL development tools.

The basic approach to designing hardware with HLS begins with a software implementation of an algorithm and continually modifying the design after processing it through HLS tools until the desired hardware is reached (Figure 3.1). Each design begins with a C description of the algorithm to be implemented in hardware. Care must be taken when developing the software implementation to ensure compatibility with HLS tools especially when using pointers as discussed in Chapter 2. The code is then modified (1) to ensure that software constructs, which translate well into hardware, were used along with the addition of directives. Directives are additional information passed to the compiler which can inform the tools of what hardware to generate.

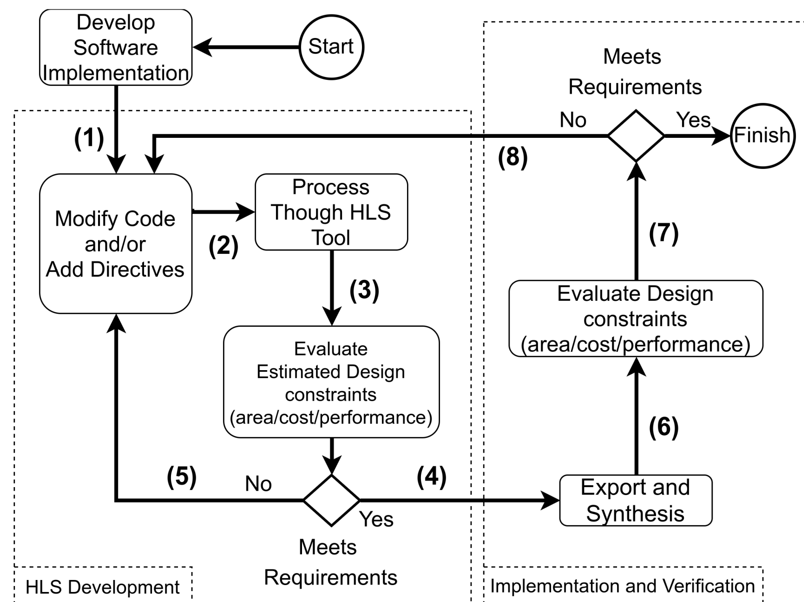


Figure 3.1: HLS design flow

After modulations are made to the software implementation, the code is processed through the HLS tools (2) and the resulting architecture is evaluated (3). Vivado HLS generates a synthesis report which gives estimations for the clock period, resource usage, and design latency. A schedule view of the hardware is also given which can be useful to find design bottlenecks and analyzing the parallelism of the generated hard-

ware architecture. Once a design is evaluated, if it does not meet design requirements or bottlenecks are found, the code can be modified or directives are added (5) and the design cycle continues until the requirements are met.

The HDL model of the architecture can then be exported for use in Vivado (4). Part of this process also allows for the design to be synthesized and implemented within an FPGA to determine the minimum clock period, resource usage, and latency when implemented. Then a final evaluation is performed on the implemented hardware to determine if all design requirements are met (7). If the final design does not meet the requirements, then the HLS design process iterates again (8).

3.1.2 Vivado HLS Steps

In Figure 3.1 when processing the code through Vivado HLS many steps can be taken for design and verification. Both the design and testbench are coded in C/C++, and an initial round of testing is completed via C simulation. This uses GCC and compiles it with all of the HLS libraries as software, and can be executed on a CPU for testing. This is a fast method for verification, though only verifies software functionality and does not guarantee the hardware generated to be correct.

The next step is C Synthesis. This is where the tools analyze and generate a hardware model, which implements the C designs functionality. For synthesized designs, Vivado HLS offers tools for analyzing the resulting hardware design. There is a report generated after synthesis, which gives estimates for minimum clock period and resource usage. The minimum and maximum latencies are also reported if all loops have fixed bounds. There are also tools to view the schedule of operations and resource usage and can be used to find areas for improvement.

After synthesis, a co-simulation can be completed to verify the HDL functionality using the C testbench. This ensures that the generated HLS model matches the functionality of the software. The co-simulation will also report the latency of the

design with provided parameters.

Finally, the design can be exported for implementation. As part of the exporting process, the design can also be implemented in Vivado to extract the resource usage and minimum clock period.

3.1.3 Directives

Directives, otherwise known as pragmas, are language constructs that inform the compiler on how to processes some of its inputs. In the use case of HLS, these directives are used to inform the HLS tools with more information on how to generate hardware for specific sections of the design. The need for directives in HLS arises from the lack of ways to model hardware in software. There are many directives which Vivado HLS supports, varying from memory organization to pipelining of designs. Below are the directives utilized in the design process for this thesis.

Array Partitioning

Memory organization is an important design consideration for almost any hardware design. By default in the C/C++ languages, only the size and shape of memory can be defined without consideration for how it is accessed in hardware. Vivado HLS provides a directive for array partitioning to describe how an array is broken up into memory units depending on how many memory ports are required. By default in HLS, every array is placed into a form of BRAM with one or two access ports. The partition directive allows for an array of any dimension to be broken up into multiple sub-arrays at the hardware level. Figure 3.2 demonstrates the impact of the 3 types of partitioning: block, cyclic, and complete.

The block type, splits an array up into a specified number of blocks of contiguous memory, each consuming separate hardware memory. Cyclic is similar, however, the memory is not-contiguous as it is cyclic based on a specific factor. Finally, the

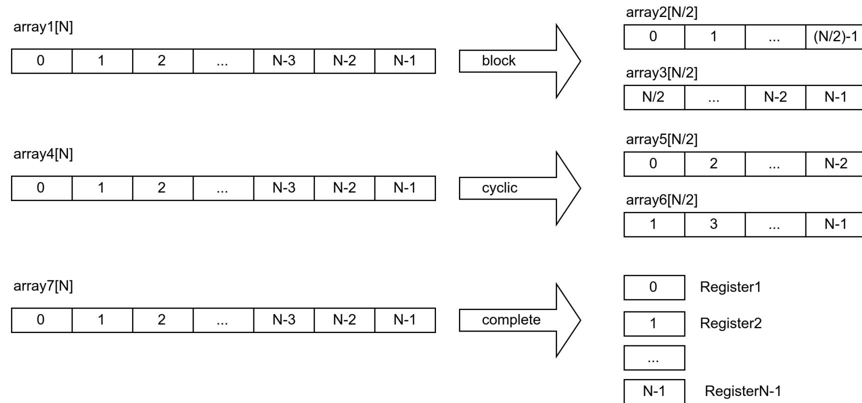


Figure 3.2: Array partitioning effect

complete type splits up an array into separate elements, for a 1D array this generates registers, and for any higher dimensional arrays, it will split them into smaller arrays of 1 less dimension. In this work, the most common use case for this directive was to split a small array into registers due to the need for parallel access to all elements. Another use case of the array partition directive was to split a large array into smaller arrays to allow for independent memory access, though care must be taken since the address calculation for the smaller blocks may require additional hardware if not split on power of 2 boundaries.

Array Reshaping

Depending on the algorithm being implemented, arrays described in C do not always translate well into a memory layout and interface which is suited for a hardware architecture. The *array reshape* directive modifies how arrays are stored. The basic function of the reshape directive is to change what information is contained within a word of an array. Figure 3.3 shows the three possible effects of the directive.

Similarly to the array partitioning, there are block, cyclic and complete methods of reshaping. Reshaping in blocks is a similar operation to partitioning, except the arrays are combined such that the word length increases, while the overall array length is split by a configurable factor. This is shown in Figure 3.3 where an array is

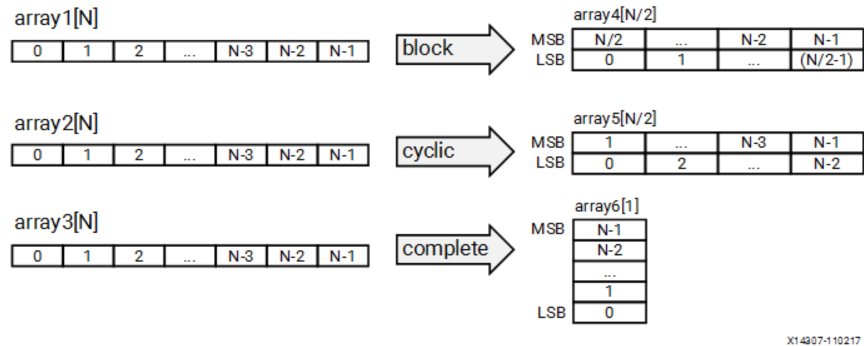


Figure 3.3: Array reshaping effect [3]

reshaped by a factor of two, with each word of the array containing an element from the first half and one element from the second half of the array. The cyclic reshaping is similar, but each word of memory contains two contiguous elements of the array. Finally, complete reshaping converts an array into one large word, which is used for multidimensional arrays to access an entire row or column in one memory access.

Pipelining

Pipelining is a very hardware specific operation, and there is no software construct to describe it explicitly. Through the usage of the pipeline directive, and following a sequential dataflow without interdependencies or loop dependencies, a design can be pipelined. In a pipeline, the initiation interval (II) is the number of stages before a new operation can be issued to avoid a data dependency. A target II can be specified and the tools will reach the closest II to its target based on data dependencies.

Dependence

The dependence directive provides information on loop dependencies such that designs can be pipelined with the lowest possible interval. This directive is used to flag a false interdependency which may allow for successful pipelining at a target II. The HLS tools detect loop carry and loop independent dependencies, however, it can be too conservative for complex indexing of memory.

Loop Unrolling

Loop unrolling is similar to software loop unrolling, however, the result is that hardware is replicated. This directive was commonly used to unroll a loop completely to allow for parallelizing independent operations. This allowed all operations in the loop to complete in the same number of clock cycles as one iteration, but at the cost of additional resources.

Inlining

The inline directive removes the hierarchy of function calls as it replaces each function call with the body of the function. This removes the overhead of calling a function and gives more freedom to the scheduler but makes analyzing the output of HLS more difficult to understand. This is due to Vivado HLS preserving the hierarchy in the analysis tools that allows for each function to be optimized. When inlining, the hierarchy is lost and the larger algorithm must be examined.

3.1.4 Data Types

When developing code for use with Vivado HLS, there are additional data types added to extend the C/C++ language for hardware development. For software development, primitive types are provided with a fixed width. This makes sense for software which is executed on a fixed width CPU, however, for hardware development, it is advantageous to limit all types to their minimum bit width to save resources. Vivado HLS provides types with a configurable bit width. For designs using floating point numbers, fixed point arithmetic can be a saving in computation complexity and resource usage. Vivado HLS also provides fixed point types with configurable widths, quantization modes, and overflow modes.

3.1.5 Memory Design

Out of Order Memory Access and Dependence

When designing memory architectures with indexes that are not directly based on a looping variable, it is important to carefully order memory accesses as to not cause a false dependency, potentially delaying other memory accesses. This can be caused when an out of order memory accesses uses an index that is stored in another area of memory. This would form a dependence and care must be taken to ensure no other memories are flagged as a false dependence.

When the HLS tools analyze C code, it must extract the data dependencies and use them to schedule the order in which operations occur including which operation can occur in parallel. When multiple memories are read in a single loop where one memory access is dependent on another, their order determines the scheduling, even if memory access after the dependent access, contains no dependencies. This scenario is demonstrated in Listing 5 and Listing 6.

Listing 5 Memory order with false dependence

```
1: for k ← 0 to BLOCK_SIZE do
2:     index = interleaver[k];
3:     value2 = array1[index];
4:     value3 = array2[k];
5:     value4 = array3[k];
```

Listing 6 Memory order with no false dependencies

```
1: for k ← 0 to BLOCK_SIZE do
2:     index = interleaver[k];
3:     value3 = array2[k];
4:     value4 = array3[k];
5:     value2 = array1[index];
```

In Listing 5, the index for array1 is captured first, and following it array1[] is read. Array2[] and array3[] are read next, and from a programmers view, the index, value2, and value3 could be read in the first cycle of this program due to them being

independent. However, in this case, they would be delayed due to HLS focusing on dependencies sequentially. Processing down the loop, there is a dependency on accessing array1 which causes a delay for future operations. Listing 6 solves this issue by specifically placing all memory accesses which are independent of another memory access before any dependencies such that they can be scheduled in parallel. The results of the algorithms are the same, but they will have differing schedules. This modification was used when memory accesses were unexpectedly delayed and were commonly traced back to an unintentional memory dependence issue in the code.

Parallel Memory Access

Attention is required when parallelizing algorithms, especially with memory accesses. Multiple approaches can be taken depending on the exact requirement of a design. The approach taken for memory organization, in this case, was to reshape a two dimensional array which required parallel memory access of all elements in a row. This array was reshaped such that each row was one read of a large word of data. This allows for the C code to access the memory as an array of two dimensions, but in hardware, each row is read in one memory access.

3.1.6 Coding Structures

Order of Operations

Due to HLS extracting the parallelism out of a sequential code structure, it is crucial to ensure the order of operations to allow for the best results. The general design flow for a system that reads memory, operates on it, and stores memory is shown in Figure 3.4.

This was especially important for the turbo decoder designs. When memory was read it was commonly done in blocks, especially in the parallel designs. In order to ensure that memory was read in large words after reshaping arrays, the memory was

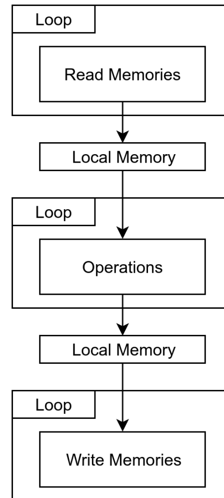


Figure 3.4: General design flow with multiple memory accesses

read as one large block and stored in local registers to be used in parallel for the algorithm. This allowed for the scheduling of the algorithmic part of code with no memory dependencies or read port issues. The algorithm would then store its results in local memory and a loop would be used on the output to write in blocks if required.

Looping

When looping, there are strict requirements for bounds when using HLS. For the scheduler to optimize a design, the bounds must be static, though this presents an issue when a design requires a loop with a variable exit condition. This is achievable through breaking from the loop after an exit condition, while still preserving static bounds for the overall loop, as shown in Listing 7.

Listing 7 Loop format with variable exit condition

1: **for** k $\leftarrow$ 0 **to** 100 **do**
2: $\{$*operations*$\}$
3: **if** $k >$ some_variable **then**
4: break;

In conclusion, the HLS design process possesses some challenges in modeling hardware architectures in a high level language such as C, but following some HLS specific

constructs can allow for quality hardware to be generated. HLS offers a unique design process with rapid iterations between modifying code, adding directives and processing though HLS tools to receive feedback. By modeling turbo decoders with the process outlined in this section, it will be determined if HLS can produce quality hardware which also reaches handcrafted levels of performance.

3.2 Clock Constraint Exploration

With any hardware design, the clock speed is a crucial design choice to meet performance and area requirements. In handcrafted architectures, it is up to the designer to craft an architecture such that the longest path delay is shorter than the desired clock period. In the case of HLS, the tools control the scheduling of logic blocks, removing some control from the designer. To allow the designer to influence the tools, a clock constraint can be altered to change how the tools schedule a design.

To determine the impact of this clock constraint and the ability of the tools to meet clock requirements, each design will be subjected to varying clock constraints. For each design, the clock constraint is varied from 1ns to 10ns in 1ns increments. The resulting estimates from the HLS tools for area and performance will be examined as shown in Figure 3.1. These designs will then be synthesized and implemented to determine the final area and performance. The impact of the clock constraint on the estimated and actual performance and area will then be compared to investigate the impact of the clock constraint on the tools. This is useful for designers so they have an understanding of the impacts of design choices specific to an HLS workflow.

3.3 Methodology

In order to explore the HLS design process when targeting a turbo decoder hardware architecture, three handcrafted architectures were modeled in C and implemented in

hardware through HLS. The overall process which facilitates this can be found in Figure 3.5.

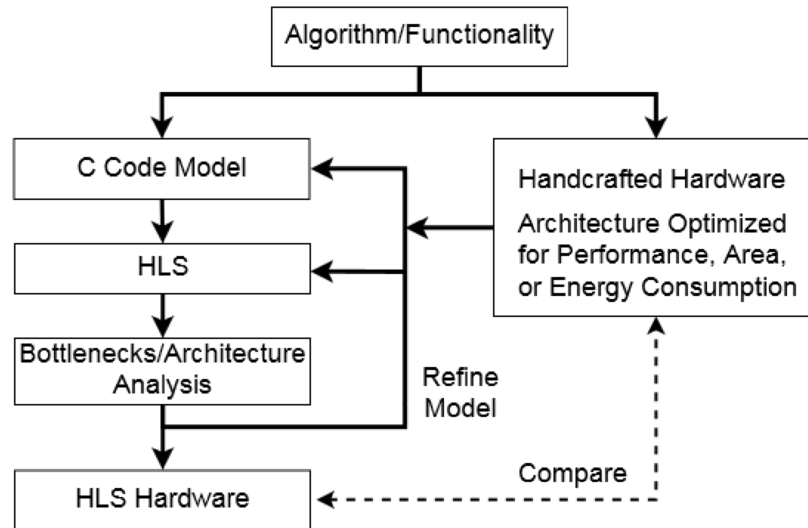


Figure 3.5: Methodology used to replicate handcrafted turbo decoder designs with HLS

The methodology followed in this work is shown in Figure 3.5 and each step is discussed below:

- Algorithm/Functionality - The algorithm and functionality of the target hardware architecture. In this work it was determined via data flow, scheduling diagrams, and equations for the algorithm presented in the papers.
- C Code Model - C code is then developed to replicate the algorithm and functionality from the hardware architecture.
- HLS - The C code is then processed through the HLS tools to generate hardware which implements the functionality of the C code.
- Bottlenecks/Architecture Analysis - The HLS design is then evaluated and if any bottlenecks are discovered, then the code must be refined and the process repeats until the hardware models the target architecture. The design is refined by modifying the C model or adding additional HLS directives.

- HLS Hardware - The final hardware generated with HLS after all design bottlenecks are mitigated. The final hardware is then compared with the handcrafted hardware description based on design performance. In this work, the throughput of the turbo decoder architectures is used for comparison.

Chapter 4

Architectures

4.1 Design 1: Sequential Decoder

4.1.1 Handcrafted Serial Architecture

The first design implemented with HLS was an LTE serial turbo decoder hardware architecture proposed in [2]. This architecture will be referred to as the handcrafted serial architecture (HSA) to differentiate it from the HLS implementation. In this architecture, the focus is on reduced hardware usage with only one SISO decoder, an interleaver/deinterleaver memory unit, and a hard decision maker. The block diagram provided of the HSA in [2] can be found in Figure 4.1. This diagram implements the LTE decoder in Figure 2.3 and describes how each block in the overall system is connected so that only one SISO unit is required.

Figure 4.1: Serial turbo decoder block diagram [2]

The high level block diagram of the architecture only provides some of the required information to model the hardware architecture. The work in [2] also provided a timing diagram of the memory operations and calculation units within the system (Figure 4.2). This diagram gives information on the order of operations and how many clock cycles each takes. The last piece of information not provided in the diagram is the computations required for each COMPUTE block, but they can be derived from the Equations (2.6-2.8).

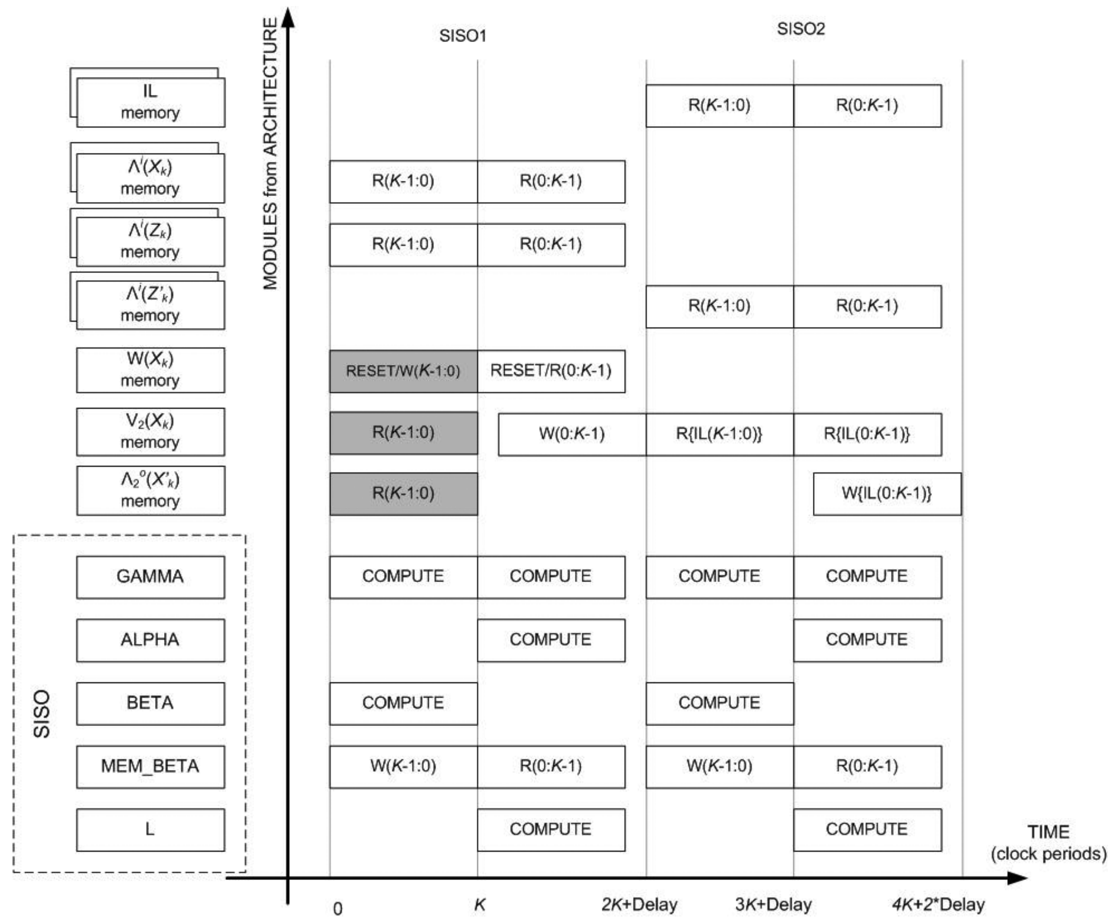


Figure 4.2: Serial turbo decoder timing diagram [2]

Figure 4.2 is presented with the memory reads and writes in the top section, denoted as R or W respectively, and the computations in the bottom half. Of note, the computations of γ and β from 0 to K-1 produce K computations. This means that a new value of γ and β are calculated every clock cycle. For the second half of SISO1,

from K to 2K+*Delay*, γ, α, and L are calculated. Since $V_2(X_k)$ is delayed in writing by *Delay* clock cycles, and produces an output every clock cycles after *Delay*, it can be inferred that a pipeline with initiation interval (II) of 1 was used for computing γ, α, and L. From the derived architecture, a C model was developed.

4.1.2 Serial Decoder HLS Implementation

To model the handcrafted turbo decoder architecture in C, the general approach of reading memories, operating on them, and then writing the results to memory was used. This was carried out for each subsection of the turbo decoding algorithm, especially for the calculations of the α, β and γ terms. The turbo decoding algorithm equations were first developed, a combination of the mathematical algorithm and the timing diagrams provided were used to develop C code.

Algorithm 1 Unrolled calculation of γ (2.3)

```
gamma0[0] = 0.0;
gamma0[1] = 0.0;
gamma0[2] = parity;
gamma0[3] = parity;
gamma0[4] = parity;
gamma0[5] = parity;
gamma0[6] = 0.0;
gamma0[7] = 0.0;

gamma1[0] = systematic + parity;
gamma1[1] = systematic + parity;
gamma1[2] = systematic;
gamma1[3] = systematic;
gamma1[4] = systematic;
gamma1[5] = systematic;
gamma1[6] = systematic + parity;
gamma1[7] = systematic + parity;
```

The design of the serial turbo decoder in HLS began with the modeling of the equations which make up the SISO algorithm in C. The C++ compiler was used, however, since Vivado HLS only provides arbitrary fixed point data types for C++.

These data types are optimized for hardware and the HSA uses fixed point operations for the decoding algorithm. The γ, β and α/LLR terms were each coded in C. The γ term is simplified for the LTE decoder due to a fixed number of states used in the encoding process. This leads to (2.4) where only 4 values are calculated. Even though there are only 4 distinct values, there are 16 γ values to store which are used in the algorithm For each state and each potential bit input, a γ value is stored and the code to implement this can be found in Algorithm 1. This could have been implemented with loops, however, to improve code readability, it was left unrolled.

Algorithm 2 Calculation of β (2.5)

```
for state ← 0 to NUM_STATES do
    beta0 = next_beta[toState0] + gamma0[state];
    beta1 = next_beta[toState1] + gamma1[state];
    if state == 0 then
        beta_zero = max(beta0, beta1);
        beta[k-1][state] = 0.0; // normalize beta of state 0
    else
        beta[k-1][state] = max(beta0, beta1) - beta_zero; //normalize to state 0
```

Next the β values were calculated according to (2.5) using a loop over all states (Algorithm 2). In this algorithm, a probability is calculated for each branch of the turbo decoding trellis. This trellis is a representation of the state machine used in the turbo encoder to produce the parity bits. Each branch off of a state within the trellis represents the transition based on an input of either a 1 or 0, and the larger probability of each branch is normalized and used as the β term. The β calculation is recursive moving in reverse through the trellis so the value of the next β is used in the calculation of the current β. The values of β are also stored in a memory since the calculation of the LLR value requires all values for β. Finally, β values are normalized to the β value of the first state within the trellis so the dynamic range of the fixed point values is not exceeded.

Finally the α/LLR values were implemented in software according to Equations

Algorithm 3 Calculation of α and LLR (2.6)

```
for state ← 0 to NUM_STATES do
    alpha0 = prev_alpha[fromState0] + prev_gamma0[state];
    alpha1 = prev_alpha[fromState1] + prev_gamma1[state];
    if state == 0 then
        alpha_zero = max(alpha0, alpha1);
        alpha[state] = 0.0; // normalize beta of state 0
    else
        alpha[state] = max(alpha0, alpha1)) - alpha_zero; //normalize to state 0
    max0[state] = alpha[state] + gamma0[state] + beta[k][toState0];
    max1[state] = alpha[state] + gamma1[state] + beta[k][toState1];
out = maxV(max0) - maxV(max1)
```

2.6-2.8. This calculation is similar to the β calculations, however, it moves forward through the trellis so the previous α value is used. Another note is that the values for γ are buffered as the previous γ values are required for the α calculations and the current γ for the LLR. The maxV operation takes the overall max of the vectors max0, and max1 which are the maximum probabilities for all branches which represent an input of a 0 or 1 respectively. The subtraction of these probabilities creates the final LLR value.

Overall, care was taken to not re-use variables whenever possible to reduce potential false dependencies. This does not translate well into hardware since every variable is a separate memory, and if a variable is re-used for independent operations, hardware will be reused inefficiently.

With the core algorithms modeled in C code, the overall design of the turbo decoder was constructed. The initial approach attempted to implement the SISO as a function that matched the HSA. The resulting SISO implementation structure is shown in Figure 4.3. This figure shows how the algorithm is implemented in two main loops, the first to calculate γ/β and the second to calculate γ, α, and the final LLR values.

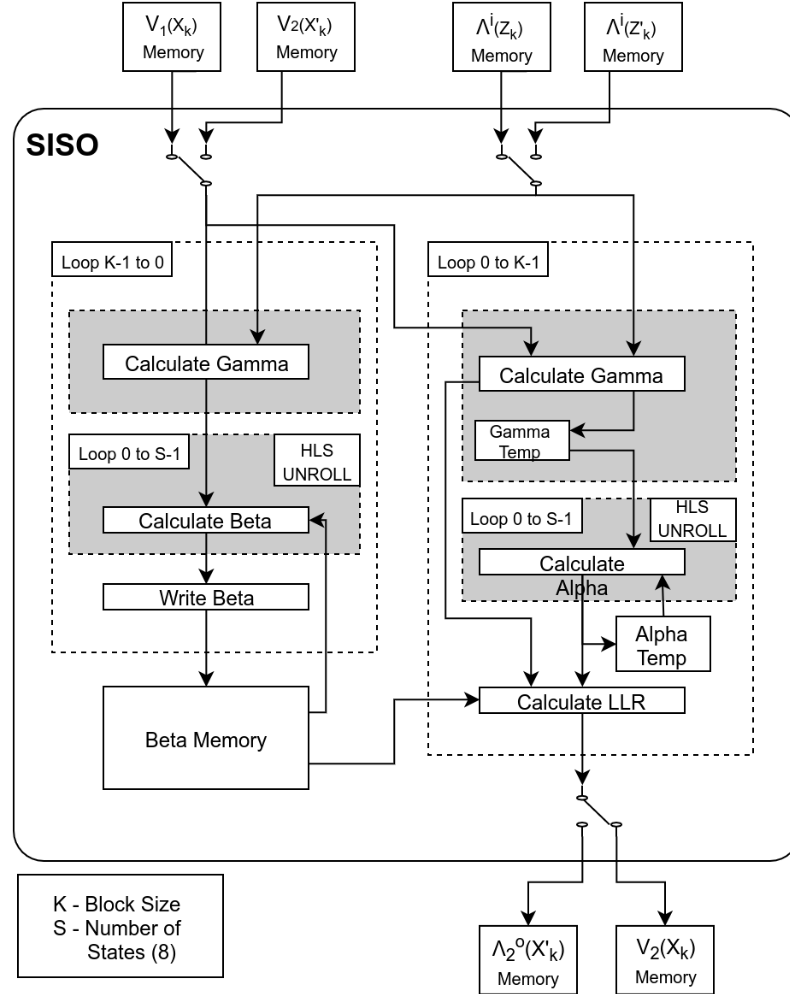


Figure 4.3: Data flow and looping structure of initial serial turbo decoder architecture modeled in C

4.1.3 Refining the Design

Once the algorithm was implemented in C, the design was then refined via code modifications and the addition of directives. The following outlines the modifications made to achieve the desired architecture.

- For all small local memories, the partition directive was applied with the complete parameter. This ensured that all local memories were implemented as registers in FF's. While the tools apply this optimization automatically in most scenarios, each memory was explicitly partitioned to ensure no scheduling

bottlenecks were created.

- The β memory was reshaped to increase the data in each word of memory. This was done since each calculation of β requires memory for each state and these calculations can be performed in parallel. This removes a memory bottleneck without an increase in the usage of hardware since the block RAMs within an FPGA are configurable.

- The α and β loops were unrolled completely as each calculation per state is independent and can occur in parallel.

- The second loop from 0 to K-1 was pipelined to replicate the pipeline within the HSA.

- The decoder in Figure 4.3 was flattened to include all memory accesses internal to the SISO function. This was done due to a bottleneck caused by data dependencies with out-of-order memory accesses when interleaving. This caused the algorithm to perform memory operations and then the SISO algorithm sequentially, whereas the desired functionality was to schedule the SISO algorithm as soon as the first memory location was available. The resulting architecture after moving the memory operations within the SISO is shown in Figure 4.4.

- The first loop was unable to be implemented in HLS such that each iteration completed in one clock cycle, so the loop was pipelined with an initiation interval of one. This was able to mitigate the bottleneck in HLS to better replicate the schedule of operations in the HSA.

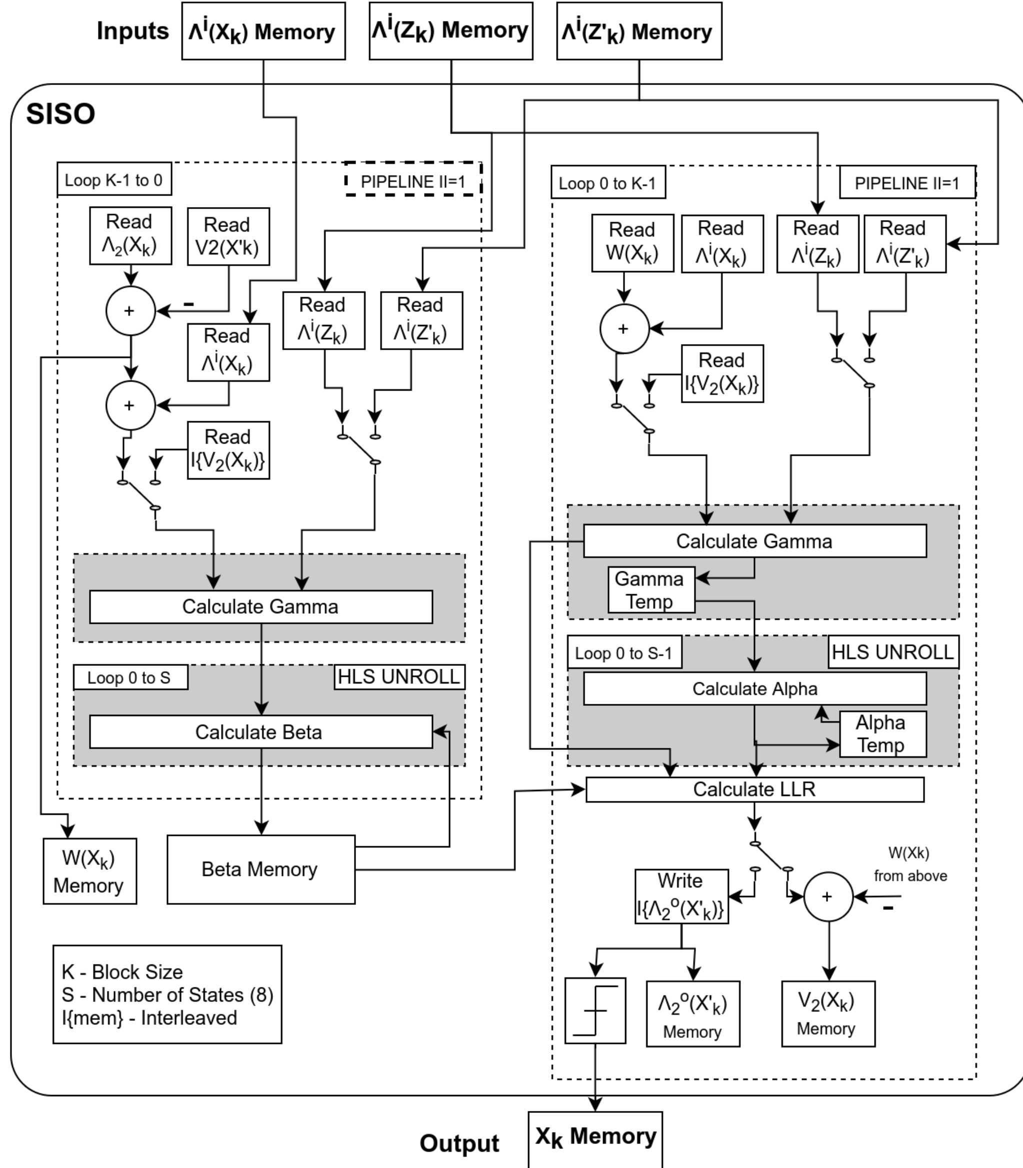


Figure 4.4: Improved data flow and looping structure of serial turbo decoder architecture modeled in C

4.2 Design 2: Parallel Decoder

4.2.1 Handcrafted Parallel Architecture (HPA)

The second design implemented was a parallelized turbo decoder which is based on the serial decoding architecture modeled in Section 4.1. For this document, this design from [2] will be referred to as the handcrafted parallel architecture (HPA).

The basis of this parallel decoder is to split the blocks of data operated on into small subsets. The SISO units are duplicated with each SISO operating on a subset of data. The parallel decoder will have some error correcting performance degradation due to operating on smaller blocks of data, but this is an acceptable trade off when high throughputs are required. The hardware architecture of this design is very similar to Figure 4.1 but with multiple SISO blocks. The scheduling of the design is also very similar to the serial design, and an example for parallelism of two is shown in Figure 4.5.

Figure 4.5 shows that the scheduling of operations is independent for each SISO, but memory operations are dependent. The main dependency between each SISO's

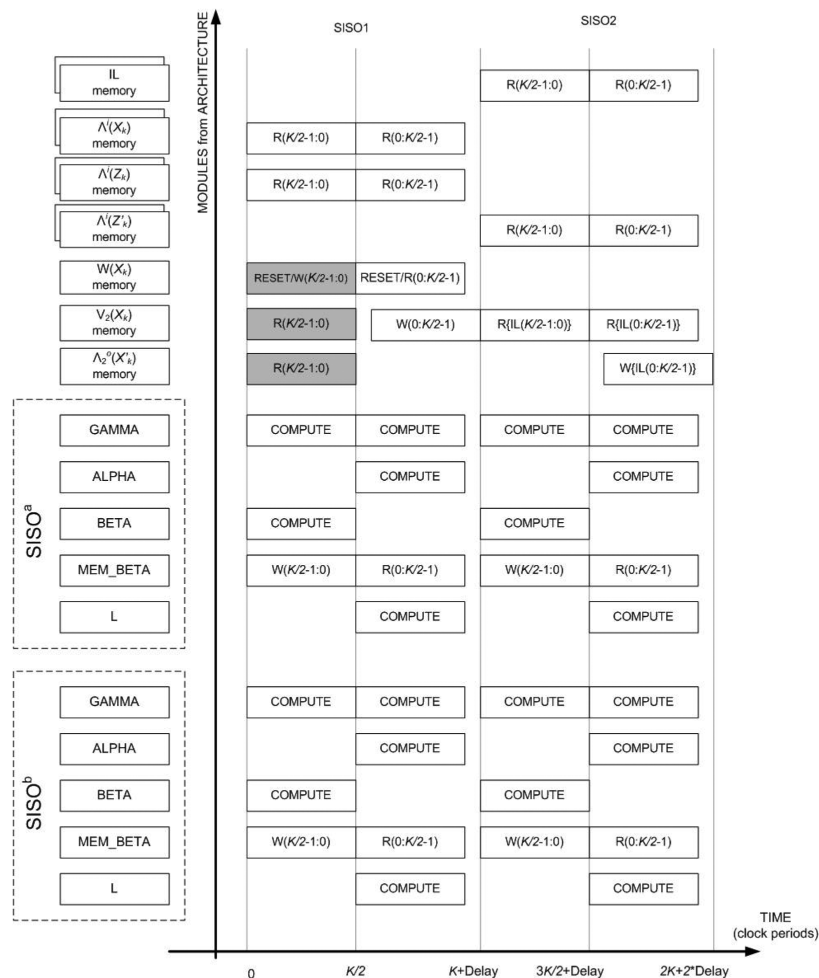


Figure 4.5: Parallel turbo decoder timing diagram [2]

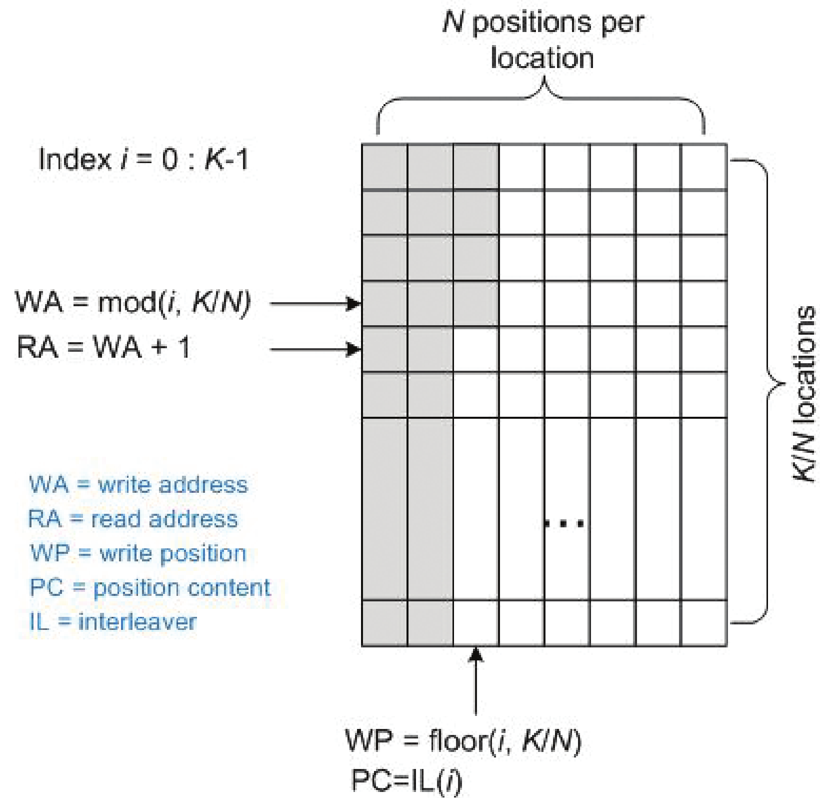


Figure 4.6: Parallel turbo decoder memory architecture [2]

memory is the interleaving and deinterleaving process. Interleaving is a pseudorandom process of swapping the position of the bits, and in the LTE specification, a Quadratic Permutation Polynomial (QPP) interleaver is utilized. This method of generating addresses to interleave the data will ensure a contention free interleaver as long as the block size is divisible by the level of parallelism [2]. The requirement of interleaving the data forces each parallel worker to be dependent on the others external to the SISO's. By design of the QPP interleaver, the memory required by all SISO's is contained within a row of data when represented as a 2D array (Figure 4.6). The act of interleaving with a QPP swaps rows of data and reorders the data within the row. This allows for an efficient interleaving process for parallel architectures while still maintaining acceptable error correcting performance. The memory architecture to achieve this requires an array with each word of the array containing the data elements for each SISO concatenated. For this architecture each data element of

10bits and with a parallelism of 8, each word of the memory would be 80bits wide by the block size divided by the parallelism. This allows a similar number of block rams to be used by the parallel design as the serial architecture. With the overall architecture changes determined compared to the HSA, an HLS implementation of the HPA can be developed.

4.2.2 Parallel Decoder HLS Implementation

Modeling the parallel architecture began with the final serial HLS implementation in C. The main alteration required involving duplicating the SISO's and modifying the memory architecture to allow for parallel memory reads. In the HPA, memory is organized into words such that each address of memory contains the values required for each SISO concatenated together. In the HPA a 10bit fixed point value is used for each memory, so for the parallel implementation, each word of memory is $10 * P$ where P is the level of parallelism. The data flow of this implementation for a parallelism of 2 is demonstrated in Figure 4.7.

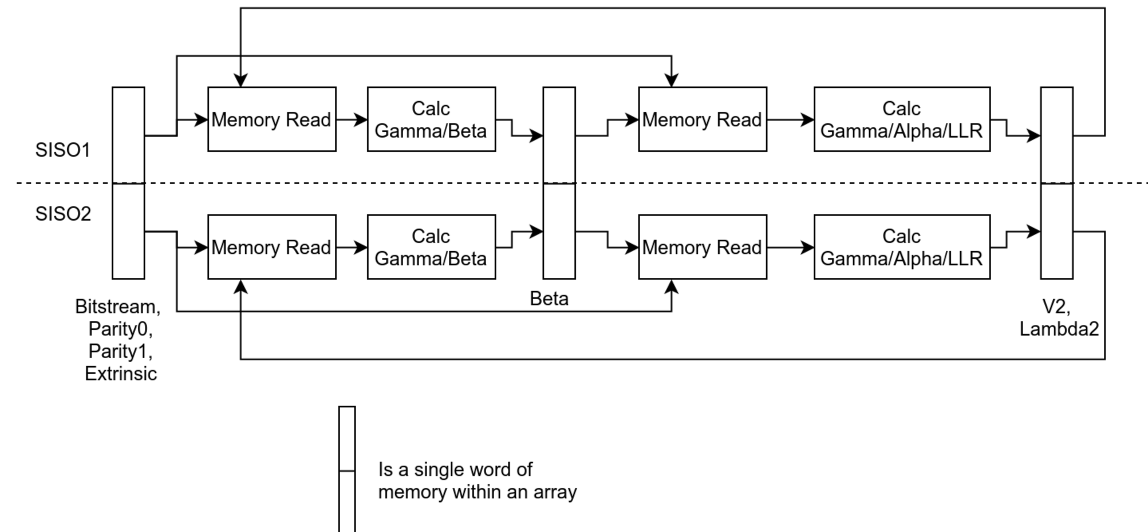


Figure 4.7: Parallel turbo decoder memory access architecture with a parallelism of two

With standard C this memory architecture would be very difficult to model. Vi-

vado HLS provides directives that make it possible by abstract the memory architecture from the C code. To implement this, a 2D array in C was utilized to implement the blocks of memory shown in Figure 4.6. The memory definition and directive applied are shown in Listing 8.

Listing 8 Directives for implementing a C array to increase the word length

```
bitstream[BLOCK_SIZE/PARALLEL][PARALLEL];
#pragma HLS ARRAY_RESHAPE variable=bitstream complete dim=2
```

The ARRAY_RESHAPE directive allows a developer to specify how memory is organized in hardware. In this case, it reshapes the array completely in the 2nd dimension. This effectively concatenates each row of the 2D array into a single word of memory. This allows a single read to receive all data required for all SISO's since they are all operating in parallel with the same operations, only on different blocks of data.

The next challenge arose to duplicate the SISO hardware to effectively parallelize the architecture. This was accomplished via additional loops that were unrolled in hardware. Unrolling loops in hardware is the same as duplicating hardware to parallelize the loops. The final architecture of the parallel decoder in HLS can be found in Figure 4.8.

The main consideration taken when parallelizing the architecture with loops was splitting each section of the algorithm up into independent operations and parallelizing them separately. This was especially important with memory operations due to the memory architecture. Since the memory contains all memory for parallel SISOs in one word, one memory read for each bank of memory must occur before all other operations. If the entire SISO loop was parallelized the tools would flag a false dependency due to the interleaving process and cause sequential operations on the memory since they are all within one word. The γ, β, and α loops were all parallelized separately for a similar reason to ensure that dependency paths are clear to the tools.

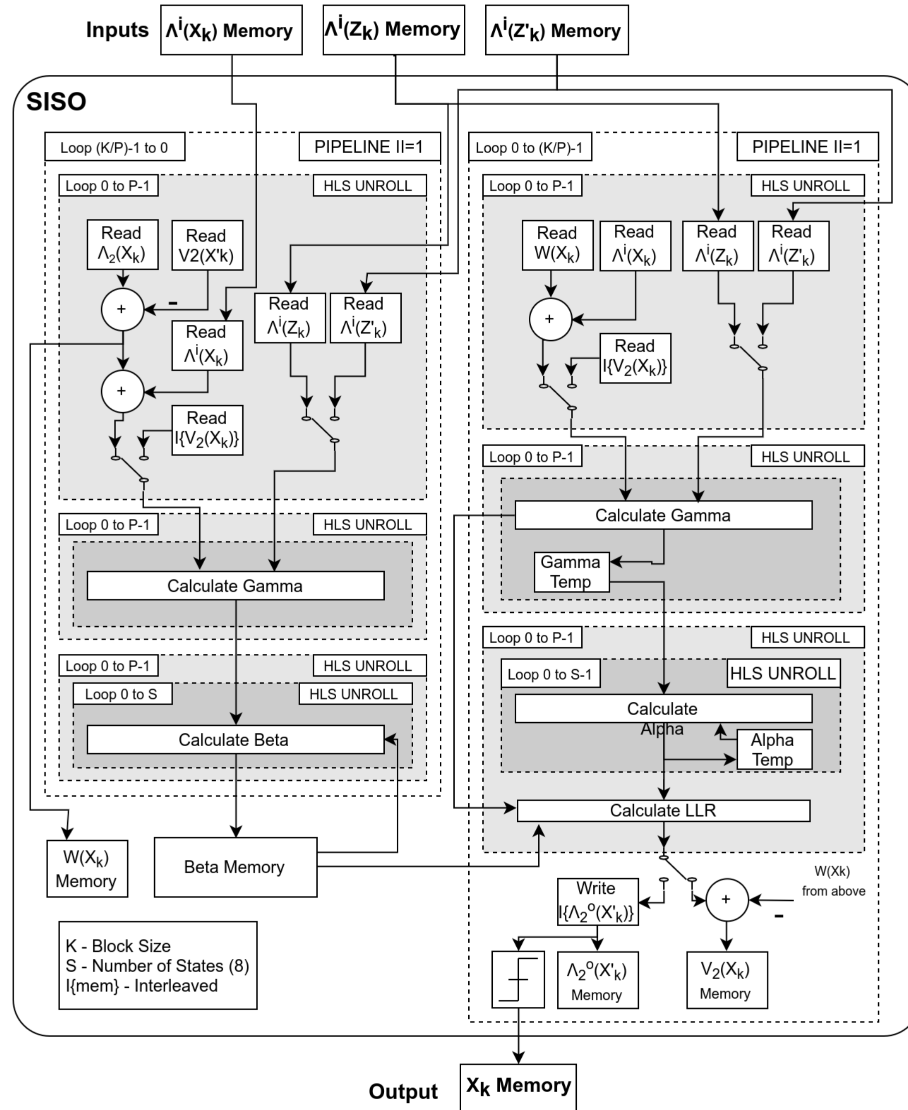


Figure 4.8: Parallel turbo decoder final architecture block diagram

With this architecture, there is no overhead with loops since they are unrolled completely to replicate hardware. Loops are utilized like a *for-generate* statement in an HDL to easily describe the replication of similar hardware. With this architecture, throughput parity should be achievable with the HPA.

4.3 Design 3: Parallel Decoder with Double Buffering

4.3.1 Double Buffering Handcrafted Architecture (DBHA)

The architecture in [4] presents an LTE parallel turbo decoder similar to [2] but with a double buffer technique to increase the throughput. To implement a double buffering system, the SISO algorithm was split into two stages, instead of two like in Design 1 and Design 2. The calculations for α and β were similar, however *alpha* was calculated i the first stage and buffered instead of β (Figure 4.9).

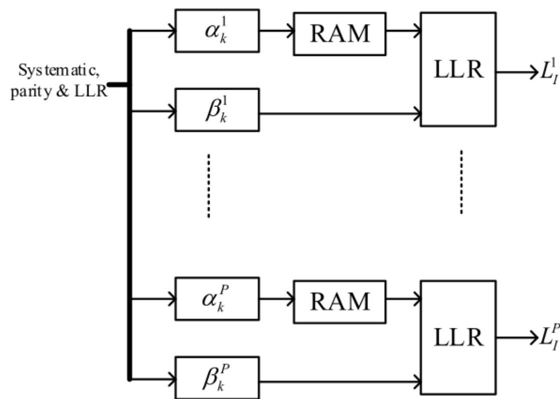


Figure 4.9: Parallel extrinsic calculation [4]

The DBHA also mitigates some of the error correcting performance losses due to parallelizing the algorithm by using the values of the prior iteration for α and β. This is demonstrated in Figure 4.9. For example, the first iteration for α, all SISO's uses α_0 as the initial α value, however, the next iterations use values from the prior iteration to provide a better estimate. This bridges the gap between sub-blocks which in a serial design would use the values from the same iteration to propagate α through the trellis. This allows for independence between parallel blocks in the current iterations without the full error correcting performance loss when operating on smaller blocks of data.

The largest change compared to the prior designs is the use of double buffering to

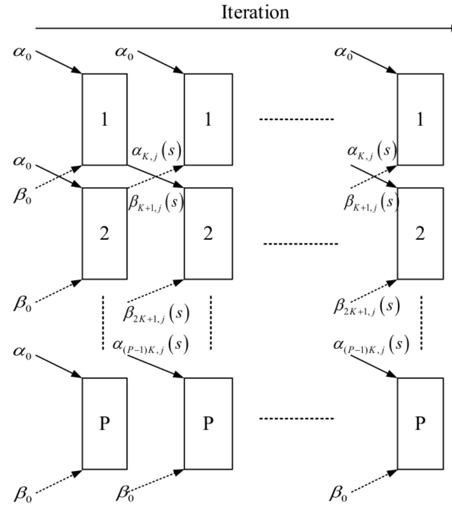


Figure 4.10: Parallel forward and backward metric calculations [4]

operate on two blocks of data. Since the α and β terms are calculated sequentially, half of the time the hardware for them is not being utilized. This can be improved by double buffering, at the cost of doubling the memory usage. The scheduling of the algorithm can be found in Figure 4.11. This diagram shows the α, β, and LLR operations and how double buffering can better utilize hardware, similar to a pipeline.

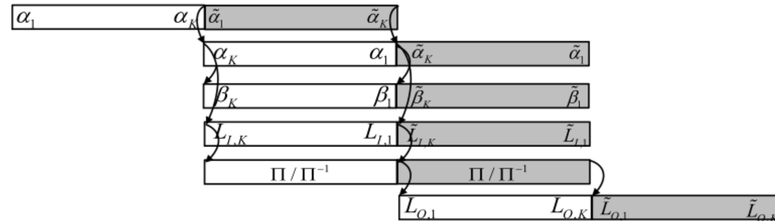


Figure 4.11: Double buffering of turbo decoder [4]

The double buffering of the algorithm causes a very similar effect to pipelining the design into 3 stages. The first, calculating α, next calculating β and the LLR, and finally interleaving and writing to the output memory. With this design, the throughput can be doubled with only twice the memory usage and a minimal impact on logic usage.

4.3.2 Double Buffering HLS Implementation

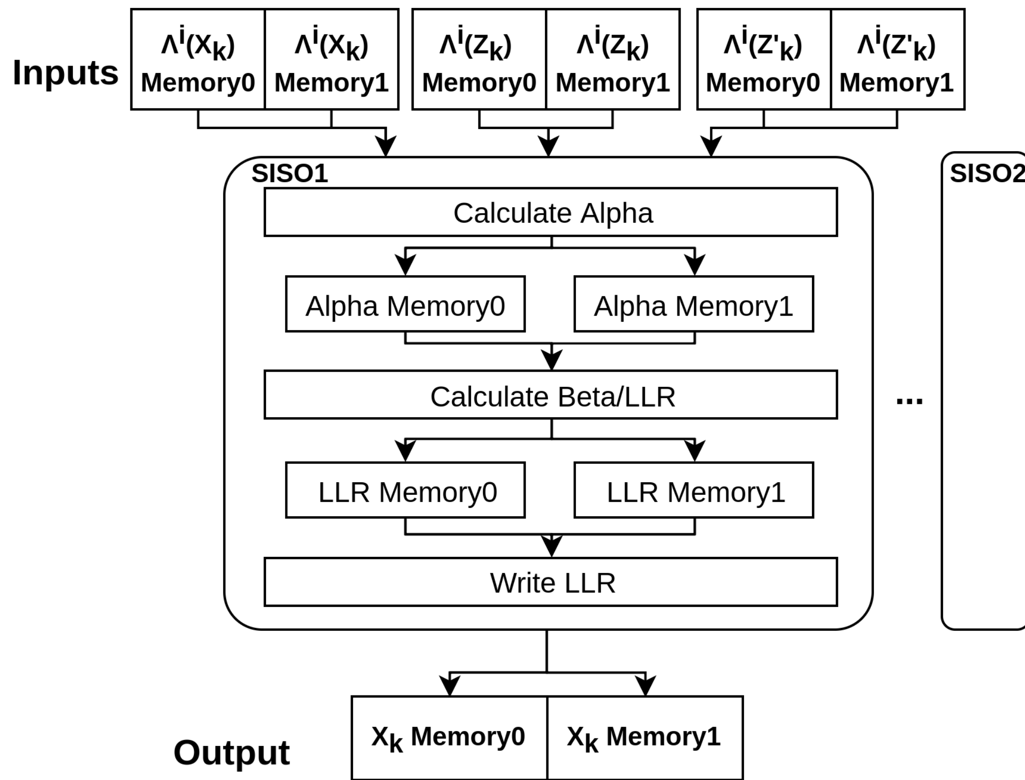


Figure 4.12: HLS turbo decoder block diagram for double buffering

The double buffering architecture drew heavily from the HPA's HLS implementation. The main requirement was to divide the algorithm into 3 stages and double all memory (Figure 4.12). For each input, output, and memory buffer between stages, a double buffer was used which is equivalent to an additional dimension added on to an array.

To allow for C to scheduling the design, each stage was made its own function. With each stage as a function, the double buffering algorithm can be implemented in C with function calls with different memory parameters (Listing 9).

With this method, the HLS tools can schedule each function call to produce the schedule of operations in Figure 4.11. The development time of this algorithm was very short compared to the design complexity. Most of this design was able to

Listing 9 Double buffering implementation in software

```
memory[2][BLOCK_SIZE/PARALLEL][PARALLEL]
alpha(memory[0])
alpha(memory[1])
beta/llr(memory[0])
beta/llr(memory[1])
write_llr(memory[0])
write_llr(memory[1])
```

reuse code from Design 2 due to the flexibility of an HLL and the abstraction out of scheduling the HLS provides. Similar to the addition of the prior designs pipeline to the first stage, this design required pipeline all stages with an II=1 to replicate the performance of the DHBA.

4.4 Summary of Designs

Table 4.1: A summary of all design architectures

Design 1	**My Design:** This design implemented a serial decoder by starting with a C implementation of the turbo decoding algorithm. The SISO algorithm was split into two main loops which were both pipelined to replicate the handcrafted schedule of operations. **Challenges:** The first half of the algorithm was unable to be executed in 1 clock cycle in HLS. The pipeline directive was used with an II of 1 to mitigate with only a small overhead. **Directives:** The array partition directive was used to split small memories into registers to remove bottlenecks. The unroll directive was used extensively to completely unroll loops to replicate similar functionality. Finally, the pipeline directive was used to remove bottlenecks in the first loop and model the pipeline in the second half of the SISO algorithm.
Design 2	**My Design:** This design parallelized the serial decoder from Design 1. This was accomplished through additional loops around each operation which were then unrolled. The memory architecture was also modified to allow for parallel memory accesses by reading blocks of memory in one read. **Challenges:** The main challenge was the memory design and how to describe it in C. The memory needed to be accessed in a block of memory which contained all elements of a row when represented as a matrix. **Directives:** The usage of the array reshape partition allowed for a two dimensional array in C to be implemented in long words in hardware. This simplified the memory access in the code, while still allowing the memory accesses to be parallelized for each row and replicate the handcrafted design.
Design 3	**My Design:** This design implemented a parallel decoder with double buffering to increase throughput. Design 2 was modified to be segmented into three stages instead of 2 with each stage in a separate function. Then through function calls with multiple memory banks, the design was double buffered. **Challenges:** The only challenge encountered was that each stage of the design could not be completed in one clock cycles, so similar to Design 1, each stage was pipelined to mitigate this. **Directives:** The pipeline directive was used to pipeline each stage to mitigate bottlenecks.

Chapter 5

Results and Analysis

To evaluate the HLS implementations of the three turbo decoder architectures, the throughputs are compared between the handcrafted and HLS designs. Latency equations are provided for Design 1 and Design 2 in [2] and for Design 3 in [4]. With the design latency and the reported clock speeds, the throughputs can be calculated for comparison. The reported latencies for Design 1, Design 2, and Design 3 are (5.1), (5.2), and (5.3) respectively.

$$L_{Design1} = (4 * N + 2 * Delay) * I \text{ [2]} \tag{5.1}$$

$$L_{Design2} = (\frac{4 * N}{P} + 2 * Delay) * I \text{ [2]} \tag{5.2}$$

$$L_{Design3} = \frac{3N}{P} * (I + 1) + \Delta t \text{ [4]} \tag{5.3}$$

Where N is the block size, P is the parallelism, I is the number of iterations, $Delay$ is the delay of 11 cycles in [2] and Δt is the implementation specific delays [4]. Latencies were calculated for a block size of 6144 and parallelism of eight. Designs 1 and 2 were calculated for three iterations, and Design 3 for eight iterations to mirror the results from each design's respective paper. Δt was not provided for the implementation in [4], so the best case performance was calculated using $\Delta t = 0$. With the latency for each design, the throughput of the decoders is calculated with Equation 5.4. For Design 1 and Design 2, $F_{clk} = 210Mhz$ and $K = 6144$, and for Design 3 $F_{clk} = 250Mhz$ and $K = 12288$. Design 3's block size for the decoder is

6144, however, due to double buffering two blocks of data are decoded, so $K = 12288$ is used.

$$Throughput = \frac{K}{L * \frac{1}{F_{clk}}} \tag{5.4}$$

The throughputs of the HLS architectures are compared against the throughput of the handcrafted architectures within [2] and [4] to gauge the quality of the HLS implementations. The handcrafted throughputs are calculated according to (5.4) with the provided latency equation for each design. To provide a fair comparison since the designs are implemented on different generations of FPGAs, the HLS design throughputs are calculated with the same clock frequency and parameters as the handcrafted designs. When designing with HLS, the clock constraint was varied from 2ns to 10ns, in 1ns increments. The clock constraint which produced a final clock period lower than the desired clock period and had the smallest latency was chosen for the best case comparison. The results are shown in Table 5.1 for all design implementations.

Table 5.1: Results for all handcrafted and HLS designs

Implementations	**Design 1**		**Design 2**		**Design 3**	
	[2] Serial	**HLS 1**	**[2] Parallel**	**HLS 2**	**[4]**	**HLS 3**
Iterations	3	3	3	3	8 [a]	8 [a]
Block Size	6144	6144	6144	6144	6144	6144
P	1	1	8	8	8	8
FPGA	XC5VFX70T	XCZU9EG	XC5VFX70T	XCZU9EG	XC7VX690T	XCZU9EG
FPGA Technology	65 nm	16 nm	65 nm	16 nm	28 nm	16 nm
Clock Constraint (ns)	-	7.0	-	6.0	-	4.0
Clock Frequency (MHz)	210	210	210	210	250	250
Latency (μs)	351.40	351.40	44.20	44.31	82.94	99.53
Throughput (Mbps)	17.48	17.48	139.00	138.65	148.15	123.46
% Difference	0.0		0.3		18.2	

[a] Design 3's iterations are equivalent to a half-iteration of the turbo decoding algorithm.

To analyze the final throughputs for each design, the throughputs are plotted as bar graphs with the HLS and Handcrafted throughputs next to each other (Fig-

ure 5.1). This figure allows for the comparison between the designs for their throughput with matching parameters between HLS and handcrafted designs.

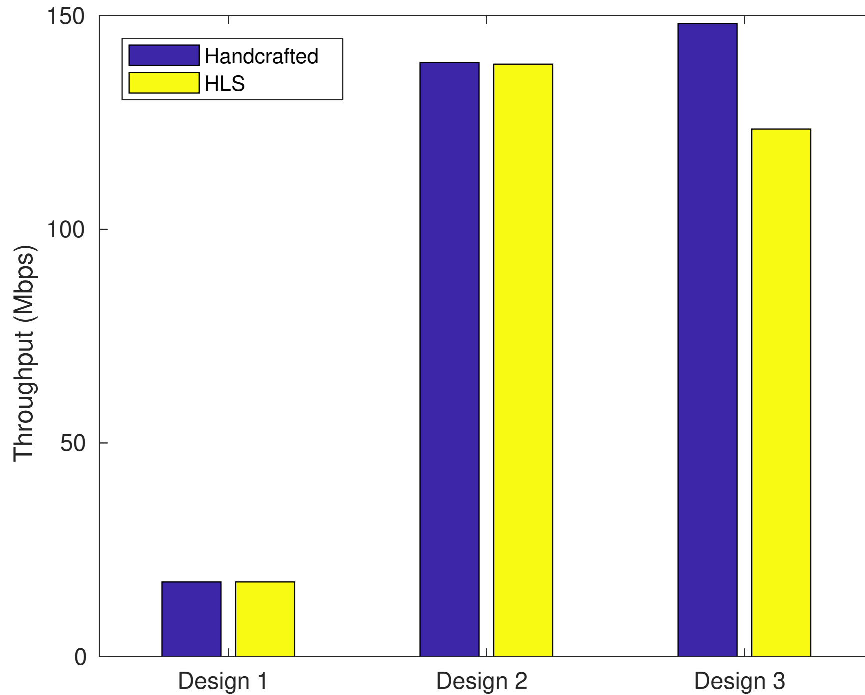


Figure 5.1: Throughput comparison between handcrafted and HLS implementations

The throughputs for Design 1 and Design 2 are nearly identical between the handcrafted and HLS implementations, with the percent difference being 0 for Design 1 and 0.3% for Design 2. This was expected given that the HLS designs were able to replicate the schedule of operations in the handcrafted architecture. This was accomplished by adding an additional pipeline with II=1 to bypass the bottleneck in the first half of the SISO algorithm. This additional pipeline did add a small overhead of a few clock cycles which explains the small reduction in throughput for Design 2 due to pipeline ramp-up and ramp-down. Design 3, on the other hand, has a larger discrepancy between HLS and Handcrafted throughputs. This could be due to how the architecture of the paper was interpreted. Equation 5.3 gives the latency as $\frac{3*N}{P}$, however with the double buffering technique in Figure 4.11, each iteration should take $\frac{4*N}{P}$ clock cycles to process two blocks of data. [4] may have implemented fur-

ther optimizations when scheduling the operations to achieve lower latency. Since the HLS design was based on the diagrams contained within the paper, a comparison to the ideal scheduling of operations can also be made. The latency equation also multiplies the iteration latency by $I + 1$, where I is the number of iterations. With the alternative latency for each iteration, this will only need to be multiplied by I. With these modifications, an alternate latency equations is shown in (5.5).

$$L_{Design3} = \frac{4 * N}{P} * I \tag{5.5}$$

Equation 5.5 models the architecture implemented in HLS more accurately than the provided equation in [4]. Without the HDL code from the paper, it is difficult to determine the specific scheduling of their design and how they were able to achieve a better latency. The throughputs and percent differences were recalculated and are shown in Table 5.2.

Table 5.2: Alternate handcrafted throughput for Design 3 based on the latency of each iteration shown in Figure 4.11

Throughput (Mbps)		% Difference
Handcrafted	HLS	
125.46	123.46	1.2

Figure 5.2 shows the throughput comparison of the HLS to Handcrafted designs with the alternate latency equation for Design 3. Design 3's HLS implementation is must closer to the throughput of the handcrafted. The handcrafted design has a throughput of 125 Mbps which is slightly larger than the 123.4 Mbps of the HLS design, though this was expected due to the overhead from the ramp-up and ramp-down of the pipelines for each stage. For this design, the percent difference is greater than Design 1 or Design 2 because of the addition of three pipelines to the design to bypass all bottlenecks in the HLS implementation. This results in a larger overhead from the pipelines, but the HLS implementations are still only 1.24% off from the handcrafted design.

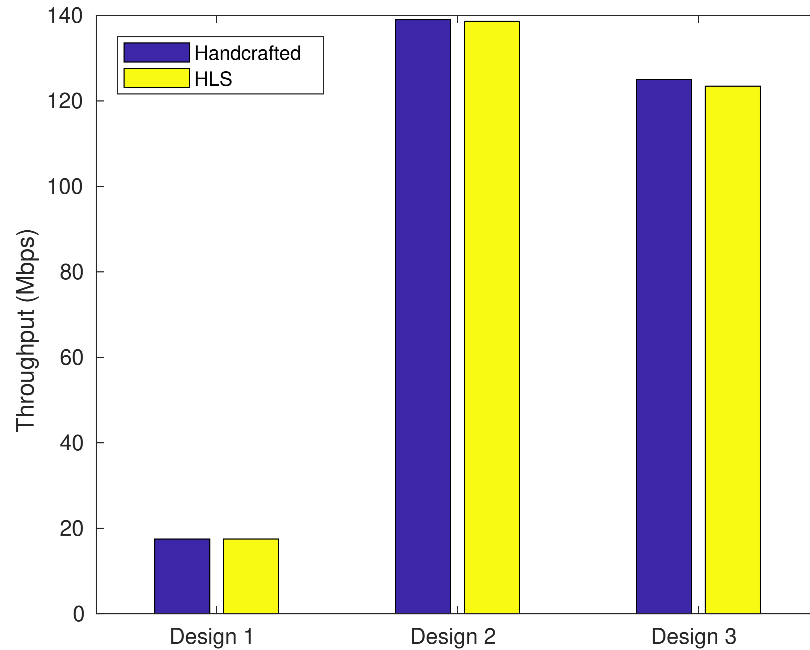


Figure 5.2: Throughput comparison between handcrafted and HLS implementations with alternate latency for Design 3

The resource usage between the HLS designs was evaluated to determine how efficiently the HLS tools were able to implement the parallel architectures. The resource usage for each HLS design from Table 5.1.

Table 5.3: Comparison of resource usage between HLS designs

Implementation	Design 1	Design 2	Design 3
Clock Constraint (ns)	7.0	6.0	4.0
BRAM	57	50	187
FF	687	6210	13317
LUT	2114	17355	31536
Latency	73794	9306	24883
Max Clock Frequency (MHz)	220	210	268
Throughput (Mbps)	18.31	138.96	132.64

Table 5.3 shows the resource usage for each HLS implementation with the clock constraints used from the comparison against handcrafted designs above. Design 2 is a parallelized version of Design 1 and parallelism of 8 was used for collecting data.

There is a 9X increase in FF usage and an 8.2X increase in LUT usage between Design 1 and Design 2 which is expected due to duplicating the hardware 8 times plus the overhead with routing and multiplexing memory for a parallel design. The BRAM usage was close with 57 used for Design 1, and 50 used for Design 2. This variance can be explained by the tools using more BRAMs due to routing differences when scheduling the design with different clock constraints. Design 1 used 48 BRAMs for an 8.0ns clock constraint which is nearly identical to Design 2.

For Design 3 there is a 3.74X increase in BRAM, 2.1X increase in FF and a 1.8X increase in LUTs compared to Design 2. The increase in BRAM usage is expected due to double buffering and an additional stage in the decoding algorithm. The storing of the β and LLR terms dominated the BRAM usage of the decoder, and with double buffering, two times the memory is used. The LUT and FF usage increases are from the additional stage since it required additional memory to buffer the data and also an additional pipeline within the stage.

Overall from the results obtained, an HLS design flow was able to produce designs that were able to match their handcrafted counterparts. While there was a small performance loss due to the addition of the pipelines, it is negligible for the design and could be mitigated with a faster clock depending on design requirements. The clock speed of a hardware implementation is an important design decision as it had a direct impact on the resource usage and overall design of a system. For HLS this is varied via a clock constraint used by the HLS scheduler. To determine how well HLS can achieve a clock constraint and its impact, each design was implemented with a clock constraint of 2ns to 10ns in 1ns increments.

5.1 Clock Constraint Impact on Designs

For each design implemented in HLS, the clock constraint parameter was varied and design metrics were recorded for each stage of the HLS design flow. The clock con-

straint is a parameter for the HLS tools and impacts the HLS scheduler's approach to segmenting an algorithm and implementing it in hardware. This parameter was varied to help determine if a hardware designer's intentions with the clock have an intended impact on the final design.

The data collected for Design 1, Design 2, and Design 3 are within Tables 5.4, 5.5, and 5.6, respectively. For each design, the clock constraint is shown on the very left and is the only parameter varied. The HLS section gives the estimated resource usage and clock period from the HLS C synthesis report. Finally, the implementation section provides resource usage, clock period, and design throughput after hardware synthesis and place and route.

Table 5.4: Design 1 HLS and implementation results for clock constraints from 2ns to 10ns with a block size of 6144

	HLS				**Implementation**					
Clock Constraint	**Est. BRAM**	**Est. FF**	**Est. LUT**	**Est. Clock Period**	**BRAM**	**FF**	**LUT**	**Clock Period**	**Latency (clk cycles)**	**Throughput (Mbps)**
2.0	45	2683	7295	4.501	57	1659	2325	2.512	73884	33.10
3.0	45	1742	7187	4.501	57	1133	2436	2.671	73839	31.15
4.0	45	1656	7187	4.501	57	1131	2315	3.017	73821	27.59
5.0	45	1616	7184	7.007	57	922	2206	3.757	73812	22.16
6.0	45	1126	7120	5.191	57	762	2150	4.176	73806	19.93
7.0	45	1066	7057	6.110	57	687	2114	4.546	73794	18.31
8.0	45	1001	7057	6.428	48	623	2117	5.350	73794	15.56
9.0	45	999	7088	7.625	48	733	2293	5.603	73794	14.86
10.0	45	974	7088	8.237	48	695	2294	6.160	73788	13.52

To determine the impact of the HLS tools and the clock constraint on the resulting implementation, the clock, resources, and throughput are plotted. When designing with HLS it is important to understand how the clock constraint parameter impacts the final design. Within the HLS design flow, there are two places where a design can

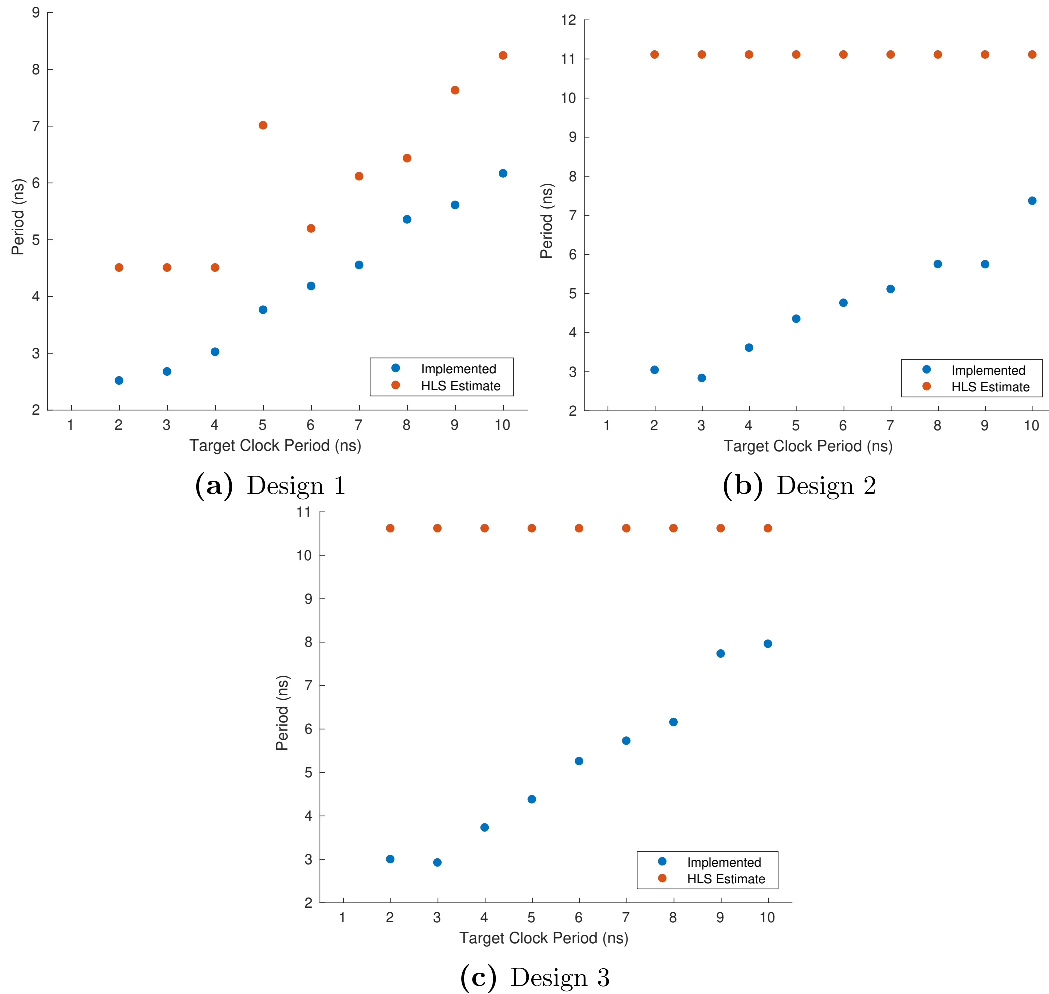
Table 5.5: Design 2 HLS and implementation results for clock constraints from 2ns to 10ns with a block size of 6144 with a parallelism of 8

Clock Constraint	HLS				Implementation					
	Est. BRAM	Est. FF	Est. LUT	Est. Clock Period	BRAM	FF	LUT	Clock Period	Latency (clk cycles)	Throughput (Mbps)
2.0	54	23909	67955	11.102	50	14542	19818	3.036	9388	215.56
3.0	54	16026	67760	11.102	50	9197	19447	2.828	9344	232.51
4.0	54	15399	67696	11.102	50	8802	18350	3.603	9326	182.85
5.0	54	14623	67885	11.102	50	7292	17610	4.343	9312	151.92
6.0	54	12622	67437	11.102	50	6210	17355	4.751	9306	138.96
7.0	54	11414	66933	11.102	50	4954	17643	5.104	9294	129.52
8.0	54	10628	66773	11.102	50	4496	18116	5.742	9294	115.13
9.0	54	10334	66773	11.102	50	4248	17598	5.739	9294	115.19
10.0	54	8797	65589	11.102	50	4669	17922	7.361	9288	89.87

Table 5.6: Design 3 HLS and implementation results for clock constraints from 2ns to 10ns with a block size of 6144 and parallelism of 8

Clock Constraint	HLS				Implementation					
	Est. BRAM	Est. FF	Est. LUT	Est. Clock Period	BRAM	FF	LUT	Clock Period	Latency (clk cycles)	Throughput (Mbps)
2.0	187	33682	97914	10.612	187	23614	32898	2.994	25030	163.97
3.0	187	22231	97621	10.612	187	16042	32669	2.917	24931	168.97
4.0	187	20000	97333	10.612	187	13317	31536	3.723	24883	132.64
5.0	187	18835	97615	10.612	184	11103	29044	4.372	24856	113.08
6.0	187	15671	97103	10.612	187	9121	28901	5.251	24832	94.24
7.0	187	13828	96599	10.612	187	7185	28060	5.720	24808	86.60
8.0	187	12783	96439	10.612	187	6460	29358	6.149	24808	80.55
9.0	187	10917	95180	10.612	187	6746	29474	7.727	24784	64.17
10.0	187	11249	95692	10.612	187	6542	29436	7.952	24784	62.35

be evaluated. The first after C synthesis in HLS where the estimated clock period and resource usage are reported. The second is after the design is exported and it undergoes hardware synthesis and place and route where the final resource usage and clock period are reported.



(a) Design 1 **(b)** Design 2

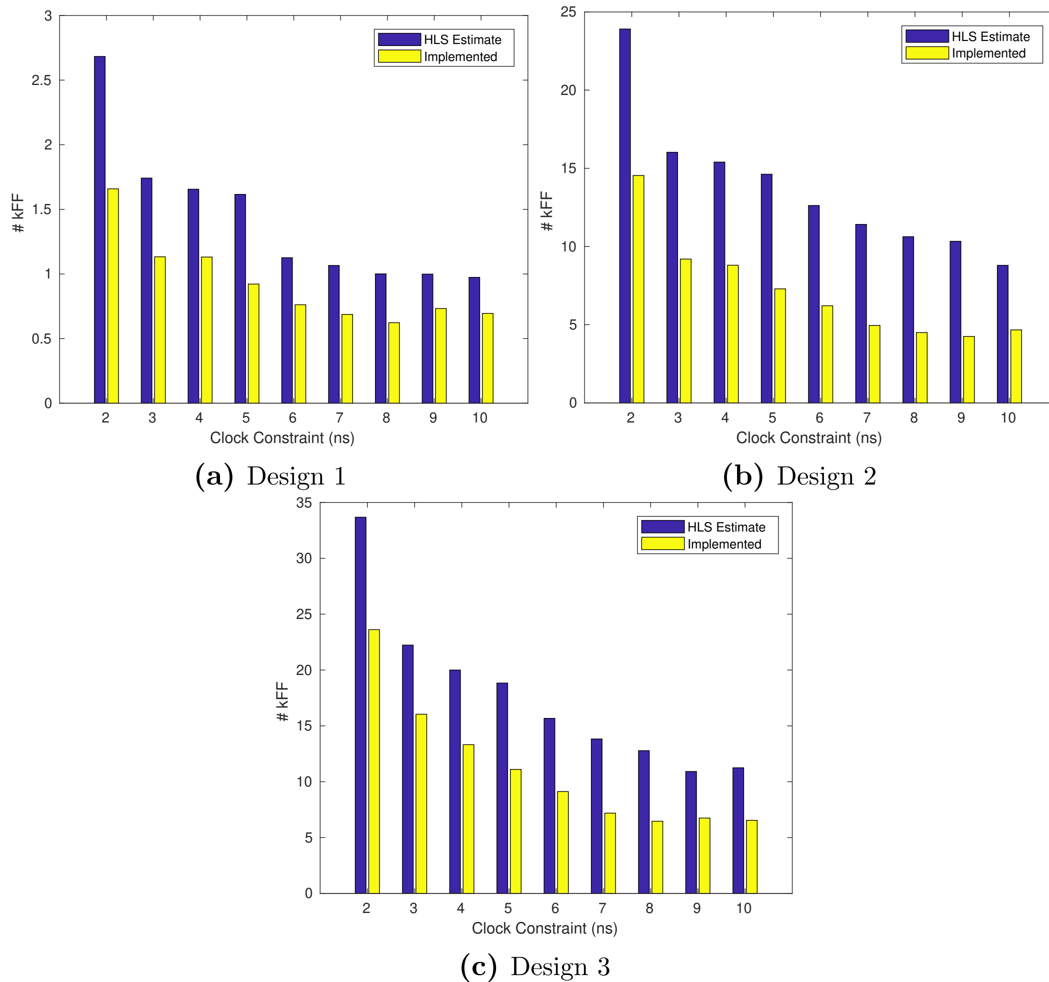
(c) Design 3

Figure 5.3: Graph of the clock constraints impact on the designs minimum clock period reported after HLS synthesis and implementation

To investigate the variance between these evaluation periods, the clock constraint is plotted against the estimate provided by the HLS tools and the implemented clock constraint after place and route (Figure 5.3). These figures show that the HLS estimate for the clock period is not accurate. For Design 1 in Figure 5.3a, the HLS estimated clock follows a similar trend to the implemented clock, however gives a

conservative estimate. For Design 2 and Design 3, however, the estimated clock is a constant value for all clock constraints and does not estimate the implemented clock. The implemented clock follows the expected trend of decreasing the clock period, so this is a result of the estimation, rather than poor scheduling by the tools. To an engineer using HLS tools, the HLS estimated clock period should not be used in its current state to gauge the quality of a design. The final implemented clock should be verified for all evaluation.

The HLS tools also provide estimates for the resource usage within the targeted FPGA. In order to determine if the estimates are accurate, the estimated and imple-



(a) Design 1 **(b)** Design 2

(c) Design 3

Figure 5.4: Bar graph of the clock constraints impact on the FF usage reported after HLS synthesis and implementation.

mented resource usage for flip-flops (FF) are plotted in Figure 5.4.

Figure 5.4 shows the HLS estimated and implemented FF usage for each design with a varying clock constraint. The HLS estimates for the FF usage are greater than the implemented usage, but still follow a similar trend to the implemented resources. Based on these results the HLS estimates could be a good indicator of the actual usage if a scaling factor was applied. The issue with this approach is that the scaling factor may vary between designs and tool versions and would require further investigation.

Figure 5.4 also demonstrates how the tools approach constrained clocks. From 3ns to 10ns for all designs, the FF usage rises in small increments. For the 2ns constraint,

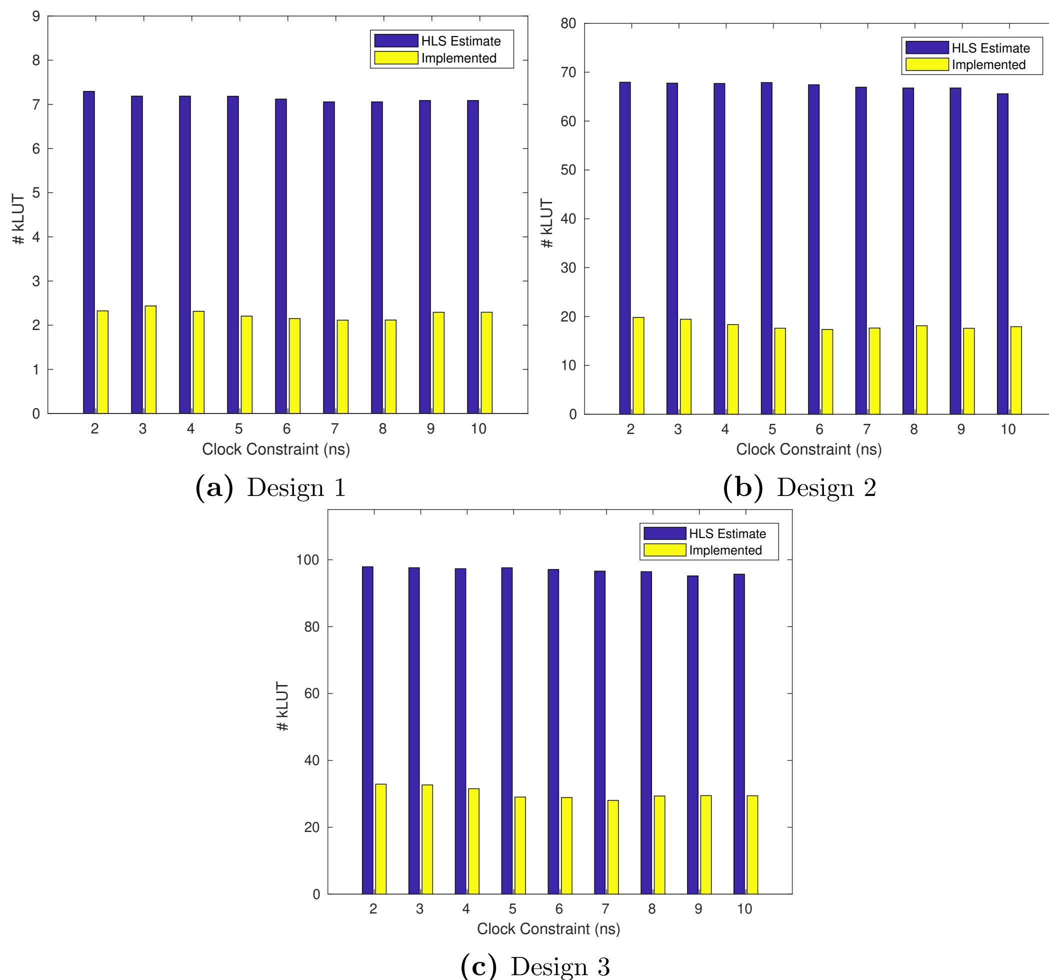


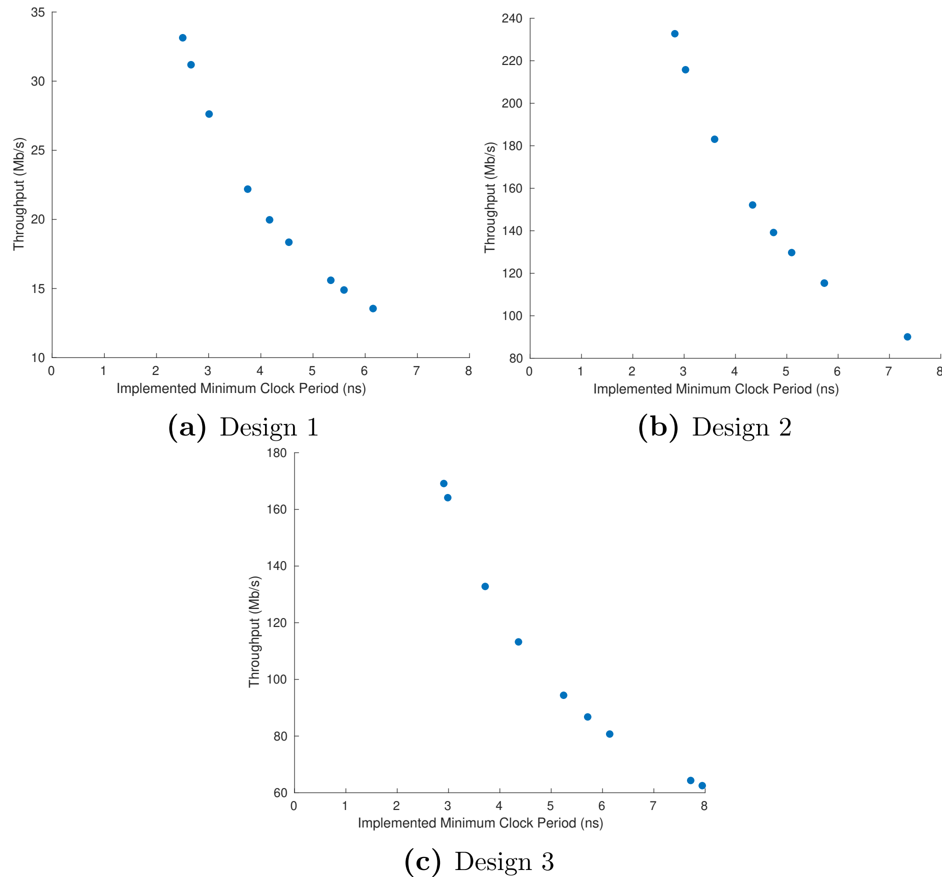
Figure 5.5: Bar graph of the clock constraints impact on the LUT usage reported after HLS synthesis and after implementation.

the clock constraint was not met and as a result, the FF usage spikes in the tools attempt to lower the clock period. Based on Figure 5.3, this spike has a minimal impact on implemented clock period. Care must be taken when pushing a design to its limits since the tools do not limit themselves in the attempt to reach a clock period. To investigate the impact of the clock constraint on the lookup table (LUT) usage, the data was plotted in a similar manner (Figure 5.5).

Figure 5.5 shows a similar trend to the FF usage where the HLS estimates are greater than the implemented resource usage. An unachievable clock constraint does not impact the LUT usage like the FF usage. This was expected since the designs were pipelined and to reduce the clock period, more pipeline stages are added, which use FF's and not LUTs. While this explains the FF usage increase, it does not mean that it is acceptable for quality hardware designs. The tools do most of the work for scheduling, but it is up to the designer to guide the tools with a reasonable clock constraint to provide a design that meets a set of requirements.

The throughput can also be used as a measure of the tools and their effectiveness. To investigate this the throughput for each design is plotted against the implemented clock period and is shown in Figure 5.6. This was done to investigate the quality of the resulting architectures for each design taking into consideration the clock period and also the impact on the latency. This is done by calculating the throughput for each design. From these graphs, it's clear that a smaller clock period resulted in higher throughput. The tools were able to change the scheduling of operations based on the clock constraint and do so without large adverse impacts on the latency. This can be expected due to all designs making use of pipelining for all stages of the algorithm. With a pipeline, the clock constraint impacts the number of stages of the pipeline, without impacting the initiation interval which would have a larger impact on latency.

Overall, the results presented demonstrate that HLS designs can reach handcrafted



(a) Design 1

(b) Design 2

(c) Design 3

Figure 5.6: Graph of the implemented clock period's impact on maximum throughput for each design. The block size is 6144 and the parallelism is 8 for Design 2 and Design 3.

levels of throughput performance for turbo decoder designs. While the tools adversely impacted the FF resource usage, this can be controlled by the designer by setting the clock constraint to a value that the tools can achieve. The flexibly of the HLS tools makes this process simple and can be automated to search for a desired trade off of resource usage, throughput, and minimum clock period. In this design process, the current tools do not provide a good estimate for the clock period or resource usage. With the tool's ability to analyze the design with a schedule viewer, rapid development can still be attained, however, evaluation of the resource usage and clock must be done after hardware synthesis and implementation, similar to handcrafted HDL development.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

This work focused on exploring coding techniques and a high level synthesis design flow to target turbo decoder architectures. By modeling three existing handcrafted turbo decoder architectures, the HLS implementations could be analyzed. The first step was extracting each handcrafted architecture from prior research papers. The algorithm and functionality of each architecture were then implemented in C/C++. Modifications were then made to the code along with the addition of directives to guide the tools with hardware specific optimizations. After each round of modifications, the C code was then processed through the tools and evaluated until the desired architecture was reached. Bottlenecks due to HLS were mitigated where possible to ensure that the HLS design could reach handcrafted levels of performance. A final verification occurred after exporting the design and performing hardware synthesis to determine the final latency, clock period, and resource usage.

When designing with HLS, a clock constraint can be provided to the tools to impact the scheduling of the algorithm. A study was performed by varying the clock constraint for each design to look at its impact on the performance and resource usage. The accuracy of the estimated clock period and resource usage reported by the HLS process was also evaluated.

The overall conclusion of this work is that its possible to achieve handcrafted levels of performance with an HLS design flow for turbo decoder architectures. The turbo decoding algorithm is computationally intensive and requires complex memory accesses which pose a challenge for HLS, however major bottlenecks in HLS were overcome with the additional pipelines. This mitigation technique was successful for the turbo decoding algorithm due to few data dependencies, however, its effectiveness will vary depending on the algorithm targeted.

Care must be taken when using HLS tools for hardware development. Processing code through the HLS tools produces estimates for the clock period and resource usage, however, these were found to be inaccurate. For two out of three designs the estimates clock period was a constant around 11ns, whereas the implemented clock period varied from 3ns to 8ns depending on the clock constraint. The resource usage estimates were overall larger than the implemented resource usage and predicted the trend of the implemented recourse usage. In their current state, their usefulness is limited and an evaluation of the design should only occur after hardware synthesis.

6.2 Future Work

While this work showed that the turbo decoder architectures were able to be implemented in HLS and achieve handcrafted levels of performance, there is more work to be done before HLS can replace the standard handcrafted RTL design flow for hardware. Exploring the use of HLS for implementing other digital signal processing algorithms to implement a software defined radio in HLS could show domains where current tools could be utilized. Additionally, further research into development in HLS vs handcrafting RTL hardware design by implementing a custom turbo decoder could allow for more insight into the benefits and limitations of HLS.

Finally a major benefit of the HLS tools it's the quick feedback provided. Unfortunately, the estimates provided by the tools in this work were not accurate or reliable

enough to be relied upon. Further research into how the estimates are generated or impacted by design elements could allow a designer to more efficiently use the HLS tools.

Bibliography

[1] T. F. Collins, R. Getz, D. Pu, and A. M. Wyglinski, *Software-Defined Radio for Engineers.* Artech House, 2018.

[2] C. Anghel, C. Stanciu, and C. Paleologu, "Efficient FPGA Implementation of a CTC Turbo Decoder for WiMAX / LTE Mobile Systems."

[3] *Vivado HLS Optimization Methodology Guide*, v2018.1 ed., Xilinx, 2018.

[4] H. Luo, Y. Zhang, W. Li, L. K. Huang, J. Cosmas, D. Li, C. Maple, and X. Zhang, "Low Latency Parallel Turbo Decoding Implementation for Future Terrestrial Broadcasting Systems," *IEEE Transactions on Broadcasting*, vol. 64, no. 1, pp. 96–104, 2018.

[5] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Proceedings of ICC '93 - IEEE International Conference on Communications*, vol. 2, May 1993, pp. 1064–1070 vol.2.

[6] 3rd Generation Parntership Project, "LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding," *(3GPP TS 36.212 version 11.1.0 Release 11)*, vol. 0, 2013.

[7] D. J. Daly and D. J. Daly, "Economics 2: Ec2," 1987. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[8] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019.

[9] B. E. Conn, "Exploring High Level Synthesis to Improve the Design of Turbo Code Error Correction in a Software Defined Radio Context," *ProQuest Dissertations and Theses*, p. 109, 2018. [Online]. Available: http://search.proquest.com.ezproxy.rit.edu/docview/2097157862?accountid=108

[10] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "From low-architectural expertise up to high-throughput non-binary ldpc decoders: Optimization guidelines using high-level synthesis," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–8.

[11] D. Bell, S. Allen, N. Chamberlain, M. Danos, C. Edwards, R. Gladden, D. Herman, S. Huh, P. Ilott, T. Jedrey, T. Khanampornpan, A. Kwok, R. Mendoza,

K. Peters, S. Sburlan, M. Shihabi, and R. Thomas, “Mro relay telecom support of mars science laboratory surface operations,” in *2014 IEEE Aerospace Conference*, March 2014, pp. 1–10.

[12] P. Angeletti, M. Lisi, and P. Tognolatti, “Software defined radio: A key technology for flexibility and reconfigurability in space applications,” in *2014 IEEE Metrology for Aerospace (MetroAeroSpace)*, May 2014, pp. 399–403.

[13] R. Ludwig and J. Taylor, *Voyager Telecommunications*. Jet Propulsion Laboratory, 2014, ch. 3, pp. 63–70. [Online]. Available: https://descanso.jpl.nasa.gov/monograph/series13/DeepCommoOverall--141030A_ama.pdf

[14] “About RTL-SDR,” Aug 2019. [Online]. Available: https://www.rtl-sdr.com/about-rtl-sdr/

[15] “HackRF.” [Online]. Available: https://greatscottgadgets.com/hackrf/

[16] “LimeSDR.” [Online]. Available: https://limemicro.com/products/boards/limesdr/

[17] “Zedboard sdr ii evaluation kit.” [Online]. Available: http://zedboard.org/product/zedboard-sdr-ii-evaluation-kit

[18] R. R. Corp, “Raptor.” [Online]. Available: https://www.rincon.com/shop/board-level/raptor/

[19] “Zynq UltraScale RFSoC.” [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/rfsoc.html

[20] “Xilinx extends its breakthrough zynq ultrascale rfsoc portfolio to full sub-6ghz spectrum support,” Feb 2019. [Online]. Available: https://www.xilinx.com/news/press/2019/xilinx-extends-its-breakthrough-zynq-ultrascale-rfsoc-portfolio-to-full-sub-6ghz-spectrum-sup html

[21] “About gnu radio · gnu radio.” [Online]. Available: https://www.gnuradio.org/about/

[22] J. D. Gaeddert. [Online]. Available: https://liquidsdr.org/doc/background/

[23] Ttsou, “ttsou/turbofec.” [Online]. Available: https://github.com/ttsou/turbofec

[24] A. CASSAGNE, “A fast forward error correction toolbox!” [Online]. Available: https://aff3ct.github.io/

[25] C. Chi and C. Kuo, “Quadratic permutation polynomial interleaver for lte turbo coding,” in *2012 International Conference on Information Security and Intelligent Control*, Aug 2012, pp. 313–316.

[26] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, March 1974.

[27] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal map decoding algorithms operating in the log domain," in *Proceedings IEEE International Conference on Communications ICC '95*, vol. 2, June 1995, pp. 1009–1013 vol.2.

[28] S. Papaharalabos, P. Sweeney, and B. G. Evans, "Constant log-map decoding algorithm for duo-binary turbo codes," *Electronics Letters*, vol. 42, no. 12, pp. 709–710, June 2006.

[29] Jung-Fu Cheng and T. Ottosson, "Linearly approximated log-map algorithms for turbo decoding," in *VTC2000-Spring. 2000 IEEE 51st Vehicular Technology Conference Proceedings (Cat. No.00CH37026)*, vol. 3, May 2000, pp. 2252–2256 vol.3.

[30] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "Asic clouds: Specializing the datacenter," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 178–190.

[31] "Sdaccel development environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[32] "SDSoC Development Environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[33] T. B. Michael Fingeroff, *High-Level Synthesis Blue Book*, 2010. [Online]. Available: http://www.amazon.com/High-Level-Synthesis-Blue-Michael-Fingeroff/dp/1450097243

[34] K. Rupnow, Yun Liang, Yinan Li, and Deming Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*, Oct 2011, pp. 1102–1105.

[35] Zelei Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 218–225.

[36] V. Bhatnagar, G. S. Ouedraogo, M. Gautier, A. Carer, and O. Sentieys, "An fpga software defined radio platform with a high-level synthesis design flow," in *2013 IEEE 77th Vehicular Technology Conference (VTC Spring)*, June 2013, pp. 1–5.

[37] M. Rößler, Hailu Wang, U. Heinkel, N. Engin, and W. Drescher, "Rapid prototyping of a dvb-sh turbo decoder using high-level-synthesis," in *2009 Forum on Specification Design Languages (FDL)*, Sep. 2009, pp. 1–6.

[38] P. Jagatheeswari and M. Rajaram, “Performance comparison of ldpc codes and turbo codes,” vol. 54, pp. 465–472, 06 2011.